

Final Project, SI1336

Carl Schiller, 9705266436

March 5, 2019

Abstract

Contents

1	Introduction	2
1.1	Problem formulation	2
1.2	Complex systems	2
2	Method	3
2.1	Graphs	3
2.2	Discretization	3
2.3	Graphics rendering	3
3	Result	4
4	Discussion	4
A	Header files	5
A.1	cars.h	5
A.2	road.h	6
A.3	roadnode.h	6
A.4	roadsegment.h	7
A.5	simulation.h	8
A.6	traffic.h	9
A.7	unittests.h	10
A.8	util.h	10
B	Source files	11
B.1	cars.cpp	11
B.2	main.cpp	18
B.3	road.cpp	19
B.4	roadnode.cpp	22
B.5	roadsegment.cpp	23
B.6	simulation.cpp	27
B.7	traffic.cpp	28
B.8	unittests.cpp	35
B.9	util.cpp	37



Figure 1: A typical ramp meter, image courtesy of [3]

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. [4]

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. [2]

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. [1]

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).

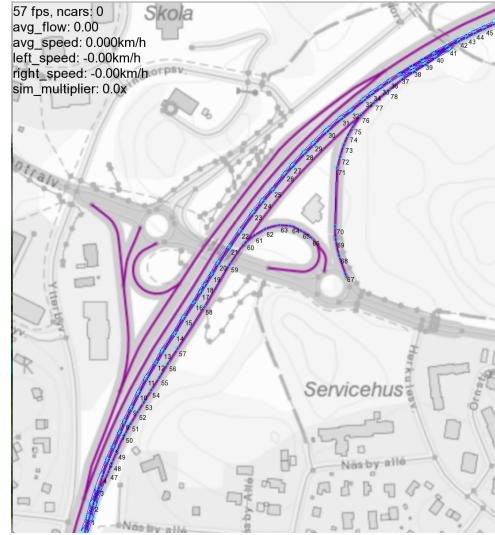


Figure 2: Setup of road with vertices and edges.

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, which has been capped at 1/60 seconds because of rendering considerations. 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.

2.3 Graphics rendering

When tweaking parameters involved in the cars' decision 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

4 Discussion

References

- [1] *Gerichteter Graph*. de. Page Version ID: 179253516. July 2018. URL: https://de.wikipedia.org/w/index.php?title=Gerichteter_Graph&oldid=179253516 (visited on 03/05/2019).
- [2] H Gould, J Tobochnik, and W Christian. “Introduction to Computer Simulation Methods”. In: (), p. 797.
- [3] Patriarca12. *English: Ramp meter on ramp from Miller Park Way to Interstate 94 east in Milwaukee, Wisconsin, USA*. July 2008. URL: https://commons.wikimedia.org/wiki/File:Ramp_meter_from_Miller_Park_Way_to_I-94_east_in_Milwaukee.jpg (visited on 03/05/2019).
- [4] 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 // Car
10 // Describes a car that moves around in Road class
11 //
12 //////////////////////////////////////
13
14 #include <map>
15 #include "roadnode.h"
16 #include "roadsegment.h"
17
18 class Car{
19 private:
20     float m_dist_to_next_node;
21     float m_speed;
22     float m_theta; // radians
23
24     float m_aggressiveness; // how fast to accelerate;
25     float m_target_speed;
26
27 public:
28     Car();
29     ~Car();
30     Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float
    aggressiveness);
31     Car(RoadSegment * spawn_point, RoadNode * lane, float vel, float target_speed, float
    aggressiveness);
32
33     // all are raw pointers
34     RoadSegment * current_segment;
35     RoadNode * current_node;
36     RoadNode * heading_to_node;
37     Car * overtake_this_car;
38
39     void update_pos(float delta_t);
40     void merge(std::vector<RoadNode*> & connections);
41     void do_we_want_to_overtake(Car * & closest_car, int & current_lane);
42     void accelerate(float delta_t);
43     void avoid_collision(float delta_t);
44     Car * find_closest_car_ahead();
45     std::map<Car *, bool> find_cars_around_car();
46
47     float x_pos();
48     float y_pos();
49
50     float & speed();
51     float & target_speed();
52     float & theta();
53
54     RoadSegment * get_segment();
55 };
56
57 #endif //HIGHWAY_CAR_H
```

../highway/headers/car.h

A.2 road.h

```
1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #ifndef HIGHWAY_ROAD_H
6 #define HIGHWAY_ROAD_H
7
8 ////////////////////////////////////////////////////////////////////
9 //
10 // Road
11 //
12 // Describes a road with interconnected nodes. Mathematically it is
13 // a graph.
14 //
15 ////////////////////////////////////////////////////////////////////
16
17 #include "roadsegment.h"
18 #include <vector>
19 #include <string>
20
21 class Road{
22 private:
23     std::vector<RoadSegment*> m_segments; // OWNERSHIP
24     std::vector<RoadSegment*> m_spawn_positions; // raw pointers
25     std::vector<RoadSegment*> m_despawn_positions; // raw pointers
26
27     const std::string M_FILENAME;
28 private:
29     Road();
30     ~Road();
31 public:
32     static Road &shared() {static Road road; return road;} // in order to only load road
33     // once in memory
34
35     Road(const Road& copy) = delete; // no copying allowed
36     Road& operator=(const Road& rhs) = delete; // no copying allowed
37
38     bool load_road();
39     std::vector<RoadSegment*> & spawn_positions();
40     std::vector<RoadSegment*> & despawn_positions();
41     std::vector<RoadSegment*> & segments();
42 };
43 #endif //HIGHWAY_ROAD_H
```

../highway/headers/road.h

A.3 roadnode.h

```
1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #ifndef HIGHWAY_ROADNODE_H
6 #define HIGHWAY_ROADNODE_H
7
8 ////////////////////////////////////////////////////////////////////
9 //
10 // RoadNode
11 //
12 // Describes the smallest element in Road, it is similar to
13 // that of a mathematical graph with nodes and edges.
14 //
15 ////////////////////////////////////////////////////////////////////
```

```

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

```

//
// Created by Carl Schiller on 2019-03-04.
//
4
#ifndef HIGHWAY.ROADSEGMENT.H
5 #define HIGHWAY.ROADSEGMENT.H
6
7 ///////////////////////////////////////////////////////////////////
8 // RoadSegment //
9 // Describes a container for several RoadNodes //
10 ///////////////////////////////////////////////////////////////////
11
12 #include <vector>
13
14 class RoadNode;
15
16 class Car;
17
18 class RoadSegment{
19 private:
20     const float m_x, m_y;
21     float m_theta;
22     const int m_n_lanes;
23
24     constexpr static float MLANE_WIDTH = 4.0f;
25
26     std::vector<RoadNode*> m_nodes; // OWNERSHIP
27     RoadSegment * m_next_segment; // raw pointer, no ownership
28 public:

```

```

34 RoadSegment() = delete;
RoadSegment(float x, float y, RoadSegment * next_segment, int lanes);
RoadSegment(float x, float y, float theta, int lanes);
36 RoadSegment(float x, float y, int lanes, bool merge);
~RoadSegment(); // rule of three
38 RoadSegment(const RoadSegment&) = delete; // rule of three
RoadSegment& operator=(const RoadSegment& rhs) = delete; // rule of three
40
bool merge;
42 std::vector<Car*> m_cars; // raw pointer, no ownership
44
RoadNode * get_node_pointer(int n);
std::vector<RoadNode *> get_nodes();
46 void append_car(Car*);
void remove_car(Car*);
48 RoadSegment * next_segment();
float get_theta();
50 const float get_x() const;
const float get_y() const;
52
int get_lane_number(RoadNode *);
54 const int get_total_amount_of_lanes() const;
void set_theta(float theta);
56 void set_next_road_segment(RoadSegment*);
void calculate_theta();
58 void calculate_and_populate_nodes();
void set_all_node_pointers_to_next_segment();
60 void set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *);
62 };
#endif //HIGHWAY.ROADSEGMENT.H

```

../highway/headers/roadsegment.h

A.5 simulation.h

```

1 //
// Created by Carl Schiller on 2018-12-19.
3 //
5 #ifndef HIGHWAY.WINDOW.H
#define HIGHWAY.WINDOW.H
7
9 ////////////////////////////////////////////////////
11 // Simulation
//
13 // Describes how to simulate Traffic class
//
15 ////////////////////////////////////////////////////
17 #include <vector>
#include "SFML/Graphics.hpp"
#include "traffic.h"
19
21 class Simulation{
private:
    sf::Mutex * m_mutex;
    Traffic * m_traffic;
    bool * m_exit_bool;
    const int M_SIM_SPEED;
    const int M_FRAMERATE;
27 public:
    Simulation() = delete;
    Simulation(Traffic *& traffic, sf::Mutex *& mutex, int sim_speed, int m_framerate, bool
    *& exitbool);
29

