

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра информационных систем

ОТЧЕТ
по Учебной практике
Тема: Поиск маршрута на карте

Студент гр. 2372

Алексеев Г.

Студент гр. 2372

Соколовский В.Д.

Студент гр. 2372

Гечис В.Р.

Руководитель

Титов Г.С.

Санкт-Петербург

2024

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент Алексеев Г.

Студент Соколовский В.Д.

Студент Гечис В.Р.

Группа 2372

Тема практики: Поиск маршрута на карте

Задание на практику:

Требуется разработать приложение построения карты и маршрутов.

а) Функциональные требования:

- В приложении должна быть предусмотрена возможность загрузки готовых, а также создания и сохранения в файл новых карт препятствий, препятствия должны наноситься на карту с помощью мыши.
- Координаты для построения маршрута задаются с помощью отметки мышью стартовой и финишной точек маршрута.
- Маршрут должен представлять собой ломаную линию, соединяющую стартовую и финишную точки. После построения, маршрут должен быть отображён на карте препятствий.
- В приложении должна быть предусмотрена возможность сохранения маршрута в файл.
- Для каждого препятствия должен быть реализован индекс непроходимости (от 0 до 100%), индекс должен быть отображён у каждого препятствия на карте.

б) Нефункциональные требования:

- Приложение должно быть реализовано на языке C++/Qt, в среде разработки QtCreator
- Код должен быть откомментирован.
- Формат сохранения карт и маршрутов – XML

Сроки прохождения практики: 4.03.2024 – 31.05.2024

Дата сдачи отчета: 24.05.2024

Дата защиты отчета: 31.05.2023

Студент гр. 2372

Алексеев Г.

Студент гр. 2372

Соколовский В.Д.

Студент гр. 2372

Гечис В.Р.

Руководитель

Титов Г.С.

АННОТАЦИЯ

Содержание учебной практики заключается в создании приложения для построения маршрутов на карте на языке C++ средствами Фреймворка Qt. Были написаны основные классы и алгоритмы, реализующие необходимые требования учебной практики. Был создан интерфейс, реализован установщик приложения.

SUMMARY

The content of the training practice is to create an application for plotting routes on map in C ++ using the QT Framework. The main classes and algorithms that implement the necessary requirements of educational practice were written. An interface was created, an application installer was implemented.

СОДЕРЖАНИЕ

	Введение	5
1.	Состав группы	6
2.	О приложении	7
2.1.	Описание приложения	7
2.2.	Руководство оператора	7
3.	Архитектура приложения	10
3.1.	Спецификация классов	10
3.2.	Обоснование выбора алгоритмов	14
3.3.	Дополнительные диаграммы	19
3.4.	Форматы входных и выходных файлов	21
4.	Тестирование	23
5.	Демонстрация	24
	Заключение	30
	Список использованных источников	31

1. СОСТАВ ГРУППЫ

ФИО	Раздел приложения	Описание
Алексеев Г.	Алгоритма поиска маршрута, модуль маршрута	Программная реализация алгоритма поиска маршрута, основного модуля маршрута
Соколовский В.Д.	Пользовательский интерфейс, классы	UI, переходы между режимами, вспомогательные модули
Гечис В.Р.	Отчёт, тестирование	Написание отчёта, создание диаграмм, спецификация классов, системное тестирование

Таблица 1 – Состав группы

2. О ПРИЛОЖЕНИИ

2.1. Описание приложения

Данное приложение (Route-Mar) было создано в рамках работы по учебной практике для построения маршрутов на карте препятствий.

В этой программе вы можете:

- Вручную создать карту, нанося препятствия с индексом непроходимости на карту.
- Загружать готовые карты препятствий.
- Строить маршруты путём указания начальной и конечной точек маршрута на карте препятствий.
- Получать графическое изображение оптимального маршрута на карте препятствий.
- Сохранять полученный маршрут в файл с информацией о скорости и времени маршрута.

2.2. Руководство оператора приложения

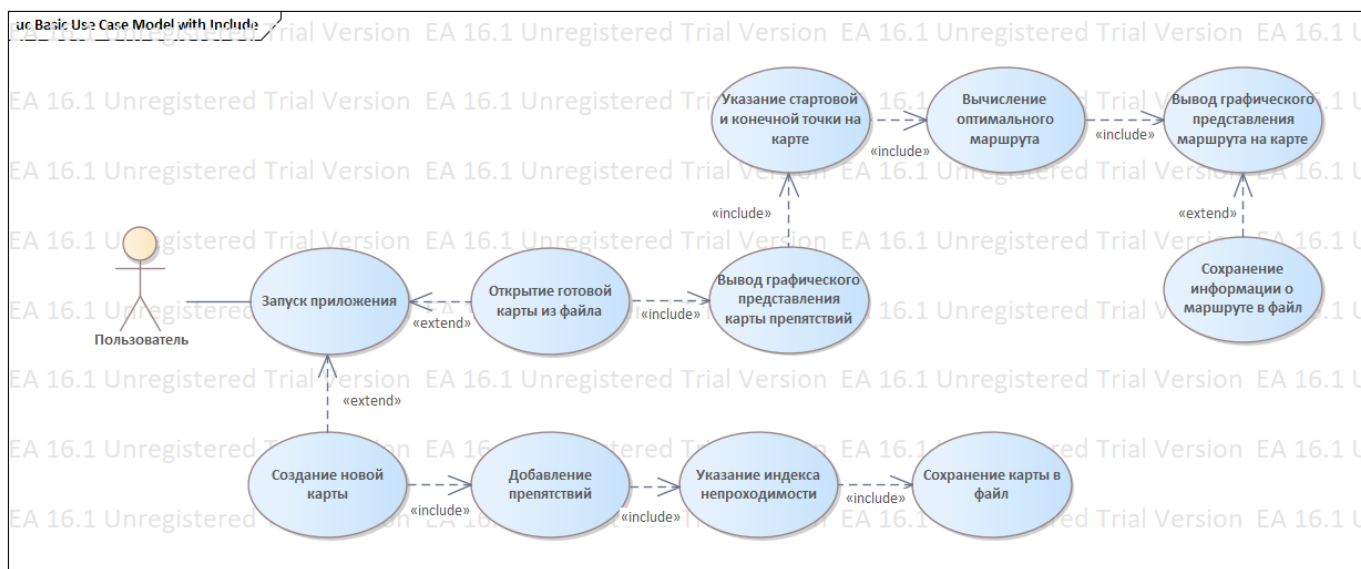


Рисунок 1 – Диаграмма вариантов использования (UseCase)

- В таблицах 2 и 4 описаны варианты использования системы с кратким описанием и условиями.
- В таблицах 3 и 5 описаны последовательности действий, приводящие к успешному выполнению соответствующего варианта использования.

Вариант использования	Проложение оптимального маршрута на карте
Актор	<ul style="list-style-type: none"> • Пользователь
Предусловия	<ul style="list-style-type: none"> • Пользователь должен запустить приложение и открыть карту препятствий
Краткое описание	<ul style="list-style-type: none"> • Пользователь устанавливает стартовую и конечную точку с помощью адреса или точки на карте, сервис предоставляет возможность проложения оптимального маршрута.
Постусловия	<ul style="list-style-type: none"> • Маршрут представлен пользователю на экране

Таблица 2 – Проложение автоматического маршрута на карте

Ход действий для проложения автоматического маршрута на карте

Действия актора	Отклик системы
1. Пользователь открывает карту	<ul style="list-style-type: none"> • Система представляет графическое изображение карты препятствий на экране
2. Пользователь ставит начальную и конечную точку	<ul style="list-style-type: none"> • Система отображает концевые точки маршрута на карте
3. Пользователь нажимает кнопку “Start Journey”	<ul style="list-style-type: none"> • Система вычисляет оптимальную траекторию маршрута и выводит её на экран

Таблица 3 – Ход действий для проложения маршрута

Вариант использования	Создание карты препятствий
Актор	<ul style="list-style-type: none"> Пользователь
Предусловия	<ul style="list-style-type: none"> Пользователь должен запустить приложение и нажать кнопку “Map editor”
Краткое описание	<ul style="list-style-type: none"> Пользователь на пустой карте устанавливает препятствия и задаёт им индекс непроходимости, затем сохраняет результат
Постусловия	<ul style="list-style-type: none"> С помощью редактора строится карта, которая затем сохраняется в отдельный файл

Таблица 4 – Создание карты препятствий

Ход действий для создания карты препятствий

Действия актора	Отклик системы
1. Пользователь заходит в режим редактора карт	<ul style="list-style-type: none"> Система представляет рабочее пространство редактора карт
2. Пользователь отмечает граничные точки препятствия	<ul style="list-style-type: none"> Система соединяет их и выводит геометрическое представление нового объекта
3. Пользователь вводит значение индекса непроходимости	<ul style="list-style-type: none"> На объекте препятствия отображается индивидуальный индекс непроходимости
4. Пользователь нажимает кнопку “Save”	<ul style="list-style-type: none"> Система сохраняет созданную карту в отдельный .xml файл

Таблица 5 – Ход действий для проложения маршрута

3. АРХИТЕКТУРА ПРИЛОЖЕНИЯ

В данном разделе описан выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов. Показаны связи выбранных элементов структуры.

3.1 Спецификация классов.

