University of Nottingham
UK | CHINA | MALAYSIA

Team 44

Software

Mannual

University of Nottingham

Part II
Software Manual

Contents

1 Overview

This software manual provides technical documentation and guidance for maintaining, extending, and understanding the VR game project developed in Unreal Engine 5. It is intended for future developers, collaborators, and maintainers who need to work with the existing codebase and system architecture. This manual covers key components including project structure, input configuration, gameplay modules, AI behaviour, and performance optimisation. Wherever possible, the document refers to industry-standard practices such as Blueprint interfaces, behaviour trees, and Git-based collaboration workflows. Diagrams, sample blueprints, and configuration references are included to aid implementation and troubleshooting.

2 System Requirements

2.1 Hardware requirements

- VR headset (Meta Quest/Pico 4)
- Motus Explore Treadmill

2.2 Software requirements

- Java SE Development Kit 17.0.10
- Android Studio Flamingo 2022.2.1 patch 2
- Unreal engine 5.4.4
- Meta XR Unreal Plugin V71
- Meta XR Platform Unreal Plugin V71
- Meta Quest Developer Hub

3 Installation Guide

3.1 Java SDK Set up

Download **Java SE Development Kit 17.0.10** From **Oracle**, extract **jdk-17.0.10** into an appropriate location.

3.2 Unreal Engine Setup

Download and install **Epic Games Launcher**, go to the **Unreal Engine** tab in the sidebar, and then go to **Library**. Press the plus sign next to **Engine Versions** until version **5.4.4** appears, install it. press the drop down menu next to the launch button, go too **Options**, tick **Android** under **Target Platform** then **Apply**

3.3 Android Studio Setup

Download and install **Android Studio Flamingo 2022.2.1 patch 2**

Open **Android Studio**, go into **More Actions**, **SDK Manager**, under **SDK Platforms** make sure **Android API 35**, **Android API 34** and **Android 12L** are installed.

In **SDK Tools** under Android SDK Build-Tools 36, install version **35.0.0-rc1**, **34.0.0** and **33.0.1**.

Under **NDK**, install version **25.1.8937393**.

install all of **Android SDK Command-line Tools** .

Under **CMake** install versions **3.22.1**, **3.10.2.4988404**.

Finally also install:

- Android Emulator
- Android Emulator hypervisor driver (installer)
- Android SDK Platform-Tools

Restart after everything is installed.

3.4 Cloning the repository

Use `git clone` to clone repository into "`Documents/Unreal Projects`" (preferably)

3.5 Installing dependencies

Download

- Meta XR Unreal Plugin V71
- Meta XR Platform Unreal Plugin V71

The two plugins need to be installed on [your Unreal Engine install path]/Engine/Plugins/Marketplace
In your uproject, go to **Settings**, **Plugins**, find and turn both plugins on and restart Unreal Engine.

3.6 Building the project

run SetupAndroid.bat on [your Unreal Engine install path]/Engine/Extras/Android In your uproject, go to **Settings**, **Project Settings Platforms**, **Android SDK** set Location of JAVA, Location of Android SDK, Location of Android NDK path

Back to the front page Navigate to **Platforms**, under **Android**, select **Package Project** then unreal engine will start the packaging process

3.7 Meta Quest

Download and install Meta Quest Developer Hub.
Ensure **Developer Mode** is turned on.

3.8 Deployment

Connect headset to PC Open **Meta Quest Developer Hub** Under the **App** tab, press **Add build**, Then open the packaged .apk

