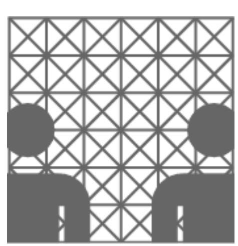
Universität Hamburg

Fachbereich Informatik
Arbeitsbereich Kognitive Systeme



Baccalaureatsarbeit

3D-Segmentierung mit Dual Simplex Meshes

Florian Heinrich
Matrikelnr.: 5409989
<florian_heinrich@arcormail.de>

Benjamin Seppke
Matrikelnr.: 5419380
<benjamin@seppke.de>

Betreuer: Dr. Ullrich Köthe

Vorwort

Diese Arbeit stellt ein Verfahren vor, das es ermöglichen soll, Objekte in Bilddaten automatisch zu segmentieren. Dabei entstand die Idee der Implementation im Rahmen des Hauptstudium Projektes „Bildverarbeitung". Diese Arbeit dient dem Abschluss dieses Projektes und stellt eine Baccalaureatsarbeit dar. Die Ausarbeitung erfolgte in einer Gruppenarbeit zweier Studierender.

Der Aufbau dieser Arbeit orientiert sich an der Vorgehensweise während des Projektes. Es wird zuerst einiges Grundlegendes zu Segmentierungsverfahren gesagt werden. Dabei werden auch einige bekannte und vielfach angewendete Verfahren kurz vorgestellt. Anschließend wird das behandelte Verfahren, die Segmentierung mit Hilfe von Dual Simplex Meshes, vorgestellt. Dabei wird, nachdem einige Grundlagen zum Verfahren erläutert werden, die Segmentierung auf zweidimensionalen Bildern näher betrachtet, ehe dann die Verarbeitung dreidimensionaler Daten betrachtet wird.

Bei der Verwendung der dreidimensionalen Daten gilt ein besonderer Dank der Firma Philips, die diese der Universität Hamburg zur Verfügung gestellt haben.

Hamburg, 21 Januar 2005

Benjamin Seppke

Florian Heinrich

Inhaltsverzeichnis

1 Einführung in Dual Simplex Meshes....1
1.1 Elementare Segmentierungsverfahren....1
1.2 Modellbasierte Segmentierung....3
1.3 Entstehung und Verwendung der Dual Simplex Meshes....4
1.4 Grundlagen für Dual Simplex Meshes....4
1.4.1 Aufbau von Maschen....4
1.4.2 Verformung der Maschen....5
1.4.3 Energie eines Knotenpunktes....6
1.4.3.1 Die interne Energie....6
1.4.3.2 Die externe Energie....6
1.5 Finden der Objekte im Bild....8
1.6 Eingabeparameter des Verfahrens von Svoboda und Matula....8
2 Formale Grundlagen....10
2.1 Definition der Dual Simplex Mesh....10
2.1.1 Simplex Mesh und Star-Shaped Simplex Mesh....10
2.1.2 Dual Simplex Mesh....11
2.2 Berechnung der internen Energie....12
2.2.1 Berechnung des Simplex Winkels in 2D....13
2.2.2 Berechnung des Simplex Winkels in 3D....15
2.3 Berechnung der externen Energie....18
2.4 Berechnung der Energie....18
2.5 Finden der Objektmarkierungen....19
2.5.1 Multilevel Adaptive Thresholding....19
2.5.2 Hough-Transformation....20
3 Ablauf des Algorithmus`....23
3.1 Benötigte Eingabewerte....23
3.2 Ausgabewerte....23
3.3 Komponenten des Algorithmus` nach Svoboda und Matula....23
3.3.1 Suche der Anfangsmarkierungen....24
3.3.2 Verformung der Maschen....24
3.4 Ablauf des Algorithmus` nach Svoboda und Matula....24
3.5 Einschränkungen der umgesetzten Implementationen....25
4 Implementation des Algorithmus`....27
4.1 Beschreibung der Arbeitsumgebung....27
4.1.1 Verwendete Hardware....27
4.1.2 Verwendete Software....28
4.2 Implementation in 2D....30
4.2.1 Erläuterung der Klassenhierarchie....30
4.2.2 Speicherrepräsentation der Maschen....31
4.2.3 Implementationen der Energien....33
4.2.3.1 Interne Energie....33
4.2.3.2 Externe Energie....34

4.2.4 Automatisches Finden der Objektmarkierungen....35
4.2.5 Ablauf des Programms....39
4.2.6 Grafische Ausgabe....42
4.2.6.1 Zeichnen der Maschen....42
4.2.6.2 Speichern der Bilder....44
4.3 Implementation in 3D....44
4.3.1 Erläuterung der Klassenhierarchie....44
4.3.2 Repräsentation der Maschen....45
4.3.2.1 Kugeltesselationen....45
4.3.2.2 Duale Graph Transformation (DGT)....47
4.3.2.3 Speicherrepräsentation der Maschen....49
4.3.3 Implementationen der Energien....49
4.3.3.1 Interne Energie....50
4.3.3.2 Externe Energie....51
4.3.4 Ablauf des Programmes....52
4.3.5 Grafische Ausgabe....53
4.3.5.1 Zeichnen der Maschen....54
4.3.5.2 Inverse Duale Graph Transformation (IDGT)....55
4.3.5.3 Grafische Darstellung der Triangulierung....57
4.3.5.4 Rotation und Projektion....59
5 Leistungsmessungen und Ergebnisse....61
5.1 Ergebnisse der Leistungsmessungen in 2D....61
5.2 Diskussion der 2D-Ergebnisse....63
5.3 Ergebnisse der Leistungsmessungen in 3D....65
5.4 Diskussion der 3D-Ergebnisse....67
6 Erfahrungen und Probleme....69
6.1 Probleme mit dem Vorlagenpaper....69
6.2 Schwierigkeiten bei der Implementation....70
6.3 Anforderungen an die Bild- bzw. Volumendaten....72
6.4 Auswirkungen durch Veränderung der Parameter in 2D....72
6.4 Auswirkungen durch Veränderung der Parameter in 3D....75
7 Vergleich der Verfahren: 2D vs. 3D....78
7.1 Anwendung des 2D-Verfahrens auf Volumendaten....78
7.2 Vergleich der Ergebnisse....79
8 Zusammenfassung und Ausblick....82

Referenzteil

A Verwendete Bilddaten....83
B Inhalt der CD....85
C Aufteilung der Ausarbeitung zwischen den Studierenden....86
Abbildungsverzeichnis....87
Literaturverzeichnis....89

1 Einführung in Dual Simplex Meshes

Zuerst soll eine Einführung in die Segmentierung im Allgemeinen erfolgen. Es wird erläutert welche Ziele Segmentierungsalgorithmen verfolgen und mit welchen Mitteln bzw. Vorgehensweisen sie versuchen diese Ziele zu erreichen.

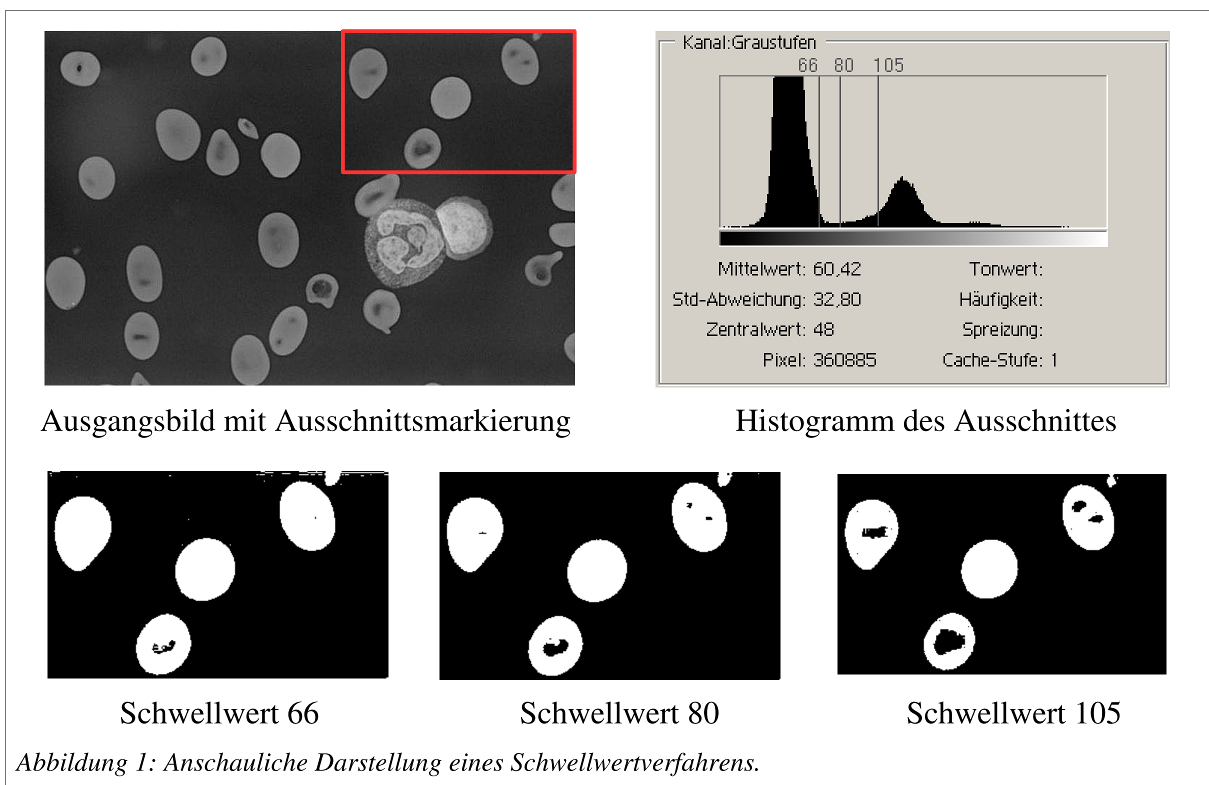
Dabei wird kurz, anschaulich und größtenteils informell der aktuelle Stand der Dinge wiedergegeben. Soweit dies nicht anders erwähnt wird, stammen diese Verfahren aus [Jäh02]. Im Anschluss an sogenannte elementare Segmentierungsverfahren wird ein Übergang zu modellbasierten Segmentierungsverfahren geschaffen, welcher schließlich zu der ersten Vorstellung der Dual Simplex Meshes führt. Am Ende dieses Kapitels erfolgen bereits erste wichtige Betrachtungen der Eigenschaften der Dual Simplex Meshes sowie einige Nomenklaturen, die in dieser Arbeit benutzt werden.

1.1 Elementare Segmentierungsverfahren

Unter den elementaren Segmentierungsverfahren werden jene verstanden, welche nur auf lokalen Informationen basieren. Zu ihnen gehören unter anderem die pixelbasierten Methoden, welche nur von den Grauwerten jedes einzelnen Pixels ausgehen, wie z.B. das Thresholding-oder Schwellwert-Verfahren, sowie regionenorientierte Verfahren, welche die Grauwerte von Regionen untersuchen. Auch kantenbasierte Methoden zählen zu ihnen, sie versuchen Kanten zu erkennen und ihnen zu folgen.

Pixelbasierte Segmentierung: Die pixelbasierten Methoden stellen die am einfachsten zu realisierende Möglichkeit der Segmentierung dar. Bei ihnen wird ein Schwellwert festgelegt, der einem Grauwert entspricht. Alle Pixel, die oberhalb dieses Schwellwertes liegen, werden daraufhin in ihrer Intensität auf das Maximum erhöht, während die Pixel mit niedrigerer Intensität auf das Minimum abgesenkt werden. Hat man beispielsweise ein Bild mit hellen Objekten auf dunklem Grund und möchte diese Objekte klar abgrenzen, so findet man einen guten Schwellwert, indem man das zum Bild gehörende Histogramm generiert, das heißt die Anzahlen aller Grauwerte zählt. Es sollten sich im idealen Fall zwei Maxima bilden, bei den Grauwerten, die den Hintergrund bilden, und bei den Grauwerten, die die Objekte ausmachen.

Der Schwellwert sollte nun zwischen diesen beiden Maxima liegen. Ein perfektes Ergebnis wird allerdings kaum gelingen, da in jedem Bild durch verschiedenste Faktoren (z.B. Aufnahmetechnik) nie hundertprozentige Grauwertübergänge zu finden sind. Nahe an einem Objekt wird also stets ein Grauverlauf bis hin zum Grauton des Hintergrundes vorliegen. Weitere Schwierigkeiten stellen ungleichmäßige Beleuchtung oder auch schlicht unterschiedlich farbige Objekte dar, denn beides erschwert die Bestimmung des optimalen Schwellwertes. Wie sich unterschiedliche Schwellwerte auf das Segmentierungsergebnis auswirken zeigt Abbildung 1.



Ausgangsbild mit Ausschnittsmarkierung — Histogramm des Ausschnittes

Schwellwert 66 — Schwellwert 80 — Schwellwert 105

Abbildung 1: Anschauliche Darstellung eines Schwellwertverfahrens.

Kantenbasierte Segmentierung: Bei dieser Methode soll der Fehler vermieden werden, dass Objekte durch den oben erwähnten weichen Übergang vom Objekt zum Hintergrund, entweder zu groß oder zu klein dargestellt werden. Für diese Methode ist der Gradient von großer Bedeutung, da seine Maxima als Startpunkt für eine Kante dienen. Unter Gradient wird eine partielle Ableitung eines Pixels in x- sowie in y-Richtung verstanden. Man tastet also das Bild zeilenweise nach einem Maximum der partiellen Grauwertdifferenzen ab. Ist ein solches gefunden, so wird versucht mit Hilfe von einem „Konturverfolgungsalgorithmus dem Maximum des Gradienten um das Objekt herum zu folgen, bis der Ausgangspunkt wieder erreicht ist“ [Jäh02]. Dabei gilt als Voraussetzung, dass ein Objekt eine zusammenhängende Region ist.

Eine weitere Möglichkeit der kantenbasierten Segmentierung ist das Wasserscheiden-Verfahren. Bei diesem Verfahren wird ein Graustufenbild als Graustufengebirge aufgefasst. Dieses wird, anschaulich erklärt, von unten mit Wasser geflutet. Diese Flutung erfolgt schrittweise, stoßen dabei zwei geflutete Gebiete aneinander, ist eine Wasserscheide gefunden.

Jedem dieser Gebiete wird dann noch ein Wert zugewiesen, der dieses repräsentiert. Als Bild ausgegeben ergeben diese Werte dann die segmentierten Objekte. Ein großer Vorteil dieses Verfahrens ist die parallele Segmentierung, da immer das gesamte Bild geflutet wird. Allerdings gibt es auch viele Nachteile. So kommt es häufig zu Übersegmentierung und auch ist das Verfahren recht anfällig bei verrauschten Daten. Zur Behebung bieten sich Glättungen z.B. durch einen Gaußfilter an. Der Übersegmentierung kann durch eine Verschmelzung der gefundenen Regionen im Anschluss an die Flutung entgegengewirkt werden. Dafür müssen

Homogenitätskriterien aufgestellt werden, deren Erfüllung dafür sorgt, dass zwei Regionen verschmolzen werden.

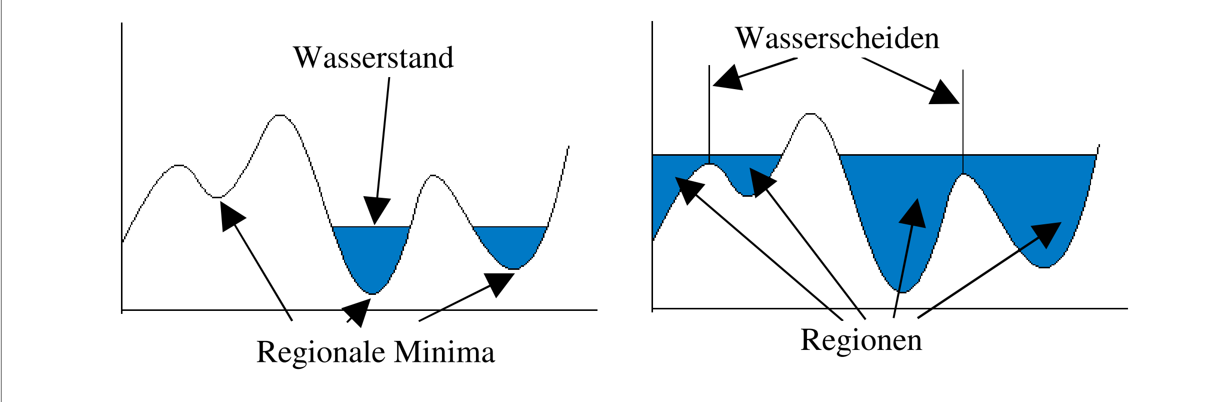


Abbildung 2: Veranschaulichung des Wasserscheidenverfahrens anhand eines Querschnittes durch das Graustufengebirge eines Bildes.

Regionenorientierte Verfahren: „Diese Verfahren betrachten Punktmengen als Gesamtheit und versuchen dadurch zusammenhängende Objekte zu finden. Häufige Verwendung finden die Verfahren des Region-Growing, Region-Splitting, Pyramid Linking und Split and Merge.“ [Wik04]

Alle diese Verfahren haben gemeinsam, dass sie jeweils nur lokale Informationen betrachten. Weiß man allerdings schon recht gut im Voraus, wie die gesuchten Objekte aussehen sollten, hat man keine Chance dieses Wissen einzubringen. Spätestens bei Bildern mit sehr schlechter Qualität hat man dann das Problem, das die Ergebnisse nicht mehr den Ansprüchen genügen. Somit mögen sie vielleicht in vielen Fällen schneller sein, als das in dieser Arbeit behandelte Verfahren der Verformung von Dual Simplex Meshes, haben aber auch einen anderen Einsatzschwerpunkt bzw. dienen nur der Vorverarbeitung der Daten, ehe ein modellbasiertes Verfahren eingesetzt wird. So kann man z.B. ein Verfahren der Schwellwertbildung (Multilevel Adaptive Thresholding) für das automatische Finden der Objektzentren (siehe Kapitel 2.5) einsetzen, denn hierfür reichen auch angenäherte Ergebnisse.

1.2 Modellbasierte Segmentierung

Hierbei wird ein Modell der gesuchten Objekte zugrundegelegt. Dies kann beispielsweise die Form betreffen, wie beim Verfahren, dem diese Arbeit gewidmet ist, bei dem von einer Kreisform bzw. Kugelform ausgegangen wird. Man setzt also Wissen über das Objekt mit ein. Ein bekanntes Verfahren ist die Hough-Transformation, mit deren Hilfe man Punkte zu Linien oder Kreisen zusammenfügen kann, indem man sie in einem Parameterraum abbildet (siehe 2.5.2). Ebenfalls finden statistische Modelle und Segmentierung über Vorlagen, sogenannte Templates (Template-Matching) Verwendung. Bei letzterem Verfahren wird im Bild nach gegebenen Vorlagen gesucht.

Einen weiteren modellbasierten Segmentierungsansatz stellen die Active Contours dar. Hierbei wird eine gegebene Kontur durch Einwirkung meist einer Vielzahl von Energien verformt. Dieses geschieht solange, bis die an der Kontur wirkenden Energien minimal sind.

Dies sollte anhand der Modellannahme dann der Fall sein, wenn die verformte Kontur das Modell des Objektes beschreibt. Jedoch besteht bei diesen Verfahren ein enormer Rechenaufwand, so dass viele Ansätze nur bei der Segmentierung zweidimensionaler Bilddaten zur Anwendung kommen.

1.3 Entstehung und Verwendung der Dual Simplex Meshes

Das Verfahren der Dual Simplex Meshes ist eine Form von Active Contours und somit ein modellbasierter Segmentierungsansatz. Das beschriebene Verfahren besitzt jedoch den Vorteil, dass die Beschränkung auf sehr wenige Energien bereits zu einem guten Ergebnis führt. Das hat zur Folge, dass natürlich auch der Rechenaufwand überschaubar ist und somit einer Verwendung auch bei dreidimensionalen Volumendaten nichts im Wege steht.

Wie bei allen Active Contours-Verfahren wird auch hier von einer gegeben Kontur ausgegangen, die dann schrittweise dem zu segmentierenden Objekt angenähert wird. Diese Methode hat große Vorteile gegenüber den bisher viel verwendeten Thresholding-Verfahren. Zum Beispiel berücksichtigen die Active Contours die Form der gesuchten Objekte, arbeiten somit mit einer Art Vorwissen oder einem Modell, wodurch eine deutliche Verbesserung der Ergebnisse bei schlechtem Bildmaterial zu beobachten ist. Stärker verrauschte oder auch fehlende Daten können deutlich besser ausgeglichen werden.

Die Dual Simplex Meshes eignen sich besonders für kreisförmige (in 2D, in 3D entsprechend kugelförmige) Objekte, und sollten daher auch Verwendung in der Medizin, z.B. zur Rekonstruktion von Zellen finden (vgl. [SM03a]). So lassen sich Zellkerne aus Gewebezellen oder auch ganze Zellen wie Blutkörper und Viren im Blut recht gut rekonstruieren bzw. segmentieren. Alle erwähnten Objekte sind in gewisser Art rund, so kann man sich dann leicht vorstellen, dass man, wenn man von einer kreisrunden Masche ausgeht, auch bei etwas Rundem landet. Auch wenn z.B. ein Teil des gesuchten Objekts gar nicht oder kaum erkennbar ist (z.B. durch Rauschen oder simples Fehlen der Daten an einigen Stellen), wird trotzdem ein rundes Objekt gefunden, dass der tatsächlichen Form sehr nahe kommt.

Um dieses garantieren zu können, wirken bei der Annäherung der Maschen an das gesuchte Objekt verschiedene Kräfte bzw. Energien, die dafür sorgen, dass die runde Form gewährleistet wird, andererseits aber auch die Bildinformationen verwendet werden.

1.4 Grundlagen für Dual Simplex Meshes

In den folgenden Kapiteln werden informell und bildhaft der Aufbau sowie die verschiedenen Energien der Dual Simplex Meshes beschrieben. Dabei handelt es sich lediglich um einen anschaulichen Einstieg in die Thematik, die formal fundierten Hintergründe hingegen finden sich in Kapitel 2.

1.4.1 Aufbau von Maschen

Da die Dual Simplex Meshes nicht viel mehr sind, als zwei Star-Shaped Simplex Meshes, sollen diese zunächst anschaulich dargestellt werden.

Um eine Star-Shaped Simplex Mesh zu erstellen bedarf es nur weniger Parameter. Zum einen benötigt man natürlich einen Punkt, der die Position der Mesh beschreibt; dieses ist dann auch der Mittelpunkt und wird im Folgenden Gravitationszentrum genannt. Um diesen Punkt legt man eine beliebige aber feste Anzahl von Knotenpunkten, die alle jeweils auf einem Strahl vom Mittelpunkt liegen, der die Masche jeweils in diesem Punkt schneidet. Der Abstand vom Mittelpunkt ist natürlich auch abhängig von der Größe des gesuchten Objekts. Als letztes sei erwähnt, dass jeder Knotenpunkt zwei direkte Nachbarn besitzt, mit denen er verbunden ist. Es ergibt sich eine Form wie in Abbildung 3.

Die Dual Simplex Mesh besteht nun aus zwei Simplex Meshes, die sich nur im Abstand der Knotenpunkte zum Mittelpunkt unterscheiden. Für eine Masche wählt man einen Abstand (Radius), so dass alle Knotenpunkte noch innerhalb des gesuchten Objektes liegen, wofür man natürlich eine ungefähre Vorstellung davon braucht, wie groß das Objekt minimal ist (dies ist dann die innere Masche). Die Knotenpunkte der zweiten Masche liegen alle außerhalb des Objekts (somit ist dies die äußere Masche). Je zwei Knotenpunkte, einer aus der inneren Masche und einer aus der äußeren Masche, bilden ein Paar und liegen auf einem gemeinsamen Strahl, der vom Mittelpunkt durch die Maschen verläuft, und diese jeweils nur einmal schneidet. Diese Eigenschaft ist sehr wichtig für die spätere Verformung der Masche und wird niemals aufgehoben. Dies veranschaulicht Abbildung 3.

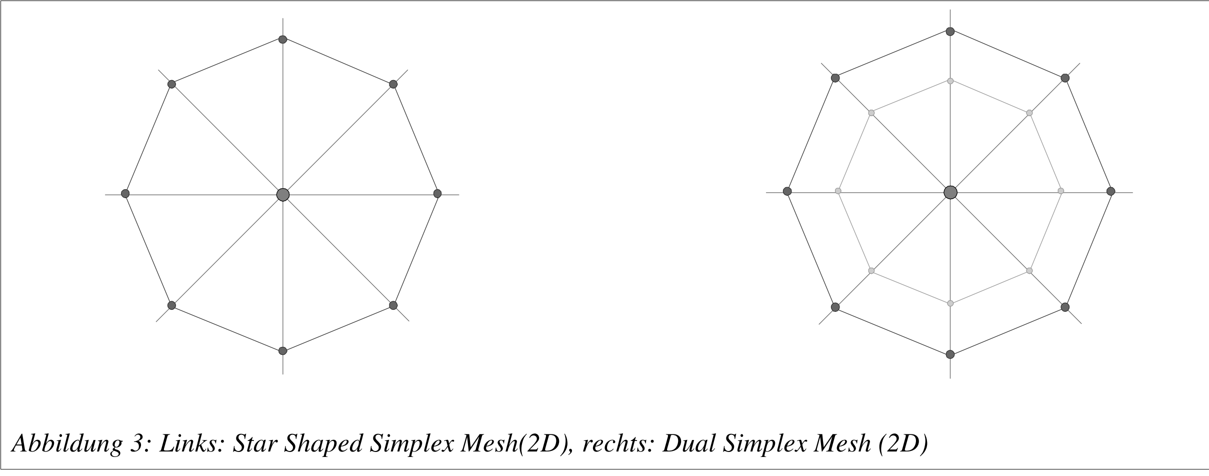


Abbildung 3: Links: Star Shaped Simplex Mesh(2D), rechts: Dual Simplex Mesh (2D)

1.4.2 Verformung der Maschen

Wie oben erwähnt, liegen je zwei Knotenpunkte auf einer Geraden, welche diese Punkte auch während der Verformung der Maschen nicht verlassen. Die Position und die Ausrichtung dieser Geraden ändert sich auch nicht, vielmehr ist sie wie eine Art Schiene, auf der die Knotenpunkte bewegt werden können. Bei der Bewegung ist allerdings zu beachten, dass sich die Knotenpunkte niemals auseinander bewegen, sondern sich in jedem Schritt stets etwas näher kommen. Die Größe für das Näherkommen muss natürlich angegeben werden, auch sollte man sich vorher Gedanken machen, welche Größe sinnvoll ist. Bei diskreten Bild- bzw. Volumendaten ist eine Schrittweite, die unter der Rasterung des Bildes bzw. des Volumens liegt im Allgemeinen nicht sinnvoll.

Um herauszufinden welcher Knotenpunkt auf der Geraden verschoben werden muss, müssen zunächst die Energien beider berechnet werden (siehe unten). Derjenige Knotenpunkt, der die größere Energie von beiden aufzuweisen hat, wird ausgewählt und um eine Schrittweite dem

anderen angenähert. Sollten beide Knotenpunkte einmal eine gleichgroße Energie besitzen, so wird stets der äußere zur Annäherung ausgewählt.

So kommen sich bei diesem Verfahren die Knotenpunkte der inneren und der äußeren Masche in jedem Schritt näher, bis sie beide denselben Abstand zum Mittelpunkt haben. Bei allen anderen Paaren von Knotenpunkten ist dies jetzt natürlich auch der Fall, da in jedem Schritt der Abstand zwischen allen Paaren gleich verkürzt wird. Hier terminiert die Verformung und das gesuchte Objekt sollte gefunden bzw. segmentiert worden sein, jedenfalls wenn man alle anzugebenden Parameter des Verfahrens optimal auf das zu verarbeitende Bild angepasst hat. Sollte das Ergebnis den Anforderungen nicht genügen, so bleibt nichts anderes übrig, als andere Werte auszuprobieren.

1.4.3 Energie eines Knotenpunktes

Die Energie eines jeden Knotenpunktes setzt sich zusammen aus zwei Energien, zum einen eine interne und zum anderen eine externe Energie, die durch einen Faktor gegeneinander gewichtet werden können. Im Folgenden werden diese beiden Energien kurz dargestellt.

1.4.3.1 Die interne Energie

Die interne Energie eines Knotens ist von dem Winkel abhängig, der durch die Verbindungslinien mit den Nachbarknoten in der Masche gebildet wird. Dieser Winkel verändert sich mit der Zeit, nämlich genau dann, wenn ein Knoten einer Masche bewegt wird, aber einer oder beide Nachbarknoten nicht, sondern der zugehörige Knoten der anderen Masche.

In den formalen Grundlagen wird vielfach von einem allgemeineren Winkelmaß ausgegangen, dem Simplex Winkel. Dieser erlaubt ein Winkel- und damit ein Krümmungsmaß für dreidimensionale Dual Simplex Meshes zu erhalten, bei denen jeder Maschenpunkt genau drei Nachbarn hat.

Es wird allerdings nicht dieser Winkel als interne Energie genommen, sondern die Abweichung von einem Referenzwinkel. Dieser Referenzwinkel ist gleich dem Anfangswinkel, durch den die optimale Kreis- bzw. Kugelform beschrieben wird. Somit bewirkt die interne Energie, dass die Maschen beim aufeinander Zulaufen eine Kreis- bzw. Kugelform beibehalten, was wiederum wichtig ist, da möglichst kreis- bzw. kugelförmige Objekte segmentiert werden sollen. Damit aber nicht immer nur Kreise bzw. Kugeln entstehen wirkt neben der internen Energie noch eine andere Energie, die externe Energie, welche im Folgenden beschrieben wird.

1.4.3.2 Die externe Energie

Die externe Energie eines Knotens hängt maßgeblich von zwei Werten ab. Zum Einen von der Intensität des Gradienten (diese lässt sich leicht am Gradientenbild oder Gradientenvolumen ablesen) an der Position des Knotens und von der Richtung dieses Gradienten (welche aus dem Gradientenvektorfeld zu entnehmen ist). Die Richtung spielt eine entscheidende Rolle, denn nur wenn die Richtung nach außen zeigt, also nicht in die Masche hinein, ist der Gradientenbetrag von Bedeutung.

Im Folgenden werden die Begriffe Gradientenbild und Gradientenvektorfeld noch einmal erläutert.

- Gradientenbild bzw. Gradientenvolumen

  Das Gradientenbild bzw. das Gradientenvolumen beschreibt den Betrag der partiellen Ableitungen eines Pixels oder Voxels in seine x-y- bzw. x-y-z-Achsen. Es wird im Allgemeinen berechnet, indem das ursprüngliche Bild bzw. Volumen mit einer Faltungsmaske gefaltet wird. Gute Ergebnisse liefert beispielsweise ein Gaußfilter. Bei einer solchen Faltung wird der Wert eines Pixels bzw. Voxels in Relation zu den Werten der Nachbarpixel bzw. der Nachbarvoxel gestellt. Je mehr der Wert des Pixels bzw. des Voxels vom Durchschnitt der anderen (eventuell mit Gauß gewichtet) abweicht, das heißt je größer die Farb- oder Grauwertdifferenz zum Umfeld ist, umso höher ist der Gradient (siehe Abbildung 4).

- Gradientenvektorfeld

  Das Gradientenvektorfeld ist ein Speicherort für die Richtungen der Gradienten. Bei zweidimensionalen Bildern müssen dementsprechend zweidimensionale Richtungsvektoren gespeichert werden. Dies kann man leicht in einem zweidimensionalen Array tun, dass die Ausmaße des Bildes hat. Um die Richtungen zu bestimmen betrachtet man einen x- und einen y-Wert. Der x-Wert ergibt sich durch Betrachtung der Pixel horizontal zum aktuellen Pixel, der y-Wert entsprechend durch die Pixel vertikal zum aktuellen. Die Masche hat es bei der Verformung schwerer gegen eine Gradientenrichtung zu laufen (siehe auch hierzu Abbildung 4).

Abbildung 4: Links ein Gradientenbild, rechts ein Gradientenvektorfeld in 2D

Während die interne Energie die Aufgabe hatte, eine möglichst runde Form der Maschen zu gewährleisten, hat die externe Energie die Aufgabe zu verhindern, dass wichtige Objektinformationen, gegeben durch die Bild- bzw. Volumendaten, verloren gehen. Die externe Energie verhindert, dass die Maschen über eine gegebene Kante des Objekts hinweg laufen. Der Gradient hält die Masche anschaulich gesagt an wichtigen Stellen fest, so dass die Masche in der Lage ist eine vollkommen runde Form aufzugeben. Eine Dual Simplex Mesh, die auf einem Schachbrettmuster verformt wird zeigt die Abbildung 5.

Die Gewichtung der beiden Energien hat somit große Auswirkungen auf das erzielte Ergebnis, manches Mal ist mehrmaliges Ausprobieren nötig, damit eine Energie nicht die andere vollkommen außer Kraft setzt, und somit zu keinem befriedigendem Ergebnis führt.

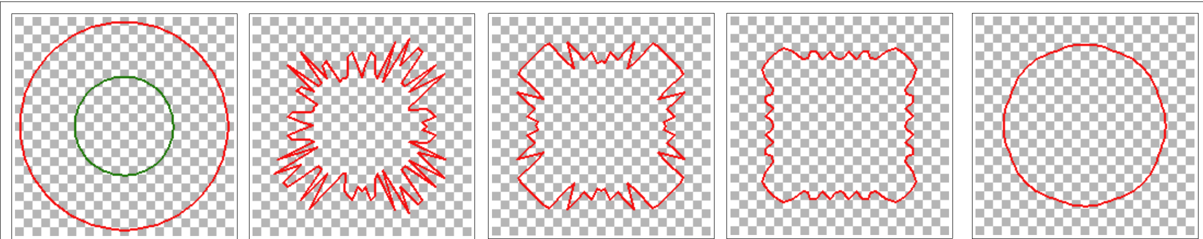


Abbildung 5: Zusammenwirken der beiden Energien auf eine Dual Simplex Mesh in 2D (linkes Bild). Zweites Bild: Volle Gewichtung auf die externe Energie. Weitere Bilder: Zunahme des Einflusses der internen Energie und entsprechend Abnahme des Einflusses der externen Energie.

1.5 Finden der Objekte im Bild

Es stellt sich natürlich die Frage, wo im Bild bzw. im Volumen genau die Maschen liegen sollen. Wie schon oben beschrieben werden die Position einer Masche durch ihren Mittelpunkt beschrieben. Um die Koordinaten dieses Punktes zu erlangen gibt es verschiedene Möglichkeiten.

Die einfachste dieser Methoden, wenn auch nicht die genaueste, ist es den Benutzer nach den Koordinaten zu fragen. Diese Möglichkeit hat aber den Nachteil, dass sie keine sehr genauen Ergebnisse liefert. Außerdem ist sie auch recht mühselig für den Benutzer, vor allem wenn man davon ausgeht, dass es in der Praxis recht häufig Eingabebilder gibt, auf denen mehr als ein Objekt zu finden ist. Auch bei Volumendaten ist es meist schwierig den Mittelpunkt eines Objektes zu schätzen, da hier schichtweise vorgegangen werden muss.

Es hat also seine Vorteile, schon vor der Segmentierung der Objekte mit ihnen zu arbeiten. So kann man durch Faltung mit bestimmten Faltungsmasken schon im Vornherein die Objektzentren herausfinden [SM03a]. Diese Methode orientiert sich an der Idee der Hough-Transformation und sieht vor, dass zum Beginn das Gradientenbild mit einer Kreismaske mit dem minimalen Durchmesser, zu falten. Das Ergebnis wird dann pixelweise vom Ursprungsbild abgezogen. Die Daten die entstehen werden erneut mit der Kreismaske gefaltet. Es entstehen lokale Maxima, von denen dann nur noch die Schwerpunkte gefunden werden müssen. Diese sind dann die Objektzentren, um welche herum die Maschen angelegt werden.

Eine weitere Möglichkeit die Objektzentren zu finden ist die des „Multilevel Adaptive Thresholding“ [SM03b]. Gegenüber dem normalen Schwellwertverfahren geht man hier davon aus, dass es mehr als nur einen Schwellwert gibt. Es werden also die Objekte im Vordergrund in unterschiedlichen Regionen des Bildes durch verschiedene Schwellwerte vom Hintergrund getrennt. Es entstehen im optimalen Fall Bereiche im Vordergrund, die den Objekten entsprechen. Aus diesen Bereichen muss dann noch jeweils der Schwerpunkt gefunden werden um die Maschenzentren zu erhalten.

1.6 Eingabeparameter des Verfahrens von Svoboda und Matula

In der Art, wie Svoboda und Matula das Verfahren vorgestellt haben, soll die Segmentierung möglichst mit wenigen Eingabedaten auskommen und somit möglichst automatisch ablaufen.

So gibt es natürlich auch nur wenige Eingabeparameter, die eingestellt werden können, um das Ergebnis zu verändern.

