Final Project, SI1336

Carl Schiller, 9705266436

March 8, 2019

Abstract

Contents

Figure 1: A typical ramp meter, image courtesy of [4]

1 Introduction

1.1 Problem formulation

This project is intended to simulate the traffic flow effect of a *time fixed ramp meter* a freeway on-ramp in Roslags Näsby trafikplats, Sweden. A *ramp meter* is a device that manages the flow of traffic onto the freeway, an example of a *ramp meter* can be seen in figure 1. More specifically, a *time fixed ramp meter* that only allow one car per green signal period will be examined. There are also more active variants of *ramp meters* which measure gaps in the traffic on the freeway to determine when to release vehicles, but this is beyond the scope of this project. Ramp metering systems have successfuly been proven to decrease congestion and reduce travel time on freeways. [5]

1.2 Complex systems

Traffic flow is a typical example of a complex system. As described in *An Introduction to Computer Simulation Methods Third Edition (revised)*, traffic flow can be simulated by modelling the system as a *Cellular Automaton*. A *Cellular Automaton* is a grid lattice which changes state on each tick based on rules and the current configuration of the lattice. [3]

2 Method

Cellular Automata was determined to not be satisfactory when trying to model the flow of the freeway. This is because lane change and collision detection worked poorly on a grid lattice in two dimensions. Another approach was considered instead.

2.1 Graphs

In order to model the road with several lanes, a *directed graph* was implemented with blocks of vertices as lanes, with directed edges as paths for the cars to drive. In other terms, cars drive on "rails" and can only change lanes on specified vertices, as can be seen in figure 2. [2]

When using a *directed graph* instead of a grid lattice, collision avoidance becomes a lot easier to implement. Time complexity also decreases, which improves simulation performance. The collision avoidance method inmplemented is $\mathcal{O}(n \cdot m^2)$, where n is the amount of cars and m is the search area. The grid lattice as previously metioned had dimensions 550x600, which was replaced by a graph with approximately 140 edges which improved performance by approximately 2000 times (if the whole system is searched for potential obstructions i.e. other cars).

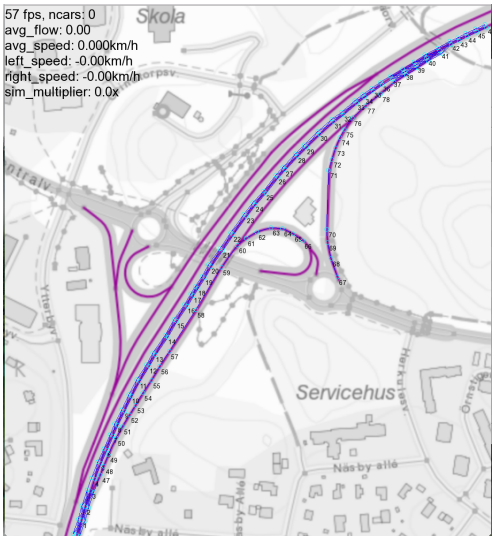


Figure 2: Setup of road with vertices and edges.

2.2 Discretization

In contrast to *Cellular Automata* there is no grid discretization, and thus the cars run on continuous "tracks". The distance traveled by each car is determined by the individual car's speed and the system wide time step size. Another benefit from the *directed graph* implementation is that the directions of the cars is not required as a parameter. All that is needed in order to simulate a car is the speed and the distance to the next vertex as well as knowing which vertex the car originated from. When stepping in time the distance traveled is subtracted from the distance to the next vertex, and when the car has reached the next vertex a new target vertex is selected.

Cars make decisions independently according to simple rules, and generates a complex behavior when interacting with each other i.e. braking or changing lanes. Some parameters are tweakable without changing the code, and each parameter influences the simulation in different ways.

2.2.1 Speed

The cars' speed is determined by a mean speed multiplied by a normally distributed variable $x \in N(1, \sigma)$, which is referred to in the code as "m_aggressiveness". "m_aggressiveness" is also involved collision detection and to determine when to overtake the car in front. σ is user tweakable.

2.2.2 Spawn rate and car headway

Cars appear in two segments, either on the on-ramp or on the beginning of the freeway. The rate of which cars appear on freeways is determined by a gamma distribution with probability density function according to equation 1. [1]

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

where α is the "shape" factor and β is the "rate" factor which are tweakable according to which behavior is sought after. The expected mean of a stochastic varaiable is $\alpha\beta$, with variance $\alpha\beta^2$. This means, a larger β implies a more spread out function.

2.2.3 Collision detection

If a car is too close to a car in front, the speed is reduced according the following rules.

```
    if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
        m_speed -= std::min(std::max((radius_to_car-min_distance)*0.5f,0.0f),10.0f*delta_t
);
    }
    else if(radius_to_car < min_distance){
        m_speed -= std::min(std::max((min_distance-radius_to_car)*0.5f,0.0f),2.0f*delta_t)
;
    }
    else if(delta_speed < 0 && radius_to_car < detection_distance){
        m_speed -= std::min(
                abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
m_aggressiveness * 0.15f,
                10.0f * delta_t);
    }
```

This ensures that a car slowly approaches the car in front. The first if statement guarantees that it will not surpass the "min_distance" distance, because the speed reduction follows this diverging sum.

$$d - \sum_{n=2}^{\infty} \frac{d}{n^2} = 0 \tag{2}$$

where d is "radius_to_car-min_distance".

2.2.4 Acceleration

If no obstruction is in the way, a car will accelerate according to:

```
    float target = m_target_speed;
    float d_vel; // proportional control.

    if(m_speed < target*0.75){
        d_vel = m_aggressiveness*elapsed*2.0f;
    }
    else{
        d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
    }

    m_speed += d_vel;
```

2.2.5 Overtake logic and merging

A car decides to overtake another car if the following conditions are met.

```
        //see if we want to overtake car.

     if(closest_car != nullptr){
         //float delta_speed = closest_car->speed()-speed();
         float delta_distance = Util::distance_to_car(this,closest_car);

         if(overtake_this_car == nullptr){
```

```
            if(delta_distance > m_min_overtake_dist_trigger && delta_distance <
    m_max_overtake_dist_trigger && (target_speed()/closest_car->target_speed() >
    m_aggressiveness*1.0f ) && current_lane == 0 && closest_car->current_node->
    get_parent_segment()->get_lane_number(closest_car->current_node) == 0){
                overtake_this_car = closest_car;
            }
        }

    }

    if(overtake_this_car !=nullptr){
        if(Util::is_car_behind(overtake_this_car,this) && (Util::distance_to_car(this,
    overtake_this_car) > m_overtake_done_dist)){
            overtake_this_car = nullptr;
        }
    }
```

A car will not merge if another car is occupying the lane it want to switch too.

2.3 Graphics rendering

When tweaking parameters involved in the cars' descision making, it is hard to get an overview of how each parameter influences the system wide behavior of the traffic. Thus a lot of effort has been spent on developing a graphical interface that shows how the traffic flows in the given configuration of parameters. An example of a test run is shown in the link below. `https://youtu.be/I7Jx8SScYZ8`

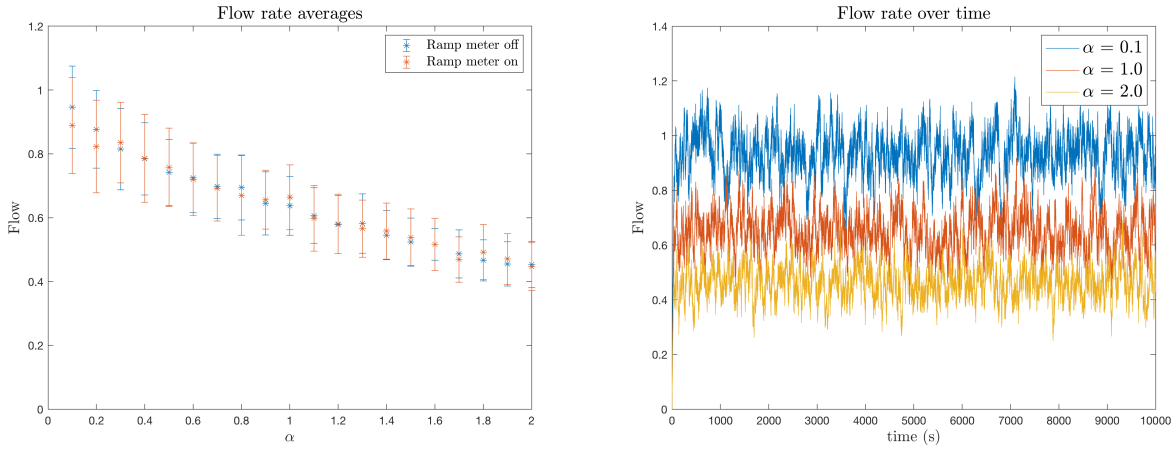
3 Result

3.1 Parameters

The following parameters have been used in the simulation. By varying Lane 1 α and Lane 2 α (the rate of which cars spawn on the freeway), the effect of a ramp meter on the system flow was determined. This was done by simulating the flow of different spawn rates with a ramp meter and without a ramp meter.

