содержание

Введение

Время — невосполнимый и поэтому самый дорогой ресурс человека. Его экономят специалисты различных отраслей и обычные люди. Компьютеры и программные приложения позволяют существенно сокращать временные затраты. Компьютеры обладают высокой скоростью и точностью вычислений, дают возможность обрабатывать огромные массивы информации. Техника с такими реализованными параметрами удовлетворяет потребность в решении как уникальных производственных, так и бытовых рутинных задач.

В начале развития вычислительных устройств были созданы сложные в понимании языки машинного уровня, программирование на этих языках позволяет добиться высокой оптимизации и скорости вычислений, однако разработка программ занимает довольно существенное количество времени. Для экономии времени и упрощения понимания алгоритмов были разработаны языки функционального уровня, они значительно сокращали временные затраты при разработке приложения, но при попытке описать вещи из реального мира возникают проблемы с поддержанием целостности программы, появляется многократное повторение одного и того же кода.

Чтобы минимизировать проблему повторения кода, были разработаны объектно-ориентированные языки программирования, они позволили ввести понятие ориентации на “объекты”. Введя данное понятие, разработчики смогли приблизить понимание кода к реальным вещам. Объекты могут хранить информацию о экземпляре, и методах.

Однако и у объектно-ориентированного языка есть проблемы, объектно-ориентированные парадигмы плохо реализуют реальное поведение вещей и из этого выливается проблема грамотной проработки архитектуры будущего приложения.

Визуальный компонент для построения графов и поиска кратчайшего пути позволяет построить условный путь между точкой старта и финиша, он может использоваться программистами как готовый к употреблению блок при реализации различных прикладных проектов, что существенно экономит время на создание новых программных продуктов.

Разрабатываемый компонент позволит прикладным программистам: быстро рассчитывать стоимость пройденного пути;

* в короткие сроки обработать большие массивы информации;
* визуализировать и взаимодействовать с динамической картинкой.

Если данный процесс производить вручную, то будет затрачено большое количество времени в сравнении со скоростью компьютера.

Компонент сможет рассчитывать минимально возможный путь до цели, что в современном мире может найти применение в таких приложениях как:

* карты;
* маршрутизаторы;
* игры, в которых есть необходимость поиска минимального пути.

Данная разработка для обычного пользователя даст удобные механизмы взаимодействия с приложением, как для выбора точек старта и финиша, так и для запуска алгоритма поиска. Прикладным программистам данная разработка даёт возможность простой настройки компонента. Для того, чтобы взаимодействовать с компонентом, потребуется всего лишь добавить исходные файлы, скомпилировать проект, переместить компонент на форму.

1. ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ

В ходе разработки компонента на рассмотрение были выбраны две среды программирования:

* Microsoft Visual Studio c# 2019;
* Borland Delphi 2007;

Рассмотрим стороны выдвинутых сред программирования:

Первая из сред разработки - Microsoft Visual Studio с# 2019. Это - молодая среда, являющаяся бесплатным решением, предоставляемым компанией Microsoft. Но в этом есть и минус – доступность среды напрямую зависит от компании. Данная аппаратное обеспечение разработки позволяет настраивать свой интерфейс под пользователя, но при первом знакомстве с интерфейсом может показаться перегруженным. Visual Studio отвечает современным требованиям: в ней есть подсказки для написания кода, удобные средства отладки кода, возможность быстро устанавливать сторонние библиотеки через пакеты NuGet, поддержка репозиториев Git. Microsoft введёт официальную документацию по языку c#. Поскольку среда программирования отвечает современным требованиям скорость её загрузки напрямую зависит от комплектующих компьютера.

Второй из рассматриваемых инструментов для реализации проекта - Borland Delphi 2007. Данная среда программирования - разработка компании Borland и является платным решением. Синтаксис данного средства разработки лёгкий для понимания. К негативной стороне можно отнести то, что: нет возможности настраивать рабочее место, проблемный метод компиляции компонентов, нет подсказок по коду. Delphi не требовательна к комплектующим компьютера.

После выявления всех достоинств и недостатков была выбрана среда разработки Microsoft Visual Studio c# 2019, так как она позволяет подстраивать свой интерфейс под разработчика, также язык c# является популярным и востребованным языком программирования, его вероятней можно встретить на будущих местах работы.

Чтобы проверить актуальность реализации проекта был произведен поиск аналогов разрабатываемого компонента. После поиска в сети доступных продуктов была найдена похожая программа с сайта https://vscode.ru/. При изучении исходного кода были найдены как достоинства, так и недостатки:

К достоинствам данной программы можно отнести то, что имеется алгоритм построения всех возможных путей.

К недостаткам можно отнести:

* код программы реализован в тестовом приложении;
* нет возможности поиска минимально возможного пути между графами;
* нет возможности динамического рисования картинки.

В итоге по сравнению с найденным программным решением разрабатываемый компонент будет обладать достоинствами:

* все методы будут описаны во внешней библиотеке.
* есть возможность построения кратчайшего пути;

Данная разработка будет конкурировать с своими аналогами тем, что продукт будет свободно распространятся в виде библиотеки.

Для разработки компонента был выбран родительский класс – Control.

Данный класс имеет необходимые для реализации компонента методы взаимодействия c манипулятором “мышь”, и так же у него есть инструменты для рисования изображения.

Для управления компонентом будет задействован манипулятор “мышь”, при взаимодействии с которым будут реализованы методы взаимодействия с компонентом. Также компонент будет взаимодействовать клавиатурой, от которой компонент будет ожидать ввода значений. Разрабатываемый компонент хорошо иллюстрирует заложенные в объектно-ориентированные языки парадигмы:

Инкапсуляция – использует активные методы описанной программы что позволяет поддерживать целостность вводимых данных и производить сокрытие.

Наследование – позволяет наследовать механизмы и функции родительского класса или классов.

Полиморфизм – способность методов обрабатывать разные данные различными способами. Полиморфизм бывает нескольких типов: параметрический и ad-hoc.

ad-hoc полиморфизм – полиморфизм принудительного типа, когда один и тот же метод перегружается различными типами данных.

Параметрический полиморфизм – определение метода с обобщённым типом данных.

Абстракция – выделение основных характеристик объекта, которые описывают его с достаточной точностью.

Главной целью проекта является разработка полноценного визуального компонента для работы с графами, который можно будет встраивать в прикладные приложения. Основная идея компонента заключается в создании готового для работы полнофункционального редактора графов. По этой причине основные усилия будут предприняты для разработки качественных и удобных механизмов взаимодействия программного продукта с пользователем. Кроме этого, в рамках редактора в качестве примера механизма работы со структурой графа будет реализован алгоритм поиска кратчайшего пути. Это позволит проиллюстрировать, как можно встраивать отдельные алгоритмы внутрь компонента, либо в наследуемые от него классы. Так как все вопросы организации взаимодействия создаваемого класса и пользователя, в сущности, являются не очень сложными типовыми алгоритмами, не требующими особого внимания при разработке, то в дальнейшем при проработке материала проекта основное внимание будет уделено наиболее сложному процессу – алгоритму поиска пути.

1. КОНСТУКТОРСКИЙ РАЗДЕЛ
   1. Математическая модель

В ходе поиска основного процесса был выявлен процесс поиска пути. Существует множество поиска пути: “Волновой алгоритм”, “Алгоритм Дейкстры”, “A\*”. Из всего списка был выбран алгоритм “A\*”, потому что в его основе лежит идея поиска пути по наилучшему совпадению на графе. Данный пошагово просматривает все пути, ведущие от начальной точки в конечную, пока не найдёт минимальный. Алгоритм сначала просматривает только те пути, которые кажутся ведущими к цели. При поиске пути алгоритм учитывает весь пройденный до этого путь. Составляющая g(x) – стоимость пути от начальной вершины. В начале работы просматриваются смежные с начальным узлы. Выбирается тот из них, который имеет минимальное значение f(x), после этого алгоритм производит анализ из этого угла. На каждом этапе алгоритм оперирует с множеством путей из начального узла до всех ещё не посещённых узлов, которые размещаются в очереди с приоритетом. Приоритет пути определяется по формуле 2.1

2.1

где h(x) – стоимость перехода в этот узел.

Алгоритм продолжает свою работу до тех пор, пока значение f(x) не окажется меньшим, чем любое значение в очереди, либо пока всё узлы не будут просмотрены. Из множества путей выбривается тот, чья стоимость будет минимальной.

Продемонстрируем работу алгоритма пошагово. Допустим изначально дан граф, представленный на рисунке 2.2. Допустим, что алгоритму надо пройти из точки A в точку F.

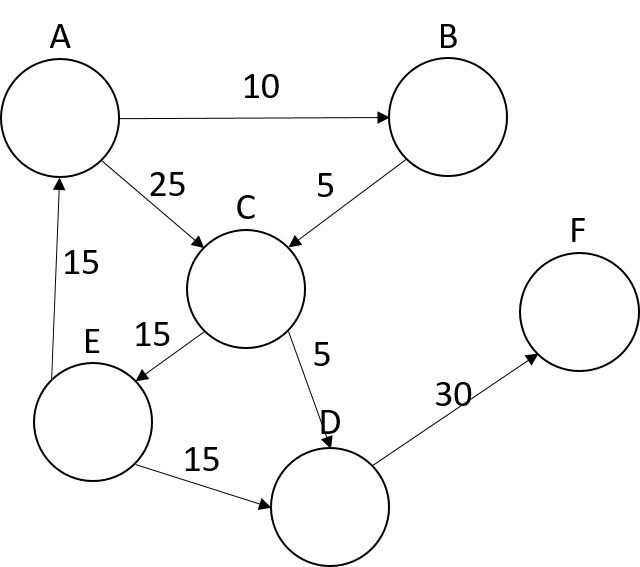


рисунок 2.2 – изначальный граф

На первом шаге, изображенном на рисунке 2.3 алгоритм, анализирует переходы в точки B и C. Поскольку переход в точку C дороже перехода в точку B алгоритм переходит в точку B

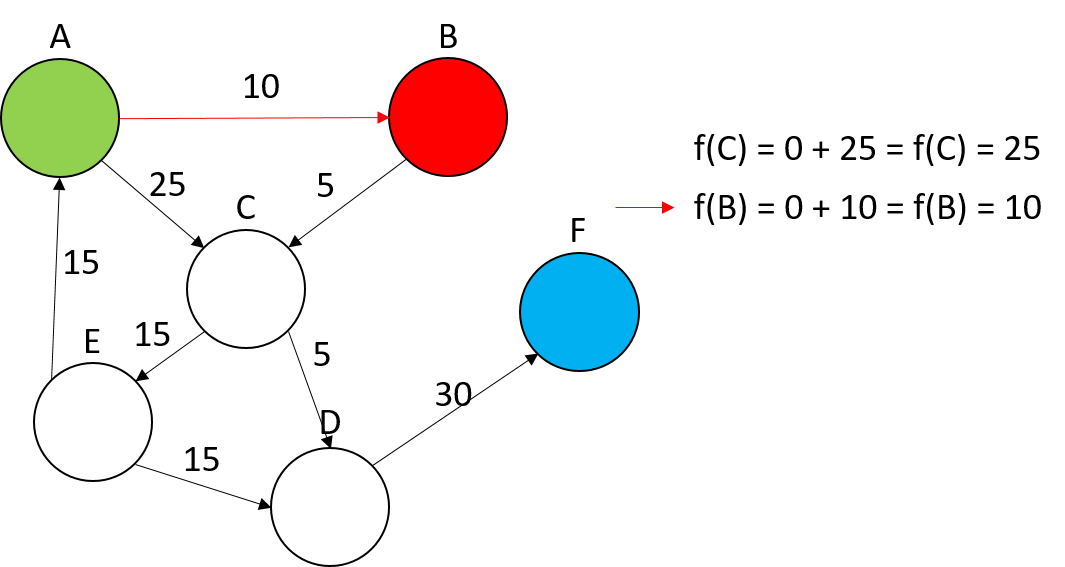


Рисунок 2.3 – переход в точку B

На следующем шаге алгоритм переходит в точку C.

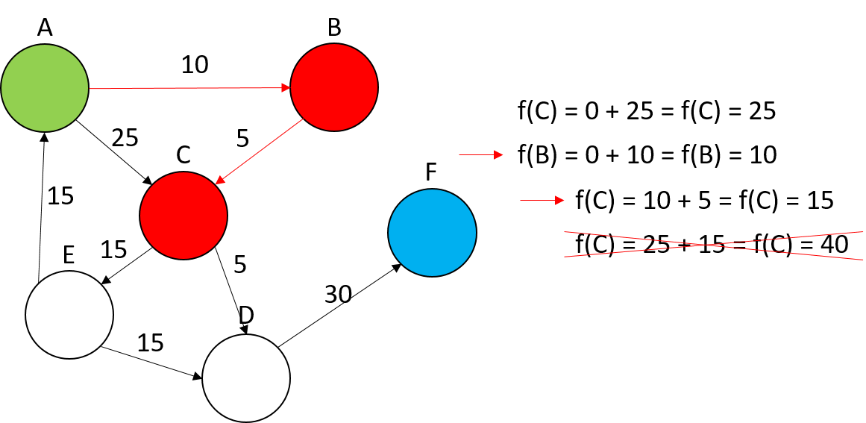


Рисунок 2.4 – переход в точку C

На третьем шаге алгоритм рассматривает переходы из точки C в точки E и D. Переход в точку E стоит больше перехода в точку D поэтому происходит переход в точку D.

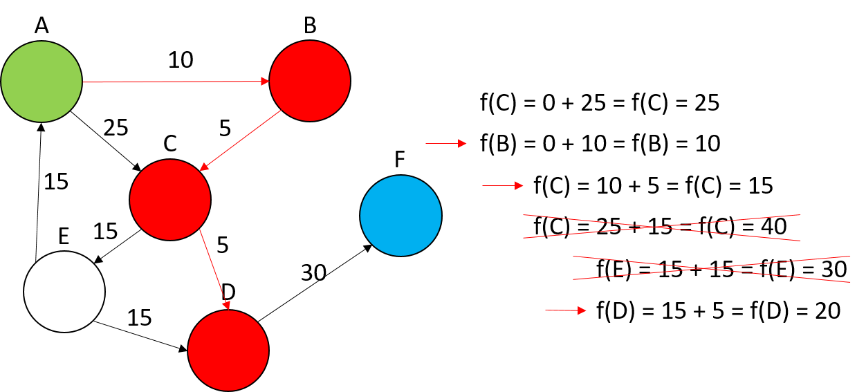


Рисунок 2.5 – переход в точку D

В последней итерации алгоритм переходит в точку F.

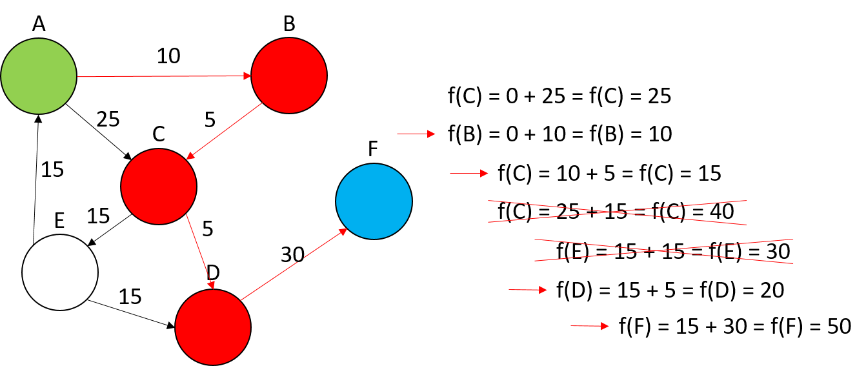


Рисунок 2.6 – переход в точку F

Алгоритм успешно построил путь: A-B-C-D-F. стоимость переходов составила: 50 ед.

* 1. Разработка структуры компонента

Чёрная сфера – термин, который используется для обозначения работы системы, внутреннее устройство которой неизвестны. По аналогии с чёрным ящиком из транспортной сферы.

