МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

**«Московский технический университет связи и информатики»**

Кафедра «Информатика»

**Отчет по курсовой работе**

**по дисциплине**

**«Проектный практикум»**

Выполнил: студент гр. БЭИ2202 Кулешов А. С.

Принял: зав. кафедры «Информатика» Халабия Р.Ф.

Москва, 2025 г.

**Оглавление**

[**1)** **Постановка задачи** 3](#_Toc184172495)

[**2)** **Цель проекта** 4](#_Toc184172496)

[**3)** **Исследование предметной области** 4](#_Toc184172497)

[4) **Требования к разработке** 6](#_Toc184172498)

[**5)** **Эскизный проект** 8](#_Toc184172499)

[**6)** **Алгоритмы** 10](#_Toc184172500)

[**7)** **Листинг кода** 14](#_Toc184172501)

[**8)** **Полученные результаты** 32](#_Toc184172502)

[**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ** 42](#_Toc184172503)

# **Постановка задачи**

1. **Создание модели перекрёстка умного города:**

* Разработать модель поведения машин.
* Создать 2 модели перекрёстка: с умным светофором и без

1. **Создание интерфейса пользователя:**

* Разработать удобный интерфейс для перемещения и изменения масштаба на модели.

1. **Анализ данных:**

* По завершению работы программы, данные о загруженности и пропускной способности должны быть выведены пользователю.

# **Цель проекта**

Анализ улучшения качеств автомобильного движения при смене светофоров на «умные»

# **Исследование предметной области**

**Обзор теоретической базы**

Для исследования предметной области были использованы следующие источники:

1. Документация C++.
2. Материалы по использованию библиотеки SFML для визуализации симуляции.

**Основные результаты анализа**

На основе изучения литературы и существующих решений выделены ключевые тенденции и проблемы:

1. **Тенденции:**

* Популяризация автомобилей и их использования в повседневной жизни.
* Увеличение загруженности на современных автомобильных дорогах.

1. **Проблемы:**

* **Отсутствие простого решения:** текущие платформы либо сосредоточены на комплексной симуляции, либо сложны в использовании для простой демонстрации

1. **Выводы:**

* Требуется модель, позволяющая проанализировать улучшение автомобильного движения при смене светофоров на «умные».
* Основные принципы реализации: отсутствие лишнего функционала, простота использования и высокая производительность.

**Оценка значимости результатов в предметной области**

* **Практическая применимость:**  
  Данное приложение демонстрирует результат улучшения автомобильного движения при смене светофоров при маленьком количестве данных, таким образом результат работы программы может поддаваться ручному анализу
* **Влияние на область знаний:**  
  Разработка минималистического приложения позволит заинтересовать большую группу людей среди сообществ, таким образом увеличив интерес к существующей проблеме

**Рекомендации и направления дальнейших исследований**

1. **Расширение функциональности:**

* Добавление новых элементов умного города
* Увеличение размеров симуляции, возможность ускорения симуляции

1. **Углубление теоретической базы:**

* Исследование моделей оптимизации портфеля для нестандартных типов активов.
* Разработка рекомендаций для индивидуальных стратегий инвестирования.

1. **Технические улучшения:**

* Повышение производительности системы при большом количестве агентов.

1. **Развитие интерфейсов:**

* Улучшение графического интерфейса приложения.
* Разработка мобильной версии приложения.

**Обзор существующих решений**

1. **Mobiliti:**

* Эта программная платформа имитирует движение людей и транспортных средств по крупному мегаполису.
* Недостатки: сложный интерфейс для новичков, сложность расширения приложения.

1. **VISSIM:**

* В этом моделировании используются данные о дорожном движении в реальном времени для обновления объемов транспортных средств и управления сигналами.
* Недостатки: необходимость в наличии реальных данных о текущем движении автомобилей.

# **Требования к разработке**

**Аппаратные требования**

* **Минимальные:**
* Операционная система: Windows 10 / Linux Ubuntu 20.04 / macOS 10.15 и выше.
* Процессор: Intel Core i3 или аналогичный.
* Оперативная память: 4 ГБ.
* Хранилище: 500 МБ свободного пространства для приложения и данных.
* **Рекомендуемые:**
* Операционная система: Windows 11 / Linux Ubuntu 22.04 / macOS 12.
* Процессор: Intel Core i5 или выше.
* Оперативная память: 8 ГБ и выше.
* Хранилище: 2 ГБ свободного пространства.

