

# Final Project, SI1336

Carl Schiller, 9705266436

March 7, 2019

## Abstract

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Problem formulation . . . . .	2
1.2	Complex systems . . . . .	2
<b>2</b>	<b>Method</b>	<b>3</b>
2.1	Graphs . . . . .	3
2.2	Discretization . . . . .	3
2.2.1	Speed . . . . .	3
2.2.2	Spawn rate and car headway . . . . .	3
2.2.3	Collision detection . . . . .	4
2.2.4	Acceleration . . . . .	4
2.2.5	Overtake logic and merging . . . . .	4
2.3	Graphics rendering . . . . .	5
<b>3</b>	<b>Result</b>	<b>5</b>
<b>4</b>	<b>Discussion</b>	<b>5</b>
<b>A</b>	<b>Header files</b>	<b>7</b>
A.1	cars.h . . . . .	7
A.2	road.h . . . . .	8
A.3	roadnode.h . . . . .	9
A.4	roadsegment.h . . . . .	9
A.5	simulation.h . . . . .	10
A.6	traffic.h . . . . .	11
A.7	unittests.h . . . . .	12
A.8	util.h . . . . .	13
<b>B</b>	<b>Source files</b>	<b>14</b>
B.1	cars.cpp . . . . .	14
B.2	main.cpp . . . . .	22
B.3	road.cpp . . . . .	23
B.4	roadnode.cpp . . . . .	26
B.5	roadsegment.cpp . . . . .	28
B.6	simulation.cpp . . . . .	31
B.7	traffic.cpp . . . . .	33
B.8	unittests.cpp . . . . .	41
B.9	util.cpp . . . . .	43



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).

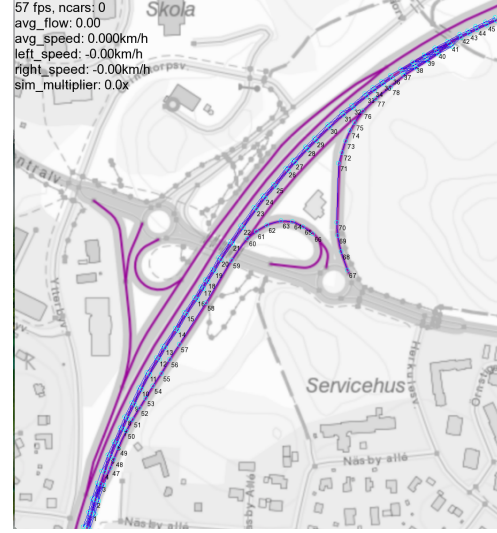


Figure 2: Setup of road with vertices and edges.

### 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.  $\sigma$  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]

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

where  $\alpha$  is the "shape" factor and  $\beta$  is the "rate" factor which are tweakable according to which behavior is sought after. The expected mean of a stochastic variable is  $\frac{\alpha}{\beta}$ , with variance  $\frac{\alpha}{\beta^2}$ . This means, a larger  $\beta$  implies a less 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 <
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){
11             overtake_this_car = closest_car;
13         }
15     }
17     if(overtake_this_car != nullptr){
18         if(Util::is_car_behind(overtake_this_car, this) && (Util::distance_to_car(this,
overtake_this_car) > m_overtake_done_dist)){
19             overtake_this_car = nullptr;
20         }
21     }

```

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.

## 3 Result

The following parameters have been used.

Agressiveness	1.0
Agressiveness sigma	0.2
Global alpha	2.0
Mean speed	20 (m/s)
Lane 0 beta	5.0
Lane 1 beta	1.0
Lane 2 beta	1.0
Ramp 0 beta	5.0
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	10.0 (s)

## 4 Discussion

## 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%202%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](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](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 cars.h

```
1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #ifndef HIGHWAY_CAR_H
6 #define HIGHWAY_CAR_H
7
8 //////////////////////////////////////
9 //
10 // Car
11 //
12 // Describes a car that moves around in Road class
13 //
14 //////////////////////////////////////
15
16 #include <map>
17 #include "roadnode.h"
18 #include "roadsegment.h"
19
20 class Car{
21 private:
22     float m_dist_to_next_node;
23     float m_speed;
24     float m_theta; // radians
25
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;
41
42     Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float
43     agresivness,
44         float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
45     m_max_overtake_dist_trigger,
46         float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
47         float m_search_radius_to_car_in_front);
48     Car(RoadSegment * spawn_point, RoadNode * lane, float vel, float target_speed, float
49     agresivness,
50         float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
51     m_max_overtake_dist_trigger,
52         float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
53         float m_search_radius_to_car_in_front);
54
55     // all are raw pointers
56     RoadSegment * current_segment;
57     RoadNode * current_node;
58     RoadNode * heading_to_node;
59     Car * overtake_this_car;
60
61     void update_pos(float delta_t);
62     void merge(std::vector<RoadNode*> & connections);
63     void do_we_want_to_overtake(Car * & closest_car, int & current_lane);
```

```

61 void accelerate(float delta_t);
void avoid_collision(float delta_t);
Car * find_closest_car_ahead();
63 std::map<Car *, bool> find_cars_around_car();

65 float x_pos();
float y_pos();

67 float & speed();
69 float & target_speed();
float & theta();

71 RoadSegment * get_segment();
73 };
75 #endif //HIGHWAY_CAR.H

```

../highway/headers/car.h

## A.2 road.h

```

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

5 #ifndef HIGHWAY_ROAD_H
#define HIGHWAY_ROAD_H

7
9 ////////////////////////////////////////////////////
9 // Road
11 //
11 // Describes a road with interconnected nodes. Mathematically it is
13 // a graph.
13 //
15 ////////////////////////////////////////////////////

17 #include "roadsegment.h"
#include <vector>
19 #include <string>

21 class Road{
private:
23     std::vector<RoadSegment*> m_segments; // OWNERSHIP
std::vector<RoadSegment*> m_spawn_positions; // raw pointers
25     std::vector<RoadSegment*> m_despawn_positions; // raw pointers

27     const std::string M_FILENAME;
private:
29     Road();
~Road();
31 public:
static Road &shared() {static Road road; return road;} // in order to only load road
once in memory

33     Road(const Road& copy) = delete; // no copying allowed
35     Road& operator=(const Road& rhs) = delete; // no copying allowed

37     bool load_road();
std::vector<RoadSegment*> & spawn_positions();
39     std::vector<RoadSegment*> & despawn_positions();
std::vector<RoadSegment*> & segments();
41     RoadSegment * ramp_meter_position;
};
43 #endif //HIGHWAY_ROAD_H

```



### A.3 roadnode.h

```
//
2 // Created by Carl Schiller on 2019-03-04.
//
4
6 #ifndef HIGHWAY_ROADNODE_H
7 #define HIGHWAY_ROADNODE_H
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
21 class RoadNode{
22 private:
23     float m_x, m_y;
24     std::vector<RoadNode*> m_nodes_from_me; // raw pointers, no ownership
25     std::vector<RoadNode*> m_nodes_to_me;
26     RoadSegment* m_is_child_of; // raw pointer, no ownership
27 public:
28     RoadNode();
29     ~RoadNode();
30     RoadNode(float x, float y, RoadSegment * segment);
31
32     void set_next_node(RoadNode *);
33     void set_previous_node(RoadNode *);
34     RoadSegment* get_parent_segment();
35     RoadNode * get_next_node(int lane);
36     std::vector<RoadNode*> & get_nodes_from_me();
37     std::vector<RoadNode*> & get_nodes_to_me();
38     float get_x();
39     float get_y();
40     float get_theta(RoadNode*);
41 };
42
43 #endif //HIGHWAY_ROADNODE_H
```

../highway/headers/roadnode.h

### A.4 roadsegment.h

```
//
2 // Created by Carl Schiller on 2019-03-04.
//
4
6 #ifndef HIGHWAY_ROADSEGMENT_H
7 #define HIGHWAY_ROADSEGMENT_H
8
9 ////////////////////////////////////////////////////////////////////
10 //
```