На рисунке 2.6 изображена чёрная сфера разрабатываемого алгоритма

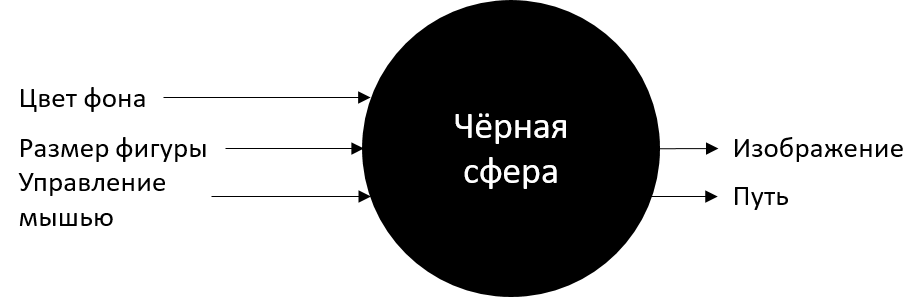


Рисунок 2.6 – Представление разрабатываемого компонента в виде модели “Чёрная сфера”

Для того, чтобы выявить структуру компонента воспользуемся методом декомпозиции.

Декомпозиция — разделение целого на части. Также декомпозиция — это научный метод, использующий структуру задачи и позволяющий заменить решение одной большой задачи решением серии меньших задач, пусть и взаимосвязанных, но более простых.

Декомпозиция, как процесс расчленения, позволяет рассматривать любую исследуемую систему как сложную, состоящую из отдельных взаимосвязанных подсистем, которые, в свою очередь, также могут быть расчленены на части. В качестве систем могут выступать не только материальные объекты, но и процессы, явления и понятия.

В задумке компонента были поставлены цели: иметь графическое отображение фигур и в зависимости от воздействий пользователя производить определённые действия будь то перемещение точек или поиск пути.

Результат первого этапа декомпозиции продемонстрирован на рисунке 2.7

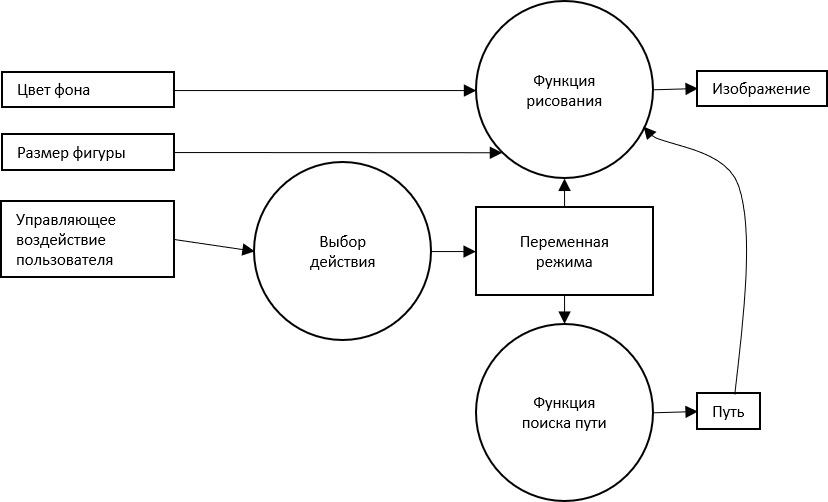


Рисунок 2.7 – Этап первой декомпозиции

На представленной схеме изображены, выходные и выходные параметры прямоугольниками, методы, реализуемые в виде процедур и функций, кругами, стрелками поясняется логика и последовательность взаимодействия свойств и методов. Слева на схеме находятся входные параметры, справа – отражены результаты работы класса в виде выходных параметров.

Чтобы получить более детальную структурную схему, произведем еще одну декомпозицию имеющейся структуры.

Результат второй декомпозиции продемонстрирован на рисунке 2.8 и плакате КП 0902.02.000000.01 ПЛ

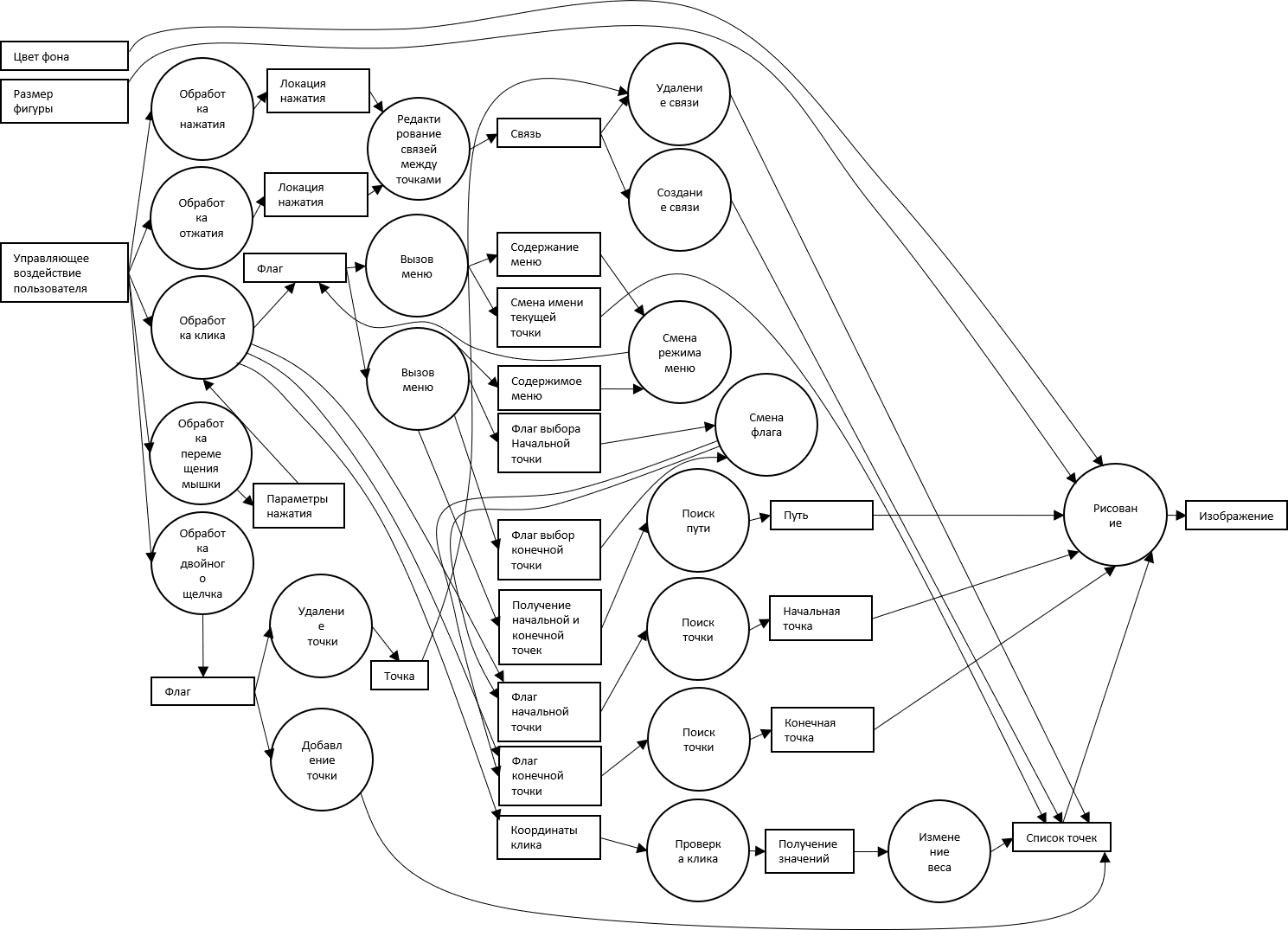


Рисунок 2.8 – Этап второй декомпозиции

В полученной окончательной структурной схеме были выявлены все нужные для реализации компонента методы, свойства и структуры для хранения данных.

* 1. Разработка графического интерфейса компонента

Реализуемый компонент будет соответствовать графической составляющей операционной системы Windows, которая была создана с идеей реализации программ в виде окон, имеющих прямоугольное представление.

Все окна, реализованные в операционных средах Windows, имеют единые инструменты взаимодействия пользователя с ними. Например, кнопки – свернуть развернуть и закрыть. Данные кнопки были впервые реализованы в Windows 95. С того времени они непрерывно сопровождали операционные среды меняясь только в дизайне. Интерфейс сред интуитивно понятен поскольку все элементы управления расположены по углам приложений.

Управление работой компонента напрямую пользователем будет осуществляться посредством контекстного меню. Меню будет вызываться при правом щелчке мыши, из которого можно будет определять, какое действие будет дальше обрабатываться. При левом щелчке мыши будет реализовано выполнение команд, выбранных в контекстном меню. Разрабатывая компонент в него закладывалась идея максимального использования манипулятора “мыши”, чтобы пользователю не приходилось отвлекаться на клавиатуру. Из-за этого в большинстве случаев для управления компонентов хватит только мыши.

Для того чтобы создать точку нужно два раза щёлкнуть левой кнопкой по полю компонента. Чтобы удалить точку надо два раза щёлкнуть левой кнопкой мыши по точке. Для перемещения точки надо: нажать левую кнопку мыши, над точкой которую надо переместить, и начать перемещать мышь. Для того чтобы переименовать точку надо: выбрать точку, это можно сделать щёлкнуть левой кнопкой мыши по точке, вызвать контекстное-меню щёлкнув правой мыши. В контекстном меню выбрать пункт “Изменить название ноды”. После этого появиться форма для ввода значения нового имени. Для смены режима работы надо: вызвать контекстное-меню в меню надо выбрать пункт “Перемещение нод”. Для того чтобы создать связь между точками надо: сменить режим меню, как говорилось выше, далее на точке “старта” опустить правую клавишу мыши. На точке “финиша” поднять правую клавишу мыши. Для удаления связи повторить процесс создания связи. Для редактирования веса между точками надо: сделать щелчок по числу между двумя точками. После щелчка появиться форма для ввода нового значения. Для выбора начальной точки для поиска надо: вызвать контекстное-меню в меню выбрать пункт “ Выбрать начальную ноду” после этого щелкнуть по точке желаемого старта. Для выбора финишной точки повторить действия по выбору начальной точки, но за одним исключением: в меню надо выбирать пункт “Выбрать конечную ноду”. Для того чтобы построить путь надо: выбрать точки старта и финиша, как описывалось выше, вызвать контекстное-меню, в котором найти пункт “Найти путь”.

Поскольку компонент разрабатывается под операционные среды Windows на форме компонент примет прямоугольное представление.

В исследовательском разделе было установлено: компонент будет разрабатываться от класса “Control” который в свою очередь имеет ряд публичных свойств, которые отвечают за: цвет заднего фона компонента, размеры компонента.

* 1. Описание входных и выходных данных компонента.

В подразделе 2.2 был выявлен ряд свойств. Некоторые свойства наследуются от родительского класса “Control” такие как BackColor, Size. Но также присутствуют и свойства, которые для реализации бизнес-логики класса

Все требующиеся для управления работой класса свойства описаны в таблице 2.9

Таблица 2.9 – Входные и выходные параметры компонента.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя свойства или события | Тип данных | Тип доступа | Метод доступа на чтение | Метод доступа на запись |
| BackColor (Цвет заднего фона) | Color | чт/зп. | Унаследован от родительского класса | Унаследован от родительского класса |
| Size (Размер компонента) | Size | чт/зп. | Унаследован от родительского класса | Унаследован от родительского класса |
| NodeSize (Размер точки) | int | чт/зп. | Стандартный механизм чтения | Модифицированная процедура set с целью контроля за диапазоном вводимых значений |
| TraversalVelocity (Стоимость перехода) | int | чт/зп. | Стандартный механизм чтения | Стандартный механизм записи |
| Incoming (Лист доступных переходов) | List<Edge> | чт. | Стандартный механизм чтения | - |
| Outgoing (Лист доступных переходов) | List<Edge> | чт. | Стандартный механизм чтения | - |
| NodeName (Название точки) | string | чт/зп. | Стандартный механизм чтения | Стандартный механизм записи |
| Position (Положение точки) | Point | чт/зп. | Стандартный механизм чтения | Стандартный механизм записи |
| Way (Путь) | List<Node> | чт. | Стандартный механизм чтения | - |

* 1. Разработка алгоритмов компонента

В подразделе 2.2 была получена структурная схема компонента. На этой схеме были выявлены методы необходимые для реализации бизнес логики. Данные методы будут прорабатываться в этом разделе.

Метод-обработчик нажатия клавиши мыши.

Входным параметром является MouseEventArgs e. В зависимости от режима работы и от нажатой клавиши будут происходить такие действия как: перемещение точки, Выбор стартовой точки, финишной точки, вызов контекстного меню, и вызов формы для ввода значений. Блок-схема алгоритма изображена на рисунке 2.10.

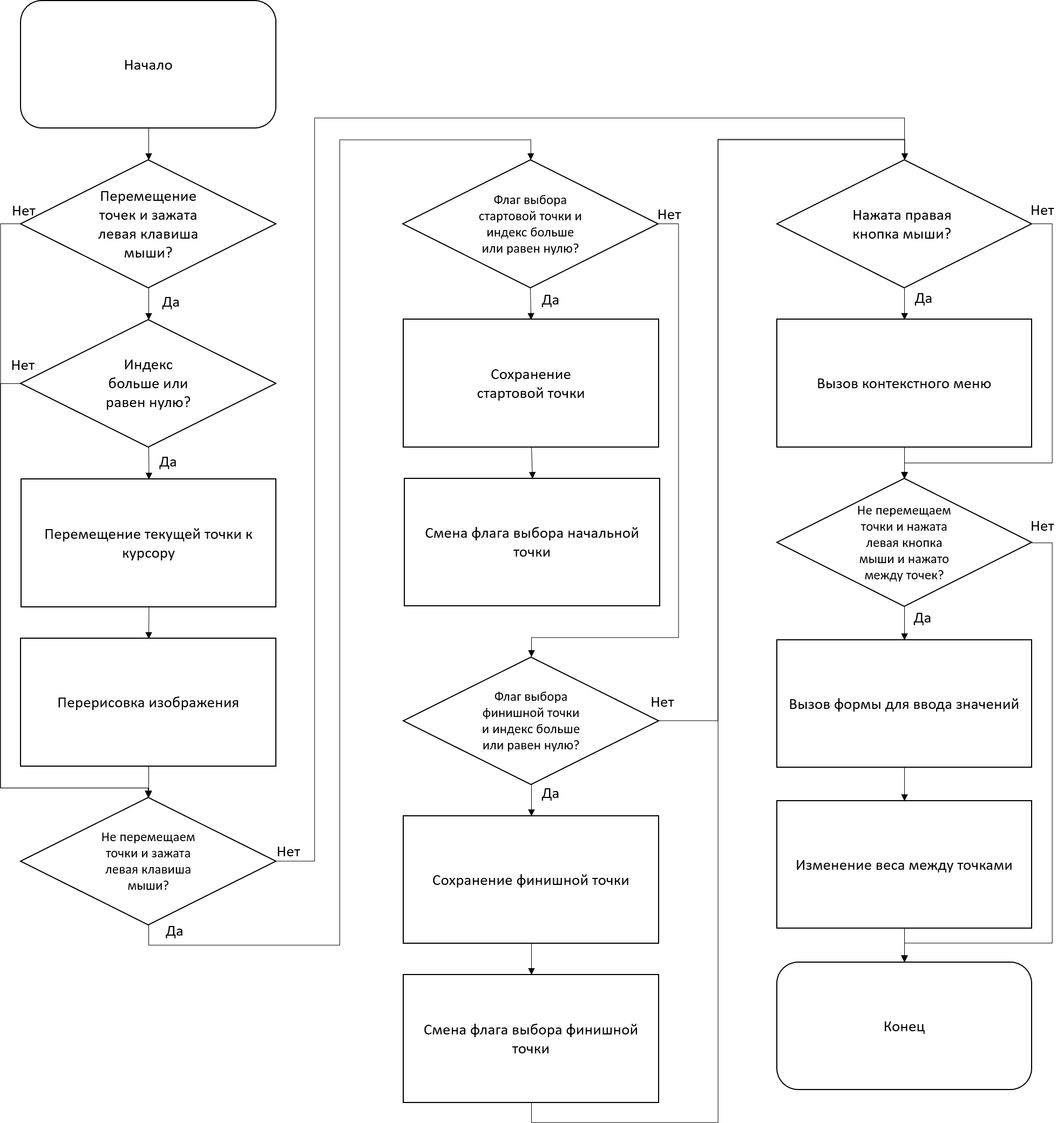


Рисунок 2.10 – Алгоритм метода нажатия кнопки мыши

Метод обработчик отжатия кнопки мыши.

Входным параметром является: MouseEventArgs e. Служит для создания и удаления связей между точками. Блок-схема алгоритма 2.11.

