

Final Project, SI1336

Carl Schiller, 9705266436

March 4, 2019

Abstract

Contents

1	Introduction	1
2	Method	1
3	Result	1
4	Discussion	1
A	Header files	2
A.1	cars.h	2
A.2	road.h	2
A.3	roadnode.h	3
A.4	roadsegment.h	4
A.5	simulation.h	5
A.6	traffic.h	5
A.7	unittests.h	6
A.8	util.h	7
B	Source files	7
B.1	cars.cpp	7
B.2	main.cpp	14
B.3	road.cpp	15
B.4	roadnode.cpp	18
B.5	roadsegment.cpp	20
B.6	simulation.cpp	23
B.7	traffic.cpp	24
B.8	unittests.cpp	30
B.9	util.cpp	31

1 Introduction

2 Method

3 Result

4 Discussion

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 public:
30     Car();
31     ~Car();
32     Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float aggressiveness);
33     Car(RoadSegment * spawn_point, RoadNode * lane, float vel, float target_speed, float aggressiveness);
34
35     // all are raw pointers
36     RoadSegment * current_segment;
37     RoadNode * current_node;
38     RoadNode * heading_to_node;
39     Car * overtake_this_car;
40
41     void update_pos(float delta_t);
42     void merge(std::vector<RoadNode*> & connections);
43     void do_we_want_to_overtake(Car * & closest_car, int & current_lane);
44     void accelerate(float delta_t);
45     void avoid_collision(float delta_t);
46     Car * find_closest_car_ahead();
47     std::map<Car *, bool> find_cars_around_car();
48
49     float x_pos();
50     float y_pos();
51
52     float & speed();
53     float & target_speed();
54     float & theta();
55
56     RoadSegment * get_segment();
57 };
58
59 #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
```

```

7  #define HIGHWAY_ROAD_H
9  ////////////////////////////////////////////////////////////////////
11 // Road
13 // Describes a road with interconnected nodes. Mathematically it is
15 // a graph.
17 ////////////////////////////////////////////////////////////////////
19 #include "roadsegment.h"
21 #include <vector>
23 #include <string>
25
27 class Road{
29 private:
31     std::vector<RoadSegment*> m_segments; // OWNERSHIP
33     std::vector<RoadSegment*> m_spawn_positions; // raw pointers
35     std::vector<RoadSegment*> m_despawn_positions; // raw pointers
37
39     const std::string M_FILENAME;
41 private:
43     Road();
45     ~Road();
47 public:
49     static Road &shared() {static Road road; return road;} // in order to only load road once in memory
51
53     Road(const Road& copy) = delete; // no copying allowed
55     Road& operator=(const Road& rhs) = delete; // no copying allowed
57
59     bool load_road();
61     std::vector<RoadSegment*> & spawn_positions();
63     std::vector<RoadSegment*> & despawn_positions();
65     std::vector<RoadSegment*> & segments();
67 };
69
71 #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  // RoadNode
11 //
13 // Describes the smallest element in Road, it is similar to
15 // that of a mathematical graph with nodes and edges.
17 ////////////////////////////////////////////////////////////////////
19
21 #include <vector>
23 #include "car.h"
25 #include "roadsegment.h"
27
29 class RoadNode{
31 private:
33     float m_x, m_y;
35     std::vector<RoadNode*> m_nodes_from_me; // raw pointers, no ownership
37     std::vector<RoadNode*> m_nodes_to_me;
39     RoadSegment* m_is_child_of; // raw pointer, no ownership
41 public:
43     RoadNode();
45     ~RoadNode();
47     RoadNode(float x, float y, RoadSegment * segment);
49
51 };

```

```

    void set_next_node(RoadNode *);
33 void set_previous_node(RoadNode *);
    RoadSegment* get_parent_segment();
35 RoadNode * get_next_node(int lane);
    std::vector<RoadNode*> & get_nodes_from_me();
37 std::vector<RoadNode*> & get_nodes_to_me();
    float get_x();
39 float get_y();
    float get_theta(RoadNode*);
41 };
43 #endif //HIGHWAY_ROADNODE_H

```

../highway/headers/roadnode.h

A.4 roadsegment.h

```

//
2 // Created by Carl Schiller on 2019-03-04.
//
4 #ifndef HIGHWAY_ROADSEGMENT_H
6 #define HIGHWAY_ROADSEGMENT_H
8 ///////////////////////////////////////////////////////////////////
10 // RoadSegment //
12 // Describes a container for several RoadNodes //
14 ///////////////////////////////////////////////////////////////////
16 #include <vector>
18 class RoadNode;
20 class Car;
22 class RoadSegment{
23 private:
24     const float m_x, m_y;
25     float m_theta;
26     const int m_n_lanes;
28     constexpr static float M_LANE_WIDTH = 4.0f;
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
44     RoadNode * get_node_pointer(int n);
45     std::vector<RoadNode*> get_nodes();
46     void append_car(Car*);
47     void remove_car(Car*);
48     RoadSegment * next_segment();
49     float get_theta();
50     const float get_x() const;
51     const float get_y() const;
52
53     int get_lane_number(RoadNode *);
54     const int get_total_amount_of_lanes() const;
55     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 //
2 // Created by Carl Schiller on 2018-12-19.
3 //
4
5 #ifndef HIGHWAY.WINDOW.H
6 #define HIGHWAY.WINDOW.H
7
8 ////////////////////////////////////////////////////////////////////
9 //                                                                    //
10 // Simulation                                                            //
11 //                                                                    //
12 // Describes how to simulate Traffic class                             //
13 //                                                                    //
14 ////////////////////////////////////////////////////////////////////
15
16 #include <vector>
17 #include "SFML/Graphics.hpp"
18 #include "traffic.h"
19
20 class Simulation{
21 private:
22     sf::Mutex * m_mutex;
23     Traffic * m_traffic;
24     bool * m_exit_bool;
25     const int M_SIM_SPEED;
26     const int M_FRAMERATE;
27 public:
28     Simulation() = delete;
29     Simulation(Traffic *& traffic, sf::Mutex *& mutex, int sim_speed, int m_framerate, bool *& exitbool);
30
31     void update();
32 };
33
34 #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"

```

