Final Project, SI1336

Carl Schiller, 9705266436

March 4, 2019

Abstract

Contents

1 Introduction

2 Method

3 Result

4 Discussion

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;

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

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

    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();
};

#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);

```

```cpp
    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

```cpp
//
// 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

    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;

public:
    Traffic();
    explicit Traffic(bool debug);
    ~Traffic();
    Traffic(const Traffic&); // rule of three
    Traffic& operator=(const Traffic&); // rule of three

    unsigned long n_of_cars();
    void spawn_cars(double & spawn_counter, float elapsed, double & threshold);
    void despawn_cars();
    void despawn_all_cars();
    void despawn_car(Car*& car);
    void force_place_car(RoadSegment * seg, RoadNode * node, float vel, float target, float aggro);


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

#endif //HIGHWAY_TRAFFIC_H
```

../highway/headers/traffic.h

A.7 unittests.h

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


#ifndef HIGHWAY_UNITTESTS_H
#define HIGHWAY_UNITTESTS_H

////////////////////////////////////////////////////////////////////////////
//                                                                        //
// 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 ”../headers/util.h”

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

Car::Car() = default;

//////////////////////////////////////////////////////////////////////////////
/// 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 aggressivness):
        m_speed(vel),
        m_aggressiveness(aggressivness),
        m_target_speed(target_speed),
        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);

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

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

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

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

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

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

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

////////////////////////////////////////////////////////////////////////////////
/// Destructor for car.

Car::˜Car(){
    if(this->current_segment != nullptr){
        this->current_segment->remove_car(this); // remove this pointer shit
    }

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

}

////////////////////////////////////////////////////////////////////////////////
/// 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
        }
        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,15.0f);
        }
        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,15.0f);
        }

        if(!can_merge){
            break;
        }
    }

    if(current_segment->merge){
        if(current_lane == 0 && connections[0]->get_parent_segment()->get_total_amount_of_lanes() != 2){
            if(can_merge){
                heading_to_node = connections[1];
            }
            else{
                heading_to_node = connections[0];
            }
        }
        else if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
            current_lane = std::max(current_lane-1,0);
            heading_to_node = connections[current_lane];
        }
        else{
            heading_to_node = connections[current_lane];
        }
    }
        // if we are in start section
    else if(current_segment->get_total_amount_of_lanes() == 3){
        if(connections.size() == 1){
            heading_to_node = connections[0];
        }
        else{
            heading_to_node = connections[current_lane];
        }
    }
        // if we are in middle section
    else if(current_segment->get_total_amount_of_lanes() == 2){
        // normal way
        if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
            // check if we want to overtake car in front
            do_we_want_to_overtake(closest_car,current_lane);

            // commited to overtaking
            if(overtake_this_car != nullptr){
                if(current_lane != 1){
                    if(can_merge){
                        heading_to_node = connections[1];
                    }
                    else{
                        heading_to_node = connections[current_lane];
                    }
                }
                else{
                    heading_to_node = connections[current_lane];
                }

            }
                // merge back if overtake this car is nullptr.
            else{
                if(can_merge){
                    heading_to_node = connections[0];
                }
                else{
                    heading_to_node = connections[current_lane];
                }
            }

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

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

void Car::do_we_want_to_overtake(Car * & closest_car, int & current_lane) {
    //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 > 10 && delta_distance < 40 && (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) > 30)){
            overtake_this_car = nullptr;
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
/// Function to accelerate this car.

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

    if(m_speed < target*0.75){
        d_vel = m_aggressiveness*elapsed*2.0f;
```

```cpp
    }
    else{
        d_vel = m_aggressiveness*(target-m_speed)*4*elapsed*2.0f;
    }

    m_speed += d_vel;
}

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

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

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

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

        if (radius_to_car < ideal && delta_speed < 0 && radius_to_car > min_distance) {
            m_speed -= std::max(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::max(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);
        }
        else {
            accelerate(delta_t);
        }

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

                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)){
                        accelerate(delta_t);
                    }
                    else{
                        m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*delta_t);
                    }
                }
                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)){
                        m_speed -= std::max(std::max((ideal-delta_dist)*0.5f,0.0f),10.0f*delta_t);
                    }
                    else{
                        accelerate(delta_t);
                    }
                }
            }
        }
        else{

        }
    }
    else{
        accelerate(delta_t);
    }

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


}

//////////////////////////////////////////////////////////////////////////////
/// Helper function to find closest car in the same lane ahead of this car.
/// Returns a car if found, otherwise nullptr.

