Virtual Tailor

Studienarbeit

vorgelegt von

Katrin Michels
Matrikelnr. 200210262

und

Kristina Gans-Eichler
Matrikelnr. 200210082

Institut für Computervisualistik
Arbeitsgruppe Computergrafik

Betreuer: Fernando Pedro Birra
Prüfer: Prof. Dr.-Ing. Stefan Müller

Oktober 2004

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Simulation von Kleidung im Computer für Spiele und Filme wird von Jahr zu Jahr besser. Die Zuschauer im Kino oder Spielejunkies können sich an realistischen Animationen und naturgetreuen Simulationen erfreuen. Der Grund für die immer bessere Qualität der Simulation sind u.a. die Umsetzung z.B. von Gravitations- und Wind-Kräften und Kollisionsbehandlung bzw. Vermeidung von Selbstdurchdringungen. Realistische Bewegungen werden auch durch korrekte Haft- und Gleitreibung hervorgerufen. Die möglichst echt wirkende Kleidung am animierten Charakter ist heute nicht mehr wegzudenken.

1.2 Ziel

Das Ziel dieser Studienarbeit ist es, in der Lage zu sein, Kleidungs-Meshes zu zeichnen und zu designen, die in einem Kleidungssimulator benutzt werden können. Dabei gibt es zwei verschiedene Anforderungen. Erstens erwartet jedes Kleidersimulatorpackage als Eingabe ein Mesh. Also ist ein einfaches CAD System, das dem Benutzer erlaubt seine eigenen Kleidungsstücke zu zeichnen obligatorisch. Ansonsten wäre die Zahl der Situationen, die der Kleidersimulator behandeln könnte sehr begrenzt. Zweitens muss das Kleidungs-Mesh ein 4-8 Mesh sein, da der Kleidersimulator diese Art von Meshes erwartet. 4-8 Meshes haben attraktive Eigenschaften im Hinblick auf dynamic level of detail angewandt auf Kleidersimulation. Um vom Programm ein 4-8 Mesh als Ausgabe zu erhalten, stehen zwei Alternativen zur Verfügung:

1. Gleich zu Beginn generiert das Programm ein 4-8 Mesh (schwierig zu implementieren)
2. Zuerst wird ein trianguliertes Mesh generiert, welches dann in ein 4-8 Mesh umgewandelt wird

Bei Verwendung der zweiten Alternative kann die Triangle Bibliothek verwendet werden, die sehr vielfältige Möglichkeiten bietet.

2 Grundlagen

2.1 Allgemeines zu Triangulierungen

Definition:[1]

Sei $P = \{P_i = (x_i, y_i) : i \in \{1, ..., n\}\}, n \in N$ eine Menge von n Punkten $P_i \in R^2$ und Ω die konvexe Hülle von P, dann bildet eine Menge $T = \{(a_j, b_j, c_j) : a_j, b_j, c_j \in \{1, ..., n\}\}$ bestehend aus m Punktindextripeln (a_j, b_j, c_j) eine Triangulierung T genau dann, wenn folgende Bedingungen gelten:

1. Die Punkte P_a, P_b, P_c bilden für jedes $j \in \{1, ..., m\}$ die Ecken eines nicht entarteten Dreiecks[2] T_j .

2. Jedes Dreieck wird exakt durch drei Punkte aus P definiert, die Dreiecke enthalten sonst keine Punkte aus P.

3. Der Durchschnitt des Inneren von zwei Dreiecken $T_k, T_l, k, l \in \{1, ..., m\}$ ist leer.

4. Die Vereinigung aller Dreiecke ergibt die konvexe Hülle Ω .

Zur Erinnerung: Wenn alle Verbindungsstrecken der Punkte, die in der Menge liegen, auch in der Menge enthalten sind, dann nennt man die Menge konvex. Die konvexe Hülle einer Punktemenge ist das kleinste konvexe Polygon, das die Punktemenge umschließt, wobei die Eckpunkte der Hülle alle aus der Ausgangsmenge stammen.

Satz: Eine Triangulierung eines einfachen Polygons mit n Ecken besteht aus n-2 Dreiecken.
Zeigen über Induktion:
Induktionsanfang:
Ist $n = 3$, dann ist das Polygon P ein Dreieck $=> 3 - 2 = 1$
Induktionsschritt:
Gegeben sei ein einfaches Polygon P mit n Ecken und eine Triangulierung T. d sei eine Diagonale von P, die in T enthalten ist. Diese Diagonale teilt das Polygon P in zwei Polygone L und R mit l bzw. r Ecken, wobei $l < n$ und $r < n$. Da die Ecken von d zu L sowie auch zu R gehören folgt daraus: $l + r = n + 2$ Nach Induktionsvorraussetzung besitzt jede Triangulierung von L bzw. R $l - 2$ bzw. $r - 2$ Dreiecke. Somit hat $T\,(l - 2) + (r - 2) = n - 2$ Dreiecke.

Bereits die Triangulierung einer Menge von vier Punkten ist nicht unbedingt eindeutig. Liegen zum Beispiel alle vier Punkte auf dem Rand der konvexen Hülle, gibt es zwei Möglichkeiten sie zu triangulieren.

Liegt dagegen ein Punkt innerhalb der konvexen Hülle oder sind drei der Punkte kollinear, ergibt sich nur eine gültige Triangulierung.

Für die Fälle einer Triangulierung von vier Punkten, die nicht eindeutig ist, gibt es verschiedene Entscheidungskriterien.

[1] Oliver Bunsen: Animationsorientierte Optimierung von Polygonen. 1996. siehe [OlBun]
[2] Bei einem entarteten Dreieck fallen alle drei Punkte auf eine Gerade.

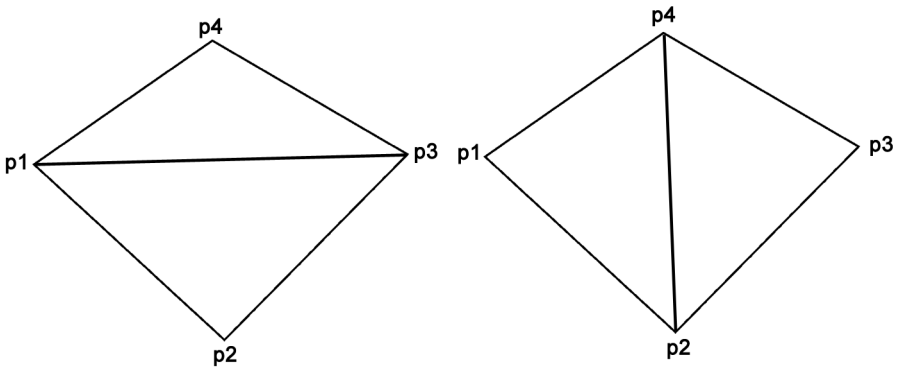

Abbildung 1: Zwei Möglichkeiten vier Punkte zu triangulieren.

1. **Kriterium der kürzeren Diagonalen**
   Gegeben sei eine konvexe Hülle mit den Punkten p_1, p_2, p_3, p_4 . Die Triangulierung T ist eine bessere Triangulierung als T', wenn die Diagonale d der Triangulierung T, die die Punkte p_2 und p_4 verbindet, kürzer ist, als die Diagonale d' der Triangulierung T', die die Punkte p_1 und p_3 der konvexen Hülle verbindet.

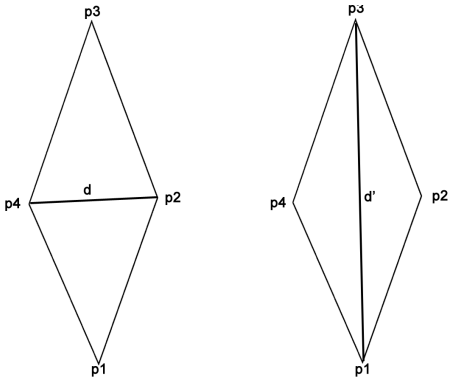

Abbildung 2: Kriterium der kürzeren Diagonalen

2. **Max-Min-Winkelkriterium**
   „Eine Triangulierung T von vier Punkten ist eine bessere Triangulierung als T' genau dann, wenn gilt $a(T) > a(T')$, mit $a(T) = min\{a(T_j)|T_j \in T\}$, wobei $a(T_j)$ den kleinsten Winkel im Dreieck T_j bezeichnet.“ [OlBun]

3. **Min-Max-Winkelkriterium**
   Der größte vorkommende Dreieckswinkel wird minimiert. Nun ist T eine

bessere Triangulierung als T', wenn $a(T) < a(T')$, mit $a(T) = max\{a(T_j)|T_j \in T\}$, wobei $a(T_j)$ den größten Winkel im Dreieck T_j bezeichnet.

4. **Max-Min- und Min-Max-Radiuskriterium**
   Bei diesen Kriterien wird der kleinste Radius der in die Dreiecke einbeschriebenen Kreise maximiert, bzw. der größte Radius der in die Dreiecke einbeschriebenen Kreise minimiert.

5. **Max-Min-Flächenkriterium**
   Bei der Triangulierung von vier Punkten, wird der kleinste Flächeninhalt der beiden Dreiecke maximiert.

6. **Max-Min-Höhenkriterium**
   Maximiere die kleinste Höhe der Dreiecke

7. **minimale Summe der Kantenlängen**
   Hier wird die kleinste Summe der fünf Kantenlängen gesucht.

8. **minimales Verhältnis von Umkreis- zu Dreiecksflächen**
   „Bei der Triangulierung von vier Punkten, wird diejenige Triangulierung gewählt, bei der über beide Dreiecke das Verhältnis der Flächeninhalte der Umkreise um die Dreieckspunkte zu den Dreiecksflächen minimiert wird.“[3]

Eine Triangulierung T wird als *lokal optimal* bezüglich eines Kriteriums bezeichnet, wenn jedes Viereck, welches durch zwei aneinandergrenzende Dreiecke (mit einer gemeinsamen Kante) von T, bezüglich des Kriteriums optimal trianguliert ist. Hierbei können verschiedene lokal optimale Triangulierungen entstehen. In der folgenden Abbildung sind beide Triangulierungen lokal optimal bezüglich des Min-Max-Winkelkriteriums.

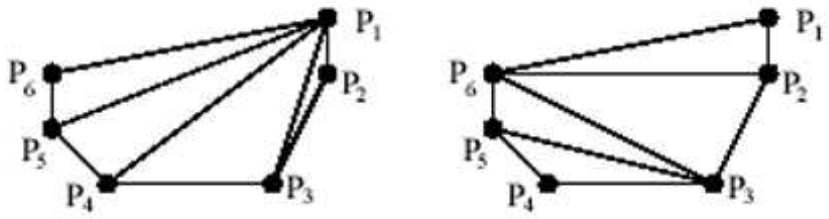

Abbildung 3: lokal optimal bezüglich des Min-Max-Winkelkriteriums

Um zwei lokal optimale Triangulierungen miteinander zu vergleichen, wird der Begriff der global optimalen Triangulierung eingeführt. Dazu werden die lokalen Maße des betrachteten Kriteriums für alle Dreiecke der Triangulierung

[3] siehe [OlBun]

in einem Vektor lexikographisch zusammengefasst und die Vektoren verglichen. Das bedeutet, dass für jede mögliche Triangulierung T einer Punktmenge mit n Dreiecken ein Vektor aus den lokalen Entscheidungskriterien erstellt wird, bei dem die Komponenten der Größe nach sortiert sind. Wenn es keine von T verschiedene Triangulierung gibt, deren Vektor größer (maximierendes Kriterium) oder kleiner (minimierendes Kriterium) ist, dann heißt die Triangulierung T *global optimal*. Ein Vekor $a(T) = (a_1, ..., a_m)$ ist lexikographisch kleiner als der Vektor $a(T')$, wenn es ein $k \in N$ gibt, so dass gilt $a_i = a_{i'}$ für $i = 1, ..., k-1$ und $a_k < a_{k'}$

Ist eine Triangulierung global optimal dann ist sie auch lokal optimal. Den Umkehrschluss kann manF nicht ziehen. Nur bei dem Max-Min-Winkelkriterium kann man aus einer lokal globalen Triangulierung auf eine global optimale Triangulierung schließen.

2.2 Triangle

Triangle ist ein in c geschriebenes, frei (http://www.cs.cmu.edu/~quake/triangle.html) erhältliches Programm zur Generierung von zweidimensionalen Netzen. „Triangle generates exact Delaunay triangulations, constrained Delaunay triangulations, Voronoi diagrams, and quality conforming Delaunay triangulations.“[4]

Im ersten Teil dieses Kapitels werden einige allgemeine Begriffe zur Delaunay Triangulierung erklärt. Desweiteren wird kurz auf drei Algorithmen eingegangen, die Triangle zur Triangulierung einer Punktemenge benutzt, sowie auf einen Algorithmus für das Refinement dieser Triangulierung. Der zweite Teil befasst sich mit einigen Komponenten zur Nutzung von Triangle.

2.2.1 Begriffserklärung

Zur weiteren Erklärung einige Begriffe vorab:

Planar Straight Line Graph (PSLG) Ein Planar Straight Line Graph ist eine Sammlung von Punkten und Segmenten. Segmente sind Kanten, deren Endpunkte Punkte im PSLG sind und deren Erscheinung in einem aus einem PSLG erzeugten Netz zwingend ist.

Delaunay Triangulierung Die Delaunay Triangulierung einer Punktemenge ist die Triangulierung einer Punktemenge mit der Eigenschaft, dass kein Punkt der Punktemenge in den Umkreis, der Kreis der durch alle drei Punkte geht, irgendeines Dreiecks fällt (Kreis-Kriterium). Durch dieses Kriterium wird der minimale Winkel der Dreiecke maximiert.

Constrained Delaunay Triangulierung Eine bedingte Delaunay Triangulierung eines PSLG ist gleich einer Delaunay Triangulierung, aber jedes Segment des PSLG erscheint als einzelne Kante in der Triangulierung. Das heißt, eine bedingte Delaunay Triangulierung hat keine anderen Punkte als die, die durch den gegebenen PSLG spezifiziert wurden. Desweiteren darf ein Punkt p im Umkreis eines anderen Dreiecks liegen, wenn p von mindestens einem Eckpunkt des Dreiecks nicht „gesehen“ werden kann (nicht „sichtbar“ ist). Das heißt, die Verbindung dieses Eckpunktes mit p schneidet eine Kante des Dreiecks.

Conforming Delaunay Triangulierung Eine konforme Delaunay Triangulierung eines PSLG ist eine „wahre“ Delaunay Triangulierung in der jedes Segment des PSLG in mehrere Kanten unterteilt werden kann, indem neue Punkte, sogenannte Steiner Punkte, eingefügt werden. Diese Punkte sind notwendig, um die Delaunay Eigenschaft zu erhalten, während gesichert bleibt, dass jedes Segment repräsentiert wird.

[4] Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. Applied Computational Geometry (Philadelphia, Pennsylvania), pages 124-133, ACM, May 1996. Abrufbar im Internet. URL:http://www-2.cs.cmu.edu/~quake/tripaper/triangle1.html

Constrained conforming Delaunay Triangulierung Eine bedingte konforme Delaunay Triangulierung eines PSLG ist eine bedingte Delaunay Triangulierung, die Steiner Punkte beinhaltet. Normalerweise werden weniger Punkte benötigt um eine gute bedingte konforme Delaunay Triangulierung zu erzeugen als eine gute konforme Delaunay Triangulierung, denn die Dreiecke müssen nicht alle die Bedingung einer Delaunay Triangulierung erfüllen.

Voronoi Diagramm Ein Voronoi Diagramm zu einer Menge von Punkten in der Ebene ist die Unterteilung der Ebene in polygonale Regionen derart, dass zu jedem Punkt p aus der Punktemenge die Region definiert ist, die alle Punkte q der Ebene enthält, die zu p näher sind als zu jedem anderen Punkt aus der Punktemenge. Eng verbunden mit dem Voronoi Diagramm ist die Delaunay Triangulierung. Nimmt man ein Voronoi Diagramm als Basis erhält man die Delaunay Triangulierung, indem man je zwei Mittelpunkte von zwei benachbarten Regionen verbindet.

Es gibt verschiedene Algorithmen um eine Delaunay Triangulierung durchzuführen. Diese werden in zwei Gruppen eingeteilt, in statische und dynamische Algorithmen. Statisch bedeutet, dass das Kreis-Kriterium erst erfüllt ist, wenn alle Punkte in das Netz eingefügt sind. Dynamisch dagegen heißt, dass zu allen Zeitpunkten ein gültiges Netz vorliegt. In Triangle sind drei dieser Algorithmen implementiert, der incremental insertion Algorithmus (dynamisch), der divide-and-conquer Algorithmus (statisch) und der sweep line Algorithmus (statisch). „I believe that Triangle is the first instance in which all three algorithms have been implemented with the same data structures and floating-point tests, by one person who gave roughly equal attention to opzimizing each.“[5]

Incremental insertion Als erstes wird eine Ausgangstriangulierung erzeugt (Bild 1 von Abbildung 4). Zu diesem Zweck wird die konvexe Hülle der Punkte trianguliert. Für diese Ausgangstriangulierung muss das „maximum angle“ Kriterium erfüllt sein. Der erste Punkt wird in das Netzwerk eingefügt. Dazu wird er mit den drei Eckpunkten des Dreiecks verbunden, in dem er liegt (Bild 2 von Abbildung 4). Die Dreiecke der Vierecke, für die die alte Seite des Dreiecks eine Diagonale war, werden auf das Kreiskriterium hin untersucht. Ist dieses nicht erfüllt, werden die Diagonalen getauscht (Bild 3 von Abbildung 4). Alle weiteren Punkte werden in der gleiche Art und Weise in das Netzwerk eingefügt.

[5]Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. Applied Computational Geometry (Philadelphia, Pennsylvania), pages 124-133, ACM, May 1996. Abrufbar im Internet. URL: http://www-2.cs.cmu.edu/~quake/tripaper/triangle2.html

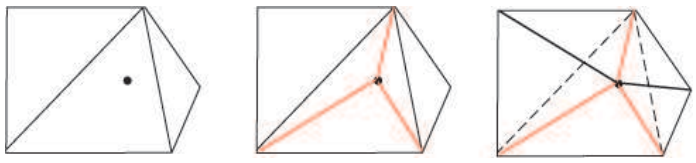

Abbildung 4: Incremental insertion

Divide-and-Conquer Zuerst wird die Menge der gegebenen Punkte in zwei Teilmengen geteilt, wobei darauf geachtet wird, dass in beiden Teilmengen die gleiche Anzahl an Punkten vorhanden ist. Diese Teilmengen werden dann solange geteilt, bis jede Teilmenge so wenig Punkte wie möglich erhält, mindestens jedoch vier Punkte. Durch Teilen an der kürzeren Diagonalen werden zwei Dreiecke erzeugt. Anschließend werden die Teilregionen rekursiv zusammengesetzt. Um eine Grenze zwischen den Teilregionen festzulegen wird die konvexe Hülle jeder Teilmenge benutzt.

Plane-sweep Eine imaginäre Linie, die sogenannte *sweep line* (Fegelinie), wird über eine Menge von geometrischen Objekten bewegt. Da sich die sweep line ausschließlich in eine Richtung bewegt, wird verhindert, dass ein Objekt mehr als ein Mal erfasst wird. Nur an bestimmten Punkten wird die sweep line angehalten und ein Teil des Problems wird an dieser Stelle gelöst.

Im sweep line Algorithmus von Fortune wird für einen Punkt p, der in der Halbebene L+ über der sweep line liegt, eine Parabel mit Fokus p gebildet. Diese Parabel besteht aus Punkten, deren Abstand zu p gleich dem Abstand des Punktes zur sweep line ist. *Beach line* wird die Linie genannt, die die untere äußere Umrandung der Parabeln für die Punkte in L+ bildet. Bewegt sich die sweep line weiter nach unten, folgt die beach line. Die Punkte, die zu einem Stück der beach line beitragen, nennt man aktiv. Die Punkte, die auf der beach line liegen und in denen sich zwei Parabeln treffen, werden Knick genannt. Punkte an denen sich ein neuer Parabelbogen bildet oder an denen ein Parabelbogen zusammenschrumpft, nennt man Ereignispunkte. Wenn eine neue Parabel entsteht, spricht man von einem site event, verschwindet eine Parabel wird das als circle event bezeichnet.

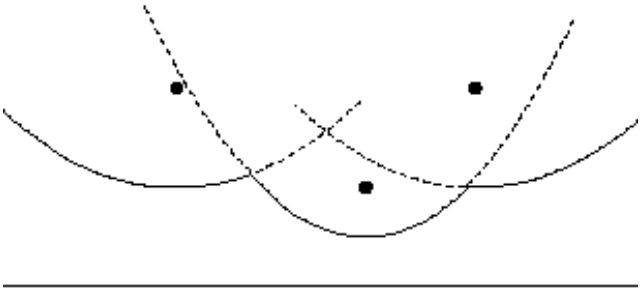

Abbildung 5: beach line (ist entnommen aus [sieg04])

Site event Bewegt sich die sweep line abwärts und erreicht einen neuen Punkt, wird eine vertikale Linie von der sweep line zur beach line gezogen. Man spricht hier auch von einer degenerierten Parabel mit der Breite Null. Schreitet die sweep line weiter fort, weitet sich die Parabel aus und auf der beach line entstehen zwei neue Schnittpunkte. Diese beiden Schnittpunkte bewegen sich während des Fortschreitens der sweep Line auf den Mittelsenkrechten zu der Verbindung der beiden Punkten, die jeweils die sich schneidenden Parabeln definieren. Eine neue Voronoi-Kante beginnt sich zu bilden.

