Software Design Document

Version 2.0

1-3-24

Project Sponsor: Dr. Ashish Amresh

Faculty Member: Tayyaba Shaheen

Team Members:

Rain Bigsby, Veronica Cardenas,

Ethan Ikhifa, Lenin Valdivia

This document discusses the overall architectural design of a burst game framework. It covers important modules and components required for a video game based on any medical subject of a developers choosing.

Table of Contents

1.0 INTRODUCTION 3
2.0 IMPLEMENTATION OVERVIEW 4
2.1 Solution Vision 4
2.2 Technologies 4
3.0 ARCHITECTURAL OVERVIEW 6
4.0 MODULE AND INTERFACE DESCRIPTIONS 10
4.1 Main Menu Component 10
4.2 Gameplay Mechanics Component 11
4.3 Characters 12
4.4 Environment 14
4.5 Level Manager 15
4.6 Cameras 16
4.7 Statistics 18
6.0 CONCLUSION 21

1.0 INTRODUCTION

COVID-19 vaccination could play a critical role in not only improving the symptoms, but preventing hospitalizations and deaths related to such a disease as well, ending the pandemic situation. Yet, despite the possibility of such benefits, as of May 3, 2023, about 8.3 million U.S. children aged 12-17 and 17.4 million children aged 5-11 had yet to receive their first COVID-19 vaccine dose [2]. Over 60% of people under 20 years of age had yet to receive a single dose of the vaccine in Arizona [3]. Now, even with the apparent need to promote vaccine uptake among adolescents, there are currently no intervention studies underway aimed at improving these vaccination rates.

Indeed, seeing such a gap within an egregious issue, with a background in video game development, Dr. Ashish Amresh aims to fill this gap with video games directed at the adolescent. We are designing a content agnostic game development framework that can be used to rapidly make games for different domains, such as COVID-19 and HPV. Now, to increase the likelihood of improving vaccination rates with such an approach, we are designing within the context of clinics where teens and their parents frequent and are waiting for their appointments. And within such a context with limited interaction time, the games must therefore be fast-paced, fun, and at the same time provide a range of decision-making choices to fully engage the adolescents in addressing the outcomes. We will refer to the concept of fast-paced and engaging games as “burst games;” how our framework will focus on making games consisting of repetitive, quick “bursts” of gameplay that promotes learning by doing while minimizing the cost of failure and frustration for the player. Some examples of commercial burst games that we may follow are Angry Birds, and Puzzle Fighter.

This document will discuss the implementation of our burst game framework, what modules will be created during development, and examine the overall architecture of the framework. The purpose for this document is to review architectural and module designs to outline the efficacy of our product. Our goal is to achieve maximum software reusability for future developers in the context of burst games within a clinical setting.

2.0 IMPLEMENTATION OVERVIEW

2.1 Solution Vision

To develop a content-agnostic video game framework, our target is to maximize software reusability with the ability to customize features of the framework according to the future developers requirements. Some features that will be implemented into our framework include placeholder knowledge-drop components, a game structure template, and extendable classes and objects. These features allow a developer to implement their own specifications related to their medical subject. Our framework's effectiveness will be demonstrated by creating two burst video games of our own focusing on the use cases of HPV and COVID-19 vaccinations. The main objective that Medical Gaming Solutions hopes to achieve is to create a video game framework tailored to fit any medical subject that will expedite development and save resources for developers.

2.2 Technologies

The framework will be developed using C# scripts in the Unity game engine, and will allow users to select given components to build the game genre of their choice. They can manage the interactions between components and modify the effects by editing the C# scripts. Itch.io is our chosen web application to host the finished project on. Games built from the framework can be easily exported and uploaded to the Itch.io servers and made public for others to view and play. The framework will support WebGL builds to allow mobile devices to run the games. This also avoids having the user forced to wait for an installation to finish, and can instead play the game on the web browser itself.

- **C#:** To write the scripts for our components, our team chose to use C# as it is the base language used within Unity. The team mostly has familiarity with languages such as C or C++, making this language easy to understand and write with. It is an object oriented programming language, adding reliability to the architecture of the framework as we develop.

- **Unity:** The overall framework will be developed within the Unity game engine. It is one of the most popular and easy to use engines for game development. The hierarchy window works as a convenient method for establishing and organizing components, and will allow developers to manage the

components they desire for their project. Components can be easily dragged and dropped into scenes and interact with each other via their defined functionalities.

- **WebGl:** The use cases for this framework must be compatible with web browsers as a host. This fulfills the task of making the game accessible in a clinical setting via a tablet. This route avoids intensive downloads and allows developers to easily make an executable after production.

3.0 ARCHITECTURAL OVERVIEW

3.0 Architectural Overview

So the architectural design of our Unity project centers around two main components so far: the Main Menu Component and the Gameplay Mechanics Module. In the next section I will go into depth on these components, outlining their roles, communication mechanisms, and architectural influences on our project thus far..

Discussion of Architecture

(a) Key Responsibilities and Features of Each Component:

3.1 Main Menu Component

Main Menu Component: The Main Menu Component acts as the main interface for user interaction, allowing familiar navigation options also while having clean transitions between different game states and menu options. It is responsible for initiating gameplay sessions, managing user preferences, and providing access to additional functionalities such as help documentation.

3.2 Gameplay Mechanics Module

Gameplay Mechanics Module: The Gameplay Mechanics Module outlines the essential game logic, conducting player actions, environmental interactions, and scoring mechanisms like a heads up display, health bar, or even a score counter. It controls the behavior of in-game entities, rule sets, to ensure a coherent and engaging gameplay experience.

3.3 Characters

Characters Module: Streamlines the process of adding characters to the game, handling controllers, character augmentation, collision handling, and AI scripts.

3.4 Environment

Environment Module: Provides basic objects to fill levels and make them interactive, including platforms, obstacles, and jump pads.

