Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Конструирование программ и языки программирования

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

МОДЕЛИРОВАНИЕ ТРАФИКА ДОРОЖНОГО ДВИЖЕНИЯ

БГУИР КП 1-40 02 01 116 ПЗ

Студент: группы 950501,
Кукла Д. И.

Руководитель: ассистент каф. ЭВМ
Желтко Ю. Ю.

Минск 2020

Учреждение образования
«Белорусский государственный университет информатики
и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____Б.В. Никульшин
(подпись)
«___» _____2020 г.

ЗАДАНИЕ
по курсовому проектированию

Студенту *Кукле Дмитрию Ивановичу*

1. Тема проекта *Моделирование трафика дорожного движения*

2. Срок сдачи студентом законченного проекта *14 декабря 2020 г.*

3. Исходные данные к проекту *Язык программирования – C++*

4. Содержание расчетно-пояснительной записки (перечень вопросов, которые подлежат разработке)
Введение. 1. Обзор литературы. 2. Системное проектирование. 3. Функциональное проектирование. 4. Разработка программных модулей. 5. Программа и методика испытаний. 6. Руководство пользователя. Заключение. Список использованных источников

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков) *1. Схема структурная. 2. Диаграмма классов.*

6. Консультант по проекту *Желтко Ю.Ю.*

7. Дата выдачи задания *12 сентября 2020 г.*

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и трудоемкости отдельных этапов):
разделы 1,2 к 1 октября 2020 г. – 20 %;
разделы 3,4 к 1 ноября 2020 г. – 30 %;
разделы 5,6,7 к 1 декабря 2020 г. – 30 %;
оформление пояснительной записки и графического материала к 14 декабря 2020 г. 20 %
Защита курсового проекта с 21 декабря 2020 г. по 28 декабря 2020 г.

РУКОВОДИТЕЛЬ _____ *Ю.Ю. Желтко*
(подпись)

Задание принял к исполнению _____ *Д.И. Кукла*

(дата и подпись студента)

я

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Транспортная инфраструктура - одна из важнейших инфраструктур, обеспечивающих жизнь городов и регионов. В последние десятилетия во многих крупных городах исчерпаны или близки к исчерпанию возможности экстенсивного развития транспортных сетей. Поэтому особую важность приобретает оптимальное планирование сетей, улучшение организации движения, оптимизация системы маршрутов общественного транспорта. Решение таких задач невозможно без математического моделирования транспортных сетей.

Темой данного курсового проекта является «Моделирование трафика дорожного движения». Данный проект представляет собой графическое приложение, содержащее возможности создания моделей дорожного движения с помощью инструментов, предоставленных в конструкторе. Созданную модель можно запустить в виде анимации, наглядно демонстрирующей ее поведение.

Логика поведения модели основана на агентно-ориентированном подходе(в дальнейшем АОП) к программированию. АОП – это частный случай объектно-ориентированного подхода, в основе которого используется понятие агента и его поведения, зависящего от среды нахождения. Агент – это объект, который может воспринимать свою среду с помощью датчиков и воздействовать на нее с помощью исполнительных механизмов.

В процессе разработки рассматриваются методы реализации поведения дорожного транспорта в среде дорожного движения, исследуется возможности отображения модели движения на экран монитора с помощью средств графики и способы создания графического интерфейса для комфортного использования полученного приложения пользователем.

Для реализация элементов графического интерфейса был использован фреймворк Qt. Qt – это программное обеспечение, включающее в себя большое количество различных возможностей для упрощения создания графических приложений, в том числе базовые классы, используемые в основе собственных элементов управления. Для создания анимация поведения модели используется OpenGL. OpenGL – спецификация, определяющая платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трехмерную графику.

Данное приложение будет полезно для использования при сравнении и анализе вариантов пересечений и примыканий дорого и автомагистралей; выборе оптимальной схемы организации движения на перекрестке и оценке пропускной способности для различных вариантов движен; оптимизации работы светофорных объектов; прогнозировании, анализе и ликвидации «узких» мест на улично-дорожной сети; прогнозировании, анализе и ликвидации автотранспортных пробок; создании наглядной презентации об организации дорожного движения.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Анализ существующих аналогов

Тема курсового проекта была выбрана в первую очередь для получения знаний в области объектно-ориентированного программирования и проектировании пользовательских приложений, поэтому моей целью не является разработать конкурентоспособный продукт. Тем не менее, чтобы создать корректно работающее приложение, необходимо иметь представление об уже реализованных аналогах и основных решениях, используемых внутри продукта.

1.1.1 Библиотека дорожного движения AnyLogic

Библиотека дорожного движения – это инструмент планирования и организации транспортных потоков. В моделях дорожного движения имитируется перемещение машин по улицам и автомагистралям, включая такие элементы как перекрестки, пешеходные переходы, круговое движение, автостоянки и остановки общественного транспорта.

Рисунок 1.1 – Скриншот анимации из AnyLogic

Каждое транспортное средство моделируется в виде агента, который имеет индивидуальные физические параметры, такие как длина, скорость и ускорение. Поведение ТС моделируется с помощью диаграмм процессов, которые легко создавать благодаря функции «drag-and-drop».

Библиотека представляет пользователю инструменты для моделирования перекрестков неравнозначных дорог, светофоров, пешеходных переходов, автобусных остановок и автостоянок.

Для наглядности в моделях дорожного движения можно создавать 2D- и 3D-анимацию. Также было установлено, что данный продукт написана на языке программирования C++.

Плюсы данной библиотеки:

- большое количество инструментов;
- гибкая настройка моделей;
- большое количество готовых примеров симуляции;
- возможность 2D- и 3D-анимации.

Минусы:

- небольшая сложность в освоении;
- платный доступ;
- закрытый исходный код.

1.1.2 Виртуальный мир Carcraft.

Данный продукт был разработан компанией Waymo в качестве тестировочной площадки для симуляции поведения разрабатываемых ими же беспилотных автомобилей в различных ситуация дорожного движения.

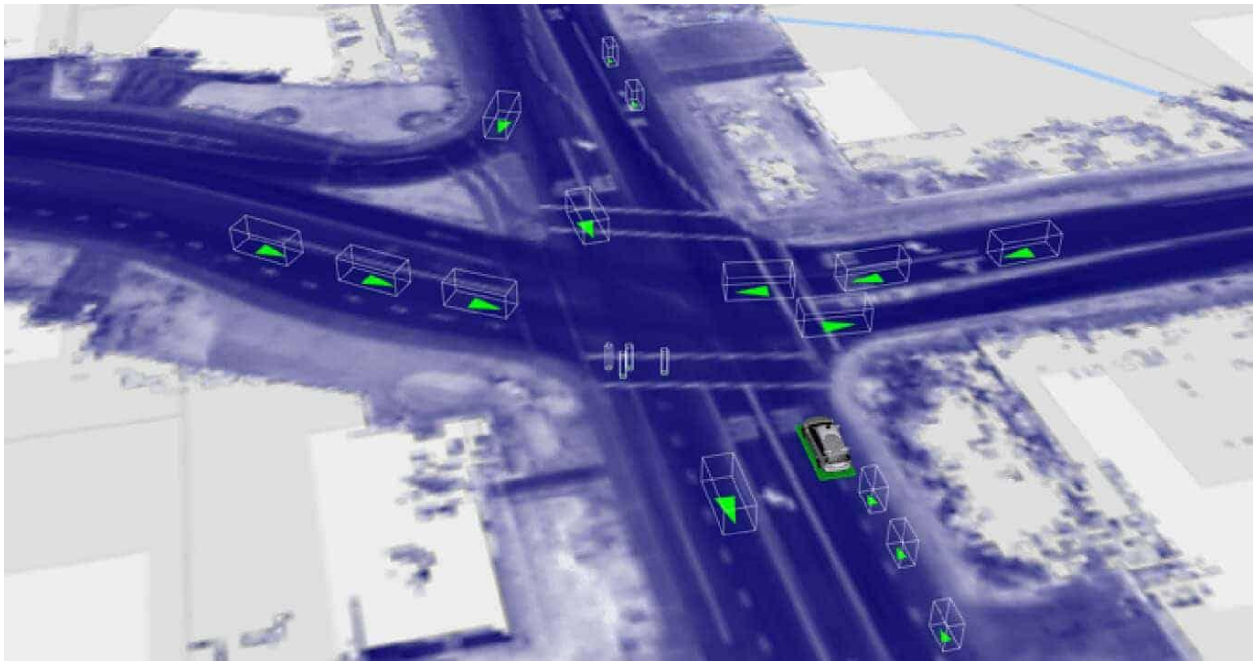

Рисунок 1.2 – Изображение из Carcraft

Участники дорожного движения в Carcraft ведут себя точно также, как и в реальном мире. Остальные же объекты являются лишь отражением реальных объектов: они представлены в том виде, в котором искусственный интеллект

воспринимает их. В данном мире происходит миллионы различных симуляций для обучения искусственного интеллекта Waymo.

Данный проект нельзя найти в открытом доступе и о внутренней реализации ничего не известно.

Плюсы Carcraft:

- полная симуляция поведения реального мира;
- визуальное представление;
- обширная создаваемая карта;

Минусы:

- отсутствие в открытом доступе;
- специализированная область применения.

1.1.3 Игра A/B Street.

A/B Street – это игра, в которой игроку выдаются задачи по оптимизации движения различных участников дорожного движения, которые включают в себя оптимизация фаз работы светофоров, путей пересечения перекрестков, расположению дорог.

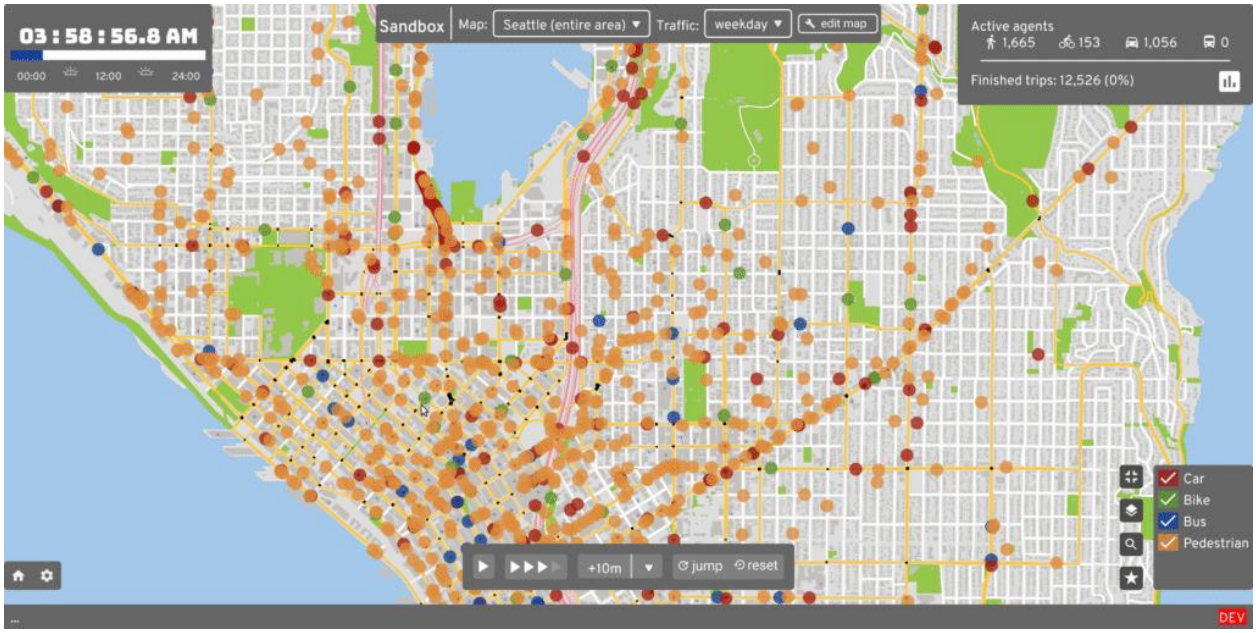

Рисунок 1.3 – Скриншот из A/B Street

Данная игра также включает в себя режим песочницы, в котором воспроизведены карты различных городов мира, можно импортировать собственную карт и существует возможность разнообразных изменений в структуре города, которые могут оптимизировать движение дорожного траффика и наглядно это продемонстрировать.

Симуляция реализована с помощью моделирования дискретных событий, когда агенты (автомобили, пешеходы, автобусы) реагируют на

окружающую среду не каждую секунду, а только в случае появления событий, которые могут повлиять на внутренне состояние объекта.

Игра создана с использованием языка программирования Rust.

Плюсы A/B Street:

- легкость в освоении;
- открытый исходный код.

Минусы:

- нет внутреннего конструктора карт.

1.2 Постановка задачи

После рассмотрения аналогов можно сказать, что все они обладают большим количеством функций, которые невозможно реализовать в курсовом проекте за данный период времени.

Поэтому были выбраны несколько ключевых возможностей, которые будут выполнены в рамках одного семестра. Ими являются:

- пользовательский интерфейс;
- конструктор, позволяющий создавать модели из реализованных инструментов с гибкой настройкой;
- 2D-симуляция модели;
- логика поведения модели.

