Traffic Light Violation Detection System

C. Lisazo, C.G. Colin Tenorio, A. Habib

Centre Universitaire Condorcet, University of Borgundy, Le Creusot, France.
clara_lisazo@etu.u-bourgogne.fr, carmen-guadalupe_colin-tenorio@etu.u-bourgogne.fr,
abdelrahman_habib@etu.u-bourgogne.fr

1. Introduction

Traffic lights are electrically-operated traffic control devices that direct traffic to either stop or proceed forward. The signals offer maximum degree of traffic control at intersections by directing drivers and pedestrians on what to do. The core function of any traffic light is to assign the right of way to the conflicting movements of vehicles and pedestrians at an intersection in order to allow the conflicting streams of traffic to use and share the same intersection by way of time separation. Traffic signals can be 3 colors (red, yellow, and green):

- **Red light:** A red light means that you must come to a complete stop and remain stopped until the light turns green.
- **Yellow Light:** A yellow light indicates that the green light is about to end – you must stop unless unable to safely do so. Treat a yellow light as the beginning of a red light, as opposed to the end of a green light.
- **Green Light:** A green light means 'GO'.

Pedestrian signals are special types of traffic-signal indications installed for the exclusive purpose of controlling pedestrian traffic. Pedestrian signals consist of the illuminated symbols of a walking person and a standing person. The meanings of the indications are as follows:

- **A steady illuminated symbol of a walking person** means that a pedestrian may enter the roadway and proceed in the direction of the indication.
- **A steady illuminated symbol of a standing person** means that a pedestrian cannot enter the roadway.

2. Objective

The objective of this project is to simulate the traffic lights for cars and pedestrians, using the python programming language. In addition, it is intended to observe whether a pedestrian or a car has committed a violation on the road. In case a violation has been committed, the program should display a warning.

3. Methods

3.1 Libraries

3.1.1 Pygame

Pygame is a free and open-source cross-platform library for the development of multimedia applications like video games using Python. It uses the Simple DirectMedia Layer library and several other popular libraries to abstract the most common functions, making writing these programs a more intuitive task. The main reason that we will be using this library is that it is simple and efficient. It is highly portable, which means it works on almost all operating systems.

Figure 1: Pygame Logo

3.1.2 Pygame Modules

- ***pygame.transform:*** module to transform surfaces. A Surface transform is an operation that moves or resizes the pixels. All these functions take a Surface to operate on and return a new Surface with the results.
- ***pygame.image:*** module for image transfer. The image module contains functions for loading and saving pictures, as well as transferring Surfaces to formats usable by other packages.
- ***pygame.display:*** module to control the display window and screen. This module offers control over the pygame display. Pygame has a single display Surface that is either contained in a window or runs full screen. Once you create the display you treat it as a regular Surface.
- ***pygame.font:*** module for loading and rendering fonts.
- ***pygame.sprite:*** module with basic game object classes. This module contains several simple classes to be used within games.
- ***pygame.event:*** pygame module for interacting with events and queues. Pygame handles all its event messaging through an event queue. The routines in this module help to manage that event queue.

3.1.3 Other modules

- **random:** is an in-built module of Python which is used to generate random numbers. These are pseudo-random numbers means these are not truly random. This module can be used to perform random actions such as generating random numbers, print random a value for a list or string, etc.

- ***Time:*** This module provides various time-related functions. It allows functionality like getting the current time, pausing the Program from executing, etc.
- ***Threading:*** This module constructs higher-level threading interfaces on top of the lower-level thread module.
- ***Sys:*** This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

3.2 Code

3.2.1 Screen Settings Section

In these sections three functions are defined that allow us to define the characteristics of our screen, such as width, length, etc. As well as to load images that will remain fixed. In this module the images of the traffic lights for cars and pedestrians are loaded and scaled.
In figure 2 is shown an example of the original background loaded in the simulation. Figure 3 and 4, show an example of the images used as traffic lights: color green, yellow and red, and pedestrians' lights, color green and red.

Figure 2. Background image.

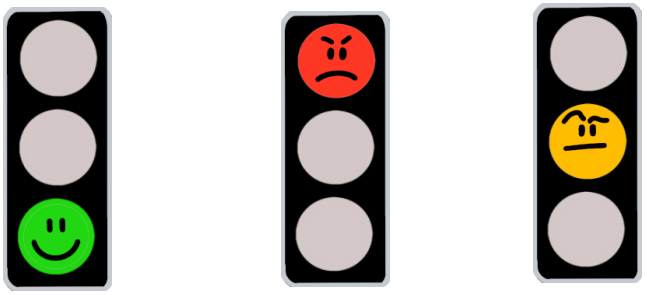

Figure 3. Traffic lights images: Green, Yellow and Red.

Figure 4. Pedestrian lights images: Red and Green

def set_screen_size():

This function initializes a window or screen for display with the width and height given of the screen using pygame.display.set_mode(). The function will return the size, height, width and the display screen.

def set_screen_objects():

This function helps us to place our images on the screen. First the background image is loaded using pygame.image.load. We scale the image using pygame.transform.scale.

The process is performed for all traffic lights, pedestrians, and background images. The function returns all scaled and transformed images.

```
import pygame

def set_screen_size():
    # Screensize
    screenWidth = 1400
    screenHeight = 900
    screenSize = (screenWidth, screenHeight)
    displayScreen = pygame.display.set_mode(screenSize)

    return displayScreen, screenWidth, screenHeight, screenSize

def blit_screen(displayScreen, source, pos):
    displayScreen.blit(source, pos)

def set_screen_objects():
    # Setting background image i.e. image of intersection
    backgroundImage = pygame.image.load('images/background.png')

    # Set the size for the image
    DEFAULT_IMAGE_SIZE = (1400, 922)
    DEFAULT_SIGNAL_SIZE = (32, 98)
    PEDESTRIAN_SIGNAL_SIZE_0 = (25,65)
    PEDESTRIAN_SIGNAL_SIZE_90 = (65,25)

    # Scale the image to your needed size
    backgroundImage = pygame.transform.scale(backgroundImage,
DEFAULT_IMAGE_SIZE)

    pygame.display.set_caption("Traffic Simulation Project")

    # Loading signal images and font
    redLight = pygame.image.load('images/signals/red.png')
    redLight = pygame.transform.scale(redLight, DEFAULT_SIGNAL_SIZE)

    yellowLight = pygame.image.load('images/signals/yellow.png')
    yellowLight = pygame.transform.scale(yellowLight, DEFAULT_SIGNAL_SIZE)

    greenLight = pygame.image.load('images/signals/green.png')
    greenLight = pygame.transform.scale(greenLight, DEFAULT_SIGNAL_SIZE)

    redLightPed = pygame.image.load('images/signals/pedestrians_red_0.png')
    redLightPed = pygame.transform.scale(redLightPed,
PEDESTRIAN_SIGNAL_SIZE_0)
```

Vehicles and Pedestrian's Movements

In the simulation, there are four crosswalks; vehicles can move straight or turn left, as shown in the figure 5.