```



```

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  // Traffic //
10 // //
11 // Describes the whole traffic situation with Cars and a Road. //
12 // Inherits from SFML Graphics.hpp in order to render the cars. //
13 // //
14 ////////////////////////////////////////////////////////////////////
15
16 #include <random>
17 #include <vector>
18 #include "SFML/Graphics.hpp"
19 #include "car.h"
20
21 class Traffic : public sf::Drawable, public sf::Transformable{
22 private:
23     std::vector<Car*> m_cars;
24     bool debug;
25     std::mt19937 & my_engine();
26     sf::Font m_font;
27
28 public:
29     Traffic();
30     explicit Traffic(bool debug);
31     ~Traffic();
32     Traffic(const Traffic&); // rule of three
33     Traffic& operator=(const Traffic&); // rule of three
34
35     unsigned long n_of_cars();
36     void spawn_cars(double & spawn_counter, float elapsed, double & threshold);
37     void despawn_cars();
38     void despawn_all_cars();
39     void despawn_car(Car*& car);
40     void force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, float
41     aggro);
42
43     void update(float elapsed_time);
44     std::vector<Car*> get_car_copies() const;
45     float get_avg_flow();
46     std::vector<float> get_avg_speeds();
47 private:
48     virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
49 public:
50     void get_info(sf::Text & text, sf::Time & elapsed);
51     double m_multiplier;
52 };

```

```
55 #endif //HIGHWAY_TRAFFIC.H
```

../highway/headers/traffic.h

A.7 unittests.h

```
1 //
2 // Created by Carl Schiller on 2019-01-16.
3 //
4
5 #ifndef HIGHWAY_UNITTESTS_H
6 #define HIGHWAY_UNITTESTS_H
7
8 ///////////////////////////////////////////////////////////////////
9 //
10 // Tests
11 //
12 // Testing the various functions.
13 //
14 ///////////////////////////////////////////////////////////////////
15
16 #include "traffic.h"
17 #include "SFML/Graphics.hpp"
18
19 class Tests{
20 private:
21     Traffic * m_traffic;
22     sf::Mutex * m_mutex;
23     void placement_test();
24     void delete_cars_test();
25     void run_one_car();
26     void placement_test_2();
27     void placement_test_3();
28 public:
29     Tests() = delete;
30     Tests(Traffic * & traffic, sf::Mutex * & mutex);
31
32     void run_all_tests();
33 };
34
35 #endif //HIGHWAY_UNITTESTS_H
```

../highway/headers/unittests.h

A.8 util.h

```
1 //
2 // Created by Carl Schiller on 2019-03-04.
3 //
4
5 #ifndef HIGHWAY_UTIL_H
6 #define HIGHWAY_UTIL_H
7
8 ///////////////////////////////////////////////////////////////////
9 //
10 // Util
11 //
12 // Help functions for Car class.
13 //
14 ///////////////////////////////////////////////////////////////////
15
16 #include "car.h"
```

```

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
5 #include "../headers/car.h"
6 #include <map>
7 #include <cmath>
8 #include <list>
9 #include "../headers/util.h"
10
11 ///////////////////////////////////////////////////
12 /// Constructor.
13
14 Car::Car() = default;
15
16 ///////////////////////////////////////////////////
17 /// Constructor for new car with specified lane numbering in spawn point.
18 /// Lane numbering @param lane must not exceed amount of lanes in
19 /// @param spawn_point, otherwise an exception will be thrown.
20
21 Car::Car(RoadSegment *spawn_point, int lane, float vel, float target_speed, float
    aggressivness):
22     m_speed(vel),
23     m_aggressiveness(aggressivness),
24     m_target_speed(target_speed),
25     current_segment(spawn_point),
26     current_node(current_segment->get_node_pointer(lane)),
27     overtake_this_car(nullptr)
28 {
29     current_segment->append_car(this);
30
31     if(!current_node->get_nodes_from_me().empty()){
32         heading_to_node = current_node->get_next_node(lane);
33
34         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());
35
36         m_theta = current_node->get_theta(heading_to_node);
37     }
38     else{
39         throw std::invalid_argument("Car spawns in node with empty connections, or with a
            nullptr segment");
40     }
41 }
42
43 ///////////////////////////////////////////////////

```

```

44 /// Constructor for new car with specified lane. Note that
45 /// @param lane must be in @param spawn_point, otherwise no guarantee on
46 /// functionality.

48 Car::Car(RoadSegment *spawn_point, RoadNode *lane, float vel, float target_speed, float
    aggressiveness) :
    m_speed(vel),
50    m_aggressiveness(aggressiveness),
    m_target_speed(target_speed),
52    current_segment(spawn_point),
    current_node(lane),
54    overtake_this_car(nullptr)
{
56    current_segment->append_car(this);

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

60        m_dist_to_next_node = Util::distance(current_node->get_x(), heading_to_node->get_x(),
            current_node->get_y(), heading_to_node->get_y());

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

70 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
71 /// Destructor for car.
72

73 Car::~Car(){
74     if(this->current_segment != nullptr){
75         this->current_segment->remove_car(this); // remove this pointer shit
76     }

78     overtake_this_car = nullptr;
79     current_segment = nullptr;
80     heading_to_node = nullptr;
81     current_node = nullptr;
82 }

84 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
85 /// Updates position for car with time step @param delta_t.

86 void Car::update_pos(float delta_t) {
87     m_dist_to_next_node -= m_speed*delta_t;
88     // if we are at a new node.

90     if(m_dist_to_next_node < 0){
91         current_segment->remove_car(this); // remove car from this segment
92         current_segment = heading_to_node->get_parent_segment(); // set new segment
93         if(current_segment != nullptr){
94             current_segment->append_car(this); // add car to new segment
95         }
96         current_node = heading_to_node; // set new current node as previous one.

98         //TODO: place logic for choosing next node
99         std::vector<RoadNode*> connections = current_node->get_nodes_from_me();

100         if(!connections.empty()){
101             merge(connections);
102         }
103     }
104 }
105
106

```

```

108         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());
110         m.theta = current_node->get_theta(heading_to_node);
112     }
}

114 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
115 /// Function to determine if we can merge into another lane depending on.
116 /// properties of @param connections.

118 void Car::merge(std::vector<RoadNode*> & connections) {
119     // check if we merge
120     int current_lane = current_segment->get_lane_number(current_node);
121     bool can_merge = true;
122     std::map<Car*, bool> cars_around_car = find_cars_around_car();
123     Car * closest_car = find_closest_car_ahead();
124
125     for(auto it : cars_around_car){
126         float delta_dist = Util::distance_to_car(it.first, this);
127         float delta_speed = abs(speed()-it.first->speed());
128
129         if(current_lane == 0 && it.first->heading_to_node->get_parent_segment()->
get_lane_number(it.first->heading_to_node) == 1 ){
130             can_merge =
131                 delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, 15.0f);
132         }
133         else if(current_lane == 1 && it.first->heading_to_node->get_parent_segment()->
get_lane_number(it.first->heading_to_node) == 0){
134             can_merge =
135                 delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, 15.0f);
136         }
137
138         if(!can_merge){
139             break;
140         }
141     }
142
143     if(current_segment->merge){
144         if(current_lane == 0 && connections[0]->get_parent_segment()->
get_total_amount_of_lanes() != 2){
145             if(can_merge){
146                 heading_to_node = connections[1];
147             }
148             else{
149                 heading_to_node = connections[0];
150             }
151         }
152         else if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
153             current_lane = std::max(current_lane-1, 0);
154             heading_to_node = connections[current_lane];
155         }
156         else{
157             heading_to_node = connections[current_lane];
158         }
159     }
160     // if we are in start section
161     else if(current_segment->get_total_amount_of_lanes() == 3){
162         if(connections.size() == 1){
163             heading_to_node = connections[0];
164         }
165         else{
166             heading_to_node = connections[current_lane];
167         }
168     }
169     // if we are in middle section
170     else if(current_segment->get_total_amount_of_lanes() == 2){

