BOGAZICI UNIVERSITY

INDUSTRIAL ENGINEERING

IE 201 FALL'21

INTERMEDIATE PROGRAMMING

GROUP 9 PROJECT REPORT

Emre Köksal

Melis Tuncer

Merve Nur Hündür

Recep Eren Durgut

Contents

Introduction to Our Game

The game we have designed is a car game on a dynamic road. In our game, the user aims to collect as much coins as possible while trying to keep the car ‘alive’, by moving it left or right.

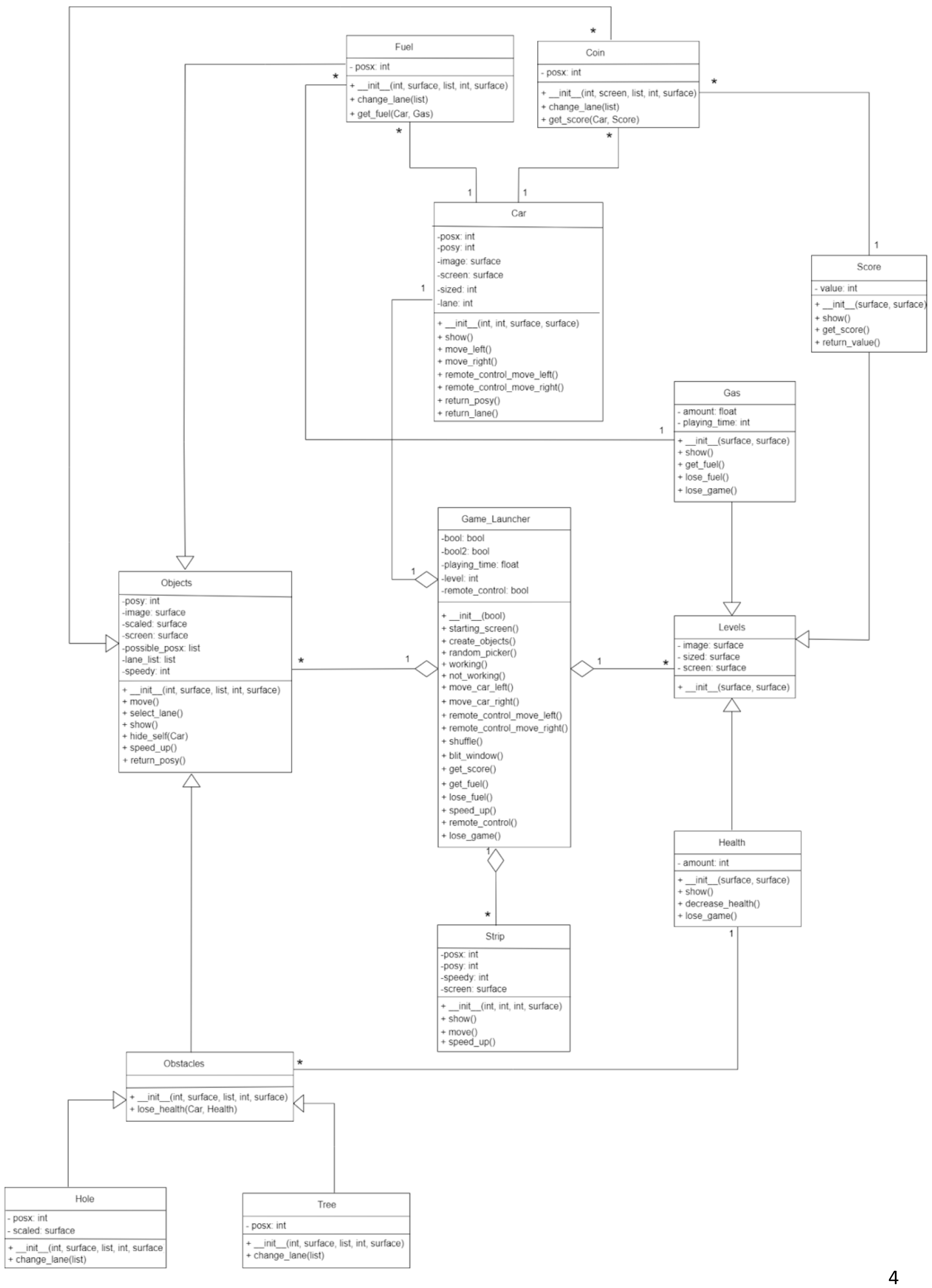
When the game begins, the car starts to move forward. Actually, the car looks like it is moving forward with the road flowing down. The gamer is allowed to move the car to the right or to the left by pressing “d” and “a” keys, respectively. On the top-right of the game window, there are 3 health icons which represents the health level of the player. On the bottom-right of the window, there exists a fuel meter, which has an initial value of 100 and decreases as the time goes on. On the road, the car will encounter some objects that represent obstacles (consisting of trees and holes), fuels, and coins. The fuel level of the car can be recharged to 100 by hitting a fuel object. In addition, each time the car crashes into an obstacle such as a tree or hole, the health level of the car will decrease by 1. The health level cannot be increased in any way.

The car is able to collect coins by hitting them. The total amount of coins collected can be seen at the top-right corner of the screen. This amount indicates the success of the player and will be represented to the player at the end of the game.

