Final Project, SI1336

Carl Schiller, 9705266436

January 2, 2019

Abstract

Contents

1 Introduction

2 Method

3 Result

4 Discussion

A Potential

A.1 Header files

A.1.1 traffic.h

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

#ifndef HIGHWAY_TRAFFIC_H
#define HIGHWAY_TRAFFIC_H



class RoadSegment;

class Car;

class RoadNode{
private:
    float m_x, m_y;
    std::vector<RoadNode*> m_connecting_nodes;
    RoadSegment* m_is_child_of;
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:
    float m_x, m_y, m_theta;
    int m_n_lanes;
    constexpr static float M_LANE_WIDTH = 4.0f;

    std::vector<RoadNode> m_nodes;
    std::map<int,bool> m_car_ids;
    RoadSegment * m_next_segment;
public:
    RoadSegment();
    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);
    ~RoadSegment();

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

    int get_lane_number(RoadNode *);
    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;
    std::vector<RoadSegment*> m_spawn_positions;
    std::vector<RoadSegment*> m_despawn_positions;

    const std::string M_FILENAME = ”../road.txt”;
public:
    Road();
    ~Road();

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

/**
 * 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(RoadSegment * spawn_point, int lane, float vel, float target_speed, float agressivness, int
    unique_id);

    int id;

    RoadSegment * current_segment;
    RoadNode * current_node;
    RoadNode * heading_to_node;

    void update_pos(float delta_t);
    void accelerate();

    //void avoid_collision(std::vector<Car> & cars, int i, float & elapsed,float delta_theta,
    //                       std::vector<std::vector<int>> & allowed_zon);

    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 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 bool find_connected_path(Car & ref, Car & car, std::vector<std::vector<int>> & allowed_zone, int buffer);
    static Car * find_closest_car(std::vector<Car> &cars, Car * ref, std::vector<std::vector<int>> & allowed_zone);
    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{
private:
    Road m_road = Road();
    std::vector<Car> m_cars;
    int m_id;

    std::mt19937 & my_engine();

    //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();

    const unsigned long n_of_cars()const;
    const Road & road()const;
    void spawn_cars(double & spawn_counter, float elapsed, double & threshold);
    void despawn_cars();
    //void force_spawn_car();
    void debug(sf::Time t0);
    void update(float elapsed_time);
    const std::vector<Car> & get_cars()const;
    float get_avg_flow();
};

#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 : public sf::Drawable, public sf::Transformable{
public:
    Simulation();
    explicit Simulation(bool debug,int sim_speed);

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

#endif //HIGHWAY_WINDOW_H
```

../highway/window.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>

Car::Car() = default;

Car::Car(RoadSegment *spawn_point, int lane, float vel, float target_speed, float aggressivness, int unique_id) {
    current_segment = spawn_point;
    id = unique_id;

    current_segment->append_car(this);
    current_node = current_segment->get_node_pointer(lane);
    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);
    m_speed = vel;
    m_target_speed = target_speed;
    m_aggressiveness = aggressivness;
}

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
        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()){
            heading_to_node = connections[connections.size()-1];
            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 target = m_target_speed;
    float d_vel; // proportional control.

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

    m_speed += d_vel;
```

```cpp
}

float Car::x_pos() {
    float x_position = heading_to_node->get_x()-m_dist_to_next_node*cos(m_theta);

    return x_position;
}

float Car::y_pos() {
    float y_position = heading_to_node->get_y()+m_dist_to_next_node*sin(m_theta);

    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() = default;

```

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

RoadSegment::RoadSegment(float x, float y, RoadSegment * next_segment, int lanes) {
    m_x = x;
    m_y = y;

    m_next_segment = next_segment;

    m_theta = atan2(m_y-m_next_segment->m_y,m_next_segment->m_x-m_x);

    m_n_lanes = lanes;
    m_nodes.reserve(lanes);

    calculate_and_populate_nodes();
}

RoadSegment::RoadSegment(float x, float y, float theta, int lanes) {
    m_x = x;
    m_y = y;

    m_next_segment = nullptr;

    m_theta = theta;

    m_n_lanes = lanes;
    m_nodes.reserve(lanes);

    calculate_and_populate_nodes();
}

RoadSegment::RoadSegment(float x, float y, int lanes) {
    m_x = x;
    m_y = y;

    m_next_segment = nullptr;

    m_n_lanes = lanes;
    m_nodes.reserve(m_n_lanes);

