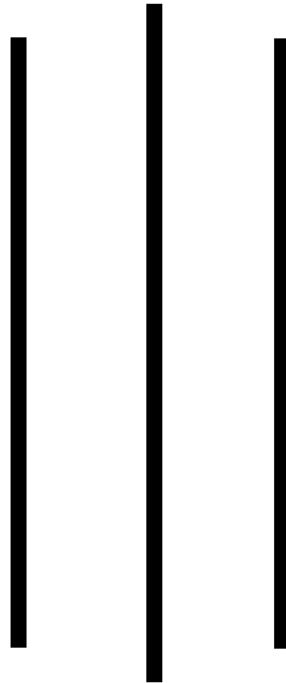
FINAL PROJECT SOFTWARE ENGINEERING



Luisana ALVAREZ MONSALVE

Xavier BELTRAN URBANO

Andrew Dwi PERMANA

ERASMUS MUNDUS JOINT MASTER DEGREE IN MEDICAL IMAGING AND APLICATIONS (MAIA)

Université de Bourgogne

2022/2023

Table of Contents

I. PROJECT 1
I.1 DEFINING THE PROJECT 1
I.2 DEFINING CLASSES 2
I.2.a Class Traffic 2
I.2.b Class Traffic_Light 17
I.2.c Class Vehicles 19
I.2.d Class Pedestrian 21
II.2 RESULT 25
II.3 CONCLUSION 28

List of Figures

Figure 68. Screen capture of the simulation 25
Figure 69. Screen capture of the infraction and its message of vehicle 26
Figure 69. Screen capture of the infraction and its message of pedestrian 26

List of Tables

Table 1. Explanation of attributes "def __init__" in class Traffic 2

I. PROJECT

I.1 DEFINING THE PROJECT

A traffic light, which may also be referred to as a traffic signal, is a device that is used to regulate road traffic amongst different types of road users, including pedestrians, vehicles, and other road users. Traffic signals for motor vehicles are typical of the three-colour variety, and they may or may not include directional arrows. Pedestrian signals often come in two different colours and depict a silhouette of a person walking. The recreation of a bicycle that is found on traffic lights that are designated for bikers can easily be recognised. The lights typically progress from two primary colours, with red representing a stop position and green representing an open one.

The goal of this project is to develop, in Python, the operation of a traffic light that also includes a pedestrian signal and to identify violations committed by drivers. Some tasks need to be fulfilled in this project, such as:

1. Simulate the operation of a traffic light
2. Simulate the operation of a pedestrian light
3. Detect if a car is in violation or not
4. Detect if a pedestrian is in violation
5. Display a message if there is a violation

The module that we have used in this project is mainly Pygame. Pygame is a set of Python modules that were developed specifically to build video games. Pygame is an extension that adds capabilities to the already powerful SDL library, which stands for Simple DirectMedia Layer. SDL provides cross-platform access to your system's underlying multimedia hardware components, such as sound, video, mouse, keyboard, and joystick. This enables to write of full-featured games and multimedia programmes in Python. Pygame makes use of the SDL package to enable real-time computer game production without the low-level mechanics of the C programming language and its derivatives. This is based on the notion that the most expensive functions inside games may be abstracted from the game logic, allowing the game to be structured using a high-level programming language such as Python.

I.2 DEFINING CLASSES

In this project, we are classifying several classes that will generate objects in the simulation, namely Traffic, Vehicle, Traffic Light, and Pedestrian.

I.2.a Class Traffic

This class is used to simulate the real traffic of both pedestrians and vehicles at an intersection. In the simulation, we will observe randomly the movement of the vehicles, both cars, and motorbikes, and also pedestrians in the intersections with the traffic lights. We are able to stimulate the traffic flow, in this case, freeway junctions with the vehicles and pedestrians. In addition to that, we added some infractions of both vehicles and pedestrians in the traffic flow.

In this section, we want to explain the coding behind the traffic simulation. Table 1 will explain the attributes in the *init* function as the initializing of some attributes in this class. The attributes and methods of the Traffic class are as follows:

Table 1. Explanation of attributes "def __init__" in class Traffic

No.	ATTRIBUTES	EXPLANATION
1.	*vecVehicles*	*vector of Vehicle()* *This is representing the vector that stores all the vehicles of the traffic.*
2.	*copy_vecVehicles*	*vector of Vehicle()* *This vector is a copy of the initial vecVehicles vector. It is used to add new vehicles in the same initial position as the vehicles in the vecVehicles vector.*
3.	*vec_Traffic_Light*	*vector of Traffic_Light()* *This is representing the vector that stores all the vehicles Traffic_Lights of the traffic.*
4.	*vec_Traffic_Light_Pedestrian*	*vector of Traffic_Light for pedestrian()* *This is representing the vector that stores all the pedestrians Traffic_Lights of the traffic.*
5.	*vec_Pedestrian*	*vector of Pedestrian()* *This is representing the vector that stores all the pedestrians in the traffic.*
6.	*stop_lines*	*vector of points (x,y)* *This vector contains the coordinates of the points where the vehicles must come to a halt.*

7.	*tl_position*	*int* *This variable is used to regulate which traffic light turns green.*
8	*infraction*	*boolean* *This attribute is a type of Boolean.* *This is used to know when the infractions are done (default False).*
9.	*coordx*	*int* *This variable is used to store a message for display (when an infraction is done) in the "x" corresponding position (default 0).*
10.	*coordy*	*int* *This variable is used to store a message for display (when an infraction is done) in the "y" corresponding position (default 0).*
11.	*rot_points*	*vector of points (x,y)* *This is representing the vector that stores the position of the points at which the vehicles have to turn.*

In this section, we are presenting the essential codes for the functions that we have used for this simulation.

```
def display_Vehicles(self):
    for Vehicle in self.vecVehicles:
        Vehicle_figure = pygame.image.load(Vehicle.return_image())
        if Vehicle.rotating_direction != '':
        Vehicle_figure = Vehicle.return_rotated_image(Vehicle.rotating,
           Vehicle.rotating_direction)
        rect = Vehicle_figure.get_rect()
        self.screen.blit(Vehicle_figure, (
        Vehicle.posx - rect.width / 2, Vehicle.posy - rect.height / 2))

def delete_Vehicles(self):
    for Vehicle in self.vecVehicles:
        posx, posy = Vehicle.return_position()
        if posx > 859 or posy > 703 or posx < 0 or posy < 0:
                self.vecVehicles.remove(Vehicle)

def add_another_Vehicle(self):
    t = threading.Timer(2, self.add_another_Vehicle)
    for i in range(len(self.copy_vecVehicles)):
        type_vehicle = random.choice([1, 2, 3, 4])
        x, y = self.copy_vecVehicles[i].return_position()

        Vehicle1 = Vehicle(x, y,
        self.copy_vecVehicles[i].return_direction(), type_vehicle)
```

```
        if not self.check_other_Vehicles(self.copy_vecVehicles[i], 0):
            self.vecVehicles.append(Vehicle1)
    t.start()
```

def display_Vehicles(self). This function is to display the vehicles both car and motorbike in "**self.vecVehicles**" vectors in the background. We place the vehicles in the center point in order to make a rotation for the image of the vehicles once they are at the edge of the turning line and decided to do a turning. Since we built up our window screen size, once the vehicles reach the end of the screen, we remove them so that they are not considered in other functions. Otherwise, when they are no longer visible on the screen, they are still stored in the vector of cars and are iterated in each iteration of the loop. To accomplish that task, we utilised the function **def delete_Vehicles(self)**.

