Whimsical Wizardry / Game Modes and Game States

Module Design Document

**Table of Contents**

Contents

[Whimsical Wizardry / Game Modes and Game States 1](#_Toc152763715)

[Module Design Document 1](#_Toc152763716)

[Contents 1](#_Toc152763717)

[1 Introduction 4](#_Toc152763718)

[2 Design Goals 4](#_Toc152763719)

[3 System Behaviour 4](#_Toc152763720)

[4 Logical View 5](#_Toc152763721)

[4.1 High-Level Design 5](#_Toc152763722)

[4.2 Mid-Level Design of the Module 6](#_Toc152763723)

[4.3 Detailed Class Design of the Module 7](#_Toc152763724)

[5 Process View 8](#_Toc152763725)

[6 Use Case View 14](#_Toc152763726)

Change History

**Version:** 0.1

**Modifier:** Izach Carberry

**Date:**  25 / 10 / 2023

**Description of Change:** Module Design Document started.

**Version:** 0.2

**Modifier:** Izach Carberry

**Date:**  02 / 11 / 2023

**Description of Change:** Added UML.

**Version:** 0.3

**Modifier:** Izach Carberry

**Date:**  07 / 11 / 2023

**Description of Change:** Added High-Level Diagram.

**Version:** 0.4

**Modifier:** Izach Carberry

**Date:**  08 / 11 / 2023

**Description of Change:** Added Mid-Level Diagram and Process View.

**Version:** 1.0

**Modifier:** Izach Carberry

**Date:**  09 / 11 / 2023

**Description of Change:** Finished First draft.

**Version:** 1.5

**Modifier:** Derek Fallows

**Date:**  15 / 11 / 2023

**Description of Change:** Updated high-level diagram.

**Version:** 1.6

**Modifier:** Izach Carberry

**Date:**  30 / 11 / 2023

**Description of Change:** Update detailed view, Mid-Level view, and use case view..

**Version:** 1.7

**Modifier:** Izach Carberry

**Date:**  1 / 12 / 2023

**Description of Change:** Changed Headings and added table of contents.

**Version:** 1.8

**Modifier:** Izach Carberry & Sydney Allen

**Date:**  1 / 12 / 2023

**Description of Change:** Added borders and numbers

**Version:** 2.0

**Modifier:** Izach Carberry & Sydney Allen

**Date:**  5 / 12 / 2023

**Description of Change:** Changed formatting.

**Version:** 2.1

**Modifier:** Izach Carberry

**Date:**  6 / 12 / 2023

**Description of Change:** Changed formatting to the base template.

**Version:** 2.2

**Modifier:** Izach Carberry

**Date:**  7 / 12 / 2023

**Description of Change:** Changed use case view format.

**Version:** 2.3

**Modifier:** Izach Carberry

**Date:**  8 / 12 / 2023

**Description of Change:** Error checked for grammar and spelling.

1 Introduction

This document describes the architecture and design for the Game Mode &States module application being developed for whimsical wizardry. The Game Mode and States module is meant to be reusable and mouldable depending on game design specifications. It should be easy to take the base game mode class and reuse it for different whimsical wizardry game modes.

The purpose of this document is to describe the architecture and design of the Game Mode and States module application in a way that addresses the interests and concerns of all major stakeholders:

* Developers – they want an architecture that will minimize complexity and development effort.
* Project Manager – the project manager is responsible for assigning tasks and coordinating development work. He or she wants an architecture that divides the system into components of roughly equal size and complexity that can be developed simultaneously with minimal dependencies. For this to happen, the modules need well-defined interfaces. Also, because most individuals specialize in a particular skill or technology, modules should be designed around specific expertise. For example, all UI logic might be encapsulated in one module. Another might have all-game logic.
* Maintenance Programmers – they want assurance that the system will be easy to evolve and maintain in the future.

2 Design Goals

The design priorities for the Modes and States system are:

* The design should be flexible and modular to use for easy use in designing game modes.
* The design should be accessible to the user for making their Match preferences.
* The design should be optimized so that other Developers can make a game mode or state easily.

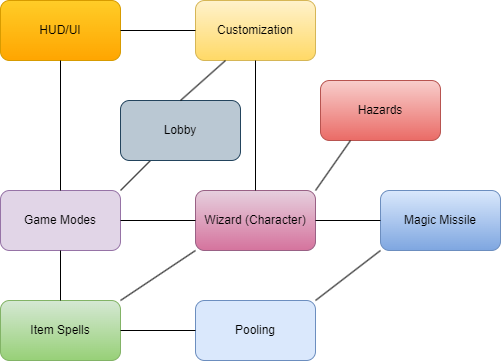
3 System Behaviour

The Modes and States module is built from the AGameMode and AGameState classes. This allows the modes and states to define clear points in the match such as: before the match begins, after it begins, before the match ends, and after the match ends. This should allow Developers to define and code what should happen during these specific parts of the match, as well as when it should enter a new State. The behavior of match states depends on the child actors of the Modes and State module.

