

Final Project, SI1336

Carl Schiller, 9705266436

March 8, 2019

Abstract

The effects of ramp meters on freeway on ramps were studied on a simulated freeway in Roslags-Näsby, Sweden. Model of road was made using a directed graph with vertices spaced approximately every 30 meters. Cars were simulated by traversing the graph with a time step size of 1/60th of a second. No significant increase in freeway flow was noticed by deploying ramp meters on the on-ramp.

Contents

1	Introduction	3
1.1	Problem formulation	3
1.2	Complex systems	3
2	Method	4
2.1	Graphs	4
2.2	Discretization	4
2.2.1	Speed	4
2.2.2	Spawn rate and car headway	4
2.2.3	Collision detection	5
2.2.4	Acceleration	5
2.2.5	Overtake logic and merging	5
2.3	Graphics rendering	6
3	Result	6
3.1	Parameters	6
3.2	Null hypothesis	7
3.3	Plots	7
4	Discussion	8
4.1	Considerations for further research	8
A	Header files	9
A.1	button.h	9
A.2	cars.h	10
A.3	cscreen.h	11
A.4	road.h	11
A.5	roadnode.h	12
A.6	roadsegment.h	13
A.7	screen0.h	14
A.8	screen1.h	14
A.9	screen2.h	15
A.10	screen3.h	15
A.11	screens.h	16

A.12	simulation.h	16
A.13	simulation2.h	16
A.14	traffic.h	17
A.15	unittests.h	18
A.16	util.h	19
B	Source files	20
B.1	button.cpp	20
B.2	cars.cpp	23
B.3	main.cpp	32
B.4	road.cpp	33
B.5	roadnode.cpp	36
B.6	roadsegment.cpp	38
B.7	screen0.cpp	41
B.8	screen1.cpp	43
B.9	screen2.cpp	45
B.10	screen3.cpp	48
B.11	simulation.cpp	50
B.12	simulation2.cpp	51
B.13	traffic.cpp	53
B.14	unittests.cpp	61
B.15	util.cpp	63
C	Matlab	66



Figure 1: A typical ramp meter, image courtesy of [4]

1 Introduction

1.1 Problem formulation

This project is intended to simulate the traffic flow effect of a *time fixed ramp meter* a freeway on-ramp in Roslags Näsby trafikplats, Sweden. A *ramp meter* is a device that manages the flow of traffic onto the freeway, an example of a *ramp meter* can be seen in figure 1. More specifically, a *time fixed ramp meter* that only allow one car per green signal period will be examined. There are also more active variants of *ramp meters* which measure gaps in the traffic on the freeway to determine when to release vehicles, but this is beyond the scope of this project. Ramp metering systems have successfully been proven to decrease congestion and reduce travel time on freeways. [5]

1.2 Complex systems

Traffic flow is a typical example of a complex system. As described in *An Introduction to Computer Simulation Methods Third Edition (revised)*, traffic flow can be simulated by modelling the system as a *Cellular Automaton*. A *Cellular Automaton* is a grid lattice which changes state on each tick based on rules and the current configuration of the lattice. [3]

2 Method

Cellular Automata was determined to not be satisfactory when trying to model the flow of the freeway. This is because lane change and collision detection worked poorly on a grid lattice in two dimensions. Another approach was considered instead.

2.1 Graphs

In order to model the road with several lanes, a *directed graph* was implemented with blocks of vertices as lanes, with directed edges as paths for the cars to drive. In other terms, cars drive on "rails" and can only change lanes on specified vertices, as can be seen in figure 2. [2]

When using a *directed graph* instead of a grid lattice, collision avoidance becomes a lot easier to implement. Time complexity also decreases, which improves simulation performance. The collision avoidance method implemented is $\mathcal{O}(n \cdot m^2)$, where n is the amount of cars and m is the search area. The grid lattice as previously mentioned had dimensions 550x600, which was replaced by a graph with approximately 140 edges which improved performance by approximately 2000 times (if the whole system is searched for potential obstructions i.e. other cars).

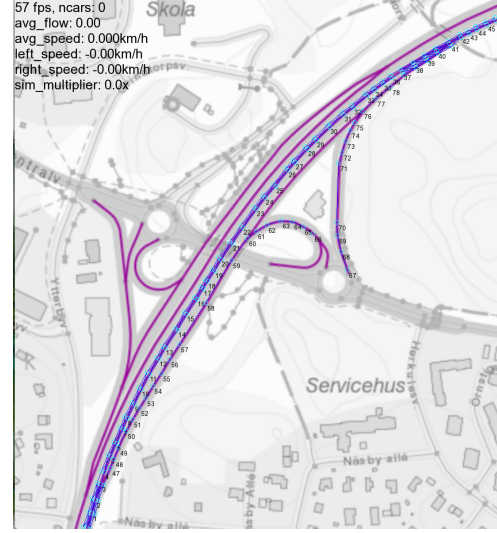


Figure 2: Setup of road with vertices and edges.

2.2 Discretization

In contrast to *Cellular Automata* there is no grid discretization, and thus the cars run on continuous "tracks". The distance traveled by each car is determined by the individual car's speed and the system wide time step size. Another benefit from the *directed graph* implementation is that the directions of the cars is not required as a parameter. All that is needed in order to simulate a car is the speed and the distance to the next vertex as well as knowing which vertex the car originated from. When stepping in time the distance traveled is subtracted from the distance to the next vertex, and when the car has reached the next vertex a new target vertex is selected.

Cars make decisions independently according to simple rules, and generates a complex behavior when interacting with each other i.e. braking or changing lanes. Some parameters are tweakable without changing the code, and each parameter influences the simulation in different ways.

2.2.1 Speed

The cars' speed is determined by a mean speed multiplied by a normally distributed variable $x \in N(1, \sigma)$, which is referred to in the code as "m_aggressiveness". "m_aggressiveness" is also involved collision detection and to determine when to overtake the car in front. σ is user tweakable.

2.2.2 Spawn rate and car headway

Cars appear in two segments, either on the on-ramp or on the beginning of the freeway. The rate of which cars appear on freeways is determined by a gamma distribution with probability density function according to equation 1. [1]

$$f(x) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-x/\beta} \quad (1)$$

where α is the "shape" factor and β is the "rate" factor which are tweakable according to which behavior is sought after. The expected mean of a stochastic variable is $\alpha\beta$, with variance $\alpha\beta^2$. This means, a larger β implies a more spread out function.

2.2.3 Collision detection

If a car is too close to a car in front, the speed is reduced according the following rules.

```

1      if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
2          m_speed -= std::min(std::max((radius_to_car-min_distance)*0.5f,0.0f),10.0f*delta_t
3      );
4      }
5      else if(radius_to_car < min_distance){
6          m_speed -= std::min(std::max((min_distance-radius_to_car)*0.5f,0.0f),2.0f*delta_t)
7      ;
8      }
9      else if(delta_speed < 0 && radius_to_car < detection_distance){
10         m_speed -= std::min(
11             abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
12             m_aggressiveness * 0.15f,
13             10.0f * delta_t);
14     }

```

This ensures that a car slowly approaches the car in front. The first if statement guarantees that it will not surpass the "min_distance" distance, because the speed reduction follows this diverging sum.

$$d - \sum_{n=2}^{\infty} \frac{d}{n^2} = 0 \quad (2)$$

where d is "radius_to_car-min_distance".

2.2.4 Acceleration

If no obstruction is in the way, a car will accelerate according to:

```

1      float target = m_target_speed;
2      float d_vel; // proportional control.
3
4      if(m_speed < target*0.75){
5          d_vel = m_aggressiveness*elapsed*2.0f;
6      }
7      else{
8          d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
9      }
10
11     m_speed += d_vel;

```

2.2.5 Overtake logic and merging

A car decides to overtake another car if the following conditions are met.

```

1      //see if we want to overtake car.
2
3      if(closest_car != nullptr){
4          //float delta_speed = closest_car->speed()-speed();
5          float delta_distance = Util::distance_to_car(this, closest_car);
6
7          if(overtake_this_car == nullptr){

```

```

9         if(delta_distance > m_min_overtake_dist_trigger && delta_distance <
10            m_max_overtake_dist_trigger && (target_speed()/closest_car->target_speed() >
11            m_aggressiveness*1.0f) && current_lane == 0 && closest_car->current_node->
12            get_parent_segment()->get_lane_number(closest_car->current_node) == 0){
13             overtake_this_car = closest_car;
14         }
15     }
16 }
17
18 if(overtake_this_car != nullptr){
19     if(Util::is_car_behind(overtake_this_car, this) && (Util::distance_to_car(this,
20     overtake_this_car) > m_overtake_done_dist)){
21         overtake_this_car = nullptr;
22     }
23 }

```

A car will not merge if another car is occupying the lane it want to switch too.

2.3 Graphics rendering

When tweaking parameters involved in the cars' descision making, it is hard to get an overview of how each parameter influences the system wide behavior of the traffic. Thus a lot of effort has been spent on developing a graphical interface that shows how the traffic flows in the given configuration of parameters. An example of a test run is shown in the link below. <https://youtu.be/I7Jx8SScYZ8>

3 Result

3.1 Parameters

The following parameters have been used in the simulation. By varying Lane 1 α and Lane 2 α (the rate of which cars spawn on the freeway), the effect of a ramp meter on the system flow was determined. This was done by simulating the flow of different spawn rates with a ramp meter and without a ramp meter.

Agressiveness	1.0
Agressiveness σ	0.2
Global β	1.5
Mean speed	20 (m/s)
Lane 0 α	4.0
Lane 1 α	2.0 to 0.1 with step 0.1
Lane 2 α	2.0 to 0.1 with step 0.1
Ramp 0 α	4.0
Minimum distance to car in front	8.0 (m)
Minimum overtake distance cutoff	10.0 (m)
Maximum overtake distance cutoff	40.0 (m)
Overtake distance shutoff	30.0 (m)
Minimum merge distance	15.0 (m)
Radial search distance	30.0 (m)
Search distance forward	50.0 (m)
Time step	1/60.0 (s)
Ramp meter period	6.0 (s)

Table 1: Parameters used

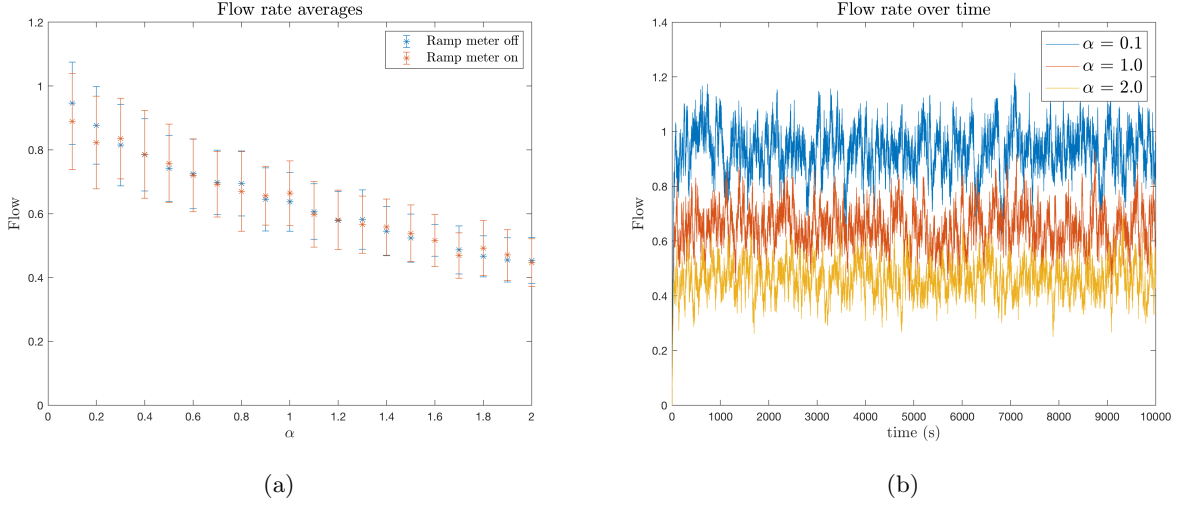


Figure 3: 3a) Fundamental diagram of flow as a function of α with time step 1/60 seconds. Total 60000 steps. Errorbars represent $\pm\sigma$ of deviation in flow. 3b) Flow versus time of different α with time step 1/60 seconds, no ramp meter applied. Total of 600000 steps

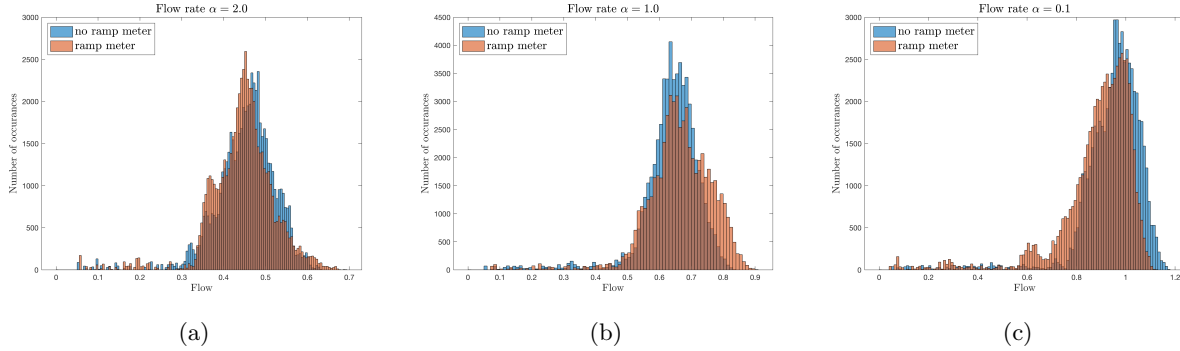


Figure 4: Histograms of flow for selected values of α . Time step 1/60 seconds, total of 60000 steps

3.2 Null hypothesis

Let X be the stochastic variable associated with the mean flow of traffic without a ramp meter over all simulated time. I.e, the outcome of X is the mean flow of a simulation at a given α . Let Y be the stochastic variable associated with the mean flow of traffic with a ramp meter over all simulated time. Then formulate the null hypothesis:

$$\begin{aligned} H_0 : X \text{ and } Y \text{ has the same distribution} \\ H_1 : X\text{'s distribution is skewed in relation to } Y \end{aligned} \quad (3)$$

3.3 Plots

Figure 3b has characteristics typical of stop-and-go traffic for higher densities. The fluctuations of the flow increases as the expected spawn time $\alpha\beta$ decreases, which also can be seen by looking at figure 3a. The error bars are larger for smaller α , which indicates a larger standard deviation from the mean flow. By using Wilcoxon's rank sum test with equation 3 in mind, the p-value of the means in figure 3a is $p = 0.9887$.

4 Discussion

Since $p = 0.9887 > 0.05$ the null hypothesis as formulated in equation 3 can not be rejected. I.e. there is no significant difference between using a ramp meter and not using a ramp meter on a 95 % confidence level with the configuration as given in table 1. Although for some specific values of α as can be seen in figure 4 a ramp meter allows for better flow.

4.1 Considerations for further research

In this study only one parameter has been examined in table 1, and there might be configurations where a ramp meter allows for better flow over all. The way cars overtake, merge, and avoid cars is also might not be a realistic representation of how cars behave. There might be better ways to model the merging, especially in the merging segment where the on-ramp connects to the freeway.

It is also worth mentioning that flow was defined as the sum of all cars divided by the total road length. That is, the whole system's flow was considered. If the flow instead was defined as the flow on the freeway only, and not the on-ramp, the result might have been different. This depends on what matters more, the total flow in the whole system or the flow on the freeway only.

References

- [1] Ahmed Abdel-Rahim. *CE571: Traffic Flow Theory - Spring 2011*. English (United States), en-US. URL: <https://www.webpages.uidaho.edu/ce571/class%20notes/Week%20%20modeling%20headway%20distribution%202011.pdf> (visited on 03/07/2019).
- [2] *Gerichteter Graph*. de. Page Version ID: 179253516. July 2018. URL: https://de.wikipedia.org/w/index.php?title=Gerichteter_Graph&oldid=179253516 (visited on 03/05/2019).
- [3] H Gould, J Tobochnik, and W Christian. "Introduction to Computer Simulation Methods". In: (), p. 797.
- [4] Patriarca12. *English: Ramp meter on ramp from Miller Park Way to Interstate 94 east in Milwaukee, Wisconsin, USA*. July 2008. URL: https://commons.wikimedia.org/wiki/File:Ramp_meter_from_Miller_Park_Way_to_I-94_east_in_Milwaukee.jpg (visited on 03/05/2019).
- [5] U.S. Department of Transportation, Federal Highway Administration. *Ramp Metering: A Proven, Cost-Effective Operational Strategy - A Primer: 1. Overview of Ramp Metering*. URL: <https://ops.fhwa.dot.gov/publications/fhwahop14020/sec1.htm> (visited on 03/05/2019).

