Erzeugung von Tetraedergittern für die Volumendeformation

Diplomarbeit

im Fach Angewandte Informatik

vorgelegt von

Björn Labitzke

Geboren am 13. März, 1980 in Gummersbach

Angefertigt am

Lehrstuhl für Computergraphik und Multimediasysteme
Fachbereich 12
Universität Siegen

Betreuer:

Dr. C. Rezk-Salama, Lehrstuhl Computergraphik und Multimediasysteme, Universität Siegen

Prof. Dr. A. Kolb, Lehrstuhl Computergraphik und Multimediasysteme, Universität Siegen

Beginn der Arbeit: 01. August 2008

Abgabe der Arbeit: 30. Januar 2009

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Köln, den 30. Januar 2009 _____

Björn Labitzke

Übersicht

Die vorliegende Arbeit zeigt einen Ansatz zur Erzeugung von Tetraedergittern für die Volumendeformation. Der Schwerpunkt der Erzeugung basiert auf einem Shrink-Wrapping-Verfahren. Hierbei wird eine initiale, grobe Dreiecksoberfläche sukzessiv an die entsprechenden Volumendaten angepasst, indem diese Oberfläche mehr und mehr geschrumpft wird. Die Konstruktion des initialen Dreiecksnetzes erfolgt nach dem Prinzip der *Surface from Contours*. Dementsprechend werden zu dem Volumen zunächst mehrere planare, aktive Konturen bestimmt, die im Anschluss durch einen Triangulierungsschritt verbunden werden. Das Shrink-Wrapping-Verfahren wird mit einem Optimierungsschritt kombiniert, der zum einen degenerierte Dreiecke entfernt und zum anderen gezielt lokale Details zur Oberfläche hinzufügt. Hierdurch werden die Stabilität des Netzes und das Potential zur Anpassung erheblich verbessert. Zum Schluss wird die ermittelte Dreiecksoberfläche an einen automatischen 3D Finite Elemente Netzgenerator (*Gmsh* [GR08]) übergeben, um die abschließende Generierung des Tetraedergitters durchzuführen.

Abstract

This thesis presents an approach to generate tetrahedral meshes for volume deformation. The main focus of the generation is based on a shrink wrapping process. This process produces the final surface by iteratively shrinking an initial and coarse triangle surface. The coarse mesh is constructed according to the principle of the *Surface from Contours*. Accordingly, at first several planar active contours are determined for the volume and are then connected by a triangulation step. The shrink wrapping procedure is combined with an optimization step, which on the one hand removes degenerated triangles and on the other hand purposefully adds local details to the surface. The stability of the mesh and the potential for the adjustment will be strongly improved by this optimization. At the end the produced triangle surface will be passed to an automatic 3D finite element grid generator (*Gmsh* [GR08]) in order to generate the tetrahedral mesh.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben.

Ein ganz besonderer Dank gilt meinem Betreuer Dr. Christof Rezk-Salama, der stets Zeit für mich hatte. Vielen Dank für die gute Betreuung und die Gespräche, die mir viele wertvolle Hinweise sowie Denkanstöße gaben und nicht zuletzt immerzu sehr motivierend waren.

Des Weiteren möchte ich mich für das Korrekturlesen und die konstruktive Kritik bei Andreas, Dennis, Marcel, Marius und meiner Schwester Nicole bedanken. Vielen Dank für Eure Zeit und Euer Interesse.

Mein ganz besonderer Dank gebührt meiner Freundin Tanja, die während der letzten Monate auf viel gemeinsame Zeit verzichten musste. Sie stand mir stets mit ihrer Liebe, ihrem Vertrauen und ihrer bedingungslosen Unterstützung zur Seite. Dafür danke ich Dir in aller Liebe.

Nicht zuletzt möchte ich mich bei meinen Eltern bedanken, die mir durch ihre fortwährende Unterstützung und ihr Vertrauen diesen Weg ermöglichten. Mit Eurer Liebe wart Ihr immer für mich da. Dafür danke ich Euch aus tiefsten Herzen.

Björn Labitzke, Januar 2009

Inhaltsverzeichnis

Verzeichnis der Bilder

Verzeichnis der Tabellen

Listings

Abkürzungsverzeichnis

AABB	Axis-Aligned Bounding Box
CPU	Central Processing Unit
DOC	Degree of Correspondence
DT	Distance Transformation / Distanztransformation
FBO	Framebuffer Objekt
GPU	Graphics Processing Unit
JFA	Jump Flooding Algorithmus
LOD	Level of Detail
RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
SDT	Signed Distance Transformation

Kapitel 1

Einleitung

Bildgebende Verfahren, wie die Computer- oder Magnetresonanztomographie, ermöglichen Schnittbilder von Körpern, die vornehmlich in der medizinischen Diagnostik und Therapieplanung eingesetzt werden. Die modernen Techniken der Volumenvisualisierung machen es möglich aus diesen Schnittbildern hochqualitative und dreidimensionale Bilder zu konstruieren. Diese anschauliche Art der Visualisierung ist intuitiv verständlich, gibt einen guten Überblick über räumliche Verhältnisse und bietet sehr gute Möglichkeiten zur interaktiven Exploration. Gerade Letzteres wird dadurch erreicht, dass die Darstellung, durch die sinnvolle Verwendung von Transferfunktionen, auf relevante Bereiche eingeschränkt werden kann.

Mit dem Ziel beispielsweise einen chirurgischen Eingriff besser vorbereiten zu können, wird vermehrt auch die interaktive Deformation statischer Volumendaten erforscht. Ein vielversprechender Ansatz ist hierbei die Deformation nicht direkt auf dem Datensatz durchzuführen, sondern auf einer Proxygeometrie. Diese Hilfsgeometrie, beispielsweise in Form eines Tetraedergitters, wird zur Diskretisierung des Datenraums genutzt und wäre optimal, wenn sie den segmentierten Bereich des Datensatzes möglichst gut beschreiben würde.

Die vorliegende Arbeit widmet sich der Erzeugung solcher Proxygeometrien. Die Anforderung, die dabei an die Geometrie gestellt wird, richtet sich primär darauf, dass der segmentierte Bereich des Datensatzes komplett umhüllt, aber nicht bis in kleinste Detail aufgelöst wird, da die Darstellung der Details Teil der eigentlichen Visualisierung ist. Der in dieser Arbeit beschriebene Ansatz wird daher im Wesentlichen den relevanten Bereich des Datensatzes durch eine umhüllende und geschlossene Oberfläche approximieren. Die erzeugte Approximation wird dann die Struktur des Datensatzes möglichst gut beschreiben, so dass auf deren Basis letztlich die dreidimensionale Hilfsgeometrie, in Form eines Tetraedergitters, generiert werden kann.

Zur Erklärung des Verfahrens werden zunächst die Grundlagen für das Verständnis dieser Arbeit dargestellt. In Form eines Überblicks wird dazu primär das theoretische Grundwissen aufgeführt und erläutert (Kap. 2).

Im darauf folgenden Kapitel 3 wird das theoretische Konzept dieses Ansatzes beschrieben. Dazu wird zunächst eine Abgrenzung des Kontextes und eine Beschreibung der Zielsetzung erfolgen.

Bevor die ausführliche und schrittweise Erklärung des Konzepts erfolgt, wird zuvor ein einführender Überblick gegeben.

Nach der theoretischen Betrachtung des Verfahrens werden im vierten Kapitel die wesentlichen Aspekte der Implementierung aufgegriffen und erörtert.

Das an die Betrachtung der praktischen Aspekte anschließende Kapitel 5 widmet sich dann der Präsentation der Ergebnisse.

Zum Abschluss der Arbeit werden im sechsten Kapitel die Ziele und wichtigsten Erkenntnisse zusammengefasst. Zudem wird eine kritische Betrachtung des Verfahrens erfolgen und ein Ausblick auf zukünftige Erweiterungsmöglichkeiten gegeben.

Kapitel 2

Grundlagen

Das Konzept des in der vorliegenden Arbeit entwickelten Ansatzes basiert auf einem Shrink-Wrapping-Verfahren, bei dem grobe Dreiecksoberflächen sukzessiv an ein Zielobjekt angepasst werden. Dieses Kapitel beschreibt dementsprechend die für das Verständnis dieser Arbeit vorausgesetzten theoretischen Grundlagen. Anfangs werden hierbei Dreiecksnetze (2.1) in einem groben Abriss dargestellt, bevor im Anschluss ein knapper Überblick über die Oberflächenrekonstruktion (2.2) verschafft wird. Der nachfolgende Abschnitt 2.3 widmet sich der kurzen Erläuterung der Distanzfelder, die innerhalb dieser Arbeit als Basis für die Anpassung verwendet werden. Abschließend beleuchtet Kapitel (2.4) das Shrink-Wrapping-Verfahren selbst und schließt somit die Vorstellung der Grundlagen ab.

2.1 Dreiecksnetze

Ein Dreiecksnetz ist ein Spezialfall eines Polygonnetzes. Daher soll zunächst, in knapper Form, der Begriff des Polygonnetzes näher beschrieben werden. Polygonnetze bestehen aus mehreren planaren Polygonen, die auch als Vielecke oder häufig auch als Flächen (engl. „Faces“) bezeichnet werden. Ein Polygon setzt sich zusammen aus mindestens drei verschiedenen Punkten (engl. „Vertices“), die durch Strecken/Kanten (engl. „Edges“) kreuzungsfrei miteinander verbunden sind. Somit umfasst ein Polygonnetz je eine Menge von Punkten, Kanten und Flächen die wie folgt definiert sind:

- Menge von Punkten: $\mathcal{V} = \{\mathbf{V}_i\}, \quad i \in \{1, \ldots, N_V\}, N_V = |\mathcal{V}|$
- Menge von Kanten: $\mathcal{E} = \{\mathbf{E}_{ij}\}, \mathbf{E}_{ij} = \overline{\mathbf{V}_i\mathbf{V}_j}, \quad i, j \in \{1, \ldots, N_V\}, N_E = |\mathcal{E}|$
- Menge von Flächen: $\mathcal{F} = \{\mathbf{F}_i\}, \quad i \in \{1, \ldots, N_F\}, N_F = |\mathcal{F}|$

Die Polygone eines Netzes müssen geschlossen sein und sind derart verbunden, dass jede Kante zwei Punkte verbindet, ein Punkt Bestandteil von mindestens zwei Kanten ist und dass benachbarte Flächen mindestens eine gemeinsame Kante besitzen [FvDFH96].

Bei einem geschlossenen Polygon sind Start- und Endpunkt identisch, wie zum Beispiel bei Dreiecken, Vierecken und einfachen konvexen Polygonen, wobei Dreiecke die einfachste Form darstellen.

Wenn ein Polygonnetz ausschließlich aus Dreiecken besteht, dann handelt es sich um den, eingangs erwähnten, Spezialfall des Dreiecksnetzes. Die Form des Dreiecksnetzes wird sehr häufig verwendet, da zum einen die Berechnungen im Dreieck (vgl. A.1) unkompliziert sind und zum anderen eine sehr effiziente Verarbeitung der Dreiecke auf der programmierbaren Grafikhardware (engl. „Graphics Processing Unit“ - GPU) erfolgt. Es gibt zahlreiche und gute Argumente, die für die Verwendung dieser Netzform sprechen. Deren Aufführung und Erörterung soll jedoch nicht Teil dieser Arbeit sein. Vielmehr soll es im Folgenden um die weitere Definition von grundlegenden Begriffen und die Beschreibung einer Datenstruktur zur effizienten Repräsentation der Netzstruktur gehen.

2.1.1 Grundlegende Definitionen

In diesem Abschnitt soll eine Auswahl verschiedener Begriffe, die innerhalb dieser Arbeit von Relevanz sind, beschrieben werden.

Topologie und Geometrie Die Oberflächendarstellung eines Körpers trägt *topologische* und *geometrische Informationen*. Während die Topologie die Nachbarschaftsbeziehungen zwischen Punkten, Kanten und Flächen beschreibt, speichert die Geometrie zum Beispiel die Lage der Punkte (Punktkoordinaten). Daraus ergibt sich, dass Oberflächen mit identischer Topologie, aufgrund unterschiedlicher Geometriedaten, verschiedene Formen aufweisen können. [BGZ02]

Adjazenz Der Begriff der Adjazenz stammt aus der Graphentheorie und beschreibt das Angrenzen von gleichen Elementen. Zwei Kanten werden als *adjazent* bezeichnet, wenn sie einen gemeinsamen Eckpunkt besitzen. Aber auch die Punkte $\mathbf{V}_i$ und $\mathbf{V}_j$ sind angrenzend, wenn sie durch die Kante $\overline{\mathbf{V}_i\mathbf{V}_j}$ miteinander verbunden sind. Gleiches gilt für Flächen, die eine gemeinsame Kante besitzen.

Valenz Die Valenz gibt die Anzahl der Kanten an, die an einen Punkt angrenzen (siehe Abb.2.1).

1-Nachbarschaft Die 1-Nachbarschaft eines Punktes sind alle direkt angrenzenden Kanten, Punkte und Polygone. Neben der ersten Nachbarschaft sind auch *n*-Nachbarschaften möglich. Hierbei ist dann die 1-Nachbarschaft der (*n*-1)-Nachbarschaft gemeint (vgl. Abb. 2.1).

Genus Der Genus eines Polygonnetzes beschreibt die Anzahl der Durchgangslöcher im Netz.

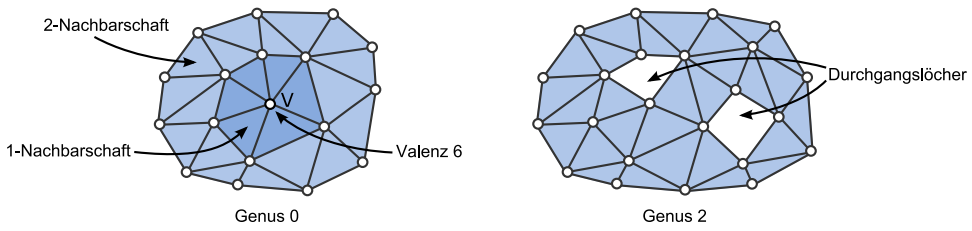


Abbildung 2.1: 1- und 2-Nachbarschaft eines Punktes V und ein Dreiecksnetz mit Genus zwei.

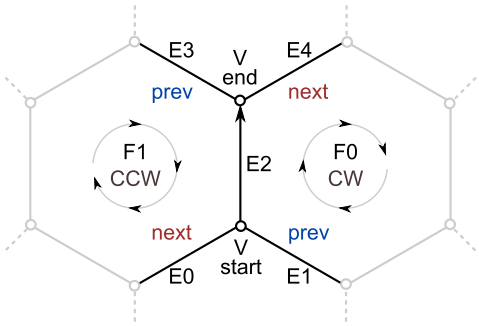
2.1.2 Winged-Edge Datenstruktur

Für die Repräsentation von Polygonnetzen gibt es viele Möglichkeiten, die jeweils Vor- und Nachteile bezüglich der Speichernutzung und der Laufzeit von Operationen haben. Einfache Datenstrukturen weisen meist schlechte Laufzeiten bei den Operationen auf, wie zum Beispiel bei der Suche nach den zwei adjazenten Flächen einer Kante. Zur Verbesserung der Laufzeiten wurden komplexere Repräsentationen entwickelt, von denen die Winged-Edge Datenstruktur von Baumgart eine der gängigsten Strukturen darstellt [FvDFH96]. Diese Datenstruktur wird auch in der vorliegenden Arbeit verwendet und soll daher im Folgenden kurz erläutert werden.

Die Winged-Edge Datenstruktur nach Baumgart [Bau72] ist kantenbasiert und verwendet drei Strukturen, in denen die Daten zu den Punkten, Flächen und Kanten gespeichert werden. Dabei werden in den Strukturen folgende Informationen abgelegt:

- Ein **Punkt** enthält die Punktkoordinaten und einen Verweis auf eine beliebige angrenzende Kante
- Eine **Fläche** speichert den Verweis auf eine adjazente Kante
- Eine **Kante** hat Verweise auf die angrenzenden Punkte (Start- und Endpunkt), Flächen sowie Vorgänger- und Nachfolger-Kanten der angrenzenden Flächen (→ bei einer Traversierung in mathematisch positiver Richtung)

Diese Strukturen können durchaus auch als Tabellen betrachtet werden. Wobei die der Kanten die primäre Rolle zugeschrieben wird, da diese die für die Traversierung des Netzes benötigten Verweise verwaltet (siehe Abb. 2.2). Bei einem Zugriff auf eine Kante ist jedoch unbedingt die Orientierung der Kante zu beachten.



Kante:	E2	
Startpunkt:	Vstart	
Endpunkt:	Vend	
Uhrzeigersinn	CW	CCW
Fläche:	F0	F1
nächste Kante:	E4	E0
vorherige Kante:	E1	E3

Abbildung 2.2: Schematische Darstellung des Winged-Edge-Schemas und der zugehörigen Kanten-Struktur.

2.2 Oberflächenrekonstruktion

Nachdem im vorhergehenden Kapitel ein knapper Überblick über Polygonnetze, beziehungsweise den Spezialfall des Dreiecksnetzes, gegeben wurde, soll nun eine Möglichkeit zur Oberflächenrekonstruk-

tion von Volumendaten beschrieben werden.

Zunächst werden hierzu in mehreren Schichten jeweils die Konturen des Objektes bestimmt. Das Resultat sind mehrere planare Konturlinien, aus denen die Oberfläche rekonstruiert wird. Die Verfahren zur Oberflächenrekonstruktion wurden unter anderem von Meyers [Mey94] ausführlich untersucht. Allgemein wird hierbei vom sogenannten Verfahren der *Surface from Contours* gesprochen. Basierend auf [Mey94] soll im Folgenden ein knapper Überblick über den Prozess der Oberflächenrekonstruktion, der in vier Probleme unterteilt werden kann, vorgestellt werden:

1. **Korrespondenz:** Das *Korrespondenzproblem* entsteht, wenn in Schichtbildern mehrere Konturen zu einem Objekt gefunden werden. Dann muss festgestellt werden, welche Konturen der Schichten zusammengehörig sind. Eine automatische Lösung dieses Problems ist sehr schwierig oder in manchen Fällen sogar unmöglich. Zudem sind die Resultate in besonderem Maße von der Auflösung der Daten abhängig. Oftmals ist daher eine Benutzerinteraktion notwendig, um gewünschte Resultate zu erhalten.

2. **Triangulierung (Tiling):** Das Problem der *Triangulierung* beschreibt die Problematik der Konstruktion der Mantelfläche zwischen den detektierten Konturen. Wünschenswerte Eigenschaften der Algorithmen sind hierbei:

   - Eine gute Qualität der Triangulierung
   - Die Möglichkeit einer Benutzerinteraktion, wenn das Resultat nicht zufriedenstellend ist
   - Mantelflächen sollten auch zu Konturen konstruiert werden können, die sich signifikant voneinander unterscheiden
   - Der Algorithmus sollte schnell sein und eine effiziente Speichernutzung haben

   Doch leider existiert kein Algorithmus der alle oben genannten Kriterien vereint. Somit ist es immer notwendig vorab zu entscheiden, welche Eigenschaften für eine Applikation von Bedeutung sind.

   Unabhängig von den Eigenschaften reduzieren die Algorithmen das Problem der Triangulierung im Allgemeinen auf einen gerichteten Graphen. Dabei stellen die Knoten die möglichen Verbindungskanten und die Kanten die möglichen Dreiecke dar. Aufgrund dessen, dass die Konturen nur in eine Richtung traversiert werden, sind die Kanten des Graphen entsprechend gerichtet. Jeder Knoten hat zwei Folgeknoten, wobei einer das nächste Konturensegment und der andere die Verbindungskante zur benachbarten Kontur darstellt. Zudem ist der erste und letzte Knoten des Graphen identisch, um die Konstruktion einer geschlossenen Mantelfläche zu ermöglichen.

   Auf Basis solcher Graphen kann dann beispielsweise eine Graphensuche durchgeführt werden, die versucht die Triangulierung der Mantelfläche zu suchen, deren Summe der Kantenlängen minimal ist (vgl. Abb. 2.3).

3. **Verzweigungen (Branching):** Das *Verzweigungsproblem* tritt auf, wenn eine Kontur in der nächsten Schicht mit zwei oder mehreren Konturen korrespondiert. In solch einem Fall muss die Mantelfläche verzweigen und entsprechend richtig konstruiert werden. Gewöhnlicherweise werden hierbei die getrennten Konturen miteinander verbunden. Dies geschieht beispielsweise indem zwischen den benachbarten Konturen eine Verbindungskante hinzugefügt wird, so dass die resultierenden Flächen zwischen diesen Konturen einem Sattel ähneln (siehe Abb. 2.3).

4. **Netzanpassung (Mesh-Fitting):** Die vorhergehenden drei Schritte führen zu einem Dreiecksnetz, das eine stückweise planare Approximation der Objektoberfläche ist. Dabei ist die entstandene Oberfläche konsistent mit den Konturen des Objekts. Wenn die Abtastrate hoch genug gewählt wurde, dann sind die Abstände zwischen den Konturen ausreichend gering, um eine gute Approximation der Oberfläche zu erhalten. Anderenfalls wird ein weiterer Schritt, der des *Mesh-Fittings*, notwendig. Hierdurch kann das grobe Dreiecksnetz besser an die gewünschte Oberfläche anpasst werden.

   Die weitere Anpassung der Oberfläche ist nach Meyers ein aktives Forschungsgebiet, deren komplette Darstellung schnell den Rahmen seiner Arbeit sprengen würde. Aus diesem Grund soll auch hier nicht weiter darauf eingegangen werden, sondern auf die Kapitel 2.4 und 3.4 verwiesen werden. Dort wird ausführlich das innerhalb dieser Arbeit umgesetzte Verfahren beschrieben.

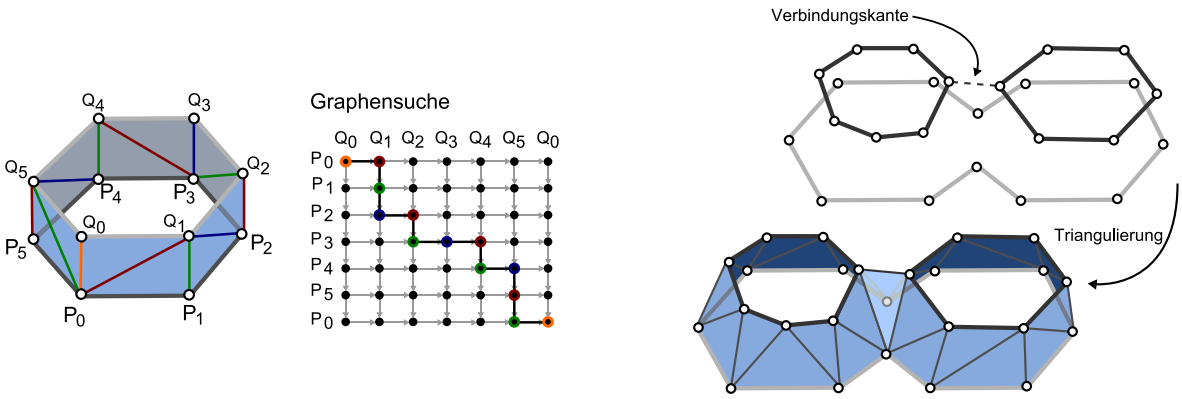


Abbildung 2.3: *Links:* Beispiel einer Triangulierung mittels Graphensuche. *Rechts:* Lösung des Verzweigungsproblems durch einfügen einer Verbindungskante.

Abschließend sei auf das Kapitel 3.3 verwiesen, in dem das Verfahren der Oberflächenrekonstruktion innerhalb dieser Arbeit im Detail beschrieben wird.

2.3 Distanzfelder

Ein Distanzfeld (engl. „Distance Field“) ist eine Textur, in der jeder Texel den euklidischen Abstand zum nächstgelegenen Texel eines Objektes enthält. Bei der Generierung eines Distanzfeldes werden

die euklidischen Abstandswerte mittels einer Distanztransformation (DT) berechnet (siehe 2.3.1). Die Berechnung einer DT und die Darstellung entsprechender Algorithmen und Einsatzmöglichkeiten werden ausführlich in der Arbeit von Cuisenaire [Cui99] thematisiert. Im Folgenden wird ein kurzer Überblick zu möglichen Ansätzen gegeben. Für weiterführende Informationen sei auf [Cui99] und die nachfolgend genannten Arbeiten verwiesen.

Abhängig von der initalen Objektrepräsentation wird bei den Algorithmen, zur Bestimmung der Distanzfelder, zwischen voxelbasierten und polygonalbasierten Verfahren differenziert. Bekannte Verfahren zur effizienten Berechnung der Distanzfelder auf der GPU sind im Fall von Polygonaldaten [SPG03, SGGM06] und für Voxeldaten [CK06, RT06].

Der in dieser Arbeit beschriebene Ansatz wird als Eingabe segmentierte Volumendaten, also binäre Voxelgitter, erhalten. Daher sollen im Weiteren nur die voxelbasierten Verfahren betrachtet werden. Unterschieden werden in dieser Kategorie Methoden, die entweder auf Voronoi-Diagrammen oder auf Propagationsansätzen basieren [CK06]. Ein Voronoi-Diagramm, für den zweidimensionalen Fall, ist die Aufteilung einer Ebene in Regionen, die durch eine vorgegebene Menge an Saatpunkten bestimmt werden. Jede Region enthält einen Saatpunkt und alle Punkte der Ebene die diesem Punkt nächstgelegen sind. Die DT kann dadurch erhalten werden, dass „Voronoi Saatpunkte“ auf den Rand des Objekts gesetzt werden. Innerhalb dieser Arbeit wird ein auf Propagation basierendes Verfahren verwendet. Generell werden hierbei Distanzinformationen auf benachbarte Voxel propagiert und gegebenenfalls modifiziert (siehe Kap. 2.3.2).

Im Folgenden wird nun die Distanztransformation und die Propagation, basierend auf den Arbeiten von [Cui99, CK06, RT06], ausführlicher betrachtet.

2.3.1 Distanztransformation

In diesem Abschnitt wird die Distanztransformation für binär kodierte Bilddaten beschrieben, welche auf der GPU zur Anwendung kommt. Bei der Betrachtung eines binären Voxelgitters, bestehend aus einem geschlossenen Objekt $\mathcal{O}$ und dem Hintergrund, entsteht durch die Anwendung der euklidischen DT das Distanzfeld (vgl. Abb. 2.4 *(a)*) und ist dabei wie folgt für einen Voxel P definiert:

$$dt(P) = (\ dist(P),\ dt_\delta(P)\ ) \tag{2.1}$$

mit

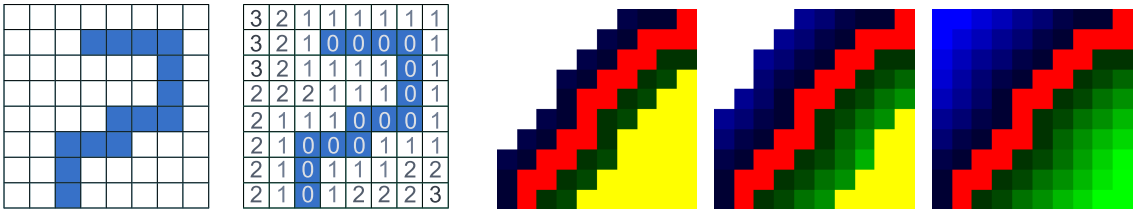
$$dist(P) = \min_{Q \in \mathcal{O}} \{\|P - Q\|\} \tag{2.2}$$

$$dt_\delta(P) = \arg\min_{Q \in \mathcal{O}} \{\|P - Q\|\} \tag{2.3}$$

Für einen Voxel bestimmt $dist(P)$ den euklidischen Abstand zwischen P und dem nächstgelegenen Punkt $Q \in \mathcal{O}$, während $dt_\delta(P)$ den entsprechenden Referenzpunkt $Q \in \mathcal{O}$ ermittelt. Somit speichert $dt(P)$ die minimale Distanz und den entsprechenden Referenzpunkt [CK06].

Wenn der Rand des Objekts $\delta\mathcal{O}$ spezifiziert ist, kann zwischen äußeren und inneren Regionen

($\mathcal{O}_+/\mathcal{O}_-$) unterschieden werden. Der Hintergrund definiert hierbei die äußere und das Objekt selbst die innere Region (ohne den Rand $\delta\mathcal{O}$). Unter der Hinzunahme des Vorzeichens ist es somit möglich Aussagen darüber zu treffen, ob ein Voxel P innerhalb oder außerhalb des Objekts liegt. Wobei es sich in solch einem Fall dann um eine vorzeichenbehaftete Distanztransformation (engl. „Signed Distance Transformation“ - SDT) handelt.



(a) (b)

Abbildung 2.4: Schematisch dargestelltes binär kodiertes Voxelgitter und ein beispielhaftes Distanzfeld *(a)*. Sequentielle Entwicklung: Von *links* nach *rechts* sind 1, 2 und 4 abgeschlossene Propagationsschritte abgebildet *(b)*. *(weiß/gelb: außen/innen „ohne Wert“, blau/grün: außen/innen „mit Wert“, rot: Referenzvoxel)*

2.3.2 Propagations-Verfahren

An dieser Stelle soll das Propagations-Verfahren, für eine vorzeichenbehaftete DT, erklärt werden. Die Erklärung bezieht sich dabei auf Voxelgitter in denen bereits der Rand des Objekts $\delta\mathcal{O}$ spezifiziert wurde. Zu Beginn der Propagation muss das Distanzfeld geeignet initialisiert werden. Hierbei werden die Voxel der äußeren und inneren Regionen ($\mathcal{O}_+/\mathcal{O}_-$) mit einem Maximalwert initialisiert, der größer sein muss als die maximal mögliche Distanz.

$$dt^0(P) = \begin{cases} (0, P) & wenn\ P \in \delta\mathcal{O} \\ (MAX, *) & wenn\ P \in \mathcal{O}_+ \\ (-MAX, *) & wenn\ P \in \mathcal{O}_- \end{cases} \tag{2.4}$$

Wie vorab beschrieben, werden die Distanzinformationen eines Voxels auf die Nachbarschaft propagiert. Die Nachbarschaft von P wird durch einen Filterkern $\mathcal{N}$ definiert. Im zweidimensionalen Fall ist $\mathcal{N}$ beispielsweise 3×3. Für den dreidimensionalen Fall gilt entsprechend $3 \times 3 \times 3$.

Ein Propagationsschritt funktioniert nun so, dass für P in der lokalen Nachbarschaft $\mathcal{N}$ die minimale Distanz wie folgt ermittelt wird:

$$dist^{i+1}(P) = sign_P \min_{Q \in \mathcal{N}(P)} \{ \| dt^i_\delta(Q) - P \| \} \tag{2.5}$$

Das Vorzeichen $sign_P$ wird dabei aus dem vorhergehenden Distanzwert von $dt^i(P)$ gezogen. Durch

die Propagation in der direkten lokalen Nachbarschaft ergibt sich, dass die Distanzwerte sequentiell vom Rand ausgehend propagiert werden (vgl. Abb. 2.4 *(b)*). Aufgrund dieser Art der sequentiellen Entwicklung sind sehr viele Propagationsschritte erforderlich, wodurch der Prozess sehr zeitaufwändig ist. Eine bessere Effizienz bietet das Jump Flooding Verfahren, welches ebenfalls auf Propagation basiert und im Rahmen der Analyse und Konzeption in Abschnitt (3.2.2) näher erläutert wird.

2.4 Shrink-Wrapping-Verfahren

Hinter dem Shrink-Wrapping-Verfahren steht ein physikalisches Modell, wie es auch beim Verpacken mit Schrumpffolie angewandt wird. Hierbei wird ein Objekt in Folie eingewickelt (engl. „wrapped“). Im Anschluss wird die Folie entweder mittels Hitze oder der Erzeugung eines Vakuums geschrumpft (engl. „shrinked“). Am Ende ist das Objekt, nahezu ideal, von der Folie umschlossen.

Die Zielsetzung des Shrink-Wrappings ist es, dieses Prinzip auf Dreiecksnetze zu übertragen. Diese Idee wurde erstmals von Kobbelt et al. [KVLS99] in einem Algorithmus zur Umwandlung beliebiger Dreiecksnetze in Netze mit Subdivisions-Konnektivität aufgegriffen. Basierend auf diesem Ansatz entstanden auch die Arbeiten von [JK02, KCC$^+$05], die Verfahren zur Oberflächenrekonstruktion aus Punktwolken entwickelten, wie sie zum Beispiel bei der Modellerfassung mit einem Laser-Scanner entstehen.

Die genannten Arbeiten unterscheiden sich, aufgrund der teilweise verschiedenen Zielsetzungen, in den Eingabedaten von einander. Zudem werden jeweils unterschiedliche Methoden genutzt, um initiale Dreiecksnetze zu generieren. Dies hat zur Folge, dass sich die Umsetzungen des eigentlichen Shrink-Wrapping-Verfahrens in allen Arbeiten etwas voneinander unterscheiden, aber dem gleichen Prinzip folgen. Aufgrund der Tatsache, dass auch die vorliegende Arbeit die Grundidee in adaptierter Form übernimmt, soll hier eine allgemeine Beschreibung der Funktionsweise ausreichend sein, um ein grundsätzliches Verständnis für dieses Verfahren und seine Probleme zu schaffen. Wobei an geeigneter Stelle, in Kürze, auf die Unterschiede der zugrunde liegenden Arbeiten eingegangen wird. Für weitere Erklärungen sei somit auf die genannten Publikationen und die Umsetzung des Verfahrens in dieser Arbeit (Kap. 3.4) verwiesen.