So können die Radien der inneren und äußeren Masche variiert werden, je nach Vorwissen über die gesuchten Objekte und deren Größe. Der Radius der inneren Masche sollte natürlich kleiner sein, als der kleinste vorkommende Radius eines gesuchten Objekts, der Radius der äußeren Masche natürlich größer als der größte Radius. Je genauer man diese Radien angibt, umso besser funktioniert, laut Svoboda und Matula, das automatische Suchen nach Objekten und deren Schwerpunkten.

Desweiteren wird noch ein Glattheitsfaktor angegeben, der der Gewichtung von interner und externer Energie entspricht. Außerdem wird noch ein Dämpfungsfaktor angegeben, der die Annäherungsgeschwindigkeit zweier Knoten beeinflusst.

2 Formale Grundlagen

Nach einem kurzen Überblick über Dual Simplex Meshes und ihrer Funktionsweise, sollen nun die formalen Hintergründe folgen. Umriss das erste Kapitel noch unscharf und recht informell die zu behandelnde Thematik, so ist es für das exakte Verständnis von allergrößter Bedeutung jene fundierten mathematischen Grundlagen zu erläutern, die das Verfahren klar und formell beschreiben und die insbesondere für eine Implementierung von Nöten sind.

2.1 Definition der Dual Simplex Mesh

Da die Dual Simplex Mesh aus zwei Star-Shaped Simplex Meshes bestehen, und diese Star-Shaped Simplex Meshes wiederum spezielle Simplex Meshes sind, wird mit der Definition der Simplex Meshes begonnen.

2.1.1 Simplex Mesh und Star-Shaped Simplex Mesh

Die Definition nach [SM01]:

Ein *0-Element* von $\mathbb{R}^d$ auch Knoten genannt ist ein Punkt in $\mathbb{R}^d$.

Ein *1-Element* aus $\mathbb{R}^d$ auch Kante genannt ist ein ungeordnetes Paar von unterschiedlichen Knoten in $\mathbb{R}^d$.

Ein *p-Element* $\mathcal{C}$ mit $p>1$ ist rekursiv definiert als eine Menge von *(p-1)-Elementen*, so dass:

1. Jeder Knoten der zu $\mathcal{C}$ gehört, gehört zu *p* unterschiedlichen *(p-1)-Elementen*.
2. Die Schnittmenge zweier *(p-1)-Elementen* ist entweder leer oder ein *(p-2)-Element*.

Ein *2-Element* ist somit ein geschlossener Polygonzug in $\mathbb{R}^d$.

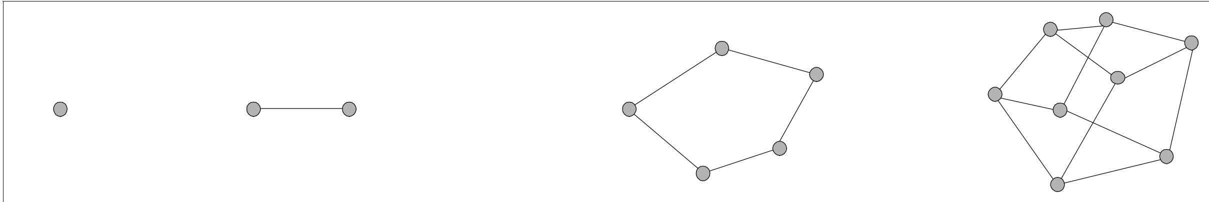


Abbildung 6: Einige p-Elemente im Überblick (von links nach rechts: 0-Element, 1-Element, 2-Element, 3-Element)

Eine *k-Simplex Mesh* in $\mathbb{R}^d$ ist ein *(k+1)-Element* aus $\mathbb{R}^d$.

Die konvexe Hülle des *p-Elementes* $\mathcal{C}$ sei mit $CH(\mathcal{C})$ bezeichnet. Sei O ein beliebiger Punkt von $\mathbb{R}^d$. Der Kegel mit dem Scheitelpunkt O in Bezug zum *p-Element* $\mathcal{C}$ von $\mathbb{R}^d$ ist die Vereinigung aller Strahlen, die bei O beginnen und $CH(\mathcal{C})$ schneiden. Dieser Kegel wird mit $\mathcal{K}(O,\mathcal{C})$ bezeichnet.

Bestehe die *k-Simplex Mesh* $\mathcal{M}$ aus *n k-Elementen* $c_i,\ i \in \{1,\ldots,n\}$. $\mathcal{M}$ ist genau dann Star-Shaped, wenn ein Punkt *O* in $\mathbb{R}^d$ existiert, so dass:

1. wenn zwei *k-Elemente* C_i und C_j sich in einem *(k-1)-Element* schneiden dann ist die Schnittmenge der Kegel $\mathcal{K}(O, C_i)$ und $\mathcal{K}(O, C_j)$ gleich dem Kegel $\mathcal{K}(O, C)$.
2. wenn zwei *k-Elemente* C_i und C_j sich nicht schneiden dann ist die Schnittmenge der Kegel $\mathcal{K}(O, C_i)$ und $\mathcal{K}(O, C_j)$ exakt der Punkt *O*.

Der Punkt *O* wird Zentrum genannt.

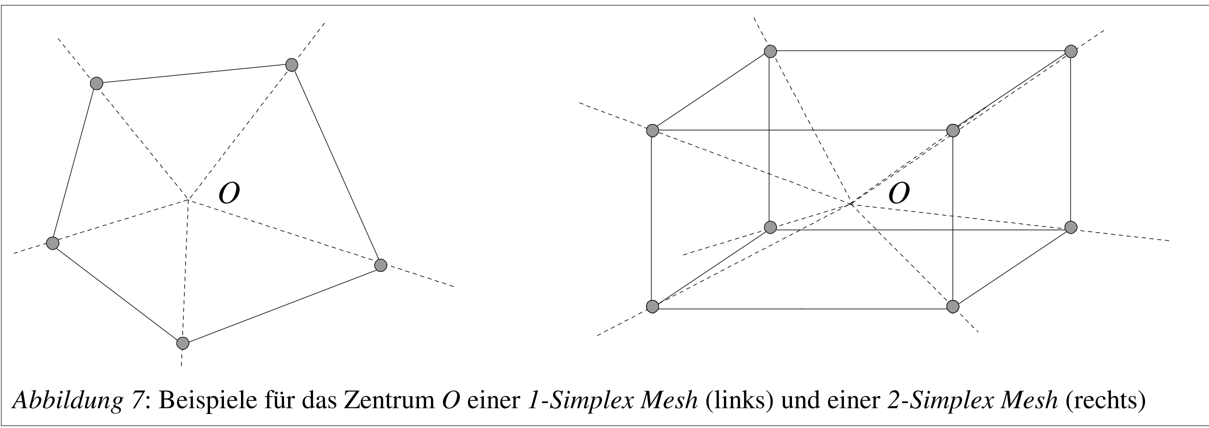


Abbildung 7: Beispiele für das Zentrum *O* einer *1-Simplex Mesh* (links) und einer *2-Simplex Mesh* (rechts)

Eine wichtige Eigenschaft der *k-Simplex Meshes* ist, dass jeder Knoten genau $k+1$ Nachbarknoten besitzt.

In diesem Kapitel wird anstatt von *1-Simplex Mesh* in 2D und *2-Simplex Mesh* in 3D einfach nur *Simplex Mesh* als Bezeichnung gewählt. In weiteren Kapiteln werden sie auch häufig abkürzend als Maschen bezeichnet.

2.1.2 Dual Simplex Mesh

Definition nach [SM03a]:

Sei $\mathcal{M}$ eine beliebige *Star-Shaped Simplex Mesh* mit dem Zentrum *Q*. Seien $\mathcal{H}(Q,\ q_i)$ die Menge der Strahlen mit dem Zentrum *Q* und den Radien $q_i,\ q_i \in \langle 1, \infty \rangle, \forall i \in \{1,\ldots,n\}$, wobei *n* die Anzahl der Knoten von $\mathcal{M}$ ist.

Sei $\mathcal{M}'$ eine andere Star-Shaped Simplex Mesh, so dass

$$\forall P_i' \in \mathcal{M}';\ P_i' = \mathcal{H}(Q,\ q_i) \cdot (P_i)\text{, mit } P_i' \in \mathcal{M},\ i \in \{1,2,\ldots,n\},$$

sowie jedes P_i' seine korrespondierenden Nachbarn $P_{i_{N1}}'$ und $P_{i_{N2}}'$ besitzt.

Die Vereinigung $D = \mathcal{M} \cup \mathcal{M}'$ wird *Dual Simplex Mesh* genannt (siehe Abbildung 12).

Da, wie bereits oben erwähnt, von runden Maschen ausgegangen wird, kann man $q_i = q_j$ für $\forall i, j \in \{1,2,\ldots,n\}$ setzen. Desweiteren bilden alle Strahlen vom Zentrum aus den gleichen

Winkel zum benachbarten Strahl. So erhält man als Form für jede einzelne *Simplex Mesh* ein regelmäßiges *n*-Eck, welches bei $n \to \infty$ in einen Kreis übergeht.

2.2 Berechnung der internen Energie

Je nach Dimensionalität der Maschen gibt es unterschiedlich viele Nachbarknoten. Da alle Nachbarknoten in die Berechnung des Simplex Winkels eingehen, welcher die Krümmung der Masche beschreibt, muss man diesen je nach Dimension anders bestimmen.

Allgemein gilt jedoch die Formel für die Berechnung der internen Energie, nachdem man den Simplex Winkel bestimmt hat. Um die interne Energie einen Knotens berechnen zu können, braucht man zusätzlich den Referenzwinkel $\tilde{\varphi}$ dieses Knotens. Dieser Referenzwinkel $\tilde{\varphi}$ eines jeden Knotens kann so gewählt werden, dass er die optimale Krümmung beschreibt. Man erhält den Referenzwinkel also, indem man bei der unverformten Masche den jeweiligen Winkel abliest. Dieses ist möglich, da die Masche am Anfang noch ihre optimale Form hat und somit auch jeder Winkel sein Referenzwinkel ist. Die interne Energie lässt sich dann mit folgender Formel berechnen:

$$E_{\text{int}}(V_i)=\left\{\begin{array}{l}\frac{\varphi-\tilde{\varphi}}{360},\ V_i\in M\\ \frac{\tilde{\varphi}-\varphi}{360},\ V_i\in M'\end{array}\right\}.$$

Man unterscheidet bei der Berechnung der internen Energie zwischen der inneren Masche *M* und der äusseren Masche *M'*, für Knoten V_i der jeweiligen Masche ist die Energie entsprechend obiger Gleichung zu berechnen.

Die Unterscheidung zwischen innerer und äußerer Masche ist ausschlaggebend für die korrekte Bestimmung der Energie eines Knotens. Denn: Der Knoten mit der größeren Energie wird bewegt. Um eine gute Form beizubehalten, ist es notwendig extremen Winkeln entgegen zu wirken. Es gibt folgende Fälle, die auftreten können:

- $\varphi \geq \tilde{\varphi}$ damit gilt $\frac{\varphi-\tilde{\varphi}}{360} \geq 0$ und $\frac{\tilde{\varphi}-\varphi}{360} \leq 0$, die interne Energie bevorzugt die innere Masche für die Bewegung, wodurch φ kleiner würde, sofern die Nachbarknoten nicht auf die gleiche Weise bewegt werden. Oder
- $\varphi < \tilde{\varphi}$ damit gilt $\frac{\varphi-\tilde{\varphi}}{360} < 0$ und $\frac{\tilde{\varphi}-\varphi}{360} > 0$, die interne Energie bevorzugt die äussere Masche für die Bewegung, wodurch φ größer würde, sofern die Nachbarknoten nicht auf die gleiche Weise bewegt werden.

Somit bewirkt die interne Energie, dass der Simplex Winkel möglichst dem Referenzwinkel entspricht.

Die Berechnung des Simplex Winkels und damit auch des Referenzwinkels wird in den folgenden Abschnitten beschrieben.

2.2.1 Berechnung des Simplex Winkels in 2D

Für die interne Energie ist der Winkel φ Ausschlag gebend, der durch zwei Geraden gebildet wird. Diese Geraden ergeben sich durch einen Knoten, den man mit seinen beiden Nachbarn verbindet (siehe Abbildung 8).

Der zweite wichtige Wert, der Referenzwinkel $\tilde{\varphi}$, lässt sich hier auch durch den optimalen Winkel für die Kreisform berechnen, sprich für $n>2$ Knoten ist $\tilde{\varphi}=(n-2)\cdot 180/n$ konstant.

Seien *M* die innere Masche und *M'* die äußere Masche der Dual Simplex Mesh und V_i die Knoten der Maschen.

Um nun den Winkel φ zu berechnen gibt es verschiedene Wege. Ein einfacher Weg ist es, die absoluten Steigungen der beiden Geraden zu berechnen (Abbildung 8).

Gegeben: Ein Maschenpunkt V_i mit den Koordinaten (x_i, y_i)

und seine beiden Nachbarknoten $V_{i-1}=(x_{i-1}, y_{i-1})$ sowie $V_{i+1}=(x_{i+1}, y_{i+1})$.

Dann gilt: $$\varphi=\left(\arctan\left(\frac{|y_i-y_{i-1}|}{|x_i-x_{i-1}|}\right)+\arctan\left(\frac{|y_i-y_{i+1}|}{|x_i-x_{i+1}|}\right)\right).$$

Allerdings ist bei einer Implementation zu beachten, dass die Nenner auch Null werden können. Rein formal stellt dies kein Problem dar, denn der Arcustangens ist für diesen Fall durchaus definiert und zwar als $\pi/2$.

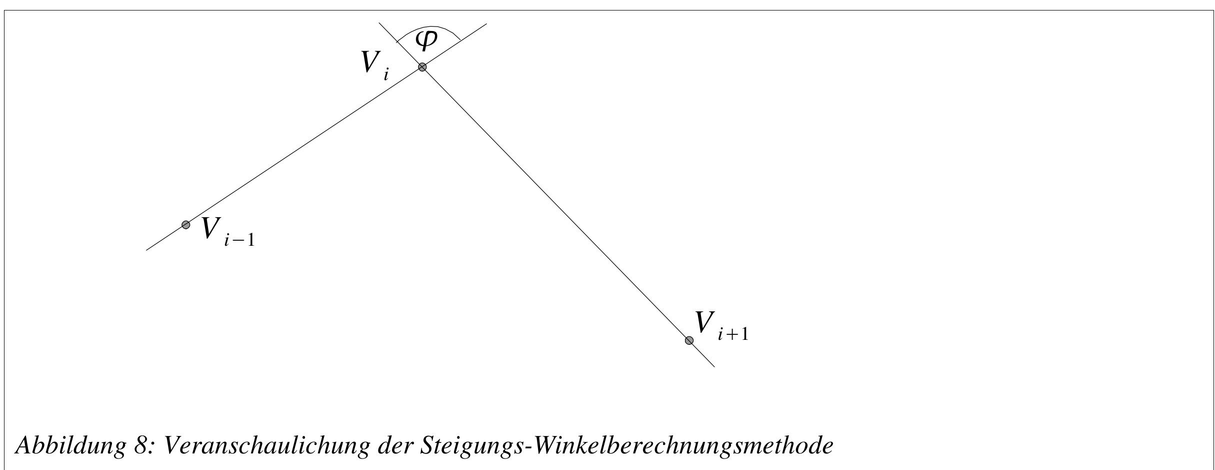


Abbildung 8: Veranschaulichung der Steigungs-Winkelberechnungsmethode

Noch einfacher ist es natürlich, mit Vektoren zu rechnen, was im dreidimensionalen Fall große Vorteile besitzt, da es genau auf die gleiche Art eingesetzt werden kann. Denn auch dort müssen Winkel zwischen zwei Geraden im Raum berechnet werden (siehe nächster Absatz). Der Schnittwinkel zweier Geraden entspricht dem Schnittwinkel ihrer Richtungsvektoren, so lässt sich das Problem auf den Winkel zwischen zwei Vektoren zurückführen, wobei man als Vektoren die Richtungsvektoren der beiden Geraden einsetzt (Abbildung 9).

Gegeben seien also zwei Geraden $g: \vec{x} = \vec{p}+r\cdot\vec{u}$ und $h: \vec{x} = \vec{q}+s\cdot\vec{v}$, die sich schneiden. Dann gilt für ihren Schnittwinkel φ : $\cos(\varphi) = \frac{\vec{u}\cdot\vec{v}}{|\vec{u}|\cdot|\vec{v}|}$.

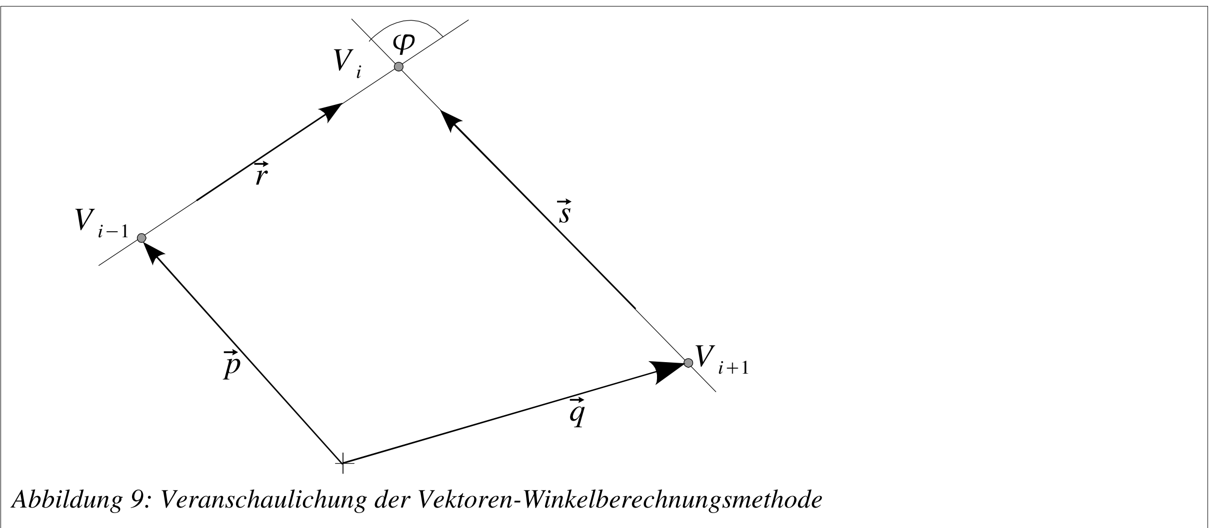


Abbildung 9: Veranschaulichung der Vektoren-Winkelberechnungsmethode

In beiden Fällen der Berechnung des Schnittwinkels erhält man allerdings keine Informationen darüber, ob es sich in Bezug auf die Masche um einen konvexen oder einen konkaven Winkel handelt. Denn es kann natürlich auch vorkommen, dass die Masche an der Stelle des gerade betrachteten Knotens eine Delle hat, es sich also um einen konkaven Winkel handelt. Um nun aus dem berechneten Schnittwinkel den Simplex Winkel bestimmen zu können, muss man vorher entscheiden, welcher Art der Schnittwinkel ist. Der Simplex Winkel soll für jeden Fall im richtigen Verhältnis zum Referenzwinkel stehen, so dass die Krümmung bestimmt werden kann. Um dieses Problem zu umgehen, kann man den Winkel auch in zwei Winkel α und β aufteilen (Abbildung 10). Durch diese Aufteilung erhält man immer den korrekten Simplex Winkel. Der Simplex Winkel berechnet sich somit als:

$$\varphi=\alpha+\beta \ .$$

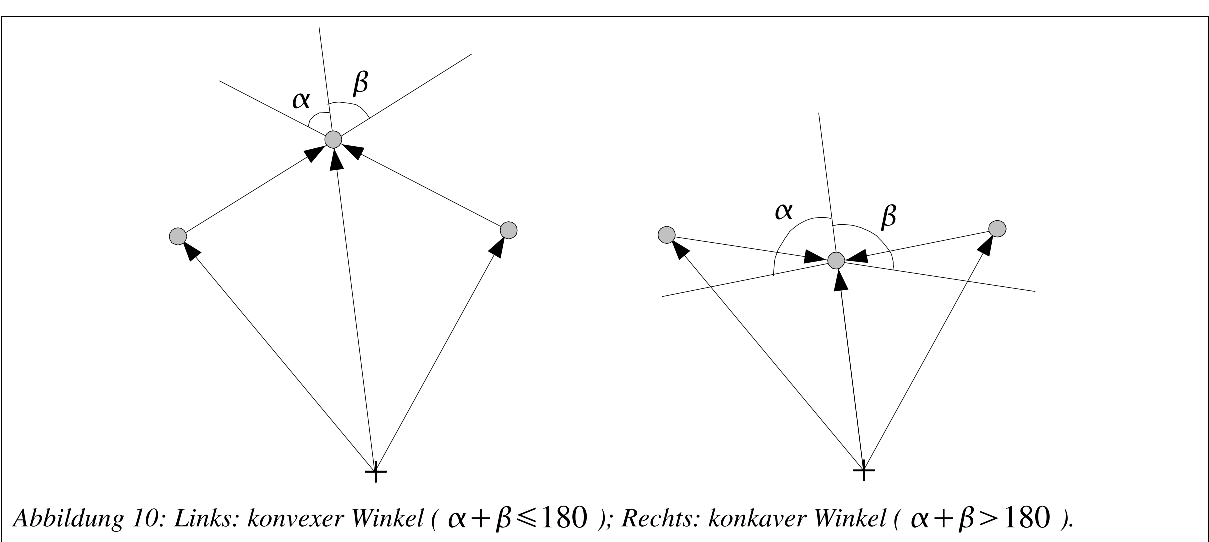


Abbildung 10: Links: konvexer Winkel ($\alpha+\beta\leqslant 180$); Rechts: konkaver Winkel ($\alpha+\beta>180$).

2.2.2 Berechnung des Simplex Winkels in 3D

Wie die interne Energie im zweidimensionalen Fall, so ist auch die interne Energie im dreidimensionalen Fall ein Maß für die Krümmung der Masche. Konnte man diese im zweidimensionalen Fall jedoch noch recht einfach als Schnittwinkel zweier Geraden berechnen, hat man nun das Problem, dass jeder Knoten drei Nachbarn besitzt. Einen Schnittwinkel von einem Paar von Geraden kann man nun also nicht mehr einfach berechnen.

Um dennoch ein geeignetes Maß für die Krümmung zu erhalten führt Delingette in [Del94] den Simplex Winkel ein. Dieser lässt sich anhand von Radien berechnen. Der Simplex Winkel eines Knotens P_i ergibt sich anhand:

$$\sin(\varphi_i)=\frac{r_i}{R_i}sign((P_{N_1(i)}-P_i)\cdot n_i)$$

Dabei ist φ_i der Simplex Winkel. Der Radius r_i ist der Radius des Umkreises der drei Nachbarn $\left(P_{N_k(i)}\, für\, k\in\{1,2,3\}\right)$, R_i ist der Radius der Kugel K_i mit Mittelpunkt O_i, auf der alle vier Knoten liegen (im Folgenden Umkugel genannt). Sollte der Knoten auf der Ebene liegen, die durch seine Nachbarn definiert wird, dann ist der Simplex Winkel Null.

Das Vorzeichen bestimmt sich dann anhand eines Skalarproduktes zweier Vektoren. Zum einen nimmt man den Richtungsvektor zwischen dem betrachteten Knoten und einer seiner Nachbarn, zum anderen den Normalenvektor der Ebene, die von den Nachbarn aufgespannt wird.

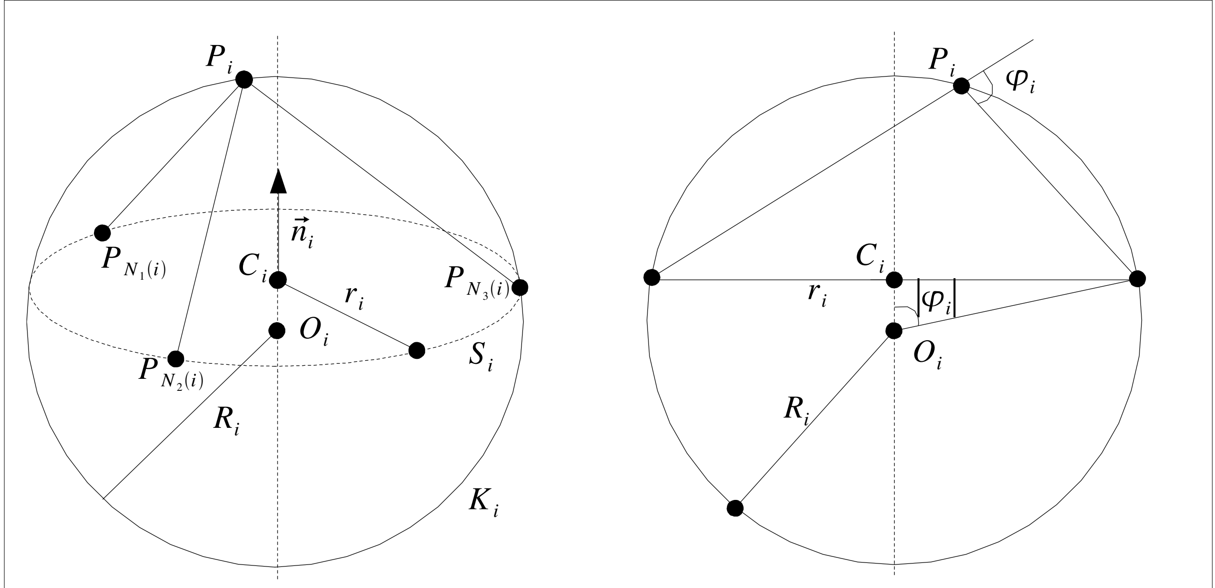


Abbildung 11: Bedeutung des Simplex Winkels φ_i. Links: Umkugel von vier Punkten. Rechts: Projektion der linken Abbildung.

Um den Simplex Winkel zu erhalten muss man zunächst alle anderen Werte berechnen. Der Radius des Umkreises der Nachbarknoten lässt sich relativ einfach über den Sinussatz

berechnen. Dieser besagt: $\frac{a}{\sin(\alpha)}=\frac{b}{\sin(\beta)}=\frac{c}{\sin(\gamma)}=2\,r_{Umkreis}$. Alles was man braucht um den Radius zu berechnen ist also eine Seitenlänge des durch die Nachbarknoten gegebenen Dreiecks und den dazu gegenüberliegenden Winkel. Wie man die Winkel berechnet, wurde bereits in (2.2.1) beschrieben.

Die Länge einer Seite ergibt sich als Länge des Vektors von einem Nachbarknoten zum anderen, also: $a=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$.

Den Radius der Umkugel kann man ebenfalls berechnen, allerdings muss hier beachtet werden, dass alle betrachteten Knoten auf einer Ebene liegen können. Dieses würde bedeuten, dass der Radius der Umkugel unendlich groß würde. Dies wiederum hat zur Folge, dass der Simplex Winkel auf Null gesetzt wird. Ein mögliches Verfahren zur Bestimmung des Radius' der Umkugel liefert [Bou02].Nehme man an, dass die vier Punkte nicht auf einer Ebene liegen, so läßt sich der Radius eindeutig bestimmen.

Seien die Punkte folgendermaßen gegeben:

$P_1=\begin{pmatrix}x_1\\x_1\\z_1\end{pmatrix}$, $P_2=\begin{pmatrix}x_2\\x_2\\z_2\end{pmatrix}$, $P_3=\begin{pmatrix}x_3\\x_3\\z_3\end{pmatrix}$ und $P_4=\begin{pmatrix}x_4\\x_4\\z_4\end{pmatrix}$,

dann läßt sich die Eigenschaft $\begin{vmatrix}x^2+y^2+z^2 & x & y & z & 1\\ x_1^2+y_1^2+z_1^2 & x_1 & y_1 & z_1 & 1\\ x_2^2+y_2^2+z_2^2 & x_2 & y_1 & z_2 & 1\\ x_3^2+y_3^2+z_3^2 & x_3 & y_1 & z_3 & 1\\ x_4^2+y_4^2+z_4^2 & x_4 & y_1 & z_4 & 1\end{vmatrix}=0$ ausnutzen.

Dabei sei $\begin{pmatrix}x\\y\\z\end{pmatrix}$ ein beliebiger Punkt auf der Kugel, und es gelte $x^2+y^2+z^2=r^2$.

Mit Hilfe des Entwicklungssatzes kann man die Determinante berechnen. Bei der Entwicklung nach der ersten Zeile entsteht die Gleichung:

$$(x^2+y^2+z^2)\cdot M_{11}-x\cdot M_{12}+y\cdot M_{13}-z\cdot M_{14}+M_{15}=0 \ ,$$

wobei M_{1k} die jeweiligen Determinanten sind, die bei Anwendung des Entwicklungssatzes entstehen.

Mit der obigen Bedingung $x^2+y^2+z^2=r^2$ erhält man

$$r^2-\frac{x\cdot M_{12}}{M_{11}}+\frac{y\cdot M_{13}}{M_{11}}-\frac{z\cdot M_{14}}{M_{11}}+\frac{M_{15}}{M_{11}}=0 \ . \quad \textit{(1)}$$

Desweiteren gelte für eine Kugel mit einem Radius r_0 und einem Mittelpunkt $\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$ die

Gleichung: $(x-x_0)^2+(y-y_0)^2+(z-z_0)^2-r_0^2=0$,

welche sich zu $r^2-2\cdot x\cdot x_0-2\cdot y\cdot y_0-2\cdot z\cdot z_0+x_0^2+y_0^2+z_0^2-r_0^2=0$ ***(2)***

vereinfachen läßt.

Setzt man nun ***(1)*** und ***(2)*** gleich, so ergibt sich:

$$x_0=0.5\cdot\frac{M_{12}}{M_{11}}\ ,\ y_0=-0.5\cdot\frac{M_{13}}{M_{11}}\ ,\ z_0=0.5\cdot\frac{M_{14}}{M_{11}}\ ,\ r_0^2=x_0^2+y_0^2+z_0^2-\frac{M_{15}}{M_{11}}\ .$$

Man ziehe noch die Wurzel und erhält den gewünschten Radius R. Liegen alle vier Punkte auf einer Ebene, so würde die Berechnung von M_{11} fehlschlagen. Sollte dieses also passieren, so ist der Simplex Winkel, wie schon vorher erwähnt, auf Null zu setzen.

Bei der Berechnung der Oberflächennormalen ist zu beachten, dass sie in die richtige Richtung zeigt. Dabei ist in diesem Fall gemeint, dass die Richtung, in die sie zeigt, und das Maschenzentrum auf unterschiedlichen Seiten der Ebene liegen. Um also die Oberflächennormale einer Ebene zu berechnen, braucht man drei Punkte P_0, P_1 und P_2 (gegeben durch die drei Nachbarknoten). Dann geht man folgendermaßen vor:

- Einen Punkt als Bezugspunkt wählen (P_0).
- Die beiden Vektoren $v_1=P_1-P_0$ und $v_2=P_2-P_0$ bestimmen.
- Diese beiden Vektoren kreuzmultiplizieren.
- Auf Richtung untersuchen und gegebenenfalls Richtung umkehren.

Das Kreuzprodukt zweier Vektoren $v_1=\begin{pmatrix} x_1 \\ x_1 \\ z_1 \end{pmatrix}$ und $v_2=\begin{pmatrix} x_2 \\ x_2 \\ z_2 \end{pmatrix}$ ist gegeben durch

$$v_1\times v_2=\begin{pmatrix} y_1 z_2-z_1 y_2 \\ z_1 x_2-x_1 z_2 \\ x_1 y_2-y_1 x_2 \end{pmatrix}\ .$$

Anschließend multipliziert man diesen Vektor dann noch mit dem Vektor $(P_{N_1(i)}-P_i)$, das Ergebnis muß nur noch hinsichtlich seines Vorzeichens untersucht werden, was die Signum-Funktion übernimmt. Nachdem auch dieses berechnet ist, stehen alle benötigten Werte zur Verfügung, um den gewünschten Simplex Winkel bestimmen zu können.

2.3 Berechnung der externen Energie

Zur Berechnung der externen Energie sind zuerst zwei verschiedene Repräsentationen der Bild- bzw. Volumendaten nötig. Diese werden im Folgenden erläutert:

- I Die Bild- bzw. Volumendaten, wie sie zur Verarbeitung vorliegen.
- $|\nabla I|$ Das Gradientenbild bzw. Gradientenvolumen welches aus I berechnet wird.

Jedem Pixel im Bild bzw. jedem Voxel im Volumen können zwei Werte I und $|\nabla I|$, welche nur einmal berechnet werden müssen, zugewiesen werden. Sei nun $l_i \in D$ eine Verbindungslinie zwischen einem Paar von Knoten P_i und P_i' (mit $P_i \in M$ und $P_i' \in M'$) mit dem Richtungsvektor $\vec{v}$. Dann ergibt sich die externe Energie eines Pixels A als:

$$E_{ext}(A) = 1.0 - \delta_{A,\vec{v}} \left(\frac{|\nabla I(A)|}{max(|\nabla I|)} \right) \text{, wobei: } \delta_{A,\vec{v}} = \begin{cases} 1, & \nabla I(A) \cdot \vec{v} > 0 \\ 0, & sonst \end{cases}$$

und $max(|\nabla I|)$ das Maximum aller Werte von $|\nabla I|$ ist, durch dessen Division der Wert der externen Energie normalisiert wird.

2.4 Berechnung der Energie

Betrachtet werde ein Paar von Knoten P_i und P_i' mit den, wie vorher beschrieben, berechneten internen bzw. externen Energien $E_{\text{int}}(P_i)$, $E_{ext}(P_i)$, $E_{\text{int}}(P_i')$, $E_{ext}(P_i')$. P_i sei der Punkt der inneren Masche, P_i' entsprechend der der äußeren. Die Energie dieser Punkte ergibt sich folgendermaßen:

$$E(P_i) = \alpha \cdot E_{\text{int}}(P_i) + (1-\alpha) \cdot E_{ext}(P_i)$$

$$E(P_i') = \alpha \cdot E_{\text{int}}(P_i') + (1-\alpha) \cdot E_{ext}(P_i')$$

hierbei ist α der Faktor, der die Glattheit des rekonstruierten Objekts bestimmt.

Die Energie der beiden Paarknoten ist ausschlaggebend für die Verformung der Masche. Hierzu vergleicht man einfach die beiden Energien beider Knoten, es ergeben sich drei Fälle:

1. $E(P_i) > E(P_i')$ P_i ist weniger stabil und wird in Richtung P_i' bewegt,
2. $E(P_i) = E(P_i')$ P_i' wird in Richtung P_i bewegt,
3. $E(P_i) < E(P_i')$ P_i' ist weniger stabil und wird in Richtung P_i bewegt.

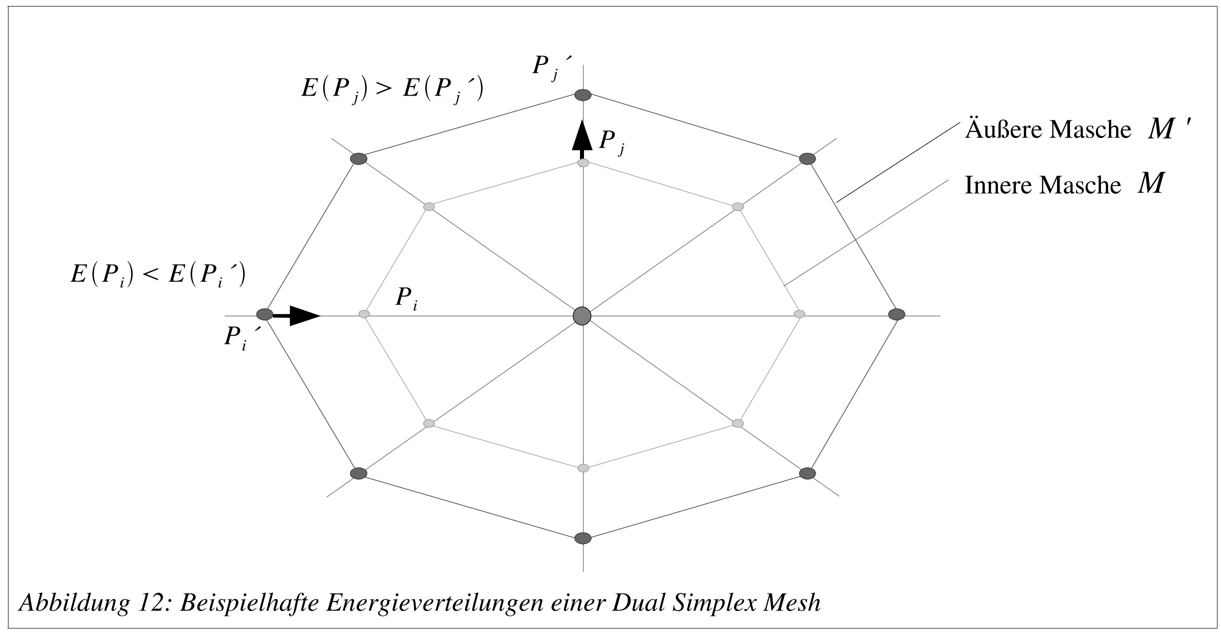


Abbildung 12: Beispielhafte Energieverteilungen einer Dual Simplex Mesh

2.5 Finden der Objektmarkierungen

Der mathematische Hintergrund für das Finden der Objektmarkierungen teilt sich in zwei Ansätze auf. Die Autoren beschreiben in [SM03b] einen Ansatz, der unter „Multilevel Adaptive Thresholding“ bekannt ist und in [SM03a] eine andere Möglichkeit Objektmarkierungen zu finden, die an die Hough-Transformation angelehnt ist. Bei der Implementation soll nur letzere Möglichkeit Verwendung finden, dennoch sollen im formalen Teil dieser Arbeit die mathematischen Grundlagen beider Verfahren kurz erläutert werden, damit die Funktionsweise derer daraus ersichtlich wird.

