Final Project, SI1336

Carl Schiller, 9705266436

January 20, 2019

Abstract

Contents

1 Introduction

2 Method

3 Result

4 Discussion

A Highway

A.1 Header files

A.1.1 traffic.h

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

#ifndef HIGHWAY_TRAFFIC_H
#define HIGHWAY_TRAFFIC_H



class RoadSegment;

class Car;

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

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


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

    constexpr static float M_LANE_WIDTH = 4.0f;

    std::vector<RoadNode*> m_nodes; // OWNERSHIP
    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 *);
};

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

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

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

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

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

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

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

public:
    Car();
    ~Car();
    Car(RoadSegment * spawn_point, int lane, float vel, float target_speed, float agressivness);
    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;
    bool is_getting_overtaken;
    //void remove_pointer(Car * car);

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

    float x_pos();
    float y_pos();

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

    RoadSegment * get_segment();
};


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 bool is_cars_in_same_lane(Car*a,Car*b);
    static bool merge_helper(Car*a, int merge_to_lane);
    static float distance_to_line(float theta,  float x,  float y);
    static float distance_to_proj_point(float theta,  float x,  float y);
    static float distance_to_car(Car * a, Car * b);
    static Car * find_closest_radius(std::vector<Car> &cars, float x, float y);
    static float get_min_angle(float ang1, float ang2);
    static float distance(float x1, float x2, float y1, float y2);
};

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

    //void update_speed(int i, float & elapsed_time);
    //float get_theta(float xpos, float ypos, float speed, float current_theta, bool & lane_switch);
public:
    Traffic();
    ~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();

private:
    virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
public:
    void get_info(sf::Text & text, sf::Time &elapsed);
    double m_multiplier;
};

struct car_deleter{
    void operator()(Car*& car);
};

#endif //HIGHWAY_TRAFFIC_H
```

../highway/traffic.h

A.1.2 window.h

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

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

```

```
#ifndef HIGHWAY_WINDOW_H
#define HIGHWAY_WINDOW_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/window.h

A.1.3 unittests.h

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

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

class Tests{
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/unittests.h

A.2 Source files

A.2.1 traffic.cpp

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

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

Car::Car() = default;
```

```
Car::Car(RoadSegment *spawn_point, int lane, float vel, float target_speed, float aggressivness):
    m_speed(vel), m_target_speed(target_speed), m_aggressiveness(aggressivness)
{
    current_segment = spawn_point;
    is_getting_overtaken = false;
    overtake_this_car = nullptr;

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

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

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

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

    m_breaking = false;
}

Car::Car(RoadSegment *spawn_point, RoadNode *lane, float vel, float target_speed, float agressivness) :
m_speed(vel), m_target_speed(target_speed), m_aggressiveness(agressivness)
{
    current_segment = spawn_point;

    overtake_this_car = nullptr;
    is_getting_overtaken = false;

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

    if(!current_node->get_connections().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{
        //std::cout << ”aa\n”;
        heading_to_node = nullptr;
        m_theta = 0;
        m_dist_to_next_node = 0;
    }

    m_breaking = false;
}


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

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

}

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

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

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

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

            if(current_segment->merge){
                if(current_lane == 0 && connections[0]->get_parent_segment()->get_total_amount_of_lanes() != 2){
                    if(!Util::merge_helper(this,1)){
                        heading_to_node = connections[1];
                    }
                    else{
                        heading_to_node = connections[0];
                    }
                }
                else if(connections[0]->get_parent_segment()->get_total_amount_of_lanes() == 2){
                    current_lane = std::max(current_lane-1,0);
                    heading_to_node = connections[current_lane];
                }