В основу логической составляющей проекта лег агентно-ориентированный подход, что является частным случаем объектно-ориентированном подхода к программированию. Это концепция очень сильно сочетается с представлением реального мира и позволяет с большей легкостью создавать модель с поведением, похожим на реальность.

В качестве языка программирования выбран C++, по причине быстрой производительности, требуемой для обработки большого количества объектов, поддержки объектно-ориентированного подхода к программированию и наличия опыта в использования данного языка.

В качестве реализация графического интерфейса используется фреймворк Qt, основанный на языке C++. Qt обладает большим количеством базовых классов, которые позволяют создавать собственные классы для реализации графического интерфейса, удобной системой общения между виджетами приложения с помощью системы сигналов и слотов и хорошей документацией, позволяющей в быстрые сроки разбираться в устройстве Qt. Также данное средство обладает поддержкой OpenGl, который используется в качестве средства для создания графики.

Как упоминалось выше, для реализации графики выбран программный интерфейс OpenGl. Данное средство выбрано в силу простоты освоения среди аналогов и возможности очень гибкой графики, подходящей для моего проекта.

Данный список средств позволяет реализовать все задачи, выбранные для курсового проекта.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

После определения требований к функционалу разрабатываемого приложения его следует разбить на функциональные блоки. Такой подход упростит понимание приложения, позволит устранить проблемы в архитектуре, обеспечит гибкость и масштабируемость приложения в будущем путем добавления новых блоков.

2.1 Логический модуль

Логический модуль отвечает за корректное поведение системы в соответствии правилами дорожного движения. Он составление из объектов, моделирующих поведение и свойство предметов реального мира. Данная часть приложения может корректно обрабатывать поведение всех объектов, участвующих в дорожного движении.

В данный модуль включены следующие функции и возможности:

- получение информации об устройстве модели дорожного движения и ее корректной обработки.
- передача текущего состония системы для отображения в графический модуль.

2.2 Графический модуль

Графический модуль отвечает за отображение на экран состояния системы в данный момент времени. Внутри него реализована полная абстракция возможностей интерфейса OpenGL для простого создания графики.

Задача данного модуля состоит в корректном и быстром отображении данных, полученной из логического модуля. Также реализована возможность предоставления дополнительной информации из этих данных: графики, статистика.

2.3 Модуль пользовательского интерфейса

Модуль пользовательского интерфейса предназначен для предоставления пользователю инструментов создания модели дорожного движения.

В данный модуль включены различные шаблоны для создания транспортных сетей, расстановки точек генерации автомобилей, расположения светофоров. Каждый шаблон имеет ряд параметров, для их настройки, что предоставляет пользователю возможности быстрого создания моделей систем дорожного движения, информация об которых передается в графический модуль для дальньшей обработки.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается функционирование и структура разрабатываемого приложения.

3.1 Логический модуль

3.1.1 Класс Vector

Класс реализует математический вектор в двумерном пространстве и основные математические операции над ним: сложение, вычитание, умножение на число, скалярное произведение.

3.1.2 Класс Particle2D

Класс предоставляет интерфейс для объектов, поведение которых схоже с частицами.

`Particle2D` содержит поля:

- `Vector m_Position` – Положение частицы в пространстве
- `Vector m_Velocity` – Скорость частицы.
- `Vector m_Acceleration` – Ускорение частицы.
- `Vector m_ForceAccumulator` – Вектор, отвечающий за накопление сил, действующих на объект.
- `float m_InverseMass` – единица, деленная на массу частицы.
- `float m_Damping` – величина, отвечающая за торможение частицы при движении.

Основные методы:

- `virtual void applyForce(Vector force)`:

`Force` - вектор силы, действующей на частицу.

Метод воздействует на частицу с заданной силой.

- `virtual void move(float duration)`:

`duration` - время, за которое изменяется положение частицы.

Метод изменяет положение частицы в течение периода времени duration.

3.1.3 Класс Car

Класс наследуется от класса `Particle2D` и переопределяет виртуальные методы.

Класс содержит поля:

- `double m_MaxSpeed` – число, ограничивающее максимальную скорость машины.

- `State m_State` – переменная, хранящее текущее состояние машины.

State может содержит флаги `LIVE, TURN, FOLLOW, YIELD, STOP.`

Класс переопределяет методы `applyForce`, move, основное назначение которых не изменилось.

Другие основные методы:

- `bool view(const Car& car, double startAngle, double endAngle, double distance)`:

`car` – объект машины.

`startAngle` - начальный угол обзора.

`endAngle` – конечный угол обзора.

`Distance` - дистанция обзора.

Метод возвращает true, если `car` находится в поле зрения объекта, иначе возвращается false.

- `bool view(const Trafficlight& trafficlight, double startAngle, double endAngle, double distance)`:

`trafficlight` – объект светофора.

Данный метод является перегрузкой `view(const Car& car, double startAngle, double endAngle, double distance).`

- `void followPath(const Road* road, Vector* position)`:

`road` – дорога вдоль, которой следует машина.

`position` – позиция, в какую точку следует ехать машине. Изменяется методом.

Метод вычисляет точку, в которую машина должна двигаться возвращает в качестве параметра position.

3.1.4 Класс Road

Класс реализует объект дороги.

Включает поля:

- `std::vector<Vector> m_Points` – массив точек, составляющих дорогу.

- `double m_Radius` – ширина дороги.

- `unsigned int m_ID` – уникальный ID, по которому можно определить дорогу в системе.

- `bool m_Main` – флаг, обозначающий вид дороги.

В классе реализуются различные методы для создания дороги и получения информации о ней.

3.1.5 Класс CarGenerator

Класс отвечает за появление новых в машин в определенный период времени.

Содержит поля:

- `Vector m_Position` – расположение начальной точки генерации.
- `int m_Rate` – частота генерации. Измеряется в миллисекундах.
- `unsigned int m_StartRoadID` – ID начальной дороги.
- `float m_Time` – текущее время.

Класс содержит методы для установления позиции, частоты генерации и ID дороги и также методы для их получения.

Также реализован метод:

- `void update(CarRegistry &cars):`

`cars` – реестр машин, в который заносятся новые машины.

Метод добавляет автомобили в реестр cars.

3.1.6 Класс TrafficLight

Класс реализует объект светофора.

Содержит поля:

- `Vector m_Position` – расположение светофора.
- `int m_Period[4]` – периоды работы светофора.
- `State m_State` – текущее состояние светофора.
- `State m_StartState` – начальное состояние светофора.
- `float m_Time` – текущее время.

Класс содержит методы для установления позиции, частоты генерации и ID дороги и также методы для их получения.

Также реализован метод:

- `void update():`

Метод обновляет состояние светофора.

3.1.7 Класс ForceGenerator

`ForceGenerator` – интерфейс по созданию сил, который содержит только метод:

`void updateForce(Particle2D particle, const float duration)`:

`particle` – частица, для которой создается сила.

`duration` – период действия силы.

Метод создает силу, действующую на частицу particle в течение duration.

3.1.8 Класс SteerForceGenerator

Класс наследуется от `ForceGenerator` и переопределяет метод `updateForce`. Данный класс предназначен для создания управляющей силы.

Включает поле:

- `Vector target` – точка, в которую направлена сила.

3.1.9 Класс BrakeForceGenerator

Класс наследуется от `ForceGenerator` и переопределяет метод `updateForce`. Данный класс предназначен для создания тормозящей силы, если на движения автомобиля находится препятствие.

Включает поле:

- `Vector desiredVelocity` – желаемая скорость после воздействия силы.
- `float distance` – расстоние до объекта столкновения.

3.1.10 Класс RoadRegistry

Класс `RoadRegistry` предназначен для хранения реестра дорог и их соединений.

Включает поля:

- `std::vector<std::vector<unsiged int >> connections` – граф соединений дорог.
- `std::vector<std::unique_ptr<Road>> roads` – хранилище всех дорог в системе.

- `std::unordered_map<unsigned int, unsigned int> m_hash` – отображение, для хранения пар (внутреннее ID дороги изнутри `CarRegistry`, ID дороги для остальной системы).

В классе реализованы методы по добавлению дороги в реестр, соединений дорог, получении информации о дорогах и их соединениях.

3.1.11 Класс CarRegistry

Класс предназначен для хранения всей информации об автомобилях системы.

Содержит поле:

- `std::list<CarInformation> m_car` – список информации об машинах.

`CarInformation` представляет собой структуру со следующими полями:

- `Car car` – автомобиль.
- `SteerForceGenerator steerForceGenerator` – генератор управляющей силы, привязанный к автомобилю.
- `BrakeForceGenerator brakeForceGenerator` – генератор тормозящей силы, привязанный к автомобилю.
- `unsigned int roadID` – ID дороги, по которой движется машина.

Класс реализует методы добавления и получения информации о машинах.

Основной метод:

- `void update(const RoadRegistry &roads, const std::vector<const Trafficlight*> &lights, float time)`:

`roads` – реестр дорог.

`lights` – список светофоров.

`time` – время, за которое происходит обновление.

Метод отвечает за удаление машин, обновления генераторов сил и движения машин.

3.1.12 Класс MDLParser

Класс отвечает за получение внешней информации из файла об устройстве модели.

Реализован статический метод:

- `static const ParseInformation parseFile(const std::string& filePath)`:

`filePath` – путь до файла.

Возвращает структуру `ParseInformation`, содержащейся в файле.

`ParseInformation` содержит поля:

- `RoadRegistry* roadRegistry` – информация о расположении дорог и их соединений.
- `std::vector<CarGenerator*> CarGenerators` – массив генераторов машин.
- `std::vector<Trafficlight*> TrafficLights` – массив светофоров.

3.2 Графический модуль

3.2.1 Класс VertexBuffer

Класс реализует абстракцию для буфера вершин из API OpenGL.

3.2.2 Класс IndexBuffer

Класс реализует абстракцию для буфера элементов из API OpenGL.

3.2.3 Класс VertexBufferLayout

Класс реализует абстракцию для расположения вершин в буфере вершин из API OpenGL.

3.2.4 Класс VertexArray

Класс реализует абстракцию для массива вершин из API OpenGL.

3.2.5 Класс Shader

Класс реализует абстракцию для шейдера из API OpenGL.

3.2.6 Класс Renderer

Класс реализует абстракцию для команд рисования из API OpenGL.

3.2.7 Класс OrthographicCamera

Класс реализует абстракцию ортогональной камеры.

3.2.8 Класс OpenglLayer

Класс реализует слой приложения, отвечающий за изображение всех графической информации об состоянии системы дорожного траффика.

3.2.9 Класс Window

Класс реализует абстракцию окна библиотеки GLFW.

3.2.10 Класс ModelApplication

Класс реализован для объединения всей графической и логической составляющей воедино. Отвечает за связь с модулем пользовательского интерфейса.

Содержит поля:

- `OpenglLayer* m_OpenglLayer` – слой Opengl.
- `Windoe* m_Window` – окно приложения.
- `std::unique_ptr<RoadRegistry> m_RoadRegistry` – реестр машин.
- `std::unique_ptr<CarRegistry> m_CarRegistry` – реестр дорог.
- `std::vector <std::unique_ptr<CarGenerator>> m_CarGenerators` – массив генераторов машин.
- `std::vector <std::unique_ptr<TrafficLight>> m_TrafficLights` – массив светофоров.

В классе реализован метод:

- `void run()` – метод реализует цикл обновления состояния системы дорожного траффика

3.3 Модуль пользовательского интерфейса

3.3.1 Класс WorkspaceScene

Класс наследуется от `QGraphicsScene` и реализует дополнительные функции.

Переопределен метод:

- `void mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent)`:

`mouseEvent` – указатель на событие мыши.

Метод вызывает функции добавления и удаления генераторов машин, точек дороги, светофоров на сцене, соединения и отсоединения объектов, вызывает контекстное меню.

3.3.2 Класс Workspace

Класс наследуется от `QGraphicsView` и реализует дополнительные функции.

Переопределены методы:

- `void mousePressEvent(QGraphicsSceneMouseEvent *mouseEvent)`
- `void mouseMoveEvent(QGraphicsSceneMouseEvent *mouseEvent)`
- `void mouseReleaseEvent(QGraphicsSceneMouseEvent *mouseEvent)`:

`mouseEvent` – указатель на событие мыши.

Данная совокупность методов реализует функцию выделения объектов.

Реализован сигнал:

- `void signalModified()`- метод сообщает об изменении сцены.

3.3.3 Класс MainWindow

Класс реализует главное окно пользовательского интерфейса и наследуется от `QMainWindow`.

Содержит поля:

- `Workspace* m_View` – указатель на объект отображения сцены.
- `WorkspaceScene* m_Scene` – указатель на объект сцены.
- `QString m_CurFile` – расположение текущего файла.
- `QMenu* m_ItemMenu` – указатель на объект контекстного меню.
- `QDockWidget* m_DockWidget` – указатель на виджет настройки.

