**File: ./software/data_analysis/analysis.py**

```python
import argparse
from classes.parsing import Parser
import json

parameters_file = "analysis_parameters.json"

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '--save_location', '-s',
    required=True,
    help='Where to save the analysis')
args = argparser.parse_args()

def main():
    try:
        parameters = read_parameters()
        parser = Parser(parameters, args.save_location)
        try:
            parser.parse()
        except Exception as e:
            print("An exception occured parsing the recording files: ", e)
        try:
            parser.save()
        except Exception as e:
            print("An exception occured saving the analysed data: ", e)
    finally:
        print("Finished analysing.")

def read_parameters():
    try:
        with open("analysis_parameters.json", "r") as file:
            parameters = json.load(file)
        return parameters
    except Exception as e:
        print("Something wrong happened trying to get data from {}".format(parameters_file), e)

if __name__ == '__main__':
    main()
```

**File: ./software/data_analysis/classes/parsing.py**

```python
from classes.data_analysers import CollisionDataAnalyser, VehicleLightDataAnalyser, RouteDataAnalyser, LaneMarkingDataAnalyser, SpeedingDataAnalyser, RoadTrafficD
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString

# Creates a DataAnalyser gicen the analysis type.
class DataAnalyserFactory:
    @staticmethod
    def create_analyser(analysis_type, scenarios, participants):
        if analysis_type == 'collision_data':
            return CollisionDataAnalyser(scenarios, participants)
        elif analysis_type == 'lane_marking_violation_data':
            return LaneMarkingDataAnalyser(scenarios, participants)
        elif analysis_type == 'vehicle_light_misuse_data':
            return VehicleLightDataAnalyser(scenarios, participants)
        elif analysis_type == 'route_data':
            return RouteDataAnalyser(scenarios, participants)
        elif analysis_type == 'speeding_data':
            return SpeedingDataAnalyser(scenarios, participants)
        elif analysis_type == 'road_traffic_violation_data':
            return RoadTrafficDataAnalyser(scenarios, participants)
        else:
            raise ValueError('Invalid analyser type')

# Used to read and process recorded data according to the specification provided
# in analysis_parameters.json
class Parser():

    def __init__(self, parameters, save_location):
        self.participants = parameters["participants"]
        self.scenarios = parameters["scenarios"]
        self.metrics = parameters["metrics"]
        self.save_location = '../../data/analysed_data/data/{}'.format(save_location)

        try:
            self.analysers = []
            for metric in self.metrics:
                analyser = DataAnalyserFactory.create_analyser(metric, self.scenarios, self.participants)
                self.analysers.append(analyser)
        except Exception as e:
            print("Could not initiate all the analysers correctly", e)

    # Make all the DataAnalysers do their analyses
    def parse(self):
        for analyser in self.analysers:
            analyser.run()

    # Save the analysed data
    def save(self):
        try:
            for analyser in self.analysers:
                analyser.save_points(self.save_location + "/points")
            self.save_participants()
            self.save_scenarios()
        except Exception as e:
            print("Something wrong happened trying to save the analysed data:", e)

    # Write relevant data to the participants.xml file
    def save_participants(self):
        try:
            root = ET.Element("Participants")
            for participant in self.participants:
                participant_element = ET.SubElement(root, "Participant")
                ET.SubElement(participant_element, "Name").text = str(participant)
                analysis = ET.SubElement(participant_element, "Analysis")
                # For each participant write what each of the analysers calculated
                total_amount_of_points = 0
                for analyser in self.analysers:
                    ET.SubElement(analysis, analyser.save_file_xml_element_name_average).text = str('%.2f' % (analyser.participant_score[participant] / len(self.s
                    if analyser.category != "route_data":
                        total_amount_of_points += analyser.participant_score[participant]

                ET.SubElement(participant_element, "TotalNumberOfPenaltyPoints").text = str('%.2f' % total_amount_of_points)
            xml_str = ET.tostring(root, 'utf-8', method='xml')
            xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
            with open("{}/participants.xml".format(self.save_location, participant), "w+") as file_writer:
                file_writer.write(xml_formatted)
        except IOError as e:
            print("An error occured trying to write analytics to participants.xml file:", e)
        except Exception as e:
            print("Something wrong happened trying to save participants analytics:", e)

    # Write relevant data to the scenarios.xml file
    def save_scenarios(self):
        try:
            root = ET.Element("Scenarios")
            for scenario in self.scenarios:
                scenario_element = ET.SubElement(root, "Scenario")
                ET.SubElement(scenario_element, "Name").text = str(scenario)
                analysis = ET.SubElement(scenario_element, "Analysis")
                # For each scenario write what each of the analysers calculated
                for analyser in self.analysers:
                    ET.SubElement(analysis, analyser.save_file_xml_element_name_average).text = str('%.2f' % (analyser.scenario_score[scenario] / len(self.partic
            # Write to file
            xml_str = ET.tostring(root, 'utf-8', method='xml')
            xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
            with open("{}/scenarios.xml".format(self.save_location, scenario), "w+") as file_writer:
                file_writer.write(xml_formatted)
        except IOError as e:
            print("An error occured trying to write analytics to scenarios.xml file:", e)
        except Exception as e:
            print("Something wrong happened trying to save scenario analytics:", e)


```

**File: ./software/data_analysis/classes/data_analysers.py**

```python
import os
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString

# DataAnalysers are used to parse data from files according to their type and then save it to analysed data files
class DataAnalyser:
    def __init__(self, scenarios, participants):
        self.path_to_recordings = '../../data/recordings/'
        self.scenarios = scenarios
        self.participants = participants

        # A list of pairs: (participant, score_on_all_scenarios)
        self.participant_score = {participant: 0.0 for participant in participants}

        # A list of pairs: (scenario, score_of_all_participants)
        self.scenario_score = {scenario: 0.0 for scenario in scenarios}

        # A list of pairs: (scenario, interesting points of all participants)
        self.scenario_points = {scenario: [] for scenario in scenarios}

    # This method goes through the recording files and processes files that are relevant
```

```
    def analyse_files(self, xml_category):
        try:
            # First it looks for a directory for each participant according to the participants specified
            # in the analysis.parameters.json
            for participant in self.participants:
                participant_directories = os.listdir(self.path_to_recordings)
                if participant in participant_directories:
                    item_path = os.path.join(self.path_to_recordings, participant)
                    if os.path.isdir(item_path):
                        # Then it looks for a directory for each scenario specified in the analysis.parameters.json
                        for scenario in self.scenarios:
                            scenario_directories = os.listdir(item_path)
                            if scenario in scenario_directories:
                                path = os.path.join(item_path, scenario, "{}.xml".format(self.category))
                                # And then it looks for a file according to the analysers category
                                # If one is found, it is processed
                                if os.path.isfile(path):
                                    try:
                                        self.process_file(path, scenario, participant, xml_category)
                                    except Exception as e:
                                        print("An exception occured trying to process file {}:".format(path), e)
        except Exception as e:
            print("An exception occured trying to gather data from recording files:", e)

    # Used to process a single file
    def process_file(self, path_to_file, scenario_name, participant_name, xml_category):
        tree = ET.parse(path_to_file)
        root = tree.getroot()
        if self.category == "route_data":
            penalty_points_total = root.find("ProportionOfRouteCompleted").text
            location_data = []
            for instance in root.findall("{}/Instance".format(xml_category)):
                if instance.find("WasReached").text == "0":
                    x = instance.find("Location").find("X").text
                    y = instance.find("Location").find("Y").text
                    location_data.append((x, y))
        else:
            penalty_points_total = root.find("PenaltyPointsTotal").text
            location_data = []
            for instance in root.findall("{}/Instance/Location".format(xml_category)):
                x = instance.find("X").text
                y = instance.find("Y").text
                location_data.append((x, y))

        self.scenario_points[scenario_name].extend(location_data)
        self.scenario_score[scenario_name] += float(penalty_points_total)
        self.participant_score[participant_name] += float(penalty_points_total)

    # Used to write all gathered point to the points directory.
    def save_points(self, path):
        try:
            for scenario, points_list in self.scenario_points.items():
                root = ET.Element("Points")
                for x, y in points_list:
                    point = ET.SubElement(root, "Point")
                    ET.SubElement(point, "X").text = str(x)
                    ET.SubElement(point, "Y").text = str(y)

                xml_str = ET.tostring(root, 'utf-8', method='xml')
                xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

                with open("{}/{}_{}.xml".format(path, scenario, self.category), "w+") as file_writer:
                    file_writer.write(xml_formatted)
        except IOError as e:
            print("An error occured trying to save points of {}:".format(self.category), e)
        except Exception as e:
            print("Something wrong happened trying to save participants analytics:", e)

    # Run the data analyser
    def run(self):
        self.process_data()

    # Abstract method
    def process_data(self):
        raise NotImplemented("The method being called is abstract")



# Different data analysers that look for different metrics
class CollisionDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "collision_data"
        self.save_file_xml_element_name_average = "CollisionPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("CollisionInstances")


class VehicleLightDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "vehicle_light_misuse_data"
        self.save_file_xml_element_name_average = "VehicleLightMisusePenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("VehicleLightsMisuseInstances")

class RouteDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "route_data"
        self.save_file_xml_element_name_average = "ProportionOfRouteCompletedAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("Waypoints")


class LaneMarkingDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "lane_marking_violation_data"
        self.save_file_xml_element_name_average = "LaneMarkingViolationPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("LaneMarkingViolationInstances")


class SpeedingDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "speeding_data"
        self.save_file_xml_element_name_average = "SpeedingPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("SpeedingInstances")


class RoadTrafficDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
```

```
        self.category = "road_traffic_violation_data"
        self.save_file_xml_element_name_average = "RoadTrafficViolationPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("RoadTrafficViolationInstances")
```

**File: ./software/data_analysis/score_calculator.py**

```
import argparse
import json
import xml.etree.ElementTree as ET

# The gamma value
GAMMA = 0.7

metrics = [
        "collision_data",
        "lane_marking_violation_data",
        "vehicle_light_misuse_data",
        "route_data",
        "speeding_data",
        "road_traffic_violation_data"
    ]

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '--scenario', '-s',
    help='The simulation to consider')
argparser.add_argument(
    '--name', '-n',
    help='The name of the participant')
args = argparser.parse_args()

# Used to print the final score of the participant to the screen
def main():
    try:
        details = read_details()
        try:
            score, ideal_score = calculate_the_score(details)
            print("The score for {} in {} is: ".format(args.name, args.scenario) + str(score))
            print("The ideal score is: {} and above".format(ideal_score))
        except Exception as e:
            print("An exception occured calculating the score:", e)
    except Exception as e:
        print("Something went wrong...", e)

# Function that calculates the final score according to the formula specified in the final report Chapter 7
def calculate_the_score(details):
    score = 0
    d = details["difficulty"]
    alpha = details["intensity"]
    v_avg = details["average_speed"]
    s = details["distance"]
    c, t, penalty_points = do_route_analysis(args.scenario, args.name)
    time_for_additional_stops = sum([entry["time"] for entry in details["stops"]])

    # The final formula in action
    t_o = (s / v_avg) * (1 + alpha) + time_for_additional_stops
    score = (c * (t_o / t) * d) - GAMMA * penalty_points
    return float('%.3f' % score), d

# Sum up all the penalty points
def do_route_analysis(scenario, name):
    penalty_points = 0
    try:
        for metric in metrics:
            tree = ET.parse("../../data/recordings/{}/{}/{}.xml".format(name, scenario, metric))
            root = tree.getroot()
            if metric == "route_data":
                c = float(root.find("ProportionOfRouteCompleted").text)
                t = float(root.find("SimulationTime").text[:-1])
            else:
                penalty_points += float(root.find("PenaltyPointsTotal").text)
    except Exception as e:
        print("An exception occured trying to gather data from recording files:", e)

    return c, t, penalty_points

# Read the details of the simulation
def read_details():
    try:
        with open("../../data/simulation_details/{}.json".format(args.scenario), "r") as file:
            details = json.load(file)
        return details
    except Exception as e:
        print("Something wrong happened trying to get data from {}.json".format(args.scenario), e)

if __name__ == '__main__':
    if args.name and args.scenario:
        main()
    else:
        "Please make sure that you pass the required arguments -s (scenario name) and -n (participant's name)"
```

**File: ./software/data_analysis/README.md**

# Data Analysis

## Introduction
This directory is dedicated to processing the recorded simulation data and presenting it in a meaningful way. In addition, it contains tools to evaluate the perfor
## Content
The directory contains the following:
- analysis_parameters.json: A JSON file that holds information about the participants, scenarios, and metrics to be analyzed.
- analysis.py: A Python script that performs the data analysis based on the parameters defined in the analysis_parameters.json file.
- run_analysis.sh: A shell script to run the analysis using the analysis.py script.
- visualisation: Subdirectory containing scripts that for graphical representation.
- classes: Additional classes to help the analysis of recorded data files.
- score_calculator.py: a script to calculate the score of the participant in the given scenario
- scenario_analyser.py: a script to analyse the scenarios generating files that help to evaluate the performance of a participant

## analysis_parameters.json
Imagine that analysis_parameters.json file has the following structure:

```
{
    "participants": [
        "ParticipantA",
        "ParticipantB"
    ],
    "scenarios": [
        "scenario1"
    ],
    "metrics": [
        "collision_data",
        "road_traffic_violation_data"
    ]
}
```
This file states that we want to analyze the collisions and road traffic violation data from scenario1 driven by Participants A and B.

**The list of all possible metrics:**

```
    "collision_data",
    "lane_marking_violation_data",
    "vehicle_light_misuse_data",
    "route_data",
    "speeding_data",
    "road_traffic_violation_data"
```

## Running the Analysis
To run the analysis, simply execute the following command in the terminal:

```
bash run_analysis.sh name_of_analysis
```
Analysed data will be saved in the ./analysed_data/data/name_of_analysis directory.

Replace "name_of_analysis" with a desired name for the analysis instance.

## Results
The analysis tool will process the recording according to the parameters specified in the analysis_parameters.json file and produce the following files:
- participants.xml: Contains information about each of the specified participants and their average scores.
- scenarios.xml: Contains information about each of the specified scenarios and the average scores of participants.
- n .xml files in the points subdirectory: Each file contains a list of x and y coordinates, representing interesting patterns in the data.

For instance, the file scenario1_collision_data.xml will contain all the collision points for Participants A and B driving scenario1.

## Visual Representation
For visual representation of the data, please refer to the subdirectory [visualisation](./visualisation/).

## Scenario analyser
Before evaluating the score of a participant, we need to run the scenario analyser that will gather data from the simulated world and will create a file that will

The `scenario_analyser.py` script takes two arguments: -p indicating the path file and -s indicating the scenario file. It creates a file in the [simulation_detail

## Score evaluation
The score evaluation is performed using the `score_calculator.py` script. The script takes arguments -n indicating the name of the participant and -s argument indi

**File: ./software/data_analysis/run_analysis.sh**

```
#!/bin/bash

name=$1


if [ -z "$name" ]; then
    echo "If you want to run data analysis, the name for the analysis has to be specified."
    echo "Example use: ./run_analysis.sh name"
    exit 1
fi

cd ../../data/analysed_data/data/

if [ -d "$name" ]; then
  echo "Directory $name already exists. Please choose another name. Aborting."
  exit 1
fi

mkdir $name
mkdir ./$name/points

cd ../../../software/data_analysis
python analysis.py -s $name
```

**File: ./software/data_analysis/visualisation/README.md**

```
## Information about drawing

This directory contains subdirectories for different kind of visual analysis methods.

Please refer to the subdirectories for more details.
```

**File: ./software/data_analysis/visualisation/diagram_drawing/draw_diagram.py**

```python
import xml.etree.ElementTree as ET
import argparse
import seaborn as sns
import matplotlib.pyplot as plt
import sys

def draw_bar_chart(args):
    # Parse XML file
    try:
        tree = ET.parse(args.xml_file)
    except ET.ParseError as e:
        print(f"Error parsing XML file: {e}")
        sys.exit(1)
    root = tree.getroot()
    names = []
    scores = []
    for participant in root:
        name = participant.find("Name").text
        try:
            score = float(participant.find("Analysis").find(args.category).text)
        except ValueError as e:
            print(f"Error parsing score value: {e}")
            sys.exit(1)
        names.append(name)
        scores.append(score)

    # Plot bar chart
    sns.set_palette("Blues", color_codes=True)
    sns.set_style("darkgrid")
    sns.set(font_scale=1.3)

    sns.set_context("notebook")
    sns.barplot(x=names, y=scores)
    plt.xlabel(args.xlabel)
    plt.ylabel("Penalty Points")
    plt.title(args.title)
    if args.save:
        try:
            plt.savefig(args.save)
        except Exception as e:
            print(f"Error saving the diagram: {e}")
            sys.exit(1)
    plt.show()

def main():
    # Parse command line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("--xml_file", type=str, help="Path to the XML file")
    parser.add_argument("--category", type=str, help="Category of the data")
    parser.add_argument("--save", type=str, help="Path to save the diagram")
    parser.add_argument("--title", type=str, help="Title of the diagram")
    parser.add_argument("--xlabel", type=str, help="xlabel text")
    args = parser.parse_args()

    # Draw bar chart
    draw_bar_chart(args)

if __name__ == "__main__":
    main()
```

**File: ./software/data_analysis/visualisation/diagram_drawing/README.md**

```markdown
## Information about diagram drawing

To draw the diagram look at the [draw_diagram_example.sh](./draw_diagram_example.sh).

**The list of all possible categories:**

```
      CollisionPenaltyPointsAverage
      LaneMarkingViolationPenaltyPointsAverage
      VehicleLightMisusePenaltyPointsAverage
      ProportionOfRouteCompletedAverage
      SpeedingPenaltyPointsAverage
      RoadTrafficViolationPenaltyPointsAverage
```
- save_filename - specifies the name of the file that the diagram should be saved to. This file is an image.
- title - specifies what shoudl be written at the top of the diagram
- xlabel - x axis label
- **data_filename** - points to the analysed data directory and then to either participants.xml or scenarios.xml. Please note, that at this moment, the diagrams c
```

**File: ./software/data_analysis/visualisation/diagram_drawing/draw_diagram_example.sh**

```bash
#!/bin/bash

####### ONLY THIS BIT IS MEANT TO BE EDITED #######
category="LaneMarkingViolationPenaltyPointsAverage"
title="Lane Marking Violation Penalty Points Average in Scenario 1"
xlabel="Participants"
save_filename="example_participants"
data_filename="example_analysis/participants.xml"
####### ####### ####### ####### ####### ####### ###

data_path="../../../../data/analysed_data/data/$data_filename"
save_path="../../../../data/analysed_data/diagrams/$save_filename"

python draw_diagram.py --xml_file "$data_path" --category "$category" --save "$save_path" --title "$title" --xlabel "$xlabel"
```

**File: ./software/data_analysis/visualisation/map_drawing/draw_map.sh**

```
#!/bin/bash

# map specified which commonroad format file to use to depict the map layout
map=$1

# first_file is path to an .xml file containing the locations to mark on the map
first_file=$2

# second_file contains locations the same way the first_file does
second_file=$3

# For this, neccessary python libraties need to be installed
# More information on this can be found in the python_libraries directory

# Check if any of the arguments were provided
if [[ -z "$map" || -z "$first_file" ]]; then
    # If either map or first_file is missing, exit the script and echo a message
    echo "Error: missing arguments. Please specify map and first_file with points."
    exit 1
fi

# Check if map or first_file is missing
if [[ ! -f "../../../../data/maps/commonroad_format/$map" || ! -f "../../../../data/analysed_data/data/$first_file" ]]; then
    # If either map or first_file is missing, exit the script and echo a message
    echo "Error: map file or first_file does not exist. Please check the paths provided."
    exit 1
fi

# Launch crdesigner with appropriate arguments and disable any output to the terminal
echo "Drawing the map and marking the points."
if [ -n "$second_file" ]; then
    crdesigner -p1 "../../../../data/analysed_data/data/$first_file" -i "../../../../data/maps/commonroad_format/$map" -p2 "../../../../data/analysed_data/data/$se
else
    crdesigner -p1 "../../../../data/analysed_data/data/$first_file" -i "../../../../data/maps/commonroad_format/$map" > /dev/null 2>&1
fi
echo "Finished drawing the map and points."

# Remove files directory and its contents
rm -rf "files"
```

**File: ./software/data_analysis/visualisation/map_drawing/README.md**

```
## Data visualisation on the map

This directory contains scripts that visualise the specific points on the map using the modified crdesigner module which can be found [here](../../../python_libra

Please note that I am not the author of the crdesigner module. It is a slightly modified version allowing for additional files to be specified when launching it re

To use the map drawing tool, please refer to the `draw_map_example.sh` script.

This script takes a CommonRoad type XML map description and a list of points from the analysed data. An example of how files containing points look like can be fou

The `draw_map_example.sh` script also allows to provide a second file containing points.

Points from the first file are marked in blue color and points form the second file are marked in red color. This can be useful to analyse where human drivers and
```

**File: ./software/data_analysis/visualisation/map_drawing/draw_map_example.sh**

```
#!/bin/bash

first_file="example_analysis/points/scenario2_collision_data.xml"
#second_file="example_analysis/points/scenario1_collision_data.xml"
map="Town07_commonRoad.xml"

# Second file can also be passed into the draw_map script to draw two depict two different datasets on the map
bash draw_map.sh $map $first_file #$second_file
```

**File: ./software/data_analysis/scenario_analyser.py**

```
import glob
import os
import sys
import time
import random
import logging
import argparse
import xml.etree.ElementTree as ET
import numpy
import json
sys.path.append("..")
from carla_scripts.config import SAMPLING_RESOLUTION
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla


argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '--scenario', '-s',
    help='The scenario to consider')
argparser.add_argument(
    '--path', '-p',
    help='The path to consider')
args = argparser.parse_args()


# From location makes a route
def get_route(map, locations):
    waypoints = []
    grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)

    for i in range(len(locations)-1):
        waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
    return waypoints

# Read the path file
def read_path_file():
    file_path = "../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
```

```
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations, data["town"]
    except Exception as e:
        print("Could not read the file {}".format(args.path))

# Retrieve road elements from the OpenDRIVE file
def get_road_elements(map):
        path_to_file = "../../data/maps/opendrive_format/{}.xodr".format(map)
        tree = ET.parse(path_to_file)
        root = tree.getroot()

        # Get all road elements from the XML
        roads = []
        for road in root.findall("./road"):
            roads.append(road)
        return roads

# Get total distance of all waypoints
def get_total_distance(waypoints):
    total_distance = 0
    for i in range(len(waypoints)-1):
        distance = waypoints[i][0].transform.location.distance(waypoints[i+1][0].transform.location)
        total_distance += distance
    return total_distance

# Calculate the intensity and retrieve the difficulty
def analyse_scenario(total_number_of_spawn_points):
    file_path = "../../data/scenario_generation_data/generated_scenarios/{}.json".format(args.scenario)
    try:
        with open(file_path) as file:
            data = json.load(file)
            total_vehicle_amount = int(data["number_of_vehicles"]) + int(data["number_of_two_wheel_vehicles"])
            # Assume that at most 80% of the spawn points are viable since if we were to put cars into all of them, it would result in the biggest gridlock in the
            possible_spawn_point_amount = int(0.8 * total_number_of_spawn_points)
            intensity = total_vehicle_amount / possible_spawn_point_amount
            if intensity > 1: intensity = 1

            return int(data["difficulty"]), intensity
    except Exception as e:
        print("Could not read the file {}".format(file_path))

# Calculate the average speed of all waypoints and also return how many road elements do not have the speed specified at all
def get_average_speed(waypoints, roads):
    road_speed, how_many_elements_have_no_max_speed_specified = get_road_speed_dictionary(roads)
    average_speed = 0
    speed_limits = []

    for waypoint in waypoints:
        id = waypoint[0].road_id
        if waypoint[0].road_id is not None and id in road_speed:
            speed_limits.append(road_speed[waypoint[0].road_id])

    if len(speed_limits) > 0:
        average_speed = sum(speed_limits) / len(speed_limits)

    return average_speed, how_many_elements_have_no_max_speed_specified

# Create a dictionary of road_id ---> max allowed speed
def get_road_speed_dictionary(roads):
    speed_dict = {}
    i = 0
    for road in roads:
        id = int(road.get('id'))
        type = road.find('type')
        speed_elem = None

        if type is not None:
            speed_elem = type.find('speed')

        if speed_elem is None:
            # The standard speed --> 50km/h converted to m/s
            i += 1
            speed_dict[id] = float(50 / 3.6)
        else:
            unit = speed_elem.get('unit')
            value = float(speed_elem.get('max'))
            if unit == 'mph':
                speed_dict[id] = value * 1.60934 / 3.6
            else:
                speed_dict[id] = value / 3.6

    return speed_dict, i

# For now count how many junctions there are and for each assign a time value
def get_stops(waypoints):
    stops = []
    junctions = 0
    last_waypoint = "road"
    for waypoint in waypoints:
        if waypoint[0].is_junction:
            if last_waypoint == "road":
                junctions += 1
                last_waypoint = "junction"
        else:
            if last_waypoint == "junction":
                last_waypoint = "road"

    for i in range (junctions):
        stops.append({
            'name': "Junction",
            'time': 12
        })
    return stops


# Save scenario to file
def write_to_file(obj):
    with open("../../data/simulation_details/{}.json".format(args.scenario), "w+") as f:
        json.dump(obj, f, indent = 4)


# Used to analyse the path and the scenario to retireve important data needed to evaluate the performance of the participants
def main():
    try:
        client = carla.Client("localhost", 2000)
        client.set_timeout(10)
        locations, map_name = read_path_file()

        # Change the map to get the map element
        world = client.load_world(map_name)
        map = world.get_map()

        waypoints = get_route(map, locations)
        roads = get_road_elements(map_name)

        total_distance = get_total_distance(waypoints)
        difficulty, intensity = analyse_scenario(len(map.get_spawn_points()))
        average_speed, how_many_elements_have_no_max_speed_specified = get_average_speed(waypoints, roads)

        stops = get_stops(waypoints)
```

```
183
184         final_obj = {
185             "path": args.path,
186             "scenario": args.scenario,
187             "difficulty": float('%.3f' % difficulty),
188             "intensity": float('%.3f' % intensity),
189             "distance": float('%.3f' % total_distance),
190             "average_speed": float('%.3f' % average_speed),
191             "stops": stops
192         }
193
194         write_to_file(final_obj)
195
196         # Uncomment to see how (not) wll the OpenDRIVE maps in carla are specified
197         # print("")
198         # print("In {} only {}% of all road elements have speed values specified".format(map_name, ('%.2f' % (100 - how_many_elements_have_no_max_speed_specified
199         # print("")
200         print("Successfully analysed and save data to data/scenario_details/{}.json file".format(args.scenario))
201     except Exception as e:
202         print("Something has gone wrong.", e)
203
204 if __name__ == '__main__':
205     main()
```

**File: ./software/carla_scripts/replay.sh**

```
1 #!/bin/bash
2
3 file="$1_scenario$2_recording.log"
4 python3 replayer.py -f $file
```

**File: ./software/carla_scripts/path_generator.py**

```
1  import argparse
2  import json
3  import logging
4  from scenarios.path_builder import PathBuilder
5
6  parser = argparse.ArgumentParser(description="Generate path parameters for a given scenario")
7  parser.add_argument('-s', '--scenario', default='example', help='Name of scenario')
8  parser.add_argument('-f', '--filename', default='example', help='Filename')
9  args = parser.parse_args()
10
11 logger = logging.getLogger(__name__)
12 logging.basicConfig(level=logging.INFO, filename="../../data/paths/logs/logs.log", filemode="w", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")
13
14 # Get a file indicating scenario parameters
15 def load_scenario():
16     logger.info("Loading scenario {} data".format(args.scenario))
17     file_path = "../../data/scenario_generation_data/generated_scenarios/{}.json".format(args.scenario)
18     try:
19         with open(file_path) as file:
20             scenario_data = json.load(file)
21             logger.info("Successfully read scenario {}.json data".format(args.scenario))
22             return scenario_data
23     except FileNotFoundError as e:
24         logger.exception("Scenario {} file could not be found.".format(args.scenario), e)
25     except json.JSONDecodeError as e:
26         logger.exception("The file {}.json could not be decoded as JSON".format(args.scenario), e)
27
28 # The main script
29 def main(agrs):
30     logger.info("Begin path generation")
31     try:
32         scenario_data = load_scenario()
33         path = PathBuilder.generate_details(scenario_data, args.filename)
34         if path: print("Generated and saved.")
35     except Exception as e:
36         logger.fatal("Fatal error. Shutting down the generation.")
37
38 if __name__ == '__main__':
39     main(args)
```

