RetroWave Rider

Chase The Neon Dream

<u>Team Members:</u>

Adel - *Programming, VFX, SFX*

Ahmad - *Designer/Artist, Level Designer, Programming*

Diego - *Input and Movement, Programming, Main Features*

<u>Game Information:</u>

Target Audience: Teenagers

Genre: Subway Surfer-style Runner Game

Goal: Reach the sunset at the end of the run while avoiding obstacles

Executive Summary:

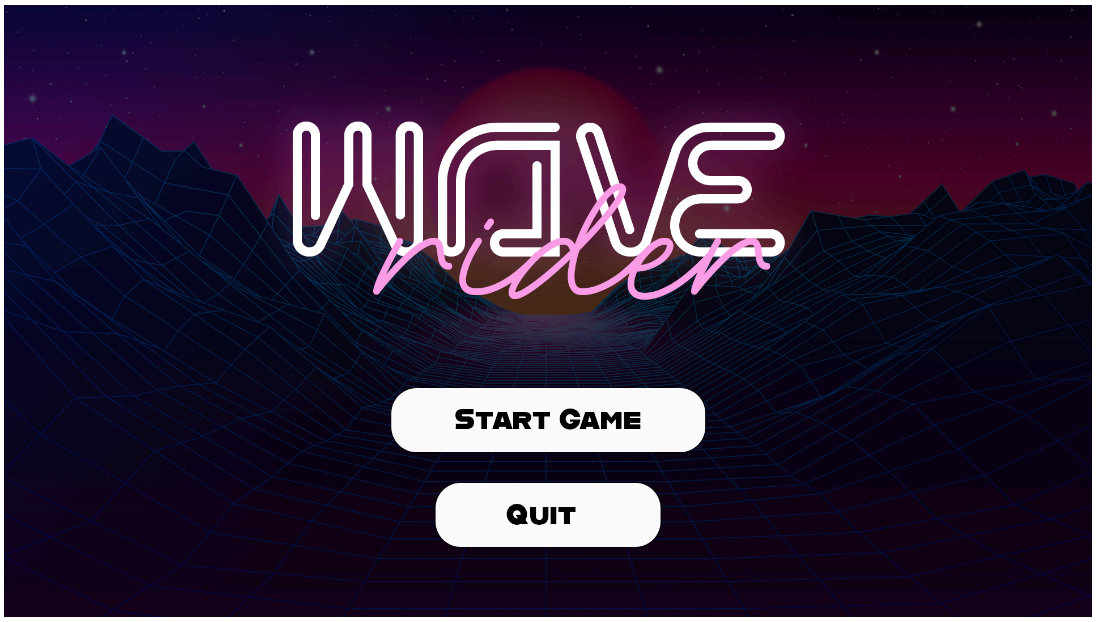
The player is driving a car in a futuristic world under a retrowave theme, a vibrant city surrounds them and their goal is to chase the sun at the end of the Level. The player is under constant movement, where the car that the player is in control of cannot be stopped or slowed down. The speed of the car will be slowly increased over time, making it more difficult for the player to stay alive and avoid obstacles that are generated along the course in blocks. The sun at the end of the level will slowly get bigger and bigger giving the player the sense that they are approaching the sun, which in turn means they are approaching the end of the game. However, there is a twist. The level is actually infinite. The sun may seem like it is approaching, however the level does not end, the true point of the game is to survive as long as possible. The player may eventually realise this, however the game will never tell the player directly that it is not beatable.

As mentioned previously, obstacles are placed along the level in blocks. There are 5 different “level blocks” that are generated in random order (but never the same multiple times in a row), which will allow each playthrough of the game to be a new and fresh experience. These obstacles must be avoided in order to stay alive. The reason for this is because each obstacle deducts a certain amount of health from the player by damaging the car more and more. There are 4 different types of obstacles: A barricade, a police car, a truck and a laser gate. Each deals damage and has different consequences. When passing through the laser gate, the player is dealt a lot of damage, however when running into a vehicle or a barricade, the player will escape with less damage and they will push these objects out of their way. Once the player’s health reaches 0, the player’s car will explode and they will be prompted to restart.