```

10 // RoadSegment //
11 // //
12 // Describes a container for several RoadNodes //
13 // //
14 //////////////////////////////////////
15
16 #include <vector>
17
18 class RoadNode;
19
20 class Car;
21
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);
37     ~RoadSegment(); // rule of three
38     RoadSegment(const RoadSegment&) = delete; // rule of three
39     RoadSegment& operator=(const RoadSegment& rhs) = delete; // rule of three
40
41     bool merge;
42     std::vector<Car*> m_cars; // raw pointer, no ownership
43     float ramp_counter;
44     bool car_passed;
45     bool meter;
46     float period;
47
48     RoadNode * get_node_pointer(int n);
49     std::vector<RoadNode *> get_nodes();
50     void append_car(Car*);
51     void remove_car(Car*);
52     RoadSegment * next_segment();
53     float get_theta();
54     const float get_x() const;
55     const float get_y() const;
56
57     int get_lane_number(RoadNode *);
58     const int get_total_amount_of_lanes() const;
59     void set_theta(float theta);
60     void set_next_road_segment(RoadSegment*);
61     void calculate_theta();
62     void calculate_and_populate_nodes();
63     void set_all_node_pointers_to_next_segment();
64     void set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *);
65 };
66 #endif //HIGHWAYROADSEGMENT.H

```

../highway/headers/roadsegment.h

## A.5 simulation.h

```

1 //
2 // Created by Carl Schiller on 2018-12-19.
3 //

```

```

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 #endif //HIGHWAY_WINDOW.H

```

../highway/headers/simulation.h

## A.6 traffic.h

```

1 //
2 // Created by Carl Schiller on 2018-12-19.
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 from 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;

```

```

29     const float m_aggro;
    const float m_aggro_sigma;
31     const float m_spawn_freq;
    const float m_speed;

33
    const float m_lane_0_spawn_prob;
35     const float m_lane_1_spawn_prob;
    const float m_lane_2_spawn_prob;
37     const float m_ramp_0_spawn_prob;

39
    const float m_min_dist_to_car_in_front;
    const float m_min_overtake_dist_trigger;
41     const float m_max_overtake_dist_trigger;
    const float m_overtake_done_dist;
43     const float m_merge_min_dist;
    const float m_search_radius_around;
45     const float m_search_radius_to_car_in_front;

47
    const float m_ramp_meter_period;
    const bool m_ramp_meter;

49
    std::vector<float> probs;
51 public:
    Traffic() = delete;
53     Traffic(std::vector<bool> bargs, std::vector<float> args);
    ~Traffic();
55     Traffic(const Traffic&); // rule of three
    Traffic& operator=(const Traffic&); // rule of three
57
    unsigned long n_of_cars();
59     void spawn_cars(std::vector<double*> & counters, float elapsed);
    void despawn_cars();
61     void despawn_all_cars();
    void despawn_car(Car*& car);
63     void force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, float
    aggro);

65
    void update(float elapsed_time);
67     std::vector<Car*> get_car_copies() const;
    float get_avg_flow();
69     std::vector<float> get_avg_speeds();
private:
71     virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
public:
73     void get_info(sf::Text & text, sf::Time &elapsed);
    double m_multiplier;
75 };
77 #endif //HIGHWAY_TRAFFIC.H

```

../highway/headers/traffic.h

## A.7 unittests.h

```

1 //
  // Created by Carl Schiller on 2019-01-16.
3 //

5
  #ifndef HIGHWAY_UNITTESTS_H
7  #define HIGHWAY_UNITTESTS_H

9  ////////////////////////////////////////
  //
11 // Tests
  //

```

```

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 *amp traffic , sf::Mutex *amp mutex);
32
33     void run_all_tests();
34 };
35
36 #endif //HIGHWAY_UNITTESTS_H

```

../highway/headers/unittests.h

## A.8 util.h

```

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

```

../highway/headers/util.h

## B Source files

### B.1 cars.cpp

```
//
2 // Created by Carl Schiller on 2019-03-04.
//
4
#include "../headers/car.h"
6 #include <map>
#include <cmath>
8 #include <list>
#include <iostream>
10 #include "../headers/util.h"

12 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
13 /// Constructor.
14
Car::Car() :
16     m_speed(0),
    m_aggressiveness(0),
18     m_target_speed(0),
    m_min_dist_to_car_in_front(0),
20     m_min_overtake_dist_trigger(0),
    m_max_overtake_dist_trigger(0),
22     m_overtake_done_dist(0),
    m_merge_min_dist(0),
24     m_search_radius_around(0),
    m_search_radius_to_car_in_front(0),
26     current_segment(nullptr),
    current_node(nullptr),
28     overtake_this_car(nullptr)
{
30 }
32

34 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
35 /// Constructor for new car with specified lane numbering in spawn point.
36 /// Lane numbering @param lane must not exceed amount of lanes in
37 /// @param spawn_point, otherwise an exception will be thrown.
38
Car::Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float
    agressivness,
40     float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
    m_max_overtake_dist_trigger,
    float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
42     float m_search_radius_to_car_in_front) :
    m_speed(vel),
44     m_aggressiveness(agressivness),
    m_target_speed(target_speed),
46     m_min_dist_to_car_in_front(m_min_dist_to_car_in_front),
    m_min_overtake_dist_trigger(m_min_overtake_dist_trigger),
48     m_max_overtake_dist_trigger(m_max_overtake_dist_trigger),
    m_overtake_done_dist(m_overtake_done_dist),
50     m_merge_min_dist(m_merge_min_dist),
    m_search_radius_around(m_search_radius_around),
52     m_search_radius_to_car_in_front(m_search_radius_to_car_in_front),
    current_segment(spawn_point),
54     current_node(current_segment->get_node_pointer(lane)),
    overtake_this_car(nullptr)
56 {
    current_segment->append_car(this);
58
    if(!current_node->get_nodes_from_me().empty()){
60         heading_to_node = current_node->get_next_node(lane);
```

```

62     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());

64     m_theta = current_node->get_theta(heading_to_node);
}
66 else{
    throw std::invalid_argument("Car spawns in node with empty connections, or with a
68     nullptr segment");
}
70 }

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
agressivness,
    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):
80     m_speed(vel),
    m_agressiveness(agressivness),
82     m_target_speed(target_speed),
    m_min_dist_to_car_in_front(m_min_dist_to_car_in_front),
84     m_min_overtake_dist_trigger(m_min_overtake_dist_trigger),
    m_max_overtake_dist_trigger(m_max_overtake_dist_trigger),
86     m_overtake_done_dist(m_overtake_done_dist),
    m_merge_min_dist(m_merge_min_dist),
88     m_search_radius_around(m_search_radius_around),
    m_search_radius_to_car_in_front(m_search_radius_to_car_in_front),
90     current_segment(spawn_point),
    current_node(lane),
92     overtake_this_car(nullptr)
{
94     current_segment->append_car(this);

96     if(!current_node->get_nodes_from_me().empty() || current_segment->next_segment() !=
nullptr){
        heading_to_node = current_node->get_next_node(0);

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());

100         m_theta = current_node->get_theta(heading_to_node);
    }
102 else{
104     throw std::invalid_argument("Car spawns in node with empty connections, or with a
    nullptr segment");
    }
106 }

108 //////////////////////////////////////////////////
109 /// Destructor for car.

110 Car::~Car(){
112     if(this->current_segment != nullptr){
        this->current_segment->remove_car(this); // remove this pointer shit
114     }

116     overtake_this_car = nullptr;
    current_segment = nullptr;
118     heading_to_node = nullptr;
    current_node = nullptr;
120 }

122 //////////////////////////////////////////////////

```

```

124  /// Updates position for car with time step @param delta_t.
125
126 void Car::update_pos(float delta_t) {
127     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 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
156 /// Function to determine if we can merge into another lane depending on.
157 /// properties of @param connections.
158
159 void Car::merge(std::vector<RoadNode*> & connections) {
160     // check if we merge
161     int current_lane = current_segment->get_lane_number(current_node);
162     bool can_merge = true;
163     std::map<Car*, bool> cars_around_car = find_cars_around_car();
164     Car * closest_car = find_closest_car_ahead();
165
166     for(auto it : cars_around_car){
167         float delta_dist = Util::distance_to_car(it.first, this);
168         float delta_speed = abs(speed()-it.first->speed());
169
170         if(current_lane == 0 && it.first->heading_to_node->get_parent_segment()->
171         get_lane_number(it.first->heading_to_node) == 1 ){
172             can_merge =
173                 delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, m_merge_min_dist
174             );
175         }
176         else if(current_lane == 1 && it.first->heading_to_node->get_parent_segment()->
177         get_lane_number(it.first->heading_to_node) == 0){
178             can_merge =
179                 delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, m_merge_min_dist
180             );
181         }
182
183         if(!can_merge){
184             break;
185         }
186     }
187
188     if(current_segment->merge){

```