- **MainWindow** – Класс, представляющий главное окно приложения.
 - Атрибуты:
 - **UI::MainWindow *ui** – указатель на объект интерфейса главного окна
 - Методы:
 - **void openMapWindow()** – Открывает окно для работы с картой
 - **void openRouteWindow()** – Открывает окно для построения маршрута
- **InterfaceMap** – Класс, представляющий интерфейс для работы с картой.
 - Атрибуты:
 - **UI::InterfaceMap *ui** – указатель на объект интерфейса InterfaceMap
 - **QGraphicsScene *scene** – Указатель на графическую сцену
 - **Obstacle obstacle** – Объект препятствия
 - Методы:
 - **void mousePressEvent(QMouseEvent *event)** – Обрабатывает событие нажатия клавиши мыши
 - **void backToMain()** – Возвращает пользователя в главное меню
 - **void on_button_ClearMap_clicked()** – Обрабатывает нажатие кнопки “Clear Map”

- **void on_button_Save_clicked()** – Обрабатывает нажатие кнопки “Save”
-
- **InterfaceRoute** – Класс, представляющий интерфейс для построения маршрутов.
 - Атрибуты:
 - **UI::InterfaceRoute *ui** – указатель на объект интерфейса InterfaceRoute
 - **QGraphicsScene *scene** – Указатель на графическую сцену
 - **Route route** – Объект маршрута
 - **QPointF* StartPoint = nullptr** – Указатель на начальную точку маршрута
 - **QPointF* FinishPoint = nullptr** – Указатель на конечную точку маршрута
 - **int n = 1200** – Параметр для определения соседей
 - Методы:
 - **void mousePressEvent(QMouseEvent *)** – Обрабатывает событие нажатия клавиши мыши
 - **void backToMain()** – Возвращает пользователя в главное меню
 - **void on_button_LoadingMap_clicked()** – Обрабатывает нажатие кнопки “Loading Map”
 - **void on_button_StartJourney_clicked()** – Обрабатывает нажатие кнопки “Start Journey”
 - **void on_button_Save_clicked()** – Обрабатывает нажатие кнопки “Save”

- **void on_button_ClearWay_clicked()** – Обрабатывает нажатие кнопки “Clear Way”
-
- **Obstacle** – Класс, представляющий препятствие на карте.
 - Атрибуты:
 - **vector<int> indexes** – Индексы препятствий
 - **vector<QPolygonF> Polygons** – Полигоны препятствий на карте
 - **QPolygonF – Polygon** – Полигон препятствия
 - Методы:
 - **void addPointToPolygon(const QPointF &point)** – Добавляет точку к полигону препятствия
 - **void finalizePolygon()** – Завершает формирования полигона препятствия и добавляет его на сцену
 - **void clearAll()** – Очищает все данные о препятствиях
 - **void saveToXml()** – Сохраняет препятствия в XML-файл

 - **Route** – Класс, представляющий препятствие на карте.
 - **struct Node**
 - Атрибуты:
 - **struct Node** - Структура, представляющая узел в графе маршрута:
 - **QPointF Point** – Координаты точки узла.
 - **Double cost** – Стоимость достижения узла

- **int heuristic** – Эвристическая оценка расстояния до цели
 - **Node* parent** – Указатель на родителя узла
 - **vector<Node> WayPoints** – Узлы пути маршрута
 - **vector<QPolygonF> Polygons** – Полигоны препятствий на карте
 - **QPolygonF – Polygon** – Полигон маршрута
 - **int flag = 1** – Флаг для обозначения состояния
 - **Obstacle obstacle** - Препятствие
- Методы:
- **void loadMapFromXml(const QString& filename, QGraphicsScene* scene)** – Загружает карту из XML - файла .
 - **void findOptimalRoute(QPointF* start, QPointF* finish)** – Находит оптимальный маршрут между двумя точками на карте.
 - **double distance(QPointF* current, QPointF* neighbor)** – Вычисляет расстояние между двумя точками
 - **int heuristic(int x1, int y1, int x2, int y2)** – Вычисляет эвристическую оценку расстояния между двумя точками.
 - **int findCost(Node* current, Node* goal)** – Находит стоимость прохода от текущей точки к цели.
 - **vector<Node> getNeighbors(Node* node, Node* goal, int n)** – Возвращает соседние узлы для заданного узла.
 - **bool searchPoint(QPointF)** – Проверяет, содержит ли сцена точку.
 - **vector<Node> aStar(Node start, Node goal, int n)** – Выполняет поиск оптимального маршрута с использованием алгоритма A*.

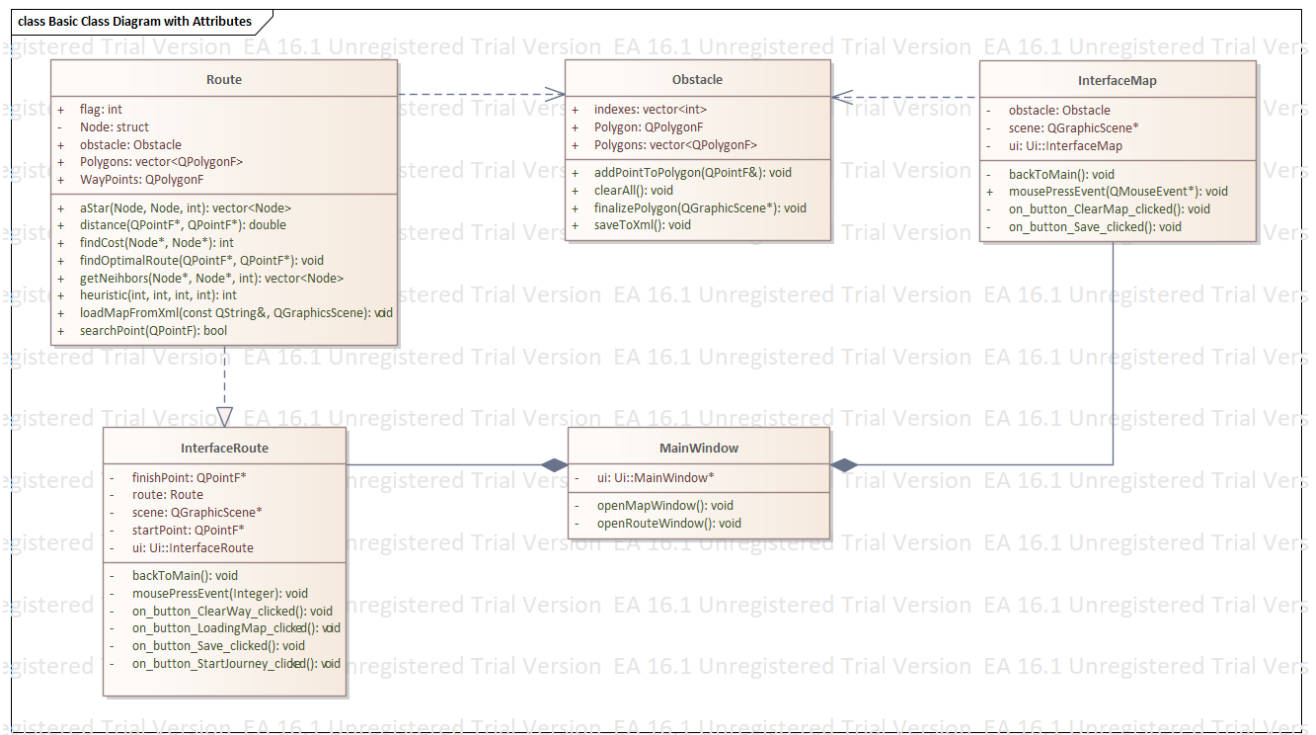


Рисунок 2 – Диаграмма классов

3.2 Обоснование выбора алгоритмов

- Для реализации требуемого функционала вычисления оптимального маршрута с учетом препятствий, в программе был реализован математический алгоритм **A*(А стар)**

Общие сведения:

- Поиск A*** — алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).
- Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как $f(x)$). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как $g(x)$ и может быть как эвристической, так и нет), и функции эвристической оценки

расстояния от рассматриваемой вершины к конечной (обозначается как $h(x)$).

- **Оценка сложности:** Временная сложность алгоритма A^* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда пространство поиска является деревом, а эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

где h^* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели.

- Выбранный алгоритм является подходящим, потому что подразумевает работу с оценкой пути через каждую точку, что являлось одним из ключевых требований при выборе алгоритма, ввиду важности учёта расположения, формы, а также индекса непроходимости препятствий на карте.
- **Логика реализации:**

Алгоритм происходит таким образом: сначала мы проходимся по точкам-соседей, добавляя им эвристическую величину и вес. Получая соседей, сравниваются сумма веса и эвристическая величина каждой точки и выбирается с наименьшим значением. Таким образом, пока не дойдем до конечной точки, алгоритм работает. После чего, когда появилось множество точек, проходимся по множеству повторно, чтобы построить маршрут из точек. Физическая модель объекта препятствия определена как объект непроходимости, через который маршрут не может проходить (за исключением случая, когда отношение индекса непроходимости препятствия и его размера позволяет сократить время пути прохождением через препятствие насквозь (индекс < 40)).

Непроходимыми считаются препятствия, у которых индекс непроходимости больше 40, для физической модели можно взять

заболоченность или гору, **труднодоступными** считаются препятствия с индексом меньше 40, через них маршрут может проходить для сокращения времени, их аналогами физической модели могут быть пустыня или лес.

