Interaktive Visualisierung von Algorithmen für Polygonzerlegungen

Diplomarbeit im Fach Informatik,
Institut für Informatik
der Ludwig-Maximilians-Universität München
von

Galina Stancheva

Beginn der Arbeit:	11.11.2002.
Abgabe der Arbeit:	11.05.2003.
Betreuer:	Prof. Dr. Hans Jürgen Ohlbach

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 9.5.2003 Galina Stancheva

Zusammenfassung

Die vorliegende Diplomarbeit beschäftigt sich mit der Visualisierung von Algorithmen aus der algorithmischen Geometrie (computational geometry) mit der Hilfe von interaktiven Algorithmus–Animationen. Dafür wurde das JAVA–Applet " *Visualisierung von Algorithmen für Polygon–Zerlegungen*" entwickelt. Es wird die Anwendung der Algorithmen-Animation als Lern- und Lehrmittel in elektronischen Unterrichten über algorithmische Geometrie präsentiert. Die untersuchten und visualisierten Algorithmen partitionieren die Fläche eines allgemeinen Polygons in einfachere Teile wie Dreiecke, Trapeze, monotone, unimonotone und konvexe Polygone.

Abstract

The topic of the diploma thesis is the visualization of algorithms in Computational Geometry via interactive algorithm-animation. To this end is developed the JAVA–Applet "Polygon–Partitioning–Visualizer". It presents animated algorithms for teaching and learning computational geometry in an electronic classroom. The Algorithms visualized here partition the area of a general polygon in simpler subpolygons such as triangles, trapeziums, monotone, unimonotone and convex polygons.

Inhaltsverzeichnis

Einführung

Die vorliegende Diplomarbeit beschäftigt sich mit der Visualisierung von Algorithmen aus der algorithmischen Geometrie (computational geometry) mit der Hilfe von interaktiven Algorithmus–Animationen. Dafür wurde das JAVA–Applet "*Visualisierung von Algorithmen für Polygon–Zerlegungen*" entwickelt, das die wichtigen Algorithmen aus einem Kapitel der algorithmischen Geometrie visualisiert. Das Applet soll die gewünschten Merkmale eines effektiven Algorithmen–Animations–Systems aus der Sicht des Benutzers (und nicht des Programmierers) erfüllen.

Die Algorithmen–Animation oder generell die Algorithmen–Visualisierung ist ein Teil der Software–Visualisierung, die eine zunehmende Rolle in vielen Bereichen spielt. Sie findet eine besonders wichtige Anwendung im Bereich der algorithmischen Geometrie, in welcher man sich oft komplizierte Szenen mit mehreren geometrischen Objekten vorstellen muss. In dieser Arbeit konzentriere ich mich auf die Anwendung der Algorithmen–Animation als Lern- oder Lehrmittel in elektronischen Unterrichten (electronic classroom), und insbesondere im elektronischen Unterricht über Algorithmen und Datenstrukturen in der Computational Geometry. Der virtuelle Unterricht existiert in den letzten Jahren in vielen Schulen und Universitäten. Das zu dieser Diplomarbeit gehörende Applet soll in einem virtuellen Unterricht in der algorithmischen Geometrie eingesetzt werden können.

Im ersten Kapitel beschäftige ich mich mit Fragen wie: Was ist ein Algorithmen–Animations–System? Wann und warum kann die Algorithmen–Animation eine effektive Rolle spielen? Welche Techniken kann man beim Erstellen der Visualisierungen benutzen?

Die in der Diplomarbeit untersuchten und im Applet visualisierten Algorithmen lösen die Probleme der Polygonzerlegungen, die ein grundlegendes Thema in der algorithmischen Geometrie sind. Es werden ausführlich wichtige Algorithmen präsentiert, die die Fläche eines Polygons in einfachere Teile wie Dreiecke, Trapeze, monotone, unimonotone und konvexe Teilpolygone zerlegen. Im zweiten Kapitel wird erläutert, in welchen Bereichen die Polygonzerlegungen Anwendung finden und was für Polygone und Zerlegungen existieren.

In den Kapiteln 3,4,5, und 7 werden die Algorithmen für die einzelnen Polygonzerlegungen diskutiert und analysiert. Sie sind im Applet mit interaktiven

Animationen versehen. Wie ich die Visualisierung der Algorithmen realisiert habe, wird am Ende jedes Kapitels beschrieben.

Da die Triangulierung eine Polygonzerlegung mit einer besonderen Rolle innerhalb der anderen Zerlegungen ist, habe ich in Kapitel 6 die bekannten zwei optimalen Algorithmen von Seidel [Sei91] und von Chazelle [Cha91] für lineare Triangulierung präsentiert, letzterer in Zusammenfassung. Diese Algorithmen werden teilweise auch visualisiert, weil sie auf den in anderen Kapiteln untersuchten und visualisierten Algorithmen basieren.

Kapitel 8 beschäftigt sich mit dem zur Diplomarbeit gehörenden JAVA–Applet bzw. Applikation. Hier beschreibe ich wie die Struktur des Applets aussieht bzw. wie die JAVA–Applikation entwickelt und implementiert wurde und wie sie in der Zukunft so erweitert werden kann, dass neue geometrische Algorithmen–Animationen damit visualisiert werden können. Beigefügt ist hier eine Benutzungsanleitung und Beschreibung von jeder Komponente der Benutzeroberfläche. Es ist empfehlenswert das Applet vor dem Lesen der schriftlichen Arbeit zu starten.

Kapitel 1

Algorithmen–Animation in der algorithmischen Geometrie

Die geometrischen Algorithmen finden in verschiedenen Bereichen der Computerwissenschaft ihre Anwendung. Die bekanntesten davon sind Computergraphik, Bildverarbeitung, Robotik, CAD/CAM (Computer Aided Design and Manufacturing), geographische Informationssysteme, kombinatorische Optimierung, Molekularmodellierung, shape reconstruction, computer vision, solid modeling. Das Design, die Analyse und die Optimierung geometrischer Algorithmen mit Hilfe von Computern werden im Bereich der algorithmischen Geometrie untersucht. Durch die Anforderungen in den Anwendungsbereichen hat sich diese Forschungsdisziplin in den letzten 2–3 Jahrzehnten stark entwickelt. Viele effiziente geometrische Methoden und Datenstrukturen wurden zur Lösung der Hauptprobleme Polygonzerlegungen, Konvexen–Hülle, Voronoi–Diagramme, Arrangements von Geraden und Ebenen, geometrische Suche, Wegeplanung entwickelt. Die algorithmische Geometrie hat einen festen Platz innerhalb des Gebiets "Algorithmen und Datenstrukturen" eingenommen.

In dieser Diplomarbeit werden Algorithmen zum Problem der Polygonzerlegungen untersucht. Im Kapitel 2 werden wir tiefer auf die Anwendungen der Polygonzerlegungen eingehen, um zu zeigen, warum sie untersucht und gelernt werden sollen.

Da die geometrischen Algorithmen sehr kompliziert und schwer begreifbar sind, spielt die Algorithmen–Animation für deren Vermittelung eine wichtige Rolle.

1.1 Was ist die Algorithmen–Animation?

Im Folgenden wird diskutiert, was die Algorithmen–Animation ist. Ist sie nutzbar und wofür? Warum spielt sie speziell in der algorithmischen Geometrie (computational geometry) eine wichtige Rolle?

Die Algorithmen–Animation ist ein Gebiet der Software–Visualisierung, das in den letzten Jahren stark an Bedeutung gewonnen hat. Sie wird als Unterstützung zur Beschreibung von komplexen Vorgängen eingesetzt und unterstützt das Lernen eines abstrakten Algorithmus. Sicherlich kann man einen Algorithmus ohne Animation lernen, indem man Text, Pseudocode oder Standbilder benutzt. Durch die Animationen lässt sich jedoch der zeitliche Ablauf eines Algorithmus direkt visualisieren, man kann den Algorithmus sozusagen “in Aktion” sehen und erleben. Sie sind kurze Filmsequenzen, die den Ablauf des Algorithmus durch die Veränderung der Daten und der Datenstruktur zeigen. Die Algorithmen–Animation wird schon seit 20 Jahren benutzt. Forscher, Algorithmen–Designer, Lehrer, Studenten und jeder, der Geometrie und Computation erforscht, kann sie brauchen. Sie ist ein grundlegendes Lehr- und Lernmittel im elektronischen Unterricht(electronic classrooms). Sie werden auch im Labor zum Erforschen des Lerneffekts und außerhalb der Vorlesungen oder Labors als Lernhilfe benutzt.

Die Algorithmen–Animation hat ihre Wurzeln im Lehrfilm[Ba81] (heutzutage noch benutzt) und der mathematischen Visualisierung und wurde beeinflusst von der Entwicklung in der wissenschaftlichen Datenvisualisierung. Die ersten Algorithmen–Animationen–Systeme (Balsa [BrSe85], BalsaII [Br88], Tango [Stasko90], Zeus [Br92], CAT[BrNa96]) wurden seit der Mitte der achtziger Jahre eingesetzt. Sie wurden einerseits für Entwurf und Entwicklung von Animationen der fundamentalen Algorithmen (Sortierungsalgorithmen, Datenstrukturen) implementiert und andererseits für die Interaktion und das Experimentieren mit den Animationen. Dann wurden spezialisierte Algorithmen–Animations–Systeme (Computational Geometry Workbench [EpsKa94], XYZ GeoBench [Schorn91], Mocha [BCLT96], GASP [TalDo95], GASPII [TalSchne98] für die Arbeit mit geometrischen Algorithmen entworfen. Mit dem Erscheinen der webbasierten ProgrammierspracheJAVA wurde die Algorithmen–Animation sehr verbessert. Sie war als laufendes Programm und nicht nur in statischen Bildern angeboten, sie wurde populär und leicht zugänglich. Heutzutage stehen im Internet eine große Zahl von Algorithmen–Animationen als JAVA–Applets zur Verfügung, die direkt in einem Webbrowser ausgeführt werden können.

Wie oben schon erwähnt, sind die geometrischen Algorithmen komplizierter als andere Algorithmen. Sie sind schwieriger zu entwerfen, zu programmieren, zu testen uns von Fehlern zu befreien. Der Grund dafür ist, dass man oft beim Entwerfen oder Verstehen eines geometrischen Algorithmus nicht nur die vom Algorithmus manipulierten geometrischen Objekte visualisieren muss, sondern auch den Prozess, der die geometrischen Objekte verändert. Die Algorithmen–Animation löst dieses Problem, da sie all dies zeigt.

Um einen Algorithmus zu präsentieren und zu erklären brauchen Wissenschaftler oder Lehrer ein Medium, um die Ideen und die Theorie hinter dem Algorithmus zu erklären und zu verbreiten. Obwohl die Forschungsdokumentationen diese Rolle zufriedenstellend gut spielen, können die Animationen schneller als sie Ideen vermitteln. Falls eine interaktive Animation als JAVA–Applet zum

Beispiel im Internet zur Verfügung steht, können die Nutzer selbst den Algorithmus leichter lernen und falls Internetzugang vorhanden ist, können sie dies auch unabhängig von Zeit und Raum.

Die Erstellung der Animation eines Algorithmus kann den Designer stimulieren, neue Variationen des Algorithmus zu finden. Tatsächlich die Visualisierung erklärt dem Autor seinen Algorithmus.

Ein geometrischer Algorithmus soll häufig auch für besondere und degenerierte Fälle funktionieren. Die Animation kann helfen, solche Fälle zu identifizieren.

Schließlich hilft die Animation beim Beseitigen von Programmierfehlern.

Die vom Computer generierten Bilder haben einen Vorteil vor den handgezeichneten Bilder: sie sind präziser; können leicht modifiziert werden; und speziell bei Bildern mit 3D–geometrichen Objekten, kann man diese unter jedem beliebigen Winkel betrachten.

Bis hierher kann man leicht den Eindruck bekommen, dass die Algorithmen–Animation einen positiven Lerneffekt haben muss. Ist das wirklich so? Es wurden viele Experimente zur Untersuchung dieses Lerneffekts durchgeführt (Beispiele: [Pa91],[StaBaL93], [Stasko98]). Nach der Zusammenfassung der Ergebnisse aller Experimente und Studien hat man aber unterschiedliche, teilweise widersprüchliche Aussagen über den Lerneffekt erhalten.

Nach [Stasko98] darf man einzelne Experimente nicht verallgemeinern, da in jedem Fall eine bestimmte Personengruppe einen bestimmten Algorithmus in einem bestimmten Kontext lernen sollte und immer ein bestimmtes Verständnis vom Lernen eines Algorithmus zugrunde gelegt wurde. Individuelle Experimente lassen sich also nicht leicht (wenn überhaupt) zusammenfassen. Sie können als Inspiration und Vorlage für immer neue Hypothesen und Experimente dienen und so schrittweise das Verständnis von Funktion und Nutzen der Algorithmen–Animation verbessern. [Stasko98] stellen auf Grundlage ihrer Studien drei Hypothesen über den Lerneffekt mit Hilfe von Algorithmen–Animationen vor:

1. Algorithmen–Animationen haben in jedem Fall einen pädagogischen Wert, auch wenn ihr Nutzen als Lernmittel minimal ist. Viele Studien zeigen den Spaß und das Interesse, also allgemein die Motivation zum Arbeiten mit den Animationen. “Auch wenn der Algorithmen–Animation nur diese Funktion zuteil kommt, sollte man sie einsetzen” [Stasko98].

2. Algorithmen–Animation in Form von passiven Videos haben keinen bemerkenswerten Effekt auf das Lernen eines völlig fremden Algorithmus. Um von einer Algorithmen–Animation zu profitieren, muss zuerst einmal die verwendete Visualisierung verstanden werden (außer natürlich bei einer sehr einfachen, intuitiven Visualisierung). Kennt man den Algorithmus noch überhaupt nicht, so kann kein Bezug zwischen aktueller Visualisierung und zugehöriger Algorithmenoperation hergestellt werden. Eine Erklärung der Visualisierung ermöglicht in jedem Fall einen schnellen Einstieg in die Animation, da die Interpretation der Visualisierung entfällt.

3. Entscheidend ist, dass die Lernenden aktiv mit den Animationen interagieren können, indem z.B. Eingabedaten gewählt werden oder die Animation angehalten und der nächste Schritt des Algorithmus vorhergesagt werden muss. Die Lernenden werden so aktiv an der Animation beteiligt, was sich auch positiv auf die schon erwähnte Motivation auswirkt.

Insgesamt meinen [Stasko 98] und viele andere Autoren, dass Algorithmen–Animationen ein wertvolles Lehrmittel sein können, falls die Visualisierung geeignet beschrieben wird und die Lernende aktiv beteiligt werden.

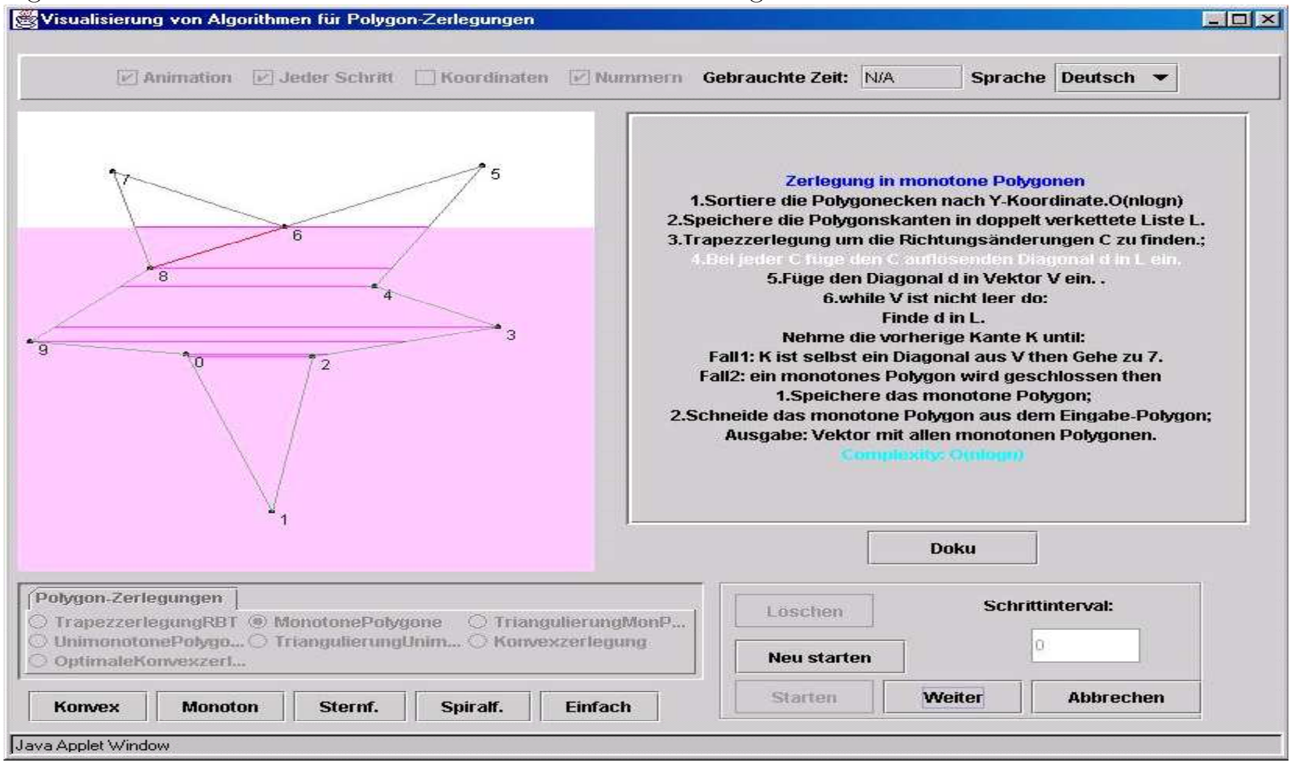

Bild1.1. Das zu dieser Diplomarbeit vorhandenes Applet ist ein Beispiel für interaktive Algorithmen–Animation.

Der Entwurf interaktiver Visualisierungen für den Einsatz in einer virtuellen Vorlesung benötigt besondere Sorgfalt. Solche Visualisierungen haben ihre eigenen besonderen Eigenschaften und Merkmale. Im folgenden werden die gewünschten Merkmale diskutiert.

1.2 Interaktive webbasierte Animation in einer virtuellen Vorlesung

In diesem Abschnitt konzentriere ich mich auf die Anwendbarkeit der interaktiven Algorithmen–Animation im Bereich der Lehre, was ein zentraler Aspekt der Algorithmen–Animation ist. Für sie werden aus didaktischer und software-ergonomischer Sicht wünschenswerte Merkmale angegeben und Beispiele aus dem zur Diplomarbeit gehörenden Applet gezeigt. Die gewünschte Merkmale helfen eine interaktive Animation nach didaktischen und Software–ergonomischen Kriterien zu bewerten.

Man kann drei Arten von interaktiven Algorithmen–Visualisierungen in einem virtuellen Unterricht nach ihren Zielen und Rahmenbedingungen unterscheiden:

1. Algorithmen–Animationen werden von den Studenten betrachtet und analysiert. Der Ablauf des Algorithmus ist fest vorgegeben. Der Student soll die Algorithmusschritte nicht voraussagen.

2. Algorithmen–Animationen werden zu Übungen eingesetzt und fordern von den Studenten, zuvor gesehene Schritte eines Algorithmus selbst auszuführen.

3. Algorithmen–Animationen werden für Algorithmen–Simulationen entwickelt. Sie erlauben dem Student, sich durch Experimentieren an die Arbeitsweise eines Algorithmus heranzutasten.

1.2.1 Interaktive Algorithmen–Animationen für das Lernen durch Betrachtung

Es wurden aus didaktischer und Software–ergonomischer Sicht einige wünschenswerte Merkmale für Animationen angegeben, die für die ***Lernform Betrachten*** entwickelt wurden. Bei dem Entwurf und der Implementierung des zu dieser Diplomarbeit gehörenden Applets und seiner interaktiven Animationen hatte ich als Ziel, möglichst viele von diesen zu erfüllen. Die gewünschten Merkmale:

Funktionale Struktur. Der Algorithmus wird in Funktionen aufgeteilt. Jede komplizierte Funktion wird in einer eigenen Animation erklärt. Bereits gelernte Funktionen können in einer Animation als Bausteine verwendet werden und werden nur noch als Schritt angezeigt.

Beispiel in dem Applet: Bei vier Polygonszerlegung–Algorithmen wird die Polygonzerlegung in Trapeze eingesetzt. Da der Trapezidierung–Algorithmus kein trivialer Algorithmus ist, habe ich für ihn eine eigene Animation und ein eigenes Kapitel vorgesehen.

Lehrtext. Mit der Animation ist ein Lehrtext verbunden, der mit Text und Bildern den Algorithmus erklärt.

Beispiel in dem Applet: Unter dem Pseudocode–Panel befindet sich der Doku–Button, mit dem der Benutzer das zu dem Algorithmus entsprechende Kapitel aus der Dokumentation öffnet.

Anzeige des Pseudocode. Zusätzlich zu den vom Algorithmus manipulierten Objekten (z.B. in der algorithmischen Geometrie Punkte, Liniensegmente, Polygone,etc.) und Datenstrukturen wird der Pseudocode des Algorithmus angezeigt. Die aktuelle Befehlszeile wird hervorgehoben. So kann der Benutzer den Pseudocode und die Aktionen auf den Objekten zueinander in Beziehung setzen.

Beispiel in dem Applet: Das Pseudocode–Fenster auf der rechten Seite.

Erklärung der Schritte. Die Schritte des Algorithmus werden textuell oder per Sprachausgabe kommentiert, damit der Lernende versteht, was der Algorithmus gerade tut.

Beispiel in dem Applet: Keine extra GUI–Komponente auf der Oberfläche. Die hervorgehobene Pseudocode–Zeile könnte diese Rolle spielen. In dem JAVA–Console wird gezeigt, welche Prozedur des Programms gerade läuft.

Interessante Eingabedaten. Viele Algorithmen zeigen bei bestimmten Eingabedaten ein besonderes Verhalten. Der Algorithmus für Zerlegung in monotone Polygone zeigt zum Beispiel bei konvexen Eingabepolygone ein anderes Laufzeitverhalten als bei konkaven Eingabepolygonen. Dem Benutzer werden verschiedene interessante Eingabedaten zur Auswahl gestellt.

Beispiel in dem Applet: Beim Drucken der Eingabe–Tasten werden interessante Polygone in das Draw–Fenster als Eingaben gezeichnet.

Kleine Datenmenge. Die Animation kann mit einer kleinen Zahl von Datenobjekten arbeiten oder anders gesagt, kann mit einfachen Eingabedaten effektiv arbeiten. Ein Beispiel: Die Animation soll durch ein Eingabepolygon mit möglichst wenig Kanten, die für den Zweck des Algorithmus genug sind, den Algorithmus zeigen können. Die Anzahl und die Position der Kanten ist gerade so gewählt, dass das Wesentliche am Algorithmus noch zu erkennen ist.

Wenige Schritte. Es werden nur so viele Wiederholungen von Schleifendurchläufen gezeigt, wie für Wahrnehmung und Verständnis nötig sind. Andernfalls wird der Lernende schnell gelangweilt und seine Aufmerksamkeitskapazität unnötig verbraucht.

Beispiel in dem Applet: Der Benutzer hat die Möglichkeit während des Ablaufs manche Algorithmusschritte schnell laufen lassen, im bestimmten Moment abzubrechen und so sie überspringen; dann weiter ab diesem bestimmten Algorithmusschritt bis zum Ende oder einem anderen Algorithmusschritt schrittweise mit der Weiter–Taste auszuführen.

Flexible Ablaufsteuerung. Die Animation kann ähnlich wie bei einem Videorecorder gesteuert werden: Einzelschritte vor und zurück, Abspielen, Pause, Stopp, Sprung zu Anfang und Ende. Aus didaktischer Sicht ist es besonders wichtig die Visualisierung in Einzelschritten vor und zurück schalten zu können. So hat der Benutzer bei jeden Schritt beliebig viel Zeit zu überlegen. Da normale Programme nicht rückwärts laufen können, stellt dies besondere Anforderung an die Software–Architektur der interaktiven Visualisierung.

Beispiel in dem Applet: Im Kontroll–Fenster stehen die Start-, Abbrechen-, Weiter-, Neustarten–Buttons zur Verfügung. Um die Animation–Geschwindigkeit zu ändern, kann der Benutzer das Warte–Intervall in dem Intervall–Textfeld eintragen. Über das Kontroll–Fenster kann der Benutzer die Animation unterbrechen; sie schnell oder langsam laufen lassen; die Algorithmus–Schritte selbst nach vorne steuern; jederzeit die Animation neu starten.

In dieser Version des Applets können die Animationen nicht rückwärts gesteuert und ausgeführt werden.

Interessante Ereignisse. Ein Schritt soll ein "interessantes Ereignis" des Algorithmus umfassen. Das kann z.B. eine Aktion des Algorithmus oder das Erreichen wichtiger Zwischenergebnisse sein. Das Konzept der "interesting events" wurde von Marc Brown entwickelt.[Br87]

Beispiel in dem Applet: In den Kapiteln, in denen die implementierte und visualisierte Algorithmen untersucht werden, ist ein Abschnitt über die Visualisierung des entsprechenden Algorithmus zu finden. Da diskutiere ich welche Ereignisse im entsprechenden Algorithmus ich interessant finde und wie ich sie visualisiert habe.

Fokussierung der Aufmerksamkeit. Der Blick des Lernenden wird auf die Teile der Darstellung gelenkt, die sich als nächstes verändern werden. Dazu kann der Autor der Animation diese Objekten blinken, ihre Größe oder(/und) ihre Farbe ändern lassen. So wird verhindert, dass der Betrachter wichtige Schritte nur als diffuse Bewegung wahrnimmt. Die Farbe–Technik gibt viele Möglichkeiten der Visualisierungsdarstellung. Ich führe dies weiter unten kurz aus.

Beispiel in dem Applet: Es werden vorallem Techniken der Änderung der Farben und der Form der geometrischen Objekten eingesetzt.

Weiche Übergänge. Veränderungen auf den Datenstrukturen werden möglichst mit weichen, fließenden Bewegungen gezeigt.

Beispiel in dem Applet: In dieser Version des Applets wird wegen des Zeitaufwands keine Ansicht des Verhaltens und Veränderungen der benutzten Datenstrukturen gezeigt.

Darstellung mehrerer Algorithmen. Werden mehrere Algorithmen gleichzeitig animiert, kann der Benutzer die unterschiedlichen Arbeitsweisen direkt miteinander vergleichen und analysieren.

Beispiel in dem Applet: Der Benutzer kann mehrere von den sieben Algorithmen in verschiedenen Browsern gleichzeitig laufen lassen.

Es gibt wenige Algorithmen–Animation-Systeme, die alle obengenannte Prinzipien umsetzen. Das zu dieser Diplomarbeit gehörende Applet in seiner jetzigen Version erfüllt auch nicht alle gewünschten Merkmale. Nach einer Zusammenfassung kann man aber feststellen, dass fast alle implementiert wurden und zwar:

- Funktionale Struktur
- Lehrtext
- Anzeige des Psoudocode
- Erklärung der Schritte (teilweise)
- Interessante Eingabedaten
- Kleine Datenmenge
- Wenige Schritte
- Flexible Ablaufsteuerung
- Interessane Ereignisse
- Fokussierung der Aufmerksamkeit
- Darstellung mehrerer Algorithmen

1.2.2 Interaktive Algorithmen–Animationen für das Lernen durch Übung

Bis jetzt wurden die spezifischen Merkmale für interaktive Algorithmen–Animationen als Lernform "Betrachten" aufgelistet. Nachdem ein Algorithmus durch Betrachten gelernt worden ist, sollte das Wissen durch Übung überprüft und vertieft werden. Die oben diskutierten Merkmale gelten auch für die interaktiven ***Algorithmen–Animationen für Übungen*** bis auf den Punkt "Anzeigen des Pseudocode". Zusätzlich sollten noch die folgenden Merkmale vorhanden sein:

Kein Pseudocode: Da der Student bei der Übung mehrfach den nächsten Schritt des Algorithmus eigenständig voraussagen soll, wird kein Pseudocode angezeigt. Falls eine Animation für das Lernen durch Betrachten und Üben geeignet sein soll, soll der Pseudocode–Fenster zugemacht und aufgemacht werden können.

GUI nutzen. Die Möglichkeiten einer grafischen Benutzeroberfläche (GUI) werden soweit möglich und sinnvoll für die Interaktion genutzt. Der Benutzer soll Objekten in der Graphik der Visualisierung direkt anwählen können (z.B mit der Maus anklicken). Ein Schritt eines Algorithmus besteht oft im Aufruf einer Funktion. Die Funktionen können durch Tasten dargestellt werden. Der Schüler ruft eine Funktion auf, indem er erst die Eingabe–Datenobjekte und die aktuellen Parameter in der Graphik anklickt und dann die Funkiontaste druckt.

Beispiel in dem Applet: Der Student kann das Eingabe–Polygon in dem Draw–Fenster durch Mausklicken zeichnen. Er soll in dem Option–Panel und dem Algorithmen–Panel die Optionen auswählen.

Bedienung vereinfachen. Es ist einfach erkennbar, welche Funktionen angeboten und welche Objekten anwählbar sind. Der Student muss also nur die Objekten und Funktionen auswählen.

Lösung erschweren. Es werden mehr Funktionen angeboten als tatsächlich benötigt werden. Auch sind möglichst alle Objekten der Visualisierung anwählbar. Dadurch wird es erschwert, den nächsten Schritt zu erraten.

Fehlermeldungen und mehrstufige Hilfe. Macht der Benutzer eine falsche Eingabe, ist es sinnvoll zu begründen, warum diese keinen Sinn macht. Eine mehrstufige Hilfe sollte aufrufbar sein, die einen Hinweis gibt, was zu tun ist und als nächste Stufe den Schritt direkt angibt.

1.2.3 Lernen durch Algorithmen–Simulationen

Die dritte Form des Lernens mit Algorithmen–Animationen in der virtuellen Vorlesung sind die ***Algorithmen–Simulationen***, die ein freies Experimentieren mit (besonders geometrischen) Objekten und Funktionen erlauben. Sie sollen dem Lernende ermöglichen den Algorithmus "nachzuentdecken". D.h., der Algorithmus wird dem Student nicht als fertiges Werk vermittelt, sondern der Student soll einen Teil der Arbeit der Entwicklung und Entdeckung wiederholen und wird so in die Rolle des Algorithmenentwicklers versetzt. Zusätzlich zu den oben genannten Kriterien für die Animationen für Übungen sollen die Algorithmen–Simulationen auch die folgende Merkmale erfüllen:

Inhaltsbezogene Fehlermeldungen. Versucht der Benutzer eine Funktion aufzurufen, deren Start–Invarianten nicht erfüllt sind oder wird dadurch eine Arbeits–Invariante verletzt, so muss die interaktive Visualisierung dies mit einer Fehlermeldung abweisen.

Fortschrittsanzeiger. Der Benutzer kann erkennen, wie nah er dem Ziel gekommen ist.

Undo. Es kann sein, dass der Benutzer Schritte tätigt, die zwar die Invarianten nicht verletzen, aber nicht zum Ziel führen. Über einen Undo–Mechanismus kann er einzelne Schritte zurücknehmen.

Das verfügbare Applet ist kein gutes Beispiel für das Lernen durch Übung mit Algorithmen–Animation und Algorithmen–Simulationen, da es nicht konkret für diese Ziele entwickelt wurde. Es kann aber trotzdem für das Üben und

Experimentieren verwendet werden, da es teilweise manche der oben genannten Wünsche erfüllt.

Es ist anzumerken, dass das Applet sich leicht erweitern lässt, so dass es konkret für das Üben und das Simulieren der Algorithmen wichtige Merkmale anbieten kann. Wie das Applet erweitert werden kann, diskutiere ich in dem speziell dem Applet gewidmeten Kapitel.

1.3 Techniken der Visualisierungsdarstellung

Bei der Software–Visualisierung ist der Animateur mit Problemen konfrontiert, mit denen die klassische Grafik–Designer nicht konfrontiert werden. Dazu gehören:

- Bildschirme sind kleiner als Papier und haben unterschiedlichen Größen;
- Die Auflösung der Bildschirme ist der von Papier deutlich unterlegen;
- Bildschirm–Anzeigen sind dynamisch, Papier nicht;
- Mehrfach Anzeigen müssen aufeinander bezogen und folgerichtig sein, Graphik–Design bezieht sich auf ein einzelnes Bild.
- Speziell dazu unterscheidet sich die Algorithmen–Animation von der konventionellen Animation, indem die Sequenz von Bildern nicht fixiert ist. Jedes Bild zeigt den Zustand des Algorithmus zu einem Zeitpunkt. Da die Eingabe–Daten unterschiedlich sein können, kann das Verhalten des Algorithmus entsprechend der Eingabe auch unterschiedlich sein, also wird auch die Animation variieren.

Bis jetzt haben wir die Eigenschaften einer effektiven Algorithmen–Animation in einem "Electronic classroom" gesehen. Diese könnten als Pflichtenheft für die Entwickler von Animationen betrachtet werden. Manche der Merkmale werden in der Literatur ([Stasko97], [Br91n]) als grundlegende Techniken zum Animieren von Algorithmen (nicht nur für die Lehre) betrachtet. Diese sind die oben geklärte Techniken der weichen Übergänge, der Darstellung mehreren Algorithmen, die Auswahl der Eingabedaten. Andere fundamentale Techniken bei der Visualisierungsdarstellung sind die Technik vom Anzeigen mehrerer Ansichten, die Technik des Zustandsanzeigen, des statischen Anzeigen, die Farbe–Technik und die Audio–Technik, die nun kurz erklärt werden.

Beim Anzeigen mehrerer Ansichten wird ein Algorithmus in mehreren verschiedenen Ansichten(engl. views) illustriert, wobei jede Ansicht nur wenige Aspekte eines Algorithmus präsentiert. Z.B kann bei einem Polygonzerlegung–Algorithmus in einem Fenster die Änderungen in dem Polygon gezeigt werden und in einem anderen Fenster die Änderungen des Zustands des benutzten Red–Black–Baums.

Jede Anzeige ist isoliert leicht zu verstehen. Die Komposition mehreren verschiedener Anzeigen ist informativer als die Summe ihrer individuelle Beiträge.

Bei der Technik der Zustandsanzeigen werden die Zustandsveränderungen an der Datenstruktur des Algorithmus durch Änderungen ihrer graphischen Repräsentation sichtbar gemacht. Die Zustandsanzeigen zeigen das dynamische Verhalten von Algorithmen.Die Technik des statischen Anzeigen bietet die Möglichkeit "Snapshots" von mehreren Animationsläufen zu generieren. So kann der Benutzer seine Lerngeschwindigkeit selbst bestimmen.

Die Farbe–Technik spielt eine besondere Rolle: Die Farbe kann große Mengen von Informationen effektiv übertragen. Sie wird in fünf klar voneinander getrennten Arten eingesetzt: Verschlüsseln des Zustands von Datenstrukturen, Hervorheben von Aktivität oder Objekt, Verbinden von Ansichten(Views), Betonen von Mustern, Anzeigen der historischen Daten. Die Farbe kann Indikator des Zustands eines Algorithmus sein. Informationen werden sowohl über die Farbe als auch über die Form eines Objekts übertragen. Es ist zu bemerken, dass für die Darstellung eines Objekts in einer eigenen Farbe weniger Pixel benötigt werden als für die Darstellung in einer eigener Form. Somit können mehrere Objekten dargestellt werden.

Viele Animationen zeigen zeitweilig eine Region mit einer transparenten kontrastreichen Farbe um die Aufmerksamkeit auf das Gebiet zu lenken. Die zur Hervorhebung benutzte Farbe ist transparent und interferiert nicht mit auf dem Bildschirm dargestellten Datenelementen.

Unterschiedliche Repräsentationen logisch verwandter Objekten oder verschiedene Aspekte der gleichen Datenstruktur werden mit gleicher Farbe gezeigt. Dadurch entsteht ein integriertes harmonisches Bild.

Die Darstellung der "Ablaufgeschichte" kann mit der Hilfe der Farbe erreicht werden. Z.B. kann man in unserem Applet bei dem Trapezidierung–Algorithmus durch eine eigene transparente Farbe zeigen, bis wo die Sweep–Linie die Objekte abgedeckt hat. Es kann auch eine Ordnung von Farben nach dem Farbton erfolgen, um eine lineare Zeitfolge zu repräsentieren.

Kapitel 2

Polygone und Polygonzerlegungen

Bevor ich mit der Präsentation der Einzelnen, in den nächsten Kapiteln untersuchten, Algorithmen für Polygonzerlegungen anfange, finde ich es sinnvoll, mich kurz mit den folgenden Fragen zu beschäftigen: Wozu und wann sind die Polygonzerlegungen anwendbar? Was für Polygone und Polygonzerlegungen gibt es überhaupt? Wie kann man sie klassifizieren?