Es wird mit dem „Multilevel Adaptive Thresholding“ Verfahren begonnen und im Anschluss daran eine kleine Einführung in die Hough-Transformation gegeben. Zum Abschluss dieses Kapitels wird auf den Zusammenhang zwischen Hough-Transformation und dem implementierten Verfahren zum Finden der Objektmarkierungen hingewiesen.

2.5.1 Multilevel Adaptive Thresholding

Bevor das „Multilevel Adaptive Thresholding“ vorgestellt wird, wird vorher noch das „einfache“ Schwellwertverfahren beschrieben.

Es ist eine sehr schnelle und gleichfalls einfache Methode ein Bild zu segmentieren. Der Schwellwert kann manuell eingegeben oder auch automatisch berechnet werden. Bei dem hier vorgestellten Verfahrens erfolgt die Berechnung des Schwellwertes automatisch. Er ergibt als gewichtete Summe der Intensitäten aller Pixel des Bildes, wobei die Gewichte jeweils den Gradienten der Pixel entsprechen:

$$T = \sum_{p \in I} \frac{I(p) * |\nabla I(p)|}{|\nabla I(p)|}$$

Hierbei ist T der Schwellwert, I stellt die Bilddaten dar, und ∇I ist das Gradientenbild. Da bei Zellbildern oft viele Zellen nahe beieinander liegen, ist die Verwendung des Gradienten hier das einzige Hilfsmittel, das zur Verfügung steht.

Der große Nachteil des beschriebenen Schwellwertverfahrens ist, dass es bei Realwelt-Bildern oft zu ungleichmäßigen Ausleuchtungen kommt. Dieses führt zu unerwarteten und unerwünschten Effekten wie Über- oder Untersegmentierung. Um diesem Problem Herr zu werden, wurde das „Adaptive Thresholding“ eingeführt. Dabei wird ein Bild zuerst in Blöcke aufgeteilt. Anschließend wird das einfache Thresholding auf jeden der Blöcke angewandt. Dennoch hat man auch hier noch weiterhin mit dem Problem der Über- bzw. Untersegmentierung zu kämpfen.

Um diese Probleme zu umgehen, wird ein weiterer Gedanke verfolgt: Bilder haben in unterschiedlichen Bereichen oft auch ganz unterschiedliche Schwellwerte. Wird beim „Adaptive Thresholding“ beispielsweise ein zu kleiner Schwellwert gewählt, so tritt das Problem der Untersegmentierung auf, und dies müsste korrigiert werden.

Deswegen geht die Methode des „Multilevel Adaptive Thresholding“ einen anderen Weg. Zuerst wird zwar, wie beim „Adaptive Thresholding“, das Bild in Blöcke aufgeteilt. Dann wird auf jeden Block das einfache Thresholding angewandt. Hinzu kommt jetzt allerdings, dass auf alle Pixel, die nach dem ersten Thresholding noch übrig bleiben, erneut einem Thresholding-Verfahren unterzogen werden. Alle Pixel, die unter dem Schwellwert des ersten Thresholding-Durchgangs liegen, werden verworfen. Dies drückt die folgende Formel aus:

$$T_{MAT} = \sum_{p \in I,\ I(p) > T} |\nabla I(p)|$$

An dieser Stelle sei darauf hingewiesen, dass das „Multilevel Adaptive Thresholding“ alleine noch nicht dazu führen kann, dass die Unter- oder Übersegmentierung vollständig beseitigt wird.

2.5.2 Hough-Transformation

Die Hough-Transformation stellt eine robuste modellbasierte Methode zum Finden geometrischer Strukturen, welche analytisch beschrieben werden können, dar. Sie arbeitet auf den Kantenpunkten eines Bildes oder Volumens, also auf dessen Gradienten. Sie erkennt so zum Beispiel Geraden und Kreise, und mit ersterem soll sie auch eingeführt werden (vgl [Jäh02]).

Hough-Transformation von Geraden in 2D

Gegeben: Das Kantenbild (Gradientenbild) $|\nabla I|$

Gesucht: Geraden, die durch die diese Kanten verlaufen

Eine Gerade ist im Zweidimensionalen durch folgende Gleichung gegeben:

$$y = m \cdot x + b \ ,$$

wobei x, y die Koordinaten des Punktes, m die Geradensteigung und b den Ordinatenabschnitt bezeichnen.

Zur Hough-Transformation ist es nun erforderlich vom Bildraum in den Parameterraum zu wechseln. Im Fall der Gerade wird von einem x, y-Koordinatensystem in ein m, b-Koordinatensystem gewechselt. Einen solchen Wechsel veranschaulicht Abbildung 13.

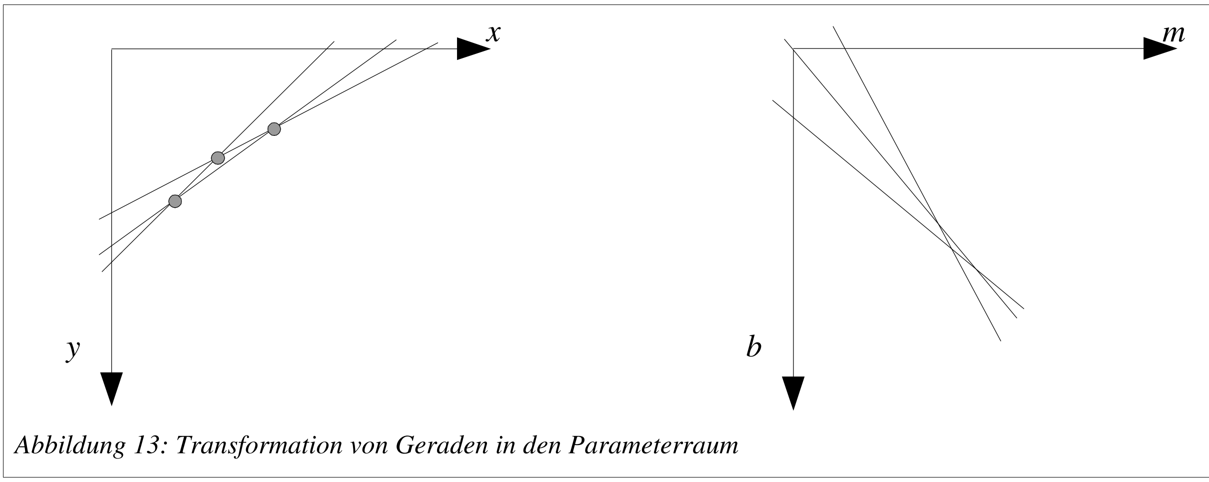


Abbildung 13: Transformation von Geraden in den Parameterraum

In der Praxis wird eine m, b-Ebene häufig nicht benutzt, da die Steigung m unter anderem unendlich sein kann. Statt dessen wird eine r, ϕ-Ebene zur Geradentransformation benutzt und mit einem Winkel ϕ gearbeitet, da hier eine unendliche Steigung ($\phi = 90°$) keinerlei Probleme darstellt.

Zur Transformation muss der Parameterraum diskretisiert werden. Anschließend wird die Transformation für jeden Pixel des Gradientenbildes durchgeführt. Nachdem alle Pixel überführt wurden, wird nach Maxima gesucht. Deren Werte im Parameterraum entsprechen dann den Parametern im Bildraum der gefunden Geraden. Falls Cluster als Maxima gefunden werden, so müssen diese erst auf einen Punkt reduziert werden.

Hough-Transformation von Kreisen

Nach dem gleichen Prinzip wie es bei der Transformation von Geraden der Fall war, lassen sich auch Kreise in den Parameterraum überführen. Dies ist für das implementierte Segmentierungsverfahren besonders interessant, wird doch von einer annähernd kreisrunden Zellform ausgegangen und nach den jeweiligen Kreis- beziehungsweise Zellmittelpunkten gesucht.

Ein Kreis mit Radius r und Mittelpunkt ergibt sich zu:

$$(x_k - x_0)^2 + (y_k - y_0)^2 = r^2$$

Es ergibt sich also ein Raum mit 3 Parametern, nämlich: x_0, y_0, r. Auch in diesen Parameterraum kann transformiert werden und dann analog zur Hough-Transformation, wie sie für Linien beschrieben wurde, vorgegangen werden.

Der Raum reduziert sich allerdings um eine Dimension, wenn man den Radius der Kreise von vornherein kennt. So bleibt man, wie bei der Hough-Transformation von Linien, in einem 2-dimensionalen Raum der Parameter. Und genau davon wird im folgenden Gebrauch gemacht.

Es sei an dieser Stelle auf die Analogien hingewiesen, die zwischen der Hough-Transformation und dem Finden der Objektmarkierungen bestehen. Man beachte hierbei speziell die Abbildung 19.

3 Ablauf des Algorithmus`

Im Folgenden soll der Algorithmus, der zur Segmentierung dient, eingeführt werden. Da der Algorithmus eine gewisse Komplexität aufweist, werden hier noch einige Bestandteile separat erläutert werden. Dies ist unter anderem notwendig, da der Algorithmus aus mehreren Teilalgorithmen besteht, die zusammen den gesamten Ablauf beschreiben. Diese Teilalgorithmen werden zunächst als Komponenten des Verfahrens beschrieben. Abschließend wird der gesamte Algorithmus als Summe der Einzelteile beschrieben.

3.1 Benötigte Eingabewerte

Ziel des Verfahrens ist es, das mühselige Auswählen vieler Parameter und bestimmter Objekte zu vermeiden. Es wurde also versucht die benötigten Daten in der Anzahl gering zu halten, um nur die wichtigsten angeben zu müssen. Das Wichtigste ist natürlich die Angabe eines Bildes oder Volumens, welches verarbeitet werden soll. Zur Verarbeitung sind dann noch vier weitere Werte nötig. Wichtig sind hier vor allem die Radien der Zellen. Wie oben beschrieben, benötigt man einen minimalen (d_{in}) und einen maximalen (d_{out}) Radius derselben. Anhand dieser Radien werden dann die Ausgangsmaschen angelegt.

Der Glattheitsfaktor α ist die wichtige Eingabe, um die beiden beschriebenen Energien gegeneinander zu gewichten. Je größer dieser Faktor ist, umso mehr Bedeutung kommt der internen Energie zu.

Als letztes geben die Autoren Svoboda und Matula noch einen Dämpfungsfaktor γ an, der die Annäherung zweier Knoten beeinflusst, und somit Auswirkungen auf die Maschenverformung hat. Er stellt eine Dämpfung der Annäherungsbewegung zweier Maschenpunkte gemäß dem Newtonschen Gesetz der Bewegung dar (siehe[SM03a]).

3.2 Ausgabewerte

Die Ausgabe besteht natürlich aus den segmentierten Objekten. Nach der Durchführung des Verfahrens geschieht dies durch die Angabe einer Menge von fertig verformten Maschen. Im Zweidimensionalen kann man diese Maschen optional auch als Konturlinien in das Ursprungsbild einzeichnen, was eine Beurteilung der Oberfläche leichter macht, und zudem das Ergebnis hinsichtlich seiner Vollständigkeit und Korrektheit leichter bewerten lässt. Für die dreidimensionale Variante des Algorithmus geben Svoboda und Matula Drahtgittermodelle der verformten Maschen als Ergebnisse aus. Denkbar sind aber auch andere Möglichkeiten der grafischen Ausgabe (vgl. 4.3.5).

3.3 Komponenten des Algorithmus` nach Svoboda und Matula

Das Verfahren wie es von Svoboda und Matula vorschlagen wurde, lässt sich recht gut in 2 Komponenten aufteilen. Diese Komponenten ergänzen sich dann zu dem kompletten Algorithmus.

3.3.1 Suche der Anfangsmarkierungen

Das Ziel dieser Komponente ist es, automatisch die Anfangsmarkierungen zu finden. Dies sind in diesem Fall die Gravitationszentren der Dual Simplex Meshes, die bei Bild- bzw. Volumendaten die angenäherten Schwerpunkte der Zellen darstellen, die es zu segmentieren gilt. Es sollten sich am Ende der Suche keine zwei Markierungen in einem zu segmentierenden Objekt befinden (Korrektheit), und es sollten genau so viele Markierungen wie zu segmentierende Zellen vorhanden sein (Vollständigkeit).

Svoboda und Matula schlagen hier einen Algorithmus vor, der sich an die *Hough-Transformation* (siehe 2.5.2) anlehnt. Damit die Faltungsoperationen nicht zu viel Zeit in Anspruch nehmen, sollten sie im Frequenzraum durchgeführt werden. Hier stellen sie – im Gegensatz zum Bild- bzw. Volumenraum – lediglich Multiplikationen dar.

Svoboda und Matula verwenden diesen Algorithmus hauptsächlich wegen seiner Robustheit gegenüber Störungen wie Rauschen. Ansonsten wären wohl auch Wasserscheidenverfahren in Betracht gekommen, jedoch bieten diese eine weitere Gefahr, die Übersegmentierung.

Die Komponente findet sich in dem kompletten Algorithmus in den Schritten 1.-7. wieder

3.3.2 Verformung der Maschen

Nach dem Finden der Anfangsmarkierungen werden die Maschen erstellt. Die Menge der Markierungen und die beiden Radien (d_{in} und d_{out}) bilden hierbei die Grundlage für die Erstellung der Dual Simplex Meshes. Jede Masche hat ihr eigenes Zentrum, welches einer der Markierungen entspricht. Hier herum werden die Knoten entsprechend der Radien angeordnet, d_{in} für die innere Masche und d_{out} für die äußere. Somit sind in dieser Phase noch die Formen aller Maschen gleich. Zu dieser Phase muss dann auch noch ein weiterer Bild- bzw. Volumenpuffer für die externe Energie berechnet werden, das Gradientenvektorfeld. Das Gradientenbild bzw. Gradientenvolumen wurde bereits in (3.3.1) erstellt. Aus diesen beiden Speichern lassen sich nun für die benötigten Pixel die externen Energien berechnen.

Die Verformung findet in einem iterativen Prozess statt. In jedem Schritt werden alle Objekte im Bild bzw. im Volumen und somit alle Verbindungslinien zwischen je zwei korrespondierenden Knoten nach dem Gesetz der Bewegung behandelt. Somit wird in jedem Iterationsschritt jede Verbindungslinie um die gleiche Weise verkürzt. Da alle Verbindungslinien endliche Länge besitzen, und die Verkürzung derselben auch sicher terminiert, so terminiert der Algorithmus für jede Masche sicher (siehe hierzu 2.2 sowie 2.4).

3.4 Ablauf des Algorithmus` nach Svoboda und Matula

1. Einlesen der Bilddaten, des minimalen und maximalen

   Durchmessers der Maschen (d_{in} , d_{out}), des Glattheitsfaktors α und des Dämpfungsfaktors γ .
2. Berechnung des Gradientenbildes $|\nabla I|$.
3. Erstellung einer Faltungsmaske K in der Form eines Kreises mit dem Radius d_{in} .

4. $buf \leftarrow |\nabla I| * K$ Faltung des Gradientenbildes mit K
5. $buf \leftarrow I \setminus buf$ Pixelweise Subtraktion mit den Bilddaten I
6. $buf \leftarrow buf * K$ Faltung des Puffers buf mit K.
7. $markers \leftarrow NonMaxSuppression(buf)$ Verwerfen der überflüssigen Markierungen.
8. Für alle gefundenen Markierungen wird je eine *Dual Simplex Mesh* erstellt.
9. Für jede *Dual Simplex Mesh* führe eine Verformung durch.
10. Die verformten Maschen sind das Ergebnis der Segmentierung und stellen die einzelnen Zellen dar.

Man beachte, dass der Algorithmus hier nur für den zweidimensionalen Fall dargestellt ist. Dies heißt aber nicht, dass der Algorithmus in 3D auf eine andere Art und Weise arbeitet. In Wirklichkeit funktioniert er ganz genau analog zum 2D-Verfahren, man ersetze beim Lesen der Ablaufschritte lediglich Bild durch Volumen, Pixel durch Voxel sowie Kreis durch Kugel und erhält die Beschreibung für den dreidimensionalen Fall.

Svoboda und Matula führen an, dass es je nach Zustand der Bilddaten erforderlich sein kann, die Daten einer Vorbehandlung zu unterziehen. So müsste beispielsweise ein stark verrauschtes Bild im Vorfeld durch einen Tiefpass-Filter geglättet werden, um gute Resultate zu erreichen.

Außerdem wird eine Evaluation der Ergebnisse nahegelegt, denn die Unterschiede der Segmentierungsergebnisse in Abhängigkeit von den Eingabeparametern können sehr stark variieren.

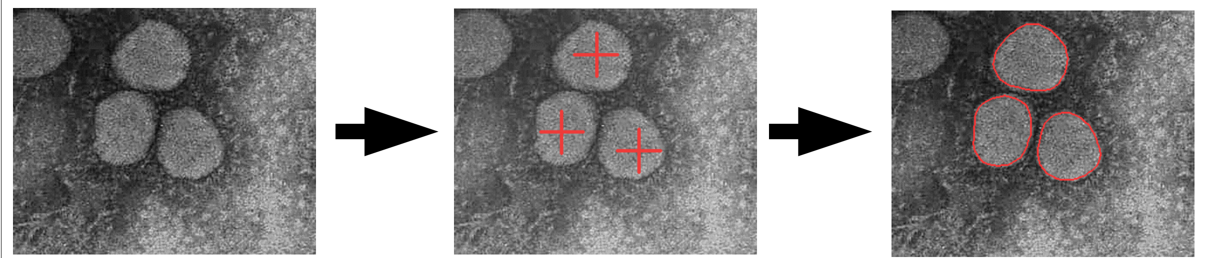


Abbildung 14: Beispielhafter Ablauf des Algorithmus` in 2D nach Svoboda und Matula. Von links nach rechts: 1. Ausgangsbild, 2. Gefundene Maschenzentren, 3. Endergebnis, nach der Verformung der Maschen

3.5 Einschränkungen der umgesetzten Implementationen

Im Zweidimensionalen wurde nicht das komplette Verfahren von Svoboda und Matula implementiert, sondern auf den Dämpfungsfaktor verzichtet, so dass sich die Maschenbewegung konstant vollzieht und nicht in ihrer Geschwindigkeit gedämpft wird. Ausserdem wurde eine Erweiterung vorgenommen, indem die Vorverarbeitung in das Verfahren integriert wurde (vgl. 4.2.5). Dieses betrifft allerdings lediglich einige Feinheiten, die keine großen Auswirkungen auf die Funktionalität des Verfahrens haben dürften. Wie später zu sehen sein wird, sind die gelieferten Ergebnisse nämlich durchaus brauchbar.

Im Dreidimensionalen finden sich größere Einschränkungen. Hier wurde zum Zeitpunkt dieser Arbeit das automatische Finden der Objektmarkierungen noch nicht implementiert.

Auch auf den Dämpfungsfaktor wurde verzichtet. Dazu kamen einige weitere Möglichkeiten die Maschen grafisch auszugeben (vgl.4.3.4)

Eine weitere Einschränkung des implementierten Verfahrens ist sowohl in 2D als auch in 3D, dass die zu segmentierenden Objekte heller als die Umgebung sein müssen. Dies hängt damit zusammen, dass das Gradientenvektorfeld ∇I und dessen Richtungsvektoren beachtet werden müssen. Im Allgemeinen hat diese Tatsache aber keinerlei Einschränkungen auf die Leistungsfähigkeit des Verfahrens, eventuell muss das Bild vor der Segmentierung invertiert werden.

4 Implementation des Algorithmus`

Im Folgenden wird auf die konkrete Implementation des Algorithmus` in C++ eingegangen. Dazu werden zunächst alle verwendeten Hilfsmittel vorgestellt und anschließend die Details und Vorgehensweise des Algorithmus` (siehe 3.4) in der Implementierung erläutert.

4.1 Beschreibung der Arbeitsumgebung

Bevor damit begonnen wird, die eigentliche Implementation zu beschreiben, wird zunächst auf die verwendete Arbeitsumgebung eingegangen. Hierzu zählen die verwendete Hardware ebenso wie die verwendete Software.

Beides zu beschreiben stellt sich als sehr nützlich heraus, da neben der Implementation des Algorithmus` auch Leistungsmessungen durchgeführt wurden. Diese ohne das dazugehörige Entwicklungssystem, bestehend aus Hardware- und Softwarekomponenten, zu vergleichen, erscheint hingegen unmöglich.

4.1.1 Verwendete Hardware

Wie schon früher erwähnt ist es zur Abschätzung, zur Einstufung und zum Vergleich der Leistungsdaten unbedingt notwendig, dass die Hardwarekomponenten des Systems vorgestellt werden, auf dem die Testläufe stattgefunden haben.

Besonders wichtig für die Implementation ist die Speicherallokation und der Speicherzugriff. Der zweite wichtige Punkt ist die CPU und dabei insbesondere die FPU, jene Einheit, welche Gleitkommaoperationen durchführt, da viele verschiedene double-genaue Berechnungen durchgeführt werden müssen. Ebenfalls noch Erwähnung finden muss hier die genaue Version des Betriebssystems und des Compilers. Es folgen die Kenndaten der wichtigsten Hardwarekomponenten des Testsystems:

- Mainboard: *ASUS© A7N8X deluxe*

  Dieses Mainboard besitzt einen nVidia© nForce©2 Chipsatz und bietet mit 333MHz Frontside-Bus die Grundlage des Testsystems. Während aller Tests wurde im BIOS das Speichertiming auf „Aggressive“ gesetzt. Dies ist eine Einstellung für Hochgeschwindigkeits-Speicherbausteine, doch dazu mehr im nächsten Punkt.

- Arbeitsspeicher (RAM): *Corsair© 512MB DDR-333 CL2 6ns*

  Die Abkürzung DDR steht Double Data Rate und bedeutet, dass dieser Speicherbausteinen pro Taktzyklus die doppelte Datenrate eines herkömmlichen SD-RAM leistet. Er arbeitet mit 333MHz und besitzt eine Zugriffszeit von 6ns.

  Die Abkürzung CL steht für „CAS-Latency“, was wiederum eine Abkürzung für „Column Address Strobe Latency“ ist. Ein Column Address Strobe ist ein Zugriffssignal für einen Speicherbaustein. Und CL2 gibt an dass der Speicherbaustein 2 Takte benötigt, die während des CAS gelieferten Daten zu verarbeiten, bevor er weitere Befehle entgegennehmen, bzw. das Ergebnis mitteilen kann.

- Prozessor (CPU): *AMD© Athlon© XP 2600+*

  Dieser Prozessor arbeitet intern „nur" mit 2083MHz, erreicht aber laut Hersteller vergleichbare Leistungen wie ein Pentium 4 mit 2,6 GHz. Desweiteren besitzt er 128kB First-Level Cache sowie 256kB Second-Level-Cache, beide laufen im Modus Pipeline-Burst-Synchronous.

4.1.2 Verwendete Software

Da eine Vielzahl von Hilfsmitteln zur Implementierung verwendet wurde, sollen hier nun alle der Implementierung dienlichen Softwarekomponenten aufgelistet werden. Zu einigen auch für die Leistungsmessungen sehr wichtigen Komponenten wie z.B. das Betriebssystem ist zudem noch die genau Versionsnummer angegeben. Desweiteren wird zu jeder Komponente erläutert, warum gerade sie für die spezifische Aufgabe ausgewählt wurde.

- Betriebssystem: *Microsoft© Windows© 2000 / XP©*

  Die Wahl des Betriebssystems erfordert schon einiges an Begründung, ist es doch nicht gerade als das stabilste System für Entwickler bekannt. Dennoch hatte es einige Vorteile, die andere Betriebssysteme nicht oder nur zu Teilen bieten konnten.

  Dazu gehört zum Einen die hohe Verfügbarkeit des Systemes, obwohl in der Universität durchaus auch andere und möglicherweise stabilere Systeme, wie zum Beispiel Sun© Solaris© oder Suse© Linux 8.4, zur Verfügung stehen. Ein großer Teil der Entwicklungsarbeit musste allerdings ebenfalls zu Hause geleistet werden, und dort stand nichts anderes als Windows zur Verfügung.

  Die Entscheidung einheitlich mit Windows zu arbeiten, bot noch einen weiteren Vorteil: Durch die MSDN©-AA der Universität wurde die komplette Entwicklungssoftware kostenlos zur freien wissenschaftlichen Verfügung für dieses Projekt gestellt.

  Die exakte Versionsnummer des Testsystems (Windows 2000 Service Pack 4) lautet: 5.00.2195.

- Entwicklungsumgebung: *Microsoft© Visual .net© Entwicklungsumgebung 2003*

  Diese Entwicklungsumgebung wurde zur Implementation vor allem deswegen benutzt, weil sie sehr gut strukturiert ist. Außerdem bietet sie Hilfefunktionen, die weit über einen Texteditor in Kooperation mit einem C++ Compiler hinausgehen.

  So bietet die Umgebung neben Syntax-Highlighting und Syntax-Vervollständigung auch einen mächtigen integrierten Debugger, eine sehr gute Integration mit dem Compiler, unterstützende Projekt-Management-Funktionen und eine Unterstützung des objektorientierten Programmierparadigmas. So bietet sie zum Beispiel Assistenten für die Erstellung neuer Klassen, sowie einen Klassenbrowser und eine sehr umfangreiche Hilfe. Zudem war sie wie schon oben erwähnt frei verfügbar.

- C++Compiler: *Microsoft© Visual C++ .net© 2003*

Bei der gewählten Entwicklungsumgebung ist es fast schon selbstredend, dass auch der mitgelieferten Compiler verwendet wurde. Dieser Compiler hat aber dennoch einige weitere Vorzüge, die hier nicht unerwähnt bleiben sollen.

Die Geschwindigkeit und Optimierungen die der Compiler vornehmen kann, sind sehr nahe an der Grenze dessen, was das gewählte Betriebssystem hergibt. So gibt es zwar auch andere Compiler, aber dessen Vorteile verblassen, wenn man sich vor Augen hält, wie gut der gewählte Compiler in die Entwicklungsumgebung eingebunden ist. Ein weiterer Vorteil ist die relativ problemlose Einbindung der benutzten zusätzlichen Bibliothek und die nahezu vorhandene C++ Standard Konformität.

Da dieser Compiler auch zur Optimierung der Implementation verwendet wurde, sei hier ebenfalls die Versionsnummer angeführt: 7.10.3077.

- Zusatzbibliothek: *VIGRA 1.2.0*

  Die meiste Arbeit wurde durch die Verwendung dieser Bibliothek erspart. Der Name steht für „Vision with Generic Algorithms“. Es ist eine Bildverarbeitungs-Bibliothek für C++, deren Hauptaugenmerk darauf gerichtet ist, dass sie leicht anzupassende Algorithmen und Datenstrukturen bereitstellt. Dabei wird die Template-Technik von C++ ausgenutzt, welche eben diese strukturierbare Anpassbarkeit ermöglicht und dabei keinen Geschwindigkeitsnachteil mit sich bringt.

  Die wichtigsten Funktionen der Bibliothek die für dieses Projekt genutzt wurden sind:

  - Template-basierte Bilddatenstrukturen für vielerlei Pixeltypen
  - Mehrdimensionale Arrays
  - Import und Export von vielen Dateiformaten, so z.B.: Windows© BMP, GIF, JPEG, PNG, etc.
  - Mehrdimensionale sowie separierbare Faltungen, Gauß´sche Filterfunktionen

  Es sei an dieser Stelle noch angemerkt, dass mittlerweile die Version 1.3.0 der Bibliothek im Internet verfügbar ist, diese sollte aber keine Beeinträchtigungen dieser Implementation mit sich bringen.

- Dokumentations-Werkzeug: *CppDoc 2.3*

  Zu einer guten Arbeit gehört auch eine gute Dokumentation. Da bereits Erfahrung mit Java© und dem dort eingebauten Dokumentationssystems JavaDoc© gesammelt wurde, fiel die Wahl auf das C++-Pendant CppDoc. Es erzeugt eine grafisch ansprechende HTML-Dokumentation und benötigt, wenn man schon mit JavaDoc© gearbeitet hat, nahezu keine Einarbeitungszeit. Zudem ist es als Freeware kostenlos erhältlich.

  Diese Vorteile waren besonders wichtig, da das Ziel war eine möglichst gute und umfassende Dokumentation zu erhalten, ohne viel Zeit investieren zu müssen.

- Verwendete Messsoftware:

  Für die Testläufe wurde sowohl ein Messinstrument für den Speicherbedarf als auch für die Geschwindigkeit benötigt.

  - Zeitmessungen: *C++ Standard-Bibliothek „time.h“*

    Diese Bibliothek ermöglicht es in einem C++ Programm Zeitmessungen durchzuführen. Die Genauigkeit liegt dabei im Millisekundenbereich, dass heißt es kann auf die Millisekunde genau gemessen werden, was auch getan wurde.

  - Speichermessungen: *Windows Task-Manager*

    Dieses Programm wurde benutzt, um den von der eigenen Implementation benötigten Speicher zu bestimmen.

4.2 Implementation in 2D

Einen Algorithmus – wie in Kapitel 3 beschrieben – direkt in 3D zu implementieren, stellt sich als recht unpraktikabel heraus. Denn ohne eine Einschätzung über das Verhalten des Algorithmus` zu haben, lassen sich Fehler in 3D nur sehr schwer finden und ein Beseitigen derer kann so schon fast unmöglich sein.

Deshalb lohnt es sich in jedem Fall den Algorithmus in seinem Verhalten zunächst nur im Zweidimensionalen zu implementieren und auch zu betrachten, um Probleme frühzeitig zu erkennen und zu beseitigen. Erst wenn dies geschehen ist kann der Übergang zu einer Implementation des Algorithmus im Dreidimensionalen erfolgen.

Deswegen wird zunächst die 2D-Variante des Algorithmus beschrieben. Dabei wird ein Überblick über die Klassenstrukturen der eigenen Implementation gegeben. Anschließend wird auf die Speicherstrukturen, die Berechnung der Energien sowie auf die Implementation der einzelnen Schritte des Algorithmus (siehe 3.3) eingegangen.

4.2.1 Erläuterung der Klassenhierarchie

Wie in Abbildung 15 erkennbar, gliedert sich die Klassenhierarchie in drei Klassen. Die Basisklasse ist die Klasse „`Smesh2D`“. In ihr enthalten sind alle Elemente die notwendig sind, um eine einfache Masche zu repräsentieren. Auf die Datenstruktur wird im Folgenden in Abschnitt 4.2.2 eingegangen werden.

Die Klasse „`DSMesh2D`“ repräsentiert im Gegensatz zur Klasse „`Smesh2D`“ eine komplette Dual Simplex Mesh, indem es zwei Exemplare der Klasse „`Smesh2D`“ als Exemplarvariablen enthält. Diese Exemplarvariablen, die die einzelnen Maschen repräsentieren, sind – entgegen dem objektorientierten Programmierparadigma – als öffentlich deklariert. In der Praxis hat sich dieser Verstoß jedoch als recht sinnvoll erwiesen, so dass auf eine Kapselung in diesem speziellen Fall verzichtet wurde.

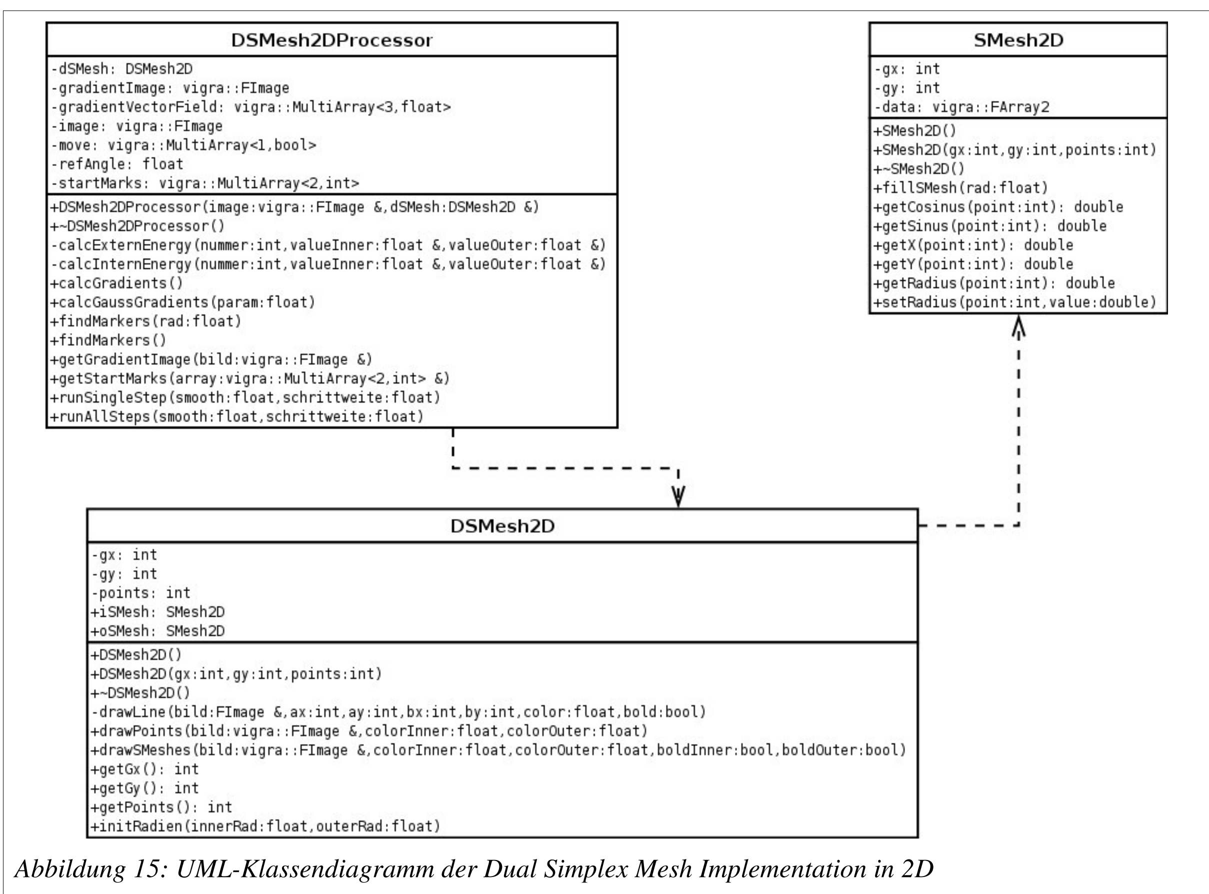


Abbildung 15: UML-Klassendiagramm der Dual Simplex Mesh Implementation in 2D

Die Klasse „DSMesh2DProcessor“ stellt den Prozess der eigentlichen Segmentierung dar. In ihr werden alle Daten berechnet, die zur Annäherung der Maschen notwendig sind und somit zur Segmentierung benötigt werden. Sie bedient sich dazu aller nötigen Eingabedaten und Eingabeparameter. So greift sie unter anderem auf die Klasse „DSMesh2D“ zu, und benutzt ein Exemplar dieses Types auch zur Annäherung.

4.2.2 Speicherrepräsentation der Maschen

Nachdem kurz ein Überblick über die Klassenhierarchie gegeben wurde, wird im Folgenden ein wenig genauer die Repräsentation der Maschen erläutert. Bei der Repräsentation wird – wie schon in (4.2.1) angedeutet das objektorientierte Programmierparadigma zu Grunde gelegt.

So wurden die Simplex Meshes als eine Klasse „Smesh2D“ modelliert. Die Exemplarvariablen beschreiben die Eigenschaften der Masche: So wird in „int gx“ und „int gy“ das Gravitationszentrum der Masche gespeichert. Dabei werden die absoluten Bildpixel-Koordinaten gespeichert. Hierbei wird der Datentyp „int“ benutzt, da davon ausgegangen werden kann, dass in der Praxis lediglich mit relativ kleinen Bilddaten gearbeitet wird. Relativ kleine Bilddaten bedeutet, dass für Höhe und Breite des Bildes gelten muss, dass sie $<2^{31}-1$ sein müssen, was in der Praxis eine sehr vernünftige Annahme ist.

Neben dem Gravitationszentrum besitzt eine Masche noch die zugehörigen Maschenpunkte (siehe 2.1), welche alle in einem bestimmten Radius vom Zentrum entfernt liegen. Zunächst soll die gewählte Verteilung der Punkte um das Gravitationszentrum beschrieben werden, bevor die Repräsentation derer im Speicher behandelt wird.

Es wurde eine gleichmäßige Verteilung der Punkte rund um das Gravitationszentrum angestrebt. Die gewählte Verteilung der Maschenpunkte geschieht wie folgt:

Gegeben:	Das Gravitationszentrum der Masche:	$\vec{g} = (gx, gy)$
	Der Radius der Masche:	rad
	Die Anzahl der Maschenpunkte:	n

Dann ergibt sich für die Maschenpunkte $p_i,\ i \in \{1, \ldots, n-1\}$ anfangs:

p_i besitzt den Einheitsvektor $\vec{e}_{p_i} = \left(\sin\left(\frac{i}{n} 2\pi\right), \cos\left(\frac{i}{n} 2\pi\right)\right)$.

p_i hat die absoluten Bildkoordinaten $\vec{c}_{p_i} = \vec{e}_{p_i} \cdot rad + \vec{g}$.

