Final Project, SI1336

Carl Schiller, 9705266436

January 18, 2019

Abstract

Contents

1 Introduction

2 Method

3 Result

4 Discussion

A Highway

A.1 Header files

A.1.1 traffic.h

```
//
// Created by Carl Schiller on 2018-12-19.
//
#include <random>
#include <vector>
#include "SFML/Graphics.hpp"

#ifndef HIGHWAY_TRAFFIC_H
#define HIGHWAY_TRAFFIC_H



class RoadSegment;

class Car;

class RoadNode{
private:
    float m_x, m_y;
    std::vector<RoadNode*> m_connecting_nodes; // raw pointers, no ownership
    RoadSegment* m_is_child_of; // raw pointer, no ownership
public:
    RoadNode();
    ~RoadNode();
    RoadNode(float x, float y, RoadSegment * segment);

    void set_pointer(RoadNode*);
    RoadSegment* get_parent_segment();
    RoadNode * get_next_node(int lane);
    std::vector<RoadNode*> & get_connections();
    float get_x();
    float get_y();
    float get_theta(RoadNode*);
};


class RoadSegment{
private:
    const float m_x, m_y;
    float m_theta;
    const int m_n_lanes;

    constexpr static float M_LANE_WIDTH = 4.0f;

    std::vector<RoadNode*> m_nodes; // OWNERSHIP
    std::vector<Car*> m_cars; // raw pointer, no ownership
    RoadSegment * m_next_segment; // raw pointer, no ownership
public:
    RoadSegment() = delete;
    RoadSegment(float x, float y, RoadSegment * next_segment, int lanes);
    RoadSegment(float x, float y, float theta, int lanes);
    RoadSegment(float x, float y, int lanes,bool merge);
    ~RoadSegment(); // rule of three
    RoadSegment(const RoadSegment&) = delete; // rule of three
    RoadSegment& operator=(const RoadSegment& rhs) = delete; // rule of three

    bool merge;

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

    int get_lane_number(RoadNode *);
```

```
    const int get_total_amount_of_lanes() const;
    void set_theta(float theta);
    void set_next_road_segment(RoadSegment*);
    void calculate_theta();
    void calculate_and_populate_nodes();
    void set_all_node_pointers_to_next_segment();
    void set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *);
};

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

    const std::string M_FILENAME;
private:
    Road();
    ~Road();
public:
    static Road &shared() {static Road road; return road;}

    Road(const Road& copy) = delete;
    Road& operator=(const Road& rhs) = delete;

    bool load_road();
    std::vector<RoadSegment*> & spawn_positions();
    std::vector<RoadSegment*> & despawn_positions();
    std::vector<RoadSegment*> & segments();
};

/**
 * Car class
 * =========
 * Private:
 * position, width of car, and velocities are stored.
 * ------------------
 * Public:
 * .update_pos(float delta_t): updates position by updating position.
 * .accelerate(float delta_v): accelerates car.
 * .steer(float delta_theta): change direction of speed.
 * .x_pos(): return reference to x_pos.
 * .y_pos(): -||- y_pos.
 */

class Car{
private:
    float m_dist_to_next_node;
    float m_speed;
    float m_theta; // radians

    float m_aggressiveness; // how fast to accelerate;
    float m_target_speed;
    bool m_breaking;

public:
    Car();
    ~Car();
    Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float agressivness);

    // all are raw pointers
    RoadSegment * current_segment;
    RoadNode * current_node;
    RoadNode * heading_to_node;
    Car * overtake_this_car;
    std::vector<Car*> want_to_overtake_me;

    std::vector<Car*> & get_overtakers();

    void update_pos(float delta_t);
    void accelerate(float delta_t);
    void avoid_collision(float delta_t);
    Car * find_closest_car();

    float x_pos();
    float y_pos();
```

```
147     float & speed();
        float & target_speed();
149     float & theta();

151     RoadSegment * get_segment();
};
153

155 class Util{
public:
157     static std::vector<std::string> split_string_by_delimiter(const std::string & str, const char delim);
        static bool is_car_behind(Car * a, Car * b);
159     static bool will_car_paths_cross(Car *a, Car*b);
        static bool is_cars_in_same_lane(Car*a,Car*b);
161     static bool merge_helper(Car*a, int merge_to_lane);
        static float distance_to_line(float theta,  float x,  float y);
163     static float distance_to_proj_point(float theta,  float x,  float y);
        static float distance_to_car(Car * a, Car * b);
165     static Car * find_closest_radius(std::vector<Car> &cars, float x, float y);
        static float get_min_angle(float ang1, float ang2);
167     static float distance(float x1, float x2, float y1, float y2);
};
169
class Traffic{
171 private:
        std::vector<Car*> m_cars;
173     std::mt19937 & my_engine();

175     //void update_speed(int i, float & elapsed_time);
        //float get_theta(float xpos, float ypos, float speed, float current_theta, bool & lane_switch);
177 public:
        Traffic();
179     ~Traffic();
        Traffic(const Traffic&); // rule of three
181     Traffic& operator=(const Traffic&); // rule of three

183     unsigned long n_of_cars();
        void spawn_cars(double & spawn_counter, float elapsed, double & threshold);
185     void despawn_cars();
        void despawn_car(Car* car);
187     //void remove_car(Car * car);
        void remove_dead_pointers();
189     void force_place_car(Car * car);


191
        void update(float elapsed_time);
193     std::vector<Car *> get_cars() const;
        float get_avg_flow();
195 };

197 #endif //HIGHWAY_TRAFFIC_H
```

../highway/traffic.h

A.1.2 window.h

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

5 #include <vector>
  #include "SFML/Graphics.hpp"
7 #include "traffic.h"

9 #ifndef HIGHWAY_WINDOW_H
  #define HIGHWAY_WINDOW_H
11
  class Simulation : public sf::Drawable, public sf::Transformable{
13 public:
      Simulation();
15    explicit Simulation(bool debug,int sim_speed);

17    void update(sf::Time elapsed, double & spawn_counter, double & threshold);
```

```
    float get_flow();
    //void car_debug(sf::Time t0);
    void get_info(sf::Text & text, sf::Time &elapsed);
private:
    virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
private:
    Traffic m_traffic;
    sf::Texture m_texture;
    bool m_debug;
    int m_sim_speed;
    sf::Font m_font;
};


#endif //HIGHWAY_WINDOW_H
```

../highway/window.h

A.1.3 unittests.h

```
//
// Created by Carl Schiller on 2019-01-16.
//

#include "traffic.h"
#include "SFML/Graphics.hpp"
#ifndef HIGHWAY_UNITTESTS_H
#define HIGHWAY_UNITTESTS_H

class Tests: public sf::Drawable, public sf::Transformable{
private:
    virtual void draw(sf::RenderTarget & target, sf::RenderStates states) const;
public:
    Tests();