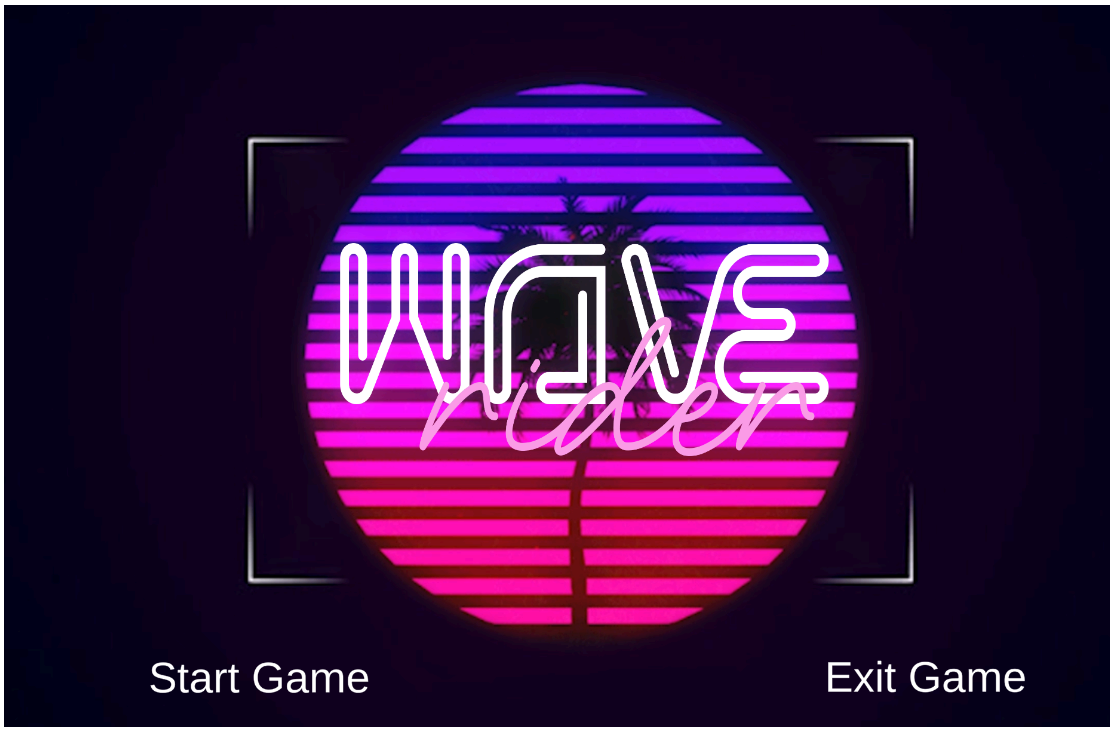
<u>User Interface</u>:

Throughout the game, there will be multiple menus for the player to interact with.

<u>**Example 1:**</u> The Main Menu original Concept (PS Sketch)

