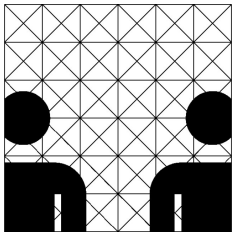
Universität Hamburg

MIN-Fakultät
Department Informatik

Arbeitsbereich Kognitive Systeme



Diplomarbeit

Repräsentation dreidimensionaler Segmentierungen durch 3-XG-Maps

Hamburg, Dezember 2006

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

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

Erstbetreuer:
Dr. Ullrich Köthe

Zweitbetreuerin:
Prof. Dr. Leonie Dreschler-Fischer

Arbeitsbereich Kognitive Systeme
MIN-Fakultät Department Informatik
Vogt-Kölln-Str. 30
22527 Hamburg

Danksagungen

An dieser Stelle möchten wir allen Personen danken, die diese Arbeit unterstützt haben.

Zuerst möchten wir uns bei unserem Erstbetreuer Dr. Ullrich Köthe bedanken, der uns das Thema dieser Arbeit vorgeschlagen hat und uns stets mit einem unerschöpflichen Fundus neuer Ideen und Anregungen zur Seite stand. Auch danken wir Prof. Dr. Leonie Dreschler-Fischer für die Zweitbetreuung. Außerdem möchten wir uns bei Hans Meine bedanken, der uns bei der Implementation einige Male hilfreich zur Seite stand. Auch Peer Stelldinger gebührt Dank. Er konnte uns vor allem bei Begriffsklärungen der Mannigfaltigkeiten weiterhelfen. Dieter Jessen danken wir dafür, dass er uns für die Zeit der Diplomarbeit ein Büro bei KOGS zur Verfügung gestellt hat.

Zudem bedanken wir uns bei unseren Korrekturlesern, die uns bei der Berichtigung dieser umfangreichen Arbeit unterstützt haben. Zu ihnen gehören Marc von Fintel, Catharina Neukirch, Evelyn Ehrhorn und Marco Möller.

Erklärung

Hiermit erklären wir, dass die vorlegende Arbeit von uns selbstständig erstellt wurde und dass wir nur die angegebenen Quellen und Hilfsmittel verwendet haben.

Außerdem erklären wir, dass wir mit der Einstellung dieser Diplomarbeit in den Bestand der Bibliotheken der Universität Hamburg einverstanden sind.

Hamburg, den .....

Benjamin Seppke	Florian Heinrich

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Diplomarbeit widmet sich dem Thema der Repräsentation dreidimensionaler Regionen im Rahmen der Bildverarbeitung. In den letzten Jahrzehnten wurden viele Algorithmen entwickelt, die Bilddaten verarbeiten und in verschiedene Regionen unterteilen. In vielen Fällen ist aber die Beurteilung dieser Unterteilungen schwierig, da wichtige Eigenschaften nicht direkt repräsentiert werden. In den letzten Jahren hat zudem die Bearbeitung dreidimensionaler Bilddaten immer mehr an Interesse gewonnen. Zum einen hat sich die verfügbare Leistung der Computer kontinuierlich gesteigert, zum anderen werden durch die verbesserten bildgebenden Verfahren in der Medizin immer mehr digitale Aufnahmen für die Untersuchungen verwendet. Aus diesem Grund entstehen immer mehr Aufgaben, die es erfordern, dreidimensionale Bilder in Regionen zu unterteilen und diese angemessen zu repräsentieren. Von entscheidender Bedeutung sind dabei die Eigenschaften der gefundenen Regionen.

In der Medizin ist es beispielsweise sinnvoll, aufgenommene Organe bezüglich ihrer Größe oder Form automatisch beurteilen zu können. Auch Fehlbildungen, wie zum Beispiel ein Loch in der Herzscheidewand, sollten so repräsentiert werden können, dass diese möglichst einfach zu erkennen sind. Ebenfalls kann die Lage von Organen im Körper oder zueinander ein wichtiges Kriterium sein, um Beurteilungen fällen zu können.

Ein weiterer Anwendungsbereich ergibt sich in der Beurteilung von Verfahren, die Bilddaten in Regionen unterteilen. Eine Analyse von Anzahl, Größe, Form und Lage der gefundenen Regionen lässt meist Rückschlüsse auf Stärken beziehungsweise Schwächen der verwendeten Verfahren zu.

Des Weiteren können mit Hilfe einer Repräsentation, die möglichst viele Informationen bereitstellt, weiterführende Bearbeitungen durchgeführt werden, die diese Informationen ausnutzen. Aufbauend auf einer solchen Repräsentation können zudem bekannte Methoden einfacher umgesetzt werden, so dass noch weitere Informationen gewonnen werden können.

Das Ziel dieser Arbeit ist es, aus beliebigen dreidimensionalen Bilddaten Regionen zu bestimmen und diese durch eine angemessene Repräsentation so darzustellen, dass Informationen über diese Regionen leicht zugänglich sind. Die Informationen, die in

dieser Arbeit repräsentiert und gewonnen werden können, sind sowohl geometrischer als auch topologischer Art. Während die Geometrie Informationen über die exakte geometrische Form beschreibt, abstrahiert die Topologie davon. Vielmehr repräsentiert sie Eigenschaften, die bei geometrischen Transformationen unverändert bleiben.

Beispielsweise stellen ein Torus und eine Tasse topologisch betrachtet äquivalente Objekte dar, auch wenn sie sich geometrisch kaum ähneln. Führt man auf einem Torus geometrische Transformationen aus, so kann man zu einer Tasse gelangen. Zu diesem Zweck wird zuerst der Torus an einer Stelle verdickt, so dass an dieser Stelle eine Kuhle hineingedrückt werden kann. Der Torus wird auf diese Weise in eine Tasse mit Henkel transformiert.

Zur Repräsentation von geometrischen und topologischen Informationen wird der Ansatz der generalisierten kombinatorischen Karten aufgegriffen und erweitert. Die in dieser Arbeit eingeführte 3-XG-Map stellt eine Möglichkeit dar, um geometrische und topologische Informationen dreidimensionaler Segmentierungsergebnisse zu repräsentieren. Die Erweiterung ist notwendig, da es Ausnahmefälle gibt, die sich durch eine normale generalisierte kombinatorische Karte nicht angemessen repräsentieren lassen.

Als Eingabedaten dienen Volumendatensätze, die zunächst durch ein Segmentierungsverfahren in Regionen unterteilt werden müssen. Aufbauend auf dieser Segmentierung werden dann die Datenstrukturen der 3-XG-Map erstellt. Mit Hilfe dieser Datenstrukturen können weitere Algorithmen umgesetzt werden, die ein Manipulieren erlauben, beziehungsweise die topologischen und geometrischen Informationen liefern. Eine Anwendung, die im Laufe dieser Arbeit präsentiert wird, ist das Herausfinden von Löchern in Regionen oder durch Regionen hindurch.

Die Vorgehensweise zum Erreichen der Ziele dieser Arbeit spiegelt sich in der Gliederung der einzelnen Kapitel wieder.

Im nächsten Kapitel werden die mathematischen Grundlagen dieser Arbeit zusammengefasst. Das Kapitel beinhaltet eine Einführung in die Graphentheorie und führt Permutationen ein, welche eine Grundlage der kombinatorischen Karten sind.

Das dritte Kapitel beschreibt, wie die Topologie im Kontext der Bildverarbeitung angewandt wird. Des Weiteren fasst es die Theorie der kombinatorischen Karten zusammen und führt die 3-XG-Map ein.

Da Nachbarschaften wichtige topologische Eigenschaften sind, wird im vierten Kapitel eine Kapselung zweier dreidimensionaler Nachbarschaften von Voxeln vorgestellt. Diese Kapselung kann universell angewandt werden.

Einen Anwendungspunkt der Nachbarschaften stellt der im fünften Kapitel vorgestellte Union-Find-Wasserscheiden-Algorithmus dar. In diesem Kapitel wird auch ein allgemeiner Überblick über Segmentierungsverfahren gegeben. Das Hauptaugenmerk liegt allerdings auf den Wasserscheiden-Verfahren.

Die danach folgenden beiden Kapitel beschreiben die Implementation der G-Map und der darauf aufbauenden dreidimensionalen XG-Map. Zudem werden einige Operationen auf diesen Datentypen vorgestellt.

Das achte Kapitel beschreibt die Möglichkeiten der visuellen Darstellung der implementierten Datenstrukturen. Dafür wurde eine Benutzungsoberfläche geschaffen, die außerdem die Erstellung der Datenstrukturen übernimmt und die vorgestellten Operationen in den vorherigen beiden Kapiteln einbindet.

Einige Beispiele und Experimente zeigt das neunte Kapitel. Dabei werden die Bearbeitung von Real-Welt-Volumen und die erzielten Ergebnisse diskutiert.

Im letzten Kapitel wird diese Arbeit bewertend zusammengefasst. Außerdem wird ein Ausblick auf mögliche Erweiterungen gegeben.

2 Elementare Definitionen

Dieses Kapitel fasst die mathematischen Grundlagen zusammen, die für diese Arbeit verwendet werden. Diese sind besonders wichtig, da auf ihnen die gesamte Arbeit aufbaut.

Dazu wird zunächst eine Einführung in die Graphentheorie gegeben. Diese ist entscheidend, wenn es um die topologische Repräsentation von Informationen geht, da mit Hilfe von Graphen Nachbarschaften sehr gut beschrieben werden können. Eine tragende Rolle spielt die Graphentheorie ebenfalls in Kapitel 5, wenn es darum geht, die Wasserscheiden-Transformation zu beschreiben.

Im Anschluss an die Graphentheorie werden Permutationen eingeführt, die, für die in Kapitel 3.4 folgenden kombinatorischen Karten, die mathematische Fundierung darstellen. Darauf aufbauend werden dann die Anker definiert, welche eine elegante Vereinfachung bei der Formalisierung der kombinatorischen Karten erlauben.

2.1 Graphentheorie

Die folgenden Notationen orientieren sich, soweit nicht anders angegeben, an [DP88][1]. Ähnliche oder gleiche Beschreibungen lassen sich in vielen Büchern der Mathematik finden, die die Graphentheorie behandeln. Dabei besteht aber selten ein Unterschied in der Bedeutung der Begriffe, so dass die Wahl auf dieses Buch fiel, da es sich eignete, die benötigten Sachverhalte kurz und verständlich zusammenzufassen.

Im Zusammenhang dieser Arbeit ist die Graphentheorie von Bedeutung, da sie formale Grundlagen schafft, Nachbarschaften und somit auch topologische Relationen zu beschreiben. Außerdem wird sie in Kapitel 5 dafür verwendet, eine bestimmte Klasse von Segmentierungsverfahren, die Wasserscheidenverfahren, zu definieren. Weiterhin stellt die Graphentheorie auch eine gute Veranschaulichung von Permutationen (siehe Kapitel 2.2) dar.

Definition 2.1.1 (Ungerichteter Graph) Ein ungerichteter Graph X besteht aus zwei disjunkten Mengen V (für *vertex*) und E (für *edge*) sowie einer Abbildung $f: E \to V^2$. Dabei heißt die Menge V **Knoten** des Graphen X, und E sind die **Kanten** des Graphen X.

In dieser Definition werden weder **Schlingen**, also Kanten, die den gleichen Startpunkt und Endpunkt – $f(e)=[v_n, v_n]$ – aufweisen, ausgeschlossen, noch wird untersagt, dass zwischen je zwei Knoten mehr als eine Kante verläuft, es also **Mehrfachkanten** – $f(e_1)=f(e_2)$ – gibt.

Definition 2.1.2 (Adjazenz) Zwei Knoten x und y aus dem Graphen X heißen adjazent, wenn es eine Kante e mit $f(e)=[x, y]$ gibt, die beide Knoten miteinander verbindet. Zwei Kanten heißen adjazent, wenn sie einen gemeinsamen Endpunkt haben.

Definition 2.1.3 (Inzidenz) Ein Knoten x und eine Kante e aus dem Graphen X heißen inzident, wenn $f(e)=[x, y]$, das heißt, einer der Endknoten von der Kante e der Knoten x ist.

Definition 2.1.4 (Grad eines Knotens) Der Grad $d(x)$ eines Knotens x im Graphen X ist die Anzahl der mit x inzidenten Kanten. Schlingen, die inzident zu x sind, werden doppelt gezählt.

Definition 2.1.5 (Isomorphismus) Es seien X, Y zwei Graphen mit den Knotenmengen $V(X)$ und $V(Y)$. Eine bijektive Abbildung $\phi: V(X) \to V(Y)$ heißt Isomorphismus von X auf Y, wenn $[x, y]$ genau dann eine Kante in $E(X)$ ist, wenn $[\phi(x), \phi(y)]$ eine Kante in $E(Y)$ ist. Gibt es einen Isomorphismus von X auf Y, so heißen die beiden Graphen isomorph.

Definition 2.1.6 (Kantenfolge) Eine Kantenfolge in einem Graphen X von einem Knoten x zu einem Knoten y ist eine endliche Folge von Knoten $x=x_1, x_2, x_3, \ldots, x_n=y$, wobei $[x_i, x_{i+1}] \in E$ für $i=1,2,\ldots,n-1$ ist.

Ist $x \neq y$, so heißt die Kantenfolge *offen*, ansonsten *geschlossen*.

Sind alle x_i unterschiedlich, so liegt ein **Kantenzug** vor.

1 Kapitel 8, S. 157-165

Ein **Weg** ist ein offener Kantenzug.

Sind alle Kanten unterschiedlich, so liegt ein **Pfad** vor.

Definition 2.1.7 (Kreis) Ein Kreis ist eine geschlossene Kantenfolge, in der bis auf den ersten und den letzten Knoten alle Knoten verschieden sind.

Definition 2.1.8 (Zusammenhängender Graph) Ein Graph *X* heißt zusammenhängend, wenn je zwei seiner Knoten durch einen Weg verbunden sind.

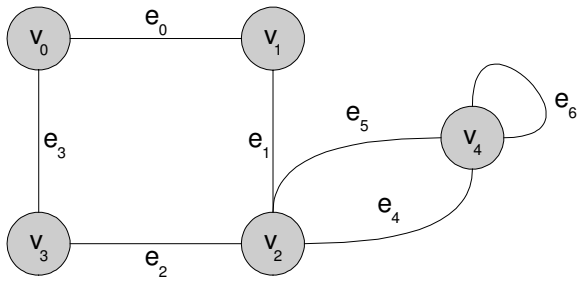


Abbildung 1: Beispiel eines ungerichteten zusammenhängenden Graphen mit der Knotenmenge $V=\{v_0, \ldots, v_4\}$ und der Kantenmenge $E=\{e_0, \ldots, e_6\}$. Die Kante e_6 stellt eine Schlinge dar, die Kanten e_4 und e_5 sind Mehrfachkanten zwischen v_2 und v_4. Ein Kreis im Graphen ist zum Beispiel gegeben durch die Kantenfolge: v_0, v_1, v_2, v_3, v_0.

Definition 2.1.9 (Zusammenhangskomponente) Eine Zusammenhangskomponente ist ein maximaler Teilgraph, bei dem zwischen je zwei Knoten ein Weg existiert.

Definition 2.1.10 (Gerichteter Graph) Ein gerichteter Graph *G* besteht aus einer endlichen Menge von Knoten *V* und einer Menge von geordneten Paaren (x, y) verschiedener Knoten $x, y \in V$. Die Menge der Kanten ist damit gegeben durch: $E \subset V \times V$. Die Elemente aus *E* heißen gerichtete Kanten von *G*.

Definition 2.1.11 (Gerichteter azyklischer Graph) Ein gerichteter azyklischer Graph ist ein gerichteter Graph, der keine Kreise enthält. Im weiteren Verlauf wird ein solcher Graph mit DAG („**d**irected **a**cyclic **g**raph“) bezeichnet.

Definition 2.1.12 (Planarer Graph) Ein planarer Graph ist ein Graph, dessen Kanten sich, wenn er auf eine Ebene abgebildet wird, nur in den Knoten treffen.

Bei dieser Definition ist zu beachten, dass ein Graph planar ist, sobald eine Projektion zu finden ist, bei der sich die Kanten nicht überschneiden. In Abbildung 2 ist ein Beispiel dafür zu sehen. So zeigen die ersten beiden Bilder den gleichen Graphen, der planar ist. Das dritte Bild zeigt einen nicht planaren Graph. Die Kante e_7 kann nicht so gelegt werden, dass sie keine der anderen Kanten schneidet.

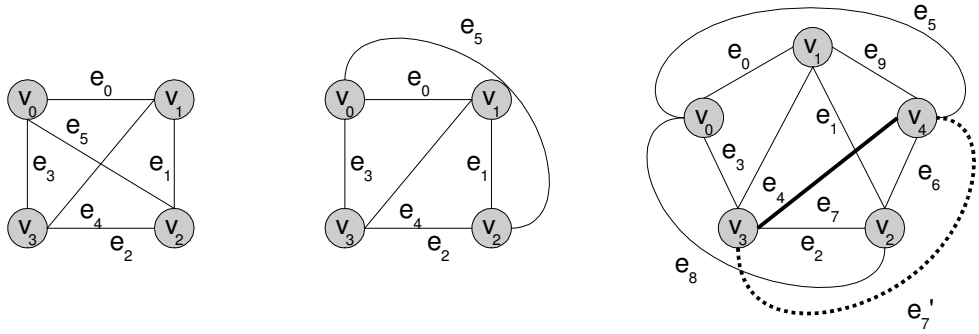


Abbildung 2: Beispiele für planare und nicht planare Graphen

Definition 2.1.13 (Baum) Ein zusammenhängender Graph ohne Kreise heißt Baum.

Definition 2.1.14 (Wurzelbaum) Ein Baum mit einem ausgezeichneten Knoten (der Wurzel) heißt Wurzelbaum.

Definition 2.1.15 (Wald) Eine Menge von einem oder mehreren Bäumen wird Wald genannt.

2.2 Permutationen

Permutationen stellen die Grundlage der kombinatorischen Karten dar. Sie bestimmen innerhalb der Karten, was benachbart ist, und welche Komponenten einen Knoten, eine Kante, eine Fläche etc. bilden. Ebenfalls wichtig sind sie bei der formalen Definition der im Rahmen dieser Arbeit entwickelten 3-XG-Map.

Die folgenden Notationen orientieren sich, soweit nicht anders angegeben, an [Big89][2]. Dieses Buch stellt die zu beschreibenden Sachverhalte in einer für diese Arbeit angemessenen Art dar und zeichnet sich so gegenüber anderer Literatur aus, bei der die Schwerpunkte anders gelegt sind.

Definition 2.2.1 (Permutation) Eine Permutation einer nicht leeren endlichen Menge *X* ist eine Bijektion von *X* nach *X*.

Für ein Beispiel eignet sich die Menge der natürlichen Zahlen $\mathbb{N}$. Als Beispiel sei α eine Permutation von $\mathbb{N}_5=\{1,\ldots,5\}$. Sie sei folgendermaßen definiert:

$$\alpha(1)=2,\quad \alpha(2)=4,\quad \alpha(3)=5,\quad \alpha(4)=1,\quad \alpha(5)=3$$

oder auch anders dargestellt:

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 2 & 4 & 5 & 1 & 3 \end{array}$$

Betrachtet man eine Permutation als Funktion, so sieht man, wie man sie verketten kann. Man kann so das obige Beispiel weiterführen, indem man einfach dieselbe Permutation nochmals anwendet. Man erhält die Permutation $\alpha\circ\alpha=\alpha^2$:

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 2 & 4 & 5 & 1 & 3 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 4 & 1 & 3 & 2 & 5 \end{array}$$

Natürlich kann man nicht nur dieselbe Permutation wieder anwenden, sondern jede, die auf dem gleichen Wertebereich definiert ist. Weiterführend zum obigen Beispiel kann eine Permutation β, die ebenfalls aus $\mathbb{N}_5$ sei, definiert werden. Die Verkettung $\alpha\circ\beta$ kann man auch als neue Permutation γ auffassen, die ebenfalls eine Permutation aus $\mathbb{N}_5$ ist.

Definition 2.2.2 (Permutationszyklus)[3] Eine Äquivalenzrelation ~ auf einer Menge *X*, gegeben eine Permutation α, sei wie folgt definiert:

$$x\sim y \Leftrightarrow \exists_{n\in\mathbb{N}}\ \alpha^n(x)=y$$

Die Äquivalenzklassen von *X* heißen dann Zyklen von α. Die Größe oder Länge eines solchen Zyklus ist gleich der Anzahl der Elemente, die ein solcher Zyklus enthält.

2 siehe [Big89] Kapitel 3, S. 55-59

3 siehe [Big89] Kapitel 14, S. 303

Eine andere Bezeichnung für Permutationszyklus ist **Orbit**.

Auf das obige Beispiel angewandt, ergibt sich beispielsweise, angefangen beim Element 1, folgendes Orbit: (1, 2, 4). Mit der Permutation α kommt man vom Element 1 zum Element 2, von dort aus zum Element 4 und dann zurück zu 1. Somit ergibt sich der Zyklus. Es lässt sich ein weiteres Orbit finden: (3, 5). Nachdem alle Orbits gefunden wurden, kann man die Permutation in Orbit-Notation angeben:

$$\alpha = (1, 2, 4)(3, 5).$$

(Ebenso lässt sich α als (5, 3)(4, 1, 2) oder ähnlich angeben.)[4]

4 siehe [Mei03] S. 12

2.3 Anker

Die Idee der Anker findet ihren Ursprung in der Diplomarbeit von Hans Meine [Mei03]. Da es in weiterer Literatur keine entsprechenden Notationen gibt, diese aber den gewünschten Sachverhalt ziemlich genau beschreiben, orientieren sich die folgenden Beschreibungen nah an [Mei03][4].

Am Ende des letzten Unterkapitels wurde verdeutlicht, dass man für eine Permutation verschiedene Orbit-Notationen angeben kann, je nachdem mit welchem Element die Zyklen begonnen werden. Das kann ein Problem werden, wenn man Orbits auf Computern speichern will. Durch ihre zyklische Struktur kann man sie beispielsweise sehr gut als eine verlinkte Liste speichern. Aber egal, wie man von Element zu Element kommt, ob mit einer Look-Up-Tabelle oder mit Pointern auf das folgende Element, man braucht immer einen Einstiegspunkt in ein Orbit. Diese werden im Folgenden **Anker** genannt. Eine gute Möglichkeit, einen Einstiegspunkt zu finden, ist es, einfach das kleinste Element als diesen zu erklären. Geht man so vor, so bleibt für das obige Beispiel nur noch eine Möglichkeit der Darstellung übrig, sofern man die Zyklen noch nach dem kleinsten Anker sortiert. Es ist also $\alpha = (1, 2, 4)(3, 5)$. Wählt man das kleinste Element als Anker, so erhält man einen kanonischen Anker.

Diese Anordnung kann man nun dahingehend nutzen, dass es ziemlich einfach ist, Orbits und somit auch Permutationen auf Gleichheit zu überprüfen. Das nächste Beispiel zeigt, wie die Wahl des kanonischen Ankers den Vergleich von Orbits erleichtert.

1. Orbits mit nicht kanonischem Anker

   $\alpha_1 = (1, 2, 4, 3)$ $\alpha_2 = (4, 3, 1, 2)$ $\alpha_3 = (2, 4, 1, 3)$

Bei diesen Orbits fällt es relativ schwer zu entscheiden, ob sie gleich sind. Im nächsten Schritt werden deshalb noch einmal dieselben Orbits mit kanonischem Anker dargestellt.

2. Orbits mit kanonischem Anker

   $\alpha_1 = (1, 2, 4, 3)$ $\alpha_2 = (1, 2, 4, 3)$ $\alpha_3 = (1, 3, 2, 4)$

Werden die Orbits in dieser Form repräsentiert, erkennt man sofort $\alpha_1 = \alpha_2 \neq \alpha_3$.

In dieser Arbeit werden die Anker vor allem für die Speicherung der Strukturen der 3-XG-Map (siehe Kapitel 7) verwendet. Dabei wird jede einzelne Struktur durch einen eindeutigen Anker beschrieben. Mit Hilfe von Relationen können dann die weiteren Informationen erlangt werden.

3 Topologie in der Bildverarbeitung

Dieses Kapitel stellt eine Einleitung in die Topologie dar, wie sie in der Bildverarbeitung verwendet wird. Allerdings wird sich im Folgenden darauf beschränkt werden, wie Regionen innerhalb von Bildern als Zellkomplexe und mit Hilfe von kombinatorischen Karten repräsentiert werden können. Auch wenn dies nur einen kleinen Bereich der Rolle der Topologie in der Bildverarbeitung darstellt, so ist dies im Rahmen der vorliegenden Arbeit ausreichend, um die Grundlagen topologischer Art zu beschreiben.

Es wird zunächst geklärt werden, was man unter einem digitalen Bild zu verstehen hat und wie solche Bilder repräsentiert werden können. In diesem Kontext wird auch auf Bilder der unterschiedlichen Dimensionen eingegangen. Dabei wird mit zweidimensionalen Bildern begonnen, bevor zu dreidimensionalen Bildern übergegangen wird.

Im Anschluss daran wird kurz vorgestellt, auf welche Weise sich Regionen innerhalb von Bildern repräsentieren lassen. Darauf folgend werden einige vorbereitende Definitionen zu Zellkomplexen und deren topologischen Eigenschaften vorgestellt, bevor die Theorie der kombinatorischen Karten und dessen Entstehung beziehungsweise Analogie zur Graphentheorie in einem umfassenden Überblick erläutert wird. Dabei werden verschiedene Modelle der Repräsentation vorgestellt und analysiert. Abschließend wird das im Rahmen dieser Arbeit entwickelte Modell der XG-Map formal definiert und erläutert.

3.1 Repräsentation von Regionen

Dieses Unterkapitel beschreibt, wie man Regionen in Bildern formal beschreiben kann. Dazu muss zunächst einmal geklärt werden, was ein Bild ist, und welche Eigenschaften es haben muss, damit es von einem Computer verarbeitet werden kann. Regionen sind Bereiche in Bildern, die auf bestimmte Weise zusammengehören. Es werden verschiedene Möglichkeiten der Repräsentation erläutert, nachdem die grundsätzlichen Fragen über die Bildrepräsentation geklärt sind.

Zunächst wird somit beschrieben, was unter einem digitalen Bild in dieser Arbeit verstanden wird. Im weiteren Verlauf werden verschiedene Formen der Verarbeitung von Bildern erwähnt oder vorgestellt. Damit diese Verarbeitungen gelingen, müssen die Bildinformationen in einer Form, die für den Computer verständlich ist, repräsentiert werden. Die einführenden Definitionen beziehen sich zwar auf zweidimensionale Bilder, darauf aufbauend werden allerdings ebenso dreidimensionale Bilder beschrieben. Diese Vorgehensweise hat sich als sehr sinnvoll erwiesen, da sich die Definitionen der komplizierteren dreidimensionalen Bilder nahezu analog aus denen der einfacheren zweidimensionalen ergeben.

Definition 3.1.1 (Grauwertbild)[5] Ein Grauwertbild entspricht einer zweidimensionalen Lichtintensitätsfunktion $f(x,y)$, wobei x und y räumliche Koordinaten sind und der Wert von f an einem Punkt (x,y) proportional zur Helligkeit des Bildes an diesem Punkt ist.

Für ein Farbbild muss man die Funktion f dahingehend verändern, dass sie nicht die Helligkeit des Bildes angibt, sondern die Farbe. Dies kann man realisieren, indem f auf die Helligkeiten der drei Farben Rot, Grün und Blau abbildet, das heißt man betrachtet die Intensität der Spektren der drei Farben einzeln. f stellt sich dann dar als $f(x,y)=(r,g,b)$. Auf diese Art und Weise erhält man ein Bild im RGB-Farbraum. Es gibt noch andere Farbräume, die man beispielsweise in [GW92] auf Seite 225ff finden kann. Diese werden aber für diese Arbeit nicht weiter benötigt und deshalb auch nicht beschrieben.

Mit der obigen Definition ist bisher allerdings nur ein analoges Bild beschrieben. Damit ein Bild von einem Computer verarbeitet werden kann, muss es noch in eine diskrete Form überführt werden. Dazu ist es wichtig, die Ausmaße des digitalen Bildes zu bestimmen. Die Größe eines Bildes bestimmt sich durch die Anzahl der Pixel (aus dem Englischen für „picture elements“). Diese Pixel werden beispielsweise in Matrixform angeordnet (siehe unten), so dass man das analoge Bild **rastert** (siehe hierzu auch die Abbildung 3 links). In einem Koordinatensystem betrachtet verläuft die X-Achse von links nach rechts und die Y-Achse von oben nach unten, was bei der Betrachtung von digitalen Bildern üblich ist. Der Ursprung eines Bildes liegt somit oben links. Die Zeilen und Spalten werden in diesem Sinne angeordnet. Die Anzahl der Pixel ergibt sich aus der Multiplikation der jeweiligen Anzahlen von Zeilen und Spalten.

Nach der Rasterung muss noch der Wertebereich festgelegt werden. Denn ein Computer kann an jeder Stelle der Bildmatrix nur einen diskreten Wert speichern. Dazu bestimmt man einen Bereich von Werten, die die Pixel eines Grauwertbildes annehmen können. In der Regel nimmt man hier ein Intervall von ganzen Zahlen, wie beispielsweise [0,255]. Das Bestimmen der diskreten Repräsentanten der analogen Pixelwerte nennt man

5 [GW92] S. 6 dient als Vorlage

Quantelung. Die Quantelung kann man durchführen, indem man das kontinuierliche Intervall mit Werten vom dunkelsten bis zum hellsten Bildpunkt in beispielsweise 256 Teilintervalle zerlegt. Diese Intervalle müssen nicht zwangsläufig gleich groß sein, sondern hängen vielmehr vom Bild oder der Anwendung ab. Danach werden alle Werte innerhalb eines Intervalls dem diskreten Wert zugeordnet, der dieses Intervall repräsentiert.

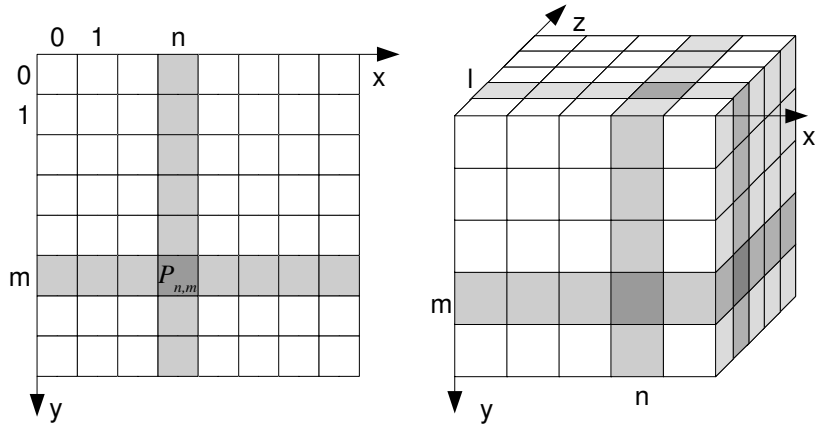


Abbildung 3: Darstellung der Achsen eines gerasterten Bildes. Links: zweidimensionales Bild, rechts: Volumen. (Abbildung aus [Jäh97] S. 30)

Es gibt verschiedene Möglichkeiten ein Bild zu rastern. Zu dem bereits erwähnten quadratischen Gitter kommen noch das Dreiecksgitter und das hexagonale Gitter hinzu. Abbildung 4 zeigt diese drei Möglichkeiten. Das Dreiecksgitter besteht aus dreieckigen Rasterstellen, das quadratische Gitter aus quadratischen Rasterstellen, welche die allgemein bekannte Art der Rasterung darstellen. Diese Art der Rasterung findet heute beispielsweise bei den CCD-Chips von Digitalkameras oder beim Aufbau von Bildschirmen aller Art (Fernseher, Computerbildschirm etc.) Anwendung. Die letzte Art der Rasterung ist die hexagonale. Die Pixel des Bildes bestehen bei dieser Art aus Sechsecken, die in einer Art Wabenform angeordnet sind. Diese Art der Rasterung entspricht am ehesten der des menschlichen Auges.

Bis zu dieser Stelle wurde nur auf zweidimensionale Bilder eingegangen, obwohl diese Arbeit sich überwiegend mit dreidimensionalen Bildern, sogenannten Volumen, beschäftigt. Bei Volumen gibt es nicht nur zwei Achsen, sondern es kommt noch eine dritte hinzu, die die Tiefe angibt, in der man sich im Bild befindet. Die Funktion *f* aus der Definition 3.3.1 wird also um eine weitere Koordinate erweitert. Dabei entspricht *f(x,y,z)* der Intensität des Bildes an der Stelle *(x,y,z)* des Raumes, der im Bild festgehalten werden soll.

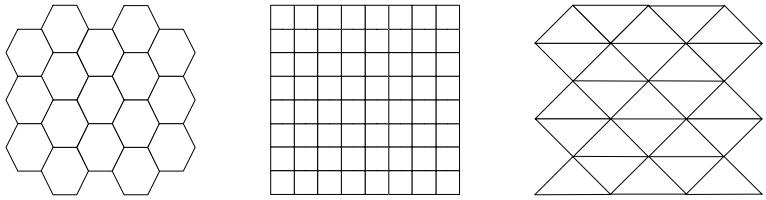


Abbildung 4: Es gibt verschiedene Möglichkeiten ein Bild zu rastern. Links: hexagonale Rasterung. Mitte: quadratische Rasterung. Rechts: trigonale Rasterung. (vgl. [Jäh97] S. 32)

Die Rasterung des dreidimensionalen Bildes erfolgt zumeist in der quadratischen Form. Aus den Quadraten werden allerdings durch Hinzunahme der dritten Koordinate Würfel. Dieser Sachverhalt wird in der Abbildung 3 (rechts) veranschaulicht. Dieser Abbildung kann man auch entnehmen, wie die Achsen verlaufen. In dieser Arbeit verläuft, soweit nicht anders angegeben, die X-Achse immer von links nach rechts, die Y-Achse immer von oben nach unten und die Z-Achse von vorne nach hinten. Das Koordinatensystem entspricht also dem eines rechtshändigen Systems. Der Ursprung liegt für den beschriebenen Fall vorne links oben. Die Quantelung unterscheidet sich nicht von der zweidimensionaler Bilder. Die Würfel, die beim Rastern einer Volumenszene entstehen, nennt man Voxel (aus dem Englischen für „volume element“). Sie entsprechen den Pixeln in Bildern, was besonders deutlich wird, wenn man ein Volumen mit der Tiefe eins betrachtet, welches wieder einem zweidimensionalen Bild entspricht.

Da es sehr schwer ist, eine dreidimensionale Szene in einem Stück mit einem technischen Gerät aufzuzeichnen, werden Volumen häufig dadurch erzeugt, dass viele zweidimensionale Bilder mit der quadratischen Rasterung hintereinander aufgenommen werden. Diese werden dann in der richtigen Reihenfolge in den Computer eingelesen. Verwendet man dafür ein dreidimensionales Array, so stellen die ersten beiden Koordinaten die Bildkoordinaten jedes einzelnen Bildes dar, und die dritte Koordinate repräsentiert die Bildnummer. Volumen entstehen beispielsweise bei medizinischen Untersuchungen mit einen CT-Scanner, bei dem ein Körper schichtenweise geröntgt wird. Diese einzelnen Schichtenbilder hintereinander angeordnet ergeben ein Volumen.

Sowohl in zweidimensionalen, wie auch in dreidimensionalen Bildern gibt es Bereiche, die zusammengehören, und somit auch als zusammengehörig repräsentiert werden sollten. Diese Bereiche in einem Bild sind beispielsweise diejenigen Bildpunkte, die ein und das selbe Objekt repräsentieren. In der Bildverarbeitung spielt es eine sehr große Rolle, dass man beschreiben kann, wo Objekte in einem Bild liegen – sei es durch Angabe von Bildpunkten (Pixel oder Voxel) oder durch relative räumliche Beschreibungen der Objekte zueinander. Regionen in digitalen Bildern lassen sich also auf verschiedene Arten beschreiben. Als Region in einem Bild bezeichnet man diejenigen Bildpunkte, die zusammen eine bedeutungstragende Einheit bilden. Dazu ist es notwendig, dass diese Bildpunkte miteinander verbunden sind (siehe auch Kapitel 3.2).

An dieser Stelle werden drei Varianten der Regionenbeschreibung erläutert (siehe [Köt00] S.164ff).

Ikonische Repräsentation: Die Regionen werden in Form eines digitalen Bildes dargestellt. Bildpunkte der selben Region erhalten den gleichen Farbwert.

Geometrische Repräsentation: „Jedes Objekt, bzw. jedes Teil eines Objektes, wird durch ein parametrisiertes geometrisches Modell beschrieben, wobei die Parameter aufgrund von Messungen in einem oder mehreren Bildern festgelegt werden.“[6]

Topologisch-geometrische Repräsentation: Bei dieser Sicht auf die Darstellung der Regionen kommt es vor allem auf die räumlichen Beziehungen zwischen den einzelnen Bildregionen an. Beispiele dieser Beziehungen sind die Betrachtung der Nachbarschaft oder, ob eine Region von einer anderen umschlossen ist. Meist werden den Bildmerkmalen auch geometrische Informationen zugeordnet.

Die einfachste Art der Repräsentation von Regionen ist die ikonische. Bei dieser Variante ist keine neue Datenstruktur notwendig, um die Informationen über den Zusammenhang von Bildpunkten zu speichern. Da Bildpunkte einer Region durch ihre Farbe zugeordnet werden (siehe dazu auch den Algorithmus in 5.3), kann man die Informationen direkt aus einem digitalen Bild ablesen. Allerdings gibt es bei dieser Darstellung einige Probleme. So ist es beispielsweise nicht immer eindeutig, wann zwei Regionen zusammengehören (also nur eine sind) oder wann sie getrennt sind (dazu später mehr in Abschnitt 3.2.1 und 3.2.2). Ein Bild in der ikonischen Regionenrepräsentation nennt man auch Label-Bild oder gelabeltes Bild.

Die geometrische Repräsentation versucht Regionen durch geeignete parametrisierte geometrische Prototypen darzustellen. Die Parameter dieser Prototypen werden durch die Informationen im Bild angepasst. Bei Punkten werden üblicherweise die Koordinaten im Bild als Parametrisierung verwendet. Geraden lassen sich durch zwei Punkte beschreiben. Bei gekrümmten Kurven gibt es ebenfalls Möglichkeiten, diese zu beschreiben. Dafür bieten sich Segmente von Kreisen, Ellipsen oder auch Hyperbeln an. Aber auch beliebige Kurven lassen sich mathematisch beschreiben. Dafür eignen sich beispielsweise Splinekurven. Flächenregionen können als Polygone aus diesen Kurven dargestellt werden. Mehrere Flächenregionen beschreiben zusammen wiederum Volumenregionen (vgl. XG-Map in Kapitel 3.4.5). Nachteilig ist allerdings, dass die Parametrisierung relativ schwer zu realisieren ist und großen Aufwand mit sich bringt. Des Weiteren fehlen Informationen über die Nachbarschaft von Regionen oder anderen topologischen Beziehungen. Geometrische Repräsentationen werden deshalb auch häufig in Kombination mit anderen Repräsentationen verwendet.

Bei der topologischen Repräsentation wird von den konkreten Eigenschaften der Objekte im Bild abstrahiert. Das Augenmerk liegt hauptsächlich auf deren gegenseitigen Beziehungen. Regionen, die die Objekte im Bild darstellen, haben nur eine untergeordnete Funktion, denn bei der topologischen Repräsentation ist es bedeutungslos, welche Bildpunkte von den Objekten eingenommen werden, sobald die topologischen Beziehungen geklärt sind. Auch die geometrischen Eigenschaften spielen keine Rolle,

6 aus [Köt00] S. 166

denn sie haben im Allgemeinen keine Auswirkung auf die Topologie. Da aber häufig ein Bezug zu den ursprünglichen Bilddaten erhalten werden soll, ist eine Erweiterung der topologischen Repräsentation um eine der anderen beiden genannten Repräsentationen sinnvoll. Man wendet deshalb auch häufig eine topologisch-geometrische Repräsentation an.

In dieser Arbeit lassen sich alle drei Repräsentationen finden. In Kapitel 5, in dem Verfahren zur Bildsegmentierung vorgestellt werden, wird ein Verfahren beschrieben, das gegebene Bilder in Regionen zerlegt, indem jede Region durch eine bestimmte Farbe gekennzeichnet wird. Die segmentierten Bilder stellen also eine Art der ikonischen Repräsentation dar. Die G-Map aus Kapitel 3.4.3 eignet sich sehr gut, um eine topologische Repräsentation von Regionen vorzunehmen. Wie später deutlich wird, stellt die G-Map aber ebenso Möglichkeiten der geometrischen Repräsentation (siehe Speicherung der geometrischen Einbettung in Abschnitt 3.4.3) zur Verfügung. Allerdings ist die geometrische Beschreibung der XG-Maps (hierzu Kapitel 3.4.5) deutlich konkreter. Bevor allerdings diese Sachverhalte näher beschrieben werden, müssen zunächst einige weitere Grundlagen geklärt werden.

3.2 Nachbarschaften von Bildpunkten

In zweidimensionalen Bildern und in dreidimensionalen Volumen ist eine elementare Frage, wann zwei Bildpunkte aneinander liegen und wann nicht. Erst wenn geklärt ist, wann dies der Fall ist, können weitere Operationen auf das Bild angewandt werden (vergleiche Kapitel 5.3). Ein Beispiel für die Bedeutung der Nachbarschaft ist das Zählen von Segmenten. Je nach gewählter Nachbarschaft gehören nämlich einige Bildpunkte zu einem Segment oder bilden ein neues Segment. Dies hängt davon ab, ob sie zu einem Bildpunkt des Segmentes als benachbart angesehen werden oder nicht.

Im folgenden Abschnitt 3.2.1 werden zunächst die Nachbarschaften von Bildpunkten in der Ebene (also von Pixeln) definiert, wie sie bei der Bearbeitung von zweidimensionalen Bildern angewandt werden. Dieses dient der Einführung und der einfacheren Veranschaulichung. Darauf aufbauend erfolgt dann die Erweiterung auf Bildpunkte im Raum (Voxel). In Abschnitt 3.2.2 werden deshalb die allgemein üblichen Nachbarschaftsbeziehungen im dreidimensionalen Raum eingeführt. Dieses Unterkapitel ist zudem als Einführung in das Kapitel 4 zu verstehen, da die vermittelte Theorie die Grundlage der dort beschriebenen Implementation darstellt.

3.2.1 Nachbarschaften in der Ebene

In der Ebene unterscheidet man in der Regel zwischen zwei verschiedenen Nachbarschaftsbeziehungen. Dabei spielt es eine Rolle, ob man Pixel auch dann als zusammenhängend betrachtet, wenn sie im Pixelraster diagonal versetzt zueinander liegen, oder nicht. Man unterscheidet daher die 4er-Nachbarschaft von der 8er-Nachbarschaft.

Definition 3.2.1 (4er-Nachbarschaft)[7] Zwei Pixel P_1 und P_2 heißen benachbart im Sinne der 4er-Nachbarschaft, falls sich genau eine Koordinate von P_1 von der korrespondierenden Koordinate von P_2 um genau eins unterscheidet. Formal ausgedrückt ergibt sich für diese beiden Pixel mit den Koordinaten (x_1, y_1) beziehungsweise (x_2, y_2) folgende Bedingung:

ist $|x_1 - x_2| + |y_1 - y_2| = 1$, dann sind P_1 und P_2 benachbart.

Definition 3.2.2 (8er-Nachbarschaft)[5] Zwei Pixel P_1 und P_2 heißen benachbart im Sinne der 8er-Nachbarschaft, falls P_1 ungleich P_2 ist und sich jede Koordinate von P_1 von der korrespondierenden Koordinate von P_2 um höchstens eins unterscheidet. Formal ausgedrückt ergibt sich für diese beiden Pixel mit den Koordinaten (x_1, y_1) beziehungsweise (x_2, y_2) folgende Bedingung:

ist $max(|x_1 - x_2|, |y_1 - y_2|) = 1$, dann sind P_1 und P_2 benachbart.

Man kann beide Nachbarschaften veranschaulichen, indem man jeden Pixel jeweils als Rechteck ansieht. Zwei Pixel sind dann 4-benachbart, wenn die Rechtecke eine gemeinsame Kante haben; sie sind 8-benachbart, wenn sie zumindest einen gemeinsamen Eckpunkt haben. Bildlich verdeutlicht diesen Sachverhalt Abbildung 5.

7 vgl. [Jäh97] S. 33f

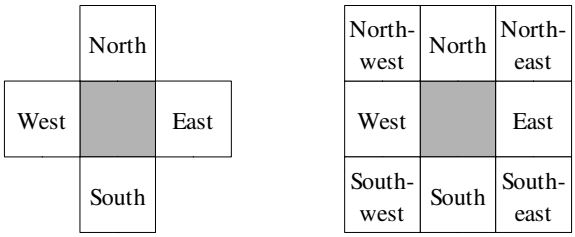


Abbildung 5: Benannte Nachbarn eines Pixels (grau); Links: 4er-Nachbarschaft; Rechts: 8er-Nachbarschaft

Auf den oben definierten Nachbarschaften lassen sich nun Eigenschaften der Verbundenheit definieren:

Definition 3.2.3 (Pfad)[6] Ein n-Pfad ist eine Folge von Pixeln $P_0, P_1, \ldots, P_m$, wobei jeweils P_i und P_{i+1} $(0 \le i \le m)$ n-benachbart sind.

Definition 3.2.4 (Verbundenheit)[8] Eine Menge von Pixeln $\boldsymbol{A}$ heißt n-verbunden, wenn es für je zwei Pixel einen n-Pfad von Pixeln aus $\boldsymbol{A}$ zwischen ihnen gibt.

Beide Nachbarschaften führen zu Konnektivitätsproblemen. Wie man in Abbildung 6 sieht, erhalten die Regionen je nach gewählter Nachbarschaft eine unterschiedliche Bedeutung. Legt man die 4er-Nachbarschaft zugrunde, so erhält man zwei Regionen von weißen Pixeln, da die Pixel in der Mitte nicht mit den äußeren verbunden sind. Dieses erscheint paradox, da die Region in den dunklen Pixeln keine geschlossene Kontur ist, sondern ebenfalls aus zwei Teilen besteht. Legt man hingegen die 8er-Nachbarschaft zugrunde, so stellt zwar die dunkle Region eine geschlossene Kontur dar, allerdings sind nun die Regionen der weißen Pixel ebenfalls verbunden. Es sind also Regionen verbunden, obwohl sie durch eine geschlossene Kontur voneinander getrennt sind. Dies stellt einen Widerspruch zum folgenden Satz dar.

Satz 3.2.1 (Jordanscher Kurvensatz)[9] Jede einfach geschlossene Kurve im $\mathbb{R}^2$ zerlegt den $\mathbb{R}^2$ in zwei disjunkte Regionen, deren gemeinsamer Rand diese Kurve ist. Eine der Regionen ist die innere, die andere die äußere Region.

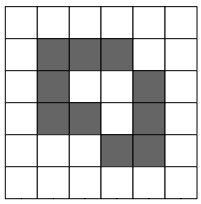


Abbildung 6: Darstellung des Konnektivitäts-Paradoxons. Wird für Vordergrundpixel (grau) und Hintergrundpixel (weiß) die 8er-Nachbarschaft zugrunde gelegt, so entsteht ein Widerspruch zum Jordanschen Kurvensatz.

8 analog zu [NP01] S. 67

9 entsprechend zu [Wik06c] beziehungsweise [Die06] S. 93

Um dieses Problem dieses Paradoxons zu umgehen, kann man Kanten zwischen zwei Regionen mit so genannten Crack-Edges darstellen.

Definition 3.2.5 (Crack-Edges)[10] Crack-Edges werden zwischen den Pixeln eines Bildes markiert. Dafür wird zwischen je zwei Zeilen beziehungsweise Spalten eine weitere eingefügt, in der dann die Crack-Edges repräsentiert werden. Das Bild erhält also eine Größe von $(2 \cdot Breite - 1) \times (2 \cdot Höhe - 1)$.

Dieser Sachverhalt wird durch die Abbildung 7 verdeutlicht. Ausgehend von dem Originalbild werden Crack-Pixel eingefügt, so dass das Bild auf die oben genannte Größe anwächst. Anschließend werden erst in eine Richtung gehend alle Crack-Pixel als Kante markiert, die an zwei verschiedene Regionen grenzen. Die Crack-Pixel, die nur an eine Region grenzen, werden dieser zugeordnet. Im zweiten Durchgang wird die andere Richtung in gleicher Weise abgelaufen. Dabei ist zu beachten, dass Crack-Pixel, die schon als Kante markiert wurden, zu keiner der ursprünglichen Regionen gehören.

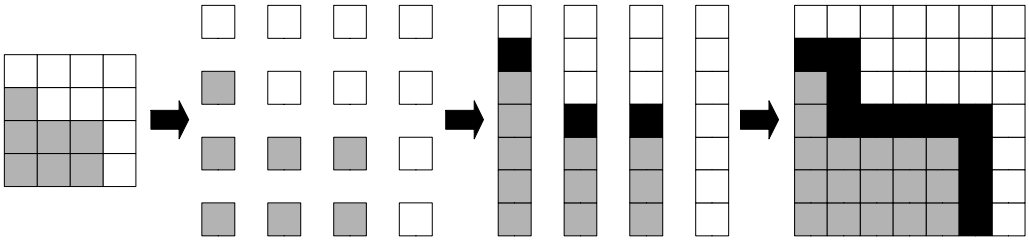


Abbildung 7: Erstellung eines Crack-Edge-Bildes

Nach Anwendung dieses Verfahrens sind alle Regionen eindeutig voneinander getrennt. Außerdem hat man den Vorteil, dass sowohl Regionen als auch Kanten bezüglich der 4er-Nachbarschaft verbunden sind.

3.2.2 Volumennachbarschaften

Ähnlich wie bei den Nachbarschaften in der Ebene gibt es auch unterschiedliche dreidimensionale Nachbarschaften. Dabei kann man die zweidimensionalen Nachbarschaften relativ einfach um eine Dimension ausweiten und gelangt zur 6er-Nachbarschaft beziehungsweise zur 26er-Nachbarschaft. Hinzu kommt noch die 18er-Nachbarschaft. Diese Nachbarschaften ergeben sich je nachdem, ob diagonal zueinander liegende Voxel als benachbart betrachtet werden oder nicht.

Definition 3.2.6 (6er-Nachbarschaft)[11] Zwei Voxel V_1 und V_2 heißen benachbart im Sinne der 6er-Nachbarschaft, falls sich genau eine Koordinate von V_1 von der korrespondierenden Koordinate von V_2 um genau eins unterscheidet. Formal ausgedrückt ergibt sich für diese beiden Voxel mit den Koordinaten (x_1, y_1, z_1) beziehungsweise (x_2, y_2, z_2) folgende Bedingung:

ist $|x1 - x2| + |y1 - y2| + |z1 - z2| = 1$, dann sind V_1 und V_2 benachbart.

10 siehe [Vig06a]
11 vgl. [NP01] S. 66

Definition 3.2.7 (18er-Nachbarschaft)[8] Zwei Voxel V_1 und V_2 heißen benachbart im Sinne der 18er-Nachbarschaft, falls sich eine oder zwei Koordinaten von V_1 von der korrespondierenden Koordinaten von V_2 um höchstens eins unterscheidet. Formal ausgedrückt ergibt sich für diese beiden Voxel mit den Koordinaten (x_1, y_1, z_1) beziehungsweise (x_2, y_2, z_2) folgende Bedingung:

ist $1 \le |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2| \le 2 \ \wedge \ max(|x_1 - x_2|, |y_1 - y_2|, |z_1 - z_2|) = 1$, dann sind V_1 und V_2 benachbart.

Abschließend ist zu sagen, dass auf eine Umsetzung der 18er-Nachbarschaft verzichtet wurde, da sie in der Praxis kaum Anwendung findet. Sie wird im Verlauf dieser Arbeit nicht weiter erwähnt.

Definition 3.2.8 (26er-Nachbarschaft)[8] Zwei Voxel V_1 und V_2 heißen benachbart im Sinne der 26er-Nachbarschaft, falls V_1 ungleich V_2 ist und sich jede Koordinate von V_1 von der korrespondierenden Koordinate von V_2 um höchstens eins unterscheidet. Formal ausgedrückt ergibt sich für diese beiden Voxel mit den Koordinaten (x_1, y_1, z_1) beziehungsweise (x_2, y_2, z_2) folgende Bedingung:

ist $max(|x_1 - x_2|, |y_1 - y_2|, |z_1 - z_2|) = 1$, dann sind V_1 und V_2 benachbart.

Diese Nachbarschaften lassen sich veranschaulichen, indem man sich die Voxel als Quader vorstellt. Zwei Voxel sind 6-benachbart, wenn diese Quader eine gemeinsame Fläche haben. Die Voxel, die nur über eine gemeinsame Kante verfügen, heißen erst ab der 18er-Nachbarschaft benachbart. 26-benachbart sind zudem noch die Voxel, welche eine gemeinsame Ecke haben. Diesen Sachverhalt stellt Abbildung 8 dar.

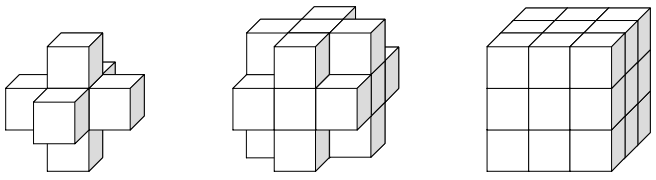


Abbildung 8: Abgebildet sind die drei Voxelnachbarschaften. Von links nach rechts: 6er-, 18er- und 26er-Nachbarschaft. (aus [Jäh97] S. 34)

Die Definitionen von Pfaden und Verbundenheit ergeben sich analog zu den Definitionen aus 3.2.1. Man ersetze entsprechend Pixel durch Voxel. Auch die Crack-Edges ergeben sich analog. Hier ist nur die zusätzliche Dimension zu beachten, was in einem dritten Durchgang geschieht. Es werden nun diejenigen Voxel als Kanten-Voxel markiert, welche an zwei unterschiedliche Regionen grenzen. Das Volumen muss für diesen Vorgang auf eine Größe von $(2 \cdot Breite - 1) \times (2 \cdot Höhe - 1) \times (2 \cdot Tiefe - 1)$ erweitert werden, was zu einem sehr viel höheren Speicheraufwand führt.

3.3 Topologie von Zellkomplexen

Im Rahmen dieser Arbeit wird sehr oft mit dem Begriff der Zellen und der Zellkomplexe gearbeitet. Sie stellen eine Möglichkeit dar, die Bildanalyse zu formalisieren und wurden von Kovalevsky 1989 eingeführt (vgl. [Kov89]). Es wird zunächst der Begriff der Zellen erklärt, bevor die darauf aufbauenden Zellkomplexe definiert werden. Anschließend werden einige wichtige Eigenschaften von Zellkomplexen beschrieben, die Aussagen darüber treffen können, wie mächtig die verschiedenen, in Kapitel 3.4 beschriebenen, kombinatorischen Karten in Bezug auf ihre topologische Repräsentation sind.

Definition 3.3.1 (*i*-Zelle)[12] Eine 0-Zelle von $\mathbb{R}^d$, auch Knoten genannt, ist ein Punkt in $\mathbb{R}^d$. Eine 1-Zelle aus $\mathbb{R}^d$, auch Kante genannt, ist ein ungeordnetes Paar von unterschiedlichen Knoten in $\mathbb{R}^d$. Eine *i*-Zelle C mit $i>1$ und $i\in\mathbb{N}$ ist rekursiv definiert als eine Menge von (*i-1*)-Zellen, so dass:

1. Jeder Knoten der zu C gehört, zu i unterschiedlichen (*i-1*)-Zellen gehört.
2. Die Schnittmenge zweier (*i-1*)-Zellen entweder leer oder eine (*i-2*)-Zelle ist.

Eine 2-Zelle ist somit ein geschlossener Polygonzug in $\mathbb{R}^d$. Siehe hierzu die Abbildung 9, welche Beispiele für verschiedene *i*-Zellen zeigt.

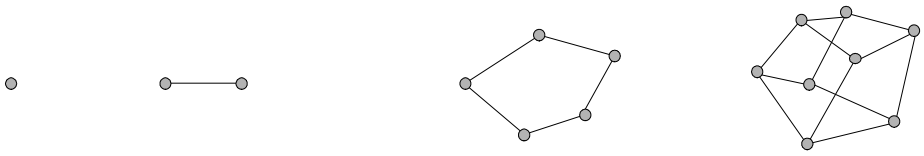


Abbildung 9: Einige i-Zellen im Überblick (von links nach rechts: 0-Zelle, 1-Zelle, 2-Zelle und 3-Zelle)

Definition 3.3.2 (Zellkomplex)[13] Ein Zellkomplex $K=(C, B, dim)$ ist ein Tupel von einer Menge C von *i*-Zellen, einer antisymmetrischen, irreflexiven und transitiven binären Angrenzrelation $B\subset C\times C$ und einer Funktion $dim: C\rightarrow\mathbb{N}$, so dass

$$\forall_{(c',c'')\in B} dim(c')<dim(c'') .$$

Ein *i*-Zellkomplex ist ein Zellkomplex von Zellen, für die gilt:

$$\forall_{c\in C} dim(c)\leq i \ \wedge \ \exists_{c_0\in C} dim(c_0)=i .$$

Nach dieser kurzen Einleitung in Zellen und Zellkomplexe werden die nun aufbauenden Definitionen beschreiben, welche topologischen Eigenschaften beide besitzen. Anhand von Abbildungen werden diese zusätzlich erläutert.

12 siehe [SM01]
13 siehe [Kov99]

Definition 3.3.3 (Regulärer Zellkomplex)[14] Ein regulärer *i*-Zellkomplex ist ein *i*-Zellkomplex, in dem alle Zellen Teil einer *i*-Zelle sind.

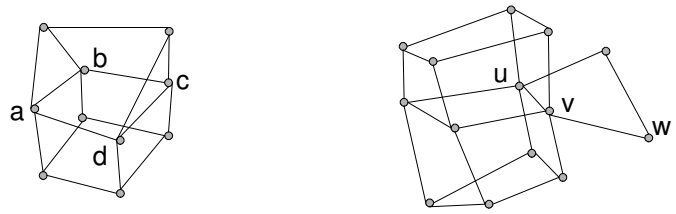


Abbildung 10: Der linke 3-Zellkomplex ist ein regulärer Zellkomplex, der rechte nicht, da die 0-Zelle w nicht Teil einer 3-Zelle ist.

Die Unterscheidung zwischen *i*-Zellkomplexen und regulären *i*-Zellkomplexen ist wichtig, wenn es darum geht, bestimmte Formen der Mannigfaltigkeit zu erklären. Mannigfaltigkeiten sind topologische Eigenschaften, die wichtig sind, um die Repräsentationsfähigkeit verschiedener topologischer Modelle zu beschreiben. So erlauben einige Modelle nur die Darstellung mannigfaltiger Topologien, während andere es erlauben pseudo- oder quasi-mannigfaltige Toplogien zu beschreiben.

3.3.1 Mannigfaltigkeiten von Zellkomplexen

Da die unterschiedlichen Modelle die Verarbeitung verschiedener Arten mannigfaltiger Zellkomplexe erlauben, werden in diesem Absatz die benötigten Mannigfaltigkeiten definiert. Es wurde hierfür auf die Definitionen von De Floriani et. al. und Cohn zurückgegriffen.

Definition 3.3.4 (Mannigfaltigkeit)[11] Sei *C* eine *(i-1)*-Zelle in einem *i*-Zellkomplex *K*.

C heißt mannigfaltig, wenn höchstens zwei *i*-Zellen aus *K* an *C* inzident sind.

In Abbildung 10 ist die 2-Zelle [a,b,c,d] mannigfaltig, da zu ihr nur zwei 3-Zellen inzident sind. Ein negatives Beispiel ist die 1-Zelle [u,v], die nicht mannigfaltig ist. Zu ihr sind vier 2-Zellen inzident.

Definition 3.3.5 (Pseudo-Mannigfaltigkeit)[11] Ein regulärer *i*-Zellkomplex, der nur aus mannigfaltigen *i*-Zellen besteht, heißt pseudo-mannigfaltig.

Satz 3.3.1 (Reguläre adjazente Zellkomplexe)[11] Reguläre adjazente *i*-Zellkomplexe werden induktiv beschrieben. Ein regulärer 1-Zellkomplex ist regulär adjazent. Ein regulärer *i*-Zellkomplex ist genau dann regulär adjazent, wenn der Link[15] eines jeden Knotens ein verbundener regulärer adjazenter *(i-1)*-Zellkomplex ist.

Definition 3.3.6 (Quasi-Mannigfaltigkeit)[11] Ein *i*-Zellkomplex ist geanau dann quasi-mannigfaltig, wenn er sowohl pseudo-mannigfaltig als auch ein regulärer adjazenter Zellkomplex ist.

14 siehe [FMMP02]

15 Ein Link eines Knotens *p* beschreibt die Teilmenge von *j*-Zellen ($j \in \{0,\ldots,i-1\}$), die nur die adjazenten Knoten von *p* enthalten.

Eine weitere Definition der Quasi-Mannigfaltigkeit findet sich in [Coh95] auf Seite 5. Sie besagt:

Ein *i*-Zellkomplex heißt quasi-mannigfaltig, wenn er nicht in zwei *i*-Zellkomplexe zerlegt werden kann, die nur durch eine Zelle der Dimension *i-2* oder weniger verbunden sind.

Nach diesen Definitionen wird nun eine interessante topologische Eigenschaft der Mannigfaltigkeiten beschrieben, die Orientierbarkeit. Sie wird im Rahmen dieser Arbeit ebenfalls dazu verwendet, die Ausdrucksstärke von topologischen Repräsentationen zu beschreiben. So lassen manche Modelle es zu, nicht orientierbare Mannigfaltigkeiten zu repräsentieren, während andere dies nicht ermöglichen.

Definition 3.3.7 (Orientierbarkeit) Eine Mannigfaltigkeit ist orientierbar, sofern sich Innen- und Außenfläche unterscheiden lassen.

Beispiele für geschlossene 2-Mannigfaltigkeiten sind: Kugel, Torus, Brezelfläche oder allgemein Kugeln mit *n* „Henkeln“. Nicht orientierbare 2-Mannigfaltigkeiten sind zum Beispiel das Möbiusband und die Klein'sche Flasche, allgemein sind es Kugeln mit *n* Kreuzhauben.

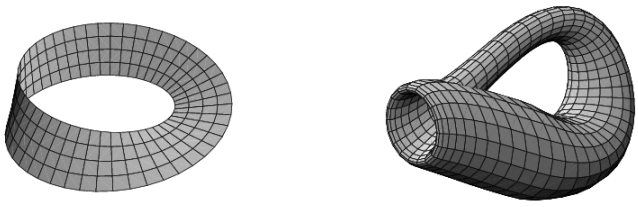


Abbildung 11: Beispiele für nicht orientierbare 2-Mannigfaltigkeiten. Links: Das Möbius Band (aus [Wik06a]); Rechts: Die Klein'sche Flasche (aus [Wik06b])

3.3.2 Charakteristik von Zellkomplexen

Eine Kategorisierung von Zellkomplexen kann anhand topologischer Invarianten erfolgen. Für mannigfaltige topologische Räume stellt die Euler-Charakteristik eine gebräuchliche Invariante dar. Unter einer topologischen Invariante ist dabei eine Abbildung zu verstehen, die topologisch gleichen Räumen einen gleichen Funktionswert zuweist. Diese Abbildung muss nicht vollständig sein.

Definition 3.3.8 (Euler-Charakteristik)[16] Sei *K* ein *i*-Zellkomplex. Die Euler-Charakteristik $\chi(K)$ ist:

$$\chi(K)=\sum_{j=0}^{i}(-1)^{j}|c_j|=\sum_{\sigma\in K\setminus\{\emptyset\}}(-1)^{dim\,\sigma},$$

wobei c_j eine in *K* enthaltene *j*-Zelle ist.

16 siehe [Zom05] S. 61

Für jeden Zellkomplex gibt es jeweils nur einen Wert für die Euler-Charakteristik, unabhängig von der gewählten Triangulierung dessen.

Die Oberflächentopologie von 2-Mannigfaltigkeiten kann mit der folgenden vereinfachten Formel berechnet werden:

$$\chi = V - E + F ,$$

dabei ist *V* die Anzahl der Knoten, *E* ist die Anzahl der Kanten, und *F* ist die Anzahl der Flächen, aus denen die Oberfläche besteht.

Betrachtet man einen Würfel, so ergibt sich $\chi = 2$, da dieser acht Knoten, zwölf Kanten und sechs Flächen besitzt. Auch für einen Tetraeder ergibt sich $\chi = 2$, da dieser aus vier Knoten, sechs Kanten und vier Flächen aufgebaut ist. Generell gilt für jeden Polyeder, der homöomorph zu einer Kugel ist, dass die Euler-Charakteristik zwei beträgt.

Gemeinsam haben alle Polyeder zudem, dass sie keine sogenannten „Henkel“ aufweisen. Ein gängiges Beispiel für ein Objekt mit Henkel ist der Torus (vgl. Abbildung 19). Die Euler-Charakteristik für solche Objekte beträgt null.

Von der Euler-Charakteristik lässt sich auf die Anzahl der Henkel eines Objekts schließen. Die Anzahl bestimmt den Genus einer Oberfläche.

Definition 3.3.9 (Genus einer Oberfläche)[17] Eine Oberfläche *S* mit *g* Henkeln, wird als Oberfläche mit Genus *g* bezeichnet.

Ist eine Oberfläche S_0 orientierbar, so gilt $\chi(S_0) = 2 - 2g$. Für eine nicht orientierbare Oberfläche S_1 gilt $\chi(S_1) = 2 - g$. Man erkennt, dass die Klein'sche Flache und ein Torus einen unterschiedlichen Genus besitzen, obwohl sie die gleiche Euler-Charakteristik aufweisen. Beide Objekte sind nicht homöomorph.

Theorem 3.3.1 (Homöomorphie von 2-Mannigfaltigkeiten)[14] Zwei geschlossene, endliche Oberflächen S_0 und S_1 sind genau dann homöomorph, wenn gilt:

1. $\chi(S_0) = \chi(S_1)$ und
2. entweder sind beide Oberflächen orientierbar oder beide nicht orientierbar.

Mit Hilfe dieses Theorems ist es sehr einfach, Objekte als homöomorph zu klassifizieren oder Objekte in topologische Kategorien einzuordnen.

17 siehe [Zom05] S. 64

3.4 Kombinatorische Karten

Bei der Bildsegmentierung (siehe Kapitel 5) geht es darum, Bilder nach bestimmten Kriterien in verschiedene Regionen zu unterteilen, welche im besten Falle die Objekte oder bedeutungstragenden Einheiten im Bild sind. Die kombinatorischen Karten stellen eine Möglichkeit dar, diese segmentierten Daten auf eine andere Art zu repräsentieren, als nur gleiche Bildpunkte mit dem gleichen Farbwert zu versehen und zusätzlich ihren Zusammenhang zu beschreiben. Sie bieten sich ebenfalls an, wenn es darum geht, sowohl Topologie als auch Geometrie der Regionen des Bildes zu repräsentieren (vgl. [Mei03] und [BDDV03]).

Wie bereits in Kapitel 3.1 erwähnt, geht die hier beschriebene Repräsentation weit über die der ikonischen Regionenrepräsentation hinaus. Die an sie gestellten Aufgaben lassen sich wie folgt zusammenfassen[18]:

1. Sie soll die topologischen und geometrischen Informationen des segmentierten Bildes darstellen können. Dazu zählen die Definitionen von Konturen, Nachbarn, Oberflächenformen und andere.
2. Die Möglichkeit der Extraktion von Merkmalen, die am Segmentierungsprozess beteiligt sind, sollte gegeben sein. Dazu gehören Kontur- und Volumenrekonstruktion, Entscheidbarkeit der Benachbartheit (z.B. benachbarte oder eingeschlossene Volumen, gemeinsame Konturen) und anderes.
3. Zudem besteht die Notwendigkeit der konsistenten Aufrechterhaltung der kombinatorischen Karte nach der Anwendung von Operatoren auf die Bilddaten. Beispiele dafür sind das Zerteilen von Volumenregionen in kleinere Regionen oder auch das Verschmelzen von verschiedenen Volumenregionen zu einer.

Kombinatorische Karten sind mathematische Repräsentationsmodelle, die Raumaufteilungen in allen Dimensionen ermöglichen. Sie wurden in den 1960ern von Cori als Modell der Repräsentation planarer Graphen eingeführt und in seiner Doktorarbeit 1975 der Öffentlichkeit vorgestellt [Cori75]. Eine *n*-dimensionale kombinatorische Karte erlaubt es, Raumaufteilungen orientierbarer *n*-Mannigfaltigkeiten darzustellen.

Zusätzlich zu den Raumaufteilungen gibt es noch die Inzidenzrelationen, die die kombinatorischen Karten ebenfalls zur Verfügung stellen. Um alle diese Informationen bereitstellen zu können, werden sie aus abstrakten Elementen – den Darts – erstellt, auf denen Anwendungen definiert werden.[19]

Definition 3.4.1 (Kombinatorische Karte)[16] Sei $n \geq 0$. Eine *n*-kombinatorische Karte ist ein Tupel $M = (B, \alpha_1, \ldots, \alpha_n)$, mit:

1. B ist eine endliche Menge von Darts,
2. α_1 ist eine Permutation auf B,
3. $\forall_i 2 \leq i \leq n$, α_i ist eine Involution auf B,

18 vgl. [BDDV03]
19 siehe [DR02] S. 221

4. $\forall_i 1 \le i \le n-2, \forall_j i+2 \le j \le n$, $\alpha_i \circ \alpha_j$ ist eine Involution.