    Traffic m_traffic;
    sf::Font m_font;
    void get_info(sf::Text & text, sf::Time &elapsed);

    void placement_test();
    void delete_cars_test();
    void run_one_car();
    void run_all_tests();
};

#endif //HIGHWAY_UNITTESTS_H
```

../highway/unittests.h

A.2 Source files

A.2.1 traffic.cpp

```
//
// Created by Carl Schiller on 2018-12-19.
//

#include "traffic.h"
#include <cmath>
#include <fstream>
#include <sstream>
#include <iostream>
#include <map>
#include <random>
#include <vector>
#include <list>

Car::Car() = default;

Car::Car(RoadSegment *spawn_point, int lane, float vel, float target_speed, float aggressivness):
    m_speed(vel), m_target_speed(target_speed), m_aggressiveness(aggressivness)
{
```

```
    current_segment = spawn_point;

    overtake_this_car = nullptr;

    current_segment->append_car(this);
    current_node = current_segment->get_node_pointer(lane);

    if(!current_node->get_connections().empty()){
        heading_to_node = current_node->get_next_node(lane);

        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());

        m_theta = current_node->get_theta(heading_to_node);
    }else{
        std::cout << "aa\n";
        heading_to_node = nullptr;
        m_theta = 0;
        m_dist_to_next_node = 0;
    }

    m_breaking = false;
}

Car::~Car(){
    if(this->current_segment != nullptr){
        this->current_segment->remove_car(this); // remove this pointer shit
    }
    std::vector<Car*> wants_to_overtake = this->get_overtakers(); // remove pointers to this car
    for(Car * car : wants_to_overtake){
        car->overtake_this_car = nullptr;
    }

    overtake_this_car = nullptr;
    current_segment = nullptr;
    heading_to_node = nullptr;
    current_node = nullptr;
}

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

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

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

        if(!connections.empty()){
            // check if we merge
            int current_lane = current_segment->get_lane_number(current_node);

            if(current_segment->merge){
                if(current_lane == 0 && connections[0]->get_parent_segment()->get_total_amount_of_lanes()
    != 2){
                    if(!Util::merge_helper(this,1)){
                        heading_to_node = connections[1];
                    }
                    else{
                        heading_to_node = connections[0];
                    }
                }
                else if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
                    current_lane = std::max(current_lane-1,0);
                    heading_to_node = connections[current_lane];
                }
                else{
                    heading_to_node = connections[current_lane];
                }
            }
```

```
94              // if we are in start section
                else if(current_segment->get_total_amount_of_lanes() == 3){
96                  if(connections.size() == 1){
                        heading_to_node = connections[0];
98                  }
                    else{
100                     heading_to_node = connections[current_lane];
                    }
102             }
                // if we are in middle section
104             else if(current_segment->get_total_amount_of_lanes() == 2){
                    // normal way
106                 if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
                        //see if we want to overtake car.
108                     Car * closest_car = find_closest_car();

110                     if(closest_car != nullptr && this->overtake_this_car == nullptr){
                            float delta_speed = closest_car->speed()-this->speed();
112                         float delta_distance = Util::distance_to_car(this,closest_car);
                            if(delta_distance/delta_speed > -5 && delta_speed < 2){
114                             this->overtake_this_car = closest_car;
                                this->overtake_this_car->get_overtakers().push_back(this);
116                         }
                        }

118
                        if(this->overtake_this_car != nullptr){
120                         if(current_lane != 1){
                                if(!Util::merge_helper(this,1)){
122                                 heading_to_node = connections[1];
                                }
124                             else{
                                    heading_to_node = connections[current_lane];
126                             }
                            }
128                         else{
                                if((!Util::is_car_behind(this,this->overtake_this_car) && Util::
    distance_to_car(this,this->overtake_this_car) > 10) || Util::distance_to_car(this,this->
    overtake_this_car) > 40){
130                                 std::vector<Car*> carpointers = (this->overtake_this_car->get_overtakers()
    );
                                    unsigned long size = carpointers.size();
132                                 for(int i = 0; i< size; i++){
                                        if(this == carpointers[i]){
134                                         carpointers.erase(carpointers.begin()+i);
                                            i--;
136                                         size--;
                                        }
138                                 }

140                                 this->overtake_this_car = nullptr;
                                }
142
                                heading_to_node = connections[current_lane];
144                         }
                        }
146                     // merge back if overtake this car is nullptr.
                        else{
148                         if(!Util::merge_helper(this,0)){
                                heading_to_node = connections[0];
150                         }
                            else{
152                             heading_to_node = connections[current_lane];
                            }
154                     }
                    }
156                 else{
                        heading_to_node = connections[0];
158                 }

160             }
                else if(current_segment->get_total_amount_of_lanes() == 1){
162                 heading_to_node = connections[0];
                }
164

                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());
```

```
166             m_theta = current_node->get_theta(heading_to_node);

168         }
    }
170 }

172 std::vector<Car*>& Car::get_overtakers() {
    return want_to_overtake_me;
174 }

176 void Car::accelerate(float elapsed){
    float target = m_target_speed;
178     float d_vel; // proportional control.

180     if(m_speed < target*0.75){
        d_vel = m_aggressiveness*elapsed;
182     }
    else{
184         d_vel = m_aggressiveness*(target-m_speed)*4*elapsed;
    }
186
    m_speed += d_vel;
188 }

190 void Car::avoid_collision(float delta_t) {
    float min_distance = 8.0f; // for car distance.
192     float ideal = min_distance+min_distance*(m_speed/20.f);
    float detection_distance = m_speed*4.0f;
194     //std::cout << "boop1\n";
    Car * closest_car = find_closest_car();
196     //std::cout << "boop2\n";
    float radius_to_car = 1000;
198     float delta_speed = 0;


200
    if(closest_car != nullptr) {
202         radius_to_car = Util::distance_to_car(this, closest_car);
        delta_speed = closest_car->speed() - this->speed();
204
        if (radius_to_car < ideal) {
206             m_speed -= std::min(abs(delta_speed)*delta_t+(ideal-radius_to_car)*0.5f, 10.0f * delta_t);
        }
208         else if(radius_to_car < detection_distance && delta_speed < 0){
            m_speed -= std::min(
210                     abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
    m_aggressiveness * 0.02f,
                    10.0f * delta_t);
212         }
        else {
214             accelerate(delta_t);
        }
216     }
    else{
218         accelerate(delta_t);
    }
220
    if(m_speed < 0){
222         m_speed = 0;
    }
224