The level of the game can be seen at the top of the screen. The game starts at level 1, which is the easier one. The level increases by 1 every 30 seconds and the game gets harder because objects start to flow down faster. In addition, in the last 10 seconds of each level, the game keys are swapped, which makes the game even harder by confusing the player. When this happens, the player should press “d” instead of pressing “a” to move the car left, and vice versa.

If the health or fuel level of the car drops to zero, the game ends and the total amount of coins collected is shown. In that sense, to keep the car on the way and to gain more coins, the gamer needs to avoid crashing into the obstacles, and collect as much coin and fuel as s/he can.

Class Diagram



Class Definitions

Game Launcher: Our project has a Game Launcher class as the main builder. Since this class creates the objects of Car, Strip, Objects, and Levels classes, there is an aggregation relationship between these classes and the Game Launcher Class. We simply execute the game by using this class. It is the manager class and is responsible for the flow of the game.

- **Car**: One of the classes that have an aggregation relationship Game Launcher class is the Car class. Car and Game Launcher classes have 1-1 aggregation kind of relationship. Car class includes the image of the car and attributes position to it. While the move_right and move_left methods enable the car to move right and move left by pressing “d” and “a” keys respectively, the remote_control_move_left and remote_control_move_right methods initiate a challenge for the gamer. At the end of the last 10 seconds of every level of the game, these methods switch the functions of the “a” and “d” key buttons.

- **Strip**: The second class has an aggregation kind of relationship with the Game Launcher class is Strip Class. The show method makes the strips of the road visible. The move method creates the feeling of the car moving forward by moving the strips down. The speed-up method increases the speed of the strips at every level to increase the challenge of the game.

- **Objects**: Another class that has an aggregation kind of relationship with the Game Launcher class is the Objects class. Objects class defines objects in the game that appear on the road and utilize the gamer to stay on the game more or less. Objects class is the parent class of Coin, Fuel and Obstacles classes. This class holds the move, show, select lane, hide self and speed up methods of its childs.

    - **Coin**: Coin class defines the coin objects that appear on the road and allow the gamer to increase his/her score by collecting the coins, which explains the 1-many association kind of relationship between the Coin and Score classes.

    - **Fuel**: Fuel class defines the fuel objects that appear on the road and allow the gamer to increase his/her gas level. Fuel objects increase the gas level of the car if s/he can collect these fuels. That's why Gas and Fuel classes have 1-many association kind of relationship between them.

    - **Obstacles**: Obstacles Class is the parent class of Tree and Hole classes, and inherits from Objects class. It defines trees and holes that appear on the road which make the gamer lose health. Therefore, Obstacles class has an association kind of relationship between Health class.

        - **Tree**: Tree Class is a special kind of Obstacles. It defines the tree objects that appear on the road. As it inherits from Obstacles Class; when the car crashes tree objects, it causes a decrease in health level.
        - **Hole**: Hole Class is a special kind of Obstacles. It defines the hole objects that appear on the road. As it inherits from Obstacles Class; when the car goes over a hole object, it causes a decrease in health score.

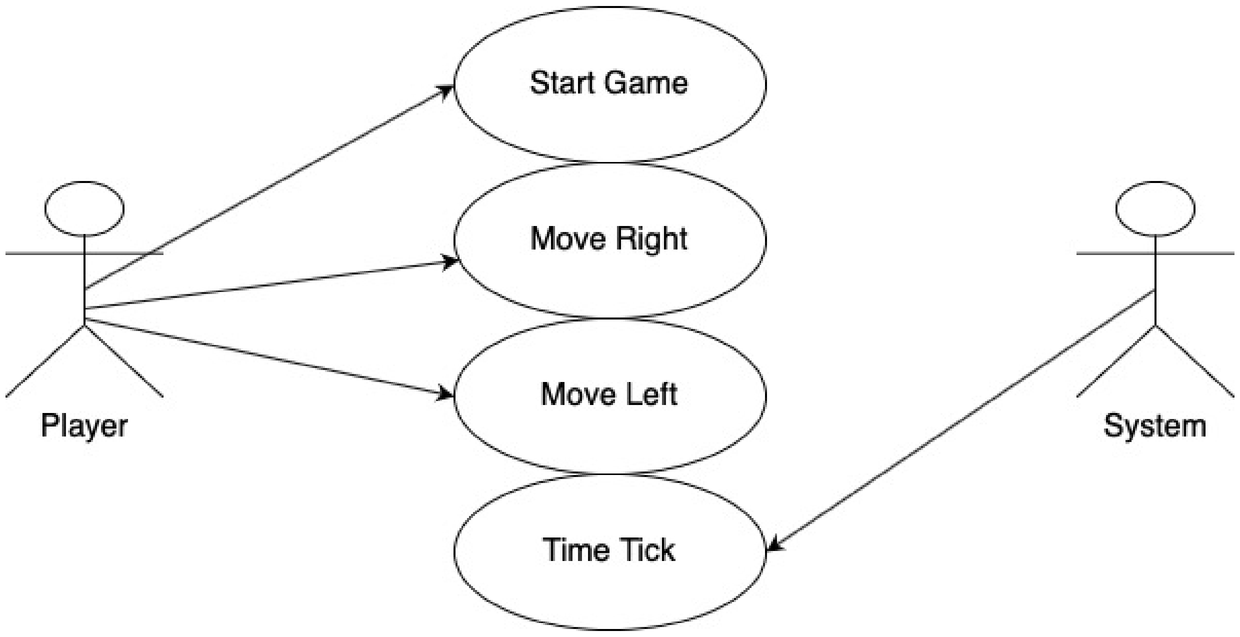
- **Levels**: The last class that has an aggregation relationship with the Game Launcher class is the Levels class. Levels class is the parent class of Health, Score, and Gas classes.