                else{
                    heading_to_node = connections[current_lane];
                }
            }
            // 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){
                    //see if we want to overtake car.
                    Car * closest_car = find_closest_car();

                    if(closest_car != nullptr && this->overtake_this_car == nullptr){
                        float delta_speed = closest_car->speed()-this->speed();
                        float delta_distance = Util::distance_to_car(this,closest_car);
                        if(delta_distance/delta_speed > -5 && delta_speed < 2){
                            this->overtake_this_car = closest_car;
                        }
                    }

                    if(this->overtake_this_car != nullptr){
                        if(current_lane != 1){
                            if(!Util::merge_helper(this,1)){
                                heading_to_node = connections[1];
                            }
                            else{
                                heading_to_node = connections[current_lane];
                            }
                        }
                        else{
                            heading_to_node = connections[current_lane];
                        }
                        /*
                        else{
                            if((!Util::is_car_behind(this,this->overtake_this_car) && Util::distance_to_car(this,this->overtake_this_car) > 10) || Util::distance_to_car(this,this->overtake_this_car) > 40){
                                for(int i = 0; i< overtake_this_car->want_to_overtake_me.size(); i++){
                                    if(this == overtake_this_car->want_to_overtake_me[i]){
                                        overtake_this_car->want_to_overtake_me[i] = nullptr;
                                    }
                                }
                                std::vector<Car*>::iterator new_end = std::remove(overtake_this_car->want_to_overtake_me.begin(),overtake_this_car->want_to_overtake_me.end(),static_cast<Car*>(nullptr));
                                overtake_this_car->want_to_overtake_me.erase(new_end,overtake_this_car->want_to_overtake_me.end());

                                this->overtake_this_car = nullptr;
                            }

                            heading_to_node = connections[current_lane];
                        }
                         */
                    }
                    // merge back if overtake this car is nullptr.
                    else{
                        if(!Util::merge_helper(this,0)){
                            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];
            }

            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);

        }
    }
}

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;
    }
    else{
        d_vel = m_aggressiveness*(target-m_speed)*4*elapsed;
    }

    m_speed += d_vel;
}

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);
    float detection_distance = m_speed*4.0f;
    //std::cout << ”boop1\n”;
    Car * closest_car = find_closest_car();
    //std::cout << ”boop2\n”;
    float radius_to_car = 1000;
    float delta_speed = 0;

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

        if (radius_to_car < ideal) {
            m_speed -= std::min(abs(delta_speed)*delta_t+(ideal-radius_to_car)*0.5f, 10.0f * delta_t);
        }
        else if(radius_to_car < detection_distance && delta_speed < 0){
            m_speed -= std::min(
                    abs(pow(delta_speed, 2.0f)) * pow(ideal * 0.25f / radius_to_car, 2.0f) * m_aggressiveness * 0.02f,
                    10.0f * delta_t);
        }
        else {
            accelerate(delta_t);
        }
    }
    else{
        accelerate(delta_t);
    }
    //std::cout << ”boop3\n”;
    if(m_speed < 0){
        m_speed = 0;
    }


}

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

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

    Car* answer = nullptr;
    float best_radius = 1000;
    while(!queue.empty()){
        RoadNode * next_node = queue.back(); // get last element
        queue.pop_back(); // remove element
        RoadSegment * next_segment = nullptr;
        if(next_node != nullptr) {
            next_segment = next_node->get_parent_segment();
        }

        if(next_segment != nullptr){
            if(!visited[next_segment] && Util::distance(origin->get_x(),next_segment->get_x(),origin->
    get_y(),next_segment->get_y()) < search_radius){
                visited[next_segment] = true;
                for(Car * car : next_segment->m_cars){
                    if(this != car){
                        float radius = Util::distance_to_car(this,car);
                        if(Util::is_car_behind(this,car) && radius < best_radius){
                            //std::cout << ”a\n”;
                            if(Util::will_car_paths_cross(this,car)){
                                best_radius = radius;
                                answer = car;
                            }
                            //std::cout << ”b\n”;
                        }

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

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

    return x_position;
}
```