226 }

228 Car* Car::find_closest_car() {
    const float search_radius = 100;
230     RoadSegment* origin = this->current_segment;
    std::map<RoadSegment*,bool> visited;
232     std::list<RoadNode*> queue;

234     for(RoadNode * node : (this->current_segment->get_nodes())){
        queue.push_front(node);
236     }

238     Car* answer = nullptr;
    float best_radius = 1000;
240
```

```
    while(!queue.empty()){
        RoadNode * next_node = queue.back(); // get last element
        queue.pop_back();
        RoadSegment * next_segment = next_node->get_parent_segment();
        //std::cout<< "hej\n";
        if(!visited[next_segment] && Util::distance(origin->get_x(),next_segment->get_x(),origin->get_y(),
    next_segment->get_y()) < search_radius){
            visited[next_segment] = true;

            for(Car * car : next_segment->get_car_vector()){
                if(this != car){
                    float radius = Util::distance_to_car(this,car);
                    if(Util::is_car_behind(this,car) && radius < best_radius){
                        if(Util::will_car_paths_cross(this,car)){
                            best_radius = radius;
                            answer = car;
                        }
                    }
                }
            }

            // push in new nodes in front of list.
            for(RoadNode * node : next_node->get_connections()){
                queue.push_front(node);
            }
        }
        //std::cout<< "hej2\n";
    }

    return answer;
}

float Car::x_pos() {
    float x_position;
    if(heading_to_node != nullptr){
        x_position = heading_to_node->get_x()-m_dist_to_next_node*cos(m_theta);
    }
    else{
        x_position = current_node->get_x();
    }

    return x_position;
}

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

    return y_position;
}

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

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

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

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


RoadNode::RoadNode() = default;

RoadNode::~RoadNode() = default;
```

```
316
RoadNode::RoadNode(float x, float y, RoadSegment * segment) {
318     m_x = x;
        m_y = y;
320     m_is_child_of = segment;
}
322
void RoadNode::set_pointer(RoadNode * next_node) {
324     m_connecting_nodes.push_back(next_node);
}
326
RoadSegment* RoadNode::get_parent_segment() {
328     return m_is_child_of;
}
330
std::vector<RoadNode*> & RoadNode::get_connections() {
332     return m_connecting_nodes;
}
334
float RoadNode::get_x() {
336     return m_x;
}
338
float RoadNode::get_y() {
340     return m_y;
}
342
float RoadNode::get_theta(RoadNode* node) {
344     for(RoadNode * road_node : m_connecting_nodes){
            if(node == road_node){
346             return atan2(m_y-node->m_y,node->m_x-m_x);
            }
348     }
        throw std::invalid_argument("Node given is not a connecting node");
350 }

352 RoadNode* RoadNode::get_next_node(int lane) {
        return m_connecting_nodes[lane];
354 }

356 //RoadSegment::RoadSegment() = default;

358 /*
RoadSegment::RoadSegment(const RoadSegment & segment):
360     m_x(segment.m_x), m_y(segment.m_y), m_n_lanes(segment.m_n_lanes),
        m_theta(segment.m_theta), m_next_segment(segment.m_next_segment),
362     m_cars(segment.m_cars), merge(segment.merge)
{
364     m_nodes.reserve(segment.m_nodes.size());
        for(RoadNode * node : m_nodes){
366         RoadNode * new_node = new RoadNode(); // new_node now is on heap.
            *new_node = *node; // copy values.
368         m_nodes.push_back(new_node); // push back pointers.
        }
370 }
*/

372
/*
374 RoadSegment& RoadSegment::operator=(const RoadSegment &rhs) {
        RoadSegment tmp(rhs);
376
        std::swap(m_nodes,tmp.m_nodes);
378
        return *this;
380 }

382 */

384 RoadSegment::~RoadSegment(){
        for(RoadNode * elem : m_nodes){
386         delete elem;
        }
388     m_nodes.clear();
}
390
RoadSegment::RoadSegment(float x, float y, RoadSegment * next_segment, int lanes):
```

```
392     m_x(x), m_y(y), m_n_lanes(lanes)
{
394     m_next_segment = next_segment;
        m_theta = atan2(m_y-m_next_segment->m_y,m_next_segment->m_x-m_x);
396
        m_nodes.reserve(m_n_lanes);
398
        calculate_and_populate_nodes();
400 }

402 RoadSegment::RoadSegment(float x, float y, float theta, int lanes) :
        m_x(x), m_y(y), m_theta(theta), m_n_lanes(lanes)
404 {
        m_next_segment = nullptr;
406     m_nodes.reserve(lanes);

408     calculate_and_populate_nodes();
}
410
412 RoadSegment::RoadSegment(float x, float y, int lanes,bool mer):
        m_x(x), m_y(y), merge(mer), m_n_lanes(lanes)
414 {
        merge = mer;
416     m_next_segment = nullptr;
        m_nodes.reserve(m_n_lanes);

418     // can't set nodes if we don't have a theta.
}
420
422 float RoadSegment::get_theta() {
        return m_theta;
}
424
426 const float RoadSegment::get_x() const{
        return m_x;
}
428
430 const float RoadSegment::get_y() const {
        return m_y;
}
432
434 int RoadSegment::get_lane_number(RoadNode * node) {
        for(int i = 0; i < m_n_lanes; i++){
            if(node == m_nodes[i]){
436             return i;
            }
438     }
        throw std::invalid_argument("Node is not in this segment");
440 }

442 void RoadSegment::append_car(Car * car) {
        m_cars.push_back(car);
444 }

446 void RoadSegment::remove_car(Car * car) {
        unsigned long size = m_cars.size();
448     bool found = false;
        for(int i = 0; i < size; i++){
450         if(car == m_cars[i]){
                m_cars.erase(m_cars.begin()+i);
452             i--;
                size--;
454             found = true;
            }
456     }
        if(!found){
458         //throw std::invalid_argument("Car is not in this segment.");
        }
460 }

462 std::vector<Car*>& RoadSegment::get_car_vector() {
        return m_cars;
464 }