**File: ./software/carla_scripts/run.py**

```
1  import argparse
2  import json
3  import os
4  import shutil
5  import logging
6  import random
7  from simulation_manager import SimulationManager
8  import subprocess
9
10 parser = argparse.ArgumentParser(description="Run all the scenarios for the participant")
11 parser.add_argument('-n', '--name', default='example', help='Name of the participant')
12 parser.add_argument(
13     '--ai',
14     action='store_true',
15     help='Is the ai driving?')
16 args = parser.parse_args()
17
18 logger = logging.getLogger(__name__)
19 logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %(
20
21 scenario_list_file = "scenario_list.json"
22
23 # Read the scenario json file and extract all names of the scenarios to run
24 def get_scenarios():
25     try:
26         with open(scenario_list_file) as file:
27             scenarios = json.load(file)
28             logger.info("Successfully extracted scenarios from file {}".format(scenario_list_file))
29             return scenarios
30     except FileNotFoundError as e:
31         logger.error("Scenario list file could not be found.", exc_info=True)
32         raise e
33     except json.JSONDecodeError as e:
34         logger.error("The file {} could not be decoded as JSON".format(scenario_list_file), exc_info=True)
35         raise e
36
37 # Create a directory in the participant's results directory to store results about the simulation of a particular scenario
38 def create_directory_for_scenario(scenario_name):
39     path = "../../data/recordings/{}/{}".format(args.name, scenario_name)
40     if os.path.exists(path):
41         shutil.rmtree(path)
42     try:
43         os.makedirs(path)
44     except OSError:
45         logger.error("Could not create a directory for scenario {} for participant {}".format(scenario_name, args.name))
```

```
        raise Exception
    logger.info("Successfully created a directory for scenario {}".format(scenario_name))

def try_to_get_name_for_new_scenario():
    path = "../../data/scenario_generation_data/generated_scenarios/"
    i = 0
    # Try to find a name for a new scenario file for a 1000000 times
    while i < 1000000:
        name = "scenario{}".format(i)
        path = os.path.join(path, name)
        if not os.path.exists("{}.json".format(path)):
            return name
        i += 1
    raise Exception("Could not find an unoccupied name for the new scenario. Most likely names scenario[1-1000000] are taken")

def check_scenario(scenario):
    if scenario["name"] is not None:
        return scenario["name"]
    else:
        logger.info("The scenario name was null. Predicting a new scenario.")
        # Generate a new scenario and give it a name and return it
        try:
            name = try_to_get_name_for_new_scenario()
            difficulty = random.randint(1, 1000)
            logger.info("Scenario generator will now try to create a new scenario named {} of difficulty {}".format(name, difficulty))
            # Launch the first person driving mode
        except Exception as e:
            logger.error("The scenario file was not specified and also a new one could not be created", exc_info=True)
            raise e

        try:
            logger.info("Launching create_scenario.py script.")
            # python3: can't open file '../software/generating_scenarios/create_scenario.py'
            process = subprocess.run(["python3", "../../software/generating_scenarios/create_scenario.py", "-d{}".format(difficulty), "-f{}".format(name)], stdou
            # Uncomment to print to the terminal
            # print(process.stdout.decode(), process.stderr.decode())
            logger.info("Successfully generated a new scenario {}".format(name))
            return name
        except Exception as e:
            logger.error("Could not open the driver.py script", exc_info=True)
            raise e

current_simulation_manager = None

# Launch a simulation manager for the scenario
def launch_scenario(scenario):
    name = check_scenario(scenario)
    create_directory_for_scenario(name)
    global current_simulation_manager
    current_simulation_manager = SimulationManager(name, scenario["details"], args.name, scenario["vehicle"], scenario["randomization_seed"], args.ai)
    current_simulation_manager.begin_simulation()


# The main script
def main(agrs):
    logger.info("THE SIMULATIONS BEGIN NOW")
    failed_scenarios = []
    try:
        data = get_scenarios()
        for scenario in data["scenarios"]:
            try:
                logger.info("Starting to work on scenario {}".format(scenario["name"]))
                launch_scenario(scenario)
            except Exception as e:
                failed_scenarios.append(scenario["name"])
                logger.error("Failed to simulate the scenario {}".format(scenario), exc_info=True)

        # The end of all the simulations
        logger.info("Successfully completed {} out of {} simulations".format((len(data["scenarios"])-len(failed_scenarios)), len(data["scenarios"])))
        if len(failed_scenarios) > 0:
            logger.warning("The failed scenarios are:")
            for f_s in failed_scenarios:
                logger.warning(f_s)
    except KeyboardInterrupt:
        logger.warning("The simulation was quit manually.")
    except Exception as e:
        logger.error("Something went horribly wrong.", exc_info=True)
    finally:
        global current_simulation_manager
        if current_simulation_manager is not None:
            current_simulation_manager.close_scenario()
        logger.info("Shutting down the simulation.")

if __name__ == '__main__':
    main(args)
```

**File: ./software/carla_scripts/replayer.py**

```python
#!/usr/bin/env python

# Copyright (c) 2019 Computer Vision Center (CVC) at the Universitat Autonoma de
# Barcelona (UAB).
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

import glob
import os
import sys

import carla

import argparse

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

# Used to replay the specified recording
# Provide the neccessary args
def main():

    argparser = argparse.ArgumentParser(
        description=__doc__)
    argparser.add_argument(
        '--host',
        metavar='H',
        default='127.0.0.1',
        help='IP of the host server (default: 127.0.0.1)')
    argparser.add_argument(
        '-p', '--port',
        metavar='P',
        default=2000,
        type=int,
        help='TCP port to listen to (default: 2000)')
    argparser.add_argument(
        '-s', '--start',
        metavar='S',
        default=0.0,
        type=float,
        help='starting time (default: 0.0)')
    argparser.add_argument(
        '-d', '--duration',
        metavar='D',
        default=0.0,
        type=float,
        help='duration (default: 0.0)')
    argparser.add_argument(
        '-f', '--recorder-filename',
        metavar='F',
        default="test.log",
        help='recorder filename (test1.log)')
    argparser.add_argument(
        '-c', '--camera',
        metavar='C',
        default=0,
        type=int,
        help='camera follows an actor (ex: 82)')
    argparser.add_argument(
        '-x', '--time-factor',
        metavar='X',
        default=1.0,
        type=float,
        help='time factor (default 1.0)')
    argparser.add_argument(
        '-i', '--ignore-hero',
        action='store_true',
        help='ignore hero vehicles')
    argparser.add_argument(
        '--spawn-sensors',
        action='store_true',
        help='spawn sensors in the replayed world')
    args = argparser.parse_args()

    try:
        print("Beginning to replay the recording at {}".format(args.recorder_filename))
        client = carla.Client(args.host, args.port)
        client.set_timeout(120.0)

        # Set the time factor for the replayer
        client.set_replayer_time_factor(args.time_factor)

        # Set to ignore the hero vehicles or not
        client.set_replayer_ignore_hero(args.ignore_hero)

        # Replay the session
        client.replay_file(args.recorder_filename, args.start, args.duration, args.camera, args.spawn_sensors)

    except KeyboardInterrupt:
        pass
    except Exception as e:
        print("Something wrong happened trying to replay the recording:", e)
    finally:
        print("Stopped replaying.")


if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/config.py**

```
# General variables

DISTANCE_FROM_WAYPOINT = 0.7
FINISH_DISTANCE_FROM_WAYPOINT = 3.0
DISTANCE_FROM_WAYPOINT_CHANGING_LANE = 1.0
SAMPLING_RESOLUTION = 0.5
SAMPLING_RESOLUTION_AI = 5
FPS = 60
RANDOMIZATION_SEED = 187349

# Penalty related variables

NO_BEAMS_NO_FOG_LIGHTS = 50
NO_BEAMS_NO_FOG_LIGHTS_TEXT = "Dark and foggy - no lights at all"

NO_BEAMS = 30
NO_BEAMS_TEXT = "Dark - no low beams"

NO_FOG_LIGHTS = 10
NO_FOG_LIGHTS_TEXT = "Dark and foggy - no fog lights"

HIT_PEDESTRIAN_TEXT = "Hit pedestrian"
HIT_PEDESTRIAN_PENALTY = 600
HIT_PEDESTRIAN_PENALTY_SPEEDING = 1200

HIT_VEHICLE_TEXT = "Hit vehicle"
HIT_VEHICLE_PENALTY = 250
HIT_VEHICLE_PENALTY_SPEEDING = 500

HIT_BICYCLE_TEXT = "Hit bicycle"
HIT_BICYCLE_PENALTY = 400
HIT_BICYCLE_PENALTY_SPEEDING = 800

HIT_ROAD_OBJECT_TEXT = "Hit road object"
HIT_ROAD_OBJECT_PENALTY = 150
HIT_ROAD_OBJECT_PENALTY_SPEEDING = 300

RED_LIGHT_TEXT = "Red light"
RED_LIGHT_PENALTY = 50
RED_LIGHT_PENALTY_SPEEDING = 100

SOLID_TEXT = "Solid lane marking"
SOLID_PENALTY = 20
SOLID_PENALTY_SPEEDING = 60

DOUBLE_SOLID_TEXT = "Double solid lane marking"
DOUBLE_SOLID_PENALTY = 40
DOUBLE_SOLID_PENALTY_SPEEDING = 100

BROKEN_NO_TURN_INDICATOR_TEXT = "Broken line wrong or no turn indicator"
BROKEN_NO_TURN_INDICATOR_PENALTY = 10
BROKEN_NO_TURN_INDICATOR_PENALTY_SPEEDING = 30

LIGHT_SPEEDING_PENALTY = 1
LIGHT_SPEEDING_TEXT = "Speeding under 20km/h"

HEAVY_SPEEDING_PENALTY = 3
HEAVY_SPEEDING_TEXT = "Speeding over 20km/h"

# If speeding exceeds this amount, it is considered heavy speeding (50 allowed ---> 71+ is considered heavy speeding)
HEAVY_SPEEDING_THRESHOLD = 20
```

**File: ./software/carla_scripts/classes/route.py**

```
from config import FINISH_DISTANCE_FROM_WAYPOINT, DISTANCE_FROM_WAYPOINT, DISTANCE_FROM_WAYPOINT_CHANGING_LANE
from classes.helpers import distanceBetweenTwoLocations

class Route():

    def __init__(self, waypoints):
        self.routepoints = []
        for waypoint in waypoints:
            self.routepoints.append((waypoint, 0))
        # Last routepoint is out finish point
        self.finish_lane_waypoints = self.get_finish_lane_waypoints(waypoints[len(waypoints) - 1][0])


    def get_finish_lane_waypoints(self, waypoint):
        return [*self.getAllLeftWaypointsRecursively(waypoint, [], 0), *self.getAllRightWaypointsRecursively(waypoint, [], 0), waypoint]

    # Return percent of route completed
    def get_route_completion(self):
        length = len(self.routepoints)
        completed = sum(x[1] for x in self.routepoints)
        return '%.2f' % (completed / length)

    # Get shortest distance comparing each waypoint in waypoints
    # with the provided location
    def get_shortest_distance(self, location):
        shortest_distance = None
        for routepoint in self.routepoints:
            waypoint_location = routepoint[0][0].transform.location
            distance = distanceBetweenTwoLocations(location, waypoint_location)
            if shortest_distance is None or distance < shortest_distance:
                shortest_distance = distance
        return shortest_distance

    # Return true if the last location on route was reached
    def is_finished(self, player_location):
        for waypoint in self.finish_lane_waypoints:
            distance = distanceBetweenTwoLocations(player_location, waypoint.transform.location)
            if distance <= FINISH_DISTANCE_FROM_WAYPOINT:
                return True
        return False

    # Marks the routepoints (x, 1) if the routepoint's distance is <= DISTANCE_FROM_WAYPOINT
    def recalculate(self, location):
        for i in range(len(self.routepoints)):
            routepoint = self.routepoints[i]
            waypoint_location = routepoint[0][0].transform.location
            distance = distanceBetweenTwoLocations(location, waypoint_location)
            if distance <= DISTANCE_FROM_WAYPOINT:
                waypoint = routepoint[0]
                self.routepoints.pop(i)
                self.routepoints.append((waypoint, 1))

        # print("Completed: " + str(self.get_completage()))

    ###          ADVANCED STUFF

    # Marks the routepoints (x, 1) if the routepoint's distance is <= DISTANCE_FROM_WAYPOINT
    def advanced_recalculate(self, location, is_changing_lane):
        for i in range(len(self.routepoints)):
            routepoint = self.routepoints[i]
            waypoint = routepoint[0]
            left_waypoints = self.getLeftValidWaypointsRecursively(waypoint[0], [], 0)
```

```
            right_waypoints = self.getRightValidWaypointsRecursively(waypoint[0], [], 0)
            waypoints = [*left_waypoints, *right_waypoints, waypoint[0]]
            for w in waypoints:
                distance = distanceBetweenTwoLocations(location, w.transform.location)
                if is_changing_lane:
                    if distance <= DISTANCE_FROM_WAYPOINT:
                        self.routepoints.pop(i)
                        self.routepoints.append((waypoint, 1))
                else:
                    if distance <= DISTANCE_FROM_WAYPOINT_CHANGING_LANE:
                        self.routepoints.pop(i)
                        self.routepoints.append((waypoint, 1))
        # print("Completed: " + str(self.get_completage()))

    def get_all_waypoints(self):
        waypoints = []
        for routepoint in self.routepoints:
            waypoint = routepoint[0][0]
            waypoints.append(waypoint)
            for w in self.getLeftValidWaypointsRecursively(waypoint, [], 0):
                waypoints.append(w)
            for w in self.getRightValidWaypointsRecursively(waypoint, [], 0):
                waypoints.append(w)
        return waypoints

    # Get all lanes on the current road (to the left of the waypoint) that are valid drive for the vehicle driving this route.
    def getLeftValidWaypointsRecursively(self, waypoint, list, recursive_calls):
        if waypoint == None or recursive_calls > 8:
            return list
        lane_change = waypoint.lane_change
        if str(lane_change) == "Left" or str(lane_change) == "Both":
            left_lane_waypoint = waypoint.get_left_lane()
            if left_lane_waypoint == None:
                return list
            list.append(left_lane_waypoint)
            recursive_calls += 1
            return self.getLeftValidWaypointsRecursively(left_lane_waypoint, list, recursive_calls)
        else:
            return list

    # Get all road lanes on the map to the left of the current lane. Primarily used when marking the finish line.
    def getAllLeftWaypointsRecursively(self, waypoint, list, recursive_calls):
        left_lane_waypoint = waypoint.get_left_lane()
        if left_lane_waypoint is not None:
            list.append(left_lane_waypoint)
            recursive_calls += 1
            return self.getLeftValidWaypointsRecursively(left_lane_waypoint, list, recursive_calls)
        else:
            return list

    # Get all lanes on the current road (to the right of the waypoint) that are valid drive for the vehicle driving this route.
    def getRightValidWaypointsRecursively(self, waypoint, list, recursive_calls):
        if waypoint == None or recursive_calls > 8:
            return list
        lane_change = waypoint.lane_change
        if str(lane_change) == "Right" or str(lane_change) == "Both":
            right_lane_waypoint = waypoint.get_right_lane()
            if right_lane_waypoint == None:
                return list
            list.append(right_lane_waypoint)
            recursive_calls += 1
            return self.getRightValidWaypointsRecursively(right_lane_waypoint, list, recursive_calls)
        else:
            return list

    # Get all road lanes on the map to the right of the current lane. Primarily used when marking the finish line.
    def getAllRightWaypointsRecursively(self, waypoint, list, recursive_calls):
        right_lane_waypoint = waypoint.get_right_lane()
        if right_lane_waypoint is not None:
            list.append(right_lane_waypoint)
            recursive_calls += 1
            return self.getRightValidWaypointsRecursively(right_lane_waypoint, list, recursive_calls)
        else:
            return list
```

**File: ./software/carla_scripts/classes/helpers.py**

```
import numpy

# Get distance between start and finish locations
# start and finish are both of type carla.Location
def distanceBetweenTwoLocations(start, finish):
    first = numpy.array((start.x, start.y, start.z))
    second = numpy.array((finish.x, finish.y, finish.z))
    return numpy.linalg.norm(first-second)

# The following list is used to match the vehicle light state and figure out what lights are on
# The first element of the tuple is the "code" that the simualtion returns about the state of car's lights
# The second element is a tuple of five True/False values representing the states of the following
# (LeftBlinker, RightBlinker, LowBeam, HighBeam, FogLights)
pairs = [
        ("NONE",(False, False, False, False, False)),
        # Left
        ("LeftBlinker",(True, False, False, False, False)),
        ("40",(True, False, False, False, False)),
        ("96",(True, False, False, False, False)),
        ("104",(True, False, False, False, False)),
        # Left Position
        ("33",(True, False, False, False, False)),
        ("41",(True, False, False, False, False)),
        ("97",(True, False, False, False, False)),
        ("105",(True, False, False, False, False)),
        # Left Low
        ("99",(True, False, True, False, False)),
        ("107",(True, False, True, False, False)),
        ("35",(True, False, True, False, False)),
        ("43",(True, False, True, False, False)),
        # Left Fog
        ("227",(True, False, True, False, True)),
        ("235",(True, False, True, False, True)),
        ("163",(True, False, True, False, True)),
        ("171",(True, False, True, False, True)),
        # Right
        ("RightBlinker",(False, True, False, False, False)),
        ("24",(False, True, False, False, False)),
        ("80",(False, True, False, False, False)),
        ("88",(False, True, False, False, False)),
        # Right Position
        ("17",(False, True, False, False, False)),
        ("25",(False, True, False, False, False)),
        ("81",(False, True, False, False, False)),
        ("89",(False, True, False, False, False)),
        # Right Low
        ("19",(False, True, True, False, False)),
        ("27",(False, True, True, False, False)),
        ("83",(False, True, True, False, False)),
        ("91",(False, True, True, False, False)),
        # Right Fog
        ("211",(False, True, True, False, True)),
        ("219",(False, True, True, False, True)),
        ("147",(False, True, True, False, True)),
        ("155",(False, True, True, False, True)),
        # Low
        ("3",(False, False, True, False, False)),
        ("11",(False, False, True, False, False)),
        ("67",(False, False, True, False, False)),
        ("75",(False, False, True, False, False)),
        # Fog
        ("131",(False, False, True, False, True)),
        ("139",(False, False, True, False, True)),
        ("195",(False, False, True, False, True)),
        ("203",(False, False, True, False, True)),
        # High
        ("HighBeam",(False, False, False, True, False)),
        ("12",(False, False, False, True, False)),
        ("68",(False, False, False, True, False)),
        ("76",(False, False, False, True, False)),
        # Left High
        ("36",(True, False, False, True, False)),
        ("44",(True, False, False, True, False)),
        ("100",(True, False, False, True, False)),
        ("108",(True, False, False, True, False)),
        # Right High
        ("20",(False, True, False, True, False)),
        ("28",(False, True, False, True, False)),
        ("84",(False, True, False, True, False)),
        ("92",(False, True, False, True, False)),
        # Low High
        ("7",(False, False, True, True, False)),
        ("71",(False, False, True, True, False)),
        ("79",(False, False, True, True, False)),
        ("15",(False, False, True, True, False)),
        # Fog High
        ("135",(False, False, False, True, True)),
        ("143",(False, False, False, True, True)),
        ("199",(False, False, False, True, True)),
        ("207",(False, False, False, True, True)),
        # Left High Low
        ("39",(True, False, True, True, False)),
        ("47",(True, False, True, True, False)),
        ("103",(True, False, True, True, False)),
        ("111",(True, False, True, True, False)),
        # Left High Fog
        ("167",(True, False, True, True, True)),
        ("175",(True, False, True, True, True)),
        ("231",(True, False, True, True, True)),
        ("239",(True, False, True, True, True)),
        # Right High Low
        ("23",(False, True, True, True, False)),
        ("31",(False, True, True, True, False)),
        ("87",(False, True, True, True, False)),
        ("95",(False, True, True, True, False)),
        # Right High Fog
        ("151",(False, True, True, True, True)),
        ("159",(False, True, True, True, True)),
        ("215",(False, True, True, True, True)),
        ("223",(False, True, True, True, True))
    ]

# The default value to return in case the match among the pairs was not found (NO LIGHTS ARE ON)
default_match = (False, False, False, False, False)
```

**File: ./software/carla_scripts/recording/lane_monitor.py**

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import SOLID_PENALTY_SPEEDING, SOLID_TEXT, SOLID_PENALTY, DOUBLE_SOLID_PENALTY_SPEEDING, DOUBLE_SOLID_TEXT,DOUBLE_SOLID_PENALTY, BROKEN_NO_TURN_INDICA

import logging
logger = logging.getLogger(__name__)
class LaneMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, lane_type, timestep, player, is_speeding, turn_indicator_text):
        location = player.get_location()
        steering_angle = player.get_control().steer
        points_message = self.points_given_and_message(lane_type, is_speeding, steering_angle, turn_indicator_text)
        # Add only violations and lleave-out 0 penalty-point entries
        if(points_message and points_message[0] != 0):
            self.sum += points_message[0]
            self.buffer.append({
                'points': points_message[0],
                'message': points_message[1],
                'time': str('%.3f' % (timestep / 1000)) + "s",
                'location': {
                    'x': '%.6f' % location.x,
                    'y': '%.6f' % location.y,
                    'z': '%.6f' % location.z
                }
            })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("LaneMarkingViolationData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfLaneMarkingViolationInstances").text = str(length)

        entries = ET.SubElement(root, "LaneMarkingViolationInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}lane_marking_violation_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Lane monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Lane monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, lane_type, speeding, steering_angle, turn_indicator_text):
        if lane_type == "Solid":
            if speeding:
                return(SOLID_PENALTY_SPEEDING, SOLID_TEXT + " speeding")
            else:
                return(SOLID_PENALTY, SOLID_TEXT)
        elif lane_type == "SolidSolid":
            if speeding:
                return(DOUBLE_SOLID_PENALTY_SPEEDING, DOUBLE_SOLID_TEXT + " speeding")
            else:
                return(DOUBLE_SOLID_PENALTY, DOUBLE_SOLID_TEXT)
        elif lane_type == "Broken":
            if turn_indicator_text == "Left" and steering_angle <= 0:
                # Turning left legally
                pass
            elif turn_indicator_text == "Right" and steering_angle >= 0:
                # Turning right legally
                pass
            else:
                if speeding:
                    return(BROKEN_NO_TURN_INDICATOR_PENALTY_SPEEDING, BROKEN_NO_TURN_INDICATOR_TEXT + " speeding")
                else:
                    return(BROKEN_NO_TURN_INDICATOR_PENALTY, BROKEN_NO_TURN_INDICATOR_TEXT)
        else:
            return(0, 'None')
```

**File: ./software/carla_scripts/recording/speeding_monitor.py**

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import LIGHT_SPEEDING_PENALTY, HEAVY_SPEEDING_PENALTY, LIGHT_SPEEDING_TEXT, HEAVY_SPEEDING_TEXT, HEAVY_SPEEDING_THRESHOLD

import logging
logger = logging.getLogger(__name__)
class SpeedingMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, allowed_speed, player_speed, timestep, location):
        points_message = self.points_given_and_message(allowed_speed, player_speed)
        self.sum += points_message[0]
        self.buffer.append({
            'points': points_message[0],
            'message': points_message[1],
            'allowed_speed': allowed_speed,
            'player_speed': player_speed,
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("SpeedingData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfSpeedingInstances").text = str(length)

        entries = ET.SubElement(root, "SpeedingInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "AllowedSpeed").text = str(entry['allowed_speed'])
            ET.SubElement(entry_xml, "PlayerSpeed").text = str(entry['player_speed'])
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}speeding_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Speeding monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Speeding monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, allowed_speed, player_speed):
        absolute_value = player_speed - allowed_speed
        if absolute_value <= HEAVY_SPEEDING_THRESHOLD:
            return (LIGHT_SPEEDING_PENALTY, LIGHT_SPEEDING_TEXT)
        else:
            return (HEAVY_SPEEDING_PENALTY, HEAVY_SPEEDING_TEXT)
```

**File: ./software/carla_scripts/recording/recorder.py**

```python
import glob
import os
import sys

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

import argparse
import random
import time
import logging

import logging
logger = logging.getLogger(__name__)

class Recorder():

    def __init__(self, participant_name, scenario_name):
        self.client = carla.Client('127.0.0.1', 2000)
        self.participant_name = participant_name
        self.scenario_name = scenario_name

    def start_recording(self):
        try:
            path = '{}_{}_recording.log'.format(self.participant_name, self.scenario_name)
            self.client.start_recorder(path, True)
            logger.info("Successfully started recording the simultion to {}".format(path))
        except Exception as e:
            logger.error("An exception occurred trying to start recording", e)

    def stop_recording(self):
        try:
            self.client.stop_recorder()
            logger.info("Stopped the recorder. The recording log file can be found in CarlaUE4/Saved/")
        except Exception as e:
            logger.error("Could not stop the recorder", e)
```

**File: ./software/carla_scripts/recording/traffic_monitor.py**

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import RED_LIGHT_PENALTY, RED_LIGHT_PENALTY_SPEEDING, RED_LIGHT_TEXT

import logging
logger = logging.getLogger(__name__)
class TrafficMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, speeding, timestep, location, type):
        points_message = self.points_given_and_message(type, speeding)
        self.sum += points_message[0]
        self.buffer.append({
            'points': points_message[0],
            'message': points_message[1],
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("RoadTrafficViolationData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfRoadTrafficViolationInstances").text = str(length)

        entries = ET.SubElement(root, "RoadTrafficViolationInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}road_traffic_violation_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Traffic monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Traffic monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, type, is_speeding):
        if type == "redlight":
            if is_speeding:
                return(RED_LIGHT_PENALTY_SPEEDING, RED_LIGHT_TEXT + " speeding")
            else:
                return(RED_LIGHT_PENALTY, RED_LIGHT_TEXT)
        else:
            return(0, "Unknown")
```

**File: ./software/carla_scripts/recording/vehicle_light_monitor.py**

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from classes.helpers import pairs, default_match
from config import FPS, NO_BEAMS_NO_FOG_LIGHTS, NO_BEAMS, NO_FOG_LIGHTS, NO_BEAMS_NO_FOG_LIGHTS_TEXT, NO_BEAMS_TEXT, NO_FOG_LIGHTS_TEXT

import logging
logger = logging.getLogger(__name__)
class VehicleLightMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0
        self.just_wrote = True

        self.left_turn_indicator = False
        self.right_turn_indicator = False
        self.fog_lights = False
        self.low_beam = False
        self.high_beam = False

    def set_lights(self, code, weather, time, location):
        lights = self.match_lights(code)
        self.left_turn_indicator = lights[0]
        self.right_turn_indicator = lights[1]
        self.low_beam = lights[2]
        self.high_beam = lights[3]
        self.fog_lights = lights[4]
        # Write every 10s
        value = (time/1000) % 10
        if not self.just_wrote and (value > 0 and value <= 0.2):
            data = self.check_lights(weather)
            if(data):
                self.add_to_buffer(data[0], data[1], data[2], data[3], time, location)
            self.just_wrote = True
        elif value > 0.2:
            self.just_wrote = False
        else:
            pass

    def check_lights(self, weather):
        sun_altitude_angle = weather.sun_altitude_angle
        fog_density = weather.fog_density
        if(sun_altitude_angle < 30):
            if(fog_density > 50):
                if(not self.fog_lights):
                    if(not self.low_beam):
                        return (NO_BEAMS_NO_FOG_LIGHTS, NO_BEAMS_NO_FOG_LIGHTS_TEXT, sun_altitude_angle, fog_density)
                    else:
                        return (NO_FOG_LIGHTS, NO_FOG_LIGHTS_TEXT, sun_altitude_angle, fog_density)
                else:
                    if(not self.low_beam):
                        return (NO_BEAMS, NO_BEAMS_TEXT, sun_altitude_angle, fog_density)
            else:
                if(not self.low_beam):
                    return (NO_BEAMS, NO_BEAMS_TEXT, sun_altitude_angle, fog_density)
        else:
            if(fog_density > 50):
                if(not self.fog_lights):
                    if(not self.low_beam):
                        return (NO_BEAMS_NO_FOG_LIGHTS, "Foggy - no lights at all", sun_altitude_angle, fog_density)
                    else:
                        return (NO_FOG_LIGHTS, "Foggy - no fog lights", sun_altitude_angle, fog_density)
                else:
                    if(not self.low_beam):
                        return (NO_BEAMS, "Foggy - no low beams", sun_altitude_angle, fog_density)
            else:
                return None

    def turning_left(self):
        return self.left_turn_indicator

    def turning_right(self):
        return self.right_turn_indicator

    def add_to_buffer(self, points_given, type, sun_altitude_angle, fog_density, timestep, location):
        self.sum += points_given
        self.buffer.append({
            'points': points_given,
            'message': type,
            'sun_altitude_angle': sun_altitude_angle,
            'fog_density': fog_density,
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("VehicleLightMisuseData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfVehicleLightMisuseInstances").text = str(length)

        entries = ET.SubElement(root, "VehicleLightMisuseInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "SunAltitudeAngle").text = str(entry['sun_altitude_angle'])
            ET.SubElement(entry_xml, "FogDensity").text = str(entry['fog_density'])
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}vehicle_light_misuse_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Vehicle light misuse monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Vehicle light misuse monitor did not save data to the file {}".format(path), e)


    def match_lights(self, code):
        for pair in pairs:
            if pair[0] == str(code):
                return pair[1]
        return default_match

```

**File: ./software/carla_scripts/recording/route_monitor.py**

```python
import carla
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from classes.route import Route
from agents.navigation.global_route_planner import GlobalRoutePlanner
from config import SAMPLING_RESOLUTION
import json

import logging
logger = logging.getLogger(__name__)

class RouteMonitor():
    def __init__(self, path_locations, map, save_file):
        self.route = self.get_route(path_locations, map)
        self.save_file = save_file

    # Takes a location as an argument and checks if it belongs to the route
    def is_on_path(self, location):
        pass

    def update(self, location, is_changing_lane):
        self.route.advanced_recalculate(location, is_changing_lane)

    def finish_reached(self, player_location):
        if self.route.is_finished(player_location):
            return True

    def write_to_file(self, finish_reached, simulation_time):
        root = ET.Element("RouteData")
        ET.SubElement(root, "SimulationTime").text = str('%.3f' % (simulation_time / 1000))+"s"
        ET.SubElement(root, "ProportionOfRouteCompleted").text = str(self.route.get_route_completion())
        ET.SubElement(root, "FinishReached").text = str(finish_reached)
        ET.SubElement(root, "NumberOfWaypoints").text = str(len(self.route.routepoints))
        waypoints = ET.SubElement(root, "Waypoints")
        for routepoint in self.route.routepoints:
            location = routepoint[0][0].transform.location
            entry_xml = ET.SubElement(waypoints, "Instance")
            ET.SubElement(entry_xml, "WasReached").text = str(routepoint[1])
            location_elem = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location_elem, "X").text = str('%.6f' % location.x)
            ET.SubElement(location_elem, "Y").text = str('%.6f' % location.y)
            ET.SubElement(location_elem, "Z").text = str('%.6f' % location.z)

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
        try:
            path = "{}route_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Route monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Route monitor did not save data to the file {}".format(path), e)


    def get_route(self, path_locations, map):
        waypoints = []
        grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
        locations = self.transform_to_carla_locations(path_locations)
        for i in range(len(locations)-1):
            waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
        return Route(waypoints)

    def transform_to_carla_locations(self, path_locations):
        locations = []
        for loc in path_locations:
            locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
        return locations

    # WORKS WHERE THERE ARE ONLY TWO LOCATIONS PROVIDED
    #
    #
    # def get_route(self, pathName, map):
    #     grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
    #     locations = self.read_path_file(pathName)
    #     # Define a route
    #     waypoints = grp.trace_route(locations[0], locations[1])
    #     return Route(waypoints)