- **Детали реализации:** Вершины графа хранятся в виде структуры **Node**(см. 3.1 Спецификация классов), при запуске алгоритма создаются три вектора типа **Node**, для посещенных, открытых и закрытых вершин:

```
std::vector<Node> closedSet;
```

```
std::vector<Node> openSet;
```

```
std::vector<Node> visited;
```

- Для получения соседей используется проход по направлениям, мы создаем:

```
static const std::vector<std::pair<int, int>> directions = {
```

```
    {1, 0}, {-1, 0}, {0, 1}, {0, -1}, // Основные направления
```

```
    {1, 1}, {-1, 1}, {1, -1}, {-1, -1} // Диагональные направления
```

```
};
```

```
QLineF Vector = QLineF(node->Point.x(), node->Point.y(), goal->Point.x(), goal->Point.y()).unitVector();
```

- При нахождении соседей, необходимо совершать проверки:

```
if(isValid(Vector.x2(), Vector.y2(), n)){
```

```
    QPointF directNeighbor(Vector.x2(), Vector.y2());
```

```
    if (!searchPoint(directNeighbor))
```

```
        neighbors.emplace_back(Node(Vector.x2(), Vector.y2(), node->cost + Vector.length(), heuristic(Vector.x2(), Vector.y2(), goal->Point.x(), goal->Point.y())));
```

```
}
```

- Где **isValid** – устанавливает границы на создание соседних точек
- **searchPoint** – проверяет нахождение конечной точки внутри препятствия, а так же индекс непроходимости, препятствие с индексом больше 40 считается непроходимым, а препятствие с индексом меньше 40 считается труднодоступным, через которое маршрут может проходить.

- Поиск соседей по направлениям с вычислением стоимости прохода:

```
for (const auto& dir : directions) {
    int nx = x + dir.first;
    int ny = y + dir.second;

    // проверка является ли новая позиция внутри допустимых границ
    if (isValid(nx, ny, n)) {
        QPointF neighborPoint(nx, ny);
        if (!searchPoint(neighborPoint)) {
            // вычисление стоимости перемещения
            double moveCost = (dir.first != 0 && dir.second != 0) ? sqrt(2) : 1;

            //добавляем соседа с учетом стоимости и эвристической оценки до цели
            neighbors.emplace_back(Node(nx, ny, node->cost + moveCost, heuristic(nx, ny, goal->Point.x(), goal->Point.y())));
        }
    }
}
```

- Заполняется вектор открытых вершин, для этого среди новых точек на каждом шаге сравнивается их сумма стоимости прохода и эвристической оценки, из новых берут ту новую точку, у которой расстояние будет меньше рассматриваемой:

```
for (int i = 1; i < openSet.size(); i++) {
    if (openSet[i].cost + openSet[i].heuristic < current.cost + current.heuristic) {
        current = openSet[i];
        currentIndex = i;
    }
}
```

- При проверке точек-соседей, мы проверяем точки на близость к соседу:

```
if (!searchPoint(neighbor.Point) || obstacle.indexes[&neighbor - &neighbors[0]] <= 40)
```

- Если проверка на близость пройдена, то проверяем на закрытость:

```
for (Node closedNode : closedSet) {
    if (closedNode.Point.x() == neighbor.Point.x() && closedNode.Point.y() ==
neighbor.Point.y()) {
```

```
isClosed = true;
```

```
break;
```

- Если флаг на предыдущем шаге флаг не изменился, то проверяем на открытость соседа:

```
for (Node& openNode : openSet) {
```

```
    if (openNode.Point.x() == neighbor.Point.x() && openNode.Point.y() ==  
neighbor.Point.y()) {
```

```
        isOpen = true;
```

- Если новый путь оказался короче, то обновляем информацию:

```
        if (current.cost < openNode.cost) {
```

```
            openNode.cost = current.cost + distance(&current.Point, &neighbor.Point);
```

```
            openNode.parent = &current;
```

```
        }
```

- Если новый сосед не был открыт, то добавляем в список открытых узлов:

```
        if (!isOpen) {
```

```
            neighbor.cost = current.cost + distance(&current.Point, &neighbor.Point);
```

```
            neighbor.parent = &current;
```

```
            openSet.emplace_back(neighbor);
```

```
        }
```

- Если текущая точка на пути является конечной:

```
if (current.Point.x() == goal.Point.x() && current.Point.y() == goal.Point.y())
```

- То алгоритм выведет итог кратчайшего пути, будет создан вектор, содержащий точки маршрута.

3.3 Дополнительные диаграммы

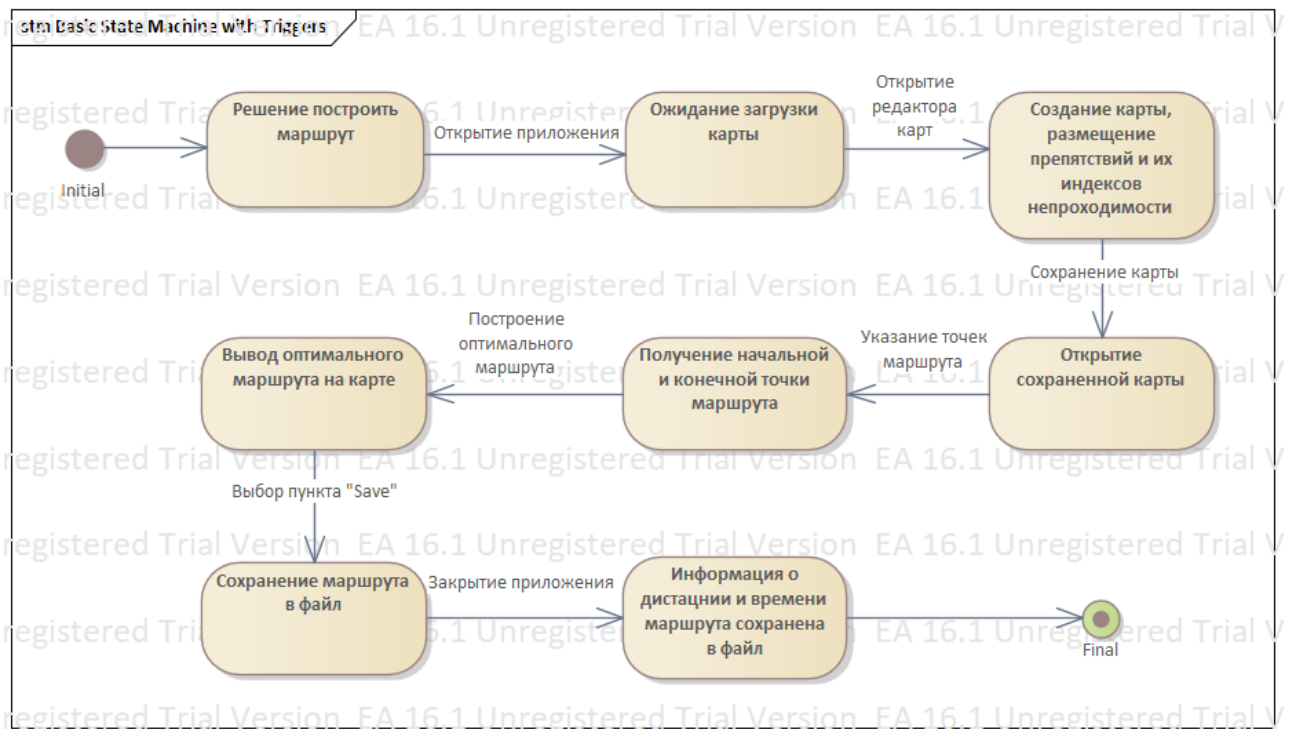


Рисунок 3 - Диаграмма состояний

- На диаграмме состояний для моделирования динамических аспектов системы представлены состояния и переходы системы.
- В элементах состояния представлены контрольные точки работы системы, такие как:
 - ожидание необходимых данных или действий от пользователя
 - проведение вычислений(построение маршрутов)
- В переходах показаны промежуточные контрольные точки, в которых происходит либо ввод необходимых данных от пользователя(в т.ч. выбор пунктов меню), либо происходят промежуточные вычисления.

На диаграмме деятельности(Рис. 4) представлены динамические аспекты поведения системы. На диаграмме деятельности в виде блок-схемы визуализирован случай использования с учётом принятия разных решений, в

зависимости от намерения пользователя(построить новую карту/открыть готовую).