Car* Car::find_closest_car_ahead() {
    float search_radius = 50;
    std::map<RoadNode*,bool> visited;
    std::list<RoadNode*> queue;

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

    Car* answer = nullptr;

    float shortest_distance = 10000000;

    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){
                        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){
                            shortest_distance = radius;
                            answer = car;
                        }

                    }
                }

                // push in new nodes in front of list.
                for(RoadNode * node : next_node->get_nodes_from_me()){
                    queue.push_front(node);
                }
            }
        }
    }
    return answer;
}

//////////////////////////////////////////////////////////////////////////////
/// Searches for cars around this car in a specified radius. Note that
/// search radius is the radius to RoadNodes, and not surrounding cars.
/// Returns a map of cars the function has found.

std::map<Car *,bool> Car::find_cars_around_car() {
    const float search_radius = 40;
    std::map<RoadNode*,bool> visited;
    std::list<RoadNode*> queue;

    for(RoadNode * node : (this->current_segment->get_nodes())){
        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;
}

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

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

////////////////////////////////////////////////////////////////////////////////
/// Returns theta of car, the direction of the car. Defined in radians as a
/// mathematitan would define angles.

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

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

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

```
}
```

../highway/cppfiles/car.cpp

B.2 main.cpp

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

sf::Mutex mutex;

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

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

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

    }

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

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

    bool exit_bool = false;

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

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

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

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

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

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

            copy.get_info(debug_info,elapsed);

            //mutex.unlock();
            window.draw(debug_info);

            window.display();
        }

    }
    else{


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

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

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

            // check all the window's events that were triggered since the last iteration of the loop
            sf::Event event;
            while (window.pollEvent(event))
            {
                // "close requested" event: we close the window
                if (event.type == sf::Event::Closed){
                    //thread.terminate();
                    window.close();
                    thread.terminate();
                    delete traffic;
                }
            }
            //Traffic copy = tests.m_traffic; // deep copy it
            sf::Time elapsed = clock.restart();

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

            mutex.lock();
            copy = *traffic;
            mutex.unlock();

            window.draw(background);
            window.draw(copy);

            copy.get_info(debug_info,elapsed);
            window.draw(debug_info);

            window.display();
        }
    }

    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;
    }
```

```cpp

    // 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{
                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"){
                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"){
                m_segments[i]->set_all_node_pointers_to_next_segment();
            }
            else if(road_vector[i][4] == "merge"){
                m_segments[i]->set_all_node_pointers_to_next_segment();
            }

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

////////////////////////////////////////////////////////////////////////////////
/// 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

```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);

    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) :
        m_x(x),
        m_y(y),
        m_theta(theta),
        m_n_lanes(lanes),
        m_next_segment(nullptr)
{
    m_nodes.reserve(m_n_lanes);

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

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

RoadSegment::RoadSegment(float x, float y, int lanes, bool mer):
        m_x(x),
        m_y(y),
        m_n_lanes(lanes),
        m_next_segment(nullptr),
        merge(mer)
{
    m_nodes.reserve(m_n_lanes);

    // can't set nodes if we don't have a theta.
}

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

```
/// Returns theta (angle) of RoadSegment, in which direction the segment points

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

////////////////////////////////////////////////////////////////////////////
/// Returns x position of RoadSegment.

const float RoadSegment::get_x() const{
    return m_x;
}

////////////////////////////////////////////////////////////////////////////
/// Returns y position of RoadSegment.

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

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

int RoadSegment::get_lane_number(RoadNode * node) {
    for(int i = 0; i < m_n_lanes; i++){
        if(node == m_nodes[i]){
            return i;
        }
    }
    throw std::invalid_argument("Node is not in this segment");
}

////////////////////////////////////////////////////////////////////////////
/// Adds a new car to the segment.