Polygone gehören zu den fundamentalen Basiskomponenten in der geometrischen Modellierung. Sie sind verwendbar für die Darstellung von vielen unterschiedlichen Formen und Gestalten in vielen der am Anfang erwähnten Anwendungsgebiete der algorithmischen Geometrie. Mit "*einfachem Polygon*" meine ich ein Gebiet der Ebene, das von einem einfachen Zyklus von geraden Liniensegmenten geschlossen wird. Mit einfachem Zyklus wird gemeint, dass sich keine zwei nicht benachbarte Liniensegmente schneiden und zwei benachbarte schneiden sich nur in deren gemeinsamen Endpunkt. In dieser Diplomarbeit wird der Begriff "Polygon" äquivalent zu dem Begriff "einfaches Polygon" verwendet. Die *Polygonzerlegung* eines einfachen Polygons stellt das Polygon als Vereinigung von Einzelteilen dar, die auch einfache Polygone sind.

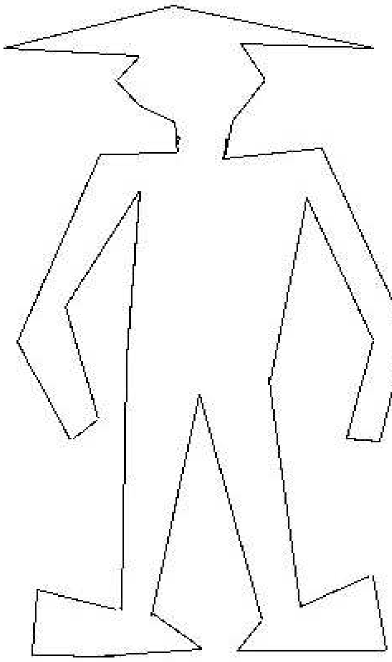

Bild2.1. Ein einfaches Polygon[1].

Es ist zu bemerken, dass die Kennzeichnung "einfach" in der Definition des einfachen Polygons eine *topologische* Eigenschaft angibt, die das "nicht–selbst–durchschneidend" beschreibt. Es soll nicht als "unkompliziertes Polygon" verstanden werden. Eigentlich sind die einfachen Polygone die kompliziertesten unter den Polygonen, die topologisch äquivalent zu einer Scheibe sind.

Die Polygonzerlegungen finden viele theoretische und praktische Anwendungen innerhalb und außerhalb der algorithmischen Geometrie. Die wichtigsten von ihnen werden kurz im Folgenden diskutiert.

Das Zerlegen der Fläche eines Polygons in einfachere Teile ist ein wichtiges Werkzeug in der Mustererkennung (pattern recognition). Techniken der Mustererkennung extrahieren Informationen aus einem Objekt, um das Objekt zu beschreiben, zu identifizieren und zu klassifizieren. Eine bekannte Strategie für die Erkennung eines allgemeinen vieleckigen (polygonal) Objekts ist, das Objekt in einfacheren Komponente zu zerlegen und dann die Komponenten und deren Zusammenhänge zu identifizieren. Diese Information wird dann verwendet, um die Gestalt des Objekts zu bestimmen.([FePav75],[Pav77])

Die Polygonzerlegungen werden bei Probleme des VLSI–Design eingesetzt. VLSI (very large scale integration) ist der aktuelle Stand der Miniaturisierung des Computermikrochips (computer microchip miniaturization), der sich auf solche Mikrochips bezieht, die Hunderttausende von Transistoren enthalten. Der Aufbau (das Layout) wird als Polygon dargestellt. Eine Methode für die Vorbereitung

[1] In meisten Abbildungen ist die Bildschirmauflösung sichtbar, da sie Ausschnitte vom Zeichen-Fenster der Applikation sind

der Vorlagen für die Lithographie mit Elektronenstrahl ist die Zerlegung dieser Polygonen in grundlegende geometrischen Figuren.([AAI86],[FSS84])

In dem Prozess der Teilung unterschiedlicher Regionen (engl. routing region) in Kanäle werden die Polygonzerlegungen auch benutzt.[LPRS82]

In der algorithmischen Geometrie sind die Algorithmen für Probleme mit allgemeinen Polygonen oft komplizierter als diese mit beschränkten Typen von Polygonen wie konvexe oder monotone Polygone. Bekannte Beispiele sind: das Problem, ob ein Punkt innerhalb eines Polygons liegt oder nicht (point inclusion problem); das Problem der Berechnung des Voronoi–Diagramms; das Problem der Überschneidung zwei oder mehrerer konvexer Polygone. Die Strategie für das Lösen mancher dieser Probleme für allgemeine Polygone ist nach der Polygonzerlegung in einfache Teile, das Problem, für jedes Teil anhand des spezialisierten Algorithmus zu lösen und dann die Teil–Lösungen zu vereinigen.

Für die Optimierung mancher Algorithmen für Polygonzerlegungen werden andere Polygonzerlegungen als Verbesserungsschritt benutzt. Beispiel sind, unterschiedliche Algorithmen zur Triangulierung eines allgemeinen Polygons (Polygonzerlegung in Dreiecke), die Zerlegungen in Trapeze oder in monotone Polygone einzusetzen, um eine bessere Zeitkomplexität zu erreichen([Gaetal78], [FoMo84]). (Solche Algorithmen werden in den folgenden Kapiteln untersucht und sind in dem zu dieser Arbeit gehörenden Applet mit Animationen versehen.)

Andere Anwendungsbereiche der Polygonzerlegungen sind Datenkomprimierung [92], Datenbanksysteme [90], Bildverarbeitung [97], und Computergraphik [131].

Es wurde auch viel über die Zerlegung eines Polyeders in drei oder höhere Dimensionen geforscht. Dies ist aber kein Thema in dieser Arbeit, in der ich mich auf Polygonzerlegungen in der Ebene konzentriere.

Triangulierung, die Zerlegung des Inneren eines Polygons in Dreiecke, ist ein zentrales Problem in der algorithmischen Geometrie. Viele Algorithmen mit Polygonen fangen mit der Triangulierung an. Diese hat selbst zahlreiche Anwendungen (z.B in der Geodesie[39], [132]), die berühmteste davon bei dem Problem der Bewachung einer Kunstgalerie(engl. Artgallery problem). Sicherlich ist die Triangulierung ein Beispiel von Polygonzerlegung, aber da sie sich als ein eigenständiges Problem entwickelt hat, das eine eigene systematische Untersuchung braucht, wird sie in dieser Diplomarbeit nicht ausführlich betrachtet. Da sie aber die meist bekannteste Polygonzerlegung ist und die hier untersuchten Algorithmen für Trapezzerlegung, monotone Zerlegung, und Konvexzerlegung fast immer einsetzt, wird sie in einer Form präsentiert. Es werden zwei Algorithmen visualisiert und diskutiert, die ein Polygon mit der Hilfe monotoner und unimonotoner Teilpolygone in Dreiecke partitionieren. Dazu werden in Kapitel 6 zwei bekannte Algorithmen ([Cha91], [Sei91]) für lineare Triangulierung untersucht und teilweise visualisiert.

2.1 Klassifikation von Polygonen.

Es existieren viele unterschiedliche Typen von Subpolygonen, in die ein allgemeines Polygon aufgeteilt werden kann. Solche Typen umfassen festgelegte Formen wie Vierecke (Quadrate), Rechtecke und Trapeze, oder Subpolygone, die konvex, sternförmig (starshaped), monoton (unimonoton) oder spiralförmig sind.

Bevor wir weitergehen, ist es sinnvoll, manche der beschränkten (restricted) Typen zu definieren. Ein Punkt a in einem Polygon P ist *sichtbar* aus einem Punkt b in P, falls das Liniensegment, dass a mit d verbindet, völlig in P liegt. Ein Polygon P ist *konvex*, falls alle Paare von Punkten in P sichtbar auseinander liegen. Alle inneren Winkel in einem konvexen Polygon sind kleiner als 180°, d.h. alle Polygonecken sind konvex. Ein Polygon ist *sternförmig*, falls mindestens ein Punkt in P existiert, aus dem das ganze Innere des Polygons sichtbar ist. Die Gruppe von Punkten in P, aus denen P sichtbar ist, wird *Kern* von P genannt. Eine *Polygonkette* in einem Polygon P ist eine Folge aufeinander folgender Ecken von P. Ein *spiralförmiges* Polygon ist ein Polygon, dessen Rand genau eine konkave Subkette enthält. Eine Polygonkette ist *monoton* bezüglich einer Linie *L'*, falls jede orthogonale zu *L'* Linie L die Polygonkette in keinem Punkt, in einem Punkt oder in einem Liniensegment trifft. Ein Polygon P ist ***monoton*** bezüglich einer Richtungslinie L, falls P in zwei Polygonketten aufgeteilt werden kann, die monoton bezüglich L sind (siehe Kapitel 4). Für Beispiele aller erwähnten Polygonen siehe Bild1.1.

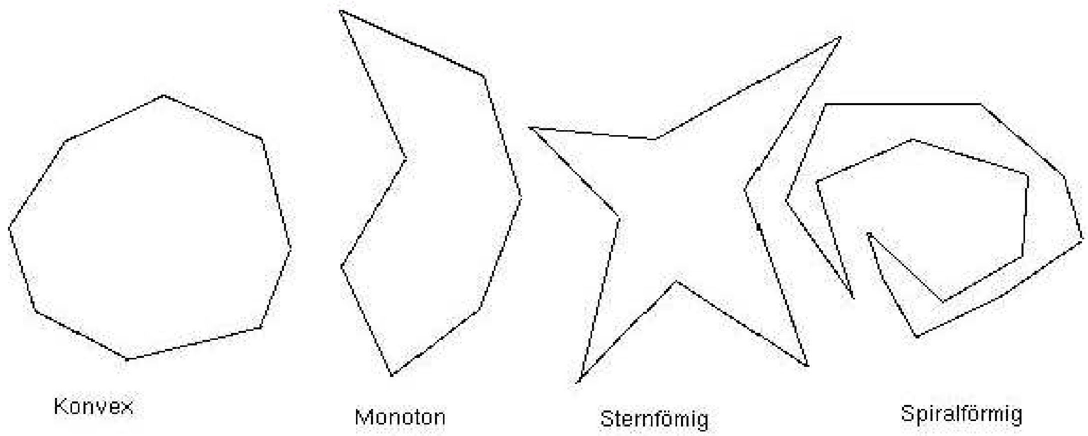

Bild2.2. Klassifikation von Polygonen.

Es ist auch sinnvoll, die Typen von Polygonen zu klassifizieren, die zerlegt werden sollen. Die Polygone können einfach verbundene *ohne Löcher* oder *mit Löcher* sein. Die einfachen Polygone sind nach der Definition *ohne Löcher,* d.h. das Innere des Polygons ist topologisch äquivalent zu einer Scheibe. Die Löcher sind nicht überlappende einfache Polygone, die sich in dem Hauptpolygon befinden und wie “Inseln” darin aussehen. Die Komplexität des Problems der Polygonzerlegung ist normalerweise höher, wenn das Polygon Löcher enthält. Die in dieser Arbeit betrachteten Polygonzerlegungen behandeln Polygone ohne Löcher. Manche von ihnen können aber leicht erweitert werden, um mit Löchern zu funktionieren.

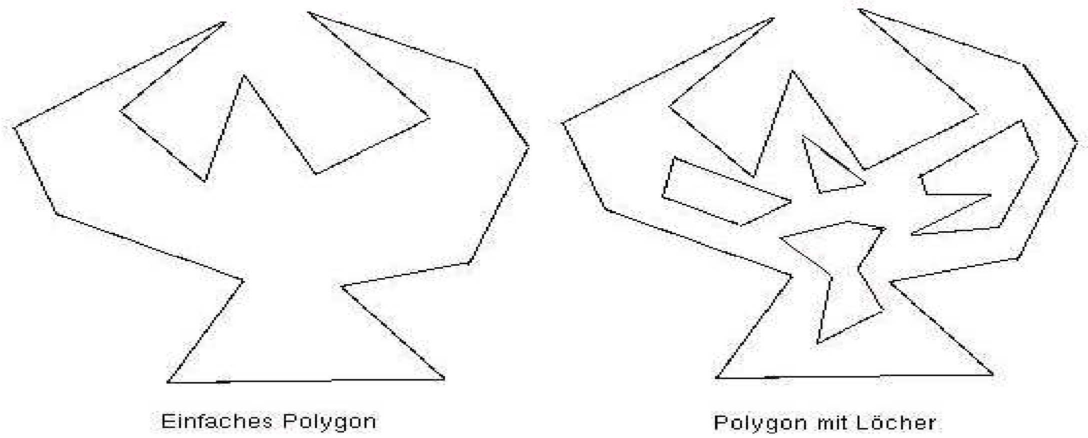

Bild2.3 Beispiel für ein einfaches Polygon und ein Polygon mit Löchern.

Ein wichtiger Typ von Polygonen ist der der *orthogonalen* Polygone, bei denen alle Kanten parallel zu den Koordinatenachsen liegen. Diese Polygone ergeben sich natürlich bei bestimmten Anwendungen wie dem VLSI–Design. Trotz der Wichtigkeit dieses Typs von Polygonen betrachte ich in der vorliegenden Diplomarbeit vor allem Zerlegungen der einfachen Polygone, die bezüglich der Orientierung der Kanten keine Beschränkung haben.

2.2 Klassifikation von Polygonzerlegungen

Die Polygonzerlegungen werden auch danach klassifiziert, wie die Subpolygone zueinander in Beziehung stehen. Eine Zerlegung wird *Partition (partition)* genannt, falls die Subpolygone sich nicht, außer auf deren Grenzen, überlappen. Falls überlappende Teile der verschiedenen Subpolygone erlaubt sind, wird die Zerlegung *Bedeckung*(cover) genannt. Hier werden nur die Partitionen von Polygonen betrachtet und mit Polygonzerlegung wird immer die Partition im obengenannten Sinn gemeint.

Die Polygonzerlegungen in einfachere (im Sinne von "weniger komplizierte") Subpolygone können mit oder ohne Hilfe von zusätzlichen Punkte ausgeführt werden. Solche zusätzlichen Punkte werden üblicherweise *Steiner–Punkte* (Steiner–points) genannt. Obwohl das Verwenden von Steiner–Punkte die nachfolgende Bearbeitung des zerlegten Polygons komplizierter macht, führt es zum Erhalt von weniger Teilpolygonen. (Siehe Beispiele in Kapitel über die Konvexzerlegung).

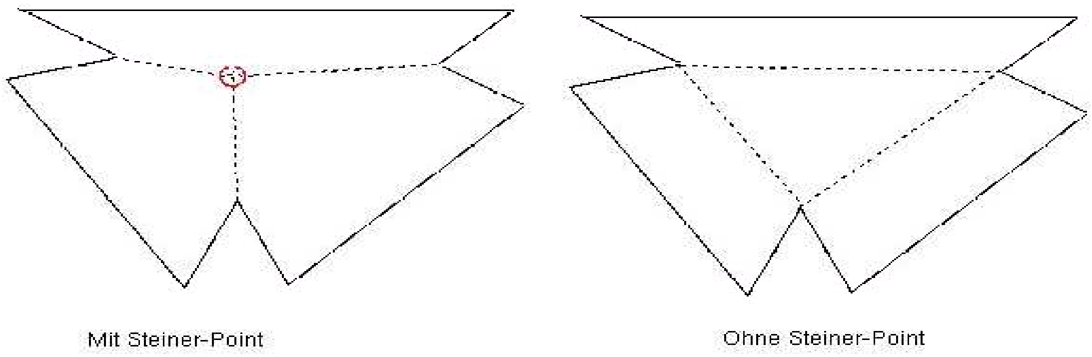

Bild2.3 Polygon, partitioniert in konvexe Teile a) mit und b) ohne Steiner–Punkte.

In einem Polygon mit n Ecken sind r davon konkav (d.h. größer als 180°). Die Zahl r kann viel kleiner als n oder null sein und manche Autoren (Keil) analysieren die Komplexität eines Polygonszerlegungs–Algorithmus relativ zu beiden Parametern n und r. (Beispiel: Kapitel 6, Konvexzerlegung).

Bei den meisten Anwendungen wird gewünscht, dass die Zerlegung minimal in einem Sinn ist. Manche brauchen ein Polygon, das in eine minimaler Anzahl von bestimmten Subpolygonen zerlegt ist. Andere brauchen die Zerlegung, bei der die summierte Länge der Liniensegmente, die die Zerlegung bilden, minimal ist. Für andere Applikationen ist einfach nur der schnellst möglichste Algorithmus interessant.

Bei der Partitionierung eines Polygons in Subpolygone wird der Typ der benutzten Subpolygone entsprechend dem Anwendungsbereich, in dem die Zerlegung eingesetzt wird, festgelegt. Z.B. setzt die syntaktische Mustererkennung die Zerlegungen in konvexe, spiralförmige und sternförmige Teilpolygone ein. VLSI–Anwendungen benutzen die Trapezzerlegung (Zerlegung in Trapeze).

Quellen der Informationen in diesem Kapitel sind vor allem [GoO'Rou] und [Kei96].

Im nächsten Kapitel wird die Trapezzerlegung untersucht, indem der meist benutzte Algorithmus, seine Implementierung und Visualisierung (Animation) erklärt wird.

Kapitel 3

Trapezzerlegung von Polygonen

Die Zerlegung eines Polygons in Trapeze wurde zum ersten Mal Anfang der achtziger Jahren von Chazelle und Incerpi [ChaIn87] und Fournier und Montuno [FoMo84] als Zwischen–Schritt der Triangulierung vorgestellt. Die häufigste Anwendung der Trapezzerlegung blieb ihr Ansatz als erster Schritt in den Algorithmen für die Zerlegung eines Polygons in Dreiecke. Beispiele für solche Algorithmen werden in den Kapiteln 4 und 5 untersucht und visualisert und in Kapitel 6 werden zwei bekannte lineare Triangulierungs–Algorithmen diskutiert, die auch Trapezzerlegung einsetzen.

Eine bekannte Anwendung dieser Zerlegung ist beim VLSI–Design (siehe Kapiel 2), wo Trapeze mit horizontaler Kante gebraucht werden.

Die Trapezzerlegung ist auch bei Sichtbarkeits– und Wegeplanungs–Probleme nutzbar, da damit eine breite und komplexe Szene in einfachere kleinere Regionen (die Trapeze) zerlegt werden kann. Nach der Zerlegung kann man die Idee der Nachbarschaft zwischen der Subregionen in der Szene einbringen. Sobald man alle Regionen hat, kann man die ganze Szene verarbeiten. Wenn z.B. ein bestimmter Punkt in der Szene gegeben ist, kann man in dem geteilten Polygon leichter den Bereich berechnen, in dem der Punkt liegt und mittels der Information über die Nachbarregionen diejenigen Regionen finden, die von dem Punkt aus sichtbar sind.

Ein anderer Anwendungsbereich der Zerlegung in Trapeze ist die Füllung von Regionen (region filling). Die Füllung von Regionen ist ein Problem in den Geoinformationssystemen für die Färbung von geographischen Karten. Für bestimmte spezielle Probleme der Färbung von geographischen Karten ist das Zeichnen von Trapezen schneller als das Zeichnen von Dreiecken.

3.1 Was ist Trapezzerlegung und wie entsteht sie?

Die Trapezzerlegung ist die Aufteilung der Fläche eines einfachen Polygons in Trapeze. Ein Trapez ist ein vierseitiges Polygon (Viereck) mit zwei parallelen unteren und oberen Kanten.

Im Unterschied zu der Zerlegung in Dreiecke darf man bei der Trapezzerlegung Liniensegmente benutzen, die keine Diagonalen sind, also Endpunkte haben, die keine Polygonecken sind. (Siehe Zerlegungen mit „Steiner–Punkte“ in Kapitel 2). Ein Polygon kann auf zwei Weisen in Trapeze geteilt werden, je nach dem ob die parallelen Kanten der Trapeze horizontale oder vertikale Liniensegmente sind. Es existieren, also *horizontale und vertikale Trapezzerlegungen*. Hier betrachten wir die horizontale Trapezzerlegung.

Wie oben schon erläutert wird, in der algorithmischen Geometrie die Trapezzerlegung oft als Zwischenschritt in vielen Algorithmen verwendet. Entsprechend der Darstellung der Zerlegung kann man verschiedene Definitionen finden. In dieser Arbeit wird sie fast in allen vorgestellten Algorithmen eingesetzt und in zwei Definitionen angegeben. Die erste folgt hier, die andere kommt in Kapitel 6.

Definition: *Sei P ein gültiges einfaches Polygon. Sei ein horizontales Liniensegment S durch jeden Polygonsknoten V gezeichnet, so dass S in dem Inneren von P liegt und der einzige Punkt, der zu P und S gehört V ist. (S kann ganz auf der linken oder auf der rechten Seite von V liegen oder kann gleich V sein) Alle Liniensegmente S in P stellen die* **horizontale Trapezzerlegung** *von P zusammen.*

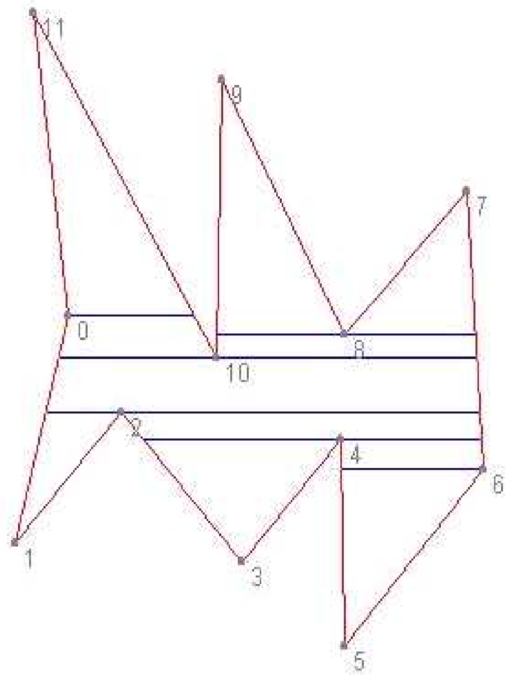

Bild3.1: Ein einfaches Polygon nach horizontaler Trapezzerlegung

In dieser Arbeit ist die Trapezzerlegung eines Polygons als horizontale Trapezzerlegung des Polygons gemeint.

In der algorithmischen Geometrie sind verschiedenen Algorithmen für Trapezzerlegung entdeckt worden. Einige ([FoMo84]) benutzen die Technik der Sweep–Line, die auch noch Scan–Line–Prinzip genannt wird [OW92]. Andere verwenden die horizontale Erweiterung der Polygonspunkte und damit die Sichtbarkeitskarte („visivility map") des Polygons. Diese werden in Kapitel 6 diskutiert. In diesem Kapitel diskutieren wir einen *O(nlogn)*-Algorithmus, der das Sweep–Line Prinzip ansetzt, das vom Nievergelt und Preparata [NP82] 1982 zum ersten Mal vorgestellt wurde und in vielen geometrischen Algorithmen angesetzt ist. Bevor der Algorithmus für Trapezzerlegung präsentiert wird, wird das Prinzip unabhängig von ihm erklärt, da es eine wichtge Rolle in der algorithmischen Geometrie spielt.

3.1.1 Sweep–Line–Prinzip

Gegeben sei eine Menge von geometrischen Objekten in der Ebene, z.B. eine Menge von Liniensegmenten, die ein einfaches Polygon bilden. Die Idee ist nun, eine horizontale Linie (die so genannte Sweep–Line) von unten nach oben (oder alternativ: eine vertikale Linie von links nach rechts) über die gegebene Menge zu schwenken, um ein die Menge betreffendes statisches zweidimensionales geometrisches Problem in eine dynamische Folge eindimensionaler Probleme zu zerlegen. Die Sweep-Linie teilt zu jedem Zeitpunkt die gegebene Menge von Objekten in drei disjunkte Teilmengen: Die *toten* Objekte, die bereits vollständig von der Sweep-Linie überstrichen wurden, die gerade *aktiven* Objekte, die gegenwärtig von der Sweep-Linie geschnitten werden und die noch *inaktiven* Objekte, die erst künftig von der Sweep-Linie geschnitten werden. Die Sweep-Linie definiert eine lokale Ordnung auf der Menge der jeweils aktiven Objekte; sie muss gegebenenfalls den sich ändernden lokalen Verhältnissen angepasst werden und kann für problemspezifische Aufgaben konsultiert werden. Während man die Sweep-Linie über die Eingabeszene hinwegschwenkt, hält man eine dynamische, d.h. zeitlich veränderliche, problemspezifische Horizontalstruktur L, in der man sich alle für das jeweils zu lösende Problem benötigten Daten merkt. Eine wichtige Beobachtung ist nun, dass man anstelle eines kontinuierlichen Schwenks (sweep) die Sweep–Linie in diskreten Schritten über die gegebene Szene führen kann. Sei Q die aufsteigend sortierte Menge aller y–Werte von Objekten. D.h.: Im Falle einer Menge von nichthorizontalen Liniensegmenten ist Q die Menge der unteren Endpunkte der Liniensegmente. Ganz allgemein wird Q gerade so gewählt, dass sich zwischen je zwei aufeinander folgenden Punkten in Q weder die Zusammensetzung der gerade aktiven noch deren relativen Anordnung (längst der Sweep-Linie) ändern. Dann genügt es, Q als Menge der *Haltepunkte* der Sweep-Linie zu nehmen. Statt eines kontinuierlichen Schwenks „springt" man mit der Sweep-Linie von Haltepunkt zu Haltepunkt in aufsteigender y–Reihenfolge (bei horizontaler Sweep-Linie; alternativ: in aufsteigender x–Reihenfolge bei vertikaler Sweep–Linie).

Algorithmus *Sweep-Linie–Prinzip*

{liefert zu einer Menge isoorientierter Objekten problemabhängige Antworten}
Q := objekt und problemabhängige Folge von Haltepunkten in aufsteigender y–Reihenfolge;
L := 0; {angeordnete Menge der jeweils aktiven Objekten}

While Q nicht leer **do**
begin

1. Wähle nächsten Haltepunkt aus Q und entferne ihn aus Q;
2. Update(L) und gib (problemabhängige) Teilantwort aus

end

Wir wollen das durch diesen Rahmen formulierte Sweep-Linie–Prinzip auf das Problem der Trapezedierung anwenden.

3.2 Algorithmus für Zerlegung eines Polygons in Trapeze

Im Folgenden wird ein Algorithmus für die Zerlegung eines allgemeinen Polygons ohne Löcher in horizontale Trapeze in O(nlogn)–Zeit untersucht. Wie schon erwähnt wurde, erlaubt der Algorithmus die Verwendung von Steiner–Punkte, die aber einfach mit der Sweep–Linie berechnet werden können. Der Algorithmus kann leicht so erweitert werden, dass er mit Polygonen mit Löchern funktioniert.

Es sei ein Polygon gegeben, dessen Fläche in Trapeze aufgeteilt werden soll. Die Idee des Sweep-Linie–Prinzips ist eine horizontale Linie über das Polygon zu schwenken, die auf jeden Polygonspunkt hält, bestimmte Aufgaben mit Daten aus ihrer Horizontalstruktur erledigt und dann die Horizontalstruktur je nach den neuen lokalen Verhältnissen aktualisiert. Die Menge Q (siehe oben) ist die Menge der Polygonspunkte, die Haltepunkte der Sweep-Linie sind. Die Haltepunkte werden in aufsteigender (da die Sweep–Linie von unten nach oben schwenkt) y–Reihenfolge sortiert.

Das Ziel bei diesem Problem ist, bei jedem neuen Haltepunkt ein neues Trapez zu bekommen. Dafür muss man bei jedem aktiven Polygonsknoten wissen, welche die Kanten rechts und links des aktiven Polygonsknoten sind. Ausgehend von dieser Frage, stellt man hier die problemspezifische Horizontalstruktur dar. Um die Frage effizient bei jedem aktiven Punkt beantworten zu können, wird die Horizontalstruktur als eine sortierte Liste dargestellt, die alle von der Sweep-Linie geschnittenen Polygonskanten erhält. Diese Liste ist in aufsteigender Reihenfolge nach der x–Koordinate der Schnittpunkte sortiert.

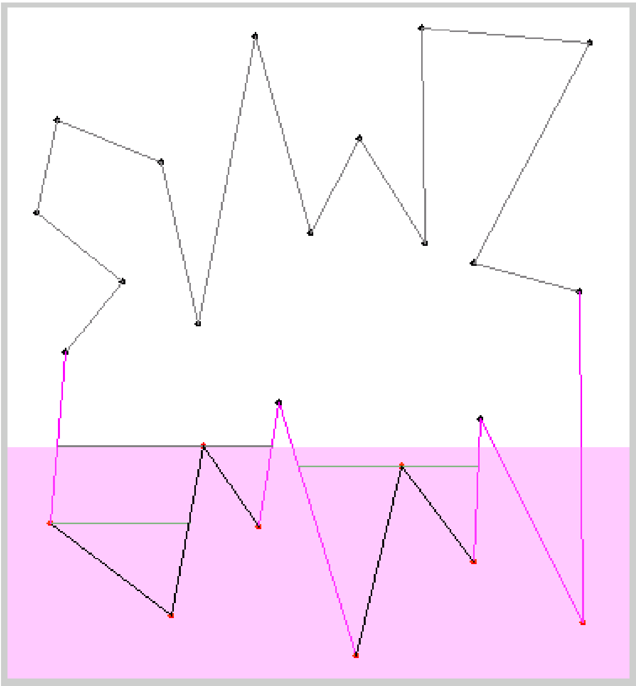

Bild3.2 Die in lila gezeichneten Kanten sind durch die Sweep-Linie geschnittene Kanten, also die aktuellen Kanten.

Es sei eine solche Liste L mit den von Sweep–Line geschnittenen Kanten gegeben. Wie kann man nun ermitteln, zwischen welchen Polygonkanten der aktuelle Haltepunkt (der letze rote) liegt? Das kann man berechnen, falls man die x-Koordinaten der Schnittpunkte der Sweep–Linie mit den Kanten auf der Höhe des Haltepunkte (wo auch die sweepline liegt) weiß. Diese Schnittpunkte sind die vom Algorithmus benutzten Steiner-Punkte. Es ist nicht schwer die x-Koordinaten dieser Punkte zu finden, da man die Koordinaten der Endpunkte der Kanten und des aktuellen Haltepunktes hat. Dann kann man die x–Koordinaten mit der Formel $x = kante.dxdy * (y - kante.y1) + kante.x1$ berechnen, wobei $dxdy = (x2 - x1)(y2 - y1)$ und x1 die x–Koordinate des Anfangspunktes der Kante ist[1]. Dann kann man diese mit der x–Koordinaten des aktuellen Haltepunktes vergleichen und so seine Lage finden.

Falls man für diese Suche die Liste einfach von links nach rechts überquert, würde die Berechnung eine Zeit dauern, die proportional der Länge der Liste (O(n)) ist. Falls man aber die Liste L in einer effizienten Datenstruktur wie Höhebalancierter Baum speichert, würde die Suche bei jedem Polygonspunkt nur *O(log n)*–Zeit dauern. Folglich die Zeit für die Berechnung der Lage jedes aktuellen Polygonspunktes wäre *O(nlogn)* für n Polygonspunkte. Es ist noch wichtig zu

[1]Die Formel ergibt eine reelle Zahl, die später zu einer natürlcihen Zahl gerundet wird, da die vom Programm benutzten Polygonpunkte Integerkoordinaten haben. Diese Approxiamtion führt aber zu keinen Ungenauigkeiten bei den nachfolgenden Berechnungen

zeigen, dass diese Horizontalstruktur bei jeden Polygonsknoten aktualisiert wird und zwar in Zeit *O(nlogn)* für das ganze Polygon. Das letzte ist leicht zu zeigen, falls die Datenstruktur die Operationen Einfügen und Löschen eines Elements in Zeit *O(logn)* unterstützt. Solche Datenstrukturen ist z.B der Red–Black–Baum [OW92]. Für die Implementierung des Trapezzerlegungs–Algorithmus in der Applikation hier, wird diesen Baum benutzt. (Unten wird die Implementierung ausführlich erklärt.)

Im folgenden möchte ich zeigen, wie sich die Horizontalstruktur bei einem Polygonpunkt ändern kann. Es gibt drei unterschiedlichen Situationen, die in den nächsten Bildern gezeigt werden. Sei v der aktuelle Haltepunkt der Sweep Linie, der den Kanten c und d gehört und zwischen den Kanten a und b liegt. Sei L die Horizontalstruktur, die z.B. mittels einen Red–Black–Baum implementiert werden kann.(Mehr über die Implementierung siehe unten)

Fall1: Der Halteknoten v ist entweder der erste Polygonspunkt, auf dessen beiden Seiten es keine Kante gibt (Fall1a) oder liegt v zwischen den Kanten a und b (Fall1b). Die beide v erhaltenden Kanten c und d liegen über der Sweep–Linie. Da die Sweep–Linie nach oben schwenkt, sollen c und d in die Datenstruktur L eingefügt werden. Die Polygonsknoten v in diesem Fall werden hier *"Start–Knoten"* genannt.

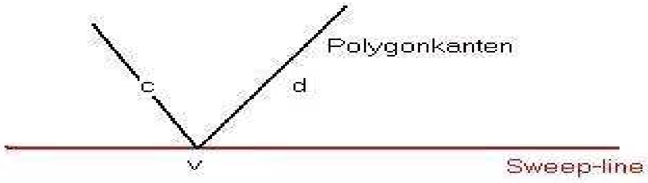

Bild3.3. Fall1a Einfügen von Kanten a und b in die Liste L. L war bis jetzt leer.

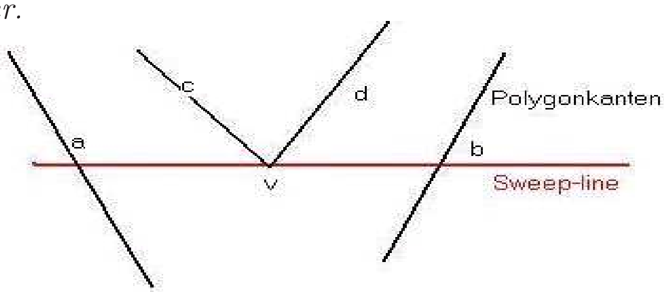

Bild3.4 Fall1b Einfügen von den Polygonkante c und d zwischen a und b in die Liste

Fall2: Der Haltepunkt v ist zwischen a und b. Es kann sein, dass v der erste oder der letzte Punkt der Liste aller Schnittpunkte ist und dann hat er nur eine Nachbarkante links oder rechts. Die Kante d liegt über und Kante c liegt unter der Sweep–Linie. Dann soll die Polygonskante c aus L gelöscht werden und an ihrer Stelle soll die Kante d eingefügt werden. Der Polygonsknoten v wird *"regulärer Knoten"* genannt.

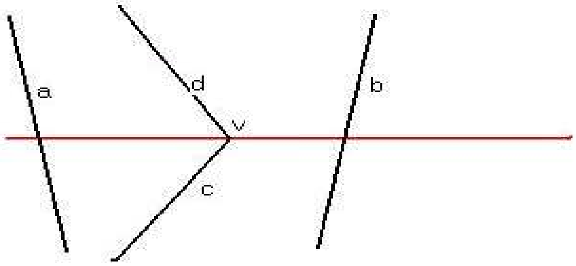

Bild3.5 Fall2 – Die Kante c soll durch d ersetzt werden.

Fall3: Der Haltepunkt v liegt entweder zwischen den Polygonskanten a und b oder ist der höchste Polygonknoten des Polygons und also hat keine Nachbarnkante auf seinen Seiten. Die v enthaltende Kanten c und d liegen unter der sweep Linie, also sollen aus der Datenstruktur L gelöscht werden. Der Haltepunkt v in diesem Fall wird *Ende–Knoten* genannt.

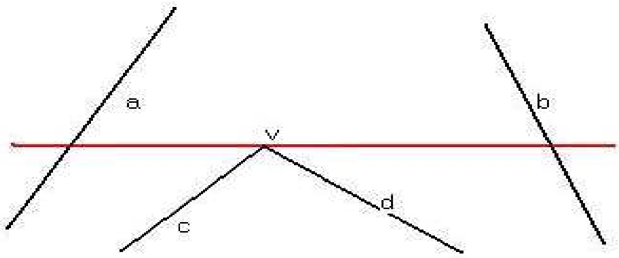

Bild3.6 Fall3a – Die Kanten c und d werden aus der Liste L gelöscht.