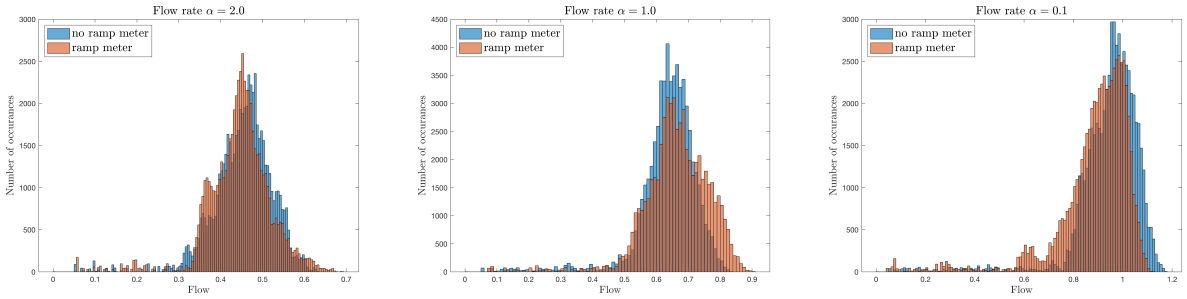
Agressiveness	1.0
Agressiveness σ	0.2
Global β	1.5
Mean speed	20 (m/s)
Lane 0 α	4.0
Lane 1 α	2.0 to 0.1 with step 0.1
Lane 2 α	2.0 to 0.1 with step 0.1
Ramp 0 α	4.0
Minimum distance to car in front	8.0 (m)
Minimum overtake distance cutoff	10.0 (m)
Maximum overtake distance cutoff	40.0 (m)
Overtake distance shutoff	30.0 (m)
Minimum merge distance	15.0 (m)
Radial search distance	30.0 (m)
Search distance forward	50.0 (m)
Time step	1/60.0 (s)
Ramp meter period	6.0 (s)

Table 1: Parameters used



(a) (b)

Figure 3: 3a) Fundamental diagram of flow as a function of α with time step 1/60 seconds. Total 60000 steps. Errorbars represent $\pm\sigma$ of deviation in flow. 3b) Flow versus time of different α with time step 1/60 seconds, no ramp meter applied. Total of 600000 steps



(a) (b) (c)

Figure 4: Histograms of flow for selected values of α. Time step 1/60 seconds, total of 60000 steps

3.2 Null hypothesis

Let X be the stochastic variable associated with the mean flow of traffic without a ramp meter over all simulated time. I.e, the outcome of X is the mean flow of a simulation at a given α. Let Y be the stochastic variable associated with the mean flow of traffic with a ramp meter over all simulated time. Then formulate the null hypothesis:

$$\begin{aligned} H_0 &: X \text{ and } Y \text{ has the same distribution} \\ H_1 &: X\text{'s distribution is skewed in relation to } Y \end{aligned} \tag{3}$$

3.3 Plots

Figure 3b has characteristics typical of stop-and-go traffic for higher densities. The fluctuations of the flow increases as the expected spawn time $\alpha\beta$ decreases, which also can be seen by looking at figure 3a. The error bars are larger for smaller α, which indicates a larger standard deviation from the mean flow. By using Wilcoxons rank sum test with equation 3 in mind, the p-value of the means in figure 3a is $p = 0.9887$.

4 Discussion

Since $p = 0.9887 > 0.05$ the null hypothesis as formulated in equation 3 can not be rejected. I.e. there is no significant difference between using a ramp meter and not using a ramp meter on a 95 % confidence level with the configuration as given in table 1. Although for some specific values of α as can be seen in figure 4 a ramp meter allows for better flow.

4.1 Considerations for further research

In this study only one parameter has been examined in table 1, and there might be configurations where a ramp meter allows for better flow over all. The way cars overtake, merge, and avoid cars is also might not be a realistic representation of how cars behave. There might be better ways to model the merging, espescially in the merging segment where the on-ramp connects to the freeway.

It is also worth mentioning that flow was defined as the sum of all cars divided by the total road length. That is, the whole system's flow was considered. If the flow instead was defined as the flow on the freeway only, and not the on-ramp, the result might have been different. This depends on what matters more, the total flow in the whole system or the flow on the freeway only.

References

[1] Ahmed Abdel-Rahim. *CE571: Traffic Flow Theory - Spring 2011*. English (United States), en-US. URL: https://www.webpages.uidaho.edu/ce571/class%20notes/Week%202%20modeling%20headway%20distribution%202011.pdf (visited on 03/07/2019).

[2] *Gerichteter Graph*. de. Page Version ID: 179253516. July 2018. URL: https://de.wikipedia.org/w/index.php?title=Gerichteter_Graph&oldid=179253516 (visited on 03/05/2019).

[3] H Gould, J Tobochnik, and W Christian. "Introduction to Computer Simulation Methods". In: (), p. 797.

[4] Patriarca12. *English: Ramp meter on ramp from Miller Park Way to Interstate 94 east in Milwaukee, Wisconsin, USA*. July 2008. URL: https://commons.wikimedia.org/wiki/File:Ramp_meter_from_Miller_Park_Way_to_I-94_east_in_Milwaukee.jpg (visited on 03/05/2019).

[5] U.S. Department of Transportation, Federal Highway Administration. *Ramp Metering: A Proven, Cost-Effective Operational Strategy - AÂ Primer: 1. Overview of Ramp Metering*. URL: https://ops.fhwa.dot.gov/publications/fhwahop14020/sec1.htm (visited on 03/05/2019).

A Header files

A.1 cars.h

```
//
// Created by Carl Schiller on 2019-03-04.
//

#ifndef HIGHWAY_CAR_H
#define HIGHWAY_CAR_H

////////////////////////////////////////////////////////////////////////
//                                                                    //
// Car                                                                //
//                                                                    //
// Describes a car that moves around in Road class                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////

#include <map>
#include "roadnode.h"
#include "roadsegment.h"

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;

    const float m_min_dist_to_car_in_front;
    const float m_min_overtake_dist_trigger;
    const float m_max_overtake_dist_trigger;
    const float m_overtake_done_dist;
    const float m_merge_min_dist;
    const float m_search_radius_around;
    const float m_search_radius_to_car_in_front;

public:
    Car();
    ~Car();
    Car& operator=(const Car&) = default;

    Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float
    agressivness,
        float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
    m_max_overtake_dist_trigger,
        float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
        float m_search_radius_to_car_in_front);
    Car(RoadSegment * spawn_point, RoadNode * lane, float vel, float target_speed, float
    agressivness,
        float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
    m_max_overtake_dist_trigger,
        float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
        float m_search_radius_to_car_in_front);

    // all are raw pointers
    RoadSegment * current_segment;
    RoadNode * current_node;
    RoadNode * heading_to_node;
    Car * overtake_this_car;

    void update_pos(float delta_t);
    void merge(std::vector<RoadNode*> & connections);
    void do_we_want_to_overtake(Car * & closest_car, int & current_lane);
```

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

    float x_pos();
    float y_pos();

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

    RoadSegment * get_segment();
};

#endif //HIGHWAY_CAR_H
```

../highway/headers/car.h

A.2 road.h

```
//
// Created by Carl Schiller on 2019-03-04.
//

#ifndef HIGHWAY_ROAD_H
#define HIGHWAY_ROAD_H

////////////////////////////////////////////////////////////////////////
//                                                                    //
// Road                                                               //
//                                                                    //
// Describes a road with interconnected nodes. Mathematically it is   //
// a graph.                                                           //
//                                                                    //
////////////////////////////////////////////////////////////////////////

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

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;} // in order to only load road
    once in memory

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

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

#endif //HIGHWAY_ROAD_H
```

../highway/headers/road.h

A.3 roadnode.h

```
//
// Created by Carl Schiller on 2019-03-04.
//

#ifndef HIGHWAY_ROADNODE_H
#define HIGHWAY_ROADNODE_H

////////////////////////////////////////////////////////////////////////////
//                                                                        //
// RoadNode                                                               //
//                                                                        //
// Describes the smallest element in Road, it is similar to               //
// that of a mathematical graph with nodes and edges.                     //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

#include <vector>
#include "car.h"
#include "roadsegment.h"

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

    void set_next_node(RoadNode *);
    void set_previous_node(RoadNode *);
    RoadSegment* get_parent_segment();
    RoadNode * get_next_node(int lane);
    std::vector<RoadNode*> & get_nodes_from_me();
    std::vector<RoadNode*> & get_nodes_to_me();
    float get_x();
    float get_y();
    float get_theta(RoadNode*);
};


#endif //HIGHWAY_ROADNODE_H
```

../highway/headers/roadnode.h

A.4 roadsegment.h

```
//
// Created by Carl Schiller on 2019-03-04.
//

#ifndef HIGHWAY_ROADSEGMENT_H
#define HIGHWAY_ROADSEGMENT_H

////////////////////////////////////////////////////////////////////////////
//                                                                        //
```

```
// RoadSegment                                                                //
//                                                                            //
// Describes a container for several RoadNodes                                //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

#include <vector>

class RoadNode;

class Car;

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
    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;
    std::vector<Car*> m_cars; // raw pointer, no ownership
    float ramp_counter;
    bool car_passed;
    bool meter;
    float period;

    RoadNode * get_node_pointer(int n);
    std::vector<RoadNode *> get_nodes();
    void append_car(Car*);
    void remove_car(Car*);
    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 *);
};

#endif //HIGHWAY_ROADSEGMENT_H
```

../highway/headers/roadsegment.h

A.5 simulation.h

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

```
#ifndef HIGHWAY_WINDOW_H
#define HIGHWAY_WINDOW_H

////////////////////////////////////////////////////////////////////////////
//                                                                        //
// Simulation                                                             //
//                                                                        //
// Describes how to simulate Traffic class                                //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

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

class Simulation{
private:
    sf::Mutex * m_mutex;
    Traffic * m_traffic;
    bool * m_exit_bool;
    const int M_SIM_SPEED;
    const int M_FRAMERATE;
public:
    Simulation() = delete;
    Simulation(Traffic *& traffic,sf::Mutex *& mutex, int sim_speed, int m_framerate, bool
    *& exitbool);

    void update();
};


