RoguePixel Games

Project Horseman

Software Design Document

Name (s):

Jean Hanna

Cailin Jefferson

Mikel Nuila

Mohammed Hussain

Shawn Takhirov

Lab Section: COMP 490/L

Date: 12/11/23

Table of Contents Page

1. INTRODUCTION 2
1.1. Purpose 2
1.2. Scope 2
1.3. Overview 2
1.4. Reference Material 2
1.5. Definitions and Acronyms 2
2. SYSTEM OVERVIEW 3
3. SYSTEM ARCHITECTURE 4
3.1 Architectural Design 4
3.2 Interface Design 4
3.3 Decomposition Description 4
3.4 Design Rationale 5
4. COMPONENT DESIGN/DETAILED DESIGN 6
5. HUMAN INTERFACE DESIGN 11
5.1. Overview of User Interface 11
5.2. Screen Images 12
5.3. Screen Objects and Actions 14
6.0. REQUIREMENTS MATRIX 15

1.0 INTRODUCTION

1.1 Purpose

The purpose of this SDD is to describe the architecture and structure of Project Horseman, our 2D top-down adventure Halloween-themed game. The intended audience is our team and Professor Edmund Dantes.

1.2 Scope

Project Horseman is a video game created in Java using LibGDX framework. The graphics are inspired by Nintendo games like Pokemon and Legend of Zelda therefore it will be a 32-bit pixelated game. It will consist of multiple challenges (quests) and a narrative that follows the main character throughout the storyline.

1.3 Overview

This document will discuss the overall construction of Project Horseman. The SDD explains the class structure and covers the user interface design. Additionally, it will cover project requirements.

1.4 Definitions and Acronyms

2D - two dimensional
NPC - non-player character
TBD - to be determined
WIP - work in progress
UI - user-interface
GUI - graphical user-interface

2.0 SYSTEM OVERVIEW

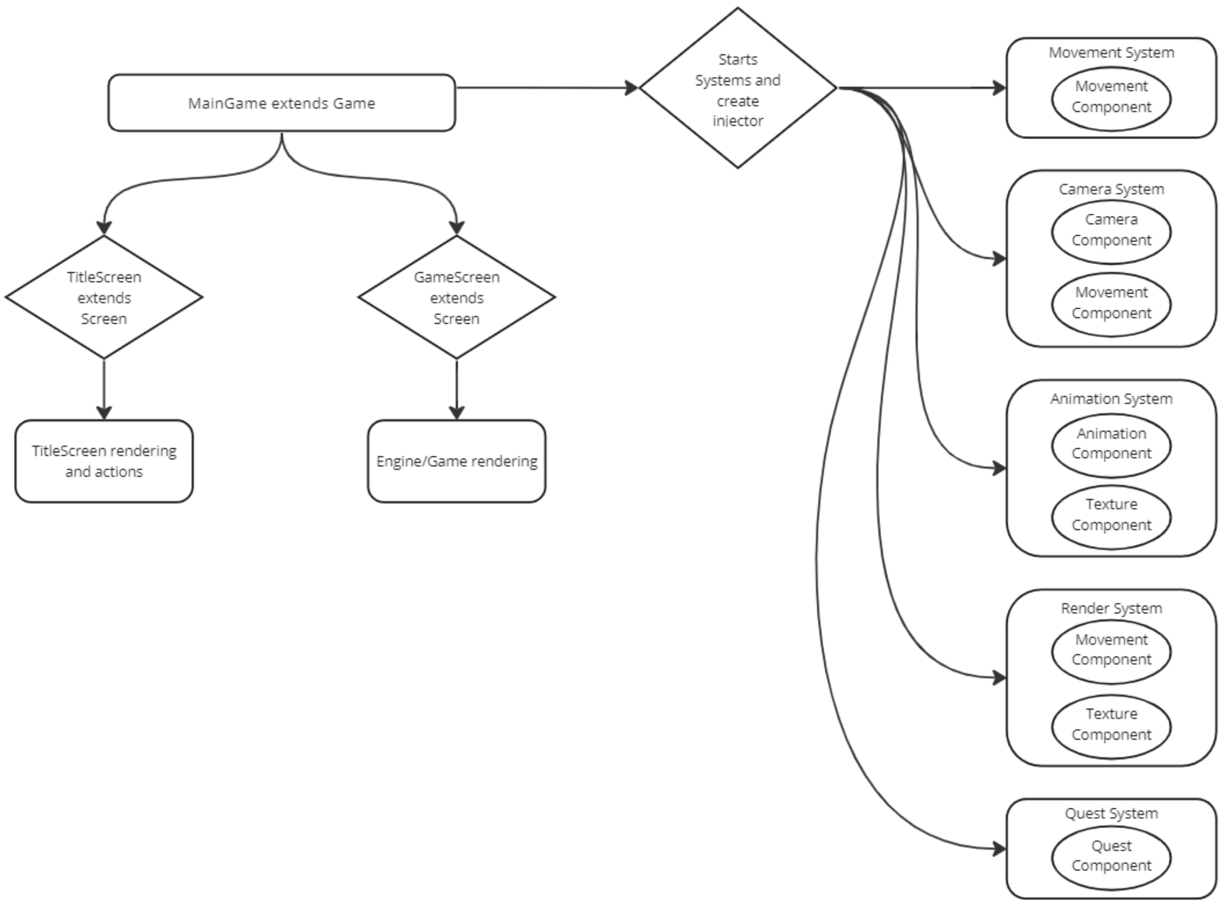
Project Horseman is a top-down view video game that is developed using the LibGDX framework based on Java programming language. The game is a 2D Halloween-themed puzzle adventure that empowers players to embark on a journey of exploration, gather valuable items, and navigate the intricacies of the game's narrative. A carefully crafted storyline serves as a guiding thread, ushering players through diverse levels of the gaming experience. The game is structured with numerous levels and characters, offering players the opportunity to interact with various elements within the gaming environment. It also features a quest system, enabling users to monitor the quest and quest items they've acquired throughout their gameplay. The graphics are inspired by Nintendo games like Pokemon and Legend of Zelda and it will be a 32-bit pixelated game. To achieve this goal our team designed and developed different Java systems and components and the general description of them is as follows:

- PlayerInput System and Player Components to
- Quest System and Quest Components to keep track of the quests throughout each act.
- Camera System and Camera Components which keep track of information about the camera's position, rotation, and scale.
- Movement System and Movement Components which control the motion of entities in the game world.
- Animation System and Animation Components which manage the playback and blending of animations for characters or objects in the game.
- Render System and Render Components which is responsible for rendering graphics to the screen.
- Texture System which handles the loading, management, and application of textures to surfaces in the game.
- Overlay system to display important information

For in-game UI design, the game is designed to have a main menu that consists of two elements play and exit. The UI design also consists of a quest-tracking system.

3.0 SYSTEM ARCHITECTURE

3.1 Architectural Design



3.2 Decomposition Description

In the Java implementation, the CameraSystem serves as the focal point for managing the camera's view and perspective, with associated components of Camera and Movement as well as including things like Viewport and specifying the rendering area. The MovementSystem class orchestrates entity motion, relying on common components such as velocity for speed and direction, MovementState, and collision detection. RenderSystem is responsible for graphics rendering which is called when the game screen opens. The TextureSystem class oversees the management and application of textures, with TextureComponent representing image data. Quest system which takes the Quest component and is called across all three acts as they will have different quests. This structured approach facilitates modular design and reuse of components across the camera, movement, animation, render, and texture systems in Java.

3.3 Design Rationale

1. ECS (Entity-Component-System):
   - Design Philosophy:
     - ECS is a design pattern that separates the concerns of an object into three major components: Entities, Components, and Systems.

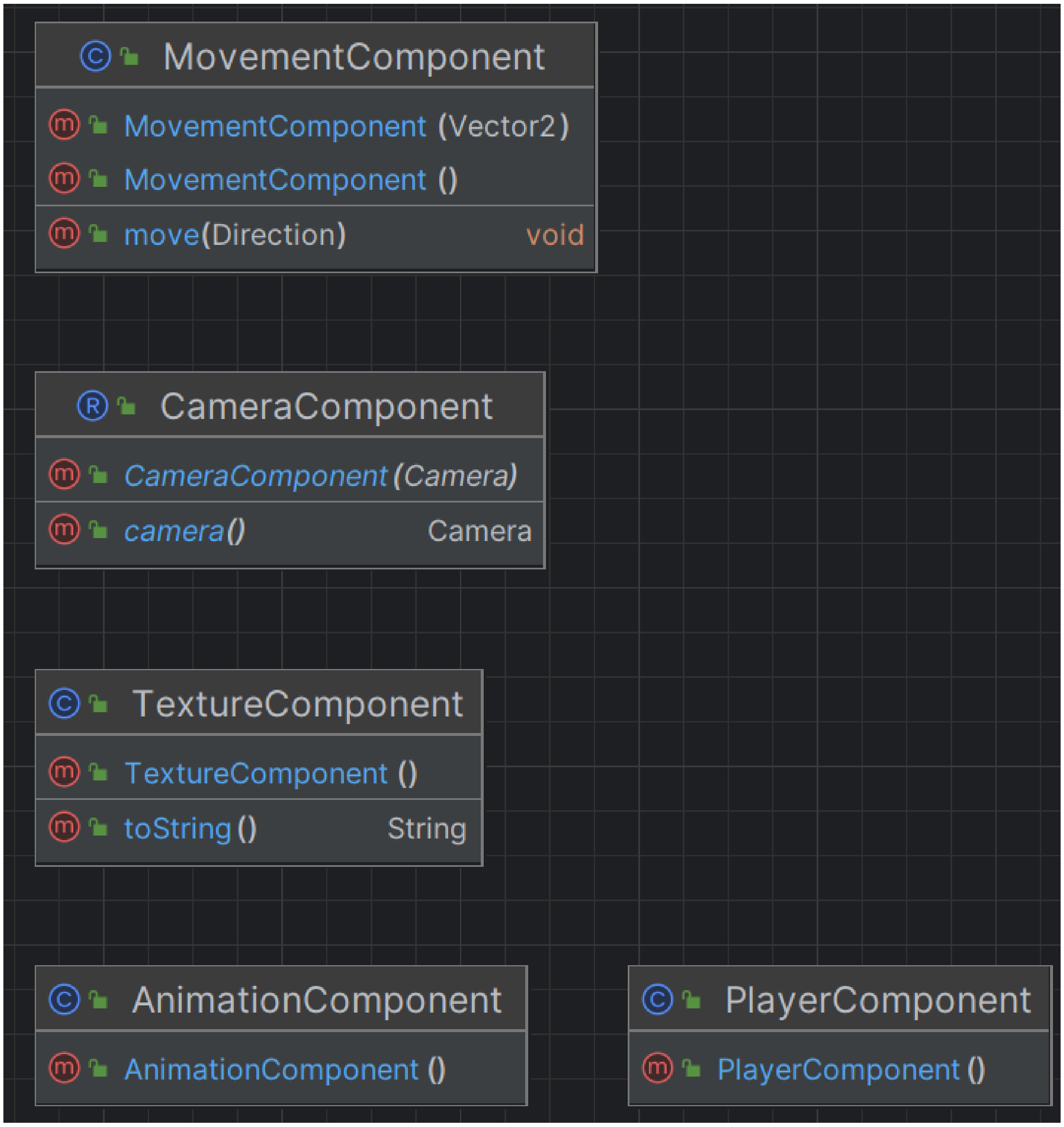
        - Entities are objects in the game world, Components are the data associated with entities, and Systems are responsible for updating components and processing game logic.
    - Benefits for Game Development:
        - Decoupling: ECS promotes a high level of decoupling between different aspects of the game. Entities are a composition of components, and systems operate on these components. This separation makes it easier to manage complexity and facilitates better code organization.
        - Performance: ECS can lead to better performance due to its data-oriented design. Systems can operate on contiguous blocks of memory, improving cache locality and reducing memory access overhead.
        - Scalability: ECS can scale well with large and complex game worlds. It allows for dynamic composition of entities and easy addition or removal of components.

