

RetroWave Rider

Chase The Neon Dream

Team Members:

Adel - *Programming, VFX, SFX*

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

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

Game Information:

Target Audience: Teenagers

Genre: Subway Surfer-style Runner Game

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

Executive Summary:

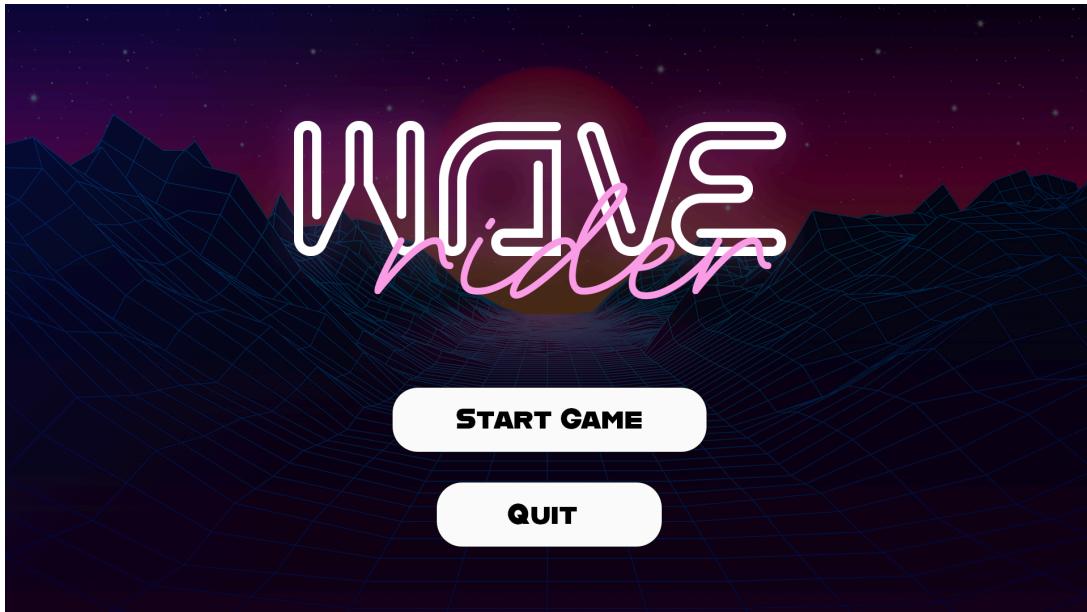
The player is driving a car in a futuristic world under a retrowave theme, a vibrant city surrounds them and their goal is to chase the sun at the end of the Level. The player is under constant movement, where the car that the player is in control of cannot be stopped or slowed down. The speed of the 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.

User Interface

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

Example 1: The Main Menu original Concept (PS Sketch)



VS The Current 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.

Here is the Final Design of the Pause Menu:

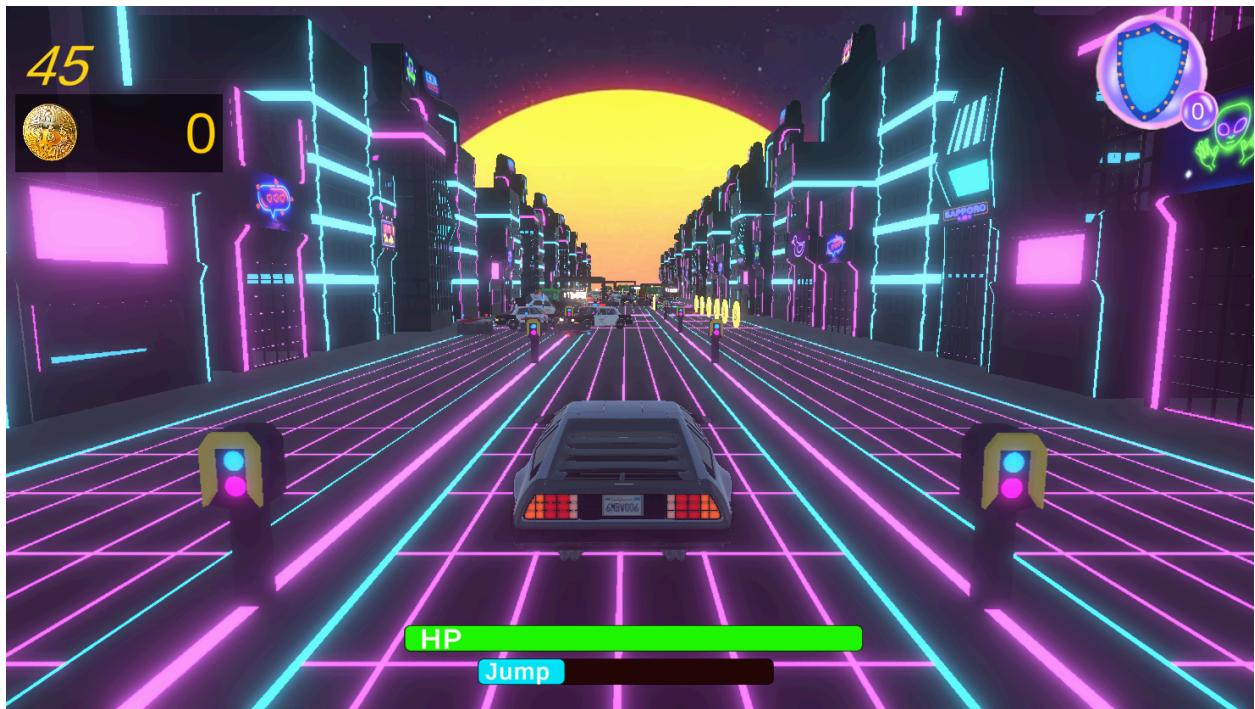


Example 3: The HUD



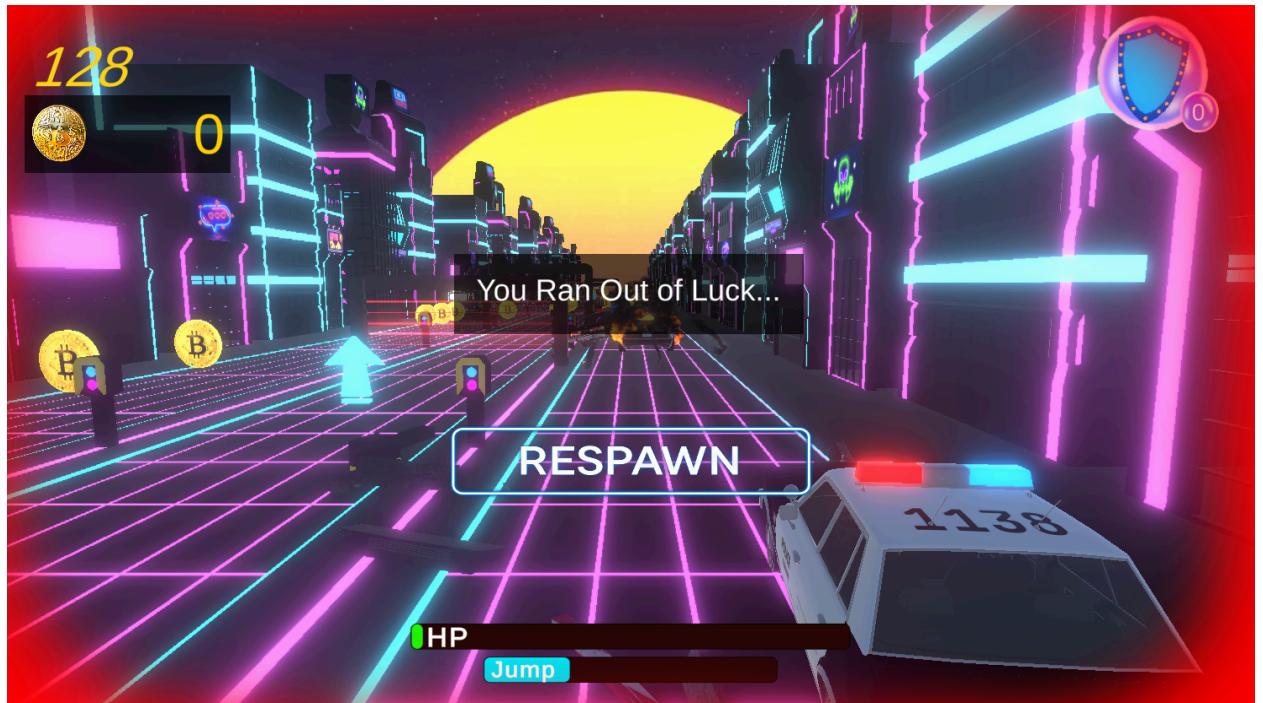
This was our basic concept for the HUD. We wanted to implement the player's health bar and in the top right corner is the powerup indicator. Unfortunately, we have not added powerups to the game yet, but we wanted to reserve the spot in the UI for it. This HUD will most definitely be reworked. We would 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.