```

186     if (current_lane == 0 && connections[0]->get_parent_segment()->
get_total_amount_of_lanes() != 2){
187         if (can_merge){
188             heading_to_node = connections[1];
189         }
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 }

```

```

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) {
257     //see if we want to overtake car.
258
259     if(closest_car != nullptr){
260         //float delta_speed = closest_car->speed()-speed();
261         float delta_distance = Util::distance_to_car(this , closest_car);
262
263         if(overtake_this_car == nullptr){
264             if(delta_distance > m_min_overtake_dist_trigger && delta_distance <
265                 m_max_overtake_dist_trigger && (target_speed()/closest_car->target_speed() >
266                 m_aggressiveness*1.0f ) && current_lane == 0 && closest_car->current_node->
267                 get_parent_segment()->get_lane_number(closest_car->current_node) == 0){
268                 overtake_this_car = closest_car;
269             }
270         }
271
272         if(overtake_this_car != nullptr){
273             if(Util::is_car_behind(overtake_this_car , this) && (Util::distance_to_car(this ,
274                 overtake_this_car) > m_overtake_done_dist)){
275                 overtake_this_car = nullptr;
276             }
277         }
278     }
279 }
280
281 ///////////////////////////////////////////////////////////////////
282 /// Function to accelerate this car.
283
284 void Car::accelerate(float elapsed){
285     float target = m_target_speed;
286     float d_vel; // proportional control.
287
288     if(m_speed < target*0.75){
289         d_vel = m_aggressiveness*elapsed*2.0f;
290     }
291     else{
292         d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
293     }
294
295     m_speed += d_vel;
296 }
297
298 ///////////////////////////////////////////////////////////////////
299 /// Helper function to avoid collision with another car.
300
301 void Car::avoid_collision(float delta_t) {
302     float min_distance = m_min_dist_to_car_in_front; // for car distance.
303     float ideal = min_distance+min_distance*(m_speed/20.f);
304
305     Car * closest_car = find_closest_car_ahead();
306     float detection_distance = m_speed*5.0f;
307
308     if(closest_car != nullptr) {
309         float radius_to_car = Util::distance_to_car(this , closest_car);
310         float delta_speed = closest_car->speed() - this->speed();
311
312         if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
313             m_speed -= std::max(std::max((radius_to_car-min_distance)*0.5f,0.0f),10.0f*
314             delta_t);
315         }
316         else if(radius_to_car < min_distance){
317             m_speed -= std::max(std::max((min_distance-radius_to_car)*0.5f,0.0f),2.0f*
318             delta_t);
319         }
320     }
321 }

```

```

314     }
315     else if(delta_speed < 0 && radius_to_car < detection_distance){
316         m_speed -= std::min(
317             abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
318             m_aggressiveness * 0.15f,
319             10.0f * delta_t);
320     }
321     else {
322         accelerate(delta_t);
323     }
324
325     if(current_segment->merge){
326         std::map<Car*, bool> around = find_cars_around_car();
327         for(auto it : around){
328             float delta_dist = Util::distance_to_car(it.first, this);
329             delta_speed = abs(speed()-it.first->speed());
330
331             if(it.first->current_node->get_parent_segment()->get_lane_number(it.first->
332             current_node) == 0 && delta_dist < ideal && this->current_segment->get_lane_number(
333             current_node) == 1 && speed()/target_speed() > 0.5){
334                 if(Util::is_car_behind(it.first, this)){
335                     accelerate(delta_t);
336                 }
337                 else{
338                     m_speed -= std::max(std::max((ideal-delta_dist)*0.5f, 0.0f), 10.0f*
339                     delta_t);
340                 }
341             }
342             else if(it.first->current_node->get_parent_segment()->get_lane_number(it.
343             first->current_node) == 1 && this->current_segment->get_lane_number(current_node) == 0
344             && speed()/target_speed() > 0.5 && delta_dist < ideal){
345                 if(Util::is_car_behind(this, it.first)){
346                     m_speed -= std::max(std::max((ideal-delta_dist)*0.5f, 0.0f), 10.0f*
347                     delta_t);
348                 }
349                 else{
350                     accelerate(delta_t);
351                 }
352             }
353         }
354     }
355     else{
356     }
357 }
358
359 if(heading_to_node->get_parent_segment()->meter){
360     if(heading_to_node->get_parent_segment()->car_passed || heading_to_node->
361     get_parent_segment()->ramp_counter < heading_to_node->get_parent_segment()->period*0.5f)
362     {
363         if (m_dist_to_next_node < ideal) {
364             m_speed -= std::max(std::max((m_dist_to_next_node-min_distance)*0.5f, 0.0f)
365             ,10.0f*delta_t);
366         }
367         else if(m_dist_to_next_node < detection_distance){
368             m_speed -= std::min(
369                 abs(pow(m_speed, 2.0f)) * pow(ideal * 0.25f / m_dist_to_next_node,
370                 2.0f) * m_aggressiveness * 0.15f,
371                 10.0f * delta_t);
372         }
373     }
374     else{
375         accelerate(delta_t);
376     }
377 }
378 else{
379     accelerate(delta_t);
380 }
381 }

```

```

372     if(m_speed < 0){
373         m_speed = 0;
374     }
375 }
376
377
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 ///////////////////////////////////////////////////////////////////
427 /// Searches for cars around this car in a specified radius. Note that
428 /// search radius is the radius to RoadNodes, and not surrounding cars.
429 /// Returns a map of cars the function has found.
430
431 std::map<Car *,bool> Car::find_cars_around_car() {
432     float search_radius = m_search_radius_around;
433     std::map<RoadNode*,bool> visited;
434     std::list<RoadNode*> queue;
435
436     for(RoadNode * node : (this->current_segment->get_nodes())){
437         queue.push_front(node);

```

```

}
438
std::map<Car *, bool> answer;
440 while(!queue.empty()){
    RoadNode * next_node = queue.back(); // get last element
442 queue.pop_back(); // remove element

    if(next_node != nullptr){
        if(!visited[next_node] && Util::distance(x_pos(), next_node->get_x(), y_pos(),
444 next_node->get_y()) < search_radius){
            visited[next_node] = true;
            for(Car * car : next_node->get_parent_segment()->m_cars){
446                 if(this != car){
448                     answer[car] = true;
450                 }
            }
            // push in new nodes in front of list.
            for(RoadNode * node : next_node->get_nodes_from_me()){
452                 queue.push_front(node);
454             }

            for(RoadNode * node : next_node->get_nodes_to_me()){
456                 queue.push_front(node);
458             }
460         }
    }
462 }
return answer;
464 }

////////////////////////////////////
466 /// Returns x position of car.
468
float Car::x_pos() {
470     float x_position;
    if(heading_to_node != nullptr){
472         x_position = heading_to_node->get_x()-m_dist_to_next_node*cos(m_theta);
    }
    else{
474         x_position = current_node->get_x();
476     }

    return x_position;
478 }

////////////////////////////////////
480 /// Returns y position of car.
482

float Car::y_pos() {
484     float y_position;
    if(heading_to_node != nullptr){
486         y_position = heading_to_node->get_y()+m_dist_to_next_node*sin(m_theta);
488     }
    else{
490         y_position = current_node->get_y();
    }

    return y_position;
492 }

////////////////////////////////////
494 /// Returns speed of car, as reference.
496

float & Car::speed() {
498     return m_speed;
500 }
502
////////////////////////////////////

```

```

504 /// Returns target speed of car as reference.
506 float & Car::target_speed() {
508     return m_target_speed;
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;
520 }
522 /// Returns current segment car is in.
524 RoadSegment* Car::get_segment() {
526     return current_segment;
528 }

```

../highway/cppfiles/car.cpp

## B.2 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<Screen*> 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;