2. Guice Dependency Injection:
    - Design Philosophy:
        - Guice is a framework that implements the dependency injection (DI) pattern. DI is a software design pattern where the dependencies of a class are injected from the outside rather than created within the class itself.
        - Guice facilitates the configuration and injection of dependencies through a combination of annotations and Java code.
    - Benefits for Game Development:
        - Decoupling: Dependency injection promotes loose coupling between components and services. It allows for easier replacement of components or services without modifying the classes that depend on them.
        - Testability: DI makes it easier to replace real implementations with mock or test implementations during unit testing, as dependencies can be easily swapped out.
        - Modularity: Guice supports the creation of modular and reusable components. This is particularly useful in game development, where different systems and features can be developed independently and then integrated.

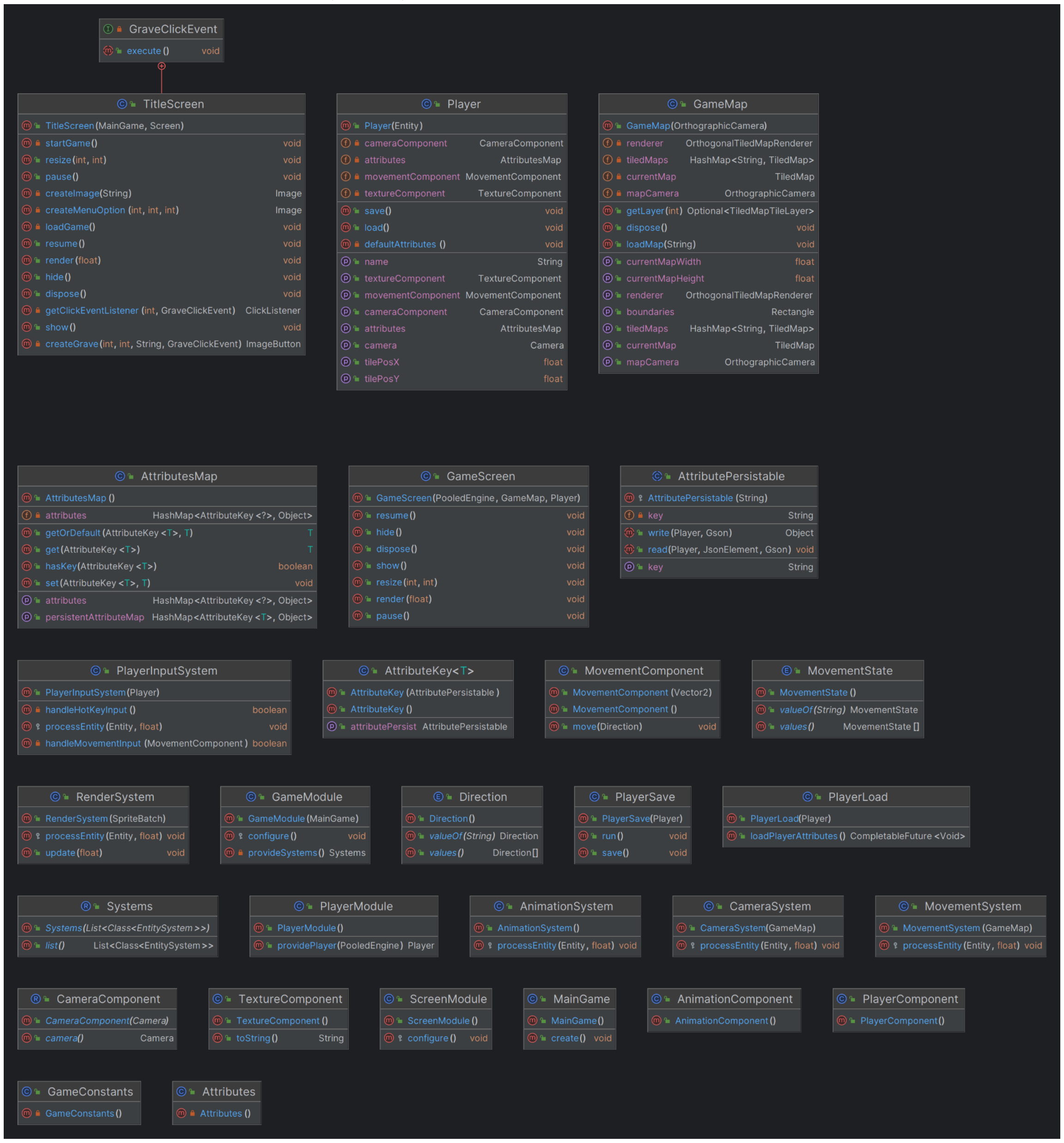
In summary, while ECS focuses on organizing game logic and improving performance through a specific architectural pattern, Guice DI focuses on managing dependencies and promoting modularity and testability. These concepts can be used together in game development to create a robust, scalable, and maintainable codebase. The combination of ECS for structuring game logic and Guice for managing dependencies can result in a flexible and modular game architecture.

