Florian Pallas
pallas.florian@gmail.com
Teilnahme-ID: **51811**
Team-ID: **01171**

Aufgabe 2: Dreiecksbeziehungen

37. BWINF - Runde 2

Inhalt

AUFGABE[1]

Situation

Mehrere dreieckige Grundstücke sollen entlang einer geraden Straße angeordnet werden. Dabei soll jedes Dreieck mit mindestens einer Ecke die Straße berühren und muss oberhalb der Küstenstraße liegen. Der Abstand zwischen den meist entfernten Grundstücken soll so klein wie möglich sein.

Gegebene Werte

Im Koordinatensystem verläuft die Straße entlang der x-Achse.

Die Dreiecke sind in Form von Polygonen mit deren Eckpunkten im Koordinatensystem gegeben.

Distanzen werden in Metern angegeben.

Lösung

Das Programm soll berechnen und darstellen, wie die Grundstücke angeordnet werden müssen. Zudem sollen die Dreiecke aufsteigend nach ihrer Platzierung sortiert mit ihren Koordinatenpaaren ausgegeben werden.

Mögliche Transformationen

Um eine Lösung zu finden, dürfen die Dreiecke verschoben, rotiert oder auch gespiegelt werden.

[1] Entnommen aus: 37. BWINF: 2. Runde - Aufgabenblatt (https://bwinf.de/bundeswettbewerb/37/2-runde/)

IDEE

Herangehensweise

Um die optimale Lösung zu finden, wäre womöglich “Brute Force” oder ein kluger Backtracking Algorithmus nötig. Allerdings sind diese Algorithmen nicht sehr effizient. Eine Annäherung an die optimale Lösung könnte deutlich zeitnäher eine Lösung liefern. In meinen Augen ist es bei dieser Aufgabe ebenso wichtig, möglichst schnell eine Lösung zu finden. Vor allem, da die Anzahl der Dreiecke bereits in den verschiedenen Beispielaufgaben schnell ansteigt.

Nach einiger Recherche habe ich mich für eine Umsetzung auf Grundlage des Behälterproblems[2] (bin packing problem) entschieden.

Eine Lösung würde der optimalen Lösung sehr nah kommen. Zudem bieten die Lösungsalgorithmen eine herausragende Laufzeit.

Behälterproblem

Das Behälterproblem basiert auf folgender Frage:

Wie viele Behälter benötige ich, um alle Objekte mit einer bestimmten Gewichtung einem Behälter zuzuordnen.

Will man das Ergebnis des Behälterproblems optimieren, muss man eine Zuordnung finden, bei der die letztendlich benötigte Anzahl an Behältern möglichst gering ist.

Standardverfahren

Es gibt verschiedene Standardverfahren zur Lösung dieses Problems. Eines davon ist das “First Fit Decreasing”-Verfahren. Hier wird nach Gewichtung der Objekte sortiert und diese dann den Behältern nacheinander zugeordnet. Ist ein Behälter voll, wird ein neuer geöffnet.

Nutzen für die Aufgabe

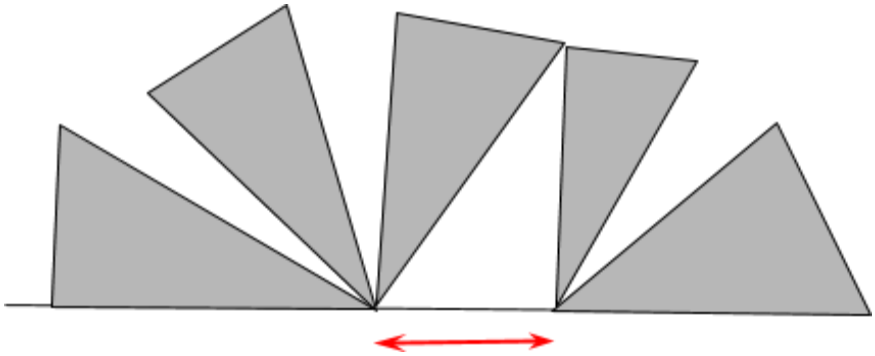
Hat man einmal mehrere Behälter mit Dreiecken befüllt, so kann man diese aneinanderreihen. Dabei wird auf Kollision der Dreiecke geprüft um den minimalen Abstand zwischen den Behältern

[2] Entnommen aus: Wikipedia - Behälterproblem

zu erzielen. Je nach Zuordnung der Dreiecke können größere und kleinere Lücken zwischen den Behältern entstehen. Dies gilt es also zu optimieren.

Meine Abwandlung

Vorteil durch Überhänge



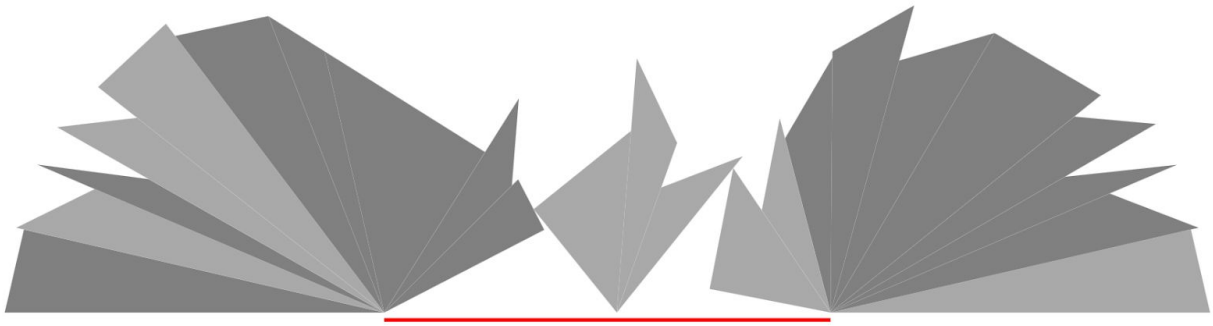
Wie man in der Abbildung sieht, zählt bei Dreiecken, welche mehrere Berührungspunkte mit der Geraden haben der innerste Berührungspunkt[3]. Dies kann man sich zunutze machen, indem man versucht, Dreiecke so anzuordnen, dass zu Beginn und zum Ende der Häuserkette möglichst große Flächen zu den Seiten überhängen. So kann man die Gesamtdistanz minimal halten.

