**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

Нижегородский государственный технический университет им.Р.Е. Алексеева

Кафедра **Автоматизация, энергетика, математика и информационные системы**

Заведующий кафедрой

Вадова Л.Ю..

*(подпись) (фамилия, и., о.)*

*(дата)*

Реализация алгоритма cтроительной трассировки на языке программирования C#

*(наименование темы проекта или работы)*

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе по дисциплине «Алгоритмы дискретной математики»

*(вид документа – проект дипломный, курсовой, исследовательская работа или часть и т.п.)*

**КОНСУЛЬТАНТЫ: РУКОВОДИТЕЛЬ**

1. По нормоконтролю Харитонова И.Ю.

*(подпись) (фамилия, и., о.)*

*(дата)*

*(подпись) (фамилия, и., о.)*

*(дата)*

**СТУДЕНТ**

2. По Шумилкин С.Д.

*(подпись) (фамилия, и., о.)*

17-ПМ

*(подпись) (фамилия, и., о.) (дата) (группа или шифр)*

*(дата)*

3. По

*(подпись) (фамилия, и., о.)*

*(дата)* Проект защищен *(дата)* С оценкой

2019 г.

Содержание

Введение 3

1 Постановка задачи 4

2 Средства разработки приложения 5

2.1 Платформа разработки .NET Framework 5

2.2 Язык программирования C# 5

2.3 Технология Windows Forms 6

2.4 Visual Studio 2017 6

3 Основные понятия 7

4 Теоритическая часть 8

5 Описание алгоритма строительной трассировки 9

6 Программная реализация 12

7 Демонстрация работы программы 13

Заключение 14

Список литературы 15

ПРИЛОЖЕНИЕ 1 16

ПРИЛОЖЕНИЕ 2 16

ПРИЛОЖЕНИЕ 3 17

ПРИЛОЖЕНИЕ 4 18

ПРИЛОЖЕНИЕ 5 19

**Введение**

**Задача о кратчайшем пути** — задача поиска самого короткого [пути](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D1%82%D1%8C_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)) (цепи) между двумя точками (вершинами) на [графе](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), в которой минимизируется сумма весов рёбер, составляющих путь. [1]

«Задача поиска кратчайшего пути» (задача о минимальном пути, задача о дилижансе), в последнее время получила широкое распространение, благодаря своему применению для решения множества других задач.

В настоящее время она применяется в алгоритмах поиска оптимального пути между двумя объектами (GPS-навигация), в системах автоматического пилотирования, для нахождения кратчайшего пути прохождения Internet-пакета по сети, и множества других.

Задача о кратчайшем пути является одной из важнейших классических задач теории графов. На сегодняшний день известно множество алгоритмов для ее решения.

Кратчайший путь рассматривается с помощью математической модели, называемой графом.

1 Постановка задачи

Пусть дано множество непересекающихся многоугольников P –полигональных препятствий. Пусть дана стартовая точка s и конечная точка f. Нам необходимо построить кратчайший маршрут из точки s в точку f, при этом, будем считать что путь может путь может проходить по границе препятствия, но не внутри него.

**Вход и выход программы**

Входные данные программы представлены нарисованными пользователем в окне Windows Forms полигонами-препятствиями и двумя точками, между которыми нужно проложить маршрут.

При нажатии определенной клавиши на клавиатуре, программа рассчитывает и отрисовывает наиболее короткий между точками путь, после чего пользователь может очистить окно с входными данными и протестировать алгоритм повторно уже на других входных данных

2 Средства разработки приложения

2.1 Платформа разработки .NET Framework

.NET Framework — программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду. [2]

Программа для .NET Framework, написанная на любом поддерживаемом языке программирования, сначала переводится компилятором в единый для .NET промежуточный байт-код Common Intermediate Language (CIL). В терминах .NET получается сборка. Затем код либо исполняется виртуальной машиной Common Language Runtime (CLR), либо транслируется утилитой NGen.exe в исполняемый код для конкретного целевого процессора. Использование виртуальной машины предпочтительно, так как избавляет разработчиков от необходимости заботиться об особенностях аппаратной части. В случае использования виртуальной машины CLR встроенный в неё JIT-компилятор «на лету» (just in time) преобразует промежуточный байт-код в машинные коды нужного процессора. Современная технология динамической компиляции позволяет достигнуть высокого уровня быстродействия. Виртуальная машина CLR также сама заботится о базовой безопасности, управлении памятью и системе исключений, избавляя разработчика от части работы.