```

#include "car.h"
21
class Traffic : public sf::Drawable, public sf::Transformable{
23 private:
    std::vector<Car*> m_cars;
25     bool debug;
    std::mt19937 & my_engine();
27     sf::Font m_font;

29 public:
    Traffic();
31     explicit Traffic(bool debug);
    ~Traffic();
33     Traffic(const Traffic&); // rule of three
    Traffic& operator=(const Traffic&); // rule of three
35
    unsigned long n_of_cars();
37     void spawn_cars(double & spawn_counter, float elapsed, double & threshold);
    void despawn_cars();
39     void despawn_all_cars();
    void despawn_car(Car& car);
41     void force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, float aggro);

43
    void update(float elapsed_time);
45     std::vector<Car *> get_car_copies() const;
    float get_avg_flow();
47     std::vector<float> get_avg_speeds();
private:
49     virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
public:
51     void get_info(sf::Text & text, sf::Time &elapsed);
    double m_multiplier;
53 };

55 #endif //HIGHWAY_TRAFFIC.H

```

../highway/headers/traffic.h

A.7 unittests.h

```

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

5
6 #ifndef HIGHWAY_UNITTESTS_H
7 #define HIGHWAY_UNITTESTS_H

9 ///////////////////////////////////////////////////////////////////
// Tests //
11 // Testing the various functions. //
13 ///////////////////////////////////////////////////////////////////

15

17 #include "traffic.h"
#include "SFML/Graphics.hpp"

19
20 class Tests{
21 private:
    Traffic * m_traffic;
23     sf::Mutex * m_mutex;
    void placement_test();
25     void delete_cars_test();
    void run_one_car();
27     void placement_test_2();
    void placement_test_3();
29 public:
    Tests() = delete;
31     Tests(Traffic *& traffic, sf::Mutex *& mutex);

33     void run_all_tests();

```

```

};
35 #endif //HIGHWAY_UNITTESTS_H

```

../highway/headers/unittests.h

A.8 util.h

```

//
2 // Created by Carl Schiller on 2019-03-04.
//
4
6 #ifndef HIGHWAY_UTIL_H
7 #define HIGHWAY_UTIL_H
8
9 ////////////////////////////////////////////////////////////////////
10 // Util
11 //
12 // Help functions for Car class.
13 //
14 ////////////////////////////////////////////////////////////////////
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 };
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
6 #include "../headers/car.h"
7 #include <map>
8 #include <cmath>
9 #include <list>
10 #include "../headers/util.h"
11
12 ////////////////////////////////////////////////////////////////////
13 /// Constructor.
14
15 Car::Car() = default;
16
17 ////////////////////////////////////////////////////////////////////
18 /// Constructor for new car with specified lane numbering in spawn point.
19 /// Lane numbering @param lane must not exceed amount of lanes in
20 /// @param spawn_point, otherwise an exception will be thrown.
21
22 Car::Car(RoadSegment *spawn_point, int lane, float vel, float target_speed, float aggressivness):
23     m_speed(vel),
24     m_aggressiveness(aggressivness),
25     m_target_speed(target_speed),
26     current_segment(spawn_point),
27     current_node(current_segment->get_node_pointer(lane)),
28     overtake_this_car(nullptr)

```

```

28 {
    current_segment->append_car(this);
30
    if(!current_node->get_nodes_from_me().empty()){
32         heading_to_node = current_node->get_next_node(lane);

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());

36         m_theta = current_node->get_theta(heading_to_node);
    }
38     else{
        throw std::invalid_argument("Car spawns in node with empty connections, or with a nullptr segment"
40     );
    }
}

42
44 //////////////////////////////////////////////////
45 /// Constructor for new car with specified lane. Note that
46 /// @param lane must be in @param spawn_point, otherwise no guarantee on
47 /// functionality.

48 Car::Car(RoadSegment *spawn_point, RoadNode *lane, float vel, float target_speed, float aggressivness) :
    m_speed(vel),
    m_aggressiveness(aggressivness),
    m_target_speed(target_speed),
    current_segment(spawn_point),
    current_node(lane),
    overtake_this_car(nullptr)
49 {
50     current_segment->append_car(this);

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

54         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());

56         m_theta = current_node->get_theta(heading_to_node);
    }
58     else{
        throw std::invalid_argument("Car spawns in node with empty connections, or with a nullptr segment"
60     );
    }
}

62
64 //////////////////////////////////////////////////
65 /// Destructor for car.

66 Car::~Car(){
67     if(this->current_segment != nullptr){
68         this->current_segment->remove_car(this); // remove this pointer shit
69     }

70     overtake_this_car = nullptr;
71     current_segment = nullptr;
72     heading_to_node = nullptr;
73     current_node = nullptr;
74 }

76
78 //////////////////////////////////////////////////
79 /// Updates position for car with time step @param delta_t.

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

83     if(m_dist_to_next_node < 0){
84         current_segment->remove_car(this); // remove car from this segment
85         current_segment = heading_to_node->get_parent_segment(); // set new segment
86         if(current_segment != nullptr){
87             current_segment->append_car(this); // add car to new segment
88         }
89         current_node = heading_to_node; // set new current node as previous one.
90
91     }
92 }

```



```

100 //TODO: place logic for choosing next node
101 std::vector<RoadNode*> connections = current_node->get_nodes_from_me();
102
103 if(!connections.empty()){
104     merge(connections);
105
106     m_dist_to_next_node += Util::distance(current_node->get_x(), heading_to_node->get_x(),
107     current_node->get_y(), heading_to_node->get_y());
108     m_theta = current_node->get_theta(heading_to_node);
109 }
110 }
111 }
112 }
113
114 //////////////////////////////////////
115 /// Function to determine if we can merge into another lane depending on.
116 /// properties of @param connections.
117
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
130         ->heading_to_node) == 1 ){
131             can_merge =
132                 delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, 15.0f);
133         }
134         else if(current_lane == 1 && it.first->heading_to_node->get_parent_segment()->get_lane_number(it
135         first->heading_to_node) == 0){
136             can_merge =
137                 delta_dist > std::max(delta_speed*4.0f/m_aggressiveness, 15.0f);
138         }
139
140         if(!can_merge){
141             break;
142         }
143     }
144
145     if(current_segment->merge){
146         if(current_lane == 0 && connections[0]->get_parent_segment()->get_total_amount_of_lanes() != 2){
147             if(can_merge){
148                 heading_to_node = connections[1];
149             }
150             else{
151                 heading_to_node = connections[0];
152             }
153         }
154         else if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
155             current_lane = std::max(current_lane-1, 0);
156             heading_to_node = connections[current_lane];
157         }
158         else{
159             heading_to_node = connections[current_lane];
160         }
161     }
162
163     // if we are in start section
164     else if(current_segment->get_total_amount_of_lanes() == 3){
165         if(connections.size() == 1){
166             heading_to_node = connections[0];
167         }
168         else{
169             heading_to_node = connections[current_lane];
170         }
171     }
172
173     // if we are in middle section
174     else if(current_segment->get_total_amount_of_lanes() == 2){
175         // normal way
176         if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){

```