    # def read_path_file(self, pathName):
    #     start = None
    #     finish = None
    #     with open("./paths/{}.txt".format(pathName), "r") as file:
    #         i = 0
    #         for line in file.readlines():
    #             values = [float(j) for j in line.split(',') if j.strip()]
    #             if i == 0:
    #                 start = carla.Location(x=values[0], y=values[1], z=values[2])
    #             else:
    #                 finish = carla.Location(x=values[0], y=values[1], z=values[2])
    #             i+=1
    #     return (start, finish)
```

**File: ./software/carla_scripts/recording/collision_monitor.py**

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import HIT_PEDESTRIAN_PENALTY_SPEEDING, HIT_PEDESTRIAN_TEXT, HIT_PEDESTRIAN_PENALTY, HIT_VEHICLE_PENALTY_SPEEDING, HIT_VEHICLE_TEXT, HIT_VEHICLE_PENAL
from config import HIT_BICYCLE_TEXT, HIT_BICYCLE_PENALTY, HIT_ROAD_OBJECT_PENALTY_SPEEDING, HIT_ROAD_OBJECT_TEXT, HIT_ROAD_OBJECT_PENALTY

import logging
logger = logging.getLogger(__name__)
class CollisionMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, intensity, other_actor, location, timestep, is_speeding):
        points_message = self.points_given_and_message(other_actor.type_id, intensity, is_speeding)
        self.sum += points_message[0]
        self.buffer.append({
            'points': points_message[0],
            'message': points_message[1],
            'intensity': intensity,
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("CollisionData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfCollisionInstances").text = str(length)

        speeding_entries = ET.SubElement(root, "CollisionInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(speeding_entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = str(entry['message'])
            ET.SubElement(entry_xml, "CollisionIntensity").text = str(entry['intensity'])
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
        try:
            path = "{}collision_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Collision monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Collision monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, victim_type, intensity, speeding):
        # Intensity to be used
        if victim_type.startswith("walker"):
            if speeding:
                return(HIT_PEDESTRIAN_PENALTY_SPEEDING, HIT_PEDESTRIAN_TEXT + " speeding")
            else:
                return(HIT_PEDESTRIAN_PENALTY, HIT_PEDESTRIAN_TEXT)
        elif victim_type.startswith("vehicle"):
            if speeding:
                return(HIT_VEHICLE_PENALTY_SPEEDING, HIT_VEHICLE_TEXT + " speeding")
            else:
                return(HIT_VEHICLE_PENALTY, HIT_VEHICLE_TEXT)
        elif victim_type.startswith("bicycle"):
            if speeding:
                return(HIT_BICYCLE_PENALTY_SPEEDING, HIT_BICYCLE_TEXT + " speeding")
            else:
                return(HIT_BICYCLE_PENALTY, HIT_BICYCLE_TEXT)
        else:
            if speeding:
                return(HIT_ROAD_OBJECT_PENALTY_SPEEDING, HIT_ROAD_OBJECT_TEXT + " speeding")
            else:
                return(HIT_ROAD_OBJECT_PENALTY, HIT_ROAD_OBJECT_TEXT)

    ### UNUSED

    def get_actor_display_name(self, actor, truncate=250):
        name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
        return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name
```

**File: ./software/carla_scripts/recording/manager.py**

```python
import carla
import weakref
import math
import time
import sys
from recording.route_monitor import RouteMonitor
from recording.lane_monitor import LaneMonitor
from recording.collision_monitor import CollisionMonitor
from recording.traffic_monitor import TrafficMonitor
from recording.speeding_monitor import SpeedingMonitor
from recording.vehicle_light_monitor import VehicleLightMonitor
from recording.recorder import Recorder

import logging
logger = logging.getLogger(__name__)
class Manager():

    def __init__(self, path_locations, world, player, participant_name, scenario_name):
        self.path_locations = path_locations
        self.world = world
        self.player = player
        self.participant_name = participant_name
        self.scenario_name = scenario_name
        self.map = self.world.get_map()
        self.time = 0

        self.speeding = False
        self.junction_visited_in_last_timestep = False

        self.collision_sensor = CollisionSensor(self.player, self.world, self)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.world, self)
        save_file = '../../data/recordings/{}/{}/'.format(participant_name, scenario_name)
        self.route_monitor = RouteMonitor(self.path_locations, self.map, save_file)
        self.lane_monitor = LaneMonitor(save_file)
        self.collision_monitor = CollisionMonitor(save_file)
        self.speeding_monitor = SpeedingMonitor(save_file)
        self.traffic_monitor = TrafficMonitor(save_file)
        self.vehicle_light_monitor = VehicleLightMonitor(save_file)
```

```
39
40          self.recorder = Recorder(participant_name, scenario_name)
41          self.recorder.start_recording()
42
43          self.time_of_last_collision = None
44          self.actor_of_last_collision = None
45          logger.info("Manager was successfully initialised")
46
47      # Called at each timestep. Checks for violations and records to respective monitors
48      def record(self, simulation_time):
49          # logger.info("Manager recording at {}".format(str('%.3f' % (simulation_time / 1000))+"s"))
50          try:
51              player_location = self.player.get_location()
52              self.time = simulation_time
53              # Check if turn indicators are on
54              weather = self.world.get_weather()
55              self.set_lights(weather, self.time, player_location)
56              # Check if turn indicators are on
57              is_changing_lane = False
58              if self.vehicle_light_monitor.turning_left() or self.vehicle_light_monitor.turning_right():
59                  is_changing_lane = True
60
61              self.route_monitor.update(player_location, is_changing_lane)
62              speeding = self.check_speeding()
63              violating_traffic = self.check_traffic_violations()
64              if speeding:
65                  self.speeding_monitor.add_to_buffer(speeding[1], speeding[0], self.time, player_location)
66              if violating_traffic:
67                  self.traffic_monitor.add_to_buffer(self.speeding, self.time, player_location, "redlight")
68              # logger.info("Observations were added to the buffers successfully.")
69              if self.route_monitor.finish_reached(player_location):
70                  logger.info("Finish was reached. Terminating the simulation.")
71                  self.shut_down(True)
72                  return True
73          except Exception as e:
74              logger.error("Something went wrong adding observations to monitor buffers.", e)
75
76      # Called when finish is reached or simulation is quit manually
77      def shut_down(self, finish_reached):
78          logger.info("Shutting the manager down. Saving data from monitors to files.")
79          try:
80              self.route_monitor.write_to_file(finish_reached, self.time)
81              self.lane_monitor.write_to_file()
82              self.collision_monitor.write_to_file()
83              self.traffic_monitor.write_to_file()
84              self.speeding_monitor.write_to_file()
85              self.vehicle_light_monitor.write_to_file()
86              self.recorder.stop_recording()
87          except Exception as e:
88              logger.error("Something went wrong saving data from monitors to files.", e)
89          try:
90              logger.info("Destroying sensors.")
91              self.delete_sensors()
92          except Exception as e:
93              logger.error("Something went wrong destroying sensors owned by the Manager.", e)
94      # After the simulation is over, delete sensors
95      def delete_sensors(self):
96          self.collision_sensor.sensor.stop()
97          self.collision_sensor.sensor.destroy()
98          self.lane_invasion_sensor.sensor.stop()
99          self.lane_invasion_sensor.sensor.destroy()
100
101     # Record the collision to the collision monitor buffer
102     def record_collision(self, intensity, other_actor, player_location):
103         # If there was a collision witht the same actor in the past 2sec, disregards this (collision sensor sometimes counts a collision multiple times)
104         if(self.actor_of_last_collision and self.actor_of_last_collision == other_actor.id and (self.time - self.time_of_last_collision)/1000 < 2):
105             pass
106         else:
107             self.time_of_last_collision = self.time
108             self.actor_of_last_collision = other_actor.id
109             self.collision_monitor.add_to_buffer(intensity, other_actor, player_location, self.time, self.speeding)
110
111     # Record the lane marking violations to the lane monitor buffer
112     def record_lane_invasion(self, text):
113         if self.vehicle_light_monitor.turning_left():
114             light_indicator_text = "Left"
115         elif self.vehicle_light_monitor.turning_right():
116             light_indicator_text = "Right"
117         else:
118             light_indicator_text = "None"
119         self.lane_monitor.add_to_buffer(text, self.time, self.player, self.speeding, light_indicator_text)
120
121     # Check if the vehicle speeding
122     def check_speeding(self):
123         current_speed = self.player.get_velocity().x
124         # Convert from m/ to km/h and round to 2 places after comma
125         current_speed = float('%.2f' % (abs(current_speed)*3.6))
126         speed_limit = self.player.get_speed_limit()
127         if current_speed and speed_limit and current_speed > speed_limit:
128             self.speeding = True
129             return (current_speed, speed_limit)
130         else:
131             self.speeding = False
132             return None
133
134     # Check if any traffic violations occurred (for now checks only for red lights)
135     def check_traffic_violations(self):
136         waypoint = self.map.get_waypoint(self.player.get_location())
137         if waypoint.is_junction:
138             if not self.junction_visited_in_last_timestep and str(self.player.get_traffic_light_state()) == "Red":
139                 self.junction_visited_in_last_timestep = True
140                 return True
141             else:
142                 return False
143         else:
144             self.junction_visited_in_last_timestep = False
145             return False
146
147     # Update vehicle light monitor state (update which ligths are on)
148     def set_lights(self, weather, time, player_location):
149         light_state = self.player.get_light_state()
150         self.vehicle_light_monitor.set_lights(light_state, weather, time, player_location)
151
152
153 # The foundations of this sensor were borrowed from CARLA developers
154 class LaneInvasionSensor(object):
155     def __init__(self, parent_actor, world, manager):
156         self.sensor = None
157         self.manager = manager
158         bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
159         self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=parent_actor)
160         # We need to pass the lambda a weak reference to self to avoid circular
161         # reference.
162         weak_self = weakref.ref(self)
163         self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))
164
165     @staticmethod
166     def _on_invasion(weak_self, event):
167         self = weak_self()
168         if not self:
```

```
            return
        lane_types = set(x.type for x in event.crossed_lane_markings)
        text = ['%r' % str(x).split()[-1] for x in lane_types]
        self.manager.record_lane_invasion(text[0].replace("'", ""))


# The foundations of this sensor were borrowed from CARLA developers
class CollisionSensor(object):
    def __init__(self, parent_actor, world, manager):
        self.sensor = None
        self.player = parent_actor
        self.manager = manager
        bp = world.get_blueprint_library().find('sensor.other.collision')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self.player)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))

    @staticmethod
    def _on_collision(weak_self, event):
        self = weak_self()
        if not self:
            return
        impulse = event.normal_impulse
        intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
        self.manager.record_collision(intensity, event.other_actor, self.player.get_location())

```

**File: ./software/carla_scripts/run.sh**

```
#!/bin/bash

name=$1
ai=$2

if [ -z "$name" ]; then
    echo "If you want to run simulations, you need to provide the name of the participant"
    echo "Example use: ./run.sh participant1"
    echo "The name is needed to associate each participant with their simualtion results"
    exit 1
fi

# First create a directory for the participant
cd ../../data/recordings/
mkdir -p $name

# Create a log file for the session inside
touch $name/session_logs.log
chmod 666 $name/session_logs.log

cd ../../software/carla_scripts

if [ -n "$ai" ]; then
    if [ "$ai" = "ai" ]; then
        echo "Starting the simulations for AI agent. For the logs, check the $name/session_logs.log file in the recordings directory."
        python3 run.py -n "$name" --ai > /dev/null 2>&1
    else
        echo "If you want the AI implementation to run the scenarios write ai as the second argument of this script"
        exit 1
    fi
else
    echo "Starting the simulations in manual control. For the logs, check the $name/session_logs.log file in the recordings directory."
    python3 run.py -n "$name" > /dev/null 2>&1
fi
echo "Finished."
```

**File: ./software/carla_scripts/README.md**

```
## About the simulations

To run the simulations, your machine must meet the software requirements stated in the [README.md](../../README.md) file in the software directory.

Launch the UE4 engine running CARLA server following the guide in CARLA's [documentation](https://carla.readthedocs.io/en/latest/).

```
bash run.sh [participant's name e.g. wilson]
```
This will run n scenarios sequentially, as stated in the [scenario_list.json](./scenario_list.json) file.

To let the CARLA Agent execute the scenarios, please run the command stated above but add 'ai' at the end.

Example:

bash run.sh [participant's name e.g. carla_agent] ai
```

## Recordings

All simulations are recorded and monitored. The recorded data will be stored in:
`data/recordings/[participant's name]/`

The recording of the simulation itself will be stored in:

 `CarlaUE4/Saved/`

```
bash replay.sh [participant's name] [scenario name]
```

This will replay the simulation of scenario driven by the participant.
```

**File: ./software/carla_scripts/self_driver.py**

```
import os
import glob
import sys
import numpy as np

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass
import carla

import math
import random
```
```

```python
import time
import base64
import pygame
import pickle
from agents.navigation.global_route_planner import GlobalRoutePlanner
from config import SAMPLING_RESOLUTION_AI
from carla import ColorConverter as cc
from agents.navigation.behavior_agent import BehaviorAgent

import argparse
import collections
import datetime
import logging
import math
import random
import re
import weakref
from recording.manager import Manager
from config import FPS
from pygame import K_TAB
from pygame import K_ESCAPE
import argparse
import logging
from config import FPS


# ==============================================================================
# -- Global functions ----------------------------------------------------------
# ==============================================================================


def find_weather_presets():
    rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')
    name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))
    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]


def get_actor_display_name(actor, truncate=250):
    name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
    return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name

def get_actor_blueprints(world, filter, generation):
    bps = world.get_blueprint_library().filter(filter)
    if generation.lower() == "all":
        return bps

    # If the filter returns only one bp, we assume that this one needed
    # and therefore, we ignore the generation
    if len(bps) == 1:
        return bps

    try:
        int_generation = int(generation)
        # Check if generation is in available generations
        if int_generation in [1, 2]:
            bps = [x for x in bps if int(x.get_attribute('generation')) == int_generation]
            return bps
        else:
            print("   Warning! Actor Generation is not valid. No actor will be spawned.")
            return []
    except:
        print("   Warning! Actor Generation is not valid. No actor will be spawned.")
        return []


# ==============================

class World(object):
    def __init__(self, carla_world, hud, args, scenario, client, path_details):
        self.world = carla_world
        self.sync = args.sync
        self.actor_role_name = args.rolename
        self.path_details = path_details
        self.scenario = scenario
        self.client = client
        self.scenario_name = args.scenario
        self.participant_name = args.name
        try:
            self.map = self.world.get_map()
        except RuntimeError as error:
            print('RuntimeError: {}'.format(error))
            print('  The server could not send the OpenDRIVE (.xodr) file:')
            print('  Make sure it exists, has the same name of your town, and is correct.')
            sys.exit(1)
        self.spawn_point = None
        self.init_spawn_point(path_details["start_location"])
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.imu_sensor = None
        self.radar_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = args.filter
        self._actor_generation = args.generation
        self._gamma = args.gamma
        self.restart(args)
        self.recording_enabled = 0

        # All about the simulation
        self.simulation_clock = None
        self.simulation_start_tick = 0
        self.simulation_time = 0
        self.manager = None

        self.recording_start = 0
        self.constant_velocity_enabled = False
        self.show_vehicle_telemetry = False
        self.doors_are_open = False
        self.current_map_layer = 0
        self.map_layer_names = [
            carla.MapLayer.NONE,
            carla.MapLayer.Buildings,
            carla.MapLayer.Decals,
            carla.MapLayer.Foliage,
            carla.MapLayer.Ground,
            carla.MapLayer.ParkedVehicles,
            carla.MapLayer.Particles,
            carla.MapLayer.Props,
            carla.MapLayer.StreetLights,
            carla.MapLayer.Walls,
            carla.MapLayer.All
        ]

    def init_spawn_point(self, start_location):
        try:
```

```
            self.spawn_point = carla.Transform(carla.Location(x=start_location["x"], y=start_location["y"], z=start_location["z"]), carla.Rotation(pitch=0.0, yaw=
        except Exception as e:
            logger.error("Could not initiate spawn location for the driver", exc_info=True)
            raise e
    def restart(self, args):
        self.player_max_speed = 1.589
        self.player_max_speed_fast = 3.713
        # Keep same camera config if the camera manager exists.
        cam_index = self.camera_manager.index if self.camera_manager is not None else 0
        cam_pos_index = self.camera_manager.transform_index if self.camera_manager is not None else 0
        # Get a random blueprint.
        blueprint = random.choice(get_actor_blueprints(self.world, self._actor_filter, self._actor_generation))
        blueprint.set_attribute('role_name', self.actor_role_name)
        if blueprint.has_attribute('color'):
            blueprint.set_attribute('color', '{},{},{}'.format(args.red, args.green, args.blue))
        if blueprint.has_attribute('driver_id'):
            driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
            blueprint.set_attribute('driver_id', driver_id)
        if blueprint.has_attribute('is_invincible'):
            blueprint.set_attribute('is_invincible', 'true')
        # set the max speed
        if blueprint.has_attribute('speed'):
            self.player_max_speed = float(blueprint.get_attribute('speed').recommended_values[1])
            self.player_max_speed_fast = float(blueprint.get_attribute('speed').recommended_values[2])

        # Spawn the player.
        if self.player is not None:
            spawn_point = self.player.get_transform()
            spawn_point.location.z += 2.0
            spawn_point.rotation.roll = 0.0
            spawn_point.rotation.pitch = 0.0
            self.destroy()
            self.player = self.world.try_spawn_actor(blueprint, spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        while self.player is None:
            if not self.map.get_spawn_points():
                print('There are no spawn points available in your map/town.')
                print('Please add some Vehicle Spawn Point to your UE4 scene.')
                sys.exit(1)
            if self.spawn_point is None:
                spawn_points = self.map.get_spawn_points()
                self.spawn_point = random.choice(spawn_points) if spawn_points else carla.Transform()
            self.player = self.world.try_spawn_actor(blueprint, self.spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        # Set up the sensors.
        self.collision_sensor = CollisionSensor(self.player, self.hud)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.hud)
        self.gnss_sensor = GnssSensor(self.player)
        self.imu_sensor = IMUSensor(self.player)
        self.camera_manager = CameraManager(self.player, self.hud, self._gamma)
        self.camera_manager.transform_index = cam_pos_index
        self.camera_manager.set_sensor(cam_index, notify=False)
        actor_type = get_actor_display_name(self.player)
        self.hud.notification(actor_type)

        if self.sync:
            self.world.tick()
        else:
            self.world.wait_for_tick()

    def toggle_radar(self):
        if self.radar_sensor is None:
            self.radar_sensor = RadarSensor(self.player)
        elif self.radar_sensor.sensor is not None:
            self.radar_sensor.sensor.destroy()
            self.radar_sensor = None

    def modify_vehicle_physics(self, actor):
        #If actor is not a vehicle, we cannot use the physics control
        try:
            physics_control = actor.get_physics_control()
            physics_control.use_sweep_wheel_collision = True
            actor.apply_physics_control(physics_control)
        except Exception:
            pass

    def startScenario(self):
        if self.simulation_clock is None:
            logger.info("Starting the scenario")
            self.scenario.start()
            self.hud.notification('The simulation has started')
            self.manager = Manager(self.path_details["path_checkpoints"], self.world, self.player, self.participant_name, self.scenario_name)
            self.simulation_clock = pygame.time.Clock()
            self.simulation_start_tick = pygame.time.get_ticks()

    def finishScenario(self):
        logger.warning("Finish was NOT REACHED. Terminating the simulation.")
        self.manager.shut_down(False)
        self.scenario.finish(self.client)

    def tick(self):
        if self.simulation_clock is not None:
            self.simulation_clock.tick(60)
            self.simulation_time += self.simulation_clock.get_time()
            if self.manager.record(self.simulation_time):
                return True
            self.hud.tick(self, self.simulation_time/1000)
        else:
            self.hud.tick(self, 0)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy_sensors(self):
        self.camera_manager.sensor.destroy()
        self.camera_manager.sensor = None
        self.camera_manager.index = None

    def destroy(self):
        if self.radar_sensor is not None:
            self.toggle_radar()
        sensors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor,
            self.imu_sensor.sensor]
        for sensor in sensors:
            if sensor is not None:
                sensor.stop()
                sensor.destroy()
        if self.player is not None:
            self.player.destroy()

# ==============================================================================
# -- HUD -----------------------------------------------------------------------
# ==============================================================================
```

```
277
278
279class HUD(object):
280    def __init__(self, width, height):
281        self.dim = (width, height)
282        font = pygame.font.Font(pygame.font.get_default_font(), 20)
283        font_name = 'courier' if os.name == 'nt' else 'mono'
284        fonts = [x for x in pygame.font.get_fonts() if font_name in x]
285        default_font = 'ubuntumono'
286        mono = default_font if default_font in fonts else fonts[0]
287        mono = pygame.font.match_font(mono)
288        self._font_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)
289        self._notifications = FadingText(font, (width, 40), (0, height - 40))
290        self.server_fps = 0
291        self.frame = 0
292        self.simulation_time = 0
293        self._show_info = True
294        self._info_text = []
295        self._server_clock = pygame.time.Clock()
296
297    def on_world_tick(self, timestamp):
298        self._server_clock.tick()
299        self.server_fps = self._server_clock.get_fps()
300        self.frame = timestamp.frame
301        self.simulation_time = timestamp.elapsed_seconds
302
303    def tick(self, world, time):
304        self._notifications.tick(world, time)
305        if not self._show_info:
306            return
307        t = world.player.get_transform()
308        v = world.player.get_velocity()
309        c = world.player.get_control()
310        compass = world.imu_sensor.compass
311        heading = 'N' if compass > 270.5 or compass < 89.5 else ''
312        heading += 'S' if 90.5 < compass < 269.5 else ''
313        heading += 'E' if 0.5 < compass < 179.5 else ''
314        heading += 'W' if 180.5 < compass < 359.5 else ''
315        colhist = world.collision_sensor.get_collision_history()
316        collision = [colhist[x + self.frame - 200] for x in range(0, 200)]
317        max_col = max(1.0, max(collision))
318        collision = [x / max_col for x in collision]
319        vehicles = world.world.get_actors().filter('vehicle.*')
320        self._info_text = [
321            'Vehicle: % 20s' % get_actor_display_name(world.player, truncate=20),
322            'Route time: % 12s' % datetime.timedelta(seconds=int(time)),
323            '',
324            'Speed:   % 15.0f km/h' % (3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)),
325            '']
326        if isinstance(c, carla.VehicleControl):
327            self._info_text += [
328                ('Throttle:', c.throttle, 0.0, 1.0),
329                ('Steer:', c.steer, -1.0, 1.0),
330                ('Brake:', c.brake, 0.0, 1.0),
331                ('Reverse:', c.reverse),
332                ('Hand brake:', c.hand_brake),
333                ('Manual:', c.manual_gear_shift),
334                'Gear:        %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)]
335        self._info_text += [
336            '',
337            'Collision:',
338            collision,
339            '']
340
341    def toggle_info(self):
342        self._show_info = not self._show_info
343
344    def notification(self, text, seconds=2.0):
345        self._notifications.set_text(text, seconds=seconds)
346
347    def error(self, text):
348        self._notifications.set_text('Error: %s' % text, (255, 0, 0))
349
350    def render(self, display):
351        if self._show_info:
352            info_surface = pygame.Surface((220, self.dim[1]))
353            info_surface.set_alpha(100)
354            display.blit(info_surface, (0, 0))
355            v_offset = 4
356            bar_h_offset = 100
357            bar_width = 106
358            for item in self._info_text:
359                if v_offset + 18 > self.dim[1]:
360                    break
361                if isinstance(item, list):
362                    if len(item) > 1:
363                        points = [(x + 8, v_offset + 8 + (1.0 - y) * 30) for x, y in enumerate(item)]
364                        pygame.draw.lines(display, (255, 136, 0), False, points, 2)
365                    item = None
366                    v_offset += 18
367                elif isinstance(item, tuple):
368                    if isinstance(item[1], bool):
369                        rect = pygame.Rect((bar_h_offset, v_offset + 8), (6, 6))
370                        pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1] else 1)
371                    else:
372                        rect_border = pygame.Rect((bar_h_offset, v_offset + 8), (bar_width, 6))
373                        pygame.draw.rect(display, (255, 255, 255), rect_border, 1)
374                        f = (item[1] - item[2]) / (item[3] - item[2])
375                        if item[2] < 0.0:
376                            rect = pygame.Rect((bar_h_offset + f * (bar_width - 6), v_offset + 8), (6, 6))
377                        else:
378                            rect = pygame.Rect((bar_h_offset, v_offset + 8), (f * bar_width, 6))
379                        pygame.draw.rect(display, (255, 255, 255), rect)
380                    item = item[0]
381                if item:  # At this point has to be a str.
382                    surface = self._font_mono.render(item, True, (255, 255, 255))
383                    display.blit(surface, (8, v_offset))
384                v_offset += 18
385        self._notifications.render(display)
386
387
388# ==============================================================================
389# -- FadingText ----------------------------------------------------------------
390# ==============================================================================
391
392
393class FadingText(object):
394    def __init__(self, font, dim, pos):
395        self.font = font
396        self.dim = dim
397        self.pos = pos
398        self.seconds_left = 0
399        self.surface = pygame.Surface(self.dim)
400
401    def set_text(self, text, color=(255, 255, 255), seconds=2.0):
402        text_texture = self.font.render(text, True, color)
403        self.surface = pygame.Surface(self.dim)
404        self.seconds_left = seconds
405        self.surface.fill((0, 0, 0, 0))
406        self.surface.blit(text_texture, (10, 11))
```

```
407
408     def tick(self, _, time):
409         delta_seconds = 1e-3 * time
410         self.seconds_left = max(0.0, self.seconds_left - delta_seconds)
411         self.surface.set_alpha(500.0 * self.seconds_left)
412
413     def render(self, display):
414         display.blit(self.surface, self.pos)
415
416
417 # ==============================================================================
418 # -- CollisionSensor -----------------------------------------------------------
419 # ==============================================================================
420
421
422 class CollisionSensor(object):
423     def __init__(self, parent_actor, hud):
424         self.sensor = None
425         self.history = []
426         self._parent = parent_actor
427         self.hud = hud
428         world = self._parent.get_world()
429         bp = world.get_blueprint_library().find('sensor.other.collision')
430         self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
431         # We need to pass the lambda a weak reference to self to avoid circular
432         # reference.
433         weak_self = weakref.ref(self)
434         self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))
435
436     def get_collision_history(self):
437         history = collections.defaultdict(int)
438         for frame, intensity in self.history:
439             history[frame] += intensity
440         return history
441
442     @staticmethod
443     def _on_collision(weak_self, event):
444         self = weak_self()
445         if not self:
446             return
447         actor_type = get_actor_display_name(event.other_actor)
448         self.hud.notification('Collision with %r' % actor_type)
449         impulse = event.normal_impulse
450         intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
451         self.history.append((event.frame, intensity))
452         if len(self.history) > 4000:
453             self.history.pop(0)
454
455
456 # ==============================================================================
457 # -- LaneInvasionSensor --------------------------------------------------------
458 # ==============================================================================
459
460
461 class LaneInvasionSensor(object):
462     def __init__(self, parent_actor, hud):
463         self.sensor = None
464
465         # If the spawn object is not a vehicle, we cannot use the Lane Invasion Sensor
466         if parent_actor.type_id.startswith("vehicle."):
467             self._parent = parent_actor
468             self.hud = hud
469             world = self._parent.get_world()
470             bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
471             self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
472             # We need to pass the lambda a weak reference to self to avoid circular
473             # reference.
474             weak_self = weakref.ref(self)
475             self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))
476
477     @staticmethod
478     def _on_invasion(weak_self, event):
479         self = weak_self()
480         if not self:
481             return
482         lane_types = set(x.type for x in event.crossed_lane_markings)
483         text = ['%r' % str(x).split()[-1] for x in lane_types]
484         self.hud.notification('Crossed line %s' % ' and '.join(text))
485
486
487 # ==============================================================================
488 # -- GnssSensor --------------------------------------------------------
489 # ==============================================================================
490
491
492 class GnssSensor(object):
493     def __init__(self, parent_actor):
494         self.sensor = None
495         self._parent = parent_actor
496         self.lat = 0.0
497         self.lon = 0.0
498         world = self._parent.get_world()
499         bp = world.get_blueprint_library().find('sensor.other.gnss')
500         self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0, z=2.8)), attach_to=self._parent)
501         # We need to pass the lambda a weak reference to self to avoid circular
502         # reference.
503         weak_self = weakref.ref(self)
504         self.sensor.listen(lambda event: GnssSensor._on_gnss_event(weak_self, event))
505
506     @staticmethod
507     def _on_gnss_event(weak_self, event):
508         self = weak_self()
509         if not self:
510             return
511         self.lat = event.latitude
512         self.lon = event.longitude
513
514
515 # ==============================================================================
516 # -- IMUSensor -----------------------------------------------------------
517 # ==============================================================================
518
519
520 class IMUSensor(object):
521     def __init__(self, parent_actor):
522         self.sensor = None
523         self._parent = parent_actor
524         self.accelerometer = (0.0, 0.0, 0.0)
525         self.gyroscope = (0.0, 0.0, 0.0)
526         self.compass = 0.0
527         world = self._parent.get_world()
528         bp = world.get_blueprint_library().find('sensor.other.imu')
529         self.sensor = world.spawn_actor(
530             bp, carla.Transform(), attach_to=self._parent)
531         # We need to pass the lambda a weak reference to self to avoid circular
532         # reference.
533         weak_self = weakref.ref(self)
534         self.sensor.listen(
535             lambda sensor_data: IMUSensor._IMU_callback(weak_self, sensor_data))
536
```

```
    @staticmethod
    def _IMU_callback(weak_self, sensor_data):
        self = weak_self()
        if not self:
            return
        limits = (-99.9, 99.9)
        self.accelerometer = (
            max(limits[0], min(limits[1], sensor_data.accelerometer.x)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.y)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.z)))
        self.gyroscope = (
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.x))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.y))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.z))))
        self.compass = math.degrees(sensor_data.compass)


# ==============================================================================
# -- RadarSensor ---------------------------------------------------------------
# ==============================================================================


class RadarSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        bound_x = 0.5 + self._parent.bounding_box.extent.x
        bound_y = 0.5 + self._parent.bounding_box.extent.y
        bound_z = 0.5 + self._parent.bounding_box.extent.z

        self.velocity_range = 7.5 # m/s
        world = self._parent.get_world()
        self.debug = world.debug
        bp = world.get_blueprint_library().find('sensor.other.radar')
        bp.set_attribute('horizontal_fov', str(35))
        bp.set_attribute('vertical_fov', str(20))
        self.sensor = world.spawn_actor(
            bp,
            carla.Transform(
                carla.Location(x=bound_x + 0.05, z=bound_z+0.05),
                carla.Rotation(pitch=5)),
            attach_to=self._parent)
        # We need a weak reference to self to avoid circular reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(
            lambda radar_data: RadarSensor._Radar_callback(weak_self, radar_data))

    @staticmethod
    def _Radar_callback(weak_self, radar_data):
        self = weak_self()
        if not self:
            return
        # To get a numpy [[vel, altitude, azimuth, depth],...[,,,]]:
        # points = np.frombuffer(radar_data.raw_data, dtype=np.dtype('f4'))
        # points = np.reshape(points, (len(radar_data), 4))

        current_rot = radar_data.transform.rotation
        for detect in radar_data:
            azi = math.degrees(detect.azimuth)
            alt = math.degrees(detect.altitude)
            # The 0.25 adjusts a bit the distance so the dots can
            # be properly seen
            fw_vec = carla.Vector3D(x=detect.depth - 0.25)
            carla.Transform(
                carla.Location(),
                carla.Rotation(
                    pitch=current_rot.pitch + alt,
                    yaw=current_rot.yaw + azi,
                    roll=current_rot.roll)).transform(fw_vec)

            def clamp(min_v, max_v, value):
                return max(min_v, min(value, max_v))

            norm_velocity = detect.velocity / self.velocity_range # range [-1, 1]
            r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
            g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
            b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
            self.debug.draw_point(
                radar_data.transform.location + fw_vec,
                size=0.075,
                life_time=0.06,
                persistent_lines=False,
                color=carla.Color(r, g, b))

# ==============================================================================
# -- CameraManager -------------------------------------------------------------
# ==============================================================================


class CameraManager(object):
    def __init__(self, parent_actor, hud, gamma_correction):
        self.sensor = None
        self.surface = None
        self._parent = parent_actor
        self.hud = hud
        self.recording = False
        Attachment = carla.AttachmentType

        self._camera_transforms = [
            # Third-person
            (carla.Transform(carla.Location(x=-5.0, z=2.0), carla.Rotation(pitch=6.0)), Attachment.SpringArm),
            # Watch back
            (carla.Transform(carla.Location(x=4.0, z=2.0), carla.Rotation(yaw=0, pitch=6)), Attachment.SpringArm),
            # Watch to the left
            (carla.Transform(carla.Location(x=0, z=2.0, y=3), carla.Rotation(yaw=-90, pitch=0)), Attachment.Rigid),
            # Watch to the right
            (carla.Transform(carla.Location(x=0, z=2.0, y=-3), carla.Rotation(yaw=90, pitch=0)), Attachment.Rigid),
            # First-person
            (carla.Transform(carla.Location(y=-0.42, x=0.03, z=1.2), carla.Rotation(yaw=0, roll=0, pitch=-10)), Attachment.Rigid)]

        self.transform_index = 1
        self.sensors = [
            ['sensor.camera.rgb', cc.Raw, 'Camera RGB', {}],
            ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)', {}],
            ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)', {}],
            ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic Gray Scale)', {}],
            ['sensor.camera.semantic_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)', {}],
            ['sensor.camera.semantic_segmentation', cc.CityScapesPalette, 'Camera Semantic Segmentation (CityScapes Palette)', {}],
            ['sensor.camera.instance_segmentation', cc.CityScapesPalette, 'Camera Instance Segmentation (CityScapes Palette)', {}],
            ['sensor.camera.instance_segmentation', cc.Raw, 'Camera Instance Segmentation (Raw)', {}],
            ['sensor.lidar.ray_cast', None, 'Lidar (Ray-Cast)', {'range': '50'}],
            ['sensor.camera.dvs', cc.Raw, 'Dynamic Vision Sensor', {}],
            ['sensor.camera.rgb', cc.Raw, 'Camera RGB Distorted',
                {'lens_circle_multiplier': '3.0',
                'lens_circle_falloff': '3.0',
                'chromatic_aberration_intensity': '0.5',
                'chromatic_aberration_offset': '0'}],
            ['sensor.camera.optical_flow', cc.Raw, 'Optical Flow', {}],
        ]
        world = self._parent.get_world()
```

```
        bp_library = world.get_blueprint_library()
        for item in self.sensors:
            bp = bp_library.find(item[0])
            if item[0].startswith('sensor.camera'):
                bp.set_attribute('image_size_x', str(hud.dim[0]))
                bp.set_attribute('image_size_y', str(hud.dim[1]))
                if bp.has_attribute('gamma'):
                    bp.set_attribute('gamma', str(gamma_correction))
                for attr_name, attr_value in item[3].items():
                    bp.set_attribute(attr_name, attr_value)
            elif item[0].startswith('sensor.lidar'):
                self.lidar_range = 50

                for attr_name, attr_value in item[3].items():
                    bp.set_attribute(attr_name, attr_value)
                    if attr_name == 'range':
                        self.lidar_range = float(attr_value)

            item.append(bp)
        self.index = None

    def toggle_camera(self):
        self.transform_index = (self.transform_index + 1) % len(self._camera_transforms)
        self.set_sensor(self.index, notify=False, force_respawn=True)

    def set_sensor(self, index, notify=True, force_respawn=False):
        index = index % len(self.sensors)
        needs_respawn = True if self.index is None else \
            (force_respawn or (self.sensors[index][2] != self.sensors[self.index][2]))
        if needs_respawn:
            if self.sensor is not None:
                self.sensor.destroy()
                self.surface = None
            self.sensor = self._parent.get_world().spawn_actor(
                self.sensors[index][-1],
                self._camera_transforms[self.transform_index][0],
                attach_to=self._parent,
                attachment_type=self._camera_transforms[self.transform_index][1])
            # We need to pass the lambda a weak reference to self to avoid
            # circular reference.
            weak_self = weakref.ref(self)
            self.sensor.listen(lambda image: CameraManager._parse_image(weak_self, image))
        if notify:
            self.hud.notification(self.sensors[index][2])
        self.index = index

    def next_sensor(self):
        self.set_sensor(self.index + 1)

    def toggle_recording(self):
        self.recording = not self.recording
        self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))

    def render(self, display):
        if self.surface is not None:
            display.blit(self.surface, (0, 0))

    @staticmethod
    def _parse_image(weak_self, image):
        self = weak_self()
        if not self:
            return
        if self.sensors[self.index][0].startswith('sensor.lidar'):
            points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
            points = np.reshape(points, (int(points.shape[0] / 4), 4))
            lidar_data = np.array(points[:, :2])
            lidar_data *= min(self.hud.dim) / (2.0 * self.lidar_range)
            lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
            lidar_data = np.fabs(lidar_data)  # pylint: disable=E1111
            lidar_data = lidar_data.astype(np.int32)
            lidar_data = np.reshape(lidar_data, (-1, 2))
            lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
            lidar_img = np.zeros((lidar_img_size), dtype=np.uint8)
            lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
            self.surface = pygame.surfarray.make_surface(lidar_img)
        elif self.sensors[self.index][0].startswith('sensor.camera.dvs'):
            # Example of converting the raw_data from a carla.DVSEventArray
            # sensor into a NumPy array and using it as an image
            dvs_events = np.frombuffer(image.raw_data, dtype=np.dtype([
                ('x', np.uint16), ('y', np.uint16), ('t', np.int64), ('pol', np.bool)]))
            dvs_img = np.zeros((image.height, image.width, 3), dtype=np.uint8)
            # Blue is positive, red is negative
            dvs_img[dvs_events[:]['y'], dvs_events[:]['x'], dvs_events[:]['pol'] * 2] = 255
            self.surface = pygame.surfarray.make_surface(dvs_img.swapaxes(0, 1))
        elif self.sensors[self.index][0].startswith('sensor.camera.optical_flow'):
            image = image.get_color_coded_flow()
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        else:
            image.convert(self.sensors[self.index][1])
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        if self.recording:
            image.save_to_disk('_out/%08d' % image.frame)


def transform_to_carla_locations(path_locations):
    locations = []
    for loc in path_locations:
        locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
    return locations

def parse_keys(world):
    for event in pygame.event.get():
        if event.type == pygame.QUIT or (event.type == pygame.KEYUP and event.key == K_ESCAPE):
            world.finishScenario()
            return True
        elif event.type == pygame.KEYUP:
            if event.key == K_TAB:
                world.camera_manager.toggle_camera()


def game_loop(args, scenario, path_details):
    pygame.init()
    pygame.font.init()
    world = None
    original_settings = None
    try:
        client = carla.Client(args.host, args.port)
        client.set_timeout(20.0)
        sim_world = client.get_world()
        if args.sync:
            original_settings = sim_world.get_settings()
            settings = sim_world.get_settings()
```

```
            if not settings.synchronous_mode:
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 0.05
            sim_world.apply_settings(settings)
            traffic_manager = client.get_trafficmanager()
            traffic_manager.set_synchronous_mode(True)

        display = pygame.display.set_mode(
            (args.width, args.height),
            pygame.HWSURFACE | pygame.DOUBLEBUF)
        display.fill((0,0,0))
        pygame.display.flip()
        scenario.setup(traffic_manager, sim_world)
        scenario.spawn(client)
        logger.info("The scenario was set. Spawning the hero vehicle and launching the simulation")
        client.set_timeout(10.0)
        hud = HUD(args.width, args.height)
        try:
            world = World(sim_world, hud, args, scenario, client, path_details)
        except Exception as e:
            logger.error("Could not initiate the simulation world correctly", exc_info=True)
        if args.sync:
            sim_world.tick()
        else:
            sim_world.wait_for_tick()

        clock = pygame.time.Clock()
        map = sim_world.get_map()
        time.sleep(3)
        try:
            locations = transform_to_carla_locations(path_details["path_checkpoints"])
            player = None
            actors = sim_world.get_actors()
            for actor in actors:
                if actor.attributes.get('role_name') == 'hero':
                    player = actor

            # traffic_manager.set_path(player, locations[1:])
            traffic_manager.auto_lane_change(player, True)
            traffic_manager.update_vehicle_lights(player, True)
            agent = BehaviorAgent(player, behavior="aggressive")
            #player.set_autopilot(True)
            world.startScenario()

        except Exception as e:
            logger.error("Could not setup the hero vehicle correctly", exc_info=True)
            raise e

        current_destination_index = 1
        agent.set_destination(locations[current_destination_index])
        while True:
            try:
                if agent.done():
                    if current_destination_index == len(locations)-1:
                        return
                    else:
                        current_destination_index += 1
                        agent.set_destination(locations[current_destination_index])
                player.apply_control(agent.run_step())

                if args.sync:
                    sim_world.tick()
                clock.tick_busy_loop(FPS)
                if world.tick():
                    return
                if parse_keys(world):
                    return
                world.render(display)
                pygame.display.flip()
            except Exception as e:
                raise e
    except Exception as e:
        logger.error("Something went wrong trying to launch the self_driver.py", exc_info=True)
    finally:
        if(scenario.populated):
            scenario.finish(client)
        if(world and world.manager):
            logger.info('Total time simulated: {}s'.format(str(world.simulation_time/1000)))

        if original_settings:
            sim_world.apply_settings(original_settings)

        if world is not None:
            world.destroy()

        pygame.quit()

argparser = argparse.ArgumentParser(
    description='CARLA AI Client')
argparser.add_argument(
    '-v', '--verbose',
    action='store_true',
    dest='debug',
    help='print debug information')
argparser.add_argument(
    '--host',
    metavar='H',
    default='127.0.0.1',
    help='IP of the host server (default: 127.0.0.1)')
argparser.add_argument(
    '-p', '--port',
    metavar='P',
    default=2000,
    type=int,
    help='TCP port to listen to (default: 2000)')
argparser.add_argument(
    '-a', '--autopilot',
    action='store_true',
    help='enable autopilot')
argparser.add_argument(
    '--res',
    metavar='WIDTHxHEIGHT',
    default='1280x720',
    help='window resolution (default: 1280x720)')
argparser.add_argument(
    '--filter', '-f',
    metavar='PATTERN',
    default='vehicle.*',
    help='actor filter (default: "vehicle.*")')
argparser.add_argument(
    '--generation',
    metavar='G',
    default='2',
    help='restrict to certain actor generation (values: "1","2","All" - default: "2")')
argparser.add_argument(
    '--rolename',
    metavar='NAME',
    default='hero',
    help='actor role name (default: "hero")')
argparser.add_argument(
```

```
    '--gamma',
    default=2.2,
    type=float,
    help='Gamma correction of the camera (default: 2.2)')
argparser.add_argument(
    '--sync',
    action='store_true',
    help='Activate synchronous mode execution')
argparser.add_argument(
    '--red',
    default="0",
    help='Red part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--green',
    default="0",
    help='Green part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--blue',
    default="0",
    help='Blue part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '-n', '--name',
    metavar='F',
    default="example",
    help='Participant name (name)')
argparser.add_argument(
    '-s', '--scenario',
    metavar='F',
    default="example",
    help='Scenario name (name)')
argparser.add_argument("s_b64")
argparser.add_argument("p_b64")
args = argparser.parse_args()

logger = logging.getLogger("self-driver-thread")
logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %(

def main():
    args.width, args.height = [int(x) for x in args.res.split('x')]

    print(__doc__)

    try:

        scenario_b64 = args.s_b64
        path_b64 = args.p_b64

        scenario_bytes = base64.b64decode(scenario_b64.encode())
        path_bytes = base64.b64decode(path_b64.encode())

        scenario = pickle.loads(scenario_bytes)
        path_details = pickle.loads(path_bytes)
        game_loop(args, scenario, path_details)

    except KeyboardInterrupt as e:
        logger.warning('Keyboard interupt while running self_driver.py script')
    except Exception as e:
        logger.error("Something went wrong.", exc_info=True)


if __name__ == '__main__':

    main()
```