Für jede Dimension gibt es eine Applikation α_i, welche zwei *i*-Zellen in Relation setzt. Sind zwei Darts durch ein α_i verbunden, so heißen sie α_i-vernäht (in der englischsprachigen Literatur als α_i-sewed bezeichnet). Jede Raumzelle wird implizit durch eine Menge von Darts repräsentiert, das heißt man kann beispielsweise eine Fläche durch die Darts beschreiben, die diese Fläche begrenzen.

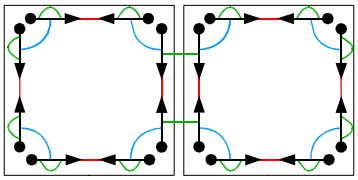


Abbildung 12: Zwei Flächen, zusammengesetzt aus Darts, am Beispiel einer G-Map

Eine besondere Form der kombinatorischen Karten ist die generalisierte Karte („generalized map“) oder kurz G-Map.

An dieser Stelle wird noch die Definition des Homöomorphismus angegeben, damit im weiteren Verlauf dessen Bedeutung geklärt ist.

Definition 3.4.2 (Homöomorphismus)[20] Ein Homöomorphismus ist eine Äquivalenzrelation und eine bijektive Abbildung zwischen zwei Punkten in geometrischen Figuren oder topologischen Räumen, welche kontinuierlich in beide Richtungen ist.

Zur Einleitung wird zunächst die graphentheoretische Grundlage der kombinatorischen Karten nach Kropatsch (siehe [BK99]) beschrieben. Dies geschieht zur besseren Übersichtlichkeit anhand zweidimensionaler kombinatorischer Karten. Es sei bereits an dieser Stelle darauf hingewiesen, dass die kombinatorischen Karten nicht der Definition 3.4.1 entsprechen, aber dennoch die gleiche Ausdrucksstärke besitzen. Sie sind in diesem Sinne sogar äquivalente Repräsentationen. Der genaue Zusammenhang wird im folgenden Abschnitt erläutert, bevor in den weiteren Abschnitten verschiedene Modelle dreidimensionaler kombinatorischer Karten diskutiert werden. Dabei werden zunächst zwei unterschiedliche Modelle kombinatorischer Karten behandelt, bevor zu einer besonderen Karte, der generalisierten Karte oder auch G-Map, übergegangen wird. Im Anschluss daran werden die Modelle miteinander verglichen, ehe auf die in dieser Arbeit entworfene XG-Map eingegangen wird. Diese bildet zugleich den Abschluss des Kapitels.

20 vgl. [Mat06]

3.4.1 Einführung in die kombinatorischen Karten

Bevor in den nächsten Abschnitten weiter auf die kombinatorischen Karten sowie ihre geometrischen Einbettungen und Anwendungen eingegangen wird, soll in diesem Abschnitt zunächst die graphentheoretische Herkunft und Entwicklung der kombinatorischen Karten veranschaulicht werden. Des Weiteren sollen Problemstellungen erläutert werden, die in den vorgestellten Modellen der nächsten Abschnitte durch unterschiedliche Methoden gelöst werden.

Bezieht man sich auf Kropatsch et al. [BK99], so finden die topologischen Karten ihren Ursprung in der Graphentheorie. Topologische Karten stellen daher in diesem Zusammenhang einen allgemeineren Ansatz gegenüber den kombinatorischen Karten dar. Aus der allgemeinen topologischen Karte lässt sich durch eine einfache Vorgehensweise eine kombinatorische Karte ableiten.

Eine topologische Karte im zweidimensionalen euklidischen Raum $\mathbb{R}^2$ kann durch einen planaren Graphen (siehe Definition 2.1.12) repräsentiert werden. Ein planarer Graph besteht aus endlichen Mengen von Knoten und Kanten. In dieser Form gibt es keine Informationen über die Orientierung der Kanten. Es ist nur bekannt, welche Kanten in welchen Knoten enden. Wenn die Orientierbarkeit des Raumes repräsentiert werden soll, kann eine planare Karte effizient durch eine kombinatorische Karte beschrieben werden (siehe hierzu das Beispiel in Abbildung 13). Zu diesem Zweck werden die Kanten des planaren Graphen an den Schnittpunkten mit ihren dualen Kanten aufgespalten.[21] Dieser Vorgang zerlegt den Graphen in verbundene Teile von Halbkanten, die an Knoten zusammenhängen. Diese Halbkanten stellen die Darts dar. Sie haben ihren Ursprung in dem Knoten zu dem sie inzident sind. Zwei Halbkanten, die aus einer Kante des Graphen hervorgegangen sind, werden durch eine Permutation α miteinander verbunden. Eine zweite Permutation σ definiert die lokale Anordnung von Darts an einem Knoten. Dabei wird eine dem Uhrzeigersinn entgegengesetzte Ordnung der Darts angenommen.

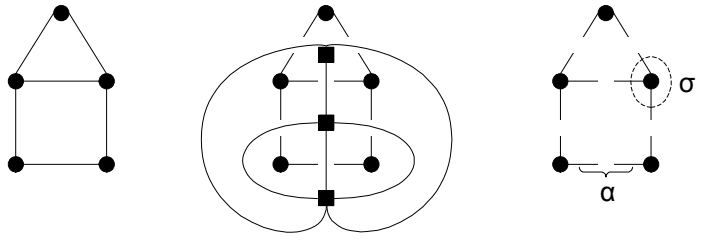


Abbildung 13: Beispiel für die Erstellung einer kombinatorischen Karte (rechts) aus einem planaren Graphen (links). Mitte: Auftrennung der Graphenkanten durch die dualen Kanten.

Die kombinatorische Karte, die durch diese Vorgehensweise entsteht, unterscheidet sich von der Definition 3.4.1 in der Repräsentation der Karte. Die nächste Definition stellt daher eine kombinatorische Karte in diesem neuen Zusammenhang dar.

21 Da dies nur eine Einführung darstellt, wird die genaue Struktur dieses dualen Kantengraphen nicht erläutert, zum Verständnis genügen die folgenden Abbildungen dieses Unterkapitels.

Definition 3.4.3 (Kombinatorische Karte nach Kropatsch) Eine kombinatorische Karte nach Kropatsch [BK99] ist eine Tripel $M=(B,\alpha,\sigma)$. Dabei ist B eine Menge von Darts sowie α und σ Permutationen definiert über *B*. Zudem sei $\forall_{d\in B}\alpha^2(d)=d$, so dass α eine Involution ist.

Durch Verkettungen der hier beschriebenen Orbits kann man zur Beschreibung der allgemeinen 2-kombinatorischen Karte gelangen. Die Orbits aus Definition 3.4.1 lassen sich wie folgt darstellen:

$$\alpha_1=\sigma\circ\alpha$$

$$\alpha_2=\alpha\,.$$

Man erkennt, dass sich beide Repräsentationen nicht in der Aussagekraft unterscheiden, sondern nur in der funktionalen Definition der Orbits. Kropatsch definiert die allgemeine kombinatorische Karte als eine duale Karte zu seiner kombinatorischen Karte (siehe [BK00], S. 12). Die Analogie des α_1-Orbits der allgemeinen kombinatorischen Karte zu dem $\sigma\circ\alpha$-Orbit der Kropatsch'schen kombinatorischen Karte wird in Abbildung 14 dargestellt.

In diesem Beispiel ist leicht zu erkennen, dass die Kante genau dann gewechselt wird, wenn im Kropatsch'schen Fall die $\sigma\circ\alpha$-Verkettung angewendet wird beziehungsweise im allgemeinen Fall die α_1-Permutation.

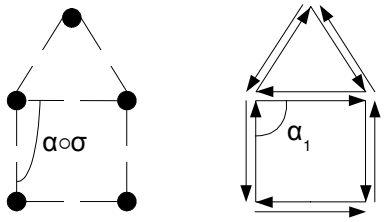


Abbildung 14: Analogie der Orbits zum Wechseln der Kante. Links: Kropatsch'sche kombinatorische Karte; rechts: allgemeine kombinatorische Karte.

Ein Beispiel für die Analogie der Orbits zum Wechseln einer Fläche zeigt die Abbildung 15. Wechselt man in der Kropatsch'schen Repräsentation das α-Orbit, so findet ein Wechsel zwischen zwei Flächen statt. Selbiges erfolgt in der allgemeinen Repräsentation durch die Anwendung des α_2-Orbits. Dies fällt in der Darstellungsform der Karten durch Halbkanten (vgl. Abbildung 13) zunächst nicht auf, wird allerdings klarer, wenn man die analoge Darstellung der allgemeinen kombinatorischen Karte heranzieht. Der Dart an dem begonnen wird, ist in Abbildung 15 jeweils durch eine breitere Kante beziehungsweise Halbkante hervorgehoben.

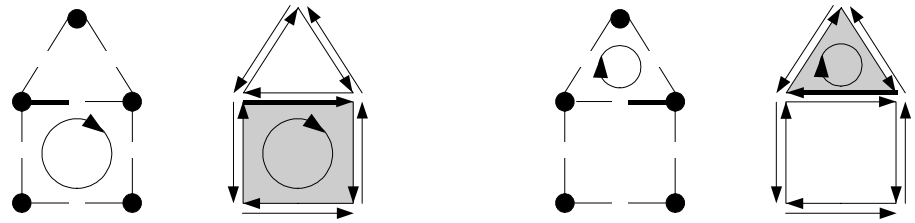


Abbildung 15: Analogie des α-Orbits der Kropatsch'schen Karte und des α_2-Orbits der allgemeinen kombinatorischen Karte.

Stellte das obige Beispiel den Fall eines recht einfachen planaren Graphen dar, so gibt es dennoch einige Besonderheiten bei der Erstellung einer kombinatorischen Karte nach Kropatsch, falls der Graph eine Struktur wie links in Abbildung 16 besitzt. Dieser Graph beschreibt ein Bild einer Region mit einem Loch in der Mitte. In der Repräsentation dessen, mittels eines planaren Graphen, muss deshalb der planare Graph der Fläche eine Verbindung zu seiner inneren Kontur (dem Loch) besitzen. Diese ist im Beispiel durch die Kante (e_0, e_1) gegeben. Sie stellt die Benachbartheit der Region mit dem enthaltenen Loch dar. Doch leider führt dies im weiteren Erstellungsprozess der kombinatorischen Karte zu einer Auftrennung der äußeren Region entlang der Kante (e_0, e_1), was in Abbildung 16 rechts deutlich wird. Dies sorgt zwar einerseits dafür, dass keine Information über den Flächeneinschluss verloren geht, da jede (äußere) Region homöomorph zu einer Scheibe bleibt, andererseits entstehen Scheinkanten im Graphen, die willkürlich gewählt werden können, und im Bild nicht vorkommen. Deshalb müssen diese Scheinkanten, auch Brücken oder Bridges genannt, im Weiteren, bei dieser Art der Repräsentation, gesondert behandelt werden.

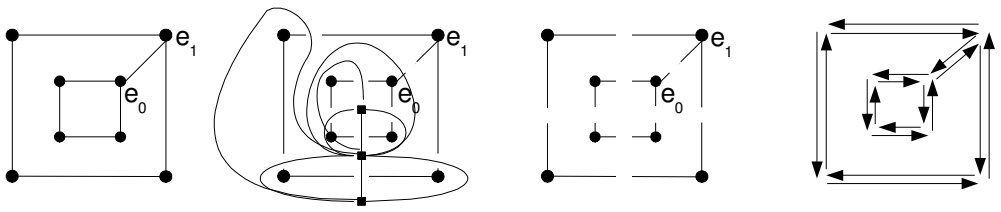


Abbildung 16: Beispiel eines planaren Graphen einer Fläche mit einem Loch (Einschluss) und seine Erweiterung zur kombinatorischen Karte nach Kropatsch beziehungsweise zur allgemeinen kombinatorischen Karte. Von links nach rechts: Der planare Graph des Bildes, der duale Kantengraph, die kombinatorische Karte nach Kropatsch und die allgemeine kombinatorische Karte.

Dies stellt einen wesentlichen Kritikpunkt dieser Repräsentationsform dar, denn Kanten in der kombinatorischen Karte sollten auch immer zu Kanten des Bildes gehören. Kropatsch definiert für diese Brücken zwar besondere Eigenschaften und beweist, dass sie lediglich bei Flächeneinschlüssen vorkommen können (vgl. [BK00]). Wünschenswert wäre allerdings eine generelle Kapselung dieser Brücken von den Kanten. Diesen Ansatz

verfolgt Köthe in der Repräsentationsform XPMap (siehe [Köt00] und [Köt01] beziehungsweise später in Abschnitt 3.3.5). Dort werden Einschluss-Relationen definiert, die unabhängig von der kombinatorischen Karte sind. Somit sind Brücken in dieser Repräsentation echte Kanten des Bildes, während Einschlüsse nicht mit Hilfe von Brücken gespeichert werden müssen.

Eine weitere Herangehensweise ist die Einbindung eines Homotopie-Baumes, welcher Informationen über Einschlüsse jedweder Art enthält (siehe zum Beispiel [DBF04]). Leider wird in der Literatur meist nur erwähnt, dass dies möglich sei, eine formale Definition einer Karte, die aus mehreren Teilkarten und einem Homotopie-Baum besteht, findet sich hingegen selten.

Man erkennt, dass bereits beim Arbeiten mit zweidimensionalen Daten Probleme auftreten können, für die Lösungen gefunden werden müssen. Allerdings wurde die kombinatorische Karte am Anfang des Kapitels allgemein für n Dimensionen definiert, und diese Arbeit befasst sich mit 3-kombinatorischen Karten. An dieser Stelle werden deshalb die auftretenden dreidimensionalen Fälle kurz erläutert.

Wie bei den zweidimensionalen Karten gibt es bei den dreidimensionalen keine Probleme, sobald alle Regionen geschlossen nebeneinander liegen. Das heißt, dass es zwischen je zwei Kanten des Graphen einen Weg gibt, ohne dass irgendwelche „Brücken“ oder Ähnliches eingefügt werden müssen. Dieser Sachverhalt ist allerdings in den seltensten Fällen gegeben.

Zuerst ist deshalb der äquivalente Fall zu den oben genannten Flächeneinschlüssen zu betrachten. Dieser hat in der höheren Dimension die Form von eingeschlossenen Volumen. Dabei ist ein Volumen vollständig von einem anderen umgeben, so dass es keine Verbindung zu noch weiteren besitzt. Ein einfaches Beispiel dafür stellt ein eingeschlossenes Insekt in einem Bernstein dar. In Abbildung 17 ist leicht zu erkennen, dass das Volumen des Insekts keine Verbindung nach Außen zur Luft hat, also vollständig vom Stein umschlossen ist. In der Abbildung ist zusätzlich die digitalisierte Repräsentation dieses Volumens zu sehen. Zu diesem Zwecke reicht dafür eine stark vereinfachte Darstellung aus, die von den Details abstrahiert.

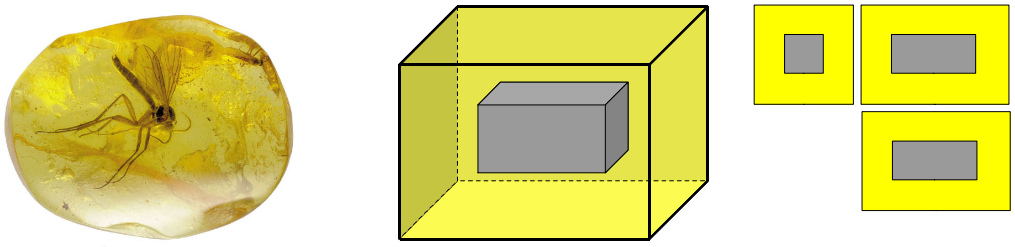


Abbildung 17: Beispiel für einen vollständigen Volumeneinschluss. Von Links nach rechts: Das eingeschlossene Insekt, eine vereinfachte Darstellung und eine Drei-Tafel-Projektion (oben links: YZ-Ebene, oben rechts: XY-Ebene, unten: XZ-Ebene).

Dieser Fall ist allerdings nicht der einzig mögliche. Rückt zum Beispiel das innere Volumen an eine der begrenzenden Flächen des Außenvolumens, so handelt es sich nicht

mehr um ein vollständig eingeschlossenes Volumen. Allerdings gibt es auch bei dieser Konfiguration Kanten zwischen denen kein Weg besteht. Schaut man direkt auf die Fläche, an denen beide Volumen zusammentreffen, so erkennt man, dass ein Flächeneinschluss vorliegt. Diese Ansicht hat Ähnlichkeit zu den Flächeneinschlüssen im zweidimensionalen Raum. Ein Beispiel dafür ist eine Baumhöhle mit einem Zugang. Abbildung 18 zeigt diese und zusätzlich wieder die abstrahierte digitalisierte Form.

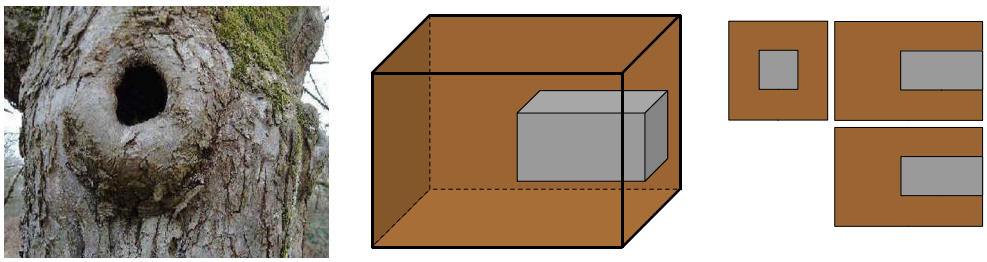


Abbildung 18: Beispiel für einen flächenverbundenen Volumeneinschluss. Von Links nach rechts: Die Baumhöhle, eine vereinfachte Darstellung und eine Drei-Tafel-Projektion (oben links: YZ-Ebene, oben rechts: XY-Ebene, unten: XZ-Ebene).

Des Weiteren ist es möglich, dass ein Volumen nicht nur an einer Stelle an eine begrenzende Fläche des umgebenen Volumens anschließt. So kann es durchaus vorkommen, dass das „innere“ Volumen mehrfach an einer Fläche anschließt oder auch an verschiedenen Flächen gleichzeitig. Denjenigen Kanten, die diese Flächeneinschlüsse darstellen, fehlt eine Verbindung zu den Kanten des „äußeren“ Volumens. Ein Beispiel hierfür stellt ein Tunnel dar, der durch einen Berg führt (siehe Abbildung 19).

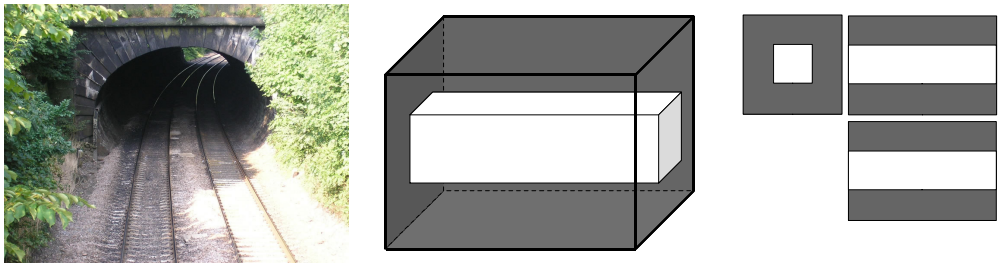


Abbildung 19: Beispiel für eine Volumendurchdringung. Von Links nach rechts: Der Tunnel, eine vereinfachte Darstellung und eine Drei-Tafel-Projektion (oben links: YZ-Ebene, oben rechts: XY-Ebene, unten: XZ-Ebene).

In diesem Unterkapitel wurde der Bogen von der Graphentheorie zur kombinatorischen Karte geschlagen. Dies erfolgte der Übersichtlichkeit halber für zweidimensionale Karten. Es wurde das Modell der Kropatsch'schen kombinatorischen Karte vorgestellt und mit der allgemeinen Definition 3.4.1 verglichen. Im Anschluss daran wurden Probleme aufgezeigt, die entstehen, wenn Flächen beziehungsweise. Volumen von anderen umgeben sind. Im nächsten Abschnitt werden zwei Modelle dreidimensionaler

kombinatorischer Karten vorgestellt, die eine gewisse Analogie, entsprechend der beiden bereits vorgestellten zweidimensionalen Karten, besitzen. Dabei entspricht der erste in Abschnitt 3.4.2 vorgestellte Ansatz eher dem Modell von Kropatsch, wohingegen die zweite beschriebene Repräsentation eher der allgemeinen Definition kombinatorischer Karten entspricht.

3.4.2 Zwei Modelle topologischer Karten und deren geometrische Einbettung

In diesem Abschnitt werden zwei Modelle topologisch-geometrischer Repräsentationen beschrieben. Diese bauen auf unterschiedlichen Repräsentationen kombinatorischer Karten auf, besitzen allerdings zusätzlich noch geometrische Informationen. Doch auch die Einbettung der geometrischen Informationen geschieht in beiden auf unterschiedliche Art. Während bei der ersten vorgestellten Karte diese Informationen global gespeichert werden, wird bei der zweiten die Geometrie hierarchisch eingebettet. Auf die Vor- und Nachteile der einzelnen Repräsentationen wird in Abschnitt 3.4.4 eingegangen werden.

Das Modell der GE topologischen Karten[22]

Die GE topologischen Karten verwenden eine globale geometrische Einbettung. Der Name ergibt sich aus dem Englischen für „topological map with **g**lobal **e**mbedding“. Das Verfahren basiert auf einer globalen Kodierung der Kanten eines segmentierten Bildes und auf der Definition der geometrischen Analogie von topologischen Darts. Die folgende Definition erzielt die Korrespondenz zwischen der topologischen Repräsentation von segmentierten Bildern und der geometrischen Beschreibung von Regionen und Regionengrenzen.

Definition 3.4.4 (GE topologische Karte) Sei *I* ein segmentiertes Volumen der Größe $\Delta_x \times \Delta_y \times \Delta_z$, welches in der Form einer ikonischen Regionenrepräsentation vorliegt. Die *i*-Zellen von *I* werden als ein dreidimensionales Array B_I kodiert, welches eine Größe von $(\Delta_x+1)\times(\Delta_y+1)\times(\Delta_z+1)$ besitzt. Der Wert von $B_I(i,j,k)$ ist ein Tupel von drei Wahrheitswerten, wovon jeder zu einer der drei Oberflächen des Voxels (i,j,k) korrespondiert, welche adjazent zum Voxelknoten *P* mit den Koordinaten $(i-\frac{1}{2}, j-\frac{1}{2}, k-\frac{1}{2})$ liegen. Das Array beschreibt das Begrenzungsvolumen von *I*.

Damit das Begrenzungsvolumen einer topologischen Karte entspricht, muss es in seine einfach verbundenen Oberflächenteile zerlegt werden (Mengen von *i*-Zellen, die homöomorph zu topologischen Scheiben sind). Über diesen Weg ist es möglich, jedes Oberflächenteil mit einer Fläche einer topologischen Karte zu assoziieren. Um die Begrenzungen in einfach verbundene Teile zu zerlegen, wird die Oberfläche durch Ziehen von geschlossenen Pfaden von adjazenten 1-Zellen entlang der Grenze getrennt. Jedes Zerlegen wird kodiert durch das Markieren der zugehörigen 1-Zellen.

22 Die Beschreibung des Modells stammt aus [BDDV03].

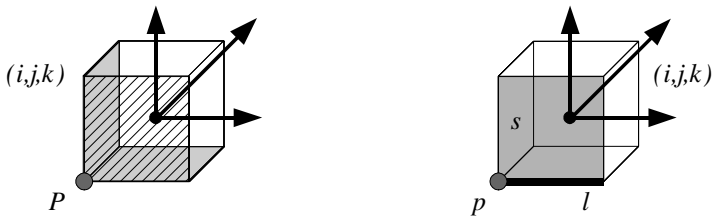


Abbildung 20: Links: Veranschaulichung der Adjazenz von Flächen zum Knoten P. Rechts: Beispiel eines geometrischen Darts (p,l,s), der aus einem Knoten p, einer Kante l und einer Fläche s besteht.

Die Zerlegung der Grenze in einzelne Teile benötigt zwei unterschiedliche Trennungsmöglichkeiten. Die erste Möglichkeit ist das Trennen zwischen den Regionen. Es besteht aus dem Zerlegen der Grenze entlang der Kante einer Oberfläche zwischen zwei Volumenregionen. Die 1-Zellen, welche auf diese Art zu markieren sind, sind leicht zu finden, denn es sind die, welche adjazent zu mehr als zwei *i*-Zellen sind. Die zweite Möglichkeit ist das Trennen innerhalb einer Volumenregion. Es wird benötigt, um die Grenzkomponenten, die aus der ersten Trennung resultieren, abzubauen. Ein Grenzelement muss einmal zerlegt werden, wenn es homöomorph zu einem Kreis ist. Es muss zweimal zerlegt werden, wenn es homöomorph zu einem Ein-Loch-Torus ist, und so weiter. Diese Zerlegung basiert auf einem topologischen Regionenwachstum („region growing"). Allerdings stellt der Algorithmus, der dieses Regionenwachstum ausführt, spezielle Anforderungen an das segmentierte Bild, so dass es nicht auf beliebige Bilder anwendbar ist.

Das Resultat dieser Zerlegungen ist eine Menge von 1-Zellen, die die Begrenzungen der Objekte im segmentierten Bild darstellen. Diese 1-Zellen werden in diesem Zusammenhang *l*-Zellen genannt. Jede verbundene Komponente dieser *l*-Zellen stellt einen Graphen dar. Die Knoten dieser Graphen werden *p*-Zellen genannt.

Die topologische Karte wird nun aus dem Begrenzungsvolumen erstellt, indem die Grenze abgelaufen wird und dabei jedes Oberflächenteil mit einer Fläche einer Oberflächenkarte assoziiert wird. Da die topologische Kodierung auf topologischen Darts aufbaut, ist es notwendig, die geometrische Entsprechung zu einem Dart zu definieren. Ein geometrischer Dart ist ein Tupel *(p,l,i)*, dabei ist *p* eine *p*-Zelle, *l* eine *l*-Zelle, die adjazent zu *p* ist, und *i* ist eine *i*-Zelle, die adjazent zu *l* ist (vgl. mit rechtem Bild von Abbildung 20).

Eine dreidimensionale kombinatorische Karte im GE-Modell wird als Tupel $M_{\mathrm{GE}}=(D,\gamma,\sigma,\alpha)$ definiert. Die Permutation γ vernäht zwei Darts, die zu adjazenten Flächen um einen selben Knoten gehören. Die Involution σ vernäht zwei Darts, die mit dem selben Knoten verbunden sind und zur selben Fläche gehören. Die Involution α vernäht zwei entgegengesetzte Darts der selben Kante, die zur selben Fläche gehören.

Ein Beispiel für eine solche Karte zeigt das mittlere Bild in Abbildung 21. Das linke Bild zeigt die Ausgangsbilddaten.

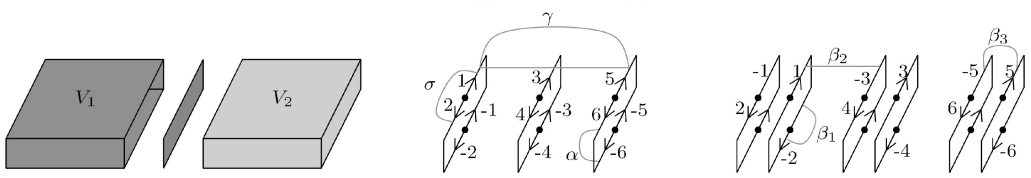


Abbildung 21: Veranschaulichung der GE-Karte und der HLE-Karte; links: Zwei adjazente Volumen, die von der unendlichen Region umgeben sind; mitte: zugehörige Karte nach dem GE-Ansatz; rechts: zugehörige Karte nach dem HLE-Ansatz; Abbildung aus [BDDV03]

Die Konstruktion der Karte geschieht durch das Verfolgen jeder Kette von *l*-Zellen, wobei jeweils zwei *p*-Zellen abhängig von der Orientierung der Oberfläche miteinander verbunden werden. Auf diesem Weg ist es möglich, die beiden geometrischen Darts am Ende der Kette miteinander zu assoziieren. Ist einer dieser geometrischen Darts (in diesem Fall *g*) mit einem topologischen Dart *d* assoziiert, so ist der andere Dart *g'* mit $\alpha(d)$ assoziiert.

Betrachtet man die Abfolge von geometrischen Darts, die man beim Herumdrehen um die *l*-Zelle von *g* erreicht, und die Abfolge der geometrischen Darts beim Herumdrehen im umgekehrten Sinne um die *l*-Zelle von *g'*, so kann man zeigen, dass beide Abfolgen die gleiche Länge haben. Seien $g_1' \ldots g_k'$ und $g_1'' \ldots g_k''$ diese Abfolgen. Es wird jedes g_i' mit einem neuen Dart d_i und g_i'' mit $\alpha(d_i)$ assoziiert. Jeder Ablauf von $d_1 \ldots d_k$ ist ein Kreis von γ. Die Permutation σ kann aus der Nachbarschaft von jedem geometrischen Dart initialisiert werden.

Diese Konstruktion einer kombinatorischen Karte assoziiert jeden geometrischen Dart mit einem topologischen, und umgekehrt assoziiert sie jeden topologischen Dart mit seiner geometrischen Einbettung.

Das Modell der HLE topologischen Karten[23]

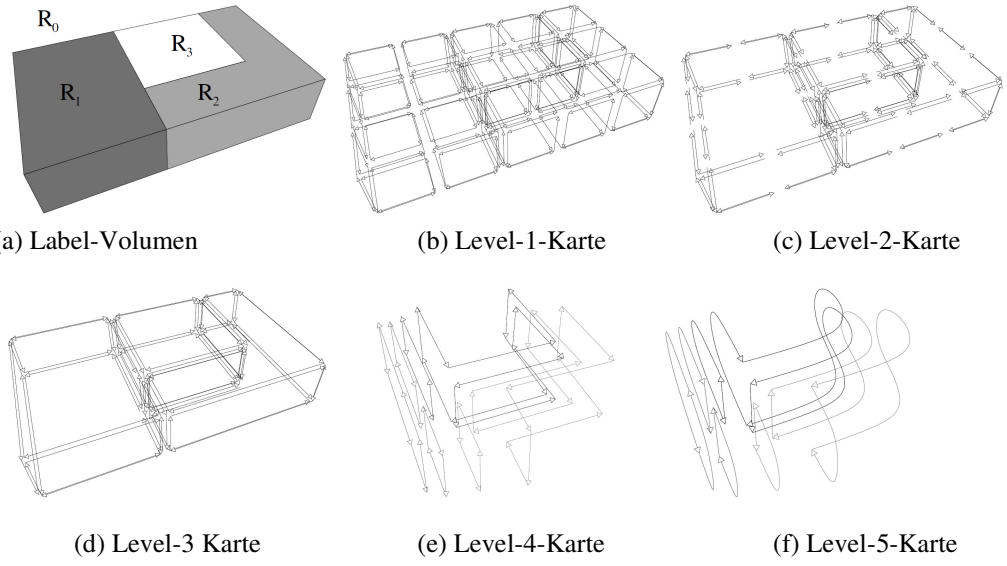
Die HLE topologischen Karten verwenden eine hierarchische lokale geometrische Einbettung. Der Name ergibt sich aus dem Englischen für „topological map with **h**ierarchical **l**ocal **e**mbedding". Das Hauptprinzip der HLE topologischen Karten besteht aus einer schrittweisen Vereinfachung der kombinatorischen Karte, die alle Voxel des Volumens repräsentiert. Im Allgemeinen werden für diese Vereinfachungen Verschmelzungsoperationen angewandt, welche zwei adjazente *i*-Zellen zu einer einzigen verschmelzen.

Eine dreidimensionale kombinatorische Karte im HLE-Modell wird als Tupel $M_{\text{HLE}}=(D, \beta_1, \beta_2, \beta_3)$ definiert. Die Permutation β_1 vernäht zwei Darts, die zu aufeinander folgenden Kanten von einer selben Fläche gehören. Die Involution β_2 vernäht zwei Darts, die zur selben Kante inzident sind und zum selben Volumen gehören. Die Involution β_3 vernäht zwei Darts der selben Kante, die zur selben Fläche inzident sind.

23 Die Beschreibung des Modells stammt aus [BDDV03].

Ein Beispiel einer topologischen Karte nach dem Modell der HLE-Karten zeigt das rechte Bild der Abbildung 21.

Die erste kombinatorische Karte, welche den Ausgangspunkt für die Vereinfachung darstellt, repräsentiert alle Voxel eines Volumens – man nennt sie Level-0-Karte. Diese Karte hat eine implizite kanonische Einbettung, nämlich die übliche Geometrie des Voxelgitters. Die schrittweise Vereinfachung dieser Karte geschieht auf der impliziten Geometrie der Voxel. Abbildung 22 zeigt die schrittweise Vereinfachung einer Karte.



(a) Label-Volumen (b) Level-1-Karte (c) Level-2-Karte

(d) Level-3 Karte (e) Level-4-Karte (f) Level-5-Karte

Abbildung 22: Schritte der Verschmelzung beim HLE-Modell (aus [BDDV03])

Zuerst werden alle adjazenten Volumen der Level-0-Karte, die zur selben Region des Bildes gehören, verschmolzen. Dieser Schritt entfernt alle inneren Flächen, so dass man die Begrenzungen aller Regionen des Bildes erhält – man erhält die Level-1-Karte. Als nächstes werden alle adjazenten und koplanaren Flächen[24] verschmolzen, die durch eine Kante vom Grad eins oder zwei voneinander getrennt sind. Der Grad einer Kante bestimmt sich durch die Anzahl der zu ihr inzident liegenden Flächen. Dieser Schritt erlaubt es überflüssige Kanten in der Mitte einer ebenen Fläche zu entfernen – nun liegt eine Level-2-Karte vor. Darauf folgend werden alle adjazenten und kollinearen Kanten[25], die durch einen Knoten vom Grad zwei getrennt werden, verschmolzen. Der Grad eines Knotens bestimmt sich durch die Anzahl der zu ihm inzidenten Kanten. Man erhält eine Level-3-Karte. Um eine minimale Repräsentation eines Volumens zu erhalten, die nur auf der Topologie beruht, ist es notwendig, nicht koplanare Flächen und nicht kollineare Kanten mit ähnlichen Operationen zu verschmelzen. So werden – ausgehend von einer Level-3-Karte – alle adjazenten und nicht koplanaren Flächen, die durch Kanten vom Grad eins oder zwei getrennt sind, verschmolzen. Nach diesem Schritt liegt die

24 Koplanare Flächen sind Flächen, die eine koplanare Einbettung besitzen.

25 Kollineare Kanten sind Kanten, die kollineare Einbettungen besitzen.

Level-4-Karte vor. Die Level-5-Karte erhält man, indem man alle adjazenten und nicht kollinearen Kanten verschmilzt, die durch Knoten vom Grad zwei getrennt sind.

Die Flächenverschmelzung kann dazu führen, dass die Karte in mehrere Zusammenhangskomponenten zerfällt, die nicht untereinander verbunden sind. Dies geschieht bei Flächen, die mehr als eine Begrenzung besitzen. Nach dem Verlust der Verbundenheit geht die Möglichkeit verloren, diese unterschiedlichen Komponenten über Orbits zu erreichen. Es geht also topologische Information verloren. Um dieses Problem zu lösen, kann man spezielle Kanten einführen, die die Karte zusammenhalten. Die Kanten werden fiktive Kanten genannt, da sie nicht die Kante einer Fläche beschreiben.

Die letzte Karte, die man erhält, ist die minimale Karte. Sie beschreibt die Topologie des Volumens. Alle bis hierher ausgeführten Vereinfachungsschritte haben die Topologie nicht verändert. Des Weiteren kann kein Schritt mehr ausgeführt werden, der die Topologie nicht verändert. Deswegen wird diese Karte auch topologische Karte genannt.

Es gibt verschiedene Möglichkeiten, geometrische Informationen zur topologischen Karte hinzuzufügen. Die Wahl sollte von der Anwendung und den benötigten Operationen abhängen. Die hierarchische lokale Einbettung eignet sich besonders für Volumen, welche aus vielen großen, ebenen Oberflächen bestehen (vgl. [BDDV03]). Um diese Art der Einbettung zu erhalten, wird jede Fläche der topologischen Karte mit einer eingebetteten 2-kombinatorischen Karte assoziiert, welche die Geometrie der Fläche repräsentiert. Diesen Sachverhalt stellt die Abbildung 23 für das obige Beispiel dar.

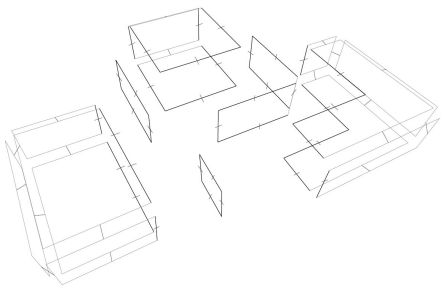


Abbildung 23: Einbettung der HLE-Karte aus dem Beispiel (aus [BDDV03])

Bei der Erstellung der topologischen Karte wird bei der Level-0-Karte begonnen, jede Voxelfläche durch eine eingebettete 2-kombinatorische Karte zu assoziieren. Bei jedem Verschmelzungsschritt muss nun natürlich auch die eingebettete Karte aktualisiert werden. Dafür lassen sich laut Braquelaire et al. einfache Algorithmen finden, die diese Aufgaben effizient lösen. So lässt sich das Prinzip auch auf höhere Dimensionen erweitern, denn die Einbettung einer *i*-dimensionalen Zelle ist in jedem Fall eine *i*-kombinatorische Karte.

3.4.3 Die generalisierte kombinatorische Karte G-Map

Im Gegensatz zu den kombinatorischen Karten, welche die Möglichkeit der Repräsentation orientierbarer Mannigfaltigkeiten haben, kann die G-Map zusätzlich auch

nicht orientierbare Quasi-Mannigfaltigkeiten repräsentieren (vgl. Kapitel 3.3). Dazu wird eine Definition, die leicht von der Definition 3.4.1 abweicht, benötigt.

Definition 3.4.5 (G-Map)[26] Sei $n \geq 0$. Eine *n*-dimensionale G-Map ist ein Tupel $G=(B, \beta_0, \ldots, \beta_n)$, mit:

1. B ist eine endliche Menge von Darts,
2. $\forall_i 0 \leq i \leq n$, β_i ist eine Involution auf B,
3. $\forall_i 0 \leq i \leq n, \forall_j i+2 \leq j \leq n$, $\beta_i \circ \beta_j$ ist eine Involution.

Die Abbildung 24 zeigt für den dreidimensionalen Fall, dass jede Permutation zweimal nacheinander angewandt wieder zum anfänglichen Element zurückführt. Somit sind diese Permutationen tatsächlich auch Involutionen.

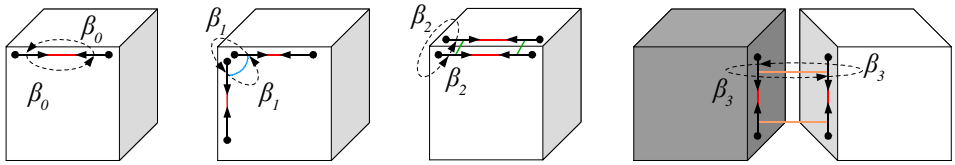


Abbildung 24: Involutionen der Beta-Orbits

Setzt man die obigen Involutionen wie in Punkt drei beschrieben zusammen, so ergeben sich auch wieder Involutionen, wie man in Abbildung 25, die den dreidimensionalen Fall darstellt, leicht erkennen kann.

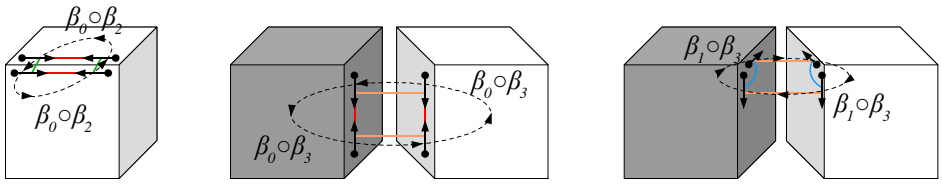


Abbildung 25: Verkettete Involutionen, die wiederum Involutionen darstellen.

Sei *G* eine *n*-G-Map und *S* die korrespondierende Unterteilung. Genau ein Dart von *G* korrespondiert zu einem *(n+1)*-Tupel von Zellen $(z_0, \ldots, z_n)$, wobei z_i eine *i*-Zelle sei und z_i und z_{i+1} inzident sind. β_i assoziiert zwei Darts mit $(z_0, \ldots, z_n)$ und $(z'_0, \ldots, z'_n)$ mit $\forall_{j \in [0,\ldots,n] \setminus \{i\}} z_j = z'_j$ und $z_i \neq z'_i$. Somit vertauscht β_i zwei *i*-Zellen, die inzident zur selben (*i-1*) und (*i+1*)-Zelle sind.

Zur Veranschaulichung kann man eine der beiden obigen Abbildungen heranziehen. So zeigt das linke Bild in Abbildung 24 den Fall des β_0-Orbits. Die in diesem Fall durch das Beta-Orbit zu vertauschenden *i*-Zellen sind die beiden Knoten, an denen die Darts

26 siehe [DL02]

starten. Knoten oder einzelne Punkte stellen bekanntermaßen 0-Zellen dar. Leicht zu erkennen ist, dass die Gerade (1-Zelle), die Fläche (2-Zelle) und auch das Volumen (3-Zelle) nicht verlassen wird. Auch bei anderen Orbits lässt sich die obige Aussage nachvollziehen. Nimmt man das dritte Bild von links in Abbildung 24 zur Anschauung, so sieht man, dass durch Anwendung des β_2-Orbits die 2-Zelle, also die Fläche gewechselt wird. Allerdings bleibt man bei der gleichen 0-Zelle. Beide Darts starten im gleichen Knoten und auch die 1-Zelle und 3-Zelle bleiben nach Anwendung des Orbits gleich. Für die anderen Orbits kann man den Sachverhalt ebenfalls analog aufzeigen.

An dieser Stelle sei darauf hingewiesen, dass G-Maps *i*-Zellen implizit repräsentieren:

Definition 3.4.6 (*i*-Zelle inzident zu einem Dart)[27] Sei G eine *n*-G-Map, *b* sei ein Dart und $i \in \{0,\ldots,n\}$. Die *i*-Zelle, welche zu *b* inzident ist, ist das Orbit $\langle \beta_{0,}\ldots,\beta_{i-1},\beta_{i+1},\ldots,\beta_n\rangle(b)$, welches als $\langle\rangle_{N-\{i\}}(b)$ notiert wird.

Anschaulich ist eine *i*-Zelle die Menge aller Darts, welche ausgehend von einem Dart *b* erreichbar sind, wenn man alle Involutionen außer β_i anwendet. Die Menge von *i*-Zellen ist eine Aufteilung der Darts der G-Map. Zwei Zellen sind disjunkt, wenn ihre Schnittmenge leer ist, also wenn diese beiden Zellen keine gemeinsamen Darts besitzen.

Definition 3.4.7 (Anker einer *i*-Zelle) Sei *G* eine *n*-G-Map, $i \in \{0,\ldots,n\}$. Sei *b* ein beliebiger Dart, der zu einer *i*-Zelle *Z* inzident ist. Der Anker der *i*-Zelle *Z* ist definiert durch den Anker des Orbits[28] $\langle\rangle_{N-\{i\}}(b)$.

Um diese Definition zu veranschaulichen, wird folgendes Beispiel in der Abbildung 26 angeführt:

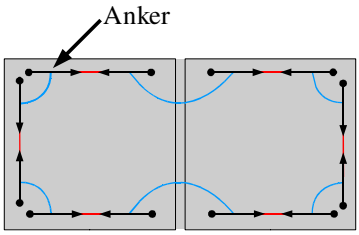


Abbildung 26: Darstellung eines Ankers einer 2-Zelle.

3.4.3.1 Orientierbarkeit einer G-Map

Die Orientierbarkeit liefert eine weitere Möglichkeit, Objekte in Volumen zu charakterisieren. Um die Orientierbarkeit einer G-Map zu erhalten, gibt es einen einfachen Algorithmus (vgl. [LM99]). Die Orientierbarkeit eines Objektes kann gewonnen werden, indem man ein Paar von Darts $(d, \beta_0(d))$ als einen Magneten betrachtet, der einen Nord- und einen Südpol hat. Ist es für ein Objekt möglich, den Darts

27 vgl. [DL02]
28 siehe Kapitel 2.3

Nord- und Südpole zuzuweisen, ohne dass die Gesetze des Magnetismus verletzt werden, so ist das Objekt orientierbar.

Das Zuweisen der Nord- und Südpole kann man erreichen, indem man die Menge von Darts D in zwei Mengen D^+ und D^- aufteilt. Die Gesetze des Magnetismus können dann auf die Weise formalisiert werden, dass nur Darts mit unterschiedlicher Polarität verbunden werden können:

$$\begin{cases} \forall_{d \in D^+} \; \forall_{0 \le i \le N} \; \beta_i(d) \in D^- \\ \text{und umgekehrt:} \\ \forall_{d \in D^-} \; \forall_{0 \le i \le N} \; \beta_i(d) \in D^+ \end{cases}$$

Die Abbildung 27 zeigt eine orientierbare 2-G-Map. Die Darts wurden in unterschiedliche Polaritäten aufgeteilt, dargestellt durch die Farben Lila und Türkis. Jeder Dart einer Farbe ist nur mit Darts der anderen Farbe über die Orbits verbunden. In der Abbildung wurden die Orbits, die am Rand liegen, der Einfachheit halber nicht mit eingezeichnet.

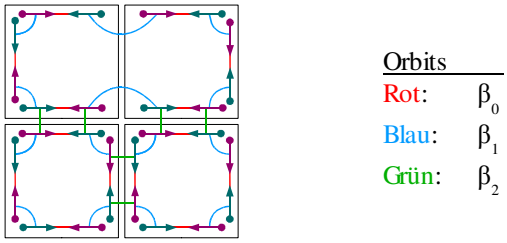


Abbildung 27: Beispiel für eine orientierbare 2-G-Map

3.4.3.2 Erstellung einer G-Map

Um eine G-Map zu erstellen, gibt es unterschiedliche Ansätze. Einer dieser Ansätze ist es (vgl. [BDF00]), mit einer vollständigen G-Map zu beginnen und nacheinander verschiedene Verschmelzungsschritte auszuführen. Diesen Vorgang verdeutlicht Abbildung 22. Der für diese Arbeit gewählte Ansatz verwendet die gleichen Verschmelzungsoperationen, die bei der Erstellung der HLE-Karte angewandt wurden (siehe Abschnitt 3.4.2). Nach Bertrand et al. kann dieser Ansatz allerdings zu Leistungsproblemen führen (vgl. [BFP99]). Daher führen sie die sogenannten Precodes ein.

Für die Erstellung einer 3-G-Map der ersten Stufe werden acht Precodes benötigt. Sie haben in diesem Fall die Form von vier Voxeln und beschreiben jeweils unterschiedliche Möglichkeiten, wie Regionen zusammenliegen können. Die acht Precodes zeigt Abbildung 28.

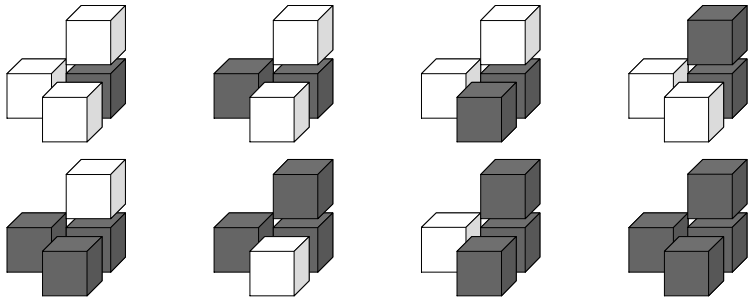


Abbildung 28: Die acht Precodes, die zur Level-1-Karte führen (der aktuelle Voxel liegt hinten rechts unten).

Mit Hilfe dieser Precodes müssen die Voxel des Volumens nur noch einmal abgelaufen werden, um die zugehörige G-Map zu erstellen. Dabei ist für jeden Voxel ein Abgleich mit den Precodes nötig, um bei dem passenden die zugehörigen Operationen auszuführen. Allerdings ist es mit dieser Methode zuerst nur möglich, eine Level-1-Karte zu erstellen. In [BDF00] beschreiben Bertrand et al. dann alle benötigten Precodes um eine sogenannte Border-Map zu erstellen, die einer Level-3-Karte entspricht (die Precodes, die zu einer Level-2-Karte führen, finden sich in Anhang B). Sie erwähnen ebenfalls, dass es ihnen gelungen sei, den Aufwand für die Precodes bis zu einer topologischen Karte (Level-5-Karte) so gering zu halten, dass die Ausführung möglich sei. Für die jeweiligen Level-Karten werden in jeder Stufe mehr Precodes benötigt. Bertrand et al. geben diese Anzahlen an:

- 8 Precodes für die Level-1-Karte
- 18 Precodes für die Level-2-Karte
- 27 Precodes für die Level-3-Karte
- 98 Precodes für die Level-4-Karte
- 216 Precodes für die Level-5-Karte

Somit müssen für jeden Voxel des Volumens 367 Precodes verarbeitet werden. Diese Verarbeitung soll allerdings relativ schnell zu berechnen sein, so dass die G-Map in kurzer Zeit erzeugt werden kann.

Ein weiterer Ansatz, der für diese Arbeit entwickelt wurde, arbeitet zunächst auf einem Volumen (der sogenannten Volume-Map), aus dem dann nach einiger Verarbeitung eine Border-Map erstellt wird (siehe hierzu auch Kapitel 6.2). Als Eingabe wird, wie in allen Fällen, ein vorsegmentiertes Volumen benötigt, in dem die Regionen in ikonischer Weise repräsentiert sind. Jeder Voxel der Volume-Map erhält einen bestimmten Wert, der bedeutet, dass an allen möglichen Stellen Darts sein sollten. In dieser Phase entspricht die Volume-Map einer Level-0-Karte. Abbildung 29 zeigt die Repräsentation der Volume-Map. Jeder Voxel darin wird als eine 3-Zelle aufgefasst und entspricht somit einem Würfel, dessen sechs Seiten jeweils vier Kanten haben, die acht Darts darstellen.

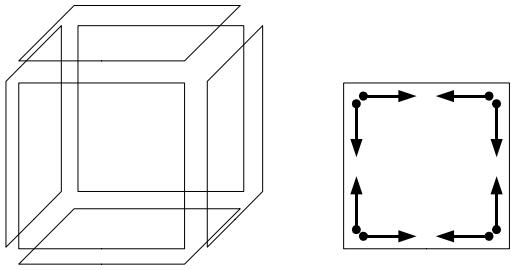


Abbildung 29: Volume-Map-Repräsentation: Darstellung eines Voxels (links) und Aufbau der Seitenflächen (rechts).

Durch Anwendung von Precodes, die denen von Bertrand et. al entsprechen (vgl. [BDF00]), wird in einem Durchgang durch das Volumen eine Level-2-Karte erstellt. Die Precodes finden sich in Abbildung 28 beziehungsweise für die Level-2-Karte in Anhang B. In der Volume-Map wird für jeden Voxel gespeichert, an welcher Position Darts sein sollten, jedoch ist es nicht notwendig, dass diese Darts explizit vorhanden sind. Diese Tatsache beschleunigt den Ablauf ungemein, da wenig Speicher verbraucht wird und auch nur einmal jeder Voxel des Volumens betrachtet wird. Aus der Volume-Map kann relativ schnell eine Level-3-G-Map, also eine Border-Map, erstellt werden. Erst bei dieser Erstellung müssen auch die Darts erzeugt werden. Da auf dem Weg zur fertigen Volume-Map schon viele mögliche Darts weggefallen sind, die nicht mehr erzeugt werden müssen, spart man einen großen Rechenaufwand und Speicherbedarf.

Eine übliche geometrische Einbettung für eine G-Map ist, jedem Dart Informationen darüber zu geben, zu welcher *i*-Zelle er gehört (siehe hierzu [LM99]). Dieser Ansatz ist ähnlich dem der HLE-Karten. Allerdings erwähnen Lévy et. al. auch die Möglichkeit anderer Repräsentanten für die geometrischen Informationen. In dieser Arbeit hat jeder Dart einen Koordinatenvektor, der die räumliche Lage beschreibt.

3.4.4 Vergleich der unterschiedlichen Modelle

Alle drei oben vorgestellten Modelle zur Repräsentation von Topologie und Geometrie stellen in erster Linie dreidimensionale topologische Karten dar. Jedes Modell hat zusätzlich noch eine Möglichkeit die Geometrie zu beschreiben. Das Modell der HLE-Karten und die G-Map unterscheiden sich in der Darstellung der topologischen Karte, sowie in der Erzeugung. Allerdings gibt es natürlich verschiedene Arten der Erzeugung beider Modelle, so dass auch beide auf die gleiche Weise erzeugt werden können. Auch die Repräsentation der Geometrie kann bei beiden Modellen auf die gleiche Weise dargestellt werden, denn wie bereits oben erwähnt, fußt diese jeweils auf eingebettete *i*-Zellen.

Der Unterschied der topologischen Karte besteht in der Anzahl der Orbits und der benötigten Darts. Bei den HLE-Karten wird jede Kante nur durch einen Dart repräsentiert. Hingegen werden bei der G-Map zwei Darts benötigt, die durch ein weiteres Orbit β_0 vernäht sind. Dieser Sachverhalt wird beispielsweise bei der Überprüfung der Orientierbarkeit benutzt (siehe 3.4.3.1). Die HLE-Karten erfahren

hingegen durch ihre Darstellung von Kanten eine Gerichtetheit, die eine Erstellung der Karten von nicht orientierbaren Objekten nicht zulassen. Durch den durchdachten Aufbau einer G-Map lässt diese sich auch richten. Dazu kann man wieder die Polaritäten aus Abschnitt 3.4.3.1 benutzen und den Umlauf darauf einschränken, dass Kanten nur vom negativen zum positiven Dart abgelaufen werden.

Zum Aufwand beider Modelle ist zu sagen, dass dieser natürlich stark von der Umsetzung abhängt. In Abschnitt 3.4.3.2 wurden bereits Möglichkeiten der Effizienzsteigerung genannt.

Es bleibt also festzuhalten, dass ein Unterschied vor allem in der Möglichkeit liegt, dass das Modell der G-Map nicht orientierbare Objekte zulässt, das der HLE-Karte nicht.

Zum Modell der GE topologischen Karten gibt es scheinbar einen viel größeren Unterschied, der sich bei genauerer Betrachtung relativiert. Allerdings haben diese Karten den Nachteil, dass nur wohlgeformte Daten verwendet werden dürfen, da es sonst bei der Anwendung des Zerlegungsalgorithmus' zu Problemen kommen kann (vgl. [BDDV03]). Letztlich liegt der Unterschied vielmehr in der Art der Erstellung und in der Repräsentation der Einbettung der Geometrie als in der Aussagekraft. So wird in [BDDV03] eine Möglichkeit genannt, HLE- und GE-Modell ineinander zu überführen. Die Permutationen aus M_{HLE} lassen sich durch die Permutationen aus M_{GE} darstellen und umgekehrt:

$$\gamma = \beta_3\beta_2,\ \ \sigma = \beta_3\beta_1,\ \ \alpha = \beta_3 \text{ beziehungsweise}$$

$$\beta_1 = \sigma\alpha,\ \ \beta_2 = \gamma^{-1},\ \ \beta_3 = \alpha\,.$$

Somit erlauben beide Modelle die gleiche Mächtigkeit der Beschreibung von Topologie. Ebenfalls ist eine Überführung der geometrischen Repräsentationen laut Braquelaire et al. zwischen beiden Modellen möglich. Die Auswahl eines Modelles sollte daher vom Einsatzzweck abhängen.

3.4.5 XG-Map als Erweiterung der G-Map

Bevor die Definition der XG-Map erfolgt, soll noch darauf eingegangen werden, was hierfür die Motivation darstellte. Denn die bisher vorgestellten kombinatorischen Karten haben, so viele Möglichkeiten der Repräsentation (geometrisch und topologisch) sie auch bieten, auch Probleme. Eines der unangenehmsten ist, dass Regionen oder Flächen, welche Löcher beinhalten, nicht explizit repräsentiert werden können.

Es gibt einige Ansätze dies zu umgehen: So schlagen zum Beispiel Damiand & Resch vor, dass jede Fläche homöomorph zu einer topologischen Scheibe bleiben muss (vgl. [DR02]). Bei diesem Ansatz werden allerdings Hilfsdarts eingeführt, die keiner Repräsentation von geometrischen Eigenschaften entsprechen, und deshalb im weiteren Verfahren auch gesondert behandelt werden müssen. Diese Repräsentation ist analog zu dem in Abschnitt 3.4.1 vorgestellten Ansatz von Kropatsch. Dennoch ermöglichen diese Ansätze keine explizite Repräsentation der Eingeschlossenheit.

Als eine Alternative zu den oben angeführten klassischen Strategien zum Einbinden von Einschlüssen in kombinatorischen Karten haben sich für zweidimensionale Bilddaten die

XPMaps hervorgetan[29], sie besitzen die Möglichkeit Flächen mit Löchern (siehe Abbildung 30) explizit zu repräsentieren:

Definition 3.4.8 (XPMap)[26] Eine erweiterte planare Karte (e**X**tended **P**lanar Map) ist ein Tupel $(C, c_{0,} exterior, contains)$ mit:

- C ist eine Menge nicht trivialer kombinatorischer Karten,
- c_0 ist eine triviale Karte, welche die unendliche Fläche der XPMap bildet,
- *exterior ist* eine Relation, die ein ϕ-Orbit jeder Komponente aus C als äußeres Orbit markiert, und
- *contains* ist eine Relation, die jedem äußeren Orbit genau ein nicht-äußeres ϕ-Orbit oder das leere Orbit in c_0 zuweist.

Wie aus der Definition ersichtlich, liefert die Relation *contains* zusammen mit der *exterior* Eigenschaft der ϕ-Orbits die notwendigen Hilfsmittel einer expliziten Darstellung des Einschlusses von Flächen in zweidimensionalen Bilddaten. Dazu erläutert die folgende Abbildung beispielhaft, wie sich die Relationen in der Praxis auswirken.

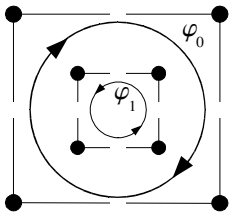


Abbildung 30: Darstellung einer XPMap: die Relation exterior weist nur ϕ_0, nicht aber ϕ_1 zu, dass es ein äußeres Orbit ist. Zusätzlich gilt: $contains(\phi_0)=\phi_1$, womit der Einschluss der inneren Fläche repräsentiert wird.

Aufbauend auf diese Definition können nun erweiterte dreidimensionale G-Maps definiert werden, die ähnliche Eigenschaften wie zweidimensionale XPMaps besitzen, aber auf dreidimensionalen G-Maps aufbauen.

Vorher wird allerdings noch eine Hilfsdefinition benötigt, die sich im Folgenden als sehr praktisch herausstellen wird (siehe Kapitel 7.2).

Definition 3.4.9 (Zellschlüssel) Ein Zellschlüssel *CK* eines Orbits α, angewandt auf einen Dart b_0, ist definiert durch:

$$CK: Orbits \rightarrow Darts$$

$CK(\alpha(b_0))=b$, wobei b Anker des Orbits $(\alpha)(b_0)$ ist.

29 siehe hierzu [Köt01] und [Köt02]

Definition 3.4.10 (3-XG-Map) Eine dreidimensionale erweiterte G-Map (eXtended G-Map) ist ein Tupel $(G, g_0, contains_2, exterior_2, contains_3, exterior_3)$ mit

- G ist eine nicht triviale 3-G-Map,
- g_0 ist eine triviale 3-G-Map, welche das unendliche Volumen der 3-XG-Map bildet,
- $exterior_i$ (mit $i \in \{2,3\}$) ist eine Relation, die je ein Orbit $(\beta_0 \circ \ldots \circ \beta_{i-1})$ jeder Zusammenhangskomponente aus G als äußeres Orbit markiert, und
- $contains_i$ (mit $i \in \{2,3\}$) ist eine Relation, die jedem äußeren Orbit eine Menge nicht äußerer Orbits $(\beta_0 \circ \ldots \circ \beta_{i-1})$ oder das leere Orbit in g_0 zuweist.

Die Erweiterung zur XG-Map ist deswegen umfangreicher als die XPMap, weil sich G-Maps für beliebige Dimensionen definieren lassen. Bei einer 3-G-Map können zusätzlich zu Flächeneinschlüssen auch Volumeneinschlüsse auftreten. So benötigt die 3-XG-Map bereits je zwei *contains* und *exterior* Relationen.

Wie an dieser Definition leicht abzulesen ist, handelt es sich im Grunde genommen um eine Erweiterung, welche nahe an der HLE topologischen Karte liegt. Wie bereits in Kapitel 3.4.4 erwähnt wird, gibt es durchaus triftige Gründe, sich für eine hierarchisch aufgebaute Karte zu entscheiden. Um diese hierarchische Sichtweise auf die topologischen Karten noch zu untermauern, wird im Folgenden eine Klasse von topologischen Modellen beschrieben, die sogenannten Schalenmodelle. Die 3-XG-Map stellt eine Möglichkeit dar, ein solches Schalenmodell zu repräsentieren. Aus diesem Grund folgen nun einige Definitionen. Die erste schränkt die Klasse der repräsentierbaren Objekte ein, da Objekte immer durch eine äußere Schale begrenzt sein müssen.

Definition 3.4.11: (Feste Körper) Objekte mit zusammenhängenden orientierbaren Oberflächen heißen *feste Körper*, wenn sie folgende Bedingungen erfüllen:

- Schnitte von Oberflächen dürfen nur an den Grenzen benachbarter Flächen schneiden. Es ist keine Selbstdurchdringung zugelassen.
- Flächen können einfach oder mehrfach zusammenhängend sein, müssen jedoch zweidimensional mannigfaltig sein.
- Kanten dürfen sich nur an gemeinsamen Endpunkten schneiden.

(Das heißt: die Klein'sche Flasche entspricht nicht dieser Definition!)

Definition 3.4.12 (Region) Unter einer *Region* wird ein Raumvolumen verstanden, welches gegebenenfalls durch *Schalen* begrenzt wird.

Definition 3.4.13 (Schale) Eine *Schale* ist eine orientierte Begrenzung jeder *Region.* Einzelne Regionen können mehr als eine Schale haben (beispielsweise Regionen mit Einschlüssen).

Definition 3.4.14 (Fläche) Eine Fläche ist ein begrenzter Teil einer Schale, welcher ebenfalls orientierbar ist.

Mit diesen Definitionen fällt es nun leicht ein hierarchisches Modell für die Topologie, aufbauend auf der 3-XG-Map, zu entwickeln.

Sei $(G, g_0, contains_2, exterior_2, contains_3, exterior_3)$ eine 3-XG-Map und *B* die Menge aller Darts von *G*.

Eine *Flächenkontur* eines Darts $b \in B$ wird beschrieben durch:

$$FaceContour(b) = (\beta_0 \circ \beta_1)_b ,$$

wobei $(\beta_0 \circ \beta_1)_b$ das $(\beta_0 \circ \beta_1)$-Orbit um den Dart *b* beschreibt.

Die Menge aller *Flächenkonturen* ist gegeben durch:

$$FaceContours = \bigcup_{b \in B} \{FaceCountour(b)\} .$$

Eine *Fläche*, zu der ein Dart $b \in B$ gehört, ist gegeben durch:

$$Face(b) = (c_{exterior}, C_{contains}) \text{ mit:}$$

$$c_{exterior} = \begin{cases} FaceContour(b) & \text{, falls } exterior_2(FaceContour(b)) \\ contains_2^{-1}(FaceContour(b)) & \text{, sonst} \end{cases} ,$$

$$C_{contains} = contains_2(c_{exterior}) ,$$

wobei $c_{exterior}$ die äußere Flächenkontur und $C_{contains}$ die Menge der inneren Flächenkonturen von $c_{exterior}$ ist.

Jede *Fläche* befindet sich in einem *Oberflächenteil* eingebettet, welches im Folgenden definiert wird. Es beschreibt eine Flächenzusammenhangskomponente der folgenden Art:

Ein Oberflächenteil, zu dem eine Fläche *Face(b)* gehört, ist gegeben durch:

$$SurfacePart(Face(b)) = \bigcup_{d \in (\beta_0 \circ \beta_1 \circ \beta_2)_b} Face(d) \quad \text{, falls } exterior_2(FaceContour(d)) ,$$

wobei $(\beta_0 \circ \beta_1 \circ \beta_2)_b$ das $(\beta_0 \circ \beta_1 \circ \beta_2)$-Orbit um den Dart *b* beschreibt.

Die Menge aller *Oberflächenteile* ist gegeben durch:

$$SurfaceParts = \bigcup_{b \in B} \{SurfacePart(b)\} .$$

Anschaulich lässt sich festhalten, dass sich ein Oberflächenteil aus den verschiedenen Flächen zusammensetzt, deren Konturen $\beta_0 \circ \beta_1 \circ \beta_2$-erreichbar sind, und keine inneren Konturen darstellen.

Diese Repräsentation heißt Oberflächenteil, da bisher darunter lediglich spezielle $(\beta_0, \beta_1, \beta_2)$-Zusammenhangskomponenten verstanden werden. Zusammenhängende Oberflächenteile werden in dieser Arbeit durch den Begriff der Oberfläche oder auch Schale gekennzeichnet:

Eine Schale *S* eines Oberflächenteils sp_0 ist eine Teilmenge aller Oberflächen, so dass gilt:

Falls $sp_0 \in S$:

Dann gilt für alle enthaltenen Flächen $f(b) \in sp_0$, mit $f(b) = (c_{exterior}, C_{contains})$:

1. Für alle Flächenkonturen $c \in C_{contains} \cup c_{exterior}$:

   $sp = SurfacePart(Face(\beta_2(c)))$ ist ebenfalls in *S* enthalten.

1. Andere Oberflächen sind nicht in *S* enthalten.

Diese Definition einer Schale beschreibt anschaulich, dass einzelne Oberflächenteile, die eine Schale bilden, auch über innere Flächenkonturen miteinander verbunden sein können (vgl. die verschiedenen Fälle aus Abschnitt 3.4.1).

Eine *Region*, zu der ein Dart $b \in B$ gehört, ist gegeben durch:

$Region(b) = (c_{exterior}, C_{contains})$ mit:

$c_{exterior} = S$, wobei *S* die Schale ist, zu der ein Oberflächenteil $sp \in S$ gehört, welches definiert ist durch:

$$sp = \begin{cases} SurfacePart(Face(b)) & \text{, falls } exterior_3((\beta_0 \circ \beta_1 \circ \beta_2)_b) \\ SurfacePart(Face(b_o)) & \text{, sonst} \end{cases},$$

wobei $b_o \in contains_3^{-1}((\beta_0 \circ \beta_1 \circ \beta_2)_b)$ und

$C_{contains} = \{ s \in S | d \in (contains_3(c_{exterior})) \wedge sp(Face(d)) \in s \}$,

wobei $c_{exterior}$ die äußere Schale und $C_{contains}$ die Menge der inneren Schalen von $c_{exterior}$ ist.

Somit ist es möglich, eine baumartige hierarchische Repräsentation der Topologie aufbauend auf einer 3-G-Map zu definieren, wie es durch die 3-XG-Map angestrebt wird.

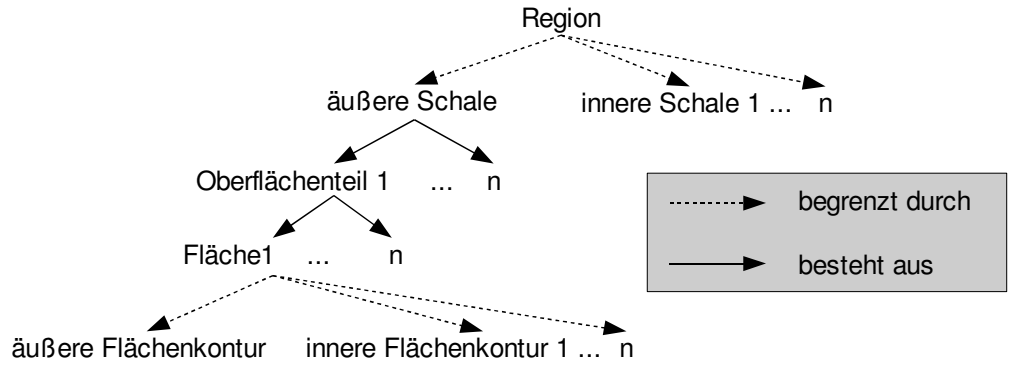


Abbildung 31: Grafische Darstellung der Hierarchie, die im Rahmen der 3-XG-Map definiert wurde. Gut erkennbar ist die Struktur des topologischen Schalenmodells.

3.4.6 Euler-Charakteristik von Schalen in der 3-XG-Map

Mit Hilfe des gewählten und oben beschriebenen Aufbaus der 3-XG-Map, in der Objekte bzw. Regionen anhand von Schalen repräsentiert werden, ist es leicht möglich topologische Eigenschaften dieser Schalen zu bestimmen. Anhand der Schalencharakteristik lässt sich auf die topologische Beschaffenheit der Objekte bzw. Regionen schließen. Ein wichtiges topologisches Merkmal von Oberflächen stellt die Euler-Charakteristik dar. Diese berechnet sich nach der Formel aus Abschnitt 3.3.2.