В классе реализованы методы работы с файлами: сохранения, создания, открытия. Так же класс предоставляет интерфейс для возможностей реализованных `Workspace` и `WorkspaceScene`

3.3.4 Класс RoadPoint

Класс наследуется от `QGraphicsItem` и реализует объект точки дороги на графической сцене.

3.3.5 Класс Road

Класс наследуется от `QGraphicsItem` реализует объект дороги на графической сцене.

3.3.6 Класс CarGenerator

Класс наследуется от `RoadPoint` и реализует объект генератора автомобилей на графической сцене.

3.3.7 Класс Trafficlight

Класс наследуется от `RoadPoint` и реализует объект светофора на графической сцене.

3.3.8 Класс CarGeneratorWindow

Класс наследуется от `QWidget` и реализует виджет настройки генератора автомобилей.

3.3.9 Класс TrafficlightWindow

Класс наследуется от `QWidget` и реализует виджет настройки светофора.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

4.1 Метод void CarRegistry::update(const RoadRegistry &roads, float time)

Шаг 1. Начало.

Шаг 2. Инициализируем итератор std::list<CarInformation>:: iterator it значением cars.begin().

Шаг 3. Если значение it равно cars.end(), переходим к шагу.

Шаг 4. Вызываем метод interectLights с параметрами: it, lights.

Шаг 5. Вызываем метод yieldInterect с параметрами: it, roads.

Шаг 6. Вызываем метод turn с параметрами: it, roads.

Шаг 7. Вызываем метод interectFollow с параметрами: it, roads.

Шаг 8. Если флаг LIVE состояния it->car равен лжи, переходим к шагу 9, иначе 10.

Шаг 9. Присваиваем значение it равным значению, возвращаемым методом m_Cars.erase с параметром it. Переходим к шагу 3.

Шаг 10. Вызывем метод applySteerForce с параметрами: it->car, it->steerForceGenerator, roads.getRoad(it->roadID).

Шаг 11. Вызывем метод applySteerForce с параметрами: it->car, it->steerForceGenerator, roads.getRoad(it->roadID).

Шаг 12. Увеличиваем значение итератора it на 1. Переходим к шагу 3.

Шаг 13. Устанавливаем значение итератора it равным cars.begin().

Шаг 14. Если значение it равно cars.end(), переходим к шагу 17.

Шаг 15. Вызываем метод it->car.move с параметром time.

Шаг 16. Увеличиваем значение итератора it на 1. Переходим к шагу 14.

Шаг 17. Конец.

4.2 Метод void CarRegistry::interectLights(std::list<CarInformation>::iterator& it, const std::vector<const TrafficLight*> &lights)

Шаг 1. Начало.

Шаг 2. Инициализируем bool stop_fl значением false.

Шаг 3. Инициализируем int i значением 0.

Шаг 4. Если i меньше lights.size(), переходим к шагу 5, иначе 14.

Шаг 5. Если значение, возвращаемое методом it→car.view с параметрами: *lights[i], углом Constants.lightViewAngle.start, Constants.lightViewAngle.end, расстоянием Constants.lightViewDistance — равно истине переходим к шагу 6, иначе 13.

Шаг 6. Если значение, возвращаемое методом lights[i]->getState() равно значению TrafficLight::State::Red или равно значению TrafficLight::State::Yellow, переходим к шагу 7, иначе 13

Шаг 7. Вызываем метод it→car.setState с параметрами: STOP, true.

Шаг 8. Инициализируем Vector desired значением, возвращаемым методом it->car.getVelocity.

Шаг 9. Устанавливаем длину desired равной значению, возвращаемым методом view_check->car.getMaxSpeed.

Шаг 10. Вызываем метод it->brakeForceGenerator.init с параметрами: desired, расстоянием между view_check->car и it->car.

Шаг 11. Устанавливаем значение follow_fl равным true.

Шаг 12. Переходим к шагу 14.

Шаг 13. Увеличиваем значение i на 1. Переходим к шагу 5.

Шаг 14. Если значение red_fl равно лжи, вызываем метод it->car.setState с параметрами: STOP, false.

Шаг 15. Конец.

4.3 Метод void CarRegistry::yieldInterect(std::list<CarInformation>::iterator &it, const RoadRegistry &roads)

Шаг 1. Начало.

Шаг 2. Если флаг Yield состояния it->car равен истине и флаг FOLLOW равен лжи, переходим к шагу 3, иначе 22.

Шаг 3. Инициализируем итератор std::list<CarInformation>:: iterator view_check значением cars.begin().

Шаг 4. Инициализируем bool stop_fl значением false.

Шаг 5. Если значение view_check равно cars.end(), переходим к шагу 21.

Шаг 6. Если it->car равен view_check, переходим к шагу 20.

Шаг 7. Инициализируем bool fl значение false.

Шаг 8. Инициализируем std::vector<unsigned int> connections значением, возвращаемым методом roads.getRoadConnections с параметром view_check→roadID.

Шаг 9. Инициализируем int i значением 0.

Шаг 10. Если i меньше connections.size(), переходим к шагу 11, иначе 13.

Шаг 11. Если connections[i] равно it->roadID, Устанавливаем значение fl равным true. Переходим к шагу 13.

Шаг 12. Увеличиваем значение i на 1.

Шаг 13. Если значение fl равно истине, переходим к шагу 20.

Шаг 14. Если значение, возвращаемое методом it->car.view с параметрами: view_check->car, углом Constants.yieldViewAngle.start, Constants.yieldViewAngle.end, расстоянием Constants.yieldViewDistance или значение, возвращаемое методом it->car.view с параметрами: view_check->car, углом Constants.yieldSideViewAngle.start, Constants.yieldSideViewAngle.end, расстоянием Constants.yieldSideViewDistance — равно истине переходим к шагу 15, иначе 20.

Шаг 15. Вызываем метод it->car.SetState с параметрами: STOP, true.

Шаг 16. Инициализируем Vector desired значением, возвращаемым методом it->car.getVelocity.

Шаг 17. Устанавливаем длину desired равной Constants.stopSpeed .

Шаг 18. Вызываем метод it->brakeForceGenerator.init с параметрами: desired, расстоянием между view_check->car и it->car.

Шаг 19. Устанавливаем значение stop_fl равным true. Переходим к шагу 22.

Шаг 20. Увеличиваем значение итератора view_check на 1. Переходим к шагу 5.

Шаг 21. Если значение stop_fl равно false, вызываем метод it→car.setState с параметрами: STOP, false.

Шаг 22. Конец.

4.4 Метод void CarRegistry::turn(std::list<CarInformation>::iterator &it, const RoadRegistry &roads)

Шаг 1. Начало.

Шаг 2. Если флаг TURN состояния it->car равен истине, переходим к шагу 3, иначе 12.

Шаг 3. Инициализируем std::vector<unsigned int> connections значением, возвращаемым методом roads.getRoadConnections с параметром it→roadID.

Шаг 4. Если размер connections равен нулю, переходим к шагу 11.

Шаг 5. Инициализируем double probability значением (std::rand() % 100) / 100.

Шаг 6. Инициализируем int i значением 0.

Шаг 7. Если probability больше значения 1 / размер connections, переходим к шагу 8, иначе 10

Шаг 8. Уменьшаем probability на значение 1 / размер connections.

Шаг 9. Увеличиваем значение i на 1. Переходим к шагу 7.

Шаг 10. Устанавливаем значение it->roadID равным i.

Шаг 11. Вызываем метод it->car.SetState с параметрами: LIVE, false.

Шаг 12. Конец.

4.5 Метод void CarRegistry::interectFollow(std::list<CarInformation>::iterator &it, const RoadRegistry &roads)

Шаг 1. Начало.

Шаг 2. Если флаг Yield состояния it->car равен лжи, переходим к шагу 3, иначе 22.

Шаг 2. Инициализируем итератор std::list<CarInformation>:: iterator view_check значением cars.begin().

Шаг 3. Инициализируем bool follow_fl значением false.

Шаг 4. Если значение view_check равно cars.end(), переходим к шагу 21.

Шаг 5. Если it->car равен view_check, переходим к шагу 20.

Шаг 6. Инициализируем bool fl значение false.

Шаг 7. Инициализируем std::vector<unsigned int> connections значением , возвращаемым методом roads.getRoadConnections с параметром view_check->roadID.

Шаг 8. Инициализируем int i значением 0.

Шаг 9. Если i меньше connections.size(), переходим к шагу 10, иначе 12.

Шаг 10. Если connections[i] равно view_check->roadID, Устанавливаем значение fl равным true. Переходим к шагу 12.

Шаг 11. Увеличиваем значение i на 1.

Шаг 12. Если значение it->roadID равно значению view_check->roadID, устанавливаем значение fl равным истине.

Шаг 13. Если значение fl равно лжи, переходим к шагу 20.

Шаг 14. Если значение, возвращаемое методом it->car.view с параметрами: view_check->car, углом Constants.followViewAngle.start, Constants.followViewAngle.end, Constants.followViewDistance — равно истине переходим к шагу 15, иначе 20.

Шаг 15. Вызываем метод it->car.SetState с параметрами: FOLLOW, true.

Шаг 16. Инициализируем Vector desired значением, возврашаемым методом it->car.getVelocity.

Шаг 17. Устанавливаем длину desired равной значению, возвращаемым методом view_check->car.getMaxSpeed.

Шаг 18. Вызываем метод it->brakeForceGenerator.init с параметрами: desired, расстоянием между view_check->car и it->car.

Шаг 19. Устанавливаем значение follow_fl равным true. Переходим к шагу 22.

Шаг 20. Увеличиваем значение итератора view_check на 1. Переходим к шагу 4.

Шаг 21. Если значение follow_fl равно false, вызываем метод it->car.setState с параметрами: FOLLOW, false.

Шаг 22. Конец.

4.6 Метод void Car::move(float time)

Шаг 1. Начало.

Шаг 2. Приравниваем значение acceleration равным нулевому вектору.

Шаг 3. Увеличиваем значение acceleration на forceAccumulator.

Шаг 4. Увеличиваем значение velocity на acceleration * time.

Шаг 5. Ограничиваем модуль velocity значением, возвращаемым методом getMaxSpeed().

Шаг 6. Увеличиваем значение position на velocity * time.

Шаг 7. Вызываем метод clearAccumulator().

Шаг 8. Конец.

4.7 Метод void SteerForceGenerator::updateForce(Paricle2D* particle, float time)

Шаг 1. Начало.

Шаг 2. Инициализируем Car* car значением particle.

Шаг 3. Инициализируем Vector desired значением target – car->getPosition().

Шаг 4. Если модуль desired равен нулю, переходим к шагу 10.

Шаг 5. Инициализируем double lim значением, возвращаемым методом car->getMaxSpeed().

Шаг 6. Устанавливаем модуль desired равным значению lim.

Шаг 7. Инициализируем Vector steer значением desired – car->getVelocity().

Шаг 8. Ограничиваем модуль steer величиной lim.

Шаг 9. Вызываем метод car->applyForce с параметром steer.

Шаг 10. Конец.

4.8 Метод void BrakeForceGenerator::updateForce(Paricle2D* particle, float time)

Шаг 1. Начало.

Шаг 2. Инициализируем Car* car значением particle.

Шаг 3. Инициализируем Vector velocity значением, возвращаемым методом car->getVelocity().

Шаг 4. Инициализируем Vector desiredVelocity значением, возвращаемым методом car->getVelocity().

Шаг 5. Устанавливаем модуль desiredVelocity равным модулю desired.

Шаг 6. Инициализируем Vector brake.

Шаг 7. Если модуль desiredVelocity меньше Velocity, переходим к шагу 8, иначе 9.

Шаг 8 Устанавливаем значение brake равным desiredVelocity – velocity.

Шаг 9. Вызываем метод car->applyForce с параметром brake.

Шаг 10. Конец.

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

5.1 Тестирование работы с файлами

При невозможности открытия пользователем файла, некорректных данных, записанных в файл, невозможности записи в файл возникает сообщение об ошибке (рисунок 5.1.1). (рисунок 5.1.2).

