Viral Impact Technical Design Document

Viral Impact Technical Design Document

By ISCREAM

Introduction

This Technical Design Document (TDD) serves as the blueprint for the design and implementation of Viral Impact. It is made to guide the development team, stakeholders, and associated personnel through the project's architectural decisions, technical requirements, and system functionalities.

At its core, the TDD is designed to help with clear communication across various teams. The document details the rationale behind critical design decisions and the interactions among system components, ensuring that every member of the team understands the framework within the project. This comprehensive approach not only supports current development needs but also lays a solid foundation for future scalability and enhancements.

The scope of this document encompasses all critical technical aspects of Viral Impact. It provides an in-depth look at the system architecture, including detailed diagrams, data flows, and component interactions. In addition, the TDD outlines the technical requirements, including hardware specifications, software tools and third-party integrations. By setting clear expectations for each component, the document ensures that the system is built with both robustness and efficiency in mind.

Delivery Platform and requirements

The target devices for our project currently are PCs and laptops, as the game is being developed exclusively for these platforms. Specifically, the project will be designed and optimized for the Windows operating system, ensuring compatibility with a wide range of Windows-based machines. At this stage, there are currently no plans to support other operating systems such as macOS or Linux. Support for other systems and consoles could be possible during development by future teams. Especially if the game is well received.

Furthermore, the game will require a keyboard and mouse as the primary input methods. Controller support is included. This helps people that are not good with handling keyboard and mouse while playing the game. Touchscreen support will not be included in the initial development, meaning devices such as tablets and 2-in-1 hybrid laptops operating in tablet mode will not be compatible. The design choices for input methods are intentional, as they align with our gameplay mechanics and accessibility considerations. Future updates may explore alternative input methods based on user feedback and demand, but for now, the focus remains on providing an optimized experience for traditional PC and laptop users.

As for our device specifications the minimum requirements would be:

For 1920x1080 60 FPS:

CPU: Intel Core i5 7200U @ 2.50GHz

GPU: NVIDIA GeForce 940MX 2Gb VRAM

RAM: 8Gb DDR4

Storage Space: 2Gb

Game Mechanics and Elements

Here all game mechanics and systems will be outlined for clarity of the projects progress:

Systems

Unity's Localization System

This is a unity package that we will use to translate text between the Dutch and English languages. For more information on how to use it, see the “Importing Dialogue Instructions” document.

You can use the LocalizedString variable to store the localization and text of both languages. Then use GetLocalizedString() to get the string of the piece of text in the current language. Furthermore. If you use the LocalizationSettings.SelectedLocaleChanged event, you can hook up a method that gets called every time you change the current language of the game. Allowing you to update the translation with GetLocalizedString().

TextMessageHandler

The TextMessageHandler is made to ensure that the text components get properly updated with the current language using the localization system. This class is inherited to allow multiple localized strings into one string.

Audio Manager

AudioManager.cs: A persistent, singleton audio controller that loads FMOD banks, initializes buses/VCAs, manages global volume sliders (master/music/SFX), starts background music, plays one-shots, and automatically triggers UI hover/click sounds using Unity's EventSystem with a small cooldown.

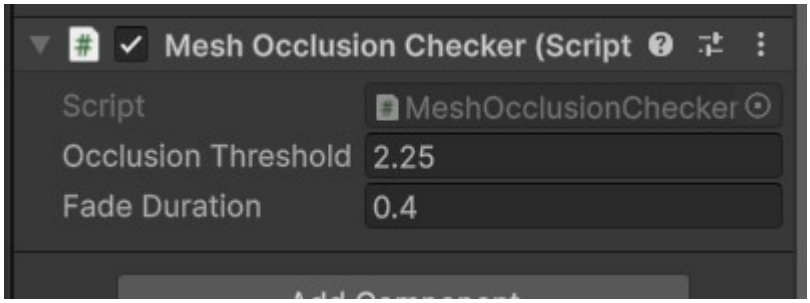
FMODEvents.cs: A persistent, singleton "event library" that stores serialized FMOD EventReferences (UI, music, player SFX, enemy SFX) so other scripts can access them easily via FMODEvents.instance. This script also helps so you don't have to put Using UnityFMOD at the top of every script.