2.4.1 Allgemeine Funktionsweise

Zur Erklärung der allgemeinen Funktionsweise des Shrink-Wrapping-Verfahrens wird zunächst angenommen, dass das Objekt mit einem Dreiecksnetz S_0 umgeben ist. Dabei soll es zunächst irrelevant sein, wie dieses initiale Netz entstanden ist. Eine genaue Erklärung, wie die Generierung im Fall dieser Arbeit verläuft, wird in Kapitel 3.3 gegeben.

Während des eigentlichen Vorgangs wird jeder Punkt, entsprechend einer auf ihn einwirkenden Kraft, bewegt. Diese Kraft ist eine Kombination zweier Komponenten:

Attracting-Force Die Anzugskraft zieht jeden Punkt des Netzes in Richtung der Oberfläche des Objekts

Relaxing-Force Die „Relaxing-Force“ versucht die Punkte zum Mittelpunkt ihrer 1-Nachbarschaft zu bewegen und somit die Verzerrungen des Netzes zu minimieren. Ohne diese Kraft wäre es nicht möglich lokale Zusammenlagerungen der Punkte (engl. „clustering“) und Selbstüberschneidungen (engl. „self-intersections“) innerhalb des Dreiecksnetzes zu vermeiden [JK02].

Dieser Ansatz ähnelt sehr dem Konzept der aktiven Konturen (*Snakes*) [Ter86, TWK87, KWT88, CC91], welches vor allem in der Bildverarbeitung (engl. „image processing“) und dem maschinellen Sehen (engl. „computer vision“) bekannt ist. Diese Technik wird zur Bestimmung von Konturen in zwei- und dreidimensionalen Bildern verwendet und kommt, wenn auch in vereinfachter Form, in Kapitel 3.3) zur Anwendung.[1] Hierbei wird ein initialer Polygonzug iterativ, mehr und mehr dem Umriss eines segmentierten Objekts angepasst. Dies geschieht mittels einer inneren Stabilisierungskraft und einer äußeren Anzugskraft. Die äußere Kraft zieht die Punkte des Polygons zur Kontur hin und resultiert gewöhnlich aus dem Gradienten, der in der Regel aus einem zugehörigen Skalarfeld berechnet wird. Die innere Kraft dagegen minimiert die Biegungs- und Dehnungsenergie des Polygonzugs (siehe Abb. 2.5), um den Polygonzug so stark wie möglich zu glätten [KVLS99].

Die *Relaxing-Force* des Shrink-Wrapping-Ansatzes kann somit auch als *innere Kraft* bezeichnet werden, während die *Attracting-Force* den Gegenpart als *äußere Kraft* darstellt. Im Folgenden sollen die beiden Komponenten näher betrachtet werden.

Abbildung 2.5: *Links:* Ein Polygonzug (Snake) zieht sich zusammen und verringert somit die *Dehnungsenergie. Rechts:* Durch Minimierung der *Biegungsenergie* soll eine Snake so „glatt“ wie möglich werden. Dies geschieht durch sukzessive Reduktion der energieaufwändigen Segmente, wie z.B. Ecken.

2.4.2 Innere Kraft

Die innere Kraft wird mittels der Laplacian-Glättung erreicht. Zur Anwendung kommt hierbei ein approximierter Laplacian-Filter $\mathcal{L}$, der den durchschnittlichen Vektor aus den Kanten der Umbrella-Umgebung (siehe Abb. 2.6) eines Punktes ermittelt. Im Anschluss wird dieser Vektor dazu genutzt,

[1] Es ist anzumerken, dass bei der Bestimmung der aktiven Konturen sehr unterschiedliche Ansätze möglich sind, deren komplette Aufführung den Rahmen dieser Arbeit sprengen würde. Aus diesem Grund ist die Erklärung hierzu, auf eine grobe Erläuterung und der Beschreibung des Verfahrens dieser Arbeit, beschränkt.

den Punkt zum geometrischen Mittelpunkt seiner Nachbarn zu verschieben [Tau95, KCVS98]. Die neuen „geglätteten“ Positionen ergeben sich wie folgt:

$$s_{new} = s_{old} + \lambda \mathcal{L}(s_{old}) \quad mit\ \lambda \in [0,1] \tag{2.6}$$

$$\mathcal{L}(s_i) = \frac{1}{m} \sum_{j=0}^{m-1} (s_j - s_i) \tag{2.7}$$

Wobei m die Valenz von s_i ist. Wenn für λ ein hoher Wert (nahe 1.0) gewählt wird, dann werden die Punkte des Dreiecksnetzes uniform verteilt. Jedoch kann gerade dies beim Shrink-Wrapping dazu führen, dass konvexe oder konkave Regionen nicht erfasst werden. Im Gegenzug führt ein zu klein gewählter Wert λ zu nicht uniformen Verteilungen und damit gegebenenfalls zu Selbstüberschneidungen. Ein geeigneter Wert muss hier experimentell festgestellt werden [JK02].

Zudem führt die Verwendung von $\mathcal{L}$ zu „Schrumpfungs-Effekten“. Um Letztere weitestgehend zu vermeiden wird lediglich der Senkrecht zur Punkt-Normalen n stehende tangentiale Anteil von $\mathcal{L}$ verwendet [JK02, KCC+05]. Die tangentiale Komponente $\mathcal{L}_t$ zu einem Punkt s_i wird wie folgt bestimmt:

$$\mathcal{L}_t(s_i) = \mathcal{L}(s_i) - (\mathcal{L}(s_i) \cdot n)n \tag{2.8}$$

Die vorangehenden Erläuterungen sind, zum weiteren Verständnis, in der nachfolgenden Abbildung beispielhaft dargestellt.

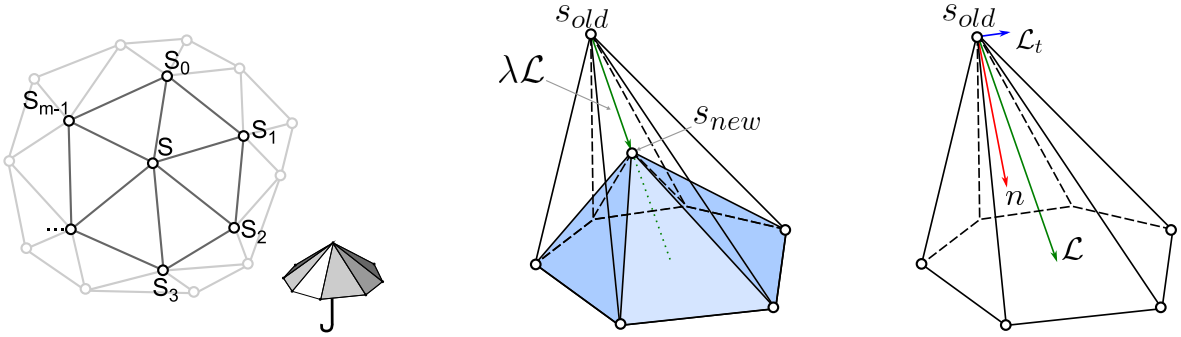


Abbildung 2.6: *Links:* Der Umbrella-Operator liefert die 1-Nachbarschaft eines Punktes s, die für die Berechnung des approximativen Laplacian-Filters benötigt wird. *Mitte:* Ein Beispiel für eine Laplacian-Glättung. Gut zu erkennen ist der „Schrumpfungs-Effekt“. *Rechts:* Eine Veranschaulichung von $\mathcal{L}_t$.

2.4.3 Äußere Kraft

Bei der Berechnung der äußeren Kraft geht es immer darum, zu einem Punkt s_i des zu schrumpfenden Dreiecksnetz S_m, die Richtung zum nächstgelegenen Punkt m_{near} auf der Oberfläche des Objektes $\mathcal{M}$ zu bestimmen. In mehreren Iterationsschritten t wird der Punkt s_i dann, mittels expliziten Euler-Schritten, sukzessiv in Richtung der Oberfläche bewegt. Hierdurch wird das Dreiecksnetz schrittweise

an die Oberfläche angepasst.

$$s_i^{t+1} = s_i^t + \tau(m_{near} - s_i^t) \quad mit\ \tau \in [0,1] \tag{2.9}$$

Zu beachten ist, dass die Genauigkeit der Anpassung von der Schrittweite τ abhängig ist. Bei der Bestimmung des nächstgelegenen Objektpunktes unterscheiden sich die drei genannten Arbeiten wie folgt voneinander:

Kobbelt et al. nutzen einen Projektions-Operator P, der jeden Punkt s_i auf den nächstgelegenen Punkt von $\mathcal{M}$ abbildet. Die Projektion wird bestimmt, indem Strahlen von S_m in Richtung der Normalen gebildet und dann die Schnittpunkte mit $\mathcal{M}$ berechnet werden [KVLS99]. Zur Erinnerung, $\mathcal{M}$ ist bei Kobbelt et al. ein beliebiges Dreiecksnetz.

Jeong und Kim arbeiten mit Punktwolken und können daher keinen Projektions-Operator wie Kobbelt et al. verwenden. Sie führen daher für jeden Punkt s_i eine globale Suche in $\mathcal{M}$, der Punktwolke, aus und bestimmen somit die entsprechende Richtung [JK02].

Koo et al. arbeiten ebenfalls mit Punktwolken, unterteilen diese jedoch in einem ersten Schritt in orthogonale Zellen. Eine Zelle hat nun entweder einen Inhalt (einen oder mehrere Punkte) oder ist leer. Basierend auf dieser Zell-Struktur wird das initiale Dreiecksnetz S_m generiert. Wenn nun nach dem nächsten Objektpunkt gesucht wird, ist es ausreichend in der lokalen Nachbarschaft der Zelle zu suchen, die s_i enthält. Da m_{near} entweder in der gleichen Zelle wie s_i oder in einer seiner 26 Nachbarn enthalten ist, müssen lediglich 27 Zellen durchsucht werden, was gegenüber einer globalen Suche sehr viel Zeit spart [KCC+05].

2.4.4 Probleme

Das vorgestellte Schema kann zu Artefakten, in Form von Falten, in der Netzstruktur führen. Dieser Effekt kann auch bei realen Verpackungen beobachtet werden und ist darauf zurückzuführen, dass sich die Hülle und das Zielobjekt signifikant voneinander unterscheiden. Das hat zur Folge, dass manche Regionen der Hülle sehr stark gedehnt werden und andere Teile sich sehr stark zusammenziehen und infolgedessen zu viel Material haben, wodurch die Falten entstehen [KVLS99].
Dieses Grundproblem des Verfahrens, was in Extremfällen zu Selbstüberschneidungen führt, wird in Kapitel 3.4 nochmals detaillierter aufgegriffen. Dabei wird der Lösungsweg dieses Ansatzes beschrieben, der die Selbstüberschneidungen bestmöglich vermeiden soll.

Kapitel 3

Analyse und Konzeption

3.1 Übersicht

Nachdem das vorhergehende Kapitel die Grundlagen des vorliegenden Ansatzes beschrieben hat, wird nun das Konzept durch eine theoretische Betrachtung erörtert. Dazu wird zu Beginn eine Abgrenzung des Kontextes und eine Beschreibung der Zielsetzung erfolgen. Im Anschluss daran wird ein knapper Überblick über die einzelnen Schritte des Verfahrens gegeben. Die im Überblick verwendete Aufteilung der Schritte, wird dann auch für die eigentliche Erklärung des Verfahrens beibehalten. Das Kapitel schließt mit einer kurzen Zusammenfassung der wichtigsten Aspekte ab.

3.1.1 Abgrenzung und Zielsetzung

Die direkte Volumenvisualisierung erzielt durch das Raycasting eine hohe Darstellungsqualität. Die Diplomarbeit von [Brü07] stellt einen echtzeitfähigen Ansatz zur direkten Deformation von Volumendaten vor, der zudem auch diese gewohnt hohe Qualität beibehält. Die Arbeit beschreibt ein Raycasting-Verfahren, dass nicht direkt auf den Volumendaten arbeitet, sondern auf einem approximierenden Tetraedergitter, welches zur Diskretisierung des Datenraums genutzt wird. Optimal wäre es, wenn die Hilfsgeometrie existierende Segmentierungsmasken des Volumens jeweils durch ein Tetraedergitter approximieren würde [Brü07].

Die Zielsetzung der vorliegenden Arbeit soll die Erzeugung solcher Hilfsgeometrien sein. Dabei wird davon ausgegangen, dass die benötigten Segementierungsmasken des Volumens bereits in Form von binären Voxelgittern existieren. Die Anforderung, die an die zu erzeugende Geometrie gestellt wird, richtet sich primär darauf, dass der segmentierte Bereich des Datensatzes komplett und geschlossen umhüllt, aber nicht bis ins kleinste Detail aufgelöst wird. Der hier beschriebene Ansatz wird daher im Wesentlichen den relevanten Bereich des Datensatzes durch eine umhüllende und geschlossene Dreiecksoberfläche approximieren. Das letztliche Tetraedergitter wird dann zu dieser Approximation, mittels der freien Software *Gmsh* [GR08], generiert.

3.1.2 Von der Eingabe bis zum Resultat - das Konzept

Nach dem Einladen der binären Voxeldaten, kann die Ermittlung eines zugehörigen Tetraedergitters in die nachfolgenden Abschnitte unterteilt werden (siehe auch Abb. 3.1).

- Vorverarbeitung der Voxeldaten
  Vor Beginn der eigentlichen Verarbeitung müssen zunächst obligatorische Vorverarbeitungsschritte (Kap. 3.2) ausgeführt werden. Zur Hauptaufgabe gehört hier das Berechnen des dreidimensionalen und vorzeichenbehafteten Distanzfeldes. Dieses wird mittels des, auf Propagation basierenden, Jump Flooding Algorithmus gewonnen. Auch die Bestimmung der umhüllenden Geometrien für das gesamte Volumen und den einzelnen planaren Ebenen der Segmentierungsmasken gehört zu den Vorverarbeitungsschritten. Zudem ist es möglich feine Strukturen der Voxeldaten mittels eines Glättungsfilters zu reduzieren.

- Konstruktion der initialen Dreiecksoberfläche
  Nachdem zu dem Datensatz das zugehörige Distanzfeld berechnet wurde, wird auf dessen Basis eine approximierende Dreiecksoberfläche aus den Volumendaten extrahiert (Kap. 3.3). Dies wird dadurch erreicht, dass zunächst in definierten Abständen, grobe und planare Konturen, nach dem Prinzip der aktiven Konturen und auf Basis des Distanzfeldes, ermittelt werden. Diese Konturen werden im Anschluss mittels der Triangulierung miteinander zu einer Mantelfläche verbunden.

- Anpassen der Dreiecksoberfläche
  Die Anpassung erfolgt nach dem Prinzip des Shrink-Wrappings (Kap. 3.4), wodurch das initiale Dreiecksnetz iterativ dem Eingabedatensatz angepasst wird. Dieser Schritt umfasst unter anderem die Ermittlung der inneren und äußeren Kraft. Die innere Kraft wird jeweils aus der 1-Nachbarschaft eines Punktes gewonnen, während die äußere Kraft mittels der Berechnung der Gradienten im Distanzfeld ermittelt wird. Die benötigte Anzugskraft wird durch die Umkehrung der Gradienten erreicht, wodurch die Punkte schrittweise in Richtung der Zieloberfläche verschoben werden. Diese Art der Annäherung macht es bekanntlich unvermeidlich, dass Regionen im Netz entstehen, die gestreckt oder gestaucht werden. Die Charakteristik dieses Ansatzes und die innere Kraft führen dazu, dass der Grad der Anpassung eines Netzes durchaus eingeschränkt ist. Zum Beispiel ist in stark gedehnten Bereichen die innere Kraft im Verhältnis zur äußeren Kraft so groß, dass die Anzugskraft nahezu keinen Einfluss mehr auf einen Punkt nimmt. Dieser Punkt wird somit im Mittelpunkt seiner lokalen 1-Nachbarschaft stagnieren, obwohl er vielleicht noch nicht ideal angepasst ist. Um die Anpassung dennoch weiter verbessern zu können, wird nach jedem Shrink-Wrapping Durchlauf ein Nachbearbeitungsschritt durchgeführt.

- Nachbearbeitung zur Optimierung der Dreiecksoberfläche
  Dieser Schritt (Kap. 3.5) dient zur Erweiterung der Oberflächendetails und der Optimierung des Dreiecksnetzes. Wie zuvor beschrieben, wird dieser immer im Anschluss an einen Shrink-

Wrapping-Durchlauf angewendet. Hierbei wird die Absicht verfolgt, dass die betreffenden Regionen, entweder durch eine lokale Reduktion oder Erweiterung der Punkte, optimiert werden. Dieser Vorgang nimmt einen großen Einfluss auf die innere Kraft des Netzes, so dass zum Beispiel in Regionen die „überdehnt“ waren, weitere Anpassungen mittels der Anzugskraft möglich werden. Dementsprechend wird, durch die wiederholte Anwendung des Shrink-Wrappings und der Nachbearbeitung, die Dreiecksoberfläche dem Zielobjekt immer mehr angepasst.

- Generieren der Tetraedergitter
  Hat die Verarbeitung diesen Schritt erreicht, so steht eine approximierende Dreiecksoberfläche des Datensatzes zur Verfügung. Dieses Dreiecksnetz wird nun in das Geometrieformat von *Gmsh* exportiert. Diese Geometrie wird daraufhin von *Gmsh* eingeladen und zur Generierung des Tetraedergitters genutzt.

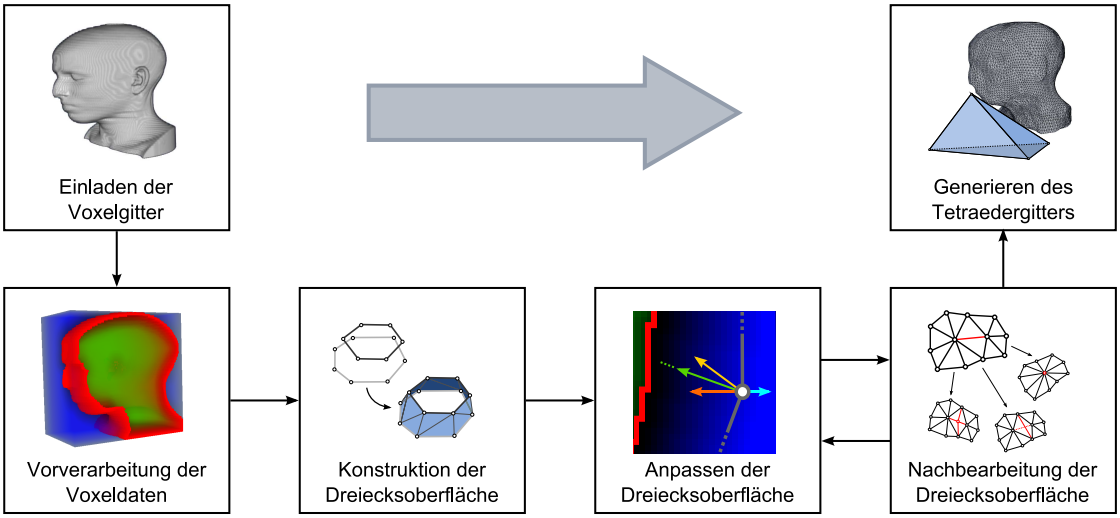


Abbildung 3.1: Ablaufdiagramm zum Konzept.

Nach diesem skizzierenden Überblick soll eine ausführliche Betrachtung der Hauptschritte erfolgen.

3.2 Vorverarbeitung der Voxeldaten

Die Vorverarbeitung generiert die notwendigen Daten, die für die anschließenden Verarbeitungsschritte obligatorisch sind. Unter anderem umfasst dieser Verarbeitungsschritt eine optionale Detailreduktion des Eingabedatensatzes sowie die Bestimmung der umhüllenden Geometrien für die später folgende Konturendetektion. Doch die primäre Aufgabe ist die Generierung des vorzeichenbehafteten Distanzfeldes, dass die Grundlage fast aller Berechnungen des vorliegenden Verfahrens darstellt.

3.2.1 Detailreduktion

Bei der Verarbeitung der Voxeldaten ist es optional möglich, eine Detailreduktion im Datensatz mittels eines linearen Glättungsfilters zu erzielen. Die Glättung erfolgt durch die Anwendung eines Gauss-Filters, indem der Datensatz mit dem Filterkern ($3 \times 3 \times 3$) gefaltet wird.

Durch das Glätten des Datensatzes verwischen die Kanten, wodurch das Objekt geringfügig vergrößert wird. Diese Vergrößerung wirkt sich auch auf die Konstruktion des initialen Dreiecksnetzes aus. Zum einen wird dieses ebenfalls entsprechend größer und zum anderen ist es nun weniger wahrscheinlich, dass bei der Detektion der Konturen feine Details nicht detektiert werden.

Primär wird durch die Vergrößerung des Dreiecksnetzes jedoch erreicht, das während des ersten Durchlaufs der Oberflächenanpassung eine verbesserte Verteilung der Punkte erzielt wird. Hierdurch wird die Qualität der Oberfläche stark verbessert. Für den Augenblick soll diese Aussage als grobe Übersicht ausreichend sein, da die genaue Erklärung in dem Kapitel 3.4.3 erfolgt.

3.2.2 Generierung des Distanzfeldes

Aus den Grundlagen ist bekannt, dass das Shrink-Wrapping-Verfahren sowie das Verfahren der aktiven Konturen mit einer inneren und äußeren Kraft arbeiten. Zudem wurde eine Übersicht über die verschiedenen Ansätze zur Gewinnung der Anzugskraft der Arbeiten von [KVLS99, JK02, KCC$^+$05] vorgestellt.

In diesem Abschnitt soll nun beschrieben werden, auf welche Art und Weise die Anzugskraft in diesem Ansatz gewonnen wird. Anders als bei den vorherigen Arbeiten steht dem vorliegenden Verfahren nämlich weder eine Referenzfläche, auf die projiziert werden kann, noch eine Punktwolke, in der eine globale Suche durchgeführt werden könnte, zur Verfügung. Daher wird zur Ermittlung der Anzugskraft ein vorzeichenbehaftetes Distanzfeld zum Datensatz berechnet, so dass es möglich ist zu den Punkten den entsprechenden Gradienten mittels zentraler Differenzen zu ermitteln.

Die Generierung des Distanzfeldes erfolgt durch ein auf Propagation basierenden Verfahren, dem sogenannten Jump Flooding Algorithmus. Doch bevor mit der eigentlichen Propagation begonnen werden kann, ist es notwendig den Rand des binären Objektes im Voxelgitter zu bestimmen. Somit kann die Ermittlung eines Distanzfeldes in die folgenden zwei Schritte aufgeteilt werden:

1. Initialisierung des Voxelgitters
2. Propagation der Distanzwerte

Diese zwei Schritte sollen nun erläutert werden.

Initialisierung des Voxelgitters

Zu der Initialisierung des Voxelgitters gehört unter anderem die Ermittlung des Objektrandes. Wie eingangs erwähnt sind die Segmentierungsdaten binär kodiert und somit ist ein Voxel entweder leer oder gehört zum Objekt. Bei der Detektion des Randes wird jeder Voxel mittels eines Filterkerns,

der die 1-Nachbarschaft im Voxelgitter abdeckt, klassifiziert. Hierbei gibt es drei mögliche Zustände (siehe auch Abb. 3.2):

Objekt-Voxel Sobald ein Voxel zugehörig zu einem Objekt ist, wird er, unabhängig von seiner Nachbarschaft, als Objekt-Voxel deklariert.

Leer-Voxel Dies ist ein Voxel der nicht zu einem Objekt gehört und auch nicht an ein Objekt-Voxel angrenzt.

Rand-Voxel Ein Rand-Voxel ist kein Objekt-Voxel, grenzt jedoch an mindestens einen Objekt-Voxel an.

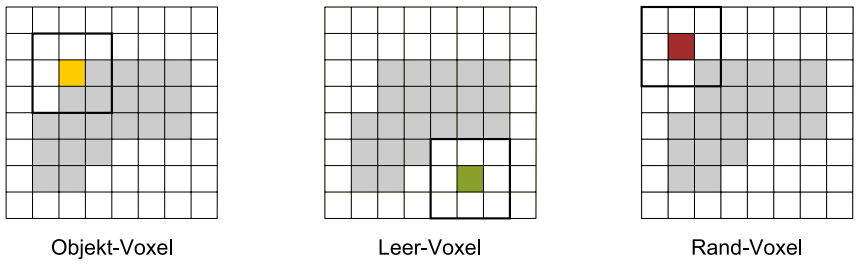


Abbildung 3.2: Zweidimensionale schematische Darstellung der möglichen Voxel-Klassifikationen.

Neben der eigentlichen Detektion des Randes wird zugleich das Voxelgitter für die anschließend erfolgende Propagation vorbereitet. Dabei werden, wie in den Grundlagen thematisiert, die Rand-Voxel mit einem Nullwert und jeder Nicht-Rand-Voxel mit einem Maximalwert initialisiert (vgl. Kap. 2.3.2). Ein vorzeichenbehaftetes Distanzfeld macht es zudem notwendig, dass die inneren und äußeren Regionen markiert werden. Infolgedessen werden Leer-Voxel mit positiven und Objekt-Voxel mit negativen Vorzeichen versehen.

Propagation der Distanzwerte

Nach der Initialisierung des Voxelgitters erfolgt die Phase der Propagation. Diese Phase stellt einen sehr zeitaufwendigen Prozess dar und ist somit nicht besonders günstig für interaktive Applikationen. Aus diesem Grund wird innerhalb dieser Arbeit der GPU-basierte Jump Flooding Algorithmus (JFA) von Rong und Tan [RT06] verwendet.

Bei dem JFA handelt es sich um ein approximatives Verfahren zur Berechnung von Voronoi-Diagrammen, welches aber zugleich für die DT geeignet ist. In den Grundlagen wurde das Propagieren mit einem Filterkern, der die 1-Nachbarschaft abdeckt, erläutert. Durch die Beschränkung auf die direkte Nachbarschaft, werden sehr viele Propagationsschritte notwendig, um jeden Voxel im Gitter zu erreichen. Genau an diesem Punkt setzen Rong und Tan an. Sie verwenden eine variable Schrittweite k, die die Beschränkung auf die 1-Nachbarschaft auflöst, indem k je Durchlauf variiert wird. Das Variieren von k erfolgt dabei entweder durch Verdoppelung oder Halbierung (siehe Abb.3.3), was dazu führt, dass jeder Voxel mit einer logarithmischen Anzahl von Schritten erreicht wird. Zur Re-

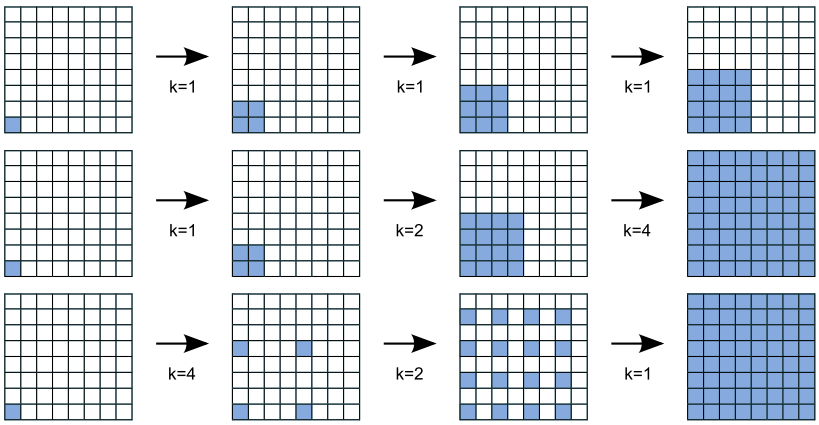


Abbildung 3.3: Propagationsdurchläufe mit verschiedenen Schrittweiten: 1-Nachbarschaft *(oben)* und Jump Flooding mit Verdoppelung und Halbierung der Schrittweite *(mitte/unten)*.

duktion möglicher Fehler empfehlen Rong und Tan, im Anschluss an das Verfahren noch einen oder mehrere weitere Propagationsdurchgänge durchzuführen. In diesem Fall wird das Verfahren auch als JFA+n bezeichnet. Wobei n die Anzahl zusätzlicher Propagationsschritte ist, die mit einer Schrittweite $k = 2^{n-1}$ durchgeführt werden [RT06]. Innerhalb dieser Arbeit kommt JFA+1 zur Anwendung und die Schrittweite wird dabei je Durchlauf halbiert.

3.2.3 Umhüllende Geometrien

Die Ermittlung der Konturen erfolgt mittels des Konzepts der aktiven Konturen. Letzteres macht es erforderlich, dass in der Nähe des zu detektierenden Objektes ein initialer Polygonzug erstellt wird, der dann mehr und mehr dem gewünschten Umriss angepasst wird. Dieser Polygonzug wird im vorliegenden Ansatz das umhüllende Rechteck des Objektrandes einer jeweiligen Schicht sein.

Da die Konstruktion des Dreiecksnetzes wahlweise in *X*-, *Y*- oder *Z*-Richtung erfolgt, ist es notwendig, entsprechend dieser drei Möglichkeiten, auch die dazugehörigen geschichteten Rechtecke zu ermitteln und zu speichern. Die Speicherung der Rechtecke erfolgt, indem pro Richtung eine Liste mit den entsprechenden Min/Max-Werten geführt wird. Ein Eintrag ist dabei eine Schicht und für den Fall, dass eine Schicht keine Randpunkte enthält, wird der entsprechende Eintrag als leer markiert.

Die Rechtecke werden während der Generierung des Distanzfeldes und zwar bei der Ermittlung des Objektrandes gewonnen. Auf diese Weise wird ein extra Renderingschritt eingespart. Sobald der Objektrand einer Schicht ermittelt ist, werden die Randpunkte aus dem Framebuffer ausgelesen und verarbeitet. In Abhängigkeit der Richtung wird dabei, mittels der entsprechenden Komponente des Randpunktes, der Eintrag der zugehörigen Liste festgelegt. Die übrigen beiden Komponenten des Punktes werden dann mit den bisherigen Min/Max-Werten der Schicht verglichen und gegebenenfalls modifiziert (vgl. Abb. 3.4).

Zugleich wird analog der minimale und maximale Punkt des gesamten Objekts bestimmt. Somit steht nach Abschluss der Initialisierung auch der Achsen ausgerichtete und umhüllende Quader (engl.

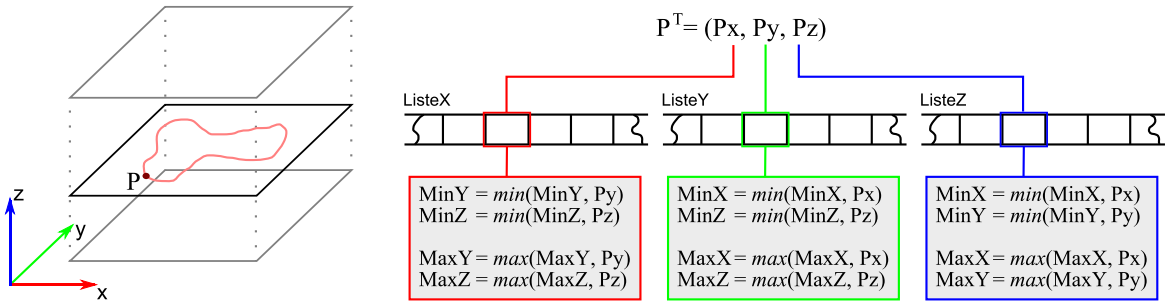


Abbildung 3.4: Schematisches Beispiel zur Ermittlung der umhüllenden Rechtecke.

„Axis-Aligned Bounding Box“ - AABB) des Volumens fest. Die AABB wird bei der Bestimmung der Konturen genutzt, um Informationen über die Größe und Position des Volumens zu haben. Mittels dieser Informationen wird zum einen die Position der ersten Kontur festgelegt und zum andern eine Kantenlänge bestimmt, die als Referenzwert gilt (vgl. Kap. 3.3.2).

3.3 Konstruktion der initialen Dreiecksoberfläche

In diesem Kapitel wird detailliert die Konstruktion der approximierenden Dreiecksoberfläche erläutert, die die Grundlage der anschließenden Anpassung darstellt. Zu Beginn wird eine Darstellung der Problematik erfolgen, um die Herausforderungen und Kernfunktionen dieses Kapitels herauszustellen. Im Anschluss daran wird das verwendete Verfahren dieser Arbeit, dass in die Abschnitte „Ermittlung der Konturen“ und „Verbinden der Konturen“ unterteilt ist, ausführlich vorgestellt. Zum Abschluss dieses Kapitels wird ein kurzer zusammenfassender Überblick über das gewonnene Resultat der Konstruktionsphase gegeben.

3.3.1 Problemstellung

Die Konstruktion der Dreiecksoberfläche basiert auf dem Prinzip der „Surface from Contours“. Somit ist es zunächst notwendig die Konturen zu den Segmentierungsmasken zu bestimmen und im Anschluss auf geeignete Weise miteinander zu verbinden. Die Problematik der Triangulierung ist einfach zu lösen, sofern die Konturen gleiche Form, Metrik und Orientierung haben (vgl. Abb. 3.5). Gerade bei natürlichen Daten ist dies jedoch häufig nicht der Fall, wodurch das Finden einer akzeptablen Triangulierung erheblich erschwert wird [Mey94]. Je größer die Unterschiede der benachbarten Konturen sind, desto mehrdeutiger fallen dann nämlich auch die Verbindungsmöglichkeiten aus. Meyers beschreibt, dass in schweren Fällen zu wenig Informationen genutzt werden, um eine gute Lösung zu finden. Denn üblicherweise basieren die Algorithmen zur Triangulierung auf einfach zu berechnenden Bewertungskriterien, die nicht ausreichend sind, um komplexe Konturen zu verbinden. Seiner Ansicht nach benötigen die Lösungsverfahren, in schwierigen Fällen, eher eine globale Betrachtung und mehr Wissen über die Daten.