#endif //HIGHWAY_WINDOW_H
```

../highway/headers/simulation.h

A.6 traffic.h

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

#ifndef HIGHWAY_TRAFFIC_H
#define HIGHWAY_TRAFFIC_H

////////////////////////////////////////////////////////////////////////////
//                                                                        //
// Traffic                                                                //
//                                                                        //
// Describes the whole traffic situation with Cars and a Road.            //
// Inherits form SFML Graphics.hpp in order to render the cars.           //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

#include <random>
#include <vector>
#include "SFML/Graphics.hpp"
#include "car.h"

class Traffic : public sf::Drawable, public sf::Transformable{
private:
    std::vector<Car*> m_cars;
    bool debug;
    std::mt19937 & my_engine();
    sf::Font m_font;
```

```
29      const float m_aggro;
        const float m_aggro_sigma;
31      const float m_spawn_freq;
        const float m_speed;
33
        const float m_lane_0_spawn_prob;
35      const float m_lane_1_spawn_prob;
        const float m_lane_2_spawn_prob;
37      const float m_ramp_0_spawn_prob;

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

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


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

79  #endif //HIGHWAY_TRAFFIC_H
```

../highway/headers/traffic.h

A.7 unittests.h

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


5
    #ifndef HIGHWAY_UNITTESTS_H
7   #define HIGHWAY_UNITTESTS_H

9   ////////////////////////////////////////////////////////////////////////////
```

```
//                                                                          //
// Tests                                                                    //
//                                                                          //
// Testing the various functions.                                           //
//                                                                          //
//////////////////////////////////////////////////////////////////////////////

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

class Tests{
private:
    Traffic * m_traffic;
    sf::Mutex * m_mutex;
    void placement_test();
    void delete_cars_test();
    void run_one_car();
    void placement_test_2();
    void placement_test_3();
public:
    Tests() = delete;
    Tests(Traffic *& traffic, sf::Mutex *& mutex);

    void run_all_tests();
};

#endif //HIGHWAY_UNITTESTS_H
```

../highway/headers/unittests.h

A.8 util.h

```
//
// Created by Carl Schiller on 2019-03-04.
//

#ifndef HIGHWAY_UTIL_H
#define HIGHWAY_UTIL_H

//////////////////////////////////////////////////////////////////////////////
//                                                                          //
// Util                                                                     //
//                                                                          //
// Help functions for Car class.                                            //
//                                                                          //
//////////////////////////////////////////////////////////////////////////////

#include "car.h"

class Util{
public:
    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);
    static bool will_car_paths_cross(Car *a, Car*b);
    static float distance_to_car(Car * a, Car * b);
    static float get_min_angle(float ang1, float ang2);
    static float distance(float x1, float x2, float y1, float y2);
};

#endif //HIGHWAY_UTIL_H
```

../highway/headers/util.h

B Source files

B.1 cars.cpp

```
//
// Created by Carl Schiller on 2019-03-04.
//

#include "../headers/car.h"
#include <map>
#include <cmath>
#include <list>
#include <iostream>
#include "../headers/util.h"

////////////////////////////////////////////////////////////////////////////
/// Constructor.

Car::Car() :
        m_speed(0),
        m_aggressiveness(0),
        m_target_speed(0),
        m_min_dist_to_car_in_front(0),
        m_min_overtake_dist_trigger(0),
        m_max_overtake_dist_trigger(0),
        m_overtake_done_dist(0),
        m_merge_min_dist(0),
        m_search_radius_around(0),
        m_search_radius_to_car_in_front(0),
        current_segment(nullptr),
        current_node(nullptr),
        overtake_this_car(nullptr)
{

}


////////////////////////////////////////////////////////////////////////////
/// Constructor for new car with specified lane numbering in spawn point.
/// Lane numbering @param lane must not exceed amount of lanes in
/// @param spawn_point, otherwise an exception will be thrown.

Car::Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float
    agressivness,
         float m_min_dist_to_car_in_front, float m_min_overtake_dist_trigger, float
    m_max_overtake_dist_trigger,
         float m_overtake_done_dist, float m_merge_min_dist, float m_search_radius_around,
         float m_search_radius_to_car_in_front) :
        m_speed(vel),
        m_aggressiveness(agressivness),
        m_target_speed(target_speed),
        m_min_dist_to_car_in_front(m_min_dist_to_car_in_front),
        m_min_overtake_dist_trigger(m_min_overtake_dist_trigger),
        m_max_overtake_dist_trigger(m_max_overtake_dist_trigger),
        m_overtake_done_dist(m_overtake_done_dist),
        m_merge_min_dist(m_merge_min_dist),
        m_search_radius_around(m_search_radius_around),
        m_search_radius_to_car_in_front(m_search_radius_to_car_in_front),
        current_segment(spawn_point),
        current_node(current_segment->get_node_pointer(lane)),
        overtake_this_car(nullptr)
{
    current_segment->append_car(this);

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

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

64          m_theta = current_node->get_theta(heading_to_node);
    }
66      else{
            throw std::invalid_argument("Car spawns in node with empty connections, or with a
    nullptr segment");
68      }
}
70
////////////////////////////////////////////////////////////////////////////
72 /// Constructor for new car with specified lane. Note that
/// @param lane must be in @param spawn_point, otherwise no guarantee on
74 /// functionality.

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

96      if(!current_node->get_nodes_from_me().empty() || current_segment->next_segment() !=
    nullptr){
            heading_to_node = current_node->get_next_node(0);
98
            m_dist_to_next_node = Util::distance(current_node->get_x(),heading_to_node->get_x(),
    current_node->get_y(),heading_to_node->get_y());
100
            m_theta = current_node->get_theta(heading_to_node);
102     }
        else{
104         throw std::invalid_argument("Car spawns in node with empty connections, or with a
    nullptr segment");
        }
106 }

108 ////////////////////////////////////////////////////////////////////////////
/// Destructor for car.
110
Car::~Car(){
112     if(this->current_segment != nullptr){
            this->current_segment->remove_car(this); // remove this pointer shit
114     }