Cinemachine

While there aren't any substantial changes it is important to note that for the camera follow and transitions we use cinemachine which is a very convenient Unity tool

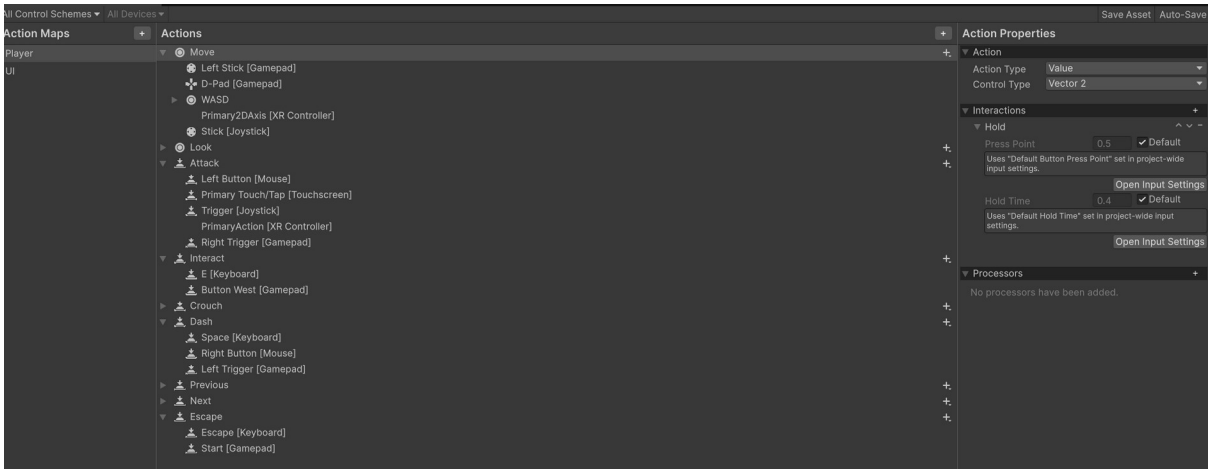
MeshOcclusionChecker

This script makes sure the player is always visible by fading out any environment meshes that block the camera's view. It checks which objects are between the camera and the player, and smoothly fades those objects to a lower transparency. When they are no longer in the way, the script fades them back in. The script is on the Main Camera.



Input Manager

The InputManager is the system that listens to all player input from the keyboard, mouse, or controller. It uses Unity's Input System together with the generated InputSystem_Actions script to keep all controls organized and easy to manage. All keybinds and input settings can be found and edited in the Assets/InputSettings folder, where the Input Actions asset is stored.

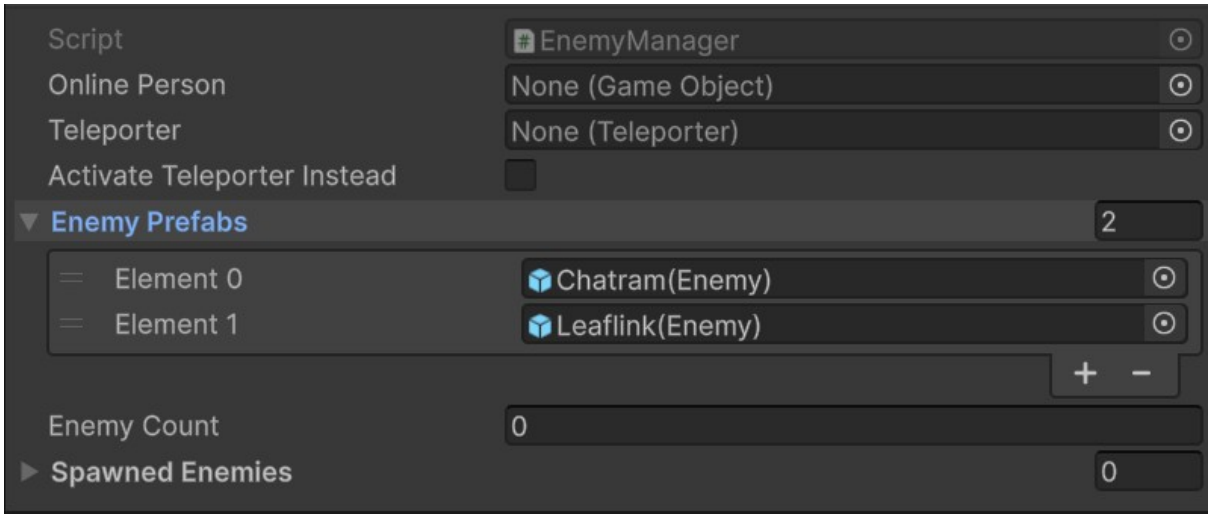


Enemy Manager

The enemy manager is currently being used in levels to notify the spawners which enemies should be spawned in the level. Each enemy spawner contains a script with the sole purpose of notifying the Enemy Manager that it can treat the game object as a spawner.