```

```

172 // normal way
173 if (connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
174     // check if we want to overtake car in front
175     do_we_want_to_overtake(closest_car, current_lane);
176
177     // committed to overtaking
178     if (overtake_this_car != nullptr){
179         if (current_lane != 1){
180             if (can_merge){
181                 heading_to_node = connections[1];
182             }
183             else{
184                 heading_to_node = connections[current_lane];
185             }
186         }
187         else{
188             heading_to_node = connections[current_lane];
189         }
190     }
191     // merge back if overtake this car is nullptr.
192     else{
193         if (can_merge){
194             heading_to_node = connections[0];
195         }
196         else{
197             heading_to_node = connections[current_lane];
198         }
199     }
200 }
201 }
202 else{
203     heading_to_node = connections[0];
204 }
205 }
206 else if (current_segment->get_total_amount_of_lanes() == 1){
207     heading_to_node = connections[0];
208 }
209 }
210
211 //////////////////////////////////////
212 /// Helper function to determine if this car wants to overtake
213 /// @param closest_car.
214
215 void Car::do_we_want_to_overtake(Car * & closest_car, int & current_lane) {
216     //see if we want to overtake car.
217
218     if (closest_car != nullptr){
219         //float delta_speed = closest_car->speed()-speed();
220         float delta_distance = Util::distance_to_car(this, closest_car);
221
222         if (overtake_this_car == nullptr){
223             if (delta_distance > 10 && delta_distance < 40 && (target_speed()/closest_car->
224 target_speed() > m_aggressiveness*1.0f) && current_lane == 0 && closest_car->
225 current_node->get_parent_segment()->get_lane_number(closest_car->current_node) == 0){
226                 overtake_this_car = closest_car;
227             }
228         }
229     }
230
231     if (overtake_this_car != nullptr){
232         if (Util::is_car_behind(overtake_this_car, this) && (Util::distance_to_car(this,
233 overtake_this_car) > 30)){
234             overtake_this_car = nullptr;
235         }
236     }
237 }

```

```

236 //////////////////////////////////////////////////
238 /// Function to accelerate this car.

240 void Car::accelerate(float elapsed){
241     float target = m_target_speed;
242     float d_vel; // proportional control.

244     if(m_speed < target*0.75){
245         d_vel = m_aggressiveness*elapsed*2.0f;
246     }
247     else{
248         d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
249     }

250     m_speed += d_vel;
251 }

252 //////////////////////////////////////////////////
253 /// Helper function to avoid collision with another car.

254 void Car::avoid_collision(float delta_t) {
255     float min_distance = 8.0f; // for car distance.
256     float ideal = min_distance+min_distance*(m_speed/20.f);

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

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

262         if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
263             m_speed -= std::max(std::max((radius_to_car-min_distance)*0.5f,0.0f),10.0f*
264             delta_t);
265         }
266         else if(radius_to_car < min_distance){
267             m_speed -= std::max(std::max((min_distance-radius_to_car)*0.5f,0.0f),2.0f*
268             delta_t);
269         }
270         else if(delta_speed < 0 && radius_to_car < detection_distance){
271             m_speed -= std::min(
272             abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
273             m_aggressiveness * 0.15f,
274             10.0f * delta_t);
275         }
276         else {
277             accelerate(delta_t);
278         }
279     }

280     if(current_segment->merge){
281         std::map<Car*,bool> around = find_cars_around_car();
282         for(auto it : around){
283             float delta_dist = Util::distance_to_car(it.first, this);
284             delta_speed = abs(speed()-it.first->speed());

285             if(it.first->current_node->get_parent_segment()->get_lane_number(it.first->
286             current_node) == 0 && delta_dist < ideal && this->current_segment->get_lane_number(
287             current_node) == 1 && speed()/target_speed() > 0.5){
288                 if(Util::is_car_behind(it.first, this)){
289                     accelerate(delta_t);
290                 }
291                 else{
292                     m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*
293                     delta_t);
294                 }
295             }
296         }

```

```

298         else if(it.first->current_node->get_parent_segment()->get_lane_number(it.
first->current_node) == 1 && this->current_segment->get_lane_number(current_node) == 0
&& speed()/target.speed() > 0.5 && delta_dist < ideal){
300             if(Util::is_car_behind(this, it.first)){
302                 m_speed -= std::max(std::max((ideal-delta_dist)*0.5f, 0.0f), 10.0f*
delta_t);
304             }
306         }
308     }
310 }
312 else{
314     accelerate(delta_t);
316 }
318 }
320 }
322 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
323 /// Helper function to find closest car in the same lane ahead of this car.
324 /// Returns a car if found, otherwise nullptr.
326 Car* Car::find_closest_car_ahead() {
327     float search_radius = 50;
328     std::map<RoadNode*, bool> visited;
329     std::list<RoadNode*> queue;
330
331     for(RoadNode * node : (this->current_segment->get_nodes())){
332         queue.push_front(node);
333     }
334
335     Car* answer = nullptr;
336
337     float shortest_distance = 10000000;
338
339     while(!queue.empty()){
340         RoadNode * next_node = queue.back(); // get last element
341         queue.pop_back(); // remove element
342
343         if(next_node != nullptr){
344             if(!visited[next_node] && Util::distance(x_pos(), next_node->get_x(), y_pos(),
next_node->get_y()) < search_radius){
345                 visited[next_node] = true;
346
347                 for(Car * car : next_node->get_parent_segment()->m_cars){
348                     if(this != car){
349                         float radius = Util::distance_to_car(this, car);
350                         if(Util::is_car_behind(this, car) && Util::will_car_paths_cross(this,
car) && radius < shortest_distance){
351                             shortest_distance = radius;
352                             answer = car;
353                         }
354                     }
355                 }
356             }
357         }
358
359         // push in new nodes in front of list.
360         for(RoadNode * node : next_node->get_nodes_from_me()){

```



```

360         queue.push_front(node);
361     }
362 }
363 }
364 }
365 return answer;
366 }

368 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
369 /// Searches for cars around this car in a specified radius. Note that
370 /// search radius is the radius to RoadNodes, and not surrounding cars.
371 /// Returns a map of cars the function has found.
372
373 std::map<Car *,bool> Car::find_cars_around_car() {
374     const float search_radius = 40;
375     std::map<RoadNode*,bool> visited;
376     std::list<RoadNode*> queue;
377
378     for(RoadNode * node : (this->current_segment->get_nodes())){
379         queue.push_front(node);
380     }
381
382     std::map<Car *,bool> answer;
383     while(!queue.empty()){
384         RoadNode * next_node = queue.back(); // get last element
385         queue.pop_back(); // remove element
386
387         if(next_node != nullptr){
388             if(!visited[next_node] && Util::distance(x_pos(),next_node->get_x(),y_pos(),
389 next_node->get_y()) < search_radius){
390                 visited[next_node] = true;
391                 for(Car * car : next_node->get_parent_segment()->m_cars){
392                     if(this != car){
393                         answer[car] = true;
394                     }
395                 }
396                 // push in new nodes in front of list.
397                 for(RoadNode * node : next_node->get_nodes_from_me()){
398                     queue.push_front(node);
399                 }
400                 for(RoadNode * node: next_node->get_nodes_to_me()){
401                     queue.push_front(node);
402                 }
403             }
404         }
405     }
406     return answer;
407 }
408
409 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
410 /// Returns x position of car.
411
412 float Car::x_pos() {
413     float x_position;
414     if(heading_to_node != nullptr){
415         x_position = heading_to_node->get_x()-m_dist_to_next_node*cos(m_theta);
416     }
417     else{
418         x_position = current_node->get_x();
419     }
420
421     return x_position;
422 }
423
424 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
425 /// Returns y position of car.
426

```

```

428 float Car::y-pos() {
    float y_position;
    if(heading_to_node != nullptr){
430         y_position = heading_to_node->get_y()+m_dist_to_next_node*sin(m_theta);
    }
432     else{
        y_position = current_node->get_y();
434     }

    return y_position;
436 }

438 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
440 /// Returns speed of car, as reference.

442 float & Car::speed() {
    return m_speed;
444 }

446 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
448 /// Returns target speed of car as reference.

450 float & Car::target_speed() {
    return m_target_speed;
452 }

454 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
456 /// Returns theta of car, the direction of the car. Defined in radians as a
458 /// mathematician would define angles.

460 float & Car::theta() {
    return m_theta;
462 }

464 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
466 /// Returns current segment car is in.

RoadSegment* Car::get_segment() {
    return current_segment;
466 }

```

../highway/cppfiles/car.cpp

B.2 main.cpp

```

#include <iostream>
2 #include <vector>
#include "SFML/Graphics.hpp"
4 #include "../headers/simulation.h"
#include "../headers/unittests.h"
6 #include "../headers/screens.h"

8 int main() {
    std::vector<cScreen*> Screens;
10     int screen = 0;

12     sf::RenderWindow App(sf::VideoMode(550*2, 600*2), "Highway");
    App.setFramerateLimit(60);

14     screen_0 s0;
    Screens.push_back(&s0);
16     screen_1 s1;
    Screens.push_back(&s1);
18

20     while(screen >= 0){
        screen = Screens[screen]->Run(App);
    }