A Header files

A.1 button.h

```
1 //
2 // Created by Carl Schiller on 2019-03-05.
3 //
4
5 #ifndef HIGHWAY_BUTTON_H
6 #define HIGHWAY_BUTTON_H
7
8 #include "SFML/Graphics.hpp"
9 #include <string>
10
11 class Button : public sf::Drawable, public sf::Transformable{
12 private:
13     sf::Font font;
14     sf::RectangleShape rect;
15     sf::Text text;
16     sf::Color normal;
17     sf::Color pressed;
18
19     bool is_mouse_in_rect(sf::RenderWindow & App);
20 public:
21     Button(sf::Font & font_copy, unsigned int font_size, int x_pos, int y_pos, const std::
        string & name, sf::Color button_col, sf::Color text_col, sf::Color pressed);
22
23     void center_text();
24     void set_origin(int x, int y);
25     void set_dim(int x, int y);
26     bool clicked(sf::RenderWindow & App);
27     sf::FloatRect get_bounds();
28     void set_text(const std::string & name);
29
30     virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
31 };
32
33 class Input : public sf::Drawable, public sf::Transformable{
34 private:
35     sf::Font font;
36     sf::RectangleShape rect;
37     sf::Text text;
38     std::string string;
39     std::string input;
40     sf::Color normal;
41     sf::Color pressed;
42     sf::Color typing;
43
44     bool bool_typing;
45
46 public:
47     Input(sf::Font & font_copy, unsigned int font_size, int x_pos, int y_pos,
        const std::string & name, sf::Color button_col, sf::Color text_col, sf::Color
        pressed,
48         sf::Color typing, std::string val);
49
50     bool is_mouse_in_rect(sf::RenderWindow & App);
51     void center_text();
52     void set_origin(int x, int y);
53     void set_dim(int x, int y);
54     virtual Input * clicked(sf::RenderWindow & App);
55     Input * inputing(sf::RenderWindow & App, std::string & str);
56     float get_val();
57     sf::FloatRect get_bounds();
58     const sf::Vector2f get_pos();
59
60 }
```

```

        friend class Button_bool;
63     virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
65 };
67 class Button_bool : public Input {
        bool toggled;
69 public:
        void set_toggled(bool tog);
71     using Input::Input;
        virtual Button_bool * clicked(sf::RenderWindow & App);
73     bool get_bool();
};
75 #endif //HIGHWAY.BUTTON.H

```

../highway/headers/button.h

A.2 cars.h

```

//
2 // Created by Carl Schiller on 2019-03-04.
//
4
6 #ifndef HIGHWAY_CAR.H
#define HIGHWAY_CAR.H
8
10 // Car
12 // Describes a car that moves around in Road class
14 //
16 #include <map>
17 #include "roadnode.h"
18 #include "roadsegment.h"
20 class Car{
21 private:
22     float m_dist_to_next_node;
23     float m_speed;
24     float m_theta; // radians
26     float m_aggressiveness; // how fast to accelerate;
27     float m_target_speed;
28
29     const float m_min_dist_to_car_in_front;
30     const float m_min_overtake_dist_trigger;
31     const float m_max_overtake_dist_trigger;
32     const float m_overtake_done_dist;
33     const float m_merge_min_dist;
34     const float m_search_radius_around;
35     const float m_search_radius_to_car_in_front;
36
37 public:
38     Car();
39     ~Car();
40     Car& operator=(const Car&) = default;
42     Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float
    agresivness,
        float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
    m_max_overtake_dist_trigger,
44     float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,

```

```

46         float m_search_radius_to_car_in_front);
Car(RoadSegment * spawn_point, RoadNode * lane, float vel, float target_speed, float
agressivness,
    float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
m_max_overtake_dist_trigger,
48     float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
    float m_search_radius_to_car_in_front);
50
// all are raw pointers
52 RoadSegment * current_segment;
RoadNode * current_node;
54 RoadNode * heading_to_node;
Car * overtake_this_car;
56
void update_pos(float delta_t);
58 void merge(std::vector<RoadNode*> & connections);
void do_we_want_to_overtake(Car * & closest_car, int & current_lane);
60 void accelerate(float delta_t);
void avoid_collision(float delta_t);
62 Car * find_closest_car_ahead();
std::map<Car *, bool> find_cars_around_car();
64
float x_pos();
66 float y_pos();
68
float & speed();
float & target_speed();
70 float & theta();
72
RoadSegment * get_segment();
};
74
#endif //HIGHWAY_CAR_H

```

../highway/headers/car.h

A.3 cscreen.h

```

1 //
// Created by Carl Schiller on 2019-03-04.
3 //
5 #ifndef HIGHWAY_CSCREEN_H
#define HIGHWAY_CSCREEN_H
7
#include "SFML/Graphics.hpp"
9 #include <vector>
11 class cScreen{
public:
13     //virtual int Run(sf::RenderWindow & App) = 0;
    virtual int Run(sf::RenderWindow & App, std::vector<float> * args, std::vector<bool> *
bargs) = 0;
15 };
17 #endif //HIGHWAY_CSCREEN_H

```

../highway/headers/cscreen.h

A.4 road.h

```

1 //
// Created by Carl Schiller on 2019-03-04.
3 //

```

```

5 #ifndef HIGHWAY_ROAD_H
6 #define HIGHWAY_ROAD_H
7
8 ///////////////////////////////////////////////////////////////////
9 //
10 // Road
11 //
12 // Describes a road with interconnected nodes. Mathematically it is
13 // a graph.
14 //
15 ///////////////////////////////////////////////////////////////////
16
17 #include "roadsegment.h"
18 #include <vector>
19 #include <string>
20
21 class Road{
22 private:
23     std::vector<RoadSegment*> m_segments; // OWNERSHIP
24     std::vector<RoadSegment*> m_spawn_positions; // raw pointers
25     std::vector<RoadSegment*> m_despawn_positions; // raw pointers
26
27     const std::string M_FILENAME;
28 private:
29     Road();
30     ~Road();
31 public:
32     static Road &shared() {static Road road; return road;} // in order to only load road
33     // once in memory
34
35     Road(const Road& copy) = delete; // no copying allowed
36     Road& operator=(const Road& rhs) = delete; // no copying allowed
37
38     bool load_road();
39     std::vector<RoadSegment*> & spawn_positions();
40     std::vector<RoadSegment*> & despawn_positions();
41     std::vector<RoadSegment*> & segments();
42     RoadSegment * ramp_meter_position;
43 };
44 #endif //HIGHWAY_ROAD_H

```

../highway/headers/road.h

A.5 roadnode.h

```

1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #ifndef HIGHWAY_ROADNODE_H
6 #define HIGHWAY_ROADNODE_H
7
8 ///////////////////////////////////////////////////////////////////
9 //
10 // RoadNode
11 //
12 // Describes the smallest element in Road, it is similar to
13 // that of a mathematical graph with nodes and edges.
14 //
15 ///////////////////////////////////////////////////////////////////
16
17 #include <vector>
18 #include "car.h"
19 #include "roadsegment.h"

```

```

20 class RoadNode{
21 private:
22     float m_x, m_y;
23     std::vector<RoadNode*> m_nodes_from_me; // raw pointers, no ownership
24     std::vector<RoadNode*> m_nodes_to_me;
25     RoadSegment* m_is_child_of; // raw pointer, no ownership
26 public:
27     RoadNode();
28     ~RoadNode();
29     RoadNode(float x, float y, RoadSegment * segment);
30
31     void set_next_node(RoadNode *);
32     void set_previous_node(RoadNode *);
33     RoadSegment* get_parent_segment();
34     RoadNode * get_next_node(int lane);
35     std::vector<RoadNode*> & get_nodes_from_me();
36     std::vector<RoadNode*> & get_nodes_to_me();
37     float get_x();
38     float get_y();
39     float get_theta(RoadNode*);
40 };
41
42
43 #endif //HIGHWAY_ROADNODE.H

```

../highway/headers/roadnode.h

A.6 roadsegment.h

```

//
// Created by Carl Schiller on 2019-03-04.
//
4
#ifndef HIGHWAY_ROADSEGMENT_H
#define HIGHWAY_ROADSEGMENT_H
8
////////////////////////////////////////////////////
//
10 // RoadSegment
//
12 // Describes a container for several RoadNodes
//
14 ////////////////////////////////////////////////////
16 #include <vector>
18 class RoadNode;
20 class Car;
22 class RoadSegment{
23 private:
24     const float m_x, m_y;
25     float m_theta;
26     const int m_n_lanes;
27
28     constexpr static float MLANE_WIDTH = 4.0f;
29
30     std::vector<RoadNode*> m_nodes; // OWNERSHIP
31     RoadSegment * m_next_segment; // raw pointer, no ownership
32 public:
33     RoadSegment() = delete;
34     RoadSegment(float x, float y, RoadSegment * next_segment, int lanes);
35     RoadSegment(float x, float y, float theta, int lanes);
36     RoadSegment(float x, float y, int lanes, bool merge);

```

```

38 ~RoadSegment(); // rule of three
RoadSegment(const RoadSegment&) = delete; // rule of three
RoadSegment& operator=(const RoadSegment& rhs) = delete; // rule of three

40
42 bool merge;
std::vector<Car*> m_cars; // raw pointer, no ownership
44 float ramp_counter;
bool car_passed;
46 bool meter;
float period;

48 RoadNode * get_node_pointer(int n);
std::vector<RoadNode *> get_nodes();
50 void append_car(Car*);
void remove_car(Car*);
52 RoadSegment * next_segment();
float get_theta();
54 const float get_x() const;
const float get_y() const;

56
int get_lane_number(RoadNode *);
58 const int get_total_amount_of_lanes() const;
void set_theta(float theta);
60 void set_next_road_segment(RoadSegment*);
void calculate_theta();
62 void calculate_and_populate_nodes();
void set_all_node_pointers_to_next_segment();
64 void set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *);
};
66 #endif //HIGHWAY_ROADSEGMENT.H

```

../highway/headers/roadsegment.h

A.7 screen0.h

```

1 //
// Created by Carl Schiller on 2019-03-04.
3 //

5 #ifndef HIGHWAY_SCREEN0.H
#define HIGHWAY_SCREEN0.H
7
9 #include "cscreen.h"
11 class screen_0 : public cScreen{
public:
    screen_0();
13     virtual int Run(sf::RenderWindow & App, std::vector<float> * args, std::vector<bool> *
        bargs);
};
15 #endif //HIGHWAY_SCREEN0.H

```

../highway/headers/screen0.h

A.8 screen1.h

```

2 //
// Created by Carl Schiller on 2019-03-04.
//
4
6 #ifndef HIGHWAY_SCREEN1.H
#define HIGHWAY_SCREEN1.H

```

```

8 #include "cscreen.h"
10 class screen_1 : public cScreen{
12 public:
14     screen_1();
15     virtual int Run(sf::RenderWindow & App, std::vector<float> * args, std::vector<bool> *
16     bargs);
17 #endif //HIGHWAY_SCREEN1.H

```

../highway/headers/screen1.h

A.9 screen2.h

```

2 //
3 // Created by Carl Schiller on 2019-03-05.
4 //
5 #ifndef HIGHWAY_SCREEN2.H
6 #define HIGHWAY_SCREEN2.H
7
8 #include "cscreen.h"
9
10 class screen_2 : public cScreen{
12 public:
14     screen_2();
15     virtual int Run(sf::RenderWindow & App, std::vector<float> * args, std::vector<bool> *
16     bargs);
17 #endif //HIGHWAY_SCREEN2.H

```

../highway/headers/screen2.h

A.10 screen3.h

```

1 //
2 // Created by Carl Schiller on 2019-03-06.
3 //
4
5 #ifndef HIGHWAY_SCREEN3.H
6 #define HIGHWAY_SCREEN3.H
7
8 #include "cscreen.h"
9 #include "traffic.h"
10
11 class screen_3 : public cScreen{
12 private:
13     bool run_bool;
14     long sim_time;
15     long frame_rate;
16 public:
17     screen_3();
18     virtual int Run(sf::RenderWindow & App, std::vector<float> * args, std::vector<bool> *
19     bargs);
20 #endif //HIGHWAY_SCREEN3.H

```

../highway/headers/screen3.h

A.11 screens.h

```
1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #ifndef HIGHWAY_MAINMENU_H
6 #define HIGHWAY_MAINMENU_H
7
8 #include "cscreen.h"
9
10 #include "screen0.h"
11 #include "screen1.h"
12 #include "screen2.h"
13 #include "screen3.h"
14
15 #endif //HIGHWAY_MAINMENU_H
```

../highway/headers/screens.h

A.12 simulation.h

```
1 //
2 // Created by Carl Schiller on 2019-03-01.
3 //
4
5 #ifndef HIGHWAY_WINDOW_H
6 #define HIGHWAY_WINDOW_H
7
8 //////////////////////////////////////
9 //
10 // Simulation
11 //
12 // Describes how to simulate Traffic class
13 //
14 //////////////////////////////////////
15
16 #include <vector>
17 #include "SFML/Graphics.hpp"
18 #include "traffic.h"
19
20 class Simulation{
21 private:
22     sf::Mutex * m_mutex;
23     Traffic * m_traffic;
24     bool * m_exit_bool;
25     const int M_SIM_SPEED;
26     const int M_FRAMERATE;
27 public:
28     Simulation() = delete;
29     Simulation(Traffic *& traffic, sf::Mutex *& mutex, int sim_speed, int m_framerate, bool
        *& exitbool);
30
31     void update();
32 };
33
34
35 #endif //HIGHWAY_WINDOW_H
```

../highway/headers/simulation.h

A.13 simulation2.h


```

1 //
2 // Created by Carl Schiller on 2019-03-06.
3 //
4
5 #ifndef HIGHWAY_SIMULATION2_H
6 #define HIGHWAY_SIMULATION2_H
7
8 #include <vector>
9 #include "SFML/Graphics.hpp"
10 #include "traffic.h"
11
12 class Sim{
13 private:
14     Traffic * m_traffic;
15     bool * m_finish_bool;
16     const long MFRAMERATE;
17     long * sim_time;
18     int * m_percent;
19 public:
20     Sim() = delete;
21     Sim(Traffic *& traffic, int m_framerate, long * time, bool * exitbool, int * percent);
22
23     void update();
24     void print_to_file(std::vector<double> * vec, long time_steps);
25 };
26
27
28
29 #endif //HIGHWAY_SIMULATION2_H

```

../highway/headers/simulation2.h

A.14 traffic.h

```

1 //
2 // Created by Carl Schiller on 2019-03-01.
3 //
4
5 #ifndef HIGHWAY_TRAFFIC_H
6 #define HIGHWAY_TRAFFIC_H
7
8 ///////////////////////////////////////////////////////////////////
9 //
10 // Traffic
11 //
12 // Describes the whole traffic situation with Cars and a Road.
13 // Inherits form SFML Graphics.hpp in order to render the cars.
14 //
15 ///////////////////////////////////////////////////////////////////
16
17 #include <random>
18 #include <vector>
19 #include "SFML/Graphics.hpp"
20 #include "car.h"
21
22 class Traffic : public sf::Drawable, public sf::Transformable{
23 private:
24     std::vector<Car*> m_cars;
25     bool debug;
26     std::mt19937 & my_engine();
27     sf::Font m_font;
28
29     const float m_aggro;
30     const float m_aggro_sigma;
31     const float m_spawn_freq;

```

```

33     const float m_speed;

34     const float m_lane_0_spawn_prob;
35     const float m_lane_1_spawn_prob;
36     const float m_lane_2_spawn_prob;
37     const float m_ramp_0_spawn_prob;

38     const float m_min_dist_to_car_in_front;
39     const float m_min_overtake_dist_trigger;
40     const float m_max_overtake_dist_trigger;
41     const float m_overtake_done_dist;
42     const float m_merge_min_dist;
43     const float m_search_radius_around;
44     const float m_search_radius_to_car_in_front;

45     const float m_ramp_meter_period;
46     const bool m_ramp_meter;

47     float road_length;

48     std::vector<float> probs;
49 public:
50     Traffic() = delete;
51     Traffic(std::vector<bool> bargs, std::vector<float> args);
52     ~Traffic();
53     Traffic(const Traffic&); // rule of three
54     Traffic& operator=(const Traffic&); // rule of three

55     unsigned long n_of_cars();
56     void spawn_cars(std::vector<double*> & counters, float elapsed);
57     void despawn_cars();
58     void despawn_all_cars();
59     void despawn_car(Car*& car);
60     void force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, float
        aggro);

61     void update(float elapsed_time);
62     std::vector<Car *> get_car_copies() const;
63     float get_avg_flow();
64     std::vector<float> get_avg_speeds();
65 private:
66     virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
67 public:
68     void get_info(sf::Text & text, sf::Time &elapsed);
69     double m_multiplier;
70 };
71 #endif //HIGHWAY_TRAFFIC_H

```

../highway/headers/traffic.h

A.15 unittests.h

```

1 //
2 // Created by Carl Schiller on 2019-03-01.
3 //
4
5 #ifndef HIGHWAY_UNITTESTS_H
6 #define HIGHWAY_UNITTESTS_H
7
8 ////////////////////////////////////////////////////////////////////
9 //
10 // Tests
11 //
12 //