```

174 // check if we want to overtake car in front
do_we_want_to_overtake(closest_car , current_lane);

176 // committed to overtaking
177 if(overtake_this_car != nullptr){
178     if(current_lane != 1){
179         if(can_merge){
180             heading_to_node = connections[1];
181         }
182         else{
183             heading_to_node = connections[current_lane];
184         }
185     }
186     else{
187         heading_to_node = connections[current_lane];
188     }
189 }
190 // merge back if overtake this car is nullptr.
191 else{
192     if(can_merge){
193         heading_to_node = connections[0];
194     }
195     else{
196         heading_to_node = connections[current_lane];
197     }
198 }
199 }
200 }
201 else{
202     heading_to_node = connections[0];
203 }
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->target_speed() >
224 m_aggressiveness*1.0f ) && current_lane == 0 && closest_car->current_node->get_parent_segment()->
225 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 , overtake_this_car) >
233 30)){
234             overtake_this_car = nullptr;
235         }
236     }
237 }
238
239 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
240 /// Function to accelerate this car.
241
242 void Car::accelerate(float elapsed){
243     float target = m_target_speed;
244     float d_vel; // proportional control.
245
246     if(m_speed < target*0.75){
247         d_vel = m_aggressiveness*elapsed*2.0f;

```

```

246     }
247     else{
248         d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
249     }
250
251     m_speed += d_vel;
252 }
253
254 //////////////////////////////////////
255 /// Helper function to avoid collision with another car.
256
257 void Car::avoid_collision(float delta_t) {
258     float min_distance = 8.0f; // for car distance.
259     float ideal = min_distance+min_distance*(m_speed/20.f);
260
261     Car * closest_car = find_closest_car_ahead();
262     float detection_distance = m_speed*5.0f;
263
264     if(closest_car != nullptr) {
265         float radius_to_car = Util::distance_to_car(this, closest_car);
266         float delta_speed = closest_car->speed() - this->speed();
267
268         if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
269             m_speed -= std::max(std::max((radius_to_car-min_distance)*0.5f,0.0f),10.0f*delta_t);
270         }
271         else if(radius_to_car < min_distance){
272             m_speed -= std::max(std::max((min_distance-radius_to_car)*0.5f,0.0f),2.0f*delta_t);
273         }
274         else if(delta_speed < 0 && radius_to_car < detection_distance){
275             m_speed -= std::min(
276                 abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
277                 m_aggressiveness * 0.15f,
278                 10.0f * delta_t);
279         }
280         else {
281             accelerate(delta_t);
282         }
283
284         if(current_segment->merge){
285             std::map<Car*,bool> around = find_cars_around_car();
286             for(auto it : around){
287                 float delta_dist = Util::distance_to_car(it.first, this);
288                 delta_speed = abs(speed()-it.first->speed());
289
290                 if(it.first->current_node->get_parent_segment()->get_lane_number(it.first->current_node)
291 == 0 && delta_dist < ideal && this->current_segment->get_lane_number(current_node) == 1 && speed()/
292 target_speed() > 0.5){
293                     if(Util::is_car_behind(it.first, this)){
294                         accelerate(delta_t);
295                     }
296                     else{
297                         m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*delta_t);
298                     }
299                 }
300                 else if(it.first->current_node->get_parent_segment()->get_lane_number(it.first->
301 current_node) == 1 && this->current_segment->get_lane_number(current_node) == 0 && speed()/
302 target_speed() > 0.5 && delta_dist < ideal){
303                     if(Util::is_car_behind(this, it.first)){
304                         m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*delta_t);
305                     }
306                     else{
307                         accelerate(delta_t);
308                     }
309                 }
310             }
311         }
312         else{
313             accelerate(delta_t);
314         }
315     }
316
317     if(m_speed < 0){
318         m_speed = 0;

```

```

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())
345             ) < search_radius){
346                 visited[next_node] = true;
347
348                 for(Car * car : next_node->get_parent_segment()->m_cars){
349                     if(this != car){
350                         float radius = Util::distance_to_car(this,car);
351                         if(Util::is_car_behind(this,car) && Util::will_car_paths_cross(this,car) && radius
352                         < shortest_distance){
353                             shortest_distance = radius;
354                             answer = car;
355                         }
356                     }
357                 }
358
359                 // push in new nodes in front of list.
360                 for(RoadNode * node : next_node->get_nodes_from_me()){
361                     queue.push_front(node);
362                 }
363             }
364         }
365     }
366     return answer;
367 }
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 std::map<Car *,bool> Car::find_cars_around_car() {
373     const float search_radius = 40;
374     std::map<RoadNode*,bool> visited;
375     std::list<RoadNode*> queue;
376
377     for(RoadNode * node : (this->current_segment->get_nodes())){
378         queue.push_front(node);
379     }
380
381     std::map<Car *,bool> answer;
382     while(!queue.empty()){
383         RoadNode * next_node = queue.back(); // get last element
384         queue.pop_back(); // remove element
385
386         if(next_node != nullptr){
387             if(!visited[next_node] && Util::distance(x_pos(),next_node->get_x(),y_pos(),next_node->get_y())
388             ) < search_radius){
389                 visited[next_node] = true;

```