2.2 Язык программирования C#

C# — объектно-ориентированный язык программирования. [3] Разработан в 1998—2001 годах группой инженеров компании Microsoft под руководством Андерса Хейлсберга и Скотта Вильтаумота как язык разработки приложений для платформы Microsoft .NET Framework. C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML. [3]

2.3 Технология Windows Forms

Windows Forms — интерфейс программирования приложений (API), отвечающий за графический интерфейс пользователя и являющийся частью Microsoft .NET Framework. Данный интерфейс упрощает доступ к элементам интерфейса Microsoft Windows за счет создания обёртки для существующего Win32 API в управляемом коде.

2.4 Visual Studio 2017

Интегрированная среда разработки Visual Studio — это оригинальная среда запуска, которая позволяет редактировать, отлаживать и создавать код, а затем публиковать приложения. [4] Интегрированная среда разработки (IDE) — это многофункциональная программа, которую можно использовать для различных аспектов разработки программного обеспечения. Помимо стандартного редактора и отладчика, которые существуют в большинстве сред IDE, Visual Studio включает в себя компиляторы, средства выполнения кода, графические конструкторы и многие другие функции для упрощения процесса разработки программного обеспечения.

3 Основные понятия

**Точка** — одно из фундаментальных понятий математики, абстрактный объект в пространстве, не имеющий никаких измеримых характеристик (нульмерный объект).

**Прямая** — одно из фундаментальных понятий евклидовой геометрии. При систематическом изложении геометрии прямые линии обычно принимаются за одно из исходных (неопределяемых) понятий.

**Отрезок** — часть прямой, ограниченная двумя точками. Точнее: это множество, состоящее из двух различных точек данной прямой (которые называются концами отрезка) и всех точек, лежащих между ними (которые называются его внутренними точками).

**Задача о кратчайшем пути** — задача поиска самого короткого пути (цепи) между двумя точками (вершинами) на графе, в которой минимизируется сумма весов рёбер, составляющих путь. Задача о кратчайшем пути является одной из важнейших классических задач теории графов.

**Алгоритм Дейкстры** — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. [5]

**Граф видимости** ­— в вычислительной геометрии и планировании движений роботов граф видимости — это граф взаимной видимости точек пространства, обычно для множества точек и преград на евклидовой плоскости. Любая вершина в графе представляет точку пространства, а любое ребро представляет прямую видимость между точками. То есть, если отрезок прямой, соединяющий две точки пространства, не проходит через какую-либо преграду, в графе будет нарисовано ребро.

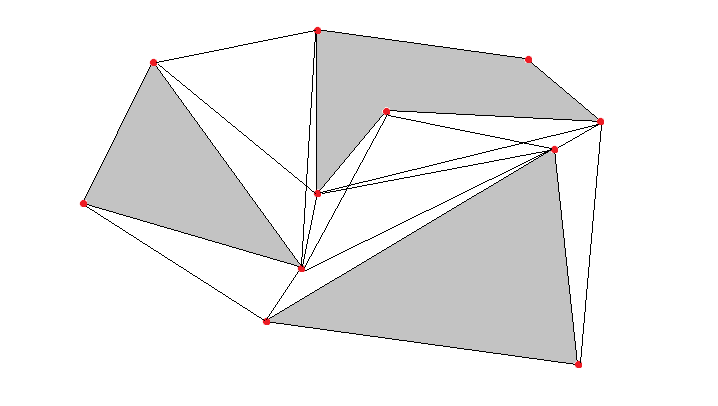


Рисунок 1- пример графа видимости

4 Теоретическая часть

**Оптимальный путь в алгоритме строительной трассировки**

Оптимальный путь — это кусочно-линейная ломаная, вершины которой

совпадают с вершинами препятствий.

**Доказательство:**

Предположим, что кратчайший путь T — не является кусочно-линейной

ломаной. В таком случае, на пути T существует такая точка p, которая не принадлежит ни одному прямому отрезку из T (рисунок 1). Это означает что существует ε-окрестность точки p (рисунок 2) в которую не попадает ни одно препятствие. В таком случае, подпуть Tε, который находится внутри ε-окрестности, может быть сокращён по хорде, соединяющий точки пересечения границы ε-окрестности с путем T. Раз часть пути может быть уменьшена, значит и весь путь может быть уменьшен, а значит исходное предположение некорректно. Путь T — кусочно-линейная ломаная.