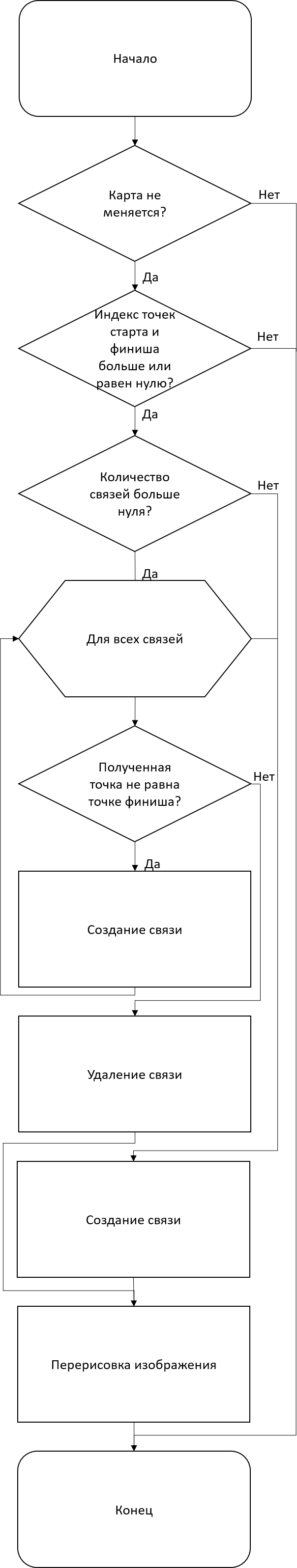


Рисунок 2.11 – Алгоритм метода поднятия кнопки мыши

Метод обработчик двойного нажатия мыши.

Входным параметром является MouseEventArgs e. Создаёт новую точку если произошло двойное нажатие по пустому полю. Если произошло двойное нажатие по точке, то удаляет её. Блок-схема алгоритма изображена на рисунке 2.12.

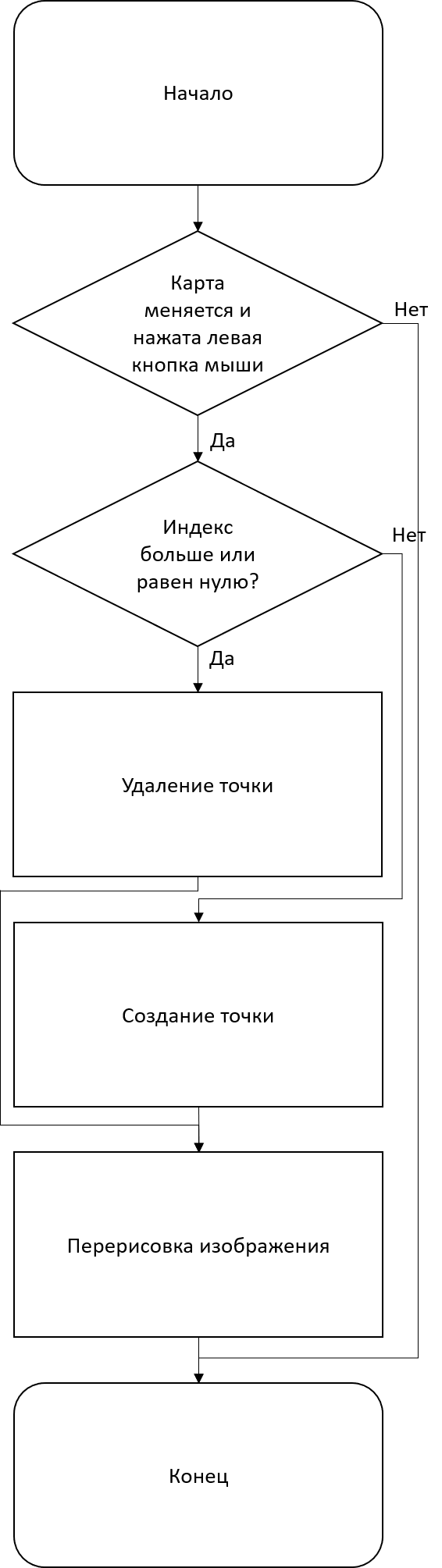


Рисунок 2.12 – Алгоритм метода двойного нажатия

Метод обработчик перемещения мыши.

Входным параметром является MouseEventArgs e. Вызывает обработчик события нажатия клавиши мыши. Блок-схема алгоритма изображена на рисунке 2.13.

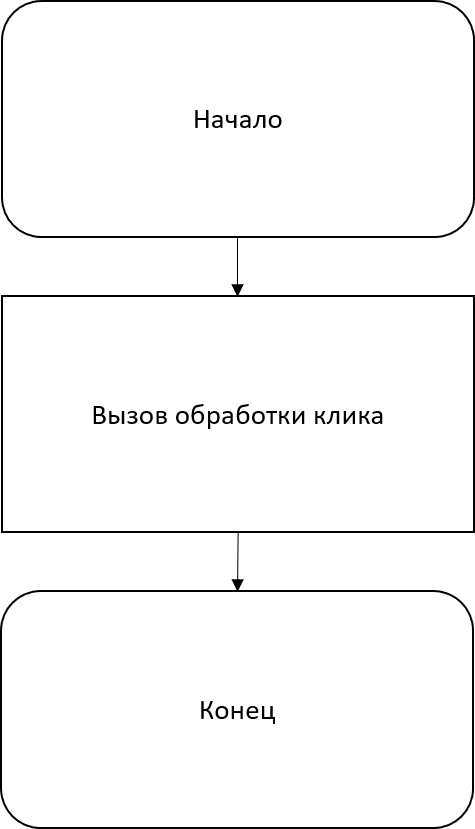


Рисунок 2.13 – Алгоритм метода перемещения мыши

Метод получения индекса.

Входным параметром является: Point pos. Служит для получения индекса точки в массиве. Блок-схема представлена на рисунке 2.14.

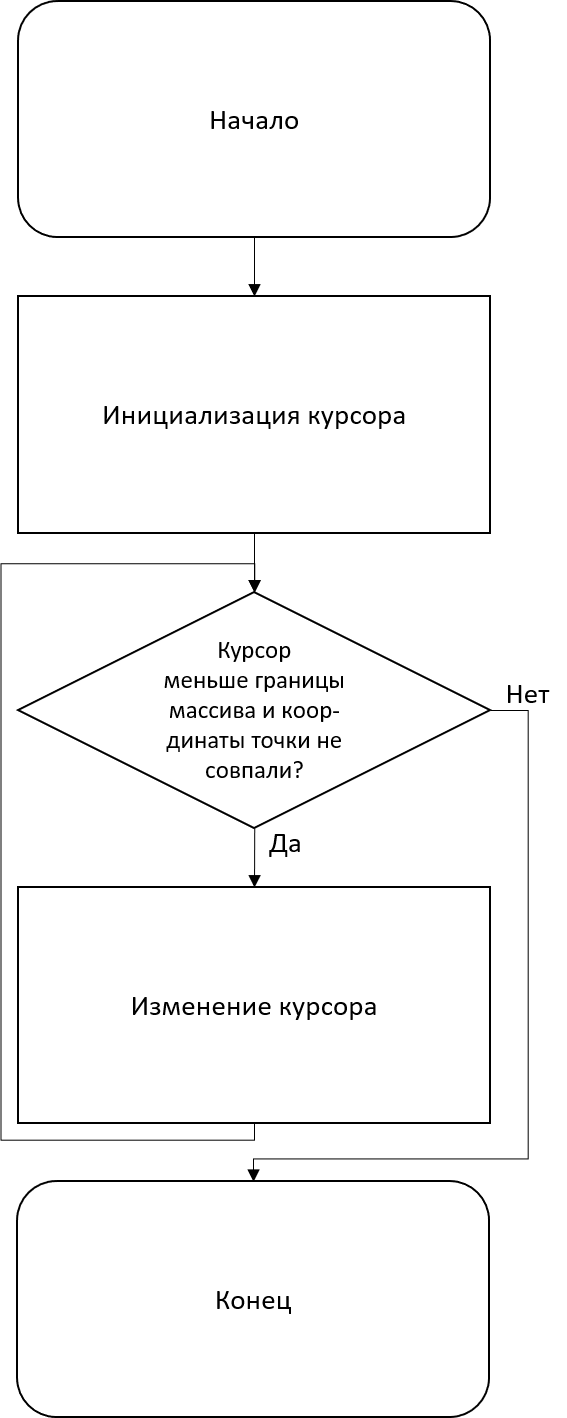


Рисунок 2.14 – Алгоритм метода получения индекса

Метод для рисования изображения.

Входным параметрам является: PaintEventArgs e. Служит для прорисовки точек и связей между ними. Блок-схема алгоритма представлена на рисунке 2.15.

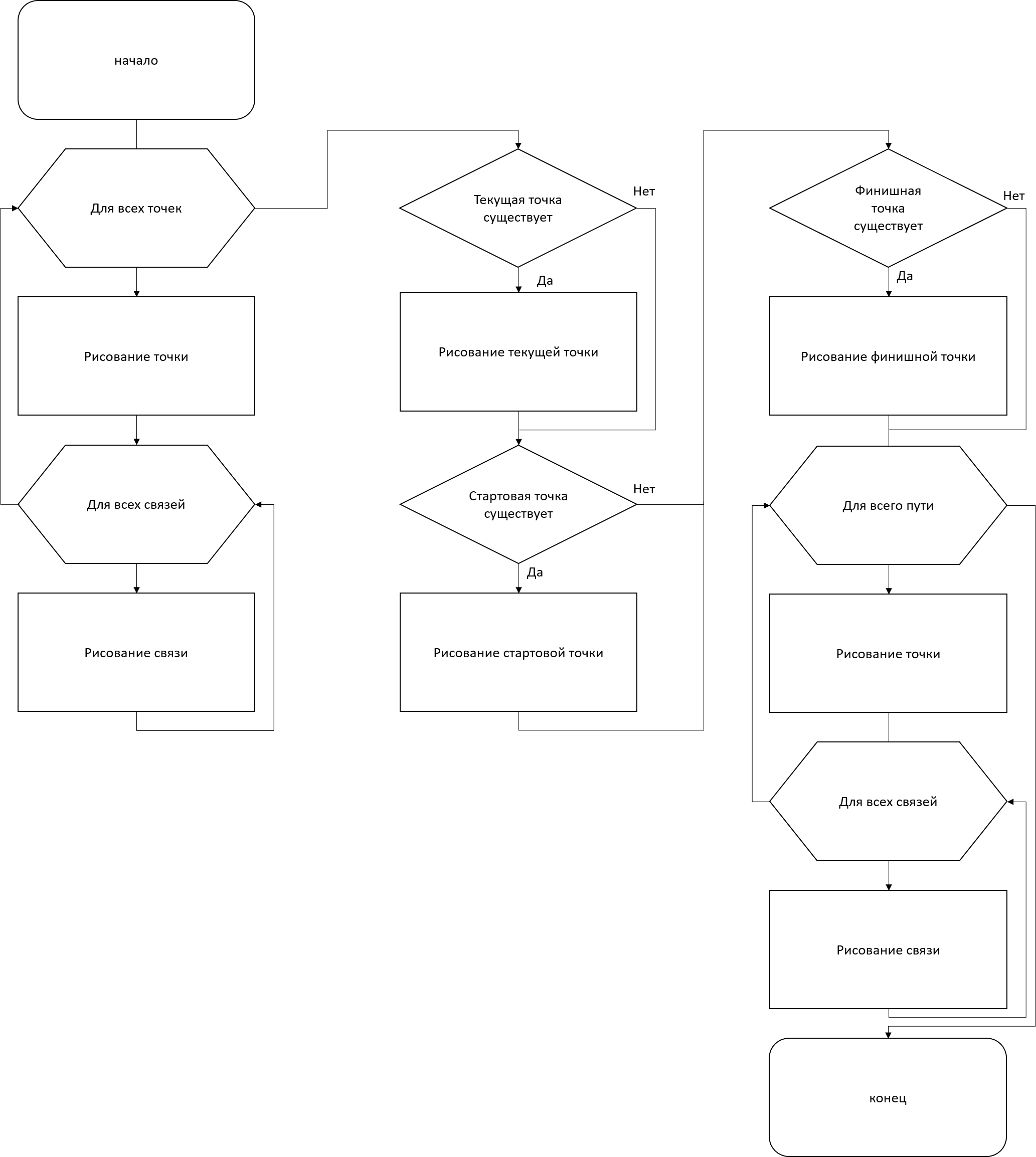


Рисунок 2.15 – Алгоритм метода рисования

Метод добавления точки в массив.

Входным параметром является: Point pos. Создаёт точку по указаны координатам. Блок-схема алгоритма представлена на рисунке 2.16.

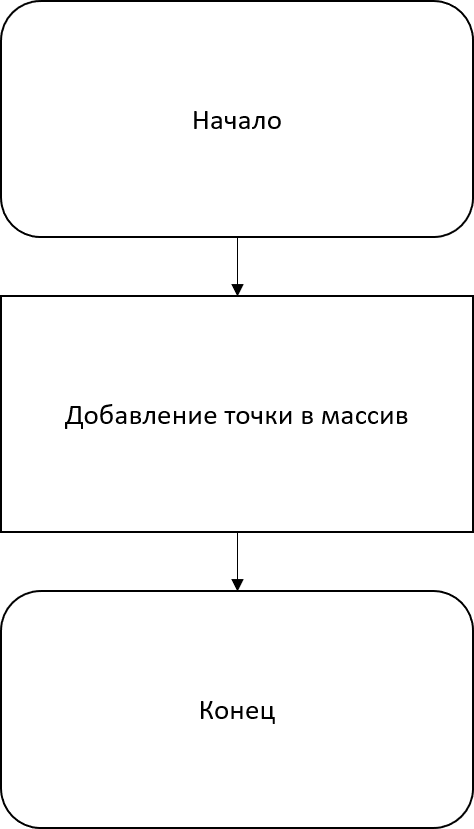


Рисунок 2.16 – Алгоритм добавления новой точки в массив

Метод-обработчик события нажатия клавиши мыши.

Входными параметрами является: MouseEventArgs e. Алгоритм сохраняет начальные координаты нажатия. Блок-схема алгоритма представлена на рисунке 2.17.



Рисунок 2.17 – Алгоритм нажатия кнопки мыши

Метод удаления точки из массива.

Входным параметром является: int index. Удаляет точку из массива по индексу. Блок-схема представлена на рисунке 2.18.

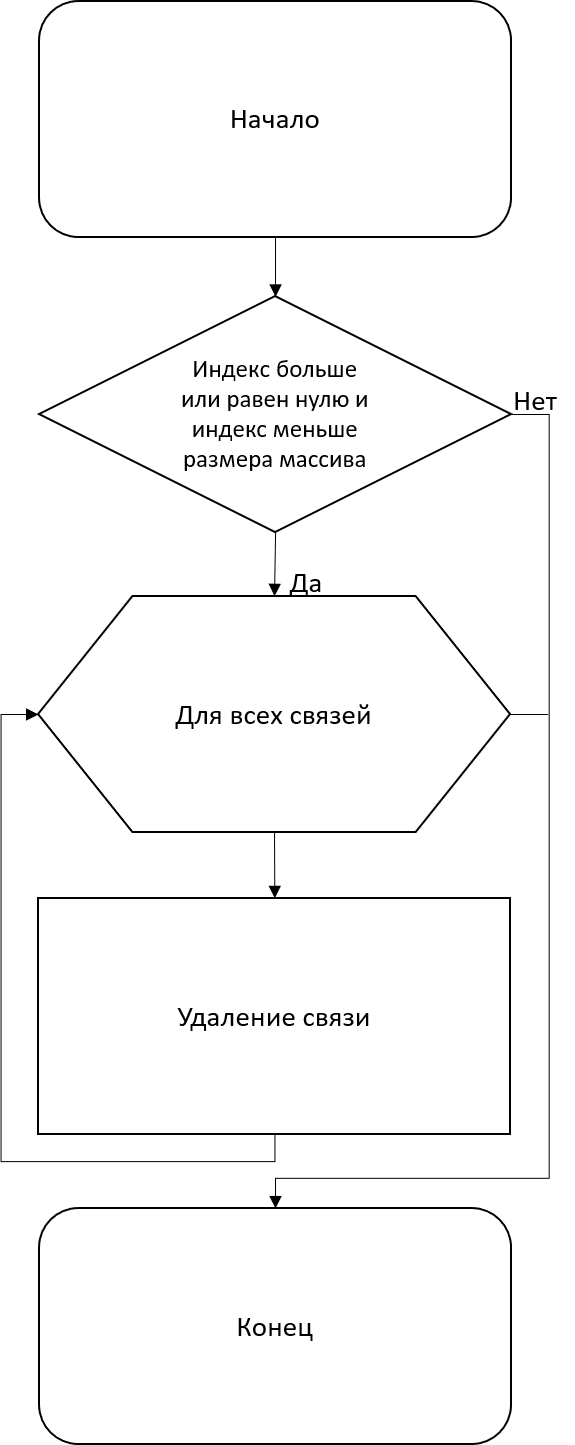


Рисунок 2.18 – Алгоритм удаления точки из массива

Метод для получения точки.

