

# Final Project, SI1336

Carl Schiller, 9705266436

January 20, 2019

## Abstract

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Method</b>	<b>1</b>
<b>3</b>	<b>Result</b>	<b>1</b>
<b>4</b>	<b>Discussion</b>	<b>1</b>
<b>A</b>	<b>Highway</b>	<b>2</b>
A.1	Header files . . . . .	2
A.1.1	traffic.h . . . . .	2
A.1.2	window.h . . . . .	4
A.1.3	unittests.h . . . . .	5
A.2	Source files . . . . .	5
A.2.1	traffic.cpp . . . . .	5
A.2.2	window.cpp . . . . .	21
A.2.3	unittests.cpp . . . . .	21
A.2.4	main.cpp . . . . .	23

## 1 Introduction

## 2 Method

## 3 Result

## 4 Discussion

# A Highway

## A.1 Header files

### A.1.1 traffic.h

```
1 //
2 // Created by Carl Schiller on 2018-12-19.
3 //
4 #include <random>
5 #include <vector>
6 #include "SFML/Graphics.hpp"
7
8 #ifndef HIGHWAY_TRAFFIC_H
9 #define HIGHWAY_TRAFFIC_H
10
11
12
13 class RoadSegment;
14
15 class Car;
16
17 class RoadNode{
18 private:
19     float m_x, m_y;
20     std::vector<RoadNode*> m_connecting_nodes; // raw pointers, no ownership
21     RoadSegment* m_is_child_of; // raw pointer, no ownership
22 public:
23     RoadNode();
24     ~RoadNode();
25     RoadNode(float x, float y, RoadSegment * segment);
26
27     void set_pointer(RoadNode*);
28     RoadSegment* get_parent_segment();
29     RoadNode * get_next_node(int lane);
30     std::vector<RoadNode*> & get_connections();
31     float get_x();
32     float get_y();
33     float get_theta(RoadNode*);
34 };
35
36
37 class RoadSegment{
38 private:
39     const float m_x, m_y;
40     float m_theta;
41     const int m_n_lanes;
42
43     constexpr static float MLANE_WIDTH = 4.0f;
44
45     std::vector<RoadNode*> m_nodes; // OWNERSHIP
46     RoadSegment * m_next_segment; // raw pointer, no ownership
47 public:
48     RoadSegment() = delete;
49     RoadSegment(float x, float y, RoadSegment * next_segment, int lanes);
50     RoadSegment(float x, float y, float theta, int lanes);
51     RoadSegment(float x, float y, int lanes, bool merge);
52     ~RoadSegment(); // rule of three
53     RoadSegment(const RoadSegment&) = delete; // rule of three
54     RoadSegment& operator=(const RoadSegment& rhs) = delete; // rule of three
55
56     bool merge;
57     std::vector<Car*> m_cars; // raw pointer, no ownership
58
59     RoadNode * get_node_pointer(int n);
60     std::vector<RoadNode *> get_nodes();
61     void append_car(Car*);
62     void remove_car(Car*);
63     RoadSegment * next_segment();
64     float get_theta();
65     const float get_x() const;
66     const float get_y() const;
67
68     int get_lane_number(RoadNode *);
69     const int get_total_amount_of_lanes() const;
```

```

71 void set_theta(float theta);
72 void set_next_road_segment(RoadSegment*);
73 void calculate_theta();
74 void calculate_and_populate_nodes();
75 void set_all_node_pointers_to_next_segment();
76 void set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *);
77 };
78
79 class Road{
80 private:
81     std::vector<RoadSegment*> m_segments; // OWNERSHIP
82     std::vector<RoadSegment*> m_spawn_positions; // raw pointers
83     std::vector<RoadSegment*> m_despawn_positions; // raw pointers
84
85     const std::string M_FILENAME;
86 private:
87     Road();
88     ~Road();
89 public:
90     static Road &shared() {static Road road; return road;}
91
92     Road(const Road& copy) = delete;
93     Road& operator=(const Road& rhs) = delete;
94
95     bool load_road();
96     std::vector<RoadSegment*> & spawn_positions();
97     std::vector<RoadSegment*> & despawn_positions();
98     std::vector<RoadSegment*> & segments();
99 };
100
101 /**
102  * Car class
103  * =====
104  * Private:
105  * position, width of car, and velocities are stored.
106  *
107  * Public:
108  * .update_pos(float delta_t): updates position by updating position.
109  * .accelerate(float delta_v): accelerates car.
110  * .steer(float delta_theta): change direction of speed.
111  * .x_pos(): return reference to x_pos.
112  * .y_pos(): -||- y_pos.
113  */
114
115 class Car{
116 private:
117     float m_dist_to_next_node;
118     float m_speed;
119     float m_theta; // radians
120
121     float m_aggressiveness; // how fast to accelerate;
122     float m_target_speed;
123     bool m_breaking;
124
125 public:
126     Car();
127     ~Car();
128     Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float aggressiveness);
129     Car(RoadSegment * spawn_point, RoadNode * lane, float vel, float target_speed, float aggressiveness);
130
131     // all are raw pointers
132     RoadSegment * current_segment;
133     RoadNode * current_node;
134     RoadNode * heading_to_node;
135     Car * overtake_this_car;
136     bool is_getting_overtaken;
137     //void remove_pointer(Car * car);
138
139     void update_pos(float delta_t);
140     void accelerate(float delta_t);
141     void avoid_collision(float delta_t);
142     Car * find_closest_car();
143
144     float x_pos();
145     float y_pos();

```

```

147     float & speed();
148     float & target_speed();
149     float & theta();
150
151     RoadSegment * get_segment();
152 };
153
154 class Util{
155 public:
156     static std::vector<std::string> split_string_by_delimiter(const std::string & str, const char delim);
157     static bool is_car_behind(Car * a, Car * b);
158     static bool will_car_paths_cross(Car *a, Car*b);
159     static bool is_cars_in_same_lane(Car*a,Car*b);
160     static bool merge_helper(Car*a, int merge_to_lane);
161     static float distance_to_line(float theta, float x, float y);
162     static float distance_to_proj_point(float theta, float x, float y);
163     static float distance_to_car(Car * a, Car * b);
164     static Car * find_closest_radius(std::vector<Car> &cars, float x, float y);
165     static float get_min_angle(float angl1, float ang2);
166     static float distance(float x1, float x2, float y1, float y2);
167 };
168
169 class Traffic : public sf::Drawable, public sf::Transformable{
170 private:
171     std::vector<Car*> m_cars;
172     std::mt19937 & my_engine();
173     sf::Font m_font;
174
175     //void update_speed(int i, float & elapsed_time);
176     //float get_theta(float xpos, float ypos, float speed, float current_theta, bool & lane_switch);
177 public:
178     Traffic();
179     ~Traffic();
180     Traffic(const Traffic&); // rule of three
181     Traffic& operator=(const Traffic&); // rule of three
182
183     unsigned long n_of_cars();
184     void spawn_cars(double & spawn_counter, float elapsed, double & threshold);
185     void despawn_cars();
186     void despawn_all_cars();
187     void despawn_car(Car*& car);
188     void force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, float aggro);
189
190     void update(float elapsed_time);
191     std::vector<Car*> get_car_copies() const;
192     float get_avg_flow();
193
194 private:
195     virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
196 public:
197     void get_info(sf::Text & text, sf::Time &elapsed);
198     double m_multiplier;
199 };
200
201 struct car_deleter{
202     void operator()(Car*& car);
203 };
204
205 #endif //HIGHWAY_TRAFFIC.H