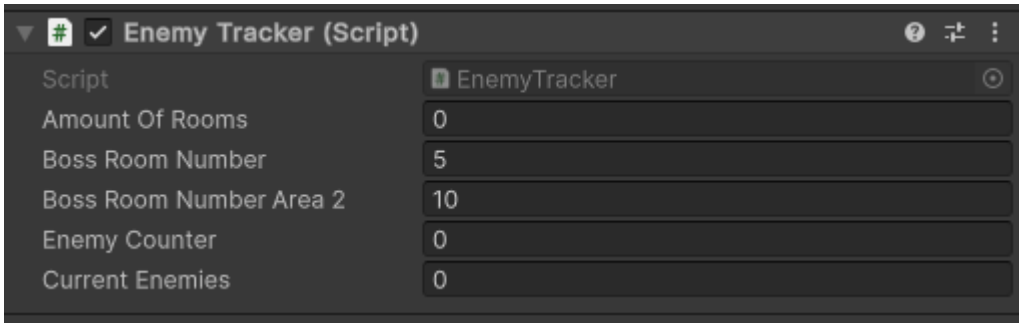
The online person and teleporter gameobjects are there to turn on the game object after all the enemies have been defeated. The enemy prefabs list variable is there to see which enemies it can randomize between every spawner.

The enemy count and spawned enemies are there purely for debugging purposes and they are not important. Because of the current state of the level generation, the enemy manager should be in every level prefab due to how the game build handles instantiating objects. This manager is not DontDestroyOnLoad.



Room tracker

The enemy and room tracker are in the EnemyTracker script. It handles the amount of enemies spawned and activates the Online Person also unlocking the portal in order to continue in the dungeon. It also tracks the amount of rooms visited which is important for encountering the boss fights and also changing the areas. On the image below the Boss Room Number and Boss Room Number Area 2 are the amount of rooms that have to be visited to encounter the bosses. After the 2nd boss the loop is completed and after a “You Win” screen the played goes back to the HUB.



Dialogue

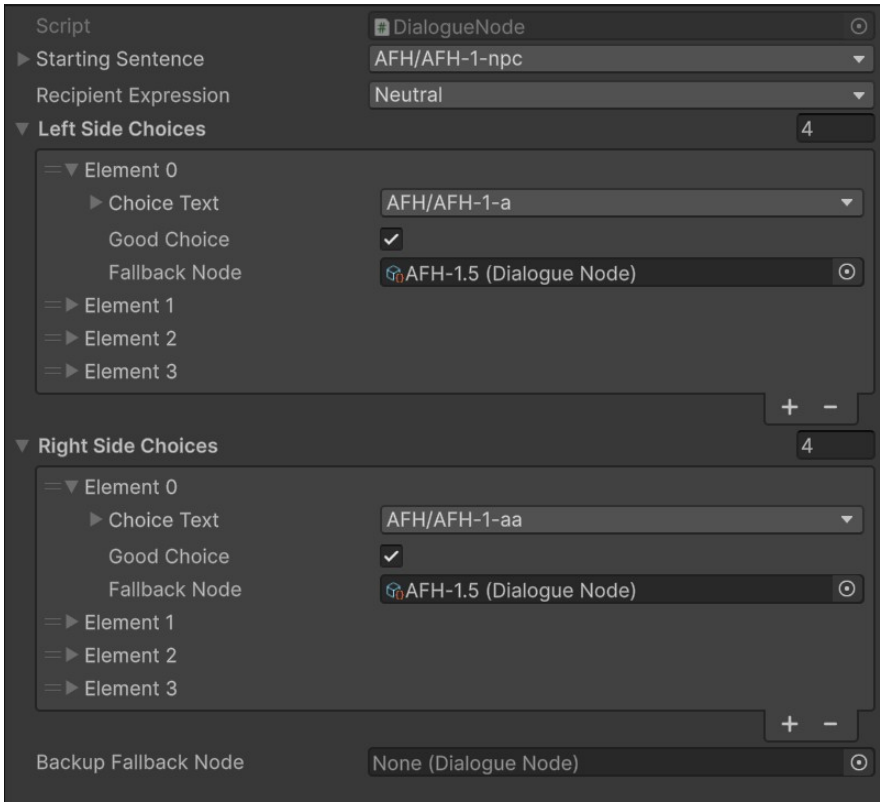
The dialogue system is a very vital part of the project responsible for the learning part. It is one canvas that contains the entire UI structure and scripts that makes the dialogue functional. To implement and use dialogue in the game, refer to the “Importing Dialogue Instructions” document.

Dialogue System Script

This script holds the biggest pieces of functionality. The onenable sets up the first message (if the starting node is not null) and some other UI elements like the image of who you are talking to and the name of said character.

Setting up the character message and your choices are done through Nodes. Each node contains what the character says, their expression, the choices you have on the left and right half of the dialogue and the backup fallback node in case no choices are filled in a node. Each choice contains the text, if it's a good option and what node the dialogue should continue with.