4 Logical View

4.1 High-Level Design

Whimsical Wizardry has nine major modules.



**Customization**  
Handles configuration of the wizard player characters, allowing for individual differences such as hats, different staffs, etc.

**Game modes**  
Handles the broad rules and situations particular to each game mode.

**Hazards**  
Represents various types of obstacles in the map that knock back and damage the player or otherwise attack them.

**HUD/UI**System of menus and HUD elements that aids in selection as well as conveying information to the player.

**Item Spells**  
Spells that a player can cast after picking them up from spell pickups during gameplay.

**Lobby**Specific game mode where players can select different maps and settings before the game starts.

**Magic Missile**Basic projectile attack players can make without any spell pickups.

**Pooling**  
Handles actors from various other modules to help avoid spawning actors on runtime.

**Wizard**  
The player character and its components

4.2 Mid-Level Design of the Module

A diagram of a computer program

Description automatically generated with medium confidence

Classes involved.

* ALobbyGM : ALobbyGM is where players start a match. When a match ends each game mode implementation will seamlessly travel back to the lobby.
* AWhimsicalWizardryGameModeBase : The base Game Mode class each playable game mode inherits from. This includes basic functionality that child actors build upon and use.
* AWimsicalWizardryGameStateBase : The base Game State class each playable game state inherits from. This includes basic functionality that child actors build upon and use.
* EGameModeType : An enum that holds the type of game mode it is. There are three enumerators currently.
* ALastWizardStanding : This game mode handles the rules of last wizard standing mode. Such as when a player should lose a life and when a player should lose.
* ALastWizardStandingState : This game state handles the scores and scoring of last wizard standing mode. This class's main job is to tell the game mode when and which player should lose a life.
* APonderTheOrb : This game mode handles the rules of ponder the orb mode. Its responsibility is to tell orb spawners when to spawn an orb, respawn the orb, and update the player's score.
* APonderTheOrbState : This game state handles the score and when orbs should respawn. Whenever a wizard is knocked off a stage this class asks game mode to respawn the orb.
* UPickupSpawner : This class is responsible for spawning all pickups in the level. This includes but is not limited to the orb.
* AOrb : This class is the main focus of ponder the orb mode. This class’s responsibility is to, when held by a wizard, ask game mode to add to that wizard's score. This class also asks game mode to apply a score bonus if the wizard is pondering a held orb.

4.3 Detailed Class Design of the Module

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

5 Process View

The base Modes and States classes have base functionality that holds and gives out information to the players. It is also responsible for things such as switching between levels and counting down the start of a match. The child modes and states build upon the base by adding win conditions and functionality for calculating player scores.

Give Lives and Index

A screenshot of a computer program

Description automatically generated

1. At the start of the process all wizards ask for an index from the game state class at the beginning of play, by calling GiveIndex and passing in itself. The index is later used for keeping track of scoring and other functionality in the game.
2. When GiveIndex is called the game state sets the passed-in wizards index to the next available index.
3. We then increment the next index so the next wizard to call the give index function is given a different index. Each person playing will appear as the first player on their own systems.
4. After receiving their index, the wizard will ask for lives from the game state class, by calling GiveLives and passing in itself.
   1. In LastWizardStandingState, the lives are used for how many times the player can be knocked off the stage before they stop respawning.
   2. In PonderTheOrbState, the lives variable is not used. Players are continuously respawning until one player has won the Match.
   3. More Game Modes are planned where lives are used.
5. When GiveLives is called the game state sets the passed-in wizards lives to a child specified variable.
   1. LastWizardStandingState will set wizard lives to its max lives variable. Max lives are easily changeable in code or blueprint (See Last Wizard Standing Use Case View)
   2. PonderTheOrbState will set wizard lives to zero. PonderTheOrbState dose not need to keep track of lives and will allow the player to continue respawning.
   3. More Game Modes are planned where lives are used.

Winning with Modes & States

Last Wizard Standing

A screenshot of a computer program

Description automatically generated

1. A wizard must be knocked off the level for the process to begin. When this happens, that wizard knock off component calls OnKnockOff on ALastwizardStandingState passing in its owning wizard.
2. ALastWizardStandingState takes one life from the wizard.
3. If the wizard’s lives are more than zero set wizard lives to what we calculated it to be in step two. Continue playing from here.
4. If the wizard’s lives are less than or equal to zero, we set wizard’s lives to zero. This wizard is now out of the game.
5. ALastWizardStandingState then call SetShouldRespawn on wizard passing in false. Wizard can no longer respawn and is now just respawning.
6. ALastWizardStanding call OnePlayerLeft on ALastWizardStanding passing in a TArray of wizards. This function will test how many wizards are still in the match.
7. If the number of wizards left is one the match is over. ALastWizardStanding calls SetMatchState passing in MatchState::WaitingPostMatch to end the process.