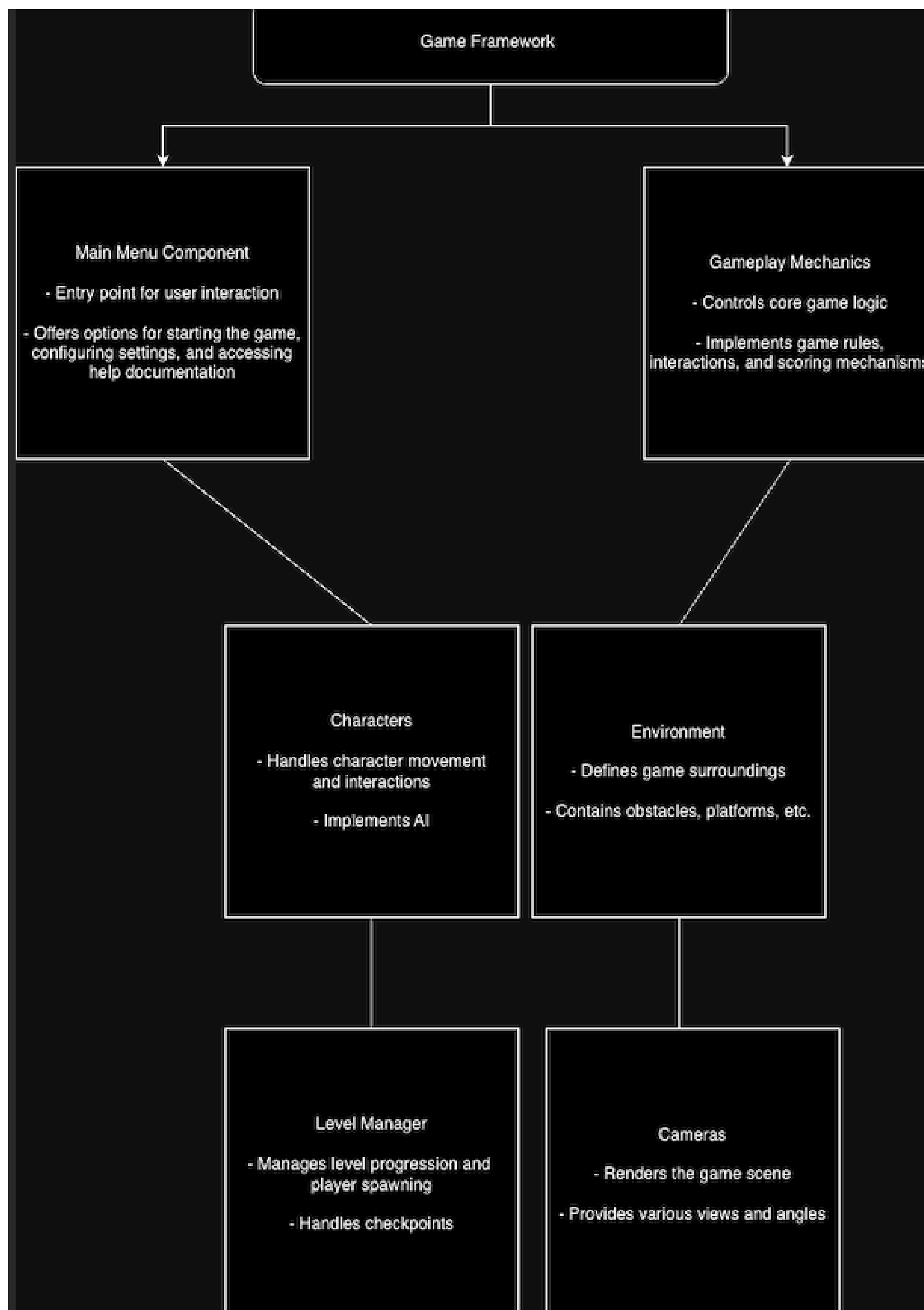
3.5 Level Manager

Level Manager Module: Manages game sessions, including spawning players, determining boundaries, managing checkpoints, and handling respawning.

3.6 Cameras

Cameras Module: Renders the scene and provides different views to the player, including UI cameras for displaying game information and main cameras for rendering the gameplay environment.

3.3 Architectural Diagram



Here in the architectural diagram, we will go over the system's high-level architecture starting with the central role of the Main Menu Component and the connected nature of the Gameplay Mechanics Module within our Unity project. The Main Menu Component is the entry point for users, offering options for starting the game, settings configuration, and of course exiting/closing the game. Aside from the Main Menu Component is the Gameplay Mechanics Module, which controls core game logic such as the player movement, object interactions, and scoring mechanisms/ UI Displays. The diagram highlights the interaction between these 2 components, showing how they help deliver a short but fun gaming experience.

(b) Main Communication Mechanisms and Information/Control Flows:

When a user interacts with the Main Menu Component, it's not just about starting the game or adjusting settings. These interactions trigger a series of events that make its way through the entire game system via Unity's event system. For example, clicking the "Play" button doesn't just launch the game - it sets off a chain reaction that initializes the characters, sets up the environment, and positions the cameras for the player's perspective.

During gameplay, the Gameplay Mechanics Module is the central hub of communication, controlling the flow of information between all components. As the player navigates through the game world, the Gameplay Mechanics Module communicates vital information back to the Main Menu Component. This includes updates on game progress, achievements unlocked, and changes in the player's score. But it's not just a one-way street. This communication flows both ways, enabling seamless transitions between different game states. For example, when the player completes a level, the Gameplay Mechanics Module notifies the Main Menu Component, which then triggers the transition to the next level or back to the main menu. This bi-directional flow of information ensures that the player's experience remains immersive and uninterrupted throughout the game. And now, with the addition of new components like Characters, Environment, Level Manager, and Cameras, this communication network becomes even more robust. Characters communicate their actions and interactions with the Gameplay Mechanics Module, while the Environment module provides feedback on the player's surroundings. The Level Manager deals with the progression of the game, while Cameras capture the game from different angles. Together, these components create a dynamic system where information flows freely, enabling a truly immersive gaming experience.

(c) Influences from Architectural Styles:

The architectural design takes on a more component-based approach, as making our project modular and properly encapsulated are key principles in building a framework while also promoting code reusability and maintainability as mentioned by our mentor. By abstracting not only the main menu component and the

Gameplay Mechanics Module but also the new components such as Characters, Environment, Level Manager, and Cameras, the design is scalable across different game projects and ideas. Player focused design principles are shown throughout the architectural layout, prioritizing navigation, visual feedback, and help carry out user interactions. Our architecture is intended to enhance user engagement and enjoyment by providing a fun yet educational gaming experience. This approach ensures that each component works harmoniously to deliver a cohesive and immersive gameplay experience for the players.

3.4 Focusing on a Game Framework

It's also important to note that our project is not focused on building a stand alone game but rather to develop a scalable game framework similar to a toolbox of components available on the Unity asset store. Doing this helps developers and future clients/developers to leverage our modular architecture to make their own prototypes, customize them as they see fit, and use them in a diverse range of their own games. Our focus on our product being modular and reusable shows our vision of changing game development and giving creators the tools they need to bring their ideas to life. Through our framework, we look at our project as an innovative and creative step taken in the right direction of gaming and healthcare within the Unity game development community.