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

    return y_position;
}

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

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

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

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




RoadNode::RoadNode() = default;

RoadNode::~RoadNode() = default;

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

void RoadNode::set_pointer(RoadNode * next_node) {
    m_connecting_nodes.push_back(next_node);
}

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

std::vector<RoadNode*> & RoadNode::get_connections() {
    return m_connecting_nodes;
}

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

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

float RoadNode::get_theta(RoadNode* node) {
    for(RoadNode * road_node : m_connecting_nodes){
        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”);
}

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

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

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();
}

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(lanes);

    calculate_and_populate_nodes();
}

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

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

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

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

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

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”);
}

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

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.”);
    }
```

```
}

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

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;
    }
}

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

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

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

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

RoadSegment* RoadSegment::next_segment() {
    return m_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_pointer(m_next_segment->get_node_pointer(i));
        }
    }
}

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

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

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

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

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{
            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;
}

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

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

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

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

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

```
}

//TODO: Bug here
// true if car paths cross
bool Util::will_car_paths_cross(Car *a, Car *b) {
    //begin with drawing a straight line
    std::list<RoadSegment*> segments;
    std::list<RoadNode*> nodes;

    segments.push_back(a->get_segment());
    nodes.push_back(a->current_node);
    // make sure this is not true
    if(a->heading_to_node == nullptr){
        return false;
    }
    nodes.push_back(a->heading_to_node);

    float dist_between_segments = Util::distance(a->current_segment->get_x(),b->current_segment->get_x(),
                                                 a->current_segment->get_y(),b->current_segment->get_y());
    bool found_b = false;

    while(!found_b){
        //std::cout << "a1\n";
        for(Car * car : segments.back()->m_cars){
            if(car == b){
                found_b = true;
            }
        }

        if(!found_b){
            if(nodes.back() == nullptr){
                return false;
            }
            //std::cout << nodes.back() << std::endl;
            segments.push_back(nodes.back()->get_parent_segment());
            int seg0_lane_n = segments.back()->get_total_amount_of_lanes();
            int lane = segments.back()->get_lane_number(nodes.back());
            std::vector<RoadNode*> node_choices = nodes.back()->get_connections();

            // if seg0==seg1 we keep lane numbering.
            if(node_choices.size() == seg0_lane_n){
                nodes.push_back(node_choices[lane]);
            }
                // if we only have one choice, stick to it
            else if(node_choices.size() == 1){
                lane = 0;
                nodes.push_back(node_choices[lane]);
            }
                // last merge
            else if(node_choices.size() == 2){
                lane = std::min(std::max(lane-1,0),0);
                nodes.push_back(node_choices[lane]);
            }
            else if(node_choices.empty()){
                return false;
            }

        }
        //std::cout << "hej3\n";
        float delta_dist = dist_between_segments - distance(b->current_segment->get_x(),segments.back()->
    get_x(),
                                                            b->current_segment->get_y(),segments.back()->
    get_y());
        if(delta_dist <0){
            return false;
        }
    }

    //std::cout << "hej4\n";

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

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

    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_connections()[merge_to_lane] && delta_speed <
    0){
                return true;
            }
        }
    }
    return false;
}

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

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

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

    return dist;
}

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

    return dist;
}

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

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

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

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

    return answer;
}

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

```
}

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

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

Traffic::Traffic(const Traffic &ref) :
    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]);
    }
}

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

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

    return *this;
}

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

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

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

```cpp
}

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

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

        std::vector<RoadSegment*> segments = Road::shared().spawn_positions();
        RoadSegment * seg;
        Car * new_car;
        if(spawn_pos < 1){
            seg = segments[0];
            if(start_lane < 0){
                new_car = new Car(seg,0,speed,target,aggressiveness);
            }
            else if(start_lane < 0.5){
                new_car = new Car(seg,1,speed,target,aggressiveness);
            }
            else{
                new_car = new Car(seg,2,speed,target,aggressiveness);
            }
        }
        else{
            seg = segments[1];
            new_car = new Car(seg,0,speed,target,aggressiveness);
        }

        Car * closest_car_ahead = new_car->find_closest_car();

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

void Traffic::despawn_car(Car *& car) {
    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;
        }
    }
}
```

```
void Traffic::despawn_cars() {
    //std::cout << ”e\n”;
    std::for_each(m_cars.begin(),m_cars.end(),car_deleter());
    //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”;
}

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

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);
}

void Traffic::update(float elapsed_time) {
    //std::cout<< ”updatin1\n”;
    for(Car * car : m_cars){
        car->avoid_collision(elapsed_time);
    }
    //std::cout<< ”updatin2\n”;
    for(Car * car : m_cars){
        car->update_pos(elapsed_time);
    }
    //std::cout<< ”updatin3\n”;
}

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

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;

    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_connections()){
                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));
            sf::Uint8 colorspeed = static_cast<sf::Uint8> ((unsigned int)std::round(255 * car->speed() / car->target_speed()));
            rectangle.setFillColor(sf::Color(255-colorspeed,colorspeed,0,255));
            target.draw(rectangle,states);

            // this caused crash earlier
            if(car->heading_to_node!=nullptr){
                // print debug info about node placements and stuff
                sf::CircleShape circle;

                circle.setRadius(4.0f);
                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);
            }
        }
    }
}

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::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"
                         + "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);
}

void car_deleter::operator()(Car *&car) {
    for(RoadSegment * seg : Road::shared().despawn_positions()){
        if(car->get_segment() == seg){
            //std::cout << ”deletin\n”;
            //std::cout << car << "\n";

            delete car;
            car = nullptr;
            //std::cout << ”deletidn\n”;
            break;
        }
    }
}
```

../highway/traffic.cpp

A.2.2 window.cpp

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

#include <iostream>
#include "traffic.h"
#include "window.h"
#include <cmath>
#include <unistd.h>

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

}

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/window.cpp

A.2.3 unittests.cpp

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

```
//

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

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

    for(RoadSegment * seg : segments){
        usleep(100000);
        std::cout<< "seg " << i << ", nlanes " << seg->get_total_amount_of_lanes() << ","<< seg << std::
    endl;
        std::cout << "next segment" << seg->next_segment() << std::endl;
        std::vector<RoadNode*> nodes =  seg->get_nodes();
        for(RoadNode * node : nodes){
            std::vector<RoadNode*> connections = node->get_connections();
            std::cout << "node" << node <<" has connections:" <<  std::endl;
            for(RoadNode * pointy : connections){
                std::cout << pointy << std::endl;
            }
        }
        i++;
        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_connections();
            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/unittests.cpp

A.2.4 main.cpp

```
#include <iostream>
#include "SFML/Graphics.hpp"
#include "window.h"
#include "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 = true;
    bool super_debug = true;

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

    }

    sf::Sprite background;
    background.setTexture(texture);
    //background.setColor(sf::Color::Black);
```

```cpp
    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();
        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/main.cpp