4 System Architecture

4.1 High-level architecture diagram

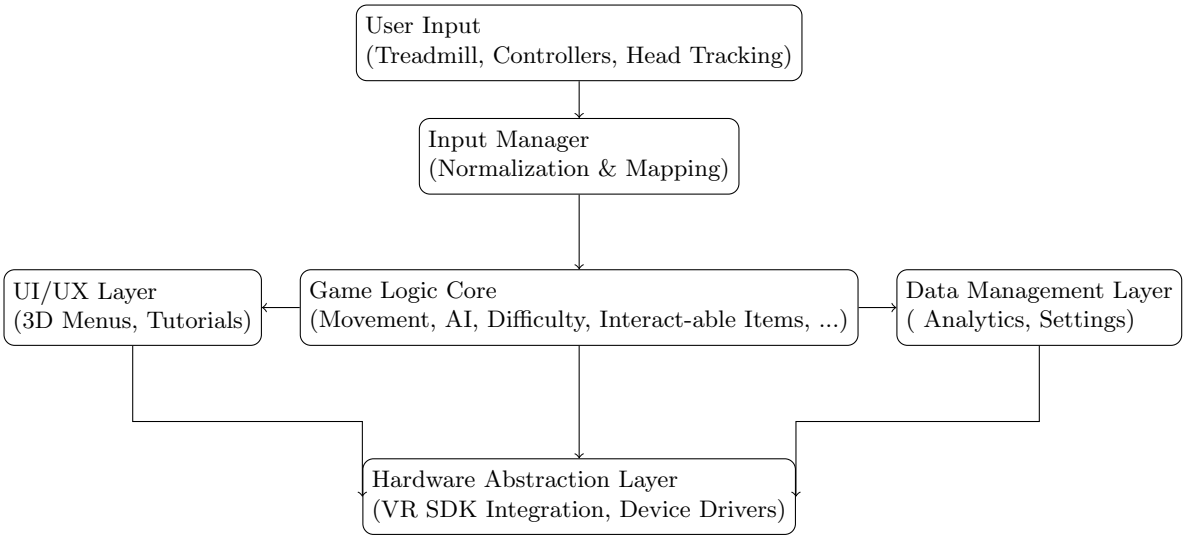

Figure 30: System Architecture of the VR Rehabilitation Game

4.2 Main modules/components overview

VRPawn Control Module: Each mini-game in the project uses a customized VRPawn Blueprint derived from Unreal Engine's default VRPawn setup. These VRPawns are responsible for player movement control, UI integration, and temporary data storage relevant to gameplay. The separation of VRPawn logic per game ensures modular design and allows each gameplay mode to define its own interaction and control scheme.

UI Binding and Interaction: The UI system is organized modularly, with each mini-game assigned its own Widget Blueprint to manage its unique interface layout and logic. These UI elements are integrated into the VRPawn and are visually anchored to the player's view to maintain immersive interaction. Each UI module handles basic in-game navigation, task-related prompts, and dynamic feedback during gameplay.

Game Level/Scene Transition Module: The level transition system enables players to enter mini-games through portals in the central Game Hub and return to the hub at any time via the in-game menu. Each game also supports difficulty selection through the same menu system. Transitions are managed through Blueprint-controlled logic and Unreal Engine 5's level streaming framework, ensuring seamless navigation between different scenes.

User Input Management: The project employs a modular input configuration system using Input Actions and Input Mapping Contexts (IMCs). High-level actions such as movement and interaction are abstracted through Input Actions and then mapped to specific controller inputs via IMCs. This design provides hardware flexibility and maintains input consistency across all mini-games.

4.3 Dependencies

1. **UExplore**: The plugin enabling Unreal Engine 5 to receive input from the VR treadmill was provided by Supervisor Joe Marshall and is available online.[3]

[3] `https://github.com/joemarshall/`

2. **Mazes**: This plugin is used to generate the maze in the Maze level.[4]

5 Module Descriptions

5.1 Chase Level Module

The `Chase` level is constructed using a variety of modular wall assets located in the `Walls` folder. These components are strategically assembled to form a closed, corner-free chase environment designed for smooth VR locomotion.

The player-controlled pawn for this level is `VRPawn_Chase`. This pawn implements all core movement functionalities internally, including forward motion, left and right turning, backward turning, toggling the map display, and switching to a rear view. These actions are bound to the VR controller using Unreal Engine's Enhanced Input System.

The pawn also contains a variable named `Total_Coins`, which is bound to the `CoinUI` interface component. When a player collects a coin in the level, the coin's blueprint logic updates the `Total_Coins` variable in `VRPawn_Chase`, and the new value is immediately reflected in the UI through the binding.

This module demonstrates a clear dependency chain between input mappings, player pawn behavior, collectible actors, and UI feedback, all of which are tightly integrated to support a responsive and immersive VR experience.

5.2 Maze Level Module

The `Maze` level is procedurally generated using the `Mazes` plugin, which creates a dynamic map layout tailored for VR exploration. The movement control in this level follows the same input logic as in the `Chase` level and is implemented within the `VRPawn_Maze` class.

The `VRPawn_Maze` pawn contains two key variables: `health` and `speed`. The `health` variable is dynamically updated based on game difficulty settings and in-game interactions. It is bound to a UI element called `Health_UI`, which reflects its value in real time. When the player collects an `aid` object, the `health` value increases by one. Conversely, when attacked by a `zombie`, the value decreases by one. Once `health` reaches zero, the `Health_UI` will display a "dead" status, indicating game failure.

The `speed` variable is also affected by game difficulty. Difficulty settings are configured through the `Maze_Menu`, which is bound to the `VRPawn_Maze`. The selected difficulty level updates the `difficulty` variable in the `MyGameInstance` class, which is subsequently accessed by `VRPawn_Maze` to dynamically adjust player speed.

Enemy behavior is handled using AI behavior trees. Specifically, the `zombie` actor uses vision detection logic to track the player: once the `VRPawn_Maze` is detected within its camera view, the zombie begins chasing the player based on its behavior tree logic.

This module showcases a tightly coupled system between procedural level generation, difficulty-aware player attributes, user interface bindings, and AI-driven enemy interactions.