When setting up a node, it instantiates enough buttons to hold all the possible choices or hides buttons that are not filled in with choices. Then it resets (if needed) the current answer in the bar above the choices and makes the continue button not interactable anymore. It also shows up whatever the character wanted to say by creating a message bubble with the localization text.

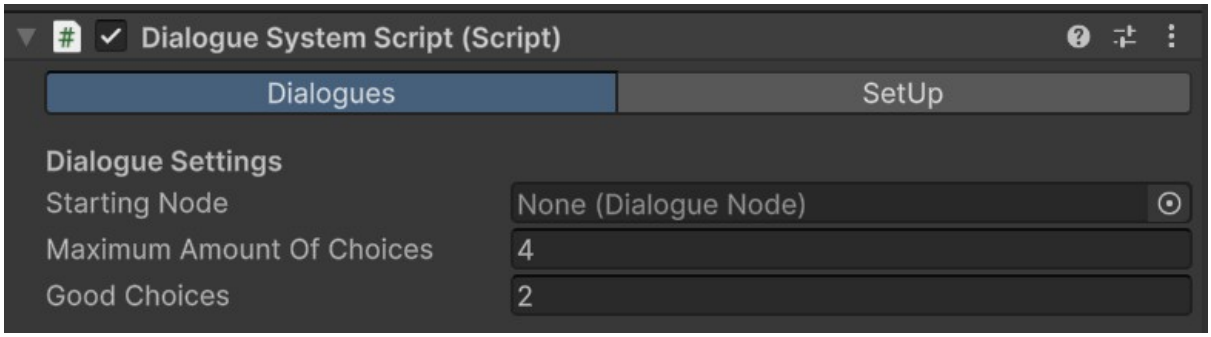


The order of choices is randomized. If there are no choices the system will automatically call OnContinue after a small interval. OnContinue looks at the choices (if any choices have been made) and saves if the outcome is good or not. Then it proceeds to see what the fallback node is or what the

backup fallback node is if there are no choices made. If there is no node to go to, the dialogue system stops.

The StopDialogue method stops the dialogue and delivers the outcome to all classes that need it. It then hides the entire dialogue canvas.

We split the variables up in two sections due to the amount of variables. The Dialogues section is purely customizable for the dialogue itself. The SetUp section contains all the variables that need a value to link everything up and make the dialogue system work. This is done using an editor script. The editor script allows us to put variables in a set place under a set section.



Continue Button Dialogue Script

This script handles the continue button. If the choices on both sides are pressed, the continue button will know and make it interactable again.

Dialogue Engager

This script is on the character. It contains some necessary information that the dialogue system will use and it is the script that activates the dialogue system when interacted with it.

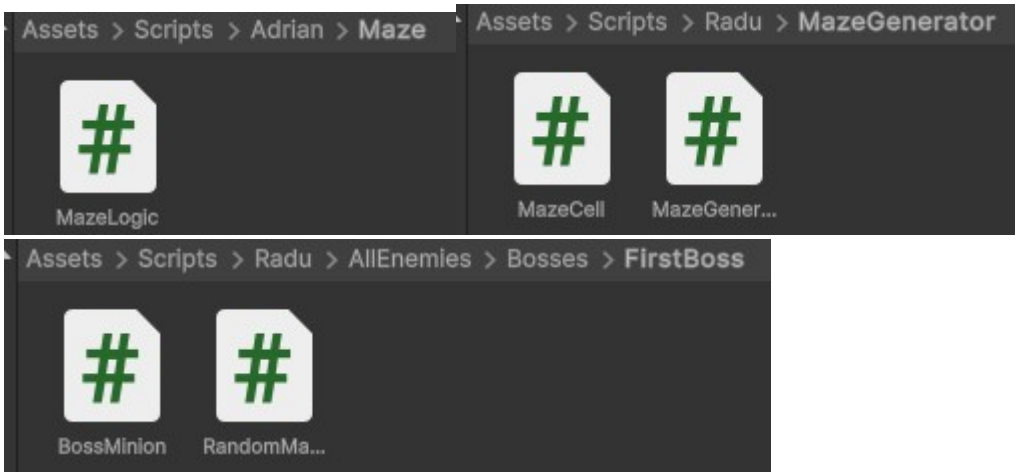
Room Spawner

The Room spawner is responsible for the Room generation and different rooms types. It uses two scripts: One for choosing the type of a room through weighted randomness (CreateRoom) and the other chooses the spawn position for the Player, Portal and Online Person (RoomPlayerSpawn). For the second one in the Room prefab there are empty objects used for spawn points of the 3 objects mentioned (Image

below). The script spawns the player on the first spawn point and after calculates to put the Portal in the furthest spawnpoint from the player and the Online Person to the closest one relative to the player spawn point.