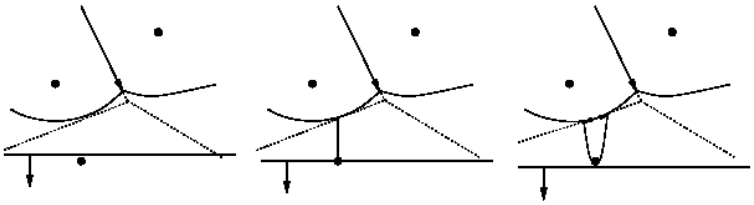

Abbildung 6: site event (ist entnommen aus [mapv04])

Circle event Wenn sich drei aufeinanderfolgende Parabelbögen, die auf der beach line liegen, in einem Punkt q schneiden tritt ein *circle event* auf. Das bedeutet, dass die zwei äußeren Parabeln über die dritte, in der Mitte liegende Parabel hinauswachsen, und diese somit entfernt wird. Die drei Punkte, durch die die Parabelbögen definiert wurden liegen auf einem Kreis mit dem Mittelpunkt q. Abbildung 7 ist entnommen aus [6]

[6]Marc Wilhelm: Generierung von Delaunay Triangulationen. 2000. URL: http://www.uni-stuttgart.de/iv-kib/generic/download/diplom/mwilhelm/Diplomtext.pdf Stand Dezember 2004

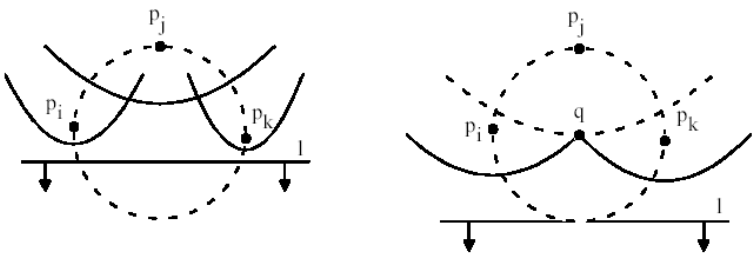

Abbildung 7: circle event

Der Punkt q ist ein neuer Knoten im Voronoi Diagramm. Eine neue Voronoi-Kante entsteht, wenn die neue Mittelsenkrechte der Line zwischen den beiden Punkten, die die beiden verbleibenden Parabeln definieren, verfolgt wird.

Ist die sweep line über alle Punkte „gefahren“, ist ein Voronoi Diagramm mit Voronoi-Kanten und Voronoi-Knoten entstanden.

Refinement der Triangulierung Um die Delaunay Triangulierung zu verfeinern verwendet Triangle „Ruppert's Delaunay Refinement Algorithm“.

Der Input für Triangle ist ein Planar Straight Line Graph (PSLG), der aus Segmenten und Punkten besteht. Im ersten Schritt wird eine Delauny Triangulierung der Punkte gesucht. Da einige der Input-Segmente nach der Triangulierung fehlen, werden im zweiten Schritt die fehlenden Segmente eingefügt. Dies kann auf zwei Arten geschehen:

1. In das Netz wird ein neuer Punkt eingefügt. Dieser Punkt ist der Mittelpunkt eines fehlenden Segmentes. Um die Delaunay Eigenschaft zu bewahren, wird zum Einfügen des Punktes der incremental insertion Algorithmus von Lawson benutzt. Das Segment wird durch diese Prozedur in zwei Hälften gespalten. Ist dies geschehen wird überprüft ob, die beiden Subsegmente im Netz auftauchen. Ist dies nicht der Fall wird der Vorgang solange rekursiv wiederholt, bis das ursprüngliche Segment durch eine lineare Sequenz von Kanten dargestellt wird.
2. Es wird eine constrained Delaunay Triangulation benutzt. Jedes Segment wird eingefügt, in dem die Dreiecke die es überlappen gelöscht werden und die Regionen auf jeder Seite des Segments neu trianguliert werden.

Der dritte Schritt weicht von Rupperts Algorithmus ab. Triangle entfernt in diesem Schritt Dreiecke aus Löchern und Konkavitäten. Im vierten Schritt wird das Netz verfeinert, indem neue Punkte eingefügt werden. Auch dazu wird der incremental insertion Algorithmus von Lawson benutzt um die Delaunay Eigenschaft zu erhalten. Die zwei grundlegenden

Operationen um neue Punkte einzufügen sind, ein Segment an seinem Mittelpunkt und ein Dreieck im Mittelpunkt seines Umkreises zu teilen. Ein Segment wird genau dann an seinem Mittelpunkt geteilt, wenn sich in seinem Durchmesserkreis (diametral circle) ein anderer Punkt befindet. Der Durchmesserkreis eines Segments ist der kleinste Kreis, welches das Segment umfasst.

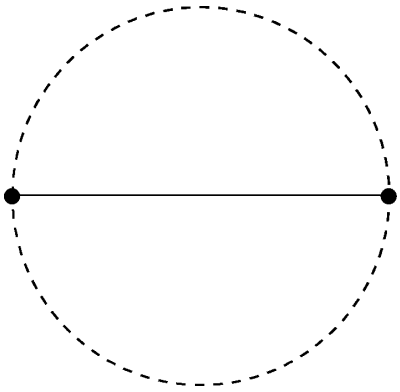

Abbildung 8: Durchmesserkreis eines Segments

Ein Dreieck wird dann gespalten, wenn ein Winkel des Dreiecks zu klein oder der Bereich den das Dreieck umfasst zu groß ist, um den Bedingungen des Nutzers gerecht zu werden. Ein solches Dreieck wird gespalten, indem im Mittelpunkt des Umkreises des Dreiecks ein neuer Punkt eingefügt wird. Das „schlechte“ Dreieck wird gelöscht, da die Delaunay Eigenschaft beibehalten wird.

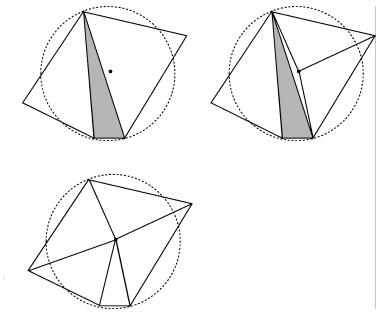

Abbildung 9: Entfernen eines „schlechten“ Dreiecks

Beeinträchtigt der neue Punkt aber ein anderes Segment (er liegt im Durchmesserkreis des Segments), wird er gelöscht und der durchgeführte Prozess des Einfügens dieses Punktes rückgängig gemacht. Statt dessen wird jedes Segment, welches durch den Punkt beeinträchtigt wurde, gespalten.

2.2.2 Datenstrukturen von Triangle

In diesem Abschnitt werden die von Triangle verwendeten Datenstrukturen und implementierten Algorithmen genauer betrachtet. Triangle ist schnell, robust und benötigt wenig Speicher. Als Besonderheiten sind zu erwähnen, dass der Benutzer die Dreiecksgröße und deren Winkel angeben kann. Weiterhin können Löcher und Konkavitität bestimmt werden.

Wie gut ein Dreiecksnetz-Generator ist, hängt davon ab, wie effizient die Triangulierungsalgorithmen arbeiten und wie speicherschonend die Datenstrukturen sind. In Triangle sind drei Algorithmen implementiert, der divide-and-conquer Algorithmus, der sich als schnellster herausstellt, der sweepline Algorithmus und der incremental insertion Algorithmus, der als langsamster abschneidet. Der divide-and-conquer Algorithmus von Dwyer kann durch die Verwendung einer horizontal und vertikal alternierenden Unterteilung der Punktmenge optimiert werden. Bis dahin war dieser Algorithmus nur mit vertikaler Unterteilung implementiert. Mit dieser Optimierung gelang es, die Triangulierung zu beschleunigen. Wenn zusätzlich die Verwendung der exakten Arithmetik abgeschaltet wird, reduziert sich obendrein noch die Fehlerquote.

Die Datenstrukturen, mit denen man arbeitet, sollen möglichst effizient sein. Deshalb stellt sich die Frage, welche Struktur wohl besser ist, eine, die jede Kante speichert oder eine, die einzelne Dreiecke speichert. Triangle wurde ursprünglich mit der quad-edge Datenstruktur von Guibas und Stolfi implementiert, später wurde Triangle dann umgeschrieben, wobei eine Dreiecksdatenstruktur gewählt wurde. Die quad-edge Datenstruktur ist beliebt, weil sie einen Graphen und gleichzeitig sein geometrisches Pendant repräsentieren kann (vergleichbar mit Delauny Triangulierung und zugehörigem Voronoidiagramm). Bei der Verwendung der Dreiecksdatenstruktur hat sich herausgestellt, dass sowohl der divide-and-conquer Algorithmus, der incremental Algorithmus als auch der Delaunay refinement Algorithmus um einen Faktor von zwei beschleunigt werden konnten (allerdings ist der Code auch doppelt so lang).

Jede quad-edge benötigt Speicher für vier Zeiger zu seinen Nachbarn und zwei Zeiger zu den eigenen Endpunkten. Jedes Dreieck speichert drei Zeiger zu seinen Nachbardreiecken und drei Zeiger zu seinen Eckpunkten. Trotz der Tatsache, dass beide Strukturen die gleiche Anzahl an Zeigern besitzen, ist die Dreiecksstruktur effizienter. Das liegt unter anderem daran, dass bei Änderungen im Programm die Geschwindigkeit davon abhängt, wie viele Zeiger gelesen und geschrieben werden müssen. In der Dreiecksstruktur sind die Verbindungen der Dreiecke untereinander impliziter als in der quad-edge Struktur und deshalb ist die Berechnung schneller.

In bestimmten Fällen will man nicht nur Dreiecke darstellen, sondern nur einzelne Kanten. Verwendet man ein degeneriertes Dreieck zur Darstellung einer isolierten Kante, macht das den Code ziemlich unelegant, da das die Notwendigkeit der Behandlung von Spezialfällen zu zur Folge hätte. Um die Eleganz des divide-and-conquer Algorithmus von Guibas und Stolfi zu erhalten, wird jede Triangulierung mit einer Umrandung von Geisterdreiecken (siehe Abbildung 11) versehen, es wird ein Geisterdreieck pro Kante der konvexen Hülle gene-

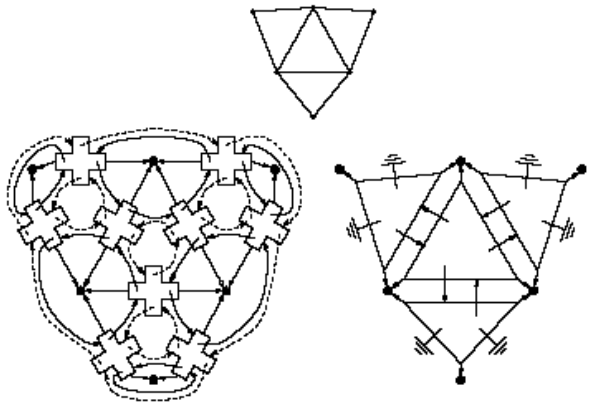

Abbildung 10: Eine Triangulierung und die korrespondierende quad-edge- und Dreiecksstruktur. Jede quad-edge und jedes Dreieck besteht aus sechs Zeigern.

riert. Die Geisterdreicke sind ringartig miteinander verbunden, wobei sie alle einen gemeinsamen Punkt haben, der unendlich weit weg ist (dieser unendlich weit entfernte Punkt ist nur ein Nullpointer). Eine einzelne Kante kann nun von zwei Geisterdreicken repräsentiert werden. Geisterdreicke sind sehr praktisch, um effizient die Kanten der konvexen Hülle zu traversieren. Zwei Triangulierungen können mit Hilfe der Geisterdreicke ganz leicht „zusammengenäht“ werden, indem ihre Geisterdreiecke wie Zähne von Zahnrädern ineinandergreifen. Beim Zusammennähen werden nur zwei neue Dreiecke erzeugt, eins am Anfang und eins am Ende der Naht. Nachdem zwei Triangulierungen verbunden worden sind, können die Geisterdreiecke gelöscht werden.

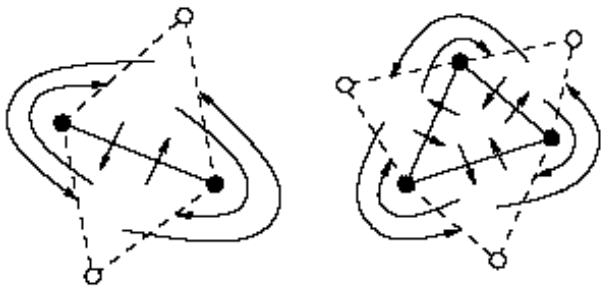

Abbildung 11: Repräsentation einer isolierten Kante (links) und eines isolierten Dreiecks (rechts) im divide-and-conquer Algorithmus. Gestrichelte Linien stellen Geisterdreiecke dar. Weisse Eckpunkte repräsentieren alle denselben Punkt in der Unendlichkeit. Nur die schwarzen Eckpunkte haben Koordinaten.

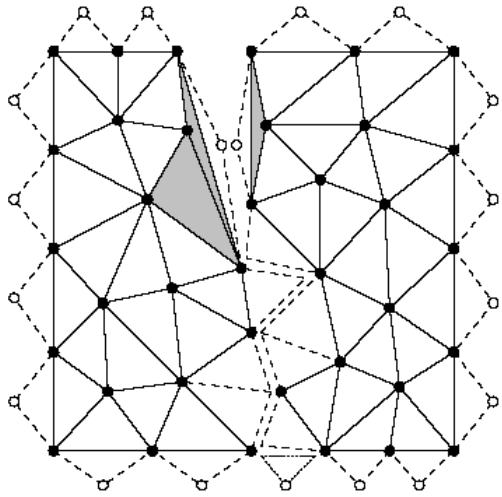

Abbildung 12: merge-Schritt des divide-and-conquer Algorithmus. Gestrichelte Linien repräsentieren Geisterdreiecke und Dreiecke die durch Kantenflips entstanden sind. Das gepunktete Dreieck am unteren Ende ist ein neu generiertes Geisterdreieck. Schattierte Dreiecke sind nicht Delaunay konform und werden durch Kantenflips ersetzt.

Rupperts Algorithmus zur zweidimensionalen Meshgenerierung erstellt Meshes ohne kleine Winkel, benutzt relativ wenige Dreiecke (wobei die Dichte der Dreiecke vom Benutzer auch erhöht werden kann) und erlaubt einen schnellen

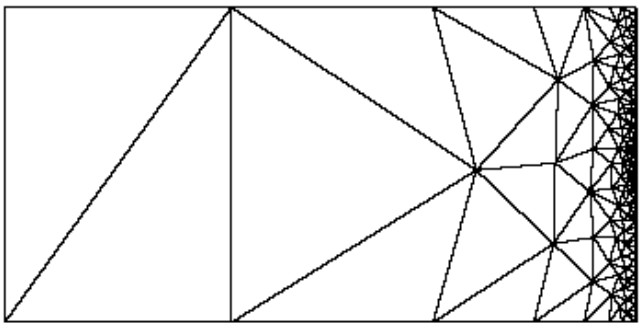

Abbildung 13: An diesem Beispiel sollen die Fähigkeiten des refinement Algorithmus von Ruppert verdeutlicht werden. Trotz der Einschränkung der Winkelgröße, kann der Delaunay refinement Algorithmus Dreiecke mit unterschiedlichen Größen produzieren. Es treten keine Winkel kleiner als 24 Grad auf.

Wechsel der Dreiecksgröße innerhalb kurzer Strecken.

Die verschiedenen Schritte des Delaunay refinement Algorithmus von Ruppert wurden im vorigen Abschnitt bereits erklärt. Hier folgen nun noch einige Abbildungen um die Konzepte PSLG, Delaunaytriangulierung, bedingte Delaunaytriangulierung und conforming Delaunaytriangulierung zu verdeutlichen.

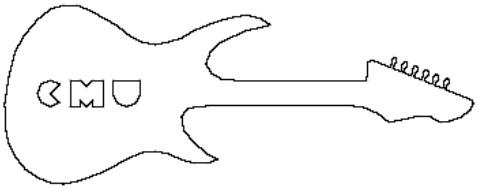

Abbildung 14: Ein anschauliches Beispiel für einen PSLG. Die Gitarre wird durch Punkte und Kanten dargestellt. Rupperts refinement Algorithmus erwartet als Eingabe einen PSLG.

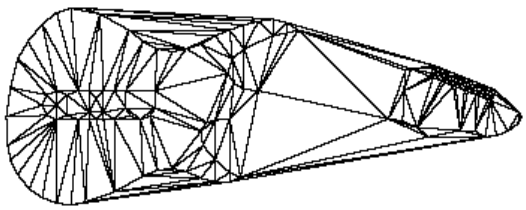

Abbildung 15: Diese Abbildung zeigt die Delaunaytriangulierung des PSLG. Wie am Griff der Gitarre zu erkennen ist, ist die Triangulierung ist nicht zu allen Eingabepunkten konform.

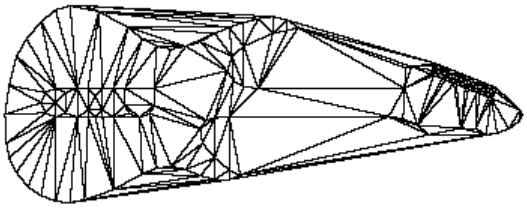

Abbildung 16: Bedingte Delaunaytriangulierung des PSLG. Bei diesem Schritt werden Bereiche mittrianguliert, die sich außerhalb der Figur befinden und nicht zu ihr gehören.

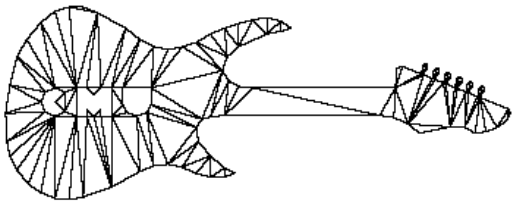

Abbildung 17: Hier ist die Gitarre besser zu erkennen, da die nicht zur Figur gehörenden Dreiecke aus konkaven Bereichen und aus Löchern entfernt wurden.

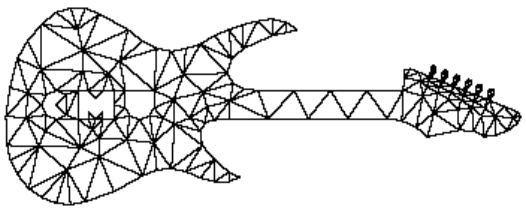

Abbildung 18: Das beste Ergebnis stellt diese Conforming Delaunay Triangulierung mit einem minimalen Winkel von 20 Grad dar.

2.2.3 Triangle nutzen

Normalerweise wird Triangle von der Kommandozeile aus aufgerufen. Die Syntax dazu sieht folgendermaßen aus:
triangle `[-prq__a__uAcjevngBPNEIOXzo_YS__LiFlsCQVh]` inputfile
Unterstriche im obigen Ausdruck bedeuten, dass dem switch eine Zahl folgen kann. Zwischen dem switch und der dazu gehörenden Zahl darf kein Leerzeichen stehen. Als inputfile muss eine Datei mit der Endung .poly oder .node verwendet werden.
Folgend die Übersicht, die auf der Website[7] aufgeführt ist, über die switches, die von Triangle benutzt werden können

-p Triangulates a Planar Straight Line Graph (.poly file).

-r Refines a previously generated mesh.

-q Quality mesh generation with no angles smaller than 20 degrees. An alternate minimum angle may be specified after the 'q'.

-a Imposes a maximum triangle area constraint. A fixed area constraint (that applies to every triangle) may be specified after the 'a', or varying area constraints may be read from a .poly file or .area file.

-u Imposes a user-defined constraint on triangle size.

-A Assigns a regional attribute to each triangle that identifies what segment-bounded region it belongs to.

-c Encloses the convex hull in segments.

-j Jettisons vertices that are not part of the final triangulation from the output .node file (including duplicate input vertices).

-e Outputs (to an .edge file) a list of edges of the triangulation.

-v Outputs the Voronoi diagram associated with the triangulation. Does not attempt to detect degeneracies, so some Voronoi vertices may be duplicated.

-n Outputs (to a .neigh file) a list of triangles neighboring each triangle.

-g Outputs the mesh to an Object File Format (.off) file, suitable for viewing with the Geometry Center's Geomview package.

-B Suppresses boundary markers in the output .node, .poly, and .edge output files.

-P Suppresses the output .poly file. Saves disk space, but you lose the ability to maintain constraining segments on later refinements of the mesh.

[7] http://www-2.cs.cmu.edu/~quake/triangle.switch.html

-N Suppresses the output .node file.

-E Suppresses the output .ele file.

-I Suppresses mesh iteration numbers.

-O Suppresses holes: ignores the holes in the .poly file.

-X Suppresses exact arithmetic.

-z Numbers all items starting from zero (rather than one). Note that this switch is normally overrided by the value used to number the first vertex of the input .node or .poly file. However, this switch is useful when calling Triangle from another program.

-o2 Generates second-order subparametric elements with six nodes each.

-Y Prohibits the insertion of Steiner points on the boundary.

-S Specifies the maximum number of added Steiner points.

-L Use this switch if you want all triangles in the mesh to be Delaunay, and not just constrained Delaunay; or if you want to ensure that all Voronoi vertices lie within the triangulation.