466 void RoadSegment::set_theta(float theta) {
        m_theta = theta;
```

```
468 }
470 void RoadSegment::calculate_and_populate_nodes() {
    // calculates placement of nodes.
472     float total_length = M_LANE_WIDTH*(m_n_lanes-1);
    float current_length = -total_length/2.0f;
474
    for(int i = 0; i < m_n_lanes; i++){
476         float x_pos = m_x+current_length*cos(m_theta+(float)M_PI*0.5f);
        float y_pos = m_y-current_length*sin(m_theta+(float)M_PI*0.5f);
478         m_nodes.push_back(new RoadNode(x_pos,y_pos,this));
        current_length += M_LANE_WIDTH;
480     }
}
482
484 void RoadSegment::set_next_road_segment(RoadSegment * next_segment) {
    m_next_segment = next_segment;
}
486
488 void RoadSegment::calculate_theta() {
    m_theta = atan2(m_y-m_next_segment->m_y,m_next_segment->m_x-m_x);
}
490
492 RoadNode* RoadSegment::get_node_pointer(int n) {
    return m_nodes[n];
}
494
496 std::vector<RoadNode *> RoadSegment::get_nodes() {
    return m_nodes;
}
498
500 RoadSegment* RoadSegment::next_segment() {
    return m_next_segment;
}
502
504 void RoadSegment::set_all_node_pointers_to_next_segment() {
    for(RoadNode * node: m_nodes){
        for(int i = 0; i < m_next_segment->m_n_lanes; i++){
506             node->set_pointer(m_next_segment->get_node_pointer(i));
        }
508     }
}
510
512 void RoadSegment::set_node_pointer_to_node(int from_node_n, int to_node_n, RoadSegment *next_segment) {
    RoadNode * pointy = next_segment->get_node_pointer(to_node_n);
    m_nodes[from_node_n]->set_pointer(pointy);
514 }
516 const int RoadSegment::get_total_amount_of_lanes() const {
    return m_n_lanes;
518 }
520 Road::Road() :
    M_FILENAME("../road.txt")
522 {
    if(!load_road()){
524        std::cout << "Error in loading road.\n";
    };
526 }
528 Road::~Road() {
    for(RoadSegment * seg : m_segments){
530         delete seg;
    }
532     m_segments.clear();
}
534
536 bool Road::load_road() {
    bool loading = true;
    std::ifstream stream;
538     stream.open(M_FILENAME);

540     std::vector<std::vector<std::string>> road_vector;
    road_vector.reserve(100);
542
    if(stream.is_open()){
```

```
544         std::string line;
            std::vector<std::string> tokens;
546         while(std::getline(stream,line)){
                tokens = Util::split_string_by_delimiter(line,' ');
548             if(tokens[0] != "#"){
                    road_vector.push_back(tokens);
550             }
            }
552     }
        else{
554         loading = false;
        }
556 
558     // load segments into memory.
        for(std::vector<std::string> & vec : road_vector){
560         if(vec.size() == 5){
                if(vec[4] == "merge"){
562                 RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std::stoi(vec[3]),
    true);
                    m_segments.push_back(seg);
564             }
                else{
566                 RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std::stoi(vec[3]),
    false);
                    m_segments.push_back(seg);
568             }
            }
570         else{
                RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std::stoi(vec[3]),
    false);
572             m_segments.push_back(seg);
            }
574 
        }
576 
578     // populate nodes.
        for (int i = 0; i < m_segments.size(); ++i) {
580         // populate nodes normally.
            if(road_vector[i].size() == 4){
582             m_segments[i]->set_next_road_segment(m_segments[i+1]);
                m_segments[i]->calculate_theta();
584             // calculate nodes based on theta.
                m_segments[i]->calculate_and_populate_nodes();
586 
            }
588         else if(road_vector[i].size() == 5){
                if(road_vector[i][4] == "false"){
590                 // take previous direction and populate nodes.
                    m_segments[i]->set_theta(m_segments[i-1]->get_theta());
592                 m_segments[i]->calculate_and_populate_nodes();
                    // but do not connect nodes to new ones.
594 
                    // make this a despawn segment
596                 m_despawn_positions.push_back(m_segments[i]);
                }
598             else if(road_vector[i][4] == "true"){
                    m_segments[i]->set_next_road_segment(m_segments[i+1]);
600                 m_segments[i]->calculate_theta();
                    // calculate nodes based on theta.
602                 m_segments[i]->calculate_and_populate_nodes();

604                 // make this a spawn segment
                    m_spawn_positions.push_back(m_segments[i]);
606             }
                else if(road_vector[i][4] == "merge"){
608                 m_segments[i]->set_next_road_segment(m_segments[i+1]);
                    m_segments[i]->calculate_theta();
610                 // calculate nodes based on theta.
                    m_segments[i]->calculate_and_populate_nodes();
612             }

614         }
            // else we connect one by one.
616         else{
```

```
            // take previous direction and populate nodes.
            m_segments[i]->set_theta(m_segments[i-1]->get_theta());
            // calculate nodes based on theta.
            m_segments[i]->calculate_and_populate_nodes();
        }
    }

    // connect nodes.
    for (int i = 0; i < m_segments.size(); ++i) {
        // do normal connection, ie connect all nodes.
        if(road_vector[i].size() == 4){
            m_segments[i]->set_all_node_pointers_to_next_segment();
        }
        else if(road_vector[i].size() == 5){
            if(road_vector[i][4] == "false"){
                // but do not connect nodes to new ones.
            }
            else if(road_vector[i][4] == "true"){
                m_segments[i]->set_all_node_pointers_to_next_segment();
            }
            else if(road_vector[i][4] == "merge"){
                m_segments[i]->set_all_node_pointers_to_next_segment();
            }

        }
            // else we connect one by one.
        else{
            // manually connect nodes.
            int amount_of_pointers = (int)road_vector[i].size()-4;
            for(int j = 0; j < amount_of_pointers/3; j++){
                int current_pos = 4+j*3;
                RoadSegment * next_segment = m_segments[std::stoi(road_vector[i][current_pos+2])];
                m_segments[i]->set_node_pointer_to_node(std::stoi(road_vector[i][current_pos]),std::stoi(
    road_vector[i][current_pos+1]),next_segment);
            }
        }
    }
    return loading;
}

std::vector<RoadSegment*>& Road::spawn_positions() {
    return m_spawn_positions;
}

std::vector<RoadSegment*>& Road::despawn_positions() {
    return m_despawn_positions;
}

std::vector<RoadSegment*>& Road::segments() {
    return m_segments;
}

std::vector<std::string> Util::split_string_by_delimiter(const std::string &str, const char delim) {
    std::stringstream ss(str);
    std::string item;
    std::vector<std::string> answer;
    while(std::getline(ss,item,delim)){
        answer.push_back(item);
    }
    return answer;
}

// if a is behind of b, return true. else false
bool Util::is_car_behind(Car * a, Car * b){
    if(a!=b){
        float theta_to_car_b = atan2(a->y_pos()-b->y_pos(),b->x_pos()-a->x_pos());
        float theta_difference = get_min_angle(a->theta(),theta_to_car_b);
        return theta_difference < M_PI*0.45;
    }
    else{
        return false;
    }