**Программные требования**

* **Язык разработки:** C++17.
* **Контроль версий:**

Git и GitHub для управления исходным кодом.

# **Эскизный проект**

1. **Общее описание системы**

* Это приложение для запуска примитивной симуляции поведения умного города, которое позволит пользователям:
* Наблюдать за поведением агентов(автомобилей) умного города
* Анализировать данные поведения модели в реальном времени
* Добавлять новый функционал при помощи платформы открытого коад

1. **Основные компоненты и их взаимодействие**
   * 1. **Интерфейс симуляции:**

* Агенты симуляции(автомобили).
* Светоформы.
* Данные анализа симуляции.
  + 1. **Агенты:**
* **Автомобиль:**
* Передвижение по участку дороги.
* Остановка перед другим агентом автомобилем или светофором.
* **Дорога:**
* Визуальное отображение на экране
* Создание маршрута для автомобилей
* **Светофор:**
* Включение и выключение движения по автомобильной дороге.
* Включение «Умного поведения»: светофор старается соблюдать баланс машин на перекрёстке

# **Алгоритмы**

1. **Метод update() у TrafficLight**

Шаг 1. Увеличить встроенный счётчик времени

Шаг 2. Задать цвет в зависимости от текущего времени

Шаг 3. Если светофор умный и счётчик близок к обнулению произвести перерасчёт интервалов сигналов на основе данных автомобилей

1. **Метод update() у Car**

Шаг 1. Рассчитать вектор направления машины в зависимости от текущей точки назначения

Шаг 2. Уменьшить вектор до длины равной максимальной скорости данной машины. Вычислить новое потенциальное расположение машины

Шаг 3. Если новое расположение находится слишком близко к другой машине, то не производить перемещение и закончить алгоритм

Шаг 4. Если машина находится в зоне действующего светофора, и машина не игнорирует данный светофор и данный светофор выключен, то не производить перемещение и закончить алгоритм

Шаг 5. Если машина находится в зоне действующего светофора, и машина не игнорирует данный светофор и данный светофор включён, то начать игнорировать светофор

Шаг 6. Произвести перемещение, если позиция светофора находится на одном из маршрутных пунктов, то выбрать следующий маршрутный пункт из списка, если его нет, то удалить машину из симуляции

# **Листинг кода**

#define SFML\_STATIC

#include <SFML/Graphics.hpp>

#include <cmath>

#include <iostream>

#include <iomanip>

#include <set>

**using** **namespace** std;

#include <sstream>

**template** <**typename** T>

std::wstring NumberToString ( T Number )

{

std::ostringstream ss;

ss << Number;

std::wstring wsTmp(ss.str().begin(), ss.str().end());

**return** wsTmp;

}

**class** **sfLine** : **public** sf::Drawable

{

**public:**

sfLine(**const** sf::Vector2f& point1, **const** sf::Vector2f& point2, sf::Color color\_ = sf::Color::Black, **float** thick = **5.f**):

color(color\_), thickness(thick)

{

sf::Vector2f direction = point2 - point1;

sf::Vector2f unitDirection = direction/std::sqrt(direction.x\*direction.x+direction.y\*direction.y);

sf::Vector2f unitPerpendicular(-unitDirection.y,unitDirection.x);

sf::Vector2f offset = (thickness/**2.f**)\*unitPerpendicular;

vertices[**0**].position = point1 + offset;

vertices[**1**].position = point2 + offset;

vertices[**2**].position = point2 - offset;

vertices[**3**].position = point1 - offset;

**for** (**int** i=**0**; i<**4**; ++i)

vertices[i].color = color;

}

**void** draw(sf::RenderTarget &target, sf::RenderStates states) **const**

{

target.draw(vertices,**4**,sf::Quads);

}

**public:**

sf::Vertex vertices[**4**];

**float** thickness;

sf::Color color;

};

pair<sf::Vector2f, sf::Vector2f> get\_Line\_from\_equ(**long** **double** a, **long** **double** b, **long** **double** c){

**long** **double** pseudo\_inf = **1e5**;

**if** (abs(a) < **0.0001** ){

**if** (abs(b) < **0.0001**){

std::cerr << "Equation cannot be a form of 0x + 0y";

exit(-**1**);