```

```

22     }
24     return 0;
}

```

../highway/cppfiles/main.cpp

B.3 road.cpp

```

1  //
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() {
27     for (RoadSegment * seg : m_segments) {
28         delete seg;
29     }
30     m_segments.clear();
31 }
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)

```

```

59  /// tokens[5+3*n]: to segment number.
60
61  bool Road::load_road() {
62      bool loading = true;
63      std::ifstream stream;
64      stream.open(MFILENAME);
65
66      std::vector<std::vector<std::string>> road_vector;
67      road_vector.reserve(100);
68
69      if(stream.is_open()){
70          std::string line;
71          std::vector<std::string> tokens;
72          while(std::getline(stream, line)){
73              tokens = Util::split_string_by_delimiter(line, ' ');
74              if(tokens[0] != "#"){
75                  road_vector.push_back(tokens);
76              }
77          }
78      }
79      else{
80          loading = false;
81      }
82
83      // load segments into memory.
84      for(std::vector<std::string> & vec : road_vector){
85          if(vec.size() == 5){
86              if(vec[4] == "merge"){
87                  RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]), true);
88                  m_segments.push_back(seg);
89              }
90              else{
91                  RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]), false);
92                  m_segments.push_back(seg);
93              }
94          }
95          else{
96              RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]), false);
97              m_segments.push_back(seg);
98          }
99      }
100
101
102
103      // populate nodes.
104      for (int i = 0; i < m_segments.size(); ++i) {
105          // populate nodes normally.
106          if(road_vector[i].size() == 4){
107              m_segments[i]->set_next_road_segment(m_segments[i+1]);
108              m_segments[i]->calculate_theta();
109              // calculate nodes based on theta.
110              m_segments[i]->calculate_and_populate_nodes();
111          }
112          else if(road_vector[i].size() == 5){
113              if(road_vector[i][4] == "false"){
114                  // take previous direction and populate nodes.
115                  m_segments[i]->set_theta(m_segments[i-1]->get_theta());
116                  m_segments[i]->calculate_and_populate_nodes();
117                  // but do not connect nodes to new ones.
118
119                  // make this a despawn segment
120                  m_despawn_positions.push_back(m_segments[i]);
121              }
122          }
123      }

```

```

123         else if(road_vector[i][4] == "true"){
124             m_segments[i]->set_next_road_segment(m_segments[i+1]);
125             m_segments[i]->calculate_theta();
126             // calculate nodes based on theta.
127             m_segments[i]->calculate_and_populate_nodes();
128
129             // make this a spawn segment
130             m_spawn_positions.push_back(m_segments[i]);
131         }
132         else if(road_vector[i][4] == "merge"){
133             m_segments[i]->set_next_road_segment(m_segments[i+1]);
134             m_segments[i]->calculate_theta();
135             // calculate nodes based on theta.
136             m_segments[i]->calculate_and_populate_nodes();
137         }
138     }
139
140     // else we connect one by one.
141     else{
142         // take previous direction and populate nodes.
143         m_segments[i]->set_theta(m_segments[i-1]->get_theta());
144         // calculate nodes based on theta.
145         m_segments[i]->calculate_and_populate_nodes();
146     }
147 }
148
149 // connect nodes.
150 for (int i = 0; i < m_segments.size(); ++i) {
151     // do normal connection, ie connect all nodes.
152     if(road_vector[i].size() == 4){
153         m_segments[i]->set_all_node_pointers_to_next_segment();
154     }
155     else if(road_vector[i].size() == 5){
156         if(road_vector[i][4] == "false"){
157             // but do not connect nodes to new ones.
158         }
159         else if(road_vector[i][4] == "true"){
160             m_segments[i]->set_all_node_pointers_to_next_segment();
161         }
162         else if(road_vector[i][4] == "merge"){
163             m_segments[i]->set_all_node_pointers_to_next_segment();
164         }
165     }
166
167     // else we connect one by one.
168     else{
169         // manually connect nodes.
170         int amount_of_pointers = (int)road_vector[i].size()-4;
171         for(int j = 0; j < amount_of_pointers/3; j++){
172             int current_pos = 4+j*3;
173             RoadSegment * next_segment = m_segments[std::stoi(road_vector[i][current_pos
174 +2])];
175             m_segments[i]->set_node_pointer_to_node(std::stoi(road_vector[i][current_pos
176 ]),std::stoi(road_vector[i][current_pos+1]),next_segment);
177         }
178     }
179 }
180 return loading;
181
182 ////////////////////////////////////////////////////
183 // Returns spawn positions of Road
184
185 std::vector<RoadSegment*>& Road::spawn_positions() {
186     return m_spawn_positions;
187 }
188
189 ////////////////////////////////////////////////////

```

```

189 /// Returns despawn positions of Road
191 std::vector<RoadSegment*>& Road::despawn_positions() {
193     return m_despawn_positions;
195 }
195 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
197 /// Returns all segments of Road.
199 std::vector<RoadSegment*>& Road::segments() {
201     return m_segments;
203 }

```

../highway/cppfiles/road.cpp

B.4 roadnode.cpp

```

2 ///
3 /// Created by Carl Schiller on 2019-03-04.
4 ///
5
6 #include "../headers/roadnode.h"
7 #include <cmath>
8
9 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
10 /// Constructor
11
12 RoadNode::RoadNode() = default;
13
14 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
15 /// Destructor
16
17 RoadNode::~RoadNode() = default;
18
19 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
20 /// Constructor, @param x is x position of node, @param y is y position of node,
21 /// @param segment is to which segment this RoadNode belongs.
22
23 RoadNode::RoadNode(float x, float y, RoadSegment * segment) {
24     m_x = x;
25     m_y = y;
26     m_is_child_of = segment;
27 }
28
29 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
30 /// Appends a new RoadNode to the list connections from this RoadNode.
31 /// I.e. to where a Car is allowed to drive.
32
33 void RoadNode::set_next_node(RoadNode * next_node) {
34     m_nodes_from_me.push_back(next_node);
35     next_node->m_nodes_to_me.push_back(this); // sets double linked chain.
36 }
37
38 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
39 /// Appends a new RoadNode to the list connections to this RoadNode.
40 /// I.e. from where a Car is allowed to drive to this Node.
41
42 void RoadNode::set_previous_node(RoadNode * prev_node) {
43     m_nodes_to_me.push_back(prev_node);
44 }
45
46 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
47 /// Returns RoadSegment to which this RoadNode belongs.
48
49 RoadSegment* RoadNode::get_parent_segment() {
50     return m_is_child_of;
51 }

```

```

50 }

52 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
53 /// Returns connections from this RoadNode.
54
55 std::vector<RoadNode*> & RoadNode::get_nodes_from_me() {
56     return m_nodes_from_me;
57 }
58
59 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
60 /// Returns connections to this RoadNode.
61
62 std::vector<RoadNode*>& RoadNode::get_nodes_to_me() {
63     return m_nodes_to_me;
64 }
65
66 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
67 /// Returns x position of RoadNode.
68
69 float RoadNode::get_x() {
70     return m_x;
71 }
72
73 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
74 /// Returns y position of RoadNode.
75
76 float RoadNode::get_y() {
77     return m_y;
78 }
79
80 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
81 /// Returns angle of this RoadNode to @param node as a mathematician
82 /// would define angles. In radians.
83
84 float RoadNode::get_theta(RoadNode* node) {
85     for(RoadNode * road_node : m_nodes_from_me){
86         if(node == road_node){
87             return atan2(m_y-node->m_y,node->m_x-m_x);
88         }
89     }
90     throw std::invalid_argument("Node given is not a connecting node");
91 }
92
93 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
94 /// Returns RoadNode according to @param lane from the vector of node
95 /// connections from this RoadNode.
96
97 RoadNode* RoadNode::get_next_node(int lane) {
98     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
5 #include "../headers/roadsegment.h"
6 #include "../headers/roadnode.h"
7 #include <cmath>
8
9 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
10 /// RoadSegment destructor, removes all RodeNode element children because of
11 /// ownership.