5.3 Red Light Green Light Level Module

The `Red Light Green Light` level features a timing-based survival mechanic where visual cues and player motion determine success or failure. The player-controlled pawn in this level is `VRPawn_GLRL`, which implements all player movement logic internally. While the movement implementation follows the same structure as in the `Chase` level, the control scheme is customized to fit the unique mechanics of this mode.

In this level, the color of the ground dynamically changes in response to the `zombie`'s attack behavior. This functionality is orchestrated by the `BP_AttackRange` blueprint, which controls three critical aspects of the gameplay:

- Triggering and managing the zombie's attack animations.

[4] `https://www.mediafire.com/file/0wau3k9d8cj48d7/Mazes_plugin_5.0.3.7z/file`

- Changing the material or color of the ground to reflect safe (green) and danger (red) states.
- Performing real-time player status checks to determine if the player moved during a red-light phase, triggering the death condition if necessary.

The combination of visual cues, precise animation timing, and conditional player input handling creates a fast-paced, high-stakes experience. This module illustrates a tightly integrated system of blueprint-driven environmental feedback, AI animation synchronization, and user control gating, tailored specifically to the "Red Light Green Light" gameplay style.

5.4 Unicycle Level Module

Inside the Unicycle module, there are a number of subsystems that are implemented in order to realise the endless mode, temple-run inspired unicycling game as set out in the game design section (4) of the main report. The main components are laid out as follows:

- Custom unicycle pawn
- Procedural tile generation system
- Powerup system

The unicycle pawn blueprint, located at `Content\Unicycle\BP_UnicycleVRPawn`, contains the core implementation details for the unicycle character. Unlike in the other minigames, the blueprint inherits from the `UPawn` class not `UCharacter`, meaning that more functionality needs to be implemented within the blueprint.

Movement system In order to make use of the functionality that is built in to Unreal Engine, the movement system is implemented using the `FloatingPawnMovementController`, which implements acceleration/deceleration but not gravity. Gravity is simply implemented via a raycast that detects if the Unicycle is grounded, and if not applies a constant movement input downwards. The forward acceleration is applied per tick based off the `treadmillForward` value given by the *Enhanced Input Action* system, in the current forward direction. To extend the movement system, the function `addMovementInput` can be called both from within the blueprint or from an external component. Additionally there is a public parameter `boost`, which acts as a multiplier for the current input. By default it is set to 1, but can be overwritten, which is how the boost mechanic of the powerup is implemented.

Procedural tiles The procedural tile system is central to the Unicycle minigame. It is also designed to be easily extensible and customizable to allow for easy extensions to the game. The `DefaultTile` blueprint forms the basis of this system, and each further tile is designed to be a child blueprint of this class. All tiles contain a green arrow at their origin representing where they will connect to their parent tile, in addition, there may be multiple red arrows, within the exit folder of the blueprint that signal where the children tiles may be spawned (a more detailed description of this this feature can be found below). In order to extend the tile system, all you need to do is add a new subclass of `DefaultTile` and change the scenery or add mechanics, you can then either use the existing exit or add multiple copies of the exit to give the player a choice of direction. Once you add this blueprint to the list parameter `TileList` of the default tile, the rest is handled for you. The tile will spawn randomly as a child of other tiles and be de-spawned automatically when the user has cleared it. Currently, if you want your tile to be more common, you can add it multiple times to the tile list in default tile; however, it could be useful future work to implement a weighted list data structure to have finer control over this functionality.

Power up system The final noteworthy module of the Unicycle game, is the powerup system. Currently there are three types of powerups, regular coins, *mega*-coins, and boosters. You may inspect the existing blueprints to see their implementation, but it is ussually a simple box collider with an `onCollision` event. It is recommended that if you want to add functionality to the VRPawn via a powerup, you create an `ActorComponent` with the relevant data, and then on

collision, to change this data you can use a `getComponentByClass`. This architecture allows a decoupling between the powerups and the specific pawn implementation. In order to add your powerups to the world, you can either create a specific tile where they spawn, or in the default tile, give a random chance of them spawning on the `beginPlay` event. It could be valuable future work to create a system where the default tile contains list of potential powerups and chooses a random one to spawn to allow for ease of implementation - especially if there are many more types of powerups added.

6 Key Feature Implementation

VRPawn-Based UI Attachment: Each mini-game creates a dedicated Widget Blueprint to design its user interface, including buttons, text, and indicators. These UI elements are instantiated in the VRPawn using the `Create Widget` node and assigned to a `Widget Component`. The component is then attached to the player's camera within the VRPawn, ensuring that the UI remains directly in front of the user and follows their head movement, enhancing immersion.

UI Interaction via VR Controller: UI interaction is mapped to the VR controller's input actions. For example, directional navigation within menus is controlled via the joystick axes, while selections are confirmed by pressing down on the joystick. Button behavior is implemented using event bindings within the Widget Blueprint, typically through `OnClicked` delegates for each interactive element.