Im Falle der 3-XG-Map lautet diese Gleichung: $\chi = V - E + F$ mit:

$V = \sum_{b \in S} \left| \langle \rangle_{N-\{0\}} \right|$ ist die Anzahl der Knoten,

$E = \sum_{b \in S} \left| \langle \rangle_{N-\{1\}} \right| + \sum_{f(b) \in S} \left| C_{contains} \right|$ ist die Anzahl der Kanten und

$F = \left| f(b) \in S \right|$ ist die Anzahl der Flächen.

Wobei $f(b) \in S$ abkürzend für alle Flächen aller Oberflächenteile der Schale *S* steht. Des Weiteren bezeichnet $b \in S$ alle Darts der Schale *S*.

Das weitere Merkmal, welches in Abschnitt 3.3.2 vorgestellt wurde, ist der Genus einer Oberfläche. Auch für die Schalen einer Region, welche in einer 3-XG-Map repräsentiert sind, lässt sich dieser bestimmen. Dazu wird die berechnete Euler-Charakteristik nach der Formel aus Definition 3.3.9 umgeformt.

Die Homöomorphie zweier Schalen lässt sich somit bestimmen, da zusätzlich ausgenutzt wird, dass Schalen orientierbare Oberflächen sind.

4 Die Kapselung der Volumennachbarschaften

Dieses Kapitel beschreibt die Funktionalität der VoxelNeighborhood, welche die Kapselung der in Kapitel 3 beschriebenen 6er- und 26er-Nachbarschaften für Voxel darstellt. Für eine Kapselung dieser Art ist es sinnvoll, dass sie generisch programmiert ist. Dadurch wird sichergestellt, dass sie auch für andere Einsatzzwecke von Nutzen ist und nicht nur Anwendung für das Wasserscheidenverfahren (siehe 5.3) findet. Des Weiteren sollte sie ermöglichen, dass der Anwender die benötigte Nachbarschaft – 6er- oder 26er-Nachbarschaft – frei auswählen kann. Ebenso sollte sie klar und einfach strukturiert sein, damit auch andere Anwender Zugriff auf die Funktionalität haben.

Im Folgenden werden daher Aufbau und Funktionsumfang beschrieben, welche sich an der PixelNeighborhood der VIGRA orientieren.[30] Dieses Vorgehen soll eine spätere Verwendung der VoxelNeighborhood in der VIGRA ermöglichen. Die Kapselung gliedert sich daher in mehrere Bereiche:

1. Rand-Definitionen
2. NeighborCode3DSix
3. NeighborCode3DTwentySix
4. NeighborOffsetTraverser
5. NeighborhoodTraverser
6. RestrictedNeighborhoodTraverser

Die jeweiligen Funktionalitäten und Zusammenhänge der hier genannten Bereiche der Volumennachbarschaften werden in den nächsten Abschnitten erläutert.

30 Eine Dokumentation der PixelNeighborhood findet sich in [Vig06b].

4.1 Rand-Definitionen

Zunächst wird definiert, in welchen Fällen, die ein Voxel im Volumen annehmen kann, dieser sich an welchem Rand befindet. Der trivialste Fall ist sicherlich der, dass der betrachtete Voxel nicht am Rand liegt (`NotAtBorder`). Diesen Fall kann man ebenso als Randfall betrachten, denn es ist der Fall, bei dem der Voxel an keinem Rand anliegt. Danach werden die Randfälle definiert, bei denen der Voxel nur an einen Rand stößt, zum Beispiel `LeftBorder`, wenn der Voxel am linken Rand des Volumens liegt, anschließend die Fälle, bei denen der Voxel an mehrere Ränder des Volumens anliegt. Diese lassen sich aus den einfachen Fällen zusammensetzen, so zum Beispiel `BottomLeftBorder`, wenn der Voxel sowohl am unteren Rand als auch am linken Rand des Volumens liegt. Auch möglich ist beispielsweise `TopRightRearBorder`, bei dem der Voxel an drei Rändern des Volumens liegt, in diesem Fall am oberen, rechten und hinteren Rand. Jeder dieser Randfälle bekommt einen eindeutigen Wert zugewiesen. Für die einfachen Fälle sind das die Zweierpotenzen (1,2,4,8,16,32), so dass man die zusammengesetzten Fälle durch bitweise Anwendung der „Oder"-Operation dieser Werte erhält. Anhand der gesetzten Bits kann somit die Randposition wieder heraus gelesen werden. Bei der hier beschriebenen Art der Rand-Definition ist es nicht zulässig, dass ein Voxel gleichzeitig an zwei gegenüberliegenden Rändern liegt. Aus diesem Grund sind Volumen mit weniger als zwei Voxeln in einer Dimension nicht für die Anwendung der VoxelNeighborhood zulässig.

Des Weiteren wird die Funktionalität bereit gestellt, mit der man erfahren kann, ob und an welchem Rand sich der betrachtete Voxel befindet (`isAtVolumeBorder`). Dies ist sowohl mit allen verfügbaren, aber auch nur unter Betrachtung der kausalen oder anti-kausalen[31] Richtungen möglich. Um diese Informationen gewinnen zu können, sind sowohl die Koordinaten des Voxels im Volumen erforderlich, als auch die Dimensionen des betrachteten Volumens.

31 Eine kausale Richtung ist die, die von einem Voxel v auf einen Voxel v_1 zeigt, mit $v_1 \prec v$ (siehe 5.2.3.2, Definition 5.2.10). Eine anti-kausale Richtung ist entsprechend eine Richtung, die von einem Voxel v auf einen Voxel v_2 zeigt, mit $v \prec v_2$.

4.2 Implementation der 3D-Nachbarschaften

In diesem Abschnitt werden die Implementationen der 6er- bzw. 26er-Nachbarschaft beschrieben. Sie stellen eine Umsetzung der in Kapitel 3 definierten Nachbarschaften dar. Da beide Klassen, wie bereits erwähnt, eine Erweiterung der 4er- bzw. 8er-Nachbarschaft der PIXELNEIGHBORHOOD sind, sei an dieser Stelle noch einmal auf diese verwiesen (siehe [Vig06b]).

Damit eine äquivalente Anwendung beider Nachbarschaften gewährleistet ist, unterscheiden sich beide Kodierungen der Nachbarschaften nur in der Anzahl der Richtungen und deren Behandlung. Ein kurzfristiges Austauschen beider in einer Anwendung ist aus diesem Grund problemlos möglich. Man erzeugt dafür einfach die jeweils andere Nachbarschaftskodierung. Spezielle Konstruktoren sind nicht notwendig, so dass der Standardkonstruktor verwendet werden kann.

4.2.1 NEIGHBORCODE3DSIX

In der Klasse NEIGHBORCODE3DSIX wird die 6er-Nachbarschaft (siehe Abschnitt 3.2.2) zur Verfügung gestellt. Um die Nachbarn eindeutig zu unterscheiden, werden die Richtungen, in denen sie an den betrachteten Voxel anliegen, definiert. Diese Richtungen haben eine feste Reihenfolge und lauten:

- `InFront=0`
- `North=1`
- `West=2`
- `Behind=3`
- `South=4`
- `East=5`

Zudem bilden die Werte `Error=-1` und `DirectionCount=6` eindeutige Grenzen beim Traversieren über die Richtungen. Des Weiteren werden die Grenzen für die kausalen und anti-kausalen Richtungen zur Verfügung gestellt (`CausalFirst=InFront` bis `CausalLast=West` bzw. `AntiCausalFirst=Behind` bis `AntiCausalLast=East`). Zusätzlich werden noch Werte für die Berechnung der entgegengesetzten Richtung benötigt. Dabei wird das `OppositeDirPrefix` nur benötigt, da alle Beschreibungen konsistent zum NEIGHBORCODE3DTWENTYSIX sein sollen. Damit wird eine Gleichbehandlung beider Nachbarschaften durch äußere Funktionen ermöglicht. Der `OppositeOffset` hat den Wert drei. Man kann oben bei den Richtungen leicht feststellen, dass man immer genau bei der entgegengesetzten Richtung ankommt, wenn man in der Liste drei Positionen weitergeht. Dabei muss man wieder oben ansetzen (also modulo `DirectionCount`), sobald man am unteren Ende angekommen ist. Hierbei zeigt sich, dass die Richtungen in Bezug auf diesen Sachverhalt sinnvoll sortiert sind. Des Weiteren kann man sich die Richtungen auch als gesetztes Bit in einem Integer angeben lassen. Dafür wird eine der obigen Richtungen als Parameter übergeben. In dem Fall der 6er-Nachbarschaft kann dieser Integer-Wert problemlos in einen 8-Bit-Datentyp umgewandelt werden, um Speicher zu sparen, da nur maximal sechs Bits gesetzt werden.

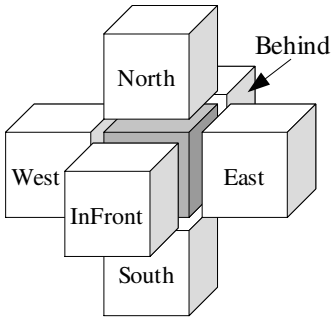


Abbildung 32: Übersicht über die Nachbarn des NEIGHBORCODE3DSIX

Bis zu dieser Stelle wurden alle möglichen Richtungen vorgestellt, in der ein Voxel Nachbarn besitzen kann. Allerdings hängen die genauen Positionen und die Anzahl der Nachbarn eines Voxels ebenfalls von der Position des betrachteten Voxels ab. Liegt der Voxel am Rand, so gibt es über diesen hinweg keine weiteren Voxel, somit hat der betrachtete Voxel in den entsprechenden Richtungen auch keine Nachbarvoxel. Durch eine spezielle Randbehandlung kann aber trotzdem ein fehlerfreies Traversieren über die existierenden Nachbarn gewährleistet werden. Dies ist dann die Randbehandlung für die 6er-Nachbarschaft.

Als erstes kann man über `nearBorderDirectionCount` unter Angabe einer Randposition (siehe 4.1) herausfinden, wie viele gültige Nachbarn der Voxel hat. Dies ermöglicht allerdings noch nicht, die zulässigen Nachbarn auch herauszufinden, was aber zum Traversieren notwendig wäre. Hierfür gibt es die Funktion `nearBorderDirections`, welche für jeden Randfall die noch übrig bleibenden Richtungen angeben kann. Unter Angabe der Randposition und des Index des Nachbarn, liefert sie dessen Richtung zurück. Dabei sind alle gültigen Nachbarn mit den kleinsten Indizes ausgestattet, so dass man diese auch solange durchlaufen kann, bis man auf eine ungültige Richtung stößt (`Error`). Die gleiche Möglichkeit gibt es auch nur mit den kausalen Richtungen. Dies ist wichtig für Scanline-Algorithmen[32], die nur bereits besuchte Voxel des Volumens betrachten (siehe Kapitel 5).

Bisher wurde beschrieben, wie man die gültigen Richtungen zu allen Nachbarn erhalten kann. Nun muss es noch eine Möglichkeit geben, auf diese im Volumen zuzugreifen. Dafür gibt es zu jeder Richtung einen Differenzvektor, den man auf die aktuellen Voxelkoordinaten addieren kann, um die Koordinaten des Nachbarn zu erhalten (`diff`). Des Weiteren kann man von einem Nachbarn direkt zum nächsten kommen, auch hierfür werden die Differenzvektoren zur Verfügung gestellt (`relativeDiff`). Eine weitere Möglichkeit ist es, aus einem Differenzvektor die Richtung zu gewinnen, in der ein Voxel liegt, der sich aus der Addition mit dem Differenzvektor ergibt (`code`).

32 Scanline-Algorithmen laufen iterativ, also geordnet, über alle Bildpunkte hinweg, was einen effizienten Speicherzugriff zur Folge hat. Die Effizienz ergibt sich daher, dass der schnelle Systemcache optimal ausgenutzt wird, was bei einem wahlfreien Zugriff nicht gewährleistet ist.

Mit all diesen Informationen können von einem Voxel aus, dessen Nachbarvoxel betrachtet werden. Dabei spielt es keine Rolle, ob sich der Voxel an einem Rand befindet oder nicht, da eine komplette Ausnahmeregelung für diese Fälle vorhanden ist. Der Benutzer dieser Nachbarschaft muss sich darum also nicht mehr kümmern und kann alle Voxel in seinem Volumen gleich behandeln.

4.2.2 NeighborCode3DTwentySix

Die hier betrachtete Nachbarschaftskodierung stellt alle für die 26er-Nachbarschaft benötigten Eigenschaften zur Verfügung. Der Aufbau orientiert sich an dem des NeighborCode3DSix. Dies ist sinnvoll, damit bei der Benutzung beide Nachbarschaften problemlos gegeneinander ausgetauscht werden können.

In diesem Fall müssen entsprechend 26 Richtungen definiert werden, in denen ein Voxel einen Nachbarn haben kann. Einige dieser Richtungen sind beispielsweise:

- *InFrontNorthWest=0*
- *InFrontNorth=1*
- *InFrontNorthEast=2*
- *InFrontWest=3*
- *InFront=4*
- ...
- *BehindSouthEast=25*

Alle diese Richtungen befinden sich in einer festen und möglichst sinnvollen Reihenfolge. Eine komplette Übersicht liefert die Tabelle „Richtungen im NeighborCode3DTwentySix“, in der auch die Bit-Codes zu den Richtungen angegeben sind, sowie die Differenzvektoren zum Zentrumsvoxel (siehe Anhang A). Das Prinzip der Anordnung der Richtungen ergibt sich, wenn man sich um den betrachteten Voxel alle 26 benachbarten Voxel vorstellt und dann diese entsprechend ihrer Koordinaten abläuft. So wird zuerst in X-Richtung gelaufen, dann in Y-Richtung und schließlich in Z-Richtung. Wie in Abbildung 33 leicht nachprüfbar, ist `InFrontNorthWest` die erste Nachbarrichtung und `BehindSouthEast` die letzte Nachbarrichtung. Abbildung 33 zeigt die Umgebung eines Voxels (in grauer Farbe) an, die Gruppen von je neun Voxeln liegen im Volumen jeweils hintereinander.

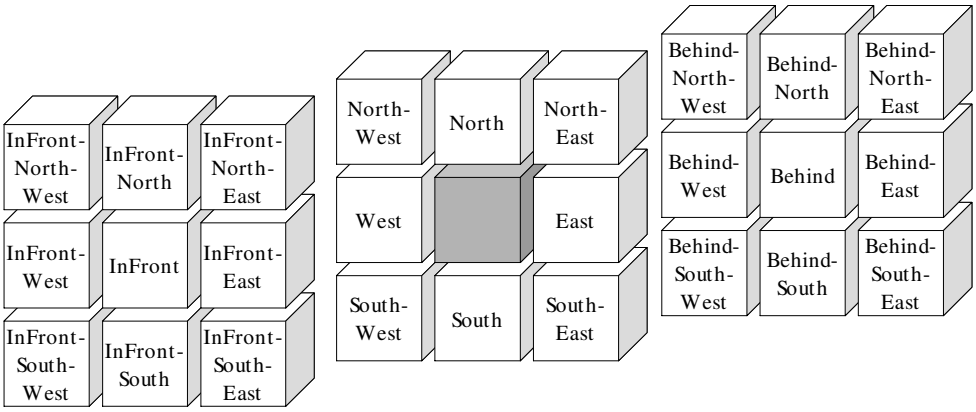


Abbildung 33: Übersicht über die Nachbarn des NeighborCode3DTwentySix (schichtenweise)

Wie bereits für den NeighborCode3DSix beschrieben, gibt es auch hier wieder die Grenzen für das Traversieren der Richtungen. `Error=-1` bleibt wie oben die untere Grenze. Die obere Grenze bildet diesmal `DirectionCount=26`. Zudem werden wieder die Grenzen für die kausalen und anti-kausalen Richtungen zur Verfügung gestellt (`CausalFirst=InFrontNorthWest` bis `CausalLast=West` beziehungsweise `AntiCausalFirst=BehindSouthEast` bis `AntiCausalLast=East`). Die Werte zur Berechnung der entgegengesetzten Richtung betragen diesmal `OppositeDirPrefix=-1` und `OppositeOffset=25`.

Alles Weitere erfolgt analog zu den beschriebenen Vorgehensweisen aus Abschnitt 4.2.1. Natürlich sind bei der 26er-Nachbarschaft deutlich mehr Fälle, zum Beispiel bei der Randbehandlung, zu betrachten. Allerdings ist dies zusammen mit der Tatsache, dass es jetzt deutlich mehr Nachbarn gibt, auch schon der einzige Unterschied zur 6er-Nachbarschaft. Letztlich unterscheidet sich das Traversieren über die Nachbarn nicht.

4.3 Implementation der Nachbarschaftstraverser

Die Nachbarschaftstraverser, die in diesem Unterkapitel vorgestellt werden, erlauben, unter Angabe einer oben beschriebenen Nachbarschaftskodierung, das Traversieren über die Nachbarn von Voxeln.

Die Angabe einer beliebigen kodierten Nachbarschaft ist möglich, da es sich bei den Traverser-Klassen um Template-Klassen handelt, die als Parameter die gewünschte Nachbarschaftskodierung übernehmen. Noch einmal zusammengefasst liefert eine solche Kodierung folgende Informationen:

- wie viele Nachbarn ein Voxel hat,
- in welcher Richtung ein Nachbar liegt und
- wie der Differenzvektor zu einem Nachbarn lautet.

4.3.1 NeighborOffsetTraverser

Die Klasse NeighborOffsetTraverser stellt eine Hilfsklasse für den NeighborhoodTraverser und den RestrictedNeighborhoodTraverser dar.

Da der NeighborOffsetTraverser nichts über die Voxel in einem beliebigen Volumen weiß, wird er nicht direkt zum Traversieren über Voxel benutzt, sondern dient vielmehr als eine Basis- oder Hilfsklasse für einen NeighborhoodTraverser, der zu einem Volumen definiert wird. Die NeighborhoodTraverser dieser Kapselung werden in den nächsten beiden Abschnitten beschrieben. Bei der Erstellung des NeighborOffsetTraversers ist daher nur die Angabe einer Nachbarschaftskodierung notwendig. Diese Angabe erfolgt über den Template-Parameter der Klasse.

In der folgenden Tabelle werden alle für diesen NeighborOffsetTraverser zur Verfügung gestellten Operatoren vorgestellt.

Tabelle 4.3.1: Operatoren der Klasse NeighborOffsetTraverser

`operator++()`
: setzt die aktuelle Richtung auf die nächste folgende. Welche Richtung dies ist, hängt von der Nachbarschaftskodierung ab.

`operator--()`
: setzt die aktuelle Richtung auf die vorherige. Welche Richtung dies ist, hängt von der Nachbarschaftskodierung ab.

`operator+=(difference_type)`
: ermöglicht die Addition auf eine Richtung (zum Beispiel einen Integer-Wert oder eine Richtung). Das Ergebnis ist eine Richtung, die um den übergebenen Wert der aktuellen Richtung folgt.

`operator-=(difference_type)`
: ermöglicht die Subtraktion von einer Richtung (zum Beispiel einen Integer-Wert oder eine Richtung). Das Ergebnis ist eine Richtung, die um den übergebenen Wert der aktuellen Richtung vorherging.

`operator+(difference_type)`
: ist der Additions-Operator für Richtungen.

`operator-(difference_type)`
: ist der Subtraktions-Operator für Richtungen.

`operator==(NeighborOffsetTraverser)`
: überprüft zwei Richtungen auf Gleichheit.

`operator!=(NeighborOffsetTraverser)`
: überprüft zwei Richtungen auf Ungleichheit.

Die nächste Tabelle beschreibt kurz die wichtigsten zur Verfügung gestellten Operationen.

Tabelle 4.3.2: Schnittstellen der Klasse NEIGHBOROFFSETTRAVERSER

`turnRound()`
: setzt die aktuelle Richtung auf die entgegengesetzte.

`turnTo(Direction)`
: setzt die aktuelle Richtung auf die übergebene.

`Diff3D diff()`
: liefert den Differenzvektor zwischen aktuellem Voxel und dem Nachbarvoxel der aktuellen Richtung zurück.

`Diff3D diff(Direction)`
: liefert den Differenzvektor zwischen aktuellem Voxel und dem Nachbarvoxel der übergebenen Richtung zurück.

`Diff3D relativeDiff(difference_type)`
: liefert den Differenzvektor zwischen aktuellem Nachbarvoxel und dem Nachbarvoxel in der Richtung, die dem übergebenen Offset plus der aktuellen Richtung entspricht, zurück.

`int dX()`
: liefert nur die x-Komponente des Differenzvektors zurück.

`int dY()`
: liefert nur die y-Komponente des Differenzvektors zurück.

`int dZ()`
: liefert nur die z-Komponente des Differenzvektors zurück.

`bool isDiagonal()`
: gibt an, ob die aktuelle Richtung auf einen nicht flächenverbundenen Voxel zeigt (bei 6er-Nachbarschaft immer `false`).

`Direction direction()`
: liefert die aktuelle Richtung zurück.

`unsigned int directionBit()`
: liefert die aktuelle Richtung in Bit-Form zurück.

`Direction opposite()`
: liefert die entgegengesetzte Richtung der aktuellen zurück.

`unsigned int oppositeDirectionBit()`
liefert die entgegengesetzte Richtung der aktuellen in Bit-Form zurück.

`Direction direction(difference_type)`
liefert die Richtung, die der Summe des übergebenen Offsets und der aktuellen Richtung entspricht, zurück.

4.3.2 NeighborhoodTraverser

Der NeighborhoodTraverser ist ebenfalls als Template-Klasse beschrieben. Somit ist er für beliebige Voxelnachbarschaften anwendbar. Des Weiteren muss noch eine Klasse angegeben werden, die das Iterieren über ein Volumen übernimmt. Somit werden beim Erstellen des NeighborhoodTraversers ein VolumeIterator und eine Nachbarschaftskodierung benötigt. Als Nachbarschaftskodierung lassen sich die beiden oben beschriebenen Varianten verwenden. Als VolumeIterator eignet sich hingegen der MultiIterator[33] der VIGRA-Bibliothek. Es ist zu beachten, dass dem MultiIterator die „Shape“ (Vektor mit der Größe eines Bildes) eines Volumens übergeben wird. Der Unterschied zum NeighborOffsetTraverser ist, dass dieser Traverser nicht über Richtungen läuft, sondern wirklich um Positionen (Voxel) in einem Volumen. Dazu benötigt er natürlich den NeighborOffsetTraverser. Der NeighborhoodTraverser zeigt also explizit auf einen Voxel im Volumen, indem er den Iterator über das Volumen verwendet.

Es ist mit seiner Hilfe beispielsweise möglich, alle Nachbarvoxel zu einem Voxel in der Reihenfolge der Richtungen abzulaufen. Da sich der Aufbau allerdings nicht sehr stark zu dem des NeighborOffsetTraversers unterscheidet, werden hier nur noch die wichtigsten Funktionen kurz vorgestellt:

Tabelle 4.3.3: Schnittstellen der Klasse NeighborhoodTraverser

`turnTo(Direction)`
setzt die Richtung des Traversers auf die angegebene Richtung. Der Voxel in dieser Richtung wird der neue aktuelle Nachbar.

`turnRound()`
setzt die Richtung des Traversers auf die entgegengesetzte Richtung. Der Voxel in dieser Richtung wird der neue aktuelle Nachbar.

`moveCenterToNeighbor()`
bewegt das Zentrum in die aktuelle Richtung. Der Nachbar in dieser Richtung wird das neue Zentrum. Die Richtung wird dabei nicht verändert.

`swapCenterNeighbor()`
bewegt das Zentrum in die aktuelle Richtung. Der Nachbar in dieser Richtung wird das neue Zentrum. Die Richtung wird dabei in die gegensätzliche Richtung gedreht, so dass sie zum alten Zentrum zeigt.

`Direction direction()`
liefert die aktuelle Richtung zurück.

33 Eine Dokumentation des MultiIterators findet sich in [Vig06c].

`Diff3D diff()`
liefert die Differenz zum aktuellen Nachbarn zurück.

4.3.3 RestrictedNeighborhoodTraverser

Der NeighborhoodTraverser des letzten Unterkapitels erlaubt nur die Behandlung von Voxeln, die nicht am Rand des Volumens liegen. In vielen Fällen reicht er aus, zum Beispiel wenn man das zu bearbeitende Volumen in ein größeres Volumen einbettet und annimmt, dass es sich in einem unendlichen Universum befindet. Möchte man allerdings auf dem eigentlichen Volumen in seiner wahren Größe arbeiten, so sind die Randvoxel speziell zu behandeln, da diese nicht in allen Richtungen Nachbarvoxel besitzen. Ansonsten ist die Arbeitsweise dieses Traversers nicht anders. Beim Traversieren werden nur die Nachbarn beachtet, die auch wirklich Voxel in dem betrachteten Volumen sind.

Mit Hilfe des in diesem Kapitel beschriebenen Umgangs mit Voxeln und ihrer Nachbarn, ist es möglich, die Anwendungen, welche in den folgenden Kapiteln beschrieben werden, umzusetzen.

Da die beschriebene Art der Kapselung einer Nachbarschaft aus dem zweidimensionalen Fall übernommen wurde, mag man sich fragen, ob sich die Vorgehensweise auch im dreidimensionalen eignet. Im Gegensatz zur zweidimensionalen 8er-Nachbarschaft, stellt die Implementation einer dreidimensionalen 26er-Nachbarschaft einen erheblichen Mehraufwand dar. Allerdings ist diese Art der Umsetzung mit einigen Vorteilen verbunden. Die bestehenden Schnittstellen der PixelNeighborhood decken die Anforderungen an eine Nachbarschaft für die Bildverarbeitung ab, und sind somit auch ein guter Ansatz für die Entwicklung der Schnittstellen der VoxelNeighborhood. Anwender, die bereits eine der beiden Nachbarschaften benutzt haben, müssen sich bei der Benutzung der anderen nicht umgewöhnen. Zudem können die implementierten Traverser in vielen Algorithmen intuitiv eingesetzt werden.

Im Anschluss an die hier vorgestellte Implementation können die zu programmierenden Algorithmen stark vereinfacht und generisch umgesetzt werden. Eine sonst notwendige zusätzliche Betrachtung von Nachbarschaften und auch der Randbehandlung entfällt somit für alle weiteren Algorithmen, die auf diesen Nachbarschaftskodierungen aufbauen. Schon im nächsten Kapitel wird deutlich, wie die generische Umsetzung eines Wasserscheiden-Verfahrens ohne Einschränkung auf eine der Nachbarschaften möglich wird.

5 Segmentierungsverfahren

Der erste Schritt in der Bildanalyse besteht im Allgemeinen darin, die aufgenommenen und digitalisierten Bilddaten in „bedeutungshaltige Regionen“ zu unterteilen. Zu diesem Zweck wurde eine Vielzahl von Segmentierungsverfahren entwickelt. Mittlerweile gibt es nahezu für jede spezielle Segmentierungsaufgabe auch spezielle Algorithmen.

Dieses Kapitel erhebt keinen Anspruch darauf, in der Vollständigkeit alle Verfahren aufzuzählen, dies würde den Umfang der Arbeit übersteigen. Es wird vielmehr ein allgemeiner Überblick über die verschiedenen Klassen von Algorithmen gegeben, bevor eine spezielle Teilklasse von Verfahren – die Wasserscheiden-Verfahren – herausgegriffen und erläutert werden. Die Spezialisierung auf diese Teilklasse ergibt sich aus den Anforderungen an Segmentierungsalgorithmen, die im Rahmen dieser Arbeit vorliegen. Sie werden bei der Auswahl des Verfahrens in Unterkapitel 5.3 näher erläutert.

Unter einem Segmentierungsverfahren versteht man ein Verfahren, das, auf ein Ausgangsbild angewandt, dieses in zusammenhängende Komponenten, auch Regionen genannt, unterteilt. Diese Komponenten sind im einfachsten Fall Gruppen von Pixeln mit ähnlichen Farbwerten (siehe 5.1.1), können aber bei modellbasierten Verfahren (siehe 5.1.2) im Idealfall auch die einzelnen Objekte oder Akteure der aufgenommenen Szene repräsentieren.

Die Bildsegmentierung gehört zu den anspruchsvollsten Aufgaben in der Bildverarbeitung, denn leider ist meist nicht bekannt, wann eine segmentierte Pixelgruppe wirklich ein Objekt der Szene repräsentiert. Ein gängiger Ansatz ist, dass zunächst mit einem übersegmentierten[34] Bild begonnen wird, welches im Anschluss interaktiv bearbeitet wird. Dabei werden Regionen gleicher Objekte solange verschmolzen, bis jedes Objekt aus lediglich einer Region besteht (vgl. [GW92], S.461ff).

34 Ein übersegmentiertes Bild liegt vor, wenn durch ein Segmentierungsverfahren ein Bild in mehr Regionen zerlegt wird, als vom Benutzer gewünscht.

5.1 Übersicht über die Verfahren

In der Bildverarbeitung gibt es zwar viele unterschiedliche Verfahren zur Bildsegmentierung, allerdings sind die meisten von ihnen im Kontext dieser Arbeit zu speziell an die Anwendungsdomäne gebunden. Diese spielen im Rahmen dieser Arbeit keine Rolle, da an die Domäne der Eingangsdaten keine besonderen Anforderungen gestellt werden.

Die folgenden Unterkapitel sollen einen Einstieg in das interessante Feld der Bildsegmentierung ermöglichen, stellen jedoch nur die wichtigsten Klassen der Verfahren dar. Wie schon in Kapitel 1 erwähnt, ist die Bildsegmentierung zwar ein wichtiger Vorverarbeitungsschritt dieser Arbeit, dennoch ruht das Hauptaugenmerk nicht auf dieser, so dass eine kurze Einführung an dieser Stelle ausreichend ist.

Für eine weiter gefasste Übersicht über die Verfahrensklassen und auch die Verfahren selbst wird an dieser Stelle auf die entsprechenden Kapitel in [Jäh97] und [GW92] verwiesen.

5.1.1 Elementare Segmentierungsverfahren

Unter den elementaren Segmentierungsverfahren werden diejenigen verstanden, die nur auf lokalen Informationen innerhalb eines Bildes aufbauen. Dazu zählen unter anderem die Bildpunkt-basierten Methoden, welche nur von den Grauwerten jedes einzelnen Bildpunktes ausgehen, wie zum Beispiel das Thresholding- oder Schwellwert-Verfahren, aber auch regionenorientierte Verfahren, welche die Grauwerte von Regionen analysieren. Des Weiteren zählen auch kantenbasierte Methoden dazu, sie basieren auf Kantenerkennung und Kantenverfolgung.

5.1.1.1 Bildpunkt-basierte Verfahren

Die Bildpunkt-basierten Verfahren stellen vom Ansatz her die einfachste und intuitivste Methode der Bildsegmentierung dar. Dennoch sollte man sich von der Einfachheit dieser Methode nicht täuschen lassen, sie ist keinesfalls von vornherein auszuschließen. Oftmals bietet sich eine solche Methode sogar als Vorverarbeitungsschritt für andere Verfahren an. Ist ein Bild beispielsweise nicht genügend beleuchtet, so kann es als erster Schritt der Segmentierung sehr sinnvoll sein, die fehlerhafte Beleuchtung zu korrigieren.

Wenn ein ideales Merkmal zur Abgrenzung der zu segmentierenden Objekte vom Hintergrund gefunden wurde, so zeigt das Grauwert-Histogramm eine Verteilung mit zwei voneinander getrennten Maxima[35]. Ein Maximum repräsentiert dann die zu segmentierenden Objekte, das andere den Hintergrund. Zwischen diesen beiden Maxima befindet sich im Idealfall ein Minimum bei null, also ein Bereich in dem das Histogramm keine Merkmale besitzt.

35 Die Spitzen dieser Maxima werden auch „Peaks“ genannt.

Die eigentliche Segmentierung liegt nun darin, mit Hilfe eines sogenannten *Schwellwertverfahrens*[36] eine Schwelle im Merkmalraum[37] festzulegen. Alle Bildpunkte, die oberhalb dieses Schwellwertes liegen, werden daraufhin in ihrer Intensität auf das Maximum erhöht, während die Bildpunkte mit niedrigerer Intensität auf das Minimum abgesenkt werden. Auf diese Weise wird ein Binärbild erzeugt, welches das Ergebnis der Segmentierung darstellt. Dieser Vorgang wird in Abbildung 34 anhand eines zweidimensionalen Ausgangsbildes veranschaulicht.

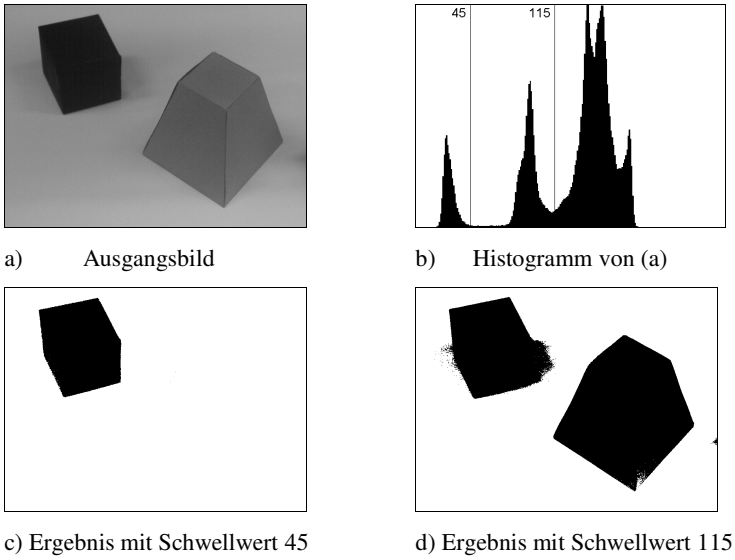


Abbildung 34: Anschauliche Arbeitsweise eines Schwellwertverfahrens: Für ein Ausgangsbild (a) wird anhand dessen Histogramms (b) ein Schwellwert gewählt, der entweder eins der beiden Objekte (c) oder beide Objekte von Hintergrund trennt (d).

Die oben beschriebene Variante des Schwellwertverfahrens mit einem globalen Schwellwert lässt sich, aufgrund der Tatsache, dass nur ein Schwellwert für ein Bild zugelassen wird, im Allgemeinen nur schwer auf Bilder der realen Welt anwenden. Zu Problemen kommt es vor allem, wenn die Objekte mit einer Lichtquelle beleuchtet wurden, welche zu einem Intensitätsverlauf in dem Aufnahmebild führt. Aber auch Schatten von Objekten führen zu Problemen bei der Segmentierung. Sollen zum Beispiel beide Objekte aus Abbildung 34 a) segmentiert werden, so wird nicht nur der Würfel im oberen linken Viertel des Bildes, sondern auch dessen Schatten segmentiert (siehe Abbildung 34 d).

Hieraus wird leicht ersichtlich, dass ein globaler Schwellwert nicht das adäquate Mittel ist, um einer Segmentierung dienlich zu sein. Es müssten zunächst die durch die Lichtquelle verursachten Farbverläufe und Schatten korrigiert werden, um gute Ergebnisse zu erhalten. Dennoch können Schwellwertverfahren nahezu beliebig komplex

36 In der englischsprachigen Literatur werden diese Verfahren auch „Thresholding“ genannt.
37 In diesem Fall entspricht der Merkmalraum dem Histogramm des Bildes.

erweitert werden. So finden sich beispielsweise in [GW92] auf Seite 443ff folgende Ansätze der Erweiterung von Schwellwertverfahren:

- Erweiterung auf mehrere Schwellwerte
- Schwellwert-Auswahl aufgrund der Grenzlinien-Charakteristik des Merkmalraumes
- Schwellwert basierend auf mehreren Bild-Kanälen (zum Beispiel: RGB-Farbraum)

Diese Ansätze werden hier nur aufgezählt, nicht aber näher erläutert, da sie den Umfang dieses Kapitels übersteigen würden. Außerdem sind sie für eine Einführung in die Bildpunkt-basierte Segmentierung als zu speziell anzusiedeln.

5.1.1.2 Regionenorientierte Verfahren

Wie im vorigen Abschnitt ersichtlich, eignen sich Bildpunkt-basierte Verfahren meist nur für sehr spezielle Anwendungen. Dies liegt darin begründet, dass jeder Bildpunkt einzig auf Basis seines Grauwertes zugeordnet wird. Diese Zuordnung geschieht also auch unabhängig davon, welche Werte seine Nachbarn (siehe Kapitel 3.2) besitzen. Aus diesem Grund können einzelne isolierte Bildpunkte im Ergebnis der Segmentierung sehr oft auftreten. Dies widerspricht dem entscheidenden Merkmal von Objekten, das meist in dessen Zusammenhang begründet liegt. Aus diesem Grund betrachten regionenorientierte Verfahren nicht das Bild selbst als Merkmalraum, sondern bereits eine Nachbarschaft von Bildpunkten, deren Größe durch den für das Verfahren verwendeten Operator bestimmt werden kann.

Wichtige Verfahren sind hier:

- Split-and-Merge-Algorithmen

  Sie betrachten das Ausgangsbild zunächst als eine Region, die durch unterteilende Vorgänge (*split*) nach und nach in mehrere Segmente unterteilt wird. Je nach Verfahren werden diese Segmente anhand eines Kriteriums in einem verschmelzenden Schritt (*merge*) wieder zu größeren Einheiten zusammengefügt. Eine gute Übersicht über diese Klasse von Verfahren findet sich beispielsweise in [RK82].

- Regionenwachstumsverfahren

  Sie gehen von automatisch erzeugten oder manuell ausgewählten Punkten im Bild aus, um von dort zugehörige Regionen zu expandieren. Wasserscheiden-Verfahren zählen je nach Sichtweise ebenfalls dazu Zur Übersicht und Vertiefung der einzelnen Verfahren bieten sich [RK82] und [GW92] S. 458f, an.

- Pyramid-Linking

  Bei diesem Verfahren wird von einer Auflösungspyramide (zum Beispiel Gauß-Pyramide) ausgegangen. In jeder Iteration des Algorithmus' werden die Knoten jeweils dem Vaterknoten mit dem ähnlichsten (Grau-)Wert zugeordnet, der Baum neu berechnet, und Knoten ohne Blätter werden eliminiert. Dies geschieht

solange, bis Iterationen die Pyramide nicht mehr verändern. Eine gute Darstellung findet sich in [Jäh97] auf S. 487ff.

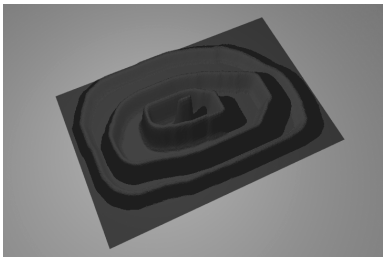
Da die Vorstellung der einzelnen regionenorientierten Verfahren den Umfang dieses Kapitels überschreiten würde und sie im Folgenden kaum mehr eine Rolle spielen, sei hier auf die entsprechenden Literaturangaben verwiesen, wo die einzelnen Verfahren im Detail erläutert werden.

5.1.1.3 Kantenbasierte Verfahren

In den vorangegangenen zwei Abschnitten wurden Verfahren vorgestellt, die Objekte aufgrund lokaler Merkmale oder Merkmale einer Nachbarschaft klassifizieren beziehungsweise segmentieren. Manchmal ist es aber gar nicht möglich, mit diesen elementaren Verfahren zu arbeiten, weil das Bildmaterial dafür nicht geeignet ist. Dennoch muss in diesen Fällen nicht zu Methoden der meist komplexeren regionenorientierten Segmentierung übergegangen werden.

Stattdessen kann die kantenbasierte Segmentierung angewendet werden. Sie liegt darin begründet, dass sich Kanten dort im Bild befinden, wo die partielle Ableitung erster Ordnung einen Maximalwert erreicht oder die Ableitung zweiter Ordnung einen Nulldurchgang aufweist (vgl. [Jäh97], S. 490). Es müssen also lediglich solche interessanten Punkte[38] im Bild gefunden werden, von denen aus dann ein *Konturverfolgungsalgorithmus* gestartet wird. Dieser läuft auf dem lokalen Maximum des Grauwertgradienten an der Kante entlang. Durch diese anschauliche Beschreibung wird schon deutlich, dass es sich bei kantenbasierten Verfahren im Allgemeinen um sequentielle Algorithmen handelt.

a) Graustufenbild



b) Graustufengebirge von (a)

Abbildung 35: Beispiel der Erzeugung eines Graustufengebirges (b) aus einem Graustufenbild (a). Die Bildwerte aus (a) werden dabei als Höhen des Gebirges (b) interpretiert (je heller, desto höher).

Es gibt aber darüber hinaus noch eine weitere interessante Sichtweise auf kantenbasierte Verfahren, die sogenannten *Wasserscheiden-Verfahren*. Bei diesen Verfahren wird ein Graustufenbild als Graustufengebirge aufgefasst (siehe Abbildung 35), dessen lokale Minima sogenannte Bassins bilden, die anschaulich mit Wasser geflutet werden. Dort, wo während des Flutungsvorgangs zwei Wassermassen aufeinander treffen, entsteht eine Wasserscheide, die die beiden Regionen voneinander trennt. Es gibt eine Vielzahl von

38 Sie werden in der englischsprachigen Literatur häufig „POI“ (Points of Interest) genannt.

Methoden, die dieses Fluten simulieren. Da in dieser Arbeit ebenfalls ein Wasserscheiden-Algorithmus verwendet wird, werden in Kapitel 5.2 die einzelnen Algorithmen kurz vorgestellt. Eines jedoch haben alle Wasserscheiden-Algorithmen gemeinsam. Sie liefern alle zusammenhängende Regionen als Ergebnis der Segmentierung, was bei kantenbasierten Verfahren im Allgemeinen nicht gegeben sein muss.

5.1.2 Modellbasierte Segmentierungsverfahren

Während der Ansatz der elementaren Segmentierungsverfahren den klassischen Bottom-Up-Ansatz darstellt, gibt es oft auch ein nicht unerhebliches Vorwissen über die zu segmentierenden Objekte. Auf dieses wurde bei den obigen Verfahren allerdings keine Rücksicht genommen. Doch gerade dieses Vorwissen kann man sich so zu nutze machen, dass es die Segmentierung positiv vorantreibt. Der Name „Modellbasierte Segmentierung“ leitet sich aus der Tatsache ab, dass hier meist ein Modell oder Prototyp der zu erwartenden Objekte zugrunde gelegt wird.

Die einfachste Möglichkeit der modellbasierten Segmentierung besteht im Musterabgleich[39] der Bilddaten mit dem Modell. Dabei wird im einfachsten Fall eine Schablone des Modells über die Bilddaten geschoben, wobei für jeden Bildpunkt die Überdeckung mit der Schablone bestimmt wird. Ab eines festzulegenden Überdeckungsgrades gilt das Modell als erkannt. Darauf aufbauend werden diese Verfahren meist für die jeweilige Segmentierungsaufgabe erweitert. Dies kann zum Beispiel dadurch geschehen, dass man die Vorkommen des Modells in den Bilddaten rotations- oder skalierungsinvariant zulässt.

Eine weitere Methode der modellbasierten Segmentierung stellt die Transformation der Bilddaten in einen Modellraum dar. Diese Transformation wird auch als *Hough-Transformation* bezeichnet.

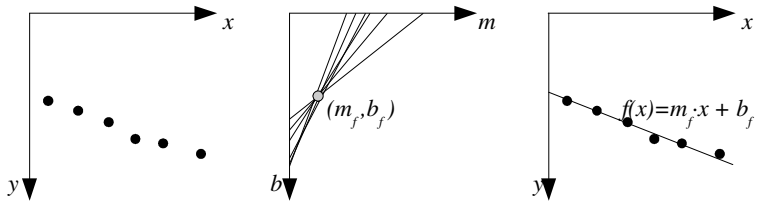


Abbildung 36: Beispiel: Erkennung von Linien mittels Hough-Transformation. Links: Die Bildpunkte, zu der eine Gerade gesucht wird. Mitte: Jeder Punkt wird im Parameterraum (m=Steigung, b=Ordinatenabschnitt) in eine Gerade transformiert. Das Maximum der Schnittpunkte entspricht den Parametern der gesuchten Geraden. Rechts: Die resultierende Gerade im ursprünglichen Koordinatensystem.

An diesem Beispiel lässt sich erkennen, dass die modellbasierte Segmentierung nicht nur in der Lage ist, Objekte zu erkennen, vielmehr lässt sich mit ihr auch die Rekonstruktion

39 Dieser wird im englischen und deutschen Sprachgebrauch auch als „Pattern Matching“ bezeichnet.

fehlender Informationen[40] bewerkstelligen. Dies kann zum Beispiel dadurch geschehen, dass die erkannte Linie in das Bild integriert wird, um fehlende Pixel zu interpolieren und so eine eventuell getrennte Kontur wieder zu verbinden.

Als letzter hier vorgestellter Ansatz sollen noch die so genannten *Active Contours* Erwähnung finden. Hier wird durch sukzessive Verformung einer Ausgangskontur durch mehrere wirkende Energien eine Annäherung an die Bilddaten erreicht. Dieses geschieht solange, bis die an der Kontur wirkenden Energien minimal sind. Am Ende entspricht die segmentierte Kontur idealerweise der wirklichen Kontur des Objektes. Leider ist eine Vielzahl der Verfahren sehr rechenintensiv, so dass sie meist lediglich für zweidimensionale Anwendungen eingesetzt werden. Eine Ausnahme bildet hier der auch in drei Dimensionen recht schnelle „Dual-Simplex Meshes"-Algorithmus von Svoboda und Matula (vgl. [SM01]). Einen anwendungsorientierten Überblick über dieses Verfahren findet man in [SH04].

40 Diese Rekonstruktion wird in der Literatur auch Regularisierung genannt, zum Beispiel bei [Jäh97], S. 494.

5.2 Wasserscheiden zur Volumensegmentierung

Im Umfeld der regionenorientierten Segmentierungsverfahren gelten die Wasserscheiden-Verfahren zur Zeit als Mittel der Wahl für eine Bildsegmentierung. Dies hat unter anderem folgende Gründe:

- Die Verfahren bauen auf bekannten Definitionen auf. Diese Definitionen beschreiben das zu erwartende Ergebnis der Segmentierung, gegen das getestet werden kann.
- Die Verfahren sind recht effizient, und das sowohl in Zeit- als auch in Speicherkomplexität. Damit eignen sich einige Verfahren sogar für die Anwendung auf dreidimensionalen Bilddaten.

Die Wasserscheiden-Verfahren wurden ursprünglich von Digabel und Lantuéjoul (siehe [Lan78]) vorgestellt, und später von Beucher und Lantuéjoul (siehe [BL79]) verbessert. Wie bereits in Abschnitt 5.1.1.3 erwähnt, lässt sich das Wasserscheiden-Verfahren recht intuitiv herleiten, wenn man eine Analogie aus der Geographie heranzieht. Eine Landschaft, welche in diesem Kontext auch als topografisches Relief (siehe Abbildung 35) bezeichnet wird, wird von oben mit Wassertropfen beregnet. Dabei sind die Wasserscheiden diejenigen Linien, welche die Domänen der Anziehung von Regentropfen abgrenzen.

Eine andere Sichtweise auf die Wasserscheiden ist, sich die Landschaft in einen See eingetaucht vorzustellen. Vor dem Eintauchen werden in die tiefsten Stellen Löcher gebohrt, sodass durch diese Wasser eintreten kann. Nach dem Eintauchen füllen sich nach und nach die einzelnen Bassins[41] rund um die Minima mit Wasser. Im Laufe des Eintauchens kommt es immer wieder vor, dass sich zwei Wassermassen aus unterschiedlichen Bassins berühren. Immer wenn dies der Fall ist, wird ein Damm aufgebaut. Wenn der Wasserstand den höchsten möglichen Punkt der Landschaft erreicht hat, ist diese komplett geflutet worden, und das Verfahren bricht ab. Am Ende sind die Bilddaten durch die Dämme in Regionen unterteilt, und die Dämme repräsentieren die Wasserscheiden.

Im Folgenden sei noch erwähnt, dass meist nicht das Graustufenbild selbst, sondern dessen Gradientenbetragsbild (siehe 5.3) für die Wasserscheiden-Verfahren herangezogen wird. Dies führt dazu, dass die Wasserscheiden dort entstehen, wo sich die größten Grauwertunterschiede[42] im Bild befinden, so wie es in der Bildverarbeitung gewünscht wird.

Wasserscheiden-Verfahren sind allerdings keinesfalls als Allheilmittel der Bildsegmentierung anzusehen, denn sie bringen auch Probleme mit sich. Bilder werden meist stark übersegmentiert, so dass eine nachfolgende Verschmelzung von Regionen mittels zu wählender Homogenitätskriterien unvermeidbar ist. Außerdem ist durch den „Boom“, den die Wasserscheiden in den letzten zehn Jahren erfahren haben, eine Menge Verwirrung um die Resultate aus den einzelnen entwickelten Algorithmen entstanden. Es

41 In der englischsprachigen Literatur auch Catchment-Bassins genannt, und häufig mit *CB* abgekürzt.

42 Die größten Grauwertunterschiede stellen die markantesten Kanten des Bildes dar.

wurde nicht aus jeder wissenschaftlichen Arbeit ersichtlich, welche Definition der Wasserscheiden verwendet wurde. Stattdessen nahm die formale Definition meist Formen einer Spezifikation eines Algorithmus' ein. So wurde es unmöglich, Aussagen über die zugesicherten Eigenschaften der Algorithmen, wie zum Beispiel deren Korrektheit in Bezug auf eine mathematische Definition, zu treffen.

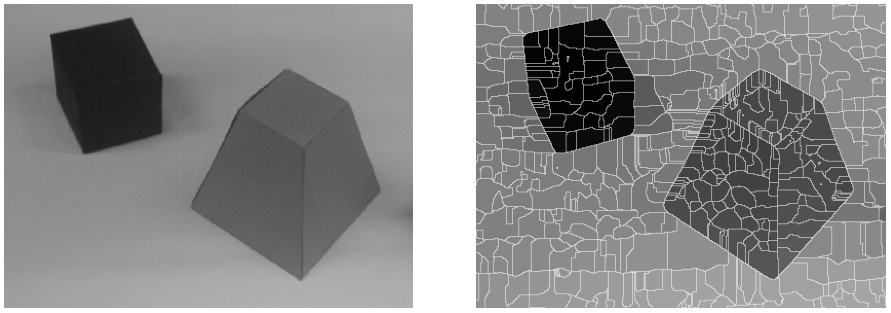


Abbildung 37: Beispielhaftes Ergebnis eines Wasserscheiden-Verfahrens. Links: Das Ausgangsbild, rechts: Das Crack-Edge-Bild des Segmentierungsergebnisses. Die einzelnen Regionen wurden mit dem Mittelwert aller Pixel des Ausgangsbildes gefüllt.

Weiterhin muss an dieser Stelle noch angemerkt werden, dass nicht jeder Algorithmus auch Wasserscheiden-Bildpunkte berechnet. Einige Algorithmen berechnen lediglich die einzelnen Regionenzugehörigkeiten, die Wasserscheiden sind dann implizit durch weitere Bildrepräsentationen, wie zum Beispiel Crack-Edge-Bilder (vgl. Abschnitt 3.2.1) gegeben. Dies hat in der Praxis den Vorteil, dass das Entstehen von „dicken“ Wasserscheiden nicht vorkommt.

Nach dieser informellen Einführung in das Gebiet der Wasserscheiden-Verfahren werden zunächst die wichtigsten Definitionen eingeführt. Diese Definitionen orientieren sich, soweit nicht anders angegeben, an [Meij05]. Dabei wird vom kontinuierlichen Fall ausgegangen und darauf aufbauend der diskrete Fall vorgestellt. Nach den Definitionen werden zwei unterschiedliche Verfahren vorgestellt, welche auf der diskreten Wasserscheiden-Transformation aufbauen und in der Praxis am häufigsten verwendet werden.

5.2.1 Die kontinuierliche Wasserscheiden-Transformation

Die kontinuierliche Wasserscheiden-Transformation basiert auf der Verwendung von Distanzfunktionen. Es gibt im allgemeinen Kontext der Bildverarbeitung einen ganzen „Zoo von Distanzfunktionen“, jedoch wird sich im speziellen Fall der Modellierung von Wasserscheiden meist auf die topographische Distanz beschränkt, da sie das intuitive Verständnis der Wasserscheiden (vgl. 5.2) sehr gut nachbildet.[43] Dafür wird die Grundvoraussetzung gewählt, dass das Bild f aus einem kontinuierlichen Funktionenraum stammt, dessen Funktionen zweifach differenzierbar sind und lediglich isolierte kritische Punkte besitzen ($f \in C(D)$).

43 vgl. [Meij05], S. 94f

Definition 5.2.1 (Topographische Distanz) Die topographische Distanz zwischen zwei Bildpunkten *p* und *q* aus *D* ist gegeben durch:

$$T_f(p,q) = \inf_{\gamma} \int_{\gamma} \|\nabla f(\gamma(s))\| \, ds ,$$

wobei das Infimum über alle Pfade $\gamma \in C(D)$ mit $\gamma(0)=p$ und $\gamma(1)=q$ gebildet wird. Der Pfad mit der geringsten T_f-Distanz zwischen *p* und *q* wird auch *Pfad des steilsten Abstiegs* genannt. Diese Pfade sind im Rahmen der Wasserscheiden-Transformation besonders interessant, wie sich im Folgenden zeigen wird.

Definition 5.2.2 (Wasserscheiden-Transformation) Besitze $f \in C(D)$ Minima $\{m_k\}_{k \in I}$ für eine Indexmenge *I*. Das Auffangbecken $CB(m_i)$ eines Minimums m_i ist die Menge von Bildpunkten $x \in D$, die topographisch näher an m_i liegen als an einem beliebigen anderen Minimum m_j:

$$CB(m_i) = \left\{ x \in D \,\middle|\, \forall_{j \in I \setminus \{i\}} f(m_i) + T_f(x, m_i) < f(m_j) + T_f(x, m_j) \right\}$$

Die *Wasserscheide* eines Bildes *f* ist die Menge von denjenigen Punkten, die keinem Auffangbecken zugeordnet werden können:

$$Wshed(f) = D \setminus \left(\bigcup_{i \in I} CB(m_i) \right) .$$

Sei *W* ein beliebiges Label, und es gelte: $W \notin I$. Dann ist die Wasserscheiden-Transformation definiert als:

$\lambda : D \to I \cup \{W\}$, mit:

$$\lambda(p) = \begin{cases} i & falls \;\; p \in CB(m_i) \\ W & falls \;\; p \in Wshed(f) \end{cases} .$$

Zusammenfassend lässt sich sagen, dass die Wasserscheiden-Transformation von *f* zu jedem Bildpunkt aus *D* ein Label zuordnet, so dass folgende Aussagen gültig sind:

1. Die Label sind einzigartig für jedes Auffangbecken.
2. Für jeden Bildpunkt, der einer Wasserscheide zugeordnet wurde, wird ein spezielles Label *W* gesetzt.

5.2.2 Die diskrete Wasserscheiden-Transformation

Bevor mit der algorithmischen Beschreibung der Wasserscheiden im diskreten Fall begonnen wird, muss noch eine Distanz definiert werden, welche besonders wichtig für diese Art der Wasserscheiden-Transformation ist:

Definition 5.2.3 (Geodätische Distanz) Sei $A \subseteq E$, mit $E = \mathbb{R}^d$ oder $E = \mathbb{Z}^d$, und seien *a* und *b* zwei Bildpunkte in *A*. Die *geodätische Distanz* $d_A(a,b)$ zwischen *a* und *b* innerhalb *A* ist die minimale Pfadlänge aller Pfade innerhalb *A* von *a* nach *b*.

Sei $B \subseteq A$, so ist $d_A(a,B) = MIN_{b \in B}(d_A(a,b))$.

Sei B ferner unterteilt in k Zusammenhangskomponenten, $B = B_1 \cup B_2 \cup ... \cup B_k$, so ist die *geodätische Einflusszone* einer Menge B_i innerhalb A definiert als:

$$iz_A(B_i) = \left\{ p \in A \,\middle|\, \forall_{j \in [1,...,k] \setminus \{i\}} d_A(p, B_i) < d_A(p, B_j) \right\}$$

Die Menge $IZ_A(B)$ ist die Vereinigung der geodätischen Einflusszonen der Zusammenhangskomponenten von B, zum Beispiel:

$$IZ_A(B) = \bigcup_{i=1}^{k} iz_A(B_i)$$

Das Komplement der Menge $IZ_A(B)$ innerhalb A wird *Skeleton der Einflusszonen* genannt:

$$SKIZ_A(B) = A \setminus IZ_A(B) .$$

Die Skeleton-Einflusszone besteht aus allen Bildpunkten, die im Sinne der geodätischen Distanz äquidistant zu mindestens zwei nächsten Nachbarn sind. In einem Binärbild f einer Domäne A kann die Skeleton-Einflusszone dadurch definiert werden, dass B mit der Menge der Bildpunkte des Vordergrunds identifiziert wird.

Der Übergang von der kontinuierlichen zur diskreten Wasserscheiden-Transformation gestaltet sich schwieriger, als nach den grundlegenden Definitionen vielleicht erwartet. Ein großes Problem, das während der Quantelung von digitalen Bildern (vgl. Kapitel 3.1) leider sehr häufig entsteht, sind Plateaus (Bildregionen gleichen Grauwertes). Allerdings sei angemerkt, dass die nächsten Definitionen bereits automatische Mechanismen besitzen, um mit Plateaus umzugehen, da sie die Wasserscheiden-Transformation Schritt für Schritt berechnen. Dabei besteht jeder Schritt aus einem Binärbild, für welches eine Skeleton-Einflusszone berechnet wird.

5.2.2.1 Wasserscheiden als Eintauchvorgang

Die ursprüngliche Definition der Wasserscheiden-Transformation als Eintauchvorgang geht auf Vincent und Soille zurück (siehe [VS91]). Die folgende Abbildung verdeutlicht die Vorgehensweise dieser Art von Wasserscheiden-Verfahren.

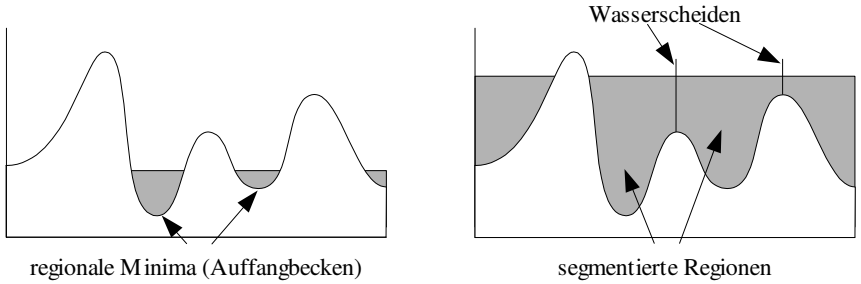


Abbildung 38: Darstellung der Wasserscheiden-Transformation durch Eintauchen. Links: Aus den lokalen Minima werden Auffangbecken. Rechts: Treffen sich zwei Wassermassen unterschiedlicher Auffangbecken, entstehen Wasserscheiden.

Seien h_{min} und h_{max} der minimale beziehungsweise maximale Grauwert eines Bildes $f: D \rightarrow \mathbb{N}$, so wird das Verfahren durch eine Rekursion über einem Grauwert *h* definiert, welcher von h_{min} bis h_{max} läuft und dabei sukzessiv die Auffangbecken mit den Minima von *f* expandiert.

Sei X_h die Vereinigung aller errechneter Auffangbecken zum Rekursionsschritt *h*. Eine Zusammenhangskomponente der Schwellwertmenge T_{h+1} zum Rekursionsschritt $h+1$ kann entweder ein neues Minimum sein oder zu einem bereits bekannten Auffangbecken aus X_h gehören. Ist letzteres der Fall, so wird die geodätische Einflusszone von X_h innerhalb T_{h+1} berechnet und anschließend X_{h+1} aktualisiert.

Sei MIN_h die Vereinigung aller Regionen-Minima zum Rekursionsschritt (zur Eintauchtiefe) *h,* so ergibt sich die folgende Definition:

Definition 5.2.4 (Wasserscheiden als Eintauchvorgang) Durch die folgende Rekursion

$$\begin{cases} X_{h_{min}} = & \{ p \in D \mid f(p) = h_{min} \} = T_{h_{min}} \\ X_{h+1} = & MIN_{h+1} \cup IZ_{T_{h+1}}(X_h), \qquad h \in [h_{min}, h_{max}) \end{cases}$$

ergibt sich die Wasserscheide von f als das Komplement von $X_{h_{max}}$ in *D*:

$$Wshed(f) = D \setminus X_{h_{max}}.$$

Auf den bekanntesten Vertreter der Implementation der Wasserscheiden-Transformation dieser Definition wird im Folgenden eingegangen werden. Deshalb wird hier kein Beispiel zum funktionellen Ablauf einer Rekursion (siehe Definition 5.2.4) angeführt. Es sei darauf hingewiesen, dass die Algorithmen, die auf dieser Definition aufbauen, keinesfalls „Greedy-Algorithmen“[44] sein müssen. So führt Meijster an, dass Wasserscheiden-Bildpunkte in einem höheren Rekursionsschritt noch zu Auffangbecken hinzugerechnet werden können und somit die ursprüngliche Entscheidung der Zuordnung revidiert werden kann. Dies trifft jedoch auf das später vorgestellte Verfahren von Vincent und Soille nicht zu.

5.2.2.2 Wasserscheiden durch topographische Distanz

Die zweite hier vorgestellte Art der Modellierung der Wasserscheiden besteht darin, ein Distanzmaß einzuführen, welches die Zugehörigkeit eines Bildpunktes zu einem Auffangbecken oder zu einer Wasserscheide bestimmt. Wie schon in Abschnitt 5.2.1 erwähnt, wird hier meist die topographische Distanz gewählt, da sie die Gegebenheiten der Wasserscheiden sehr gut modelliert. Dieser Modellierungsansatz führt im Gegensatz zum vorherigen Abschnitt zu keiner Simulation eines Eintauchens oder Ähnlichem, um die Wasserscheiden-Transformation zu berechnen.

44 Greedy-Algorithmen bezeichnen eine spezielle Klasse von Algorithmen. Sie erreichen die Lösung eines Problems durch schrittweise Annäherung mittels mehrerer Teillösungen. Dabei besitzt eine einmal getroffene Entscheidung während des gesamten Ablaufs immer Gültigkeit. Sie kann nicht aufgrund später getroffener Entscheidungen verworfen werden. Deshalb führen Algorithmen dieser Klasse auch häufig nur zu lokalen Maxima.

Definition 5.2.5 (Maximaler Abstieg) Sei f ein Grauwertbild, das *absteigend vollständig*[45] ist, das heißt, dass jeder Bildpunkt, der kein Minimum darstellt, einen Nachbarn kleineren Grauwertes hat.

Diese Annahme ist nur für den Einstieg wichtig, da sie Plateaus ausschließt. Sie wird jedoch später abgeschwächt.

Der maximale Abstieg $LS(p)$ eines Bildes f am Bildpunkt p ist definiert als der maximale Abstieg, der von einem gegebenen Bildpunkt p zu seinen Nachbarn möglich ist:

$$LS(p) = \max_{q \in N_G(p) \cup \{p\}} \left(\frac{f(p) - f(q)}{d(p,q)} \right),$$

wobei die Menge $N_G(p)$ die Nachbarn des Bildpunktes enthält[46] und d die verwendete Distanz darstellt. Für den Fall $p = q$ wird der Maximum-Operator mit null definiert, was dazu führt, dass Minima-Bildpunkte einen maximalen Abstieg von null besitzen.

Definition 5.2.6 (Diskrete topographische Distanz) Sei die Kostenfunktion für das Laufen von einem Bildpunkt p zu einem benachbarten Bildpunkt q definiert als:

$$cost(p,q) = \begin{cases} LS(p) \cdot d(p,q) & falls \quad f(p) > f(q) \\ LS(q) \cdot d(p,q) & falls \quad f(p) < f(q) \\ \frac{1}{2}\left(LS(p) + LS(q)\right) \cdot d(p,q) & falls \quad f(p) = f(q) \end{cases}.$$

Dann wird die *diskrete topographische Distanz entlang eines Pfades* $\pi = (p_0, \ldots, p_l)$ mit $p_0 = p$ und $p_l = q$ definiert als:

$$T_f^{\pi}(p,q) = \sum_{i=0}^{l-1} d(p_i, p_{i+1}) \cdot cost(p_i, p_{i+1}).$$

Die *diskrete topographische Distanz zwischen zwei Bildpunkten* p und q wird durch das Minimum der diskreten topographischen Distanzen aller Pfade zwischen p und q beschrieben:

$$T_f(p,q) = \underset{\pi \in [p \to q]}{MIN}\, T_f^{\pi}(p,q),$$

wobei $[p \to q]$ die Menge aller Pfade von p zu q beschreibt.

Die diskrete topographische Distanz hat für die Wasserscheiden-Transformation eine wichtige Eigenschaft:

Satz 5.2.1 Sei $f(p) > f(q)$. Ein Pfad π von p nach q ist der Pfad des steilsten Abstiegs genau dann, wenn $T_f^{\pi}(p,q) = f(p) - f(q)$. Wenn ein Pfad π von p nach q nicht der des steilsten Abstiegs ist, gilt: $T_f^{\pi}(p,q) = f(p) - f(q)$.

Dieser Satz impliziert, dass ein Pfad steilsten Abstiegs auch gleichzeitig eine Geodäte (kürzeste Pfadlänge) darstellt. Damit können die Auffangbecken und Wasserscheiden in

45 In der verwendeten Literatur von Meijster [Meij05] wird dies als „lower complete“ bezeichnet.

46 Die Kardinalität von N_G hängt von der gewählten Nachbarschaft ab. Einen Überblick über die verschiedenen zwei- und dreidimensionalen Nachbarschaften zeigt Kapitel 3.

gleicher Art und Weise definiert werden, wie dies im kontinuierlichen Fall in Abschnitt 5.2.1 durchgeführt wurde. Aus dem obigen Satz folgt, dass $CB(m_i)$ die Menge von denjenigen Punkten ist, die oberhalb eines einzigen Minimums liegen. Wasserscheiden liegen demnach dort, wo mehr als ein Minimum auf einen Bildpunkt einwirkt. An diesen Stellen gibt es mindestens zwei Pfade steilsten Abstiegs, die bei *p* beginnen und in verschiedenen Minima enden.

Korollar 5.2.1 Jeder Bildpunkt, der oberhalb eines Wasserscheiden-Bildpunktes liegt, ist selbst ein Wasserscheiden-Bildpunkt.

Das Wasserscheiden-Verfahren, das durch die vorherigen Definitionen beschrieben wurde, zeigt bei der Anwendung deutliche Unterschiede.

2	**0**	1	3
3	2	4	2
5	3	2	**1**
6	5	4	3
9	6	7	8

a) Ausgangsbild

0	**0**	0	W
0	0	W	1
0	W	1	**1**
W	1	1	1
1	1	1	1

b) 4-Verbunden

0	**0**	0	0
0	0	W	W
0	0	W	**1**
0	0	W	1
0	0	W	1

c) 8-Verbunden

0	**0**	0	0
0	0	0	1
W	W	1	**1**
W	W	1	1
W	W	1	1

d) 4-Verbunden

0	**0**	0	0
0	0	0	W
0	W	1	**1**
W	1	1	1
1	1	1	1

e) 8-Verbunden

Abbildung 39: Übersicht über die Ergebnisse der beiden vorgestellten Verfahren. a) Das Ausgangsbild, b) und c) das Ergebnis der Segmentierung mit dem „Eintauchen"-Verfahren und den beschriebenen Nachbarschaften. d) und e) die Ergebnisse, die „Topographische-Distanz"-Verfahren liefern.

Eine Konsequenz der Definition 5.2.1 ist das Auftauchen von „dicken" Wasserscheiden, welche bei der Sichtweise der Wasserscheiden als Eintauchvorgang nicht zu erkennen sind. Obwohl es auch bei letzerem Verfahren möglich ist, dass Wasserscheiden in gleicher Gestalt auftreten, so geschieht dies laut [Meij05][47] doch weniger ausgeprägt als bei der Definition der Wasserscheiden durch topographische Distanz.

Allerdings bleibt noch ein Problem bei dieser Definition der Wasserscheiden-Transformation: Das Plateau-Problem. Bei der Definition 5.2.5 wurde davon ausgegangen, dass das zu verarbeitende Bild absteigend vollständig ist. Leider ist dies meist nicht der Fall. Deshalb wird im Folgenden (vgl. [Meij05], S. 99) eine Ordnungsrelation zwischen Bildpunkten auf einem Plateau eingeführt. Die Standard-Vorgehensweise hierbei ist, dass als Ordnungsrelation der geodätische Abstand der auf dem Plateau liegenden Bildpunkte zum Rand gewählt wird.

Dieses kann dadurch formalisiert werden, dass das Bild erst in ein absteigend vervollständigtes Bild transformiert wird und dann darauf die bereits bekannten Definitionen angewendet werden.

Definition 5.2.7 (Absteigende Vervollständigung) Sei *f* ein digitales Grauwertbild mit Domäne *D*, und sei $\Pi_f^{\downarrow}(p)$ die Menge aller absteigenden Pfade, die im Bildpunkt *p*

47 siehe S. 97ff

beginnen und in einem Bildpunkt *q* enden, mit $f(q)<f(q)$. Sei $\text{length}(\pi)$ die Länge des Pfades π. Die Funktion $d:D\to\mathbb{N}$ sei definiert durch:

$$d(p)=\begin{cases}0 & falls\ \ \Pi_f^{\downarrow}(p)=\emptyset\\ \min_{\pi\in\Pi_f^{\downarrow}(p)}\text{length}(\pi) & sonst\end{cases}.$$

Sei $L_c=max_{p\in D}d(p)$. Dann ist die absteigende Vervollständigung f_{LC} von *f* definiert als:

$$f_{LC}(p)=\begin{cases}L_c\cdot f(p) & falls\ \ d(p)=0\\ L_c\cdot f(p)+d(p)-1 & sonst\end{cases}$$

Ein Beispiel für den Prozess der absteigenden Vervollständigung ist in der folgenden Abbildung 40 zu sehen. Durch diesen Prozess ist es nun möglich, eine Ordnungsrelation $\sqsubset$ auf den Bildpunkten zu definieren:

$$x\sqsubset y\Leftrightarrow f_{LC}(x)<f_{LC}(y).$$

Dadurch wird die Funktion T_{f^*} mit $f^*=f_{LC}$ zu einem geeigneten Distanzmaß auf $D'\times D'$, wobei D' der Domäne *D* ohne Minima entspricht.