4.0 MODULE AND INTERFACE DESCRIPTIONS

Module and Interface Descriptions

The Module and Interface Descriptions section gives an overview of the Main Menu Component and the Gameplay Mechanics Module, showing their responsibilities, class diagrams, and public interfaces within the Unity project architecture we are working on.

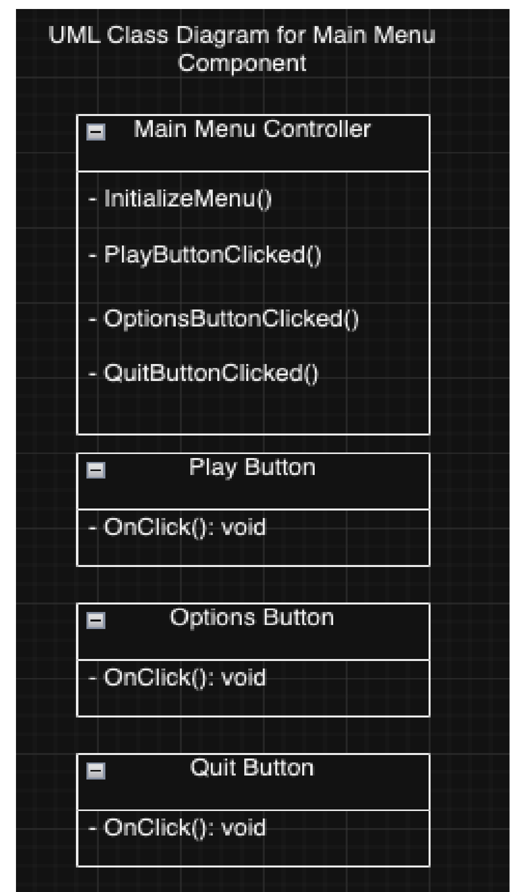
4.1 Main Menu Component

The Main Menu Component acts as the entry point for users, facilitating navigation, settings configuration, and application exit functionalities. Within the larger context of the architecture, the Main Menu Component acts as the primary interface for user interaction, seamlessly integrating with other game systems to orchestrate gameplay sessions and manage user preferences.

Responsibilities:

- showing navigation options, including Play, Options, and Quit buttons.
- Handling our user input to start gameplay sessions, configure game settings, and exit the game.
- Managing transitions between different game states, such as the main menu, options menu, and gameplay scenes.
- Providing access to other functions, help documentation, and FAQS/Credits .

UML Class Diagram:



Public Interface:

- PlayButtonClicked(): Initiates a new gameplay session.
- OptionsButtonClicked(): Displays the options menu for configuring game settings.
- QuitButtonClicked(): Exits the application.
- Gameplay Mechanics Module

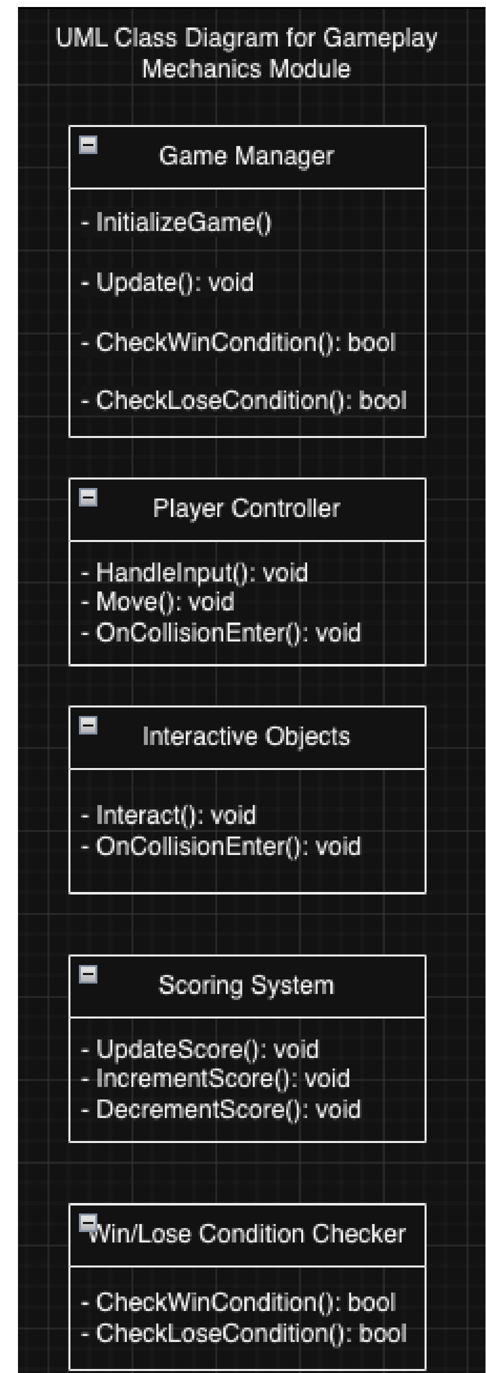
4.2 Gameplay Mechanics Component

The Gameplay Mechanics Module encapsulates core game logic, governing player actions, object interactions, and scoring mechanisms. It plays a pivotal role in orchestrating the behavior of in-game entities and enforcing rule sets to ensure a coherent and engaging gameplay experience.

Responsibilities:

- Managing player movement, object interactions, and environmental physics.
- creating game rules, including collision detection, scoring systems, and win/lose conditions.
- Facilitating real-time feedback mechanisms, such as UI updates, sound effects, and visual cues.
- Coordinating with other game systems, such as the Main Menu Component, to align game states and transitions.

UML Class Diagram:



Public Interface:

- MovePlayer(Vector3 direction): Moves the player character in the specified direction/way.
- InteractWithObject(GameObject object): Triggers interactions with interactive game objects.
- UpdateScore(int scoreDelta): Updates the player's score based on the specified delta value.
- CheckWinCondition(): Checks if the win condition of the game has been met.
- CheckLoseCondition(): Checks if the lose condition of the game has been met.

4.3 Characters

The primary function of the character module is to streamline the process of making characters move and act within the game. Indeed, from controllers, character augmentation, and collision handling, to AI scripts and item interaction, future developers will be provided with a thorough basis in adding characters to their games.