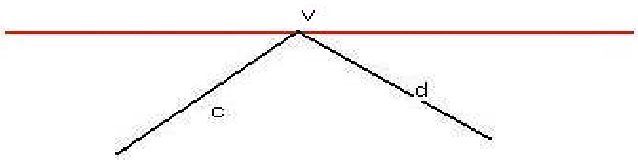

Bild3.7. Fall3b – Die Polygonskante a und b werden aus der Liste L gelöscht. Die Liste L wird leer.

Im folgenden Bild sind die drei Fälle mit entsprechend den Start–, regulär und Ende–Knoten zu sehen.

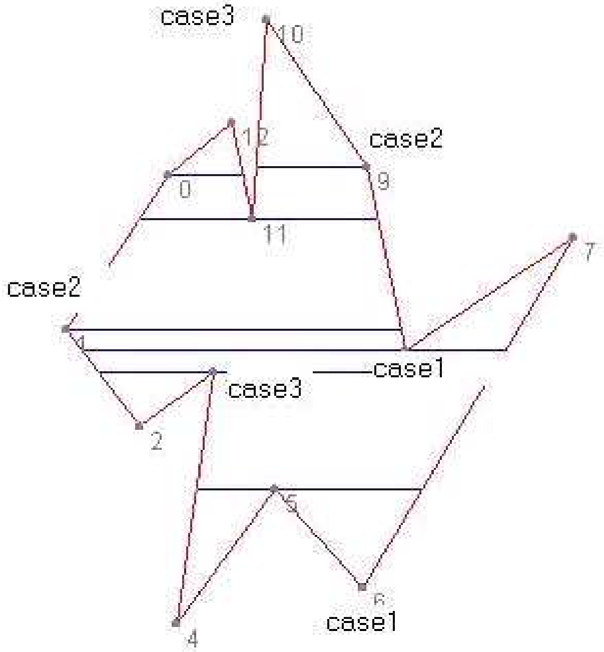

Bild3.8. Polygonspunkte 6 uns 8 sind Start–Ecken, Punkte 4 und 9 sind regulär Ecken und Punkte 3 und 10 sind Ende–Ecken.

Im folgenden Pseudocode des Algorithmus werden mit dem Begriff "aktive Kanten" diese Kanten bezeichnet, die gerade von der sweep–Linie geschnitten werden.

Algorithmus: *Horizontale Trapezzerlegung eines Polygons*

Eingabe: Einfaches (simples) allgemeines Polygon, dessen n Knoten in einem Vektor gespeichert werden

Ausgabe: Vektor V mit Liniensegmente, die Trapezzerlegung des Polygons zusammenstellen.

1. Sortiere die n Polygonsknoten aufsteigend nach deren y–Koordinate.

2. Sortiere die Polygonskanten aufsteigend nach der y–Koordinate deren unteren Endpunkten. Die sortierten Kanten werden im Vektor Q gespeichert.

3. Initialisiere einen leeren Red–Black–Baum T, der die aktiven Kanten enthalten soll.

4. **for** jeden Polygonvertex v_{i} mit $0 < i < n$ und y-Koordinaten der Punkten v_i und v_{i+1} $y_{v_i} < y_{v_{i+1}}$ **do**

4.1 Finde die rechte und/oder linke Nachbarkante von v_{i} in T und führe entsprechend die nächsten Schritte aus.

case1a v_{i} ist eine Start–Ecke, also ist keine Kante in T, die v_{i} enthält, gefunden. v_{i} hat keine zwei Nachbarkanten in T, die ein Trapez bilden können.

4.2 Hole die ersten zwei Polygonskanten aus Q.

4.3 Lösche die Kanten aus Q.
4.4 Füge diese zwei Kanten in T ein

case1b v_i ist eine Start–Ecke also keine Kante die v_i enthält ist gefunden.v_i liegt zwischen zwei Kanten a und b, die zusammen mit dem horizontalen Liniensegment durch v_i und dem letzten horizontalen Liniensegment durch a und b, ein neues Trapez produzieren.

4.2 Füge die neue Trapezkante in V ein.
4.3 Hole die nächsten zwei Polygonkanten aus Q.
4.4 Lösche diese aus Q.
4.5 Füge sie in T ein.

case2 Haltepunkt v_i ist eine reguläre Ecke. Eine v_i enthaltende Kante ist gefunden. (Die Kante hat v_i als Endpunkt, da die Polygonspunkte von unten nach oben bearbeitet werden bzw. die Sweep–Linie schwenkt von unten nach oben)

4.2 Nimm ihre rechte oder linke Nachbarkante aus T, mit der sie eine neue obere Trapezkante durch vi produziert.
4.3 Füge die neue Trapezkante in V ein.
4.4 Hole die neue Polygonkante d aus Q.
4.5 Lösche d aus$Q$.
4.6 Lösche die Kante c aus T, die v_i als Endpunkt enthält.
4.7 Füge die Kante d, die v_i als Anfangspunkt enthält in T ein.

case3a Haltepunkt v_i ist eine Ende–Ecke. Es sind zwei aktuelle Polygonkanten c und d in T gefunden, die v_i als Endpunkt haben. v_i hat Nachbarkanten rechts und links, die auch aktuelle Kanten und in T gespeichert sind.

4.2 Hohle die rechte Nachbarkante a von c und die linke Nachbarkante b von d, die eigentlich entsprechend rechte und linke Nachbarkanten vom vi sind.
4.3 Füge die nächste obere Trapezkante in V ein.
4.4 Lösche c und d in T.

case3b Haltepunkt v_i ist eine Ende–Ecke. Es sind zwei aktuelle Polygonkanten c und d in T gefunden, die v_i als Endpunkt haben. v_i hat keine Nachbarkanten von beiden Seiten.

4.2. Lösche c und d in T .

Analyse: Das Sortieren der Polygonpunkte verlangt Zeit *O(nlogn)*. Einfügen von Liniensegment in Vektor braucht Zeit *O(1)*. Alle für die Aktualisierung der horizontalen Datenstruktur benötigten Operationen Finden, Einfügen, Löschen eines Elements, Finden des Vorgängers oder Nachfolgers eines Elements in einem

Red–Black–Baum sind in Zeit *O(logn)* erfüllbar. Also kann man feststellen, dass für ein Polygon mit n Punkten der Algorithmus *O(nlogn)*–Zeit braucht.

Theorem: *Ein einfaches gültiges Polygon P mit n Polygonspunkten kann in Trapeze in Zeit O(nlogn) zerlegt werden.*

Im zu dieser schriftlichen Arbeit gehörenden Applet „Visualisierung von Algorithmen für Polygonzerlegungen“ ist der Algorithmus mit seinen Schritten und Fällen visualisiert. Man kann verfolgen, wo die Sweep–Linie hält, welche Kanten aktuell sind, welche Nachbarkanten von v_i gefunden, eingefügt und gelöscht werden. In dem Pseudocode–Panel wird beschrieben, welcher Algorithmusschritt momentan geschieht.

3.2.1 Implementierung

Die Implementierungs–Klasse des Algorithmus ist „*TrapezidierungByRBTree*“. In der Methode *Trapezoidation()* werden die Algorithmusschritte implementiert und visualisiert. Sie ruft die Methoden *sortVectorPoints*() um die Punkte nach y–Koordinate zu sortieren und *SortVectorEdges*(), um die Polygonskante, sortiert nach ihren Anfangspunkt in einem Vektor (die Variable vectorSortedEdges), zu speichern. (Der Anfangspunkt ist der mit der kleineren y–Koordinate). Dann wird eine for–Schleife durch die sortierten Polygonecken von unten nach oben gestartet.

Die von der Sweep–Linie abhängige horizontale Datenstruktur wird von der Klasse „*DictionaryActualEdges*“, die eine Erweiterung der *RedBlackTree_ withRemove* –Klasse ist und einen speziellen Red-Black–Baum darstellt, der keine horizontale Polygonskante enthält. Es ist anzumerken, dass zwei solcher Kanten a und b nur dann verglichen werden, wenn sie gleichzeitig von einer horizontalen Linie (die Sweep–Linie) geschnitten werden. Beim Vergleich ist a kleiner als b, wenn a links von b entlang der horizontalen Linie liegt.

Wie im Kapitel 2 erwähnt, wird es angenommen, dass die Polygonspunkte des Eingabe–Polygons P unterschiedliche y–Koordinaten haben. Diese Eigenschaft ist keine Einschränkung der Allgemeinheit, da sie durch eine kleine Drehung des Koordinatensystems erzielt werden kann. Die Drehung kann auch durch eine lexikographische Anordnung simuliert werden: Haben zwei verschiedene Endpunkte gleiche Y–Koordinaten, dann liegt der Punkt mit der kleineren X–Koordinate tiefer.

Bei jedem Punkt in der Schleife wird nach seiner benachbarten Kante in der gerade erkärten *DictionaryActualEdges* gesucht und je nach dem Suchergebnis wird festgestellt, welcher Typ bzw. in welchem oben erklärten Fall sich der Haltepunkt befindet. Entsprechend des Falles wird mit Hilfe der Methode *makeTrapeziumUpperEdge*() das nächste Trapez dargestellt. Mit den insert() und remove() Methoden von der *DictionaryActualEdges*Klasse wird die Datenstruktur in *O(logn)*Zeit aktualisiert.

Information über die Implementierung enthält die JAVA–Dokumentation zur Applikation.

3.2.2 Visualisierung

Bei der Visualisierung von Algorithmen, stellt sich als erstes die Frage: Was vom Algorithmus visualisiert werden soll? Wie in Kapitel 1 diskutiert, spielt das Konzept der "interessante Ereignisse" (siehe 1.2.1) bei der Visualisierung eine wichtige Rolle. Ein Algorithmus–Schritt soll visualisiert werden, wenn er zu einem interessanten Ereignis führt. Ein interessantes Ereignis könnte z.B. sein, wenn die behandelten geometrischen Objekte ihre Position wechseln oder überhaupt von den Datenstrukturen entfernt werden oder wenn ein geometrisches Objekt vom Algorithmus erzeugt wird und in der Szene erscheint.

Normalerweise kann man die im Pseudocode gegebenen Schritte als interessant bezeichnen, da sie immer zu Änderungen bei den Daten und Datenstrukturen führen.

In dem Trapezzerlegung–Algorithmus nach dem ersten Sortierungs-Schritt entstehen die wichtigen Änderungen bei jedem Punkt, auf dem die Sweep–Linie hält. Deswegen wird so ein Haltepunkt rot gekennzeichnet. Da die Sweep–Linie eine entscheidende Rolle beim Aufbau der gesuchten Trapeze spielt, finde ich es sinnvoll, ihre Bewegung über die Polygonecken zu visualisieren. Wenn sie auf einer Polygonecke stoppt, kann der Betrachter in der Animation sofort sehen, wo sie die benachbarten Kanten des Haltepunkts schneidet und also welche die obere horizontale Kante des nächsten Trapeziums sein soll. Beim nächsten Schritt wird diese Kante in grauer Farbe gezeichnet, wobei das letzte gebaute Trapezoid erkennbar wird. Durch die Visualisierung der Sweep–Linie wird auch gezeigt, welche Polygonkanten die Linie überhaupt in dem entsprechenden Moment schneidet, womit automatisch die aktuellen Kanten (siehe oben) erkennbar werden.

Nachdem das Trapezoid gefunden wurde, soll die horizontale Datenstruktur mit den aktuellen Kanten, nämlich der Red-Black–Baum, aktualisiert werden. Wie oben schon diskutiert wurde, werden während der Aktualisierung je nach Fall entweder manche Kante(n) gelöscht oder manche eingefügt oder beides. Solche Änderungen der Datenstrukturen werden als interessant betrachtet und sollen visualisiert werden. Deswegen wechseln die gelöschten Kanten ihre Farbe in schwarz und die Kanten, die aktuell bzw. eingefügt werden, werden lila. In welchem Fall sich der akuelle Haltepunkt befindet, erklären die im Pseudocode hervorgehobene Zeile und die entsprechende Farbe des Punktes: dunkelblau für den ersten, hellblau für den zweiten und hellgrün für den dritten Fall.

Mittels der transparenten rosa Farbe, die hinter der Sweep–Linie das Zeichen–Fenster auffüllt, kann man die historische Entwicklung des Algorithmus erfolgen. Man sieht welches Teil des Polygons schon partitioniert wurde und welches noch berechnet werden soll.

Kapitel 4

Zerlegung eines Polygons in monotone Teilpolygone

Im vorigen Kapitel wurde die Trapezidierung untersucht, die in dem ersten schnellen Triangilierungs–Algorithmus eingesetzt wurde, um das Polygon zuerst in einfachere Sub–Polygone zu partitionieren. Dies haben Garey at al.[GJPT78] mit dem *O(nlogn)*–Algorithmus vorgestellt. Sie haben das Polygon zuerst in *O(nlogn)*–Zeit in monotone Subpolygone geteilt und dann jedes Subpolygon in linearer Zeit in Dreiecke partitioniert. Da der Algorithmus erst die Trapezidierung einsetzt, sind Steiner–Points erlaubt. Er kann in Fällen benutzt werden, wo keine minimale monotone Zerlegung verlangt wird. Wenn eine Zerlegung allgemeiner Polygone ohne Löcher in minimaler Zahl monotoner Teilpolygone benötigt wird, kann der Algorithmus von Keil [Keil83] mit Zeit $O(rn^4)$ (r ist die Anzahl der konkaven Ecken) benutzt werden . Die Autoren dieses Algorithmus haben gezeigt, dass das Problem der minimalen Zerlegung ohne Steiner–Punkte von Polygone mit Löcher NP–hart ist.

Das Konzept der monotonen Polygone wurde 1977 von Lee und Preparata[LP77] vorgestellt und eingesetzt. Hier verwende ich den Ausdruck "*monotone Zerlegung*" für eine Zerlegung in monotone Teilpolygone.

Da die monotone Zerlegung nicht nur für eine schnellere Zerlegung in Dreiecke, aber auch bei anderen Problemen eingesetzt wird, lohnt es, sich mit ihrer Untersuchung zu beschäftigen.

Außer der Anwendung als Zwischenschritt zu Triangulation wird sie auch zur Lösung anderer Probleme angewendet. Die monotone Zerlegung wird z.B. beim Erzeugen von Prozess–Planer (processplaner) benutzt. Zu den Funktionen des Prozess–Planer gehört die Funktion, numerisch den Verlauf der Shape Deposition Manufacturing (abgekürzt SDM) zu kontrollieren. SDM kombiniert die Operationen für die Zu- und Abnahme von Materialien bei der Herstellung von Metallobjekten, die willkürlich komplexen Formen (Geometrien) haben können. Die monotone Zerlegung hilft konkret bei dem Problem, die effizienteste Methode zu finden, die eine numerische Zerlegung von einem 3D CAD Model in 2D Schichten produziert. Diese numerische Zerlegung kann der Planer bei der Entwicklung eines Plans für die spezifische Prozesse einsetzen.

4.1 Definitionen und Eigenschaften

Die Monotonie wird immer relativ einer Richtungslinie definiert. Die Richtungslinie kann eine horizontale (die x–Achse) oder eine vertikale (die y–Achse) Gerade sein. Im Folgenden wird die y–Achse als Richtunglinie der untersuchten Polygone genommen. (Im [BKOS97] im Algorithmus für Zerlegung in monotone Polygone werden die x-monotone Polygone betrachtet).

Eine *Polygonkette* entsteht aus zwei oder mehr nacheinander folgenden Polygonkanten.

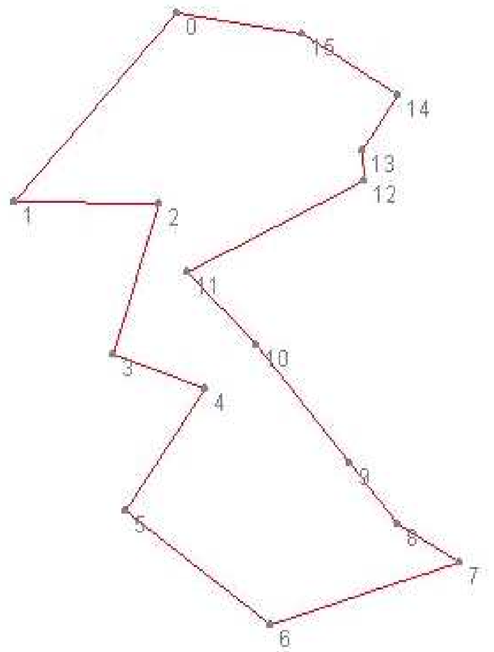

Bild4.1 Ein y–monotones Polygon. Die zwei monotone Polygonketten sind A=(p6, ... , p15,p0)und B=(p0,...,p6). Diese sind streng monoton.

Definition *Sei C eine Polygonkette eines Polygons P. Dann ist C* **streng monoton** *relativ einer Richtungslinie L', falls jede zu L' orthogonale Linie L entweder in einem oder in keinem Punkt C trifft.*

Mit anderen Worten ist ein Polygon *streng monoton,* wenn keine zwei Polygonecken gleiche y–Koordinaten haben

Definition: *Eine Polygonkette C ist* **monoton** *bezüglich einer Linie L, wenn jede zu L orthogonale Linie L' C in keinem Punkt, in einem Punkt oder in* *einem Liniensegment trifft.*

Definition: *Ein Polygon P ist* **monoton** *bezüglich einer Richtungslinie L, falls P in zwei Polygonketten geteilt werden kann, die monoton bezüglich L sind. Die zwei Polygonsketten haben 2 gemeinsame Polygonknoten und zwar die Punkte, die auf den beiden Enden jeder Kette liegen.*

Manche Polygone können monoton bezüglich mehreren Geraden sein und manche bezüglich keiner Gerade.

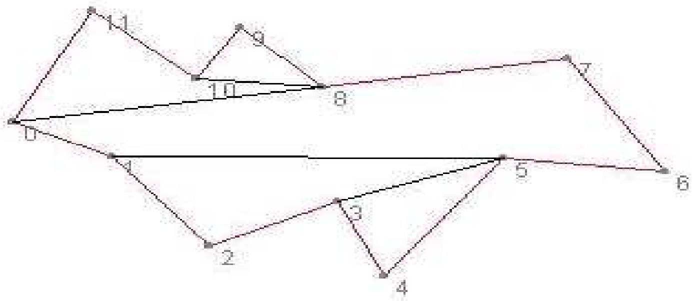

Bild4.2 Ein x–monotones Polygon, dessen Fläche in y–monotone Polygone geteilt wurde.

Es ist eine schwierige Aufgabe festzustellen, ob ein beliebiges Polygon monoton in irgendeiner unbekannten Richtungslinie ist. Die Lösung braucht nur Zeit *O(n)*, aber sie ist ziemlich kompliziert.

Viele Algorithmen, die kompliziert mit allgemeinen Polygonen funktionieren, funktionieren schnell und einfach mit monotonen Polygonen. Der Grund dafür ist die wichtigste Eigenschaft der monotonen Polygone und zwar, dass die Polygonknoten in jeder Polygonkette bezüglich der Linie der Monotonie sortiert sind.

Es sei angenommen, dass die Linie der Monotonie die vertikale y–Achse ist. Dann kann man in linearer Zeit die Knoten nach ihren y–Koordinaten sortieren. Wie geht das? Man findet den höchsten und den niedrigsten Polygonknoten und verteilt die Polygonknoten in zwei Ketten. Die erste Kette enthält die Knoten vom niedrigsten bis zum höchsten Knoten und die zweite Kette enthält vom höchsten bis zum niedrigsten Knoten. Wegen der Tatsache, dass das Polygon monoton ist, ist es sicher, dass die Knoten in jeder Kette nach der y–Koordinate sortiert sind. Um dann alle Polygonknoten zu sortieren, braucht man nur die beiden Polygonketten zusammenzulegen (to merge).

Es existiert eine bzgl. der Lage der Punkte einfache Charakteristik der Monotonie. Im Wesentlichen lautet sie, dass ein Polygon P monoton ist, wenn P monoton in der Nachbarschaft jeder seiner Knoten ist. Dies bedeutet, dass wenn man entlang der linken (oder rechten) Polygonkette vom höchsten bis zum niedrigsten Knoten geht, man sich immer nur nach unten oder horizontal bewegt, aber nie nach oben. Diese Tatsache kann die Basis für den in Teil 4.2 untersuchten Algorithmus für die Zerlegung eines Polygons in monotone Teile sein und zwar mittels Schneiden des Polygons entlang der Knoten, bei denen die Monotonie verletzt wird. Diese Knoten werden *Richtungsänderungen* genannt. Eine Richtungsänderung (interiour cusp) ist ein solcher Knoten k, dessen zwei Nachbarknoten wie folgt liegen: Beide über k (Z.B. im Bild4.3 Knoten 6) oder beide unter k (Z.B. der Knoten 2)

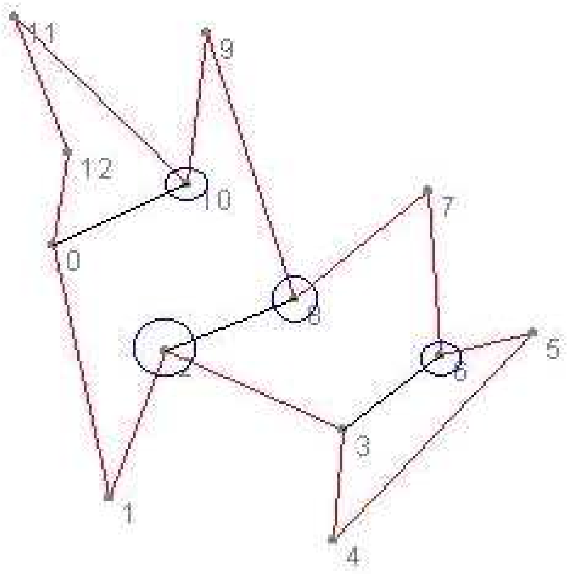

Bild4.3. Die in blauen Kreisen gezeichneten Knoten sind Richtungsänderungen.

Das folgende Lemma beweist eine grundlegende Behauptung bezüglich dieser Knoten.

Lemma 1 *Falls ein Polygon P keine Richtungsänderungen enthält, dann ist P monoton.*

Beweis

Wir nehmen an, das Polygon P sei nicht monoton und zeigen dann, dass es Richtungsänderungen besitzt. Es sei also G eine horizontale Gerade, die einen nicht zusammenhängenden Schnitt mit P hat (Wie im Bild 4.4 unten). Es seien weiter r, s und t Randpunkte von P auf G, so dass die offene Strecke $(r,\ s)$ ganz innerhalb und die Strecke $(s\ ,\ t)$ ganz außerhalb von P liegt. Wir dürfen annehmen, dass r, s und t bezüglich ihrer x–Koordinaten aufsteigend sortiert sind. Verfolgt man nun den Rand von P von s aus gegen den Uhrzeigersinn, so erreicht man entweder zuerst r oder zuerst t (siehe Bild4.4b).

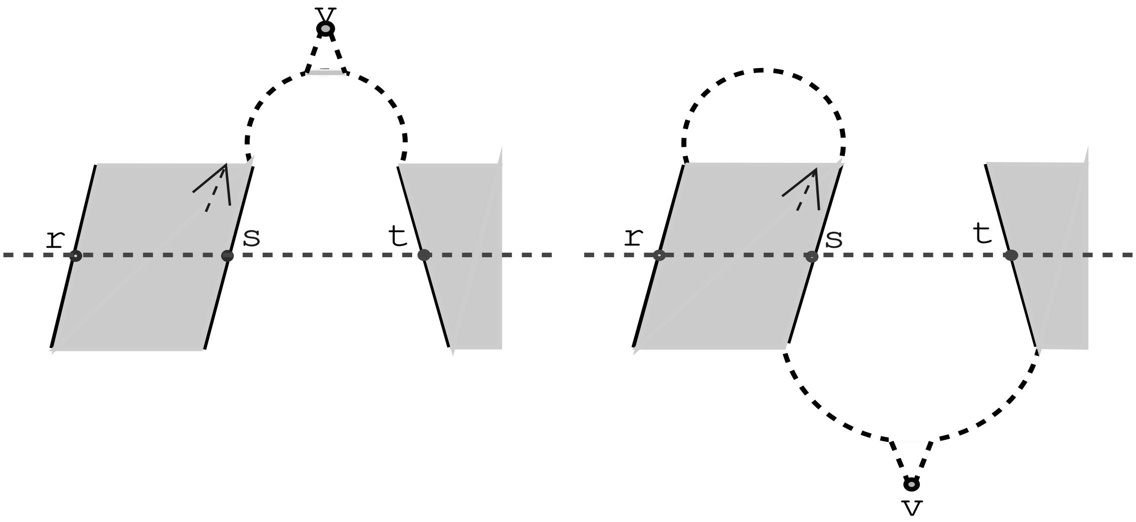

Bild4.4 Der Knoten v muss in beiden Fällen eine Richtungsänderung sein.

Wird zuerst t erreicht, so muss die am weitesten unten liegende Ecke v auf diesem Weg zwischen s und t eine Richtungsänderung (oder eine "untere Richtungsänderung") sein. Andernfalls muss, wenn man den Rand von P von s aus im Uhrzeigersinn verfolgt (Bild4.4a), t vor s erreicht werden, da sonst r und s in einer anderen Randkomponente als t liegen würden, was bei einfachen Polygone nicht möglich ist. In diesem Fall ist die am am weitesten oben liegende Ecke zwischen s und t eine Richtungsänderung (oder eine "obere Richtungsänderung").
q.e.d

4.2 Zerlegung eines allgemeinen Polygons in monotone Subpolygone

4.2.1 Der Algorithmus

Der in diesem Teil vorgestellte Algorithmus setzt die im vorigen Kapitel erklärte Trapezzerlegung ein. Er ist auf Lemma 1 begründet. Die Hauptidee ist ein Polygon P in monotone Teile zu unterteilen, alle existierenden Richtungsänderungen in P zu finden und sie dann irgendwie zu entfernen.

So ergibt sich die Frage: Wie kann man die Richtungsänderungen finden? Die Antwort gibt die Trapezzerlegung. Es wurde festgestellt, dass, wenn ein Polygonknoten im Inneren einer Trapezkante liegt, dann ist er eine Richtungsänderung. Diese Behauptung ist eigentlich die Verbindung zwischen der Trapezzerlegung und den monotonen Polygonen. Die Knoten, die im Inneren einer Trapezkante liegen sind Start– oder Ende–Knoten (im vorigen Kapitel erklärt) von den Fällen 1b und 3a in dem bekannten Tapezidierungs-Algorithmus. Benennen wir die Knoten in Fall1b *untere Richtungsänderungen* und diejenigen aus Fall3a *obere Richtungsänderungen.*

Nachdem man die Richtungsänderungen gefunden hat, will man sie auf irgendeine Weise entfernen. Dies kann folendermaßen geschehen: Wenn jeder der unteren oder oberen Richtungsänderungsknoten entsprechend nach unten oder oben mit dem nächsten Polygonknoten durch eine Diagonale verbunden ist, teilen alle so erzeugten Diagonalen das Polygon in y–monotone Subpolygone. Dies folgt aus dem oberen Lemma, da alle *Richtungsänderungen* durch das Einfügen dieser Diagonalen verschwinden. Z.B wird im Bild unten die untere Richtungsänderung 9 durch das Einfügen der Diagonale *d(11,9)* aufgelöst und die obere Richtungsänderung 2 durch die Diagonale *d(2,7).*

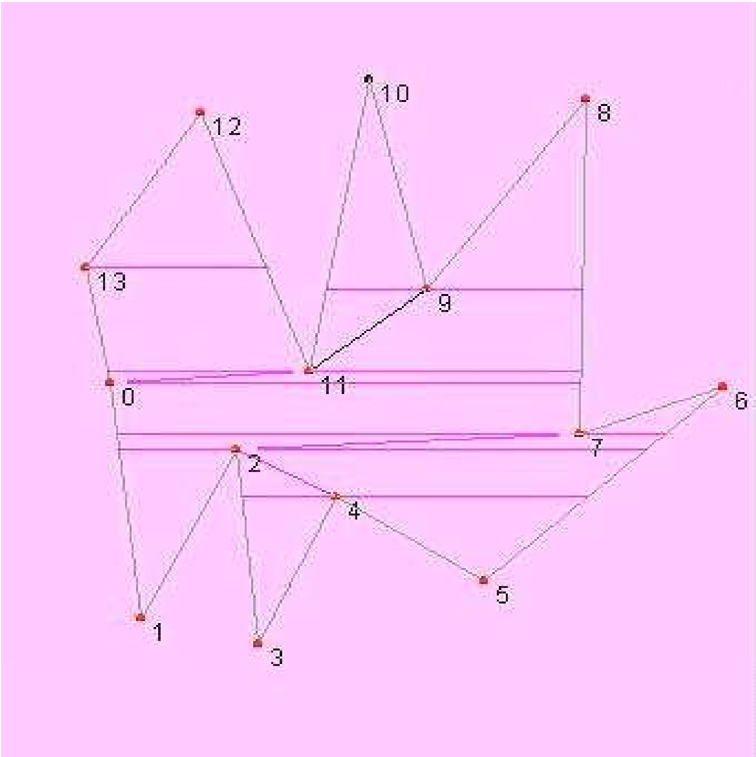

Bild4.5 Der untere Richtungsänderungsknoten 11 wird von der Diagonale d(11,0) aufgelöst. Der obere Richtungsänderungsknoten 4 wird von d(4, 2) aufgelöst, wobei 2 auch ein Richtungsänderungsknoten ist und von d(2,7) aufgelöst wird und so weiter.

Die bisherige Diskussion soll zeigen, dass die Trapezzerlegung eines einfachen Polygons direkt zu einer monotonen Zerlegung führen kann.

Es folgt der Pseudocode des Algorithmus.

Algorithmus *"Polygonzerlegung in y–monotone Teilpolygone"*

Eingabe: Ein Einfaches allgemeines Polygon, dessen Knoten in einem Vektor gespeichert sind.

Ausgabe: Vektor Q mit den disjunkten monotonen Teilpolygonen, die das

Eingabepolygon zusammenstellen. 1. Sortiere die Polygonknoten aufsteigend

nach y–Koordinate

2. Speichere die Polygonkanten (bzw. das Polygon) in eine *doppelt verkettete lineare Liste L.*

3. Führe Trapezzerlegung aus. In Fall 1b und Fall 3a, füge die neuen Diagonale in L ein und speichere sie in einen Vektor V.

4. **for** jede Diagonale d aus V **do:**

4.1 Finde d in L

4.2 Bilde das von d erzeugte y–monotone Subpolygon S.

4.3 Speichere S in Q.

4.4 Schneide S aus dem Eingabe–Polygon ab.

Analyse des Algorithmus: Wir haben in Kapitel 3 gezeigt, dass die Trapezzerlegung eine Zeitkomplexität *O(nlogn)* hat. Dies wird hier wegen des Einfügens der neuen Diagonale in die doppelt verkettete Liste nicht geändert, da das Einfügen in eine solche Liste eine konstante Zeit *O(1)* braucht. Die letzten ausgeführten Operationen, die alle einzelnen monotonen Teile erzeugen, sind das Finden und das Einfügen eines Elements (hier eine Kante) in die doppelt verkettete Liste. Auch das benötigt konstante Zeit. Das Speichern der Diagonalen in einen Vektor benötigt Zeit *O(1)*. Das Sortieren der Polygonknoten hat die maximale Zeit *O(nlogn)* (siehe [O'R94] p.48). Daraus folgt, dass der ganze Algorithmus *O(nlogn)*–Zeit braucht.

4.2.2 Implementierung und Visualisierung

Die Implementierungs–Klasse des Algorithmus für Polygonzerlegung in monotone Subpolygone ist *VertMonotonePartition.* Wie bei den anderen Algorithmen ruft die Methode *runAlg*() die Funktionen auf, die die Algorithmus–Schritte implementieren und visualisieren. Die Methode *TrapezoidationMPO* implementiert den Trapezidiering–Algorithmus, wobei bei jedem Richtungsänderungs–Knoten die Diagonale, die diesen Knoten "auflöst", in einem Vector (Variable *diagonalsBetweenInteriourCusps*) gespeichert werden. Mit Hilfe dieses Vektors und einer doppeltverketteten Liste baut die Methode *getAllMonotonePolygons*() die monotone Teile und gibt den Vektor mit den Polygonen zurück.

Ausführlichere Information enthält die JAVA–Dokumentation.

Da der Algorithmus die Trapezidierung einsetzt, kann der Benutzer merken, dass seine Animation der Animation der Trapezidierung ähnelt.

Welche sind aber die "besondere Ereignisse" in diesem Algorithmus, die an seiner Visualisierung teilnehmen sollen? Die interessanten Ereignisse geschehen verständlicherweise bei jeder während der Trapezidierung gefundenen Richtungsänderung. Deswegen wird dieser Typ von Knoten mit roter Farbe gekennzeichnet. Es werden auch nur diejenigen Trapezoid–Kanten aus der Trapezzerlegung gezeichnet, in deren Inneren eine Richtungsänderung gefunden wird, da nur diese eine Verbindung mit den gesuchten monotonen Teilpolygonen haben.

Die Visualisierung der Sweep–Linie zeigt auch hier (wie in der Animation der Trapezidierung) die zeitliche Entwicklung des Algorithmus. Da man sieht, wie sie über alle Polygonspunkte bis zum höchsten aufsteigt und das Zeichnen–Fenster mit transparenter Farbe auffüllt, wird deutlich, dass die Trapezidierung abgeschlossen wurde und die gesuchte Diagonale gefunden sind. Dann zeigt die Animation wie die Subpolygone in der entsprechenden Reihenfolge gebaut werden.

Für die Zerlegung in monotone Teilpolygone würde es reichen, dass die Animation die Schritte bis zum Ende der Trapezidierung zeigt bzw. bis alle Diagonalen gefunden wurden. Ich hielt es aber für sinvoll, auch den Aufbau der monotonen Subpolygone zu visualisieren, da erstens der Betrachter ein besseres Verständnis für das Ganze bekommt und auch weil dies bei der Implementierung und Visualisierung des nächsten Algorithmus eingesetzt wird (Siehe im Kapitel 1 "funktionale Struktur"), der ein monotones Polygon in Dreiecke partitioniert.

4.3 Triangulierung eines monotonen Polygons

Wie schon am Anfang des Kapitels bemerkt wurde, wird die monotone Zerlegung am häufigsten bei der Triangulierung der allgemeinen Polygone eingesetzt. Grund dafür ist eine wichtige Eigenschaft der monotonen Polygone, die im Folgenden diskutiert wird. Die Eigenschaft lautet, dass diese Art von Polygone einfach und schnell in Dreiecke zerlegt werden können und zwar gilt, dass jedes monotone Polygon, dessen Richtungslinie der Monotonie bekannt ist, in linearer Zeit "trianguliert" werden kann. Hier wird der Algorithmus für Triangulierung von streng–monotonen (siehe Definition oben) Polygonen präsentiert, der so erweitert werden kann, dass er auch für die Triangulierung von monotonen Polygonen funktioniert. Dies wird später erklärt werden.

4.3.1 Der Algorithmus

Die Hauptidee des Algorithmus ist ziemlich einfach: Es sei P ein streng–monotones Polygon mit n Knoten; Zuerst sortiert man absteigend alle Polygonknoten in linearer Zeit (mit der oben erklärten Sortierungs–Methode) und dann werden absteigend auf eine bestimmte Weise (Methode "greedy way") die produzierten Dreiecke nacheinander abgeschnitten. Diese bedeutet, dass beim "Bewegen" auf dem Rand des Polygons von oben nach unten auf der rechten oder der linken Polygonkette, man bei jedem Schritt das erste verfügbare Dreieck löschen kann. Der Grund dafür ist, dass bei jedem Schritt nach unten eine Diagonale gefunden werden kann, die ein Dreieck produziert (Dies sieht man in der Animation "TriangulierungMonP" in dem Applet). Wie wird die Diagonale gebaut? Bei jeder

Polygonencke p, wird p mit den über p liegenden Ecken verbunden, die sichtbar aus p sind.

Im allgemeinen funktioniert der Algorithmus so: Bei jeder Polygonencke p, verbinde p mit allen Knoten, die über p liegen und eine Diagonale mit p haben; Lösche das obere Teil des Polygons, das schon trianguliert ist; Wiederhole mit dem nächsten unter p liegenden Knoten.

