Aufgabe 2: Dreiecksbeziehungen

Teilnahme-Id: 50997
Bearbeiter/-in dieser Aufgabe:
Jan Niklas Groeneveld

27. April 2019

Inhaltsverzeichnis

1 Lösungsidee

Um den Abstand entlang der Straße der Dreiecke minimieren zu können, müssen zunächst die Regeln für die gültigen Anordnungen erarbeitet werden, bevor der eigentliche Algorithmus erläutert wird, der auf Grundlage dieser Regeln eine Anordnung mit geringem Abstand ermittelt. Dabei wird sich zeigen, dass davon Abstand genommen werden muss, die mathematisch beweisbar beste Lösung zu ermitteln, weil einerseits die implementierten Regeln für die Dreiecksanordnungen auf Annahmen basieren, und andererseits der Algorithmus bei steigender Zahl von Dreiecken nicht alle regelkonformen Anordnungen ausprobieren kann. Folglich wird ein heuristisches Lösungsverfahren gesucht.

1.1 Standardisierung der Speicherung und Anordnung der Dreiecke

Um die einzelnen Anordnungen standardisiert nutzen zu können, müssen zunächst einige Konventionen diesbezüglich getroffen werden, nach denen die Dreiecke nach dem Auslesen aus der Textdatei gespeichert werden:

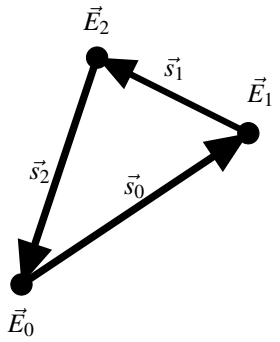

Abbildung 1: Die Nummerierung der Ecken und Kanten

1. Sämtliche Dreiecke sind gegen den Uhrzeigersinn nummeriert.
2. Als nullte Ecke wird diejenige definiert, die den geringsten Innenwinkel aufweist. Wenn zwei Ecken den gleichen Innenwinkel besitzen, ist es nicht weiter entscheidend, welche von beiden genommen wird.
3. Die Strecken sind ebenfalls gegen den Uhrzeigersinn orientiert. Sie werden als Vektor der Differenz zweier aufeinanderfolgender Ecken berechnet (Beispiel: $\vec{s}_0 = \vec{E}_1 - \vec{E}_0$). Effektiv genutzt werden jedoch nur die Beträge der Seitenvektoren.
4. Zur eindeutigen Definition wird zusätzlich der Innenwinkel der nullten Ecke herangezogen (α). Damit sind alle Dreiecke über den Kongruenzsatz Seite-Winkel-Seite definiert.

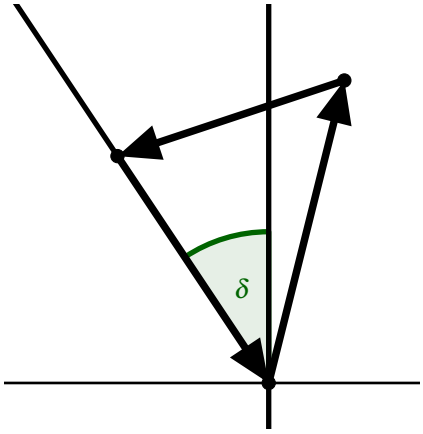

Abbildung 2: Die Definition des Winkels δ

Das eigentliche Positionieren der Dreiecke erfolgt durch ein festgelegtes Vorgehen, das die Daten des Dreiecks, den Winkel δ, unter dem die am linkesten liegende Seite des Dreiecks bezüglich einer Orthogonalen der Straße liegt, die Verschiebung der nullten Ecke in x-Richtung (e_x) und die Verschiebung der nullten Ecke in y-Richtung (e_y) benötigt. Der Winkel wird dabei so festgelegt, dass solche, die sich von der Orthogonalen zur Straße nach links abspreizen, einen negativen Wert besitzen, während die nach rechts abspreizenden als positiv festgelegt werden. Winkel, die einen Wert kleiner als $-90°$ oder größer als $90°$ haben, widersprechen der Aufgabenstellung und sind somit verboten. Hier wird bereits deutlich, dass sich die anzuordnenden Dreiecke in zwei Typen unterteilen lassen: solche, deren nullte Ecke auf der Straße liegen, und solche, bei denen eine ihr gegenüberliegende Ecke an die Straße grenzt. Diese zwei Typen sind im Folgenden zu unterscheiden.

Der Algorithmus, der ein Dreieck nach diesen Kriterien positioniert, kann recht einfach gehalten werden: Nachdem die nullte Ecke auf ihre Position gebracht wurde, gelten für die Positionen der beiden anderen Ecken:

$$E_{1x} = |s_0| \sin \eta + e_x \tag{1}$$

$$E_{1y} = |s_0| \cos \eta + e_y \tag{2}$$

$$E_{2x} = |s_2| \sin \vartheta + e_x \tag{3}$$

$$E_{2y} = |s_2| \cos \vartheta + e_y \tag{4}$$

Dabei sind η und ϑ die Winkel, unter denen die Ecken $\vec{E}_1$ und $\vec{E}_2$ bezüglich der Orthogonalen stehen sollen. Hier muss jedoch beachtet werden, dass diese Winkel abhängig davon, ob die nullte Ecke oder eine andere auf der Straße steht, unterschiedlich berechnet werden müssen. Liegt die nullte Ecke auf der Straße, gilt: $\vartheta = \delta$ und $\eta = \delta + \alpha$. Wenn

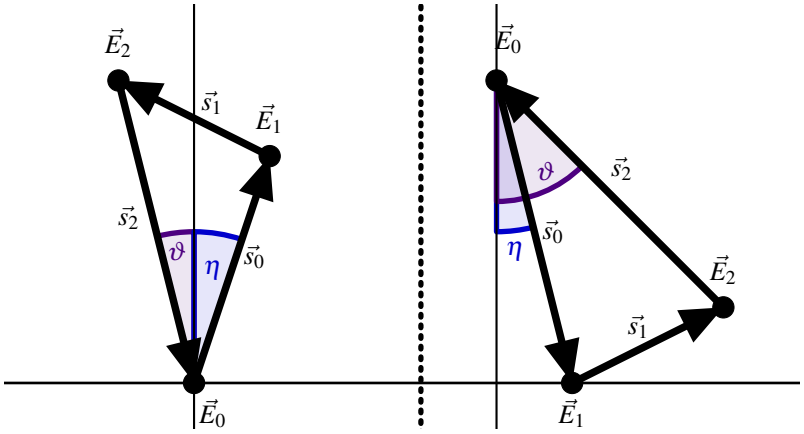

Abbildung 3: Anordnung mit nullter Kante an der Straße (links) und anderer Ecke an der Straße (rechts)

jedoch eine der beiden anderen Ecken auf der Straße liegt, können die Winkel η und ϑ kleiner sein als $-90°$ oder größer als $90°$, weswegen sie durch folgende geometrische Überlegung bestimmt werden: $\eta = \delta + 180°$, da jetzt η die linke Seite des Dreiecks beschränkt, und für die Richtigkeit des Vorzeichens in der oberen Rechnung um $180°$ erhöht werden muss. Dann gilt: $\vartheta = \eta - \alpha$.

1.2 Ermittlung von ε

Für sämtliche Anordnungen eines Dreiecks wird der Winkel gesucht, der das vorherige Dreieck auf der rechten Seite begrenzt. Er wird in dieser Dokumentation ε genannt. Zur Bestimmung dieses Winkels muss derjenige Punkt ermittelt werden, von dem ausgehend die Strecke beginnt, die unter diesem Winkel steht. Das ist die nullte Ecke des vorherigen Dreiecks ($\vec{e_0}$), sofern dieses auf der Straße steht, und die zweite Ecke ($\vec{e_2}$) in jedem anderen Fall. Die auf diese Art bestimmte Ecke wird im Folgenden als »relevante Ecke« bezeichnet, und in Formeln als $\vec{P}$ dargestellt. Anschließend wird die Strecke berechnet, indem die relevante Ecke von der gegen den Uhrzeigersinn nächsten Ecke vektoriell subtrahiert wird, und über den Arkustangens des Verhältnisses von x- und y-Verschiebung erhält man den Winkel ε.

1.3 Anordnung eines Dreiecks mit der Spitze nach unten

Die Grundidee bei dieser Dreiecksanordnung besteht darin, dass das anzuordnende Dreieck mit seiner linken Seite an die Seite des vorherigen Dreiecks gelegt wird. Im Normalfall wird also über das oben erläuterte Verfahren der Winkel ε bestimmt und das Dreieck unter diesem Winkel nach dem in Abschnitt 1.2 erläuterten Verfahren angeordnet (in diesem Fall gilt: $\delta = \varepsilon$). Dabei wird als e_y der Wert null übergeben, und der Wert e_x

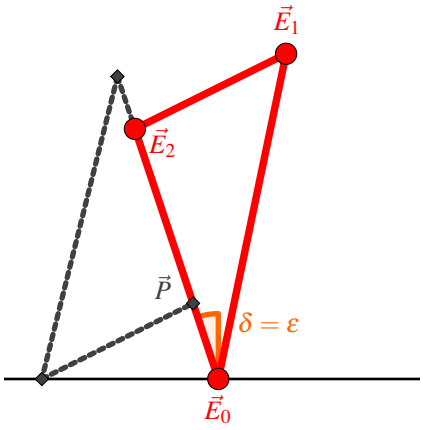

Abbildung 4: Grundidee der Anordnung mit der Spitze nach unten

folgendermaßen berechnet:

$$e_x = P_x - P_y \cdot \tan \delta \tag{5}$$

Diese Formel drückt den x-Wert, um den die Spitze verschoben wird, als Differenz der x-Position der relevanten Ecke (P_x) und dem Produkt aus y-Höhe dieser Ecke (P_y) und dem Tangens des Winkels, unter dem das Dreieck angeordnet werden soll, aus. In der Abbildung 4 wird ihre Bedeutung veranschaulicht: Wenn der Winkel δ einen negativen Wert zwischen $0°$ und $-90°$ aufweist, ist der Tangens negativ. Dadurch wird e_x so weit nach rechts verschoben, dass die Grundstücksgrenzen unter dem Winkel δ übereinander liegen.

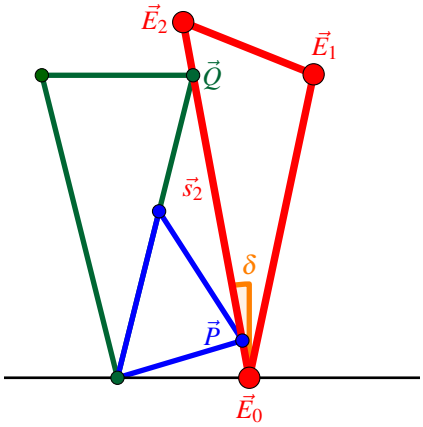

Abbildung 5: Grundidee der Anordnung mit der Spitze nach unten

Wie in der Abbildung 5 deutlich wird, kann es jedoch passieren, dass die Gleichung $\delta = \varepsilon$ für die Positionierung des Dreiecks nicht immer erfüllt sein darf, nämlich dann, wenn das anzuordnende Dreieck eines der vorherigen Dreiecke sonst schnitte. Stattdessen muss es so angeordnet werden, dass seine Grenze genau auf der Ecke $\vec{Q}$ liegt. Folgendes Vorgehen löst das Problem: Nach dem obigen Vorgehen sind bereits die Koordinaten für

$\vec{E}_0$ bekannt, nämlich $E_{0x} = e_x$ und $E_{0y} = 0$. Dann wird der Hilfsvektor $\vec{w}$ als Differenz aus $\vec{Q}$ und $\vec{E}_0$ berechnet. Anschließend wird nach folgender Frage entschieden, ob die Ecke $\vec{Q}$ ein Problem darstellt: Es muss genau dann eine neue Anordnung berechnet werden, wenn der Winkel, unter dem $\vec{w}$ steht, größer ist als der bisherige Winkel δ, und gleichzeitig der Betrag des Vektors $\vec{w}$ kleiner ist als die Länge der betroffenen Seitenkante $\vec{s}_2$. Sonst läge die Kante nicht auf der Ecke. Änderungsbedarf besteht ebenfalls ausschließlich dann, wenn $Q_y > 0$ gilt. Andernfalls beeinflussen Störeffekte aufgrund von Rechenungenauigkeiten das Ergebnis. Wenn hier aufgrund einer Ecke $\vec{Q}$ Änderungsbedarf festgestellt wird, muss ein neuer Winkel δ berechnet werden, der sich nicht mit dem Winkel decken muss, unter dem der Hilfsvektor $\vec{w}$ stand. Der neue Winkel δ wird als Winkel eines Vektors berechnet, der zwischen der »relevanten Ecke« des vorherigen Dreiecks $\vec{P}$ und $\vec{Q}$ bestimmt wird. Im allgemeinen Fall zieht das auch eine Anpassung des Wertes e_x nach sich, dessen neuer Wert wieder über Formel 5 berechnet wird. Um sicherzustellen, dass durch diese Regeln der richtige Winkel δ ermittelt wird, muss dieses Verfahren für alle Punkte derjenigen Dreiecke durchgeführt werden, die vor dem Dreieck liegen, das das Vorgängerdreieck des anzuordnenden Dreiecks ist.

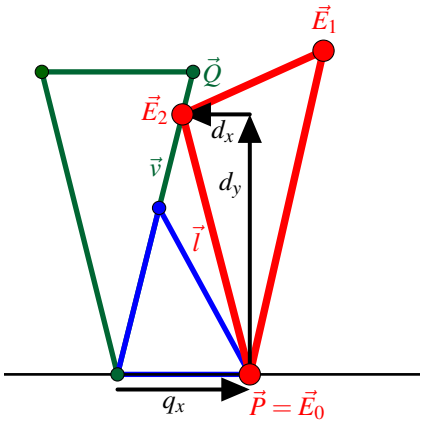

Abbildung 6: Grundidee der Anordnung mit der Spitze nach unten

Wie in der Abbildung 6 zu sehen ist, kann allerdings auch eine andere Situation eine Änderung der Position notwendig machen. Hier reicht die der Betrag des Vektors $\vec{s}_2$, der abgekürzt als l bezeichnet wird, nicht bis an die Ecke des davorliegenden Dreiecks heran, jedoch muss, um ein Überschneiden der Dreiecke zu verhindern, die Ecke $\vec{E}_2$ auf der Dreieckkante liegen, dessen Vektor als $\vec{v}$ bezeichnet wird. Die Verschiebung des Punktes $\vec{E}_2$ bezüglich $\vec{E}_0$ wird mit d_x und d_y bezeichnet. Zusätzlich wird die Variable q_x eingeführt, die sich als Differenz aus e_x und der x-Koordinate der Ecke, von der $\vec{v}$ ausgeht, ergibt.

