Action RPG in VR Template  
Game Design Document

# Project’s Purpose/Aim

This template project aims to test the capabilities of the main mechanics that interpret a VR Action RPG genre. The project is a base template for any future developer to pick up from this concept and make their version of a VR game with this genre. This template will be licensed under Creative Commons Attribution (allowed to be taken and modified as long as it is credited for use).

This document will present assets used to create certain key objects and features of the project and describe the main three mechanics being focused on: movement, weapon handling, and battle system.

# Assets & Features

The assets used in Unity are purely primitive shapes and necessary objects used from templates and the XR Interaction Toolkit. The Unity version in this case is ‘2022.3.37f1’.

The Unity XR template was used to start this project. It contains a sample scene that tests the capabilities of the ‘XR Origin Rig’ object. That object is the main controller for the player as it uses a CharacterController component to utilise the headset and hand-controller capabilities. Additionally, it uses a Locomotion Mediator script to handle player controls in terms of movement, such as teleportation, climbing, and poke interactions. The XR Origin uses continuous movement and rotation for a smooth experience. The player's hands use primitive spheres as a replacement for sample controller objects. Lastly, there is a built-in script for the player as it handles damage detection and invincibility times. This will be explained later in the Movement documentation.

The sword weapon in this project is created using Unity primitive cubes and a built-in script attached to the sword blade. The sword is a child of the RightController object, meaning that moving the right controller will also move the sword as they please. The script on the sword blade handles damage detection against other specific objects. This will be explained later in the Weapon Handling documentation.

Enemies are created using Unity primitive capsules with their built-in script and NavMesh behaviour. NavMesh is used as an AI Pathfinding algorithm to allow the enemy to travel along the map to a destination by avoiding obstacles and finding the nearest possible path. By default, the enemy’s destination is set to the player’s location. The built-in script handles damage detection, chase detection, and health. When the players get close to an enemy within a certain radius, the speed in NavMesh applies from zero to a given value so that the enemy starts chasing. This can be toggled on and off when needed (if enemies will move towards the destination regardless of radius). This will be explained later in the Battle System mechanic documentation.

The rest of the objects not described in the scene are minor additions to test and create variants of damageable objects:

* Breakable Walls
* Test Dummies

There are also some singleton objects used in scenes such as GameManager and UIManager. Again, they will be explained later in the Battle System mechanic documentation.

# Main Mechanics

Three main mechanics are focused upon in this template:

* Movement
* Weapon Handling
* Battle System

By taking these mechanics into account for an Action RPG style, they can be used generally to create similar experiences to how the template operates. However, future developers can pick up this template and make their own versions. For example, they can add more mechanics such as new weapons, more sophisticated battle systems, and new enemy behaviours.

## Movement

The Locomotion Mediator script controls and handles the use of player interactions, such as movement. For this template, the player is using continuous movement and rotation. These controls are defaulted to the following controls:

* Left Stick – Moving
* Right Stick – Turning

Despite the player’s rotation, the player can instead look at a certain direction to move towards, which is known as dynamic movement. Instead of relying on a static forward axis, the direction is handled in the direction where the headset is pointing.

Additionally, the speed of movement and turning can interact with other scripts if they are referring to required namespaces via the ‘using’ keyword.

## Weapon Handling

The sword in this template is only used and is scripted as the main weapon that a player possesses. It has a hit detection on the sword blade when it collides with a damageable object.

When the sword hits an object, it calls OnCollisionStay() and collects the collider from the other object, allowing it to call any public functions and methods on the other object’s collider.

However, there is a built-in cooldown that the sword has, which prevents the player from striking anyone with the sword for a set amount of time. This is shown when the sword blade turns red. The red blade indicates that the sword needs to cool down before the next strike. This is to prevent players from spamming their sword strikes endlessly but can be toggled off by adjusting the cooldown time to zero.

Lastly, the sword has a speed requirement to encourage players to swing their swords fast enough to attack them. Since the sword is a part of the CharacterController, using a Rigidbody is highly discouraged as it may cause conflicts with physics. Instead, it uses a traditional method of calculating speed, which is to find the distance gap between the current and previous transform positions over time. This is done without the influence of the player moving about while swinging their sword (that is to negate the player’s speed when calculating the sword’s speed).

## Battle System

The battle system is handled by numerous scripts to control, and display based on health systems and damage.

### GameManager

This script handles the player’s health and condition in any scene due to its singleton nature and being marked as ‘DontDestroyOnLoad’. When a player is hit, it calls a public function called PlayerHit(), which lets the game know that it should reduce its health by one. If the player’s health is zero, it will call Dead() and set the respawn timer to a set amount of time. It also lets the player know that it should disable movement and take away the sword. Once the respawn timer is zero, it will reload the scene.

### UIManager

This script handles the UI elements such as displaying the player’s health bar. It is another singleton object that can be called anytime. When the player is hit, it calls PlayerHit() to let it update the health bar with the new change in health, resizing via localScale. It also displays a red screen effect to indicate that the player is hit. This handles the alpha value directly for fade effects, such as fading out over time. It can also be used to display a debug element (showing the speed of the sword), which can be toggled anytime.

### IDamageable

This is an interface script that only has one global function: Damage().

Certain objects like the Player, Enemy, BreakableWall, and SwordTriggerTest (Test Dummy) have scripts that implement the IDamageable interface. This is useful to gather objects that can be damaged without referring to them individually (using CompareTags to do different functions).

The sword makes good use of calling functions on other objects when it is damaged. Using TryGetComponent<IDamageable>(out IDamageable *var*), it will only connect when it hits an object with the IDamageable script, and when it does, call Damage(). That function can depend on how it behaves in each script and will always have the same function call if the function is the same as the one from the interface.