-Our Final Iteration of the HUD:



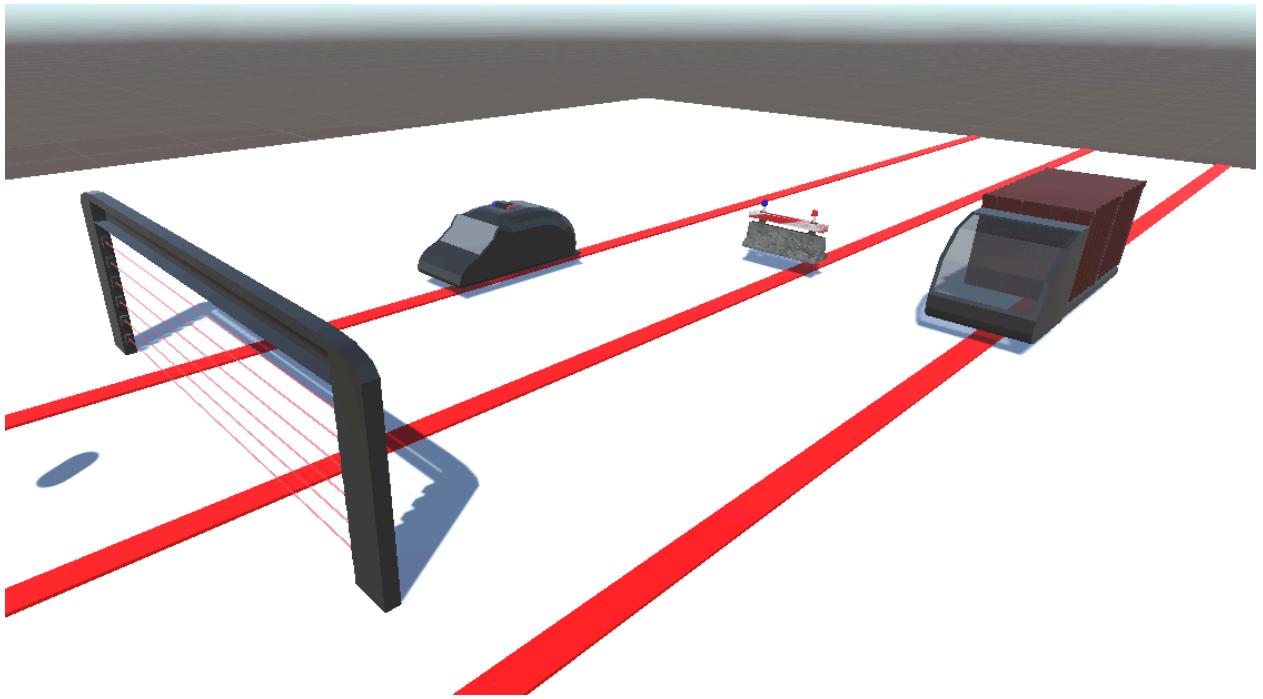
In the top left, we have the Score Counter and the Coin counter. The score counter is made up of the distance the player has traveled, the powerups and jumps that they've collected, coins and other miscellaneous things. In the bottom middle of the screen, the health has been relocated and attached to the jump meter. Whenever the player picks up a jump powerup, the meter will partially fill up, once it is full, the player can jump and the meter will empty.