-i Uses the incremental algorithm for Delaunay triangulation, rather than the divide-and-conquer algorithm.

-F Uses Steven Fortune's sweepline algorithm for Delaunay triangulation, rather than the divide-and-conquer algorithm.

-l Uses only vertical cuts in the divide-and-conquer algorithm. By default, Triangle uses alternating vertical and horizontal cuts, which usually improve the speed except with vertex sets that are small or short and wide. This switch is primarily of theoretical interest.

-s Specifies that segments should be forced into the triangulation by recursively splitting them at their midpoints, rather than by generating a constrained Delaunay triangulation. Segment splitting is true to Ruppert's original algorithm, but can create needlessly small triangles. This switch is primarily of theoretical interest.

-C Check the consistency of the final mesh. Uses exact arithmetic for checking, even if the -X switch is used. Useful if you suspect Triangle is buggy.

-Q Quiet: Suppresses all explanation of what Triangle is doing, unless an error occurs.

-V Verbose: Gives detailed information about what Triangle is doing. Add more 'V's for increasing amount of detail. '-V' gives information on algorithmic progress and detailed statistics.

-h Help: Displays complete instructions.

Im Virtual Tailor wird die Funktion `triangulate()` mit den switches „z“, „p“, „n“, „a“, „e“ und „q“ aufgerufen. Diese switches werden im Folgenden genauer erklärt. Wird der „z“ switch benutzt, werden alle Punkte oder andere Elemente von null an gezählt. Das erste Element, egal welchen Typs, startet immer am Index [0] des dazugehörenden Arrays. Der „p“ switch bedeutet, dass ein Planar Straight Line Graph trianguliert wird. Dazu muss eine .poly Datei mit einer bestimmten Grundstruktur übergeben werden.

.poly Datei:

- First line: `<# of vertices> <dimension (must be 2)>`
  `<# of attributes> <# of boundary markers (0 or 1)>`
- Following lines: `<vertex #> <x> <y> [attributes] [boundary marker]`
- One line: `<# of segments> <# of boundary markers (0 or 1)>`
- Following lines: `<segment #> <endpoint> <endpoint> [boundary marker]`
- One line: `<# of holes>`
- Following lines: `<hole #> <x> <y>`
- Optional line: `<# of regional attributes and/or area constraints>`
- Optional following lines: `<region #> <x> <y> <attribute> <maximum area>`

Der erste Teil gibt an, wie viele Punkte vorhanden sind und welche Koordinaten diese haben. Ist `<# of vertices>` auf null gesetzt, bedeutet dies, dass die Punkte seperat in einer .node Datei gespeichert sind. Ist eine Punktemenge derart aufgezeigt, hat dies den großen Vorteil, dass sie einfach mit oder ohne Segmente, abhängig davon ob eine .poly oder eine .node Datei eingelesen wird, trianguliert werden kann. Im zweiten Teil werden die Segmente aufgelistet. Ein Segment ist eine Kante zwischen zwei Punkten des PSLG. Jedes Segment besteht aus den Indizes der beiden Endpunkte. Diese Endpunkte müssen auch in der Punkteliste vorhanden sein.

Der dritte Abschnitt enthält die Informationen über Löcher in der Triangulierung. Jedes Loch wird durch einen Punkt charakterisiert, der innerhalb dieses Loches liegt. Nachdem die Triangulierung beendet ist, entstehen die Löcher indem die Dreiecke aus dem Loch „gegessen“ werden. Von dem Punkt innerhalb des Loches ausgehend werden die Dreiecke so lange gelöscht, bis der Prozess von einem Segment des „Eingabe-PSLG“ gestoppt wird. Das bedeutet, dass jedes Loch von Segmenten umschlossen sein muss, da sonst die gesamte Triangulierung entfernt werden kann.

Der vierte optionale Teil beinhaltet Informationen über regionale Attribute, die auf alle Dreiecke einer Region bezogen sind und regionale Bedingungen bezüglich des maximalen Bereichs den ein Dreieck umfasst. Ein regionales Attribut kann zum Beispiel eine physikalische Größe sein.

Dieser Teil wird nur beachtet wenn der „A“ switch benutzt wird oder der „a“ switch ohne eine ihm folgende Zahl benutzt wird. Zusätzlich darf der „r“ switch nicht gesetzt sein. Für regionale Attribute und regionale Bedingungen wird ein Punkt für jedes Attribut/jede Bedingung angegeben. Dieses Attribut/diese Bedingung wirkt sich dann auf die gesamte Region aus, die diesen Punkt beinhaltet. Stehen zwei Werte hinter der x und y Koordinate, stellt der erste Wert das regionale Attribut (nur wenn der „A“ switch gewählt wurde) dar und der zweite Wert die regionale Bedingung bezüglich des Bereichs, den ein Dreieck umfasst (nur wenn der „a“ switch gewählt wurde).

Ist der „n“ switch angegeben wird das Programm angewiesen, eine .neigh Datei auszugeben. Diese enthält Nachbarschaftsinformationen zu den Dreiecken. Jedes Dreieck besitzt drei Nachbarn, welche Indizes in die korrespondierende .ele Datei sind. Ist der Index eines Nachbarn -1, bedeutet dies, dass kein Nachbar vorhanden ist, da das Dreieck an einer äußeren Kante liegt. In der .ele Datei ist in der ersten Zeile die Anzahl der Dreiecke mit Anzahl der Knoten pro Dreieck aufgeführt. In den folgenden Zeilen werden die Indizes der Dreiecke mit ihren dazugehörenden Knoten aufgeführt. Der Nachbar in einer .neigh Datei verweist also auf den Index eines Dreicks in einer .ele Datei.

Der „e“ switch weist Triangle an, eine „edgelist“ auszugeben. Dies ist ein Array von Kanten-Endpunkten. Eine Kante besitzt zwei Endpunkte, die je durch einen integer-Wert beschrieben werden. Dieser Wert stellt den Index eines Punktes in die korrespondierende .node Datei dar. Die Endpunkte der ersten Kante befinden sich im Array an den Stellen [0] und [1], gefolgt von den restlichen Kanten. Alle Kanten, die durch die Triangulierung entstehen, sind in der „edgelist“ gespeichert.

Der „a“ switch gefolgt von einer Zahl gibt den maximalen Bereich des Dreiecks an. Das heißt, kein Dreieck das generiert wurde überschreitet die Größe dieses Bereichs. Wird zum Beispiel „a3000“ angegeben ist, hat kein Dreieck einen größeren Bereich als 3000. Das wiederum bedeutet, je größer die Zahl wird, desto größer werden die Dreiecke. Um den minimalen Winkel der Dreiecke kontrollieren zu können wurde der „q“ switch eingeführt. Die Zahl die hinter q steht gibt an, dass keine Dreiecke, die einen kleineren Winkel als diese Zahl haben auftreten. Um dies zu gewährleisten werden zusätzliche Punkte eingefügt. Wird keine Zahl hinter „q“ angegeben beträgt der minimale Winkel 20 °. Immer wenn der „q“ und/oder „a“ switch benutzt werden, generiert Triangle eine conforming constrained Delaunay Triangulierung.

2.3 4-8 meshes von Luiz Velho

Wie in der Einleitung bereits erwähnt, erwartet der Kleidersimulator ein 4-8 mesh. Es handelt sich um ein Netz bestehend aus miteinander verbundenen Knotenpunkten, wobei diese abwechselnd die Wertigkeit 4 und 8 haben. Ein Knotenpunkt, von dem vier Verbindungslinien zu benachbarten Knotenpunkten ausgehen, hat die Wertigkeit 4, entsprechend gilt hat ein Knotenpunkt mit 8 ausgehenden Verbindungen die Wertigkeit 8. Ein Beispiel für ein 4-8 mesh zeigt Abbildung 19.

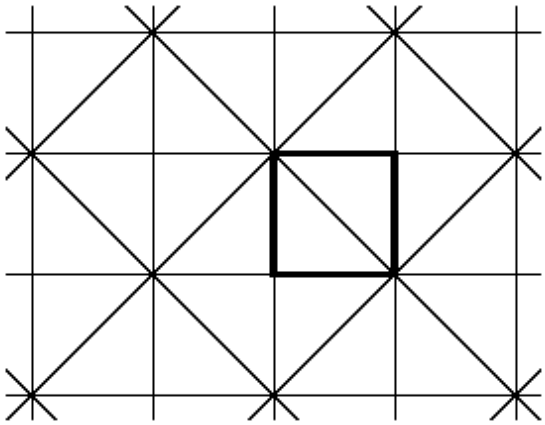

Abbildung 19: 4-8 mesh. Einer der basic blocks ist hervorgehoben, er besteht wie alle anderen aus zwei Dreiecken, besitzt eine innere und vier äußere Kanten.

Die Grundstruktur eines 4-8 meshes ist der sogenannte basic block. Er besteht aus zwei benachbarten Dreiecken, die zusammen ein Viereck ergeben. Die Diagonale des basic block, die beiden Dreiecken gemeinsam ist, ist die einzige innere Kante, alle anderen Kanten sind äußere Kanten des basic blocks. Analog hat jedes enthaltene Dreieck zwei äußere Kanten und eine Kante, die als innere Kante bezeichnet wird. Diese Art der Unterteilung ergibt eine triangulierte Viereckssstruktur. Um Kleidungsstücke bei der Simualtion natürlicher erscheinen zu lassen, kann das Netz weiter verfeinert werden. Die Möglichkeit der Verfeinerung des Netzes hängt von der topologischen Struktur des Netzes ab, nämlich der Tatsache, dass es in basic blocks vorliegt. Ein solches Netz wird auch trianguliertes quadrilaterales Netz genannt, oder kurz tri-quad mesh. Es gibt eine Reihe von Methoden um ein tri-quad mesh zu erzeugen. Im einfachsten Fall ist ein Netz bestehend aus Vierecken gegeben; durch simples Halbieren an einer der Diagonalen in zwei Dreiecke erhält man ein tri-quad mesh. Liegt hingegen ein trianguliertes Netz vor, so wird die Umwandlung etwas aufwendiger. Es wird nach einem möglichst großen Set von basic blocks gesucht und von jedem block jeweils die innere Kante entfernt. Was man erhält ist ein „intermediate mesh“,

das aus Dreiecksflächen und Viereckflächen zusammengesetzt ist. Ein solches Netz ist in Abbildung 20 als „intermediate mesh“ zu sehen.

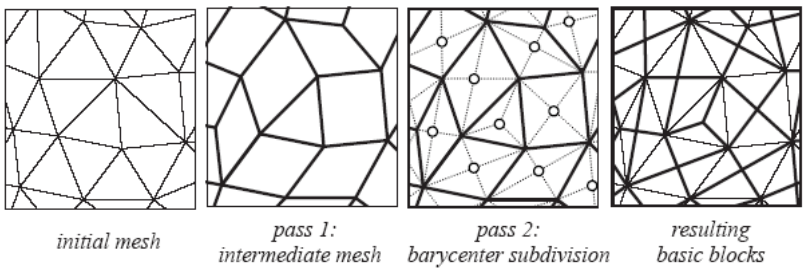

Abbildung 20: Die Schritte vom triangulierten Netz zum 4-8 Netz

Der nächste Schritt ist, Barycenter-Verfeinerung auf dem erhaltenen Netz durchzuführen (siehe Abbildung 20 , pass 2 barycenter subdivision) und die resultierenden basic blocks zu markieren. Bei der Implementation des ersten Schritts, ist die Kantenlänge der inneren Kante das Kriterium zum Suchen eines basic blocks. Man beginnt mit der längsten Kante. Durch diese Herangehensweise erhält man garantiert Dreiecke mit nicht zu spitzen Winkeln und für planare Netze, viereckige basic blocks. Folgender Pseudocode wurde implementiert.

```
find blocks
store interior edges in priority queue Q
while Q  not ∅
do
get e from Q
   mark basic block corresponding to e
   remove from Q edges sharing a face with e
```

Es werden basic blocks gesucht. Dazu werden alle Kanten, die nicht zur äußeren Begrenzung des Netzes gehören in einer Liste gespeichert. Dann wird die Liste sortiert und solange sich noch Kanten darin befinden, wird die längste Kante (die sich an erster Position befindet) entfernt. Der dazugehörige basic block wird markiert. Um Überschneidungen der basic blocks zu verhindern, werden alle Kanten, die äußere Kanten des gerade gebildeten basic blocks sind und sich noch in der Liste befinden gelöscht.

Im Allgemeinen ist es nicht möglich das ganze Netz mit basic blocks abzudecken. Es werden immer einige isolierte Dreiecke übrig bleiben. Das „intermediate mesh“ wird, nachdem die inneren Kanten entfernt worden sind, zwei Arten von Flächen enthalten, Dreiecke und Vierecke. Im zweiten Schritt werden Barycenter in das Netz integriert, jedes Viereck wird in vier Dreiecke und jedes

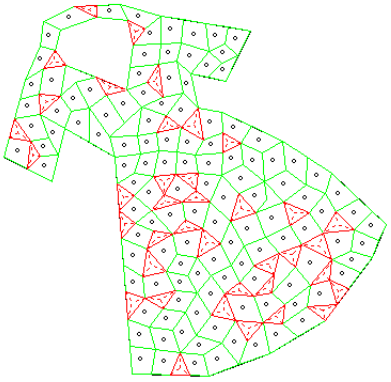

Abbildung 21: Tri-quad mesh. Dreiecke sind rot markiert, Vierecke sind grün markiert und enthalten jeweils ein Barycenter. Es lässt sich nicht vermeiden, dass einzelne Dreiecke übrig bleiben.

Dreieck in drei Dreiecke unterteilt. Es ist leicht zu erkennen, dass das Ergebnis ein tri-quad mesh ist. Genau ein Knoten jedes Dreiecks nach der Barycenter-Verfeinerung (siehe Abbildung 20 pass 2) ist das Barycenter einer Fläche des Netzes vor der Barycenter-Verfeinerung. Die innere Kante jedes Dreiecks ist nach der Verfeinerung eine äußere Kante eines basic blocks. Luiz Velho schlägt vor, die inneren Kanten im ersten Schritt nicht zu entfernen, statt dessen kann man sie im Netz belassen und nicht die Barycenter mit den vier Eckpunkten verbinden, sondern die basic blocks, die man erhält, halbieren.

2.3.1 Refinement

Praktisch alle bisher bekannten Verfeinerungsverfahren (Refinement) arbeiten mit einer Unterteilung der Fläche in dreieckige oder in viereckige Kacheln. Diese Kachelungen können sehr einfach verfeinert werden. Die Idee von Luiz Velho ist, die Kacheln nach einem 4-8 System anzulegen. Dieses System bietet einige Vorteile:

- Die grundlegende Operation zur Verfeinerung ist Zweiteilung (bisection). 4-8 Netze und auch die generellere Form, die 4-k Netze werden durch Kantenzweiteilung verfeinert. Im Gegensatz zu anderen Refinementverfahren bleibt ein konformes 4-8 Netz nach einem einzigen bisection-Schritt immer noch konform. Das heißt, es können keine Brüche auftreten. Diese Eigenschaft vereinfacht das adaptive Verfeinern des Netzes. Ist ein Netz nicht uniform verfeinert, ist trotzdem garantiert, dass es konform ist.

- Graduelles Refinement. Bei den meisten Refinementverfahren erhöht ein einziger Refinementschritt die Anzahl der Flächen oder Knoten um den Faktor vier. Das Refinementverfahren von Luiz Velho erhöht die Anzahl der Flächen nur um den Faktor zwei.

- Die beim Refinement verwendeten Masken führen zu glatteren Oberflächen als z.B. das von Catmull-Clark verwendete Refinement.

2.3.2 Gleichmäßiges Refinement und Kacheln

Eine Refinementregel ist ein Algorithmus, der aus einem gegebenen Netz ein verfeinertes Netz herstellt. Diese Regel bezieht sich nur auf die reine Topologie des Netzes. Es nutzt nur die Informationen darüber, wie die Knoten und Flächen miteinander verbunden sind und kennt nicht deren geometrische Daten wie z.B. die Knotenpositionen.

Typische Refinementmethoden sind eng verwandt mit regulären Kachelungen, wie die Tesselierung einer Ebene, die aus gleichmäßigen n-Ecken besteht. Wendet man eine Refinementregel auf die zugehörige Kachelung an, belässt dieser Schritt das Netz invariant, d.h. es ist isomorph zum Originalnetz.

Es gibt nur drei Arten von gleichmäßigen Kachelungen, die Form der Kachel ist entweder ein Quadrat, ein gleichseitiges Dreieck oder ein gleichseitiges Hexagon. Die meisten Refinementalgorithmen arbeiten mit Dreiecken oder Vierecken. Eine oft verwendete Refinementregel, die für quadratische Kacheln invariant ist, heißt „face split“. Sie kann auf beliebige Netze angewendet werden. An jeder Kante und in jeder Fläche wird ein Knoten eingefügt; die neu eingefügten Knoten werden durch Kanten verbunden. Auf diese Weise wird jede Fläche in n Flächen aufgeteilt, wobei n die Anzahl der Ecken dieser Fläche ist. Nach ein paar Refinementschritten hat das verfeinerte Netz lokal dieselbe Struktur wie das gesamte Netz.

Das Refinement eines 4-8 Netzes durch Halbierung (bisection) ergibt wieder ein 4-8 Netz. Das bisection Refinement der Kanten hat die Eigenschaft, dass zwei Schritte dasselbe Ergebnis erzielen wie ein face split eines Quads.

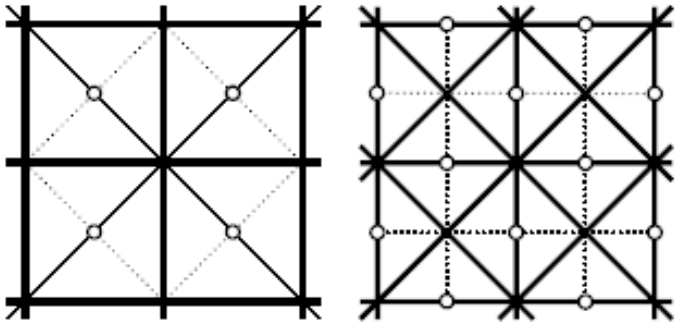

Abbildung 22: Zwei bisection-schritte sind äquivalent zu einem face split. Neu eingefügte Knoten werden als Kreis dargestellt, neue Kanten werden als gestrichelte Linien dargestellt.

In einem tri-quad Netz sind immer zwei Knoten auf der Diagonalen eines Blocks, während die anderen beiden Knoten nicht auf der Diagonalen liegen. Einem Knoten wird der Typ 1 zugeordnet, wenn er nicht auf der Diagonalen liegt, andernfalls wird ihm der Typ 2 zugeordnet. Alle Knoten, die in einem einzigen Schritt des Refinement eingefügt werden, sind vom Typ 1 und haben die Wertigkeit 4. Ein einziger Refinementschritt wandelt alle Knoten vom Typ 1 in Knoten vom Typ 2 um. Die Wertigkeit der Knoten vom Typ 2 wird vom Refinement nicht verändert.

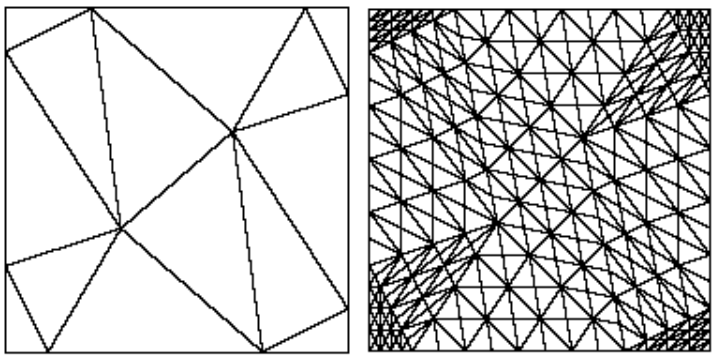

Abbildung 23: Refinement durch bisection (vier Schritte)

2.3.3 Adaptives Verfeinern

Refinementverfahren wie face split oder vertex split können nicht adaptiv auf ein Netz angewendet werden, ohne Inkonsistenzen in der Topologie zu verur-

sachen, d.h. es würden Brüche im Netz entstehen. Beim face split werden alle Kanten einer Fläche halbiert. Um das Netz konsistent zu halten, müssen alle anliegenden Flächen auch unterteilt werden. Das bedeutet, dass alle Flächen des gesamten Netzes unterteilt werden müssten. Im Gegensatz dazu generiert bisection refinement eine hierarchische Netzstruktur, die eine variable Auflösung unterstützt. Für jede einzelne Kante kann ausgewertet werden, ob sie unterteilt werden soll oder nicht. Es gibt zwei mögliche Fälle, entweder ist es eine innere oder eine äußere Kante. Ist es eine innere Kante, dann müssen keine anderen Blocks verfeinert werden um die Triangulierung konform zu halten. Handelt es sich um eine äußere Kante, muss nur der Nachbarblock verfeinert werden.