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

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

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

float RoadSegment::get_y() {
    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_car_ids[(car->id)] = true;
}

void RoadSegment::remove_car(Car * car) {
    if(m_car_ids[car->id]){
        m_car_ids[car->id] = false;
    }
    else{
        throw std::invalid_argument("Car cannot be found in segment");
    }
}
```

```
std::map<int,bool>& RoadSegment::get_car_map() {
    return m_car_ids;
}

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.emplace_back(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);
}

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

Road::~Road() = default;

void Road::insert_segment(RoadSegment & segment) {
    m_segments.push_back(segment);
}

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){
        m_segments.emplace_back(RoadSegment(std::stof(vec[1]),std::stof(vec[2]),std::stoi(vec[3])));
    }

    // 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 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 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;
}

const std::vector<RoadSegment>& Road::segments()const {
    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;
    }

}

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)));
}


bool Util::find_connected_path(Car & ref, Car & car, std::vector<std::vector<int>> & allowed_zone, const
    int buffer){
    auto init_x = (int)std::round(ref.x_pos());
    auto init_y = (int)std::round(ref.y_pos());
    auto target_x = (int)std::round(car.x_pos());
    auto target_y =  (int)std::round(car.y_pos());

    auto search_radius = (unsigned int)ceil(std::max(abs(init_x-target_x),abs(target_y-init_y)))+buffer;
    unsigned int search_diameter = 2*search_radius+1;

    std::vector<std::vector<bool>> visited(search_diameter,std::vector<bool>(search_diameter,false));
    bool connected = false;

    std::vector<std::vector<int>> next_square;
    int current_x = init_x;
    int current_y = init_y;
    std::vector<int> current_square = {current_x,current_y};
    next_square.push_back(current_square);

    while(!next_square.empty()){
        current_square = next_square.back();
        next_square.pop_back();

        if(current_square[0] == target_x && current_square[1] == target_y){
            connected = true;
            break;
        }

        if(current_square[0] >= 0 && current_square[0] < allowed_zone[0].size() && current_square[1] >= 0
    && current_square[1] < allowed_zone.size()){
            if(allowed_zone[current_square[1]][current_square[0]] == 1){
                if(abs(current_square[1]-init_y) <= search_radius &&abs(current_square[0]-init_x) <=
    search_radius){
                    if(!visited[current_square[1]-init_y+search_radius][current_square[0]-init_x+
    search_radius]){

                        visited[current_square[1]-init_y+search_radius][current_square[0]-init_x+
    search_radius] = true;

                        std::vector<int> new_square = current_square;

                        new_square[0]++;
                        next_square.push_back(new_square);
                        new_square[0]--;

                        new_square[0]--;
                        next_square.push_back(new_square);
                        new_square[0]++;

                        new_square[1]++;
                        next_square.push_back(new_square);
                        new_square[1]--;

                        new_square[1]--;
                        next_square.push_back(new_square);
                    }
                }
            }
        }
    }

    return connected;
}

Car * Util::find_closest_car(std::vector<Car> &cars, Car * ref, std::vector<std::vector<int>> &
    allowed_zone){
    Car * answer = nullptr;
    float search_radius = 100;
    int buffer = 10;

    std::map<float,Car*> candidates;

    // calculate distances
    for(Car & car : cars){
        if(ref!=&car){
            float dist = distance_to_car(*ref,car);
            if(is_car_behind(ref,&car) && dist < search_radius){
                candidates[dist] = &car;
            }
        }
    }

    // loop through by smallest distance and check if it is connected.
    for(auto it : candidates){
        if(find_connected_path(*ref,*it.second,allowed_zone,buffer)){
            answer = it.second;
            break;
        }
    }

    return answer;
}

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)));
}

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

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

    float radius_to_next_car, theta_to_car, theta_diff_to_car_position,
            theta_diff_between_car_directions;

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

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


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

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

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

    float delta_speed = 0;
    float radius_to_car = 200;