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

122 ////////////////////////////////////////////////////////////////////////////
```

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

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
            if(current_segment->meter){
                current_segment->car_passed = true;
            }
        }

        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_nodes_from_me();

        if(!connections.empty()){

            merge(connections);

            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);

        }
    }
}

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

void Car::merge(std::vector<RoadNode*> & connections) {
    // check if we merge
    int current_lane = current_segment->get_lane_number(current_node);
    bool can_merge = true;
    std::map<Car*,bool> cars_around_car = find_cars_around_car();
    Car * closest_car = find_closest_car_ahead();

    for(auto it : cars_around_car){
        float delta_dist = Util::distance_to_car(it.first,this);
        float delta_speed = abs(speed()-it.first->speed());

        if(current_lane == 0 && it.first->heading_to_node->get_parent_segment()->
    get_lane_number(it.first->heading_to_node) == 1 ){
            can_merge =
                    delta_dist > std::max(delta_speed*4.0f/m_aggressiveness,m_merge_min_dist
    );
        }
        else if(current_lane == 1 && it.first->heading_to_node->get_parent_segment()->
    get_lane_number(it.first->heading_to_node) == 0){
            can_merge =
                    delta_dist > std::max(delta_speed*4.0f/m_aggressiveness,m_merge_min_dist
    );
        }

        if(!can_merge){
            break;
        }
    }

    if(current_segment->merge){
```

```
        if(current_lane == 0 && connections[0]->get_parent_segment()->
    get_total_amount_of_lanes() != 2){
186             if(can_merge){
                    heading_to_node = connections[1];
188             }
                else{
190                 heading_to_node = connections[0];
                }
192         }
            else if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
194             current_lane = std::max(current_lane-1,0);
                heading_to_node = connections[current_lane];
196         }
            else{
198             heading_to_node = connections[current_lane];
            }
200     }
            // if we are in start section
202     else if(current_segment->get_total_amount_of_lanes() == 3){
            if(connections.size() == 1){
204             heading_to_node = connections[0];
            }
206         else{
                heading_to_node = connections[current_lane];
208         }
        }
210         // if we are in middle section
        else if(current_segment->get_total_amount_of_lanes() == 2){
212         // normal way
            if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
214             // check if we want to overtake car in front
                do_we_want_to_overtake(closest_car,current_lane);
216
                // commited to overtaking
218             if(overtake_this_car != nullptr){
                    if(current_lane != 1){
220                     if(can_merge){
                            heading_to_node = connections[1];
222                     }
                        else{
224                         heading_to_node = connections[current_lane];
                        }
226                 }
                    else{
228                     heading_to_node = connections[current_lane];
                    }
230
                }
232                 // merge back if overtake this car is nullptr.
                else{
234                 if(can_merge){
                        heading_to_node = connections[0];
236                 }
                    else{
238                     heading_to_node = connections[current_lane];
                    }
240             }

242         }
            else{
244             heading_to_node = connections[0];
            }
246     }
        else if(current_segment->get_total_amount_of_lanes() == 1){
248         heading_to_node = connections[0];
        }
250 }
```

```
252 ////////////////////////////////////////////////////////////////////////////
    /// Helper function to determine if this car wants to overtake
254 /// @param closest_car.

256 void Car::do_we_want_to_overtake(Car * & closest_car, int & current_lane) {
        //see if we want to overtake car.
258
        if(closest_car != nullptr){
260         //float delta_speed = closest_car->speed()-speed();
            float delta_distance = Util::distance_to_car(this,closest_car);
262
            if(overtake_this_car == nullptr){
264             if(delta_distance > m_min_overtake_dist_trigger && delta_distance <
    m_max_overtake_dist_trigger && (target_speed()/closest_car->target_speed() >
    m_aggressiveness*1.0f ) && current_lane == 0 && closest_car->current_node->
    get_parent_segment()->get_lane_number(closest_car->current_node) == 0){
                    overtake_this_car = closest_car;
266             }
            }
268
        }
270
        if(overtake_this_car !=nullptr){
272         if(Util::is_car_behind(overtake_this_car,this) && (Util::distance_to_car(this,
    overtake_this_car) > m_overtake_done_dist)){
                overtake_this_car = nullptr;
274         }
        }
276 }

278 ////////////////////////////////////////////////////////////////////////////
    /// Function to accelerate this car.
280
    void Car::accelerate(float elapsed){
282     float target = m_target_speed;
        float d_vel; // proportional control.
284
        if(m_speed < target*0.75){
286         d_vel = m_aggressiveness*elapsed*2.0f;
        }
288     else{
            d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
290     }

292     m_speed += d_vel;
    }
294
    ////////////////////////////////////////////////////////////////////////////
296 /// Helper function to avoid collision with another car.

298 void Car::avoid_collision(float delta_t) {
        float min_distance = m_min_dist_to_car_in_front; // for car distance.
300     float ideal = min_distance+min_distance*(m_speed/20.f);

302     Car * closest_car = find_closest_car_ahead();
        float detection_distance = m_speed*5.0f;
304
        if(closest_car != nullptr) {
306         float radius_to_car = Util::distance_to_car(this, closest_car);
            float delta_speed = closest_car->speed() - this->speed();
308
            if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
310             m_speed -= std::max(std::max((radius_to_car-min_distance)*0.5f,0.0f),10.0f*
    delta_t);
            }
312         else if(radius_to_car < min_distance){
                m_speed -= std::max(std::max((min_distance-radius_to_car)*0.5f,0.0f),2.0f*
    delta_t);
```

```
314         }
            else if(delta_speed < 0 && radius_to_car < detection_distance){
316             m_speed -= std::min(
                        abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) *
     m_aggressiveness * 0.15f,
318                     10.0f * delta_t);
            }
320         else {
                accelerate(delta_t);
322         }

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

330                 if(it.first->current_node->get_parent_segment()->get_lane_number(it.first->
    current_node) == 0 && delta_dist < ideal && this->current_segment->get_lane_number(
    current_node) == 1 && speed()/target_speed() > 0.5){
                        if(Util::is_car_behind(it.first,this)){
332                         accelerate(delta_t);
                        }
334                     else{
                            m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*
    delta_t);
336                     }
                    }
338                 else if(it.first->current_node->get_parent_segment()->get_lane_number(it.
    first->current_node) == 1 && this->current_segment->get_lane_number(current_node) == 0
    && speed()/target_speed() > 0.5 && delta_dist < ideal){
                        if(Util::is_car_behind(this,it.first)){
340                         m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*
    delta_t);
                        }
342                     else{
                            accelerate(delta_t);
344                     }
                    }
346             }
            }
348         else{

350         }
        }
352
        if(heading_to_node->get_parent_segment()->meter){
354         if(heading_to_node->get_parent_segment()->car_passed || heading_to_node->
    get_parent_segment()->ramp_counter < heading_to_node->get_parent_segment()->period*0.5f)
    {
                if (m_dist_to_next_node < ideal) {
356                 m_speed -= std::max(std::max((m_dist_to_next_node-min_distance)*0.5f,0.0f)
    ,10.0f*delta_t);
                }
358             else if(m_dist_to_next_node < detection_distance){
                    m_speed -= std::min(
360                         abs(pow(m_speed, 2.0f)) * pow(ideal * 0.25f / m_dist_to_next_node,
    2.0f) * m_aggressiveness * 0.15f,
                        10.0f * delta_t);
362             }
            }
364         else{
                accelerate(delta_t);
366         }
        }
368     else{
            accelerate(delta_t);
370     }
```

```
372     if(m_speed < 0){
            m_speed = 0;
374     }


376
}
378
//////////////////////////////////////////////////////////////////////////////
380 /// Helper function to find closest car in the same lane ahead of this car.
    /// Returns a car if found, otherwise nullptr.
382
Car* Car::find_closest_car_ahead() {
384     float search_radius = m_search_radius_to_car_in_front;
        std::map<RoadNode*,bool> visited;
386     std::list<RoadNode*> queue;

388     for(RoadNode * node : (this->current_segment->get_nodes())){
            queue.push_front(node);
390     }

392     Car* answer = nullptr;

394     float shortest_distance = 10000000;

396     while(!queue.empty()){
            RoadNode * next_node = queue.back(); // get last element
398         queue.pop_back(); // remove element

400         if(next_node != nullptr){
                if(!visited[next_node] && Util::distance(x_pos(),next_node->get_x(),y_pos(),
    next_node->get_y()) < search_radius){
402                 visited[next_node] = true;

404                 for(Car * car : next_node->get_parent_segment()->m_cars){
                        if(this != car){
406                         float radius = Util::distance_to_car(this,car);
                            if(Util::is_car_behind(this,car) && Util::will_car_paths_cross(this,
    car) && radius < shortest_distance){
408                             shortest_distance = radius;
                                answer = car;
410                         }

412                     }
                    }
414
                    // push in new nodes in front of list.
416                 for(RoadNode * node : next_node->get_nodes_from_me()){
                        queue.push_front(node);
418                 }
                }
420         }
        }
422     return answer;
}
424
//////////////////////////////////////////////////////////////////////////////
426 /// Searches for cars around this car in a specified radius. Note that
    /// search radius is the radius to RoadNodes, and not surrounding cars.
428 /// Returns a map of cars the function has found.

430 std::map<Car *,bool> Car::find_cars_around_car() {
        float search_radius = m_search_radius_around;
432     std::map<RoadNode*,bool> visited;
        std::list<RoadNode*> queue;
434
        for(RoadNode * node : (this->current_segment->get_nodes())){
436         queue.push_front(node);
```

```
    }

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

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

                for(RoadNode * node: next_node->get_nodes_to_me()){
                    queue.push_front(node);
                }
            }
        }
    }
    return answer;
}

////////////////////////////////////////////////////////////////////////////
/// Returns x position of car.

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;
}

////////////////////////////////////////////////////////////////////////////
/// Returns y position of car.

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;
}

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

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

////////////////////////////////////////////////////////////////////////////
```

```
504 /// Returns target speed of car as reference.
506 float & Car::target_speed() {
        return m_target_speed;
508 }
510 ////////////////////////////////////////////////////////////////////////////
    /// Returns theta of car, the direction of the car. Defined in radians as a
512 /// mathematitan would define angles.
514 float & Car::theta() {
        return m_theta;
516 }
518 ////////////////////////////////////////////////////////////////////////////
    /// Returns current segment car is in.
520
    RoadSegment* Car::get_segment() {
522     return current_segment;
    }
```

../highway/cppfiles/car.cpp

B.2 main.cpp

```
1  #include <iostream>
   #include <vector>
3  #include "SFML/Graphics.hpp"
   #include "../headers/simulation.h"
5  #include "../headers/unittests.h"
   #include "../headers/screens.h"
7
   int main() {
9      std::vector<cScreen*> Screens;
       int screen = 0;
11
       sf::RenderWindow App(sf::VideoMode(550*2, 600*2), "Highway");
13     App.setFramerateLimit(60);

15     screen_0 s0;
       Screens.push_back(&s0);
17     screen_1 s1;
       Screens.push_back(&s1);
19     screen_2 s2;
       Screens.push_back(&s2);
21     screen_3 s3;
       Screens.push_back(&s3);
23
       std::vector<float> args;
25
       float m_aggro = 1.0f;
27     args.push_back(m_aggro);
       float m_aggro_sigma = 0.2f;
29     args.push_back(m_aggro_sigma);
       float m_spawn_freq = 2.0f;
31     args.push_back(m_spawn_freq);
       float m_speed = 20.f;
33     args.push_back(m_speed);

35     float m_lane_0_spawn_prob = 5.f;
       args.push_back(m_lane_0_spawn_prob);
37     float m_lane_1_spawn_prob = 1.f;
       args.push_back(m_lane_1_spawn_prob);
39     float m_lane_2_spawn_prob = 1.f;
       args.push_back(m_lane_2_spawn_prob);
41     float m_ramp_0_spawn_prob = 5.f;
```

```
    args.push_back(m_ramp_0_spawn_prob);

    float m_min_dist_to_car_in_front = 8;
    args.push_back(m_min_dist_to_car_in_front);
    float m_min_overtake_dist_trigger = 10;
    args.push_back(m_min_overtake_dist_trigger);
    float m_max_overtake_dist_trigger = 40;
    args.push_back(m_max_overtake_dist_trigger);
    float m_overtake_done_dist = 30;
    args.push_back(m_overtake_done_dist);
    float m_merge_min_dist = 15.0f;
    args.push_back(m_merge_min_dist);
    float m_search_radius_around = 30;
    args.push_back(m_search_radius_around);
    float m_search_radius_to_car_in_front = 50;
    args.push_back(m_search_radius_to_car_in_front);
    float sim_speed = 10;
    args.push_back(sim_speed);
    float framerate = 60;
    args.push_back(framerate);
    float ramp_meter_period = 10;
    args.push_back(ramp_meter_period);

    std::vector<bool> bool_args;
    bool debug = false;
    bool_args.push_back(debug);
    bool ramp_meter = false;
    bool_args.push_back(ramp_meter);

    while(screen >= 0){
        screen = Screens[screen]->Run(App,&args,&bool_args);
    }

    return 0;
}
```