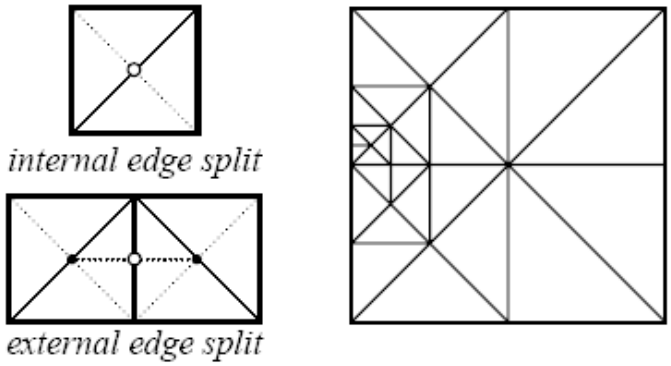

Abbildung 24: Adaptives refinement eines 4-8 Netzes. Links: die zwei möglichen Fälle der Unterteilung, die Unterteilung einer inneren Kante zieht keine weiteren Unterteilungen nach sich um die Konformität des Netzes aufrecht zu erhalten. Eine äußere Kante erfordert die Unterteilung des anliegenden Blocks. Rechts: ein Beispiel für ein adaptiv unterteiltes 4-8 Netz.

2.3.4 Basic Blocks finden, ein Beispiel

Die Kanten liegen im Kantenvektor der Länge nach absteigend sortiert vor. Das heißt, die erste Kante ist die längste. Zu dieser Kante werden die beiden anliegenden Dreiecke gesucht.

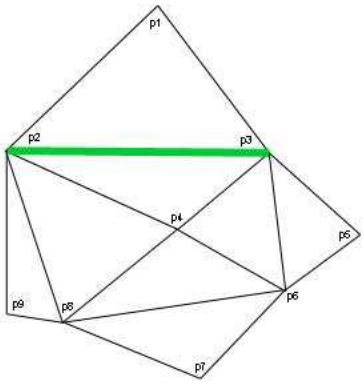

Abbildung 25: In allen Dreiecken wird die Kante, die als Start- und Endpunkt p2 und p3 hat gesucht, die Kante kann in genau zwei Dreiecken gefunden werden.

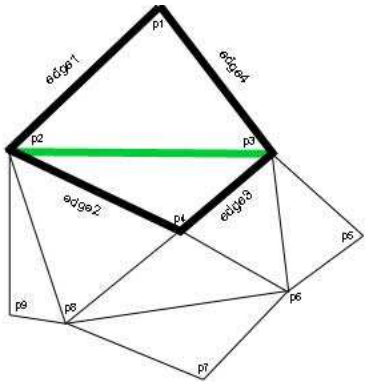

Abbildung 26: Sind die beiden Dreiecke gefunden, werden alle äußeren Kanten und zusätzlich die Diagonalen abgespeichert.

Die Kanten, die im Block abgespeichert sind, werden nicht mehr zur Suche von blocks herangezogen. Damit es keine Überschneidungen bei der Bildung der Blocks gibt, müssen die Nachbarn der beiden verarbeiteten Dreiecke ausfindig gemacht werden. Die Kanten, die an der Bildung eines Basic blocks beteiligt sind, werden gelöscht.

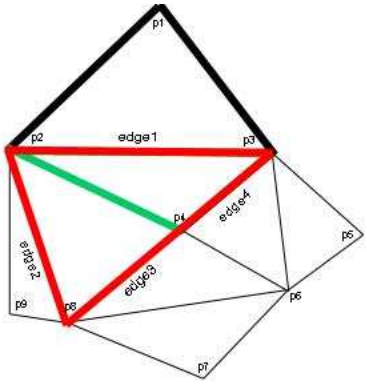

Abbildung 27: Der rote Block darf so nicht gebildet werden, da er sich mit dem schwarzen überschneidet.

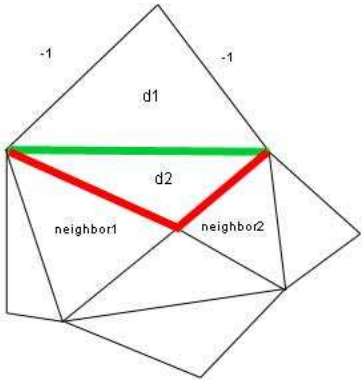

Abbildung 28: Die beiden roten Kanten werden aus dem Kantenvektor gelöscht und nicht mehr zur Suche nach einem Basic block herangzogen.

Dreieck d1 und d2 werden auf Nachbarn überprüft. Dreieck d1 hat außer d2 keine Nachbarn, deshalb steht in seiner Nachbarliste nur eine -1. Das Dreieck d2 hat zwei Nachbarn, es wird geprüft, welche der Kanten die neighbor1 und neighbor2 bilden schon in einem Block gespeichert sind. Diese werden gelöscht. Je nachdem wo sich die längsten Kanten befinden, ist es unvermeidbar, dass nachdem alle Basic blocks gefunden sind, einige einzelne Dreiecke übrig bleiben.

3 Implementation

In den ersten Abschnitten des folgenden Kapitels sind die wichtigsten Funktionen erläutert, die notwendig sind, um Polygone zu zeichnen, sie auszuwählen, Punkte eines Polygons zu verschieben, sowie zusätzliche Punkte in die Linien eines Polygons einzufügen. In den folgenden Abschnitten wird darauf eingegangen, wie Triangle in den Virtual Tailor eingebunden und der Algorithmus von Luiz Velho umgesetzt wurde.

3.1 Zeichnen eines Polygons

Um ein Polygon zu zeichnen werden folgende Methoden benötigt:

```
void GetCoordinates(LPARAM lParam)
void boundary::InsertPoint(POINT point)
void boundary::DrawBoundary(HWND hWnd, LPARAM lParam, HDC hDC)
```

Wählt der Benutzer den „DrawButton“ aus, wird das Flag `drawButtonPressed = true` gesetzt und bei jedem Betätigen der linken Maustaste die Funktion `GetCoordinates(LPARAM lParam)` aufgerufen:

```
void GetCoordinates(LPARAM lParam)
{
point.x = LOWORD(lParam); //store x-mouseposition
point.y = HIWORD(lParam); //store y-mouseposition
bnd.InsertPoint(point);
}
```

Diese Funktion speichert den x- und den y-Wert der Mausposition und ruft die Funktion
`InsertPoint(POINT point)` auf, die diesen Wert an das Ende des Vektors speichert:

```
void boundary::InsertPoint(POINT point)
{
vect.push_back(point); //store the point in the vector
}
```

Um die eben im Vektor gespeicherten Werte zu zeichnen, wird folgende Methode aufgerufen:

```
void boundary::DrawBoundary(HWND hWnd, LPARAM lParam, HDC hDC)
{
if (vect.size( )>0)

{
for(int i = 0; i< vect.size( )-1; i++)
```

```
{
pen = CreatePen( PS_SOLID,1,RGB(0,0,0));
SelectObject(hDC, pen);
MoveToEx(hDC, vect[i].x, vect[i].y, NULL); //move to the previous //point
LineTo(hDC, vect[i+1].x, vect[i+1].y);  //draw line to the //current point
DeleteObject(pen);
}
}
}
```

Beim ersten Mausklick wird noch nichts gezeichnet, da `vect.size() -1 = 0` ist. Der Wert wird jedoch im Vektor gespeichert. Beim zweiten Mausklick ist `vect.size() -1 = 1` und es wird von den Koordinaten, die an Position „null“ stehen zu den Koordinaten, die an Position „eins“ stehen, eine Linie gezeichnet. Beim dritten Mausklick wird dann von Position „eins“ nach Position „zwei“ gezeichnet usw.

3.2 Auswählen eines Polygons

Um in ein Polygon neue Punkte einzufügen oder Punkte zu verschieben, muss das Polygon ausgewählt werden, das verändert werden soll. Das ausgewählte Polygon wird blau gezeichnet. Hier werden hauptsächlich die Funktionen

```
int boundary::LineCrossLine(LPARAM lParam, HINSTANCE hInstance,
POINT point, POINT point2)
```

und

```
int boundaryVector::LineCrossLineVector(LPARAM lParam,
HINSTANCE hInstance, POINT point, POINT point2)
```

benötigt. Zum Auswählen muss der „Pfeil-Button“ gedrückt werden. Der Flag `chooseButtonPressed` wird auf `true` gesetzt. Benutzt man jetzt die linke Maustaste, wird die Position des Mauszeigers und die linke obere Ecke des Fensters (Koordinate 0,0) gespeichert und der Funktion

```
LineCrossLineVector(LPARAM lParam, HINSTANCE hInstance, POINT point, POINT point2)
```

übergeben.

```
int boundaryVector::LineCrossLineVector(LPARAM lParam,
HINSTANCE hInstance, POINT point, POINT point2)
{
int numberOfIntersections;
boundaryNr = -1;
for(int i = 0; i<= vectbound.size()-1; i++)
{
numberOfIntersections = vectbound[i].LineCrossLine(lParam, hInstance,
point, point2);
```

```
if(numberOfIntersections %  2 ==1)
{
boundaryNr = i;
}
}
return boundaryNr;
}
```

Diese Methode benutzt die Funktion
`LineCrossLine(LPARAM lParam, HINSTANCE hInstance, POINT point, POINT point2)`, welche berechnet, wie viele Schnittpunkte eine Linie mit allen Linien eines Polygons hat. Auch dieser Methode werden die Mausposition und die Position (0,0) übergeben. `LineCrossLineVector` berechnet also durch die Funktion `LineCrossLine`, wie viele Schnittpunkte mit den Linien eines Polygons und einer Linie, die von der Position des Mauszeigers zu der Position (0,0) führt, vorliegen. Bei einer geraden Anzahl (oder keinem Schnittpunkt) von Schnittpunkten befindet sich der Mauszeiger außerhalb des Polygons, bei einer ungeraden Zahl innerhalb.

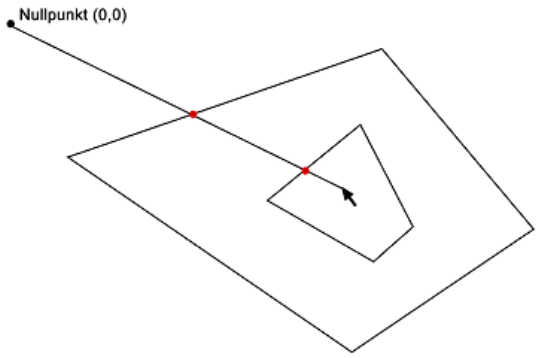

Abbildung 29: Zwei Schnittpunkte vom Mauszeiger zum Punkt (0,0)

Das Bild zeigt zwei Polygone, wobei das Äußere die Umrandung des Stoffstückes und das Innere ein Loch darstellt. Soll das Loch ausgewählt werden, muss man mit der Maus hineinklicken. Dann wird zuerst die Anzahl der Schnittpunkte mit dem äußeren Polygon berechnet. Im obigen Beispiel liegt ein Schnittpunkt vor. Dieser bleibt unbeachtet, denn es liegt auch noch ein Schnittpunkt mit dem Loch vor. Von dem Loch wird die Stelle, an der es gespeichert ist, zurückgegeben. Man kann auf diese Stelle zurückgreifen und das Polygon, das an dieser Stelle steht blau zeichnen. `LineCrossLineVector` geht davon aus, dass das erste gezeichnete Polygon das Äußere ist und die nachfolgend gezeichneten Polygone die Löcher des Kleidungsstückes sind. Dabei darf kein Loch in einem Loch liegen.
Mit der Methode `LineCrossLine` soll, wie beschrieben, überprüft werden, wie

viele Schnittpunkte eine Linie mit allen Linien eines Polygons hat. Dazu wird berechnet, wie viele Schnittpunkte der Vektor vom Mauszeiger zum Nullpunkt (linke obere Ecke des Fensters) mit den gezeichneten Linien hat. Ist die Anzahl der Schnittpunkte gerade (oder null) liegt der Punkt außerhalb, ist die Anzahl ungerade liegt der Punkt innerhalb des Polygons.

3.3 Berechnung, ob zwei Linien einen Schnittpunkt haben

Eine Linie wird folgendermaßen dargestellt:

$$g : \vec{x} = \vec{p} + s * \vec{u}$$

Eine Linie besteht aus einem Stützvektor und einem Richtungsvektor. Der Stützvektor zeigt vom Nullpunkt zum Anfangspunkt der Linie und der Richtungsvektor gibt die Richtung der Linie an. Mit s kann man die gesamte Linie abfahren. Hat s zum Beispiel den Wert 0,75 wandert man von dem Stützvektor $\vec{p}$ 0,75 mal den Richtungsvektor entlang der Linie. Für eine Linie müssen s und t zwischen 0 und 1 liegen, ansonsten befindet man sich vor oder hinter der Linie. Um den Schnittpunkt von zwei Linien

$$l_1 : \vec{x} = \vec{p} + s * \vec{u}$$

$$l_2 : \vec{x} = \vec{q} + t * \vec{v}$$

zu berechnen, muss je ein Wert für t und s gefunden werden, so dass wenn man s in die erste und t in die zweite Gleichung einsetzt, der gleiche Wert, nämlich der Schnittpunkt der beiden Linien rauskommt. Demnach müssen als erstes die beiden Gleichungen für x gleichgesetzt werden.

$$\vec{p} + s * \vec{u} = \vec{q} + t * \vec{v} \Rightarrow s * \vec{u} - t * \vec{v} = \vec{q} - \vec{p}$$

Teilt man diese Vektorgleichung in zwei Koordinatengleichungen auf, erhält man folgendes Gleichungssystem:

$$s * x_u - t * x_v = x_q - q_p$$

$$s * y_u - t * y_v = y_q - y_p$$

Man kann die Determinantenrechnung nutzen, um s und t auszurechnen:

$$s = D_s / D$$

$$t = D_t / D$$

Da die Determinante D in den Termen im Nenner steht, darf sie nicht Null werden, ansonsten würde man durch Null dividieren. Eine Determinante ist dann

Null, wenn die beiden Vektoren (hier u und v) linear abhängig sind. Das wiederum bedeutet, dass u ein Vielfaches von v ist, oder umgekehrt. Da u der Richtungsvektor von der ersten Linie ist und v der der zweiten Linie, bedeutet das, dass die Linien parallel sind und sie keinen Schnittpunkt haben. Ansonsten haben die Linien einen Schnittpunkt. Die Implementierung der Funktion `LineCrossLine` sieht folgendermaßen aus:

```
int boundary::LineCrossLine(LPARAM lParam, HINSTANCE hInstance,
POINT point, POINT point2)
{
numberOfIntersections = 0;
```

Zuerst werden der Vektor der ersten Linie des Polygons berechnet, daraufhin der Vektor zwischen Mausposition und Nullpunkt.

```
for(int i = 0; i < vect.size()-1; i++)
{
float startLineX = vect[i].x;
float startLineY = vect[i].y;
float endLineX = vect[i+1].x;
float endLineY = vect[i+1].y;

float mouseVectorX = point2.x - point.x;
float mouseVectorY = point2.y - point.y;

float polygonVectorX = endLineX - startLineX;
float polygonVectorY = endLineY - startLineY;
```

Als nächstes berechnet die Funktion die Determinante der beiden Vektoren

```
float determinant = (mouseVectorY * polygonVectorX) -
(mouseVectorX *  polygonVectorY);
```

Ist die Determinate ungleich null, sind die Linien nicht parallel und haben einen Schnittpunkt.

```
if(determinant != 0.0)
{
```

s und t werden berechnet, wobei die beiden Werte zwischen 0 und 1 liegen müssen.

```
float determinantS = ((startLineY - point.y) * polygonVectorX) -
((startLineX - point.x) * polygonVectorY);

float s = determinantS/determinant;

if((s > 0.0) && (s < 1.0))
```

```
{
float determinantT = ((startLineY - point.y) * mouseVectorX) -
((startLineX - point.x) * mouseVectorY);

float t = determinantT/determinant;

if((t > 0.0) && (t < 1.0))
{
numberOfIntersections ++;
}
}
}
}
return numberOfIntersections;
}
```

Die Anzahl der Schnittpunkte wird zurückgegeben. Möchte man wissen, ob der Punkt (z.B. die Stelle, an der sich die Maus befindet), inner- oder außerhalb eines Polygons liegt, muss nur noch die Anzahl der Schnittpunkte modulo zwei gerechnet werden. Kommt dabei ein Rest von null heraus, war die Anzahl der Schnittpunkte gerade und der Punkt liegt außerhalb. Bei einem Rest von eins, liegt der Punkt innerhalb des Polygons.

3.4 Hinzufügen eines Punktes

Um einen zusätzlichen Punkt in eine beliebige Linie des Polygons oder eines Loches hinzuzufügen, muss man zuerst das Polygon auswählen, in welches der Punkt eingefügt werden soll. Dann muss der Button „Add Point“ ausgewählt werden. Bewegt man die Maus wird der Funktion `CheckPointOnLineVector` die Stelle im Vektor, an der sich das ausgewählte Polygon befindet, übergeben. Nun wird die Funktion `CheckPointOnLine` für das gewählte Polygon aufgerufen. Mit Hilfe dieser Methode soll überprüft werden, ob sich der Mauszeiger in der Nähe einer gezeichneten Linie befindet. Ist dies der Fall, ändert sich der Mauszeiger und signalisiert so dem Benutzer, dass er einen Punkt in eine Linie einfügen kann.

```
int boundary::CheckPointOnLine(LPARAM lParam, HINSTANCE hInstance)
{

int PosX = LOWORD(lParam); //store x-mouseposition
int PosY = HIWORD(lParam); //store y-mouseposition
AktPoint.x = PosX;
AktPoint.y = PosY;

for(int i = 0; i < vect.size()-1; i++)
// loop over all lines (2 points that
```

```
//are neighbours define a line)
{
int StartLineX = vect[i].x;
int StartLineY = vect[i].y;
int EndLineX = vect[i+1].x;
int EndLineY = vect[i+1].y;
int vectorX;
int vectorY;
int vector2X;
int vector2Y;
double scalar;
double lengthVectorX;
double lengthVectorY;
double angle;

vectorX = EndLineX - StartLineX;
//compute vector from the start of
//the line to the end of the line

vectorY = EndLineY - StartLineY;
vector2X = PosX - StartLineX;
//compute vector from the start of
//the line to the mouseposition

vector2Y = PosY - StartLineY;

//cos a = (x * y)/(|x| * |y|) -> a = arccos ((x * y)/(|x| * |y|))
scalar = (vectorX * vector2X) + (vectorY * vector2Y);

//length of a vector: |x| = squareroot(x * x)
lengthVectorX = sqrt((double) ((vectorX * vectorX) + (vectorY * vectorY)));
lengthVectorY = sqrt((double) ((vector2X * vector2X) + (vector2Y * vector2Y)));

//angle in degrees
angle = (acos(scalar/(lengthVectorX * lengthVectorY))) *180/PI;

if (angle < 2.0)
{
if(lengthVectorY <= lengthVectorX)
{
HCURSOR hCurs2; // cursor handles

hCurs2 = LoadCursor(hInstance, MAKEINTRESOURCE(116));
SetCursor(hCurs2); DestroyCursor(hCurs2);
insertPos = i;
}
```

```
}
}
return insertPos; //give the position back
}
```

Die aktuelle Position des Mauszeigers wird gespeichert. Eine Linie besteht aus einem Anfangs- und einem Endpunkt (immer zwei nachfolgende Punkte in dem Vektor). Der Vektor vom Anfangspunkt zu dem dazugehörigen Endpunkt wird berechnet, sowie der Vektor vom Anfangspunkt zu den Koordinaten des Mauszeigers.

```
vectorX = EndLineX - StartLineX;
//compute vector from the start of the line to the
//end of the line

vectorY = EndLineY - StartLineY;

vector2X = PosX - StartLineX;
//compute vector from the start of the line to the
//mouseposition

vector2Y = PosY - StartLineY;
```

Da der Benutzer erkennen soll, wann er sich mit dem Mauszeiger nahe einer Linie befindet, in die er einen neuen Punkt einfügen kann, soll der Mauszeiger sich verändern, wenn er in der Nähe einer Linie ist. Die Funktion berechnet zu diesem Zweck den Winkel zwischen den eben berechneten Vektoren. Beträgt der Winkel null Grad, liegt die Koordinate des Mauszeigers auf der Linie. Da aber angezeigt werden soll, ob sich der Mauszeiger in der Nähe einer Linie befindet, muss sich der Winkel unter einem Schwellenwert befinden. Es wird geprüft, ob der Winkel unter zwei Grad liegt. Zuerst wird das Skalarprodukt und die Länge der Vektoren berechnet. Mithilfe dieser beiden Werte kann der Winkel berechnet und vom Bogenmaß in Grad umgerechnet werden.

```
angle = (acos(scalar/(lengthVectorX * lengthVectorY))) *180/PI;
//angle in degrees
```

Liegt der Winkel unter zwei Grad, wird zusätzlich abgefragt, ob die Länge des Vektors des Polygons kleiner ist als der andere Vekor, um zu überprüfen, ob der Mauszeiger sich nicht vor oder hinter der Linie des Polygons befindet.

```
if(lengthVectorY <= lengthVectorX)
```

Liegt der Winkel unter dem Schwellenwert, verändert sich der Mauszeiger und der Benutzer weiß, dass er jetzt einen Punkt in die Linie einfügen kann.

```
if (angle < 2.0)
{
```

```
if(lengthVectorY <= lengthVectorX)
{
HCURSOR hCurs2; // cursor handles

hCurs2 = LoadCursor(hInstance, MAKEINTRESOURCE(116));
SetCursor(hCurs2); // change cursor when over a point
DestroyCursor(hCurs2);
insertPos = i;
}
}
```

Die Funktion gibt den Anfangspunkt der Linie des Polygons zurück, damit bekannt ist, nach welcher Stelle im Polygon ein Punkt eingefügt werden muss. Um den neuen Punkt in den Vektor einzufügen, wird die Funktion

```
void boundary::InsertPointInLine(int index)
{
it = vect.begin();
vect.insert(it + (index +1), AktPoint); //inserts the actual point
}
```

aufgerufen. Dieser Funktion wird der von der Funktion `CheckPointOnLine` zurückgegebene Wert (Position nach der der Punkt eingefügt werden muss) übergeben. Ein Iterator wird angelegt, der auf den Anfang des Array zeigt, in dem alle gespeicherten Punkte abgelegt sind. Die Methode `vect.insert` fügt den aktuellen Punkt (also den Punkt, bzw. die Koordinaten dieses Punktes, auf den der Mauszeiger zeigt) in den Vektor, hinter den übergebenen Punkt, ein.