}

// true if car paths cross
```

```
692 bool Util::will_car_paths_cross(Car *a, Car *b) {
    std::list<RoadSegment*> segments;
694     std::list<RoadNode*> nodes;
    segments.push_back(a->current_segment);
696     nodes.push_back(a->current_node);
    nodes.push_back(a->heading_to_node);
698 
    float dist_between_segments = Util::distance(a->current_segment->get_x(),b->current_segment->get_x(),
700             a->current_segment->get_y(),b->current_segment->get_y());
702     bool found_b = false;
704     while(!found_b){
        for(Car * car : segments.back()->get_car_vector()){
706             if(car == b){
                found_b = true;
708             }
        }
710         if(!found_b){
            segments.push_back(nodes.back()->get_parent_segment());
712             int seg0_lane_n = segments.back()->get_total_amount_of_lanes();
            int lane = segments.back()->get_lane_number(nodes.back());
714 
            std::vector<RoadNode*> node_choices = nodes.back()->get_connections();
716 
            // if seg0==seg1 we keep lane numbering.
718             if(node_choices.size() == seg0_lane_n){
                nodes.push_back(nodes.back()->get_connections()[lane]);
720             }
            // if we only have one choice, stick to it
722             else if(node_choices.size() == 1){
                lane = 0;
724                 nodes.push_back(nodes.back()->get_connections()[lane]);
            }
726             // last merge
            else if(node_choices.size() == 2){
728                 lane = std::min(lane-1,0);
                nodes.push_back(nodes.back()->get_connections()[lane]);
730             }

732         }
        float delta_dist = dist_between_segments - distance(b->current_segment->get_x(),segments.back()->
    get_x(),
734                 b->current_segment->get_y(),segments.back()->get_y());
        if(delta_dist <0){
736             return false;
        }
738     }

740     if(nodes.back() == b->heading_to_node){
        return true;
742     }

744     nodes.pop_back();
    // redo it.
746     if(nodes.back() == b->current_node){
        return true;
748     }

750     return false;

752     // check if cars originate at same segment and are heading to same segment
    /*
754     if(a->current_segment->get_total_amount_of_lanes() == a->heading_to_node->get_parent_segment()->
    get_total_amount_of_lanes() &&
    b->current_segment->get_total_amount_of_lanes() == b->heading_to_node->get_parent_segment()->
    get_total_amount_of_lanes()){
756         int a0 = a->current_segment->get_lane_number(a->current_node);
        int a1 = a->heading_to_node->get_parent_segment()->get_lane_number(a->heading_to_node);
758 
        int b0 = b->current_segment->get_lane_number(b->current_node);
760         int b1 = b->heading_to_node->get_parent_segment()->get_lane_number(b->heading_to_node);

762         int lane_dist_0 = a0 - b0;
        int lane_dist_1 = a1 - b1;
764         // if they originate and end at same node, they will cross. If not, they will not cross.
```

```
        if(lane_dist_0 == 0){
            return lane_dist_1 == 0;
        }
        else{
            // they cross if this product is either zero or less
            return lane_dist_0 * lane_dist_1 <= 0;
        }
    }
    // always false if we drive off highway
    else if((a->heading_to_node->get_parent_segment()->get_total_amount_of_lanes() == 2 &&
    b->heading_to_node->get_parent_segment()->get_total_amount_of_lanes() == 1 )||(
            b->heading_to_node->get_parent_segment()->get_total_amount_of_lanes() == 2 &&
            a->heading_to_node->get_parent_segment()->get_total_amount_of_lanes() == 1 )){
        return false;
    }
    else{
        return false;
    }
    */
}

bool Util::merge_helper(Car *a, int merge_to_lane) {
    RoadSegment * seg = a->current_segment;
    std::vector<Car*> cars = seg->get_car_vector();
    for(Car * car : cars){
        if(car != a){
            float delta_speed = a->speed()-car->speed();
            if(car->heading_to_node == a->current_node->get_connections()[merge_to_lane] && delta_speed <
    0){
                return true;
            }
        }
    }
    return false;
}

// this works only if a's heading to is b's current segment
bool Util::is_cars_in_same_lane(Car *a, Car *b) {
    return a->heading_to_node == b->current_node;
}

float Util::distance_to_line(const float theta, const float x, const float y){
    float x_hat,y_hat;
    x_hat = cos(theta);
    y_hat = -sin(theta);

    float proj_x = (x*x_hat+y*y_hat)*x_hat;
    float proj_y = (x*x_hat+y*y_hat)*y_hat;
    float dist = sqrt(abs(pow(x-proj_x,2.0f))+abs(pow(y-proj_y,2.0f)));

    return dist;
}

float Util::distance_to_proj_point(const float theta, const float x, const float y){
    float x_hat,y_hat;
    x_hat = cos(theta);
    y_hat = -sin(theta);
    float proj_x = (x*x_hat+y*y_hat)*x_hat;
    float proj_y = (x*x_hat+y*y_hat)*y_hat;
    float dist = sqrt(abs(pow(proj_x,2.0f))+abs(pow(proj_y,2.0f)));

    return dist;
}

float Util::distance_to_car(Car * a, Car * b){
    float delta_x = a->x_pos()-b->x_pos();
    float delta_y = b->y_pos()-a->y_pos();

    return sqrt(abs(pow(delta_x,2.0f))+abs(pow(delta_y,2.0f)));
}

Car * Util::find_closest_radius(std::vector<Car> &cars, const float x, const float y){
    Car * answer = nullptr;

    float score = 100000;
    for(Car & car : cars){
```

```
840         float distance = sqrt(abs(pow(car.x_pos()-x,2.0f))+abs(pow(car.y_pos()-y,2.0f)));
            if(distance < score){
                score = distance;
                answer = &car;
            }
        }

        return answer;
}

float Util::get_min_angle(const float ang1, const float ang2){
    float abs_diff = abs(ang1-ang2);
    float score = std::min(2.0f*(float)M_PI-abs_diff,abs_diff);
    return score;
}

float Util::distance(float x1, float x2, float y1, float y2) {
    return sqrt(abs(pow(x1-x2,2.0f))+abs(pow(y1-y2,2.0f)));
}