Die spezielle Definition des steilsten Abstieges und der Kostenfunktion stellt an dieser Stelle sicher, dass am steilsten absteigende Pfade auch der kürzesten topographischen Distanz entsprechen. Somit lässt sich jetzt die Wasserscheiden-Transformation für beliebige Bilder definieren.

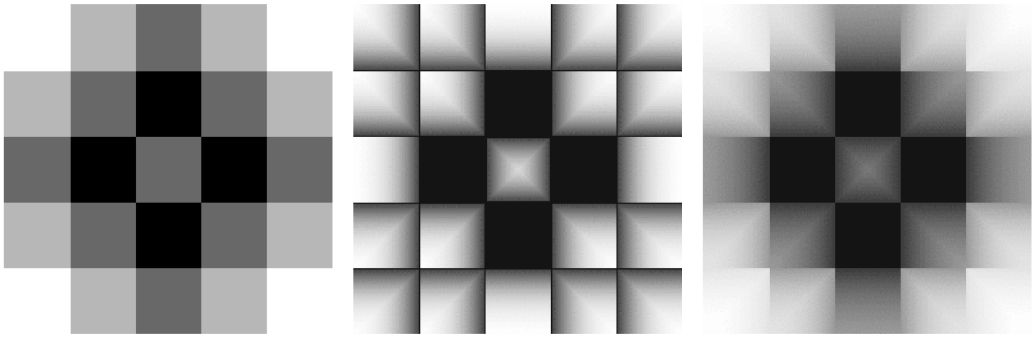


Abbildung 40: Der Vorgang der absteigenden Vervollständigung. Links: Das Ausgangsbild. Mitte: Das Bild der absteigende Distanz. Rechts: Das absteigend vervollständigte Bild. (aus [Meij05], S. 99)

Definition 5.2.8 (Absteigend vollständiger Graph) Sei $G=(V,E,f)$ ein digitales Grauwertbild. Der absteigend vollständige Graph $G'=(V,E')$ ist definiert durch alle Punkte *p*, die einen Nachbarn kleineren Grauwertes haben:

$$(p,p')\in E'\Leftrightarrow p'\in\Gamma(p),$$

wobei $\Gamma(p)$ die Menge der Nachbarn kleineren Grauwertes von *p* ist, dessen Abstieg maximal ist.

Definition 5.2.9 (Wasserscheiden-Transformation durch topographische Distanz)

Sei *f* ein Grauwertbild, mit der absteigenden Vervollständigung $f^* = f_{LC}$. Seien $(m_i)_{i \in I}$ die Minima von *f*. Das Auffangbecken $CB(m_i)$ von *f*, welches zu einem Minimum m_i gehört, wird durch die absteigende Vervollständigung von *f* definiert:

$$CB(m_i) = \left\{ p \in D \,\middle|\, \forall_{j \in I \setminus \{i\}} f^*(m_i) + T_{f^*}(p, m_i) < f^*(m_j) + T_{f^*}(p, m_j) \right\},$$

die Definition der Wasserscheiden von *f* ergibt sich aus Definition 5.2.2.

5.2.3 Zwei ausgewählte Algorithmen

In diesem Abschnitt werden zwei der in der Praxis am häufigsten implementierten Wasserscheiden-Verfahren vorgestellt. Da sie in dem Kontext dieser Arbeit als Verfahren, welche auf diskreten Bilddaten operieren, eingeführt werden, finden sich die theoretischen Grundlagen beider Verfahren in den vorherigen Abschnitten.

Die Stärken beziehungsweise Schwächen beider Algorithmen werden in Kapitel 5.3 herausgearbeitet, wenn es darum geht, die Auswahl des Verfahrens, welches in dieser Arbeit verwendet wird, zu begründen. Da diese Analyse aber sehr stark vom Kontext der Fragestellung dieser Arbeit beeinflusst ist, sei darauf hingewiesen, dass ein ausführlicher Vergleich in [Meij05] auf Seite 103ff zu finden ist.

5.2.3.1 Wasserscheiden-Transformation nach Vincent und Soille

Der erste hier präsentierte Wasserscheiden-Algorithmus stammt von Vincent und Soille. Er wurde 1990 vorgestellt (siehe [VS91]). Eine umfassendere Übersicht findet sich in [Meij05]. Es handelt sich hierbei um eine Implementation der Modellierung der Wasserscheiden-Transformation durch Eintauchen (vgl. 5.2.2.1). Noch exakter lässt sich sagen, dass es sich hierbei um eine Implementation handelt, die der Transformationsgleichung aus Definition 5.2.4 sehr nahe ist.

Die Arbeitsweise des Algorithmus' lässt sich in zwei Arbeitsschritte unterteilen:

1. Sortiere alle Bildpunkte ansteigend nach ihren Grauwerten.

   Dies ist wichtig, damit das Verfahren einen schnellen Zugriff auf alle Bildpunkte eines Grauwertes erhält.

2. Flute alle Bildpunkte, ausgehend von den Minima, für jeden Grauwert einzeln.

Vincent und Soille benutzen dazu in ihrer Implementation eine „FIFO-Schlange“[48] von Bildpunkten. Diese muss nur einige grundlegende Operationen unterstützen:

- Bildpunkt ans Ende der Schlange hinzufügen
- Vordersten Bildpunkt aus der Schlange entfernen
- Test, ob die Schlange leer ist

48 Der Begriff FIFO-Schlange bezeichnet einen abstrakten Datentyp, aus dem zuerst eingefügte Elemente auch zuerst wieder aus der Schlange entfernt werden. (First-In-First-Out-Prinzip)

Der Algorithmus ordnet anfangs jedem Minimum ein eindeutiges Label zu. Anschließend wird im Flutungsschritt auch den Auffangbecken der Minima das jeweilige Label zugeordnet. Diese Flutung wird durch eine iterative Breitensuche auf dem Graphen des Bildes vollzogen. Im Flutungsschritt werden zunächst alle Bildpunkte eines Grauwerts h auf ein spezielles Maskierungslabel *MASK* gesetzt. Diejenigen Bildpunkte, die gelabelte benachbarte Bildpunkte aus der vorangegangenen Iteration haben, werden ans Ende der Schlange angefügt, und dessen geodätische Einflusszonen werden innerhalb der Menge der maskierten Bildpunkte propagiert.

Falls ein Bildpunkt zu mehr als einem Auffangbecken benachbart ist, wird er als Wasserscheiden-Bildpunkt klassifiziert. Ist der Bildpunkt nur über ein Auffangbecken erreichbar, so wird er diesem beziehungsweise dessen Label zugeordnet. Alle Bildpunkte, die am Ende dieses Schrittes immer noch den Wert *MASK* besitzen, gehören zu einer Menge neuer Minima im Flutungsschritt h. Die Zusammenhangskomponenten dieser Bildpunkte bilden jeweils ein neues Label.

Die Zeitkomplexität dieses Algorithmus' ist linear in Bezug auf die Anzahl der Bildpunkte des zu verarbeitenden Bildes. Auf eine Darstellung dieses Algorithmus' in Pseudo-Code wird an dieser Stelle verzichtet, es sei aber auf [VS91] und [Meij05] hingewiesen, wo sich dieser nachschlagen lässt. Weiterhin sei noch angemerkt, dass der hier beschriebene Algorithmus nicht exakt die in Definition 5.2.4 beschriebene Rekursion implementiert, sondern sich in folgenden Punkten davon unterscheidet:

1. Zum Schritt h werden nur Bildpunkte mit Grauwert h für den Flutungsschritt maskiert, während in Definition 5.2.4 alle Bildpunkte, die zu keinem Auffangbecken gehören, mit Grauwert $\leq h$ betrachtet werden.
2. Nicht nur Label von Auffangbecken werden propagiert, sondern auch Label von Wasserscheiden-Bildpunkten. Dies ist eine Konsequenz des ersten Punktes.
3. Ein Bildpunkt, der zu mehr als einem Auffangbecken gehört und deswegen als Wasserscheide klassifiziert wurde, kann in einem anderen Schritt mit dem Grauwert eines Nachbarn überschrieben werden. Dies geschieht dann, wenn der Nachbar Teil nur eines Auffangbeckens ist. Die Motivation hierbei ist, dass keine „abweichenden“ Wasserscheidenkonturen entstehen sollen. Dieser Schritt liegt im kontinuierlichen Fall begründet. (vgl. [Meij05] S. 105f)

Der hier vorgestellte Algorithmus lässt sich leicht so abwandeln, dass er die Rekursion aus Definition 5.2.4 implementiert. Allerdings besitzt er dann im schlechtesten Fall eine quadratische Komplexität. Meijster betont in [Meij05] daher, dass diese theoretisch schlechtere Komplexität im normalen Anwendungsfall kaum spürbar sein würde.

5.2.3.2 Union-Find-Wasserscheiden-Transformation

Dieses Verfahren unterscheidet sich deutlich von dem des vorigen Abschnitts. So implementiert es die Sichtweise auf Wasserscheiden durch topographische Distanz und nicht durch Eintauchen. Diese Gruppe von Verfahren lässt sich in zwei Untergruppen gliedern:

1. Geordnete Algorithmen

   Hierbei werden die Bildpunkte, für die die kürzeste topographische Distanz bereits bekannt ist, anhand der Distanz sortiert. Die weitere Verarbeitung geschieht dann auf dieser Ordnung der Bildpunkte.

2. Ungeordnete Algorithmen

   Bei dieser Klasse von Algorithmen wird die Distanz auf dem gegebenen Raster von Bildpunkten ermittelt. Eine Neuordnung der Bildpunkte ist nicht notwendig.

Beide Klassen von Algorithmen haben gemeinsam, dass sie in zwei Schritten ablaufen:

1. Bestimmung der Minima im Bild
2. Zuordnung der nicht minimalen Bildpunkte zu den Minima (oder Klassifizierung als Wasserscheiden-Bildpunkt)

Für den ersten Schritt, die Suche nach Minima im Bild, bietet sich der Union-Find-Algorithmus an, der von Tarjan 1975 vorgestellt wurde (vgl. [Tar75]). Dieser ist in der Praxis meist sehr viel schneller als flutungsbasierte Verfahren[49], da das Bild einerseits sequentiell (in Scan-Order) abgearbeitet werden kann; andererseits entstehen durch das Verfahren auch weniger unstrukturierte Speicherzugriffe als dies bei geordneten Verfahren der Fall wäre.

Wichtig ist, in den nächsten Abschnitten die Übersicht über die Algorithmen zu wahren. Wenn vom Union-Find-Algorithmus die Rede ist, so wird auch nur dieser beschrieben und nicht die darauf aufbauende Wasserscheiden-Transformation.

Union-Find-Algorithmus

Der Union-Find-Algorithmus speichert disjunkte Mengen in Bäumen, welche einen Wald bilden. In jedem Baum zeigt jeder Knoten p auf seinen Vater p' (siehe hierzu auch Abbildung 41). Ein Knoten p heißt *Wurzel eines Baumes*, falls gilt: $p = p'$, wobei p' der Vater von p ist.

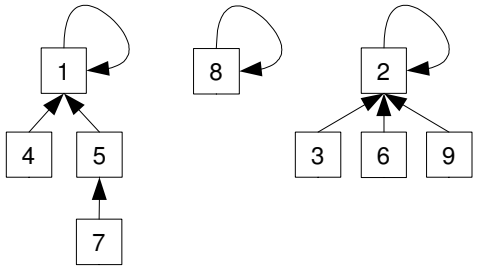


Abbildung 41: Ein Wald disjunkter Mengen von Zahlen: {1,4,5,7} {8} {2,3,6,9}.

Durch diese Repräsentation der Bäume sind Verschmelzungen besonders einfach zu realisieren: Seien B_1 und B_2 zwei Bäume disjunkter Mengen. B_1 und B_2 können

49 Sie werden in der englisch- wie deutschsprachigen Literatur ebenfalls Floodfill-Verfahren genannt.

verschmolzen werden, indem, ohne Beschränkung der Allgemeinheit, der Vater der Wurzel $r_1 \in B_1$ auf die Wurzel $r_2 \in B_2$ gesetzt wird. Dieses kann aber recht schnell zu Bäumen mit großer Höhe führen, was sich in längeren Suchzeiten des Verfahrens niederschlagen würde.

Aufgrund dessen wird noch eine Pfadkomprimierung durchgeführt, die zum Ziel hat, alle Knoten, die keine Wurzeln sind, direkt mit der Wurzel zu verbinden. Dadurch gelingt es in der Praxis häufig, dass die Bäume eine Höhe von drei nicht überschreiten. Tarjan schlägt in [Tar83] eine weitere Technik vor, die nicht nur die Höhe des Baumes beschränkt hält, sondern auch die Balance des Baumes beim Verschmelzen gewährleistet. Diese spezielle Technik wäre im Rahmen dieser Arbeit ein zu großer Aufwand und findet deshalb im Weiteren keine Verwendung.

Mit Hilfe dieser Technik ist es einfach, das Labeln von Zusammenhangskomponenten mit einem sequentiell über die Bilddaten laufenden Verfahren zu realisieren (siehe Algorithmus 5.3.1). In diesem Fall sind die Bildpunkte die Knoten der Bäume.

Definition 5.2.10 (Lexikographische Ordnung zwischen Voxeln) Sei $\prec$ die lexikographische Ordnungsrelation zwischen den einzelnen Bildpunkten. Für ein Volumen gilt daher:

Seien $p=(x_1, y_1, z_1)$ und $q=(x_2, y_2, z_2)$ zwei Voxel.

$$p \prec q \quad \equiv \quad (z_1 < z_2) \vee \left(z_1 = z_2 \wedge \left(y_1 < y_2 \vee \left(y_1 = y_2 \wedge x_1 < x_2\right)\right)\right)$$

Dazu lässt sich analog $p \preceq q \quad \equiv \quad (p \prec q \vee p = q)$ definieren.

Diese lexikographische Ordnung sei, bei sequentiell über die Bilddaten laufenden Verfahren, die Ordnung, die den Zeitpunkt der Abarbeitung eines Bildpunktes definiert. Sei p_0 der erste verarbeitete Bildpunkt, und sei *curpoint* der aktuell zu verarbeitende Bildpunkt. Dann wird die folgende Ordnung stets beibehalten:

$\forall_p (p_0 \preceq p \preceq curpoint) \Rightarrow p_0 \preceq p' \preceq p$, wobei p' der Vater von p ist.

Dadurch wird gewährleistet, dass keine Zyklen auftreten. Außerdem ist es so einfach, den Vater eines Knotens iterativ zu berechnen, um die Wurzel des Baumes zu bestimmen.

Zum weiteren Ablauf des Algorithmus' lässt sich Folgendes festhalten:

Sei p der aktuell zu verarbeitende Bildpunkt. Falls p keine Nachbarn q (mit $p \prec q$) mit dem gleichen Intensitätswert besitzt, wird ein neuer Baum erstellt, in dem der Vater von p auf p gesetzt wird. Somit wird p zu einem neuen Baum mit Wurzel p.

Falls es benachbarte Bildpunkte q_i von p (mit $q_i \prec p$) mit gleichem Intensitätswert gibt, so werden zunächst die Repräsentanten der benachbarten Bildpunkte ermittelt. Dies geschieht durch den weiter oben beschriebenen Algorithmus zum Finden der Wurzel eines Baumes. Der lexikographisch kleinste Repräsentant wird dann als neuer Repräsentant für die Vereinigung der Menge, die diese Nachbarn enthält, gewählt. Anschließend werden die Pfade noch komprimiert und p wird mit der vereinigten Menge verschmolzen.

Im zweiten Schritt des Algorithmus' wird das Label-Bild erstellt. In ihm bekommen alle Wurzeln ein eindeutiges Label zugewiesen. Alle anderen Bildpunkte bekommen die Werte ihrer Repräsentanten.

Der Algorithmus kann für die Berechnung von Zusammenhangskomponenten innerhalb von Bilddaten beliebiger Dimension, Größe und Verbundenheit von Bildpunkten verwendet werden[50]. Andere Algorithmen, wie zum Beispiel von Rosenfeld und Pfaltz (siehe [RP66]), sind auf zweidimensionale Bilddaten mit einer 4er-Nachbarschaft beschränkt. Was diesen Algorithmus aber am interessantesten macht, ist die Tatsache, dass sich mit seiner Hilfe recht elegant eine Wasserscheiden-Transformation beschreiben lässt.

Wasserscheiden-Transformation mittels des Union-Find-Algorithmus'

Um mit dem Union-Find-Algorithmus die eigentliche Wasserscheiden-Transformation zu berechnen, schlägt Meijster (in [Meij05], S. 116) folgende Schritte vor:

1. Die in dem Bild f vorhandenen Plateaus müssen entfernt werden. Dies kann dadurch geschehen, dass die absteigende Vervollständigung f_{LC} gebildet wird (siehe Definition 5.2.7). Der letzte Schritt der absteigenden Vervollständigung kann zudem leicht angepasst werden, so dass alle Bildpunkte, die Minima darstellen, das Label null zugewiesen bekommen.
2. Aus dem absteigend vervollständigten Bild wird der absteigend vollständige Graph $G'=(V,E)$ erstellt (siehe Definition 5.2.8). Dieser kann mit einem Durchlauf des Bildes berechnet werden. Ein Beispiel für einen solchen Graphen ist in Abbildung 42 gegeben.

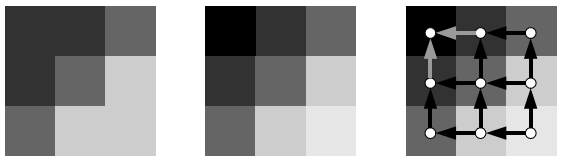


Abbildung 42: Absteigend vervollständigter Graph: links: das Ausgangsbild; mitte: Das Bild nach der absteigenden Vervollständigung; rechts: der zugehörige DAG.

Meijster schlägt vor, den gerichteten azyklischen Graphen (DAG) in einem Array sln zu speichern, wobei $sln[p,i]$ einen Zeiger auf den *i*-steilsten niedrigeren Nachbarn von *p* repräsentiert. Für jedes Minimum *m* wird ein Bildpunkt $r \in m$ als Repräsentant ausgewählt, und ein Zeiger von *r* auf sich selbst erstellt.

Wie hieraus ersichtlich wird, spielt in dem Vorschlag von Meijster das Array sln die Rolle der Vater-Relation auf den Bäumen (siehe oben), allerdings mit der Erweiterung, dass ein Knoten nun mehr als einen Vater haben kann. Dies führt dazu, dass der Graph nun nicht mehr aus einer disjunkten Menge von Wäldern besteht.

50 Dies führt dazu, dass er sich für eine generische Implementation in Zusammenhang mit der in Kapitel 4 eingeführten, gekapselten Nachbarschaft eignet.

3. Der letzte Schritt des Verfahrens besteht darin, dass der Union-Find-Algorithmus (siehe oben) auf den DAG angewendet wird. Dabei ist der erste Durchlauf ähnlich dem des ursprünglichen Algorithmus. Der zweite Durchlauf muss dann angepasst werden, wenn anstatt der Regionen auch Wasserscheiden-Bildpunkte markiert werden sollen. Hierzu genügt es jedoch, den ursprünglichen Algorithmus zum Finden der Wurzel eines Baumes anzupassen. (vgl. [Meij05] S. 116)

Der in diesem Abschnitt beschriebene Algorithmus berechnet die exakte Wasserscheiden-Transformation durch topographische Distanz (siehe Definition 5.2.9). Zudem führt das Verfahren die Wasserscheiden-Transformation sehr schnell durch. Besonders wichtig ist dies, wenn nicht nur zweidimensionale sondern auch dreidimensionale Bilddaten transformiert werden sollen, oder, wie in dieser Arbeit, das Segmentierungsverfahren nur einen Vorverarbeitungsschritt der eigentlichen Analyse der Bilddaten darstellt.

5.3 Implementation der Union-Find-Wasserscheiden-Transformation

Dieses Unterkapitel beschäftigt sich mit der Implementation des Segmentierungsverfahrens. Da dieses im Rahmen der Arbeit einen Grundbaustein aller weiteren Verarbeitungsschritte darstellt, war die korrekte Wahl des zu implementierenden Verfahrens von großer Bedeutung.

Folgende Kriterien sollte das Verfahren erfüllen:

1. Die Eingabe beliebiger dreidimensionaler Bilddaten sollte möglich sein.

   Es sollte kein Problem sein, Grauwertvolumen mit einem Wertebereich von 2^8 oder 2^{16} einzulesen und zu verarbeiten.

2. Die Ausgabe sollte in einem gelabelten Volumen bestehen.

   Dabei sollten stets geschlossene Regionen ein Label bilden. Es sollten keine offenen Kantenverläufe als Segmentierungsergebnisse zurückgegeben werden.

3. Eine Übersegmentierung ist erlaubt.

   Da in weiterführenden Arbeitsschritten die Möglichkeit der nachträglichen Analyse von Volumenelementen und deren Verschmelzung besteht, ist es nicht unbedingt erforderlich, dass die Segmentierungsergebnisse bereits bestmöglich der menschlichen Wahrnehmung entsprechen. Wie später ersichtlich wird, ist diese Annahme jedoch mit Vorsicht zu genießen, denn eine sehr starke Übersegmentierung kann auch Auswirkungen auf die Geschwindigkeit der nachfolgenden Prozesse, insbesondere der Erstellung der topologischen Karte (siehe Kapitel 9.2), haben.

4. Es sollte eine generische Vorgehensweise durchführbar sein.

   Das Verfahren sollte die in Kapitel 4 beschriebene gekapselte Nachbarschaft benutzen und generisch für verschiedenste Arten von Grauwertvolumen funktionieren können. Das Verfahren sollte idealerweise nur ein Mal implementiert werden und dann für alle Volumentypen und Nachbarschaften automatisch korrekte Ergebnisse liefern.

   Außerdem sollte das Verfahren nicht nur im Rahmen dieser Arbeit seinen Platz erhalten, sondern sich auch in der VIGRA-Bibliothek[51] eingliedern.

5. Die Zeit- und Speicherkomplexität sollte möglichst gering sein.

   Diese Annahme ist aus zweierlei Gründen sehr wichtig: Zum einen muss das Verfahren auf dreidimensionalen Bilddaten arbeiten, was bereits eine grundlegend komplexere Problemstellung darstellt als die Segmentierung eines zweidimensionalen Bildes. Es steigt hierbei sowohl die Speicher- als auch die Zeitkomplexität deutlich an.

51 VIGRA ist eine C++-basierte Plattform zur generischen Programmierung in der Bildverarbeitung (siehe auch [Köt00]).

Zum anderen stellt das Segmentierungsverfahren in dieser Arbeit lediglich eine Vorverarbeitung dar, daher sollte es auch möglichst wenig Zeit und Speicher in Anspruch nehmen.

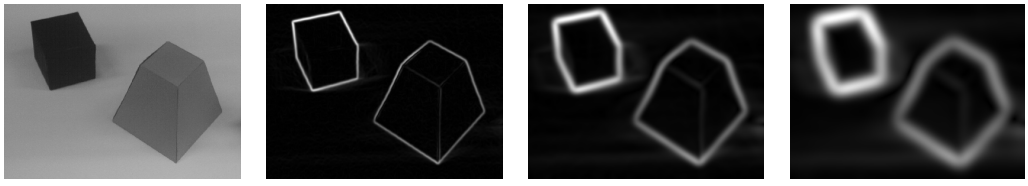
Die Anforderungen 1. und 4. sind größtenteils durch die konkrete Implementation zu erfüllen, an diesen lassen sich noch keine Präferenz auf einen Algorithmus oder auf eine Gruppe von Algorithmen ablesen. Der zweite Punkt lenkt die Auswahl der Verfahren in Richtung Wasserscheiden-Transformation. Hier ist durch die Definition gewährleistet, dass nur zusammenhängende Regionen zu einem Minima und somit zu einem Auffangbecken (vgl. Kapitel 5.2) zugeordnet werden. Es entstehen also nur – im Sinne der verwendeten Nachbarschaft – zusammenhängende gelabelte Regionen. Auch der dritte Punkt der Anforderungen spricht für diese Klasse von Verfahren, spiegelt er doch eine der grundlegenden Eigenschaften der Wasserscheiden-Transformation wider. Der fünfte Punkt schließlich grenzt die Suche nach einem Algorithmus noch enger ein. Wenn die Wasserscheiden-Transformation zur Volumensegmentierung angewandt werden und die Zeit- und Speicherkomplexität möglichst gering sein soll, so bleibt nur das im vorigen Abschnitt 5.2.3.2 beschriebene Verfahren der Union-Find-Wasserscheiden-Transformation.

Da sich die verwendete Implementation des Union-Find-Wasserscheiden-Verfahrens von dem Vorschlag von Meijster unterscheidet, werden in den nächsten Abschnitten die einzelnen Schritte erneut aufgeführt und erläutert, allerdings in der Form, in der sie implementiert wurden.

5.3.1 Vorverarbeitung der Volumendaten

Es wird, wie bereits in Kapitel 5.2 erwähnt, nicht das Volumen selbst, sondern dessen Gradientenbetragsvolumen $\|\nabla I\|$ als Eingabevolumen des Verfahrens betrachtet. Dabei beschreibt das Gradientenbetragsvolumen den Betrag der partiellen Ableitungen eines Voxels in seine Achsen. Es wird im Allgemeinen berechnet, indem das ursprüngliche Bild beziehungsweise Volumen I mit einer Faltungsmaske gefaltet wird. Gute Ergebnisse liefert eine Faltung mit einer einfach abgeleiteten Gauß-Funktion $G'(\sigma)$, die $\|\nabla I\| = \|G'(\sigma) * I\|$ liefert.

Als einziger festzulegender Parameter dieser Art der Vorverarbeitung bleibt noch σ, die Standardabweichung der Gauß-Funktion.



a) Ausgangsbild I b) $\|\nabla I\|$, $\sigma = 1.6$ c) $\|\nabla I\|$, $\sigma = 5$ d) $\|\nabla I\|$, $\sigma = 10$

Abbildung 43: Beispiele für Gradientenbetragsvolumen, welche mit einer einfach abgeleiteten Gauß-Funktion $G'(\sigma)$ berechnet wurden.

Folgt man Meijster[52], so müsste nach der Erstellung des Gradientenbetragsvolumen geprüft werden, ob dieses Plateaus besitzt.[53] Wäre dies der Fall, so müsste zu dem Ausgangsvolumen das absteigend vervollständigte Volumen erstellt werden. Dieses würde dann das neue Ausgangsvolumen werden. Nun gibt es aber einige Gründe, diese Vervollständigung nicht durchzuführen, auch wenn dadurch das Ergebnis des Algorithmus' nicht mehr in allen Fällen der Definition der Wasserscheiden-Transformation durch topographische Distanz (siehe Definition 5.2.9) entspricht. So ist der Vorgang der absteigenden Vervollständigung recht rechenintensiv. Es müssen zunächst alle Plateaus im Volumen gefunden werden, um anschließend eine Ordnungsrelation auf diesen einzuführen.

Folgende Alternativen zur absteigenden Vervollständigung sind denkbar:

1. Ignorieren von Plateaus

   Das Volumen wird, so wie es ist, an das Verfahren weitergeleitet.

2. Anpassen der Erstellung des Graphen des Volumens

   Hierbei wird der Graph, welcher im ersten Schritt des Verfahrens erzeugt wird, so in seiner Struktur verändert, dass er mit Plateaus „umgehen“ kann. Was dies genau bedeutet wird später in Kapitel 5.4 behandelt werden.

3. Vor dem Erstellen des Gradientenbetragsvolumens Rauschen einfügen

   Falls das Rauschen nur eine sehr geringe Energie und einen sehr hohen Signalabstand besitzt, so wird es die Bilddaten nicht negativ beeinflussen, wohl aber dafür sorgen, dass keine Plateaus mehr auftreten können.

Die hier angeführte Liste erhebt keinen Anspruch darauf, vollständig zu sein. Sie soll vielmehr eine Ideensammlung darstellen. Wie sich die Punkte eins und zwei in der Praxis auswirken, wird in Kapitel 5.4 gezeigt werden. Der dritte hier erwähnte Punkt wird im Folgenden nicht weiter betrachtet. Es sei aber darauf hingewiesen, dass auch er zu einer Lösung des Problems beitragen kann.

Für den im nachfolgenden Abschnitt beschriebenen Algorithmus der Wasserscheiden-Transformation kann weiterhin davon ausgegangen werden, dass er korrekt (gemäß der Definition 5.2.9) arbeitet, solange das Ausgangsbild absteigend vollständig ist.

52 Siehe 5.2.3.2, unter dem ersten Punkt bei der Union-Find-Wasserscheiden-Transformation.

53 Dies ist nur in der Praxis nötig, da Plateaus beim Falten mit einer diskreten abgeleiteten Gauß-Funktion entstehen können.

5.3.2 Segmentierung der vorverarbeiteten Daten

Die Segmentierung der vorverarbeiteten Volumendaten mittels der Union-Find-Wasserscheiden-Transformation lässt sich in zwei Schritte aufteilen, in einen vorbereitenden Schritt (die Erstellung des DAG) und das anschließende Finden der Zusammenhangskomponenten des DAG (vgl. Abschnitt 5.2.3).

Der vorbereitende Schritt erzeugt in der hier diskutierten Implementation allerdings keinen expliziten DAG, sondern ein Volumen mit Voxeln, deren Werte die Richtung zum lokalen Nachbarn geringsten Grauwertes darstellen (siehe Definition 5.2.5). Dabei wird die Bit-basierte Codierung der Nachbarn verwendet, welche bereits in Kapitel 4.2 beschrieben und definiert wurde.

Der folgende Pseudo-Code zeigt den algorithmischen Ablauf des ersten Schrittes der Union-Find-Wasserscheiden-Transformation:

Algorithmus 5.1: Vorbereitender Algorithmus der Union-Find-Wasserscheiden-Transformation

```
function prepareWatersheds3D

   input : volume, neighborhood
   output: direction_volume, local_minimum_count

   local_minimum_count = 0

   for all voxels of volume in lexical order
      orientation = 0
      if current voxel is at volumeborder
         find lowest neighbor in the neighborhood using the
         RestrictedNeighborhoodTraverser
      else
         find lowest neighbor in the neighborhood using the NeighborhoodTraverser
      orientation = directionBit of lowest neighbor

      if orientation = 0
         increment local_minimum_count

      current voxel of direction_volume = orientation
```

Ein Beispiel zu dem `prepareWatersheds3D`-Verfahren zeigt Abbildung 42. Dazu wurde eine 4er-Nachbarschaft auf einem Grauwertbild verwendet.

Nachdem die implizite Repräsentation des DAG, gegeben durch das mittels `prepareWatersheds3D` berechnete Richtungsvolumen, erstellt wurde, kann diese als Basis des Union-Find-Algorithmus' verwendet werden (vgl. 5.2.3.2). Der offensichtlichste Unterschied der Implementation ist, wie bereits in `prepareWatersheds3D`, die Optimierung des Verfahrens auf die implizite Repräsentation des Richtungsvolumens. Damit diese Implementation einfacher beschrieben werden kann, folgt nun noch eine Definition:

Definition 5.3.1 (Guter Nachbar) Sei f ein Bild dessen Wertebereich die Bit-kodierten Richtungen der Nachbarschaft NH ist. Sei NT ein NEIGHBORHOODTRAVERSER mit der Nachbarschaft NH auf einem Bildpunkt $p \in dom(f)$. Ein Nachbar $q \in NH(p)$ heißt gut, falls mindestens eine der beiden nachfolgenden Bedingungen erfüllt ist:

1. $NT(p)_{direction\_bit=f(p)}=q$

   Die im Bildpunkt p gespeicherte Richtung zeigt auf den Nachbarn q.

2. $NT(q)_{direction\_bit=f(q)}=p$

   Die im Bildpunkt q gespeicherte Richtung zeigt auf den Bildpunkt p.

Diese Definition folgt analog aus dem Auswahlkriterium von Meijster (siehe 5.2.3.2), welches bestimmt, wann Nachbarn im Bild zu gleichen Bäumen zugeordnet werden und wann nicht. Mit dieser Definition wird es nun recht einfach, den implementierten Algorithmus zu beschreiben:

Algorithmus 5.2: Union-Find-Labeling zur Wasserscheiden-Transformation

```
function watershedLabeling3D

   input : volume, neighborhood
   output: outputVolume, region_count

   for all voxels of volume in lexical order
      current voxel of outputVolume = lexical number
      if current voxel is at causal volumeborder
         for all good neighbors in given neighborhood using the causal allowed
         directions
            find the root_label of a label tree
            if root_label < current voxel of outputVolume
               outputVolume(outputVolume(current voxel)) = root_label
               current voxel of outputVolume = neighborLabel
            else
               outputVolume(root_label) = current voxel of outputVolume
      else
         for all good neighbors in given Neighborhood using the causal  allowed
         directions
            find the root_label of a label tree
            if root_label < current voxel of outputVolume
               outputVolume(outputVolume(current voxel)) = root_label
               current voxel of outputVolume = neighborLabel
            else
               outputVolume(root_label) = current voxel of outputVolume

   compress trees and count regions
```

Nun sind beide Teilalgorithmen beschrieben, sodass sich die eigentliche Wasserscheiden-Transformation durch folgenden Algorithmus beschreiben lässt:

Algorithmus 5.3: Union-Find-Wasserscheiden-Transformation

```
function watersheds3D

   input : inputVolume,  neighborhood
   output: outputVolume, region_count

   define inputVolume, directionVolume, outputVolume of given size;

   call prepareWatersheds3D with     input: inputVolume, neighborhood
                                     output: directionVolume

   return watershedLabeling3D with   input: directionVolume, neighborhood
                                     output: outputVolume
```

Damit die Implementation in der Praxis noch komfortabler verwendet werden kann, stehen neben der generischen Version, die die Übergabe einer kodierten Nachbarschaft (vgl. Kapitel 4) erwartet, noch zwei weitere zur Verfügung:

1. `watersheds3DSix`, welches mit der dreidimensionalen 6er-Nachbarschaft arbeitet, und
2. `watersheds3DTwentySix`, welches das Verfahren mit der dreidimensionalen 26er-Nachbarschaft aufruft.

Die hier vorgestellte Implementation orientiert sich an allen generischen Voraussetzungen, die im Rahmen der generischen Bildverarbeitung im Verlauf der Arbeit an sie gestellt wurden. Folgende Anforderungen wurden berücksichtigt und erfüllt:

1. Kapselung vom Wertebereich und von der Größe der Daten

   Die Implementation kann für beliebige Bilddaten verwendet werden.

2. Kapselung von der Nachbarschaft

   Die Implementation kann für beliebige dreidimensionale Nachbarschaften verwendet werden, die die Anforderungen erfüllen, die in Kapitel 4 genannt wurden.

Des Weiteren ist auch die Geschwindigkeit der Implementation vielversprechend, wie sich in diesem Kapitel noch zeigen wird. Zunächst sollen aber einige Anwendungsbeispiele folgen.

5.4 Anwendung der Union-Find-Wasserscheiden-Transformation

In diesem Kapitel sollen einige Beispiele gezeigt werden, die die Funktion und den Ablauf der Union-Find-Wasserscheiden-Transformation veranschaulichen und verdeutlichen. Wie bereits im obigen Kapitel 5.3 beschrieben, arbeitet das Verfahren in mehreren Teilschritten. Diese werden anhand von Beispielen nun erläutert werden.

Im nächsten Abschnitt werden zunächst Beispiele zweidimensionaler Bilder gezeigt werden. Diese können in einer gedruckten Arbeit besser präsentiert werden als Volumen, die meist erst durch eine Rotation oder eine andere Art der Bewegung für den Betrachter verständlich werden. Des Weiteren ist es für das Verständnis der Arbeitsweise des vorgestellten Algorithmus unerheblich, von welcher Dimension die zu verarbeitenden Bilder sind.

Ein Vorteil der ersteren Bearbeitung von zweidimensionalen Bildern besteht zusätzlich in der Bewertung des Verfahrens bezüglich der Korrektheit, da im Rahmen der VIGRA bereits ein Algorithmus für zweidimensionale Bilder existiert, der diese auf die gleiche Weise bearbeitet.

Diese Tests sind allerdings nicht ausreichend, um den entwickelten Algorithmus in seiner Funktionalität zu untersuchen und zu beschreiben, da sie ebenfalls durch den bereits bestehenden Algorithmus zur Bearbeitung zweidimensionaler Bilder ausgeführt werden können. Daher wird ein weiterer Abschnitt folgen, welcher die Ergebnisse der Segmentierung von Volumen zeigt. Auch bei der Bearbeitung von relativ großen Volumendatensätzen soll das Verfahren gute und vor allem schnelle Ergebnisse liefern. Eine Bewertung dieser Aspekte wird deshalb ebenfalls später durchgeführt werden.

5.4.1 Segmentierung von zweidimensionalen Bildern

Zunächst einmal soll jedoch das Verfahren in seinen Einzelheiten beschrieben werden und seine Funktionsweise weiter verdeutlicht werden. Für Beispiele eignen sich zweidimensionale Bilder deutlich besser als Volumen, so dass solche Bilder ausgewählt werden.

Da das Verfahren allerdings nur wirkliche dreidimensionale Volumen bearbeiten kann, also keine Volumen, die nur aus einer Schicht bestehen und somit einem zweidimensionalen Bild entsprechen, bedarf es eines Tricks, um solche Bilder bearbeiten zu können. Dieser Trick ist notwendig, da die dreidimensionalen Nachbarschaften es nicht zulassen, dass ein Voxel an zwei gegenüberliegenden Rändern gleichzeitig liegt (zum Beispiel am vorderen Rand und am hinteren Rand, wenn das Volumen die Tiefe eins besitzt). Umgehen kann man dieses Problem, wenn man nach der Berechnung des Gradientenbildes dieses in ein Volumen einfügt, und als zweite Schicht entweder das gleiche Bild in aufgehellter Form oder ein einfarbiges Bild einfügt, dessen Grauwert höher als der höchste gefundene Gradient ist. So wird beim Finden des kleinsten Nachbarn sichergestellt, dass dieser auf der ersten Schicht liegt. Durch diese Maßnahme wird das Ergebnis nicht verfälscht. Das Ergebnis besteht in diesem Fall jeweils aus der ersten Schicht des Ausgabevolumens.

Ein Beispiel ist in Abbildung 44 dargestellt. Als Eingabe dient ein Bild mit zwei Regionen. Aus diesem wird das Gradientenbild gebildet, woraus dann die erste Vorverarbeitung des Union-Find-Wasserscheiden-Algorithmus erstellt wird. Die Ausgabe besteht in einem Bild, welches in zwei Regionen unterteilt ist. Man kann erkennen, dass Eingangs- und Ausgangsbild identisch sind (abgesehen von den Label-Werten). Für dieses Beispiel wurde das Bild in die gewünschten Regionen unterteilt. Damit der Algorithmus funktionierte, wurde die zweite Schicht des Gradientenvolumens mit einem hohen Wert gefüllt. Diese Maßnahme hatte zur Folge, dass im nächsten Schritt auf der ersten Schicht nur minimale Nachbarn auf dieser gefunden wurden.

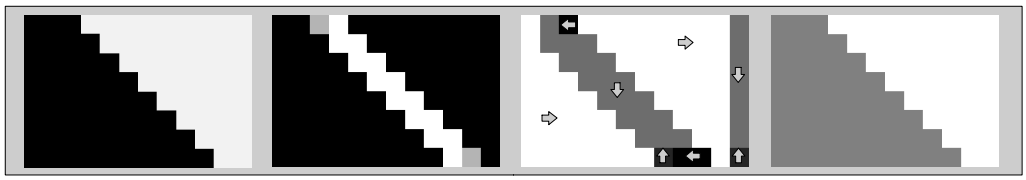


Abbildung 44: Einfaches Beispiel der Verarbeitungsschritte der Bildsegmentierung. Von links nach rechts: Ausgangsbild, Gradientenbild, Richtungsbild, Segmentierungsergebnis.

Wie aus dem obigen Beispiel ersichtlich, ist es möglich, Bilder zu verarbeiten, die nach der Faltung mit einer abgeleiteten Gauß-Funktion weiterhin Plateaus enthalten. Generell wird aber von absteigend vollständigen Eingabebildern ausgegangen. Die Probleme, die in der Praxis mit solchen Bildern auftreten können, und einige Lösungsstrategien dieser, werden im Abschnitt 5.4.3 beschrieben.

Um die korrekte Arbeitsweise eines Wasserscheiden-Verfahrens zu testen, bietet sich zudem noch eine weitere Klasse von Bildern an, nämlich die der Distanztransformationen. Dabei sei angemerkt, dass diese Bilder unmittelbar der Wasserscheiden-Transformation unterzogen werden, da eine Berechnung des Gradienten hier entfällt. In Abbildung 45 ist eine Distanztransformation zweier Punkte auf einer Ebene zu sehen.

Arbeitet ein Wasserscheiden-Verfahren korrekt, so muss es folgendes Ergebnis liefern:

1. Die Anzahl an gefundenen Regionen entspricht der Anzahl der Punkte der Distanztransformation.
2. Die Wasserscheide liegt jeweils genau in der Mitte zweier Punkte der Distanztransformation.

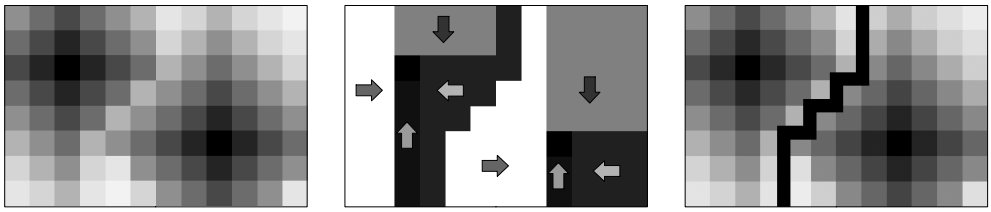


Abbildung 45: Das Union-Find-Wasserscheiden-Verfahren angewandt auf eine Distanztransformation zweier Punkte (oben links, unten rechts). Von links nach rechts: Ausgangsbild, Richtungsbild, resultierende Wasserscheide in CrackEdge-Darstellung, welche das Ausgangsbild in zwei Regionen unterteilt.

Wie in der obigen Abbildung sichtbar, liefert das Verfahren das gewünschte Ergebnis. Aus den zwei Punkten der Distanztransformation werden zwei Auffangbecken, die jeweils in der Mitte zusammentreffen und dort eine Wasserscheide bilden.

Nach diesen grundlegenden Eigenschaften, die anhand von Beispielen gezeigt wurden, wird nun ein weiterer wichtiger Einflussfaktor auf das Segmentierungsergebnis beschrieben: Die Wahl der Standardabweichung σ der verwendeten einfach abgeleiteten Gauß-Funktion, die zur Berechnung des Gradientenbildes verwendet wird. Wie schon in Abbildung 43 aufgezeigt, verändern sich die Gradientenbetragsbilder erheblich mit einer Änderung von σ.

Im Allgemeinen lässt sich festhalten, dass eine Vergrößerung von σ dazu führt, dass von Details des Bildes abstrahiert wird. Man könnte das so erhaltene Bild auch als „weichgezeichnet“ bezeichnen. Diese Abstraktion führt meist dazu, dass sich σ antiproportional zur Anzahl der, mittels der Wasserscheiden-Transformation gefundenen, Regionen verhält. Besonders gut lässt sich diese Auswirkung bei den in Abbildung 46 gezeigten Bildern erkennen. Bei der Wahl einer verhältnismäßig kleinen Standardabweichung sind die einzelnen Buchstaben noch klar voneinander zu unterscheiden. Es fallen mit ansteigender Standardabweichung viele Details weg, bis im untersten Teilbild schließlich nur noch die einzelnen Wörter unterschieden werden können.

Bei dieser Abbildung wurden die Wasserscheiden als Crack-Edges eingefügt und die einzelnen Regionen nicht mit der Farbe ihres Labels, sondern mit dem mittleren Grauwert aller enthaltenen Pixel der jeweiligen Region gefüllt. Dies vermittelt visuell eine bessere Einschätzbarkeit des Ergebnisses.

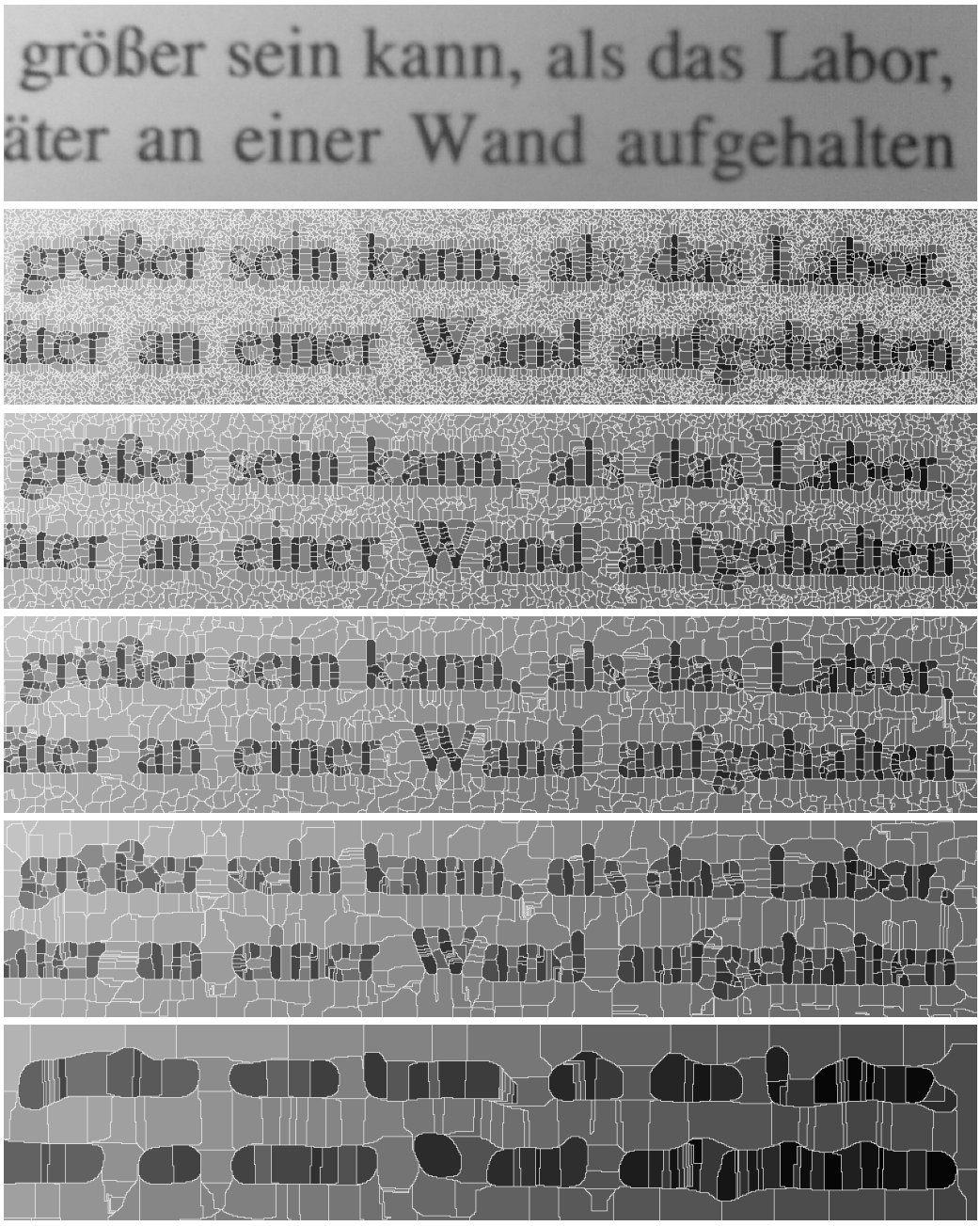


Abbildung 46: Auswirkung der Wahl der Standardabweichung σ auf das Ergebnis der Segmentierung mittels Wasserscheiden-Transformation. Von oben nach unten: Ausgangbild, Ergbnis mit: $\sigma=1$ (12405 Regionen), $\sigma=2$ (4269 Regionen), $\sigma=3$ (2001 Regionen), $\sigma=5$ (890 Regionen), $\sigma=10$ (236 Regionen).

Um noch einmal den Zusammenhang zwischen der Wahl der Standardabweichung und der Anzahl der gefundenen Regionen zu verdeutlichen, zeigt die folgende Abbildung 47 ein Diagramm, welches die Verhältnismäßigkeiten hervorhebt.

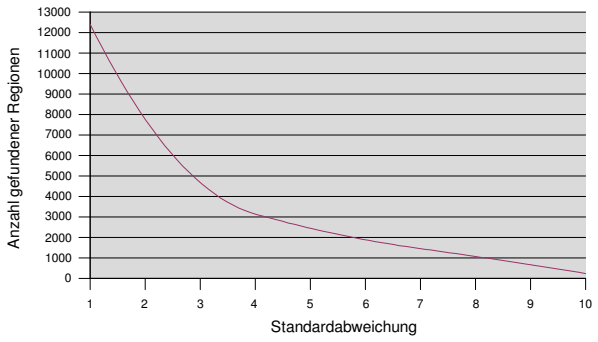


Abbildung 47: Verhältnis zwischen Standardabweichung der ersten Ableitung der Gauß-Funktion, mit der das Bild gefaltet wird, und der Anzahl der gefundenen Regionen.

Bei zweidimensionalen Bildern ist die Wahl der Standardabweichung bezüglich der Leistung nicht von großer Bedeutung. Allerdings kann es bei dreidimensionalen Bildern von entscheidender Bedeutung sein, einen verhältnismäßig hohen Wert zu wählen. Dieser Wert darf dennoch nicht so hoch liegen, dass Details, die als Ergebnis der Segmentierung gewünscht sind, verschwinden. Ein zu niedrig gewählter Wert hingegen verlangsamt die nachfolgenden Arbeitsschritte, die im Laufe dieser Arbeit ausgeführt werden. Dies geschieht aufgrund der Tatsache, dass, durch die mehr gefundenen Regionen, mehr Elemente und damit mehr Speicher benötigt wird. Diese Überlegungen leiten den nächsten Abschnitt ein, der sich mit der Segmentierung von Volumen mit Hilfe des implementierten Union-Find-Wasserscheiden-Verfahrens beschäftigt.

5.4.2 Segmentierung von Volumen

Während im vorigen Abschnitt Volumen künstlich hergestellt wurden, nämlich durch Einfügen eines Bildes in eine Schicht des Volumens, behandelt dieser Abschnitt die Segmentierung „echter“ dreidimensionaler Volumendaten. Wie schon einführend in Kapitel 3.1 erwähnt, stammen diese Datensätze meist aus medizinischen Aufzeichnungsgeräten und stellen eine dreidimensionale Repräsentation der gescannten Objekte dar.

Bevor jedoch mit diesen Beispielen aus der realen Welt begonnen wird, wird das Verfahren erneut auf eine Distanztransformation angewandt. Nur geschieht dies nun mit einem Volumen, in dessen Raum zwei Punkte eingebettet wurden. Die Größe des Volumens beträgt bei diesem Test $(100,100,100)$. Die Punkte liegen bei den Koordinaten $\vec{p}_1=(20,20,20)$ beziehungsweise $\vec{p}_2=(80,80,80)$. Analog zum vorigen Abschnitt lassen sich auch in Volumen Distanztransformationen dazu benutzen, um grundlegende Eigenschaften sicherzustellen. Die einzige Änderung an den vorher genannten Forderungen ist, dass nun keine Kante, sondern eine Ebene jeweils zwei

Regionen voneinander trennt. In dem oben angegebenen Beispiel wird eine trennende Fläche erwartet, die den Punkt $\vec{a}_p = (50,50,50)$ enthält und eine Normale von $\vec{a}_n = (1,1,1)$ besitzt. Dies ist auch tatsächlich der Fall, wie durchgeführte Tests ergeben haben.

Nach dieser Anwendung des Verfahrens auf dreidimensionale Distanztransformationen folgt der erste Test mit einem Volumen. Dazu wurde ein Volumen der Größe $(100,100,100)$ erstellt, welches zwei ineinander verschachtelte Würfel enthält. Die einzelnen Würfel liegen jeweils bei $BoundingBox_{w_1} = \{(20,20,20),(80,80,80)\}$ beziehungsweise bei $BoundingBox_{w_2} = \{(40,40,40),(60,60,60)\}$.

Nach einer Anwendung des implementierten Verfahrens auf dieses Volumen wird erwartet, dass drei Regionen gefunden werden, nämlich die des umgebenden Volumens, die des ersten Würfels und die des zweiten Würfels. Dies ist auch dann der Fall, wenn die Standardabweichung σ der verwendeten ersten Ableitung der Gauß-Funktion kleiner gleich drei ist. Wird die Standardabweichung höher gewählt, so treten am Rand des Volumens und innerhalb der einzelnen Würfel Scheinkonturen auf, die zu deutlich mehr als den gewünschten drei Regionen führen. Dieses Fehlverhalten liegt alleine in der Wahl der Standardabweichung begründet und zeigt keinen Fehler des Segmentierungsverfahrens auf. Abbildung 48 gibt einen Überblick über die Ergebnisse der Segmentierung des oben beschriebenen Volumens, in Abhängigkeit der Wahl von σ.

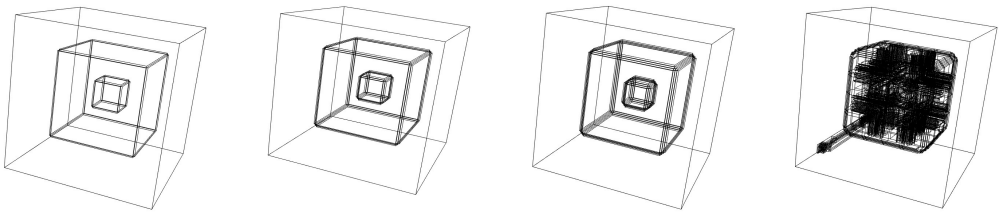


Abbildung 48: Verhältnis der verwendeten Standardabweichung σ auf das Ergebnis des implementierten Union-Find-Wasserscheiden-Verfahrens. Von links nach rechts: $\sigma = 1$, $\sigma = 2$, $\sigma = 3$ und $\sigma = 4$.

Wie sich anhand der obigen Abbildung erkennen lässt, findet auch bei dreidimensionalen Bildern ein Vorgang der „Weichzeichnung“ statt, und zwar in Abhängigkeit von der Wahl der Standardabweichung σ. Auf den ersten drei Bildern von Abbildung 48 lässt sich dies daran erkennen, dass die Kanten der Würfel mit zunehmenden σ immer weiter abgerundet werden.

Nach dieser Einführung in die Segmentierung von Volumen mit Hilfe des implementierten Wasserscheiden-Verfahrens, soll nun eine Anwendung mit einem Volumen der realen Welt aufgezeigt werden. In diesem Zusammenhang wird auch auf die Verarbeitungsgeschwindigkeit eingegangen. Dies wäre bei den bisher vorgestellten Volumen nicht sehr sinnvoll gewesen, zumal sie künstlich erzeugt wurden und somit keine objektive Beurteilung der praktischen Leistung des Verfahrens ermöglichen.

Das erste verwendete Volumen ist die Computertomographie-Aufnahme einer Teekanne[54], in deren Innenraum sich ein in Kunststoff eingegossener Hummer befindet. Dieses Volumen besitzt die Größe $(256,256,178)$. Die folgende Abbildung 49 zeigt zwei verschieden gerenderte Ansichten dieses Volumens, damit eine Vorstellung dessen entsteht.

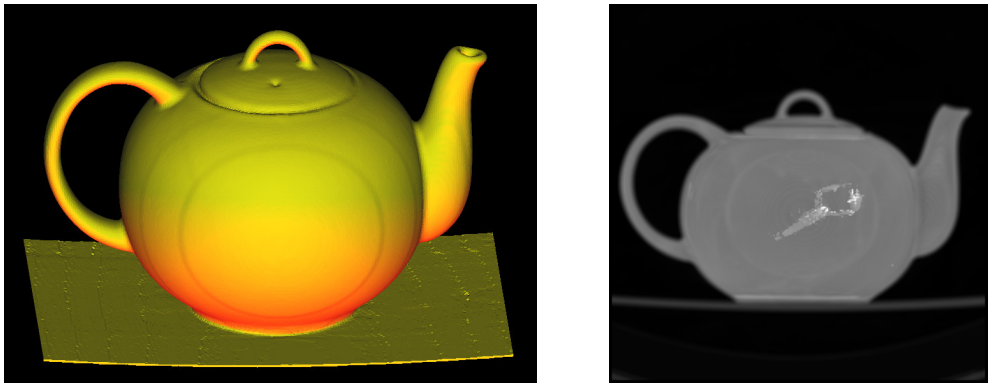


Abbildung 49: Das für diesen Test verwendete Volumen: Der Boston Teapot. Links: Ein Oberflächenrendering, rechts eine Maximum-Intensitäts-Projektion.

Das Volumen, welches das Ergebnis des Segmentierungsverfahrens darstellt, lässt sich im Rahmen dieser Arbeit nicht angemessen präsentieren, da zum einen sehr viele Regionen gefunden werden und zum anderen ebenfalls sehr viele Schichten für die Präsentation abgebildet werden müssten. Aus diesen Gründen werden indiesem Kapitel nur einige ausgewählte Schichtenbilder des Segmentierungsergebnisses vorgestellt. Zunächst wird dabei erneut die Standardabweichung der ersten Ableitung der Gauß-Funktion variiert, und das Ergebnis anhand einer Schicht der Volumens veranschaulicht.

54 Diese Teekanne heißt aufgrund ihrer Herkunft auch „Boston Teapot“ und ist eins der bekanntesten Volumen in der dreidimensionalen Bildverarbeitung.

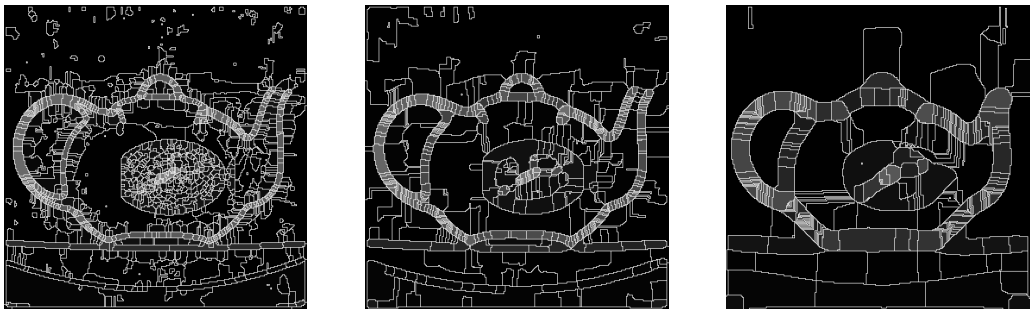


Abbildung 50: Der Algorithmus angewandt auf das "Boston Teapot"-Volumen (am Beispiel der Schicht mit der Nummer 89) zeigt die Auswirkung der Wahl der Standardabweichung (von links nach rechts: $\sigma = 1$, $\sigma = 3$ *und* $\sigma = 7$ *) auf das Segmentierungsergebnis.*

Wie schon vermutet werden konnte, zeigt sich auch bei realen Volumen, dass die Wahl der Standardabweichung sowohl die Anzahl der gefundenen Regionen als auch die segmentierten Details bestimmt. Dieses setzt sich hierbei in allen Dimensionen fort und ist dementsprechend auch so vorzustellen. Leider vermögen die abgebildeten Schichtenbilder dies nicht zu zeigen. Um dennoch einen Eindruck über die Anzahl der gefundenen Regionen zu bekommen, werden diese im Folgenden in Zusammenhang mit der Verarbeitungsgeschwindigkeit vorgestellt.

Die Verarbeitungsgeschwindigkeit ist eine weitere wichtige Eigenschaft des Segmentierungsverfahrens. Da das implementierte Verfahren einen zu der Größe des Volumens proportionalen Speicherverbrauch aufweist, es wird genau der doppelte Speicher des Ausgangsvolumens benötigt, wird an dieser Stelle auf eine Auflistung des Speicherverbrauches verzichtet. Da es für die Einschätzung der Rechengeschwindigkeit des Verfahrens notwendig ist, verschiedene Volumen zu betrachten, wird noch ein weiteres Volumen benötigt. Dieses besitzt lediglich eine Größe von $(128,128,62)$ und eignet sich somit, um den Einfluss der Volumengröße auf die Geschwindigkeit des Verfahrens zu dokumentieren. Dieses Volumen (der CT-Scan eines Teddybären) ist in Abbildung 51 gezeigt.

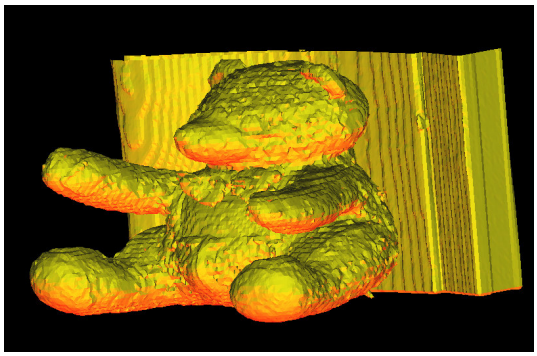


Abbildung 51: Oberflächenrendering des verwendeten Teddybär-Volumens. Es besteht im Gegensatz zu dem „Boston Teapot"-Volumen aus sehr viel weniger Voxeln.

Um die Geschwindigkeit des Verfahrens sowohl in Abhängigkeit von der Anzahl der gefundenen Regionen als auch in Bezug auf die Größe des Volumens beurteilen zu können, wurden bei beiden Volumen die Standardabweichungen der ersten Ableitung der Gauß-Funktion variiert.

Die Anzahl der gefundenen Regionen in Abhängigkeit von der gewählten Standardabweichung σ kann aus der folgenden Abbildung 52 entnommen werden. Hierbei fällt auf, dass die Anzahl der gefundenen Regionen bei dem Teddybär-Volumen absolut betrachtet sehr viel schneller abnimmt, als bei dem des „Boston Teapot". Dies ist damit zu erklären, dass eine stetige Erhöhung von σ in einem kleineren Volumen dazu führt, dass die Regionen stetig größer werden. Dadurch werden in einem kleineren Volumen recht schnell viele Regionen durch Weichzeichnung miteinander verschmolzen.

Standard-abweichung	Anzahl segmentierter Volumen	
	Boston Teapot	Teddybear
1	73201	52942
2	50591	11002
3	35869	3818
5	23345	1014
7	17164	331
10	14734	

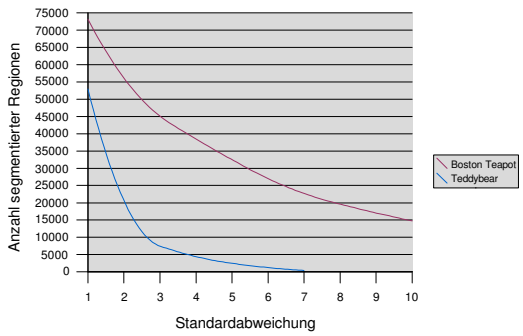


Abbildung 52: Verhältnis zwischen der Volumengröße und der Anzahl der segmentierten Regionen in Abhängigkeit von der Standardabweichung. Bei $\sigma = 10$ ist die Faltungsmaske zu groß, um auf das Teddybär-Volumen angewandt werden zu können.

Nach dieser ersten Beurteilung des Einflusses der Volumengröße auf die Anzahl der gefundenen Regionen, wird nun die benötigten Rechendauer beschrieben. Dazu werden

beide Volumen mit variierenden Standardabweichungen segmentiert. Im Anschluss daran wird die Zeit genommen, die der Algorithmus benötigt hat, um das entsprechende Volumen zu segmentieren.

Diese Zeitnahme erfolgte auf einem Testsystem mit folgender Konfiguration:

- Prozessor: Intel® Pentium® M 1,6 GHz
- Arbeitsspeicher: 1280 MB DDR RAM
- Betriebssystem: Microsoft® Windows XP Professional
- C++ Compiler: Microsoft® Visual C++ .net® 1.1
- Compiler-Optimierungen: Komplette Optimierung.

Die weiteren Systemkomponenten werden an dieser Stelle nicht mit aufgezählt, da sie keinen nennbaren Einfluss auf die gemessenen Zeiten haben.

Abbildung 53 zeigt zum einen die realen Zeiten, die gemessen wurden, zum anderen wird auch die durchschnittliche Zeit der Verarbeitung von einer Million Voxel angezeigt. Wie zu erwarten war, ist diese Zeit relativ konstant und unabhängig von der Volumengröße. Dies ist wichtig, da die meisten Verfahren bei steigender Volumengröße einen überproportional ansteigenden Zeitbedarf bei der Verarbeitung einzelner Voxel haben.

Standard-abweichung	Benötigte Rechendauer		Rechendauer pro 1.000.000 Voxel	
	Boston Teapot	Teddybär	Boston Teapot	Teddybär
1	3,48	0,33	0,3	0,32
2	3,49	0,32	0,3	0,32
3	3,54	0,3	0,3	0,3
5	3,52	0,31	0,3	0,31
7	3,55	0,3	0,3	0,3
10	3,57		0,31	

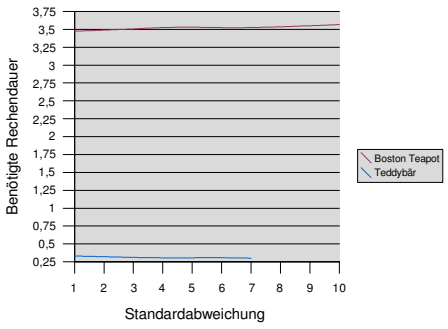


Abbildung 53: Der Zusammenhang von benötigter Rechendauer des Verfahrens und der Größe des Volumens unter dem Einfluss variierter Standardabweichungen (und damit auch unterschiedlichen Anzahlen von Regionen siehe Abbildung 52).

Die Analyse des implementierten Verfahrens, sowie die Anwendung an einigen Beispielen, hat noch einmal verdeutlicht, was dieses Verfahren auszeichnet und wo die Stärken beziehungsweise Schwächen liegen. Eine starke Übersegmentierung wird im Rahmen dieser Arbeit in Kauf genommen, zeichnet sich das Verfahren doch andererseits durch einen sehr günstigen Zeit- und Speicherbedarf aus. Dennoch darf nicht unerwähnt bleiben, dass eine sehr starke Übersegmentierung (>10.000 Regionen) zusammen mit einem verhältnismäßig großen Volumen ($>(256,256,256)$) in Bezug auf die nachfolgenden Verarbeitungsschritte keineswegs unproblematisch ist. Hier muss vorher der Wert der Standardabweichung der ersten Ableitung der Gauß-Funktion entsprechend angepasst werden.

Nach dieser Beurteilung der Leistung wird im Folgenden auf die Anmerkungen in Abschnitt 5.3.1 eingegangen. Es wird erläutert, wie sich die einzelnen Strategien des Umgangs mit Plateaus in Bildern auf die Segmentierungsergebnisse auswirken.

5.4.3 Anwendung auf Bilder mit Plateaus

Besitzt ein zu segmentierendes Bild Plateaus, so müssen diese eliminiert werden, damit das Ergebnis der Wasserscheiden-Transformation mittels des Union-Find-Verfahrens genau der formalen Definition entspricht.

Wie bereits in Abschnitt 5.3.1 aufgezählt, gibt es aber auch noch andere Möglichkeiten, um mit Plateaus in Bildern umzugehen, die keine aufwendige Vorverarbeitung, wie zum Beispiel die absteigende Vervollständigung, benötigen. Werden diese Verfahren angewandt, so resultieren daraus einige unerwartete Verhaltenseigenschaften des Verfahrens. In diesem Abschnitt werden diese Eigenschaften nicht nur beschrieben, es wird auch deren Ursache aufgezeigt werden. Dabei wird zunächst mit der einfachsten Strategie begonnen: Die Plateaus werden ignoriert und somit wie normale Regionen behandelt. Danach wird die Strategie erläutert, die den Graphen des Bildes während der Erstellung manipuliert.

5.4.3.1 Ignorieren von Plateaus

Wird ein Bild mit Plateaus in das bestehende Verfahren eingelesen, so wird zunächst dessen Richtungsbild, welches den DAG des Bildes bestimmt, erstellt. Dazu wird in der Nachbarschaft eines Bildpunktes nach Bildpunkten gesucht, die einen Wert haben, der kleiner gleich dem Wert des aktuellen Bildpunktes ist. Ist ein solcher Bildpunkt gefunden, so wird die Richtung auf diesen gesetzt. Zudem müssen alle im Folgenden betrachteten Bildpunkte der Nachbarschaft kleiner gleich dem Wert des letzten gefundenen Nachbarn sein.

Befindet sich ein Bildpunkt nun inmitten eines Plateaus, so werden diese Vergleiche für alle seine Nachbarn erfolgreich sein. Doch liegt an dieser Stelle ein Informationsverlust vor. In Abhängigkeit von der definierten Nachbarschaft bleibt der Algorithmus immer am letzten Nachbarn stehen und speichert dessen Richtung für den aktuellen Bildpunkt in das Richtungsbild ab. Diesen Vorgang veranschaulicht Abbildung 54.