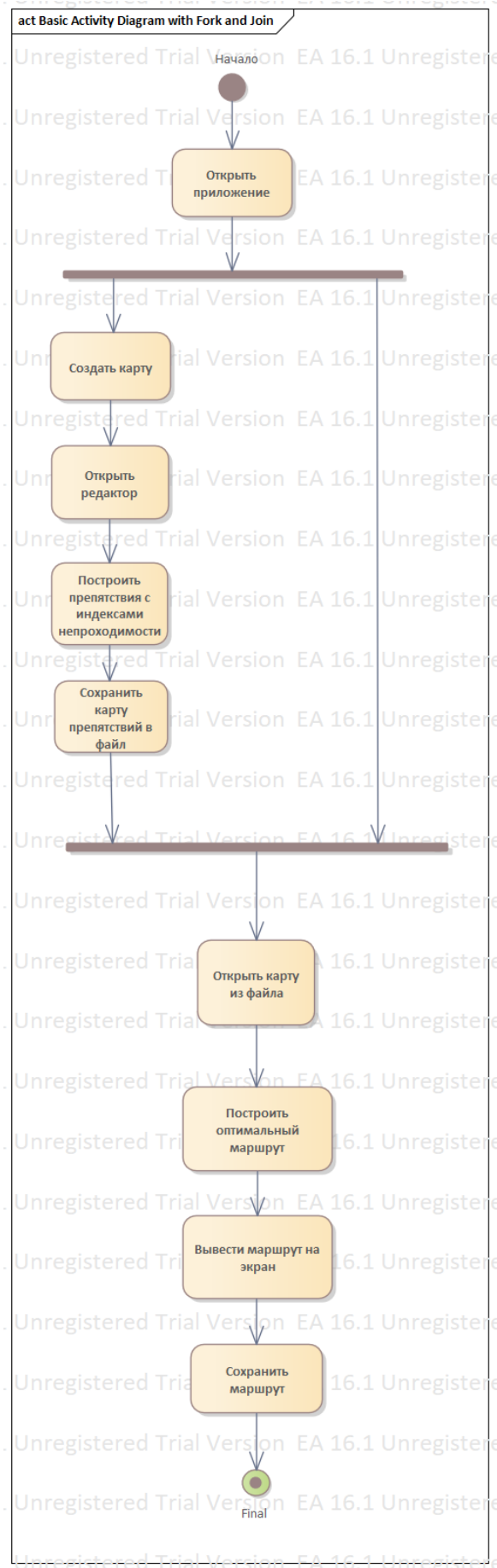
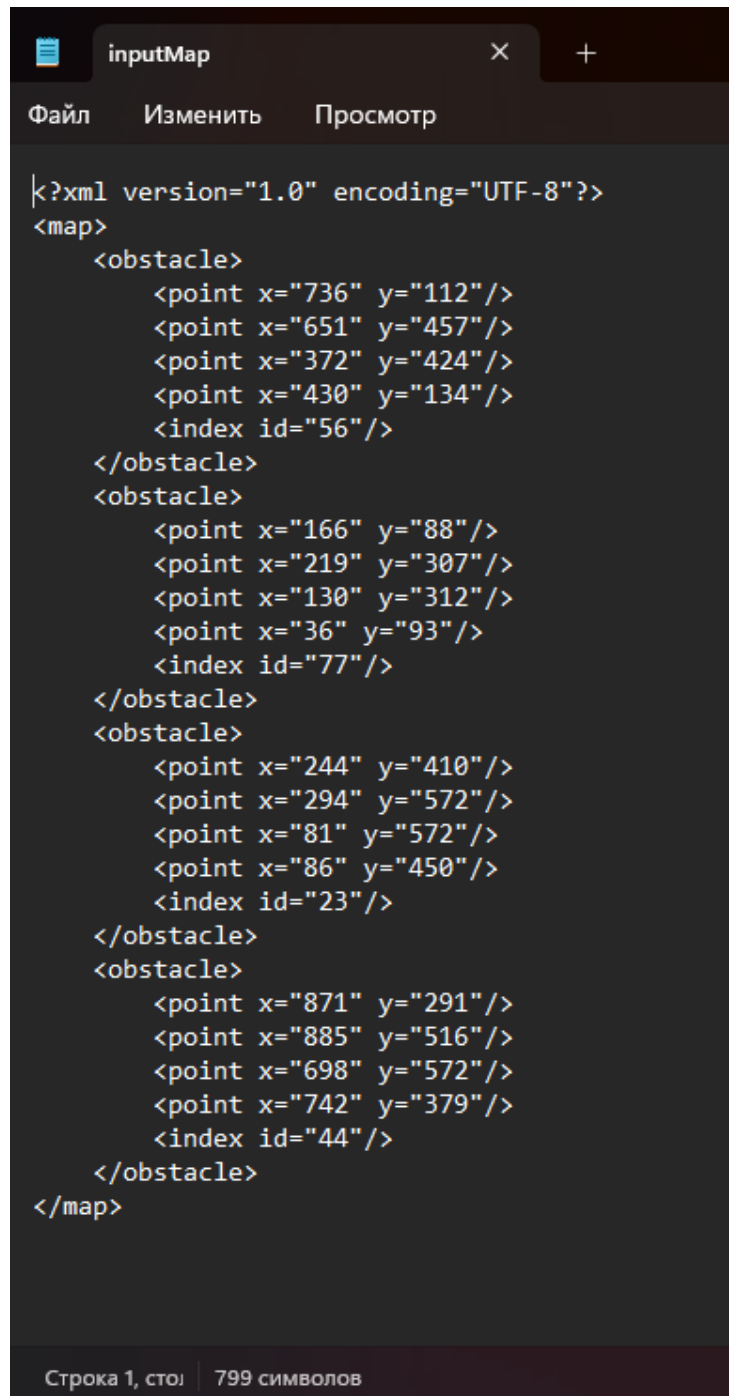


Рисунок 4 - Диаграмма деятельности

3.4 Форматы входных и выходных файлов

- Согласно требованиям, формат сохранения и загрузки карт и маршрутов – XML.
- После создания новой карты и размещения на ней препятствий предлагается сохранить её в отдельный файл формата XML. В сохранённом файле хранится информация о каждом препятствии, координаты граничных точек, а также индекс непроходимости(Рис.5.)



```
<?xml version="1.0" encoding="UTF-8"?>
<map>
  <obstacle>
    <point x="736" y="112"/>
    <point x="651" y="457"/>
    <point x="372" y="424"/>
    <point x="430" y="134"/>
    <index id="56"/>
  </obstacle>
  <obstacle>
    <point x="166" y="88"/>
    <point x="219" y="307"/>
    <point x="130" y="312"/>
    <point x="36" y="93"/>
    <index id="77"/>
  </obstacle>
  <obstacle>
    <point x="244" y="410"/>
    <point x="294" y="572"/>
    <point x="81" y="572"/>
    <point x="86" y="450"/>
    <index id="23"/>
  </obstacle>
  <obstacle>
    <point x="871" y="291"/>
    <point x="885" y="516"/>
    <point x="698" y="572"/>
    <point x="742" y="379"/>
    <index id="44"/>
  </obstacle>
</map>
```

Строка 1, столбец 1 | 799 символов

Рисунок 5 – Пример входного файла

- После проложения маршрута предлагается сохранить информацию о нём в отдельный файл формата XML. В полученном файле(см. Рис.6) хранится информация о проложенном маршруте, а именно:
 - Все контрольные точки (**WayPoints**) с их координатами, “стоимостью” прохождения и эвристической оценкой
 - В конце выходного файла создается раздел “**Statistics**”, в котором сохранены численные значения пройденной дистанции, общее время прохождения пути, а также средняя скорость.

```
<WayPoint x="890" y="87" cost="0" heuristic="18917"/>
<WayPoint x="891" y="86" cost="0" heuristic="18975"/>
<WayPoint x="892" y="85" cost="0" heuristic="19033"/>
<WayPoint x="893" y="84" cost="0" heuristic="19092"/>
<WayPoint x="894" y="83" cost="0" heuristic="19151"/>
<WayPoint x="896" y="81" cost="0" heuristic="19268"/>
</WayPoints>
<Statistics>
  <TotalDistance>878</TotalDistance>
  <TotalTime>175.6</TotalTime>
  <AverageSpeed>5</AverageSpeed>
</Statistics>
</Route>
```

Рисунок 5 – Пример выходного файла с информацией о маршруте.

4. ТЕСТИРОВАНИЕ

- В ходе разработки проекта на разных этапах производились тестовые мероприятия:
- При разработке каждого структурного модуля программы перед интеграцией в общую структуру производилось тестирование:
 - 1) Статическое – ревизия кода
 - 2) Динамическое – модульное тестирование
- После подключения модуля в общую архитектуру производились мероприятия по интеграционному тестированию на проверку взаимодействия между собой частей общей программы.
- Системное тестирование производилось в следующем тестовом окружении:
 - ОС – Windows 10, 11
 - Версия приложения 1.1.0

5. ДЕМОНСТРАЦИЯ.

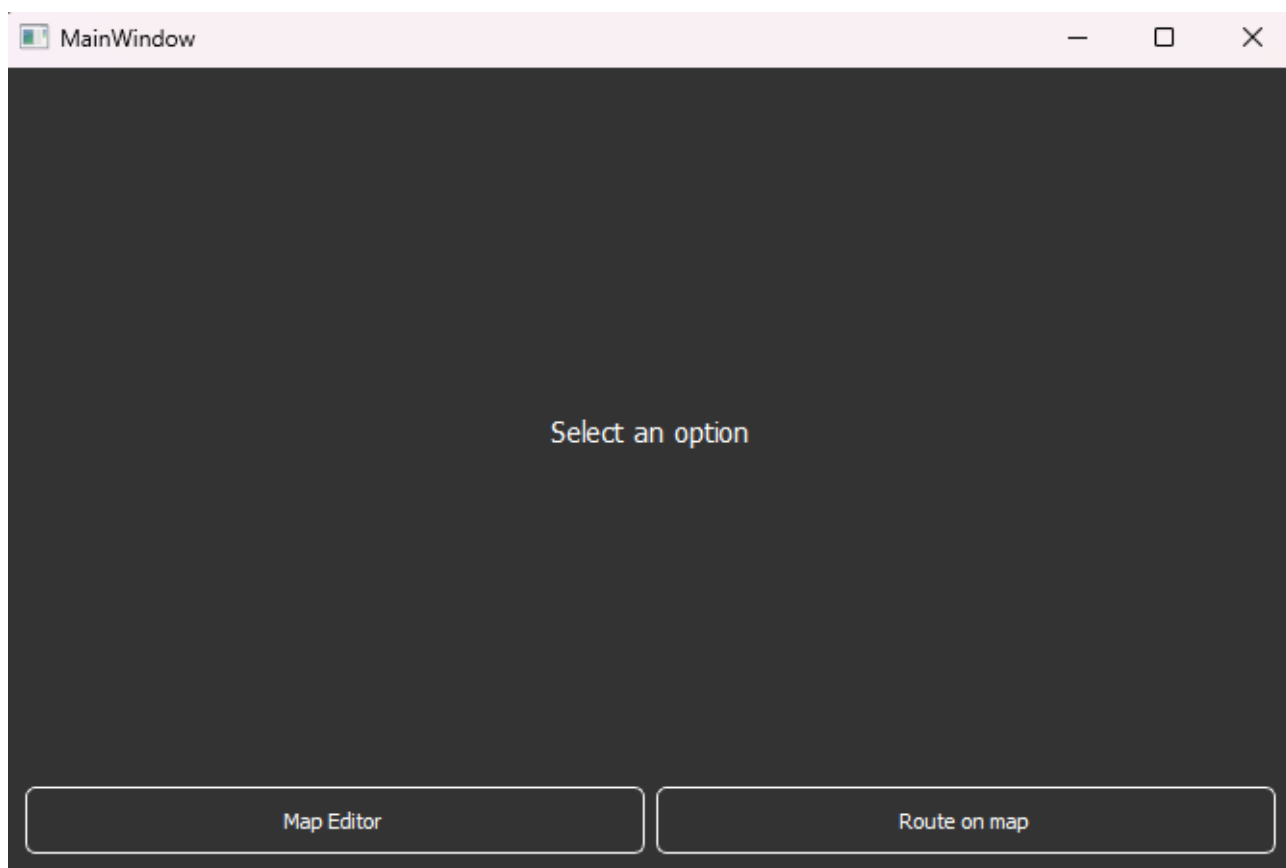


Рисунок 6 – Открытие главного меню.

