Project: Abyss - DiverMovementComponent

Architecture/Design Document

**Table of Contents**

1 Introduction… 3

2 Design Goals… 3

3 System Behavior… 4

4 Logical View... 4

4.1 High-Level Design (Architecture)… 5-6

4.2 Mid-Level Design… 7

4.3 Detailed Class Design… 8

5 Process Views... 9-10

6 Use Case Views... 11-13

Version: 0.1

Modifier: Maxime Paradis

Date: 6 Nov 23

Changes: Created and start of base doc structure (filled out sections 2 - 5.1).

Version: 0.2

Modifier: Maxime Paradis

Date: 7 Nov 23

Changes: Finished most of the diagrams (mostly filled out sections 5.1 - 6).

Version: 0.3

Modifier: Maxime Paradis

Date: 9 Nov 23

Changes: Finished and updated the diagrams.

Version: 0.4

Modifier: Maxime Paradis

Date: 06 Dec 23

Changes: Quick touch ups

Version: 0.5

Modifier: Maxime Paradis

Date: 06 Dec 23

Changes: Changed to doxs, updated format and other touch ups (TODO: Update diagrams)

Version: 0.5

Modifier: Maxime Paradis

Date: 07 Dec 23

Changes: Changed the diagrams

# **1 Introduction**

This document describes the architecture and design for Project: Abyss, a game being developed and designed by Astral Waters Interactive. Project: Abyss is an exploration horror game leaning into the inherently terrifying environment of deep seas adding heavy co-op elements and alien environments to enhance the terrifying setting.

The purpose of this document is to describe the architecture and design of the DiverMovementComponent in a way that addresses the interests and concerns of all major stakeholders. For this application the major stakeholders are:

● 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 on into the future.

# **2 Design Goals**

The design priorities for the DiverMovementComponent are:

● The component should be easily modifiable and accessible by designers.

● The component should allow the player to have smooth and “Good Feeling” movement through the game world.

● Should be balanced and lend itself to the exploration horror gerne

● Results should be intuitive and user friendly.

**3 System Behavior**

The DiverMovementComponent inherits from UCharacterMovementComponent. This allows the component by default to have proper movement systems, water, and land. The Component should allow the player to smoothly transition in and out of water, while feeling natural, smooth, and intuitive. The component will pair up with the character actor them selves and the AWaterTransitionVolume Actor to accomplish this behavior.

# **4 Logical View**

The logical view describes the main functional components of the system. This includes modules, the static relationships between modules, and their dynamic patterns of interaction.

In this section the modules of the system are first expressed in terms of high-level components (architecture) and progressively refined into more detailed components and eventually classes with specific attributes and operations.

## 4.1 High-Level Design (Architecture of the Entire system)

The high-level view consists of…

* **Player System** is the central component that manages the player's actions, interactions, and progression throughout the game. It includes the player character, input handling. Key responsibilities involve controlling the player's movements, handling inputs, and triggering movement states such as swimming.
* **Photography System** is responsible for implementing the mechanics related to underwater photography. It includes the Photography Camera and associated components. This system allows players to take pictures during dives, manage a limited film capacity, store pictures in a gallery, and upload relevant ones to the Collection journal, contributing to the player's knowledge about underwater creatures.
* **Creature System** manages the behavior, interactions, and characteristics of underwater creatures. This system includes various types of fish, their AI, and the logic governing their movement, appearance, and responses to the player's actions. The system ensures a dynamic and realistic underwater ecosystem, providing challenges and opportunities for the player.
* **AI System** governs the artificial intelligence of both hostile and non-hostile entities in the game. It includes the behaviors and decision-making processes of underwater creatures, as well as any AI-driven challenges or enemies the player may encounter during dives.
* **Missions System** oversees the organization of missions into distinct categories, monitors the progression of objectives, and governs the allocation of rewards and unlocks. Accessible through the PC Terminal, players can seamlessly navigate through available, active, and completed missions. The PC Terminal serves as the central hub for mission management in the form of an email system, allowing players to accept new missions and submit completed ones, providing a comprehensive interface for tracking and advancing in the game's narrative.
* **Health Component System** manages the player's health. It includes components that handle damage, healing, and any other health-related mechanics. This system ensures that the player's survival is a key consideration during dives and encounters with underwater creatures.
* **Stress Component System** adds a psychological horror element to the game. It manages the player's stress levels based on in-game events, environmental factors, and encounters. High stress may affect performance or trigger adverse effects, adding an immersive layer to the overall experience.
* **Boat System** encompasses the functionality related to the player's boat. This includes navigation between dive locations, boat inventory, and serving as a safe zone between dives. The boat system also facilitates equipment upgrades, PC Terminal interaction, and overall progression.
* **DiveCage System** is used as a later game upgrade that can be used to speed up travel time to get to deeper locations.
* **Equipment System** is the main item system in the game. Players will be able to buy and use equipment from the shop, including a camera, deployable light sources, and more. Some equipment is consumable, and some is permanent.
* **Divebot System** encompasses the functionality to the controllable Divebot available in the game. The Divebot will be possessed on death as a spectator mechanic, and can be possessed on command to allow remote control for scouting purposes. This will include a custom movement component to assist in executing some of these tasks.
* **Upgrades System** enables the enhancement and customization of the player's equipment, boat, and dive-related tools. Players can earn or purchase upgrades to improve their camera, dive cage, and other components, providing a sense of progression and empowerment.
* **Shop System** allows players to buy in-game equipment and upgrades. It provides a marketplace where players can spend earned resources to acquire new equipment, upgrade existing gear, or replenish consumables.
* **PC System** refers to the personal computer within the game. It serves as the hub for managing various aspects, such as reviewing your research journal, viewing photographs, accessing the in-game shop, and planning future dives. The PC system contributes to the player's overall agency and strategic decision-making.
* **Inventory System** refers to the inventory component that is attached to the player and storage chest. The player will be able to swap equipment to and from the storage chest.