```

390         for(Car * car : next_node->get_parent_segment()->m_cars){
391             if(this != car){
392                 answer[car] = true;
393             }
394         }
395         // push in new nodes in front of list.
396         for(RoadNode * node : next_node->get_nodes_from_me()){
397             queue.push_front(node);
398         }
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
427 float Car::y_pos() {
428     float y_position;
429     if(heading_to_node != nullptr){
430         y_position = heading_to_node->get_y()+m_dist_to_next_node*sin(m_theta);
431     }
432     else{
433         y_position = current_node->get_y();
434     }
435
436     return y_position;
437 }
438
439 //////////////////////////////////////
440 /// Returns speed of car, as reference.
441
442 float & Car::speed() {
443     return m_speed;
444 }
445
446 //////////////////////////////////////
447 /// Returns target speed of car as reference.
448
449 float & Car::target_speed() {
450     return m_target_speed;
451 }
452
453 //////////////////////////////////////
454 /// Returns theta of car, the direction of the car. Defined in radians as a
455 /// mathematician would define angles.
456
457 float & Car::theta() {
458     return m_theta;
459 }
460
461 //////////////////////////////////////
462 /// Returns current segment car is in.
463
464 RoadSegment* Car::get_segment() {
465     return current_segment;

```

B.2 main.cpp

```

#include <iostream>
#include "SFML/Graphics.hpp"
#include "../headers/simulation.h"
#include "../headers/unittests.h"

sf::Mutex mutex;

int main() {
    sf::RenderWindow window(sf::VideoMode(550*2, 600*2), "My window");
    window.setFramerateLimit(60);

    int sim_speed = 1;
    bool debug = false;
    bool super_debug = false;

    sf::Texture texture;
    if(!texture.loadFromFile("../mall2.png"))
    {

    }

    sf::Sprite background;
    background.setTexture(texture);
    //background.setColor(sf::Color::Black);
    background.scale(2.0f, 2.0f);

    sf::Clock clock;
    sf::Clock t0;

    bool exit_bool = false;

    if(!super_debug){
        sf::Mutex * mutex1 = &mutex;
        bool * exit = &exit_bool;
        //thread.launch();
        auto * traffic = new Traffic(debug);
        Simulation sim = Simulation(traffic, mutex1, sim_speed, 60, exit);
        sf::Text debug_info;
        Traffic copy;

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

        // run the program as long as the window is open
        while (window.isOpen())
        //while(false)
        {
            // check all the window's events that were triggered since the last iteration of the loop
            sf::Event event;
            while (window.pollEvent(event))
            {
                // "close requested" event: we close the window
                if (event.type == sf::Event::Closed){
                    exit_bool = true;
                    thread.wait();
                    window.close();
                }
            }
            sf::Time elapsed = clock.restart();

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

            window.clear(sf::Color(255, 255, 255, 255));

```

```

70     window.draw(background);
71     //mutex.lock();
72     window.draw(copy);
73
74     copy.get_info(debug_info, elapsed);
75
76     //mutex.unlock();
77     window.draw(debug_info);
78
79     window.display();
80 }
81 }
82 else{
83
84
85     //sf::Thread thread(&Tests::run_all_tests,&tests);
86     sf::Mutex * mutex1 = &mutex;
87     //thread.launch();
88     auto * traffic = new Traffic();
89     Tests tests = Tests(traffic, mutex1);
90     Traffic copy;
91     sf::Text debug_info;
92
93     sf::Thread thread(&Tests::run_all_tests,&tests);
94     thread.launch();
95
96     // run the program as long as the window is open
97     while (window.isOpen())
98     {
99
100         // check all the window's events that were triggered since the last iteration of the loop
101         sf::Event event;
102         while (window.pollEvent(event))
103         {
104             // "close requested" event: we close the window
105             if (event.type == sf::Event::Closed){
106                 //thread.terminate();
107                 window.close();
108                 thread.terminate();
109                 delete traffic;
110             }
111         }
112         //Traffic copy = tests.m_traffic; // deep copy it
113         sf::Time elapsed = clock.restart();
114
115         window.clear(sf::Color(255,255,255,255));
116
117         mutex.lock();
118         copy = *traffic;
119         mutex.unlock();
120
121         window.draw(background);
122         window.draw(copy);
123
124         copy.get_info(debug_info, elapsed);
125         window.draw(debug_info);
126
127         window.display();
128     }
129 }
130
131 return 0;
132 }

```

../highway/cppfiles/main.cpp

B.3 road.cpp

```

2 //
3 // Created by Carl Schiller on 2019-03-04.
4 //
5
6 #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)
58 /// tokens[5+3*n]: to segment number.
59
60 bool Road::load_road() {
61     bool loading = true;
62     std::ifstream stream;
63     stream.open(MFILENAME);
64
65     std::vector<std::vector<std::string>> road_vector;
66     road_vector.reserve(100);
67
68     if(stream.is_open()){
69         std::string line;
70         std::vector<std::string> tokens;
71         while(std::getline(stream, line)){
72             tokens = Util::split_string_by_delimiter(line, ' ');
73             if(tokens[0] != "#"){
74                 road_vector.push_back(tokens);
75             }
76         }
77     }
78     else{
79         loading = false;
80     }
81 }

```



```

82 // load segments into memory.
83 for (std::vector<std::string> & vec : road_vector){
84     if (vec.size() == 5){
85         if (vec[4] == "merge"){
86             RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]),
87 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]),
92 false);
93             m_segments.push_back(seg);
94         }
95     }
96     else{
97         RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]),
98 false);
99         m_segments.push_back(seg);
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         else if (road_vector[i][4] == "true"){
123             m_segments[i]->set_next_road_segment(m_segments[i+1]);
124             m_segments[i]->calculate_theta();
125             // calculate nodes based on theta.
126             m_segments[i]->calculate_and_populate_nodes();
127
128             // make this a spawn segment
129             m_spawn_positions.push_back(m_segments[i]);
130         }
131         else if (road_vector[i][4] == "merge"){
132             m_segments[i]->set_next_road_segment(m_segments[i+1]);
133             m_segments[i]->calculate_theta();
134             // calculate nodes based on theta.
135             m_segments[i]->calculate_and_populate_nodes();
136         }
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     }

```