Dadurch ergibt sich folgendes Gleichungssystem:

$$d_x^2 + d_y^2 = l^2 \tag{6}$$

$$v_x \cdot k = d_x + q_x \tag{7}$$

$$v_y \cdot k = d_y \tag{8}$$

Die erste dieser drei Bedingungen legt fest, dass d_x und d_y Katheten eines rechtwinkligen Dreiecks mit der Hypothenusenlänge l sind, während die zweite und dritte Bedingung den Zusammenhang zwischen d_x und d_y und dem Vektor $\vec{v}$ festlegen. Die eingeführte Konstante k dient dazu, einen Anteil des Vektors $\vec{v}$ festzulegen, nach dem der gemeinsame Punkt mit $\vec{E_2}$ liegt. Nur wenn sie einen Wert zwischen null und eins aufweist, besteht Handlungsbedarf. Neben der Konstante k sind d_x und d_y Unbekannte dieses Gleichungssystems. Die Lösung des Gleichungssystems ergibt folgende Lösungsmenge:

$$k_1 = \frac{q_x \cdot {v_x}^2 - \sqrt{l^2 \cdot {v_x}^4 + {v_x}^2 \cdot {v_y}^2 \cdot (l - q_x) \cdot (l + q_x)}}{v_x \cdot ({v_x}^2 + {v_y}^2)} \tag{9}$$

$$d_{x1} = -\frac{\sqrt{l^2 \cdot {v_x}^4 + {v_x}^2 \cdot {v_y}^2 \cdot (l - q_x) \cdot (l + q_x)} + q_x \cdot {v_y}^2}{{v_x}^2 + {v_y}^2} \tag{10}$$

$$d_{y1} = \frac{q_x \cdot {v_x}^2 \cdot v_y - v_y \cdot \sqrt{l^2 \cdot {v_x}^4 + {v_x}^2 \cdot {v_y}^2 \cdot (l - q_x) \cdot (l + q_x)}}{v_x \cdot ({v_x}^2 + {v_y}^2)} \tag{11}$$

$$k_2 = \frac{\sqrt{l^2 \cdot {v_x}^4 + {v_x}^2 \cdot {v_y}^2 \cdot (l - q_x) \cdot (l + q_x)} + q_x \cdot {v_x}^2}{v_x \cdot ({v_x}^2 + {v_y}^2)} \tag{12}$$

$$d_{x2} = \frac{\sqrt{l^2 \cdot {v_x}^4 + {v_x}^2 \cdot {v_y}^2 \cdot (l - q_x) \cdot (l + q_x)} - q_x \cdot {v_y}^2}{{v_x}^2 + {v_y}^2} \tag{13}$$

$$d_{y2} = \frac{v_y \cdot \sqrt{l^2 \cdot {v_x}^4 + {v_x}^2 \cdot {v_y}^2 \cdot (l - q_x) \cdot (l + q_x)} + q_x \cdot {v_x}^2 \cdot v_y}{v_x \cdot ({v_x}^2 + {v_y}^2)} \tag{14}$$

Wenn man Beispielwerte einsetzt, erkennt man, dass nur der zweite Teil dieser Lösungsmenge die gewünschten Werte liefert, während der erste Teil nach unten geklappte Dreiecke verursachen würde. Darüber hinaus erkennt man, dass jede dieser Lösungsformeln den Faktor $\sqrt{l^2 \cdot {v_x}^4 + {v_x}^2 \cdot {v_y}^2 \cdot (l - q_x) \cdot (l + q_x)}$ enthält, aus dessen Zusammensetzung bereits wichtige Schlüsse gezogen werden können. Ist nämlich der Teil unter der Wurzel negativ, so ergäbe das Berechnen der Wurzel eine irrationale Zahl. Es gibt also kein Ergebnis im reellen Zahlenraum. Im Sachzusammenhang bedeutet das, dass die Länge des anzuordnenden Dreiecks nicht an das abgeprüfte vorangegangene Dreieck heranreicht, also kein Handlungsbedarf besteht, da die betrachteten Dreiecke nicht kollisionsgefährdet sind. Wenn jedoch unter der Wurzel eine positive Zahl steht, ist ein Überschneiden denkbar und ein Winkel wird aus dem Arkustangens des Verhältnisses von d_x und d_y ermittelt.

Wenn dieser Winkel größer ist als der vorher gespeicherte linksbegrenzende Winkel δ, wird dieser ersetzt. Wie in der vorangegangenen Korrektur der Dreiecksposition müsste

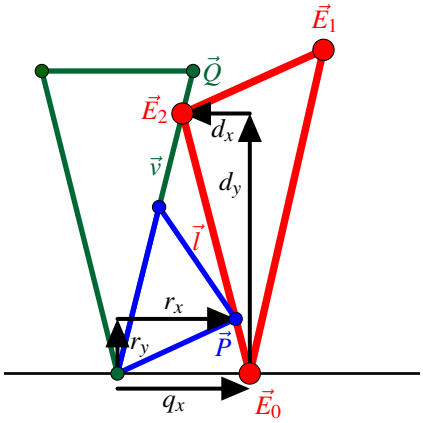

Abbildung 7: Grundidee der Anordnung mit der Spitze nach unten

jetzt noch eine Korrektur der x-Koordinate der Straßenecke unternommen werden, die sich jedoch recht schwierig gestaltet, da jede Änderung der x-Koordinate der Straßenecke den Wert q_x direkt ändert. Weil q_x im obigen Gleichungssystem jedoch als gegeben angesehen wird, wäre zur vollständigen Anpassung ein neues Gleichungssystem erforderlich, das zusätzlich q_x als Unbekannte enthält. Die dafür notwendigen Bedingungen ergeben sich, wenn man einsieht, dass nicht nur die dritte Ecke des Dreiecks auf der Seite eines der Vorgängerdreiecke liegen muss, die mit $\vec{v}$ gekennzeichnet wurde, sondern auch, dass die relevante Ecke des vorherigen Dreiecks auf der Seitenkante des anzuordnenden Dreiecks liegen muss (siehe Abbildung 7). Das obige Gleichungssystem mit drei Unbekannten erweitert sich dadurch zu einem Gleichungssystem mit fünf Unbekannten, die beschreiben, dass der Punkt nach einem bestimmten Anteil des Seitenvektors $\vec{d}$, der mit m bezeichnet wird, gleich der Verschiebung der relevanten Ecke bezüglich der Straßenecke des betrachteten, davorliegenden Dreiecks (r_x, r_y) ist:

$$d_x^2 + d_y^2 = l^2 \tag{15}$$

$$v_x \cdot k = d_x + q_x \tag{16}$$

$$v_y \cdot k = d_y \tag{17}$$

$$d_x \cdot m + q_x = rx \tag{18}$$

$$d_y \cdot m = r_y \tag{19}$$

Die analytische Lösung dieses nichtlinearen Gleichungssystems mit den fünf Unbekannten k, d_x, d_y, m und q_x gelingt nur noch äußerst schwierig, sodass ein numerisches Lösungsverfahren in der Implementierung notwendig wäre, was unverhältnismäßig aufwändig wäre, weswegen die kleine Ungenauigkeit der x-Koordinate bei der obigen Lösung

hingenommen wird, da sie kaum zu schlechteren Lösungen führt, und darüber hinaus die Gültigkeit der Lösung nicht beeinträchtigt. In einer Erweiterung könnte man diesen Punkt jedoch aufgreifen.

Diese Ermittlung eines neuen Winkels für diese Dreiecksanordnung muss wie die oben beschriebene, erste Prüfung für jedes vorangegangene Dreieck außer des direkten Vorgängers geschehen. Darüber hinaus darf sie sich nicht nur auf die linke Seite des anzuordnenden Dreiecks begrenzen, sondern muss auch die rechte prüfen, wobei von dem ermittelten Winkel der Innenwinkel der Straßenecke subtrahiert wird, um dem Fall vorzubeugen, dass die dritte Ecke des anzuordnenden Dreiecks zwar auf der betrachteten Seite liegt, der Winkel, unter dem der Vektor liegt, der von der ersten zur zweiten Ecke führt, jedoch geringer ist als der Winkel, unter dem der Vektor $\vec{v}$ steht, und dadurch die zweite Ecke des anzuordnenden Dreiecks im Inneren des betrachteten Vorgängerdreiecks läge.

Eine Erweiterung dieser Anordnung, um in einem Spezialfall noch einen gewissen Vorteil herauszuarbeiten, besteht darin, Dreiecke, die zum rechten Viertelkreis gehören, also die Anordnung der Dreiecke auf der rechten Seite begrenzen, nicht unbedingt mit dem Winkel ε an das Vorgängerdreieck zu legen, sondern so, dass das letzte zum rechten Rand gehörige Dreieck auf der Straße liegt. Der Vorteil wird besonders bei dem dritten Beispiel deutlich, da die Dreiecke nun tiefer unter das letzte Dreieck mit der Spitze nach oben geschoben werden können, wenn seine erste Ecke als Straßenecke dient. Wenn also die Summe der Innenwinkel aller verbleibenden Dreiecke und dem ermittelten Winkel ε kleiner ist als 180°, wird als Winkel δ die Differenz aus 180° und der Summe aller minimalen Innenwinkel der verbleibenden Dreiecke verwendet, nachdem natürlich geprüft wurde, dass keine anderen Dreiecke geschnitten werden.

1.4 Anordnung eines Dreiecks mit der Spitze nach oben

Bei der Anordnung eines Dreiecks mit der Spitze nach oben muss – wie im obigen Fall – zuerst der Winkel ε ermittelt werden, der das Vorgängerdreieck an der rechten Seite begrenzt. Dann wird mithilfe eines Verfahrens, das dem ersten Konfliktvermeidungsverfahren der Anordnung mit der Spitze nach unten ähnelt, untersucht, ob der Winkel δ angepasst werden muss. Dieses Verfahren ist weniger tief ausgearbeitet als das bei der Anordnung mit der Spitze nach unten geschieht, weil solche Überschneidungsprobleme bei den Dreiecken mit der Spitze nach unten selten sind, und die nachfolgende Regel immer Überschneidung verhindert, und dabei nur sehr geringe Verluste in der Lösung verursacht. Das Verfahren läuft so ab, dass von der relevanten Ecke des vorherigen Dreiecks ein Vektor zu jeder anderen Ecke gebildet wird. Über den Arkustangens des Verhältnisses von x- und y-Verschiebung dieses Vektors wird dann ein Winkel berechnet. Wenn dieser größer ist als derjenige, der bisher als δ gespeichert ist, wird er durch jenen ersetzt.

Wurde diese Prüfung für jede Ecke ausgeführt, ist jede Art der Überschneidung zwischen

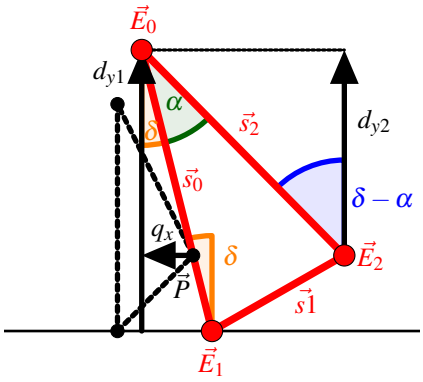

Abbildung 8: Grundidee der Anordnung mit der Spitze nach unten

dem anzuordnenden Dreieck und den vorherigen ausgeschlossen, weil es auf jeden Fall rechts der anderen angeordnet wird. Durch wenige Rechnungen müssen dann noch die Parameter ermittelt werden, die das in Abschnitt 1.1 beschriebene Vorgehen zur Positionierung eines Dreiecks benötigt. Dazu wird zuerst die Verschiebung der nullten Ecke in y-Richtung (d_y) ermittelt. In Frage kommen dabei zwei Entfernungen, die in der Abbildung 8 mit d_{y1} und d_{y2} beschriftet wurden. Wie man der Abbildung entnehmen kann, gilt für sie:

$$d_{y1} = |\vec{s_0}| \cdot \cos \alpha \tag{20}$$

$$d_{y1} = |\vec{s_2}| \cdot \cos (\delta - \alpha) \tag{21}$$

In der Rechnung und der Abbildung kann überraschen, dass α von δ subtrahiert wird, was jedoch der Konvention geschuldet ist, dass die nach links abspreizenden Winkel negativ sind. Von diesen wird als tatsächliche y-Verschiebung die längere von beiden genommen, woraus sich dann auch ergibt, ob $\vec{E}_1$ oder $\vec{E}_2$ die Straßenecke ist. Gemäß der Konvention beschreibt e_x die Verschiebung der nullten Ecke bezüglich der definierten Null des Koordinatensystems, q_y beschreibt in der Abbildung die Verschiebung der nullten Ecke bezüglich der relevanten Ecke des vorherigen Dreiecks $\vec{P}$, sodass e_x die Summe aus P_x und q_x ist. Für e_x gilt dann:

$$e_x = P_x + (d_y - P_y) \cdot \tan \delta \tag{22}$$

1.5 Algorithmus zum Anordnen der Dreiecke

Auf Grundlage dieser zwei Vorgehensweisen für eine Anordnung eines Dreiecks mit der Spitze nach unten, bzw. mit der Spitze nach oben sollen die Dreiecke so angeordnet wer-

den, dass der Abstand der äußersten Straßenecken minimal ist. Dazu wird zuerst folgende Überlegung angestellt: Gering ist der Abstand dann, wenn ein Dreieck möglichst aufrecht steht, da dann die Fläche pro Dreieck möglichst weit in y-Richtung ausgebreitet wird, und am wenigsten in x-Richtung. Eine Ausbreitung der Fläche in y-Richtung verursacht nach den Regeln der Aufgabenstellung keine Kosten, weshalb die Dreiecke nach folgender Regel angeordnet werden: Bevor ein Dreieck angeordnet wird, wird zunächst der Winkel ε ermittelt, also der Winkel, unter dem das Vorgängerdreieck auf der rechten Seite begrenzt wird. Gemäß der Konvention weisen solche Winkel, die eine Abspreizung nach links (gegen den Uhrzeigersinn) beschreiben, einen negativen Wert auf und alle, die eine Abspreizung nach rechts (im Uhrzeigersinn) beschreiben, einen positiven Wert. Weil die Dreiecke möglichst aufrecht stehen sollen, wird (bis auf eine gleich erläuterte Ausnahme) ein Winkel ε angestrebt, der so wenig wie möglich von 0° abweicht, weshalb immer bei einem negativen ε-Wert die Anordnung mit der Spitze nach unten gewählt wird und bei einem positiven Winkel die Anordnung mit der Spitze nach oben. Die angekündigte Ausnahme bezieht sich auf das rechtsseitige Ende der Anordnung: Die Regel zur Wahl, ob die Spitze nach oben oder nach unten angeordnet wird, liefert auf der linken Seite richtige Ergebnisse, da die Dreiecke so lange mit der Spitze nach unten angeordnet werden, bis der Viertelkreis ausgefüllt wurde. Für die rechte Seite muss noch eine Regel hinzugefügt werden: Wenn die Summe aller Innenwinkel der nullten Ecken der verbleibenden Dreiecke kleiner ist als die Differenz von 180° und dem ermittelten Winkels ε, ist es wahrscheinlich, dass alle verbleibenden Dreiecke mit der Spitze nach unten angeordnet werden können, und so den Viertelkreis auf der rechten Seite bilden.[1] In diesem Fall wird das anzuordnende Dreieck folglich auch dann mit der Spitze nach unten angeordnet, wenn ε größer ist als 0°.