Behältertypen

Aufgrund dieses Vorteils habe ich mich dazu entschieden vom Standard-Verfahren zur Lösung des Behälterproblems abzuweichen und das Problem etwas anders zu lösen. Ich unterteile zunächst die Behälter in 3 verschiedene Typen:

Startbehälter, Mittelbehälter und Endbehälter

[3] Siehe: Aufgabenblatt - Abbildung zu Aufgabe 2

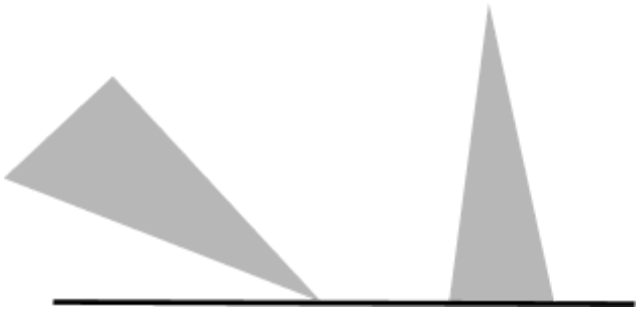


Start- und Endbehälter machen sich den vorher genannten Vorteil zunutze. Die Dreiecke werden so angeordnet, dass möglichst viel Fläche nach außen gerichtet ist. So ist mehr Platz in der Mitte und die Behälter können näher aneinanderrücken. Start- und Endbehälter existieren jeweils nur einmal.

Die Mittelbehälter befinden sich zwischen Start- und Endbehälter. Die Dreiecke werden so angeordnet, dass möglichst viel Fläche der Dreiecke nach oben zeigt.

Anlegen an die Gerade

Da jedes Dreieck an der geraden Straße anliegen muss, lohnt es sich herauszufinden, welche Positionierung am effizientesten ist.



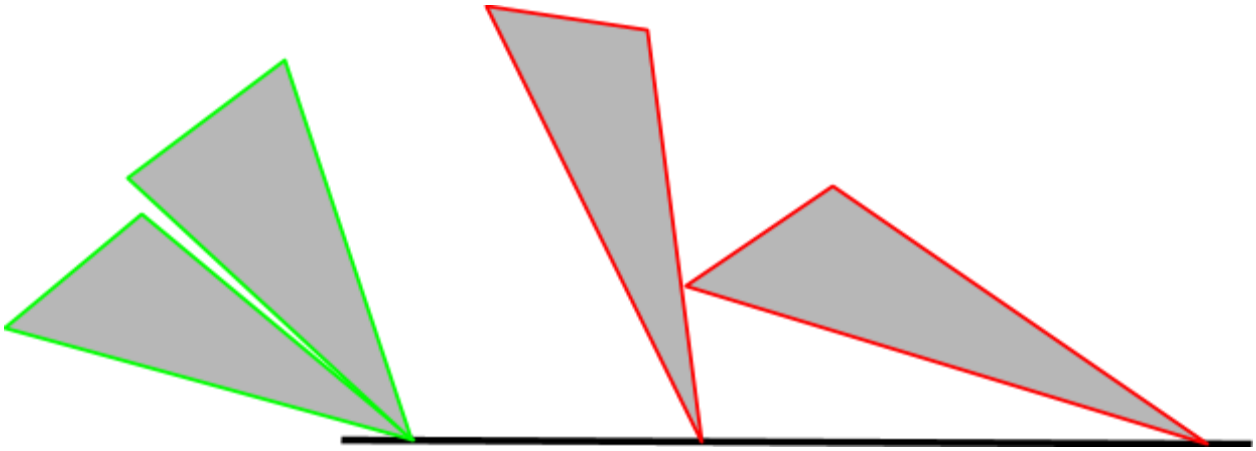
Erstellt man eine Skizze, sieht man, dass man am meisten Platz spart, wenn der spitzeste Winkel an der Geraden anliegt. So lässt man Raum für andere Dreiecke.

Jedes Dreieck sollte also stets mit dem spitzesten Winkel an der Geraden anliegen.

Sortierung nach Länge der Seiten

Dadurch, dass ich nun mehrere Typen von Behältern nutze, kann ich diese auch unterschiedlich befüllen. Im Gegensatz zum Standardverfahren kann ich Dreiecke nun unterschiedlich auf die 3 verschiedenen Behältertypen verteilen.

Den Vorteil durch Überhang von Start- und Endbehältern kann ich am meisten nutzen, indem ich ihnen Dreiecke, die potenziell am meisten Platz einnehmen zuordne.



Dabei ist zu beachten, dass nicht die tatsächliche Fläche zählt. Es zählt vielmehr die Länge der Seiten des Dreiecks. Ein Dreieck mit einer langen Seite, kann vielmehr Platz einnehmen als eines mit kleineren Seiten. Für solche Dreiecke wäre also eine Positionierung am Rand (grün) deutlich vorteilhafter als eine Positionierung in der Mitte (rot).

Um die Dreiecke zuordnen zu können bietet sich das Sortieren nach der Länge der Seiten des Dreiecks an.

Abschätzung der Behälteranzahl

Um die Dreiecke noch spezifischer sortieren zu können, kann man die Behälteranzahl im Voraus berechnen. Da jedes Dreieck mit dem spitzesten Winkel an die Gerade angelegt wird, können wir die spitzesten Winkel aller Dreiecke summieren und durch 180° teilen. So erhalten wir die Mindestanzahl an Behältern, denn nicht alle Behälter müssen komplett gefüllt werden.

Wir können auch berechnen, ob Mittelbehälter notwendig sind. Dazu füllen wir testweise zwei Behälter. Sind danach noch Dreiecke übrig wissen wir, dass ein Mittelbehälter notwendig ist.

Zuordnung der Dreiecke

Wenn wir nun die Dreiecke nach ihrer längsten Seite sortiert haben, können wir anfangen diese zuzuordnen.

Reicht ein einziger Behälter aus, so wird dieser einfach befüllt.

Reichen Start- und Endbehälter aus, um alle Dreiecke zu fassen, so werden den 2 Behälter abwechselnd Dreiecke hinzugefügt.

Müssen Mittelcontainer benutzt werden, so werden zunächst Start- und Endbehälter bis auf maximal 90° gefüllt, um die meisten Dreiecke mit langen Seiten loszuwerden. Im nächsten Schritt werden die restlichen Dreiecke gleichmäßig auf alle Behälter verteilt. Es kommen nun so viele Mittelbehälter dazu wie im Voraus abgeschätzt wurden. Sind trotzdem alle Behälter gefüllt, wird ein neuer Mittelbehälter hinzugefügt.

Permutation

Wenn nun alle Behälter befüllt worden sind, werden alle möglichen Anordnungen (Permutationen) der Mittelbehälter berechnet. Man erhält eine Liste von möglichen Anordnungen der Behälter. Start- und Endbehälter werden nicht miteinbezogen. Das Einbeziehen der Start- und Endbehälter ist nicht zielführend, denn diese sind dafür ausgelegt die Dreiecke nach außen Überhängen zu lassen. Sie sind schlechte Mittelbehälter, da sie zu viel Platz einnehmen. Zusätzlich spart man so an Permutationen und somit weiteren Berechnungen.

Annäherung

Die verschiedenen Anordnungen von Behältern müssen nun alle aneinander angenähert werden. Das heißt, mithilfe von Kollisionsabfrage die Behälter so nah wie möglich aneinander zu positionieren. Dies ist am einfachsten mit Matrizenrechnung möglich.