Eine Veranschaulichung der gewählten Verteilung findet sich in Abbildung 16.

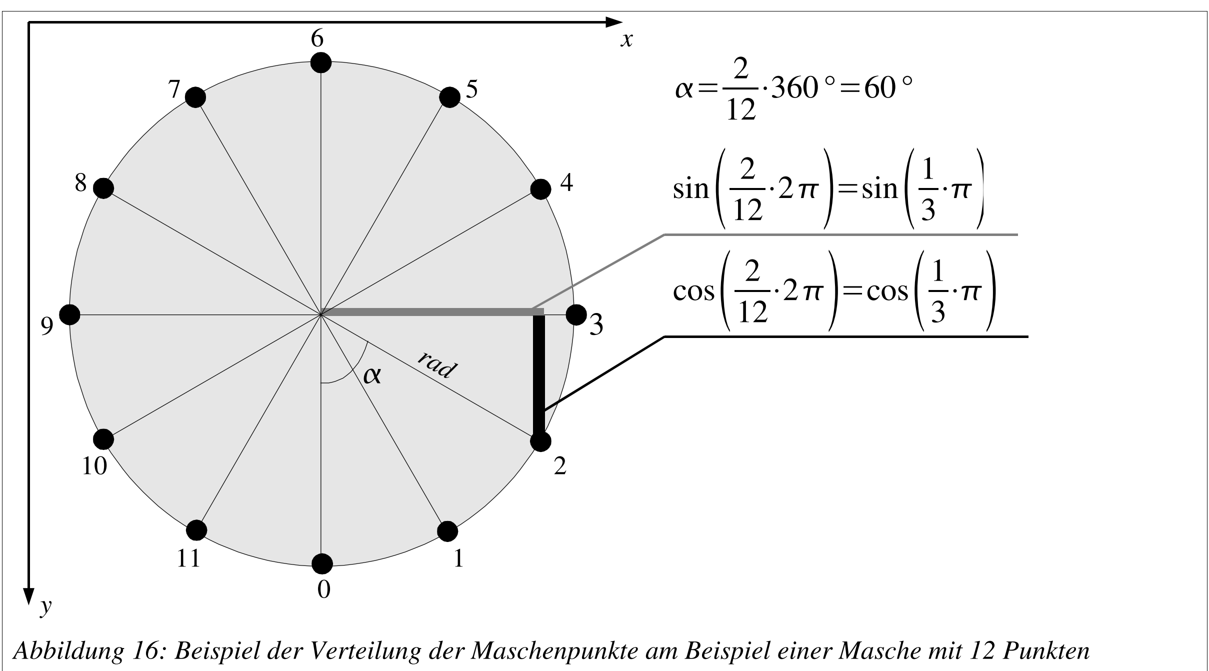


Abbildung 16: Beispiel der Verteilung der Maschenpunkte am Beispiel einer Masche mit 12 Punkten

Nach der Verteilung der Maschen kann jetzt bestimmt werden, welche Komponenten eines jeden Maschenpunktes in der Klasse „`Smesh2D`" gespeichert werden muss. Dazu werden zu jedem Punkt folgende Werte gespeichert:

- x- und y-Komponente des Einheitsvektors $\vec{e}_{p_i}$

- Radius rad_i
- Absolute x- und y-Koordinate von $\vec{c}_{p_i}$

Beim weiteren Verformungsprozess der Maschen bleiben die Einheitsvektoren der Maschenpunkte gleich. Wie schon früher erwähnt bezeichnen sie die Schiene, auf der der Maschenpunkt bewegt werden kann. Wird der Radius über die Methode „`setRadius`“ geändert, so werden automatisch auch die absoluten x- und y-Koordinaten aktualisiert.

Die absoluten x- und y-Koordinaten der Maschenpunkte lassen sich zwar auch einfach aus ihren Einheitsvektoren, Radien und dem Gravitationszentrum der Masche berechnen, da aber oft auf sie zugegriffen werden muss, macht es Sinn sie zusätzlich zwischenzuspeichern.

Als Speicherstruktur für die zu den Maschenpunkten gehörenden Eigenschaften fiel die Entscheidung auf ein 2-dimensionales Array „`data`“ vom Datentyp „`double`“, da für die Einheitsvektoren Gleitkommaarithmetik benötigt wird. Dieses 2-dimensionale Array kann man sich wie eine Tabelle vorstellen. Die Semantik der Zeilen und Spalten ist in Abbildung 17 kurz dargestellt, um die Speicherstruktur zu veranschaulichen.

	0	1	2	3	4
		Einheitsvektor		Absolute Koordinaten	
Punktnr.	Radius	x-Komponente	y-Komponente	x-Koordinate	y-Koordinate
0	data(0,0)	data(0,1)	data(0,2)	data(0,3)	data(0,4)
1	data(1,0)	.			.
2	data(2,0)		.		.
.	.			.	.
.	.			.	.
n	data(n,0)	.	.	.	data(n,4)

Abbildung 17: Semantik der Zeilen und Spalten des Arrays der Daten der Maschenpunkte

4.2.3 Implementationen der Energien

Für die Annäherung der beiden Maschen der Dual Simplex Mesh zueinander und somit dem zweiten Teil des Algorithmus, sind vor allem die an einem Maschenpunkt herrschenden Energien wichtig.

Die folgenden beiden Unterkapitel befassen sich genau mit den beiden verwendeten Energien, der internen Energie und der externen Energie, und sie beschreiben wie diese Energien jeweils implementiert wurden.

4.2.3.1 Interne Energie

Die Berechnung der internen Energie wird in der Methode „`calcInternEnergy`“ der Klasse „`DSMesh2DProcessor`“ durchgeführt. Die interne Energie wird immer von zwei Knoten gleichzeitig berechnet, den jeweils zusammengehörigen Knoten der inneren und äußeren Masche, da immer beide Werte benötigt werden um letztlich die Energien zu vergleichen. Zur Berechnung sind drei Übergabewerte erforderlich: „`int nummer, float & valueInner,`

`float & valueOuter`“. Die Werte „`valueInner`“ und „`valueOuter`“ werden als Referenz übergeben, sie dienen dazu die berechneten Werte der internen Energie des äußeren und inneren Maschenpunktes zu speichern. „`nummer`“ gibt die Nummer der Knoten an, für die die Energie berechnet werden soll.

Als erster Schritt wird die Konstante `pi` eingeführt, die notwendig ist, da C++ im Bogenmaß rechnet, aber Werte im Winkelmaß zur weiteren Verwendung benötigt werden. Die Umrechnung erfolgt später und ist relativ simpel, nämlich die Multiplikation mit $180/pi$.

Als nächstes werden die Koordinaten der zu betrachtenden Knoten, sowie derer direkter Nachbarknoten, jeweils in einem Array gespeichert (in „`inner`“ bzw. „`outer`“). Damit auch hier so wenig Rundungsfehler wie möglich auftreten, sind die Koordinaten vom Typ „`double`“. Da die *n* Knoten einer Masche kreisförmig angeordnet sind (siehe Abbildung 16), muss man beim Aufaddieren bzw. beim Subtrahieren von Eins auf die aktuelle Knotennummer aufpassen, dass man nicht eine ungültige Nummer bekommt. Dies kann vor allem beim Übergang von *n-1* nach *0* passieren. Deswegen wird eine *modulo* Rechnung für die Berechnung der Nachbarn eines Knotens verwendet.

Im nächsten Schritt werden die folgenden Winkel bestimmt:

- „`innenAlpha`“, der Schnittwinkel zwischen dem Vektor vom ersten Nachbarn zum aktuellen Knoten und dem Richtungsvektor des aktuellen Knotens der inneren Masche.
- „`innenBeta`“, der Schnittwinkel zwischen dem Vektor vom zweiten Nachbarn zum aktuellen Knoten und dem Richtungsvektor des aktuellen Knotens der inneren Masche.
- „`aussenAlpha`“ und „`aussenBeta`“ werden analog berechnet, entsprechend mit den Knoten der äußeren Masche.

Zuletzt müssen noch die internen Energien der beiden Knoten berechnet werden, auch dieses ist in (2.2) beschrieben, und „`valueInner`“ und „`valueOuter`“ zugewiesen werden. Dabei ist der Simplex Winkel jeweils der Alpha- plus der Beta-Winkel. Weichen die internen Energien um einen bestimmten Wert (kleiner als 0.00001) von Null ab, so werden sie auf Null gesetzt. Diese letzte Modifikation soll verhindern, dass sich Maschen durch Ungenauigkeiten bei der Winkelberechnung ungleichmäßig verformen.

4.2.3.2 Externe Energie

Die Implementation der Berechnung der externen Energie findet natürlich in der gleichen Klasse wie die der internen Energie statt. Wie auch bei der internen Energie werden immer die externen Energien zweier korrespondierender Knoten gleichzeitig berechnet.

Zur Berechnung sind auch hier die gleichen drei Übergabewerte, welche auch die gleiche Semantik besitzen, erforderlich, wie bei der Berechnung der internen Energie: „`int nummer, float & valueInner, float & valueOuter`“. Der Name der Methode ist in diesem Fall allerdings „`calcExternEnergy`“.

Die Methode wird wegen ihrer Kompaktheit im Folgenden recht knapp beschrieben: Zunächst werden die absoluten x- und y-Koordinaten des Punktes der inneren Masche ermittelt. Dann wird für diese Koordinaten geprüft, ob der Einheitsvektor dieses Punktes in die gleiche

Halbebene zeigt wie der Vektor des Gradientenvektorfeldes an den gleichen Koordinaten. Ist dies der Fall, so könnte man den Sachverhalt als ein Ziehen des Maschenpunktes mit Hilfe des Gradientenvektorfeldes bezeichnen. Und genau dann wird die externe Energie auf den Maximalwert gesetzt.

Falls die Bedingung nicht erfüllt wird, so wird die Energie auf den Wert $1-|\nabla I|$ gesetzt. Dies entspricht genau der vorgegebenen Definition in (2.3).

Im Anschluss erfolgt die gleiche Berechnung ebenfalls für den Maschenpunkt der äußeren Masche. Die berechneten Energien werden im Anschluss in die dafür vorgesehenen und als Referenz übergebenen Variablen „`float & valueInner, float & valueOuter`" geschrieben.

4.2.4 Automatisches Finden der Objektmarkierungen

Das Finden der Objektmarkierungen ist erst relativ spät zu der Implementation hinzugekommen, da zuerst noch einige Verständnisprobleme mit dem von Svoboda und Matula vorgeschlagenen Algorithmus [SM03a] vorlagen. In (3.3.1) findet sich die Beschreibung dieses Algorithmus, dessen Implementierung in diesem Abschnitt kommentiert und beschrieben werden soll. Bildhaft wird der Ablauf zusätzlich in Abbildung 19 beschrieben.

Als günstigster Ort im Programm erschien die Klasse „`DSMesh2DProcessor`", da hier auch schon die gesamte Funktionalität bezüglich des Umgangs mit den Maschen implementiert wurde. So befindet sich beispielsweise die Methode zur Berechnung des Gradienten in dieser Klasse, und das Gradientenbild ist eine Voraussetzung für die Suche nach den Objektmarkierungen. Trotzdem ist diese Unterbringung nicht ganz glücklich, da im Gebrauch des Algorithmus zum Finden der Objektmarkierungen erst eine beliebige Masche erstellt werden muss, um dann die öffentliche Methode zur Suche aus der main-Methode aufzurufen.

Um die Suche zu starten gibt es zwei mögliche Aufrufe. So kann man den Aufruf mit Angabe eines Parameters stellen, oder auch ohne (entweder „`findMarkers(float rad)`" oder „`findMarkers()`"). Der Parameter entspricht dem Radius der Faltungsmaske, der in der Methode erstellt wird und sollte im Regelfall dem kleineren der beiden vom Benutzer anzugebenden Radien entsprechen. Dazu ist der Funktionsaufruf ohne Parameter gedacht. Nach deren Aufruf wird dann „`findMarkers`" mit Parameter aufgerufen, wobei der Parameter dann der minimale Radius ist.

Um nun die Suche nach den Objektzentren zu beginnen, welche die Objektmarkierungen darstellen, muss zunächst eine kreisrunde Faltungsmaske erstellt werden. Der Durchmesser dieses Kreises ergibt sich durch den übergebenen Parameter, der mit zwei multipliziert wird. Da ein ungerader Durchmesser besser für die Faltung im Bildraum geeignet ist, wird nach dieser Multiplikation noch Eins aufaddiert. Um sich die so erstellte Maske auch anschauen zu können, kann diese als Bildstruktur gespeichert werden, zur Bildrepräsentation wird hier ein Objekt der Klasse „`FImage`" aus der *VIGRA*-Bibliothek benutzt.

Nachdem sicher gegangen wurde, dass auch alle Werte in diesem Bild Null sind, wird angefangen den Kreis in die Mitte dieses Bildes zu setzen. Dank Pythagoras gelingt dieses auch recht einfach. Dabei werden alle Pixel des Bildes, die innerhalb des Kreises liegen, auf den Wert Eins gesetzt.

Zum Zwischenspeichern einiger Daten werden noch zwei Bildpuffer angelegt, die wieder Exemplare der Klasse „`FImage`“ sind. Der erste („`buffer`“) erhält den Inhalt des zu verarbeitenden Bildes. Das Umkopieren der Daten hat den Vorteil, das durch alle weiteren Schritte die original Bilddaten nicht verloren gehen. Ein weiterer Puffer („`buffer2`“) wird später noch gebraucht.

Die Variable „`buf`“ dient der Berechnung eines neuen Pixelwertes eines Puffers und wird schon in der darauf folgenden Faltung verwendet. Die Faltung findet im Bildraum statt, das hat zwar den Nachteil, dass es sehr langsam ist, vor allem für große Faltungsmasken, erleichtert aber erst einmal die Implementation. Später ist eine Faltung im Frequenzraum durchaus denkbar, ist aber in dieser Implementation dem Zeitmangel zum Opfer gefallen.

Die beiden Schleifen, mit denen das Bild durchlaufen wird, berücksichtigen nicht alle Pixel des Bildes, dies schadet jedoch nicht, da am Bildrand ohnehin keine Maschen gezeichnet werden können, da spätestens die äußere Masche außerhalb des Definitionsbereichs liegen würde.

An dieser Stelle wird nun zuerst die Faltung des Gradientenbildes mit der Faltungsmaske durchgeführt. Dabei werden alle Pixel unter der Maske aufaddiert nachdem sie mit der Maske an der jeweiligen Position multipliziert wurden. Das bedeutet, das ein Pixelwert entweder aufaddiert wird oder auch nicht, nämlich dann wenn die Maske den Wert Null hat, dies ist bei der verwendeten Maske außerhalb des Kreises der Fall.

Der berechnete Wert wird dann wieder normalisiert und zwar durch die Division mit den Ausmaßen bzw. der Fläche des Kernels. Letztlich bekommt der Puffer noch seinen neuen Wert zugewiesen.

Nun folgt der zweite Schritt der Suche, die pixelweise Subtraktion des Ausgangsbildes mit dem gerade erstellten Inhalt des Puffers. Dabei erhält jeder Pixel im Puffer den Wert der entsteht, wenn man den gerade berechneten Wert vom Ursprungswert abzieht. Sollte hierbei einmal ein Wert kleiner Null herauskommen, so wird dieser auf Null gesetzt.

Da die Randpixel bei der Faltung nicht beachtet wurden und jetzt stören, da sie falsche Werte besitzen, bekommen alle Pixel innerhalb des Randes ebenfalls den Wert Null zugewiesen.

Im nun folgenden Schritt wird „`buffer2`“ initialisiert, indem er den Inhalt des ersten Puffers bekommt, und somit auch seine Dimension, auf was es hier mehr ankommt. Dieser zweite Puffer ist notwendig, da bei einer Faltung unschöne Effekte auftreten, wenn Quell- und Zielbild identisch sind. Die Faltung erfolgt wie oben beschrieben, nur dass jetzt „`buffer`“ gefaltet wird und „`buffer2`“ als Zwischenspeicherort des Ergebnisses der Faltung dient. Direkt im Anschluss erhält dann der erste Puffer wieder die aktuellen Werte des zweiten zugewiesen.

Da durch die Faltung auch die Pixel im Randbereich in ihrer Intensität verändert worden sein können, werden diese in `buffer2` auf Null gesetzt, bevor dieser als erneuter Zwischenspeicher verwendet wird. Denn jetzt müssen die lokalen Maxima des Puffers gefunden werden, dass heißt die Pixel, die in ihrer Umgebung die hellsten sind, sollen herausgehoben werden. Dieses geschieht, indem der Puffer wieder fast vollständig (bis auf den Rand) durchlaufen wird und für jeden Pixel, der eine Intensität von größer Null aufweist, geprüft wird, ob es in seiner kreisförmigen Umgebung noch hellere gibt. Ist dies nicht der Fall, so wird der Wert des Pixels auf 255 gesetzt, ansonsten auf Null, denn er ist kein lokales Maximum.

Die gefundenen Maxima verbleiben in „`buffer2`“ , allerdings gibt es noch zu viele von ihnen, so dass von jeder Ansammlung vom Maxima noch das Zentrum gefunden werden muss. Zu diesem Zweck werden diese Ansammlungen, die als zusammenhängende „Flecken“ sichtbar werden, gelabelt, was bedeutet, dass jeder dieser „Flecken“ einen eigenen einzigartigen Grauwert bekommt. Gleichzeitig wird die Anzahl der Labels gezählt, dieser Wert wird in „`maxLabel`“ gespeichert. Mit diesem Wert ist es nun möglich eine Speicherstruktur für die Objektzentren zu erstellen.

Hierzu wird mit „`regionCenters`“ ein Exemplar der Klasse „`vigra::MultiArray<2,int>`“ erstellt, was einem dynamisch veränderbaren, zweidimensionalen Array des Datentypes „`int`“ entspricht. Jeder Eintrag in „`regionCenters`“ soll, nachdem es gefüllt wurde, den Koordinaten des Zentrums eines Labels entsprechen. Dazu werden für alle Labels jeweils alle enthaltenen Pixelkoordinaten aufaddiert und durch die absolute Anzahl der Pixel dividiert. So erhält man den Schwerpunkt eines jeden Labels.

Nun kann es immer noch vorkommen, dass zwei Schwerpunkte näher aneinander liegen, als der minimale Radius der Masche es zulässt. Da dieses Problem in der Praxis häufig dadurch entstand, dass die zu segmentierenden Zellen keine geschlossen „Kleckse“ sondern eher umrandete Gebilde darstellen. So werden binnen einer Zelle vermehrt zwei Objektzentren gefunden, die sich von der Entfernung her innerhalb des minimalen Maschenradius befinden (siehe Abbildung 18).

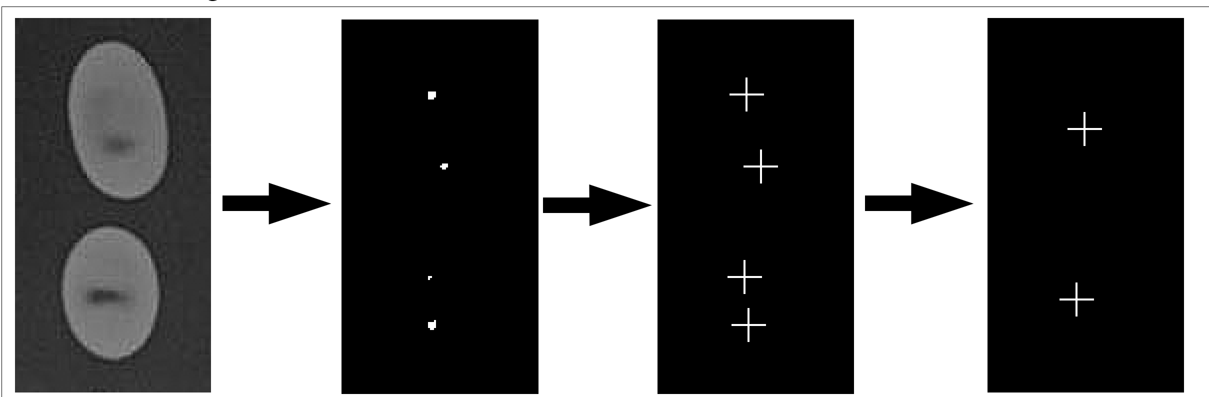


Abbildung 18: Von links nach rechts: 1. Ausschnitt eines Bildes mit zwei Blutzellen. 2. Gefundene Labels. 3. Aus den Labels ermittelte Zentren. 4. Ermittelte Objektzentren durch Verschmelzung der Schwerpunkte

Um diese Punkte wieder hinaus zu nehmen, werden alle Entfernungen zwischen je zwei Punkten in der nächsten Schleife berechnet. Nahe beieinander liegende Punkte werden aber nicht einfach nur gelöscht, sondern es wird der Mittelwert der Koordinaten aus beiden gebildet. Diese werden dem einen Punkt zugeordnet, der andere wird zum löschen vorbereitet, das heißt er bekommt negative Koordinaten, so dass er später leicht zu identifizieren ist. Alle übrig bleibenden Punkte werden danach in ein entsprechend kleineres „`MultiArray`“ gespeichert, dieses wird mit „`realRegionCenters`“ bezeichnet.

Mit dieser Verschmelzung ist das automatische Finden der Objektzentren im Prinzip abgeschlossen. Die durch Verschmelzung ermittelten Zentren, die nun in „`realRegionCenters`“ liegen, werden in der Exemplarvariable „`startMarks`“ der Klasse „`DSMesh2DProcessor`“ abgespeichert. Sie stehen so zur weiteren Verwendung als Gravitationszentren von Dual Simplex Meshes, die angenähert werden sollen, bereit.

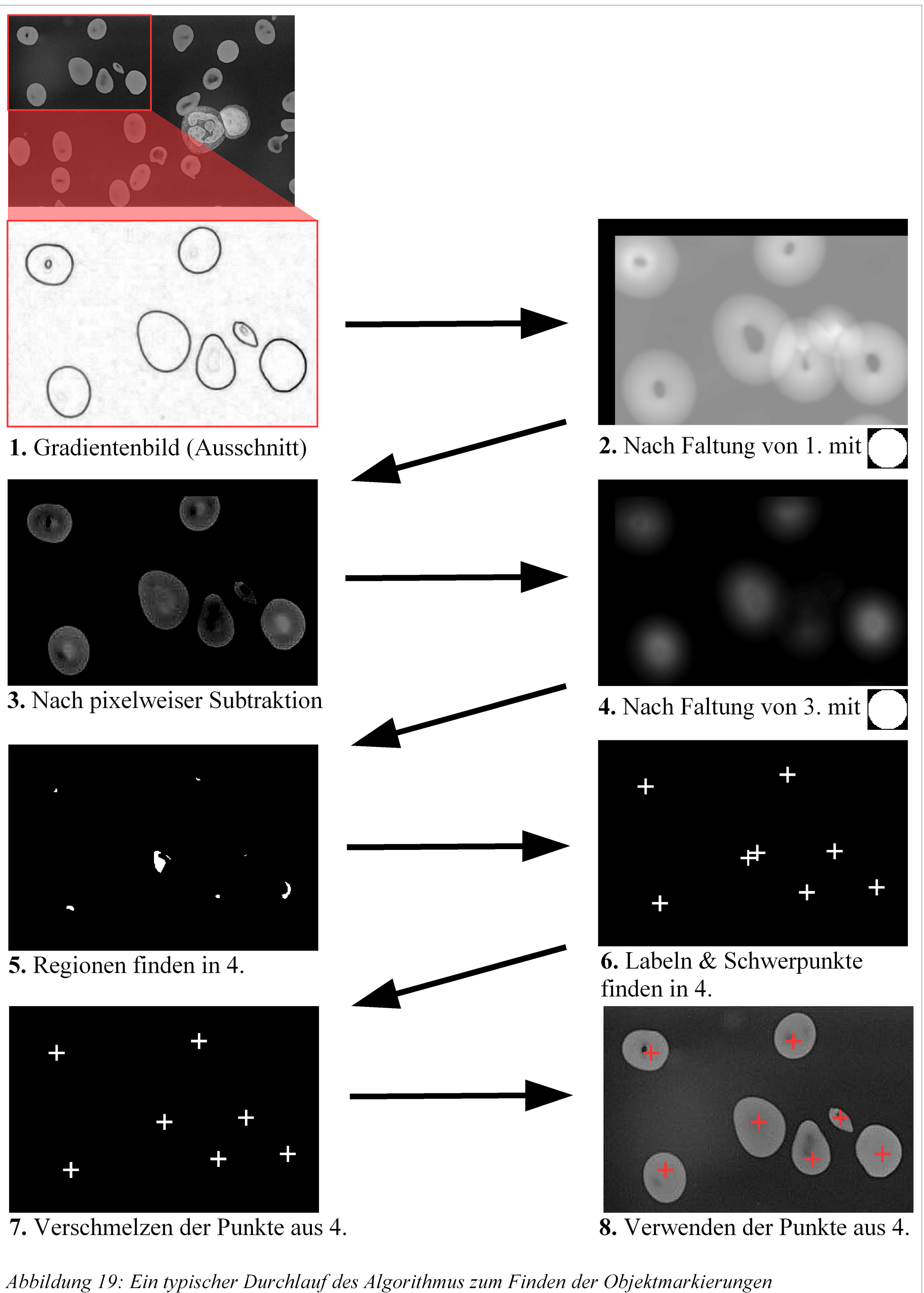


Abbildung 19: Ein typischer Durchlauf des Algorithmus zum Finden der Objektmarkierungen

4.2.5 Ablauf des Programms

In diesem Abschnitt wird nun ein typischer Ablauf des Programms dargestellt. Dabei wird auch erläutert, welche und in welcher Art Eingaben zu machen sind. Wie bereits erwähnt, wird an einigen Stellen (siehe 3.5) vom Algorithmus, wie er in [SM03a] beschrieben wird, abgewichen, so wird beispielsweise vollständig auf den Dämpfungsfaktor verzichtet. Dieser wird durch einen konstanten Annäherungsfaktor ersetzt.

Der Programmablauf beginnt mit der `main`-Methode der `main`-Klasse. Am Anfang befinden sich zuerst wichtige Typdefinitionen, die die Verarbeitung von Bildern sehr vereinfachen. Sie erlauben die Funktionalität der *VIGRA*-Bibliothek zu verwenden, in der schon viele Algorithmen zur Verfügung stehen. Deshalb werden die Bilddaten in einem für *VIGRA* benutzbarem Format gespeichert. Es gibt eine ausführliche Dokumentation zur *VIGRA*, hierzu [Köt04].

Danach folgt ein Array von Strings, die in der Reihenfolge folgende Bedeutung haben:

- Pfad (falls nicht der gleiche Ordner wie Die Datei „`main.cpp`“) und Name des zu bearbeitenden Bildes.
- Pfad und Name für ein Ausgabebild, das die Initialisierung zeigt.
- Pfad und Name für die Ausgabebilder, die die einzelnen Schritte der Verformung zeigen. Dazu ist weiter unten noch eine Erweiterung des Strings um eine Nummer vorgesehen. Diese Bilder dienen der Kontrolle bzw. Veranschaulichung des Ablaufs, und sind nicht unbedingt notwendig bzw. werden im Normalfall gar nicht benötigt.
- Dateiendung (Format) der Ausgabebilder.
- Pfad und Name für ein Ausgabebild, das das Endergebnis zeigt.
- Pfad und Name für ein Ausgabebild, das das Gradientenbild zeigt. (auch nur zu Kontrollzwecken gedacht).

Die dann folgenden Konstanten sind Werte, die vom Benutzer verändert werden können, um das Ergebnis zu beeinflussen. Hierbei ist „`smooth`“ der bereits in vorherigen Kapiteln erwähnte Glattheitsfaktor. „`gauss`“ gibt den Parameter für die Berechnung der externen Energie an, auch deren Funktion wurde schon in (4.2.3.1) erwähnt. Dann bleibt noch „`stepwidth`“ welche die Schrittweite der Maschenannäherung je Iterationsschritt angibt. Dieser Wert sollte bei Eins bleiben, um ein „übereinander weg laufen“ der inneren und äußeren Masche zu vermeiden.

Mit am wichtigsten sind die Radien, die jeweils für die inneren und äußeren Maschen angeben werden, sie werden mit „`innerRad`“ bzw. „`outerRad`“ bezeichnet. Standardwerte gibt es auch hier nicht, sondern sie müssen so exakt wie möglich für das vorliegende Bild bestimmt werden, diese Aufgabe obliegt natürlich, da es ein Eingabewert ist, dem Benutzer.

Die Intensitäten der Grauwerte der inneren und äußeren Masche kann man je nach Ausgangsbild einstellen, so das die Maschen auch dann gut sichtbar sind, wenn sie in dieses gezeichnet werden. Zu guter letzt gibt „`points`“ an, aus wie vielen Knoten eine Masche bestehen soll. Jede Masche ist zu Anfang ein gleichseitiges und symmetrisches Vieleck.

Bevor das Einlesen der Bilddaten beginnt, werden noch einige Vorkehrungen getroffen, um die benötigte Zeit des Verfahrens zu bestimmen. Da dies aber keinerlei Bedeutung für den Ablauf hat, sondern nur dazu dient etwas über die Performance zu sagen, wird hier nicht weiter darauf eingegangen (siehe hierzu Kapitel 5). Auch werden die späteren dazu gehörenden Aufrufe und Ausgaben nicht erläutert.

Jetzt erfolgt das Einlesen der Bilddaten. Mit Hilfe der *VIGRA*-Bibliothek geht dies recht einfach und man erhält ebenfalls die Dimensionen des Bildes. Von dem eben erstellten Bild werden zwei Kopien erstellt, eine um das Gradientenbild zu speichern und das andere zum Einzeichnen der Dual Simplex Mesh. Da diese später in jedem Schritt gezeichnet werden kann, braucht man das ursprüngliche Bild noch, um die Masche wieder zu „löschen“ und die aktuell verformte zeichnen zu können. Da anfangs die Objektzentren nicht automatisch gefunden wurden, sondern jeweils für eine Masche die Koordinaten von Hand eingegeben werden mussten, ist der weitere Ablauf vielleicht etwas umständlich geraten. Man möge dieses Verzeihen, da es generell schwierig ist, in ein schon funktionierendes Programm ein anfangs nicht vorgesehenes Feature einzubringen.

Zunächst wird eine beliebige Dual Simplex Mesh erzeugt, mit der einzigen Voraussetzung, dass sie innerhalb des Bildes liegt. Dieses geschieht durch die Erzeugung eines Objektes der Klasse „`DSMesh2D`“. Es werden dafür drei Parameter benötigt. Der erste ist die x-Koordinate, der zweite die y-Koordinate des Objektzentrums im Bild. Der dritte gibt schließlich noch die Anzahl der Knoten pro Masche an. Hat man die Dual Simplex Mesh erstellt, so werden noch die Ausgangsradien angegeben. Erst mit diesem erstellten Objekt ist es möglich ein Objekt der Klasse „`DSMesh2DProcessor`“ zu erstellen, um alle benötigten Berechnungen durchführen zu können. Dieser Prozessor kümmert sich zum Beispiel darum, dass die korrekte Anzahl an Schritten durchgeführt wird oder auch um die Berechnung des Gradientenbildes, was auch gleich getan wird, und anschließend zur Kontrolle als Bild gespeichert werden kann. Als Parameter zur Erstellung eines solchen Prozessors benötigt man sowohl das zu bearbeitende Bild, wie auch die eben gerade erstellte Dual Simplex Mesh.

Mit Hilfe dieses Prozessorobjektes ist es nun endlich möglich, die Objektzentren automatisch zu finden. Dazu ruft man dann die Methode „`findMarkers()`“ der Klasse „`DSMesh2DProcessor`“ auf. Der genaue Ablauf dieser Suche wurde in (4.2.4) bereits ausführlich beschrieben. Die gesamte Maschenverformung die jetzt folgt stammt noch von der Version, in der die Objektzentren nicht automatisch gefunden wurden, so war es damals z.B. auch üblich, pro Programmablauf immer nur ein Objekt zur Zeit zu segmentieren.

Entsprechend werden nun auch alle Dual Simplex Meshes nacheinander erstellt und verformt. Dazu wird erst einmal ein „`vigra::MultiArray<2,int>`“ erstellt. Es dient dazu alle gefundenen Koordinaten der Maschenzentren zu speichern, wobei an erster Stelle die x-Koordinate und an zweiter Stelle die y-Koordinate gespeichert wird. In der darauf folgenden Schleife werden alle Einträge im Array durchlaufen und dann daraufhin untersucht, ob die Koordinaten günstig im Bild liegen, dass heißt es muss noch möglich sein eine Masche einzuzeichnen, ohne das Bild zu verlassen (die Koordinaten dürfen entsprechend dem äußeren Maschenradius nicht zu nahe am Rand des Bildes liegen). Wurde diese Prüfung erfolgreich absolviert, so wird eine Masche mit diesen Werten und der konstanten, da vom Benutzer vorgegebenen, Knotenanzahl erstellt. Diese Masche bekommt natürlich auch, wie alle anderen, die gleichen Ausgangsradien und Intensitäten zur grafischen Darstellung. Anschließend ist es nur noch notwendig die Methode „`runAllSteps`“ des Prozessors

aufzurufen und die Maschenannäherung erfolgt automatisch, es werden nur noch die beiden Werte „`smooth`“ und „`stepwidth`“ benötigt.

Anschließend kann das Ergebnis als Bild ausgegeben werden, und muss dann vom Benutzer bewertet werden. Entspricht das erzielte Ergebnis nicht den Vorstellungen, so muss man das Programm noch einmal mit veränderten Parametern laufen lassen.

Der Ablauf der Annäherung oder Deformierung je einer Masche stellt sich wie folgt dar: Ausgelöst wird sie durch die Methode „`runAllSteps`“. Diese tut selbst nicht viel, in ihr läuft nur eine Schleife, die als Abbruchbedingung prüft, ob irgendein Paar korrespondierender Punkte schon zu nahe beieinander liegt, so dass eine Annäherung nicht mehr möglich ist. Solange dies nicht der Fall ist, wird die Methode „`runSingleStep`“ aufgerufen, an die auch die beiden Parameter weitergegeben werden. Erst hier findet die eigentliche Annäherung der Maschen statt.

Zunächst werden zwei eindimensionale Arrays der Länge drei und vom Typ „`float`“, um möglichst genau zu rechnen, erstellt. „`intEnergy`“ speichert an erster Stelle die interne Energie der inneren Masche, an zweiter Stelle die interne Energie der äußeren Masche und an dritter Stelle die gesamte Energie des Knotens der inneren Masche. Entsprechend speichert „`extEnergy`“ die externen Energien, sowie an letzter Stelle die gesamte Energie des Knotens der äußeren Masche. In einer Schleife werden alle Punkte der Masche einmal durchlaufen und die eben gerade erwähnten Werte berechnet. Zur Berechnung der internen und externen Energie siehe (2.2.1 und 2.3) sowie (4.2.3.1 und 4.2.3.2). Danach kann dann geprüft werden, welcher der Knoten die größere Energie besitzt und somit bewegt werden muss. Bei gleicher Energie wird der äußere Knoten in Richtung des inneren gezogen. Die Bewegung geschieht relativ einfach, indem einfach die Entfernung des Knotens (hier auch Radius genannt) zum Gravitationszentrum verändert wird. Wird der Knoten der äußeren Masche bewegt, so muss der Radius um die Länge `stepwidth` verkürzt werden:

„`(this->dSMesh.oSMesh.getRadius(i)-stepwidth)`“.

Für den Knoten der inneren Masche muss der Radius entsprechend verlängert werden:

“`(this->dSMesh.iSMesh.getRadius(i)+stepwidth)`”.

Wurden alle Knoten der Dual Simplex Mesh durchlaufen, so wird die Methode wieder verlassen. Alle Knoten sind sich nun um die Schrittweite „`stepwidth`“ nähergekommen. Aus der Methode „`runAllSteps`“ wird sie aber so häufig aufgerufen, bis eine weitere Annäherung der Knoten nicht mehr möglich ist. Die gesuchte Kontur sollte nun gefunden sein. In der Main-Methode findet nun noch die Ausgabe, also die Speicherung des Bildes statt, auf dem das Ergebnis der Verformung der Masche betrachtet werden kann.

Um eine Animation zu erhalten, in der sich die innere und äußere Masche aneinander annähern, kann man auch die Methode „`runAllSteps`“ umgehen, und aus der `main`-Methode direkt „`runSingleStep`“ aufrufen. Nun ist es möglich nach jedem Schritt ein Bild auszugeben. Diese schnell hintereinander präsentiert vermitteln den Eindruck, als bewegten sich die Maschen aufeinander zu und näherten dabei das gesuchte Objekt an, bis nur noch eine Kontur um dieses besteht.