4.3.1 UML Diagrams

Figure 4.3.A - Character Game Object UML Diagram

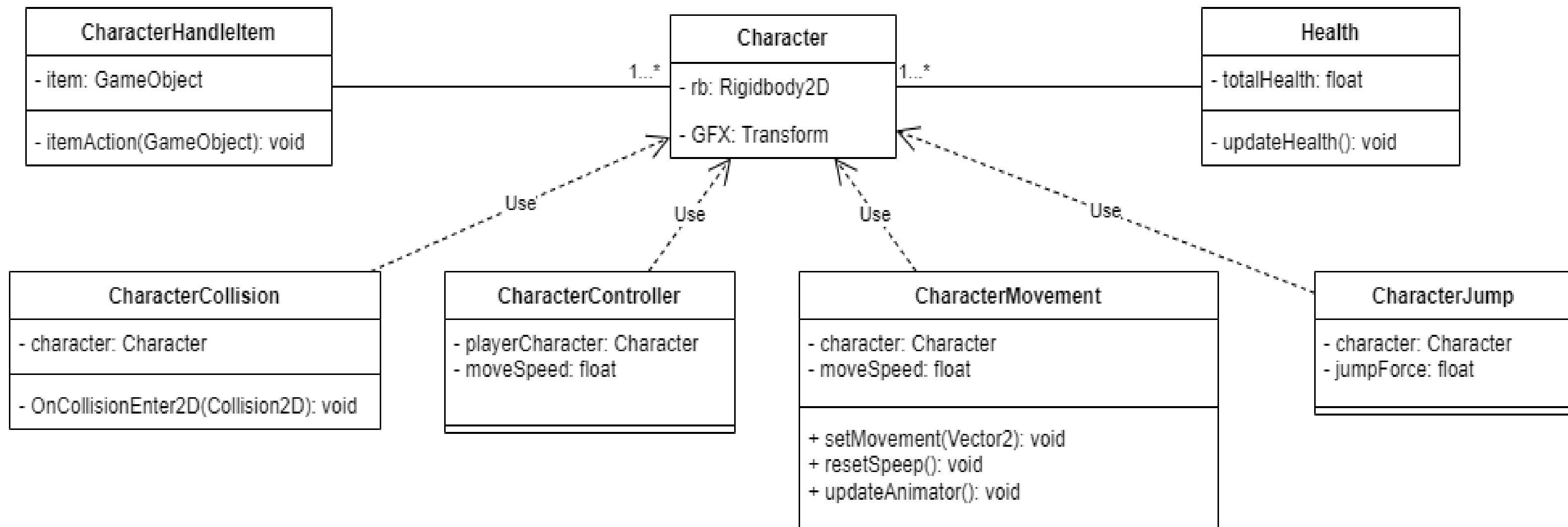


Figure 4.3.B - AI Handling UML Diagram

AIBrain
+ owner: GameObject
+ states: AIState
+ target: Transform
+ brainActive: bool
+ transitionState(string): void
+ resetBrain(): void
- findState(string): AIState
- initializeDec(): void
- initializeAct(): void
1
1...*
1...*
1...*
AIState
+ stateName: string
- brain: AIBrain
+ setBrain(AIBrain): void
+ enterState(): void
+ exitState(): void
+ performActions(): void
AIAction
+ actionName: string
- brain: AIBrain
- isInitialized: bool
+ init(): void
+ doAction(): void
AIDecision
+ decName: string
- brain: AIBrain
+ init(): void
+ decide(): void
1...*
1
AITransition
+ decision: AIDecision
+ trueState: string
+ falseState: string

Figure 4.3.A displays all of the classes involved in adding a character to the game scene. Indeed, as one can discern from the diagram, the foundation of the character module is a simple character class that references the Rigidbody and Transform components of the actual game object within the game scene representing the character. Indeed, all of the classes that encompass the physical interactions and movement of a character, such as the CharacterCollision and the CharacterMovement class, must utilize the base Character class to carry out their function. Now, while some of the classes may seem primitive, the purpose of such a primitive nature is to leave room for developers to extend said classes as they need. Now, as for figure 4.3.B, this diagram demonstrates all of the classes involved in giving autonomy to any characters within the game scene. Given how most of the classes are aggregations of the AIBrain class, it is expected that the entirety of the presented structure for AI handling should be used when future developers want to give autonomy to their characters.

4.3.2 Public Interface

The character module provides developers with a way to add characters to their games, along with giving them autonomy should they choose to. Indeed, since all games do not require some form of AI, this module is split into two components so as to enforce the idea that in-game characters can exist without autonomy; the AI handling component is merely optional. Now, for the character component, it provides the ability to add both a player-character and a non-playable-character. In particular, the differentiating factor between a player-character and a non-playable-character is the utilization of the CharacterController class: a simple class responsible for handling player input. But regardless of the playable status of a character, all characters within the game must have the following attributes: a reference to the corresponding Rigidbody, and a reference to the corresponding Transform which represents the visualization of the character. A particular class of note is the CharacterMovement class; how its public methods, setMovement, resetSpeed, and updateAnimator, directly manipulate the Rigidbody and Transform of the corresponding character game object.

