Documentation for y2 traffic simulator

Only relevant files are included in documentation, functions are not in the same order as in the project.

File: main

The file consists of the main function and the part that calls it. The main function is only executed if the file is being called directly. The main function calls first the load_city function, after which it gives the resulting dictionary to the City-class and gives the class to the class GUI

File: city_file_reader.py

Any time the .city-file is incorrect, or the program encounters another error it first closes the file it is reading and raises a CorruptedCityFileError.

load_city

- The main function of the file, it calls every other function that is present in the file and it is called from outside the file.
- It returns a dictionary into which it has added all the information from the .city-file
- Structure:
    - It first calls the open_cityfile function to return a valid file which can be read
    - After receiving a valid file the file is being read line by line and when it detects the beginning of a new block it gives the block to a function, which handles reading the block. After getting to the end of the file it will close the file and then check if every necessary block has been read.

open_cityfile

- Returns an opened file
- Structure:
    - Prompts the user to give a filename for the .city-file with the extension or “q” if the user wants to quit the program. If it cannot find the file it prompts the user again indefinitely, until the user either gives a valid file or exits the program.

read_point

- The function that reads most of the information of the file
- Input is the .city-file, the dictionary used for storing the information and the dictionary key of the block that is being read.
- Structure:
    - It reads every line and checks that the values of the line are as specified in the documentation for the .city-file. After checking the values of a line it appends them into the dictionary that holds all of the information. The place where the input is stored is defined by the key that was one of the input values. If it encounters the header for a block it returns the header to the main file.

read_cars

- When the load_city function encounters a car block it calls this function to read the information of the car templates
- Input for this function is the file and the dictionary where all of the information is being stored.
- Structure:
    - It reads every line and checks that the values of the line are as specified in the documentation for the .city-file. After checking the values of a line it appends them into the dictionary that holds all of the information. If it encounters the header for a block it returns the header to the main file.

read_size / read_interval / read_simulation_speed

- The three functions are basically identical, except that read size has a component that recognizes blank lines between the header and the value.
- Input is the file that is being read
- Returns either a tuple or int depending on the function
- Structure:
    - Reads the line after the header and determines if it is as specified in the documentation. If is returns the value, the read_size function does not care about blank lines, it can read multiple until it finds a valid line, but the read_time and read_interval read just the next line.

class File_reader_tests:

every method in the class use the unittest framework to test a single exit in the city_file_reader file. The methods are named as clearly as possible, while still trying to retain compact names.

class CorruptedCityFileError

A custom error used in the functions that process the .city-file into the dictionary.

class GUI

This class is responsible for building the graphicsScene and periodically updating the whole program

attributes
city: a city object
buildings: a list of coordinates where the buildings are
counter: counts the simulation steps that have passed, is used in adding new cars and reset
on: Boolean value determining if the simulation is running
cars_with_graphicsitems: all car-objects that have an graphicsitem attached to them
timer: timer that calls the update-functions repeatedly

init_window

- This method is called during the init of the class and it adds the window with the graphicsScene to the desktop
- Input is the size of the city
- Structure:
    - Sets up first the window for the scene, after which it sets up the scene. Both sizes are dependent on the dimensions of the city

init_button / toggle_simultion

- Initialises the button that toggles the simulation
- Connects the button to the method toggle_simulation, which switches a variable from True to False and the other way, dependent on the current state of the variable

add_car_graphicsitems

- checks that every car that has been placed onto the scene has a graphicsitem attached to it and if not adds one

simulate_city

- Simulates one step of the city
- checks the state of the variable that toggle_simulation manipulates and if it is True, then it calls the update function for all the cars
- It also periodically adds a car onto the scene and removes the cars which have finished the route they were assigned

update_city

- calls the update function for all graphicsitems and removes the graphicsitems of finished cars

class City

the class stores the information that was gained from the .city-file and is responsible for creating the car objects

attributes
building: list of coordinates for buildings
size: tuple that indicates the size of the city in squares
startpoint: the startpoints found in the file
crosspoint: dictionary of all linked crosspoints
carlist: dictionary that holds the information for all car templates
cars: all current car-objects
interval: the amount of simulation steps between cars spawning
time: the time in milliseconds between each simulation step

the get_something style methods are quite self explanatory, it just returns a attribute and the attribute is indicated in the name.

link_crosspoints

- turns the list of crosspoints into a dictionary, where every crosspoint is a key to an entry and in the entry are all the crosspoints and startpoints it is directly linked to, meaning a straight line between them and no buildings.
- Structure:
    - The method goes into every direction from the origin point and checks if there is a cross/ starting point with those coordinates, if there is it adds it and stops looking into that direction if it does not find anything or hits a wall it does not add anything for that direction

create_car

- Adds a car from the available templates, if there is already a car near the spawnpoint the function will not do anything.
- Structure
    - Takes an random index of the car lists and builds an car-object with that information. Adds the car object to the cars-list

remove_car

- removes a car from the cars-list

class: CarGraphicsItem

attributes
car: the car-object it is linked to
color: the colour of the car as a QColor

set_color
- converts the written colour into QColor
- Returns the color
- Structure:
    - tests for the different words designating the colour and if it doesn't find a match it returns a default colour