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

////////////////////////////////////////////////////////////////////////////
/// Removes car from segment, if car is not in list we throw exception

void RoadSegment::remove_car(Car * car) {
    unsigned long size = m_cars.size();
    bool found = false;
    for(int i = 0; i < size; i++){
        if(car == m_cars[i]){
            m_cars[i] = nullptr;
            found = true;
        }
    }
    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());

    if(!found){
        throw std::invalid_argument("Car is not in this segment.");
    }
}

////////////////////////////////////////////////////////////////////////////
/// Sets theta of RoadSegment according to @param theta.

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

////////////////////////////////////////////////////////////////////////////
/// Automatically populates segment with nodes according to amount of lanes
/// specified and theta specified.

void RoadSegment::calculate_and_populate_nodes() {
    // calculates placement of nodes.
    float total_length = M_LANE_WIDTH*(m_n_lanes-1);
    float current_length = -total_length/2.0f;

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

////////////////////////////////////////////////////////////////////////////////
/// Sets next segment to @param next_segment

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

////////////////////////////////////////////////////////////////////////////////
/// Calculates theta according to next_segment. Throws if m_next_segment is
/// nullptr

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>
#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;
    double threshold = 0.0;

    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(spawn_counter,1.0f/(float)M_FRAMERATE, threshold);
            //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 "../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/Arial.ttf")){
        //crash
    }
}

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

Traffic::Traffic(bool debug) : debug(debug){
    if(!m_font.loadFromFile("/Library/Fonts/Arial.ttf")){
        //crash
    }
}

////////////////////////////////////////////////////////////////////////////////
/// Copy constructor, deep copies all content.

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

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

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

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

    for(auto it : from_to){
        m_cars[it.first]->overtake_this_car = m_cars[it.second];
        //m_cars[it.second]->want_to_overtake_me.push_back(m_cars[it.first]);
    }
}

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

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

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

    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(double & spawn_counter, float elapsed, double & threshold) {
    spawn_counter += elapsed;
    if(spawn_counter > threshold){
        std::exponential_distribution<double> dis(5);
        std::normal_distribution<float> aggro(1.0f,0.2f);
        float sp = 30.0f;
        std::uniform_real_distribution<float> lane(0.0f,1.0f);
        std::uniform_real_distribution<float> spawn(0.0f,1.0f);

        threshold = dis(my_engine());
        float aggressiveness = aggro(my_engine());
        float speed = sp*aggressiveness;
        float target = speed;

        spawn_counter = 0;
        float start_lane = lane(my_engine());
        float spawn_pos = spawn(my_engine());

        std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
        RoadSegment * seg;
        Car * new_car;
        if(spawn_pos < 0.95){
            seg = segments[0];
            if(start_lane < 0.457){
                new_car = new Car(seg,2,speed,target,aggressiveness);
            }
            else if(start_lane < 0.95){
                new_car = new Car(seg,1,speed,target,aggressiveness);
            }
            else{
                new_car = new Car(seg,0,speed,target,aggressiveness);
            }
        }
        else{
            seg = segments[1];
            new_car = new Car(seg,0,speed,target,aggressiveness);
        }

        Car * closest_car_ahead = new_car->find_closest_car_ahead();

        if(closest_car_ahead == nullptr && closest_car_ahead != new_car){
            m_cars.push_back(new_car);
        }
        else{
            float dist = Util::distance_to_car(new_car,closest_car_ahead);
            if(dist < 10){
                delete new_car;
            }
            else if (dist < 150){
                new_car->speed() = closest_car_ahead->speed();
                m_cars.push_back(new_car);
            }
            else{
                m_cars.push_back(new_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 (by creating a new traffic object).

void Traffic::despawn_all_cars() {
    *this = Traffic();
}

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

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_cars.push_back(car);
}

////////////////////////////////////////////////////////////////////////////////
/// Updates traffic according by stepping @param elapsed_time seconds in time.

void Traffic::update(float elapsed_time) {
    for(Car * & car : m_cars){
        car->avoid_collision(elapsed_time);
    }

    for(Car * & car : m_cars){
        car->update_pos(elapsed_time);
    }
}

////////////////////////////////////////////////////////////////////////////////
/// Returns vector of all cars.