**File: ./software/carla_scripts/driver_steeringwheel.py**

```
#!/usr/bin/env python

# Copyright (c) 2019 Intel Labs
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

# Allows controlling a vehicle with a keyboard. For a simpler and more
# documented example, please take a look at tutorial.py.

"""
Welcome to CARLA manual control with steering wheel Logitech G29.

To drive start by preshing the brake pedal.
Change your wheel_config.ini according to your steering wheel.

To find out the values of your steering wheel use jstest-gtk in Ubuntu.

"""

from __future__ import print_function


# ==============================================================================
# -- find carla module ---------------------------------------------------------
# ==============================================================================


import glob
import os
import sys

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass


# ==============================================================================
# -- imports -------------------------------------------------------------------
# ==============================================================================


import carla

from carla import ColorConverter as cc

import argparse
import collections
import datetime
import logging
import math
import random
import re
import weakref
from recording.manager import Manager
```

```
from config import FPS
import pickle
import base64

if sys.version_info >= (3, 0):

    from configparser import ConfigParser

else:

    from ConfigParser import RawConfigParser as ConfigParser

try:
    import pygame
    from pygame.locals import KMOD_CTRL
    from pygame.locals import KMOD_SHIFT
    from pygame.locals import K_0
    from pygame.locals import K_9
    from pygame.locals import K_BACKQUOTE
    from pygame.locals import K_BACKSPACE
    from pygame.locals import K_COMMA
    from pygame.locals import K_DOWN
    from pygame.locals import K_ESCAPE
    from pygame.locals import K_F1
    from pygame.locals import K_LEFT
    from pygame.locals import K_PERIOD
    from pygame.locals import K_RIGHT
    from pygame.locals import K_SLASH
    from pygame.locals import K_SPACE
    from pygame.locals import K_TAB
    from pygame.locals import K_UP
    from pygame.locals import K_a
    from pygame.locals import K_c
    from pygame.locals import K_d
    from pygame.locals import K_h
    from pygame.locals import K_m
    from pygame.locals import K_p
    from pygame.locals import K_q
    from pygame.locals import K_r
    from pygame.locals import K_s
    from pygame.locals import K_w
except ImportError:
    raise RuntimeError('cannot import pygame, make sure pygame package is installed')

try:
    import numpy as np
except ImportError:
    raise RuntimeError('cannot import numpy, make sure numpy package is installed')


# ==============================================================================
# -- Global functions ----------------------------------------------------------
# ==============================================================================


def find_weather_presets():
    rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')
    name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))
    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]


def get_actor_display_name(actor, truncate=250):
    name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
    return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name


# ==============================================================================
# -- World ---------------------------------------------------------------------
# ==============================================================================


class World(object):
    def __init__(self, carla_world, hud, args, scenario, client, path_details):
        self.world = carla_world
        self.sync = args.sync
        self.path_details = path_details
        self.scenario = scenario
        self.client = client
        self.scenario_name = args.scenario
        self.participant_name = args.name

        self.spawn_point = None
        self.init_spawn_point(path_details["start_location"])
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = args.filter
        self._gamma = args.gamma
        self.restart(args)

        # All about the simulation
        self.simulation_clock = None
        self.simulation_start_tick = 0
        self.simulation_time = 0
        self.manager = None

        self.world.on_tick(hud.on_world_tick)

    def init_spawn_point(self, start_location):
        try:
            self.spawn_point = carla.Transform(carla.Location(x=start_location["x"], y=start_location["y"], z=start_location["z"]), carla.Rotation(pitch=0.0, yaw
        except Exception as e:
            logger.error("Could not initiate spawn location for the driver", e)

    def restart(self, args):
        # Keep same camera config if the camera manager exists.
        cam_index = self.camera_manager.index if self.camera_manager is not None else 0
        cam_pos_index = self.camera_manager.transform_index if self.camera_manager is not None else 0
        # Get a random blueprint.
        blueprint = random.choice(self.world.get_blueprint_library().filter(self._actor_filter))
        blueprint.set_attribute('role_name', 'hero')
        if blueprint.has_attribute('color'):
            blueprint.set_attribute('color', '{},{},{}'.format(args.red, args.green, args.blue))
        # Spawn the player.
        if self.player is not None:
            spawn_point = self.player.get_transform()
            spawn_point.location.z += 2.0
            spawn_point.rotation.roll = 0.0
            spawn_point.rotation.pitch = 0.0
            self.destroy()
            self.player = self.world.try_spawn_actor(blueprint, spawn_point)
        while self.player is None:
            self.player = self.world.try_spawn_actor(blueprint, self.spawn_point)
        # Set up the sensors.
```

```
        self.collision_sensor = CollisionSensor(self.player, self.hud)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.hud)
        self.gnss_sensor = GnssSensor(self.player)
        self.camera_manager = CameraManager(self.player, self.hud, args.gamma)
        self.camera_manager.transform_index = cam_pos_index
        self.camera_manager.set_sensor(cam_index, notify=False)
        actor_type = get_actor_display_name(self.player)
        self.hud.notification(actor_type)

        if self.sync:
            self.world.tick()
        else:
            self.world.wait_for_tick()

    def next_weather(self, reverse=False):
        self._weather_index += -1 if reverse else 1
        self._weather_index %= len(self._weather_presets)
        preset = self._weather_presets[self._weather_index]
        self.hud.notification('Weather: %s' % preset[1])
        self.player.get_world().set_weather(preset[0])

    def startScenario(self):
        if self.simulation_clock is None:
            logger.info("Starting the scenario")
            self.scenario.start()
            self.hud.notification('The simulation has started')
            self.manager = Manager(self.path_details["path_checkpoints"], self.world, self.player, self.participant_name, self.scenario_name)
            #self.collision_sensor = self.manager.collision_sensor
            self.simulation_clock = pygame.time.Clock()
            self.simulation_start_tick = pygame.time.get_ticks()

    def finishScenario(self):
        logger.warning("Finish was NOT REACHED. Terminating the simulation.")
        self.manager.shut_down(False)
        self.scenario.finish(self.client)

    def tick(self):
        if self.simulation_clock is not None:
            self.simulation_clock.tick(60)
            self.simulation_time += self.simulation_clock.get_time()
            if self.manager.record(self.simulation_time):
                return True
            self.hud.tick(self, self.simulation_time/1000)
        else:
            self.hud.tick(self, 0)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy(self):
        sensors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor]
        for sensor in sensors:
            if sensor is not None:
                sensor.stop()
                sensor.destroy()
        if self.player is not None:
            self.player.destroy()

# ==============================================================================
# -- DualControl -----------------------------------------------------------
# ==============================================================================


class DualControl(object):
    def __init__(self, world, start_in_autopilot):
        self._autopilot_enabled = start_in_autopilot
        if isinstance(world.player, carla.Vehicle):
            self._control = carla.VehicleControl()
            self._lights = carla.VehicleLightState.NONE
            world.player.set_autopilot(self._autopilot_enabled)
        elif isinstance(world.player, carla.Walker):
            self._control = carla.WalkerControl()
            self._autopilot_enabled = False
            self._rotation = world.player.get_transform().rotation
        else:
            raise NotImplementedError("Actor type not supported")
        self._steer_cache = 0.0
        world.hud.notification("Press 'H' or '?' for help.", seconds=4.0)

        # initialize steering wheel
        pygame.joystick.init()
        self.word = world
        joystick_count = pygame.joystick.get_count()
        if joystick_count > 1:
            raise ValueError("Please Connect Just One Joystick")

        self._joystick = pygame.joystick.Joystick(0)
        self._joystick.init()

        self._parser = ConfigParser()
        self._parser.read('wheel_config.ini')
        self._steer_idx = int(
            self._parser.get('G29 Racing Wheel', 'steering_wheel'))
        self._throttle_idx = int(
            self._parser.get('G29 Racing Wheel', 'throttle'))
        self._brake_idx = int(self._parser.get('G29 Racing Wheel', 'brake'))
        self._reverse_idx = int(self._parser.get('G29 Racing Wheel', 'reverse'))
        self._handbrake_idx = int(
            self._parser.get('G29 Racing Wheel', 'handbrake'))

    def parse_events(self, client, world, clock, sync_mode):
        if isinstance(self._control, carla.VehicleControl):
            current_lights = self._lights
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return True
            elif event.type == pygame.JOYBUTTONDOWN:
                if event.button == 1:
                    world.hud.toggle_info()
                elif event.button == 19:
                    world.camera_manager.toggle_camera()
                elif event.button == 19:
                    world.camera_manager.toggle_camera_backward()
                elif event.button == self._reverse_idx:
                    self._control.gear = 1 if self._control.reverse else -1
                elif event.button == 23:
                    if not self._lights & carla.VehicleLightState.Position:
                        world.hud.notification("Position lights")
                        current_lights |= carla.VehicleLightState.Position
                    else:
                        world.hud.notification("Low beam lights")
                        current_lights |= carla.VehicleLightState.LowBeam
                    if self._lights & carla.VehicleLightState.LowBeam:
                        world.hud.notification("Fog lights")
                        current_lights |= carla.VehicleLightState.Fog
```

```
                    if self._lights & carla.VehicleLightState.Fog:
                        world.hud.notification("Lights off")
                        current_lights ^= carla.VehicleLightState.Position
                        current_lights ^= carla.VehicleLightState.LowBeam
                        current_lights ^= carla.VehicleLightState.Fog
                elif event.button == 3 and not world.simulation_clock:
                    world.startScenario()
                elif event.button == 3 and world.simulation_clock:
                    world.finishScenario()
                    return True
                elif event.button == 5:
                    current_lights ^= carla.VehicleLightState.LeftBlinker
                elif event.button == 4:
                    current_lights ^= carla.VehicleLightState.RightBlinker

            elif event.type == pygame.KEYUP:
                if self._is_quit_shortcut(event.key):
                    return True
                elif event.key == K_BACKSPACE and not world.simulation_clock:
                    world.startScenario()
                elif event.key == K_BACKSPACE and world.simulation_clock:
                    world.finishScenario()
                    return True
                elif event.key == K_F1:
                    world.hud.toggle_info()
                elif event.key == K_h or (event.key == K_SLASH and pygame.key.get_mods() & KMOD_SHIFT):
                    world.hud.help.toggle()
                elif event.key == K_TAB:
                    world.camera_manager.toggle_camera()
                elif event.key == K_c and pygame.key.get_mods() & KMOD_SHIFT:
                    world.next_weather(reverse=True)
                elif event.key == K_c:
                    world.next_weather()
                elif event.key == K_BACKQUOTE:
                    world.camera_manager.next_sensor()
                elif event.key > K_0 and event.key <= K_9:
                    world.camera_manager.set_sensor(event.key - 1 - K_0)
                elif event.key == K_r:
                    world.camera_manager.toggle_recording()
                if isinstance(self._control, carla.VehicleControl):
                    if event.key == K_q:
                        self._control.gear = 1 if self._control.reverse else -1
                    elif event.key == K_m:
                        self._control.manual_gear_shift = not self._control.manual_gear_shift
                        self._control.gear = world.player.get_control().gear
                        world.hud.notification('%s Transmission' %
                                               ('Manual' if self._control.manual_gear_shift else 'Automatic'))
                    elif self._control.manual_gear_shift and event.key == K_COMMA:
                        self._control.gear = max(-1, self._control.gear - 1)
                    elif self._control.manual_gear_shift and event.key == K_PERIOD:
                        self._control.gear = self._control.gear + 1
                    elif event.key == K_p:
                        self._autopilot_enabled = not self._autopilot_enabled
                        world.player.set_autopilot(self._autopilot_enabled)
                        world.hud.notification('Autopilot %s' % ('On' if self._autopilot_enabled else 'Off'))

        if not self._autopilot_enabled:
            if isinstance(self._control, carla.VehicleControl):
                self._parse_vehicle_keys(pygame.key.get_pressed(), clock.get_time())
                self._parse_vehicle_wheel()
                self._control.reverse = self._control.gear < 0

                if self._control.brake:
                    current_lights |= carla.VehicleLightState.Brake
                else: # Remove the Brake flag
                    current_lights &= ~carla.VehicleLightState.Brake
                if self._control.reverse:
                    current_lights |= carla.VehicleLightState.Reverse
                else: # Remove the Reverse flag
                    current_lights &= ~carla.VehicleLightState.Reverse
                if current_lights != self._lights: # Change the light state only if necessary
                    self._lights = current_lights
                    world.player.set_light_state(carla.VehicleLightState(self._lights))

            elif isinstance(self._control, carla.WalkerControl):
                self._parse_walker_keys(pygame.key.get_pressed(), clock.get_time())
            world.player.apply_control(self._control)

    def _parse_vehicle_keys(self, keys, milliseconds):
        self._control.throttle = 1.0 if keys[K_UP] or keys[K_w] else 0.0
        steer_increment = 5e-4 * milliseconds
        if keys[K_LEFT] or keys[K_a]:
            self._steer_cache -= steer_increment
        elif keys[K_RIGHT] or keys[K_d]:
            self._steer_cache += steer_increment
        else:
            self._steer_cache = 0.0
        self._steer_cache = min(0.7, max(-0.7, self._steer_cache))
        self._control.steer = round(self._steer_cache, 1)
        self._control.brake = 1.0 if keys[K_DOWN] or keys[K_s] else 0.0
        self._control.hand_brake = keys[K_SPACE]

    def _parse_vehicle_wheel(self):
        numAxes = self._joystick.get_numaxes()
        jsInputs = [float(self._joystick.get_axis(i)) for i in range(numAxes)]
        # print (jsInputs)
        jsButtons = [float(self._joystick.get_button(i)) for i in
                     range(self._joystick.get_numbuttons())]

        # Custom function to map range of inputs [1, -1] to outputs [0, 1] i.e 1 from inputs means nothing is pressed
        # For the steering, it seems fine as it is
        K1 = 1.0  # 0.55
        steerCmd = K1 * math.tan(1.1 * jsInputs[self._steer_idx])

        K2 = 1.6  # 1.6
        throttleCmd = K2 + (2.05 * math.log10(
            -0.7 * jsInputs[self._throttle_idx] + 1.4) - 1.2) / 0.92
        if throttleCmd <= 0:
            throttleCmd = 0
        elif throttleCmd > 1:
            throttleCmd = 1

        brakeCmd = 1.6 + (2.05 * math.log10(
            -0.7 * jsInputs[self._brake_idx] + 1.4) - 1.2) / 0.92

        if brakeCmd <= 0:
            brakeCmd = 0
        elif brakeCmd > 1:
            brakeCmd = 1

        self._control.steer = steerCmd
        self._control.brake = brakeCmd
        self._control.throttle = throttleCmd

        #toggle = jsButtons[self._reverse_idx]

        self._control.hand_brake = bool(jsButtons[self._handbrake_idx])

    def _parse_walker_keys(self, keys, milliseconds):
        self._control.speed = 0.0
```

```
        if keys[K_DOWN] or keys[K_s]:
            self._control.speed = 0.0
        if keys[K_LEFT] or keys[K_a]:
            self._control.speed = .01
            self._rotation.yaw -= 0.08 * milliseconds
        if keys[K_RIGHT] or keys[K_d]:
            self._control.speed = .01
            self._rotation.yaw += 0.08 * milliseconds
        if keys[K_UP] or keys[K_w]:
            self._control.speed = 5.556 if pygame.key.get_mods() & KMOD_SHIFT else 2.778
        self._control.jump = keys[K_SPACE]
        self._rotation.yaw = round(self._rotation.yaw, 1)
        self._control.direction = self._rotation.get_forward_vector()

    @staticmethod
    def _is_quit_shortcut(key):
        return (key == K_ESCAPE) or (key == K_q and pygame.key.get_mods() & KMOD_CTRL)


# ==============================================================================
# -- HUD -----------------------------------------------------------------------
# ==============================================================================


class HUD(object):
    def __init__(self, width, height):
        self.dim = (width, height)
        font = pygame.font.Font(pygame.font.get_default_font(), 20)
        font_name = 'courier' if os.name == 'nt' else 'mono'
        fonts = [x for x in pygame.font.get_fonts() if font_name in x]
        default_font = 'ubuntumono'
        mono = default_font if default_font in fonts else fonts[0]
        mono = pygame.font.match_font(mono)
        self._font_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)
        self._notifications = FadingText(font, (width, 40), (0, height - 40))
        self.help = HelpText(pygame.font.Font(mono, 24), width, height)
        self.server_fps = 0
        self.frame = 0
        self.simulation_time = 0
        self._show_info = True
        self._info_text = []
        self._server_clock = pygame.time.Clock()

    def on_world_tick(self, timestamp):
        self._server_clock.tick()
        self.server_fps = self._server_clock.get_fps()
        self.frame = timestamp.frame
        self.simulation_time = timestamp.elapsed_seconds

    def tick(self, world, time):
        self._notifications.tick(world, time)
        if not self._show_info:
            return
        t = world.player.get_transform()
        v = world.player.get_velocity()
        c = world.player.get_control()
        heading = 'N' if abs(t.rotation.yaw) < 89.5 else ''
        heading += 'S' if abs(t.rotation.yaw) > 90.5 else ''
        heading += 'E' if 179.5 > t.rotation.yaw > 0.5 else ''
        heading += 'W' if -0.5 > t.rotation.yaw > -179.5 else ''
        colhist = world.collision_sensor.get_collision_history()
        collision = [colhist[x + self.frame - 200] for x in range(0, 200)]
        max_col = max(1.0, max(collision))
        collision = [x / max_col for x in collision]
        vehicles = world.world.get_actors().filter('vehicle.*')
        self._info_text = [
            'Vehicle: % 20s' % get_actor_display_name(world.player, truncate=20),
            'Route time: % 12s' % datetime.timedelta(seconds=int(time)),
            '',
            'Speed:   % 15.0f km/h' % (3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)),
            '']
        if isinstance(c, carla.VehicleControl):
            self._info_text += [
                ('Throttle:', c.throttle, 0.0, 1.0),
                ('Steer:', c.steer, -1.0, 1.0),
                ('Brake:', c.brake, 0.0, 1.0),
                ('Reverse:', c.reverse),
                ('Hand brake:', c.hand_brake),
                ('Manual:', c.manual_gear_shift),
                'Gear:        %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)]
        elif isinstance(c, carla.WalkerControl):
            self._info_text += [
                ('Speed:', c.speed, 0.0, 5.556),
                ('Jump:', c.jump)]
        self._info_text += [
            '',
            'Collision:',
            collision,
            '']

    def toggle_info(self):
        self._show_info = not self._show_info

    def notification(self, text, seconds=2.0):
        self._notifications.set_text(text, seconds=seconds)

    def error(self, text):
        self._notifications.set_text('Error: %s' % text, (255, 0, 0))

    def render(self, display):
        if self._show_info:
            info_surface = pygame.Surface((220, self.dim[1]))
            info_surface.set_alpha(100)
            display.blit(info_surface, (0, 0))
            v_offset = 4
            bar_h_offset = 100
            bar_width = 106
            for item in self._info_text:
                if v_offset + 18 > self.dim[1]:
                    break
                if isinstance(item, list):
                    if len(item) > 1:
                        points = [(x + 8, v_offset + 8 + (1.0 - y) * 30) for x, y in enumerate(item)]
                        pygame.draw.lines(display, (255, 136, 0), False, points, 2)
                    item = None
                    v_offset += 18
                elif isinstance(item, tuple):
                    if isinstance(item[1], bool):
                        rect = pygame.Rect((bar_h_offset, v_offset + 8), (6, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1] else 1)
                    else:
                        rect_border = pygame.Rect((bar_h_offset, v_offset + 8), (bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect_border, 1)
                        f = (item[1] - item[2]) / (item[3] - item[2])
                        if item[2] < 0.0:
                            rect = pygame.Rect((bar_h_offset + f * (bar_width - 6), v_offset + 8), (6, 6))
                        else:
                            rect = pygame.Rect((bar_h_offset, v_offset + 8), (f * bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect)
                    item = item[0]
```

```
                if item:  # At this point has to be a str.
                    surface = self._font_mono.render(item, True, (255, 255, 255))
                    display.blit(surface, (8, v_offset))
                v_offset += 18
        self._notifications.render(display)
        self.help.render(display)


# ==============================================================================
# -- FadingText ----------------------------------------------------------------
# ==============================================================================


class FadingText(object):
    def __init__(self, font, dim, pos):
        self.font = font
        self.dim = dim
        self.pos = pos
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)

    def set_text(self, text, color=(255, 255, 255), seconds=2.0):
        text_texture = self.font.render(text, True, color)
        self.surface = pygame.Surface(self.dim)
        self.seconds_left = seconds
        self.surface.fill((0, 0, 0, 0))
        self.surface.blit(text_texture, (10, 11))

    def tick(self, _, time):
        delta_seconds = 1e-3 * time
        self.seconds_left = max(0.0, self.seconds_left - delta_seconds)
        self.surface.set_alpha(500.0 * self.seconds_left)

    def render(self, display):
        display.blit(self.surface, self.pos)


# ==============================================================================
# -- HelpText ------------------------------------------------------------------
# ==============================================================================


class HelpText(object):
    def __init__(self, font, width, height):
        lines = __doc__.split('\n')
        self.font = font
        self.dim = (680, len(lines) * 22 + 12)
        self.pos = (0.5 * width - 0.5 * self.dim[0], 0.5 * height - 0.5 * self.dim[1])
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)
        self.surface.fill((0, 0, 0, 0))
        for n, line in enumerate(lines):
            text_texture = self.font.render(line, True, (255, 255, 255))
            self.surface.blit(text_texture, (22, n * 22))
            self._render = False
        self.surface.set_alpha(220)

    def toggle(self):
        self._render = not self._render

    def render(self, display):
        if self._render:
            display.blit(self.surface, self.pos)


# ==============================================================================
# -- CollisionSensor -----------------------------------------------------------
# ==============================================================================


class CollisionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self.history = []
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.collision')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))

    def get_collision_history(self):
        history = collections.defaultdict(int)
        for frame, intensity in self.history:
            history[frame] += intensity
        return history

    @staticmethod
    def _on_collision(weak_self, event):
        self = weak_self()
        if not self:
            return
        actor_type = get_actor_display_name(event.other_actor)
        self.hud.notification('Collision with %r' % actor_type)
        impulse = event.normal_impulse
        intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
        self.history.append((event.frame, intensity))
        if len(self.history) > 4000:
            self.history.pop(0)


# ==============================================================================
# -- LaneInvasionSensor --------------------------------------------------------
# ==============================================================================


class LaneInvasionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))

    @staticmethod
    def _on_invasion(weak_self, event):
        self = weak_self()
        if not self:
            return
        lane_types = set(x.type for x in event.crossed_lane_markings)
        text = ['%r' % str(x).split()[-1] for x in lane_types]
        self.hud.notification('Crossed line %s' % ' and '.join(text))
```

```
710
711 # ==============================================================================
712 # -- GnssSensor ----------------------------------------------------------------
713 # ==============================================================================
714
715
716 class GnssSensor(object):
717     def __init__(self, parent_actor):
718         self.sensor = None
719         self._parent = parent_actor
720         self.lat = 0.0
721         self.lon = 0.0
722         world = self._parent.get_world()
723         bp = world.get_blueprint_library().find('sensor.other.gnss')
724         self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0, z=2.8)), attach_to=self._parent)
725         # We need to pass the lambda a weak reference to self to avoid circular
726         # reference.
727         weak_self = weakref.ref(self)
728         self.sensor.listen(lambda event: GnssSensor._on_gnss_event(weak_self, event))
729
730     @staticmethod
731     def _on_gnss_event(weak_self, event):
732         self = weak_self()
733         if not self:
734             return
735         self.lat = event.latitude
736         self.lon = event.longitude
737
738
739 # ==============================================================================
740 # -- CameraManager -------------------------------------------------------------
741 # ==============================================================================
742
743
744 class CameraManager(object):
745     def __init__(self, parent_actor, hud, gamma_correction):
746         self.sensor = None
747         self.surface = None
748         self._parent = parent_actor
749         self.hud = hud
750         self.recording = False
751         self._camera_transforms = [
752             carla.Transform(carla.Location(x=-5.5, z=2.8), carla.Rotation(pitch=-15)),
753             carla.Transform(carla.Location(x=1.6, z=1.7))]
754         self.transform_index = 1
755         self.sensors = [
756             ['sensor.camera.rgb', cc.Raw, 'Camera RGB'],
757             ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)'],
758             ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)'],
759             ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic Gray Scale)'],
760             ['sensor.camera.semantic_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)'],
761             ['sensor.camera.semantic_segmentation', cc.CityScapesPalette,
762                 'Camera Semantic Segmentation (CityScapes Palette)'],
763             ['sensor.lidar.ray_cast', None, 'Lidar (Ray-Cast)']]
764         world = self._parent.get_world()
765         bp_library = world.get_blueprint_library()
766         for item in self.sensors:
767             bp = bp_library.find(item[0])
768             if item[0].startswith('sensor.camera'):
769                 bp.set_attribute('image_size_x', str(hud.dim[0]))
770                 bp.set_attribute('image_size_y', str(hud.dim[1]))
771             elif item[0].startswith('sensor.lidar'):
772                 bp.set_attribute('range', '50')
773             item.append(bp)
774         self.index = None
775
776     def toggle_camera(self):
777         self.transform_index = (self.transform_index + 1) % len(self._camera_transforms)
778         self.sensor.set_transform(self._camera_transforms[self.transform_index])
779
780     def toggle_camera_backward(self):
781         self.transform_index = (self.transform_index -1 ) % len(self._camera_transforms)
782         self.sensor.set_transform(self._camera_transforms[self.transform_index])
783
784     def set_sensor(self, index, notify=True):
785         index = index % len(self.sensors)
786         needs_respawn = True if self.index is None \
787             else self.sensors[index][0] != self.sensors[self.index][0]
788         if needs_respawn:
789             if self.sensor is not None:
790                 self.sensor.destroy()
791                 self.surface = None
792             self.sensor = self._parent.get_world().spawn_actor(
793                 self.sensors[index][-1],
794                 self._camera_transforms[self.transform_index],
795                 attach_to=self._parent)
796             # We need to pass the lambda a weak reference to self to avoid
797             # circular reference.
798             weak_self = weakref.ref(self)
799             self.sensor.listen(lambda image: CameraManager._parse_image(weak_self, image))
800         if notify:
801             self.hud.notification(self.sensors[index][2])
802         self.index = index
803
804     def next_sensor(self):
805         self.set_sensor(self.index + 1)
806
807     def toggle_recording(self):
808         self.recording = not self.recording
809         self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))
810
811     def render(self, display):
812         if self.surface is not None:
813             display.blit(self.surface, (0, 0))
814
815     @staticmethod
816     def _parse_image(weak_self, image):
817         self = weak_self()
818         if not self:
819             return
820         if self.sensors[self.index][0].startswith('sensor.lidar'):
821             points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
822             points = np.reshape(points, (int(points.shape[0] / 4), 4))
823             lidar_data = np.array(points[:, :2])
824             lidar_data *= min(self.hud.dim) / 100.0
825             lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
826             lidar_data = np.fabs(lidar_data) # pylint: disable=E1111
827             lidar_data = lidar_data.astype(np.int32)
828             lidar_data = np.reshape(lidar_data, (-1, 2))
829             lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
830             lidar_img = np.zeros(lidar_img_size)
831             lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
832             self.surface = pygame.surfarray.make_surface(lidar_img)
833         else:
834             image.convert(self.sensors[self.index][1])
835             array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
836             array = np.reshape(array, (image.height, image.width, 4))
837             array = array[:, :, :3]
838             array = array[:, :, ::-1]
839             self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
```

```
        if self.recording:
            image.save_to_disk('_out/%08d' % image.frame)


# ==============================================================================
# -- game_loop() ---------------------------------------------------------------
# ==============================================================================


def game_loop(args, scenario, path_details):
    pygame.init()
    pygame.font.init()
    world = None
    original_settings = None
    try:
        client = carla.Client(args.host, args.port)
        client.set_timeout(20.0)
        sim_world = client.get_world()
        if args.sync:
            original_settings = sim_world.get_settings()
            settings = sim_world.get_settings()
            if not settings.synchronous_mode:
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 0.05
            sim_world.apply_settings(settings)
            traffic_manager = client.get_trafficmanager()
            traffic_manager.set_synchronous_mode(True)

        display = pygame.display.set_mode(
            (args.width, args.height),
            pygame.HWSURFACE | pygame.DOUBLEBUF)

        display.fill((0,0,0))
        pygame.display.flip()
        ### MY PART ###
        scenario.setup(traffic_manager, sim_world)
        scenario.spawn(client)
        logger.info("The scenario was set. Spawning the hero vehicle and launching the simulation")
        ###   ###   ###
        hud = HUD(args.width, args.height)
        try:
            world = World(sim_world, hud, args, scenario, client, path_details)
            controller = DualControl(world, args.autopilot)
        except Exception as e:
            logger.error(e)
        if args.sync:
            sim_world.tick()
        else:
            sim_world.wait_for_tick()

        clock = pygame.time.Clock()
        while True:
            try:
                if args.sync:
                    sim_world.tick()
                clock.tick_busy_loop(FPS)
                if controller.parse_events(client, world, clock, args.sync):
                    return
                if world.tick():
                    return
                world.render(display)
                pygame.display.flip()
            except Exception as e:
                logger.error(e)
    except Exception as e:
        logger.error("Something went wrong trying to launch the driver.py", e)
    finally:
        ### MY PART ###
        if(scenario.populated):
            scenario.finish(client)
        if(world and world.manager):
            logger.info('Total time simulated: {}s'.format(str(world.simulation_time/1000)))
        ###   ###   ###

        if original_settings:
            sim_world.apply_settings(original_settings)

        if world is not None:
            world.destroy()

        pygame.quit()


# ==============================================================================
# -- main() --------------------------------------------------------------------
# ==============================================================================

argparser = argparse.ArgumentParser(
    description='CARLA Manual Control Client')
argparser.add_argument(
    '-v', '--verbose',
    action='store_true',
    dest='debug',
    help='print debug information')
argparser.add_argument(
    '--host',
    metavar='H',
    default='127.0.0.1',
    help='IP of the host server (default: 127.0.0.1)')
argparser.add_argument(
    '-p', '--port',
    metavar='P',
    default=2000,
    type=int,
    help='TCP port to listen to (default: 2000)')
argparser.add_argument(
    '-a', '--autopilot',
    action='store_true',
    help='enable autopilot')
argparser.add_argument(
    '--res',
    metavar='WIDTHxHEIGHT',
    default='1280x720',
    help='window resolution (default: 1280x720)')
argparser.add_argument(
    '--filter', '-f',
    metavar='PATTERN',
    default='vehicle.*',
    help='actor filter (default: "vehicle.*")')
argparser.add_argument(
'--generation',
metavar='G',
default='2',
help='restrict to certain actor generation (values: "1","2","All" - default: "2")')
argparser.add_argument(
    '--rolename',
    metavar='NAME',
    default='hero',
    help='actor role name (default: "hero")')
argparser.add_argument(
```

```
    '--gamma',
    default=2.2,
    type=float,
    help='Gamma correction of the camera (default: 2.2)')
argparser.add_argument(
    '--sync',
    action='store_true',
    help='Activate synchronous mode execution')
argparser.add_argument(
    '--red',
    default="0",
    help='Red part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--green',
    default="0",
    help='Green part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--blue',
    default="0",
    help='Blue part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '-n', '--name',
    metavar='F',
    default="example",
    help='Participant name (name)')
argparser.add_argument(
    '-s', '--scenario',
    metavar='F',
    default="example",
    help='Scenario name (name)')
argparser.add_argument("s_b64")
argparser.add_argument("p_b64")
args = argparser.parse_args()

logger = logging.getLogger("driver-thread")
logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %

def main():
    args.width, args.height = [int(x) for x in args.res.split('x')]

    print(__doc__)

    try:

        scenario_b64 = args.s_b64
        path_b64 = args.p_b64

        scenario_bytes = base64.b64decode(scenario_b64.encode())
        path_bytes = base64.b64decode(path_b64.encode())

        scenario = pickle.loads(scenario_bytes)
        path_details = pickle.loads(path_bytes)
        game_loop(args, scenario, path_details)

    except KeyboardInterrupt as e:
        logger.warning('Keyboard interupt while running driver.py script', e)
    except Exception as e:
        logger.error("Something went wrong.", e)


if __name__ == '__main__':

    main()
```