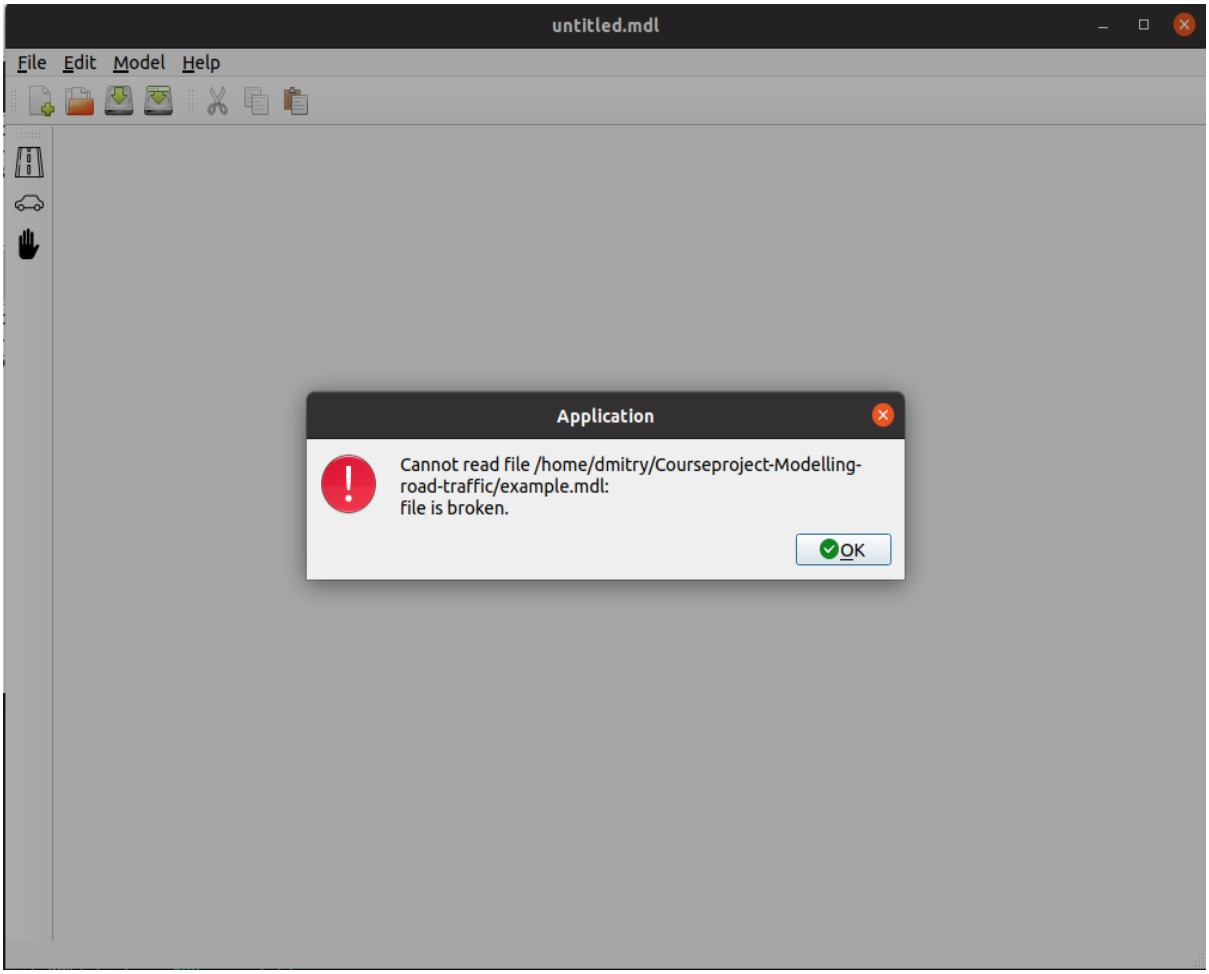

Рисунок 5.1.1 – Ошибка при неправильных данных

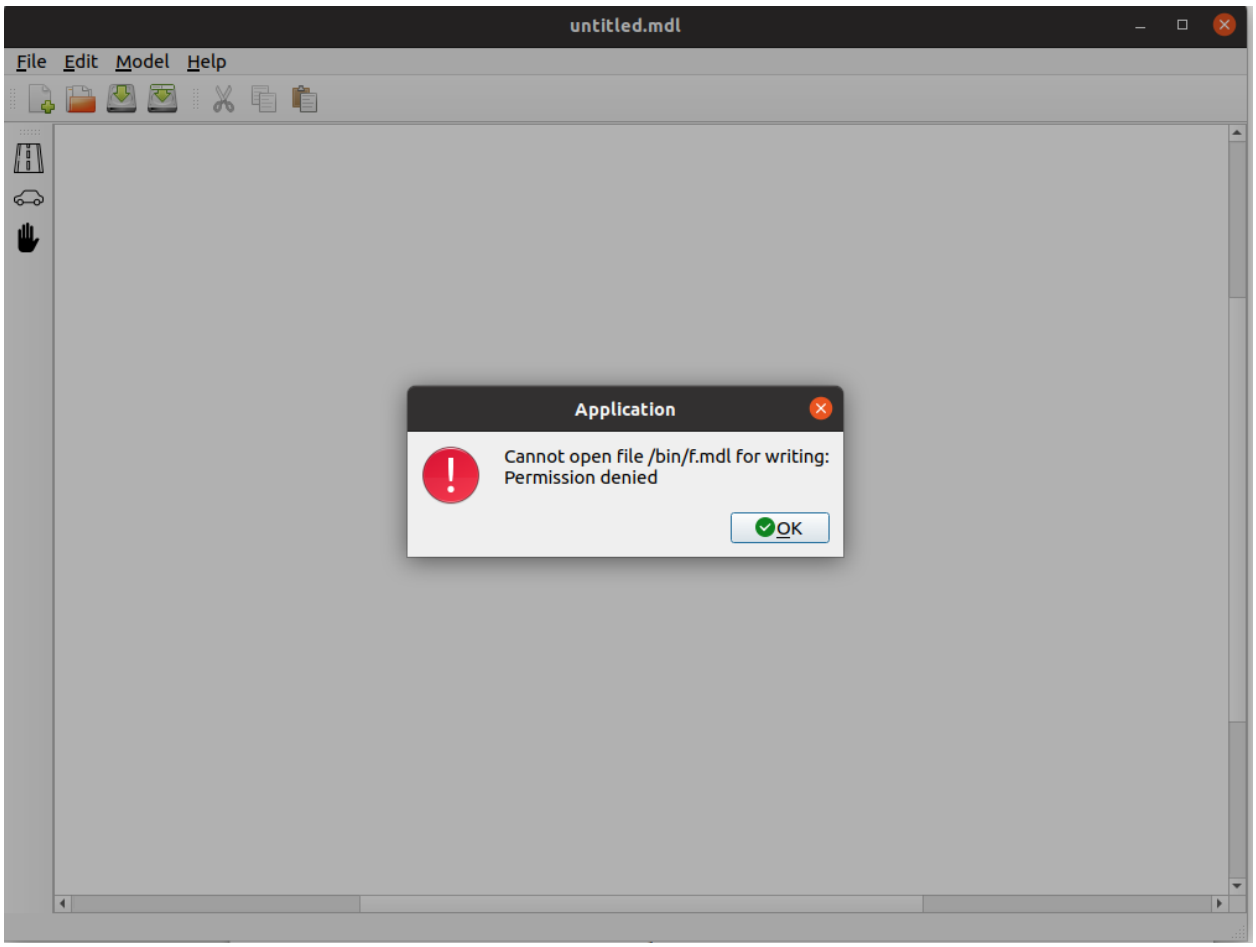

Рисунок 5.1.2 – Ошибка при отказе записи.

5.2 Тестирование работы приложения при ошибках, с функциями OpenGL

О любой ошибке, возникающей внутри функций OpenGL, сообщается в консоль. Данный информация будет полезна разработчику для устранения неполадок программы.

В рисунке 5.2.1 приведен результат ошибки, вызванной отсутствием необходимых шейдеров:

```
dmitry@dmitry-Lenovo:~/Courseproject-Modelling-road-traffic$ ./main test1.mdl
[OpenGL error] (1282): glUseProgram(m_RendererID) /home/dmitry/Courseproject-Mod
elling-road-traffic/src/opengl/Shader.cpp:69
main: /home/dmitry/Courseproject-Modelling-road-traffic/src/opengl/Shader.cpp:69
: void Shader::bind() const: Assertion `GLLogCall("glUseProgram(m_RendererID)",
__FILE__, __LINE__)' failed.
Aborted (core dumped)
```

Рисунок 5.2.1 – Ошибка при отсутствии шейдеров

В рисунке 5.2.2 приведен результат ошибки, вызванной отсутствием функции создания буффера:

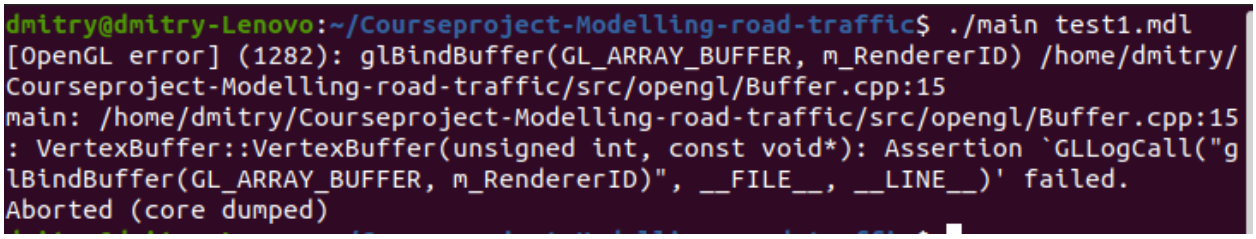

Рисунок 5.2.2 – Ошибка при отсутствии функции создания буффера

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Для запуска программы необходимо открыть файл «Road Traffic.exe». После этого откроется окно программы (рисунок 6.1).

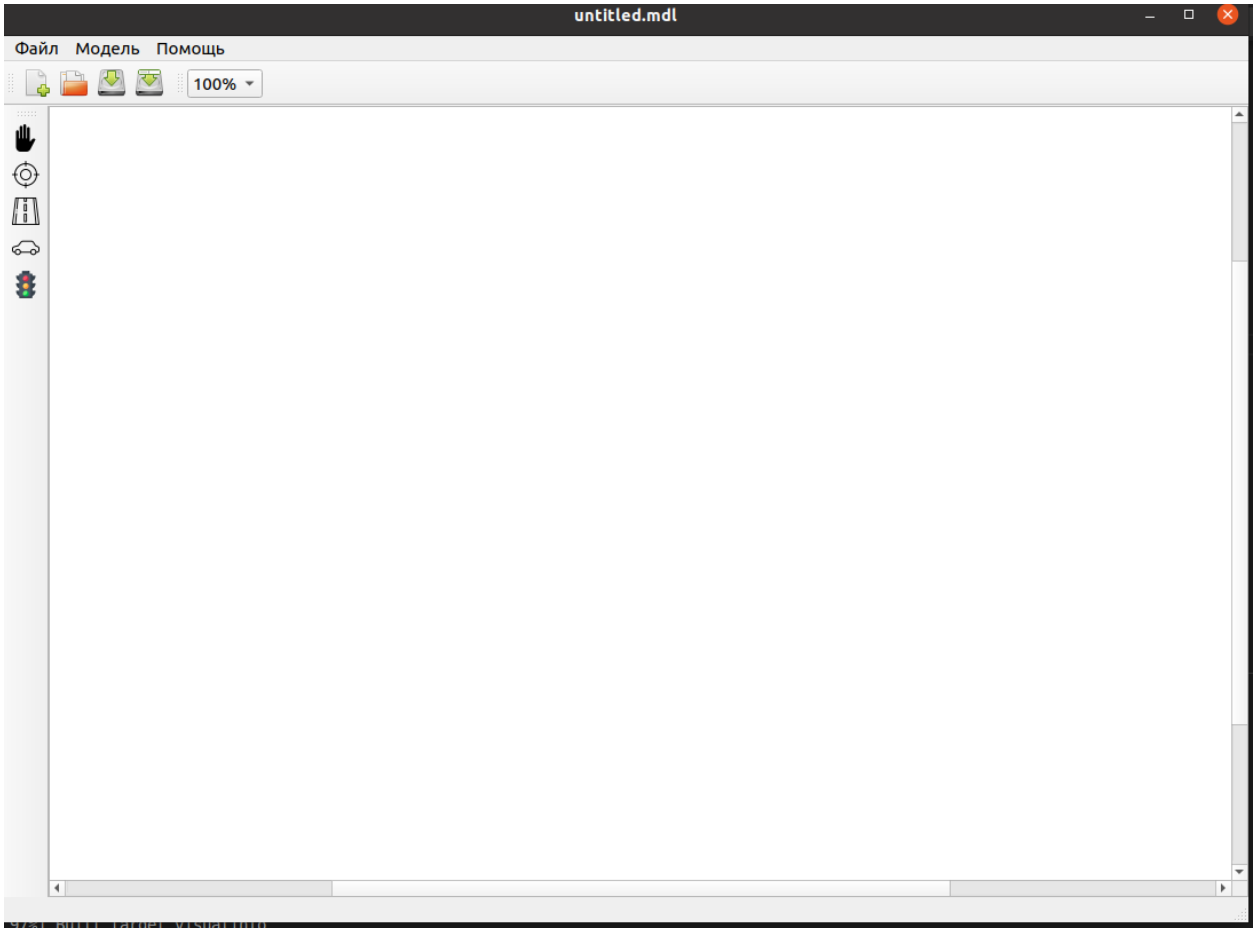

Рисунок 6.1 – Окно программы.

Окно состоит из основных компонент: меню, панель инструментов, рабочей области и окна настройки (рисунок 6.2).