```

```

12 RoadSegment::~~RoadSegment(){
13     for(RoadNode * elem : m_nodes){
14         delete elem;
15     }
16     m_nodes.clear();
17 }
18
19
20 ///////////////////////////////////////////////////////////////////
21 /// Constructor, creates a new segment with next connecting segment as
22 /// @param next_segment
23
24 RoadSegment::RoadSegment(float x, float y, RoadSegment * next_segment, int lanes):
25     m_x(x),
26     m_y(y),
27     m_n_lanes(lanes),
28     m_next_segment(next_segment)
29 {
30     m_theta = atan2(m_y-m_next_segment->m_y, m_next_segment->m_x-m_x);
31
32     m_nodes.reserve(m_n_lanes);
33
34     calculate_and_populate_nodes(); // populates segment with RoadNodes.
35 }
36
37 ///////////////////////////////////////////////////////////////////
38 /// Constructor, creates a new segment with manually entered @param theta.
39
40 RoadSegment::RoadSegment(float x, float y, float theta, int lanes) :
41     m_x(x),
42     m_y(y),
43     m_theta(theta),
44     m_n_lanes(lanes),
45     m_next_segment(nullptr)
46 {
47     m_nodes.reserve(m_n_lanes);
48
49     calculate_and_populate_nodes(); // populates segment with RoadNodes.
50 }
51
52 ///////////////////////////////////////////////////////////////////
53 /// Constructor, creates a new segment without creating RoadNodes. This
54 /// needs to be done manually with functions below.
55
56 RoadSegment::RoadSegment(float x, float y, int lanes, bool mer):
57     m_x(x),
58     m_y(y),
59     m_n_lanes(lanes),
60     m_next_segment(nullptr),
61     merge(mer)
62 {
63     m_nodes.reserve(m_n_lanes);
64
65     // can't set nodes if we don't have a theta.
66 }
67
68 ///////////////////////////////////////////////////////////////////
69 /// Returns theta (angle) of RoadSegment, in which direction the segment points
70
71 float RoadSegment::get_theta() {
72     return m_theta;
73 }
74
75 ///////////////////////////////////////////////////////////////////
76 /// Returns x position of RoadSegment.
77
78 const float RoadSegment::get_x() const{
79     return m_x;
80 }

```



```

80 }

82 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
83 /// Returns y position of RoadSegment.
84
85 const float RoadSegment::get_y() const {
86     return m_y;
87 }
88
89 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
90 /// Returns int number of @param node. E.g. 0 would be the right-most lane.
91 /// Throws exception if we do not find the node in this segment.
92
93 int RoadSegment::get_lane_number(RoadNode * node) {
94     for(int i = 0; i < m_n_lanes; i++){
95         if(node == m_nodes[i]){
96             return i;
97         }
98     }
99     throw std::invalid_argument("Node is not in this segment");
100 }
101
102 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
103 /// Adds a new car to the segment.
104
105 void RoadSegment::append_car(Car * car) {
106     m_cars.push_back(car);
107 }
108
109 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
110 /// Removes car from segment, if car is not in list we do nothing.
111
112 void RoadSegment::remove_car(Car * car) {
113     unsigned long size = m_cars.size();
114     bool found = false;
115     for(int i = 0; i < size; i++){
116         if(car == m_cars[i]){
117             m_cars[i] = nullptr;
118             found = true;
119         }
120     }
121     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(),
122     static_cast<Car*>(nullptr));
123     m_cars.erase(new_end, m_cars.end());
124
125     /*
126     if(!found){
127         throw std::invalid_argument("Car is not in this segment.");
128     }
129     */
130 }
131
132 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
133 /// Sets theta of RoadSegment according to @param theta.
134
135 void RoadSegment::set_theta(float theta) {
136     m_theta = theta;
137 }
138
139 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
140 /// Automatically populates segment with nodes according to amount of lanes
141 /// specified and theta specified.
142
143 void RoadSegment::calculate_and_populate_nodes() {
144     // calculates placement of nodes.
145     float total_length = MLANE_WIDTH*(m_n_lanes-1);
146     float current_length = -total_length/2.0f;

```

```

148     for(int i = 0; i < m_n_lanes; i++){
149         float x_pos = m_x+current_length*cos(m_theta+(float)M_PI*0.5f);
150         float y_pos = m_y-current_length*sin(m_theta+(float)M_PI*0.5f);
151         m_nodes.push_back(new RoadNode(x_pos,y_pos,this));
152         current_length += MLANE.WIDTH;
153     }
154 }
155
156 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
157 /// Sets next segment to @param next_segment
158 void RoadSegment::set_next_road_segment(RoadSegment * next_segment) {
159     m_next_segment = next_segment;
160 }
161
162 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
163 /// Calculates theta according to next_segment. Throws if m_next_segment is
164 /// nullptr
165 void RoadSegment::calculate_theta() {
166     if(m_next_segment == nullptr){
167         throw std::invalid_argument("Can't calculate theta if next segment is nullptr");
168     }
169     m_theta = atan2(m_y-m_next_segment->m_y,m_next_segment->m_x-m_x);
170 }
171
172 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
173 /// Returns node of lane number n. E.g. n=0 is the right-most lane.
174 RoadNode* RoadSegment::get_node_pointer(int n) {
175     return m_nodes[n];
176 }
177
178 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
179 /// Returns all nodes in segment.
180 std::vector<RoadNode*> RoadSegment::get_nodes() {
181     return m_nodes;
182 }
183
184 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
185 /// Returns next segment
186 RoadSegment* RoadSegment::next_segment() {
187     return m_next_segment;
188 }
189
190 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
191 /// Automatically populates node connections by connecting current node to
192 /// all nodes in next segment.
193 void RoadSegment::set_all_node_pointers_to_next_segment() {
194     for(RoadNode * node: m_nodes){
195         for(int i = 0; i < m_next_segment->m_n_lanes; i++){
196             node->set_next_node(m_next_segment->get_node_pointer(i));
197         }
198     }
199 }
200
201 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
202 /// Manually set connection to next segment's node. No guarantee is made
203 /// on @param from_node_n and @param to_node_n. Can crash if index out of range.
204 void RoadSegment::set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *
205     next_segment) {
206     RoadNode * pointy = next_segment->get_node_pointer(to_node_n);
207     m_nodes[from_node_n]->set_next_node(pointy);
208 }

```

```

214 //////////////////////////////////////////////////
216 /// Returns amount of lanes in this segment.
218 const int RoadSegment::get_total_amount_of_lanes() const {
220     return m_n_lanes;
}

```

../highway/cppfiles/roadsegment.cpp

B.6 simulation.cpp

```

//
2 /// Created by Carl Schiller on 2018-12-19.
//
4
6 #include <iostream>
6 #include "../headers/traffic.h"
8 #include "../headers/simulation.h"
8 #include <cmath>
8 #include <unistd.h>
10
12 //////////////////////////////////////////////////
12 /// Constructor
14 /// @param traffic : pointer reference to Traffic, this is to be able to
14 /// draw traffic outside of this class.
16 /// @param mutex : mutex thread lock from SFML.
16 /// @param sim_speed : Simulation speed multiplier, e.g. 10 would mean 10x
18 /// real time speed. If simulation can not keep up it lowers this.
18 /// @param framerate : Framerate of simulation, e.g. 60 FPS. This is the
20 /// time step of the system.
20 /// @param exit_bool : If user wants to exit this is changed outside of the class.
22 Simulation::Simulation(Traffic *&traffic, sf::Mutex *&mutex, int sim_speed, int framerate,
24 bool *& exit_bool):
24     m_mutex(mutex),
24     m_traffic(traffic),
26     m_exit_bool(exit_bool),
26     M_SIM_SPEED(sim_speed),
26     M_FRAMERATE(framerate)
28 {
30 }
32
32 //////////////////////////////////////////////////
34 /// Runs simulation. If M_SIM_SPEED = 10, then it simulates 10x1/(M_FRAMERATE)
34 /// seconds of real time simulation.
36 void Simulation::update() {
38     sf::Clock clock;
38     sf::Time time;
40     double spawn_counter = 0.0;
40     double threshold = 0.0;
42
42     while(!*m_exit_bool){
44         m_mutex->lock();
44         //std::cout << "calculating\n";
46         for(int i = 0; i < M_SIM_SPEED; i++){
46             //std::cout << "a\n";
48             m_traffic->update(1.0 f / (float)M_FRAMERATE);
48             //std::cout << "b\n";
50             m_traffic->spawn_cars(spawn_counter, 1.0 f / (float)M_FRAMERATE, threshold);
50             //m_mutex->lock();
50             //std::cout << "c\n";
52             m_traffic->despawn_cars();
52             //m_mutex->unlock();
}