```

```

43     args.push_back(m_ramp_0_spawn_prob);
45     float m_min_dist_to_car_in_front = 8;
46     args.push_back(m_min_dist_to_car_in_front);
47     float m_min_overtake_dist_trigger = 10;
48     args.push_back(m_min_overtake_dist_trigger);
49     float m_max_overtake_dist_trigger = 40;
50     args.push_back(m_max_overtake_dist_trigger);
51     float m_overtake_done_dist = 30;
52     args.push_back(m_overtake_done_dist);
53     float m_merge_min_dist = 15.0f;
54     args.push_back(m_merge_min_dist);
55     float m_search_radius_around = 30;
56     args.push_back(m_search_radius_around);
57     float m_search_radius_to_car_in_front = 50;
58     args.push_back(m_search_radius_to_car_in_front);
59     float sim_speed = 10;
60     args.push_back(sim_speed);
61     float framerate = 60;
62     args.push_back(framerate);
63     float ramp_meter_period = 10;
64     args.push_back(ramp_meter_period);
65
66     std::vector<bool> bool_args;
67     bool debug = false;
68     bool_args.push_back(debug);
69     bool ramp_meter = false;
70     bool_args.push_back(ramp_meter);
71
72     while(screen >= 0){
73         screen = Screens[screen]->Run(App,&args,&bool_args);
74     }
75
76     return 0;
77 }

```

../highway/cppfiles/main.cpp

## B.3 road.cpp

```

//
2  // Created by Carl Schiller on 2019-03-04.
3  //
4
5  #include "../headers/road.h"
6  #include <fstream>
7  #include <vector>
8  #include "../headers/roadsegment.h"
9  #include <iostream>
10 #include "../headers/util.h"
11
12 //////////////////////////////////////
13 /// Constructor of Road.
14
15 Road::Road() :
16     MFILENAME("../road.txt")
17 {
18     if(!load_road()){
19         std::cout << "Error in loading road.\n";
20     };
21 }
22
23 //////////////////////////////////////
24 /// Destructor of Road.
25
26 Road::~~Road() {

```

```

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

32
33     //////////////////////////////////////
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() {
    bool loading = true;
62     std::ifstream stream;
    stream.open(MFILENAME);

64
65     std::vector<std::vector<std::string>> road_vector;
66     road_vector.reserve(100);

67
68     if(stream.is_open()){
        std::string line;
        std::vector<std::string> tokens;
        while(std::getline(stream, line)){
70             tokens = Util::split_string_by_delimiter(line, ' ');
71             if(tokens[0] != "#"){
72                 road_vector.push_back(tokens);
73             }
74         }
75     }
76
77     else{
78         loading = false;
79     }
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             }

```



```

142         ramp.meter_position = seg;
143     }
144     else{
145         RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std
146         ::stoi(vec[3]),false);
147         m_segments.push_back(seg);
148     }
149 }
150 else{
151     RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std::
152     stoi(vec[3]),false);
153     m_segments.push_back(seg);
154 }
155 }
156
157 // populate nodes.
158 for (int i = 0; i < m_segments.size(); ++i) {
159     // populate nodes normally.
160     if(road_vector[i].size() == 4){
161         m_segments[i]->set_next_road_segment(m_segments[i+1]);
162         m_segments[i]->calculate_theta();
163         // calculate nodes based on theta.
164         m_segments[i]->calculate_and_populate_nodes();
165     }
166     else if(road_vector[i].size() == 5){
167         if(road_vector[i][4] == "false"){
168             // take previous direction and populate nodes.
169             m_segments[i]->set_theta(m_segments[i-1]->get_theta());
170             m_segments[i]->calculate_and_populate_nodes();
171             // but do not connect nodes to new ones.
172
173             // make this a despawn segment
174             m_despawn_positions.push_back(m_segments[i]);
175         }
176         else if(road_vector[i][4] == "true"){
177             m_segments[i]->set_next_road_segment(m_segments[i+1]);
178             m_segments[i]->calculate_theta();
179             // calculate nodes based on theta.
180             m_segments[i]->calculate_and_populate_nodes();
181
182             // make this a spawn segment
183             m_spawn_positions.push_back(m_segments[i]);
184         }
185         else if(road_vector[i][4] == "merge" || road_vector[i][4] == "ramp"){
186             m_segments[i]->set_next_road_segment(m_segments[i+1]);
187             m_segments[i]->calculate_theta();
188             // calculate nodes based on theta.
189             m_segments[i]->calculate_and_populate_nodes();
190         }
191     }
192     // else we connect one by one.
193     else{
194         // take previous direction and populate nodes.
195         m_segments[i]->set_theta(m_segments[i-1]->get_theta());
196         // calculate nodes based on theta.
197         m_segments[i]->calculate_and_populate_nodes();
198     }
199 }
200
201 // connect nodes.
202 for (int i = 0; i < m_segments.size(); ++i) {
203     // do normal connection, ie connect all nodes.
204     if(road_vector[i].size() == 4){
205         m_segments[i]->set_all_node_pointers_to_next_segment();
206     }
207 }

```

```

160         else if(road_vector[i].size() == 5){
161             if(road_vector[i][4] == "false"){
162                 // but do not connect nodes to new ones.
163             }
164             else if(road_vector[i][4] == "true" || road_vector[i][4] == "merge" ||
165                 road_vector[i][4] == "ramp"){
166                 m_segments[i]->set_all_node_pointers_to_next_segment();
167             }
168         }
169         // else we connect one by one.
170         else{
171             // manually connect nodes.
172             int amount_of_pointers = (int)road_vector[i].size()-4;
173             for(int j = 0; j < amount_of_pointers/3; j++){
174                 int current_pos = 4+j*3;
175                 RoadSegment * next_segment = m_segments[std::stoi(road_vector[i][current_pos
176                 +2])];
177                 m_segments[i]->set_node_pointer_to_node(std::stoi(road_vector[i][current_pos
178                 ]),std::stoi(road_vector[i][current_pos+1]),next_segment);
179             }
180         }
181     }
182     return loading;
183 }
184
185 //////////////////////////////////////
186 /// Returns spawn positions of Road
187
188 std::vector<RoadSegment*>& Road::spawn_positions() {
189     return m_spawn_positions;
190 }
191
192 //////////////////////////////////////
193 /// Returns despawn positions of Road
194
195 std::vector<RoadSegment*>& Road::despawn_positions() {
196     return m_despawn_positions;
197 }
198
199 //////////////////////////////////////
200 /// Returns all segments of Road.
201
202 std::vector<RoadSegment*>& Road::segments() {
203     return m_segments;
204 }

```

../highway/cppfiles/road.cpp

## B.4 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

```

```

RoadNode::~~RoadNode() = default;

17
19 //////////////////////////////////////////////////
21 // Constructor, @param x is x position of node, @param y is y position of node,
23 // @param segment is to which segment this RoadNode belongs.
25
RoadNode::RoadNode(float x, float y, RoadSegment * segment) {
23     m_x = x;
25     m_y = y;
25     m_is_child_of = segment;
27 }

27 //////////////////////////////////////////////////
29 // Appends a new RoadNode to the list connections from this RoadNode.
31 // I.e. to where a Car is allowed to drive.

void RoadNode::set_next_node(RoadNode * next_node) {
33     m_nodes_from_me.push_back(next_node);
35     next_node->m_nodes_to_me.push_back(this); // sets double linked chain.
37 }

37 //////////////////////////////////////////////////
39 // Appends a new RoadNode to the list connections to this RoadNode.
41 // I.e. from where a Car is allowed to drive to this Node.

void RoadNode::set_previous_node(RoadNode * prev_node) {
43     m_nodes_to_me.push_back(prev_node);
45 }

45 //////////////////////////////////////////////////
47 // Returns RoadSegment to which this RoadNode belongs.

RoadSegment* RoadNode::get_parent_segment() {
49     return m_is_child_of;
51 }

51 //////////////////////////////////////////////////
53 // Returns connections from this RoadNode.

std::vector<RoadNode*> & RoadNode::get_nodes_from_me() {
55     return m_nodes_from_me;
57 }

57 //////////////////////////////////////////////////
59 // Returns connections to this RoadNode.

std::vector<RoadNode*>& RoadNode::get_nodes_to_me() {
61     return m_nodes_to_me;
63 }

63 //////////////////////////////////////////////////
65 // Returns x position of RoadNode.

float RoadNode::get_x() {
67     return m_x;
69 }

69 //////////////////////////////////////////////////
71 // Returns y position of RoadNode.

float RoadNode::get_y() {
73     return m_y;
75 }

75 //////////////////////////////////////////////////
77 // Returns angle of this RoadNode to @param node as a mathematician
79 // would define angles. In radians.