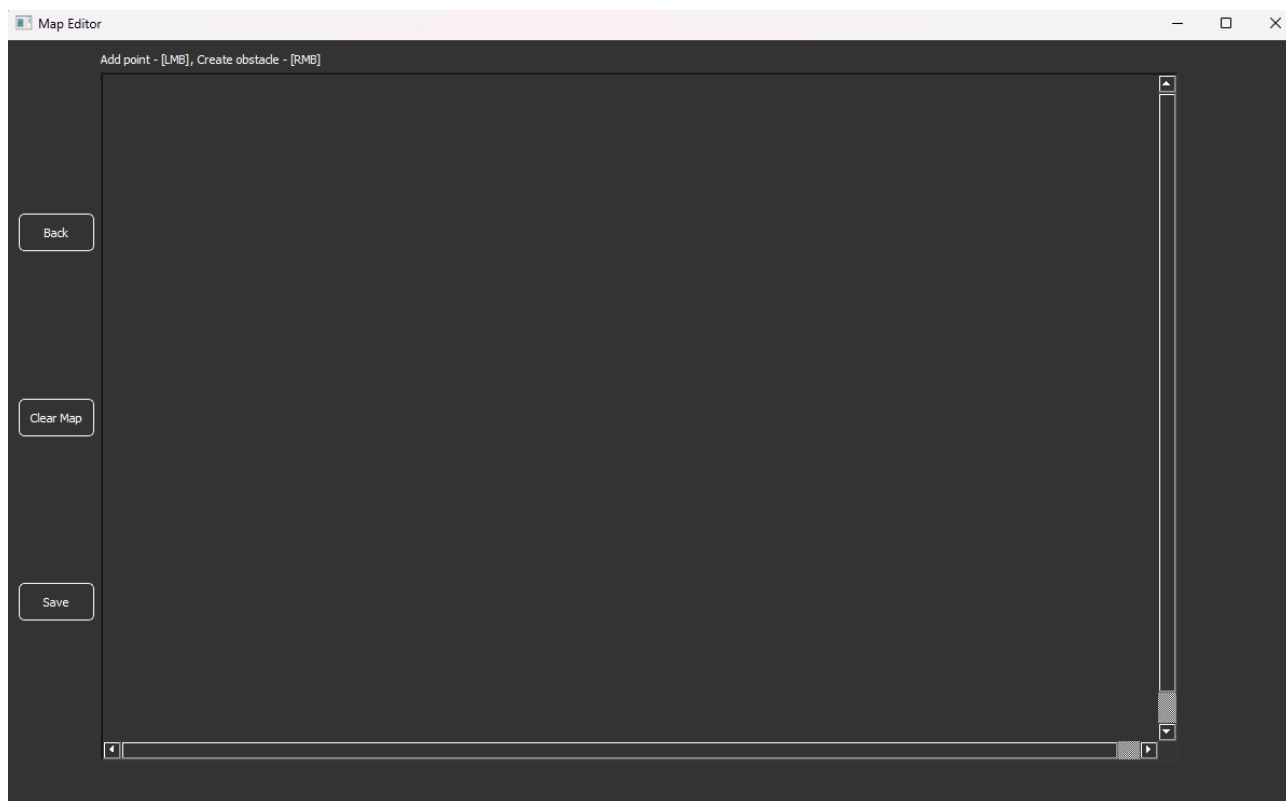


Рисунок 7 – Режим создания карты.

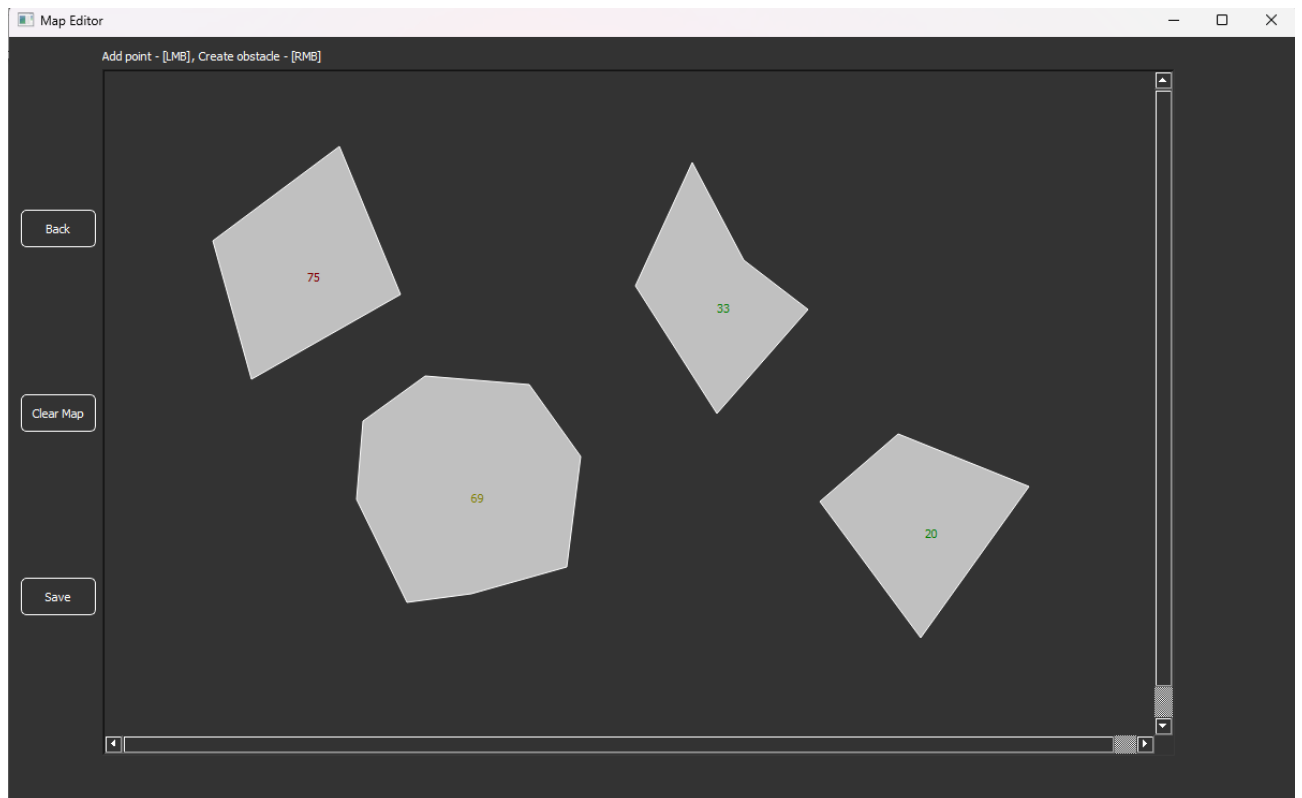
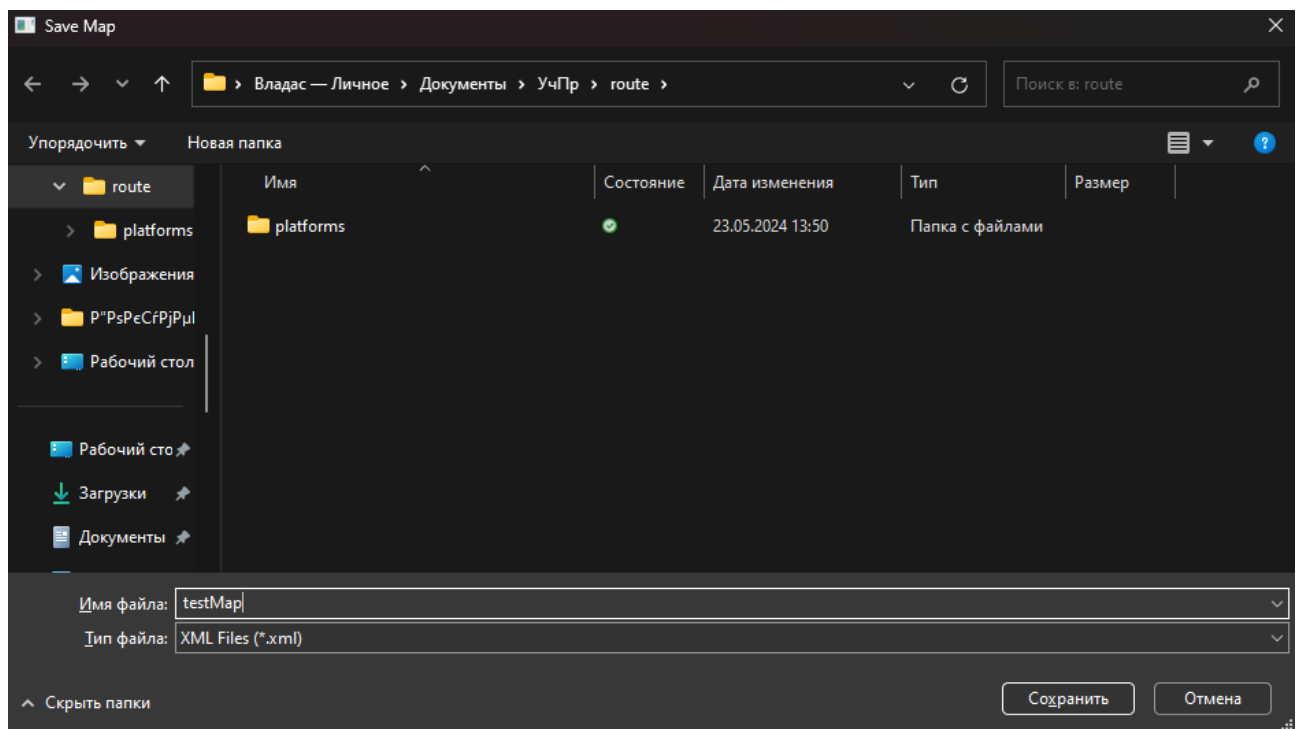
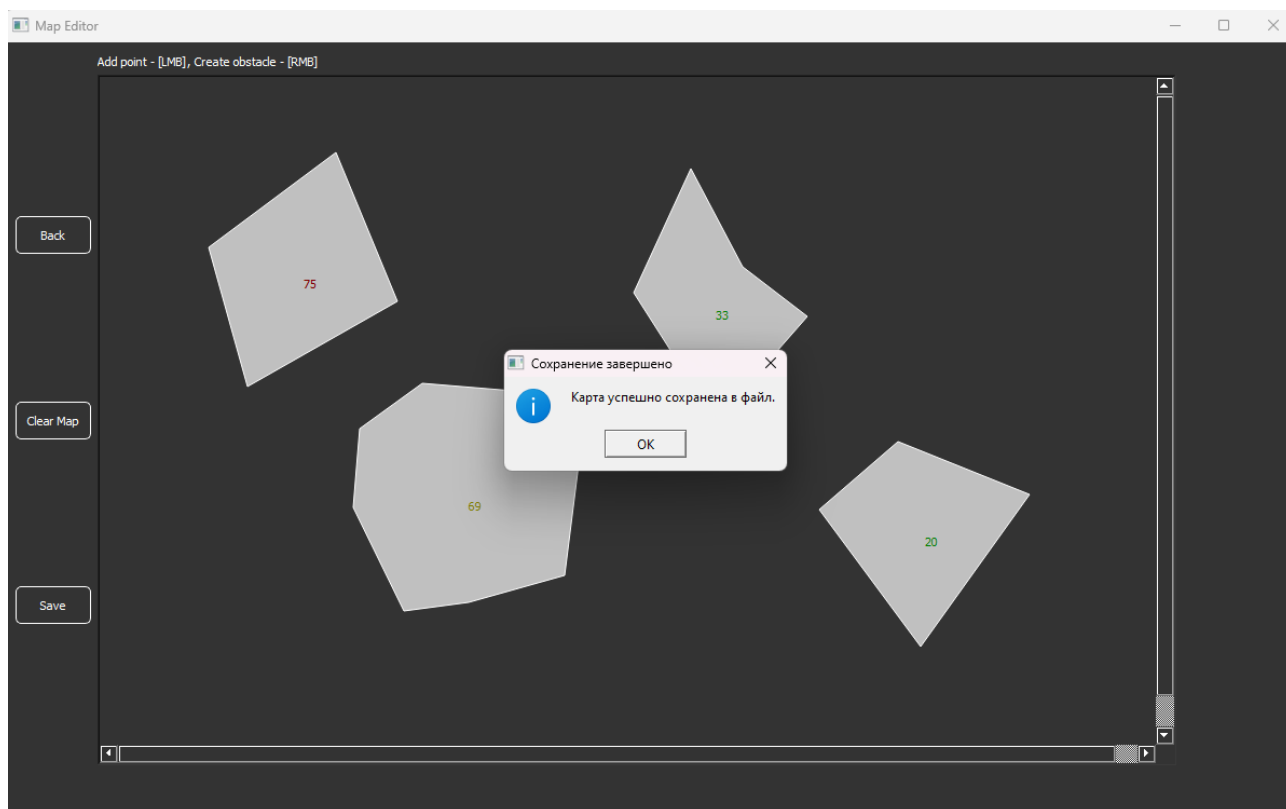