    - **Health**: The first class that inherits from Levels class is the Health Class. Health class mainly keeps track of the health level of the car. The car has 3 health levels at the beginning of the game. When the car hits one of the obstacles, the health level drops by 1 and one of the health images disappears. So, the Health class has an association relationship with the Obstacles class. If the health level reaches zero, the game is over.

- **Score**: The second class that inherits from the Levels class is Score Class. Score class also has an association kind of relationship with Coin class because if the car aligns with the coins, the score level will increase.

- **Gas**: The third and the last class that inherits from the Levels class is Gas Class, which holds the gas level of the car. Gas class also has an association kind of relationship with Fuel class. That is because while gas level decreases by time, it can be increased by hitting fuel objects. The game is over when the gas level drops to zero if the player cannot collect enough fuel objects to meet the positive gas level.

Use Case Diagram



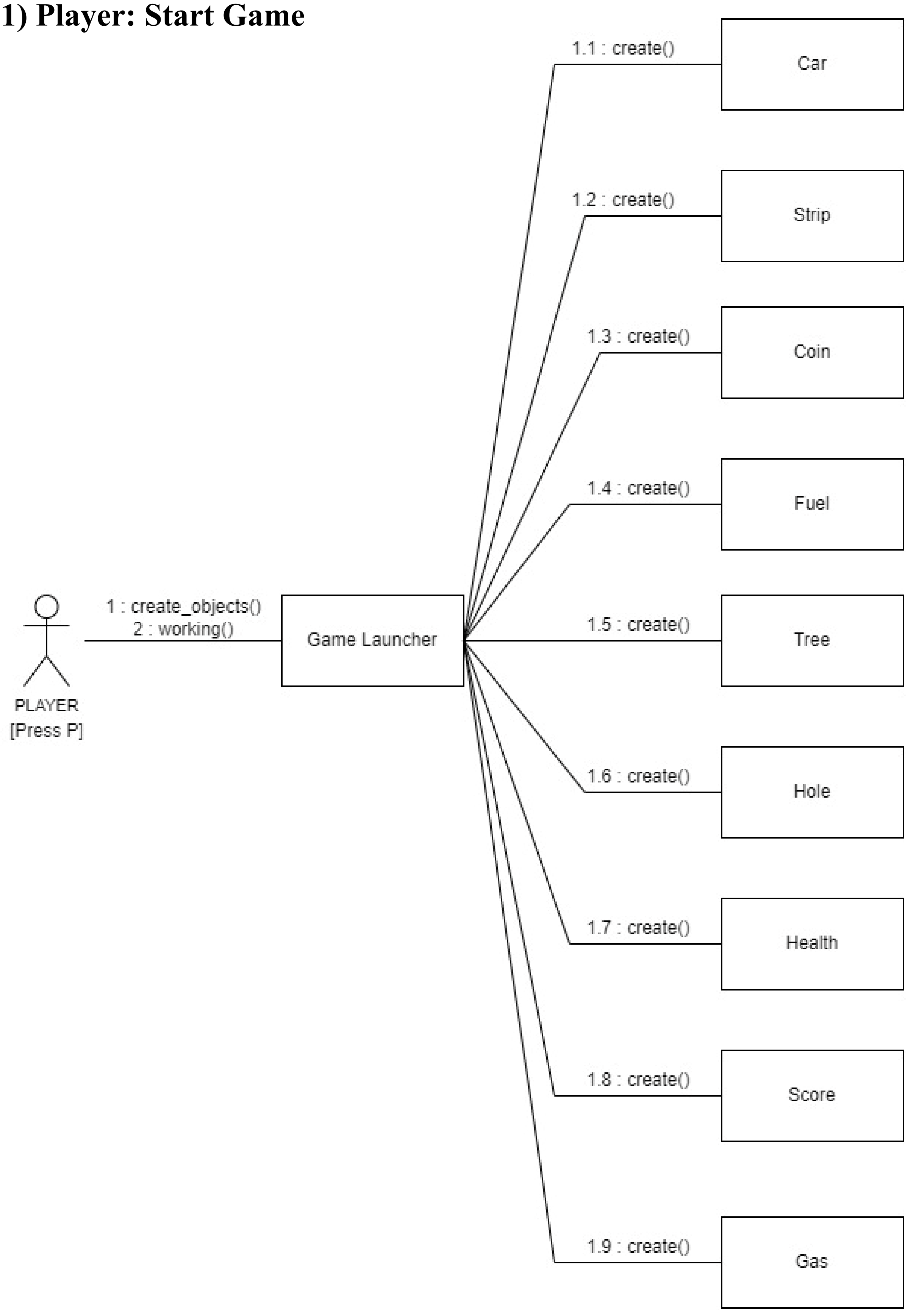
We have two actors in the game: System and Player.