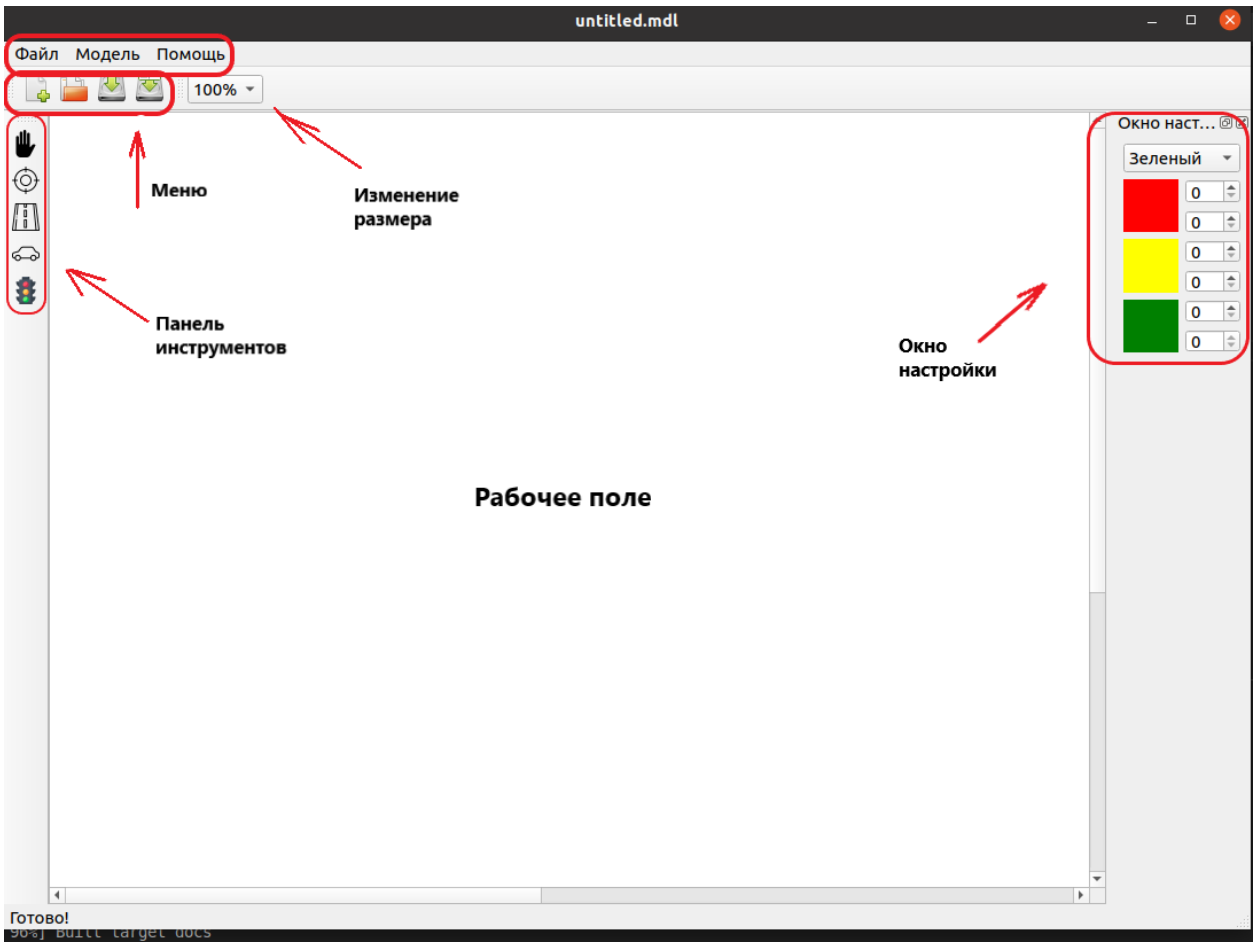

Рисунок 6.2 – Компоненты программы

При нажатии на меню «Файл» возникает окно с действиями:

- Новый файл(Ctrl + N) — создает новый файл.
- Открыть файл(Ctrl + O) — открывает файл.
- Сохранить файл(Ctrl + S) — сохраняет файл.
- Сохранить как(Ctrl + Shift + S) — сохраняет файл в указанном месте.
- Exit(Ctrl + Q) — выходит из программы.

Аналогичные им действия, расположены на панели инструментов вверху рабочей зоны.

При нажатии на меню «Модель» возникает окно с действиями:

- Запустить — запускает текущую модель.

При нажатии на меню «Помошь» возникает окно с действиями:

- Об приложении — при нажатии возникает окно с информации об приложении.
- Об Qt — при нажатии возникает окно с информации о Qt.

Слева расположена панель инструментов, включающая:

- Инструмент руки — при нажатии приложение переходит в обычный режим. В данном режиме доступны действия выделения объектов, перемещения, вызова контекстного меню.

- Инструмент создания точки дороги — при нажатии приложение переходит в режим создания точки дороги. Находясь в данном режиме, при нажатии на рабочую область возникает точка дороги.

- Инструмент создания дороги — при нажатии приложение переходит в режим создания дороги. Находясь в данном режиме, при нажатии на рабочую область возникает точка дороги, связанная с уже выделенными точками. Конец связи обозначается зеленым цветом.

- Инструмент создания генераторов машин - при нажатии приложение переходит в режим создания генераторов машин. Находясь в данном режиме, при нажатии на рабочую область создается генератор машин.

- Инструмент создания светофоров- при нажатии приложение переходит в режим создания светофоров. Находясь в данном режиме, при нажатии на рабочую область создается светофор.

При нажатии правой кнопки мыши на рабочей области возникает контекстное меню с действиями:

- Удалить(Delete) — удаляет выделенные объекты.

- Соединить главную дорогу — при нажатии приложения переходит в режим соединения. В данном режиме, при нажатии на объект, выделенные объекты соединятся с ним сплошной линией, обозначающей главную дорогу.

- Соединить дорогу — при нажатии приложения переходит в режим соединения. В данном режиме, при нажатии на объект, выделенные объекты соединятся с ним пунктирной линией, обозначающей второстепенную дорогу.

- Отсоединить - при нажатии приложения переходит в режим отсоединения. В данном режиме, при нажатии на объект, выделенные объекты отсоединятся от него.

При выборе соединения дороги генератора автомобилей или светофора возникает фиолетовая пунктирная линия.

При выделении объекта светофора или генератора машин справа возникает окно их настройки (рис. 6.3, 6.4).

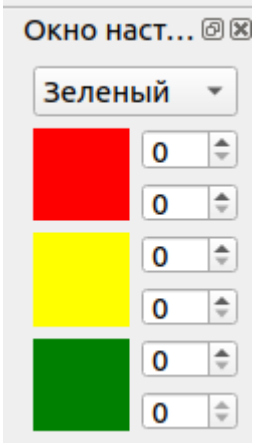

Рисунок 6.3 – Окно настройки светофора

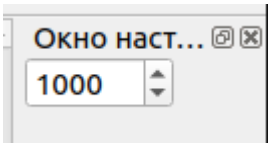

Рисунок 6.4 – Окно настройки генератора автомобилей

Окно настройки генератора автомобилей содержит ползунок изменения скорости генерации, измеряемое в миллисекундах.

Окно настройки светофора содержит выбор начального сигнала и ползунки периода сигналов светофора.

ЗАКЛЮЧЕНИЕ

В результате работы над данным курсовым проектом была разработана работоспособное приложение со своим набором функций, графическим интерфейсом и возможностью моделировать систему дорожного движения. Данный курсовой проект был разработан в соответствии с поставленными задачами, весь функционал был реализован в полном объеме.

В ходе разработки были углублены знания языка программирования C++ и в области объектно-ориентированного программирования, а также получен опыт работы с Фреймворком Qt, с программным интерфейсов OpenGL.

Работа была разделена на такие этапы, как анализ существующих аналогов, литературных источников, постановка требований к проектируемому программному продукту, системное и функциональное проектирование, конструирование программного продукта, разработка программных модулей и тестирование проекта. После последовательного выполнения вышеперечисленных этапов разработки было получено исправно работающее приложение.

В дальнейшем планируется усовершенствование текущего функционала приложения, путем улучшения графического интерфейса, добавления новых функций и модулей, улучшения поведения моделей, добавление других участников дорожного движения, добавления новых правил.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Shiffman, Daniel The Nature of Code. – D. Shiffman, 2012. – 498 p.

[2] Ginsburg, Dan. OpenGL ES 3.0 programming guide / Dan Ginsburg, Budirijanto Purnomo ; with earlier contributions from Dave Shreiner, Aaftab Munshi.—Second edition.- Crawfordfsville, Indiana, 2014. – 560 p.

[3] Шлее М. Qt 5.10. Профессиональное программирование на C++. - СПб.: БХВ-Петербург, 2018. - 1072 с.: ил. -(В подлиннике).

[4] Millington, Ian. Game physics engine development / Ian Millington. – San Francisco, 2007. – p. 458.

[5] Object-oriented analysis and design with applications / Grady Booch...[et al.]. — 3rd ed. - Westford, Massachusetts, 2007. – p.692.

[6] Язык программирования C++ : специальное издание / Бьерн Страуструп ; перевод с английского под редакцией Н. Н. Мартынова. - [3-е изд., перераб. и доп.]. - Москва : Бином, 2012. - 1136 с. ; 24 см.

[7] Документация Qt [Электронный ресурс] – The Qt Company Ltd. 2019. – Режим доступа: https://doc.qt.io/

[8] docs.GL OpenGL API Documentation [Электронный ресурс]. – Режим доступа: https://docs.gl/

[9] GLFW Documentation [Электронный ресурс]. – Режим доступа: https://www.glfw.org/documentation.html

ПРИЛОЖЕНИЕ А
(обязательное)

Схема структурная

ПРИЛОЖЕНИЕ Б
(обязательное)

Диаграмма классов

ПРИЛОЖЕНИЕ В
(обязательное)