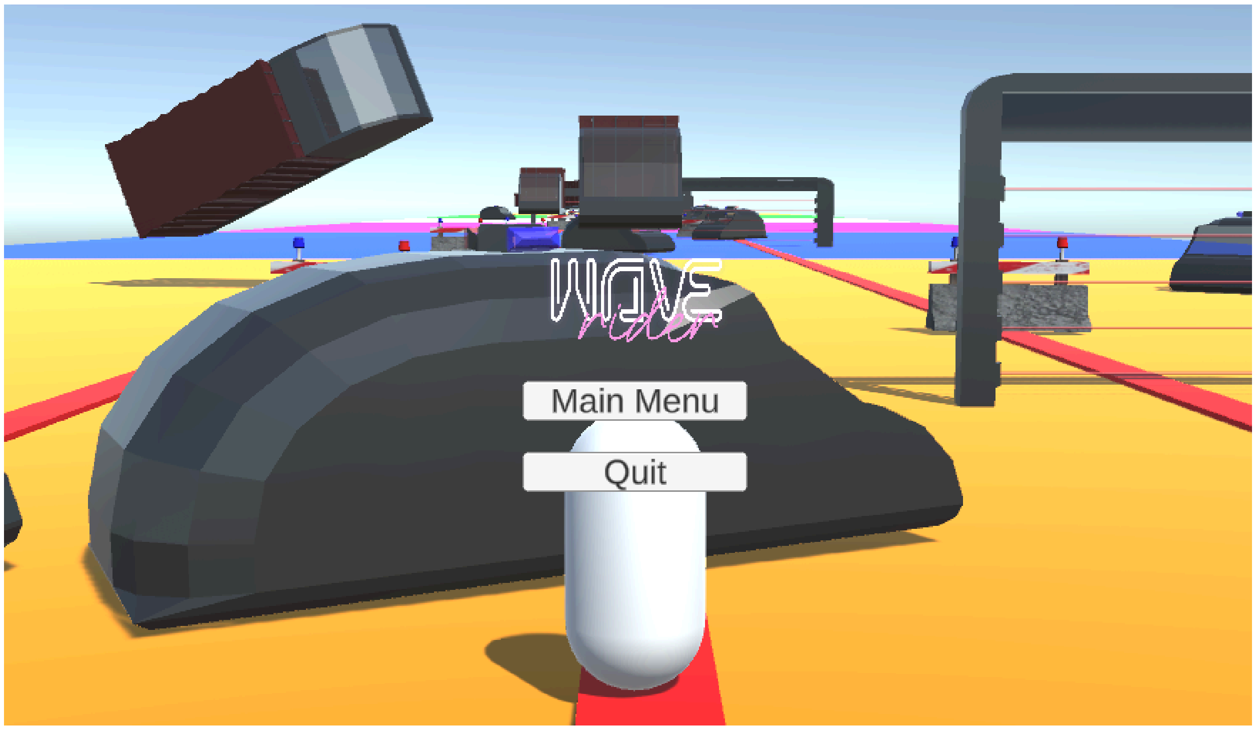


VS The <u>**Current**</u> iteration of our Main Menu:



This current iteration will most likely be slightly reworked for our final game.

-**Example 2**: The Pause Menu

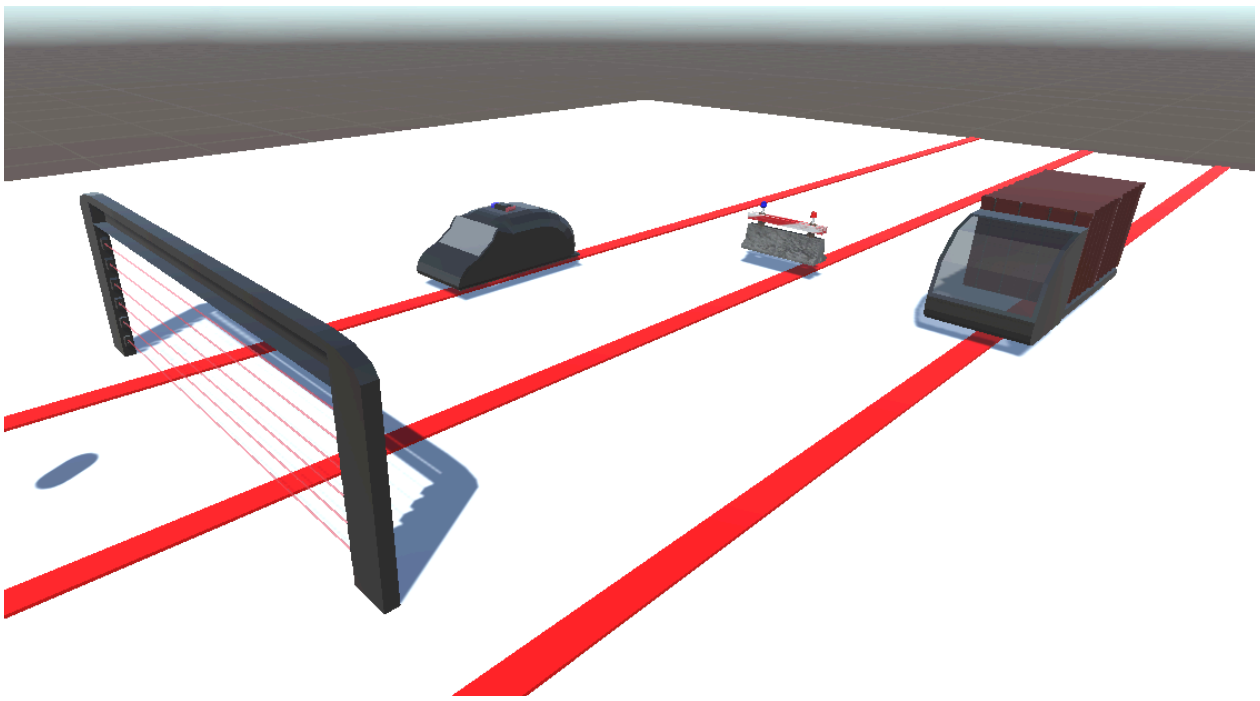


The Pause Menu will be restyled soon, this was not on our team's priority list for our first deliverable. Our goal with adding this iteration of the pause menu was to implement the functionality as opposed to focusing on the design.