3.5 Verschieben eines Punktes

Um Punkte zu verschieben, muss vorher das Polygon ausgewählt werden, dessen Punkte man verschieben möchte. Dann muss der Verschiebenbutton ausgewählt werden und `editButtonPressed` wird auf `true` gesetzt. Die Punkte des Polygons, die man verschieben kann, werden angezeigt.

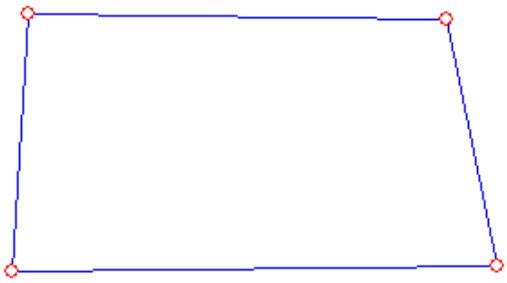

Abbildung 30: Polygon mit markierten Eckpunkten

Bewegt man die Maus wird die Funktion `DetectMatchingPoint` aufgerufen. Hier wird ebenfalls die aktuelle Position des Mauszeigers gespeichert.

```
editPoint.x = LOWORD(lParam); //store x-mouseposition
editPoint.y= HIWORD(lParam); //store y-mouseposition
```

Als nächstes wird die Funktion `FindPointVector(editPoint, actualBoundary)` aufgerufen und der Rückgabewert dieser Funktion in `foundElementNumber` gespeichert.

```
foundElementNr = bndvct.FindPointVector(editPoint, actualBoundary);
```

`FindPointVector(editPoint, actualBoundary)` ruft die Funktion `FindPoint(POINT point)` für das ausgewählte Polygon auf. Ihr wird die aktuelle Mausposition übergeben. Es wird überprüft, ob diese Position in der Nähe eines verschiebbaren Punktes liegt. `point.x` und `point.y` sind die Koordinaten der Mausposition, `vectorX` und `vectorY` die Koordinaten des gespeicherten Punktes eines Polygons.

```
if((point.x > vectorX -10 && point.x < vectorX +10)&&
(point.y > vectorY -10 && point.y < vectorY +10))
```

Es wird überprüft, ob sich der Mauszeiger in 10 Pixel Entfernung in x- und y-Richtung eines Punktes des Polygons befindet. Ist dies der Fall, ändert sich der Mauszeiger und der Benutzer weiß, dass er diesen Punkt verschieben kann. Die Funktion gibt die Stelle des Vektors zurück, an der der Punkt gespeichert ist. Wird kein Punkt gefunden, gibt die Funktion den Wert -1 zurück.

```
int boundary::FindPoint(POINT point)
{
//go over all points stored in the boundary vector
for(int elementNr = 0; elementNr < vect.size(); elementNr++)
{
```

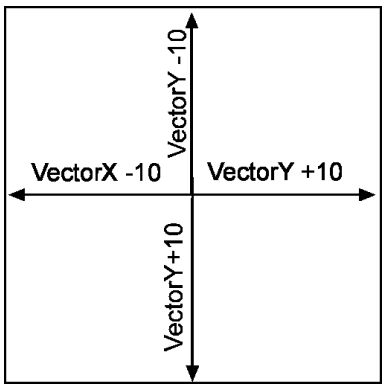

Abbildung 31: Der Mauszeiger ändert sich, wenn er in den „Bereich“ des Punktes kommt

```
int vectorX = vect[elementNr].x;
int vectorY = vect[elementNr].y;
//compare the current mouse-position with all elements in the vector
if((point.x > vectorX -10 && point.x < vectorX +10)&&
(point.y > vectorY -10 && point.y < vectorY +10))
{
HCURSOR hCurs1; // cursor handles

hCurs1 = LoadCursor(NULL, IDC_SIZEALL);
SetCursor(hCurs1); // change cursor when over a point
DestroyCursor(hCurs1);

return elementNr; //return the current element number
//of the matched point
}
}
return -1;
}
```

Der Parameter `foundElementNr` der Funktion `DetectMatchingPoint` bekommt demnach entweder -1 oder die Speicherstelle eines Punktes übergeben. Nachfolgend wird überprüft, welcher der beiden Werte übergeben wurde.

```
if (foundElementNr < 0)
{
pointFound = false;
}
else
{
```

```
pointFound = true;
}
```

Wurde eine -1 übergeben, wird `pointFound = false` gesetzt, andernfalls wird `pointFound` auf `true` gesetzt. Ist `pointFound = true` kann man den Punkt anfassen und an eine neue Position schieben. Um den alten Punkt im Vektor durch den neuen zu ersetzen, wird die Methode `OverwritePoint(LPARAM lParam, int foundElementNr)` aufgerufen. Diese bekommt die Stelle im Vektor übergeben, an der der alte Punkt gespeichert wurde. An dieser Stelle muss der neue Punkt, also der Punkt an dem sich die Maus befindet, gespeichert werden. Es muss noch beachtet werden, dass zur Vereinfachung beim Zeichnen, der letzte und der erste Punkt eines Polygons gleich sind. Wurde also der erste Punkt, der an der Stelle 0 im Vektor steht, ausgewählt, muss die erste und die letzte Stelle mit dem neuen Punkt überschrieben werden.

```
void boundary::OverwritePoint(LPARAM lParam, int foundElementNr)
{

if(foundElementNr == 0)
{
vect[foundElementNr].x = LOWORD(lParam);
//overwrite the found element with the new position
vect[foundElementNr].y = HIWORD(lParam);

vect[vect.size()-1].x = LOWORD(lParam);
//last and first element are
//the same, you also have to
//overwrite the last element
//when moving the first point
vect[vect.size()-1].y = HIWORD(lParam);
}
else
{

vect[foundElementNr].x = LOWORD(lParam);
//overwrite the found element with the new position
vect[foundElementNr].y = HIWORD(lParam);
}
}
```

3.6 Einbindung von Triangle

Das Programm Triangle wird nicht, wie im Kapitel Triangle vorgestellt, von der Kommandozeile aus aufgerufen. Statt dessen wird die Funktion `triangulate(triswitches, in, out, vorout)` im Programm aufgerufen. Der

erste zu übergebene Parameter ist eine Zeichenkette, die die switches beinhaltet, die man benutzen möchte, während `in`, `out`, und `vorout` Beschreibungen des Input und Output sind.

Für die Triangulierung muss `in.pointlist`, `in.segmentlist` und `in.holelist` der Struktur `triangulateio *in` korrekt gefüllt werden. `in.pointlist` stellt ein Array dar, in dem an der ersten Stelle der x-Wert des ersten Punktes des ersten Polygons gespeichert ist und an der zweiten Stelle dazugehörige y-Wert, usw.
`in.pointlist` muss alle Punkte enthalten, die vom Benutzer beim Zeichnen der Polygone, durch Klicken mit der linken Maustaste, gesetzt wurden.

Die `in.segmentlist` erwartet die Endpunkte aller Segmente. Ein Segment ist die Kante, die sich zwischen zwei, vom Benutzer gesetzten, Punkten befindet. Durch diese beiden Punkte (Endpunkte) wird ein Segment definiert. Die Endpunkte des ersten Segments befinden sich an den Stellen null und eins, gefolgt von den restlichen Segmenten. Ein Endpunkt wird durch eine Variable vom Typ integer dargestellt.

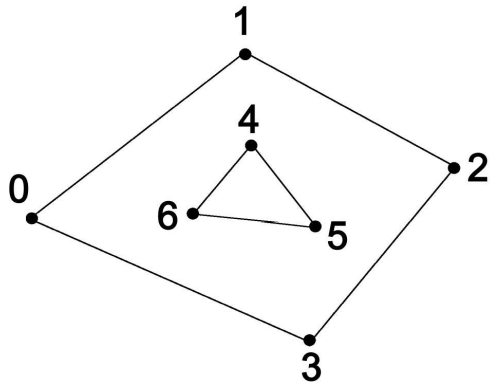

Abbildung 32: Beispiel: Zeichnet man zwei Polygone, zum Beispiel eine Umrandung eines Stoffstückes mit einem Loch, erwartet die Segmentliste die Segmente beider Polygone.

Im Array müssen die Werte wie folgt abgespeichert werden:

0	1	1	2	2	3	3	0	4	5	5	6	6	4

Abbildung 33: Werte im Array

Dies wird durch folgende Schleife realisiert, die gleichzeitig die Punktliste füllt:

```
int pt = 0;
for(int b = 0; b<bndvct.CheckSize(); b++)
{
int base = pt; // Index of first point in this boundary
int npts = bndvct.NrOfPointsInSpecialBoundary(b);

for(int i=0; i<npts; i++)
{
in.pointlist[2*pt] = bndvct.GiveXFromBoundaryVector(b, i);
in.pointlist[2*pt+1] = bndvct.GiveYFromBoundaryVector(b, i);

in.segmentlist[2*pt] = base+i;
in.segmentlist[2*pt+1] = base+(i+1)%npts;
pt++;
}
}
```

Die Funktion `NrOfPointsInSpecialBoundary` errechnet die Anzahl der Punkte in einem Polygon. Zu diesem Zweck benutzt sie die Funktion `NrOfPointsInBoundary`,

```
int boundary::NrOfPointsInBoundary()
{
return vect.size();
}
```

welche die Größe des Vektors wiedergibt, in dem die Punkte des Polygons gespeichert sind. Da der letzte Punkt im Vektor gleich dem Ersten ist und nicht als zusätzlicher Punkt gezählt werden soll, wird in der Funktion `NrOfPointsInSpecialBoundary` 1 von der Größe des Vektor abgezogen.

```
int boundaryVector::NrOfPointsInSpecialBoundary(int b)
{
return vectbound[b].NrOfPointsInBoundary()-1;
}
```

Die Funktionen `GiveXFromBoundaryVector` und `GiveYFromBoundaryVector` geben die x- und y-Werte der Punkte der einzelnen Polygone zurück. In `in.holelist` sind die Daten gespeichert, die zur Generierung von Löchern dienen. Ein Loch wird durch einen Punkt innerhalb des Loches definiert. Um `in.holelist` richtig zu füllen, muss demnach automatisch ein Punkt in dem Polygon erzeugt werden, das ein Loch darstellt. Da man davon ausgeht, dass das erste gezeichnete Polygon die Umrandung des Stoffstückes und alle weiteren gezeichneten Polygone Löcher sind, fängt die Schleife erst bei eins an zu zählen. Zu jedem Loch wird ein Punkt erzeugt, der innerhalb eines Loches liegt.

```
int start=0;
for(b=1; b<bndvct.CheckSize(); b++)
{
POINT k;

k = bndvct.GeneratePointInTriangleVector(b, lParam, hInstance);

in.holelist[start] = k.x; //generatedPointX
in.holelist[start + 1] = k.y; //generatedPointY
start = start +2;
}
```

Die Generierung dieses Punktes übernimmt die Funktion `GeneratePointInTriangleVector(b, lParam, hInstance)`. Diese Funkion ruft wiederum die Funktion `GeneratePointInTriangle` auf.

```
POINT boundary::GeneratePointInTriangle(LPARAM lParam, HINSTANCE hInstance)
{
POINT generatedPoint;

while (true)
{
for(int i=0; i<vect.size()-3; i++)
{
//calculates barycenter of the triangle
int x1 = vect[i].x;
int x2 = vect[i+1].x;
int x3 = vect[i+2].x;

//calculate x-value of barycenter
generatedPoint.x = (x1 + x2 + x3)/3;

 //y-value of triangle vertex
int y1 = vect[i].y;
int y2 = vect[i+1].y;
int y3 = vect[i+2].y;
```

```
//calculating y-value of barycenter
generatedPoint.y = (y1 + y2 + y3)/3;


if(IsTrianglePointInside(lParam, hInstance, generatedPoint)== 1)
break;
}
break;
}
return generatedPoint;
}
```

`GeneratePointInTriangle` berechnet den Schwerpunkt eines Dreieckes, der innerhalb dieses Dreieckes liegt. Der Vektor, in dem die Punkte des Loches gespeichert sind, wird durchlaufen. Immer drei aufeinanderfolgende Punkte des Vektors werden als Dreieck angesehen.

```
int x1 = vect[i].x;
int x2 = vect[i+1].x;
int x3 = vect[i+2].x;
```

Aus diesen drei Werten berechnet die Funktion den x-Wert des Schwerpunktes des Dreiecks, in dem alle drei Werte erst addiert und dann durch drei geteilt werden.

```
generatedPoint.x = (x1 + x2 + x3)/3;
```

Das gleiche wird mit den y-Werten der ersten drei Punkte gemacht. Hat man dadurch den x- und y-Wert des Schwerpunktes errechnet, muss überprüft werden, ob sich der Punkt innerhalb des Polygons befindet. An folgendem Beispiel kann man sehen, dass dies nicht immer der Fall ist:

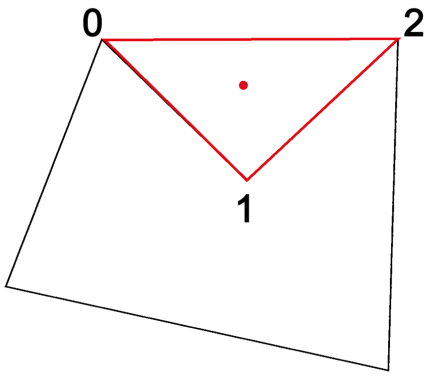

Abbildung 34: generierter Punkt, der außerhalb des Polygons liegt

Die Punkte null, eins und zwei stellen die ersten drei Punkte des schwarz gezeichneten Polygons dar. Diese drei Punkte bilden das rote Dreieck. Wie man leicht erkennen kann, liegt der Schwerpunkt des Dreiecks außerhalb des Polygons. Die Funktion `IsTrianglePointInside` überprüft, ob ein Punkt inner- oder außerhalb eines Polygons liegt. Sie benutzt dazu die Funktion `LineCrossLine`, die, wie bereits erklärt, berechnet wie viele Schnittpunkte eine Linie mit allen Linien eines Polygons besitzt.

```
int boundary::IsTrianglePointInside(LPARAM lParam, HINSTANCE hInstance,
 POINT generatedPoint)
{
insideHole = 0;
POINT zero;
zero.x = 0;
zero.y = 0;

int numberOfIntersections = LineCrossLine(lParam, hInstance, zero,
generatedPoint);

if(numberOfIntersections %2 == 1)
{
insideHole = 1;
}

return insideHole;
}
```

In der Variable `numberOfIntersections` ist die Anzahl der Schnittpunkte einer

Linie vom generierten Punkt zum Nullpunkt mit allen Linien des Loches gespeichert. Die Funktion `IsTrianglePointInside` gibt eine „eins“ zurück, wenn die Anzahl der Schnittpunkte ungerade, der Punkt also innerhalb des Polygons liegt. Liegt der Punkt nicht innerhalb, dann wird aus den nächsten drei Punkten im Vektor ein Dreieck gebildet und von diesem der Schwerpunkt berechnet und überprüft, ob er innerhalb des Polygons liegt, usw. Hat man den Punkt generiert, wird er in `in.holelist` eingetragen. Sind die drei erforderlichen Arrays gefüllt, kann man `triangulate` aufrufen. Diese Funktion generiert unter anderem `out.pointlist`, `out.trianglelist` und `out.neighborlist`, sowie `out.numberofpoints` und `out.numberoftriangles`. In der Punktliste, die `triangulate` generiert, sind die Punkte aller Dreiecke gespeichert. Diese Punkte werden in das Array `nodes` gespeichert, in dessen einzelnen Feldern die Struktur `POINT` abgespeichert werden kann. Das heißt, jedes Feld erhält x- und y-Wert eines Punktes.

```
nNodes = out.numberofpoints; //number of points

//filling the array nodes
//one node consists of the x and y value of a point
for(int l=0; l<nNodes; l++)
{
nodes[l].x = out.pointlist[l*2];
nodes[l].y = out.pointlist[l*2+1];
}
```

out.pointlist

180	160	200	190	260	190	220	160

nodes

180	200	260	220
160	190	190	160

Abbildung 35: out.poinlist und Array nodes

Das Array `out.trianglelist` ist mit integer-Werten gefüllt, die angeben aus welchen Punkten, die in `out.pointlist` gespeichert sind, die Dreiecke bestehen.

Immer drei aufeinanderfolgende Zahlen gehören zu einem Dreieck. Von jedem Dreieck sind seine drei Eckpunkte gespeichert. Dreiecke, die nebeneinander liegen, haben eine gemeinsame Kante, und somit sind auch zwei ihrer Eckpunkte gleich. In dem unteren Bild sieht man zwei Dreiecke. Für das nullte Dreieck sind die Werte 1, 0 und 2 abgespeichert, für das Erste 3, 2 und 0 usw.

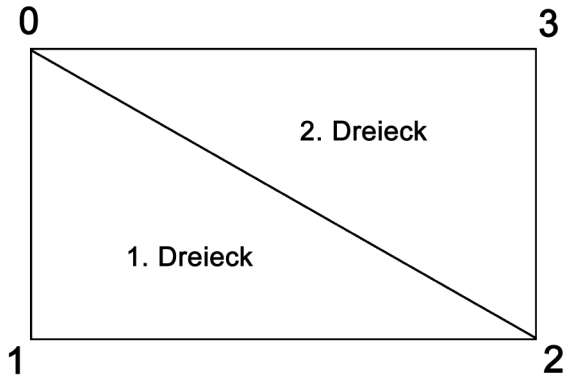

Abbildung 36: zwei angrenzende Dreiecke mit den zugehörigen Eckpunkten

Das Array out.trianglelist sieht also folgendermaßen aus:

1	0	2	3	2	0

Abbildung 37: out.trianglelist

Zusätzlich gibt `triangulate out.neighborlist` zurück. Ein Dreieck hat drei Nachbarn; der erste Nachbar liegt gegenüber des ersten Eckpunktes, der Zweite gegenüber des zweiten Eckpunktes und der dritte Nachbar gegenüber des dritten Eckpunktes. Auch die Nachbarn sind als integer-Werte abgespeichert. Dieser Wert gibt den Index des Nachbardreiecks an. Der Wert null zum Beispiel definiert das Erste in `out.trianglelist` gespeicherte Dreieck.

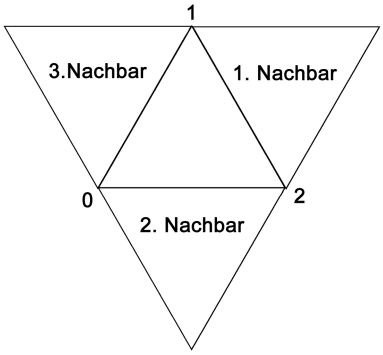

Abbildung 38: Dreieck mit seinen drei Nachbarn

Hat ein Dreieck eine oder zwei äußere Kanten, beträgt der Wert des entsprechenden Nachbarn -1.

Um die Zusammenhänge zu verdeutlichen noch ein Beispiel:

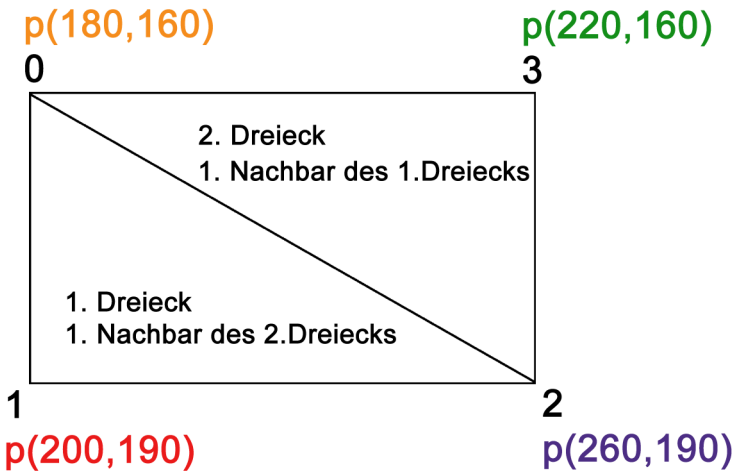

Abbildung 39: zwei aneinandergrenzende Dreiecke mit Informationen über Eckpunkte, Koordinaten der Eckpunkte und Nachbarn der Dreiecke

Das dazugehörige Array `out.pointlist` sieht so aus

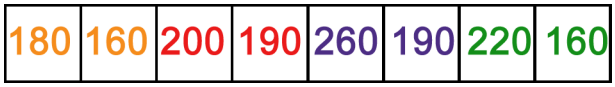

Abbildung 40: zum Beispiel gehörende out.pointlist

und `out.trianglelist` folgendermaßen:

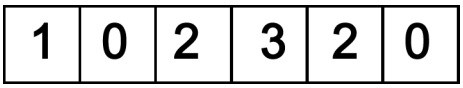

Abbildung 41: zum Beispiel gehörende out.trianglelist

So verweist eine eins in `out.trianglelist` auf den zweiten Punkt (200,190) in `out.pointlist` etc.