Рисунок 8 – Размещение препятствий на карте.



a)



б)

Рисунки 9а, 9б – Сохранение карты в файл.

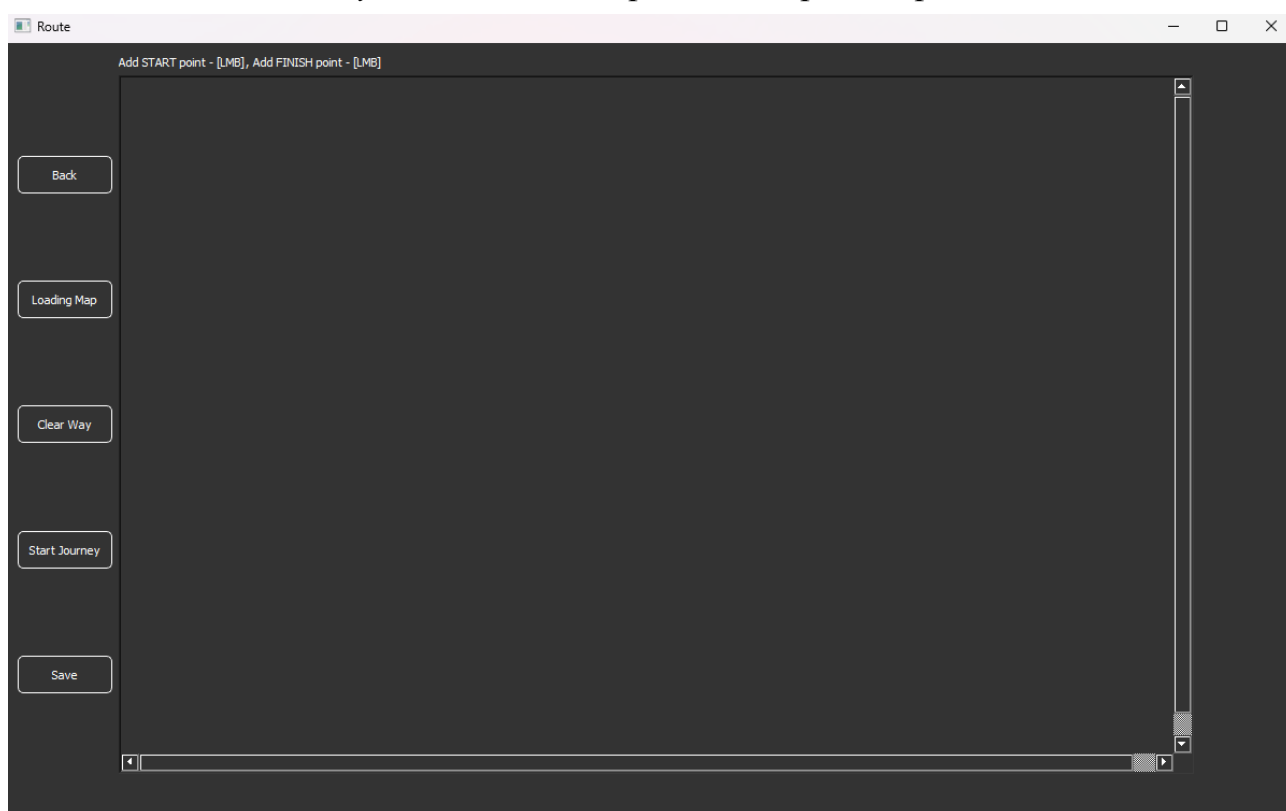
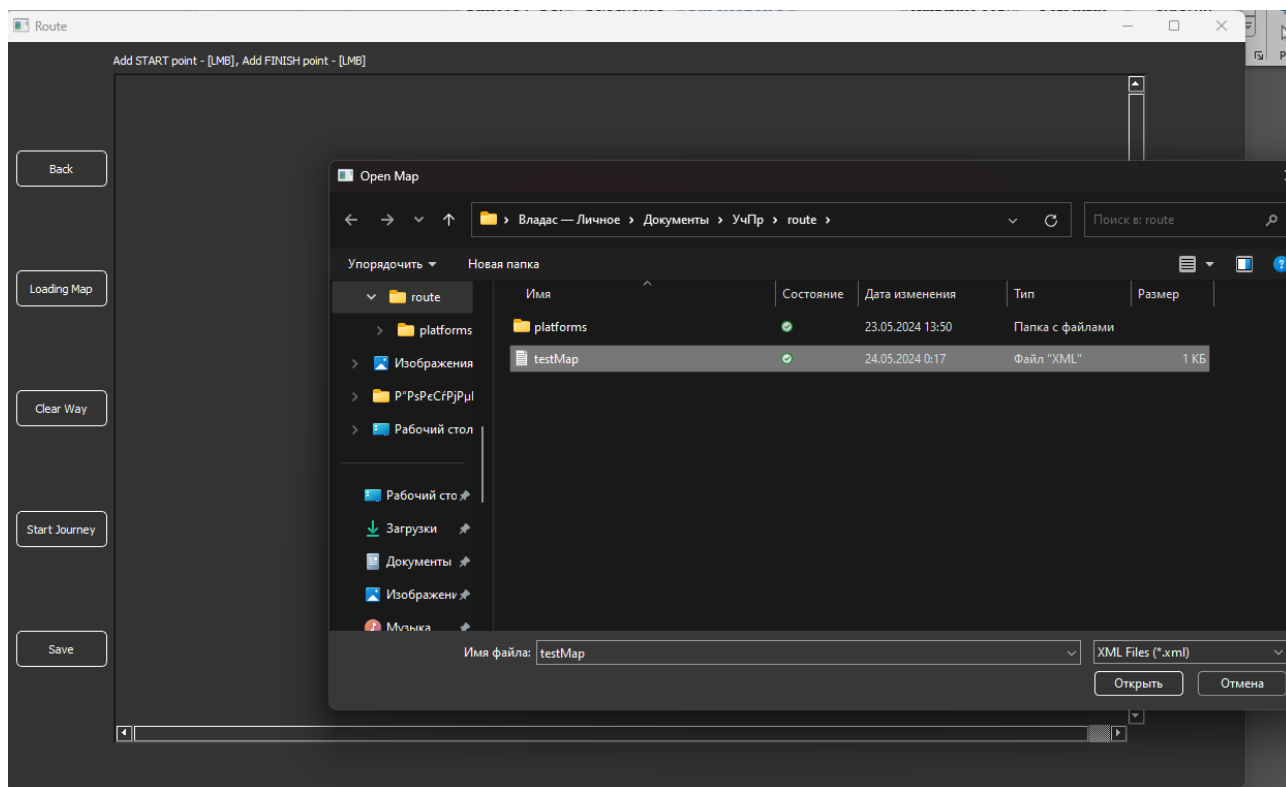
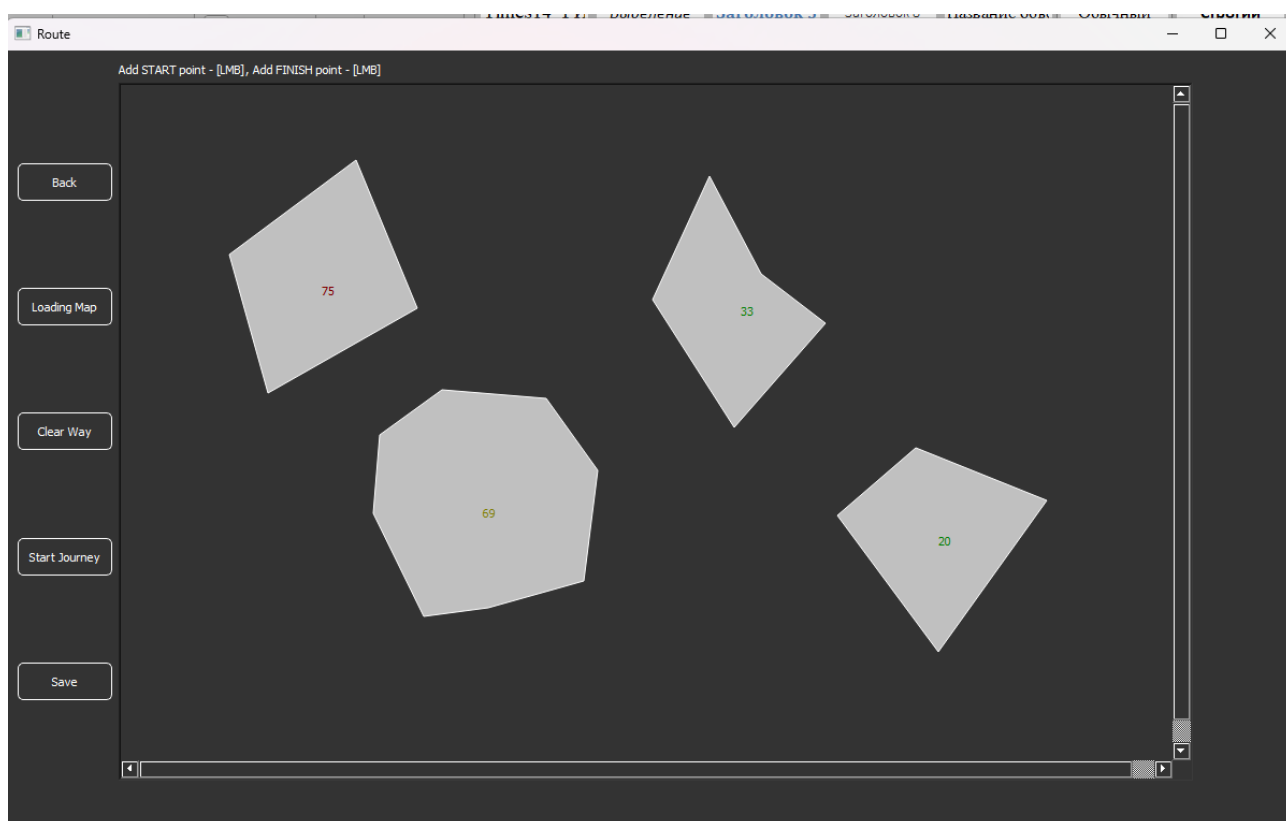