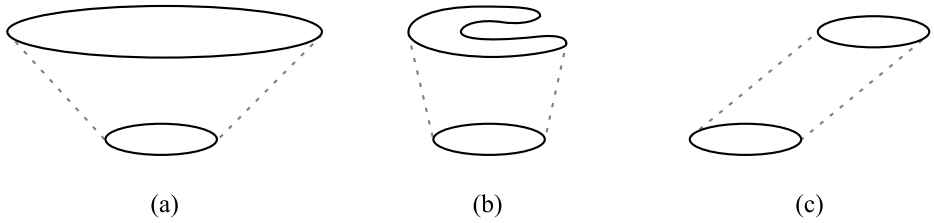


Abbildung 3.5: Korrespondierende Konturenpaare mit unterschiedlichen Metriken *(a)*, Formen *(b)* und Orientierungen *(c)*.

Aus den Grundlagen ist bekannt, dass für die Eigenschaften des Triangulierungsschritts Prioritäten gesetzt werden müssen, da nicht alle miteinander vereinbar sind. Generell ist die Zuordnung der Verbindungen zwischen den Konturen, wie zuvor beschrieben, aufgrund von Mehrdeutigkeiten, eine schwere Aufgabe. Aus diesem Grund muss das Hauptaugenmerk dieser Arbeit vor allem auf die Stabilität gerichtet sein.

Im Folgenden soll nun die Oberflächenrekonstruktion innerhalb dieser Arbeit ausführlich beschrieben werden. Bevor jedoch der Problemschwerpunkt der Triangulierung erläutert wird, soll zuvor die Ermittlung der Konturen betrachtet werden.

3.3.2 Ermittlung der Konturen

Bekanntlich basiert die Konturendetektion dieses Ansatzes auf dem Konzept der aktiven Konturen. Ziel ist es die umhüllenden Rechtecke als initiale Polygonzüge zu verwenden und diese sukzessiv dem gewünschten Umriss anzupassen. Somit kann die Konturendetektion in die Bestimmung und die Anpassung der Polygonzüge unterteilt werden.

Bestimmung der initialen Polygonzüge

Die umhüllenden Rechtecke bestehen bislang nur aus dem Minimal- und Maximalpunkt (P_{min}/P_{max}) und würden somit als Polygonzug nur eine Strecke darstellen. Da aber ein Rechteck gewollt ist, müssen aus den Min/Max-Werten zunächst die zwei anderen Eckpunkte (P_{new1}/P_{new2}) abgeleitet werden. Diese fehlenden Punkte werden hierbei passend aus den Komponenten der vorhandenen Punkte zusammengesetzt. Durch die schichtweise Abtastung des Objekts, haben die Min/Max-Werte eine konstante Komponente. Wobei es von der Abtastrate abhängig ist, welche Komponente konstant ist. Für den weiteren Verlauf der Erklärung wird nun angenommen, dass die Abtastrichtung entlang der *Z*-Achse verläuft. Somit ist die *Z*-Komponente die Konstante und daraus folgt, dass die neuen Punkte ebenfalls den gleichen *Z*-Wert erhalten. Daher werden nun lediglich noch die jeweiligen *XY*-Komponenten benötigt. Aufgrund der Orthogonalität im Rechteck können die fehlenden Punkte wie

folgt gewonnen werden:

$$P_{new1} = \begin{pmatrix} P_{min}x \\ P_{min}y \\ P_{min}z \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} P_{max}x \\ P_{max}y \\ P_{max}z \end{pmatrix} \tag{3.1}$$

$$P_{new2} = \begin{pmatrix} P_{min}x \\ P_{min}y \\ P_{min}z \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} P_{max}x \\ P_{max}y \\ P_{max}z \end{pmatrix} \tag{3.2}$$

Nach der Bestimmung der zwei fehlenden Punkte müssen die vier Kanten des Rechtecks, aus der Menge der Punkte, gebildet werden.

$$E_0 = \overline{P_{min}P_{new1}}\ ;\ E_1 = \overline{P_{new1}P_{max}}\ ;\ E_2 = \overline{P_{max}P_{new2}}\ ;\ E_3 = \overline{P_{new2}P_{min}} \tag{3.3}$$

Ein Polygonzug mit lediglich vier Punkten ist für die nachfolgende Anpassung jedoch eine denkbar schlechte Ausgangslage. Um ein gutes Resultat zu ermöglichen, werden die entstandenen Kanten noch durch Einfügen weiterer Punkte $P^i_{E_x}$, unterteilt. Dies geschieht indem eine Kantenlänge l definiert wird, auf deren Basis dann zum Beispiel E_0 wie folgt unterteilt wird:

$$P^i_{E_0} = P_{min} + i \cdot inc \cdot \frac{P_{new1} - P_{min}}{\|E_0\|},\ i = 1, \ldots, n-1 \tag{3.4}$$

mit

$$n = \max\left(2, \left\lfloor \frac{\|E_0\|}{l} \right\rfloor\right),\ inc = \frac{1}{n} \cdot \|E_0\|$$

Die Unterteilung der anderen Kanten erfolgt analog. Die Definition von l wird zuvor durch den Benutzer vorgenommen, indem er mittels der grafischen Benutzeroberfläche die Anzahl der Schichten ($numPlanes$) und die Verarbeitungsrichtung spezifiziert. Entsprechend der gewählten Richtung wird l aus der Größe des Objekts wie folgt berechnet:

$$l = \frac{1}{numPlanes} \cdot ObjectSize \tag{3.5}$$

Wobei $ObjectSize$ entweder die *Höhe*, *Breite* oder *Tiefe* der AABB des Volumens darstellt (vgl. Kap. 3.2.3). Da für diese Erläuterung die Z-Richtung als Verarbeitungsrichtung angenommen wurde, würde die Kantenlänge also basierend auf der Tiefe berechnet werden.

Nach Abschluss der Unterteilung ist der Polygonzug fertig vorbereitet und kann nun an den Umriss des Objekts angepasst werden.

Anpassung der Polygonzüge

Die Anpassung erfolgt iterativ mittels einer inneren und äußeren Kraft. Dabei steht die äußere Kraft für

die Kraft, die sozusagen von außen auf den Polygonzug einwirkt und ihn an ein Objekt anpasst. Diese Anzugskraft wird mittels der Berechnung des Gradienten erreicht. Bleibt noch die innere Kraft, die bekanntlich die Dehnungs- und Biegungsenergie im Polygonzug minimiert. Diese Kraft wird durch den Polygonzug selbst erzeugt, indem für jeden Punkt die innere Kraft aus seiner Nachbarschaft berechnet wird. Bevor jedoch das Zusammenspiel beider Kräfte erläutert wird, sollen zuvor die jeweiligen Berechnungen betrachtet werden.

Berechnung der Anzugskraft

Wie zuvor beschrieben wird der Gradient als Anzugskraft genutzt, doch bevor nun die eigentliche Berechnung erläutert wird, soll zunächst kurz auf seine Definition und Eigenschaften eingegangen werden.

Die partiellen Ableitungen erster Ordnung einer differenzierbaren skalaren Funktion $f(x, y, z)$ ermöglichen Aussagen über die Änderungen des Funktionswertes f, wenn man von einem Punkt P aus in Richtung der betreffenden Koordinatenachsen fortschreitet. Die Zusammenfassung der Ableitungen zu einem Vektor, wird als Gradient bezeichnet [Pap01].

$$grad_{3D}\ f(x,y,z) \ = \ \begin{pmatrix} \frac{\partial f(x,y,z)}{\partial x} \\ \frac{\partial f(x,y,z)}{\partial y} \\ \frac{\partial f(x,y,z)}{\partial z} \end{pmatrix} \tag{3.6}$$

Im Fall eines ebenen Skalarfeldes reduziert sich der Gradient auf zwei Komponenten ($grad_{2D}$) und ist somit ein ebener Vektor [Pap01].
Der Gradient hat im Wesentlichen die folgenden drei Eigenschaften:

- Er zeigt in Richtung des steilsten Anstiegs,
- steht Senkrecht auf den Isolinien von f
- und ist gleich null bei einem lokalen Extremum.

Da der Gradient, innerhalb dieser Arbeit, als Anzugskraft genutzt werden soll, macht es die erste Eigenschaft notwendig, dass dieser invertiert werden muss. Andernfalls würden die Punkte nicht zum Rand des Objekts hingezogen, sondern würden sich mehr und mehr entfernen. Dies gilt allerdings nur für Gradienten die in positiven Bereichen der SDT berechnet werden. In negativen Bereichen zeigt der Gradient bereits in die richtige Richtung, da der Rand bekanntlich mit null initialisiert ist.

Des Weiteren ist durch die dritte Eigenschaft sichergestellt, dass ein Punkt der den Objektrand erreicht hat, sich nicht mehr weiterbewegt, da sein Gradient und somit seine Anzugskraft gleich null ist. Dies ist darauf zurückzuführen, dass der Rand ein lokales Extremum im Distanzfeld darstellt. Das heißt, er ist somit entsprechend als Senke/Quelle für positive/negative Regionen zu betrachten.

Wie zuvor beschrieben ist der Gradient das Differential einer Funktion $f(x, y, z)$. Innerhalb dieses Ansatzes findet die Berechnung des Gradienten jedoch auf Gittern statt, weshalb das Differential durch

finite Differenzen ersetzt wird. Die Bestimmung der Ableitung, zum Beispiel einer eindimensionalen Funktion $f(x_i)$ an der Stelle x_i, erfolgt dann entweder durch die Verwendung der *Vorwärtsdifferenzen*

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{h}, \tag{3.7}$$

der *Rückwärtsdifferenzen*

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{h} \tag{3.8}$$

oder der *zentralen Differenzen*

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{2h}. \tag{3.9}$$

Wobei h der Abstand zwischen den Stellen ist. Diese drei Quotienten stellen einfache Methoden zur Berechnung der Ableitung dar. Da mit Hilfe der zentralen Differenzen die beste Näherung erreicht wird, werden diese auch in der vorliegenden Arbeit verwendet. Aufgrund dessen, dass die zu detektierenden Konturen immer in einer Ebene liegen, wird hierbei stets der ebene Gradient ($grad_{2D}$) ermittelt. Die Bestimmung der zentralen Differenzen erfolgt auf der GPU, genauer in dem *Fragment-Shader* und wird im Kapitel 4.3.1 zur Implementierung nochmals aufgegriffen und genauer beschrieben.

Berechnung der inneren Kraft

Aus den Grundlagen ist bekannt, dass die innere Kraft mittels der lokalen Nachbarpunkte gewonnen wird. Die Berechnung erfolgt nach der Formel für die Laplacian-Glättung (vgl. Formel 2.7) und wird hier nochmals zur Erinnerung aufgeführt. Allerdings geschieht dies direkt für den Fall der Konturendetektion, wodurch die Umgebung eines Punktes auf genau zwei direkte Nachbarn beschränkt wird.

$$\mathcal{L}(P_i) = 0.5 \sum_{j=0}^{1} (P_j - P_i) \tag{3.10}$$

Die Berechnung ergibt einen Vektor, der in das geometrische Zentrum von der Umgebung von P_i zeigt. Werden die Punkte nur in Abhängigkeit der inneren Kraft bewegt, so wird jeder Punkt in sein geometrisches Zentrum gerückt. Dies hat zur Folge, dass sowohl die Dehnungsenergie, als auch die Biegungsenergie minimiert werden. In Abbildung 3.6 *(b)* ist diese Minimierung und damit die Glättung sehr gut zu erkennen. Besonders deutlich zeigt sich dies durch die abgerundeten Ecken.

Kombination der beiden Kräfte

Damit die Konturen möglichst gut detektiert werden können, müssen die beiden Kräfte geeignet gewichtet werden. Eine zu starke innere Kraft würde eine Anpassung durch die Anzugskraft verhindern. Umgekehrt würde eine zu starke Anzugskraft dazu führen können, dass es im Extremfall zu Selbstüberschneidungen im Polygonzug kommen könnte. Daher wird zwischen beiden Kräften linear

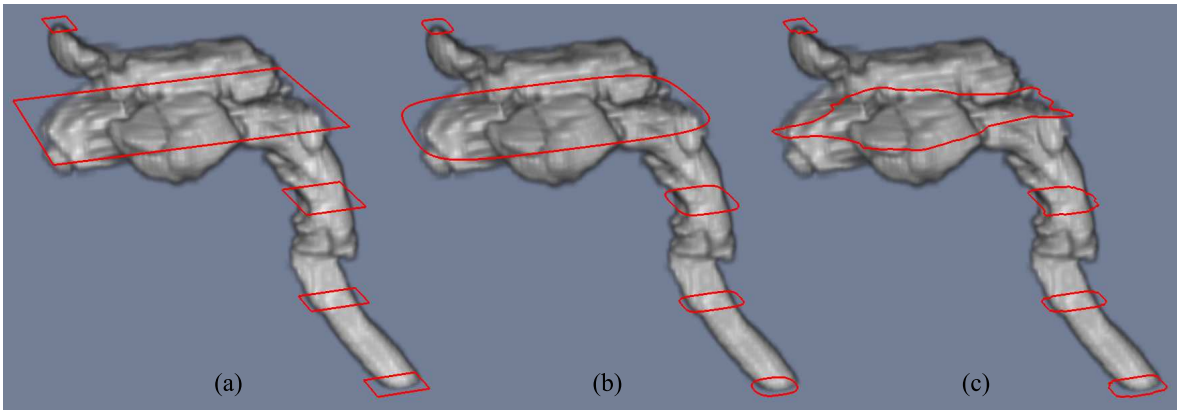


Abbildung 3.6: Initiale Polygonzüge *(a)*, Auswirkungen der inneren Kraft *(b)* und Kombination beider Kräfte *(c)*.

interpoliert, wobei eine geeignete Gewichtung experimentell festzustellen ist.

$$P_{new} = P_{old} + \tau\,(\,(1-\alpha)(-grad_{2D}(P_{old}) \cdot Dist(P_{old})) + \alpha\mathcal{L}(P_{old})\,) \quad mit\ \alpha \in [0,1] \qquad (3.11)$$

Mittels iterativer Anwendung dieser Formel, wird der Polygonzug nun sukzessive dem Umriss angepasst. Dies zeigt auch die Abbildung 3.6 *(c)*. Wichtig ist hierbei die Wahl einer passenden Schrittweite τ. Ein zu großer Wert erhöht die Fehlerquote und ein zu kleiner Wert treibt den Berechnungsaufwand in die Höhe. Ein geeigneter Wert ist auch hier experimentell festzustellen. $Dist(P_{old})$ steht für die Distanz des aktuellen Punktes, die mittels eines Texture Lookups im Distanzfeld ausgelesen wird. Durch die Multiplikation des Gradienten und der Distanz wird die Anzugskraft je nach Distanz, entweder verstärkt oder abgeschwächt. Zudem ist es wichtig, dass der Gradient dadurch nochmal invertiert wird wenn der betreffende Punkt eine negative Distanz hat. Andernfalls würde sich ein Punkt in

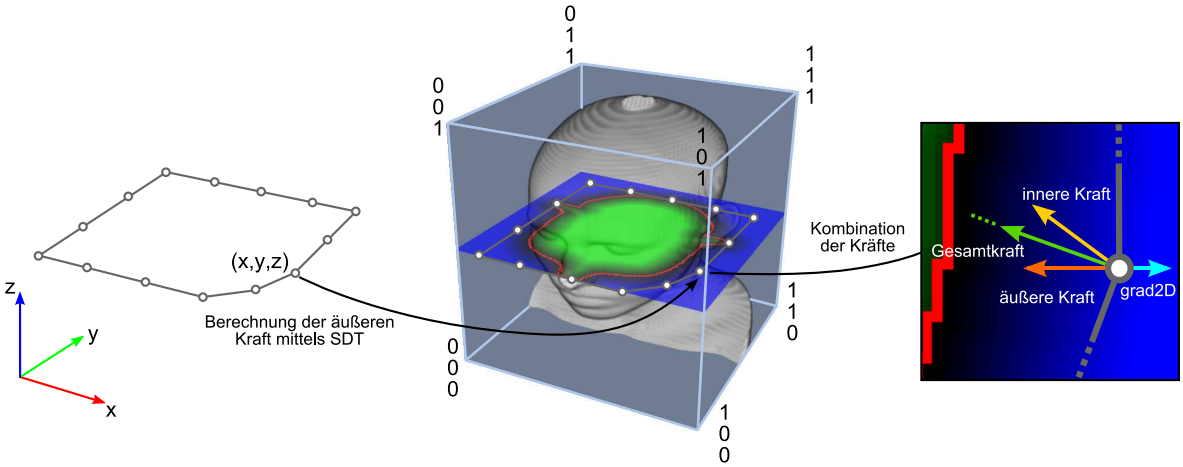


Abbildung 3.7: Anpassung eines Polygons im schematischen Überblick.

negativen Regionen nicht dem Rand annähern, sondern entfernen.

Abstände der Konturen

Die Position der ersten Kontur wird in Abhängigkeit der Abtastrichtung mittels der entsprechenden minimalen Komponente der AABB bestimmt. Das heißt, wenn das Volumen in Z-Richtung abgetastet wird, dann werden mittels des minimalen Z-Werts der AABB die Min/Max-Punkte der entsprechenden Schicht gewählt.

Nach der Bestimmung der ersten Kontur werden die weiteren Konturen detektiert. Eine uniforme Verteilung der Schichten führt bei Volumen die, innerhalb nur weniger Schichten, starke Änderungen in ihrer Metrik aufweisen allerdings dazu, dass unterschiedlich lange Verbindungskanten erzielt werden. Dies hat implizit zur Folge, dass die resultierenden Dreiecke sehr stark variieren. Um dieses Problem etwas zu entschärfen, wird das zu approximierende Objekt schichtweise abgetastet. Dabei wird in jeder Ebene eine sehr grobe Kontur bestimmt. Mittels dieser groben Kontur und der zuvor gefundenen Kontur ist es möglich die zukünftigen Längen der Verbindungskanten abzuschätzen. Wenn die Abschätzung ergibt, dass die Verbindungskanten in etwa der gewünschten Kantenlänge (vgl. Formel 3.5) entsprechen, dann wird die detaillierte Kontur für diese Ebene bestimmt. Andernfalls wird zu der nächsten Schicht gewechselt und diese geprüft.

Abschließende Konturen

Da die Konturen nur an den äußeren Umrissen angepasst werden, müssen noch weitere Konturen zum verschließen der beiden Enden bestimmt werden. Damit im Anschluss an die Triangulierung eine geschlossene Dreiecksoberfläche zur Verfügung steht.

Das Verschließen der Oberfläche erfolgt nach dem Prinzip, dass die beiden äußersten Konturen jeweils kopiert und etwas geschrumpft werden. Dieser Vorgang wiederholt sich so lange, bis die äußersten Konturen zu klein zum Schrumpfen werden. In diesem Fall wird ein letzter Punkt eingefügt, der mit allen Punkten der angrenzenden Kontur verbunden wird und somit zum Verschließen der Oberfläche führt. Der Schrumpffaktor wird hierbei so gewählt, dass die Abstände der geschrumpften Konturen annähernd der Kantenlänge entsprechen, die durch den Benutzer gewünscht wird (vgl. Formel 3.5). Daraus wird auch ersichtlich, wann eine Kontur zu klein für den Schrumpfvorgang ist. Dies ist genau dann der Fall, wenn keine weitere Kontur mehr mit dem gewünschten Abstand eingefügt werden kann.

Bewertung des Zwischenergebnisses

Diese Art der Konturendetektion hat die Einschränkung, dass lediglich die Kontur zu einem Objekt bestimmt werden kann. Tritt beispielsweise der Fall auf, dass mehrere Objekte vorhanden sind, so wird sich die aktive Kontur um alle diese Objekte legen. Eine jeweils separate Detektion ist auf diese Weise nicht möglich. Es sei denn, es würde zuvor entsprechend ein Polygonzug für jedes einzelne Objekt initialisiert. Doch das soll hier nicht weiter verfolgt werden, da das Ziel dieses Schritts be-

kanntlich nicht in der Bestimmung der exakten Rekonstruktion, sondern in der groben Approximation des Objektes liegt. Diese grobe Approximation soll durch ein umhüllendes Dreiecksnetz erfolgen. Die beschriebene Einschränkung ist daher, im Kontext dieser Arbeit, als ein Vorteil zu betrachten. Zudem können hierdurch zwei Probleme der Oberflächenrekonstruktion ausgeschlossen werden. Denn aufgrund der Tatsache, dass in jeder Schicht nur eine Kontur vorhanden sein kann, ergibt sich implizit, dass die nur schwer automatisch lösbaren Probleme der Korrespondenz und der Verzweigung vernachlässigt werden können.

Abschließend bleibt noch zu sagen, dass mit dieser Methode, anders als beispielsweise im Falle eines Verfahrens mit *Gradient Vector Flow*[XP97], nicht jede konvexe oder konkave Region detailliert erfasst wird. Doch auch diese Tatsache bringt einen wertvollen Nutzen mit sich. Denn daraus ergibt sich, dass die Konturenpaare sich nicht so stark voneinander unterscheiden, wodurch eine gute Voraussetzung für eine stabile Triangulierung gegeben ist. Letztlich soll es auch die Aufgabe der späteren Anpassung sein, weitere Details zu erfassen.

3.3.3 Verbinden der Konturen

Wie aus der anfänglichen Problemstellung hervorgeht, muss das Verfahren auch Konturen erfolgreich zu einer Mantelfläche verbinden können, die aufgrund diverser Unterschiede mehrdeutige Verbindungsmöglichkeiten aufweisen. Die Grundlagen haben gezeigt, dass das Problem der Triangulierung im Allgemeinen auf einen gerichteten Graphen reduziert wird, auf dem dann eine Graphensuche durchgeführt wird, um die letztliche Mantelfläche zu bestimmen.

Graphensuche

In der Literatur werden die Verfahren der Graphensuche in der Kategorie der Optimierungsverfahren geführt. Populäre Arbeiten zu dieser Art der Verbindungssuche, sind hier unter anderem die Arbeiten von Keppel [Kep75] und Fuchs et al. [FKU77]. Keppel nutzt dabei eine Heuristik zur Bewertung der Verbindungen, um die Metrik des innerhalb der Konturen liegenden Volumens zu maximieren. Dieses Verfahren hat allerdings Schwierigkeiten bei nicht konvexen Objekten, was zur Folge hat, dass Konturen vor der Verarbeitung gegebenenfalls zunächst in Teilabschnitte unterteilt werden müssen [Kep75]. Fuchs et al. nutzen dagegen keine Heuristik, sondern den Flächeninhalt der möglichen Dreiecke, um eine minimale Oberfläche zu suchen. Dieses Verfahren produziert gewöhnlich gute Ergebnisse, wenn die Positionen und Radien der Konturen normalisiert werden [Mey94].

Die auf einer Graphensuche basierenden Verfahren haben im Allgemeinen, aufgrund der Suchproblematik, eine erhöhte Laufzeit, bieten dafür aber in der Regel vergleichsweise gute Triangulierungsergebnisse.

Greedy-Prinzip

Dieses erhöhte Laufzeitverhalten von Algorithmen mit Graphensuche führte unter anderem dazu, dass Verfahren entwickelt wurden, die nach dem Greedy-Prinzip vorgehen und daher eine lineare Laufzeit

haben [Mey94]. Greedy-Verfahren versuchen globale Probleme möglichst gut und schnell durch lokale Bewertungsfunktionen zu lösen. Hierbei können getroffene Entscheidungen nicht mehr rückgängig gemacht werden. Das heißt, ein einmal eingeschlagener Weg muss weiterverfolgt werden.

Meyers nennt hier vor allem die Arbeiten von Christiansen und Sederberg [CS78] und von Ganapathy und Dennehy [GD82]. Christiansen und Sederberg beschreiben eine Methode, die von den zwei möglichen Folgekanten immer diejenige wählt, die die kürzere Verbindungskante in der Mantelfläche ergibt. Ganapathy und Dennehy verwenden dagegen ein Verfahren, dass die normalisierten Streckenfortschritte auf den Konturen bestimmt. Die Folgekanten werden hierbei immer so gewählt, dass der Fortschritt so gleich wie möglich bleibt.

Im Fall von ähnlichen Konturen führen die genannten Greedy-Verfahren zu akzeptablen und vor allem zu schnellen Lösungen. Doch sobald die Konturen signifikante Unterschiede aufweisen, sind die Resultate sehr schlecht oder sogar unbrauchbar, wie die Abbildung 3.8 verdeutlicht. Es ist sehr gut zu erkennen, dass die lokale Bewertung nicht immer zum gewünschten Ergebnis führt. Manchmal kann es nämlich sein, dass eine kürzeste Verbindung immer zum gleichen Polygonzug führt. Auf diese Weise werden die neuen Kanten beispielsweise immer von der ersten zur zweiten Kontur eingefügt, was zwangsläufig zu Selbstüberschneidungen führt (vgl. Abb. *3.8(b)*). Bei der Bewertungsfunktion von Ganapathy und Dennehy geschieht dies in der Regel nicht. Denn hierbei ist garantiert, dass die Verbindungen zwischen den Polygonen wechseln. Doch dies kann dazu führen, dass das Verfahren zwar mit einer guten Triangulierung startet, doch irgendwann dazu neigt gescherte Kanten einzufügen (vgl. Abb. *3.8(c)*).

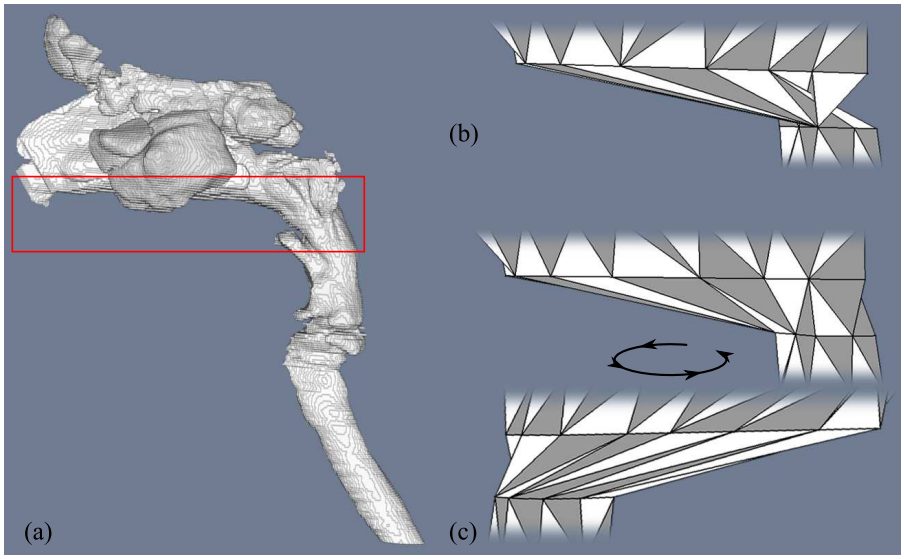


Abbildung 3.8: Die binären Segmentierungsdaten einer Luftröhre *(a)*. Die rot markierte Region führt Greedy-Verfahren schnell an ihre Grenzen, wie *(b)* und *(c)* zeigen. In *(b)* wurde die kürzeste Kantenlänge und in *(c)* der normierte Streckenfortschritt als Bewertungsfunktion genutzt.

Verfahren zur Bestimmung der Korrespondenz von Punkten

In den zuvor genannten Arbeiten wird jeweils nur ein Kriterium zur Bewertung herangezogen, um zu entscheiden, welche Verbindung als nächstes gewählt wird. Die Probleme der Greedy-Verfahren zeigen, dass die Verwendung eines Kriteriums nicht in jeder Situation zu einem gewünschten Ergebnis führt. Die aufwendigeren Optimierungsverfahren bieten hier mehr Stabilität, besitzen aber den Nachteil einer höheren Laufzeit. Meyers stellte des Weiteren fest, dass im Allgemeinen zu wenig Informationen genutzt werden, um schwierige Fälle gut lösen zu können [Mey94].

Die Arbeit von Klinski v. et al. [vKGDT99] nutzt drei Kriterien zur Bestimmung eines Bewertungskriteriums. Dabei führen sie den „Degree of Correspondence“ (DOC) ein, der den Grad der Übereinstimmung der Umrisslinien in zwei betrachteten Punkten beschreibt. Der DOC beinhaltet:

- die Orientierungsdifferenz (OD),
- die Vektorwinkeldifferenz (VWD),
- und die Distanz (Dist).

Optimale Werte sind hier eine minimale Distanz, eine Orientierungsdifferenz von null Grad und ein Vektorwinkel von 90 Grad [vKGDT99]. Die Resultate der Kriterien werden so normiert, dass sie ihr Maximum bei dem jeweiligen optimalen Wert annehmen. Durch die Addition der einzelnen Ergebnisse, wird letztlich der Grad der Korrespondenz einer Verbindung berechnet. Je größer hierbei das Resultat des DOC ist, desto besser ist die Korrespondenz einer Verbindung. Die aufgeführten Kriterien werden im Verlauf des nachfolgenden Abschnitts weiterführend betrachtet.

Bewertungskriterium

Das Konzept mehrere Kriterien zur Bestimmung der Korrespondenz zu nutzen, wird auch innerhalb der vorliegenden Arbeit adaptiert verwendet und als Bewertungsfunktion genutzt. Der DOC wird hierbei noch um einen Test auf Bijektivität (Bijek) und um das Kriterium der Fortschrittsdifferenz (FD) erweitert. Somit ergibt sich folgende Formel für den DOC:

$$DOC(i,j) = F_{Dist}(i,j) + F_{OD}(i,j) + F_{VWD}(i,j) + F_{FD}(i,j) + F_{Bijek}(i,j) \tag{3.12}$$

Die Punkte der jeweiligen Konturen werden im weiteren Verlauf mit P und Q bezeichnet und mittels i und j indiziert. Die einzelnen Funktionen des DOC liefern normierte Ergebnisse im Intervall $[0,1]$ und werden gleich gewichtet. Wobei bekanntlich gilt: Je größer ein Resultat ist, desto besser ist die entsprechende Korrespondenz. Nachfolgend sollen diese Funktionen nun genauer beschrieben werden.

Distanz Die Distanz ist der einfachste Parameter und drückt die euklidische Distanz zwischen zwei Punkten aus.

$$F_{Dist}(i,j) = 1 - \frac{\|Q_j - P_i\| - min\left\|\overline{QP}\right\|}{max\left\|\overline{QP}\right\| - min\left\|\overline{QP}\right\|} \tag{3.13}$$

Wobei $min\left\|\overline{QP}\right\|$ und $max\left\|\overline{QP}\right\|$ die minimal und maximal möglichen Kantenlängen zwischen den Konturen darstellen. Mit ihnen wird die aktuelle Kantenlänge $\|Q_i - P_i\|$ normiert.

Orientierungsdifferenz Die Orientierungsdifferenz beschreibt den Winkel zwischen den Tangenten der betrachteten Punkte. Dazu werden auf den beiden Polygonzügen zunächst die beiden Tangenten mittels der zentralen Differenzen berechnet.

$$\vec{t}_{P_i} = \frac{(P_{i+1} - P_{i-1})}{\|(P_{i+1} - P_{i-1})\|}; \quad \vec{t}_{Q_j} = \frac{(Q_{j+1} - Q_{j-1})}{\|(Q_{j+1} - Q_{j-1})\|} \tag{3.14}$$

Nachdem die Tangenten ermittelt worden sind, wird anhand des Skalarprodukts der gewünschte Winkel berechnet und in das gewünschte Intervall umgerechnet.

$$F_{OD}(i,j) = \frac{\langle \vec{t}_{P_i}, \vec{t}_{Q_j} \rangle + 1}{2} \tag{3.15}$$

Vektorwinkeldifferenz Im Idealfall sollten die Tangenten der beiden Punkte normal zu der prüfenden Kante stehen. Die Vektorwinkeldifferenz drückt nun die gemittelte Abweichung von dieser Norm aus.

$$F_{VWD}(i,j) = 1 - \frac{\left|\left\langle \vec{t}_{P_i}, \vec{k}_{QP} \right\rangle\right| + \left|\left\langle \vec{t}_{Q_j}, \vec{k}_{QP} \right\rangle\right|}{2} \quad \text{mit } \vec{k}_{QP} = \frac{Q_j - P_i}{\|Q_j - P_i\|} \tag{3.16}$$

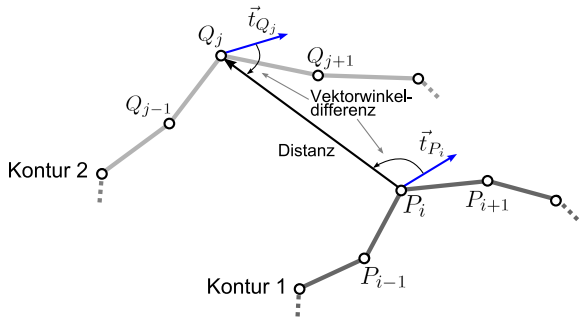


Abbildung 3.9: Anschauungsbeispiel zur Verdeutlichung der Beschreibung der DOC-Kriterien.