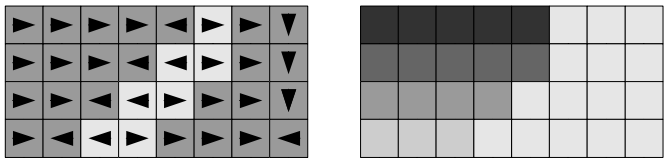


Abbildung 54: Plateau-Darstellungen im DAG des Bildes. Links: Der Graph des Bildes; rechts: Die mit dem Union-Find-Verfahren gefundenen Regionen. (verwendete Suchreihenfolge der 4er-Nachbarschaft: North, West, South, East)

Welche Auswirkungen diese Auswahl der Richtung auf das Ergebnis hat, kann man sich leicht vorstellen: Im schlimmsten Fall werden Plateaus in viele einzelne Regionen

unterteilt erkannt. Da diese in Abhängigkeit der letzten Richtung der definierten Nachbarschaft verlaufen, besitzen sie jeweils lediglich die Breite eins. Sie treten immer dann auf, wenn die Charakteristik der Grenze zwischen zwei Plateaus so verläuft, dass sie nicht in der Lage ist, die einzelnen Streifen der Plateaus miteinander zu verbinden.

Da in Bildern generell Kanten unterschiedlicher Gestalt auftreten können, gibt es mit der hier vorgestellten Strategie des Ignorierens von Plateaus keine Möglichkeit das Auftreten von Streifen-Regionen zu unterbinden. Die im folgenden Abschnitt vorgestellte Strategie vermag dies, misst Plateaus zu diesem Zweck aber eine neue Bedeutung zu.

5.4.3.2 Veränderung der Erstellung des Graphens des Bildes

Das Auftreten von Plateaus in Bildern kann viele Ursachen haben. Dementsprechend können Plateaus selbst auch unterschiedlich interpretiert werden. Einerseits treten Plateaus in der Nähe eines lokalen Minimums auf, andererseits können sie auch in der Nähe von Maxima vorkommen. Die folgende Abbildung 55 veranschaulicht die beiden Fälle und das gewünschte Ergebnis. Dabei ist festzuhalten, dass Plateaus in der Nähe lokaler Minima als eine Region angesehen werden sollen, wohingegen Plateaus, welche sich in der Nähe eines Maximums befinden, eventuell als mehrere Regionen repräsentiert werden sollen.

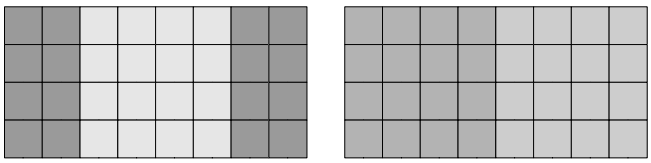


Abbildung 55: Durch das Wasserscheiden-Verfahren nach Definition 5.2.2 erwartet man für das linke Bild, welches sowohl Minimum- als auch Maximumplateaus besitzt, das rechte Bild als Ergebnis der Segmentierung.

Nimmt man in Kauf, dass Plateaus in der Nähe von Maxima ebenfalls als eine Region erkannt werden, so lässt sich die Erstellung des Richtungsbildes so ändern, dass dieses, zumindest in Bereichen von Plateaus im Bild, keinen gerichteten azyklischen Graphen mehr darstellt. Betrachtet man allerdings die Plateaus als Singularitäten[55], so stellt der Graph wieder einen DAG dar, so wie es gewünscht ist. Es sei an dieser Stelle darauf hingewiesen, dass in der Praxis – durch die Erstellung des Gradientenbetragsbildes – Plateaus in der Nähe lokaler Minima häufig vorkommen. Dies stellte die Motivation für die hier vorgestellte Strategie dar. Die folgende Abbildung 56 zeigt, wie sich der Graph des Bildes durch diese Sichtweise auf Plateaus verändert.

55 Diese Sichtweise auf Plateaus ist ohnehin sinnvoll, wenn davon ausgegangen wird, dass ein Plateau eines Bildes auch als eine Region segmentiert wird.

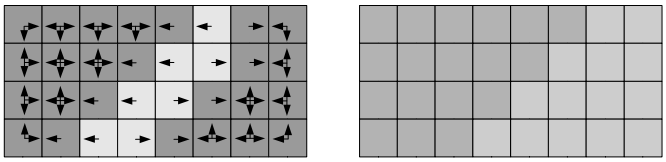


Abbildung 56: Links:Der Graph verändert sich bei gleicher Nachbarschaft und Suchreihenfolge der Nachbarn gegenüber dem in Abbildung 54; rechts: Das dadurch erreichte Segmentierungsergebnis.

Durch diese Sichtweise ergibt sich allerdings auch ein neues Problem. Dicke Kanten, also Kanten mit Plateaus, werden nun so behandelt, als gäbe es sie nicht. Doch diese Kanten tauchten in der Praxis in Tests mit erstellten Gradientenbildern nicht auf, auch wenn man die Standardabweichung der abgeleiteten Gauß-Funktion variierte. Dennoch kann ein Auftreten dieses Problems auch bei realen Bildern nicht ausgeschlossen werden. Ein Beispiel für das Verschmelzen über ein Maximum hinweg zeigt die folgende Abbildung 57.

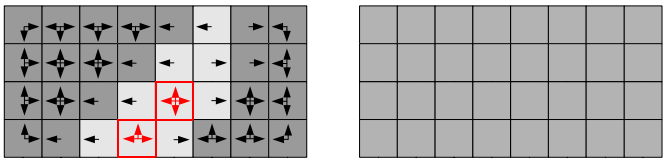


Abbildung 57: Die in rot eingefärbten Bildpunkte liegen inmitten eines Maximum-Plateaus und sorgen so dafür, dass die beiden getrennten Regionen links und rechts fälschlicherweise miteinander verbunden werden. Das rechte Bild zeigt das Ergebnis der Segmentierung.

Man mag sich nun fragen, welche der Strategien zur Behandlung von Plateaus die beste ist. Dieses kann jedoch nicht allgemein festgelegt werden. Es sollte vielmehr vom Anwendungszweck abhängen. Folgende Punkte lassen sich festhalten:

- Möchte man auf einem Bild eine Wasserscheiden-Transformation durchführen, welche exakt der Definition 5.2.2 entspricht, so müssen die Plateaus aus dem Bild entfernt werden. Dies kann entweder mit der absteigenden Vervollständigung erreicht werden, oder mit einem Aufaddieren von Rauschen auf das Bild.
- Ist es kein Problem, dass Plateaus durch das Verfahren in mehrere Regionen segmentiert werden, so kann das Ausgangsbild direkt an das Verfahren übergeben werden. Damit könnten allerdings sehr viele *überflüssige Regionen* gefunden werden.
- Soll sichergestellt werden, dass um Minima liegende Plateaus als zusammenhängende Regionen erkannt werden, so kann das in 5.4.3.2 beschriebene Verfahren angewandt werden. Dabei können allerdings Regionen

trotz einer Grenze[56] miteinander verschmolzen werden. Daher könnten *weniger Regionen* als gewünscht gefunden werden.

Soweit nicht anders angegeben, wurde in diesem Abschnitt die letzte Variante des Verfahrens auf die Beispielbilder angewandt.

56 Diese Grenze stellt dann eine dicke Kante dar, das heißt, dass sie mindestens einen Bildpunkt besitzt, in dessen Nachbarschaft alle Nachbarn den gleichen Wert wie er selbst besitzen.

6 Implementation der 3-G-Map

In diesem Kapitel wird die Umsetzung des GMAP-Datentyps beschrieben. Dazu gehört zum einen seine Erstellung, aber auch die Schnittstellen, die das Arbeiten mit diesem Datentyp ermöglichen. Dabei werden alle benötigten Datentypen zunächst dimensionsunabhängig definiert, um eine generische Struktur zu ermöglichen. Erst anschließend werden Spezialisierungen ausgeprägt, wie zum Beispiel die GMAP3D, welche eine Umsetzung der 3-G-Map darstellt.

Wie bereits in Kapitel 3 deutlich wurde, sind kombinatorische Karten, und speziell die generalisierte kombinatorische Karte, durchaus gebräuchliche Repräsentationen, um Oberflächen topologisch beschreiben zu können. Allerdings bezieht sich die Literatur häufig nur auf die Untersuchung von Flächen und somit Oberflächen in zweidimensionalen Bildern. Die Bearbeitung von dreidimensionalen Daten hat in den letzten Jahren allerdings zugenommen (siehe beispielsweise [LM99]), seitdem die benötigte Hardware vorhanden ist. Somit ist die in dieser Arbeit beschriebene Umsetzung einer 3-G-Map nicht die erste ihrer Art. Es sei allerdings hervorgehoben, dass die Erstellung, so wie sie in diesem Kapitel beschrieben wird, eine Neuentwicklung darstellt, die bisher so noch nicht in der Literatur zu finden ist.

Der in dieser Arbeit entwickelte Ansatz der Erstellung soll vor allem die Bearbeitung von Datensätzen ermöglichen, deren Umsetzung in eine 3-G-Map die Erzeugung sehr vieler Darts erfordert. Wie in Abschnitt 6.2 deutlich wird, werden mögliche Darts schon vor der Erstellung der GMAP3D ausgeschlossen, wenn sie für eine Border-Map nicht benötigt werden.

Bevor allerdings auf den Erstellungsvorgang präziser eingegangen wird, folgen in den nächsten Abschnitten zunächst die abstrakten Datentypen mit ihren Schnittstellen. Diese stellen die Grundlage der Umsetzung dar und müssen daher vor den weiteren Schritten geklärt sein.

Am Ende dieses Kapitels gibt es einen Abschnitt, der verschiedene Möglichkeiten nennt, wie man mit der Datenstruktur einer GMAP3D arbeiten kann. Diese sind besonders wichtig, da die Datenstruktur kaum explizite Informationen liefert.

6.1 Charakteristik der verwendeten Datenstruktur

Die hier vorgestellte GMAP trägt eine Menge an Informationen, zu denen nicht nur topologische, sondern auch geometrische gehören. Damit der Datentyp der GMAP nicht zu unübersichtlich wird, benutzt er einen speziell für ihn entwickelten weiteren Datentyp, der zum Speichern dieser Informationen dient.

6.1.1 Der abstrakte Datentyp DART

Ein Dart ist die kleinste Einheit, aus der eine G-Map aufgebaut ist. In ihm finden sich viele grundlegende Informationen, die erst den Zusammenhang einer G-Map ermöglichen. Denn wie bereits in Kapitel 3 ausführlich beschrieben, definiert sich eine G-Map über eine Menge von Darts und einigen Involutionen, die in diesem Zusammenhang Orbits genannt werden. Die Orbits sind über die Darts definiert. Bei dieser Umsetzung trägt deshalb jeder Dart explizite Informationen darüber, welcher andere Dart über welches Orbit mit ihm vernäht ist.

Die Klasse DART, die in diesem Abschnitt näher erläutert wird, ist eine Template-Klasse. Dadurch ist es möglich, Darts für verschiedene Dimensionen zu definieren. Der zu übergebende Template-Parameter `dimension` muss vom Typ Integer sein und zudem zwischen eins und drei liegen.

Folgende weitere Anforderungen werden in dieser Arbeit an den ADT DART gestellt:

1. Ein Dart sollte sich für unterschiedliche Dimensionen definieren lassen.

   Zwar wird für diese Arbeit nur ein dreidimensionaler Dart verwendet, aber eine Erweiterung sollte möglich sein.

2. Da keine weitere Struktur für die Einbettung der geometrischen Informationen vorliegt, muss der Dart in der Lage sein, diese Informationen zu speichern.

3. Für eine übersichtliche Darstellung in der Visualisierung (siehe Kapitel 8) und verschiedene Anwendungen der 3-XG-Map muss ein Dart auf verschiedene Arten markierbar sein.

4. Ein Dart braucht Informationen über die mit ihm über Orbits vernähten anderen Darts.

5. Jeder Dart benötigt einen eindeutigen Bezeichner.

Damit der Datentyp in der Lage ist, diesen an ihn gestellten Anforderungen gerecht zu werden, werden die folgenden Member-Variablen benötigt:

- `int dartID`
- `vector<dimension> embedding`
- `Dart<dimension>* beta[dimension+1]`
- `unsigned int data`

Um sicherzustellen, dass jeder Dart einen eindeutigen Bezeichner erhält, bekommt jeder Dart diesen in Form einer DartID zugeteilt.

Definition 6.1.1 (DartID) Sei g eine Funktion $g: B \to \mathbb{Z}$, die jedem Dart d aus der Menge B aller Darts einen eindeutigen Wert aus $\mathbb{Z}$ zuweist. Sind d_0 und d_1 zwei Darts aus der Menge B mit $d_0 \neq d_1$, dann gelte $g(d_0) \neq g(d_1)$. $g(d)$ heißt DartID vom Dart d.

In der Implementation sei die Member-Variable `dartID` dieser eindeutige Wert für jeden Dart. Damit sichergestellt ist, dass es keine zwei Darts mit der selben DartID gibt, lässt sich diese für den dreidimensionalen Fall aus der Position des Darts im Volumen berechnen. Im Kapitel zur Erstellung der GMAP (siehe Kapitel 6.2) wird die Berechnung im Detail beschrieben.

Die DartID wird schon bei der Erstellung eines Dart-Objektes benötigt. Das bedeutet, dass der Konstruktor einen Integer-Wert verlangt, der die Identifikationsnummer des Dart-Objektes darstellt. Ein späteres Ändern der DartID ist ausgeschlossen. Wählt man die Funktion zur Vergabe von DartIDs so aus, dass sie aus der Position eines Darts im Bilddatenraum dessen ID berechnet und umkehrbar ist, so lassen sich die zur Berechnung verwendeten Werte bestimmen. Man kann beispielsweise mit Hilfe der Volumendimensionen aus der DartID die Koordinaten des Voxels bestimmen, für den der Dart erstellt wurde. Eine sehr wichtige Funktion hat die DartID bei der Umwandlung der VOLUMEMAP zur GMAP3D (siehe Abschnitt 6.2.3). Eine weitere Anwendung der DartID ist der einfache Gleichheitstest zweier Darts.

Bei den Anforderungen an den ADT DART wurde die Speicherung der geometrischen Einbettung im Dart gefordert.

Definition 6.1.2 (geometrische Einbettung) Sei h eine Funktion $h: B \to \vec{a}$, die jeden Dart d der Menge B auf einen Vektor $\vec{a}$ der Dimension des Darts abbildet. $\vec{a}$ stellt dabei die geometrische Lage des Darts im Raum dar.

Die geometrische Einbettung geschieht durch die Speicherung der Koordinaten eines Raumpunktes in einem TINYVECTOR[57]. Dieser Raumpunkt stellt somit eine Verankerung des Darts dar und hält den Bezug zu den geometrischen Informationen im Ausgangsbild.

Eine andere Möglichkeit wäre gewesen, Knoten zu erstellen, die die Raumkoordinaten speichern. Die Darts könnten dann auf diese Knoten verweisen. Dies würde eine Speicherersparnis bedeuten, da dann jeder Vektor nur einmal gespeichert werden müsste, auch wenn er einen Verankerungspunkt für bis zu 24 Darts darstellt. In der Praxis sind aber nur selten 24 Darts an einem Knoten vorhanden. Aus diesem Grund ist die Speicherersparnis sehr viel geringer, als es den Anschein hat. Zudem würde sich der Rechenaufwand geringfügig erhöhen, da bei der Erstellung der Darts der passende Knoten herausgefunden werden müsste. Des Weiteren müsste bei der Löschung von Darts immer beachtet werden, ob auf einen Knoten noch verwiesen wird oder nicht. Sollte keine Referenz mehr auf einen Knoten vorliegen, so müsste er gelöscht werden. Insgesamt ergibt sich daher in der Praxis ein Vorteil für den hier verwendeten Ansatz, zumal diese Art der Speicherung den Zeichnungsvorgang (siehe Kapitel 8.1) der Darts deutlich beschleunigt.

Die nächste Member-Variable ist ein Array von Dart-Zeigern. Dieses Array hat die Aufgabe, die über die Orbits vernähten Darts für den Dart zu speichern. Deshalb hat es

57 siehe [Vig06d]

auch eine Größe von `dimension+1`, denn wie bereits in Abschnitt 3.4.3 beschrieben, braucht man für eine G-Map ein Orbit mehr, als die Anzahl der Dimensionen beträgt. In dem Array `beta` werden Zeiger auf Darts gespeichert und nicht etwa die DartID von den entsprechenden Darts. Dies wäre zwar auch denkbar, ist aber sehr viel aufwendiger, da man beim Zugriff erst den passenden Dart zur DartID finden müsste. Über den Zeiger hat man hingegen direkten und somit schnellen Zugriff auf den vernähten Dart.

Für die Verwendung einer G-Map ist es sehr wichtig, dass man Darts auf gewisse Weise markieren kann, um beispielsweise anzuzeigen, dass an diesem Dart schon eine Bearbeitung vorgenommen wurde. Hierfür reichen in der Regel Bool'sche Werte. Da aber in einigen Fällen nicht nur eine Markierungsmöglichkeit benötigt wird, ist es sinnvoll, alle in einer Variable zu speichern. Hierfür wird ein `unsigned int` mit dem Namen `data` zur Verfügung gestellt, von dem jedes Bit als Bool'scher Wert aufzufassen ist. Der Zugriff auf die verschiedenen Bits erfolgt über spezielle Schnittstellen und nicht direkt auf den Wert. Da in `data` nicht nur Markierungen eines Darts gespeichert werden, stehen dafür nur die niederen 28 Bits zur Verfügung. Das bedeutet, dass ein Dart auf 28 verschiedene Weisen markiert werden kann. Dabei dient das niederste Bit nur zur Markierung des ausgewählten Darts in der Anzeige einer GMAP (siehe Kapitel 8). Die höchsten vier Bits werden benutzt, um einen Dart als Anker (vgl. Kapitel 3.4) zu markieren.

Die Member-Variablen sind alle als private Variablen definiert. Somit erfolgt der Zugriff auf sie nur über Schnittstellen. Durch diese Maßnahme wird ein unerlaubter Zugriff auf die Variablen verhindert, des Weiteren kann man nur soweit auf die Variablen zugreifen, wie die Schnittstellen es zulassen. Die für den Dart zur Verfügung stehenden Schnittstellen sind in Tabelle 6.1.1 erläutert.

Tabelle 6.1.1: Schnittstellen der Klasse DART

`int getDartID()`
: erlaubt die Abfrage der DartID.

`Dart<dimension>* getBetaOrbit(int)`
: liefert einen Zeiger auf den Dart des entsprechenden Orbits zurück. Welches Orbit verwendet werden soll, kann über den Übergabewert angegeben werden. Der Übergabewert sollte nicht größer als `dimension+1` sein, da dafür keine Orbits definiert sind.

`setBetaOrbit(int, Dart<dimension>*)`
: ordnet einem Orbit einen Zeiger auf einen Dart zu. Welches Orbit verändert werden soll, kann über den ersten Übergabewert angegeben werden. Der Übergabewert sollte nicht größer als `dimension+1` sein, da dafür keine Orbits definiert sind. Der zweite Übergabewert ist der entsprechende Zeiger auf einen Dart der entsprechenden Dimension.

`vector getEmbedding()`
: liefert einen Vektor mit den Koordinaten der geometrischen Einbettung zurück.

`mark()`
: markiert einen Dart, indem das zweite Bit von `data` auf eins gesetzt wird.

`unmark()`
: setzt das zweite Bit von `data` auf null.

`unmarkActive()`
: setzt die niedersten sechs Markierungsbits (Bit 0-5) von `data` auf null.

`unmarkAll()`
: setzt die Markierungsbits (Bit 0-27) von `data` auf null.

`bool isMarked()`
: liefert `true` zurück, wenn mindestens eines der niederen 28 Bits von `data` gesetzt ist, ansonsten `false`.

`mark(unsigned int)`
: erlaubt es, ein bestimmtes Bit von Data auf eins zu setzen. Zugriff wird nur bis Bit 27 gewährt.

`bool isMarked(unsigned int)`
: liefert `true` zurück, wenn an der übergebenen Position das Bit von `data` gesetzt ist, ansonsten `false`.

`bool isActive()`
: liefert `true` zurück, wenn mindestens eines der sechs niedersten Bits von `data` gesetzt ist, ansonsten `false`.

`unsigned int getMark()`
: liefert die niederen 28 Bit von `data` als Wert zurück.

`unsigned int getLowestMark()`
: liefert die kleinste Position eines gesetzten Bits von `data` zurück. Die Funktion sollte nur aufgerufen werden, wenn sichergestellt ist, dass etwas markiert ist. Dies kann mit `isMarked()` geprüft werden.

`setCellKey(unsigned char)`
: setzt den Zellschlüssel an der übergebenen Position für diesen Dart auf eins. Der Parameter darf nicht größer als `dimension+1` sein.

`resetCellKey(unsigned char)`
: setzt den Zellschlüssel an der übergebenen Position für diesen Dart auf null zurück. Der Parameter darf nicht größer als `dimension+1` sein.

`bool isCellKey(unsigned char)`
: liefert `true` zurück, wenn der Dart ein *n*-Zellschlüssel ist, ansonsten `false`. Der Parameter, der *n* darstellt, darf nicht größer als `dimension+1` sein.

Neben diesen Schnittstellen werden noch zwei Operatoren zur Verfügung gestellt. Diese werden in Tabelle 6.1.2 vorgestellt.

Tabelle 6.1.2: Operatoren von DART

`operator==(Dart<dimension>)`
: untersucht zwei Darts auf Gleichheit. Dazu wird die Gleichheit der Member-Variable `dartID` beider Darts geprüft.

`operator!=(Dart<dimension>)`
: untersucht zwei Darts auf Ungleichheit.

Der durch die obigen Schnittstellen beschriebene abstrakte Datentyp Dart stellt die Grundlage für alle weiteren Strukturen dieser Arbeit bereit. Er ermöglicht es, auf einfache Art zwischen vernähten Darts zu wechseln. Ebenso ermöglicht er eine Vielzahl von verschiedenen Markierungen, die in den folgenden Kapiteln noch von Interesse sein werden. Die geometrische Einbettung stellt den Grundbaustein für die Visualisierung bereit, welche in Kapitel 8 beschrieben werden wird.

6.1.2 Der abstrakte Datentyp GMAP

Eine 3-G-Map besteht, wie schon in Abschnitt 3.4.3 beschrieben, aus einer Menge von Darts und $\beta_0, \ldots, \beta_3$ -Orbits, welche Involutionen auf den Darts darstellen. Die Orbits sind schon explizit in dem abstrakten Datentyp DART repräsentiert, sodass sie nicht noch einmal separat für den Datentyp GMAP gespeichert werden müssen (vgl. Abschnitt 6.1.1).

Die einfachste Form der Implementation wäre also eine Liste von Exemplaren des abstrakten Datentyps DART. Diese Liste stellt auch in der Implementation der GMAP die Basis dar, allerdings wurde sie noch erweitert um folgenden Ansprüchen zu genügen:

1. Sie sollte sich – einmal definiert – auch einfach für andere Dimensionen anwenden lassen.
2. Es sollte globale Operationen für alle Darts geben, wie zum Beispiel: „Lösche alle Markierungen aller Darts“.
3. Anhand der DartID eines Darts sollte die dreidimensionale Spezialisierung der GMAP den zugehörigen Voxel des Ausgangsvolumens errechnen können, dem dieser Dart bei der Erstellung der GMAP angehörte (siehe hierzu auch Kapitel 6.2).

Der erste Punkt erfordert einen generischen Implementationsansatz – dies stellt jedoch keine große Herausforderung dar, da bereits der Dart als generische Datenstruktur implementiert wurde. Die globalen Operationen wurden als Member-Funktionen umgesetzt und finden sich in der Beschreibung der Schnittstelle der GMAP in Tabelle 6.1.3. Für den dritten Punkt wurde die teilweise Spezialisierung der GMAP für den dreidimensionalen Fall um eine Member-Funktion erweitert. Außerdem enthält eine GMAP noch Informationen über die Größe der Datenstruktur, aus welcher sie erzeugt wurde. Dies ist vor allem für weitere Spezialisierungen wichtig, wie zum Beispiel die bereits oben erwähnte Funktionalität, die an die dreidimensionale Variante gestellt wird.

Um die im obigen Abschnitt erwähnte Funktionalität zu ermöglichen, gliedert sich die Implementation in eine Template-Basisklasse, mit Template-Parameter `dimension`, welcher die Dimension der GMAP angibt, und die eigentliche GMAP-Klasse, welche ebenfalls eine Template-Klasse beschreibt. Die Basisklasse enthält bereits alle nötigen Member-Variablen:

- `list<Dart<dimension>> dart_list;`
- `vector<dimension> srcShape;`

Die `dart_list` repräsentiert den globalen Container der Darts, die in der GMAP enthalten sind. Die `srcShape` stellt hingegen die Größe der Datenstruktur dar, aus der die GMAP erzeugt wurde. Die folgende Tabelle beschreibt die Schnittstellen der Basisklasse:

Tabelle 6.1.3: Schnittstellen der Klasse GMAPBASE

`unmarkMap()`
führt `unmark()` auf allen enthaltenen Darts aus.

`unmarkMap(int)`
löscht alle Markierungen vom Typ des Übergabeparameters auf allen Darts.

`UnmarkMapActive()`
löscht alle Markierungen aller Darts, die zur Visualisierung genutzt werden. Betroffen hiervon sind die Markierungen vom Typ $0,\ldots,6$.

`unmarkMapAll()`
löscht alle Markierungen, egal welchen Typs, aller Darts.

`list<Dart<dimension>> & getDartList()`
liefert eine Referenz auf die enthaltene Liste von Darts zurück.

`vector getShape()`
gibt die Ausmaße der Datenstruktur (Bild, Volumen etc.) zurück, aus der die GMAP erzeugt wurde.

`setShape(vector)`
Setzt die Ausmaße der Datenstruktur (Bild, Volumen etc.), aus der die GMAP erzeugt wurde.

Die zweite implementierte Template-Klasse – die GMAP – erbt, bis auf den Fall von $dimension = 3$, von der Basisklasse. Für $dimension = 3$ gibt es den in Punkt 3. weiter oben erwähnten Sonderfall einer zusätzlichen Funktion. Somit ergeben sich für alle GMAPs mit $dimension \neq 3$ die oben genannten Schnittstellen, für den Fall $dimension = 3$ ergibt sich zusätzlich noch folgende Spezialisierung:

Tabelle 6.1.4: Teilweise Spezialisierung der GMap<3>

`vector<3> getVoxelFromDartID(Dart<3>)`
Gegeben ein Dart, berechnet diese Funktion aus der DartID den ursprünglichen Voxel, aus dem dieser erzeugt wurde, und liefert ihn in Form eines dreidimensionalen Vektors zurück.

Des Weiteren werden folgende Abkürzungen eingeführt, die den Umgang mit der GMAP in unterschiedlichen Dimensionen erleichtern sollen:

- GMAP1D für GMAP<1>
- GMAP2D für GMAP<2>
- GMAP3D für GMAP<3>

Es sei an dieser Stelle darauf hingewiesen, dass eine GMAP nicht aus sich selbst heraus erstellt werden kann, sondern nur mit Hilfe einer Hilfsklasse, wie zum Beispiel dem GMAP3DCREATOR (siehe Kapitel 6.2). Der Erstellungsvorgang ist sehr komplex und wird daher im folgenden Unterkapitel beschrieben.

6.2 Von einer ikonischen zu einer topologischen Regionenrepräsentation

In diesem Unterkapitel wird der Erstellungsvorgang einer GMAP3D beschrieben. Als Eingabe für einen solchen Algorithmus dient dabei ein Volumen, welches schon vorher bearbeitet worden sein muss. Zu bearbeitende Volumen müssen einer ikonischen Regionenrepräsentation entsprechen, in der alle Voxel einer Region jeweils über 6-verbundene Pfade zu erreichen sind. Für diese Art der Vorverarbeitung eignet sich zum Beispiel der Algorithmus aus Kapitel 5, die Union-Find-Wasserscheiden-Transformation. Allerdings kann jeder andere Algorithmus verwendet werden, der Ergebnisse der gleichen Art liefert.

Zur Erstellung einer GMAP3D wird in diesem Unterkapitel ein neuer Ansatz beschrieben werden, der diesen Vorgang vereinfachen, beschleunigen und die Speicherkomplexität reduzieren soll. Aus diesem Grund wurde die Volume-Map eingeführt, die im folgenden Absatz beschrieben wird. Aus dieser Struktur kann dann sofort eine Level-3-3-G-Map erstellt werden. Dieser Ansatz stellt eine neue Möglichkeit der Erstellung dar, die in dieser Art noch nicht in der Literatur zu finden ist.

Da die Erstellung im Laufe der Arbeit sukzessive verbessert wurde, folgt am Ende dieses Unterkapitels ein Abschnitt, der diese Optimierungen näher beleuchtet. In diesem soll zusätzlich die Herangehensweise an die Lösung von Problemen verdeutlicht werden. Zunächst folgt aber die Beschreibung der finalen Vorgehensweise.

6.2.1 Motivation einer Volume-Map

Die Volume-Map stellt einen Zwischenschritt bei der Erstellung einer GMAP3D dar. Ihre Aufgabe ist es, die große Komplexität bei der Erstellung der GMAP3D zu minimieren. Wie in Abschnitt 3.4.3.2 bereits beschrieben wurde, ist eine übliche Art der Erstellung einer 3-G-Map, mit einer Level-0-Karte zu beginnen und diese dann sukzessive zu vereinfachen, bis man sie nicht weiter verändern kann, ohne dass die Topologie verändert wird. Da in der Level-0-Karte für jeden Voxel des Ausgangsbildes 24 Darts erzeugt werden, hat man schon bei recht kleinen Volumen einen hohen Speicheraufwand. Die Volume-Map stellt einen Ansatz dar, dieses Problem zu umgehen. Es hat sich herausgestellt, dass die Anzahl der benötigten Darts zur Darstellung einer Level-*i*-Karte bei jedem Schritt stark abnimmt. So werden für eine Level-0-Karte noch ein Vielfaches an Darts gegenüber der Level-1-Karte benötigt. Und selbst beim Übergang von einer Level-2- zu einer Level-3-Karte sinkt der Bedarf an Darts auf circa die Hälfte[58].

Es lässt sich also eine bedeutende Leistungssteigerung vermuten, wenn nicht erst eine 3-G-Map des Levels null mit all ihren Darts erzeugt werden muss. Die Volume-Map stellt eine Möglichkeit dar, diese Karte Speicher sparend darzustellen, und dient zusätzlich als Ausgangspunkt für die Erstellung einer Level-3-3-G-Map. Um dieses zu ermöglichen, stellt die Volume-Map eine Möglichkeit dar, die benötigten Darts durch Platzhalter zu repräsentieren. Die Volume-Map kann dann entsprechend der Precodes (siehe Abschnitt

58 Das Verhältnis hängt stark vom Eingangsbild ab. Bei natürlichen Bildern hat sich aber ein entsprechendes Verhältnis als relativ exakt erwiesen.

6.2.2) bis zu einer Level-2 vergleichbaren Karte vereinfacht werden. Einer direkten Erstellung einer Level-3-3-G-Map, mit den in ihrer Anzahl stark reduzierten Darts, steht danach nichts mehr im Weg.

Definition 6.2.1 (Volume-Map) Eine Volume-Map g sei eine Funktion $g: \mathbb{N}^3 \times \{0,\ldots,23\} \to \{0,1\}$ mit folgenden Eigenschaften:

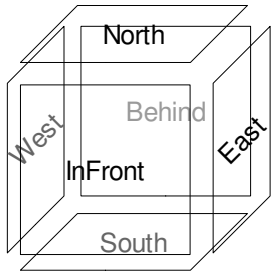
$$g(x,y,z,b) = \begin{cases} 0 & \text{falls am Voxel } (x,y,z) \text{ kein Dart-Platzhalter an Position } b \text{ ist,} \\ 1 & \text{sonst.} \end{cases}$$

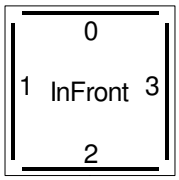
Im Programm lässt sich eine Volume-Map durch ein MULTIARRAY, einem Template-Datentyp der VIGRA[59], realisieren. Für die Volume-Map wird ein MULTIARRAY der Dimension drei und einem Template-Parameter des Typs `unsigned long` verwendet. Die Dimensionen der Volume-Map müssen dabei denen des Ausgangsbildes entsprechen. Ein `unsigned int` stellt mit seinen 32 Bit genügend Platz zur Kodierung der 24 Platzhalter für die Darts bereit.

Eine Volume-Map ist also nicht anderes als ein Volumen, in dem jeder Voxel einen Wert zwischen 0_b und $1111.1111.1111.1111.1111.1111_b$ besitzt. Dieser Wert jedes Voxels zeigt an, welche Dartplatzhalter gesetzt sind und welche nicht, und ist somit das entscheidende Kriterium bei der Umwandlung von der Volume-Map zur GMAP3D.

In Abbildung 58 ist ein Voxel der Volume-Map dargestellt. Dieser besteht aus sechs Flächen, die nach der Richtung benannt sind, in die sie zeigen (es wurden die Richtungen der Volumennachbarschaft übernommen, siehe Abschnitt 4.2.1). Die Flächen bestehen wiederum aus vier Kanten, welche die Platzhalter für die Darts sind, die nach ihrer Position bezüglich des Mittelpunktes benannt sind (vgl. mit den Richtungen aus Abschnitt 3.2.2).

59 siehe [Vig06e]





Fläche	InFront (I)				North (N)				West (W)				Behind (B)				South (S)				East (E)			
Dart-Platzhalter	N	W	S	E	I	E	B	W	I	N	B	S	N	E	S	W	I	W	B	E	I	S	B	N
Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Abbildung 58: Repräsentation der Volume-Map. Oben links: Ein Voxel der Volume-Map mit seinen sechs Flächen. Oben rechts: Aufbau einer Voxel-Fläche am Beispiel der InFront-Fläche. Unten: Repräsentation der weiteren Flächen und deren Dartplatzhalter.

6.2.2 Erzeugung einer Volume-Map

Wie bereits oben beschrieben, soll die Volume-Map die Erstellung einer GMAP3D erleichtern und beschleunigen. Dies geschieht dadurch, dass die Volume-Map die Aufgabe der Repräsentation einer Level-2-Karte übernimmt. Um eine Volume-Map dieser Art zu erzeugen, wird ein Volumen in der ikonischen Regionenrepräsentation als Eingabe benötigt. Damit eine explizite Randbehandlung dieses Volumens überflüssig wird, ist es notwendig, dass dieses Volumen in ein größeres Volumen kopiert wird, sodass am Rand des ursprünglichen Volumens jeweils ein Voxel eines äußeren Universums entsteht.

Die Funktion, welche für die Erstellung der Volume-Map zuständig ist, heißt `createVolumeMap_Level2` und findet sich in der Klasse GMAP3DCREATOR. Eine Beschreibung der Funktionalität ist durch Algorithmus 6.2.1 gegeben.

Algorithmus 6.2.1: Berechnung einer Volume-Map

```
function createVolumeMap_Level2

input : source_volume
output: dest_volume

for all voxels of source_volume in lexical order
   initialize current voxel of dest_volume with value of a level-0 map voxel

   for all causal neighbors of current_voxel in source_volume
      if value of current_voxel = value of neighbor
         delete bits of neighbor_face from current_voxel in dest_volume
         delete bits of neighbor_face from neighbor in dest_volume

         for all other causal directions
            if the values of the two voxels in direction to current_voxel and its
               neighbor are equivalent

               delete adequate edge bits of this voxels in dest_volume
```

Anschaulich betrachtet, werden in diesem Algorithmus die Precodes umgesetzt, die zu einer Level-2-Karte führen (siehe Anhang B). Allerdings werden die Precodes nicht in Form von Look-Up-Tabellen repräsentiert, sondern durch den expliziten Vergleich des aktuellen Voxels mit den entsprechenden Nachbarn. Dafür werden nur einmal fast alle Voxel des Eingabevolumens betrachtet. Da man von jedem Voxel seine kausalen Nachbarn anschaut, macht es keinen Sinn die Voxel zu betrachten, die kausal gesehen am Rand des Volumens liegen.

Wie in den Precodes zu sehen war, macht ein Entfernen von Darts nur Sinn, wenn der Nachbarvoxel das gleiche Label besitzt. In diesem Fall werden in der Volume-Map die entsprechenden Dartplatzhalter gelöscht. Dieser Vorgang wird durch ein Nullen der Bits der entsprechenden Dartplatzhalter im Ausgabevolumen umgesetzt.

Haben zwei benachbarte Voxel die gleiche Regionsnummer, so werden im Ausgabevolumen die Dartplatzhalter der zugewandten Flächen entfernt. Ein Beispiel für den Übergang von einer Level-0- zu einer Level-1-Konfiguration gibt die Abbildung 59, bei der die Dartplatzhalter gelöscht werden (hierzu sei auch die Abbildung 58 zur Übersicht über die Dartplatzhalter empfohlen).

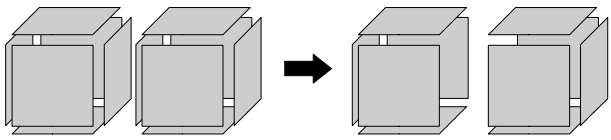


Abbildung 59: Entfernen von Dartplatzhaltern in einer Volume-Map (Level-0 zu Level-1)

Eine weiteres Löschen von Dartplatzhaltern beim Übergang von einer Level-1- zu einer Level-2-Konfiguration ist immer dann möglich, wenn zwei benachbarte Voxel das gleiche Label haben und zwei andere benachbarte Voxel, die zu den beiden ersten benachbart sind, auch ein gleiches Label haben. Liegt eine solche Konfiguration vor, so ist es möglich, jeweils zwei Flächen zu jeweils einer zusammenzufassen, indem die Dart-Platzhalter gelöscht werden, die trennende Kanten darstellen. In Abbildung 60 ist dieser Vorgang an einem Beispiel zu sehen.

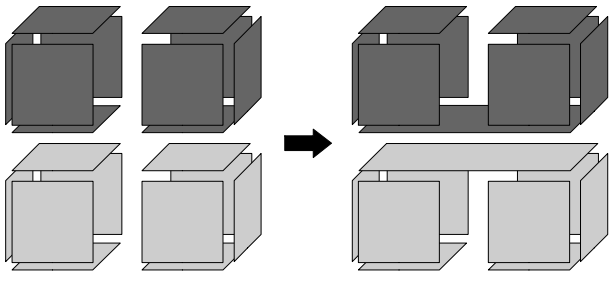


Abbildung 60: Entfernen von Dartplatzhaltern in einer Volume-Map (Level-1 zu Level-2)

Wurden für alle Voxel des Volumens diese Operationen ausgeführt, so haben die Voxel im Ausgabevolumen genau die Werte, die sie eine Level-2-Karte repräsentieren lassen.

Zur Überführung in eine GMAP3D müssen nun für die Dartplatzhalter richtige Darts erstellt werden. Außerdem müssen aus der Volume-Map die Orbits abgeleitet werden. Der Vorgang der Erstellung der GMAP3D wird im nächsten Abschnitt beschrieben.

6.2.3 Erstellung der GMAP3D aus der Volume-Map

Nachdem eine Volume-Map erstellt wurde, kann diese durch recht wenig Aufwand in eine GMAP3D umgewandelt werden. Der Aufwand ist relativ gering, da nur die Darts erstellt werden müssen, die auch für eine Level-3-Karte notwendig sind. Da die Darts (siehe Abschnitt 6.1.1) Träger aller Informationen und damit verhältnismäßig speicherintensiv sind, sinkt die Komplexität, so dass auch größere Volumen in Form einer GMAP3D repräsentiert werden können.

Der Vorgang der Erstellung einer GMAP3D geschieht in der Funktion `volumeMap_Level2ToGMap_Level3` der Klasse GMAP3DCREATOR. Diese Funktion benötigt als Übergabeparameter eine Referenz auf eine Liste vom Typ DART3D, sowie Zugriff auf das Volumen, welches die Volume-Map repräsentiert.

Algorithmus 6.2.2: Erzeugung einer GMAP

```
function volumeMap_Level2ToGMap_Level3

   input : volumeMap
   output: list<Dart3d>

   for all voxels of source_volume in lexical order
      for all set bits of current_voxel
         if dartDirectionsAtVoxelNode > 2
            create positive and negative dart
            add created darts to list
            calculate dartIDs of beta-orbit darts and link darts if they already
            exist


function dartDirectionsAtVoxelNode

   input : voxel of volumeMap
   output: count of directions

   for all voxel incident to the InFront-North-West corner of input-voxel
      for all dart placeholder ending in this corner
         save direction of placeholder

   return sum of different directions
```

Die Umwandlung geschieht in einem Durchlauf durch das Volumen, welches die Volume-Map repräsentiert. Dabei werden für die gesetzten Bits (siehe Abbildung 58) jedes Voxels maximal zwei Darts erstellt. Es werden nicht alle Darts erstellt, da aus der Volume-Map, die eine Level-2-Karte repräsentiert, direkt eine Level-3-Karte erzeugt wird (vgl. Abschnitt 3.4.2). Daher wird für jeden möglichen Dart geprüft, welche Darts zu dem gleichen Knoten inzident sind und welche Richtungen diese besitzen. Nur wenn Darts mit mindestens zwei unterschiedlichen Richtungen zu einem Knoten inzident liegen, wird ein Dart auch wirklich erzeugt. Die drei unterschiedlichen Richtungen sind in diesem Zusammenhang: *InFront*↔*Behind*, *West*↔*East* und *North*↔*South*. Zur

Ausführung dieses Tests wird die Funktion `dartDirectionsAtVoxelNode` verwendet, sie gibt die Anzahl der Richtungen zurück.

Die Abbildung 61 verdeutlicht Algorithmus 6.2.2 anhand eines Beispiels. Am Knoten V_1 sind drei verschiedene Richtungen zu zählen, hingegen kommt an Knoten V_2 nur eine Richtung vor. Dies führt dazu, dass Darts, die inzident zu V_2 wären, nicht erstellt werden. Zur besseren Übersicht werden nur jeweils vier Voxel an einem Knoten angezeigt. Die fehlenden Voxel sind entsprechend so vorzustellen, dass sich an der Anzahl der Richtungen nichts verändert.

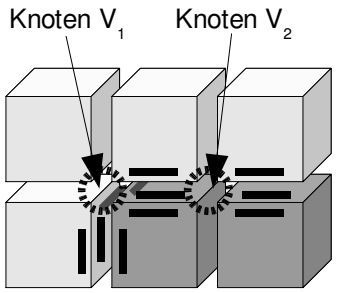


Abbildung 61: Knoten V_1 hat Dartplatzhalter in alle drei Richtungen, Knoten V_2 nur von West↔East.

Ergibt die Funktion `dartDirectionsAtVoxelNode` einen Wert größer als eins für einen aus dem Dartplatzhalter zu erstellenden Dart, so wird für diesen zu erstellenden Dart die DartID berechnet.

Die DartID ergibt sich zunächst aus der Position des Voxels im Volumen, also den Koordinaten (x, y, z), auf dem der Dartplatzhalter liegt. Ein weiterer wichtiger Wert ist die Position des Platzhalters auf diesem Voxel. Denn wie in Abbildung 58 beschrieben, gibt es auf einem Voxel 24 verschiedene Positionen für einen Platzhalter, von denen dieser die *i*-te einnimmt. Die eindeutige DartID lässt sich somit berechnen durch:

$$\pm\left[(x+(w\cdot y)+(w\cdot h\cdot z))\cdot 24+(i+1)\right].$$

In dieser Formel werden die Maße des Eingangsvolumens benötigt. Dabei steht *w* für die Breite und *h* für die Höhe des Volumens. Das $\pm$ wird benötigt, da jeder Voxel maximal durch 48 Darts repräsentiert wird, in der Volume-Map aber nur 24 Positionen zur Verfügung stehen. Das kommt daher, dass jede Kante auf einer Voxelfläche in der Volume-Map durch zwei Darts repräsentiert wird, die durch das Orbit β_0 vernäht werden. Einer dieser Darts bekommt ein positives, der andere ein negatives Vorzeichen. Auf diese Weise werden alle Darts mit einer eindeutigen Kennung erstellt.

Ein Objekt des Typs DART3D kann nun erstellt und an die Liste mit den Darts angehängt werden. Des Weiteren wird in einer Hash-Tabelle ein Eintrag für diesen Dart erzeugt. Ein solcher Eintrag besteht aus der DartID, welche den Schlüssel darstellt, und der Speicheradresse des Darts. Diese Art der Speicherung beschleunigt die spätere Vernähung der Darts. Im Anschluss an die Erstellung wird die geometrische Einbettung für diesen Dart gespeichert. Zu diesem Zweck muss ein Vektor berechnet werden, der den

Startpunkt des Darts im Raum darstellt. In Abbildung 62 kann man erkennen, dass sich die geometrische Einbettung (der Knoten) aus einem Voxel der Volume-Map, der Position des Dartplatzhalters und dem Vorzeichen des Darts ergibt.

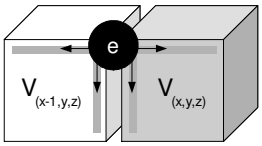


Abbildung 62: Berechnung der geometrischen Einbettung der Darts. Die vier abgebildeten Darts erhalten die selbe Einbettung $e=(x, y, z)$.

Definition 6.2.2 (Berechnung der geometrischen Einbettung eines Darts) Die geometrische Einbettung $\vec{e}$ eines Darts d ergibt sich aus der Position $\vec{p}$ des Voxels in der Volume-Map, dem Vorzeichen des Darts und dem Dartplatzhalter i.

Sei $\vec{p}=(x, y, z)$ und $i'=i$ für einen positiven Dart beziehungsweise $i'=-i$ für einen negativen Dart. Dann gilt:

Falls $i' \in \{-9,-8,-1,2,5,10\}$, dann liegt d an dem Knoten $\left(x-\frac{1}{2}, y-\frac{1}{2}, z-\frac{1}{2}\right)$.

Falls $i' \in \{-16,-10,-7,8,11,13\}$, dann liegt d an dem Knoten $\left(x-\frac{1}{2}, y-\frac{1}{2}, z+\frac{1}{2}\right)$.

Falls $i' \in \{-17,-12,-2,3,9,18\}$, dann liegt d an dem Knoten $\left(x-\frac{1}{2}, y+\frac{1}{2}, z-\frac{1}{2}\right)$.

Falls $i' \in \{-18,-15,-11,12,16,19\}$, dann liegt d an dem Knoten $\left(x-\frac{1}{2}, y+\frac{1}{2}, z+\frac{1}{2}\right)$.

Falls $i' \in \{-24,-5,-4,1,6,21\}$, dann liegt d an dem Knoten $\left(x+\frac{1}{2}, y-\frac{1}{2}, z-\frac{1}{2}\right)$.

Falls $i' \in \{-23,-13,-6,7,14,24\}$, dann liegt d an dem Knoten $\left(x+\frac{1}{2}, y-\frac{1}{2}, z+\frac{1}{2}\right)$.

Falls $i' \in \{-21,-20,-3,4,17,22\}$, dann liegt d an dem Knoten $\left(x+\frac{1}{2}, y+\frac{1}{2}, z-\frac{1}{2}\right)$.

Falls $i' \in \{-22,-19,-14,15,20,23\}$, dann liegt d an dem Knoten $\left(x+\frac{1}{2}, y+\frac{1}{2}, z+\frac{1}{2}\right)$.

Die geometrische Einbettung des Darts d ist dann $\vec{e}=\left(\lceil x_n \rceil, \lceil y_n \rceil, \lceil z_n \rceil\right)$ [60], wobei (x_n, y_n, z_n) die Koordinaten des Knotens sind, an dem d liegt.

Um die folgenden Schritte zu begründen, wird eine Ordnungsrelation eingeführt:

Definition 6.2.3 (Ordnungsrelation $\prec_d$ über Darts) Die Ordnungsrelation $\prec_d$ über zwei Darts d_1 und d_2 mit den DartIDs id_1 beziehungsweise id_2 aus der Menge *B* aller Darts ist definiert durch:

$$d_1 \prec_d d_2 \Leftrightarrow |id_1|<|id_2| \vee (|id_1|=|id_2| \wedge id_1<0 \wedge id_2>0).$$

60 Die Funktion $\lceil\ \rceil$ berechnet den nächst größeren ganzzahligen Wert einer Zahl.

Ein Dart d_1 heißt *kleiner* bezüglich eines zweiten Darts d_2, wenn $d_1 \prec_d d_2$ gilt.

Durch den Erstellungsvorgang, also das sequentielle Durchlaufen der Volume-Map und der Dartplatzhalter, ist gewährleistet, dass der jeweils zuletzt erstellte Dart der größte von allen bereits erstellten Darts ist. Anschließend werden nur noch größere Darts erstellt. Diese Eigenschaft ist sehr wichtig bei der Vernähung von Darts, da im nächsten Schritt zwei Darts nur dann vernäht werden, wenn der aktuelle Dart der größere ist.

Bevor ein Dart mit anderen vernäht werden kann, muss herausgefunden werden, welche dies sind. Dafür gibt es die Hilfsfunktionen `getBeta0Orbit`, `getBeta1Orbit`, `getBeta2Orbit` und `getBeta3Orbit`. Gegeben die aktuelle DartID, berechnet jede Funktion die DartID des zu vernähenden Darts des entsprechenden Orbits. Dabei ist diese Suche für die Orbits β_0 und β_3 recht einfach, da die Suche stets eindeutig ist. Das heißt, es gibt nur einen Dart, der als Kandidat für eine Vernähung in Frage kommt. In speziellen Ausnahmefällen kann es allerdings dazu kommen, dass für die β_1- und β_2-Orbits mehrere Kandidaten zur Verfügung stehen. Daher wird nun die Such- und Auswahlreihenfolge der zu vernähenden Darts näher ausgeführt.

Dabei gilt für das Finden des β_1 zu vernähenden Darts folgende Suchreihenfolge:

1. Überprüfe den Dartplatzhalter auf dem selben Voxel der Volume-Map am gleichen Knoten.
2. Überprüfe den Dartplatzhalter auf benachbarten Voxeln am gleichen Knoten, und zwar im Uhrzeigersinn, falls der aktuelle Dart ein negativer Dart ist, sonst gegen den Uhrzeigersinn.

Diese Suchreihenfolge ist in der folgenden Abbildung aufgezeigt.

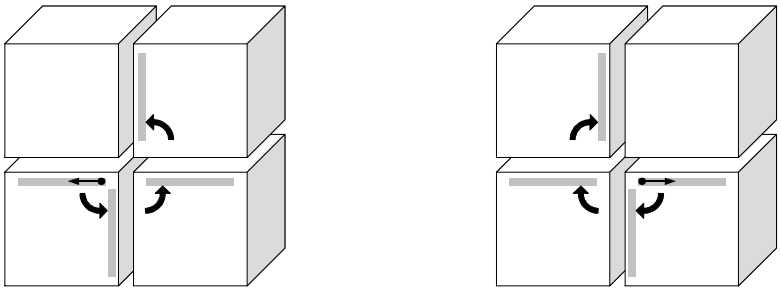


Abbildung 63: Suchreihenfolge der Dartplatzhalter der zu vernähenden Darts im β_1-Orbit. Links: Gegen den Uhrzeigersinn für positive Darts. Rechts: Im Uhrzeigersinn für negative Darts.

Für das Finden des β_2 zu vernähenden Darts gilt folgende Suchreihenfolge:

1. Überprüfe den Dartplatzhalter auf dem selben Voxel der Volume-Map an der gleichen Kante.

2. Überprüfe die Dartplatzhalter auf benachbarten Voxeln der Volume-Map an der gleichen Kante, und zwar im Uhrzeigersinn, falls der aktuelle Dart ein positiver Dart ist, sonst gegen den Uhrzeigersinn.

Diese Suchreihenfolge veranschaulicht die folgende Abbildung 64.

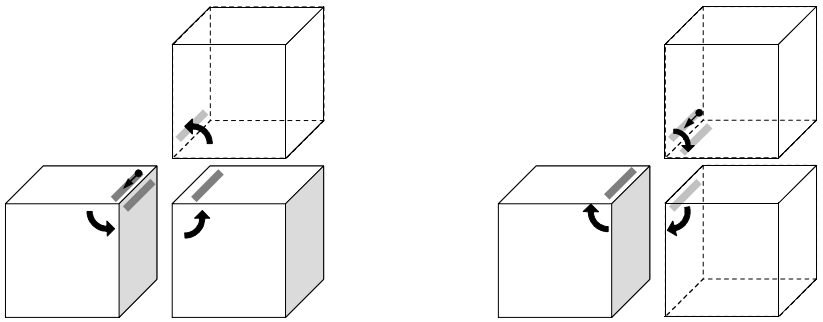


Abbildung 64: Suchreihenfolge der Dartplatzhalter der zu vernähenden Darts im β_2-Orbit. Links: Gegen den Uhrzeigersinn für negative Darts. Rechts: Im Uhrzeigersinn für positive Darts.

Die Funktionen zum Finden der Orbits liefern eine DartID zurück. Über diese kann man die Speicheradresse des entsprechenden Darts in der oben genannten Hash-Tabelle finden. Diese Speicheradressen werden dem aktuellen Dart mit der Funktion `setBetaOrbit` zugewiesen. Ebenfalls wird den gefundenen Darts die Speicheradresse des aktuellen Darts für das entsprechende Orbit zugewiesen.

Nach Ablauf aller Voxel der Volume-Map und der darauf befindlichen Dartplatzhalter ist der Vorgang der Umwandlung in eine Level-3-3-G-Map abgeschlossen. Es wurden alle für diese 3-G-Map verwendeten Darts erstellt, und es wurden außerdem alle nötigen Member-Variablen der Darts mit Informationen gefüllt.

6.2.4 Optimierungen bei der Erstellung einer GMAP3D

Die oben beschriebene Erstellung einer Level-3-3-G-Map aus einer Volume-Map ist das Ergebnis eines Entwicklungsprozesses, in dem eine kontinuierliche Verbesserung und Optimierung stattgefunden hat.

Anfangs bestand das Bedürfnis, mit möglichst einfachen Mitteln eine 3-G-Map zu erstellen. Diese war ebenfalls aus den gleichen Datentypen aufgebaut und nur die Erzeugung der Darts und den damit verbundenen Orbits unterschied sich von der endgültigen Vorgehensweise.

Es wurde damit begonnen die Volume-Map, welche eine Level-2-Karte repräsentiert, in eine Level-2-3-G-Map umzuwandeln. Dafür mussten für alle Dartplatzhalter in der Volume-Map je zwei Darts für die GMAP3D erstellt werden. Da das Ziel aber auch zu diesem Zeitpunkt eine Level-3-3-G-Map war, mussten im Anschluss daran noch Darts aus der GMAP3D entfernt werden, wobei natürlich auf eine Konsistenzerhaltung zu achten war. Wie bereits oben beschrieben, führte die Erstellung einer Level-2-3-G-Map zu einem

sehr viel größeren Speicherbedarf und Rechenaufwand und erwies sich dadurch bei größeren Volumendatensätzen als ineffizient. Daher wurde im ersten Optimierungsschritt die Level-3-3-G-Map direkt aus der Volume-Map berechnet. Zu diesem Zweck war es notwendig herauszufinden, wann Darts aus den Dartplatzhaltern der Volume-Map erzeugt werden müssen und wann nicht. Des Weiteren konnte die anfangs recht einfach umgesetzte Suche nach dem β_0-Orbit – β_0-vernäht wurden jeweils die Darts mit gleicher absoluter DartID – nicht weiter verwendet werden.

Ein weiteres Problem stellte aber grundsätzlich die Vernähung der Darts dar. Zu diesem Zeitpunkt wurden zunächst alle Darts erstellt, ohne dass sie Informationen über die vernähten Darts erhielten. Erst in einem zweiten Schritt wurden diese Informationen hinzugefügt. In Algorithmus 6.2.3 ist dieser Ansatz beschrieben.

Algorithmus 6.2.3: Der erste Weg zu einer GMAP

```
function volumeMap_Level2ToGMap_Level3

   input : volumeMap
   output: list<Dart3d>

   for all voxels of source_volume in lexical order
      for all set bits of current_voxel
         if dartDirectionsAtVoxelNode > 2
            create positive and negative dart
            add created darts to list
            calculate dartIDs of beta-orbit darts and store them

   for all darts in list
      find darts to stored beta-orbit dartIDs and sew them
```

Wie zu sehen ist, verlief das Verfahren in zwei Schritten. Im ersten Schritt wurden alle Darts der GMAP3D erstellt. Eine Referenzierung der Darts mit anderer Darts (Setzen der Orbit-Relationen) war zu diesem Zeitpunkt schwer möglich, da noch nicht die Speicheradressen aller Darts bekannt waren. Sie standen erst nach Fertigstellung der gesamten Liste von Darts zur Verfügung. Aus diesem Grund wurde im Anschluss ein zweiter Schritt durchlaufen, in dem die Referenzierung (auch Vernähung genannt) nachgeholt wurde, so dass jeder Dart die Speicheradressen der zu ihm über die Orbits in Relation stehenden anderen Darts erhielt (siehe Abschnitt 6.2.3).

Auch bei dieser Vorgehensweise waren allerdings Optimierungsmöglichkeiten gegeben, die letztlich auch in die endgültige Form des Algorithmus' einflossen. Die im obigen Abschnitt 6.2.3 beschriebene Hash-Tabelle bestand noch nicht, so dass kein direkter Zugriff auf die Speicheradressen der Darts anhand der DartID möglich war. Zunächst wurde zu jedem Dart bei seiner Erstellung ein Array angelegt, in dem nacheinander die DartIDs der zu vernähenden Darts gespeichert wurden. Diese Arrays wurden an einer dafür angelegten Liste angefügt. Die Position in der Liste entsprach dabei der Position des Darts auf der Liste der Darts. Nachdem alle Darts erstellt wurden, wurde im zweiten Schritt die Liste aller Darts einmal durchlaufen. Für jeden Dart wurden dann die zu vernähenden Darts in der Liste gesucht, indem die DartIDs aus dem Array nachgeschlagen wurden. Im schlechtesten Fall musste jeweils die gesamte Liste von Darts abgesucht werden.

Bei diesem Ansatz ist leicht zu erkennen, dass viele Suchvorgänge überflüssig sind, da es ausreicht, einmal die beiden Darts zu finden, die über ein Orbit vernäht sind, und für beide Darts `setBetaOrbit` auszuführen. Dadurch konnte die Hälfte der Suchvorgänge eingespart werden.

Um bei langen Listen die einzelnen Suchvorgänge weiter zu optimieren, mussten die Darts innerhalb der Liste der Größe nach geordnet vorliegen. Aus diesem Grund wurde die Ordnungsrelation aus Definition 6.2.3 eingeführt. Sie ermöglichte es, die Suche nach einem zu vernähenden Dart, beginnend beim aktuellen Dart, auf eine Richtung zu beschränken. Durch diese Maßnahme konnte der Aufwand jeder einzelnen Suche reduziert werden.

Doch auch die sequentielle Suche in nur eine Richtung erwies sich als verhältnismäßig langsam, wenn die Differenz der absoluten DartIDs hoch war. Ein schneller Zugriff auf die Speicheradressen der Darts über eine Hash-Tabelle löste dieses Problem. Mit ihrer Hilfe ist ein expliziter Zugriff auf die Speicheradresse eines Darts anhand seiner DartID sehr effizient möglich.

Nach Umsetzung der bisherigen Optimierungen erwies sich der Schritt der nachträglichen Zuweisung von β-Orbits als überflüssig. Da nur noch von einem Dart aus nach kleineren Darts gesucht wurde, konnte diese Suche auch direkt bei der Erstellung der einzelnen Darts erfolgen. Nachdem auch diese Änderung umgesetzt war, war die Vorgehensweise aus Abschnitt 6.2.3 erreicht.

Zusammengefasst wurden folgende Optimierungen bei der Erstellung einer GMAP vorgenommen:

1. Direktes Erstellen der Level-3-3-G-Map aus der Volume-Map,
2. β-Orbits für beide beteiligten Darts gleichzeitig setzen,
3. die Liste von Darts anhand der Ordnungsrelation $\prec_d$ nur in eine Richtung durchsuchen,
4. Einführung einer Hash-Tabelle zum Suchen der Speicheradressen und
5. Setzen der Beta-Orbits während der Erstellung der Darts.

Alle diese Optimierungsschritte erlauben es, jetzt auch größere Volumendatensätze in relativ kurzer Zeit zu verarbeiten (siehe Abschnitt 9.2.1). Die Rechendauer zur Erstellung liegt je nach Größe und Beschaffenheit des zu verarbeitenden Volumens bei bis zu einem mehrfachen Tausendstel der ursprünglichen Dauer. Auch der Speicheraufwand wurde deutlich reduziert, wofür vor allem die Schritte eins und fünf verantwortlich sind.

6.3 Operationen auf der GMAP3D

Dieses Unterkapitel beschreibt einige denkbare Operationen, welche auf der GMAP3D durchgeführt werden können. Den Anfang bildet ein Traverser, welcher die Durchquerung einer GMAP3D erleichtert. Er ist im Rahmen der generischen Struktur der GMAP ebenfalls für alle Dimensionen kleiner als vier definiert, obwohl im Rahmen dieser Arbeit nur von der dreidimensionalen Spezialisierung Gebrauch gemacht wird.

Das Traversieren stellt die Grundlage für alle weiteren Schritte dar, so auch für den des Verschmelzens zweier Regionen an einer trennenden Fläche, wie in Abschnitt 6.3.2 beschrieben. Anhand dieser Operation des Verschmelzens wird ein Ausblick darauf gegeben, wie einfach es ist, grundlegende Operationen auf der implementierten Datenstruktur umzusetzen.

6.3.1 Traversieren über die GMAP3D

Um aus einer GMAP3D Eigenschaften abzulesen, müssen die verwendeten Algorithmen die GMAP3D traversieren. Dieses Durchqueren der GMAP3D geschieht durch ein „Springen“ von einem Dart der GMAP3D zu einem anderen Dart. Die Verbundenheit zweier Darts ist gegeben, wenn sie β_i -vernäht sind (mit $i \in \{0,1,2,3\}$).

Ein allgemeiner generischer Ansatz zur Kapselung eines solchen Durchlaufens eines abstrakten Datentypes besteht in der Definition eines Iterators über die gegebene Datenstruktur. Dies geschieht unabhängig von den Template-Parametern, mit denen der Datentyp initialisiert wurde.[61] Da im Kontext der GMAP3D allerdings nicht von einem Iterieren über die Struktur gesprochen werden kann, wird der implementierte Datentyp als Traverser bezeichnet. Ein DARTITERATOR hingegen lässt sich auf der verwendeten Datenstruktur der GMAP recht einfach als ein Iterator über die Liste von Darts, welche in der GMAP gespeichert wird, definieren. Allerdings gilt es hierbei zu bedenken, dass ein solcher Iterator nur in wenigen Ausnahmefällen (siehe Kapitel 6.1.2 und 8.1.2) sinnvoll ist. Deshalb wird im Folgenden auch nicht mehr auf die mögliche Implementation eines DARTITERATORS eingegangen.

Folgende Anforderungen wurden in dieser Arbeit an den ADT DARTTRAVERSER gestellt:

1. Er sollte über GMAPs der Dimension $d \in \{1,2,3\}$ traversieren können.

   Der Traverser sollte also in Abhängigkeit von der Dimension der GMAP über alle verfügbaren Orbits zwischen den Darts wechseln können.

2. Er sollte einen Zugriff auf den aktuellen Dart bieten.

   Der aktuelle Dart sollte dabei platzsparend als Zeiger repräsentiert werden. Über diesen können alle Eigenschaften des aktuellen Darts ausgelesen werden.

3. Die Gleichheit beziehungsweise Ungleichheit zweier DARTTRAVERSER sollte definiert sein.

61 Vgl. hierzu auch die Kapselung der Nachbarschaften in Kapitel 4. Dort wurde ein MultiIterator beschrieben, der nach dem gleichen Ansatz über dreidimensionale Bilddaten, gleich welchen Typs (Template-Parameter), iteriert.

Dabei sollten zwei DARTTRAVERSER immer dann gleich sein, wenn ihre aktuellen Darts gleich sind.

4. Einige wichtige Traversierungen sollten direkt verfügbar sein.

   Damit erhält man den Vorteil, nicht jeden häufig auftretenden Traversierungsalgorithmus mehrfach zu schreiben. Allerdings müssen diese auch mit der verwendeten Dimension konsistent sein. So kann beispielsweise ein eindimensionaler DARTTRAVERSER keine Flächenkonturen wechseln, wie es für eine Traversierung von Oberflächen notwendig wäre.

Da aus den obigen Punkten schon ersichtlich ist, dass die Funktionalität sehr stark von der übergebenen Dimension abhängt, wurde die Implementation, wie auch schon die der GMAP, in zwei Template-Klassen gegliedert.

Die erste Klasse DARTTRAVERSERBASE stellt dabei die grundlegenden Funktionen eines DARTTRAVERSERS bereit, welche sich in Abhängigkeit des Template-Parameters immer gleich verhalten. Erstellt werden kann ein DARTTRAVERSER mit folgenden Möglichkeiten:

- Ohne die Übergabe eines Parameters

  Dabei wird ein DARTTRAVERSER erstellt, der nicht auf einen Dart sondern auf NULL zeigt.

- Durch die Übergabe eines Dart-Zeigers

  Ein DARTTAVERSER mit dem übergebenen Dart-Zeiger wird erstellt.

- Durch die Übergabe eines weiteren DARTTRAVERSERS gleicher Dimension

  Dies erstellt einen DARTTRAVERSER, der auf den gleichen Dart zeigt, wie der übergebene DARTTRAVERSER.

Die einzige Member-Variable des DARTTRAVERSERS ist ein Zeiger auf einen Dart der GMAP:

- `Dart<dimension>* currentDart`

wobei `dimension` der Template-Parameter des Traversers ist. Die Schnittstelle der Basisklasse wird in Tabelle 6.3.1 erläutert.

Tabelle 6.3.1: Schnittstellen der Klasse DARTTRAVERSERBASE

`Dart<dimension>* getCurrentDart()` liefert den aktuellen Dart-Zeiger, nicht jedoch den Dart, zurück.
`nextBeta(int i)` setzt den aktuellen Dart-Zeiger auf den Zeiger des β_i-Orbits des aktuellen Darts. Sollte es kein β_i-Orbit für die Dimension des Darts geben, so wird eine Ausnahme ausgelöst.
`traverseMap(int, int)` Sei *maxOrbit* der erste und *label* der zweite übergebene Parameter. Die Methode traversiert ab dem aktuellen Dart alle Darts, die mit diesem transitiv über die Orbits β_i mit $0 \le i \le maxOrbit$ vernäht sind, und markiert diese mit einer Markierung *label*. Diese Funktion wird in den spezialisierten Klassen (siehe

unten) benötigt.

`isReachableWithMaxOrbit(Dart<dimension>*, int, int)`
: Sei *dart* der erste, *maxOrbit* der zweite und *label* der dritte übergebene Parameter. Diese Funktion prüft, ob *dart* transitiv über die Orbits β_i mit $0 \leq i \leq maxOrbit$ mit dem aktuellen Dart des DartTraversers vernäht ist. Zum Traversieren werden alle besuchten Darts mit *label* markiert.

`vector traverseMapReturnCoG(int, int)`
: führt die oben beschriebene Methode `traverseMap` aus, hat jedoch als Rückgabewert einen Vektor, der den Schwerpunkt aller besuchten geometrischen Einbettungen der Darts zurückliefert.

Um den oben erwähnten Vergleich zweier DartTraverser zu implementieren, werden Operatoren benötigt. Damit auf Elemente, über die ein Traverser läuft, noch einfacher zugegriffen werden kann als über die Funktion `getCurrentDart()`, welche zudem nur einen Zeiger auf den aktuellen Dart zurückgibt, wird zusätzlich noch der Dereferenzierungs-Operator überladen. Die folgende Tabelle veranschaulicht die Operatoren des DartTraversers:

Tabelle 6.3.2: Operatoren der Klasse DartTraverserBase

`bool operator==(DartTraverserBase<dimension>)`
: liefert `true` zurück, wenn die aktuellen Darts der beiden DartTraverser identisch sind, sonst `false`.

`bool operator!=(DartTraverserBase<dimension>)`
: liefert `true` zurück, wenn die aktuellen Darts der beiden DartTraverser verschieden sind, sonst `false`.

`Dart<dimension> operator*()`
: gibt den aktuellen Dart zurück, auf den der Traverser zeigt (nicht den Zeiger auf diesen).

Diese Basisklasse enthält alles, um als Grundlage zu dienen, auf der der generische DartTraverser und seine Spezialisierungen für den ein-, zwei- und dreidimensionalen Fall aufbauen können. Da der generische unspezialisierte DartTraverser der Dimension *n* lediglich von der Klasse DartTraverserBase der Dimension *n* erbt, wird sie hier nicht weiter beschrieben. Alle enthaltenen Schnittstellen und Operatoren finden sich in vorigen zwei Tabellen.

Die teilweise Spezialisierung des DartTraversers bringt hingegen einige neue Schnittstellen mit sich. Sie sind in der folgenden Tabelle 6.3.3 beschrieben:

Tabelle 6.3.3: Teilweise Spezialisierung der Template-Klasse DARTTRAVERSER<*dimension*>

DartTraverser<1>	**DartTraverser<2>**	**DartTraverser<3>**
`nextBeta0()`	`nextBeta0()`	`nextBeta0()`
`nextBeta1()`	`nextBeta1()`	`nextBeta1()`
	`nextBeta2()`	`nextBeta2()`
		`nextBeta3()`
`traverseFaceContour(int)`	`traverseFaceContour(int)`	`traverseFaceContour(int)`
	`traverseSurface(int)`	`traverseSurface(int)`
		`traverseConnectedComponents(int)`
`vector traverseFaceContourReturnCoG(int)`	`vector traverseFaceContourReturnCoG(int)`	`vector traverseFaceContourReturnCoG(int)`
	`vector traverseSurfaceReturnCoG(int)`	`vector traverseSurfaceReturnCoG(int)`
		`vector traverseConnectedComponentsReturnCoG(int)`

Dabei stellen die Methoden `nextBeta0()`, `nextBeta1()`, `nextBeta2()` und `nextBeta3()` lediglich weitere Bezeichner für die Methode `nextBeta(int)` der Template-Basisklasse dar.