../highway/cppfiles/main.cpp

B.3 road.cpp

```
//
// Created by Carl Schiller on 2019-03-04.
//

#include "../headers/road.h"
#include <fstream>
#include <vector>
#include "../headers/roadsegment.h"
#include <iostream>
#include "../headers/util.h"

////////////////////////////////////////////////////////////////////////////
/// Constructor of Road.

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

////////////////////////////////////////////////////////////////////////////
/// Destructor of Road.

Road::~Road() {
```

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

////////////////////////////////////////////////////////////////////////////
/// Function to load Road from txt file. Parsing as follows:
///
/// # ignores current line input.
///
/// If there are 4 tokens in current line:
/// tokens[0]: segment number
/// tokens[1]: segment x position
/// tokens[2]: segment y position
/// tokens[3]: amount of lanes
///
/// If there are 5 tokens in current line:
/// tokens[0]: segment number
/// tokens[1]: segment x position
/// tokens[2]: segment y position
/// tokens[3]: amount of lanes
/// tokens[4]: spawn point or if it's a merging lane (true/false/merge)
///
/// If there are 4+3*n tokens in current line:
/// tokens[0]: segment number
/// tokens[1]: segment x position
/// tokens[2]: segment y position
/// tokens[3]: amount of lanes
/// tokens[3+3*n]: from lane number of current segment
/// tokens[4+3*n]: to lane number of segment specified in next token (below)
/// tokens[5+3*n]: to segment number.

bool Road::load_road() {
    bool loading = true;
    std::ifstream stream;
    stream.open(M_FILENAME);

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

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


    // load segments into memory.
    for(std::vector<std::string> & vec : road_vector){
        if(vec.size() == 5){
            if(vec[4] == "merge"){
                RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std
    ::stoi(vec[3]),true);
                m_segments.push_back(seg);
            }
            else if(vec[4] == "ramp"){
                RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std
    ::stoi(vec[3]),false);
                m_segments.push_back(seg);
```

```
                ramp_meter_position = seg;
            }
            else{
                RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std
    ::stoi(vec[3]),false);
                m_segments.push_back(seg);
            }
        }
        else{
            RoadSegment * seg = new RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std::
    stoi(vec[3]),false);
            m_segments.push_back(seg);
        }

    }


    // populate nodes.
    for (int i = 0; i < m_segments.size(); ++i) {
        // populate nodes normally.
        if(road_vector[i].size() == 4){
            m_segments[i]->set_next_road_segment(m_segments[i+1]);
            m_segments[i]->calculate_theta();
            // calculate nodes based on theta.
            m_segments[i]->calculate_and_populate_nodes();

        }
        else if(road_vector[i].size() == 5){
            if(road_vector[i][4] == "false"){
                // take previous direction and populate nodes.
                m_segments[i]->set_theta(m_segments[i-1]->get_theta());
                m_segments[i]->calculate_and_populate_nodes();
                // but do not connect nodes to new ones.

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

                // make this a spawn segment
                m_spawn_positions.push_back(m_segments[i]);
            }
            else if(road_vector[i][4] == "merge" || road_vector[i][4] == "ramp"){
                m_segments[i]->set_next_road_segment(m_segments[i+1]);
                m_segments[i]->calculate_theta();
                // calculate nodes based on theta.
                m_segments[i]->calculate_and_populate_nodes();
            }
        }
            // else we connect one by one.
        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" || road_vector[i][4] == "merge" ||
    road_vector[i][4] == "ramp"){
                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;
}

////////////////////////////////////////////////////////////////////////////
/// Returns spawn positions of Road

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

////////////////////////////////////////////////////////////////////////////
/// Returns despawn positions of Road

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

////////////////////////////////////////////////////////////////////////////
/// Returns all segments of Road.

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

../highway/cppfiles/road.cpp

B.4 roadnode.cpp

```
//
// Created by Carl Schiller on 2019-03-04.
//

#include "../headers/roadnode.h"
#include <cmath>

////////////////////////////////////////////////////////////////////////////
/// Constructor

RoadNode::RoadNode() = default;

////////////////////////////////////////////////////////////////////////////
/// Destructor

```

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

////////////////////////////////////////////////////////////////////////////////
/// Constructor, @param x is x position of node, @param y is y position of node,
/// @param segment is to which segment this RoadNode belongs.

RoadNode::RoadNode(float x, float y, RoadSegment * segment) {
    m_x = x;
    m_y = y;
    m_is_child_of = segment;
}

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

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

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

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

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

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

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

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

////////////////////////////////////////////////////////////////////////////////
/// Returns connections to this RoadNode.

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

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

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

////////////////////////////////////////////////////////////////////////////////
/// Returns y position of RoadNode.

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

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

```

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

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

RoadNode* RoadNode::get_next_node(int lane) {
    return m_nodes_from_me[lane];
}
```

../highway/cppfiles/roadnode.cpp

B.5 roadsegment.cpp

```
//
// Created by Carl Schiller on 2019-03-04.
//

#include "../headers/roadsegment.h"
#include "../headers/roadnode.h"
#include <cmath>

////////////////////////////////////////////////////////////////////////////
/// RoadSegment destructor, removes all RodeNode element children because of
/// ownership.

RoadSegment::~RoadSegment(){
    for(RoadNode * elem : m_nodes){
        delete elem;
    }
    m_nodes.clear();
}

////////////////////////////////////////////////////////////////////////////
/// Constructor, creates a new segment with next connecting segment as
/// @param next_segment

RoadSegment::RoadSegment(float x, float y, RoadSegment * next_segment, int lanes):
        m_x(x),
        m_y(y),
        m_n_lanes(lanes),
        m_next_segment(next_segment)
{
    m_theta = atan2(m_y-m_next_segment->m_y,m_next_segment->m_x-m_x);

    m_nodes.reserve(m_n_lanes);

    ramp_counter = 0;
    car_passed = false;
    meter = false;
    period = 0;

    calculate_and_populate_nodes(); // populates segment with RoadNodes.
}

////////////////////////////////////////////////////////////////////////////
/// Constructor, creates a new segment with manually entered @param theta.

RoadSegment::RoadSegment(float x, float y, float theta, int lanes) :
```

```
46          m_x(x),
            m_y(y),
48          m_theta(theta),
            m_n_lanes(lanes),
50          m_next_segment(nullptr)
{
52      m_nodes.reserve(m_n_lanes);

54      ramp_counter = 0;
        car_passed = false;
56      meter = false;
        period = 0;
58
        calculate_and_populate_nodes(); // populates segment with RoadNodes.
60  }

62  ////////////////////////////////////////////////////////////////////////////
    /// Constructor, creates a new segment without creating RoadNodes. This
64  /// needs to be done manually with functions below.

66  RoadSegment::RoadSegment(float x, float y, int lanes, bool mer):
            m_x(x),
68          m_y(y),
            m_n_lanes(lanes),
70          m_next_segment(nullptr),
            merge(mer)
72  {
        m_nodes.reserve(m_n_lanes);
74
        ramp_counter = 0;
76      car_passed = false;
        meter = false;
78      period = 0;

80      // can't set nodes if we don't have a theta.
    }
82
    ////////////////////////////////////////////////////////////////////////////
84  /// Returns theta (angle) of RoadSegment, in which direction the segment points

86  float RoadSegment::get_theta() {
        return m_theta;
88  }

90  ////////////////////////////////////////////////////////////////////////////
    /// Returns x position of RoadSegment.
92
    const float RoadSegment::get_x() const{
94      return m_x;
    }
96
    ////////////////////////////////////////////////////////////////////////////
98  /// Returns y position of RoadSegment.

100 const float RoadSegment::get_y() const {
        return m_y;
102 }

104 ////////////////////////////////////////////////////////////////////////////
    /// Returns int number of @param node. E.g. 0 would be the right-most lane.
106 /// Throws exception if we do not find the node in this segment.

108 int RoadSegment::get_lane_number(RoadNode * node) {
        for(int i = 0; i < m_n_lanes; i++){
110         if(node == m_nodes[i]){
                return i;
112         }
        }
```

```
114     throw std::invalid_argument("Node is not in this segment");
}
116
////////////////////////////////////////////////////////////////////////////
118 /// Adds a new car to the segment.

120 void RoadSegment::append_car(Car * car) {
    m_cars.push_back(car);
122 }

////////////////////////////////////////////////////////////////////////////
124 /// Removes car from segment, if car is not in list we do nothing.
126
void RoadSegment::remove_car(Car * car) {
128     unsigned long size = m_cars.size();
    bool found = false;
130     for(int i = 0; i < size; i++){
        if(car == m_cars[i]){
132             m_cars[i] = nullptr;
            found = true;
134         }
    }
136     std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(),m_cars.end(),
    static_cast<Car*>(nullptr));
    m_cars.erase(new_end,m_cars.end());
138
    /*
140     if(!found){
        throw std::invalid_argument("Car is not in this segment.");
142     }
    */
144 }

146 ////////////////////////////////////////////////////////////////////////////
/// Sets theta of RoadSegment according to @param theta.
148
void RoadSegment::set_theta(float theta) {
150     m_theta = theta;
}
152
////////////////////////////////////////////////////////////////////////////
154 /// Automatically populates segment with nodes according to amount of lanes
/// specified and theta specified.
156
void RoadSegment::calculate_and_populate_nodes() {
158     // calculates placement of nodes.
    float total_length = M_LANE_WIDTH*(m_n_lanes-1);
160     float current_length = -total_length/2.0f;

162     for(int i = 0; i < m_n_lanes; i++){
        float x_pos = m_x+current_length*cos(m_theta+(float)M_PI*0.5f);
164         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));
166         current_length += M_LANE_WIDTH;
    }
168 }

170 ////////////////////////////////////////////////////////////////////////////
/// Sets next segment to @param next_segment
172
void RoadSegment::set_next_road_segment(RoadSegment * next_segment) {
174     m_next_segment = next_segment;
}
176
////////////////////////////////////////////////////////////////////////////
178 /// Calculates theta according to next_segment. Throws if m_next_segment is
/// nullptr
180
```

```
void RoadSegment::calculate_theta() {
    if(m_next_segment == nullptr){
        throw std::invalid_argument("Can't calculate theta if next segment is nullptr");
    }
    m_theta = atan2(m_y-m_next_segment->m_y,m_next_segment->m_x-m_x);
}

////////////////////////////////////////////////////////////////////////////
/// Returns node of lane number n. E.g. n=0 is the right-most lane.

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

////////////////////////////////////////////////////////////////////////////
/// Returns all nodes in segment.

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

////////////////////////////////////////////////////////////////////////////
/// Returns next segment

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

////////////////////////////////////////////////////////////////////////////
/// Automatically populates node connections by connecting current node to
/// all nodes in next segment.

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++){
            node->set_next_node(m_next_segment->get_node_pointer(i));
        }
    }
}

////////////////////////////////////////////////////////////////////////////
/// Manually set connection to next segment's node. No guarantee is made
/// on @param from_node_n and @param to_node_n. Can crash if index out of range.

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_next_node(pointy);
}

////////////////////////////////////////////////////////////////////////////
/// Returns amount of lanes in this segment.

const int RoadSegment::get_total_amount_of_lanes() const {
    return m_n_lanes;
}
```