Example 3: Death Screen



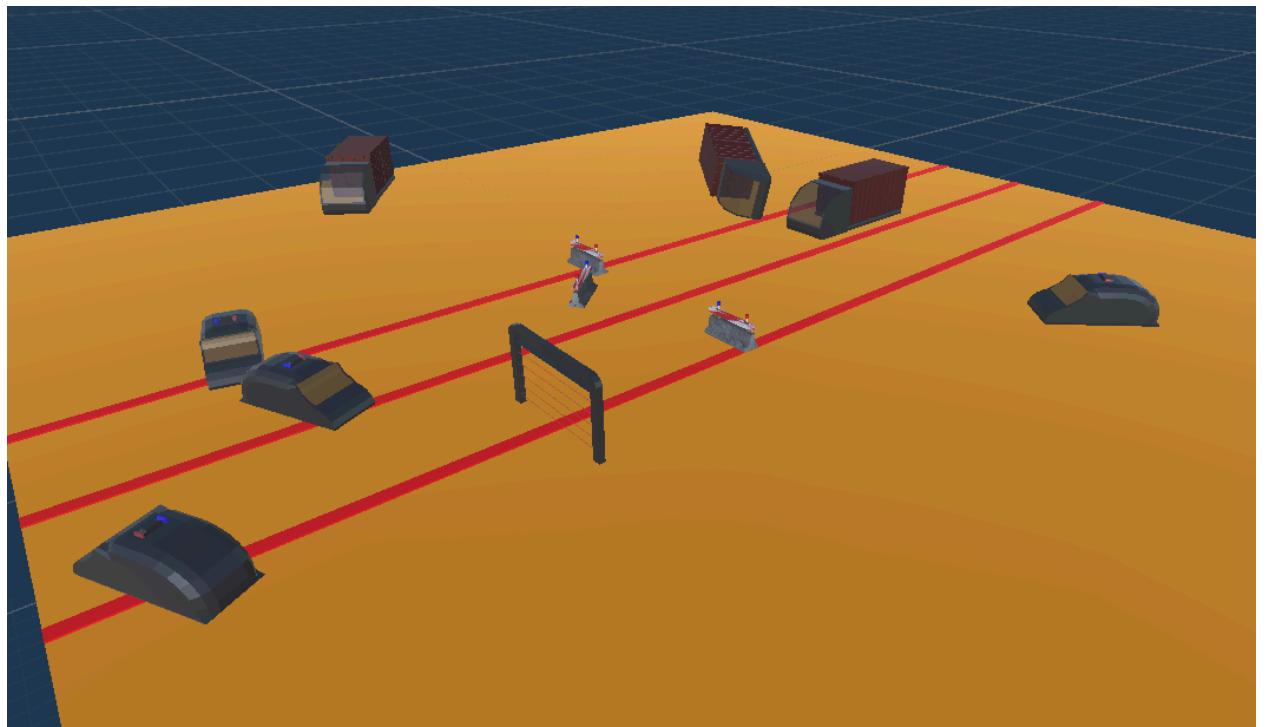
Upon death, the player's vehicle separates from the camera and an animation plays. They are prompted to respawn which will completely reset the game.