```

```

13 // Testing the various functions. //
14 // //
15 ////////////////////////////////////////////////////////////////////
16
17 #include "traffic.h"
18 #include "SFML/Graphics.hpp"
19
20 class Tests{
21 private:
22     Traffic * m_traffic;
23     sf::Mutex * m_mutex;
24     void placement_test();
25     void delete_cars_test();
26     void run_one_car();
27     void placement_test_2();
28     void placement_test_3();
29 public:
30     Tests() = delete;
31     Tests(Traffic *& traffic, sf::Mutex *& mutex);
32
33     void run_all_tests();
34 };
35 #endif //HIGHWAY_UNITTESTS.H

```

../highway/headers/unittests.h

A.16 util.h

```

1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #ifndef HIGHWAY_UTIL_H
6 #define HIGHWAY_UTIL_H
7
8 ////////////////////////////////////////////////////////////////////
9 // Util //
10 // Help functions for Car class. //
11 // //
12 ////////////////////////////////////////////////////////////////////
13
14 #include "car.h"
15
16 class Util{
17 public:
18     static std::vector<std::string> split_string_by_delimiter(const std::string & str, const
19     char delim);
20     static bool is_car_behind(Car * a, Car * b);
21     static bool will_car_paths_cross(Car *a, Car*b);
22     static float distance_to_car(Car * a, Car * b);
23     static float get_min_angle(float angl, float ang2);
24     static float distance(float x1, float x2, float y1, float y2);
25 };
26
27 #endif //HIGHWAY_UTIL_H

```

../highway/headers/util.h

B Source files

B.1 button.cpp

```
#include <utility>
2
3 //
4 // Created by Carl Schiller on 2019-03-05.
5 //
6
7 #include <button.h>
8
9 Button::Button(sf::Font & font_copy, unsigned int font_size, int x_pos, int y_pos,
10             const std::string & name, sf::Color button_col, sf::Color text_col, sf::Color
11             pressed) :
12     font(font_copy),
13     normal(button_col),
14     pressed(pressed)
15 {
16     text.setString(name);
17     text.setFont(font);
18     text.setCharacterSize(font_size);
19     text.setFillColor(text_col);
20     text.setOutlineThickness(0);
21
22     sf::FloatRect bounds = text.getLocalBounds();
23
24     rect.setPosition((float)x_pos, (float)y_pos);
25     rect.setSize({bounds.width+bounds.left*2, bounds.height+bounds.top*2});
26     rect.setFillColor(normal);
27     rect.setOutlineThickness(2);
28     rect.setOutlineColor(sf::Color::Black);
29 }
30
31 void Button::set_origin(int x, int y) {
32     sf::Vector2f box_origin = rect.getOrigin();
33     sf::Vector2f text_origin = text.getOrigin();
34     sf::Vector2f move_to = {(float)x, (float)y};
35
36     sf::Vector2f diff = text_origin - box_origin;
37
38     rect.setPosition(move_to);
39     text.setPosition(diff + move_to);
40 }
41
42 void Button::set_dim(int x, int y) {
43     rect.setSize({(float)x, (float)y});
44 }
45
46 void Button::center_text() {
47     sf::FloatRect rect_bounds = rect.getLocalBounds();
48     sf::FloatRect text_bounds = text.getLocalBounds();
49
50     sf::Vector2f new_origin = {rect_bounds.width/2 - (text_bounds.width + text_bounds.left * 2)
51                               / 2, (rect_bounds.height - (text_bounds.height + text_bounds.top * 2)) / 2};
52     new_origin + rect.getPosition();
53     text.setPosition(new_origin);
54 }
55
56 bool Button::clicked(sf::RenderWindow & App) {
57     if (is_mouse_in_rect(App)) {
58         rect.setFillColor(pressed);
59         if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
60             rect.setFillColor(normal);
61             return true;
62         }
63     }
64     else {
65         rect.setFillColor(normal);
66     }
67 }
```

```

62         return false;
63     }
64 }
65 else{
66     rect.setFillColor(normal);
67     return false;
68 }
69 }
70
71 sf::FloatRect Button::get_bounds() {
72     return rect.getLocalBounds();
73 }
74
75 bool Button::is_mouse_in_rect(sf::RenderWindow & App) {
76     sf::Vector2i rel_to_pos = {sf::Mouse::getPosition(App).x-(int)rect.getPosition().x, sf::
77     Mouse::getPosition(App).y-(int)rect.getPosition().y};
78     if(rel_to_pos.x < 0 || rel_to_pos.y < 0){
79         return false;
80     }
81     else if(rect.getLocalBounds().width < rel_to_pos.x || rect.getLocalBounds().height <
82     rel_to_pos.y){
83         return false;
84     }
85     else{
86         return true;
87     }
88 }
89
90 void Button::draw(sf::RenderTarget &target, sf::RenderStates states) const {
91     target.draw(rect);
92     target.draw(text);
93 }
94
95 void Button::set_text(const std::string &name) {
96     text.setString(name);
97     center_text();
98 }
99
100 Input::Input(sf::Font &font_copy, unsigned int font_size, int x_pos, int y_pos, const std::
101     string &name,
102     sf::Color button_col, sf::Color text_col, sf::Color pressed, sf::Color typ, std
103     ::string val) :
104     font(font_copy),
105     string(name),
106     input(val),
107     normal(button_col),
108     pressed(pressed),
109     typing(typ),
110     bool_typing(false)
111 {
112     text.setString(string+input);
113     text.setFont(font);
114     text.setCharacterSize(font_size);
115     text.setFillColor(text_col);
116     text.setOutlineThickness(0);
117
118     sf::FloatRect bounds = text.getLocalBounds();
119
120     rect.setPosition((float)x_pos, (float)y_pos);
121     rect.setSize({bounds.width+bounds.left*2, bounds.height+bounds.top*2});
122     rect.setFillColor(normal);
123     rect.setOutlineThickness(2);
124     rect.setOutlineColor(sf::Color::Black);
125 }
126
127 void Input::set_origin(int x, int y) {
128     sf::Vector2f box_origin = rect.getOrigin();

```

```

126     sf::Vector2f text_origin = text.getOrigin();
127     sf::Vector2f move_to = {(float)x, (float)y};
128
129     sf::Vector2f diff = text_origin - box_origin;
130
131     rect.setPosition(move_to);
132     text.setPosition(diff + move_to);
133 }
134
135 Input * Input::clicked(sf::RenderWindow & App) {
136     if (!bool_typing) {
137         if (is_mouse_in_rect(App)) {
138             rect.setFillColor(pressed);
139             if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
140                 rect.setFillColor(typing);
141                 bool_typing = true;
142                 text.setString(string);
143                 input = "";
144                 return this;
145             }
146             else {
147                 return nullptr;
148             }
149         }
150         else {
151             rect.setFillColor(normal);
152             return nullptr;
153         }
154     }
155     return this;
156 }
157
158 bool Input::is_mouse_in_rect(sf::RenderWindow & App) {
159     sf::Vector2i rel_to_pos = {sf::Mouse::getPosition(App).x - (int)rect.getPosition().x, sf::
160     Mouse::getPosition(App).y - (int)rect.getPosition().y};
161     if (rel_to_pos.x < 0 || rel_to_pos.y < 0) {
162         return false;
163     }
164     else if (rect.getLocalBounds().width < rel_to_pos.x || rect.getLocalBounds().height <
165     rel_to_pos.y) {
166         return false;
167     }
168     else {
169         return true;
170     }
171 }
172
173 Input * Input::inputting(sf::RenderWindow & App, std::string & str) {
174     if (bool_typing) {
175         if (str == "\n") {
176             bool_typing = false;
177             rect.setFillColor(normal);
178             return nullptr;
179         }
180         else if (str == "\b") {
181             input.pop_back();
182             text.setString(string + input);
183             center_text();
184             return this;
185         }
186         else {
187             input += str;
188             text.setString(string + input);
189             center_text();
190             return this;
191         }
192     }
193     return nullptr;

```

```

192 }
194 float Input::get_val() {
195     return std::stof(input);
196 }
198 void Input::draw(sf::RenderTarget &target, sf::RenderStates states) const {
199     target.draw(rect);
200     target.draw(text);
201 }
202 void Input::center_text() {
203     sf::FloatRect rect_bounds = rect.getLocalBounds();
204     sf::FloatRect text_bounds = text.getLocalBounds();
205
206     sf::Vector2f new_origin = {rect_bounds.width/2-(text_bounds.width+text_bounds.left*2)
207     /2,(rect_bounds.height-(text_bounds.height+text_bounds.top*2))/2};
208     new_origin = new_origin + rect.getPosition();
209     text.setPosition(new_origin);
210 }
212 void Input::set_dim(int x, int y) {
213     rect.setSize({(float)x,(float)y});
214 }
216 sf::FloatRect Input::get_bounds() {
217     return rect.getLocalBounds();
218 }
220 const sf::Vector2f Input::get_pos() {
221     return rect.getPosition();
222 }
224 Button_bool* Button_bool::clicked(sf::RenderWindow &App) {
225     if(is_mouse_in_rect(App)){
226         rect.setFillColor(pressed);
227         if(sf::Mouse::isButtonPressed(sf::Mouse::Left)){
228             rect.setFillColor(normal);
229             toggled = !toggled;
230             text.setString(string+(toggled ? "true" : "false"));
231             center_text();
232             return this;
233         }
234         else{
235             return nullptr;
236         }
237     }
238     else{
239         rect.setFillColor(normal);
240         return nullptr;
241     }
242 }
244 bool Button_bool::get_bool() {
245     return toggled;
246 }
248 void Button_bool::set_toggled(bool tog) {
249     toggled = tog;
250     text.setString(string+(toggled ? "true" : "false"));
251 }
252 }

```

../highway/cppfiles/button.cpp

B.2 cars.cpp

```

2  //
3  // Created by Carl Schiller on 2019-03-04.
4  //
5
6  #include "../headers/car.h"
7  #include <map>
8  #include <cmath>
9  #include <list>
10 #include <iostream>
11 #include "../headers/util.h"
12
13 //////////////////////////////////////
14 /// Constructor.
15
16 Car::Car() :
17     m_speed(0),
18     m_aggressiveness(0),
19     m_target_speed(0),
20     m_min_dist_to_car_in_front(0),
21     m_min_overtake_dist_trigger(0),
22     m_max_overtake_dist_trigger(0),
23     m_overtake_done_dist(0),
24     m_merge_min_dist(0),
25     m_search_radius_around(0),
26     m_search_radius_to_car_in_front(0),
27     current_segment(nullptr),
28     current_node(nullptr),
29     overtake_this_car(nullptr)
30 {
31 }
32
33 //////////////////////////////////////
34 /// Constructor for new car with specified lane numbering in spawn point.
35 /// Lane numbering @param lane must not exceed amount of lanes in
36 /// @param spawn_point, otherwise an exception will be thrown.
37
38 Car::Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float
    aggressivness,
39         float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
    m_max_overtake_dist_trigger,
40         float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
    float m_search_radius_to_car_in_front) :
41     m_speed(vel),
42     m_aggressiveness(aggressivness),
43     m_target_speed(target_speed),
44     m_min_dist_to_car_in_front(m_min_dist_to_car_in_front),
45     m_min_overtake_dist_trigger(m_min_overtake_dist_trigger),
46     m_max_overtake_dist_trigger(m_max_overtake_dist_trigger),
47     m_overtake_done_dist(m_overtake_done_dist),
48     m_merge_min_dist(m_merge_min_dist),
49     m_search_radius_around(m_search_radius_around),
50     m_search_radius_to_car_in_front(m_search_radius_to_car_in_front),
51     current_segment(spawn_point),
52     current_node(current_segment->get_node_pointer(lane)),
53     overtake_this_car(nullptr)
54 {
55     current_segment->append_car(this);
56
57     if (!current_node->get_nodes_from_me().empty()) {
58         heading_to_node = current_node->get_next_node(lane);
59
60         m_dist_to_next_node = Util::distance(current_node->get_x(), heading_to_node->get_x(),
        current_node->get_y(), heading_to_node->get_y());
61
62         m_theta = current_node->get_theta(heading_to_node);
63

```



```

66     }
67     else{
68         throw std::invalid_argument("Car spawns in node with empty connections, or with a
        nullptr segment");
69     }
70 }
71
72 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
73 /// Constructor for new car with specified lane. Note that
74 /// @param lane must be in @param spawn_point, otherwise no guarantee on
75 /// functionality.
76 Car::Car(RoadSegment * spawn_point, RoadNode * lane, float vel, float target_speed, float
    aggressiveness,
77         float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
    m_max_overtake_dist_trigger,
78         float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
    float m_search_radius_to_car_in_front):
79     m_speed(vel),
80     m_aggressiveness(aggressiveness),
81     m_target_speed(target_speed),
82     m_min_dist_to_car_in_front(m_min_dist_to_car_in_front),
83     m_min_overtake_dist_trigger(m_min_overtake_dist_trigger),
84     m_max_overtake_dist_trigger(m_max_overtake_dist_trigger),
85     m_overtake_done_dist(m_overtake_done_dist),
86     m_merge_min_dist(m_merge_min_dist),
87     m_search_radius_around(m_search_radius_around),
88     m_search_radius_to_car_in_front(m_search_radius_to_car_in_front),
89     current_segment(spawn_point),
90     current_node(lane),
91     overtake_this_car(nullptr)
92 {
93     current_segment->append_car(this);
94
95     if(!current_node->get_nodes_from_me().empty() || current_segment->next_segment() !=
    nullptr){
96         heading_to_node = current_node->get_next_node(0);
97
98         m_dist_to_next_node = Util::distance(current_node->get_x(), heading_to_node->get_x(),
    current_node->get_y(), heading_to_node->get_y());
99
100         m_theta = current_node->get_theta(heading_to_node);
101     }
102     else{
103         throw std::invalid_argument("Car spawns in node with empty connections, or with a
        nullptr segment");
104     }
105 }
106
107 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
108 /// Destructor for car.
109
110 Car::~Car(){
111     if(this->current_segment != nullptr){
112         this->current_segment->remove_car(this); // remove this pointer shit
113     }
114
115     overtake_this_car = nullptr;
116     current_segment = nullptr;
117     heading_to_node = nullptr;
118     current_node = nullptr;
119 }
120
121 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
122 /// Updates position for car with time step @param delta_t.
123
124 void Car::update_pos(float delta_t) {
125     m_dist_to_next_node -= m_speed*delta_t;

```

```

128 // if we are at a new node.
129
130 if(m_dist_to_next_node < 0){
131     current_segment->remove_car(this); // remove car from this segment
132     current_segment = heading_to_node->get_parent_segment(); // set new segment
133     if(current_segment != nullptr){
134         current_segment->append_car(this); // add car to new segment
135         if(current_segment->meter){
136             current_segment->car_passed = true;
137         }
138     }
139
140     current_node = heading_to_node; // set new current node as previous one.
141
142     //TODO: place logic for choosing next node
143     std::vector<RoadNode*> connections = current_node->get_nodes_from_me();
144
145     if(!connections.empty()){
146         merge(connections);
147
148         m_dist_to_next_node += Util::distance(current_node->get_x(), heading_to_node->
149         get_x(), current_node->get_y(), heading_to_node->get_y());
150         m_theta = current_node->get_theta(heading_to_node);
151     }
152 }
153
154 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
155 /// Function to determine if we can merge into another lane depending on.
156 /// properties of @param connections.
157
158 void Car::merge(std::vector<RoadNode*> & connections) {
159     // check if we merge
160     int current_lane = current_segment->get_lane_number(current_node);
161     bool can_merge = true;
162     std::map<Car*, bool> cars_around_car = find_cars_around_car();
163     Car * closest_car = find_closest_car_ahead();
164
165     for(auto it : cars_around_car){
166         float delta_dist = Util::distance_to_car(it.first, this);
167         float delta_speed = abs(speed()-it.first->speed());
168
169         if(current_lane == 0 && it.first->heading_to_node->get_parent_segment()->
170         get_lane_number(it.first->heading_to_node) == 1 ){
171             can_merge =
172             delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, m_merge_min_dist
173             );
174         }
175         else if(current_lane == 1 && it.first->heading_to_node->get_parent_segment()->
176         get_lane_number(it.first->heading_to_node) == 0){
177             can_merge =
178             delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, m_merge_min_dist
179             );
180         }
181     }
182
183     if(!can_merge){
184         break;
185     }
186
187     if(current_segment->merge){
188         if(current_lane == 0 && connections[0]->get_parent_segment()->
189         get_total_amount_of_lanes() != 2){
190             if(can_merge){
191                 heading_to_node = connections[1];
192             }
193         }
194     }
195 }