The following function **def add_another_Vehicle(self)** adds vehicles to the simulation at random, utilising the attribute **copy_vecVehicles** to use the same initial position as the vehicles in the vector **vecVehicles**. Moreover, we utilised the method **check_other_Vehicles** in order to check if the vehicle can be added or moved. On the one hand, when the variable "value" is equal to 0, we will check if there is any vehicle in the position we want to add the vehicle to (we will return True if there is a vehicle in that position or False otherwise). On the other hand, when the variable "value" is equal to 1, we will check if there is any vehicle in front of the vehicle we are looking at. If there is no vehicle in front of the vehicle, we will return True, and the vehicle will be moved. Otherwise, we will return False and the vehicle will remain motionless.

```
def move_Vehicles(self):

    for Vehicle in self.vecVehicles:
        Vehiclex, Vehicley = Vehicle.return_position()

    # Vehicle 1
        if Vehicle.direction == "right":
             sx, sy = self.stop_lines[0]
             if Vehiclex > sx - 5:
                if self.vec_Traffic_Light[0].isRed() and
                   Vehicle.isInfractor() == True:
                   self.Vehicle_Infractor_Changes(Vehicle)
             if Vehiclex == sx:
```

```
                Vehicle.decide_direction()
            if Vehiclex < sx and (not self.check_other_Vehicles(Vehicle,
                1)):
                Vehicle.change_position()
        else:
                self.check_position_vehicle(Vehicle)

def check_other_Vehicles(self, Vehicle, value):
    direction=Vehicle.return_direction_copy()
    Vehicle_direction = self.vec_same_direction(direction)
    Vehiclex, Vehicley = Vehicle.return_position()
    distance = 80
    if value==1:
        index= Vehicle_direction.index(Vehicle)
        if index==0:
            return False
        else:
            vex, vey = Vehicle_direction[index-1].return_position()
            if (direction == "right" and Vehiclex + distance > vex) or
            (direction == "down" and Vehicley + distance > vey) or
            (direction == "left" and Vehiclex - distance < vex) or
            (direction == "up" and Vehicley - distance < vey):
                return True
    else:
        if len(Vehicle_direction)==0:
            return False
        else:
            vex, vey = Vehicle_direction[len(Vehicle_direction) -
            1].return_position()
            if (direction == "right" and Vehiclex + distance > vex) or
                (direction == "down" and Vehicley + distance > vey) or
                (direction == "left" and Vehiclex - distance < vex) or
                (direction == "up" and Vehicley - distance < vey):
                return True
    . . .
```

For the movement of the different vehicles, in this simulation, we utilised the **def move_Vehicles(self)** function. This function is one of the most crucial parts of this simulation, also it is the main base for the commute of the vehicles, which is in this project, we are using five different pictures of the vehicles in order to give variability and a more realistic appearance to the project. This function contains all the logic behind the moving vehicles in the simulation. To make the report simple and short, we will illustrate only one direction, the left direction as presented in the above code.

Further function, namely **def check_position_vehicle(self, Vehicle)** in the argument, we have Vehicle, this attribute contains the vehicle that we want to check. We gave the condition to check the pedestrian when the position of the vehicles is after the stop lines and in front of the zebra crossing and detect the pedestrian.

```
def check_position_vehicle(self, Vehicle):
     direction = ["right", "down", "left", "up"]
     i = direction.index(Vehicle.return_direction())
     sx, sy = self.stop_lines[i]
     Vehiclex, Vehicley = Vehicle.return_position()
     if self.vec_Traffic_Light[i].isGreen():
             Vehicle.change_position(
     else:
          if (Vehicley < sy and Vehicle.return_direction() == "up") or
             (Vehiclex < sx and Vehicle.return_direction() == "left"):
             if (Vehicley == 286 and Vehicle.direction == "up") or (
                     Vehiclex == 352 and Vehicle.direction == "left"):
                self.detect_pedestrians(Vehicle)
          Vehicle.change_position()
     elif (Vehicley > sy and Vehicle.return_direction() == "down")or
             (Vehiclex > sx and Vehicle.return_direction() == "right"):
                 if (Vehicley == 406 and Vehicle.direction == "down") or
                        (Vehiclex == 484 and Vehicle.direction == "right"):
                self.detect_pedestrians(Vehicle)
          Vehicle.change_position()
```

In addition, when a vehicle reaches a stop line, it will decide randomly whether to continue straight, or turn left or right (see **def move_Vehicles(self)**). If the vehicle decides to turn, the turning direction will be memorized. Using the function **def set_turning_parameters(self, Vehicle)** and knowing the turning direction and the current direction of the vehicle, the rotating

point and the vehicle direction after turning are defined. For efficiency purposes, we presented not the full complete of the code.

```
def set_turning_parameters(self, Vehicle):
      if Vehicle.return_direction() == "right":
         if Vehicle.rotating_direction == "right":
            rot_point = self.rot_points[0]
            next_dir = "down"
         elif Vehicle.rotating_direction == "left":
            rot_point = self.rot_points[1]
```

The following code corresponds to the turning movement of the vehicle in the function **`def move_Vehicles(self)`**. If the vehicle decided to turn, as it was explained before, the turning parameters will be settled, and once the vehicle reaches the turning point, the rotation will start. The rotation lasts 10 iterations, during which the vehicle will move in x and y. In the last iteration, the "direction" attribute of the vehicle will be changed to the new direction, and the rotation will end.

```
      if Vehicle.rotating_direction != '':
                 rot_point, next_dir = self.set_turning_parameters(Vehicle)
                 if (Vehicle.return_direction() == "right" or
                        Vehicle.return_direction() == "left") and
                        Vehicle.posx == rot_point:
                     Vehicle.start_rotation()
                 if (Vehicle.return_direction() == "up" or
                     Vehicle.return_direction() == "down") and Vehicle.posy
                     == rot_point:
                     Vehicle.start_rotation()

                 i = Vehicle.rotating

                 if i < 10 and i > 0:
                     Vehicle.rotate(
```

```
                Vehicle.rotating_direction)
            Vehicle.start_rotation()
            i = Vehicle.rotating
            if i == 10:
                Vehicle.change_direction(
                    next_dir)
                Vehicle.end_rotation()
```