/*
Car * find_car_to_side(std::vector<Car> &cars, int i, Car & ref_car, float min_radius ,float view_angle){
    Car * answer = nullptr;

    std::vector<Car*> candidates;
    candidates.reserve(cars.size());

    float radius_to_next_car, theta_to_car, theta_diff_to_car_position,
            theta_diff_between_car_directions;

    float best_radius = min_radius;
    for(int j = 0; j < cars.size(); j++){
        if(i!=j){
            radius_to_next_car = sqrt(abs(pow(cars[j].x_pos()-ref_car.x_pos(),2.0f))
                                      +abs(pow(cars[j].y_pos()-ref_car.y_pos(),2.0f)));
            theta_to_car = atan2(-cars[j].y_pos()+ref_car.y_pos(),cars[j].x_pos()-ref_car.x_pos());

            theta_diff_to_car_position = get_min_angle(theta_to_car,ref_car.theta());
            theta_diff_between_car_directions = get_min_angle(ref_car.theta(),cars[j].theta());


            if(abs(theta_diff_to_car_position) > view_angle && abs(theta_diff_to_car_position) < M_PI*0.5
    &&
               abs(theta_diff_between_car_directions) < M_PI*0.1 && radius_to_next_car < best_radius){
                best_radius = radius_to_next_car;
                answer = &cars[j];
            }
        }
    }
    return answer;
}
*/

/*
void Car::avoid_collision(std::vector<Car> &cars, int i,float & elapsed, float delta_theta,
                          std::vector<std::vector<int>> & allowed_zone) {
    float min_distance = 8.0f; // for car distance.
    float ideal = min_distance+min_distance*(m_vel/20.f);
    float detection_distance = m_vel*4.0f;

    Car * closest_car_ahead = Util::find_closest_car(cars,this,allowed_zone);

    float delta_speed = 0;
    float radius_to_car = 200;

    if(closest_car_ahead != nullptr) {
        radius_to_car = Util::distance_to_car(*this, *closest_car_ahead);
        delta_speed = closest_car_ahead->speed() - this->speed();

        if (radius_to_car < ideal) {
            m_breaking = true;
        }

    }

    if(m_breaking) {
```

```
        m_vel -= std::min(std::max((ideal - radius_to_car),0.0f) * 0.5f + abs(pow(delta_speed,2.0f)), 10.0
    f * elapsed);
        if(radius_to_car > ideal*1.3f){
            m_breaking = false;
        }
    } else if(radius_to_car < detection_distance && delta_speed < 0){
        m_vel -= std::min(
                abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) * m_aggressiveness
    * 2,
                10.0f * elapsed);
    }
    else {
        accelerate(delta_theta,closest_car_ahead);
    }

    if(m_vel < 0){
        m_vel = 0;
    }

    else{
        m_vel -= std::min(abs(delta_speed)*ideal/radius_to_car + abs(pow(delta_speed,2.0f))*0.25f , 10.0f*
    elapsed);
    }
    else if () {
        m_vel -= std::min(
                abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.5f / radius_to_car, 2.0f) * m_aggressiveness *
     2,
                10.0f * elapsed);
        } else{
            accelerate(delta_theta,closest_car_ahead);
        }
    else {

    }


}

*/

/*
Traffic::Traffic(const Traffic & traffic) {

}
*/

Traffic::Traffic() = default;

Traffic::Traffic(const Traffic &ref) {
    // clear values if there are any.
    for(Car * delete_this : m_cars){
        delete delete_this;
    }
    m_cars.clear();

    // reserve place for new pointers.
    m_cars.reserve(ref.m_cars.size());

    // copy values into new pointers
    for(Car * car : ref.m_cars){
        auto new_car_pointer = new Car;
        *new_car_pointer = *car;

    }

    // values we copied are good, except the car pointers inside the car class.
    std::map<int,Car*> overtake_this_car;
    std::map<Car*,int> labeling;
    for(int i = 0; i < m_cars.size(); i++){
        overtake_this_car[i] = ref.m_cars[i]->overtake_this_car;
        labeling[ref.m_cars[i]] = i;
        m_cars[i]->overtake_this_car = nullptr; // clear copied pointers
        m_cars[i]->want_to_overtake_me.clear(); // clear copied pointers
    }
    std::map<int,int> from_to;
    for(int i = 0; i < m_cars.size(); i++){
```

```
        if(overtake_this_car[i] != nullptr){
            from_to[i] = labeling[overtake_this_car[i]];
        }
    }

    for(auto it : from_to){
        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]);
    }
}

Traffic& Traffic::operator=(const Traffic & rhs) {
    Traffic tmp(rhs);

    std::swap(m_cars,tmp.m_cars);

    return *this;
}

Traffic::~Traffic() {
    for(int i = 0; i<m_cars.size(); i++){
        delete Traffic::m_cars[i];
    }
    Traffic::m_cars.clear();
}

unsigned long Traffic::n_of_cars(){
    return m_cars.size();
}

std::mt19937& Traffic::my_engine() {
    static std::mt19937 e(std::random_device{}());
    return e;
}

void Traffic::spawn_cars(double & spawn_counter, float elapsed, double & threshold) {
    spawn_counter += elapsed;
    if(spawn_counter > threshold){
        std::exponential_distribution<double> dis(1);
        std::normal_distribution<float> aggro(3.0f,0.5f);
        std::normal_distribution<float> sp(20.0,2.0);
        std::uniform_real_distribution<float> lane(0.0f,1.0f);
        std::uniform_real_distribution<float> spawn(0.0f,1.0f);

        float speed = sp(my_engine());
        float target = speed;
        threshold = dis(my_engine());
        float aggressiveness = aggro(my_engine());
        spawn_counter = 0;
        float start_lane = lane(my_engine());
        float spawn_pos = spawn(my_engine());

        std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
        RoadSegment * seg;
        Car * new_car;
        if(spawn_pos < 0.95){
            seg = segments[0];
            if(start_lane < 0.33){
                new_car = new Car(seg,0,speed,target,aggressiveness);
            }
            else if(start_lane < 0.67){
                new_car = new Car(seg,1,speed,target,aggressiveness);
            }
            else{
                new_car = new Car(seg,2,speed,target,aggressiveness);
            }
        }
        else{
            seg = segments[1];
            new_car = new Car(seg,0,speed,target,aggressiveness);
        }

        Car * closest_car_ahead = new_car->find_closest_car();

        if(closest_car_ahead == nullptr && closest_car_ahead != new_car){
            m_cars.push_back(new_car);
```