Bijektivität Der Parameter der Bijektivität basiert auf der Idee von Tiede et al. [TBWH85] und funktioniert wie nachfolgend beschrieben: Zu jedem Knoten eines Polygons wird mittels der euklidischen Distanz der nächstgelegene Punkt auf dem benachbarten Polygonzug ermittelt. Die gleiche Berechnung erfolgt nochmals in umgekehrter Richtung. Wenn sich zwei Punkte nun gegenseitig als nächstgelegene Punkte finden, so liegt eine bijektive Korrespondenz vor. Die

Bewertung erfolgt dabei wie folgt:

$$F_{Bijek}(i,j) = \begin{cases} 1 & wenn: \ (corspnd(P_i) = Q_j) \ \& \ (corspnd(Q_j) = P_i) \\ 0 & sonst \end{cases} \tag{3.17}$$

Dabei steht $corspnd(x)$ jeweils für den ermittelten korrespondierenden Punkt. Die einfache Verwendung der euklidischen Distanz kann gerade bei stark unterschiedlichen Konturen bekanntlich zu inakzeptablen Ergebnissen, in Form von Selbstüberschneidungen, führen. Damit sich dieser Fehler nicht auch bei dem Bijektivitätstest reproduziert, wird die Kontur zuvor in vier korrespondierende Teilbereiche unterteilt. Eine Verbindungskante kann den Bijektivitätstest nur dann bestehen, wenn die Punkte der Konturen auch in korrespondierenden Teilbereichen liegen. Zur Erinnerung: Bei der Bestimmung der Konturen wird ein Rechteck als Ausgangssituation verwendet, dessen vier Kanten für die Detektion unterteilt werden. Dadurch, dass die Rechtecke aller Konturen immer gleich ermittelt werden, können durch das Merken der jeweiligen Eckindizes vier korrespondierende Teilbereiche geschaffen werden. Diese Beschränkung verhindert vor allem, dass Verbindungskanten den Bijektivitätstest bestehen, die aufgrund großer Unterschiede falsch gewählt würden (vgl. Abb. 3.10).

(a) (b) (c)

Abbildung 3.10: Teilbereiche einer Kontur *(a)*; Korrespondierende Bereiche zweier Konturen *(b)*; Kürzeste Verbindungskante (ohne Teilbereiche) und der daraus folgende Triangulierungsfehler *(c)*

Fortschrittsdifferenz Für jedes Punktepaar wird, wie auch in der Arbeit von Ganapathy und Dennehy [GD82], der jeweilige normierte Streckenfortschritt auf der Kontur ermittelt. Damit dieser Streckenfortschritt bestimmt werden kann, muss allerdings zuvor ein initiales Punktepaar bestimmt werden, bei dem der Fortschritt anfangs gleich null ist. Als Startkante wird hier die kürzeste Verbindungskante genutzt, die während der Untersuchung auf Bijektivität ermittelt wurde. Auch hierbei gilt somit die Einschränkung auf die korrespondierenden Teilbereiche.

$$u_i = \frac{\sum_{k=1}^{i} \|P_k - P_{k-1}\|}{\sum_{k=1}^{m} \|P_k - P_{k-1}\|} \text{ mit } [1 \leq i \leq m]; \quad v_j = \frac{\sum_{k=1}^{j} \|Q_k - Q_{k-1}\|}{\sum_{k=1}^{n} \|Q_k - Q_{k-1}\|} \text{ mit } [1 \leq j \leq n] \tag{3.18}$$

Wobei m und n jeweils die Anzahl der Knoten im Polygonzug sind. Die Differenz zwischen u_i

und v_j wird letztlich als Fortschrittsdifferenz bezeichnet und wie folgt ermittelt:

$$F_{FD}(i,j) = 1 - |u_i - v_j| \tag{3.19}$$

Bestimmung der Triangulierung

Sobald jeder Knoten des Graphen mittels des DOC bewertet wurde, kann mit der eigentlichen Triangulierung der zwei benachbarten Konturen begonnen werden. Ursprünglich sollte die Triangulierung mittels einer Graphensuche erfolgen. Doch praktische Tests zeigten, dass die zuvor genannten Probleme des Greedy-Prinzips durch die komplexere Bewertungsfunktion vermieden werden. Daher kann eine zeitaufwendige Graphensuche eingespart werden.

Zu Beginn wird das initiale Punktepaar der Fortschrittsdifferenz als Startkante gewählt, wodurch die beiden Konturen an dieser Stelle erstmals verbunden werden. Aufbauend auf dieser Kante wird der Graph nach dem Greedy-Prinzip traversiert, indem immer von den zwei möglichen Folgeknoten derjenige mit der besseren Korrespondenz gewählt wird. Auf diese Weise wird die Mantelfläche nun Schritt für Schritt geschlossen. Wurde einer der beiden Polygonzüge schon komplett traversiert, so wird nur noch die „offene“ Kontur verarbeitet und die Wahl entfällt.

3.3.4 Abschließende Betrachtung der Oberflächenrekonstruktion

Nach Durchlaufen aller zuvor beschriebenen Schritte ist eine approximierende Dreiecksoberfläche entstanden, die einen Genus null hat und das gesamte Objekt einschließt [1]. Ein Beispiel dazu zeigt die Abbildung 3.11, die das binäre Voxelgitter eines menschlichen Schädels und eine zugehörige Dreiecksoberfläche darstellt. Der Detailgrad des Dreiecksnetzes ist primär von den Parametern der

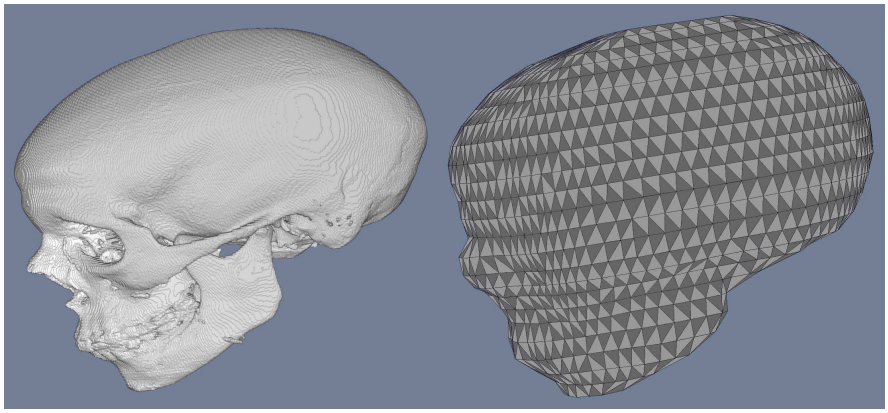


Abbildung 3.11: Beispiel einer generierten Dreiecksoberfläche.

Konturendetektion und dem Abstand der einzelnen Konturen abhängig. Ist das Ergebnis nicht zufriedenstellend, so können die Parameter modifiziert und die Oberfläche entsprechend neu bestimmt wer-

[1] Es wird vorausgesetzt, dass der Benutzer die Parameter, vor allem die Abtastrichtung und den entsprechenden Abstand der Schichten, passend gewählt hat.

den. Eine automatische Bewertung der Oberfläche erfolgt dabei allerdings nicht, sondern geschieht allein durch den Benutzer. Sobald dieser mit dem Resultat zufrieden ist, kann der Anpassungsvorgang durchgeführt werden, wodurch die entstandene Oberfläche immer weiter an das Zielobjekt angepasst wird.

3.4 Anpassen der Dreiecksoberfläche

Nach der Konstruktion der initialen Dreiecksoberfläche soll nun die weitere Anpassung des Netzes erläutert werden. Bevor jedoch die Beschreibung des eigentlichen Verfahrens der Anpassung erfolgt, wird zunächst die Zielsetzung und die damit verbundenen Probleme beschrieben.

3.4.1 Zielsetzung und Probleme

Wie in den Grundlagen beschrieben, wird zur Anpassung des Dreiecksnetzes ein physikalisches Modell verwendet, wie es beispielsweise auch beim Verpacken mit Schrumpffolie angewendet wird. Das sogenannte Shrink-Wrapping hat im vorliegenden Fall das Ziel, das approximierende Netz weiter an das Objekt anzupassen, indem eine Schrumpfung simuliert wird.

Diese Schrumpfung erfolgt mittels einer inneren und äußeren Kraft, ähnlich wie bei der vorhergehenden Konturendetektion. Jedoch ist das Verfahren, aufgrund von signifikanten Unterschieden zwischen dem Zielobjekt und dem aktuellen Netz, nicht frei von Problemen. Dies liegt daran, dass in stark unterschiedlichen Regionen die Punkte des Netzes weit von der Zieloberfläche entfernt sind. Die sukzessive Anpassung solcher Regionen führt nun oftmals dazu, dass Bereiche im Netz dazu neigen zu konvergieren oder zu divergieren, was im Wesentlichen zu zwei Problemen führt (vgl. Abb. 3.12):

Artefakte Durch die Tendenz zum Konvergieren und Divergieren entstehen Verzerrungen im Netz, die Artefakte in Form von Falten hervorrufen.

Selbstüberschneidungen Ein gravierenderes Problem stellt die Problematik der Selbstüberschneidungen dar, da diese die Weiterverarbeitung zu einem Tetraedergitter unmöglich machen. Die Selbstüberschneidungen treten immer dann auf, wenn Punkte sich überlagern oder bei der Anpassung kreuzen. Im Prinzip sind die Selbstüberschneidungen eine Extremform der zuvor beschriebenen Artefakte.

Mitunter ist die Qualität der Anpassung somit auch stark abhängig von dem initialen Dreiecksnetz, da eine gute Approximation zu geringeren Unterschieden zwischen dem Netz und dem Zielobjekt führt. Um eine möglichst gute Ausgangssituation für die Anpassung zu schaffen, sollte der Benutzer daher dieses Wissen bei der Konstruktion der Dreiecksoberfläche beachten.
Letztlich wird es sich jedoch immer um eine Approximation der Oberfläche handeln und daher sind die Probleme nicht alleine durch eine geeignete Konstruktion zu lösen. Vielmehr müssen diese Probleme bei der eigentlichen Anpassung berücksichtigt werden. Daher soll im Folgenden das Verfahren der Anpassung vorgestellt werden.

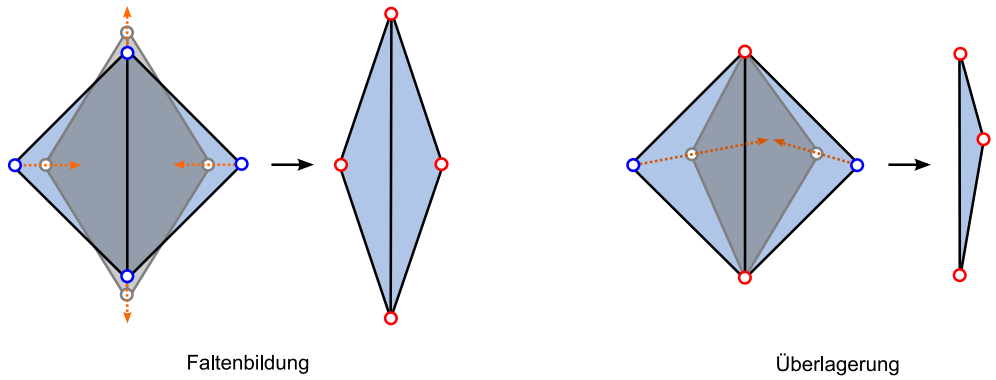


Abbildung 3.12: *Links:* Faltenbildung durch Konvergenz und Divergenz. *Rechts:* Überlagerung zweier Dreiecksflächen aufgrund von Konvergenz. *(rot/blau = fixierter/beweglicher Punkt)*

3.4.2 Verfahren der Anpassung

Im Grunde ähnelt das Shrink-Wrapping bekanntlich dem Verfahren der aktiven Konturen. Dementsprechend werden auch hier eine innere und eine äußere Kraft verwendet.

Die Dämpfung

Aus den Grundlagen ist bekannt, dass die vorhergehenden Arbeiten ([KVLS99, JK02, KCC+05]) die innere Kraft mittels der Formel der approximierenden Laplacian-Glättung $\mathcal{L}$ berechnen (siehe Formel 2.7). Diese innere Kraft hat das Bestreben, die Punkte des Netzes uniform zu verteilen und schwächt somit die Auswirkungen der äußeren Kraft ab. Daher kann diese Kraft auch als Dämpfung des Dreiecksnetzes betrachtet werden. Ohne eine Dämpfung würde die Anpassung gegebenenfalls zu Selbstüberschneidungen führen, die das Netz bekanntlich unbrauchbar machen würden.

Die Arbeiten von [JK02] und [KCC+05] verwenden im Speziellen nur den tangentialen Anteil von $\mathcal{L}$, um Schrumpfungseffekte bestmöglich zu vermeiden. $\mathcal{L}$ hat die Eigenschaft, einen Punkt immer zum geometrischen Mittelpunkt seiner lokalen Nachbarn zu bewegen, wodurch auch die Biegungsenergie des Netzes minimiert wird. Durch die alleinige Verwendung des tangentialen Anteils, wird das Potential zur Minimierung der Biegungsenergie äußerst stark eingeschränkt (vgl. Abb. 3.13 *(a)*). Hierdurch ist der Einfluss der äußeren Kraft auf die Punkte verhältnismäßig stark und oftmals gänzlich ohne Gegenkraft. Dementsprechend ist eine detaillierte und schnelle Anpassung möglich (vgl. Abb. 3.13 *(b)*). Das geringe Potential zur Minimierung der Biegungsenergie, führte im Rahmen dieser Arbeit jedoch leider dazu, dass die gewünschte innere Stabilität des Netzes vereinzelt nicht erreicht wurde. Konkret führte die Anpassung, in Regionen mit großer Krümmung und sehr kleinen Dreiecken, oftmals zu Selbstüberschneidungen.

Eine höhere innere Stabilität wird folglich dadurch erreicht, dass $\mathcal{L}$ nicht nur auf den tangentialen Anteil eingeschränkt wird. Dies führt im Gegenzug allerdings auch dazu, dass die Anpassung in kleineren Schritten verläuft und schneller stagniert. Eine weitere Anpassung kann jedoch durch hinzufügen lokaler Details (Punkte) ermöglicht werden. Dieses Hinzufügen der Punkte erfolgt in einem

Nachbearbeitungsschritt, der im Anschluss an dieses Kapitel beschrieben wird.

(a) (b)

Abbildung 3.13: *2D-Beispiele zur Veranschaulichung:* Unterschiede zwischen $\mathcal{L}$ und $\mathcal{L}_t$ *(a)*. Auswirkungen auf die Anzugskraft *(b)*.

Die Folgeposition eines Punktes, durch den Einfluss der Dämpfung, wird nun wie folgt bestimmt:

$$P_{relax} = P_{old} + \|sign(Dist(P_{old}))\| \cdot \mathcal{L}(P_{old}) \tag{3.20}$$

mit

$$sign(Dist(P_{old})) = \begin{cases} 1 & wenn:\ 0 < Dist(P_{old}) \\ 0 & wenn:\ 0 = Dist(P_{old}) \\ -1 & wenn:\ 0 > Dist(P_{old}) \end{cases}$$

Punkte die das Zielobjekt erreicht haben, werden von der Anzugskraft nicht mehr weiter bewegt. Damit auch sichergestellt ist, dass die Dämpfung keinen Einfluss mehr auf solche Punkte nimmt, wird $\mathcal{L}(P_{old})$ noch mit dem Betrag von $sign(Dist(P_{old}))$ multipliziert.

Die Anzugskraft

Die Anzugskraft wird, wie auch bei der Konturendetektion (vgl. Kap. 3.3.2), mittels des umgekehrten Gradienten erreicht. Jedoch geschieht die Anpassung hier nicht in der Ebene, sondern im dreidimensionalen Raum, wodurch mittels der zentralen Differenzen der dreidimensionale Gradient ($grad_{3D}$) bestimmt wird.

Bei Objekten mit tiefen Hohlräumen neigt die Methode mit dem Gradienten allerdings dazu, alle Punkte die sich direkt über diesen Vertiefungen oder Löchern befinden, zu den Rändern dieser Regionen zu bewegen (siehe Abb. 3.14). Wenn die Dämpfung in solchen Fällen zu gering gewählt ist, kommt es an den Rändern zu Überlagerungen der Punkte. Um dieses Problem zu verhindern, wird der invertierte Gradient vorbeugend mit dem Richtungsvektor der Laplacian-Glättung $\mathcal{L}$ (siehe Formel 2.7) kombiniert.

$$P_{attract} = P_{old} + \tau \cdot \|Dist(P_{old})\| \cdot ((1-\alpha)(-grad_{3D}(P_{old}) \cdot sign(Dist(P_{old}))) + \alpha \mathcal{L}(P_{old})) \tag{3.21}$$

$\mathcal{L}$ wird die Punkte immer zu dem lokalen Mittelpunkt ihrer 1-Nachbarschaft ziehen, wodurch die

Selbstüberschneidungen, auch bei niedrig gewichteter Dämpfung, verhindert werden. Der Einfluss kann jedoch mit $\alpha \in [0, 1]$ variiert werden. Die Distanz eines Punktes zur Zieloberfläche $Dist(P_{old})$ wird mittels eines Texture Lookups im Distanzfeld ermittelt. Durch die Multiplikation des invertierten Gradienten und $sign(Dist(P_{old}))$ wird die korrekte Orientierung der äußeren Kraft auch für negative Regionen sichergestellt. Wenn die letztliche Richtung der Anzugskraft ausgemacht ist, wird ihre Stärke durch die Multiplikation mit dem Betrag der $Dist(P_{old})$ festgelegt. Hierdurch wird erreicht, dass weit entfernte Punkte sich schneller, beziehungsweise mit größeren Schritten, zur Oberfläche bewegen. Im Gegenzug bewirkt das Annähern an die Oberfläche somit eine Verlangsamung und damit eine feinere Anpassung. Für ein zufriedenstellendes Resultat ist nicht zuletzt auch eine geeignete Schrittweite τ, analog zur Konturendetektion, zu wählen.

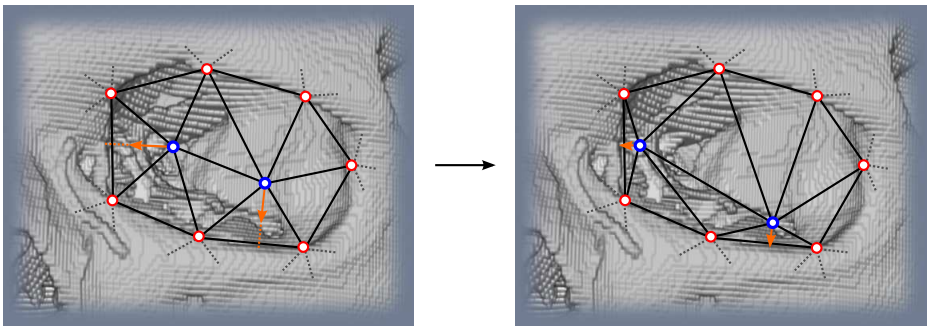


Abbildung 3.14: Problematik der Selbstüberschneidungen bei Vertiefungen: Bei geringer Dämpfung und alleiniger Verwendung des Gradienten *(orange)*, würden die blauen Punkte innerhalb weniger Iterationen zum Rand wandern und somit zu Überlagerungen führen.

Der Anpassungsvorgang

Die Anpassung wird iterativ, mittels der Kombination der inneren und der äußeren Kraft durchgeführt:

$$P_{new} = (1 - \beta) \cdot P_{attract} + \beta \cdot P_{relax} \quad mit\ \beta \in [0, 1] \tag{3.22}$$

Die Formeln 3.20 und 3.21 zeigen, dass für die Dämpfung und die Anzugskraft jeweils die Folgeposition bestimmt wird. Zwischen diesen neuen Positionen wird mittels des Dämpfungsfaktors β linear interpoliert, um die resultierende Position zu erhalten. Ein Anpassungsvorgang beginnt hierbei mit $\beta = 0.5$ und wird von Anpassungsdurchlauf zu Anpassungsdurchlauf verringert, bis ein spezifizierter Minimalwert erreicht wird. Somit ergibt sich, das vor allem in den ersten Schritten der Anpassung die Verteilung der Punkte optimiert wird. Mit fortschreitender Zahl der Durchläufe steigt dann mehr und mehr der Einfluss der Anzugskraft, wodurch das Netz letztlich schrittweise angepasst wird.

3.4.3 Optimierung der Punktverteilung

Wie bereits im Kapitel (3.2.1) zur Vorverarbeitung der Voxeldaten erwähnt, ist es optional möglich die initiale Dreiecksoberfläche, basierend auf dem geglätteten Datensatz, zu konstruieren. Wodurch

zwei wichtige Vorteile gewonnen werden:

1. Detailreduktion → erleichtert die Konturendetektion
2. Oberflächenvergrößerung → verbesserte Verteilung der Oberflächenpunkte

Der erste Vorteil ist intuitiv recht gut nachvollziehbar, da implizit klar ist, dass es einfacher ist zu einem detailreduzierten Objekt den Umriss zu finden, als zu einem unveränderten und damit detailreichen. Der Vorteil der Oberflächenvergrößerung ist hingegen nicht selbsterklärend und soll daher an dieser Stelle näher betrachtet werden.

Initiale Optimierung der Verteilung der Oberflächenpunkte

Die Anpassung der Dreiecksoberfläche basiert normalerweise auf dem Distanzfeld der unveränderten Originaldaten oder auf einem Datensatz der weniger geglättet ist als bei der Konstruktion der Oberfläche. Demzufolge ist bei einer vergrößerten Dreiecksoberfläche kein Punkt vollständig angepasst, da ein Lookup im Distanzfeld folglich immer einen Distanzwert ungleich null liefern wird. Aus dem vorhergehenden Abschnitt ist bekannt, dass gerade während der ersten Anpassungsdurchläufe die Verteilung der Punkte optimiert wird. Dadurch, dass noch kein Punkt vollständig angepasst ist, wirkt sich die Optimierung der Verteilung auf alle Punkte aus und die gesamte Oberfläche wird somit etwas „entspannt“ (vgl. Abb. 3.15).

Wie beschrieben ist diese Optimierungsmöglichkeit optional und vor allem dann zu empfehlen, wenn das initiale Dreiecksnetz spitze Dreiecke enthält, deren Punkte bereits fixiert sind. In diesen Fällen wird innerhalb der ersten Anpassungsdurchläufe eine deutlich verbesserte Ausgangslage geschaffen.

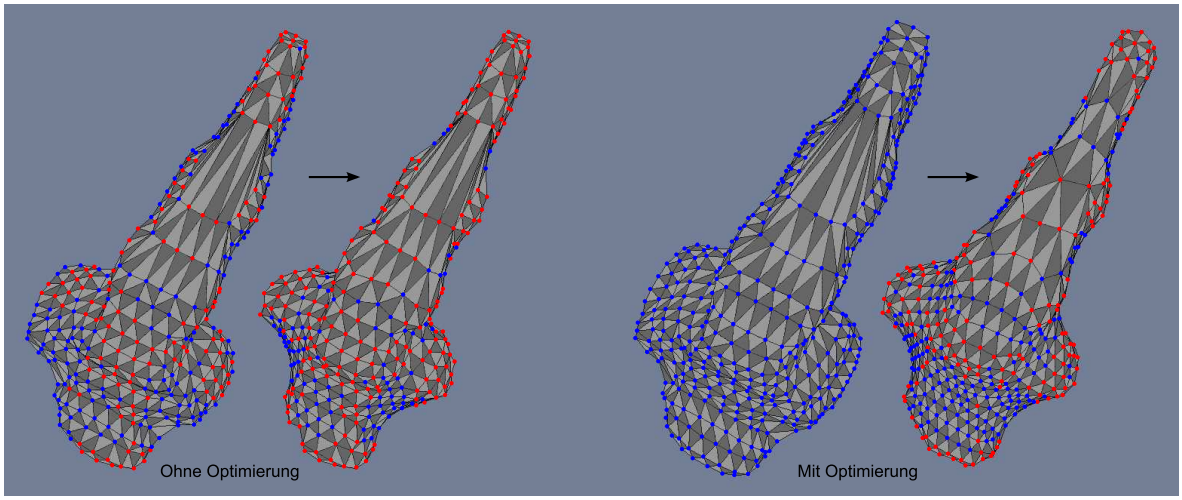


Abbildung 3.15: Auswirkungen der Optimierung am Beispiel der Luftröhre. Die Anpassungen unterscheiden sich nur in der initialen Größe der Dreiecksoberfläche. *(rot/blau = fixiert/pos. Distanz)*

3.4.4 Abschließende Betrachtung der Anpassung

Während eines Anpassungsvorgangs bewirkt die Dämpfung in den ersten Durchläufen, durch die Tendenz die Punkte uniform zu verteilen, eine hohe Stabilität des Netzes. Durch die schrittweise Verringerung der Dämpfung wird ein zunehmender Einfluss der Anzugskraft erreicht, was zur weiteren Anpassung des Netzes führt. Aus Gründen der Stabilität ist die Anzugskraft eine Kombination aus invertiertem Gradienten und dem Richtungsvektor der Laplacian-Glättung. Wodurch der Grad der Anpassung, auch bei geringer Dämpfung, limitiert ist.

Aufgrund der Unterschiede zwischen dem Dreiecksnetz und der Zieloberfläche wird es trotz allem nicht möglich sein die Folgen der Konvergenz und Divergenz vollends zu verhindern. Vielmehr stellt dieses Verfahren eine erste Kompensation der Probleme dar. Dementsprechend wird die Oberfläche nach einem Anpassungsvorgang aus überdehnten und stark geschrumpften Regionen bestehen.

Diese Problematik hat dazu geführt, dass die Dreiecksoberfläche nach einem Anpassungsvorgang zunächst mittels eines Nachbearbeitungsschritts optimiert wird. Nach der erfolgreichen Optimierung wird die Oberfläche dann weiter, nach dem oben beschriebenen Verfahren, angepasst. Diese Vorgehensweise wird wiederholt, bis keine weitere Anpassung mehr erfolgt oder ein ausreichend gutes Resultat zur Verfügung steht.

3.5 Nachbearbeitungsschritt zur Optimierung des Dreiecksnetzes

In dem vorhergehenden Kapitel hat sich herausgestellt, dass die Unterschiede zwischen dem anzupassenden Dreiecksnetz und dem Zielobjekt bei der Anpassung dazu führen, das Regionen im Netz divergieren oder konvergieren. Die Auswirkungen wurden zuvor ausreichend dargestellt und dabei wurde zur weiterführenden Kompensation dieser Probleme der, in diesem Kapitel beschriebene, Nachbearbeitungsschritt zur Optimierung des Netzes genannt. Zu Beginn soll ein Überblick über die elementaren Transformationsoperatoren zur Optimierung eines Dreiecksnetzes verschafft werden. Im Anschluss daran wird die Anwendung der Operatoren im Kontext dieser Arbeit beschrieben.

3.5.1 Elementare Transformationsoperatoren zur Optimierung eines Dreiecksnetzes

Der Nachbearbeitungsschritt benötigt insgesamt drei lokale Operatoren, wie sie auch beispielsweise in der Arbeit von Hoppe et al. [HDD+93] zur Netz-Optimierung verwendet werden.

Edge Collapse Ein Edge Collapse kollabiert eine betreffende Kante und somit auch ihre zwei adjazenten Dreiecksflächen. Auf diese Weise können zu kurz gewordene Kanten aus dem Netz entfernt werden. Zu beachten ist dabei allerdings, dass ein Edge Collapse nur dann ausgeführt werden darf, wenn alle Punkte der adjazenten Dreiecke mindestens eine Valenz von vier aufweisen. Andernfalls führt eine Edge Collapse nämlich zu Überfaltungen im Dreiecksnetz.

Edge Split Bei einem Edge Split wird eine Kante, durch Einfügen eines Punktes, in zwei Kanten unterteilt. Dies hat implizit auch zur Folge, dass die adjazenten Dreiecke entsprechend geteilt

werden müssen. Durch die Unterteilung können einem Netz weitere Details hinzugefügt werden.

Edge Swap Der Edge Swap bezeichnet das „Drehen“(Flippen) einer betreffenden Kante. Dabei wird das Ziel verfolgt, dass Verhältnis zwischen In- und Umkreis der jeweiligen adjazenten Dreiecke zu verbessern. Wobei gilt, dass dieses Verhältnis im Optimalfall maximal ist. Auf diese Weise kann die Triangulierung und somit die visuelle Qualität eines Netzes verbessert werden.

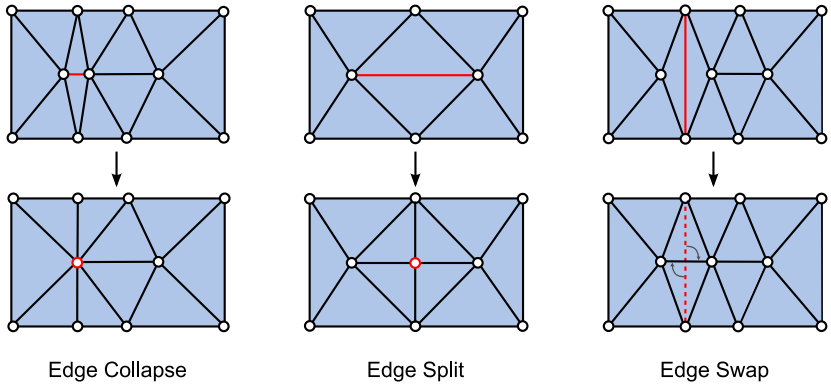


Abbildung 3.16: Elementare Transformationsoperatoren: Edge Collapse, Edge Split und Edge Swap.

Im nachfolgenden Abschnitt werden diese Operatoren nun dazu eingesetzt, die durch die Anpassung entstehenden Probleme zu kompensieren.

3.5.2 Optimierung des Netzes

Die Dämpfung verhindert bei der Anpassung Selbstüberschneidungen und zu starkes Divergieren, doch bekanntlich lassen sich überdehnte und stark geschrumpfte Regionen nicht vermeiden. Kobbelt et al. [KVLS99] beschreiben, dass gleiches auch beim realen Verpacken mit Schrumpffolie beobachtet werden kann. Auch hier ist, wie in den Grundlagen thematisiert, der signifikante Unterschied zwischen dem Objekt und der Hülle für die Probleme verantwortlich. Im Fall der Schrumpffolie kann dieses Problem auch wie folgt formuliert werden: In gedehnten Regionen war bei der Anpassung zu wenig, in stark geschrumpften Bereichen hingegen zu viel Material vorhanden.

Optimal wäre es also, wenn während des Schrumpfvorgangs die Folie noch angepasst werden könnte. Dies ist in der Realität nicht der Fall und auch nicht notwendig. Im Fall dieser Arbeit ist dies allerdings, zur weiteren Kompensation der genannten Probleme, durchaus wünschenswert und zudem notwendig für die Stabilität. Dementsprechend soll die Dreiecksoberfläche neben der eigentlichen Anpassung auch gezielt optimiert werden. Das heißt, es sollen Details im Netz gezielt entfernt und hinzugefügt werden können.

Diese Optimierung kann in den Bereich der allgemeinen Optimierung und der adaptiven Unterteilung aufgeteilt werden. Der Nachbearbeitungsschritt beginnt dabei immer mit der allgemeinen Optimierung des Netzes, die vor allem zu kurz gewordene Kanten entfernt. Im Anschluss daran folgt

eine Überprüfung des Netzes auf Dreiecksflächen die eine Unterteilung benötigen, um der Oberfläche weitere Details hinzuzufügen. Diese beiden Bereiche werden nun im Weiteren beschrieben.

Allgemeine Optimierung des Netzes

Die allgemeine Optimierung verfolgt das Ziel Dreiecke aus dem Netz zu entfernen, die aufgrund der Anpassung degeneriert sind. Als degeneriert wird ein Dreieck bezeichnet, wenn sein Flächeninhalt, bedingt durch eine zu kurze oder zu lange Kante, nahezu null ist. Zum Entfernen dieser Dreiecke werden, ähnlich wie es unter anderem in der Arbeit von Jung et al. [JSC04] gemacht wird, der Edge Collapse und der Edge Swap angewendet. Durch das Entfernen dieser sehr spitzen Dreiecke, wird zum einen das visuelle Ergebnis verbessert und zum anderen das Risiko der Selbstüberschneidung enorm reduziert. Die Optimierung mittels der beiden Operatoren erfolgt nun wie folgt:

- Der **Edge Collapse** kollabiert Kanten die eine Länge $l \leq \epsilon_{ZeroLength}$ haben
- Der **Edge Swap** flippt Kanten bei einem minimalen Winkel von $\alpha \leq \epsilon_{ZeroAngle}$

Wobei mit den Toleranzwerten $\epsilon_{ZeroLength}$ und $\epsilon_{ZeroAngle}$ festgelegt wird, ab wann eine Kante zu kurz und ein Winkel entsprechend zu klein ist. Folglich kann mit diesen beiden Werten die Empfindlichkeit der Optimierung beeinflusst werden. Zu beachten ist, dass ein degeneriertes Dreieck mittels Edge

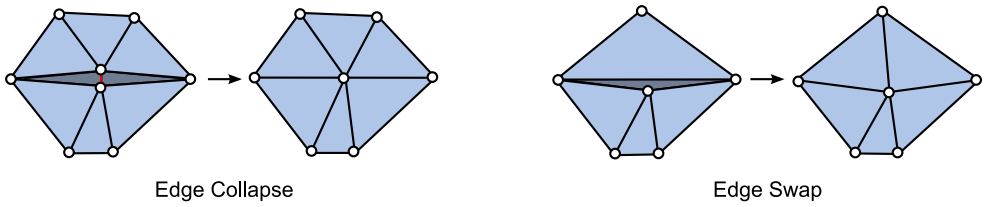


Abbildung 3.17: Entfernen degenerierter Dreiecke.