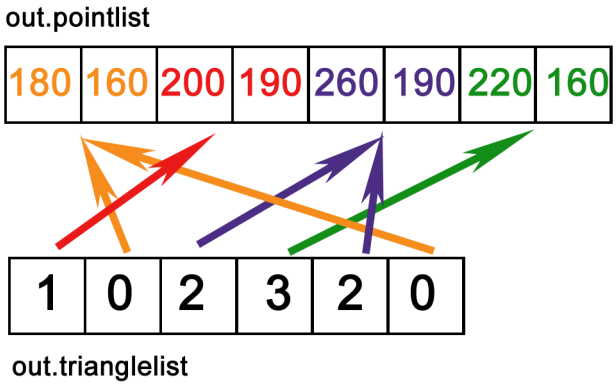

Abbildung 42: Zusammenhang zwischen out.pointlist und out.trianglelist

`out.neighborlist` sieht so aus:

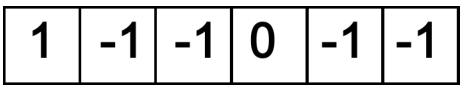

Abbildung 43: zum Beispiel gehörende out.neighborlist

Wie man erkennen kann, sind nur zwei Nachbarn, folglich eine innere Kante und vier äußere Kanten, vorhanden. Die ersten drei Felder des Arrays geben die Nachbarn des ersten Dreiecks an. Das erste Dreieck hat nur einen Nachbarn, den Ersten. Die Ziffer eins verweist darauf, dass das Dreieck, welches diesen Nachbarn darstellt, das Zweite in `out.trianglelist` ist. Das zweite Dreieck hat ebenfalls nur einen Nachbarn, welcher das erste in `out.trianglelist` gespeicherte Dreieck darstellt.
Um mehr Informationen in einem Dreieck speichern zu können, wurde eine Struktur `Triangle` angelegt.

```
typedef struct
{
int v1, v2, v3;
int v4; //centerpoint of borderedge of bordertriangle (no coordinate)
int v6; //barycenter of lonely triangle
int neighbor1, neighbor2, neighbor3;
int marker1; //for to know if triangle is a lonely triangle,
0 if it is lonely
int marker2;  //for to know if it has two outer edges (-2)
//and to know if it was used to build basic block (-3)
int ind; //position in trianglearray
POINT pointOfGravity;
POINT centerOfLine;
POINT centerOfBlock; //a block consists of two triangles;
each of this //triangles knows the center of the bloc
int numberOfCenter; //int of center of block
} Triangle;
```

Ein neues Array `triangles`, das Daten vom Typ Triangle enthält, wird erzeugt.
Die Werte aus
`out.trianglelist` und `out.neighborlist` werden in dieses Array übernommen. Jedem Dreieck werden seine drei Eckpunkte v1, v2 und v3 zugeordnet; v1 für den ersten Punkt des Dreiecks, v2 für den zweiten Punkt und v3 für den dritten Punkt. Zusätzlichen bekommt jedes Dreieck seine Nachbarn `neighbor1, neighbor2` und `neighbor3` zugewiesen.

```
nTriangles = out.numberoftriangles; //number of triangles
//filling the array triangles
//one triangle consists of v1, v2 and v3, these are the edges of a triangle
```

```
for(int j=0; j<nTriangles; j++)
{
triangles[j].v1 = out.trianglelist[j*3];
triangles[j].v2 = out.trianglelist[j*3+1];
triangles[j].v3 = out.trianglelist[j*3+2];
triangles[j].marker1 = 0;
triangles[j].marker2 = 0;
triangles[j].v4 = 0;
triangles[j].ind = j;
triangles[j].v6 = 0;
}
//every triangle has three neighbours
//neighbour1 is opposite v1
//neighbour2 is opposite v2
//neighbour3 is opposite v3
for(int t = 0; t<nTriangles; t++)
{
triangles[t].neighbor1 = out.neighborlist[t*3];
triangles[t].neighbor2 = out.neighborlist[t*3+1];
triangles[t].neighbor3 = out.neighborlist[t*3+2];
}
```

Die restlichen Werte, die zu der Struktur `Triangle` gehören, werden vorerst mit null gefüllt. Sie bekommen im Laufe des Programms Werte zugeordnet.

3.7 Implementierung des Algorithmus von Luiz Velho

Alle Dreiecke der von triangulate erzeugten Triangulierung sind im Array triangles gespeichert. Alle inneren Kanten der Dreiecke müssen genommen und, mit der längsten inneren Kante beginnend, je zwei Dreiecke, die eine Kante teilen zu einem basic block zusammengefasst werden. Die äußeren Kanten müssen also unbeachtet bleiben. Zu Beginn werden alle Kanten der Dreiecke in einen Vektor gespeichert. In diesem Vektor enthält jedes Feld Daten vom Typ `Edge`.

```
typedef struct
{
int id, from, to;
double length;
} Edge;
```

Aus jedem Dreieck erhält man drei Kanten. Die erste Kante verläuft von v1 (from-Wert der ersten Kante) nach v2 (to-Wert der ersten Kante), die zweite Kante von v2 nach v3 und die dritte Kante von v3 nach v1. Die Id einer Kante ist die Speicherstelle selbiger Kante im Vektor. Zusätzlich wird die Länge der einzelnen Kanten berechnet.

```
for(int index = 0; index<nTriangles; index++)
{
edges.from = triangles[index].v1;
edges.to = triangles[index].v2;

edges.length = length(edges.from, edges.to);
edges.id = index*3;
edgeVector.push_back(edges);

edges.from = triangles[index].v2;
edges.to = triangles[index].v3;

edges.length = length(edges.from, edges.to);
edges.id = index*3 +1;

edgeVector.push_back(edges);

edges.from = triangles[index].v3;
edges.to = triangles[index].v1;

edges.length = length(edges.from, edges.to);
edges.id = index*3 +2;

edgeVector.push_back(edges);
}
```

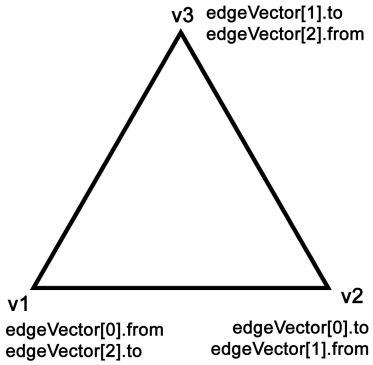

Abbildung 44: Dreieck mit Kanteninformationen

Um aus den inneren Kanten basic blocks bilden zu können, müssen zunächst alle äußeren Kanten aus dem `edgeVector` gelöscht werden. Dies geschieht in der Funktion `eraseOuterEdge`. In dieser Funktion werden bei jedem Schleifendurchlauf durch das Array `triangles` die drei Nachbarn eines Dreiecks zurückgegeben. Es wird überprüft welcher dieser Nachbarn den Wert -1 hat; dies bedeutet, dass dieser Nachbar nicht vorhanden ist. So kann man überprüfen, welche Kante eines Dreiecks eine Äußere ist und diese löschen. Ist der `neighbor1` = -1, muss die Kante, die aus den Punkten v2 und v3 besteht, aus dem Vektor gelöscht werden. Ist `neighbor2` = -1 muss die Kante von v1 nach v3 gelöscht werden. Die Kante von v1 nach v2 muss demnach gelöscht werden, wenn der `neighbor3` den Wert -1 hat. Da ein Dreieck auch zwei äußere Kanten besitzen kann, müssen drei weitere Möglichkeiten betrachtet werden.

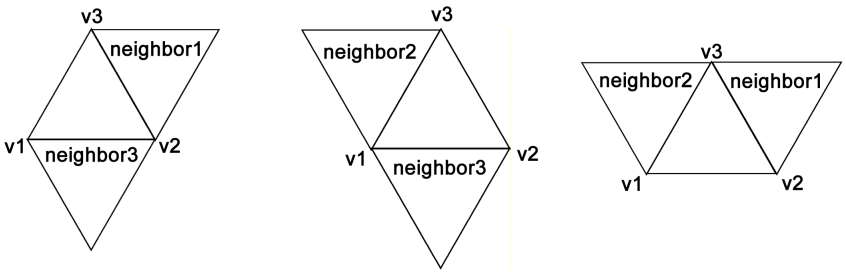

Abbildung 45: Kombinationsmöglichkeiten Dreieck-zwei Nachbarn

Aus der Abbilddung lässt sich erkennen, welche Kante zu löschen ist. Zusätzlich wird der `marker2` dieser Dreiecke auf -2 gesetzt, damit später noch zu erkennen ist, dass das Dreieck zwei äußere Kanten besitzt.

```
//erases the outer edges
void allTriangulations::eraseOuterEdge()
{
for(int t=0; t<nTriangles; t++)
{
//give the neighbours of the triangle
int n1 = triangles[t].neighbor1;
int n2 = triangles[t].neighbor2;
int n3 = triangles[t].neighbor3;
//if one neighbour is <= -1, the neighbor doesn't exist
//it is a triangle with an outer edge
if(n1 <= -1)
{
EraseUsedEdges(triangles[t].v2, triangles[t].v3);
}
if(n2 <= -1)
{
EraseUsedEdges(triangles[t].v1, triangles[t].v3);
}
if(n3 <= -1)
{
EraseUsedEdges(triangles[t].v1, triangles[t].v2);
}
if(n1 <= -1 && n2 <= -1)
{
EraseUsedEdges(triangles[t].v1, triangles[t].v3);
EraseUsedEdges(triangles[t].v3, triangles[t].v2);
//this triangle has two outer edges
triangles[t].marker2 = -2;
}
if(n2 <= -1 && n3 <= -1)
{
EraseUsedEdges(triangles[t].v1, triangles[t].v3);
EraseUsedEdges(triangles[t].v1, triangles[t].v2);
//this triangle has two outer edges
triangles[t].marker2 = -2;
}
if(n1 <= -1 && n3 <= -1)
{
EraseUsedEdges(triangles[t].v2, triangles[t].v3);
EraseUsedEdges(triangles[t].v1, triangles[t].v2);
//this triangle has two outer edges
triangles[t].marker2 = -2;
}
}
}
```

Sind alle Kanten aus dem Vektor gelöscht, kann aus den verbleibenden inneren Kanten das „intermediate mesh“ gebildet werden, das aus basic blocks und einzelnen Dreiecken besteht. Ein basic block setzt sich aus zwei angrenzenden Dreiecken zusammen. Begonnen wird mit der längsten inneren Kante; zu dieser Kante werden die beiden Dreiecke gesucht, die die Kante teilen. Aus den beiden Dreiecken wird ein basic block gebildet. Als nächstes bildet man den basic block ausgehend von der zweitlängsten Kante usw. Da nicht alle Dreiecke zu basic blocks zusammengefasst werden können, besteht das Netz nachher aus basic blocks und einzelnen Dreiecken.

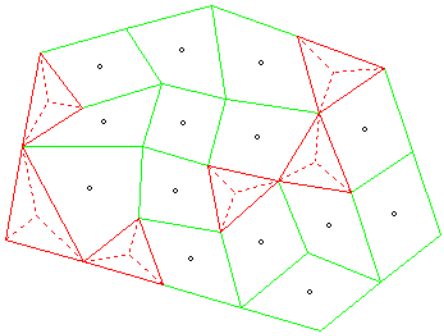

Abbildung 46: intermediate mesh

Wenn das „intermediate mesh“ vorliegt, werden neue basic blocks gebildet. Dazu werden nach dem Algorithmus von Luiz Velho die Schwerpunkte der Dreiecke, sowie der basic blocks, benötigt. Im Virtual Tailor kann der Benutzter jedoch selbst wählen an welcher Stelle der basic block gesplittet werden soll.

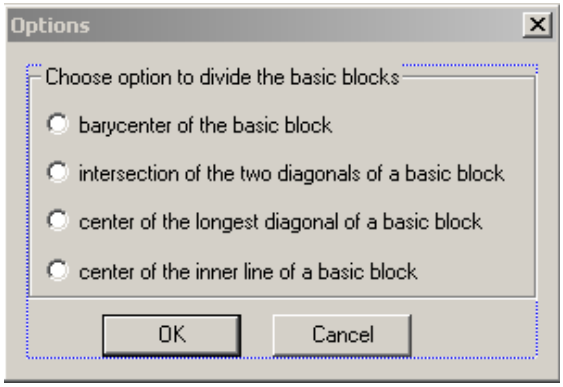

Abbildung 47: Dialog mit Auswahlmöglichkeiten zum Splitpunkt

Welche Option der Nutzer gewählt hat, wird der Funktion `BuildIntermediateMesh(int numberOfPressedButton, int numberOfChoosenOption)` im ersten Übergabewert `numberOfPressedButton` mitgeteilt. Der zweite Wert teilt mit, welche Option der Benutzer gewählt hat, um konkave basic blocks zu behandeln.

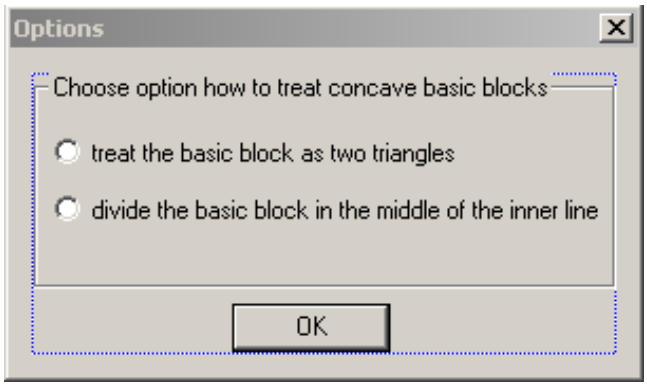

Abbildung 48: Dialog mit Auswahlmöglichkeiten zum Behandeln von konkaven basic blocks

Die Funktion `BuildIntermediateMesh` bildet das „intermediate mesh“ mit seinen basic blocks und Dreiecken und berechnet den Splitpunkt der basic blocks und den Schwerpunkt der einzeln liegenden Dreiecke. Der Vektor mit den doppelt gespeicherten inneren Kanten wird durchlaufen. Immer zwei gleiche Kanten liegen im Vektor hintereinander. Mit den längsten Kanten beginnend werden die beiden Dreiecke die diese Kante besitzen, und somit nebeneinander liegen müssen, gesucht.

```
int id = edgeVector[i].id; //id of the longest edge
int triangleId = id/3; //which triangle
int ed1 = id % 3; //which edge is inner edge
//if ed1 equals 0 v1v2 is inner edge
//if ed1 equals 1 v2v3 is inner edge
//if ed1 equals 2 v3v1 is inner edge

int id2 = edgeVector[i+1].id;
int triangleId2 = id2/3;
int ed2 = id2 % 3;
```

Die Id einer Kante gibt an, an welcher Stelle im Vektor die Kante gespeichert wurde. Pro Dreieck wurden drei Kanten gespeichert, die im Vektor hintereinander lagen. Um von der Id der Kante auf die Stelle im Array zu schließen, an der das zu der Kante gehörende Dreieck gespeichert wurde, wird die Id der Kante durch drei dividiert. Da bekannt sein muss, welche Kante des Dreiecks, die von

v1 nach v2, von v2 nach v3 oder die von v3 nach v1, die gerade vorliegende Kante ist, wird die Id der Kante modulo drei geteilt. Die Kante v1v2 wurde als erste Kante des Dreiecks, die Kante v2v3 als Zweite und v3v1 als dritte Kante des Dreiecks gespeichert. Dividiert man die Id einer Kante durch drei, erhält man immer einen Rest von null, eins oder zwei. Da die Kante v1v2 jedes Mal als erstes gespeichert wurde, weiß man bei einem Rest von null, dass diese Kante vorliegt. Der Rest von eins weist auf die Kante v2v3 hin und der Rest von drei auf die Kante v3v1. Hat man die beiden zueinander gehörenden Dreiecke gefunden, können die Daten in die neue Struktur Block gespeichert werden. Diese sieht folgendermaßen aus:

```
typedef struct{
Edge edge1, edge2, edge3, edge4;
Edge innerEdge;
Edge diagonal;
POINT center; //coordinates of the center
int v5; //number of the center
Triangle triangle1, triangle2; //block consists of this two triangles
} Block;
```

Ein basic block besteht aus vier äußeren Kanten, einer inneren Kante, einer Diagonalen, dem Mittelpunkt, einem integer-Wert für den Mittelpunkt und zwei Dreiecken. Die ersten beiden Kanten des basic blocks, `edge1` und `edge2`, werden aus dem ersten Dreieck gewonnen, die dritte und die vierte Kante aus dem zweiten Dreieck. Hierbei muss darauf geachtet werden welches die gemeinsame Kante, also die innere Kante des späteren basic blocks, des Dreiecks ist. Die beiden anderen Kanten des Dreiecks werden als Kanten des Blocks abgespeichert. Als innere Kante wird die Kante aus dem `edgeVector` abgespeichert, die zum ersten Dreieck gehört.

```
//first triangle
if(ed1 == 0)
{
//edge v1v2 is inner edge
block.edge1.from = triangles[triangleId].v1;
block.edge1.to = triangles[triangleId].v3;
block.edge2.from = triangles[triangleId].v3;
block.edge2.to = triangles[triangleId].v2;

block.innerEdge.from = triangles[triangleId].v1;
block.innerEdge.to = triangles[triangleId].v2;
```

Da die innere Kante des basic blocks bereits bekannt ist, kann man als Anfangspunkt der Diagonalen den verbleibenden dritten Punkt des ersten Dreiecks abspeichern. Der Endpunkt der Diagonalen ist einer der drei Punkte aus dem

zweiten Dreieck. Da man auch vom zweiten Dreieck die „gemeinsame“ Kante kennt, ist der dritte Punkt dieses Dreiecks der Endpunkt der Diagonalen des basic blocks.

```
//first point of second diagonal line
block.diagonal.from = triangles[triangleId].v3;


//second point of second diagonal line
//you need to look at the second triangle
switch(ed2)
{
case 0:
{
block.diagonal.to = triangles[triangleId2].v3;
break;
}
```

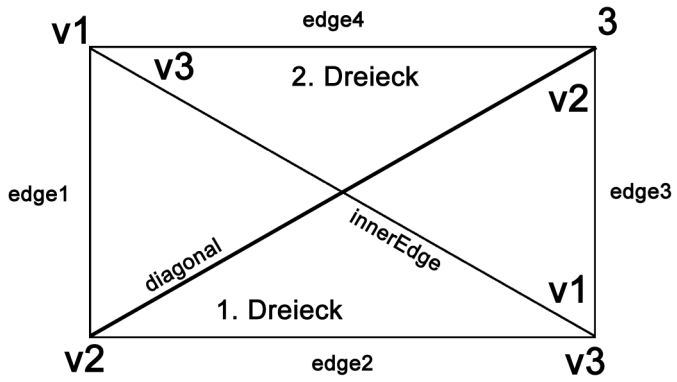

Abbildung 49: Dreieck mit Zusatzinformationen

Ist die dritte und die vierte Kante im basic block abgespeichert, kann man den Punkt berechnen an dem die basic blocks zur Bildung der neuen basic blocks gesplittet werden. Dieser Punkt wird als `block.center` abgespeichert. Je nachdem welche Option der Benutzer ausgewählt hat, kann der Splitpunkt der Schwerpunkt (Flächenschwerpunkt) des basic blocks, der Schnittpunkt der beiden Diagonalen des basic blocks, der Mittelpunkt der längeren Diagonalen oder der Mittelpunkt der `innerEdge` des Blocks sein. Das Problem bei konkaven basic blocks ist, dass der Schwerpunkt des basic blocks und der Schnittpunkt

der Diagonalen außerhalb des basic blocks liegen und somit nicht als Punkt für die Bildung der neuen basic blocks herangezogen werden können. Wird die Option gewählt, am Mittelpunkt der längeren Diagonalen zu teilen, ist nicht sicher, dass der Punkt innerhalb des basic blocks liegt. Um zu überprüfen ob ein basic block konkav ist wird bei den verschiedenen Optionen unterschiedlich vorgegangen. Um den Schwerpunkt in einem basic block zu berechnen, teilt man diesen an seinen beiden Diagonalen. Man erhält zu jeder Seite jeder Diagonalen zwei Dreiecke, in der Summe demnach vier. Von diesen vier Dreiecken werden die Schwerpunkte berechnet.