The next function, which is **`def detect_pedestrians(self, vehicle),`** is used to check the pedestrian in the zebra crossing by the vehicles when they decide to turn after the traffic light. The variable "Vehicle" is a vehicle object that is near the zebra crossing after turning and it needs to check whether no pedestrians are crossing before continue driving. This function's important in order to detect the pedestrian while the vehicles are about to continue driving after turning, so it will avoid overlapping or crushing between two user's road. The condition is when the vehicle detects the pedestrian after turning, then they need to stop according to the stop points after the traffic light until the pedestrian passes. We initiated the detected value equal to zero as no pedestrian crossing and equal to one when there is a pedestrian crossing.

```
def detect_pedestrians(self, vehicle):
        detected = 0
        if self.vec_Traffic_Light_Pedestrian[5].isGreen():
            if vehicle.direction == "right":
                for pedestrian in self.vec_Pedestrian:
                    if pedestrian.state == "crossing" and pedestrian.posx
                        detected = 1
            if vehicle.direction == "left":
                for pedestrian in self.vec_Pedestrian:
                    if pedestrian.state == "crossing" and pedestrian.posx
                        detected = 1
        if self.vec_Traffic_Light_Pedestrian[7].isGreen():
            if vehicle.direction == "up":
                for pedestrian in self.vec_Pedestrian:
                    if pedestrian.state == "crossing" and pedestrian.posy
                        detected = 1
            if vehicle.direction == "down":
                for pedestrian in self.vec_Pedestrian:
                    if pedestrian.state == "crossing" and pedestrian.posy
                        detected = 1
        vehicle.save_detected(detected)
```

The next thing we did in this project is to display the traffic lights both for the vehicles and pedestrians. To accomplish this task, we use the function **def display_Traffic_Light**, the main task of which consists of displaying the traffic lights that we stored in "**self.vec_Traffic_Light**" for the traffic light for vehicles and "**self.vec_Traffic_Light_Pedestrian**" vectors for the pedestrian on the background.

```
def display_Traffic_Lights(self):
    for Traffic_Light in self.vec_Traffic_Light:
        image = Traffic_Light.return_image()
        Tlight_figure = pygame.image.load(image)
        self.screen.blit(Tlight_figure, (Traffic_Light.posx,
       Traffic_Light.posy))

    for Traffic_Light in self.vec_Traffic_Light_Pedestrian:
        image = Traffic_Light.return_image()
        Tlight_figure = pygame.image.load(image)
        self.screen.blit(Tlight_figure, (Traffic_Light.posx,
       Traffic_Light.posy))
```

After we placed the traffic lights according to their places, then, we described the algorithm for changing its colours. For those, we utilised the functions **def change_colour_Traffic_Light** and **def change_Traffic_Light_pedestrian** for the traffic light for vehicles and pedestrians respectively. For the changing colour in the traffic light for the vehicle, we described two functions: the first function **def change_colour_Traffic_Light(self)** changes the colour of the current green traffic light to red and it also changes the colour of the next one to green (after 2 seconds we call the other function through a timer). The second function **def change_Traffic_Light(self)** is only used to change the colour from green to yellow and to set a timer of 3 seconds (when this timer ends, the other function will be called). These processes will be happening recursively while the program is running, simulating the traffic lights on the road for both vehicles and pedestrians.

```
def change_colour_Traffic_Light(self):
        self.vec_Traffic_Light[self.tl_position].change_color("red")
        if self.tl_position < (len(self.vec_Traffic_Light) - 1):
            self.vec_Traffic_Light[self.tl_position +
            1].change_color("green")
            self.tl_position += 1
```

```
        else:
            self.vec_Traffic_Light[0].change_color("green")
            self.tl_position = 0

        self.change_Traffic_Light_pedestrian()
        t = threading.Timer(3, self.change_Traffic_Light)
        t.start()

def change_Traffic_Light(self):
        self.vec_Traffic_Light[self.tl_position].change_color("yellow")
        t = threading.Timer(3, self.change_colour_Traffic_Light)
        t.start()
```

The pedestrian traffic light operates in such a way that all of them change at the same time, first and foremost, we defined a vector called "odd". This vector contains the traffic lights that are always the same colour, thus in each iteration, we must change those to one colour and the others to the other. To begin, we colour the traffic lights of the vector's odd position in red. The variable "**`self.tl_position`**", which contains the current position of the vehicle traffic light with green colour, will be checked each time we need to modify the colour of the traffic lights since we will change the pedestrian traffic lights to green or red depending on the value of that variable. This code was constructed to have the corresponding traffic lights (in order to avoid hitting any pedestrians) in red when the cars moving are those with the direction "left or right" (horizontal). The same scenario for the vertical ones, turning red the corresponding traffic lights when a car is moving in an "up or down" direction.

```
def change_Traffic_Light_pedestrian(self):
    odd = [0, 1, 4, 5]
    for i in range(len(self.vec_Traffic_Light_Pedestrian)):
        if self.tl_position % 2 == 0:
            if i in odd:
                self.vec_Traffic_Light_Pedestrian[i].change_color("red")
            else:
                self.vec_Traffic_Light_Pedestrian[i].change_color("green")
        else:
            if i in odd:
                self.vec_Traffic_Light_Pedestrian[i].change_color("green")
            else:
                self.vec_Traffic_Light_Pedestrian[i].change_color("red")
```

The next road user which is Pedestrian, we describe is almost similar to Vehicles. Firstly, we wanted to display on the screen from the saved vector in "**self.vec_Pedestrian**". Two different images are displayed alternatively to simulate that the pedestrian is walking. To see the change in the movement, the image is changed every ten iterations. For this, we utilised the function **def display_pedestrians**.

```
def display_pedestrians(self):
    for pedestrian in self.vec_Pedestrian:
            if self.iteration == 10:
                if pedestrian.stop == 0:
                    pedestrian.change_image()
                self.iteration = 0
            else:
                self.iteration += 1
            pedestrian_figure =
            pygame.image.load(pedestrian.return_image(pedestrian.i))
            self.screen.blit(pedestrian_figure, ( pedestrian.posx,
            pedestrian.posy))
```