Example 5: Levels

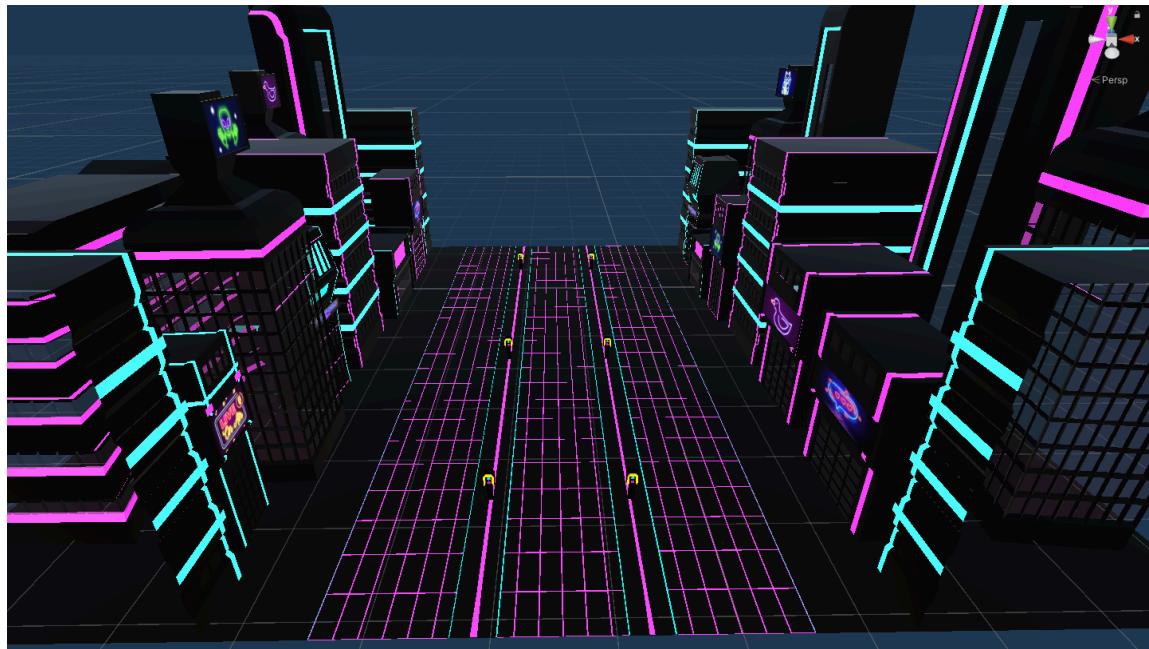


As shown in the image above, the definition for “Level” for our game is a bit more vague than usual. The *entire* level that the player will drive on will be generated of multiple smaller blocks that we call levels. Each level is randomly placed by the Level Spawner and each level has its own placement 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.

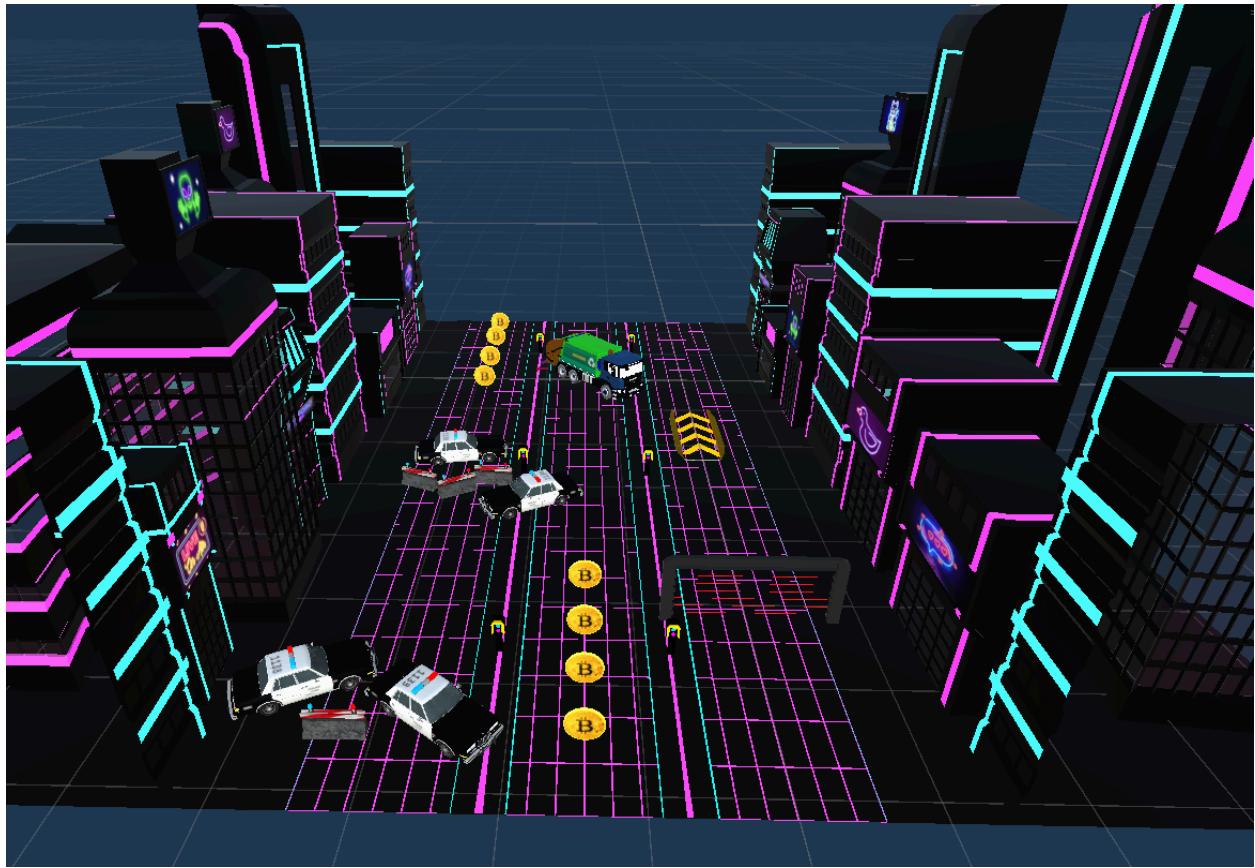
-Here's an example of one of the actual Level layouts:



-Here is the final iteration of the Basic Empty Level:



-Here is an Example of a Spawning Level:



There are 5 different implementations of the basic level and each one also has randomized aspects to it. The coins, powerups, and certain obstacles are all randomized upon loading so that even if the player runs into the same level, it may be different each time. Sometimes coins will appear in one corner but not the other, sometimes both, sometimes there will be different powerups in the same spot each time and sometimes certain obstacles will be moved. For example, on one of the levels, a police car spawns on either the right or left side of a Laser Gateway, depending which randomized value is assigned.

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.

In the final iteration of the game, there are now a total of 5 inputs, which is an additional 2 from our first deliverable. The first one is the “E” button which allows the player to use a shield powerup if they have picked up some of the collectibles on the map. If one is used, a timer starts and another can’t be used until the timer is over. The final input is the “Space Bar” which is the jump button. As mentioned previously, there is a jump meter that fills up as the player collects Jump powerups and once it is full, the player can click the “Space Bar” to jump and deplete the jump meter.

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.

-Chain of Responsibility Pattern:

The first pattern that was implemented in this iteration of our game was the Chain of Responsibility pattern. This pattern allows the game to handle progressive checkpoints to pre-made definitions. It inherits from an abstract base class that defines each handler that as mentioned act as checkpoints that is managed by

the DamageHandlerManager. In our case, we use it while subscribed to the observer pattern to detect how much health the player currently has and compare it to pre-made thresholds that are assigned to each handler. If it passes this threshold, it will effectuate a command set to this handler. The commands we assign are all damage based because we use the pattern as a damage handler. Therefore once the player's health reaches our first threshold, it will enter the first damage handler and either ignite an animation or destroy an object on the player's vehicle to represent damage. We have 4 handlers all handling different thresholds and once the end is reached, the chain is cut. By the time the Manager reaches the last handler, all the different possibilities have been used up, because we randomize the things that happen in each handler. For example in the first damage handler, a light can come off, or it could be a mirror, a bumper, etc... But when reaching handler 4, everything has come off by that point and the car is also set on fire.

-The Strategy Pattern:

The final pattern in our game is a modified version of the strategy pattern. Usually the strategy Design Pattern inherits from an interface that then defines what methods will be used, however, we decided to use a parent class instead. This gave us more flexibility to set a pre-definition for the method and then override it in each child class. It was easier for testing and easier in case we wanted to add extra functionality. Regardless, there are 2 methods that have as a part of the strategy pattern inside the Level Parent Class and that is to place Obstacles and Place the coin trail. Each one has its own overridden definition in the child classes where the values are randomized so that both the obstacles, powerups and the coin trail can be different for each level and for each instance of the

same level. The final spawned powerups and obstacles also depend on each other. If certain obstacles spawn in a certain location, then some powerups won't or may spawn somewhere else. Same goes for the coins.

Task and Schedule Breakdown:

- Finish Observer pattern implementation: (23rd March)
 - X - Red HUD marker (**Diego**)
 - X - Impact Sound (**Adel**)
 - X - Physical Car Damage FX (**Diego**)
 - X - Speed fluctuation for damage representation (**Diego**)
- Finish Main Menu (24th March)
 - X - Button Layout (**Ahmad**)
 - X - Title Screen (**Diego**)
- Finish Pause Menu (30th March)
 - X - Live Pause Menu (**Diego**)
 - X - Button Layout (**Ahmad**)
 - X - Graphics (**Diego**)
- Create Powerups (30th March)
 - Flying (**Adel**)
 - X - Jump(**Diego**) - (Under Health Bar in blue)
 - X - Speed-up (**Diego**)
 - Wave Slingshot (Maybe)
 - X - Pickup Sound Effects (**Adel**)
- Create Actual Car (6th April)
 - X - Create Animations (**Adel**)

~~X~~ - Create Powerup Effects (**Diego**)

~~X~~ - Death Animation

~~X~~ - Create Damage Effects (**Diego**)

~~X~~ - Add Sound (Adel)