}

**long** **double** y\_1 = (-a\*pseudo\_inf - c) / b;

**long** **double** y\_2 = (-a\*(-pseudo\_inf) - c) / b;

**return** {{pseudo\_inf, -y\_1},{-pseudo\_inf, -y\_2,}};

}

**long** **double** x\_1 = (-b\*pseudo\_inf - c) / a;

**long** **double** x\_2 = (-b\*(-pseudo\_inf) - c) / a;

**return** {{x\_1,-pseudo\_inf},{x\_2,-(-pseudo\_inf)}};

}

vector<vector<**long** **double**>> equations\_input;

vector<**long** **double**> target\_function; /// 2x1 + 5x2 -> max

sfLine **get\_line**(**const** vector<**long** **double**>& equation, sf::Color clr = sf::Color::Red){

**auto** p = get\_Line\_from\_equ(equation[**0**], equation[**1**], -equation[**2**]);

**return** sfLine (p.first, p.second, clr, **0.25**); /// 2x1 + 5x2 -> max <=>

}

sfLine **get\_contour\_line**(**double** F\_value, sf::Color clr = sf::Color::Red){

**auto** p = get\_Line\_from\_equ(target\_function[**0**], target\_function[**1**], -F\_value);

**return** sfLine (p.first, p.second, clr, **0.15**); /// 2x1 + 5x2 -> max <=>

}

**double** **dist**(**double** x1, **double** y1, **double** x2, **double** y2){

**return** sqrt((x1 - x2)\*(x1 - x2) + (y1 - y2)\*(y1 - y2));

}

**double** **dist**(**const** sf::Vector2f& vec){

**return** dist(**0**,**0**, vec.x, vec.y);

}

// LINE/LINE

pair<**bool**, sf::Vector2f> lineLine(**float** x1, **float** y1, **float** x2, **float** y2, **float** x3, **float** y3, **float** x4, **float** y4) {

// calculate the direction of the lines

**float** uA = ((x4-x3)\*(y1-y3) - (y4-y3)\*(x1-x3)) / ((y4-y3)\*(x2-x1) - (x4-x3)\*(y2-y1));

**float** uB = ((x2-x1)\*(y1-y3) - (y2-y1)\*(x1-x3)) / ((y4-y3)\*(x2-x1) - (x4-x3)\*(y2-y1));

// if uA and uB are between 0-1, lines are colliding

**if** (uA >= **0** && uA <= **1** && uB >= **0** && uB <= **1**) {

// optionally, draw a circle where the lines meet

**float** intersectionX = x1 + (uA \* (x2-x1));

**float** intersectionY = y1 + (uA \* (y2-y1));

**return** {true, {intersectionX, intersectionY}};

}

**return** {false, {-**1**,-**1**}};

}

**struct** geom\_line{

**long** **double** x1, y1, x2, y2;

};

**long** **double** min\_ = **1e10**; sf::Vector2f min\_point = {-**1**,-**1**};

**long** **double** max\_ = -**1e10**; sf::Vector2f max\_point = {-**1**,-**1**};

vector<sf::Vector2f> convex\_points;

**bool** **check\_point**(**const** sf::Vector2f& point){

**long** **double** value = point.x \* target\_function[**0**] + point.y \* target\_function[**1**];

**for**(**auto** equation : equations\_input){

**long** **double** check\_value = point.x \* equation[**0**] + point.y \* equation[**1**];

**if** (!(check\_value - **0.001** <= equation[**2**]) || point.x < **0** || point.y < **0** ){

**return** false;

}

}

**if** (value < min\_){

min\_ = value;

min\_point = point;

}

**if** (value > max\_){

max\_ = value;

max\_point = point;

}

convex\_points.push\_back(point);

**return** true;

}

**int** **orientation**(sf::Vector2f a, sf::Vector2f b, sf::Vector2f c) {

**double** v = a.x\*(b.y-c.y)+b.x\*(c.y-a.y)+c.x\*(a.y-b.y);

**if** (v < **0**) **return** -**1**; // clockwise

**if** (v > **0**) **return** +**1**; // counter-clockwise

**return** **0**;

}

**bool** **cw**(sf::Vector2f a, sf::Vector2f b, sf::Vector2f c, **bool** include\_collinear) {

**int** o = orientation(a, b, c);

**return** o < **0** || (include\_collinear && o == **0**);

}

**bool** **collinear**(sf::Vector2f a, sf::Vector2f b, sf::Vector2f c) { **return** orientation(a, b, c) == **0**; }

**void** **convex\_hull**(vector<sf::Vector2f>& a, **bool** include\_collinear = false) {

sf::Vector2f p0 = \*min\_element(a.begin(), a.end(), [](sf::Vector2f a, sf::Vector2f b) {