81
83

```

```

float RoadNode::get_theta(RoadNode* node) {
85     for(RoadNode * road_node : m_nodes_from_me){
            if(node == road_node){
87                 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.5 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 RoadNode 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) :

```

```

46         m_x(x),
47         m_y(y),
48         m_theta(theta),
49         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    }

```

```

114     throw std::invalid_argument("Node is not in this segment");
115 }
116
117 //////////////////////////////////////////////////
118 /// Adds a new car to the segment.
119
120 void RoadSegment::append_car(Car * car) {
121     m_cars.push_back(car);
122 }
123
124 //////////////////////////////////////////////////
125 /// Removes car from segment, if car is not in list we do nothing.
126
127 void RoadSegment::remove_car(Car * car) {
128     unsigned long size = m_cars.size();
129     bool found = false;
130     for(int i = 0; i < size; i++){
131         if(car == m_cars[i]){
132             m_cars[i] = nullptr;
133             found = true;
134         }
135     }
136     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(),
137         static_cast<Car*>(nullptr));
138     m_cars.erase(new_end, m_cars.end());
139
140     /*
141     if(!found){
142         throw std::invalid_argument("Car is not in this segment.");
143     }
144     */
145 }
146
147 //////////////////////////////////////////////////
148 /// Sets theta of RoadSegment according to @param theta.
149
150 void RoadSegment::set_theta(float theta) {
151     m_theta = theta;
152 }
153
154 //////////////////////////////////////////////////
155 /// Automatically populates segment with nodes according to amount of lanes
156 /// specified and theta specified.
157
158 void RoadSegment::calculate_and_populate_nodes() {
159     // calculates placement of nodes.
160     float total_length = M_LANE_WIDTH*(m_n_lanes-1);
161     float current_length = -total_length/2.0f;
162
163     for(int i = 0; i < m_n_lanes; i++){
164         float x_pos = m_x+current_length*cos(m_theta+(float)M_PI*0.5f);
165         float y_pos = m_y-current_length*sin(m_theta+(float)M_PI*0.5f);
166         m_nodes.push_back(new RoadNode(x_pos, y_pos, this));
167         current_length += M_LANE_WIDTH;
168     }
169 }
170
171 //////////////////////////////////////////////////
172 /// Sets next segment to @param next_segment
173
174 void RoadSegment::set_next_road_segment(RoadSegment * next_segment) {
175     m_next_segment = next_segment;
176 }
177
178 //////////////////////////////////////////////////
179 /// Calculates theta according to next_segment. Throws if m_next_segment is
180 /// nullptr

```

```

182 void RoadSegment::calculate_theta() {
183     if(m_next_segment == nullptr){
184         throw std::invalid_argument("Can't calculate theta if next segment is nullptr");
185     }
186     m_theta = atan2(m_y-m_next_segment->m_y, m_next_segment->m_x-m_x);
187 }
188
189 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
190 // Returns node of lane number n. E.g. n=0 is the right-most lane.
191
192 RoadNode* RoadSegment::get_node_pointer(int n) {
193     return m_nodes[n];
194 }
195
196 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
197 // Returns all nodes in segment.
198
199 std::vector<RoadNode*> RoadSegment::get_nodes() {
200     return m_nodes;
201 }
202
203 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
204 // Returns next segment
205
206 RoadSegment* RoadSegment::next_segment() {
207     return m_next_segment;
208 }
209
210 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
211 // Automatically populates node connections by connecting current node to
212 // all nodes in next segment.
213
214 void RoadSegment::set_all_node_pointers_to_next_segment() {
215     for(RoadNode * node: m_nodes){
216         for(int i = 0; i < m_next_segment->m_n_lanes; i++){
217             node->set_next_node(m_next_segment->get_node_pointer(i));
218         }
219     }
220 }
221
222 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
223 // Manually set connection to next segment's node. No guarantee is made
224 // on @param from_node_n and @param to_node_n. Can crash if index out of range.
225
226 void RoadSegment::set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *
227     next_segment) {
228     RoadNode * pointy = next_segment->get_node_pointer(to_node_n);
229     m_nodes[from_node_n]->set_next_node(pointy);
230 }
231
232 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
233 // Returns amount of lanes in this segment.
234
235 const int RoadSegment::get_total_amount_of_lanes() const {
236     return m_n_lanes;
237 }

```

../highway/cppfiles/roadsegment.cpp

## B.6 simulation.cpp

```

1 //
2 // Created by Carl Schiller on 2018-12-19.
3 //
4
5 #include <iostream>

```

```

#include "../headers/traffic.h"
7 #include "../headers/simulation.h"
#include <cmath>
9 #include <unistd.h>

11 //////////////////////////////////////////////////
12 /// Constructor
13 /// @param traffic : pointer reference to Traffic, this is to be able to
14 /// draw traffic outside of this class.
15 /// @param mutex : mutex thread lock from SFML.
16 /// @param sim_speed : Simulation speed multiplier, e.g. 10 would mean 10x
17 /// real time speed. If simulation can not keep up it lowers this.
18 /// @param framerate : Framerate of simulation, e.g. 60 FPS. This is the
19 /// time step of the system.
20 /// @param exit_bool : If user wants to exit this is changed outside of the class.
21
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 {
29 }
30
31 //////////////////////////////////////////////////
32 /// Runs simulation. If M_SIM_SPEED = 10, then it simulates 10x1/(M_FRAMERATE)
33 /// seconds of real time simulation.
34
35 void Simulation::update() {
36     sf::Clock clock;
37     sf::Time time;
38     double spawn_counter_0 = 0.0;
39     double spawn_counter_1 = 0.0;
40     double spawn_counter_2 = 0.0;
41     double spawn_counter_3 = 0.0;
42
43     std::vector<double *> counter;
44     counter.push_back(&spawn_counter_0);
45     counter.push_back(&spawn_counter_1);
46     counter.push_back(&spawn_counter_2);
47     counter.push_back(&spawn_counter_3);
48
49     while(!m_exit_bool){
50         m_mutex->lock();
51         //std::cout << "calculating\n";
52         for(int i = 0; i < M_SIM_SPEED; i++){
53             //std::cout << "a\n";
54             m_traffic->update(1.0 f / (float)M_FRAMERATE);
55             //std::cout << "b\n";
56             m_traffic->spawn_cars(counter, 1.0 f / (float)M_FRAMERATE);
57             //m_mutex->lock();
58             //std::cout << "c\n";
59             m_traffic->despawn_cars();
60             //m_mutex->unlock();
61             //std::cout << "d\n";
62         }
63         //std::cout << "calculated\n";
64         m_mutex->unlock();
65
66         time = clock.restart();
67         sf::Int64 acutal_elapsed = time.asMicroseconds();
68         double sim_elapsed = (1.0 f / (float)M_FRAMERATE)*1000000;
69
70         if(acutal_elapsed < sim_elapsed){
71             usleep((useconds_t)(sim_elapsed-acutal_elapsed));

```



```

73         m_traffic->m_multiplier = M_SIM_SPEED;
74     }
75     else{
76         m_traffic->m_multiplier = M_SIM_SPEED*(sim_elapsed/acutal_elapsed);
77     }
78 }
79 }

```

../highway/cppfiles/simulation.cpp

## B.7 traffic.cpp

```

1  //
2  // Created by Carl Schiller on 2018-12-19.
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]),
44     m_search_radius_to_car_in_front(args[14]),
45     m_ramp_meter_period(args[17]),
46     m_ramp_meter(bargs[1]),
47     m_multiplier(args[15])
48 {
49     probs.push_back(m_lane_0_spawn_prob);
50     probs.push_back(m_lane_1_spawn_prob);
51     probs.push_back(m_lane_2_spawn_prob);
52     probs.push_back(m_ramp_0_spawn_prob);
53
54     if (!m_font.loadFromFile("/Library/Fonts/Andale mono.ttf")){

```