4.0 COMPONENT DESIGN/DETAILED DESIGN

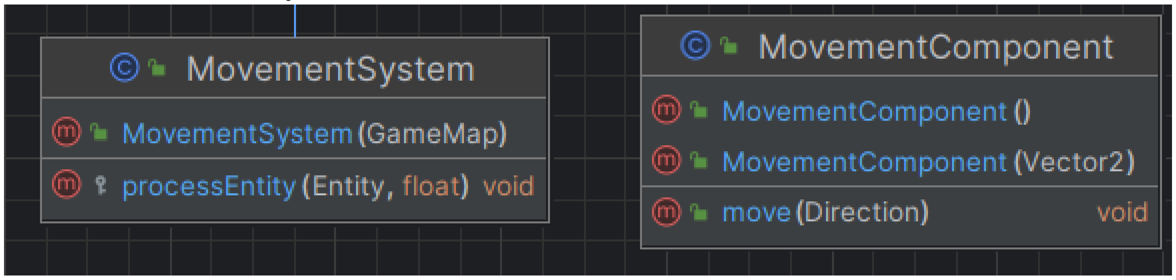
Still designing the components so this is a WIP so far.



4.1 UML DIAGRAM (Java)

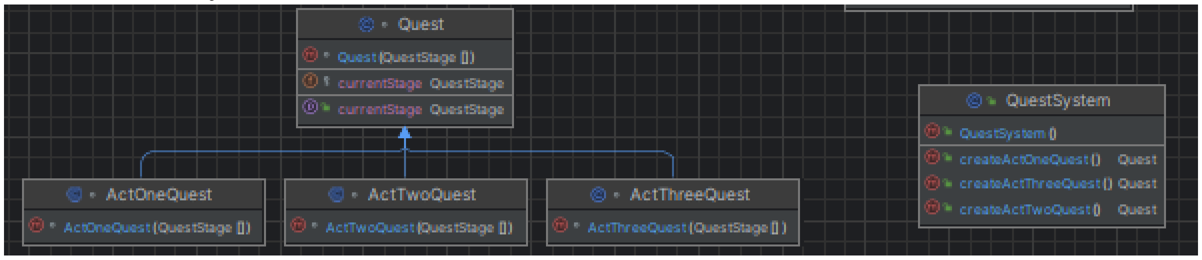

GraveClickEvent
execute() void
TitleScreen
TitleScreen(MainGame, Screen)
startGame() void
resize(int, int) void
pause() void
createImage(String) Image
createMenuOption(int, int, int) Image
loadGame() void
resume() void
render(float) void
hide() void
dispose() void
getClickEventListener(int, GraveClickEvent) ClickListener
show() void
createGrave(int, int, String, GraveClickEvent) ImageButton
Player
Player(Entity)
cameraComponent CameraComponent
attributes AttributesMap
movementComponent MovementComponent
textureComponent TextureComponent
save() void
load() void
defaultAttributes() void
name String
textureComponent TextureComponent
movementComponent MovementComponent
cameraComponent CameraComponent
attributes AttributesMap
camera Camera
tilePosX float
tilePosY float
GameMap
GameMap(OrthographicCamera)
renderer OrthogonalTiledMapRenderer
tiledMaps HashMap<String, TiledMap>
currentMap TiledMap
mapCamera OrthographicCamera
getLayer(int) Optional<TiledMapTileLayer>
dispose() void
loadMap(String) void
currentMapWidth float
currentMapHeight float
renderer OrthogonalTiledMapRenderer
boundaries Rectangle
tiledMaps HashMap<String, TiledMap>
currentMap TiledMap
mapCamera OrthographicCamera
AttributesMap
AttributesMap()
attributes HashMap<AttributeKey<?>, Object>
getOrDefault(AttributeKey<T>, T) T
get(AttributeKey<T>) T
hasKey(AttributeKey<T>) boolean
set(AttributeKey<T>, T) void
attributes HashMap<AttributeKey<?>, Object>
persistentAttributeMap HashMap<AttributeKey<T>, Object>
GameScreen
GameScreen(PooledEngine, GameMap, Player)
resume() void
hide() void
dispose() void
show() void
resize(int, int) void
render(float) void
pause() void
AttributePersistable
AttributePersistable(String)
key String
write(Player, Gson) Object
read(Player, JsonElement, Gson) void
key String
PlayerInputSystem
PlayerInputSystem(Player)
handleHotKeyInput() boolean
processEntity(Entity, float) void
handleMovementInput(MovementComponent) boolean
AttributeKey<T>
AttributeKey(AttributePersistable)
AttributeKey()
attributePersist AttributePersistable
MovementComponent
MovementComponent(Vector2)
MovementComponent()
move(Direction) void
MovementState
MovementState()
valueOf(String) MovementState
values() MovementState[]