Swap nur dann entfernt wird, wenn das benachbarte Dreieck nicht degeneriert ist (siehe Abb. 3.17). Andernfalls kann es sein, dass durch den Swap eine zu kollabierende Kante entsteht. Wenn sich jedoch aus zwei degenerierten Dreiecken nur zwei andersartig degenerierte Dreiecke ergeben, so bleibt nur noch die Optimierung durch die Dämpfung im nächsten Anpassungsdurchlauf.

Der Edge Collapse ist unkomplizierter und führt, durch das definitive Kollabieren der zu kurzen Kanten, zu der gewünschten Reduktion des lokalen Detailgrads im Netz.

Adaptive Unterteilung der Dreiecksflächen

Nachdem zuvor die Reduktion der Details besprochen wurde, soll es nun um die gezielte Erweiterung der Oberflächendetails gehen. Dies geschieht mittels einer adaptiven Unterteilung (engl. „Subdivision“) der Dreiecksflächen, um ein lokales Hinzufügen von Details zu ermöglichen.

Hierzu wird zunächst überprüft, welche Dreiecke eine weitere Anpassung und somit eine Unterteilung benötigen. Diese Überprüfung erfolgt anhand eines mehrfachen und zufälligen Samplings der Dreiecksflächen. Dabei wird für jedes Sample die Distanz im Distanzfeld, durch einen Texture

Lookup ermittelt. Im Anschluss werden diese Distanzen dann zu einer durchschnittlichen Distanz gemittelt. Ein Dreieck wird nun zum Unterteilen markiert, wenn es eine durchschnittliche Distanz von $Dist_{Average} \geq Dist_{Defined}$ hat und nicht zu klein geworden ist. Mit Hilfe von $Dist_{Defined}$ kann dabei die Toleranz spezifiziert werden, ab wann ein Dreieck als gut oder entsprechend schlecht angepasst gilt. Eine detaillierte Erklärung zu diesem Samplingverfahren erfolgt in der Beschreibung der Implementierung in Kapitel 4.4.3.

Durch die zusätzliche Bedingung, dass der Flächeninhalt eines Dreiecks nicht kleiner als ein definierter Toleranzwert sein darf, wird zudem auch die Einstellung einer Detailstufe (engl. „Level of Detail“ - LOD) ermöglicht.

Die Kanten der zu unterteilenden Dreiecke werden mittels des Edge Split Operators geteilt. Bekanntlich sind davon auch die adjazenten Dreiecke betroffen. Dies gilt selbst für diejenigen die vielleicht ursprünglich keine Unterteilung benötigten. Dementsprechend müssen bei der Unterteilung, wie in Abbildung 3.18 dargestellt, drei Fälle beachtet werden: Ein Dreieck wurde...

- ... zum Unterteilen markiert → es wird in vier Dreiecke unterteilt *(a)*
- ... nicht markiert, aber zwei seiner Nachbarn → es wird in drei Dreiecke aufgeteilt *(b)*
- ... nicht markiert, jedoch einer seiner Nachbarn → es wird einmal geteilt *(c)*

Wobei der erste Fall implizit auch für nicht markierte Dreiecke gilt, wenn diese drei zu unterteilende Nachbarn haben.

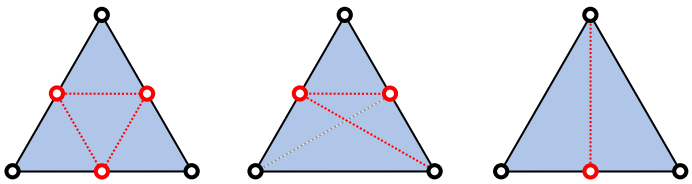


Abbildung 3.18: Adaptive Unterteilung der Dreiecksflächen: Vollständig unterteiltes Dreieck *(a)*, Dreieck mit zwei zu unterteilenden Kanten *(b)*, Dreieck mit einer zu teilenden Kante *(c)*.

Auf diese Weise wird sichergestellt, dass die Unterteilungen sich auf die markierten Dreiecke und deren jeweils direkte Nachbarschaft beschränken.

3.5.3 Abschließende Betrachtung der Optimierung

Der Nachbearbeitungsschritt nimmt lokale Optimierungen im Dreiecksnetz vor. Dabei hat sich herausgestellt, dass durch das Entfernen von degenerierten Dreiecken die Stabilität und auch das visuelle Resultat verbessert werden. Zudem ermöglicht die adaptive Unterteilung gezielt genau jene Regionen im Netz mit Details zu erweitern, die dieses auch benötigen. Wodurch die weitere Anpassung solcher Bereiche ermöglicht wird.

Anzumerken ist jedoch noch, dass die allgemeine Optimierung nur Dreiecke betrifft die Extremfälle darstellen. Dreiecke die nur „leicht degeneriert“ sind werden nicht berücksichtigt. Somit werden grobe Falten, so wie es für die Art des Verfahrens typisch ist, leider auch weiterhin noch bestehen bleiben.

3.6 Erzeugung der Tetraedergitter

Das Resultat der vorhergehenden Schritte ist eine approximierende Dreiecksoberfläche, zu der nun in diesem letzten Schritt das gewünschte Tetraedergitter generiert werden soll.

Die letztliche Generierung erfolgt mit Hilfe der freien Software *Gmsh*. *Gmsh* ist ein automatischer 3D Finite Elemente Netzgenerator, der für akademische Testfälle entworfen wurde. Das Programm besteht insgesamt aus vier Modulen: Geometrie, Netzgenerierung, Gleichungsauflöser und Nachbearbeitung. Die Eingabe erfolgt entweder interaktiv über die grafische Benutzeroberfläche oder mittels ASCII-kodierter Textdateien und der *Gmsh* eigenen Skriptsprache [GR08].

Die Generierung der Tetraedergitter erfolgt nach dem Prinzip der Eingabe-Verarbeitung-Ausgabe (EVA), wobei die Verarbeitung mittels *Gmsh* als Black Box erfolgt. Dieses Prinzip soll zu Beginn in Form einer knappen Übersicht dargestellt werden, bevor auf die eigentliche Ein- und Ausgabe eingegangen wird.

Von der Geometrie zum Tetraedergitter

Zur Generierung eines Tetraedergitters wird die erzeugte Dreiecksoberfläche als Geometrie, mit Hilfe einer Textdatei, an *Gmsh* übergeben. Das Modul der Netzgenerierung wird dann zu dieser eingelesenen Geometrie ein entsprechendes 3D Netz erzeugen. Da die Geometrie in Form einer Dreiecksoberfläche vorliegt, wird das resultierende 3D Netz ein Tetraedergitter sein.

Eingabe und Ausgabe

Die Geometrie wird mittels der *Gmsh* eigenen Skriptsprache in einer ASCII-kodierten Textdatei gespeichert. Dabei werden die einzelnen Elemente der Geometrie in der nachfolgenden Reihenfolge definiert und erhalten alle eine einmalige Identifikationsnummer (vgl. Abb. 3.19):

$$\text{Punkte}^{①} \Rightarrow \text{Linien}^{②} \Rightarrow \text{Flächen}^{③} \Rightarrow \text{Volumen}^{④}$$

Zudem besteht die Möglichkeit zu spezifizieren, wie fein die einzelnen resultierenden Elemente bei der Netzgenerierung werden sollen. Hierzu kann jeder Punkt der Geometrie mit einer gewünschten charakteristischen Länge (Characteristic length) versehen werden. Die Größe der einzelnen Elemente des Netzes werden dann durch linearer Interpolation dieser charakteristischen Längen berechnet.

Die auf diese Weise erstellte Textdatei dient als Eingabe für Gmsh. Dementsprechend kann das Modul der Netzgenerierung nun das gewünschte Tetraedergitter zu dieser Geometrie generieren.

Nach erfolgreicher Verarbeitung wird die Ausgabe des Gitters mittels des *Gmsh* eigenen MSH

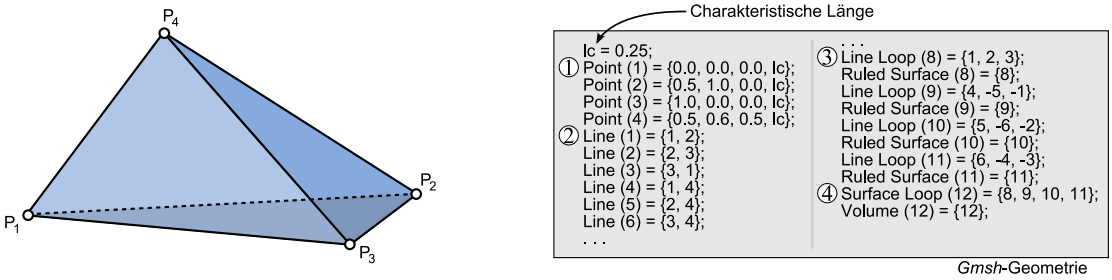


Abbildung 3.19: Definition einer *Gmsh*-Geometrie am Beispiel eines Tetraeders.

Dateiformats, wahlweise ASCII oder binär kodiert, erfolgen. Das Tetraedergitter kann nun beispielsweise durch Konvertieren in andere Dateiformate beliebig eingesetzt werden.[2]

3.7 Zusammenfassung

In diesem Kapitel wurden die wichtigsten Aspekte des Ansatzes zur Erzeugung von Tetraedergittern theoretisch betrachtet: Das Verfahren beginnt mit dem Vorverarbeitungsschritt, der im Wesentlichen das vorzeichenbehaftete Distanzfeld berechnet und die umhüllenden Geometrien ermittelt.

Die eigentliche Erzeugung erfolgt mit der Konstruktion einer approximierenden Dreiecksoberfläche. Hierbei werden zunächst planare aktive Konturen mit Hilfe der umhüllenden Geometrien und der SDT bestimmt. Diese Konturen werden schließlich durch einen Triangulierungsschritt miteinander zu einer Mantelfläche verbunden. Als Bewertungskriterium, für die Wahl der Verbindungskanten, wird hier die Korrespondenz der Punktepaare genutzt. Nach dem Verbinden aller Konturen steht die grobe Dreiecksoberfläche für die weitere Anpassung bereit.

Die Dreiecksoberfläche wird nun weiter an das Zielobjekt angepasst, indem ein Shrink-Wrapping Verfahren angewendet wird. Dabei werden mit der äußeren Kraft, in Form der Anzugskraft, die Punkte des Netzes näher an das Zielobjekt bewegt. Als Anzugskraft wird primär der umgekehrte Gradient genutzt, der für jeden Punkt im Distanzfeld ermittelt wird. Die alleinige Anwendung der äußeren Kraft würde allerdings schnell zu Selbstüberschneidungen im Dreiecksnetz führen. Um dieses zu verhindern wird eine innere Kraft zur Stabilisierung verwendet. Diese innere Kraft basiert auf dem Prinzip der Laplacian Glättung, die einen Punkt immer zum geometrischen Mittelpunkt seiner lokalen Nachbarschaft bewegt. Sie dämpft sozusagen die Anzugskraft und hat somit eine stabilisierende Wirkung.

Mit einem Nachbearbeitungsschritt werden lokale Regionen des Netzes, die zu sehr gedehnt oder gestaucht sind, optimiert. Die Optimierung erfolgt indem entsprechend Details hinzugefügt respektive entfernt werden. Das Entfernen trägt zur Stabilität des Netzes bei, während das Hinzufügen eine weiterführende Anpassung ermöglicht. Der Anpassungsvorgang und der Nachbearbeitungsschritt werden jeweils im Wechsel durchlaufen, bis ein gewünschtes Resultat erreicht ist oder keine Verbesserung der

[2] Eine genaue Erläuterung der Skriptsprache und des MSH Dateiformats würde leider den Rahmen dieser Arbeit sprengen und wäre auch nicht zweckmäßig. Daher sei für weitere Informationen auf die Dokumentation der Software verwiesen, die auf der Webseite der Autoren einzusehen ist.

Anpassung mehr eintritt.

Wurde eine gute Approximation der Oberfläche gefunden, so wird die Geometrie an die freie Software *Gmsh* übergeben, um letztlich das benötigte Tetraedergitter zu generieren.

Kapitel 4

Implementierung

Dieses Kapitel greift die wesentlichen Aspekte des zuvor beschriebenen Verfahrens auf und beschreibt deren Implementierung. Die Gliederung orientiert sich, wie auch das vorhergehende Kapitel, im Wesentlichen an den Schritten des Programmablaufs. Jedoch wird die Erklärung zu Beginn um ein Kapitel, zur Erläuterung der zugrunde liegenden Repräsentation des Dreiecksnetzes, erweitert. Danach wird die vorhergehende Gliederung durch die Darstellung der Konstruktion des initialen Dreiecksnetzes, der Anpassung, der Optimierung und dem letztlichen Export zu *Gmsh* fortgesetzt. Allerdings wird die Beschreibung der Anpassung, der Optimierung und der Export nach *Gmsh* nicht wie zuvor separat, sondern gemeinsam erfolgen. Abschließend werden die wichtigsten Aspekte der Implementierung noch in einem letzten Kapitel zusammengefasst.

4.1 Datenstruktur

Die Generierung eines Dreiecksnetzes und vor allem die Anpassung und Optimierung macht es erforderlich, dass das Netz effizient repräsentiert wird. Im Folgenden wird zunächst ein knapper Überblick zur Umsetzung der Datenstruktur gegeben. Ein Großteil der Verarbeitung erfolgt auf der GPU und daher wird im Anschluss auch erläutert, wie die geometrischen und topologischen Informationen in Texturen gespeichert werden.

4.1.1 Repräsentation des Dreiecksnetzes

Für die effiziente Repräsentation des Dreiecksnetzes wird die Winged-Edge Datenstruktur (vgl. Kap. 2.1.2) verwendet. Nicht nur die Effizienz, sondern auch die komfortable Möglichkeit ein Netz in das Geometrieformat von *Gmsh* zu konvertieren sind ausschlaggebende Faktoren für die Wahl dieser Struktur. In beiden Fällen werden orientierte Kanten verwendet. Aufgrund dieser Ähnlichkeit ist es auf einfache Weise möglich das repräsentierte Netz zu konvertieren.

Wie bereits in den Grundlagen erläutert, verwendet diese Repräsentation jeweils eine Struktur zur Speicherung der Punkte, Kanten und Flächen. Die einzelnen Strukturen werden bei der Umsetzung

um ein paar Attribute erweitert, um die Implementierung von Operationen zu vereinfachen (siehe Listing A.1). Die meisten Attribute sind bereits selbsterklärend oder werden im späteren Verlauf der Erklärung aufgegriffen. Weshalb an dieser Stelle keine explizite Erläuterung der einzelnen Attribute erfolgen soll.

Die Repräsentation erfolgt durch die Klasse `CWingedEdge`, indem für jede der drei oben genannten Entitäten eine Liste geführt wird:

- `QList<WE_Vertex> m_listVertices //Liste der Punkte`
- `QList<WE_Face> m_listFaces //Liste der Flächen`
- `QList<WE_Edge> m_listEdges //Liste der Kanten`

Des Weiteren sind hier verschiedene obligatorische Funktionen zur Verwaltung der Elemente und zur schnellen Ermittlung entsprechender Zugehörigkeitsbeziehungen definiert. Beispiele hierzu sind unter anderem das Hinzufügen weiterer Entitäten, das Auflisten der Kanten einer Fläche oder die Ermittlung der 1-Nachbarschaft eines Punktes. Das heißt, dass alle grundlegenden Funktionen, die direkt auf dem Dreiecksnetz angewendet werden, hier definiert sind.

4.1.2 Bereitstellung der Geometrie und Topologie durch Texturen

Ein Großteil der Verarbeitung erfolgt auf der GPU. Dazu gehört unter anderem die Ermittlung der Konturen, die Anpassung des Netzes und auch die Evaluierung der Dreiecke bezüglich der Notwendigkeit einer Optimierung. Damit eine Verarbeitung durch die GPU möglich ist, müssen die geometrischen und topologischen Informationen im lokalen Speicher der Grafikkarte abgelegt werden. Eine Übersicht über die in diesem Verfahren verwendeten Texturen verschafft die Tabelle 4.1. Die Generierung der einzelnen Texturen erfolgt hierbei durch die zentrale Klasse `CTextureManager`, die statische Methoden zur Erzeugung bereit hält (wie z.B. `generateVertexTexture(...)`).

Textur zur Speicherung der Geometrie

Die Speicherung der Geometrie erfolgt indem die Punktkoordinaten der `m_listVertices` in einer zweidimensionalen RGB-Textur (*Vertices*) gespeichert werden. Hierbei wird pro Punkt ein Texel belegt, indem die *xyz*-Werte eines Punktes in den jeweiligen RGB-Komponenten geschrieben werden. Die Einfügereihenfolge entspricht dabei der Reihenfolge der Punkte in `m_listVertices`.

Bei der Textur handelt es sich um eine quadratische Textur, deren Größe von der Anzahl der Punkte ($numVertices$) abhängig ist. Dementsprechend wird für die Breite, beziehungsweise Höhe der Textur die kleinstmögliche Zweierpotenz 2^n gewählt, deren Quadrat die Speicherung aller Punkte ermöglicht.

$$numVertices \leq \min(\ (2^n)^2\ ),\quad n \in \mathbb{N} \tag{4.1}$$

Nach der Generierung der Textur (*Vertices*) können im Shader die Koordinaten eines Punktes mittels eines Texture Lookups abgerufen werden. Hierzu muss lediglich der Index des Punktes bekannt sein

und in die entsprechenden Texturkoordinaten (*u,v*) umgerechnet werden. Die Umrechnung erfolgt in Abhängigkeit von der Texturgröße. Damit diese Umrechnung nicht immerzu wiederholt werden muss, wird die Berechnung einmal bei der Erzeugung der topologischen Texturen durchgeführt und gespeichert.

Textur		**Beschreibung**
Vertices	[2D RGB float]	Punktkoordinaten der Geometrie
NeighborInfo	[2D RGBA float]	Informationen zur 1-Nachbarschaft eines Punktes
NeighborIndex	[2D RGBA float]	Texturkoordinaten zu benachbarten Punkten
TriangleTex1 TriangleTex2	[2D RGB float] [2D RGB float]	Texturkoordinaten zu den Punkten eines Dreiecks
RandSamplingInfo	[2D RGBA float]	Informationen zum zufälligen Mehrfachsampling
RandTexCoords	[2D RGBA float]	Zufällige Texturkoordinaten
RandBarycentricWeights	[2D RGB float]	Zufällige baryzentrische Gewichte
SignedDT	[3D RGBA float]	Vorzeichenbehaftetes Distanzfeld

Tabelle 4.1: Eine Übersicht zu den verwendeten Texturen.

Texturen zur Speicherung der 1-Nachbarschaften

Neben der Geometrie (*Vertices*) werden auch die 1-Nachbarschaften der Punkte gespeichert, indem zwei zweidimensionale RGBA-Texturen (*NeighborInfo/NeighborIndex*) angelegt werden.

NeighborIndex In dieser Textur werden die Texturkoordinaten zu den Nachbarpunkten abgelegt, so dass die einzelnen Koordinaten der Nachbarpunkte durch Texture Lookups abgerufen werden können. In jedem Texel stehen vier Komponenten zur Verfügung und somit können die Texturkoordinaten s_n, t_n von maximal zwei Punkten v_n gespeichert werden. Für die Speicherung der 1-Nachbarschaft eines Punktes ist ein Texel allerdings viel zu wenig. Da die Valenz von Punkt zu Punkt variiert, ist es zudem notwendig, eine dynamische Methode zur Speicherung der Nachbarschaftsbeziehungen zu verwenden. Daher werden die Texturkoordinaten s_n, t_n der einzelnen Nachbarpunkte v_n hintereinander weg, über mehrere Texel verteilt, geschrieben. Die Anzahl der Texel $numTexel_n$, die für die Speicherung der 1-Nachbarschaft eines Punktes benötigt werden, berechnet sich somit aus der Valenz eines Punktes:

$$numTexel_n = \frac{Valenz_n + (Valenz_n \bmod 2)}{2} \tag{4.2}$$

Die Bestimmung der Texturgröße erfolgt nach dem gleichen Prinzip wie bei der *Vertices*-Textur, nur das die Größe hier von der Summe aller benötigten Texel abhängig ist.

NeighborInfo Die Textur ist genauso groß wie die *Vertices*-Textur und jeder Texel enthält Nachbarschaftsinformationen zu einem Punkt. Die Einfügereihenfolge richtet sich ebenfalls, wie bei den *Vertices*, nach `m_listVertices`. Daher ergibt sich, dass ein Punkt und seine zu-

gehörigen Nachbarschaftsinformationen mit identischen Texturkoordinaten erreicht werden. Pro Punkt werden in einem Texel Informationen zu der Valenz, die Größe der *NeighborIndex*-Textur und die Texturkoordinaten zum ersten zugehörigen Texel der *NeighborIndex*-Textur gespeichert. Diese Informationen werden dabei wie folgt auf die RGBA-Komponenten verteilt. Die R-Komponente wird mit der Valenz und die BA-Komponenten mit den Texturkoordinaten belegt. Da es sich bei *NeighborIndex* um eine quadratische Textur handelt, ist für die Speicherung der Texturgröße eine einzelne Komponente ausreichend. Somit wird die Texturgröße in der freien G-Komponente gespeichert.

Die Verwendung dieser zwei Texturen und der *Vertices*-Textur ermöglicht, dass die 1-Nachbarschaft zu jedem Punkt abgerufen und verarbeitet werden kann:

```
// Texture Lookup der aktuellen Nachbarschaftsinformationen
float4 currInfoNB = tex2D(NeighborInfo, texCoord);
// Texturkoordinaten für den Lookup in NeighborIndex
float2 currNeighborIndexTexCoord = currInfoNB.ba;

for(int nCount=0; nCount < currInfoNB.r;)
{
  // Lookup des aktuellen Texels
  currSample = tex2D(NeighborIndex, currNeighborIndexTexCoord);
  // Bestimmung der Texturkoordinaten zum Lookup in den Vertices
  (0 == (nCount%2) )? currNeighborTexCoord = currSample.rg,
                    : currNeighborTexCoord = currSample.ba;
  // Texture Lookup des Nachbarn
  currVtx = tex2D(Vertices, currNeighborTexCoord);
  // BEGIN Verarbeitung
  // ...
  // END Verarbeitung
  nCount++;
  // Wechsel zum nächten Texel, wenn zwei Punkte verarbeitet wurden.
  if(0 == (nCount%2) )
  {
    currNeighborIndexTexCoord.x += 1.0/currInfoNB.g;
    if(1.0 < currNeighborIndexTexCoord.x) {
      currNeighborIndexTexCoord.x -= 1.0;
      currNeighborIndexTexCoord.y += 1.0/currInfoNB.g;
    }
  }
} // END for(...)
```

Listing 4.1: Der prinzipielle Texture Lookup der benachbarten Punktkoordinaten.

Mit den Texturkoordinaten in *NeighborInfo.ba* findet der erste Texture Lookup in *NeighborIndex*

statt und durch die Valenz ist bekannt, wieviele Punkte hintereinander stehen. So kann mittels einer Schleife ein Nachbarpunkt nach dem anderen verarbeitet werden. Aus der Größe der Textur wird der benötigte Offset berechnet, um zum nächsten Texel wechseln zu können.

Texturen zur Speicherung der Dreiecke

Auch die Dreiecksflächen sollen in Texturen gespeichert werden. Generell verläuft die Speicherung der in `m_listFaces` enthaltenen Dreiecke analog zu der Erzeugung der *Vertices*-Textur. Da jedoch für einen Punkt v_n immer die Texturkoordinaten s_n, t_n abgelegt werden und ein Dreieck aus drei Punkten besteht, müssen für ein Dreieck somit sechs Werte gespeichert werden. Daher ist es auch hier nicht möglich den Speicherbedarf durch ein Texel abzudecken. Aufgrund der Tatsache, dass der Bedarf für jedes Dreieck konstant ist, kann durch die Verwendung zweier RGB-Texturen (*TriangleTex1/TriangleTex2*) genügend Speicher zur Verfügung gestellt werden.

Die Lookup-Koordinaten des ersten Punktes werden in den RG-Komponenten der ersten Textur gespeichert. Die Texturkoordinaten s_n, t_n des zweiten Punktes werden hingegen auf die beiden Texturen verteilt, so dass s_n in *TriangleTex1.b* und t_n in *TriangleTex2.r* geschrieben wird. Die letzten beiden verbleibenden Komponenten von *TriangleTex2* werden dann durch den dritten Punkt belegt.

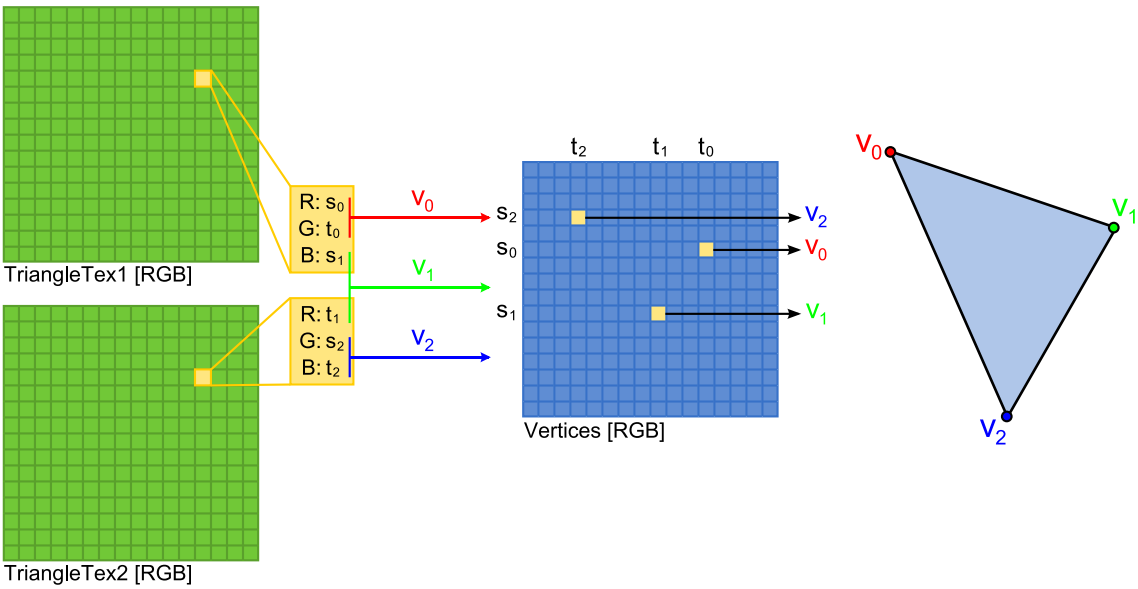


Abbildung 4.1: Der prinzipielle Texture Lookup der Punktkoordinaten für ein Dreieck.

4.2 Vorverarbeitung der Voxeldaten

Die obligatorische Vorverarbeitung der Voxeldaten umfasst, wie aus Kapitel 3 bekannt, mehrere Teilschritte: Optional ist es zu Beginn möglich das Voxelgitter zu glätten. Die Hauptaufgabe der Vorver-

arbeitung ist die Erzeugung eines dreidimensionalen und vorzeichenbehafteten Distanzfeldes, durch Anwenden des JFA. Doch bevor die Propagation durchgeführt werden kann, muss das Voxelgitter zunächst in einem vorhergehenden Schritt geeignet initialisiert werden. Bei der Initialisierung werden dann zugleich die für die später folgende Konturendetektion benötigten umhüllenden Geometrien des Volumens ermittelt. Vor der Erklärung der einzelnen Teilschritte beginnt dieses Kapitel mit der Beschreibung des generellen Verarbeitungsvorgangs.

4.2.1 Allgemeiner Verarbeitungsvorgang

Der Verarbeitungsvorgang wird durch die Klasse `CVolumeDataManager` gestartet, indem die Methode `processVolData()` aufgerufen wird. Die einzelnen Verarbeitungsschritte werden dann durch die Klasse `CVolumeDataProcessing` koordiniert und durchgeführt. Die Verarbeitung erfolgt hier primär auf der GPU, weshalb auch das Konfigurieren der einzelnen Shaderprogramme zu den Aufgaben dieser Klasse gehört.

Die Berechnungen erfolgen mittels Multipass-Rendering durch die Verwendung zweier Instanzen von `CVolumeBuffer`. Diese Klasse definiert ein dreidimensionales Framebuffer Objekt (FBO) und realisiert auch zugleich das Rendering mittels der Methode `runProgram(...)`.

```
void CVolumeBuffer::runProgram(QCgShaderProgram *fragProg)
{
  // ... Aktiviere FBO und fragProg
  for(int nSliceZ=0; nSliceZ < m_dDepth; nSliceZ++)
  {
    // Anhängen des aktuellen Slices an das FBO
    glFramebufferTexture3DEXT( GL_FRAMEBUFFER_EXT,
                               GL_COLOR_ATTACHMENT0_EXT,
                               GL_TEXTURE_3D,
                               m_pFBO->texId(0),
                               0, // MipLevel
                               nSliceZ );
    // Rendere das aktuelle Slice
    drawSlice( (nSliceZ+0.5)/m_dDepth );
  }
  // Deaktiviere FBO und fragProg ...
}
```

Listing 4.2: Rendern in ein dreidimensionales FBO im CVolumeBuffer.

Ein Berechnungsprozess wird durch das Aufrufen dieser Methode gestartet, dabei wird das anzuwendende Shaderprogramm per Zeiger übergeben. Die beiden Instanzen werden im Ping-Pong-Verfahren verwendet und tauschen daher nach jedem Rendering Pass ihren Verwendungszweck. Das heißt, wenn ein Volumen-Buffer als Eingabe genutzt wird, so ist der andere zugleich das Renderingziel. Im darauf folgenden Berechnungsschritt werden diese Rollen entsprechend getauscht. Das Überschreiben der

Daten ist hier kein Problem. Denn auf diese Weise wird immer nur die Eingabe des vorhergehenden Berechnungsschritts überschrieben, dessen Resultat bereits die Eingabe der aktuellen Berechnung ist.

Die gewonnenen Ergebnisse der Verarbeitung werden in einer Instanz von `CVolumeData` gespeichert und werden mit Hilfe dieses Objektes (`m_pVolData`) anderen Klassen per Zeiger verfügbar gemacht.

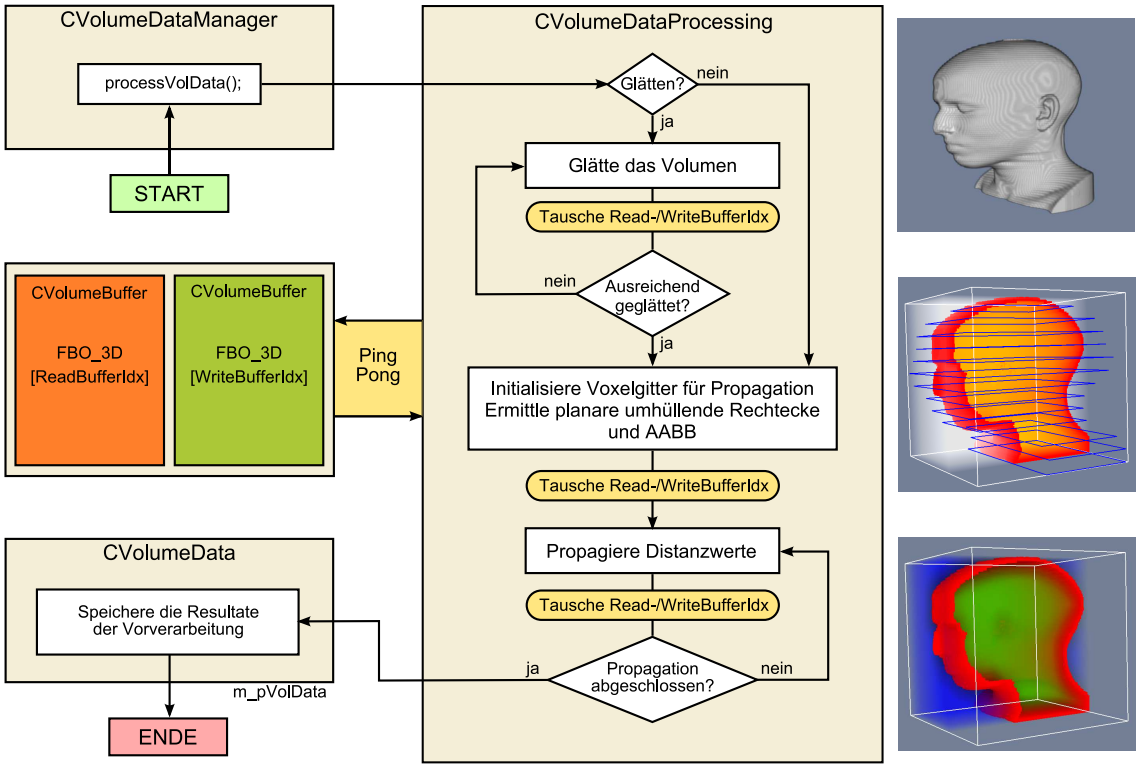


Abbildung 4.2: Ablauf der Vorverarbeitung der Voxeldaten.

4.2.2 Teilschritte der Vorverarbeitung

Jeder Verarbeitungsschritt wird mit einem separaten Fragment Programm realisiert. Wie zuvor beschrieben, werden sie zu gegebener Zeit in `CVolumeDataProcessing` konfiguriert und ausgeführt. Die Reihenfolge der Ausführung der einzelnen Verarbeitungsschritte ist in der Abbildung 4.2 dargestellt. Im Folgenden sollen diese Teilschritte nun näher betrachtet werden.