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

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

    }

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

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

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

    }


}

*/

/*
std::vector<std::vector<int>> load_from_text(const std::string & name){
    std::vector<std::vector<int>> vec;
    std::ifstream stream(name);
    char delim = ',';
    std::string line;

    while(std::getline(stream,line)){
        std::vector<int> subvec;
        std::string character;
        std::stringstream str(line);
        while(std::getline(str,character,delim)){
            subvec.push_back(std::stoi(character));
        }

        vec.push_back(subvec);
    }

    return vec;
}

std::map<Spawn_positions,std::vector<float>> load_spawn_points(){
    std::map<Spawn_positions ,std::vector<float>> my_map;
    std::vector<float> pos1 = {82,599,1.36f};
    std::vector<float> pos2 = {377,309,1.88f};
    my_map[Spawn_positions::LOWER_LEFT] = pos1;
    my_map[Spawn_positions::RAMP] = pos2;
    return my_map;
};

std::map<Despawn_positions,std::vector<float>> load_despawn_points(){
    std::map<Despawn_positions ,std::vector<float>> my_map;
    std::vector<float> pos1 = {546,21,1.88f};
    std::vector<float> pos2 = {344,274,1.88f};
    my_map[Despawn_positions::UPPER_RIGHT] = pos1;
    my_map[Despawn_positions::RAMP] = pos2;
    return my_map;
};

std::map<Lane_positions ,std::vector<float>> load_lane_points(){
    std::map<Lane_positions ,std::vector<float>> my_map;
    std::vector<float> lower_left = {99,547,(float)M_PI*0.5f*0.8f,(float)M_PI*0.5f*0.6f};

    my_map[Lane_positions ::LOWER_LEFT] = lower_left;
    return my_map;
};

*/

Traffic::Traffic() {
    m_id = 0;
};

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

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

void Traffic::spawn_cars(double & spawn_counter, float elapsed, double & threshold) {
    spawn_counter += elapsed;
    if(spawn_counter > threshold && m_cars.size() < 2){
        std::exponential_distribution<double> dis(0.5);
        std::normal_distribution<float> aggro(0.05f,0.01f);
        std::normal_distribution<float> sp(20.0,2.0);
        std::uniform_real_distribution<float> ramp(0.0f,1.0f);

        float speed = sp(my_engine());
        float target = speed;
        threshold = dis(my_engine());
        float aggressiveness = aggro(my_engine());
        spawn_counter = 0;

        RoadSegment * seg = m_road.spawn_positions()[0];
        m_cars.emplace_back(Car(seg,1,speed,target,aggressiveness,m_id));
        m_id++; // new id for next car
    }
}

void Traffic::despawn_cars() {
    int car_amount = static_cast<int>(m_cars.size());
    for(int i = 0; i < car_amount; i++){
        for(RoadSegment * seg : m_road.despawn_positions()){
            if(m_cars[i].get_segment() == seg){
                m_cars.erase(m_cars.begin()+i);
                i--;
                car_amount--;
            }
        }
    }
}

/*
float Traffic::get_theta(float xpos, float ypos, float speed, float current_theta, bool & lane_switch) {
    std::vector<float> theta_candidates;
    float radius = 2.0f;

    const int divisions = 60;

    for(int i = 0; i < divisions; i++){
        float angle = (float)(i)/(float)divisions*2.0f*3.141f;
        auto x_temp = (int)round(xpos + radius*cos(angle));
        auto y_temp = (int)round(ypos - radius*sin(angle));
        if( y_temp < m_allowed_zone.size() && y_temp >= 0 &&
            x_temp < m_allowed_zone[0].size() && x_temp >= 0 ){
            if(m_allowed_zone[y_temp][x_temp] == 1) {
                theta_candidates.push_back(angle);
            }
        }
    }

    if(theta_candidates.empty()){
        return current_theta;
    }
    else{
        float best_score = 100000;
        float best_theta = 100000;
        for(float c : theta_candidates){
            float score = Util::get_min_angle(c,current_theta);
            if( score < best_score){
                best_score = score;
                best_theta = c;
            }
        }

        for(const auto & it : m_lane_switch_points){
            float rad = sqrt(abs(pow(xpos-it.second[0],2.0f))+abs(pow(ypos-it.second[1],2.0f)));
            if(!lane_switch && rad < 2){
                std::uniform_real_distribution<float> prob(0.0f,1.0f);
                float coin_flip = prob(my_engine());
                if(coin_flip > 1.0){
                    best_theta = it.second[3];
                }
                lane_switch = true;
            }
            else if (rad > 2){
                lane_switch = false;
            }
        }