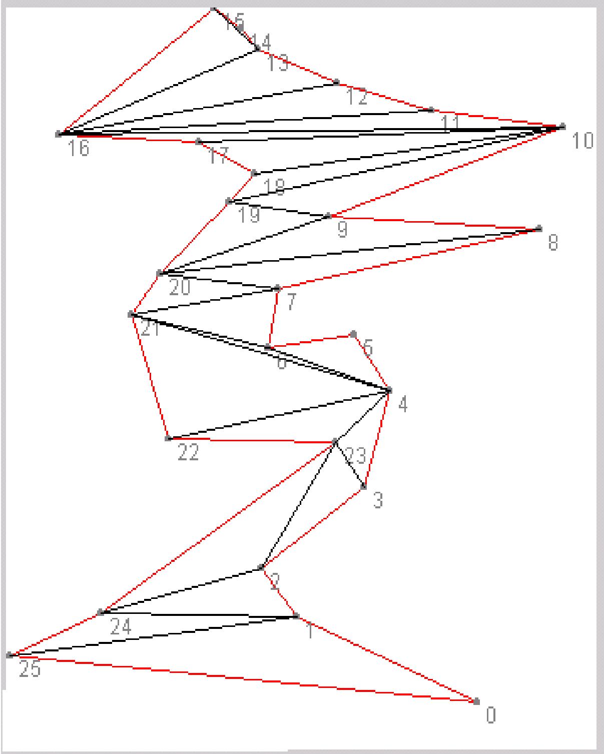

Bild4.6 Ein mit diesem Algorithmus "trianguliertes" Polygon. Das ist Ergebnis der Ausführung des Algorithmus in dem Applet.

Obwohl der Algorithmus bei der ersten Betrachtung trivial aussieht, ist dies nicht der Fall. Meine Erfahrungen während der Implementierung und Visualisierung des Algorithmus haben dies bestätigt.

Im Foldenden wird der Algorithmus ausführlicher untersucht. Es wird vorgestellt wie er genau funktioniert, welche unterschiedlichen Zustände existieren und warum er das Ergebnis schnell (d.h. in lineare Zeit) zurückliefert.

Bei jedem Iterationschritt während der Schleife über den Knoten ist es sinnvoll festzustellen, welche Struktur das Polygonteil über die in diesem Schritt behandelte Ecke p hat. Vermuten wir, dass irgendwann die Ecke p durch eine Diagonale mit einigen höheren Knoten verbunden wird. Dann behauptet man folgendes: Wenn alle mit p gebundenen Knoten auf der gleichen Polygonskette liegen, ist diese Kette "*konkav*". Sie ist "*konkav*" in dem Sinne, dass alle Ecken einen Innenwinkel größer als 180° haben.

Es wird die folgende Eigenschaft der Knoten in einem monotonen Polygon behauptet.

Lemma 2: *Die Knoten über p am Start einer neuen Iteration in dem Algorithmus für Triangulation monotoner Polygone*

A: liegen auf der gleichen Polygonkette (wobei der oberste Knoten zu den beiden Polygonketten gehört) und

B: die Kette von A ist konkav, d.h. Alle Innenwinkel sind größer als 180°.

Es gibt nun drei unterschiedliche Fälle bzgl. der Position des Knotens bei einem neuen Iterationsschritt relativ der konkaven Kette. Sie sind im Bild unten gezeigt.

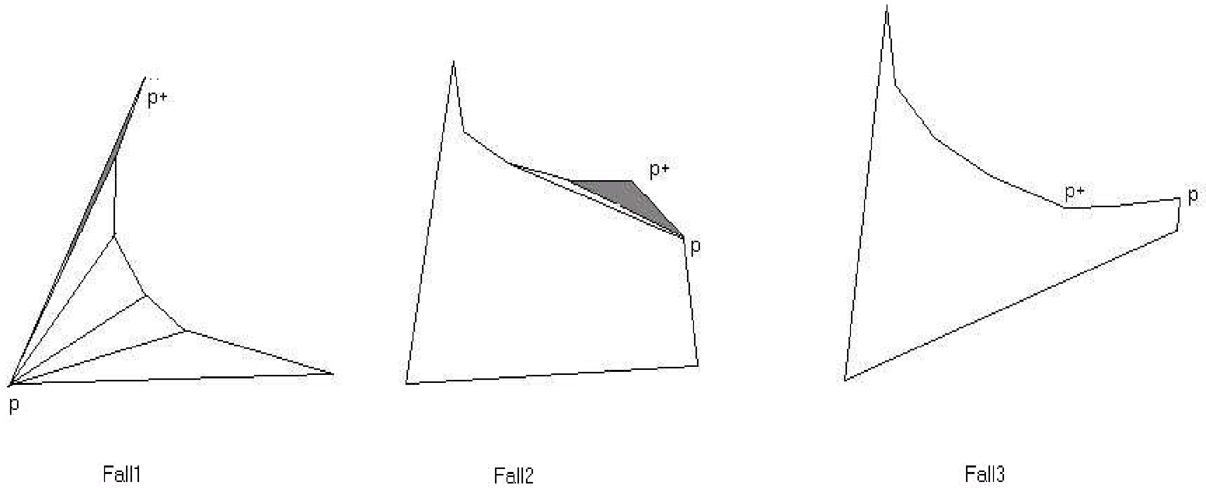

Bild4.7. Fall 1: p ist auf der gegenüberliegenden Polygonkette; Fall2: p ist auf der gleichen Polygonkette, der obere Nachbarknoten p+ ist konvex; Fall3: p ist auf der gleichen Kette, die obere Ecke p+ ist konkav oder flach.

Es existieren also zwei Grundsituationen: entweder liegt p auf der Polygonkette, auf der auch die über p liegende Knoten liegen (Fall2 und Fall3) oder liegt p gegenüber der Kette mit der höheren Knoten (Fall1). In der ersten Situation sind zwei Sub–Fälle zu unterscheiden: entweder ist der Nachbarknoten p+ über p strikt konvex (Fall2 im Bild) oder $p+$ ist konkav oder flach. In den Fällen 1 und 3 kann p mit der Spitze oder mit dem Ende der konkaven Kette eine Diagonale und so ein Dreieck produzieren. Im Fall 3 sieht p entweder die Spitze oder das Ende der konkaven Ecke, also kann in diesem Fall kein Dreieck geschnitten werden. Es soll nur p am Ende der konkaven Ecke addiert werden.

Es folgt der Pseudocode des Algorithmus.

Algorithmus *Triangulierung von monotonen Polygonen*

1. Sortiere die Polygonknoten von oben nach unten in linearer Zeit (wird oben gezeigt)

2. Initialisiere konkave Kette mit den 2 obersten Polygonsknoten.

Sei p der dritte Knoten von oben nach unten.

3. **while** p ungleich dem untersten Knotten **do**

case1: p ist auf der Polygonkette, gegenüberliegend der konkaven Kette.
1. Zeichne Diagonale von p bis zu der Spitze der konkaven Kette $p+$;
2. Entferne das oberste Ende aus der konkaven Kette.
3. **If** konkave Kette leer **then** bewege p nach unten.
case2: p ist der untere Nachbarknoten des unteren Ende $p+$ der konkaven Kette
Case2a: $p+$ ist konvex:
1. Zeichne die Diagonale von p bis zum Ende der konkaven Kette
2. Entferne das Ende der Kette.
3. **If** konkave Kette leer **then** bewege p nach unten.
Case2b: $p+$ ist konkav oder flach:
1.Füge p am Ende der konkaven Kette.
2. Bewege p nach unten.

Die unterschiedlichen Fälle sollten in der entsprechenden Visualisierung des Applets erkannt werden. Über die Visualisierung in 4.3.2.

Jetzt beschäftige ich mich mit der Frage: Warum hat der Algorithmus eine lineare Zeitkomplexität? Die Antwort wird anhand der folgenden Behauptung über die Position der Polygonknoten in jedem neuen Iterationschritt erklärt.

Nach Lemma 2 kann man sich die teuere Überprüfung sparen, ob ein Liniensegment eine Diagonale ist, da man bei gewissen Liniensegmente ohne einen Test feststellen kann, dass diese gültige Diagonalen sind.

Man speichert die konkave Kette in einem Vektor V, dessen Anfangs- und Endelement zugreifbar sind, wobei das Anfangselement das höchst und das Endelement das niedrigste Vektorelement ist. Falls p auf der gegenüberliegende Seite der höheren als p Punkte (siehe Fall 1) liegt, wird eine Diagonale von p bis zum Anfangselement gezeichnet und das daraus entstandene Dreieck weggelöscht. Falls p auf der gleichen Polygonkette mit den darüberliegenden Knoten liegt und $p+$ konvex ist (Fall2), zeichnet man eine Diagonale bis zum Endelement der konkaven Kette und löscht das daraus entstandene Dreieck. Falls p auf der gleichen Polygonkette liegt und $p+$ konkav oder flach ist (Fall3), wird p am Schluss der konkaven Kette bzw. eines Vektors addiert, da p keine Knoten über sich "sieht". Falls der Vektor der konkaven Kette leer ist, bewegt man p zu dem nächsten darunterliegenden Knoten. Die Berechnung per Diagonale dauert eine konstante Zeit. Dies bedeutet eine lineare Zeitkomplexität für den ganzen Algorithmus, da bei Polygon mit n Knoten die Anzahl der Diagonalen $n - 3$ ist. Also folgt das folgende Theorem

Theorem 4.1: *Jedes streng monotone Polygon mit n Knoten kann in linearer Zeit in Dreiecke zerlegt werden.*

Am Anfang wurde gesagt, dass der Algorithmus in dieser Form nur für streng monotone Polygone funktioniert. Mit einigen Erweiterungen kann man ihn auch für monotone Polygone brauchbar machen. Welche Änderungen müssen dafür gemacht werden? Bei den Knoten mit gleichen y–Koordinaten soll derjenige mit kleinerer x–Koordinate als höher betrachtet werden. Dies hat den gleichen Effekt wie bei der Rotation der Ebenen in der Uhrzeigerrichtung, so dass keine zwei Knoten auf einer gleichen Horizontale liegen. Der Algorithmus funktioniert also auch dann richtig, wenn zwei Knoten auf gleicher Höhe von links nach rechts betrachtet und behandelt werden.

Theorem 4.2: *Jedes monotone Polygon mit n Knoten kann in linearer Zeit in Dreiecke zerlegt werden.*

Im Anschluss an dieses Kapitel kann man zusammenfassen, wie man einen Algorithmus für die Triangulierung von allgemeinen Polygonen mit Zeitkomplexität O(nlogn) kriegt. Zuerst wurde gezeigt, dass die monotone Zerlegung mittels einer Trapezzerlegung eine Zeit O(nlogn) braucht. Dann wurde jedes monotone Teil in linearer Zeit mit dem letzten O(n) Algorithmus in Dreiecke zerlegt. Es gilt also das folgende Theorem:

Theorem 4.3 : *Ein allgemeines Polygon mit n Polygonknoten kann in O(nlogn)–Zeit in Dreiecke partitioniert werden und zwar mit O(n) Speicherplatzbedarf.*

Algorithmus (Garey et all)*: "Triangulierung eines allgemeinen einfachen Polygons"*

1.Sortiere die Polygonknoten aufsteigend nach y–Koordinate.
2.Führe Trapezzerlegung aus, um die Richtungsänderungen zu finden.
3.Teile das Polygon in monotone Teilpolygone.
4.Trianguliere jedes monotone Polygon in linearer Zeit.

4.3.2 Implementierung und Visualisierung

Die oben erklärte Triangulierung monotoner Polygone ist hier implementiert und visualisiert innerhalb der Triangulierung allgemeiner einfacher Polygone.

Die Implementierung der Triangulierung monotoner Polygone wurde in der Diskussion von Theorem 4.1 erklärt. Die Implementierungs–Klasse ist "*TriangulationTrapMon*" und stellt auch die Animation des Algorithmus dar. Da die Klasse die monotone Zerlegung implementieren soll, enthält sie manche Methode dieses Algorithmus, die aber die monotone Zerlegung hier anders (fast kaum) visualisieren. Die Methode *"TrinagulateMonotonPolygon"* implementiert und visualisiert die Schritte der Triangulierung monotoner Polygone. Falls die Eingabe ein monotones Polygon ist, wird verständlich nur diese Methode ausgeführt. Ausführlichere Informationen über die Implementierung erhält die JAVA–Dokumentation der Applikation.

In der Animation "TriangulateTrapMon" der dazugehörigen Applikation kann der Benutzer das Verhalten des Algorithmus für Triangulation allgemeiner

Polygone mit Hilfe monotoner Polygone betrachten. In dieser Animation wird die Zerlegung in monotone Teile nur allgemein visualisiert, da dies in einer eigenen Animation (siehe 4.2.1) realisiert wurde. Hier wird wieder gezeigt wie die Richtungsänderungen gefunden und mit den richtigen Diagonalen (wie in 4.2. erklärt wurde) "aufgelöst" werden. Letzteres wird hier aber nicht anhand des Pseudocodes erläutert. Danach werden die gefundene Teile und ihre lineare Triangulierung gezeichnet. In der linearen Triangulierung monotoner Polygone ist das Wichtigste, zu visualisieren wie sich das Polygon in der Iteration verändert. Der Betrachter soll bei jedem Iterationschritt die folgenden Änderungen merken können: Welche ist die gerade behandelte Ecke, in welchem Fall bezüglich der konkaven Kette befindet sich diese Ecke und welches Dreieck des Polygons wird entfernt. So kann die Animation das "greedy fashion" des Algorithmus präsentieren. Der Betrachter soll auch unterscheiden können, welches Restteil des Polygons bis zum bestimmten Iterationschritt abgeschnitten wurde. Mittels der Farbe–Technik und des Pseudocode–Panels sollte die Animation dies erfüllen. Die Polygonecke, die in dem aktuellen Iterationschritt behandelt wird, wird in rotem Kreis gezeichnet im Unterschied zu dem darüber liegenden Knoten, der in dunkelblauer Farbe gezeichnet wird. In Fall 1 wird das entfernte Dreieck in grün und in Fall 2a in hellblau gezeichnet. Da von Anfang der Animation an das Polygon in roter Farbe dargestellt ist, sieht man, welches Teil des Polygons geschnitten wurde.

Kapitel 5

Polygon–Zerlegung in unimonotone Teilpolygone

Es existiert eine Variante des letzten präsentierten Algorithmus, die einfacher und öfter für die Triangulation der allgemeinen Polygone angewendet wird. Mit der Hilfe der Trapezidierung (Kapitel 3) werden zu den Diagonalen zwischen den Richtungsänderungen auch bestimmte andere Diagonalen eingefügt, um das Polygon wieder in einfachere Sub–Teile zu partitionieren. Diese Sub–Teile werden von [FoMo84] als "*unimonotone Polygone*" (unimonotone polygons) definiert und im [O'R94] werden sie "*monotone Berge*" (*monoton mountains*) benannt. Diese geometrischen Formen können leichter als die monotonen Polygone trianguliert werden und dies auch in linearer Zeit.

Diese Gruppe Polygone haben auch andere Anwendungen außerhalb der Triangulierung. Die Zerlegung in solcher Polygone wird z.B. verwendet, wenn ein beliebiges Polygon in Teile partitioniert sein soll, die monoton bezüglich der gleichen Richtungslinie sind. Falls die Zerlegung minimal sein soll, kann der Algorithmus von [LN88] eingesetzt werden, der entsprechend in $O(nr^3+ r^{2\mathrm{nlogn}}+ r^5)$–Zeit ohne Steiner–Punkte und in $O(nr^3+r^5)$ –Zeit mit erlaubten Steiner–Punkten funktioniert. In 5.2. präsentieren wir einen von [FoMo84] verwendeten $O(nlogn)$–Algorithmus für die Zerlegung in unimonotone Polygone, der Steiner–Punkte erlaubt und eine minimale Zahl von Subpolygonen nicht berücksichtigt. Vorher möchte mich kurz mit den Eigenschaften dieser Subpolygone beschäftigen.

5.1 Definition und Eigenschaften

Anschließend verwende ich den Begriff "*unimonotones Polygon*". Die unimonotonen Polygone sind eigentlich monotone Polygone, die ein besonderes Merkmal haben und zwar enthalten sie eine Basis–Kante. Im folgenden wird gezeigt, dass die Basis eine Hauptrolle spielt, da sie der Grund der für die Triangulation wichtigen Eigenschaft ist.

Definition: *Ein* **unimonotones Polygon** *ist ein monotones Polygon, dessen eine der beiden monotonen Polygonketten ein Liniensegment ist, das die* ***Basis*** *des unimonotonen Polygons genannt wird.*

Es werden Beispiele im unteren Bild gezeigt. Die Basis ist oft, aber nicht unbedingt, die längste Polygonkante.

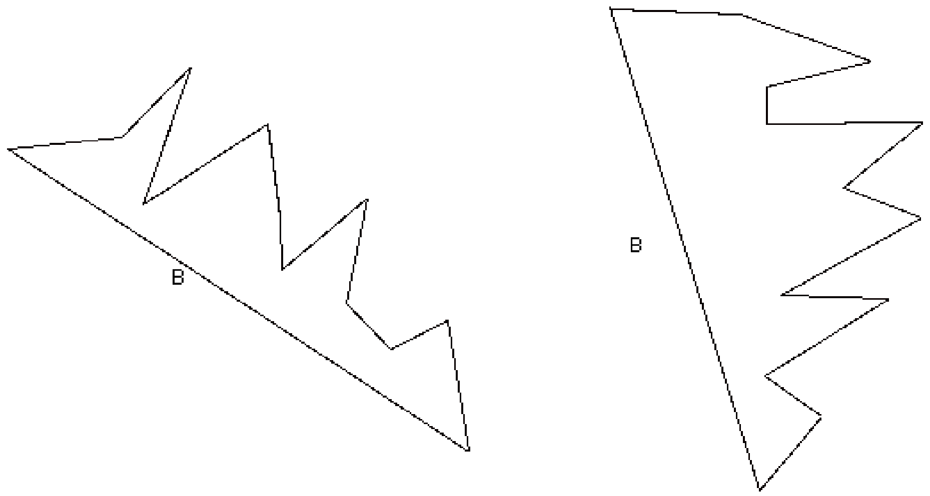

Bild5.1. Beispiele für ein horizontales und ein vertikales unimonotones Polygon. Falls die Linie der Monotonie wie beim linken Polygon horizontal ist, ähnelt das Polygon einem Gebirgszug.

Es ist wichtig zu bemerken, dass beide Endpunkte der Basis konvex sind. Falls es nicht so wäre, würde eine der beiden Ketten (die Basis) mehr als eine Liniensegment enthalten.

Die unimonotonen Polygone besitzen eine wichtige Eigenschaft, die mit Ohr–Knoten (eartips) verbunden ist. Ein Polygonpunkt v_i ist ein *Ohr–Knoten*, falls seine Nachbarknoten v_{i-} und v_{i+} eine Diagonale (v_{i-}, v_{i+}) in dem Polygon produzieren. Nun folgt die Eigenschaft:

Lemma 5.1: *Jede strikt konvexe Ecke eines unimonotonen Polygons, außer den beiden Enden der Basis–Kante, ist ein Ohr–Knoten .*

Beweis Seien *a, b, c* drei sich folgende Knoten wie im Bild 5.2. unten. Sei *b* streng konvex und kein Ende–Knoten der Basis. Sei die Linie der Monotonie horizontal. Nun ist das Ziel zu beweisen, dass *(a,c)* eine Diagonale ist. Dies zeigen wir, indem wir die Gegenbehauptung annehmen und damit einen Widerspruch erreichen.

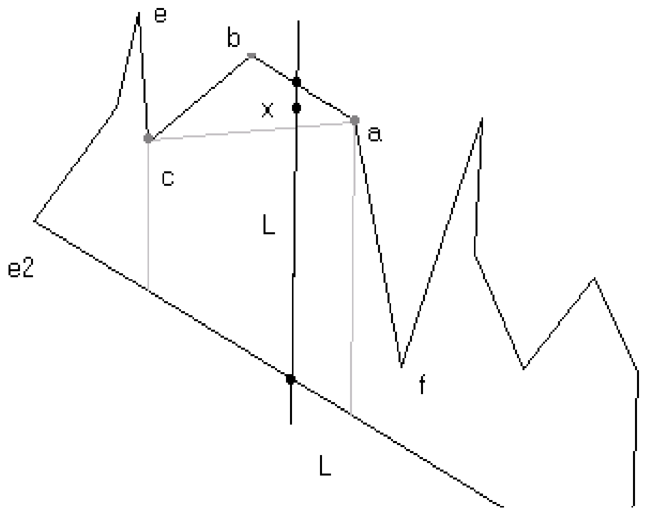

Bild5.2

Angenommen sei *(a,c)* keine Diagonale. Dann existieren zwei Fälle: *(a,c)* ist entweder außerhalb des Polygons oder *(a,c)* kreuzt die Polygonkanten. (Nach der Definition von Diagonalen)

1.Fall: Sei *(a,c)* außerhalb P und hat das Ende a wie im Bild oben (c ist symmetrisch zu a, braucht also nicht separat betrachtet werden). Falls a kein Basis–Ende ist, liegen die Kanten mit dem gemeinsamen Ende a rechts und links von a und haben das Polygon unter sich. Um außerhalb des Polygons zu liegen, muss sich das Liniensegment *(a,c)* über *(a,b)* befinden. Dies widerspricht sich der Voraussetzung, dass b konvex ist. Falls nun a das rechte Ende *e1* von Basis B ist, dann liegt entweder *(a,c)* über *(a,b)*, was zu demselben Widerspruch führt, oder *(a,c)* liegt unter der Basis B. Im letzten Fall könnte man a gar nicht mit c verbinden, weil der Punkt a über der Basis B liegen müsste. Daraus kann man schließen, dass das Liniensegment *(a,c)* immer innerhalb des Polygons liegt.

2.Fall: Das Liniensegment *(a,c)* schneidet den Rahmen dP des Polygons P. Dies verlangt, dass ein nicht konvexer Knoten x innerhalb des Dreiecks $\triangle$ *(a,b,c)* liegt. (Siehe Bild 5.2 oben). Da x intern liegt, kann sich x nicht am Rand des Dreiecks $T = \triangle(a,\ b,\ c)$ befinden. Der Punkt x kann aber auch nicht auf der Basis B liegen, da B ein Liniensegment mit zwei konvexen Enden ist. Eine vertikale Linie L durch x würde also dP in wenigstens drei Punkten treffen: ein Punkt am Rand des Dreiecks $T \cap L$, ein Punkt auf der Basis $B \cap L$ und der Punkt x. Dies wiederspricht aber der Definition eines monotonen Polygons (siehe Kapitel4). q.e.d

Die Eigenschaft aus Lemma 5.1 erlaubt eine schnelle und einfache Zerlegung in Dreiecke, da man damit sichergehen kann, dass es bei jedem konvexen Knoten eine Diagonale gibt, die ein Dreieck erzeugt.

Die Endpunkte der Basis–Kante werden aus der Voraussetzung des Lemma ausgeschlossen, da keiner von ihnen ein Ohr–Knoten ist. Im Bild 5.2 sind die Basisenden *e1* und *e2* keine Ohr–Knoten.

Es ist noch anzumerken, dass das Lemma für monotone Polygone nicht gilt, obwohl ein unimonotones Polygon nach Definition ein monotones Polygon ist. Beispiel dazu im Bild 5.3.

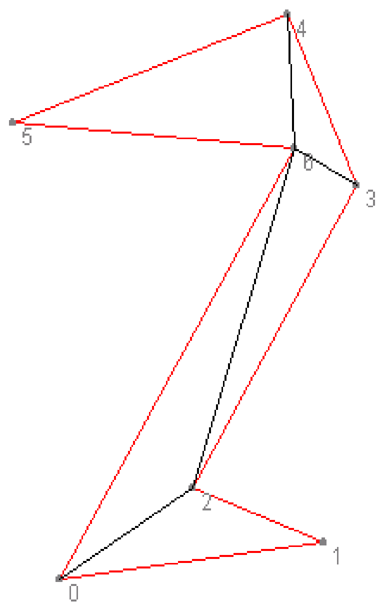

Bild5.3 Die Polygonencke mit Index 3 ist konvex, aber kein Ohr, da das Liniensegment (2,4) keine gültige Diagonale ist.

5.2 Triangulierung von unimonotonen Polygonen

Das Lemma 5.1 ergibt einen fast trivialen linearen Algorithmus für die Triangulierung dieser Polygone. Die Hauptschritte des Algorithmus sind wie folgt: Finde einen konvexen Knoten, der nicht auf der Basis liegt und ein Ohr ist; Entferne das Ohr. Wiederhole.

Zuerst möchte ich zeigen, warum der Algorithmus in linearer Zeit funktioniert. Um die lineare Zeit zu behalten, muss man zweierlei:

a) die Basis des unimonotonen Polygons in linearer Zeit identifizieren und

b) jeden nächsten konvexen Knoten ohne Suchen in konstanter Zeit finden.

Das erste ist nicht schwer zu zeigen. Die Enden der Basis sind die äußersten Polygonknoten entlang der Richtung der Monotonie. Falls z.B. die Richtunglinie vertikal ist, sind die gesuchten Knoten der oberste und der niedrigste Knoten. Da die Knoten monotoner Polygone in linearer Zeit sortiert werden können, verlangt dieser Schritt auch so viel Zeit, weil nach der Sortierung man einfach das erste und das letzte Element aus dem sortierten Vektor nimmt.

Nachdem die Basiskante identifiziert wurde, fängt man mit dem "Ohr–Abschneiden" an, ohne die Basisenden zu betrachten. Beim "Ohr–Abschneiden" werden die Ohr–Knoten gelöscht, was das das entsprechende Dreieck vom ganzen Polygon (wie im 4.3) entfernt. Es sei der Knoten b im Bild 5.2 beim "Ohr–Abschneiden" gerade gelöscht. Dann wird im Algorithmus der Konvexität–Status der Nachbarknoten a und c (siehe Bild 5.2) aktualisiert, da die Innenwinkel dieser Ecken sich verändert haben. Diese Aktualisierung ergibt ihrerseits aufgrund Lemma 5.1 die Aktualisierung des Ohr–Status der Nachbarknoten a und c.

Die Aktualisierung des Konvexitäts–Status kann man auf verschiedenen Wegen implementieren. Eine Möglichkeit ist, mit jedem Knoten seinen Innenwinkel zu speichern und dann aus den Winkeln der Nachbarknoten a und c (Bild5.2) die entsprechenden Winkel $\angle(c,a,b)$ und $\angle(b,c,a)$ nach Entfernen des Dreiecks $\triangle$(a,b,c) subtrahieren.

Eine andere Möglichkeit, die ich in der Applikation eingesetzt habe, ist, den Konvexitäts–Zustand zu aktualisieren, mittels der Überprüfung, ob ein Knoten eine linke Kurve am Rand des Polygons formt. Dies ist korrekt, denn wenn man den Rand eines Polygons gegen die Uhrzeigerrichtung verfolgt, muss man bei jeder konvexen Ecke links "abbiegen" bzw. muss eine linke Kurve machen. Falls also ein Knoten eine linke Kurve ist, muss der Knoten konvex sein. Im Bild 5.2 ist der Knoten b eine linke Kurve, wenn er links vom Liniensegment (a,c) liegt, wenn er rechts davon liegt, ist er eine rechte Kurve. Ob b links oder rechts vom Liniensegment (a,c) liegt, kann in konstanter Zeit überprüft werden, indem man die Fläche des Dreiecks $\triangle(a,b,c)$ berechnet und je nach dem, ob sie eine positive oder negative Zahl ist, die Antwort entsprechend finden. (Siehe [O'R94],S.28 und S.34) Unter der Bedingung, dass die Polygonecken gegen die Uhrzeigerrichtung gegeben sind, stellt man bei jeder Ecke in konstanter Zeit fest, ob die Ecke konvex oder konkav ist. Daraus folgt, dass für die n Knoten die Aktualisierung auch in $O(n)$–Zeit geschieht und die lineare Zeit des Algorithmus behält.

Es bleibt noch zu zeigen, wie man jeden nächsten konvexen Knoten ohne Suchen in konstanter Zeit finden kann. Dafür muss man alle konvexen Knoten in einer zyklischen Liste (z.B. die Datenstruktur doppelt verkette Liste) speichern und nach jedem Ohr–Abschneiden die Liste entsprechend aktualisieren. Aktualisieren heißt, dass je nach dem, ob die Nachbarecken konkav sind oder nicht, sie aus der Liste gelöscht oder in die Liste addiert werden. Einfügen und Löschen in doppelt verkettete Listen verlangt konstante Zeit. So lange die Liste nicht leer ist, kann man immer einfach das erste Element für das neue Dreieck aus der Liste nehmen, was bekannterweise eine konstante Zeit verlangt. Daraus folgt, dass der Algorithmus bei n Polygonpunkten in linearer Zeit funktioniert.

Es folgt der Pseudocode:

Algorithmus : *"Triangulierung unimonotoner Polygone" in $O(n)$–Zeit*

1. Identifiziere die Basisenden
2. Initialisiere den Ohr–Status jedes Knotens, der kein Endpunkt der Basis ist.
3. Speichere die konvexen Knoten bzw. die Ohren in doppelt verkettete Liste.
4. **while** die Liste nicht leer **do**:
4.1. Speichere die Diagonale *d(ac)*;
4.2. für den konvexen Knoten b, entferne das Dreieck $\triangle(a,b,c)$;
4.3. Aktualisiere den Konvexität–Status der Nachbarnecken von b;
4.4. Aktualisiere die Liste.

5.2.1 Implementierung und Visualisierung

Der Algorithmus ist in dem Applet in der Animation "TriangulierungUnimP" visualisiert, als letzter Schritt des allgemeineren Algorithmus für die Triangulierung einfacher Polygone. Falls das Eingabe–Polygon kein unimonotones Polygon ist, wird es zuerst in unimonotone Subpolygone zerlegt. Dies wird in 5.3 untersucht und in eigener Animation "UnimonotPolygone" visualisiert.

Die Implementierung des Algorithmus für die Triangulation unimonotoner Polygone wurde bei der Diskussion über die lineare Zeitkomplexität oben erklärt. Die Methode, in der die Schritte programmiert und visualisiert werden heißt *TriangulateMonotMount.* Die JAVA–Dokumentation enthält ausführlichere Information dazu.

Um das Verhalten eines Algorithmus verfolgen und verstehen zu können, muss der Betrachter in der Animation die geometrischen Objekte und deren Modifizierung nach jedem Algorithmus–Schritt sehen. Mit diesem Ziel zeigt die Animation hier die Antworten der folgenden Fragen: Welche ist die gefundene Basis–Kante nach Schritt 1, wo sind die konvexen Ecken nach Schritt 2, wo ist die gefundene Diagonale aus 4.1 und wie sieht das Restteil des Polygons nach dem Abschneiden eines Ohres bzw. Dreiecks nach Schritt 4.2 aus. Es ist auch sinnvoll das Ergebnis aus dem Schritt 4.3 zu zeigen, d.h. zu zeigen, ob die Nachbarecken konkav oder konvex sind. Die Animation zeigt die Antworten auf diese Fragen wieder mit Hilfe der auffälligen Farben und dem Kontrast zwischen den benutzen Farben. Die identifizierte Basis–Kante wird z.B. in hellgrün gezeichnet. Auch die konvexen Ecken aus Schritt 2 werden hellgrün und in hellgrünen Kreisen gezeichnet. Die gefundene Diagonale erscheint in dunkelblau. Jedes abgeschnittene Dreieck nach dem Iterationschritt 4.3. wird in hellblauer Farbe gezeigt, die sich von der schwarzen Farbe des Restteils des Polygons unterscheidet. Falls die Nachbarecke des entfernten Ohrs konkav sind, werden sie mit hellgrünen Kreisen wie die konkaven Ecken aus Schritt 2 bezeichnet. Am Bild unten kann man diese Visualisierung sehen.

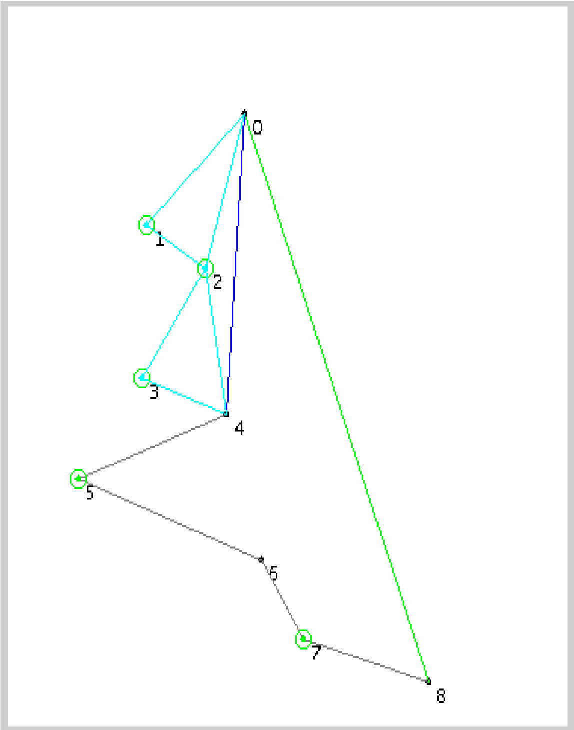

Bild5.4. Unimonotones Polygon, dessen Triangulation in dem Applet visualisiert wird.

5.3 Zerlegung in unimonotone Polygone mit Einfügen von Diagonalen zur Trapezzerlegung

Wie am Anfang des Kapitels bereits erwähnt wurde, kann die Trapezzerlegung auch benutzt werden, um die Fläche eines einfachen allgemeinen Polygons in unimonotone Polygone zu zerlegen. Wie das genau passiert, zeigt der obere Titel. Es werden einfach bestimmte Diagonale hinzugefügt. Ein Teil von ihnen sind diejenigen Diagonalen, die wir schon von der Zerlegung in monotoner Polygone kennen. Wie die anderen Diagonalen aussehen und erzeugt werden sollen, wird im Folgenden erklärt. Zuerst zeigt ein Bild die Zerlegung eines allgemeinen Polygons in unimonotone Teilpolygone. Man kann sehen, dass alle der gleichen Richtungslinie der Monotonie folgen.

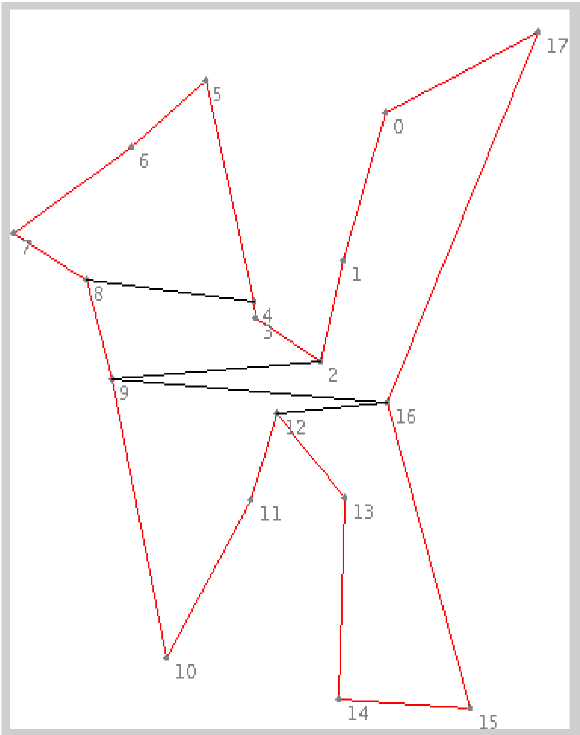

Bild5.5 Zerlegung in unimonotone Teilpolygone eines allgemeinen Polygons

Nun möchten wir anhand des Beispiels aus dem nächsten Bild erklären, wie der Algorithmus während der Trapezedierung die gesuchten unimonotonen Teile berechnet.

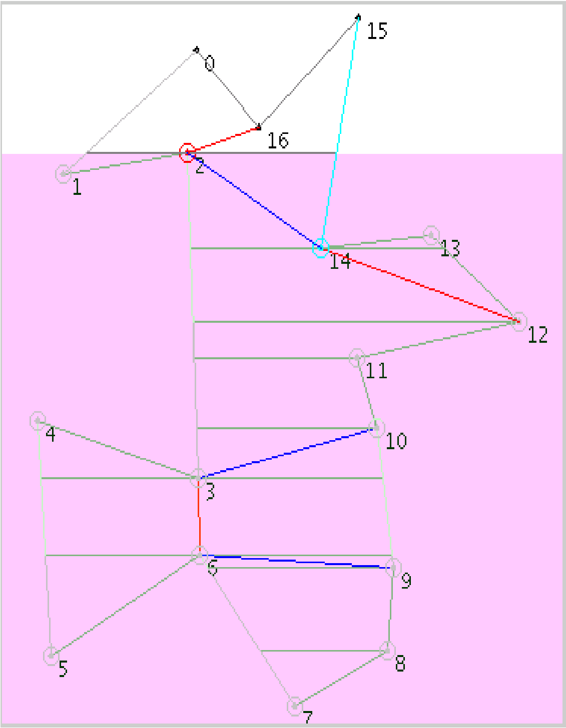

Bild5.6. Ein Polygon, das im Applet in unimonotone Polygone zerlegt wird.