Входным параметром является: int index. Возвращает точку по индексу в массиве. Блок-схема алгоритма представлена на рисунке 2.19.

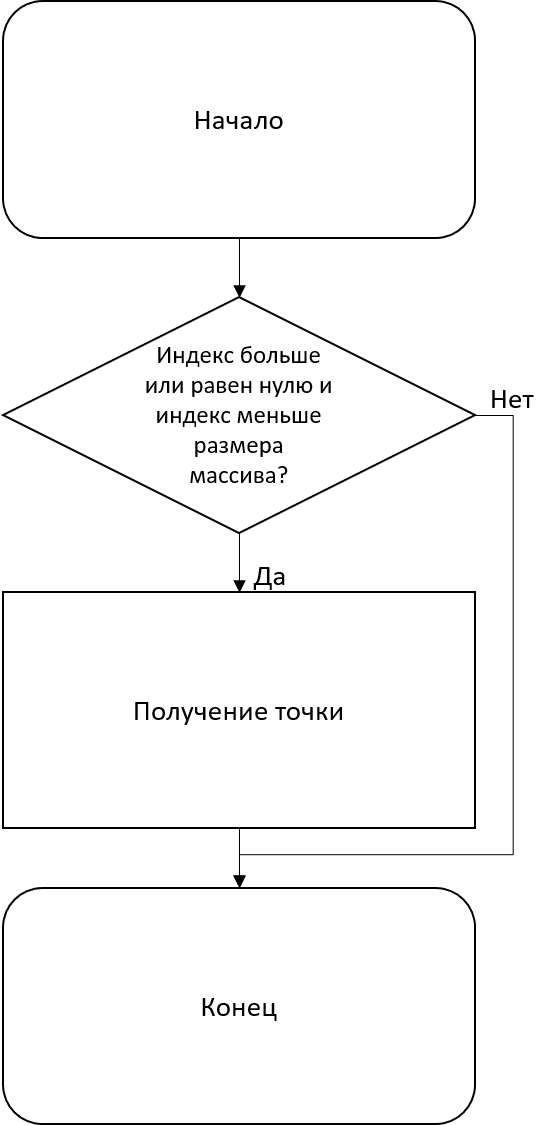


Рисунок 2.19 – Алгоритм получения точки из массива

Метод обработчик события нажатия на пункт контекстного-меню.

Входными параметрами являются: object sender, EventArgs e. Служит для смены флага выбора финишной точки. Алгоритм представлен на рисунке 2.20

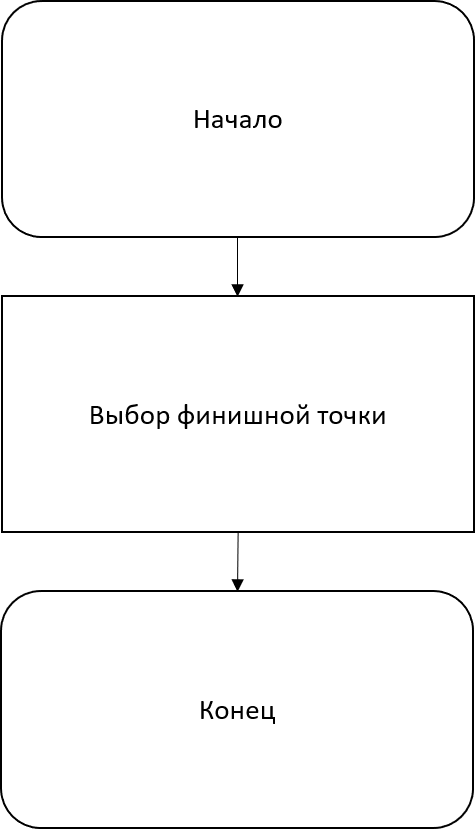


Рисунок 2.20 – Алгоритм обработки нажатия по пункту из контекстного меню

Метод обработчик события нажатия на пункт контекстного-меню.

Входными параметрами являются: object sender, EventArgs e. Служит для смены флага выбора начальной точки. Алгоритм представлен на рисунке 2.21

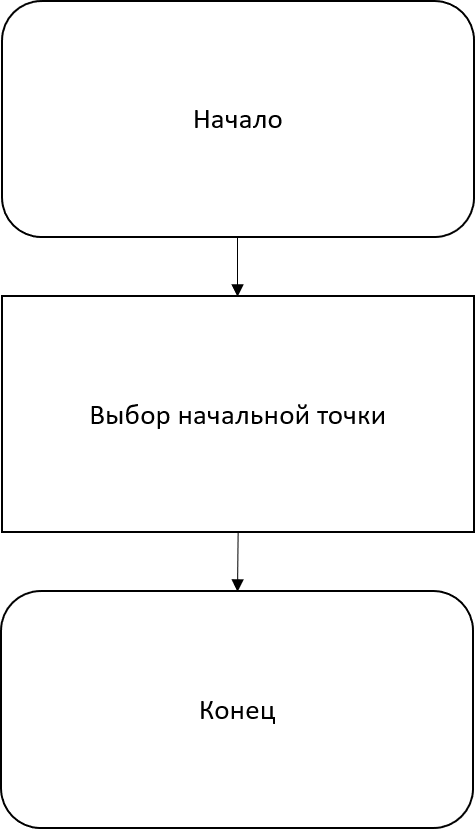


Рисунок 2.21 – Алгоритм обработки нажатия по пункту из контекстного меню

Метод определения попадания между точек.

Входной параметр: Point pos. Служит для определения попадания между двумя точками. Блок-схема алгоритма представлена на рисунке 2.22.

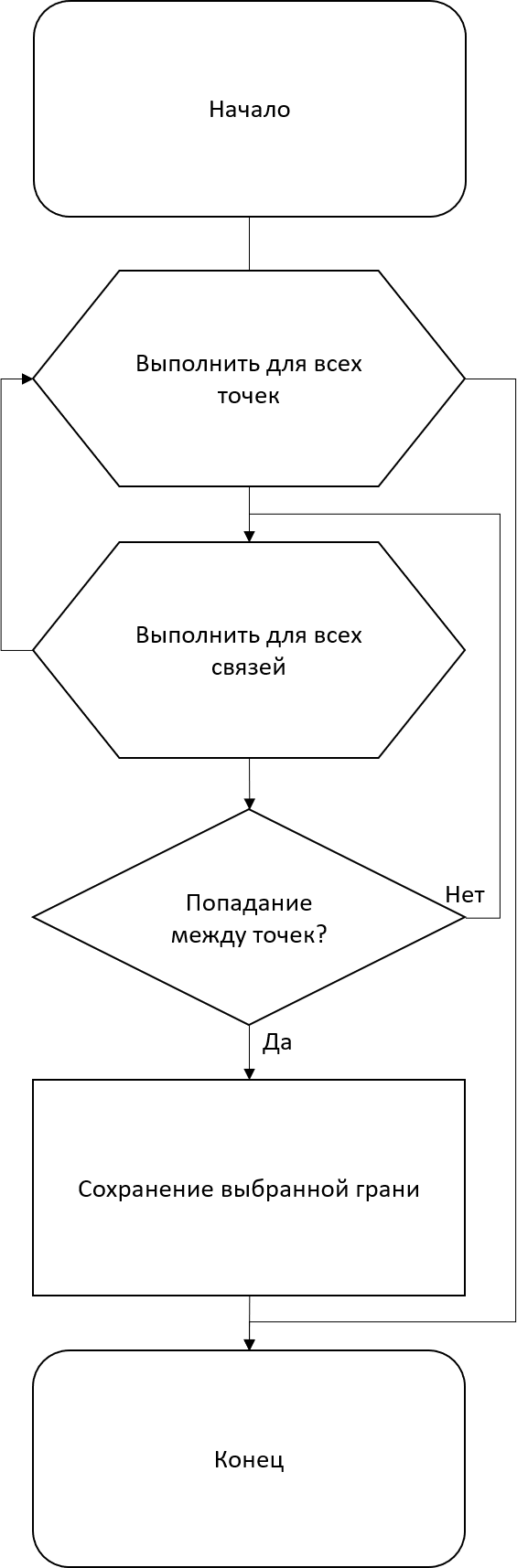


Рисунок 2.22 – Алгоритм попадания между точками

Метод для проверки попадания по точке.

Входным параметром является: Point pos. Определяет попали ли координаты по точке. Блок-схема алгоритма представлена на рисунке 2.23.

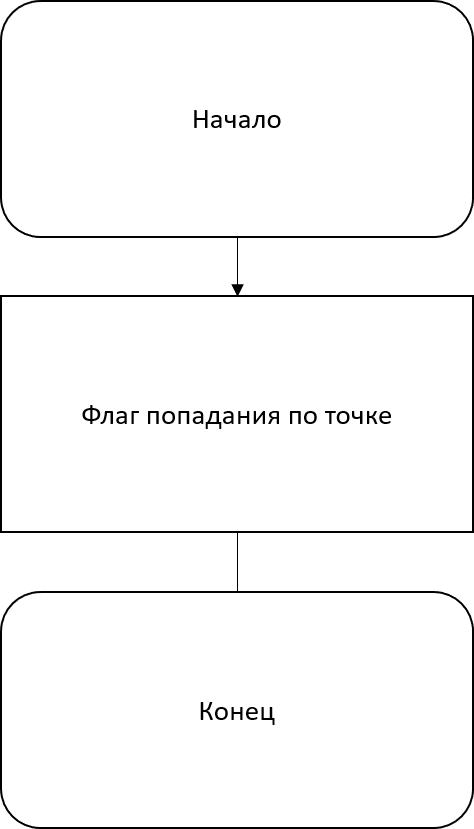


Рисунок 2.23 – Алгоритм проверки попадания по точке

Метод для сохранения пройденного пути.

Входными параметрами являются: Node node, Edge via. Алгоритм сохраняет из какой точки по какой грани прошёл он прошёл.

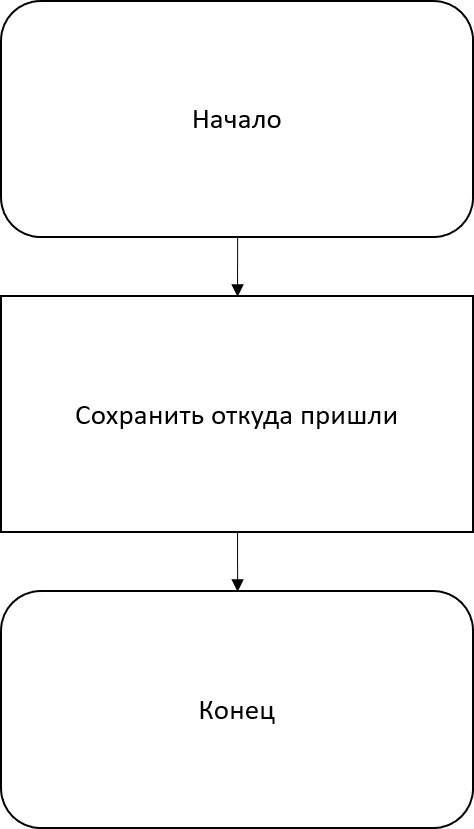


Рисунок 2.24 – Алгоритм сохранения пути

Метод обработчик события нажатия на пункт контекстного-меню.

Входными параметрами являются: object sender, EventArgs e. Служит для смены имени выбранной точки. Алгоритм представлен на рисунке 2.25

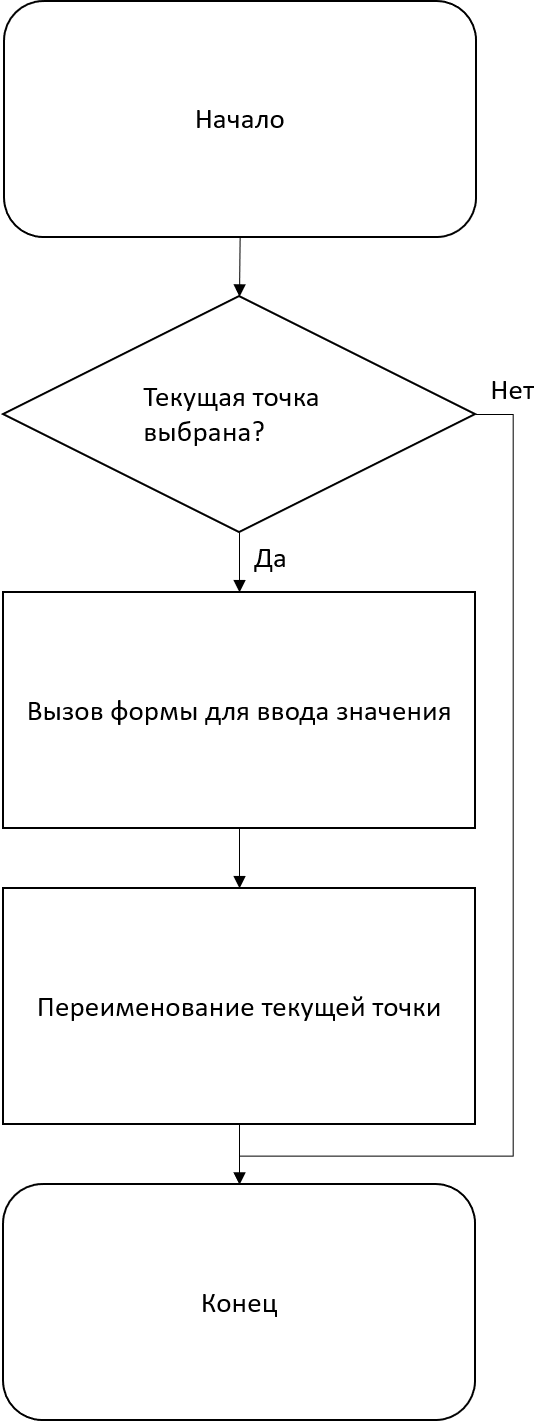


Рисунок 2.25 – Алгоритм обработки нажатия по пункту из контекстного меню

Алгоритм удаления связи.

Входным параметром является: Node node. Служит для удаления связи между точками. Алгоритм представлен на рисунке 2.26.

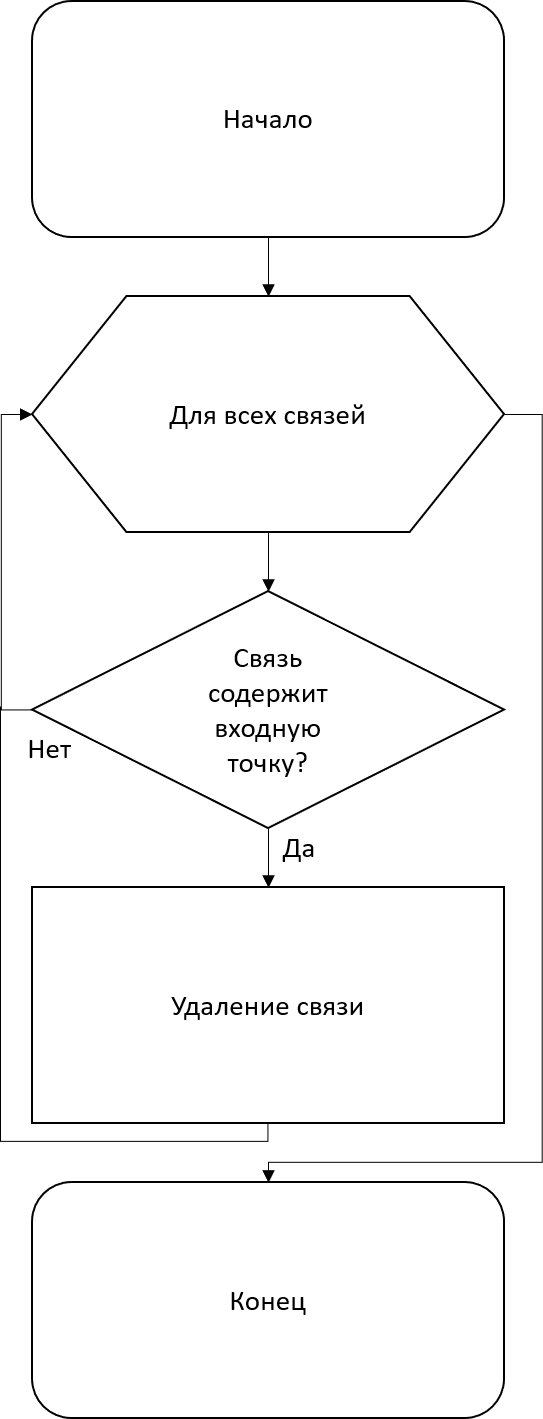


Рисунок 2.26 – Алгоритм удаления связи между точками

Алгоритм служит для создания связи между двумя связями.