```

156     else if(road_vector[i].size() == 5){
157         if(road_vector[i][4] == "false"){
158             // but do not connect nodes to new ones.
159         }
160         else if(road_vector[i][4] == "true"){
161             m_segments[i]->set_all_node_pointers_to_next_segment();
162         }
163         else if(road_vector[i][4] == "merge"){
164             m_segments[i]->set_all_node_pointers_to_next_segment();
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+2])];
174             m_segments[i]->set_node_pointer_to_node(std::stoi(road_vector[i][current_pos]), std::stoi(
175             road_vector[i][current_pos+1]), next_segment);
176         }
177     }
178     return loading;
179 }
180
181 //////////////////////////////////////////////////
182 /// Returns spawn positions of Road
183
184 std::vector<RoadSegment*>& Road::spawn_positions() {
185     return m_spawn_positions;
186 }
187
188 //////////////////////////////////////////////////
189 /// Returns despawn positions of Road
190
191 std::vector<RoadSegment*>& Road::despawn_positions() {
192     return m_despawn_positions;
193 }
194
195 //////////////////////////////////////////////////
196 /// Returns all segments of Road.
197
198 std::vector<RoadSegment*>& Road::segments() {
199     return m_segments;
200 }

```

../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;
    m_y = y;
    m_is_child_of = segment;
26 }

28 ///////////////////////////////////////////////////////////////////
/// Appends a new RoadNode to the list connections from this RoadNode.
30 /// I.e. to where a Car is allowed to drive.

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

36 ///////////////////////////////////////////////////////////////////
38 /// Appends a new RoadNode to the list connections to this RoadNode.
/// I.e. from where a Car is allowed to drive to this Node.
40
42 void RoadNode::set_previous_node(RoadNode * prev_node) {
    m_nodes_to_me.push_back(prev_node);
44 }

46 ///////////////////////////////////////////////////////////////////
/// Returns RoadSegment to which this RoadNode belongs.

48 RoadSegment* RoadNode::get_parent_segment() {
    return m_is_child_of;
50 }

52 ///////////////////////////////////////////////////////////////////
/// Returns connections from this RoadNode.

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

58 ///////////////////////////////////////////////////////////////////
60 /// Returns connections to this RoadNode.

62 std::vector<RoadNode*>& RoadNode::get_nodes_to_me() {
    return m_nodes_to_me;
64 }

66 ///////////////////////////////////////////////////////////////////
/// Returns x position of RoadNode.

68 float RoadNode::get_x() {
70     return m_x;
}

72 ///////////////////////////////////////////////////////////////////
74 /// Returns y position of RoadNode.

76 float RoadNode::get_y() {
    return m_y;
78 }

80 ///////////////////////////////////////////////////////////////////
/// Returns angle of this RoadNode to @param node as a mathematician
82 /// would define angles. In radians.

84 float RoadNode::get_theta(RoadNode* node) {
    for(RoadNode * road_node : m_nodes_from_me){
86         if(node == road_node){
            return atan2(m_y-node->m_y, node->m_x-m_x);
88         }
    }
90     throw std::invalid_argument("Node given is not a connecting node");
}

92 ///////////////////////////////////////////////////////////////////
94 /// Returns RoadNode according to @param lane from the vector of node
/// connections from this RoadNode.

96 RoadNode* RoadNode::get_next_node(int lane) {
98     return m_nodes_from_me[lane];

```

```
}
```

```
../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     m_nodes.reserve(m_n_lanes);
33     calculate_and_populate_nodes(); // populates segment with RoadNodes.
34 }
35
36 //////////////////////////////////////
37 // Constructor, creates a new segment with manually entered @param theta.
38
39 RoadSegment::RoadSegment(float x, float y, float theta, int lanes) :
40     m_x(x),
41     m_y(y),
42     m_theta(theta),
43     m_n_lanes(lanes),
44     m_next_segment(nullptr)
45 {
46     m_nodes.reserve(m_n_lanes);
47     calculate_and_populate_nodes(); // populates segment with RoadNodes.
48 }
49
50 //////////////////////////////////////
51 // Constructor, creates a new segment without creating RoadNodes. This
52 // needs to be done manually with functions below.
53
54 RoadSegment::RoadSegment(float x, float y, int lanes, bool mer):
55     m_x(x),
56     m_y(y),
57     m_n_lanes(lanes),
58     m_next_segment(nullptr),
59     merge(mer)
60 {
61     m_nodes.reserve(m_n_lanes);
62     // can't set nodes if we don't have a theta.
63 }
64
65 //////////////////////////////////////
```

```

70 // Returns theta (angle) of RoadSegment, in which direction the segment points
71 float RoadSegment::get_theta() {
72     return m_theta;
73 }
74
75 // Returns x position of RoadSegment.
76
77 const float RoadSegment::get_x() const {
78     return m_x;
79 }
80
81 // Returns y position of RoadSegment.
82
83 const float RoadSegment::get_y() const {
84     return m_y;
85 }
86
87 // Returns int number of @param node. E.g. 0 would be the right-most lane.
88 // Throws exception if we do not find the node in this segment.
89
90 int RoadSegment::get_lane_number(RoadNode * node) {
91     for(int i = 0; i < m_n_lanes; i++){
92         if(node == m_nodes[i]){
93             return i;
94         }
95     }
96     throw std::invalid_argument("Node is not in this segment");
97 }
98
99 // Adds a new car to the segment.
100
101 void RoadSegment::append_car(Car * car) {
102     m_cars.push_back(car);
103 }
104
105 // Removes car from segment, if car is not in list we throw exception
106
107 void RoadSegment::remove_car(Car * car) {
108     unsigned long size = m_cars.size();
109     bool found = false;
110     for(int i = 0; i < size; i++){
111         if(car == m_cars[i]){
112             m_cars[i] = nullptr;
113             found = true;
114         }
115     }
116     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(), static_cast<Car*>(
117         nullptr));
118     m_cars.erase(new_end, m_cars.end());
119
120     if(!found){
121         throw std::invalid_argument("Car is not in this segment.");
122     }
123 }
124
125 // Sets theta of RoadSegment according to @param theta.
126
127 void RoadSegment::set_theta(float theta) {
128     m_theta = theta;
129 }
130
131 // Automatically populates segment with nodes according to amount of lanes
132 // specified and theta specified.
133
134 void RoadSegment::calculate_and_populate_nodes() {
135     // calculates placement of nodes.
136     float total_length = M_LANE_WIDTH*(m_n_lanes-1);
137     float current_length = -total_length/2.0f;