**return** make\_pair(a.y, a.x) < make\_pair(b.y, b.x);

});

sort(a.begin(), a.end(), [&p0](**const** sf::Vector2f& a, **const** sf::Vector2f& b) {

**int** o = orientation(p0, a, b);

**if** (o == **0**)

**return** (p0.x-a.x)\*(p0.x-a.x) + (p0.y-a.y)\*(p0.y-a.y)

< (p0.x-b.x)\*(p0.x-b.x) + (p0.y-b.y)\*(p0.y-b.y);

**return** o < **0**;

});

**if** (include\_collinear) {

**int** i = (**int**)a.size()-**1**;

**while** (i >= **0** && collinear(p0, a[i], a.back())) i--;

reverse(a.begin()+i+**1**, a.end());

}

vector<sf::Vector2f> st;

**for** (**int** i = **0**; i < (**int**)a.size(); i++) {

**while** (st.size() > **1** && !cw(st[st.size()-**2**], st.back(), a[i], include\_collinear))

st.pop\_back();

st.push\_back(a[i]);

}

**if** (include\_collinear == false && st.size() == **2** && st[**0**] == st[**1**])

st.pop\_back();

a = st;

}

**class** **Car**;

**class** **TrafficLight** : **public** sf::Drawable{

**public:**

sf::RectangleShape area = sf::RectangleShape({**10**,**10**});

**int** ticks\_cycle = **100**;

**int** ticks = **0**;

**bool** isSmart = false;

**int** registered1 = **0**;

**int** registered2 = **0**;

TrafficLight(**const** sf::Vector2f& pos = {**100**,**100**}, **bool** \_isSmart = false){

area.setPosition(pos.x-**5**,pos.y-**5**);

isSmart = \_isSmart;

}

string state = "red";

**int** road1\_proportion = **30**; /// 10-50

**int** green1\_start = **0**;

**int** green1\_end = **30**;

**int** green2\_start = **50**;

**int** green2\_end = **80**;

**void** **register\_car**(**int** type){

**if** (type == **1**) ++registered1;

**if** (type == **2**) ++registered2;

}

**void** **update**(){

ticks = (ticks + **1**) % ticks\_cycle;

**if** (ticks == **99** && isSmart){

/// recalculate timings;

**double** density1 = (**double**)(registered1) / (road1\_proportion + **15**);

**double** density2 = (**double**)(registered2) / (**60** - road1\_proportion + **15** );

**if** (density2 / **1.1** > density1){

road1\_proportion -= **5**;

}

**if** (density2 / **1.4** > density1){

road1\_proportion -= **5**;

}

**if** (density2 / **1.7** > density1){

road1\_proportion -= **5**;

}

**if** (density1 / **1.1** > density2){

road1\_proportion += **5**;

}

**if** (density1 / **1.4** > density2){

road1\_proportion += **5**;

}

**if** (density1 / **1.7** > density2){

road1\_proportion += **5**;

}

road1\_proportion = max(**10**, road1\_proportion);

road1\_proportion = min(**50**, road1\_proportion);

cout << "New proportion " << road1\_proportion << endl;

registered1 = **0**;

registered2 = **0**;

}

sf::Color clr = sf::Color::White;

**if** (allowed1()){

clr = sf::Color::Green;

}

**if** (allowed2()){

clr = sf::Color::Red;

}

clr.a = **40**;

area.setFillColor(clr);

}

**bool** **allowed1**(){

**return** ticks > green1\_start && ticks < road1\_proportion;

}

**bool** **allowed2**(){

**return** ticks > road1\_proportion + **20** && ticks < green2\_end;

}

**bool** **allowed**(**int** type){

**if** (type == **1**) **return** allowed1();

**if** (type == **2**) **return** allowed2();

**return** false;

}

**friend** **bool** **operator**<(**const** TrafficLight& one, **const** TrafficLight& other){

**if** (one.area.getSize().x != other.area.getSize().x) **return** one.area.getSize().x < other.area.getSize().x;

**if** (one.area.getSize().y != other.area.getSize().y) **return** one.area.getSize().y < other.area.getSize().y;

**if** (one.area.getPosition().x != other.area.getPosition().x) **return** one.area.getPosition().x < other.area.getPosition().x;