```

55     }
56
57     Road::shared().ramp_meter_position->ramp_counter = 0;
58     Road::shared().ramp_meter_position->meter = m_ramp_meter;
59     Road::shared().ramp_meter_position->period = m_ramp_meter_period;
60 }
61
62 ////////////////////////////////////////////////////
63 // Copy constructor, deep copies all content.
64
65 Traffic::Traffic(const Traffic &ref) :
66     debug(ref.debug),
67     m_font(ref.m_font),
68     m_aggro(ref.m_aggro),
69     m_aggro_sigma(ref.m_aggro_sigma),
70     m_spawn_freq(ref.m_spawn_freq),
71     m_speed(ref.m_speed),
72     m_lane_0_spawn_prob(ref.m_lane_0_spawn_prob),
73     m_lane_1_spawn_prob(ref.m_lane_1_spawn_prob),
74     m_lane_2_spawn_prob(ref.m_lane_2_spawn_prob),
75     m_ramp_0_spawn_prob(ref.m_ramp_0_spawn_prob),
76     m_min_dist_to_car_in_front(ref.m_min_dist_to_car_in_front),
77     m_min_overtake_dist_trigger(ref.m_min_overtake_dist_trigger),
78     m_max_overtake_dist_trigger(ref.m_max_overtake_dist_trigger),
79     m_overtake_done_dist(ref.m_overtake_done_dist),
80     m_merge_min_dist(ref.m_merge_min_dist),
81     m_search_radius_around(ref.m_search_radius_around),
82     m_search_radius_to_car_in_front(ref.m_search_radius_to_car_in_front),
83     m_ramp_meter_period(ref.m_ramp_meter_period),
84     m_ramp_meter(ref.m_ramp_meter),
85     probs(ref.probs),
86     m_multiplier(ref.m_multiplier)
87 {
88     // clear values if there are any.
89     for(Car * delete_this : m_cars){
90         delete delete_this;
91     }
92     m_cars.clear();
93
94     // reserve place for new pointers.
95     m_cars.reserve(ref.m_cars.size());
96
97     // copy values into new pointers
98     for(Car * car : ref.m_cars){
99         Car * new_car_pointer = new Car(*car);
100         // *new_car_pointer = *car;
101         m_cars.push_back(new_car_pointer);
102     }
103
104     // values we copied are good, except the car pointers inside the car class.
105     std::map<int, Car*> overtake_this_car;
106     std::map<Car*, int> labeling;
107     for(int i = 0; i < m_cars.size(); i++){
108         overtake_this_car[i] = ref.m_cars[i]->overtake_this_car;
109         labeling[ref.m_cars[i]] = i;
110         m_cars[i]->overtake_this_car = nullptr; // clear copied pointers
111         //m_cars[i]->want_to_overtake_me.clear(); // clear copied pointers
112     }
113     std::map<int, int> from_to;
114     for(int i = 0; i < m_cars.size(); i++){
115         if(overtake_this_car[i] != nullptr){
116             from_to[i] = labeling[overtake_this_car[i]];
117         }
118     }
119
120     for(auto it : from_to){
121         m_cars[it.first]->overtake_this_car = m_cars[it.second];

```

```

123         //m_cars[it.second]->want_to_overtake_me.push_back(m_cars[it.first]);
124     }
125 }
126
127 ///////////////////////////////////////////////////
128 /// Copy-assignment constructor, deep copies all content and swaps.
129
130 Traffic& Traffic::operator=(const Traffic & rhs) {
131     Traffic tmp(rhs);
132
133     std::swap(debug,tmp.debug);
134     std::swap(m_font,tmp.m_font);
135     std::swap(m_cars,tmp.m_cars);
136     std::swap(m_multiplier,tmp.m_multiplier);
137     std::swap(probs,tmp.probs);
138
139     return *this;
140 }
141
142 ///////////////////////////////////////////////////
143 /// Destructor, deletes all cars.
144
145 Traffic::~Traffic() {
146     for(Car * & car : m_cars){
147         delete car;
148     }
149     Traffic::m_cars.clear();
150 }
151
152 ///////////////////////////////////////////////////
153 /// Returns size of car vector
154
155 unsigned long Traffic::n_of_cars(){
156     return m_cars.size();
157 }
158
159 ///////////////////////////////////////////////////
160 /// Random generator, returns reference to random generator in order to,
161 /// not make unnecessary copies.
162
163 std::mt19937& Traffic::my_engine() {
164     static std::mt19937 e(std::random_device{}());
165     return e;
166 }
167
168 ///////////////////////////////////////////////////
169 /// Logic for spawning cars by looking at how much time has elapsed.
170 /// @param spawn_counter : culmulative time elapsed
171 /// @param elapsed : time elapsed for one time step.
172 /// @param threshold : threshold is set by randomly selecting a poission
173 /// distributed number.
174 ///
175 /// Cars that are spawned are poission distributed in time, the speed of the
176 /// cars are normally distributed according to their agresiveness.
177
178 void Traffic::spawn_cars(std::vector<double*> & spawn_counter, float elapsed) {
179     int i = 0;
180     std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
181     std::vector<Car*> cars;
182     for(int j = 0; j < 4; j++){
183         cars.push_back(nullptr);
184     }
185
186     for(double * counter : spawn_counter){
187         if(*counter < 0){
188             std::gamma_distribution<double> dis(m_spawn_freq, probs[i]);
189             std::normal_distribution<float> aggro(m_aggro, m_aggro_sigma);

```

```

191         *counter = dis(my_engine());
192         float aggressiveness = aggro(my_engine());
193         float speed = m_speed*aggressiveness;
194         float target = speed;
195
196         if(i < 3){
197             Car * new_car = new Car(segments[0], i, speed, target, aggressiveness,
m_min_dist_to_car_in_front,
198             m_min_overtake_dist_trigger, m_max_overtake_dist_trigger,
199             m_overtake_done_dist,
200             m_merge_min_dist, m_search_radius_around,
m_search_radius_to_car_in_front);
201             cars[i] = new_car;
202         }
203         else{
204             Car * new_car = new Car(segments[1], 0, speed, target, aggressiveness,
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,
m_search_radius_to_car_in_front);
208             cars[i] = new_car;
209         }
210         i++;
211         *counter -= elapsed;
212     }
213
214     for(Car * car : cars) {
215         if(car != nullptr){
216             Car * closest_car_ahead = car->find_closest_car_ahead();
217
218             if(closest_car_ahead == nullptr && closest_car_ahead != car){
219                 m_cars.push_back(car);
220             }
221             else{
222                 float dist = Util::distance_to_car(car, closest_car_ahead);
223                 if(dist < 10){
224                     delete car;
225                 }
226                 else if (dist < 150){
227                     car->speed() = closest_car_ahead->speed();
228                     m_cars.push_back(car);
229                 }
230                 else{
231                     m_cars.push_back(car);
232                 }
233             }
234         }
235     }
236 }
237
238 ///////////////////////////////////////////////////
239 /// Despawn @param car
240
241 void Traffic::despawn_car(Car *& car) {
242     unsigned long size = m_cars.size();
243     for(int i = 0; i < size; i++){
244         if(car == m_cars[i]){
245             //std::cout << "found " << car << ", " << m_cars[i] << std::endl;
246             delete m_cars[i];
247             m_cars[i] = nullptr;
248             //std::cout << car << std::endl;
249             m_cars.erase(m_cars.begin()+i);
250             car = nullptr;
251             //std::cout << "deleted\n";
252             break;
253         }
254     }

```

```

253     }
254 }
255
256 //////////////////////////////////////////////////
257 /// Despawn cars that are in the despawn segment.
258
259 void Traffic::despawn_cars() {
260     //std::cout << "e\n";
261     std::map<Car *, bool> to_delete;
262     for(Car * car : m_cars){
263         for(RoadSegment * seg : Road::shared().despawn_positions()){
264             if(car->get_segment() == seg){
265
266                 to_delete[car] = true;
267                 break;
268             }
269         }
270     }
271
272     for(Car * car : m_cars){
273         for(auto it : to_delete){
274             if(it.first == car->overtake_this_car){
275                 car->overtake_this_car = nullptr;
276             }
277         }
278     }
279
280     for(Car * & car : m_cars){
281         if(to_delete[car]){
282             delete car;
283             car = nullptr;
284         }
285     }
286
287     //std::cout << "f\n";
288     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(),
289     static_cast<Car*>(nullptr));
290     m_cars.erase(new_end, m_cars.end());
291     //std::cout << "g\n";
292 }
293
294 //////////////////////////////////////////////////
295 /// Despawn all cars.
296
297 void Traffic::despawn_all_cars() {
298     for(Car * car : m_cars){
299         car->overtake_this_car = nullptr;
300     }
301
302     for(Car * & car : m_cars){
303         delete car;
304         car = nullptr;
305     }
306
307     m_cars.clear();
308 }
309
310 //////////////////////////////////////////////////
311 /// Force places a new car with user specified inputs.
312 ///
313 /// \param seg : segment of car
314 /// \param node : node of car
315 /// \param vel : (current)velocity of car
316 /// \param target : target velocity of car
317 /// \param aggro : agressiveness of car
318
319 void Traffic::force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target,
320 float aggro) {

```