```

../highway/traffic.h

## A.1.2 window.h

```

2 //
3 // Created by Carl Schiller on 2018-12-19.
4 //
5
6 #include <vector>
7 #include "SFML/Graphics.hpp"
8 #include "traffic.h"

```

```

10 #ifndef HIGHWAY.WINDOW.H
12 #define HIGHWAY.WINDOW.H
14
16 class Simulation{
18 private:
20     sf::Mutex * m_mutex;
22     Traffic * m_traffic;
24     bool * m_exit_bool;
26     const int M_SIM_SPEED;
28     const int MFRAMERATE;
30 public:
32     Simulation() = delete;
34     Simulation(Traffic *& traffic, sf::Mutex *& mutex, int sim_speed, int m_framerate, bool *& exitbool);
36
38     void update();
40 };
42
44 #endif //HIGHWAY.WINDOW.H

```

../highway/window.h

### A.1.3 unittests.h

```

1 //
2 // Created by Carl Schiller on 2019-01-16.
3 //
4
5 #include "traffic.h"
6 #include "SFML/Graphics.hpp"
7 #ifndef HIGHWAY_UNITTESTS.H
8 #define HIGHWAY_UNITTESTS.H
9
10 class Tests{
12 private:
14     Traffic * m_traffic;
16     sf::Mutex * m_mutex;
18     void placement_test();
20     void delete_cars_test();
22     void run_one_car();
24     void placement_test_2();
26     void placement_test_3();
28 public:
30     Tests() = delete;
32     Tests(Traffic *& traffic, sf::Mutex *& mutex);
34
36     void run_all_tests();
38 };
40
42 #endif //HIGHWAY_UNITTESTS.H

```

../highway/unittests.h

## A.2 Source files

### A.2.1 traffic.cpp

```

1 //
2 // Created by Carl Schiller on 2018-12-19.
3 //
4
5 #include "traffic.h"
6 #include <cmath>
7 #include <fstream>
8 #include <sstream>
9 #include <iostream>
10 #include <map>
11 #include <random>
12 #include <vector>
13 #include <list>
14
15 Car::Car() = default;

```

```

16 Car::Car(RoadSegment *spawn_point, int lane, float vel, float target_speed, float aggressivness):
17     m_speed(vel), m_target_speed(target_speed), m_aggressiveness(aggressivness)
18 {
19     current_segment = spawn_point;
20     is_getting_overtaken = false;
21     overtake_this_car = nullptr;
22
23     current_segment->append_car(this);
24     current_node = current_segment->get_node_pointer(lane);
25
26     if(!current_node->get_connections().empty()){
27         heading_to_node = current_node->get_next_node(lane);
28
29         m_dist_to_next_node = Util::distance(current_node->get_x(), heading_to_node->get_x(), current_node->
30 get_y(), heading_to_node->get_y());
31
32         m_theta = current_node->get_theta(heading_to_node);
33     }else{
34         //std::cout << "aa\n";
35         heading_to_node = nullptr;
36         m_theta = 0;
37         m_dist_to_next_node = 0;
38     }
39
40     m_breaking = false;
41 }
42
43 Car::Car(RoadSegment *spawn_point, RoadNode *lane, float vel, float target_speed, float agressivness) :
44 m_speed(vel), m_target_speed(target_speed), m_aggressiveness(agressivness)
45 {
46     current_segment = spawn_point;
47
48     overtake_this_car = nullptr;
49     is_getting_overtaken = false;
50
51     current_segment->append_car(this);
52     current_node = lane;
53
54     if(!current_node->get_connections().empty() || current_segment->next_segment() != nullptr){
55         heading_to_node = current_node->get_next_node(0);
56
57         m_dist_to_next_node = Util::distance(current_node->get_x(), heading_to_node->get_x(), current_node->
58 get_y(), heading_to_node->get_y());
59
60         m_theta = current_node->get_theta(heading_to_node);
61     }
62     else{
63         //std::cout << "aa\n";
64         heading_to_node = nullptr;
65         m_theta = 0;
66         m_dist_to_next_node = 0;
67     }
68
69     m_breaking = false;
70 }
71
72 Car::~Car(){
73     if(this->current_segment != nullptr){
74         this->current_segment->remove_car(this); // remove this pointer shit
75     }
76
77     overtake_this_car = nullptr;
78     current_segment = nullptr;
79     heading_to_node = nullptr;
80     current_node = nullptr;
81 }
82
83 void Car::update_pos(float delta_t) {
84     m_dist_to_next_node -= m_speed*delta_t;
85     // if we are at a new node.
86
87     if(m_dist_to_next_node < 0){
88         current_segment->remove_car(this); // remove car from this segment

```

```

90     current_segment = heading_to_node->get_parent_segment(); // set new segment
91     if(current_segment != nullptr){
92         current_segment->append_car(this); // add car to new segment
93     }
94     current_node = heading_to_node; // set new current node as previous one.
95
96     //TODO: place logic for choosing next node
97     std::vector<RoadNode*> connections = current_node->get_connections();
98
99     if(!connections.empty()){
100         // check if we merge
101         int current_lane = current_segment->get_lane_number(current_node);
102
103         if(current_segment->merge){
104             if(current_lane == 0 && connections[0]->get_parent_segment()->get_total_amount_of_lanes()
105 != 2){
106                 if(!Util::merge_helper(this,1)){
107                     heading_to_node = connections[1];
108                 }
109                 else{
110                     heading_to_node = connections[0];
111                 }
112             }
113             else if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
114                 current_lane = std::max(current_lane-1,0);
115                 heading_to_node = connections[current_lane];
116             }
117             else{
118                 heading_to_node = connections[current_lane];
119             }
120         }
121         // if we are in start section
122         else if(current_segment->get_total_amount_of_lanes() == 3){
123             if(connections.size() == 1){
124                 heading_to_node = connections[0];
125             }
126             else{
127                 heading_to_node = connections[current_lane];
128             }
129         }
130         // if we are in middle section
131         else if(current_segment->get_total_amount_of_lanes() == 2){
132             // normal way
133             if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
134                 //see if we want to overtake car.
135                 Car * closest_car = find_closest_car();
136
137                 if(closest_car != nullptr && this->overtake_this_car == nullptr){
138                     float delta_speed = closest_car->speed()-this->speed();
139                     float delta_distance = Util::distance_to_car(this,closest_car);
140                     if(delta_distance/delta_speed > -5 && delta_speed < 2){
141                         this->overtake_this_car = closest_car;
142                     }
143                 }
144
145                 if(this->overtake_this_car != nullptr){
146                     if(current_lane != 1){
147                         if(!Util::merge_helper(this,1)){
148                             heading_to_node = connections[1];
149                         }
150                         else{
151                             heading_to_node = connections[current_lane];
152                         }
153                     }
154                     else{
155                         heading_to_node = connections[current_lane];
156                     }
157                     /*
158                     else{
159                         if((!Util::is_car_behind(this,this->overtake_this_car) && Util::
160 distance_to_car(this,this->overtake_this_car) > 10) || Util::distance_to_car(this,this->
161 overtake_this_car) > 40){
162                             for(int i = 0; i < overtake_this_car->want_to_overtake_me.size(); i++){
163                                 if(this == overtake_this_car->want_to_overtake_me[i]){
164                                     overtake_this_car->want_to_overtake_me[i] = nullptr;
165                                 }
166                             }
167                         }
168                     }