UMSETZUNG

Einlesen der Beispieldatei

Bevor die Berechnung starten kann, muss die entsprechende Beispieldatei eingelesen werden. Wenn alle Dreiecke eingelesen wurden, werden Dreieck-Objekte erstellt, welche die Eckpunkte, Strecken und Winkel des Dreiecks berechnen und speichern.

Dreieck-Objekte

Bevor das Dreieck korrekt transformiert werden kann, müssen die Punkte korrekt gesetzt werden. Für spätere Berechnungen ist es wichtig, dass das Dreiecke rechtsdrehend ist, also die Beschriftung der Punkte gegen der Uhrzeigersinn verläuft.

```
private void SetzePunkte(Point _A, Point _B, Point _C)
{
    // Punkt A ist immer Punkt A
    A = _A;

    // Determinante bestimmen
    Vector AB = _B - _A;
    Vector AC = _C - _A;

    double Determinante = Vector.Determinant(AB, AC);
    if(Determinante == 0)
    {
        MessageBox.Show("Invalides Dreieck");
        return;
    }

    // Bestimmen ob B und C getauscht werden müssen
    if (Determinante < 0)
    {
        B = _C;
        C = _B;
    }
    else
    {
        B = _B;
        C = _C;
    }
}
```

Ist die Determinante kleiner 0, dreht das Dreieck falsch. Um dies zu korrigieren, werden die Punkte B und C getauscht.

```
private void SetzePunkteNachWinkel()
{
    // Punkt A auf Punkt mit kleinstem Winkel setzen
    if(WB < WA && WB < WC)
    {
        // Punkt B hat kleinsten Winkel
        SetzePunkte(B, A, C);
    }
    else if(WC < WA && WC < WB)
    {
        // Punkt C hat kleinsten Winkel
        SetzePunkte(C, A, B);
    }
}
```

Punkt A wird dabei immer so gesetzt, dass A der Punkt ist, an dem der spitzeste Winkel liegt. So kann das Programm immer davon ausgehen, dass A an der Gerade anliegt und der Winkel im Punkt A der spitzeste ist. Dies ist wichtig für spätere Berechnungen wie z. B. das Drehen.

Transformationen

Verschieben

Um das Dreieck zu Verschieben wird auf jeden Eckpunkt der Vektor der Verschiebung aufaddiert.

Drehen

```
public void Drehen(double Winkel)
{
    // Gradmaß in Bogemaß umrechnen
    Winkel *= Math.PI / 180;

    // Punkt speichern
    Point A_Alt = A;

    // Zum Ursprung verschieben
    Point Verschiebung = A;
    Verschiebung.X *= -1;
    Verschiebung.Y *= -1;
    Verschieben(new Vector(Verschiebung.X, Verschiebung.Y));

    // Drehen mit Drehmatrix
    double MA = Math.Cos(Winkel);
    double MB = -Math.Sin(Winkel);
    double MC = Math.Sin(Winkel);
    double MD = Math.Cos(Winkel);

    TMatrix M = new TMatrix(MA, MB, MC, MD);

    Vector VA = M.VektorMult(new Vector(A.X, A.Y));
    Vector VB = M.VektorMult(new Vector(B.X, B.Y));
    Vector VC = M.VektorMult(new Vector(C.X, C.Y));

    A = new Point(VA.X, VA.Y);
    B = new Point(VB.X, VB.Y);
    C = new Point(VC.X, VC.Y);

    // Zurück verschieben
    Verschieben(new Vector(A_Alt.X, A_Alt.Y));
}
```

Das Drehen wird mit einer Matrizenrechnung realisiert. Diese dreht alle Eckpunkte um einen bestimmten Winkel im Bogenmaß um den Ursprung.

Da wir allerdings nicht um den Ursprung, sondern um Punkt A, der an der Gerade anliegt drehen wollen, müssen wir das Dreieck zuerst zum Ursprung verschieben. Ist die Drehung abgeschlossen wird das Dreieck wieder zurückverschoben.

Spiegeln

```
public void Spiegeln()
{
    Point CAlt = C;
    Point BAlt = B;

    double XDistAB = A.X - B.X;
    double XDistAC = A.X - C.X;

    B = new Point(A.X + XDistAC, CAlt.Y);
    C = new Point(A.X + XDistAB, BAlt.Y);

    BerechneStrecken();
    BerechneWinkel();

    Gespiegelt = !Gespiegelt;
}
```

Beim Spiegeln wird das Dreieck horizontal gespiegelt. Die Spiegelachse verläuft dabei in positiver y-Richtung, an der x-Position des Punktes A.

Hier wird die Distanz zur Spiegelachse auf der x-Achse berechnet und ein neuer Punkt gegenüber, mit dem gleichen x-Abstand erstellt.

Zuordnung der Dreiecke

Ordnen nach Seitenlänge

```
private List<TDreieck> OrdneNachKleinsterStrecke(List<TDreieck> Liste)
{
    return Liste.OrderBy(D => D.LaengsteStrecke).ToList();
}
```

Alle Dreieck-Objekte besitzen den Wert “LaengsteStrecke”. Dieser Wert entspricht logischerweise der längsten Strecke, die sie besitzen.

Nach diesem Wert werden die Dreiecke geordnet. Es ist später effizienter vom Ende der Liste Einträge zu entfernen. Deswegen ordnen wir hier nach der kleinsten Strecke, damit die längsten Strecken am Ende der Liste stehen.

1 Behälter ausreichend

Reicht ein Behälter aus, so wird dieser einfach mit allen Dreiecke befüllt.

```
// Handeln nach Gesamtwinkel
if (GesamtWinkel < 180.0)
{
    // 1 Behaelter ausreichend
    TBehaelter B = new TBehaelter();
    B.BehaelterTyp = TBehaelter.TBehaelterTyp.Startbehaelter;

    _Behaelter.Add(B);

    while (DreieckeUnzugeteilt.Count > 0)
    {
        TDreieck D = DreieckeUnzugeteilt[DreieckeUnzugeteilt.Count - 1];
        _Behaelter[0].DreieckHinzufügen(D);
        DreieckeUnzugeteilt.RemoveAt(DreieckeUnzugeteilt.Count - 1);
    }
}
```

2 Behälter ausreichend

Dies wird ermittelt, indem testweise 2 Behälter befüllt werden (siehe: Idee - Abschätzen der Behälteranzahl).