Diese bereits erläuterten Vorgehensweisen ermöglichen es, korrekte Dreiecke, die in fester Reihenfolge vorgegeben sind, so anzuordnen, dass der zu minimierende Abstand zwischen den beiden äußersten Straßenecken recht gering ist. Um jedoch die Aufgabe vollumfänglich zu erfüllen, muss auch die Reihenfolge der Dreiecke angepasst werden. Ein einfacher Algorithmus, bei dem man sich sicher sein kann, dass er im Rahmen der erläuterten, auf Annahmen beruhenden Regeln die beste Lösung findet, probiert alle Reihenfolgen der Dreiecke aus, und speichert die Reihenfolge ab, die die geringste Entfernung ermöglicht, ist also mit der Laufzeit $O(n!)$ zu klassifizieren. Um eine gewisse Optimierung zu erreichen, kann bereits nach einer Zahl von angeordneten Dreiecken kleiner der Gesamtanzahl abgebrochen werden, wenn die Entfernung von der Straßenecke des

[1] Es handelt sich nicht um ein sicheres Ereignis, da der gemessene Winkel ε nicht in jedem Fall der gleiche ist wie δ, also dem Winkel, unter dem das Dreieck tatsächlich angeordnet wird, wie oben erläutert wurde. Das gefährdet jedoch nicht die Gültigkeit der Lösung, da das Verhindern von Überschneidungen aus Abschnitt 1.3 nicht außer Kraft gesetzt wird.

erstplatzierten Dreiecks zu derjenigen des zuletzt angeordneten die bisher als minimal gespeicherten Entfernung überschreitet. Dieses Backtrackingverfahren wird noch einmal im Folgenden beschrieben: Man befindet sich auf einer Tiefe d, auf der bereits d Dreiecke angeordnet wurden (die Zählung beginnt bei null). Nachdem der das vorher liegende Dreieck begrenzende Winkel ε ermittelt wurde, wird über die noch nicht angeordneten Dreiecke iteriert. Dieses in der Schleife ausgewählte Dreieck wird mit der Spitze nach unten angeordnet, wenn $\varepsilon \leq 0^\circ$ gilt, und mit der Spitze nach oben, wenn $\varepsilon > 0^\circ$ gilt. Nach der Anordnung dieses Dreiecks wird auf die Entfernung der Straßenecke dieses Dreiecks zur Straßenecke des zuerst angeordneten Dreiecks geachtet und in dem Fall, in dem diese Entfernung geringer ist als die bisher als minimal gespeicherte Gesamtentfernung, die nächste Tiefe aufgerufen, auf der die verbleibenden Dreiecke angeordnet werden. Sollte $d = n$ gelten, also alle Dreiecke angeordnet sein, liegt mit Sicherheit eine neue geringste Gesamtentfernung vor, weil die davorliegende Tiefe, die das letzte Dreieck angeordnet hat, ansonsten diese Tiefe nicht aufgerufen hätte. Dieses Verfahren bringt bereits signifikante Vorteile gegenüber dem reinen Bruteforce-Ansatz, ist allerdings weiterhin mit $O(n!)$ zu klassifizieren, weshalb ein anderer Ansatz gewählt werden muss, um auch bei größeren Datenmengen in vertretbarer Zeit Lösungen zu ermitteln.

1.6 Begründung eines schnelleren Algorithmus mit näherungsweise linearem Aufwand

Unter Aufgabe des Anspruches, innerhalb der Regeln über die Anordnung der Dreiecke die beste Lösung zu finden, wird folgender Ansatz gewählt: Die Menge an Dreiecken wird in Pakete mit vorgegebener Anzahl m sortiert, weshalb n/m Pakete existieren. Dabei muss darauf geachtet werden, dass bei der Division aufgerundet wird, sodass ein Paket am Ende nur teilweise gefüllt bleiben kann. Die Reihenfolge der Pakete bleibt zu jedem Zeitpunkt der Laufzeit unverändert, und nur innerhalb der Pakete findet Umverteilung statt. So wird für jedes Paket das Backtrackingverfahren angewandt, um innerhalb des Pakets die beste Reihenfolge zu finden, es müssen also im Worst-Case maximal $m!$-Kombinationen ausprobiert werden, in der Regel wegen des früheren Abbrechens signifikant weniger. Wurde ein Paket angeordnet, wird das nächste Paket mit konstanter Anordnung der Dreiecke der vorherigen Pakete angeordnet, wobei die Daten der vorherigen Dreiecke für die Anordnung der Dreiecke des neuen Pakets verwendet werden. Wenn alle Pakete angeordnet wurden, ist das Verfahren abgeschlossen und eine heuristische geringe Entfernung bekannt. Zur Rechenzeit lässt sich sagen, dass die Laufzeit in jedem Paket wegen der Konstanz der Paketgröße näherungsweise konstant ist, und die Zahl der Pakete proportional zur Zahl der Dreiecke steigt. Wenn man eine genaue Komplexitätsanalyse durchführt, wird man

jedoch feststellen, dass zwar die Zahl der auszuprobierenden Möglichkeiten pro Paket (abgesehen von dem letzten Paket, das weniger als m Pakete enthalten kann) unabhängig von der Position um einen Mittelwert pendelt, die Laufzeit für weiter hinten liegende Pakete jedoch zunimmt. Das liegt an den bei der Anordnung der Dreiecke gemäß der in den Abschnitten 1.3 und 1.4 erläuterten Regeln, die eine Überschneidung mit vorherliegenden Dreiecken verhindern. Demnach müssen mehr Vorgängerdreiecke abgeprüft werden müssen. Aus diesem Grund wächst die Laufzeit des Algorithmus etwas stärker als linear. Die genauen Verhältnisse lassen sich überblicksartig nicht abbilden und sind in hohem Maße implementierungsabhängig.

1.7 Vorsortierung der Daten

Um bessere Ergebnisse zu erzielen, können die Dreiecke in ihrer Anfangsreihenfolge noch vor Beginn dieses Algorithmus angepasst werden. Es hat sich als sinnvoll erwiesen, die Dreiecke nach dem Innenwinkel der nullten Ecke zu sortieren, und anschließend immer den kleinsten mit dem größten, den drittkleinsten mit dem drittgrößten, usw. zu tauschen, um sicherzustellen, dass besonders große Innenwinkel neben besonders kleinen liegen. Das lässt sich auch als sinnvoll begründen, da so einerseits die Dynamik erhöht wird, weil die Dreiecke mit geringem Innenwinkel der nullten Ecke leichter platzsparend angeordnet werden können. Damit werden die eher schwerfälligen Dreiecke mit größerem Innenwinkel der nullten Ecke ausgleichen. Andererseits werden Dreiecke mit geringem Innenwinkel an die Ränder sortiert, wo sie effizient zu den Viertelkreisen der beiden Begrenzungen gestapelt werden können.

1.8 Optimierung der Ränder

Als eine Optimierung dieses Algorithmus können noch Überlegungen zu den Rändern angestellt werden: An den Stellen, an denen die Dreiecke zu einem Viertelkreis zusammengefasst werden können, also am linken und rechten Rand der Anordnung, ist die Reihenfolge der Dreiecke nur bedingt entscheidend, sodass hier nicht alle Möglichkeiten ausprobiert werden müssen. Auf der rechten Seite wird das Problem ohne weiteres Zutun bereits durch den Algorithmus gelöst: Sobald die Dreiecke hier einmal vollständig angeordnet wurden, und dadurch eine neue minimale Entfernung bestimmt wurde, wird diese Entfernung als neuer Minimalwert gespeichert. Alle darunterliegenden Tiefen der Dreiecke dieses Endbereiches fallen so zurück, da ihre x-Position nicht mehr kleiner ist als die der minimalen Entfernung. Auf der linken Seite ist dieser Mechanismus jedoch nicht gegeben, da die Dreiecke von links nach rechts angeordnet werden, und dadurch die Anordnung am linken Rand am Anfang getätigt wird, nicht am Ende. Um hier die Anzahl der

Möglichkeiten zu reduzieren, wird folgende Überlegung angestellt: Entscheidend ist nur, welche der Dreiecke eines Pakets an den linken Rand sortiert werden, ihre Reihenfolge ist zunächst ohne Relevanz. Dadurch kann die Anzahl der Möglichkeiten für das linke Paket von $m!$ auf $m!/r!$ reduziert werden, wenn r die Zahl der Dreiecke ist, die zum linken Rand gehören. Das ist natürlich nur eine grobe Abschätzung, weil r nicht konstant ist, sondern stark davon abhängt, welche Innenwinkel die gewählten Dreiecke aufweisen. Tatsächlich werden so viele Dreiecke zum linken Rand geordnet, bis die Summe der Innenwinkel der nullten Ecken größer als 90° ist. Um die nachfolgenden Dreiecke möglichst wenig zu beeinträchtigen, wird die Menge der Dreiecke des linken Randes absteigend nach der Länge ihrer ersten Seite sortiert, weil die Dreiecke mit langer erster Seite mit höherer Wahrscheinlichkeit die nachfolgenden Dreiecke einschränken.

2 Umsetzung

Das Programm zur Lösung dieses Problems wurde in C implementiert, wobei zu Beginn triviale Datenstrukturen für geometrische Operationen wie `Punkt`, `Vektor` und `Gerade`, sowie die dazugehörigen Funktionen implementiert wurden, die für das Rechnen in den anderen Bereichen des Programms notwendig sind. Sie müssen hier nicht weiter erläutert werden, da sie im Quellcode selbsterklärend sind. Die Datenstruktur `Dreieck` enthält einen Array, der drei Elemente der Struktur `Punkt` speichert, und einen, der drei Elemente der Struktur `Gerade` speichert, die die Strecken beschreiben. Wichtig ist hierbei vor allem, dass darüber auch die Beträge der Strecken gespeichert werden, die für die weiteren Rechnungen verwendet werden. Des Weiteren wird der Modus des Dreiecks gespeichert, der den Wert 3 enthält, wenn das Dreieck mit der Spitze nach unten angeordnet werden soll, und 4, wenn es mit der Spitze nach oben angeordnet wird. Dass genau diese Werte gewählt werden, erscheint willkürlich, ist aber damit zu erklären, dass frühere Versionen des Programms noch mehr Modi enthielten. Nachdem aus der `main()`-Methode die Funktion `einlesen()` aufgerufen wurde, der der Dateiname der Datei, die die Dreieckspositionen beinhaltet, übergeben wurde, und dadurch die notwendigen Informationen eingelesen und in dem Array `dreiecke[]` der Datenstruktur `Dreieck` gespeichert wurden, wird die Funktion `dreiecksstrecken()` aufgerufen. Sie ist dafür verantwortlich, dass alle Dreiecke so in den jeweiligen Datenstrukturen vorliegen, wie es die Konventionen aus Abschnitt 1.1 fordern. Dazu wird zunächst die Funktion `einzel_dreieck_sw()` aufgerufen, die die Strecken der Dreiecke bestimmt, und darüber hinaus die jeweiligen Innenwinkel deklariert. Die Funktion `gegen_uhrzeiger()` richtet die Dreiecke dann gegen den Uhrzeigersinn aus. Diese Funktionen gleichen sich zu großen Teilen mit denen aus der Implementierung der Aufgabe 1 (»Lisa rennt«), sodass bei Unklarheiten in dieser Dokumenta-

tion nachgeschaut werden kann. Dabei ist ausschließlich zu beachten, dass die Teile, die hier mit »Dreieck« bezeichnet werden, dort »Hindernis« genannt werden. Die anschließend für jedes Dreieck aufgerufene Funktion `kleinsten_winkel_deklarieren()` iteriert durch die Ecken des angewählten Dreiecks und speichert die Position des kleinsten Innenwinkels, um anschließend die Positionen der Ecken so zu verändern, dass die Ecke mit dem kleinsten Innenwinkel auf die Position mit dem Index null sortiert wird. Danach liegen alle Dreiecke so in den Datenstrukturen vor, wie es die Konvention in Abschnitt 1.1 fordert. Im Anschluss wird die Funktion `dreieckesortieren()` aus der `main()`-Methode aufgerufen, die die Dreiecke zunächst absteigend nach ihrem kleinsten Innenwinkel sortiert, und danach – entsprechend des in Abschnitt 1.5 erklärten Verfahrens – umsortiert.