Glättung des Volumens

Die Glättung des Datensatzes wird durch Anwenden eines linearen Gauss-Filters erreicht. Lineare Filter haben im Allgemeinen die Eigenschaft bestimmte Frequenzen abzuschwächen und andere passieren zu lassen. Der Gauss-Filter kommt hierbei einem Tiefpassfilter gleich, der kleinere Strukturen vermindert und gröbere dagegen erhält.

Der Nutzen dieser optionalen Glättung besteht bekanntlich neben der Detailreduktion auch in der

Vergrößerung der initialen Oberfläche. Da beides bereits zuvor thematisiert wurde (siehe Kap. 3.4.3), sollen diese beiden Punkte hier nicht weiter thematisiert werden.

Die Umsetzung erfolgt indem mittels des Gauss-Filterkerns ($3 \times 3 \times 3$) die Nachbarschaft eines jeden Voxels gefaltet wird.

```
// ...
int nIdx = 0;  // Index zum aktuellen Gaussgewicht
// Definition des Filterkerns (fGaussian[nIdx]/39.0)
float fGaussian[] = {0, 1, 0,  1, 3, 1,  0, 1, 0,
                     1, 3, 1,  3, 9, 3,  1, 3, 1,
                     0, 1, 0,  1, 3, 1,  0, 1, 0};
// Falte den (3x3x3)-Filterkern mit dem Signal
float fCoord[] = { -1.0, 0.0, 1.0 };
for(int w=0; w<3; w++)
  for(int v=0; v<3; v++)
    for(int u=0; u<3; u++)
    {
      float3 currPosNB = texCoord + float3(fCoord[u]*fVoxelSize.x,
                                           fCoord[v]*fVoxelSize.y,
                                           fCoord[w]*fVoxelSize.z);
      // Jedes Sample wird gewichtet und aufaddiert
      result += (fGaussian[nIdx]/39.0) * tex3D(DataSet, currPosNB);
      nIdx++;
    } // END for("u")
// ...
```

Listing 4.3: Glätten des Volumens durch einen Gauss-Filter.

Der Grad der Glättung kann durch Vergrößern des Filterkerns oder durch wiederholtes Anwenden des Filters gesteigert werden. In dieser Arbeit hat der Benutzer die Möglichkeit den Grad der Glättung durch Festlegen der Zahl der Anwendungen zu bestimmen.

Initialisierung und umhüllende Geometrien

Wenn keine Glättung erwünscht ist, dann ist die Initialisierung des Voxelgitters für die Propagation der erste Schritt der Vorverarbeitung. In diesem Fall sind beide Volumen-Buffer leer und die Verarbeitung erfolgt auf Basis der Originaldaten.

Die Initialisierung sieht die Bestimmung des Objektrandes (Referenz) und der inneren sowie äußeren Regionen vor. Dementsprechend wird, wie zuvor definiert, jeder Voxel des Gitters entweder als Objekt-, Rand- oder Leer-Voxel klassifiziert (vgl. Kap. 3.2.2). Auch hierbei wird ein (3x3x3)-Filterkern verwendet, um die 1-Nachbarschaft eines Voxels zu untersuchen.

Bei der Untersuchung wird der Farbunterschied zwischen dem aktuellen Voxel und aller seiner, durch den Filterkern abgedeckten, Nachbarn ermittelt. Ist eine Differenz gefunden worden und ist der aktuelle Voxel zudem ein Leer-Voxel, so wird dieser als Rand-Voxel klassifiziert.

```
// ...
bool bIsBoundary = false;
// ... In dreifacher for-Schleife des Filterkerns (wie Gauss-Filter)
  // ... Bestimmung des Farbunterschieds
  fCurrColorDiff = length( currVoxelColor-currNeighborColor );
  // Wenn eine Differenz in den Farbwerten vorhanden ist und
  // der aktuelle Voxel ein Leer-Voxel ist...
  if( 0 != fCurrColorDiff
   && ( currVoxelColor.r == 0.0 && currVoxelColor.g == 0.0
     && currVoxelColor.b == 0.0 && currVoxelColor.a == 0.0 ) )
  { // ...dann handelt es sich um einen Rand-Voxel
    bIsBoundary = true;
  }
//...
```

Listing 4.4: Bestimmung der Rand-Voxel mittels Farbdifferenzen.

Wurde nach Untersuchung der 26 Nachbarn jedoch keine Differenz ermittelt, dann ist der Voxel von gleichartigen Voxeln umgeben und somit ist allein sein Farbwert entscheidend für seine Klassifikation. Das nachfolgende Listing zeigt das abschließende Setzen des aus der Voxel-Klassifikation hervorgehenden Resultats. Hierbei wird bei nicht Rand-Voxeln noch bekanntlich zwischen inneren und äußeren Regionen unterschieden und entsprechend eine negative oder positive maximale Distanz gesetzt.

```
// ... Setze das Resultat RGBA = (Dist, xyz-CoordsToNearestBoundary)
if( bIsBoundary ) {
  result = float4(0, 0,0,0); // Rand
} else
  (bIsEmpty)? result = float4( MAX_DIST, 0,0,0), // Äußere Region
            : result = float4(-MAX_DIST, 0,0,0); // Innere Region
```

Listing 4.5: Setzen des Resultats der Voxel-Klassifikation.

Wie das Listing 4.2 verdeutlicht, wird beim Rendern in ein dreidimensionales FBO schichtweise in Z-Richtung vorgegangen. Diese Gegebenheit wird für die Bestimmung der AABB ausgenutzt. Jede dieser Schichten wird mit `glReadPixels(...)` ausgelesen und verarbeitet. Hierbei wird jeder Eintrag des Framebuffers iterativ untersucht. Bei Finden eines Rand-Pixels werden seine Koordinaten aus dem Iterationsindex und der bekannten Größe des Framebuffers bestimmt. Die so gewonnenen Koordinaten werden mit den bislang gespeicherten minimalen und maximalen *xyz*-Werten verglichen und gegebenenfalls modifiziert (vgl. Kap. 3.2.3). Wenn alle Schichten des Volumens verarbeitet wurden, dann stehen die Min/Max-Koordinaten, die die AABB des Volumens definieren, fest und werden in `m_pVolData` gespeichert.

Zugleich wird ebenfalls die Ermittlung der umhüllenden Rechtecke durchgeführt. Von einer vertiefenden Beschreibung soll an dieser Stelle jedoch abgesehen werden, da die generelle Vorgehens-

weise bereits in Kapitel 3.2.3 ausreichend thematisiert wurde und zudem analog zu der Bestimmung der AABB verläuft. Festgehalten werden soll jedoch, dass die resultierenden Ergebnislisten, der drei Richtungen, in `m_pVolData` gespeichert werden. Zur Erinnerung: Je *x*-, *y*- und *z*-Richtung wird ein Rechteck pro Schicht in der jeweiligen Liste als Element abgelegt.

Propagation der Distanzwerte

Die eigentliche Propagation wurde bereits in den Grundlagen (Kap. 2.3) und in der Analyse (Kap. 3.2.2) umfangreich dargestellt. Daher soll die Darstellung der Implementierung an dieser Stelle auf das Nötigste beschränkt werden.

Die Propagation der Distanzwerte erfolgt mittels des Ping-Pong-Prinzips. Umgesetzt wird dabei der Jump Flooding Algorithmus von [RT06], bei dem die Schrittweite k je Durchlauf halbiert wird. In `CVolumeDataProcessing` wird die Methode `executeComputeSDT(...)` bereitgestellt, die zunächst die initiale Schrittweite berechnet und im Anschluss daran das Starten der einzelnen Propagationsdurchläufe durchführt.

```
int k = 2;
// Bestimmung der maximale Zweierpotenz für diesen Datensatz
while(k < max( nDepth, max( nWidth, nHeight ) ) ) {
  k = k << 1;
}
// Durchführen der Propagation mit variabler Schrittweite
while( 1 <= k )
{
  // Propagationsdurchgang für k
  computeSignedDT(k, bComputeSDT_3D);
  // Halbieren der Schrittweite
  k = k >> 1;
}
computeSignedDT(1, bComputeSDT_3D); // JFA+1
```

Listing 4.6: Berechnung der Schrittweite und Durchführen der Propagation.

Die eigentliche Bestimmung der euklidischen Distanz, eines jeden Voxels, erfolgt auf der GPU und auf Basis eines variablen $(3 \times 3 \times 3)$-Filterkerns. Der Filterkern definiert hierbei die Lookup Koordinaten der Nachbarn, die für den aktuellen Propagationsschritt untersucht werden.

```
float fCoordU[3];
  fCoordU[0] = -1.f * fVoxelSize.r * k;
  fCoordU[1] =  0.f * fVoxelSize.r * k;
  fCoordU[2] =  1.f * fVoxelSize.r * k;
float fCoordV[3], fCoordW[3]; // -> analog zu fCoordU
```

Listing 4.7: Der variable Filterkern des JFA.

Wenn einer dieser Nachbarn dabei eine geringere Distanz als der im Zentrum des Filterkerns liegende Voxel hat, so wird die Distanz zum Referenzpunkt (Punkt auf dem Rand) dieses Nachbarn berechnet. Sollte die dabei ermittelte Distanz geringer als die gespeicherte sein, dann wird der alte Wert durch den Neuen ersetzt.

```
float4 result = tex3D(DataSet, texCoord); // Zentraler Voxel
// ... In dreifacher for-Schleife des Filterkerns [0<=(u, v, w)<3]
  bComputeDist = false;
  // Bestimme die aktuelle Nachbarposition...
  currPosNB = texCoord + float3(fCoordU[u], fCoordV[v], fCoordW[w]);
  // ...und hole die dazugehörigen Werte
  currValueNB = tex3D(DataSet, currPosNB);

  if(currValueNB.r == 0.0) // Der Nachbar ist ein Rand-Voxel, daher
  {                        // ist currPosNB eine Referenz-Koordinate
    currRefCoord = currPosNB;
    bComputeDist = true;
  }
  else if( abs(currValueNB.r) < abs(result.r) )
  {
    // Dieser Nachbar ist kein Rand-Voxel, aber seine Distanz zum
    // Rand ist geringer als die derzeitig gespeicherte.
    currRefCoord = currValueNB.gba; // speichere Referenz-Koordinate
    bComputeDist = true;
  }
  // Wenn true: es wurde eine zu überprüfende Referenz gefunden
  if(bComputeDist)
  {
    // Berechne die aktuelle Distanz mit neuer Referenz
    float currDist = distance(vertex, (currRefCoord*TextureSize));
    // Update, wenn die neue Distanz geringer als die alte ist
    if( currDist < abs(result.r)) {
      result = float4( sign(result.r)*currDist, currRefCoord);
    }
  }
// END for(u,v,w) ...
```

Listing 4.8: Die Propagation der euklidischen Distanz.

Nach Ablauf der gesamten Propagation ist die Vorverarbeitung abgeschlossen. Das zuletzt beschriebene FBO enthält das Resultat, in Form des dreidimensionalen und vorzeichenbehafteten Distanzfeldes (*SignedDT*). Die R-Komponente eines jeden Voxels enthält dabei die geringste Distanz zum Rand des Zielobjekts. Die für die Propagation benötigten Referenz-Koordinaten sind zwar noch immer in den GBA-Komponenten enthalten, doch für die später folgende Verarbeitung sind sie nicht mehr von

Belang. Damit *SignedDT* von den nachfolgenden Verarbeitungsschritten auch genutzt werden kann, wird zum Abschluss noch die Texturgröße und auch die TexturID in `m_pVolData` gespeichert.

4.2.3 Problem der Vorverarbeitung

Wie zuvor thematisiert, erfolgt die Vorverarbeitung im Ping-Pong-Verfahren. Aufgrund dessen, dass hierbei der lesende Zugriff nicht nur auf den zum aktuellen Rasterpunkt (Pixel) zugehörigen Texel eingeschränkt werden kann, sind zwei FBOs notwendig. Die Verwendung zweier dreidimensionaler FBOs führt allerdings zu einer hohen Speicheranforderung.

Texturgröße	R	RGB	RGBA
128^3 [16 Bit]	4.096 kB	12.288 kB	16.384 kB
256^3 [16 Bit]	32.768 kB	98.304 kB	131.072 kB
512^3 [16 Bit]	262.144 kB	786.432 kB	1048.576 kB

Tabelle 4.2: Beispiele zur Speicheranforderung eines dreidimensionalen FBOs.

Der in der Tabelle 4.2 gegebene Überblick zum Speicherbedarf bezieht sich jeweils auf ein FBO und muss demgemäß noch entsprechend verdoppelt werden. Zudem ist die Bit-Präzision zu beachten. Wird diese beispielsweise von 16- auf 32-Bit erhöht, verdoppelt sich auch entsprechend die Speicheranforderung. Dementsprechend wird bei hochaufgelösten Datensätzen schnell das Limit des lokalen Grafikspeichers erreicht.

Damit jedoch auch Geometrien zu hochaufgelösten Datensätzen generiert werden können, wird in solchen Fällen das Distanzfeld auf der Grundlage eines verkleinerten Datensatzes berechnet. Der Grad der Skalierung ist hierbei abhängig von der Größe des lokalen Grafikspeichers.[1] Dies hat zweifellos zur Folge, dass eine, um den entsprechenden Skalierungsfaktor, verkleinerte Geometrie konstruiert wird. Infolgedessen wird das entstandene Dreiecksnetz bei dem Export nach *Gmsh* angemessen vergrößert.

4.3 Konstruktion des Dreiecksnetzes

Die Konstruktion der initialen Dreiecksoberfläche wurde bereits ausführlich in der Analyse (Kap. 3.3) betrachtet und besteht aus den Teilschritten der Konturendetektion und der Triangulierung. Daher wird das Kapitel in diese beiden Schritte unterteilt sein. Dabei wird die Erklärung der Umsetzung mit der Darstellung der Konturendetektion beginnen und mit der Betrachtung der Triangulierung abschließen. Allerdings soll bei der Erklärung, aus Platzgründen, von einer Beschreibung der Verfahren zum Verschließen des Netzes und dem Abtasten des Objektes abgesehen werden.

[1] Die Skalierung des Datensatzes erfolgt bislang außerhalb der Applikation. Es werden stets der verlustfreie und verlustbehaftete Datensatz eingeladen. Innerhalb der Applikation wird der verlustfreie Datensatz und das vergrößerte Netz dargestellt, damit eine visuelle Überprüfung durch den Benutzer erfolgen kann.

Doch bevor mit der Erklärung begonnen wird, soll zuvor noch erwähnt werden, dass alle Teilschritte der Konstruktion des Dreiecksnetzes in der Klasse `CTriangleMesh` erfolgen. Das dabei gewonnene Dreiecksnetz wird in einem hier instantiierten Objekt der Klasse `CWingedEdge` gespeichert und verwaltet.

4.3.1 Konturendetektion

In diesem Abschnitt geht es um die Beschreibung der Umsetzung der generellen Konturendetektion, die nach dem Prinzip der aktiven Konturen erfolgt. Daher ist es bekanntlich notwendig, dass zu Beginn einer Ermittlung zunächst ein initialer Polygonzug bestimmt wird, der dann sukzessiv an das Objekt angepasst wird, wodurch sich nach und nach die Kontur ergibt.

Initialer Polygonzug

Die initialen Polygonzüge werden bekanntlich Achsen ausgerichtete, umhüllende Rechtecke sein. Da diese Rechtecke zuvor bereits ermittelt wurden, werden sie, in Abhängigkeit von der Verarbeitungsrichtung, aus der entsprechenden Liste in `m_pVolData` ausgelesen.

```
// pVolumeData ist ein Zeiger auf m_pVolData (in CVolumeDataManager)
switch(pVolumeData->m_nProcessDir)
{ // Verarbeitungsrichtung:
  case PROCESS_DIR_X:
    layerMinAABB = pVolumeData->listLayerInfo_X.at(nCurrLayer).minAABB;
    layerMaxAABB = pVolumeData->listLayerInfo_X.at(nCurrLayer).maxAABB;
    break;
  case PROCESS_DIR_Y: // analog zu PROCESS_DIR_X
    break;
  case PROCESS_DIR_Z: // analog zu PROCESS_DIR_X
    break;
}
```

Listing 4.9: Auslesen des umhüllenden Rechtecks einer Schicht.

Die ausgelesenen Min/Max-Koordinaten werden dann in `detectPolyRectangle(...)` dazu genutzt einen Polygonzug, in Form eines Rechtecks, zu definieren. Wie in Kapitel 3.3.2 beschrieben, werden die vier Kanten eines Rechtecks, gemäß der gewünschten Kantenlänge, in mehrere kurze Abschnitte unterteilt, um ein besseres Resultat bei der Detektion zu ermöglichen. Hierbei werden die Formeln 3.1 bis 3.4 umgesetzt. Da diese Formeln bereits zuvor ausreichend thematisiert wurden, soll an dieser Stelle von einer detaillierten Beschreibung anhand von Quellcode abgesehen werden.

Aktive Kontur

Die Anpassung des geschlossenen Polygons ist ein iterativer Prozess der durch Multipass-Rendering ausgeführt wird. Hierbei erfolgt die Umsetzung im Fragment Shader. In jedem Iterationsschritt wird

für jeden Punkt die neue Folgeposition aus der Kombination der inneren und äußeren Kraft berechnet. Die für die Berechnung der inneren Kraft (siehe Formel 3.10) benötigte 1-Nachbarschaft eines Punktes wird, wie in Kapitel 4.1.2 beschrieben, mittels der Texturen *NeighborInfo* und *NeighborIndex* realisiert.

```
float3 vertex = tex2D(Vertices, texCoord).rgb; // Aktueller Punkt
float3 vtxTexCoord = float3(vertex * fVoxelSize); // Lookup Koordinaten
float3 intForce = float3(0.0, 0.0, 0.0); // Innere Kraft
// ... for(Lookup Nachbarn)
  // ... BEGIN Verarbeitung des aktuellen Nachbarn
  float4 currNB = tex2D(Vertices, currNeighborTexCoord);
  intForce += 0.5 * (currNB.xyz - vertex.xyz); // Berechne innere Kraft
  // END Verarbeitung
// END for(Lookup Nachbarn) ...
```

Listing 4.10: Ermittlung der inneren Kraft bei konstanter Valenz (=2).

Als äußere Kraft dient der invertierte, ebene Gradient, der mittels der zentralen Differenzen (siehe Formel 3.9) im Distanzfeld (*SignedDT*) berechnet wird.

```
float3 grad = float3(0.0, 0.0, 0.0);
// ... Zentrale Differenzen
// -> Verarbeitungsrichtung: Z (X und Y analog)
// X-Komponente
float4 x1 = tex3D(SignedDT, vtxTexCoord + float3( 1, 0, 0)*fVoxelSize);
float4 x2 = tex3D(SignedDT, vtxTexCoord + float3(-1, 0, 0)*fVoxelSize);
grad.x = x1.r - x2.r;
// Y-Komponente
float4 y1 = tex3D(SignedDT, vtxTexCoord + float3( 0, 1, 0)*fVoxelSize);
float4 y2 = tex3D(SignedDT, vtxTexCoord + float3( 0,-1, 0)*fVoxelSize);
grad.y = y1.r - y2.r;
// Texture Lookup der Distanz des aktuellen Punktes
float fDist = tex3D(SignedDT, vtxTexCoord).r;
// Berechnung der äußeren Kraft
float3 extForce = -grad * fDist;
```

Listing 4.11: Die Umsetzung der äußeren Kraft.

Die Folgeposition wird nun aus der Kombination beider Kräfte bestimmt (vgl. Formel 3.11). Wobei die Parameter `Stepsize` und `Alpha` per Parameter an den Shader übergeben werden und `Stepsize` entsprechend die Schrittweite τ darstellt.

```
// ... Berechnung der Folgeposition (Kombination der Kräfte)
result = vertex + Stepsize * lerp(extForce, intForce, Alpha);
```

Listing 4.12: Die Kombination der beiden Kräfte.

Die Modifikation der Punktkoordinaten führt zu Änderungen in der Geometrie (*Vertices*), aber nicht in der Topologie des Polygonzugs (*NeighborInfo/NeighborIndex*). Aus diesem Grund ist es ausreichend, wenn die Texturen der Nachbarschaftsbeziehungen einmal zu Beginn in dem lokalen Speicher der Grafikkarte abgelegt werden und die Geometrie mittels des Ping-Pong-Prinzips verarbeitet wird. Das heißt, im ersten Rendering Pass wird die Berechnung auf Basis der *Vertices*-Textur erfolgen und das Ergebnis in einen der FBOs gerendert. Die weiteren Durchläufe erfolgen nun, nach dem bekannten Prinzip, indem die beiden FBOs ständig ihre Rollen tauschen. Die Größe der FBOs ist hierbei gleich der Texturgröße von *Vertices*.

Die Detektion einer Kontur ist abgeschlossen, wenn die Punktkoordinaten nicht weiter modifiziert werden. Wurde eine Kontur gefunden, so werden ihre Punkte und Kanten in der Instanz zur Repräsentation des Dreiecksnetzes (`m_pWingedEdge`) gespeichert. Zudem werden die Indizes zu den Punkten in `m_pWingedEdge` temporär in einer Liste abgespeichert, um die Punkte der Kontur in der Gesamtmenge aller Punkte leichter ausfindig machen zu können. Die Indexliste wird gelöscht, sobald diese Kontur zu beiden Seiten, also mit der vorherigen und der nachfolgenden Kontur, verbunden wurde.

4.3.2 Triangulierung

In dieser Arbeit wird die Triangulierung der benachbarten Konturen dadurch erreicht, dass zunächst jede mögliche Verbindung durch die Ermittlung der Korrespondenz der Punkte (DOC) bewertet wird. Im Folgenden wird daher erläutert wie der dazu nötige Graph erstellt und im Anschluss daran genutzt werden kann, um eine erfolgreiche Triangulierung zu erreichen. Die Umsetzung erfolgt hierbei komplett auf dem Hauptprozessor (engl. „Central Processing Unit“ - CPU).

Erstellen des Korrespondenz-Graphen

Gestartet wird das Verbinden zweier Konturen durch den Aufruf von `connectPolylines(...)`. Hierbei werden die Indexlisten der beiden zu verbindenden Konturen als Parameter übergeben. Die in einer Liste gespeicherten Indizes repäsentieren, wie zuvor beschrieben, die Punkte der entsprechenden Kontur. Durch die Multiplikation der beiden Anzahlen von Punkten ergibt sich die Anzahl der möglichen Verbindungskanten. Diese Verbindungen werden bewertet und in dem Graph gespeichert, der durch ein eindimensionales Array (`double* pdDOC`) realisiert wird. Der hierfür zu allokierende Speicher richtet sich nach der Anzahl der möglichen Verbindungen.

Nach der Allokation beginnt die eigentliche Bewertung mittels der DOC-Kriterien. Anfangs erfolgt jedoch ein Vorverarbeitungsschritt, in dem die Bestimmung der Gesamtlängen beider Konturen und die Ermittlung der minimalen und maximalen Kantenlängen der möglichen Verbindungen erfolgt. Zugleich wird, auf Basis des Kriteriums der kürzesten Verbindung, zu jedem Punkt ein korrespondierender Punkt und das initiale Punktepaar gesucht. Wie in der Analyse beschrieben erfolgt hierbei jedoch die Einschränkung auf korrespondierende Teilbereiche. Die aus dieser Vorverarbeitung gewonnenen Werte dienen zur Berechnung der DOC-Kriterien und werden im weiteren Verlauf näher betrachtet.

```
double dMinLengthStartEdge = MAX_VALUE;
for(int p=0; p<nNumVtxPL_1; p++) // Punkte der ersten Kontur (PL_1)
{
  Vertex grVtxP = m_pWingedEdge->m_listVertices.at(pPolyline1->at(p));
  int nNextIdxP = pPolyline1->at( (p+1)%nNumVtxPL_1 );
  dShortest = MAX_VALUE; // Für die Suche nach korrespondierenden Punkten
  // Berechne die Gesamtlänge der ersten Kontur
  tmpVector = m_pWingedEdge->m_listVertices.at(nNextIdxP).vec-grVtxP.vec;
  dTotalRangePL_1 += tmpVector.getLength();
  // Vergleiche diesen Punkt mit allen Punkten der anderen Kontur
  for(int q=0; q<nNumVtxPL_2; q++)  // Punkte der zweiten Kontur (PL_2)
  {
    int nCurrIdxQ = pPolyline2->at(q);
    Vertex grVtxQ = m_pWingedEdge->m_listVertices.at(nCurrIdxQ);
    // Berechne die Min/Max-Längen
    tmpVector = grVtxQ.vec - grVtxP.vec;
    dMinLength = min(tmpVector.getLength(), dMinLength)
    dMaxLength = max(tmpVector.getLength(), dMaxLength)
    // Bestimme die korrespondierenden Punkte (für Bijektivitäts-Test)
    // -> wenn beide Punkte im korres. Teilbereich liegen
    if(tmpVector.getLength() < dShortest && PQareInSameSubarea)
    {
      dShortest = tmpVector.getLength(); // Speichere kürzeste Länge und
      anShortestIdxPL1_to_PL2[p] = nCurrIdxQ; // Korrespondierenden Index
      // Bestimmung des initialen Punktepaars
      if(dShortest < dMinLengthStartEdge)
      {
        dMinLengthStartEdge = dShortest;
        pnStartIndices[0] = p; // Initialer Index auf PL_1
        pnStartIndices[1] = q; // Initialer Index auf PL_2
      }
    }
  } // END for(Kontur2)
} // END for(Kontur1)
// Die Bestimmung der Werte (ohne Suche der Startkante) erfolgt analog
// in umgekehrter Richtung für die zweite Kontur (PL_2 to PL_1)...
```

Listing 4.13: Vorverarbeitungsschritt zur Bestimmung des Korrespondenz-Graphen.

Jede mögliche Verbindung wird nun bezüglich ihrer Korrespondenz separat bewertet und das Resultat im Graphen gespeichert. Die Umsetzung der fünf Bewertungskriterien werden im Folgenden erläutert. Hierbei sei zur Erinnerung auf die Erklärung in Kapitel 3.3.3 und besonders auf die darin enthaltenen Formeln 3.12 bis 3.17 verwiesen.

Distanz Die Berechnung der euklidischen Distanz erfolgt durch die Bildung des Differenzvektors zwischen den beiden aktuellen Punkten der Konturen (P, Q).

```
CVector edgeQP = currQ-currP;
double dNormFactor = dMaxLength-dMinLength; // Für die Normierung
// Berechnung der normierten Distanz
dDist = 1.0 - ( (edgeQP.getLength()-dMinLength) / dNormFactor );
```

Listing 4.14: Berechnung der euklidischen Distanz zwischen zwei Punkten.

Dadurch, dass `dMinLength` von der aktuellen Länge subtrahiert und das daraus folgende Ergebnis durch `dNormFactor` dividiert wird, wird das Ergebnis in das Intervall $[0, 1]$ abgebildet. Dabei ist das Ergebnis null, wenn die aktuelle Länge gleich `dMinLength` ist. Da das Ergebnis bei bester Korrespondenz allerdings maximal sein soll, wird das Resultat noch von der Obergrenze des Intervalls subtrahiert.

Orientierungsdifferenz Die Orientierungsdifferenz beschreibt den Winkel zwischen den Tangenten der aktuellen Punkte. Nach der Berechnung der Tangenten anhand zentraler Differenzen, erfolgt die anschließende Ermittlung der gewünschten Orientierungsdifferenz mittels des Skalarprodukts.

```
// Bestimmung der Tangenten für P,Q mittels zentraler Differenzen
CVector tangentP = nextP-prevP; // Tangente für P
tangentP.normalize();
CVector tangentQ = nextQ-prevQ; // Tangente für Q
tangentQ.normalize();
// Berechnung der normierten Orientierungsdifferenz
dOrientDiff = ( tangentP.getInnerProduct(tangentQ) + 1 ) * 0.5;
```

Listing 4.15: Berechnung der Orientierungsdifferenz.

Auch dieses Ergebnis wird in das gewünschte Intervall umgerechnet.

Vektorwinkeldifferenz Die im vorhergehenden Schritt ermittelten Tangenten werden auch zur Berechnung der Vektorwinkeldifferenz benötigt. Zur Erinnerung: Im Idealfall stehen die Tangenten normal zu der zu bewertenden Kante. Da es durchaus möglich ist, dass ein Winkel optimal und der andere zugleich weniger gut ist, werden die Resultate der beiden Skalarprodukte gemittelt.

```
edgeQP.normalize();
dVectDiff = 1-(( fabs( tangentP.getInnerProduct(edgeQP) )
              + fabs( tangentQ.getInnerProduct(edgeQP) ) )*0.5);
```

Listing 4.16: Berechnung der gemittelten Winkeldifferenz.

Durch die Normierung folgt wieder, dass im bestmöglichen Fall das Ergebnis gleich eins ist.

Fortschrittsdifferenz Die auf Basis des initialen Punktepaars gewonnenen Streckenfortschritte, auf beiden Konturen, werden durch die Division der jeweiligen Gesamtlänge normiert und somit vergleichbar. Die Differenz stellt somit den normierten Unterschied beider Fortschritte dar.

```
double dProgressPL_1 = dCurrRangePL_1/dTotalRangePL_1;
double dProgressPL_2 = dCurrRangePL_2/dTotalRangePL_2;
// Differenz der Fortschritte
dProgressDiff = 1 - fabs(dProgressPL_1-dProgressPL_2);
```

Listing 4.17: Bestimmung der normierten Fortschrittsdifferenz.

Bijektivität Im Vorfeld wurde zu jedem Punkt, auf Basis des Kriteriums der minimalen Kantenlänge, ein korrespondierender Punkt gesucht. Bei dem Test auf Bijektivität wird nun überprüft, ob das aktuelle Punktepaar sich gegenseitig als korrespondierend gefunden hat.

```
double dBijective = 0.0;
if( nCurrP == nShortestIdxPL2_to_PL1  // P korrespondierend zu Q?
 && nCurrQ == nShortestIdxPL1_to_PL2) // Q korrespondierend zu P?
{
  dBijective = 1.0;
}
```

Listing 4.18: Test auf Bijektivität.

Verbinden der Konturen

Nach der Erzeugung des DOC-Graphen, beginnt das Verbinden der Konturen mit dem Einfügen der ersten Kante in `m_pWingedEdge`.

```
// Verbinde beide Polygonzüge mittels des initialen Punktepaars
m_pWingedEdge->addEdge(Edge(pPolyline1->at(pnStartIndices[0]),
                            pPolyline2->at(pnStartIndices[1])));
// Update der Winged-Edge-Konnektivität
```

Listing 4.19: Hinzufügen der Startkante zur WingedEdge-Datenstruktur.

Im Anschluss daran wird die Mantelfläche Schritt für Schritt geschlossen. Dies geschieht indem der Graph, ausgehend vom ersten Knoten (initiales Punktepaar), nach dem Greedy-Prinzip iterativ traversiert wird. Hierbei wird mittels `ncDOC_PL_1` und `ncDOC_PL_2` über die jeweilige Kontur iteriert.

```
// Auslesen der relevanten DOC-Werte
dDOC_PL_1 = pdDOC[ncDOC_PL_1 + ((ncDOC_PL_2+1)%nSizePL_2)*nSizePL_1];
dDOC_PL_2 = pdDOC[(ncDOC_PL_1+1)%nSizePL_1 + (ncDOC_PL_2*nSizePL_1)];
// Vergleiche beide Werte um nächste hinzuzufügende Kante zu bestimmen
```

Listing 4.20: Auslesen der DOC-Werte aus dem Array.

Entsprechend des größeren Wertes, wird dann entweder auf der ersten oder zweiten Kontur weiter fortgeschritten, bis eine der Konturen vollständig verarbeitet wurde. Dieser Vergleich entfällt sobald nur noch eine Kontur „offen“ ist, da die geschlossene Kontur nicht mehr weiterverarbeitet wird. Die auf diese Weise, Schritt für Schritt, ermittelten Kanten werden jeweils in der Repräsentation des Netzes eingefügt und die Konnektivität wird entsprechend modifiziert. Nach der Verarbeitung beider Konturen sind diese zu einer Mantelfläche verbunden worden.

Nachdem auf diese Weise alle Konturen, auch die skalierten Konturen zum Verschließen des Netzes, verbunden wurden, ist das grobe initiale Dreiecksnetz vollständig konstruiert und wird entsprechend durch `m_pWingedEdge` repräsentiert.

4.4 Anpassung und Optimierung der Dreiecksoberfläche

Nachdem in dem vorhergehenden Kapitel die Umsetzung der Konstruktion des groben Dreiecksnetzes erläutert wurde, wird in diesem Kapitel die weitere Anpassung erklärt. Wie bereits zuvor angemerkt, wird hierbei die Anpassung, der Optimierungsschritt und zuletzt auch der Abschluss der Anpassung beschrieben. Doch zuvor wird mit einem kurzen Überblick über den Vorgang begonnen, um ein grundlegendes Verständnis für die Beschreibung der Umsetzung und deren Zusammenhänge zu vermitteln.

4.4.1 Überblick

Die Konstruktion des Dreiecksnetzes erfolgt bekanntlich in der Klasse `CTrinagleMesh`. Bisher wurde jedoch nicht erwähnt, dass die zugehörige Instanz in der Klasse `CSubdivisionMesh` instantiiert wird. Diese Klasse übernimmt die Verwaltung des gesamten Vorgangs, der vor allem aus den in Abbildung 4.3 dargestellten Schritten besteht.