Example 3: The HUD

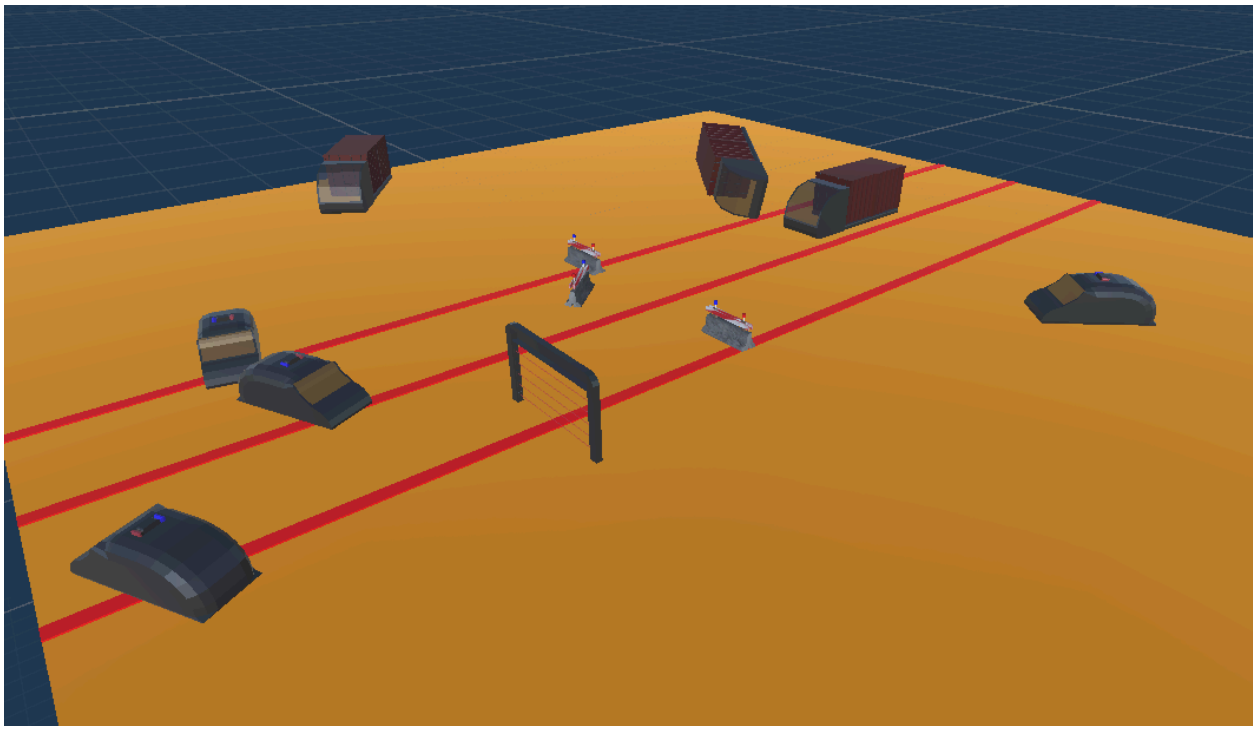
This was our basic concept for the HUD. We wanted to implement the player's health bar and in the top right corner is the powerup indicator. Unfortunately, we have not added powerups to the game yet, but we wanted to reserve the spot in the UI for it. This HUD will most definitely be reworked. We would like to add more, such as potentially a score or distance counter at the bottom middle of the screen to display how far the player has survived.

<u>Example 4</u>: Levels



As shown in the image above, the definition for “Level” for our game is a bit more vague than usual. The *entire* level that the player will drive on will be generated of multiple smaller blocks that we call levels. Each level is randomly placed by the Level Spawner and each level has its own placement of obstacles. The image above just shows a test placement of the 4 obstacles that we spoke about earlier. In this iteration, we did not create the sun in the distance, nor any background details such as buildings or lights, etc… However, each level will essentially have the same side details, it will only be the obstacles that change, therefore we already have our framework ready for all the buildings and lights, all we are missing is to implement the sun, which will appear in our next deliverable.