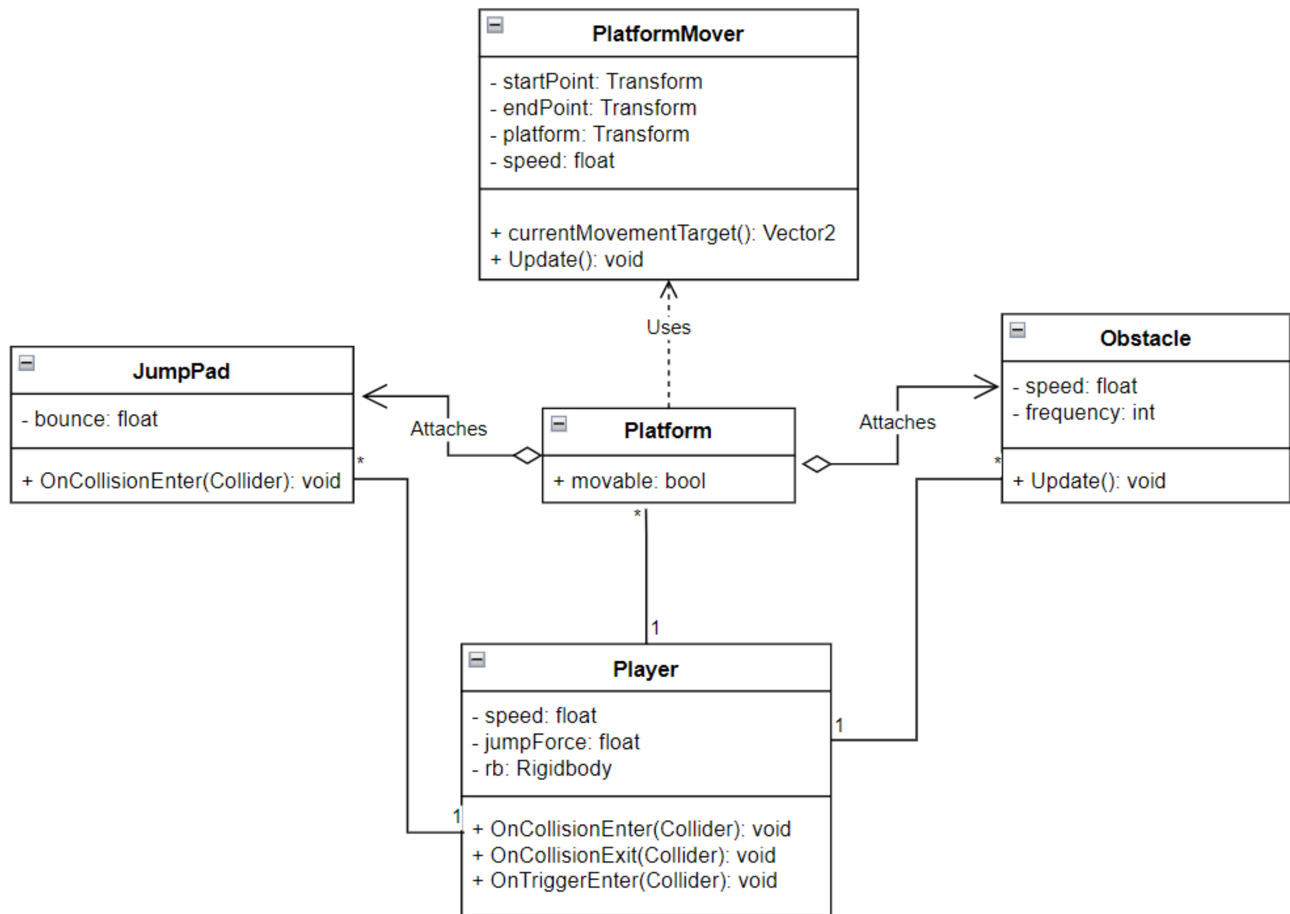
Now, if the developer chooses to add autonomy to their characters, the developer will have jurisdiction over the current state of the character, and the actions and decisions that the character may take. Indeed, as all of the aggregate classes utilize the AIBrain class to carry out their services, future developers will need to flesh out their own state, action, and decision definitions; in particular, developers will need to define their own list of action objects in order to carry out the processes of the AIBrain class.

4.4 Environment

The environment module will provide developers with all the basic objects needed to fill in levels and make them more interactive for the users. Platforms and obstacles will be included to define a base for the player to walk through, with multiple types to give variety when designing environments.

4.4.1 UML Diagrams

Figure 4.4.A - Environment Platforms UML Diagram



The diagram above demonstrates how the player will be able to interact with the environment using the objects within the scene. The main platform components will be static with the option to become movable, using the PlatformMover class. Obstacles and JumpPad classes include methods for player interaction: OnCollisionEnter to give the player velocity depending on their jumpForce, and Update to manage the performance of the game and keep the amount of Obstacles to a minimum.

4.4.2 Public Interface

The environment module defines the surroundings and a border for the area of interactivity for the player. Platforms are the base for objects to be placed upon. Obstacles can spawn on top of these to give the player a challenging level, using a set spawn point and using the speed variable to adjust how fast the obstacles travel from that spawn point. JumpPads assist the player by multiplying the jumpForce variable by the bounce variable to give them a higher velocity. The PlatformMover can also alter the main platform class to have a start and end point set on the map to travel between, along with the speed variable to set how fast the platform can move between these points.

4.5 Level Manager

The level manager handles multiple objects that may be visible to the player. This includes spawning the player, determining boundaries for characters, checkpoints, handling respawning after the player loses, and loading the environment and objects of the current level. Developers will be able to customize this class by adding or removing attributes and methods of their choice. Specifically, if they would like checkpoints, respawning delay, or if the player should respawn at all.