        return best_theta;
    }
}
*/

/*
void Traffic::update_speed(int i, float & elapsed_time) {
    // look in a circle speed/4 m around car to find next angle to drive in.
    Car & car = m_cars[i];
    float old_theta = car.theta();
    float theta = get_theta(car.x_pos(),car.y_pos(), car.speed(),car.theta(),car.lane_switch);
    car.steer(theta);
    car.avoid_collision(m_cars,i,elapsed_time,theta-old_theta,m_allowed_zone);
}
*/

void Traffic::update(float elapsed_time) {
    for(Car & car : m_cars){
```

```cpp
        car.update_pos(elapsed_time);
    }
}

void Traffic::debug(sf::Time t0) {
    if(n_of_cars() > 0){
        std::string message;
        message += "Vel: " + std::to_string(m_cars[0].speed()*3.6f).substr(0,4) + " km/h, time: " +
                   std::to_string(t0.asSeconds()).substr(0,3) + " s, theta:" + std::to_string(m_cars[0].theta()).substr(0,4) +
                " ,x:" + std::to_string(m_cars[0].x_pos()).substr(0,3) + " ,y:" + std::to_string(m_cars[0].y_pos()).substr(0,3);
        std::cout << message << std::endl;
    }
}


/*
void Traffic::force_spawn_car() {
    m_cars.emplace_back(Car(82,599,20.0,13,20,0.05));
}
*/

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

float Traffic::get_avg_flow() {
    float flow = 0;
    for(Car & car : m_cars){
        flow += car.speed()/car.target_speed();
    }
    return flow/(float)n_of_cars();
}

const Road & Traffic::road() const {
    return m_road;
}
```

../highway/traffic.cpp

A.2.2 window.cpp

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

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


void Simulation::draw(sf::RenderTarget &target, sf::RenderStates states) const {
    if(m_debug){
        // 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: m_traffic.road().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);

    for(Car car : m_traffic.get_cars()){
        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);
    }

}

Simulation::Simulation() {
    m_debug = false;
    m_sim_speed = 1;

    if (!m_font.loadFromFile(”/Library/Fonts/Arial.ttf”))
    {
        // error...
    }
}

Simulation::Simulation(bool debug, int speed) {
    m_debug = debug;
    m_sim_speed = speed;

    if (!m_font.loadFromFile(”/Library/Fonts/Arial.ttf”))
    {
        // error...
    }
}

void Simulation::update(sf::Time elapsed, double & spawn_counter, double & threshold) {
    float elapsed_time = elapsed.asSeconds();
    for(int i = 0; i < m_sim_speed; i++){
        m_traffic.update(elapsed_time);
        m_traffic.despawn_cars();
        m_traffic.spawn_cars(spawn_counter,elapsed_time,threshold);
    }
}

void Simulation::car_debug(sf::Time t0){
    m_traffic.debug(t0);
}

float Simulation::get_flow() {
    return m_traffic.get_avg_flow();
}

void Simulation::get_info(sf::Text & text,sf::Time &elapsed) {
    float fps = 1.0f/elapsed.asSeconds();
    float flow = get_flow();
    std::string speedy = std::to_string(fps).substr(0,2) +
            " fps, speed: " + std::to_string(m_sim_speed).substr(0,1) + " x\nFlow " +
            std::to_string(get_flow()).substr(0,4);
    text.setString(speedy);
    text.setPosition(0,0);
    text.setFillColor(sf::Color::Green);
    text.setFont(m_font);
}
```

../highway/window.cpp

A.2.3 main.cpp

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

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

    int sim_speed = 1;
    bool debug = true;

    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;

    Simulation simulation = Simulation(debug,sim_speed);
    double spawn_counter = 0.0;
    double threshold = 0.0;

    sf::Text debug_info;

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

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

        simulation.update(elapsed,spawn_counter,threshold);

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

        window.draw(background);
        window.draw(simulation);
        if(debug){
            simulation.get_info(debug_info,elapsed);
            window.draw(debug_info);
        }
        window.display();
    }
    return 0;
}
```

../highway/main.cpp