-Here’s an example of one of the actual Level layouts:



Controls:

In this iteration of our game, there are 3 user inputs on the keyboard. One is the “A” button which moves the player over to the left and the other is the “D” button which moves the player over to the right. The player does not directly control where in the world they want to go. There are only 3 “rails” that the car can be placed on. This way we can assure that each “rail” has a certain amount of obstacles that makes it challenging for the player to navigate. Once the player reaches the left-most “rail”, the “A” key is deactivated and they cannot move left anymore. The same can be said for the “D” key, once the player is completely to the right, they can no longer move right. The final input currently is the Escape button, which lets you open up the pause menu. This input can only be accessed once in the main game. From the main menu, there is no way to access the pause menu and the “ESC” key has no functionality. The pause menu gives the player the choice to either return to the main menu or to quit the game. Our team would like to implement a space bar functionality that allows the car to use a powerup that it has picked up along the way. This will only arrive for our next deliverable.

Design Patterns:

-The Singleton:

The singleton is a pattern that allows you to assure there is only one object of a certain class within the whole project. This is useful to prevent duplicate objects of something that only needs to be created once to perform its job or to avoid corruption of tasks. In Unity, the singleton pattern is a bit different than the way it is used in straight C#. For instance, usually you are meant to create a GetInstance() method in order to access the object already created. Within Unity however, you use the GetInstance() method as an awake function and then create a static variable named Singleton of the class from which you can access anywhere statically through the class, all while the awake function assures that no new instance is created. We used the singleton pattern for multiple things in our project and don't plan on stopping. The largest feature we use it for is the InputManager, which detects whether or not the user has activated a certain input on a device and then performs the action that we attach to this input. Of course, we do not need to detect the same input twice, therefore we only want one InputManager. We also used it for the AudioManager and UIManager. In unity, sound is played through this AudioManager through the detection of audio sources. Therefore we do not want to detect the same audio source twice and corrupt the sound. The UIManager only exists within the game scene because it is used to swap between different canvas'(Pause Menu and HUD), therefore we only need one instance of it. And finally, we also use the singleton for the

HealthSystem. We do not have any enemies in our game and there will only ever be one player at a time, therefore the HealthSystem is unique to the player and all values can be global for obstacles, UI or events to be able to access it.

-**The Observer Pattern**:

The observer is a pattern that uses events in order to “subscribe” certain classes to a certain event, in order to perform an action as a result of this event. This helps avoid hard coding and promotes decoupled code. In our game, the observer pattern is used as a part of the HealthSystem. Any class that is of interface type IChannel is allowed to subscribe to the HealthSystem’s events. The HealthSystem holds a list of channels that are currently subscribed to it, meaning they are allowed to listen to anything happening within the class. They do also have the choice of removing themselves as a subscriber. As a subscriber, they will hear when the player receives damage (And potentially in our next deliverable, gain health). Obstacles(or any class) are able to call the ApplyDamage() method through the singleton object for the HealthSystem, however they do not have access to the health field directly. Once the health is modified through the ApplyDamage() method, there is a loop that cycles through the list of Channels that are currently subscribed to the HealthSystem. Each one of these channels will receive a notification stating that the player’s health has been modified. In this current iteration, there is only 1 subscriber to the HealthSystem and that is the health bar. Whenever the player’s health changes, the change is reflected on the health bar, whether it be a deduction or an increase. However, the HUD will soon be changed to display when damage is taken using red markers. The AudioManager will also use the observer to know when the car has taken damage to play impact sounds, taking damage will also slow down the car speed temporarily. Finally, once damage is added to the player’s car as a special effect, this will use the observer pattern to understand any changes in health to better represent to the player how low their health is, through flames on the car or smoke, etc....

-**The Factory Pattern**:

The factory pattern is the final pattern implemented in this current iteration of our game. The factory pattern allows the main program to ask a subclass which class it should be currently (in our case) instantiating for the task at hand. This allows the code to be more decoupled and very flexible. However, our game uses a modified version of the factory pattern. We decided to use the pattern for level spawning. Our physical Factory class is the LevelSpawner class and our interface is ILevel. As mentioned previously, each level is a block that can be used multiple times, but there are only 5 Levels. Each Level has its own script that then inherits from the parent Level, as well as the Ilevel interface. Each Level has its own ID that then within the Factory class (LevelSpawner) is used to call the instantiation of the level. Instead of having a switch statement that decides which ID is associated to which class, there is a random number generator that allows these levels to be generated in a random order. So although we are using the factory pattern to find and generate these levels that are spawned in afterwards, we are finding them in a different way compared to the usual factory pattern.