Im Anschluss beginnt der eigentliche Algorithmus, für den noch drei Arrays implementiert werden, die jeweils Ganzzahlen speichern: Die Arrays `besteanordnung[]` und `laufanordnung[]` speichern jeweils die Reihenfolge der Anordnungen der Dreiecke. Sie speichern jedoch nicht die Dreiecke direkt, sondern ihre Indizes, weil die eigentlichen Dreiecke in dem Array `dreiecke[]` gespeichert wurden, und hier in ihrer Reihenfolge nach dem Aufruf der Funktion `dreieckesortieren()` nicht mehr verändert werden. Der Array `besteanordnung[]` speichert dabei immer die beste momentan bekannte Anordnung und `laufanordnung[]` die Anordnung der Dreiecke, während verschiedene Möglichkeiten ausprobiert werden. Auf letzteren Array erfolgen also deutlich mehr Speicherzugriffe. Der dritte Ganzzahl-Array, `besteanordnung_modi[]`, speichert die Modi der Dreiecke bei ihrer besten bisher bekannten Anordnung, während die Modi zur Zeit des Ausprobierens in der Datenstruktur `Dreieck` gespeichert sind. Die Variable `geringster_abstand` speichert den Wert des bisher minimalen Abstandes als Gleitkommazahl, und der Boolean-Array `used[]` speichert, welche Dreiecke bereits verwendet wurden, und markiert die bereits verwendeten mit `TRUE`. In der `main()`-Methode wird zunächst die Anzahl der Pakete berechnet, und anschließend in einer Zählschleife über sie iteriert. Weil die Funktion `durchrechnen()`, die die Reihenfolge der Dreiecke innerhalb eines Pakets festlegt, rekursiv arbeitet, werden die einzelnen Ebenen als Tiefen bezeichnet. In der Variable `ausgangsdepth` wird diejenige Tiefe gespeichert, von der ausgehend das Paket der umzusortierenden Dreiecke angeordnet wird, und in der Variable `zieldepth` die Tiefe, bis zu der das Paket reicht. Mit diesen Parametern wird dann die Funktion `durchrechnen()` aufgerufen – die Erweiterung wird später erläutert –, um die Dreiecke mit den Indizes `ausgangsdepth` bis `zieldepth - 1` mit möglichst geringem Abstand der äußersten Straßenecken anzuordnen. Diese Funktion prüft zuerst, ob die Tiefe ihrer Rekursion der Tiefe `zieldepth` entspricht. In diesem Fall liegt auf jeden Fall eine neue minimale Entfernung vor, sodass der Abstand der Straßenecke des

ersten Dreiecks und der des Dreiecks mit dem Index `zieldepth - 1` in die Variable `geringster_abstand` geschrieben wird. Anschließend werden die Daten aus dem Arrays `laufanordnung[]` in `besteanordnung[]` übertragen und die Modi der Anordnungen in den Array `besteanordnung_modi[]`. Die Funktion `standardanordnung()`, die den Abstand der Straßenecken bestimmt, wird später noch erläutert. In dem Fall, dass die Zieltiefe noch nicht erreicht wurde, wird zunächst über die Funktion `restwinkelberechnen()` die Summe der minimalen Innenwinkel all derjenigen Dreiecke ermittelt, die noch nicht angeordnet wurden. Das sind die Dreiecke, die in dem Array `used[]` mit `FALSE` markiert sind. Die Funktion `vorherrechtswinkel()` bestimmt den Winkel, der im Abschnitt »Lösungsidee« mit ε bezeichnet wurde. Dann zählt eine Zählschleife durch die Menge der Dreiecke, die zu dem aktuell anzuordnenden Paket zählen, also alle Dreiecke mit den Indizes `ausgangsdepth` bis `zieldepth - 1`. Wenn das Dreieck, auf dessen Index die Zählschleife gerade steht, im Array `used[]` noch nicht als verwendet markiert wurde, wird es folgendermaßen ausprobiert: Zuerst wird der Index des Dreiecks in das Element des Arrays `laufanordnung[]`, das der aktuellen Rekursionstiefe entspricht, geschrieben und anschließend wird das Dreieck im Array `used[]` als bereits verwendet markiert. Dann steht die Entscheidung an, ob das Dreieck mit der Spitze nach oben oder nach unten angeordnet werden soll. Mit der Spitze nach unten wird das Dreieck, wie im Abschnitt »Lösungsidee« erläutert, genau dann angeordnet, wenn entweder der Winkel ε kleiner oder gleich null ist, oder der Restwinkel kleiner gleich $2\pi - \varepsilon$.[2] In diesem Fall wird die Funktion `spitze_unten()` aufgerufen, ansonsten die Funktion `spitze_oben()`. Diese Funktionen ordnen entsprechend der in den Abschnitten 1.4 und 1.3 ausführlich erläuterten Vorgehensweisen die Dreiecke an, wobei ein Teil der Arbeit in die Funktionen `winkel_gegen_ueberschneidung()` und `konfliktwinkel()` ausgelagert ist. Diese Funktionen bedürfen trotz ihrer relativen Länge keiner ausführlichen Erklärung in diesem Teil, weil sie ausschließlich die oben erläuterten Funktionalitäten in Anweisungsfolgen enthalten. Die Funktionen `spitze_unten()` und `spitze_oben()` geben am Schluss die x-Koordinate der Straßenecke des angeordneten Dreiecks zurück, das dem Abstand zur Straßenecke des ersten Dreiecks entspricht. Wenn dieser Abstand geringer ist als der, der in der Variable `geringster_abstand` gespeichert wurde, wird die nächste Rekursionstiefe der Funktion `durchrechnen()` aufgerufen. Anschließend wird das Dreieck wieder als nicht verwendet markiert und an die Position im Array `laufanordnung[]` der Platzhalter -1 geschrieben. Wurde ein Paket gerechnet, werden in der `main()`-Methode die Daten aus den Arrays `besteanordnung[]` und `besteanordnung_modi[]` wieder in `laufanordnung[]` und die Dreiecksstrukturen

[2] Während im Abschnitt »Lösungsidee« zur besseren Anschaulichkeit im Gradmaß gearbeitet wurde, finden sämtliche Berechnungen des Programms im Bogenmaß statt.

übertragen, um dann die Funktion `standardanordnung()` aufzurufen. Diese wurde bereits weiter oben angekündigt und ordnet die Dreiecke von links nach rechts entsprechend des gespeicherten Anordnungsmodus an, um sicherzustellen, dass für das weitere Vorgehen die richtigen Koordinaten bei den Dreiecken eingetragen wurden. Am Ende des Programms wird die Funktion `dateischreiben()` aufgerufen, die alle Koordinaten auf die Kommandozeile ausdruckt und die graphische Darstellung in eine SVG-Datei schreibt.

Zum Schluss wird die Funktion `anfangsstapeln()` erläutert, die die Erweiterung zur Geschwindigkeitsoptimierung des ersten Pakets realisiert. Sie wird aus der `main()`-Methode genau dann aufgerufen, wenn das erste Paket angeordnet werden soll. Sie benötigt den Array `anfangsstapel[]`, in dem die Indizes derjenigen Dreiecke gespeichert werden, die zum Anfangsstapel gehören. Um zu ermöglichen, dass diese Funktion nur auswählt, welche Dreiecke zum Anfangsstapel gehören, und keine Reihenfolge festlegt, wird ihr neben den Werten der aktuellen Tiefe, Ausgangstiefe und Zieltiefe auch die Variable `cnum` übergeben. Sie legt den Anfangswert fest, von dem ausgehend die Zählschleife durch die anzuordnenden Dreiecke zählt, wenn weitere Dreiecke dem Anfangsstapel zugefügt werden sollen. Das ist dann der Fall, wenn der Gesamtwinkel des Anfangsstapels kleiner ist als $\pi/2$ und die Differenz aus Zieltiefe und momentaner Tiefe größer ist als eins. Innerhalb des Zählbereiches dieser Zählschleife werden nur noch die Dreiecke angesprochen, die noch nicht verwendet werden, da bei der Wahl eines Dreiecks, nachdem dieses als verwendet markiert wurde und sein Index in den Array `anfangsstapel[]` eingetragen wurde, die Funktion `anfangsstapeln()` rekursiv aufgerufen wird, wobei als `cnum` der Wert der aktuellen Tiefe plus eins übergeben wird, sodass in einer späteren Tiefe nie solche Dreiecke gewählt werden, die schon auf einer früheren Tiefe hätten gewählt werden können. Ist die Bedingung, neue Dreiecke in den Anfangsstapel aufzunehmen, nicht erfüllt, werden die Dreiecke zunächst in einen Zwischenspeicher, der `sortierstapel[]` genannt wird, übertragen. Hier müssen sie nach der Länge der nullten Seite sortiert werden, sind jedoch vor Beginn noch nach ihrem Index sortiert. Zur Sortierung genügt ein einfaches Sortierverfahren, da die Datenmenge klein ist, und anschließend wird die sortierte Menge in den Array `laufanordnung[]` übertragen und ihre Modi auf 3 gesetzt, was eine Anordnung mit der Spitze nach unten bedeutet. Die Funktion `standardanordnung()` lässt diese Dreiecke dann anordnen, und anschließend wird die Funktion `durchrechnen()` aufgerufen, die die verbleibenden Dreiecke anordnet und kürzeste Entfernungen bestimmt.

3 Erweiterungsausblick: Trianguläre auf unwegsamem Gelände

Eine spannende Erweiterung besteht darin, die Trianguläre auf einem Gebiet anzusiedeln, dessen Oberfläche nicht flach ist, sondern hügelig. Dann gibt jedes Clubmitglied ein Wunschgrundstück mit drei Winkeln und einem Flächeninhalt ab, das dann auf den Hügeln angeordnet werden muss. Weil nur bei bestimmten Oberflächenkrümmungen der gewünschte Flächeninhalt realisiert werden kann, ergeben sich interessante Bedingungen für die Anordnung der Dreiecke, die nun noch sehr viel rechenaufwändiger wird. Dabei könnte passieren, dass unter bestimmten Bedingungen der Landschaft ein Mitglied seine Wünsche überhaupt nicht erfüllt bekommt, was natürlich durch das Programm erkannt werden muss und gegebenenfalls Änderungsvorschläge gibt.

Mathematisch und algorithmisch wäre das Programm jedoch sehr anspruchsvoll, da man zuerst die gekrümmte Oberfläche darstellen muss. Danach muss man all diese Orte erkennen, an denen das Dreieck gemäß der Wünsche überhaupt angeordnet werden kann, um danach eine Reihenfolge für geringen Abstand der Straßenecken zu generieren. Dafür hat man aber eine Aufgabe, die die Berechnungen nicht nur Geländedaten realistischer macht, sondern auch die in der Aufgabe begründete Seltsamkeit des Clubs durch noch weitere Ansprüche erweitert.

4 Beispiele

Das Programm wird über die Kommandozeile aufgerufen und der Name der Textdatei mit den Koordinaten der Dreiecke übergeben. Als weiterer Übergabeparameter kann die Größe der Pakete übergeben werden, die standardmäßig zehn beträgt. Wenn Änderungsbedarf bei der Größe des Ausschnittes besteht, kann der x-Offset, die Breite und die Höhe über die nächsten Parameter mitgegeben werden.

4.1 Erstes Beispiel

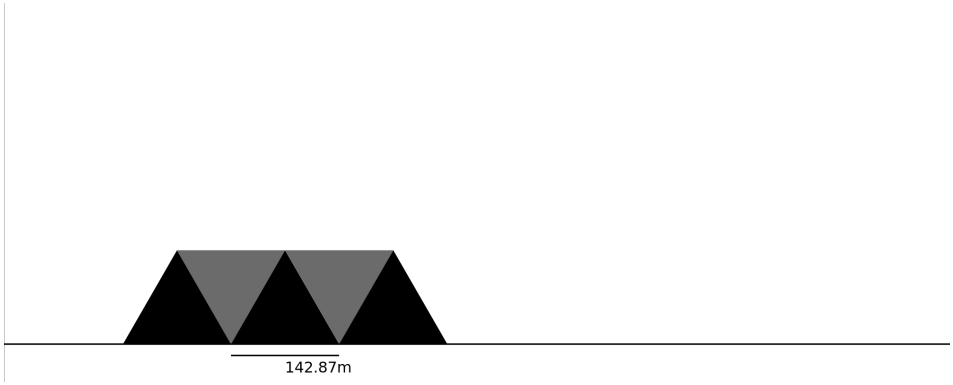

Abbildung 9: Das erste Beispiel

```
Standardmaessig wird mit 10 Dreiecken pro Paket gerechnet.
Jetzt wurden die Dreiecke von 0 bis 5 gerechnet und der bisherige
    Abstand ist 142.87m.
                x1|    y1;     x2|    y2;     x3|    y3;
Dreieck  0:    0.0|   0.0;  -71.5| 123.7; -142.9|   0.0;
Dreieck  1:    0.0|   0.0;   71.4| 123.8;  -71.5| 123.7;
Dreieck  2:   71.4| 123.8;    0.0|  -0.0;  142.8|   0.1;
Dreieck  3:  142.9|   0.0;  214.3| 123.7;   71.5| 123.8;
```

4.2 Zweites Beispiel

```
Jetzt wurden die Dreiecke von 0 bis 5 gerechnet und der bisherige
    Abstand ist 0.00m.
                x1|    y1;     x2|    y2;     x3|    y3;
Dreieck  0:    0.0|   0.0; -510.6| 149.9; -516.4|   0.0;
Dreieck  1:    0.0|   0.0;  229.2| 464.1;   82.4| 494.6;
Dreieck  2:    0.0|   0.0;  322.1| 326.2;  210.9| 426.8;
Dreieck  3:    0.0|   0.0;  277.9|  75.3;  202.3| 204.9;
```