```

```

164         }
165         std::vector<Car*>::iterator new_end = std::remove(overtake_this_car->
want_to_overtake_me.begin(),overtake_this_car->want_to_overtake_me.end(),static_cast<Car*>(nullptr));
166         overtake_this_car->want_to_overtake_me.erase(new_end,overtake_this_car->
want_to_overtake_me.end());
167
168         this->overtake_this_car = nullptr;
169     }
170
171     heading_to_node = connections[current_lane];
172 }
173     */
174 // merge back if overtake this car is nullptr.
175 else{
176     if(!Util::merge_helper(this,0)){
177         heading_to_node = connections[0];
178     }
179     else{
180         heading_to_node = connections[current_lane];
181     }
182 }
183 }
184 else{
185     heading_to_node = connections[0];
186 }
187
188 }
189 else if(current_segment->get_total_amount_of_lanes() == 1){
190     heading_to_node = connections[0];
191 }
192
193     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());
194     m_theta = current_node->get_theta(heading_to_node);
195 }
196 }
197 }
198 }
199
200 void Car::accelerate(float elapsed){
201     float target = m_target_speed;
202     float d_vel; // proportional control.
203
204     if(m_speed < target*0.75){
205         d_vel = m_aggressiveness*elapsed;
206     }
207     else{
208         d_vel = m_aggressiveness*(target-m_speed)*4*elapsed;
209     }
210
211     m_speed += d_vel;
212 }
213
214 void Car::avoid_collision(float delta_t) {
215     float min_distance = 8.0f; // for car distance.
216     float ideal = min_distance+min_distance*(m_speed/20.f);
217     float detection_distance = m_speed*4.0f;
218     //std::cout << "boop1\n";
219     Car * closest_car = find_closest_car();
220     //std::cout << "boop2\n";
221     float radius_to_car = 1000;
222     float delta_speed = 0;
223
224     if(closest_car != nullptr) {
225         radius_to_car = Util::distance_to_car(this, closest_car);
226         delta_speed = closest_car->speed() - this->speed();
227
228         if (radius_to_car < ideal) {
229             m_speed -= std::min(abs(delta_speed)*delta_t+(ideal-radius_to_car)*0.5f, 10.0f * delta_t);
230         }
231         else if(radius_to_car < detection_distance && delta_speed < 0){
232             m_speed -= std::min(
233                 abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
m_aggressiveness * 0.02f,
234                 10.0f * delta_t);

```



```

236         }
237         else {
238             accelerate(delta_t);
239         }
240     }
241     else{
242         accelerate(delta_t);
243     }
244     //std::cout << "boop3\n";
245     if(m_speed < 0){
246         m_speed = 0;
247     }
248 }
249
250 Car* Car::find_closest_car() {
251     const float search_radius = 100;
252     RoadSegment* origin = this->current_segment;
253     std::map<RoadSegment*,bool> visited;
254     std::list<RoadNode*> queue;
255
256     for(RoadNode * node : (this->current_segment->get_nodes())){
257         queue.push_front(node);
258     }
259
260     Car* answer = nullptr;
261     float best_radius = 1000;
262     while(!queue.empty()){
263         RoadNode * next_node = queue.back(); // get last element
264         queue.pop_back(); // remove element
265         RoadSegment * next_segment = nullptr;
266         if(next_node != nullptr) {
267             next_segment = next_node->get_parent_segment();
268         }
269
270         if(next_segment != nullptr){
271             if(!visited[next_segment] && Util::distance(origin->get_x(),next_segment->get_x(),origin->
272             get_y(),next_segment->get_y()) < search_radius){
273                 visited[next_segment] = true;
274                 for(Car * car : next_segment->m_cars){
275                     if(this != car){
276                         float radius = Util::distance_to_car(this,car);
277                         if(Util::is_car_behind(this,car) && radius < best_radius){
278                             //std::cout << "a\n";
279                             if(Util::will_car_paths_cross(this,car)){
280                                 best_radius = radius;
281                                 answer = car;
282                             }
283                             //std::cout << "b\n";
284                         }
285                     }
286                 }
287             }
288             // push in new nodes in front of list.
289             for(RoadNode * node : next_node->get_connections()){
290                 queue.push_front(node);
291             }
292         }
293     }
294 }
295 return answer;
296 }
297
298 float Car::x_pos() {
299     float x_position;
300     if(heading_to_node != nullptr){
301         x_position = heading_to_node->get_x()-m_dist_to_next_node*cos(m_theta);
302     }
303     else{
304         x_position = current_node->get_x();
305     }
306     return x_position;
307 }
308 }

```

```

310 float Car::y-pos() {
311     float y_position;
312     if(heading_to_node != nullptr){
313         y_position = heading_to_node->get_y()+m_dist_to_next_node*sin(m_theta);
314     }
315     else{
316         y_position = current_node->get_y();
317     }
318
319     return y_position;
320 }
321
322 float & Car::speed() {
323     return m_speed;
324 }
325
326 float & Car::target_speed() {
327     return m_target_speed;
328 }
329
330 float & Car::theta() {
331     return m_theta;
332 }
333
334 RoadSegment* Car::get_segment() {
335     return current_segment;
336 }
337
338
339
340 RoadNode::RoadNode() = default;
341
342 RoadNode::~~RoadNode() = default;
343
344 RoadNode::RoadNode(float x, float y, RoadSegment * segment) {
345     m_x = x;
346     m_y = y;
347     m_is_child_of = segment;
348 }
349
350 void RoadNode::set_pointer(RoadNode * next_node) {
351     m_connecting_nodes.push_back(next_node);
352 }
353
354 RoadSegment* RoadNode::get_parent_segment() {
355     return m_is_child_of;
356 }
357
358 std::vector<RoadNode*> & RoadNode::get_connections() {
359     return m_connecting_nodes;
360 }
361
362 float RoadNode::get_x() {
363     return m_x;
364 }
365
366 float RoadNode::get_y() {
367     return m_y;
368 }
369
370 float RoadNode::get_theta(RoadNode* node) {
371     for(RoadNode * road_node : m_connecting_nodes){
372         if(node == road_node){
373             return atan2(m_y-node->m_y, node->m_x-m_x);
374         }
375     }
376     throw std::invalid_argument("Node given is not a connecting node");
377 }
378
379 RoadNode* RoadNode::get_next_node(int lane) {
380     return m_connecting_nodes[lane];
381 }
382
383 RoadSegment::~~RoadSegment(){
384     for(RoadNode * elem : m_nodes){