Входные параметры: Node node, int traversalVelocity. Создаёт связь между точками. Блок-схема алгоритма представлена на рисунке 2.27.

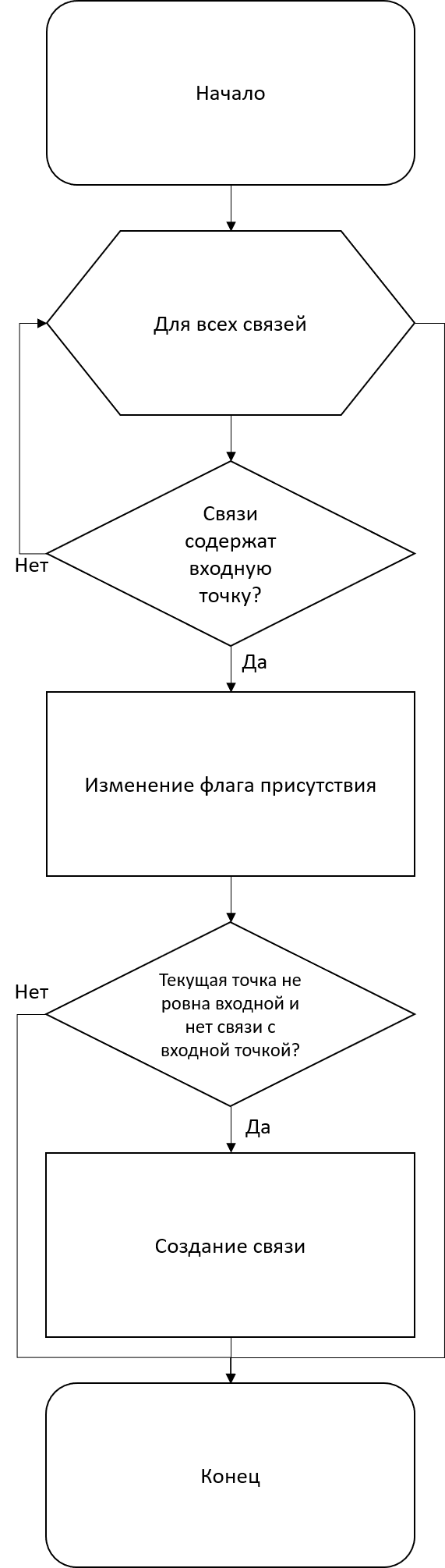


Рисунок 2.27 – Алгоритм создания связи между точками

Алгоритм построения пути.

Входные параметры: Node start, Node goal. Строит путь между двумя точками. Алгоритм представлен на рисунке 2.28.

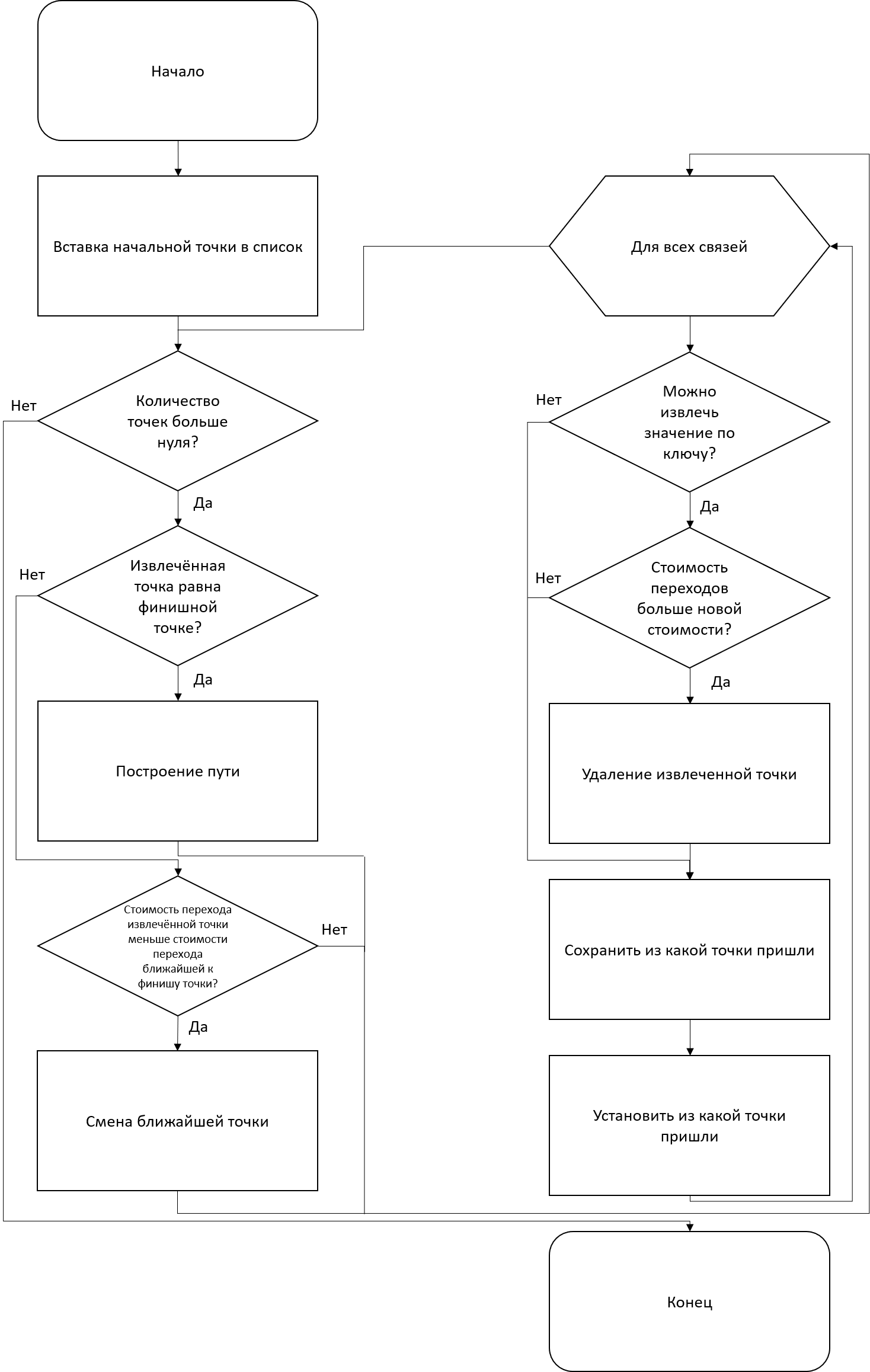


Рисунок 2.28 – Алгоритм построения пути

Метод обработчик события нажатия на пункт контекстного-меню.

Входными параметрами являются: object sender, EventArgs e. Служит для активации метода поиска пути. Алгоритм представлен на рисунке 2.29

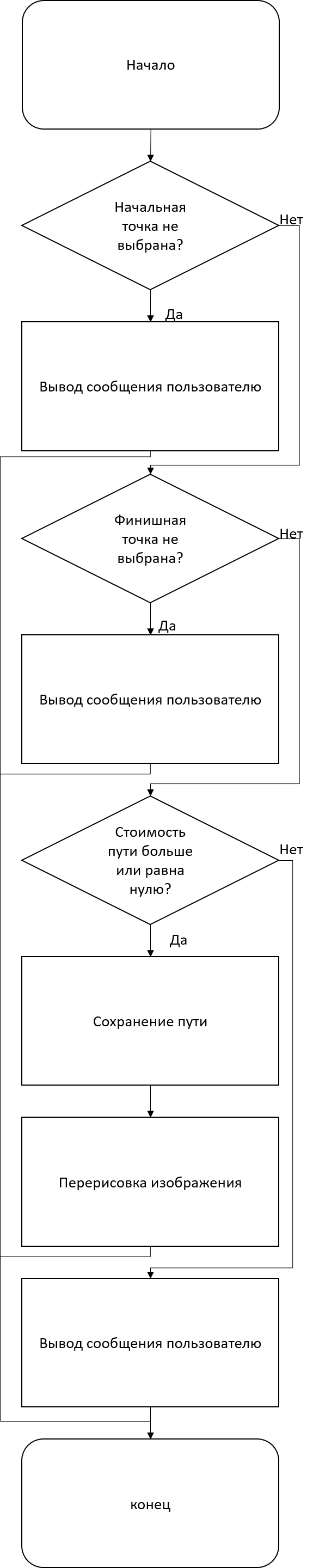


Рисунок 2.29 – Алгоритм обработки нажатия по пункту из контекстного меню

Метод обработчик события нажатия на пункт контекстного-меню.

Входными параметрами являются: object sender, EventArgs e. Служит для смены режима работы. Алгоритм представлен на рисунке 2.30

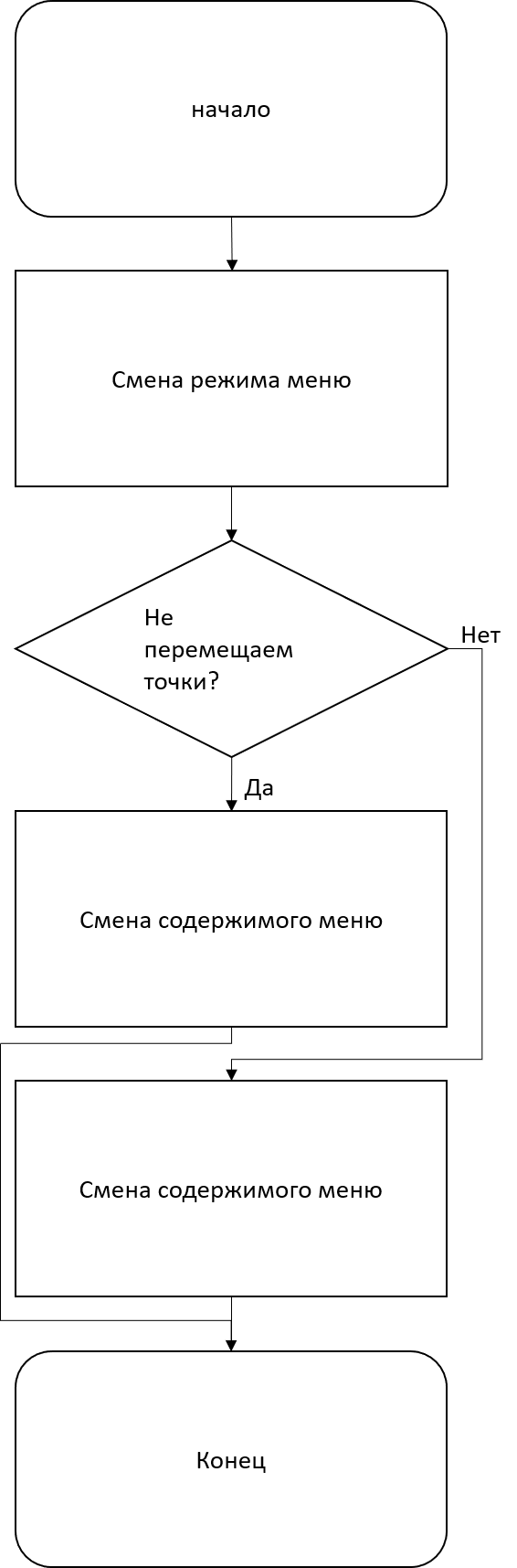


Рисунок 2.30 – Алгоритм обработки нажатия по пункту из контекстного меню

Метод восстановления пути

Входной параметр: Node start. Служит для восстановления пути из указанной точки. Блок-схема алгоритма представлена на рисунке 2.31.

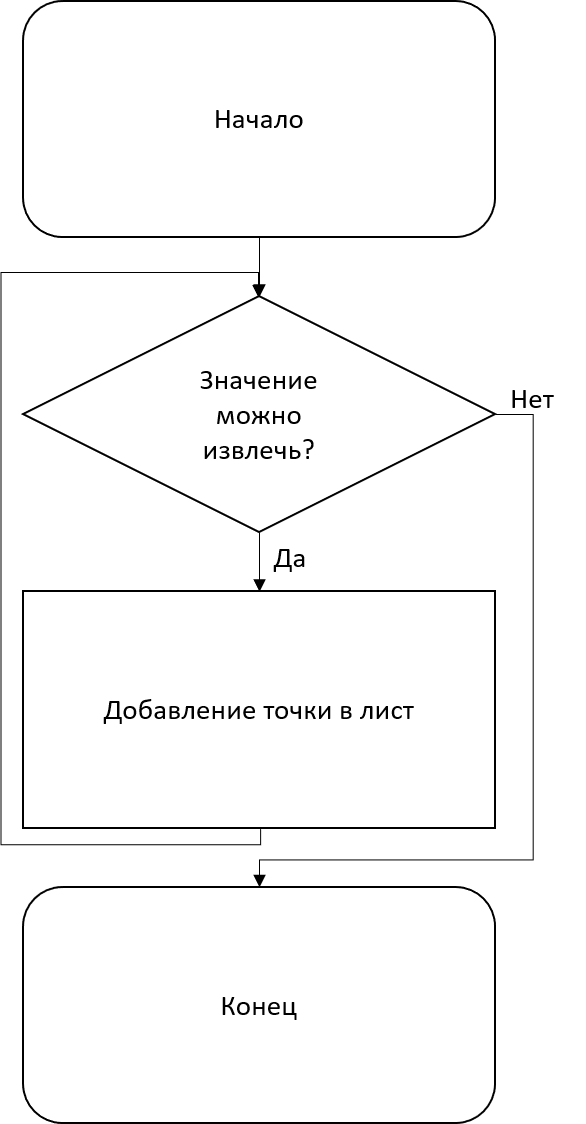


Рисунок 2.31 – Алгоритм восстановления пути

На основании блок-схем был разработан программный код. Код приведён в приложении А.

1. Технический раздел
   1. Отладка компонента

Отладка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка, приходится отлаживать программу.

Отладка программы заключается в поиске и устранении ошибок в программе. Ошибки в программе могут быть трёх типов:

* синтаксические, возникающие в результате нарушения правил написания предложений языка;
* семантические, связанные с недопустимыми значениями параметров, недопустимыми действиями над параметрами и т.д.; выявляются эти ошибки во время работы программы (обычно на этапе отладки);
* логические ошибки, связанные с неправильным использованием тех или иных алгоритмических конструкций, эти ошибки приводят к неправильным результатам при работе программы, но нарушение работы программы не вызывают.

Отладка предполагает выполнение следующих операций:

* выявление факта наличия ошибки в программе;
* определение листа нахождения ошибки (локализация);
* устранение ошибки.

Для ошибок первого типа все эти три этапа выполняются довольно просто с помощью компилятора среды, который выдает сообщение о любой обнаруженной ошибке этого типа, а курсор указывает место в тексте программы, где эта ошибка обнаружена. При этом если сообщение об ошибке, которое является довольно лаконичным, не позволяет выявить его природу, можно воспользоваться системой информационной контекстной помощи.

Наиболее сложно выявляются ошибки второго и особенно третьего типа. Выявлению ошибок второго типа часто помогает использование директив компилятора, позволяющих проверять допустимые значения тех или иных параметров (границы индексов элементов массивов, границы типа-диапазона, переполнение стека, ошибки ввода-вывода и т.д.).

Ошибки второго и третьего типа можно выявить тестированием программы с использованием встроенного отладчика. Отладчик позволяет выполнять следующие действия:

* получать значения любых параметров программы;
* модифицировать значения параметров;
* останавливать программу в заданной точке останова или в месте, соответствующем строке текста, где расположен курсор;
* осуществлять тестировку программы (выполнение программы по шагам) и т.д.