Maze Logic

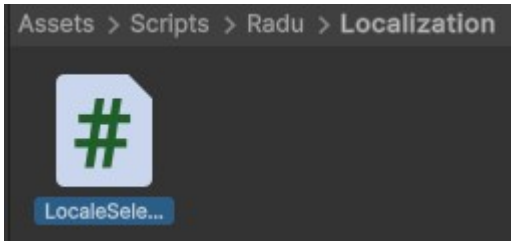
For the Maze Logic we have a preset of 3 mazes. Each time you enter the 1st boss room a random maze is chosen between the presets. This is done with the Random Maze script. Each maze part has a specific Maze Logic script that is responsible for moving the maze up and down and also for lowering each individual part.



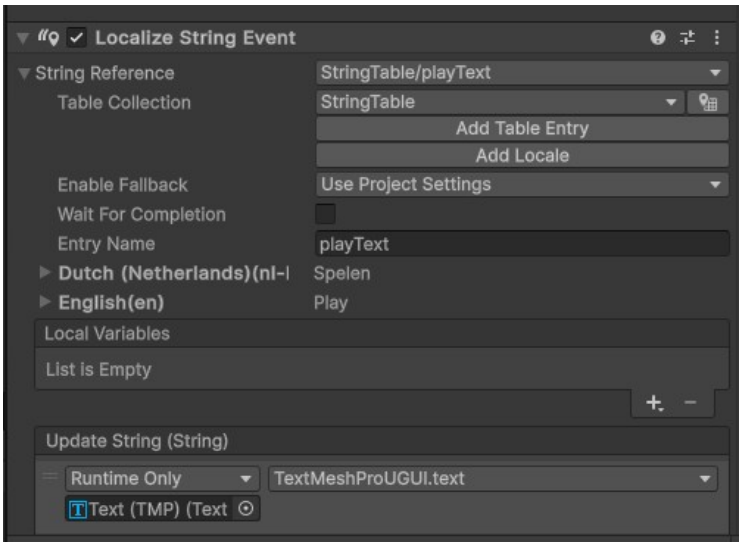
Localization (Language) system

The localization system is a unity package that we are using for smoother translation of the whole game since we have 2 languages (Dutch and English). The localization system uses a table that can be found in Window/Asset Management/Localization Tables. There you can find all translation keys and can add more to the project. Translation of static text can be done with the Localize String Event component added to the UI text object where the text needs to be translated (image below). Important to note that in the Update String (String) event you have to reference the same object the component is attached to. For dynamic translation please

check the examples throughout the project of the Locale Selector script for an example.



Mechanics

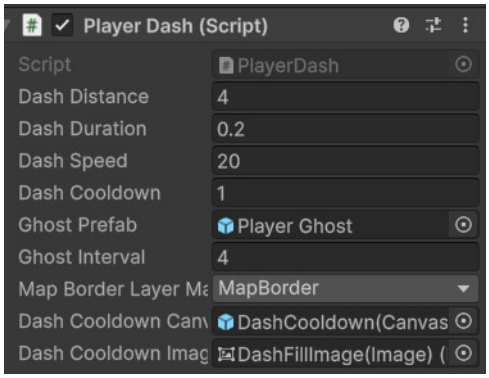


Movement

Movement is done in the PlayerMovement script and through the input manager instance.

Dashing

Dashing has its own script on the player's prefab called PlayerDash.cs. To receive input it subscribes to the Inputmanager dash action. The Dash feature has a ghost effect where sprites get spawned in behind the player and faded out.

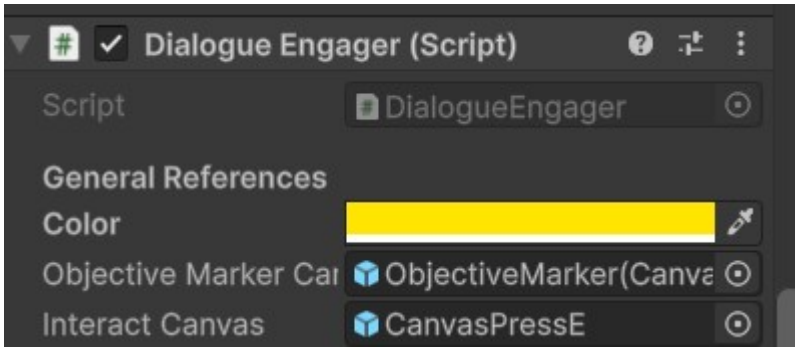


Attacking

Attacking happens in the PlayerAttack.cs script. It is important to note that the Attack itself is a 45 degree swing that follows the direction of the mouse cursor, not the position the player is facing, however with the setting PathPointer this is reversed.

Interacting

Interactions in the game are triggered when the player enters an *IsTrigger* collider. This activates a UI canvas prompting the player to interact by pressing a button. The logic for this is handled in DialogueEngager.cs.