```

```

386         delete elem;
387     }
388     m_nodes.clear();
389 }
390 RoadSegment::RoadSegment(float x, float y, RoadSegment * next_segment, int lanes):
391     m_x(x), m_y(y), m_n_lanes(lanes)
392 {
393     m_next_segment = next_segment;
394     m_theta = atan2(m_y-m.next_segment->m_y, m.next_segment->m_x-m_x);
395
396     m_nodes.reserve(m_n_lanes);
397
398     calculate_and_populate_nodes();
399 }
400
401 RoadSegment::RoadSegment(float x, float y, float theta, int lanes) :
402     m_x(x), m_y(y), m_theta(theta), m_n_lanes(lanes)
403 {
404     m_next_segment = nullptr;
405     m_nodes.reserve(lanes);
406
407     calculate_and_populate_nodes();
408 }
409
410 RoadSegment::RoadSegment(float x, float y, int lanes, bool mer):
411     m_x(x), m_y(y), merge(mer), m_n_lanes(lanes)
412 {
413     merge = mer;
414     m_next_segment = nullptr;
415     m_nodes.reserve(m_n_lanes);
416
417     // can't set nodes if we don't have a theta.
418 }
419
420 float RoadSegment::get_theta() {
421     return m_theta;
422 }
423
424 const float RoadSegment::get_x() const {
425     return m_x;
426 }
427
428 const float RoadSegment::get_y() const {
429     return m_y;
430 }
431
432 int RoadSegment::get_lane_number(RoadNode * node) {
433     for(int i = 0; i < m_n_lanes; i++){
434         if(node == m_nodes[i]){
435             return i;
436         }
437     }
438     throw std::invalid_argument("Node is not in this segment");
439 }
440
441 void RoadSegment::append_car(Car * car) {
442     m_cars.push_back(car);
443 }
444
445 void RoadSegment::remove_car(Car * car) {
446     unsigned long size = m_cars.size();
447     bool found = false;
448     for(int i = 0; i < size; i++){
449         if(car == m_cars[i]){
450             m_cars[i] = nullptr;
451             found = true;
452         }
453     }
454     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(), static_cast<Car*>(
455         nullptr));
456     m_cars.erase(new_end, m_cars.end());
457
458     if(!found){
459         //throw std::invalid_argument("Car is not in this segment.");
460     }

```

```

}
462
void RoadSegment::set_theta(float theta) {
464     m_theta = theta;
}
466
void RoadSegment::calculate_and_populate_nodes() {
468     // calculates placement of nodes.
    float total_length = MLANE_WIDTH*(m_n_lanes-1);
470     float current_length = -total_length/2.0f;

    for(int i = 0; i < m_n_lanes; i++){
472         float x_pos = m_x+current_length*cos(m_theta+(float)M_PI*0.5f);
474         float y_pos = m_y-current_length*sin(m_theta+(float)M_PI*0.5f);
        m_nodes.push_back(new RoadNode(x_pos, y_pos, this));
476         current_length += MLANE_WIDTH;
    }
478 }

void RoadSegment::set_next_road_segment(RoadSegment * next_segment) {
480     m_next_segment = next_segment;
482 }

void RoadSegment::calculate_theta() {
484     m_theta = atan2(m_y-m_next_segment->m_y, m_next_segment->m_x-m_x);
486 }

RoadNode* RoadSegment::get_node_pointer(int n) {
488     return m_nodes[n];
490 }

std::vector<RoadNode*> RoadSegment::get_nodes() {
492     return m_nodes;
494 }

RoadSegment* RoadSegment::next_segment() {
496     return m_next_segment;
498 }

void RoadSegment::set_all_node_pointers_to_next_segment() {
500     for(RoadNode * node: m_nodes){
502         for(int i = 0; i < m_next_segment->m_n_lanes; i++){
            node->set_pointer(m_next_segment->get_node_pointer(i));
504         }
    }
506 }

void RoadSegment::set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *next_segment) {
508     RoadNode * pointy = next_segment->get_node_pointer(to_node_n);
510     m_nodes[from_node_n]->set_pointer(pointy);
}

512
const int RoadSegment::get_total_amount_of_lanes() const {
514     return m_n_lanes;
}
516

Road::Road() :
518     MFILENAME("../road.txt")
{
520     if(!load_road()){
        std::cout << "Error in loading road.\n";
522     };
}

524
Road::~~Road() {
526     for(RoadSegment * seg : m_segments){
        delete seg;
528     }
    m_segments.clear();
530 }

532 bool Road::load_road() {
    bool loading = true;
534     std::ifstream stream;
    stream.open(MFILENAME);
536

```

```

538 std::vector<std::vector<std::string>> road_vector;
road_vector.reserve(100);

540 if(stream.is_open()){
    std::string line;
542     std::vector<std::string> tokens;
    while(std::getline(stream, line)){
544         tokens = Util::split_string_by_delimiter(line, ' ');
        if(tokens[0] != "#"){
546             road_vector.push_back(tokens);
        }
548     }
}
550 else{
    loading = false;
552 }

554 // load segments into memory.
556 for(std::vector<std::string> & vec : road_vector){
    if(vec.size() == 5){
558         if(vec[4] == "merge"){
            RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]),
560 true);
            m_segments.push_back(seg);
        }
562         else{
            RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]),
564 false);
            m_segments.push_back(seg);
        }
566     }
    else{
568         RoadSegment * seg = new RoadSegment(std::stof(vec[1]), std::stof(vec[2]), std::stoi(vec[3]),
570 false);
        m_segments.push_back(seg);
    }
572 }

574 // populate nodes.
576 for (int i = 0; i < m_segments.size(); ++i) {
    // populate nodes normally.
578     if(road_vector[i].size() == 4){
        m_segments[i]->set_next_road_segment(m_segments[i+1]);
580         m_segments[i]->calculate_theta();
        // calculate nodes based on theta.
582         m_segments[i]->calculate_and_populate_nodes();
    }
584 }
    else if(road_vector[i].size() == 5){
586         if(road_vector[i][4] == "false"){
            // take previous direction and populate nodes.
588             m_segments[i]->set_theta(m_segments[i-1]->get_theta());
            m_segments[i]->calculate_and_populate_nodes();
590             // but do not connect nodes to new ones.

            // make this a despawn segment
592             m_despawn_positions.push_back(m_segments[i]);
        }
594         else if(road_vector[i][4] == "true"){
            m_segments[i]->set_next_road_segment(m_segments[i+1]);
596             m_segments[i]->calculate_theta();
            // calculate nodes based on theta.
598             m_segments[i]->calculate_and_populate_nodes();

            // make this a spawn segment
600             m_spawn_positions.push_back(m_segments[i]);
        }
602     }
    else if(road_vector[i][4] == "merge"){
604         m_segments[i]->set_next_road_segment(m_segments[i+1]);
606         m_segments[i]->calculate_theta();
        // calculate nodes based on theta.
608         m_segments[i]->calculate_and_populate_nodes();
    }
}