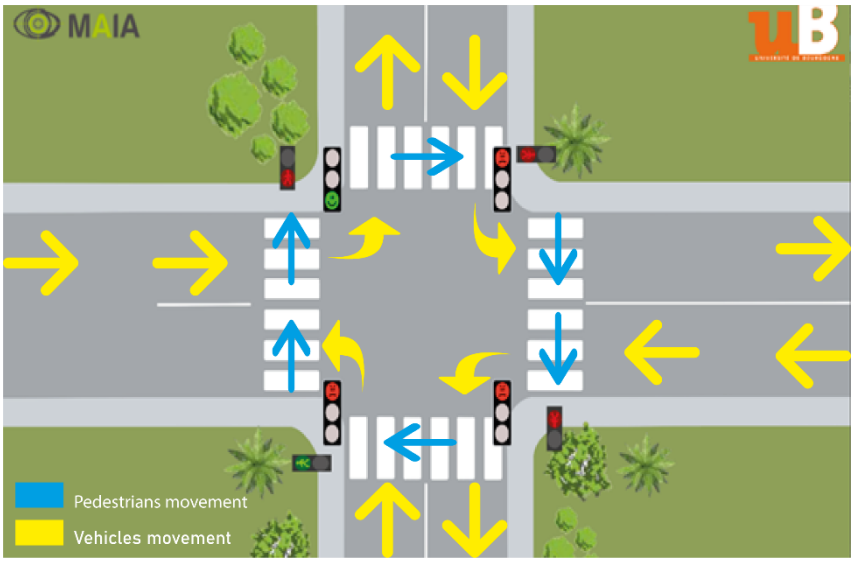

Figure 5: Movements and directions allowed by pedestrians (blue color) and vehicles (yellow color) in the simulation.

Objects in Movement

The program generates four objects that can perform movements: **pedestrian 1, pedestrian 2, motorcycle and car**, as shown in the figure 6. Each of these objects will have the function of moving in four directions depending on the street in which it is located, the directions are: **Up, down, left and right**. Therefore, the images of our objects in those directions are required. So, it is required to have folders with the appropriate directions of each of the objects.

Figure 6. Types of objects in the simulation. Left to right: Pedestrian 1, Pedestrian 2, motorbike and car.

Directions

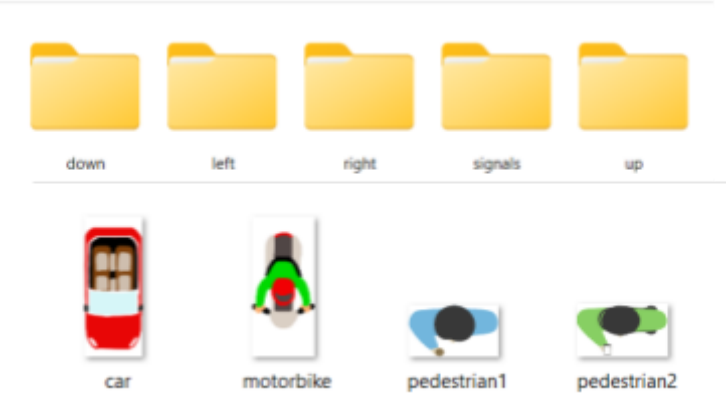

Figure 7 Folders containing the images of the corresponding direction of each moving object.

Object's speed and directions

This section of the code stores different values that are used repeatedly in the code such as the speeds for different objects, directions, violations messages, and colors. This is to make the code more efficient and reusable.

```
from enum import Enum

class OBJECTS_SPEEDS(Enum):
    car     = 3.0
    motorbike    = 3.1
    pedestrian1 = 2.0
    pedestrian2 = 2.25


class DIRECTIONS(Enum):
    UP      = 'up'
    RIGHT   = 'right'
    BOTTOM  = 'left'
    LEFT    = 'left'

class VIOLATIONS(Enum):
    PEDESTRIAN  = "A violation by a pedestrian."
    CAR         = "A violation by a car or motorbike."

class COLORS(Enum):
    BLACK   = (0, 0, 0)
    WHITE   = (255, 255, 255)
    GREEN   = '#00FF00'
    RED     = '#FF0000'
    GRAY    = '#808080'
    HOVER   = '#666666'
```

1.1.1 Traffic Signal

This section generates the TrafficSignal class, which allows us to create a Traffic Signal object. Two functions are also created, which will allow us to define the duration in seconds of the traffic signal in each color. As well as to initialize the settings of our traffic lights.

def get_defaults_colors():

This function will assign the timers to each of the traffic lights. One timer for each green traffic light. One for red and one for yellow. The green timer is different because the traffic light changes clockwise, so it must wait to go all the way around to be green. The function returns the timers for each color.

def get_signal_settings():

In this function the traffic light settings are initialized.

The number of traffic lights is defined. There are 3 variables that indicate if the current traffic light is green, if a yellow light is on or off, and which is the next traffic light to be green. It returns these three variables and the list of traffic lights.

class TrafficSignal:

This class is used to create a traffic light object. It receives three arguments that represent the traffic light colors.

```
def get_defaults_colors():
    """
    Default timer values for signals
    """
    timerGreen = {0:10, 1:10, 2:10, 3:10}
    timerRed = 150
    timerYellow = 5
    return timerGreen, timerRed, timerYellow

def get_signal_settings():
    """
    Initialization of traffic lights' settings.
    """
    signalsList = []
    numOfSignals = 4
    nowGreen = 0   # Indicates which traffic light is green currently
    nextGreen = (nowGreen+1)%numOfSignals  #  Indicates which traffic light
will turn green next
    nowYellow = 0   # Indicates if the yellow traffic light is on or off

    return signalsList, numOfSignals, nowGreen, nextGreen, nowYellow


class TrafficSignal:
    '''This class is used to create a traffic signal object. It contains
three attributes representing the different colors of traffic light.
    '''
    def __init__(self, red, yellow, green):
        self.red = red
        self.yellow = yellow
        self.green = green
```

1.1.2 Violation Detection

In this module we create the ViolationDetection class that allows us to detect and visually alarm if a violation has been committed either by a vehicle or a pedestrian. The functions allow the alarm to be triggered in a text message or displaying an image on the screen as a violation alert (Figure 8 and 9).