**if** (one.area.getPosition().y != other.area.getPosition().y) **return** one.area.getPosition().y < other.area.getPosition().y;

**return** false;

}

**private:**

**virtual** **void** draw(sf::RenderTarget& target, sf::RenderStates states) **const**

{

target.draw(area);

}

};

vector<Car> cars;

vector<TrafficLight> traffics;

**bool** **CircleToManyCircle**(vector<sf::Vector2f> poses, sf::Vector2f position, **float** radius){

**int** count = **0**;

**for**(**int** i = **0**; i < poses.size(); ++i){

**if** (dist( poses[i] - position ) < radius) ++count ;

}

**return** count > **1**;

}

**bool** **RectToPoint**(sf::RectangleShape r, sf::Vector2f point){

**return** (point.x >= r.getPosition().x && point.x <= r.getPosition().x+r.getSize().x &&

point.y >= r.getPosition().y && point.y <= r.getPosition().y+r.getSize().y);

}

**const** **float** eps = **0.001**;

**class** **Car** : **public** sf::Drawable{

**public:**

vector<sf::Vector2f> path;

sf::Vector2f position = {**0**,**0**};

**float** Speed = **1**;

sf::Color color;

**float** radius = **1**;

**int** current\_road\_type = **1**;

**int** path\_progress;

set<TrafficLight> ignore\_list;

string extraInfo = "None";

Car(vector<sf::Vector2f> \_path, sf::Vector2f \_position, **float** \_Speed = **1**,

sf::Color \_color = sf::Color::Red, **int** \_current\_road\_type = **1**, string \_extraInfo = "None"){

path = \_path;

position = \_position;

path\_progress = **0**;

Speed = \_Speed;

color = \_color;

current\_road\_type = \_current\_road\_type;

extraInfo = \_extraInfo;

}

string update(){

sf::Vector2f deltaVec = -position + path[path\_progress];

deltaVec.x = max(-Speed, min(deltaVec.x, Speed));

deltaVec.y = max(-Speed, min(deltaVec.y, Speed));

sf::Vector2f new\_position = position + deltaVec;

vector<sf::Vector2f> poses;

**for**(**int** i = **0**; i < cars.size(); ++i){

poses.push\_back(cars[i].position);

} // this is bad

**if** (CircleToManyCircle(poses, new\_position, **3**\*radius)) **return** "next\_car\_brake";

**for**(**int** i = **0**; i < traffics.size(); ++i){

**if** ( RectToPoint(traffics[i].area, new\_position)){

**if** (ignore\_list.find(traffics[i]) != ignore\_list.end() ){

**continue**;

}

**if** (traffics[i].allowed(current\_road\_type)){

ignore\_list.insert(traffics[i]);

traffics[i].register\_car(current\_road\_type);

}**else**{

**return** "brake\_traffic";

}

}

}

position = new\_position;

**if** ( dist(position - path[path\_progress]) < eps ){

path\_progress += **1**;

}

**if** (path\_progress == path.size()){

**return** "delete";

}

**return** "normal";

}

**private:**

**virtual** **void** draw(sf::RenderTarget& target, sf::RenderStates states) **const**

{

sf::CircleShape cs(radius);

cs.setPosition(position - sf::Vector2f(radius, radius));

cs.setFillColor(color);

target.draw(cs);

}

};

**int** **main**() {

sf::RenderWindow window(sf::VideoMode(**800**, **800**), "Plot");

sfLine O1({**100**,**0**}, {**100**,**200**}, sf::Color::Black, **5**);

sfLine O2({**0**,**100**}, {**200**,**100**}, sf::Color::Black, **5**);

sfLine O3({**400**,**0**}, {**400**,**200**}, sf::Color::Black, **5**);

sfLine O4({**300**,**100**}, {**500**,**100**}, sf::Color::Black, **5**);

traffics.push\_back(TrafficLight({**100**,**100**}, false));

traffics.push\_back(TrafficLight({**400**,**100**}, true));

sf::Font font;