```

```

610     }
612     // else we connect one by one.
614     else{
616         // take previous direction and populate nodes.
618         m_segments[i]->set_theta(m_segments[i-1]->get_theta());
620         // calculate nodes based on theta.
622         m_segments[i]->calculate_and_populate_nodes();
624     }
626 }
628
630 // connect nodes.
632 for (int i = 0; i < m_segments.size(); ++i) {
634     // do normal connection, ie connect all nodes.
636     if(road_vector[i].size() == 4){
638         m_segments[i]->set_all_node_pointers_to_next_segment();
640     }
642     else if(road_vector[i].size() == 5){
644         if(road_vector[i][4] == "false"){
646             // but do not connect nodes to new ones.
648         }
650         else if(road_vector[i][4] == "true"){
652             m_segments[i]->set_all_node_pointers_to_next_segment();
654         }
656         else if(road_vector[i][4] == "merge"){
658             m_segments[i]->set_all_node_pointers_to_next_segment();
660         }
662     }
664     // else we connect one by one.
666     else{
668         // manually connect nodes.
670         int amount_of_pointers = (int)road_vector[i].size()-4;
672         for(int j = 0; j < amount_of_pointers/3; j++){
674             int current_pos = 4+j*3;
676             RoadSegment * next_segment = m_segments[std::stoi(road_vector[i][current_pos+2])];
678             m_segments[i]->set_node_pointer_to_node(std::stoi(road_vector[i][current_pos]),std::stoi(
680 road_vector[i][current_pos+1]),next_segment);
682         }
684     }
686 }
688 return loading;
690 }
692
694 std::vector<RoadSegment*>& Road::spawn_positions() {
696     return m_spawn_positions;
698 }
700
702 std::vector<RoadSegment*>& Road::despawn_positions() {
704     return m_despawn_positions;
706 }
708
710 std::vector<RoadSegment*>& Road::segments() {
712     return m_segments;
714 }
716
718 std::vector<std::string> Util::split_string_by_delimiter(const std::string &str, const char delim) {
720     std::stringstream ss(str);
722     std::string item;
724     std::vector<std::string> answer;
726     while(std::getline(ss,item,delim)){
728         answer.push_back(item);
730     }
732     return answer;
734 }
736
738 // if a is behind of b, return true. else false
740 bool Util::is_car_behind(Car * a, Car * b){
742     if(a!=b){
744         float theta_to_car_b = atan2(a->y_pos()-b->y_pos(),b->x_pos()-a->x_pos());
746         float theta_difference = get_min_angle(a->theta(),theta_to_car_b);
748         return theta_difference < MPI*0.45;
750     }
752     else{
754         return false;
756     }
758 }

```

```

686 }
688 //TODO: Bug here
688 // true if car paths cross
690 bool Util::will_car_paths_cross(Car *a, Car *b) {
690     //begin with drawing a straight line
692     std::list<RoadSegment*> segments;
692     std::list<RoadNode*> nodes;
694
694     segments.push_back(a->get_segment());
696     nodes.push_back(a->current_node);
696     // make sure this is not true
698     if(a->heading_to_node == nullptr){
698         return false;
700     }
700     nodes.push_back(a->heading_to_node);
702
702     float dist_between_segments = Util::distance(a->current_segment->get_x(), b->current_segment->get_x(),
704                                                a->current_segment->get_y(), b->current_segment->get_y());
704     bool found_b = false;
706
706     while(!found_b){
708         //std::cout << "a1\n";
708         for(Car * car : segments.back()->m_cars){
710             if(car == b){
710                 found_b = true;
712             }
712         }
714
714         if(!found_b){
716             if(nodes.back() == nullptr){
716                 return false;
718             }
718             //std::cout << nodes.back() << std::endl;
720             segments.push_back(nodes.back()->get_parent_segment());
722             int seg0_lane_n = segments.back()->get_total_amount_of_lanes();
722             int lane = segments.back()->get_lane_number(nodes.back());
722             std::vector<RoadNode*> node_choices = nodes.back()->get_connections();
724
724             // if seg0==seg1 we keep lane numbering.
726             if(node_choices.size() == seg0_lane_n){
726                 nodes.push_back(node_choices[lane]);
728             }
728             // if we only have one choice, stick to it
730             else if(node_choices.size() == 1){
730                 lane = 0;
732                 nodes.push_back(node_choices[lane]);
732             }
734             // last merge
736             else if(node_choices.size() == 2){
736                 lane = std::min(std::max(lane-1,0),0);
736                 nodes.push_back(node_choices[lane]);
738             }
738             else if(node_choices.empty()){
740                 return false;
740             }
742         }
742     }
744     //std::cout << "hej3\n";
744     float delta_dist = dist_between_segments - distance(b->current_segment->get_x(), segments.back()->
746                                                         b->current_segment->get_y(), segments.back()->
746                                                         get_y());
746     if(delta_dist < 0){
748         return false;
748     }
750 }
750
752 //std::cout << "hej4\n";
754 if(nodes.back() == b->heading_to_node){
754     return true;
756 }
758 nodes.pop_back();

```

```

760 // redo it.
761 if(nodes.back() == b->current_node){
762     return true;
763 }
764 return false;
765 }
766 bool Util::merge_helper(Car *a, int merge_to_lane) {
767     RoadSegment * seg = a->current_segment;
768     for(Car * car : seg->m_cars){
769         if(car != a){
770             float delta_speed = a->speed()-car->speed();
771             if(car->heading_to_node == a->current_node->get_connections()[merge_to_lane] && delta_speed <
772 0){
773                 return true;
774             }
775         }
776     }
777     return false;
778 }
779 // this works only if a's heading to is b's current segment
780 bool Util::is_cars_in_same_lane(Car *a, Car *b) {
781     return a->heading_to_node == b->current_node;
782 }
783
784 float Util::distance_to_line(const float theta, const float x, const float y){
785     float x_hat, y_hat;
786     x_hat = cos(theta);
787     y_hat = -sin(theta);
788
789     float proj_x = (x*x_hat+y*y_hat)*x_hat;
790     float proj_y = (x*x_hat+y*y_hat)*y_hat;
791     float dist = sqrt(abs(pow(x-proj_x, 2.0f))+abs(pow(y-proj_y, 2.0f)));
792
793     return dist;
794 }
795
796 float Util::distance_to_proj_point(const float theta, const float x, const float y){
797     float x_hat, y_hat;
798     x_hat = cos(theta);
799     y_hat = -sin(theta);
800     float proj_x = (x*x_hat+y*y_hat)*x_hat;
801     float proj_y = (x*x_hat+y*y_hat)*y_hat;
802     float dist = sqrt(abs(pow(proj_x, 2.0f))+abs(pow(proj_y, 2.0f)));
803
804     return dist;
805 }
806
807 float Util::distance_to_car(Car * a, Car * b){
808     float delta_x = a->x_pos()-b->x_pos();
809     float delta_y = b->y_pos()-a->y_pos();
810
811     return sqrt(abs(pow(delta_x, 2.0f))+abs(pow(delta_y, 2.0f)));
812 }
813
814 Car * Util::find_closest_radius(std::vector<Car> &cars, const float x, const float y){
815     Car * answer = nullptr;
816
817     float score = 100000;
818     for(Car & car : cars){
819         float distance = sqrt(abs(pow(car.x_pos()-x, 2.0f))+abs(pow(car.y_pos()-y, 2.0f)));
820         if(distance < score){
821             score = distance;
822             answer = &car;
823         }
824     }
825
826     return answer;
827 }
828
829 float Util::get_min_angle(const float angl, const float ang2){
830     float abs_diff = abs(ang1-ang2);
831     float score = std::min(2.0f*(float)M_PI-abs_diff, abs_diff);
832     return score;