4.5.1 UML Diagram

Figure 4.5.A - Level Manager UML Diagram

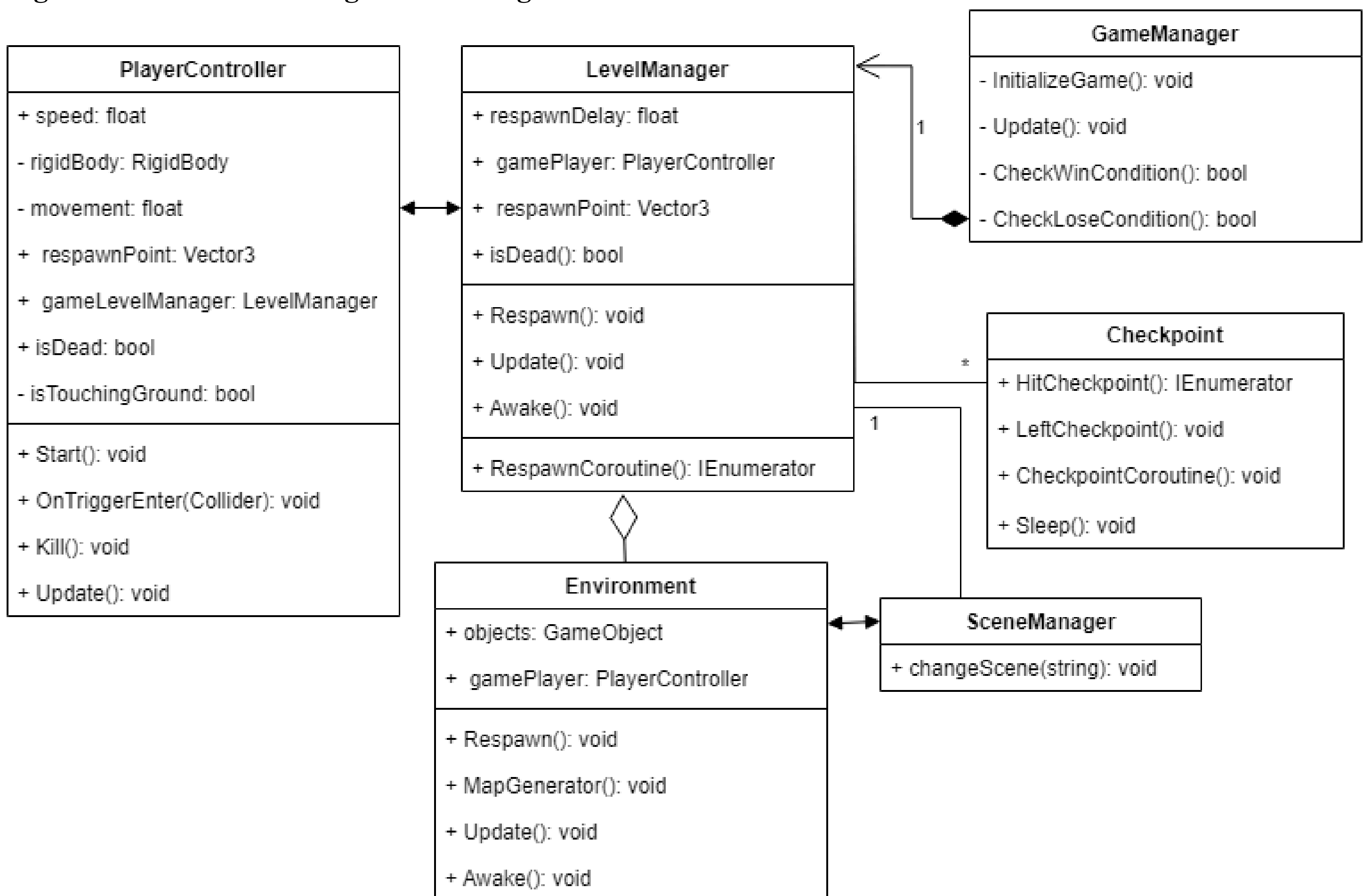


Figure 4.5.A is the current UML implementation plan for the LevelManager class. This class is required for basic game management for a level system in any type of burst game. It requires attributes that determine the boundaries for characters, but respawning is optional depending on the developer's choice. The PlayerController and GameManager classes are necessary to include in the framework architecture if a level manager is implemented due to the references that the classes make to each other. The Checkpoint class is optional and the level manager can have multiple checkpoints at the discretion of the developer. The Environment can exist independently of the level manager.

This diagram displays the classes that the LevelManger requires or offers an optional implementation. LevelManger is accessible to all other classes within the architecture. Since all classes in the framework are not concrete, the LevelManager should be open to customization and allow for code reuse.

4.5.2 Public Interface

The LevelManager class contains methods that manage spawning the player when beginning the game and after a player loses. It uses the position from the PlayerController class to determine the place where to spawn the player before they last died. The LevelManager also offers an optional method to create a respawn delay after a player loses, this could be beneficial when implementing the knowledge drop component. An important attribute within this class is the isDead boolean which is set to true when the player dies or loses a level. This is needed to reset the environment, although the environment can exist independently. Before resetting the environment, the developer could implement the optional Checkpoint class which contains a CheckpointCoroutine method. This is responsible for spawning the character at the last checkpoint in the game.

4.6 Cameras

Cameras in Unity are an important module to discuss within the architecture of a game as its main responsibility is to render the scene. This component can be either static or dynamic and Unity offers a wide range of attributes to customize cameras within the engine inspector. The two types of cameras that will be utilized within this framework are the UI camera and main camera, but developers are free to implement their own as these two cameras are a starting point for development.

4.6.1 UML Diagrams

Figure 4.6.A - UI Camera UML Diagram

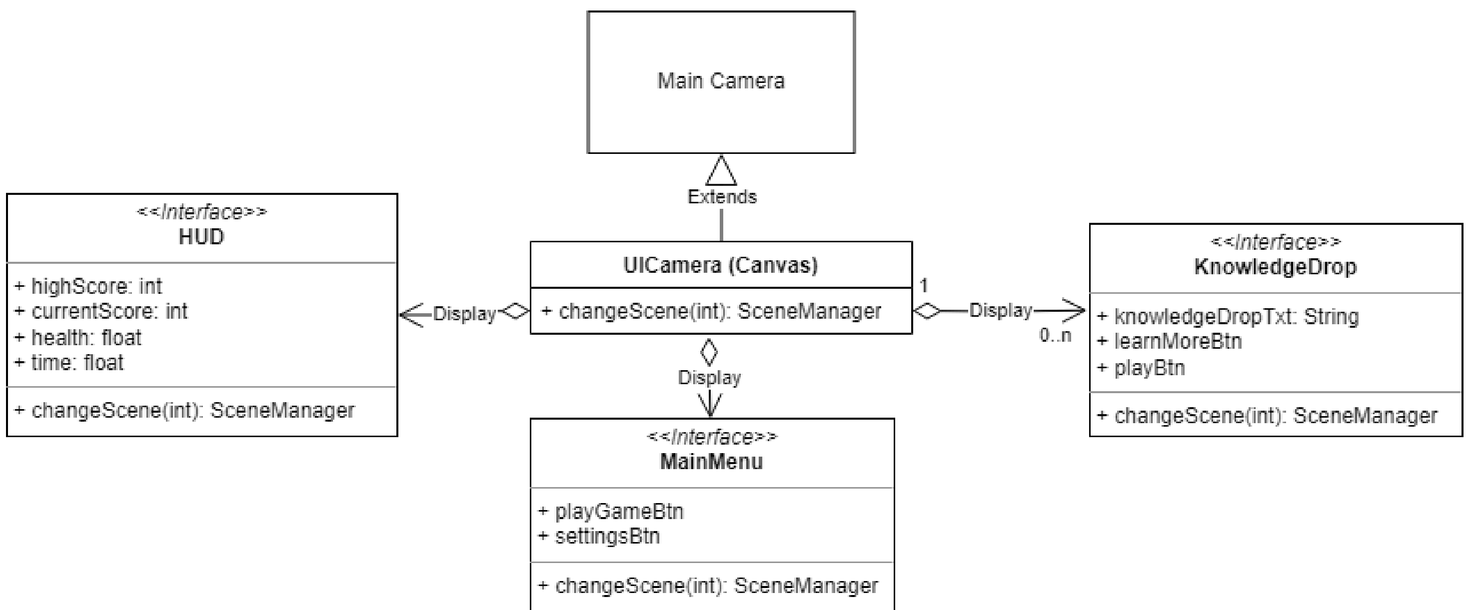


Figure 4.6.A represents the possibilities of what the UICamera could implement, as this component's functionality is left up to the future developer. The UICamera uses a canvas asset within Unity to be able to display text, buttons, and any other GUI object for the player to see. This component should be fully customizable, as different GUI is required for different medical subjects. The interface classes are representative of this customization because it is an abstract implementation.