Die Methoden `traverseFaceContour(int)`, `traverseSurface(int)` und `traverseConnectedComponents(int)` stellen intuitivere Bezeichner für `traverseMap(1,int)`, `traverseMap(2,int)` und `traverseMap(3,int)` dar. Die Methoden, die auf `ReturnCoG` enden, leisten das gleiche für die Funktion `traverseMapReturnCoG(int,int)`.

Mit diesen zusätzlichen Methoden ist ein DARTTRAVERSER beschrieben, der alle an ihn gestellten Anforderungen erfüllt.

6.3.2 Verschmelzung von Regionen an einer Fläche

In diesem Abschnitt wird die Verschmelzung von zwei benachbarten Regionen in der GMAP3D beschrieben. Dabei geht es lediglich darum, eine klar definierte vorhandene Grenze in Form einer Fläche zwischen zwei Regionen zu entfernen. In der Praxis müssten selbstverständlich alle Flächen, welche beide Regionen voneinander trennen, entfernt werden, um eine konsistente und sinnvolle GMAP3D sicherzustellen. Des Weiteren wird nach dem Verschmelzen zweier Regionen keine Level-3-Karte mehr vorliegen, da das Verfahren dies nicht sicherstellt. Dennoch eignet sich dieses einfache Beispiel gut, um den im vorigen Abschnitt beschriebenen DART3DTRAVERSER anzuwenden und eine Operation auf der GMAP3D durchzuführen.

In der folgenden Abbildung 65 ist eine GMAP3D dargestellt, die lediglich aus zwei Regionen besteht, die an einer gemeinsamen Fläche liegen.

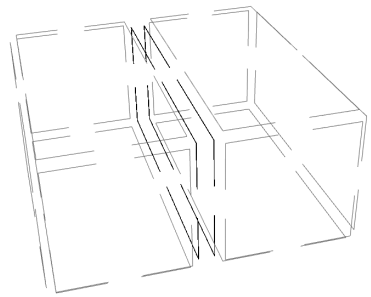


Abbildung 65: Eine GMAP3D, in der die Darts der zu löschenden Fläche dunkel hervorgehoben sind.

Der Verschmelzungsalgorithmus wird im Folgenden als Pseudo-Code formuliert, bevor die Vorgehensweise beschrieben und das Resultat der Anwendung auf die obige Abbildung 65 gezeigt wird.

Algorithmus 6.3.1: Verschmelzen von Regionen an einer Fläche

```
function mergeRegionsAtFace

   input : any dart of face that will be deleted

   while not traversed around the whole face
      set current dart of traverser beta3-beta2-sewed dart
         to current dart of traverser beta-2-sewed dart
            and vice versa
      mark current dart of traverser and its beta3-sewed dart
      traverse to next dart of face

   delete all marked darts
```

Dieser Algorithmus durchläuft alle Darts der zu löschenden Fläche und lässt sich in zwei Verarbeitungsschritte gliedern:

1. Umnähen der Darts

   Zum Umnähen wird folgende Regel benutzt:

   Sei b ein Dart, welcher zu der zu löschenden Fläche gehört. Dann müssen folgende Umnähungen vorgenommen werden:

   i. $(\beta_3 \circ \beta_2)(b)$ muss β_2-vernäht werden mit $(\beta_2)(b)$ und

   ii. $(\beta_2)(b)$ muss β_2-vernäht werden mit $(\beta_3 \circ \beta_2)(b)$.

2. Löschen aller Darts der Fläche

   Zum Löschen werden alle Darts, die an der Fläche anliegen und ihre β_3-vernähten Darts aus der GMAP3D entfernt.

Das Resultat dieser Verschmelzungsoperation zeigt die folgende Abbildung 66, welche die GMAP3D aus Abbildung 65 vor und nach der Verschmelzung zeigt. Zur

Verdeutlichung der umgenähten Darts sind zwei Darts hervorgehoben, anhand derer die Orbits aufgezeigt werden, die umgenäht wurden.

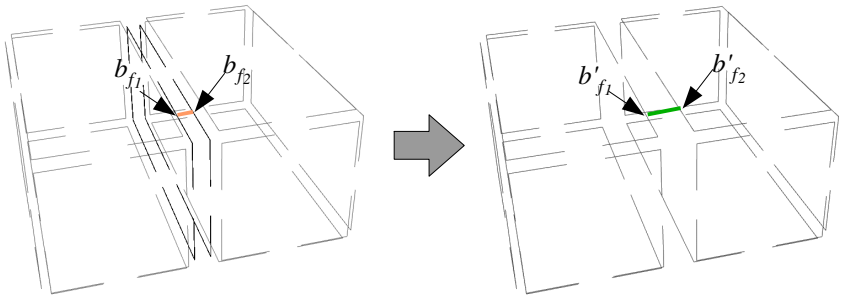


Abbildung 66: Die linke Teilabbildung zeigt die GMAP3D vor der Verschmelzungsoperation, die rechte den am Ende vorliegenden Zustand. Beispiel für die Umnähung: Die ehemals β_3-vernähten Darts b_{f_1} und b_{f_2} werden gelöscht und deren β_2-vernähte Darts b'_{f_1} und b'_{f_2} werden über das β_2-Orbit miteinander vernäht.

Die hier vorgestellte Verschmelzung zeigt lediglich eine sehr überschaubare Operation auf der GMAP3D auf. Durch Anwendung der hier beschriebenen Verschmelzung bleibt die Struktur einer Level-3-Karte nicht erhalten. Außerdem sollten Verschmelzungsoperationen direkt auf der XGMAP3D (siehe Kapitel 7.2) ausgeführt werden, um eine angemessene Behandlung von Einschlüssen zu gewährleisten. Aus diesen Gründen wurde auf die Umsetzung dieses einfachen Algorithmus' verzichtet.

7 Umsetzung der erweiterten 3-G-Map

In diesem Kapitel wird die Datenstruktur der dreidimensionalen XG-Map erläutert. Bei dieser handelt es sich um eine Lösung der in Abschnitt 3.4.1 beschriebenen Probleme, die auftreten, wenn Teile einer Oberfläche keine direkte Verbindung zueinander haben.

Um diese Probleme zu beseitigen, werden für die XGMAP3D verschiedene abstrakte Datentypen benötigt. Die Implementationen dieser werden im Laufe dieses Kapitels näher erläutert. Diese Datentypen erlauben es, einen erweiterten Zusammenhang von Darts der GMAP3D zu speichern und zugreifbar zu machen.

Nach der Beschreibung dieser Datentypen und ihrer Zusammenfassung in dem abstrakten Datentyp XGMAP3D wird ausführlich die Gewinnung der Informationen zur Erstellung der Datenstruktur erläutert.

Im Anschluss daran werden noch einige Möglichkeiten der Nutzung der XGMAP3D genannt, sowie ein Übergang zur grafischen Darstellung geschaffen, welche im weiteren Verlauf dieser Arbeit noch näher beschrieben wird. Für diese hat die XGMAP3D eine große Bedeutung.

7.1 Motivation für die 3-XG-Map

Im vorherigen Kapitel wurden der Datentyp der GMAP und seine Möglichkeiten der Anwendung vorgestellt. Der DARTTRAVERSER aus Abschnitt 6.3.1 stellte in diesem Zusammenhang eine Möglichkeit dar, die verschiedenen Darts, die in der GMAP3D gespeichert wurden, zu durchlaufen. Dafür wurden verschiedene Orbits benutzt, die ein Wechseln von Darts zu anderen Darts ermöglichen.

Allerdings gibt es das Problem, dass durch diesen Traverser nicht in allen Fällen auch alle Darts zu erreichen sind, wenn man bei einem beliebigen Dart startet. In Kapitel 3 in Abschnitt 3.4.1 wurden bereits Fälle aufgezeigt, bei denen Volumenbegrenzungen eines Objekts nicht aneinander lagen und somit keine Verbindung besaßen. Nach der Erstellung einer GMAP3D aus einem solchen Volumen gibt es daher auch Darts, die nicht über die Orbits zu erreichen sind. In der Abbildung 67 wird dieser Sachverhalt noch einmal dargestellt. Das rechte Bild der Abbildung stellt den Teil der GMAP3D dar, der den Bernstein repräsentiert. In diesem Bild werden die Darts der weiteren Schalen (Außenwelt und Insekt) ausgeblendet, um die Übersichtlichkeit zu wahren. Die komplette GMAP3D des in der Mitte dargestellten Volumens besitzt daher deutlich mehr Darts als angezeigt.

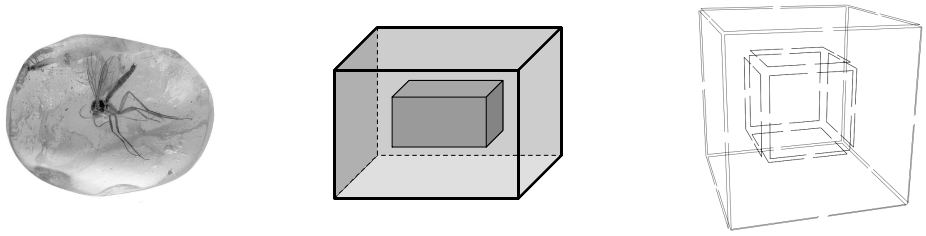


Abbildung 67: Repräsentation eines Volumeneinschlusses als GMAP3D (rechts). Der große „Würfel“ stellt die äußere Schale des Steins dar, der kleine „Würfel“ stellt die innere Schale des Steins dar. (vgl. Abschnitt 3.4.1)

Die GMAP3D des Steins besteht aus zwei voneinander getrennten „Würfeln“, deren Darts jeweils keine Verbindung zu Darts des anderen haben. Das ist in diesem Fall aber nicht sinnvoll, da nur beide „Würfel“ zusammen die Begrenzung des Volumens, welches den Bernstein repräsentiert, darstellen. Möchte man die gesamte Begrenzung eines Objektes anzeigen, beziehungsweise alle zugehörigen Darts zuordnen, so muss eine Verbindung zwischen den Darts des inneren und äußeren „Würfels“ erzeugt werden, so dass ein Traverser alle erreichen kann. Auch die anderen beschriebenen Fälle aus Abschnitt 3.4.1 müssen entsprechend behandelt werden.

Da es vermieden werden sollte, weitere Darts einzuführen, die nicht für eine Kontur notwendig sind, sondern nur als Verbindung zwischen nicht verbundenen Konturteilen dienen, stellt die XGMAP3D die Möglichkeit dar, eingeschlossene Konturen einer äußeren zuzuordnen. Der Vorteil ist hierbei, dass sofort ersichtlich wird, welcher Teil der Begrenzung zur äußeren Schale und welcher zu inneren Schalen gehört. Bezogen auf das

Beispiel mit dem Bernstein kann somit schnell die Aussage getroffen werden, dass das Insekt vom Bernstein eingeschlossen ist und nicht etwa der umgekehrte Fall vorliegt. Dies liegt darin begründet, dass die Kontur des Insekts nur mit der inneren Schale des Steins über das β_3 -Orbit vernäht ist. Würde das Insekt den Stein umschließen, so wäre seine Kontur mit der äußeren Begrenzung des Steins über das β_3 -Orbit vernäht.

Um solche Einschlüsse zu repräsentieren, wird in der XGMAP3D der Datentyp DARTVOLUME zur Verfügung gestellt. Dieser Datentyp und dessen Erstellung wird in Abschnitt 7.2.3 beziehungsweise 7.3.3 beschrieben. Über die Orbits zusammenhängende Teile einer Kontur werden in sogenannten DARTSURFACEPARTS repräsentiert (siehe Abschnitt 7.2.2 bzw. 7.3.2). Mehrere DARTSURFACEPARTS können wiederum eine äußere oder eine innere Schale des Volumens bilden. DARTSURFACEPARTS gehören auch dann zu einer gemeinsamen Schale eines Volumens (äußere oder innere), wenn sie nicht über die Orbits miteinander verbunden sind, aber jeweils Darts gemeinsam auf einer Fläche haben.

Dieser Fall liegt beispielsweise bei der Baumhöhle in Abschnitt 3.4.1 vor. Bei diesem Beispiel haben zwei DARTSURFACEPARTS Darts auf einer Fläche, die in der XGMAP3D DARTFACE (siehe Abschnitt 7.2.1 bzw. 7.3.1) genannt wird. In diesem Fall wird die Schale durch zwei DARTSURFACEPARTS repräsentiert. Eine entsprechende XGMAP3D, bei der nur die Darts der Oberfläche des Baumes angezeigt werden, zeigt Abbildung 68. In diesem Beispiel gibt es keine vollständig eingeschlossene Kontur und somit auch nur eine Schale. Diesen Sachverhalt kann man sich veranschaulichen, wenn man sich eine Ameise vorstellt, die über diese Oberfläche krabbelt. Die Ameise kann alle Punkte der Oberfläche erreichen, ohne diese jemals verlassen zu müssen. Gäbe es eine innere Schale, so wäre diese für die Ameise nicht erreichbar.

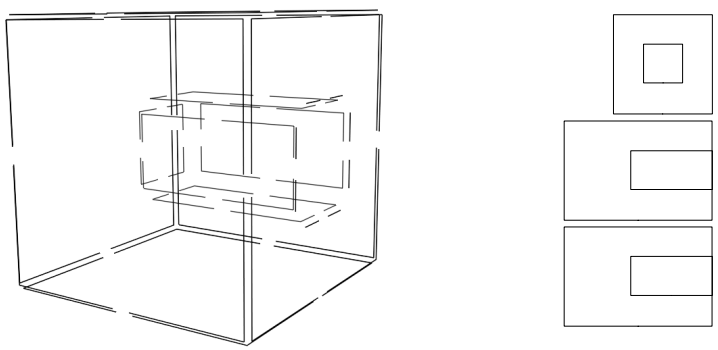


Abbildung 68: Links ist die GMAP3D-Repräsentation einer Höhle abgebildet, auf der rechten Seite deren Drei-Tafel-Projektion.

Der Vorteil der DARTFACES ist der, dass auch diese Datenstruktur unterscheidet, welche Flächenkontur eine äußere Kontur ist und welche eine innere – also ein Loch – ist.

Die XGMAP3D ermöglicht somit die Repräsentation der Enthalten-Relation, ohne dass zusätzliche Darts eingeführt werden müssen, die sich in ihrer Funktion von den anderen unterscheiden. Des Weiteren erlaubt sie die schnelle Untersuchung der Objekte

hinsichtlich der Anzahl ihrer begrenzenden Schalen, wobei jede innere Schale einen Einschluss von anderen Objekten darstellt.

7.2 Datentypen zur Speicherung der Zusammengehörigkeit von Darts

Dieses Unterkapitel beschäftigt sich mit der Modellierung der einzelnen abstrakten Datentypen der dreidimensionalen XG-Map. Diese Modellierung gestattet es, die in dem vorigen Unterkapitel erwähnten Probleme auf eine elegante Art so abzubilden, dass sie handhabbar werden.

Bei der vorliegenden Modellierung steht das allgemeine Konzept des Darts (siehe Abschnitt 6.1.1) im Mittelpunkt. Dies hat einige Vorteile. So vereinfacht es die verschiedenen später erwähnten Listenstrukturen innerhalb der XGMAP3D. Zugleich stellt ein Dart indirekt den Ausgangspunkt zu einem Element der XGMAP3D dar. Dazu wurde das Konzept des Zellschlüssels eines Darts (siehe Abschnitte 3.4.5 und 6.1.1) ausgenutzt. Dieser Sachverhalt spiegelt sich in den Namen der beteiligten abstrakten Datentypen (außer der XGMAP3D selbst) wider, sie erhalten das Präfix „Dart“.

Zum Einstieg in das Kapitel werden zunächst die beteiligten abstrakten Datentypen der XGMAP3D vorgestellt, bevor zu der Beschreibung des eigentlichen Datentyps XGMAP3D übergegangen wird.

7.2.1 Der abstrakte Datentyp DARTFACE

Der Datentyp DARTFACE beschreibt eine einzelne zweidimensionale Fläche, welche Teil einer topologischen Schale ist (siehe Abschnitt 3.4.5). Wie in der Motivation erwähnt, ist dies dann erforderlich, wenn Flächen Einschlüsse enthalten, da die verschiedenen Darts dort keine Information über ihren Zusammenhang besitzen.

Laut formaler Definition der 3-XG-Map (siehe Definition 3.4.10) wird eine Fläche durch eine äußere Flächenkontur und beliebig viele innere Flächenkonturen beschrieben, die dessen Lochgrenzen beschreiben. Die Member-Variablen ergeben sich wie folgt:

- `Dart3d* faceStartDart`
- `int volumeLabel`
- `list<Dart3d*> innerContoursDartList`
- `int faceID`

Die Variable `faceStartDart` wird beim Erstellen der DARTFACE gesetzt und stellt den Zeiger auf den Anker-Dart der DARTFACE dar. Dieser besteht in dem kleinsten Dart (gemäß der Ordnungsrelation aus Definition 6.2.3) der äußeren Kontur der DARTFACE. Die Variable `volumeLabel` wird ebenfalls während der Erstellung gesetzt. Sie beschreibt die Regionsnummer des Volumens, an der die DARTFACE Teil einer Schale ist. In der Liste `innerContoursDartList` sind Zeiger auf die Anker-Darts der inneren Konturen der DARTFACE abgelegt. Die Variable `FaceID` stellt eine eindeutige Nummer dar, welche im Laufe des Erstellungsvorgangs gesetzt wird.

Die Schnittstellen der Klasse ergeben sich unmittelbar aus den Anforderungen und den damit verbundenen Member-Variablen, wie die folgende Tabelle zeigt.

Tabelle 7.2.1: Schnittstellen der Klasse DARTFACE
`Dart3d* getStartDartPointer()` gibt den Zeiger auf den Anker-Dart der DARTFACE zurück. Dieser besteht in dem kleinsten Dart der äußeren Kontur der DARTFACE.
`int getVolumeLabel()` liefert die Regionsnummer des Volumens zurück, an der die DARTFACE Teil einer Schale ist.
`addInnerContourDart(Dart3d*)` ordnet der DARTFACE eine innere Kontour zu. Diese innere Kontur wird durch den Zeiger auf den Anker-Dart der Kontur übergeben.
`list<Dart3d*> & getInnerContoursDartList()` liefert eine Referenz der Liste mit allen in der DARTFACE enthaltenen inneren Konturen zurück.
`void setFaceID(int)` ordnet der DARTFACE eine eindeutige Nummer zu.
`int getFaceID()` liefert die eindeutige Nummer einer DARTFACES zurück.

Diese Beschreibung der Schnittstelle erfüllt alle Anforderungen, die an die DARTFACE gestellt wurden. Außerdem besitzt die Klasse eine Implementation des < -Operators, auf welchen in Kapitel 7.3 zur Erstellung zurückgegriffen werden wird, um den Erstellungsvorgang zu beschleunigen.

Tabelle 7.2.2: Operatoren der Klasse DARTFACE
`operator< (DartFace& )` gibt TRUE zurück, falls die Regionsnummer der aktuellen DARTFACE kleiner als die der zu vergleichenden ist.

Somit ermöglicht die Klasse DARTFACE, den Zusammenhang von zweidimensionalen Flächen und deren Einschlüssen zu repräsentieren, was durch die GMAP3D nicht in allen Fällen gegeben ist.

7.2.2 Der abstrakte Datentyp DARTSURFACEPART

Die Notwendigkeit des nächsten Datentypes folgt direkt aus der Notwendigkeit der DARTFACE. Denn sobald Flächeneinschlüsse auftreten, zerfallen Schalen in mehrere Oberflächenteile, die lediglich über die inneren Konturen der Flächen miteinander verbunden sind, nicht aber innerhalb der 3-G-Map eine Zusammenhangskomponente bilden. Ein Beispiel, das die Einführung eines Oberflächenteils als Datentyp motiviert, ist durch die Abbildung 69 gegeben.

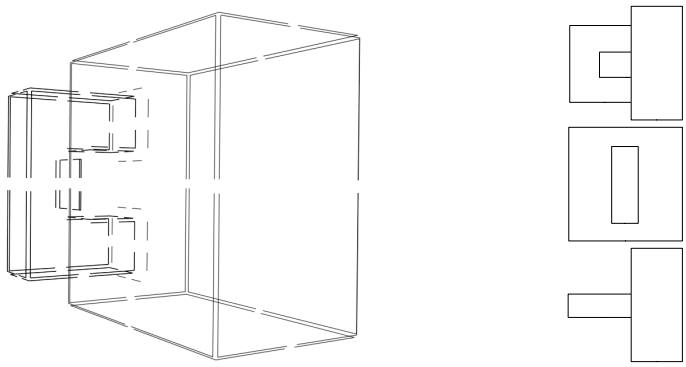


Abbildung 69: Beispiel zur Motivation von Oberflächenteilen als eigene Datenstruktur. Die äußere Schale des Objektes ist in zwei getrennte Oberflächenteile zerfallen, welche über eine Fläche mit inneren Flächenkonturen miteinander verbunden sind. (rechts befindet sich die Drei-Tafel-Projektion)

Innerhalb der zwei Flächeneinschlüsse des quaderförmigen Oberflächenteils liegt jeweils der gleiche, zweite Oberflächenteil eingebettet. Nur die Einführung eines Oberflächenteils ermöglicht hier eine baumartige Repräsentation der äußeren Schale des Volumens. Diese baumartige Repräsentation ist besonders dann wichtig, wenn es darum geht, die Visualisierung der XGMAP3D mit Hilfe der in Kapitel 8 beschriebenen Anwendung zu realisieren.

Nach dieser einführenden Motivation wird der abstrakte Datentyp des DARTSURFACEPARTS beschrieben. Dazu werden zunächst die Member-Variablen aufgezählt und erläutert:

- `list<Dart3d*> facesStartDartsList`
- `int volumeLabel`
- `int surfacePartID`

Prinzipiell beschreibt ein DARTSURFACEPART, wie in Abschnit 3.4.5 vorgestellt, eine $\beta_0 \circ \beta_1 \circ \beta_2$-Zusammenhangskomponente. Um schnelleren Zugriff auf die in einem Oberflächenteil enthaltenen Flächen zu gewährleisten, wird dennoch nicht nur ein Anker-Dart, sondern eine Liste von Zeigern auf Darts – die `facesStartDartsList` – gespeichert. Sie speichert die Anker der enthaltenden DARTFACES. Die Variable `volumeLabel` wird während der Erstellung eines DARTSURFACEPARTS gesetzt. Sie beschreibt die Regionsnummer des Volumens, zu dem der Oberflächenteil gehört. Die `surfacePartID` wird während des Erstellungsvorgangs zu einem eindeutigen Bezeichner jedes einzelnen Oberflächenteils.

Da aber ein direkter Zugriff auf die Member-Variablen im Rahmen der objekt-orientierten Kapselung nicht möglich ist, wird im Folgenden die Schnittstelle der Klasse DARTSURFACEPART beschrieben. Ihre Funktionalität liegt nahezu einzig im Zugriff auf die Member-Variablen begründet, wie in der folgenden Tabelle ersichtlich ist.

Tabelle 7.2.3: Schnittstellen der Klasse DARTSURFACEPART
`int getVolumeLabel()` liefert die Regionsnummer des Volumens zurück, an dem der DARTSURFACEPART Teil einer Schale ist.
`list<Dart3d*> & getFacesStartDarts()` liefert eine Referenz der Liste zurück, die zu jeder enthaltenen DARTFACE den Zeiger auf den zugehörigen Anker-Dart enthält.
`setFacesStartDarts(list<Dart3d*> & )` ersetzt die Liste, die zu jeder enthaltenen DARTFACE den Zeiger auf den zugehörigen Anker-Dart enthält, durch die übergebene Liste von Zeigern auf Darts.
`void setSurfacePartID(int)` ordnet dem DARTSURFACEPART eine eindeutige Nummer zu.
`int getSurfacePartID()` liefert die eindeutige Nummer eines DARTSURFACEPARTS zurück.

Abschließend lässt sich zusammenfassen, dass in dieser Arbeit die Klasse DARTSURFACEPART vor allem zwei wichtige Eigenschaften liefert:

1. Sie vereinfacht die Repräsentation der Zusammenhangskomponenten in der Art, dass sich mit ihrer Hilfe Schalen in GMAP3D-Zusammenhangskomponenten unterteilen lassen.
2. Sie ermöglicht es, die Topologie eines Volumens in einer Baumstruktur zu repräsentieren. Dieses erhöht die Übersichtlichkeit und macht es zudem möglich, die Struktur in der graphischen Benutzungsoberfläche in einer Baumansicht zu präsentieren.

Auch wenn diese Klasse die topologische Ausdruckskraft nicht erhöht, so ist sie dennoch – aufgrund der beiden obigen Punkte – als eine sinnvolle Ergänzung zu betrachten.

7.2.3 Der abstrakte Datentyp DARTVOLUME

Stellte die Klasse DARTFACE die Möglichkeit dar, dass Flächeneinschlüsse einer gegebenen Fläche beschrieben werden konnten, so liefert die Klasse DARTVOLUME selbiges für dreidimensionale Regionen und deren Volumeneinschlüsse. Beispiele für diese Einschlüsse finden sich in Kapitel 7.1.

Die Datenstruktur der Klasse DARTVOLUME enspricht in etwa der der DARTFACE. Der abstrakte Datentyp besitzt folgende Member-Variablen:

- `Dart3d* volumeStartDart`
- `list<Dart3d*> innerVolumecontoursDartList`
- `int volumeLabel`

Einem DARTVOLUME wird bei der Erstellung ein Zeiger auf einen Anker-Dart der äußeren Volumenkontur zugewiesen. Dieser wird in der Variablen `volumeStartDart` abgelegt.

Zudem wird bei der Erstellung die Regionsnummer der ikonischen Regionenrepräsentation in der Variable `volumeLabel` abgelegt. Sie dient ebenfalls als eindeutiger Bezeichner eines jeden DARTVOLUMES. Analog zum zweidimensionalen Fall wird hier auch eine Liste von Zeigern auf Darts verwendet, um die Volumeneinschlüsse zu speichern. Zu diesem Zweck werden in `innerVolumecontoursDartList` die Zeiger der Anker-Darts der jeweiligen inneren Schalen gespeichert.

Wichtig anzumerken ist an dieser Stelle, dass pro Schale nur ein Zeiger auf einen Dart gespeichert wird. Es ist aber dennoch sichergestellt, damit die komplette Schale zu beschreiben, da ein DARTSURFACEPART jeder Schale ausreicht, um die komplette Schale zu beschreiben. Dies liegt darin begründet, dass, ausgehend von den DARTFACES des einen DARTSURFACEPARTS, alle verbundenen DARTSURFACEPARTS transitiv über innere Flächenbegrenzungen erreicht werden können.

Die einzelnen Member-Variablen werden durch eine klar definierte Schnittstelle durch den direkten Zugriff von Außen gekapselt. Diese Schnittstelle wird durch folgende Tabelle beschrieben:

Tabelle 7.2.5: Schnittstellen der Klasse DARTVOLUME

`Dart3d* getStartDartPointer()` gibt den Zeiger auf den Anker-Dart des DARTVOLUMES zurück. Dieser besteht aus dem kleinsten Dart (gemäß der Ordnungsrelation aus Definition 6.2.3) der äußeren Kontur des DARTVOLUMES.
`int getVolumeLabel()` liefert die Regionsnummer des DARTVOLUMES zurück, die zugleich eindeutiger Bezeichner ist.
`addInnerVolumecontourDart(Dart3d*)` ordnet dem DARTVOLUME eine innere Schale zu. Diese innere Schale wird durch den Zeiger auf den Anker-Dart der Kontur übergeben.
`list<Dart3d*> & getInnerVolumecontoursDartList()` liefert eine Referenz der Liste mit allen in dem DARTVOLUME enthaltenen inneren Schalen zurück.

Mit der hier vorgestellten Repräsentation ist es möglich, Volumeneinschlüsse darzustellen. Dieses war das letzte noch fehlende topologisch notwendige Element, welches zur Repräsentation der XGMAP3D benötigt wurde.

7.2.4 Der abstrakte Datentyp der XGMAP3D

Die Implementation der dreidimensionalen XG-Map erfolgt in dieser Arbeit durch den abstrakten Datentyp XGMAP3D. Da dieser allerdings auf den vorher vorgestellten Datentypen aufbaut, wird er erst jetzt beschrieben.

Die Datenstruktur lässt sich recht gut in zwei Bereiche aufteilen, einen speichernden Bereich, welcher im Wesentlichen die verschiedenen Exemplare vom Typ DARTFACE, DARTSURFACEPART und DARTVOLUME bereitstellt, und einem verbindenden Teil, welcher,

gegeben einen Dart der GMAP3D, eine Verbindung zu diesen Strukturen ermöglicht. Da dieser Datentyp eine recht umfangreiche Schnittstelle bereitstellt, wird zunächst der speichernde Teil der Klasse erläutert, bevor zu dem verbindenden Teil übergegangen wird.

Der speichernde Teil der Klasse enthält folgende Member-Variablen:

- `list<DartVolume> dartVolumes`
- `list<DartSurfacePart> dartSurfaceParts`
- `list<DartFace> dartFaces`

Diese Member-Variablen stellen die Speicherorte der einzelnen Elemente der XGMAP3D dar. So befinden sich alle DARTVOLUMES in der Liste `dartVolumes`, alle DARTSURFACEPARTS in der Liste `dartSurfaceParts` et cetera.

Die Schnittstelle des speichernden Teils der XGMAP3D besteht lediglich aus folgenden Zugriffsfunktionen auf die oben beschriebenen Member-Variablen, wie die folgende Tabelle zeigt:

Tabelle 7.2.6: Schnittstellen der speichernden Elemente der Klasse XGMAP3D

`list<DartVolume> & getDartVolumes()` gibt eine Referenz auf die Liste `dartVolumes` zurück.
`list<DartSurfacePart> getDartSurfaceParts()` gibt eine Referenz auf die Liste `dartSurfaceParts` zurück.
`list<DartFace> getDartFaces()` gibt eine Referenz auf die Liste `dartFaces` zurück.

Neben diesen Containern für die bei der Erstellung entstehenden Datenstrukturen, besitzt die XGMAP3D noch eine Schnittstelle zu den Darts, auf denen sie aufsetzt. So ist es nicht nur möglich, von den Datenstrukturen der XGMAP3D auf die Darts der GMAP3D zuzugreifen, sondern auch von einem beliebigen Dart der GMAP3D auf die Elemente der XGMAP3D zuzugreifen. Zu diesem Zweck stellt die XGMAP3D folgende zusätzliche Member-Variablen bereit:

- `map<int, DartFace*> dartFaceHashMap`
- `map<int, DartSurfacePart*> dartSurfacePartHashMap`
- `map<int, DartVolume*> dartVolumeHashMap`
- `map<int, std::list<DartSurfacePart*> > surfaceHashMap`

Die ersten drei Variablen stellen Hash-Tabellen dar, die Paare aus der DartID eines Anker-Darts des jeweiligen Orbits und Zeiger auf die dazugehörige Datenstruktur der XGMAP3D enthalten. Dies bedeutet für die einzelnen Tabellen folgendes:

1. Falls ein Dart d der GMAP3D G ein Anker des $\beta_0 \circ \beta_1$ -Orbits ist, so lässt sich über die DartID von d und die Hash-Tabelle `dartFaceHashMap` die DARTFACE finden, zu der dieser Dart gehört.
2. Falls ein Dart d der GMAP3D G ein Anker des $\beta_0 \circ \beta_1 \circ \beta_2$ -Orbits ist, so lässt sich

a) über die DartID von *d* und die Hash-Tabelle `dartSurfacePartHashMap` der DARTSURFACEPART finden, zu dem dieser Dart gehört.

b) über die DartID von *d* und die Hash-Tabelle `dartVolumeHashMap` das DARTVOLUME finden, zu dem dieser Dart gehört.

Die Abbildung 70 veranschaulicht den Zugriff auf die Hash-Tabellen der XGMAP3D.

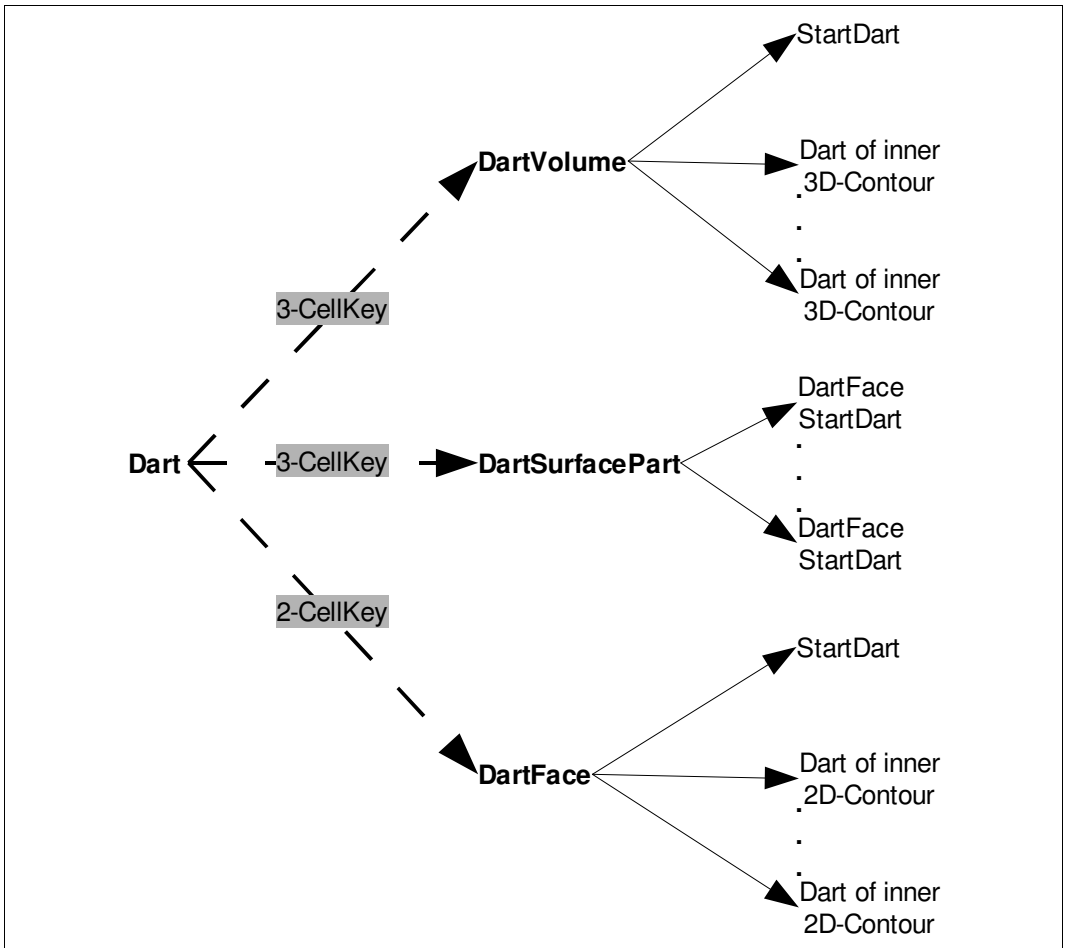


Abbildung 70: Die Datenstruktur der XGMAP3D. Dicke gestrichelte Pfeile stellen den Zugriff über Hash-Tabellen – mittels der grau hinterlegten Bedingung des Ankers des Darts – dar. Dünne durchgehende Pfeile stellen die Rolle der Darts als Member-Variablen der Datenstrukturen der XGMAP3D dar. Der Dart als zentrale Komponente wird hier gut sichtbar, steht er doch am Anfang und am Ende des Diagramms.

Für die Hash-Tabelle `surfaceHashMap` gibt es eine besondere Funktionalität. Sie speichert den Zusammenhang zwischen mehreren DARTSURFACEPART-Elementen zu einer Volumenschale. Dies geschieht dadurch, dass für jeden Dart, auf den ein DARTVOLUME – mittels innerer oder äußerer Volumenschale – zeigt, die zugehörige DartID dazu benutzt

werden kann, um mit Hilfe der `surfaceHashMap` eine Liste aller DARTSURFACEPART-Elemente zu erhalten, die zu der entsprechenden Schale gehören.

Diese Funktionalität ist rein topologisch betrachtet überflüssig, zumal sie die Ausdruckskraft der bis hierher beschriebenen Datenstruktur nicht erweitert. Allerdings erweist es sich in der Praxis als vorteilhaft, einen schnellen Zugriff auf alle Oberflächenteile einer Schale zu erhalten, um beispielsweise die graphische Baumansicht der XGMAP3D in der Anwendung (siehe Kapitel 8) schneller aufzubauen. Mit den vorhandenen Möglichkeiten der Traversierung der XGMAP3D wäre es ebenfalls möglich, die gewünschte Funktionalität zu erhalten, allerdings mit einem sehr viel höheren Aufwand.

Die oben beschriebenen Hash-Tabellen stellen die Grundlage der Verbindung zwischen Darts der GMAP3D und Datenstrukturen der XGMAP3D dar. Der Zugriff auf diese Tabellen erfolgt durch eine Schnittstelle, deren Funktionen in der folgenden Tabelle aufgeführt und beschrieben werden.

Tabelle 7.2.6: Schnittstellen der verbindenden Elemente der Klasse XGMAP3D

`map<int, DartFace*>& getDartFaceHashMap()`
: gibt eine Referenz auf die Hash-Tabelle `dartFaceHashMap` zurück.

`map<int, DartSurfacePart*>& getDartSurfacePartHashMap()`
: gibt eine Referenz auf die Hash-Tabelle `dartSurfacePartHashMap` zurück.

`map<int, DartVolume*>& getDartVolumeHashMap()`
: gibt eine Referenz auf die Hash-Tabelle `dartVolumeHashMap` zurück.

`map<int, list<DartSurfacePart*>>& getSurfaceHashMap()`
: gibt eine Referenz auf die Hash-Tabelle `surfaceHashMap` zurück.

`DartVolume* getDartVolumeFromDart(Dart3d *)`
: Falls der Dart, auf den der übergebene Zeiger zeigt, ein 3-Zellschlüssel eines DARTVOLUMES ist, so wird ein Zeiger auf dieses zurückgeliefert. Sonst wird auf der Schale, zu der der Dart gehört, solange gesucht, bis ein 3-Zellschlüssel eines DARTVOLUMES gefunden wird, und der Zeiger auf dieses zurückgeliefert.

`DartSurfacePart* getDartSurfacePartFromDart(Dart3d *)`
: Falls der Dart, auf den der übergebene Zeiger zeigt, ein 3-Zellschlüssel ist, so wird ein Zeiger auf dessen DARTSURFACEPART zurückgeliefert. Ansonsten wird ausgehend vom Dart mit den Orbits β_0, β_1 und β_2 gesucht, bis ein passender 3-Zellschlüssel gefunden wird, und der Zeiger auf dessen DARTSURFACEPART zurückgeliefert.

`DartFace * getDartFaceFromDart(Dart3d *)`
: Falls der Dart, auf den der übergebene Zeiger zeigt, ein 2-Zellschlüssel ist, so wird ein Zeiger auf dessen DARTFACE zurückgeliefert. Ansonsten wird ausgehend vom Dart mit den Orbits β_0, β_1 gesucht, bis ein passender 2-Zellschlüssel gefunden wird, und der Zeiger auf dessen DARTFACE zurückgeliefert.

`Dart3d * getDartFaceContourDartFromDart(Dart3d *)`
: liefert für den Dart, auf den der übergebene Zeiger zeigt, den Zeiger auf den Anker-Dart des $\beta_0 \circ \beta_1$ -Orbits zurück.

Aus der vorgestellten Modellierung der XGMAP3D ergeben sich folgende Konsequenzen für einen Dart, der ein 2-Zellschlüssel ist:

- Er ist eindeutig für jede Flächenkontur definiert, das heißt, es gibt nur einen für jedes $\beta_0 \circ \beta_1$ -Orbit.
- Es lässt sich mit Hilfe der XGMAP3D direkt herausfinden, zu welcher DARTFACE er gehört. Falls er gleich dem Anker-Dart der DARTFACE ist, so bildet er eine äußere Kontur, sonst ist er Teil einer inneren Kontur.

Folgendes gilt für einen Dart, der einen 3-Zellschlüssel darstellt:

- Er ist eindeutig für jedes Oberflächenteil definiert, das heißt, es gibt nur einen für jedes $\beta_0 \circ \beta_1 \circ \beta_2$ -Orbit.
- Mit Hilfe der XGMAP3D kann direkt herausgefunden werden, zu welchem DARTSURFACEPART der Dart gehört, da seine DartID als Hash-Wert dient.
- Man gelangt nicht notwendigerweise von diesem Dart direkt mit Hilfe der XGMAP3D zu einem DARTVOLUME-Element, da sich die Schalen dessen eventuell in mehrere DARTSURFACEPART-Elemente unterteilen und somit nicht gewährleistet ist, dass der aktuelle Dart der Anker einer der Schalen ist.

  Daher kann das Volumen von einem DARTSURFACEPART über zwei Wege erreicht werden:

  1. Die Regionsnummer des Volumens, zu dem der DARTSURFACEPART gehört, kann benutzt werden, um in der Liste der DARTVOLUMES zu suchen und so das passende DARTVOLUME zu finden. Diese Methode ist besonders dann sinnvoll, wenn sich die Regionen aus sehr vielen Darts zusammensetzen und die Liste der DARTVOLUMES überschaubar ist.
  2. Ausgehend vom Dart kann die komplette Schale des Volumens, zu dem der DARTSURFACEPART gehört, traversiert werden. Bei diesem Durchqueren wird man auf weitere 3-Zellschlüssel stoßen, von denen die DartID eines Darts direkt auf das DARTVOLUME verweist. Dieser Ansatz ist dann sinnvoll, wenn die Regionen relativ wenige Darts enthalten und die Liste der DARTVOLUMES sehr groß ist.

Mit Hilfe der vorliegenden Klasse der XGMAP3D ist es nicht nur möglich, von den „High-Level“-Elementen DARTFACE, DARTSURFACEPART und DARTVOLUME auf die damit verbundenen Darts der GMAP3D zuzugreifen. Auch der umgekehrte Weg ist, wie oben aufgezeigt, möglich. Dies ist besonders wichtig, wenn beide Repräsentationen benötigt werden, um eine Aufgabe zu bewältigen, wie zum Beispiel das Ablaufen einer kompletten Volumenschale (siehe 7.5.1.3).

7.3 Erstellung der XGMAP3D aus der GMAP3D

Nachdem der Aufbau aller benötigten Datenstrukturen der XGMAP3D im obigen Abschnitt erläutert wurde, soll an dieser Stelle die Beschreibung für die korrekte Erstellung dieser aus beliebigen GMAP3Ds erfolgen.

Die Erstellung einer XGMAP3D erfolgt in der Funktion `createXGMap` der Klasse XGMAP3DCREATOR. Diese Funktion benötigt als Übergabeparameter eine Referenz auf eine XGMAP3D, sowie Zugriff auf das Volumen in der ikonischen Regionenrepräsentation. Die XGMAP3D, auf die referenziert wird, muss vorher mit dem entsprechenden Standardkonstruktor erstellt werden. Alle Informationen der XGMAP3D werden dann innerhalb dieser Funktion erstellt. Das Volumen wird für mehrere Operationen benötigt. Zum einen ist es nicht notwendig, dass der übergebenen XGMAP3D schon eine GMAP3D zugewiesen wurde, denn in diesem Fall wird dessen Erstellung zunächst nachgeholt. Des Weiteren wird das Volumen bei der Zuweisung der Regionsnummer benötigt.

Der Algorithmus 7.3.1 zeigt die Schritte, die notwendig sind, um eine XGMAP3D mit Informationen zu füllen.

Algorithmus 7.3.1: Erzeugung einer XGMAP3D

```
function createXGMap

   input : xGMap, volume

   if gmap not created
      create gmap

   create DartFaces with inner contours
   create DartSurfaceParts
   create DartVolumes with inner shells
```

Nachdem alle Darts der GMAP3D erstellt wurden, werden nacheinander die Strukturen erstellt, die in der XGMAP3D benötigt werden. Dabei werden zunächst die in der GMAP3D vorhandenen Flächen gesucht und erstellt, wobei innere Konturen zugeordnet werden müssen. Anschließend werden alle Oberflächenteile gesucht und mit Hilfe der Flächen erzeugt. Im Anschluss daran lassen sich alle vollständigen Begrenzungen aller Regionen finden, welche aus mehreren nicht zusammenhängenden Schalen bestehen können, beispielsweise wenn es innere Schalen gibt.

7.3.1 Finden von zusammengehörigen Darts einer Fläche

Das Erstellen der Flächen übernimmt die Funktion `findFaces`, wobei zunächst innere Konturen nicht beachtet werden und ebenfalls als Flächen gespeichert werden. Dieser Sachverhalt ergibt sich, da Flächenkonturen gesucht werden, indem, vom ersten Dart der GMAP3D ausgehend, alle Darts markiert werden, die über β_0 - und β_1 -Orbits zu erreichen sind. Anschließend wird zum nächsten nicht markierten Dart der Dartliste der GMAP3D weitergegangen. Von diesem ausgehend, werden wiederum alle erreichbaren Darts markiert. Dieser Vorgang wird wiederholt, bis das Ende der Dartliste erreicht ist.

Jeder nicht markierte Dart, der in der Dartliste gefunden wird, bildet den Anker für eine zu erstellende DARTFACE. Des Weiteren muss zu diesem Dart noch die Regionsnummer des Volumens herausgefunden werden, zu dessen Begrenzung er gehört. Dies geschieht, indem über die Funktion `getVoxelFromDartID` der Voxel herausgefunden wird, für den der Dart erstellt wurde (siehe hierzu Abschnitt 6.1.2). Das Label dieses Voxels ist die gesuchte Nummer.

Da der Dart als Anker für eine Fläche dient, muss er entsprechend als 2-Zellschlüssel markiert werden. Beim Durchsuchen einer Menge von Darts kann dieser so leicht als Repräsentant dieser Flächenkontur erkannt werden.

Mit den beiden gefundenen Werten – Zeiger auf einen nicht markierten Dart, sowie dessen Regionsnummer – kann nun ein Objekt der Klasse DARTFACE erstellt werden. Die Zuweisung der FaceID erfolgt durch eine Durchnummerierung aller gefundenen Flächenkonturen. Diese Namensgebung dient vor allem der späteren Verwendung in der Anzeige einer XGMAP3D, so dass der Benutzer einzelne Flächen schnell unterscheiden kann.

Damit über den Dart, der durch den gesetzten Zellschlüssel als Repräsentant dieser Flächenkontur dient, schnell auf die DARTFACE zugegriffen werden kann, wird noch ein Eintrag in der Hash-Tabelle `DartFaceHashMap` erstellt.

Algorithmus 7.3.2: Finden von Flächen in einer GMAP3D

```
function findfaces

   input : XGMap3d, volumen

   for all darts in the GMap3d
      if dart is currently not marked
         make dart cellKey of a face
         mark all other darts reachable with max beta-1-Orbit
         create dartface object


function findInnerFaces

   input : XGMap3d

   for all dartfaces in the XGMap3d
      compare with other dartFaces
      if another dartFace is a inner contour of current dartFace
         make the other dartFace a inner contour of current dartFace
         delete the other dartFace


function pointInPoly

   input : point dart, polygon dart, plain
   output: bool

   current dart is the polygon dart
   while polygon dart is not visited twice
      shoot a ray from point dart in plain-x direction
      if ray intersects line between coordinates of current dart and its beta-0
      dart on plain
         count up crossovers
      new current dart is the beta-0-beta-1 dart of old current dart
```

```
if number of crossovers is odd
   return true
else
   return false
```

In Algorithmus 7.3.2 ist der Ablauf zum Finden der Flächen in einer GMAP3D dargestellt.

Nachdem zunächst eventuell zu viele DARTFACES erstellt wurden, müssen nun alle daraufhin untersucht werden, ob sie nur eine innere Kontur einer anderen sind. Sollte dies herausgefunden werden, so müssen diese DARTFACES aus der XGMAP3D gelöscht werden, und der umgebenden Kontur als innere Kontur genannt werden.

Die Überprüfung läuft in mehreren Teilschritten in der Funktion `findInnerFacecontours` ab, um sie so effizient wie möglich zu machen und nicht jede zuerst erstellte DARTFACE daraufhin zu untersuchen, ob all ihre Ecken innerhalb der Kontur einer anderen DARTFACE liegen. Dieser Vergleich ist relativ aufwendig und wird in der Funktion `pointInPoly` ausgeführt (siehe hierzu Algorithmus 7.3.2).

Aufgrund dessen werden nur DARTFACES miteinander verglichen, die die selbe Regionsnummer haben. Durch ein vorheriges Sortieren aller DARTFACES nach der Regionsnummer braucht man in der Liste aller DARTFACES in der XGMAP3D nur eine geringe Anzahl an anderen DARTFACES beachten. Man beginnt bei der aktuellen DARTFACE und untersucht die weiteren Elemente der Liste, bis eine DARTFACE mit einer anderen Regionsnummer erreicht wird. Dies ist möglich, da innere Konturen grundsätzlich nach äußeren Konturen in der Liste liegen. Dieser Sachverhalt ist durch die Berechnung der DartID und der Suche nach Konturen gegeben. Bei der Suche wird die Liste aller Darts betrachtet. Die Darts sind in ihr der Größe nach sortiert (das Vorzeichen eines Darts hat für die Größe keine Bedeutung, sondern nur der absolute Wert). Wird ein nicht markierter Dart gefunden, so werden von ihm aus alle erreichbaren Darts markiert. Alle Konturen werden somit von ihrem kleinsten Dart repräsentiert. Da innere Konturen innerhalb der äußeren Kontur liegen, müssen auch deren kleinste Darts größer sein als der kleinste Dart der äußeren Kontur.

Haben zwei DARTFACES die gleiche Regionsnummer, so muss geprüft werden, ob beide auf einer Ebene liegen, denn nur wenn dies gegeben ist, macht es Sinn zu prüfen, ob eine DARTFACE innerhalb der anderen liegt. Die Überprüfung erfolgt anhand der DartID der Anker der beiden Konturen. Aus diesen Nummern lässt sich berechnen, aus welchen Dartplatzhaltern die Darts erstellt wurden. Zunächst prüft man daher, ob beide Dartplatzhalter auf ein und der selben Voxelseite lagen. Wie bereits in Kapitel 6.2.3 beschrieben, berechnet sich die DartID aus der Lage des Dartplatzhalters in der Volume-Map ($\pm[(x+(w\cdot y)+(w\cdot h\cdot z))\cdot 24+(i+1)]$). Es gibt 24 Positionen für Dartplatzhalter auf einem Voxel einer Volume-Map (sie fließen als i in die Gleichung ein), von denen jeweils vier auf einer Voxelseite liegen. Zur Berechnung der Seite, auf der der Dartplatzhalter lag, ist zunächst eins abzuziehen (dies wurde aufaddiert, damit jede DartID in einem Wertebereich von $[1\ldots 24]$ plus der Position des Voxels liegt). Anschließend muss dieser Wert nur noch der Operation Modulo 24 unterzogen werden. Die Voxelseite erhält man nach der Division mit vier, indem nur der ganzzahlige Wert betrachtet wird. Koplanar sind zwei Konturen mit diesen Darts, wenn zudem die entsprechende Dimension der Einbettung beider Darts gleich ist. Stammen beispielsweise

beide Dartplatzhalter von der *North*-Seite (vgl. Abschnitt 6.2.1), so muss von beiden Einbettungen die Y-Koordinate gleich sein, damit die Flächen der Darts koplanar sind.

Die Würfelseite bestimmt somit auch die Ebene, auf der beide Flächen koplanar sind. Diese Ebene wird benötigt, um schlussendlich zu überprüfen, ob die eine Fläche wirklich eine innere Kontur der anderen ist. Die Funktion `pointInPoly` bekommt daher sowohl beide Darts als auch die herausgefundene Ebene als Parameter übergeben.

Eine Kontur befindet sich innerhalb der anderen, sobald einer ihrer Punkte – gegeben durch die Einbettung des Ankers der Kontur – innerhalb der anderen Kontur liegt. Es ist ausreichend, einen Punkt zu überprüfen, da es in der gegebene Datenstuktur keine sich überschneidenden Darts gibt. Somit liegen alle anderen Punkte ebenfalls innerhalb dieser Kontur.

Allerdings ist nach diesem Test noch nicht sichergestellt, dass es sich bei der gefundenen Kontur, die innerhalb der anderen liegt, auch wirklich um eine innere Kontur der Fläche handelt. Es gibt nämlich die Ausnahme, dass eine Kontur keine innere Flächenkontur ist, wenn sie innerhalb einer der bereits bekannten inneren Flächenkonturen liegt. Durch den Aufbau der Datenstruktur ist gewährleistet, dass solche Konturen immer nach der umgebenden inneren Kontur bearbeitet werden. Es gibt zwei Fälle, bei denen diese Konfiguration auftreten kann.

Zum einen können Doppelkonturen auftreten. Diese liegen dann vor, wenn die β_3 -Orbit-Darts beider Anker der Konturen zu Flächen mit unterschiedlichen Regionsnummern gehören. Je zwei Darts dieser Konturen haben die gleiche Einbettung. Somit sind beide Konturen – geometrisch betrachtet – identisch bezüglich ihrer Lage.

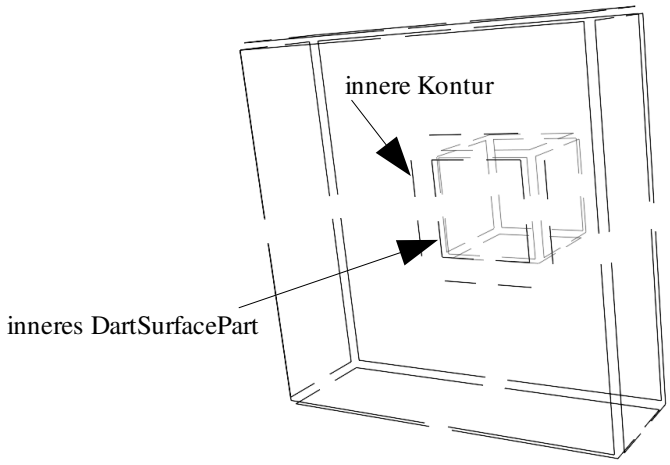


Abbildung 71: Zwei Volumen sind eingebettet im unendlichen Universum. Das kleine Volumen liegt an der Rückwand des größeren an.

In Abbildung 71 ist ein Beispiel für einen solchen Fall gegeben. Die Verschiebung der Darts, so dass keine zwei Darts auf der selben Position liegen, ergibt sich dabei aus der jeweiligen DartID und somit aus dem Voxel der Volume-Map. Es sind zwei Regionen

durch alle ihre Darts abgebildet. Die Darts der umgebenden Region sind in der Abbildung nicht mit angezeigt. Auf der hinteren Fläche der dunkleren Region erkennt man zwei innen liegende Konturen, von denen allerdings nur eine als innere Kontur erkannt werden soll. Da in diesem Sonderfall die Fläche aber gar keine Aussparung besitzt, sondern vollständig gegeben ist, muss innerhalb dieser Aussparung etwas liegen, das zur selben Fläche gehört. Um es konsistent mit allen anderen Fällen zu halten, wird dieser Teil dann als ein DARTSURFACEPART bezeichnet. Alle Darts innerhalb der Fläche entstehen nur, damit die Konsistenz der β-Orbits gewahrt bleibt. Denn die Darts des anliegenden kleinen Volumens müssen mit anderen Darts vernäht sein. Aus diesem Grund sind die eigentlich überflüssigen Darts der hinteren Fläche notwendig, und die Ausnahmeprüfung muss durchgeführt werden.

Der zweite Ausnahmefall entsteht, wenn eine Kontur wirklich innerhalb einer inneren Kontur liegt. Dies tritt dann auf, wenn an die innere Kontur der Fläche ein solcher DARTSURFACEPART anliegt, der wiederum eine Fläche besitzt, welche koplanar zur ersten Fläche ist. Vorstellen kann man sich dies als eine Art Wall, der auf einer Ebene liegt und einen inneren Teil vom äußeren dieser abtrennt. In Abbildung 72 ist dafür ein Beispiel angegeben.

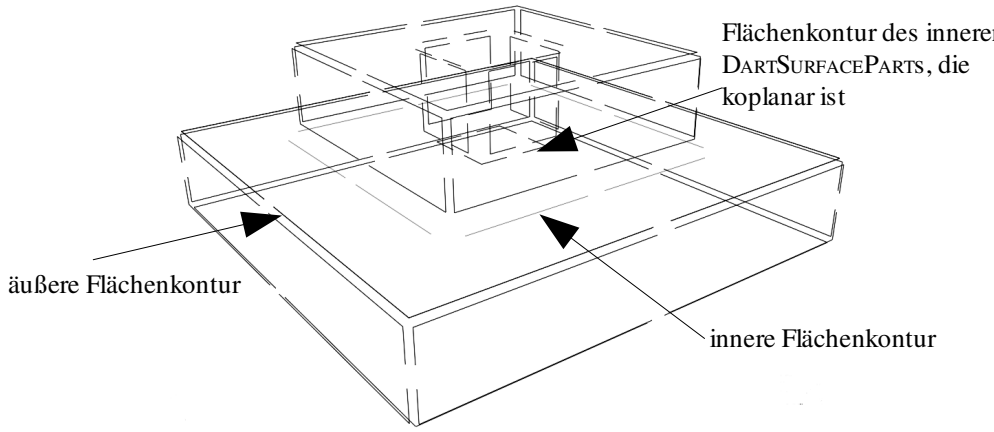


Abbildung 72: Die Flächenkontur des inneren DARTSURFACEPARTS gehört nicht zur äußeren Fläche.

Sollte nach all diesen Überprüfungen herausgefunden worden sein, dass es sich um eine innere Kontur handelt, so wird der Dart-Repräsentant der inneren Kontur der äußeren als Dart einer inneren Kontur mitgeteilt. Zudem muss die entsprechende DARTFACE aus der Hash-Tabelle gelöscht werden. Es wird ein neuer Eintrag in der Hash-Tabelle erzeugt, da der Dart, der Repräsentant der inneren Kontur ist, auch weiterhin ein 2-Zellschlüssel-Dart bleibt, der auf eine DARTFACE verweist. Es handelt sich dabei um die DARTFACE, in die seine Kontur als innere Kontur hinzugefügt wurde. Durch diesen Schritt ist es später möglich, auch wenn man sich auf einem Dart einer inneren Kontur befindet, die zugehörige DARTFACE zu erhalten, durch die man wiederum Zugriff auf alle anderen Konturen dieser DARTFACE erhält (äußere und innere). Im Anschluss daran wird die DARTFACE, welche nur eine innere Kontur darstellt, noch aus der XGMAP3D gelöscht.

Nachdem alle zuerst erstellten DartFaces überprüft wurden, befinden sich in der Liste der DartFaces nur noch vollständige Flächen, die durch eine äußere und gegebenenfalls mehrere innere Konturen begrenzt sind.

7.3.2 Finden von zusammengehörigen Flächen zu Oberflächenteilen

Oberflächenteile (DartSurfaceParts) stellen die Repräsentation für zusammenhängende Flächen – im Sinne der Erreichbarkeit über die Orbits β_0 bis β_2 – dar. Aus diesem Grund speichert jedes Oberflächenteil auch die Dart-Repräsentanten aller zusammenhängender Flächen (siehe Abschnitt 7.2.2). Die Funktion `findSurfaceParts` behandelt die Erstellung dieser Datenstruktur in der XGMap3d. In Algorithmus 7.3.3 ist das Prinzip dieses Vorgangs kurz beschrieben.

Algorithmus 7.3.3: Finden von Oberflächenteilen in einer XGMap3d

```
function  findSurfaceParts

   input : XGmap

   for all dartFaces in the XGmap
      if current dartFace is currently not part of a dartSurfacePart
         dartFace is a face of a new dartSurfacePart
         search for all dartFaces, that are reachable from current dartFace
         all found dartFaces belong to the same dartSurfacePart
```

Um den beschriebenen Algorithmus umzusetzen, wird in der Funktion `findSurfaceParts` über alle DartFaces iteriert. Bevor dieser Vorgang allerdings gestartet wird, werden alle Darts bezüglich eines bestimmten Labels demarkiert. So lassen sich später schon abgearbeitete DartFaces leicht erkennen, da ihre Dart-Repräsentanten schon markiert sind.

Wird beim Iterieren über die DartFaces eine DartFace gefunden, deren Dart-Repräsentant noch nicht markiert ist, so wird ein neuer DartSurfacePart mit der Regionsnummer der DartFace erstellt. Dieser neue DartSurfacePart wird dann an die Liste aller DartSurfaceParts der XGMap3d angehängt. Zudem wird der Dart-Repräsentant der Fläche auch zum Dart-Repräsentanten des neuen DartSurfaceParts. Der Dart wird also als 3-Zellschlüssel gekennzeichnet. Danach kann ein neuer Eintrag in der Hash-Tabelle der DartSurfaceParts erstellt werden. Die DartID des Darts dient als Schlüssel für einen Zeiger auf den DartSurfacePart.

Da aber ein DartSurfacePart meist aus mehr als einer DartFace besteht, müssen diese noch gesucht werden. Dazu wird die Funktion `traverseDartSurfacePartReturnFaces` aufgerufen. Dieser Funktion wird die XGMap3d, der Anker der aktuellen Fläche, sowie eine leere Liste von Darts, die gefüllt werden soll, und das verwendete Label zum Markieren übergeben. In der Liste von Darts sollen die Darts gespeichert werden, die Repräsentanten für Flächen sind, die vom Dart der aktuellen Fläche aus erreichbar sind, dazu gehört trivialerweise auch dieser Dart selbst.

Die Funktion `traverseDartSurfacePartReturnFaces` arbeitet auf simple Weise. Zunächst wird der übergebene Dart zum Starten auf einen Stack geschrieben und markiert. Dann wird dieser Stack abgearbeitet. Dies geschieht, indem der jeweils vorne liegende Dart

betrachtet wird und alle β_0 - bis β_2 -verbundenen Darts, die noch nicht markiert sind, ebenfalls auf den Stack geschrieben und markiert werden. Sollte der aktuell betrachtete Dart Anker einer DARTFACE sein, also als 2-Zellschlüssel gekennzeichnet sein, und keine innere Kontur repräsentieren, so wird der Dart auf die Liste der Repräsentanten von Flächen für diesen DARTSURFACEPART gespeichert. Anschließend wird der aktuelle Dart vom Stack gelöscht. Ist der Stack leer, so sind alle Dart-Repräsentanten von zugehörigen Flächen gefunden.

Nachdem die Liste nun vollständig gefüllt ist, wird der Inhalt dem erstellten DARTSURFACEPART-Objekt als `facesStartDartsList` übergeben (siehe 7.2.2). Zudem erhält der DARTSURFACEPART noch einen Bezeichner in Form einer Identifikationsnummer. DARTSURFACEPARTS werden zu diesem Zweck fortlaufend nummeriert.

Die erstellten DARTSURFACEPARTS sind aufgrund ihrer Erstellungsreihenfolge und der vorher sortierten Liste der DARTFACES ebenfalls nach ihren Regionsnummern sortiert. Ein Sortieren dieser Liste ist daher für die weiteren Schritte nicht notwendig.

7.3.3 Zusammenfassen von Oberflächenteilen zu DARTVOLUMES

Wie bereits in Abschnitt 7.2.4 beschrieben, speichert eine XGMAP3D neben den schon erstellten DARTFACES und DARTSURFACEPARTS noch eine dritte Struktur, nämlich die DARTVOLUMES. Jedes DARTVOLUME stellt alle Teile einer kompletten Begrenzung einer Region dar. Die Begrenzung umfasst dabei natürlich äußere wie innere Schalen. Bisher wurden alle gefundenen Strukturen mit der zugehörigen Regionsnummer ausgestattet, so dass leicht ersichtlich ist, welche Teile zu einer kompletten Begrenzung gehören. Allerdings ist noch keine explizite Information darüber vorhanden, welche Teile äußere Konturen oder Aussparungen beschreiben. Diese Informationen tragen die DARTVOLUMES (vgl. 7.2.3).

Die Funktion `findVolumes` führt die Erstellung der DARTVOLUMES mit oben genannten Eigenschaften aus. Die Funktionsweise wird im Algorithmus 7.3.4 kurz beschrieben.

Algorithmus 7.3.4: Erstellung von vollständigen Oberflächen in einer XGMAP3D

```
function  findVolumes

   input : XGmap

   for all dartSurfaceParts in the XGmap
      if not exists a dartVolume with same label as current dartSurfacePart
         dartSurfacePart is a outer contour of a new dartVolume
         search for all dartSurfaceParts, that are reachable from current
         dartSurfacePart
         all found dartSurfaceParts belong to the outer contour
      else
         dartSurfacePart is a inner contour of a dartVolume
         search for all dartSurfaceParts, that are reachable from current
         dartSurfacePart
         all found dartSurfaceParts belong to the same inner contour
```

In dieser Funktion werden mit Hilfe eines Iterators über die Liste der DARTSURFACEPARTS, alle diese jeweils einmal betrachtet. Aufgrund der bisherigen Erstellung aller Datenstrukturen ist gewährleistet, dass ein zuerst gefundener DARTSURFACEPART einer

Regionsnummer auch den äußeren Teil einer Begrenzung einer Region mit dieser Nummer darstellt. Alle danach gefundenen müssen demnach Schalen von Einschlüssen in dieser Region sein, soweit sie keinerlei Verbindung zu dem ersten gefundenen DartSurfacePart haben. Um dies nachzuweisen, gibt es die Funktion `traverseCompleteSurfaceComponent`, welche alle bisher verfügbaren Informationen ausnutzt, um eine Verbundenheit festzustellen. Diese Funktion arbeitet wieder mit einem Stack. Man übergibt ihr die XGMap3d, den Anker eines DartSurfaceParts und das Label, mit dem markiert werden soll. Mit der Markierung wird sichergestellt, dass kein DartSurfacePart mehr als einmal in ein DartVolume eingefügt wird, da nur nicht markierte DartSurfaceParts verarbeitet werden.

Auf den Stack kommt zunächst der übergebene Anker-Dart. Dieser wird markiert. Nun wird der Stack abgearbeitet, indem jeweils der vorderste Dart betrachtet wird und alle mit ihm über die Orbits β_0, β_1 und β_2 vernähten Darts ebenfalls auf den Stack gelegt werden, sofern sie noch nicht markiert sind. Die Markierung wird sofort ausgeführt, um ein mehrmaliges Hinzufügen zu vermeiden.

Um aber wirklich alle miteinander verbundenen Darts zu erreichen[62], spielen die eingeschlossenen Flächenkonturen eine entscheidende Rolle. Immer dann, wenn ein Dart bearbeitet wird, der als 2-Zellschlüssel gekennzeichnet ist, muss geprüft werden, ob er eine innere oder eine äußere Flächenkontur repräsentiert. Zu diesem Zweck wird geprüft, ob der Anker der zugehörigen Fläche schon markiert ist oder nicht.

Handelt es sich beim aktuellen Dart um den Anker einer inneren Kontur, so wird der Dart der äußeren ebenfalls auf den Stack gelegt, sofern dieser noch nicht markiert ist. Bei einem Repräsentanten einer äußeren Kontur ist der Anker der zugehörigen Fläche trivialerweise schon markiert. Im Anschluss daran werden alle Anker aller inneren Konturen der aktuellen Fläche ebenfalls dem Stack hinzugefügt, falls sie noch nicht markiert sind. Erst jetzt sind alle Darts gefunden, die zusammen eine Schale bilden. Dazu gehören jetzt auch die Darts, die miteinander verbunden sind, da sie einer gemeinsamen Fläche angehören.

Alle Darts, die als 3-Zellschlüssel gekennzeichnet sind und beim Abarbeiten des Stacks erreicht werden, werden in einer Liste gespeichert. Diese Darts repräsentieren jeweils einen DartSurfacePart und stellen zusammen eine vollständige innere oder äußere Schale dar. Diese Liste wird in der `surfaceHashMap` gespeichert. Als Schlüssel dient die DartID des ersten Darts.

Zuvor musste in der Funktion `findVolumes` das DartVolume erstellt werden, dem der Anker der äußere Kontur ebenfalls als Anker dient. Innere Konturen müssen den bestehenden DartVolumes als solche genannt werden. Zudem muss die Hash-Tabelle der DartVolumes gefüllt werden. Anker von den gefundenen zusammenhängenden DartSurfaceParts dienen wiederum als Schlüssel für die entsprechenden DartVolumes.

62 Es wurde bereits beschrieben, dass Darts einer inneren Flächenkontur nicht über die Orbits von einer äußeren Flächenkontur zu erreichen sind und umgekehrt.

Nachdem auch alle DARTVOLUMES gefunden und in der XGMAP3D gespeichert wurden, ist deren Erstellung vollständig abgeschlossen. Die Informationen, die beim Erstellungsvorgang einer GMAP3D verloren gegangen sind, wie zum Beispiel die Eingeschlossen-Relation, sind jetzt explizit und schnell zugreifbar. Des Weiteren ermöglicht die Repräsentation einen schnellen Überblick über den Aufbau und die Zusammensetzung von Volumenbegrenzungen in einer GMAP3D.