The player presses the “P” key and starts the game. After the game starts, to avoid hitting obstacles or to collect fuel and coins, the player can go left by pressing the key “A” and go right by pressing the key “D” unless remote control of the system works. If this is the case, the keys swap their functions.

As the game continues, the system implements the Time Tick use case that decreases the amount of gas level over time.

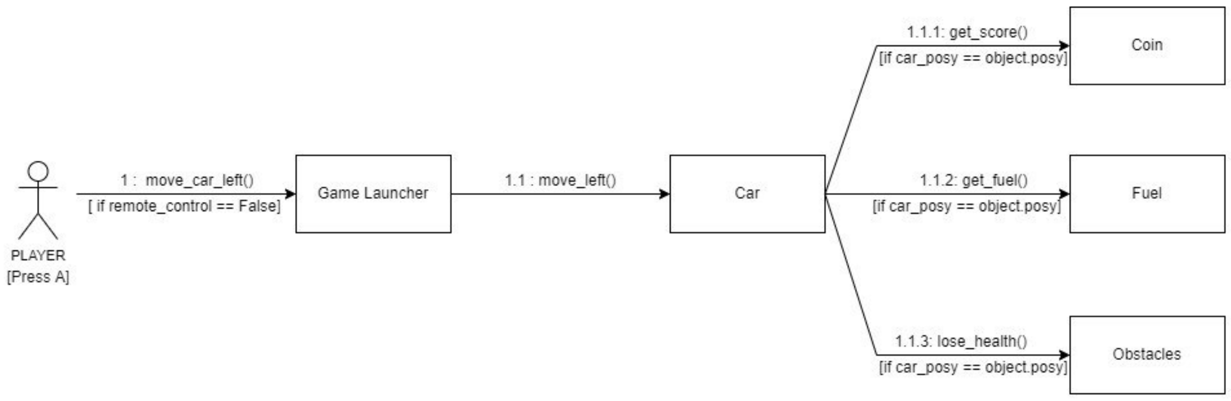
Collaboration Diagrams

1) Player: Start Game

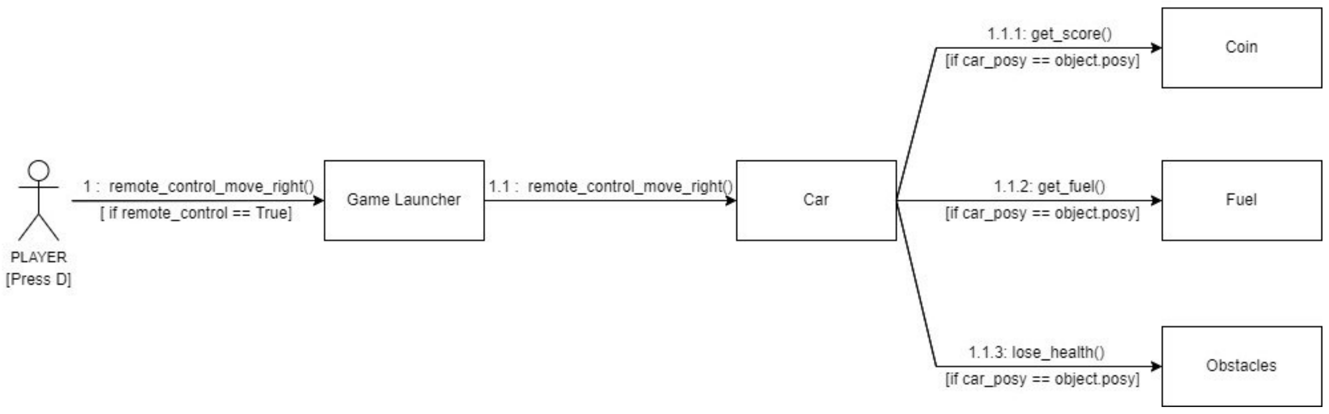


The first use case of the Player includes starting the game. The Player should press the “p” button to start the game. This use case triggers some sequence of methods in the Game Launcher class. Firstly, it calls the *create_objects* method which initializes all the required objects of the game. In other words, the *create_objects* method calls __*init*__ methods of all other classes that the game has. After these objects are created, the *working* method of the Game Launcher class is called. This method changes the bool type of the Game Launcher object to True, which was False before. This change enables other time tick use cases, that controls the events in the game, to work.