For the movement of the pedestrian, we initiated them according to the pedestrian way, which means since they will not only walk towards the zebra crossing but also, they can decide if they want to continue their journey without crossing the zebra crossing. We need also to create the condition, where they are at the corners, they always turn whether right or left, depending on the pathway, then when they are at the zebra crossing, they need to decide whether to cross or not randomly. Then, if the pedestrian decided to cross the zebra crossing, we constructed the condition that they need to decide randomly whether turn left or right after crossing.

```
def move_pedestrians(self):
    for pedestrian in self.vec_Pedestrian:
        x, y = pedestrian.return_position()
           for i in range(len(self.pedestrian_corners)):
                if self.pedestrian_corners[i] == (x, y):
                    pedestrian.change_direction(i, "corner")

            if pedestrian.state == "crossing":
                if ((pedestrian.direction == "right" and x == 457) or (
                          pedestrian.direction == "left" and x == 388))
                          and pedestrian.already_crossed == 0:
                    pedestrian.change_direction(0, "after crossing")
                    pedestrian.crossed(1)
                elif ((pedestrian.direction == "up" and y == 309) or (
```

```
            pedestrian.direction == "down" and y == 378))
            and pedestrian.already_crossed == 0:
        pedestrian.change_direction(0, "after crossing")
        pedestrian.crossed(1)
```

Further movement, after they cross the zebra crossing and every time a pedestrian gets to another zebra crossing, this condition is checked to avoid them to cross it infinitely. The pedestrian has two possible statuses: "walking" and "crossing". The pedestrian will be "walking" until it reaches a zebra crossing, where he will decide randomly whether to cross or not. If he decides to cross, the status will change to "crossing". While the pedestrian is "crossing" he will follow the traffic light (if he is not an infractor). After crossing, he will change the state to "walking" and continue their journey.

```
for i in range(len(self.zebra_cross)):
    if self.zebra_cross[i] == (x, y):
       if pedestrian.already_crossed == 0:
          if pedestrian.state == "walking":
             if pedestrian.decide_direction() == 1:
                pedestrian.change_state("crossing")
                pedestrian.change_direction(i, "before crossing")
                        if pedestrian.state == "crossing":
                            if self.check_traffic_light(i).isGreen():
                                pedestrian.change_stop(0)
                            else:
                                if pedestrian.infractor == 0:
                                    pedestrian.change_stop(1)
                                else:
                                    pedestrian.change_stop(0)
       elif pedestrian.already_crossed == 1:
            pedestrian.change_state("walking")
            pedestrian.crossed()

pedestrian.change_position()
```

The same situation with the vehicles, after certain points, if the journey will pass to the edge of the screen window, we need to remove the pedestrian in order to dismiss the pedestrian from the display screen, moreover, we wanted to generate a new one afterward, the position of the new pedestrian will be chosen randomly between the positions of the initial pedestrians, to do so, we utilised the function **`def delete_Pedestrians(self)`**. First, we will check if the position of the pedestrian is on one of the edges of the screen, if so, it will be removed, then we will create

a new one from the original vector of the pedestrian that we stored in "**self.vec_Pedestrian_copy**".

```
def delete_Pedestrians(self):
        for Pedestrian in self.vec_Pedestrian:
            posx, posy = Pedestrian.return_position()
            if posx > 859 or posy > 703 or posx < 0 or posy < 0:
                self.vec_Pedestrian.remove(Pedestrian)
                Pedestrian1 = copy.copy(random.choice(
                    self.vec_Pedestrian_copy))
                self.vec_Pedestrian.append(Pedestrian1)
```

After we finished with the main simulation, we created an infraction in order to fulfill the task for this project. The infraction will happen to all road users, vehicles and pedestrian. First, the infraction for the vehicles, every 10.5 seconds, a pygame event will be executed, when this is ongoing, we will call the function **def find_vehicle_infraction.** This function will choose the next infractor vehicle randomly (we choose the closest one to the traffic light in order not to cause accidents). For the infraction in pedestrian, this function **def find_ pedestrian _infraction** will be executed every 5 seconds. When it is executed, it looks for a pedestrian that at that moment is stopped at a zebra cross waiting because the light is red. If it finds one, it will become an infractor and will cross even if the light is red. Then, if it does not find any pedestrian in this situation, there will not be any infraction.

When the infraction is performed, a message will pop up on the screen. We settled up the position of the pop-up message close to the stop lines or zebra crossing where the infraction occurred, it specified in the function **def Vehicle_Infractor_Changes(self, Vehicle)** and **def pedestrian_infraction_changes(self, pedestrian).**

```
def find_vehicle_infraction(self):
        position = random.choice([0, 1, 2, 3])
        direction = ["right", "down", "left", "up"]
        vec_vehicles_all = self.vec_same_direction(direction[position])
        vec_vehicles_only =
        self.only_traffic_light_vehicle(vec_vehicles_all)
        if len(vec_vehicles_only) != 0:
            vehicle = vec_vehicles_only[0]
            posx, posy = vehicle.return_position()
            if posx >= 245 and vehicle.return_direction() == "right":
```

```
                vehicle.become_infractor()
            elif posy >= 175 and vehicle.return_direction() == "down":
                vehicle.become_infractor()
            elif posx <= 585 and vehicle.return_direction() == "left":
                vehicle.become_infractor()
            elif posy <= 511 and vehicle.return_direction() == "up":
                vehicle.become_infractor()

def Vehicle_Infractor_Changes(self, Vehicle):
        self.vehicle_infractor = Vehicle
        Vehicle.change_infractor_status()
        Vehicle.change_position()
        self.infraction = True
        if Vehicle.return_direction() == "right":
            self.coordx = 50
            self.coordy = 500
        elif Vehicle.return_direction() == "left":
            self.coordx = 550
            self.coordy = 75
        elif Vehicle.return_direction() == "down":
            self.coordx = 40
            self.coordy = 100
        elif Vehicle.return_direction() == "up":
            self.coordx = 575
            self.coordy = 500


def find_pedestrian_infraction(self):
        for pedestrian in self.vec_Pedestrian:
            x, y = pedestrian.return_position()
            for i in range(len(self.zebra_cross)):
                if self.zebra_cross[i] == (x, y) and
                self.check_traffic_light(i).isGreen() == False:
                    pedestrian.become_infractor()
                    self.pedestrian_infraction_changes(pedestrian)
                    break

def pedestrian_infraction_changes(self, pedestrian):
        self.pedestrian_infraction = True
        posx, posy = pedestrian.return_position()
        for i in range(len(self.zebra_cross)):
            if self.zebra_cross[i] == (posx, posy):
                break
        if i == 6 or i == 7:
            self.coordx2 = 50
            self.coordy2 = 500
        elif i == 2 or i == 3:
            self.coordx2 = 550
            self.coordy2 = 75
        elif i == 0 or i == 1:
```

```
            self.coordx2 = 40
            self.coordy = 100
        elif i == 4 or i == 5:
            self.coordx2 = 575
            self.coordy2 = 500
```