Untersuchen wir nun, wie das unimonotone Polygon aus denjenigen Trapeze gebaut werden kann, die an die gemeinsame Kante *(3,2)* grenzen. Diese Kante wäre dann die Basis–Kante des vom Algorithmus gelieferten unimonotonen Subpolygons. Sei *T(i,j)* die Bezeichnung vom Trapez, dessen obere und untere horizontale Kanten entsprechend auf den Polygonecken i und j liegen. Fangen wir von unten nach oben an. Betrachten wir das Trapez *T(3,10)*, dessen linke Seite das unterste Stück der Kante *(3,2)* ist. Vielleicht wird bereits deutlich, dass das Trapez mit der Diagonale *(3,10)* durchgeschnitten werden soll, um die Konvexität des unteren Endes der Basis (der Knoten 3) zu gewährleisten. Die Trapeze *T(10,11)* und *T(11,12)* können vollständig genommen werden. Das Trapez *T(12,14)* muss mit der Diagonale *(12, 14)* geschnitten werden, um die Richtungsänderung Knoten 14 zu entfernen, und auch, um nach Lemma 5.2, die Unmonotonie bei Ecke 14 wegzuschaffen. Schließlich kommen wir zu Trapez *T(14, 2)*, das mit der Diagonale *(14,2)* geteilt werden soll, um die Konvexität des oberen Ende der Basis-Kante (der Knoten 2) zu garantieren.

Es ist, also festzustellen, dass man diejenigen Trapeze mit Diagonalen zerlegt, deren Polygonecken auf unterschiedlichen Trapezseiten liegen. Diese Behauptung wird mit dem folgenden Lemma bewiesen.

Lemma 5.2 *Sei P ein in Trapeze partitioniertes Polygon. Falls man mit einer Diagonale alle zwei Polygonecken verbindet, die zu einem Trapez gehören und auf unterschiedlichen Trapezseiten liegen, dann wird P in unimonotone Teile* *zerlegt.*

Beweis

Bild5.4 lässt klar erkennen, dass die Teile monoton sind, da keine Richtungsänderungen auf der linken oder der rechten Seiten der von ihnen umfassten Trapeze liegen. Nach Lemma 4.1 sind die Teile also monoton. Es bleibt nun zu zeigen, dass jedes Teil eine einzige Kette hat, die von einem einzigen Liniensegment dargestellt wird. Dazu betrachten wir das nächste Bild5.7

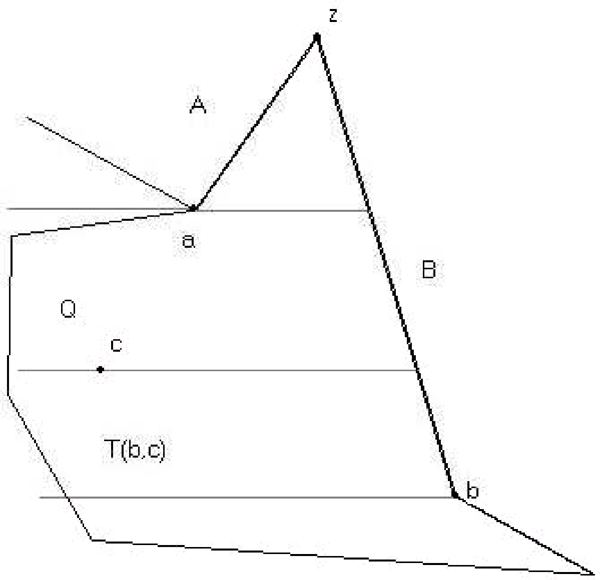

Bild5.7 zu dem Beweis vom Lemma5.2

Angenommen ein Stück Q des Teiles in Bild 5.7 hat zwei monotone Ketten A und B und jede enthält mindestens zwei Kanten. Sei z der höchste Polygonpunkt von Q, wo A B kreuzt und die Nachbarknoten a aus A und b aus B hat, so dass b unter a wie im Bild liegt. Nach der Annahme soll B mehr als eine Kette*(z, b)* enthalten. Dazu muss$b$ nicht das Ende einer über$b$ liegenden Diagonale sein. Betrachten wir jetzt das Trapez*T(b, c)*, auf dessen unteren Kante b liegt. Der Knoten*c,* der auf der oberen Horizontalen liegt, kann nicht auf *(z, b)* liegen, da c neben oder unter dem Punkt a sein muss (es kann auch $a = c$ sein). Der Punkt c liegt nicht auf der gleichen Trapezkante von *T(b,c)*, auf der auch b liegt und so gehört die Diagonale *(c,b)* der Zerlegung, was sich mit dem Annahme widerspricht, dass B sich unter b erweitert. q.e.d

Lemma 5.2. ist also die Grundlage für den Algorithmus, der ein allgemeines Polygon in unimonotone Teile zerlegen kann. Der in der Applikation implementierte Algorithmus funktioniert in folgender Weise: Starte Trapezidierung; für jeden Punkt, falls möglich, füge die Diagonalen, die das Polygon in unimonotone Teile zerlegt. Bei jedem Iterationschritt wird festgestellt, ob der auf der oberen horizontale Trapezkante liegende Knoten zu der gleichen Trapezseite gehört, zu der auch der Polygonknoten, der auf der unteren horizontalen Trapezkante liegt.

Z.B. bei dem Polygon aus Bild 5.6: bei Punkt 6 wird überprüft, ob der Polygonknoten auf der unteren horizontalen Kante von *T(9,6)*, d.h. Knoten 9, zu der Seite von 6 (Kante*(7,6)*)oder zu der gegenüberliegenden Seite (Kante*(9,10)*) gehört. Dieser Test wird unten bei der Implementierung diskutiert.

Im Bild 5.6 sind dunkelblaue und rote Diagonalen zu sehen. Die roten erzeugen monotone und die dunkelblauen unimonotone Teile. Man kann erkennen, dass noch 3 zusätzliche Diagonalen für die monotonen Teile nötig sind, um die Zerlegung in unimonotone Polygone zu erstellen. Die unimonotone Zerlegung enthält 7 unimonotone Teile im Unterschied zu den 5 monotonen Teile in der monotonen Zerlegung.

In 5.2 und 5.3 haben wir einen zweiten *O(nlogn)*–Algorithmus für die Triangulierung präsentiert, der zusammengefasst in folgenden Hauptschritten funktioniert: Trapezzerlegung; Addiere die entsprechende Diagonale nach Lemma 7.2; Trianguliere jedes unimonotone Teilpolygon mit dem Algorithmus aus 5.2. Wie oben schon erwähnt wurde, ist dieser in der Animation “TriangulationUnimP” in der Applikation visualisiert. Der zweite Hauptschritt bzw. die in diesem Teil untersuchte Zerlegung ist in der eigenen Animation “UnimonPolygonen” gezeigt. Wie er visualisiert wurde, wird im nächsten Teil erklärt.

5.3.1 Implementierung und Visualisierung

Der Algorithmus ist in der Klasse *VerMonotonMountains.java* implementiert. Für die Implementierung verwende ich die Methode *TrapezoidationMM()*, die ein Polygon mit der Hilfe der Trapezidiering in monotone Teile partitioniert. Diese speichert und zeichnet die gefundene Diagonale bei den Start– und Ende–Knoten (aus Fall 1 und Fall 3 siehe Trapezidierung) zwischen der Richtungsänderungen. Hier werden bei den regulären Knoten aus Fall 2 und den Ende–Knoten aus Fall 3 neue Diagonalen hinzugefügt. (Siehe Beispiel im Bild5.8 unten) Bei den Knoten aus Fall 1 werden keine neuen Diagonalen für unimonotone Teile zugefügt.

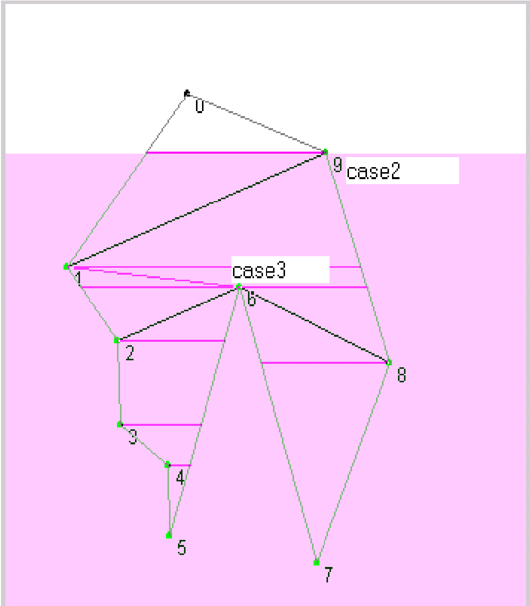

Bild5.8 Die unterschiedlichen Fällen, in denen Diagonalen für unimonotone Zerlegung hingefügt werden.

Der Pseudocode:

1. Fange Trapezzerlegung an;
2. IF Polygonecke p Richtungsänderung in Fall 1 THEN
2.1 Speichere die p auflösende Diagonale in Vektor v;
3. IF Polygonecke p in Fall 2 auf Trapezseite x THEN
3.1 IF der Anfang q der anderen Trapezseite y sichtbar aus p THEN
3.2 Speichere die Diagonale zwischen p und q in v;
4. IF Polygonecke p ist Richtungsänderung in Fall 3 THEN
4.1.Speichere die p auflösende Diagonale in Vektor v;
4.2. DO Schritt 3.1 mit p im Trapez links von p;
4.3. DO Schritt 3.1 mit p im Trapez rechts von p;

Wie im Pseudocode steht, überprüft man, ob eine neue Diagonale zwischen dem Punkt aus case2 (oder case3) und dem unteren Ende der gegenüberliegenden Trapezseite hinzugefügt werden soll. Dafür wird getestet, ob die Punkte auseinander sichtbar sind. Dieser Test wird in dieser Version der Applikation durch zwei Hilfe–Tests realisiert. Der erste überprüft, ob das Liniensegment zwischen den Punkten eine gültige Diagonale im Polygon ist (Methode*diagonal*() aus package von Gosper). Der zweite überprüft, ob das Liniensegment keine der bis jetzt gespeicherten horizontalen Trapezkanten schneidet. Der Test ist korrekt, braucht aber mehr Zeit als die Zeit, die der ganze Algorithmus benötigt. Grund dafür ist die Implementierung des ersten Hilfe–Tests, der selber O(n)–Zeit braucht. Das soll aber unser Hauptziel, die Visualisierung des Algorithmus, nicht beeinträchtigen.

Die Visualisierung zeigt die interessanten Ereignisse bei jedem Iterationsschritt, der die in roten Kreis gezeichneten aktuellen Knoten bearbeitet.

Die Überprüfung für die nächste gesuchte Diagonale in den Algorithmus–Schritten 3 und 4 (siehe Pseudocode oben) wird auf folgende Weise visualisiert. Die gegenüber dem roten Knoten liegende Trapezseite und ihr unteres Ende, die bei der Überprüfung teilnehmen, werden in der auffallenden hellblauen Farbe gezeichnet. Ist das Ergebnis positiv bzw. eine neue Diagonale gefunden wird, wird sie dunkelblau gezeichnet. Der Betrachter kann sie so von den roten Diagonalen aus der monotonen Zerlegung unterscheiden.

Kapitel 6

Algorithmen für lineare Triangulierung

Wie schon anfangs in Kapitel 2 gesagt wurde, ist die Triangulierung die bekannteste und am häufigsten verwendete Polygonzerlegung, die einen wichtigen Platz in der algorithmischen Geometrie hat. Die Berechenbarkeit des Problems war schon früher von großer Bedeutung. Sie blieb aber lange Zeit ungelöst.

Innerhalb 13 Jahren wurde in der Computational Geometry–Gesellschaft aktiv nach einer linearen Lösung des Problems gesucht. Schließlich entdeckte 1991 Chazelle ein deterministisches Verfahren [Cha91] mit linearer Laufzeit. Dieses Verfahren wird zusammenfassend in 6.2 diskutiert.

Obwohl dieser Algorithmus die Antwort auf eine langeoffene Frage war, blieb das Problem, einen einfachen, schnellen und praktischen Algorithmus zu finden. Es wurden einige Algorithmen entdeckt, die Kandidaten für Lösungen dieses Problems waren. Einer von ihnen war ein randomisierter $O(nlog^*n)$ Algorithmus [Sei91], der 1991 von Seidel präsentiert wurde (die Notation $O(nlog^*n)$ ist in 6.2 erklärt). Er verwendet das randomisierte Verfahren, das oft in der algorithmischen Geometrie effektive Lösungen bringt. Der Algorithmus wird in Teil 6.3. vorgestellt.

Die zwei obengenannten Algorithmen ([Cha91] und [Sei91]) werden in diesem Kapitel diskutiert, da sie einerseits bekannte Algorithmen in der algorithmischen Geometrie sind und andererseits, weil sie auf den in vorigen Kapiteln untersuchten Algorithmen basieren.

6.1 Kurze Geschichte der Triangulierungs–Algorithmen

Der erste Algorithmus [Len11] zur Zerlegung der Fläche eines einfachen Polygons in Dreiecke wurde 1911 vorgestellt und benötigte Zeit $O(n^2)$.

Einer der ersten Erfolge der algorithmische Geometrie war der $O(nlogn)$–Algorithmus (in Kapitel 4) für die Zerlegung in monotone Teilpolygone [GJPT78], der von Garey 1978 veröffentlicht wurde .

Danach blieb die Frage, ob die Zeitkomplexität *O(nlogn)* für die Triangulation optimal ist, lange unbeantwortet und brachte eine Reihe intelligenter Algorithmen hervor. Manche davon haben die *O(nlogn)*–Zeitgrenze unterboten, aber dies geschah nur in bestimmten Sonderfällen und die benötigte Zeit beim "worst case" blieb *O(nlogn)*.

1984 wurde ein Algorithmus [ChaIn84] von Chazelle und Incerpi präsentiert, der Zeit $O(nlogc_p)$ brauchte, wobei c_p einen "shape"–Parameter beschreibt. Der "shape"–Parameter ist eine Zahl kleiner als n (Anzahl der Polygonpunkte) und hängt von der Form des Polygons ab. Die Namen der anderen Algorithmen dieser Reihe sind in [O'R94],Seite57 zu finden.

Tarjan und Van Wyk haben 1988 nach einem Jahrzehnt einen "Durchbruch" mit ihrem *O(nloglogn)*Algorithmus [TaVW88] erreicht. Dann folgten zwei *O(nlog*n)* Algorithmen. Das einfachere Verfahren [KKT90] von ihnen, das gerundete Integer–Koordinaten verwendet, erreichte 1990 auch diese Zeitschranke. In der Zwischenzeit präsentierte Clarcson 1989 einen randomisierten Algorithmus mit erwarteter Zeit *O(nlog*n)* [CTV89]

Danach hat 1991 Chazelle mit seinem Algorithmus bewiesen, dass *O(nlogn)* nicht die optimale Zeitkomplexität für die Triangulierung ist, sondern *O(n)*.

6.2 Zusammenfassung des linearen Trangulierungs–Algorithmus von Chazelle

Die vom Algorithmus berechnete Hauptstruktur ist die *Sichtbarkeitskarte* (visibility map) des Polygons, die eine Trapezzerlegung ergibt, da sie aus allen horizontalen Erweiterungen der Endpunkte einer Polygonkette besteht (Beispiel in Bild 6.3.1). Wenn diese Kette ein gültiges einfaches Polygon bildet, liegen die horizontalen Liniensegmente innerhalb und außerhalb des Polygons.

Der Algorithmus von Chazelle setzt das *Divide-and–Conquer*–Verfahren ein. Das Polygon mit n Punkten wird in zwei Polygonketten aufgeteilt, die jeweils $n/2$ Punkte enthalten. Diese Polygonketten werden auch in 2 Ketten mit $n/4$ Punkte aufgeteilt, die auch aufgeteilt werden und so weiter. Nun wird die Sichtbarkeitskarte des Polygons gebaut, indem die Sichtbarkeitskarten aller Subketten zusammengelegt werden. Dieser ganze Prozess braucht Zeit *O(nlogn)*.

Die Hauptidee von Chazelle ist, die Zeit dieses Vorgehens zu verbessern, indem er es in zwei Phasen ausführt. In der ersten Phase werden die Sichtbarkeitskarten nur so grob berechnet, dass ihre Zusammenstellung in linearer Zeit erreicht werden kann. Sie werden grob berechnet in dem Sinn, dass die horizontalen Erweiterungen mancher Punkte fehlen.

In der zweiten Phase wird die grobe Sichtbarkeitskarte zu einer vollständigen Karte verfeinert und zwar in linearer Zeit. Dann wird mit der Hilfe der vollständigen Sichtbarkeitskarte des Polygons die Trapezzerlegung (Kapitel 3) in linearer Zeit ermittelt. (Anmerkung: Zum Vergleich, die in Kapitel 3 untersuchte Trapezedierung braucht *O(nlogn)* Zeit.) Nach diesem Schritt wird aus der Trapezzerlegung das Polygon in monotone Polygone in linearer Zeit geteilt, die dann in Dreiecke partitioniert werden. In Kapitel 4 haben wir diesen letzten Schritt bzw. den Weg von der Trapezedierung bis Triangulierung schon erklärt; er ist auch im Applet visualisiert.

Die Details des Chazelle–Algorithmus sind ziemlich kompliziert und aufwändig. Er lässt sich schwer implementieren. Deswegen werden in der Praxis andere Algorithmen bevorzugt. Ein Beispiel wird im nächsten Teil vorgestellt.

6.3 O(nlog*n)–Algorithmus von Seidel

Die Tugend dieses Algorithmus liegt in der Einfachheit. Er ist praktisch und lässt sich leicht implementieren, was von den meisten o.g. Algorithmen (in 6.1 und 6.2) nicht behauptwerden kann. Wie die meisten seiner Vorgänger kann der Algorithmus ein Polygon P nicht direkt triangulieren. Er folgt dem Arbeitsweg "Trapezzerlegung $\longrightarrow$ Zerlegung in unimonotone Polygone $\longrightarrow$ Triangulierung", den in Kapitel 6 beschrieben wurde und im Applet implementiert und visualisiert ist. Die Verbesserung des Algorithmus aus Kapitel 6 ist, dass die Trapezidierung hier schneller ermittelt wird. Die Trapezidierung wird, wie von Chazelle oben, anhand der horizontalen Erweiterungen aller Polygonecken erzeugt und definiert. Wie in 6.2. schon erwähnt wurde, stellen die horizontalen Erweiterungen aller Polygonecken die ***Sichtbarkeitskarte*** des Polygons dar.

Nun folgen die Definitionen der für den Algorithmus wichtigen Begriffe.

Definition 6.3.1 *Zwei Liniensegmente in der Ebene heißen* **kreuzungsfrei***, falls ihre Schnittmenge leer ist, oder aus einem gemeinsamen Punkt besteht.*

Definition 6.3.2 *Sei* ***S*** *eine Menge von n kreuzungsfreien und nichthorizontalen Liniensegmenten in der Ebene. Für jedes Segment aus S und jeden Endpunkt p eines Liniensegments zeichnet man zwei horizontalen Strahlen, die sich nach rechts und links von p erstrecken, bis sie auf andere Liniensegmente in S stoßen. Für jeden Endpunkt eines Liniensegments bezeichnen wir zwei horizontale Strahlen, die durch p verlaufen, als* **horizontale Erweiterung** *durch p.*

Und nun eine zweite Definition des Begriffes "Trapezzerlegung", die sich wenig von der schon in Kapitel 3 gegebenen Definition unterscheidet.

Definition 6.3.3 *Die Liniensegmente in S bilden zusammen mit ihren horizontalen Erweiterungen einen planaren Graphen, den wir* **Trapezzerlegung** *von S, oder kurz T(S) nennen.*

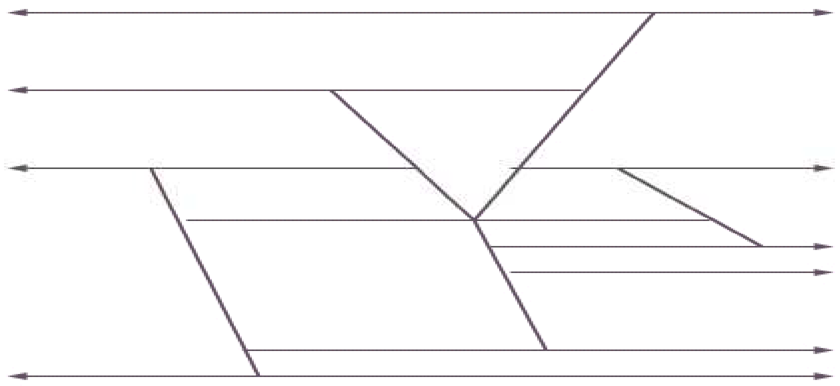

Bild6.1 Trapezzerlegung von 5 Liniensegmenten

Da jede Fläche von *T(S)* aus zwei horizontalen Seiten besteht, (wobei eine die Länge 0 haben kann) werden diese Flächen *Trapeze*genannt.

Der Algorithmus erfordert, dass jedes Trapez in *T(S)* höchstens mit jeweils zwei Trapezen oberhalb und unterhalb benachbart ist. Seidel bezeichnet ein Trapez als ***nicht degeneriert***, wenn seine Ober– bzw. Unterseite sich höchstens über zwei andere Trapezunter– bzw. Oberseiten erstrecken. Diese Eigenschaft wird automatisch eingehalten, wenn die Endpunkte der Liniensegmente in S unterschiedliche y–Koordinaten haben. Wie für die Polygonpunkte in den anderen in dieser Arbeit untersuchten Algorithmen wird auch hier angenommen, dass S diese Eigenschaften besitzt.

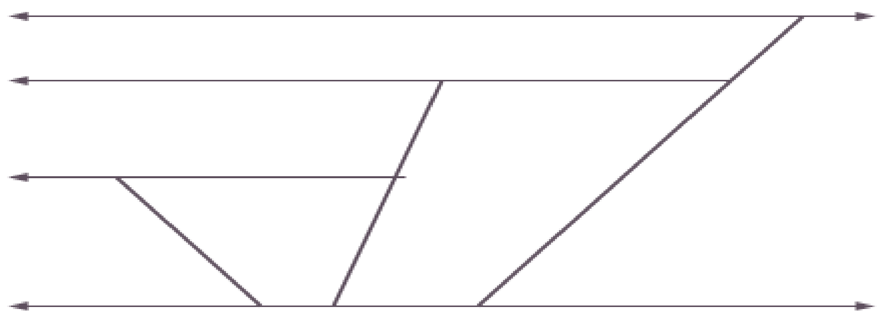

Bild6.2 Degenerierte Menge von Liniensegmenten

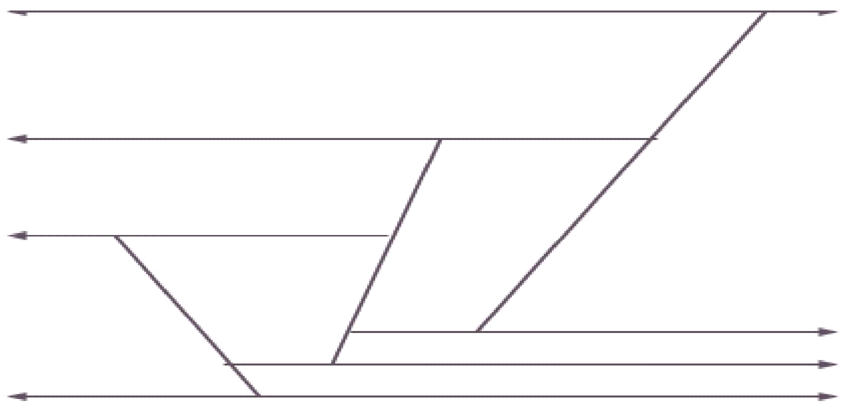

Bild6.3 Die Degeneration wurde durch Drehung oder lexikographische Anordnung behoben.

6.3.1 Der Algorithmus

Um die Trapezidierung schnell zu bekommen setzt Seidel eine Datenstruktur voraus, die für jedes Trapez t aus $T(S)$ einen Zugriff auf die benachbarten Trapeze in konstanter Zeit ermöglicht. Weiterhin wird vorausgesetzt, dass $T(S)$ nicht degeneriert ist. Dadurch ist gewährleistet, dass jedes Trapez in $T(S)$ höchstens zwei obere bzw. untere Nachbarn hat. Eine derartige Datenstruktur erlaubt uns eine effiziente Navigation durch $T(S)$, d.h. wir können eine Kurve C durch $T(S)$ günstig verfolgen. Die Kosten, die dabei anfallen, sind proportional zur Komplexität von C plus der Anzahl der Trapeze, die man bei dieser Tour durchläuft. Seidel hält fest, dass die Größe dieser Datenstruktur linear zur Anzahl der Liniensegmente in S ist. Zusätzlich führt Seidel eine Suchstruktur (Suchbaum) $Q(S)$ mit. $Q(S)$ soll helfen, für einen beliebigen Punkt p das Trapez zu finden, das p enthält. Im allgemeinen ist $Q(S)$ ein gerichteter, azyklischer Graph mit einem Wurzelknoten und genau einer Senke(Blatt) für jedes Trapez aus $T(S)$. Jeder Knoten, der kein Blatt ist, hat zwei ausgehende Kanten und ist entweder mit Etikett X oder Y versehen. Ist ein Knoten mit X gekennzeichnet, ist der Schlüssel dieses Knotens ein Liniensegment aus S. Falls der Knoten mit Y versehen ist, hat er eine reelle Zahl als Schlüssel, die für die Y–Koordinate eines Endpunktes eines Liniensegments aus S steht (Es ist die horizontale Erweiterung durch diesen Endpunkt.)

Falls man nun zu einem beliebigen Punkt q das Trapez aus $T(S)$ sucht, das q enthält, kann man wie folgt verfahren: Man beginnt an der Wurzel von $Q(S)$ und bewegt sich entlang eines Pfades bis zu einem Blatt, das bekanntlich einem Trapez aus $T(S)$ entspricht. An jedem Knoten auf dem Pfad erfolgt ein Vergleich mit dem Schlüssel des Knotens. An einem X–Knoten stellt man fest, ob q sich links oder rechts von dem Liniensegment befindet, welches der Schlüssel dieses Knotens ist. Liegt q links davon, folgt man der linken Kante. Wenn q

rechts vom Schlüsselsegment liegt, folgt man der rechten Kante. Der Fall, dass q auf dem Schlüsselsegment liegt, kann vernachlässigt werden, da dieser für unsere Zwecke nicht relevant ist. Kommt man an einem Y–Knoten an, vergleicht man die Y–Koordinate von q mit dem Schlüssel des Knotens, der bekanntlich eine horizontale Erweiterung eines Endpunktes aus S repräsentiert. Liegt q oberhalb der horizontalen Erweiterung, wird der rechten Kante gefolgt. Wenn q unterhalb der horizontalen Erweiterung liegt, wird der linken Kante gefolgt. Da man laut der Annahme ein nichtdegeneriertes System vorfindet, ist der Fall, dass q auf der horizontalen Erweiterung liegt, irrelevant. (Siehe Beispiel in Bild 6.4 unten.)

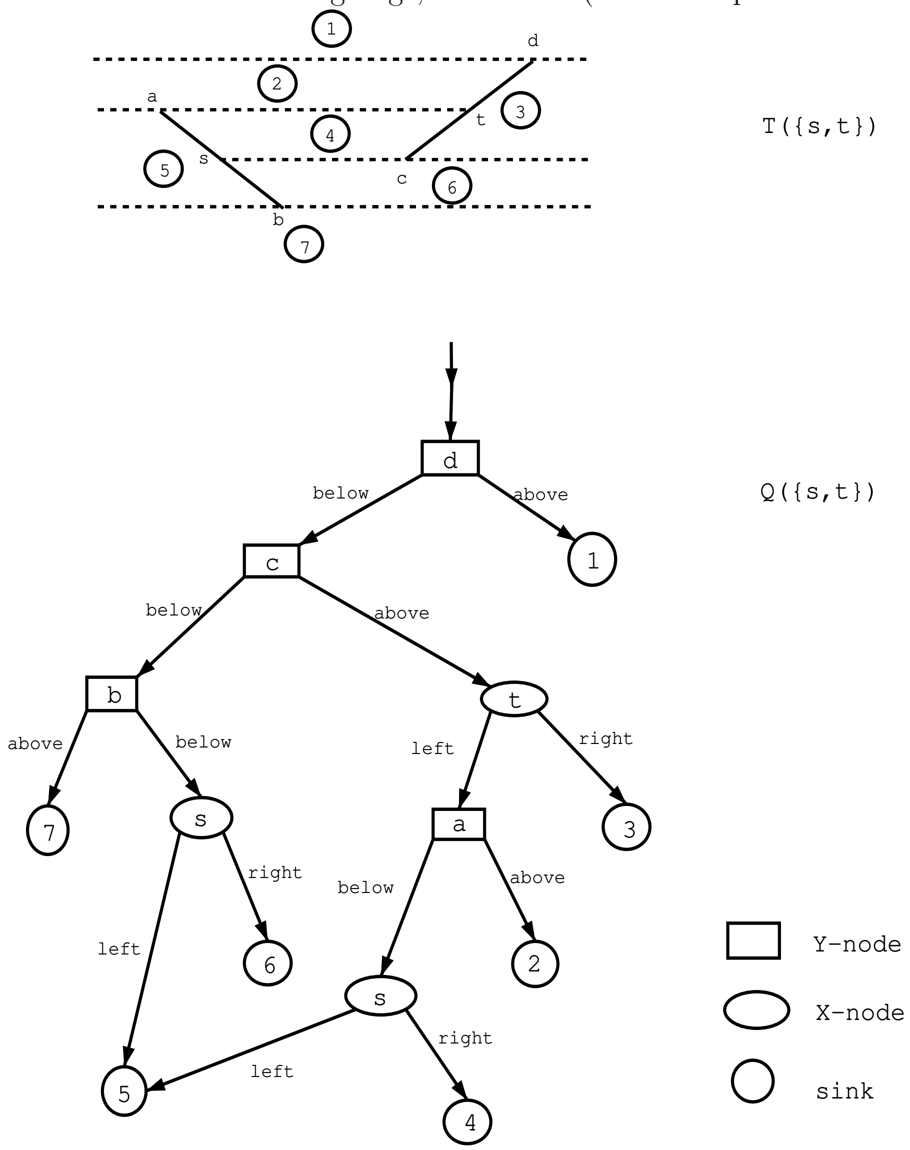

Bild6.4 : Trapezzerlegung T({s,t}) für die Liniensegmente s und t und die dazugehörige Suchstruktur Q({s,t}).

Sei nun s ein nichthorizontales Liniensegment mit dem oberen Endpunkt a und dem unteren Endpunkt b, das keine Liniensegmente aus S kreuzt. Sei

$S' = S \cup \{s\}$. Im Folgenden wird gezeigt wie der Algorithmus von Seidel die neue Trapezzerlegung *T(S')* und ihre zugehörige Suchstruktur *Q(S')* berechnet, wenn *T(S)* und *Q(S)* gegeben sind.

Falls der obere Endpunkt a kein Endpunkt der Liniensegmente aus S ist, benutzt man *Q(S)* um das Trapez T_a, welches a enthält, zu lokalisieren. Das Trapez T_a wird mit einer horizontalen Linie, die durch a verläuft, geteilt und wir erhalten damit eine neue Trapezzerlegung *T'*. Das Blatt aus *Q(S)*, das T_a entspricht, verwandelt sich zu einem Y–Knoten, dessen Schlüssel die Y–Koordinate von a ist. Die Kinderknoten sind die zwei neuen Trapeze, die man durch Teilung des Trapezes T_a erzeugt hat. Damit hat man die Strukturen *T'* und *Q'*.

Ist a ein Endpunkt eines Liniensegments aus S, setzt man *T'*=*T* und *Q'*=*Q*.

Man will nun mit dem unteren Endpunkt b ähnlich verfahren und die Trapezzerlegung *T"* sowie die Suchstruktur *Q"* aus *T'* bzw. *Q'* ableiten.

Nun "fädelt" (threading) man das Liniensegment s durch *T"*, indem man alle Trapeze bestimmt, die von s durchschnitten werden. Diese Trapeze werden in zwei geteilt und auf jeder Seite verschmilzt man die horizontalen Erweiterungen aus *T"* mit s . So erhält man *T(S').*

Die Suchstruktur *Q(S')* = *Q"* erhält man wie folgt: Für jedes neue Trapez aus *T(S')* = *T"* erzeugt man das entsprechende Blatt in der Suchstruktur. Jedes Blatt aus *Q"*, das von s zerschnitten wurde, wird zu einem X–Knoten mit dem Schlüssel s und den zwei neuerzeugten Blättern als Kinderknoten. Das Trapez aus *T"*, das b enthält, bekommt noch einen Y–Knoten, da es zusätzlich durch die horizontale Erweiterung, die durch b verläuft, geteilt wird.

Die Zeit, die man benötigt, um *T(S')* und *Q(S')* aus *T(S)* bzw. *Q(S)* abzuleiten, setzt sich aus der Zeit für die Lokalisierung der beiden Endpunkte und der "threading" Zeit zusammen. Letzteres ist proportional zur Anzahl der horizontalen Erweiterungen aus *T(S)*, die s schneiden, oder der Anzahl der horizontalen Erweiterungen aus *T(S')*, die in s münden.

Jetzt kommt die Frage: Wieviel Zeit braucht man, um *T(S)* zu erzeugen, wenn man die Liniensegmente nacheinander einfügt, beginnend mit leeren Strukturen? Offensichtlich ist dies abhängig von der Einfügereihenfolge. Zwar ist *T(S)* von dieser Reihenfolge unabhängig, *Q(S)* und die Zeit, um einen Punkt zu lokalisieren, hängen aber sehr stark von dieser Reihenfolge ab.

Künftig wird argumentiert, dass wenn die Liniensegmente in *zufälliger* (randomized) Reihenfolge eingefügt werden, wobei jede Anordnung mit gleicher Wahrscheinlichkeit vorkommt, sich *Q(S)* in Erwartung besser verhält. Dadurch wird die erwartete Zeit für die Konstruktion der Strukturen auch angemessen, die sich bekanntlich aus der erwarteten Zeit für die Lokalisierung der Punkte und der erwarteten "threading" –Zeit zusammensetzt. Die folgenden Lemmata geben Auskunft über diese Zeiten.

Lemma 1 *Sei* s_1, ..., s_n *eine zufällige Reihenfolge der Liniensegmente aus S, und* $S_i = \{ s_1, \ldots, s_i \}$ *mit* $0 \le i \le n$. *Für* $1 < i \le n$ *ist die erwartete An-*

zahl der horizontalen Erweiterungen aus $T(S_{i-1})$, die von S_i geschnitten werden, höchstens 4.

Sei $H_n = 1 + 1/2 + 1/3 + ... + 1/n$. Wir bemerken, dass $H_n = \Theta(logn)$. Genauer gilt für $n > 1$, $logn < Hn < 1+logn$.

Lemma 2 *Sei $s_1, ..., s_n$ eine zufällige Reihenfolge der Liniensegmente aus S, und $S_i = \{s_1, ..., s_i\}$ mit $0 \leq i \leq n$. Für $1 < i \leq n$ seien $T(S_i)$ und $Q(S_i)$ die Trapezzerlegung und Suchstruktur für S_i, die aus $T(S_{i-1})$ bzw. $Q(S_{i-1})$ durch Einfügen von s_i entstanden sind.*

Sei q ein Punkt, den man in $T(S_n)$ mittels $Q(S_n)$ lokalisieren möchte. Unter Berücksichtigung aller Anordnungen von S beträgt die erwartete Anzahl von Schlüsselvergleichen $5H_n$ ($O(logn)$)

Aus den Lemmata 2 und 3 und der davor geführten Diskussion folgt unmittelbar:

Theorem 1 *Sei S eine Menge von n kreuzungsfreien, nichthorizontalen beliebigen Liniensegmenten in der Ebene. Seien T(S) die Trapezzerlegung und Q(S) die Suchstruktur von S, die durch inkrementelles Einfügen der Liniensegmente aus S in zufälliger (randomized) Reihenfolge entstanden sind. Dann gilt:*

1. T(S) und Q(S) können in der erwarteten Zeit O(nlogn) aufgebaut werden.

2. Die erwartete Größe von Q(S) ist O(n).

3. Für einen beliebigen Punkt q brauchen wir in Erwartung O(logn) Zeit, um q in T(S) mittels Q(S) zu lokalisieren.

(Die Erwartungswerte gelten hinsichtlich der zufälligen Anordnung der Liniensegmente aus S, wobei jede Permutation von S mit gleicher Wahrscheinlichkeit vorkommt.)

Im folgenden wird gezeigt, dass, wenn S Liniensegmente enthält, die eine einfache polygonale Kette bzw. ein einfaches Polygon bilden, *T(S)* und *Q(S)* noch schneller aufgebaut werden können.