Figure 8 Example of a visual alarm in case of a car violation.

Figure 9 Example of a visual alarm in case of a pedestrian violation

class ViolationDetection():

This class is used to detect and warn of both vehicle and pedestrian violations. Receives as argument vehicleClass: which will indicate the type of vehicle that triggered the violation. The class checks if the vehicleClass corresponds to a pedestrian, otherwise it corresponds to a Car.

def text_objects(self, text, font):

Function that allows us to create a text object in pygame. It receives as arguments the string to display and the font type. Returns tuple consisting of the generated rectangle and the text with the selected font.

def displayViolation(self):

Function that allows us to display a text message on the screen. The font is set using pygame.font.SysFont.

It is displayed on the screen using screen.blit(textSurf, textRect).

def displayViolationImage(self):

This function will display an image in the event of a violation.

The function checks which type of vehicle committed the infraction and in each case the corresponding image will be loaded by pygame.image.load.

The image is scaled to the corresponding size by pygame.transform.scale and displayed on the screen with the correct dimensions by screen.blit(imp, self.dimensions).

```python
import pygame
from enums import VIOLATIONS, COLORS
from screen_settings import set_screen_size

screen, screenWidth, screenHeight, screenSize = set_screen_size()

class ViolationDetection():
    '''This class is used to detect and alert for both vehicles and
pedestrians violations.
       Current implementation supports displaying both image and text
messages as a violation alert. It receives the vehicle class and uses it to
decide whether
       it refers to a pedestrian or any type of vehicle.'''
    def __init__(self, vehicleClass):
        '''Handles all violation types detection.

        Args:
         - vehicleClass: `str` for indicating the type of object that made
the violation.
        '''
        self.violationMessages = {i.name: i.value for i in VIOLATIONS}
        self.vehicleClass = vehicleClass
        self.violationType = self.vehicleClass in ['pedestrian1',
'pedestrian2']
        self.violationText = self.violationMessages['PEDESTRIAN'] if
vehicleClass in ['pedestrian1', 'pedestrian2'] else
self.violationMessages['CAR']
        self.dimensions = (1000, 50)
        self.font = pygame.font.Font(None, 30)

    def text_objects(self, text, font):
        '''Function used to create a pygame text object.

        Args:
         - text: `str` for the text to be displayed
         - font: `pygame SysFont` for the font type

        Returns: `tuple` consists of both the surface and rectangle
generated.
        '''
        textSurface = font.render(text, True, COLORS.RED.value)
        textRect = textSurface.get_rect()
        return textSurface, textRect

    def displayViolation(self):
```

```
        '''Displays a text violation on the screen.
        '''
        smallText = pygame.font.SysFont('Arial',20)  # Typo: Sysfont ->
SysFont
        textSurf, textRect = self.text_objects(self.violationText,
smallText)
        textRect.center = self.dimensions
        screen.blit(textSurf, textRect)
        # time.sleep(2)

    def displayViolationImage(self):
        '''Displays a image violation on the screen.
        '''
        match self.vehicleClass:
            case "motorbike":
                imp =
pygame.image.load("images\VIOLATION_MOTORBIKE.png").convert()
            case "car":
                imp =
pygame.image.load("images\VIOLATION_CAR.png").convert()
            case _:
                imp =
pygame.image.load("images\VIOLATION_PEDESTRIAN.png").convert()

        imp = pygame.transform.scale(imp, (350, 120))
        # imp = pygame.image.load("images\VIOLATION_CAR.png").convert()
        screen.blit(imp, self.dimensions)
        pygame.display.flip()
```

3.1.3 Main Code

In this part of the main code called app, as a first step we import the libraries and functions necessary for the program to work correctly. Including necessary functions from modules that we created such as screen_settings, traffic_signal and violation_detection. A list of speeds, the timers corresponding to each color and the list of traffic lights are generated. The false violation flag is also initialized. Finally, the screen settings are adjusted. In this part we initialize our speed list. We also initialize our traffic light objects. The variables nowGreen, nextGreen, nowYellow are created.

```
import os
# Next line just to stop showing Pygame welcoming message on every run.
os.environ['PYGAME_HIDE_SUPPORT_PROMPT'] = "hide"

import random
import time
import threading
import pygame
import sys

from loguru import logger
from enums import OBJECTS_SPEEDS, VIOLATIONS, COLORS


# Imports
from violation_detection import ViolationDetection
from screen_settings import set_screen_size, blit_screen, set_screen_objects
from traffic_signal import get_defaults_colors, get_signal_settings,
TrafficSignal

# Flag cars
SIMULATION_VIOLATION = False

VIOLATION = None

SLEEP_TIME_5 = 5
SLEEP_TIME_1 = 1


speeds = {i.name: i.value for i in OBJECTS_SPEEDS}

timerGreen, timerRed, timerYellow = get_defaults_colors()
signalsList, numOfSignals, nowGreen, nextGreen, nowYellow =
get_signal_settings()

# Pygame screen settings
displayScreen, screenWidth, screenHeight, screenSize = set_screen_size()
```

In this part of the code, we initialize the starting coordinates of our vehicles and pedestrians. In this case, the pedestrians and vehicles start at the edges of the screen.

We generate the coordinates in x and y, corresponding to the start of the vehicles, and we do it for each direction: **UP, RIGHT, LEFT** and **DOWN.**

The process is repeated for the pedestrian coordinates, but stored in the variables x_ped and y_ped.

The **roadObj** list is initialized, with the corresponding directions and the crossed variable in 0, that will allow us to know if an object has crossed the stopping line or not.

roadObjTypes is initialized with the types of vehicles and a number is assigned to each type to be accessed later in the code.

directionsNum, is initialized with the directions and a number is assigned to each one.

The Coordinates of the traffic light positions are defined:

lightCoords, represent the coordinates of the traffic lights on the screen.

lightPedCoords, represent the coordinates of the pedestrian traffic lights on the screen.

The coordinates of the vehicle and pedestrian stop lines are defined:

stopCoords, contains the coordinates of a stopping line, used to check if the vehicles have already crossed or not. In the case when the traffic light turns red and the position of a vehicle is beyond this point, the vehicle will continue moving (an appropriate timer for the yellow traffic light is set, so that the vehicle has enough time to finish crossing). If the traffic light turns red and the vehicle has not reached this line yet, it will stop. This is implemented in order to avoid vehicles stopping on the crosswalk and blocking the way for pedestrians to cross.

stopCoordsPed, contains the coordinates of a stopping line, used to check if the pedestrians have already crossed or not.

defaultStopCoords contains the coordinates at which the vehicles (cars and motorbikes) must stop if the traffic light is red.

defaultStopPed contains the coordinates at which the pedestrians must stop if the pedestrian light is red.