```



```

834 }
836 float Util::distance(float x1, float x2, float y1, float y2) {
837     return sqrt(abs(pow(x1-x2,2.0f))+abs(pow(y1-y2,2.0f)));
838 }
840 Traffic::Traffic() {
841     if(!m_font.loadFromFile("/Library/Fonts/Arial.ttf")){
842         //crash
843     }
844 }
846 Traffic::Traffic(const Traffic &ref) :
847     m_multiplier(ref.m_multiplier)
848 {
849     // clear values if there are any.
850     for(Car * delete_this : m_cars){
851         delete delete_this;
852     }
853     m_cars.clear();
854
855     // reserve place for new pointers.
856     m_cars.reserve(ref.m_cars.size());
857
858     // copy values into new pointers
859     for(Car * car : ref.m_cars){
860         auto new_car_pointer = new Car;
861         *new_car_pointer = *car;
862         m_cars.push_back(new_car_pointer);
863     }
864
865     // values we copied are good, except the car pointers inside the car class.
866     std::map<int, Car*> overtake_this_car;
867     std::map<Car*, int> labeling;
868     for(int i = 0; i < m_cars.size(); i++){
869         overtake_this_car[i] = ref.m_cars[i]->overtake_this_car;
870         labeling[ref.m_cars[i]] = i;
871         m_cars[i]->overtake_this_car = nullptr; // clear copied pointers
872         //m_cars[i]->want_to_overtake_me.clear(); // clear copied pointers
873     }
874     std::map<int, int> from_to;
875     for(int i = 0; i < m_cars.size(); i++){
876         if(overtake_this_car[i] != nullptr){
877             from_to[i] = labeling[overtake_this_car[i]];
878         }
879     }
880
881     for(auto it : from_to){
882         m_cars[it.first]->overtake_this_car = m_cars[it.second];
883         //m_cars[it.second]->want_to_overtake_me.push_back(m_cars[it.first]);
884     }
885 }
886 Traffic& Traffic::operator=(const Traffic & rhs) {
887     Traffic tmp(rhs);
888
889     std::swap(m_cars, tmp.m_cars);
890     std::swap(m_multiplier, tmp.m_multiplier);
891
892     return *this;
893 }
894
895 Traffic::~~Traffic() {
896     for(int i = 0; i < m_cars.size(); i++){
897         delete Traffic::m_cars[i];
898     }
899     Traffic::m_cars.clear();
900 }
902 unsigned long Traffic::n_of_cars(){
903     return m_cars.size();
904 }
906
907 std::mt19937& Traffic::my_engine() {
908     static std::mt19937 e(std::random_device{}());
909     return e;

```

```

910 }
912 void Traffic::spawn_cars(double & spawn_counter, float elapsed, double & threshold) {
913     spawn_counter += elapsed;
914     if(spawn_counter > threshold){
915         std::exponential_distribution<double> dis(1);
916         std::normal_distribution<float> aggro(3.0f,0.5f);
917         std::normal_distribution<float> sp(20.0,2.0);
918         std::uniform_real_distribution<float> lane(0.0f,1.0f);
919         std::uniform_real_distribution<float> spawn(0.0f,1.0f);
920
921         float speed = sp(my_engine());
922         float target = speed;
923         threshold = dis(my_engine());
924         float aggressiveness = aggro(my_engine());
925         spawn_counter = 0;
926         float start_lane = lane(my_engine());
927         float spawn_pos = spawn(my_engine());
928
929         std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
930         RoadSegment * seg;
931         Car * new_car;
932         if(spawn_pos < 1){
933             seg = segments[0];
934             if(start_lane < 0){
935                 new_car = new Car(seg,0,speed,target,aggressiveness);
936             }
937             else if(start_lane < 0.5){
938                 new_car = new Car(seg,1,speed,target,aggressiveness);
939             }
940             else{
941                 new_car = new Car(seg,2,speed,target,aggressiveness);
942             }
943         }
944         else{
945             seg = segments[1];
946             new_car = new Car(seg,0,speed,target,aggressiveness);
947         }
948
949         Car * closest_car_ahead = new_car->find_closest_car();
950
951         if(closest_car_ahead == nullptr && closest_car_ahead != new_car){
952             m_cars.push_back(new_car);
953         }
954         else{
955             float dist = Util::distance_to_car(new_car,closest_car_ahead);
956             if(dist < 10){
957                 delete new_car;
958             }
959             else if (dist < 150){
960                 new_car->speed() = closest_car_ahead->speed();
961                 m_cars.push_back(new_car);
962             }
963             else{
964                 m_cars.push_back(new_car);
965             }
966         }
967     }
968 }
970 void Traffic::despawn_car(Car *& car) {
971     unsigned long size = m_cars.size();
972     for(int i = 0; i < size; i++){
973         if(car == m_cars[i]){
974             //std::cout << "found " << car << ", " << m_cars[i] << std::endl;
975             delete m_cars[i];
976             m_cars[i] = nullptr;
977             //std::cout << car << std::endl;
978             m_cars.erase(m_cars.begin()+i);
979             car = nullptr;
980             //std::cout << "deleted\n";
981             break;
982         }
983     }
984 }