Der kostenintensive Teil beim Einfügen eines Segments scheint die Lokalisierung der Endpunkte zu sein. Falls nun S aus einer polygonalen Kette C besteht, kommt der Gedanke auf, die Liniensegmente der Reihe nach entlang C einzufügen. Beim Einfügen eines Liniensegments S_i erspart man sich die Lokalisierung des gemeinsamen Endpunktes von S_i und S_{i-1}, der bereits beim Einfügen von S_{i-1} lokalisiert wurde. Die Position des anderen Endpunktes kann beim "threading" ermittelt werden. Leider verzichtet diese Strategie auf Randomisierung und man kann nicht mehr von konstanten Kosten beim "treading" ausgehen.

Seidel verfolgt die folgende Strategie: Die Liniensegmente werden weiterhin in *zufälliger Reihenfolge* eingefügt. Gelegentlich hält man aber an, um alle Endpunkte in der vorhandenen Trapezzerlegung zu lokalisieren, indem man sich entlang C bewegt. Um die Effizienz dieses Verfahrens zu zeigen, braucht man zwei Lemmata. Das erste sagt, wie die gewonnenen Informationen über die zwischenzeitliche Lokalisierung der Endpunkte weiterhelfen, das zweite Lemma hingegen gibt Auskunft über die Kosten der Verfolgung von C durch die Trapezzerlegung.

Lemma 3 *Sei s_1, ..., s_n eine zufällige Reihenfolge der Liniensegmente aus S, und $S_i = \{ s_1, ..., s_i \}$ mit $0 \leq i \leq n$.*

Sei $1 \leq j \leq k \leq n$. Ist q ein Punkt, dessen Position in $T(S_j)$ bekannt ist, dann kann q mit Hilfe von $Q(S_k)$ in $T(S_k)$ innerhalb der erwarteten Zeit $5(H_k – H_j)$ $(O(\log(k/j))$ lokalisiert werden.

Lemma 4 *Sei S eine Menge von kreuzungsfreien, nichthorizontalen Liniensegmenten in der Ebene und R eine beliebige Teilmenge von S der Größe r. Sei Z die erwartete Anzahl von Schnitten zwischen den horizontalen Trapezseiten aus T(R) und den Liniensegmente aus $S \backslash R$.*

Der Erwartungswert für Z ist höchstens $4(n-r)$, wobei die Erwartung sich auf alle Untermengen R von S mit $|R| = r$ bezieht.

Bevor der Algorithmus präsentiert wird, muss noch eine Notation eingefügt sein. Für $n > 0$ bezeichnet ***log*n*** die größte natürliche Zahl l, so dass $log^l n \geq 1$, und für $n > 0$ und $0 \leq h \leq log*n$ sei ***N(h)*** die Abkürzung für $\lceil n/log^l n \rceil$ Oder mit einfachen Worten ***log*n*** bezeichnet, wieviele Male der Logarithmus iteriert werden muss, um n zu 1 oder < 1 zu reduzieren. So z.B. für $n = 2^{2^{16}} = 10^{19728}$, $log^* 2^{2^{16}} = 5$, weil $log 2^{2^{16}} = 2^{16}$, $\log 2^{16} = 16$, $\log 2^4 = 4$, $\log 2^2 = 2$ und $\log 2 = 1$. Welche Größenordnung hat log*n ? Für "praktische" Werte von n kann log*n als Konstante angesehen werden. Z.B. $log^* n \leq 5$ für alle $n \leq 10^{20000}$.

Nun wird der Pseudocode des Algorithmus angegeben. Die Eingabe für den unteren Algorithmus ist ein einfaches Polygon P mit einfacher polygonaler Kette C, bestehend aus n aufeinanderfolgenden Liniensegmenten entlang C.

1. Erzeuge s_1, ..., s_n eine zufällige Reihenfolge der Liniensegmente aus C.

2. Erzeuge die Trapezzerlegung T_1 und die dazugehörige Suchstruktur Q_1 für die Menge $\{s_1\}$.

3. **for** h = 1 **to** log*n **do**

3.1 **for** N(h1) < i <= N(h) **do**

3.1.1 Ermittle die Trapezzerlegung T_i und die Suchstruktur Qi aus $Ti–1$ und $Qi–1$, durch Einfügen des Liniensegments s_i .

3.1.2.Verfolge C entlang $T_{N(h)}$, um für jeden Endpunkt der noch nicht eingefügten Liniensegmente das zugehörige Trapez in $T_{N(h)}$ zu bekommen.

4. **for** $N(log*n) < i \leq n$ **do**

4.1. Ermittle die Trapezzerlegung T_i und die Suchstruktur Q_i aus T_{i-1} und Q_{i--1}, durch Einfügen des Liniensegments s_i.

Analyse: Man nimmt an, dass Schritt 1 in linearer Zeit ausgeführt wird. Schritt 2 benötigt konstante Zeit. Für Schritt 3 betrachtet man die h, mit $1 \leq h \leq log*n$. Lemma 4 sagt aus, dass Schritt 3.2. in der erwarteten Zeit $O(n)$ ausgeführt werden kann. Wieviel Zeit braucht man für Schritt3.1? Die erwarteten Kosten für 3.1.1 setzten sich aus den erwarteten Kosten um die Endpunkte von s_i zu lokalisieren und den erwarteten Kosten für das "threading" von s_i durch T_{i-1}

zusammen. Durch Lemma 1 weiß man, dass Letzteres konstante Kosten verursacht. Da die Endpunkte von s_i in $T_{\mathrm{N(h-1)}}$ schon lokalisiert sind, durch Lemma 3 weiß man, dass die Lokalisierung in Erwartung nur *O(log(i/N(h–1)))* Zeit benötigt. Da $i \leq n$ und $N(h-1) = \lceil n/log^{\mathrm{h-1}}n \rceil$, liegt dieser Wert in $O(log^{\mathrm{h}}n)$. Für einen festen Wert von h wird der Schritt 3.1.1 höchstens $N(h) = \lceil n/log^{\mathrm{h}}n \rceil$ Mal ausgeführt. Zusammengefasst verursacht Schritt 3 lineare Kosten. Da Schritt 3 insgesamt *log*h* mal ausgeführt wird, ergibt sich die erwartete Laufzeit von *O(nlog*n)*.

Schritt 4 lässt sich ähnlich wie Schritt 3.1 analysieren. Es ist dabei zu beachten, dass *N(log*n)* $\geq n/e$ und damit die erwarteten Lokalisierungskosten konstant sind. Folglich benötigt Schritt 4 in Erwartung *O(n)* Zeit. Damit ist folgendes bewiesen:

Theorem 2 *Sei eine Menge von kreuzungsfreien Liniensegmenten in der Ebene, die eine einfache polygonale Kette C bilden.*

*Der oben beschriebene Algorithmus berechnet die Trapezzerlegung T(S) und die dazugehörige Suchstruktur Q(S) in der erwarteten Zeit O(nlog*n).*

Die Suchstruktur benötigt O(n) Platz und erledigt Suchanfragen in der erwarteten Zeit O(logn).

Kapitel 7

Zerlegung eines Polygons in konvexe Teilpolygone

Wir betrachten nun das Problem, wie man ein Polygon in konvexe Teile partitionieren kann. Die schon bekannte Triangulierung eines einfachen Polygons kann als Sonderfall der Konvexzerlegung betrachtet werden, da ein Dreieck immer konvex ist. Es wurde bereits in Kapitel 6 gezeigt, dass ein zeitoptimaler Algorithmus für Triangulierung existiert, also gibt es auch einen zeitoptimalen Algorithmus für die Konvexzerlegung. Es ist auch offensichtlich, dass die Triangulation keine optimale Konvexzerlegung bezüglich der Anzahl der konvexen Teile ist.

Man hat nach Lösungen gesucht, die ein Polygon in möglichst wenig konvexe Teile in minimaler Zeit zerlegen. Es hat sich als relativ schwer herausgestellt, Algorithmen zum Auffinden solcher optimalen Lösungen zu finden. Die bekannten optimalen Algorithmen haben darüber hinaus relativ hohe Laufzeiten. Deswegen haben sich zwei Ansätze getrennt entwickelt.

Der erste Ansatz ist, einen schnellen Algorithmus anzuwenden, für den man bweisen kann, dass er gegenüber dem Optimum nicht allzu schlecht abschneidet. Der zweite Ansatz ist, einen Algorithmus zu benutzen, der eine relativ hohe Zeitkomplexität hat, aber optimal bezüglich der Anzahl der konvexen Teile ist. In 7.2. und 7.3. werden Algorithmen beider Ansätze untersucht, nämlich der zeitoptimale Algorithmus von Hertel und Mehlhorn [HM83] und der Algorithmus von Keil [KS98] für minimale Konvexzerlegung. Sie sind auch in der Applikation zu dieser Diplomarbeit visualisiert.

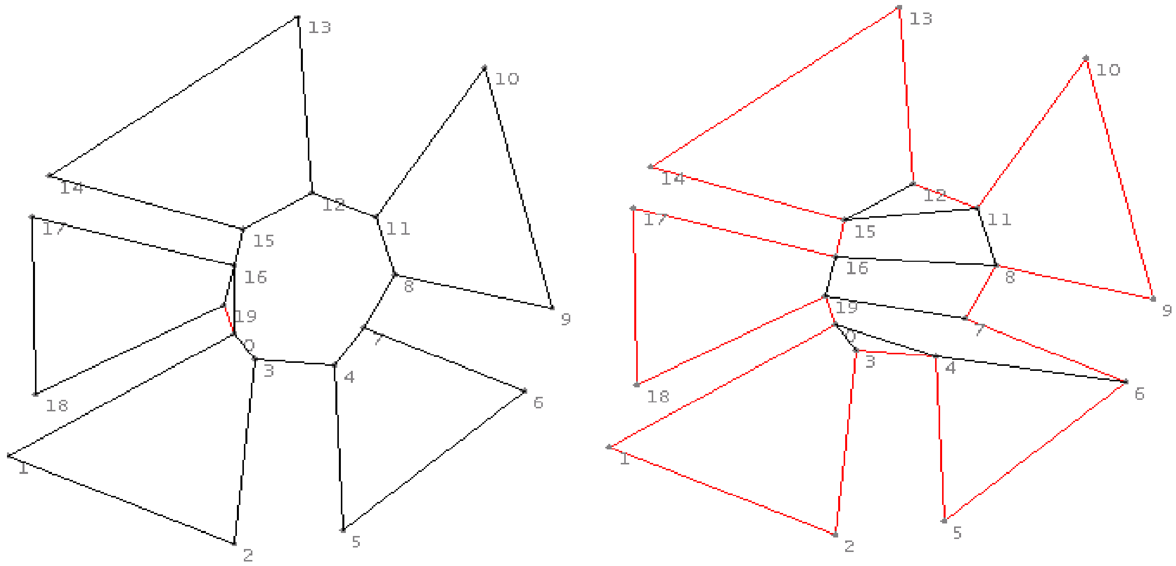

Bild7.1. Ein Eingabe–Polygon, das im Applet mit den zwei Algorithmen partitioniert wurde.

Als in Kapitel 2.2 die Polygonzerlegungen klassifiziert wurden, haben wir gesehen, dass manche Zerlegungen Steiner–Punkte benutzen dürfen. Aus diesem Grund kann man auf zwei unterschiedliche Arten die Konvexzerlegung eines allgemeinen Polygons ausführen (Siehe Bild2.3). Der erste Weg lässt zum Zerlegen nur Diagonalen zu. Der zweite Weg läßt auch Liniensegmenten zu, deren Enden keine Polygonecken, also Steiner–Punkte, sind. Zerlegungen mit beliebigen Liniensegmenten oder Zerlegungen mit Steiner–Punkten sind prinzipiell komplizierter, da diese irgendwie explizit berechnet werden müssen. Sie können aber oft effizienter sein, da es die Begrenzung, nur die Polygonecken verwenden zu dürfen, nicht gibt. Die beiden in diesem Kapitel präsentierten und visualisierten Algorithmen funktionieren nur mit Diagonalen, erlauben also keine Steiner–Punkte.

7.1 Optimum für Konvexzerlegung

Um die Effizienz einer Zerlegung schätzen zu können, braucht man die Grenzen der besten möglichen Zerlegung. Für die kleinste mögliche Anzahl konvexer Teile, in die sich ein gegebenes Polygon zerlegen lässt, gilt die folgende Abschätzung:

Satz 7.1 *Für ein Polygon P sei r die Anzahl seiner konkaven Ecken und F die kleinste Anzahl von konvexen Subpolygonen, in die sich P zerlegen lässt. Dann gilt*

$\lceil r \setminus 2 \rceil + 1 \leq F \leq r + 1$

Beweis:

Jede aufteilende Diagonale kann maximal zwei konkave Ecken verschwinden lassen. Man braucht also zum Zerlegen in konvexe Teil mindestens $\lceil r \setminus 2 \rceil$ neue Kanten und wird dann $\lceil r \setminus 2 \rceil + 1$ konvexe Teile erhalten. Siehe Beispiel im Bild7.2 unten.

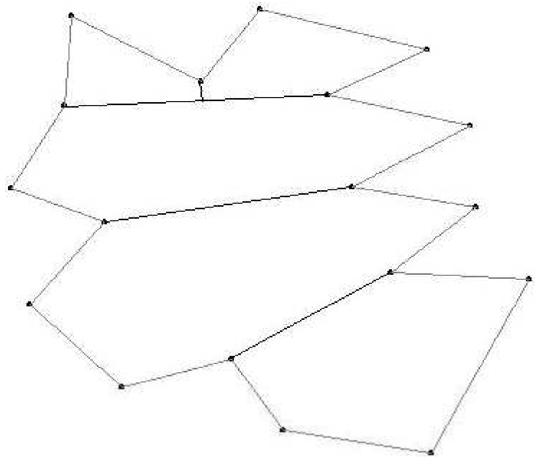

Bild7.2 $\lceil r \setminus 2 \rceil + 1$ *konvexe Teilen:* $r = 7$*, 5 konvexe Teilen.*

Unterteilt man andererseits von einer beliebigen konkaven Ecke aus entlang der inneren Winkelhalbierenden, so kann man damit induktiv eine Zerlegung mit genau r Strecken konstruieren. Dann erhält man also $r + 1$ konvexe Teile.

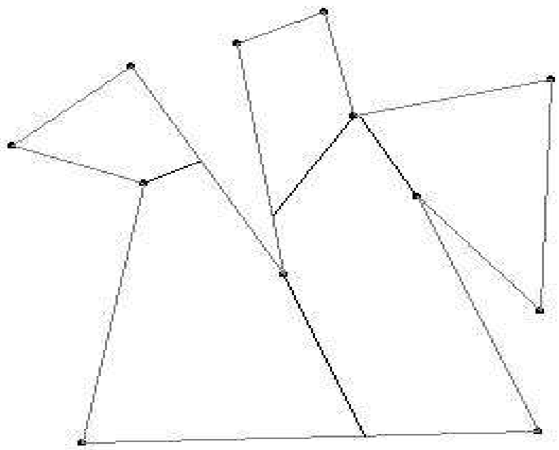

Bild7.3 r+1 konvexe Teile: $r = 4$*, 5 konvexe Teile* q.e.d

7.2 Algorithmus von Hertel und Mehlhorn

Hertel und Mehlhorn haben 1983 einen eleganten und schnellen Algorithmus [HM83] zur Konvexzerlegung mit Diagonalen vorgestellt, der das Polygon in Teile partitioniert, deren Anzahl weniger oder gleich 4 mal der Anzahl der Teile in der optimalen Konvexzerlegung sind.

Im folgenden wird der Algorithmus erklärt und dann über seine Implementierung und Visualisierung in der Applikation zu dieser Diplomarbeit diskutiert.

Für den Algorithmus ist der Begriff der “essentiellen Diagonale” wichtig.

Definition: Sei *v eine Ecke eines mit Diagonalen in konvexen Teilen partitionierten Polygons. Dann heiße eine dieser Diagonalen* **essentiell** *für v, falls ihr Weglassen ein Stück produzieren würde, das bei v eine nicht konvexe Ecke hat. Eine Diagonale, die nicht essentiell für irgendeine Ecke ist, heißt* **inessentiell**.

Dies kann offenbar nur dann passieren, wenn v einer der Endpunkte dieser Diagonalen ist und außerdem, wenn v im ursprünglichen Polygon konkav ist. Zur Erinnerung: eine Ecke ist *konkav*, falls ihr Innenwinkel größer als 180° ist im Gegensatz zu einer *konvexen* Ecke, deren Innenwinkel kleiner als 180° ist.

Der Algorithmus von Hertel und Mehlhorn besteht einfach darin, beginnend mit einer Triangulierung, so lange fortgesetzt inessentielle Diagonalen zu suchen und zu entfernen, bis alle verbleibenden Diagonalen für irgendeine Ecke essentiell sind und also als Ergebnis die Fläche des Polygons in konvexe Teile zerlegen.

Algorithmus *“Zerlegung der Fläche eines Polygons P in konvexe Teilpolygone”*

1. Starte Triangulierung von P.
2. **Do** für jede Diagonale d :
3. **If** d inessentiell **then** lösche d.

Im Schritt 3 kann für jede Diagonale lokal in linearer Zeit festgestellt werden, ob sie essentiell für eine Polygonencke oder inessentiell ist. (Wird unten bei der Implementierung gezeigt.) Die erwartete Zeit des Algorithmus hängt also von der Zeit der Triangulierung ab. Wir haben im vorigen Kapitel 6.2 gezeigt, dass ein linearer Triangulierungs–Algorithmus existiert [Cha91], woraus folgt, dass es auch einen linearen Konvexzerlegung–Algorithmus gibt. Die Autoren des Algorithmus haben eine *O(nlogn)*Triangulierung angesetzt. Hier habe ich auch eine *O(nlogn)*–Variante visualisiert, dadurch, dass ich den *O(nlogn)*–Algorithmus für Triangulation mit Hilfe der unimonotonen Polygone (Kapitel 5) verwendet habe.

Es ist bereits bekannt, dass die Konvexzerlegung nach Hertel und Mehlhorn in optimaler Zeit ausgeführt werden kann. Nun kommt natürlich die Frage: Wie unterscheidet sich die Anzahl der damit ergebenen konvexen Teilen von der Anzahl der minimalen Konvexzerlegung? Um die Antwort auf diese Frage zu finden, hilft uns das folgende Lemma.

Vor dem Lemma, möchte ich darauf hinweisen, dass man den Unterschied zwischen den beiden Verfahren durch Vergleich der entsprechenden Animationen im Applet sehen kann.

Lemma 7.1 *Für jede konkave Ecke kann es höchstens zwei essentielle Diagonalen geben.*

Beweis

Sei v eine konkave Ecke und seien $v+$ und v- ihre Nachbarecken wie im Bild 7.3 unten. Sei H- die Halbebene links der Kante *(v,v–)* und sei $H+$ die Halbebene links der Kante *(v, v+)*. Es kann höchstens eine essentielle Diagonale in $H+$ geben, weil falls es zwei gäbe, könnte diejenige, die näher zur Kante *(v, v+)* liegt, weggelassen werden, ohne ein Stück zu produzieren, das bei v eine nicht konvexe Ecke hat. Dies ist aber entgegen der Definition der essentiellen Diagonale. Analog kann es in H höchstens eine essentielle Diagonale zu v geben. Die beiden Halbebenen decken den Innenwinkel von v ab, also hat v höchstens zwei essentielle Diagonale. q.e.d

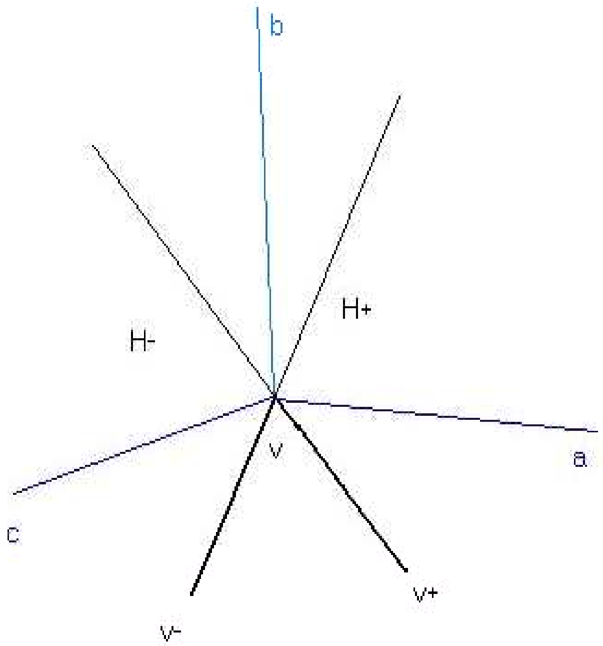

Bild7.3. Essentielle Diagonalen. Die Diagonale a ist inessentiell für den Knoten v, da b auch in der Halbebene H+ liegt. Analog ist c auch inessentiell.

Das Lemma ist ein Ausgangspunkt bei der Implementierung des Algorithmus in der Software–Applikation zur Diplomarbeit.

Nun kann man die folgende Behauptung als Antwort auf die oben gestellte Frage aufstellen:

Satz 7.2 *Der Algorithmus von Hertel und Mehlhorn zerlegt ein Polygon P in maximal 2r+1 konvexe Teile, wobei r die Anzahl der konkaven Ecken von P ist.*

Beweis

Sei M die Anzahl der konvexen Teile, die der Algorithmus liefert und sei F (wie beim Satz 7.1) die kleinste Anzahl von konvexen Teilpolygonen, in die sich P partitionieren lässt. Nach Lemma 7.1 kann es für jede konkave Ecke höchstens zwei essentielle Diagonale geben. Man benötigt also insgesamt maximal $2r + 1$

Teile, also $M \leq 2r+1$. Nach Satz7.1 gilt $F \geq \lceil r/2 \rceil$. Also $4F \geq 2r+42r+1 \geq M$. q.e.d

Satz 7.3 *Der Hertel–Mehlhorn Algorithmus zerlegt ein Polygon in höchstens viermal so viele konvexe Stücke wie eine optimale Zerlegung. Siehe Beweis von Satz 7.2*

7.2.1 Implementierung des Hertel–Mehlhorn Algorithmus

Für die Implementierung habe ich die Triangulierung mit Hilfe unimonotoner Polygone benutzt (Kapite 5), die, wie bereits bekannt, den Algorithmus für Trapezidierung (Kapitel 3) verwendet. Diese zwei Ansätze wurden schon in den entsprechenden Kapitel erklärt. Wird die Zeit der Triangulierung nicht betrachtet, kann man behaupten, dass der Algorithmus in *O(n)*–Zeit funktioniert. Der Algorithmus folgt den folgenden Schritte:

Algorithmus 1: *"Zerlegung der Fläche eines Polygons P in konvexe Teile"*

Eingabe: Allgemeines einfaches Polygon P mit n Polygonecken.
Ausgabe: Vektor v mit Diagonalen, die P in unimonotone Teile partitioniert.

1. Starte Triangulierung mit unimonotonen Polygonen.
2. Für jede im 1 gefundene Diagonale d do:
3. **If** d inessentiell **then** speichere d nicht in V.
4. **Repeat** 2 bis die Triangulierung ausgeführt wurde.

Bei jeder während der Triangulierung bzw. der Trapezidierung gefundenen Diagonale, muss man in Schritt 3 feststellen, ob die Diagonale essentiell für ihre beiden Endknoten ist. Das gilt auch während der Triangulierung jedes einzelnen unimonotonen Teilpolygons. Der Überprüfungsschritt dazu ist eigentlich der wichtigste im Algorithmus.

In der hier realisierten Implementierung verbraucht der Überprüfungsschritt *O(n)*–Zeit, die hier nicht unbedingt nötig ist, da die Triangulierung selbst *O(nlogn)*–Zeit dauert. Die lineare Zeit wäre aber entscheidend und wichtig bei einer Implementierung mit *O(n)*–Triangulierung (Kapitel 6), wo diese Zeit bei jedem Zwischenschritt einbehalten werden soll. Im folgenden wird erklärt, wie die Überprüfung ausgeführt wird, was relativ einfach ist. Wegen Lemma 7.1 ist man sicher, dass eine Polygonecke höchstens 2 essentielle Diagonalen haben kann. Aus diesem Grund können die Polygonecken Instanzen einer Object–Klasse sein, die einen Polygonpunkt mit zwei besonderen Attributen darstellt (Klasse: c*omputational_geometry.myIPoint.java*
Die besonderen Attribute beschreiben die zwei möglichen essentielle Diagonalen und können verständlicherweise null sein. Wie im Bild 7.3 gehört die erste Diagonale der Halbebene *H–* und die andere der Halbebene *H+.* Sei nun v eine der Enden der Diagonale d (z.B im Bild 7.4 ist v der Knoten X und d ist (X,Y)).

Unter der Bedingung, dass die Polygonecken Instanzen dieser Klassen sind, kann man mit dem nächsten Algorithmus den Überprüfungsschritt realisieren.

Algorithmus 2 *"Ist eine Polygon–Diagonale d essentiell für ihr Endpunkt v"*

Eingabe: Diagonale d mit Endpunkte v und q.
Ausgabe: true oder false
Notation: *dH–* und *dH+* sind die mögliche essentiellen Diagonalen von v.

Case1. **If** v konvex **then** d inessentiell für v.
Case2. **If** v konkav :

Case2.1: v hat *dH-* und *dH+:*
1. **If** d liegt in H– und $H+$ **then**
If *dH+* oder *dH-* inessentiell für ihre andere Enden **then**
d essentiell und *dH-=d* und *dH+=d;*
2. **If** Der Winkel zwischen *dH-* und *dH+* $> 180°$ **then**
2.1 **If** d in H– **then** d ersetzt *dH-* und ist damit essentiell.
2.2 **If** d in $H+$ **then** d ersetzt *dH+* und ist damit essentiell

Case2.2 : v hat nur *dH-* oder nur *dH+* oder keine
1. **If** d liegt in H- und $H+$ **then**
1.1 **If** *dH-* existiert **then** *dH+* $=d$.
1.2 **If** *dH+* existiert **then** *dH-* $=d$.
2. **If** d liegt nur in H- **then**
If *dH-* $=$ **null then** *dH-* $=d$.
3.**If** d liegt nur in $H+$ **then**
3.1.**If** *dH+* $=$ **null then** *dH+*$=d$.

Falls v konvex ist, ist d für v inessentiell. Falls v schon zwei essentielle Diagonalen hat, wird trotzdem getestet, ob d "besser" als die eine der beiden (case2.1) oder beide Diagonalen zusammen (case2.1) ist. Falls v konkav ist und keine essentiellen Diagonalen hat, wird d als eine der beiden essentiellen Diagonalen von v gesetzt, je nach dem ob d in H– oder in $H+$ liegt. Dies wird in konstanter Zeit festgestellt. Das macht die Methode i*sLeft(a,b,c)*, die in konstanter Zeit berechnet, ob der Punkt c links von dem Liniensegment *(a,b)* liegt. Wie oben schon geklärt wurde, ist die Frage, ob eine Ecke v konkav oder konvex ist, äquivalent zu der Frage "Bildet die Ecke $\angle(v-, v, v+)$ entsprechend eine linke oder rechte Kurve?". Dies kann auch in konstanter Zeit kalkuliert werden, indem man die Fläche des Dreiecks $\triangle(v-, v, v+)$ betrachtet.

Wenn d besser als eine existierende essentielle Diagonale *d'* ist, ersetzt d die Diagonale *d'* (Im Pseudocode wird dies mit *d'=d* bezeichnet). Falls *d'* inessentiell für ihr anderes Ende $q \neq p$ ist, wird *d'* aus dem Vektor mit den essentiellen Diagonalen gelöscht. Sonst bleibt sie im Vektor.

Mehr über die Implementierung ist in der JAVA–Dokumentation zu finden.

7.2.2 Visualisierung des Hertel-Mehlhorn Algorithmus

In der Algorithmus–Animation *"Konvexzerlegung"* im Applet "Visualisierung von Algorithmen für Polygonzerlegungen" werden die Schritte des Algorithmus visualisiert. Wie in den anderen Animationen geschieht dies mit Hilfe der Farbe–Technik und dem Hervorheben der entsprechenden Zeile in dem Pseudocode–Panel.

Wie bei den anderen Algorithmen, die Sweep–Line Methode anwenden, wird der Haltepunkt, wo die Sweep–Linie für die Berechnung der Diagonalen hält in rot gezeichnet. Die Diagonalen, die während der Trapezidierung genommen werden, bilden die Zerlegung des Polygons in unimonotone Polygone (Schritt 2, Algorithmus 1). Jede gebaute Diagonale d wechselt ihre Farbe entsprechend dem Fall von dem Algorithmus 2 in dem sie sich befindet und dem Ergebnis der Überprüfung. Nachdem$d$ gebaut wurde, wird sie schwarz dargestellt. Während der Überprüfung, ob sie essentiell für eins, keins oder beide ihrer Enden ist, wird sie rot. Die Überprüfung läuft zweimal nacheinander für beide Ecken von d. Diejenige Ecke v, für die gerade d getestet wird, wird im hellblauen Kreis gezeichnet. Dann werden die verschiedenen Fälle von Algorithmus 2 unterschieden.

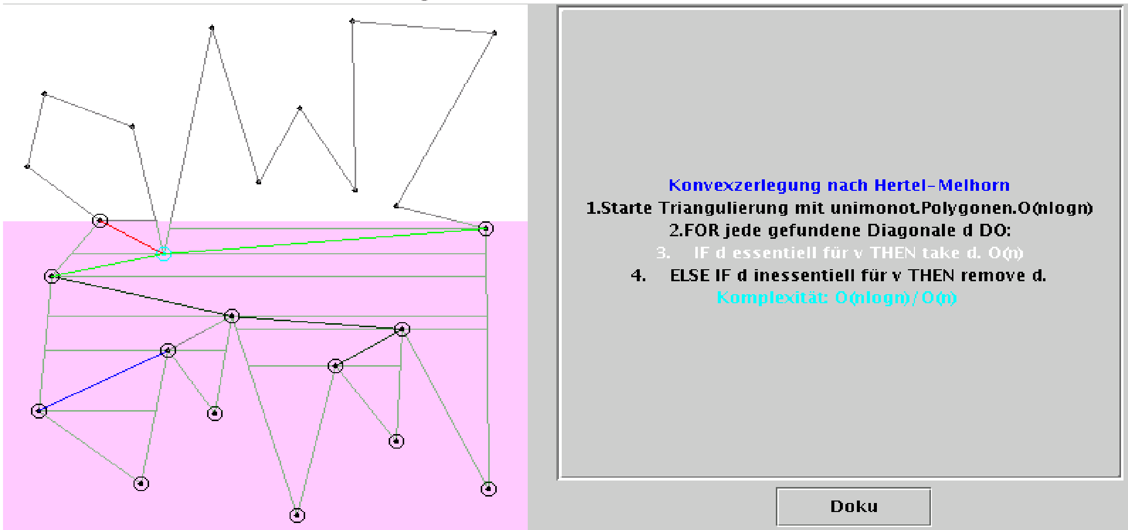

Bild7.4. Visualisierung der Schritte in der Algorithmus–Animation im Applet.

Falls v eine oder zwei essentielle Diagonalen bereits hat, erscheinen diese in hellgrüner Farbe (wie im Bild oben). Falls d in $H-$ und $H+$ gleichzeitig liegt (case2.1.1 und case2.2.1) wird sie lila. Die orange Farbe bezeichnet, dass der Winkel zwischen den essentiellen Diagonalen des hellblauen Punktes größer als 180° ist (case2.1.2). Alle diese Farben sollen auf das Ergebnis der Überprüfung hinweisen, da sie den möglichen Algorithmus–Fällen entsprechen. So sagen z.B. die orange und die lila Farben aus, dass d, noch rot gezeichnet, essentiell für v

ist, da sie besser als die grüne existierende essentielle Diagonale von v ist. Falls d nicht essentiell für v, ist also das Ergebnis negativ, dann wird d weiß gezeichnet.

Falls das Ergebnis der Überprüfung negativ ist, d.h d ist für ihre beiden Ecken inessentiell, wird sie nicht mehr gezeichnet.

Nach der Trapezidierung, also nachdem das Polygon in unimonotone Teile zerlegt wurde, beginnt die Triangulierung jedes unimonotonen Teils. Jedes unimonotone Teil wird in Dunkelgrau gezeichnet.

7.3 Optimale Konvexzerlegungen eines Polygons

In diesem Teil werden wir uns mit dem zweiten Konvexzerlegung–Algorithmus [KS98] des Kapitels beschäftigen, der weitaus zeitaufwändiger als der vorige Algorithmus ist, aber dafür optimal bezüglich der Anzahl der konvexen Teile. Das erste optimale Verfahren [Gr83] zu diesem Problem wurde von Green 1983 entdeckt und funktionierte in erwarteter Zeit $O(n^4)$. Diese Zeit wurde später 1985 von [Keil85] zu $O(r^2 nlogn) = O(n^3 logn)$ von Keil verbessert.

Der unten präsentierte Algorithmus [KS98] wurde 1989 von Keil und Sonyek in der Studie "On the time bond for convex decomposition of simple polygons" vorgestellt. Er partitioniert ein allgemeines Polygon ohne Steiner–Punkte, also nur mit Diagonalen, in minimaler Anzahl von konvexen Teilpolygonen in $O(nr^2)$–Zeit, wobei n die Anzahl der Polygonpunkte ist und r die Anzahl der konkaven davon. Dieser Algorithmus ist seinerseits also eine Verbesserung von [Keil85].

Wenn die optimale Konvexzerlegung Steiner–Punkte benutzen darf, kann sie mit dem Algorithmus [CD85] in Zeit $O(n+r^3)$ berechnet werden. Das Problem mit beliebigen Liniensegmenten, deren Ende die Steiner–Punkte sind, ist generell komplizierter zu lösen, da die Steiner–Punkte den Rand des Polygons gar nicht berühren und berechnet werden müssen.

Alle oben genannten Algorithmen setzen die Technik der *dynamischen Programmierung* ein, die hier geklärt wird.

7.3.1 Algorithmus von Keil für minimale Konvexzerlegung

Gegeben sei ein einfaches Polygon $P = \{p_1, p_1, ..., p_n\}$ mit n Polygonpunkten, die entgegen der Uhrzeigerrichtung gegeben sind. Zur Erinnerung: "einfach" bedeutet, dass die einzigen Schnittpunkte zwischen den Polygonkanten (p_i, p_{i+1}), die gemeinsamen Punkte zwischen den benachbarten Kanten sind. Hier wird die Notation d_{ij} für die Diagonale (p_i, p_j), $i < j$, verwendet.

Alle Diagonalen durch einen bestimmten Polygonpunkt p_i, können mit der Berechnung des *Sichtbarkeitspolygons* von p_i in linearer Zeit produziert werden [EL81].

Wichtig für den Algorithmus zur Berechnung des Sichtbarkeitspolygons eines Polygonspunktes als auch für den hier präsentierten Algorithmus ist die folgende Bemerkung:

Bemerkung 1**:** *Die Ordnung der Diagonalen* d_{ij_1}*,* d_{ij_2}*,...,* $d_{\mathrm{ij}_\mathrm{k}}$*, die entgegen der Uhrzeigerrichtung um dem Punkt* $\mathrm{p_i}$ *liegen, ist identisch der Ordnung, in der die Polygonecken* $\mathrm{p}_{\mathrm{j}_1}$ *,* $\mathrm{p}_{\mathrm{j}_2}$ *,...,* $\mathrm{p}_{\mathrm{j}_\mathrm{k}}$ *gegen der Uhrzeigerrichtung um p liegen.*

Es ist verständlich, dass die minimale Konvexzerlegung eines Polygons ohne Steiner–Punkte nur solche Diagonale verwenden soll, die mindestens bei einer konkaven Polygonecke enden. Eine Diagonale in P, die zwei konkave Endpunkte hat, nimmt nicht unbedingt in der minimalen Konvexzerlegung des Polygons P teil.

7.3.1.1 Konvexzerlegung mittels dynamischer Programmierung

Definition: *Jedes Problem kann in Subprobleme gegliedert werden. Die* **dynamische Programmierung** *ist ein algorithmisches Paradigma, das die beste Lösung zu dem gegebenen Problem gibt, indem es die einzelnen optimalen Lösungen der Subprobleme kombiniert.*(Siehe [Bel87])

Nun kommt die Frage: Wie kann man mit der Hilfe der dynamischen Programmierung ein einfaches Polygon in eine minimale Zahl konvexer Teile zerlegen? Keil definiert ein *Subproblem* P_{ik} *zu jeder Diagonale* d_{ik}, bei der einer der Enden p_{i} oder p_{k} *konkav ist.*

Definition. Sei p_{i}, $p_{\mathrm{i}+1}$,...,p_{k} *eine Polygonkette, die das Subproblem* P_{ik} *darstellt. Dann ist die* **Größe** *von* P_{ik} *die Anzahl der Punkte in* P_{ik} und das **Gewicht** $\mathbf{W}_{\mathrm{ik}}$ von P_{ik} *ist die minimale Anzahl der Diagonalen, die* P_{ik} *in konvexe Teile partitioniert.*

Die Autoren des Algorithmus machen die Konvention, dass die Polygonkante $d_{(0,n-1)}$ auch eine Diagonale ist. Sie berechnen die Anzahl der Diagonalen in der minimalen Konvexzerlegung eines Polygons P, indem sie das Gewicht $W_{0,n-1}$ des Subproblems $P_{0,n-1}$ berechnen, da $P_{0,n-1} = P$.