Reichen Start- und Endbehälter aus, so werden diese abwechselnd befüllt.

```
else
{
    // Start- und Endbehälter
    TBehaelter BStart = new TBehaelter();
    TBehaelter BEnde = new TBehaelter();
    BStart.BehaelterTyp = TBehaelter.TBehaelterTyp.Startbehaelter;
    BEnde.BehaelterTyp = TBehaelter.TBehaelterTyp.Endbehaelter;

    _Behaelter.Add(BStart);
    _Behaelter.Add(BEnde);

    // Abwechselnd Dreiecke zu Start- und Endbehälter hinzufügen (Dreiecke mit größter Strecke zuerst)
    bool InStartcontainer = true;
    while (DreieckeUnzugeteilt.Count > 0)
    {
        TDreieck D = DreieckeUnzugeteilt[DreieckeUnzugeteilt.Count - 1];

        if (InStartcontainer)
        {
            BStart.DreieckHinzufügen(D);
        }
        else
        {
            BEnde.DreieckHinzufügen(D);
        }

        DreieckeUnzugeteilt.RemoveAt(DreieckeUnzugeteilt.Count - 1);

        // Flag umkehren
        InStartcontainer = !InStartcontainer;
    }
}
```

Hier werden solange Dreiecke abwechselnd den Behälter zugeteilt, bis keine mehr übrig sind. Welchem Behälter das Dreieck zugeordnet wird, entscheidet der ständig wechselnde Boolean-Wert “InStartcontainer”.

Mehr als 2 Behälter notwendig

Reichen Start- und Endbehälter nicht aus wird die abgeschätzte Anzahl an Mittelbehältern hinzugefügt.

```
// 2 oder mehr Behaelter notwendig?
if (BerechnenMittelbehaelterNoetig(Dreiecke.ToList()))
{
    // Start-, Mittel- und Endbehälter
    TBehaelter BStart = new TBehaelter();
    TBehaelter BEnde = new TBehaelter();
    BStart.BehaelterTyp = TBehaelter.TBehaelterTyp.Startbehaelter;
    BEnde.BehaelterTyp = TBehaelter.TBehaelterTyp.Endbehaelter;

    _Behaelter.Add(BStart);
    _Behaelter.Add(BEnde);

    // Mittelbehälterzahl (ungefähr - im worst case sind es mehr)
    int BehaelterZahlMindestens = (int)Math.Ceiling(GesamtWinkel / 180.0);
    int MittelbehaelterZahlMindestens = BehaelterZahlMindestens - 2;

    if(MittelbehaelterZahlMindestens > 0)
    {
        for(int I = 0; I < MittelbehaelterZahlMindestens; I++)
        {
            TBehaelter BMittel = new TBehaelter();
            BMittel.BehaelterTyp = TBehaelter.TBehaelterTyp.Mittelbehaelter;

            _Behaelter.Insert(1,BMittel);
        }
    }
```

Nun werden als erster Schritt nur Start- und Endbehälter auf 90° gefüllt, um Dreiecke mit den längsten Strecken loszuwerden.

```
// Abwechselnd Start- und Endbehälter auf 90° auffüllen (Dreiecke mit größter Strecke zuerst)
bool InStartcontainer = true;
double WinkelStart = 0.0;
double WinkelEnd = 0.0;
while ((WinkelStart < 90.0) && (WinkelEnd < 90.0) && DreieckeUnzugeteilt.Count > 0)
{
    TDreieck D = DreieckeUnzugeteilt[DreieckeUnzugeteilt.Count - 1];

    if (InStartcontainer)
    {
        WinkelStart += D.WA;

        if (WinkelStart > 90.0)
        {
            continue;
        }

        BStart.DreieckHinzufügen(D);
    }
    else
    {
        WinkelEnd += D.WA;

        if (WinkelEnd > 90.0)
        {
            continue;
        }

        BEnde.DreieckHinzufügen(D);
    }

    DreieckeUnzugeteilt.RemoveAt(DreieckeUnzugeteilt.Count - 1);

    // Flag umkehren
    InStartcontainer = !InStartcontainer;
}
```

Im Anschluss werden die restlichen Dreiecke auf alle noch nicht gefüllten Behälter gleichmäßig verteilt.

```
// Temporären Winkel setzen
foreach (TBehaelter B in _Behaelter)
{
    B.TempWinkel = 0.0;
}

// Start- und Endbehälter sind bereits teilweise gefüllt
BStart.TempWinkel = BStart.GefuellterWinkel();
BEnde.TempWinkel = BEnde.GefuellterWinkel();

// Restliche Dreiecke auch unter Mittelbehältern zuteilen
int Index = 0;
int Fehlzaehler = 0;
while (DreieckeUnzugeteilt.Count > 0)
{
    TBehaelter B = _Behaelter[Index];
    TDreieck D = DreieckeUnzugeteilt[DreieckeUnzugeteilt.Count - 1];

    // Dreieck hinzufügen wenn noch Platz ist
    B.TempWinkel += D.WA;

    if (B.TempWinkel < 180.0)
    {
        // Dreieck hinzufügen
        B.DreieckHinzufügen(D);

        DreieckeUnzugeteilt.RemoveAt(DreieckeUnzugeteilt.Count - 1);

        Fehlzaehler = 0;
    }
    else
    {
        Fehlzaehler++;
    }

    // Falls die Behälter nicht ausreichen wird ein neuer hinzugefügt
    if(Fehlzaehler >= _Behaelter.Count)
    {
        TBehaelter BNeu = new TBehaelter();
        BNeu.BehaelterTyp = TBehaelter.TBehaelterTyp.Mittelbehaelter;
        BNeu.TempWinkel = 0;

        _Behaelter.Insert(1,BNeu);
    }

    // Nächste Iteration
    Index++;
    if (Index >= _Behaelter.Count)
    {
        Index = 0;
    }
}
```

Falls die Behälter nicht ausreichen sollten (Die Anzahl an Mittelbehältern ist eine Mindestanzahl), werden weiter Behälter hinzugefügt. Dies wird umgesetzt mit einem “Fehlzähler”, welcher jedem

vollen Behälter um eins hochzählt. Sind keine freien Behälter mehr verfügbar, so zeigt der Zähler die Anzahl an Behältern. Ein neuer Behälter wird hinzugefügt.