```

```

144     for(int i = 0; i < m_n_lanes; i++){
145         float x_pos = m_x+current_length*cos(m_theta+(float)M_PI*0.5f);
146         float y_pos = m_y-current_length*sin(m_theta+(float)M_PI*0.5f);
147         m_nodes.push_back(new RoadNode(x_pos,y_pos,this));
148         current_length += MLANE.WIDTH;
149     }
150 }
151
152 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
153 /// Sets next segment to @param next_segment
154
155 void RoadSegment::set_next_road_segment(RoadSegment * next_segment) {
156     m_next_segment = next_segment;
157 }
158
159 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
160 /// Calculates theta according to next_segment. Throws if m_next_segment is
161 /// nullptr
162
163 void RoadSegment::calculate_theta() {
164     if(m_next_segment == nullptr){
165         throw std::invalid_argument("Can't calculate theta if next segment is nullptr");
166     }
167     m_theta = atan2(m_y-m_next_segment->m_y,m_next_segment->m_x-m_x);
168 }
169
170 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
171 /// Returns node of lane number n. E.g. n=0 is the right-most lane.
172
173 RoadNode* RoadSegment::get_node_pointer(int n) {
174     return m_nodes[n];
175 }
176
177 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
178 /// Returns all nodes in segment.
179
180 std::vector<RoadNode*> RoadSegment::get_nodes() {
181     return m_nodes;
182 }
183
184 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
185 /// Returns next segment
186
187 RoadSegment* RoadSegment::next_segment() {
188     return m_next_segment;
189 }
190
191 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
192 /// Automatically populates node connections by connecting current node to
193 /// all nodes in next segment.
194
195 void RoadSegment::set_all_node_pointers_to_next_segment() {
196     for(RoadNode * node: m_nodes){
197         for(int i = 0; i < m_next_segment->m_n_lanes; i++){
198             node->set_next_node(m_next_segment->get_node_pointer(i));
199         }
200     }
201 }
202
203 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
204 /// Manually set connection to next segment's node. No guarantee is made
205 /// on @param from_node_n and @param to_node_n. Can crash if index out of range.
206
207 void RoadSegment::set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *next_segment) {
208     RoadNode * pointy = next_segment->get_node_pointer(to_node_n);
209     m_nodes[from_node_n]->set_next_node(pointy);
210 }
211
212 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
213 /// Returns amount of lanes in this segment.
214
215 const int RoadSegment::get_total_amount_of_lanes() const {
216     return m_n_lanes;
217 }
218 }

```

B.6 simulation.cpp

```

2 //
3 // Created by Carl Schiller on 2018-12-19.
4 //
5
6 #include <iostream>
7 #include "../headers/traffic.h"
8 #include "../headers/simulation.h"
9 #include <cmath>
10 #include <unistd.h>
11
12 //////////////////////////////////////
13 /// Constructor
14 /// @param traffic : pointer reference to Traffic, this is to be able to
15 /// draw traffic outside of this class.
16 /// @param mutex : mutex thread lock from SFML.
17 /// @param sim_speed : Simulation speed multiplier, e.g. 10 would mean 10x
18 /// real time speed. If simulation can not keep up it lowers this.
19 /// @param framerate : Framerate of simulation, e.g. 60 FPS. This is the
20 /// time step of the system.
21 /// @param exit_bool : If user wants to exit this is changed outside of the class.
22
23 Simulation::Simulation(Traffic *&traffic, sf::Mutex *&mutex, int sim_speed, int framerate, bool *&
24 exit_bool):
25     m_mutex(mutex),
26     m_traffic(traffic),
27     m_exit_bool(exit_bool),
28     M_SIM_SPEED(sim_speed),
29     M_FRAMERATE(framerate)
30 {
31 }
32
33 //////////////////////////////////////
34 /// Runs simulation. If M_SIM_SPEED = 10, then it simulates 10x1/(M_FRAMERATE)
35 /// seconds of real time simulation.
36
37 void Simulation::update() {
38     sf::Clock clock;
39     sf::Time time;
40     double spawn_counter = 0.0;
41     double threshold = 0.0;
42
43     while(!m_exit_bool){
44         m_mutex->lock();
45         //std::cout << "calculating\n";
46         for(int i = 0; i < M_SIM_SPEED; i++){
47             //std::cout << "a\n";
48             m_traffic->update(1.0f/(float)M_FRAMERATE);
49             //std::cout << "b\n";
50             m_traffic->spawn_cars(spawn_counter, 1.0f/(float)M_FRAMERATE, threshold);
51             //m_mutex->lock();
52             //std::cout << "c\n";
53             m_traffic->despawn_cars();
54             //m_mutex->unlock();
55             //std::cout << "d\n";
56         }
57         //std::cout << "calculated\n";
58         m_mutex->unlock();
59
60         time = clock.restart();
61         sf::Int64 actual_elapsed = time.asMicroseconds();
62         double sim_elapsed = (1.0f/(float)M_FRAMERATE)*1000000;
63
64         if(actual_elapsed < sim_elapsed){
65             usleep((useconds_t)(sim_elapsed-actual_elapsed));
66             m_traffic->m_multiplier = M_SIM_SPEED;
67         }
68         else{