Листинг кода

```
//Car.h

#ifndef _CAR_H_
#define _CAR_H_
#include <list>
#include <chrono>

#include "Vector.h"
#include "Road.h"
#include "TrafficLight.h"
#include "Particle.h"
#include "ForceGenerator.h"


class Car : public Particle2D
{
public:
    enum State{
        LIVE    = 1L << 0,
        TURN    = 1L << 1,
        FOLLOW  = 1L << 2,
        YIELD    = 1L << 3,
        STOP    = 1L << 4
    };
public:
    Car(const Vector& position);
    Car(float x, float y);
    ~Car() {}
    Car(const Car& );
    inline virtual void applyForce(const Vector &force) override
        { m_ForceAccumulator += force; }
    virtual void move(float time) override;
    bool followPath(const Road* , Vector*);
    inline virtual Vector getPosition() const override { return m_Position; }
    inline virtual Vector getVelocity() const override { return m_Velocity; }
    inline virtual Vector getAcceleration() const override
        { return m_Acceleration; }
    inline State getState() const { return m_State; }
    void setState(State state, bool value);
    double getMaxSpeed() const;
    inline double getMaxPossibleSpeed() const { return m_MaxSpeed; }
    bool view(const Car &car,
              double startAngle, double endAngle, double dist) const;
    bool view(const TrafficLight &light,
              double startAngle, double endAngle, double dist) const;
    inline virtual Vector getForceAccumulator() const override
        { return m_ForceAccumulator; }
private:
    inline virtual void clearAccumulator() override { m_ForceAccumulator *=
0;};
    virtual float getMass() const override { return 0; }
    virtual float getInverseMass() const override { return 0;}
    virtual bool hasFiniteMass() const override { return 0;}
    virtual void setDamping(float damping) override { }
    virtual float getDamping() const override { return 0;}
```

```
private:
    double m_MaxSpeed;
    State m_State;
};

inline constexpr Car::State operator&(Car::State __a, Car::State __b)
{ return Car::State(static_cast<int>(__a) & static_cast<int>(__b)); }
inline constexpr Car::State operator|(Car::State __a, Car::State __b)
{ return Car::State(static_cast<int>(__a) | static_cast<int>(__b)); }
inline constexpr Car::State operator^(Car::State __a, Car::State __b)
{ return Car::State(static_cast<int>(__a) ^ static_cast<int>(__b)); }
inline constexpr Car::State operator~(Car::State __a)
{ return Car::State(~static_cast<int>(__a)); }
inline const Car::State& operator|=(Car::State& __a, Car::State __b)
{ return __a = __a | __b; }
inline const Car::State& operator&=(Car::State& __a, Car::State __b)
{ return __a = __a & __b; }
inline const Car::State& operator^=(Car::State& __a, Car::State __b)
{ return __a = __a ^ __b; }

struct CarInformation
{
    Car car;
    SteerForceGenerator steerForceGenerator;
    BrakeForceGenerator brakeForceGenerator;
    unsigned int roadID;
};

class CarRegistry
{
public:
    CarRegistry()  : m_Cars() {}
    ~CarRegistry() {}

    void addCar(const Car &car, const unsigned int start_roadID);
    void update(const RoadRegistry &roads,
                const std::vector<const TrafficLight*>& lights,
                float time);
    std::vector<Car> getCars();
private:
    void interectLights(std::list<CarInformation>::iterator& it,
                        const std::vector<const TrafficLight*> &lights);
    void yieldInterect(std::list<CarInformation>::iterator& it,
                       const RoadRegistry &roads);
    void turn(std::list<CarInformation>::iterator& it,
              const RoadRegistry &roads);
    void interectFollow(std::list<CarInformation>::iterator& it,
                        const RoadRegistry &roads);
    void applySteerForce(Car &car,
                         SteerForceGenerator& steerForceGenerator,
                         const Road* road);
    void applyBrakeForce(Car &car, BrakeForceGenerator& brakeForceGenerator);
private:
    std::list<CarInformation> m_Cars;
};

class CarGenerator
{
public:
    CarGenerator();
    CarGenerator(const Vector &pos, const int rate = 1000);
    ~CarGenerator() {}
```

```
    inline void setPosition(const Vector &pos) { m_Position = pos; }
    inline const Vector& getPosition() const { return m_Position; }
    inline int getRate() const { return m_Rate; }
    inline void setRate(int rate) { this->m_Rate = rate; }
    inline void setStartRoadID(const unsigned int ID) { m_StartRoadID = ID; }
    inline unsigned int getStartRoadId() { return m_StartRoadID; }

    void update(CarRegistry &cars);
private:
    Vector m_Position;
    int m_Rate;
    unsigned int m_StartRoadID;
    std::chrono::time_point<std::chrono::steady_clock> time;
};

#endif

//Car.cpp

#include "../pch.h"

#include "Car.h"
#include "Vector.h"
#include "Road.h"

struct angle
{
    double start;
    double end;
};
static struct
{
    double pi = 3.14f;
    double infinity = 1000000000000000.0f;
    double epsilon = 0.00000000000000001f;
    angle lightViewAngle = {0.0f,  pi / 2.0f + 0.2};
    unsigned int lightViewDistance = 50;
    angle yieldViewAngle = {0.0f, pi / 3};
    unsigned int yieldViewDistance = 50;
    angle yieldSideViewAngle = {pi / 6, 4 * pi / 6};
    unsigned int yieldSideViewDistance = 75;
    angle followViewAngle = { 0.0f, pi / 2.0f + 0.2};
    unsigned int followViewDistance = 50;
    double stopSpeed = 1.0f;
    unsigned int turnViewDistance = 10;
    unsigned int standartCarGeneratorRate = 1000;
    unsigned int minMaxSpeed = 6000;
    unsigned int maxMaxSpeed = 10000;
} Constants;


/*
------------------------------------------------------------------------
||||||||||||||||||||||Car||||||||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------------------
*/

Car::Car(const Vector& pos)
    : Particle2D(pos)
    , m_MaxSpeed(Constants.minMaxSpeed
                + rand() % (Constants.maxMaxSpeed - Constants.minMaxSpeed))
    , m_State(LIVE)
```

```
{
    m_Position = pos;
}

Car::Car(const float x, const float y)
    : Particle2D(x, y)
    ,  m_MaxSpeed(Constants.minMaxSpeed
                + rand() % (Constants.maxMaxSpeed - Constants.minMaxSpeed))
    , m_State(LIVE)
{
    m_Position = Vector(x, y);
}

Car::Car(const Car& copy)
    : Particle2D(copy.m_Position)
    , m_MaxSpeed(copy.m_MaxSpeed)
    , m_State(copy.m_State)
{
    m_Acceleration = copy.m_Acceleration;
    m_Velocity = copy.m_Velocity;
    m_ForceAccumulator = copy.m_ForceAccumulator;
    m_InverseMass = copy.m_InverseMass;
    m_Damping = copy.m_Damping;
}

void Car::setState(State state, bool value)
{
    if (value)
        this->m_State |= state;
    else
        this->m_State &= ~state;
}

double Car::getMaxSpeed() const
{
    if (m_State & State::STOP || m_Velocity.getSquareMagnitude() < 1e-50)
        return Constants.stopSpeed;
    else if (m_State & State::FOLLOW)
        return m_Velocity.getMagnitude();

    return m_MaxSpeed;
}

bool Car::followPath(const Road* road, Vector* target)
{
    Vector predict = m_Velocity;
    predict.setMagnitude(10);
    Vector predictPos = predict + m_Position;
    Vector pos = getPosition();
    double worldRecord = Constants.infinity;
    Vector normal;
    double distance = 0;

    for (int i = 0; i < road->getRoadSize() - 1; ++i) {
        Vector a = road->getPoint(i);
        Vector b = road->getPoint(i + 1);

        Vector normalPoint = getNormalPoint(predictPos, a, b);
        if (!(normalPoint.getX() + road->getRadius() >
                std::min(a.getX(), b.getX()) &&
              normalPoint.getX() - road->getRadius() <
                std::max(a.getX(), b.getX()) &&
```

```
                normalPoint.getY() + road->getRadius() >
                  std::min(a.getY(), b.getY()) &&
                normalPoint.getY() - road->getRadius() <
                  std::max(a.getY(), b.getY()))) {
                  normalPoint = b;
            }
            if (getState() & State::TURN) {
                setState(State::TURN, false);
                if (road->isMain() == false) {
                    setState(State::YIELD, true);
                    setState(State::STOP, false);
                }
                else {
                    setState(State::YIELD | State::STOP, false);
                }
            }

            distance = predictPos.distance(normalPoint);
            if (distance < worldRecord) {
                normal = normalPoint;
                Vector dir = b - a;
                dir.setMagnitude(10);
                (*target) = normal + dir;
                worldRecord = distance;
                if (normal == road->getPoint(road->getRoadSize() - 1) &&
                    normal.distance(getPosition()) < Constants.turnViewDistance)
                    setState(State::TURN, true);
            }
        }
        return true;
    }

    void Car::move(float time)
    {
        m_Acceleration *= 0;
        m_Acceleration += m_ForceAccumulator;
        m_Velocity += (m_Acceleration * time);
        m_Velocity.limitMagnitude(getMaxSpeed());
        m_Position += (m_Velocity  * time);
        clearAccumulator();
    }

    bool Car::view(const Car &car,
                   double startAngle, double endAngle, double d) const
    {
        Vector dist = car.getPosition() - getPosition();
        Vector vel = getVelocity();
        double a = dist.getAngle(vel);
        if (dist.getMagnitude() < d && startAngle <= a  && a <= endAngle)
            return true;
        return false;
    }

    bool Car::view(const TrafficLight &light,
                   double startAngle, double endAngle, double d) const
    {
        Vector dist = light.getPosition() - getPosition();
        Vector vel = getVelocity();
        double a = dist.getAngle(vel);
        if (dist.getMagnitude() < d && startAngle <= a && a <= endAngle)
            return true;
        return false;
```

```
}

/*
------------------------------------------------------------------------
||||||||||||||||||||CarRegistry||||||||||||||||||||||||||||||||||||||||
------------------------------------------------------------------------
*/

void CarRegistry::addCar(const Car &car, const unsigned int start_roadID)
{
    m_Cars.push_back({car,
                      SteerForceGenerator(),
                      BrakeForceGenerator(),
                      start_roadID});
}

void CarRegistry::update(const RoadRegistry &roads,
                         const std::vector<const TrafficLight*> &lights,
                         float time)
{
    std::list<CarInformation>::iterator it = m_Cars.begin();
    while(it != m_Cars.end()) {

        interectLights(it, lights);
        yieldInterect(it, roads);
        turn(it, roads);
        interectFollow(it, roads);
        if ((it->car.getState() & Car::LIVE) == 0) {
            it = m_Cars.erase(it);
            continue;
        }
        applySteerForce(it->car,
                        it->steerForceGenerator,
                        roads.getRoad(it->roadID));
        applyBrakeForce(it->car, it->brakeForceGenerator);
        ++it;
    }
    it = m_Cars.begin();
    while(it != m_Cars.end()) {
        it->car.move(time);
        ++it;
    }

}

std::vector<Car> CarRegistry::getCars()
{
    std::vector<Car> ret;
    std::list<CarInformation>::iterator it = m_Cars.begin();
    while(it != m_Cars.end()) {
        ret.push_back(it->car);
        ++it;
    }
    return ret;
}

void CarRegistry::interectLights(std::list<CarInformation>::iterator& it,
                                 const std::vector<const TrafficLight*>
&lights)
{
    bool red_fl = false;
    for (unsigned int i = 0; i < lights.size(); ++i) {
```

```
        if (it->car.view(*lights[i],
                         Constants.lightViewAngle.start,
                         Constants.lightViewAngle.end,
                         Constants.lightViewDistance)) {
            if (lights[i]->getState() == TrafficLight::State::Red ||
               lights[i]->getState() == TrafficLight::State::Yellow) {
                it->car.setState(Car::State::STOP, true);
                Vector desired = it->car.getVelocity();
                desired.setMagnitude(it->car.getMaxSpeed());
                it->brakeForceGenerator
                    .init(desired,
                         (float)lights[i]->getPosition()
                            .distance(it->car.getPosition()));
                red_fl = true;
            }
            break;
        }
    }
    if (!red_fl)
        it->car.setState(Car::State::STOP, false);
}

void CarRegistry::yieldInterect(std::list<CarInformation>::iterator& it,
                                const RoadRegistry &roads)
{
     if ((it->car.getState() & Car::State::YIELD) /*&& (it->car.getState() &
Car::State::FOLLOW) == 0*/) {
            std::list<CarInformation>::iterator view_check = m_Cars.begin();
            bool stop_fl = false;
            while(view_check != m_Cars.end()) {
                if (/*it->roadID == view_check->roadID || */&it->car ==
&view_check->car) {
                    ++view_check;
                    continue;
                }
                bool fl = false;
                std::vector<unsigned int> conn =
                    roads.getRoadConnections(view_check->roadID);
                for (auto u : conn) {
                    if (it->roadID == u) {
                        fl = true;
                        break;
                    }
                }
                if (fl) {
                    ++view_check;
                    continue;
                }
                if (it->car.view(view_check->car,
                                 Constants.yieldViewAngle.start,
                                 Constants.yieldViewAngle.end,
                                 Constants.yieldViewDistance) ||
                    it->car.view(view_check->car,
                                 Constants.yieldSideViewAngle.start,
                                 Constants.yieldSideViewAngle.end,
                                 Constants.yieldSideViewDistance)) {

                    it->car.setState(Car::State::STOP, true);
                    Vector desired = it->car.getVelocity();
                    desired.setMagnitude(it->car.getMaxSpeed());
                    it->brakeForceGenerator
                        .init(desired,
```

```
                                (float)view_check->car.getPosition()
                                   .distance(it->car.getPosition()));
                        stop_fl = true;
                        break;
                    }
                    ++view_check;