```
        }
        else{
            float dist = Util::distance_to_car(new_car,closest_car_ahead);
            if(dist < 10){
                delete new_car;
            }
            else if (dist < 40){
                new_car->speed() = closest_car_ahead->speed();
                m_cars.push_back(new_car);
            }
            else{
                m_cars.push_back(new_car);
            }
        }
    }
}

void Traffic::despawn_car(Car *car) {
    for(int i = 0; i < m_cars.size(); i++){
        if(car == m_cars[i]){
            delete m_cars[i];
            m_cars[i] = nullptr;
        }
    }
    remove_dead_pointers();
}
/*
void Traffic::remove_car(Car *car) {
    for(int i = 0; i < m_cars.size(); i++){
        if(car == m_cars[i]){
            delete m_cars[i];
            m_cars[i] = nullptr;
        }
    }
}
*/
void Traffic::remove_dead_pointers() {
    m_cars.erase(std::remove(m_cars.begin(),m_cars.end(),nullptr),m_cars.end()); // safe remove all
    nullptrs
}

void Traffic::despawn_cars() {
    int car_amount = static_cast<int>(m_cars.size());
    for(int i = 0; i < car_amount; i++){
        for(RoadSegment * seg : Road::shared().despawn_positions()){
            if(m_cars[i] != nullptr){
                if(m_cars[i]->get_segment() == seg){
                    delete m_cars[i];
                    m_cars[i] = nullptr;
                }
            }
        }
    }
    remove_dead_pointers();
}

void Traffic::force_place_car(Car * car) {
    m_cars.push_back(car);
}

void Traffic::update(float elapsed_time) {
    //std::cout<< "boop1\n";
    for(Car * car : m_cars){
        car->avoid_collision(elapsed_time);
    }
    //std::cout<< "boop2\n";
    for(Car * car : m_cars){
        car->update_pos(elapsed_time);
    }
}

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

float Traffic::get_avg_flow() {
```

```
1138     float flow = 0;
         float i = 0;
1140     for(Car * car : m_cars){
             i++;
1142         flow += car->speed()/car->target_speed();
         }
1144     return flow/i;
}
```

../highway/traffic.cpp

A.2.2 window.cpp

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

5  #include <iostream>
   #include "traffic.h"
7  #include "window.h"
   #include <cmath>
9

11 void Simulation::draw(sf::RenderTarget &target, sf::RenderStates states) const {
       if(m_debug){
13         // print debug info about node placements and stuff
           sf::CircleShape circle;
15         circle.setRadius(4.0f);
           circle.setOutlineColor(sf::Color::Cyan);
17         circle.setOutlineThickness(1.0f);
           circle.setFillColor(sf::Color::Transparent);
19
           sf::Text segment_n;
21         segment_n.setFont(m_font);
           segment_n.setFillColor(sf::Color::Black);
23         segment_n.setCharacterSize(14);

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

29         int i = 0;

31         for(RoadSegment * segment : Road::shared().segments()){
               for(RoadNode * node : segment->get_nodes()){
33                 circle.setPosition(sf::Vector2f(node->get_x()*2-4,node->get_y()*2-4));
                   line[0].position = sf::Vector2f(node->get_x()*2,node->get_y()*2);
35                 for(RoadNode * connected_node : node->get_connections()){
                       line[1].position = sf::Vector2f(connected_node->get_x()*2,connected_node->get_y()*2);
37                     target.draw(line,states);
                   }
39                 target.draw(circle,states);


41
               }
43             segment_n.setString(std::to_string(i));
               segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4,segment->get_y()*2+4));
45             target.draw(segment_n,states);
               i++;
47         }
       }
49
       // one rectangle is all we need :)
51     sf::RectangleShape rectangle;
       rectangle.setSize(sf::Vector2f(9.4,3.4));
53     rectangle.setFillColor(sf::Color::Green);
       rectangle.setOutlineColor(sf::Color::Black);
55     rectangle.setOutlineThickness(2.0f);
       for(Car * car : m_traffic.get_cars()){
57         rectangle.setPosition(car->x_pos()*2,car->y_pos()*2);
           rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
59         sf::Uint8 colorspeed = static_cast<sf::Uint8> ((unsigned int)std::round(255 * car->speed() / car->
       target_speed()));
           rectangle.setFillColor(sf::Color(255-colorspeed,colorspeed,0,255));
```

```
61          target.draw(rectangle,states);
63          // print debug info about node placements and stuff
65          if(car->heading_to_node!=nullptr){
                sf::CircleShape circle;
67              circle.setRadius(4.0f);
                circle.setOutlineColor(sf::Color::Red);
69              circle.setOutlineThickness(2.0f);
                circle.setFillColor(sf::Color::Transparent);
71              circle.setPosition(sf::Vector2f(car->current_node->get_x()*2-4,car->current_node->get_y()*2-4)
    );
                target.draw(circle,states);
73              circle.setOutlineColor(sf::Color::Green);
                circle.setPosition(sf::Vector2f(car->heading_to_node->get_x()*2-4,car->heading_to_node->get_y
    ()*2-4));
75              target.draw(circle,states);
            }
77      }
}
79
Simulation::Simulation() {
81      m_debug = false;
        m_sim_speed = 1;
83      m_traffic = Traffic();
        if (!m_font.loadFromFile("/Library/Fonts/Arial.ttf"))
85      {
            // error...
87      }
}
89
Simulation::Simulation(bool debug, int speed) {
91      m_debug = debug;
        m_sim_speed = speed;
93
        if (!m_font.loadFromFile("/Library/Fonts/Andale Mono.ttf"))
95      {
            // error...
97      }
}
99
void Simulation::update(sf::Time elapsed, double & spawn_counter, double & threshold) {
101     float elapsed_time = elapsed.asSeconds();
        for(int i = 0; i < m_sim_speed; i++){
103         //std::cout<< "boop1\n";
            m_traffic.update(elapsed_time);
105         //std::cout<< "boop2\n";
            m_traffic.despawn_cars();
107         //std::cout<< "boop3\n";
            m_traffic.spawn_cars(spawn_counter,elapsed_time,threshold);
109     }
}
111
/*
113 void Simulation::car_debug(sf::Time t0){
        m_traffic.debug(t0);
115 }
*/
117
float Simulation::get_flow() {
119     return m_traffic.get_avg_flow();
}
121
void Simulation::get_info(sf::Text & text,sf::Time &elapsed) {
123     //TODO: SOME BUG HERE.
        float fps = 1.0f/elapsed.asSeconds();
125     float flow = get_flow();
        std::string speedy = std::to_string(fps).substr(0,2) +
127             " fps, speed: " + std::to_string(m_sim_speed).substr(0,1) + " x\nFlow " +
                std::to_string(flow).substr(0,4);
129     text.setString(speedy);
        text.setPosition(0,0);
131     text.setFillColor(sf::Color::Green);
        text.setFont(m_font);
133 }
```