```

```

190         else{
191             heading_to_node = connections[0];
192         }
193     }
194     else if (connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
195         current_lane = std::max(current_lane-1,0);
196         heading_to_node = connections[current_lane];
197     }
198     else{
199         heading_to_node = connections[current_lane];
200     }
201 }
202 // if we are in start section
203 else if (current_segment->get_total_amount_of_lanes() == 3){
204     if (connections.size() == 1){
205         heading_to_node = connections[0];
206     }
207     else{
208         heading_to_node = connections[current_lane];
209     }
210 }
211 // if we are in middle section
212 else if (current_segment->get_total_amount_of_lanes() == 2){
213     // normal way
214     if (connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
215         // check if we want to overtake car in front
216         do_we_want_to_overtake(closest_car, current_lane);
217
218         // committed to overtaking
219         if (overtake_this_car != nullptr){
220             if (current_lane != 1){
221                 if (can_merge){
222                     heading_to_node = connections[1];
223                 }
224                 else{
225                     heading_to_node = connections[current_lane];
226                 }
227             }
228             else{
229                 heading_to_node = connections[current_lane];
230             }
231         }
232         // merge back if overtake this car is nullptr.
233         else{
234             if (can_merge){
235                 heading_to_node = connections[0];
236             }
237             else{
238                 heading_to_node = connections[current_lane];
239             }
240         }
241     }
242     else{
243         heading_to_node = connections[0];
244     }
245 }
246 }
247 else if (current_segment->get_total_amount_of_lanes() == 1){
248     heading_to_node = connections[0];
249 }
250 }
251
252 //////////////////////////////////////
253 /// Helper function to determine if this car wants to overtake
254 /// @param closest_car.
255
256 void Car::do_we_want_to_overtake(Car * & closest_car, int & current_lane) {

```

```

//see if we want to overtake car.
258
if(closest_car != nullptr){
260     //float delta_speed = closest_car->speed()-speed();
    float delta_distance = Util::distance_to_car(this, closest_car);
262
    if(overtake_this_car == nullptr){
264         if(delta_distance > m_min_overtake_dist_trigger && delta_distance <
m_max_overtake_dist_trigger && (target_speed()/closest_car->target_speed() >
m_aggressiveness*1.0f) && current_lane == 0 && closest_car->current_node->
get_parent_segment()->get_lane_number(closest_car->current_node) == 0){
            overtake_this_car = closest_car;
266        }
    }
268
}

270
if(overtake_this_car != nullptr){
272     if(Util::is_car_behind(overtake_this_car, this) && (Util::distance_to_car(this,
overtake_this_car) > m_overtake_done_dist)){
        overtake_this_car = nullptr;
274     }
}
276
}

278 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/// Function to accelerate this car.
280
void Car::accelerate(float elapsed){
282     float target = m_target_speed;
    float d_vel; // proportional control.
284
    if(m_speed < target*0.75){
286         d_vel = m_aggressiveness*elapsed*2.0f;
    }
    else{
288         d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
290     }

    m_speed += d_vel;
292
}

294
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
296 /// Helper function to avoid collision with another car.

298 void Car::avoid_collision(float delta_t) {
    float min_distance = m_min_dist_to_car_in_front; // for car distance.
300     float ideal = min_distance+min_distance*(m_speed/20.f);

    Car * closest_car = find_closest_car_ahead();
    float detection_distance = m_speed*5.0f;
304

    if(closest_car != nullptr) {
306         float radius_to_car = Util::distance_to_car(this, closest_car);
        float delta_speed = closest_car->speed() - this->speed();
308

        if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
310             m_speed -= std::max(std::max((radius_to_car-min_distance)*0.5f,0.0f),10.0f*
delta_t);
        }
        else if(radius_to_car < min_distance){
312             m_speed -= std::max(std::max((min_distance-radius_to_car)*0.5f,0.0f),2.0f*
delta_t);
314         }
        else if(delta_speed < 0 && radius_to_car < detection_distance){
316             m_speed -= std::min(
                abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
m_aggressiveness * 0.15f,

```

```

318         10.0f * delta_t);
320     }
322     else {
324         accelerate(delta_t);
326     }
328     if(current_segment->merge){
330         std::map<Car*,bool> around = find_cars_around_car();
332         for(auto it : around){
334             float delta_dist = Util::distance_to_car(it.first, this);
336             delta_speed = abs(speed()-it.first->speed());
338             if(it.first->current_node->get_parent_segment()->get_lane_number(it.first->
current_node) == 0 && delta_dist < ideal && this->current_segment->get_lane_number(
current_node) == 1 && speed()/target_speed() > 0.5){
340                 if(Util::is_car_behind(it.first, this)){
342                     accelerate(delta_t);
344                 }
346                 else{
348                     m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*
delta_t);
350                 }
352             }
354             else if(it.first->current_node->get_parent_segment()->get_lane_number(it.
first->current_node) == 1 && this->current_segment->get_lane_number(current_node) == 0
&& speed()/target_speed() > 0.5 && delta_dist < ideal){
356                 if(Util::is_car_behind(this, it.first)){
358                     m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*
delta_t);
360                 }
362                 else{
364                     accelerate(delta_t);
366                 }
368             }
370         }
372     }
374     else{
376         }
378     }
380     if(heading_to_node->get_parent_segment()->meter){
382         if(heading_to_node->get_parent_segment()->car_passed || heading_to_node->
get_parent_segment()->ramp_counter < heading_to_node->get_parent_segment()->period*0.5f)
384         {
386             if (m_dist_to_next_node < ideal) {
388                 m_speed -= std::max(std::max((m_dist_to_next_node-min_distance)*0.5f,0.0f)
,10.0f*delta_t);
390             }
392             else if(m_dist_to_next_node < detection_distance){
394                 m_speed = std::min(
396                     abs(pow(m_speed, 2.0f)) * pow(ideal * 0.25f / m_dist_to_next_node ,
2.0f) * m_aggressiveness * 0.15f,
398                     10.0f * delta_t);
400             }
402             }
404             else{
406                 accelerate(delta_t);
408             }
410         }
412     }
414     if(m_speed < 0){
416         m_speed = 0;
418     }

```

```

376 }
378
380 // Helper function to find closest car in the same lane ahead of this car.
381 // Returns a car if found, otherwise nullptr.
382
383 Car* Car::find_closest_car_ahead() {
384     float search_radius = m_search_radius_to_car_in_front;
385     std::map<RoadNode*,bool> visited;
386     std::list<RoadNode*> queue;
387
388     for(RoadNode * node : (this->current_segment->get_nodes())){
389         queue.push_front(node);
390     }
391
392     Car* answer = nullptr;
393
394     float shortest_distance = 10000000;
395
396     while(!queue.empty()){
397         RoadNode * next_node = queue.back(); // get last element
398         queue.pop_back(); // remove element
399
400         if(next_node != nullptr){
401             if(!visited[next_node] && Util::distance(x_pos(),next_node->get_x(),y_pos(),
402 next_node->get_y()) < search_radius){
403                 visited[next_node] = true;
404
405                 for(Car * car : next_node->get_parent_segment()->m_cars){
406                     if(this != car){
407                         float radius = Util::distance_to_car(this,car);
408                         if(Util::is_car_behind(this,car) && Util::will_car_paths_cross(this,
409 car) && radius < shortest_distance){
410                             shortest_distance = radius;
411                             answer = car;
412                         }
413                     }
414                 }
415
416                 // push in new nodes in front of list.
417                 for(RoadNode * node : next_node->get_nodes_from_me()){
418                     queue.push_front(node);
419                 }
420             }
421         }
422     }
423     return answer;
424 }
425
426 // Searches for cars around this car in a specified radius. Note that
427 // search radius is the radius to RoadNodes, and not surrounding cars.
428 // Returns a map of cars the function has found.
429
430 std::map<Car *,bool> Car::find_cars_around_car() {
431     float search_radius = m_search_radius_around;
432     std::map<RoadNode*,bool> visited;
433     std::list<RoadNode*> queue;
434
435     for(RoadNode * node : (this->current_segment->get_nodes())){
436         queue.push_front(node);
437     }
438
439     std::map<Car *,bool> answer;
440     while(!queue.empty()){
441         RoadNode * next_node = queue.back(); // get last element

```



```

510 ///////////////////////////////////////////////////////////////////
512 /// Returns theta of car, the direction of the car. Defined in radians as a
514 /// mathematician would define angles.
516 float & Car::theta() {
518     return m_theta;
519 }
520 ///////////////////////////////////////////////////////////////////
522 /// Returns current segment car is in.
524 RoadSegment* Car::get_segment() {
526     return current_segment;
527 }

```

../highway/cppfiles/car.cpp

B.3 main.cpp

```

1  #include <iostream>
2  #include <vector>
3  #include "SFML/Graphics.hpp"
4  #include "../headers/simulation.h"
5  #include "../headers/unittests.h"
6  #include "../headers/screens.h"
7
8  int main() {
9      std::vector<cScreen*> Screens;
10     int screen = 0;
11
12     sf::RenderWindow App(sf::VideoMode(550*2, 600*2), "Highway");
13     App.setFramerateLimit(60);
14
15     screen_0 s0;
16     Screens.push_back(&s0);
17     screen_1 s1;
18     Screens.push_back(&s1);
19     screen_2 s2;
20     Screens.push_back(&s2);
21     screen_3 s3;
22     Screens.push_back(&s3);
23
24     std::vector<float> args;
25
26     float m_aggro = 1.0f;
27     args.push_back(m_aggro);
28     float m_aggro_sigma = 0.2f;
29     args.push_back(m_aggro_sigma);
30     float m_spawn_freq = 2.0f;
31     args.push_back(m_spawn_freq);
32     float m_speed = 20.f;
33     args.push_back(m_speed);
34
35     float m_lane_0_spawn_prob = 5.f;
36     args.push_back(m_lane_0_spawn_prob);
37     float m_lane_1_spawn_prob = 1.f;
38     args.push_back(m_lane_1_spawn_prob);
39     float m_lane_2_spawn_prob = 1.f;
40     args.push_back(m_lane_2_spawn_prob);
41     float m_ramp_0_spawn_prob = 5.f;
42     args.push_back(m_ramp_0_spawn_prob);
43
44     float m_min_dist_to_car_in_front = 8;
45     args.push_back(m_min_dist_to_car_in_front);
46     float m_min_overtake_dist_trigger = 10;

```



```

47     args.push_back(m_min_overtake_dist_trigger);
    float m_max_overtake_dist_trigger = 40;
49     args.push_back(m_max_overtake_dist_trigger);
    float m_overtake_done_dist = 30;
51     args.push_back(m_overtake_done_dist);
    float m_merge_min_dist = 15.0f;
53     args.push_back(m_merge_min_dist);
    float m_search_radius_around = 30;
55     args.push_back(m_search_radius_around);
    float m_search_radius_to_car_in_front = 50;
57     args.push_back(m_search_radius_to_car_in_front);
    float sim_speed = 10;
59     args.push_back(sim_speed);
    float framerate = 60;
61     args.push_back(framerate);
    float ramp_meter_period = 10;
63     args.push_back(ramp_meter_period);

    std::vector<bool> bool_args;
    bool debug = false;
67     bool_args.push_back(debug);
    bool ramp_meter = false;
69     bool_args.push_back(ramp_meter);

    while(screen >= 0){
        screen = Screens[screen]->Run(App,&args,&bool_args);
73     }

75     return 0;
}

```

../highway/cppfiles/main.cpp

B.4 road.cpp

```

//
// Created by Carl Schiller on 2019-03-04.
//
4
#include "../headers/road.h"
6 #include <fstream>
#include <vector>
8 #include "../headers/roadsegment.h"
#include <iostream>
10 #include "../headers/util.h"

12 //////////////////////////////////////
/// Constructor of Road.
14
Road::Road() :
16     MFILENAME("../road.txt")
{
18     if(!load_road()){
        std::cout << "Error in loading road.\n";
20     };
}

22
24 //////////////////////////////////////
/// Destructor of Road.

26 Road::~~Road() {
    for(RoadSegment * seg : m_segments){
28         delete seg;
    }
    m_segments.clear();
30 }

```

```

32 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
34 /// Function to load Road from txt file. Parsing as follows:
35 ///
36 /// # ignores current line input.
37 ///
38 /// If there are 4 tokens in current line:
39 /// tokens[0]: segment number
40 /// tokens[1]: segment x position
41 /// tokens[2]: segment y position
42 /// tokens[3]: amount of lanes
43 ///
44 /// If there are 5 tokens in current line:
45 /// tokens[0]: segment number
46 /// tokens[1]: segment x position
47 /// tokens[2]: segment y position
48 /// tokens[3]: amount of lanes
49 /// tokens[4]: spawn point or if it's a merging lane (true/false/merge)
50 ///
51 /// If there are 4+3*n tokens in current line:
52 /// tokens[0]: segment number
53 /// tokens[1]: segment x position
54 /// tokens[2]: segment y position
55 /// tokens[3]: amount of lanes
56 /// tokens[3+3*n]: from lane number of current segment
57 /// tokens[4+3*n]: to lane number of segment specified in next token (below)
58 /// tokens[5+3*n]: to segment number.

60 bool Road::load_road() {
61     bool loading = true;
62     std::ifstream stream;
63     stream.open(MFILENAME);
64
65     std::vector<std::vector<std::string>>> road_vector;
66     road_vector.reserve(100);
67
68     if(stream.is_open()){
69         std::string line;
70         std::vector<std::string> tokens;
71         while(std::getline(stream, line)){
72             tokens = Util::split_string_by_delimiter(line, ' ');
73             if(tokens[0] != "#"){
74                 road_vector.push_back(tokens);
75             }
76         }
77     }
78     else{
79         loading = false;
80     }
81
82     // load segments into memory.
83     for(std::vector<std::string> & vec : road_vector){
84         if(vec.size() == 5){
85             if(vec[4] == "merge"){
86                 RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std
::stoi(vec[3]), true);
87                 m_segments.push_back(seg);
88             }
89             else if(vec[4] == "ramp"){
90                 RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std
::stoi(vec[3]), false);
91                 m_segments.push_back(seg);
92                 ramp_meter_position = seg;
93             }
94             else{
95                 RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std
::stoi(vec[3]), false);
96

```

```

98         m_segments.push_back(seg);
99     }
100     }
101     else{
102         RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std::
stoi(vec[3]),false);
103         m_segments.push_back(seg);
104     }
105 }
106
107 // populate nodes.
108 for (int i = 0; i < m_segments.size(); ++i) {
109     // populate nodes normally.
110     if(road_vector[i].size() == 4){
111         m_segments[i]->set_next_road_segment(m_segments[i+1]);
112         m_segments[i]->calculate_theta();
113         // calculate nodes based on theta.
114         m_segments[i]->calculate_and_populate_nodes();
115     }
116     else if(road_vector[i].size() == 5){
117         if(road_vector[i][4] == "false"){
118             // take previous direction and populate nodes.
119             m_segments[i]->set_theta(m_segments[i-1]->get_theta());
120             m_segments[i]->calculate_and_populate_nodes();
121             // but do not connect nodes to new ones.
122
123             // make this a despawn segment
124             m_despawn_positions.push_back(m_segments[i]);
125         }
126         else if(road_vector[i][4] == "true"){
127             m_segments[i]->set_next_road_segment(m_segments[i+1]);
128             m_segments[i]->calculate_theta();
129             // calculate nodes based on theta.
130             m_segments[i]->calculate_and_populate_nodes();
131
132             // make this a spawn segment
133             m_spawn_positions.push_back(m_segments[i]);
134         }
135         else if(road_vector[i][4] == "merge" || road_vector[i][4] == "ramp"){
136             m_segments[i]->set_next_road_segment(m_segments[i+1]);
137             m_segments[i]->calculate_theta();
138             // calculate nodes based on theta.
139             m_segments[i]->calculate_and_populate_nodes();
140         }
141     }
142     // else we connect one by one.
143     else{
144         // take previous direction and populate nodes.
145         m_segments[i]->set_theta(m_segments[i-1]->get_theta());
146         // calculate nodes based on theta.
147         m_segments[i]->calculate_and_populate_nodes();
148     }
149 }
150
151 // connect nodes.
152 for (int i = 0; i < m_segments.size(); ++i) {
153     // do normal connection, ie connect all nodes.
154     if(road_vector[i].size() == 4){
155         m_segments[i]->set_all_node_pointers_to_next_segment();
156     }
157     else if(road_vector[i].size() == 5){
158         if(road_vector[i][4] == "false"){
159             // but do not connect nodes to new ones.
160         }
161     }

```

```

164         else if(road_vector[i][4] == "true" || road_vector[i][4] == "merge" ||
road_vector[i][4] == "ramp"){
166             m_segments[i]->set_all_node_pointers_to_next_segment();
168         }
170         // else we connect one by one.
172         else{
174             // manually connect nodes.
176             int amount_of_pointers = (int)road_vector[i].size()-4;
178             for(int j = 0; j < amount_of_pointers/3; j++){
180                 int current_pos = 4+j*3;
182                 RoadSegment * next_segment = m_segments[std::stoi(road_vector[i][current_pos
+2])];
184                 m_segments[i]->set_node_pointer_to_node(std::stoi(road_vector[i][current_pos
]),std::stoi(road_vector[i][current_pos+1]),next_segment);
186             }
188         }
190     }
192     return loading;
194 }

196 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
198 /// Returns spawn positions of Road
200 std::vector<RoadSegment*>& Road::spawn_positions() {
202     return m_spawn_positions;
204 }

206 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
208 /// Returns despawn positions of Road
210 std::vector<RoadSegment*>& Road::despawn_positions() {
212     return m_despawn_positions;
214 }

216 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
218 /// Returns all segments of Road.
220 std::vector<RoadSegment*>& Road::segments() {
222     return m_segments;
224 }