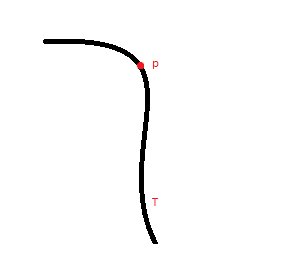
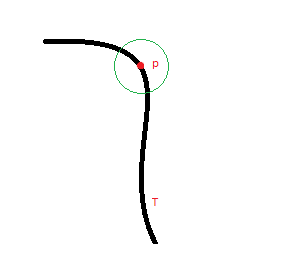
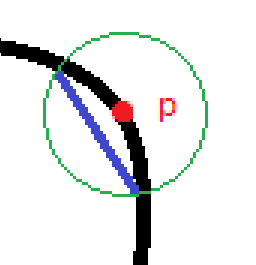


Рис. 1. Точка p на пути T

Рис. 3. Хорда, являющаяся кратчайшим путём

Рис. 2. ε-окрестность точки p

**Вершины пути совпадают с вершинами препятствий**

Предположим теперь что вершины пути T не совпадают с вершинами многоугольников-препятствий. Рассмотрим вершину v пути T. Вершина не может лежать внутри свободного пространства, поскольку в противном случае, применив аналогичный подход, мы нашли бы в ε-окрестности вершины v более короткий путь по хорде (рисунок 4). Значит, вершина v обязана лежать на границах препятствий (в их вершинах или на рёбрах). Но и на рёбрах вершина лежать не может, поскольку (опять рассматриваем ε-окрестность), мы сможем срезать по хорде (рисунок 5).

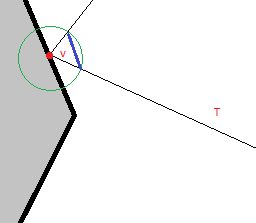
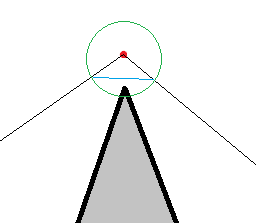


Рис. 5. ε-окрестность вершины пути, лежащей на ребре

Рис. 4. ε-окрестность вершины пути, лежащей в свободном пространстве

5 Описание алгоритма строительной трассировки

**Идея алгоритма**: Нужно построить граф, состоящий из сторон многоугольников и из прямолинейных отрезков, соединяющих вершины разных многоугольников или вершины одного многоугольника при условии, что они «простреливаются» друг из друга. Начальную и конечную точки трассы необходимо «соединить» с «простреливаемыми» из них вершинами. После того, как граф построен, на нём нужно построить кратчайший путь из ***i*** в ***j*** пользуясь алгоритмом Дейкстры.

Количество вершин сети   
Формирование сети (матрицы расстояний размером *n* на *n*) – тройной цикл:

внешний – по *i* – перебор вершин, откуда «стреляют»;  
 средний – по *j* (*j*:=*i*+1 to *n*) - перебор вершин, куда «стреляют»;  
 внутренний – по *k* (*k* :=1 to *n-2*) – это проверка, не пересекает ли  
 ребро (*i*, *j*) *k*-ю сторону какого-нибудь многоугольника-препятствия.

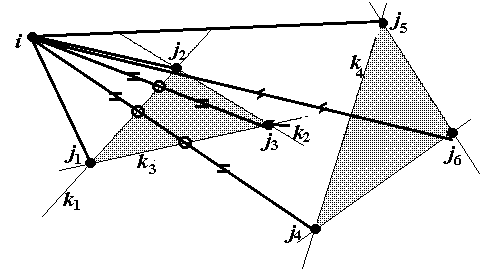


Рисунок 3- формирование сети

**Формирование сети**

Граф видимости хранится в видете матрицы расстояний.

Элементы матрицы расстояний *R* рассчитываются для тех пар вершин, которые «простреливаются». Формула вычисления расстояний:



После того, как сеть построена, на ней нужно построить кратчайший путь из i в j пользуясь алгоритмом Дейкстры.