                }
                if (!stop_fl)
                    it->car.setState(Car::State::STOP, false);
            }
}

void CarRegistry::turn(std::list<CarInformation>::iterator& it,
                       const RoadRegistry &roads)
{
    if (it->car.getState() & Car::TURN) {
        std::vector<unsigned int> conn = roads.getRoadConnections(it-
>roadID);
        if (conn.size() != 0) {
            double prob = (double)(std::rand() % 100) / 100;
            int i = 0;
            while (prob > 1.0f / conn.size()) {
                prob -= 1.0f / conn.size();
                ++i;
            }
            it->roadID = conn[i];
        }
        else {
            it->car.setState(Car::LIVE, false);
        }
    }
}

void CarRegistry::interectFollow(std::list<CarInformation>::iterator& it,
                                 const RoadRegistry &roads)
{
    if ((it->car.getState() & Car::State::TURN) == 0 &&
        (it->car.getState() & Car::State::STOP) == 0) {
        std::list<CarInformation>::iterator view_check = m_Cars.begin();
        bool follow_fl = false;
        while(view_check != m_Cars.end()) {
            if (&it->car == &view_check->car) {
                ++view_check;
                continue;
            }
            bool fl = false;
            std::vector<unsigned int> conn =
                roads.getRoadConnections(it->roadID);
            for (auto u : conn) {
                if (view_check->roadID == u) {
                    fl = true;
                    break;
                }
            }
            if (it->roadID == view_check->roadID)
                fl = true;
            if (!fl) {
                ++view_check;
                continue;
            }

            if (it->car.view(view_check->car,
```

```
                Constants.followViewAngle.start,
                Constants.followViewAngle.end,
                Constants.followViewDistance)) {

                it->car.setState(Car::State::FOLLOW, true);
                Vector desired = it->car.getVelocity();
                desired.setMagnitude(view_check->car.getMaxSpeed());
                it->brakeForceGenerator
                    .init(desired,
                          (float)view_check->car
                            .getPosition().distance(it->car.getPosition()));
                follow_fl = true;
                break;
            }
            ++view_check;
        }
        if (!follow_fl)
            it->car.setState(Car::State::FOLLOW, false);
    }
}

void CarRegistry::applySteerForce(Car &car,
                                  SteerForceGenerator& steerForceGenerator,
                                   const Road* road)
{
    Vector target;
    if (car.followPath(road, &target)) {
        steerForceGenerator.setTarget(target);
        steerForceGenerator.updateForce(&car, 0);
    }
}
void CarRegistry::applyBrakeForce(Car &car,
                                  BrakeForceGenerator& brakeForceGenerator)
{
   if ((car.getState() & Car::State::FOLLOW) ||
       (car.getState() & Car::State::STOP)) {
        brakeForceGenerator.updateForce(&car, 0);
    }

}

/*
--------------------------------------------------------------------------
||||||||||||||||||||CarGenerator||||||||||||||||||||||||||||||||||
--------------------------------------------------------------------------
*/

CarGenerator::CarGenerator()
    :m_Position(), m_Rate(Constants.standartCarGeneratorRate)
{
    time = std::chrono::steady_clock::now();
}

CarGenerator::CarGenerator(const Vector &pos, const int rate)
    :m_Position(pos), m_Rate(rate)
{
    time = std::chrono::steady_clock::now();
}

void CarGenerator::update(CarRegistry& carRegistry)
{
    auto end = std::chrono::steady_clock::now();
```

```
    auto dif =
        std::chrono::duration_cast<std::chrono::milliseconds>(end - time);
    bool add_fl = true;
    if (dif >= std::chrono::milliseconds(m_Rate)) {
        Car car(m_Position);
        std::vector<Car> cars = carRegistry.getCars();
        for (auto u : cars) {
            if (u.getPosition().distance(m_Position) <
                    Constants.followViewDistance) {
                add_fl = false;
                break;
            }
        }
        if (add_fl)
            carRegistry.addCar(car, m_StartRoadID);

        time = std::chrono::steady_clock::now();
    }
}

//ForceGenerator.h

#ifndef _FORCEGENERATOR_H_
#define _FORCEGENERATOR_H_
#include "Particle.h"

class ForceGenerator
{
public:
    virtual void updateForce(Particle2D* Particle2D, const float duration) {}
    virtual ~ForceGenerator() {}
};

class SteerForceGenerator : public ForceGenerator
{
public:
    SteerForceGenerator();
    SteerForceGenerator(const Vector &target);
    ~SteerForceGenerator() {}
    inline void setTarget(const Vector &target) { m_Target = target; }
    inline Vector getTarget() const { return m_Target; }
    virtual void updateForce(Particle2D* particle, const float time)
override;
private:
    Vector m_Target;
};

class BrakeForceGenerator : public ForceGenerator
{
public:
    BrakeForceGenerator();
    BrakeForceGenerator(const Vector& desired,  float distance);
    ~BrakeForceGenerator() {}
    void init(const Vector& desired,  float distance);
    inline void setDesiredVelocity(const Vector& desired) { m_Desired =
desired; }
    inline Vector getDesiredVelocity() const { return m_Desired; }
    inline void setDistance( float distance) { m_Distance = distance; }
    inline float getDistance() const { return m_Distance; }
    virtual void updateForce(Particle2D* Particle2D,  float duration)
override;
private:
```

```
    Vector m_Desired;
    float m_Distance;
};

#endif

//ForceGenerator.cpp

#include "ForceGenerator.h"
#include "Car.h"
#include "./../pch.h"

/*
-----------------------------------------------------------------------
||||||||||||||||||||||SteerForceGenerator||||||||||||||||||||||||||||
-----------------------------------------------------------------------
*/

SteerForceGenerator::SteerForceGenerator()
    : m_Target()
{
}

SteerForceGenerator::SteerForceGenerator(const Vector &target)
    : m_Target(target)
{
}

void SteerForceGenerator::updateForce(Particle2D* particle,  float time)
{
    Car* car = dynamic_cast<Car*>(particle);
    Vector desired = m_Target - car->getPosition();
    if (desired.getMagnitude() == 0)
        return;
    double lim = car->getMaxSpeed();
    desired.setMagnitude(lim);
    Vector steer = desired - car->getVelocity()/* - car->getAcceleration()*/;
    steer.limitMagnitude(lim);
    car->applyForce(steer);
}

/*
-----------------------------------------------------------------------
||||||||||||||||||||BrakeForceGenerator||||||||||||||||||||||||||||
-----------------------------------------------------------------------
*/

BrakeForceGenerator::BrakeForceGenerator()
    : m_Desired(), m_Distance(0)
{
}
BrakeForceGenerator::BrakeForceGenerator(const Vector& desired,
                                         float distance)
    : m_Desired(desired), m_Distance(distance)
{
}

void BrakeForceGenerator::init(const Vector& desired, float distance)
{
    m_Desired = desired;
    m_Distance = distance;
}
```

```
void BrakeForceGenerator::updateForce(Particle2D* particle, float duration)
{
    Car* car = dynamic_cast<Car*>(particle);
    Vector desiredBrake = car->getVelocity();
    Vector vel = car->getVelocity();
    Vector desVel = car->getVelocity();
    desVel.setMagnitude(m_Desired.getMagnitude());
    Vector brake(0.0f, 0.0f);
    if (m_Desired.getSquareMagnitude() <= vel.getSquareMagnitude()) {
        brake = desVel - vel;
    }
    car->applyForce(brake);
}

//MDLParser.h

#ifndef _MDLPARSER_H_
#define _MDLPARSER_H_
#include "Car.h"
#include "Road.h"
#include "TrafficLight.h"
#include <memory>
#include <vector>
#include <string>

struct ParseInformation
{
    RoadRegistry* roadRegistry;
    std::vector<CarGenerator*> CarGenerators;
    std::vector<TrafficLight*> TrafficLights;
};

struct RoadGraph
{
    std::vector<std::vector<int>> connections;
    std::vector<std::vector<int>> back_connections;
};


class MDLParser
{
public:
    static const ParseInformation parseFile(const std::string& filePath);
private:
    static RoadRegistry* parseRoads(std::ifstream& file);
    static std::vector<CarGenerator*> parseCarGenerators(std::ifstream&
file);
    static std::vector<Vector> parsePoints(std::ifstream& file);
    static RoadGraph parseConnections(std::ifstream& file);
    static std::vector<TrafficLight*>
    parseTrafficLights(std::ifstream& file,
                       std::vector<int> &lightsConnections);
    static void constructRoads(RoadGraph& graph,
                               int x,
                               RoadRegistry* roadRegistry,
                               std::vector<Vector>& points,
                               std::vector<bool> &vis);
    static void connectRoads(RoadGraph& graph, RoadRegistry* roadRegistry);
};

#endif
```

```
//MDLParser.cpp

#include "../pch.h"
#include "MDLParser.h"
#include <cstdio>

static std::unordered_multimap<int, std::pair<int, int>> m;
static std::set<std::pair<int, int>> roadType;


const ParseInformation MDLParser::parseFile(const std::string& filePath)
{
    static std::string keyWords[] = {
        "#CarGenerators",
        "#Roads",
        "#Connections",
        "#Points",
        "#TrafficLights"
    };
    ParseInformation info;
    std::ifstream file(filePath);
    std::string line;
    std::vector<Vector> points;
    std::vector<bool> vis;
    RoadGraph graph;
    std::vector<int> lightsConnections;
    while(getline(file, line)) {
        if (line == "#CarGenerators")
            info.CarGenerators = parseCarGenerators(file);
        else if (line == "#Points")
            points = parsePoints(file);
        else if (line == "#Connections")
            graph = parseConnections(file);
        else if (line == "#TrafficLights")
            info.TrafficLights = parseTrafficLights(file, lightsConnections);
    }
    for (int i = 0; i < info.TrafficLights.size(); ++i) {
        info.TrafficLights[i]->setPosition(points[lightsConnections[i]]);
    }
    vis.resize(points.size(), false);
    info.roadRegistry = new RoadRegistry();
    for (int i = 0; i < info.CarGenerators.size(); ++i) {
        info.CarGenerators[i]->
            setPosition(points[info.CarGenerators[i]->getStartRoadId()]);
    }
    for (int i = 0; i < points.size(); ++i) {
        if (!vis[i])
            constructRoads(graph, i, info.roadRegistry, points, vis);
    }
    connectRoads(graph, info.roadRegistry);
    file.close();
    return info;
}

RoadRegistry* MDLParser::parseRoads(std::ifstream& file)
{
    std::string line;
    RoadRegistry* registry = new RoadRegistry();
    while (getline(file, line)) {
        if (line == "}" || line == "#Connections")
            break;
        if (line == "" || line == "{")
```

```
            continue;
        int ID, count;
        sscanf(line.c_str(), "%d%d", &ID, &count);
        std::unique_ptr<Road> road = std::make_unique<Road>(ID);
        for (unsigned int i = 0; i < count; ++i) {
            getline(file, line);
            float x, y;
            sscanf(line.c_str(), "%f%f", &x, &y);
            road->addPoint(x, y);
        }
        registry->addRoad(std::move(road));
    }
    while (getline(file, line)) {
        if (line == "}")
            break;
        if (line == "" || line == "{")
            continue;
        int IDfrom, IDto;
        sscanf(line.c_str(), "%d%d", &IDfrom, &IDto);
        registry->connectRoads(IDfrom, IDto);
    }
    return registry;
}

std::vector<CarGenerator*> MDLParser::parseCarGenerators(std::ifstream& file)
{
    std::string line;
    std::vector<CarGenerator*> generators;
    while (getline(file, line)) {
        if (line == "}")
            break;
        if (line == "" || line == "{")
            continue;
        int startID;
        float x, y;
        unsigned int rate;
        sscanf(line.c_str(), "%d%f%f%d", &startID, &x, &y, &rate);
        CarGenerator* generator = new CarGenerator(Vector(x, y), rate);
        generator->setStartRoadID(startID);
        generators.push_back(generator);
    }
    return generators;
}

std::vector<Vector> MDLParser::parsePoints(std::ifstream& file)
{
    std::string line;
    std::vector<Vector> points;
    while (getline(file, line)) {
        if (line == "}")
            break;
        if (line == "" || line == "{")
            continue;
        int ID;
        float x, y;
        sscanf(line.c_str(), "%d%f%f", &ID, &x, &y);
        points.push_back(Vector(x, y));
    }
    return points;
}

std::vector<TrafficLight*>
```

```
MDLParser::parseTrafficLights(std::ifstream& file,
                              std::vector<int>& lightsConnections)
{
    std::string line;
    std::vector<TrafficLight*> lights;
    while (getline(file, line)) {
        if (line == "}")
            break;
        if (line == "" || line == "{")
            continue;
        int ConnectionID;
        float x, y;
        int green, yellow, red;
        sscanf(line.c_str(), "%d%f%f%d%d%d",
               &ConnectionID, &x, &y, &green, &yellow, &red);
        TrafficLight::State state;
        if (green <= yellow) {
            if (green <= red)
                state = TrafficLight::State::Green;
            else
                state = TrafficLight::State::Red;
        }
        else {
            state = TrafficLight::State::Yellow;
        }
        TrafficLight* light =
            new TrafficLight(Vector(x, y), state,
            green * 1000, yellow * 1000, red * 1000);
        lights.push_back(light);
        lightsConnections.push_back(ConnectionID);
    }
    return lights;
}


RoadGraph MDLParser::parseConnections(std::ifstream& file)
{
    std::string line;
    RoadGraph graph;
    while (getline(file, line)) {
        if (line == "}")
            break;
        if (line == "" || line == "{")
            continue;
        int IDfrom, IDto, type;
        sscanf(line.c_str(), "%d%d%d", &IDfrom, &IDto, &type);
        if (!type)
            roadType.insert({IDfrom, IDto});
        if (IDfrom >= graph.connections.size()) {
            graph.connections.resize(IDfrom + 1);
        }
        graph.connections[IDfrom].push_back(IDto);
        if (IDto >= graph.back_connections.size()) {
            graph.back_connections.resize(IDto + 1);
        }
        graph.back_connections[IDto].push_back(IDfrom);
    }