**File: ./software/carla_scripts/simulation_manager.py**

```
import json
import logging
import sys
from scenarios.scenario_factory import ScenarioFactory
from scenarios.path_builder import PathBuilder
import glob
import os
import time
import pickle
import subprocess
import base64
from threading import Thread

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla


logger = logging.getLogger(__name__)

class SimulationManager():

    def __init__(self, scenario_name, path_name, participant_name, vehicle_blueprint, randomization_seed, ai):
        self.vehicle_blueprint = vehicle_blueprint
        self.scenario_name = scenario_name
        self.path_name = path_name
        self.participant_name = participant_name
        self.ai = ai
        self.seed = randomization_seed
        self.scenario = None

    def begin_simulation(self):
        try:
            scenario_data = self.load_scenario()
            path_details = self.determine_path_details(scenario_data)
            self.change_map(path_details["town"])
            self.change_weather(scenario_data)
            self.scenario = ScenarioFactory.make_scenario(scenario_data, path_details, self.seed)
            self.launch_driving_mode(self.scenario, path_details)

        except Exception as e:
            raise e

    # Either generate the path or get it from file if such is defined
    def determine_path_details(self, scenario_data):
        if self.path_name:
            details = self.read_path_details()
        else:
            logger.info("No predefined details to load. The details will be generated automatically.")
            details = PathBuilder.generate_details(scenario_data)
        return details

```

```
    # Read path details to predefine the scenario if the path is specified
    def read_path_details(self):
        logger.info("Loading predefined details for the scenario")
        file_path = "../../data/paths/{}.json".format(self.path_name)
        try:
            with open(file_path) as file:
                details = json.load(file)
                logger.info("Successfully read additional scenario details from {}.json data".format(self.path_name))
                return details
        except FileNotFoundError as e:
            logger.error("Path {} file could not be found.".format(self.path_name), exc_info=True)
            raise e
        except json.JSONDecodeError as e:
            logger.error("The file {}.json could not be decoded as JSON".format(self.path_name), exc_info=True)
            raise e


    # Get a file indicating scenario parameters
    def load_scenario(self):
        logger.info("Loading {} data".format(self.scenario_name))
        file_path = "../../data/scenario_generation_data/generated_scenarios/{}.json".format(self.scenario_name)
        try:
            with open(file_path) as file:
                scenario_data = json.load(file)
                logger.info("Successfully read scenario {}.json data".format(self.scenario_name))
                return scenario_data
        except FileNotFoundError as e:
            logger.error("Scenario {} file could not be found.".format(self.scenario_name), exc_info=True)
            raise e
        except json.JSONDecodeError as e:
            logger.error("The file {}.json could not be decoded as JSON".format(self.scenario_name), exc_info=True)
            raise e

    # Used to kill all the actors in the simulation
    def close_scenario(self):
        client = carla.Client('127.0.0.1', 2000)
        client.set_timeout(20.0)
        if(self.scenario is not None and self.scenario.populated):
            logger.info("The scenario was running. Finishing it.")
            self.scenario.finish(client)

    # Change the weather in the simulation
    def change_weather(self, scenario_data):
        try:
            logger.info("Changing the weather")
            client = carla.Client('localhost', 2000)
            world = client.get_world()
            weather = carla.WeatherParameters(
                cloudiness = scenario_data["cloudiness"],
                precipitation = scenario_data["precipitation"],
                precipitation_deposits = scenario_data["precipitation_deposits"],
                wind_intensity = scenario_data["wind_intensity"],
                sun_azimuth_angle = scenario_data["sun_azimuth_angle"],
                sun_altitude_angle = scenario_data["sun_altitude_angle"],
                fog_density = scenario_data["fog_density"],
                fog_distance = scenario_data["fog_distance"],
                wetness = scenario_data["wetness"],
                fog_falloff = scenario_data["fog_falloff"],
                scattering_intensity = scenario_data["scattering_intensity"],
                mie_scattering_scale = scenario_data["mie_scattering_scale"],
                rayleigh_scattering_scale = scenario_data["rayleigh_scattering_scale"],
                # dust_storm = scenario_data["dust_storm"],
            )

            world.set_weather(weather)
            logger.info("Finished changing the weather")
        except Exception as e:
            logger.error("Something wrong happened trying to change the weather. Aborting...", exc_info=True)
            raise e

    # Change the simulation map
    def change_map(self, map):
        try:
            logger.info("Changing the map to {}.".format(map))
            client = carla.Client('localhost', 2000)
            client.set_timeout(10.0)
            world = client.load_world(map)
            # Give 2 sec for the map to load
            time.sleep(2)
            logger.info("Finished changing the map")
        except Exception as e:
            logger.exception("Something wrong happened trying to change the map. Aborting...", exc_info=True)
            raise e


    # Launch the first person driving mode
    def launch_driving_mode(self, scenario, path_details):
        scenario_bytes = pickle.dumps(scenario)
        path_bytes = pickle.dumps(path_details)

        s_b64 = base64.b64encode(scenario_bytes).decode()
        p_b64 = base64.b64encode(path_bytes).decode()
        try:
            if self.ai:
                logger.info("Launching self_driver.py script as a subprocess to let AI implementation drive the scenarios.")
                process = subprocess.run(["python3", "self_driver.py", s_b64, p_b64, "-n{}".format(self.participant_name), "--sync", "-s{}".format(self.scenario_
            else:
                logger.info("Launching driver.py script as a subprocess to drive the generated scenario.")
                process = subprocess.run(["python3", "driver_keyboard.py", s_b64, p_b64, "-n{}" format(self.participant_name), "--sync", "-s{}" format(self.scena
                # process = subprocess.run(["python3", "driver_steeringwheel.py", s_b64, p_b64, "-n{}".format(self.participant_name), "--sync", "-s{}".format(self
            # Uncomment to print statemennts coming from driver.py to the terminal
            # print(process.stdout.decode(), process.stderr.decode())
        except Exception as e:
            logger.error("Could not launch driving script properly", exc_info=True)
            raise e

    # # MULTI-THREADING

    # # Launch the first person driving mode
    # def launch_driving_mode(self, scenario, path_details):
    #     try:
    #         flag = [False]

    #         threads = [
    #             Thread(target=simulate, args=[scenario, path_details, self.participant_name, self.scenario_name, self.vehicle_blueprint, self.ai, flag]),
    #             Thread(target=draw_lanes, args=[path_details["path_checkpoints"], self.participant_name]),
    #             Thread(target=draw_hero, args=[self.participant_name])
    #         ]

    #         for thread in threads:
    #             thread.start()

    #         while True:
    #             time.sleep(1)
    #             if flag[0]:
    #                 for thread in threads:
    #                     thread.join()
    #             break
    #     except Exception as e:
    #         logger.error("Something went wrong when starting the threads.", e)
```

```
189
190 def simulate(scenario, path_details, participant_name, scenario_name, vehicle_blueprint, ai, flag):
191     try:
192         scenario_bytes = pickle.dumps(scenario)
193         path_bytes = pickle.dumps(path_details)
194
195         s_b64 = base64.b64encode(scenario_bytes).decode()
196         p_b64 = base64.b64encode(path_bytes).decode()
197         if ai:
198             logger.info("Launching self_driver.py script as a subprocess to let AI implementation drive the scenarios.")
199             simulating = subprocess.Popen(["python3", "self_driver.py", s_b64, p_b64, "-n{}".format(participant_name), "--sync", "-s{}".format(scenario_name), "-
200         else:
201             logger.info("Launching driver.py script as a subprocess to drive the generated scenario.")
202             simulating = subprocess.Popen(["python3", "driver.py", s_b64, p_b64, "-n{}".format(participant_name), "--sync", "-s{}".format(scenario_name), "-f{}".
203
204     except Exception as e:
205         logger.error("Could not the simulation", e)
206
207 def draw_lanes(locations, participant_name):
208     try:
209         locations_bytes = pickle.dumps(locations)
210         locations_b64 = base64.b64encode(locations_bytes).decode()
211
212         log_path = "../../data/recordings/{}/session_logs.log".format(participant_name)
213
214         logger.info("Launching draw_lines.py script as a subprocess to mark the lanes and the driver on the map.")
215         drawing = subprocess.Popen(["python3", "helper_scripts/draw_lanes.py", locations_b64, "-l{}".format(log_path)])
216     except Exception as e:
217         logger.error("Could not start drawing lines", e)
218
219 def draw_hero(participant_name):
220     try:
221         log_path = "../../data/recordings/{}/session_logs.log".format(participant_name)
222
223         logger.info("Launching draw_hero.py script as a subprocess to mark the location of the participant on the map.")
224         drawing = subprocess.Popen(["python3", "helper_scripts/draw_hero.py", "-l{}".format(log_path)])
225     except Exception as e:
226         logger.error("Could not start hero vehicle", e)
```

**File: ./software/carla_scripts/scenarios/scenario_tester.py**

```
1  import glob
2  import os
3  import sys
4  import argparse
5  import argparse
6
7  try:
8      sys.path.append('..')
9      sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
10         sys.version_info.major,
11         sys.version_info.minor,
12         'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
13 except IndexError:
14     pass
15
16 import carla
17 from scenarios import ScenarioOne
18
19 argparser = argparse.ArgumentParser(description=__doc__)
20 argparser.add_argument(
21     '-n', '--name',
22     default='scenario1',
23     help='Specify the name of the scenario')
24 args = argparser.parse_args()
25
26 # When a new scenario needs to be simulated
27 # Its subclass has to be imported and used in this function to retrieve it in the main
28 def get_scenario(traffic_manager, sim_world):
29     if(args.name == "scenario1"):
30         return ScenarioOne(traffic_manager, sim_world)
31     else:
32         return ScenarioOne(traffic_manager, sim_world)
33
34 # Used to run and test if the scenario is working as expected
35 # Spawns all the actors, runs them and then destroys
36 def main():
37     try:
38         print("Beginning to simulate {}".format(args.name))
39         client = carla.Client('localhost', 2000)
40         client.set_timeout(20.0)
41         sim_world = client.get_world()
42         original_settings = sim_world.get_settings()
43         settings = sim_world.get_settings()
44         if not settings.synchronous_mode:
45             settings.synchronous_mode = True
46             settings.fixed_delta_seconds = 0.05
47         sim_world.apply_settings(settings)
48
49         traffic_manager = client.get_trafficmanager()
50         traffic_manager.set_synchronous_mode(True)
51         scenario = get_scenario(traffic_manager, sim_world)
52         scenario.spawn(client)
53         scenario.start()
54         while True:
55             sim_world.tick()
56     except KeyboardInterrupt:
57         print("Finished simulating the scenario.")
58     except:
59         print("Something wrong happened trying to simulate the scenario.")
60     finally:
61         scenario.finish(client)
62         sim_world.apply_settings(original_settings)
63
64 if __name__ == '__main__':
65     main()
```

**File: ./software/carla_scripts/scenarios/scenario_factory.py**

```
from scenarios.scenario import Scenario
import sys
import os
import glob
import logging

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

logger = logging.getLogger(__name__)


class ScenarioFactory():

    @staticmethod
    def make_scenario(parameters, path_details, seed):
        logger.info("Creating a scenario according to the generated path and scenario details")
        return Scenario(parameters, path_details, seed)

```

**File: ./software/carla_scripts/scenarios/change_weather.py**

```
import glob
import os
import sys
import time
import random
import argparse

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

import logging
logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO, filename="log.log", filemode="a", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '-n', '--name',
    default='scenario1',
    help='Specify the name of the scenario')
argparser.add_argument(
    '--log',
    action='store_true',
    help='Activate logging')
args = argparser.parse_args()

# Logging is used only when maps are changed during the simulations (logged to the participant's directory)
if args.log:
    logging.basicConfig(level=logging.INFO, filename='/recording/recordings/{}/session_logs.log'.format(args.name), filemode="a", format="%(asctime)s - %(levelnam
    logger = logging.getLogger("ChangeWeather")

# Log INFO message
def inform(message):
    logger.info(message) if args.log else print(message)

def main():
    try:
        inform("Changing the weather for {}.".format(args.name))
        if args.name == "scenario1":
            pass
        else:
            pass
        inform("Finished changing the weather")
    except KeyboardInterrupt:
        message = "Manually exited the weather changing script."
        logger.warn(message) if args.log else print(message)
    except:
        message = "Something wrong happened trying to change the weather. Aborting..."
        logger.exception(message) if args.log else print(message)


if __name__ == '__main__':
    main()

```

**File: ./software/carla_scripts/scenarios/change_map.py**

```python
import glob
import os
import sys
import time
import random
import argparse

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '-m', '--map',
    default='Town04',
    help='Specify the name of the map')
argparser.add_argument(
    '--name',
    default="ParticipantA",
    help='Name of the participant')
args = argparser.parse_args()


def main():
    try:
        print("Changing the map to {}.".format(args.map))
        client = carla.Client('localhost', 2000)
        world = client.load_world('{}'.format(args.map))
        time.sleep(10)
        print("Finished changing the map")
    except KeyboardInterrupt:
        message = "Manually exited the weather changing script."
        print(message)
    except Exception as e:
        message = "Something wrong happened trying to change the map. Aborting..."
        print(message)

if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/scenarios/scenario.py**

```python
import carla
import random
import numpy as np
import logging

logger= logging.getLogger(__name__)


class Scenario():

    def __init__(self, parameters, path_details, seed):
        self.scenario_parameters = parameters
        self.path_details = path_details
        # Used for reproducable randomizarion
        self.seed = seed

        self.vehicles_list = []
        self.walkers_list = []
        self.two_wheel_vehicles_list = []
        self.all_id = []
        self.all_actors = []

        self.traffic_manager = None
        self.synchronous_master = False
        self.world = None
        self.populated = False

        self.walker_speed = None

    def spawn(self, client):
        logger.info("Spawning the scenario actors")
        try:
            blueprints_vehicles_all = get_actor_blueprints(self.world, "vehicle.*", "All")

            blueprints_vehicles = get_safe_vehicle_blueprints(blueprints_vehicles_all)
            blueprints_walkers = get_actor_blueprints(self.world, "walker.pedestrian.*", "2")
            blueprints_two_wheel_vehicles = get_two_wheel_vehicle_blueprints(blueprints_vehicles_all)
        except Exception as e:
            logger.error("An error occured trying to get the blueprints", exc_info=True)
            raise e


        try:
            # Read all the parameters from the scenario object
            parameters = self.scenario_parameters
            number_of_pedestrians = int(parameters["number_of_pedestrians"])
            number_of_vehicles = int(parameters["number_of_vehicles"])
            number_of_two_wheel_vehicles = int(parameters["number_of_two_wheel_vehicles"])
            proportion_of_speeding_vehicles = float(parameters["proportion_of_speeding_vehicles"])
            proportion_of_vehicles_without_lights = float(parameters["proportion_of_vehicles_without_lights"])
            proportion_of_light_ignoring_vehicles = float(parameters["proportion_of_light_ignoring_vehicles"])
            light_ignoring_percent = float(parameters["light_ignoring_percent"])
            proportion_of_sign_ignoring_vehicles = float(parameters["proportion_of_sign_ignoring_vehicles"])
            sign_ignoring_percent = float(parameters["sign_ignoring_percent"])
            proportion_of_vehicle_ignoring_vehicles = float(parameters["proportion_of_vehicle_ignoring_vehicles"])
            vehicle_ignoring_percent = float(parameters["vehicle_ignoring_percent"])
            proportion_of_walker_ignoring_vehicles = float(parameters["proportion_of_walker_ignoring_vehicles"])
            walker_ignoring_percent = float(parameters["walker_ignoring_percent"])
            proportion_of_keeping_right_vehicles = float(parameters["proportion_of_keeping_right_vehicles"])
            keeping_right_percent = float(parameters["keeping_right_percent"])
            proportion_of_lane_changing_vehicles = float(parameters["proportion_of_lane_changing_vehicles"])
            lane_change_percent = float(parameters["lane_change_percent"])
            proportion_of_misbehaving_pedestrians = float(parameters["proportion_of_misbehaving_pedestrians"])
            proportion_of_running_pedestrians = float(parameters["proportion_of_running_pedestrians"])
            proportion_of_road_crossing_pedestrians = float(parameters["proportion_of_road_crossing_pedestrians"])
        except Exception as e:
            logger.error("An error occured trying to get retrieve scenario parameters", exc_info=True)
            raise e


        # ALL NECCESSARY PARAMETERS WERE EXTRACTED

        try:
            SpawnActor = carla.command.SpawnActor
            SetAutopilot = carla.command.SetAutopilot
            FutureActor = carla.command.FutureActor

```