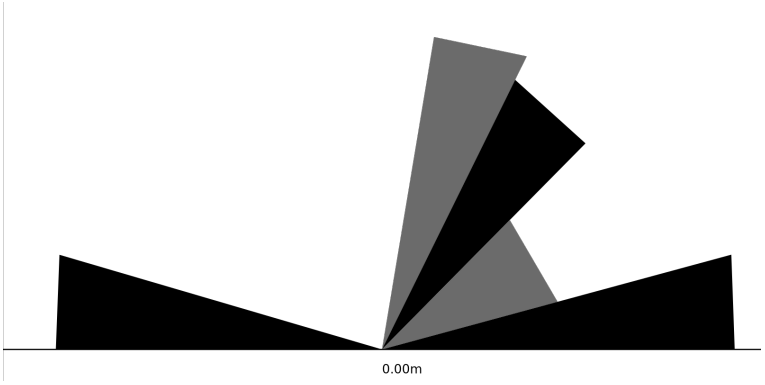

Abbildung 10: Beim zweiten Beispiel wurden anere Maße für das Bild übergeben, damit alle Dreiecke sichtbar sind

```
Dreieck 4:    0.0|   0.0;  558.4|   0.0;  553.0| 149.9;
Eine graphische Darstellung wurde in die Datei "dreieckergebnis.svg"
    geschrieben.
```

4.3 Drittes Beispiel

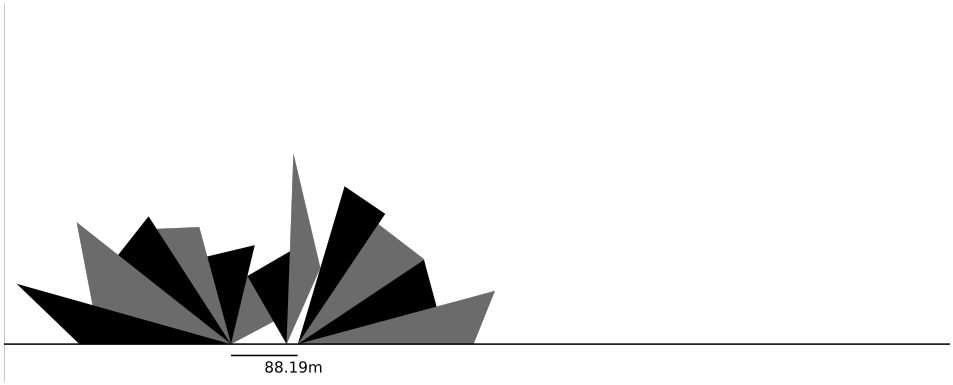

Abbildung 11: Beim dritten Beispiel wurden alle Dreiecke in einem Paket gerechnet

```
Standardmaessig hat das Bild beim Speichern in der Datei einen xoffset
    von 300 die Breite 1250 und die Hoehe 500
Jetzt wurden die Dreiecke von 0 bis 12 gerechnet und der bisherige
    Abstand ist 88.19m.
                x1|    y1;     x2|    y2;     x3|    y3;
Dreieck  0:    0.0|   0.0; -283.4|  79.6; -200.5|   0.0;
Dreieck  1:    0.0|   0.0; -204.0| 160.7; -183.0|  51.4;
Dreieck  2:    0.0|   0.0; -109.0| 168.6; -149.2| 117.6;
Dreieck  3:    0.0|   0.0;  -42.0| 154.4;  -98.2| 151.9;
Dreieck  4:    0.0|   0.0;   31.3| 130.5;  -31.4| 115.7;
Dreieck  5:   21.6|  89.9;    0.0|  -0.0;   56.1|  29.9;
```

```
Dreieck  6:    73.3|    0.0;    77.8|  122.4;    21.6|   89.8;
Dreieck  7:    82.5|  251.6;    73.3|    0.0;   118.1|  100.6;
Dreieck  8:    88.2|    0.0;   203.8|  171.7;   150.0|  208.3;
Dreieck  9:    88.2|    0.0;   255.6|  111.7;   194.7|  158.3;
Dreieck 10:    88.2|    0.0;   271.6|   49.5;   255.0|  111.2;
Dreieck 11:    88.2|    0.0;   320.4|   -0.0;   348.7|   70.2;
Eine graphische Darstellung wurde in die Datei "dreieckergebnis.svg"
    geschrieben.
```

4.4 Viertes Beispiel

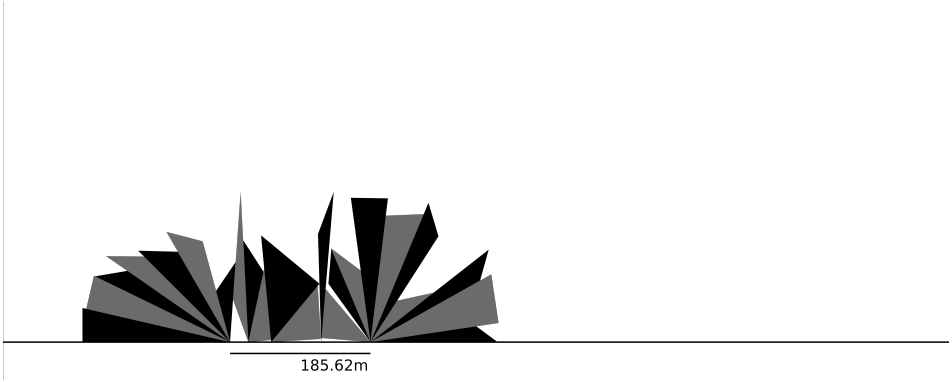

Abbildung 12: Beim vierten Beispiel wurde mit zwölf Dreiecke pro Paket gerechnet

```
Standardmaessig hat das Bild beim Speichern in der Datei einen xoffset
    von 300 die Breite 1250 und die Hoehe 500
Jetzt wurden die Dreiecke von 0 bis 12 gerechnet und der bisherige
    Abstand ist 54.41m.
Jetzt wurden die Dreiecke von 12 bis 23 gerechnet und der bisherige
    Abstand ist 185.62m.
                 x1|     y1;      x2|     y2;      x3|     y3;
Dreieck  0:     0.0|    0.0;  -195.0|   45.0;  -195.0|    0.0;
Dreieck  1:     0.0|    0.0;  -179.9|   87.7;  -190.0|   43.8;
Dreieck  2:     0.0|    0.0;  -135.7|   93.9;  -178.2|   86.9;
Dreieck  3:     0.0|    0.0;  -113.5|  112.8;  -163.7|  113.3;
Dreieck  4:     0.0|    0.0;   -69.0|  119.3;  -121.3|  120.5;
Dreieck  5:     0.0|    0.0;   -36.2|  132.9;   -84.1|  145.4;
Dreieck  6:     0.0|    0.0;     7.5|  105.7;   -18.4|   67.5;
Dreieck  7:    14.1|  198.3;     4.0|   56.0;    24.3|    0.0;
Dreieck  8:    24.3|    0.0;    44.2|   92.9;    17.4|  134.5;
Dreieck  9:    45.2|   97.4;    24.3|    0.0;    54.1|    2.8;
Dreieck 10:    54.4|    0.0;   118.1|   77.1;    41.1|  140.8;
Dreieck 11:   115.7|   74.1;    54.4|    0.0;   120.3|    4.3;
```

```
Dreieck 12:   120.4|   0.0;   136.9|  198.2;   116.2|  142.7;
Dreieck 13:   126.2|  70.2;   120.8|    5.4;   185.6|    0.0;
Dreieck 14:   185.6|   0.0;   133.8|  122.9;   130.0|   77.0;
Dreieck 15:   185.6|   0.0;   172.7|   94.1;   133.3|  124.0;
Dreieck 16:   185.6|   0.0;   208.5|  189.7;   159.5|  190.4;
Dreieck 17:   185.6|   0.0;   255.9|  168.9;   205.8|  166.8;
Dreieck 18:   185.6|   0.0;   275.4|  139.7;   262.1|  183.8;
Dreieck 19:   185.6|   0.0;   269.1|   65.2;   220.8|   54.7;
Dreieck 20:   185.6|   0.0;   330.4|   81.2;   341.9|  122.0;
Dreieck 21:   185.6|   0.0;   354.8|   25.2;   345.2|   89.5;
Dreieck 22:   185.6|   0.0;   353.6|    0.0;   325.5|   20.8;
Eine graphische Darstellung wurde in die Datei "dreieckergebnis.svg"
    geschrieben.
```

4.5 Fünftes Beispiel

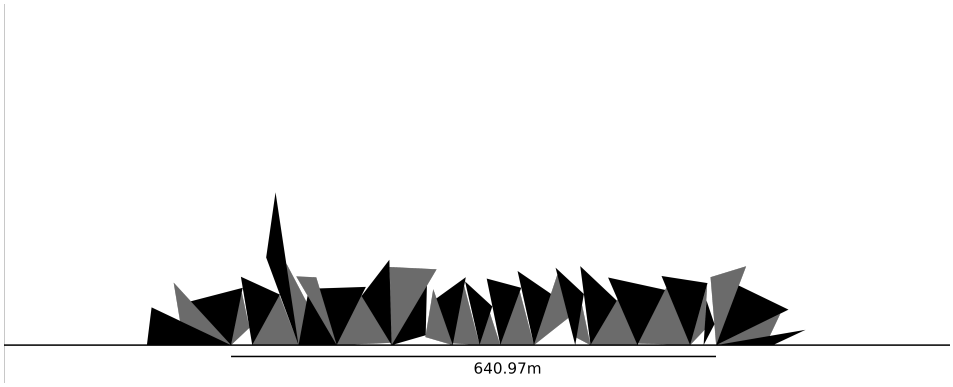

Abbildung 13: Beim fünften Beispiel wurde mit zwölf Dreiecke pro Paket gerechnet

```
Standardmaessig hat das Bild beim Speichern in der Datei einen xoffset
    von 300 die Breite 1250 und die Hoehe 500
Jetzt wurden die Dreiecke von 0 bis 12 gerechnet und der bisherige
    Abstand ist 139.68m.
Jetzt wurden die Dreiecke von 12 bis 24 gerechnet und der bisherige
    Abstand ist 400.19m.
Jetzt wurden die Dreiecke von 24 bis 36 gerechnet und der bisherige
    Abstand ist 640.97m.
Jetzt wurden die Dreiecke von 36 bis 37 gerechnet und der bisherige
    Abstand ist 640.97m.
                 x1|    y1;      x2|    y2;      x3|    y3;
Dreieck  0:    0.0|   0.0;  -105.1|  49.7;  -111.2|   0.0;
Dreieck  1:    0.0|   0.0;   -75.8|  82.5;   -67.3|  31.8;
Dreieck  2:    0.0|   0.0;    15.4|  75.6;   -53.2|  57.9;
```

```
Dreieck  3:   14.2|  69.6;    0.0|  -0.0;   24.1|  22.9;
Dreieck  4:   27.9|   0.0;   64.5|  66.7;   13.0|  90.2;
Dreieck  5:   64.0|  65.7;   27.9|   0.0;   89.2|   0.2;
Dreieck  6:   89.3|   0.0;   58.9| 201.5;   46.3| 115.6;
Dreieck  7:   89.3|   0.0;   99.6|  57.1;   73.0| 107.8;
Dreieck  8:  101.1|  65.4;   89.3|   0.0;  139.6|   0.2;
Dreieck  9:  139.7|   0.0;  112.6|  89.3;   86.2|  90.6;
Dreieck 10:  139.7|   0.0;  178.1|  76.8;  117.0|  74.8;
Dreieck 11:  172.3|  65.1;  139.7|   0.0;  210.3|   3.1;
Dreieck 12:  212.2|   0.0;  209.2| 112.6;  172.4|  64.8;
Dreieck 13:  212.2|   0.0;  271.6| 100.1;  209.5| 102.9;
Dreieck 14:  258.7|  78.5;  212.2|   0.0;  257.2|  13.1;
Dreieck 15:  267.2|  73.8;  256.8|  10.7;  292.9|   0.0;
Dreieck 16:  292.9|   0.0;  310.3|  89.6;  271.4|  61.7;
Dreieck 17:  308.0|  78.0;  293.4|   2.4;  328.4|   0.0;
Dreieck 18:  328.4|   0.0;  345.3|  51.0;  309.2|  83.9;
Dreieck 19:  342.2|  41.8;  328.6|   0.4;  355.8|   0.0;
Dreieck 20:  355.8|   0.0;  383.5|  75.7;  337.8|  87.7;
Dreieck 21:  381.5|  70.4;  355.8|   0.0;  400.2|   0.2;
Dreieck 22:  400.2|   0.0;  422.7|  67.7;  378.6|  97.8;
Dreieck 23:  431.2|  93.1;  400.2|   0.0;  445.7|  35.8;
Dreieck 24:  454.7|   0.0;  466.3|  67.5;  428.9| 102.2;
Dreieck 25:  463.7|  52.6;  456.4|   9.9;  475.0|   0.0;
Dreieck 26:  475.0|   0.0;  509.8|  58.6;  461.5| 104.2;
Dreieck 27:  510.8|  60.4;  475.0|   0.0;  537.0|   0.1;
Dreieck 28:  537.1|   0.0;  573.6|  72.7;  498.3|  89.2;
Dreieck 29:  575.1|  75.8;  538.2|   2.2;  606.6|   0.0;
Dreieck 30:  606.6|   0.0;  629.7|  81.8;  568.7|  91.2;
Dreieck 31:  628.9|  79.1;  606.6|   0.0;  625.8|  20.1;
Dreieck 32:  628.2|  59.0;  624.5|   0.0;  638.7|  27.6;
Dreieck 33:  641.0|   0.0;  680.5| 104.1;  633.6|  89.6;
Dreieck 34:  641.0|   0.0;  736.6|  46.7;  670.9|  78.6;
Dreieck 35:  641.0|   0.0;  711.8|  12.1;  725.7|  41.4;
Dreieck 36:  641.0|   0.0;  715.9|   0.0;  759.3|  20.2;
Eine graphische Darstellung wurde in die Datei "dreieckergebnis.svg"
    geschrieben.
```

4.6 Beispiel mit 40 Dreiecken

```
Standardmaessig wird mit 10 Dreiecken pro Paket gerechnet.
Jetzt wurden die Dreiecke von 0 bis 10 gerechnet und der bisherige
    Abstand ist 108.52m.
```