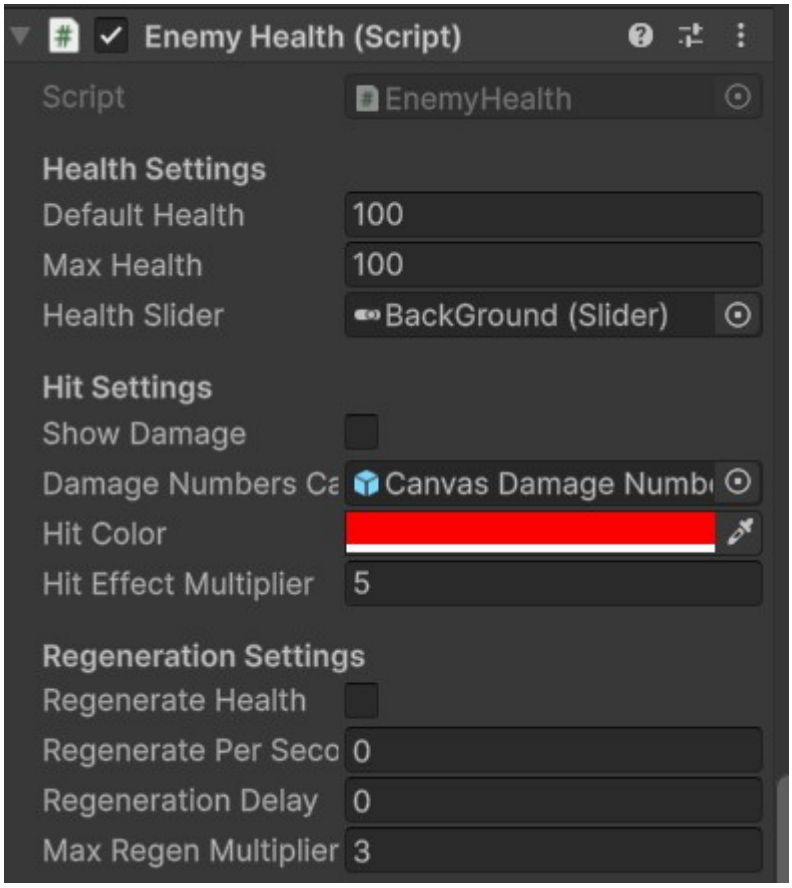
Enemies

This script defines the core behavior for all enemies in the game. It handles movement, roaming, detecting the player, attacking, and playing animations. The class is meant to be inherited, so different enemy types can extend or override specific behaviors while keeping all shared logic in one place.

Health

This script handles the health of any character or object that uses it. It keeps track of the current health, updates the health bar UI and optionally regenerates health over time. The script also includes a basic hit-flash effect and can spawn floating damage numbers.