Bei Betrachten des Ablaufs ist zu sehen, dass die Anpassung des Netzes in `CTriangleMesh` und die Optimierung in `CSubdivisionMesh` erfolgt. Der Name der verwaltenden Klasse leitetet sich entsprechend aus dem Hauptaugenmerk des Optimierungsschritts ab, dass sich bekanntlich auf die adaptive Unterteilung der Dreiecksflächen richtet, um weitere Oberflächendetails hinzuzufügen.

4.4.2 Anpassungsvorgang

Wie unter anderem die Analyse zeigt, ähnelt das Grundprinzip des Anpassungsvorgangs dem der aktiven Konturen. Denn auch hier wird die Folgeposition eines Punktes aus der Kombination der inneren und äußeren Kraft berechnet. Aufgrund der großen Analogie, sind die Rahmenbedingungen der Verfahren zur Konturendetektion und zur Anpassung eines Dreiecksnetzes vergleichbar. So ist die Anpassung eines Netzes ebenfalls ein iterativer Prozess, der durch ein Fragment Programm auf der GPU ausgeführt wird. Auch hierbei beschränkt sich die Modifikation auf die alleinige Änderung der Geometrie, also der Punktkoordinaten. Daher werden die Punkte des Netzes zunächst einmal anhand der *Vertices*-Textur in den lokalen Grafikspeicher geladen und im Anschluss nach dem Ping-Pong-Prinzip verarbeitet (vgl. Kap. 4.3.1).

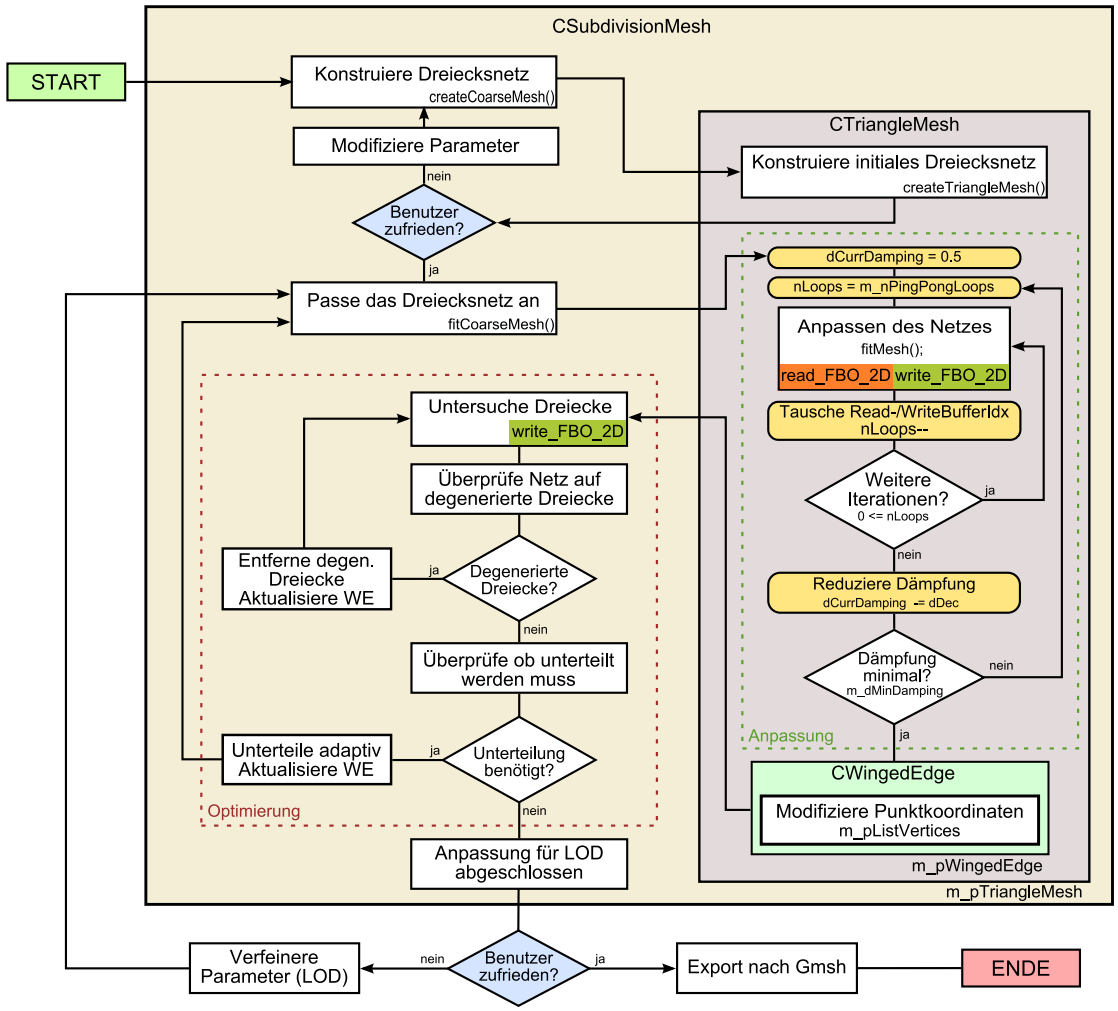


Abbildung 4.3: Ablauf der Anpassung und Optimierung eines Dreiecksnetzes.

Die Berechnungen der einzelnen Kräfte unterscheiden sich allerdings, aufgrund der komplexeren Dimensionalität der Problemstellung, etwas von denen der Konturendetektion und sollen im Folgenden kurz erläutert werden.

Innere Kraft

Die innere Kraft wird mittels der approximierenden Laplacian-Glättung berechnet (siehe Formel 2.7). Dazu wird die benötigte 1-Nachbarschaft eines Punktes auch hier mittels der Texturen *NeighborInfo* und *NeighborIndex* realisiert (vgl. Kap. 4.1.2). Zur Erinnerung: In `currInfoNB.r` ist die Valenz des aktuellen Punktes (`vertex`) gespeichert.

Nach der Bestimmung des Laplacian-Vektors wird die Folgeposition der Dämpfung berechnet (vgl. Formel 3.20) und in `relaxingForce` gespeichert.

```
// ... Lookup Nachbarn ...
currNB = tex2D(Vertices, currNeighborTexCoord);
// Bestimmung des Laplacian-Vektors
laplacian += (1.0/currInfoNB.r) * (currNB.xyz - vertex.xyz);
// END for(Lookup Nachbarn) ...
// ... Texture Lookup der Distanz des aktuellen Punktes
float3 vtxTexCoord = float3(vertex.rgb * fVoxelSize);
float fDist = tex3D(SignedDT, vtxTexCoord).r;
// Berechnung der inneren Kraft (Relaxing-Force)
relaxingForce = vertex.xyz + abs(sign(fDist)) * laplacian;
```

Listing 4.21: Ermittlung der inneren Kraft bei variierender Valenz.

Äußere Kraft

Die äußere Kraft ist, aus Gründen der Stabilität, eine Kombination aus dem invertierten dreidimensionalen Gradienten und der inneren Kraft (vgl. Kap. 3.4.2).

```
// Berechnung des dreidimensionalen Gradienten (Zentrale Differenzen)
float3 grad = -getGradient3D(vtxTexCoord, fVoxelSize, SignedDT);
// Berechnung der Anzugskraft
attractingForce = vertex.xyz + Stepsize * abs(fDist)
                * lerp(grad*sign(fDist), relaxingForce, Alpha);
```

Listing 4.22: Berechnung der Anzugskraft.

Wobei `Stepsize` die gewünschte Schrittweite τ der Formel 3.21 ist. Die Schrittweite und auch `Alpha` werden per Parameter an den Shader übergeben und beeinflussen maßgeblich die Folgeposition der `attractingForce`.

Kombination der Kräfte

Ein Anpassungsdurchgang nimmt generell eine durch den Benutzer spezifizierte Anzahl von Iterationen (`m_nPingPongLoops`) vor. Entsprechend des, per Parameter übergebenen, Dämpfungsfaktors `Damping` wird in jeder Iteration zwischen den beiden zuvor ermittelten Folgepositionen linear interpoliert (vgl. Formel 3.22).

```
// ... Berechnung der Folgeposition (Kombination der Kräfte)
result = lerp(attractingForce, relaxingForce, Damping);
```

Listing 4.23: Kombination der inneren und äußeren Kraft.

Wie in Kapitel 3.4.2 beschrieben, beginnt ein Anpassungsvorgang anfangs mit `Damping` = 0.5 und wird nach jedem abgeschlossenen Durchgang verringert. Auf diese Weise wird von Durchgang zu Durchgang die Dämpfung immer weiter reduziert, bis der Dämpfungsfaktor den minimalen Dämpfungswert (`m_dMinDamping`) erreicht hat. Wenn der Minimalwert erreicht wurde, so wird noch ein letztes Mal ein Anpassungsdurchgang ausgeführt.

Nach Abschluss der Anpassung werden die in `m_pWingedEdge->m_listVertices` gespeicherten Punktkoordinaten durch die ermittelten Folgepositionen ersetzt. Die auf diese Weise modifizierte Geometrie steht somit, als aktuelles Dreiecksnetz, für die nun folgende Optimierung bereit.

4.4.3 Optimierung eines Dreiecksnetzes

In diesem Kapitel soll die Umsetzung der Optimierung verdeutlicht werden. Doch bevor knapp auf die allgemeine Optimierung und auf die adaptive Unterteilung eingegangen wird, soll zu Beginn die vorgelagerte Untersuchung der einzelnen Dreiecke dargestellt werden. Hierbei wird der Schwerpunkt auf das zufällige Mehrfachsampling gerichtet.

Untersuchung der Dreiecke

Die Untersuchung der Dreiecke erfolgt ebenfalls durch den Fragment Shader. Hierbei werden zu jedem Dreieck die benötigten Werte ermittelt, um beurteilen zu können ob ein Dreieck...

- ... *degeneriert* ist (→ Verhältnis zw. In- und Umkreis, kürzeste Kante und kleinster Winkel)
- oder eine *Unterteilung* benötigt (→ Durchschnittliche Distanz und Flächeninhalt).

Durch die Verwendung der Texturen *TriangleTex1* und *TriangleTex2* sind für jedes Dreieck die Texturkoordinaten zu den drei Eckpunkten bekannt. Somit können die für die Berechnungen benötigten Eckpunkte eines Dreiecks mittels drei Texture Lookups in *Vertices* ausgelesen werden (vgl. Kap. 4.1.2).

```
// ... Lookup der Texturkoordinaten zu den Punkten
float4 textCoordV0V1s  = tex2D(TriangleV0V1, texCoord);
float4 textCoordV1tV2  = tex2D(TriangleV2, texCoord);
// Lookup der Punkte in Vertices
float4 vertex0 = tex2D(Vertices, textCoordV0V1s.rg);
float2 mergedTexCoordV1 = float2(textCoordV0V1s.b, textCoordV1tV2.r);
float4 vertex1 = tex2D(Vertices, mergedTexCoordV1);
float4 vertex2 = tex2D(Vertices, textCoordV1tV2.gb);
// ...
```

Listing 4.24: Lookup der Eckpunkte eines Dreiecks.

Aus den so erhaltenen Eckpunkten wird nun das Verhältnis zwischen In- und Umkreis, der Flächeninhalt, die Länge der kürzesten Kante und der kleinste Winkel berechnet. Da die detaillierte Beschreibung der Berechnungen für das Verständnis der Arbeit nicht notwendig ist, soll hier lediglich auf die im Anhang A.1 aufgeführten Formeln verwiesen werden. Demzufolge wird auch auf die weitere Beschreibung anhand von Quellcode verzichtet.

Mehrfachsampling zur Bestimmung der durchschnittlichen Distanz

Die Ermittlung der durchschnittlichen Distanz eines Dreiecks erfolgt, wie in Kapitel 3.5.2 kurz beschrieben, mittels eines zufälligen Mehrfachsamplings der Dreiecksflächen. Die dafür benötigten Zufallswerte werden der GPU mittels Texturen bereitgestellt.

Generell verläuft die Umsetzung des zufälligen Mehrfachsamplings analog zu dem Verfahren des Lookups der 1-Nachbarschaft eines Punktes. Auch hier werden drei Texturen benötigt: *RandSamplingInfo*, *RandTexCoords* und *RandBarycentricWeights* (vgl. Tab. 4.1). Im Folgenden sollen diese Texturen in einem kurzen Vergleich zu ihrem Pendant näher beschrieben werden.

RandBarycentricWeights Diese zweidimensionale RGB-Textur enthält zufällig berechnete baryzentrische Gewichte, mit denen die Lookup-Koordinaten auf den Dreiecksflächen bestimmt werden (vgl. A.2). Die Berechnung der zufälligen Gewichte wird wie folgt durchgeführt:

```
// Berechnung der zufälligen baryzentrischen Gewichte
double dBW_V0 = static_cast<double>(rand())/RAND_MAX;
double dBW_V1 = (1.0-dWeightV0)*static_cast<double>(rand())/RAND_MAX;
double dBW_V2 = 1.0-(dWeightV0+dWeightV1);
```

Listing 4.25: Berechnung der zufälligen baryzentrischen Gewichte.

Diese Textur ist vom Prinzip her mit der *Vertices*-Textur zu vergleichen. Jedoch mit dem Unterschied, dass in *RandBarycentricWeights* keine Punktkoordinaten, sondern baryzentrische Gewichte gespeichert werden. Zudem ist sie ebenfalls quadratisch, wobei ihre Größe sich nach der vom Benutzer spezifizierten Anzahl der Gewichte richtet.

RandTexCoords In *RandTexCoords* werden zufällige Texturkoordinaten gespeichert, um auf diese Weise zufällige Texture Lookups in *RandBarycentricWeights* durchführen zu können. Somit ist diese Textur das Pendant zu der *NeighborIndex*-Textur und wird aufgrund der Vergleichbarkeit analog generiert.
Sie unterscheiden sich hierbei lediglich in der Berechnung der Texturkoordinaten und der Bestimmung der benötigten Anzahl der Texel, ergo der Texturgröße. Die zufälligen Texturkoordinaten s_n, t_n zu einem baryzentrischen Gewicht werden hierbei wie folgt bestimmt:

```
double s = static_cast<double>( rand() ) / RAND_MAX;
double t = static_cast<double>( rand() ) / RAND_MAX;
```

Listing 4.26: Erzeugung der zufälligen Lookup-Koordinaten.

Die Anzahl der Texel (*NumTexel*) errechnet sich aus der Formel 4.2, allerdings nicht in Abhängigkeit der Valenz, sondern der gewünschten Anzahl der Samples pro Dreiecksfläche (*NumSamples*). Jedes Dreieck wird hierbei gleich häufig gesampelt, weshalb die Größe der Textur aus der Multiplikation der Anzahl der Dreiecke und der *NumTexel* berechnet wird.

RandSamplingInfo Analog zur *NeighborInfo*-Textur werden hier ebenfalls die für die mehrfachen Texture Lookups notwendigen Informationen gespeichert. Die RGBA-Komponenten werden identisch verwendet, so dass *NumSamples* in der R-Komponente und die Texturgröße von *RandTexCoords* in der G-Komponente gespeichert wird. Die Texturkoordinaten für den ersten Lookup eines jeden Dreiecks in *RandTexCoords* werden in den freien BA-Komponenten abgelegt.

Mittels dieser drei Texturen wird das zufällige Mehrfachsampling genutzt, um die durchschnittliche Distanz wie folgt zu bestimmen:

```
// Lookup der aktuellen baryzentrischen Gewichte (Mehrfachsampling)
float4 currBaryWeights = tex2D(RandBarycentricWeights, currRandTexCoord);
// Bestimmung der aktuellen Schwerpunktkoordinaten
float3 currBarycenter = vertex0*currBaryWeights.x
                      + vertex1*currBaryWeights.y
                      + vertex2*currBaryWeights.z;
// Texture Lookup der Distanz des aktuellen Punktes
float3 currLookUpCoord = currBarycenter.xyz * fVoxelSize;
float fCurrDist = tex3D(SignedDT, currLookUpCoord).x;
// Berechnung der durchschnittlichen Distanz des Dreiecks
fAvgDist += (1.0/infoSampling.r) * fCurrDist;
```

Listing 4.27: Bestimmung der durchschnittlichen Distanz einer Dreiecksfläche.

Wie eingangs beschrieben, verläuft dieses Mehrfachsampling analog zum Verfahren des Lookups der 1-Nachbarschaft. Daher soll das Prinzip des Lookups an dieser Stelle nicht weiter thematisiert werden.

Optimierungsschritte

Nachdem die Untersuchung der Dreiecke abgeschlossen wurde, werden die gewonnenen Ergebnisse für jedes Dreieck in `m_pWingedEdge` gespeichert und zur Weiterverarbeitung auf der CPU genutzt. Aus Platzgründen wird hierzu jedoch nur ein knapper Ablauf skizziert.

Entfernen der degenerierten Dreiecke Zunächst wird das Netz auf degenerierte Dreiecke überprüft und bei positivem Resultat entsprechend mittels Edge Collapse oder Edge Swap bereinigt. Die Überprüfung erfolgt, indem über die Elemente von `m_listFaces` iteriert wird und die betreffenden Werte mit definierten Toleranzwerten verglichen werden. Wird hierbei ein degeneriertes Dreieck gefunden, wird die betroffene Kante entweder zum Kollabieren oder Flippen markiert und der Index der Kante entsprechend zu `m_listEdgeCollapse` oder `m_listEdgeSwap` in `m_pWingedEdge` hinzugefügt. Bei zu kollabierenden Kanten muss zudem noch beachtet werden, dass die angrenzenden Punkte über eine ausreichende Valenz verfügen, um Überfaltungen zu vermeiden.

Wurden alle Dreiecke überprüft, erfolgt die eigentliche Optimierung mittels Edge Collpase und Edge Swap, wobei zunächst `m_listEdgeCollapse` und dann `m_listEdgeSwap` abgearbeitet

wird. Beim Durchführen der einzelnen Transformationsoperatoren wird folglich die Topologie modifiziert. Dementsprechend wird zugleich auch die Winged-Edge Datenstruktur modifiziert. Für den Fall, dass mehrere benachbarte Kanten transformiert werden sollen, wird zunächst nur eine Kante optimiert und die anderen aus der betreffenden Liste entfernt. Denn oftmals wird durch die einzelne Optimierung bereits erreicht, dass die Degenerierung der benachbarten Dreiecke ebenfalls aufgelöst wird. Dies hat den Vorteil, dass die Auswirkungen auf das Netz so minimal wie möglich bleiben.

Im Anschluss an die Optimierung werden die Dreiecke des Netzes nochmals untersucht, um sicherzustellen, dass die Optimierung erfolgreich war und keine neuen degenerierten Dreiecke entstanden sind (vgl. Kap. 3.5.2). Dies ist gerade dann sinnvoll, wenn Edge Swaps durchgeführt wurden oder mehrere benachbarte Kanten für Operationen vorgesehen waren. Bei einem Fund wird die Optimierung entsprechend wiederholt.

Adaptive Unterteilung Zu Beginn wird ebenfalls über `m_listFaces` iteriert und dabei werden Dreiecke zur Unterteilung markiert, die eine weitere Anpassung benötigen. Wie in der Konzeption thematisiert, sind dies Dreiecke, die einen zu großen durchschnittlichen Abstand haben und deren Flächeninhalt nicht zu klein ist.

In einem zweiten Iterationsdurchgang wird dann über `m_listEdges` iteriert. Hierbei wird eine Kante mittels des Edge Split Operators geteilt, wenn mindestens eines der beiden adjazenten Dreiecke zu unterteilen ist.

Im Anschluss an die Unterteilung aller Kanten wird die Winged-Edge Datenstruktur aktualisiert, indem vor allem neue Kanten und Flächen hinzugefügt und verknüpft werden. Wie aus der Analyse bekannt ist, müssen hierbei drei Fälle unterschieden werden (vgl. Kap. 3.5.1).

Nach der Modifikation der Repräsentation des Netzes ist die Optimierung abgeschlossen und das Dreiecksnetz kann weiter angepasst werden.

4.4.4 Abschluss der Anpassung und Export nach *Gmsh*

Es wurde gezeigt, dass ein Anpassungsdurchgang eine definierte Anzahl von Iterationen vornimmt und dass die Dämpfung nach jedem Anpassungsdurchgang reduziert wird. Der Benutzer hat die Möglichkeit die Zahl der Iterationen (`m_nPingPongLoops`) und das Dekrement der Dämpfung mittels der grafischen Benutzeroberfläche zu definieren. Durch das Modifizieren dieser beiden Parameter kann das Resultat der Anpassung und zugleich auch die Performance maßgeblich beeinflusst werden.

Generell gilt eine Oberfläche als angepasst, wenn Anpassungsvorgänge keine weiteren Verbesserungen mehr erzielen und keine zu unterteilenden Dreiecke mehr gefunden werden. Wobei die Zahl der zu unterteilenden Dreiecke als Abbruchbedingung genutzt wird. Das heißt die Anpassung gilt als abgeschlossen, wenn keine Unterteilungen mehr benötigt werden. Hierbei ist allerdings nicht völlig auszuschließen, dass in einem Netz Regionen vorhanden seien können, bei der die Anpassung und die Optimierung einen Zyklus bilden. Ein Zyklus entsteht, wenn Dreiecke durch die Anpassung degenerieren, im Zuge dessen optimiert werden und infolge der Optimierung wieder zu unterteilen sind.

In solchen Fällen würde die Anpassung in eine Endlosschleife führen. Um dieses zu verhindern wird zudem die Zahl der Gesamtdurchgänge überwacht und der Vorgang gegebenenfalls abgebrochen.

Im Allgemeinen schließt die adaptive Unterteilung allerdings aufgrund des Erreichens der Detailstufe ab. Sollte der Benutzer dann einen höheren Detailgrad wünschen, so kann er den Toleranzwert des minimalen Flächeninhalts passend modifizieren und die Anpassung wiederholen.

Ist der Benutzer mit dem Resultat zufrieden, dann wird das Dreiecksnetz mittels der *Gmsh* eigenen Skriptsprache in einer ASCII-kodierten Textdatei gespeichert. Dieser Export wird in `CWingedEdge` durch die Funktion `outputGmshGeo(...)` umgesetzt. Hierbei werden die einzelnen Elemente des Netzes wie in Kapitel 3.6 erläutert ausgegeben.

Aufgrund der hohen Speicheranforderung wird, bei großen Datensätzen, das Distanzfeld auf Basis einer verkleinerten Textur bestimmt. Dementsprechend wird auch ein, um den Skalierungsfaktor der Textur, verringertes Dreiecksnetz generiert (vgl. Kap. 4.2.3). Damit die Geometrie allerdings in der unskalierten Größe exportiert wird, werden die Punktkoordinaten beim Export mit dem entsprechenden Skalierungsfaktor multipliziert.

```
int nCurrVtx = 1;
for(itv=m_listVertices.begin(); itv!=m_listVertices.end(); itv++)
{
  // Ausgabe-String:        ID      x,  y,  z, Characteristic length
  strDom += QString("Point (%1) = {%2, %3, %4, lc};\n")
                                    .arg(nCurrVtx )
                                    .arg(itv->vec[0]*padScaleFact[0])
                                    .arg(itv->vec[1]*padScaleFact[1])
                                    .arg(itv->vec[2]*padScaleFact[2]);
  nCurrVtx++;
}
```

Listing 4.28: Skalierung der Geometrie beim Export nach *Gmsh*.

4.5 Zusammenfassung

Die Erzeugung eines Tetraedergitters beginnt mit der Vorverarbeitung des gegebenen Voxelgitters. Hierbei werden vor allem das benötigte Distanzfeld und die umhüllenden Geometrien gewonnen. Aufgrund der hohen Speicheranforderung des Multipass-Renderings, wird bei hochauflösenden Datensätzen die Verarbeitung auf verkleinerten Voxelgittern durchgeführt.

Die anschließende Konstruktion des Dreiecksnetzes beginnt mit der Konturendetektion, die auf der GPU ausgeführt wird. Für die Verarbeitung auf der GPU werden die geometrischen Informationen und die benötigten topologischen Beziehungen der Polygonzüge mittels Texturen in den lokalen Grafikspeicher geladen. Die eigentliche Detektion erfolgt nach dem Prinzip der aktiven Konturen und modifiziert lediglich die Geometrie. Daher ist es ausreichend, die benötigten topologischen Informationen einmal zu Beginn der Ermittlung in den Grafikspeicher abzulegen und die Geometrie im

Ping-Pong-Prinzip zu verarbeiten. Detektierte Konturen werden in einer Instanz der Winged-Edge Repräsentation gespeichert und mittels des Triangulierungsschritts zu einer Mantelfläche verbunden.

Die Triangulierung erfolgt, anders als die Detektion, auf der CPU und beginnt jeweils mit der Erstellung eines Graphen. Die Knoten eines Graphen repräsentieren hierbei die möglichen Verbindungskanten zwischen zwei benachbarten Konturen. All diese Punktepaare werden bezüglich ihrer Korrespondenz separat bewertet. Aufgrund der höheren Komplexität des beschriebenen Bewertungsverfahrens ist es möglich einen Graphen nach dem Greedy-Prinzip zu traversieren, um eine geschlossene Mantelfläche zu erzeugen. Eine zeitintensive Graphensuche wird somit vermieden.

Nach der Konstruktion des initialen Dreiecksnetzes folgt die sukzessive weitere Anpassung. Hierbei wird, vergleichbar zur Konturendetektion, ebenfalls nur die Geometrie mittels eines Multipass-Renderings modifiziert. Zudem werden, zur Gewinnung der inneren Kraft, die topologischen Informationen zur 1-Nachbarschaft eines Punktes in Texturen gespeichert. Die variierende Valenz der Punkte erfordert daher ein dynamisches Verfahren, um das mehrfache Sampling auf der GPU zu ermöglichen. Hierzu werden zwei Texturen genutzt. Während die eine Textur die Informationen zum Sampling bereithält, enthält die andere die nötigen Lookup-Koordinaten für die *Vertices*-Textur. Wurde der Anpassungsvorgang abgeschlossen, so folgt die anschließende Optimierung.

Die Optimierung beginnt jeweils mit der Bewertung der Dreiecke, die anhand eines Renderingschritts auf der GPU ausgeführt wird. Hierzu wird die Topologie der Dreiecke mittels zweier Texturen bereitgehalten. Während der Bewertung werden verschiedene Werte ermittelt, um beurteilen zu können, ob ein Dreieck degeneriert ist oder eine weitere Unterteilung benötigt. Alle Werte, bis auf den der durchschnittlichen Distanz eines Dreiecks, lassen sich durch die bekannten Berechnungsformeln eines Dreiecks berechnen. Die durchschnittliche Distanz wird jedoch mit Hilfe eines zufälligen Samplingverfahrens bestimmt. Wobei die dazu benötigten Zufallswerte auf der CPU generiert und mittels Texturen, analog zur Vorgehensweise bei den 1-Nachbarschaften, der GPU bereitgestellt werden.

Im Anschluss an die Bewertung erfolgt die Optimierung des Dreiecksnetzes auf der CPU. Falls vorhanden, werden hierbei zunächst die degenerierten Dreiecke durch anwenden der Transformationsoperatoren, Edge Collapse und Edge Swap, entfernt. Erst danach werden gegebenenfalls die Oberflächendetails gezielt mittels des Edge Split Operators erweitert.

Der Prozess der Anpassung ist ein iterativer Vorgang und erfolgt immerzu im Wechsel mit der Optimierung. Das Verfahren schließt ab, wenn keine zu unterteilenden Dreiecke mehr gefunden werden oder eine maximale Anzahl von Durchläufen erreicht wird. Wurde eine zufriedenstellende Approximation der Oberfläche erzeugt, so wird die Geometrie an *Gmsh* übergeben, um letztlich das benötigte Tetraedergitter zu generieren. Sollte die Approximation hierbei auf einem verkleinerten Datensatz beruhen, so wird die Geometrie beim Export entsprechend vergrößert.

Kapitel 5

Ergebnisse

In diesem Kapitel wird ein Überblick über die Ergebnisse dieser Diplomarbeit gegeben, die mit dem implementierten Verfahren erzielt wurden. Bevor die eigentlichen Resultate dargestellt werden, sollen die für den Test genutzten Datensätze vorgestellt werden.

Testdaten

Das Verfahren wurde an elf Datensätzen erfolgreich getestet. Davon sollen im Folgenden die Resultate zweier Datensätze ausführlich vorgestellt werden. Verwendet wurden zum einen die binären Segmentierungsdaten des menschlichen Schädels und zum anderen die einer Luftröhre. Diese Seg-

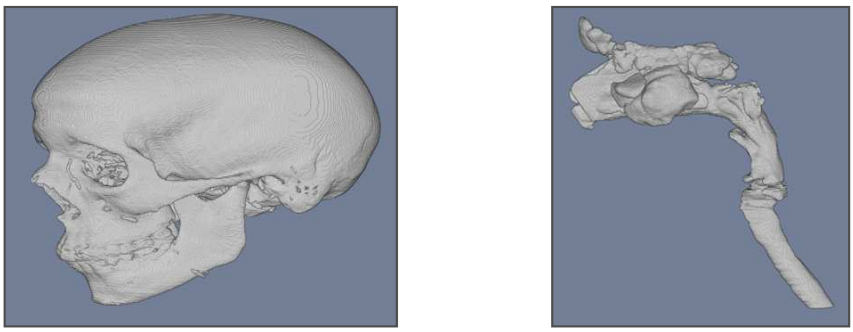


Abbildung 5.1: Testdatensätze: Menschlicher Schädel und Luftröhre.

mentierungsdaten basieren auf einem Datensatz der Größe ($512 \times 512 \times 333$). Dementsprechend haben beide Datensätze, unabhängig von der Größe des segmentierten Bereichs, ebenfalls eine Gesamtgröße von ($512 \times 512 \times 333$). Aus Gründen der Speicheranforderung wurden die Datensätze für die Generierung auf ($128 \times 128 \times 84$) herunterskaliert.

Die Wahl der Testdatensätze begründet sich wie folgt: Beide Datensätze enthalten die wesentlichen Probleme, die im Zuge der Beschreibung dieses Verfahrens erläutert wurden. So ergibt zum Beispiel die Konturendetektion bei der Luftröhre Konturen, die teilweise sehr unterschiedlich sind und daher erhöhte Anforderungen an das Triangulierungsverfahren stellen (vgl. Kap. 3.3.1 und 3.3.3).

Der Datensatz des Schädels dagegen enthält vor allem mehrere Löcher und Vertiefungen, die bei geringer Dämpfung problematisch seien können (vgl. Kap. 3.4.2).

Resultate

Nachfolgend sind die zu den Testdaten erzeugten Gitter abgebildet. Die Generierung erfolgte hierbei jeweils für vier verschiedene Detailstufen. Wobei die Detailstufen bekanntlich bei der Unterteilung mittels der Größe der minimal zulässigen Dreiecksflächen spezifiziert werden. Neben den einzelnen Abbildungen sei zudem auch auf die zugehörige Tabelle 5.1 verwiesen, die die Eigenschaften der

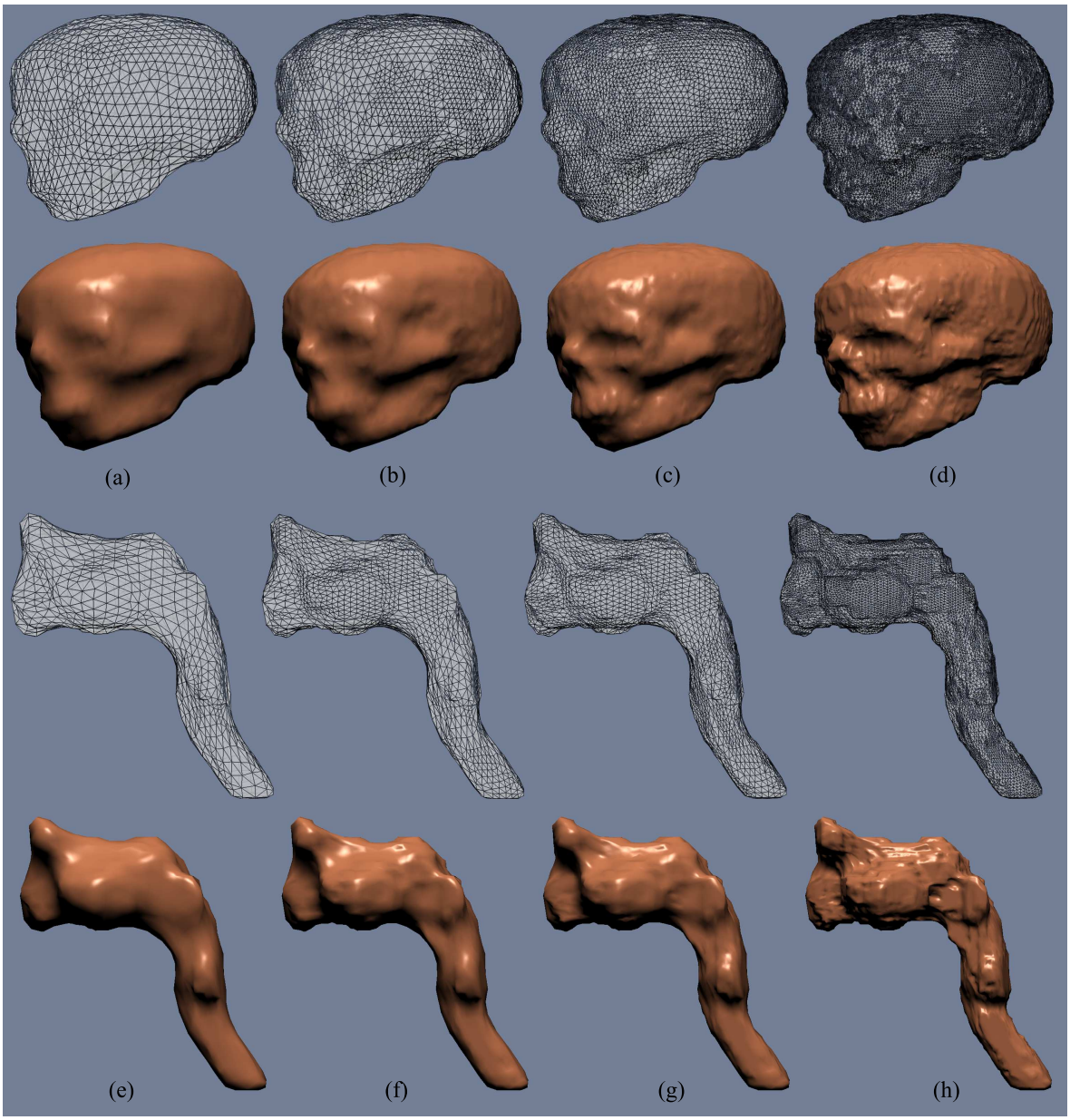


Abbildung 5.2: Resultate des Verfahrens in verschiedenen Detailstufen *(jeweils Gitter/Shaded)*.