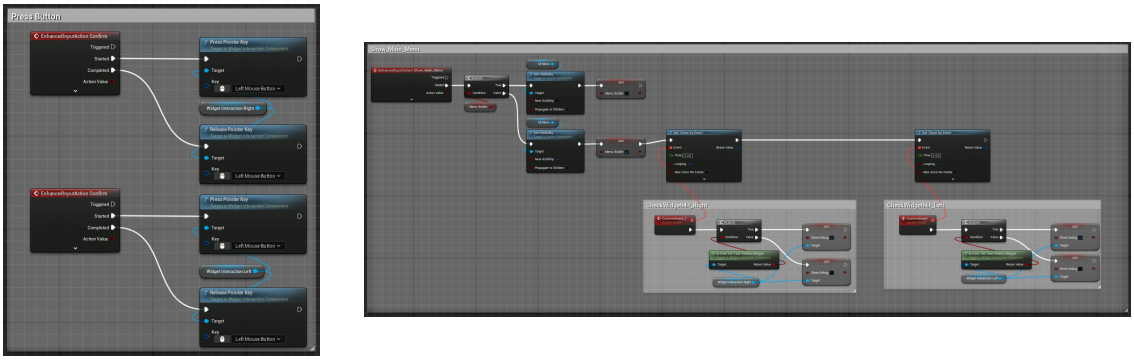

Figure 31: Button Press and Main Menu

Scene and Difficulty Switching: Level transitions are handled using UE5's `Open Level` or `Load Stream Level` nodes, depending on whether a full load or streamed transition is required. Menu widgets provide navigation options to switch between mini-games or return to the Game Hub. Additionally, each mini-game supports dynamic difficulty selection (Easy, Medium, Hard), controlled by setting a difficulty variable via the menu prior to level loading.

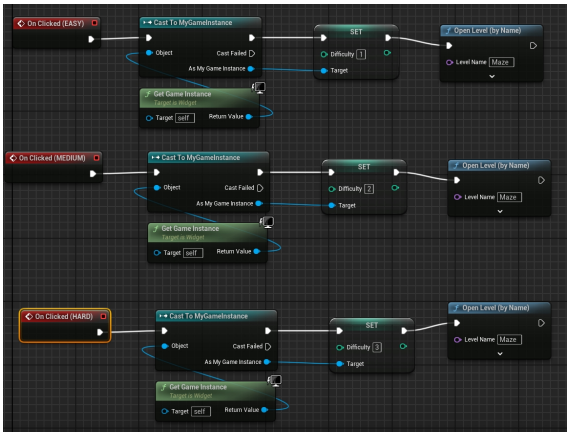

Figure 32: Difficulty Switching

Enemy AI The enemy AI system in our game leverages Unreal Engine 5's Behavior Tree and Blackboard components in combination with blueprint-based logic. Enemies operate in two main states: patrolling and chasing. The behavior tree first checks for line-of-sight; if the player is detected, the AI transitions into a pursuit sequence where the enemy rotates to face the player, moves toward them, and performs an attack upon collision. If no player is in sight, the AI patrols by selecting random locations using `BTT_FindRandomPatrol`, navigating to those points, and waiting for a short random delay before continuing. When the player overlaps the enemy's attack sphere, a blueprint sequence is triggered: the system casts to `VRPawn_Maze` and verifies validity, delays 1 second to simulate wind-up, plays a zombie attack montage via the skeletal mesh's animation instance, and invokes the `TakeDamage` function on the player. A looping timer ensures continuous attacks while the player remains in range. Once the player exits the area, the timer is cleared and the animation is stopped. This system allows for dynamic transitions between idle and combat states based on proximity and visibility, and is modular enough to support further extensions such as varied attack types or smarter behaviors.

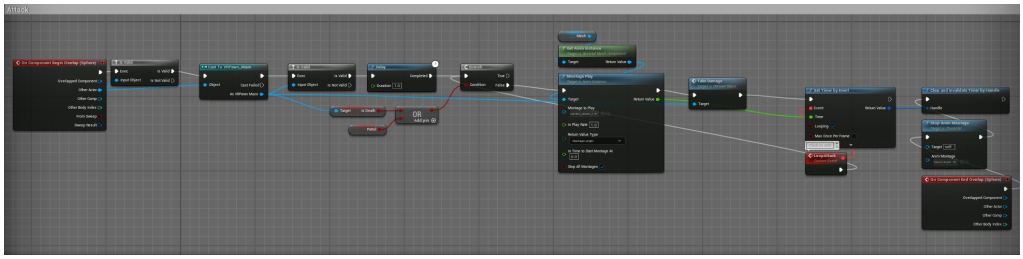

Figure 33: Enemy AI Blueprint

Custom Input System Configuration: The input system uses Unreal Engine's Enhanced Input framework. Input Actions such as `MoveForward`, `TurnLeft`, and `Confirm` are defined as data assets. These actions are linked to physical inputs through Input Mapping Contexts. Within Blueprints, the `Enhanced Input Action` events are used to bind these actions to gameplay behaviour, such as movement or UI control. This setup allows the input configuration to remain abstract and device-agnostic.