```

```

68         m_traffic->m_multiplier = M_SIM_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());
44
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
    for(auto it : from_to){
69        m_cars[it.first]->overtake_this_car = m_cars[it.second];
        //m_cars[it.second]->want_to_overtake_me.push_back(m_cars[it.first]);
71    }
}

73
75 ///////////////////////////////////////////////////
76 /// Copy-assignment constructor, deep copies all content and swaps.
77
78 Traffic& Traffic::operator=(const Traffic & rhs) {
79     Traffic tmp(rhs);
80
81     std::swap(m_cars,tmp.m_cars);
82     std::swap(m_multiplier,tmp.m_multiplier);
83     std::swap(debug,tmp.debug);
84
85     return *this;
86 }
87
88 ///////////////////////////////////////////////////
89 /// Destructor, deletes all cars.
90
91 Traffic::~Traffic() {
92     for(Car * & car : m_cars){
93         delete car;
94     }
95     Traffic::m_cars.clear();
96 }
97
98 ///////////////////////////////////////////////////
99 /// Returns size of car vector
100
101 unsigned long Traffic::n_of_cars(){
102     return m_cars.size();
103 }
104
105 ///////////////////////////////////////////////////
106 /// Random generator, returns reference to random generator in order to,
107 /// not make unnecessary copies.
108
109 std::mt19937& Traffic::my_engine() {
110     static std::mt19937 e(std::random_device{}());
111     return e;
112 }
113
114 ///////////////////////////////////////////////////
115 /// Logic for spawning cars by looking at how much time has elapsed.
116 /// @param spawn_counter : culmulative time elapsed
117 /// @param elapsed : time elapsed for one time step.
118 /// @param threshold : threshold is set by randomly selecting a poission
119 /// distributed number.
120 ///
121 /// Cars that are spawned are poission distributed in time, the speed of the
122 /// cars are normally distributed according to their aggressiveness.
123
124 void Traffic::spawn_cars(double & spawn_counter, float elapsed, double & threshold) {
125     spawn_counter += elapsed;
126     if(spawn_counter > threshold){
127         std::exponential_distribution<double> dis(5);
128         std::normal_distribution<float> aggro(1.0f,0.2f);
129         float sp = 30.0f;
130         std::uniform_real_distribution<float> lane(0.0f,1.0f);
131         std::uniform_real_distribution<float> spawn(0.0f,1.0f);
132
133         threshold = dis(my_engine());
134         float aggressiveness = aggro(my_engine());
135         float speed = sp*aggressiveness;
136         float target = speed;
137
138         spawn_counter = 0;
139         float start_lane = lane(my_engine());
140         float spawn_pos = spawn(my_engine());
141
142         std::vector<RoadSegment*> segments = Road::shared().spawn_positions();

```

```

143 RoadSegment * seg;
144 Car * new_car;
145 if(spawn_pos < 0.95){
146     seg = segments[0];
147     if(start_lane < 0.457){
148         new_car = new Car(seg,2,speed,target,aggressiveness);
149     }
150     else if(start_lane < 0.95){
151         new_car = new Car(seg,1,speed,target,aggressiveness);
152     }
153     else{
154         new_car = new Car(seg,0,speed,target,aggressiveness);
155     }
156 }
157 else{
158     seg = segments[1];
159     new_car = new Car(seg,0,speed,target,aggressiveness);
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 }
180 }
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){

```

```

219         for(auto it : to_delete){
220             if(it.first == car->overtake_this_car){
221                 car->overtake_this_car = nullptr;
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(), static_cast<Car*>(
234         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.
248 ///
249 /// \param seg : segment of car
250 /// \param node : node of car
251 /// \param vel : (current)velocity of car
252 /// \param target : target velocity of car
253 /// \param aggro : agressiveness of car
254
255 void Traffic::force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, 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
268     for(Car * & car : m_cars){
269         car->update_pos(elapsed_time);
270     }
271 }
272
273 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
274 /// Returns vector of all cars.
275
276 std::vector<Car *> Traffic::get_car_copies() const {
277     return m_cars;
278 }
279
280 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
281 /// Returns average flow of all cars. Average value of
282 /// quotient of current speed divided by target speed for all cars.
283
284 float Traffic::get_avg_flow() {
285     float flow = 0;
286     float i = 0;
287     for(Car * car : m_cars){
288         i++;
289         flow += car->speed()/car->target_speed();
290     }
291     if(m_cars.empty()){
292         return 0;
293     }

```

```

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){
314         i++;
315         flow += car->speed()*3.6f;
316
317         if(car->current_segment->get_total_amount_of_lanes() == 2){
318             if(car->current_segment->get_lane_number(car->current_node) == 1){
319                 flow_left += car->speed()*3.6f;
320                 j++;
321             }
322             else{
323                 flow_right += car->speed()*3.6f;
324                 k++;
325             }
326         }
327     }
328     if(m_cars.empty()){
329         return speedy;
330     }
331     else{
332         flow = flow/i;
333         flow_left = flow_left/j;
334         flow_right = flow_right/k;
335         speedy.push_back(flow);
336         speedy.push_back(flow_left);
337         speedy.push_back(flow_right);
338         return speedy;
339     }
340 }
341
342 ///////////////////////////////////////////////////////////////////
343 /// Draws cars (and nodes if debug = true) to @param target, which could
344 /// be a window. Blue cars are cars that want to overtake someone,
345 /// green cars are driving as fast as they want (target speed),
346 /// red cars are driving slower than they want.
347
348 void Traffic::draw(sf::RenderTarget &target, sf::RenderStates states) const {
349     // print debug info about node placements and stuff
350
351     sf::CircleShape circle;
352     circle.setRadius(4.0f);
353     circle.setOutlineColor(sf::Color::Cyan);
354     circle.setOutlineThickness(1.0f);
355     circle.setFillColor(sf::Color::Transparent);
356
357     sf::Text segment_n;
358     segment_n.setFont(m_font);
359     segment_n.setFillColor(sf::Color::Black);
360     segment_n.setCharacterSize(14);
361
362     sf::VertexArray line(sf::Lines, 2);
363     line[0].color = sf::Color::Blue;
364     line[1].color = sf::Color::Blue;
365
366     if(debug){
367         int i = 0;