../highway/cppfiles/roadsegment.cpp

B.6 simulation.cpp

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

#include <iostream>
```

```cpp
#include "../headers/traffic.h"
#include "../headers/simulation.h"
#include <cmath>
#include <unistd.h>

////////////////////////////////////////////////////////////////////////////
/// Constructor
/// @param traffic : pointer reference to Traffic, this is to be able to
/// draw traffic outside of this class.
/// @param mutex : mutex thread lock from SFML.
/// @param sim_speed : Simulation speed multiplier, e.g. 10 would mean 10x
/// real time speed. If simulation can not keep up it lowers this.
/// @param framerate : Framerate of simulation, e.g. 60 FPS. This is the
/// time step of the system.
/// @param exit_bool : If user wants to exit this is changed outside of the class.

Simulation::Simulation(Traffic *&traffic, sf::Mutex *&mutex, int sim_speed, int framerate,
    bool *& exit_bool):
        m_mutex(mutex),
        m_traffic(traffic),
        m_exit_bool(exit_bool),
        M_SIM_SPEED(sim_speed),
        M_FRAMERATE(framerate)
{

}

////////////////////////////////////////////////////////////////////////////
/// Runs simulation. If M_SIM_SPEED = 10 , then it simulates 10x1/(M_FRAMERATE)
/// seconds of real time simulation.

void Simulation::update() {
    sf::Clock clock;
    sf::Time time;
    double spawn_counter_0 = 0.0;
    double spawn_counter_1 = 0.0;
    double spawn_counter_2 = 0.0;
    double spawn_counter_3 = 0.0;

    std::vector<double *> counter;
    counter.push_back(&spawn_counter_0);
    counter.push_back(&spawn_counter_1);
    counter.push_back(&spawn_counter_2);
    counter.push_back(&spawn_counter_3);

    while(!*m_exit_bool){
        m_mutex->lock();
        //std::cout << "calculating\n";
        for(int i = 0; i < M_SIM_SPEED; i++){
            //std::cout<< "a\n";
            m_traffic->update(1.0f/(float)M_FRAMERATE);
            //std::cout<< "b\n";
            m_traffic->spawn_cars(counter,1.0f/(float)M_FRAMERATE);
            //m_mutex->lock();
            //std::cout<< "c\n";
            m_traffic->despawn_cars();
            //m_mutex->unlock();
            //std::cout<< "d\n";
        }
        //std::cout << "calculated\n";
        m_mutex->unlock();

        time = clock.restart();
        sf::Int64 acutal_elapsed = time.asMicroseconds();
        double sim_elapsed = (1.0f/(float)M_FRAMERATE)*1000000;

        if(acutal_elapsed < sim_elapsed){
            usleep((useconds_t)(sim_elapsed-acutal_elapsed));
```