```

../highway/cppfiles/road.cpp

B.5 roadnode.cpp

```

1  //
2  // Created by Carl Schiller on 2019-03-04.
3  //
4
5  #include "../headers/roadnode.h"
6  #include <cmath>
7
8  ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
9  /// Constructor
10
11  RoadNode::RoadNode() = default;
12
13  ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
14  /// Destructor
15
16  RoadNode::~~RoadNode() = default;
17
18  ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
19  /// Constructor, @param x is x position of node, @param y is y position of node,

```

```

21  /// @param segment is to which segment this RoadNode belongs.
22
23  RoadNode::RoadNode(float x, float y, RoadSegment * segment) {
24      m_x = x;
25      m_y = y;
26      m_is_child_of = segment;
27  }
28
29  //////////////////////////////////////
30  /// Appends a new RoadNode to the list connections from this RoadNode.
31  /// I.e. to where a Car is allowed to drive.
32
33  void RoadNode::set_next_node(RoadNode * next_node) {
34      m_nodes_from_me.push_back(next_node);
35      next_node->m_nodes_to_me.push_back(this); // sets double linked chain.
36  }
37
38  //////////////////////////////////////
39  /// Appends a new RoadNode to the list connections to this RoadNode.
40  /// I.e. from where a Car is allowed to drive to this Node.
41
42  void RoadNode::set_previous_node(RoadNode * prev_node) {
43      m_nodes_to_me.push_back(prev_node);
44  }
45
46  //////////////////////////////////////
47  /// Returns RoadSegment to which this RoadNode belongs.
48
49  RoadSegment* RoadNode::get_parent_segment() {
50      return m_is_child_of;
51  }
52
53  //////////////////////////////////////
54  /// Returns connections from this RoadNode.
55
56  std::vector<RoadNode*> & RoadNode::get_nodes_from_me() {
57      return m_nodes_from_me;
58  }
59
60  //////////////////////////////////////
61  /// Returns connections to this RoadNode.
62
63  std::vector<RoadNode*>& RoadNode::get_nodes_to_me() {
64      return m_nodes_to_me;
65  }
66
67  //////////////////////////////////////
68  /// Returns x position of RoadNode.
69
70  float RoadNode::get_x() {
71      return m_x;
72  }
73
74  //////////////////////////////////////
75  /// Returns y position of RoadNode.
76
77  float RoadNode::get_y() {
78      return m_y;
79  }
80
81  //////////////////////////////////////
82  /// Returns angle of this RoadNode to @param node as a mathematician
83  /// would define angles. In radians.
84
85  float RoadNode::get_theta(RoadNode* node) {
86      for(RoadNode * road_node : m_nodes_from_me){
87          if(node == road_node){
88              return atan2(m_y-node->m_y, node->m_x-m_x);

```

```

    }
89     }
    throw std::invalid_argument("Node given is not a connecting node");
91 }

93 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
94 /// Returns RoadNode according to @param lane from the vector of node
95 /// connections from this RoadNode.

97 RoadNode* RoadNode::get_next_node(int lane) {
    return m_nodes_from_me[lane];
99 }

```

../highway/cppfiles/roadnode.cpp

B.6 roadsegment.cpp

```

//
2 // Created by Carl Schiller on 2019-03-04.
//
4
6 #include "../headers/roadsegment.h"
7 #include "../headers/roadnode.h"
8 #include <cmath>
9
10 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
11 /// RoadSegment destructor, removes all RodeNode element children because of
12 /// ownership.
13
14 RoadSegment::~RoadSegment(){
15     for(RoadNode * elem : m_nodes){
16         delete elem;
17     }
18     m_nodes.clear();
19 }
20
21 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
22 /// Constructor, creates a new segment with next connecting segment as
23 /// @param next_segment
24
25 RoadSegment::RoadSegment(float x, float y, RoadSegment * next_segment, int lanes):
26     m_x(x),
27     m_y(y),
28     m_n_lanes(lanes),
29     m_next_segment(next_segment)
30 {
31     m_theta = atan2(m_y-m_next_segment->m_y, m_next_segment->m_x-m_x);
32
33     m_nodes.reserve(m_n_lanes);
34
35     ramp_counter = 0;
36     car_passed = false;
37     meter = false;
38     period = 0;
39
40     calculate_and_populate_nodes(); // populates segment with RoadNodes.
41 }
42
43 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
44 /// Constructor, creates a new segment with manually entered @param theta.
45
46 RoadSegment::RoadSegment(float x, float y, float theta, int lanes) :
47     m_x(x),
48     m_y(y),
49     m_theta(theta),
50     m_n_lanes(lanes),

```

```

50         m_next_segment(nullptr)
51     {
52         m_nodes.reserve(m_n_lanes);
53
54         ramp_counter = 0;
55         car_passed = false;
56         meter = false;
57         period = 0;
58
59         calculate_and_populate_nodes(); // populates segment with RoadNodes.
60     }
61
62     //////////////////////////////////////
63     /// Constructor, creates a new segment without creating RoadNodes. This
64     /// needs to be done manually with functions below.
65
66     RoadSegment::RoadSegment(float x, float y, int lanes, bool mer):
67         m_x(x),
68         m_y(y),
69         m_n_lanes(lanes),
70         m_next_segment(nullptr),
71         merge(mer)
72     {
73         m_nodes.reserve(m_n_lanes);
74
75         ramp_counter = 0;
76         car_passed = false;
77         meter = false;
78         period = 0;
79
80         // can't set nodes if we don't have a theta.
81     }
82
83     //////////////////////////////////////
84     /// Returns theta (angle) of RoadSegment, in which direction the segment points
85
86     float RoadSegment::get_theta() {
87         return m_theta;
88     }
89
90     //////////////////////////////////////
91     /// Returns x position of RoadSegment.
92
93     const float RoadSegment::get_x() const {
94         return m_x;
95     }
96
97     //////////////////////////////////////
98     /// Returns y position of RoadSegment.
99
100    const float RoadSegment::get_y() const {
101        return m_y;
102    }
103
104    //////////////////////////////////////
105    /// Returns int number of @param node. E.g. 0 would be the right-most lane.
106    /// Throws exception if we do not find the node in this segment.
107
108    int RoadSegment::get_lane_number(RoadNode * node) {
109        for(int i = 0; i < m_n_lanes; i++){
110            if(node == m_nodes[i]){
111                return i;
112            }
113        }
114        throw std::invalid_argument("Node is not in this segment");
115    }
116
117    //////////////////////////////////////

```

```

118 /// Adds a new car to the segment.
120 void RoadSegment::append_car(Car * car) {
122     m_cars.push_back(car);
124 }
126 /// Removes car from segment, if car is not in list we do nothing.
128 void RoadSegment::remove_car(Car * car) {
130     unsigned long size = m_cars.size();
132     bool found = false;
134     for(int i = 0; i < size; i++){
136         if(car == m_cars[i]){
138             m_cars[i] = nullptr;
140             found = true;
142         }
144     }
146     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(),
148         static_cast<Car*>(nullptr));
150     m_cars.erase(new_end, m_cars.end());
152     /*
154     if(!found){
156         throw std::invalid_argument("Car is not in this segment.");
158     }
160     */
162 }
164 /// Sets theta of RoadSegment according to @param theta.
166 void RoadSegment::set_theta(float theta) {
168     m_theta = theta;
170 }
172 /// Automatically populates segment with nodes according to amount of lanes
174 /// specified and theta specified.
176 void RoadSegment::calculate_and_populate_nodes() {
178     // calculates placement of nodes.
180     float total_length = MLANE.WIDTH*(m_n_lanes-1);
182     float current_length = -total_length/2.0f;
184     for(int i = 0; i < m_n_lanes; i++){
186         float x_pos = m_x+current_length*cos(m_theta+(float)M_PI*0.5f);
188         float y_pos = m_y-current_length*sin(m_theta+(float)M_PI*0.5f);
190         m_nodes.push_back(new RoadNode(x_pos, y_pos, this));
192         current_length += MLANE.WIDTH;
194     }
196 }
198 /// Sets next segment to @param next_segment
200 void RoadSegment::set_next_road_segment(RoadSegment * next_segment) {
202     m_next_segment = next_segment;
204 }
206 /// Calculates theta according to next_segment. Throws if m_next_segment is
208 /// nullptr
210 void RoadSegment::calculate_theta() {
212     if(m_next_segment == nullptr){
214         throw std::invalid_argument("Can't calculate theta if next segment is nullptr");
216     }
218 }

```



```

    m_theta = atan2(m_y-m_next_segment->m_y, m_next_segment->m_x-m_x);
186 }

188 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
189 /// Returns node of lane number n. E.g. n=0 is the right-most lane.
190
191 RoadNode* RoadSegment::get_node_pointer(int n) {
192     return m_nodes[n];
193 }
194
195 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
196 /// Returns all nodes in segment.
197
198 std::vector<RoadNode*> RoadSegment::get_nodes() {
199     return m_nodes;
200 }
201
202 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
203 /// Returns next segment
204
205 RoadSegment* RoadSegment::next_segment() {
206     return m_next_segment;
207 }
208
209 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
210 /// Automatically populates node connections by connecting current node to
211 /// all nodes in next segment.
212
213 void RoadSegment::set_all_node_pointers_to_next_segment() {
214     for(RoadNode * node: m_nodes){
215         for(int i = 0; i < m_next_segment->m_n_lanes; i++){
216             node->set_next_node(m_next_segment->get_node_pointer(i));
217         }
218     }
219 }
220
221 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
222 /// Manually set connection to next segment's node. No guarantee is made
223 /// on @param from_node_n and @param to_node_n. Can crash if index out of range.
224
225 void RoadSegment::set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *
226     next_segment) {
227     RoadNode * pointy = next_segment->get_node_pointer(to_node_n);
228     m_nodes[from_node_n]->set_next_node(pointy);
229 }
230
231 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
232 /// Returns amount of lanes in this segment.
233
234 const int RoadSegment::get_total_amount_of_lanes() const {
235     return m_n_lanes;
236 }

```

../highway/cppfiles/roadsegment.cpp

B.7 screen0.cpp

```

1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #include "../headers/screen0.h"
6 #include <iostream>
7 #include "button.h"
8 #include <unistd.h>
9

```

```

screen_0::screen_0() = default;
11
int screen_0::Run(sf::RenderWindow &App, std::vector<float> * args, std::vector<bool> * bargs
) {
13     sf::Color normal = sf::Color(253,246,227);
    sf::Color hover = sf::Color(253,235,227);
15
    sf::Sprite sprite;
    sf::Texture texture;
    sf::Font font;
17
19     if(!texture.loadFromFile("../iu.png")){
21         return -1;
    }
23     if(!font.loadFromFile("/Library/Fonts/Andale mono.ttf")){
        return -1;
25     }

27     sf::Event event;

29     sprite.setTexture(texture);
    sprite.setColor(sf::Color::White);
31     sprite.setScale(App.getSize().x/sprite.getLocalBounds().width,App.getSize().y/sprite.
        getLocalBounds().height);

33     Button button1 = Button(font,28*2,500,500,"Visualize simulation",normal,sf::Color::Black
        ,hover);
    button1.set_origin(0,0);
35     button1.set_dim(App.getSize().x,100);
    button1.center_text();
37

    Button button2 = Button(font,28*2,500,500,"Settings",normal,sf::Color::Black,hover);
39     button2.set_origin(0,100);
    button2.set_dim(App.getSize().x,100);
41     button2.center_text();

43     Button button3 = Button(font,28*2,500,500,"Run simulation",normal,sf::Color::Black,hover
        );
    button3.set_origin(0,200);
45     button3.set_dim(App.getSize().x,100);
    button3.center_text();
47

    std::vector<Button *> buttons;
49

    bool just_arrived = true;
51

    buttons.push_back(&button1);
53     buttons.push_back(&button2);
    buttons.push_back(&button3);
55

    int micro = 1000000;
57     usleep((useconds_t)micro/8);

59     while(true){
        while(App.pollEvent(event)){
61             if(event.type == sf::Event::Closed){
                return -1;
63             }

65             if(event.type == sf::Event::MouseButtonPressed && just_arrived){

67             }
            else if(!just_arrived){
69                 if(button1.clicked(App)){
                    return 1;
71                 }
                else if(button2.clicked(App)){
73                     return 2;

```

```

75         }
76         else if(button3.clicked(App)){
77             return 3;
78         }
79     }
80     else{
81         just_arrived = false;
82     }
83 }
84
85 App.clear();
86
87 App.draw(sprite);
88
89 for (Button * but : buttons) {
90     App.draw(*but);
91 }
92
93 App.display();
94 }

```

../highway/cppfiles/screen0.cpp

B.8 screen1.cpp

```

//
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #include "../headers/screen1.h"
6 #include "../headers/traffic.h"
7 #include "../headers/simulation.h"
8 #include "../headers/unittests.h"
9 #include "button.h"
10 #include <iostream>
11 #include <unistd.h>
12
13 screen_1::screen_1() = default;
14
15 int screen_1::Run(sf::RenderWindow &App, std::vector<float> * args, std::vector<bool> * bargs) {
16     sf::Mutex mutex;
17
18     sf::Font font;
19
20     sf::Texture texture;
21     if(!texture.loadFromFile("../mall2.png"))
22     {
23
24     }
25
26     if(!font.loadFromFile("/Library/Fonts/Andale mono.ttf")){
27         return -1;
28     }
29
30     sf::Sprite background;
31     background.setTexture(texture);
32     //background.setColor(sf::Color::Black);
33     background.scale(2.0f, 2.0f);
34
35     sf::Clock clock;
36     sf::Clock t0;
37
38     bool just_arrived = true;

```

```

40     sf::Event event;

42     bool exit_bool = false;

44     sf::Time time1;

46     sf::Mutex * mutex1 = &mutex;
    bool * exit = &exit_bool;
48     //thread.launch();
    auto * traffic = new Traffic(*bargs,*args);
50     Simulation sim = Simulation(traffic , mutex1 , args[0][15] , args[0][16] , exit);
    sf::Text debug_info;

52

54     sf::Thread thread(&Simulation::update,&sim);
    thread.launch();

56     Button button = Button(font,24,0,215,"Go back" , sf::Color(253,246,227) , sf::Color::Black ,
    sf::Color(253,235,227));
    button.center_text();

58

60     int micro = 1000000;
    usleep((useconds_t)micro/8);

62     while(true){
        // check all the window's events that were triggered since the last iteration of the
        loop

64         while (App.pollEvent(event))
66         {
            // "close requested" event: we close the window
68             if (event.type == sf::Event::Closed){
                exit_bool = true;
                thread.wait();
                delete traffic;
70                 return -1;
72             }

74             if(event.type == sf::Event::MouseButtonPressed && just_arrived){
76                 }
                else if(!just_arrived){
78                     if(button.clicked(App)){
80                         exit_bool = true;
                        thread.wait();
                        delete traffic;
82                         return 0;
84                     }
                }
86                 else{
                    just_arrived = false;
88                 }
            }

90             sf::Time elapsed = clock.restart();

92             mutex.lock();
            //std::cout << "copying\n";
            Traffic * copy = new Traffic(*traffic);
94             //std::cout << "copied\n";
            mutex.unlock();

96

98             App.clear(sf::Color(255,255,255,255));

100             App.draw(background);
            //mutex.lock();
102             App.draw(*copy);

104             copy->get_info(debug_info , elapsed);

```

```

106         //mutex.unlock();
108         App.draw(debug_info);
109         App.draw(button);
110
111         App.display();
112     }
113     /*
114     else{
115         //sf::Thread thread(&Tests::run_all_tests,&tests);
116         sf::Mutex * mutex1 = &mutex;
117         //thread.launch();
118         auto * traffic = new Traffic(debug,*args);
119         Tests tests = Tests(traffic, mutex1);
120         Traffic copy;
121         sf::Text debug_info;
122
123         sf::Thread thread(&Tests::run_all_tests,&tests);
124         thread.launch();
125
126         // run the program as long as the window is open
127         while (true)
128         {
129
130             // check all the window's events that were triggered since the last iteration of
the loop
131             while (App.pollEvent(event))
132             {
133                 // "close requested" event: we close the window
134                 if (event.type == sf::Event::Closed){
135                     thread.terminate();
136                     delete traffic;
137                     return 0;
138                 }
139             }
140             //Traffic copy = tests.m_traffic; // deep copy it
141             sf::Time elapsed = clock.restart();
142
143             App.clear(sf::Color(255,255,255,255));
144
145             mutex.lock();
146             copy = *traffic;
147             mutex.unlock();
148
149             App.draw(background);
150             App.draw(copy);
151
152             copy.get_info(debug_info,elapsed);
153             App.draw(debug_info);
154
155             App.display();
156         }
157     }
158     */
159     return -1;
160 }