**if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {

std::cerr << "Ошибка загрузки шрифта" << std::endl;

**return** **0**;

}

sf::Text analys("txt", font, **30**);

analys.setFillColor(sf::Color::Black);

analys.setPosition(**0**, **250**);

sf::View view(sf::Vector2f(**100.f**, **100.f**), sf::Vector2f(**100.f**, **100.f**));

window.setView(view);

window.setFramerateLimit(**120**);

**int** ticks = **0**;

**int** road\_density = **7**;

**int** Cross1CarsExited = **0**;

**int** Cross1CarsIn = **0**;

**int** Time1 = **0**;

**int** Cross2CarsExited = **0**;

**int** Cross2CarsIn = **0**;

**int** Time2 = **0**;

**while** (window.isOpen()) {

ticks += **1**;

sf::Event event;

**while** (window.pollEvent(event)) {

**if** (event.type == sf::Event::Closed) {

window.close();

}

**else** **if**(event.type == sf::Event::MouseWheelMoved)

{

**double** scale\_factor = **0.9**;

**for**(**int** i = **0**; i < abs(event.mouseWheel.delta); ++i){

**if** (event.mouseWheel.delta >= **0**)

view.zoom(scale\_factor);

**else**

view.zoom(**1** / scale\_factor);

}

window.setView(view);

}

}

**int** dx = **0**;

**int** dy = **0**;

**if** (sf::Keyboard::isKeyPressed( sf::Keyboard::Key::W)) dy -= **1**;

**if** (sf::Keyboard::isKeyPressed( sf::Keyboard::Key::S)) dy += **1**;

**if** (sf::Keyboard::isKeyPressed( sf::Keyboard::Key::A)) dx -= **1**;

**if** (sf::Keyboard::isKeyPressed( sf::Keyboard::Key::D)) dx += **1**;

view.setCenter(view.getCenter() + sf::Vector2f(dx,dy) \* (view.getSize().x) \* **0.03f** );

window.setView(view);

window.clear(sf::Color::White);

**if** (ticks % **2000** == **0**){

road\_density = **10** - road\_density;

cout << "New road density " << road\_density << endl;

}

**if** (ticks % **10** == **0**){

vector<sf::Vector2f> poses;

**for**(**int** i = **0**; i < cars.size(); ++i){

poses.push\_back(cars[i].position);

} // this is bad

**if** (rand() % **10** < road\_density){

//cars.push\_back(Car({ {0,40} }, {0,0}, 0.5, sf::Color::Blue, 2 ));

**if** (!CircleToManyCircle(poses, {**100**,**0**}, **5**\***1**)){

cars.push\_back(Car({ {**100**,**200**} }, {**100**,**0**}, **0.5**, sf::Color::Blue, **2**, "Cross1" ));

Cross1CarsIn++;

}

**if** (!CircleToManyCircle(poses, {**400**,**0**}, **5**\***1**)){

cars.push\_back(Car({ {**400**,**200**} }, {**400**,**0**}, **0.5**, sf::Color::Blue, **2**, "Cross2" ));

Cross2CarsIn++;

}

}**else**{

**if** (!CircleToManyCircle(poses, {**0**,**100**}, **5**\***1**)){

cars.push\_back(Car({ {**200**,**100**} }, {**0**,**100**}, **1**, sf::Color::Red, **1**, "Cross1" ));

Cross1CarsIn++;

}

**if** (!CircleToManyCircle(poses, {**300**,**100**}, **5**\***1**)){

cars.push\_back(Car({ {**500**,**100**} }, {**300**,**100**}, **1**, sf::Color::Red, **1**, "Cross2" ));

Cross2CarsIn++;

}

}

}

///cout << "sz " << cars.size() << endl;;

vector<Car> cars2;

**for** (**int** i = **0**; i < cars.size(); ++i){

string ret\_code = cars[i].update();

**if** (ret\_code != "delete"){

cars2.push\_back(cars[i]);

}**else**{

**if** (cars[i].extraInfo == "Cross2"){

Cross2CarsExited++;

}

**if** (cars[i].extraInfo == "Cross1"){

Cross1CarsExited++;

}

}

}

Time1 += (Cross1CarsIn - Cross1CarsExited);

Time2 += (Cross2CarsIn - Cross2CarsExited);

**for**(**int** i = **0**; i < traffics.size(); ++i){

traffics[i].update();