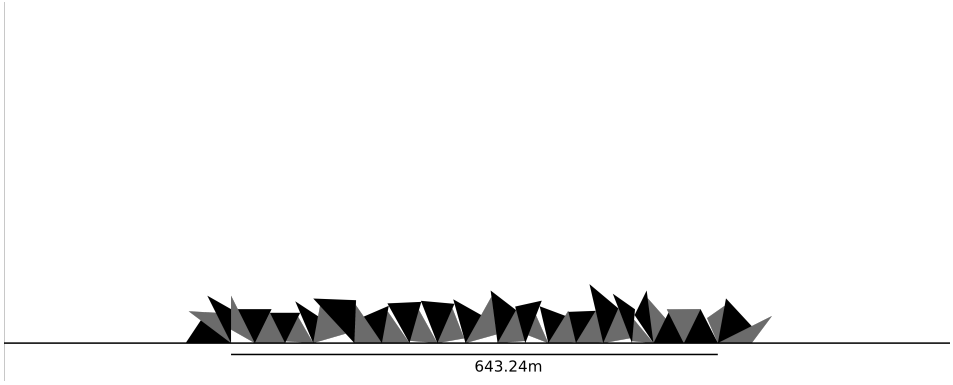

Abbildung 14: Ein eigenes Beispiel mit 40 Dreiecken

```
Jetzt wurden die Dreiecke von 10 bis 20 gerechnet und der bisherige
    Abstand ist 309.80m.
Jetzt wurden die Dreiecke von 20 bis 30 gerechnet und der bisherige
    Abstand ist 491.88m.
Jetzt wurden die Dreiecke von 30 bis 40 gerechnet und der bisherige
    Abstand ist 643.24m.
                x1|    y1;     x2|    y2;     x3|    y3;
Dreieck  0:    0.0|   0.0;  -40.0|  30.0;  -60.0|   0.0;
Dreieck  1:    0.0|   0.0;  -20.0|  40.0;  -56.0|  42.0;
Dreieck  2:    0.0|   0.0;    0.0|  44.7;  -31.3|  62.6;
Dreieck  3:    0.0|  62.6;    0.0|  17.9;   31.3|   0.0;
Dreieck  4:   31.3|   0.0;   53.7|  44.7;    8.9|  44.7;
Dreieck  5:   51.3|  40.0;   31.3|  -0.0;   71.3|   0.0;
Dreieck  6:   71.3|   0.0;   91.3|  40.0;   51.3|  40.0;
Dreieck  7:   88.7|  34.7;   72.5|   2.5;  108.5|   0.0;
Dreieck  8:  108.5|   0.0;  115.0|  35.5;   84.8|  55.1;
Dreieck  9:  117.5|  49.2;  108.5|   0.0;  151.5|  12.4;
Dreieck 10:  163.0|   0.0;  165.2|  56.5;  108.7|  58.8;
Dreieck 11:  165.0|  50.0;  163.0|   0.0;  199.0|   0.6;
Dreieck 12:  199.4|   0.0;  208.4|  49.2;  175.3|  34.9;
Dreieck 13:  207.1|  41.7;  199.4|   0.0;  235.5|   0.6;
Dreieck 14:  235.8|   0.0;  251.3|  54.4;  206.6|  52.4;
Dreieck 15:  248.2|  43.7;  236.6|   2.9;  272.5|   0.0;
Dreieck 16:  272.5|   0.0;  295.3|  51.8;  250.7|  55.9;
Dreieck 17:  292.6|  45.8;  272.5|   0.0;  308.0|   6.3;
Dreieck 18:  309.8|   0.0;  330.6|  37.0;  293.6|  57.8;
Dreieck 19:  344.1|  61.0;  309.8|   0.0;  350.7|  11.5;
Dreieck 20:  352.2|   0.0;  376.6|  43.6;  343.0|  69.4;
Dreieck 21:  376.6|  43.6;  352.2|  -0.0;  388.1|   2.8;
Dreieck 22:  388.9|   0.0;  410.6|  55.9;  375.4|  48.1;
Dreieck 23:  403.1|  36.6;  392.9|  10.2;  419.2|   0.0;
Dreieck 24:  419.2|   0.0;  441.5|  36.1;  407.8|  48.7;
```

```
Dreieck 25:  445.5| 42.5;  419.2|  0.0;  455.3|  1.2;
Dreieck 26:  455.6|  0.0;  481.9| 42.5;  445.8| 41.3;
Dreieck 27:  481.9| 42.5;  455.6|  0.0;  491.6|  1.2;
Dreieck 28:  491.9|  0.0;  511.9| 45.8;  473.5| 77.9;
Dreieck 29:  511.9| 45.8;  491.9|  0.0;  527.4|  6.3;
Dreieck 30:  529.9|  0.0;  533.9| 44.5;  504.3| 65.2;
Dreieck 31:  532.6| 30.7;  530.1|  2.5;  558.3| -0.0;
Dreieck 32:  558.3|  0.0;  549.1| 69.4;  533.2| 37.0;
Dreieck 33:  558.3|  0.0;  574.2| 32.4;  550.4| 59.5;
Dreieck 34:  578.0| 40.1;  558.3|  0.0;  598.3|  0.3;
Dreieck 35:  598.4|  0.0;  620.5| 44.9;  575.7| 44.6;
Dreieck 36:  620.5| 44.9;  598.4| -0.0;  643.1|  0.3;
Dreieck 37:  643.2|  0.0;  652.2| 49.2;  628.9| 33.1;
Dreieck 38:  643.2|  0.0;  688.0| 22.4;  654.0| 59.0;
Dreieck 39:  643.2|  0.0;  688.0| -0.0;  714.8| 35.8;
Eine graphische Darstellung wurde in die Datei "dreieckergebnis.svg"
    geschrieben.
```

4.7 Beispiel mit 50 Dreiecken

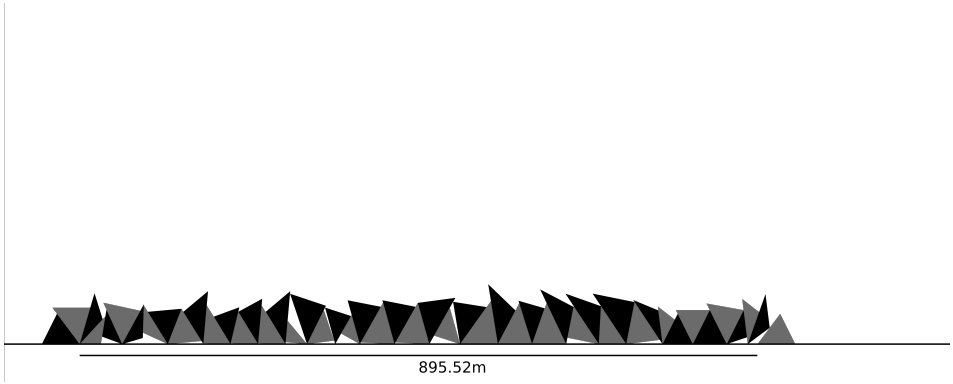

Abbildung 15: Ein eigenes Beispiel mit 50 Dreiecken

```
Standardmaessig wird mit 10 Dreiecken pro Paket gerechnet.
Jetzt wurden die Dreiecke von 0 bis 10 gerechnet und der bisherige
    Abstand ist 116.33m.
Jetzt wurden die Dreiecke von 10 bis 20 gerechnet und der bisherige
    Abstand ist 300.37m.
Jetzt wurden die Dreiecke von 20 bis 30 gerechnet und der bisherige
    Abstand ist 502.54m.
Jetzt wurden die Dreiecke von 30 bis 40 gerechnet und der bisherige
    Abstand ist 722.14m.
Jetzt wurden die Dreiecke von 40 bis 50 gerechnet und der bisherige
    Abstand ist 895.52m.
```

```
                x1|    y1;    x2|    y2;    x3|    y3;
Dreieck  0:   0.0|   0.0;  −30.0|  40.0;  −50.0|   0.0;
Dreieck  1:   0.0|   0.0;   14.0|  48.0;  −36.0|  48.0;
Dreieck  2:   0.0|   0.0;   30.4|  32.8;   19.6|  67.2;
Dreieck  3:  34.0|  36.7;    0.0|   0.0;   28.3|   1.1;
Dreieck  4:  35.4|  45.5;   29.7|   9.9;   56.2|   0.0;
Dreieck  5:  56.2|   0.0;   80.1|  43.9;   31.2|  54.6;
Dreieck  6:  84.9|  52.7;   56.2|   0.0;   83.3|   8.0;
Dreieck  7:  84.8|  51.1;   83.6|  15.0;  116.3|   0.0;
Dreieck  8: 116.3|   0.0;  134.6|  46.5;   90.1|  42.5;
Dreieck  9: 134.6|  46.5;  116.3|   0.0;  160.9|   4.0;
Dreieck 10: 163.3|   0.0;  169.7|  69.7;  137.1|  42.5;
Dreieck 11: 168.0|  51.1;  163.5|   1.3;  199.5|   0.0;
Dreieck 12: 199.5|   0.0;  211.0|  48.7;  177.2|  36.1;
Dreieck 13: 209.6|  42.5;  199.8|   1.3;  235.8|   0.0;
Dreieck 14: 235.8|   0.0;  241.2|  59.8;  209.6|  42.5;
Dreieck 15: 240.5|  51.1;  235.9|   1.3;  272.0|   0.0;
Dreieck 16: 272.0|   0.0;  278.3|  69.7;  245.7|  42.5;
Dreieck 17: 274.8|  30.7;  272.2|   2.6;  300.4|   0.0;
Dreieck 18: 300.4|   0.0;  325.5|  50.7;  278.0|  66.3;
Dreieck 19: 322.6|  44.8;  300.4|  −0.0;  336.1|   4.6;
Dreieck 20: 337.7|   0.0;  358.5|  37.0;  324.2|  48.2;
Dreieck 21: 359.4|  38.5;  345.5|  13.9;  370.1|   0.0;
Dreieck 22: 370.1|   0.0;  397.9|  49.3;  354.0|  57.8;
Dreieck 23: 400.8|  54.5;  371.4|   2.2;  416.1|   0.0;
Dreieck 24: 416.1|   0.0;  443.8|  49.3;  399.9|  57.8;
Dreieck 25: 446.7|  54.5;  417.3|   2.2;  462.0|   0.0;
Dreieck 26: 462.0|   0.0;  496.3|  61.0;  446.7|  54.5;
Dreieck 27: 489.1|  48.2;  468.3|  11.2;  502.5|   0.0;
Dreieck 28: 502.5|   0.0;  537.6|  48.7;  493.4|  55.8;
Dreieck 29: 543.4|  56.8;  502.5|   0.0;  552.5|   1.0;
Dreieck 30: 552.7|   0.0;  575.8|  44.3;  539.8|  79.0;
Dreieck 31: 580.9|  54.0;  553.1|   0.8;  597.8|   0.0;
Dreieck 32: 597.8|   0.0;  614.5|  47.1;  579.9|  57.2;
Dreieck 33: 616.9|  54.1;  598.1|   0.8;  642.8|   0.0;
Dreieck 34: 642.8|   0.0;  652.6|  49.0;  608.3|  72.2;
Dreieck 35: 652.8|  49.9;  644.5|   8.3;  686.1|   0.0;
Dreieck 36: 686.1|   0.0;  688.9|  49.9;  641.7|  66.6;
Dreieck 37: 688.9|  49.9;  686.1|   0.0;  722.1|   0.0;
Dreieck 38: 722.1|   0.0;  733.2|  55.5;  677.8|  66.6;
Dreieck 39: 733.2|  55.5;  722.1|   0.0;  766.5|   5.5;
Dreieck 40: 770.2|   0.0;  764.7|  44.4;  731.4|  58.3;
Dreieck 41: 770.2|   0.0;  786.3|  32.2;  764.0|  49.6;
Dreieck 42: 790.2|  40.0;  770.2|   0.0;  810.2|   0.0;
Dreieck 43: 810.2|   0.0;  832.6|  44.7;  787.9|  44.7;
```

```
Dreieck 44:  832.6|  44.7;  810.2|   0.0;  854.9|   0.0;
Dreieck 45:  854.9|   0.0;  877.3|  44.7;  828.1|  53.7;
Dreieck 46:  877.3|  44.7;  854.9|   0.0;  881.8|   8.9;
Dreieck 47:  882.9|   0.0;  897.8|  42.2;  875.5|  59.5;
Dreieck 48:  906.2|  66.0;  882.9|   0.0;  911.7|  21.6;
Dreieck 49:  895.5|   0.0;  945.5|   0.0;  925.5|  40.0;
Eine graphische Darstellung wurde in die Datei "dreieckergebnis.svg"
    geschrieben.
```