```

```

369     for(RoadSegment * segment : Road::shared().segments()){
370         for(RoadNode * node : segment->get_nodes()){
371             circle.setPosition(sf::Vector2f(node->get_x()*2-4,node->get_y()*2-4));
372             line[0].position = sf::Vector2f(node->get_x()*2,node->get_y()*2);
373             for(RoadNode * connected_node : node->get_nodes_from_me()){
374                 line[1].position = sf::Vector2f(connected_node->get_x()*2,connected_node->get_y()*2);
375                 target.draw(line , states);
376             }
377             target.draw(circle , states);
378
379         }
380         segment_n.setString(std::to_string(i));
381         segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4,segment->get_y()*2+4));
382         target.draw(segment_n , states);
383         i++;
384     }
385 }
386
387 // one rectangle is all we need :)
388 sf::RectangleShape rectangle;
389 rectangle.setSize(sf::Vector2f(9.4,3.4));
390 //rectangle.setFill(sf::Color::Green);
391 rectangle.setOutlineColor(sf::Color::Black);
392 rectangle.setOutlineThickness(2.0f);
393
394 //std::cout << "start drawing\n";
395 for(Car * car : m_cars){
396     //std::cout << "drawing" << car << std::endl;
397     if(car != nullptr){
398         rectangle.setPosition(car->x_pos()*2,car->y_pos()*2);
399         rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
400         unsigned int colval = (unsigned int)std::min(255.0f*(car->speed()/car->target_speed()),255.0f)
401 ;
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()
420 *2-4));
421             target.draw(circle , states);
422             circle.setOutlineColor(sf::Color::Green);
423             circle.setPosition(sf::Vector2f(car->heading_to_node->get_x()*2-4,car->heading_to_node->
424 get_y()*2-4));
425             target.draw(circle , states);
426         }
427     }
428 }
429
430 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
431 /// Modifies @param text by inserting information about Traffic ,
432 /// average speeds and frame rate among other things.
433
434 void Traffic::get_info(sf::Text & text , sf::Time &elapsed) {
435     //TODO: SOME BUG HERE.
436
437     float fps = 1.0f/elapsed.asSeconds();
438     unsigned long amount_of_cars = n_of_cars();
439     float flow = get_avg_flow();
440     std::vector<float> spe = get_avg_speeds();
441     std::string speedy = std::to_string(fps).substr(0,2) +
442         " fps , ncars: " + std::to_string(amount_of_cars) + "\n"

```

```

443         + "avg-flow: " + std::to_string(flow).substr(0,4) + "\n"
         + "avg-speed: " + std::to_string(spe[0]).substr(0,5) + "km/h\n"
         + "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 "../headers/unittests.h"
6 #include "../headers/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;
        m_traffic->despawn_car(car);
44         m_mutex->unlock();
        std::cout << car << std::endl;
46     }
    std::cout << "Car despawn tests passed\n";
48 }

50 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";

```

```

60     while(m_traffic->n_of_cars() != 0) {
        usleep((useconds_t)(1000000.0/(fps*multiplier)));
        m_traffic->update(1.0f/(float)fps);
        m_traffic->despawn_cars();
    }
64 }

66 void Tests::placement_test_2() {
    std::cout << "Starting placement tests 2\n";
68     std::vector<RoadSegment*> segments = Road::shared().segments();
    int i = 0;
70
    for(RoadSegment * seg : segments){
72         usleep(100000);
        std::cout<< "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ", " << seg << std::
endl;
74         std::cout << "next segment" << seg->next_segment() << std::endl;
        std::vector<RoadNode*> nodes = seg->get_nodes();
76         for(RoadNode * node : nodes){
            std::vector<RoadNode*> connections = node->get_nodes_from_me();
78             std::cout << "node" << node << " has connections:" << std::endl;
            for(RoadNode * pointy : connections){
80                 std::cout << pointy << std::endl;
            }
82             m_traffic->force_place_car(seg,node,1,1,0.1);
            std::cout << "placed car" << std::endl;
84         }
        i++;
86
    }
88     m_traffic->despawn_all_cars();
    std::cout << "Placement tests 2 passed\n";
90 }

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

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

100
    delete_cars_test();
102     //m_traffic.despawn_all_cars();
    std::cout << "Placement tests 3 passed\n";
104 }

106 // do all tests
108 void Tests::run_all_tests() {
    usleep(2000000);
110     placement_test();
    delete_cars_test();
112     run_one_car();
    placement_test_2();
114     placement_test_3();

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

118 Tests::Tests(Traffic *& traffic, sf::Mutex *& mutex) {
120     m_traffic = traffic;
    m_mutex = mutex;
122 }

```

../highway/cppfiles/unittests.cpp

B.9 util.cpp

```

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

```

```

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 < M_PI*0.45;
31     }
32     else{
33         return false;
34     }
35 }
36
37 ///////////////////////////////////////////////////////////////////
38 /// Returns true if @param a will cross paths with @param b, else false.
39 /// NOTE: @param a MUST be behind @param b.
40
41 bool Util::will_car_paths_cross(Car *a, Car *b) {
42     //simulate car a driving straight ahead.
43     RoadSegment * inspecting_segment = a->get_segment();
44     //RoadNode * node_0 = a->current_node;
45     RoadNode * node_1 = a->heading_to_node;
46
47     //int node_0_int = inspecting_segment->get_lane_number(node_0);
48     int node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
49
50     while(!node_1->get_nodes_from_me().empty()){
51         for(Car * car : inspecting_segment->m_cars){
52             if(car == b){
53                 // place logic for evaluating if we cross cars here.
54                 // heading to same node, else return false
55                 return node_1 == b->heading_to_node;
56             }
57         }
58
59         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().size()){
65             node_1 = node_1->get_nodes_from_me()[node_1_int];
66         }
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->merge){
75             node_1 = node_1->get_nodes_from_me()[std::max(node_1_int-1,0)];
76         }
77
78         node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
79     }
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] && delta_speed
91                 < 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 /*
124 float Util::distance_to_proj_point(const float theta, const float x, const float y){
125     float x_hat,y_hat;
126     x_hat = cos(theta);
127     y_hat = -sin(theta);
128     float proj_x = (x*x_hat+y*y_hat)*x_hat;
129     float proj_y = (x*x_hat+y*y_hat)*y_hat;
130     float dist = sqrt(abs(pow(proj_x,2.0f))+abs(pow(proj_y,2.0f)));
131
132     return dist;
133 }
134 */
135
136 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
137 /// Returns distance between @param a and @param b.
138
139 float Util::distance_to_car(Car * a, Car * b){
140     if(a == nullptr || b == nullptr){
141         throw std::invalid_argument("Can't calculate distance if cars are nullptrs");
142     }
143
144     float delta_x = a->x_pos()-b->x_pos();
145     float delta_y = b->y_pos()-a->y_pos();
146
147     return sqrt(abs(pow(delta_x,2.0f))+abs(pow(delta_y,2.0f)));
148 }
149
150 /*
151 Car * Util::find_closest_radius(std::vector<Car> &cars, const float x, const float y){
152     Car * answer = nullptr;
153
154     for(Car * car : cars){
155         float dist = Util::distance_to_car(car, new Car(x,y));
156         if(answer == nullptr || dist < Util::distance_to_car(answer, new Car(x,y))){
157             answer = car;
158         }
159     }
160     return answer;
161 }
162
163 */

```

```

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

```

../highway/cppfiles/util.cpp