After the infraction was performed and the message displayed, we defined the timer to remove the infractor vehicle or pedestrian and the displayed message after 3 seconds with the function **`def remove_vehicle_infractor(self)`** and **`def remove_pedestrian_infractor`**. The infractor vehicles are removed without replacement, while the infractor pedestrian is eliminated and replaced with a new pedestrian placed randomly in one of the positions of the initial pedestrians.

```
def display_infraction_message(self, type):
        if type == "vehicle":
            infraction = pygame.image.load(
                "D:/MAIA 7/STUDY/uB/Software Engineering/TD/Traffic Light
                 Project/images/infraction.jpeg")
            infraction = pygame.transform.scale(infraction, size_image)
            self.screen.blit(infraction, (self.coordx, self.coordy))
            if self.iteration == 0:
                t4 = threading.Timer(3, self.remove_vehicle_infractor)
                t4.start()
                self.iteration += 1
        elif type == "pedestrian":
            infraction = pygame.image.load(
                "D:/MAIA 7/STUDY/uB/Software Engineering/TD/Traffic Light
                 Project/images/infraction.jpeg")
            self.screen.blit(infraction, (self.coordx2, self.coordy2))
            if self.pedestrian_iteration == 0:
                t5 = threading.Timer(3, self.remove_pedestrian_infractor)
                t5.start()
                self.pedestrian_iteration += 1

def remove_vehicle_infractor(self):
        self.iteration = 1
        if self.vehicle_infractor in self.vecVehicles:
            self.vecVehicles.remove(self.vehicle_infractor)
        self.infraction = False

def remove_pedestrian_infractor(self):
        self.pedestrian_iteration = 0
        for pedestrian in self.vec_Pedestrian:
            if pedestrian.infractor == 1:
                self.vec_Pedestrian.remove(pedestrian)
```

```
            Pedestrian1 = copy.copy(random.choice(
                self.vec_Pedestrian_copy))
            self.vec_Pedestrian.append(Pedestrian1)
    self.pedestrian_infraction = False
```

Finally, in class Vehicle, we established the main screen function, this function consists of the simulation window's screen width and size, the background and title or caption to be displayed, initialising pygame function, and the infinite loop of the simulation. We created a customised event to create randomly an infraction both for vehicles and pedestrian and it will appear on the event queue every 10.5 seconds and 5 seconds for vehicles and pedestrian respectively. Also, we settled up the timer to add a new vehicle every 4 seconds and to change the color of the traffic lights every 5 seconds. After that, we execute an infinite loop that keeps updating our simulation screen. We start by defining the exit criteria within the loop.

```
def main_screen(self):
        screenWidth = 859
        screenHeight = 703
        screenSize = (screenWidth, screenHeight)


        pygame.init()
        self.screen = pygame.display.set_mode(
            screenSize)
        background = pygame.image.load(
            'D:/MAIA 7/STUDY/uB/Software Engineering/TD/Traffic Light
        Project/images/background2.png')
        self.screen.blit(background, (0, 0))
        pygame.display.set_caption("SIMULATION")
        INFRACTION = pygame.USEREVENT + 1
        PEDESTRIAN_INFRACTION = pygame.USEREVENT + 2
        pygame.time.set_timer(INFRACTION, 10500)
        pygame.time.set_timer(PEDESTRIAN_INFRACTION, 5000)
        t1 = threading.Timer(4, self.add_another_Vehicle)
        t1.start()
        t2 = threading.Timer(5, self.change_Traffic_Light)
        t2.start()

        # Infinite loop
        while True:
            self.screen.blit(background, (0, 0))
```

```
self.screen.blit(pygame.image.load("D:/MAIA
7/STUDY/uB/Software Engineering/TD/Traffic Light
Project/images//logoUB.png"),(10, 10))
self.screen.blit(pygame.image.load("D:/MAIA
7/STUDY/uB/Software Engineering/TD/Traffic Light
Project/images/MAIA_logo.png"),(750, 10))

for event in pygame.event.get():
    if event.type == pygame.QUIT:
        pygame.quit()
        sys.exit()
    if event.type == INFRACTION:
        print("AN INFRACTION IS GOING TO BE DONE!")
        self.find_vehicle_infraction()
    if event.type == PEDESTRIAN_INFRACTION:
        self.find_pedestrian_infraction()

self.display_Traffic_Light()
self.display_pedestrians()
self.move_Vehicles()
self.move_pedestrians()
self.delete_Vehicles()
self.delete_Pedestrians()
self.display_Vehicles()
if self.infraction or self.pedestrian_infraction:
    if self.infraction == True:
        self.display_infraction_message("vehicle")
        pygame.display.update()
    if self.pedestrian_infraction == True:
        self.display_infraction_message("pedestrian")
        pygame.display.update()

pygame.display.update()
```

I.2.b Class Traffic_Light

This class is used to define all the functions that will be used in the class. In this project, we use two different traffic lights according to the road user, namely for vehicles and pedestrians at an intersection. We are using three colours of traffic signals for vehicles, which are red, yellow, and green. Then, for the pedestrian traffic flight, we use only two traffic signals, red and green.

Firstly, we initialised the several attributes of this class in the `def __init__` function. After that, some functions of this class are presented. The first function we will explain is the one called `def return_image(self)`. The purpose of this function is very simple, mainly consist

of returning the path of the corresponding function of the traffic light we are looking at (we storage all the images in a folder and each time we have to change the colour of the traffic light, we called this function to return the path of the image that contains the traffic light with the corresponding colour). To not extend the length of this report, only the section related to the colour "green" of the above-mentioned function has been presented.

```
def return_image(self):
    if self.type == 0:
       if self.color == "green":
          if self.position == 1:
             return 'D:/MAIA 7/STUDY/uB/Software Engineering/TD/Traffic
             Light Project/images/semaforos_images/green_light1.png'
          if self.position == 2:
             return 'D:/MAIA 7/STUDY/uB/Software Engineering/TD/Traffic
             Light Project/images/semaforos_images/green_light2.png'
          if self.position == 3:
             return 'D:/MAIA 7/STUDY/uB/Software Engineering/TD/Traffic
             Light Project/images/semaforos_images/green_light3.png'
          if self.position == 4:
             return 'D:/MAIA 7/STUDY/uB/Software Engineering/TD/Traffic
             Light Project/images/semaforos_images/green_light4.png'

             . . .
```