The Distance between vehicles is defined:

positionGap, represents the space between vehicles.

Some parameters are defined for turning vehicles:

midCoords, contains the coordinates of the midpoint of the road intersection, used as the reference point from where the vehicles will turn.

Then, pygame is initialized with the following lines:

```
pygame.init()

simulation = pygame.sprite.Group()
```

```
# Coordinates of vehicles' and pedestrians' start
x = {'right':[0,0,0], 'down':[765,737,697], 'left':[1400,1400,1400],
'up':[602,627,657]}
y = {'right':[348,370,398], 'down':[0,0,0], 'left':[548,506,456],
'up':[800,800,800]}
x_ped = {'right':[0,0,0], 'down':[870,900,930],
'left':[1400,1400,1400],'up':[430,460,490]}
y_ped = {'right':[190,220,250], 'down':[0,0,0],
'left':[630,660,690],'up':[900,900,900]}

roadObj = {'right': {0:[], 1:[], 2:[], 'crossed':0}, 'down': {0:[], 1:[],
2:[], 'crossed':0}, 'left': {0:[], 1:[], 2:[], 'crossed':0}, 'up': {0:[],
1:[], 2:[], 'crossed':0}}
roadObjTypes = {1:'car', 2:'motorbike',3:'pedestrian1',4:'pedestrian2'}
directionsNum = {0:'right', 1:'down', 2:'left', 3:'up'}

# Coordinates of traffic lights' positions
lightCoords = [(530,230),(810,230),(810,570),(530,570)]
lightPedCoords = [(460,230),(850,230),(900,610),(480,680)]

# Coordinates of vehicles' and pedestrians' stop lines
stopCoords = {'right': 430, 'down': 220, 'left': 990, 'up': 710}
stopCoordsPed = {'right': 510, 'down': 330, 'left': 840, 'up': 610}
defaultStopCoords = {'right': 430, 'down': 210, 'left': 1000, 'up': 720}
defaultStopPed = {'right': 500, 'down': 330, 'left': 850, 'up': 600}

# Gap between vehicles
positionGap = 25    # stopping gap

# Parameters for turning vehicles:
roadObjTurned = {'right': {0:[], 1:[], 2:[]}, 'down': {0:[], 1:[], 2:[]},
'left': {0:[], 1:[], 2:[]}, 'up': {0:[], 1:[], 2:[]}}
roadObjNotTurned = {'right': {0:[], 1:[], 2:[]}, 'down': {0:[], 1:[], 2:[]},
'left': {0:[], 1:[], 2:[]}, 'up': {0:[], 1:[], 2:[]}}
angleRotation = 3
midCoords = {'right': {'x':705, 'y':445}, 'down': {'x':695, 'y':450},
'left': {'x':695, 'y':425}, 'up': {'x':695, 'y':400}} #represents the
coordinates of the middle point of the road intersectioin, from where
vehicles would turn

pygame.init()
simulation = pygame.sprite.Group()
```

```
class RoadObjects(pygame.sprite.Sprite):
```

The **RoadObjects** class allows us to create a roadObject (car, motorcycle or pedestrian). It gives it all its attributes and movements linked to the traffic light logic.

Receives as arguments**: laneNum, roadObjClass, directionNumber, direction, willTurn.** The lane number, the type of vehicle, the number associated to the direction in which it should move, the direction of movement and a number between 0 and 1 that indicates whether the vehicle will turn or not.

The constructor of the class, __init__, contains all the attributes of the road objects, including:

- laneNum: the lane number given as argument when creating the object.
- roadObjClass: the type of road object (car, motorbike or pedestrian)
- speed of the objects
- directionNumber: number associated to the moving direction of the object
- direction: the direction associated with the movement of the object (string).
- x, y: the current coordinates (x and y) of the road object
- crossed: binary value that will indicate if the object has crossed the stopping line or not.
- willTurn: indicates if the car or motorbike will turn at the intersection or not.
- turned: similar as crossed, it will be 1 if the vehicle has already turned, and 0 if it has not.
- rotationAngle: gives the angle of rotation for turning vehicles
- objIndex: gives the index of the object with respect of the other objects that are on the same lane moving in the same direction.

The class will load the corresponding image into the folder using this:
imagePath = "images/" + direction + "/" + roadObjClass + ".png"
self.originalImage = pygame.image.load(imagePath)

In case the object is a pedestrian we scale the image, taking into account the direction in which it moves with pygame.transform.scale

If the vehicle or pedestrian has crossed the stopping line, the simulation variables are started to constantly check if a violation has been committed.
global SIMULATION_VIOLATION
global VIOLATION

The space between the vehicles is checked and the stopping coordinates are defined for each case, i.e. in each of the corresponding directions.

```
def updateRotationAngles(self):
```

This function is used to update the angle of rotation of the object. When the car or motorbike has already crossed the reference turning point:

- If **turned** == 0: it will start turning (this is obtained by applying the pygame.transform.rotate function). The angle will start to increase as the car is turning, and when it finally reaches 90 degrees, the **turned** variable will be set

to 1, to indicate that the vehicle has already finished turning, and the vehicle will be appended to the list that contains the vehicles that have turned. Also, the index of the crossing vehicles will be updated.

- In the other case, if **turned** == 1, it means that another car has already turned before on the same lane, so this car will only turn if it has enough space gap with regards to the vehicle in front of it.
- For both cases, a vehicle can only turn if it also has a green light, or if it has already crossed the stopping line when the light turns red.

Once all the variables in the constructor have been initialized, the program checks if there are any road objects moving in the same lane and direction as the object that is being created.

- If there are other objects on the same lane, moving in the same direction, the **stop** value of the new object is set, considering the dimensions of the road object ahead of it, plus the stopping space gap.
- If there are no other objects ahead, this **stop** value will be equal to the defaultStopCoords for cars and motorbikes, or defaultStopPed for pedestrians, previously defined.

Finally, these **stop** coordinates will indicate the object where to stop if the traffic light or the pedestrian light is red.