Figure 4.6.B - Main Camera UML Diagram

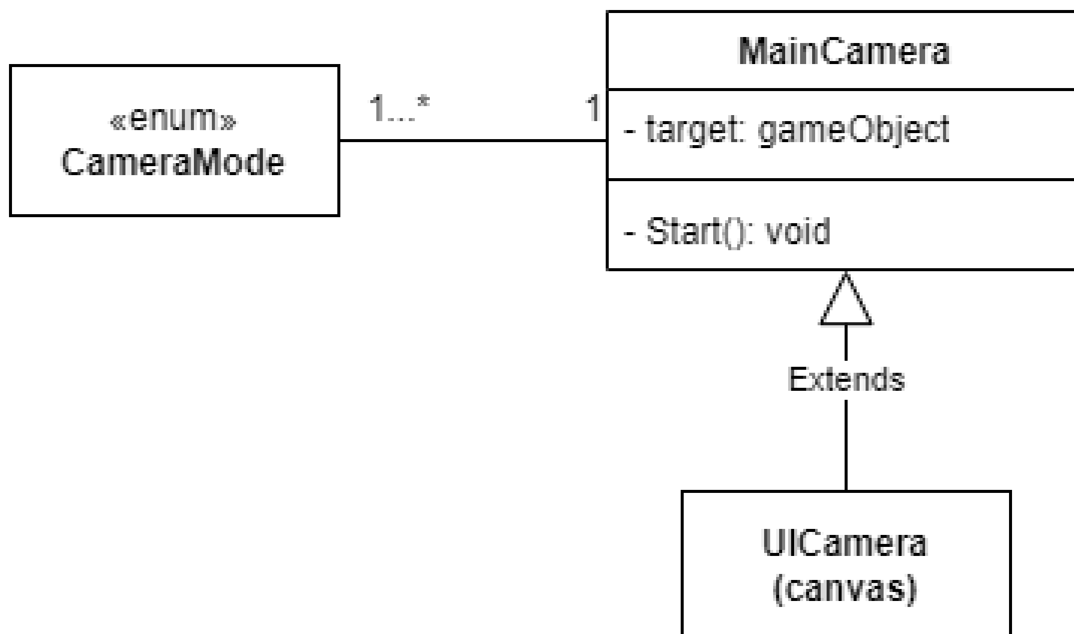


Figure 4.6.B illustrates the basic functionality of the main camera in which the game will be rendered through. The aspect of note here is how there are enumerated modes in which the camera can sift through; and of course, such modes will be defined by future developers should they want to make their game scenes more dynamic.

4.6.2 Public Interface

The services that the UICamera provides is a visual interface for the user to keep track of their player data and navigate the main menu and settings. The main game camera is required to implement the UICamera because it extends from the main camera onto a canvas asset. The UICamera can display the HUD during gameplay, a main menu, and the knowledge drop component customized to its medical subject. Now, if the developer wants to make their game scenes more dynamic, they may utilize the services provided by the enumerated CameraMode component. And within such a component, developers can define as many of their own camera states as they need.

4.7 Statistics

The statistics module encapsulates everything needed for data management within the framework. This will assist researchers by providing easily accessible data from the players to analyze. Our knowledge drop component will be the main source of data, with multiple ways of gathering and recording player actions. Events can be customized by developers to assist with this process and make recordings react only to specific scenarios.

4.7.1 UML Diagrams

Figure 4.7.A - Knowledge Drop Data Diagram

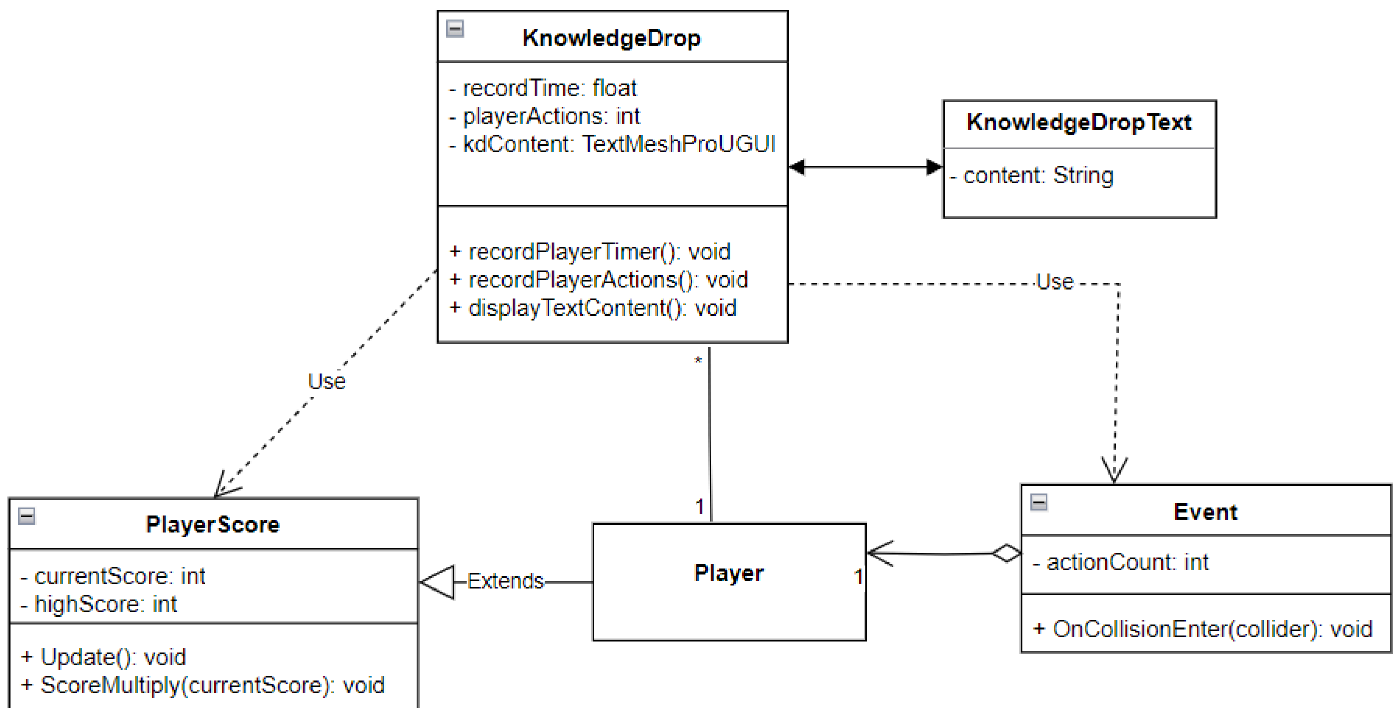


Figure 4.7.A showcases the interactions between the player data and knowledge drop component. Simple actions from the player can be recorded and stored for further analysis. The data collected will either be related to the amount of actions done by the player or the amount of time spent doing a specific action, which can be customized to satisfy the user's preferences.