2) Player: Move Left

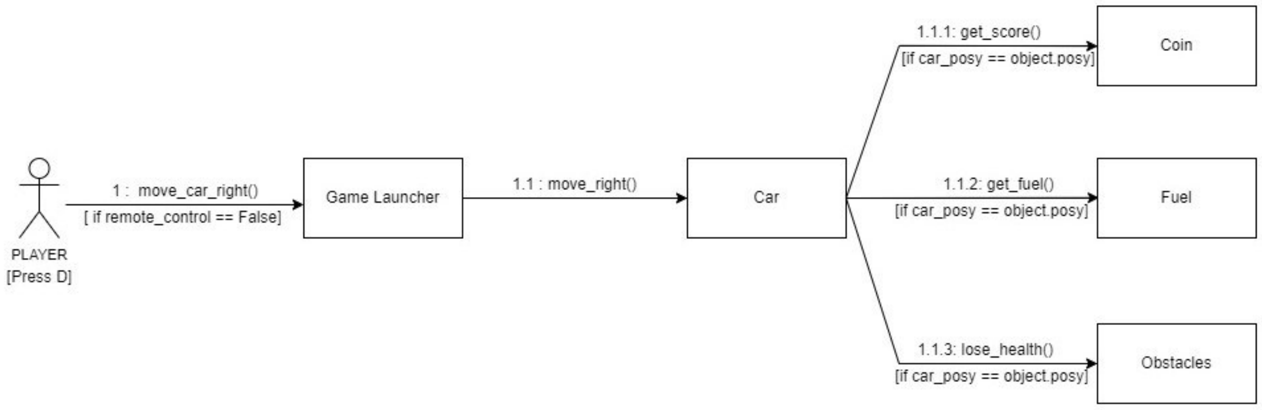


Another use case of the Player includes moving the car to the left lane. When the player presses “A” button on the keyboard, *move_car_left* method executes and Game Launcher calls *move_left* method from Car class to send a message to the car. The *move_left* method moves the car to the left lane till there is no lane to the left of the car. There is a condition for *move_car_left* method: *remote_control* attribution of the Game Launcher object must be false. Otherwise, controls would be reversed as mentioned below.

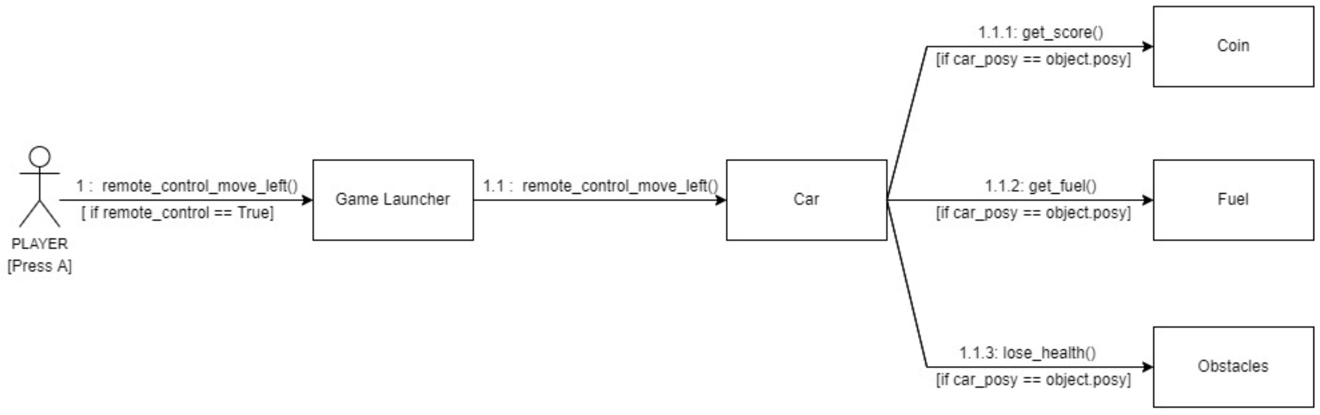

PLAYER
[Press D]
1 : remote_control_move_right()
[if remote_control == True]
Game Launcher
1.1 : remote_control_move_right()
Car
1.1.1: get_score()
[if car_posy == object.posy]
Coin
1.1.2: get_fuel()
[if car_posy == object.posy]
Fuel
1.1.3: lose_health()
[if car_posy == object.posy]
Obstacles

The next sequence of messages that is related to move left use case of the Player includes a challenge for the player. The system is the same as the basic movement functions above. However, this time pressing the “D” button makes the car move left if *remote_control* attribution of the Game Launcher object is true.

3) Player: Move Right

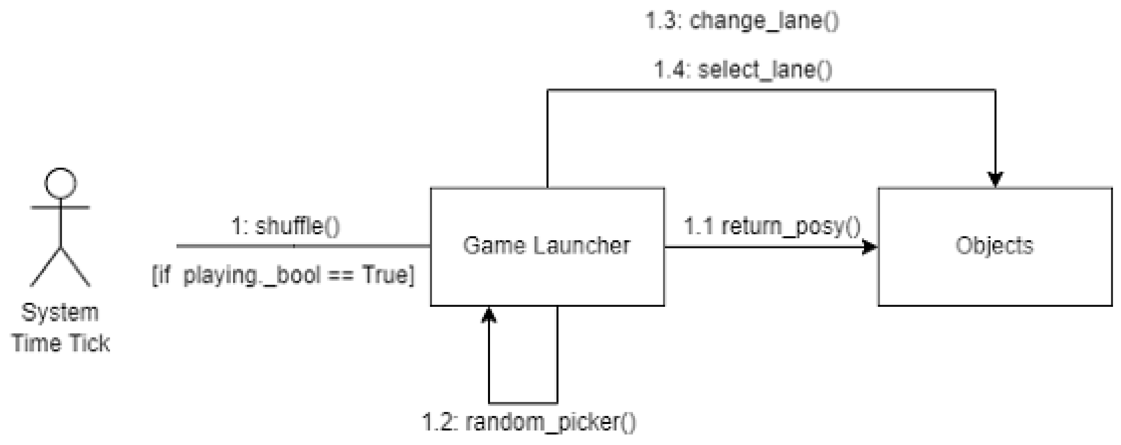


Another use case of the Player includes moving the car to the left lane. When the player presses the “D” button on the keyboard, *move_car_right* method executes and Game Launcher calls *move_right* method from Car class to send a message to the car. The *move_right* method moves the car to the right lane till there is no lane to the right of the car. There is a condition for *move_car_right* method: -remote_control attribution of the Game Launcher object must be false. Otherwise, controls would be reversed as mentioned below.