## A black screen with white text Description automatically generated4.2 Mid-Level Design of the DiverMovementComponent

## The DiverMovementComponent interacts with a wide range of different component and Actors, here is a description of their interactions.

* **UCharacterMovement –** Inherited by **UDiverMovementComponent** to allow it to access some useful functionality such as, AddMovementInput and its different states**.**
* **ADiverController –** Passes Inputs through **ADiverCharacter** to **UDiverMovementComponent**. These inputs are stored and used to apply specific movements depending on its movement state.
* **AWaterTransitionVolume –** Changes movement state of **UDiverMovementComponent** through ADiverCharacter. Will switch from walking to swimming using the base functionality of UCharacterMovementComponent.

## 4.3 Detailed Class Design of the DiverMovementComponent

# A screenshot of a computer program Description automatically generated**5 Process View of the DiverMovementComponent Module**

**PlayerMovementComponent Base Functionality:**

A screenshot of a computer

Description automatically generatedReceive Input Sequence:

1. ADiverController Receives input from gamepad or keyboard.
2. ADiverController sends the received input action value to UDiverMovementComponent (Accessing it through ADiverCharacter) sending them to method “HandleInput()” where the values are put into a FVector.
3. UDiverMovementComponent then calls move on itself.
4. In move the newly made vector is stored. And **if** the player isn't swimming the input vector is passed to ADiverCharacter via “AddMovementInput()” and **loop back to the start**.
5. The component then waits for “Tick()” where the swim speed is adjusted to the movement direction
6. Then passed to ADiverCharacter via “AddMovementInput()”
7. In UDiverCharacter the movement is applied in “Super::Tick()” functions. Part of BaseClass “ACharacter”

Transitioning States Sequence:

A computer screen shot of a black screen

Description automatically generated

1. The player enters the collision box of AWaterTransitionVolume and triggers “OnBeginOverlap()”
2. In AWaterTransitionVolume it checks if the actor entered is inheriting from IStateTransitionInterface, if so Calls “TransitionMovementStates()” (A Method inherited from the interface) on the overlapping actor, in our case the player.
3. The bool called bIsSwimming is toggled.
4. If bIsSwimming is true call the LaunchCharacter(), a function inherited from ACharacter
5. If bIsSwimming is false call TransitionMovementState on UDiverMovementComponent of the diver
6. Toggle bIsSwimming on the movement component
7. If bIsSwimming is true set gravity scale to 0, this allows the player to not be pulled down.
8. Then calls SetMovementComponent() passing in Move\_Swimming, this is a method inherited from ACharacterMovementComponent
9. If bIsSwimming is false set gravity scale to 0.
10. Then calls SetMovementComponent() passing in Move\_Walking.
11. Back in “TransitionMovementStates()” in ADiverCharacter the method “ChangeEnvironmen()t” is called on UStressComponent
12. In UStressComponent a timer is set to start ticking up stress overtime.

# **6 Use Case View**

## A screenshot of a computer Description automatically generated

## A screenshot of a computer Description automatically generated

## A screenshot of a computer Description automatically generated

## A screenshot of a computer Description automatically generated

## A screenshot of a computer Description automatically generated

## A screenshot of a computer Description automatically generated