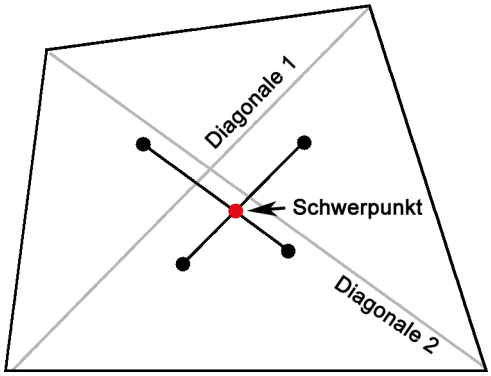

Abbildung 50: Schwerpunkt eines basic blocks

Verbindet man die jeweils beiden gegenüberliegenden Schwerpunkte erhält man zwei sich schneidende Linien. Der Schnittpunkt dieser Linien ist der Schwerpunkt des basic blocks. Wenn der basic block konkav ist, liegt kein Schnittpunkt dieser beiden Linien vor und `block.center` wird auf -1 gesetzt. Man kann mit einer Abfrage, ob `block.center == -1` ist prüfen, ob der basic block konkav ist. Ähnlich funktioniert die Überprüfung wenn der Benutzer den Schnittpunkt der Diagonalen als Splitpunkt wählt; denn wenn sich die Diagonalen eines basic blocks nicht schneiden, dann ist der basic block konkav. Wenn als Splitpunkt der Mittelpunkt der längeren Diagonalen gewählt ist, kann man mittels Schnittpunktberechnung überprüfen, ob der basic block konkav ist oder nicht. Für den Fall, dass der Benutzer den Mittelpunkt der `innerEdge` gewählt hat, wird ebenfalls berechnet ob sich die beiden Diagonalen dieses basic blocks schneiden. Anhand dieser Überprüfungen weiß man, welche basic blocks konkav sind. Ist ein basic block konkav, wurde, außer der Benutzer hat als Splitpunkt den Mittelpunkt der `innerEdge` gewählt, kein gültiger Punkt innerhalb des basic blocks berechnet. Zu diesem Zweck bekommt die Funktion `BuildIntermediateMesh` den zweiten Wert, `int numberOfChoosenOption`, übergeben. Dieser Wert zeigt an, wie der Benutzer die konkaven basic blocks behandeln möchte. Er hat die Möglichkeit zu entscheiden, dass die konkaven basic blocks als zwei einzelne Dreiecke angesehen werden, oder dass als Splitpunkt des konkaven basic blocks der Mittelpunkt der `innerEdge` gewählt wird (siehe Abbildung 48). Wählt man

die zweite Option kann es allerdings passieren, dass auch unter den neuen basic blocks konkave auftreten.

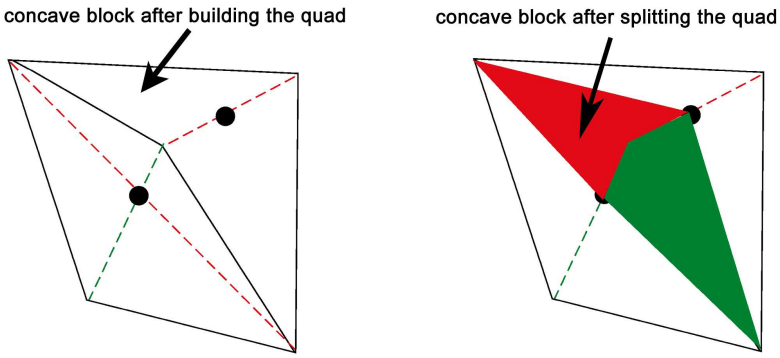

Abbildung 51: Auftreten konkaver Blocks

Wurden alle benötigten Daten in der Struktur Block gespeichert, müssen alle Kanten die zu den beiden Dreiecken gehören aus denen man den basic block gebildet hat aus dem `edgeVector` gelöscht werden, damit sie nicht mehr zur Bildung eines weiteren basic blocks herangezogen werden. Da alle Kanten doppelt im `edgeVector` gespeichert sind, müssen auch die identischen Kanten aus den angrenzenden Dreiecken gelöscht werden, die nicht zur Bildung des basic blocks beigetragen haben. Diese dürfen nicht weiter betrachtet werden, da sie nur noch einzeln im `edgeVector` gespeichert sind.

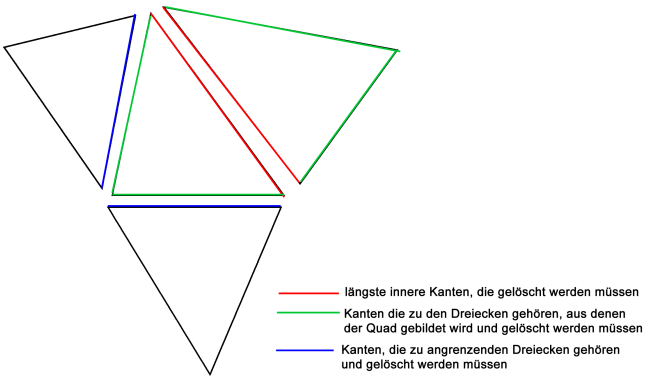

Abbildung 52: Kanten, die gelöscht werden müssen

Dieses führt die Funktion `EraseUsedEdges` durch. Sie bekommt einen to- und einen from-Wert übergeben, durchsucht den `edgeVector` nach Kanten mit diesen beiden Werten und löscht die Kanten aus dem `edgeVector`.

```
void allTriangulations::EraseUsedEdges(int from, int to)
```

```
{
for(int i = 0; i<edgeVector.size(); i++)
{
if((from == edgeVector[i].from && to == edgeVector[i].to)
|| (from == edgeVector[i].to && to == edgeVector[i].from))
{
vector<Edge>::iterator iter = edgeVector.begin();
edgeVector.erase(iter +i);
i--;
}
}
}
```

Um die Information welche Dreiecke zur Bildung von basic blocks herangezogen wurden nicht zu verlieren, wird der `marker1` der Dreiecke auf -1 gesetzt. Da nicht alle Dreiecke zur Bildung von basic blocks herangezogen werden können, bleiben im „intermediate mesh“ einige Dreiecke einzeln zurück. Diese sind die Dreiecke deren `marker1` noch auf null gesetzt ist.

Es liegen demnach basic blocks vor, deren Dreiecke mit dem `marker1 = -1` besetzt sind und einzelne Dreiecke deren `marker1` noch auf null gesetzt ist. Zusätzlich haben Dreiecke die am Rand liegen, unabhängig davon, ob sie einzelne Dreiecke, oder zu einem basic block gehörende Dreiecke sind, den `marker2` auf -2 gesetzt. Um aus den vorliegenden basic blocks und Dreiecken neue basic blocks zu bilden wurde der Splitpunkt innerhalb eines basic blocks, sowie die Schwerpunkte der einzelnen Dreiecke, berechnet. Es gibt verschiedene Möglichkeiten, wie die Elemente (basic blocks und Dreiecke) angeordnet sind:

1. basic blocks neben basic blocks,
2. basic blocks neben Dreiecken
3. Dreiecke neben Dreiecken, falls der Benutzer die Option gewählt hat, konkave basic blocks als zwei Dreiecke zu behandeln.

Aufgrund dieser Varianten, müssen verschiedene Verfahren entwickelt werden um die neuen basic blocks zu speichern. Liegen zwei basic blocks nebeneinander besteht der neue Block aus den Splitpunkten der beiden basic blocks und dem Start- und Endpunkt ihrer gemeinsamen Kante.

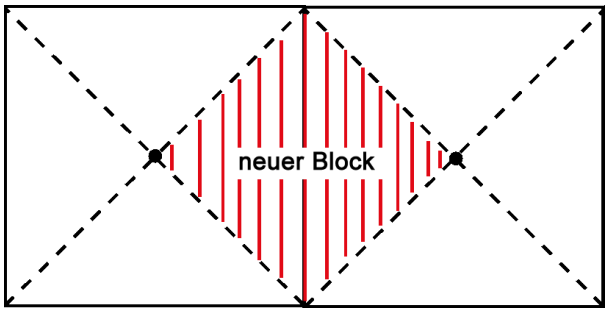

Abbildung 53: Bildung eines neuen basic blocks aus zwei basic blocks

Liegt ein basic block neben einem einzelnen Dreieck, besteht der neue Block aus dem Splitpunkt des basic blocks, dem Schwerpunkt des Dreiecks und dem Start- und Endpunkt der gemeinsamen Kante.

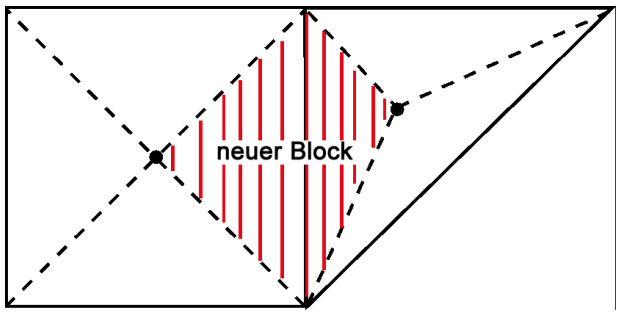

Abbildung 54: Bildung eines neuen basic blocks aus einem basic block und einem Dreieck

Hat der Benutzer die Option gewählt, konkave basic blocks als jeweils zwei Dreiecke zu betrachten, können auch zwei einzelne Dreiecke nebeneinander liegen. Hier besteht der neue Block aus den beiden Schwerpunkten der nebeneinanderliegenden Dreiecke sowie dem Start- und Endpunkt ihrer gemeinsamen Kante.

Sind die neuen basic blocks gebildet worden, liegen am Rand sogenannte „single triangle boundary blocks“. Die neuen basic blocks werden in einer neuen Struktur abgespeichert, welche aus vier Partikeln, fünf Kanten und zwei Dreiecken besteht.

```
typedef struct{
Particle p1, p2, p3, p4;
```

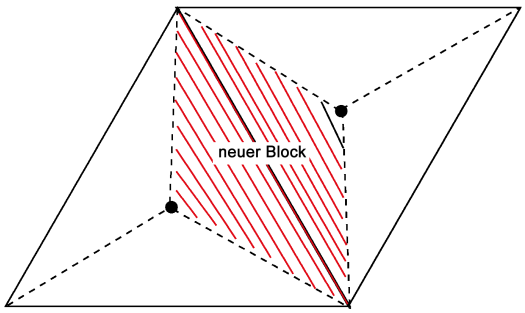

Abbildung 55: Bildung eines neuen basic blocks aus zwei Dreiecken

```
NewEdge edge1, edge2, edge3, edge4, innerEdge;
NewTriangle triangle1, triangle2; //newBlock consists of two triangles
} NewBlock;
```

Ein Partikel beinhaltet die Koordinate eines Punktes, sowie einem dem Punkt zugeordneten integer-Wert.

```
typedef struct{
POINT point;
int identifier;
} Particle;
```

p1 und p4 sind die Splitpunkte der basic blocks bzw. Schwerpunkte der Dreiecke, p2 und p3 sind Start- und Endpunkt der vorherigen gemeinsamen Kante.

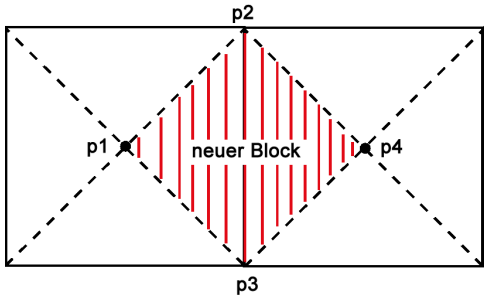

Abbildung 56: Bezeichnung der Punkte eines neuen basic blocks

`NewEdge` besteht aus einem identifier `(int id)` und dem Start- und Endpunkt einer Kante `(int from, int to)`. `edge1` beschreibt die Kante zwischen `p1` und `p2`, `edge2` die Kante zwischen `p2` und `p4`, `edge3` zwischen `p4` und `p3` und `edge4` die Kante zwischen `p3` und `p1`. `inneredge` verbindet die Partikel `p2` und `p3` und stellt somit die vorherige gemeinsame Kante dar. In der Struktur

NewTriangle sind die drei Kanten eines Dreiecks gespeichert, wobei die dritte Kante (edge3) die innerEdge des neuen basic blocks darstellt. Jedes Dreieck erhält zusätzlich einen identifier.

```
typedef struct{
NewEdge edge1, edge2, edge3; //third edge is former innerEdge of newBlock
int identifier;
}NewTriangle;
```

Das Abspeichern der neuen basic blocks in der neuen Struktur erfüllt die Funktion BuildBasicBlocks(int numberOfChoosenOption).

```
void allTriangulations::BuildBasicBlocks(int numberOfChoosenOption)
{
//one block has four little triangles
//one lonely triangle has three little triangles
//store these little triangles
StoreLittleTriangles();
for(int b = 0; b<blockVector.size(); b++)
{
//first triangle:
//if a triangle has two outer edges the marker2 is set to -2
//if this is the case "ignore" this triangle
//ignore the triangle if the marker2 is set to -3
//because this means that this triangle was already used to build a
//new block
//use the triangle if the marker2 is set to 0
if(blockVector[b].triangle1.marker2 == 0)
{
//neighbors of first subtriangle
int n1 = triangles[blockVector[b].triangle1.ind].neighbor1;
int n2 = triangles[blockVector[b].triangle1.ind].neighbor2;
int n3 = triangles[blockVector[b].triangle1.ind].neighbor3;
//every triangle has two or three neighbors, but one of them is
//the second subtriangle that should be "ignored"
if((triangles[n1].marker1 == -1) && (n1 != -1) &&
(n1 != blockVector[b].triangle2.ind) && (triangles[n1].marker2 != -3))
{
//store new block
int place = StoreNewBlock1(n1,b,1, blockVector[b].triangle1.ind,
triangles[n1].centerOfBlock, triangles[n1].numberOfCenter);

DeleteLittleTriangles(place);
}
else if((triangles[n1].marker1 == 0) && (n1 != -1) &&
(n1 != blockVector[b].triangle2.ind) && (triangles[n1].marker2 != -3))
{
```

```
//store it this way if neighbor is a lonely triangle
int place = StoreNewBlock1(n1,b,1, blockVector[b].triangle1.ind,
triangles[n1].pointOfGravity, triangles[n1].v6);
DeleteLittleTriangles(place);
}
if((triangles[n2].marker1 == -1) && (n2 != -1) &&
(n2 != blockVector[b].triangle2.ind ) && (triangles[n2].marker2 != -3))
{
//store new block
int place = StoreNewBlock1(n2,b,2, blockVector[b].triangle1.ind,
triangles[n2].centerOfBlock, triangles[n2].numberOfCenter);
DeleteLittleTriangles(place);
}
else if((triangles[n2].marker1 == 0) && (n2 != -1) &&
(n2 != blockVector[b].triangle2.ind) && (triangles[n2].marker2 != -3))
{
//store it this way if neighbor is a lonely triangle
int place = StoreNewBlock1(n2,b,2, blockVector[b].triangle1.ind,
triangles[n2].pointOfGravity, triangles[n2].v6);
DeleteLittleTriangles(place);
}
if((triangles[n3].marker1 == -1) && (n3 != -1) &&
(n3 != blockVector[b].triangle2.ind) && (triangles[n3].marker2 != -3))
{
//store new block
int place = StoreNewBlock1(n3,b,3, blockVector[b].triangle1.ind,
triangles[n3].centerOfBlock, triangles[n3].numberOfCenter);
DeleteLittleTriangles(place);
}
else if((triangles[n3].marker1 == 0) && (n3 != -1) &&
(n3 != blockVector[b].triangle2.ind) && (triangles[n3].marker2 != -3))
{
//store it this way if neighbor is a lonely triangle
int place = StoreNewBlock1(n3,b,3, blockVector[b].triangle1.ind,
triangles[n3].pointOfGravity, triangles[n3].v6);
DeleteLittleTriangles(place);
}
}
//second triangle:
//if a triangle has two outer edges the marker2 is set to -2
//if this is the case "ignore" this triangle
//ignore the triangle if the marker2 is set to -3
//because this means that this triangle was already used to build a
// new block use the triangle if the marker2 is set to 0
if(blockVector[b].triangle2.marker2 == 0)
{
```

```
//neighbors of second subtriangle
int n12 = triangles[blockVector[b].triangle2.ind].neighbor1;
int n22 = triangles[blockVector[b].triangle2.ind].neighbor2;
int n32 = triangles[blockVector[b].triangle2.ind].neighbor3;
if((triangles[n12].marker1 == -1) && (n12 != -1) &&
(n12 != blockVector[b].triangle1.ind) && (triangles[n12].marker2 != -3))
{
//store new block
int place = StoreNewBlock1(n12,b,1, blockVector[b].triangle2.ind,
triangles[n12].centerOfBlock, triangles[n12].numberOfCenter);
DeleteLittleTriangles(place);
}
else if((triangles[n12].marker1 == 0) && (n12 != -1) &&
(n12 != blockVector[b].triangle1.ind) && (triangles[n12].marker2 != -3))
{
//store it this way if neighbor is a lonely triangle
int place = StoreNewBlock1(n12,b,1, blockVector[b].triangle2.ind,
triangles[n12].pointOfGravity, triangles[n12].v6);
DeleteLittleTriangles(place);
}
if((triangles[n22].marker1 == -1) && (n22 != -1) &&
(n22 != blockVector[b].triangle1.ind) && (triangles[n22].marker2 != -3))
{
//store new block
int place = StoreNewBlock1(n22,b,2, blockVector[b].triangle2.ind,
triangles[n22].centerOfBlock, triangles[n22].numberOfCenter);
DeleteLittleTriangles(place);
}
else if((triangles[n22].marker1 == 0) && (n22 != -1) &&
(n22 != blockVector[b].triangle1.ind) && (triangles[n22].marker2 != -3))
{
//store it this way if neighbor is a lonely triangle
int place = StoreNewBlock1(n22,b,2, blockVector[b].triangle2.ind,
triangles[n22].pointOfGravity, triangles[n22].v6);
DeleteLittleTriangles(place);
}
if((triangles[n32].marker1 == -1) && (n32 != -1) &&
(n32 != blockVector[b].triangle1.ind) && (triangles[n32].marker2 != -3))
{
//store new block
int place = StoreNewBlock1(n32,b,3, blockVector[b].triangle2.ind,
triangles[n32].centerOfBlock, triangles[n32].numberOfCenter);
DeleteLittleTriangles(place);
}
else if((triangles[n32].marker1 == 0) && (n32 != -1) &&
(n32 != blockVector[b].triangle1.ind) && (triangles[n32].marker2 != -3))e
```

```
{
//store it this way if neighbor is a lonely triangle
int place = StoreNewBlock1(n32,b,3, blockVector[b].triangle2.ind,
triangles[n32].pointOfGravity, triangles[n32].v6);
DeleteLittleTriangles(place);
}
}
//triangle was already used
blockVector[b].triangle1.marker2 = -3;
blockVector[b].triangle2.marker2 = -3;
triangles[blockVector[b].triangle1.ind].marker2 = -3;
triangles[blockVector[b].triangle2.ind].marker2 = -3;
}
if(numberOfChoosenOption == 5)
{
StoreBlockOutOfTwoLonelyTriangles();
}
//deletes all the double blocks
DeleteDoubleBlocks();
//stores the edges of the new blocks into the newBlockVector
StoreEdgesInNewBlockVector();
//stores the triangles in the newBlockVector
StoreTrianglesInNewBlock();
//stores the ids of the triangles which are lying at the
//border of the final mesh
StoreIdOfBorderTriangles();
}
```

In der Funktion wird der `blockVector` durchlaufen, in dem alle basic blocks gespeichert sind. Zu jedem basic block gehören zwei Dreiecke. Vom ersten dieser Dreiecke werden die Nachbardreiecke überprüft. Folgende Bedingungen müssen gelten, damit von dem ersten Dreieck des basic blocks und einem Nachbardreieck ein neuer basic block gebildet werden kann:

1. der Nachbar darf nicht das zweite Dreieck des basic blocks sein
2. der Nachbar muss vorhanden sein, das Dreieck des basic blocks ist demnach kein Randdreieck
3. mit dem Nachbardreieck soll noch kein neuer basic block gebildet worden sein

Sind mit dem ersten Dreieck des basic blocks alle neuen basic blocks gebildet worden, wird das Gleiche mit dem zweiten Dreieck des basic blocks durchgeführt. Zusätzlich muss für beide Dreiecke des basic blocks überprüft werden, ob die Nachbardreiecke zu einem basic block oder zu einzelnen Dreiecken gehören, um zu entscheiden welche Punkte für den neuen basic block abgespeichert werden müssen.

Hat der Benutzer die Option gewählt konkave basic blocks wie zwei einzelne Dreiecke zu behandeln, wird zusätzlich die Funktion `StoreBlockOutOfTwoLonelyTriangles` aufgerufen, die neue basic blocks aus nebeneinanderliegenden Dreiecken bildet. Hierbei muss beachtet werden, dass neben einem konkaven basic block, der aus zwei einzelnen Dreiecken besteht, noch ein einzelnes Dreieck liegen könnte mit dem ein neuer basic block gebildet werden muss. Um ganz sicher zu gehen, dass am Schluss in dem Netz keine Blocks doppelt vorhanden sind, wird zusätzlich eine Funktion aufgerufen, die doppelte Blocks löscht.

4 GUI

Das Kapitel GUI beinhaltet eine Gebrauchsanleitung für den Umgang mit dem Virtual Tailor. Die Oberfläche, die dem Benutzer zum Zeichnen seiner Klei-

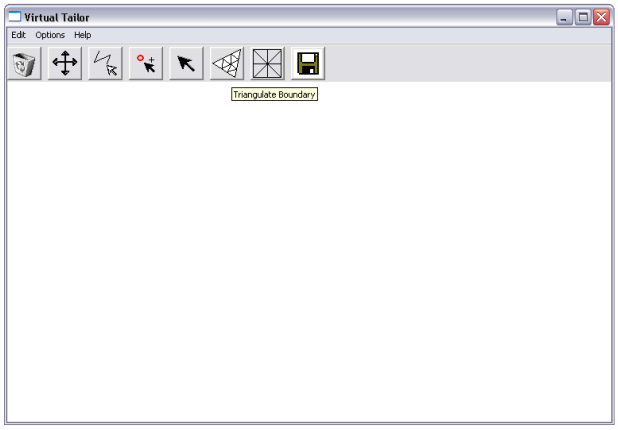

Abbildung 57: einfache Benutzeroberfläche

dungsstücke zur Verfügung steht, ist einfach und übersichtlich gehalten. Im oberen Bereich befinden sich die Buttons zur Werkzeugauswahl und darunter ist die Zeichenfläche. Es können einfache Strichzeichnungen angefertigt werden, wobei das Kleidungsstück auch mit Löchern versehen werden kann. Um eventuelle Verwechslungen oder Missverständnisse bezüglich des Gebrauchs der Buttons zu vermeiden, bietet das Programm die sogenannten Tooltipps. Somit hat der Benutzer im Zweifelsfall einen textuellen Hinweis, welche Aktion beim Drücken des entsprechenden Buttons ausgelöst wird. Im Übrigen sind alle Funktionen, die über die Buttons erreicht werden können, ebenfalls im Menü unter dem Punkt „Edit“ zu finden.