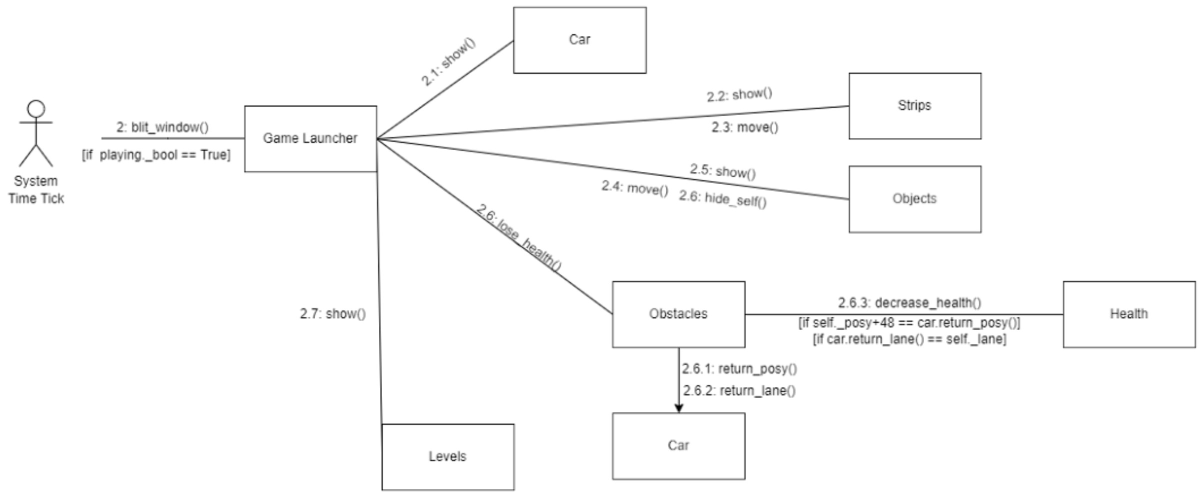

PLAYER
[Press A]
1 : remote_control_move_left()
[if remote_control == True]
Game Launcher
1.1 : remote_control_move_left()
Car
1.1.1: get_score()
[if car_posy == object.posy]
Coin
1.1.2: get_fuel()
[if car_posy == object.posy]
Fuel
1.1.3: lose_health()
[if car_posy == object.posy]
Obstacles

The next sequence of messages that is related to move right use case of the Player also includes a challenge for the player. The system is the same as the basic movement functions above. However, this time pressing the “A” button makes the car move right if *remote_control* attribution of the Game Launcher object is true.

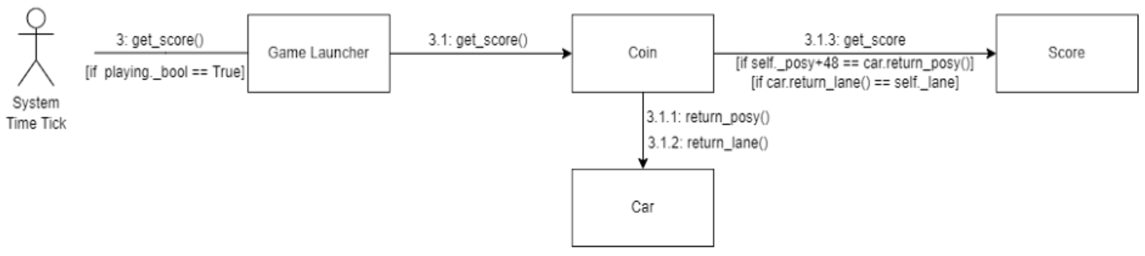
4) System: Time Tick



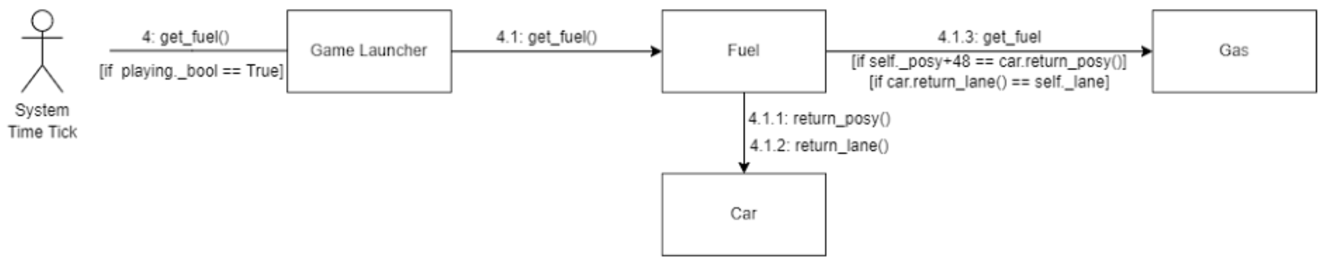
The first message sent by time tick use case is executed by the *shuffle* method of Game Launcher class. After that, objects return their y-position to Game Launcher class in order it to check if they reached to the bottom of the window. If this is the case, the *random_picker* method is called on Game Launcher, which prepares a new shuffled x-position list for the flowing objects (coin, fuel, tree, and hole) when they reach the bottom of the window. After that, they are made to select a new lane and change their existing -lane attribution accordingly. By doing this, objects appear in a different lane from the top to the bottom of the window in the next view.