7.4 Optimierung der XGMAP3D und deren Erstellung

Die bisher in diesem Kapitel beschriebene Struktur und Erstellung einer XGMAP3D aus einer GMAP3D ist das Resultat einer kontinuierlichen Entwicklung in Hinblick auf Effizienz und eine übersichtliche und verständliche Struktur.

Da im Laufe der Arbeit unterschiedliche Anforderungspunkte an die XGMAP3D in den Vordergrund traten, musste sie hinsichtlich dieser angepasst werden. Dabei zeigte sich nach der Anpassung, dass der bis dahin gewählte Ansatz den Anforderungen nur noch unzureichend genügte. Daher war eine Anpassung an die neuen Gegebenheiten zwingend erforderlich.

7.4.1 Veränderungen der Datenstrukturen

In der Motivation für die XGMAP3D (siehe 7.1) wurde bereits darauf eingegangen, dass ihre Aufgabe vor allem darin besteht, Volumen- und Flächeneinschlüsse zu modellieren. Es entstanden somit zunächst Strukturen, die diese speichern konnten. So wurden die Datentypen DARTFACE und DARTVOLUME eingeführt, die diese Aufgaben erfüllten.

In den Anfängen der Datentypen unterschieden sie sich allerdings deutlich von denen am Ende vorliegenden. So fußten diese vor allem auf den Zusammenhangskomponenten von Darts über bestimmte Orbits. Beispielsweise stellten Darts, die über β_0 - und β_1 -Orbits verbunden waren, eine DARTFACE dar. Dabei spielte es keine Rolle, ob eine DARTFACE eine äußere oder eine innere Kontur darstellte. Ähnliches galt für die DARTVOLUMES.

Um die Einschlüsse zu modellieren, wurde zu jedem Objekt eine Liste von Kindern, welche die Einschlüsse darstellten, und ein Vater gespeichert. Objekte, die äußere Konturen darstellten, besaßen keinen Vater und so viele Kinder, wie es Einschlüsse gab. Innere Konturen hatten selber keine Kinder, aber dafür einen Vater. Somit gab es eine bidirektionale Verbindung zwischen den Objekten der inneren und der äußeren Konturen. Dadurch war zum Beispiel das Traversieren über eine komplette Fläche – bestehend aus einer äußeren Kontur und inneren Konturen – möglich, unabhängig davon bei welcher Kontur gestartet wurde.

In Abbildung 73 ist die ursprüngliche Gliederung der Datenstrukturen dargestellt. Allerdings ist in dieser bereits der DARTSURFACEPART integriert, welcher erst nachträglich eingeführt wurde (vgl. 7.2.2). Der Nutzen dieser Oberflächenteile liegt vor allem in der baumartigen Darstellung der XGMAP3D.

Ein großer Nachteil der hier beschriebenen Strukturierung war die Notwendigkeit der Speicherung eines Zeigers auf eine DARTFACE in jedem Dart. Dadurch erhielt der Dart Informationen über höheres Wissen, welches klar getrennt vom Dart gespeichert sein sollte. Auch die gegenseitige Kenntnis von Flächen und Volumen (später mit zwischenliegenden Oberflächenteilen) untereinander war keine optimale Lösung des Problems.

Des Weiteren gab es keine klare Unterscheidung zwischen Flächen und Flächenkonturen sowie zwischen Volumen und Schalen. Dieser Sachverhalt führte zur Fehlanwendung der Begriffe und deren eigentlichen Bedeutungen. Beispielsweise sollte eine Fläche aus einer

äußeren Flächenkontur und gegebenenfalls inneren Flächenkonturen bestehen, nicht aber aus einer äußeren Fläche und eventuellen inneren Flächen.

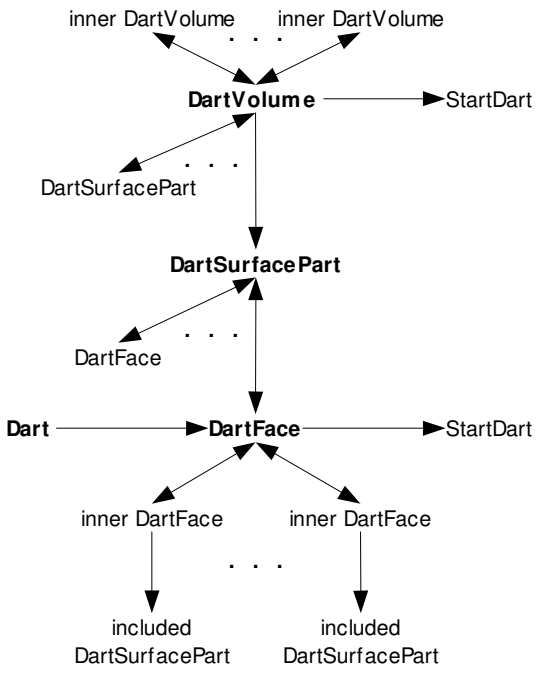


Abbildung 73: Erste Gliederung der GMAP3D und XGMAP3D Datenstrukturen.

Ein erster Schritt bestand darin, dass der Zeiger auf die DARTFACE aus dem Dart entfernt wurde. Stattdessen wurde eine Hash-Tabelle innerhalb der XGMAP3D angelegt, die den Zugriff ausgehend von einem Dart auf eine DARTFACE ermöglicht. Jeweils ein Dart einer Flächenbegrenzung wurde zu einem 2-Zellschlüssel erklärt und seine DartID diente als Schlüssel in der Hash-Tabelle, um den Zeiger auf die DARTFACE zu bekommen (siehe hierzu 7.2.1). Somit war das Problem der unklaren Trennung von Darts und höherem Wissen beigelegt.

Auf die gleiche Weise wurde die gegenseitige Kenntnis der Datentypen DARTFACE, DARTSURFACEPART und DARTVOLUME untereinander aufgehoben. Wie bereits in Abschnitt 7.2.4 beschrieben, sind als 3-Zellschlüssel erklärte Darts durch ihre DartID Schlüssel der entsprechenden Hash-Tabellen. Auch durch diesen Ansatz ist es möglich, die bisherige Aussagekraft beizubehalten. Beispielsweise kann man zu einer inneren Kontur die äußere finden.

Die Datentypen konnten durch diese Maßnahmen so verändert werden, dass sie dadurch auch die gewünschte Bedeutung erhielten. So stellen DARTFACES vollständige Flächen dar, begrenzt durch eine oder mehrere Konturen. Selbiges gilt für die DARTVOLUMES. Sie speichern Schalen in Form von Zeigern auf Darts und deren Vernähtheit mit anderen Darts. So ist beispielsweise eine Flächenkontur gegeben durch alle β_0- und β_1-

erreichbaren Darts ausgehend vom gespeicherten Dart. Für die anderen Strukturen gelten ähnliche Traversierungs-Vorschriften (siehe 7.5.1).

Es wird nun kein Wissen mehr über zugehörige Elemente der XGMAP3D gespeichert, da sich dieses durch Traversieren der GMAP3D und durch Zugriff über die Hash-Tabellen ebenfalls errechnen lässt.

Diese neue Datenstruktur ermöglicht es, durch ihre bessere Übersichtlichkeit, auch den Erstellungsvorgang der XGMAP3D aus der GMAP3D zu optimieren. Dieser effizientere Erstellungsvorgang wird im folgenden Abschnitt beschrieben.

7.4.2 Optimierungen bei der XGMAP3D-Erzeugung

Beim Test der Datenstruktur mit großen Volumendatensätzen, für die viele Darts und entsprechend viele DARTFACES erstellt wurden, stellte sich heraus, dass die Suche nach inneren Flächenkonturen viel Zeit beanspruchte. Ein Grund dafür war die unsortierte Liste von Flächenkonturen, die dazu führte, dass bei der Suche jeweils für jedes Element jedes andere betrachtet werden musste.

Nach einer Sortierung nach der Regionsnummer brauchte man nur noch die Elemente der Liste im Bereich dieser Nummer zu betrachten. Durch den Erstellungsvorgang der GMAP3D und dem späteren Finden von Flächenkonturen ergibt sich, dass innere Konturen immer nach äußeren Konturen in der Liste folgen. Somit müssen bei der Suche nach inneren Konturen nur noch folgende Konturen beachtet werden, die die gleiche Regionsnummer wie die gerade betrachtete aufweisen.

Durch diese Maßnahmen ließ sich der Aufwand an Vergleichen zwischen Konturen deutlich reduzieren. Trotzdem war der Aufwand pro Vergleich noch recht hoch, so dass noch weiteres Optimierungspotential bestand.

Wie bereits in Abschnitt 7.3.1 beschrieben, besteht jeder Vergleich aus vier verschiedenen Schritten. Zur Optimierung musste herausgefunden werden, welche Reihenfolge dieser Schritte im Mittel die optimale ist. Eine optimale Lösung zu finden ist schwierig, da Volumen mit unterschiedlichen Charakteristika ungleiche Anforderungen bezüglich der Reihenfolge stellen. Da die hier erstellte Anwendung aber nicht auf ein Gebiet spezialisiert ist, wurde der Weg wie oben beschrieben gewählt (vgl. 7.3.1).

Die anfängliche Reihenfolge der Schritte sah wie folgt aus:

1. Test auf gleiche Regionsnummer
2. Test auf gleiche Regionsnummer der über das β_3 -Orbit erreichbaren Darts
3. Koplanaritätstest
4. „Polygon in Polygon“-Test

Der Test auf die gleiche Regionsnummer ist mit dem wenigsten Aufwand verbunden. Ebenso ermöglicht er, bei der sortierten Liste von Konturen, das Herausspringen aus der Schleife der Flächenkonturvergleiche. Er wurde deshalb an erster Stelle belassen.

Der bisherige zweite Test stellte sich als sehr aufwendig heraus. Er wird deshalb nun als letzter Schritt ausgeführt. Zunächst war er sogar noch aufwendiger, da zur Bestimmung

der Regionsnummer auf das Label-Volumen zugegriffen wurde. Da durch das Wechseln des Darts über das β_3 -Orbit wieder ein Dart einer DARTFACE erreicht wird, welche die Regionsnummer speichert, kann diese auch auf diesem Wege erlangt werden. So ist kein langsamer Zugriff auf das Label-Volumen notwendig, und es werden nur die erforderlichen Strukturen der XGMAP3D benutzt, sodass ein effizienter Speicherzugriff erfolgt.

Der Koplanaritätstest konnte ebenfalls vereinfacht werden, indem vorhandenes Wissen, ausgenutzt wurde. Die übliche Art die Ebene zu erlangen, auf der eine Fläche liegt, ist es, drei Punkte auf dieser Fläche zu bestimmen. Diese drei Punkte sind durch drei unterschiedliche Einbettungen dreier Darts einer DARTFACE gegeben. Diese verhältnismäßig aufwendige Berechnung ist nicht notwendig, da die Ebene ebenfalls aus der DartID eines Darts berechenbar ist (vgl. 7.3.1).

Der „Polygon in Polygon“-Test prüfte anfangs ganze Polygone auf Enthaltensein in einem anderen Polygon. Dafür wurde jeder Eckpunkt des einen Polygons, gegeben durch die Einbettungen der Darts, daraufhin untersucht, ob er im anderen Polygon liegt. Der Test aller Punkte ist nicht notwendig, da es keine Überschneidungen von Darts in der Datenstruktur gibt. Daher reicht es aus, nur einen Eckpunkt der Kontur zu überprüfen.

Der Vorgang der Suche nach inneren Konturen konnte durch alle diese Veränderungen deutlich beschleunigt werden.

Durch die im vorherigen Abschnitt beschriebenen Änderungen der Datenstrukturen ließen sich die weiteren Schritte der Erstellung der XGMAP3D vereinfachen und beschleunigen. Da sich die ursprünglichen Erstellungsansätze zu stark von den am Ende bestehenden unterscheiden, wird auf einen Vergleich beider an dieser Stelle verzichtet.

7.5 Operationen auf der XGMAP3D

Mit Hilfe der Datenstrukturen der XGMAP3D sind im Vergleich zur GMAP3D mehr Operationen möglich. Außerdem werden für die Strukturen einer XGMAP3D Traverser benötigt, die das Ablaufen derer Darts einfach anwendbar machen. Diese Traverser über die XGMAP3D-Strukturen DARTFACE, DARTSURFACEPART und DARTVOLUME (siehe Kapitel 7.2) werden daher am Anfang dieses Kapitels beschrieben.

Aufbauend auf die Informationen der XGMAP3D und die Traverser über ihre Strukturen können weitere Operationen effizient ausgeführt werden. Zum einen bietet es sich an, die Euler-Charakteristik von Schalen zu berechnen, um topologische Eigenschaften dieser abzulesen. Zum anderen werden Möglichkeiten des Löschens von Strukturen der XGMAP3D und das damit zusammenhängende Löschen von Darts gezeigt.

7.5.1 Ablaufen der Darts der XGMAP3D-Strukturen

Um die Darts, die zu den bereits oben beschriebenen Strukturen der XGMAP3D gehören, möglichst schnell und einfach ablaufen und gegebenenfalls markieren zu können, stellt die Klasse XGMAP3DTRAVERSER einige Funktionen bereit. In diesem Abschnitt soll deren Funktionsweise beschrieben werden, während Kapitel 8 die Anwendung und Anwendbarkeit dieser Mechanismen aufzeigt.

Die Klasse XGMAP3DTRAVERSER stellt Traverser für alle drei Datenstrukturen in einer XGMAP3D zur Verfügung. Somit gibt es folgende Funktionen:

1. `traverseDartFace`,
2. `traverseDartSurfacePart` und
3. `traverseDartVolume`.

Diese Funktionen laufen über Darts, die auf unterschiedlich starke Weise zusammengehören. Besuchte Darts werden in allen Fällen markiert. Dabei ist es ebenfalls möglich, äußere und innere Begrenzungen durch verschiedene Markierungen zu unterscheiden.

7.5.1.1 Traversieren über eine DARTFACE

Das Traversieren über die Darts eine DARTFACE kann von der Funktion `traverseDartFace` übernommen werden. Diese Funktion nutzt die Informationen der XGMAP3D aus, so dass der Vorgang effizient und korrekt ausgeführt werden kann. Des Weiteren müssen Informationen über eventuell vorhandene innere oder zugehörige äußere Konturen so nicht noch einmal gewonnen werden. Es sei daran erinnert, dass es keine Verbindung durch Darts zwischen solchen Konturen gibt.

Der Funktion `traverseDartFace` werden drei Parameter übergeben. Wichtig ist die DARTFACE, deren Darts abgelaufen werden sollen. Aus diesem Grund wird ein Zeiger auf diese übergeben. Die weiteren beiden Parameter stellen Label dar, mit denen die Darts

der DARTFACE markiert werden. Das erste Label wird für die äußere Kontur verwendet, das zweite für die inneren Konturen. Das Verfahren funktioniert nach der in Algorithmus 7.5.1 beschriebenen Vorgehensweise.

Algorithmus 7.5.1: Traversieren der Darts einer DARTFACE

```
function  traverseDartFace

   input : dartFace, label_1, label_2

   traverse all beta-0 and beta-1 reachable Darts from anchor of DartFace and
   mark them with label_1

   for all inner contours of dartFace
      traverse all beta-0 and beta-1 reachable Darts from anchor of current
      inner contour and mark them with label_2
```

Beim Traversieren der β_0 - und β_1 -erreichbaren Darts wird der DART3DTRAVERSER herangezogen, der bereits in Abschnitt 6.3.1 beschrieben wurde. Dieser Traverser übernimmt auch die Markierung der abzulaufenden Darts.

In der Funktion `traverseDartFace` wird deshalb zuerst ein DART3DTRAVERSER erstellt, der über Flächenkonturen läuft. Diesem wird der Dart-Repräsentant der DARTFACE mitgeteilt, so dass zunächst die äußere Kontur mit dem ersten Label markiert wird. Im Anschluss daran wird geprüft, ob die betrachtete Fläche innere Konturen besitzt, die bei Vorhandensein mit dem zweiten Label markiert werden. Zu diesem Zweck wird für jede innere Kontur der DART3DTRAVERSER mit dem entsprechenden Anker erstellt. Dabei wird die Liste der Darts ausgenutzt, die die Anker aller inneren Konturen speichert.

7.5.1.2 Traversieren über einen DARTSURFACEPART

Das Traversieren über die Darts eines DARTSURFACEPARTS kann mit Hilfe der Funktion `traverseDartSurfacePart` ebenso effizient und korrekt ausgeführt werden, wie das Traversieren der Darts einer DARTFACE. Dabei ist anzumerken, dass diese Methode wiederum nur über Darts von DARTFACES traversiert. Allerdings gehören in der Regel zu einem DARTSURFACEPART mehrere DARTFACES, die alle zu beachten sind.

Entsprechend einfach gestaltet sich dieser Vorgang, wie in Algorithmus 7.5.2 zu sehen ist.

Algorithmus 7.5.2: Traversieren der Darts eines DARTSURFACEPARTS

```
function  traverseDartSurfacePart

   input : dartSurfacePart, label

   for all dartFaces in dartSurfacePart
      traverseDartFace
```

In einem DARTSURFACEPART sind alle Anker von Flächen, die zu diesem gehören, in einer Liste gespeichert (siehe hierzu 7.2.2). Für alle Elemente dieser Liste muss in der Funktion `traverseDartSurfacePart` nur einmal die zugehörige DARTFACE in der Hash-Tabelle nachgeschlagen werden, so dass für jede DARTFACE die Funktion `traverseDartFace`

aufgerufen werden kann. Da alle Darts eines DARTSURFACEPARTS mit dem gleichen Label markiert werden, wird beim Funktionsaufruf zweimal das gleiche Label übergeben.

7.5.1.2 Traversieren über ein DARTVOLUME

Da ein DARTVOLUME nur aus DARTSURFACEPARTS besteht, welche wiederum nur aus DARTFACES bestehen, kann das Problem des Traversierens über ein DARTVOLUME ebenfalls auf das Traversieren über Darts der DARTFACES zurückgeführt werden. In Algorithmus 7.5.3 ist dieser Ansatz dargestellt.

Algorithmus 7.5.3: Traversieren der Darts eines DARTVOLUMES

```
function traverseDartVolume

   input : dartVolume, label_1, label_2

   for all dartSurfaceParts of outer contour of dartVolume
      traverseDartSurfacePart

   for all inner contours of dartVolume
      for all dartSurfaceParts of current inner contour
         traverseDartSurfacePart
```

Die Funktion `traverseDartVolume` stellt nun die obige Funktionalität zur Verfügung. Ihr werden drei Parameter übergeben, zum einen ein Zeiger auf das DARTVOLUME, dessen Darts traversiert werden sollen, zum anderen die Label für die unterschiedliche Markierung von inneren und äußeren Schalen.

Zunächst werden die Darts der äußeren Schale des DARTVOLUMES markiert. Aus diesem Grund werden alle DARTSURFACEPARTS, die zu dieser Kontur gehören, mit Hilfe des Ankers dieses DARTVOLUMES in der `surfaceHashMap` nachgeschlagen. Diesen Vorgang übernimmt die Hilfsfunktion `getAllDartSurfacePartsFromVolumecontour`. Bei der Erstellung des DARTVOLUMES wurde diese Hash-Tabelle angelegt (siehe 7.3.3). Anschließend werden alle diese DARTSURFACEPARTS mit Hilfe der entsprechenden Funktion traversiert. Es wird immer das erste Label übergeben, da alle Teile zur äußeren Schale gehören.

Nachdem der Vorgang für die äußere Schale abgeschlossen ist, wird ermittelt, ob das DARTVOLUME innere Konturen besitzt. Sollte dies der Fall sein, so ist im DARTVOLUME die Liste der Darts, die diese repräsentieren, nicht leer. Es wird über diese Liste iteriert und für jeden Dart die Funktion `getAllDartSurfacePartsFromVolumecontour` aufgerufen. Die Darts der erhaltenen DARTSURFACEPARTS werden mit dem oben genannten Verfahren mit dem zweiten Label markiert.

Nachdem dies ausgeführt wurde, sind alle Darts, die zum DARTVOLUME gehören, markiert.

7.5.2 Berechnung der Euler-Charakteristik

Ebenfalls in der Klasse XGMAP3DTRAVERSER findet sich eine Funktion, die es ermöglicht, die Euler-Charakteristik einer beliebigen Schale zu bestimmen. Zur Bestimmung der Euler-Charakteristik sind drei Werte notwendig (vgl. Abschnitt 3.4.6), die in der Funktion `traverseVolumecontourReturnVolumecontourInfo` bestimmt werden können. Der Funktion

werden zwei Parameter übergeben. Zum einen wird ein Zeiger auf einen Dart benötigt, durch den bestimmt wird, um welche Schale (Oberfläche) es sich handelt. Zum anderen wird ein Zeiger auf ein Array von ganzen Zahlen der Länge drei übergeben. In diesem Array werden die berechneten Werte für

- die Anzahl der Knoten,
- die Anzahl der Kanten und
- die Anzahl der Flächen

der übergebenen Schale gespeichert.

Um diese Werte bestimmen zu können, ist ein Traversieren der Schale notwendig. In Algorithmus 7.5.4 ist der Ablauf des Verfahrens beschrieben.

Algorithmus 7.5.4: Traversieren einer Volumenoberfläche zur Bestimmung der Euler-Charakteristik

```
function traverseVolumecontourReturnVolumecontourInfo

   input : dart
   output: faces, edges, vertices

   for all dartSurfaceParts of this darts volumecontour
      increment faces for each dartFace of current dartSurfacePart
      for all faces of current dartSurfacePart
         increment edges for each face inclusion of current face

         traverse all darts of current dartFace
            if current dart is not marked as edge
               increment edges
               mark all darts of same edge
            if current dart is not marked as vertex
               increment vertices
               mark all darts at same vertex
```

Auch für diese Aufgabe reicht es aus, die einzelnen DartFaces einer Kontur eines DartVolumes zu betrachten. Der Aufwand dieser Funktion bleibt dabei recht gering, da die Struktur einer XGMap3d ausgenutzt wird. Als Voraussetzung wird angenommen, dass der Dart, auf den der übergebene Zeiger verweist, ein Anker für eine Schale ist. Mit Hilfe eines solchen Darts kann man direkt auf der XGMap3d alle zugehörigen DartSurfaceParts zu dieser Kontur bekommen. Diese DartSurfaceParts wiederum speichern Zeiger auf Anker-Darts, die die zugehörigen DartFaces repräsentieren.

Addiert man die Anzahl der Zeiger auf Anker-Darts, die DartFaces repräsentieren, für alle DartSurfaceParts, so erhält man die gesamte Anzahl von Flächen der Schale. Es bleibt somit nur noch die Aufgabe, die Knoten und Kanten zu zählen. Dieser Vorgang kann einfach durch ein einmaliges Ablaufen der Darts aller DartFaces geschehen. Besuchte Darts werden markiert, damit sie nicht mehrfach in den Zählvorgang einfließen.

Um die Kanten zu zählen, wird für jeden gefundenen, nicht markierten Dart der Zähler der Kanten inkrementiert. Zudem müssen alle Darts, die die betrachtete Kante bilden, markiert werden. Aus diesem Grund werden alle Darts des $\beta_0 \circ \beta_2 \circ \beta_3$ -Orbits markiert.

Die Knoten werden auf die gleiche Weise gezählt. Nur werden in diesem Fall alle Darts, die an dem selben Knoten enden (das $\beta_1 \circ \beta_2 \circ \beta_3$-Orbit) mit einer anderen Markierung versehen.

Nach diesen Schritten sind alle gewünschten Knoten, Kanten und Flächen erfasst, so dass sich die Euler-Charakteristik berechnen lässt. Die Berechnung selbst findet in der aufrufenden Klasse GLXGMAP3DVIEWER in der Funktion `showDartVolumeInfo` statt. Somit können die gezählten Werte ebenfalls als Informationen ausgegeben werden (siehe Kapitel 9.3).

7.5.3 Löschen von Strukturen einer XGMAP3D

Die Strukturen der XGMAP3D grenzen verschiedene Regionen gegeneinander ab. Allerdings gibt es Fälle, in denen es nicht erwünscht ist, dass zwei Regionen voneinander abgegrenzt sind. Diese Regionen, die beispielsweise durch eine Übersegmentierung entstehen können, sollten dann zu einer einzigen zusammengefasst werden. Aus diesem Grund kann es notwendig sein, die trennenden Strukturen zu löschen.

Für die Durchführung solcher Operationen stellt die Klasse XGMAP3DMANIPULATOR einige Funktionen zur Verfügung. In dieser Klasse wird zudem die Konsistenz der GMAP3D beachtet, die hinter der XGMAP3D steht. Das bedeutet, dass nach dem Löschen der XGMAP3D-Strukturen und der zugehörigen Darts, die GMAP3D weiterhin eine korrekte Border-Map darstellt.

Auf der untersten Ebene sind Regionen immer durch DARTFACES voneinander getrennt. Sollen also zwei Regionen miteinander verschmolzen werden, so sind die trennenden DARTFACES zu entfernen. Hierbei sind zwei Dinge zu beachten. Zum einen müssen die Darts, die über β-Orbits mit Darts der trennenden DARTFACES vernäht sind, umgenäht werden. Erst im Anschluss daran dürfen die Darts der DARTFACES gelöscht werden. Zum anderen müssen die Strukturen der XGMAP3D auf die neu entstandenen Flächen angepasst werden.

Dieser Fall ist sehr komplex und zieht eine große Zahl von Ausnahmefällen nach sich. Aus diesem Grund wird zunächst ein einfacherer Fall betrachtet, bei dem alle Darts der zu löschenden DARTFACES aus der Dartliste gelöscht werden dürfen. Dieser Fall tritt dann auf, wenn man Volumeneinschlüsse löscht.

7.5.3.1 Löschen von Volumeneinschlüssen

Beim vollständigen Löschen von Volumeneinschlüssen werden alle über β-Orbits verbundene Darts einer Zusammenhangskomponente gelöscht. Diese Darts hingen nur über die Enthalten-Relation mit den übrigen Darts der GMAP3D zusammen. Diese Relation ist in der XGMAP3D gespeichert. Die Darts von Volumeneinschlüssen können somit problemlos aus der GMAP3D gelöscht werden, ohne dass in dieser Inkonsistenzen auftreten.

In der XGMAP3D müssen natürlich die entsprechenden Strukturen (vgl. 7.2) angepasst werden. Auch müssen die Strukturen, welche den Einschluss bilden, gelöscht werden. In

Algorithmus 7.5.5 ist die Vorgehensweise zum Löschen von Einschlüssen beschrieben. Die entsprechende Funktion `deleteVolumeInclusion` findet sich in der Klasse XGMAP3DMANIPULATOR.

Algorithmus 7.5.5: Löschen von Volumeneinschlüssen

```
function deleteVolumeInclusion

   input : dart //pointer of the startDart of volume inclusion

   find all XGMap structures reachable from dart which belong to the given
   isolated inclusion, add them to delete_list and mark them with label using
   findInnerVolumes(dart, delete_list, label)
   delete dart from the list of inner volume contours of parent volume
   delete the complete surface of volume inclusion
   delete all found DartVolumes inside      // stored in delete_list
   relable all necessary voxels in region image
   delete all marked darts


function findInnerVolumes

   input : dart, delete_list, label

   first get all dartSurfaceParts of volume contour of dart
   mark dart with label
   for all dartSurfaceParts
      push startDart on a stack

   while darts inside stack
      for all beta-orbits
         store beta dart of stack front dart on stack and mark it
         if stored dart is a cellKey-3 of an outer volume contour
            push dart on delete_list
            push all start darts of volume inclusions on stack
            for all start darts of volume inclusions
               findInnerVolumes(dart, delete_list, label)
      delete front dart of stack
```

Wie im obigen Algorithmus zu sehen ist, sind selbst beim Löschen von Volumeneinschlüssen viele Dinge zu beachten und auszuführen, obwohl die entsprechenden Darts in der GMAP3D einfach gelöscht werden dürfen. Da diese Darts, beziehungsweise deren DartID, aber als Schlüssel der Hash-Tabellen der XGMAP3D dienen, und diese wiederum auf Strukturen der XGMAP3D verweisen, müssen diese Einträge alle herausgefunden und gelöscht werden, bevor die Darts gelöscht werden können.

Das Finden dieser Einträge übernehmen dabei zahlreiche Hilfsfunktionen. Mit `findInnerVolumes` ist eine dieser angegeben. Für diese Funktion gibt es eine rekursive und eine nicht rekursive Fassung. Da die rekursive Version verständlicher (allerdings auch langsamer) ist, ist diese beschrieben. Mit dieser Funktion lassen sich alle DARTVOLUMES herausfinden, selbst wenn diese ebenfalls innere Regionen von inneren Regionen darstellen.

Ausgehend von den gefundenen Anker-Darts, die die gefundenen DARTVOLUMES repräsentieren, müssen alle anderen XGMAP3D-Strukturen dieser (DARTSURFACEPARTS, DARTFACES) gefunden und aus den Hash-Tabellen entfernt werden.

Des Weiteren müssen die Voxel des Volumens der ikonischen Regionenrepräsentation umgelabelt werden. Alle Voxel der gelöschten Regionen müssen nun das Label bekommen, das die Voxel der umgebenden Region besitzen. Auch wird die Durchschnittsfarbe des übrig bleibenden DARTVOLUMES an dieser Stelle neu bestimmt, da sich diese durch Hinzunahme der Voxel der anderen Regionen verändert haben kann.

Nun können alle markierten Darts gelöscht werden, und man erhält eine neue XGMAP3D, die ebenfalls konsistent ist.

7.5.3.2 Löschen von beliebigen Regionengrenzen

Das Löschen von Regionengrenzen bewirkt, dass zwei Regionen miteinander verschmolzen werden. Dieser Vorgang kann so ausgeführt werden, dass durch Angabe einer DARTFACE, die zwei Regionen voneinander trennt, diese zwei Regionen zur Verschmelzung vorbereitet werden.

Der erste Schritt besteht darin, alle Flächen zu finden, die beide Regionen voneinander abgrenzen. Diese müssen im Zuge des Verschmelzungsvorgangs wegfallen.

Anschließend sind noch zwei Schritte nötig, damit nach Beendigung wieder eine konsistente XGMAP3D vorliegt. Zuerst müssen alle Darts, die mit den Darts der zu löschenden DARTFACES vernäht sind, mit anderen Darts vernäht werden.

Dabei ist zu beachten, dass, wenn weiterhin eine Border-Map vorliegen soll, noch weitere Darts zu löschen sind. Danach müssen noch die XGMAP3D-Strukturen angepasst werden. Hierbei ist insbesondere zu beachten, dass die Enthalten-Relationen für alle Strukturen korrekt angepasst werden. Durch dass Verschmelzen können neue Elemente in den Relationen hinzukommen, aber eventuell auch welche wegfallen.

Soll am Ende der Verschmelzung weiterhin eine Border-Map vorliegen, reicht die Operation aus Abschnitt 6.3.2 nicht aus. Dadurch entstandene adjazente koplanare Flächen vom Grad eins oder zwei und adjazente kollineare Kanten, die von einem Knoten des Grades zwei getrennt sind, müssen verschmolzen werden.

In erster Instanz sind bei der Verschmelzung zwei Ausnahmefälle zu beachten. Besitzt die zu verschmelzende Fläche einen Einschluss, so können nach der Verschmelzung folgende Konfigurationen auftreten:

1. Eine oder mehrere „Höhlen“ werden zu einem Volumeneinschluss.
2. Aus einem „Torus“ wird eine „Höhle“.

Der zweite Ausnahmefall entsteht bei der Verschmelzung koplanarer Flächen. Hierbei können innere Flächenkonturen entstehen. Diese neuen Flächenkonturen führen eventuell zu neuen DARTSURFACEPARTS.

Da diese komplexen Ausnahmen alle behandelt werden müssen, ist ein anderer Ansatz sinnvoll, wenn man viele Flächen verschmelzen möchte. Dieser Ansatz wird im folgenden Abschnitt beschrieben.

7.5.3.3 Verschmelzen von mehreren Regionen

Im vorigen Abschnitt wurden die Schwierigkeiten der Verschmelzungsoperation zweier Regionen diskutiert. Dennoch kann ein schnelles Verschmelzen von vielen Regionen in einem Schritt ausgeführt werden. Dazu muss zunächst geklärt werden, welche Regionen für die Verschmelzung vorgesehen sind. Gruppen von benachbarten Regionen können dann jeweils zu einer Region zusammengefasst werden. Eine Auswahl der zu verschmelzenden Regionen kann über die jeweiligen trennenden Flächen erfolgen.

Wenn viele Flächen zum Löschen vorgesehen sind, bedeutet dies, dass viele Regionen mit anderen Regionen zusammengefasst werden. Ein solcher Schritt hat zur Folge, dass viele Strukturen der XGMAP3D verändert beziehungsweise gelöscht werden müssen. Auch zahlreiche Darts der GMAP3D müssen gelöscht werden, viele andere müssen aus diesem Grund neu vernäht werden. Eine effiziente Möglichkeit der Umsetzung eines solchen Verschmelzungsvorgangs besteht darin, dass alle Strukturen der XGMAP3D und auch die Darts der GMAP3D verworfen und anschließend neu erstellt werden.

Vor dem Erstellen wird die ikonische Regionenrepräsentation dahingehend angepasst, dass Regionen, die durch die zu löschenden DARTFACES getrennt sind, verschmolzen werden. Zu diesem Zweck werden zunächst jeweils alle Regionen gesucht, die nach der Verschmelzung eine Region bilden sollen. Jede Gruppe von gefundenen Regionen bekommt das Label einer enthaltenden Region zugewiesen. Durch diesen Schritt zeigt die ikonische Regionenrepräsentation bereits die neuen Regionen, die durch die Verschmelzung entstanden sind.

Aus dieser neuen ikonischen Regionenrepräsentation kann dann eine XGMAP3D mit ihrer GMAP3D erzeugt werden. Im Anschluss daran liegt eine korrekte Border-Map vor, die die neuen Regionen repräsentiert.

In der Anwendung hat sich gezeigt, dass das gleichzeitige Verschmelzen vieler Regionen häufig benötigt wird (siehe hierzu Abschnitt 9.2.3). Die hier vorgestellte Methode löst diese Aufgabe in den meisten Fällen sehr effizient.

8 Grafische Darstellung einer 3-XG-Map

In diesem Kapitel werden die erarbeiteten Möglichkeiten zur Anzeige einer 3-XG-Map aufgezeigt. Bei der automatisierten Arbeitsweise des Verfahrens ist es nahezu unmöglich, nur aufgrund der Datenstrukturen die bisher vorgestellt wurden, eine Aussage darüber zu treffen, ob beispielsweise die Entwicklungsarbeit am Erstellungsvorgang der GMAP3D oder XGMAP3D erfolgreich war.

Schon zu Beginn dieser Arbeit wurde es daher notwendig, eine grafische Ansicht der erzeugten Datenstrukturen zu erhalten. So war es leicht möglich, wichtige Eigenschaften der zugrunde liegenden Datenstrukturen in Bezug auf ihre Korrektheit zu überprüfen.

Im Rahmen dieser Visualisierung wurden zwei Ansichten der XGMAP3D implementiert. Eine zeigt die Darts gerendert an, die andere stellt die Struktur der XGMAP3D in Form einer baumartigen Ansicht dar. Damit beide Darstellungsarten parallel verwendet werden können, werden sie miteinander synchronisiert.

Weiterhin wird der Funktionsumfang der Anwendung knapp erörtert. Dies soll den Benutzern den Einstieg in die Anwendung erleichtern. Außerdem werden einige Statusmeldungen erläutert, die das Programm während des Ablaufes ausgibt.

Dieses Kapitel grenzt sich gegenüber dem folgenden Kapitel ab, in dem es die grundlegenden Eigenschaften der Anwendung beschreibt, wohingegen das folgende Kapitel konkrete Anwendungen an Beispielen aufzeigt.

8.1 Grafische Benutzungsoberfläche

Die vorliegende grafische Benutzungsoberfläche stellt das Endergebnis eines andauernden Entwicklungsprozesses dar. Angefangen wurde mit viel einfacheren Mitteln. Die erste grafische Ausgabe einer Volume-Map (vgl. 6.2.1) gestaltete sich noch deutlich vereinfacht, diente sie doch lediglich dazu, beurteilen zu können, ob wirklich alle Dartplatzhalter korrekt entfernt wurden. Dazu wurden Koordinaten der Dartplatzhalter ermittelt und als Zylinder in eine POV-Ray Szenenbeschreibung exportiert. Diese wurde anschließend gerendert, um eine übersichtliche Darstellung zu erhalten. Diese Ansicht vereinfachte die Überprüfung der Korrektheit erheblich, zumal die Alternative darin bestand, aus der binären Repräsentation der Voxel der Volume-Map alle Bits zu überprüfen. Ein Beispiel für diese Anfänge zeigt die folgende Abbildung 74, welche die Volume-Map für das in Abschnitt 3.4.2 vorgestellte Volumen darstellt.

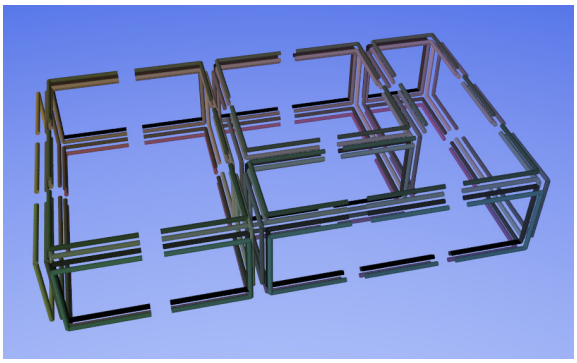


Abbildung 74: Mit POV-Ray gerenderte Ansicht der Dartplatzhalter der Volume-Map zu dem in Abschnitt 3.4.2 vorgestellten Volumen

Doch schon bei der Erstellung der GMAP3D aus der Volume-Map wurden die Datenstrukturen zu komplex um sie nur anzuzeigen. Es musste eine interaktive Benutzungsoberfläche entwickelt werden, welche folgenden Anforderungen genügen sollte:

- Laden von Volumen in unterschiedlichen Repräsentationen
- Erstellung und Anzeige der daraus berechneten Datenstruktur (zum Beispiel: GMAP3D, XGMAP3D)
- Ablaufen dieser Datenstruktur (zum Beispiel durch Benutzung der β_i-Vernähtheit)
- Dreidimensionale Navigation innerhalb der Datenstruktur
- Einfache Erweiterbarkeit für zukünftige Anwendungen auf der Datenstruktur
- Weitgehende Unabhängigkeit vom Betriebssystem

Diese Anforderungen führten dazu, dass die im Rahmen dieser Arbeit entwickelte grafische Benutzungsoberfläche (auch kurz: GUI) mit Qt programmiert wurde, welches eine gute Plattformunabhängigkeit bietet. Des Weiteren verfügt Qt über eine OpenGL-Schnittstelle, welche es erlaubt, dreidimensionale Anzeigeelemente zu entwickeln. Das folgende Unterkapitel befasst sich zunächst mit dem Teil der Darstellung, der eine strukturelle Ansicht der 3-XG-Map ermöglicht, bevor darauf folgend die OpenGL-Darstellung der XGMAP3D erläutert wird.

8.1.1 Strukturelle Ansicht der 3-XG-Map

Die Benutzungsoberfläche der Anwendung „OpenGL XGMap3d Viewer“ ist in zwei voneinander getrennte Bereiche aufgegliedert. Beide zeigen die aktuelle Datenstruktur der XGMAP3D auf unterschiedliche Art an. Dieses Kapitel beschäftigt sich mit der baumartigen Ansicht (auch: ListView) der XGMAP3D, welche sich auf der rechten Seite des Hauptfensters befindet.

Das Hauptfenster selbst wird an dieser Stelle nur in seiner Funktionalität der Anzeige der XGMAP3D in einem ListView beschrieben. Es wurde mit dem Qt-Designer erstellt, welcher die Interaktion der einzelnen Programmbausteine recht intuitiv ermöglichte. Daher wird nur an sehr wichtigen Stellen auf diese Interaktion aus einer programmtechnischen Sicht eingegangen werden, wohingegen die daraus resultierende Funktionalität im Kapitel 8.2 beschrieben wird.

Wird ein neues Volumen in die GUI geladen, so wird zunächst die XGMAP3D erstellt. Ist dieser Erstellungsvorgang abgeschlossen, so wird ein Signal, das `dartVolumeSignal`, an das Hauptfenster gesendet, welches daraufhin veranlasst, dass der ListView neu erstellt wird und im Anschluss daran die erfolgreiche Erstellung zurück meldet.

Damit ein effizienter Zugriff vom ListView auf die Darts der darunterliegenden Datenstrukturen möglich ist, wurde ein neuer Datentyp QCHECKLISTDARTITEM erstellt, welcher alle Eigenschaften des allgemeinen „auswählbaren ListView Eintrag“ (in Qt auch QCHECKLISTVIEWITEM genannt) erbt, aber zusätzlich noch einen Zeiger auf einen Dart bereitstellt. Dieser Zeiger wird bei der Erstellung auf einen vorhandenen Dart gesetzt und macht somit einen schnellen Zugriff auf die Datenstrukturen möglich.

Außerdem besitzt der Datentyp eine überladene Methode, welche auf Veränderungen in der Markierung des Eintrags reagiert. Dies ist für die Anzeige besonders wichtig, da markierte inaktive Elemente ebenfalls grafisch (siehe 8.1.2) angezeigt werden sollen. Zu diesem Zweck werden bei einer Markierung eines Elementes die jeweiligen zur Struktur gehörenden Darts traversiert und mit einem bestimmten Label markiert. Wird eine Markierung aufgehoben, so geschieht das gleiche Traversieren, jedoch werden nun alle Darts demarkiert.

Auf eine Beschreibung der kompletten Schnittstelle wird an dieser Stelle allerdings verzichtet, da ein Großteil der Schnittstelle nur geerbt wird. Stattdessen werden in der folgenden Tabelle nur die Methoden der Schnittstelle erläutert, die neu hinzugekommen sind:

Nicht geerbte Schnittstelle der Klasse QCHECKLISTDARTITEM

`Dart3d* getStartDartPointer()`
: gibt den Zeiger auf den DART zurück, mit dem der Listeneintrag assoziiert ist.

`setDartPointer(Dart3d*)`
: ordnet dem Listeneintrag den übergebenen Zeiger auf einen DART zu.

`stateChange (bool)`
: reagiert auf eine Veränderung der Markierung des Listeneintrags.

Außerdem wurden die Konstruktoren der Klasse angepasst, sodass die Standard-Konstruktoren Exemplare dieser Klasse mit einem Null-Zeiger initialisieren. Zudem gibt es weitere Konstruktoren, welche einen Zeiger auf einen DART übergeben bekommen und diesen während der Erstellung mit dem QCHECKLISTDARTITEM assoziieren.

Nachdem der obige Datentyp definiert wurde, kann nun beschrieben werden, wie das ListView-Anzeigeelement mit Daten der XGMAP3D gefüllt wird. Diese Funktion gehört zum Hauptfenster und läuft nach folgendem Algorithmus ab:

Algorithmus 8.1.1: Erzeugung des ListViews einer XGMAP

```
function fillListView

   input : xGMap

   for all DartVolumes of the XGMap
      create a QListViewDartItem "Volume #volume_label"
      and append it to the end of the ListView
```

Wie aus dem obigen Algorithmus sichtbar wird, wurde für jedes Element vom Typ DARTVOLUME der XGMAP3D ein Listeneintrag erstellt. Dies geschieht aus Gründen der Effizienz, da ein kompletter Aufbau der Liste mit allen Elementen der XGMAP3D sehr viel Speicher benötigen würde. Deshalb werden bei einem Ausklappen eines Volumen-Items der Liste alle beteiligten Strukturen dessen nachgeladen. Dieser Nachladevorgang ist im folgenden Algorithmus 8.1.2 beschrieben.

Algorithmus 8.1.2: Nachladen der untergeordneten Elemente eines Volumen-Elements

```
function expandListViewItem

   input : xGMap, volumeItem

   create a QListViewDartItem "Outer Volume Contour"
   and append it to the volumeItem

   for all DartSurfaceParts of the outer volume contour
      create a QListViewDartItem "Surface Part #surface_id"
      and append it to the outer volume contour

      for all DartFaces of the DartSurfacePart
         create a QListViewDartItem "Face #face_id"
         and append it to the surfacepart item
         for all inner face contours of the dartFace
            create a QListViewDartItem "InnerFace #face_id/#innerF_count"
```

```
            and append it to the face item
            create a QListViewDartItem "Included SurfacePart #surfacepart_id"
            and append it to the inner face contour item

    create the same structure for all inner volume contours
```

Ebenso werden die nachgeladenen Elemente beim Zuklappen eines Volumen-Eintrags wieder gelöscht, um nicht mehr benötigten Speicher wieder freizugeben.

Bei der Erstellung der einzelnen Elemente werden folgende Zeiger auf Darts zu den jeweiligen Einträgen gespeichert:

- Volume, SurfacePart, Face: Die jeweiligen Anker der Datenstruktur (siehe hierzu auch Kapitel 7.2)
- innere Flächenkonturen: Der 2-Zellschlüssel-Dart der entsprechenden Flächenkontur
- innere Volumenkonturen: Der Zeiger auf den Dart, der in der DartVolume-Klasse zu diesem Volumeneinschluss gespeichert ist.

Der so erstellte ListView besitzt aber nicht nur die Fähigkeit die XGMap3D anzuzeigen, mit ihm lässt sich auch auf die aktuelle XGMap3D zugreifen. Klickt man beispielsweise auf ein Element, so wird, sofern dieses nicht bereits ausgewählt war, auf der angezeigten GLXGMap3D (siehe folgenden Abschnitt 8.1.2) der aktuelle Dart und damit auch die aktuelle DartFace, das aktuelle DartVolume etc. gewechselt.

Außerdem existiert die Möglichkeit, nicht aktuell ausgewählte ListView-Einträge mit einem Haken zu versehen, damit sie in grau gezeichnet werden, falls sie inaktiv sind. Die grafische Anzeige einer XGMap3D im Hauptfenster behandelt der folgende Abschnitt.

8.1.2 Darstellung der 3-XG-Map mit OpenGL

Im Gegensatz zu der Darstellung der XGMap3D in der Form eines ListViews existiert noch eine weitere Möglichkeit der Visualisierung, nämlich die direkte Anzeige der enthaltenen Darts, welche mittels ihrer Einbettungen in den dreidimensionalen Raum dargestellt werden. Diese Form lässt sich in den Grundzügen bereits in Abbildung 74 erahnen. Allerdings war diese Anzeige auf Volume-Maps beschränkt und erforderte für jede neue Ansicht der Szene einen Render-Vorgang in POV-Ray. Weiterhin war es nicht wünschenswert, das Rendern an ein externes Programm zu übergeben, welches zudem nicht für diesen Einsatzzweck ausgelegt war.

Deshalb entstand im Rahmen dieser Arbeit ein Qt-Widget[63], welches einen Container um eine XGMap3D bildet und zusätzlich OpenGL zur Anzeige derselben benutzt. Dieses Widget wird durch die Klasse GLXGMap3D beschrieben und soll in diesem Kapitel näher

63 Ein Qt-Widget bezeichnet ein Qt-Steuerelement, welches eine bestimmte Aufgabe erfüllt. So gibt es zum Beispiel ein Widget zur Texteingabe, zur Zeitanzeige etc.

erläutert werden. Um einen groben Überblick über die Strukturen der Datentypen zu erhalten, werden in der folgenden Abbildung 75 die Zusammenhänge skizziert.

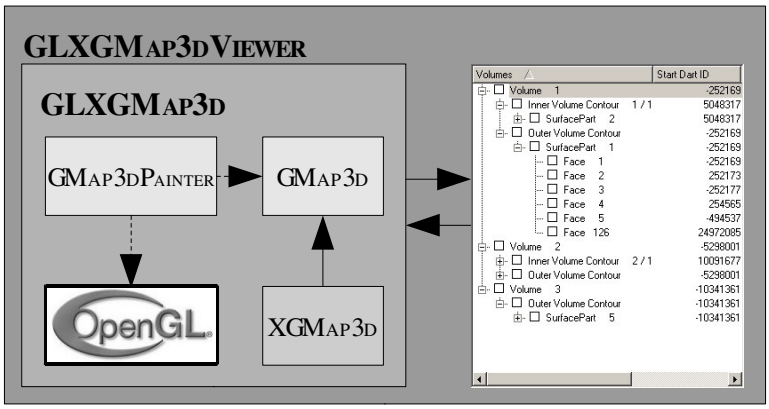


Abbildung 75: Grober Aufbau der implementierten Struktur der grafischen Benutzungsoberfläche

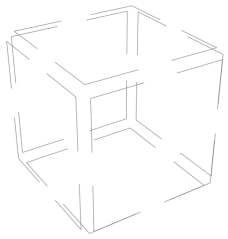
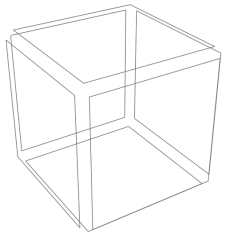
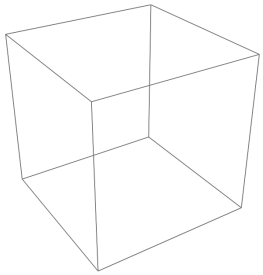
Die Klassen GMAP3D beziehungsweise XGMAP3D wurden bereits in den vorigen Kapiteln beschrieben und stellen lediglich die Grundlage für die Anzeige ihrer dar. Das zentrale Element zum Zeichnen der Darts ist der GMAP3DPAINTER. Er besitzt Zugriff auf die GMAP3D, und damit auf alle Darts, sowie eine OpenGL-Schnittstelle, welche es ihm erlaubt, die Darts im dreidimensionalen Raum zu zeichnen. Da diese Klasse einen so zentralen Platz einnimmt und dementsprechend von der GLXGMAP3D delegiert wird, wird sie zunächst ausführlicher beschrieben.

Der grundlegende Algorithmus zum Zeichnen der XGMAP3D besteht im Ablaufen der kompletten GMAP3D und ist im Folgenden beschrieben. Stößt der Zeichner während des Ablaufens auf einen markierten Dart, so wird für diesen ausgewertet, mit welcher Farbe er gezeichnet werden soll. Die Farbe ergibt sich durch seine Markierung. Dabei gelten folgende Zuordnungen von Markierung zu Farbe:

Markierung	Gezeichnete Farbe
0	rot
1	hellgrün
2	dunkelgrün
3	goldfarben
4	hellblau
5	blau
6	pink
7	weiß

Besitzt ein Dart mehrere Markierungen, so wird die Farbe der kleinsten vorhandenen Markierung zum Zeichnen ausgewählt. Dies ergibt sich aus den Bedeutungen der Farben (siehe hierzu Kapitel 8.2). Je kleiner die Markierung, um so spezieller ist der zugehörige Farbcode zu verstehen.

Wurde die Farbe ermittelt, so werden die Koordinaten des zu zeichnenden Darts berechnet. Diese ergeben sich aus der Einbettung des Darts und, je nach zu zeichnendem Modus, ebenfalls aus dem Dartplatzhalter der Volume-Map (vgl. Definition 6.2.1). In der vorliegenden Anwendung existieren drei Modi für die Anzeige von Darts. Sie heißen „Explicit Darts“, „Implicit Darts“ und „Compact Darts“ und verändern die Anzeige der Darts so, dass sie Start- und Endpunkte der Linie eines gezeichneten Darts beeinflussen. Die folgende Tabelle beschreibt die Auswirkungen der Wahl des Modus' auf die Positionen der zu zeichnenden Darts.

Modus	**Eigenschaften**	**Beispiel**
Explicit Darts	Zeichnet jeden Dart einzeln, das heißt, es ist jeder einzelne Dart sichtbar. Dies wird dadurch erreicht, dass die zu zeichnende Linie des Darts folgende Endpunkte besitzt: $\vec{p}_1 = Emb(dart) + Offset$ $\vec{p}_2 = (Emb(\beta_0(dart)) + Offset + \vec{p}_1)/2$ wobei Emb für die geometrische Einbettung eines Darts steht.	
Implicit Darts	Zeichnet jeden Dart bis zu seinem β_0 Nachbarn (inklusive Offsets). Dies wird dadurch erreicht, dass die zu zeichnende Linie des Darts folgende Endpunkte besitzt: $\vec{p}_1 = Emb(dart) + Offset$ $\vec{p}_2 = Emb(\beta_0(dart)) + Offset$ wobei Emb für die geometrische Einbettung eines Darts steht.	
Compact Darts	Zeichnet jeden Dart bis zu seinem β_0 Nachbarn (ohne Offsets). Dies wird dadurch erreicht, dass die zu zeichnende Linie des Darts folgende Endpunkte besitzt: $\vec{p}_1 = Emb(dart)$ $\vec{p}_2 = Emb(\beta_0(dart))$ wobei Emb für die geometrische Einbettung eines Darts steht.	

Die drei vorgestellten Render-Modi werden direkt durch den GMAP3DPAINTER abgewickelt. Wahlweise bietet er zudem noch die Möglichkeit, den Rahmen des Ausgangsvolumens der GMAP3D zu zeichnen. Damit sind seine Möglichkeiten der Zeichnung aber auch erschöpft.

Dennoch gibt es in der GUI noch eine weitere Möglichkeit der Anzeige von Elementen durch Oberflächenrendering. Da diese Methode der Darstellung sehr viel näher an der Repräsentation des Label-Volumen als an der der XGMAP3D liegt, wurde diese direkt in der Klasse GLXGMAP3D implementiert. Da diese Klasse direkten Zugriff auf die ikonische Repräsentation und auf die XGMAP3D besitzt, bietet sich dies an.

Folgende Möglichkeiten existieren für das Rendern von Oberflächen:

1. Oberflächen werden nicht gerendert.
2. Alle Oberflächen des aktiven DARTVOLUMES werden gerendert.
3. Alle Oberflächen aller inaktiven markierten DARTVOLUMES werden dargestellt.

Zudem kann wahlweise ein Hineinschauen in die gerenderten Oberflächen aktiviert werden. Dies ist besonders dann sinnvoll, wenn eine Region Einschlüsse besitzt, da diese ohne einen Einblick ins Innere nicht sichtbar sind.

Einen Überblick über diese verschiedenen Render-Modi bietet Abbildung 76:

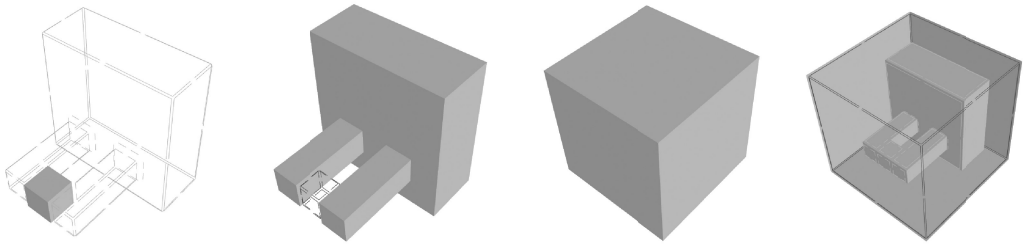


Abbildung 76: Die verschiedenen Möglichkeiten des Oberflächenrenderings. Von links nach rechts: Rendering des aktiven DARTVOLUMES, Rendering aller markierter inaktiver DARTVOLUMES, Rendering ohne Hineinschauen und Rendering mit Hineinschauen.

Die vom GMAP3DPAINTER gezeichneten Darts und die errechneten Oberflächen der Regionen werden allerdings im dreidimensionalen Raum nicht an den Punkten der Einbettungen gespeichert, sondern unterliegen einer Translation zu einem gegebenen GMAP3D-Ursprung $\overrightarrow{mapC}$. Der Zeichenvorgang dieser zentrierten GMAP3D ergibt sich durch folgenden Ablauf:

1. $translate\left(-\overrightarrow{mapC}\right)$
2. *zeichne XGMAP3D*
3. $translate\left(\overrightarrow{mapC}\right)$

Dieser Zeichenvorgang findet in der GLXGMAP3D, wiederum nicht notwendigerweise im Ursprung des Koordinatensystems, statt. Durch die Möglichkeit innerhalb dieses Widgets mittels der Maus eine Translation des Rotationszentrums $\overrightarrow{rotC}$ zu bewirken, wird die Ausgabe der XGMAP3D auf dieses Rotationszentrum zentriert. Die Translation geschieht

dabei innerhalb der Klasse GLXGMAP3D in Abhängigkeit von der Größe des Ursprungsvolumens der GMAP3D.

Um dieses vom Ursprung verschobene Zentrum kann die angezeigte XGMAP3D nun wiederum mittels Maus-Interaktion des Benutzers gedreht werden. Diese Drehung um das Rotationszentrum erfolgt durch eine Umrechnung der zweidimensionalen Differenz zwischen der letzten und der aktuellen Mausposition auf eine Kugeloberfläche. Abbildung 77 veranschaulicht diese Projektion einer zweidimensionalen Mausbewegung auf eine dreidimensionale Kugeldrehung. Durch diese Transformation lässt sich eine dreidimensionale Rotationsmatrix bestimmen, die die momentane Drehung repräsentiert.

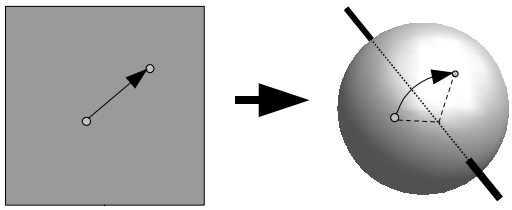


Abbildung 77: Projektion des zweidimensionalen Bewegungsvektors der Maus auf eine Kugeloberläche. Hierdurch entsteht eine Rotationsachse und ein Drehungswinkel um diese Achse.

Des Weiteren gibt es noch die Möglichkeiten zu vergrößern oder zu verkleinern. Dabei entspricht dies einer Bewegung der virtuellen Kamera, durch deren Linse die Szene betrachtet wird, entlang der Z-Achse. Dabei bedeutet kein Zoom, dass die gesamte XGMAP3D sichtbar ist, und der volle Zoomfaktor, dass sich die Kamera im Mittelpunkt der XGMAP3D befindet, vorausgesetzt, dass es keine weiteren Translationen des Benutzers gibt. Die Szenerie der angezeigten XGMAP3D lässt sich anhand der durchgeführten Operationen demnach wie folgt beschreiben:

1. $translate(-\overrightarrow{mapC})$
2. *zeichne XGMAP3D*
3. $translate(\overrightarrow{mapC})$
4. $rotate(\boldsymbol{rotM})$
5. $translate(\overrightarrow{rotC})$
6. *zoome um den angegebenen Faktor*

Neben diesen Möglichkeiten der grafischen Darstellung einer XGMAP3D bietet das GLXGMAP3D-Widget noch die Möglichkeit, einen vollständigen Import von Volumendaten zu realisieren. Dabei gibt es zwei unterschiedliche Methoden, die auf die beiden unterschiedlichen Volumenarten abgestimmt sind, die geladen werden können.

Die erste Funktion, `loadVolume`, lädt zunächst ein angegebenes Volumen von der Festplatte in den Arbeitsspeicher und bildet dessen Gradientenbetragsvolumen (vgl. Kap. 5). Die Standardabweichung σ des dazu verwendeten Gauß-Filters kann hierbei frei

vom Benutzer gewählt werden. Auf diesem Gradientenbetragsvolumen wird dann der in Kapitel 5 beschriebene Union-Find-Wasserscheiden-Algorithmus angewendet. Die daraus erhaltene Regionenrepräsentation in ikonischer Form wird verwendet, um eine XGMAP3D zu erstellen. Zu diesem Zweck wird der XGMAP3DCREATOR dazu delegiert, mittels des Volumens der ikonischen Regionenrepräsentation zunächst eine GMAP3D und darauf aufbauend eine XGMAP3D zu erstellen. Anschließend werden einige Grundeinstellungen des Anzeigefensters gesetzt und der ListView neu aufgebaut.

Neben dieser Funktion gibt es noch eine weitere, `loadLabeledVolume`, welche bereits ein gelabeltes Volumen erwartet, und deshalb das Volumen nur lädt, um daraus direkt die XGMAP3D zu erstellen. Dabei ist zu beachten, dass das gewählte Volumen einige Eigenschaften erfüllen muss. Es muss die selben Form besitzen, wie ein Volumen, welches in einer ikonischen Regionenrepräsentation vorliegt, bei der die Regionsnummern eindeutig sind, und 6-verbundene Voxel Regionen darstellen.

Der komplette Erstellungsvorgang macht den Zusammenhang der einzelnen Themen dieser Arbeit sehr gut sichtbar, wie sich in Abbildung 78 erkennen lässt.

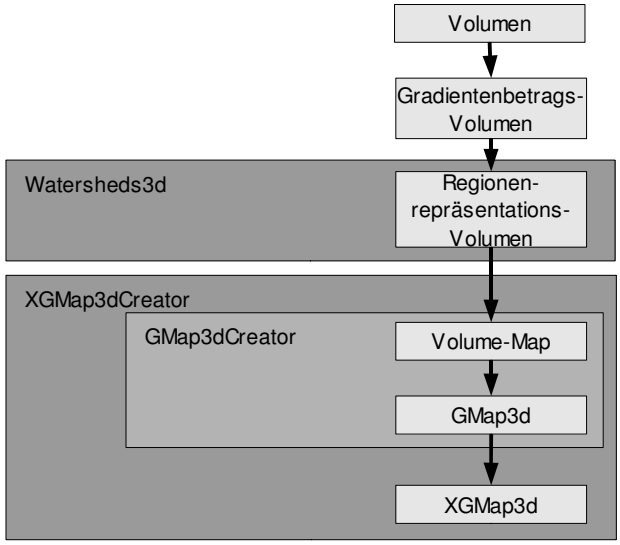


Abbildung 78: Struktureller Ablauf nach dem Laden eines Volumens bis hin zu Erstellung der XGMAP3D

8.1.3 Synchronisation beider Darstellungen der 3-XG-Map

Sowohl die baumartige als auch die OpenGL-Ansicht sind nicht nur zur Anzeige bestimmt, sondern können auch den aktuellen Dart der XGMAP3D verändern. Damit beide Ansichten während der Anwendung konsistent bleiben, müssen sie synchronisiert werden.

Im Strukturdiagramm der Anwendung (vgl. Abbildung 75) wurde dies bereits anhand der Pfeile zwischen beiden Anzeigeelementen angedeutet. Einen Rückschluss auf die genaue Art der Synchronisierung beider ließ sie allerdings noch nicht zu.

Bei dem Abgleich der beiden Anzeigeelemente müssen folgende Aspekte beachtet werden:

- Wer verändert die aktuell angezeigte XGMAP3D?
- Welche Änderungen müssen anschließend ausgeführt werden?
- Wie werden Zyklen des Aktualisierens behandelt?

Da die komplette OpenGL-Anzeige der XGMAP3D in der Klasse GLXGMAP3D erfolgt, und diese damit auch für die Zeichnungsoperationen verantwortlich ist, bekommt sie die ausführende Rolle der Änderungen. Wird zum Beispiel der aktuelle Dart mit der Maus verändert, so empfängt sie dieses Signal und leitet es an die Zeichenmethoden weiter. Damit sichergestellt ist, dass die baumartige Ansicht nach wie vor konsistent zu dem neuen aktuellen Dart ist, sendet die GLXGMAP3D anschließend ein Signal aus, das sogenannte `dartFaceChangedSignal`, welches die DartID des aktuellen Darts und die Regionsnummer seines DARTVOLUMES enthält.

Dieses Senden geschieht allerdings nur, wenn sich die Flächenkontur mit dem Umsetzen des aktuellen Darts ändert oder sie dazu gezwungen wird, ein Signal zu senden. Bevor das Signal abgesendet wird, wird die GLXGMAP3D in einen besonderen Zustand versetzt, in dem gespeichert wird, dass das Signal gesendet wurde. In diesem Zustand reagiert sie nicht mehr auf weitere Änderungsanfragen der angezeigten XGMAP3D.

Das `dartFaceChangedSignal` wird vom ListView (der baumartigen Listenansicht der XGMAP3D) des Hauptfensters in der Methode `changeCurrentFace` empfangen. In dieser Methode wird ermittelt, welches aktuelle Element im ListView nach einem Wechsel, welchen das `dartFaceChangeSignal` einleitet, ausgewählt werden muss. Dabei berücksichtigt es zwei verschiedene Fälle:

1. Der Dart gehört zu einem zugeklappten DARTVOLUME-Element der Liste.

   Dann wird dieses DARTVOLUME-Element ausgewählt.

2. Der Dart gehört zu einem ausgeklappten DARTVOLUME-Element der Liste.

   Dann wird das Flächen- beziehungsweise Flächenkontur-Element in der Liste ausgewählt, zu dem der Dart gehört.

Da die Auswahl eines Listenelements auch vom Benutzer vorgenommen werden kann, führt sie normalerweise dazu, dass ein Signal vom Hauptfenster an die GLXGMAP3D gesendet wird. In dem hier beschriebenen Fall hat aber die GLXGMAP3D bereits ein Signal gesendet und wartet nach wie vor auf dessen Abarbeitung. Diesen Umstand nutzt die Methode, die sich um eine Änderung der Auswahl im ListView kümmert, aus, um herauszufinden, ob eine Aktualisierung der GLXGMAP3D notwendig ist oder nicht. Anschließend akzeptiert der ListView die Änderungen, die die GLXGMAP3D ausgelöst hat, und hebt somit deren speziellen Zustand wieder auf.

Sollte sich die GLXGMAP3D nicht in einem solchen Zustand befinden, so wird der aktuelle Dart auf den Anker-Dart des ausgewählten Listeneintrags gesetzt.

Die folgende Abbildung zeigt zwei exemplarische Abläufe der Interaktion mit dem Benutzer. Im ersten Fall findet ein Dart-Wechsel durch Klicken mit der mittleren Maustaste innerhalb der GLXGMAP3D statt. Im zweiten Fall wird der aktuelle Dart durch einen Klick auf die Listenansicht verändert.

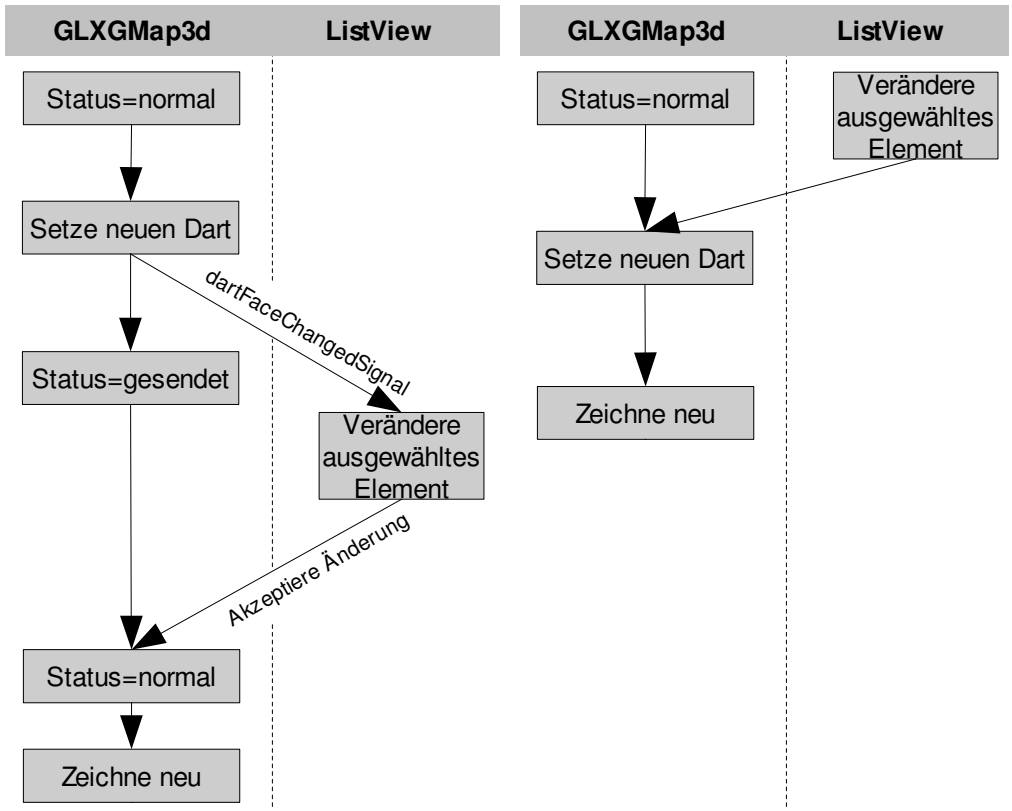


Abbildung 79: Die linke Abbildung zeigt, was sich ereignet, wenn der Benutzer den aktuellen Dart mit der mittleren Maustaste verändert. Rechts wird gezeigt, welche Aktionsfolge ausgeführt wird, wenn der Benutzer den aktuellen Dart mit Hilfe des ListViews ändert.

Durch diese Maßnahme ist es möglich, dass unterschieden werden kann, welches Widget ein Ereignis ausgelöst hat und was zu tun ist. Somit können Darts innerhalb der GLXGMAP3D mit allen verfügbaren Orbits gewechselt werden, während gleichzeitig sichergestellt ist, dass das aktuelle Element der XGMAP3D auch im ListView ausgewählt ist. Eine gesonderte Behandlung von Zyklen entfällt durch das hier beschriebene Verfahren.

8.2 Funktionsumfang und Bedienung der Anwendung

Der „OpenGL XGMap3d Viewer“ lässt sich mit wenigen Tasten der Tastatur und der Maus bedienen und steuern. In diesem Abschnitt sollen die Bedienkonzepte der Funktionalität des Viewers vorgestellt und erläutert werden.

Nach dem Start des Programms erscheint ein Fenster mit einer Menüleiste (1)[64] über zwei Anzeigefeldern. Unten rechts befinden sich zudem zwei Auswahlboxen. In Abbildung 80 ist das Programmfenster zu sehen. Die unterschiedlichen Bereiche sind durch Ziffern gekennzeichnet.