```

../highway/cppfiles/screen1.cpp

B.9 screen2.cpp

```

2 //
3 // Created by Carl Schiller on 2019-03-05.
4 //
#include "screen2.h"

```

```

6 #include <iostream>
  #include "button.h"
8 #include <unistd.h>

10 screen_2::screen_2() = default;

12 int screen_2::Run(sf::RenderWindow &App, std::vector<float> * args, std::vector<bool> * bargs)
  {
14     sf::Color normal = sf::Color(253,246,227);
        sf::Color hover = sf::Color(253,235,227);

16     sf::Sprite sprite;
        sf::Texture texture;
18     sf::Font font;

20     if(!texture.loadFromFile("../iu.png")){
        return -1;
22     }
        if(!font.loadFromFile("/Library/Fonts/Andale mono.ttf")){
24         return -1;
        }

26     sf::Event event;

28     bool just_arrived = true;

30     sprite.setTexture(texture);
        sprite.setColor(sf::Color::White);
32     sprite.setScale(App.getSize().x/sprite.getLocalBounds().width,App.getSize().y/sprite.
        getLocalBounds().height);

34     Button button1 = Button(font,28*2,500,500,"Go back",normal,sf::Color::Black,hover);
        button1.set_origin(0,0);
36     button1.set_dim(App.getSize().x,100);
        button1.center_text();

38     std::map<int, std::string> names;

40     names[0] = "Aggressiveness: ";
        names[1] = "Aggro sigma: ";
42     names[2] = "Global beta: ";
        names[3] = "Speed: ";
44     names[4] = "Lane 0 alpha: ";
        names[5] = "Lane 1 alpha: ";
46     names[6] = "Lane 2 alpha: ";
        names[7] = "Ramp 0 alpha: ";
48     names[8] = "Min car distance: ";
        names[9] = "Min overtake dist: ";
50     names[10] = "Max overtake dist: ";
        names[11] = "Overtake dist shutoff: ";
52     names[12] = "Min merge dist: ";
        names[13] = "Search radius around: ";
54     names[14] = "Search radius front: ";
        names[15] = "Sim multiplier: ";
56     names[16] = "Framerate: ";
        names[17] = "Ramp meter period: ";
58     names[17] = "Ramp meter period: ";

60     std::vector<Input*> inputs;

62     Input * input = new Input(font,28,500,500,names[0],normal,sf::Color::Black,hover,sf::
        Color(240,255,255,255),std::to_string(args[0][0]));
        input->set_origin(0,button1.get_bounds().height);
64     input->set_dim(App.getSize().x,50);
        input->center_text();

66     inputs.push_back(input);

68     for(int i = 1; i < args->size(); i++){

```

```

    input = new Input(font,28,500,500,names[i],normal,sf::Color::Black,hover,sf::Color
(240,255,255,255),std::to_string(args[0][i]));
72     input->set_origin(0,inputs[i-1]->get_bounds().height+inputs[i-1]->get_pos().y);
    input->set_dim(App.getSize().x,50);
74     input->center_text();
    inputs.push_back(input);
76 }

Button_bool bool_button = Button_bool(font,28,500,500,"Debug: ",normal,sf::Color::Black,
hover,sf::Color::White,"false");
bool_button.set_origin(0,input->get_bounds().height + input->get_pos().y);
80 bool_button.set_dim(App.getSize().x,50);
bool_button.set_toggled(bargs[0][0]);
82 bool_button.center_text();

Button_bool bool_button1 = Button_bool(font,28,500,500,"Ramp meter: ",normal,sf::Color::
Black,hover,sf::Color::White,"false");
bool_button1.set_origin(0,bool_button.get_bounds().height + bool_button.get_pos().y);
86 bool_button1.set_dim(App.getSize().x,50);
bool_button1.set_toggled(bargs[0][1]);
88 bool_button1.center_text();

Input * current_input = nullptr;

92 int micro = 1000000;
usleep((useconds_t)micro/8);

94 while(true){
96     while(App.pollEvent(event)){
98         if(event.type == sf::Event::Closed){
            return -1;
100         }

102         if(event.type == sf::Event::MouseButtonPressed && just_arrived){

104         }
        else if(!just_arrived && current_input == nullptr){
106             if(button1.clicked(App)){
                int i = 0;
108                 for(Input * inp : inputs){
                    args[0][i] = inp->get_val();
110                     i++;
                }
                bargs[0][0] = bool_button.get_bool();
                bargs[0][1] = bool_button1.get_bool();
112                 return 0;
            }
114             for(Input * inp : inputs){
                current_input = inp->clicked(App);
118                 if(current_input != nullptr){
                    break;
120                 }
            }
122             bool_button.clicked(App);
            bool_button1.clicked(App);
124         }
        else{
126             just_arrived = false;
128         }

130         if(event.type == sf::Event::TextEntered && current_input != nullptr){
            sf::String str = event.text.unicode;
            std::string to_str = str.toAnsiString();
            current_input = current_input->inputing(App,to_str);
132         }
134     }
}

```

```

136     App.clear();
138     App.draw(sprite);
140     App.draw(button1);
142     for(Input * inp : inputs){
143         App.draw(*inp);
144     }
145     App.draw(bool_button);
146     App.draw(bool_button1);
148     App.display();
150 }

```

../highway/cppfiles/screen2.cpp

B.10 screen3.cpp

```

//
// Created by Carl Schiller on 2019-03-06.
//
4
#include "screen3.h"
6 #include "button.h"
#include <unistd.h>
8 #include <iostream>
#include "traffic.h"
10 #include "simulation2.h"

12 screen_3::screen_3() {
13     run_bool = false;
14 };

16 int screen_3::Run(sf::RenderWindow &App, std::vector<float> * args, std::vector<bool> * bargs
17 ) {
18     sf::Color normal = sf::Color(253,246,227);
19     sf::Color hover = sf::Color(253,235,227);

20     sf::Sprite sprite;
21     sf::Texture texture;
22     sf::Font font;

24     if(!texture.loadFromFile("../iu.png")){
25         return -1;
26     }
27     if(!font.loadFromFile("/Library/Fonts/Andale mono.ttf")){
28         return -1;
29     }

30     sf::Event event;

32     bool just_arrived = true;

34     sprite.setTexture(texture);
35     sprite.setColor(sf::Color::White);
36     sprite.setScale(App.getSize().x/sprite.getLocalBounds().width,App.getSize().y/sprite.
37     getLocalBounds().height);

38     Button button1 = Button(font,28*2,500,500,"Go back",normal,sf::Color::Black,hover);
39     button1.set_origin(0,0);
40     button1.set_dim(App.getSize().x,100);
41     button1.center_text();

42     std::string stri = "Simulate for (seconds): ";
44

```



```

46     std::vector<Input *> inputs;

48     Input * input = new Input(font,28,500,500,stri,normal,sf::Color::Black,hover,sf::Color
(240,255,255,255),std::to_string(1000));
input->set_origin(0,button1.get_bounds().height+button1.getPosition().y);
50     input->set_dim(App.getSize().x,50);
input->center_text();

52     inputs.push_back(input);

54     Button button2 = Button(font,28*2,500,500,"Run simulation",normal,sf::Color::Black,hover
);
button2.set_origin(0,input->get_bounds().height+input->get_pos().y);
56     button2.set_dim(App.getSize().x,100);
button2.center_text();

60     Input * current_input = nullptr;

62     int micro = 1000000;
usleep((useconds_t)micro/8);

64     int * percent = new int;
66     *percent = 0;

68     auto traffic = new Traffic(*bargs,*args);
frame_rate = (int)args[0][16];
70     Sim sim = Sim(traffic,frame_rate,&sim_time,&run_bool,percent);

72     sim_time = 1000;
sf::Thread thread(&Sim::update,&sim);

74     bool stop_bool = false;

76     while(true) {

78         while (App.pollEvent(event)) {
80             if (event.type == sf::Event::Closed) {
return -1;
82             }

84             if (event.type == sf::Event::MouseButtonPressed && just_arrived) {

86             } else if (!just_arrived && current_input == nullptr && !stop_bool) {
if (button1.clicked(App)) {
88                 delete traffic;
delete percent;
90                 return 0;
}
92                 else if (button2.clicked(App)) {
// launch thread here...
sim_time = (int)input->get_val();
stop_bool = true;
thread.launch();
button2.set_text("Wait...");

98                 }
100                 for (Input *inp : inputs) {
current_input = inp->clicked(App);
102                 if (current_input != nullptr) {
break;
104                 }
}
106             } else {
just_arrived = false;
108             }

110             if (event.type == sf::Event::TextEntered && current_input != nullptr) {

```

```

112         sf::String str = event.text.unicode;
            std::string to_str = str.toAnsiString();
            current_input = current_input->inputting(App, to_str);
114     }
116 }
118 App.clear();
120 App.draw(sprite);
122 App.draw(button1);
124 for (Input *inp : inputs) {
    App.draw(*inp);
126 }
128 if(stop_bool){
    button2.set_text("Wait ..." + std::to_string(*percent) + "%");
130 }
132 App.draw(button2);
134 App.display();
136 if(run_bool){
    thread.terminate();
    run_bool = false;
    stop_bool = false;
    button2.set_text("Run simulation");
138 }
140 }
}

```

../highway/cppfiles/screen3.cpp

B.11 simulation.cpp

```

//
2 // Created by Carl Schiller on 2019-03-04.
//
4
#include <iostream>
6 #include "../headers/traffic.h"
#include "../headers/simulation.h"
8 #include <cmath>
#include <unistd.h>
10
12 //////////////////////////////////////
13 /// Constructor
14 /// @param traffic : pointer reference to Traffic, this is to be able to
15 /// draw traffic outside of this class.
16 /// @param mutex : mutex thread lock from SFML.
17 /// @param sim_speed : Simulation speed multiplier, e.g. 10 would mean 10x
18 /// real time speed. If simulation can not keep up it lowers this.
19 /// @param framerate : Framerate of simulation, e.g. 60 FPS. This is the
20 /// time step of the system.
21 /// @param exit_bool : If user wants to exit this is changed outside of the class.
22 Simulation::Simulation(Traffic *&traffic, sf::Mutex *&mutex, int sim_speed, int framerate,
    bool *& exit_bool):
23     m_mutex(mutex),
24     m_traffic(traffic),
25     m_exit_bool(exit_bool),
26     M_SIM_SPEED(sim_speed),
27     M_FRAMERATE(framerate)
28 {

```

```

30 }

32 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
33 /// Runs simulation. If M_SIM_SPEED = 10 , then it simulates 10x1/(MFRAMERATE)
34 /// seconds of real time simulation.

36 void Simulation::update() {
37     sf::Clock clock;
38     sf::Time time;
39     double spawn_counter_0 = 0.0;
40     double spawn_counter_1 = 0.0;
41     double spawn_counter_2 = 0.0;
42     double spawn_counter_3 = 0.0;

44     std::vector<double*> counter;
45     counter.push_back(&spawn_counter_0);
46     counter.push_back(&spawn_counter_1);
47     counter.push_back(&spawn_counter_2);
48     counter.push_back(&spawn_counter_3);

50     while(!m_exit_bool){
51         m_mutex->lock();
52         //std::cout << "calculating\n";
53         for(int i = 0; i < M_SIM_SPEED; i++){
54             //std::cout << "a\n";
55             m_traffic->update(1.0f/(float)MFRAMERATE);
56             //std::cout << "b\n";
57             m_traffic->spawn_cars(counter, 1.0f/(float)MFRAMERATE);
58             //m_mutex->lock();
59             //std::cout << "c\n";
60             m_traffic->despawn_cars();
61             //m_mutex->unlock();
62             //std::cout << "d\n";

63         }
64         //std::cout << "calculated\n";
65         m_mutex->unlock();

66         time = clock.restart();
67         sf::Int64 acutal_elapsed = time.asMicroseconds();
68         double sim_elapsed = (1.0f/(float)MFRAMERATE)*1000000;

70         if(acutal_elapsed < sim_elapsed){
71             usleep((useconds_t)(sim_elapsed-acutal_elapsed));
72             m_traffic->m_multiplier = M_SIM_SPEED;
73         }
74         else{
75             m_traffic->m_multiplier = M_SIM_SPEED*(sim_elapsed/acutal_elapsed);
76         }
77     }
78 }

```

../highway/cppfiles/simulation.cpp

B.12 simulation2.cpp

```

1  //
2  // Created by Carl Schiller on 2019-03-06.
3  //

5
6
7  #include <iostream>
8  #include "../headers/traffic.h"
9  #include "../headers/simulation2.h"
10 #include <cmath>
11 #include <unistd.h>
12 #include <iomanip>

```

```

#include <sstream>
13 #include <fstream>

15 ///////////////////////////////////////////////////////////////////
16 /// Constructor
17 /// @param traffic : pointer reference to Traffic, this is to be able to
18 /// draw traffic outside of this class.
19 /// @param mutex : mutex thread lock from SFML.
20 /// @param sim_speed : Simulation speed multiplier, e.g. 10 would mean 10x
21 /// real time speed. If simulation can not keep up it lowers this.
22 /// @param framerate : Framerate of simulation, e.g. 60 FPS. This is the
23 /// time step of the system.
24 /// @param exit_bool : If user wants to exit this is changed outside of the class.
25
26 Sim::Sim(Traffic *&traffic, int framerate, long * time, bool * finish_bool, int * percent):
27     m_traffic(traffic),
28     m_finish_bool(finish_bool),
29     MFRAMERATE(framerate),
30     sim_time(time),
31     m_percent(percent)
32 {
33 }
34
35 ///////////////////////////////////////////////////////////////////
36 /// Runs simulation. If M_SIM_SPEED = 10, then it simulates 10x1/(MFRAMERATE)
37 /// seconds of real time simulation.
38
39 void Sim::update() {
40     sf::Clock clock;
41     sf::Time time;
42
43     double spawn_counter_0 = 0.0;
44     double spawn_counter_1 = 0.0;
45     double spawn_counter_2 = 0.0;
46     double spawn_counter_3 = 0.0;
47
48     long one_percent = *sim_time*MFRAMERATE/100;
49     int per = 0;
50
51     std::vector<double*> counter;
52     counter.push_back(&spawn_counter_0);
53     counter.push_back(&spawn_counter_1);
54     counter.push_back(&spawn_counter_2);
55     counter.push_back(&spawn_counter_3);
56
57     std::vector<double> answer;
58     answer.reserve(*sim_time * MFRAMERATE);
59
60     for(int i = 0; i < *sim_time*MFRAMERATE; i++){
61         m_traffic->update(1.0f/(float)MFRAMERATE);
62         m_traffic->spawn_cars(counter, 1.0f/(float)MFRAMERATE);
63         m_traffic->despawn_cars();
64         answer.push_back(m_traffic->get_avg_flow());
65
66         if(i%one_percent == 0){
67             *m_percent = per;
68             per++;
69         }
70     }
71
72     print_to_file(&answer, *sim_time*MFRAMERATE);
73
74     *m_finish_bool = true;
75 }
76
77 void Sim::print_to_file(std::vector<double> * vec, long time_steps){
78     std::string filename;
79     auto t = std::time(nullptr);

```

```

81     auto tm = *std::localtime(&t);
82
83     std::ostringstream oss;
84     oss << std::put_time(&tm, "%d-%m-%Y-%H-%M-%S");
85     auto str = oss.str();
86
87     filename += str + "steps" + std::to_string(time_steps) + ".txt";
88
89     std::ofstream file_stream;
90     file_stream.open(filename);
91
92     for(auto subvec : *vec){
93         file_stream << subvec << std::endl;
94     }
95     file_stream.close();
96
97     std::cout << filename << " has been created\n";
98 }

```

../highway/cppfiles/simulation2.cpp

B.13 traffic.cpp

```

1  //
2  // Created by Carl Schiller on 2019-03-01.
3  //
4
5  #include <iostream>
6  #include "../headers/traffic.h"
7  #include "../headers/car.h"
8  #include "../headers/road.h"
9  #include "../headers/util.h"
10
11  //////////////////////////////////////
12  /// Constructor.
13
14  /*
15  Traffic::Traffic() {
16      debug = false;
17      if(!m_font.loadFromFile("/Library/Fonts/Andale mono.ttf")){
18
19      }
20  }
21  */
22
23  //////////////////////////////////////
24  /// Constructor with debug bool, if we want to use debugging information.
25
26  Traffic::Traffic(std::vector<bool> bargs, std::vector<float> args) :
27      debug(bargs[0]),
28      m_aggro(args[0]),
29      m_aggro_sigma(args[1]),
30      m_spawn_freq(args[2]),
31      m_speed(args[3]),
32
33      m_lane_0_spawn_prob(args[4]),
34      m_lane_1_spawn_prob(args[5]),
35      m_lane_2_spawn_prob(args[6]),
36      m_ramp_0_spawn_prob(args[7]),
37
38      m_min_dist_to_car_in_front(args[8]),
39      m_min_overtake_dist_trigger(args[9]),
40      m_max_overtake_dist_trigger(args[10]),
41      m_overtake_done_dist(args[11]),
42      m_merge_min_dist(args[12]),
43      m_search_radius_around(args[13]),