4.8 Beispiel mit 100 Dreiecken

Abbildung 16: Ein eigenes Beispiel mit 100 Dreiecken, wobei mit 12 Dreiecken pro Paket gerechnet wurde (Die Größe des abgebildeten Bereichs wurde entsprechend vergrößert)

```
Standardmaessig hat das Bild beim Speichern in der Datei einen xoffset
    von 300 die Breite 1250 und die Hoehe 500
Jetzt wurden die Dreiecke von 0 bis 12 gerechnet und der bisherige
    Abstand ist 108.16m.
Jetzt wurden die Dreiecke von 12 bis 24 gerechnet und der bisherige
    Abstand ist 273.86m.
Jetzt wurden die Dreiecke von 24 bis 36 gerechnet und der bisherige
    Abstand ist 467.14m.
Jetzt wurden die Dreiecke von 36 bis 48 gerechnet und der bisherige
    Abstand ist 669.06m.
Jetzt wurden die Dreiecke von 48 bis 60 gerechnet und der bisherige
    Abstand ist 861.15m.
Jetzt wurden die Dreiecke von 60 bis 72 gerechnet und der bisherige
    Abstand ist 1092.00m.
Jetzt wurden die Dreiecke von 72 bis 84 gerechnet und der bisherige
    Abstand ist 1279.82m.
Jetzt wurden die Dreiecke von 84 bis 96 gerechnet und der bisherige
    Abstand ist 1484.41m.
```

```
Jetzt wurden die Dreiecke von 96 bis 100 gerechnet und der bisherige
    Abstand ist 1549.10m.
                 x1|    y1;     x2|    y2;     x3|    y3;
Dreieck  0:    0.0|   0.0;   -30.0|  40.0;   -60.0|   0.0;
Dreieck  1:    0.0|   0.0;    -8.0|  44.0;   -42.0|  56.0;
Dreieck  2:    0.0|   0.0;    12.5|  42.9;   -10.7|  59.0;
Dreieck  3:   10.1|  34.6;    0.0|   0.0;    35.8|   4.0;
Dreieck  4:   38.2|   0.0;    34.8|  44.6;     7.4|  51.5;
Dreieck  5:   38.2|   0.0;    55.1|  41.4;    32.9|  69.8;
Dreieck  6:   57.1|  46.3;    38.2|   0.0;    82.8|   3.4;
Dreieck  7:   84.8|   0.0;   103.8|  46.3;    54.0|  51.5;
Dreieck  8:   94.3|  23.1;    84.8|   0.0;   107.1|   1.7;
Dreieck  9:  108.2|   0.0;   102.6|  34.6;    94.3|  23.1;
Dreieck 10:  108.2|   0.0;   132.0|  65.8;   103.0|  44.4;
Dreieck 11:  125.2|  47.0;   108.2|  -0.0;   152.6|   5.2;
Dreieck 12:  156.0|   0.0;   147.9|  69.5;   131.5|  37.4;
Dreieck 13:  156.0|   0.0;   172.4|  32.1;   149.0|  59.6;
Dreieck 14:  176.5|  40.2;   156.1|   0.3;   196.1|   0.0;
Dreieck 15:  196.1|   0.0;   196.5|  44.7;   165.4|  62.9;
Dreieck 16:  196.4|  34.5;   196.2|   5.3;   216.7|   0.0;
Dreieck 17:  216.7|   0.0;   216.4|  60.0;   196.5|  39.9;
Dreieck 18:  216.7|   0.0;   236.6|  20.1;   216.5|  40.0;
Dreieck 19:  236.5|  20.0;   226.5|   9.9;   236.6|   0.0;
Dreieck 20:  236.6|   0.0;   256.6|  40.0;   236.6|  70.0;
Dreieck 21:  259.0|  44.7;   236.6|  -0.0;   272.4|   4.5;
Dreieck 22:  273.9|   0.0;   299.2|  50.6;   251.7|  66.4;
Dreieck 23:  300.7|  53.7;   273.9|  -0.0;   318.6|   0.0;
Dreieck 24:  318.6|   0.0;   334.4|  47.4;   293.3|  75.9;
Dreieck 25:  333.5|  44.7;   320.1|   4.5;   355.9|   0.0;
Dreieck 26:  355.9|   0.0;   375.8|  52.9;   331.1|  54.7;
Dreieck 27:  366.9|  29.2;   360.5|  12.4;   373.6|   0.0;
Dreieck 28:  373.6|   0.0;   407.5|  36.8;   371.8|  42.4;
Dreieck 29:  407.5|  36.8;   373.6|   0.0;   401.9|   1.2;
Dreieck 30:  408.3|  41.9;   401.7|   0.0;   437.7|   1.5;
Dreieck 31:  438.8|   0.0;   439.5|  50.0;   391.7|  64.6;
Dreieck 32:  439.2|  28.6;   438.9|   0.4;   467.1|   0.0;
Dreieck 33:  467.1|   0.0;   458.0|  49.2;   442.0|  25.8;
Dreieck 34:  467.1|   0.0;   489.3|  44.8;   456.1|  59.0;
Dreieck 35:  493.7|  53.8;   467.1|   0.0;   511.9|   0.2;
Dreieck 36:  511.9|   0.0;   537.0|  50.7;   486.3|  75.8;
Dreieck 37:  540.0|  56.8;   515.0|   6.1;   559.3|   0.0;
Dreieck 38:  559.3|   0.0;   574.8|  47.5;   533.6|  75.8;
Dreieck 39:  568.6|  28.4;   560.8|   4.7;   578.2|   0.0;
Dreieck 40:  578.2|   0.0;   606.0|  32.0;   574.0|  59.9;
Dreieck 41:  606.0|  32.0;   579.7|   1.8;   604.7|   0.0;
```

```
Dreieck 42:  605.9|  29.4;  604.7|   1.2;  633.0|   0.0;
Dreieck 43:  633.0|   0.0;  630.2|  49.9;  581.5|  61.2;
Dreieck 44:  633.0|   0.0;  660.7|  41.6;  629.6|  59.9;
Dreieck 45:  660.7|  41.6;  633.0|   0.0;  669.0|   0.0;
Dreieck 46:  669.1|   0.0;  700.4|  47.1;  653.3|  78.4;
Dreieck 47:  696.8|  41.6;  669.1|  -0.0;  705.1|   0.0;
Dreieck 48:  705.1|   0.0;  715.9|  22.6;  699.2|  29.4;
Dreieck 49:  717.3|  25.6;  705.7|   1.3;  725.7|  -0.0;
Dreieck 50:  725.7|   0.0;  751.3|  50.4;  700.8|  76.0;
Dreieck 51:  749.8|  47.5;  730.6|   9.7;  765.3|   0.0;
Dreieck 52:  765.3|   0.0;  784.5|  37.8;  749.8|  47.5;
Dreieck 53:  784.6|  38.0;  771.8|  12.8;  797.1|   0.0;
Dreieck 54:  797.1|   0.0;  816.3|  37.8;  778.4|  57.0;
Dreieck 55:  816.9|  39.2;  802.4|  10.6;  825.1|   0.0;
Dreieck 56:  825.1|   0.0;  858.0|  50.2;  813.6|  55.4;
Dreieck 57:  852.5|  41.8;  825.1|   0.0;  861.1|   0.3;
Dreieck 58:  861.1|   0.0;  898.5|  42.4;  857.4|  59.9;
Dreieck 59:  907.4|  52.5;  861.1|   0.0;  903.5|   2.7;
Dreieck 60:  906.7|  43.7;  903.9|   7.7;  931.1|   0.0;
Dreieck 61:  931.1|   0.0;  940.5|  69.4;  906.7|  43.7;
Dreieck 62:  938.7|  56.1;  931.1|   0.0;  975.0|   8.3;
Dreieck 63:  981.4|   0.0;  980.8|  70.0;  951.1|  39.8;
Dreieck 64:  981.4|   0.0; 1011.1|  40.2;  980.9|  60.0;
Dreieck 65: 1012.4|  42.0;  982.7|   1.7; 1018.7|   0.0;
Dreieck 66: 1018.7|   0.0; 1042.4|  44.0; 1009.7|  59.3;
Dreieck 67: 1042.4|  44.0; 1018.7|  -0.0; 1054.6|   3.4;
Dreieck 68: 1055.6|   0.0; 1082.4|  49.8; 1032.6|  76.6;
Dreieck 69: 1084.0|  52.8; 1055.6|  -0.0; 1091.5|   3.4;
Dreieck 70: 1092.0|   0.0; 1127.6|  48.3; 1083.6|  55.9;
Dreieck 71: 1133.6|  56.3; 1092.0|   0.0; 1142.0|   0.4;
Dreieck 72: 1142.1|   0.0; 1175.6|  45.5; 1133.1|  59.3;
Dreieck 73: 1171.7|  40.2; 1142.1|   0.0; 1170.0|   4.2;
Dreieck 74: 1175.6|  45.0; 1170.7|   9.2; 1197.4|  -0.0;
Dreieck 75: 1197.4|   0.0; 1198.2|  44.7; 1162.9|  72.2;
Dreieck 76: 1197.9|  28.8; 1197.4|   0.5; 1225.7|   0.0;
Dreieck 77: 1225.7|   0.0; 1211.5|  68.5; 1198.0|  35.1;
Dreieck 78: 1225.7|   0.0; 1258.7|  50.1; 1214.3|  55.4;
Dreieck 79: 1253.2|  41.8; 1225.7|  -0.0; 1261.8|   0.2;
Dreieck 80: 1261.8|   0.0; 1279.8|  31.3; 1258.5|  49.9;
Dreieck 81: 1274.3|  21.7; 1261.8|  -0.0; 1279.8|   1.2;
Dreieck 82: 1279.8|   0.0; 1322.2|  42.5; 1279.8|  56.6;
Dreieck 83: 1329.3|  49.5; 1279.8|  -0.0; 1315.2|   7.1;
Dreieck 84: 1329.2|  49.3; 1313.4|   1.9; 1363.4|   0.0;
Dreieck 85: 1363.4|   0.0; 1373.5|  55.7; 1329.2|  49.3;
Dreieck 86: 1368.2|  26.5; 1363.4|   0.0; 1387.9|   5.0;
```

```
Dreieck 87: 1392.5| 0.0; 1389.4| 59.9; 1370.4| 38.9;
Dreieck 88: 1392.5| 0.0; 1420.5| 64.1; 1390.2| 44.7;
Dreieck 89: 1412.5| 45.8; 1392.5| 0.0; 1437.1| 2.3;
Dreieck 90: 1438.4| 0.0; 1435.3| 59.9; 1416.4| 38.9;
Dreieck 91: 1438.4| 0.0; 1456.4| 41.0; 1435.3| 59.9;
Dreieck 92: 1458.5| 45.8; 1438.4| 0.0; 1483.1| 2.3;
Dreieck 93: 1484.4| 0.0; 1482.1| 44.7; 1449.9| 60.9;
Dreieck 94: 1484.4| 0.0; 1508.4| 55.0; 1482.1| 44.7;
Dreieck 95: 1504.4| 45.8; 1484.4| 0.0; 1529.1| 2.3;
Dreieck 96: 1530.4| 0.0; 1546.6| 32.2; 1510.7| 34.8;
Dreieck 97: 1553.0| 44.8; 1540.9| 20.8; 1549.1| 0.0;
Dreieck 98: 1549.1| 0.0; 1593.8| 22.4; 1559.8| 59.0;
Dreieck 99: 1549.1| 0.0; 1593.8| -0.0; 1611.7| 31.3;
Eine graphische Darstellung wurde in die Datei "dreieckergebnis.svg"
    geschrieben.
```