After that, the coordinates from which the objects are generated are updated in order to avoid objects overlapping between themselves.

```
class RoadObjects(pygame.sprite.Sprite):
    """
    This class contains all the attributes and methods that correspond to
road objects (cars, motorbikes and pedestrians).
    It defines all the parameters of the road objects, such as size, lane,
direction, speed and whether they will turn or not.
    It controls all their movements, and determines when they can cross the
intersection according to the traffic lights' logic.

    """
    def __init__(self, laneNum, roadObjClass, directionNumber, direction,
willTurn):
        pygame.sprite.Sprite.__init__(self)
        self.laneNum = laneNum
        self.roadObjClass = roadObjClass
        self.speed = speeds[roadObjClass]
        self.directionNumber = directionNumber
        self.direction = direction
        if (self.roadObjClass == 'pedestrian1') or (self.roadObjClass ==
'pedestrian2'):
            self.x = x_ped[direction][laneNum]
        else:
            self.x = x[direction][laneNum]
        if (self.roadObjClass == 'pedestrian1') or (self.roadObjClass ==
'pedestrian2'):
            self.y = y_ped[direction][laneNum]
        else:
            self.y = y[direction][laneNum]
        self.crossed = random.randint(0,1)
        # self.crossed = 0 #I commented the random version of this line to
see if it works without the violations, then change it back
        self.willTurn = willTurn
        self.turned = 0
        self.rotationAngle = 0
        roadObj[direction][laneNum].append(self)
        self.objIndex = len(roadObj[direction][laneNum]) - 1
        self.crossedObjIndex = 0
        imagePath = "images/" + direction + "/" + roadObjClass + ".png"
        self.originalImage = pygame.image.load(imagePath)
        self.image = pygame.image.load(imagePath)

        if self.roadObjClass not in ['pedestrian1', 'pedestrian2']:
            if(direction == 'up' or direction == 'down'):
                dim = (33, 80)
            else:
```

```
                dim = (80, 33)
            self.originalImage =
pygame.transform.scale(self.originalImage,dim)
            self.image = pygame.transform.scale(self.image, dim)
        else:
            self.image = pygame.transform.scale(self.image, (25,25))

        if self.crossed == 1:
            global SIMULATION_VIOLATION
            global VIOLATION

            VIOLATION = ViolationDetection(vehicleClass = self.roadObjClass)
            SIMULATION_VIOLATION = not SIMULATION_VIOLATION
        else:
            SIMULATION_VIOLATION = False

        if(len(roadObj[direction][laneNum])>1 and
roadObj[direction][laneNum][self.objIndex-1].crossed==0):    # if more than
1 vehicle in the lane of vehicle before it has crossed stop line
            if(direction=='right'):
                self.stop = roadObj[direction][laneNum][self.objIndex-
1].stop - roadObj[direction][laneNum][self.objIndex-
1].image.get_rect().width - positionGap         # setting stop coordinate
as: stop coordinate of next vehicle - width of next vehicle - gap
            elif(direction=='left'):
                self.stop = roadObj[direction][laneNum][self.objIndex-
1].stop + roadObj[direction][laneNum][self.objIndex-
1].image.get_rect().width + positionGap
            elif(direction=='down'):
                self.stop = roadObj[direction][laneNum][self.objIndex-
1].stop - roadObj[direction][laneNum][self.objIndex-
1].image.get_rect().height - positionGap
            elif(direction=='up'):
                self.stop = roadObj[direction][laneNum][self.objIndex-
1].stop + roadObj[direction][laneNum][self.objIndex-
1].image.get_rect().height + positionGap
        else:
            self.stop = defaultStopCoords[direction]

        if (self.roadObjClass == 'pedestrian1') or (self.roadObjClass ==
'pedestrian2'):
            self.stop = defaultStopPed[direction]
        else:
            self.stop = defaultStopCoords[direction]
```

```
        # Set new starting and stopping coordinate
        if(direction=='right'):
            temp = self.image.get_rect().width + positionGap
            x[direction][laneNum] -= temp
        elif(direction=='left'):
            temp = self.image.get_rect().width + positionGap
            x[direction][laneNum] += temp
        elif(direction=='down'):
            temp = self.image.get_rect().height + positionGap
            y[direction][laneNum] -= temp
        elif(direction=='up'):
            temp = self.image.get_rect().height + positionGap
            y[direction][laneNum] += temp
        simulation.add(self)

 def updateRotationAngles(self):
        """
        This function is used to update the angle of rotation of the road
object
        """
        self.rotationAngle += angleRotation

        if(self.rotationAngle==90):
                                    self.turned = 1
                                    roadObjTurned[self.direction][self.laneNum].
append(self)
                                    self.crossedObjIndex =
len(roadObjTurned[self.direction][self.laneNum]) - 1

        return pygame.transform.rotate(self.originalImage,
self.rotationAngle)
```

def controlMovement(self):

This function allows us to control the movement of each of our roadObjects.

The logic is divided depending on whether the object is a pedestrian or a vehicle (car, motorcycle) and depending on the direction that is being followed, for example in the movement from top to bottom or from right to left.

- For pedestrians:

  The function first checks whether the pedestrian has crossed the stop line. If the pedestrian has already crossed its stop line, **crossed** = 1, and it will continue moving, whether it has a green light or not (to avoid them stopping in the middle of the crosswalk). The traffic lights of the vehicles give the pedestrians enough time to finish crossing before cars and motorbikes start moving again.

  It then verifies that the following is true: whether the pedestrian has not reached the stop coordinate, or has crossed the stop line, or has the traffic light green) and (is the first pedestrian in that lane or has enough space until the next pedestrian in the same lane: in case this is true, the pedestrian can advance.

  The advancement or moving of the pedestrian is achieved by incrementing or decrementing the coordinates of the pedestrian (depending on the direction in which it is moving) by the value of its corresponding speed.

  Also, the function will consider the case in which there is another pedestrian ahead, and the new pedestrian will only be allowed to advance if it has a space gap greater than the stopping gap with regards to the other pedestrian ahead.

- For the movement of cars and motorcycles:

  The direction in which the vehicle is moving is first checked to follow the correct logic.

  The function also checks if the vehicle has crossed the stopline.

  Unlike the movement of pedestrians, it also checks whether the vehicle is going to turn or not.

  Then the following is checked:

  If the road object has not reached the stop coordinate, or has a green traffic light, or has crossed the stopline, and it is the first object in that lane or it has enough space gap with respect to the next vehicle in the same lane, also if the vehicle ahead has already turned (see explanation of updateRotationAngles for more information about the logic implemented for turning vehicles).

  In case this is true, the vehicle can move. If not it will stop at its corresponding **stop** value.