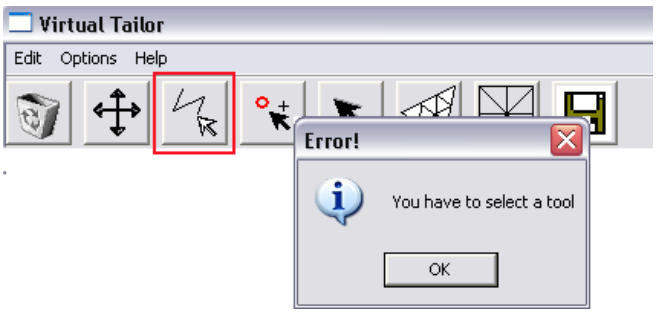

Abbildung 58: Zeichentool

Bei Programmstart ist per default das Zeichentool angewählt und der Benutzer kann anfangen zu zeichnen. Ist jedoch zwischenzeitlich einer der anderen Buttons gedrückt worden, muss das Zeichentool erneut angewählt werden. Vergisst der Benutzer dies, wird eine Warnung angezeigt.

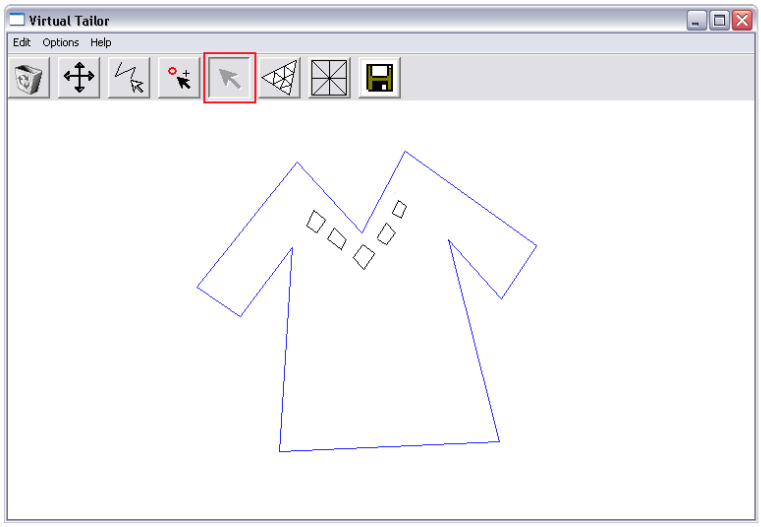

Abbildung 59: Pfeilwerkzeug

Zeichnen kann der Benutzer durch einfaches Klicken mit der Maus innnerhalb der vorgegebenen Zeichenfläche. Um ein Polygon zu schließen, klickt man mit der rechten Maustaste. In diesem Beispiel erkennt der Virtual Tailor, dass es sich um sechs separate Polygone handelt. Hat der Benutzer einen ersten groben Entwurf seines Kleidungsstückes gezeichnet, kann dieser noch weiter verfeinert werden. Dazu stehen die Optionen zum Einfügen und zum Verschieben von Punkten zur Verfügung. Bevor z.B. Punkte verschoben oder hinzugefügt werden können, muss eines der Polygone angewählt werden, da diese nur einzeln bearbeitet werden können. Mit dem im Bild markierten Pfeilwerkzeug kann man ein Polygon auswählen. Ein ausgewähltes Polygon wird blau dargestellt, wie im oberen Bild zu sehen ist.

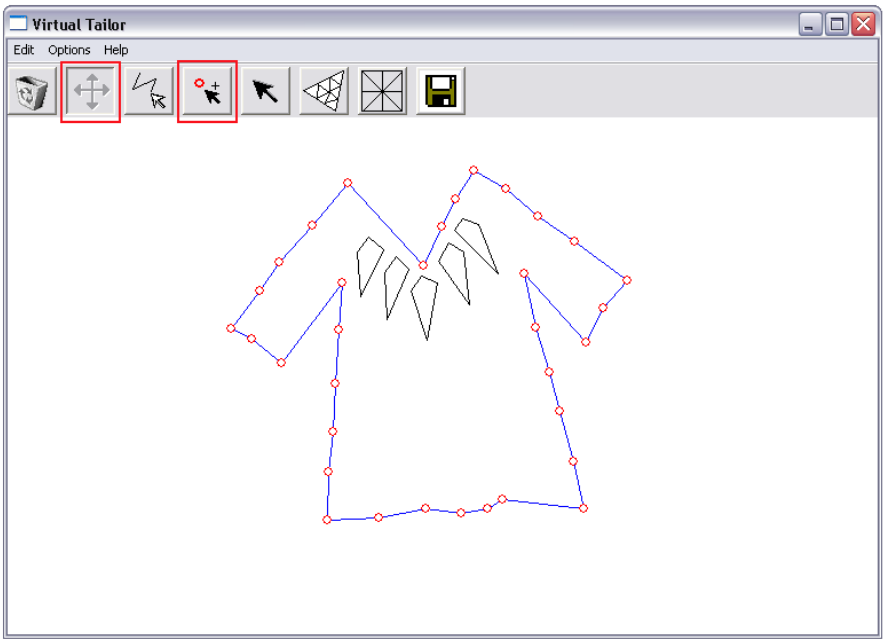

Abbildung 60: Verschieben von Punkten

Sobald das Werkzeug zum Bewegen oder das zum Erzeugen neuer Punkte angewählt wird, werden die Punkte des zuvor gewählten Polygons rot markiert um dem Benutzer das „Anfassen“ zu erleichtern. Kommt man im Verschiebemodus in die Nähe eines Punktes, ändert sich der Cursor in das Verschiebe-Symbol, das sich auch auf dem Button befindet und zeigt somit eine mögliche Aktion an. Ist man im Hinzufügenmodus, werden nicht nur Punkte, sondern auch Linien detektiert und der Cursor im Nahbereich einer Linie angepasst. Im Bild ist das Kleidungsstück nach der Bearbeitung zu sehen. Der Benutzer hat die Form durch Verschieben der Punkte geändert und mehrere Punkte hinzugefügt.

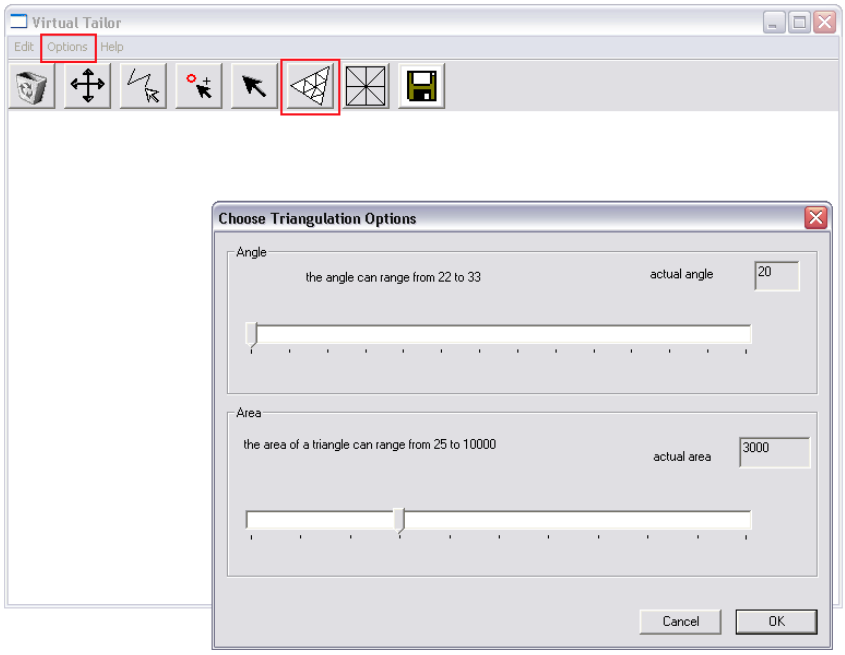

Abbildung 61: Optionsdialog

Ist der Benutzer mit der Form seiner Polygone zufrieden, kann trianguliert werden. Dazu geht er entweder direkt über den Triangulierungs-Button (im Bild markiert), oder er nutzt die Möglichkeit vorher noch bestimmte Bedingungen für die Triangulierung festzulegen. Dazu ruft er im Menü unter dem Punkt „Options“ die Funktion „change constraints“ auf und erhält einen Dialog. Der Dialog enthält zwei Schieber mit denen Winkel und Größe der Dreiecke eingestellt werden können. Mit dem oberen Schieber wird die minimale Größe des Winkels in den zu produzierenden Dreiecken eingestellt. Mit dem unteren Schieber kann die Anzahl der Quadratpixel, die ein Dreieck maximal enthalten darf eingestellt. Welche Werte gerade ausgewählt sind, lässt sich in einem Anzeigefeld ablesen. Beim Klick auf den Button „OK“ wird sofort trianguliert, da davon ausgegangen wird, dass der Benutzer dies sowieso als nächstes tun würde.

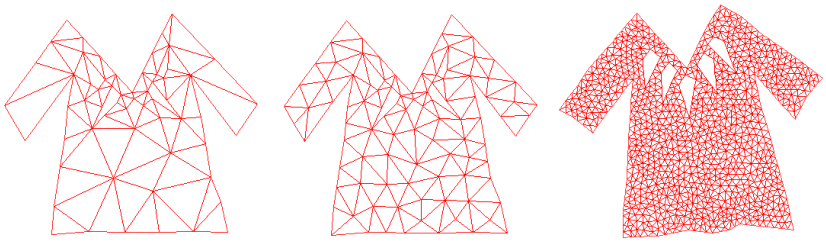

Abbildung 62: Triangulierung eines Kleidungsstückes in drei verschiedenen Granularitätsstufen. Dabei sind für die Größe der Dreiecke die Werte 3000, 1000 und 100 (von links nach rechts) eingestellt worden.

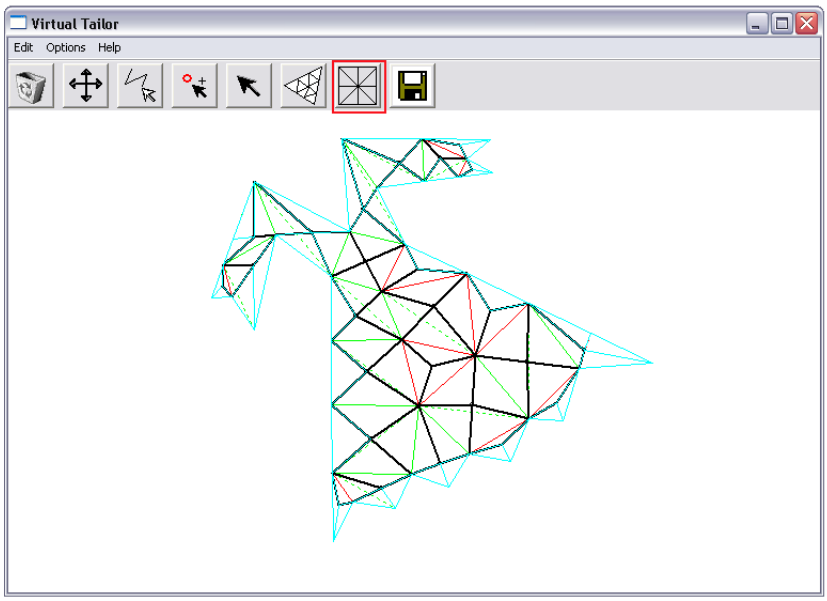

Abbildung 63: 4-8 mesh

Nach der Triangulierung kann das 4-8 mesh erstellt werden. Auf dem entsprechenden Button ist ein kleines 4-8 mesh dargestellt. Die vom Programm berechneten Basic Blocks sind in schwarz dargestellt. Bei den roten und grünen Linien handelt es sich um das intermediate mesh und die blauen Linien kennzeichnen die äußeren Dreiecke, die nicht für die Basic Block-Bildung verwendet werden. Die genaue Erklärung zum Aufbau kann im Kapitel „Implementierung“ nachgelesen werden.

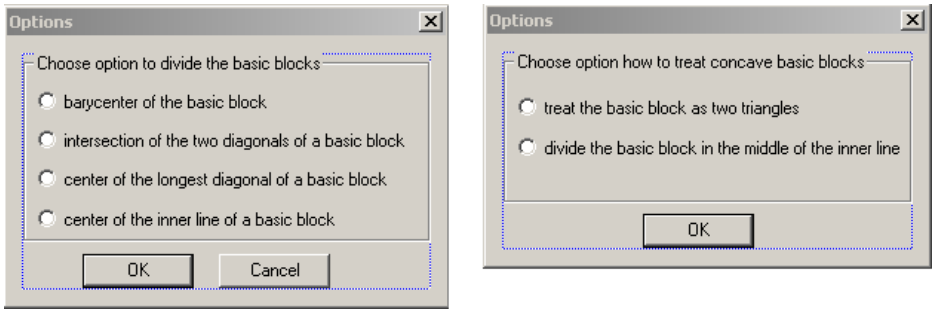

Abbildung 64: Optionsdialoge

Wurde der „4-8 mesh-Button“ gedrückt, müssen Angaben gemacht werden, wie bei der Erstellung des 4-8 meshes vorgegangen werden soll. Der erste Dialog, der erscheint, legt fest in welcher Weise die Vierecke aus dem „intermediate mesh“ geteilt werden sollen. Hat man seine Wahl getroffen, erscheint ein zweiter Dialog, in dem nach der Behandlung der eventuell auftretenden konkaven Blöcke gefragt wird. Dabei stehen zur Auswahl, einmal den Block nicht als Block zu behandeln, sondern als zwei Dreiecke, oder den Block in der Mitte der inneren Kante zu teilen.

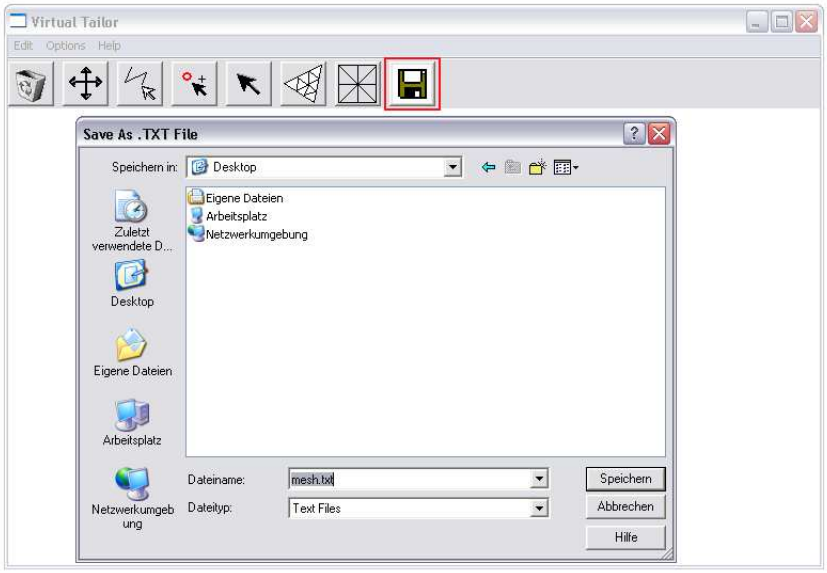

Abbildung 65: Beim Klick auf den Speicher-Button erscheint der dem Benutzer bekannte Speicherdialog, in dem Speicherort und Name der Datei ausgewählt werden können.

5 Fazit und Ausblick

Im Virtual Tailor kann der Benutzer einfache Stoffstücke zeichnen. Zusätzlich besteht die Möglichkeit, die gezeichneten Objekte auszuwählen, Punkte der Objekte zu verschieben oder die Objekte zu löschen. Vom Benutzer gezeichnete Stoffstücke können trianguliert und in 4-8 Netze umgewandelt werden. Zwischendurch kann sowohl die von Triangle erzeugte Triangulierung, als auch das fertige 4-8 Netz in einer Datei abgespeichert werden, die vom Kleidungssimulator, der von dem Betreuer Fernando Birra von der Universidade Nova de Lisboa implementiert wurde, verarbeitet werden kann.

Zu Beginn war zusätzlich geplant, mehrere Objekte miteinander „vernähen“ zu können, dieses wurde aber aus Zeitgründen nicht mehr implementiert. Darin besteht eine Möglichkeit zu Erweiterung des Programms. Weitere Funktionen des Programms könnten darin bestehen, gezeichnete Objekte verschieben oder rotieren lassen zu können. Eine weitere Möglichkeit zur Erweiterung wäre, das in der Dokumentation beschriebene Refinement von Luiz Velho zu implementieren, so dass das gezeichnete Kleidungsnetz adaptiv verfeinert werden kann.

Schwierigkeiten bei der Implementation ergaben sich durch die Verwendung der Bibliothek Triangle, da während unserer Arbeit eine neue Version veröffentlicht wurde, die dazugehörige Dokumentation aber nicht vorhanden war.

6 Literaturverzeichnis

Literatur

[Tri04] *Jonathan Richard Shewchuk: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. URL:* http://www-2.cs.cmu.edu/~quake/triangle.html Stand: Dezember 2004

[Microsoft04] http://msdn.microsoft.com/library/ Stand: Dezember 2004

[Kol04] *David Scherfgen: 2D-Kollisionserkennung.* URL: http://www.scherfgen-software.net/index.php?action=tutorials&topic=collision2d_1 Stand: Dezember 2004

[Inc04] *Jens Apel: Der Incremental Algorithmus.* URL: http://www.stud.tu-ilmenau.de/~japel/vortrag/gdv/increm.htm Stand: Dezember 2004

[MicBur04] *Michael Burghard: Incremental-Algorithmus.* URL: http://www.michael-burghardt.de/diss/node83.html Stand: Dezember 2004

[MicBurg04] *Michael Burghard: Rupperts Algorithmus.* URL: http://www.michael-burghardt.de/diss/node87.html#2739 Stand: Dezember 2004

[Allan04] *Allan Odgaard, and Benny Kjær Nielsen: A visual implementation of Fortune's Voronoi algorithm. Mai 2000.* URL: http://www.diku.dk/hjemmesider/studerende/duff/Fortune/ Stand: Dezember 2004

[Pro02] Einführung in die Win32-Programmierung mit C. 2002. URL: http://www.pronix.de/modules/C/win32/ Stand: Dezember 2004

[Forger04] *theForger's Windows API Programming Tutorial.1999.* URL: http://www.fortunecity.com/rivendell/final/355/files/Forgers.htm Stand: Dezember 2004

[BroMile04] *Brook Miles: App Part 3: Tool and Status bars. 1998-2003.* URL: http://www.winprog.org/tutorial/app_three.html Stand: Dezember 2004

[Wilh04] *Marc Wilhelm: Generierung von Delaunay Triangulationen. 2000.* URL: http://www.uni-stuttgart.de/iv-kib/generic/download/diplom/mwilhelm/Diplomtext.pdf Stand: Dezember 2004

[mapv04] http://mapviewer.skynet.ie/docs/Voronoi_Diagram_Notes_1.pdf Stand: Dezember 2004

[sieg04] http://www.stud.uni-siegen.de/fsr6/scripte/algorithmische-geometrie_schnell_ws01_02.pdf Stand: Dezember 2004

[zimmer04] *Henrik Zimmer: Ausgewählte Kapitel der Computergrafik. Voronoi and Delaunay Techniques.* 2004. URL: http://www-i8.informatik.rwth-aachen.de/teaching/ss04/proseminar/downloads/reports.pdf Stand: Dezember 2004

[Ruppert04] *Jim Ruppert: A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation.* http://graphics.stanford.edu/~guibas/GeomSem/99winter/RNR-94-002.pdf Stand: Dezember 2004

[Wellstein04] *Prof. Dr. Hartmut Wellstein: Schwerpunkte. 2000.* URL: http://www.uni-flensburg.de/mathe/zero/veranst/elemgeom/schwerpunkte/schwerpunkte.html Stand: Dezember 2004

[Tri04] Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. Applied Computational Geometry (Philadelphia, Pennsylvania), pages 124-133, ACM, May 1996. Abrufbar im Internet. URL: http://www-2.cs.cmu.edu/~quake/tripaper/triangle0.html Stand: Dezember 2004

[OlBun] Oliver Bunsen: Animationsorientierte Optimierung von Polygonen. 1996.URL: www.khm.de/~actor/publications/bunsen/anim-meshing.pdf Stand: Januar 2005

[Cottbus] Informatik TU Cottbus www-gs.informatik.tu-cottbus.de/~wwwgs/cg-v08c.pdf Stand: Januar 2005

[KetHof] Ketill Gunnarsson, Johannes Hofmann: Triangulierung von Polygonen. 2003 URL: www.inf.fu-berlin.de/lehre/ SS03/alg-geometrie/script-abgabe.doc Stand: Januar 2005