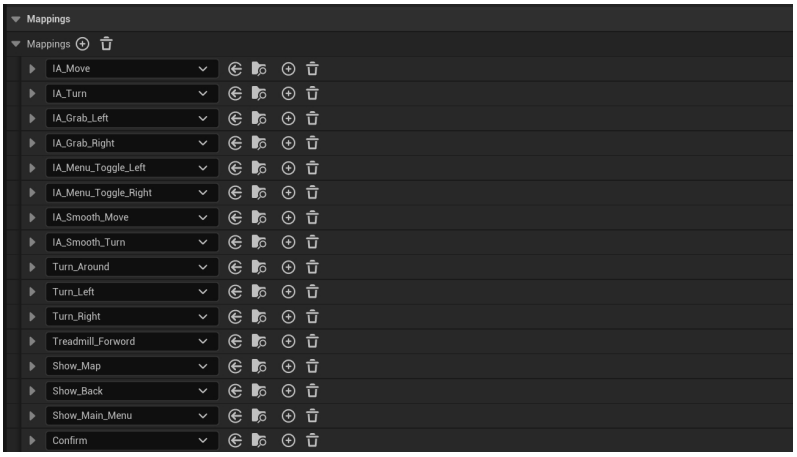

Figure 34: Enhanced Input

Procedural Level Generation: The Unicycle mini-game is designed around an infinite random cave that you have to ride through. Technically, this means that the level and world need to be generated dynamically as you progress through the level. Furthermore, you need to be careful to dynamically unload bits of the level that are no longer in use, especially considering the performance limitations inherent of VR systems. To implement this we actually use something akin to a doubly linked tree structure; where the nodes are subclasses of a base tile blueprint which store references to their previous and future nodes. There are collision boxes which pass a message to the future tiles to spawn their children if they haven't, and if they have then to pass that message forward. Simultaneously, a messaged is passed backwards in a similar way to delete the nodes at the other boundary of the tree to clean up old unused tiles. Finally we keep track of directions that haven't been visited so that there are no 'hanging pointers' - i.e. tiles that don't get de-spawned. This method allows us to continuously traverse through a world whilst making sure only a small number of tiles are ever loaded at once. In addition, because the tiles are implemented as arbitrary subclasses of a default tile class, they have complete control in the look and feel of the tile as well as to how many future tiles are able to be spawned and where they should spawn from. This means that not only are we able to develop a variety of different tiles, but also that a future develop who might take up this project will be able to seamlessly and easily extend the existing system with their own content.

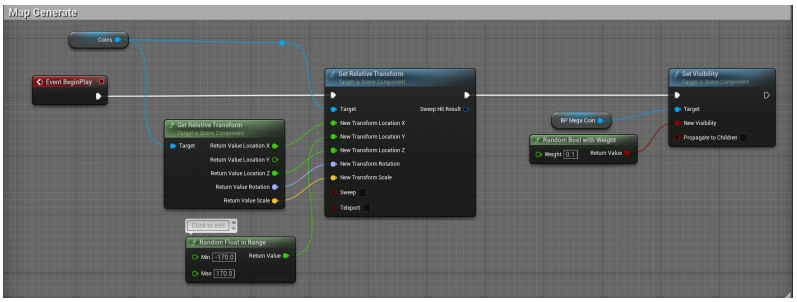

Figure 35: Generation

Unicycle Pawn and movement controller: By default, Unreal Engine provides a subclass of `UPawn` called a character. This character contains a pre-implemented bipedal movement system, which for the standard first person controls used in the maze, chase and red light green light was a suitable base to build their respective pawns over; however, because the movement model for the Unicycle is very different, it was not possible to use the `UCharacter` as a base. To implement the equivalent functionality we needed to instead create a child of the base `UPawn` class. We could make use of the simple floating movement controller that is built in to the engine which handles

(de)acceleration for us but we still needed to implement gravity and steering that responds to the head movement.

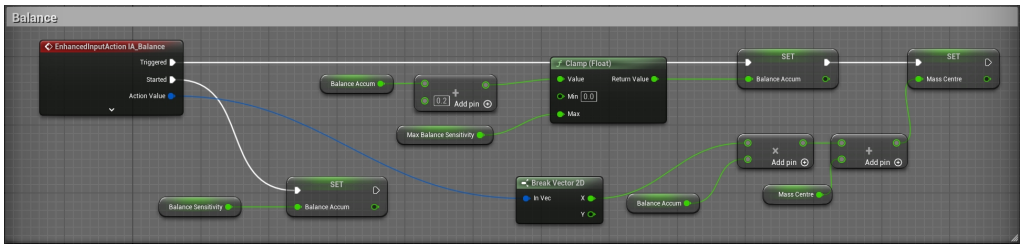

Figure 36: Unicycle Balance

7 Configuration

7.1 Game configuration

7.1.1 Maze Game Parameters

The `Maze` level includes several configurable parameters that control gameplay dynamics, enemy behavior, and player performance. The current default values and their corresponding implementation locations are listed below:

- **Enemy Behavior** (defined in `BT_Enemy`):
  - `Chase Speed:` 300 units
  - `Patrol Speed:` 100 units
  - `Patrol Radius:` 1000 units
  - `Wait Time:` 1.0 ± 0.5 seconds
- **Game Difficulty** (stored in `MyGameInstance`):
  - `difficulty:` Integer values from 1 (easy) to 3 (hard)
- **Player Attributes** (defined in `VRPawn_Maze`):
  - `speed:` 0.8 (easy), 1.0 (medium), 1.2 (hard)
  - `health:` Integer range from 0 to 3

7.1.2 Chase Game Parameters

The `Chase` level includes a set of configurable parameters that govern enemy behavior, player attributes, game difficulty, and collectible tracking. The current default values and their implementation references are listed as follows:

- **Enemy Behavior** (defined in `BT_Enemy`):
  - `Chase Speed:` 300 units
  - `Patrol Speed:` 100 units
  - `Patrol Radius:` 1000 units
  - `Wait Time:` 1.0 ± 0.5 seconds
- **Game Difficulty** (stored in `MyGameInstance`):
  - `difficulty:` Integer values from 1 (easy) to 3 (hard)
- **Player Attributes** (defined in `VRPawn_Chase`):

  - `speed:` 0.8 (easy), 1.0 (medium), 1.2 (hard)
  - `health:` Integer range from 0 to 3

- **Coin Tracking** (handled in `VRPawn_Chase` and associated blueprints):
  - `howmanycoins:` Number of coins collected during gameplay
  - `Total_Coins:` Total number of collectible coins in the level, typically ranging from 50 to 150

7.1.3 Red Light Green Light Game Parameters

The `Red Light Green Light` level includes timing-based enemy behavior configurations that control the attack frequency and animation variation of the zombie character. The current gameplay parameters are as follows:

- **Zombie Attack Behavior**:
  - `Attack Interval:` Randomized between 4 to 5 seconds
  - `Attack Animation Probabilities:`
    - Jump attack: 50%
    - Left swipe: 25%
    - Right swipe: 25%

7.1.4 Unicycle Parameters

In addition to the tunable and extensible tile and powerup system the following parameters are available, in particular to customise the Unicycle Pawns movement:

- Unicycle Pawn Parameters:
  - `TurnAmmount`: Controls the sensitivity to head tilt
  - `Boost`: A temporary multiplication to the treadmill input
  - `FloatingPawnMovementController`: There are a number of parameters in the floating pawn movement controller that control the feel of the movement. These include `acceleration, deceleration` and `maxSpeed`.

- Default Tile Parameters:
  - `TileList`: Controls the tiles that will be spawned. It is possible to add the same tile multiple times to increase the probability of certain tiles spawning.
  - `BackpropThresh`: This controls the recursion depth of the tile despawning algorithm. Note that this can have a big impact on performance.

- Misc:
  - `FogDepth`: In the Unicycle map world, there is a postprocessing effect box. This contains the settings for the fog, most importantly fog depth. The purpose of the fog is twofold: firstly to create an atmospheric environment, but also to hide the fact that not many tiles can be spawned at once without affecting performance. If future work is done on the performances issues of the Unicycle map, then it may be possible to turn down the fog.