construct_car_outline
- constructs the outline of the car and sets the turning point of the car to be in the middle of the shape

update_all
- calls both update functions

def update_position
- updates the position of the graphicsitem to match the position of the car, it is a bit off centered to move the center of the car to the same point it turns on

def update_rotation
- rotates the graphicsitem to match the rotation of the car

Car

builds the car-object and is responsible for pathfinding and driving

attributes:
finished: Boolean value has the car reached the final point of its path
color: the colour of the car
path: list of coordinates designating the path of the car between startingpoint and endpoint
orientation: the orientation of the car as a unit vector
position: the position of the car
count_point: int of how which index is next on the path
target: current coordinates the car goes towards
outline: 4 points outlining the car
mass: static value to represent the mass of the car
velocity: a vector representing the velocity of the car
max_force: maximum amount of force the car can use to navigate
max_speed: maximum amount of speed the car can have, measured in pixels/simulation step

get-functions
- quite self explanatory
- those with a bit more code duplicate a list so that the original won't be affected by any changes to the returned list
- with get_tot_velocity it calculates the length of the sum vector of the velocity vectors

update_target:
- sets the target to the next index in the path-list
- if the car has reached the final point the attribute finished is changed to True

seek_target:
- Moves the car towards the specified point, follows quite closely the simple vehicle model
- Input: a steering vector
- Structure:
    - Forms a slight drag force, then adds the drag and the steering vector together, after which the velocity is changed according to Newtons 2. law. If the total velocity is under a 0.9 the orientation of the car is not changed. Else a normalized velocity vector becomes the new orientation. Also the velocity is adjusted that it is always lower than the speed limit

move_position:
- Adds a vector to a point moving the point
- Input: point, vector
- returns the moved point

offset_target:

- offsets a target along the normal of the direction vector by a specified amount
- Input point, amount
- Returns the point

multiply_vector:

- multiplies the vector with a number
- Input: vector, amount
- Returns: vector

form_car_outline:

- Forms the four points that form the outline of the car
- Returns a tuple with all of the positions of the points

form_building_outline:

- Forms the four lines that form the outline of a building
- Returns a list with four two point lists

construct_front_feelers:

- Forms three lines that allow the car to see in front
- Input: outline of the car
- Returns: three points that are the ends of the feelers

construct_side_feelers:

- Constructs two lines parallel to the car that extend a bit past the edges that allow the car to see into the close viscinity beside it
- Input: outline of the car
- Returns: a list with two two-point lists

normalize_vector:

- normalizes a vector to the length of one
- Input: vector
- Returns: vector

orientation / check_if_intersect:

- checks if two lines cross each other basis from:
    - https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
- Input for check_if_intersect: four points representing the two lines
- Returns: True or False
- Structure:
    - Checks the rotation of any three-point combination, where the first two are from the same line. If both orientations for both lines are different from the other result the line got the lines intersect.

braking:

- inverts the steering vector if the speed is over 1, otherwise sets the steering vector to 0 and sets the speed of the car to 0
- Input: Steering vector
- Output: Steering vector

turning:

- turns the steering vector by x degrees around its origin
- Input: steering vector, amount in deg
- Returns: steering vector

define_behaviour:

- Defines the direction of the steering vector that is given to seek_target, this method accounts for dodging obstacles, braking, and steering in general
- Input: the cars list that are on the scene, list of buildings on the scene
- Structure
    - The method first constructs all of the feelers and the current outline of the car. After that it builds the default steering vector by offsetting the target by 35 and then shortening the vector from its position to the offset target to the length of max_force.
    - After that it searches for all cars in a certain radius and places them into a list, so in the next step it does a few less iterations
    - After that it starts to iterate over all of the cars and checking if any of the feelers intersect with the cars outline, if yes, it checks the cars orientation and compares to its own then either tries to avoid the other car by turning the steering vector more to the right or by braking.
    - An exception is if the feeler on the right notice a car, it the car is coming into the opposite direction it tries to go to the left, and in the other cases stop
    - It also checks if any buildings intersect with the feelers and if yes tries to avoid them to the left
    - finally it calls the method seek_target and gives it the steering vector

update_car:

- calls the methods for each simulation step
- Input: list of cars, list of buildings
- Structure:
    - it checks if the car is in need to change the target and if yes changes it. Afterwards it calls the define_behaviour method and lasly updates the outline of the car

form_path:

- Forms a path for the car
- Input: startpoint, stoppoint, the dictionary with all the linked points
- Returns: list with points
- Structure:
    - starts to add all of the possible linking points for each step beginning from the stoppoint, until it reaches the startpoint. Then it cleans all of the useless points leaving only one path between the points.

define_starting_orientation:

- Sets the cars starting orientation so that the front points towards the target
- Input: the path of the car
- Structure:
    - Checks into which direction the car has to first move and sets the car into the cardinal direction that matches the direction.

define_starting_position:

- Offsets the starting position of the car so that the startpoint is on the road a bit offset to the cars right of the center
- Input: the starting orientation of the car
- Structure:
    - First moves the car to the centre of the tile, after which it moves it a bit right according to the starting direction.

Missing features:

- The cars still have considerable blind spots on their sides, which affect moving especially when there is a need to cross the driving line of another car
- collision detection is not implemented.