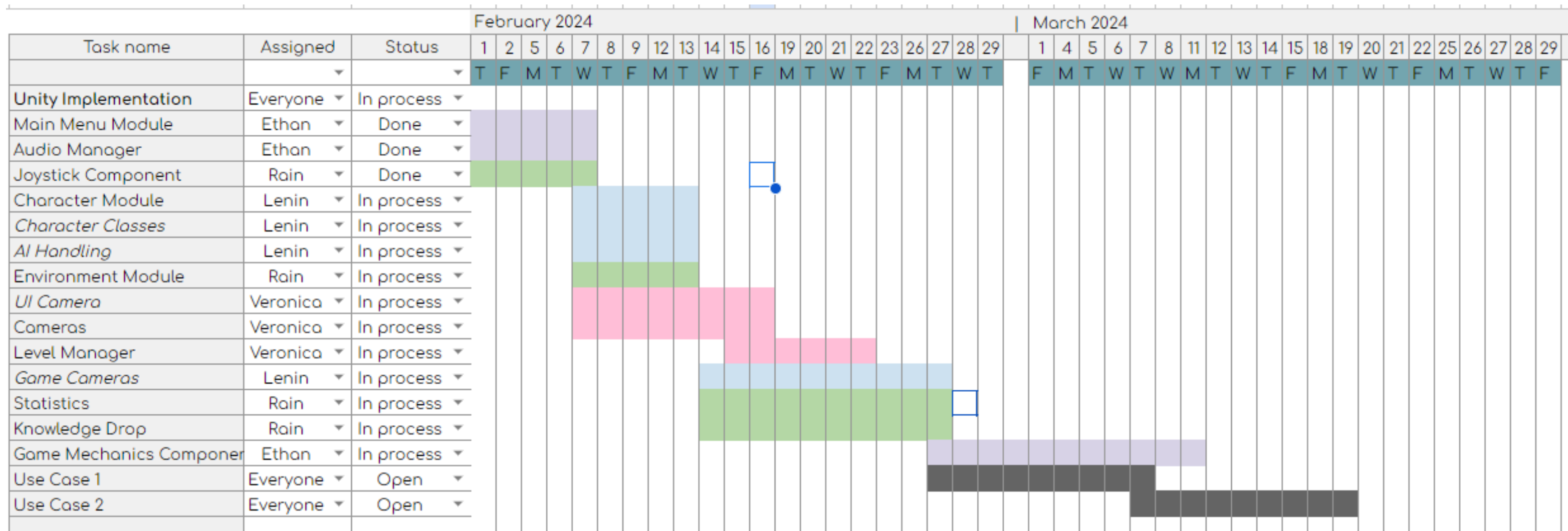
4.7.2 Public Interface

The KnowledgeDrop uses data from the PlayerScore when necessary. The highScore and currentScore can both be used to display in the kdContent text, allowing for comparison between them while users simultaneously display educational content to the player. The KnowledgeDropText can be modified on its own, using the content variable, to make for a simpler process when wanting to customize or insert additional text into the content being displayed. Events can activate whenever the player interacts with them, which the knowledge drop can then use to gather data on what the player is doing. This will be stored in the playerActions variable for further use. Time can also be recorded from these events with the recordPlayerTimer method and adding to the recordTime variable. Developers may create as many knowledge drop components assigned to different events, and customized to record different types of data.

5.0 IMPLEMENTATION PLAN

As per client request, our project schedule will be executed using the Agile methodology with sprints assigned according to client meetings. Sprints will be individually assigned to different team members throughout the semester, with progress updates twice a week in team meetings and mentor meetings. If a component implementation is not achieved within its time frame, free members will be assigned to help with the task.

Figure 5.A: Gantt Chart With Agile Methodology



Task name	Assigned	Status
Unity Implementation	Everyone	In process
Main Menu Module	Ethan	Done
Audio Manager	Ethan	Done
Joystick Component	Rain	Done
Character Module	Lenin	In process
Character Classes	Lenin	In process
AI Handling	Lenin	In process
Environment Module	Rain	In process
UI Camera	Veronica	In process
Cameras	Veronica	In process
Level Manager	Veronica	In process
Game Cameras	Lenin	In process
Statistics	Rain	In process
Knowledge Drop	Rain	In process
Game Mechanics Componer	Ethan	In process
Use Case 1	Everyone	Open
Use Case 2	Everyone	Open

Figure 5.A depicts our schedule for implementing our modules listed in section 4.0. All components should be implemented by mid march, parallel with our testing phase. The testing phase of our framework will be producing two burst games use cases in the medical subjects of COVID-19 and HPV vaccination. Alongside module implementation, our GitHub repository will be updated with the latest completed module. By the end of the semester, our GitHub repository should be user friendly for developers. All module implementation should be completed in March, before the semester break and Alpha Demo specification.

6.0 CONCLUSION

Vaccinations could play a critical role in improving symptoms of viruses, long term effects of illness, and fatalities related to such viruses. Currently, there are no ongoing studies that target adolescent vaccination rate improvements for COVID-19 and HPV. That is where our client, Dr. Amresh, comes in; how he plans to conduct such a study aimed at improving the adolescent vaccination rates through video games. In particular, our client's proposed workflow consists of developing a vaccine literacy burst game, then deploying that game to conduct his studies. The bottleneck in such a workflow, however, is the development process of the video game; how developing a video game from scratch will take up a substantial amount of time and resources. And so, our solution to such a bottleneck is to provide a video game framework in which future developers can easily build off of and extend. Thus far, we have detailed the architectural design and its various modules that our framework is expected to be implemented after; modules such as the Main Menu component, the collection of various game mechanics, the foundational architecture of a character component, and the design basis for adding gameplay to the in-game environment. And given the proposed implementation plan, the great number of milestones to achieve in the road ahead becomes evermore apparent. We hope that this burst video game framework will result in our client achieving maximum workflow efficiency when conducting his studies, ultimately seeing a significant improvement to the adolescent vaccination rates, and quite possibly ending the current pandemic situation.