7.2 Input mappings

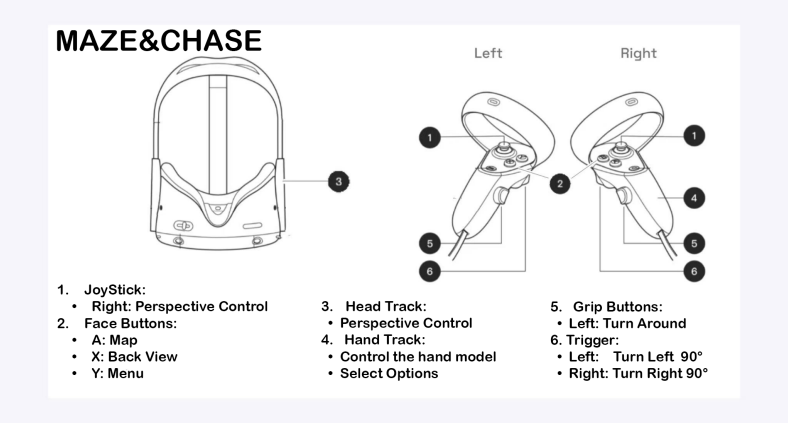

Figure 37: Control Mapping for Maze & Chasing Game

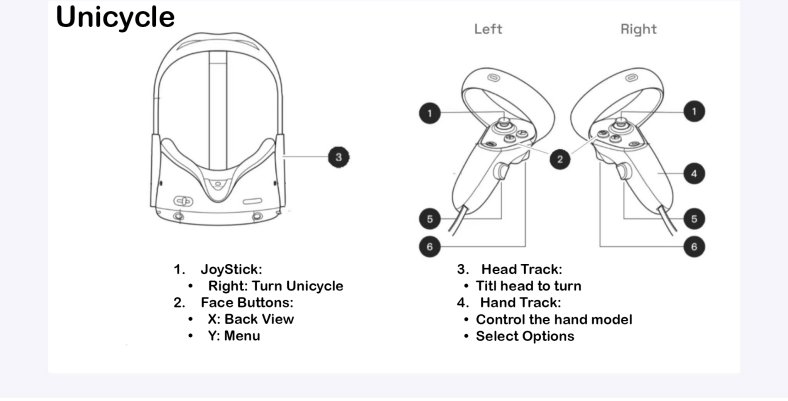

Figure 38: Control Mapping for Unicycle Game

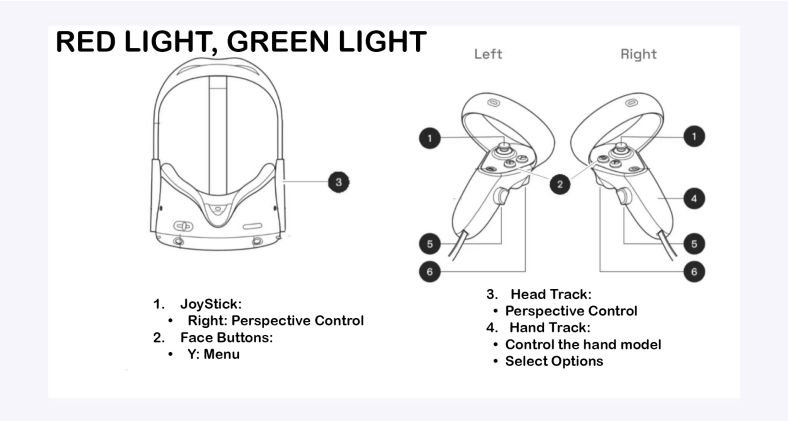

Figure 39: Control Mapping for Red Light, Green Light Game

8 Troubleshooting

Issue	Solution
File content missing	Since this repository has Git LFS enabled, installing it was necessary for proper functionality.
Modules failed to build	Delete the folder **Binaries**, **DerivedDataCashe**, **Intermediate** and **Saved**. Run **buildproject.bat**. This will directly build the project and any necessary modules.
Unable to package - ExitCode: 8	This happens because Unreal Engine is trying to package a module that is editor-only. To fix this, either disable the plugin that depends on that module or rewrite the build file to exclude the editor-only module.
Unable to package - ExitCode: 1	This happens when a dependency that Unreal Engine is trying to package is not in the game Content folder. To fix this, try to migrate the required asset into the content folder.

Table 9: Caption

9 Version Control & Collaboration

9.1 Git Branching Model

Given the binary-heavy nature of Unreal Engine 5 projects—particularly with assets such as textures, 3D models, and blueprint files—version control posed significant technical challenges. Large binary files quickly inflated the repository size and degraded performance for common Git operations like `clone`, `fetch`, and `pull`.

To address this, the team adopted Git Large File Storage (Git LFS)[5], a Git extension designed to handle versioning of large binary assets. Git LFS replaces large files in the repository with lightweight pointers and stores the actual files externally. This significantly reduced the repository size and improved operation speed.

9.2 Merging Workflow

While Git LFS mitigated repository bloat, it introduced challenges in merging conflicting binary assets, such as Unreal blueprint files. Due to Git's inability to perform meaningful diffs on binary content, even minor, non-overlapping changes required full manual resolution.

To minimize such conflicts, the team adopted a collaborative practice based on structured coordination. Team members were encouraged to:

- Work in isolated branches scoped to specific features or levels.
- Synchronize frequently with the `main` branch through small, incremental merges.
- Prioritize communication when potential overlap in blueprint editing was expected.

Although this approach temporarily relaxed the convention of merging only fully complete features, it was effective in reducing merge effort and maintaining momentum in development. For larger teams or enterprise environments, more robust solutions—such as Git LFS file locking or specialized game version control systems (e.g., Perforce Helix)—may be more appropriate.

[5] `https://docs.gitlab.com/topics/git/lfs/`

9.3 Handling Merge Conflicts

During early stages of development, the team encountered a situation where two branches had diverged significantly, both containing essential features. Manual conflict resolution across blueprint assets proved time-consuming and error-prone.

To resolve the situation, the team agreed on a short-term *feature freeze*, during which no new features were developed. All efforts focused on integrating the divergent branches, resolving conflicts, and validating project integrity. This incident reinforced the importance of regular merging and effective communication. Subsequent development cycles proceeded more smoothly, with merge conflicts kept to a minimum.