}

cars = cars2;

window.draw(O1);

window.draw(O2);

window.draw(O3);

window.draw(O4);

**auto** display = \

wstring( L"Магистраль #1**\n**") + \

L"Вошло: " + to\_wstring(Cross1CarsIn) + L"**\n**" +\

L"Вышло: " + to\_wstring(Cross1CarsExited) + L"**\n**"+\

L"Осталось: " + to\_wstring(Cross1CarsIn - Cross1CarsExited) + L"**\n**"+\

L"Среднее время в пути: " + ((Cross1CarsExited > **10**) ? to\_wstring(**double**(Time1)/ Cross1CarsExited) : L"Расчитывается...") + L"**\n\n**"+\

wstring( L"Магистраль #2**\n**") + \

L"Вошло: " + to\_wstring(Cross2CarsIn) + L"**\n**" +\

L"Вышло: " + to\_wstring(Cross2CarsExited) + L"**\n**"+\

L"Осталось: " + to\_wstring(Cross2CarsIn - Cross2CarsExited) + L"**\n**"+\

L"Среднее время в пути: " + ((Cross2CarsExited > **10**) ? to\_wstring(**double**(Time2)/ Cross2CarsExited) : L"Расчитывается...") + L"**\n**";

**if** (ticks % **120** == **0**){

analys.setString(display);

}

window.draw(analys);

**for**(**const** Car& car : cars){

window.draw(car);

}

**for**(**const** TrafficLight& traf : traffics){

window.draw(traf);

}

window.display();

}

**return** **0**;

}

# **Полученные результаты**

При запуске программы, предлагается войти в свой аккаунт либо зарегистрироваться. Если пользователь захочет войти с неправильными учетными данными, то будет предупрежден об этом, рисунки 1 – 2.

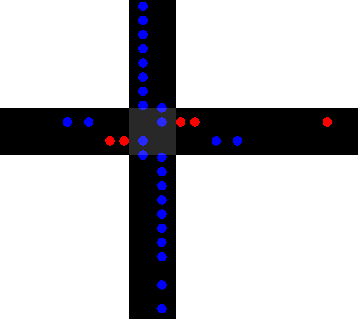


Рисунок 1 – Запущенная симуляция

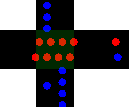


Рисунок 2 – Динамическое изменение состояния светофора

Результаты работы программы можно увидеть ниже, рисунок 3.

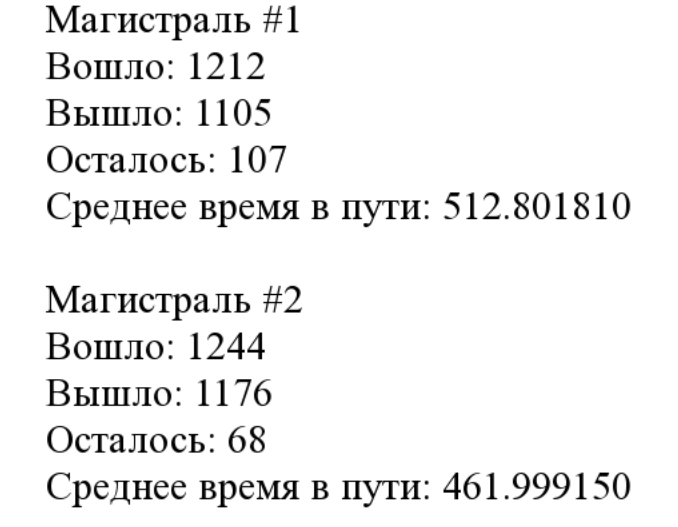


Рисунок 3 – Результат анализа

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Как писать выводы по главам в дипломе [Электронный ресурс]. — Режим доступа: <https://studwork.ru/spravochnik/oformlenie/diplomnye-raboty/kak-pisat-vyvody-po-glavam-v-diplome>. — Дата обращения: 11.12.2024.
2. Документация библиотеки "SFML" [Электронный ресурс]. — Режим доступа: <https://www.sfml-dev.org/documentation> — Дата обращения: 11.12.2024.
3. StackOverlow: Сайт «Вопрос-Ответ», посвящённый программированию [Электронный ресурс]. — Режим доступа: <https://stackoverflow.com>. — Дата обращения: 11.12.2024.