```

319     Car * car = new Car(seg,node,vel,target,aggro,m_min_dist_to_car_in_front ,
                        m_min_overtake_dist_trigger ,m_max_overtake_dist_trigger ,
321                        m_merge_min_dist ,m_search_radius_around ,
                        m_search_radius_to_car_in_front);
                        m_cars.push_back(car);
323 }

325 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
326 /// Updates traffic according by stepping @param elapsed_time seconds in time.
327
328 void Traffic::update(float elapsed_time) {
329     if(m_ramp_meter){
330         float temp = Road::shared().ramp_meter_position->ramp_counter;
331         temp += elapsed_time;
332         if(temp >= m_ramp_meter_period){
333             temp -= m_ramp_meter_period;
334             Road::shared().ramp_meter_position->car_passed = false;
335         }
336         Road::shared().ramp_meter_position->ramp_counter = temp;
337     }

338     for(Car * & car : m_cars){
339         car->avoid_collision(elapsed_time);
340     }

341     for(Car * & car : m_cars){
342         car->update_pos(elapsed_time);
343     }
344 }

345
346 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
347 /// Returns vector of all cars.
348
349 std::vector<Car *> Traffic::get_car_copies() const {
350     return m_cars;
351 }

352
353 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
354 /// Returns average flow of all cars. Average value of
355 /// quotient of current speed divided by target speed for all cars.
356
357 float Traffic::get_avg_flow() {
358     float flow = 0;
359     float i = 0;
360     for(Car * car : m_cars){
361         i++;
362         flow += car->speed()/car->target_speed();
363     }
364     if(m_cars.empty()){
365         return 0;
366     }
367     else{
368         return flow/i;
369     }
370 }

371
372 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
373 /// Returns average speeds of all cars in km/h. First entry in vector
374 /// is average speed of all cars, second entry is average speed of cars in left
375 /// lane, third entry is average speed of cars in right lane.
376
377 std::vector<float> Traffic::get_avg_speeds() {
378     std::vector<float> speedy;
379     speedy.reserve(3);
380
381     float flow = 0;
382     float flow_left = 0;

```

```

385 float flow_right = 0;
386 float i = 0;
387 float j = 0;
388 float k = 0;
389 for(Car * car : m_cars){
390     i++;
391     flow += car->speed() * 3.6f;

393     if(car->current_segment->get_total_amount_of_lanes() == 2){
394         if(car->current_segment->get_lane_number(car->current_node) == 1){
395             flow_left += car->speed() * 3.6f;
396             j++;
397         }
398         else{
399             flow_right += car->speed() * 3.6f;
400             k++;
401         }
402     }
403 }
404 if(m_cars.empty()){
405     return speedy;
406 }
407 else{
408     flow = flow/i;
409     flow_left = flow_left/j;
410     flow_right = flow_right/k;
411     speedy.push_back(flow);
412     speedy.push_back(flow_left);
413     speedy.push_back(flow_right);
414     return speedy;
415 }
416 }
417
418 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
419 /// Draws cars (and nodes if debug = true) to @param target, which could
420 /// be a window. Blue cars are cars that want to overtake someone,
421 /// green cars are driving as fast as they want (target speed),
422 /// red cars are driving slower than they want.
423
424 void Traffic::draw(sf::RenderTarget &target, sf::RenderStates states) const {
425     // print debug info about node placements and stuff

427     sf::CircleShape circle;
428     circle.setRadius(4.0f);
429     circle.setOutlineColor(sf::Color::Cyan);
430     circle.setOutlineThickness(1.0f);
431     circle.setFill(sf::Color::Transparent);

433     sf::Text segment_n;
434     segment_n.setFont(m_font);
435     segment_n.setFill(sf::Color::Black);
436     segment_n.setCharacterSize(14);

437     sf::VertexArray line(sf::Lines, 2);
438     line[0].color = sf::Color::Blue;
439     line[1].color = sf::Color::Blue;

441     if(debug){
442         int i = 0;

444         for(RoadSegment * segment : Road::shared().segments()){
445             for(RoadNode * node : segment->get_nodes()){
446                 circle.setPosition(sf::Vector2f(node->get_x() * 2 - 4, node->get_y() * 2 - 4));
447                 line[0].position = sf::Vector2f(node->get_x() * 2, node->get_y() * 2);
448                 for(RoadNode * connected_node : node->get_nodes_from_me()){
449                     line[1].position = sf::Vector2f(connected_node->get_x() * 2, connected_node
->get_y() * 2);
450                     target.draw(line, states);

```

```

453         }
        target.draw(circle, states);

455     }
    segment_n.setString(std::to_string(i));
457    segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4,segment->get_y()*2+4));
    target.draw(segment_n, states);
459
    i++;
461 }
}
463 if(m_ramp_meter){
    RoadSegment * meter = Road::shared().ramp_meter_position;
465    circle.setPosition(sf::Vector2f(meter->get_x()*2+4-25,meter->get_y()*2-4));
    circle.setOutlineColor(sf::Color::Black);
467    if(meter->ramp_counter > m_ramp_meter_period*0.5f){
        circle.setFillColor(sf::Color::Green);
469
    }
471    else{
        circle.setFillColor(sf::Color::Red);
473    }
    target.draw(circle, states);
475    circle.setOutlineColor(sf::Color::Cyan);
    circle.setFillColor(sf::Color::Transparent);
477 }

479 // one rectangle is all we need :)
sf::RectangleShape rectangle;
481 rectangle.setSize(sf::Vector2f(9.4,3.4));
//rectangle.setFillColor(sf::Color::Green);
483 rectangle.setOutlineColor(sf::Color::Black);
rectangle.setOutlineThickness(2.0f);
485
487 //std::cout << "start drawing\n";
for(Car * car : m_cars){
    if(car != nullptr){
489        //std::cout << "a\n";
        rectangle.setPosition(car->x_pos()*2,car->y_pos()*2);
491        rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
        unsigned int colval = (unsigned int)std::min(255.0f*(car->speed()/car->
target_speed()),255.0f);
493        sf::Uint8 colorspeed = static_cast<sf::Uint8>(colval);
        //std::cout << "b\n";
495        if(car->overtake_this_car != nullptr){
            rectangle.setFillColor(sf::Color(255-colorspeed,0,colorspeed,255));
497        }
        else{
499            rectangle.setFillColor(sf::Color(255-colorspeed,colorspeed,0,255));
        }

501        target.draw(rectangle, states);

503        // this caused crash earlier
505        if(car->heading_to_node!=nullptr && debug){
            // print debug info about node placements and stuff
507            circle.setOutlineColor(sf::Color::Red);
            circle.setOutlineThickness(2.0f);
509            circle.setFillColor(sf::Color::Transparent);
            circle.setPosition(sf::Vector2f(car->current_node->get_x()*2-4,car->
current_node->get_y()*2-4));
511            target.draw(circle, states);
            circle.setOutlineColor(sf::Color::Green);
513            circle.setPosition(sf::Vector2f(car->heading_to_node->get_x()*2-4,car->
heading_to_node->get_y()*2-4));
            target.draw(circle, states);
515        }
    }
}

```



```

517     }
518     //std::cout << "stop drawing\n";
519 }

521 ///////////////////////////////////////////////////////////////////
522 /// Modifies @param text by inserting information about Traffic,
523 /// average speeds and frame rate among other things.

525 void Traffic::get_info(sf::Text & text, sf::Time &elapsed) {
526     //TODO: SOME BUG HERE.