В ходе синтаксического контроля программы транслятором выявляются конструкции и сочетания символов, недопустимые с точки зрения правил их построения или написания, принятые в данном языке. Сообщения об ошибках ЭВМ выдаёт программисту, при этом вид и форма выдачи подобных сообщений зависят от вида языка и версии используемого транслятора. После устранения синтаксических ошибок проверяется логика работы программы в процессе её выполнения с конкретными исходными данными. Для этого используются специальные методы, например, в программе выбираются контрольные точки, для которых вручную рассчитываются междуточные результаты. Эти результаты сверяются со значениями, получаемыми ЭВМ в данных точках при выполнении отлаживаемой программы.

На этапе отладки для обнаружения ошибок в программе используются как текст программы, так и результаты её проверочного выполнения. Для контроля текста программы используются ручные методы контроля, которые проводятся без использования персонального компьютера, а затем машинные с применением персонального компьютера. Просмотр. Текст составленной программы и алгоритма внимательно просматривается, т.е. читается на предмет обнаружения ошибок или смысловых расхождений с текстом алгоритма, в котором производится программирование.

Первым вариантом тестирования является - Проверка. При проверке программы и алгоритма программист по тексту программы старается восстановить тот процесс, который определяет программу, после чего сверяет его с требованиями технического задания. К трудностям проверки программы, особенно логических её участков относится то, что сверять приходится не тексты, алгоритмы и программы, а вычислительный процесс. Поэтому проверка часто имеет не виртуальный характер, а мысленный. Увеличение количества этапов алгоритмизации приводит к облегчению и ускорению проверки и отладки, несмотря на увеличенный объём работы при алгоритмизации.

Вторым вариантом тестирования является - Прокрутка. Основой прокрутки является имитация выполнения программы или алгоритма на машине с целью представления о процессе или определение функций программы и последовательности её выполнения. Она позволяет проверить программу, как бы в динамике её работы, проверить элементы вычислительного процесса, а не только текст программы. Прокрутку следует применять лишь для контроля логически сложных программ или блоков. Арифметические блоки нужно проверить обычным способом. Вычислять числовые значения нужно для тех величин, от которых зависит последовательность выполнения операторов программы. Исходные данные, влияющие на логику программы должны выбираться таким образом, чтобы была минимальная прокрутка программы. В ходе прокрутки необходимо применять работу программы и для особых случаев. Прокрутка применяется не только на этапе алгоритмизации и программирования, но в ходе отладки программы на ЭВМ, для повышения контроля и локализации ошибок. Прокручивается не вся программа, а подозрительные места. Прокрутка является необходимой для программ, написанных в машинных кодах со многими внешними носителями или использующих оперативную память сложным образом.

Таким образом, в результате отладки программа должна соответствовать некоторой фиксированной совокупности правил и показателей качества, принимаемой за эталонную для данной программы. Процесс отладки программ включает:

создание совокупности тестовых эталонных значений и правил, которым должна соответствовать программа по выполняемым функциям, структуре, правилам описания, значениям исходных и соответствующих им результирующих данных;

статистическое тестирование текстов разработанных программ и данных на выполнение всех заданных правил построения и описания без исполнения объектного кода;

тестирование программы с её исполнением в объектном коде и с разными уровнями детализации: детерминированное, стохастическое и тестирование в реальном масштабе времени;

диагностику и локализацию причин отклонения результатов тестирования от заданных эталонных значений и правил;

разработку изменения программы с целью исключения причин отклонения результатов от эталонных;

реализацию корректировки программы, обеспечивающую соответствие программы заданному эталону.

Контроль правил построения и описания программ и данных предполагает точную формализацию этих правил и проверку степени их выполнения. Относительно небольшое число используемых правил описания и построения программ и данных, а также чёткая их формализация позволяют построить высокоавтоматизированные методы и средства контроля, и автоматически выявлять отклонения от этих эталонов. Для создания любых изделий применяются соответствующие технологии. При этом в понятие технология включается совокупность производственных процессов, методов и средств, предназначенных для создания определенных видов изделий с заданными показателями качества.

В результате отладки программа должна соответствовать некоторой фиксированной совокупности правил и показателей качества, принимаемых за эталонную для данной программы.

Таблица 3.1 – методы для отладки.

|  |  |  |
| --- | --- | --- |
| Имя свойства | Управляющее воздействие | Результат воздействия |
| Обработка нажатия  OnMouseDown | Нажатие левой кнопки по точке | В поле StartPoint перемещены новые координаты |
| Обработка отжатия  OnMouseUp | Отжатие левой кнопки на точке | В локальную переменную end перемещены новые координаты. |
| Обработка клика  OnMouseClick | Щелчок левой кнопки мыши | Выбранная точка переместилась на место курсора |
| Обработка клика  OnMouseClick | Щелчок правой кнопки мыши по компоненту | Вызов контекстного меню в области курсора |
| Обработка клика  OnMouseClick | Щелчок левой кнопки мыши | В свойство Node перемещена ссылка на точку по полученным координатам |
| Обработка клика  OnMouseClick | Щелчок левой кнопки мыши | Вызов формы для ввода значений |
| Обработка передвижения  OnMouseMove | Перемещение мыши с зажатой левой кнопки мыши | Выбранная точка переместилась на место курсора |
| Обработка двойного клика  OnMouseDoubleClick | Двойное нажатие левой кнопки мыши по компоненту | В список Nods добавлена новая точка |
| Обработка двойного клика  OnMouseDoubleClick | Двойное нажатие левой кнопки мыши по точке | Из списка Nods пропала удаляемая точка |

* 1. Инструкция по установке компонента

Для интеграции компонента прикладному программисту надо:

Установить библиотеку в свой проект, для этого надо: сперва вызвать обозреватель решений. По умолчанию обозреватель решений находиться в правой части экрана. Если обозреватель решений закрыт его можно вызвать из меню, которое находиться в левом верхнем углу. Во вкладке “Вид” найти элемент “Обозреватель решений” рисунок 3.2.

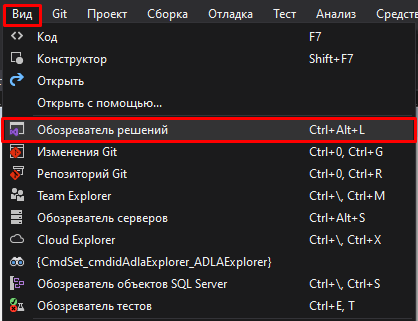


Рисунок 3.2 – Процесс выбора команды «Обозреватель решений» из меню «Вид»

После правым щелчком мыши по решению (которое находиться в обозревателе решений) вызвать контекстное меню. В появившемся списке найти пункт “Добавить” далее найти пункт “Существующий проект” рисунок 3.3.

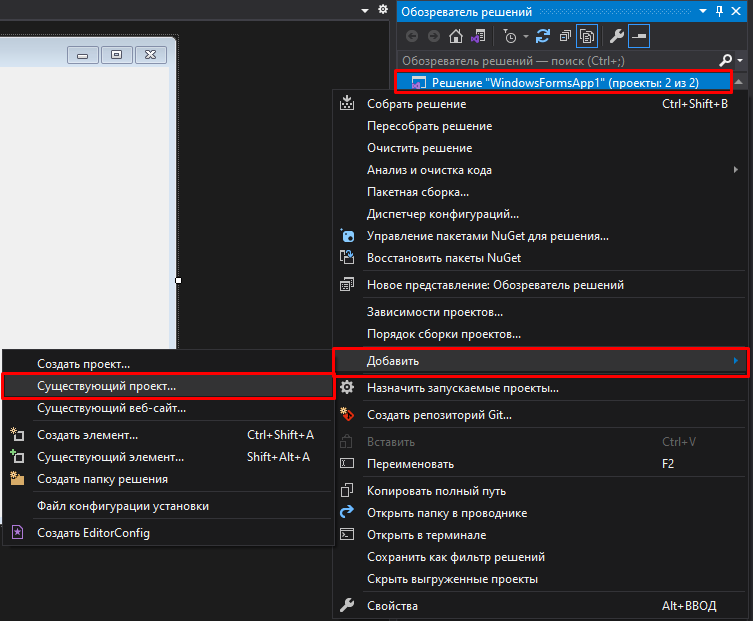


Рисунок 3.3 – Процесс добавления нового проекта

После этого среда разработки попросит указать файл проекта с расширением “.csproj”. рисунок 3.4

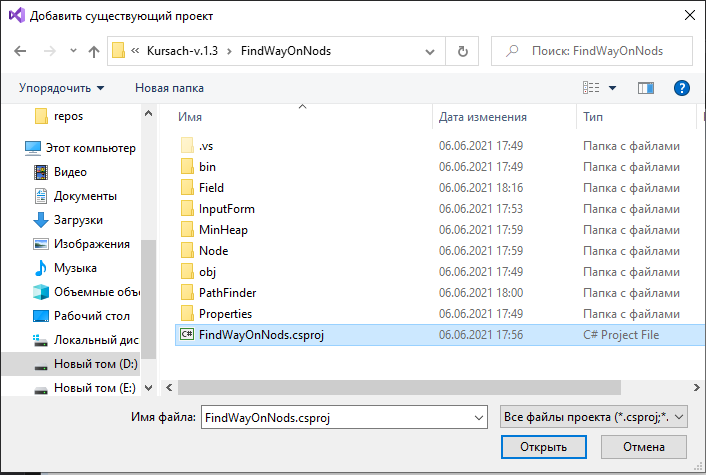


Рисунок 3.4 – Поиск папки с устанавливаемым решением

После установки проекта следует пересобрать решение, для этого: правым щелчком мыши по решению вызвать контекстное меню. В появившемся списке выбрать пункт “Пересобрать решение” как показано на рисунке 3.5. В ходе сборки не должно возникнуть ошибок.

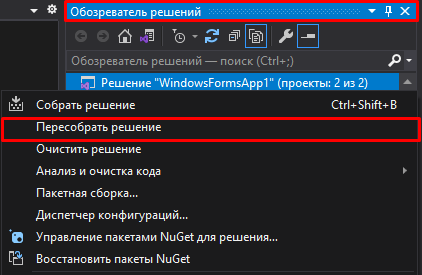


Рисунок 3.5 – Процесс пересборки решения

После успешной сборки решения можно приступать к размещению компонента на тестовом приложении. Для этого надо: перейти во вкладку с дизайнером формы. Дизайнер можно найти на открытых вкладках, он будет помечен как “название\_формы.cs [Конструктор]”, или открыв его путём двойного щелчка по форме в обозревателе решений как показано на рисунке 3.6.

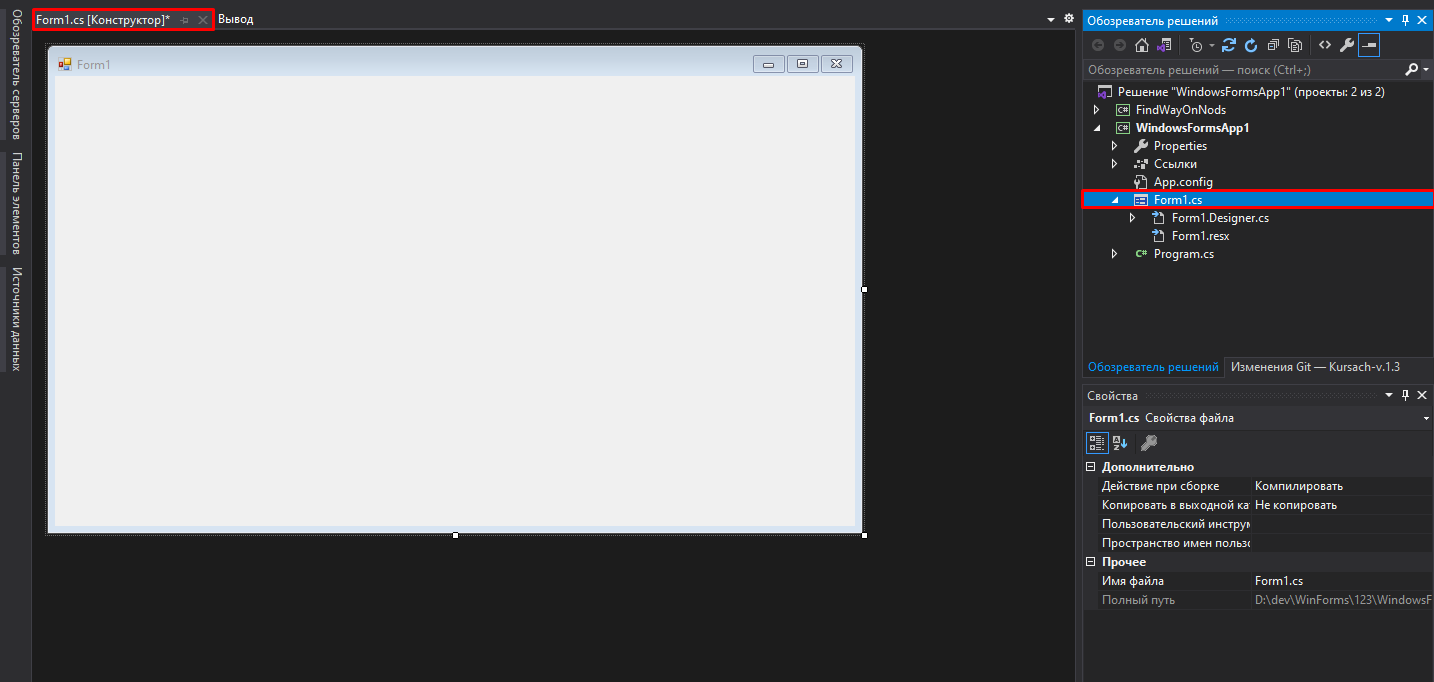


Рисунок 3.6 – Процесс открытия дизайнера

После перехода в дизайнер формы требуется открыть “Панель элементов”, которая находиться справа, или можно выполнить команду «Панель элементов» из меню «Вид» как показано на рисунке 3.7.

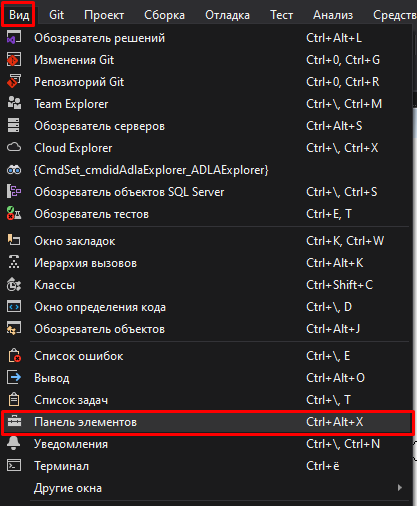


Рисунок 3.7 – Процесс выбора команды «Панель элементов» из меню «Вид»

Далее в панели элементов найти установленный компонент как показано на рисунке 3.8.

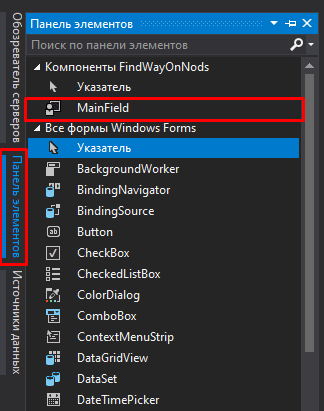


Рисунок 3.8 – Процесс поиска компонента для установки

Затем необходимо переместить данный компонент на форму как показано на рисунке 3.9. После того, как все указанные действия будут выполнены, компонент готов к работе.

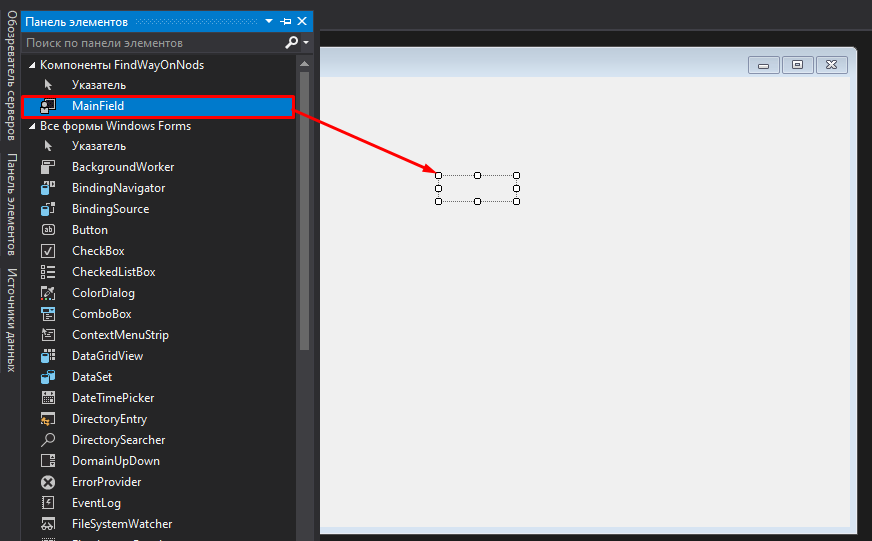


Рисунок 3.9 – Процесс установки компонента на форму

* 1. Инструкция по эксплуатации компонента

Для удобства использования разработанного визуального компонента вся инструкция по эксплуатации представлена в виде таблиц. В таблице 3.10 представлены свойства компонента.