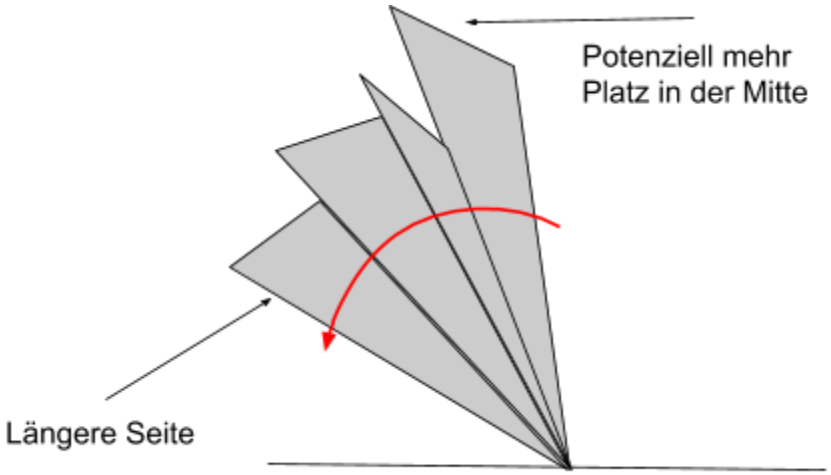
```
private bool AnordnungsFlag = true;
public void DreieckHinzufügenMittelbehaelter(TDreieck D)
{
    D.Verschieben(TempPos - new Point(D.A.X, D.A.Y));

    if (AnordnungsFlag)
    {
        Dreiecke.Insert(0, D);
    }
    else
    {
        Dreiecke.Add(D);
    }

    AnordnungsFlag = !AnordnungsFlag;
}
```

Wichtig ist hierbei, bei den Mittelbehältern die Dreiecke so hinzugefügt werden, dass Dreiecke mit langer Seite in der Mitte und Dreiecke mit kleineren Seiten links und rechts positioniert werden.

Anordnung der Dreiecke im Behälter

Um die Dreiecke möglichst effizient anzuordnen, werden die Dreiecke je nach Behältertyp anders angeordnet.

Im Start- und Endbehälter werden die Dreiecke nach außen gedreht, dabei wird durch spiegeln der Dreiecke versucht, die längere Seite in Drehrichtung zeigen zu lassen.



So lässt man noch ein wenig Platz für andere Dreiecke in den Mittelbehältern.

```
private void AnordnenStartbehaelter()
{
    const double Epsilon = 10E-10;

    Vector AnlegGerade = new Vector(-1, 0); // Horizontale Gerade als Start

    foreach (TDreieck D in Dreiecke)
    {
        // Winkel zum Anlegen berechnen
        Vector AC = D.C - D.A;
        double WinkelZuGerade = Vector.AngleBetween(AnlegGerade, AC);

        D.Drehen(-WinkelZuGerade);

        AC = D.C - D.A;
        WinkelZuGerade = Vector.AngleBetween(AnlegGerade, AC);

        if (WinkelZuGerade > Epsilon)
        {
            D.Drehen(-WinkelZuGerade);
        }

        Vector AB = D.B - D.A;
        AnlegGerade = AB;
    }
}
```

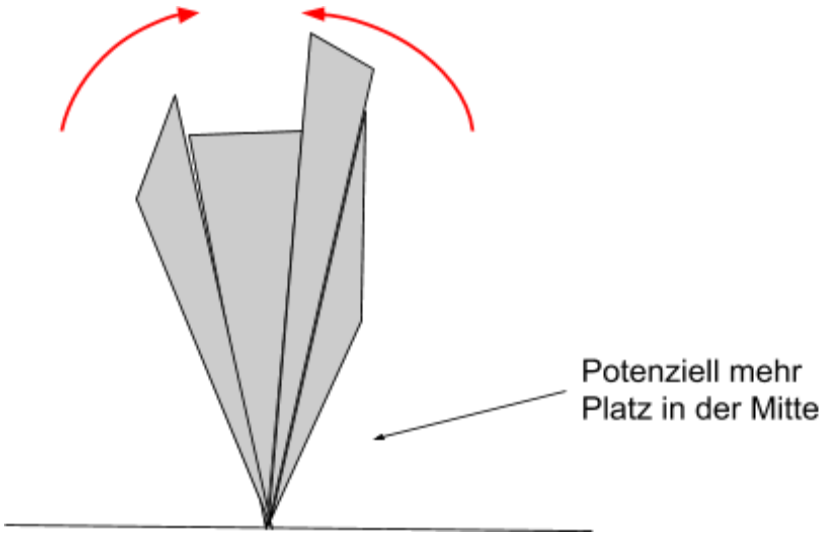
Drehen:

Zuerst wird bestimmt um wie viel Grad, das Dreieck gedreht werden muss. Dafür nutze ich die Methode: “Vector.AngleBetween”, welche mir bei korrekt gedrehten Dreiecken den gewünschten Winkel liefert.

Es wird der Winkel zwischen der untern Seite und dem Anlegvektor (zu beginn die x-Achse) berechnet.

Um diesen Winkel wird das Dreieck gedreht und die nach oben liegende Seite des Dreiecks wird zum neuen Anlegvektor.

In den Mittelbehältern wird die linke Seite der Dreiecke nach rechts und die rechte Seite der Dreiecke nach links gedreht. Somit werden die Dreiecke zur Mitte hin gedreht.



Permutationen

Um alle möglichen Kombinationen von Mittelbehältern zu berechnen, nutze ich eine rekursive Methode namens “BehaelterPermutationen”. Diese gibt mir eine Liste an Anordnungen zurück.

```
private IEnumerable<IEnumerable<TBehaelter>> BehaelterPermutationen(IEnumerable<TBehaelter> list, int length)
{
    if (length == 1) return list.Select(t => new TBehaelter[] { t });
    return BehaelterPermutationen(list, length - 1)
        .SelectMany(t => list.Where(o => !t.Contains(o)),
            (t1, t2) => t1.Concat(new TBehaelter[] { t2 }));
}
```

Sind weniger als 4 Behälter notwendig, ist keine Permutation notwendig, da zwei abgezogen werden (Start- und Endbehälter werden nicht inbegriffen) und mindestens zwei Behälter existieren müssen, damit es Kombinationen geben kann. In diesem Fall wird nur einmal angenähert und die Distanz berechnet (siehe unten)