```

```

54         //std::cout<< "d\n";
55     }
56     //std::cout << "calculated\n";
57     m_mutex->unlock();
58
59     time = clock.restart();
60     sf::Int64 acutal_elapsed = time.asMicroseconds();
61     double sim_elapsed = (1.0f/(float)MFRAMERATE)*1000000;
62
63     if(acutal_elapsed < sim_elapsed){
64         usleep((useconds_t)(sim_elapsed-acutal_elapsed));
65         m_traffic->m_multiplier = MSIM_SPEED;
66     }
67     else{
68         m_traffic->m_multiplier = MSIM_SPEED*(sim_elapsed/acutal_elapsed);
69     }
70 }

```

../highway/cppfiles/simulation.cpp

B.7 traffic.cpp

```

1  //
2  // Created by Carl Schiller on 2018-12-19.
3  //
4
5  #include "../headers/traffic.h"
6  #include "../headers/car.h"
7  #include "../headers/road.h"
8  #include "../headers/util.h"
9
10 ///////////////////////////////////////////////////////////////////
11 /// Constructor.
12
13 Traffic::Traffic() {
14     debug = false;
15     if(!m_font.loadFromFile("/Library/Fonts/Arial.ttf")){
16         //crash
17     }
18 }
19
20 ///////////////////////////////////////////////////////////////////
21 /// Constructor with debug bool, if we want to use debugging information.
22
23 Traffic::Traffic(bool debug) : debug(debug){
24     if(!m_font.loadFromFile("/Library/Fonts/Arial.ttf")){
25         //crash
26     }
27 }
28
29 ///////////////////////////////////////////////////////////////////
30 /// Copy constructor, deep copies all content.
31
32 Traffic::Traffic(const Traffic &ref) :
33     debug(ref.debug),
34     m_multiplier(ref.m_multiplier)
35 {
36     // clear values if there are any.
37     for(Car * delete_this : m_cars){
38         delete delete_this;
39     }
40     m_cars.clear();
41
42     // reserve place for new pointers.
43     m_cars.reserve(ref.m_cars.size());

```

```

45 // copy values into new pointers
46 for (Car * car : ref.m_cars){
47     auto new_car_pointer = new Car;
48     *new_car_pointer = *car;
49     m_cars.push_back(new_car_pointer);
50 }
51
52 // values we copied are good, except the car pointers inside the car class.
53 std::map<int, Car*> overtake_this_car;
54 std::map<Car*, int> labeling;
55 for (int i = 0; i < m_cars.size(); i++){
56     overtake_this_car[i] = ref.m_cars[i]->overtake_this_car;
57     labeling[ref.m_cars[i]] = i;
58     m_cars[i]->overtake_this_car = nullptr; // clear copied pointers
59     //m_cars[i]->want_to_overtake_me.clear(); // clear copied pointers
60 }
61 std::map<int, int> from_to;
62 for (int i = 0; i < m_cars.size(); i++){
63     if (overtake_this_car[i] != nullptr){
64         from_to[i] = labeling[overtake_this_car[i]];
65     }
66 }
67
68 for (auto it : from_to){
69     m_cars[it.first]->overtake_this_car = m_cars[it.second];
70     //m_cars[it.second]->want_to_overtake_me.push_back(m_cars[it.first]);
71 }
72 }
73
74 ///////////////////////////////////////////////////
75 /// Copy-assignment constructor, deep copies all content and swaps.
76
77 Traffic& Traffic::operator=(const Traffic & rhs) {
78     Traffic tmp(rhs);
79
80     std::swap(m_cars, tmp.m_cars);
81     std::swap(m_multiplier, tmp.m_multiplier);
82     std::swap(debug, tmp.debug);
83
84     return *this;
85 }
86
87 ///////////////////////////////////////////////////
88 /// Destructor, deletes all cars.
89
90 Traffic::~Traffic() {
91     for (Car * & car : m_cars){
92         delete car;
93     }
94     Traffic::m_cars.clear();
95 }
96
97 ///////////////////////////////////////////////////
98 /// Returns size of car vector
99
100 unsigned long Traffic::n_of_cars(){
101     return m_cars.size();
102 }
103
104 ///////////////////////////////////////////////////
105 /// Random generator, returns reference to random generator in order to,
106 /// not make unnecessary copies.
107
108 std::mt19937& Traffic::my_engine() {
109     static std::mt19937 e(std::random_device{}());
110     return e;
111 }

```

```

113 //////////////////////////////////////////////////
114 /// Logic for spawning cars by looking at how much time has elapsed.
115 /// @param spawn_counter : culmulative time elapsed
116 /// @param elapsed : time elapsed for one time step.
117 /// @param threshold : threshold is set by randomly selecting a poission
118 /// distributed number.
119 ///
120 /// Cars that are spawned are poission distributed in time, the speed of the
121 /// cars are normally distributed according to their aggressiveness.
122
123 void Traffic::spawn_cars(double & spawn_counter, float elapsed, double & threshold) {
124     spawn_counter += elapsed;
125     if(spawn_counter > threshold){
126         std::exponential_distribution<double> dis(5);
127         std::normal_distribution<float> aggro(1.0f,0.2f);
128         float sp = 30.0f;
129         std::uniform_real_distribution<float> lane(0.0f,1.0f);
130         std::uniform_real_distribution<float> spawn(0.0f,1.0f);
131
132         threshold = dis(my_engine());
133         float aggressiveness = aggro(my_engine());
134         float speed = sp*aggressiveness;
135         float target = speed;
136
137         spawn_counter = 0;
138         float start_lane = lane(my_engine());
139         float spawn_pos = spawn(my_engine());
140
141         std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
142         RoadSegment * seg;
143         Car * new_car;
144         if(spawn_pos < 0.95){
145             seg = segments[0];
146             if(start_lane < 0.457){
147                 new_car = new Car(seg,2,speed,target,aggressiveness);
148             }
149             else if(start_lane < 0.95){
150                 new_car = new Car(seg,1,speed,target,aggressiveness);
151             }
152             else{
153                 new_car = new Car(seg,0,speed,target,aggressiveness);
154             }
155         }
156         else{
157             seg = segments[1];
158             new_car = new Car(seg,0,speed,target,aggressiveness);
159         }
160
161         Car * closest_car_ahead = new_car->find_closest_car_ahead();
162
163         if(closest_car_ahead == nullptr && closest_car_ahead != new_car){
164             m_cars.push_back(new_car);
165         }
166         else{
167             float dist = Util::distance_to_car(new_car,closest_car_ahead);
168             if(dist < 10){
169                 delete new_car;
170             }
171             else if (dist < 150){
172                 new_car->speed() = closest_car_ahead->speed();
173                 m_cars.push_back(new_car);
174             }
175             else{
176                 m_cars.push_back(new_car);
177             }
178         }
179     }

```

```

181 }
182 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
183 /// Despawn @param car
184
185 void Traffic::despawn_car(Car *& car) {
186     unsigned long size = m_cars.size();
187     for(int i = 0; i < size; i++){
188         if(car == m_cars[i]){
189             //std::cout << "found " << car << ", " << m_cars[i] << std::endl;
190             delete m_cars[i];
191             m_cars[i] = nullptr;
192             //std::cout << car << std::endl;
193             m_cars.erase(m_cars.begin()+i);
194             car = nullptr;
195             //std::cout << "deleted\n";
196             break;
197         }
198     }
199 }
200
201 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
202 /// Despawn cars that are in the despawn segment.
203
204 void Traffic::despawn_cars() {
205     //std::cout << "e\n";
206     std::map<Car *, bool> to_delete;
207     for(Car * car : m_cars){
208         for(RoadSegment * seg : Road::shared().despawn_positions()){
209             if(car->get_segment() == seg){
210
211                 to_delete[car] = true;
212                 break;
213             }
214         }
215     }
216
217     for(Car * car : m_cars){
218         for(auto it : to_delete){
219             if(it.first == car->overtake_this_car){
220                 car->overtake_this_car = nullptr;
221             }
222         }
223     }
224
225     for(Car * & car : m_cars){
226         if(to_delete[car]){
227             delete car;
228             car = nullptr;
229         }
230     }
231
232     //std::cout << "f\n";
233     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(),
234         static_cast<Car*>(nullptr));
235     m_cars.erase(new_end, m_cars.end());
236     //std::cout << "g\n";
237 }
238
239 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
240 /// Despawn all cars (by creating a new traffic object).
241
242 void Traffic::despawn_all_cars() {
243     *this = Traffic();
244 }
245
246 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
247 /// Force places a new car with user specified inputs.