5 Quellcode

Im Folgenden sind wichtige Ausschnitte aus dem Quellcode abgedruckt. Weil nicht alles abgedruckt wurde, entspricht die Zeilennummerierung nicht der des kompilierfähigen Quellcodes:

```
// Struktur, die fuer die Beschreibung eines polygonfoermigen
    Hindernisses benoetigt wird
typedef struct Dreieck
{
    Punkt ecken[3];
    Gerade strecken[3];
    int modus;
} Dreieck;

int n_dreiecke = 0;             // tatsaechliche Anzahl der Dreiecke
Dreieck dreiecke[MAXDREIECKE]; // Array, in dem alle Dreiecke
    gespeichert werden

short besteanordnung[MAXDREIECKE];       // In diesem Array wird die
    beste bisher bekannte Reihenfolge der Dreiecke gespeichert
short besteanordnung_modi[MAXDREIECKE]; // In diesem Array werden die
    Modi der Dreiecke der besten bisher bekannten Reihenfolge
    gespeichert
short laufanordnung[MAXDREIECKE];        // In diesem Array wird die
    aktuell ausprobierte Reihenfolge der Dreiecke gespeichert
```

```
16 double geringster_abstand = 10000; // Hier wird der geringste bisher
       erreichte Abstand gespeichert
   bool used[MAXDREIECKE];              // Hier wird gespeichert, welche
       Dreiecke bereits fuer eine Anordnung verwendet wurden
18
   // Hier werden die Dreiecke fuer den Beginn sinnvoll sortiert
20 void Dreiecksortieren()
   {
22     for (int i = 0; i < n_dreiecke; i++)
       {
24         for (int j = 0; j < n_dreiecke - 1 - i; j++)
           {
26             if (dreiecke[j].ecken[0].innenwinkel > dreiecke[j + 1].
       ecken[0].innenwinkel)
               {
28                 Dreieck tmp = dreiecke[j];
                   dreiecke[j] = dreiecke[j + 1];
30                 dreiecke[j + 1] = tmp;
               }
32         }
       }
34     for (int i = 0; i < n_dreiecke / 2; i += 2)
       {
36         Dreieck tmp = dreiecke[i];
           dreiecke[i] = dreiecke[n_dreiecke - i - 1];
38         dreiecke[n_dreiecke - i - 1] = tmp;
       }
40 }

42 // Diese Funktion positioniert ein Dreieck mit gegebenen Parametern
   void positionieren(int index, double ey, double ex, double linkswinkel)
44 {
       double rechtswinkel = linkswinkel + dreiecke[index].ecken[0].
       innenwinkel;
46     if (ey > TOLERANZ)
       {
48         rechtswinkel = linkswinkel + PI;
           linkswinkel = rechtswinkel - dreiecke[index].ecken[0].
       innenwinkel;
50     }
       dreiecke[index].ecken[0].x = ex;
52     dreiecke[index].ecken[0].y = ey;
       dreiecke[index].ecken[1].x = ex + dreiecke[index].strecken[0].
       richtung.laenge * sin(rechtswinkel);
```

```
    dreiecke[index].ecken[1].y = ey + dreiecke[index].strecken[0].
    richtung.laenge * cos(rechtswinkel);
    dreiecke[index].ecken[2].x = ex + dreiecke[index].strecken[2].
    richtung.laenge * sin(linkswinkel);
    dreiecke[index].ecken[2].y = ey + dreiecke[index].strecken[2].
    richtung.laenge * cos(linkswinkel);
}

// Diese Funktion bestimmt den Winkel, der das vorherige Dreieck auf
    der rechten Seite begrenzt
double vorherrechtswinkel(int i)
{
    int preindex = laufanordnung[i - 1];
    int vorherrelevantecke = (dreiecke[preindex].ecken[0].y < TOLERANZ)
    ? 0 : 2;
    return myatan(vektorbestimmen(dreiecke[preindex].ecken[
    vorherrelevantecke], dreiecke[preindex].ecken[(vorherrelevantecke +
    1) % 3]));
}

// Fuer die Anordnung mit der Spitze nach unten ermittelt diese
    Funktion den Winkel, unter dem das Dreieck gerade noch angeordnet
    werden darf (zweite Ueberschneidungspraevention)
double winkel_gegen_ueberschneidung(int j, double l, double ex)
{
    int hilfsindex = laufanordnung[j];
    int hilfsrelevant = (dreiecke[hilfsindex].ecken[0].y < TOLERANZ) ?
    0 : 2;
    double qx = ex - dreiecke[hilfsindex].ecken[hilfsrelevant].x;
    if (qx == 0)
        return -PI / 2;
    double qy = -dreiecke[hilfsindex].ecken[hilfsrelevant].y;
    Vektor v = vektorbestimmen(dreiecke[hilfsindex].ecken[hilfsrelevant
    ], dreiecke[hilfsindex].ecken[(hilfsrelevant + 1) % 3]);
    double unter_wurzel = pow(v.deltax, 2) * (-pow((qy * v.deltax - qx
    * v.deltay), 2) + pow(l, 2) * (pow(v.deltax, 2) + pow(v.deltay, 2))
    );
    double wurzel = sqrt(unter_wurzel);
    double tmpdivisor = pow(v.deltax, 2) + pow(v.deltay, 2);
    double faktor = qx + qy * v.deltax * v.deltay / tmpdivisor - qx *
    pow(v.deltay, 2) / tmpdivisor + wurzel / tmpdivisor;
    faktor /= v.deltax;
    if (faktor > 0 && faktor < 1)
    {
```

```
84          double divident = v.deltax * (qy * v.deltax * v.deltay - qx *
    pow(v.deltay, 2) + wurzel);
            double divisor = qy * pow(v.deltax, 3) - v.deltay * (qx * pow(v
    .deltax, 2) + wurzel);
86          double quotient = divident / divisor;
            double newangle = -atan(quotient);
88          return newangle;
        }
90      return -PI / 2;
}
92
// Funktion, die fuer die Anordnung mit der Spitze nach oben den Winkel
     ermittelt, unter dem das Dreieck gerade noch angeordnet werden
    darf
94 double konfliktwinkel(int i, int vorherrelevantecke)
{
96      int index = laufanordnung[i];
        int preindex = laufanordnung[i - 1];
98      double startwinkel = -PI / 2;
        for (int j = i - 2; i >= 0; i--)
100     {
            for (int z = 0; z < 3; z++)
102         {
                Punkt hilfspunkt = dreiecke[preindex].ecken[
    vorherrelevantecke];
104             int hilfsindex = laufanordnung[j];
                Gerade test = geradebestimmen(hilfspunkt, dreiecke[
    hilfsindex].ecken[z]);
106             double testwinkel = myatan(test.richtung);
                if (testwinkel > startwinkel && dreiecke[hilfsindex].ecken[
    z].y > TOLERANZ)
108                 startwinkel = myatan(vektorbestimmen(dreiecke[preindex
    ].ecken[vorherrelevantecke], dreiecke[hilfsindex].ecken[z]));
            }
110     }
        return startwinkel;
112 }

114 // Funktion, die die Summe aller verbleibenden Innenwinkel der nullten
    Ecke bildet
double restwinkelberechnen()
116 {
        double winkel = 0;
118     for (int i = 0; i < n_dreiecke; i++)
            if (used[i] == FALSE)
```

```
120             winkel += dreiecke[i].ecken[0].innenwinkel;
    return winkel;
122 }

124 // Funktion, die ein Dreieck mit der Spitze nach unten entsprechend der
    in der Dokumentation erklaerten Regeln anordnet
double spitze_unten(int i)
126 {
    int index = laufanordnung[i];
128     dreiecke[index].modus = 3;
    double min_innenwinkel = dreiecke[index].ecken[0].innenwinkel;
130     double startwinkel = -PI / 2;
    double ex = 0;
132     if (i > 0)
    {
134         int preindex = laufanordnung[i - 1];
        int vorherrelevantecke = (dreiecke[preindex].ecken[0].y <
    TOLERANZ) ? 0 : 2;
136         startwinkel = vorherrechtswinkel(i);
        double restwinkel = restwinkelberechnen() + dreiecke[index].
    ecken[0].innenwinkel;
138         if (restwinkel + startwinkel < PI / 2)
            startwinkel = (PI / 2) - restwinkel;
140         ex = -dreiecke[preindex].ecken[vorherrelevantecke].y * tan(
    startwinkel) + dreiecke[preindex].ecken[vorherrelevantecke].x;

142         // In den nachfolgenden Zeilen wird die erste Moeglichkeit der
    Ueberschneidung verhindert
        for (int j = i - 2; j >= 0; j--)
144         {
            for (int z = 0; z < 3; z++)
146             {
                Punkt hilfspunkt;
148                 int hilfsindex = laufanordnung[j];
                hilfspunkt.x = ex;
150                 hilfspunkt.y = 0;
                Gerade test = geradebestimmen(hilfspunkt, dreiecke[
    hilfsindex].ecken[z]);
152                 double testwinkel = myatan(test.richtung);
                if (test.richtung.laenge <= dreiecke[index].strecken
    [2].richtung.laenge && testwinkel > startwinkel && dreiecke[
    hilfsindex].ecken[z].y > TOLERANZ)
154                 {
                    startwinkel = myatan(vektorbestimmen(dreiecke[
    preindex].ecken[vorherrelevantecke], dreiecke[hilfsindex].ecken[z])
```

```
    );
156                 ex = dreiecke[preindex].ecken[vorherrelevantecke].y
     * tan(betrag(startwinkel)) + dreiecke[preindex].ecken[
    vorherrelevantecke].x;
                }
158         }
        }
160     // Ab hier wird die zweite Moeglichkeit der Ueberschneidungen
    verhindert
        for (int j = i - 2; j >= 0; j--)
162     {
            double newangle1 = winkel_gegen_ueberschneidung(j, dreiecke
    [index].strecken[2].richtung.laenge, ex);
164         double newangle2 = winkel_gegen_ueberschneidung(j, dreiecke
    [index].strecken[0].richtung.laenge, ex) - dreiecke[index].ecken
    [0].innenwinkel;
            double newangle = (newangle1 > newangle2) ? newangle1 :
    newangle2;
166         if (newangle > startwinkel)
                startwinkel = newangle;
168     }
    }
170 positionieren(index, 0, ex, startwinkel);
    return dreiecke[index].ecken[0].x;
172 }

174 double spitze_oben(int i)
{
176     int index = laufanordnung[i];
    dreiecke[index].modus = 4;
178     double min_innenwinkel = dreiecke[index].ecken[0].innenwinkel;
    int laengst_index = (dreiecke[index].strecken[0].richtung.laenge >=
     dreiecke[index].strecken[2].richtung.laenge) ? 0 : 2;
180     int preindex = laufanordnung[i - 1];
    int vorherrelevantecke = (dreiecke[preindex].ecken[0].y < TOLERANZ)
     ? 0 : 2;
182     double startwinkel = vorherrechtswinkel(i);
    double newangle = konfliktwinkel(i, vorherrelevantecke); // Hier
    wird geprueft, ob ein neuer Winkel zur Verhinderung von
    Ueberschneidungen gewaehlt werden muss
184     if (newangle > startwinkel)
        startwinkel = newangle;
186     double dy1 = dreiecke[index].strecken[0].richtung.laenge * cos(
    startwinkel);
```

```
    double dy2 = dreiecke[index].strecken[2].richtung.laenge * cos(
    startwinkel - min_innenwinkel);
188 double dy = (dy1 > dy2) ? dy1 : dy2;
    double ex = dreiecke[preindex].ecken[vorherrelevantecke].x + (dy -
    dreiecke[preindex].ecken[vorherrelevantecke].y) * tan(startwinkel);
190 positionieren(index, dy, ex, startwinkel);
    return ((dreiecke[index].ecken[1].y < TOLERANZ) ? dreiecke[index].
    ecken[1].x : dreiecke[index].ecken[2].x);
192 }

194 // Diese Funktion ordnet Dreiecke mit bekannten Modi an
double standardanordnung(int ziel)
196 {
    double strecke = 0;
198 for (int i = 0; i < ziel; i++)
        used[laufanordnung[i]] = FALSE;
200 for (int i = 0; i < ziel; i++)
    {
202     int index = laufanordnung[i];
        used[index] = TRUE;
204     int modus = dreiecke[index].modus;
        if (modus == 3)
206         strecke = spitze_unten(i);
        else if (modus == 4)
208         strecke = spitze_oben(i);
    }
210 return strecke;
}
212

// Diese Funktion rechnet die Moeglichkeiten der unterschiedlichen
    Anordnungen eines Pakets durch
214 void durchrechnen(int depth, int ausgangsdepth, int zieldepth)
{
216 if (depth == zieldepth)
    {
218     double abstand = standardanordnung(depth);
        geringster_abstand = abstand;
220     for (int i = 0; i < n_dreiecke; i++)
        {
222         besteanordnung[i] = laufanordnung[i];
            besteanordnung_modi[i] = dreiecke[laufanordnung[i]].modus;
224     }
    }
226 else
    {
```

```
228         double restwinkel = restwinkelberechnen();
            double vorherwinkel = (depth > 0) ? vorherrechtswinkel(depth) :
     -PI / 2; // Dieser Winkel wurde im Abschnitt >>Loesungsidee<< als
    Epsilon bezeichnet
230         for (int i = ausgangsdepth; i < zieldepth; i++)
            {
232             if (used[i] == FALSE)
                {
234                 laufanordnung[depth] = i;
                    used[i] = TRUE;
236                 double c_weg = (vorherwinkel <= 0 || restwinkel -
    TOLERANZ <= PI / 2 - vorherwinkel) ? spitze_unten(depth) :
    spitze_oben(depth);
                    if (c_weg < geringster_abstand)
238                     durchrechnen(depth + 1, ausgangsdepth, zieldepth);
                    used[i] = FALSE;
240                 laufanordnung[depth] = -1;
                }
242         }
        }
244 }

246 int anfangsstapel[MAXDREIECKE]; // In diesem Array werden alle Dreiecke
     des Anfangsstapels gespeichert

248 // Diese Funktion bildet den Anfangsstapel, sortiert ihn und ruft zum
    Schluss die Funktion >>durchrechnen<< auf
void anfangsstapeln(int cnum, double gesamtwinkel, int depth, int
    ausgangsdepth, int zieldepth)
250 {
    if (gesamtwinkel < PI / 2 && zieldepth - depth > 1)
252 {
        for (int i = cnum; i < zieldepth; i++)
254     {
            used[i] = TRUE;
256         anfangsstapel[depth] = i;
            anfangsstapeln(i + 1, gesamtwinkel + dreiecke[i].ecken[0].
    innenwinkel, depth + 1, ausgangsdepth, zieldepth);
258         used[i] = FALSE;
        }
260 }
    else
262 {
        int sortierstapel[MAXDREIECKE];
264     for (int i = 0; i < depth; i++)
```

```
            sortierstapel[i] = anfangsstapel[i];
        for (int i = 0; i < depth; i++)
        {
            for (int j = 0; j < depth - i - 1; j++)
            {
                if (dreiecke[sortierstapel[j]].strecken[0].richtung.
    laenge < dreiecke[sortierstapel[j + 1]].strecken[0].richtung.laenge
    )
                {
                    int tmp = sortierstapel[j];
                    sortierstapel[j] = sortierstapel[j + 1];
                    sortierstapel[j + 1] = tmp;
                }
            }
        }
        for (int i = 0; i < depth; i++)
            laufanordnung[i] = sortierstapel[i];
        for (int i = 0; i < depth; i++)
            dreiecke[laufanordnung[i]].modus = 3;
        standardanordnung(depth);
        durchrechnen(depth, ausgangsdepth, zieldepth);
    }
}

int main(int argc, char *argv[])
{
    int dreieckepropaket = 10;
    if (argc == 1)
    {
        printf("Bitte uebergeben Sie den Dateinamen!\n");
        return 0;
    }
    if (argc == 2)
        printf("Standardmaessig wird mit %d Dreiecken pro Paket
    gerechnet.\n", dreieckepropaket);
    else
    {
        dreieckepropaket = atoi(argv[2]);
        if (argc == 6)
        {
            xoffset = atoi(argv[3]) * zoomfaktor;
            breite = atoi(argv[4]) * zoomfaktor;
            hoehe = atoi(argv[5]) * zoomfaktor;
        }
        else
```

```
            printf("Standardmaessig hat das Bild beim Speichern in der
    Datei einen xoffset von %d die Breite %d und die Hoehe %d\n",
    xoffset / zoomfaktor, breite / zoomfaktor, hoehe / zoomfaktor);
308  }
    // Zuerst werden die zur Initialisierung notwendigen Funktionen
    aufgerufen
310  einlesen(argv[1]);
    dreieckstrecken();
312  Dreiecksortieren();
    for (int i = 0; i < MAXDREIECKE; i++)
314      used[i] = FALSE;
    int pakete = n_dreiecke / dreieckepropaket + ((n_dreiecke %
    dreieckepropaket != 0) ? 1 : 0);
316  // Berechnung der einzelnen Pakete wird angeordnet
    for (int durchlauf = 0; durchlauf < pakete; durchlauf++)
318  {
        geringster_abstand = 10000;
320      int ausgangsdepth = durchlauf * dreieckepropaket;
        int zieldepth = ((durchlauf + 1) * dreieckepropaket >
    n_dreiecke) ? n_dreiecke : (durchlauf + 1) * dreieckepropaket;
322      if (durchlauf == 0)
            anfangsstapeln(0, 0, 0, ausgangsdepth, zieldepth);
324      else
            durchrechnen(ausgangsdepth, ausgangsdepth, zieldepth);
326      for (int i = 0; i < zieldepth; i++)
        {
328          laufanordnung[i] = besteanordnung[i];
            dreiecke[laufanordnung[i]].modus = besteanordnung_modi[i];
330          used[i] = TRUE;
        }
332      standardanordnung(zieldepth);
        printf("Jetzt wurden die Dreiecke von %d bis %d gerechnet und
    der bisherige Abstand ist %.2lfm.\n", ausgangsdepth, zieldepth,
    geringster_abstand);
334  }
    dateischreiben(n_dreiecke);
336  return 0;
}
```