Annäherung

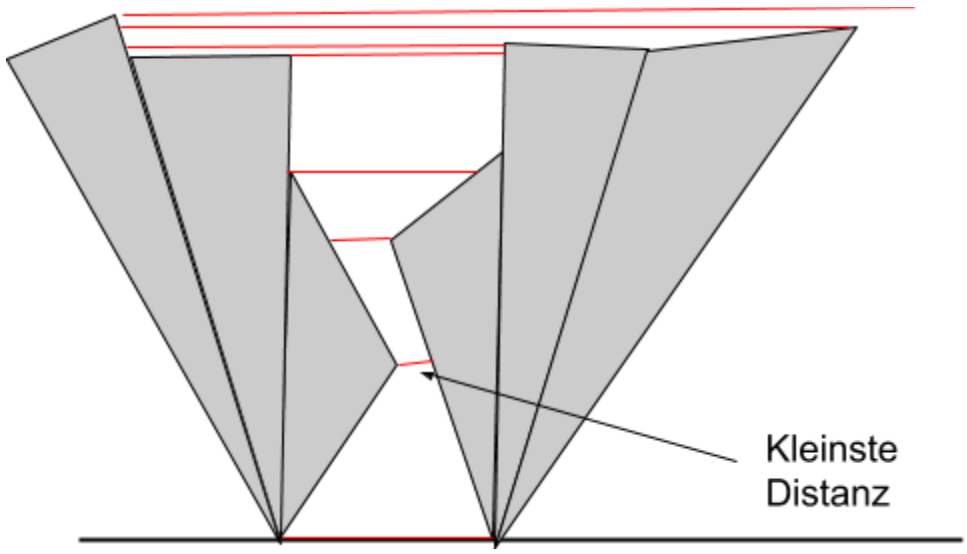
Jede mögliche Anordnung wird nun mit einem Berechnungs-Objekt angenähert. Das Berechnungs-Objekt speichert die Anordnung und nähert die Behälter an. Letztendlich gibt das Objekt nur noch die Distanz zurück.

```
TBehaelter[] BesteAnordnung;
double BesteStrecke;
if(_Behaelter.Count > 3) // Ab 3 sind Permuatation möglich (2 Behälter werden abgezogen)
{
    // Permuatationen ohne Start- und Endbehälter
    List<TBehaelter> PermuatationsBehaelter = new List<TBehaelter>(_Behaelter);
    PermuatationsBehaelter.RemoveAt(0);
    PermuatationsBehaelter.RemoveAt(PermuatationsBehaelter.Count - 1);

    // Alle möglichen Kombinationen bestimmen
    var PermutationenRoh = BehaelterPermutationen(PermuatationsBehaelter, PermuatationsBehaelter.Count).ToArray();

    List<TBehaelter[]> Permutationen = new List<TBehaelter[]>();

    double[] Ergebnisse = new double[PermutationenRoh.Length];
    for (int I = 0; I < PermutationenRoh.Length; I++)
    {
        List<TBehaelter> PermutationenUnvollstaendig = PermutationenRoh[I].ToList();
        PermutationenUnvollstaendig.Insert(0, _Behaelter[0]);
        PermutationenUnvollstaendig.Add(_Behaelter[_Behaelter.Count - 1]);

        TBehaelter[] PermutationenVollstaendig = PermutationenUnvollstaendig.ToArray();

        Permutationen.Add(PermutationenVollstaendig);

        TBerechnung Berechnung = new TBerechnung(PermutationenVollstaendig);
        Ergebnisse[I] = Berechnung.Berechnen();
    }

    int BestePermuatationIndex = -1;
    for (int I = 0; I < Ergebnisse.Length; I++)
    {
        if (BestePermuatationIndex == -1 || Ergebnisse[I] < Ergebnisse[BestePermuatationIndex])
        {
            BestePermuatationIndex = I;
        }
    }

    // Beste Permutation anzeigen
    TBehaelter[] BestePermutation = Permutationen[BestePermuatationIndex].ToArray();
    BestePermutation[0].Verschieben(new Vector(-(BestePermutation[0].TempPos.X - 200), 0));

    TBerechnung BerechnungBeste = new TBerechnung(BestePermutation);
    BerechnungBeste.Berechnen();

    BesteAnordnung = BestePermutation;
    BesteStrecke = Ergebnisse[BestePermuatationIndex];
}
```

Die Annäherung passiert in mehreren Schritten. Die Behälter werden nacheinander angenähert. Zuerst wird der 1. und 2. Behälter angenähert, dann der 2. und der 3. usw.

```
public double Berechnen()
{
    // Behälter verschieben
    for (int I = 0; I < Behaelter.Length - 1; I++)
    {
        BehaelterAnnaehern(Behaelter[I], Behaelter[I + 1]);
    }

    double EntfernungFinal;
    if (Behaelter.Length < 2)
    {
        EntfernungFinal = 0;
    }
    else
    {
        EntfernungFinal = Behaelter[Behaelter.Length - 1].TempPos.X - Behaelter[0].TempPos.X;
    }

    return EntfernungFinal;
}
```

Die Annäherung an sich wird mit einer Kollisionsabfrage realisiert. Jeder Eckpunkt des Dreiecks erstellt einen virtuellen Strahl, welcher in Richtung des anderen Behälters geht. Gleiches mit dem andern Behälter in Gegenrichtung. Die Distanz vom Eckpunkt zum ersten Schnittpunkt ist die Distanz, um die sich der Behälter bewegen kann bis er diesen Punkt schneidet. Damit die Behälter sich nicht überschneiden, wird die kleinste aller dieser Distanzen genommen. Sie ist die maximale Distanz um die sich der Behälter bewegen kann, ohne zu kollidieren.



Die Kollisionsabfrage wird mit Matrizenrechnung realisiert. Diese berechnet den Schnittvektor, der beiden Geraden. Sind die Parameter zwischen 0 und 1 schneiden sich die beiden Strecken.

```
private bool Kollision(Point A, Point B, Point P, Point Q, out Vector ST)
{
    Vector BA = new Vector(A.X - B.X, A.Y - B.Y);
    Vector PQ = new Vector(Q.X - P.X, Q.Y - P.Y);
    Vector PA = new Vector(A.X - P.X, A.Y - P.Y);

    TMatrix M = new TMatrix(PQ.X, BA.X, PQ.Y, BA.Y);

    if (M.Determinante() == 0.0)
    {
        ST = new Vector();
        return false;
    }

    Vector X = M.Invertiert().VektorMult(PA);

    double s = X.X;
    double t = X.Y;

    ST = new Vector(s, t);

    const double Eps = 1E-12;

    if ((s < 0.0) || (t <= Eps) || (s > 1.0) || (t >= 1.0 - Eps)) { return false; }

    return true;
}
```

Berechnung der Distanz

Die Berechnung der Distanz ist recht einfach. Die Differenz der Position des Endbehälters und der Position des Startbehälters entspricht der Gesamtdistanz.

Die Gesamtdistanz wird gerundet in Metern ausgegeben.

Abspeichern als neue Beispieldatei

Die Reihenfolge in den Behältern ist bereits korrekt. Daher müssen nur alle Dreiecke von jedem Behälter nacheinander in Form der Beispieldatei gebracht werden.

Qualität & Verbesserungen

Das Lösen der Aufgabe auf Grundlage des Behälterproblems ist eine Annäherung. Es ist durchaus möglich mit einer anderen Herangehensweise eine bessere Lösung zu finden. Auch mit einer anderen Lösung des Behälterproblems könnte durchaus noch eine bessere Lösung möglich sein, als die meines Programms.

Das Programm erfüllt aber seinen Zweck voll und ganz. Es liefert eine Lösung, welche sich nah an der Optimallösung befindet und das vor allem in einer extrem kurzen Zeit.

Eine komplette Suche nach der optimalen Lösung würde deutlich länger dauern.

Komplexität

Um die Komplexität zu bestimmen, muss man das Programm in seine Einzelschritte unterteilen:

Nehmen wir an, es gibt N Dreiecke und durchschnittlich 5 pro Behälter.

Zuerst werden die Dreiecke nach ihrer längsten Seite sortiert. Ich benutze die List.Sort Methode, welche eine N*log N Operation darstellt.

