Programming Documentation for Project Twitch V3

Purpose

The purpose of this documentation is not to explain all of the minute details and intricacies of the code presented in the game. If programmers wanted to look at details regarding individual scripts, they could just look at the source code themselves and read the comments regarding the code. The purpose of this documentation is to provide a much bigger picture of how the scripts work together and why the larger system was designed that way.

Unity Version and Repo Structure

The git repository can be found with the following link:

https://github.com/ZauniteKoopa/ProjectTwitchV3

The Unity project was built using Unity version 2020.3.25f1. The repo imitates a typical Unity project repository with an additional "Documentation" folder. All assets can be found within the "Assets" folder. All assets are organized in categories. For the purpose of this documentation, here are the folders of interest:

- **Scripts:** Holds all C# scripts of the game
- **Prefabs:** Holds all prefabs that will be used within the game. Designers will use prefabs as a way to make the game. Programmers will use C# code as a way to make the prefabs
- **Resources/DesignerCSVs:** Folder to hold all designer excel sheets in the form of CSVs. These CSVs allow designers to make easy edits to constant values without having to look at code. Additionally, excel sheets provides math functionality to help calculate values

Scripts are organized categorically as well with the following folders:

- **AttackBehavs:** Folder that holds all information regarding general attack behaviors shared between enemies and players (StraightLineProjectile, Damage Zones, slow zones, etc)
- **Enemies:** All scripts concerning enemies
- **Interface:** All global interfaces
- **Player Package:** Most of the scripts involving the PlayerPackage
- **PoisonVial:** All information regarding PoisonVials, crafting, and handling of Poison Vials (Inventory)
- **Stage Elements:** Scripts regarding elements that can be interacted with in the world
- **UI:** Scripts regarding UI elements

- **VisualEffects:** Scripts regarding visual effects that enhance attacks but have no interaction with the world

Test Scenes

When working on this project, coders will use the TestScene framework to avoid making merge errors within the project. Within this framework, all contributors will have a test scene with the following naming convention: “TestScene_<your name>”. Within this test scene you can make as many changes as you want, including the structure of your scene to make test cases and added elements to prefabs. You only override the prefab when you merge changes into the master repo.

In this framework, only the programmer assigned can edit and work in their test scene. Everyone else is prohibited from going into their test scene to prevent merge errors. If you must go into someone else’s test scene, DO NOT save any changes you made to that test scene.

Collision Layers

The project makes massive use of collision layers to make processing of collisions easier. Here are the collision layers used within the game