```
            m_traffic->m_multiplier = M_SIM_SPEED;
        }
        else{
            m_traffic->m_multiplier = M_SIM_SPEED*(sim_elapsed/acutal_elapsed);
        }
    }
}
```

../highway/cppfiles/simulation.cpp

B.7 traffic.cpp

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

#include <iostream>
#include "../headers/traffic.h"
#include "../headers/car.h"
#include "../headers/road.h"
#include "../headers/util.h"

////////////////////////////////////////////////////////////////////////////////
/// Constructor.

/*
Traffic::Traffic() {
    debug = false;
    if(!m_font.loadFromFile("/Library/Fonts/Andale mono.ttf")){

    }
}
*/

////////////////////////////////////////////////////////////////////////////////
/// Constructor with debug bool, if we want to use debugging information.

Traffic::Traffic(std::vector<bool> bargs, std::vector<float> args) :
    debug(bargs[0]),
    m_aggro(args[0]),
    m_aggro_sigma(args[1]),
    m_spawn_freq(args[2]),
    m_speed(args[3]),

    m_lane_0_spawn_prob(args[4]),
    m_lane_1_spawn_prob(args[5]),
    m_lane_2_spawn_prob(args[6]),
    m_ramp_0_spawn_prob(args[7]),

    m_min_dist_to_car_in_front(args[8]),
    m_min_overtake_dist_trigger(args[9]),
    m_max_overtake_dist_trigger(args[10]),
    m_overtake_done_dist(args[11]),
    m_merge_min_dist(args[12]),
    m_search_radius_around(args[13]),
    m_search_radius_to_car_in_front(args[14]),
    m_ramp_meter_period(args[17]),
    m_ramp_meter(bargs[1]),
    m_multiplier(args[15])
{
    probs.push_back(m_lane_0_spawn_prob);
    probs.push_back(m_lane_1_spawn_prob);
    probs.push_back(m_lane_2_spawn_prob);
    probs.push_back(m_ramp_0_spawn_prob);

    if(!m_font.loadFromFile("/Library/Fonts/Andale mono.ttf")){
```

```
55
    }
57
    Road::shared().ramp_meter_position->ramp_counter = 0;
59  Road::shared().ramp_meter_position->meter = m_ramp_meter;
    Road::shared().ramp_meter_position->period = m_ramp_meter_period;
61
    road_length = 0;
63
    for(RoadSegment * seg : Road::shared().segments()){
65      if(seg->next_segment() != nullptr){
            road_length += Util::distance(seg->get_x(),seg->next_segment()->get_x(),seg->
    get_y(),seg->next_segment()->get_y());
67      }
    }
69  }

71  ////////////////////////////////////////////////////////////////////////////
    /// Copy constructor, deep copies all content.
73
    Traffic::Traffic(const Traffic &ref) :
75      debug(ref.debug),
        m_font(ref.m_font),
77      m_aggro(ref.m_aggro),
        m_aggro_sigma(ref.m_aggro_sigma),
79      m_spawn_freq(ref.m_spawn_freq),
        m_speed(ref.m_speed),
81      m_lane_0_spawn_prob(ref.m_lane_0_spawn_prob),
        m_lane_1_spawn_prob(ref.m_lane_1_spawn_prob),
83      m_lane_2_spawn_prob(ref.m_lane_2_spawn_prob),
        m_ramp_0_spawn_prob(ref.m_ramp_0_spawn_prob),
85      m_min_dist_to_car_in_front(ref.m_min_dist_to_car_in_front),
        m_min_overtake_dist_trigger(ref.m_min_overtake_dist_trigger),
87      m_max_overtake_dist_trigger(ref.m_max_overtake_dist_trigger),
        m_overtake_done_dist(ref.m_overtake_done_dist),
89      m_merge_min_dist(ref.m_merge_min_dist),
        m_search_radius_around(ref.m_search_radius_around),
91      m_search_radius_to_car_in_front(ref.m_search_radius_to_car_in_front),
        m_ramp_meter_period(ref.m_ramp_meter_period),
93      m_ramp_meter(ref.m_ramp_meter),
        road_length(ref.road_length),
95      probs(ref.probs),
        m_multiplier(ref.m_multiplier)
97  {
    // clear values if there are any.
99  for(Car * delete_this : m_cars){
        delete delete_this;
101 }
    m_cars.clear();
103
    // reserve place for new pointers.
105 m_cars.reserve(ref.m_cars.size());

107 // copy values into new pointers
    for(Car * car : ref.m_cars){
109     Car * new_car_pointer = new Car(*car);
        //*new_car_pointer = *car;
111     m_cars.push_back(new_car_pointer);
    }
113
    // values we copied are good, except the car pointers inside the car class.
115 std::map<int,Car*> overtake_this_car;
    std::map<Car*,int> labeling;
117 for(int i = 0; i < m_cars.size(); i++){
        overtake_this_car[i] = ref.m_cars[i]->overtake_this_car;
119     labeling[ref.m_cars[i]] = i;
        m_cars[i]->overtake_this_car = nullptr; // clear copied pointers
121     //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]);
    }
}

////////////////////////////////////////////////////////////////////////////
/// Copy-assignment constructor, deep copies all content and swaps.

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

    std::swap(debug,tmp.debug);
    std::swap(m_font,tmp.m_font);
    std::swap(m_cars,tmp.m_cars);
    std::swap(m_multiplier,tmp.m_multiplier);
    std::swap(probs,tmp.probs);

    return *this;
}

////////////////////////////////////////////////////////////////////////////
/// Destructor, deletes all cars.

Traffic::~Traffic() {
    for(Car * & car : m_cars){
        delete car;
    }
    Traffic::m_cars.clear();
}

////////////////////////////////////////////////////////////////////////////
/// Returns size of car vector

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

////////////////////////////////////////////////////////////////////////////
/// Random generator, returns reference to random generator in order to,
/// not make unneccesary copies.

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

////////////////////////////////////////////////////////////////////////////
/// Logic for spawning cars by looking at how much time has elapsed.
/// @param spawn_counter : culmulative time elapsed
/// @param elapsed : time elapsed for one time step.
/// @param threshold : threshold is set by randomly selecting a poission
/// distributed number.
///
/// Cars that are spawned are poission distributed in time, the speed of the
/// cars are normally distributed according to their aggresiveness.

void Traffic::spawn_cars(std::vector<double*> & spawn_counter, float elapsed) {
    int i = 0;
    std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
```

```
    std::vector<Car *> cars;
    for(int j = 0; j < 4; j++){
        cars.push_back(nullptr);
    }

    for(double * counter : spawn_counter){
        if(*counter < 0){
            std::gamma_distribution<double> dis(probs[i],m_spawn_freq);
            std::normal_distribution<float> aggro(m_aggro,m_aggro_sigma);

            *counter = dis(my_engine());
            float aggressiveness = aggro(my_engine());
            float speed = m_speed*aggressiveness;
            float target = speed;

            if(i < 3){
                Car * new_car = new Car(segments[0],i,speed,target,aggressiveness,
    m_min_dist_to_car_in_front,
                                        m_min_overtake_dist_trigger,m_max_overtake_dist_trigger,
    m_overtake_done_dist,
                                        m_merge_min_dist,m_search_radius_around,
    m_search_radius_to_car_in_front);
                cars[i] = new_car;
            }
            else{
                Car * new_car = new Car(segments[1],0,speed,target,aggressiveness,
    m_min_dist_to_car_in_front,
                                        m_min_overtake_dist_trigger,m_max_overtake_dist_trigger,
    m_overtake_done_dist,
                                        m_merge_min_dist,m_search_radius_around,
    m_search_radius_to_car_in_front);
                cars[i] = new_car;
            }
        }
        i++;
        *counter -= elapsed;
    }

    for(Car * car : cars) {
        if(car != nullptr){
            Car * closest_car_ahead = car->find_closest_car_ahead();

            if(closest_car_ahead == nullptr && closest_car_ahead != car){
                m_cars.push_back(car);
            }
            else{
                float dist = Util::distance_to_car(car,closest_car_ahead);
                if(dist < 10){
                    delete car;
                }
                else if (dist < 150){
                    car->speed() = closest_car_ahead->speed();
                    m_cars.push_back(car);
                }
                else{
                    m_cars.push_back(car);
                }
            }
        }
    }
}

////////////////////////////////////////////////////////////////////////////
/// Despawn @param car

void Traffic::despawn_car(Car *& car) {
    unsigned long size = m_cars.size();
    for(int i = 0; i < size; i++){
```

```
        if(car == m_cars[i]){
            //std::cout << "found " << car << "," << m_cars[i] << std::endl;
            delete m_cars[i];
            m_cars[i] = nullptr;
            //std::cout << car << std::endl;
            m_cars.erase(m_cars.begin()+i);
            car = nullptr;
            //std::cout << "deleted\n";
            break;
        }
    }
}

////////////////////////////////////////////////////////////////////////////
/// Despawn cars that are in the despawn segment.

void Traffic::despawn_cars() {
    //std::cout << "e\n";
    std::map<Car *, bool> to_delete;
    for(Car * car : m_cars){
        for(RoadSegment * seg : Road::shared().despawn_positions()){
            if(car->get_segment() == seg){

                to_delete[car] = true;
                break;
            }
        }
    }

    for(Car * car : m_cars){
        for(auto it : to_delete){
            if(it.first == car->overtake_this_car){
                car->overtake_this_car = nullptr;
            }
        }
    }

    for(Car * & car : m_cars){
        if(to_delete[car]){
            delete car;
            car = nullptr;
        }
    }

    //std::cout << "f\n";
    std::vector<Car*>::iterator new_end = std::remove(m_cars.begin(),m_cars.end(),
    static_cast<Car*>(nullptr));
    m_cars.erase(new_end,m_cars.end());
    //std::cout << "g\n";
}

////////////////////////////////////////////////////////////////////////////
/// Despawn all cars.

void Traffic::despawn_all_cars() {
    for(Car * car : m_cars){
        car->overtake_this_car = nullptr;
    }

    for(Car * & car : m_cars){
        delete car;
        car = nullptr;
    }

    m_cars.clear();
}

////////////////////////////////////////////////////////////////////////////
```

```
319 /// Force places a new car with user specified inputs.
    ///
321 /// \param seg : segment of car
    /// \param node : node of car
323 /// \param vel : (current)velocity of car
    /// \param target : target velocity of car
325 /// \param aggro : agressiveness of car

327 void Traffic::force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target,
        float aggro) {
        Car * car = new Car(seg,node,vel,target,aggro,m_min_dist_to_car_in_front,
329                             m_min_overtake_dist_trigger,m_max_overtake_dist_trigger,
        m_overtake_done_dist,
                             m_merge_min_dist,m_search_radius_around,
        m_search_radius_to_car_in_front);
331     m_cars.push_back(car);
    }
333
    ////////////////////////////////////////////////////////////////////////////
335 /// Updates traffic according by stepping @param elapsed_time seconds in time.

337 void Traffic::update(float elapsed_time) {
        if(m_ramp_meter){
339         float temp = Road::shared().ramp_meter_position->ramp_counter;
            temp += elapsed_time;
341         if(temp >= m_ramp_meter_period){
                temp -= m_ramp_meter_period;
343             Road::shared().ramp_meter_position->car_passed = false;
            }
345         Road::shared().ramp_meter_position->ramp_counter = temp;
        }
347
        for(Car * & car : m_cars){
349         car->avoid_collision(elapsed_time);
        }
351
        for(Car * & car : m_cars){
353         car->update_pos(elapsed_time);
        }
355 }

357 ////////////////////////////////////////////////////////////////////////////
    /// Returns vector of all cars.
359
    std::vector<Car *> Traffic::get_car_copies() const {
361     return m_cars;
    }
363
    ////////////////////////////////////////////////////////////////////////////
365 /// Returns average flow of all cars. Average value of
    /// quotient of current speed divided by target speed for all cars.
367
    float Traffic::get_avg_flow() {
369     float flow = 0;
        for(Car * car : m_cars){
371         flow += car->speed();
        }
373     if(m_cars.empty()){
            return 0;
375     }
        else{
377         return flow/(road_length);
        }
379 }

381 ////////////////////////////////////////////////////////////////////////////
    /// Returns average speeds of all cars in km/h. First entry in vector
383 /// is average speed of all cars, second entry is average speed of cars in left
```

```
/// lane, third entry is average speed of cars in right lane.
385
std::vector<float> Traffic::get_avg_speeds() {
387     std::vector<float> speedy;
        speedy.reserve(3);
389
        float flow = 0;
391     float flow_left = 0;
        float flow_right = 0;
393     float i = 0;
        float j = 0;
395     float k = 0;
        for(Car * car : m_cars){
397         i++;
            flow += car->speed()*3.6f;
399
            if(car->current_segment->get_total_amount_of_lanes() == 2){
401             if(car->current_segment->get_lane_number(car->current_node) == 1){
                    flow_left += car->speed()*3.6f;
403                 j++;
                }
405             else{
                    flow_right += car->speed()*3.6f;
407                 k++;
                }
409         }
        }
411     if(m_cars.empty()){
            return speedy;
413     }
        else{
415         flow =  flow/i;
            flow_left = flow_left/j;
417         flow_right = flow_right/k;
            speedy.push_back(flow);
419         speedy.push_back(flow_left);
            speedy.push_back(flow_right);
421         return speedy;
        }
423 }

425 ////////////////////////////////////////////////////////////////////////////
/// Draws cars (and nodes if debug = true) to @param target, which could
427 /// be a window. Blue cars are cars that want to overtake someone,
/// green cars are driving as fast as they want (target speed),
429 /// red cars are driving slower than they want.

431 void Traffic::draw(sf::RenderTarget &target, sf::RenderStates states) const {
        // print debug info about node placements and stuff
433
        sf::CircleShape circle;
435     circle.setRadius(4.0f);
        circle.setOutlineColor(sf::Color::Cyan);
437     circle.setOutlineThickness(1.0f);
        circle.setFillColor(sf::Color::Transparent);
439
        sf::Text segment_n;
441     segment_n.setFont(m_font);
        segment_n.setFillColor(sf::Color::Black);
443     segment_n.setCharacterSize(14);

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

449     if(debug){
            int i = 0;
451
```

```
        for(RoadSegment * segment : Road::shared().segments()){
            for(RoadNode * node : segment->get_nodes()){
                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);
                for(RoadNode * connected_node : node->get_nodes_from_me()){
                    line[1].position = sf::Vector2f(connected_node->get_x()*2,connected_node
    ->get_y()*2);
                    target.draw(line,states);
                }
                target.draw(circle,states);

            }
            segment_n.setString(std::to_string(i));
            segment_n.setPosition(sf::Vector2f(segment->get_x()*2+4,segment->get_y()*2+4));
            target.draw(segment_n,states);

            i++;
        }
    }
    if(m_ramp_meter){
        RoadSegment * meter = Road::shared().ramp_meter_position;
        circle.setPosition(sf::Vector2f(meter->get_x()*2+4-25,meter->get_y()*2-4));
        circle.setOutlineColor(sf::Color::Black);
        if(meter->ramp_counter > m_ramp_meter_period*0.5f){
            circle.setFillColor(sf::Color::Green);

        }
        else{
            circle.setFillColor(sf::Color::Red);
        }
        target.draw(circle,states);
        circle.setOutlineColor(sf::Color::Cyan);
        circle.setFillColor(sf::Color::Transparent);
    }

    // one rectangle is all we need :)
    sf::RectangleShape rectangle;
    rectangle.setSize(sf::Vector2f(9.4,3.4));
    //rectangle.setFillColor(sf::Color::Green);
    rectangle.setOutlineColor(sf::Color::Black);
    rectangle.setOutlineThickness(2.0f);

    //std::cout << "start drawing\n";
    for(Car * car : m_cars){
        if(car != nullptr){
            //std::cout << "a\n";
            rectangle.setPosition(car->x_pos()*2,car->y_pos()*2);
            rectangle.setRotation(car->theta()*(float)360.0f/(-2.0f*(float)M_PI));
            unsigned int colval = (unsigned int)std::min(255.0f*(car->speed()/car->
    target_speed()),255.0f);
            sf::Uint8 colorspeed = static_cast<sf::Uint8> (colval);
            //std::cout << "b\n";
            if(car->overtake_this_car != nullptr){
                rectangle.setFillColor(sf::Color(255-colorspeed,0,colorspeed,255));
            }
            else{
                rectangle.setFillColor(sf::Color(255-colorspeed,colorspeed,0,255));
            }

            target.draw(rectangle,states);

            // this caused crash earlier
            if(car->heading_to_node!=nullptr && debug){
                // print debug info about node placements and stuff
                circle.setOutlineColor(sf::Color::Red);
                circle.setOutlineThickness(2.0f);
                circle.setFillColor(sf::Color::Transparent);
```

```
                circle.setPosition(sf::Vector2f(car->current_node->get_x()*2-4,car->
    current_node->get_y()*2-4));
                target.draw(circle,states);
                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));
                target.draw(circle,states);
            }
        }
    }
    //std::cout << "stop drawing\n";
}

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

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

    float fps = 1.0f/elapsed.asSeconds();
    unsigned long amount_of_cars = n_of_cars();
    float flow = get_avg_flow();
    std::vector<float> spe = get_avg_speeds();
    std::string speedy = std::to_string(fps).substr(0,2) +
                         " fps, ncars: " + std::to_string(amount_of_cars) + "\n"
                         + "avg_flow: " + std::to_string(flow).substr(0,4) +"\n"
                         + "avg_speed: " + std::to_string(spe[0]).substr(0,5) +"km/h\n"
                         + "left_speed: " + std::to_string(spe[1]).substr(0,5) +"km/h\n"
                         + "right_speed: " + std::to_string(spe[2]).substr(0,5) +"km/h\n"
                         + "sim_multiplier: " + std::to_string(m_multiplier).substr(0,3) + "
    x";
    text.setString(speedy);
    text.setPosition(0,0);
    text.setFillColor(sf::Color::Black);
    text.setFont(m_font);
}
```