- ~~X~~ - Add Score Counter (**Diego or Ahmad**) (6th April)

- ~~X~~ - Coin/Collectible Trail

~~X~~ - Randomized Trail

~~X~~ - Make Collectible

- Level Completion (23rd March)

~~X~~ - Background Sun (Approaching) (**Diego**)

~~X~~ - Side Buildings + Scenery (**Ahmad**)

~~X~~ - Render Distance (**Diego**) -(Flyweight)

- Finish Obstacles (26th March)

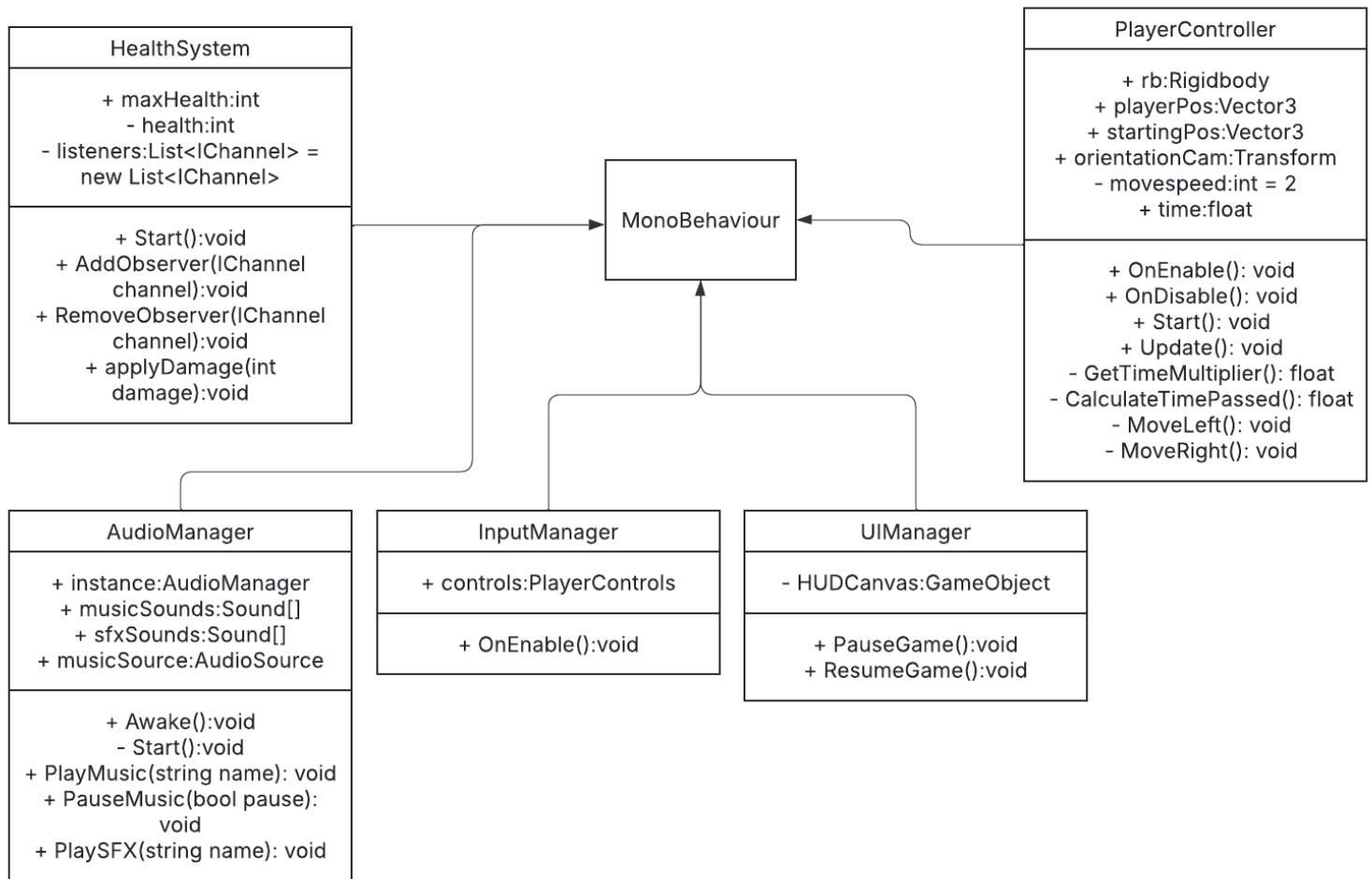
~~X~~ - Neon Puddles (**Diego**)