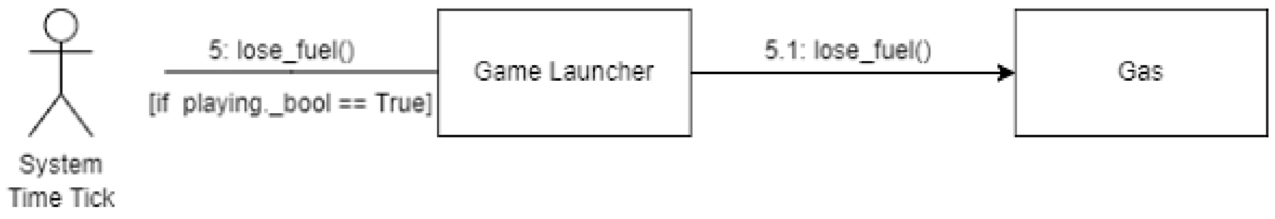
The second message sent by time tick use case is executed by the *blit_window* method of Game Launcher class. This method makes all the objects visible on the screen via *show* methods of them. The *show* and *move* methods are executed on strips, objects and obstacles respectively. The Levels and Car classes only *show* methods as they do not move, while other objects flow from the top to the bottom of the window. Objects, in general, execute *hide_self* method as they might collide with the car. Also the *return_posy* and *return_lane* methods are called from the Car class to check whether the car collides with an obstacle or not, after calling the lose-health method of the Obstacles class. If the position requirements are met, the *decrease_health* method is called on Obstacles to decrease the health amount.



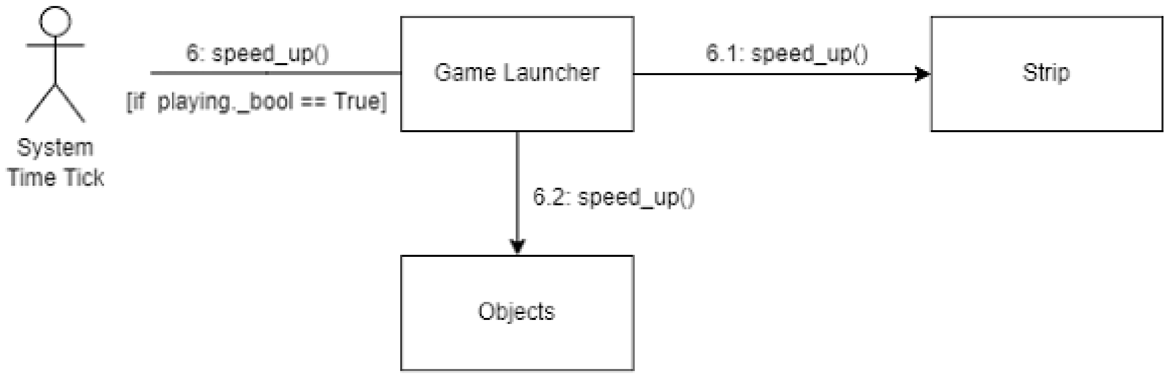
The third message sent by time tick use case is executed by the *get_score* method of Game Launcher class. After that, *the get_score* method is called in the Coin class. Then, *return_posy* and *return_lane* methods are called from Car class to check if the coin object and the car have been crushed. If this is the case, the *get_score* method is called on the Score class to increase the score value.



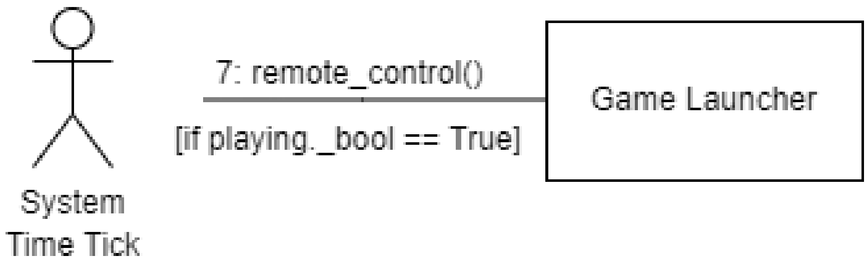
The fourth message sent by time tick use case is executed by the *get_fuel* method of the Game Launcher class. The mechanism of this diagram is very similar to the *get_score* method. This time, the Fuel class executes the *get_fuel* method. Then, position requirements are asked to the Car class to check if it is hit to a fuel object or not. At the end, the *get_fuel* method is executed if the requirements are met.