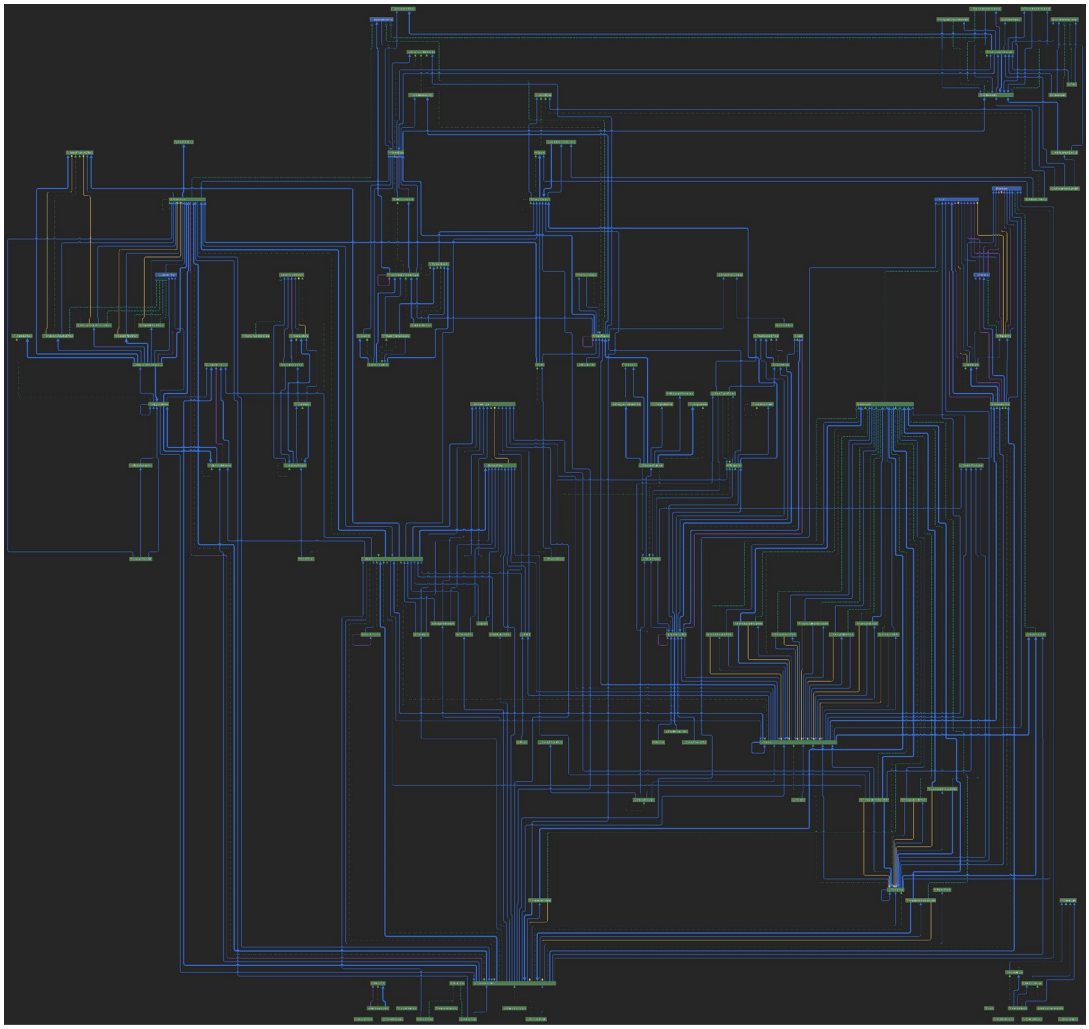
The class is designed to be inherited, so different characters can override specific behaviors especially what happens when health reaches zero. For example, enemies can play a death animation, players can trigger a respawn, and objects can simply disappear. All shared health logic stays in this base class, while special behavior is added in child classes.



Code organization overview

To easily understand our code structure and the dependencies we have made a code diagram with a hierarchical structure from bottom to top showing how the code is interacting one with another.

For easier access to said script structure there will be a folder called “Code Structure” in the project “Assets” folder where there will be multiple ways to access the shows Code Diagram.

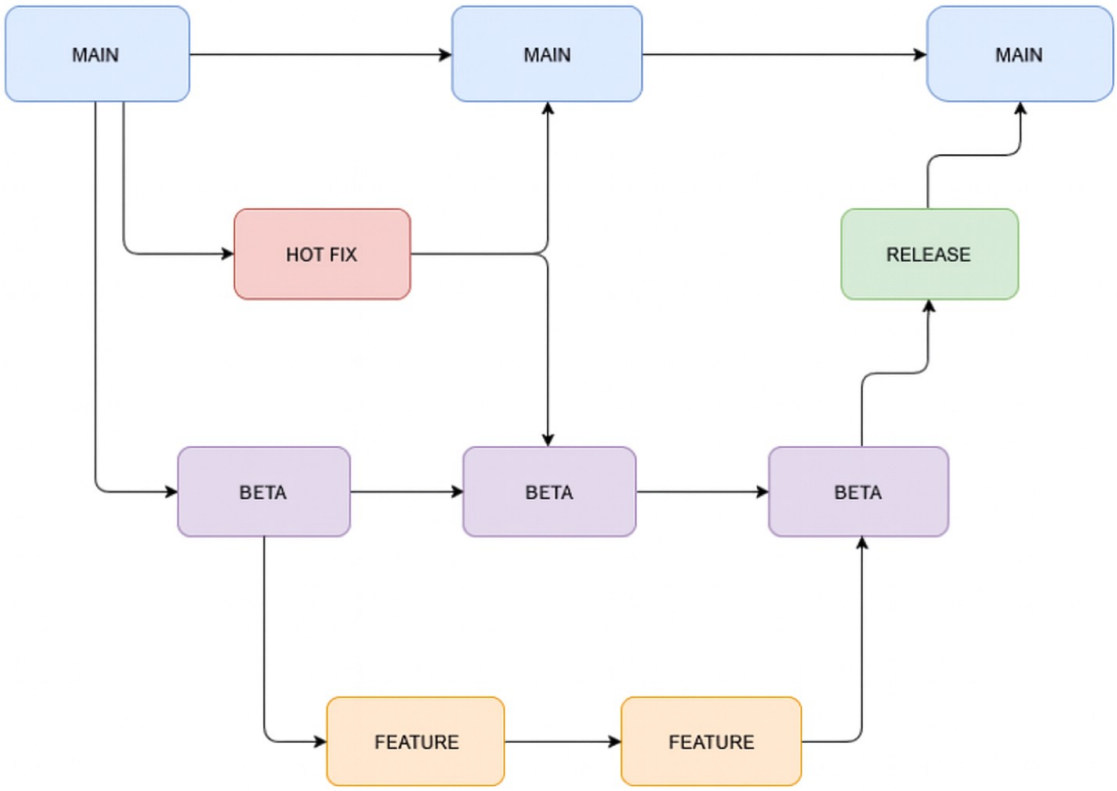


Version Control & Workflow

We use GitHub for version control, with our repository hosted in the GameLab organization. To keep the workflow accessible for everyone, we start with GitHub Desktop, which provides a clear and beginner-friendly interface. As team members gain more experience, they are free to use other tools such as Fork or GitKraken.