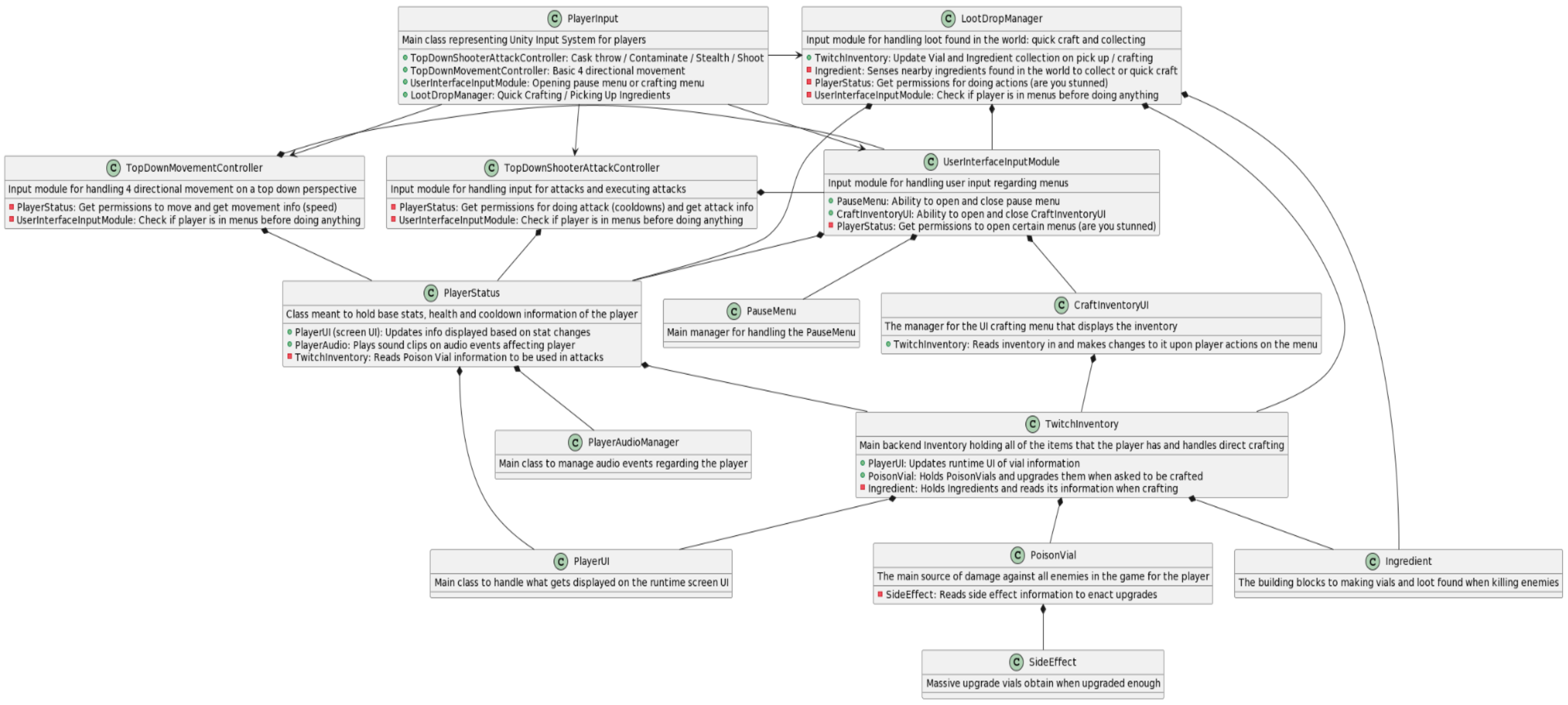
- **Loot:** Represents all loot in the game. Only collides with LootSensors
- **LootSensor:** Represents objects meant to sense loots, only collides with loot
- **EnemySensing:** Represents objects used for enemies to sense the player: only collides with PlayerHurtboxes
- **EnemyHitboxes:** Collision for enemy attacks. Only collides with SolidEnviornment and PlayerHurtboxes
- **EnemyHurtboxes:** Represents where an enemy can be hit. Only collides with PlayerHitboxes and SolidEnviornment
- **PlayerHitboxes:** Represents player attacks. Only collides with SolidEnviornment and EnemyHurtboxes
- **PlayerHurtboxes:** Represents where a player can be hit. Only collides with SolidEnviornment and EnemyHitboxes
- **CollisionSensors:** Represents sensors for possible collisions. Only collides with SolidEnviornment
- **UI:** Represents UI elements found within the game. Doesn’t collide with anything
- **SolidEnviornment:** Represents the solid walls found within the world. Collides with all hitboxes, hurtboxes, and collision sensors.

Player Package

The PlayerPackage is a prefab used by the designers to represent the player in a given level. When playing the game, the player package is what the player controls. As such, the player package is responsible for handling user input, manipulating the player character using that input, and displaying accurate, realtime information to the player through various UI elements and menus. The player package handles the following actions:

- Moving in 4 directions
- Shooting a bullet
- Throwing a poisonous cask
- Contaminate ability - apply burst damage to infected enemies around you
- Camouflaging
- Quick Crafting
- Collecting Ingredients in the world
- Opening the pause menu
- Opening the inventory

The Player Package is a very complicated player system that uses multiple classes, modularized so that each individual class only fulfills one general purpose. This package can be seen within the PlayerPackage and PoisonVial folder under Assets/Scripts. To create a simplified view of the larger player package system as a whole. A more high quality version can be found in the Documentation folder in the repository:



C PlayerInput
Main class representing Unity Input System for players
TopDownShooterAttackController: Cask throw / Contaminate / Stealth / Shoot
TopDownMovementController: Basic 4 directional movement
UserInterfaceInputModule: Opening pause menu or crafting menu
LootDropManager: Quick Crafting / Picking Up Ingredients
C LootDropManager
Input module for handling loot found in the world: quick craft and collecting
TwitchInventory: Update Vial and Ingredient collection on pick up / crafting
Ingredient: Senses nearby ingredients found in the world to collect or quick craft
PlayerStatus: Get permissions for doing actions (are you stunned)
UserInterfaceInputModule: Check if player is in menus before doing anything
C TopDownMovementController
Input module for handling 4 directional movement on a top down perspective
PlayerStatus: Get permissions to move and get movement info (speed)
UserInterfaceInputModule: Check if player is in menus before doing anything
C TopDownShooterAttackController
Input module for handling input for attacks and executing attacks
PlayerStatus: Get permissions for doing attack (cooldowns) and get attack info
UserInterfaceInputModule: Check if player is in menus before doing anything
C UserInterfaceInputModule
Input module for handling user input regarding menus
PauseMenu: Ability to open and close pause menu
CraftInventoryUI: Ability to open and close CraftInventoryUI
PlayerStatus: Get permissions to open certain menus (are you stunned)
C PlayerStatus
Class meant to hold base stats, health and cooldown information of the player
PlayerUI (screen UI): Updates info displayed based on stat changes
PlayerAudio: Plays sound clips on audio events affecting player
TwitchInventory: Reads Poison Vial information to be used in attacks
C PauseMenu
Main manager for handling the PauseMenu
C CraftInventoryUI
The manager for the UI crafting menu that displays the inventory
TwitchInventory: Reads inventory in and makes changes to it upon player actions on the menu
C PlayerAudioManager
Main class to manage audio events regarding the player
C TwitchInventory
Main backend inventory holding all of the items that the player has and handles direct crafting
PlayerUI: Updates runtime UI of vial information
PoisonVial: Holds PoisonVials and upgrades them when asked to be crafted
Ingredient: Holds Ingredients and reads its information when crafting
C PlayerUI
Main class to handle what gets displayed on the runtime screen UI
C PoisonVial
The main source of damage against all enemies in the game for the player
SideEffect: Reads side effect information to enact upgrades
C Ingredient
The building blocks to making vials and loot found when killing enemies
C SideEffect
Massive upgrade vials obtain when upgraded enough

Keep in mind that the diagram doesn't include all classes the Player Package embodies. Many of the classes seen in the diagram have minor classes within them as components that help with the general purpose of the parent class. For example, the TopDownShooterAttackController has an AimAssist component to help better aim bullet shots when executing the primary shooting attack. Additionally, the CraftInventoryUI class has multiple UI elements such as IngredientIcons, VialIcons, and ResourceBars to build the multiple UI elements. The classes seen in the diagram were chosen because they had important relationships with other classes. The goal of the diagram is to educate future programmers about the overall structure of the code without bogging them down with multiple classes.

For each class within the UML diagram, there will be a small description at the very top that indicates what the purpose of the class is. A list of relationships can be found under that description, usually indicated by composition. If you see a red dot next to a relationship, that indicates that the relationship is read-only. If you see a green dot, that means the relationship is read/write.

The best way to explain the PlayerPackage is to start with the beginning and the very top of the diagram, PlayerInput. The PlayerInput is a builtin class from Unity's Input System that handles user input for the entire code. This PlayerInput will trigger events once certain mapped keys have been pressed. These events will trickle down to the classes under it. The classes that PlayerInput sends events to are 4 User Input Modules:

- **Top Down Movement Controller**: Handles four directional movement on a top down plane
- **Top Down Shooter Attack Controller:** Handles ability usage and basic attacks made by the player character
- **Loot Drop Manager:** Handles collecting and quick crafting using ingredients found in the in-game world
- **User Interface Input Module:** Handles menus such as the PauseMenu and the Craft Inventory menu

Within these user input modules, the modules will ask permissions to certain classes to make sure they can execute the action, generally the PlayerStatus to check if the player can move on that frame and the UserInterfaceInputModule to check if the player is in a menu. Once the input modules have permissions to do the action, they can either act outward on the world (moving the player character or creating attacks) or they can interact with other components of the player package such as the inventory or the various UI menus.

The root of all interactions in the game are found in 2 classes within the PlayerPackage: Player Status and TwitchInventory. PlayerStatus is the function that holds all of the base stats of the player character and provides an interface for InputModules to access information such as movement speed, armor, and health. As such, this class is also responsible for handling changes to these stats such as status effects like slows and stuns, damage to health, and death / respawn.

The TwitchInventory holds all of the ingredients and poison vials that the player has. These Poison Vials are important because they are the source of all damage that the player can inflict. As such, Poison Vials are used immensely in attack properties. The inventory is also responsible for handling upgrading the vials.