4.2.6 Grafische Ausgabe

Da es zur Veranschaulichung der Ergebnisse nicht als ausreichend bezeichnet werden kann, nur die Punkte auf dem Bildschirm zu betrachten, wurde eine Routine zum Zeichnen der Maschen erstellt. Diese der grafischen Ausgabe dienliche Funktionalität wird im Folgenden kurz beschrieben und ihre Funktionsweise erläutert.

4.2.6.1 Zeichnen der Maschen

Bevor ein Bild mit den gewonnenen Ergebnissen abgespeichert werden kann, müssen zuerst die Daten der Maschen aus dem Objekt in ein Bild gezeichnet werden. Zu den gewünschten Ausgabeformen gehören zum einen die Knoten der Masche, die entsprechend ihrer Koordinaten in das Bild gezeichnet werden, so wie die Verbindungslinien zwischen je zwei benachbarten Knoten, so dass sich eine geschlossene Form ergibt.

Der Aufruf der Methode zur Ausgabe „`drawSMeshes`“ findet in der `main`-Methode statt. Über ein Exemplar einer Dual Simplex Mesh kann man diese Methode aufrufen, die sich in der Klasse „`DSMesh2D`“ befindet. Als Parameter notwendig sind zum einen ein zweidimensionales Bildarray, in das gezeichnet werden soll, sowie die Farbwerte (vom Typ „`float`“, also nur Graustufen) für die innere und die äußere Masche, so dass man diese gut unterscheiden kann. Ausserdem sollte bei dessen Auswahl Wert darauf gelegt werden, dass sie sich von den Grauwerten des Ursprungsbildes genügend deutlich abheben. Mit „`boldInner`“, bzw. „`boldOuter`“ kann angegeben werden ob die inneren bzw. äußeren Maschenpunkte mit Linien doppelter Stärke verbunden werden sollen Ein Aufruf könnte also folgendermaßen aussehen:

```
„dualesmaschenobjekt.drawSMeshes( bild, colorInner, colorOuter,
                                  boldInner, boldouter);“
```

Hierbei ist „`dualesmaschenobjekt`“ die Dual Simplex Mesh (genauer ein Objekt der Klasse „`DSMesh2D`“), die gezeichnet werden soll, und „`bild`“ eine Referenz auf das Bild, das später gespeichert wird, damit man dieses direkt verändern kann. Die vier weiteren Parameter stehen natürlich für die Farbwerte, sowie für die jeweiligen Linienstärken.

In der Methode „`drawSMeshes`“ werden zunächst Variablen zur temporären Speicherung der Knotenkoordinaten erstellt. Diese haben den Vorteil, dass der weitere Verlauf deutlich besser lesbar und übersichtlicher ist, und dient somit lediglich der Vermeidung von Fehlern. In der nachfolgenden Schleife werden alle Knoten der inneren Masche durchlaufen (zur Repräsentation dieser Knoten im Datentyp „`DSMesh2D`“ siehe 4.2.2). In jedem Schleifendurchlauf werden nun die Koordinaten des aktuellen Knotens und seines folgenden Nachbarknotens den temporären Variablen zugewiesen. Dabei ist zu beachten, dass der nächste Knoten nach dem letzten wieder der erste ist, was durch

```
„(i+1) % this->points“
```

sichergestellt ist. Ist dieses abgeschlossen, wird die Methode „`drawLine`“ aufgerufen. Für diese Methode muss man wiederum das Bild angeben, sowie die vorher gespeicherten Koordinaten der Knoten und die Farbe. Anschließend wird im gleichen Schleifendurchlauf noch mal das Ganze für die äußere Masche durchgeführt.

Die private Methode „drawLine“ erwartet die eben erwähnten Parameter, wobei das Bild wieder als Referenz übergeben wird. Zunächst werden zwei Variablen definiert, die dem temporären Speichern dienen. Bevor angefangen werden kann, die Linien zu zeichnen, muss überprüft werden, wie die Knoten zueinander liegen.

Da das Bild zum Zeichnen der Linien sowohl entlang der x- als auch in y-Achse abgetastet wird, ergeben sich folgende Spezialfälle:

- Gleiche x-Koordinate: Beim Abtasten entlang der x-Achse ensteht eine unendliche Steigung.
- Gleiche y-Koordinate: Beim Abtasten entlang der y-Achse entsteht eine unendliche Steigung.

Als erstes wird entlang der x-Achse abgetastet, d.h. es werden die x-Werte zwischen den Knoten entlang abgetastet und zu jedem ein passender y-Wert der Geraden berechnet. Es wird als erstes geprüft, ob die Knoten im Bild nebeneinander liegen, sprich die beiden Knoten eine unterschiedliche x-Koordinate haben. Ist dies der Fall, beginnt die Abtastung entlang der x-Achse. Dazu werden die Werte der Variablen „ax“ und „bx“, sowie „ay“ und „by“ genau dann vertauscht, wenn „ax“ > „bx“ erfüllt ist, dieses kann man recht leicht mit „swap“ realisieren. Anschließend wird die Steigung „my“, der zu zeichnenden Linie, aus dem Differenzenquotienten berechnet:

```
„dx = bx-ax;
 dy = by-ay;
 my = (float)dy / dx;“
```

Man beachte, dass eine Division durch Null bereits ausgeschlossen wurde! Nun geht man in einer Schleife von „ax“ nach „bx“, und ermittelt zu jedem x-Wert den entsprechenden y-Wert der Linie zwischen den beiden Knoten:

```
„tempy =(int)((tempx-ax)*my + ay + 0.5);“
```

Hierbei ist „tempy“ der benötigte y-Wert welcher diskret sein muss, da er eine Bildkoordinate darstellt, „tempx“ ist der aktuelle Zähler der Schleife. An jeder (x,y)-Stelle wird die Intensität des Pixels im Bild dann entsprechend geändert.

Ist der Parameter „bold“ gesetzt, so werden doppelt so starke Linien wie üblich gezeichnet. Damit dies gelingt, wird neben dem obigen Aufruf noch eine weitere temporäre y-Koordinate berechnet und das entsprechende Pixel gefüllt.

Da sich nach dieser Abtastung noch „Lücken“ in der Zeichnung der Linie befinden würden, erfolgt nun noch eine Abtastung entlang der y-Achse. Dazu werden die y-Werte zwischen den Knoten entlang abgetastet. Dieses erfolgt beinahe analog zur Abtastung entlang der x-Achse, allerdings ergibt sich hier für die Steigung:

```
„mx = (float)dx / dy;“
```

Außerdem müssen nun „ay“ und „by“ hinsichtlich ihrer Größe untersucht werden und vertauscht werden, falls „ay“ > „by“. Der zu jedem y-Wert zugehörige x-Wert ergibt sich diesmal als:

```
„tempx =(int)((tempy-ay)*mx + ax + 0.5);“
```

Die Variablen besitzen wieder jeweils die gleiche Semantik, und auch das Zeichnen von Linien doppelter Stärke stellt sich analog dar.

Durch diese Abtastung sowohl entlang der x- als auch der y-Achse wurde erreicht, dass keine störenden Lücken zwischen den Punkten entstehen. So erhält man durch das beschriebene Verfahren eine geschlossene Kontur für jede einzelne Masche, die je zwei Knoten einer Masche zeichnerisch miteinander verbindet.

4.2.6.2 Speichern der Bilder

Um ein Bild schließlich als formatiertes Bild abzuspeichern, so dass man es mit einem beliebigen Bildbetrachter anschauen kann, gibt es in der `main`-Klasse die Methode „`speichernBMP`“. Dazu übergibt man dieser Methode ein Bild vom Typ „`vigra::BasicImage`“ als Referenz, sowie einen Dateinamen. Die Methode ruft dann aus der *VIGRA*-Bibliothek die Methode „`exportImage`“ auf. Weitere Hinweise und Dokumentationen zu der *VIGRA*-Bibliothek gibt es in [Köt04].

4.3 Implementation in 3D

Nachdem der Algorithmus für den 2D Fall in nahezu aller Funktionalität implementiert wurde, folgt nun der Übergang zum Dreidimensionalen. Dies soll im folgenden ebenfalls sehr ausführlich beschrieben werden. Denn unter anderem treten hier ganz andere Problematiken als im zweidimensionalen Fall auf.

Dennoch lehnt sich die Implementation in vielen Teilen an die im Zweidimensionalen an. Vor allem die Programmstruktur wurde weitgehend erhalten. Dies macht diese Implementation leichter verständlich, und sorgt – obwohl die Komplexität des Verfahrens angestiegen ist – für eine gute Lesbarkeit.

4.3.1 Erläuterung der Klassenhierarchie

Wie aus der Abbildung 20 erkennbar, orientiert sich die Klassenhierarchie sehr stark an der Implementation in 2D. Die Basisklasse trägt diesmal den Namen „`Smesh3D`“. Sie enthält alle Elemente die notwendig sind, um eine einfache Masche in 3D zu repräsentieren. Desweiteren besitzt sie bereits die Fähigkeiten, sich auf vielerlei Weisen grafisch auszugeben (siehe 4.3.5). Die eigentliche Datenstruktur wird im Detail in den folgenden Abschnitten beschrieben.

Ganz analog zur Klasse „`DSMesh2D`“ repräsentiert die Klasse „`DSmesh3D`“ eine komplette Dual Simplex Mesh in 3D. Lediglich die Zeichenfunktionen erfahren eine veränderte Implementierung, denn sie werden hier – im Gegensatz zur 2D-Implementation – nicht mehr implementiert, sondern nur von den enthaltenen Simplex Meshes aufgerufen.

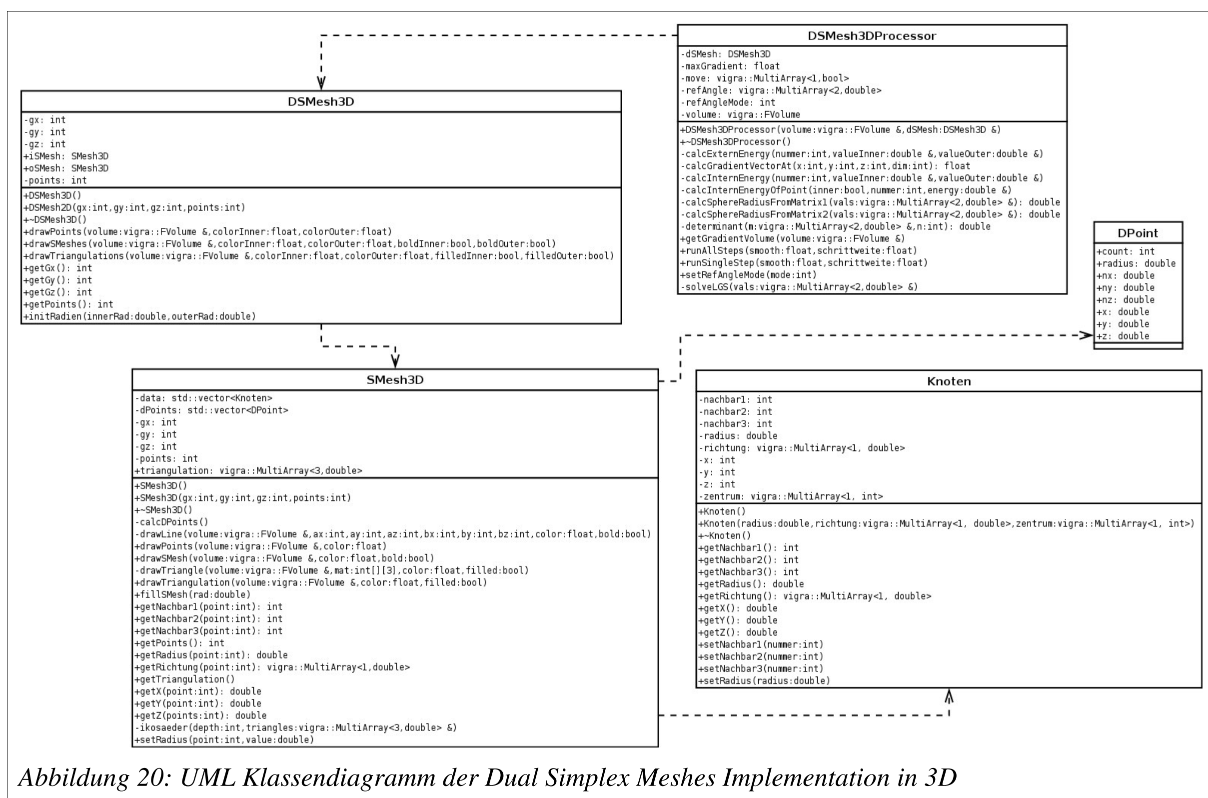


Abbildung 20: UML Klassendiagramm der Dual Simplex Meshes Implementation in 3D

Die Klasse „`DSMesh3DProcessor`“ stellt ebenfalls analog zur Klasse „`DSMesh2DProcessor`“ den Prozess der eigentlichen Segmentierung in 3D dar. Die einzigen Unterschiede, die sich hier zeigen sind, dass die Methoden zum Finden der Anfangsmarkierungen fehlen und einige Hilfsmethoden zum Berechnen der Simplex Winkel hinzugekommen sind.

4.3.2 Repräsentation der Maschen

Die Repräsentation der Maschen gestaltete sich in 2D noch recht kompakt beschreibbar (siehe hierzu auch 4.2.2), da sie im wesentlichen aus einer Beschreibung der Maschenpunkte als Eckpunkte eines regelmäßigen n -Ecks bestand. Jeder dieser Eckpunkte gewährleistete die an eine Simplex Mesh geforderte Bedingungen, und eine Winkelberechnung gestaltete sich auch als recht einfach zu realisieren, der Referenzwinkel war sogar konstant und äquivalent für alle Maschenpunkte.

Nun scheint das für den dreidimensionalen Fall nicht mehr so einfach möglich zu sein, und in der Tat fangen hier die Probleme schon damit an, wie die Punkte auf der Kugeloberfläche verteilt werden sollen. Aufbauend auf diese Frage zeigen die nun folgenden Unterkapitel den Weg auf, der am Ende zu einer korrekten Maschenstruktur – einer Simplex Mesh bzw. Dual Simplex Mesh – und deren Speicherrepräsentation führen wird.

4.3.2.1 Kugeltesselationen

Die Frage, wie die Punkte auf der Kugeloberfläche verteilt werden sollen führt unweigerlich zur Kugeltesselation. Zuerst jedoch soll dieser Begriff näher erläutert werden (vgl. hierzu [Wol04]):

Tesselation bedeutet Parkettierung einer Fläche in 2D oder eines Volumens in 3D mit regelmäßigen Polygonen in 2D bzw. Polyedern in 3D. Es gibt verschiedene Arten der Tesselation, so besteht eine reguläre *Tesselation* aus lauter gleichen Polygonen, bei einer *semi-regulären Tesselation* können sie verschieden sein, aber bei jeder Ecke stößt die gleiche Anzahl an Polygonen zusammen. Bei einer *demi-regulären Tesselation* hingegen muss diese Einschränkung nicht mehr gelten.

Es gibt nun verschiedene Möglichkeiten eine Kugeloberfläche in Polygone aufzuteilen. Eine Möglichkeit besteht darin, dass die Kugeloberfläche durch eine Anzahl von Längen- und Breitenkreisen in Vierecke aufgeteilt wird. Dies Verfahren kann man sich ähnlich vorstellen wie die Längen- und Breitenkreise der Erde. An den Schnittpunkten jeweils eines Längen- und eines Breitenkreises würde durch diese Tesselation ein Knoten entstehen.

Mit Ausnahme der Pole, besitzt dann jeder dieser Punkte eine 4er-Nachbarschaft. Hierbei sei bemerkt, dass die Pole offensichtlich zu Problemen bei der Nachbarschaftsbeziehung führen, da sie genau doppelt so viele Nachbarn besitzen wie es Längenkreise gibt. Hier besitzt eine Seite der Vierecke, in die die Oberfläche zerlegt wurde die Länge Null. Eine Möglichkeit dieses Problem leicht zu entschärfen, ist dass die Pole einfach weggelassen werden und jeder Polnachbar mit seinem korrespondierenden Knoten auf der anderen Seite des Pols verbunden wird. Dann allerdings liegt auch keine sehr gute Annäherung an eine Kugel mehr vor, und dies war doch gerade eine der Eigenschaften, die die Masche später erfüllen sollte.

Doch noch ein weiterer Nachteil ergibt sich, wenn man die Kugeloberfläche durch Vierecke tesseliert: Die Punkte werden recht ungleich auf der Kugel verteilt. Zwar kann dies durch eine geeignete Wahl an Längen- und Breitenkreisen leicht kompensiert werden, dennoch liegt es in der Natur dieser Aufteilung, dass mit zunehmender Nähe zu den Polen auch die Anzahl der Vierecke pro Fläche ansteigt.

Ein weiterer Nachteil ist eben jene 4er Nachbarschaft der Knoten. Um zu einer Simplex Mesh zu kommen müsste jedes entstandene Viereck noch durch eine Diagonale zerteilt werden, damit im Dreidimensionalen die Bedingung der 3er Nachbarschaft überhaupt erst erfüllt werden kann. Ohne diese lässt sich eine Berechnung des Simplex Winkels nicht durchführen.

Es sei aber an dieser Stelle auch noch ein Vorteil erwähnt: Diese Zerteilung der Kugeloberfläche ist relativ einfach – nämlich genau analog zum 2D Fall (siehe 4.2.2) zu implementieren und die Nachbarschaftsinformation muss nicht explizit gespeichert werden, da sie wieder aus der Knotennummer implizit entnommen werden kann.

Eine Tesselation welche den Forderungen schon adäquater genügt, ist die Tesselation der Kugel durch einen rekursiv verfeinerten Ikosaeder. Er bietet neben der Tatsache, dass er keine Pole besitzt einige weitere positive Merkmale wie z.B. eine relativ gleich verteilte Punktmenge über die Kugeloberfläche.

Dieser Ikosaeder sei wie folgt definiert:

Ein *n*-fach unterteilter Ikosaeder mit Radius rad ist

- ein Ikosaeder, falls $n=0$
- ein $(n-1)$ -fach unterteilter Ikosaeder bei dem jedes zur Oberfläche gehörende Dreieck $\Delta_i=\{\vec{a}_i, \vec{b}_i \vec{c}_i\}$ durch folgende 4 Dreiecke ersetzt wird:

$$\Delta_{i_1}=\left\{\vec{a}_i, \|\frac{\vec{a}_i+\vec{b}_i}{2}\|\cdot rad, \|\frac{\vec{a}_i+\vec{c}_i}{2}\|\cdot rad\right\}$$

$$\Delta_{i_2}=\left\{\vec{b}_i, \|\frac{\vec{b}_i+\vec{a}_i}{2}\|\cdot rad, \|\frac{\vec{b}_i+\vec{c}_i}{2}\|\cdot rad\right\}$$

$$\Delta_{i_3}=\left\{\vec{c}_i, \|\frac{\vec{c}_i+\vec{a}_i}{2}\|\cdot rad, \|\frac{\vec{c}_i+\vec{b}_i}{2}\|\cdot rad\right\}$$

$$\Delta_{i_4}=\left\{\|\frac{\vec{b}_i+\vec{c}_i}{2}\|\cdot rad, \|\frac{\vec{a}_i+\vec{b}_i}{2}\|\cdot rad, \|\frac{\vec{a}_i+\vec{c}_i}{2}\|\cdot rad\right\}$$

Dieser rekursiv unterteilte Ikosaeder (Abbildung 21) mag einige Vorteile bei der Tesselation der Oberfläche haben, dennoch erfüllt er eine Bedingung nicht: Er besitzt keine implizite Nachbarschaftsinformation, und jeder Knoten hat entweder fünf oder sechs Nachbarn, keinesfalls aber die geforderten drei Nachbarn. Wie aus dieser Triangulierung der Kugeloberfläche eine Simplex Mesh gewonnen werden kann, beschreibt das folgende Unterkapitel.

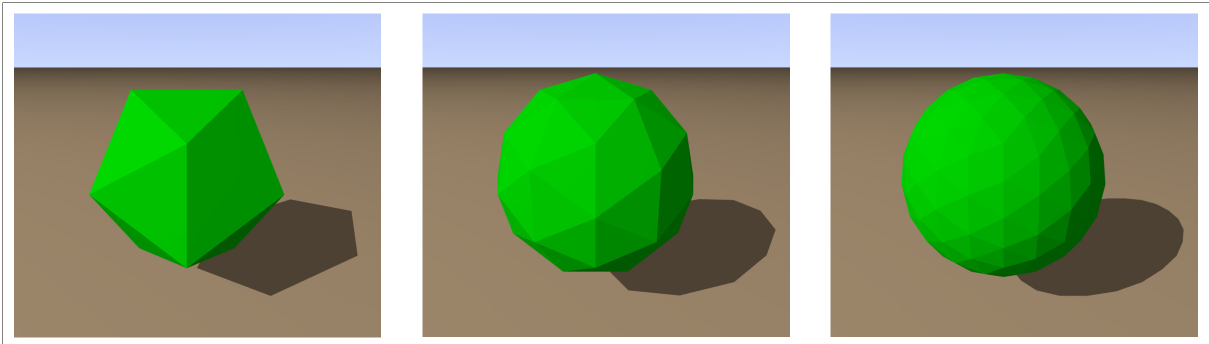


Abbildung 21: Die Unterteilung eines Ikosaeders, von links nach rechts: Ikosaeder, nach erster Unterteilung, nach zweiter Unterteilung

4.3.2.2 Duale Graph Transformation (DGT)

Die vorher beschriebene Tesselation der Kugeloberfläche lieferte also eine Triangulierung derselben. Aus dieser muss jetzt eine Masche gewonnen werden, und dies mit dem Kriterium der Dreiernachbarschaft eines jeden Knotenpunktes, und einer möglichst kugelförmigen Anordnung der Punkte um das Zentrum der Masche. Da die Kugeloberfläche angemessen tesseliert wird scheint die zweite Bedingung bereits erfüllt, die erste jedoch nicht.

An dieser Stelle kommt die Duale Graph Transformation (DGT) ins Spiel. Durch sie wird zu einer bereits bestehenden Triangulierung einer Kugeloberfläche ein dualer Graph erzeugt, der die Nachbarschaftskriterien erfüllt. Dabei wird von der Tatsache der Dualität von Graphen bzgl. ihrer Punkte- bzw. Kantenkonnektivität Gebrauch gemacht. So besitzt in der Triangulierung jeder Eckpunkt eines Dreiecks mehr als drei Nachbarn. Fasst man hingegen die Dreieckskanten als Kanten und die Dreiecke als Knoten eines Graphen auf, so besitzt – aufgrund der Tatsache, dass Dreiecke lediglich drei umschreibende Kanten besitzen – jeder

neue Knoten dieses Graphes genau drei Nachbarn. Bevor die genaue Funktionsweise dieser Transformation erläutert wird, soll sie zunächst auf ihre geometrischen Eigenschaften untersucht werden.

Als wichtigstes sei darauf hingewiesen, dass es sich bei dieser Transformation nicht um einen geometrischen Homeomorphismus handelt. Die Transformation erlaubt lediglich eine topologische Dualität, da es keine geometrischen Bijektionen zwischen Simplex Meshes und Triangulierungen gibt. Dieses läßt sich leicht nachweisen, wenn die Bild sowie die Urbildmenge der Transformation herangezogen wird (aus [Del94] und [Gum04]):

Sei V_{TR} die Menge der Knoten der Triangulierung, die diese charakterisiert, und

sei V_{SM} die Menge der Knoten der Simplex Mesh, die diese charakterisiert, und

sei DGT die Transformation die aus V_{TR} nach V_{SM} abbildet.

Annahme: DGT ist bijektiv

daraus folgt $|V_{TR}|=|V_{SM}|$

Es gilt aber mit der Euler-Gleichung:

$$
\begin{aligned}
& |V_{TR}|-\tfrac{3}{2}|V_{SM}|+|V_{SM}|=2 \\
\Leftrightarrow\quad & |V_{TR}|-\tfrac{|V_{SM}|}{2}=2
\end{aligned}
$$

Diese Ungleichheit der Kardinalitäten führt zum Widerspruch der Annahme!

Dieser Beweis zeigte, dass es unmöglich ist, einen geometrischen Homeomorphismus zwischen den beiden geometrischen Repräsentationen in 3D zu konstruieren. Hieraus folgt wiederum unmittelbar, dass die geometrische Deformierung der Maschen, wie sie im weiteren beschrieben wird ebenfalls nicht äquivalent einer Deformation ist, welche auf Ebene der Triangulierung arbeitet. Dies ist der Grund, warum Simplex Meshes im Gegensatz zu Triangulierungen auch Oberflächenrepräsentationen genannt werden.

Nach den geometrischen Eigenschaften wird nun die Transformation in ihren Einzelheiten erläutert werden, und in einem algorithmischen Ablauf beschrieben:

Gegeben: Die Dreiecksstruktur eines n-fach rekursiv unterteilten Ikosaeders bestehend aus $4^n \cdot 20$ Dreiecken.

1. Für jedes Dreieck Δ_i: Ermittle Schwerpunkt $\vec{s}_i$ und Nachbardreiecke N_{i1}, N_{i2}, N_{i3}
2. Erstelle neue Datenstruktur bestehend aus $4^n \cdot 20$ Knoten.
3. Für jeden neuen Knoten K_i, setze seine Koordinaten auf den Schwerpunkt $\vec{s}_i$ und setze die Nachbarn des Knotens auf N_{i1}, N_{i2}, N_{i3} .

Nach Ablauf des Algorithmus liegt eine Knotenstruktur vor, die eine Simplex Mesh in 3D repräsentiert. In Abbildung 22 ist dieses Ergebnis grafisch zu sehen.

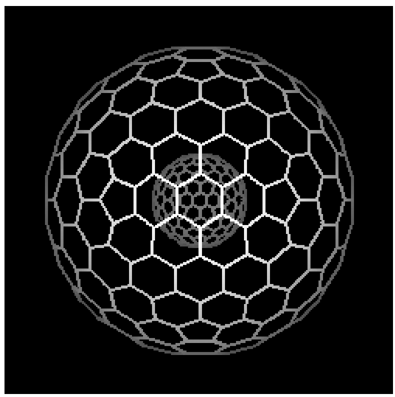


Abbildung 22: Eine durch DGT aus zwei rekursiv unterteilten Ikosaedern erzeugte Dual Simplex Mesh

4.3.2.3 Speicherrepräsentation der Maschen

Nachdem in den vorigen Kapiteln die recht abstrakte Grundlage zur Maschenerstellung gegeben wurde, soll nun die Erläuterung der eigentlichen Implementation der Maschen folgen. Dazu wird deren Repräsentation beschrieben.

Eine Simplex Mesh wird in der Klasse „`Smesh2D`" repräsentiert. Die Klasse besitzt ein Gravitationszentrum, welches in den Exemplarvariablen „`gx`", „`gy`" und „`gz`" gespeichert wird. Die Knotenmenge wird in einem „`vector`" vom Typ „`Knoten`" abgelegt. Da die Knoten dynamisch erzeugt werden, war diese Datenstruktur einem Array vorzuziehen.

Die Klasse `Knoten` wiederum repräsentiert den abstrakten Datentyp eines Simplex Mesh Knotens mit folgenden Exemplarvariablen:

- Zentrum des Knotens: $\vec{c}$
- Richtungsvektor vom Zentrum $\vec{c}$ aus: $\vec{r}$
- Radius des Knotens (Länge des Richtungsvektors): rad
- Absolute Koordinaten des Knotens: x, y, z, wobei $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \vec{c} + \vec{r} \cdot rad$

4.3.3 Implementationen der Energien

In diesem Kapitel werden analog zu Kapitel (4.2.3) die Energien, bzw. dessen Berechnungen in der Implementation erläutert. Dies erfolgt sowohl bei externer als auch bei interner Energie auf eine andere Art und Weise, als es bei der 2D Implementation der Fall war. Dies liegt daran, dass sich im Dreidimensionalen viele neue Randbedingungen ergeben, die es zu beachten gilt.

4.3.3.1 Interne Energie

Um die interne Energie zweier zusammengehöriger Knoten (Knoten der inneren sowie der äußeren Masche) zu berechnen wird die Methode „`calcInternEnergy`“ der Klasse „`DSMesh3DProcessor`“ aufgerufen. An die Funktion müssen drei Parameter übergeben werden: „`int nummer, double & valueInner, double & valueOuter`“. Die Parameter „`valueInner`“ und „`valueOuter`“ werden als Referenz übergeben, die neu berechneten Werte werden also direkt in den vorgesehenen Speicher geschrieben. „`nummer`“ gibt die Nummer der Knoten an, für die die Energie berechnet werden soll. Diese Funktion rechnet die Energien allerdings nicht selber aus. Dafür ist die Funktion „`calcInternEnergyOfPoint`“ verantwortlich. Nach der Berechnung wird nur noch geprüft, ob die berechneten Werte sehr nahe bei Null liegen. Ist dieses der Fall, so wird der Wert der Energie auf Null gesetzt, da kleine Abweichungen durch Rundungsfehler zustande kommen, welche vor allem dann auffallen, wenn die interne Energie die volle Gewichtung bekommt, d.h. die externe Energie nicht beachtet wird. Für diesen Fall ist dadurch eine gleichmäßige Verformung gesichert (siehe Absatz 2.4).

An die Funktion „`calcInternEnergyOfPoint`“ werden drei Parameter übergeben, zum einen „`bool inner`“, durch den man angeben kann, ob die Energie einer inneren oder äußeren Masche berechnet werden soll. Gilt „`inner=true`“, dann wird die Energie eines Knotens einer inneren Masche berechnet, ansonsten die eines Knotens der äußeren Masche. Weiter wird die Nummer des Knotens, dessen interne Energie berechnet werden soll, übergeben. Als letztes noch „`double & energy`“, welcher als Referenz übergeben wird. Dieser Parameter ist zur Übergabe des Ergebnisses, also der internen Energie des Knotens, erforderlich.

Als erstes wird in der Funktion die Masche, auf der gearbeitet wird, festgelegt, welches nach dem Wert des Parameters „`inner`“ geschieht. Die Konstante pi wird später häufig gebraucht und somit an dieser Stelle eingeführt. Die weiteren Variablen, die eingeführt werden, werden benötigt, um den Simplex Winkel zu berechnen (siehe 2.2.2).

Zunächst wird der Radius des Umkreises der Nachbarknoten berechnet. Um den Schnittwinkel zweier Vektoren zu berechnen, wird das Verfahren aus (2.2.1) benutzt. Die Länge eines Vektors wird auch nach bekannter Art berechnet. Anschließend wird die Länge durch zwei mal den Sinus des Schnittwinkels geteilt (Sinussatz anwenden), man erhält den Radius.

Anschließend wird der Normalenvektor der Fläche des Umkreises berechnet. Dafür wird zunächst das Array „`double n_i[3]`“ angelegt, in welches der Normalenvektor gespeichert wird. Zwei weitere Arrays des gleichen Typs dienen der Vereinfachung der weiteren Berechnungen. In sie werden jeweils die Richtungsvektoren von zwei Nachbarknoten zu einem der Nachbarknoten gespeichert. Diese Vektoren spannen die Ebene auf, dessen Normale berechnet werden soll. Dazu werden das Kreuzprodukt beider gebildet. Da dieser Vektor noch in eine beliebige Richtung von der Fläche weg zeigen kann, wird im folgenden noch das Skalarprodukt mit dem Richtungsvektor des gerade betrachteten Knotens gebildet. Ist dieses kleiner Null, so wird der Normalenvektor umgekehrt, d.h. mit „`-1`“ multipliziert. Letztendlich wird der nun korrekte Normalenvektor noch mit dem Vektor des ersten Nachbarknotens zum Knoten multipliziert und in der Variablen „`mult`“ gespeichert.

An dieser Stelle muss nun noch der Radius der Umkugel berechnet werden. Dazu werden zunächst die Koordinaten der vier beteiligten Knoten in das „`MultiArray spherePoints`“

gespeichert. Die Berechnung des Radius übernimmt dann die Funktion „`calcSphereRadiusFromMatrix1`“ unter Verwendung des Parameters „`spherePoints`“, also der Knotenkoordinaten. Dieses geschieht nach dem bereits in (2.2.2) beschriebenen Verfahren. Die Funktion wird in einem „`try-catch`“-Block aufgerufen, da sie eine Exception für den Fall wirft, dass die Knoten auf einer Ebene liegen. Der Radius wird in einem solchen Fall einfach auf einen recht großen Wert gesetzt.

Nun kann der Simplex Winkel berechnet werden, wobei das Vorzeichen des Arguments von der Variablen „`mult`“ abhängt, ist „`mult`“ negativ bekommt es ein negatives.

Nachdem der Simplex Winkel erfolgreich bestimmt wurde, kann nun nach (2.2) die interne Energie des Knotens berechnet werden. Dabei ist wiederum zu beachten, ob es sich um einen Knoten der inneren oder äußeren Masche handelt.

4.3.3.2 Externe Energie

Die Berechnung der externen Energie stellte sich für den zweidimensionalen Fall noch recht einfach dar, bestand sie hier doch größtenteils aus einem Zugriff auf das im Speicher gehaltene Gradientenbild sowie auf das Gradientenvektorfeld. Dies ist wegen einiger Gegebenheiten für den dreidimensionalen Fall nicht mehr praktikabel.

So ist bei einem Volumendatensatz der Auflösung 256 · 256 · 256 (Länge · Breite · Tiefe) und 16 Bit Wertebereich der Speicherbedarf mit 33,55 MB bereits recht hoch. Dennoch ist dies noch problemlos möglich im Speicher zu halten. Das Gradientenvektorfeld ist allerdings, da es jetzt vierdimensional ist, auf eine Größe von 100,66 MB angewachsen. Insgesamt ergibt sich für das Arbeitsvolumen, das Gradientenvektorfeld und das Gradientenvolumen ein Speicherbedarf von 167,77 MB. Auch dies bereitet aktuellen Systemen keine Probleme. Nun liefern moderne CT-Scanner aber schon hochauflösende Volumendaten mit einer Auflösung von 512 · 512 · 512 · 16 Bit. Wird hierfür ebenfalls ein Gradientenvolumen sowie ein Gradientenvektorfeld zusammen mit dem Arbeitsvolumen im Speicher gehalten, so ergibt sich ein Speicherbedarf von 1342,18 MB. Spätestens hier ergeben sich doch einige Probleme, auch auf aktuellen Systemen. Denn muss das Betriebssystem Arbeitsspeicher auf die Festplatte auslagern, verlangsamt sich der komplette Ablauf des Verfahrens extrem.

So wurde für die externe Energie in 3D eine Anpassung notwendig, die dies umgeht. Das Speicher raubende Gradientenvolumen bzw. Vektorfeld wird nicht mehr im Speicher abgelegt. Dies ist vor allem deswegen möglich, weil es im dreidimensionalen Verfahren außer zur Annäherung der Maschen nicht benötigt wird. Dies liegt daran, dass das automatische Finden der Zentren wie bereits erwähnt noch nicht implementiert wurde.

Die externe Energie wird nun immer noch aus Gradientenbild und Gradientenvektorfeld berechnet, allerdings werden die Werte bzw. Vektoren zur Laufzeit für jedes Voxel berechnet werden. Dies erledigt die Hilfsmethode:

```
„float DSMesh3DProcessor::calcGradientVectorAt(int x, int y, int z,
                                                int dim)“
```

Sie berechnet zu einem Voxel $\vec{v}=\begin{pmatrix}x\\y\\z\end{pmatrix}$ des Volumens das Gradientenvektorfeld wie folgt $\nabla_{\vec{v}}=\begin{pmatrix}calcGradientVectorAt(\vec{v},0)\\calcGradientVectorAt(\vec{v},1)\\calcGradientVectorAt(\vec{v},2)\end{pmatrix}$.

Dabei wird der Gradient aus der 6er Nachbarschaft (siehe Abbildung 23) des betreffenden Voxels errechnet. Aus diesem Gradientenvektorfeld lässt sich dann durch vektorielle Betragsbildung der Wert des entsprechenden Gradienten errechnen.

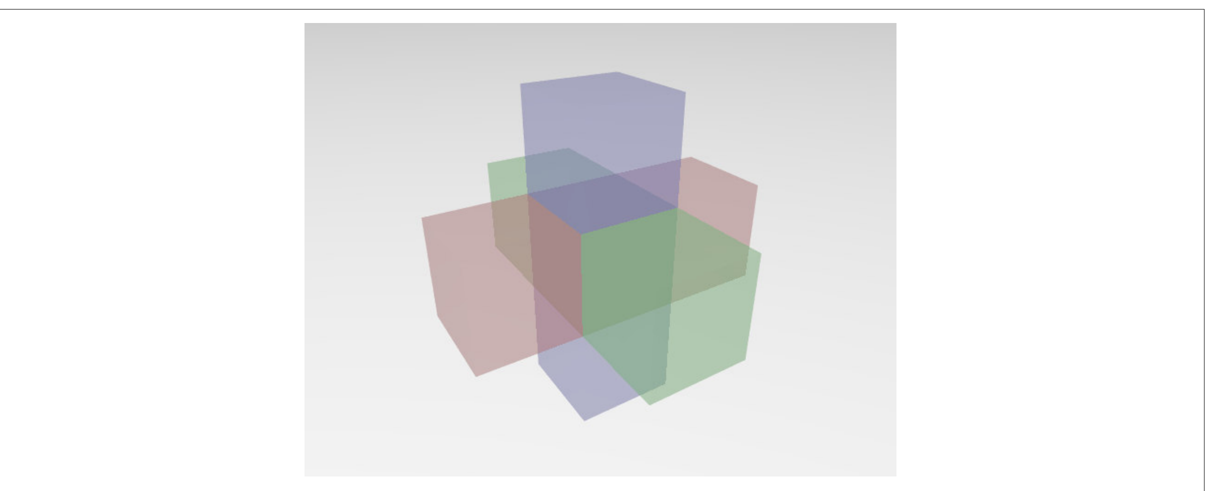


Abbildung 23: 6er Nachbarschaft eines Voxels. Rot: Nachbarn in x-Richtung. Violett: Nachbarn in y-Richtung. Grün: Nachbarn in z-Richtung.