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
LOD	40	10	5	2	10	5	3	1
Punkte	1692	3938	8217	20907	1204	2659	3952	13299
Kanten	5070	11808	24645	62715	3606	7971	11850	39891
Dreiecke	3380	7872	16430	41810	2404	5314	7900	26594
Tetraeder	32617	170419	595808	1937800	32020	105706	244955	1441010

Tabelle 5.1: Eigenschaften der erzeugten Dreiecksnetze und Anzahl der daraus generierten Tetraeder.

jeweiligen Dreiecksnetze und die entsprechend zugehörige Anzahl der Tetraeder enthält.
Zur Erinnerung: Das erzeugte Tetraedergitter dient als Proxygeometrie, zur Diskretisierung des Datenraums und nicht zur detaillierten Visualisierung des Volumens (vgl. Kap. 3.1.1).

Dementsprechend sind hohe Detailstufen, wie beispielsweise in *(c)*, *(d)*, *(g)* und *(h)*, für das Anwendungsgebiet der Volumendeformation nicht notwendig und zudem sehr speicherintensiv. Besonders die in *(d)* und *(h)* abgebildeten Gitter sind bereits so detailreich aufgelöst, dass die Voxelstruktur des zugrunde liegenden Datensatzes erkennbar wird. Die Gitter *(a)*, *(b)*, *(e)* und *(f)* sind dagegen, auch entsprechend des Performance-Tests in [Brü07], als durchaus praktikabel einzustufen.

Des Weiteren zeigen besonders die Abbildungen zum Schädel *(a-d)* die adaptive Unterteilung der Dreiecksflächen und somit das gezielte Hinzufügen weiterer Details. Generell ist bei der Betrachtung der Dreiecksnetze zu erkennen, dass die Dreiecksflächen größtenteils recht gleichmäßig sind. Diese Gleichmäßigkeit ist bekanntlich vor allem auf die Dämpfung des Netzes zurückzuführen. Doch trotz der Dämpfung und auch der allgemeinen Optimierung, ist die für das Verfahren typische Faltenbildung nicht auszuschließen (vgl. Kap. 3.5.3). Diese Faltenbildungen sollen jedoch nicht mit den Kanten in hochaufgelösten Netzen verwechselt werden, die auf die Voxelstruktur zurückzuführen sind.

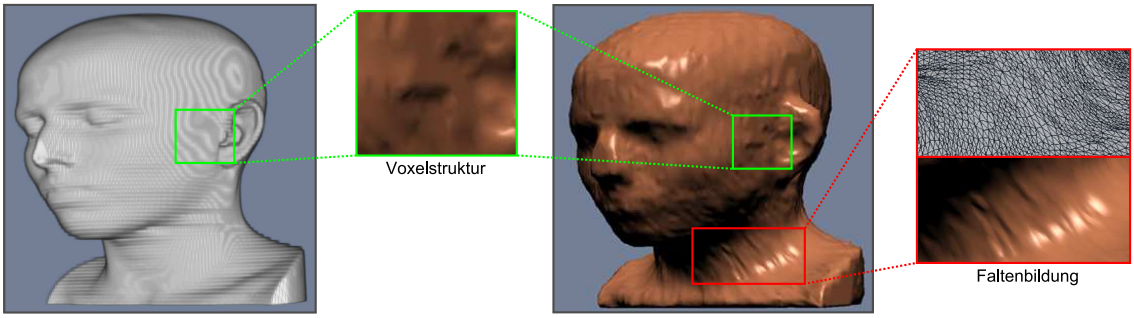


Abbildung 5.3: Sichtbare Voxelstruktur und verfahrenstypische Faltenbildung.

Wie zuvor thematisiert, ist eine gezielte Benutzerinteraktion für das Verfahren von fundamentaler Bedeutung, um gute Ergebnisse zu erzielen. Denn wie in Kapitel 3.3.4 beschrieben, ist die Konstruktion der initialen Dreiecksoberfläche maßgeblich von der Wahl der betreffenden Parameter abhängig. Dies sind vor allem die Anzahl der gewünschten Schichten, die Verarbeitungsrichtung und der Grad

der Glättung. Auch bei der, an die Konstruktion anschließenden, Anpassung ist die Interaktion des Benutzers von hoher Bedeutung. Bekanntermaßen hat der Benutzer die Möglichkeit, vor und zwischen den Anpassungsvorgängen, verschiedene Parameter zu modifizieren, um das Resultat entscheidend zu beeinflussen. Beispiele sind hier vor allem die Parameter für die Detailstufe, das Dekrement der Dämpfung und die Anzahl der Iterationen pro Anpassungsdurchgang. Neben der Qualität des Resultats ist auch die Performance des Verfahrens sehr stark von den Werten dieser Parameter abhängig. Aufgrund der Interaktivität und der Variabilität der Parameter soll von einer ausführlichen Analyse der Performance abgesehen werden.

Um jedoch einen Eindruck von der Performance zu geben, wird nachfolgend eine Übersicht zu den Verarbeitungszeiten unter Verwendung der implementierten Standardwerte gegeben. Die Standardwerte sind zehn Iterationen und ein Dekrement von „0.05“ und stellen einen guten Kompromiss aus Geschwindigkeit und Anpassungsvermögen dar. Die Messungen der Zeiten wurden auf einem System mit Intel Core 2 Duo E6750 (2,66 GHz), 2 GB RAM und einer GeForce 8800GT durchgeführt.

Zunächst werden die Einstellungen zur Konstruktion der initialen Dreiecksoberflächen und der daraus resultierenden Eigenschaften anhand der nachfolgenden Tabelle dargestellt. Denn basierend

Datensatz	**Einstellungen**	**Zeit**	Punkte	Kanten	Dreiecke
Schädel	21 Schichten in Z-Richtung [1×geglättet]	$\sim$ 1,4s	1480	4434	2956
Luftröhre	26 Schichten in Y-Richtung [1×geglättet]	$\sim$ 0,8s	934	2796	1864

Tabelle 5.2: Eigenschaften der initialen Dreiecksnetze.

auf diesen Dreiecksnetzen wurde, je Detailstufe, ein kompletter Anpassungsvorgang mit den Standardwerten durchgeführt. Die dabei gemessenen Zeiten geben demnach an, nach welcher Zeit ein Vorgang terminierte und eine erneute Benutzerinteraktion notwendig wurde. Wie zu erwarten, steigt

	Schädel				Luftröhre			
LOD	40	10	5	2	10	5	3	1
Dauer ($\sim$)	2,4s	5,1s	33s	239s	1,3s	2,6s	4,8s	81s

Tabelle 5.3: Dauer der Anpassungsvorgänge.

die Verarbeitungszeit bei feinen Detailstufen an. Die teils immense Vergrößerung des Zeitbedarfs ist allerdings vor allem auf die Optimierungsschritte zurückzuführen. Denn die Optimierung erfolgt auf der CPU und führt im Zuge dessen teure Iterationen, über die Listen der Entitäten, aus.

Bei den Detailstufen fünf und zwei des Schädels und der Detailstufe eins der Luftröhre terminierte die Anpassung allerdings aufgrund des Erreichens der maximalen Anzahl von Durchläufen. Wobei der Maximalwert die Anpassungsvorgänge standardmäßig auf zehn Durchläufe beschränkt. Die Ergebnisse waren jedoch in allen drei Fällen bereits so gut angepasst, so dass ein weiterer Durchgang keine beachtliche Verbesserung der Details erbracht hätte.

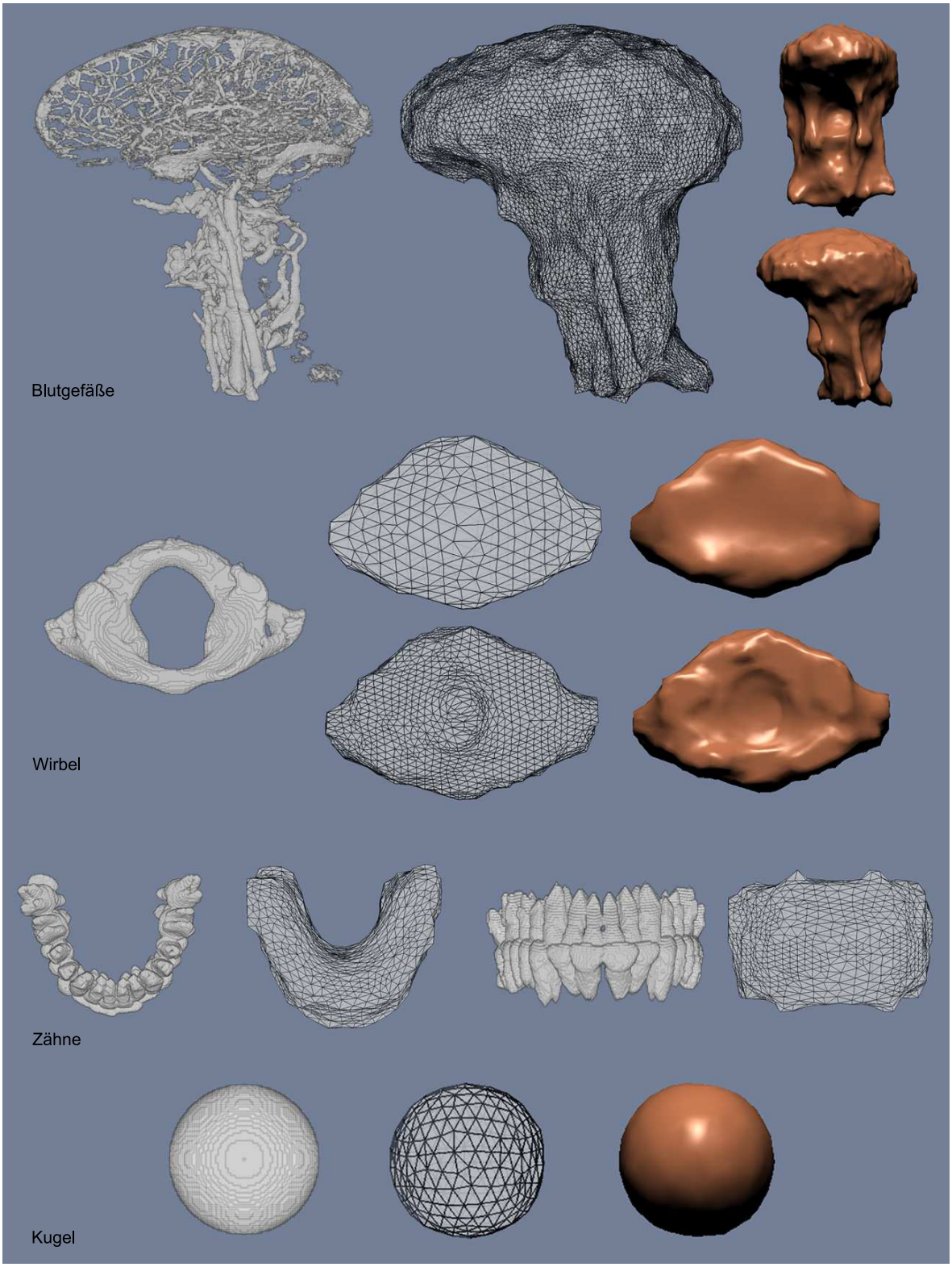


Abbildung 5.4: Weitere Resultate in einem veranschaulichenden Überblick.

Kapitel 6

Zusammenfassung und Ausblick

In diesem abschließenden Kapitel werden zunächst die wichtigsten Erkenntnisse der Arbeit zusammengefasst. Im Anschluss daran wird eine kritische Bewertung des Ansatzes erfolgen und ein Ausblick auf mögliche Optimierungen gegeben.

6.1 Zusammenfassung

Die Zielsetzung dieser Diplomarbeit war die Erzeugung approximierender Proxygeometrien, in Form von Tetraedergittern, für die direkte Deformation von Volumendaten. Die Anforderungen an die zu erzeugenden Geometrien richteten sich primär darauf, dass der segmentierte Bereich des Volumens komplett und geschlossen umhüllt, aber nicht bis in kleinste Detail aufgelöst wird.

Der Ansatz, der in dieser Arbeit vorgestellt und implementiert wurde, ermöglicht die Generierung solcher Hilfsgeometrien auf Basis eines Shrink-Wrapping-Verfahrens. Hierbei wird eine grobe, umhüllende und geschlossene Dreiecksoberfläche sukzessiv an die entsprechenden Volumendaten angepasst, indem das Netz mehr und mehr geschrumpft wird. Abschließend erfolgt, auf Basis der gefundenen Dreiecksoberfläche, die Generierung des Tetraedergitters mit Hilfe der freien Software *Gmsh*.

Die Konstruktion des initialen groben Dreiecksnetzes erfolgt nach dem Prinzip der *Surface from Contours*. Dementsprechend werden zu dem Volumen zunächst mehrere planare aktive Konturen bestimmt, die im Anschluss durch einen Triangulierungsschritt verbunden werden.

Die resultierende Oberfläche wird anschließend sukzessiv weiter an das Volumen angepasst. Der verwendete Shrink-Wrapping-Ansatz wird dabei um ein Verfahren zur adaptiven Unterteilung der Dreiecksflächen ergänzt. Hierdurch wird ermöglicht, dass dem Netz gezielt lokale Details hinzugefügt werden können. Die eigentliche Anpassung erfolgt, indem die Punkte der Geometrie durch den Einfluss einer inneren und äußeren Kraft bewegt werden. Hierbei fungiert die äußere Kraft, primär in Form des umgekehrten Gradienten, als Anzugskraft zur Anpassung des Netzes. Die innere Kraft wird dagegen aus der Laplacian-Glättung gewonnen und zur Stabilisierung verwendet, um Selbstüberschneidungen im Netz bestmöglich zu vermeiden. Die alleinige Verwendung der inneren

Kraft, kann jedoch nicht immer verhindern, dass Dreiecke der Oberfläche, aufgrund von Konvergenz und Divergenz, etwas verzerrt werden. Starke Verzerrungen können jedoch zu Selbstüberschneidungen führen. Daher wird gegebenenfalls noch ein Optimierungsschritt, zum Entfernen von degenerierten Dreiecken, angewendet.

Wenn auf diese Weise eine zufriedenstellende Dreiecksoberfläche erzeugt wurde, dann wird im letzten Schritt das gewünschte Tetraedergitter, mittels der freien Software *Gmsh*, generiert.

6.2 Kritische Bewertung

Tests haben ergeben, dass der oben beschriebene Ansatz in der Lage ist, bei zielgerichteter Benutzerinteraktion und geeigneter Parametrisierung, gute Resultate zu erzielen. Jedoch ist gerade in der Parametrisierung ein Schwachpunkt des Verfahrens zu sehen. Denn die Angabe einer allgemeingültigen Parametrisierung, mit der es möglich ist beliebige Datensätze unter allen Umständen effektiv und gut zu modellieren, kann, aufgrund der Komplexität des Problems, nicht gegeben werden. Eine gewählte Parametrisierung ist hier generell als eine Heuristik, zur Lösung des Problems, zu betrachten. Die optimale Lösung wird hierbei meist nicht gefunden. Doch durch das iterative Anwenden wird das Resultat immer weiter verbessert. Zweifellos kann leider nicht garantiert werden, dass für jede beliebige Parametrisierung ein zufriedenstellendes Ergebnis gefunden wird.

Bislang wird bei der Anpassung immer eine spezifizierte Anzahl von Iterationen durchgeführt. Durch Entwicklung eines Abbruchkriteriums könnte dieser Parameter entfallen. Hierbei ist jedoch zu beachten, dass wahrscheinlich wiederum ein Toleranzwert eingeführt werden muss, um entscheiden zu können, ab wann eine Anpassung abgeschlossen ist. Denn es können durchaus Zustände entstehen, indem Punkte sich gegenseitig so beeinflussen, dass sie sich immerzu „hin und her“ bewegen.

Die während der Oberflächenrekonstruktion detektierten Konturen sind recht grob, da die eigentliche Detailerzeugung während des Anpassungsvorgangs, mit Hilfe der adaptiven Unterteilung, stattfindet. Das in dieser Arbeit verwendete Triangulierungsverfahren arbeitet nach dem Greedy-Prinzip und liefert gute Resultate. Dies ist maßgeblich auf das komplexere Bewertungskriterium zurückzuführen. Es bleibt jedoch zu prüfen, inwiefern das Greedy-Prinzip auch, außerhalb dieses Kontextes, bei sehr komplexen Konturen ausreichend ist. Höchstwahrscheinlich wird hierbei ein Optimierungsverfahren (Graphensuche) notwendig werden.

In dem Kapitel 4.2.3 wurde das Problem der hohen Speicheranforderung erläutert. Generell ist die Lösung mittels Skalierung unerwünscht, da das Herunterskalieren einen Informationsverlust zur Folge hat. Doch wie auch in den Ergebnissen beschrieben, ist es gar nicht notwendig, dass die erzeugten Gitter sehr detailliert seien müssen. Essenziell ist jedoch, dass die Segmentierungsdaten komplett umhüllt werden. Wenn dies wider Erwarten nicht der Fall sein sollte, dann muss zunächst der Detailgrad des

initialen Dreiecksnetzes kontrolliert werden. In Extremfällen ist es zudem ratsam die Konstruktion der initialen Dreiecksoberfläche auf stärker geglätteten Segmentierungsdaten durchzuführen und dann je Anpassungsvorgang die Glättung wieder Schritt für Schritt zu reduzieren.

Die Möglichkeit zur Definition einer gewünschten Detailstufe macht dieses Verfahren recht adaptiv. Jedoch zeigte sich, dass bei feinen Detailstufen sehr hochaufgelöste und somit speicherintensive Gitter entstehen. Zudem steigt die Verarbeitungszeit bei feinen Einstellungen, primär wegen der Optimierung des Netzes auf der CPU, sehr stark an. Doch da diese Anwendung als Vorverarbeitung verwendet wird ist sie als nicht zeitkritisch einzustufen.

Im Allgemeinen erzeugt dieser Ansatz, durch das Verwenden der approximierenden Laplacian-Glättung, in Verbindung mit der Optimierung der Punktverteilung (vgl. Kap. 3.4.3), in der Regel qualitativ gute Netze. Das heißt, die Oberflächen bestehen primär aus recht gleichmäßigen Dreiecksflächen. Die verfahrenstypische Faltenbildung kann jedoch weiterhin nicht ausgeschlossen werden.

Fazit
Das in dieser Arbeit vorgestellte Verfahren kann nicht ohne Benutzerinteraktion auskommen. Doch bei zielgerichteter Verwendung und nicht zuletzt durch die Adaptivität der Detailstufe können qualitativ gute Hilfsgeometrien für die Volumendeformation generiert werden. Der primäre Schwachpunkt des Ansatzes ist jedoch, dass keine allgemeingültige Parametrisierung genannt werden kann.

6.3 Ausblick

Wie zuvor thematisiert, erfolgt die Optimierung des Dreiecksnetzes, in Form der adaptiven Unterteilung und dem Entfernen degenerierter Dreiecke, auf der CPU. Aufgrund der zahlreichen Iterationen steigt die Verarbeitungszeit bei detailreichen Netzen sehr stark an. Auch wenn der Ansatz einen Vorverarbeitungsschritt darstellt und somit als nicht zeitkritisch zu beurteilen ist, soll trotzdem die Möglichkeit zur Optimierung berücksichtigt werden. Zum Teil erfolgen in der vorliegenden Lösung derzeit Iterationen, die der Gebrauchstauglichkeit (engl. „Usability“) der Anwendung dienen. Zum Beispiel, um dem Benutzer zu visualisieren ob und wo Unterteilt werden muss. Dieser Nutzen kommt jedoch während der eigentlichen Anpassung nicht zum Ausdruck und dementsprechend können diese Iterationen hierbei eingespart werden. Des Weiteren könnte untersucht werden, inwieweit es möglich ist die Optimierung auf die GPU zu portieren.

Die Nutzung des Verfahrens ist ohne Kenntnisse des Schemas und der zugrunde liegenden Probleme als recht schwierig einzustufen. Dies begründet sich vor allem in der umfangreichen Parametrisierung. Mit dem Ziel das Verfahren praxistauglicher zu machen, sollte die Automatisierbarkeit der Parametrisierung zur Konstruktion der initialen Oberfläche und der Anpassung erforscht werden.

Die Parametrisierung der Anpassung ist bekanntlich ein komplexes Problem, dass sehr wahrscheinlich nicht einfach zu automatisieren ist. Derzeit haben die Parameter eine globale Gültigkeit. Das heißt, dass jeder Punkt zum Beispiel mit der gleichen Gewichtung bezüglich äußerer und innerer Kraft angepasst wird. Daher sollte bei der Erforschung des Problems mit der Untersuchung begonnen werden, inwiefern von lokalen Eigenschaften des Netzes Rückschlüsse auf eine entsprechend lokale Parametrisierung möglich sind.

Für die Automatisierung der anfänglichen Konstruktion der Oberfläche könnten beispielsweise Überlegungen angestellt werden, wie bestimmt werden kann an welchen Stellen definitiv eine Kontur benötigt wird. Ausgehend von diesen Stellen, könnte dann im Anschluss die entsprechende Konstruktion erfolgen. Eine erste vage Idee ist hier zu untersuchen inwiefern ein Histogramm des Volumens analytisch ausgenutzt werden könnte.

Die hohe Speicheranforderung bietet weitere Möglichkeiten zur Optimierung. Zur Erinnerung: Innerhalb der Vorverarbeitung werden zwei dreidimensionale FBOs (RGBA) verwendet. Wobei die GBA-Komponenten lediglich bei der Propagation für die Speicherung der kartesischen Koordinaten der Referenzpunkte benötigt werden. Anstelle der kartesischen Koordinaten wäre es möglich Polarkoordinaten zu verwenden und somit den Speicherbedarf je FBO um eine Komponente, also um ein Viertel, zu reduzieren. Außerdem ist es nicht immer nötig ein Distanzfeld zum gesamten Datensatz zu berechnen. Von Interesse ist nur der „direkte“ Bereich der Segmentierung. Auf diese Weise kann, in Fällen wo die Segmentierung im Verhältnis zum gesamten Datensatz recht klein ist, die Speicheranforderung drastisch reduziert werden.

Die Verarbeitung von beliebig großen Datensätzen, ohne Skalierung, wird jedoch auch mittels der beiden genannten Optimierungen nicht ermöglicht. Daher sollte dieses Problem weiter untersucht werden. Zu prüfen ist hierbei die Realisierbarkeit einer Bricking-Methode oder eines Nachbearbeitungsschritts zur weiteren Anpassung der vergrößerten Oberfläche.

Wie zuvor festgestellt, können mit diesem Ansatz sehr detailreiche Oberflächen erzeugt werden. Diese sind für den Kontext der Volumendeformation jedoch oftmals zu detailreich. Damit diese besser angepassten Netze nutzbar werden, muss deren Komplexität, unter Beibehaltung der Qualität, reduziert werden. Demnach ist es empfehlenswert den Einsatz von Algorithmen zur Mesh Simplification zu untersuchen.

Anhang A

Formeln und Codefragmente

A.1 Berechnung eines Dreiecks

Berechnungsformeln für ein beliebiges Dreieck mit den Seiten a, b und c.

$$\text{(Winkel)} \quad \alpha = \arccos\left(\frac{b^2 + c^2 - a^2}{2bc}\right), \ \beta \text{ und } \gamma \text{ analog} \tag{A.1}$$

$$\text{(Flächeninhalt)} \quad A = \frac{1}{2} a \cdot h_a \text{ mit } h_a = c \cdot \sin\beta \quad \text{oder analog für } b \text{ und } c \tag{A.2}$$

$$\text{(Umkreisradius)} \quad r = \frac{\alpha}{2\sin\alpha} = \frac{\beta}{2\sin\beta} = \frac{\gamma}{2\sin\gamma} \tag{A.3}$$

$$\text{(Inkreisradius)} \quad p = \frac{A}{s} \text{ mit } s = \frac{a+b+c}{2} \tag{A.4}$$

$$\tag{A.5}$$

A.2 Baryzentrische Koordinaten

Seien P_i, P_j und P_k die Eckpunkte eines Dreiecks. Dann kann der Punkt P bezüglich dieser Eckpunkte ausgedrückt werden als

$$P = a_i P_i + a_j P_j + a_k P_k \tag{A.6}$$

mit

$$a_i + a_j + a_k = 1 \tag{A.7}$$

Punkt P ist der Flächenschwerpunkt, wenn gilt

$$a_i = a_j = a_k = \frac{1}{3} \tag{A.8}$$

A.3 Entitäten der Winged-Edge Repräsentation

```
struct WE_Vertex
{
  int nEdgeIdx;    // Index zu einer zugehörigen Kante
  CVector vec;     // Punktkoordinaten
  // Attribute für Operationen
  int nValence;    // Valenz
  float fDistance; // Distanz zum Zielobjekt (Lookup SDT)
  bool bCorner;    // Eckpunkt des initialen Rechtecks?
};

struct WE_Face
{
  int nEdgeIdx;    // Index zu einer Kante des Faces
  // Attribute für Operationen
  double dAvgDist;            // durchschnittliche Distanz der Fläche
  bool   bSubdivide;          // Subdivision der Fläche?
  double dArea;               // Flächeninhalt
  double dRatioRadius;        // Verhältnis zwischen Inkreis/Umkreis
  double dSmallestAngleInTri; // kleinste Winkel im Dreieck
  double dShortestEdgeLength; // kürzeste Kante im Dreieck
};

struct WE_Edge
{
  int nVertexStart; // Der Index zum Startpunkt
  int nVertexEnd;   // Der Index zum Endpunkt
  // Orientierung im Uhrzeigersinn (CW)
  int nFaceCW;      // Index zur adjazenten Fläche
  int nPrevEdgeCW;  // Index zur vorherigen Kante
  int nNextEdgeCW;  // Index zur nächsten Kante
  // Orientierung gegen den Uhrzeigersinn (CCW)
  int nFaceCCW;     // Index zur adjazenten Fläche
  int nPrevEdgeCCW; // Index zur vorherigen Kante
  int nNextEdgeCCW; // Index zur nächsten Kante
  // Flags für Operationen
  bool bEdgeCollapse; // Benötigt diese Kante einen EdgeCollapse?
  bool bEdgeSwap;     // Benötigt diese Kante einen EdgeSwap?
};
```

Listing A.1: Die drei Strukturen der Winged-Edge Repräsentation.

Literaturverzeichnis

[Bau72] B. G. Baumgart. Winged-edge Polyhedron Representation. Technical Report STAN-CS-320, Computer Science Department, Stanford University, Stanford, CA, 1972.

[BGZ02] Hans-Joachim Bungartz, Michael Griebel, and Christoph Zenger. Einführung in die Computergraphik - Grundlagen, Geometrische Modellierung, Algorithmen; 2., überarbeitete und erweiterte Auflage. Book, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, June 2002.

[Brü07] Lisa Brückbauer. Hardwarebasiertes Volume Raycasting mit deformierbaren Tetraedernetzen. Diplomarbeit, Universität Siegen, 2007.

[CC91] Laurent D. Cohen and Isaac Cohen. Finite Element Methods for Active Contour Models and Balloons for 2D and 3D Images, July 11 1991.

[CK06] Nicolas Cuntz and Andreas Kolb. Fast Hierarchical 3D Distance Transforms on the GPU. Technical report, Institute of Vision and Graphics, Universität Siegen, 2006.

[CS78] H. N. Christiansen and T. W. Sederberg. Conversion of Complex Contour Line Definitions Into Polygonal Element Mosaics. *Computer Graphics*, 12(3):187–192, 1978.

[Cui99] Oliver Cuisenaire. *Distance Transformations: Fast Algorithms and Applications to Medical Image Processing.* PhD thesis, Université catholique de Louvain, 1999.

[FKU77] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal Surface Reconstruction from Planar Contours. *Communications of the ACM*, 20(10):693–702, October 1977.

[FvDFH96] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer graphics: Principles and practice in C.* Addison-Wesley, 2nd edition, 1996.

[GD82] S. Ganapathy and T. G. Dennehy. A New General Triangulation Method for Planar Contours. *Computer Graphics*, 16:69–75, 1982.

[GR08] Christophe Geuzaine and Jean-François Remacle. *Gmsh (Version 2.2.4)*, August 2008. GNU General Public License (GPL), http://www.geuz.org/gmsh.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh Optimization. *Proceedings of SIGGRAPH'93*, pages 19–26, 1993.

[JK02] Won-Ki Jeong and Chang-Hun Kim. Direct Reconstruction of a Displaced Subdivision Surface from Unorganized Points. *Graphical models*, 64(2):78–93, March 2002.

[JSC04] Wonhyung Jung, Hayong Shin, and Byoung K. Choi. B.K.: Self-intersection Removal in Triangular Mesh Offsetting. In *Computer-Aided Design and Applications*, pages 477–484, 2004.

[KCC+05] Bon Ki Koo, Young Kyu Choi, Chang Woo Chu, Jae Chul Kim, and Byoung Tae Choi. Shrink-Wrapped Boundary Face Algorithm for Mesh Reconstruction from Unorganized Points. *ETRI Journal*, 27(2):235–238, April 2005.

[KCVS98] Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. Interactive Multi-Resolution Modeling on Arbitrary Meshes. In *SIGGRAPH*, pages 105–114, 1998.

[Kep75] Eric Keppel. Approximating Complex Surfaces by Triangulation of Contour Lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975.

[KVLS99] Leif Kobbelt, Jens Vorsatz, Ulf Labsik, and Hans-Peter Seidel. A Shrink Wrapping Approach to Remeshing Polygonal Surfaces. *Comput. Graph. Forum*, 18(3):119–130, 1999.

[KWT88] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active Contour Models. *International Journal of Computer Vision*, 1(4):321–331, 1988.

[Mey94] David Meyers. *Reconstruction of Surfaces From Planar Contours*. PhD thesis, University of Washington, August 11 1994.

[Pap01] Lothar Papula. *Mathematik für Ingenieure und Naturwissenschaftler Band 3. Vektoranalysis, Wahrscheinlichkeitsrechnung, mathematische Statistik, Fehler- und Ausgleichsrechnung*. Vieweg+Teubner Verlag, 2001.

[RT06] Guodong Rong and Tiow Seng Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In Marc Olano and Carlo H. Séquin, editors, *SI3D*, pages 109–116. ACM, 2006.

[SGGM06] Avneesh Sud, Naga K. Govindaraju, Russell Gayle, and Dinesh Manocha. Interactive 3D distance field computation using linear factorization. In Marc Olano and Carlo H. Séquin, editors, *SI3D*, pages 117–124. ACM, 2006.

[SPG03] Christian Sigg, Ronald Peikert, and Markus Gross. Signed Distance Transform Using Graphics Hardware. In Greg Turk Jarke J. van Wijk and Robert J. Moorhead, editors,

Proceedings of IEEE Visualization 2003, pages 83–90. IEEE Computer Society, IEEE Computer Society Press, October 2003.

[Tau95] Gabriel Taubin. A signal processing approach to fair surface design. In *SIGGRAPH*, pages 351–358, 1995.

[TBWH85] Ulf Tiede, F. R. P. Boecker, G. Witte, and Karl Heinz Höhne. Eine neue Heuristik für die 3D-Rekonstruktion medizinischer Bildsequenzen mittels Triangulation. In Heinrich Niemann, editor, *DAGM-Symposium*, volume 107 of *Informatik-Fachberichte*, pages 207–211. Springer, 1985.

[Ter86] D. Terzopoulos. Regularization of Inverse Visual Problems Involving Discontinuities. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8:413–424, 1986.

[TWK87] D. Terzopoulos, A. Witkin, and M. Kass. Symmetry-Seeking Models and 3D Object Reconstruction. *International Journal of Computer Vision*, 1:211–221, 1987.

[vKGDT99] Sebastian von Klinski, Andreas Glausch, Claus Derz, and Thomas Tolxdorff. Modellbasierte Rekonstruktion von Organoberflächen auf der Basis von zweidimensionalen Schnittdaten. In *Bildverarbeitung für die Medizin*, pages 312–316, 1999.

[XP97] Chenyang Xu and Jerry L. Prince. Gradient Vector Flow: A New External Force for Snakes. In *CVPR*, page 66. IEEE Computer Society, 1997.