PoisonVials generally have 4 stats associated with them with the following properties:

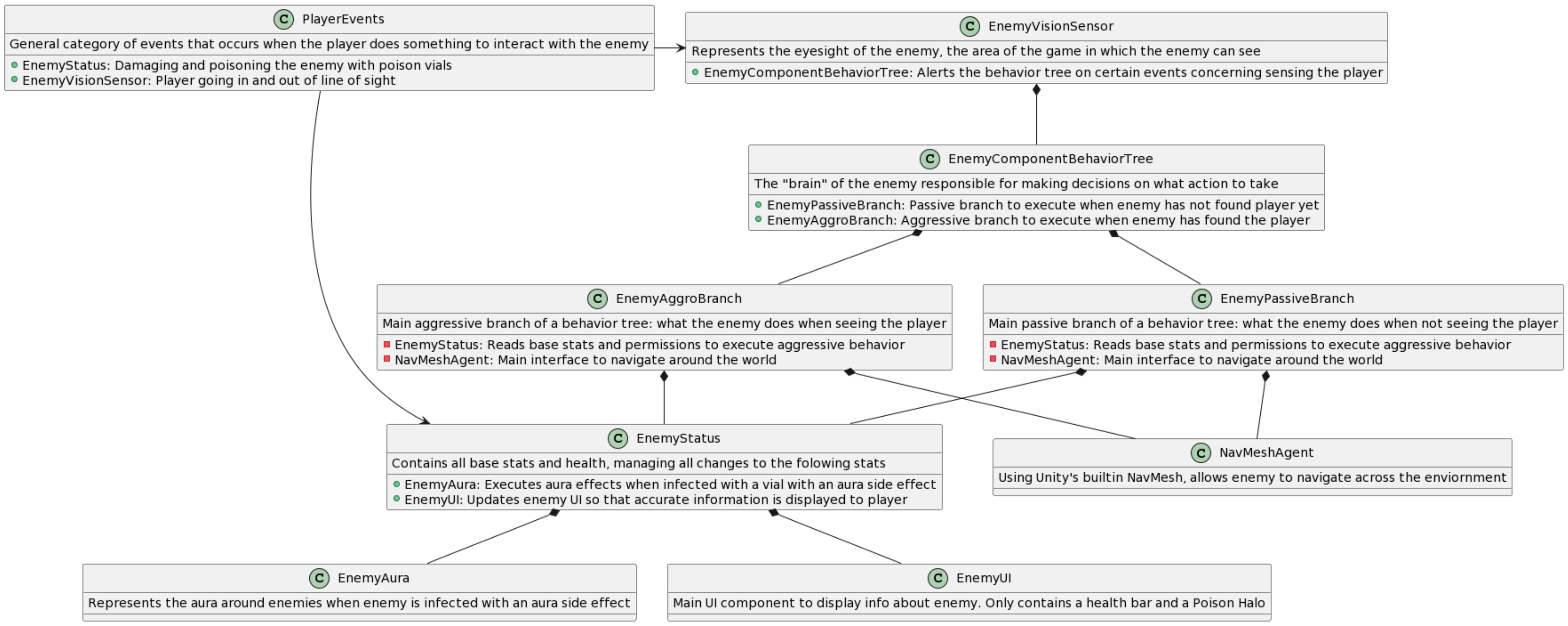
- **Potency:** The immediate damage of a bullet
- **Poison:** The damage over time the enemy feels when infected with a poison. Unaffected by armor
- **Reactivity:** Ability damage (specifically contaminate)
- **Stickiness:** The CC ability of passive poison stacks and casks

Vials can be upgraded using 1 or 2 ingredients. Each ingredient contributes 2 stats depending on the ingredient's properties, up to a max stat cap of 10. Each stat category has a max stat of 5. If a stat category reaches a value of 3 or more, the vial will receive a side effect with a specialization in that given stat category. These side effects are special, super-powered buffs to Twitch's kit that can give him a special edge. For example, a potency upgrade can give him piercing bullets, a reactive upgrade can make the contaminate ability explode, dealing AoE damage around infected enemies, a poison upgrade can make damage over time rate go faster, etc.

Side effects are implemented using a virtual parent class with a specialization of "NONE". Using this virtual class, a child of the class only has to override the functions that it needs to make the effect happen and change its specialization. However, one flaw with this design choice is the need to augment seemingly random places of the player package or player attacks with calls to these side effects to get the intended effect. Additionally, in-game, it leads to unnecessary checking of the side effects in places where the side effect shouldn't apply.