../highway/cppfiles/traffic.cpp

B.8 unittests.cpp

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

#include "unittests.h"
#include "road.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_nodes_from_me();
            std::cout << "node" << node <<" has connections:" <<  std::endl;
            for(RoadNode * pointy : connections){
```

```
                std::cout << pointy << std::endl;
            }
        }
        i++;
        m_traffic->force_place_car(seg,seg->get_nodes()[0],1,1,0.01);
        std::cout << "placed car" << std::endl;
    }
    std::cout << "Placement tests passed\n";
}

void Tests::delete_cars_test() {
    std::vector<Car*> car_copies = m_traffic->get_car_copies();

    for(Car * car : car_copies){
        std::cout << car << std::endl;
        usleep(100);
        m_mutex->lock();
        std::cout << "deleting car\n";
        //usleep(100000);
        //std::cout << "Removing car " << car << std::endl;
        m_traffic->despawn_car(car);
        m_mutex->unlock();
        std::cout << car << std::endl;
    }
    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();
    }
    */
}

void Tests::placement_test_2() {
    std::cout << "Starting placement tests 2\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_nodes_from_me();
            std::cout << "node" << node <<" has connections:" <<  std::endl;
            for(RoadNode * pointy : connections){
                std::cout << pointy << std::endl;
            }
            m_traffic->force_place_car(seg,node,1,1,0.1);
            std::cout << "placed car"  << std::endl;
        }
        i++;

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

```
}

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

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

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


// do all tests
void Tests::run_all_tests() {
    usleep(2000000);
    placement_test();
    delete_cars_test();
    run_one_car();
    placement_test_2();
    placement_test_3();

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

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

../highway/cppfiles/unittests.cpp

B.9 util.cpp

```
//
// Created by Carl Schiller on 2019-03-04.
//

#include "../headers/util.h"
#include <sstream>
#include <string>
#include <cmath>

////////////////////////////////////////////////////////////////////////////
/// Splits @param str by @param delim, returns vector of tokens obtained.

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;
}

////////////////////////////////////////////////////////////////////////////
/// Returns true if @param a is behind @param b, else false

bool Util::is_car_behind(Car * a, Car * b){
    if(a!=b){
```

```
28          float theta_to_car_b = atan2(a->y_pos()-b->y_pos(),b->x_pos()-a->x_pos());
30          float theta_difference = get_min_angle(a->theta(),theta_to_car_b);
            return theta_difference < M_PI*0.45;
32      }
        else{
34          return false;
        }

36  }

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

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

48      //int node_0_int = inspecting_segment->get_lane_number(node_0);
        int node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
50
        while(!node_1->get_nodes_from_me().empty()){
52          for(Car * car : inspecting_segment->m_cars){
                if(car == b){
54                  // place logic for evaluating if we cross cars here.
                    // heading to same node, else return false
56                  return node_1 == b->heading_to_node;
                }
58          }

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

64          // if we are at say, 2 lanes and heading to 2 lanes, keep previous lane numbering.
            if(inspecting_segment->get_total_amount_of_lanes() == node_1->get_nodes_from_me().
    size()){
66              node_1 = node_1->get_nodes_from_me()[node_1_int];
            }
68              // if we get one option, stick to it.
            else if(node_1->get_nodes_from_me().size() == 1){
70              node_1 = node_1->get_nodes_from_me()[0];

72          }
                // we merge from 3 to 2.
74          else if(inspecting_segment->get_total_amount_of_lanes() == 3 && inspecting_segment->
    merge){
                node_1 = node_1->get_nodes_from_me()[std::max(node_1_int-1,0)];
76          }

78          node_1_int = node_1->get_parent_segment()->get_lane_number(node_1);
        }
80
        return false;
82  }

84  /*

86  bool Util::merge_helper(Car *a, int merge_to_lane) {
        RoadSegment * seg = a->current_segment;
88      for(Car * car : seg->m_cars){
            if(car != a){
90              float delta_speed = a->speed()-car->speed();
                if(car->heading_to_node == a->current_node->get_nodes_from_me()[merge_to_lane]
    && delta_speed < 0){
92                  return true;
```

```
            }
94          }
        }
96      return false;
}
98
*/
100
/*
102
// this works only if a's heading to is b's current segment
104 bool Util::is_cars_in_same_lane(Car *a, Car *b) {
    return a->heading_to_node == b->current_node;
106 }

108 */

110 /*
float Util::distance_to_line(const float theta, const float x, const float y){
112     float x_hat,y_hat;
    x_hat = cos(theta);
114     y_hat = -sin(theta);

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

120     return dist;
}
122 */

124 /*
float Util::distance_to_proj_point(const float theta, const float x, const float y){
126     float x_hat,y_hat;
    x_hat = cos(theta);
128     y_hat = -sin(theta);
    float proj_x = (x*x_hat+y*y_hat)*x_hat;
130     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)));
132
    return dist;
134 }
*/
136
////////////////////////////////////////////////////////////////////////////
138 /// Returns distance between @param a and @param b.

140 float Util::distance_to_car(Car * a, Car * b){
    if(a == nullptr || b == nullptr){
142         throw std::invalid_argument("Can't calculate distance if cars are nullptrs");
    }
144
    float delta_x = a->x_pos()-b->x_pos();
146     float delta_y = b->y_pos()-a->y_pos();

148     return sqrt(abs(pow(delta_x,2.0f))+abs(pow(delta_y,2.0f)));
}
150
/*
152
Car * Util::find_closest_radius(std::vector<Car> &cars, const float x, const float y){
154     Car * answer = nullptr;

156     float score = 100000;
    for(Car & car : cars){
158         float distance = sqrt(abs(pow(car.x_pos()-x,2.0f))+abs(pow(car.y_pos()-y,2.0f)));
        if(distance < score){
160             score = distance;
```

```
            answer = &car;
        }
    }

    return answer;
}

*/

////////////////////////////////////////////////////////////////////////////
/// Returns min angle between @param ang1 and @param ang2

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;
}

////////////////////////////////////////////////////////////////////////////
/// Returns distance between two points in 2D.

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)));
}
```

../highway/cppfiles/util.cpp