```
            start = self.path_details["start_location"]
            hero_spawn_location = carla.Location(x=float(start["x"]), y=float(start["y"]), z=float(start["z"]))

            # Removes ones too close to the player's spawn location
            spawn_points = self.world.get_map().get_spawn_points()
            spawn_points = filter_spawn_points(spawn_points, hero_spawn_location, 5)
            number_of_spawn_points = len(spawn_points)

            total_number_of_vehicles = (number_of_vehicles + number_of_two_wheel_vehicles)
            if total_number_of_vehicles <= number_of_spawn_points:
                random.shuffle(spawn_points)

            else:
                logger.warning("There are too many vehicles for too few spawn points. Reducing the number of vehicles. There are {} spawn points for vehicles in t
                difference = total_number_of_vehicles - number_of_spawn_points
                if (number_of_vehicles - difference) >= 0:
                    number_of_vehicles = (number_of_vehicles - difference)
                else:
                    # Over is going to be negative
                    over = (number_of_vehicles - difference)
                    number_of_two_wheel_vehicles = number_of_two_wheel_vehicles + over

            total_number_of_vehicles = (number_of_vehicles + number_of_two_wheel_vehicles)
            vehicle_spawn_points, two_wheel_vehicle_spawn_points = divide_list(spawn_points, number_of_vehicles, number_of_two_wheel_vehicles)

            # Make all vehicles aim for the speed limit
            self.traffic_manager.global_percentage_speed_difference(0)
        except Exception as e:
            logger.error("An error occured trying to get the spawn points", exc_info=True)
            raise e

        try:
            # --------------
            # Spawn vehicles
            # --------------
            batch = []
            for n, transform in enumerate(vehicle_spawn_points):
                if n >= number_of_vehicles:
                    break
                blueprint = random.choice(blueprints_vehicles)
                if blueprint.has_attribute('color'):
                    color = random.choice(blueprint.get_attribute('color').recommended_values)
                    blueprint.set_attribute('color', color)
                if blueprint.has_attribute('driver_id'):
                    driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
                    blueprint.set_attribute('driver_id', driver_id)
                blueprint.set_attribute('role_name', 'autopilot')
                # Spawn the cars and set their autopilot and light state all together
                batch.append(SpawnActor(blueprint, transform)
                    .then(SetAutopilot(FutureActor, False, self.traffic_manager.get_port())))

            for response in client.apply_batch_sync(batch, self.synchronous_master):
                if response.error:
                    logger.error(response.error)
                else:
                    self.vehicles_list.append(response.actor_id)

            logger.info("The {}/{} vehicles were successfully spawned".format(len(self.vehicles_list), number_of_vehicles))

        except Exception as e:
            logger.error("Error creating vehicles on the map.", exc_info=True)
            raise e

        try:
            # --------------
            # Spawn two-wheel vehicles
            # --------------
            batch = []
            for n, transform in enumerate(two_wheel_vehicle_spawn_points):
                if n >= number_of_two_wheel_vehicles:
                    break
                blueprint = random.choice(blueprints_two_wheel_vehicles)
                if blueprint.has_attribute('color'):
                    color = random.choice(blueprint.get_attribute('color').recommended_values)
                    blueprint.set_attribute('color', color)
                if blueprint.has_attribute('driver_id'):
                    driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
                    blueprint.set_attribute('driver_id', driver_id)
                blueprint.set_attribute('role_name', 'autopilot')
                # Spawn the cars and set their autopilot and light state all together
                batch.append(SpawnActor(blueprint, transform)
                    .then(SetAutopilot(FutureActor, False, self.traffic_manager.get_port())))

            for response in client.apply_batch_sync(batch, self.synchronous_master):
                if response.error:
                    logger.error(response.error)
                else:
                    self.two_wheel_vehicles_list.append(response.actor_id)

            logger.info("The {}/{} two-wheel vehicles were successfully spawned".format(len(self.two_wheel_vehicles_list), number_of_two_wheel_vehicles))


        except Exception as e:
            logger.error("Error creating two-wheel vehicles on the map.", exc_info=True)
            raise e

        try:
            logger.info("Setting the vehicle behavior according to the scenario parameters")
            ### SETUP VEHICLE BEHAVIOUR

            all_vehicles = self.vehicles_list + self.two_wheel_vehicles_list
            amount_vehicles = len(all_vehicles)
            all_vehicle_actors = self.world.get_actors(all_vehicles)


            # Distance
            for actor in all_vehicle_actors:
                # Select a distance to keep between 1 and 5 metres randomly.
                diff = random.randint(1, 5)
                self.traffic_manager.auto_lane_change(actor, True)
                self.traffic_manager.distance_to_leading_vehicle(actor, diff)

            # Set not update vehicle lights for certain vehicles
            amount_of = int(np.round(amount_vehicles * proportion_of_vehicles_without_lights))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.update_vehicle_lights(actor, False)

            # Update the lights for the rest
            for v in all_vehicles:
                if(v not in random_vehicles):
                    actor = self.world.get_actor(v)
                    self.traffic_manager.update_vehicle_lights(actor, True)


            # Speeding
            amount_of = int(np.round(amount_vehicles * proportion_of_speeding_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
```

```
                actor = self.world.get_actor(v)
                # Select a speeding increase randomly
                diff = random.randint(-30, -1)
                self.traffic_manager.vehicle_percentage_speed_difference(actor, diff)

            # Light ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_light_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_lights_percentage(actor, light_ignoring_percent)

            # Sign ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_sign_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_signs_percentage(actor, sign_ignoring_percent)

            # Vehicle ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_vehicle_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_vehicles_percentage(actor, vehicle_ignoring_percent)

            # Walker ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_walker_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_walkers_percentage(actor, walker_ignoring_percent)

            # Keep right rule
            amount_of = int(np.round(amount_vehicles * proportion_of_keeping_right_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.keep_right_rule_percentage(actor, keeping_right_percent)

            # Lane changing left
            amount_of = int(np.round(amount_vehicles * proportion_of_lane_changing_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.random_left_lanechange_percentage(actor, lane_change_percent)

            # Lane changing right
            amount_of = int(np.round(amount_vehicles * proportion_of_lane_changing_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.random_right_lanechange_percentage(actor, lane_change_percent)

            logger.info("The behaviour of vehicles was successfully set")

        except Exception as e:
            logger.error("Error setting the behaviour of vehicles.", exc_info=True)
            raise e


        try:
            # -------------
            # Spawn Walkers
            # -------------
            # Random locations to spawn

            number_of_pedestrians_initial = number_of_pedestrians

            try:
                spawn_points = []
                for i in range(number_of_pedestrians):
                    spawn_point = carla.Transform()
                    loc = self.world.get_random_location_from_navigation()
                    if (loc != None):
                        spawn_point.location = loc
                        spawn_points.append(spawn_point)

                if(len(spawn_points) >= number_of_pedestrians):
                    logger.debug("There is a required amount of spawn points for pedestrians on the map.")
                else:
                    number_of_pedestrians = len(spawn_points)
            except Exception as e:
                logger.error(e)

            # Spawn walker objects
            walker_speed = []
            walker_speed2 = []
            logger.debug("Need to spawn {} pedestrians".format(number_of_pedestrians))
            for i in range(number_of_pedestrians):

                # Modify the blueprint
                walker_bp = random.choice(blueprints_walkers)
                # set as not invincible
                if walker_bp.has_attribute('is_invincible'):
                    walker_bp.set_attribute('is_invincible', 'false')
                # set the max speed
                if walker_bp.has_attribute('speed'):
                    if (random.random() > proportion_of_running_pedestrians):
                        # walking
                        walker_speed.append(walker_bp.get_attribute('speed').recommended_values[1])
                    else:
                        # running
                        walker_speed.append(walker_bp.get_attribute('speed').recommended_values[2])
                else:
                    walker_speed.append(0.0)

                logger.debug("The blueprint was set.")
                try:
                    # Try to spawn the pedestrian 10000 times and
                    y = 0
                    not_spawned = True
                    while(not_spawned and y < 10000):
                        spawn_point = carla.Transform()
                        loc = self.world.get_random_location_from_navigation()
                        if (loc != None):
                            spawn_point.location = loc
                            batch = []
                            batch.append(SpawnActor(walker_bp, spawn_point))
                            results = client.apply_batch_sync(batch, True)
                            if results[0].error:
                                logger.debug(results[0].error)
                                y += 1
                            else:
                                not_spawned = False
                                self.walkers_list.append({"id": results[0].actor_id})
                                walker_speed2.append(walker_speed[i])
                                break
                        else:
                            y += 1
```

```
                except Exception as e:
                    logger.error(e)
                logger.debug("There were {} respawns for this walker".format(y))


            logger.info("Successfully spawned {}/{} walkers.".format(len(self.walkers_list), number_of_pedestrians_initial))
            self.walker_speed = walker_speed2

            # Spawn the walker controller
            batch = []
            walker_controller_bp = self.world.get_blueprint_library().find('controller.ai.walker')
            for i in range(len(self.walkers_list)):
                batch.append(SpawnActor(walker_controller_bp, carla.Transform(), self.walkers_list[i]["id"]))
            results = client.apply_batch_sync(batch, True)
            for i in range(len(results)):
                if results[i].error:
                    logger.error(results[i].error)
                else:
                    self.walkers_list[i]["con"] = results[i].actor_id

            # Put together the walkers and controllers id to get the objects from their id
            for i in range(len(self.walkers_list)):
                self.all_id.append(self.walkers_list[i]["con"])
                self.all_id.append(self.walkers_list[i]["id"])
            self.all_actors = self.world.get_actors(self.all_id)

        except Exception as e:
            logger.error("Error creating walkers on the map.", exc_info=True)
            raise e

        self.world.tick()
        self.world.set_pedestrians_cross_factor(proportion_of_road_crossing_pedestrians)
        self.populated = True

    # Setup the traffic manager and other parameters before the simulation
    def setup(self, traffic_manager, world):
        logger.info("Setting up the scenario")
        self.traffic_manager = traffic_manager
        self.world = world
        self.traffic_manager.set_global_distance_to_leading_vehicle(2.5)
        self.traffic_manager.set_random_device_seed(self.seed)
        self.world.set_pedestrians_seed(self.seed)
        random.seed(self.seed)
        self.traffic_manager.set_hybrid_physics_mode(True)
        self.traffic_manager.set_hybrid_physics_radius(70.0)

        settings = self.world.get_settings()
        self.traffic_manager.set_synchronous_mode(True)
        if not settings.synchronous_mode:
            self.synchronous_master = True
            settings.synchronous_mode = True
            settings.fixed_delta_seconds = 0.05
        else:
            self.synchronous_master = False

    # Unfreeze all actors and let them move as specified
    def start(self):
        logger.info("Unfreezing all actors in the simulation.")
        port = self.traffic_manager.get_port()

        # Unfreeze vehicles
        for v in self.vehicles_list:
            actor = self.world.get_actor(v)
            actor.set_autopilot(True, port)

        # Unfreeze two-wheel vehicles
        for b in self.two_wheel_vehicles_list:
            actor = self.world.get_actor(b)
            actor.set_autopilot(True, port)

        # Unfreeze walkers
        for i in range(0, len(self.all_id), 2):
            self.all_actors[i].start()
            self.all_actors[i].go_to_location(self.world.get_random_location_from_navigation())
            self.all_actors[i].set_max_speed(float(self.walker_speed[int(i/2)]))

    # Change settings back to default and destroy all actors
    def finish(self, client):
        logger.info("Finishing the scenario")
        if self.synchronous_master:
            settings = self.world.get_settings()
            settings.synchronous_mode = False
            settings.no_rendering_mode = False
            settings.fixed_delta_seconds = None
            self.world.apply_settings(settings)

        logger.info('Destroying {} vehicles'.format(len(self.vehicles_list)))
        client.apply_batch([carla.command.DestroyActor(x) for x in self.vehicles_list])

        logger.info('Destroying {} two-wheel vehicles'.format(len(self.two_wheel_vehicles_list)))
        client.apply_batch([carla.command.DestroyActor(x) for x in self.two_wheel_vehicles_list])
        # Stop walker controllers (list is [controller, actor, controller, actor ...])
        for i in range(0, len(self.all_id), 2):
            self.all_actors[i].stop()
        logger.info('Destroying {} walkers'.format(len(self.walkers_list)))
        client.apply_batch([carla.command.DestroyActor(x) for x in self.all_id])

        self.populated = False


def get_actor_blueprints(world, filter, generation):
    bps = world.get_blueprint_library().filter(filter)
    if generation.lower() == "all":
        return bps

    # If the filter returns only one bp, we assume that this one needed
    # and therefore, we ignore the generation
    if len(bps) == 1:
        return bps
    try:
        int_generation = int(generation)
        # Check if generation is in available generations
        if int_generation in [1, 2]:
            bps = [x for x in bps if int(x.get_attribute('generation')) == int_generation]
            return bps
        else:
            logger.warning("Actor Generation is not valid. No actor will be spawned.")
            return []
    except:
        logger.warning("Actor Generation is not valid. No actor will be spawned.")
        return []

def get_safe_vehicle_blueprints(blueprints):
    blueprints = [x for x in blueprints if int(x.get_attribute('number_of_wheels')) == 4]
    blueprints = [x for x in blueprints if not x.id.endswith('microlino')]
    blueprints = [x for x in blueprints if not x.id.endswith('carlacola')]
    blueprints = [x for x in blueprints if not x.id.endswith('cybertruck')]
    blueprints = [x for x in blueprints if not x.id.endswith('t2')]
    blueprints = [x for x in blueprints if not x.id.endswith('sprinter')]
    blueprints = [x for x in blueprints if not x.id.endswith('firetruck')]
```

```
    blueprints = [x for x in blueprints if not x.id.endswith('ambulance')]
    blueprints = sorted(blueprints, key=lambda bp: bp.id)
    return blueprints

def get_two_wheel_vehicle_blueprints(blueprints):
    return [x for x in blueprints if int(x.get_attribute('number_of_wheels')) == 2]

def distanceBetweenTwoLocations(start, finish):
    first = np.array((start.x, start.y))
    second = np.array((finish.x, finish.y))
    return np.linalg.norm(first-second)

def filter_spawn_points(spawn_points, hero_spawn_location, min_distance):
    try:
        filtered_spawn_points = []
        for point in spawn_points:
            distance = distanceBetweenTwoLocations(point.location, hero_spawn_location)
            # If the spawn location is further than two metres from the player's location, add to the filtered ones
            if distance > min_distance:
                filtered_spawn_points.append(point)
        return filtered_spawn_points
    except Exception as e:
        logger.error(e)

def divide_list(input_list, x, y):
    first_list = input_list[:x]
    second_list = input_list[x:x+y]
    return first_list, second_list
```

**File: ./software/carla_scripts/scenarios/path_builder.py**

```
import logging
import xml.etree.ElementTree as ET
import os
import random
import json

logger = logging.getLogger(__name__)
# How many times to try and recreate the path if it keeps on failing or reaching a dead-end before
MAX_SEARCH_LENGTH = 10000
path_to_save_path = "../../data/paths/"

class PathBuilder():

    @staticmethod
    # Used to find the path, the map and the start location automatically
    def generate_details(scenario_data, save_filename = None):
        try:
            map, road_elements = PathBuilder.find_map_and_path(scenario_data)
        except Exception as e:
            logger.exception("Something went wrong generating the path", e)
            raise Exception
        try:
            start_location = PathBuilder.find_start_location(road_elements[0])
            path_details = PathBuilder.form_json(map, road_elements, start_location, save_filename)
            logger.info(path_details)
            if save_filename:
                logger.info("Path was saved to file {}.json in paths directory amongst other data".format(save_filename))
            return path_details
        except Exception as e:
            logger.exception("Something went wrong combining the path object", e)
            raise Exception

    @staticmethod
    # Used to return the newly generated path and everything in a json-like python structure
    # And save it, if the save filename is specified
    def form_json(map, road_elements, start_location, save_filename):
        # Build a json-like file
        # Save if needed and return
        locations = []
        for i in range(len(road_elements)):
            if(i == len(road_elements)-1):
                locations.append(PathBuilder.get_road_location_object(road_elements[i], finish_road=True))
            else:
                locations.append(PathBuilder.get_road_location_object(road_elements[i]))
        attributes = {
            "path_checkpoints": locations,
            "start_location": start_location,
            "town": map.replace(".xodr", "")
        }
        if save_filename:
            with open("{}{}.json".format(path_to_save_path, save_filename), "w+") as f:
                json.dump(attributes, f, indent=4)
        return attributes

    @staticmethod
    # Used to find and return a start location among the given path_checkpoints
    def find_start_location(road_element):
        return PathBuilder.get_road_location_object(road_element, yaw=True)


    @staticmethod
    # Extract the location from the road object
    def get_road_location_object(road, finish_road = None, yaw = None):
        geometries = road.find("planView").findall("geometry")
        geometry = geometries[len(geometries)-1] if finish_road else geometries[0]
        x = float(geometry.attrib["x"])
        y = float(geometry.attrib["y"])
        rotation = float(geometry.attrib["hdg"])
        # Determine z according to the height in that place
        # yaw determine from the road_element
        z = 0.4
        start_location = {
            "x": x,
            "y": y,
            "z": z,
            "yaw": rotation
        }
        return start_location

    @staticmethod
    # Used to find the appropriate city
    def find_map_and_path(parameters):
        # Look for an appropriate map iterating through all files in maps/opendrive_format drirectory
        logger.info("Looking for a map that could be used in the scenario and that satisfies the scenario parameters.")
        map_filenames = os.listdir("../../data/maps/opendrive_format")
        already_considered_maps = []
        for i in range(len(map_filenames)):
            # Loop through all mapfiles randomly and if the match is found, return
            unconsidered_maps = [file for file in map_filenames if file not in already_considered_maps and file.endswith(".xodr")]
            random_map = random.sample(unconsidered_maps, 1)[0]
            processed_map = PathBuilder.process_map(random_map, parameters)
            if processed_map:
                return random_map, processed_map
            else:
                logger.warning("The algorithm was unable to find a path given the scenario parameters in the map {}".format(random_map))
```

```
                already_considered_maps.append(random_map)
                if i == (len(map_filenames)-1):
                    logger.warning("The search was exhausted and no valid path was found for the given scenario and all the map files. RETURNING NONE.")
        # If this place was reached, it is impossible to create a map
        return None

    @staticmethod
    # Process the map file to check if it is appropriate for this scenario
    def process_map(filename, parameters):
        logger.info("Checking if {} is a valid map for this scenario".format(filename))
        path_to_file = "../../data/maps/opendrive_format/{}".format(filename)
        tree = ET.parse(path_to_file)
        root = tree.getroot()

        # Get all road and junction elements from the XML
        roads = []
        junctions = []
        for road in root.findall("./road"):
            roads.append(road)
        for junction in root.findall("./junction"):
            junctions.append(junction)

        verified = PathBuilder.check_if_parameters_are_satisfied(parameters, roads, junctions, filename)
        if not verified: return None

        ## MAKE PATH
        #print("Working on map: {}".format(filename))
        logger.info("Working on map: {}".format(filename))
        return PathBuilder.make_path(roads, junctions, parameters)


    @staticmethod
    # Checks if the scenario parameters are satisfied in the map
    def check_if_parameters_are_satisfied(parameters, roads, junctions, filename):
        # Check if there are enough junctions in the map
        if len(junctions) >= parameters["number_of_junctions"]:
            logger.info("Junction parameter is satisfied")
        else:
            logger.warning("There are not enough junctions in {} to accomodate the scenario.".format(filename))
            return None

        # Check if there is distance requirement is satisfied
        total_road_length = 0.0
        for road in roads:
            total_road_length += float(road.attrib["length"])
        if total_road_length >= float(parameters["distance_in_metres"]):
            logger.info("Distance parameter is satisfied")
        else:
            logger.warning("There is not enough unique drivable road to accomodate the scenario {}".format(filename))
            return None
        return True

    @staticmethod
    # Used to run iteration and try to create a path from the given road and junction objects
    def make_path(roads, junctions, parameters):
        go = True
        i = 0
        path = None
        # Try to create a path for a MAX_SEARCH_LENGTH times
        while(go and i < MAX_SEARCH_LENGTH):
            temp = None
            try:
                temp = PathBuilder.get_path(roads, junctions, parameters["number_of_junctions"], parameters["distance_in_metres"])
            except Exception as e:
                logger.warning("Something went weong trying to create the path. Retrying.")
            if temp:
                logger.info("The path was successfully generated")
                go = False
                path = temp
            i += 1
        if not path:
            logger.warning("No success in {} tries to create a path".format(MAX_SEARCH_LENGTH))
            # print("No success in {} tries...".format(i))
        return path

    @staticmethod
    # Used to generate a random path over the map
    def get_path(roads, junctions, max_number_of_junctions_allowed, min_distance):
        path = []
        chosen_road_queue = []
        current_road = random.choice(roads)
        path.append(current_road)
        id, length, predecessor, successor = PathBuilder.analyse_road_element(current_road)
        chosen_road_queue.append(id)
        # print("Id of the first one: {}".format(id))
        current_junctions = 0
        current_length = length
        while current_length < min_distance and current_junctions < max_number_of_junctions_allowed:
            if len(chosen_road_queue) > 7 and chosen_road_queue[-8:].count(id) >= 3:
                # Check if the same road id appeared among the last 8 roads chosen
                # It means that the algorithm got stuck and now is moving back and forth
                return None
            # print("Current length: {}".format(current_length))
            # print("Already used roads: {}".format(chosen_road_queue))
            if successor[0] == "junction":
                # print("Junction approached. Its id is: {}".format(successor[1]))
                current_junctions += 1
                junction = PathBuilder.get_the_right_junction(junctions, successor[1])
                all_exits = PathBuilder.analyse_junction_and_get_possible_exits(junction, id, chosen_road_queue)
                # If a dead-end was reached (no more exits)
                if len(all_exits) == 0:
                    # Return a failure, because a dead-end was reached
                    return None
                # print("All possible exits from this are: {}".format(all_exits))
                random_exit = random.choice(all_exits)
                new_road_to_take = PathBuilder.get_the_right_road(roads, random_exit)
                id, length, predecessor, successor = PathBuilder.analyse_road_element(new_road_to_take)
                # print("The chosen way is: {}".format(id))
                current_length += length
                chosen_road_queue.append(id)
            else:
                new_road_to_take = PathBuilder.get_the_right_road(roads, successor[1])
                id, length, predecessor, successor = PathBuilder.analyse_road_element(new_road_to_take)
                # print("The chosen way is: {}".format(id))
                current_length += length
                chosen_road_queue.append(id)
                path.append(new_road_to_take)
            # print("\n")

        return path


    @staticmethod
    # Used to parse the xml road element and retrieve useful data
    def analyse_road_element(road):
        id = road.attrib["id"]
        length = float(road.attrib["length"])
        link = road.findall("link")[0]
        predecessor = (link.find("predecessor").attrib["elementType"], link.find("predecessor").attrib["elementId"])
        successor = (link.find("successor").attrib["elementType"], link.find("successor").attrib["elementId"])
```

```
        return id, length, predecessor, successor

    @staticmethod
    # Used to
    def analyse_junction_and_get_possible_exits(junction, incoming_road_id, already_used_ids):
        # First try to get new roads to drive on
        possible_next_roads = []
        id = junction.attrib["id"]
        for connection in junction.findall("connection"):
            if connection.attrib["incomingRoad"] == incoming_road_id and connection.attrib["connectingRoad"] not in already_used_ids:
                possible_next_roads.append(connection.attrib["connectingRoad"])

        # If there are no unique roads, allow picking of old ones
        if len(possible_next_roads) == 0:
            for connection in junction.findall("connection"):
                if connection.attrib["incomingRoad"] == incoming_road_id:
                    possible_next_roads.append(connection.attrib["connectingRoad"])
        # print("Possible next moves: ".format(possible_next_roads))
        return possible_next_roads

    @staticmethod
    # Used to find the right junction object from all junctions using its id
    def get_the_right_junction(junctions, id):
        for junction in junctions:
            if junction.attrib["id"] == id:
                return junction
        return None

    @staticmethod
    # Used to find the right road object from all road objects using its id
    def get_the_right_road(roads, id):
        for road in roads:
            if road.attrib["id"] == id:
                return road
        return None
```

**File: ./software/carla_scripts/clean.sh**

```
#!/bin/bash
find . -name __pycache__ -type d -exec rm -rf {} +

find . -name ".*" -type f -delete

find . -name "*.sh" -exec chmod +x {} \;
```

**File: ./software/carla_scripts/wheel_config.ini**

```
[G29 Racing Wheel]
steering_wheel = 0
throttle = 2
brake = 3
reverse = 6
handbrake = 4
```

**File: ./software/carla_scripts/driver_keyboard.py**

```
#!/usr/bin/env python

# Copyright (c) 2019 Computer Vision Center (CVC) at the Universitat Autonoma de
# Barcelona (UAB).
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

# Allows controlling a vehicle with a keyboard. For a simpler and more
# documented example, please take a look at tutorial.py.
"""
Welcome to CARLA manual control.

Use ARROWS or WASD keys for control.

    W            : throttle
    S            : brake
    A/D          : steer left/right
    Q            : toggle reverse
    Space        : hand-brake
    P            : toggle autopilot
    M            : toggle manual transmission
    ,/.          : gear up/down

    L            : toggle next light type
    SHIFT + L    : toggle high beam
    Z/X          : toggle right/left blinker
    I            : toggle interior light

    TAB          : change camera position

    ENTER        : start route

    F1           : toggle HUD
    H/?          : toggle help
    ESC          : quit
"""

from __future__ import print_function



# ==============================================================================
# -- find carla module ---------------------------------------------------------
# ==============================================================================


import glob
import os
import sys
import pickle
from scenarios.scenario import Scenario
import base64

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass
```

```
63
64
65  # ==============================================================================
66  # -- imports -------------------------------------------------------------------
67  # ==============================================================================
68
69
70  import carla
71
72  from carla import ColorConverter as cc
73
74  import argparse
75  import collections
76  import datetime
77  import logging
78  import math
79  import random
80  import re
81  import weakref
82  from recording.manager import Manager
83  from config import FPS
84
85  try:
86      import pygame
87      from pygame import KMOD_CTRL
88      from pygame import KMOD_SHIFT
89      from pygame import K_0
90      from pygame import K_9
91      from pygame import K_BACKQUOTE
92      from pygame import K_BACKSPACE
93      from pygame import K_DOWN
94      from pygame import K_ESCAPE
95      from pygame import K_COMMA
96      from pygame import K_F1
97      from pygame import K_LEFT
98      from pygame import K_PERIOD
99      from pygame import K_RIGHT
100     from pygame import K_SLASH
101     from pygame import K_SPACE
102     from pygame import K_TAB
103     from pygame import K_UP
104     from pygame import K_a
105     from pygame import K_b
106     from pygame import K_c
107     from pygame import K_d
108     from pygame import K_g
109     from pygame import K_h
110     from pygame import K_i
111
112     from pygame import K_l
113     from pygame import K_m
114     from pygame import K_n
115     from pygame import K_o
116     from pygame import K_p
117     from pygame import K_q
118     from pygame import K_r
119     from pygame import K_s
120     from pygame import K_t
121     from pygame import K_v
122     from pygame import K_w
123     from pygame import K_x
124     from pygame import K_z
125     from pygame import K_MINUS
126     from pygame import K_EQUALS
127 except ImportError:
128     raise RuntimeError('cannot import pygame, make sure pygame package is installed')
129
130 try:
131     import numpy as np
132 except ImportError:
133     raise RuntimeError('cannot import numpy, make sure numpy package is installed')
134
135
136 # ==============================================================================
137 # -- Global functions ----------------------------------------------------------
138 # ==============================================================================
139
140
141 def find_weather_presets():
142     rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')
143     name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))
144     presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
145     return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]
146
147
148 def get_actor_display_name(actor, truncate=250):
149     name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
150     return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name
151
152 def get_actor_blueprints(world, filter, generation):
153     bps = world.get_blueprint_library().filter(filter)
154     if generation.lower() == "all":
155         return bps
156
157     # If the filter returns only one bp, we assume that this one needed
158     # and therefore, we ignore the generation
159     if len(bps) == 1:
160         return bps
161
162     try:
163         int_generation = int(generation)
164         # Check if generation is in available generations
165         if int_generation in [1, 2]:
166             bps = [x for x in bps if int(x.get_attribute('generation')) == int_generation]
167             return bps
168         else:
169             print("   Warning! Actor Generation is not valid. No actor will be spawned.")
170             return []
171     except:
172         print("   Warning! Actor Generation is not valid. No actor will be spawned.")
173         return []
174
175
176 # ==============================================================================
177 # -- World ---------------------------------------------------------------------
178 # ==============================================================================
179
180
181 class World(object):
182     def __init__(self, carla_world, hud, args, scenario, client, path_details):
183         self.world = carla_world
184         self.sync = args.sync
185         self.actor_role_name = args.rolename
186         self.path_details = path_details
187         self.scenario = scenario
188         self.client = client
189         self.scenario_name = args.scenario
190         self.participant_name = args.name
191         try:
192             self.map = self.world.get_map()
```

```
        except RuntimeError as error:
            print('RuntimeError: {}'.format(error))
            print('  The server could not send the OpenDRIVE (.xodr) file:')
            print('  Make sure it exists, has the same name of your town, and is correct.')
            sys.exit(1)
        self.spawn_point = None
        self.init_spawn_point(path_details["start_location"])
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.imu_sensor = None
        self.radar_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = args.filter
        self._actor_generation = args.generation
        self._gamma = args.gamma
        self.restart(args)
        self.recording_enabled = 0

        # All about the simulation
        self.simulation_clock = None
        self.simulation_start_tick = 0
        self.simulation_time = 0
        self.manager = None

        self.recording_start = 0
        self.constant_velocity_enabled = False
        self.show_vehicle_telemetry = False
        self.doors_are_open = False
        self.current_map_layer = 0
        self.map_layer_names = [
            carla.MapLayer.NONE,
            carla.MapLayer.Buildings,
            carla.MapLayer.Decals,
            carla.MapLayer.Foliage,
            carla.MapLayer.Ground,
            carla.MapLayer.ParkedVehicles,
            carla.MapLayer.Particles,
            carla.MapLayer.Props,
            carla.MapLayer.StreetLights,
            carla.MapLayer.Walls,
            carla.MapLayer.All
        ]

    def init_spawn_point(self, start_location):
        try:
            self.spawn_point = carla.Transform(carla.Location(x=start_location["x"], y=start_location["y"], z=start_location["z"]), carla.Rotation(pitch=0.0, ya
        except Exception as e:
            logger.error("Could not initiate spawn location for the driver", e)
    def restart(self, args):
        self.player_max_speed = 1.589
        self.player_max_speed_fast = 3.713
        # Keep same camera config if the camera manager exists.
        cam_index = self.camera_manager.index if self.camera_manager is not None else 0
        cam_pos_index = self.camera_manager.transform_index if self.camera_manager is not None else 0
        # Get a random blueprint.
        blueprint = random.choice(get_actor_blueprints(self.world, self._actor_filter, self._actor_generation))
        blueprint.set_attribute('role_name', self.actor_role_name)
        if blueprint.has_attribute('color'):
            blueprint.set_attribute('color', '{},{},{}'.format(args.red, args.green, args.blue))
        if blueprint.has_attribute('driver_id'):
            driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
            blueprint.set_attribute('driver_id', driver_id)
        if blueprint.has_attribute('is_invincible'):
            blueprint.set_attribute('is_invincible', 'true')
        # set the max speed
        if blueprint.has_attribute('speed'):
            self.player_max_speed = float(blueprint.get_attribute('speed').recommended_values[1])
            self.player_max_speed_fast = float(blueprint.get_attribute('speed').recommended_values[2])