**Начальное состояние алгоритма**: имеется множество полигонов-препятствий и координаты каждой из их вершин в виде {*x; y*};

Перебираем все пары точек (*a, b*) и проверяем, можно ли добавить ребро между ними, то есть нет ли пересечений с полигонами. Если отрезок пересекает хотя бы одну грань полигона, то он не включается в граф видимости. У нас есть O(n2)  пар вершин и O(n) ребер, то есть временная сложность алгоритма порядка O(n3).

Каноническое уравнение прямой, проходящей через две заданные точки (*x*1, *y*1) и (*x*2, *y*2) имеет вид:

или в общем виде:

т.е. получили общее уравнение прямой линии на плоскости в декартовых координатах:

Отсюда следует:

*A = y1-y2*

*B = x2-x1*

Последний коэффициент ищем в виде:

*C = A\*x*1*+B\*y*1, где вместо *x*1, *y*1 – координаты точки – начала отрезка, через которую проходит прямая.

Таким способом мы находим уравнения прямых для пары точек (*a, b*) и для одной из граней выбранного многоугольника *P*i (*c, d*):

*A1x + B1y = C1  
A2x + B2y = C2*

после чего проверяем полученную пару прямых на коллинеарность:

1. если определитель системы det = *A*1\**B*2 - *A*2\**B*1=0, то прямые коллинеарны.
2. если же определитель не равен нулю, то ищем точку пересечения методом Крамера:

*x* = (*B*2\**C*1 - *B*1\**C*2)/*det*

*y* = (*A*1\**C*2 - *A*2\**C*1)/*det*

Далее важно проверить, чтобы найденные координаты пересечения *прямых* принадлежали каждому из соответствующих *отрезков -* (*a, b*), (*c, d*).

Если отрезок идет из точки (*x*1, *y*1) в точку (*x*2, *y*2), то достаточно проверить выполнение условий

min (*x*1, *x*2) ≤ *x* ≤ max (*x*1, *x*2)

min (*y*1, *y*2) ≤ *y* ≤ max (*y*1, *y*2)

и сделать то же самое для второго отрезка, который задает грань полигона.

Те отрезки, которые не пересеклись с гранями полигонов включаются в граф видимости.

В конце концов одна из точек (*i*) задается как исходная, а другая как конечная (*j*) и с помощью алгоритма Дейкстры на полученном графе находится кратчайший путь между ними.

6 Программная реализация

1. Основные классы:
   1. class Form1 – основной класс, экземпляр которого создается автоматически при запуске программы. Внутри него находятся все нижеописанные методы.
   2. class Polygon – пользовательский класс, представляющий собой структуру «полигон». Внутри класса хранится множество точек – вершин полигона.
2. Основные методы:
   1. static void VisibilityGraph() – метод строит граф видимости (стр 9).Для этого берутся все пары точек, на их основе строятся отрезки и проверяются на пересечение с гранями полигонов.
   2. static void Deikstra()- метод представляет из себя алгоритм Дейкстры по нахождению кратчайшего пути из точки *a* в точку *b*.
   3. static void DrawPolCont() – метод отрисовывает контуры для каждого существующего на полотне полигона.
   4. static bool CheckIfNear(Polygon p, Point a, Point b) – метод принимает на вход полигон и две точки, проверяет принадлежат ли эти точки данному полигону и, если принадлежат проверяет являются ли точки соседними в полигоне. Если они не являются соседними, то отрезок (a,b) пересекает внутр. часть полигона и его не нужно добавлять в граф видимости.
   5. static double Distance(Point a, Point b) – метод принимает на вход две точки и находит расстояние между ними, которое он возвращает в вызывающую функцию.
   6. static bool Intercept(Point a, Point b) – метод принимает на вход две точки и проверяет пересекает ли отрезок, образованный из них какую-либо грань какого-либо полигона. Если пересекает, то этот отрезок не добавляется в граф видимости.
   7. static void DrawEdges() – метод выполняет прорисовку граней полигонов во время добавления новых полигонов.
   8. static void Redraw() – метод вызывает множество других методов для перерисовки всех графических объектов.
   9. static void Clear() – метод осуществляет очистку полотна рисования и некоторых массивов.
   10. private void pictureBox1\_MouseMove(object sender, MouseEventArgs e) – метод для обработки события движения мыши. Используется для получения координат курсора по осям X и Y;
   11. private void Form1\_KeyPress(object sender, KeyPressEventArgs e) – метод для обработки нажатия ключевых клавиш.
   12. static void DrawPolys() – метод, выполняющий отрисовку полигонов.
   13. static void DrawDots() – метод, выполняющий отрисовку всех точек на полотне.
   14. private void pictureBox1\_Click(object sender, EventArgs e) – метод для обработки события клика мыши. Используется для добавлении новых точек на полотно.
3. Используемые структуры данных:
   1. Point – структура, содержащая в себе координаты точки по осям *X* и *Y*.
   2. List<> - структура, представляющая собой динамический массив с данными выбранного типа.