10 Known Issues & Limitations

10.1 Current bugs

Right-hand Controller Trigger Unresponsiveness: Occasionally, the trigger on the right-hand VR controller fails to respond when performing click interactions, resulting in inconsistent input detection.

Chase Map Flickering Issue: In the Chase level, overlapping map assets cause visible texture flickering upon game start, due to Z-fighting between duplicate or closely positioned meshes.

Chase Enemy Navigation Bug: Enemies in the Chase level may occasionally get stuck in certain areas of the map, especially near corners or geometry seams, preventing them from pursuing the player as intended.

10.2 Limitations of Motus hardware or Quest 2

Quest 2 Headset: The Quest 2 headset offers limited hand gesture recognition capabilities, primarily detecting basic gestures without capturing precise or nuanced hand movements. Additionally, it relies heavily on straps for secure fitting, making prolonged use uncomfortable due to pressure and strain around the head.

Motus Explore Treadmill: The Motus Explore Treadmill can only detect whether the user's feet are in motion, outputting a simple movement signal. It does not track the direction of foot movement, and therefore cannot be used to determine the user's movement direction based on foot orientation.

11 Future Work & Extension Points

11.1 Architectural Extensibility and Refactoring Plans

The project is currently developed using Unreal Engine 5 Blueprints, which facilitates rapid prototyping but requires structure for long-term scalability. The following architectural practices are suggested to support future extension:

- **Dedicated Folder Structure per Mini-game:** Each mini-game should reside in its own folder under the `/Games` directory. Inside each game folder, include:
  - `/Blueprints` – all logic-specific Blueprints (e.g., GameMode, PlayerPawn).
  - `/Assets` – static meshes, textures, and audio used specifically for this mini-game.
  - `/Animations` – animation sequences and blendspaces.
  - `/UI` – widgets, HUD elements, and interface Blueprints.
  - `/Data` – any DataTables, configuration files, or tuning data.

- **Class Inheritance Structure:** Introduce a base class hierarchy to simplify behavior reuse. For instance:
  - All mini-game `VRPawn`s inherit from a generic `BaseVRPawn`.
  - Shared functionality (movement, camera control, tracking input) is handled in the base class.
- **Modular Blueprint Structure:** Refactor gameplay logic into reusable modules (movement, tracking, UI, etc.) to ensure separation of concerns and ease of updates.
- **Resource Cleanup:** Periodically audit and remove unused assets from the project to reduce clutter, save disk space, and improve loading performance.
- **Consistent Naming Conventions:** Apply standard naming schemes across folders, Blueprints, and variables to improve clarity and support collaborative development.
- **Data-Driven Design:** Externalize tunable values (e.g., speed, scoring thresholds) via Blueprint Data Assets or DataTables to allow non-programmers to tweak gameplay.
- **Event-driven Component Interaction:** Use Blueprint Interfaces and Event Dispatchers to create decoupled communication between actors and components.

11.2 Potential New Features

Feature	Description
Reward Systems	Include motivational incentives such as progress badges, trophies, and performance milestones to encourage consistent engagement during rehabilitation exercises.
Rest and Pause Mechanisms	Incorporate clear and user-friendly pause and rest functions, enabling users to take breaks comfortably. This is especially valuable for those undergoing physical therapy.
Customisable Visuals	Provide options to adjust text size, color contrast, and interface simplicity to accommodate users with visual impairments or cognitive processing difficulties.
More Natural and Flexible Controls	Replace or complement snap 90° turning with joystick or smooth-turn options. Allow adjustable sensitivity for foot or head movement.
Expanded Gameplay Mechanics	Add more in-game mechanics and layers of interaction to prevent repetition and maintain user interest, such as stretch-release or heel-lift movements.
Difficulty Scaling	Introduce levels of difficulty or progression systems that adapt to users' rehabilitation stages and physical capabilities.
Immersive Tracking Features	Enhance hand and head tracking fidelity to increase immersion and physical responsiveness.
Time-based Goals and Leaderboards	Include timers, speed-based rewards, and high score boards to introduce friendly competition and performance tracking.
Boost Elements and Game Flow Enhancers	Implement dynamic speed boosters (e.g., auto-triggered accelerators) to intensify pacing and engagement.
Immersive Audio Experience	Incorporate vivid and adaptive background music to enhance user immersion and emotional engagement.
Rehabilitation Analytics Integration	Work with medical advisors to track step frequency and movement data to generate progress records and insights.
Bug Fixes and UX Improvements	Address issues such as confusing layouts (e.g., maze printers) and reward coin glitches to improve satisfaction.
Directional Treadmill Integration	Integrate hardware that can detect footstep direction to improve locomotion control in future iterations.
More 3D Models	Multiple 3D models have already been created but due to time constraints and some changes during development, they weren't used. Many of these models could be re-used at a later date.

Table 10: Proposed Features and Enhancements for Rehabilitation-focused VR Gameplay