Таблица 3.10 – свойства компонента.

|  |  |  |  |
| --- | --- | --- | --- |
| Имя свойства | Тип данных | Тип доступа | Назначение |
| End | Node | чт. | Не рекомендуется к использованию прикладным программистом. Используется для построения логики компонента |
| Start | Node | чт | Не рекомендуется к использованию прикладным программистом. Используется для построения логики компонента |
| TraversalVelocity | int | чт\зп. | Определяет стоимость пути соединённых точек. |
| Incoming | List<Edge> | чт. | Не рекомендуется к использованию прикладным программистом. Служит для связи между точками. |
| Outgoing | List<Edge> | чт. | Не рекомендуется к использованию прикладным программистом. Служит для связи между точками. |
| NodeName | string | чт\зп. | Позволяет установить имя точки. |
| NodeSize | int | чт\зп. | Позволяет установить размеры точки. |
| position | Point | чт\зп. | Позволяет устанавливать координаты точки. |
| castSoFar | int | чт. | Не рекомендуется к использованию прикладным программистом. Используется для построения логики компонента. |

В таблице 3.11 представлены события компонента.

Таблица 3.11 – события компонента.

|  |  |
| --- | --- |
| Имя события | Назначение. |
| ChangeWayEvent | Сообщает о том, что путь изменился. |

В таблице 3.12 представлены все методы компонента.

Таблица 3.12 – методы компонента.

|  |  |  |
| --- | --- | --- |
| Имя | Входные и выходные параметры | Назначение |
| AddNode | Входной параметр: pos - координаты по которым будет создана точка. | Добавление новой точки. |
| GetBetween | Входной параметр: pos - координаты по которым был произведён щелчок.  Выходной параметр: флаг, определяющий попадание. | Определение попадания между двумя точками. |
| GetIndex | Входной параметр: pos - координаты щелчка.  Выходные параметры: индекс точки в массиве. | Служит для получения индекса точки по координатам. |
| GetNodeAt | Входной параметр: index - индекс.  Выходной параметр: точка. | Получение точки из массива по индексу. |
| Connect | Входной параметр: node - точка. | Создаёт связь между двумя точками. |
| Disconnect | Входной параметр: node - точка. | Разрушает связь между двумя точками. |
| NodeClick | Входной параметр: pos - координаты щелчка мыши.  Выходной параметр: флаг, попадания по точке. | Служит для определения попадания по точке. |
| SetCameFrom | Входные параметры: node - точка из которой пришли, Edge грань по который перешли. | Служит для запоминания пути, по которому прошли. |
| ConstructPath | Входной параметр: node - точка. | Служит для восстановления пути |

1. Раздел охраны труда

Безопасность жизнедеятельности — это наука о сохранении здоровья и обеспечении безопасности человека в среде обитания. Это достигается путем выявления и идентификации опасных и вредных факторов, разработкой методов и средств защиты человека от их влияния в условиях быта и производства, методов и средств защиты людей в условиях чрезвычайных ситуаций, а также мер по ликвидации последствий таких ситуаций.

Охрана здоровья трудящихся, обеспечение безопасности условий труда, ликвидация профессиональных заболеваний и производственного травматизма составляет одну из главных забот человеческого общества. Обращается внимание на необходимость широкого применения прогрессивных форм научной организации труда, сведения к минимуму ручного, малоквалифицированного труда, создания обстановки, исключающей профессиональные заболевания и производственный травматизм. На рабочем месте должны быть предусмотрены меры защиты от возможного воздействия опасных и вредных факторов производства. Уровни этих факторов не должны превышать предельных значений, оговоренных правовыми, техническими и санитарно-техническими нормами. Эти нормативные документы обязывают к созданию на рабочем месте условий труда, при которых влияние опасных и вредных факторов на работающих либо устранено совсем, либо находится в допустимых пределах. Главными элементами рабочего места инженера-программиста являются стол и кресло. Положение сидя является основным рабочим положением.

Рабочая поза сидя вызывает минимальное утомление инженера-программиста. Рациональная планировка рабочего места предусматривает четкий порядок и постоянство размещения предметов, средств труда и документации. То, что требуется для выполнения работ чаще, расположено в зоне легкой досягаемости рабочего пространства.

Оптимальное размещение предметов труда в зонах досягаемости:

* дисплей размещается в зоне максимальной досягаемости. (в центре);
* системный блок размещается в предусмотренной нише стола;
* клавиатура перед собой для удобной работы;
* «мышь» - в зоне в справа от клавиатуры;
* сканер располагается слева от дисплея;
* принтер находится справа от дисплея.

Схема размещения основных периферийных составляющих ПК продемонстрированно на рисунке 4.1, где: 1 – сканер; 2 – монитор; 3 – принтер; 4 – поверхность рабочего стола; 5 – клавиатура; 6 – манипулятор типа «мышь».

.

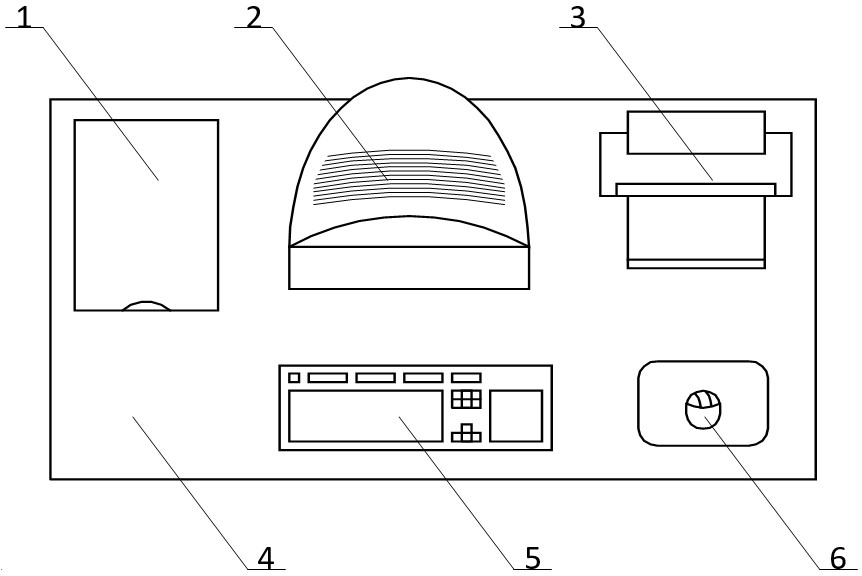


Рисунок 4.1 – Схема размещения основных периферийных устройств.

Окраска помещений и мебели должна способствовать созданию благоприятных условий для зрительного восприятия, хорошего настроения. Правильно спроектированное и выполненное производственное освещение улучшает условия зрительной работы, снижает утомляемость, способствует повышению производительности труда, благотворно влияет на производственную среду, оказывая положительное психологическое воздействие на работающего, повышает безопасность труда и снижает травматизм. Недостаточность освещения приводит к напряжению зрения, ослабляет внимание, приводит к наступлению преждевременной утомленности. Чрезмерно яркое освещение вызывает ослепление, раздражение и резь в глазах. Неправильное направление света на рабочем месте может создавать резкие тени, блики, дезориентировать работающего. Все эти причины могут привести к несчастному случаю или профзаболеваниям, поэтому столь важен правильный расчет освещенности. Кроме того все поле зрения должно быть освещено достаточно равномерно – это основное гигиеническое требование. Иными словами, степень освещения помещения и яркость экрана компьютера должны быть примерно одинаковыми, т.к. яркий свет в районе периферийного зрения значительно увеличивает напряженность глаз и, как следствие, приводит к их быстрой утомляемости.

Также для обеспечения комфортных условий используются как организационные методы (рациональная организация проведения работ в зависимости от времени года и суток, чередование труда и отдыха), так и технические средства (вентиляция, кондиционирование воздуха, отопительная система).

Шум ухудшает условия труда оказывая вредное действие на организм человека. Работающие в условиях длительного шумового воздействия испытывают раздражительность, головные боли, головокружение, снижение памяти, повышенную утомляемость, понижение аппетита, боли в ушах и т. д. Такие нарушения в работе ряда органов и систем организма человека могут вызвать негативные изменения в эмоциональном состоянии человека вплоть до стрессовых. Под воздействием шума снижается концентрация внимания, нарушаются физиологические функции, появляется усталость в связи с повышенными энергетическими затратами и нервно-психическим напряжением, ухудшается речевая коммутация. Все это снижает работоспособность человека и его производительность, качество и безопасность труда.

Режим труда. Как уже было неоднократно отмечено, при работе с персональным компьютером очень важную роль играет соблюдение правильного режима труда и отдыха. В противном случае у персонала отмечаются значительное напряжение зрительного аппарата с появлением жалоб на неудовлетворенность работой, головные боли, раздражительность, нарушение сна, усталость и болезненные ощущения в глазах, в пояснице, в области шеи и руках. В таблице 11.2 представлены сведения о регламентированных перерывах, которые необходимо делать при работе на компьютере.

Таблица 4.2 – Описание событий с примерами использования.

|  |  |
| --- | --- |
| График работы | Время, мин |
| Работа с ПЭВМ | 90 |
| Регламентированный перерыв | 25 |
| Работа с ПЭВМ | 95 |
| Обеденный перерыв | 60 |
| Работа с ПЭВМ | 90 |
| Регламентированный перерыв | 25 |
| Работа с ПЭВМ | 95 |
| Итого |  |
| Работа | 370 |
| Отдых | 110 |

При работе с вычислительной техникой могут возникнуть различные чрезвычайные ситуации. Наиболее вероятная чрезвычайная ситуация которая может возникнуть при работе с ПЭВМ – пожар, так как в современных ЭВМ очень высокая плотность размещения элементов электронных схем, в непосредственной близости друг от друга располагаются соединительные провода и кабели, при протекании по ним электрического тока выделяется значительное количество теплоты, при этом возможно оплавление изоляции и возникновение возгорания. Возникновение других видов ЧС – маловероятно.

При работе компьютерной техники выделяется много тепла, что может привести к пожароопасной ситуации. Источниками зажигания так же могут служить приборы, применяемые для технического обслуживания, устройства электропитания, кондиционеры воздуха. Серьёзную опасность представляют различные электроизоляционные материалы, используемые для защиты от механических воздействий отдельных радиодеталей. В связи с этим, участки, на которых используется компьютерная техника, по пожарной опасности относятся к категории пожароопасных «В». В помещениях с компьютерной техникой, недопустимо применение воды и пены ввиду опасности повреждения или полного выхода из строя дорогостоящего электронного оборудования. Для тушения пожаров необходимо применять углекислотные и порошковые огнетушители, которые обладают высокой скоростью тушения, большим временем действия, возможностью тушения электроустановок, высокой эффективностью борьбы с огнем. Воду разрешено применять только во вспомогательных помещениях. Исходя из норм пожарной безопасности, для машинного зала площадью до 100 м2 требуются следующие первичные средства пожаротушения:

* один углекислотный огнетушитель типа ОУ-5 или ОУ-8, с помощью которого можно тушить загорания различных материалов и установок напряжением до 1000 В;
* один порошковый огнетушитель.

Также помещение офиса должно быть оборудовано пожарными извещателями, которые позволяют оповестить дежурный персонал о пожаре.

Заключение

В ходе выполнения курсового проекта был получен рабочий программный продукт реализующий поиск минимально возможного пути, так же опыт написания технической документации.

После анализа полученного компонента были выявлены положительные стороны: компонент прост в управлении для конечного пользователя. Разработаны удобные методы взаимодействия с компонентом для дальнейшей разработки.

Разработанный компонент позволит прикладным программистам использовать реализованные методы, не тратя время на написание их реализации.

Как говорилось выше: компонент сможет рассчитывать минимально возможный путь до цели, что в современном мире может найти применение в таких приложениях как:

* карты;
* маршрутизаторы;
* игры, в которых есть необходимость поиска минимального пути.

В дальнейшем данный проект можно модифицировать, дополнив его различными алгоритмами поиска пути.

ССЫЛКИ НА РЕСУРСЫ

1. <https://softwareengineering.stackexchange.com/questions/29743/delphi-vs-c-for-gui-programming>;
2. https://habr.com/ru/post/474518/
3. <https://vscode.ru/tag/grafyi>;
4. [https://ru.wikipedia.org/wiki/A\*](https://ru.wikipedia.org/wiki/A*);
5. <https://ru.wikipedia.org/wiki/Декомпозиция>;
6. <https://ru.wikipedia.org/wiki/Отладка_программы>;
7. https://github.com/roy-t/AStar/blob/master/Roy-T.AStar/Collections/MinHeap.cs
8. https://vuzlit.ru/87295/trebovaniya\_rabochemu\_mestu\_programmista;
9. <https://gigabaza.ru/doc/65461.html>;
10. <https://works.doklad.ru/view/0iWwVHxu6VQ/all.html>;

**Приложение А**

Код класса MainField

using FindWayOnNods.Node;

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Windows.Forms;

namespace FindWayOnNods.Field

{

public class MainField : Control

{

private Node.Node Node { get; set; } = null;

private Node.Node StartNode { get; set; } = null;

private Node.Node EndNode { get; set; } = null;

private List<Node.Node> Nods = new List<Node.Node>();

private Point StartPoint;

private ContextMenuStrip contextMenuStrip = new ContextMenuStrip();

private ToolStripMenuItem ChangeMapItem = new ToolStripMenuItem();

private ToolStripMenuItem SearchWayItem = new ToolStripMenuItem();

private ToolStripMenuItem ChangeNameNodeItem = new ToolStripMenuItem();

private ToolStripMenuItem StartNodeItem = new ToolStripMenuItem();

private bool StartNodeFlag;

private ToolStripMenuItem EndNodeItem = new ToolStripMenuItem();

private bool EndNodeFlag;

private List<Node.Node> Way { get; set; } = new List<Node.Node>();

private Edge Edge { get; set; } = null;

public MainField()

{

ChangeMapItem.Checked = true;

ChangeMapItem.Text = "Перемещение нод";

ChangeMapItem.Click += ChangeMapItemClick;

ChangeNameNodeItem.Text = "Изменить название ноды";

ChangeNameNodeItem.Click += ChangeNodeName;

StartNodeItem.Text = "Выбрать начальную ноду";

StartNodeItem.Click += StartNodeItemClick;

EndNodeItem.Text = "Выбрать конечную ноду";

EndNodeItem.Click += EndNodeItemClick;

SearchWayItem.Text = "Найти путь";

SearchWayItem.Click += SearchWayItemClick;

contextMenuStrip.Items.Add(ChangeMapItem);

if (!ChangeMapItem.Checked)

{

contextMenuStrip.Items.Remove(ChangeNameNodeItem);

contextMenuStrip.Items.Add(StartNodeItem);

contextMenuStrip.Items.Add(EndNodeItem);

contextMenuStrip.Items.Add(SearchWayItem);

}

else

{

contextMenuStrip.Items.Add(ChangeNameNodeItem);

contextMenuStrip.Items.Remove(StartNodeItem);

contextMenuStrip.Items.Remove(EndNodeItem);

contextMenuStrip.Items.Remove(SearchWayItem);

}

#region Тестовые данные

var n1 = new Node.Node(new Point(200, 150));

n1.NodeName = "Node1";

var n2 = new Node.Node(new Point(200, 200));

n2.NodeName = "Node2";

var n3 = new Node.Node(new Point(200, 300));

n3.NodeName = "Node3";

var n4 = new Node.Node(new Point(200, 400));

n4.NodeName = "Node4";

var n5 = new Node.Node(new Point(400, 400));

n5.NodeName = "Node5";

var n6 = new Node.Node(new Point(500, 300));

var n = new Node.Node(new Point(100, 150));

n.NodeName = "Node";

n.Connect(n1, 10);

n1.Connect(n4, 20);