To conclude, the remaining function of this class are in charge of modifying, changing and returning the attributes such as position, colour and direction. To make the report simple and short we are not presenting the complete codes.

```
def return_position(self):
    return self.posx, self.posy

def return_direction(self):
    return self.direction

def change_color(self, color):
    self.color = color

def return_color(self):
     return self.color

def isGreen(self):
     if self.color == "green":
        return True
```

```
        else:
            return False

    def isRed(self):
        if self.color == "red":
           return True
        else:
           return False

def isYellow(self):
    if self.color == "yellow":
       return True
    else:
       return False
```

I.2.c Class Vehicles

This class is used to define the vehicles that are used in the traffic light simulation. In this project, we use two different images of vehicles, which are a car and a motorbike. The vehicles will commute from starting point until the end. The movement of the vehicles, we make it automatically. The vehicles will follow certain conditions that we already settled up. At some point, the vehicle can perform an infraction, thus we defined the vehicle as an infractor. During the infraction, a message will pop up to show that an infraction has been made and, after 3 seconds, we will remove the vehicle from the traffic in order to not disturb the traffic flow or cause accidents. So, in this class, we defined functions that will be used as well in the Traffic class. Firstly, we initialised the several attributes of this class in the `def __init__ function`. To make the report simple and short, only the function whose purpose is not modifying, changing or returning the attributes of this class, will be presented.

The function **`def_decide direction`** is used to decide randomly whether to continue straight or to turn left ot right. If the vehicle decides to turn, the turning direction will be saved in the attribute "rotating_direction" using the function **`def define_rotation(self, new_direction).`**

```
def decide_direction(self):
       direction = random.choice(["straight", "right", "left"])
       if direction != "straight":
            self.define_rotation(direction)
```

The function **`def rotate(self, dir_rotation)`** is used to change the position (x and y) of the vehicle while it is turning. The changes on x and y depend on the current direction of the vehicle and the direction it is turning to.

```
def rotate(self, dir_rotation):
        if self.return_direction == "right":
            self.posx += 1
            if dir_rotation == "right":
                self.posy += 3
            if dir_rotation == "left":
                self.posy -= 3
        elif self.return_direction == "left":
            self.posx -= 1
            if dir_rotation == "right":
                self.posy -= 3
            if dir_rotation == "left":
                self.posy += 3
        elif self.return_direction == "up":
            self.posy -= 1
            if dir_rotation == "right":
                self.posx += 3
            if dir_rotation == "left":
                self.posx -= 3
        elif self.return_direction == "down":
            self.posy += 1
            if dir_rotation == "right":
                self.posx -= 3
            if dir_rotation == "left":
                self.posx += 3
```

The following functions are important for the turning movement of the vehicle. With **`def start_rotation(self)`** the rotating iteration is started and incremented, using the variable "rotating" to memorize in which iteration we are in. When the last iteration finishes, the function **`def end_rotation(self)`** will erase the attributes "rotating" and "rotating_direction" to end the rotation and continue with a straight movement (see the function **`def move_vehicles(self)`** of the class Vehicle). Finally, the function **`def return_rotated_image(self)`** will return the image rotated an specific angle in an specific direction depending on the turning direction and the iteration we are in. In each iteration, the image will turn 9º, so after 10 iterations, the vehicle will have turn 90º.

```
def start_rotation(self):
        if self.rotating == 0:
            self.rotating = 1
        else:
            self.rotating += 1

def end_rotation(self):
        self.rotating = 0
        self.rotating_direction = ''

def return_rotated_image(self, i, dir_rotation):
        original_image = pygame.image.load(self.return_image())
        if dir_rotation == "right":
            rotated_image = pygame.transform.rotate(original_image, -9 * i)
        if dir_rotation == "left":
            rotated_image = pygame.transform.rotate(original_image, 9 * i)
        return rotated_image
```

I.2.d Class Pedestrian

Similar to the class Vehicle, this class is used to define the pedestrian that is used in the traffic light simulation. The pedestrian will commute from starting point until the end. The movement of the pedestrian, we make it automatically. The pedestrian will follow certain conditions that were already settled up. We make randomly the movement of the pedestrian when they are at the zebra crossing and after crossing it, however, when they reach the end of the corners, they will always turn according to the pathway where they are in. They will follow the traffic light to cross the zebra crossing or continue their journey. However, at some point, the pedestrian can perform an infraction, thus we defined the vehicles as an infractor, during the infraction, a message will pop up to show that an infraction has been made. So, in this class, we defined functions that will be used as well in the Traffic class.

Firstly, we initialised the class, next the function `def __init__` and its variables and then we defined some functions that are like in the class Vehicle. To make the report simple and short we will only present the essential codes.

```
class Pedestrian:

def __init__(self, posx, posy, direction, section):
        self.posx = posx
```

```
        self.posy = posy
        self.vel = 1
        self.direction = direction
        self.section = section
        self.i = 0
        self.state = "walking"
        self.stop = 0
        self.already_crossed = 0
        self.infractor = 0
```

For pedestrian movement, we defined some functions according to the pedestrian way. In the function `def change_direction(self, i, type),` "type" is a variable which defines the positions where the pedestrian needs to change its direction: "corner", "before crossing" and "after crossing". Moreover, we created the variable "i", which represents the points where the pedestrian needs to decide the direction for its next movement. Before and after crossing, "i" defines the index of the zebra crossing coordinates in the vector "zebra_cross". In the case in which the pedestrian is in a corner, "i" indicates the index of the coordinates of the corner in the vector "pedestrian_corners". When the pedestrian reaches a zebra cross, he will decide randomly whether to cross or not. To this end, the function `def decide_direction(self)` was created. It returns a random value (1 to cross or 0 to continue walking). There are more chances for the result to be 1 because it is more interesting for the simulation. After crossing he will decide, also randomly, whether to turn left or right. Moreover, when they reach the corner of the pedestrian way, they will always turn weather right or left, depending on their pathway.