```

```

986 void Traffic::despawn_cars() {
987     //std::cout << "e\n";
988     std::for_each(m_cars.begin(), m_cars.end(), car_deleter());
989     //std::cout << "f\n";
990     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(), m_cars.end(), static_cast<Car*>(
991         nullptr));
992     m_cars.erase(new_end, m_cars.end());
993     //std::cout << "g\n";
994 }
995
996 void Traffic::despawn_all_cars() {
997     *this = Traffic();
998 }
999
1000 void Traffic::force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, float aggro) {
1001     Car * car = new Car(seg, node, vel, target, aggro);
1002     m_cars.push_back(car);
1003 }
1004
1005 void Traffic::update(float elapsed_time) {
1006     //std::cout << "updatin1\n";
1007     for(Car * car : m_cars){
1008         car->avoid_collision(elapsed_time);
1009     }
1010     //std::cout << "updatin2\n";
1011     for(Car * car : m_cars){
1012         car->update_pos(elapsed_time);
1013     }
1014     //std::cout << "updatin3\n";
1015 }
1016
1017 std::vector<Car*> Traffic::get_car_copies() const {
1018     return m_cars;
1019 }
1020
1021 float Traffic::get_avg_flow() {
1022     float flow = 0;
1023     float i = 0;
1024     for(Car * car : m_cars){
1025         i++;
1026         flow += car->speed()/car->target_speed();
1027     }
1028     if(m_cars.empty()){
1029         return 0;
1030     }
1031     else{
1032         return flow/i;
1033     }
1034 }
1035
1036 void Traffic::draw(sf::RenderTarget &target, sf::RenderStates states) const {
1037     // print debug info about node placements and stuff
1038     sf::CircleShape circle;
1039     circle.setRadius(4.0f);
1040     circle.setOutlineColor(sf::Color::Cyan);
1041     circle.setOutlineThickness(1.0f);
1042     circle.setFill(sf::Color::Transparent);
1043
1044     sf::Text segment_n;
1045     segment_n.setFont(m_font);
1046     segment_n.setFill(sf::Color::Black);
1047     segment_n.setCharacterSize(14);
1048
1049     sf::VertexArray line(sf::Lines, 2);
1050     line[0].color = sf::Color::Blue;
1051     line[1].color = sf::Color::Blue;
1052
1053     int i = 0;
1054
1055     for(RoadSegment * segment : Road::shared().segments()){
1056         for(RoadNode * node : segment->get_nodes()){
1057             circle.setPosition(sf::Vector2f(node->get_x()*2-4, node->get_y()*2-4));
1058             line[0].position = sf::Vector2f(node->get_x()*2, node->get_y()*2);
1059             for(RoadNode * connected_node : node->get_connections()){
1060                 line[1].position = sf::Vector2f(connected_node->get_x()*2, connected_node->get_y()*2);
1061                 target.draw(line, states);

```

```

1062         }
1063         target.draw(circle, states);
1064
1065     }
1066     segment_n.setString(std::to_string(i));
1067     segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4, segment->get_y()*2+4));
1068     target.draw(segment_n, states);
1069     i++;
1070 }
1071
1072 // one rectangle is all we need :)
1073 sf::RectangleShape rectangle;
1074 rectangle.setSize(sf::Vector2f(9.4, 3.4));
1075 rectangle.setFillColor(sf::Color::Green);
1076 rectangle.setOutlineColor(sf::Color::Black);
1077 rectangle.setOutlineThickness(2.0f);
1078
1079 //std::cout << "start drawing\n";
1080 for(Car * car : m_cars){
1081     //std::cout << "drawing" << car << std::endl;
1082     if(car != nullptr){
1083         rectangle.setPosition(car->x_pos()*2, car->y_pos()*2);
1084         rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
1085         sf::Uint8 colorspeed = static_cast<sf::Uint8>((unsigned int)std::round(255 * car->speed() /
1086 car->target_speed()));
1087         rectangle.setFillColor(sf::Color(255-colorspeed, colorspeed, 0, 255));
1088         target.draw(rectangle, states);
1089
1090         // this caused crash earlier
1091         if(car->heading_to_node!=nullptr){
1092             // print debug info about node placements and stuff
1093             sf::CircleShape circle;
1094
1095             circle.setRadius(4.0f);
1096             circle.setOutlineColor(sf::Color::Red);
1097             circle.setOutlineThickness(2.0f);
1098             circle.setFillColor(sf::Color::Transparent);
1099             circle.setPosition(sf::Vector2f(car->current_node->get_x()*2-4, car->current_node->get_y()
1100 *2-4));
1101             target.draw(circle, states);
1102             circle.setOutlineColor(sf::Color::Green);
1103             circle.setPosition(sf::Vector2f(car->heading_to_node->get_x()*2-4, car->heading_to_node->
1104 get_y()*2-4));
1105             target.draw(circle, states);
1106         }
1107     }
1108 }
1109
1110 void Traffic::get_info(sf::Text & text, sf::Time &elapsed) {
1111     //TODO: SOME BUG HERE.
1112
1113     float fps = 1.0f/elapsed.asSeconds();
1114     unsigned long amount_of_cars = n_of_cars();
1115     float flow = get_avg_flow();
1116     std::string speedy = std::to_string(fps).substr(0,2) +
1117         " fps, ncars: " + std::to_string(amount_of_cars) + "\n"
1118         + " avg_flow: " + std::to_string(flow).substr(0,4) + "\n"
1119         + " sim.multiplier: " + std::to_string(m_multiplier).substr(0,3) + "x";
1120     text.setString(speedy);
1121     text.setPosition(0,0);
1122     text.setFillColor(sf::Color::Black);
1123     text.setFont(m_font);
1124 }
1125
1126 void car_deleter::operator()(Car *&car) {
1127     for(RoadSegment * seg : Road::shared().despawn_positions()){
1128         if(car->get_segment() == seg){
1129             //std::cout << "deletin\n";
1130             //std::cout << car << "\n";
1131
1132             delete car;
1133             car = nullptr;
1134             //std::cout << "deletidn\n";
1135             break;

```

```

1134     }
1135 }
1136 }

```

../highway/traffic.cpp

## A.2.2 window.cpp

```

1 //
2 // Created by Carl Schiller on 2018-12-19.
3 //
4
5 #include <iostream>
6 #include "traffic.h"
7 #include "window.h"
8 #include <cmath>
9 #include <unistd.h>
10
11 Simulation::Simulation(Traffic *&traffic, sf::Mutex *&mutex, int sim_speed, int framerate, bool *&
    exit_bool):
12     MFRAMERATE(framerate),
13     MSIM_SPEED(sim_speed),
14     m_traffic(traffic),
15     m_mutex(mutex),
16     m_exit_bool(exit_bool)
17 {
18
19 }
20
21 void Simulation::update() {
22     sf::Clock clock;
23     sf::Time time;
24     double spawn_counter = 0.0;
25     double threshold = 0.0;
26
27     while(!*m_exit_bool){
28         m_mutex->lock();
29         //std::cout << "calculating\n";
30         for(int i = 0; i < MSIM_SPEED; i++){
31             //std::cout << "a\n";
32             m_traffic->update(1.0f/(float)MFRAMERATE);
33             //std::cout << "b\n";
34             m_traffic->spawn_cars(spawn_counter, 1.0f/(float)MFRAMERATE, threshold);
35             //m_mutex->lock();
36             //std::cout << "c\n";
37             m_traffic->despawn_cars();
38             //m_mutex->unlock();
39             //std::cout << "d\n";
40         }
41         //std::cout << "calculated\n";
42         m_mutex->unlock();
43
44         time = clock.restart();
45         sf::Int64 acutal_elapsed = time.asMicroseconds();
46         double sim_elapsed = (1.0f/(float)MFRAMERATE)*1000000;
47
48         if(acutal_elapsed < sim_elapsed){
49             usleep((useconds_t)(sim_elapsed-acutal_elapsed));
50             m_traffic->m_multiplier = MSIM_SPEED;
51         }
52         else{
53             m_traffic->m_multiplier = MSIM_SPEED*(sim_elapsed/acutal_elapsed);
54         }
55     }
56 }