    if (graph.connections.size() != graph.back_connections.size()) {
        graph.connections.resize(std::max(graph.connections.size(),
                                          graph.back_connections.size()));
        graph.back_connections.resize(std::max(graph.connections.size(),
```

```
graph.back_connections.size()));
    }
    return graph;
}

void MDLParser::constructRoads(RoadGraph& graph,
                               int x, RoadRegistry* roadRegistry,
                               std::vector<Vector>& points,
                               std::vector<bool>& vis)
{
    static unsigned int lastID = points.size();
    while (graph.back_connections[x].size() == 1 &&
           graph.connections[x].size() <= 1) {
        x = graph.back_connections[x][0];
    }
    vis[x] = true;
    bool road_value = false;
    for (unsigned int i = 0; i < graph.connections[x].size(); ++i) {
        std::unique_ptr<Road> road=
            std::make_unique<Road>(i == 0 ? x : lastID++);
        road_value = false;
        road->addPoint(points[x].getX(), points[x].getY());
        std::queue<int> queue;
        int it = graph.connections[x][i];
        queue.push(it);
        while (graph.back_connections[it].size() == 1 &&
               graph.connections[it].size() == 1) {
            vis[it] = true;
            it = graph.connections[it][0];
            queue.push(it);
        }
         if (graph.back_connections[it].size() == 1 &&
             graph.connections[it].size() == 0)
            vis[it] = true;
          if (i != 0)
            m.insert({x, {lastID - 1, it}});
        else
            m.insert({x, {x, it}});
        std::pair<int, int> key;
        key.first = x;
        while (!queue.empty()) {
            road->addPoint(points[queue.front()].getX(),
                           points[queue.front()].getY());
            key.second = queue.front();
            if (!roadType.count(key))
                road_value = true;
            key.first = queue.front();
            queue.pop();
        }
        road->setMain(road_value);
        roadRegistry->addRoad(std::move(road));
    }

}

void MDLParser::connectRoads(RoadGraph& graph, RoadRegistry* roadRegistry)
{
    std::pair<std::unordered_multimap<int, std::pair<int, int>>::iterator,
        std::unordered_multimap<int, std::pair<int, int>>::iterator> start;
    std::pair<std::unordered_multimap<int, std::pair<int, int>>::iterator,
        std::unordered_multimap<int, std::pair<int, int>>::iterator> end;
```

```
    std::unordered_multimap<int, std::pair<int, int>>::iterator it;
    std::unordered_multimap<int, std::pair<int, int>>::iterator jt;
    start = m.equal_range(i);
    for (it = start.first; it != start.second; ++it) {
        end = m.equal_range(it->second.second);
        for (jt = end.first; jt != end.second; ++jt) {
            roadRegistry->connectRoads(it->second.first, jt->second.first);
        }
    }
}

//Particle.h

#ifndef _PARTICLE_H_
#define _PARTICLE_H_
#include "Vector.h"

class Particle2D
{
public:
    Particle2D(const Vector& position, float mass = 1.0f,
               float damping = 0.98f);
    Particle2D(const float x, const float y, float mass = 1.0f,
               float damping = 0.98f);
    virtual ~Particle2D() { }
    virtual void move(float duration) = 0;
    virtual float getMass() const = 0;
    virtual float getInverseMass() const = 0;
    virtual bool hasFiniteMass() const = 0;
    virtual void setDamping(float damping) = 0;
    virtual float getDamping() const = 0;
    virtual Vector getPosition() const = 0;
    virtual Vector getVelocity() const = 0;
    virtual Vector getAcceleration() const = 0;
    virtual Vector getForceAccumulator() const = 0;
    virtual void clearAccumulator() = 0;
    virtual void applyForce(const Vector& force) = 0;
protected:
    Vector m_Position;
    Vector m_Velocity;
    Vector m_Acceleration;
    Vector m_ForceAccumulator;
    float m_InverseMass;
    float m_Damping;
};

#endif

//Particle.cpp

#include "Particle.h"


Particle2D::Particle2D(const Vector& pos, float mass, float damping)
    : m_Position(pos), m_Velocity(), m_Acceleration(), m_ForceAccumulator(),
    m_InverseMass(1 / mass), m_Damping(damping)
{

}

Particle2D::Particle2D(const float x, const float y, float mass, float
damping)
```

```
    : m_Position(x, y), m_Velocity(), m_Acceleration(), m_ForceAccumulator(),
    m_InverseMass(1 / mass), m_Damping(damping)
{

}
```

```
      //Road.h

#ifndef _ROAD_H_
#define _ROAD_H_
#include <vector>
#include <memory>
#include <unordered_map>
#include "Vector.h"

class Road
{
public:
    Road(unsigned int id, bool main = true)
        : m_Points(), m_Radius(5), m_ID(id), m_Main(false) { }
    ~Road() { }
    inline void addPoint(double x, double y)
        { m_Points.push_back(Vector(x, y)); }
    inline int getRoadSize() const { return m_Points.size(); }
    inline Vector getPoint(int i) const { return m_Points[i]; }
    inline double getRadius() const { return m_Radius; }
    inline unsigned int getID() const { return m_ID; }
    inline bool isMain() const { return m_Main; }
    inline void setMain(bool value) { m_Main = value; }
private:
    std::vector <Vector> m_Points;
    double m_Radius;
    unsigned int m_ID;
    bool m_Main;
};

class RoadRegistry
{
public:
    RoadRegistry() : connections(), roads(), m_hash() { }
    ~RoadRegistry() { }

    const Road* getRoad(const unsigned int roadID) const;
    const std::vector<const Road*> getRoads() const;
    void addRoad(std::unique_ptr<Road> road);
    void connectRoads(const unsigned int ID_from, const unsigned int ID_to);
    const std::vector<unsigned int>&
    getRoadConnections(const unsigned int roadID) const;
    unsigned int getRoadsCount() const;
private:
    std::vector<std::vector<unsigned int>> connections;
    std::vector<std::unique_ptr<Road>> roads;
    mutable std::unordered_map<unsigned int, unsigned int> m_hash;
};

#endif

//Road.cpp

#include "../pch.h"
#include "Road.h"
#include "Vector.h"
```

```
/*
--------------------------------------------------------------------------
||||||||||||||||||||||Road||||||||||||||||||||||||||||||||||||||||||||||||
--------------------------------------------------------------------------
*/

/*
--------------------------------------------------------------------------
||||||||||||||||||||||RoadRegistry|||||||||||||||||||||||||||||||||||||||
--------------------------------------------------------------------------
*/

const Road* RoadRegistry::getRoad(const unsigned int roadID) const
{
    if (m_hash.find(roadID) != m_hash.end())
        return roads[m_hash[roadID]].get();
    return nullptr;
}

const std::vector<const Road*> RoadRegistry::getRoads() const
{
    std::vector<const Road*> ret(roads.size());
    for (unsigned int i = 0; i < roads.size(); ++i) {
        ret[i] =roads[i].get();
    }
    return ret;
}
void RoadRegistry::addRoad(std::unique_ptr<Road> road)
{
    m_hash[road->getID()] = connections.size();
    roads.push_back(std::move(road));
    connections.push_back(std::vector<unsigned int>());
}
void RoadRegistry::connectRoads(const unsigned int ID_from,
                                const unsigned int ID_to)
{
    if (m_hash.find(ID_from) != m_hash.end() &&
        m_hash.find(ID_to) != m_hash.end())
    connections[m_hash[ID_from]].push_back(ID_to);
}

const std::vector<unsigned int>&
RoadRegistry::getRoadConnections(const unsigned int roadID) const
{
    if (m_hash.find(roadID) != m_hash.end())
        return connections[m_hash[roadID]];
}

//TrafficLight.h

#ifndef _TRAFFICLIGHT_H_
#define _TRAFFICLIGHT_H_
#include "Vector.h"
#include <chrono>

class TrafficLight
{
public:
    enum class State
    {
        Green   = 0,
```

```
        Yellow  = 1,
        Red     = 2
    };
public:
    TrafficLight(Vector position, State startState,
                 int greenTime, int yellowTime, int redTime);
    ~TrafficLight() { }
    void update();
    inline State getState() const { return m_State; }
    inline Vector getPosition() const { return m_Position; }
    inline void setPosition(Vector pos) { m_Position = pos; }
private:
    Vector m_Position;
    int m_Period[4];
    State m_State;
    State m_StartState;
    std::chrono::time_point<std::chrono::steady_clock> m_CurrentTime;
};

#endif

//TrafficLight.cpp

#include "../pch.h"
#include "TrafficLight.h"

TrafficLight::TrafficLight(Vector position,
                           State startState,
                           int greenTime, int yellowTime, int redTime)
    : m_Position(position)
    , m_CurrentTime(std::chrono::milliseconds(0))
    , m_State(startState)
    , m_StartState(startState)
{
    m_Period[(int)m_StartState] = 0;
    int i = 0;
    if (i == (int)m_StartState)
        ++i;
    m_Period[i++] = greenTime;
    if (i == (int)m_StartState)
        ++i;
    m_Period[i++] = yellowTime;
    if (i == (int)m_StartState)
        ++i;
    m_Period[i++] = redTime;

}

void TrafficLight::update()
{
    using namespace std::chrono;
    auto end = steady_clock::now();
    auto dif = duration_cast<milliseconds>(end - m_CurrentTime);
    if (dif >= milliseconds(m_Period[((int)m_StartState + 3) % 4])) {
        m_CurrentTime = steady_clock::now();
        dif = milliseconds(0);
    }
    int i = (int)m_StartState;
    int cnt = 4;
    while (cnt--) {
        if (dif >= milliseconds(m_Period[i % 4]) &&
            dif <= milliseconds(m_Period[(i + 1) % 4]))
```

```
            m_State = (State)(i % 3);
        ++i;
    }
}

//Vector.h

#ifndef _VECTOR_H_
#define _VECTOR_H_
#include <cmath>
class Vector
{
private:
    double x;
    double y;
public:
    Vector(double x = .0, double y = .0) : x(x), y(y){ }
    ~Vector() {}
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    Vector& operator+=(const Vector&);
    Vector& operator+=(const double);
    inline Vector operator+(const Vector& v)
        { return Vector(getX() + v.getX(), getY() + v.getY()); }
    inline friend Vector operator+(const double value, const Vector& right)
      { return Vector(value + right.getX(), value + right.getY()); }
    inline Vector operator+(const double value)
        { return Vector(getX() + value, getY() + value); }
    Vector& operator-=(const Vector&);
    Vector& operator-=(const double);
    inline Vector operator-(const Vector& v)
        { return Vector(getX() - v.getX(), getY() - v.getY()); }
    inline Vector operator-(const double value)
        { return Vector(getX() - value, getY() - value); }
    Vector& operator/=(const double);
    Vector& operator/(const double);
    Vector& operator*=(const double);
    Vector& operator*(const double);
    inline friend Vector operator*(const double value, const Vector& right)
        { return Vector(value * right.getX(), value * right.getY()); }
    inline friend double operator*(const Vector& left, const Vector& right)
        { return left.getX() * right.getX() + left.getY() * right.getY(); }
    inline friend double scalarDotProduct(const Vector& left, const Vector&
right)
        { return left.getX() * right.getY() - left.getY() * right.getX(); }
    inline friend bool operator==(const Vector& left, const Vector& right)
        { return (left.getX() == right.getX() && left.getY() ==
right.getY()); }

    inline double getMagnitude() const
        { double s = getSquareMagnitude(); if (s < 0) s = 0; return sqrt(s);
}
    inline double getSquareMagnitude() const
        { return getX() * getX() + getY() * getY(); }
    inline void normalize() { (*this) /= getMagnitude(); }
    inline void setMagnitude(double magnitude)
        { normalize(); (*this) *= magnitude; }
    void limitMagnitude(double limit);
    inline double heading2D() const { return atan2(getY(), getX()); }
    void rotate(double angle);
    inline double getX() const { return x; }
    inline double getY() const { return y; }
```

```
    inline void setX(double value) { x = value; }
    inline void setY(double value) { y = value; }
    double distance(const Vector &);
    double getAngle(const Vector &);
    friend Vector getNormalPoint(Vector& point, Vector& a, Vector& b);
};


#endif

//Vector.cpp

#include "../pch.h"

#include "Vector.h"

Vector::Vector(const Vector &obj)
{
    x = obj.getX();
    y = obj.getY();
}

Vector& Vector::operator=(const Vector &v)
{
    if (this == &v)
        return *this;
    setX(v.getX());
    setY(v.getY());
    return *this;
}

Vector& Vector::operator+=(const Vector& v)
{
    setX(getX() + v.getX());
    setY(getY() + v.getY());
    return *this;
}

Vector& Vector::operator+=(const double value)
{
    setX(getX() + value);
    setY(getY() + value);
    return *this;
}

Vector& Vector::operator-=(const Vector& v)
{
    setX(getX() - v.getX());
    setY(getY() - v.getY());
    return *this;
}

Vector& Vector::operator-=(const double value)
{
    setX(getX() - value);
    setY(getY() - value);
    return *this;
}

Vector& Vector::operator/(const double value)
{
    if (value == 0.0)
```

```
        return *this;
    setX(getX() / value);
    setY(getY() / value);
    return *this;
}

Vector& Vector::operator/=(const double value)
{
    if (value == 0.0)
        return *this;
    setX(getX() / value);
    setY(getY() / value);
    return *this;
}

Vector& Vector::operator*=(const double value)
{
    setX(getX() * value);
    setY(getY() * value);
    return *this;
}

Vector& Vector::operator*(const double value)
{
    setX(getX() * value);
    setY(getY() * value);
    return *this;
}

void Vector::rotate(double angle)
{
    double magnitude = getMagnitude();
    double start_angle = heading2D();
    setX(cos(angle + start_angle));
    setY(sin(angle + start_angle));
    (*this) *= magnitude;
}

double Vector::distance(const Vector& point)
{
    Vector temp(point.getX() - getX(), point.getY() - getY());
    return temp.getMagnitude();
}

void Vector::limitMagnitude(double limit)
{
    if (getMagnitude() > limit)
        setMagnitude(limit);
}

double Vector::getAngle(const Vector& v)
{
    double m = getMagnitude();
    double mv = v.getMagnitude();
    double angle;
    if (mv == 0 || m == 0)
        angle = 0.0f;
    else
         angle = (*this) * v /(mv * m);
    if (angle > 1.0f)
        angle = 1.0f;
    if (angle < -1.0f)
```

```
        angle ==-1.0f;
    return acos(angle);
}

Vector getNormalPoint(Vector&p, Vector&a, Vector&b)
{
    Vector ap = p - a;
    Vector ab = b - a;
    ab.normalize();
    ab *= ab * ap;
    return ab + a;
}
```

ПРИЛОЖЕНИЕ Г
(обязательное)

Ведомость документо