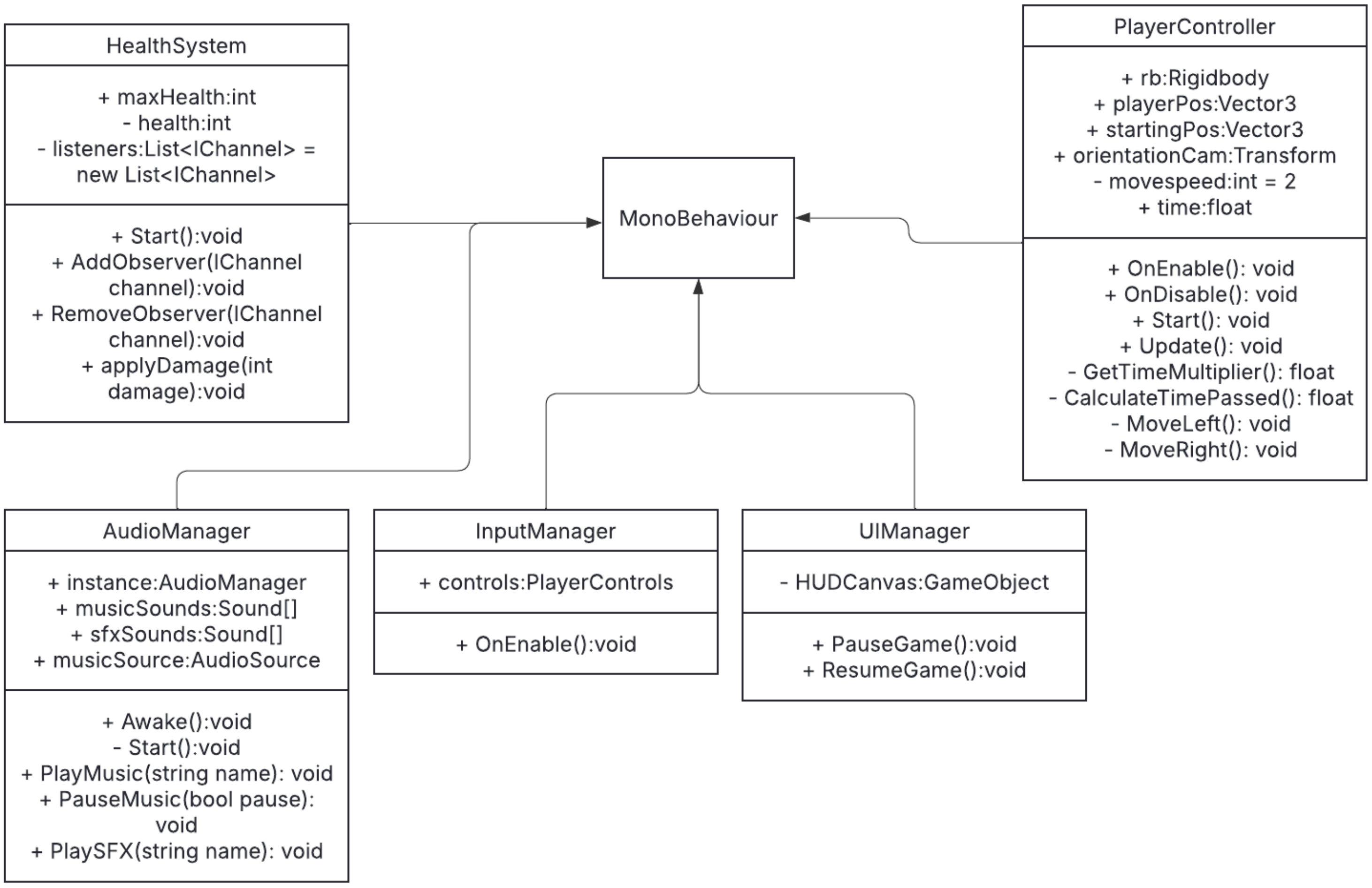
Task and Schedule Breakdown:

- Finish Observer pattern implementation: (23rd March)
    - Red HUD marker (Diego)
    - Impact Sound (Adel)
    - Physical Car Damage FX (Adel)
    - Speed fluctuation for damage representation (Diego)
- Finish Main Menu (24th March)
    - Button Layout (Ahmad)
    - Title Screen (Diego)
- Finish Pause Menu (30th March)
    - Live Pause Menu (Diego)
    - Button Layout (Ahmad)
    - Graphics (Diego)
- Create Powerups (30th March)
    - Flying (Adel)
    - Speed-up (Diego)
    - Wave Slingshot (Maybe)
    - Pickup Sound Effects (Adel)
- Create Actual Car (6th April)
    - Create Animations (Adel)
    - Create Powerup Effects (Diego)
    - Create Damage Effects (Diego)
    - Add Sound (Adel)
- Add Score Counter (Diego or Ahmad) (6th April)
- Level Completion (23rd March)
    - Background Sun (Approaching) (Ahmad)

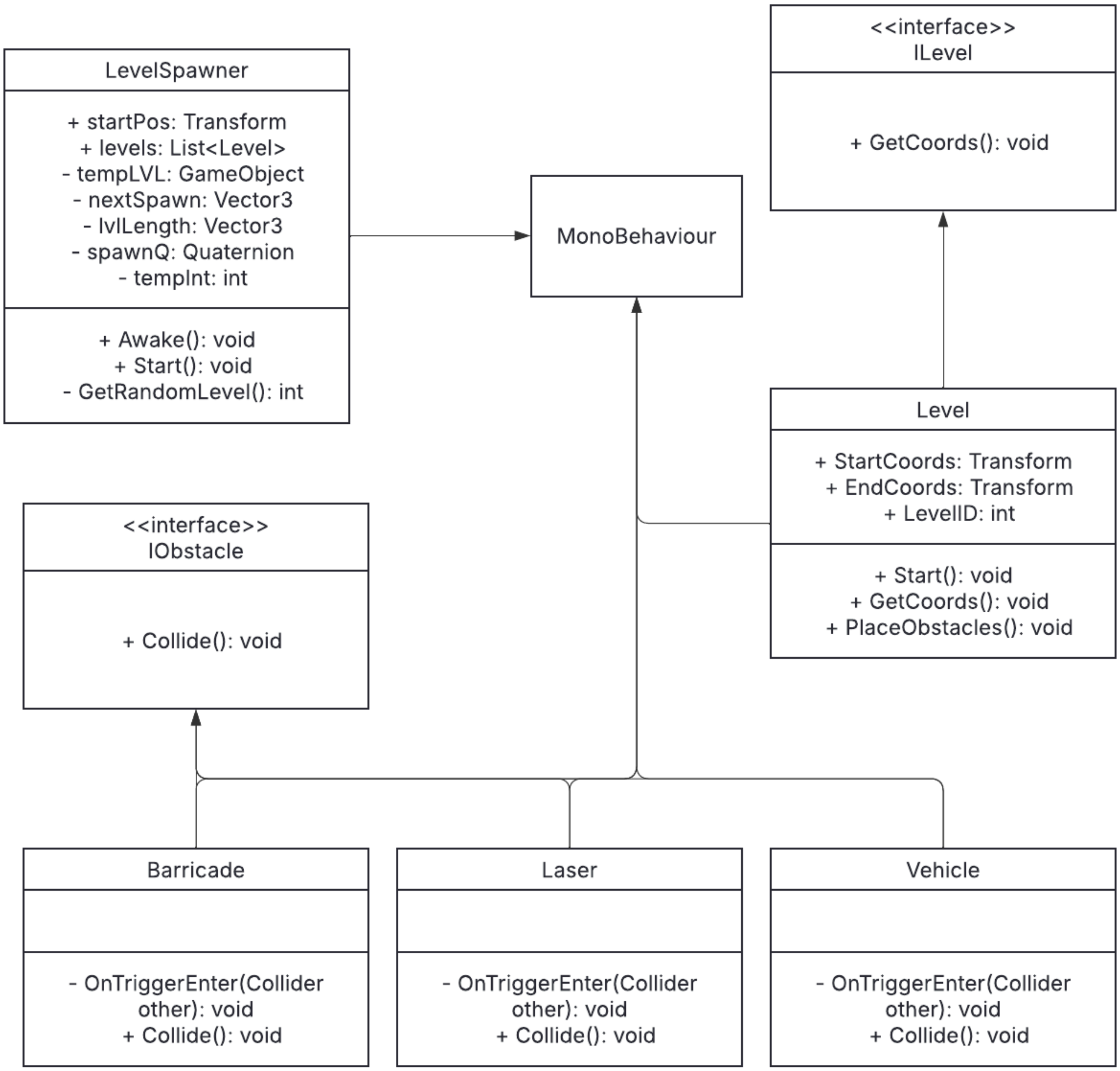
    - Side Buildings + Scenery (Ahmad)
- Finish Obstacles (26th March)
    - Neon Puddles (Diego)
    - Small TouchUps
    - Shatterable Barricade(Diego)