Other simpler functions were created. `def change_state(self,state)` is used to change the attribute "state" of the class Pedestrian from "walking" to "crossing" when the pedestrian decides to cross and the opposite when he finishes crossing. The rest of the functions are also used to change the value of different attributes that are consulted during the pedestrian movement (the attribute "stop" to stop or continue walking, "already_crossed" to avoid crossing a zebra cross infinitely and "infractor" to label the pedestrian as an infractor).

```
def change_position(self):

        if self.stop == 0:
            if self.direction == "left":
                self.posx -= self.vel
```

```
            if self.direction == "right":
                self.posx += self.vel
            if self.direction == "up":
                self.posy -= self.vel
            if self.direction == "down":
                self.posy += self.vel


def change_direction(self, i, type):
      if type == "corner":
            if i == 0:
                if self.direction == "right":
                    self.direction = "up"
                if self.direction == "down":
                    self.direction = "left"
            elif i == 1:
                if self.direction == "left":
                    self.direction = "up"
                if self.direction == "down":
                    self.direction = "right"
            elif i == 2:
                if self.direction == "right":
                    self.direction = "down"
                if self.direction == "up":
                    self.direction = "left"
            elif i == 3:
                if self.direction == "left":
                    self.direction = "down"
                if self.direction == "up":
                    self.direction = "right"
     elif type == "before crossing":
            if i == 0 or i == 3:
                self.direction = "down"
            elif i == 1 or i == 6:
                self.direction = "right"
            elif i == 2 or i == 5:
                self.direction = "left"
            elif i == 4 or i == 7:
                self.direction = "up"
      elif type == "after crossing":
            if self.direction == "up" or self.direction == "down":
                self.direction = random.choice(["right", "left"])
            elif self.direction == "right" or self.direction == "left":
                self.direction = random.choice(["up", "down"])


def decide_direction(self):
      cross = random.choice([0, 1, 1, 1])
      return cross
```

```
def change_state(self, state):
      self.state = state

def crossed(self, value):
      self.already_crossed = value

def change_stop(self, value):
      self.stop = value

def become_infractor(self):
      self.infractor = 1
```

II.2 RESULT

In this section, we want to present the result of the simulation by showing the screen capture. The running simulation is shown in Figure 68. The pop-up message when the infraction is detected both for vehicles and pedestrians is shown in Figures 69 and 70 respectively.