Diese Methode verlangsamt das Verfahren nur sehr gering, dafür garantiert sie aber einen Speicherbedarf, der keine Probleme bereitet. Mit den erhaltenen Werten für das Gradientenvektorfeld wird nun die externe Energie genau analog zur zweidimensionalen Implementation berechnet (siehe hierzu auch 2.3 sowie 4.2.3.2).

4.3.4 Ablauf des Programmes

Wie schon für den 2D-Part wird auch für den dreidimensionalen Fall ein typischer Ablauf beschrieben. Dieses soll in diesem Abschnitt geschehen. Da es weiterhin die Aufgabe ist Bilder zu verarbeiten, wird wieder die *VIGRA*-Bibliothek verwendet. So befinden sich am Anfang der `main`-Methode auch wieder einige Typdefinitionen, die für den späteren Ablauf nötig sind. Doch wichtiger sind die Parameter, an denen etwas verändert werden muss, sobald man die Eingabedaten ändert. So werden als nächstes auch die Dimensionen des zu bearbeitenden Volumens angegeben. Da man später eine kugelförmige Masche in das Volumen legt, bewähren sich kubische Dimensionen. Allerdings ist es möglich „`width`“, „`height`“ und „`depth`“ einzeln anzugeben. Als nächstes kommt wohl der wichtigste Parameter, der sehr entscheidend für das Ergebnis ist. Je nachdem, welchen Wert man der Konstanten `smooth` zuweist, werden später die Energien gewichtet. Die Konstante `stepwidth` sollte so gelassen werden, wie sie eingestellt ist, da eine dynamische Annäherungsschrittweite

nicht weiter berücksichtigt wird. Die Konstanten „innerRad“ und „outerRad“ legen den Radius der inneren und äußeren Masche fest. Diese Werte müssen vom Benutzer auf das zu verarbeitende Volumen angepasst werden. Gerade in Bezug auf die spätere Angabe des Maschenzentrums sollte beachtet werden, dass die Maschen vollständig im Volumen liegen.

Die Anzahl der Maschenpunkte wird als nächstes angegeben. Der Benutzer kann diese wählen, allerdings ist zu beachten, dass die sich die Anzahl der Punkte immer durch die Gleichung $4^n \cdot 20$ berechnen, und die Eingabe x des Benutzers stets eine sein sollte, für die gilt:

$$x \in \mathbb{N} \wedge \exists i \in \mathbb{N}: x = 4^i \cdot 20 .$$

„colorInner“ und „colorOuter“ dienen der Farbangabe der inneren bzw. äußeren Masche. Die gewählte Farbe macht sich bei gewissen Ergebnisausgaben bemerkbar. In der nächsten Zeile wird der Name des einzulesenen Volumendatensatzes angegeben. Danach müssen auch noch die Ausmaße dessen angegeben werden. Diese können sich durchaus von den Dimensionen des Volumens, mit dem gearbeitet wird, unterscheiden. Die Daten müssen dann nur entsprechend in das Arbeitsvolumen hineingeschrieben werden.

Der nächste für den Benutzer bedeutende Schritt, ist die Erstellung der Dual Simplex Mesh. Dazu wird ein Objekt der Klasse „DSMesh3D“ erstellt. Bei der Erstellung werden vier Parameter übergeben, sie werden in folgender Reihenfolge angegeben:

1. x-Koordinate des Maschenzentrums
2. y-Koordinate des Maschenzentrums
3. z-Koordinate des Maschenzentrums
4. Anzahl der Maschenpunkte (siehe oben).

Nachdem eine Masche erstellt wurde, werden die Radien der inneren und äußeren Masche initialisiert, diese Werte wurden bereits oben Konstanten zugewiesen.

Interessant wird es erst wieder wenn der Prozessor für die Berechnung erstellt wird. Dazu wird ein Objekt der Klasse „DSMesh3DProcessor“ erstellt. Für den Konstruktor sind das Arbeitsvolumen und die Dual Simplex Mesh als Parameter zu übergeben. Anschließend kann die Art der Referenzwinkel eingestellt werden (siehe hierzu [SM01]), dieses geschieht über die Funktion „setRefAngleMode“.

Der Rest des Programms verläuft selbständig. Am Ende werden noch verschiedene Arten der Ausgabe der Ergebnisse vorgenommen, durch Abspeichern einzelner Bilder, die allerdings nicht immer alle durchlaufen werden müssen und vor allem zur Auswertung während der Entwicklung dienten.

4.3.5 Grafische Ausgabe

Bereits für die Visualisierung der Maschen im zweidimensionalen Fall wurden Zeichenprozeduren erstellt und der Implementation hinzugefügt. Diese Erweiterungen erwiesen sich als so sinnvoll bei der subjektiven Beurteilung der Ergebnisse, dass sie ebenfalls für den dreidimensionalen Fall erfolgen mussten.

Doch gibt es hier einige weitere Probleme, die im Folgenden beschrieben werden sollen und die weit über die Anpassung und Erweiterung der Zeichenprozeduren für den zweidimensionalen Fall hinausgehen. Sämtliche Zeichenprozeduren für die dreidimensionalen Maschen operieren auf Volumenbasis, also im dreidimensionalen Raum. Zur Darstellung der erzeugten Volumendaten dienen dann die am Ende dieses Kapitels beschriebenen Projektions- und Rotationsverfahren.

4.3.5.1 Zeichnen der Maschen

Die erste Methode befasst sich mit der Ausgabe einer Drahtgitterstruktur der Masche. Diese erlaubt ein erste Darstellung der Maschendaten und eine erste Evaluation. So wurde sie vor allem zur schnellen Darstellung sowie zur Verifizierung des Verfahrens eingesetzt.
Der Ablauf des Verfahrens stellt sich folgendermaßen dar:

Für innere und äußere Masche
  Für jeden Maschenpunkt M_i mit Nachbarn $N_1(M_i),\ N_2(M_i),\ N_3(M_i)$.
    Zeichne je eine Linie von M_i zu seinen Nachbarn.

Bei dem Zeichnen der Verbindungslinien zwischen zwei Punkten muss hierbei jedoch berücksichtigt werden, dass die zweidimensionale Zeichenprozedur nicht mehr zu gebrauchen ist. Sie wurde erweitert und enthielt ebenso die Möglichkeit, auch Linien doppelter Stärke zeichnen zu können. Diese lassen sich später besser erkennen.

Die Prozedur wurde – wie alle Zeichenprozeduren – in die Klasse Smesh3D verlagert, so dass jetzt auch schon eine einzelne Masche die Fähigkeit besitzt sich grafisch auszugeben. Das Zeichnen von Linien in einem dreidimensionalen euklidischen Raum funktioniert ganz genau analog zu dem Verfahren im Zweidimensionalen (siehe 4.2.6.1). Es muss lediglich ein dritter Differenzenquotient „`dz=bz-az`" eingeführt und beachtet werden.

Der Aufruf kann für verschiedene Anwendungen auf verschiedene Weisen erfolgen:

1. Eine Dual Simplex Mesh als Drahtgittermodell ausgeben:

   „`dualesmaschenobjekt.drawSMeshes(volume, colorInner, colorOuter boldOuter, boldInner);`"

   wobei „`volume`" das Volumen,

   „`colorInner`", „`colorOuter`" die Intensitätswerte der inneren bzw. äußeren Masche,

   „`boldInner`", „`boldOuter`" die Stärke der Linien (`TRUE` = doppelte Stärke) beschreiben.

2. Eine einzelne Masche als Drahtgittermodell ausgeben:

   „`maschenobjekt.drawSMesh(volume, color, bold);`"

Die erste Methode ist selbstverständlich lediglich ein doppelter Aufruf der zweiten. Ein duales Maschenobjekt muss ja, um selbst gezeichnet zu werden, lediglich seine innere und äußere Masche zeichnen.

Wie aus der Beschreibung ersichtlich, werden mit dem oben beschriebenen Algorithmus alle Verbindungslinien doppelt gezeichnet, da eine Nachbarschaftsbeziehung für jeden einzelnen Maschenpunkt beschrieben wird.

Beweis:

Sei M_i ein Maschenpunkt mit Nachbarn $N_{j\in\{1,2,3\}}(M_i)$,

und sei o.B.d.A M_k ein zweiter Maschenpunkt mit $M_k \neq M_i$ der selben Masche und es gelte: $\underset{j}{\exists} \; N_{j\in\{1,2,3\}}(M_i)=M_k$.

Dann folgt aber aufgrund der Symmetrie der Nachbarschaftsbeziehung sofort: $\underset{j}{\exists} \; N_{j\in\{1,2,3\}}(M_k)=M_i$.

Somit wird eine jede Linie doppelt gezeichnet!

Dennoch ist der zusätzliche Aufwand, der durch das doppelte Zeichnen entsteht vernachlässigbar, wenn man den Aufwand, der nötig wäre, auf bereits gezeichnete Linien zu prüfen ihm gegenüberstellt. So ist dies sicher nicht der eleganteste Weg, aber es ist dennoch ein sehr guter Kompromiss aus Einfachheit und Geschwindigkeit.

Was dieses Drahtgittermodell allerdings nicht zu leisten imstande ist, ist eine Aussage darüber zu erlauben, welche Voxel sich innerhalb und welche sich außerhalb der Segmentation befinden. Ein Ansatz, der es erlaubt dieses zu leisten, wird im Folgenden beschrieben.

4.3.5.2 Inverse Duale Graph Transformation (IDGT)

In 4.3.2.2 wurde bereits die Duale Graph Transformation beschrieben. Sie liefert ein Verfahren zum Erstellen einer Simplex und damit auch einer Dual Simplex Mesh. Die so erzeugte Dual Simplex Mesh ermöglicht im Anschluss korrekte und fundierte Energieberechnungen der Maschenpunkte (siehe 4.3.2.1 sowie 2.2.2) und damit auch die Deformation an die gesuchte Struktur.

Dennoch haben diese einen Nachteil. Wie im vorigen Abschnitt erkennbar, enthält eine Dual Simplex Mesh keinerlei Informationen darüber, ob ein Voxel innerhalb oder außerhalb des segmentierten Bereiches liegt. Eine einfache Interpolation über die Maschenpunkte, die das Drahtgittermodell der Masche beschreibt, schlägt ebenfalls fehl, denn die Maschenpunkte besitzen zwar je drei Nachbarn, bilden aber geometrisch gesehen Fünf- und Sechsecke. Dies hat seinen Ursprung an dem der Struktur zugrundeliegenden rekursiv unterteiltem Ikosaeder.

Da diese Fünf- und Sechsecke aus zueinander windschiefen Kanten bestehen können, ist trivialerweise nicht gegeben, dass sie auf einer Ebene liegen. Dies wäre lediglich bei Dreiecken gegeben, denn dessen drei Eckpunkte definieren eine Ebene in $\mathbb{R}^3$, wenn sie paarweise disjunkt sind.

Es liegt also nahe, die Dual Simplex Meshes in geeigneter Art und Weise zu triangulieren, und genau hier setzt die IDGT an. Gegeben eine Dual Simplex Mesh, erzeugt sie dessen Duale Graph Rücktransformierte – eine mögliche Triangulierung. Es sei darauf hingewiesen, dass sie lediglich eine mögliche Triangulierung erzeugt, und nicht die Triangulierung, durch den die Simplex Mesh unter Anwendung der DGT entstanden ist. Dies liegt in der Tatsache

begründet, dass die DGT keinen geometrischen Homeomorphismus darstellt (siehe 4.3.2.2). Gewissermaßen handelt es sich bei der IDGT also nur um eine Pseudo-Inverse Transformation. Gegeben diese Triangulierung müssen lediglich noch die Dreiecksflächen berechnet werden, um eine Hülle oder Oberfläche des segmentierten Objektes zu gewinnen.

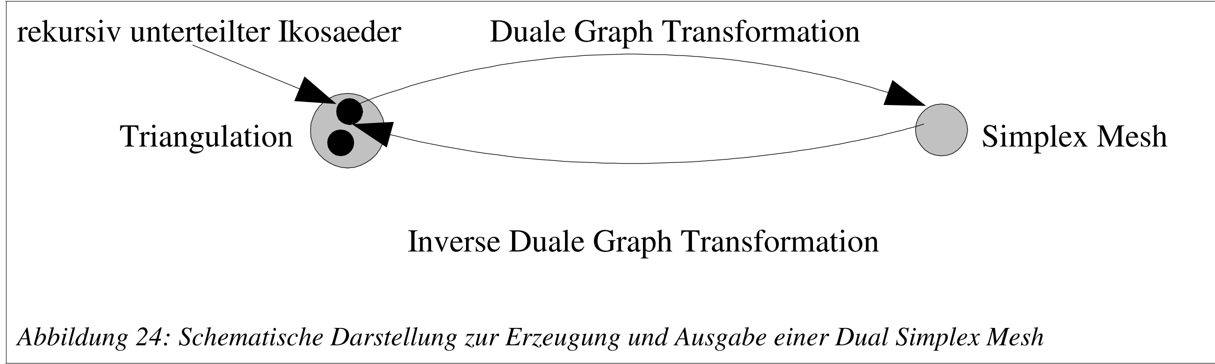


Abbildung 24: Schematische Darstellung zur Erzeugung und Ausgabe einer Dual Simplex Mesh

Im Folgenden wird der Ablauf der IDGT beschrieben:

Gegeben: Eine deformierte Dual Simplex Mesh bestehend aus n Maschenpunkten

Gesucht: Eine Triangulierung durch die IDGT

Algorithmus:

1. Erstelle einen rekursiv unterteilten Ikosaeder bzw. dessen Dreiecksstruktur, welcher genau aus n Dreiecken besteht.
2. Extrahiere aus den Dreiecksstrukturen die globalen Dreieckspunkte.

Da aus jedem Dreieck $\Delta_{i\in n}$ der Duale Maschenpunkt $M_{i\in n}$ entstand, folgen nun weitere Schritte:

3. Speichere additiv für jeden Maschenpunkt $M_{i\in n}$ seinen aktuellen Radius für jeden seiner Dreieckspunkte $\vec{v}_{1,2,3}\in\Delta_{i\in n}$, und erhöhe den Radius-Zähler des Dreieckspunktes um 1.
4. Bilde das arithmetische Mittel aller Radien, indem für jeden Dreieckspunkt die addierten Radien durch die mitgezählte Anzahl geteilt werden.
5. Bilde das Produkt aus dem so erhaltenen Radius rad_{new} und dem Einheitsvektor des Dreieckspunktes und speichere diesen neuen Dreieckspunkt in den Dreiecksstrukturen überall dort, wo vorher der alte Dreieckspunkt vorkam.

Die Dreiecksstrukturen stellen nun eine durch inverse Duale Graph Transformation herbeigeführte Triangulierung dar (Abbildung 25).

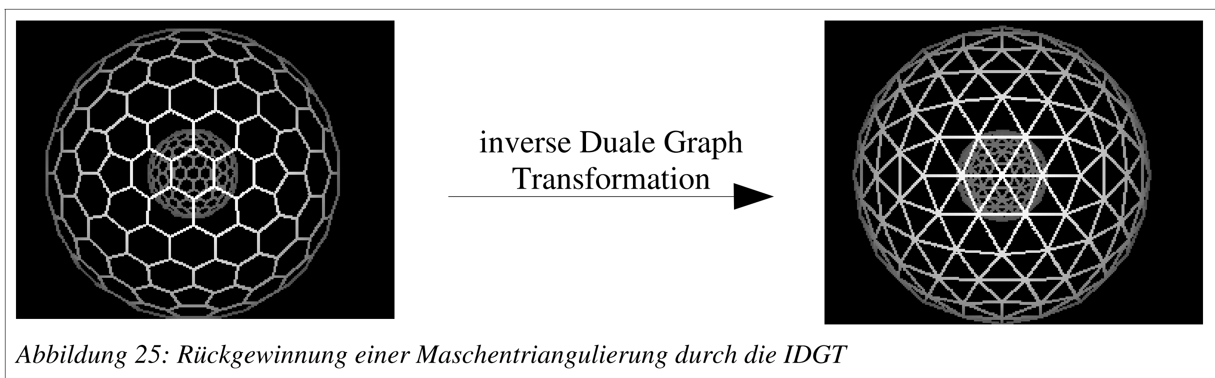


Abbildung 25: Rückgewinnung einer Maschentriangulierung durch die IDGT

4.3.5.3 Grafische Darstellung der Triangulierung

Die durch die IDGT erhaltenen Dreiecksstrukturen bestehen aus vielen einzelnen Dreiecken gegeben durch die Eckpunkte dieser. Diese auszugeben kann durch unterschiedliche Art und Weise erfolgen. Hierzu wurden zwei verschiedene Verfahren eingesetzt:

1. Die veränderten Dreiecksstrukturen werden an einen externen Renderer übergeben.

   Dies hat sich besonders in der Testphase als sehr dienlich erwiesen, da die Arbeit der Abbildung der dreidimensionalen Dreiecksdaten an ein externes Programm übergeben wurde.

   Es wurde in folgender Methode der Klasse „`SMesh`" implementiert:

   „`void Smesh3D::getTriangulation()`"

   Diese gibt ein Array mit den Dreiecksstrukturen in der Konsole aus. Diese Struktur ist auf den Renderer POV-Ray© abgestimmt, zur Anzeige müssen lediglich einige Zeilen, die auch in der Methode kommentiert enthalten sind, der Szenenbeschreibung hinzugefügt werden.

   Das Problem, dass bei dieser Art der Ausgabe auftritt ist, dass die Triangulierung der Masche komplett vom Volumen losgelöst ist. Es ist hier leicht möglich, die Form der deformierten Masche subjektiv zu beurteilen, eine objektive Beurteilung anhand der Volumendaten kann aber nicht erfolgen.

2. Die Dreiecke werden mithilfe eines eigenen Verfahrens im dreidimensionalen Bildraum gefüllt und stehen dann für weitere Bearbeitungsschritte zur Verfügung.

   Das Verfahren beschreibt sich als ein leicht variierter Scanline-Algorithmus in 3D und soll im Folgenden kurz erläutert werden.

   Gegeben sei ein Dreieck durch seine drei Eckpunkte $\vec{a}$, $\vec{b}$, $\vec{c}$.

   1. Ausgehend vom Punkt $\vec{a}$, bestimme die Distanzen zu den Nachbarn $\vec{b}$: $d_b=|\vec{b}-\vec{a}|$ und $\vec{c}$: $d_c=|\vec{c}-\vec{a}|$ sowie die Steigungen zu den Nachbarn $\vec{b}$: $\vec{m}_b=\vec{b}-\vec{a}$ und $\vec{c}$: $\vec{m}_c=\vec{c}-\vec{a}$.

2. Setze die Anzahl der zu zeichnenden Linien $steps = max(d_b, d_c) \cdot n$, mit $n \geq 2$.

3. Setze die Schrittweite zum Nachbarn $\vec{b}$ auf $\vec{\delta}_b = \frac{\vec{m}_b}{steps}$ zum Nachbarn $\vec{c}$ auf $\vec{\delta}_b = \frac{\vec{m}_c}{steps}$.

4. Für $i = 0$ bis $steps$:

   Zeichne eine Linie der Breite 2 von $\vec{a} + \vec{\delta}_b \cdot i$ nach $\vec{a} + \vec{\delta}_c \cdot i$.

   Setze $\vec{a} = \vec{b}$, $\vec{b} = \vec{c}$ und $\vec{c} = \vec{a}$ und führe die Schritte 1. - 4. erneut aus.

   Setze $\vec{a} = \vec{c}$, $\vec{b} = \vec{a}$ und $\vec{c} = \vec{b}$ und führe die Schritte 1. - 4. erneut aus.

Der Algorithmus führt also drei Füllungen ähnlich eines Scanline-Algorithmus` durch, und zwar durch Zeichnen von parallelen Linien jeweils von einer Dreiecksseite ausgehend auf den gegenüberliegenden Punkt. Dies wird in Abbildung 26 veranschaulicht.

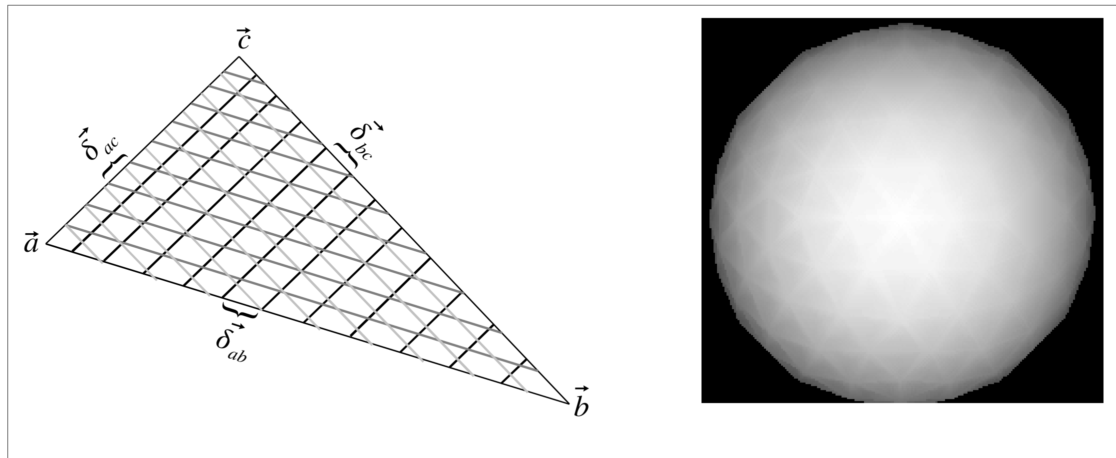


Abbildung 26: Links: Schematische Veranschaulichung des Dreieckfüllverfahrens in 3D. Rechts: Eine Masche, die mit der IDGT trianguliert wurde, nach Anwendung des Dreieckfüllverfahrens

Ausgehend von dem zweiten der beschriebenen Dreieckfüllverfahren kann nun also die Triangulierung einer Masche im Volumenraum gefüllt werden. Dieses Vorgehen hat den Vorteil, dass eine durchgängige Kontur in die segmentierten Daten gezeichnet werden kann. Dies ist vor allem für den Vergleich zwischen dem zweidimensionalen und dem dreidimensionalen Verfahren sehr nützlich.

Aber eine viel bedeutendere Möglichkeit offenbart sich zugleich: Durch eine geschlossene Oberfläche im dreidimensionalen Raum und das Zentrum der Dual Simplex Mesh ist das segmentierte Volumen gegeben. Da sich das Maschenzentrum trivialerweise in der Masche befindet, befinden sich auch alle Punkte, die auf einem Strahl ausgehend vom

Maschenzentrum und vor der Kontur – gegeben durch die Füllung der Dreiecke – innerhalb des segmentierten Volumens.

Mit diesen Informationen lässt sich eine Segmentierungsmaske erstellen, ein Volumen, das für alle Voxel innerhalb der Segmentierung 1 und für alle anderen Voxel 0 ist. Um zu diesem Maskierungsvolumen zu gelangen wurde ein einfacher Volumenfüll-Algorithmus („`simpleFill3D`“) in der `main`-Klasse implementiert. Dieser arbeitet wie folgt:

1. Erstelle eine Knotenliste *frontier* .
2. Füge den Startpunkt $\vec{s}$ der Liste hinzu.
3. Solange die Liste nicht leer ist:
   3.1.Entnehme das letzte Element der Liste *frontier* .
   3.2.Prüfe, ob das Volumen an der Stelle des Elementes gleich 0 ist.
      3.2.1.Falls ja: Setze das Volumen an der Stelle des Elementes auf 1.
      3.2.2.Füge den oberen, unteren, linken, rechten, vorderen und hinteren Nachbarn des Elementes der Liste *frontier* hinzu.
   3.3.Gehe zu Schritt 3.

Nach Ablauf des Algorithmus` befindet sich nun also in dem Volumen eine Maske, anhand derer das Volumen maskiert werden kann. Nach der Maskierung müssen die segmentierten Daten ausgegeben werden. Diese Visualisierung dreidimensionaler Volumendaten behandelt das nächste Unterkapitel.

Der Algorithmus besitzt dabei allerdings die Schwäche, dass die Knotenliste *frontier* nicht beim Einfügen von Elementen überprüft, ob diese sich bereits auf der Liste befinden. So kann sich ein Knoten mehrfach auf der Liste befinden, demzufolge wird der Speicherbedarf höher, als er es eigentlich sein müsste. Auch der Zeitbedarf ist keineswegs als ideal zu bezeichnen, da die mehrfach vorkommenden Knoten auch mehrfach überprüft werden müssen.

4.3.5.4 Rotation und Projektion

Damit ein dreidimensionales Volumen nicht immer nur von einer Seite aus betrachtet werden muss, wurde eine dreidimensionale Transformation – nämlich die Rotation – implementiert. Sie ist in der Klasse „`Rotator`“ in der Methode „`rotiere`“ vertreten. Diese Rotationsmethode stellt eine Look-From-Target Funktion dar, d.h. es wird im Zielvolumen für jedes Voxel das entsprechende Voxel vor der Rotation im Quellvolumen berechnet. Dies vermeidet fehlende, durch Rundung entstandene Voxel im rotierten Volumen.

Der Methode werden folgende Parameter übergeben:

- Ein eindimensionales Array (`args`) vom Typ „`float`“ mit folgenden Einträgen:
    - x-Komponente des Einheitsvektors der Rotationsachse
    - y-Komponente des Einheitsvektors der Rotationsachse
    - z-Komponente des Einheitsvektors der Rotationsachse
    - Drehung um die Rotationsachse (in Grad)
- Die Referenz auf das Quellvolumen „`volume`“
- Die Referenz auf das Zielvolumen „`rotvol`“

Nach Aufruf der Funktion befindet sich ein in Abhängigkeit der Parameter rotiertes Volumen in „`rotvol`“.

Um diese dreidimensionalen Volumendaten auf einer zweidimensionalen Bildfläche abzubilden, wird eine Projektion benötigt. Unter Projektion versteht man in der Computergrafik eine Abbildung von einem k-dimensionalen Vektorraum in einen $(k-1)$-dimensionalen (Unter-)Vektorraum. Im Speziellen bezeichnet hier eine Projektion eine Abbildung aus einem dreidimensionalen Raum in eine zweidimensionale Bildfläche.

Es wurden drei verschiedene Arten der Projektion in der Klasse „`Renderer`“ implementiert. Allen gemein ist, dass sie orthogonale Projektionen darstellen, d.h. die Strahlen, die die Volumendaten auf die Bildfläche abbilden, verlaufen parallel, im Speziellen der Implementation verlaufen sie zusätzlich orthogonal zur x-y-Ebene. Folgende Methoden wurden implementiert:

- *Maximum Intensity*-Projektion (Methode: „`renderMIP`“)

  Das Maximum der Intensität eines jeden Voxels, auf das der Projektionsstrahl trifft wird auf die Bildebene abgebildet.

- *Summen*-Projektion (Methode: „`renderSUM`“)

  Die Intensitäten aller Voxel, auf die der Projektionsstrahl trifft werden aufsummiert und diese Summe wird auf die Bildebene abgebildet. Das Ergebnis entspricht anschaulich einer Röntgenaufnahme.

- *Tiefen*-Projektion (Methode: „`renderDEP`“)

  Die Intensität des ersten Voxels oberhalb einer Intensitätsgrenze, auf das der Projektionsstrahl trifft, wird in Abhängigkeit von der Tiefe des Voxels gewichtet auf die Bildebene abgebildet. Dabei nimmt die Intensität der Voxel mit der Tiefe quadratisch ab. Diese Projektion entspricht einer Oberflächenabbildung des Köpers.

Anhand der Rotation und Projektionen kann eine subjektive Beurteilung der Volumendaten auf einem Bildschirm erfolgen, indem ein Volumen in verschiedene Richtungen rotiert und anschließend auf eine Bildfläche projiziert wird.

5 Leistungsmessungen und Ergebnisse

In diesem Kapitel soll es um die Untersuchung der Geschwindigkeit und des Speicherbedarfes der 2D- sowie der 3D-Implementation gehen. Zuerst sei gesagt, dass die Testumgebung in der die Messungen stattgefunden haben, schon in Kapitel 4.1 beschrieben wurde.

Es werden anhand beispielhafter Bild- bzw. Volumendaten einige Testläufe vorgestellt, und jeweils die verschiedenen Leistungsdaten gemessen. Abgeschlossen werden die einzelnen Kapitel durch die Diskussion der Ergebnisse der Leistungsdaten im jeweils folgenden Kapitel. Dies erfolgt sowohl für den zweidimensionalen als auch für den dreidimensionalen Fall. Hier sollen auch einige Komplexitätseinschätzungen gegeben werden.

5.1 Ergebnisse der Leistungsmessungen in 2D

Das Testszenario bestand aus zwei zu segmentierenden Bildern. Jedes Bild wurde in Bezug auf den Einfluss der Eingabeparameter untersucht, wobei die Referenzwerte (jeweils die erste Spalte der Tabellen) jedes Bildes mit sechs Variationen verglichen wird.

Es werden für jede Konfiguration folgende Daten gemessen:

- Zeitbedarf [s]

  Der Zeitbedarf wird für drei Bereiche gemessen:

  1. Allokation des Speichers, Vorberechnung des Gradientenbildes und des Gradientenvektorfeldes
  2. Finden der Objektmarkierungen
  3. Annäherung der Maschen

- Speicherbedarf [kB]

  Der Speicherbedarf wird an drei Stellen während des Ablaufes gemessen:

  1. Direkt nach dem Programmstart
  2. Ungefähr bei der Hälfte der Ausführungszeit
  3. Beim Programmende

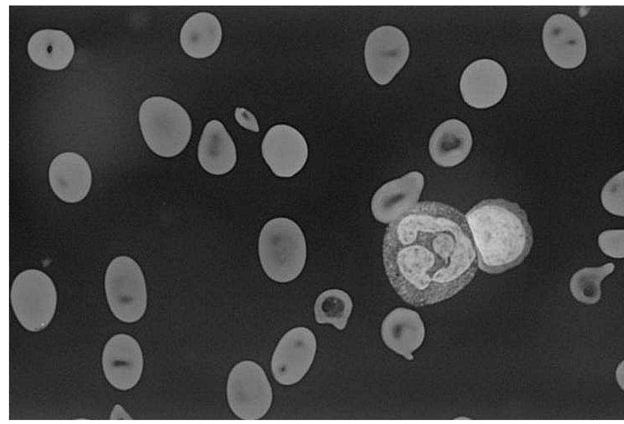
Für jede Konfiguration wurden diese Werte zwei Mal gemessen, einmal mit den Standard Compiler Optionen und einmal mit optimierten Einstellungen. Dabei wurden folgende Einstellungen gewählt:

- Geschwindigkeit optimieren
- Globale Optimierung aktivieren
- Systeminterne Funktionen verwenden
- Geschwindigkeit vorziehen
- Für Prozessor Pentium© 4 und höher optimieren
- Für Windows optimieren

Durch dieses Testszenario ergeben sich 28 Einzelmessungen, welche im Folgenden in zwei Tabellen dargestellt sind (Abbildung 27 und Abbildung 28).

Hier nun die Ergebnisse des Testlaufes mit den Bilddaten des ersten Bildes:

1. Bild: 360885 Pixel, ~24 Objekte



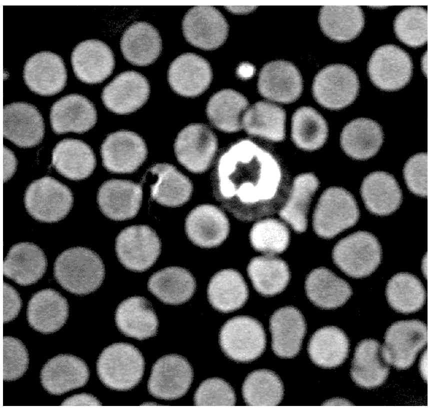
Messergebnisse:

			Referenz	Variationen					
Eingabewerte		smooth	0,6	0,2	0,6	0,6	0,6	0,6	0,6
		gauss	2	2	5	2	2	2	2
		stepwidth	1	1	1	0,1	1	1	1
		innerRad	20	20	20	20	10	30	20
		outerRad	50	50	50	50	50	50	50
		points	50	50	50	50	50	50	100
ohne Compiler Optimierung	Zeit	1. Messung	1,406	1,313	3,390	1,312	1,297	1,297	1,282
		2. Messung	126,437	126,218	114,844	126,422	34,093	273,218	126,375
		3. Messung	0,110	0,110	0,094	0,906	0,282	0,063	0,187
		Summe	127,95	127,64	118,33	128,64	35,67	274,58	127,84
	Speicher	1. Messung	18.484	18.484	18.484	18.488	18.472	18.500	18.484
		2. Messung	19.896	19.896	19.896	19.900	19.884	19.912	19.896
		3. Messung	17.128	17.128	17.128	17.132	17.116	17.164	17.128
		Mittelwert	18.503	18.503	18.503	18.507	18.491	18.525	18.503
mit Compiler Optimierung	Zeit	1. Messung	0,328	0,328	0,609	0,328	0,328	0,328	0,328
		2. Messung	79,937	79,937	73,469	79,781	20,781	171,515	80,047
		3. Messung	0,047	0,047	0,047	0,344	0,110	0,032	0,078
		Summe	80,312	80,312	74,125	80,453	21,219	171,875	80,453
	Speicher	1. Messung	18.312	18.316	18.316	18.312	18.304	18.328	18.316
		2. Messung	19.724	19.728	19.728	19.724	19.716	19.740	19.728
		3. Messung	16.952	16.976	16.976	16.952	16.944	16.952	16.976
		Mittelwert	18.329	18.340	18.340	18.329	18.321	18.340	18.340

Abbildung 27: Bilddaten und Messergebnisse des ersten Bildes

Und hier die Ergebnisse des Testlaufes mit den Bilddaten des zweiten Bildes:

2. Bild: 381000 Pixel,
~38 Objekte



Messergebnisse:

			Referenz	Variationen					
Eingabewerte		smooth	0,4	0,1	0,4	0,4	0,4	0,4	0,4
		gauss	3	3	5	3	3	3	3
		stepwidth	1	1	1	0,1	1	1	1
		innerRad	20	20	20	20	10	30	20
		outerRad	50	50	50	50	50	50	50
		points	30	30	30	30	30	30	100
ohne Compiler Optimierung	Zeit	1. Messung	2,156	2,156	3,531	2,172	2,172	2,172	2,156
		2. Messung	155,437	155,453	154,078	155,500	41,781	333,984	155,547
		3. Messung	0,141	0,141	0,156	1,266	0,313	0,094	0,438
		Summe	157,734	157,750	157,765	158,938	44,266	336,250	158,141
	Speicher	1. Messung	19.048	19.048	19.052	19.056	19.040	19.056	19.044
		2. Messung	20.540	20.540	20.544	20.548	20.532	20.548	20.536
		3. Messung	17.620	17.620	17.624	17.628	17.616	17.628	17.620
		Mittelwert	19.069	19.069	19.073	19.077	19.063	19.077	19.067
mit Compiler Optimierung	Zeit	1. Messung	0,422	0,406	0,594	0,406	0,422	0,422	0,422
		2. Messung	95,250	95,454	94,453	95,266	24,937	205,515	95,187
		3. Messung	0,063	0,062	0,063	0,468	0,391	0,047	0,188
		Summe	95,735	95,922	95,110	96,140	25,750	205,984	95,797
	Speicher	1. Messung	18.800	18.800	18.800	18.876	18.872	18.888	18.800
		2. Messung	20.372	20.372	20.372	20.368	20.364	20.380	20.372
		3. Messung	17.448	17.448	17.448	17.416	17.420	17.480	17.448
		Mittelwert	18.873	18.873	18.873	18.887	18.885	18.916	18.873

Abbildung 28: Bilddaten und Messergebnisse des zweiten Bildes

5.2 Diskussion der 2D-Ergebnisse

Nach diesen zugegebenermaßen recht umfangreichen Tabellen wird im Folgenden diskutiert werden, welche Veränderungen in welchen Messergebnissen resultieren.

Zuerst fällt der recht konstante Speicherbedarf der Implementation auf. Er scheint lediglich von der Bildauflösung (also von der Anzahl der Pixel) abzuhängen, denn keine Veränderung der Eingabeparameter führt hier zu einem signifikant verändertem Ergebnis. Lediglich die höhere Auflösung des zweiten Bildes kann also dazu beigetragen haben.

Wenn der Compiler die Geschwindigkeit des Programmablaufes optimiert, so sinkt auch der Speicherverbrauch, allerdings variiert er auch hier nicht in Abhängigkeit der Eingabeparameter. Ebenfalls nicht verwunderlich ist die Tatsache, dass das Programm bei allen Messergebnissen während des Ablaufes einen höheren Speicherbedarf aufweist, als zu Anfang oder zu Ende.