```

```

45     m_search_radius_to_car_in_front(args[14]),
46     m_ramp_meter_period(args[17]),
47     m_ramp_meter(bargs[1]),
48     m_multiplier(args[15])
49 {
50     probs.push_back(m_lane_0_spawn_prob);
51     probs.push_back(m_lane_1_spawn_prob);
52     probs.push_back(m_lane_2_spawn_prob);
53     probs.push_back(m_ramp_0_spawn_prob);
54
55     if(!m_font.loadFromFile("/Library/Fonts/Andale mono.ttf")){
56     }
57
58     Road::shared().ramp_meter_position->ramp_counter = 0;
59     Road::shared().ramp_meter_position->meter = m_ramp_meter;
60     Road::shared().ramp_meter_position->period = m_ramp_meter_period;
61
62     road_length = 0;
63
64     for(RoadSegment * seg : Road::shared().segments()){
65         if(seg->next_segment() != nullptr){
66             road_length += Util::distance(seg->get_x(), seg->next_segment()->get_x(), seg->
67             get_y(), seg->next_segment()->get_y());
68         }
69     }
70 }
71
72 //////////////////////////////////////
73 /// Copy constructor, deep copies all content.
74
75 Traffic::Traffic(const Traffic &ref) :
76     debug(ref.debug),
77     m_font(ref.m_font),
78     m_aggro(ref.m_aggro),
79     m_aggro_sigma(ref.m_aggro_sigma),
80     m_spawn_freq(ref.m_spawn_freq),
81     m_speed(ref.m_speed),
82     m_lane_0_spawn_prob(ref.m_lane_0_spawn_prob),
83     m_lane_1_spawn_prob(ref.m_lane_1_spawn_prob),
84     m_lane_2_spawn_prob(ref.m_lane_2_spawn_prob),
85     m_ramp_0_spawn_prob(ref.m_ramp_0_spawn_prob),
86     m_min_dist_to_car_in_front(ref.m_min_dist_to_car_in_front),
87     m_min_overtake_dist_trigger(ref.m_min_overtake_dist_trigger),
88     m_max_overtake_dist_trigger(ref.m_max_overtake_dist_trigger),
89     m_overtake_done_dist(ref.m_overtake_done_dist),
90     m_merge_min_dist(ref.m_merge_min_dist),
91     m_search_radius_around(ref.m_search_radius_around),
92     m_search_radius_to_car_in_front(ref.m_search_radius_to_car_in_front),
93     m_ramp_meter_period(ref.m_ramp_meter_period),
94     m_ramp_meter(ref.m_ramp_meter),
95     road_length(ref.road_length),
96     probs(ref.probs),
97     m_multiplier(ref.m_multiplier)
98 {
99     // clear values if there are any.
100     for(Car * delete_this : m_cars){
101         delete delete_this;
102     }
103     m_cars.clear();
104
105     // reserve place for new pointers.
106     m_cars.reserve(ref.m_cars.size());
107
108     // copy values into new pointers
109     for(Car * car : ref.m_cars){
110         Car * new_car_pointer = new Car(*car);
111         // *new_car_pointer = *car;

```

```

111         m_cars.push_back(new_car_pointer);
112     }
113
114     // values we copied are good, except the car pointers inside the car class.
115     std::map<int, Car*> overtake_this_car;
116     std::map<Car*, int> labeling;
117     for(int i = 0; i < m_cars.size(); i++){
118         overtake_this_car[i] = ref.m_cars[i]->overtake_this_car;
119         labeling[ref.m_cars[i]] = i;
120         m_cars[i]->overtake_this_car = nullptr; // clear copied pointers
121         //m_cars[i]->want_to_overtake_me.clear(); // clear copied pointers
122     }
123     std::map<int, int> from_to;
124     for(int i = 0; i < m_cars.size(); i++){
125         if(overtake_this_car[i] != nullptr){
126             from_to[i] = labeling[overtake_this_car[i]];
127         }
128     }
129
130     for(auto it : from_to){
131         m_cars[it.first]->overtake_this_car = m_cars[it.second];
132         //m_cars[it.second]->want_to_overtake_me.push_back(m_cars[it.first]);
133     }
134 }
135
136 ///////////////////////////////////////////////////
137 /// Copy-assignment constructor, deep copies all content and swaps.
138
139 Traffic& Traffic::operator=(const Traffic & rhs) {
140     Traffic tmp(rhs);
141
142     std::swap(debug, tmp.debug);
143     std::swap(m_font, tmp.m_font);
144     std::swap(m_cars, tmp.m_cars);
145     std::swap(m_multiplier, tmp.m_multiplier);
146     std::swap(probs, tmp.probs);
147
148     return *this;
149 }
150
151 ///////////////////////////////////////////////////
152 /// Destructor, deletes all cars.
153
154 Traffic::~Traffic() {
155     for(Car * & car : m_cars){
156         delete car;
157     }
158     Traffic::m_cars.clear();
159 }
160
161 ///////////////////////////////////////////////////
162 /// Returns size of car vector
163
164 unsigned long Traffic::n_of_cars(){
165     return m_cars.size();
166 }
167
168 ///////////////////////////////////////////////////
169 /// Random generator, returns reference to random generator in order to,
170 /// not make unnecessary copies.
171
172 std::mt19937& Traffic::my_engine() {
173     static std::mt19937 e(std::random_device{}());
174     return e;
175 }
176
177 ///////////////////////////////////////////////////
178 /// Logic for spawning cars by looking at how much time has elapsed.

```

```

179 /// @param spawn_counter : culmulative time elapsed
180 /// @param elapsed : time elapsed for one time step.
181 /// @param threshold : threshold is set by randomly selecting a poission
182 /// distributed number.
183 ///
184 /// Cars that are spawned are poission distributed in time, the speed of the
185 /// cars are normally distributed according to their agresiveness.

187 void Traffic::spawn_cars(std::vector<double*> & spawn_counter, float elapsed) {
188     int i = 0;
189     std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
190     std::vector<Car *> cars;
191     for(int j = 0; j < 4; j++){
192         cars.push_back(nullptr);
193     }

194     for(double * counter : spawn_counter){
195         if(*counter < 0){
196             std::gamma_distribution<double> dis(probs[i], m_spawn_freq);
197             std::normal_distribution<float> aggro(m_aggro, m_aggro_sigma);

198             *counter = dis(my_engine());
199             float aggressiveness = aggro(my_engine());
200             float speed = m_speed*aggressiveness;
201             float target = speed;

202             if(i < 3){
203                 Car * new_car = new Car(segments[0], i, speed, target, aggressiveness,
204                 m_min_dist_to_car_in_front,
205                 m_min_overtake_dist_trigger, m_max_overtake_dist_trigger,
206                 m_overtake_done_dist,
207                 m_merge_min_dist, m_search_radius_around,
208                 m_search_radius_to_car_in_front);
209                 cars[i] = new_car;
210             }
211             else{
212                 Car * new_car = new Car(segments[1], 0, speed, target, aggressiveness,
213                 m_min_dist_to_car_in_front,
214                 m_min_overtake_dist_trigger, m_max_overtake_dist_trigger,
215                 m_overtake_done_dist,
216                 m_merge_min_dist, m_search_radius_around,
217                 m_search_radius_to_car_in_front);
218                 cars[i] = new_car;
219             }
220             i++;
221             *counter -= elapsed;
222         }

223         for(Car * car : cars) {
224             if(car != nullptr){
225                 Car * closest_car_ahead = car->find_closest_car_ahead();

226                 if(closest_car_ahead == nullptr && closest_car_ahead != car){
227                     m_cars.push_back(car);
228                 }
229                 else{
230                     float dist = Util::distance_to_car(car, closest_car_ahead);
231                     if(dist < 10){
232                         delete car;
233                     }
234                     else if (dist < 150){
235                         car->speed() = closest_car_ahead->speed();
236                         m_cars.push_back(car);
237                     }
238                     else{
239                         m_cars.push_back(car);
240                     }
241                 }
242             }
243         }
244     }
245 }

```



```

241     }
242 }
243 }
244 }
245 ///////////////////////////////////////////////////
247 /// Despawn @param car
248
249 void Traffic::despawn_car(Car *& car) {
250     unsigned long size = m_cars.size();
251     for(int i = 0; i < size; i++){
252         if(car == m_cars[i]){
253             //std::cout << "found " << car << ", " << m_cars[i] << std::endl;
254             delete m_cars[i];
255             m_cars[i] = nullptr;
256             //std::cout << car << std::endl;
257             m_cars.erase(m_cars.begin()+i);
258             car = nullptr;
259             //std::cout << "deleted\n";
260             break;
261         }
262     }
263 }
264
265 ///////////////////////////////////////////////////
267 /// Despawn cars that are in the despawn segment.
268
269 void Traffic::despawn_cars() {
270     //std::cout << "e\n";
271     std::map<Car *, bool> to_delete;
272     for(Car * car : m_cars){
273         for(RoadSegment * seg : Road::shared().despawn_positions()){
274             if(car->get_segment() == seg){
275                 to_delete[car] = true;
276                 break;
277             }
278         }
279     }
280
281     for(Car * car : m_cars){
282         for(auto it : to_delete){
283             if(it.first == car->overtake_this_car){
284                 car->overtake_this_car = nullptr;
285             }
286         }
287     }
288
289     for(Car * & car : m_cars){
290         if(to_delete[car]){
291             delete car;
292             car = nullptr;
293         }
294     }
295
296     //std::cout << "f\n";
297     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(),
298         static_cast<Car*>(nullptr));
299     m_cars.erase(new_end, m_cars.end());
300     //std::cout << "g\n";
301 }
302
303 ///////////////////////////////////////////////////
305 /// Despawn all cars.
306
307 void Traffic::despawn_all_cars() {
308     for(Car * car : m_cars){
309         car->overtake_this_car = nullptr;

```

```

    }
309     for(Car * & car : m_cars){
311         delete car;
            car = nullptr;
313     }

315     m_cars.clear();
}

317
319     //////////////////////////////////////
319     /// Force places a new car with user specified inputs.
319     ///
321     /// \param seg : segment of car
321     /// \param node : node of car
323     /// \param vel : (current) velocity of car
323     /// \param target : target velocity of car
325     /// \param aggro : agressiveness of car

327 void Traffic::force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target,
    float aggro) {
    Car * car = new Car(seg,node,vel,target,aggro,m_min_dist_to_car_in_front,
329        m_min_overtake_dist_trigger,m_max_overtake_dist_trigger,
        m_overtake_done_dist,
        m_merge_min_dist,m_search_radius_around,
        m_search_radius_to_car_in_front);
331    m_cars.push_back(car);
}

333
335     //////////////////////////////////////
335     /// Updates traffic according by stepping @param elapsed_time seconds in time.

337 void Traffic::update(float elapsed_time) {
    if(m_ramp_meter){
339        float temp = Road::shared().ramp_meter_position->ramp_counter;
        temp += elapsed_time;
341        if(temp >= m_ramp_meter_period){
            temp -= m_ramp_meter_period;
343            Road::shared().ramp_meter_position->car_passed = false;
        }
345        Road::shared().ramp_meter_position->ramp_counter = temp;
    }

347    for(Car * & car : m_cars){
349        car->avoid_collision(elapsed_time);
    }

351    for(Car * & car : m_cars){
353        car->update_pos(elapsed_time);
    }
355 }

357     //////////////////////////////////////
357     /// Returns vector of all cars.

359 std::vector<Car *> Traffic::get_car_copies() const {
361     return m_cars;
}

363
365     //////////////////////////////////////
365     /// Returns average flow of all cars. Average value of
365     /// quotient of current speed divided by target speed for all cars.

367 float Traffic::get_avg_flow() {
369     float flow = 0;
    for(Car * car : m_cars){
371        flow += car->speed();
    }
}

```

```

373     if(m_cars.empty()){
374         return 0;
375     }
376     else{
377         return flow/(road_length);
378     }
379 }

381 ///////////////////////////////////////////////////////////////////
382 /// Returns average speeds of all cars in km/h. First entry in vector
383 /// is average speed of all cars, second entry is average speed of cars in left
384 /// lane, third entry is average speed of cars in right lane.
385
386 std::vector<float> Traffic::get_avg_speeds() {
387     std::vector<float> speedy;
388     speedy.reserve(3);
389
390     float flow = 0;
391     float flow_left = 0;
392     float flow_right = 0;
393     float i = 0;
394     float j = 0;
395     float k = 0;
396     for(Car * car : m_cars){
397         i++;
398         flow += car->speed()*3.6f;
399
400         if(car->current_segment->get_total_amount_of_lanes() == 2){
401             if(car->current_segment->get_lane_number(car->current_node) == 1){
402                 flow_left += car->speed()*3.6f;
403                 j++;
404             }
405             else{
406                 flow_right += car->speed()*3.6f;
407                 k++;
408             }
409         }
410     }
411     if(m_cars.empty()){
412         return speedy;
413     }
414     else{
415         flow = flow/i;
416         flow_left = flow_left/j;
417         flow_right = flow_right/k;
418         speedy.push_back(flow);
419         speedy.push_back(flow_left);
420         speedy.push_back(flow_right);
421         return speedy;
422     }
423 }

425 ///////////////////////////////////////////////////////////////////
426 /// Draws cars (and nodes if debug = true) to @param target, which could
427 /// be a window. Blue cars are cars that want to overtake someone,
428 /// green cars are driving as fast as they want (target speed),
429 /// red cars are driving slower than they want.
430
431 void Traffic::draw(sf::RenderTarget &target, sf::RenderStates states) const {
432     // print debug info about node placements and stuff
433
434     sf::CircleShape circle;
435     circle.setRadius(4.0f);
436     circle.setOutlineColor(sf::Color::Cyan);
437     circle.setOutlineThickness(1.0f);
438     circle.setFillColor(sf::Color::Transparent);
439
440     sf::Text segment_n;

```

```

441 segment_n.setFont(m_font);
442 segment_n.setFillColor(sf::Color::Black);
443 segment_n.setCharacterSize(14);

445 sf::VertexArray line(sf::Lines,2);
446 line[0].color = sf::Color::Blue;
447 line[1].color = sf::Color::Blue;

449 if(debug){
450     int i = 0;

451     for(RoadSegment * segment : Road::shared().segments()){
452         for(RoadNode * node : segment->get_nodes()){
453             circle.setPosition(sf::Vector2f(node->get_x()*2-4,node->get_y()*2-4));
454             line[0].position = sf::Vector2f(node->get_x()*2,node->get_y()*2);
455             for(RoadNode * connected_node : node->get_nodes_from_me()){
456                 line[1].position = sf::Vector2f(connected_node->get_x()*2,connected_node
->get_y()*2);
457                 target.draw(line,states);
458             }
459             target.draw(circle,states);

460         }
461     }
462     segment_n.setString(std::to_string(i));
463     segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4,segment->get_y()*2+4));
464     target.draw(segment_n,states);

465     i++;
466 }
467 }
468 if(m_ramp_meter){
469     RoadSegment * meter = Road::shared().ramp_meter_position;
470     circle.setPosition(sf::Vector2f(meter->get_x()*2+4-25,meter->get_y()*2-4));
471     circle.setOutlineColor(sf::Color::Black);
472     if(meter->ramp_counter > m_ramp_meter_period*0.5f){
473         circle.setFillColor(sf::Color::Green);

474     }
475     else{
476         circle.setFillColor(sf::Color::Red);

477     }
478     target.draw(circle,states);
479     circle.setOutlineColor(sf::Color::Cyan);
480     circle.setFillColor(sf::Color::Transparent);
481 }

482 // one rectangle is all we need :)
483 sf::RectangleShape rectangle;
484 rectangle.setSize(sf::Vector2f(9.4,3.4));
485 //rectangle.setFillColor(sf::Color::Green);
486 rectangle.setOutlineColor(sf::Color::Black);
487 rectangle.setOutlineThickness(2.0f);

488 //std::cout << "start drawing\n";
489 for(Car * car : m_cars){
490     if(car != nullptr){
491         //std::cout << "a\n";
492         rectangle.setPosition(car->x_pos()*2,car->y_pos()*2);
493         rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
494         unsigned int colval = (unsigned int)std::min(255.0f*(car->speed()/car->
target_speed()),255.0f);
495         sf::Uint8 colorspeed = static_cast<sf::Uint8>(colval);
496         //std::cout << "b\n";
497         if(car->overtake_this_car != nullptr){
498             rectangle.setFillColor(sf::Color(255-colorspeed,0,colorspeed,255));

499         }
500         else{
501             rectangle.setFillColor(sf::Color(255-colorspeed,colorspeed,0,255));

502         }
503     }
504 }