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

////////////////////////////////////////////////////////////////////////////////
/// Returns average flow of all cars. Average value of
/// quotient of current speed divided by target speed for all cars.

float Traffic::get_avg_flow() {
    float flow = 0;
    float i = 0;
    for(Car * car : m_cars){
        i++;
        flow += car->speed()/car->target_speed();
    }
    if(m_cars.empty()){
        return 0;
    }
    else{
        return flow/i;
    }
}

////////////////////////////////////////////////////////////////////////////////
/// Returns average speeds of all cars in km/h. First entry in vector
/// 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.

std::vector<float> Traffic::get_avg_speeds() {
    std::vector<float> speedy;
    speedy.reserve(3);

    float flow = 0;
    float flow_left = 0;
    float flow_right = 0;
    float i = 0;
    float j = 0;
    float k = 0;
    for(Car * car : m_cars){
        i++;
        flow += car->speed()*3.6f;

        if(car->current_segment->get_total_amount_of_lanes() == 2){
            if(car->current_segment->get_lane_number(car->current_node) == 1){
                flow_left += car->speed()*3.6f;
                j++;
            }
            else{
                flow_right += car->speed()*3.6f;
                k++;
            }
        }
    }
    if(m_cars.empty()){
        return speedy;
    }
    else{
        flow =  flow/i;
        flow_left = flow_left/j;
        flow_right = flow_right/k;
        speedy.push_back(flow);
        speedy.push_back(flow_left);
        speedy.push_back(flow_right);
        return speedy;
    }
}

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

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

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

    sf::Text segment_n;
    segment_n.setFont(m_font);
    segment_n.setFillColor(sf::Color::Black);
    segment_n.setCharacterSize(14);

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

    if(debug){
        int i = 0;
        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++;
        }
    }

    // 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){
        //std::cout << "drawing" << car << std::endl;
        if(car != nullptr){
            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);

            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);
            }
        }
    }
}

///////////////////////////////////////////////////////////////////////////////
/// 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 "../headers/unittests.h"
#include "../headers/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){
        float theta_to_car_b = atan2(a->y_pos()-b->y_pos(),b->x_pos()-a->x_pos());
        float theta_difference = get_min_angle(a->theta(),theta_to_car_b);
        return theta_difference < M_PI*0.45;
    }
    else{
        return false;
    }

}

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

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

    //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);

    while(!node_1->get_nodes_from_me().empty()){
        for(Car * car : inspecting_segment->m_cars){
            if(car == b){
                // place logic for evaluating if we cross cars here.
                // heading to same node, else return false
                return node_1 == b->heading_to_node;
            }
        }

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

        // 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()){
            node_1 = node_1->get_nodes_from_me()[node_1_int];
        }
            // if we get one option, stick to it.
        else if(node_1->get_nodes_from_me().size() == 1){
            node_1 = node_1->get_nodes_from_me()[0];

        }
            // we merge from 3 to 2.
        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)];
        }

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

/*

bool Util::merge_helper(Car *a, int merge_to_lane) {
    RoadSegment * seg = a->current_segment;
    for(Car * car : seg->m_cars){
        if(car != a){
            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){
                return true;
            }
        }
    }
    return false;
}

*/

/*

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

*/

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

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

    return dist;
}
*/

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

    return dist;
}
*/

////////////////////////////////////////////////////////////////////////////////
/// Returns distance between @param a and @param b.

float Util::distance_to_car(Car * a, Car * b){
    if(a == nullptr || b == nullptr){
        throw std::invalid_argument(”Can’t calculate distance if cars are nullptrs”);
    }

    float delta_x = a->x_pos()-b->x_pos();
    float delta_y = b->y_pos()-a->y_pos();

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

/*

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

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

    return answer;
}

*/

////////////////////////////////////////////////////////////////////////////////
/// 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