Zu den Auswirkungen, die eine Veränderung der Eingabeparameter auf den Zeitbedarf des Programmes haben lässt sich folgendes festhalten:

- Der Glattheitsfaktor: `smooth`

  Eine Variation des Glattheitsfaktors scheint keine Auswirkungen auf den Zeitbedarf zu haben. Dies erscheint nur logisch, so werden hier lediglich Variablen diverser Gleitkommarechnungen ausgetauscht.

- Die Varianz des Gauß-Filters des Gradienten: `gauss`

  Der erste Einfluss dieser Variable lässt sich leicht feststellen: Sie beeinflusst unmittelbar den Zeitbedarf, der nötig ist und das Gradientenvektorfeld und damit auch das Gradientenbild zu erstellen (1. Zeitnahme in den Tabellen bei 5.1). Je höher diese Eingabevariable gewählt wird, desto höher wird auch der Wert bei dieser Zeitnahme ausfallen.

  Beim ersten Bild fällt zudem ein verminderter Zeitbedarf für das Finden der Marken sowie für die Annäherung der Maschen ins Auge. Dies lässt sich dadurch erklären, dass das Erscheinungsbild des Gradientenbildes auch maßgeblich den Prozess des Findens der Objektmarkierungen beeinflusst. So werden nach der Subtraktion (Schritt 5 in (3.4), Abbildung 19) schon deutlich mehr Pixel verworfen, womit sich dieser Geschwindigkeitszuwachs erklären lässt.

  Dies wiederum beeinflusst auch die Anzahl der gefundenen Objektmarkierungen und damit auch die Ausführungsgeschwindigkeit das Prozesses der Annäherung der Maschen. So lässt sich auch hier der Geschwindigkeitszuwachs erklären.

- Die Schrittweite: `stepwidth`

  Dieser Wert diente mehr der Kontrolle, denn hier sollte sich zeigen, dass sich nur die Zeit der Annäherung der Maschen verändert. Dies sollte annähernd umgekehrt proportional geschehen, da eine Verkürzung der Schrittweite zu einer Vergrößerung der Anzahl der zu tätigenden Schritte führt. Und tatsächlich wurde ohne Compiler Optimierungen ungefähr eine Erhöhung um den Faktor acht erreicht, wenn die Schrittweite um den Faktor zehn verringert wurde.

- Der Radius der inneren Masche: `innerRad`

  Diese Variable hat einen sehr großen Einfluss auf den Zeitbedarf des automatischen Findens der Objektmarkierungen. Dies begründet sich darin, dass die dabei erstellte Faltungsmaske genau „`(2 · innerRad+1)`^{2}“ Pixel umfasst, und die Faltung damit entsprechend zeitaufwendiger wird.

  So ergeben sich im ersten Bild schon bei einer Vergrößerung des inneren Radius um 10 Pixel ein Wachstum im Zeitbedarf um mehr als das doppelte. Dabei wird interessanterweise der Prozess der Maschenannäherung beschleunigt. Dies

begründet sich darin, dass durch den vergrößerten inneren Maschenradius weniger Objektzentren gefunden werden, für die eine Masche erstellt wird und schließlich eine Annäherung erfolgt.

Genau der umgekehrte Fall tritt bei einer Verkleinerung des inneren Maschenradius` auf. Im Beispiel des ersten Bildes führt eine Halbierung des inneren Maschenradius` zu einem um den Faktor vier kürzeren Zeitbedarf bei dem Finden der Objektzentren. Dafür werden derer mehr gefunden, und so erklärt sich dann auch ein leichter Anstieg in den Zeitnahmen des Prozesses, welcher für die Maschenannäherung verantwortlich ist.

- Die Anzahl der Maschenpunkte: `points`

  Wie die Variable `stepwidth` sollte eine erhöhte Anzahl der Maschenpunkte auch lediglich zu einer Erhöhung des Zeitbedarfes der Maschenannäherung führen. Dies, und dass sonst keine Zeitwerte beeinflusst werden, bestätigt sich in den Messergebnissen wie erwartet.

Abschließend lassen sich noch folgende Dinge festhalten:

Die Annäherung der Maschen benötigt bei weitem die wenigste Zeit. Das Laden in den Speicher ist auch noch relativ wenig zeitaufwendig. Mit einem riesigen Abstand und dem bei weitem höchsten Zeitbedarf zeichnet sich das Finden der Objektmarkierungen aus. Deswegen ist der innere Maschenradius, welcher diesen Zeitbedarf am meisten beeinflusst auch am vorsichtigsten zu wählen.

Außerdem ist die Möglichkeit, die der Compiler durch Optimierungsmöglichkeiten bereitstellt, sehr effektiv. So sind die Zeiten, die mit aktivierter Optimierung des Compilers gemessen wurden, um etwa den Faktor 1,6 geringer als jene ohne Optimierung.

5.3 Ergebnisse der Leistungsmessungen in 3D

Das Testszenario bestand aus Knappheit an Zeit sowie Volumendatensätzen lediglich aus einem Volumen. Es wurde der Einfluss der wichtigsten Eingabeparameter untersucht, wobei die Referenzwerte (die erste Spalte der Tabelle) dieses Volumens mit drei Variationen verglichen wurden.

Dabei werden für jede Konfiguration folgende Daten gemessen:

- Zeitbedarf [s]

  Der Zeitbedarf wird für drei Bereiche gemessen:

  1. Erstelle die Speicherstrukturen
  2. Laden der Volumendaten
  3. Vorbereitung des Dual Simplex Meshes - Prozessors
  4. Annäherung der Maschen
  5. Maskierung des Volumen
  6. Ausgabe inkl. Drehen und Rendern des Volumens

- Speicherbedarf [kB]

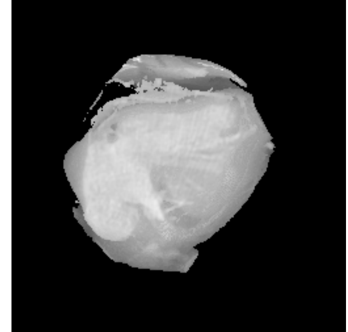
  Hier wird der maximale Speicherbedarf des Programmes während seiner Ausführung gemessen.

Für jede Konfiguration wurden diese Werte zwei Mal gemessen, einmal mit den Standard Compiler Optionen und einmal mit optimierten Einstellungen. Dabei wurden für die Optimierungen die gleichen Parameter verwendet wie in Kapitel 5.1.

Durch dieses Testszenario ergeben sich acht Einzelmessungen, welche in der folgenden Tabelle (Abbildung 29) dargestellt sind.

Zu ladendes Volumen: 512^3 Voxel,

Arbeitsvolumen: 256^3 Voxel.



Messergebnisse:

			Referenz	Variationen		
Eingabe-werte		smooth	0,2	0,9	0,2	0,2
		stepwidth	1	1	0,5	1
		points	320	320	320	1.280
ohne Compiler Optimierung	Zeit	1. Messung	1,219	1,157	1,203	3,765
		2. Messung	15,187	14,640	14,812	14,578
		3. Messung	13,797	13,782	13,641	14,047
		4. Messung	7,078	7,078	14,344	28,531
		5. Messung	11,281	8,937	11,406	12,110
		6. Messung	146,359	145,703	146,218	147,516
		Summe	194,921	191,297	201,624	220,547
		Summe ohne 6.	48,562	45,594	55,406	73,031
	Speicher	Maximalwert	339.080	339.080	339.080	339.436
mit Compiler Optimierung	Zeit	1. Messung	0,468	0,391	0,406	0,719
		2. Messung	11,907	11,109	10,360	10,750
		3. Messung	6,859	6,875	6,859	7,063
		4. Messung	3,469	3,485	6,953	14,781
		5. Messung	7,937	6,687	7,953	8,234
		6. Messung	79,390	76,484	76,750	76,922
		Summe	110,030	105,031	109,281	118,469
		Summe ohne 6.	30,640	28,547	32,531	41,547
	Speicher	Maximalwert	317.016	317.016	317.016	317.392

Abbildung 29: Messergebnisse der Segmentierung des Volumens

5.4 Diskussion der 3D-Ergebnisse

Die Tabelle, welche im vorigen Kapitel abgebildet ist, ist ja schon übersichtlicher, als jene beiden Leistungsmesstabellen aus Kapitel 5.1. Dennoch kann auch aus ihr einiges abgelesen werden.

Zu erst fällt, wie auch im zweidimensionalen Teil, der recht konstante Speicherbedarf der Implementation auf. Er hängt lediglich davon ab, ob die Compiler Optimierungen an- oder abgestellt sind. Außerdem kann man ihn nun mit der Punktanzahl der Maschen in Bezug setzen. Denn je mehr Punkte eine Masche enthält, desto größer wird auch die Knotenliste jeder Masche im Speicher, was den erhöhten Bedarf an Speicher erklärt.

Die Diskussion der Zeitwerte stellt sich als weniger konstant dar. Da aber die sechste Messung nur das Rotieren und Rendern des Ergebnisses der Segmentierung beinhaltet, wird es im Folgenden nicht weiter beachtet. Zu den Auswirkungen, die eine Veränderung der Eingabeparameter auf den Zeitbedarf ohne diesen Schritt des Programmes haben, lässt sich folgendes festhalten:

- Der Glattheitsfaktor: `smooth`

  Eine Variation des Glattheitsfaktors scheint beinahe keine Auswirkungen auf den Zeitbedarf zu haben. Allerdings lässt sich beim Maskierungsschritt des Volumens ein Geschwindigkeitszuwachs feststellen. Dies erklärt sich dadurch, dass durch einen höheren Wert das Ergebnisvolumen nach der Segmentierung deutlich kleiner ist, und somit der in (4.3.5.3) beschriebene Füllalgorithmus schneller arbeitet.

- Die Schrittweite: `stepwidth`

  Dieser Wert diente wiederum der Kontrolle, denn hier sollte sich zeigen, dass sich nur die Zeit der Annäherung der Maschen verändert. Dies sollte annähernd umgekehrt proportional geschehen, da eine Verkürzung der Schrittweite zu einer Vergrößerung der Anzahl der zu tätigenden Schritte führt. Und tatsächlich wurde sowohl mit, als auch ohne Compiler Optimierungen ungefähr eine Erhöhung des Zeitbedarfs um den Faktor zwei erreicht, wenn die Schrittweite um den Faktor zwei verringert wurde (siehe vierter Zeitmesspunkt). Die anderen Werte blieben im Vergleich zu den Referenzwerten unverändert.

- Die Anzahl der Maschenpunkte: `points`

  Eine erhöhte Anzahl der Maschenpunkte sollte einerseits zu einer Erhöhung des Zeitbedarfes der Maschenannäherung führen. Andererseits sollte sich auch der Zeitbedarf des ersten Messpunktes erhöhen, da hier der rekursiv unterteilte Ikosaeder um einige Rekursionsebenen tiefer absteigen muss. Dies bestätigt sich in den Messergebnissen wie erwartet. Allerdings fällt auf, dass sich bei einer Erhöhung der Maschenpunkte um den Faktor 40 das Ergebnis der Zeitmessung im vierten Messpunkt nur um den Faktor vier erhöht – was dieses Verfahren auch für höhere Anzahlen an Maschenpunkten noch durchaus praktikabel macht

Abschließend lassen sich noch folgende Dinge festhalten:

Der Speicherbedarf ist erwartungsgemäß sehr viel höher als der der 2D Implementation. Außerdem benötigen zur Zeit alle Schritte, ausser dem Laden der Volumendaten und dem

Rendern der segmentierten Daten, ähnlich viel Zeit. Was sich grundlegend geändert hat, ist der Zeitbedarf der Maschenannäherung. Dieser benötigt jetzt aufgrund der komplexeren Berechnung (siehe 2.2.2) deutlich mehr Zeit. Auch die Maskierung des Volumens benötigt bei der vorliegenden Implementation noch recht viel Zeit, hier sei aber auf (4.3.5.3) hingewiesen, wo schon einige Verbesserungsmöglichkeiten des verwendeten Algorithmus` beschrieben wurden.

Außerdem ist die Möglichkeit, die Compileroptimierung zu aktivieren, auch für die 3D Implementation sehr effektiv. So sind die Zeiten, die mit aktivierter Optimierung des Compilers gemessen wurden, um etwa den Faktor 1,7 geringer als jene ohne Optimierung.

6 Erfahrungen und Probleme

Im Laufe der Zeit der Beschäftigung mit den Dual Simplex Meshes ergaben sich immer wieder neue Erkenntnisse. Häufig waren die ersten Ansätze der Implementation nicht die optimalen, was sich aber erst durch Ausprobieren mit verschiedenen Testdaten herausstellte. Dabei spielte häufig auch die Qualität dieser Daten (Testbilder) eine große Rolle. Hier kommt auch schon die erste große Schwierigkeit ins Spiel: Bilder bzw. Volumendaten, die sich verwenden lassen, sind relativ schwer und vereinzelt beschaffbar. Selbst erstellte Bilder bieten sich zwar für die ersten Tests durchaus an, zum Beispiel weil man gut sehen kann, ob die Maschen sich gleichmäßig bewegen, d.h. ihre Kreis- bzw. Kugelform beibehalten, bis sie an einen Gradienten stoßen. Da das Verfahren aber für natürliche Bilder bzw. Volumendaten vorgesehen ist, sollte man auch solche verwenden, und kann auch nur anhand dieser die Wichtigkeit der richtigen Parameter beurteilen.

Um also an natürliche Bilder zu gelangen, wurde die Funktion bekannter Internet-Suchmaschinen, wie z.B. *Google*©, *Altavista*© und *Lycos*©, nach Bildern zu suchen benutzt. Leider waren fast alle Suchergebnisse unbrauchbar, da sie zu gering aufgelöst waren (was nicht heißt, dass die Bilder immer selbst eine geringe Auflösung hatten, sondern auch, dass häufig einfach zu viele Objekte in diesen Bildern abgebildet waren, die dann nur noch wenige Pixel groß waren). Letztendlich fanden sich aber doch einige Bilder, die verwendet werden konnten, und die auch teilweise in dieser Ausarbeitung als Beispiele zu finden sind. Beim Suchen nach Volumendaten stellte sich dies noch deutlich schwerer dar, so dass in dieser Arbeit lediglich ein Volumen benutzt wird um die Implementation in 3D zu veranschaulichen.

6.1 Probleme mit dem Vorlagenpaper

Das Vorlagenpaper [SM03a], welches die Basis des Projektes darstellt, war leider recht kurz und vermittelte nur einen zu kleinen Einblick in das Thema. Zwar war es zum Anfang genau das Richtige, um einen Überblick über die Thematik zu bekommen, und dieses auch anderen im Projekt vorzustellen, doch war es spätestens beim Beginn der Implementation viel zu knapp. Wesentliche Dinge wurden einfach viel zu kurz oder gar nicht erläutert. So wurde zum Beispiel auf die Winkelberechnung für die interne Energie oder die Definition der Dual Simplex Mesh, welche auf den Star-Shaped Simplex Meshes basieren nicht näher beschrieben.

Leider wurde anfangs versucht im Internet nach weiterem Material zu suchen, wobei das Ergebnis sehr häufig aus Seiten bestand, bei denen die Texte bezahlt werden mussten, z.B. *SpringerLink*©. Später wurde bekannt, dass von der Universität aus alle Texte von *SpringerLink*© kostenfrei zugänglich sind, was half, da diese Methode der Segmentierung noch relativ neu war, und deswegen auch ausschließlich neuere Publikationen zur speziellen Vertiefung in Frage kamen.

Mit Hilfe von [SM01] war so zumindest die genaue Definition der Dual Simplex Meshes bekannt. Auch wurde darin erläutert, welcher Winkel der ausschlaggebende für die interne Energie ist. Nachdem alle anfänglichen Probleme aus dem Weg geräumt waren, begann die Implementation, und schon sehr bald stellten sich erste Erfolge ein. Denn es stellte sich sehr bald heraus, dass das Verfahren - zumindest in 2D - trotz seiner von den Autoren Svoboda

und Matula erwähnten guten Ergebnisse, nicht zu komplex war und auch von Studierenden erfolgreich gemeistert werden konnte.

Für die Implementation in 3D stellte sich zuerst die Frage der Punktverteilung auf einer Kugeloberfläche. Leider äußerten sich die Autoren in keinster Weise darüber, in welcher Art dieses geschehen kann. Auch in anderen ihrer Publikationen blieb dieses Verfahren ungenannt, so dass nur übrigblieb, die Verweise im Literaturverzeichnis nach Arbeiten zu durchsuchen, die dieses Thema behandelten.

6.2 Schwierigkeiten bei der Implementation

Zuerst wurde die Implementation für die Segmentierung in zweidimensionalen Bildern vorgenommen. Dabei standen zuerst die elementaren Bestandteile des Algorithmus` im Vordergrund. So war zum Beispiel anfangs der Algorithmus zur automatischen Bestimmung der Objektmarkierungen nicht von höchster Priorität, da dieser entbehrlich war. Zudem sind dieser Algorithmus und dessen formale Hintergründe anfangs nicht richtig verstanden worden. Erst ganz am Schluss, als eigentlich schon die Implementation des Algorithmus in 2D in der Endfassung vorlag, wurde entschieden auch die Implementierung dieses Teilalgorithmus` anzugehen.

Immerhin wuchsen im Laufe des Semesters die einigen Erkenntnisse, vor allem durch die Vorlesung ‚Bildverarbeitung I'. So waren Hough-Transformation und Faltung keine Fremdwörter mehr, sondern verstandenes Handwerkszeug. Auch die größer gewordene Erfahrung mit der *VIGRA*-Bibliothek war natürlich hilfreich, obwohl Hilfe von Personen mit mehr Erfahrung doch immer wieder erforderlich war.

Ebenfalls fremd war die verwendete Programmiersprache C++. Im Laufe des Studiums wurden zwar schon Programmiersprachen wie Java oder Bruchstücke von C gelernt, allerdings stellte C++ anfangs eine neue Herausforderung dar. Dies wurde jedoch durch eine achtwöchige Einführung in die Programmiersprache kompensiert. Jedoch war dies nur ein Einstieg, zur Implementation des beschriebenen Algorithmus fehlte doch noch einiges. So traten doch anfangs noch häufiger spezifische Probleme der Programmiersprache auf, und auch Optimierungen des Quelltextes kamen erst im Laufe der Zeit hinzu.

Auch bei der Aufteilung in Klassen war es nicht immer einfach zu entscheiden, welche Methode in welche Klasse gehört. So gab es zum Beispiel am Ende doch erhebliche Probleme, das automatische Finden der Objektmarkierungen in das bestehendes Projekt einzugliedern. Dennoch ist es hoffentlich ausreichend gut gelungen.

Die Tücke bei der Implementation lag oft auch im Detail. So kam es recht schnell zu Erfolgserlebnissen, als das erste Testbild mit sehr hohen Gradienten und nur einem Objekt erfolgreich segmentiert wurde. Aber schon der darauffolgende Umstieg auf nicht-idealisierte Bilddaten offenbarte, dass es nicht so einfach gewesen war. Je schlechter das Bildmaterial gewählt wurde (mehr Rauschen, unscharfe Konturen), desto offensichtlicher wurden die Fehler.

Dies fiel beim idealisierten Anfangsbild noch nicht auf, hier konnte sich noch auf den Gradienten verlassen werden. So wurde die Berechnung der internen Energie mehrmals

überarbeitet und sogar noch die Berechnung der externen Energie verbessert, indem eine implizite Gauß-Faltung eingebettet wurde.

Abschließend kann deshalb festgehalten werden, dass zumindest die Implementation in 2D recht gut geklappt hat, und somit motivierte diesen Algorithmus auch für sein eigentliches Einsatzgebiet – die Segmentierung in 3D – zu implementieren. Dabei sei angemerkt, dass hierbei ganz neue Probleme auftraten, und ein Wechsel von 2D nach 3D keineswegs eine triviale Aufgabe ist, da es eine Vielzahl an Dingen zu beachten gibt, die im 2D Teil noch keine oder nur eine untergeordnete Rolle spielten.

Wie bereits oben in diesem Kapitel erwähnt, stellte die Punktverteilung auf einer Kugeloberfläche kein triviales Problem dar. Gesucht war eine Möglichkeit, eine vom Benutzer anzugebende beliebige Anzahl von Punkten auf einer Kugeloberfläche gleichmäßig zu verteilen. Der erste Gedanke ging zu einer Verteilung nach dem Weltkoordinatensystem (4.3.2.1), d.h. an jeder Stelle, an der sich ein Längen- und ein Breitenkreis treffen ist ein Punkt. Diese Möglichkeit hat allerdings einige Probleme. Zum einen sind die Punkte nicht vollkommen gleichmäßig auf der Kugeloberfläche verteilt, da zu den Polen hin die Breitengrade einen immer geringeren Umfang aufweisen, auf dem sich aber immer noch gleich viele Punkte befinden, wie auf dem Äquator. Nachdem diese Möglichkeit trotzdem ausgeschöpft wurde, und die Punkte entsprechend verteilt waren, stellten sich weitere Probleme ein. Zwar gab es eine eindeutige Nachbarschaftsbeziehung zwischen den Punkten, doch erfüllte diese nicht den Anforderungen von Simplex Meshes (siehe 2.1), da in einer dreidimensionalen Simplex Mesh jeder Knoten genau drei Nachbarn besitzt. Bei dieser Verteilung besaß aber jeder vier Nachbarn, wobei das Problem mit den Polen noch gar nicht weiter betrachtet wurde. Somit war es auch nicht möglich den Simplex Winkel (2.2.2) auf die Art und Weise zu bestimmen, wie es die Svoboda und Matula in einer ihrer Arbeiten erwähnen [SM01]. Eine eigene Entwicklung eines Simplex Winkels führte bei der Ausführung des Programms zu keinen befriedigenden Ergebnissen, so dass sich die Möglichkeit dieser Punktverteilung schließlich als unbrauchbar erwies.

Es blieb also nur die Möglichkeit der rekursiven Unterteilung eines Ikosaeders mit anschließender Dualer Graph Transformation (4.3.2.2). Mit dieser Implementation wurden die Ergebnisse dann auch brauchbar, wie sich in einigen Ergebnisbildern in (6.4 und 7) zeigt. Allerdings bestand gerade bei der Präsentation der Ergebnisse ein großes Problem, denn die als Drahtgittermodell vorliegende Mesh sollte als geschlossene Kontur in einem Bild betrachtet werden können. Aber eine Dual Simplex Mesh besteht in diesem Fall aus vielen Fünf- bzw. Sechsecken, deren Fläche nicht einfach ausgemalt werden kann, zumal nicht alle Eckpunkte auf einer Ebene liegen müssen. Erst durch IDGT (siehe 4.3.5.2) ist die gewünschte Ausgabe möglich geworden.

Ein weiteres Problem bei der Implementierung in 3D war die Ausführung des Programms. Da während der Ausführung des Programms das komplette Volumen im Arbeitsspeicher des Computers liegt, war dieser dadurch schon ziemlich stark ausgelastet. Durch weitere große Speicherstrukturen, wie das Gradientenvolumen oder das Gradientenvektorfeld, welches noch eine Dimension größer ist, war der Arbeitsspeicher mehr als überlastet. Die Auslagerung der Daten auf die Festplatte machten das Programm dann so langsam, das es unbrauchbar wurde. Somit stand es im Vordergrund der Optimierung, Speicher einzusparen. Dazu wurde auf das Abspeichern des Gradientenvolumens und des Gradientenvektorfeldes verzichtet, so dass die benötigten Werte jedesmal einzeln für jeden Knoten berechnet werden müssen. Dieses

verlangsamt wiederum die Berechnung der Annäherung der Maschen, führt aber trotzdem zu annehmbaren Ergebnissen (siehe hierzu auch 5.3 und 5.4).

6.3 Anforderungen an die Bild- bzw. Volumendaten

Zuerst sei gesagt, dass alle zur Verarbeitung eingesetzten Bilder bzw. Volumendaten im Graustufenformat vorliegen müssen. Diese Voraussetzung ist sinnvoll, da an keiner Stelle des Verfahrens auf Farbwerte eingegangen wird, sondern nur auf die Intensitäten. Es reichen also Bilder mit 256 Graustufen aus, was auch den Speicheraufwand deutlich reduziert, da nur noch ein Kanal anstatt beispielsweise drei bei RGB pro Bild gespeichert werden muss. Ein Problem ist dies allerdings nicht, denn mit vielen Programmen kann man Bilder entsprechend umwandeln. Volumendaten liegen zumeist sowieso nur in Graustufen abgetastet vor, hier ist eine Umwandlung nicht erforderlich.

Eine zweite Voraussetzung an die Daten ist es, dass die gesuchten Objekte heller sein (eine stärkere Intensität aufweisen) müssen als der Hintergrund. Entsprechend müssen die Bilddaten vorher invertiert werden, falls es nicht der Fall sein sollte. Diese Bedingung hat deswegen eine große Bedeutung, da zum einen beim Finden der Objektmarkierungen (siehe Kapitel 3.4) davon ausgegangen wird, aber auch das Gradientenvektorfeld sonst nicht korrekt ist, und die externe Energie nicht richtig bestimmt werden kann. Abbildung 30 verdeutlicht dies.

Zur automatischen Finden der Objektmarkierungen kann es auch zu einer deutlichen Verbesserung führen, wenn im Voraus der Kontrast des Bildes erhöht wird, da die gleichmäßige Verteilung der Grauwerte eines Bildes über das gesamte Spektrum von 256 Graustufen nicht automatisch erfolgt. Hier liegt natürlich noch eine Erweiterungsmöglichkeit des Verfahrens.

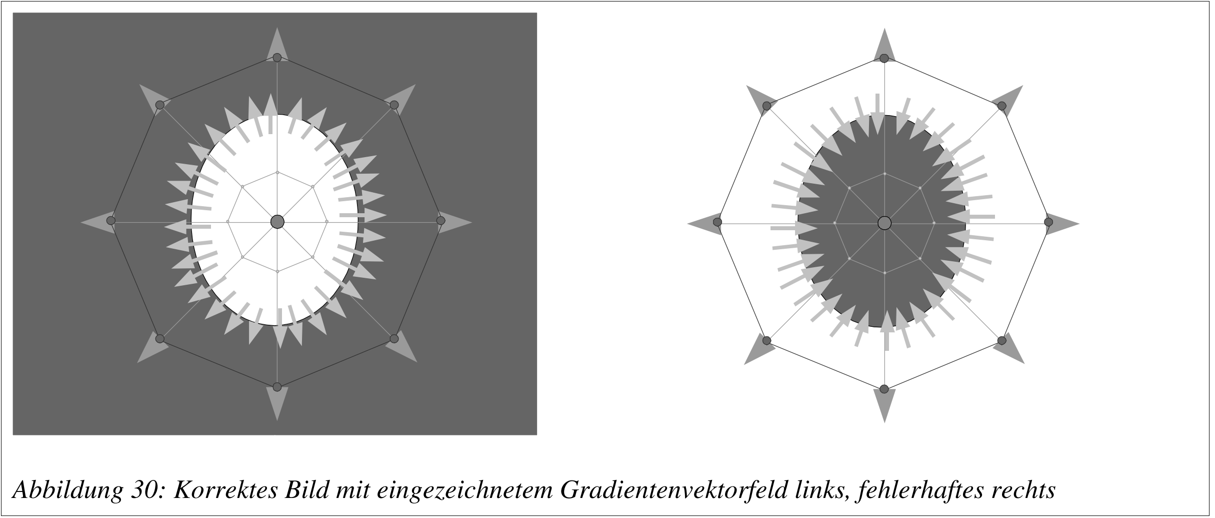


Abbildung 30: Korrektes Bild mit eingezeichnetem Gradientenvektorfeld links, fehlerhaftes rechts

6.4 Auswirkungen durch Veränderung der Parameter in 2D

Dieser Abschnitt zeigt durch seine vielen Beispiele die Verwendungsmöglichkeiten dieser Implementation, und die Einstellung der Parameter. Es ist somit auch eine Hilfe für andere,

ebenfalls das Verfahren zu benutzen, die Ergebnisse zu deuten um dann die Eingaben zu optimieren.

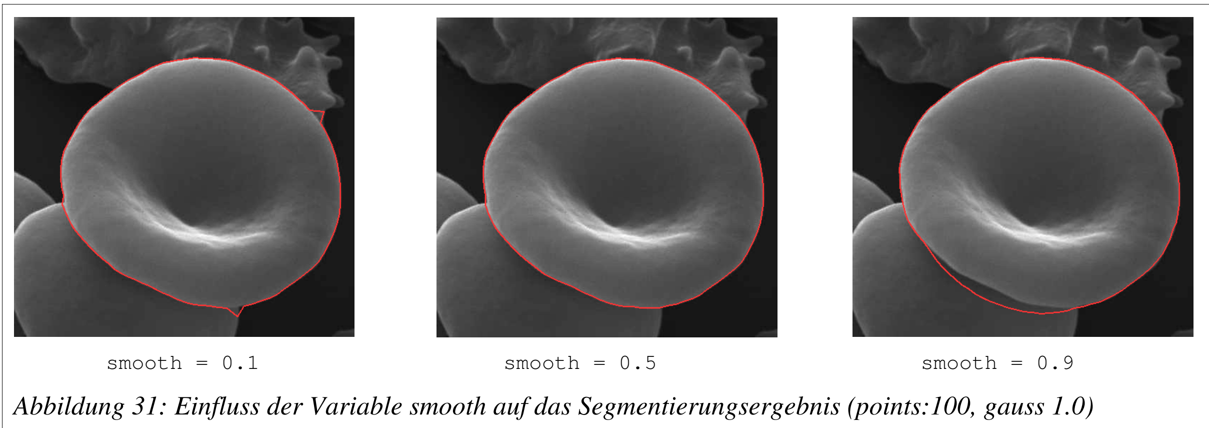
Die vom Benutzer veränderbaren Parameter sind (siehe hierzu auch Kapitel 3.1):

1. `smooth`: Der Faktor, der interne und externe Energie gegeneinander gewichtet.
2. `inner/outerRad`: Die Radien für die Ausgangsmasche, kleinster vorkommender Radius bzw. größter vorkommender Radius eines Objekts.
3. `gauss`: Varianz des Gaußfilters mit dem das Gradientenbild erstellt wird.
4. `points`: Die Anzahl der Knoten pro Masche.

Hierzu jeweils Beispiele:

1. `smooth`:

   Das erste Bild in Abbildung 31 zeigt das Ergebnis, wenn `smooth` zu klein gewählt wird. Das heißt, dass die externe Energie einen zu hohen Stellenwert bekommt. Das dritte Bild zeigt das Ergebnis mit zu hohem Wert der Variablen `smooth`, also bekommt die interne Energie eine zu große Bedeutung zugesprochen. Das mittlere Bild der Reihe zeigt das Ergebnis, nachdem die optimalen Einstellungen getroffen wurden.



smooth = 0.1 smooth = 0.5 smooth = 0.9

Abbildung 31: Einfluss der Variable smooth auf das Segmentierungsergebnis (points:100, gauss 1.0)

2. `outerRadius`:

   Je nach Eingabebild kann es von Bedeutung sein, dass man die Radien für die innere und die äußere Masche möglichst genau angibt. So kann man sich leicht vorstellen, dass es Probleme gibt, wenn man den Radius für die äußere Masche deutlich zu groß angibt. Liegen zwei Objekte nah beisammen, so kann es sein, dass eine Kontur um beide gezeichnet wird, und nicht um jedes einzeln. Aber auch andere Störfaktoren können auftreten. So ist es mit einer möglichst guten Angabe der Radien sehr viel wahrscheinlicher, dass der erste Gradient, auf den eine Masche trifft der richtige ist. Ausserdem werden bei einem zu großen äußeren Radius die Objekte am Rand nicht mehr segmentiert, da die Koordinaten der Schwerpunkte im Allgemeinen näher am Rand liegen,

als ein zu groß gewählter äußerer Radius und somit verworfen werden. Zur Illustration Abbildung 32.

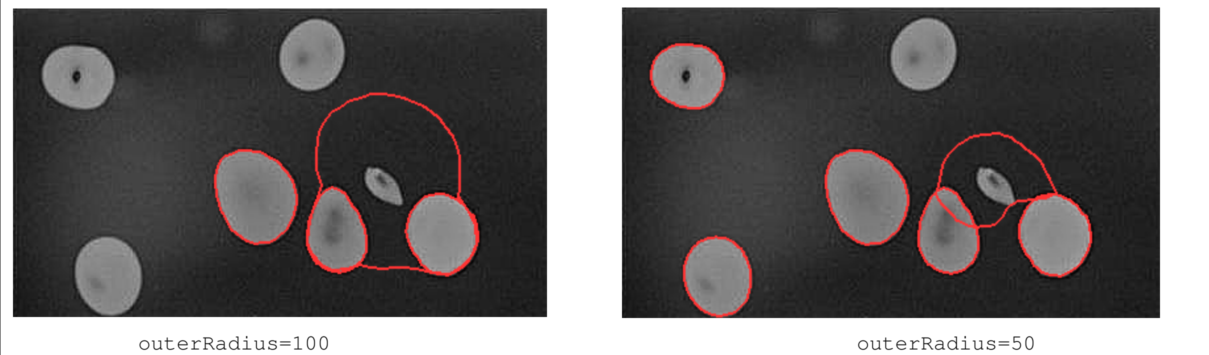


Abbildung 32: Auswirkungen von ungenauen äußeren Maschenradien auf die Segmentierung. Links ein zu hoher äußerer Radius, rechts ein passenderer .

3. `gauss`:

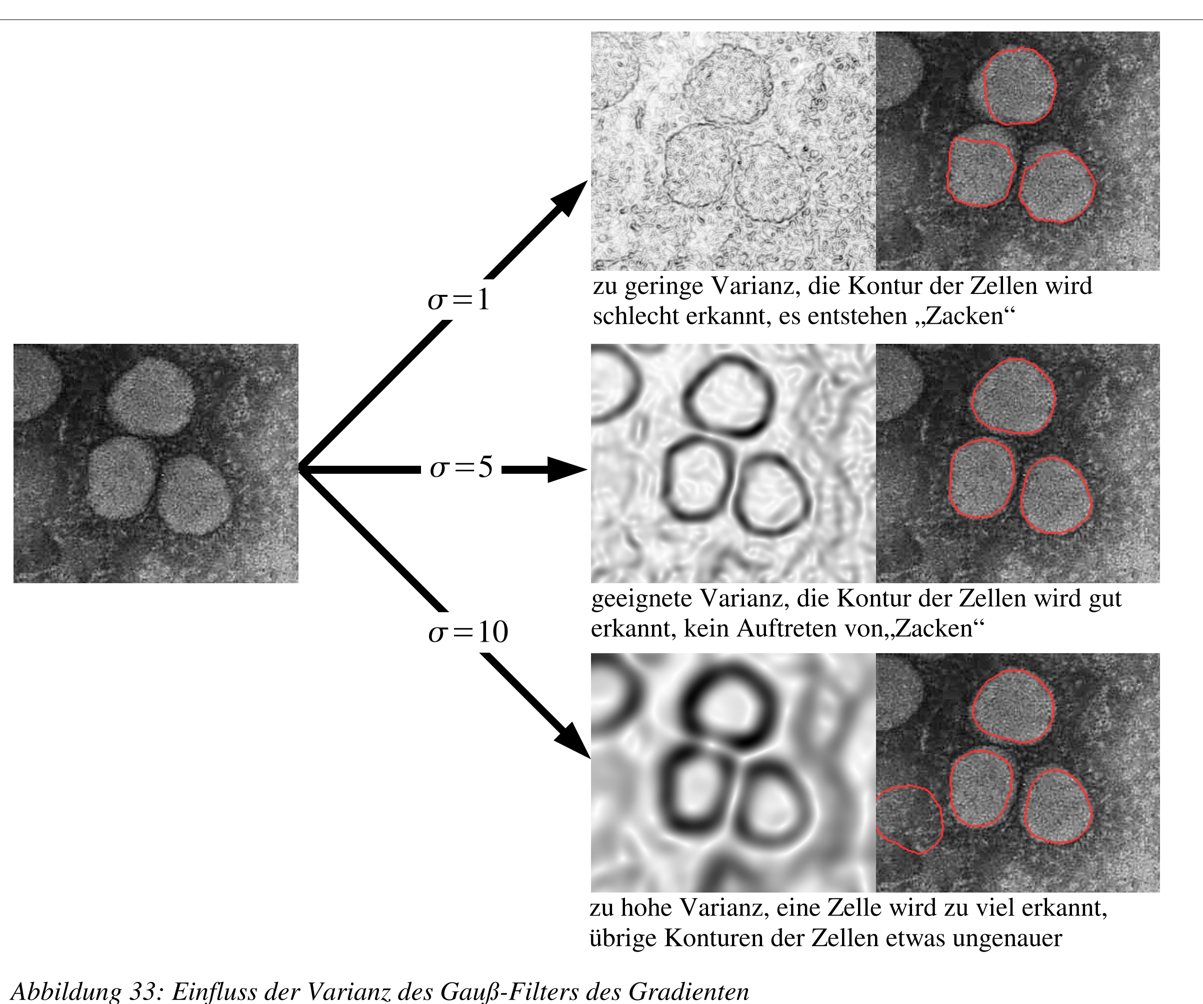
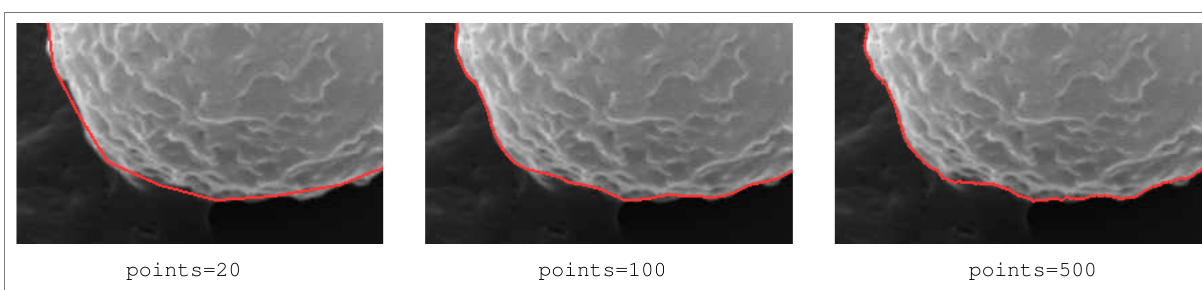


Abbildung 33: Einfluss der Varianz des Gauß-Filters des Gradienten

Hier (Abbildung 33) das stark verrauschte Bild von drei SARS-Viren mit den zugehörigen Gradientenbildern, zuerst mit niedriger Varianz dann mit höherer. Daraus folgend dann die Ergebnisse.