  The same logic is followed for each of the allowed movements: Right, Left, Up and Down.

```
    def controlMovement(self):
        """
        This function controls the movements of all road objects (cars,
motorbikes and pedestrians).
        """
        if(self.direction=='right'):
            if (self.roadObjClass == 'pedestrian1') or (self.roadObjClass ==
'pedestrian2'):
                """
                Control of the movement and crossing of pedestrians going
from left to right
                """
                if(self.crossed==0 and
self.x+self.image.get_rect().width>stopCoordsPed[self.direction]):   # if
the pedestrian has crossed the stopline
                    self.crossed = 1
                if((self.x+self.image.get_rect().width<=self.stop or
self.crossed==1 or (nowGreen==2 and nowYellow==0)) and (self.objIndex==0 or
self.x+self.image.get_rect().width<(roadObj[self.direction][self.laneNum][se
lf.objIndex-1].x - positionGap))):
                # (if the pedestrian has not reached the stop coordinate, or
has crossed the stopline, or has a green traffic light) and (it is the first
pedestrian in that lane or it has enough gap to the next pedestrian in the
same lane)
                    self.x += self.speed  # move the pedestrian
            else:
                """
```

```
                Control of the movement of cars and motorbikes going from
left to right
                """
                if(self.crossed==0 and
self.x+self.image.get_rect().width>stopCoords[self.direction]): # if the
road object has crossed the stopline
                    self.crossed = 1
                    roadObj[self.direction]['crossed'] += 1
                    if(self.willTurn==0): # if the vehicle will not turn
                        roadObjNotTurned[self.direction][self.laneNum].appen
d(self)
                        self.crossedObjIndex =
len(roadObjNotTurned[self.direction][self.laneNum]) - 1
                if(self.willTurn==1): # if the vehicle will turn
                    if(self.crossed==0 or
self.x+self.image.get_rect().width<stopCoords[self.direction]+200):
                        if((self.x+self.image.get_rect().width<=self.stop or
(nowGreen==0 and nowYellow==0) or self.crossed==1) and (self.objIndex==0 or
self.x+self.image.get_rect().width<(roadObj[self.direction][self.laneNum][se
lf.objIndex-1].x - positionGap) or
roadObj[self.direction][self.laneNum][self.objIndex-1].turned==1)):
                        # (if the road object has not reached the stop
coordinate, or has a green traffic light, or has crossed the stopline) and
(it is the first object in that lane or it has enough gap to the next
vehicle in the same lane, also if the vehicle ahead has already turned, then
overlap is no longer an issue)
                            self.x += self.speed
                    else:
                        if(self.turned==0):
                            """
                            Once the vehicle crosses the turning point, if
the turned value is 0, it turns as it rotates alongthe x and y axes. Once
the angle of rotations is 90 degrees, the turned variable is set to 1, the
vehicle is added to the vehiclesTurned list, and its crossedObjIndex is
updated.
                            """
                            self.image = self.updateRotationAngles()
                            self.x += 2.4
                            self.y -= 2.8
                        else:
                            """
                            Else, if the turned value is 1, the vehicle
moves only if there is enough gap from the vehicle ahead. The decision is
based on the same conditions as previous case.
                            """
```

```
                            if(self.crossedObjIndex==0 or
(self.y>(roadObjTurned[self.direction][self.laneNum][self.crossedObjIndex-
1].y + roadObjTurned[self.direction][self.laneNum][self.crossedObjIndex-
1].image.get_rect().height + positionGap))):
                                self.y -= self.speed
                else: #in case the vehicle will not turn:
                    if(self.crossed == 0):
                        if((self.x+self.image.get_rect().width<=self.stop or
(nowGreen==0 and nowYellow==0)) and (self.objIndex==0 or
self.x+self.image.get_rect().width<(roadObj[self.direction][self.laneNum][se
lf.objIndex-1].x - positionGap))):
                            self.x += self.speed
                    else:
                        if((self.crossedObjIndex==0) or
(self.x+self.image.get_rect().width<(roadObjNotTurned[self.direction][self.l
aneNum][self.crossedObjIndex-1].x - positionGap))):
                            self.x += self.speed
        elif(self.direction=='down'):
            if (self.roadObjClass == 'pedestrian1') or (self.roadObjClass ==
'pedestrian2'):
                """
                Control of the movement and crossing of pedestrians going
from the top to the bottom of the screen.
                The logic is the same as the one used in the previous case.
                """
                if(self.crossed==0 and
self.y+self.image.get_rect().height>stopCoordsPed[self.direction]):
                    self.crossed = 1
                if((self.y+self.image.get_rect().height<=self.stop or
self.crossed == 1 or (nowGreen==3 and nowYellow==0)) and (self.objIndex==0
or
self.y+self.image.get_rect().height<(roadObj[self.direction][self.laneNum][s
elf.objIndex-1].y - positionGap))):
                    self.y += self.speed
            else:
                """
                Control of the movement and crossing of cars and motorbikes
going from the top to the bottom of the screen.
                The logic is the same as the one used in the previous case.
                """
                if(self.crossed==0 and
self.y+self.image.get_rect().height>stopCoords[self.direction]):
                    self.crossed = 1
                    roadObj[self.direction]['crossed'] += 1
                    if(self.willTurn==0):
```

```
                        roadObjNotTurned[self.direction][self.laneNum].appen
d(self)
                        self.crossedObjIndex =
len(roadObjNotTurned[self.direction][self.laneNum]) - 1
                if(self.willTurn==1):
                    if(self.crossed==0 or
self.y+self.image.get_rect().height<stopCoords[self.direction]+200):
                        if((self.y+self.image.get_rect().height<=self.stop
or (nowGreen==1 and nowYellow==0) or self.crossed==1) and (self.objIndex==0
or
self.y+self.image.get_rect().height<(roadObj[self.direction][self.laneNum][s
elf.objIndex-1].y - positionGap) or
roadObj[self.direction][self.laneNum][self.objIndex-
1].turned==1)):
                            self.y += self.speed
                    else:
                        if(self.turned==0):
                            self.image = self.updateRotationAngles()
                            self.x += 1.2
                            self.y += 1.8
                        else:
                            if(self.crossedObjIndex==0 or ((self.x +
self.image.get_rect().width) <
(roadObjTurned[self.direction][self.laneNum][self.crossedObjIndex-1].x -
positionGap))):
                                self.x += self.speed
                else:
                    if(self.crossed == 0):
                        if((self.y+self.image.get_rect().height<=self.stop
or (nowGreen==1 and nowYellow==0)) and (self.objIndex==0 or
self.y+self.image.get_rect().height<(roadObj[self.direction][self.laneNum][s