7 Демонстрация работы программы

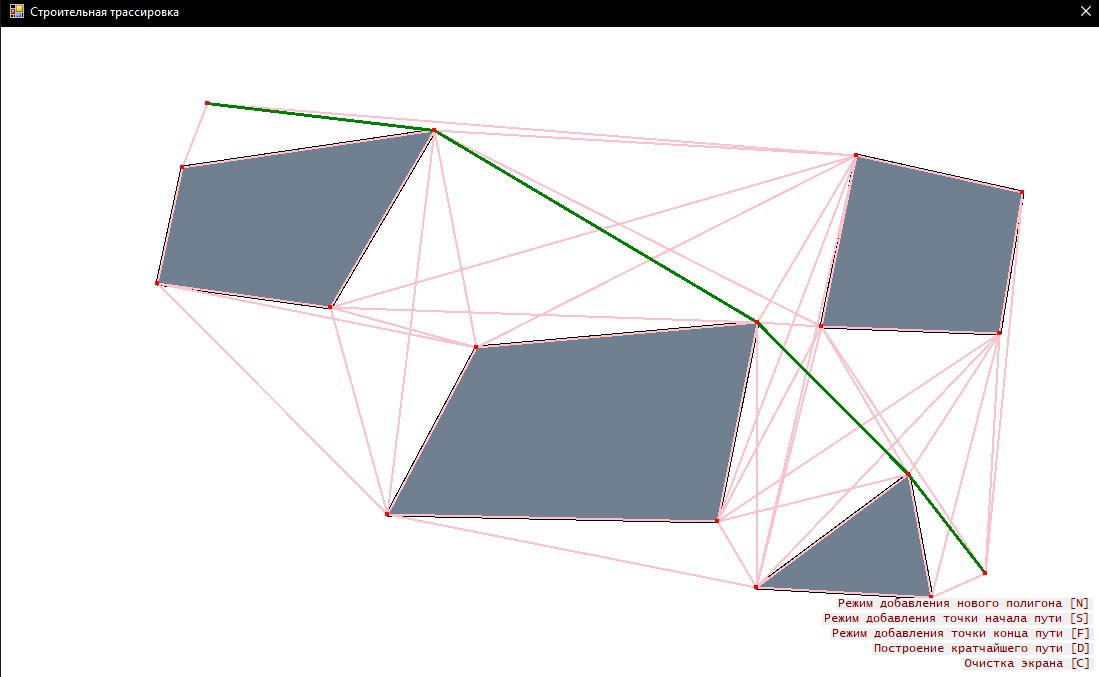


Рисунок 4- Окно программы с результатами работы алгоритма

На данном рисунке в виде серых многоугольников изображены препятствия. Розовыми линии образуют граф видимости. Зеленым цветом обозначается путь из исходной вершины в конечную.

Инструкция по работе с программой приведена в приложении 5.

**Заключение**

В настоящей курсовой работе был рассмотрен и программно реализован алгоритм строительной трассировки. Для этого был использован язык C# и технология Windows Forms, о которой можно узнать побольше из соответствующих книг. [7]

Написанная программа решает актуальную в наше время проблему по поиску кратчайшего пути из точки A в точку B, минуя при этом различные препятствия. Эта проблема встречается в робототехнике, прокладке путей с помощью систем GPS и в других областях повседневной жизни.

Интерфейс программы сделан интуитивно понятным, в следствии чего у конечных пользователей не должно возникнуть проблем с её использованием.