Рисунок 10 – Открытие меню маршрута.



а)



б)

Рисунки 11а, 9б – Открытие карты из файла.

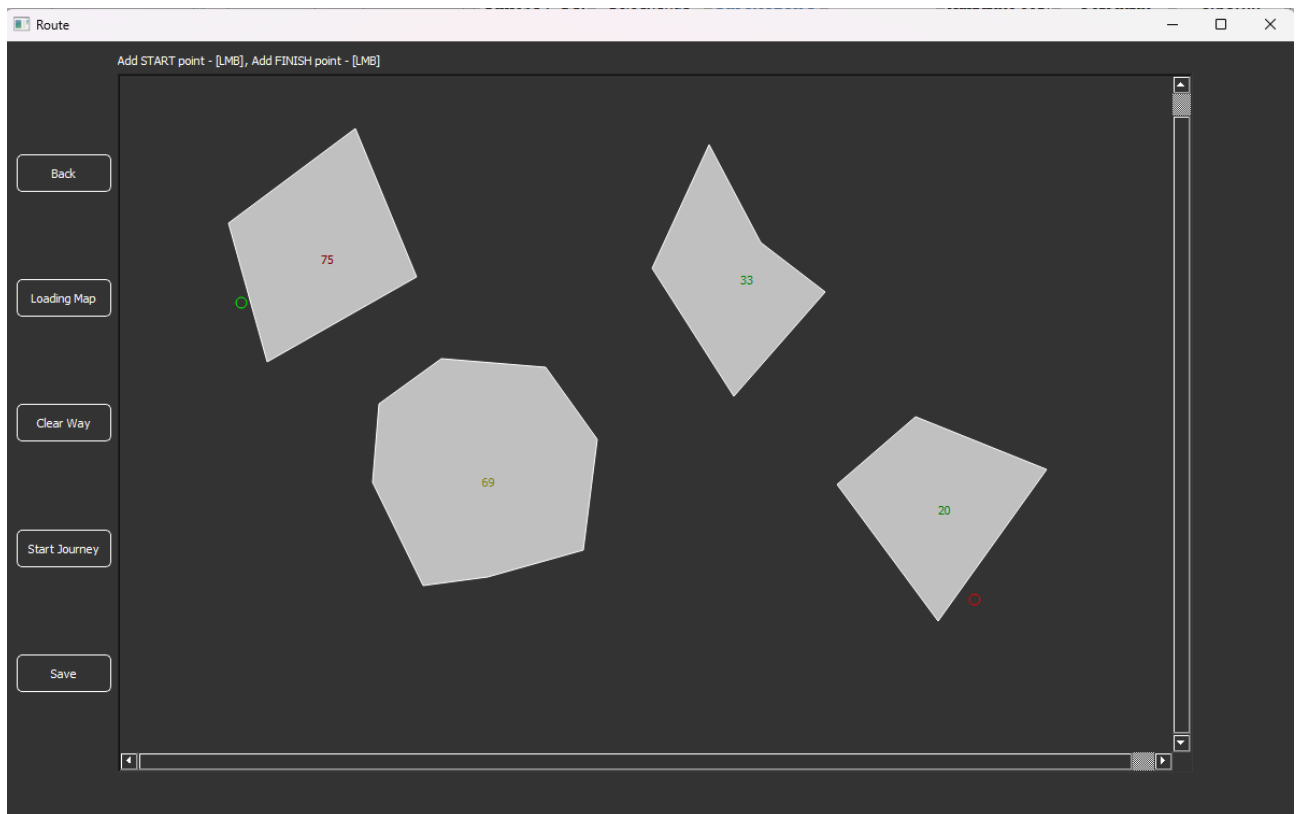


Рисунок 12 – Установка начальной и конечной точки маршрута.