```

```

507     }
509     target.draw(rectangle, states);
511     // this caused crash earlier
512     if(car->heading_to_node!=nullptr && debug){
513         // print debug info about node placements and stuff
514         circle.setOutlineColor(sf::Color::Red);
515         circle.setOutlineThickness(2.0f);
516         circle.setFill_color(sf::Color::Transparent);
517         circle.setPosition(sf::Vector2f(car->current_node->get_x()*2-4,car->
current_node->get_y()*2-4));
518         target.draw(circle, states);
519         circle.setOutlineColor(sf::Color::Green);
520         circle.setPosition(sf::Vector2f(car->heading_to_node->get_x()*2-4,car->
heading_to_node->get_y()*2-4));
521         target.draw(circle, states);
522     }
523 }
524 }
525 //std::cout << "stop drawing\n";
526 }
527
528 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
529 // Modifies @param text by inserting information about Traffic,
530 // average speeds and frame rate among other things.
531
532 void Traffic::get_info(sf::Text & text, sf::Time &elapsed) {
533     //TODO: SOME BUG HERE.
534
535     float fps = 1.0f/elapsed.asSeconds();
536     unsigned long amount_of_cars = n_of_cars();
537     float flow = get_avg_flow();
538     std::vector<float> spe = get_avg_speeds();
539     std::string speedy = std::to_string(fps).substr(0,2) +
540         " fps, ncars: " + std::to_string(amount_of_cars) + "\n"
541         + "avg_flow: " + std::to_string(flow).substr(0,4) + "\n"
542         + "avg_speed: " + std::to_string(spe[0]).substr(0,5) + "km/h\n"
543         + "left_speed: " + std::to_string(spe[1]).substr(0,5) + "km/h\n"
544         + "right_speed: " + std::to_string(spe[2]).substr(0,5) + "km/h\n"
545         + "sim_multiplier: " + std::to_string(m_multiplier).substr(0,3) + "
x";
546     text.setString(speedy);
547     text.setPosition(0,0);
548     text.setFill_color(sf::Color::Black);
549     text.setFont(m_font);
550 }

```

../highway/cppfiles/traffic.cpp

B.14 unittests.cpp

```

1 //
2 // Created by Carl Schiller on 2019-03-02.
3 //
4
5 #include "unittests.h"
6 #include "road.h"
7 #include <unistd.h>
8 #include <iostream>
9
10 void Tests::placement_test() {
11     std::cout << "Starting placement tests\n";
12     std::vector<RoadSegment*> segments = Road::shared().segments();
13     int i = 0;

```

```

15     for(RoadSegment * seg : segments){
16         usleep(100000);
17         std::cout<< "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ", "<<
seg << std::endl;
18         std::cout << "next segment" << seg->next_segment() << std::endl;
19         std::vector<RoadNode*> nodes = seg->get_nodes();
20         for(RoadNode * node : nodes){
21             std::vector<RoadNode*> connections = node->get_nodes_from_me();
22             std::cout << "node" << node << " has connections:" << std::endl;
23             for(RoadNode * pointy : connections){
24                 std::cout << pointy << std::endl;
25             }
26         }
27         i++;
28         m_traffic->force_place_car(seg, seg->get_nodes()[0], 1, 1, 0.01);
29         std::cout << "placed car" << std::endl;
30     }
31     std::cout << "Placement tests passed\n";
32 }
33
34 void Tests::delete_cars_test() {
35     std::vector<Car*> car_copies = m_traffic->get_car_copies();
36
37     for(Car * car : car_copies){
38         std::cout << car << std::endl;
39         usleep(100);
40         m_mutex->lock();
41         std::cout << "deleting car\n";
42         //usleep(100000);
43         //std::cout << "Removing car " << car << std::endl;
44         m_traffic->despawn_car(car);
45         m_mutex->unlock();
46         std::cout << car << std::endl;
47     }
48     std::cout << "Car despawn tests passed\n";
49 }
50
51 void Tests::run_one_car() {
52     double ten = 10.0;
53     double zero = 0;
54     //m_traffic->spawn_cars(ten, 0, zero);
55     double fps = 60.0;
56     double multiplier = 10.0;
57     /*
58     std::cout << "running one car\n";
59     while(m_traffic->n_of_cars() != 0) {
60         usleep((useconds_t)(1000000.0/(fps*multiplier)));
61         m_traffic->update(1.0f/(float)fps);
62         m_traffic->despawn_cars();
63     }
64     */
65 }
66
67 void Tests::placement_test_2() {
68     std::cout << "Starting placement tests 2\n";
69     std::vector<RoadSegment*> segments = Road::shared().segments();
70     int i = 0;
71
72     for(RoadSegment * seg : segments){
73         usleep(100000);
74         std::cout<< "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ", "<<
seg << std::endl;
75         std::cout << "next segment" << seg->next_segment() << std::endl;
76         std::vector<RoadNode*> nodes = seg->get_nodes();
77         for(RoadNode * node : nodes){
78             std::vector<RoadNode*> connections = node->get_nodes_from_me();
79             std::cout << "node" << node << " has connections:" << std::endl;
80             for(RoadNode * pointy : connections){

```

```

81         std::cout << pointy << std::endl;
82     }
83     m_traffic->force_place_car(seg,node,1,1,0.1);
84     std::cout << "placed car" << std::endl;
85 }
86 i++;
87
88 }
89 m_traffic->despawn_all_cars();
90 std::cout << "Placement tests 2 passed\n";
91 }
92
93 void Tests::placement_test_3() {
94     std::cout << "Starting placement tests 3\n";
95     std::vector<RoadSegment*> segments = Road::shared().segments();
96
97     for (int i = 0; i < 10000; ++i) {
98         usleep(100);
99         m_traffic->force_place_car(segments[0], segments[0]->get_nodes()[0], 1, 1, 1);
100     }
101
102     delete_cars_test();
103     //m_traffic.despawn_all_cars();
104     std::cout << "Placement tests 3 passed\n";
105 }
106
107 // do all tests
108 void Tests::run_all_tests() {
109     usleep(2000000);
110     placement_test();
111     delete_cars_test();
112     run_one_car();
113     placement_test_2();
114     placement_test_3();
115
116     std::cout << "all tests passed\n";
117 }
118
119 Tests::Tests(Traffic *& traffic, sf::Mutex *& mutex) {
120     m_traffic = traffic;
121     m_mutex = mutex;
122 }
123

```

../highway/cppfiles/unittests.cpp

B.15 util.cpp

```

12 //
13 // Created by Carl Schiller on 2019-03-04.
14 //
15
16 #include "../headers/util.h"
17 #include <sstream>
18 #include <string>
19 #include <cmath>
20
21 ///////////////////////////////////////////////////////////////////
22 /// Splits @param str by @param delim, returns vector of tokens obtained.
23
24 std::vector<std::string> Util::split_string_by_delimiter(const std::string &str, const char
25     delim) {
26     std::stringstream ss(str);
27     std::string item;
28     std::vector<std::string> answer;
29     while(std::getline(ss, item, delim)){
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
254
```

```

18         answer.push_back(item);
19     }
20     return answer;
21 }
22
23 ///////////////////////////////////////////////////////////////////
24 /// Returns true if @param a is behind @param b, else false
25
26 bool Util::is_car_behind(Car * a, Car * b){
27     if(a!=b){
28         float theta_to_car_b = atan2(a->y_pos()-b->y_pos(),b->x_pos()-a->x_pos());
29         float theta_difference = get_min_angle(a->theta(),theta_to_car_b);
30         return theta_difference < M_PI*0.45;
31     }
32     else{
33         return false;
34     }
35 }
36
37 ///////////////////////////////////////////////////////////////////
38 /// Returns true if @param a will cross paths with @param b, else false.
39 /// NOTE: @param a MUST be behind @param b.
40
41 bool Util::will_car_paths_cross(Car *a, Car *b) {
42     //simulate car a driving straight ahead.
43     RoadSegment * inspecting_segment = a->get_segment();
44     //RoadNode * node_0 = a->current_node;
45     RoadNode * node_1 = a->heading_to_node;
46
47     //int node_0_int = inspecting_segment->get_lane_number(node_0);
48     int node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
49
50     while(!node_1->get_nodes_from_me().empty()){
51         for(Car * car : inspecting_segment->m_cars){
52             if(car == b){
53                 // place logic for evaluating if we cross cars here.
54                 // heading to same node, else return false
55                 return node_1 == b->heading_to_node;
56             }
57         }
58     }
59
60     inspecting_segment = node_1->get_parent_segment();
61     //node_0_int = node_1_int;
62     //node_0 = node_1;
63
64     // if we are at say, 2 lanes and heading to 2 lanes, keep previous lane numbering.
65     if(inspecting_segment->get_total_amount_of_lanes() == node_1->get_nodes_from_me().
66     size()){
67         node_1 = node_1->get_nodes_from_me()[node_1_int];
68     }
69     // if we get one option, stick to it.
70     else if(node_1->get_nodes_from_me().size() == 1){
71         node_1 = node_1->get_nodes_from_me()[0];
72     }
73     // we merge from 3 to 2.
74     else if(inspecting_segment->get_total_amount_of_lanes() == 3 && inspecting_segment->
75     merge){
76         node_1 = node_1->get_nodes_from_me()[std::max(node_1_int-1,0)];
77     }
78     node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
79 }
80
81 return false;
82 }

```



```

84  /*
86  bool Util::merge_helper(Car *a, int merge_to_lane) {
87      RoadSegment * seg = a->current_segment;
88      for(Car * car : seg->m_cars){
89          if(car != a){
90              float delta_speed = a->speed()-car->speed();
91              if(car->heading_to_node == a->current_node->get_nodes_from_me()[merge_to_lane]
92              && delta_speed < 0){
93                  return true;
94              }
95          }
96      }
97      return false;
98  }
99  */
100  /*
102  // this works only if a's heading to is b's current segment
104  bool Util::is_cars_in_same_lane(Car *a, Car *b) {
105      return a->heading_to_node == b->current_node;
106  }
107  */
108  /*
110  float Util::distance_to_line(const float theta, const float x, const float y){
112      float x_hat, y_hat;
113      x_hat = cos(theta);
114      y_hat = -sin(theta);
115
116      float proj_x = (x*x_hat+y*y_hat)*x_hat;
117      float proj_y = (x*x_hat+y*y_hat)*y_hat;
118      float dist = sqrt(abs(pow(x-proj_x, 2.0f))+abs(pow(y-proj_y, 2.0f)));
119
120      return dist;
121  }
122  */
123  /*
124  float Util::distance_to_proj_point(const float theta, const float x, const float y){
126      float x_hat, y_hat;
127      x_hat = cos(theta);
128      y_hat = -sin(theta);
129      float proj_x = (x*x_hat+y*y_hat)*x_hat;
130      float proj_y = (x*x_hat+y*y_hat)*y_hat;
131      float dist = sqrt(abs(pow(proj_x, 2.0f))+abs(pow(proj_y, 2.0f)));
132
133      return dist;
134  }
135  */
136  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
138  /// Returns distance between @param a and @param b.
139
140  float Util::distance_to_car(Car * a, Car * b){
141      if(a == nullptr || b == nullptr){
142          throw std::invalid_argument("Can't calculate distance if cars are nullptrs");
143      }
144
145      float delta_x = a->x_pos()-b->x_pos();
146      float delta_y = b->y_pos()-a->y_pos();
147
148      return sqrt(abs(pow(delta_x, 2.0f))+abs(pow(delta_y, 2.0f)));
149  }
150

```

```

152  /*
153  Car * Util::find_closest_radius(std::vector<Car> &cars, const float x, const float y){
154      Car * answer = nullptr;
155
156      float score = 100000;
157      for(Car & car : cars){
158          float distance = sqrt(abs(pow(car.x-pos()-x,2.0f))+abs(pow(car.y-pos()-y,2.0f)));
159          if(distance < score){
160              score = distance;
161              answer = &car;
162          }
163      }
164
165      return answer;
166  }
167
168  */
169
170  //////////////////////////////////////
171  /// Returns min angle between @param angl and @param ang2
172
173  float Util::get_min_angle(const float angl, const float ang2){
174      float abs_diff = abs(angl-ang2);
175      float score = std::min(2.0f*(float)M_PI-abs_diff, abs_diff);
176      return score;
177  }
178
179  //////////////////////////////////////
180  /// Returns distance between two points in 2D.
181
182  float Util::distance(float x1, float x2, float y1, float y2) {
183      return sqrt(abs(pow(x1-x2,2.0f))+abs(pow(y1-y2,2.0f)));
184  }

```

../highway/cppfiles/util.cpp

C Matlab

```

1  close all;
2  clc;
3
4  delimiterIn = ' ';
5  path = "cmake-build-debug/";
6  ext = "steps600000.txt";
7  ext2 = "steps600000ramp.txt";
8  file30 = importdata("cmake-build-debug/30steps600000.txt",delimiterIn);
9
10 strcut = cell(1,20);
11 strcutramp = cell(1,20);
12 for i=1:20
13     if(i < 10)
14         substr = strcat("0",num2str(i));
15     else
16         substr = num2str(i);
17     end
18     filename = strcat(strcat(path,substr),ext);
19     filename2= strcat(strcat(path,substr),ext2);
20     strcut{1,i} = importdata(filename,delimiterIn);
21     strcutramp{1,i} = importdata(filename2,delimiterIn);
22 end
23
24 sajs = size(strcut{1,20});
25
26 time = linspace(1,10000,sajs(1,1));

```

```

28 means = zeros(1,20);
   stds = zeros(1,20);
30 alphas = linspace(0.1,2,20);

32 meansramp = zeros(1,20);
   stdsramp = zeros(1,20);
34
   for i=1:20
36       means(1,i) = mean(strcut{1,i}(1:60000,1));
       meansramp(1,i) = mean(strcutramp{1,i});
38       stds(1,i) = std(strcut{1,i}(1:60000,1));
       stdsramp(1,i) = std(strcutramp{1,i});
40   end

42 covariance_matrix = cov(means,meansramp);
   corr_coeff = covariance_matrix(1,2)/(std(means)*std(meansramp))
44
   p = ranksum(means,meansramp);
46 p1 = ranksum(strcut{1,10}(1:60000,1),strcutramp{1,10})
   %% plots
48
   % figure
50 % e = errorbar(alphas,means,stds,'*');
   % hold on
52 % errorbar(alphas,meansramp,stdsramp,'*');
   % hold off
54 % axis([0 2 0 1.2])
   % lgd = legend({"Ramp meter off","Ramp meter on"},'Interpreter','latex');
56 % lgd.FontSize = 12;
   % tit = title("Flow rate averages",'Interpreter','latex');
58 % tit.FontSize = 16;
   % xl = xlabel("$\alpha$",'Interpreter','latex');
60 % xl.FontSize = 14;
   % yl = ylabel("Flow",'Interpreter','latex');
62 % yl.FontSize = 14;
   % print(gcf,'fig1.png','-dpng','-r300')
64 %
   % figure
66 % plot(time,strcut{1,1})
   % hold on
68 % plot(time,strcut{1,10})
   % plot(time,strcut{1,20})
70 % hold off
   % lgd2 = legend({"$\alpha$ = 0.1","$\alpha$ = 1.0","$\alpha$ = 2.0"},'Interpreter','latex');
72 % lgd2.FontSize = 16;
   %
74 % tit2 = title("Flow rate over time",'Interpreter','latex');
   % tit2.FontSize = 16;
76 % xl2 = xlabel("time (s)",'Interpreter','latex');
   % xl2.FontSize = 14;
78 % yl2 = ylabel("Flow",'Interpreter','latex');
   % yl2.FontSize = 14;
80 % print(gcf,'fig2.png','-dpng','-r300')
   %
82 % figure
   % histogram(strcut{1,1}(1:60000,1))
84 % hold on
   % histogram(strcutramp{1,1})
86 % %histogram(strcut{1,20})
   % hold off
88 % lgd2 = legend({"no ramp meter","ramp meter"},'Interpreter','latex','Location','northwest')
   %
   % lgd2.FontSize = 16;
90 %
   % tit2 = title("Flow rate $\alpha=0.1$",'Interpreter','latex');
92 % tit2.FontSize = 16;
   % xl2 = xlabel("Flow",'Interpreter','latex');

```

```

94 % x12.FontSize = 14;
    % y12 = ylabel("Number of occurances",'Interpreter','latex');
96 % y12.FontSize = 14;
    % print(gcf,'fig5.png','-dpng','-r300')
98
100 %
    % mean1 = mean(flow);
    % mean2 = mean(flow2);
102 % var1 = std(flow);
    % var2 = std(flow2);
104 % alpha1 = 2;
    % alpha2 = 1.4;
106 %
    % figure
108 % plot(time,flow)
    % hold on
110 % plot(time,flow2)
    %
112 % figure
    % e = errorbar([alpha1 alpha2],[mean1 mean2],[var1 var2],'*');
114 % axis([0 3 0 1])

```

../highway/plotter.m