Реализованная программа имеет в себе некоторые проблемы:

1. При некоторых входных данных граф видимости может строиться некорректно.
2. Использованный в программе алгоритм Дейкстры изредка давал сбой, связанный с вызовом бесконечного цикла, поэтому пришлось проконтролировать условие его завершения с помощью введения доп. переменной.
3. Из-за некоторых особенностей программной реализации, не получилось сделать полотно рисования “растягиваемым” под необходимый пользователю размер.

Тем не менее, после множества тестов, я могу сделать заключение, что в большинстве случаев программа работает нормально и выдаёт верный результат за сравнительно короткой время.

С основными отрывками кода можно ознакомиться в приложения 1-4.

**Список используемых источников**

1. <http://ru.wikipedia.org/Задача_о_кратчайшем_пути>

1. <https://flexberry.github.io/ru/gbt_dotnet.html>

1. <https://ru.wikipedia.org/wiki/C_Sharp>
2. <https://docs.microsoft.com/ru-ru/visualstudio/get-started/visual-studio-ide?view=vs-2019>
3. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е. — М.: Вильямс, 2005. — 1296 с
4. Шилдт, Г. C# 4.0 Полное руководство / Г. Шилдт. – Спб.: Вильямс,

2011 – 1056с.

### [Петцольд, Ч.](https://www.libex.ru/?cat_author=%CF%E5%F2%F6%EE%EB%FC%E4,%20%D7.&author_key=207) Программирование с использованием Microsoft Windows Forms, 2006 – 432 с.

**Приложение 1**

class Polygon

{

public List<Point> pts; // массив точек полигона

public Polygon()

{

pts = new List<Point>();

}

public Polygon(List<Point> newPts)

{

pts = new List<Point>(newPts.Count);

pts.AddRange(newPts);

}

public void AddPt(Point pt)

{

this.pts.Add(pt);

}

public Point[] GetPts()

{

// возвращаем МАССИВ с точками полигона

Point[] pts\_temp = new Point[pts.Count];

pts.CopyTo(pts\_temp);

return pts\_temp;

}

}

**Приложение 2**

static void VisibilityGraph()

{

Matr = new double[pts\_all.Count, pts\_all.Count];

for(int i = 0; i<pts\_all.Count;i++)

{

for(int j =0;j<pts\_all.Count;j++)

{

if (!Intercept(pts\_all[i],pts\_all[j]))

{

Matr[i, j] = Distance(pts\_all[i], pts\_all[j]); // записываем

длину ребра в матр. расст.

g.DrawLine(new Pen(Color.Pink, 2), pts\_all[i], pts\_all[j]);

}

else

{

Matr[i, j] = INF; // записываем INF, если пути нет

}

}

}

}

**Приложение 3**

static bool Intercept(Point a, Point b)

{

double A1= a.Y - b.Y;

double B1 = b.X - a.X;

double C1 = A1 \* a.X + B1 \* a.Y; // создаем уравнение прямой

foreach (Polygon pg in pols)

{

if (!CheckIfNear(pg, a, b)) return true;

for (int i = 0; i < pg.pts.Count - 1; i++)

{

double A2 = pg.pts[i].Y - pg.pts[i+1].Y;

double B2 = pg.pts[i+1].X - pg.pts[i].X;

double C2 = A2 \* pg.pts[i].X + B2 \* pg.pts[i].Y; // уравнение прямой

- грани полигона

double det = A1 \* B2 - A2 \* B1;

if (det == 0) return false; // проверка на коллинеарность

else

{

double x = (B2 \* C1 - B1 \* C2) / det; // находим коорд.

пересечения

double y = (A1 \* C2 - A2 \* C1) / det;

// проверяем принадлежит ли найденная точка обоим ОТРЕЗКАМ

if (Math.Min(a.X, b.X) < x && x < Math.Max(a.X, b.X))

{

if (Math.Min(a.Y, b.Y) < y && y < Math.Max(a.Y, b.Y))

{

if( Math.Min(pg.pts[i].X, pg.pts[i+1].X) < x && x < Math.Max(pg.pts[i].X, pg.pts[i + 1].X) )

if (Math.Min(pg.pts[i].Y, pg.pts[i + 1].Y) < y && y < Math.Max(pg.pts[i].Y, pg.pts[i + 1].Y))

return true;

}

}

}

}

// аналогично проверяем прямую [перв. точка полигона; посл.