~~X~~ - Small TouchUps

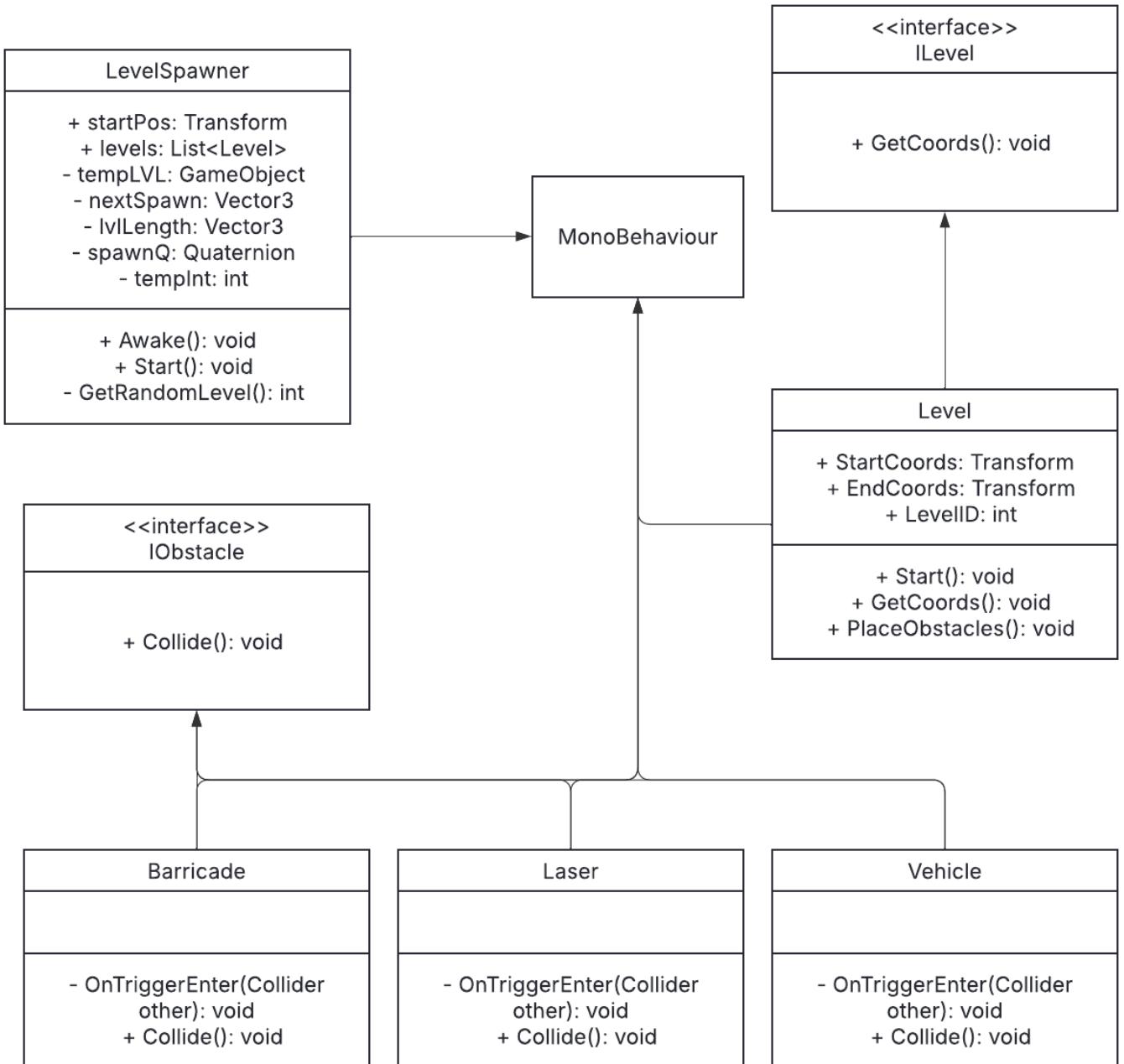
~~X~~ - Shatterable Barricade(**Diego**)

UML Diagrams

Core Systems



Levels & Obstacles



Scene Management