Das Füllen der Behälter ist eine normale N Operation.

Das Annähern der Behälter ist jeweils eine N^2 Operation, doch diese wird für jede mögliche Kombination von Behältern durchgeführt.

Nehmen wir an, wir haben B Behälter, so gibt es !B Kombinationen. Der Gesamtaufwand für das Annähern der Behälter ist also !B * N^2

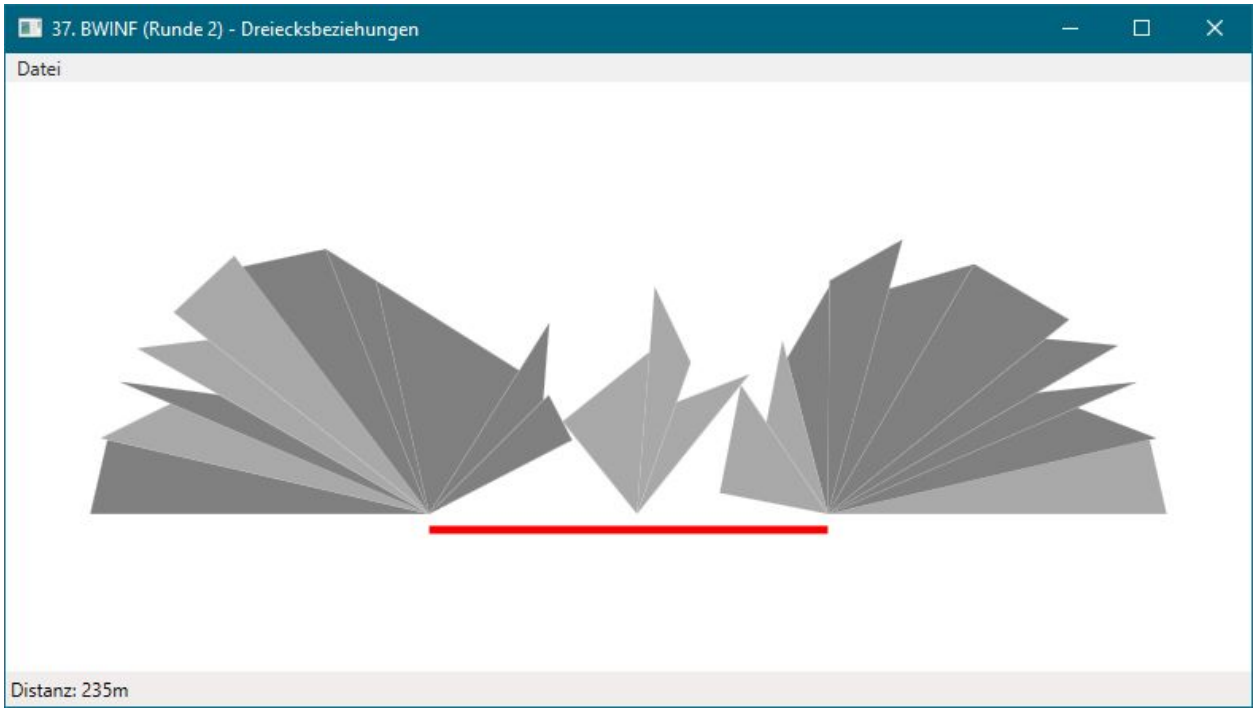
Insgesamt: $O(N) = !B * N^2$

Diese Komplexität ist bei den Beispieldateien kaum ein Problem. Im Vergleich eine “BruteForce” Herangehensweise würde eine Komplexität von $O(N) = !N$ bedeuten.

BEDIENUNG

Das Programm ist mit seiner visuellen Oberfläche leicht zu bedienen. Um eine Beispieldatei zu öffnen, klickt man in der Menüleiste (oben) auf “Datei” und danach im Kontextmenü auf “Öffnen”. Im Dateidialog wählt man nun die gewünschte Beispieldatei (*.txt).

Die Berechnung startet sofort.



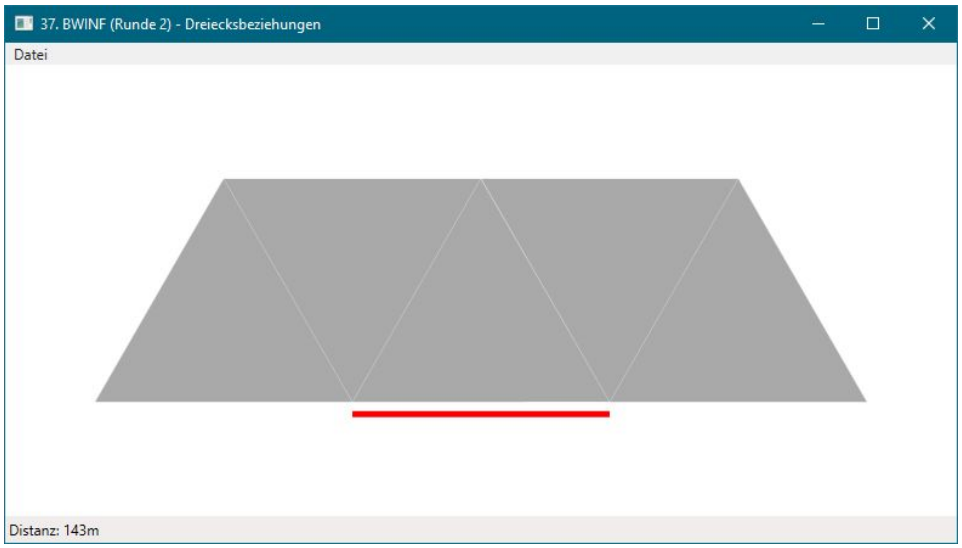
In der Mitte des Fensters wird nun die optimale Anordnung von Dreiecken angezeigt. Gespiegelte Dreiecke werden in dunklem Grau angezeigt. Die Distanz ist mit einem roten Balken markiert. Die Distanz in Metern ist unten in der Statusleiste zu sehen.

Möchte man nun diese Reihenfolge sich ausgeben lassen, wählt man erneut “Datei”, dann aber “Speichern unter”. Diese Option speichert eine neue Beispieldatei mit der berechneten Anordnung von Dreiecken. Die Rundung auf Ganzzahlen ist ebenfalls möglich.