точка полигона]

if (pg.pts.Count > 2)

{

double A2 = pg.pts[pg.pts.Count - 1].Y - pg.pts[0].Y;

double B2 = pg.pts[0].X - pg.pts[pg.pts.Count - 1].X;

double C2 = A2 \* pg.pts[pg.pts.Count - 1].X + B2 \* pg.pts[pg.pts.Count - 1].Y;

double det = A1 \* B2 - A2 \* B1;

if (det == 0) return false;

else

{

double x = (B2 \* C1 - B1 \* C2) / det;

double y = (A1 \* C2 - A2 \* C1) / det;

if (Math.Min(a.X, b.X) < x && x < Math.Max(a.X, b.X))

{

if (Math.Min(a.Y, b.Y) < y && y < Math.Max(a.Y, b.Y))

if (Math.Min(pg.pts[0].X, pg.pts[pg.pts.Count - 1].X) < x && x < Math.Max(pg.pts[0].X, pg.pts[pg.pts.Count - 1].X))

if (Math.Min(pg.pts[0].Y, pg.pts[pg.pts.Count - 1].Y) < y && y < Math.Max(pg.pts[0].Y, pg.pts[pg.pts.Count - 1].Y))

return true;

}

}

}

}

return false;

}

**Приложение 4**

static void Deikstra()

{

int[] a = new int[pts\_all.Count];

int[] c = new int[pts\_all.Count];

double[] b = new double[pts\_all.Count]; // массив с мин. путём до данной

вершины

double dmin; // мин. расстояние в массиве b

int i, k, z, j, m;

// i - номер исх. вершины, k,z,j,m - вспомог. перем-ые

i = 0;

for (j = 0; j < pts\_all.Count; j++)

{

a[j] = 0; c[j] = i; b[j] = Matr[i, j];

// заполняем массивы числами

// a заполняем нулями, в b копируем строку расстояний из исх

// матрицы. В массив c пишем номер исходной вершины

}

a[i] = 1; c[i] = -1; // помечаем исх. вершину в массивах a и c

for (m = 0; m < pts\_all.Count; m++)

{

dmin = 300000;

for (k = 0; k < pts\_all.Count; k++)

if ((a[k] == 0) && (b[k] < dmin))

{ // если вершина не помечена и расстояние до неё < минимума

j = k; dmin = b[k];

}

a[j] = 1; // помечаем выбранную вершину

for (k = 0; k < pts\_all.Count; k++)

if (b[k] > b[j] + Matr[j, k])

{

// если текущ расст. до вершины больше, чем

// расстояние до неё через выбранную вершину

b[k] = b[j] + Matr[j, k];

c[k] = j;

}

}

k = pts\_all.Count - 1; // выбираем посл. вершину

if (b[pts\_all.Count - 1] == INF) return;

else

{

g.DrawLine(new Pen(Color.Green, 3), pts\_all[k], pts\_all[c[k]]);

}

int d = 0;

z = c[k];

while (c[z] != -1 && d < pts\_all.Count)

{ // пока не встретим метку 1-ой вершины

d++;

g.DrawLine(new Pen(Color.Green, 3), pts\_all[z], pts\_all[c[z]]);

z = c[z];

}

}

**Приложение 5**

**Инструкция по работе с программой**

Для работы с программой необходимо предварительно переключить раскладку клавиатуры на английский язык!!!

1. Для добавления нового полигона необходимо нажать кнопку N на клавиатуре, после чего кликать курсором на место, в которое необходимо поставить сл. Вершину полигона. Когда полигон завершен нужно нажать кнопку ESC и полигон будет добавлен на полотно. После чего можно будет добавить еще несколько полигонов.
2. Для задания начальной точки нужно нажать кнопку S и кликнуть в место установки точки на полотне.
3. Для задания конечной точки нужно нажать кнопку F кликнуть в место установки точки на полотне.
4. Для построения пути нужно нажать кнопку D, после чего будет нарисован кратчайший путь между начальной и конечной точкой.
5. Для очистки полотна нужно нажать кнопку C, после чего можно нарисовать новые препятствия.

Не рекомендуется рисовать полигоны с большим кол-вом вершин, и в принципе рисовать много полигонов, т.к. это может привести к ошибкам.