Ponder the Orb

A screenshot of a computer program

Description automatically generated

1. The process begins with AOrb class calling AddScore passing in the wizard currently holding it. This is done every tick for as long as a wizard is holding the orb. If no wizard is holding it this function is not called.
2. If the game isn’t over yet and if a wizard is holding an orb, we continue. APonderTheOrb Grabs the index of the wizard holding the orb. We will add score to ScoreBoard at that wizard’s index.
3. When holding an orb, a wizard can ponder the orb. If the wizard is pondering the orb increment, their score by base score and multiplied by the ponder bonus. Add that wizard's score to the scores array at the wizard's index.
4. If the wizard is not pondering the orb increment their score by the base score. Add that wizard's score to the scores array at the wizard's index.
5. If a wizard’s score is more than or equal to the goal that wizard has won. Set that wizard’s score to the goal value.
6. APonderTheOrb calls SetMatchState passing in MatchState::WaitingPostMatch to end the process.

Lobby Travel

A diagram of a game

Description automatically generated

1. At the start of the process a player has achieved a game mode’s win condition. When this happens that game mode will call SetMatchState passing in MatchState::WaitingPostMatch. This tells the base game mode that the match has ended.
   1. In LastWizardStanding this is called when only one player is left with more than zero lives.
   2. In PonderTheOrb this is called when one player reaches the goal ponder time variable.
2. When the match has ended HandleMatchHasEnded is automatically called. This function creates a timer to call HandleLeavingMap.
3. If the timer has ended, HandleLeavingMap is called. Finds the information needed for the server to jump to a new level. In the case of Modes and States we are always traveling back to the lobby level when the match ends.
4. The process ends with all players traveling to the lobby level.

Count Down

A screenshot of a computer program

Description automatically generated

\*\* This process involves function from the LobbyGM class. For more information on LobbyGM class please refer to the Lobby module. \*\*

1. The process begins when all players are ready to start a match. ALobbyGM is calling StartCountdown. This function is how we tell the other actors important to this process to start counting down the start of a match. It may pass in an integer call time the process will begin counting down from. By default, time is five.
2. StartCountdown sets a looping timer for UpdateCountdown to be called. Then it calls AWizardController StartCountdown
3. StartCountdown calls the GameHuds StartCountdown function. AGameHud’s StartCountdown function turns on the countdown widget’s visibility.
4. If all players are still ready, the process will continue with calling UpdateCountdown. This function decreases time integer by one. If at least one player is no longer ready skip to step eleven calling ResetCountdown.
5. If the time is equal to 0 Call ResetCountDown. ResetCountDown resets time to whatever value it was before any updates happened. Then we stop calling UpdateCountdown. We then call StopCountdown on AWimsicalWizardryGameStateBase.
6. StopCountdown calls StopCountdowon on AWizardController.
7. StopCountdown calls StopCountdowon on AGameHud. AGameHud’s StopCountdowon function turns off the countdown widget’s visibility.
8. ToggleLoadScreen calls ToggleLoadScreen on AWizardController.
9. ToggleLoadScreen calls ToggleLoadScren on AGameHud. AGameHud’s ToggleLoadScren function inverses the loadScreen widget’s visibility. Turning it off if it was on, and on if it was off.
10. We are then ready to start the game. This is where the process ends.
11. ResetCountDown resets time to whatever value it was before any updates happened. Then we stop calling UpdateCountdown. We then call StopCountdown on AWimsicalWizardryGameStateBase.
12. StopCountdown calls StopCountdowon on AWizardController.
13. StopCountdown calls StopCountdowon on AGameHud. AGameHud’s StopCountdowon function turns off the countdown widget’s visibility.

6 Use Case View

Modifying Win Conditions

Last Wizard Standing

**C++ Code**

Location: LastWizardStandingState.h



**Editor**



Last Wizard Standing only has one Variable related to the win condition. It can be edited either within code or inside the unreal editor.

* m\_MaxLives: the maximum amount of lives the player can have.

Ponder The Orb

**C++ Code**

Location: PonderTheOrb.h

A black background with white text

Description automatically generated

A black and grey background

Description automatically generated**Editor**

Ponder The Orb has several variables related to win condition. They can be edited wither within code or inside the unreal editor.

* GoalPonderTime: The amount of time you need to ponder in seconds in order to win.
* ScoreRate: The rate the score goes up every tick while holding the orb.
* PonderBonus: The bonus multiplying the ScoreRate while pondering.