We've also created a Git rules document that outlines our workflow standards. All changes go through pull requests, ensuring that teammates can review and approve work before anything is merged into the main branch.

Branching Policy



For this project we have set particular conditions to naming conventions in terms of creating branches and also a branching structure which we will cover in this point:

Core branches

Branch	Purpose	When to use
main	Stable branch – Production-ready code only (e.g., public release versions, milestones)	When deploying or tagging final versions.
beta	Integration branch – Reflects the latest state of development; used for integrating features and bug fixes.	Where most development happens.

These are the core branches we will be using for our Github workflow. We will use 2 branches which fits with the scale of our project. We have the main branch which will have significant milestones, public release versions. The develop branch will reflect the latest state of development including new features and bug fixes. This ensures that while the main branch remains reliable for production, ongoing development continues smoothly in the develop branch.

Supporting branches

Branch Type	Naming Convention	Purpose
Feature	feature-[feature-name]	Developing new features or mechanics
Art	art-[art-name]	Art related development
Design	design-[design-name]	Design related development
Bugfix	bugfix-[bug-description]	Fixing bugs before a release is finalized
Hotfix	hotfix-[hotfix-description]	Critical fixes to main (e.g., post-release bug)
Release	release-[version-number]	Final preparation for a version release (e.g., bug testing & polishing)

These are the branch types we will be using for development. The first three branches (Feature, Art and Design) are created when we want to develop in the respective type. Those branches are created from the latest develop branch and should be short and concise in their reason.

Bad example: feature-characterMovement

Good examples: feature-characterMovement-and-jumping

In this case both character movement and character jumping should be two different branches.

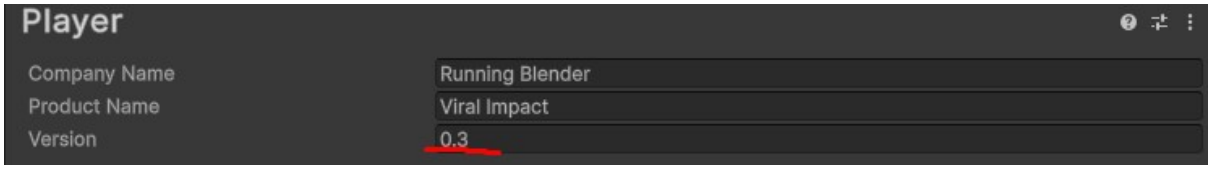
From the latest develop branch we can make a Release branch where final preparations are being made for a version. This includes testing bugs, errors, interactions and polishing before actually releasing the version. If a bug arises we create a Bugfix branch.

Bugfix would be a small branch which derives from the Release branch. This one has the purpose of applying fixes to bugs and problem found while
testing on the Release branch.

Hotfix is a crucial branch derived from main. The purpose of this branch is to fix bugs and errors that have been found post-release.

Build Delivery Method

For our build delivery method, each time a new build is made with the purpose of testing with the target audience the version number should be increased.



Also because of saving and loading sometimes some PC's would have a warning if they have downloaded our games from our Itchio page we can use the official microsoft file submission for malware analysis here which can solve this problem since the windows defender error strongly implies to the target audience to not open the file therefore we cannot test our project.

Current Build Size is ~550Mb.

Version List

Date-based versioning: https://launchdarkly.com/blog/software-release-versioning/#:~:text=forward%20to%20enhancements.-,Date%2DBased%20Versioning,-On%20the%20other

Version 0.5 (v0.5):

2025/06/26

Added Game Mechanics and Elements and descriptions of each one.

Version 0.4 (v0.4):

2025/06/24

Added minimum device requirements. Wrote Build Delivery Method. Added Code Architecture diagram.

Version 0.3 (v0.3):

2025/03/13

Wrote down the Branching Policy in the Version Control & Workflow point.

Version 0.2 (v0.2):

2025/03/11

Made the Introduction point. Added Insight to the Choice of game engine point. Added the Team logo.

Version 0.1 (v0.1):

2025/02/18

Created a template of the Technical Design Document. Added a branching policy diagram.