Nods.Add(n);

Nods.Add(n1);

Nods.Add(n2);

Nods.Add(n3);

Nods.Add(n4);

Nods.Add(n5);

Nods.Add(n6);

#endregion

//оптимизация компонента

SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.OptimizedDoubleBuffer | ControlStyles.ResizeRedraw | ControlStyles.UserPaint, true);

//включение 2-ой буфферизации

DoubleBuffered = true;

}

//событие поиска пути

private void SearchWayItemClick(object sender, EventArgs e)

{

//проверка на пустоту начальной вершины

if (StartNode == null)

{

MessageBox.Show("Выберете стартовую ноду!", "Ошибка!");

}

else

{

//проверка на пустоту конечной вершины

if (EndNode == null)

{

MessageBox.Show("Выберете конечную ноду!", "Ошибка!");

}

else

{

var path = new PathFinder.PathFinder();

var p = path.FindPath(StartNode, EndNode);

if (p.TraversalVelocity != -1)

{

Way = p.ListWay;

Invalidate();

}

else

MessageBox.Show("Пути нет!");

}

}

}

private void EndNodeItemClick(object sender, EventArgs e)

{

EndNodeFlag = true;

}

private void StartNodeItemClick(object sender, EventArgs e)

{

StartNodeFlag = true;

}

private void ChangeNodeName(object sender, EventArgs e)

{

if (Node != null)

using (InputForm.InputBox form = new InputForm.InputBox(Node.NodeName, PointToScreen(Node.position)))

{

if (form.ShowDialog() == DialogResult.OK)

{

Node.NodeName = form.InputBoxValue;

Invalidate();

}

}

}

private void ChangeMapItemClick(object sender, EventArgs e)

{

var item = sender as ToolStripMenuItem;

item.Checked = !item.Checked;

//смена содержимого меню

if (!ChangeMapItem.Checked)

{

contextMenuStrip.Items.Remove(ChangeNameNodeItem);

contextMenuStrip.Items.Add(StartNodeItem);

contextMenuStrip.Items.Add(EndNodeItem);

contextMenuStrip.Items.Add(SearchWayItem);

}

else

{

contextMenuStrip.Items.Add(ChangeNameNodeItem);

contextMenuStrip.Items.Remove(StartNodeItem);

contextMenuStrip.Items.Remove(EndNodeItem);

contextMenuStrip.Items.Remove(SearchWayItem);

}

}

protected override void OnMouseDown(MouseEventArgs e)

{

StartPoint = e.Location;

}

protected override void OnMouseUp(MouseEventArgs e)

{

//если мне не меняем положение вершин

if (!ChangeMapItem.Checked)

{

//получаем индексы точек

var start = GetIndex(StartPoint);

var end = GetIndex(e.Location);

//если успешно нашли

if (start >= 0 && end >= 0)

{

Node.Node buf = GetNodeAt(start);

if (buf.Outgoing.Count > 0)

{

for (int i = buf.Outgoing.Count - 1; i >= 0; i--)

{

var edge = buf.Outgoing[i];

//если у начальной вершины нет связи с конечной фершиной

if (edge.End != GetNodeAt(end))

{

//создаём связь начальная вершина -> конечная вершина

Nods[start].Connect(Nods[end], 10);

}

else

{

buf.Disconnect(GetNodeAt(end));

break;

}

}

}

else

{

GetNodeAt(start).Connect(GetNodeAt(end), 10);

}

Invalidate();

}

}

}

protected override void OnMouseClick(MouseEventArgs e)

{

//перемещение ноды

if (ChangeMapItem.Checked && e.Button == MouseButtons.Left)

{

//находим идекс ноды по её локации

int i = GetIndex(e.Location);

if (i != -1)

{

//перемещаем ноду к позиции мышки

Node = GetNodeAt(i);

Node.position = (e.Location);

Way = new List<Node.Node>();

Invalidate();

}

}

//Если пытаемся получить вершины для поиска

if (!ChangeMapItem.Checked && e.Button == MouseButtons.Left)

{

int i = GetIndex(e.Location);

//если ищем стартовую вершину

if (StartNodeFlag && i != -1)

{

StartNode = GetNodeAt(i);

StartNodeFlag = false;

}

else

{

//если ищем конечную вершину

if (EndNodeFlag && i != -1)

{

EndNode = GetNodeAt(i);

EndNodeFlag = false;

}

}

}

//вызов контекстного меню

if (e.Button == MouseButtons.Right)

{

contextMenuStrip.Show(this, e.Location);

}

//изменение веса между нодами

if (!ChangeMapItem.Checked && e.Button == MouseButtons.Left && GetBetween(e.Location))

{

//вызов формы для ввода данных

using (InputForm.InputBox form = new InputForm.InputBox(Edge.TraversalVelocity.ToString(), PointToScreen(e.Location)))

{

if (form.ShowDialog() == DialogResult.OK)

{

try

{

Edge.TraversalVelocity = int.Parse(form.InputBoxValue);

Invalidate();

}

catch (FormatException ex)

{

MessageBox.Show(ex.Message, "Ошибка!");

}

}

}

}

}

protected override void OnMouseMove(MouseEventArgs e)

{

OnMouseClick(e);

}

protected override void OnMouseDoubleClick(MouseEventArgs e)

{

//если флаг изменения карты = true

if (ChangeMapItem.Checked && e.Button == MouseButtons.Left)

{

//пытаемся получить индекс по локации клика

if (GetIndex(e.Location) >= 0)

{

//если клик по ноде, то удаляем её

RemoveNodeAt(GetIndex(e.Location));

Node = null;

}

else

{

//если клик не по ноде, то создаём новую ноду

AddNode(e.Location);

}

Invalidate();

}

}

public void AddNode(Point pos)

{

Nods.Add(new Node.Node(pos));

}

public Node.Node GetNodeAt(int index)

{

if (index >= 0 && index < Nods.Count)

{

return Nods[index];

}

return null;

}

public void RemoveNodeAt(int index)

{

if (index >= 0 && index < Nods.Count)

{

var node = Nods[index];

for (var i = node.Incoming.Count - 1; i >= 0; i--)

{

var edge = node.Incoming[i];

edge.Start.Outgoing.Remove(edge);

}

Nods.RemoveAt(index);

}

}

//проверяем если произошел клик между нодами

public bool GetBetween(Point pos)

{

for (int i = Nods.Count - 1; i >= 0; i--)

{

for (int j = Nods[i].Outgoing.Count - 1; j >= 0; j--)

{

var e = Nods[i].Outgoing[j];

var start = e.Start.position;

var end = e.End.position;

var m = (end.X - start.X) / 2 + start.X;

var n = (end.Y - start.Y) / 2 + start.Y;

if (20 > (int)Math.Sqrt(Math.Pow(m - pos.X, 2) + Math.Pow(n - pos.Y, 2)))

{

Edge = e;

return true;

}

}

}

return false;

}

//поиск индекса в массиве по локации клика

public int GetIndex(Point point)

{

int i;

for (i = Nods.Count - 1; i >= 0 && !Nods[i].NodeClick(point); i--) { }

return i;

}

//отрисовка нод и соеднинений между ними

protected override void OnPaint(PaintEventArgs e)

{

e.Graphics.FillRectangle(new SolidBrush(Color.LightBlue), ClientRectangle);

foreach (Node.Node n in Nods)

{

e.Graphics.FillEllipse(new SolidBrush(Color.White), n.position.X - n.NodeSize / 2, n.position.Y - n.NodeSize / 2, n.NodeSize, n.NodeSize);

e.Graphics.DrawEllipse(new Pen(Color.Black), n.position.X - n.NodeSize / 2, n.position.Y - n.NodeSize / 2, n.NodeSize, n.NodeSize);

e.Graphics.DrawString(n.NodeName, new Font("Arial", 16), new SolidBrush(Color.Black), n.position.X - 25, n.position.Y - 35);

foreach (Edge ed in n.Outgoing)

{

e.Graphics.DrawLine(new Pen(Color.Red, 2), n.position.X, n.position.Y, ed.End.position.X, ed.End.position.Y);

e.Graphics.DrawString(ed.ToString(), new Font("Arial", 16), new SolidBrush(Color.Black), new Point((ed.End.position.X - n.position.X) / 2 + n.position.X - 15, (ed.End.position.Y - n.position.Y) / 2 + n.position.Y - 25));

}

}

if (Node != null)

e.Graphics.FillEllipse(new SolidBrush(Color.Red), Node.position.X - Node.NodeSize / 2, Node.position.Y - Node.NodeSize / 2, Node.NodeSize, Node.NodeSize);

if (StartNode != null)

e.Graphics.FillEllipse(new SolidBrush(Color.Blue), StartNode.position.X - StartNode.NodeSize / 2, StartNode.position.Y - StartNode.NodeSize / 2, StartNode.NodeSize, StartNode.NodeSize);

if (EndNode != null)

e.Graphics.FillEllipse(new SolidBrush(Color.Green), EndNode.position.X - EndNode.NodeSize / 2, EndNode.position.Y - EndNode.NodeSize / 2, EndNode.NodeSize, EndNode.NodeSize);

foreach (Node.Node n in Way)

{

e.Graphics.FillEllipse(new SolidBrush(Color.Aqua), n.position.X - n.NodeSize / 2, n.position.Y - n.NodeSize / 2, n.NodeSize, n.NodeSize);

e.Graphics.DrawEllipse(new Pen(Color.Black), n.position.X - n.NodeSize / 2, n.position.Y - n.NodeSize / 2, n.NodeSize, n.NodeSize);

foreach (Edge ed in n.Outgoing)

{

e.Graphics.DrawLine(new Pen(Color.Blue, 2), n.position.X, n.position.Y, ed.End.position.X, ed.End.position.Y);

}

}

}

}

}

Код класса Edge

namespace FindWayOnNods.Node

{

public class Edge

{

public Edge(Node start, Node end, int traversalVelocity)

{

Start = start;

End = end;

TraversalVelocity = traversalVelocity;

}

public int TraversalVelocity { get; set; }

public Node Start { get; }

public Node End { get; }

public override string ToString() => TraversalVelocity.ToString();

}

}

Код класса Node

using System;

using System.Collections.Generic;

using System.Drawing;

namespace FindWayOnNods.Node

{

public class Node

{

public Node(Point position)

{

Incoming = new List<Edge>();

Outgoing = new List<Edge>();

this.position = position;

NodeSize = 25;

}

public List<Edge> Incoming { get; }

public List<Edge> Outgoing { get; }

public int NodeSize

{

get => nodeSize;

set

{

if (value > 0)

{

nodeSize = value;

}

}

}

public string NodeName { get; set; } = "";

public Point position { get; set; }

private int nodeSize;

public void Connect(Node node, int traversalVelocity)

{

var f = true;

var f1 = true;

foreach (Edge e in Outgoing)

{

if (e.End == node)

{

f = false;

break;

}

}

foreach (Edge e in Incoming)

{

if (e.Start == node)

{

f1 = false;

break;

}

}

if (this != node && f && f1)

{

var edge = new Edge(this, node, traversalVelocity);

Outgoing.Add(edge);

node.Incoming.Add(edge);

}

}

public void Disconnect(Node node)

{

for (var i = Outgoing.Count - 1; i >= 0; i--)

{

var edge = Outgoing[i];

if (edge.End == node)

{

Outgoing.Remove(edge);

node.Incoming.Remove(edge);

}

}

}

public bool NodeClick(Point ClickPosition)

{

return NodeSize / 2 > (int)Math.Sqrt(Math.Pow(position.X - ClickPosition.X, 2) + Math.Pow(position.Y - ClickPosition.Y, 2));

}

}

}

Код класса Path

using FindWayOnNods.Node;

using System.Collections.Generic;

namespace FindWayOnNods.PathFinder

{

public class Path

{

public Path(List<Edge> Edges)

{

ListWay = new List<Node.Node>();

if (Edges.Count > 0)

{

for (var i = 0; i < Edges.Count; i++)

{

TraversalVelocity += Edges[i].TraversalVelocity;

var start = new Node.Node(Edges[i].Start.position);

var end = new Node.Node(Edges[i].End.position);

start.Connect(end, Edges[i].TraversalVelocity);

ListWay.Add(start);

}

var buf = new Node.Node(Edges[Edges.Count-1].End.position);

ListWay.Add(buf);

}

else

{

TraversalVelocity = -1;

}

}

public List<Node.Node> ListWay { get; }

public int TraversalVelocity { get; }

}

}

Код класса PathFinder

using FindWayOnNods.MinHeap;

using System.Collections.Generic;

namespace FindWayOnNods.PathFinder

{

public class PathFinder

{

private MinHeap<PathFinderNode> minHeap;

private Dictionary<Node.Node, PathFinderNode> Nodes;

private PathReconstructor PathReconstructor;

private PathFinderNode NodeClosestToGoal;

public PathFinder()

{

minHeap = new MinHeap<PathFinderNode>();

PathReconstructor = new PathReconstructor();

Nodes = new Dictionary<Node.Node, PathFinderNode>();

}

public Path FindPath(Node.Node start, Node.Node goal)

{

minHeap.Clear();

Nodes.Clear();

var st = new PathFinderNode(start, 0);

NodeClosestToGoal = st;

Nodes.Add(st.Node, st);

minHeap.Insert(st);

while (minHeap.Count > 0)

{

var current = minHeap.Extract();

if (current.Node == goal)

{

return PathReconstructor.ConstructPath(current.Node, goal);

}

if (current.castSoFar < NodeClosestToGoal.castSoFar)

{

NodeClosestToGoal = current;

}

foreach (var edge in current.Node.Outgoing)

{

var n = edge.End;

var cast = current.castSoFar + edge.TraversalVelocity;

if (Nodes.TryGetValue(n, out var path))

{

if (path.castSoFar > cast)

{

minHeap.Remove(path);

}

}

PathReconstructor.SetCameFrom(n, edge);

var node = new PathFinderNode(n, cast);

minHeap.Insert(node);

Nodes[n] = node;

}

}

return PathReconstructor.ConstructPath(NodeClosestToGoal.Node, goal);

}

}

}

Код класса PathFinderNode

using FindWayOnNods.MinHeap;

using System.Collections.Generic;

namespace FindWayOnNods.PathFinder

{

public class PathFinder

{

private MinHeap<PathFinderNode> minHeap;

private Dictionary<Node.Node, PathFinderNode> Nodes;

private PathReconstructor PathReconstructor;

private PathFinderNode NodeClosestToGoal;

public PathFinder()

{

minHeap = new MinHeap<PathFinderNode>();

PathReconstructor = new PathReconstructor();

Nodes = new Dictionary<Node.Node, PathFinderNode>();

}

public Path FindPath(Node.Node start, Node.Node goal)

{

minHeap.Clear();

Nodes.Clear();

var st = new PathFinderNode(start, 0);

NodeClosestToGoal = st;

Nodes.Add(st.Node, st);

minHeap.Insert(st);

while (minHeap.Count > 0)

{

var current = minHeap.Extract();

if (current.Node == goal)

{

return PathReconstructor.ConstructPath(current.Node, goal);

}

if (current.castSoFar < NodeClosestToGoal.castSoFar)

{

NodeClosestToGoal = current;

}

foreach (var edge in current.Node.Outgoing)

{

var n = edge.End;

var cast = current.castSoFar + edge.TraversalVelocity;

if (Nodes.TryGetValue(n, out var path))

{

if (path.castSoFar > cast)

{

minHeap.Remove(path);

}

}

PathReconstructor.SetCameFrom(n, edge);

var node = new PathFinderNode(n, cast);

minHeap.Insert(node);

Nodes[n] = node;

}

}

return PathReconstructor.ConstructPath(NodeClosestToGoal.Node, goal);

}

}

}

Код класса PathReconstructor

using FindWayOnNods.Node;

using System.Collections.Generic;

namespace FindWayOnNods.PathFinder

{

class PathReconstructor

{

private Dictionary<Node.Node, Edge> CameFrom;

public PathReconstructor()

{

CameFrom = new Dictionary<Node.Node, Edge>();

}

public void SetCameFrom(Node.Node node, Edge via)

{

CameFrom[node] = via;

}

public Path ConstructPath(Node.Node start, Node.Node goal)

{

var current = start;

var edges = new List<Edge>();

while (CameFrom.TryGetValue(current, out var via))

{

edges.Add(via);

current =via.Start;

}

edges.Reverse();

return new Path(edges);

}

}

}