```

```

247 ///
248 /// \param seg : segment of car
249 /// \param node : node of car
250 /// \param vel : (current)velocity of car
251 /// \param target : target velocity of car
252 /// \param aggro : agressiveness of car
253
254 void Traffic::force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target,
255     float aggro) {
256     Car * car = new Car(seg, node, vel, target, aggro);
257     m_cars.push_back(car);
258 }
259
260 ///////////////////////////////////////////////////
261 /// Updates traffic according by stepping @param elapsed_time seconds in time.
262 ///////////////////////////////////////////////////
263 void Traffic::update(float elapsed_time) {
264     for(Car * & car : m_cars){
265         car->avoid_collision(elapsed_time);
266     }
267     for(Car * & car : m_cars){
268         car->update_pos(elapsed_time);
269     }
270 }
271
272 ///////////////////////////////////////////////////
273 /// Returns vector of all cars.
274 ///////////////////////////////////////////////////
275 std::vector<Car *> Traffic::get_car_copies() const {
276     return m_cars;
277 }
278
279 ///////////////////////////////////////////////////
280 /// Returns average flow of all cars. Average value of
281 /// quotient of current speed divided by target speed for all cars.
282 ///////////////////////////////////////////////////
283 float Traffic::get_avg_flow() {
284     float flow = 0;
285     float i = 0;
286     for(Car * car : m_cars){
287         i++;
288         flow += car->speed()/car->target_speed();
289     }
290     if(m_cars.empty()){
291         return 0;
292     }
293     else{
294         return flow/i;
295     }
296 }
297
298 ///////////////////////////////////////////////////
299 /// Returns average speeds of all cars in km/h. First entry in vector
300 /// is average speed of all cars, second entry is average speed of cars in left
301 /// lane, third entry is average speed of cars in right lane.
302 ///////////////////////////////////////////////////
303 std::vector<float> Traffic::get_avg_speeds() {
304     std::vector<float> speedy;
305     speedy.reserve(3);
306
307     float flow = 0;
308     float flow_left = 0;
309     float flow_right = 0;
310     float i = 0;
311     float j = 0;
312     float k = 0;
313     for(Car * car : m_cars){

```



```

315     i++;
    flow += car->speed() * 3.6 f;

317     if (car->current_segment->get_total_amount_of_lanes() == 2){
319         if (car->current_segment->get_lane_number(car->current_node) == 1){
            flow_left += car->speed() * 3.6 f;
            j++;
321         }
        else{
323             flow_right += car->speed() * 3.6 f;
            k++;
325         }
    }
327 }
    if (m_cars.empty()){
329         return speedy;
    }
331     else{
        flow = flow / i;
333         flow_left = flow_left / j;
        flow_right = flow_right / k;
335         speedy.push_back(flow);
        speedy.push_back(flow_left);
337         speedy.push_back(flow_right);
        return speedy;
339     }
}

341 //////////////////////////////////////
343 /// Draws cars (and nodes if debug = true) to @param target, which could
/// be a window. Blue cars are cars that want to overtake someone,
345 /// green cars are driving as fast as they want (target speed),
/// red cars are driving slower than they want.

347 void Traffic::draw(sf::RenderTarget &target, sf::RenderStates states) const {
349     // print debug info about node placements and stuff

351     sf::CircleShape circle;
    circle.setRadius(4.0 f);
353     circle.setOutlineColor(sf::Color::Cyan);
    circle.setOutlineThickness(1.0 f);
355     circle.setFillColor(sf::Color::Transparent);

357     sf::Text segment_n;
    segment_n.setFont(m_font);
359     segment_n.setFill-color(sf::Color::Black);
    segment_n.setCharacterSize(14);
361

    sf::VertexArray line(sf::Lines, 2);
363     line[0].color = sf::Color::Blue;
    line[1].color = sf::Color::Blue;
365

    if (debug){
367         int i = 0;

369         for (RoadSegment * segment : Road::shared().segments()){
            for (RoadNode * node : segment->get_nodes()){
371                 circle.setPosition(sf::Vector2f(node->get_x() * 2 - 4, node->get_y() * 2 - 4));
                line[0].position = sf::Vector2f(node->get_x() * 2, node->get_y() * 2);
373                 for (RoadNode * connected_node : node->get_nodes_from_me()){
                    line[1].position = sf::Vector2f(connected_node->get_x() * 2, connected_node
->get_y() * 2);
375                     target.draw(line, states);
                }
377                 target.draw(circle, states);
379             }
        }
    }
}

```

```

381         segment_n.setString(std::to_string(i));
382         segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4,segment->get_y()*2+4));
383         target.draw(segment_n, states);
384         i++;
385     }
386 }
387
388 // one rectangle is all we need :)
389 sf::RectangleShape rectangle;
390 rectangle.setSize(sf::Vector2f(9.4,3.4));
391 //rectangle.setFill(sf::Color::Green);
392 rectangle.setOutlineColor(sf::Color::Black);
393 rectangle.setOutlineThickness(2.0f);
394
395 //std::cout << "start drawing\n";
396 for(Car * car : m_cars){
397     //std::cout << "drawing" << car << std::endl;
398     if(car != nullptr){
399         rectangle.setPosition(car->x_pos()*2,car->y_pos()*2);
400         rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
401         unsigned int colval = (unsigned int)std::min(255.0f*(car->speed()/car->
target_speed()),255.0f);
402         sf::Uint8 colorspeed = static_cast<sf::Uint8>(colval);
403
404         if(car->overtake_this_car != nullptr){
405             rectangle.setFill(sf::Color(255-colorspeed,0,colorspeed,255));
406         }
407         else{
408             rectangle.setFill(sf::Color(255-colorspeed,colorspeed,0,255));
409         }
410
411         target.draw(rectangle, states);
412
413         // this caused crash earlier
414         if(car->heading_to_node!=nullptr && debug){
415             // print debug info about node placements and stuff
416             circle.setOutlineColor(sf::Color::Red);
417             circle.setOutlineThickness(2.0f);
418             circle.setFill(sf::Color::Transparent);
419             circle.setPosition(sf::Vector2f(car->current_node->get_x()*2-4,car->
current_node->get_y()*2-4));
420             target.draw(circle, states);
421             circle.setOutlineColor(sf::Color::Green);
422             circle.setPosition(sf::Vector2f(car->heading_to_node->get_x()*2-4,car->
heading_to_node->get_y()*2-4));
423             target.draw(circle, states);
424         }
425     }
426 }
427 }
428
429 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
430 /// Modifies @param text by inserting information about Traffic ,
431 /// average speeds and frame rate among other things.
432
433 void Traffic::get_info(sf::Text & text ,sf::Time &elapsed) {
434     //TODO: SOME BUG HERE.
435
436     float fps = 1.0f/elapsed.asSeconds();
437     unsigned long amount_of_cars = n_of_cars();
438     float flow = get_avg_flow();
439     std::vector<float> spe = get_avg_speeds();
440     std::string speedy = std::to_string(fps).substr(0,2) +
441         " fps, ncars: " + std::to_string(amount_of_cars) + "\n"
442         + "avg-flow: " + std::to_string(flow).substr(0,4) + "\n"
443         + "avg-speed: " + std::to_string(spe[0]).substr(0,5) + "km/h\n"
444         + "left-speed: " + std::to_string(spe[1]).substr(0,5) + "km/h\n"
445         + "right-speed: " + std::to_string(spe[2]).substr(0,5) + "km/h\n"

```

```

+ "sim.multiplier: " + std::to_string(m_multiplier).substr(0,3) + "
x";
447 text.setString(speedy);
text.setPosition(0,0);
449 text.setFillColor(sf::Color::Black);
text.setFont(m_font);
451 }

```

../highway/cppfiles/traffic.cpp

B.8 unittests.cpp

```

//
2 // Created by Carl Schiller on 2019-01-16.
//
4
#include "unittests.h"
6 #include "road.h"
#include <unistd.h>
8 #include <iostream>

10 void Tests::placement_test() {
std::cout << "Starting placement tests\n";
12 std::vector<RoadSegment*> segments = Road::shared().segments();
int i = 0;
14
for(RoadSegment * seg : segments){
16     usleep(100000);
std::cout << "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ", "<<
seg << std::endl;
18     std::cout << "next segment" << seg->next_segment() << std::endl;
std::vector<RoadNode*> nodes = seg->get_nodes();
20     for(RoadNode * node : nodes){
std::vector<RoadNode*> connections = node->get_nodes_from_me();
22     std::cout << "node" << node << " has connections:" << std::endl;
for(RoadNode * pointy : connections){
24         std::cout << pointy << std::endl;
}
26     }
i++;
28     m_traffic->force_place_car(seg, seg->get_nodes()[0], 1, 1, 0.01);
std::cout << "placed car" << std::endl;
30 }
std::cout << "Placement tests passed\n";
32 }

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
52 void Tests::run_one_car() {
double ten = 10.0;
double zero = 0;