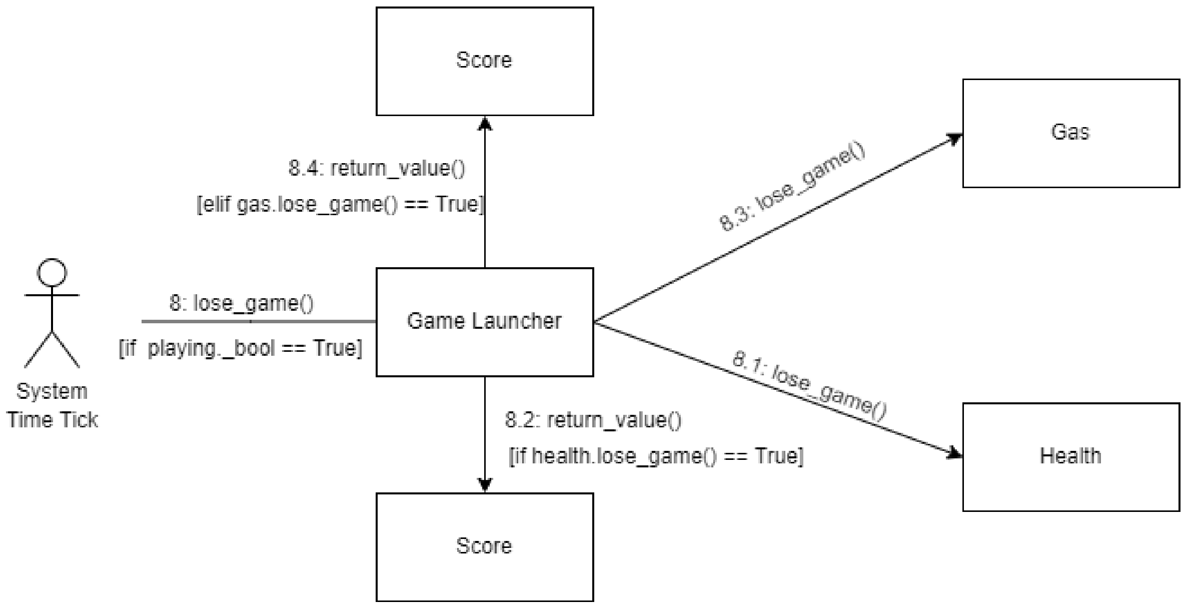
The fifth message sent by time tick use case is executed by the *lose_fuel* method of the Game Launcher class. When the method is called, *lose_fuel* method is called from the Gas Class, to reduce the amount of fuel left by the time.



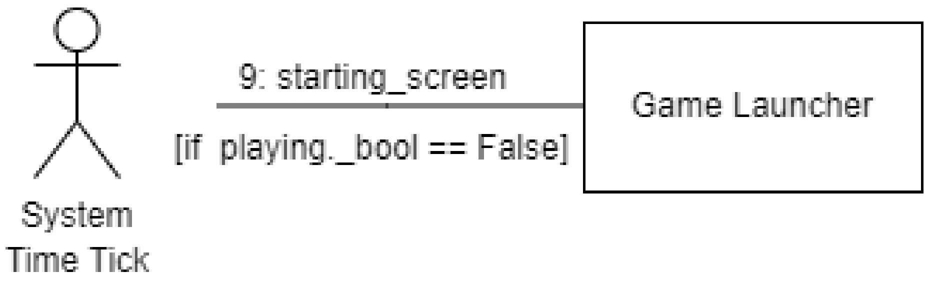
The sixth message sent by time tick use case is executed by the *speed_up* method of the Game Launcher class. This *speed_up* method calls other *speed_up* methods from Strip and Objects classes. By those methods, the vertical speed of the moving objects increases, making the game more challenging at every last 10 seconds of each level.



The seventh message sent by time tick use case is executed by the *remote_control* method of the Game Launcher class. This method changes the state of the control mechanism. If executed, the functions of control keys is reversed.



The eighth message sent by time tick use case is executed by the *lose_game* method of the Game Launcher class. Other *lose_game* methods are called from Health and Gas classes respectively. Required information is taken from Gas and Health classes. Then, the *return_value* method is executed from the Score class twice. One in order to check whether the health value is zero and other to check whether the gas amount is zero. This whole sequence is done in two phases. First, checking the health value then checking the gas amount. If one of those is zero. Current score of the player is printed on the screen via *return_value* method.



The ninth message sent by time tick use case is executed by the *starting_screen* method of the Game Launcher class. This method is only executed at the beginning of the game. When the player presses the “P” button, this method no longer works.

User Documentation

- Press “P” → If you are ready, press “P” to start game.

- Press “A” → If you want the car to switch to left lane, press “A”.

- Press “D” → If you want the car to switch to right lane, press “D”.

- You should change lanes to avoid hitting tree and hole obstacles and save your life. Remember you only have 3 lives.

- You should check the gas level regularly, collect the fuel emojis as the gas level gets low.

- You are expected to collect as many coins as possible to achieve the highest score in the game.

- As the game progresses, the level will increase every 30 seconds, the car will accelerate and hence, the game will become more difficult.

- !! Attention !! The last 10 seconds of each level, control keys will swap their functions. When this happens, press "D" button to switch to left lane and press "A" button to switch to right lane. You are not able to know when this is going to be happen, so be careful.