RenderSystem
RenderSystem(SpriteBatch)
processEntity(Entity, float) void
update(float) void
GameModule
GameModule(MainGame)
configure() void
provideSystems() Systems
Direction
Direction()
valueOf(String) Direction
values() Direction[]
PlayerSave
PlayerSave(Player)
run() void
save() void
PlayerLoad
PlayerLoad(Player)
loadPlayerAttributes() CompletableFuture<Void>
Systems
Systems(List<Class<EntitySystem>>)
list() List<Class<EntitySystem>>
PlayerModule
PlayerModule()
providePlayer(PooledEngine) Player
AnimationSystem
AnimationSystem()
processEntity(Entity, float) void
CameraSystem
CameraSystem(GameMap)
processEntity(Entity, float) void
MovementSystem
MovementSystem(GameMap)
processEntity(Entity, float) void
CameraComponent
CameraComponent(Camera)
camera() Camera
TextureComponent
TextureComponent()
toString() String
ScreenModule
ScreenModule()
configure() void
MainGame
MainGame()
create() void
AnimationComponent
AnimationComponent()
PlayerComponent
PlayerComponent()
GameConstants
GameConstants()
Attributes
Attributes()

4.1.1 Movement System



Responsible for processing entities with a MovementComponent. It utilizes a ComponentMapper to access relevant components and checks for entity movement based on the specified direction and velocity. The system considers collisions with a TiledMapTileLayer in the game map and updates the entity's position accordingly. The MovementComponent stores information about an entity's position (pos), direction (dir), movement state (state), velocity (velocity), and a flag indicating whether movement is disabled (disabled). The move method in MovementComponent initiates movement in a specified direction. Overall, these classes facilitate the handling of entity movement within the game.

4.1.2 Quest System (WIP)


Quest
Quest (QuestStage [])
currentStage QuestStage
currentStage QuestStage
ActOneQuest
ActOneQuest (QuestStage [])
ActTwoQuest
ActTwoQuest (QuestStage [])
ActThreeQuest
ActThreeQuest (QuestStage [])
QuestSystem
QuestSystem ()
createActOneQuest () Quest
createActThreeQuest () Quest
createActTwoQuest () Quest

QuestSystem will keep track of the quests throughout the stages and Quest will keep track of current quests and the description of the quest so it can be displayed using the interface.

4.1.3 Weather System (WIP)

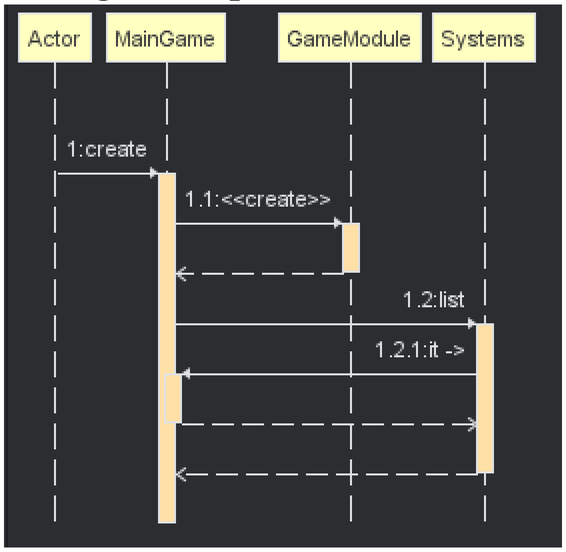
WeatherSystem will shuffle through the weather effects depending on the area of the map.