Enemies

Enemies within this game are a lot more straightforward than the PlayerPackage. The series of classes that compose a single enemy can be seen in the following diagram:


PlayerEvents
General category of events that occurs when the player does something to interact with the enemy
EnemyStatus: Damaging and poisoning the enemy with poison vials
EnemyVisionSensor: Player going in and out of line of sight
EnemyVisionSensor
Represents the eyesight of the enemy, the area of the game in which the enemy can see
EnemyComponentBehaviorTree: Alerts the behavior tree on certain events concerning sensing the player
EnemyComponentBehaviorTree
The "brain" of the enemy responsible for making decisions on what action to take
EnemyPassiveBranch: Passive branch to execute when enemy has not found player yet
EnemyAggroBranch: Aggressive branch to execute when enemy has found the player
EnemyAggroBranch
Main aggressive branch of a behavior tree: what the enemy does when seeing the player
EnemyStatus: Reads base stats and permissions to execute aggressive behavior
NavMeshAgent: Main interface to navigate around the world
EnemyPassiveBranch
Main passive branch of a behavior tree: what the enemy does when not seeing the player
EnemyStatus: Reads base stats and permissions to execute aggressive behavior
NavMeshAgent: Main interface to navigate around the world
EnemyStatus
Contains all base stats and health, managing all changes to the folowing stats
EnemyAura: Executes aura effects when infected with a vial with an aura side effect
EnemyUI: Updates enemy UI so that accurate information is displayed to player
NavMeshAgent
Using Unity's builtin NavMesh, allows enemy to navigate across the enviornment
EnemyAura
Represents the aura around enemies when enemy is infected with an aura side effect
EnemyUI
Main UI component to display info about enemy. Only contains a health bar and a Poison Halo

In general, enemies are ultimately controlled by events created by the player. These events include damaging the enemy with abilities and primary fire and being within enemy line of sight. Upon receiving these events, an enemy's stats and behaviors are subject to change.

Enemy stats work very similarly to the PlayerStatus in which it is used to manage base stats and health while also providing an interface to access that information without worrying about stat changes. Changes to these stats incurred by enemy behavior or the player will change the enemy UI accordingly and any possible Poison Vial side effects.

The AI makes decisions based on a very scalable component based behavior tree. The behavior tree is framed as a very simple statement: if the enemy can see the player, act aggressive towards that enemy, otherwise, go back to a passive state. The act of switching between these branches is handled by the BehaviorTree class and the switching is often triggered by events broadcasted by the EnemyVisionSensor.

In order to make this scalable, the EnemyComponentBehaviorTree uses interfaces to communicate with EnemyAggroBranch and EnemyPassiveBranch, meaning that any script can fill in for these 2 branches as long as it fulfills the interface requirements. This allows for code to be reused in multiple different ways, allowing enemies to have similar passive branches and similar aggressive branches. These branches will then read stat information and permissions from EnemyStatus and navigate around the map using the NavMeshAgent. In the behavior tree, only 1 branch can be active at any given time.

Stage Elements

Interactable Stage elements can be found holding an IHittable class, an interface that indicates an object is interactable if you fire a bullet at it. Within the IHittable class is an event that indicates whether or not the object is destroyed. This is useful for handling interactables that interact with player collisions so that when the object is destroyed, the player's collision sensors can be handled appropriately.

Within an IHittable object, there are 2 necessary functions: reset() and hit(). Reset() is called when the game wants to reset that object to its initial state and hit() is called when the player hits the interactable object.

In the game right now, there are 3 types of IHittable objects:

- **BreakableCrates:** Crates that are breakable by the player. Blocks vision and doesn't give anything in return
- **IngredientCrates:** Crates that, when broken, return 2 ingredients that can be used to craft Poisons. 1 can be found in each checkpoint
- **ActivatableSwitch:** A switch that can be activated to do something. If you look at this component in the inspector, it has events for activation and reset.

Enemy Groups and Checkpoints

To better handle death, players will be given checkpoints throughout the game so that they have a place to go to after they die. Checkpoints are easily triggered by manipulating the collider of an object that has the Checkpoint component. However, checkpoints also need to be able to reset previously done enemies. In order to do this, the game makes use of EnemyRooms that are resettable. However, not all checkpoints need an EnemyRoom and all player packages need to be assigned to one checkpoint so that they can die appropriately.

EnemyRooms are simply groups of enemies and interactable stage elements that can be reset en masse. These elements are typically grouped by room so that designers can keep what's being reset organized. Additionally, when killing all enemies in the room, it can trigger an event that can lead to revealing chests and unlocking doors.