        # Spawn the player.
        if self.player is not None:
            spawn_point = self.player.get_transform()
            spawn_point.location.z += 2.0
            spawn_point.rotation.roll = 0.0
            spawn_point.rotation.pitch = 0.0
            self.destroy()
            self.player = self.world.try_spawn_actor(blueprint, spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        while self.player is None:
            if not self.map.get_spawn_points():
                print('There are no spawn points available in your map/town.')
                print('Please add some Vehicle Spawn Point to your UE4 scene.')
                sys.exit(1)
            if self.spawn_point is None:
                spawn_points = self.map.get_spawn_points()
                self.spawn_point = random.choice(spawn_points) if spawn_points else carla.Transform()
            self.player = self.world.try_spawn_actor(blueprint, self.spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        # Set up the sensors.
        self.collision_sensor = CollisionSensor(self.player, self.hud)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.hud)
        self.gnss_sensor = GnssSensor(self.player)
        self.imu_sensor = IMUSensor(self.player)
        self.camera_manager = CameraManager(self.player, self.hud, self._gamma)
        self.camera_manager.transform_index = cam_pos_index
        self.camera_manager.set_sensor(cam_index, notify=False)
        actor_type = get_actor_display_name(self.player)
        self.hud.notification(actor_type)

        if self.sync:
            self.world.tick()
        else:
            self.world.wait_for_tick()

    def toggle_radar(self):
        if self.radar_sensor is None:
            self.radar_sensor = RadarSensor(self.player)
        elif self.radar_sensor.sensor is not None:
            self.radar_sensor.sensor.destroy()
            self.radar_sensor = None

    def modify_vehicle_physics(self, actor):
        #If actor is not a vehicle, we cannot use the physics control
        try:
            physics_control = actor.get_physics_control()
            physics_control.use_sweep_wheel_collision = True
            actor.apply_physics_control(physics_control)
        except Exception:
            pass

    def startScenario(self):
        if self.simulation_clock is None:
            logger.info("Starting the scenario")
```

```
            self.scenario.start()
            self.hud.notification('The simulation has started')
            self.manager = Manager(self.path_details["path_checkpoints"], self.world, self.player, self.participant_name, self.scenario_name)
            #self.collision_sensor = self.manager.collision_sensor
            self.simulation_clock = pygame.time.Clock()
            self.simulation_start_tick = pygame.time.get_ticks()

    def finishScenario(self):
        logger.warning("Finish was NOT REACHED. Terminating the simulation.")
        self.manager.shut_down(False)
        self.scenario.finish(self.client)

    def tick(self):
        if self.simulation_clock is not None:
            self.simulation_clock.tick(60)
            self.simulation_time += self.simulation_clock.get_time()
            if self.manager.record(self.simulation_time):
                return True
            self.hud.tick(self, self.simulation_time/1000)
        else:
            self.hud.tick(self, 0)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy_sensors(self):
        self.camera_manager.sensor.destroy()
        self.camera_manager.sensor = None
        self.camera_manager.index = None

    def destroy(self):
        if self.radar_sensor is not None:
            self.toggle_radar()
        sensors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor,
            self.imu_sensor.sensor]
        for sensor in sensors:
            if sensor is not None:
                sensor.stop()
                sensor.destroy()
        if self.player is not None:
            self.player.destroy()


# ==============================================================================
# -- KeyboardControl -----------------------------------------------------------
# ==============================================================================


class KeyboardControl(object):
    """Class that handles keyboard input."""
    def __init__(self, world, start_in_autopilot):
        self._autopilot_enabled = start_in_autopilot
        if isinstance(world.player, carla.Vehicle):
            self._control = carla.VehicleControl()
            self._lights = carla.VehicleLightState.NONE
            world.player.set_autopilot(self._autopilot_enabled)
            world.player.set_light_state(self._lights)
        elif isinstance(world.player, carla.Walker):
            self._control = carla.WalkerControl()
            self._autopilot_enabled = False
            self._rotation = world.player.get_transform().rotation
        else:
            raise NotImplementedError("Actor type not supported")
        self._steer_cache = 0.0
        world.hud.notification("Press 'H' or '?' for help.", seconds=4.0)

    def parse_events(self, client, world, clock, sync_mode):
        if isinstance(self._control, carla.VehicleControl):
            current_lights = self._lights
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return True
            elif event.type == pygame.KEYUP:
                if self._is_quit_shortcut(event.key):
                    return True
                elif event.key == K_F1:
                    world.hud.toggle_info()
                elif event.key == K_h or (event.key == K_SLASH and pygame.key.get_mods() & KMOD_SHIFT):
                    world.hud.help.toggle()
                elif event.key == K_TAB:
                    world.camera_manager.toggle_camera()
                elif event.key == K_BACKSPACE and not world.simulation_clock:
                    world.startScenario()
                elif event.key == K_BACKSPACE and world.simulation_clock:
                    world.finishScenario()
                    return True
                # Driving key scenarios
                if isinstance(self._control, carla.VehicleControl):
                    if event.key == K_q:
                        self._control.gear = 1 if self._control.reverse else -1
                    elif event.key == K_m:
                        self._control.manual_gear_shift = not self._control.manual_gear_shift
                        self._control.gear = world.player.get_control().gear
                        world.hud.notification('%s Transmission' %
                                               ('Manual' if self._control.manual_gear_shift else 'Automatic'))
                    elif self._control.manual_gear_shift and event.key == K_COMMA:
                        self._control.gear = max(-1, self._control.gear - 1)
                    elif self._control.manual_gear_shift and event.key == K_PERIOD:
                        self._control.gear = self._control.gear + 1
                    elif event.key == K_p and not pygame.key.get_mods() & KMOD_CTRL:
                        if not self._autopilot_enabled and not sync_mode:
                            print("WARNING: You are currently in asynchronous mode and could "
                                  "experience some issues with the traffic simulation")
                        self._autopilot_enabled = not self._autopilot_enabled
                        world.player.set_autopilot(self._autopilot_enabled)
                        world.hud.notification(
                            'Autopilot %s' % ('On' if self._autopilot_enabled else 'Off'))
                    elif event.key == K_l and pygame.key.get_mods() & KMOD_CTRL:
                        current_lights ^= carla.VehicleLightState.Special1
                    elif event.key == K_l and pygame.key.get_mods() & KMOD_SHIFT:
                        current_lights ^= carla.VehicleLightState.HighBeam
                    elif event.key == K_l:
                        # Use 'L' key to switch between lights:
                        # closed -> position -> low beam -> fog
                        if not self._lights & carla.VehicleLightState.Position:
                            world.hud.notification("Position lights")
                            current_lights |= carla.VehicleLightState.Position
                        else:
                            world.hud.notification("Low beam lights")
                            current_lights |= carla.VehicleLightState.LowBeam
                        if self._lights & carla.VehicleLightState.LowBeam:
                            world.hud.notification("Fog lights")
                            current_lights |= carla.VehicleLightState.Fog
                        if self._lights & carla.VehicleLightState.Fog:
                            world.hud.notification("Lights off")
```

```
                            current_lights ^= carla.VehicleLightState.Position
                            current_lights ^= carla.VehicleLightState.LowBeam
                            current_lights ^= carla.VehicleLightState.Fog
                    elif event.key == K_i:
                        current_lights ^= carla.VehicleLightState.Interior
                    elif event.key == K_z:
                        current_lights ^= carla.VehicleLightState.LeftBlinker
                    elif event.key == K_x:
                        current_lights ^= carla.VehicleLightState.RightBlinker

        if not self._autopilot_enabled:
            if isinstance(self._control, carla.VehicleControl):
                self._parse_vehicle_keys(pygame.key.get_pressed(), clock.get_time())
                self._control.reverse = self._control.gear < 0
                # Set automatic control-related vehicle lights
                if self._control.brake:
                    current_lights |= carla.VehicleLightState.Brake
                else: # Remove the Brake flag
                    current_lights &= ~carla.VehicleLightState.Brake
                if self._control.reverse:
                    current_lights |= carla.VehicleLightState.Reverse
                else: # Remove the Reverse flag
                    current_lights &= ~carla.VehicleLightState.Reverse
                if current_lights != self._lights: # Change the light state only if necessary
                    self._lights = current_lights
                    world.player.set_light_state(carla.VehicleLightState(self._lights))
            elif isinstance(self._control, carla.WalkerControl):
                self._parse_walker_keys(pygame.key.get_pressed(), clock.get_time(), world)
            world.player.apply_control(self._control)

    def _parse_vehicle_keys(self, keys, milliseconds):
        if keys[K_UP] or keys[K_w]:
            self._control.throttle = min(self._control.throttle + 0.01, 1.00)
        else:
            self._control.throttle = 0.0

        if keys[K_DOWN] or keys[K_s]:
            self._control.brake = min(self._control.brake + 0.2, 1)
        else:
            self._control.brake = 0

        steer_increment = 5e-4 * milliseconds
        if keys[K_LEFT] or keys[K_a]:
            if self._steer_cache > 0:
                self._steer_cache = 0
            else:
                self._steer_cache -= steer_increment
        elif keys[K_RIGHT] or keys[K_d]:
            if self._steer_cache < 0:
                self._steer_cache = 0
            else:
                self._steer_cache += steer_increment
        else:
            self._steer_cache = 0.0
        self._steer_cache = min(0.7, max(-0.7, self._steer_cache))
        self._control.steer = round(self._steer_cache, 1)
        self._control.hand_brake = keys[K_SPACE]

    def _parse_walker_keys(self, keys, milliseconds, world):
        self._control.speed = 0.0
        if keys[K_DOWN] or keys[K_s]:
            self._control.speed = 0.0
        if keys[K_LEFT] or keys[K_a]:
            self._control.speed = .01
            self._rotation.yaw -= 0.08 * milliseconds
        if keys[K_RIGHT] or keys[K_d]:
            self._control.speed = .01
            self._rotation.yaw += 0.08 * milliseconds
        if keys[K_UP] or keys[K_w]:
            self._control.speed = world.player_max_speed_fast if pygame.key.get_mods() & KMOD_SHIFT else world.player_max_speed
        self._control.jump = keys[K_SPACE]
        self._rotation.yaw = round(self._rotation.yaw, 1)
        self._control.direction = self._rotation.get_forward_vector()

    @staticmethod
    def _is_quit_shortcut(key):
        return (key == K_ESCAPE) or (key == K_q and pygame.key.get_mods() & KMOD_CTRL)


# ==============================================================================
# -- HUD -----------------------------------------------------------------------
# ==============================================================================


class HUD(object):
    def __init__(self, width, height):
        self.dim = (width, height)
        font = pygame.font.Font(pygame.font.get_default_font(), 20)
        font_name = 'courier' if os.name == 'nt' else 'mono'
        fonts = [x for x in pygame.font.get_fonts() if font_name in x]
        default_font = 'ubuntumono'
        mono = default_font if default_font in fonts else fonts[0]
        mono = pygame.font.match_font(mono)
        self._font_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)
        self._notifications = FadingText(font, (width, 40), (0, height - 40))
        self.help = HelpText(pygame.font.Font(mono, 16), width, height)
        self.server_fps = 0
        self.frame = 0
        self.simulation_time = 0
        self._show_info = True
        self._info_text = []
        self._server_clock = pygame.time.Clock()

    def on_world_tick(self, timestamp):
        self._server_clock.tick()
        self.server_fps = self._server_clock.get_fps()
        self.frame = timestamp.frame
        self.simulation_time = timestamp.elapsed_seconds

    def tick(self, world, time):
        self._notifications.tick(world, time)
        if not self._show_info:
            return
        t = world.player.get_transform()
        v = world.player.get_velocity()
        c = world.player.get_control()
        compass = world.imu_sensor.compass
        heading = 'N' if compass > 270.5 or compass < 89.5 else ''
        heading += 'S' if 90.5 < compass < 269.5 else ''
        heading += 'E' if 0.5 < compass < 179.5 else ''
        heading += 'W' if 180.5 < compass < 359.5 else ''
        colhist = world.collision_sensor.get_collision_history()
        collision = [colhist[x + self.frame - 200] for x in range(0, 200)]
        max_col = max(1.0, max(collision))
        collision = [x / max_col for x in collision]
        vehicles = world.world.get_actors().filter('vehicle.*')
        self._info_text = [
            'Vehicle: % 20s' % get_actor_display_name(world.player, truncate=20),
            'Route time: % 12s' % datetime.timedelta(seconds=int(time)),
            '',
```

```
            'Speed:   % 15.0f km/h' % (3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)),
            '']
        if isinstance(c, carla.VehicleControl):
            self._info_text += [
                ('Throttle:', c.throttle, 0.0, 1.0),
                ('Steer:', c.steer, -1.0, 1.0),
                ('Brake:', c.brake, 0.0, 1.0),
                ('Reverse:', c.reverse),
                ('Hand brake:', c.hand_brake),
                ('Manual:', c.manual_gear_shift),
                'Gear:        %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)]
        self._info_text += [
            '',
            'Collision:',
            collision,
            '']

    def toggle_info(self):
        self._show_info = not self._show_info

    def notification(self, text, seconds=2.0):
        self._notifications.set_text(text, seconds=seconds)

    def error(self, text):
        self._notifications.set_text('Error: %s' % text, (255, 0, 0))

    def render(self, display):
        if self._show_info:
            info_surface = pygame.Surface((220, self.dim[1]))
            info_surface.set_alpha(100)
            display.blit(info_surface, (0, 0))
            v_offset = 4
            bar_h_offset = 100
            bar_width = 106
            for item in self._info_text:
                if v_offset + 18 > self.dim[1]:
                    break
                if isinstance(item, list):
                    if len(item) > 1:
                        points = [(x + 8, v_offset + 8 + (1.0 - y) * 30) for x, y in enumerate(item)]
                        pygame.draw.lines(display, (255, 136, 0), False, points, 2)
                    item = None
                    v_offset += 18
                elif isinstance(item, tuple):
                    if isinstance(item[1], bool):
                        rect = pygame.Rect((bar_h_offset, v_offset + 8), (6, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1] else 1)
                    else:
                        rect_border = pygame.Rect((bar_h_offset, v_offset + 8), (bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect_border, 1)
                        f = (item[1] - item[2]) / (item[3] - item[2])
                        if item[2] < 0.0:
                            rect = pygame.Rect((bar_h_offset + f * (bar_width - 6), v_offset + 8), (6, 6))
                        else:
                            rect = pygame.Rect((bar_h_offset, v_offset + 8), (f * bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect)
                    item = item[0]
                if item:  # At this point has to be a str.
                    surface = self._font_mono.render(item, True, (255, 255, 255))
                    display.blit(surface, (8, v_offset))
                v_offset += 18
        self._notifications.render(display)
        self.help.render(display)


# ==============================================================================
# -- FadingText ----------------------------------------------------------------
# ==============================================================================


class FadingText(object):
    def __init__(self, font, dim, pos):
        self.font = font
        self.dim = dim
        self.pos = pos
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)

    def set_text(self, text, color=(255, 255, 255), seconds=2.0):
        text_texture = self.font.render(text, True, color)
        self.surface = pygame.Surface(self.dim)
        self.seconds_left = seconds
        self.surface.fill((0, 0, 0, 0))
        self.surface.blit(text_texture, (10, 11))

    def tick(self, _, time):
        delta_seconds = 1e-3 * time
        self.seconds_left = max(0.0, self.seconds_left - delta_seconds)
        self.surface.set_alpha(500.0 * self.seconds_left)

    def render(self, display):
        display.blit(self.surface, self.pos)


# ==============================================================================
# -- HelpText ------------------------------------------------------------------
# ==============================================================================


class HelpText(object):
    """Helper class to handle text output using pygame"""
    def __init__(self, font, width, height):
        lines = __doc__.split('\n')
        self.font = font
        self.line_space = 18
        self.dim = (780, len(lines) * self.line_space + 12)
        self.pos = (0.5 * width - 0.5 * self.dim[0], 0.5 * height - 0.5 * self.dim[1])
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)
        self.surface.fill((0, 0, 0, 0))
        for n, line in enumerate(lines):
            text_texture = self.font.render(line, True, (255, 255, 255))
            self.surface.blit(text_texture, (22, n * self.line_space))
            self._render = False
        self.surface.set_alpha(220)

    def toggle(self):
        self._render = not self._render

    def render(self, display):
        if self._render:
            display.blit(self.surface, self.pos)



# ==============================================================================
# -- CollisionSensor -----------------------------------------------------------
# ==============================================================================


```

```
class CollisionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self.history = []
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.collision')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))

    def get_collision_history(self):
        history = collections.defaultdict(int)
        for frame, intensity in self.history:
            history[frame] += intensity
        return history

    @staticmethod
    def _on_collision(weak_self, event):
        self = weak_self()
        if not self:
            return
        actor_type = get_actor_display_name(event.other_actor)
        self.hud.notification('Collision with %r' % actor_type)
        impulse = event.normal_impulse
        intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
        self.history.append((event.frame, intensity))
        if len(self.history) > 4000:
            self.history.pop(0)


# ==============================================================================
# -- LaneInvasionSensor --------------------------------------------------------
# ==============================================================================


class LaneInvasionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None

        # If the spawn object is not a vehicle, we cannot use the Lane Invasion Sensor
        if parent_actor.type_id.startswith("vehicle."):
            self._parent = parent_actor
            self.hud = hud
            world = self._parent.get_world()
            bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
            self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
            # We need to pass the lambda a weak reference to self to avoid circular
            # reference.
            weak_self = weakref.ref(self)
            self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))

    @staticmethod
    def _on_invasion(weak_self, event):
        self = weak_self()
        if not self:
            return
        lane_types = set(x.type for x in event.crossed_lane_markings)
        text = ['%r' % str(x).split()[-1] for x in lane_types]
        self.hud.notification('Crossed line %s' % ' and '.join(text))


# ==============================================================================
# -- GnssSensor ----------------------------------------------------------------
# ==============================================================================


class GnssSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        self.lat = 0.0
        self.lon = 0.0
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.gnss')
        self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0, z=2.8)), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: GnssSensor._on_gnss_event(weak_self, event))

    @staticmethod
    def _on_gnss_event(weak_self, event):
        self = weak_self()
        if not self:
            return
        self.lat = event.latitude
        self.lon = event.longitude


# ==============================================================================
# -- IMUSensor -----------------------------------------------------------------
# ==============================================================================


class IMUSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        self.accelerometer = (0.0, 0.0, 0.0)
        self.gyroscope = (0.0, 0.0, 0.0)
        self.compass = 0.0
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.imu')
        self.sensor = world.spawn_actor(
            bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(
            lambda sensor_data: IMUSensor._IMU_callback(weak_self, sensor_data))

    @staticmethod
    def _IMU_callback(weak_self, sensor_data):
        self = weak_self()
        if not self:
            return
        limits = (-99.9, 99.9)
        self.accelerometer = (
            max(limits[0], min(limits[1], sensor_data.accelerometer.x)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.y)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.z)))
        self.gyroscope = (
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.x))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.y))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.z))))
        self.compass = math.degrees(sensor_data.compass)
```

```
843
844
845 # ==============================================================================
846 # -- RadarSensor ---------------------------------------------------------------
847 # ==============================================================================
848
849
850 class RadarSensor(object):
851     def __init__(self, parent_actor):
852         self.sensor = None
853         self._parent = parent_actor
854         bound_x = 0.5 + self._parent.bounding_box.extent.x
855         bound_y = 0.5 + self._parent.bounding_box.extent.y
856         bound_z = 0.5 + self._parent.bounding_box.extent.z
857
858         self.velocity_range = 7.5 # m/s
859         world = self._parent.get_world()
860         self.debug = world.debug
861         bp = world.get_blueprint_library().find('sensor.other.radar')
862         bp.set_attribute('horizontal_fov', str(35))
863         bp.set_attribute('vertical_fov', str(20))
864         self.sensor = world.spawn_actor(
865             bp,
866             carla.Transform(
867                 carla.Location(x=bound_x + 0.05, z=bound_z+0.05),
868                 carla.Rotation(pitch=5)),
869             attach_to=self._parent)
870         # We need a weak reference to self to avoid circular reference.
871         weak_self = weakref.ref(self)
872         self.sensor.listen(
873             lambda radar_data: RadarSensor._Radar_callback(weak_self, radar_data))
874
875     @staticmethod
876     def _Radar_callback(weak_self, radar_data):
877         self = weak_self()
878         if not self:
879             return
880         # To get a numpy [[vel, altitude, azimuth, depth],...[,,,]]:
881         # points = np.frombuffer(radar_data.raw_data, dtype=np.dtype('f4'))
882         # points = np.reshape(points, (len(radar_data), 4))
883
884         current_rot = radar_data.transform.rotation
885         for detect in radar_data:
886             azi = math.degrees(detect.azimuth)
887             alt = math.degrees(detect.altitude)
888             # The 0.25 adjusts a bit the distance so the dots can
889             # be properly seen
890             fw_vec = carla.Vector3D(x=detect.depth - 0.25)
891             carla.Transform(
892                 carla.Location(),
893                 carla.Rotation(
894                     pitch=current_rot.pitch + alt,
895                     yaw=current_rot.yaw + azi,
896                     roll=current_rot.roll)).transform(fw_vec)
897
898             def clamp(min_v, max_v, value):
899                 return max(min_v, min(value, max_v))
900
901             norm_velocity = detect.velocity / self.velocity_range # range [-1, 1]
902             r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
903             g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
904             b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
905             self.debug.draw_point(
906                 radar_data.transform.location + fw_vec,
907                 size=0.075,
908                 life_time=0.06,
909                 persistent_lines=False,
910                 color=carla.Color(r, g, b))
911
912 # ==============================================================================
913 # -- CameraManager -------------------------------------------------------------
914 # ==============================================================================
915
916
917 class CameraManager(object):
918     def __init__(self, parent_actor, hud, gamma_correction):
919         self.sensor = None
920         self.surface = None
921         self._parent = parent_actor
922         self.hud = hud
923         self.recording = False
924         bound_x = 0.5 + self._parent.bounding_box.extent.x
925         bound_y = 0.5 + self._parent.bounding_box.extent.y
926         bound_z = 0.5 + self._parent.bounding_box.extent.z
927         Attachment = carla.AttachmentType
928
929         self._camera_transforms = [
930             # Third-person
931             (carla.Transform(carla.Location(x=-5.0, z=2.0), carla.Rotation(pitch=6.0)), Attachment.SpringArm),
932             # Watch back
933             (carla.Transform(carla.Location(x=4.0, z=2.0), carla.Rotation(yaw=0, pitch=6)), Attachment.SpringArm),
934             # Watch to the left
935             (carla.Transform(carla.Location(x=0, z=2.0, y=3), carla.Rotation(yaw=-90, pitch=0)), Attachment.Rigid),
936             # Watch to the right
937             (carla.Transform(carla.Location(x=0, z=2.0, y=-3), carla.Rotation(yaw=90, pitch=0)), Attachment.Rigid),
938             # First-person
939             (carla.Transform(carla.Location(y=-0.42, x=0.03, z=1.2), carla.Rotation(yaw=0, roll=0, pitch=-10)), Attachment.Rigid)]
940
941         self.transform_index = 1
942         self.sensors = [
943             ['sensor.camera.rgb', cc.Raw, 'Camera RGB', {}],
944             ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)', {}],
945             ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)', {}],
946             ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic Gray Scale)', {}],
947             ['sensor.camera.semantic_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)', {}],
948             ['sensor.camera.semantic_segmentation', cc.CityScapesPalette, 'Camera Semantic Segmentation (CityScapes Palette)', {}],
949             ['sensor.camera.instance_segmentation', cc.CityScapesPalette, 'Camera Instance Segmentation (CityScapes Palette)', {}],
950             ['sensor.camera.instance_segmentation', cc.Raw, 'Camera Instance Segmentation (Raw)', {}],
951             ['sensor.lidar.ray_cast', None, 'Lidar (Ray-Cast)', {'range': '50'}],
952             ['sensor.camera.dvs', cc.Raw, 'Dynamic Vision Sensor', {}],
953             ['sensor.camera.rgb', cc.Raw, 'Camera RGB Distorted',
954                 {'lens_circle_multiplier': '3.0',
955                 'lens_circle_falloff': '3.0',
956                 'chromatic_aberration_intensity': '0.5',
957                 'chromatic_aberration_offset': '0'}],
958             ['sensor.camera.optical_flow', cc.Raw, 'Optical Flow', {}],
959         ]
960         world = self._parent.get_world()
961         bp_library = world.get_blueprint_library()
962         for item in self.sensors:
963             bp = bp_library.find(item[0])
964             if item[0].startswith('sensor.camera'):
965                 bp.set_attribute('image_size_x', str(hud.dim[0]))
966                 bp.set_attribute('image_size_y', str(hud.dim[1]))
967                 if bp.has_attribute('gamma'):
968                     bp.set_attribute('gamma', str(gamma_correction))
969                 for attr_name, attr_value in item[3].items():
970                     bp.set_attribute(attr_name, attr_value)
971             elif item[0].startswith('sensor.lidar'):
972                 self.lidar_range = 50
```

```
                for attr_name, attr_value in item[3].items():
                    bp.set_attribute(attr_name, attr_value)
                    if attr_name == 'range':
                        self.lidar_range = float(attr_value)

            item.append(bp)
        self.index = None

    def toggle_camera(self):
        self.transform_index = (self.transform_index + 1) % len(self._camera_transforms)
        self.set_sensor(self.index, notify=False, force_respawn=True)

    def set_sensor(self, index, notify=True, force_respawn=False):
        index = index % len(self.sensors)
        needs_respawn = True if self.index is None else \
            (force_respawn or (self.sensors[index][2] != self.sensors[self.index][2]))
        if needs_respawn:
            if self.sensor is not None:
                self.sensor.destroy()
                self.surface = None
            self.sensor = self._parent.get_world().spawn_actor(
                self.sensors[index][-1],
                self._camera_transforms[self.transform_index][0],
                attach_to=self._parent,
                attachment_type=self._camera_transforms[self.transform_index][1])
            # We need to pass the lambda a weak reference to self to avoid
            # circular reference.
            weak_self = weakref.ref(self)
            self.sensor.listen(lambda image: CameraManager._parse_image(weak_self, image))
        if notify:
            self.hud.notification(self.sensors[index][2])
        self.index = index

    def next_sensor(self):
        self.set_sensor(self.index + 1)

    def toggle_recording(self):
        self.recording = not self.recording
        self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))

    def render(self, display):
        if self.surface is not None:
            display.blit(self.surface, (0, 0))

    @staticmethod
    def _parse_image(weak_self, image):
        self = weak_self()
        if not self:
            return
        if self.sensors[self.index][0].startswith('sensor.lidar'):
            points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
            points = np.reshape(points, (int(points.shape[0] / 4), 4))
            lidar_data = np.array(points[:, :2])
            lidar_data *= min(self.hud.dim) / (2.0 * self.lidar_range)
            lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
            lidar_data = np.fabs(lidar_data)  # pylint: disable=E1111
            lidar_data = lidar_data.astype(np.int32)
            lidar_data = np.reshape(lidar_data, (-1, 2))
            lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
            lidar_img = np.zeros((lidar_img_size), dtype=np.uint8)
            lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
            self.surface = pygame.surfarray.make_surface(lidar_img)
        elif self.sensors[self.index][0].startswith('sensor.camera.dvs'):
            # Example of converting the raw_data from a carla.DVSEventArray
            # sensor into a NumPy array and using it as an image
            dvs_events = np.frombuffer(image.raw_data, dtype=np.dtype([
                ('x', np.uint16), ('y', np.uint16), ('t', np.int64), ('pol', np.bool)]))
            dvs_img = np.zeros((image.height, image.width, 3), dtype=np.uint8)
            # Blue is positive, red is negative
            dvs_img[dvs_events[:]['y'], dvs_events[:]['x'], dvs_events[:]['pol'] * 2] = 255
            self.surface = pygame.surfarray.make_surface(dvs_img.swapaxes(0, 1))
        elif self.sensors[self.index][0].startswith('sensor.camera.optical_flow'):
            image = image.get_color_coded_flow()
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        else:
            image.convert(self.sensors[self.index][1])
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        if self.recording:
            image.save_to_disk('_out/%08d' % image.frame)


# ==============================================================================
# -- game_loop() ---------------------------------------------------------------
# ==============================================================================


def game_loop(args, scenario, path_details):
    pygame.init()
    pygame.font.init()
    world = None
    original_settings = None
    try:
        client = carla.Client(args.host, args.port)
        client.set_timeout(20.0)
        sim_world = client.get_world()
        if args.sync:
            original_settings = sim_world.get_settings()
            settings = sim_world.get_settings()
            if not settings.synchronous_mode:
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 0.05
            sim_world.apply_settings(settings)
            traffic_manager = client.get_trafficmanager()
            traffic_manager.set_synchronous_mode(True)

        if args.autopilot and not sim_world.get_settings().synchronous_mode:
            print("WARNING: You are currently in asynchronous mode and could "
                  "experience some issues with the traffic simulation")

        display = pygame.display.set_mode(
            (args.width, args.height),
            pygame.HWSURFACE | pygame.DOUBLEBUF)
        display.fill((0,0,0))
        pygame.display.flip()
        ### MY PART ###
        scenario.setup(traffic_manager, sim_world)
        scenario.spawn(client)
        logger.info("The scenario was set. Spawning the hero vehicle and launching the simulation")
        ###   ###   ###
        client.set_timeout(10.0)
        hud = HUD(args.width, args.height)
```