elf.objIndex-1].y - positionGap))):
                            self.y += self.speed
                    else:
                        if((self.crossedObjIndex==0) or
(self.y+self.image.get_rect().height<(roadObjNotTurned[self.direction][self.
laneNum][self.crossedObjIndex-1].y - positionGap))):
                            self.y += self.speed


        elif(self.direction=='left'):
            if (self.roadObjClass == 'pedestrian1') or (self.roadObjClass ==
'pedestrian2'):
                """
```

```
                Control of the movement and crossing of pedestrians going
from right to left.
                The logic is the same as the one used in the previous cases.
                """
                if(self.crossed==0 and
self.x<stopCoordsPed[self.direction]):
                    self.crossed = 1
                if((self.x>=self.stop or self.crossed == 1 or (nowGreen==0
and nowYellow==0)) and (self.objIndex==0 or
self.x>(roadObj[self.direction][self.laneNum][self.objIndex-1].x +
roadObj[self.direction][self.laneNum][self.objIndex-
1].image.get_rect().width + positionGap))):
                    self.x -= self.speed
            else:
                """
                Control of the movement and crossing of cars and motorbikes
going from the right to the left.
                The logic is the same as the one used in the previous cases.
                """
                if(self.crossed==0 and self.x<stopCoords[self.direction]):
                    self.crossed = 1
                    roadObj[self.direction]['crossed'] += 1
                    if(self.willTurn==0):
                        roadObjNotTurned[self.direction][self.laneNum].appen
d(self)
                        self.crossedObjIndex =
len(roadObjNotTurned[self.direction][self.laneNum]) - 1
                if(self.willTurn==1):
                    if(self.crossed==0 or self.x>stopCoords[self.direction]-
200):
                        if((self.x>=self.stop or (nowGreen==2 and
nowYellow==0) or self.crossed==1) and (self.objIndex==0 or
self.x>(roadObj[self.direction][self.laneNum][self.objIndex-1].x +
roadObj[self.direction][self.laneNum][self.objIndex-
1].image.get_rect().width + positionGap) or
roadObj[self.direction][self.laneNum][self.objIndex-
1].turned==1)):
                            self.x -= self.speed
                    else:
                        if(self.turned==0):
                            self.image = self.updateRotationAngles()
                            self.x -= 1
                            self.y += 1.2
                        else:
```

```
                            if(self.crossedObjIndex==0 or ((self.y +
self.image.get_rect().height)
<(roadObjTurned[self.direction][self.laneNum][self.crossedObjIndex-
1].y  -  positionGap))):
                                self.y += self.speed
                else:
                    if(self.crossed == 0):
                        if((self.x>=self.stop or (nowGreen==2 and
nowYellow==0)) and (self.objIndex==0 or
self.x>(roadObj[self.direction][self.laneNum][self.objIndex-1].x +
roadObj[self.direction][self.laneNum][self.objIndex-
1].image.get_rect().width + positionGap))):
                            self.x -= self.speed
                    else:
                        if((self.crossedObjIndex==0) or
(self.x>(roadObjNotTurned[self.direction][self.laneNum][self.crossedObjIndex
-1].x + roadObjNotTurned[self.direction][self.laneNum][self.crossedObjIndex-
1].image.get_rect().width + positionGap))):
                            self.x -= self.speed
                            if(self.crossed==0 and
self.x<stopCoords[self.direction]):
                                self.crossed = 1
                            if((self.x>=self.stop or self.crossed == 1 or
(nowGreen==2 and nowYellow==0)) and (self.objIndex==0 or
self.x>(roadObj[self.direction][self.laneNum][self.objIndex-1].x +
roadObj[self.direction][self.laneNum][self.objIndex-
1].image.get_rect().width + positionGap))):
                                self.x -= self.speed
        elif(self.direction=='up'):
            if (self.roadObjClass == 'pedestrian1') or (self.roadObjClass ==
'pedestrian2'):
                """
                Control of the movement and crossing of pedestrians going
from the bottom to the top of the screen.
                The logic is the same as the one used in the previous case.
                """
                if(self.crossed==0 and
self.y<stopCoordsPed[self.direction]):
                    self.crossed = 1
                    #print("PEDESTRIAN ALLOWED TO CROSS UP, ALREADY CROSSED
STOPLINE")
                if((self.y>=self.stop or self.crossed == 1 or (nowGreen==1
and nowYellow==0)) and (self.objIndex==0 or
self.y>(roadObj[self.direction][self.laneNum][self.objIndex-1].y +
```

```
roadObj[self.direction][self.laneNum][self.objIndex-
1].image.get_rect().height + positionGap))):
                    self.y -= self.speed
                    #print("PEDESTRIAN ALLOWED TO CROSS UP")
            else:
                """
                Control of the movement and crossing of cars and motorbikes
going from the bottom to the top of the screen.
                The logic is the same as the one used in the previous cases.
                """
                if(self.crossed==0 and self.y<stopCoords[self.direction]):
                    self.crossed = 1
                    roadObj[self.direction]['crossed'] += 1
                    if(self.willTurn==0):
                        roadObjNotTurned[self.direction][self.laneNum].appen
d(self)
                        self.crossedObjIndex =
len(roadObjNotTurned[self.direction][self.laneNum]) - 1
                if(self.willTurn==1):
                    if(self.crossed==0 or self.y>stopCoords[self.direction]-
170):
                        if((self.y>=self.stop or (nowGreen==3 and
nowYellow==0) or self.crossed == 1) and (self.objIndex==0 or
self.y>(roadObj[self.direction][self.laneNum][self.objIndex-1].y +
roadObj[self.direction][self.laneNum][self.objIndex-
1].image.get_rect().height +  positionGap) or
roadObj[self.direction][self.laneNum][self.objIndex-1].turned==1)):
                            self.y -= self.speed
                    else:
                        if(self.turned==0):
                            self.image = self.updateRotationAngles()
                            self.x -= 2
                            self.y -= 1.2

                        else:
                            if(self.crossedObjIndex==0 or
(self.x>(roadObjTurned[self.direction][self.laneNum][self.crossedObjIndex-
1].x + roadObjTurned[self.direction][self.laneNum][self.crossedObjIndex-
1].image.get_rect().width + positionGap))):
                                self.x -= self.speed
                else:
                    if(self.crossed == 0):
                        if((self.y>=self.stop or (nowGreen==3 and
nowYellow==0)) and (self.objIndex==0 or
self.y>(roadObj[self.direction][self.laneNum][self.objIndex-1].y +
```

```
roadObj[self.direction][self.laneNum][self.objIndex-
1].image.get_rect().height + positionGap))):
                                self.y -= self.speed
                    else:
                        if((self.crossedObjIndex==0) or
(self.y>(roadObjNotTurned[self.direction][self.laneNum][self.crossedObjIndex
-1].y + roadObjNotTurned[self.direction][self.laneNum][self.crossedObjIndex-
1].image.get_rect().height + positionGap))):
                                self.y -= self.speed
```

```
    def initializeTrafficSignal():
```