Daraus ergibt sich die nächste Frage, wie man das Gewicht eines Subproblems aus den Gewichten seiner Subprobleme mittels dynamischer Programmierung finden kann? Man findet die Antwort in den folgenden zwei Ansätzen:

1. Angenommen man kann alle Dreiecke in P_{ik} produzieren, die als Kante die Diagonale d_{ik} in Subproblem P_{ik} haben. Sei T ein solches Dreieck. Das Entfernen von T aus dem P_{ik} lässt zwei Subprobleme. Dann ist das Gewicht von P_{ik} die Summe aus der Gewichte der gebliebenen Subprobleme plus 0, 1, oder 2 Kanten von T, die später bei der Zusammenlegung der Lösungen aller Subprobleme von P eventuell genommen werden müssen, um die konkaven Polygonknoten zu vermeiden. Dieser Ansatz ist von [Keil85] und findet hier die Lösungen jedes Subproblems P_{ik}. Er wird in 7.3.1.2 zusammengefasst vorgestellt.

2. Man betrachtet jedes Dreieck T, das ein Teil der *kanonischen Triangulierung* von P sein könnte. Die minimale Konvexzerlegung verwendet immer eine oder zwei Kanten dieses Dreiecks und macht das Kombinieren (merging)

der einzelnen Lösungen der verschiedenen Subprobleme leichter zu realisieren. Trotzdem müssen viele unterschiedliche Subproblem–Lösungen gespeichert werden. Keil speichert sie in speziellen Kellern, T_{ij} oder S_{jk}, die unten erklärt werden,

Obwohl diese Ansätze als "top-down" rekursive Prozeduren oben beschrieben sind, implementiert Keil diese mit "bottom-up"–Iterationen, indem er die Subprobleme in der Ordnung ihrer steigender Größen löst und schließend das Gewicht der Konvexzerlegung von P findet. Er initialisiert nach der oben erwähnten Konvention die Gewichte $W_{(i,i+1)} = -1$.

7.3.1.2 Äquivalente Zerlegungen und engste Paare (narrowest pairs)

Da exponentiell viele Konvexzerlegungen eines Subproblems P_{ik} das Gewicht W_{ik} erreichen können, definiert und speichert Keil nur bestimmte Äquivalenzklassen von ihnen, um sie für die minimale Konvexzerlegung zu verwenden.

Man ordnet jeder Konvexzerlegung von P_{ik} ein Indizes–Paar *[a,b]* zu. Das Indizes–Paar *[a,b]* bedeutet, dass ein konvexes Teil der Zerlegung die Diagonale d_{ik} als Kante haben wird und noch die Polygonknoten *a, i, k* und *b* in Uhrzeigerrichtung enthalten wird.

Definition*: Zwei Konvexzerlegungen eines Subproblems* P_{ik} *sind* **äquivalent***, falls sie gleiche Gewichte und gleiche zugeordnete Indizes–Paare haben.*

Definition: **Engste Indizes–Paare** *sind diejenigen Paare von Indizes, in deren Zerlegungen ein solches Teilpolygon existiert, das als Kante die Diagonale* d_{ik} *besitzt und kein konvexes Teil von irgendeiner anderen Konvexzerlegung von* P_{ik} *enthält.*

Im Bild unten sind die minimalen Konvexzerlegungen des Polygons P_{09} mit ihren zugeordneten Indizes–Paaren gezeichnet. Um festzustellen, dass diese minimale Konvexzerlegungen sind, kann man darauf achten, ob alle verwendeten Diagonalen die drei konvexen Polygonecken 3, 4, und 6 entfernen. Z.B enthält die Konvexzerlegung mit dem engsten Indizes–Paar [1, 3] das konvexe Teilpolygon mit Polygonkette (0, 1, 2, 3, 9) im Gegenuhrzeigersinn, das nach der Definition kein konvexes Teil mit Kante (0,9) einer anderen optimalen Konvexzerlegung enthält.

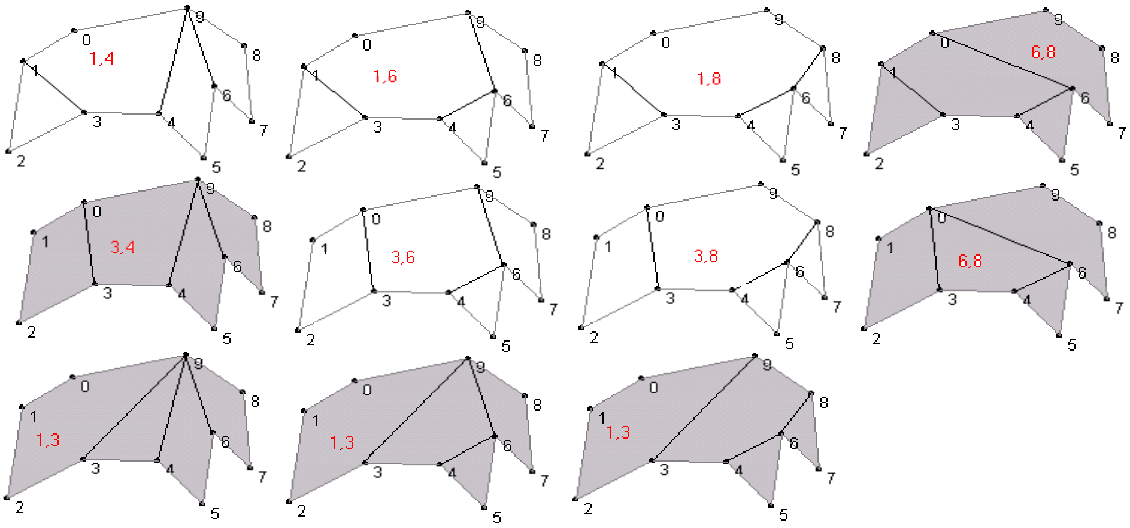

Bild7.5: Die elf minimalen Konvexzerlegungen von Polygon $P_{0,9}$ mit deren engsten Paaren, die in grauer Farbe gezeichnet sind.

Keil hat festgestellt, dass nur Subprobleme mit engsten Paaren für die Zusammenstellung der minimalen Konvexzerlegung brauchbar sind. Aufgrund *Bemerkung 1* sucht bzw. überprüft man nach den engsten Indizes–Paare von P_{ik} einfach durch Vergleichen der Indizes: man lenkt jedes Intervall [a, b] ab, das ein anderes (engeres) Intervall in sich beinhaltet.

Nachdem die beschränkten Paaren eines Subproblems gefunden wurden, werden sie in einem Keller S_{ik} gespeichert. Das Speichern in Keller wird so ausgeführt, dass die Liniensegmente bzw. die Diagonalen um p_{i} und p_{k} von unten nach oben gegen der Uhrzeigerrichtung liegen.

Zum Beispiel enthält der Stapel $S_{0,9}$ des Polygons $P_{0,9}$ im Bild 7.5, die Indizes–Paare [1,3], [3,4], und [6,8] von unten nach oben.

7.3.1.3 Kanonische Triangulierung

Der zweite Ansatz (siehe oben) für die Zusammenstellung der optimalen Konvexzerlegung von [KS98] verwendet die kanonische Triangulierung des Polygons P. Keil erweitert irgendeine optimale Konvexzerlegung von P zu kanonischer Triangulierung durch Hinzufügen von Diagonalen. Das Entfernen des d_{ik} enthaltenden Dreiecks, lässt meistens zwei kleinere (relativ der Größe) Subprobleme, wobei jedes davon eine kanonische Triangulierung hat.

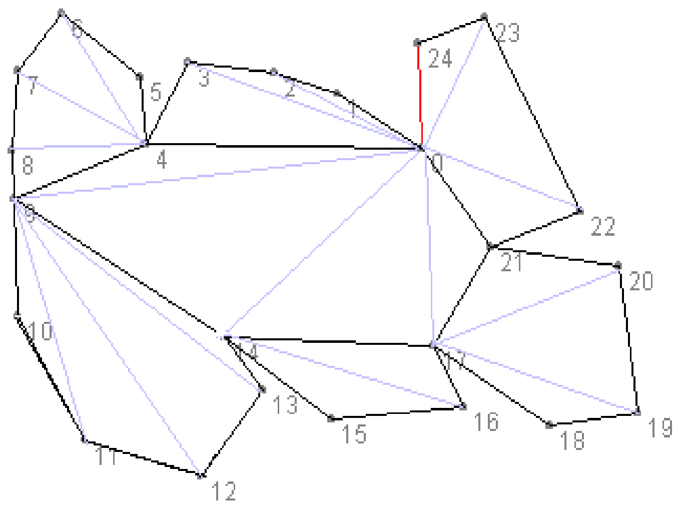

Bild7.6 Kanonische Triangulierung aus minimaler Konvexzerlegung.

Wie erweitert Keil eine Konvexzerlegung von P, bei der jede Diagonale eine konkave Ecke hat in kanonische Triangulierung? Jedes konvexes Teil C in der Konvexzerlegung von P hat mindestens eine Polygonecke, die konkav in P ist. Man kann die konkave Ecke mit dem kleinsten Index in C mit denjenigen Ecken verbinden, die größere Indizes haben(siehe Bild 7.6 oben). Falls nach diesem Schritt das konvexes Teil C noch nicht trianguliert wurde, war die Polygonecke mit dem höchsten Index konkav in P, soll also mit den anderen Polygonecken in C verbunden werden.

Wichtig für den Algorithmus sind die folgenden Behauptungen bezüglich der Diagonalen in der kanonischen Triangulierung und die Subprobleme, die sie produzieren.

Bemerkung 2*: Jede Diagonale* d_{ik}*,* $i < k$*, in der kanonischen Triangulierung eines Polygons P besitzt die folgenden drei Eigenschaften:*

1. Die Diagonalen, deren Endpunkte die Punkte entlang der Polygonslinie des Subproblems Pik sind, definieren die kanonische Triangulierung von P_{ik}*.*

2. Sei die Polygonecke p_i *konkav in P. Dann hat das* d_{ik} *enthaltendes Dreieck T(*p_i*,* p_j*,* p_k*),* $i < j < k$*, entweder Index* $j = k - 1$ *oder die Diagonale* d_{jk} *gehört zu Konvexzerlegung von P.*

3. Sei die Polygonecke p_i *nicht konkav in P. Dann muss die Polygonecke* p_k *konkav sein. Dann hat das* d_{ik} *enthaltende Dreieck T(*p_i*,* p_j*,* p_k*),* $i < j < k$*, entweder* $j = i{+}1$ *oder die Diagonale* d_{ij} *gehört zu der Konvexzerlegung von P.*

Aufgrund dieser Eigenschaften der Diagonalen einer kanonischen Triangulierung kann die optimale Konvexzerlegung so zusammengestellt werden:

1. Es werden nur diejenigen Polygonecken p_j berücksichtigt, die mit der Diagonale d_{ik} ein kanonisches Dreieck gestalten können.

2. Es wird beachtet, ob die Diagonale d_{ij} und d_{ik} in der Konvexzerlegungen

von P_{ij} und P_{jk} hinzugefügt werden müssen oder Diagonale der kanonischen Triangulierung sind. Ein Keller S_{ik} wird jedem Subproblem P_{ik} zugeordnet. Er speichert alle Äquivalenzklassen der Konvexzerlegungen von P_{ik}.

7.3.1.4 Wie wird nach der Lösung eines Subproblems gesucht?

Die Lösung eines Subproblems in diesem Algorithmus ist, das Gewicht des Subproblems zu finden. Angenommen man möchte das Gewicht W_{ik} vom Subproblem P_{ik} berechnen: Im Folgenden wird diskutiert, wie das erfüllt werden kann.

Der Algorithmus untersucht systematisch die kanonischen Triangulierungen aller optimalen Konvexzerlegungen von P_{ik} mit engsten Paaren. Dies wird mit Hilfe der Eigenschaften von Bemerkung 2 gemacht. Es wird auch angenommen, dass die engsten Paare aller minimalen Konvexzerlegungen von jedem als P_{ik} kleineren Subproblem P_{xy} zur Verfügung stehen. Diese engsten Paare werden in zwei Kellern S_{xy} und T_{xy}, entsprechend gegen die Uhrzeigerrichtung und in Uhrzeigerrichtung gespeichert. Um alle Paare der optimalen Konvexzerlegungen von P_{ik} entgegen der Uhrzeigerrichtung zu speichern, werden zwei Keller T_{ij} oder S_{jk} verwendet, je nach dem ob die Polygonecke p_{i} konkav oder konvex ist. Nachdem S_{jk} auf dieser Weise dargestellt wird, wird T_{ik}vom S_{jk} dargestellt, so dass beide für spätere Berechnungen vorhanden sind.

Im folgenden werden die zwei Methoden beschrieben, mit ihren der Algorithmus die minimale Konvexzerlegung eines Subproblems P_{ik} berechnet.

FallA. Polygonpunkt p_{i} ist konkav (Bild77a):

Die minimale Konvexzerlegung vonP_{ik}, $i < j < k$, benutzt die Diagonale d_{ik}, die Konvexzerlegung vom kleineren Subproblem P_{jk} und die Konvexzerlegung vom kleineren Subproblem P_{ij}, die wahrscheinlich durch die Diagonale d_{ij} teilnimmt. Um also das Gewicht W_{ik} von Subproblem P_{ik} zu berechnen, kann man die folgende Rekursion der dynamischen Programmierung ausführen:

$$W_{ik} = min_{i<j<k \ d_{\text{ij}},d_{\text{jk}} existieren} \begin{cases} W_{ij} + W_{jk} + 2 & : \ falls\ d_{ij}\ teilnehmen\ muss \\ W_{ij} + W_{jk} + 1 & : \ sonst \end{cases}$$

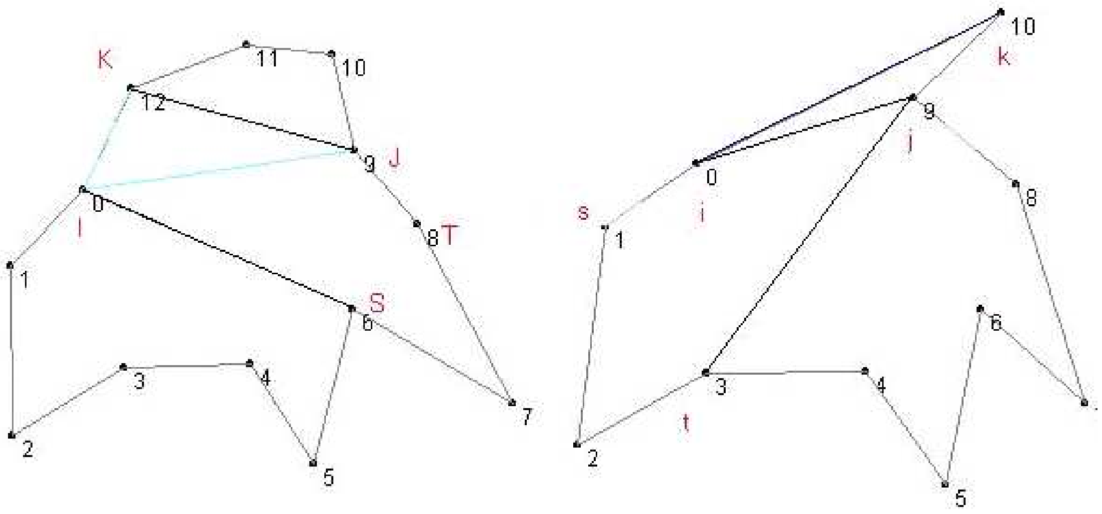

Bild7.7 a) Polygonecke p_{i} *ist nicht konkav. b)* p_{i} *ist konkav: Diagonale* d_{jk} *und vielleicht* d_{ij} *werden in der Konvexzerlegung benutzt.*

Um alle engste Indizes–Paare von P_{ik} zu bekommen, betrachtet Keil alle Polygoneckenp_j, $i < j < k$, und alle Diagonale d_{ij} und d_{jk} in dem Sichtbarkeitspolygon von p_j. Falls p_j nicht sichtbar aus p_{j} ist, gehört d_{jk} nicht zu den engsten Paare von P_{ik}.

Wie wird bestimmt, ob die Diagonale d_{ij} in der optimalen Konvexzerlegung von P_{ik} teilnimmt? Für ein gegebenes Index j werden so lange Paare aus dem Keller T_{ij} genommen, bis das letzte Paar *[s, t]* gefunden wurde (siehe Bild 7.7), in dessen Zerlegung die Diagonale d_{tj} und d_{ik} keinen konkaven Innenwinkel bei p_{j} schaffen. Falls wie im Bild 7.7a kein solches Paar existiert oder falls die Diagonalen d_{is} und d_{ik} einen konkaven Innenwinkel bei p_{j} produzieren, muss die d_{ij} in der Konvexzerlegung von P_{ik} mit Gewicht $W_{\mathrm{ik}} = W_{\mathrm{ij}}+W_{\mathrm{jk}}+2$ und engstem Paar *[j,j]* verwendet werden. Sonst(siehe Bild7.7b), bekommt man die Konvexzerlegung von P_{ik} mit Gewicht $W_{\mathrm{ik}} = W_{\mathrm{ij}}+W_{\mathrm{jk}}+1$ und engstem Paar *[s,j]*.

Es ist nicht schwer den Keller S_{ik} (gegen die Uhrzeigerrichtung) mit den engsten Paaren von P_{ik} zu erstellen, da das zweite Element des Paares der Iterations–Index der Schleife ist. Ein Paar *[x, j]*, das das minimale Gewicht von P_{ik} erreicht, kann auf der Spitze (top) von S_{ik} eingefügt werden, nur wenn das erste Index vom obersten Paar in S_{ik} kleiner als x ist. Sonst ist das oberste Paar von S_{ik}enger als *[x, j]* und man soll *[x, j]* nicht nehmen.

FallB. Polygonpunkt p_{i} ***ist nicht konkav (Bild 7.7b) :***

FallB ist symmetrisch zu FallA, nur weiß man jetzt, dass p_{k} konkav ist. Da die minimale Konvexzerlegung die Diagonale d_{ij} benutzt, ist entweder p_{j} eine konkave Ecke oder d_{ij} ist eine Polygonkante. In dieser Methode betrachtet man in steigender Ordnung die Indizes $j = i+1$ und die Indizes der konkaven Ecken p_{j} , $i+1<j<k$, für die d_{ij}und d_{jk}in dem Sichtbarkeitsgraph von P_{ik} liegen.

Während des Löschens des obersten Elements in Stapel S_{jk} werden die Paare in Uhrzeigerrichtung durchgetestet, bis das letzte Paar *[s,t]* gefunden wird, in dessen Zerlegung die Diagonale d_{ij} und d_{js} keine konkave Ecke p_{j} schafft. Falls es ein solches Paar nicht gibt oder d_{tk} und d_{ik} konkave Ecke p_{k} produzieren, muss d_{jk} in der Konvexzerlegung von P_{ik} mit Gewicht $W_{\mathrm{ik}} = W_{\mathrm{ij}}+W_{\mathrm{jk}}+2$ und beschränktem Paar *[j, j]* genommen werden. Sonst bekommt man die Konvexzerlegung von P_{ik} mit Gewicht $W_{\mathrm{ik}} = W_{\mathrm{ij}}+W_{\mathrm{jk}}+1$ und Paar *[j,t]*.

Der Keller S_{ik} mit den engsten Paaren von P_{ik}wird folgendermaßen erzeugt: Das erste Element jedes Paares ist der Index der Schleife. Für jedes Paar *[j, x]* mit minimalem Gewicht, entferne das oberste Paar von S_{ik} bis das zweite Element dieses Paares größer als x ist. Dann füge das Paar *[j, x]* ein.

Jetzt folgt der Pseudocode des Algorithmums, der die Realisierung der Datenflusskontrolle im Algorithmus zeigt.

Algorithmus "*Minimale Konvexzerlegung eines allgemeinen Polygons*"

Eingabe: Gegeben sei ein Polygon $P = \{p_{\mathrm{o}},\ p_1,\ .....,\ p_{\mathrm{n1}}\}$
Ausgabe: minimale Konvexzerlegung von P mit dynamischer Programmierung.

1. Initializiere die Gewichte der Kanten Wi(i+1) = 1;
2. FOR sichtbare(i,i+2) DO : w(i,i+2)=0; push[i+1,i+1] in S(i,i+2);
3. FOR size = 3 TO n DO{
3.1. FOR konkave Ecken Pi mit $i + size \leq n$ DO:
k=i+size;
IF sichtbar(i,k) THEN {
IF (Pk konkav) FOR j=i+1, bis k1 DO TypeA(i,j,k);
ELSE {
FOR konkav Pj mit i<j<k1 DO TypeA(i,j,k);
TypeA(i,k1,k);}
}
3.2. FOR konkave Ecken Pk mit $size \leq k < n$ DO:
i = k size;
IF (Pi nicht konkav und sichtbar(Pi,Pk)) THEN {
TypeB(i,i+1,k);
FOR konkav Pj mit i+1 < j < k DO TypeB(i,j,k);
Stapel Sik ist vollständig im Gegenuhrzeigersinn. Bilde Stapel Tik in Uhrzeigersinn.
}

Die Methoden TypeA und TypeB sind oben geklärt worden.

Satz 7.3: *Es sei ein einfaches Polygon P mit n Punkten gegeben, von denen r konkav sind. Der Algorithmus [KS98] löst das Problem der minimalen Konvexzerlegung von P mit Zeit und Speicherplatzkomplexität* $O(nr^2)$.

7.3.2 Implementierung des Algorithmus von Keil

Die wichtigen JAVA– Klassen in dem Package computational_geometry, die den Algorithmus implementieren sind *OptimalConvexPartiotion.java, DecompPoly.java, VisPoly.java, PointList.java, PairDeque.java, SubDecomp.java und Point.java.* In *OptimalConvexPartiotion.java* befindet sich die Methode *ComputeOCP(PointList list)*, die den Algorithmus startet. Sie hat die Hauptschleife, die den Datenfluss und Datenaustausch kontrolliert (flow of control for the dynamic programming).

Die Klasse *DecompPoly* implementiert die folgenden Schritte des Algorithmus: Finden aller konkaven Polygonecken (Methode *InitReflex()*), Zusammenstellen des Sichtbarkeitspolygons jedes Polygonpunktes (Methode *InitVisibility*()), Definieren und Speichern aller Subprobleme (Methode *InitSubproblems*()), Initialisieren ,Setzen, und Speichern der Gewichte und der engsten Paaren der Subproblemen (*InitSubproblems*(), TypeA(), TypeB(), update(), recoverSolutions()). Die oben geklärten zwei Grundfälle Fall A und Fall B zum Finden der Lösungen der Subprobleme werden auch in dieser Klasse mit den Methoden *TypeA*() und *TypeB*() implementiert. Die Funktion *getDecompPol*() in Klasse *DecompPol* ruft die Methode *DrawHelper*(), die Diagonalen, für die Konvexzerlegungen zeichnet und in Vektor speichert. Der Vector ist einer statischen Variablen zugewiesen, die in der Hauptmethode *ComputeOCP*() (siehe oben) in Klasse *OptimalConvexDecomposition* aufgerufen wird und als Ausgabewert von *ComputeOCP*() zurückgegeben wird.

Das Speichern der Paare und der Gewichte der Subprobleme wird von der Klasse *SubDecomp* realisiert, die die beschränkten Paare der Konvexzerlegungen mit Hilfe der Klasse *PairDeque* speichert und mit den Methoden *setWeight*(), *weight*(), *init*() und *pairs*() die Paare und die Gewichte der Subproblemen manipuliert. Die Klasse *SubDecomp* speichert also alle Subprobleme, die für die Konvexzerlegung mittels dynamischer Programmierung nötig sind.

Die Klasse *VisPoly*.java berechnet den Sichtbarkeitsgraph des gegebenen Polygons P, indem sie die einzelnen Sichtbarkeitspolygone jeder Polygonecke findet.

Die Klassen PointList.java und Point.java implementieren die Polygonspunkte, das Polygon selbst und ihre benötigten Eigenschaften.

Manche der obengenannte Klassen sind ursprünglich von Jack Snoeyink programmiert worden, der zusammen mit Mark Keil bei diesem Algorithmus mitgearbeitet hat, und als Freesource zur Verfügung gestellt. Ich habe sie als Basis für meine Implementierung und Visualisierung verwendet.

7.3.3 Visualisierung von Algorithmus von Keil

Es sollte klar verständlich sein, was man bei den ersten Algorithmusschritten sieht. Nach dem ersten Schritt gefundene konvexe Ecken werden in roten Kreisen gezeichnet. Die Diagonalen in den Sichtbarkeitspolygonen dieser Ecken und der

Ecke mit Index 0, werden in gelber Farbe gekennzeichnet. Der Punkt, bei dem gerade das Sichtbarkeitspolygon berechnet wird, wird in hellgrün gezeigt.

Bei den Algorithmus-Schritten 3 und 4 werden Subprobleme initialisiert, und deren Gewichte und beschränkte Paare gesetzt und gespeichert. Wie oben erklärt wurde, ist jedem Subproblem eine Diagonale zugeordnet. Es wird vereinbart, dass die Polygonkanten auch Diagonalen sein können. Diese Diagonalen (i,i+1) und (i, i+2) (falls sie existieren) werden in dunkelblau gezeichnet. Beispiele im Bild unten.

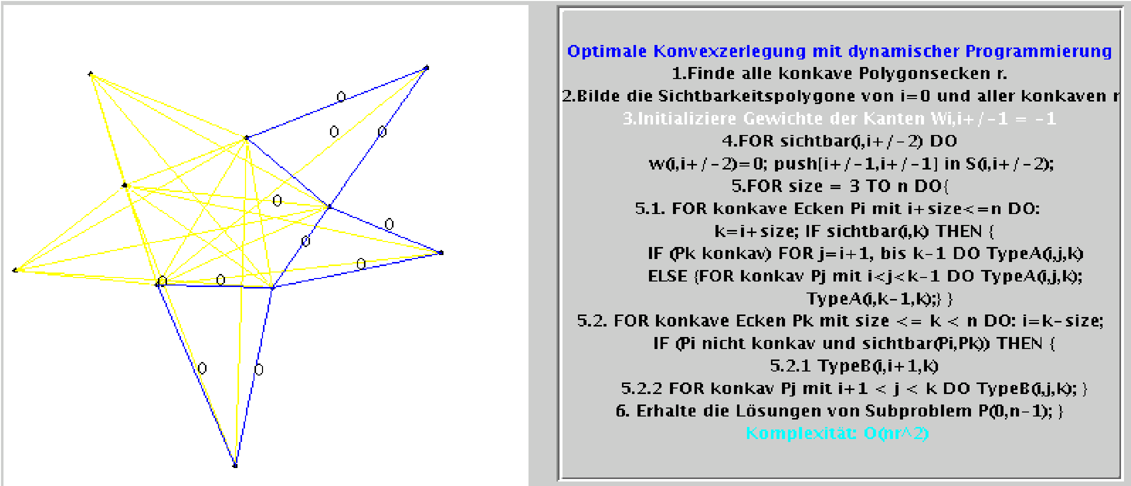

Bild7.8 Die Algorithmus–Animation im Applet während der Ausführung vom Algorithmus für optimale Konvexzerlegung.

Im Schritt 5.1 im Pseudocode oben wird die konkave Ecke p_i in einem lila Kreis gezeichnet und p_k in einem roten. Sei p_k aus p_i sichtbar, d.h. sei das Liniensegment *(i,k)* eine Diagonale in dem gegebenen Polygon, startet eine der beiden Schleifen, abhängig davon, ob p_k konkav oder nicht konkav ist. Falls p_k sichtbar aus $p_i$ist, wird die Diagonale dazwischen gelb gezeichnet. In den beiden Schleifen werden immer drei Punkte mit Indizes *i, j, k* in der Methode *TypeA*() übergeben (siehe oben). Sie berechnet die Gewichte der einzelnen existierenden Subprobleme mit der Hilfe eines speziellen Kellers (Klasse: *PairDeque*). Die drei Punkte gestalten ein Dreieck der oben diskutierten kanonischen Triangulierung, das hellgrüne Farbe in der Animation hat. Nachdem ein besseres Gewicht für ein bestimmtes Subproblem gefunden wurde, wird das Gewicht in dem entsprechenden Keller aktualisiert. Die Diagonalen dieses Subproblems wird in grau visualisiert und das Gewicht wird daneben beschrieben.

Die Ereignisse aus Schritt 5.2 werden analog aber mit unterschiedlichen Farben visualisiert. Die konkaven Ecken p_k werden hellgrün eingekreist. Die nicht konkaven Ecken p_i werden (analog zu den p_k oben in 5.1) rot gezeichnet und falls p_i sichtbar aus p_k sind, werden die Diagonalen gelb gezeichnet. Dann werden in der Schleife drei bestimmte Punkte (siehe Pseudocode) in der Methode *TypeB* übergeben, um die Gewichte der Subprobleme mittels den entsprechenden Kellern mit beschränkten Paaren zu aktualisieren oder neu zu setzen. Diese drei Punkte und das von ihnen produzierte kanonische Dreieck erscheinen rosarot. Nachdem ein besseres Gewicht für ein bestimmtes Subproblem gefunden wurde, wird die Diagonale dieses Subproblems in grau gezeichnet.

Im vorletzten Schritt werden alle Indizes–Paare der Lösungen vom Subproblem $P_{0,\mathrm{n}-1}=P$ ausgesucht. Die Diagonalen aus zu diesen Paaren gehörenden Konvexzerlegungen werden rot gezeichnet.

Schließlich werden in blau die Diagonalen einer optimalen Konvexzerlegung von P gezeichnet und in orange Farbe werden die Diagonalen der anderen möglichen optimalen Konvexzerlegungen dargestellt.

Kapitel 8

Die Applikation

Im Folgenden beschreibe ich die Möglichkeiten, die das Applet bzw. die Applikation “Visualisierung von Algorithmen für Polygon–Zerlegungen” anbietet und wie man damit umgehen kann. Wie in Kapitel 1 erwähnt wurde, soll das Applet die gewünschten Merkmale einer interaktiven Algorithmen–Animation für den elektronischen Unterricht in algorithmischer Geometrie haben. Es hat die Eigenschaften Funktionale Struktur, Anzeige des Pseudocode, Lehrtext, Interessante Eingabedaten, Flexible Ablaufsteuerung, Interessante Ereignisse, Fokussierung der Aufmerksamkeit, Darstellung mehrerer Algorithmen, die im Kapitel 1 im Einzelnen erklärt werden

Das Applet kann in jedem Internet–Browser gestartet werden, der JAVA–PlugIn1.3. oder höhere Versionen von JAVA–PlugIns installiert hat. Es wurde vor allem unter Netscape 6.2 und Mozilla getestet. Die Größe der GUI–Komponente ändert sich abhängig von der JAVA–Version (dem Plug–In). Die zu dem Applet gehörende Applikation kann über die main()-Methode der Klasse CGVisualser.java ausgeführt werden.

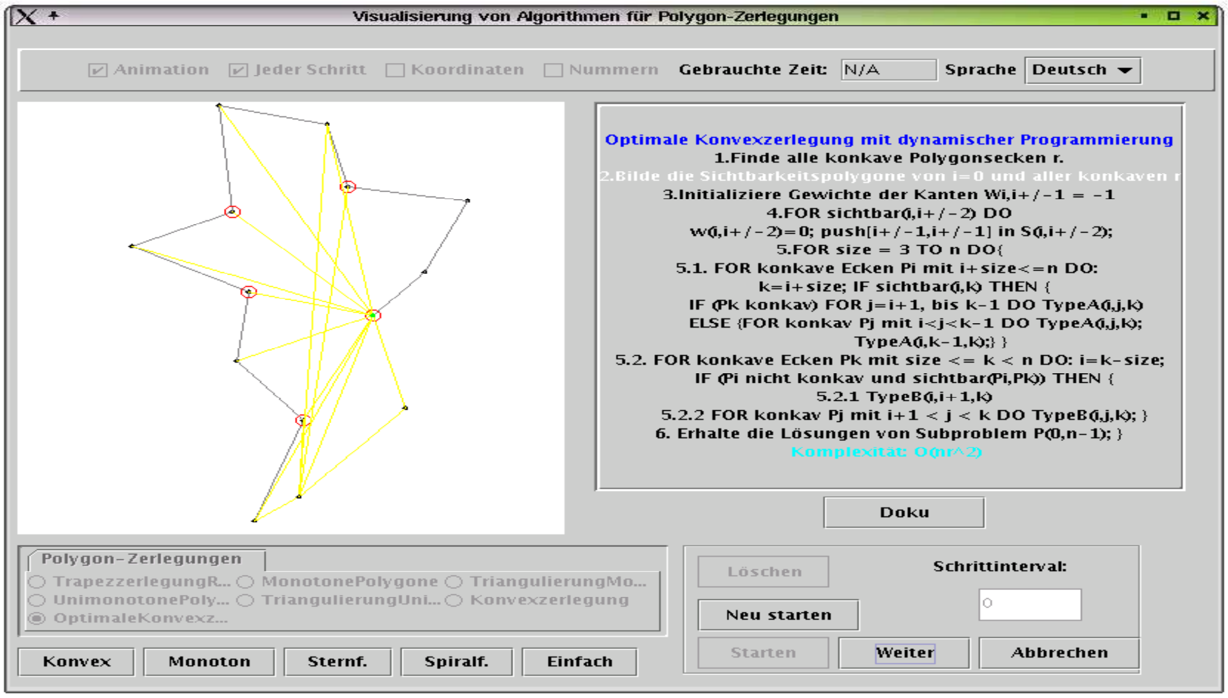

Bild8.1 Das Applet "Visualisierung von Algorithmen für Polygon–Zerlegungen" während der Ausführung des Algorithmus von Keil für optimale Konvexzerlegung eines monotonen Polygons.

8.1 Benutzungsanweisungen.

Die Reihenfolge der folgenden Untertitel wird als Reihenfolge bei der Arbeit mit dem Applet empfohlen.

8.1.1 Sprache auswählen

Das Applet sollte das Konzept der Internationalisierung von Software–Applikationen berücksichtigen. Über die *Sprache–Combobox* kann man aus den zur Verfügung stehenden Sprachen diese auswählen, in der die Überschritte in der graphischen Benutzeroberfläche und die Programm–Meldungen in der JAVA–Konsole geschrieben werden. Zur Zeit sind Englisch und Deutsch wählbar.

Man kann aber leicht eine neue Sprache einprogrammieren. Dies wird unten bei der Implementierung des Applets erklärt.

8.1.2 Das Polygon für das Partitionieren eingeben

Bevor der Benutzer das Polygon, das partitioniert werden soll, eingibt, kann er über die *Koordinaten–Combobox* und *Indizes–Combobox* auswählen, ob die Koordinaten und die Indizes der Polygonecken gezeigt werden.

Es gibt zwei Möglichkeiten, ein Polygon für einen Polygonzerlegungs–Algorithmus einzugeben:

1. Man kann eine der *Eingabe–Tasten* drücken und so erscheint eins der fünf Beispiel–Eingabepolygone in das Zeichnen–Fenster.

2. Der Benutzer kann mit der linken Maustaste ein beliebiges einfaches Polygon selber zeichnen. "Einfaches" heißt, dass die Kanten des Polygons sich nicht schneiden sollen. Falls dies aber passiert, wird eine Fehlermeldung in der Konsole angezeigt.

(!!!) Es ist wichtig, das Polygon ***entgegen der Uhrzeigerrichtung*** zu zeichnen, um ein richtiges Ergebnis zu bekommen. Diese Bedingung wird oft bei den Algorithmen in der Computational Geometrie (algorithmischen Geometrie) angenommen.

(!!!) Bezüglich der Eingaben aller Algorithmen im Applet wird angenommen, das keine zwei Polygonpunkte identische y–Koordinaten haben. Es wird in Kapitel 3 gezeigt, dass diese Annahme keine Begrenzung der Allgemeinheit für die Funktionalität der Algorithmen ist. Auf diesem Grund wird ein Punkt, der auf der selben Höhe eines schon existierenden Punktes, nicht gezeichnet. Es erscheint eine Meldung in der JAVA–Konsole, die dies erklärt.