527     float fps = 1.0f/elapsed.asSeconds();
528     unsigned long amount_of_cars = n_of_cars();
529     float flow = get_avg_flow();
530     std::vector<float> spe = get_avg_speeds();
531     std::string speedy = std::to_string(fps).substr(0,2) +
532         " fps, ncars: " + std::to_string(amount_of_cars) + "\n"
533         + "avg_flow: " + std::to_string(flow).substr(0,4) + "\n"
534         + "avg_speed: " + std::to_string(spe[0]).substr(0,5) + "km/h\n"
535         + "left_speed: " + std::to_string(spe[1]).substr(0,5) + "km/h\n"
536         + "right_speed: " + std::to_string(spe[2]).substr(0,5) + "km/h\n"
537         + "sim_multiplier: " + std::to_string(m_multiplier).substr(0,3) + "
538     x";
539     text.setString(speedy);
540     text.setPosition(0,0);
541     text.setFillColor(sf::Color::Black);
542     text.setFont(m_font);
543 }

```

../highway/cppfiles/traffic.cpp

## B.8 unittests.cpp

```

//
2 // Created by Carl Schiller on 2019-01-16.
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;
14
15     for(RoadSegment * seg : segments){
16         usleep(100000);
17         std::cout << "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ", "<<
18         seg << std::endl;
19         std::cout << "next segment" << seg->next_segment() << std::endl;
20         std::vector<RoadNode*> nodes = seg->get_nodes();
21         for(RoadNode * node : nodes){
22             std::vector<RoadNode*> connections = node->get_nodes_from_me();
23             std::cout << "node " << node << " has connections:" << std::endl;
24             for(RoadNode * pointy : connections){
25                 std::cout << pointy << std::endl;
26             }
27         }
28         i++;
29         m_traffic->force_place_car(seg, seg->get_nodes()[0], 1, 1, 0.01);
30         std::cout << "placed car" << std::endl;
31     }
32     std::cout << "Placement tests passed\n";
33 }

```

```

34 void Tests::delete_cars_test() {
    std::vector<Car*> car_copies = m_traffic->get_car_copies();
36
    for(Car * car : car_copies){
38         std::cout << car << std::endl;
        usleep(100);
40         m_mutex->lock();
        std::cout << "deleting car\n";
42         //usleep(100000);
        //std::cout << "Removing car " << car << std::endl;
44         m_traffic->despawn_car(car);
        m_mutex->unlock();
46         std::cout << car << std::endl;
    }
48     std::cout << "Car despawn tests passed\n";
}
50
51 void Tests::run_one_car() {
52     double ten = 10.0;
    double zero = 0;
54     //m_traffic->spawn_cars(ten,0,zero);
    double fps = 60.0;
56     double multiplier = 10.0;
    /*
58     std::cout << "running one car\n";
    while(m_traffic->n_of_cars() != 0) {
60         usleep((useconds_t)(1000000.0/(fps*multiplier)));
        m_traffic->update(1.0f/(float)fps);
62         m_traffic->despawn_cars();
    }
64     */
}
66
67 void Tests::placement_test_2() {
68     std::cout << "Starting placement tests 2\n";
    std::vector<RoadSegment*> segments = Road::shared().segments();
70     int i = 0;
72
    for(RoadSegment * seg : segments){
        usleep(100000);
74         std::cout << "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ", "<<
seg << std::endl;
        std::cout << "next segment" << seg->next_segment() << std::endl;
76         std::vector<RoadNode*> nodes = seg->get_nodes();
        for(RoadNode * node : nodes){
78             std::vector<RoadNode*> connections = node->get_nodes_from_me();
            std::cout << "node" << node << " has connections:" << std::endl;
80             for(RoadNode * pointy : connections){
                std::cout << pointy << std::endl;
82             }
            m_traffic->force_place_car(seg,node,1,1,0.1);
84             std::cout << "placed car" << std::endl;
        }
86         i++;
    }
88     m_traffic->despawn_all_cars();
90     std::cout << "Placement tests 2 passed\n";
}
92
93 void Tests::placement_test_3() {
94     std::cout << "Starting placement tests 3\n";
    std::vector<RoadSegment*> segments = Road::shared().segments();
96
    for (int i = 0; i < 10000; ++i) {
98         usleep(100);
        m_traffic->force_place_car(segments[0],segments[0]->get_nodes()[0],1,1,1);

```

```

100     }
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 }

```

../highway/cppfiles/unittests.cpp

## B.9 util.cpp

```

//
2 // Created by Carl Schiller on 2019-03-04.
//
4
5 #include "../headers/util.h"
6 #include <sstream>
7 #include <string>
8 #include <cmath>
9
10 ///////////////////////////////////////////////////////////////////
11 /// Splits @param str by @param delim, returns vector of tokens obtained.
12
13 std::vector<std::string> Util::split_string_by_delimiter(const std::string &str, const char
14     delim) {
15     std::stringstream ss(str);
16     std::string item;
17     std::vector<std::string> answer;
18     while(std::getline(ss, item, delim)){
19         answer.push_back(item);
20     }
21     return answer;
22 }
23
24 ///////////////////////////////////////////////////////////////////
25 /// Returns true if @param a is behind @param b, else false
26
27 bool Util::is_car_behind(Car * a, Car * b){
28     if(a!=b){
29         float theta_to_car_b = atan2(a->y_pos()-b->y_pos(),b->x_pos()-a->x_pos());
30         float theta_difference = get_min_angle(a->theta(),theta_to_car_b);
31         return theta_difference < M_PI*0.45;
32     }
33     else{
34         return false;
35     }
36 }

```

```

38 ///////////////////////////////////////////////////////////////////
39 /// Returns true if @param a will cross paths with @param b, else false.
40 /// NOTE: @param a MUST be behind @param b.

42 bool Util::will_car_paths_cross(Car *a, Car *b) {
43     //simulate car a driving straight ahead.
44     RoadSegment * inspecting_segment = a->get_segment();
45     //RoadNode * node_0 = a->current_node;
46     RoadNode * node_1 = a->heading_to_node;

48     //int node_0_int = inspecting_segment->get_lane_number(node_0);
49     int node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);

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         inspecting_segment = node_1->get_parent_segment();
59         //node_0_int = node_1_int;
60         //node_0 = node_1;

61         // if we are at say, 2 lanes and heading to 2 lanes, keep previous lane numbering.
62         if(inspecting_segment->get_total_amount_of_lanes() == node_1->get_nodes_from_me().
63 size()){
64             node_1 = node_1->get_nodes_from_me()[node_1_int];
65         }
66         // if we get one option, stick to it.
67         else if(node_1->get_nodes_from_me().size() == 1){
68             node_1 = node_1->get_nodes_from_me()[0];
69         }
70         // we merge from 3 to 2.
71         else if(inspecting_segment->get_total_amount_of_lanes() == 3 && inspecting_segment->
72 merge){
73             node_1 = node_1->get_nodes_from_me()[std::max(node_1_int-1,0)];
74         }

75         node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
76     }

77     return false;
78 }

79
80
81
82 }

83
84 /*

85
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 */
101
102 /*

```

```

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) {
106     return a->heading_to_node == b->current_node;
108 }
110 /*
112 float Util::distance_to_line(const float theta, const float x, const float y){
114     float x_hat, y_hat;
116     x_hat = cos(theta);
118     y_hat = -sin(theta);
120
122     float proj_x = (x*x_hat+y*y_hat)*x_hat;
124     float proj_y = (x*x_hat+y*y_hat)*y_hat;
126     float dist = sqrt(abs(pow(x-proj_x, 2.0f))+abs(pow(y-proj_y, 2.0f)));
128
130     return dist;
132 }
134 */
136 /*
138 float Util::distance_to_proj_point(const float theta, const float x, const float y){
140     float x_hat, y_hat;
142     x_hat = cos(theta);
144     y_hat = -sin(theta);
146     float proj_x = (x*x_hat+y*y_hat)*x_hat;
148     float proj_y = (x*x_hat+y*y_hat)*y_hat;
150     float dist = sqrt(abs(pow(proj_x, 2.0f))+abs(pow(proj_y, 2.0f)));
152
154     return dist;
156 }
158 */
160 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
162 /// Returns distance between @param a and @param b.
164
166 float Util::distance_to_car(Car * a, Car * b){
168     if(a == nullptr || b == nullptr){
170         throw std::invalid_argument("Can't calculate distance if cars are nullptrs");
172     }
174
176     float delta_x = a->x_pos()-b->x_pos();
178     float delta_y = b->y_pos()-a->y_pos();
180
182     return sqrt(abs(pow(delta_x, 2.0f))+abs(pow(delta_y, 2.0f)));
184 }
186 /*
188 Car * Util::find_closest_radius(std::vector<Car> &cars, const float x, const float y){
190     Car * answer = nullptr;
192
194     float score = 100000;
196     for(Car & car : cars){
198         float distance = sqrt(abs(pow(car.x_pos()-x, 2.0f))+abs(pow(car.y_pos()-y, 2.0f)));
200         if(distance < score){
202             score = distance;
204             answer = &car;
206         }
208     }
210
212     return answer;
214 }
216 */
218

```

```

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