4.1.4 NPC System (WIP)

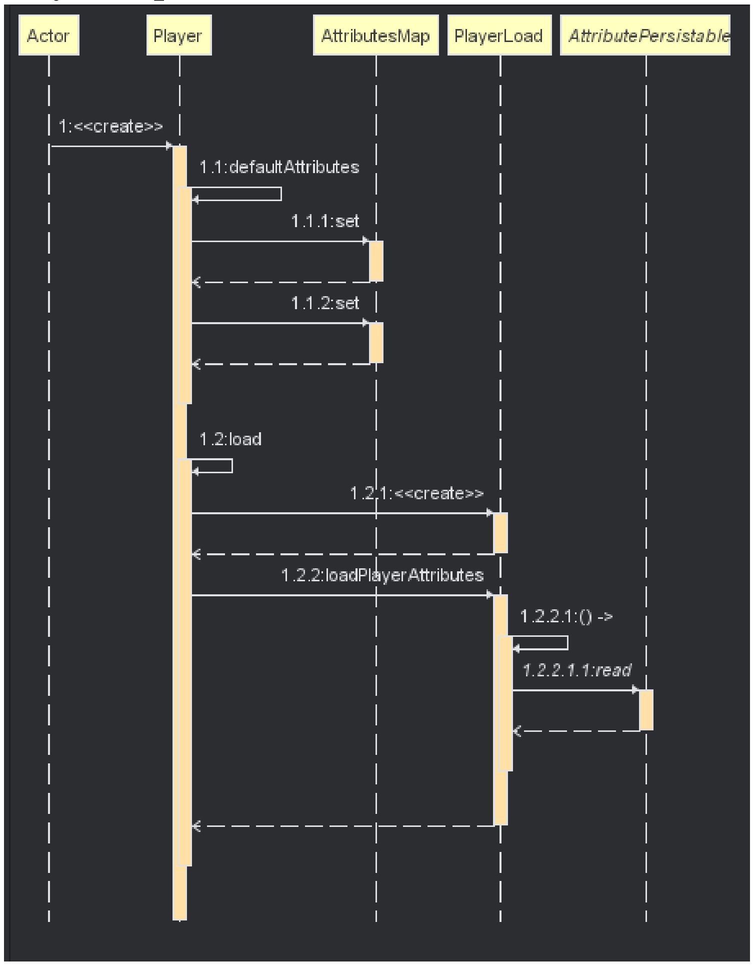
We have not begun to design this but it will consist of NPC involvement in quests as well as the NPC path which is where they will walk across the map.

4.2 SEQUENCE DIAGRAMS

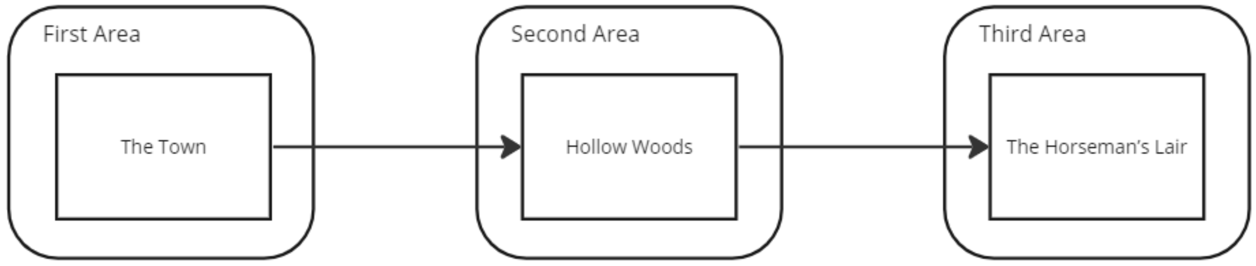
Main game sequence



Player sequence



4.3 LEVEL DESIGN



4.3.1 The Town

The Town is the first area that the player will explore in the game. During this section, the player will play as a child who lives in the town and will be in search of clues to help eliminate the evil that plagues his home.

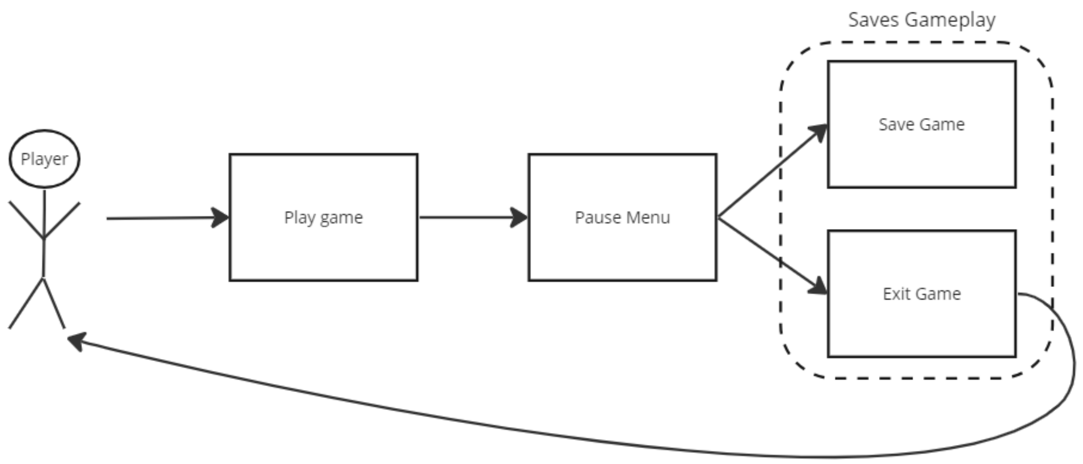
4.3.2 Hollow Woods

The Hollow Woods is the second area that the player will explore in the game. During this section, the player will continue as the child as he searches for clues to the Horsemen's lair and finds a way out of the cursed woods.