```

../highway/window.cpp

## A.2.3 unittests.cpp

```

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

```

```

4 //
5
6 #include "unittests.h"
7 #include <unistd.h>
8 #include <iostream>
9
10 void Tests::placement_test() {
11     std::cout << "Starting placement tests\n";
12     std::vector<RoadSegment*> segments = Road::shared().segments();
13     int i = 0;
14
15     for(RoadSegment * seg : segments){
16         usleep(100000);
17         std::cout<< "seg " << i << " , nlanes " << seg->get_total_amount_of_lanes() << " , "<< seg << std::
18     endl;
19     std::cout << "next segment" << seg->next_segment() << std::endl;
20     std::vector<RoadNode*> nodes = seg->get_nodes();
21     for(RoadNode * node : nodes){
22         std::vector<RoadNode*> connections = node->get_connections();
23         std::cout << "node" << node << " has connections:" << std::endl;
24         for(RoadNode * pointy : connections){
25             std::cout << pointy << std::endl;
26         }
27     }
28     i++;
29     m_traffic->force_place_car(seg, seg->get_nodes()[0], 1, 1, 0.01);
30     std::cout << "placed car" << std::endl;
31 }
32 std::cout << "Placement tests passed\n";
33 }
34
35 void Tests::delete_cars_test() {
36     std::vector<Car*> car_copies = m_traffic->get_car_copies();
37
38     for(Car * car : car_copies){
39         std::cout << car << std::endl;
40         usleep(100);
41         m_mutex->lock();
42         std::cout << "deleting car\n";
43         //usleep(100000);
44         //std::cout << "Removing car " << car << std::endl;
45         m_traffic->despawn_car(car);
46         m_mutex->unlock();
47         std::cout << car << std::endl;
48     }
49     std::cout << "Car despawn tests passed\n";
50 }
51
52 void Tests::run_one_car() {
53     double ten = 10.0;
54     double zero = 0;
55     m_traffic->spawn_cars(ten, 0, zero);
56     double fps = 60.0;
57     double multiplier = 10.0;
58
59     std::cout << "running one car\n";
60     while(m_traffic->n_of_cars() != 0) {
61         usleep((useconds_t)(1000000.0/(fps*multiplier)));
62         m_traffic->update(1.0f/(float)fps);
63         m_traffic->despawn_cars();
64     }
65 }
66
67 void Tests::placement_test_2() {
68     std::cout << "Starting placement tests 2\n";
69     std::vector<RoadSegment*> segments = Road::shared().segments();
70     int i = 0;
71
72     for(RoadSegment * seg : segments){
73         usleep(100000);
74         std::cout<< "seg " << i << " , nlanes " << seg->get_total_amount_of_lanes() << " , "<< seg << std::
75     endl;
76     std::cout << "next segment" << seg->next_segment() << std::endl;
77     std::vector<RoadNode*> nodes = seg->get_nodes();
78     for(RoadNode * node : nodes){
79         std::vector<RoadNode*> connections = node->get_connections();

```

```

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 }
90 void Tests::placement_test_3() {
91     std::cout << "Starting placement tests 3\n";
92     std::vector<RoadSegment*> segments = Road::shared().segments();
93
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
99     delete_cars_test();
100    //m_traffic.despawn_all_cars();
101    std::cout << "Placement tests 3 passed\n";
102 }
103
104 // do all tests
105 void Tests::run_all_tests() {
106     usleep(2000000);
107     placement_test();
108     delete_cars_test();
109     run_one_car();
110     placement_test_2();
111     placement_test_3();
112
113     std::cout << "all tests passed\n";
114 }
115
116 Tests::Tests(Traffic *& traffic, sf::Mutex *& mutex) {
117     m_traffic = traffic;
118     m_mutex = mutex;
119 }

```

../highway/unittests.cpp

#### A.2.4 main.cpp

```

1 #include <iostream>
2 #include "SFML/Graphics.hpp"
3 #include "window.h"
4 #include "unittests.h"
5
6 sf::Mutex mutex;
7
8 int main() {
9     sf::RenderWindow window(sf::VideoMode(550*2, 600*2), "My window");
10    window.setFramerateLimit(60);
11
12    int sim_speed = 1;
13    bool debug = true;
14    bool super_debug = true;
15
16    sf::Texture texture;
17    if(!texture.loadFromFile("../mall2.png"))
18    {
19
20    }
21
22    sf::Sprite background;
23    background.setTexture(texture);
24    //background.setColor(sf::Color::Black);

```

```

background.scale(2.0f,2.0f);

26
sf::Clock clock;
28
sf::Clock t0;

30
bool exit_bool = false;

32
if(!super_debug){
    sf::Mutex * mutex1 = &mutex;
34
    bool * exit = &exit_bool;
    //thread.launch();
36
    auto * traffic = new Traffic();
    Simulation sim = Simulation(traffic , mutex1,sim_speed,60,exit);
38
    sf::Text debug_info;
    Traffic copy;

40

    sf::Thread thread(&Simulation::update,&sim);
42
    thread.launch();

44
    // run the program as long as the window is open
    while (window.isOpen())
46
    //while(false)
    {
48
        // check all the window's events that were triggered since the last iteration of the loop
        sf::Event event;
50
        while (window.pollEvent(event))
        {
52
            // "close requested" event: we close the window
            if (event.type == sf::Event::Closed){
54
                exit_bool = true;
                thread.wait();
56
                window.close();
            }
58
        }
        sf::Time elapsed = clock.restart();

60

        mutex.lock();
62
        //std::cout << "copying\n";
        copy = *traffic;
64
        //std::cout << "copied\n";
        mutex.unlock();

66

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

68

        window.draw(background);
70
        //mutex.lock();
        window.draw(copy);

72

        copy.get_info(debug_info,elapsed);
74
        //mutex.unlock();
        window.draw(debug_info);

76

        window.display();
78
    }
80
}
else{
82

84
    //sf::Thread thread(&Tests::run_all_tests,&tests);
    sf::Mutex * mutex1 = &mutex;
86
    //thread.launch();
    auto * traffic = new Traffic();
88
    Tests tests = Tests(traffic , mutex1);
    Traffic copy;
90
    sf::Text debug_info;

92

    sf::Thread thread(&Tests::run_all_tests,&tests);
    thread.launch();

94

    // run the program as long as the window is open
    while (window.isOpen())
96
    {
98

        // check all the window's events that were triggered since the last iteration of the loop
        sf::Event event;
100

```



```

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 return 0;
131 }

```

../highway/main.cpp