```

```

54     m_traffic->spawn_cars(ten,0,zero);
55     double fps = 60.0;
56     double multiplier = 10.0;

58     std::cout << "running one car\n";
59     while(m_traffic->n_of_cars() != 0) {
60         usleep((useconds_t)(1000000.0/(fps*multiplier)));
61         m_traffic->update(1.0f/(float)fps);
62         m_traffic->despawn_cars();
63     }
64 }

66 void Tests::placement_test_2() {
67     std::cout << "Starting placement tests 2\n";
68     std::vector<RoadSegment*> segments = Road::shared().segments();
69     int i = 0;

70     for(RoadSegment * seg : segments){
71         usleep(100000);
72         std::cout<< "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ", "<<
73         seg << std::endl;
74         std::cout << "next segment" << seg->next_segment() << std::endl;
75         std::vector<RoadNode*> nodes = seg->get_nodes();
76         for(RoadNode * node : nodes){
77             std::vector<RoadNode*> connections = node->get_nodes_from_me();
78             std::cout << "node" << node << " has connections:" << std::endl;
79             for(RoadNode * pointy : connections){
80                 std::cout << pointy << std::endl;
81             }
82             m_traffic->force_place_car(seg,node,1,1,0.1);
83             std::cout << "placed car" << std::endl;
84         }
85         i++;
86     }

87     m_traffic->despawn_all_cars();
88     std::cout << "Placement tests 2 passed\n";
89 }

91 void Tests::placement_test_3() {
92     std::cout << "Starting placement tests 3\n";
93     std::vector<RoadSegment*> segments = Road::shared().segments();

94     for (int i = 0; i < 10000; ++i) {
95         usleep(100);
96         m_traffic->force_place_car(segments[0],segments[0]->get_nodes()[0],1,1,1);
97     }

98     delete_cars_test();
99     //m_traffic.despawn_all_cars();
100     std::cout << "Placement tests 3 passed\n";
101 }

102 // do all tests
103 void Tests::run_all_tests() {
104     usleep(2000000);
105     placement_test();
106     delete_cars_test();
107     run_one_car();
108     placement_test_2();
109     placement_test_3();

110     std::cout << "all tests passed\n";
111 }

112 Tests::Tests(Traffic *& traffic, sf::Mutex *& mutex) {
113     m_traffic = traffic;

```

```

122 }
    m_mutex = mutex;

```

../highway/cppfiles/unittests.cpp

B.9 util.cpp

```

1  //
2  // Created by Carl Schiller on 2019-03-04.
3  //
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
    delim) {
14     std::stringstream ss(str);
15     std::string item;
16     std::vector<std::string> answer;
17     while(std::getline(ss, item, delim)){
18         answer.push_back(item);
19     }
20     return answer;
21 }
22
23 ///////////////////////////////////////////////////////////////////
24 /// Returns true if @param a is behind @param b, else false
25
26 bool Util::is_car_behind(Car * a, Car * b){
27     if(a!=b){
28         float theta_to_car_b = atan2(a->y_pos()-b->y_pos(),b->x_pos()-a->x_pos());
29         float theta_difference = get_min_angle(a->theta(),theta_to_car_b);
30         return theta_difference < MPI*0.45;
31     }
32     else{
33         return false;
34     }
35 }
36
37 ///////////////////////////////////////////////////////////////////
38 /// Returns true if @param a will cross paths with @param b, else false.
39 /// NOTE: @param a MUST be behind @param b.
40
41 bool Util::will_car_paths_cross(Car *a, Car *b) {
42     //simulate car a driving straight ahead.
43     RoadSegment * inspecting_segment = a->get_segment();
44     //RoadNode * node_0 = a->current_node;
45     RoadNode * node_1 = a->heading_to_node;
46
47     //int node_0_int = inspecting_segment->get_lane_number(node_0);
48     int node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
49
50     while(!node_1->get_nodes_from_me().empty()){
51         for(Car * car : inspecting_segment->m_cars){
52             if(car == b){
53                 // place logic for evaluating if we cross cars here.
54                 // heading to same node, else return false
55                 return node_1 == b->heading_to_node;
56             }
57         }
58     }
59 }

```

```

59     inspecting_segment = node_1->get_parent_segment();
60     //node_0_int = node_1_int;
61     //node_0 = node_1;
62
63     // if we are at say, 2 lanes and heading to 2 lanes, keep previous lane numbering.
64     if(inspecting_segment->get_total_amount_of_lanes() == node_1->get_nodes_from_me().
65 size()){
66         node_1 = node_1->get_nodes_from_me()[node_1_int];
67     }
68     // if we get one option, stick to it.
69     else if(node_1->get_nodes_from_me().size() == 1){
70         node_1 = node_1->get_nodes_from_me()[0];
71     }
72
73     // we merge from 3 to 2.
74     else if(inspecting_segment->get_total_amount_of_lanes() == 3 && inspecting_segment->
75 merge){
76         node_1 = node_1->get_nodes_from_me()[std::max(node_1_int-1,0)];
77     }
78
79     node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
80 }
81 return false;
82 }
83
84 /*
85 bool Util::merge_helper(Car *a, int merge_to_lane) {
86     RoadSegment * seg = a->current_segment;
87     for(Car * car : seg->m_cars){
88         if(car != a){
89             float delta_speed = a->speed()-car->speed();
90             if(car->heading_to_node == a->current_node->get_nodes_from_me()[merge_to_lane]
91 && delta_speed < 0){
92                 return true;
93             }
94         }
95     }
96     return false;
97 }
98
99 */
100
101 /*
102 // this works only if a's heading to is b's current segment
103 bool Util::is_cars_in_same_lane(Car *a, Car *b) {
104     return a->heading_to_node == b->current_node;
105 }
106
107 */
108
109 /*
110 float Util::distance_to_line(const float theta, const float x, const float y){
111     float x_hat, y_hat;
112     x_hat = cos(theta);
113     y_hat = -sin(theta);
114
115     float proj_x = (x*x_hat+y*y_hat)*x_hat;
116     float proj_y = (x*x_hat+y*y_hat)*y_hat;
117     float dist = sqrt(abs(pow(x-proj_x,2.0f))+abs(pow(y-proj_y,2.0f)));
118
119     return dist;
120 }
121 */
122
123

```

```

125 /*
126 float Util::distance_to_proj_point(const float theta, const float x, const float y){
127     float x_hat,y_hat;
128     x_hat = cos(theta);
129     y_hat = -sin(theta);
130     float proj_x = (x*x_hat+y*y_hat)*x_hat;
131     float proj_y = (x*x_hat+y*y_hat)*y_hat;
132     float dist = sqrt(abs(pow(proj_x,2.0f))+abs(pow(proj_y,2.0f)));
133
134     return dist;
135 }
136 */
137
138 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
139 /// Returns distance between @param a and @param b.
140
141 float Util::distance_to_car(Car * a, Car * b){
142     if(a == nullptr || b == nullptr){
143         throw std::invalid_argument("Can't calculate distance if cars are nullptrs");
144     }
145
146     float delta_x = a->x_pos()-b->x_pos();
147     float delta_y = b->y_pos()-a->y_pos();
148
149     return sqrt(abs(pow(delta_x,2.0f))+abs(pow(delta_y,2.0f)));
150 }
151
152 /*
153 Car * Util::find_closest_radius(std::vector<Car> &cars, const float x, const float y){
154     Car * answer = nullptr;
155
156     float score = 100000;
157     for(Car & car : cars){
158         float distance = sqrt(abs(pow(car.x_pos()-x,2.0f))+abs(pow(car.y_pos()-y,2.0f)));
159         if(distance < score){
160             score = distance;
161             answer = &car;
162         }
163     }
164
165     return answer;
166 }
167 */
168
169 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
170 /// Returns min angle between @param ang1 and @param ang2
171
172 float Util::get_min_angle(const float ang1, const float ang2){
173     float abs_diff = abs(ang1-ang2);
174     float score = std::min(2.0f*(float)M_PI-abs_diff,abs_diff);
175     return score;
176 }
177
178 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
179 /// Returns distance between two points in 2D.
180
181 float Util::distance(float x1, float x2, float y1, float y2) {
182     return sqrt(abs(pow(x1-x2,2.0f))+abs(pow(y1-y2,2.0f)));
183 }

```

../highway/cppfiles/util.cpp