4.3.3 The Horseman's Lair

The Horseman's Lair is the third and last area that the player will explore in the game. During this section, the child will have to solve puzzles to unlock the horseman's study and end his terror

4.4 SAVING AND LOADING



4.4.1 Saving and loading

The game will save itself when you close the game. Additionally, the player can save the game throughout the play using the save button on the pause screen and that save file goes into your user's home directory. After exiting the game, the player can start up the game again and will be returned to their previous saved game.

5.0 HUMAN INTERFACE DESIGN

5.1 Overview of User Interface

From the user's perspective, the following UI subsystems facilitate the game experience

1. Main Menu
2. Pause Menu

Main Menu (fig.1):
Upon opening the game, the user is presented with a main menu.

1. Play button
2. Quit button

Pause Menu (fig.2):
During gameplay, the user can hit the escape key to bring up the pause menu. This pauses the game and the following options are shown.

1. Exit to Main Menu: exits to the main menu
2. Exit to Desktop: closes the game and exits to the desktop
3. Continue

During gameplay, the player can hit the Q key to bring up the quest log or press the quest button on the GUI. (fig.4) This pauses the game and brings up the quest panel. (fig. 3)

1. Quest Name
2. Quest Description
3. Goals to move the quest forward
4. List of active quests
5. Task description: the description of the current item. Will turn green if the task is complete otherwise it is red.
6. Task Widget: houses the task icon

In-game GUI (fig. 4):
During gameplay, two elements are shown on screen as pictures.

1. Quest Menu: opens quest menu
2. Pause Menu: opens the pause menu

5.2 Screen Images

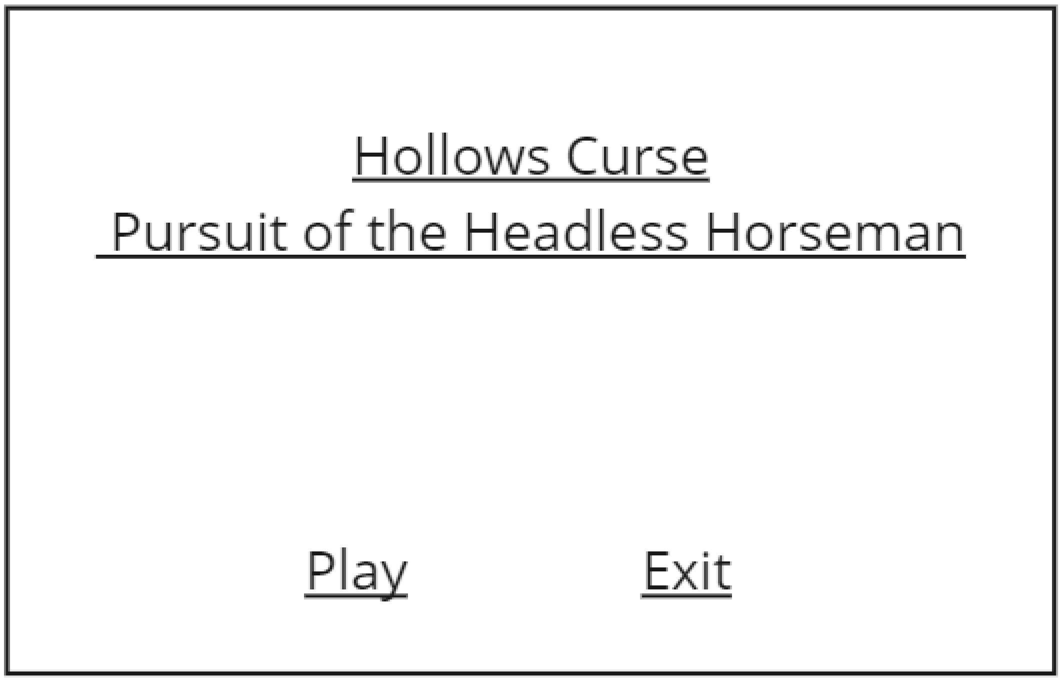


Figure 1. Main Menu (WIP)