This function initializes four objects of the TrafficSignal class clockwise, with their respective timers.

```
    def iterateUpdateValues():
```

This function is going to allow us to visually control our traffic lights.

```
    def updateValues():
```

This function allows us to update the values of the traffic every delay duration.

```
def initializeTrafficSignal():
    """
    This function initializes four objects of the class TrafficSignal in a
clockwise directions, with their respective timers.

    Args:
        -None

    Returns:
        None

    """
    traffic_signal_1 = TrafficSignal(0, timerYellow, timerGreen[0])
    signalsList.append(traffic_signal_1)
    signalsList.append(TrafficSignal(traffic_signal_1.red+traffic_signal_1.y
ellow+traffic_signal_1.green, timerYellow, timerGreen[1]))
    signalsList.append(TrafficSignal(timerRed, timerYellow, timerGreen[2]))
    signalsList.append(TrafficSignal(timerRed, timerYellow, timerGreen[3]))
    iterateUpdateValues()

def iterateUpdateValues():
    """This function is called repeatedly to control all traffic light
visuals.

    Args:
        - None

    Returns: None

    """
    global nowGreen, nowYellow, nextGreen
    while(signalsList[nowGreen].green>0):
        updateValues()
        time.sleep(1)
    nowYellow = 1

    for i in range(0,3):
        for vehicle in roadObj[directionsNum[nowGreen]][i]:
            vehicle.stop = defaultStopCoords[directionsNum[nowGreen]]
    while(signalsList[nowGreen].yellow>0):
        updateValues()
        time.sleep(2)
    nowYellow = 0
```

```
    signalsList[nowGreen].green = timerGreen[nowGreen]
    signalsList[nowGreen].yellow = timerYellow
    signalsList[nowGreen].red = timerRed

    nowGreen = nextGreen
    nextGreen = (nowGreen+1)%numOfSignals
    signalsList[nextGreen].red =
signalsList[nowGreen].yellow+signalsList[nowGreen].green
    iterateUpdateValues()
# Update values of the signal timers after every second
def updateValues():
    for i in range(0, numOfSignals):
        if(i==nowGreen):
            if(nowYellow==0):
                signalsList[i].green-=1
            else:
                signalsList[i].yellow-=1
        else:
            signalsList[i].red-=1
```

def generateRoadObjects():

This function will allow us to generate objects of the roadObject class randomly. The function creates objects in random positions, directions and lanes. It is also random if the object rotates or not. The function generates them at random distances between 25,50,75 or 100.

```
def generateRoadObjects():
    """This function will be called to generate all type of moving road
objects, which include any object saved in the
    enum `ObjectTypes`.

    Reference: None

    Args:
     - None

    Returns:
     - A road object (motorbike, car, or pedestrian) randomly generated on
different lanes and moving on different directions.

    """

    while(True):
        vehicle_type = random.randint(1,len(OBJECTS_SPEEDS))
        willTurn = 0
        temp = random.randint(0,99)
        if temp<40:
            willTurn = 1
        temp = random.randint(0,99)
        lane_n = random.randint(0,1)
        directionNumber = 0
        dist = [25,50,75,100]

        if(temp<dist[0]):
            directionNumber = 0
        elif(temp<dist[1]):
            directionNumber = 1
        elif(temp<dist[2]):
            directionNumber = 2
        elif(temp<dist[3]):


            directionNumber = 3

        RoadObjects(random.randint(1,2) if vehicle_type <= 2 else lane_n,
roadObjTypes[vehicle_type], directionNumber,
directionsNum[directionNumber],willTurn if vehicle_type <= 2 else 0)
        time.sleep(SLEEP_TIME_1)
```

class Run:

This class is called to activate the simulation of the traffic light violation detection project.

It controls all processes and activates the necessary functions.

It initializes all our images on the screen using set_screen_objects() and set the background with blit_screen(displayScreen, backgroundImage,(0,0))

Displays the images of our traffic lights in the corresponding color. Check the variable nowGreen to place the green traffic light image in the correct coordinates for pedestrians and vehicles.

Display violation alerts if any and display images of vehicles at the correct coordinates.

```
class Run:
    """ This class is called to trigger the simulation of traffic light
violation detection project.
    It controls all of the processes and triggers the necessary functions.

    """
    thread1 =
threading.Thread(name="initialization",target=initializeTrafficSignal,
args=())
    thread1.daemon = True
    thread1.start()

    backgroundImage, greenLightPed, greenLightPed90, greenLightPed180,
greenLightPed270 , \
    redLight, yellowLight, greenLight, redLightPed, redLightPed180,
redLightPed90, \
    redLightPed270 ,font = set_screen_objects()

    thread2 =
threading.Thread(name="generateRoadObjects",target=generateRoadObjects,
args=())
    thread2.daemon = True
    thread2.start()

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()

        blit_screen(displayScreen, backgroundImage,(0,0))
        for i in range(0,numOfSignals):
            if(i==nowGreen):
                if(nowYellow==1):
                    blit_screen(displayScreen, yellowLight, lightCoords[i])
                else:
                    blit_screen(displayScreen, greenLight, lightCoords[i])
            else:
                blit_screen(displayScreen, redLight, lightCoords[i])

        if nowGreen == 1:
            blit_screen(displayScreen, greenLightPed, lightPedCoords[0])
        else:
            blit_screen(displayScreen, redLightPed, lightPedCoords[0])

        if nowGreen == 2:
```

```
            blit_screen(displayScreen, greenLightPed270, lightPedCoords[1])
        else:
            blit_screen(displayScreen, redLightPed270, lightPedCoords[1])

        if nowGreen == 3:
            blit_screen(displayScreen, greenLightPed180, lightPedCoords[2])
        else:
            blit_screen(displayScreen, redLightPed180, lightPedCoords[2])

        if nowGreen == 0:
            blit_screen(displayScreen, greenLightPed90, lightPedCoords[3])
        else:
            blit_screen(displayScreen, redLightPed90, lightPedCoords[3])

        for vehicle in simulation:
            blit_screen(displayScreen, vehicle.image, [vehicle.x,
vehicle.y])
            vehicle.controlMovement()

        if(SIMULATION_VIOLATION):
            VIOLATION.displayViolationImage()

        pygame.display.update()


Run()
```