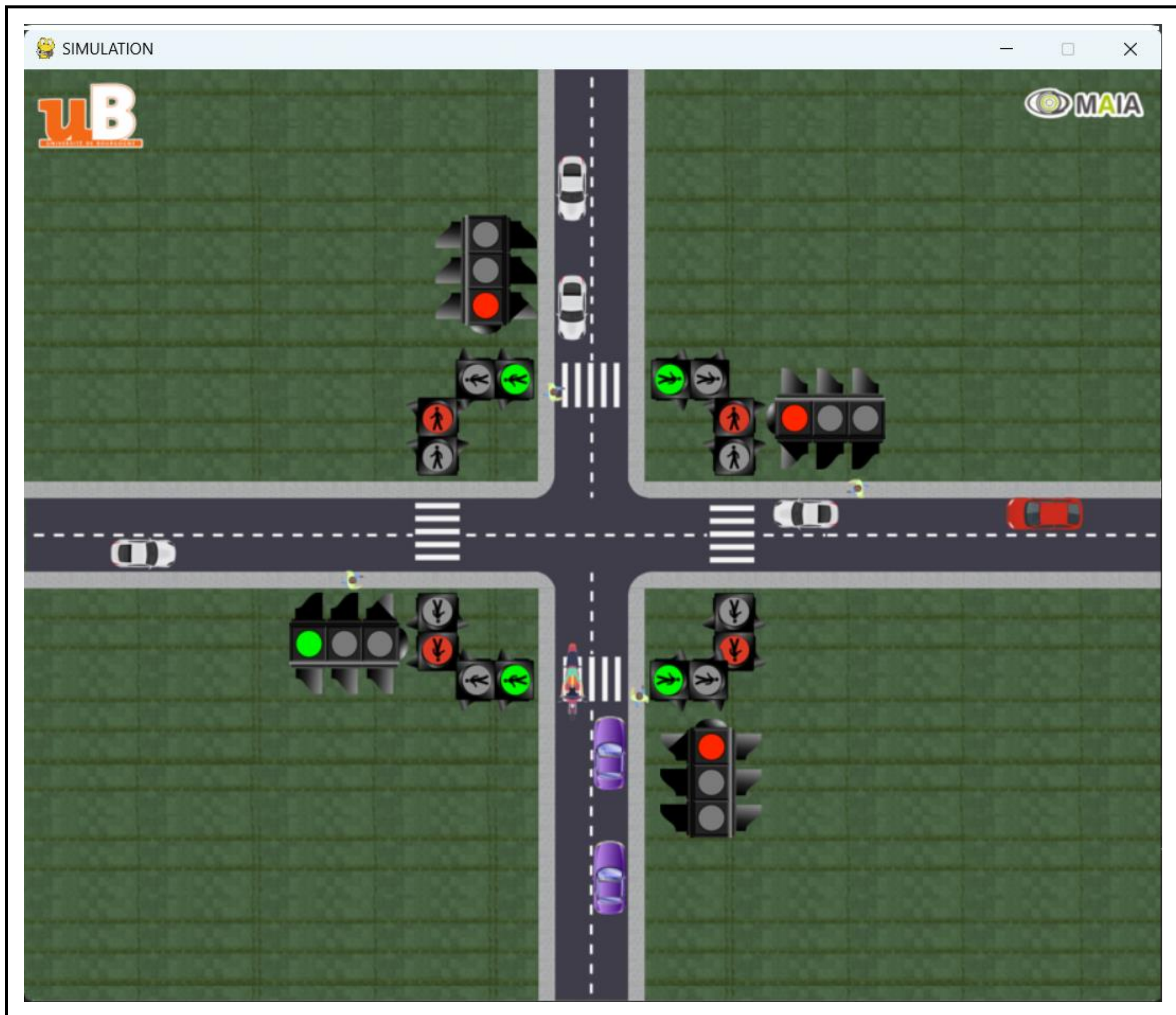


Figure 1. Screen capture of the simulation

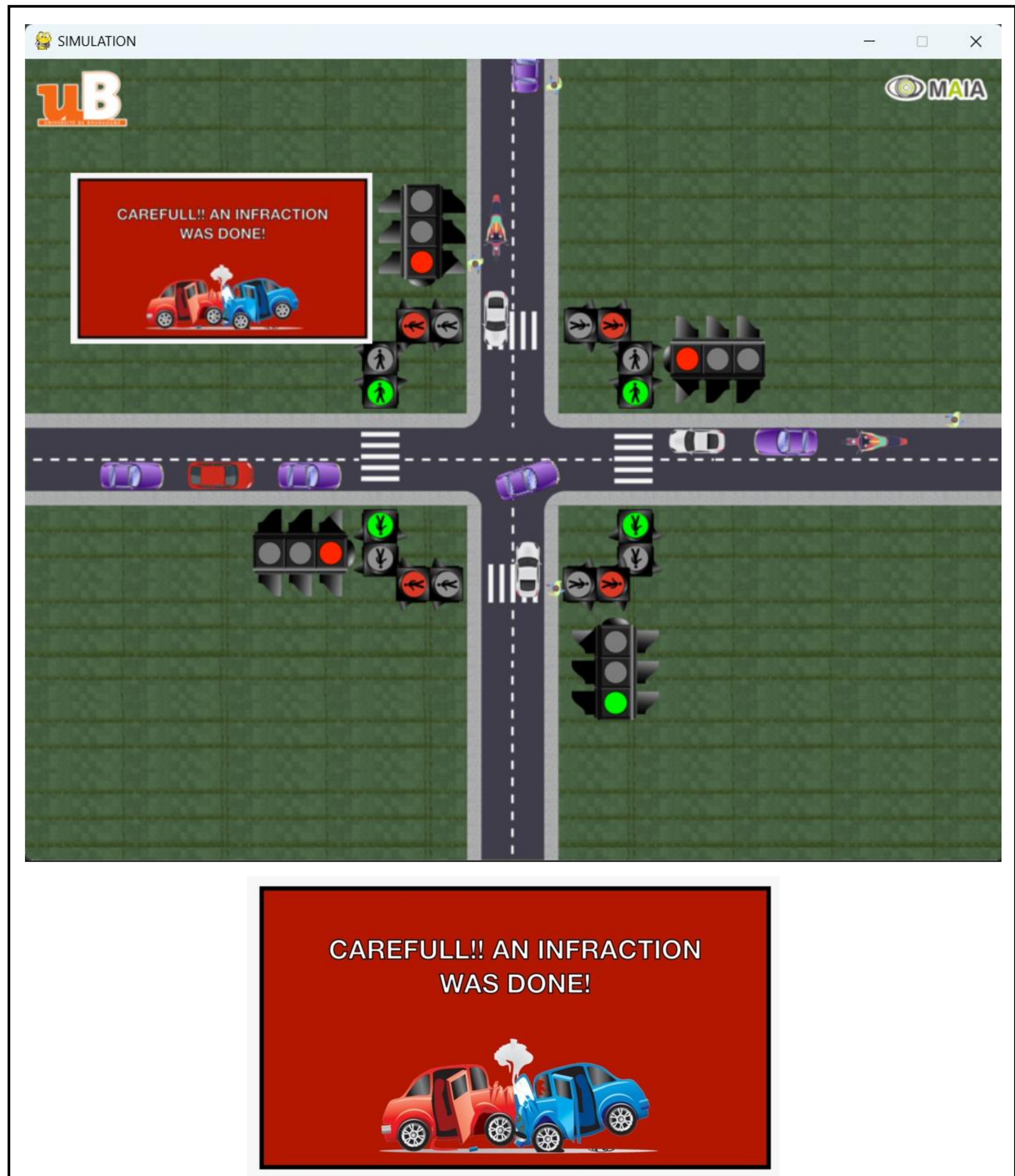


Figure 2. Screen capture of the infraction and its message of vehicle

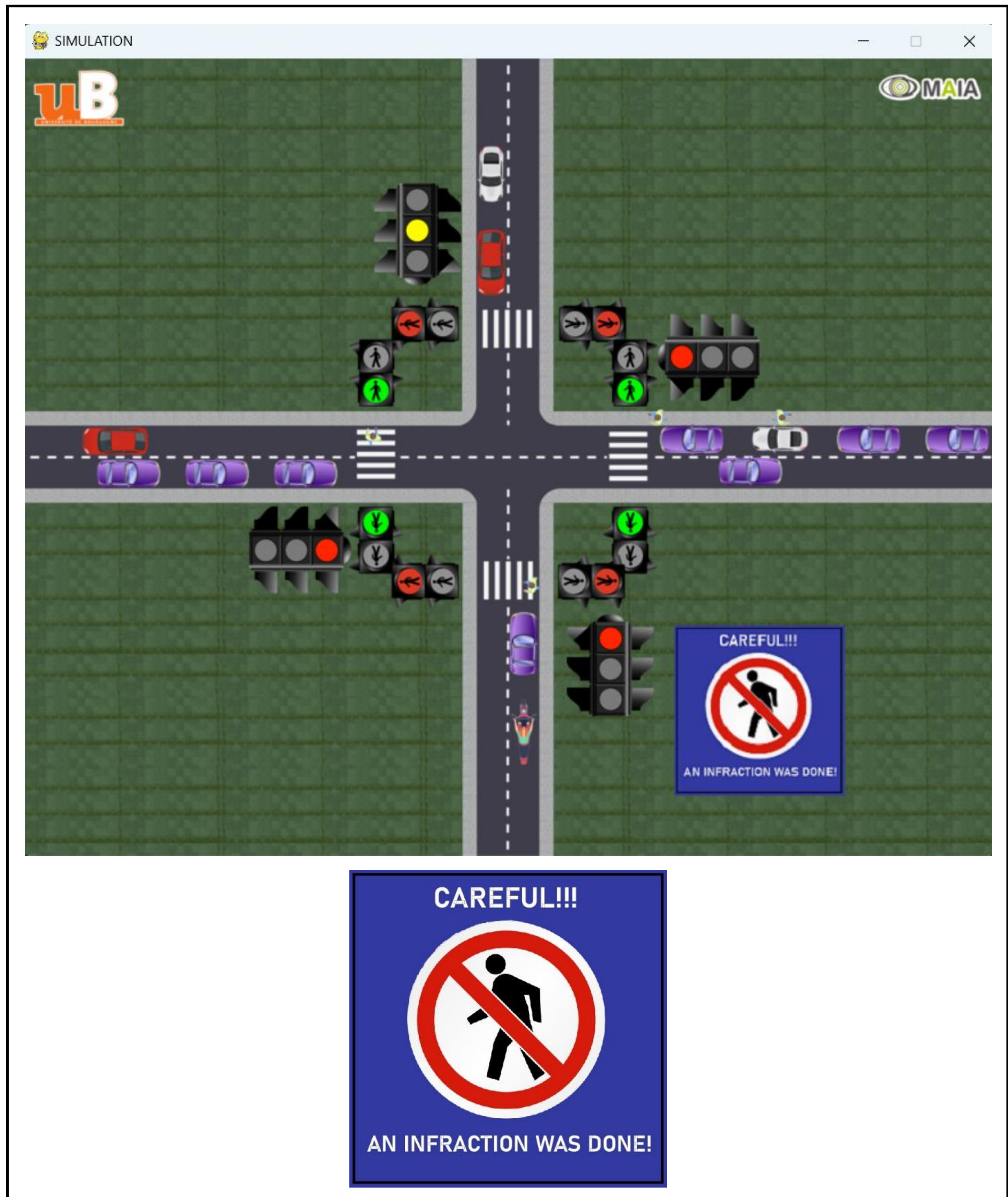


Figure 70. Screen capture of the infraction and its message of pedestrian

II.3 CONCLUSION

In this project, we have successfully fulfilled the tasks, such as simulating the operation of a traffic light both for vehicles and pedestrian, detecting if both a car and pedestrian are in violation or not, and displaying a message if there is a violation or infraction.