```
        world = World(sim_world, hud, args, scenario, client, path_details)
        controller = KeyboardControl(world, args.autopilot)
        if args.sync:
            sim_world.tick()
        else:
            sim_world.wait_for_tick()

        clock = pygame.time.Clock()
        while True:
            try:
                if args.sync:
                    sim_world.tick()
                clock.tick_busy_loop(FPS)
                if controller.parse_events(client, world, clock, args.sync):
                    return
                if world.tick():
                    return
                world.render(display)
                pygame.display.flip()
            except Exception as e:
                logger.error(e)
    except Exception as e:
        logger.error("Something went wrong trying to launch the driver.py", e)
    finally:
        ### MY PART ###
        if(scenario.populated):
            scenario.finish(client)
        if(world and world.manager):
            logger.info('Total time simulated: {}s'.format(str(world.simulation_time/1000)))
        ###   ###   ###

        if original_settings:
            sim_world.apply_settings(original_settings)

        if (world and world.recording_enabled):
            client.stop_recorder()

        if world is not None:
            world.destroy()

        pygame.quit()

# ==============================================================================
# -- main() --------------------------------------------------------------------
# ==============================================================================

argparser = argparse.ArgumentParser(
    description='CARLA Manual Control Client')
argparser.add_argument(
    '-v', '--verbose',
    action='store_true',
    dest='debug',
    help='print debug information')
argparser.add_argument(
    '--host',
    metavar='H',
    default='127.0.0.1',
    help='IP of the host server (default: 127.0.0.1)')
argparser.add_argument(
    '-p', '--port',
    metavar='P',
    default=2000,
    type=int,
    help='TCP port to listen to (default: 2000)')
argparser.add_argument(
    '-a', '--autopilot',
    action='store_true',
    help='enable autopilot')
argparser.add_argument(
    '--res',
    metavar='WIDTHxHEIGHT',
    default='1280x720',
    help='window resolution (default: 1280x720)')
argparser.add_argument(
    '--filter', '-f',
    metavar='PATTERN',
    default='vehicle.*',
    help='actor filter (default: "vehicle.*")')
argparser.add_argument(
    '--generation',
    metavar='G',
    default='2',
    help='restrict to certain actor generation (values: "1","2","All" - default: "2")')
argparser.add_argument(
    '--rolename',
    metavar='NAME',
    default='hero',
    help='actor role name (default: "hero")')
argparser.add_argument(
    '--gamma',
    default=2.2,
    type=float,
    help='Gamma correction of the camera (default: 2.2)')
argparser.add_argument(
    '--sync',
    action='store_true',
    help='Activate synchronous mode execution')
argparser.add_argument(
    '--red',
    default="0",
    help='Red part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--green',
    default="0",
    help='Green part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--blue',
    default="0",
    help='Blue part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '-n', '--name',
    metavar='F',
    default="example",
    help='Participant name (name)')
argparser.add_argument(
    '-s', '--scenario',
    metavar='F',
    default="example",
    help='Scenario name (name)')
argparser.add_argument("s_b64")
argparser.add_argument("p_b64")
args = argparser.parse_args()

logger = logging.getLogger("driver-thread")
logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %

def main():
    args.width, args.height = [int(x) for x in args.res.split('x')]

    print(__doc__)
```

```
    try:

        scenario_b64 = args.s_b64
        path_b64 = args.p_b64

        scenario_bytes = base64.b64decode(scenario_b64.encode())
        path_bytes = base64.b64decode(path_b64.encode())

        scenario = pickle.loads(scenario_bytes)
        path_details = pickle.loads(path_bytes)
        game_loop(args, scenario, path_details)

    except KeyboardInterrupt as e:
        logger.warning('Keyboard interupt while running driver.py script', e)
    except Exception as e:
        logger.error("Something went wrong.", e)


if __name__ == '__main__':

    main()
```

**File: ./software/carla_scripts/helper_scripts/print_coordinates.py**

```
import glob
import os
import sys
import time
import random

from agents.navigation.behavior_agent import BehaviorAgent, BasicAgent

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

# Used to print coordinates of the spectator (the one flying in the simulator and observing the world) to the terminal
# Mainly used to assist in scenario designing to know exact coordinates of specific locations
def main():
    client = carla.Client('localhost', 2000)
    client.set_timeout(2.0)
    world = client.get_world()
    try:
        print("Started printing spectator's coordinates.")
        while(True):
            spectator = world.get_spectator().get_transform()
            coordinates_string = "{}, {}, {}  |  x, y, z".format('%.3f' % spectator.location.x, '%.3f' % spectator.location.y, '%.3f' % spectator.location.z)
            print(coordinates_string)
            time.sleep(1)
    except KeyboardInterrupt:
        print("Stopped printing coordinates.")
    except:
        print("Something wrong happened trying to print the spectator's coordinates.")


if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/helper_scripts/draw_lanes_terminal.py**

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-p', '--path',
    metavar='P',
    default="path1",
    help='Path filename (path1)')
args = argparser.parse_args()

# Reads file and extracts locations
def get_route(map):
    waypoints = []
    grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
    locations = read_path_file()

    for i in range(len(locations)-1):
        waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
    return Route(waypoints)

def read_path_file():
    file_path = "../../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations
    except Exception as e:
        print("Could not read the file {}".format(args.path))


# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    map = world.get_map()


    route = get_route(map)
    try:
        print("Started drawing {}.".format(args.path))
        finished = False
        all_waypoints = route.get_all_waypoints()
        start_color = carla.Color(r=0, g=0, b=255)
        finish_color = carla.Color(r=255, g=0, b=0)
        num_waypoints = len(all_waypoints)
        while(not finished):

            # Draw lanes that can be driven
            for i, waypoint in enumerate(all_waypoints):
                color_progress = float(i) / (num_waypoints - 1)

                r = (finish_color.r - start_color.r) / (num_waypoints - 1)
                g = (finish_color.g - start_color.g) / (num_waypoints - 1)
                b = (finish_color.b - start_color.b) / (num_waypoints - 1)
                color = carla.Color(
                    r = int(start_color.r + i * r),
                    g = int(start_color.g + i * g),
                    b = int(start_color.b + i * b)
                )
                world.debug.draw_string(waypoint.transform.location, 'o', draw_shadow=False,color=color, life_time=10,persistent_lines=True)

            time.sleep(9)
    except KeyboardInterrupt:
        print("Stopped drawing the path.")
    except Exception as e:
        print("Something wrong happened trying to draw the path. Aborting...", e)

if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/helper_scripts/save_opendrive_map.py**

```python
#!/usr/bin/env python

# Copyright (c) 2019 Computer Vision Center (CVC) at the Universitat Autonoma de
# Barcelona (UAB).
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

import glob
import os
import sys

import carla

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

# Used to save the current map in the simulation to an .xml file
def main():

    try:
        client = carla.Client('127.0.0.1', 2000)
        client.set_timeout(10)
        world = client.get_world()
        map = world.get_map()

        name = map.name.rsplit('/', 1)[-1]
        world.get_map().save_to_disk("maps/{}.xml".format(name))

    except Exception as e:
        print("Something wrong happened:", e)
    finally:
        print("Finished")


if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/helper_scripts/hero_info.py**

```python
import glob
import os
import sys
import time
import random

from agents.navigation.behavior_agent import BehaviorAgent, BasicAgent

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

# Used to print the bounding box of the "hero" vehicle
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    try:
        # Get the player
        player = None
        actors = world.get_actors()
        for actor in actors:
            if actor.attributes.get('role_name') == 'hero':
                player = actor

        box = player.bounding_box.extent
        print("The bounding box (x, y, z):")
        print(box.x)
        print(box.y)
        print(str(box.z) + "\n")
    except Exception as e:
        print("Something wrong happened trying to get the bounding box of the actor named hero:", e)

if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/helper_scripts/draw_points_terminal.py**

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-p', '--path',
    metavar='P',
    default="path1",
    help='Path filename (path1)')
args = argparser.parse_args()

def read_path_file():
    file_path = "../../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations
    except Exception as e:
        print("Could not read the file {}".format(args.path))


# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    map = world.get_map()

    locations = read_path_file()
    try:
        finished = False
        while(not finished):

            for l in locations[1:]:
                world.debug.draw_string(l, 'o000o', draw_shadow=False,color=carla.Color(r=0, g=0, b=255), life_time=10,persistent_lines=True)

            world.debug.draw_string(locations[0], 'o000o', draw_shadow=False,color=carla.Color(r=255, g=0, b=0), life_time=10,persistent_lines=True)
            time.sleep(5)
    except KeyboardInterrupt:
        print("Stopped drawing the path.")
    except Exception as e:
        print("Something wrong happened trying to draw the path. Aborting...", e)

if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/helper_scripts/draw_hero.py**

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-l', '--log',
    help='Log path')
args = argparser.parse_args()

logger = None
if args.log:
    logger = logging.getLogger("draw_hero-thread")
    logging.basicConfig(level=logging.INFO, filename=args.log, filemode="a", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")

# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()

    try:
        message = "Begin marking the player's location in green"
        if logger is not None: logger.info(message)
        else: print(message)
        # Every step
        finished = False
        while(not finished):
            # Get the player
            player = None
            actors = world.get_actors()
            for actor in actors:
                if actor.attributes.get('role_name') == 'hero':
                    player = actor
            if player:
                world.debug.draw_string(player.get_location(), 'O', draw_shadow=False,color=carla.Color(r=0, g=255, b=0), life_time=0.2,persistent_lines=True)
            else:
                time.sleep(2)
            time.sleep(0.1)
    except KeyboardInterrupt:
        if logger is None: print("Stopped marking the player's location.")
    except Exception as e:
        message = "Something wrong happened trying to mark the player's location. Aborting..."
        if logger is not None: logger.error(message, e)
        else:  print(message)
    finally:
        message = "Stopped drawing lanes."
        if logger is not None: logger.info(message)
        else: print(message)

if __name__ == '__main__':
    main()
```

**File: ./software/carla_scripts/helper_scripts/draw_lanes.py**

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-p', '--path',
    metavar='F',
    default="path1",
    help='Path filename (path1)')
argparser.add_argument(
    '-l', '--log',
    help='Log path')
argparser.add_argument("locations")
args = argparser.parse_args()

logger = None
if args.log:
    logger = logging.getLogger("draw_lanes-thread")
    logging.basicConfig(level=logging.INFO, filename=args.log, filemode="a", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")


def extract_locations(path_details):
    locations = []
    for loc in path_details:
        locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
    return locations

# Reads file and extracts locations
def get_route(map, details):
    waypoints = []
    grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
    if details is not None: locations = extract_locations(details)
    else: locations = read_path_file

    for i in range(len(locations)-1):
        waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
    return Route(waypoints)

def read_path_file():
    file_path = "../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations
    except Exception as e:
        print("Could not read the file {}".format(args.path))


# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    map = world.get_map()

    locations = None
    if args.locations:
        locs = args.locations
        locs_decoded = base64.b64decode(locs.encode())
        locations = pickle.loads(locs_decoded)
    route = get_route(map, locations)
    try:
        if logger is not None: logger.info("Started drawing the path")
        else: print("Started drawing {}.".format(args.path))
        # Every step
        finished = False
        while(not finished):
            all_waypoints = route.get_all_waypoints()

            # Draw lanes that can be driven
            for waypoint in all_waypoints:
                world.debug.draw_string(waypoint.transform.location, 'o', draw_shadow=False,color=carla.Color(r=0, g=0, b=255), life_time=10,persistent_lines=Tru

            # Draw finish line
            for waypoint in route.finish_lane_waypoints:
                world.debug.draw_string(waypoint.transform.location, 'o', draw_shadow=False,color=carla.Color(r=255, g=0, b=0), life_time=10,persistent_lines=Tru
            time.sleep(9)
    except KeyboardInterrupt:
        if logger is None: print("Stopped drawing the path.")
    except Exception as e:
        if logger is not None: logger.error("Something wrong happened trying to draw the path. Aborting...", e)
        else:  print("Something wrong happened trying to draw the path. Aborting...", e)
    finally:
        if logger is not None: logger.info("Stopped drawing lanes.")

if __name__ == '__main__':
    main()
```

**File: ./software/python_libraries/requirements.txt**

```
antlr4-python3-runtime==4.9.3
appnope==0.1.3
asgiref==3.5.2
asttokens==2.2.1
backcall==0.2.0
certifi==2022.12.7
charset-normalizer==3.0.1
click==8.1.3
```

```
commonroad-all==0.0.1
commonroad-drivability-checker==2022.2.1
commonroad-io==2022.3
commonroad-route-planner==2022.3
commonroad-scenario-designer==0.6.0
commonroad-vehicle-models==3.0.2
contourpy==1.0.7
cycler==0.11.0
decorator==5.1.1
Django==3.2
docker==6.0.1
enum34==1.1.10
executing==1.2.0
ffmpeg==1.4
fonttools==4.38.0
grpcio==1.51.1
idna==3.4
ipython==8.9.0
iso3166==2.1.1
jedi==0.18.2
joblib==1.2.0
kiwisolver==1.4.4
libsumo==1.15.0
lxml==4.9.2
matplotlib==3.6.3
matplotlib-inline==0.1.6
mercantile==1.2.1
networkx==3.0
numpy==1.24.1
omegaconf==2.3.0
opendrive2lanelet==1.2.1
ordered-set==4.1.0
packaging==23.0
parso==0.8.3
pexpect==4.8.0
pickleshare==0.7.5
Pillow==9.4.0
prompt-toolkit==3.0.36
protobuf==3.20.1
ptyprocess==0.7.0
pure-eval==0.2.2
Pygments==2.14.0
pyparsing==3.0.9
pyproj==3.4.1
PyQt5==5.15.8
PyQt5-Qt5==5.15.2
PyQt5-sip==12.11.1
python-dateutil==2.8.2
pytz==2022.5
PyYAML==6.0
requests==2.28.2
Rtree==1.0.1
scikit-learn==1.2.0
scipy==1.10.0
Shapely==1.8.5
six==1.16.0
sqlparse==0.4.3
stack-data==0.6.2
sumocr==2023.1
sumolib==1.15.0
threadpoolctl==3.1.0
tqdm==4.64.1
traci==1.15.0
traitlets==5.9.0
urllib3==1.26.14
utm==0.7.0
wcwidth==0.2.6
websocket-client==1.5.0
```

**File: ./software/python_libraries/README.md**

```
## Introduction
This directory contains the `crdesigner` python library and the requirements.txt file containing a list of python libraries needed to work with this framework.

`crdesigner` is a modified version of the `crdesigner` library (version 0.6.0). This version was the most recent release available at the time of the research.
More about crdesigner can be found [here](https://gitlab.lrz.de/tum-cps/commonroad-scenario-designer).

## Modification
The original `commonroad-scenario-designer` library was modified because it could not read input map files correctly and could not draw a list of specific points c

## Requirements
The `requirements.txt` file lists the dependencies required to use this project.
These dependencies can be installed by creating a virtual environment at the root of this project and running the command
``pip install -r requirements.txt``
```

**File: ./software/README.md**

```
# Software directory

This directory is part of the CS Bachelor's thesis on the topic "Comparing human and AI behavior in simulated driving scenarios". It contains all the software impl

## Technicalities

The software is working with [CARLA](https://carla.org) driving simulator.

- CARLA Version 0.9.13
- CARLA API Version 0.9.13
- OS: Ubuntu 18.04 and 20.04
- Python 3.6.9

All the required python libraries are listed [here](./python_libraries/requirements.txt).

## Structure

The following is the structure of this software bundle. More details can be found in the respective directories.

- [CARLA scripts](./carla_scripts/) - contains scripts for scenario processing and simulations.
- [Data analytics tools](./data_analysis/) - contains tools for parsing and analysing recorded data.
- [Python libraries](./python_libraries/) - required libraries for the software.
- [Scenario generation tool](./generating_scenarios/) - machine learning algorithm able to estimate the parameters of new scernarios.
- [Testing](./testing/) - directory containing various tests of software components.
```

**File: ./software/testing/README.md**

```
The approach to testing...
```

**File: ./software/generating_scenarios/README.md**

```
## General information

This directory contains scripts needed to generate new driving scenarios.

The data used in learning is stored in the [learning_data](../../data/scenario_generation_data/learning_data/) directory.

Newly generated scenarios go to the [generated_scenarios](../../data/scenario_generation_data/generated_scenarios/) directory.

The `regressor` file contains the algorithm used to train the model and estimate new scenarios.

To create a new scenario, run the `create_scenario.py` file giving flags `-d` specifying the difficulty of the scenario, `-s` specifying the sun's altitude, `-w` s

Example: `python create_scenario.py -d 200 -w 20 -s 30 -f new_scenario`
```

**File: ./software/generating_scenarios/regressor.py**

```python
import pandas as pd
import numpy as np
from sklearn.multioutput import MultiOutputRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
import json

SCENARIO_GENERATION_DATA_PATH = "../../data/scenario_generation_data"

# This class is responsible for initialising the MultiOutputRegressor algorithm which
# is trained on the data contained in scenarios.csv file and is able to estimate new
# scenario parameters given difficulty
class Regressor():

    def __init__(self, randomise):
        # First read the .csv file containing scenario entries
        scenarios_dataframe = pd.read_csv("{}/learning_data/scenarios.csv".format(SCENARIO_GENERATION_DATA_PATH), delimiter = ",")
        random_state = 10 if randomise else None
        # Create a new dataset with only the difficulty as a feature and the rest of the attributes as targets
        X = scenarios_dataframe[["difficulty", "wetness", "sun_altitude_angle"]]
        y = scenarios_dataframe.drop(["difficulty", "wetness", "sun_altitude_angle"], axis=1)
        self.attribute_names = y.columns
        # Split the data into training and testing sets using the standard 80-20 approach
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X, y, test_size=0.1, random_state=random_state)
        # Train the multi-target regression model
        self.model = MultiOutputRegressor(RandomForestRegressor(n_estimators=100, random_state=random_state))
        self.model.fit(self.X_train.values, self.y_train)

    # This method will create a new scenario given the difficulty and then save it to the file
    def create_new_scenario(self, difficulty, wetness, sun, filename):
        attributes = self.predict_scenario_attributes(difficulty, wetness, sun)
        self.write_scenario_to_file(attributes, filename)

    # Given the difficulty, predict what attributes a scenario could have
    def predict_scenario_attributes(self, difficulty, wetness, sun):
        arr = np.array([difficulty, wetness, sun]).reshape(1, -1)
        attributes = self.model.predict(arr)[0]
        rounding = pd.read_csv("{}/learning_data/rounding.csv".format(SCENARIO_GENERATION_DATA_PATH))
        maximums = pd.read_csv("{}/learning_data/maximums.csv".format(SCENARIO_GENERATION_DATA_PATH))
        minimums = pd.read_csv("{}/learning_data/minimums.csv".format(SCENARIO_GENERATION_DATA_PATH))
        rounded_attributes = []
        # For each predicted value, check if it is not exceeding the maximum allowed value and is not lower than the mimum allowed value
        # Also round the final value accordingly
        list_of_attribute_names = self.attribute_names.tolist()
        for i in range(len(attributes)):
            value = self.format_attrbute_value(value = float(attributes[i]), minimum = float(minimums.loc[0, list_of_attribute_names[i]]), maximum = maximums.loc[
            rounded_attributes.append(value)

        # Also add difficulty, wetness and sun_altitude_angle to the final scenario file
        dictionary = dict(zip(self.attribute_names, rounded_attributes))
        dictionary["difficulty"] = self.format_attrbute_value(value = float(difficulty), minimum = float(minimums.loc[0, "difficulty"]), maximum = maximums.loc[0,
        dictionary["wetness"] = self.format_attrbute_value(value = float(wetness), minimum = float(minimums.loc[0, "wetness"]), maximum = maximums.loc[0, "wetness
        dictionary["sun_altitude_angle"] = self.format_attrbute_value(value = float(sun), minimum = float(minimums.loc[0, "sun_altitude_angle"]), maximum = maximu
        return dictionary

    def format_attrbute_value(self, value, minimum, maximum, roundn):
        if value < minimum:
            value = minimum
        if maximum == "inf":
            pass
        else:
            if value > float(maximum):
                value = float(maximum)
        value = int(np.round(value)) if roundn == 0 else np.round(value, roundn)
        return value

    # Create a .json file and save the attribute values there
    def write_scenario_to_file(self, attributes, filename):
        with open("{}/generated_scenarios/{}.json".format(SCENARIO_GENERATION_DATA_PATH, filename), "w+") as f:
            json.dump(attributes, f, indent=4)
```

**File: ./software/generating_scenarios/create_scenario.py**

```python
from regressor import Regressor
import argparse

# The purpose of this script is to create new scenario.json file according to the difficulty, wetness amd sun altitude angle
# and save it in the file specified in the arguments
def main():
    parser = argparse.ArgumentParser(description='Create a new scenario based on difficulty.')
    parser.add_argument('--difficulty', '-d', type=float, default='70', help='Difficulty value')
    parser.add_argument('--sun', '-s', type=float, default='68', help='Sun altitude angle')
    parser.add_argument('--wetness', '-w', type=float, default='10', help='Wetness value')
    parser.add_argument('--filename', '-f', type=str, default='example', help='Filename to save the scenario')
    parser.add_argument('--random', '-r', action='store_false', help='Everytime build and train the algortithm differently')
    args = parser.parse_args()

    try:
        regressor = Regressor(args.random)
        regressor.create_new_scenario(args.difficulty, args.wetness, args.sun, args.filename)
        print("New scenario was created and save into {}.json file.".format(args.filename))
    except Exception as e:
        print("An error occurred trying to create a new scenario:", str(e))

if __name__ == '__main__':
    main()
```

**File: ./README.md**

```markdown
# Testing vehicle safety in simulated driving scenarios

This repository holds data and code related to the CS Bachelor's thesis.

- [Forms](./forms/)
- [Software bundle](./software)
- [Data storage](./data)

---

I verify that I am the sole author of this project, except where explicitly stated to the contrary.

**Author:**

> Vakaris Paulaviius

**Date:**

> 2023-04-03
```

**File: ./.gitignore**

```
# Created by https://www.toptal.com/developers/gitignore/api/windows,linux,macos,intellij,pycharm,visualstudiocode,python
# Edit at https://www.toptal.com/developers/gitignore?templates=windows,linux,macos,intellij,pycharm,visualstudiocode,python

### Intellij ###
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion, Android Studio, WebStorm and Rider
# Reference: https://intellij-support.jetbrains.com/hc/en-us/articles/206544839

# User-specific stuff
.idea/**/workspace.xml
.idea/**/tasks.xml
.idea/**/usage.statistics.xml
.idea/**/dictionaries
.idea/**/shelf

# AWS User-specific
.idea/**/aws.xml

# Generated files
.idea/**/contentModel.xml

# Sensitive or high-churn files
.idea/**/dataSources/
.idea/**/dataSources.ids
.idea/**/dataSources.local.xml
.idea/**/sqlDataSources.xml
.idea/**/dynamic.xml
.idea/**/uiDesigner.xml
.idea/**/dbnavigator.xml

# Gradle
.idea/**/gradle.xml
.idea/**/libraries

# Gradle and Maven with auto-import
# When using Gradle or Maven with auto-import, you should exclude module files,
# since they will be recreated, and may cause churn. Uncomment if using
# auto-import.
# .idea/artifacts
# .idea/compiler.xml
# .idea/jarRepositories.xml
# .idea/modules.xml
# .idea/*.iml
# .idea/modules
# *.iml
# *.ipr

# CMake
cmake-build-*/

# Mongo Explorer plugin
.idea/**/mongoSettings.xml

# File-based project format
*.iws

# IntelliJ
out/

# mpeltonen/sbt-idea plugin
.idea_modules/
```

```
# JIRA plugin
atlassian-ide-plugin.xml

# Cursive Clojure plugin
.idea/replstate.xml

# SonarLint plugin
.idea/sonarlint/

# Crashlytics plugin (for Android Studio and IntelliJ)
com_crashlytics_export_strings.xml
crashlytics.properties
crashlytics-build.properties
fabric.properties

# Editor-based Rest Client
.idea/httpRequests

# Android studio 3.1+ serialized cache file
.idea/caches/build_file_checksums.ser

### Intellij Patch ###
# Comment Reason: https://github.com/joeblau/gitignore.io/issues/186#issuecomment-215987721

# *.iml
# modules.xml
# .idea/misc.xml
# *.ipr

# Sonarlint plugin
# https://plugins.jetbrains.com/plugin/7973-sonarlint
.idea/**/sonarlint/

# SonarQube Plugin
# https://plugins.jetbrains.com/plugin/7238-sonarqube-community-plugin
.idea/**/sonarIssues.xml

# Markdown Navigator plugin
# https://plugins.jetbrains.com/plugin/7896-markdown-navigator-enhanced
.idea/**/markdown-navigator.xml
.idea/**/markdown-navigator-enh.xml
.idea/**/markdown-navigator/

# Cache file creation bug
# See https://youtrack.jetbrains.com/issue/JBR-2257
.idea/$CACHE_FILE$

# CodeStream plugin
# https://plugins.jetbrains.com/plugin/12206-codestream
.idea/codestream.xml

# Azure Toolkit for IntelliJ plugin
# https://plugins.jetbrains.com/plugin/8053-azure-toolkit-for-intellij
.idea/**/azureSettings.xml

### Linux ###
*~

# temporary files which can be created if a process still has a handle open of a deleted file
.fuse_hidden*

# KDE directory preferences
.directory

# Linux trash folder which might appear on any partition or disk
.Trash-*

# .nfs files are created when an open file is removed but is still being accessed
.nfs*

### macOS ###
# General
.DS_Store
.AppleDouble
.LSOverride

# Icon must end with two \r
Icon


# Thumbnails
._*

# Files that might appear in the root of a volume
.DocumentRevisions-V100
.fseventsd
.Spotlight-V100
.TemporaryItems
.Trashes
.VolumeIcon.icns
.com.apple.timemachine.donotpresent

# Directories potentially created on remote AFP share
.AppleDB
.AppleDesktop
Network Trash Folder
Temporary Items
.apdisk

### macOS Patch ###
# iCloud generated files
*.icloud

### PyCharm ###
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion, Android Studio, WebStorm and Rider
# Reference: https://intellij-support.jetbrains.com/hc/en-us/articles/206544839

# User-specific stuff
```

```
# AWS User-specific

# Generated files

# Sensitive or high-churn files

# Gradle

# Gradle and Maven with auto-import
# When using Gradle or Maven with auto-import, you should exclude module files,
# since they will be recreated, and may cause churn. Uncomment if using
# auto-import.
# .idea/artifacts
# .idea/compiler.xml
# .idea/jarRepositories.xml
# .idea/modules.xml
# .idea/*.iml
# .idea/modules
# *.iml
# *.ipr

# CMake

# Mongo Explorer plugin

# File-based project format

# IntelliJ

# mpeltonen/sbt-idea plugin

# JIRA plugin

# Cursive Clojure plugin

# SonarLint plugin

# Crashlytics plugin (for Android Studio and IntelliJ)

# Editor-based Rest Client

# Android studio 3.1+ serialized cache file

### PyCharm Patch ###
# Comment Reason: https://github.com/joeblau/gitignore.io/issues/186#issuecomment-215987721

# *.iml
# modules.xml
# .idea/misc.xml
# *.ipr

# Sonarlint plugin
# https://plugins.jetbrains.com/plugin/7973-sonarlint

# SonarQube Plugin
# https://plugins.jetbrains.com/plugin/7238-sonarqube-community-plugin

# Markdown Navigator plugin
# https://plugins.jetbrains.com/plugin/7896-markdown-navigator-enhanced

# Cache file creation bug
# See https://youtrack.jetbrains.com/issue/JBR-2257

# CodeStream plugin
# https://plugins.jetbrains.com/plugin/12206-codestream

# Azure Toolkit for IntelliJ plugin
# https://plugins.jetbrains.com/plugin/8053-azure-toolkit-for-intellij

### Python ###
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
# Usually these files are written by a python script from a template
# before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt
```

```
# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover
*.py,cover
.hypothesis/
.pytest_cache/
cover/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
db.sqlite3
db.sqlite3-journal

# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
.pybuilder/
target/

# Jupyter Notebook
.ipynb_checkpoints

# IPython
profile_default/
ipython_config.py

# pyenv
# For a library or package, you might want to ignore these files since the code is
# intended to run in multiple environments; otherwise, check them in:
# .python-version

# pipenv
# According to pypa/pipenv#598, it is recommended to include Pipfile.lock in version control.
# However, in case of collaboration, if having platform-specific dependencies or dependencies
# having no cross-platform support, pipenv may install dependencies that don't work, or not
# install all needed dependencies.
#Pipfile.lock

# poetry
# Similar to Pipfile.lock, it is generally recommended to include poetry.lock in version control.
# This is especially recommended for binary packages to ensure reproducibility, and is more
# commonly ignored for libraries.
# https://python-poetry.org/docs/basic-usage/#commit-your-poetrylock-file-to-version-control
#poetry.lock

# pdm
# Similar to Pipfile.lock, it is generally recommended to include pdm.lock in version control.
#pdm.lock
# pdm stores project-wide configurations in .pdm.toml, but it is recommended to not include it
# in version control.
# https://pdm.fming.dev/#use-with-ide
.pdm.toml

# PEP 582; used by e.g. github.com/David-OConnor/pyflow and github.com/pdm-project/pdm
__pypackages__/

# Celery stuff
celerybeat-schedule
celerybeat.pid

# SageMath parsed files
*.sage.py

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json
```

```
# Pyre type checker
.pyre/

# pytype static type analyzer
.pytype/

# Cython debug symbols
cython_debug/

# PyCharm
# JetBrains specific template is maintained in a separate JetBrains.gitignore that can
# be found at https://github.com/github/gitignore/blob/main/Global/JetBrains.gitignore
# and can be added to the global gitignore or merged into this file. For a more nuclear
# option (not recommended) you can uncomment the following to ignore the entire idea folder.
#.idea/

### Python Patch ###
# Poetry local configuration file - https://python-poetry.org/docs/configuration/#local-configuration
poetry.toml

# ruff
.ruff_cache/

# LSP config files
pyrightconfig.json

### VisualStudioCode ###
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
!.vscode/*.code-snippets

# Local History for Visual Studio Code
.history/

# Built Visual Studio Code Extensions
*.vsix

### VisualStudioCode Patch ###
# Ignore all local history of files
.history
.ionide

### Windows ###
# Windows thumbnail cache files
Thumbs.db
Thumbs.db:encryptable
ehthumbs.db
ehthumbs_vista.db

# Dump file
*.stackdump

# Folder config file
[Dd]esktop.ini

# Recycle Bin used on file shares
$RECYCLE.BIN/

# Windows Installer files
*.cab
*.msi
*.msix
*.msm
*.msp

# Windows shortcuts
*.lnk

# End of https://www.toptal.com/developers/gitignore/api/windows,linux,macos,intellij,pycharm,visualstudiocode,python
```

**File: ./.code2pdf**

```
:directories:
- .git
- data
- forms
- software/python_libraries/crdesigner

:files:
- .DS_Store
- __init__.py
- analysis_parameters.json
- scenario_list.json
```