ERGEBNISSE

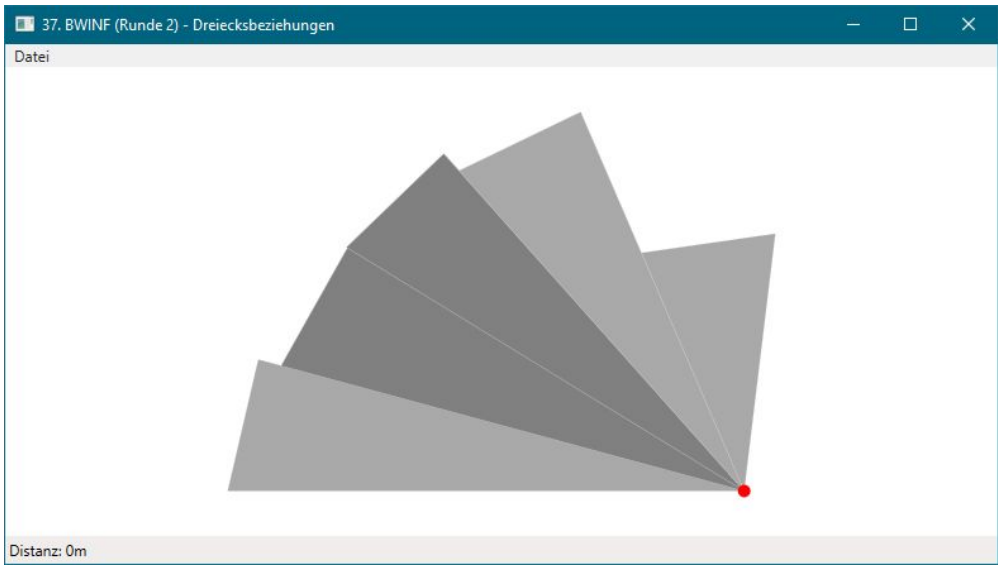
Beispiel 1 (Distanz: 143m):

Die Dreiecke passen perfekt ineinander.



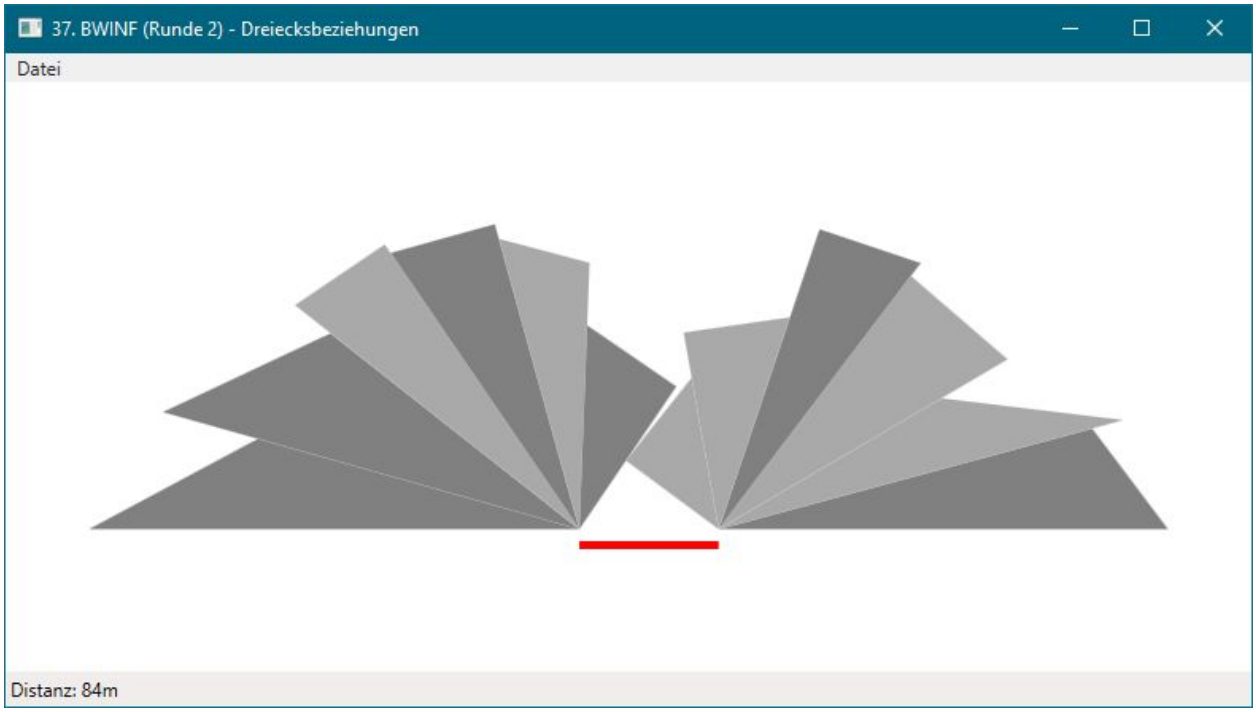
Beispiel 2 (Distanz: 0m):

Ein Behälter reicht aus. Alle Dreiecke können in einem Punkt gesammelt werden.



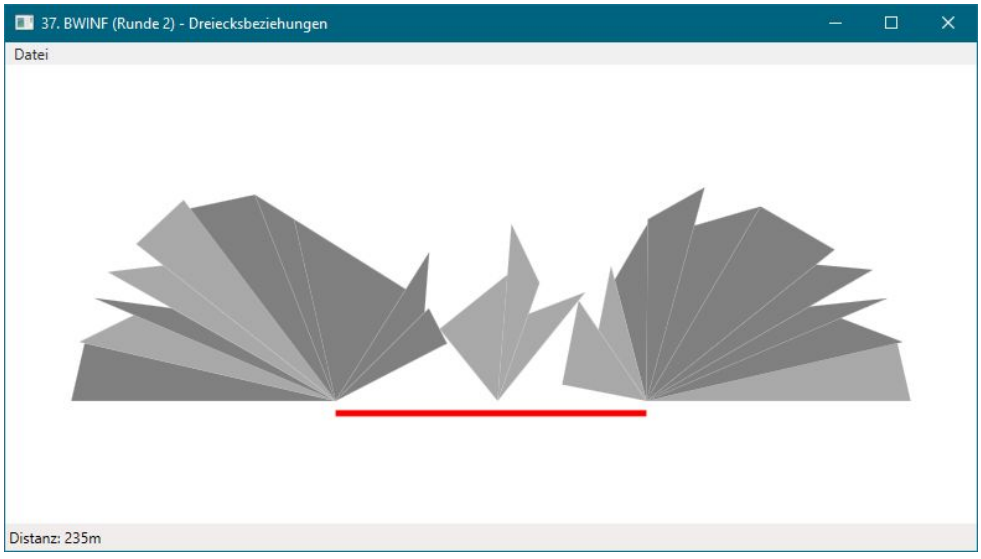
Beispiel 3 (Distanz: 84m):

Hier sind zwei Behälter erforderlich.



Beispiel 4 (Distanz: 235m):

Dies ist die erste Aufgabe, welche einen Mittelbehälter benötigt. Trotzdem ist die Lösung lange nicht perfekt.



Beispiel 5 (Distanz: 747m):

“BruteForce” Verfahren haben bei dieser Aufgabe Probleme. Mit Behältern ist die Lösung aber eine gute Annäherung.