4. `points`:

   Die Anzahl der Knoten ist sehr wichtig für das Erscheinungsbild des Ergebnisses. Wählt man zu wenige Knoten für eine Masche, so wirkt die gefundene Kontur kantig und somit unnatürlich. Wählt man allerdings zu viele, so erhöht sich nicht nur der Rechenaufwand, sondern die Erfahrungen haben gezeigt, dass sich das Ergebnis auch wieder verschlechtern kann. Mehr ist also nicht immer besser, die Anzahl sollte sich dementsprechend an den Größen der gesuchten Objekte orientieren. Abbildung 34 zeigt hierzu ein Beispiel.



points=20 points=100 points=500

Abbildung 34: Qualitätsunterschiede in Abhängigkeit von der Anzahl der Maschenpunkte. Das linke Bild zeigt eine angenäherte Masche mit zu wenigen Punkten, das mittlere eine mit einer guten Punktanzahl und das rechte eine mit einer zu hohen Punktanzahl.

Anhand dieser Beispiele wird klar, dass es keine Standardeinstellungen gibt. Für jedes Bild der Eingabe, manchmal auch für ein Objekt im Bild, muss man die richtigen Einstellungen herausfinden. Das klappt nur, durch wiederholtes Durchführen und Beurteilung der Ergebnisse. Allerdings muss man bei der Veränderung der Parameter sehr vorsichtig sein, denn kleinste Veränderungen können schon enorme Auswirkungen zeigen.

Vor allem der `smooth`-Wert ist relativ schwer genau einzustellen, denn nicht in allen Bereichen des Eingabebildes ist der Gradient gleich stark. So kann es sein, dass in einigen Bereichen die Annäherung der Maschen recht gut gelingt in anderen aber nicht. Hier muss die Grenze herausgefunden werden, bei denen alle Bereiche möglichst gut abschneiden. Ein großer Vorteil hierbei ist allerdings, dass das gesamte Verfahren im Zweidimensionalen recht schnell arbeitet, so dass gute Ergebnisse trotz des Probierens noch relativ schnell gefunden werden können.

6.4 Auswirkungen durch Veränderung der Parameter in 3D

Dieser Abschnitt hat die gleiche Funktion wie das vorherige (6.3). Wieder sind für die Ausführung des Programmes einige Werte vom Benutzer einzustellen. Welche das sind, und welche Auswirkungen unterschiedliche Einstellungen haben, wird an dieser Stelle gezeigt.

Wie der Benutzer die Werte einstellt ist in (4.3.4) beschrieben. Es gibt folgende Parameter:

1. `smooth:` Der Faktor, der interne und externe Energie gegeneinander gewichtet.
2. `inner/outerRad:` Die Radien für die Ausgangsmasche, kleinster vorkommender Radius bzw. größter vorkommender Radius eines Objekts.
3. `points:` Die Anzahl der Knoten pro Masche.

Bei der Angabe der Anzahl der Punkte ist Vorsicht geboten, da nur Werte zulässig sind, die sich durch die rekursive Unterteilung eines Ikosaeders erreichen lassen.

Zu den Auswirkungen auf die Ergebnisse, die durch Veränderungen an den Parametern entstehen, im folgenden wieder einige Beispiele:

1. `smooth:`

   Wie bereits mehrfach beschrieben, gewichtet der `smooth`-Wert die beiden Energien gegeneinander. Je höher dieser Wert eingestellt wird, umso stärker wirkt sich dieses auf die Glattheit der gefundenen Kontur aus. Dagegen hängt die gefundene Kontur bei einem niedrigen Wert fast nur vom Gradienten im Bild ab, was zu einer stark gezackten Kontur führen kann. So sieht man in Abbildung 35 das Ergebnis der verschiedenen `smooth`-Werte. Dabei hängt es auch von den Eingabedaten und der gesuchten Kontur ab, welcher Wert der richtige ist. In diesem Fall trifft der mittlere Wert das beste Ergebnis. Im ersten Bild wird zuviel des gesuchten Objektes abgeschnitten, da die Kontur rund bleiben soll, hingegen beim letzten Bild die Kontur zu leicht an einem Falschen Gradienten hängen bleibt.

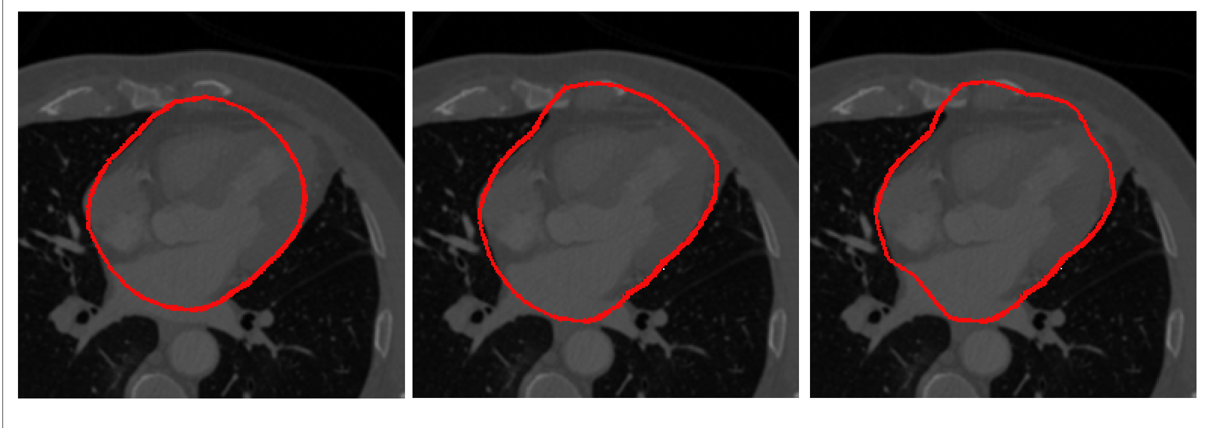


Abbildung 35: `smooth`-Werte von Links nach Rechts: 0.5, 0.2, 0.05.

2. `outerRad:`

   Je nachdem, wie man den äußeren Radius wählt, kann es vorkommen, dass man wichtige Teile des gesuchten Objekts abschneidet, oder auch, dass man umliegende Objekte mit in die Kontur hineinbekommt. Daher ist es wichtig, dass der äußere Radius möglichst exakt angegeben wird, dieses wird auch in Abbildung 36 deutlich. Eine Möglichkeit, diesem Problem zu entgehen, ist es von einer nicht kugelartigen Mesh auszugehen, sondern diese in ihrer Form schon der gesuchten Form anzupassen. Dadurch kann dann auch das Problem umgangen werden, dass an einer Stelle schon das Objekt abgeschnitten wird, während an

einer anderen Stelle zuviel segmentiert wird. Man sollte bei der Wahl des äußeren Radius auch beachten, dass sich der unterschied zwischen Radius der inneren und der äußeren Masche stark auf den Rechenaufwand auswirkt.

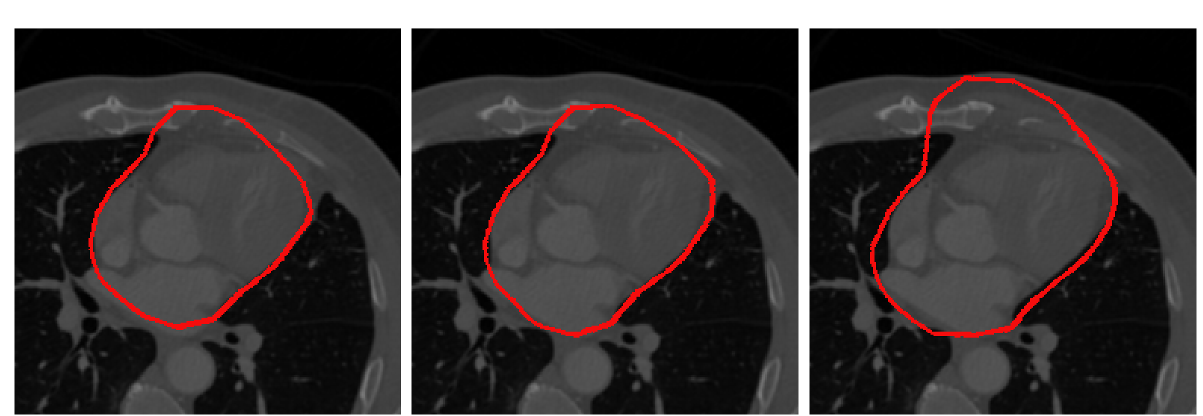


Abbildung 36: Links: outerRad von 85, das Herz wird vor allem oben an der rechten Seite abgeschnitten. Mitte: outerRad von 94, es wird deutlich weniger abgeschnitten. Rechts: outerRad von 98, trotz kleiner Vergrößerung große Auswirkung; Mesh bleibt am falschen Gradienten (oben) hängen.

3. `points`:

   Die folgende zeigt die Auswirkung auf die Wahl der Anzahl der Punkte. Da eine große Anzahl an Punkten auch lange Rechenzeiten bei der Maschenannäherung bedeutet, ist die Wahl auch von diesem Faktor abhängig. Desweiteren verbessert eine weitere Vergrößerung der Anzahl nicht unbedingt das Ergebnis.

Abbildung 37: Links: 80 Punkte führen zu einer kantigen Kontur. Mitte: 320 Punkte führen zu einer guten Kontur. Rechts: 1280 Punkte verbessern die Kontur nicht.

7 Vergleich der Verfahren: 2D vs. 3D

Nachdem das Verfahren der Dual simplex Meshes in den bisherigen Kapiteln sowohl für Dual 1-Simplex Meshes (in 2D) als auch für Dual 2-Simplex Meshes (in 3D) behandelt und untersucht wurde, sollen nun die Ergebnisse beider Verfahren anhand identischer Daten untersucht und verglichen werden.

CT-Bilder liegen in der Regel als eine Vielzahl zweidimensionaler Slices vor, dessen Anzahl von der Abtastrate der Aufnahme abhängt. Bei der Segmentierung von Objekten in diesen Daten kann man somit auch direkt auf diesen Slices arbeiten, das heißt man segmentiert jede Schicht einzeln durch ein 2D-Verfahren. Diese einzelnen Schichten sind in der Regel nicht unabhängig voneinander, da Objekte auf mehreren dieser abgebildet werden. 3D-Verfahren berücksichtigen diese Abhängigkeit zwischen den Schichten.

7.1 Anwendung des 2D-Verfahrens auf Volumendaten

Bei der Anwendung des 2D-Verfahrens auf Volumendaten liegt es nahe, die Schichten so zu verarbeiten, wie sie ein CT-Scanner ausgibt. Allerdings kann dieses dazu führen, dass das Verfahren keine guten Ergebnisse liefert (siehe 7.2), da nicht gewährleistet ist, dass die gesuchten Objekte optimal in dieser Richtung liegen. So kann es vorkommen, dass bei einer Abtastung in eine Richtung x kleinere Differenzen der Objektgrenzen zwischen zwei benachbarten Schichten auftreten, als bei einer Abtastung in eine andere Richtung y. Da es aber beliebig viele Abtastrichtungen durch Drehen des Volumens gibt, ist es kein triviales Problem die optimale zu finden.

Somit scheint es also gleichgültig, ob man das Volumen dreht und dann dieses gedrehte Volumen als Schichten ausgibt, oder man direkt die Schichten des CT-Scanners benutzt. In der Anwendung, wie in diesem Kapitel beschrieben, werden deswegen auch die Schichten in dieser Art benutzt.

Ein weiterer Punkt, den man beachten muss, ist der, dass das 2D-Verfahren große Probleme liefert, sobald mehrere Objekte im Volumen segmentiert werden sollen. Es ist nicht notwendiger Weise der Fall, dass diese Objekte auf allen Schichten annähernd die gleiche Größe haben. Dieses Verfahren kann aber nur mit Objekten gleicher oder ähnlicher Größe auf einem Bild – in diesem Fall eine Schicht – arbeiten. Außerdem kann man sich auch hier vorstellen, dass das oben beschriebene Problem der Abtastung eine Rolle bei der Findung der Objektzentren darstellt.

Sollte man ein Volumen haben, bei dem alle diese Probleme nicht auftreten, so muss man immer noch für jede Schicht die Radien der Dual 1-Simplex Mesh neu wählen, was einen sehr großen Arbeitsaufwand für den Benutzer darstellt.

Es sei an dieser Stelle darauf hingewiesen, dass das 2D-Verfahren auf die 3D-Thematik übertragen keinesfalls eine Kugelform im Dreidimensionalen als Initialisierung lieftert, noch dass die interne Energie eine solche erstrebt. Vielmehr kann man sich hier als Referenzform einen Zylinder im weitesten Sinne vorstellen, der annähernd orthogonal zur Abtastung des Volumens im Volumen liegt. Dabei sei erwähnt, dass die einzelnen Maschen, welche für die Schichten segmentiert wurden, alleine noch keinen Zusammenhang zu darüber- bzw.

darunterliegenden Schichten besitzen. Nach erfolgter Segmentierung müssten diese also noch miteinander korreliert werden, um herauszufinden, welche Objekte zu welchen Maschen auf welchen Ebenen (Slices) des Volumens führen. Dies könnte zum Beispiel durch einen „Nearest Neighbor"-Test erfolgen.

Ebenfalls beachten sollte man, dass es Objekte gibt, die an einigen Stellen eingedellt sind. Bei einem Schnitt durch dieses Objekt, welcher eine Schicht des CT-Scanners repräsentiert, kann es nun vorkommen, dass genau eine solche Delle, also eine konkave Stelle dieses Objektes, auf einer Schicht liegt. Auf dieser Schicht wird das Objekt also durch einen Ring repräsentiert. Selbst wenn dessen äußere Kontur richtig erfasst wird, gehen wichtige Informationen verloren. Beim Betrachten der Konturen sind die im Objekt vorhandenen Dellen nicht mehr vorhanden. Somit handelt es sich an dieser Stelle um keine gefundene Kontur für dieses Objekt.

Zusammengefasst ergeben sich also bei einer Anwendung des 2D Algorithmus` auf Volumendaten folgende Probleme:

- Beste Teilung eines Volumens in Schichten schwer bestimmbar
- Mehrere Objekte zu segmentieren ist nur stark eingeschränkt und mit hohem Aufwand möglich
- Referenzform standardmäßig zylindrisch
- Andere Referenzformen nur mit manueller Veränderung der Eingabeparameter möglich
- Fehlsegmentierung von Objekten mit konkaven Anteilen

Um dennoch einen Vergleich beider Verfahren zu ermöglichen, wurden für die betrachteten Schichten alle Parameter (siehe 4.2.5) per Hand angepasst, so dass ein Vergleich mit der Kugelform des 3D-Verfahrens zumindest in Ansätzen gerechtfertigt ist.

7.2 Vergleich der Ergebnisse

Beide bisher beschriebenen Verfahren werden nun auf ein und dasselbe Volumen angewendet. Dabei wird für die dreidimensionale Bearbeitung wie in (4.3.4) vorgegangen, bei der zweidimensionalen Bearbeitung wie oben (7.1) beschrieben.

Ein erster Vergleich der Ergebnisse kann ein zeitlicher sein. Dabei ist es sinnvoll sich auf den Vergleich des Zeitbedarfs der Maschenannäherung zu konzentrieren. Die Vorverarbeitung der Daten, bzw. das Einlesen, unterscheiden sich in beiden Verfahren zu stark. Auch ist ein automatisches Finden der Anfangsmarkierungen nicht in der dreidimensionalen Implementation enthalten. Beachten muss man allerdings auch, dass für den dreidimensionalen Fall die Berechnung des Gradienten und des Gradientenvektors für einen Voxel während der Annäherung der Maschen geschieht. Trotzdem soll der Zeitbedarf beider Verfahren an dieser Stelle einmal gegenübergestellt werden, dabei werden die Daten des Herzens verwendet.

Für den dreidimensionalen Fall ergibt sich ein Zeitbedarf von ca. 3,5 Sekunden, wie sich auch schon in Kapitel 5.3 zeigte. Die Annäherung im zweidimensionalen Fall hat einen Zeitbedarf

von ca. 0,01 Sekunden pro Schicht. Somit ergibt sich für die ca. 110 relevanten Schichten eine Gesamtzeit von ungefähr 1,1 Sekunden. Der zeitliche Gewinn, der sich durch die Segmentierung mit dem 2D-Verfahren ergibt, ist also um den Faktor drei besser, als bei der Segmentierung mit dem 3D Verfahren. Allerdings ist die zeitliche Dauer nicht das einzige entscheidende Kriterium. Man sollte auch die in (7.1) beschriebenen Probleme nicht außer Acht lassen. Diese Probleme zu beheben, lässt den zeitlichen Vorteil des 2D-Verfahrens geringer werden.

Viel wichtiger sollte das Ergebnis der Segmentierung sein. Beim Vergleich der angenäherten Konturen, die mithilfe des 2D-Verfahrens schichtenweise erzeugt wurden, sollten sich – um ein gutes Segmentierungsergebnis zu erhalten – diese von Schicht zu Schicht nur in sehr geringen Anteilen unterscheiden, um eine möglichst gleichmäßige Oberfläche zu gewährleisten. Dies ist mit dem 3D-Verfahren bereits deswegen gegeben, da hier eine dreidimensionale Nachbarschaftsbeziehung der Knoten bei der Annäherung berücksichtigt wird.

Für den Vergleich der Qualität des Segmentierungsergebnis` wurden die mit dem 2D-Verfahren angenäherten Maschen für drei Schichten ausgegeben und übereinander gelegt, um deren Ähnlichkeit zu prüfen. Für das 3D-Verfahren wurde die angenäherte Masche mithilfe der IDGT (siehe 4.3.5.2) in das Volumen eingezeichnet. Dann wurden die gleichen drei Schichten ausgegeben und ebenfalls übereinander gelegt. Hierbei sollte sich zeigen, ob das 3D-Verfahren mithilfe der erweiterten Nachbarschaftsbeziehung gegenüber dem 2D-Verfahren Vorteile mit sich bringt.

In Abbildung 38 erkennt man einige wichtige Unterschiede in den Ergebnissen der Segmentierung. Sehen die Ergebnisse, die das 3D-Verfahren liefert, für alle drei benachbarten Schichten sehr ähnlich aus, so zeigen sich bei der Anwendung des 2D-Verfahrens auf die gleichen Bilddaten doch erhebliche Unterschiede. Dies wird vor allem bei der überlappten Darstellung deutlich.

Aus diesen überlappt dargestellten Schichten setzt sich aber das spätere Segmentierungsergebnis, nämlich eine Volumenbeschreibung des Objektes, zusammen. Damit dies zum gewünschten Ergebnis führt, sollte in jeder Ebene nur das relevante Objekt segmentiert werden. Dies ist nach augenscheinlichem Betrachten der Abbildung 38 nicht der Fall. Teilweise bleiben die Maschen des 2D-Verfahrens an Bestandteilen von Knochen hängen, teilweise wird eine Arterie in die erkannte Kontur des Objektes mit aufgenommen. Dieses Problem liegt aber in der Vorgehensweise des 2D-Verfahrens begründet. Ohne Information über Schichten übergreifende Nachbarschaften wird es sich nicht beheben lassen.

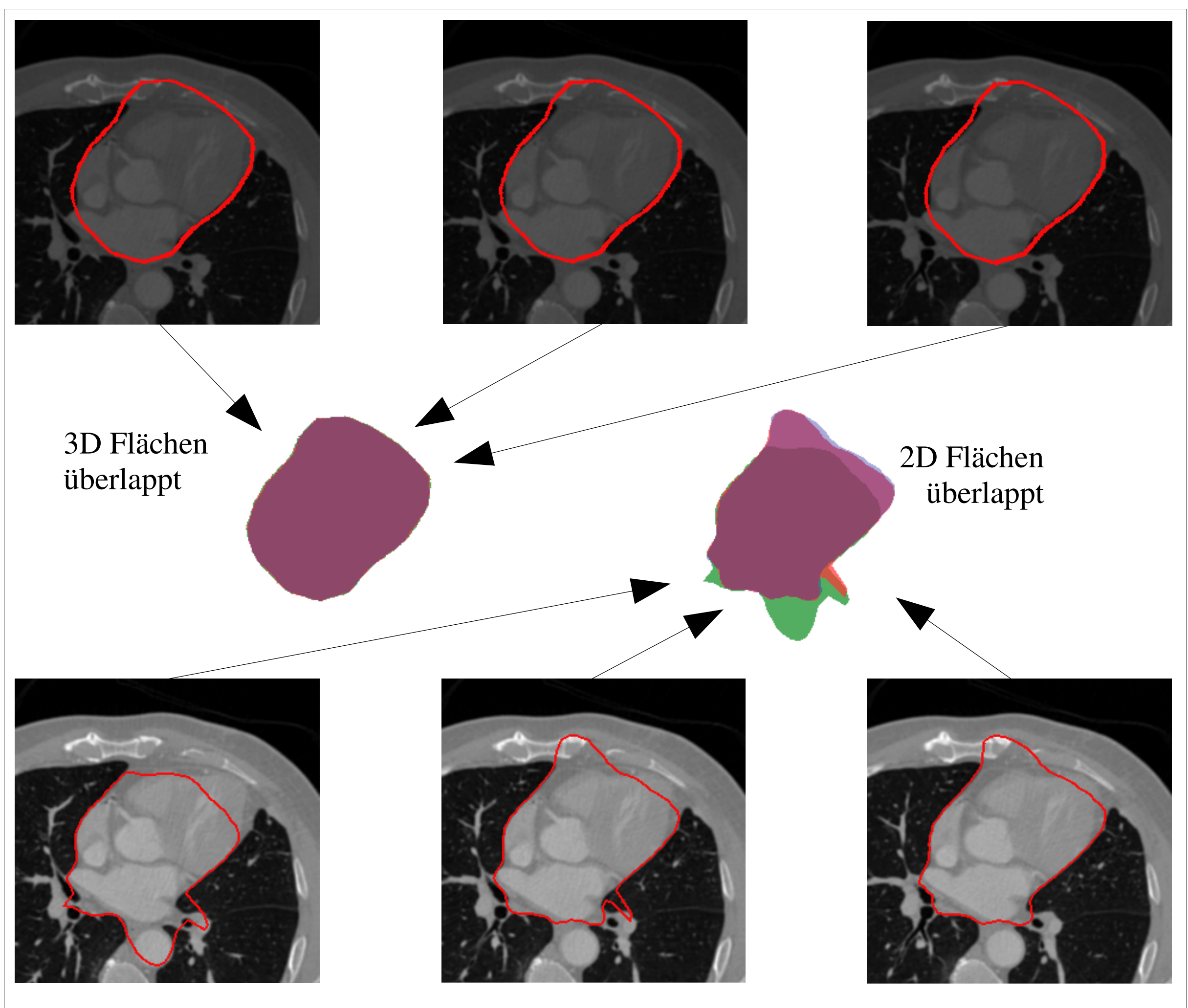


Abbildung 38: Vergleich des 2D- und des 3D-Verfahrens anhand dreier übereinanderliegender Slices desselben Volumens. Ergebnisse: Oben: 3D-Verfahren. Unten: 2D-Verfahren. Mitte: Überdeckungen der segmentierten Flächen.

8 Zusammenfassung und Ausblick

In den vorherigen Kapiteln wurde zuerst eine Einführung in die Segmentierung gegeben. Dabei wurde zwischen elementaren und modellbasierten Verfahren unterschieden. Das dann eingeführte Verfahren der Dual Simplex Meshes gehört zu den modellbasierten und stellt eine Form der Active Contours dar.

Nach den ersten Grundlagen wurden im darauf folgenden Kapitel die formalen Grundlagen einer Dual Simplex Mesh und deren Verformung erläutert. Anschließend wurde auf den Algorithmus eingegangen, der die Segmentierung ausführt. Dieser Algorithmus wurde dann in zwei Implementationen verwirklicht. Die erste arbeitete nur auf zweidimensionalen Daten, sprich Bildern, während die zweite auf dreidimensionalen Daten, sprich Volumen, arbeitete. Dabei wurden nicht nur vom Vorlagenpaper beschriebene Vorgehensweisen verwendet, sondern auch einige eigenständig erarbeitet. Nach der Ausführung der Implementationen wurden deren Leistungen gemessen und diskutiert.

Im Anschluss daran wurden die Erfahrungen und Probleme, die das implementierte Verfahren bereitete, erläutert. Dazu gehörte eine intensive Diskussion der Auswirkungen, die durch unterschiedliche Einstellungen der Parameter entstanden.

Zum Ende wurden beide implementierten Verfahren miteinander verglichen. Dabei stellte sich heraus, dass das dreidimensionale Verfahren auf dreidimensionalen Daten besser abschneidet als das zweidimensionale. Allerdings bieten beide vorgestellten Verfahren noch Verbesserungsmöglichkeiten.

Für die Zukunft sind noch einige Erweiterungen und Optimierungen denkbar. So kann man die Daten vor der Segmentierung in geeigneter Weise vorverarbeiten. Auch ist eine Geschwindigkeitsoptimierung möglich.

Desweiteren ist es zur Zeit recht umständlich dieses Verfahren zu benutzen. Eine grafische Benutzungsoberfläche würde eine sinnvolle Erweiterung darstellen. Durch die Interaktivität hätte der Benutzer die Möglichkeit die Eingabeparameter schneller optimal zu bestimmen. Auch könnte der Algorithmus in bestehende Systeme eingebunden werden.

Abschließend lässt sich sagen, dass eine Segmentierung von medizinischen Daten mit diesem Verfahren teils schon recht gute Ergebnisse liefert und ein weiterer Ausbau daher sehr sinnvoll erscheint.

A Verwendete Bilddaten

Bild	*Quelle (abgerufen am 19.01.2005)*
	http://www.jostjahn.de/sarsbig.jpg
	http://www.zoologie.sbg.ac.at/struktur/abteilung/elektronenmikroskopie/rem99/blut-1-t.jpg
	http://www.zoologie.sbg.ac.at/struktur/abteilung/elektronenmikroskopie/rem99/blut-3-t.jpg
	http://www.edu-graphics.com/marcia/images/Albums/Anatomy/Blood%20Slides/thumbs/Blood_01%20011.jpg
	http://www.medpol.de/fall2/unters2/2blut.jpg

Bild	***Quelle (abgerufen am 19.01.2005)***
	Volumendatensatz: s6270r2.ids Bereitgestellt durch: Firma Philips

B Inhalt der CD

Die zu dieser Arbeit gehörende CD enthält:

- Ordner: `Dual Simplex Meshes 2D`

  Die Implementation des Algorithmus` für den zweidimensionalen Fall inklusive aller Quelldateien als Visual Studio .net© Projekt

- Ordner: `Doku 2D`

  Die Dokumentation zur 2D Implementation des Verfahrens

- Ordner: `Dual Simplex Meshes 3D`

  Die Implementation des Algorithmus` für den dreidimensionalen Fall inklusive aller Quelldateien als Visual Studio .net© Projekt

- Ordner: `Doku 3D`

  Die Dokumentation zur 3D Implementation des Verfahrens

- Datei: `Baccalaureatsarbeit DSM.pdf`

  Diese Arbeit in elektronischer Form.

C Aufteilung der Ausarbeitung zwischen den Studierenden

Kapitel	*Unterkapitel*	*Autor*
1	alle	Florian Heinrich
2	(2.1) bis (2.4)	Florian Heinrich
	(2.5)	Benjamin Seppke
3	alle	Gemeinschaftsarbeit
4	(4.1) bis (4.2.2)	Benjamin Seppke
	(4.2.3) bis (4.2.5)	Florian Heinrich
	(4.2.6) bis (4.3.2)	Benjamin Seppke
	(4.3.3.1)	Florian Heinrich
	(4.3.3.2)	Benjamin Seppke
	(4.3.4)	Florian Heinrich
	(4.3.5)	Benjamin Seppke
5	alle	Benjamin Seppke
6	alle	Florian Heinrich
7	alle	Gemeinschaftsarbeit

Abbildungsverzeichnis

Abbildung 1: Anschauliche Darstellung eines Schwellwertverfahrens....................................2
Abbildung 2: Veranschaulichung des Wasserscheidenverfahrens anhand eines Querschnittes durch das Graustufengebirge eines Bildes..................................3
Abbildung 3: Links: Star Shaped Simplex Mesh(2D), rechts: Dual Simplex Mesh (2D)........5
Abbildung 4: Links ein Gradientenbild, rechts ein Gradientenvektorfeld in 2D......................7
Abbildung 5: Zusammenwirken der beiden Energien auf eine Dual Simplex Mesh in 2D …… (linkes Bild). Zweites Bild: Volle Gewichtung auf die externe Energie. .... Weitere Bilder: Zunahme des Einflusses der internen Energie und .... entsprechend Abnahme des Einflusses der externen Energie.............................8
Abbildung 6: Einige p-Elemente im Überblick (von links nach rechts: 0-Element, 1-Element, 2-Element, 3-Element)...10
Abbildung 7: Beispiele für das Zentrum O einer 1-Simplex Mesh (links) und einer 2-Simplex Mesh (rechts)...11
Abbildung 8: Veranschaulichung der Steigungs-Winkelberechnungsmethode......................13
Abbildung 9: Veranschaulichung der Vektoren-Winkelberechnungsmethode......................14
Abbildung 10: Links: konvexer Winkel; Rechts: konkaver Winkel..14
Abbildung 11: Bedeutung des Simplex Winkels . Links: Umkugel von vier Punkten. Rechts: Projektion der linken Abbildung. 15
Abbildung 12: Beispielhafte Energieverteilungen einer Dual Simplex Mesh..........................19
Abbildung 13: Transformation von Geraden in den Parameterraum.......................................21
Abbildung 14: Beispielhafter Ablauf des Algorithmus` in 2D nach Svoboda und Matula. Von links nach rechts: 1. Ausgangsbild, 2. Gefundene Maschenzentren, 3. Endergebnis, nach der Verformung der Maschen.......................................25
Abbildung 15: UML-Klassendiagramm der Dual Simplex Mesh Implementation in 2D........31
Abbildung 16: Beispiel der Verteilung der Maschenpunkte am Beispiel einer Masche mit 12 Punkten...32
Abbildung 17: Semantik der Zeilen und Spalten des Arrays der Daten der Maschenpunkte...33
Abbildung 18: Von links nach rechts: 1. Ausschnitt eines Bildes mit zwei Blutzellen. 2. Gefundene Labels. 3. Aus den Labels ermittelte Zentren. 4. Ermittelte Objektzentren durch Verschmelzung der Schwerpunkte............37
Abbildung 19: Ein typischer Durchlauf des Algorithmus zum Finden der Objektmarkierungen...38
Abbildung 20: UML Klassendiagramm der Dual Simplex Meshes Implementation in 3D.....45
Abbildung 21: Die Unterteilung eines Ikosaeders, von links nach rechts: Ikosaeder, nach erster Unterteilung, nach zweiter Unterteilung...47
Abbildung 22: Eine durch DGT aus zwei rekursiv unterteilten Ikosaedern erzeugte Dual Simplex Mesh 49
Abbildung 23: 6er Nachbarschaft eines Voxels. Rot: Nachbarn in x-Richtung. Violett: Nachbarn in y-Richtung. Grün: Nachbarn in z-Richtung. 52
Abbildung 24: Schematische Darstellung zur Erzeugung und Ausgabe einer Dual Simplex Mesh..56
Abbildung 25: Rückgewinnung einer Maschentriangulierung durch die IDGT......................57
Abbildung 26: Links: Schematische Veranschaulichung des Dreieckfüllverfahrens in 3D. Rechts: Eine Masche, die mit der IDGT trianguliert wurde, nach Anwendung des Dreieckfüllverfahrens..58

Abbildung 27: Bilddaten und Messergebnisse des ersten Bildes....62
Abbildung 28: Bilddaten und Messergebnisse des zweiten Bildes....63
Abbildung 29: Messergebnisse der Segmentierung des Volumens....66
Abbildung 30: Korrektes Bild mit eingezeichnetem Gradientenvektorfeld links, fehlerhaftes rechts....72
Abbildung 31: Einfluss der Variable smooth auf das Segmentierungsergebnis (points:100, gauss 1.0)....73
Abbildung 32: Auswirkungen von ungenauen äußeren Maschenradien auf die Segmentierung. Links ein zu hoher äußerer Radius, rechts ein passenderer ...74
Abbildung 33: Einfluss der Varianz des Gauß-Filters des Gradienten....74
Abbildung 34: Qualitätsunterschiede in Abhängigkeit von der Anzahl der Maschenpunkte. Das linke Bild zeigt eine angenäherte Masche mit zu wenigen Punkten, das mittlere eine mit einer guten Punktanzahl und das rechte eine mit einer zu hohen Punktanzahl. 75
Abbildung 35: smooth-Werte von Links nach Rechts: 0.5, 0.2, 0.05....76
Abbildung 36: Links: outerRad von 85, das Herz wird vor allem oben an der rechten Seite abgeschnitten. Mitte: outerRad von 94, es wird deutlich weniger abgeschnitten. Rechts: outerRad von 98, trotz kleiner Vergrößerung große Auswirkung; Mesh bleibt am falschen Gradienten (oben) hängen....77
Abbildung 37: Links: 80 Punkte führen zu einer kantigen Kontur.
Mitte: 320 Punkte führen zu einer guten Kontur.
Rechts: 1280 Punkte verbessern die Kontur nicht....77
Abbildung 38: Vergleich des 2D- und des 3D-Verfahrens anhand dreier übereinanderliegender Slices desselben Volumens. Ergebnisse:
Oben: 3D-Verfahren. Unten: 2D-Verfahren. Mitte: Überdeckungen der segmentierten Flächen....81

Literaturverzeichnis

[Bou02] Paul Bourke, Equation of a Sphere from 4 Points on the Surface, http://astronomy.swin.edu.au/~pbourke/geometry/spherefrom4/, abgerufen am 3.11.2004

[Del94] Hervé Delingette. General Object Reconstruction based on Simplex Mashes, in *International Journal of Computer Vision*, 32, pp. 111-142, 1999, Kluwer Academic Publishers

[Gum04] Stefan Gumhold, Vorlesungsfolien zu „Einführung in die Topologie und Verbundenheit von Flächen“, http://www.mpi-sb.mpg.de/~sgumhold/surf_tc.pdf, abgerufen am 27.12.2004

[Jäh02] Bernd Jähne. *Digitale Bildverarbeitung*, 5., überarbeitete und erweiterte Auflage, 2002, Springer-Verlag Berlin Heidelberg New York

[Köt04] Ullrich Köthe. VIGRA Reference Manual, verfügbar unter http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/vigra/index.html, abgerufen am 22.12.2004

[SM01] P. Matula, D. Svoboda. Spherical Object Reconstruction Using Star-Shaped Simplex Meshes, in Figueiredo M.A.T., Zerubia J., Jain A.K. (Eds.): EMMCVPR 2001, LNCS 2134, pp. 608-620, 2001, Springer-Verlag

[SM03a] D. Svoboda, P. Matula. Tissue Reconstruction Based on Deformation of Dual Simplex Meshes, in Nyström I. et al. (Eds.): DGCI 2003, LNCS 2886, pp. 514-523, 2003, Springer-Verlag

[SM03b] D. Svoboda, P. Matula. Multilevel Adaptive Thresholding and Dual Simplex Mesh Deformation as a Tool for Tissue Reconstruction, in *Visualization, Imaging, and Image Processing*, pp. 499-503, 2003, ACTA Press

[Wik04] Wikipedia, die freie Enzyklopädie: http://de.wikipedia.org/wiki/, abgerufen am 29.11.2004

[Wol04] Wolfram Research (Mathworld): Tessellation, http://mathworld.wolfram.com/Tessellation.html, abgerufen am 19.12.2004