8.1.3 Algorithmus auswählen

Der Benutzer kann einen der sieben Algorithmen im *Algorithmen–Panel* auswählen, indem er auf den Namen des Algorithmus oder auf den Radio–Button klickt. Als Vorgabe ist der Algorithmus für Trapezzerlegung ausgewählt. In dieser Version des Applets wird bei der Auswahl bzw. bei dem Klick auf einen bestimmten Algorithmus automatisch die Berechnung des Algorithmus gestartet.

8.1.4 Interaktion mit der Algorithmus–Animation

Die Berechnung jedes Algorithmus kann über die *Start–Taste* gestartet werden. Dann wird entsprechend dem Status der *Animations–Combobox* eines der drei Szenarien durchgeführt:

8.1.4.1 Keine Animation

Falls die *Animation–Combox* nicht ausgewählt wird, läuft das Programm in keinem Demonstrationsmodus, also läuft keine Algorithmus–Animation. Das Ergebnis des Algorithmus bzw. das partitionierte Polygon erscheint fast sofort nach dem Start in dem Zeichen–Fenster. Zur Analyse und Vergleich der Algorithmen wird die für die Berechnung benötigte Zeit vom *Zeit–Label* (oben in dem Optionen–Panel) gemeldet.

Es existieren zwei Demonstrations–Modi, in denen die Animationen ausgeführt und so die Algorithmen visualisiert werden:

8.1.4.2 Selbstständige Animation

Dieser Modus wird ausgewählt, wenn die *Animation–Checkbox* eingeschaltet und die *Schritt–Checkbox* nicht eingeschaltet sind. Bevor der Algorithmus gestartet wird, ist es möglich, die Geschwindigkeit der Animation über das *Schrittintervall–Feld* (in dem *Kontroll–Panel*) einzugeben. Was man eingibt, ist eine integer–Zahl, die Zeitlänge (in ms) des Intervalls zwischen den Algorithmusschritten, die das Programm warten muss. Je größer die Zahl ist, desto langsamer läuft die Algorithmen–Animation. Als Vorgabe ist das Intervall auf 0 gesetzt, womit die Animation möglichst schnell läuft.

In diesem Demonstrationsmodus werden die Algorithmusschritte nacheinander vom Programm ausgeführt und visualisiert. Der Benutzer muss nicht jeden Schritt über die *Weiter–Taste* selber starten.

Er kann die Animation auch nur betrachten. Bei Wunsch kann er den Ablauf der Ausführung wie folgt steuern:

- Mit der *Abbrechens–Taste* wird die Animation abgebrochen, wobei die *Schritt–Combobox* ausgewählt wird und das Programm die Ausführung des Algorithmus und die Animation in dem anderen Demonstrations–Modus fortsetzt.
- Nach dem Abbruch der Animation, kann der Benutzer die Geschwindigkeit der Animation verlangsamen oder beschleunigen und, um weiter im gleichen Modus zu bleiben, muss er die Schritt–Combobox ausschalten.
- Nach dem Abbruch der Animation, kann der Benutzer alle sensitiven Option–Checkboxen aus dem Optionen–Panel ändern und dann im gleichen Modus bleiben, indem er die Schritt–Combobox ausschaltet.
- Um die Berechnung des Programms zu stoppen und neu den gleichen oder einen anderen Algorithmus (mit seiner Animation) mit den selben oder unterschiedlichen Programm–Parametern neu zu starten, kann der Benutzer auf die *Neustarten–Taste* drucken.

8.1.4.3 Benutzerabhängige Animation

Eine bestimmte Animation kann in dem anderen Demonstrationsmodus gestartet werden, wenn die *Animations–Checkbox* und die *Schritt–Checkbox* eingeschaltet sind. In diesem Modus kann der Benutzer mittels der *Weiter–Tast*e die Visualisierung in Einzelschritte vorschalten. Wie im ersten Kapitel bei der Beschreibung der Eigenschaft “Flexible Ablaufsteuerung” gesagt wurde, ist dies aus didaktischer Sicht sehr wichtig, da der Student z.B. beliebig viel Zeit zum Überlegen bei jeder Änderung in der Animation hat.

Hier wirkt sich das gegebene Intervall (in dem Schrittintervall–Feld) auf die Berechnungsgeschwindigkeit bzw. wie schnell sich die Animation verändert aus, aber da der Benutzer jeden einzelnen Schritt selbst schaltet, spielt das gegebene Intervall eine nicht so entscheidende Rolle.

Der Benutzer muss hier die Animation selbst nach vorne navigieren. Er hat die folgenden Steuermöglichkeiten:

- Mit Klick auf die *Abbrechen–Taste* wird die *Schritt–Combobox* sensitiv und kann ausgeschaltet werden. So kann das Programm die Ausführung des Algorithmus im anderen Demonstrations–Modus fortführen.
- Der Benutzer kann die Geschwindigkeit der Animation verlangsamen oder beschleunigen.
- Nach Drücken der Abbrechen–Taste, werden die Koordinaten und *Indizes–Checkbox* aus dem Optionen–Panel sensitiv und können beliebig geschaltet werden.
- Um die Berechnung des Programms zu stoppen und den gleichen oder einen anderen Algorithmus mit den selben oder unterschiedlichen Programm–Parametern neu zu starten, kann der Benutzer auf die *Neustarten–Taste* drucken.

(!!!) Es ist wichtig, dass in diesem Modus, nach der Neustarten–Taste die *Weiter–Taste* so lange gedrückt werden muss, bis die Berechnung aufhört.

Falls ein Algorithmus gestartet wird, während noch das Zeichnen–Fenster leer ist bzw. keine Eingabe gegeben wurde, kommt eine Fehlermeldung in der JAVA–Konsole. Die Meldung lautet, dass mindestens ein Dreieck zum Starten eines Algorithmus benötigt wird.

8.1.5 Dokumentation aufrufen

Unter dem Pseudocode–Panel befindet sich die *Doku–Taste*. Falls der Rechner in das Internet geschlossen ist, öffnet sich beim Klick, die Internet–Seite, in der die Theorie hinter dem gewählten Algorithmus und auch seine Visualisierung ausführlich erklärt wird. Wenn kein Internetzugang zur Verfügung steht, erscheint eine Meldung in der JAVA–Konsole.

8.1.6 Eingabe löschen

Mit Hilfe der *Löschen–Taste* kann das Eingabe aus dem Zeichnen–Fenster gelöscht werden. Dann kann der Benutzer ein neues Polygon zeichnen. Diese Taste ist praktisch in den Fällen, wenn ein nicht einfaches Polygon, also ein sich selbst schneidendes Polygon oder ein Polygon mit Ecken in der Uhrzeigerrichtung gezeichnet wird, da solche Eingaben nach den Annahmen (siehe 8.1.2) falsch sind und damit entfernt werden können.

8.1.7 Eingabe modifizieren

Der Benutzer verfügt über zwei Möglichkeiten, ein schon gezeichnetes Polygon, das schon partitioniert oder nicht partitioniert wurde, zu ändern.

8.1.7.1 Neue Punkte hinfügen

(!!!) Es ist anzumerken, dass in dieser Version des Applets, neue Punkte nur zwischen der letzten (mit Index *n–1*) und der ersten (mit Index *0*) Polygonecke eingefügt werden können, oder anders gesagt, nur am Ende des Polygons. Dafür ist nützlich die Indizes–Combobox einzuschalten, so dass die Indizes der Punkte sichtbar sind. Es ist auch verstädlich, dass nach dem Einfügen neuer Punkte, das Polygon einfach (d.h. keine zwei Kanten schneiden sich) bleiben muss. Das Zeichnen eines Punktes geschieht mit der linken Maustaste.

8.1.7.2 Punkte bewegen.

Mit der rechten Maustaste kann der Benutzer einen beliebigen Polygonpunkt auswählen, indem er darauf klickt. Dann kann er den Punkt in dem Zeichnen–Fenster bewegen, indem er die Maus mit der gedruckten linken Taste bewegt, und den Punkt auf einer neuen Stelle lässt (Maustaste lösen).

Diese Funktion des Applets ist praktisch für das Testen eines Algorithmus und für das Suchen nach speziellen Eingaben, bei ihren der Algorithmus bzw. die Animation ein besonderes Verhalten hat.

8.2 Entwurf und Implementierung der Applikation bzw. des Applets

In diesem Teil möchte einen Überblick über die innere Struktur der Applikation "Visualisierung von Algorithmen für Polygonzerlegungen" geben.

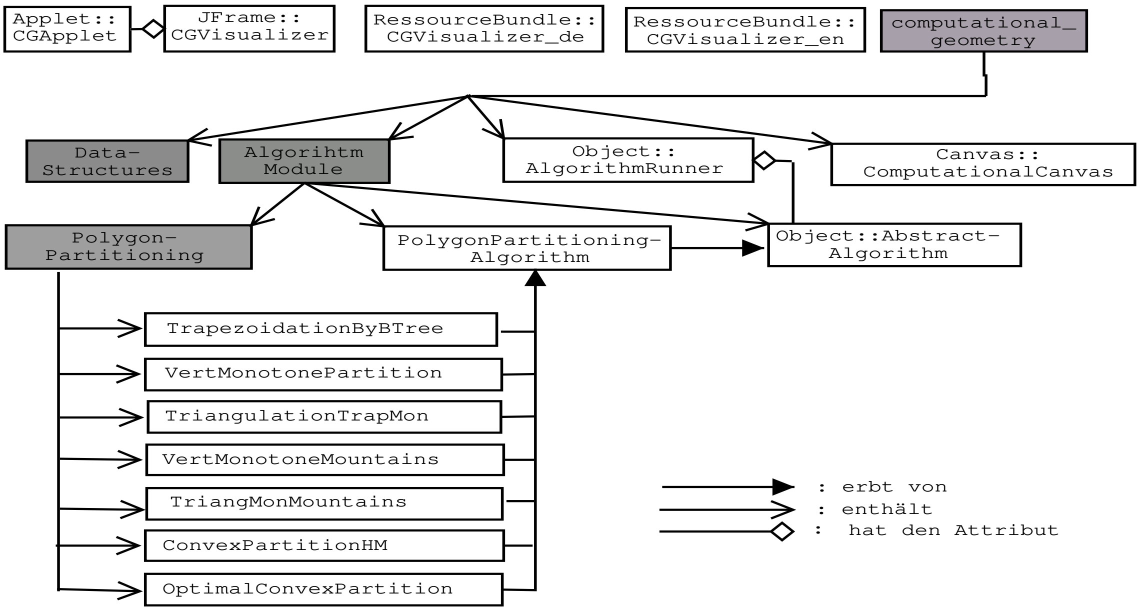

Bild8.2 Struktur der Applikation "Visualisierung von Algorithmen für Polygonzerlegungen"

Im Bild oben sieht man die wichtigsten Klassen und die Packages, die die Applikation implementieren. Die Namen der Packages sind in den grauen Rechtecken eingegeben. Das Package *computationa_geometry* enthält alle für die Implementierung benötigte Klassen ausser der Klasse "*CGVisualizer*" für die Rahmen der Applikation und der Klassen für die Internationalisierung "*CGVisualizer_de*" und "*CGVisualizer_en*".

Im Package *computational_geometry* befinden sich die Klassen, die verschiedenen geometrische Objekte wie Punkt, Liniensegment, Sammlung von Punkten, Liste von Punkten, Polygon, sowie auch die von den Algorithmen benötigte Methoden und Funktionen auf diese Objekte implementieren. Die Klasse *algorithms*(von Gosper) implementiert grundliegende Eigenschaften der geometrischen Objekte wie z.B. ob zwei Liniensegmente sich schneiden oder ob ein Punkt links oder rechts eines Liniensegments liegt. Dort befinden sich auch einige Hilfklassen (*DecompPoly, SubDecomp, VisPoly, Homog, PairDeque*), die entsprechende Algorithmusschritte für die minimale Konvexzerlegung eines Polygons realisieren. Die wichtigsten Klassen in diesem Package sind *AlgorithmRunner* und *ComputationalCanvas,* die oben in Bild eingebaut sind.

Der *AlgorithmRunner* spielt die wichtige Rolle einer Schnittstelle zwischen der Benutzeroberfläche und der Klasse, die einen bestimmten Algorithmus implementiert. Der Konstruktor jedes Algorithmus hat als Parameter eine Instanz der *AlgorithmRunner*–Klasse. Da der *AlgorithmRunner*–Konstruktor als Parametern alle GUI–Komponente hat, weiß er Bescheid, was der Benutzer auf der graphischen Oberfläche eingegeben hat. So kann die Algorithmus–Klasse alle von ihr, nach dem Start, benötigten Daten, wie z.B ob und welches Demonstrationsmodus oder welches Schrittintervall ausgewählt wurde, über den *Algo-*

rithmRunner bekommen. Es gibt aber auch Datenverkehr in die andere Richtung (von Algortihmus zur Benutzeroberfläche). Im Demonstrationsmodus kann die Algorithmus–Klasse über den *AlgorithmRunner* auf dem *Zeichnen–Fenster* zeichnen.

Die Klasse *ComputationalCanvas* implementiert das *Zeichen–Fenster*. Sie ist eine Erweiterung von JAVA–MouseListener und MouseMotionListener und registriert, wenn der Benutzer darauf klickt, um z.B. neu zu zeichnen oder einen Punkt zu bewegen. Durch die *ComputationalCanvas*–Klasse wird ein wichtiger Teil der Animationen realisiert, da sie die Methoden zum Zeichnen aller behandelten geometrischen Objekten implementiert. Die Applikation zeichnet also über diese Klasse.

In dem Package *DataStructure* (im Bild oben) befinden sich die Datenstrukturen, die verwendet werden: *LinkedList*, *DoubleLinkedList*, *RedBlackTree*.

Das Package *"AlgorithmModule"* beinhaltet die Klassen, die tatsächlich die Algorithmen implementieren und visualisieren. Jeder Algorithmus wird in drei Schichten betrachtet und implementiert. Betrachten wir als Beispiel den Algorithmus für Trapezzerlegung. Die erste grundlegende Schicht wird von der Klasse *AbstractAlgorithm,* die zweite Schicht wird von der Klasse *"PolygonPartiotioningAlgorithm"* und die dritte wird von der *TrapezoidationByRBTreer*–Klasse dargestellt. Wie man im Bild sieht, ist die zweite Klasse eine Erweiterung der ersten und die dritte Klasse ist eine Erweiterung der zweiten. *AbstractAlgorithm* implementiert allgemeine Methoden für die Vorbereitung der Umgebung zum Ausführen des aktuellen Algorithmus (prepareEnvironment(), clearEnvironment()), Methode für die Ablaufsteuerung der Ausfühung((runAlg(), calculate(), calcWait(), continueCalc(), interruptCalc()) und Methoden, die für das Setzen des Pseudocodes helfen und eine Pseudocode–Zeile während der Animation hervorheben. *PolygonPartitioningAlgorithm*–Klasse implementiert die folgende abstrakte Methode aus ihrer VaterKlasse "setTabName", die spezifisch für die Gruppe der Algorithmen ist. Die implementierende Klasse des konkreten Algorithmus (z.B. *TrapezoidationByRBTreer*) implementiert auch die abstrakte Methoden aus ihrer VaterKlasse "setAlgName", "setAlgTitle", "setPseudoCode", die spezifischen Daten für den Algorithmus setzen.

Alle Klassen, die die einzelnen Algorithmen implementieren und visualisieren befinden sich in dem Package *PolygonPartitioning.* Über die Implementierung der einzelnen Algorithmen und ihre Visualisierung wird in den entsprechenden Kapiteln diskutiert. Es ist hier anzumerken, dass manche Methoden wie z.B. Trapezoidation(), getAllMonotonePolygons() und getMonPol(), die in mehreren Klassen vorkommen, beim ersten Anblick identisch aussehen, sie sind aber nicht, da sie einen Algorithmusschritt auf verschiedene Weise visualisieren.

Die ***Internationalisierung*** wird mit Hilfe der JAVA–Klassen *RessourceBundle* und *Locale* implementiert. Alle Strings einer beliebigen Sprache, die in der Applikation benutzt werden, sind in einer *RessourceDatei* gespeichert, die eine Erweiterung von *ListRessourceBundle* ist. Solche Ressource–Dateien sind die im Bild gezeigten *CGVisualizer_de* und *CGVisualizer_en.* Die Klasse *RessourceBundle* besitzt die Methode getString(), durch die in bestimmten Stellen im Programm(z.B die Methode *updateDisplay()* in der FrameKlasse) Strings geholt und gesetzt werden können. Je nach der ausgewählten Sprache in der Sprache–Combobox, wird die richtige RessorceDatei geholt. Dies ist in der Methode *setCurrentLocale(int LocaleIdx)* in der Frame–Klasse "CGVisualizer" programmiert, die in der Array mit allen *Locale*Objekten, je nach dem gegebenen Land–Index, das gewählte Land (die Variable *currentLocale*) findet und dann damit über die Methode *Ressourceundle.getRessource(ApplName, currentLocale)* die entsprechende RessourceDatei setzt. Die *Locale*–Objekten präsentieren die Länder bzw. die Sprachen. Die RessourceDateien sollen im Verzeichnis der Frame-Klasse der Applikation liegen und ihre Namen sollen nach dem Muster *ProgramResource_sprache_land* oder *ProgramResource_sprache* gegeben werden. (z.B. CGVisiaizer_de_DE)

Mehr Information und gute Beispiele für Internationalisierung in JAVA gibt das Buch "core JAVA2 Band2 Expertenwissen" [HoCo].

Die oben geklärte Architektur wurde mit der Idee entworfen, die Applikation leicht mit interaktiven Animationen neuer Algorithmen aus dem existierenden oder aus einem anderen Bereich der Computational Geometry zu erweitern.

JAVA–Threads werden in der Implementierung benutzt, um zu ermöglichen, das Applet in mehreren Browser–Fenstern zu starten und das Verhalten unterschiedlicher Algorithmen mit gleicher Eingabe oder desselben Algorithmus mit verschiedenen Eingaben zu vergleichen.

8.2.1 Neue Algorithmus–Animation in die Applikation implementieren.

Falls der neue Algorithmus eine andere Polygonzerlegung produziert, kann seine Implementierungsklasse eine Erweiterung der Klasse "PolygonPartitioningAlgortihm" sein. Diese Erweiterung soll die abstrakte Methoden "setAlgName", "setAlgTitle", "setPseudoCode" entsprechend implementieren.

Falls der neue Algortihmus aus einem anderen Thema der algorithmischen Geometrie ist, kann seine Implementierungsklasse eine Erweiterung der AbstractAlgorithmusKlasse sein, die für den Algorithmus besondere Funktionen darstellt.

Die Instanzen der neuen Klassen sollen im Konstruktor der Rahmen–Klasse "CGVisualzer" deklariert werden.

8.2.2 Neue Sprache in die Applikation programmieren.

Um eine neue Sprache in die Sprache–Combobox hinzufügen, soll man die entsprechende RessourceDatei (Erweiterung von ListRessourceBundle) erzeugen, die alle in der Anwendung benutzten Strings enthält. Es ist wichtig, dass der Name der RessourceDatei das Schema *ProgramResource_ sprache_ land* folgt. Der Programmierer muss auch das Land der neuen Sprache in das Array “locales” (in der FrameKlasse “CGVisualizer”) addieren.

8.2.3 Ein beliebiges Polygon zu einer der Eingabe–Tasten programmieren

Wenn der Benutzer ein beliebiges Polygon zu eine der Eingabe–Tasten einstellen möchte, kann er dies sehr einfach realisieren. Da die Koordinaten der Polygonecken beim Zeichnen in der JAVA–Konsole aufgelistet werden, kann er sie dort speichern. Dann muss er die Punkte in einer der fünf ButtonActionPerformed()–Methoden in der Frame–Klasse “CGVisualizer” mit den Punkten des neuen gewünschten Polygons ersetzen. So bleibt das Polygon gespeichert und nur mit einem Mausklick in dem *Zeichnen–Fenster* angezeigt.

8.3 Beschreibung der Benutzeroberfläche

Am Ende der Dokumentation möchte ich kurz die Komponente der graphischen Benutzeroberfläche (GUI) beschreiben. Die graphische Benutzeroberfläche kann im allgemeinen in fünf Haupt–fenster geteilt werden: Optionen–Panel, Zeichnen–Fenster, Pseudocode–Panel, Algorithmen–Panel, Eingaben–Panel, Kontroll–Panel.

8.3.1 Das Optionen–Panel

Bild8.3 Optionen–Panel

Wenn die ***Animation–Checkbox*** ausgewählt ist, wird die Algorithmus–Animation gezeigt. Bei ihrer Auswahl wird die ***Schritt–Checkbox*** automatisch auch aktiviert, also wird die benutzerabhängige Animation starten.

Die ***Schritt–Checkbox*** kann nur dann aktiviert werden, wenn die Animation–Checkbox aktiviert ist. Sie wird automatisch deaktiviert und ist nicht wählbar (insensitiv), wenn die Animation–Checkbox ausgeschaltet wird. Sie startet die benutzerabhängige Animation. Sonst läuft die selbstständige Animation.

Nach dem Status der ***Koordinaten*** und der ***Indizes Checkbox*** werden die Koordinaten und die Indizes der Polygonpunkte in dem Zechnen–Fenster gezeigt.

Zeit–Label zeigt die Berechnungszeit, die für die Ausführung des bestimmten Algorithmus in *“nicht Demonstration–Modus”* benötigt wird. Da die Zeit in den zwei Demonstration–Modi von den Benutzer–Eingaben (Warteintervall, auf Weiter–Taste drucken) abhängt, wird sie dann nicht gemeldet.

Sprache–Combobox gibt dem Benutzer die Möglichkeit, die Sprache in der Anwendung zu bestimmen. Bei Wahl einer Sprache wird sie sofort aktiviert. Unter 8.2.2 wird geklärt, wie eine andere Sprache in der Sprache–Combobox zugefügt werden kann.

8.3.2 Das Zeichnen–Fenster

Im ***Zeichnen–Fenster*** kann der Benutzer das Polygon, das einer der Algorithmen partitionieren soll, mit der linken Maustaste zeichnen.

Wenn eine der Eingabe–Tasten gedrückt wird, erscheint das entsprechende Polygon in diesem Fenster. Hier werden die Algorithmen–Schritte visualisiert bzw. laufen die visualisierende Animationen.

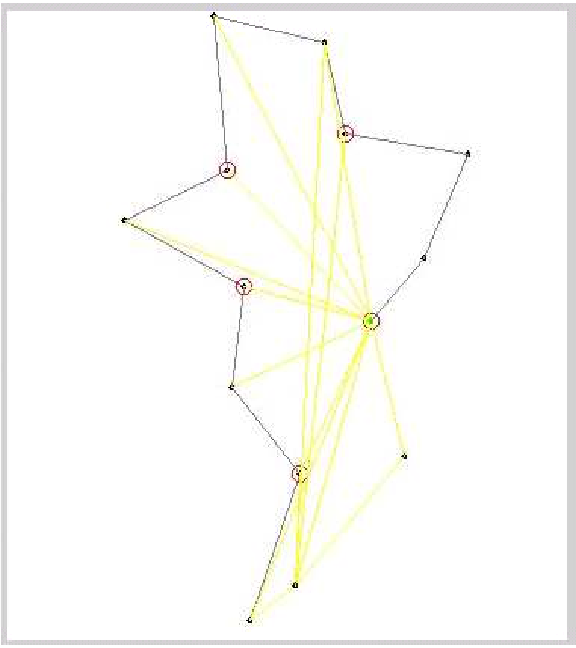

Bild8.4 Das Zeichen–Fenster während der Animation des Algorithmus für optimale Konvexzerlegung mit der Eingabe ein monotones Polygon.

8.3.3 Das Pseudocode–Panel und die Doku–Taste

In diesem Panel wird der Pseudocode des aktuellen Algorithmus gezeigt, sobald der Algorithmus gewählt wird. Es wird dazu auch die Zeitkomplexität für die Analyse angegeben. In den Demonstration–Modi wird eine Zeile des Pseudocodes mittels weißer Farbe hervorgehoben. Die Zeile erklärt den gerade geschehenden Schritt. (Siehe 1.2.1).

Mittels der ***Doku–Taste*** kann die Dokumentation zu dem bestimmten aktivierten Algorithmus in einem Browser–Fenster geöffnet werden. Dies ist möglich, falls Internetzugang zur Verfügung steht. So kann der Benutzer während der Arbeit mit dem Algorithmus ausführliche Erklärungen zu den Algorithmus–Schritte und zu deren Visualisierung bekommen.

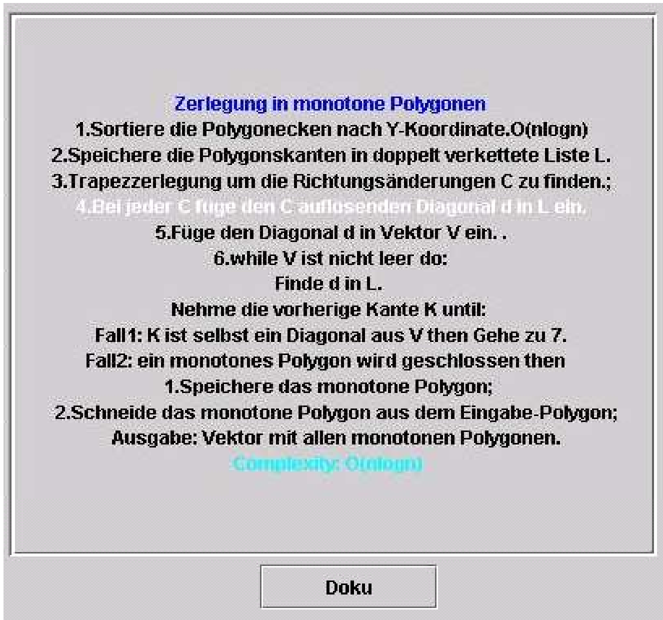

Bild8.5. Das Pseudocode–Panel und die Doku–Taste.

8.3.4 Das Algorithmen–Panel

Bild8.6 Das Algorithmen–Panel mit den sieben visualisierten Agorithmen aus der algorithmischen Geometrie.

Algorithmen–Panel beinhaltet von der Applikation bzw. dem Applet implementierten und visualisierten Algorithmen. Es ist verständlich, dass nur ein Algorithmus aktiviert sein kann. Deswegen wird nach dem Start eines Algorithmus das ganze Panel unselektiv.

In dieser Version wird beim Auswählen der Algorithmus automatisch gestartet.

8.3.5 Das Eingabe–Panel

Bild8.7 Das Eingabe–Panel, das fünf Beispiele für verschiedene Typen von Polygonen anbietet.

Beim Drücken der Tasten im ***Eingabe–Panel*** wird im Zeichnen–Fenster entsprechend ein konvexes, ein monotones, ein sternförmige, ein spiralförmiges oder ein einfaches Polygon gezeigt. (Die Definitionen und Anwendungen werden in Kapitel 2 erklärt.) Das Polygon kann als Eingabe für den Algorithmen genommen werden.

8.3.6 Das Kontroll–Panel

Bild8.8 Das Kontroll–Panel, damit der Benutzer den Ablauf der Algorithmen–Animationen steuern kann. Das Bild zeigt das Panel während einer benutzerabhängigen Animation (siehe 8.1.4.3).

Über das Kontroll–Panel geschieht die Ablaufsteuerung der Animationen.

Die ***Start–Taste*** startet die Ausführung des Algorithmus bzw. der Animation.

Mit der ***Weiter–Taste*** schaltet man die Algorithmusschritte in Demonstrationsmodus nach vorne. (Siehe 8.1.4).

Mit der ***Abbrechen–Taste*** kann die Animation abgebrochen werden. Sie ist nur im Demonstrationsmodus selektierbar. Beim Anklicken wird auch die Schritt–Taste ausgewählt. Diese erlaubt dem Benutzer zwischen den zwei Demonstration–Modi (siehe 8.1.4) während der Berechnung zu wechseln und manche Programmparametern (Koordinaten, Indizes, Sprache) zu ändern.

Die ***Neustarten–Taste*** stoppt die laufende Berechnung, löscht alle Programmvariablen. So kann ein neuer Algorithmus oder ein neues Polygon eingegeben werden.

In dieser Version muss nach dem Anklicken dieser Taste, die Weiter–Taste so lange gedrückt werden, bis die Berechnung aufhört.

In dem ***Intervall–Textfeld*** kann das Zeitintervall (eine integer–Zahl) zwischen den Algorithmus–Schritten in den selbstständigen Animationen in Millisekunde eingetragen werden. Das ist das Intervall zwischen den Berechnungsschritten bzw. zwischen den Änderungen in der Animation.

Die ***Löschen–Taste*** entfernt alle geometrischen Objekten aus dem Zeichnen–Fenster.

Kapitel 9

Literaturverzeichnis

[AAI86] Ta.Asano, Te.Asano, H.Imai. Partiotioning a polygonal region into trapezoids. J. ACM, 33:290312,1986

[BCLT96] J.E.Baker, I.F.Cruz, G.Liotta, Tamasia. Algorithm Animation over the World Wide Web, *Proc. Int. Workshop on Advanced Visual Interfaces*, S.203–212, 1996

[Ba81] Baeker, “Sorting out Sorting” (video), in Siggraph Video Review 7, 1981

[Bel87] R.E.Bellman. Dynamic Programming. Princeton University Press, Princeton, NJ, 1987

[BKOS97] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, SpringerVerlag, Berlin, 1997.

[Br88] Marc Brown. Algorithm Animation. MIT Press, 1988

[Br91n] Marc Brown, John Hershberger. Color and Sound in Algorithm Animation. SRC,1991

[BrSe85] Marc Brown, Robert Sedgewick, Techinques for algorithm animation. *IEEE Software*, 2(1):2839,Jan 1985

[Br92] Marc Brown.“Zeus: a system for algortih animation and multiview editing”. *Computer Graphics*, 18(3):177186, May 1992

[BrNa96] M.Brown, Najork. Colaborative active textbooks: a webbased algorithm animatoin system for an electronic classroom. SRC Research Report 142, May 1996

[CD85] B.Chazelle, D.P.Dobkin. Optimal convex decompositions.1985 In Toussoint,ed.,*Computational Geometry,* Amsterdam.

[Cha91] B.Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput.Geom.*, 6:485524,1991.

[ChaIn84] B.Chazelle, Incerpi. Triangulaiton and shape complexity. 1984,*ACM Trans. On Graphics* 3,13552.

[CTV89] K. Clarcson, R.E. Tarjan, C.J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon, *Discrete Comput. Geom.* 4, 42332, 1989.

[EL81] H.ElGindy, L.Avis. A linear algorithm for computing the visibility polygon from a point.*J.Algorithms*,2:186197,1981

[ET89] H.ELGindy, G.T.Toussaint. On geodesic properties of polygons relevant to linear time triangulation. *Visual Comput.* , 5(12):6874,1989.

[EpsKa94] P.Epstein, J.Kavanagh,et al.: "A workbench for computational geometry" .Algorithmica, 11(4):404428, April 1994

[FePav75] H.Y.F. Feng, T.Pavlidis. Decomposition of polygons into simpler components: feature generation for syntactic pattern recognition. *IEEE Trans. Comput.*,C24:363650,1975

[FSS84] L.Ferrari, P.V.Sankar, J.Sklnsky. Minimal Rectangular Partitions and digitized blobs. Computer Vision, Graphics, and Image Processing,28:5871,1984

[FoMo84] A. Fournier, D.Y.Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. on Graphics* 3, 15374

[GJPT78] M.R.Garey, D.S.Johnson, F.P.Preparata, and R.E.Tarjan.Traingulating a simple polygon. *Inform. Process. Lett.*,7:175179.

[Gr83] D.H.Greene. The decomposition of polygons into convex parts. 1983. in F.P.Preparata, ed.,*Advances in Computing Research*, JAI PressGreenwich,CT,pp 23559.

[HM83] S.Hertel, K.Mehlhorn. Fast triangulation of simple Polygons. 1983. *Proc. 4^{th} Internat. Conf. Found. Comput. Theory, Vol. 158 of Lecture Notes in Comput. Sci.*,Springer Verlag Berlin,p.207.

[HoCo] Cay S. Hortsmann und Gary Cornell. "Core JAVA2 Band2 Expertenswissen", S. 888 Kapitel10, Markt+Technik, 2000.

[Keil83] M.Keil. Decomposing a polygon into simpler components. Ph.D. Thesis,Univ . Of Toronto,Canada,1983. Report 163/83

[Keil85] M.Keil. Decomposing a polygon into simple components.1985.*SIAM J. Comput.* 14, 799817.

[Keil96] M.Keil. Survey: “Polygon Decomposition”, University of Saskatchewan Saskatoon, Canada,S7N 5A9, 1996

[KS98] M.Keil, J.Snoeyink. Survey: On the time bond for convex decomposition of simple polygons.*International Journal of Computational Geometry and Applications.*

[KKT90] D. Kirkpatrick, M. Klawe, R. Tarjan, Polygon triangulation in O(nloglogn) time with simple data structures, in *Proc. 6th Annu. ACM Sympos. Comput. Geom. ,* pp. 3443.

[Len11] N.J.Lennes. Theorems on the simple finite polygon and polyhedron, *Am J. Math.* 33, 3762

[LLMP79] E.Lodi, F.Luccio,C.Mugnai, and L.Pagli. On twodimensional data organization. I.*Fundam.Inform.*,2.221226,1979.

[LP77] D.T.Lee,F.P.Preparata:Location of a point in a planar subdivision and its applikation, *SIAM J. Comput.* 6, 594606.

[LPRS82] A.Lingas, R.Pinter, R.Rivest, and A.Shamir. Minimum edge length partiotioning of rectilinear polygons. *In Proc. 20*$^{\text{th}}$ *Allerton Conf.Commun. Control Comput.*,pages 5363,1982.

[Lub90] The boolean basis problem and how to cover some polygons by rectangles. *SIAM J. On Discrete Mathematics*, 3:98115,1990.

[LN88] R.Liu, S.Ntafos, On decomposing polygons into uniformly monoton parts. *Informating Processing Letters*, 27:8589,1988

[Moi91] D.Moitra. Finding a minimal cover for binary images: an optimal parallel algorithm. *Algorithmica*,6:624657,1991.

[NP82] J.Nievergelet, F.P. Preparata, Plane sweeping algortihms for intersecting geometric figures, *Commun. ACM* 25, 73947.

[O’R94] J.O’Rourke. Computational Geometry in C. Cambridge University Press,1994.

[OW92] 1992 T. Ottmann und P. Widmayer: “Algorithmen und Datenstrukturen”, BIWissenschafts verlag, Mannheim

[Pav77] T.Pavlidis. Structural Pattern Recognition. SpringerVerlag, Berlin Heidelberg,1977

[Pa91] S.Palmiteer and J.Elketron, “An Evaluation of Animated Demonstrations for Learning Computerbased Tasks”, Proceedings of ACM SIGCHI’91 Conference on Human Factors in Computing Systems, S. 257263.

[Schorn91] P.Schorn. The XYZ GeoBench: a programming environment for geometric algorithms, Lecture Notes in Computer Science 553 (Springer Verlag, Berlin,1991), pp. 187202

[Sei91] R. Seidel. A simple and fast Incremental Randomized Algorithm for Computing Trapezoidal Decomposition and for Triangulating Polygons.

[Stasko90] J.Stasko, TTango: a framework ans system for algoroithm animationÏEEE Computer, September, 1990

[StaBaL93] J.Stasko, A.Badre, C.Lewis "Do Algorithn Animation Assist Learning? An Empirical Study and Analysis", Proceedings of the INTERCHI'93 Conference on Human Factors in Computing Systems, Amsterdam, April 1993,S.6166

[Stasko97] John Stasko: "Software Visualization Programming as a Multimedia Expiriance", 1997, MIT Press.

[Stasko98] J.Stasko, A.Lawrence "Empirically Assessing Algorithm Animation as Learning Aids", In J.Stasko et al. "Software Visalization", 1998 MIT Press, Cambridge, Massachussets, London, England, S. 419438

[TaVW88] R.E.Tarjan, C.J. Van Wyk. An O(nloglogn)time algorithm for triangulating a simle polygon. *SIAM J. Comput.* 17,14378

[TalDo95] A. Tal, D.P.Dobkin. "Visualisation of geometric algorithms" IEEE Transactions on Visualization and Computer Graphics, 1:194204, 1995

[TalSchne98] A.Tal, Maria Schneerson, "Visualisation of geometric algorithms in an Electronic Classroom",

[ToMid84] S.B.Tor and A.E.Middletich. Convex Decomposition of simple polygons. *ACM Trans. Graph.*,3:244265,1984.

[Tou91] G.Toussaint. Efficient triangulation of simple polygons. *Visual Comput.*,7:280295,1991.