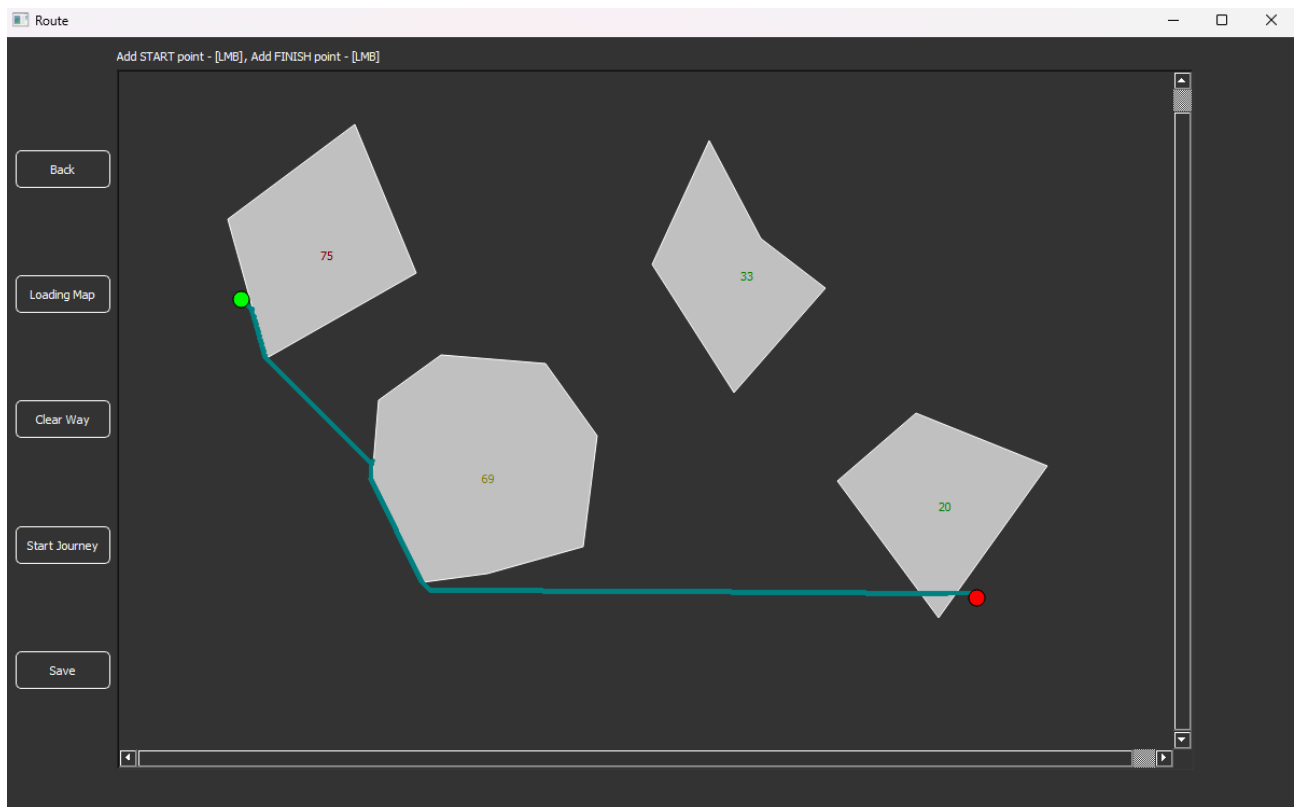
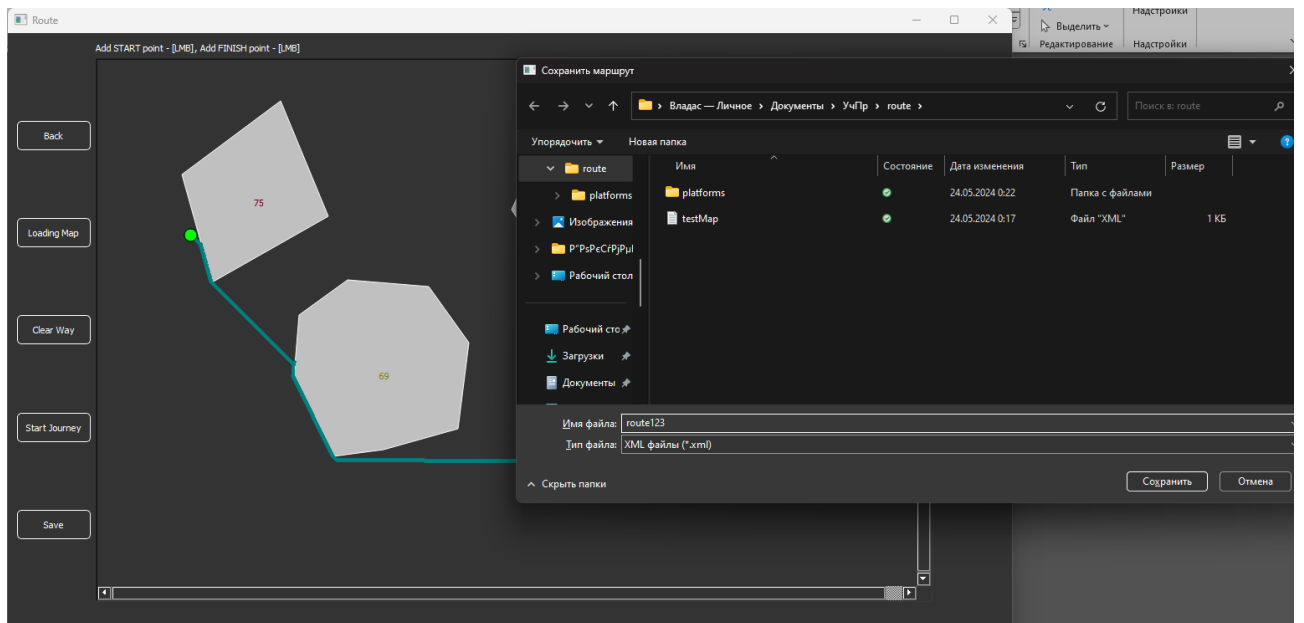
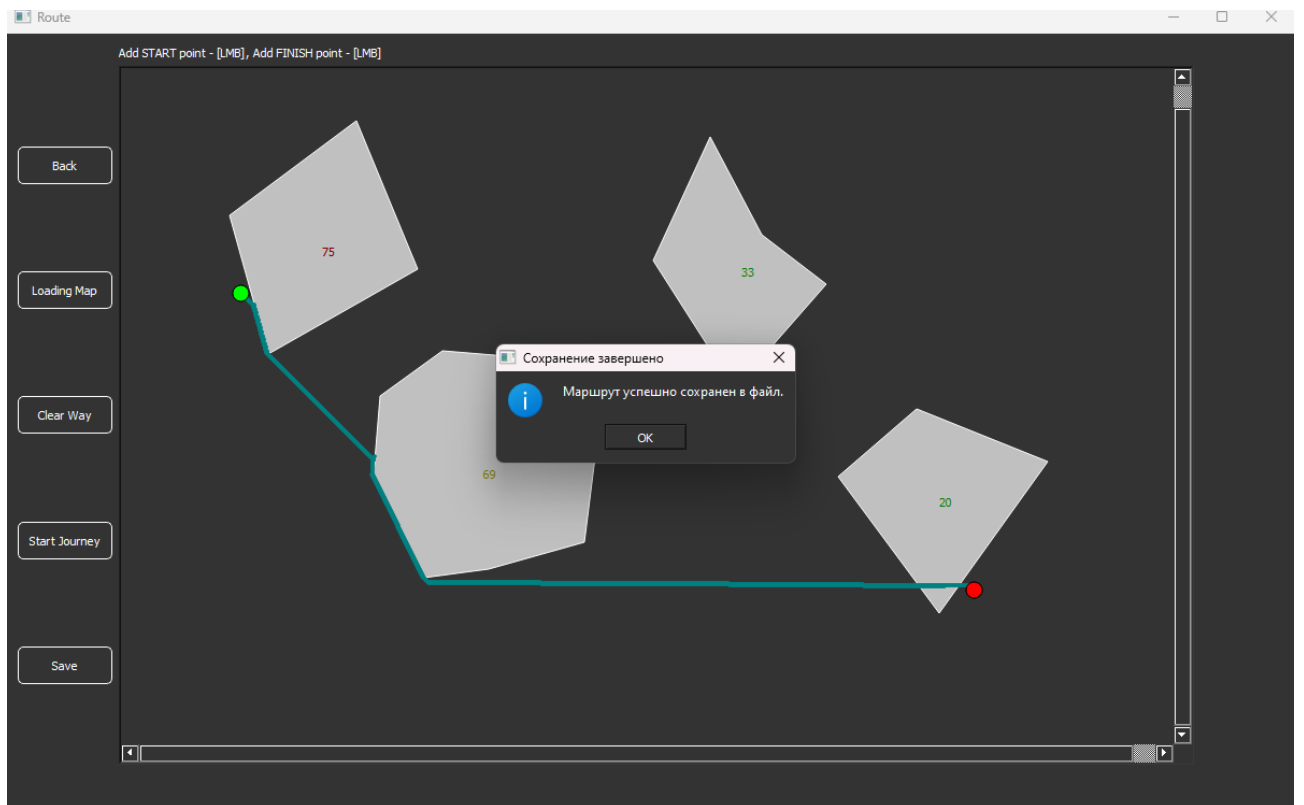


Рисунок 13 – Построенный маршрут.



а)



б)

Рисунки 14а, 14б – Сохранение маршрута в файл.

ЗАКЛЮЧЕНИЕ

В ходе выполнения проекта в рамках учебной практики было реализовано приложение для построения карты препятствий и нахождения оптимального маршрута на ней. При разработке приложения была использована среда разработки Qt Creator и язык программирования C++. В ходе разработки были учтены и исполнены заданные требования, описанные на основе заданного описания работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Жасмин Бланшет, Марк Саммерфилд. «Qt 4: Программирование Qt на C++» - 2006 г. – 555 с.
2. Документация Qt. URL: <https://doc.qt.io>
3. Википедия – Поиск A*: https://ru.wikipedia.org/wiki/A*/