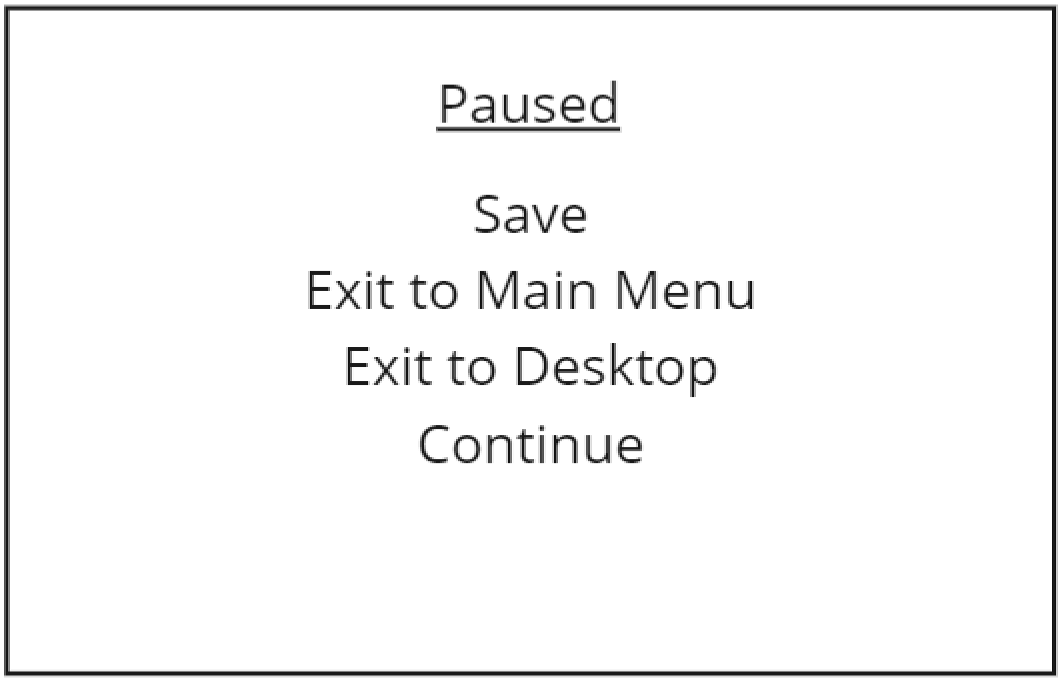


Figure 2. Pause Menu (WIP)

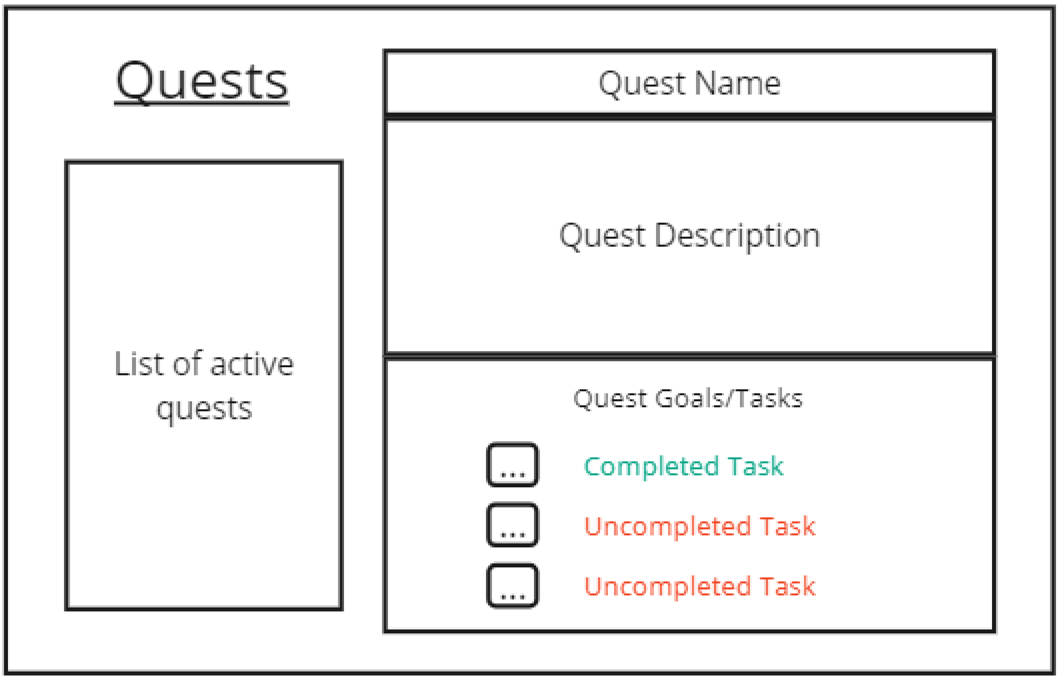


Figure 3. Quest System UI (WIP)

Game Play Occurring
Q
Esc

Figure 4. GUI for UI screens (WIP)

5.3 Screen Objects and Actions

TBD in 491

6.0 REQUIREMENTS MATRIX

SRS Req. ID	Paragraph Title	Component Num.	Component Name
FUNC_SRS_001	The game shall provide the ability to move the character up/down/left/right	4.1.1	Movement System/ Component
FUNC_SRS_002	The game shall have a quest system.	4.1.2	Quest System/ Component
FUNC_SRS_003	The game shall support single-player mode.	4.3.1, 4.3.2, 4.3.3	“The Town, Hollow Woods, The Horseman’s Lair”
FUNC_SRS_004	Project Horseman will incorporate a save and load system, allowing players to continue from their last checkpoint.	4.4	Save and load system
FUNC_SRS_005	The game's design shall include interactive NPCs (Non-Playable Characters) that provide quests, and quest items.	4.1.2, 4.1.4	Quest System/ Component, NPC Component
FUNC_SRS_007	The game will have an adaptive background music and sound effects that respond to in-game events and player actions.	4.1.3	Weather System
FUNC_SRS_008	Project Horseman shall feature a distinct map area. It will consist of puzzles and narratives.	4.3.1, 4.3.2, 4.3.3	“The Town, Hollow Woods, The Horseman’s Lair”