../highway/window.cpp

A.2.3 unittests.cpp

```
//
// Created by Carl Schiller on 2019-01-16.
//

#include "unittests.h"
#include <unistd.h>
#include <iostream>


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

    for(RoadSegment * seg : segments){
        usleep(100000);
        std::cout<< "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ","<< seg << std::
    endl;
        std::cout << "next segment" << seg->next_segment() << std::endl;
        std::vector<RoadNode*> nodes =  seg->get_nodes();
        for(RoadNode * node : nodes){
            std::vector<RoadNode*> connections = node->get_connections();
            std::cout << "node" << node <<" has connections:" <<  std::endl;
            for(RoadNode * pointy : connections){
                std::cout << pointy << std::endl;
            }
        }
        i++;
        auto car = new Car(seg,0,1,1,0.01);
        m_traffic.force_place_car(car);
        std::cout << "placed car" << car << std::endl;
    }
    std::cout << "Placement tests passed\n";
}

void Tests::delete_cars_test() {
    std::vector<Car*> car_copies = m_traffic.get_cars();
    for(Car * car : car_copies){
        usleep(100000);
        std::cout << "Removing car " << car << std::endl;
        m_traffic.despawn_car(car);
    }
    std::cout << "Car despawn tests passed\n";
}

void Tests::run_one_car() {
    double ten = 10.0;
    double zero = 0;
    m_traffic.spawn_cars(ten,0,zero);
    double fps = 60.0;
    double multiplier = 10.0;

    std::cout << "running one car\n";
    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();
    }
}

// do all tests
void Tests::run_all_tests() {
    usleep(2000000);
    placement_test();
    delete_cars_test();
    run_one_car();
    std::cout << "all tests passed\n";
}
```

```
69 void Tests::draw(sf::RenderTarget &target, sf::RenderStates states) const {
    // print debug info about node placements and stuff
71     sf::CircleShape circle;
    circle.setRadius(4.0f);
73     circle.setOutlineColor(sf::Color::Cyan);
    circle.setOutlineThickness(1.0f);
75     circle.setFillColor(sf::Color::Transparent);

77     sf::Text segment_n;
    segment_n.setFont(m_font);
79     segment_n.setFillColor(sf::Color::Black);
    segment_n.setCharacterSize(14);
81
    sf::VertexArray line(sf::Lines,2);
83     line[0].color = sf::Color::Blue;
    line[1].color = sf::Color::Blue;
85
    int i = 0;
87
    for(RoadSegment * segment : Road::shared().segments()){
89         for(RoadNode * node : segment->get_nodes()){
            circle.setPosition(sf::Vector2f(node->get_x()*2-4,node->get_y()*2-4));
91             line[0].position = sf::Vector2f(node->get_x()*2,node->get_y()*2);
            for(RoadNode * connected_node : node->get_connections()){
93                 line[1].position = sf::Vector2f(connected_node->get_x()*2,connected_node->get_y()*2);
                target.draw(line,states);
95             }
            target.draw(circle,states);
97

99         }
        segment_n.setString(std::to_string(i));
101         segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4,segment->get_y()*2+4));
        target.draw(segment_n,states);
103         i++;
    }
105
    // one rectangle is all we need :)
107     sf::RectangleShape rectangle;
    rectangle.setSize(sf::Vector2f(9.4,3.4));
109     rectangle.setFillColor(sf::Color::Green);
    rectangle.setOutlineColor(sf::Color::Black);
111     rectangle.setOutlineThickness(2.0f);

113     for(Car * car : m_traffic.get_cars()){
        rectangle.setPosition(car->x_pos()*2,car->y_pos()*2);
115         rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
        sf::Uint8 colorspeed = static_cast<sf::Uint8> ((unsigned int)std::round(255 * car->speed() / car->
    target_speed()));
117         rectangle.setFillColor(sf::Color(255-colorspeed,colorspeed,0,255));
        target.draw(rectangle,states);
119
        // this caused crash earlier
121         if(car->heading_to_node!=nullptr){
            // print debug info about node placements and stuff
123             sf::CircleShape circle;

125             circle.setRadius(4.0f);
            circle.setOutlineColor(sf::Color::Red);
127             circle.setOutlineThickness(2.0f);
            circle.setFillColor(sf::Color::Transparent);
129             circle.setPosition(sf::Vector2f(car->current_node->get_x()*2-4,car->current_node->get_y()*2-4)
    );
            target.draw(circle,states);
131             circle.setOutlineColor(sf::Color::Green);
            circle.setPosition(sf::Vector2f(car->heading_to_node->get_x()*2-4,car->heading_to_node->get_y
    ()*2-4));
133             target.draw(circle,states);
        }
135     }
}
137
Tests::Tests() {
139     if (!m_font.loadFromFile("/Library/Fonts/Arial.ttf"))
    {
```

```
        // error...
    }
    m_traffic = Traffic();
}

void Tests::get_info(sf::Text & text,sf::Time &elapsed) {
    //TODO: SOME BUG HERE.
    float fps = 1.0f/elapsed.asSeconds();
    unsigned long amount_of_cars = m_traffic.n_of_cars();
    std::string speedy = std::to_string(fps).substr(0,2) +
                         " fps, ncars: " + std::to_string(amount_of_cars) + "\n";
    text.setString(speedy);
    text.setPosition(0,0);
    text.setFillColor(sf::Color::Black);
    text.setFont(m_font);
}
```

../highway/unittests.cpp

A.2.4 main.cpp

```
#include <iostream>
#include "SFML/Graphics.hpp"
#include "window.h"
#include "unittests.h"

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

    int sim_speed = 4;
    bool debug = true;
    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;

    if(!super_debug){
        Simulation simulation = Simulation(debug,sim_speed);
        double spawn_counter = 0.0;
        double threshold = 0.0;

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

        sf::Thread thread(std::bind(&Simulation::update,elapsed,spawn_counter,threshold),&simulation);

        sf::Text debug_info;

        // run the program as long as the window is open
        while (window.isOpen())
        {
            // 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){
                    window.close();
                }
            }

            elapsed = clock.restart();
```

```
54              //simulation.update(elapsed,spawn_counter,threshold);
56              window.clear(sf::Color(255,255,255,255));
58              window.draw(background);
                window.draw(simulation);
60              if(debug){
                    simulation.get_info(debug_info,elapsed);
62                  window.draw(debug_info);
                }
64
                window.display();
66          }
        }
68      else{
            Tests tests = Tests();
70
            sf::Thread thread(&Tests::run_all_tests,&tests);
72
            thread.launch();
74
            sf::Text debug_info;
76          // run the program as long as the window is open
            while (window.isOpen())
78          {
80              // check all the window's events that were triggered since the last iteration of the loop
                sf::Event event;
82              while (window.pollEvent(event))
                {
84                  // "close requested" event: we close the window
                    if (event.type == sf::Event::Closed){
86                      thread.terminate();
                        window.close();
88                  }
                }
90
                sf::Time elapsed = clock.restart();
92
                window.clear(sf::Color(255,255,255,255));
94
                window.draw(background);
96              window.draw(tests);
98              tests.get_info(debug_info,elapsed);
                window.draw(debug_info);
100
                window.display();
102         }
        }
104
        return 0;
106 }
```

../highway/main.cpp