UML Diagrams

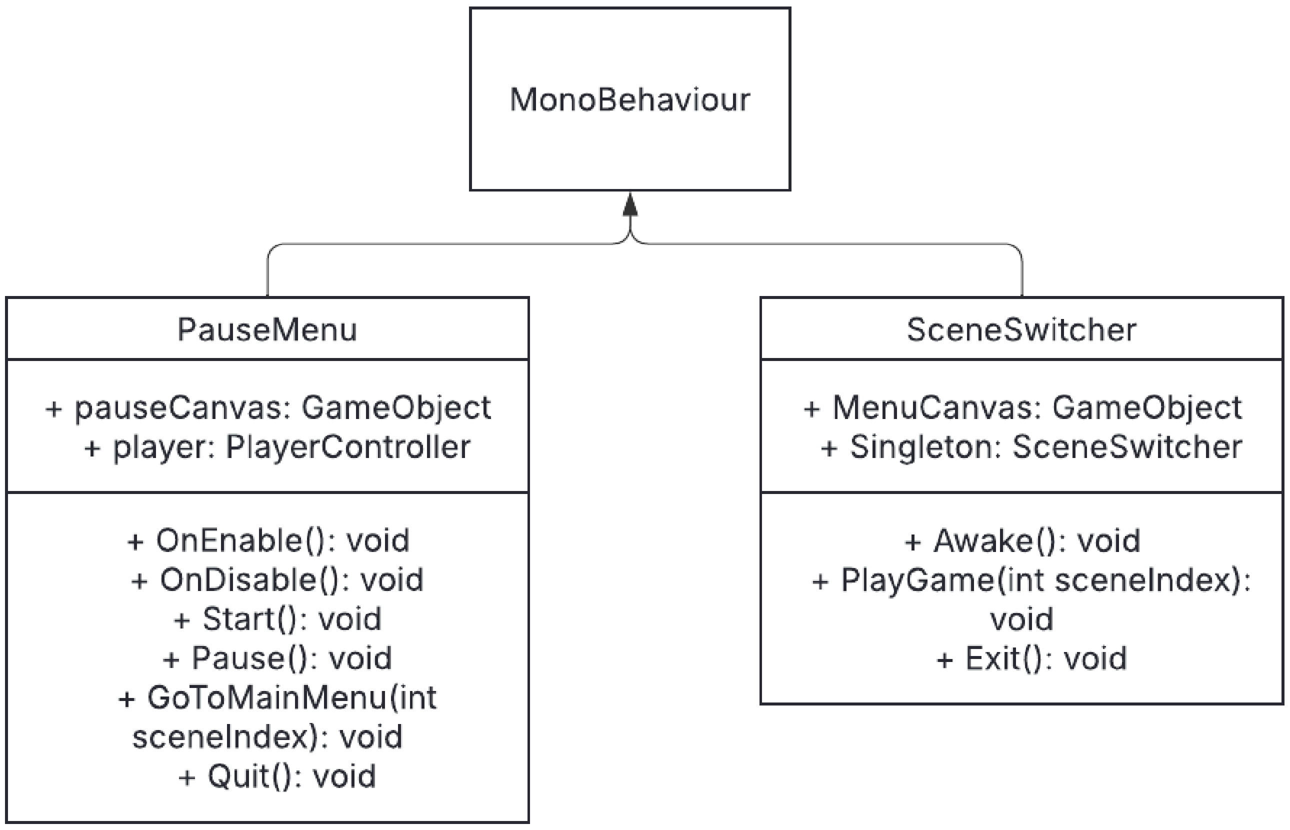
Core Systems



Levels & Obstacles



Scene Management


MonoBehaviour
PauseMenu
+ pauseCanvas: GameObject
+ player: PlayerController
+ OnEnable(): void
+ OnDisable(): void
+ Start(): void
+ Pause(): void
+ GoToMainMenu(int sceneIndex): void
+ Quit(): void
SceneSwitcher
+ MenuCanvas: GameObject
+ Singleton: SceneSwitcher
+ Awake(): void
+ PlayGame(int sceneIndex): void
+ Exit(): void