Das linke Anzeigefeld (2) dient der grafischen Anzeige einer XGMAP3D. Das heißt, in diesem werden die Darts oder eine Auswahl an Darts einer XGMAP3D mit Hilfe von OpenGL gezeichnet. Das rechte Feld (3) dient der Anzeige und Auswahl der XGMAP3D-Strukturen in einem ListView. In ihm werden die Strukturen baumartig aufgelistet. In erster Ebene werden die verschiedenen DARTVOLUMES der geladenen Volumen aufgelistet. Als Kinder enthalten diese eine äußere Kontur und gegebenenfalls inneren Konturen. An diesen Kindern hängen wiederum die DARTSURFACEPARTS und an diesen die DARTFACES. DARTFACES können ebenfalls innere Konturen besitzen, die an die entsprechende DARTFACE angehängt werden. Die in eine innere Kontur eingebetteten DARTSURFACEPARTS sind zusätzlich noch einmal als Kinder von diesen inneren Konturen im Baum vorhanden, so dass diese vom Benutzer leicht zuzuordnen sind (ein Beispiel hierzu ist in Abbildung 81 zu sehen). Soweit vorhanden, werden in diesem Feld noch weitere Informationen über die Strukturen ausgegeben. An der Überschriftenzeile des Feldes ist dieses ersichtlich (vgl. Abbildung 81). Durch das Anklicken von einzelnen Strukturen mit der rechten Maustaste können ebenfalls noch Informationen über diese angefordert werden.

Die unter diesem Anzeigefeld befindlichen Auswahlboxen (4) dienen der Auswahl von Volumen in der baumartigen Liste. Es ist ein Intervall einstellbar, in welchem Volumen im rechten Anzeigefeld (3) als ausgewählt markiert werden sollen. Entscheidend für diese Auswahl ist die durchschnittliche Farbe der Volumen. Jedes Volumen, dessen Farbe innerhalb des Intervalls liegt, wird ausgewählt. Auf das linke Fenster (2) hat diese Einstellung die Auswirkung, dass Darts dieser Strukturen gezeichnet werden.

64 vgl. mit markiertem Bereich in Abbildung 80

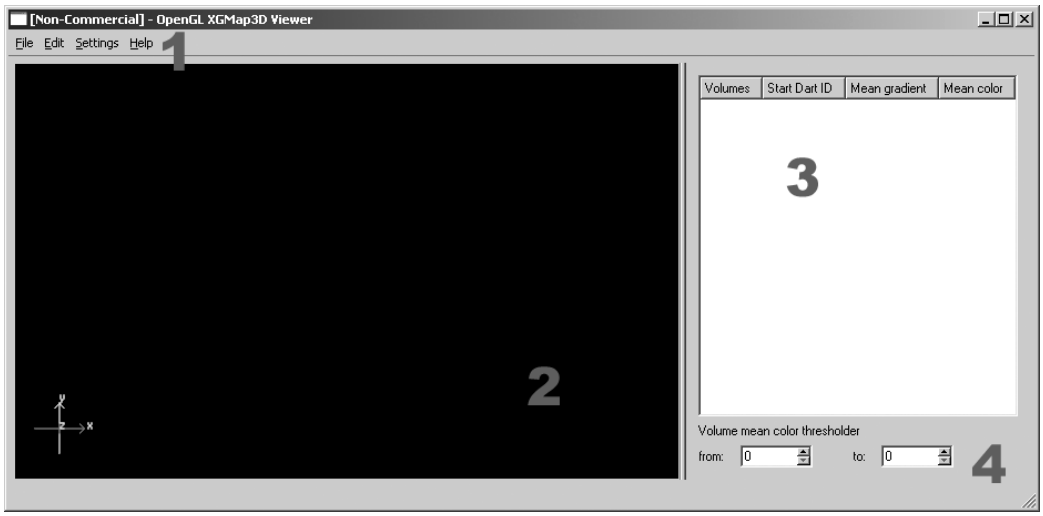


Abbildung 80: Ansicht des OpenGL XGMap3d Viewers und dessen Aufteilung in vier Bereiche.

Da bisher allerdings noch kein Volumen geladen wurde, sind beide Anzeigefelder leer. Im Menüeintrag „File“ gibt es die Möglichkeit Volumendaten zu laden. Ebenfalls kann man hier das Programm über „Quit“ beenden. In Abbildung 82 ist das Menü mit seinen Einträgen abgebildet. Es stehen zwei Varianten zum Laden von Volumen zur Verfügung. Zum einen lassen sich Volumen laden, die noch keine Vorverarbeitung erfahren haben. Diese Volumen müssen, damit eine XGMAP3D erstellt werden kann, gelabelt werden. Solche Volumen lassen sich über „Open Volume“ bearbeiten. Für bereits gelabelte Volumen steht der Eintrag „Open labeled Volume“ zur Verfügung. Bei diesem Aufruf wird die Durchführung des Union-Find-Wasserscheiden-Algorithmus (siehe Kapitel 5) ausgelassen und sofort die XGMAP3D erstellt. Da man mit Hilfe dieses Programms auch die geladenen Volumen verändern kann, ist der Eintrag „Save labeled volume“ zum Speichern dieser Änderungen vorgesehen. Dabei wird das geladene gelabelte Volumen in Schichtenbilder exportiert.

Nach dem Laden eines Volumens werden standardmäßig alle aktiven Elemente der XGMAP3D im linken Fenster angezeigt. Im rechten Fenster wird das aktive DARTVOLUME hervorgehoben. Möchte man die weiteren Strukturen, die dem DARTVOLUME untergeordnet sind, ansehen, so muss der entsprechende Eintrag aufgeklappt werden. Es ist immer das Element aktiv, auf dem der aktuelle Dart liegt. Ein Dart ist immer der aktuelle, somit auch am Anfang, bevor der Benutzer irgendetwas getan hat.

Die Darts der aktiven Elemente werden in unterschiedlichen Farben dargestellt. Die Farben haben folgende Bedeutung:

- rot: aktiver Dart
- hellgrün: äußere Kontur der Fläche, auf der der aktive Dart liegt
- dunkelgrün: innere Kontur der Fläche, auf der der aktive Dart liegt

- hellblau: äußere Schale des Volumens, auf dem der aktive Dart liegt
- blau: innere Schale des Volumens, auf dem der aktive Dart liegt

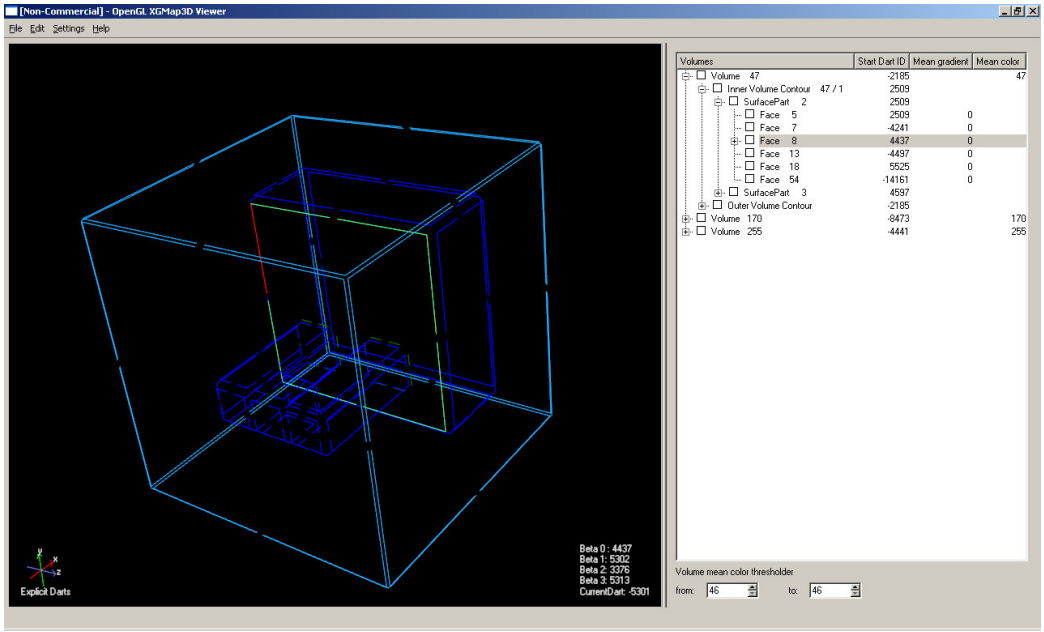


Abbildung 81: Markierungen der aktiven Elemente in unterschiedlichen Farben

In Abbildung 81 ist ein Beispiel für die Verwendung der unterschiedlichen Farben gegeben. In der rechten Liste ist zu erkennen, dass „Face 8“ aktiv ist. Es ist daher in der grafischen Darstellung durch grüne Darts dargestellt. „Face 8“ lässt sich noch weiter ausklappen, da es zwei innere Konturen besitzt. Zur Zeit ist allerdings die äußere Kontur aktiv. Alle weiteren Flächen der Volumenkontur, zu der „Face 8“ gehört, sind dunkelblau dargestellt, sie gehören zu einer inneren Kontur eines Volumens. Dieser Sachverhalt ist im ListView ablesbar. Die Flächen der entsprechenden äußeren Kontur des Volumens werden hellblau dargestellt.

Des Weiteren gibt es noch die Möglichkeit, nicht aktive Elemente grau darzustellen, indem man sie in der Liste der rechten Anzeige auswählt (oder entsprechend das Volumenfarben-Intervall so einstellt, dass alle Volumen berücksichtigt werden). Auf diese Weise hat man auch die Möglichkeit, alle Darts der XGMAP3D auf einmal anzuzeigen.

Vor allem bei Volumen, bei denen eine XGMAP3D mit sehr vielen Elementen entsteht, zeigt sich aber das Problem, dass eine übersichtliche Darstellung schwer möglich ist. Aus diesem Grund wird nach dem Laden auch nicht die gesamte XGMAP3D angezeigt (vgl. Abschnit 9.2.2).

Über den ListView auf der rechten Seite kann der Benutzer allerdings schnell zwischen verschiedenen Regionen des zu bearbeitenden Volumens umschalten. Dies geschieht

durch Anklicken eines „Volumes“ oder einer untergeordneten Struktur. Durch dieses Vorgehen wird auch das schnelle Zugreifen auf die gewünschte Struktur möglich.

Eine weitere Möglichkeit den Dart zu wechseln und somit auch zwischen den Strukturen zu wechseln, ist eine kombinierte Maus- und Tastatursteuerung. Befindet man sich mit dem Mauszeiger auf der linken Anzeige, so kann mit der mittleren Maustaste die Kante der aktuellen Flächenkontur gewechselt werden. Wird zusätzlich dazu die Umschalt-Taste gehalten, so kann man zur benachbarten Kante der anliegenden Fläche wechseln. Diese wird somit die aktive Fläche, da auf ihr nun der aktuell aktive Dart liegt. Hält man anstatt der Umschalt-Taste die Alt-Taste gedrückt, so wird mit einem Klick auf die mittlere Maustaste zur benachbarten Region gewechselt. Wieder werden die aktivierten Elemente angepasst. Diese Anpassung wird nicht nur im OpenGL-Anzeigefenster links dargestellt, sondern auch durch Markierung eines Elementes im ListView veranschaulicht.

Die Identifikationsnummer des aktuell ausgewählten Darts, und die der von ihm aus über die Orbits erreichbaren anderen Darts, werden unten rechts im linken Anzeigefeld dargestellt. Zudem wird noch eine Information ausgegeben, wenn der aktuelle Dart ein Zellschlüssel ist.

Mit den bisher beschriebenen Aktionen lassen sich Volumen laden und anzeigen. Des Weiteren lässt sich eine Auswahl der anzuzeigenden Strukturen treffen und verändern. Allerdings gibt es noch weitere Möglichkeiten, die Anzeige an die eigenen Wünsche anzupassen.

Unten links im linken Anzeigefeld ist ein Koordinatensystem abgebildet, welches die Ausrichtung der angezeigten Szene anzeigt. Um die gewünschte Auswahl aus dem richtigen Winkel zu betrachten, lässt sich mit Hilfe der linken Maustaste die Ansicht drehen. Dazu ist bei gedrückter linker Maustaste die Maus zu bewegen. Die Bewegungen werden in Drehungen umgesetzt (siehe Abschnitt 8.1.2). Standardmäßig wird um den Mittelpunkt aller Regionen gedreht, dies lässt sich aber durch eine Option im Menü verändern.

Neben dem Drehen ist auch ein Verschieben des angezeigten Inhalts möglich. Dazu muss, während die linke Maustaste gedrückt wird, ebenfalls die Umschalt-Taste gedrückt sein. Durch Bewegen der Maus findet dann die Verschiebung nach unten, links oder rechts statt. Möchte man in der Tiefe verschieben, so ist anstatt der Umschalt-Taste die Alt-Taste zu drücken. Die letzte Art der Verschiebung kommt einem Zoom gleich. Dieser kann auch über das Mausrad direkt ausgeführt werden.

Der Benutzer kann durch diese Operationen den Ausschnitt so bestimmen, wie er am geeignetsten ist. Ein Zurücksetzen der Ansicht ist durch Klicken der rechten Maustaste durchführbar.

Damit ist das Potential der Anwendung aber noch nicht erschöpft. Im Menü „Settings“ lassen sich noch weitere Einstellungen zur Individualisierung der Anzeige vornehmen. Es gibt folgende Menüeinträge:

- Rotate around: Wahl des Rotationsursprungs
- Dart rendering mode: Einstellung der Dartvisualisierung

- Highlight active elements: Wahl der zu zeichnenden Strukturen
- Volume rendering mode: Auswahl zwischen verschiedenen Arten der Darstellung
- Show volume border frame: Einschalten eines Rahmens

Zudem gibt es noch „Options“ worin sich mit „Import gaussian gradiant value“ der Wert der Standardabweichung des Gauß-Filters zur Erstellung des Gradientenvolumens einstellen lässt. Dieser Wert ist entscheidend, wenn für ein Volumen der Union-Find-Wasserscheiden-Algorithmus ausgeführt werden muss (vgl. Kapitel 5). Des Weiteren kann man mit „Face gradient threshold“ einen Wert einstellen, so dass über „Select low gradient DartFaces“ alle DARTFACES mit einem Gradientenwert kleiner diesem ausgewählt werden.

Die Haupteinträge des Menüs zeigt Abbildung 82.

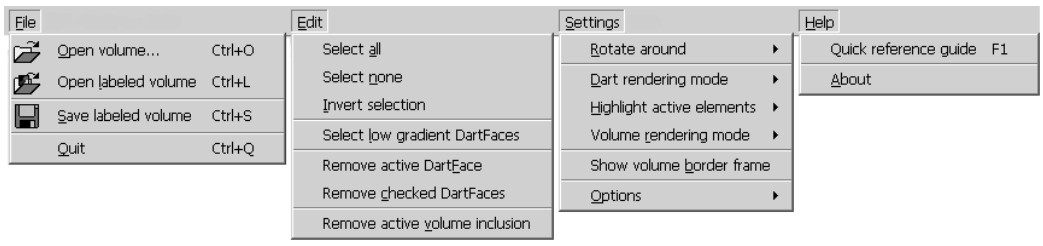


Abbildung 82: Das Menü des OpenGL XGMap3d Viewers

Bei der Wahl des Rotationsursprungs stehen folgende Optionen zur Verfügung:

- Node of current dart: Gedreht wird um den Knoten des aktuell aktiven Darts.
- Current DartFace: Als Rotationsursprung dient der Mittelpunkt der Bounding-Box der aktuell aktiven Fläche.
- Current DartVolume: Als Rotationsursprung dient der Mittelpunkt der Bounding-Box des aktuell aktiven Volumens.
- All visible items: Als Rotationsursprung dient der Mittelpunkt der Bounding-Box um alle angezeigten Darts.
- Volume center: Standardeinstellung, gedreht wird um den Mittelpunkt des gesamten Volumens.

Bei einer großen XGMAP3D mit vielen Elementen eignen sich vor allem die erste und letzte Möglichkeit, da bei den anderen der Berechnungsaufwand steigt. Der Rotationsursprung wird automatisch in die Mitte des Bildschirms gelegt, soweit nicht schon eine Translation stattgefunden hat.

Im Menüpunkt „Dart rendering mode“ kann man die Darstellung der Darts verändern. Bei der Einstellung „Explicit darts“ gibt es keine Überlagerungen zwischen Darts, selbst dann nicht, wenn sie vom gleichen Knoten aus in die gleiche Richtung zeigen. Dafür werden alle Darts ein wenig verschoben zu ihrer eigentlichen Lage gezeichnet. „Implicit

darts“ arbeitet ähnlich, allerdings bilden je zwei Darts eine durchgezogene Kante. Bei „Compact darts“ wird zusätzlich die Verschiebung der Darts aufgehoben (siehe Beispiele in Abschnitt 8.1.2).

Da es sinnvoll sein kann, nicht immer alle aktiven Elemente anzuzeigen, kann man unter „Highlight active elements“ einstellen, welche aktiven Elemente angezeigt werden sollen. Es stehen folgende Möglichkeiten zur Verfügung:

- Current DartVolume: Das aktive DARTVOLUME wird blau bzw. hellblau angezeigt.
- Current SurfacePart: Der DARTSURFACEPART, auf dem der aktive Dart liegt, wird goldfarben angezeigt (Option ist standardmäßig deaktiviert).
- Current DartFace: Die aktive DARTFACE wird hell- bzw. dunkelgrün angezeigt.
- Current Dart: Der aktive Dart wird rot hervorgehoben.

Sind die entsprechenden Optionen abgewählt, so werden die Darts der zugehörigen Strukturen auch nicht dargestellt. Nur wenn sie im ListView ausgewählt sind, werden sie noch mit grauer Farbe gezeichnet. Durch entsprechendes Abwählen ist es so beispielsweise möglich, nur einzelne Flächen oder Ebenen anzeigen zu lassen.

Der nächste Menüeintrag „Volume rendering mode“ stellt die folgenden Optionen zur Wahl (vgl. mit Abbildung 76):

- Render nothing: Die dargestellten Strukturen werden wie bisher durch ihre Darts angezeigt.
- Render volumecontour: Die Oberflächen des aktiven DARTVOLUMES werden gerendert dargestellt.
- Render selected volumes: Die Oberflächen aller ausgewählten DARTVOLUMES werden gerendert dargestellt.
- Look inside: Diese Option kann zusätzlich zu einer der Render-Optionen angewählt werden. Sie ermöglicht ein Hineinschauen in die angezeigten Objekte. Die vorderen Flächen werden in diesem Fall nicht gerendert.

Als letztes steht im Menü „Settings“ noch „Show volume border frame“ zur Verfügung. Ist dieser Eintrag angewählt, so wird ein Rahmen um die angezeigten Strukturen der XGMAP3D angezeigt, der der Größe des geladenen Volumens entspricht. Dieser Rahmen erleichtert in einigen Fällen die Orientierung, zum Beispiel, wenn nur wenige oder kleine Strukturen angezeigt werden.

Zur Bearbeitung des geladenen Volumens lassen sich im Menü „Edit“ einige Funktionen auswählen. Diese heißen:

- Select all: Alle DARTVOLUMES werden ausgewählt.
- Select none: Alle DARTVOLUMES werden abgewählt.
- Invert selection: Diese Funktion kehrt die aktuelle Auswahl an Strukturen der XGMAP3D um.
- Select low gradient DartFaces: Mit Hilfe dieser Funktion lassen sich alle DARTFACES – mit einem Gradienten kleiner als dem eingestellten – auf einmal anwählen. Alle anderen werden abgewählt.
- Remove active DartFace: Die aktive DARTFACE wird gelöscht. Dieses Löschen bewirkt, dass die Regionen, die an diese DARTFACE anschließen, verschmolzen werden.
- Remove checked DartFaces: Alle ausgewählten DARTFACES werden gelöscht. Durch das Löschen werden wiederum Regionen verschmolzen.
- Remove active volume inclusion: Ist ein Volumeneinschluss aktiv, so kann dieser mit Hilfe dieser Funktion gelöscht werden. Alle Regionen innerhalb dieses Einschlusses erhalten das Label der äußeren.

Durch diese Funktionalitäten ist es möglich, die automatische Segmentierung nachträglich zu bearbeiten. Gibt es zwei Regionen, die eigentlich nur eine darstellen, so können beide zu einer zusammengefasst werden. Nach solchen Änderungen ist dann auch die Speicherfunktion sinnvoll anzuwenden.

Weitere Einstellungsmöglichkeiten oder Funktionen gibt es nicht, allerdings kann im Menüeintrag „Help“ eine Kurzanleitung zur Bedienung des Programms unter „Quick reference guide“ gefunden werden. Diese Anleitung ist auch mit der Taste „F1“ zugänglich.

8.3 Statusanzeigen in der Konsole

Während das Programm ausgeführt wird, werden in der Konsole Informationen über den Programmablauf ausgegeben. Neben verschiedenen Statusanzeigen werden auch Fehlermeldungen ausgegeben, soweit der Benutzer fehlerhafte Eingaben tätigt.

Die Statusanzeigen betreffen den Erstellungsvorgang einer XGMAP3D inklusive des Einlesevorgangs eines Volumens und gegebenenfalls die Ausführung des Wasserscheiden-Algorithmus. Die Statusanzeigen betreffen den Fortschritt des Erstellungsvorgangs. Eine typische Ausgabe bei der erfolgreichen Erstellung einer XGMAP3D aus einem nicht gelabelten Volumendatensatz ist im Folgenden dargestellt:

```
==================== Einlesen der Bilddaten wird begonnen. ====================

      Einzulesende Volumendaten: C:/.../Vol_cube06.png
      VolumeImportInfo zum Erstellen des Volumen wurde erstellt
      Volumen wurde erstellt
      Volumendaten wurden eingelesen
      Anzahl der gefundenen Regionen = 3
      Ikonische Regionenrepraesentation wurde erstellt

~~~~~~~~~~~~~~~~~  Erstellung der GMap wird begonnen. ~~~~~~~~~~~~~~~~~

      Die Level-2 VolumeMap wurde erstellt.
      Eine Level-3 GMap mit 1008 Darts wurde erstellt.

~~~~~~~~~~~~ Erstellung der GMap in 1.375s abgeschlossen. ~~~~~~~~~~~~~

---------------------------- Erstellung der XGMap wird begonnen. ----------------------------

      126 Flaechenkonturen wurden erstellt und sortiert
      126 DartFaces wurden erstellt
      5 DartSurfaceParts wurden erstellt
      3 DartVolumes wurden erstellt

------------------------- Erstellung der XGMap in 0s abgeschlossen. -------------------------

************************ Erstellung des Listviews wird begonnen. ************************

      Anzahl der Eintraege im Listview: 4

******************** Erstellung des Listviews in 0s abgeschlossen. **********************

==================== Alle Datenstrukturen wurden erstellt. ====================
```

Diese Ausgabe ermöglicht es, schon während der Erstellung einen Überblick über die erstellten Strukturen und deren Anzahl zu bekommen. Des Weiteren wird die Rechendauer in Sekunden ausgegeben, um den Zeitaufwand besser beurteilen zu können.

9 Experimente und Beispiele der Anwendung

Nachdem im letzten Kapitel die Funktionsweise und der Aufbau des „OpenGL XGMap3d Viewers“ erläutert wurde, soll in diesem Kapitel anhand verschiedener Beispiele und Experimente ein Einblick in dessen Arbeitsweise und dessen Aussagemöglichkeiten erfolgen.

Zunächst werden die Strukturen der XGMAP3D noch einmal durch Beispiele einfacher synthetischer Volumendaten erläutert. In diesem Zusammenhang wird zudem auf einige besondere Fälle von Flächeneinschlüssen hingewiesen, und wie diese entstehen und zu deuten sind.

Anschließend wird anhand ausgewählter Real-Welt-Volumen der eigentliche Aufgabenbereich der Anwendung beschrieben werden. Da bei einer benutzbaren Anwendung immer auch die Verarbeitungsdauer und die Anforderungen an die Hardware eine Rolle spielen, wird auch darauf eingegangen werden.

Die Beurteilung von Segmentierungsergebnissen durch den Benutzer und mit Hilfe der Anwendung folgt in einem weiteren Abschnitt, bevor dann einige Experimente mit Nicht-Mannigfaltigkeiten präsentiert werden.

9.1 Beispiele anhand synthetischer Volumendaten

In diesem Unterkapitel sollen noch einmal kurz die Datenstrukturen einer XGMap3d dargestellt werden. Um die Übersicht in den Abbildungen zu wahren, werden die Beispiele anhand einfacher synthetischer Daten erfolgen.

Bereits in den vorigen Kapiteln wurde beschrieben, dass eine XGMap3d aus mehreren Teilstrukturen aufgebaut ist. Zu diesen gehören die DartFaces und die DartSurfaceParts. Diese Strukturen repräsentieren jeweils unterschiedlich große Anteile der Schalen von Regionen, welche als DartVolume-Struktur repräsentiert werden.

In Abbildung 83 ist eine gerenderte Region zu sehen (links in der Abbildung). Danach folgt die Darstellung des DartVolumes dieser Region. Es werden alle Darts, die zu dieser Region gehören, durch das DartVolume dargestellt. Die Oberfläche der angezeigten Region besteht aus zwei zusammenhängenden Komponenten. Dies wird ersichtlich, wenn man den aktuellen DartSurfacePart anzeigen lässt. Wie man sieht, wird dabei die DartFace nicht gewechselt (die aktuelle DartFace wird in grüner Farbe gezeichnet). Der zweite DartSurfacePart besteht in diesem Beispiel aus dem Henkel, der von der zweiten zur dritten Darstellung wegfällt. Zuletzt wird nur noch die aktuelle DartFace angezeigt. Diese besteht aus einer äußeren und zwei inneren Konturen.

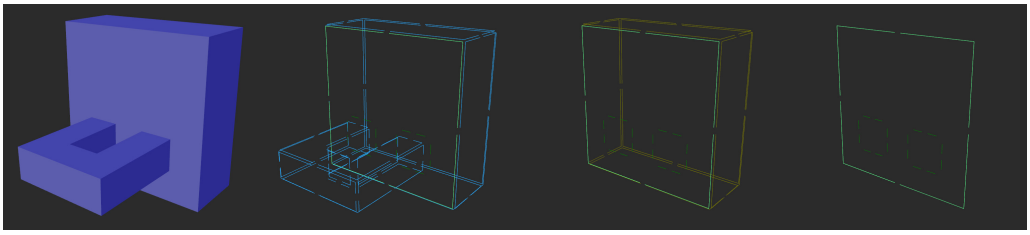


Abbildung 83: Es wird links ein Beispiel einer Region gezeigt. Danach folgen die Darstellungen des zugehörigen DartVolumes, eines zugehörigen DartSurfaceParts und eines DartFaces.

Das obige Beispiel zeigt schon einen interessanten Fall, da das dargestellte DartVolume durch die beiden inneren Konturen der einen DartFace in zwei Zusammenhangskomponenten zerfällt. Hierbei wird die Stärke der XGMap3d deutlich, da die Region trotzdem komplett als eine repräsentiert wird. Dies ist am gezeigten DartVolume ersichtlich, das aus allen Darts, die diese Region zu anderen abgrenzen, besteht.

In einigen Fällen hat sich gezeigt, dass ein DartSurfacePart nicht immer aus mehreren DartFaces bestehen muss. Es gibt einen Fall, bei dem ein DartSurfacePart nur aus einer DartFace besteht. In Abbildung 84 ist dieser Fall zu sehen.

Dieser Fall tritt immer dann auf, wenn die Oberfläche der inneren Kontur eben ist, also durch eine anliegende Region hervorgerufen wird, die an dieser Position vollständig von einer anderen Region umschlossen ist (vgl. hierzu auch Abschnitt 7.3.1). Diese

umschlossene Region ist in der Abbildung 84 grau gerendert dargestellt. Der besagte DARTSURFACEPART liegt daran an.

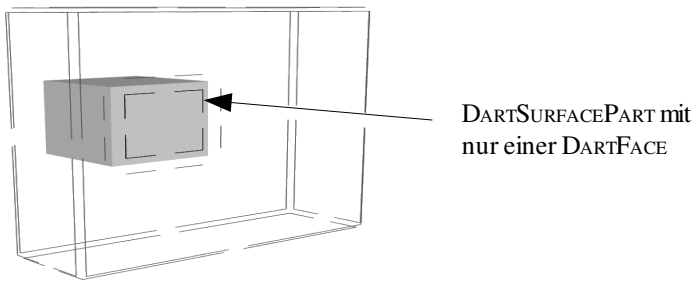


Abbildung 84: Darstellung eines DARTSURFACEPARTS, der nur aus einer DARTFACE besteht.

Es ist aber wichtig, dass diese einzelne DARTFACE einem eigenen DARTSURFACEPART untergeordnet wird. Sie gehört zu keinem anderen DARTSURFACEPART und kann deshalb auch nicht willkürlich einem anderen zugeordnet werden. Es gilt, dass jede DARTFACE ein Teil eines DARTSURFACEPARTS ist.

Werden die beiden im Beispiel dargestellten Region verschmolzen, so erhält man anstatt des DARTSURFACEPARTS mit nur einer DARTFACE, einen mit fünf DARTFACES. Dieser DARTSURFACEPART kann allerdings nicht einfach neu hinzukommen, es ist vielmehr der, welcher schon vorher bestand. Aus diesem Grund ist es wichtig, dass die Strukturen gleich richtig aufgebaut werden.

Die obigen Beispiele lassen vermuten, dass, wenn es eine DARTFACE mit inneren Konturen gibt, es auch immer mehrere DARTSURFACEPARTS gibt, da in einer inneren Kontur auch immer ein DARTSURFACEPART eingebettet ist. Dabei stimmt allerdings nur, dass in einer inneren Kontur immer ein DARTSURFACEPART eingebettet ist. Dieser muss sich aber nicht zwangsläufig vom DARTSURFACEPART der betrachteten DARTFACE unterscheiden. In der Abbildung 85 ist ein Beispiel zu sehen, bei dem eine DARTFACE der Region eine innere Kontur besitzt, die Region aber trotzdem nur aus einem DARTSURFACEPART besteht.

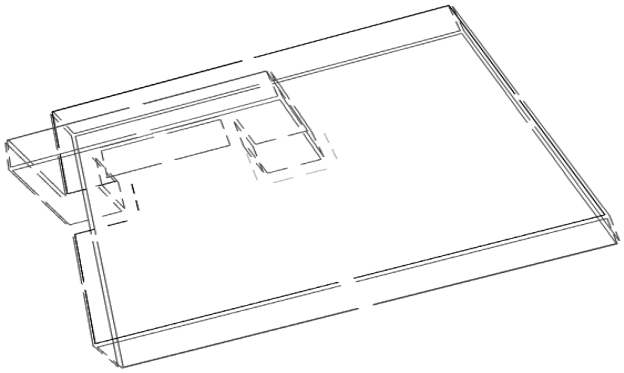


Abbildung 85: Das DARTVOLUME der Region besteht nur aus einer zusammenhängenden Komponente, besitzt aber trotzdem eine DARTFACE mit einer inneren Kontur.

Wie man sieht, kann es im dreidimensionalen Raum vorkommen, dass es Verbindungen zwischen einzelnen Komponenten gibt, die auf den ersten Blick nicht ersichtlich sind. Betrachtet man allerdings die innere Kontur, so wird dieser Zusammenhalt sehr schnell deutlich, da der eingebettete DARTSURFACEPART der gleiche, wie der des zugehörigen DARTFACES ist.

Anhand der gezeigten Beispiele wird deutlich, wie die Anwendung den Benutzer bei der Zuordnung einzelner Strukturen, beispielsweise DARTFACES zu DARTSURFACEPARTS, unterstützt. Auch erhält der Benutzer einen schnellen Überblick über die Zusammenhangskomponenten und wie diese untereinander verbunden sind. Zu diesem Zweck ist nicht nur die grafische Darstellung von Bedeutung, sondern ebenfalls der ListView (siehe Kapitel 8), der diese Zusammenhänge anhand der eindeutigen Namen der Strukturen ausdrückt.

9.2 Bearbeitung von Real-Welt-Volumen

Ein Ziel bei der Entwicklung der in dieser Arbeit beschriebenen Anwendung war es, dass die Bearbeitung von beliebigen Volumen möglich ist. In Kapitel 3.1 wurde schon kurz erwähnt, auf welche Weise Volumen entstehen können. Die Bearbeitung solcher Daten stellt gewisse Anforderungen, die erfüllt werden müssen.

Ein wichtiger Punkt ist, dass die Zeiten, in denen der Benutzer nicht aktiv eingreifen kann, so kurz wie möglich sind. Für diese Anwendung bedeutet dies, dass der Erstellungsvorgang der XGMAP3D-Strukturen nicht zu lange dauern darf. Weiterhin sollte beachtet werden, dass die Anwendung auf gängiger Hardware problemlos ausführbar ist. Aus diesem Grund wird zunächst die Komplexität des Erstellungsvorgangs (vgl. Kapitel 6.2 und 7.3) für verschiedene Real-Welt-Volumen Erwähnung finden.

Aber auch spätere Verarbeitungsschritte sollen den Benutzer nicht lange warten lassen. Ein Überblick über die benötigten Zeiten der einzelnen Operationen wird daher ebenfalls erfolgen.

Ein weiterer wichtiger Punkt für die effiziente Anwendbarkeit ist, dass die dargestellten Ergebnisse – sei es nun graphisch oder textuell – übersichtlich und individuell anpassbar sind. Die vorhandenen Möglichkeiten diesbezüglich werden in Abschnitt 9.2.2 erwähnt werden.

Weiterhin sind die zur Verfügung gestellten Operationen von Bedeutung. Dabei ist es für den Benutzer wichtig zu wissen, wie diese anzuwenden sind, und was sie bewirken. Aus diesem Grund werden Beispiele der Operationen mit Real-Welt-Volumen vorgestellt werden.

9.2.1 Komplexität des Erstellungsvorgangs

Soll eine XGMAP3D aus einem Volumen erstellt werden, so sind mehrere Schritte notwendig. Diese Schritte wurden im Laufe dieser Arbeit alle ausführlich besprochen. Der Ablauf erfolgt nach dem Laden der Volumendaten immer nach dem folgenden Schema:

1. Es wird die Berechnung der ikonischen Regionenrepräsentation mit Hilfe der in Kapitel 5 vorgestellten Union-Find-Wasserscheiden-Transformation durchgeführt.
2. Aus der ikonischen Regionenrepräsentation wird eine GMAP3D mit dem in Kapitel 6 beschriebenen Verfahren erstellt.
3. Anschließend kann eine XGMAP3D mit ihren Strukturen erstellt werden (vgl. Kapitel 7).
4. Die erstellten Strukturen werden, wie in Kapitel 8 beschrieben, in der Anwendung dargestellt.

Jeder dieser Schritte braucht, je nach zu bearbeitendem Volumendatensatz, entsprechend Zeit und Speicher. Um einen Einblick in die Komplexität zu geben, werden diese Schritte für vier unterschiedliche Volumendatensätze ausgeführt. Abbildung 86 zeigt die

verwendeten Volumendatensätze. Das Volumen „Teddybär“ besitzt rund eine Million Voxel. Der „Zahn“ besteht hingegen schon aus über 2,6 Millionen Voxeln. Das „Motor“-Volumen hat mehr als sieben Millionen Voxel und das Volumen des „Boston Teapot“ über 11,6 Millionen.

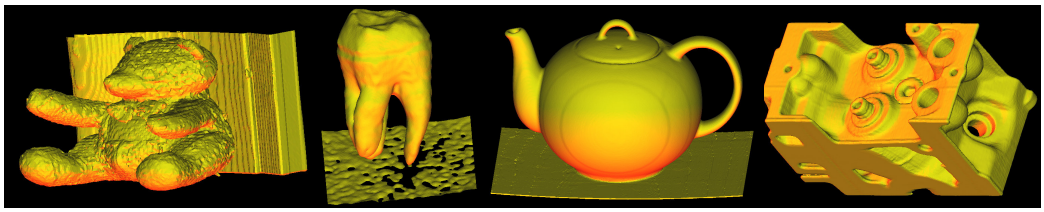


Abbildung 86: Die Abbildung zeigt die verwendeten Volumendaten in einer gerenderten Darstellung. Von links nach rechts: Teddybär, Zahn, Boston Teapot und Motor.

Anhand dieser Beispiele kann also gezeigt werden, wie sich die unterschiedlichen Größen der Volumendatensätze auf die Verarbeitung auswirken.

Wie bereits in Kapitel 5.4 erwähnt, hat die Größe eines Volumens auf den ersten Verarbeitungsschritt nur eine lineare Auswirkung bezüglich des Zeitbedarfs. Dies liegt im implementierten Union-Find-Wasserscheiden-Algorithmus begründet. Auch der Einfluss der Standardabweichung des verwendeten Gauß-Filters wurde bereits in diesem Kapitel diskutiert.

Allerdings hat die Wahl der Standardabweichung direkten Einfluss auf die Anzahl der zu verarbeitenden Regionen. Dieser Einfluss ist sogar größer als der Einfluss der Größe des Volumens. Für die Beispiele in diesem Abschnitt wurde die Standardabweichung jeweils so gewählt, dass die nachfolgenden Schritte problemlos ausführbar und die erzielten Ergebnisse akzeptabel sind. In der Tabelle 9.1 ist die jeweilige Standardabweichung im Feld der Anzahl der Regionen in Klammern angegeben.

Je größer ein Volumen ist, und je größer die Standardabweichung gewählt wird, umso länger dauert die Berechnung des Gradientenvolumens. Da der Schritt zur ikonischen Regionenrepräsentation allerdings nur eine Vorverarbeitung bei der Erstellung einer XGMAP3D darstellt, und auch durch das Laden von bereits gelabelten Volumen umgangen werden kann, wird an dieser Stelle nicht genauer auf den Berechnungsaufwand eingegangen.

Nachdem ein Volumen in Regionen unterteilt wurde, kann der zweite Schritt der Bearbeitung ausgeführt werden. Der Zeitaufwand der Erstellung ergibt sich zum Teil aus der Größe des Volumens. Dies hängt mit der Erstellung der Volume-Map (siehe Kapitel 6.2) zusammen. Von der Regionenanzahl kann man hingegen nicht direkt auf den Aufwand schließen. Da viele Regionen nicht zwangsläufig auch viele zu erstellende Darts nach sich ziehen. Diesen Sachverhalt kann man deutlich sehen, wenn man die Beispiele „Zahn“ und „Motor“ miteinander vergleicht. Für den Zahn wurden mehr Regionen gefunden, aber trotzdem deutlich weniger Darts erstellt.

Da die Volume-Map relativ schnell erstellt werden kann, der Aufwand ist linear zur Voxelanzahl des Volumens, ist der entscheidende Faktor für den Zeitaufwand die Anzahl der zu erstellenden Darts. Dies liegt darin begründet, dass für jeden erstellten Dart die zu vernähenden Darts gesucht und gespeichert werden müssen (vgl. Abschnitt 6.2.3). Ein Beispiel hierfür sind die Erstellungen der GMAP3DS für den „Boston Teapot“ und den „Motor“. Der Zeitaufwand ist ungefähr gleich, obwohl der „Motor“ deutlich weniger Voxel hat. Trotzdem lässt sich festhalten, dass der Berechnungsaufwand vor allem in der Größe des zu verarbeitenden Volumens begründet liegt, da für die Segmentierungsergebnisse großer Volumen meist auch mehr Darts erstellt werden.

Tabelle 9.1: Aufwand der Verarbeitung verschiedener Volumendatensätze[65]

Volumen	Anzahl der Regionen	Darts der GMAP3D			XGMAP3D	
		Anzahl	Speicherbedarf [MB]	Erstellungszeit [sec]	Speicherbedarf [MB]	Erstellungszeit [sec]
Teddybär (128,128,62)	1860 (3)	1.252.992	~43,018	~3,625	~4,650	~3,836
Zahn (128,128,160)	7093 (5)	4.185.544	~143,699	~11,567	~16,015	~13,659
Boston Teapot (256,256,178)	6919 (15)	5.291.564	~181,671	~22,012	~53,567	~20,090
Motor (256,256,110)	6734 (5)	6.308.556	~216,587	~20,630	~23,266	~59,555

Zum Speicherbedarf einer GMAP3D ist zu sagen, dass dieser linear von der Anzahl der Darts abhängt. Der Speicherbedarf einer GMAP3D hängt nur von der Länge der Liste der Darts ab. In Tabelle 9.1 setzt sich aus diesem Grund der abzulesende Speicherbedarf aus der Größe eines Darts von 36 Byte und der Anzahl der Darts zusammen.

Der Speicherbedarf und die Erstellungszeit einer XGMAP3D setzen sich im Gegensatz zur GMAP3D aus sehr viel komplexeren Faktoren zusammen.

Für den Speicherbedarf ist zu sagen, dass sich dieser vor allem aus der Anzahl zu erstellender Strukturen einer XGMAP3D ableiten lässt. Wie bereits in Kapitel 7.3 beschrieben, müssen DARTFACES, DARTSURFACEPARTS und DARTVOLUMES erstellt und gespeichert werden. Hinzu kommen die Hash-Tabellen, deren Längen von der Anzahl dieser Strukturen abhängig sind. Da jede Region im Volumen durch ein DARTVOLUME repräsentiert wird, hat die Anzahl der gefundenen Regionen einen Einfluss auf den Speicherbedarf. Allerdings kann man diesen nicht direkt aus der Anzahl der Regionen ableiten, da je nach Volumendatensatz die DartVolumes aus unterschiedlich vielen anderen Strukturen (DARTSURFACEPARTS und DARTFACES) aufgebaut sind. Wie viele von den anderen Strukturen erstellt werden, lässt sich leider nicht direkt ablesen, denn auch die Anzahl der Darts ist dafür nicht ausschlaggebend. Dies wird deutlich, wenn man die Beispiele „Boston Teapot“ und „Motor“ miteinander vergleicht. Bei ungefähr gleich vielen Regionen und weniger Darts für den „Boston Teapot“, wird für diesen deutlich mehr Speicher benötigt.

65 Die verwendete Systemkonfiguration entspricht der in Abschnitt 5.4.2 angegebenen.

Die Erstellungszeit hängt sowohl von der Zahl der Regionen als auch von der Zahl der Darts ab. Dies hängt damit zusammen, dass der langsamste Schritt bei der Erstellung der XGMAP3D das Finden der inneren Flächenkonturen ist (siehe Abschnitt 7.3.1). Da in diesem Schritt nur Flächenkonturen der gleichen Region auf Enthaltensein untersucht werden, geht dieser deutlich schneller, wenn es viele Regionen mit wenig Flächenkonturen gibt. Dies ist meist der Fall, wenn es in Bezug auf die Anzahl der Regionen wenige Darts gibt. Da der Test auf Enthaltensein in mehreren Schritten abläuft, ist auch die Beschaffenheit des Volumens ausschlaggebend. Liegen viele Flächenkonturen auf einer Ebene, so müssen die weiteren, deutlich aufwendigeren Überprüfungen durchgeführt werden (vgl. Abschnitt 7.3.1). Aus diesem Grund dauert auch die Berechnung der XGMAP3D des „Motor"-Volumens recht lange. Der abgebildete Motor hat relativ ebene Flächen. Dies führt aufgrund der Übersegmentierung dazu, dass viele DARTFACES auf einer Ebene liegen.

Ein weiterer Faktor für den Zeitbedarf, der allerdings gegenüber dem gerade genannten, nur einen geringen Einfluss hat, stellt das Finden von Flächenkonturen dar. Im Verlauf des Findens der Flächenkonturen werden alle Darts abgearbeitet. Dieser Schritt dauert natürlich umso länger, je mehr Darts vorhanden sind.

Nachdem alle Datenstrukturen erstellt wurden, werden diese innerhalb einer Benutzungsoberfläche dargestellt. Auch für die Erstellung der Inhalte der Anzeige wird noch Zeit benötigt. Dabei bedarf vor allem die Erstellung des ListViews (siehe Abschnitt 8.1.1) einer kurzen Berechnungszeit. Aufgrund dessen, dass zunächst nur Einträge für DARTVOLUMES erzeugt werden, hängt die benötigte Zeit von der Zahl der Regionen ab. Für die obigen Beispiele werden Zeiten von unter einer Sekunde benötigt. Zumeist liegt die Berechnungszeit im Zehntelsekundenbereich und hat beim gesamten Zeitbedarf der vorigen Schritte nur eine sehr geringe Auswirkung auf den vollständigen Zeitbedarf. Der Speicherbedarf hält sich aus den gleichen Gründen in Grenzen. Werden allerdings alle Strukturen im ListView angezeigt, so steigt dieser deutlich an.

9.2.2 Optimierungen der Darstellung

In diesem Abschnitt wird gezeigt, welche verschiedenen Möglichkeiten es gibt, Elemente einer XGMAP3D auszuwählen. Außerdem wird darauf eingegangen, wozu sich die verschiedenen Auswahlmodi in der Praxis eignen. Eine geeignete Auswahl der Elemente zu treffen, ist enorm wichtig, da die Anzahl der Darts einer GMAP3D meist deutlich über 1.000.000 liegt (siehe Abschnitt 9.2.1). Diese hohe Anzahl führt dazu, dass man in einer komplett dargestellten XGMAP3D nur sehr schwer Strukturen erkennen kann.

Anschließend wird diskutiert, welche Veränderungen in der Darstellungsart auf den ersten Blick sinnvoll erscheinen, im Rahmen dieser Arbeit jedoch aus guten Gründen nicht umgesetzt wurden.

Generell besitzt die in Kapitel 8 vorgestellte Anwendung folgende Möglichkeiten der Auswahl von Elementen der XGMAP3D:

1. Manuelle Auswahl der anzuzeigenden Elemente

2. Automatische Auswahl aller DARTVOLUMES, deren mittlere Intensität innerhalb eines frei wählbaren Intervalls liegt
3. Automatische Auswahl aller DARTFACES, deren mittlerer Gradient unterhalb einer auswählbaren Schwelle liegt

Der erste Punkt beschreibt den Vorgang, in dem der Benutzer die einzelnen Elemente der XGMAP3D im ListView mit einem Haken versieht, beziehungsweise einen gesetzten Haken entfernt. In diesem Auswahlmodus können einzelne DARTVOLUMES, DARTSURFACEPARTS, DARTFACES und innere Flächenkonturen ausgewählt werden, die angezeigt werden sollen.

Diese Art der Auswahl eignet sich besonders gut bei Test-Volumen oder bei Segmentierungen, die nur aus verhältnismäßig wenigen Regionen bestehen, da eine manuelle Auswahl sonst sehr viel Zeit in Anspruch nehmen würde. Die folgende Abbildung 87 zeigt das bekannte Teddybär-Volumen. Hierbei wurden in einem Experiment die DARTVOLUMES, die zu der hinteren Wand gehören, entfernt. Dieses führte bei insgesamt 309 DARTVOLUMES der XGMAP3D dazu, dass die Zahl der ausgewählten DARTVOLUMES von 98 auf 68 sank. Die manuelle Auswahl nahm ca. 4 Minuten in Anspruch.

Abbildung 87: Das linke Bild zeigt die anfänglich ausgewählten DARTVOLUMES als Oberflächenrendering. Im rechten Bild wurde die Wand manuell entfernt.

Der zweite Punkt beschreibt einen automatischen Auswahlmodus, welcher auf DARTVOLUMES arbeitet. Dazu kann durch zwei Eingabefelder in der Anwendung (siehe Kapitel 8) ein Intervall festgelegt werden, in dessen Grenzen sich die mittleren Intensitäten aller ausgewählten DARTVOLUMES befinden sollen.

Die mittlere Intensität einer Region ist gegeben durch:

$$\overline{intensity}_R = \frac{1}{|R|} \cdot \sum_{v_i \in R} f(v_i),$$

wobei R die Menge aller Voxel der Region ist und

f die Intensitätsfunktion des Volumens darstellt.

Die mittlere Intensität eines DARTVOLUMES ist die mittlere Intensität der Voxel, die als Label die Regionsnummer des DARTVOLUMES besitzen.

Die Auswahl anhand mittlerer Intensitäten eignet sich in segmentierten Real-Welt-Volumen sehr gut dazu, um die Anzahl der angezeigten Darts stark einzuschränken und einen besseren Überblick über die XGMAP3D zu erhalten. Wie bereits zu Punkt eins erwähnt, können DARTVOLUMES, die fälschlicherweise durch dieses Verfahren ausgewählt wurden, im Anschluss noch manuell deselektiert werden. Ebenso können nicht angezeigte DARTVOLUMES zusätzlich manuell ausgewählt werden.

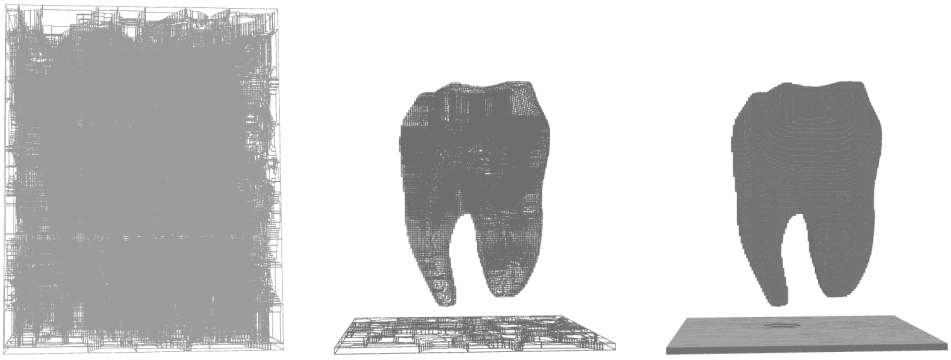


Abbildung 88: Diese Abbildung verdeutlicht anhand des Zahn-Volumens, den Einfluss der Auswahl von DARTVOLUMES aufgrund deren mittlerer Intensitätswerte. Von links nach rechts: Die Darts aller DARTVOLUMES, die Darts aller DARTVOLUMES, die innerhalb eines festgelegten Intensitäts-Intervalls liegen und ein Oberflächenrendering des mittleren Bildes.

Der dritte Punkt eignet sich am besten bei stark übersegmentierten Real-Welt-Volumen. Denn viele der entstandenen Regionen können dadurch miteinander verschmolzen werden, dass man die entsprechenden begrenzenden Flächen entfernt. Ein gutes Merkmal dafür, dass eine Fläche entfernt werden muss, ist ein geringer mittlerer Gradient der Fläche.

Der mittlere Gradient $\overline{grad_F}$ einer Fläche ist gegeben durch:

$$\overline{grad_F} = \frac{1}{|F|} \cdot \sum_{v \in F} \left\| \nabla I(v) \right\| ,$$

wobei F die Menge aller Voxel der Fläche ist und

∇I das Gradientenvolumen darstellt.

Der mittlere Gradient einer DARTFACE dF ist definiert als $\overline{grad_H}$,

wobei H die Menge aller Voxel v_H ist, für die gilt:

1. v_H liegt innerhalb der DARTFACE dF oder
2. v_H liegt innerhalb der DARTFACE, die über einen Dart von dF β_3-erreichbar ist.

Die folgende Abbildung 89 zeigt anhand zweier Beispiele, welche Voxel zu einer DARTFACE gehören.

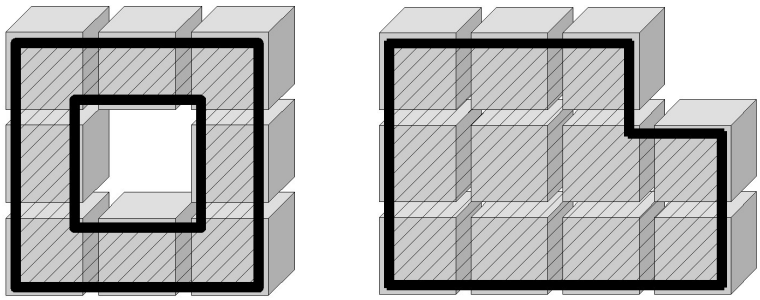


Abbildung 89: Beispiele für die Voxel, die zu DARTFACES (schraffiert) gehören: Die Konturen der DARTFACE beschränken jeweils die Zugehörigkeit der Voxel. Die linke Abbildung zeigt eine DARTFACE mit einem Einschluss, die rechte zeigt eine DARTFACE ohne Einschlüsse.

Der mittlere Gradient einer DARTFACE gibt die Stärke der trennenden Fläche an, die in dem Ausgangsvolumen vorhanden ist. Viele dieser Flächen haben einen eher geringen mittleren Gradienten, was ein Zeichen dafür sein kann, dass diese durch die Übersegmentierung entstanden sind. Die DARTFACES, die diese Flächen repräsentieren, können durch die in Punkt drei beschriebene Methode markiert werden. Das folgende Beispiel in Abbildung 90 zeigt einen Anwendungsfall für diese Methode, auf dem bereits vorgestellten Teddybär-Volumen.

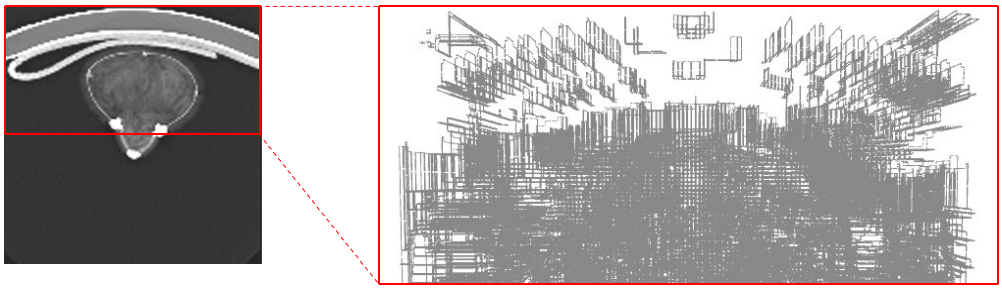


Abbildung 90: Wählt man bei dem Teddybär-Volumen (links) alle DARTFACES aus, deren mittlerer Gradient kleiner als 4.0 ist, so erkennt man, dass für die hintere Wand des Volumens einige trennende Flächen übersegmentierter Regionen ausgewählt werden.

Eine weitere Anzeigeoption ist die Aktivierung des Hineinschauens in Volumen. Diese eignet sich jedoch nur für segmentierte Volumen, die aus sehr wenigen Regionen bestehen. Zusätzlich sollten die Regionen Einschlüsse besitzen, denn ansonsten führt dieser Anzeigemodus zu keinem Mehrwert. Generell lässt sich sagen, dass es sich eher anbietet, ihn in Zusammenhang mit dem Oberflächenrendering eines einzelnen DARTVOLUMES einzusetzen, um dessen Einschlüsse zu veranschaulichen. Bei der Anwendung auf ein Oberflächenrendering aller ausgewählter DARTVOLUMES kann ansonsten leicht der Überblick verloren gehen.

Eine Optimierung der Darstellung kann allerdings nicht nur darin bestehen, eine möglichst übersichtliche Anzeige der Elemente zu bieten, sondern auch darin, die Anzeigegeschwindigkeit zu erhöhen. Dies ist notwendig, da für jede Neuberechnung der Anzeige folgende Schritte durchgeführt werden müssen:

1. Für jede Berechnung der Anzeige der Elemente der XGMAP3D muss die zugrunde liegende GMAP3D komplett durchlaufen werden.
2. Für jede Berechnung der gerenderten Anzeige der Oberflächenkonturen muss das zugrunde liegende Volumen der ikonischen Regionenrepräsentation durchlaufen werden.

Das verwendete System zur Anzeige dieser Elemente, OpenGL, verfügt bereits über eingebaute Mechanismen, die es erlauben, dass Anzeigeelemente zwischengespeichert werden können, um eine Neuberechnung überflüssig zu machen. Die hier zu nennenden Verfahren sind verwaltete Listen von Anzeigeelementen und sogenannte Vertex-Buffer.[66]

Dabei bezeichnen Vertex-Buffer ein ähnliches Konzept, wie die verwalteten Listen, sind allerdings auf die Speicherung von Knoten spezialisiert und können dadurch die Anzeige noch weiter beschleunigen, als verwaltete Listen. Sie wurden nur der Vollständigkeit halber aufgezählt, werden aber im Weiteren nicht genauer beschrieben.

Die verwalteten Listen werden in OpenGL auch *displayLists* genannt. Sie können, ähnlich einer Funktion, Gruppen von Zeichenanweisungen kapseln. Dies bietet den Vorteil, dass eine solche Liste zur Anzeige aller Elemente nur einmal erstellt werden muss und dann für beliebige Rotationen, Translationen und Zoomstufen der dargestellten Szene immer wieder aufgerufen werden kann, ohne dass eine Neuberechnung notwendig ist. Auf den oben beschriebenen Fall angewandt, bedeutet dies, dass die GMAP3D beziehungsweise das Volumen nur einmal durchlaufen werden muss, damit die *displayList* der darzustellenden Szene erstellt werden kann. Anschließend können sowohl die Rotation, als auch die Translation und der Zoom sehr viel schneller ausgeführt werden. Ein zusätzlicher Vorteil der *displayLists* ist, dass die Zeichenbefehle, die sich auf dieser Liste befinden, in einer kompilierten Form gespeichert werden. Dies führt zu einer schnelleren Ausführung der Zeichenbefehle. Außerdem werden die Zeichenbefehle in dem schnelleren Speicher der Grafikkarte abgelegt, sofern dies möglich ist.

Abbildung 91 veranschaulicht die Umstellung von normal abgearbeiteten Zeichenbefehlen auf *displayLists*.

66 Einen guten Überblick über diese grundlegenden Beschleunigungsverfahren bei der Darstellung dreidimensionaler Szenen mit OpenGL findet sich in den entsprechenden Kapiteln in [SWND06].

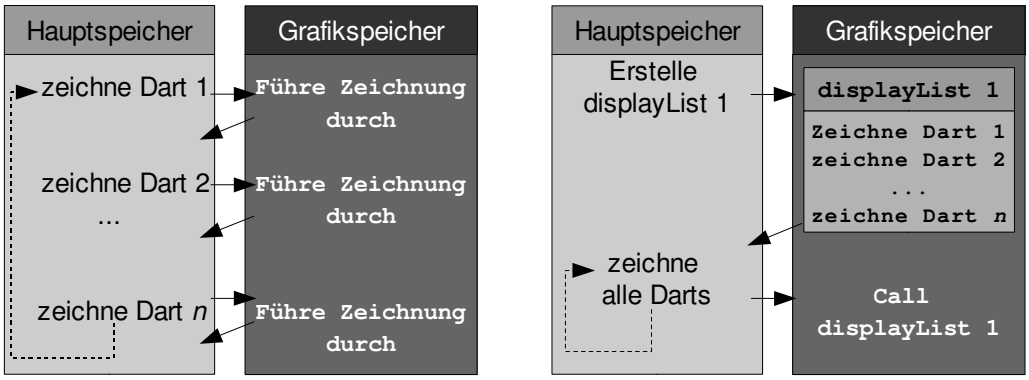


Abbildung 91: Diese Abbildung zeigt, welchen Unterschied der Verarbeitung die Verwendung einer displayList bewirkt. Die linke Teilabbildung zeigt den Ablauf ohne displayLists, die rechte zeigt den Ablauf mit displayLists. Der gestrichelte Pfeil deutet jeweils die Hauptzeichenroutine des Programms an.

Leider hat die Anwendung von *displayLists* auch einige Nachteile. So können die Listen im Nachhinein nicht mehr manipuliert werden. Einzig das Überschreiben einer vorhandenen *displayList* mit einer neuen ist möglich. So bleibt bei jeder Veränderung eine Neuerstellung mindestens einer der Listen unumgänglich. Leider dauert eine Erstellung einer umfangreichen Liste aber recht lange (> 10 Sekunden).

Da die Anzeige sich aber nicht auf ein einmaliges Rendern von Darts und Oberflächen beschränkt, sondern vielmehr alle angezeigten Elemente zur Laufzeit verändert werden können, werden die oben vorgestellten *displayLists* nicht verwendet. In der Praxis hat sich gezeigt, dass es zu akzeptieren ist, wenn die Rotation, Translation und der Zoom einer angezeigten Karte verhältnismäßig langsam sind, dafür aber die Anzeige schnell auf Interaktionen reagiert. Die Alternative mit der Verwendung von *displayLists* würde diesen Mangel nicht aufweisen, schränkt aber die Interaktivität der Anwendung so stark ein, dass von einer Interaktion des Nutzers mit der Anwendung keine Rede mehr sein kann. Da diese Interaktivität aber im Rahmen der vorliegenden Anwendung besonders wichtig ist, wird auf den Einsatz von *displayLists* verzichtet.

Diese Entscheidung führt dazu, dass das Rendern von Volumenoberflächen bei großen Volumen verhältnismäßig lange dauern kann (> 1 Sekunde). Gleiches gilt für die Darstellung einer GMAP3D mit sehr vielen Darts.

9.2.3 Regionenreduktion durch Veränderungen der Strukturen

Wie bereits erwähnt, liefert das implementierte Segmentierungsverfahren meist eine Übersegmentierung des Ausgangsvolumens. Da eine XGMAP3D mit weniger DARTVOLUMES (also Regionen) aber sehr viel übersichtlicher darzustellen ist, muss die Anzahl der Regionen durch geeignete Verfahren reduziert werden können. Eine Möglichkeit der Reduktion ist die Verschmelzung von Flächen, die nach einem bestimmten Kriterium ausgewählt wurden. Ein Kriterium, das Auswählen von zu

entfernenden Flächen anhand des mittleren Gradienten, wurde bereits im vorigen Abschnitt 9.2.1 beschrieben. Aufbauend darauf wird ein Experiment durchgeführt:

Die ursprüngliche XGMAP3D besteht bei dem Teddybär-Volumen aus 3818 DARTVOLUMES. Wählt man nun, wie in Abbildung 90 beschrieben, einige DARTFACES automatisch aus und entfernt diese anschließend, so erhält man eine XGMAP3D, welche aus lediglich 309 DARTVOLUMES besteht. Diese 309 Regionen stellen immer noch eine Übersegmentierung dar, doch hält sich diese gegenüber dem ursprünglichen Segmentierungsergebnis in Grenzen.

Um das Ergebnis eines solchen Verschmelzungsvorgangs beurteilen zu können, ist die automatische Auswahl von DARTVOLUMES anhand der mittleren Intensitätswerte hilfreich. Die folgende Abbildung 92 zeigt die gerenderten Oberflächen aller DARTVOLUMES des Teddybär-Volumens, deren mittlere Intensitäten innerhalb eines gewählten Intervalls liegen, sowohl vor als auch nach der Verschmelzung. Dabei sank die Anzahl der angezeigten Regionen von 860 auf 98.

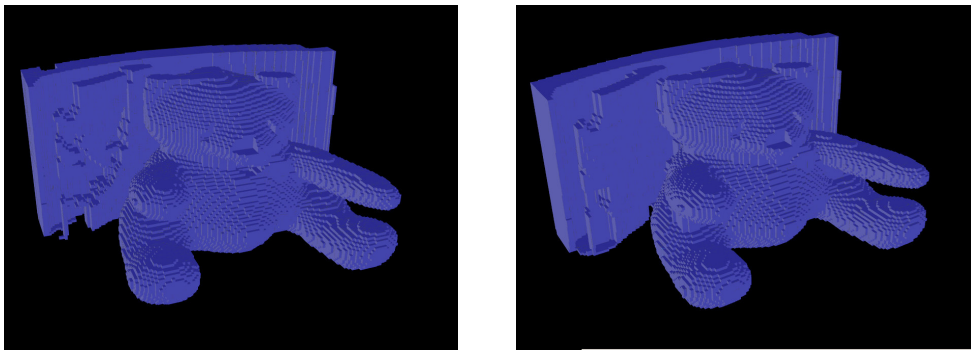


Abbildung 92: Das Teddybär-Volumen vor (links) und nach (rechts) der Entfernung der in Abbildung 90 angezeigten Flächen.

Die Laufzeit für die Verschmelzungsoperationen aller Regionen beträgt circa 25 Sekunden, wobei circa 2,5 Sekunden auf das Umlabeln des Label-Volumes entfallen. Der Rest der Zeit wird zur Neuberechnung der Datenstrukturen benötigt.

Eine weitere Möglichkeit der Reduktion von Regionen ist das Löschen von Volumeneinschlüssen. Sie wurde bereits in Abschnitt 7.5.3.1 beschrieben und läuft, im Gegensatz zu dem Verschmelzen von Flächen, komplett auf den Datenstrukturen der XGMAP3D beziehungsweise der zugrunde liegenden GMAP3D ab, so dass eine Neuberechnung beider Strukturen nicht benötigt wird. Diese Methode eignet sich vor allem dann, wenn Volumeneinschlüsse als Folge von Scheinkonturen auftreten oder nicht als Ergebnis der Segmentierung gewünscht sind.

Um das Ergebnis einer solchen Verschmelzung darzustellen, folgt nun ein Experiment. Dabei wurde als Ausgangsvolumen das Würfel-Volumen herangezogen, welches schon in Abschnitt 5.4.2 kurz erläutert wurde. Es besteht aus zwei ineinander verschachtelten Würfeln.

Die einzelnen Würfel liegen jeweils bei $BoundingBox_{w_1} = \{(20,20,20),(80,80,80)\}$ beziehungsweise bei $BoundingBox_{w_2} = \{(40,40,40),(60,60,60)\}$.

Nach der Faltung mit einer abgeleiteten Gauß-Funktion der Standardabweichung $\sigma = 4$ erhält man ein Gradientenbetragsvolumen. Wird dieses mit dem implementierten Wasserscheiden-Verfahren segmentiert, so entstehen Scheinkonturen, welche unter anderem einen Volumeneinschluss bilden, wie die folgende Abbildung 93 zeigt.

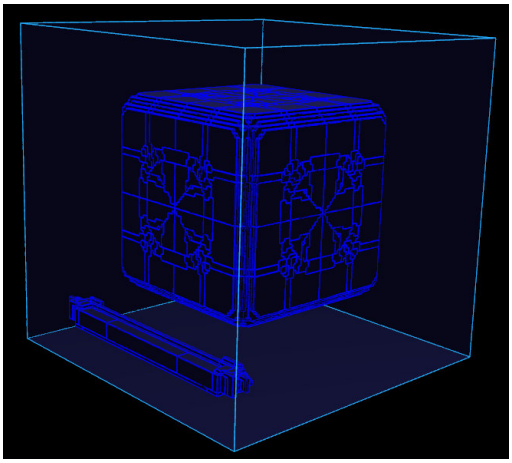


Abbildung 93: Die Abbildung zeigt das Ergebnis der beschriebenen Segmentierung (mit $\sigma = 4$)des Würfel-Volumens. Der fälschlicherweise segmentierte innere Volumeneinschluss ist die längliche Region, die sich unten links im Bild befindet.

Da dieser Volumeneinschluss als Ergebnis der Segmentierung nicht erwünscht ist, muss er entfernt werden. So benötigt die XGMAP3D zu diesem Zeitpunkt noch 478 Regionen, was sich durch die Entfernung des Einschlusses noch verringern lässt. Die in Abschnitt 7.5.3.1 beschriebene Methode entfernt einen Volumeneinschluss und ist zudem sehr schnell ausgeführt. So wird für das Entfernen des Volumeneinschlusses lediglich eine Zeit von deutlich unter einer Sekunde benötigt, obwohl 46 Regionen entfernt wurden. Die daraus resultierende XGMAP3D besteht nun lediglich aus 432 DARTVOLUMES. Des Weiteren gehört zu dem äußersten DARTVOLUME, welches den unerwünschten Volumeneinschluss besaß, nun nur noch eine innere Schale. Die XGMAP3D nach dem beschriebenen Entfernungsvorgang zeigt die folgende Abbildung 94.

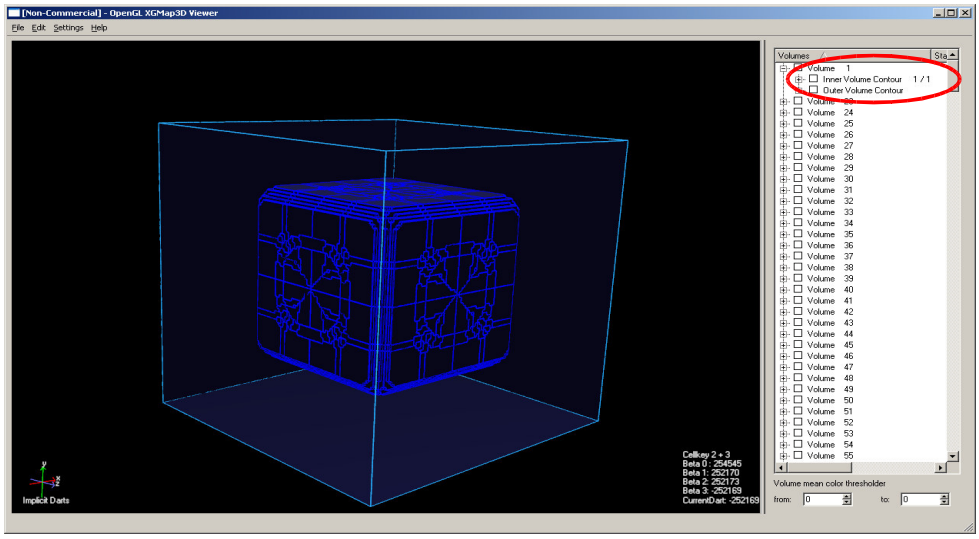


Abbildung 94: Das Ergebnis, welches nach der Entfernung des inneren Volumeneinschlusses vorliegt. Man erkennt gegenüber Abbildung 93 deutlich, dass der Volumeneinschluss entfernt wurde und dass das äußere DARTVOLUME *nun lediglich eine innere Schale besitzt.*

Dieser Abschnitt hat anschaulich vermittelt, wofür sich die implementierten Operationen auf der XGMAP3D eignen. Außerdem konnte gezeigt werden, dass sie sich auch für die Anwendung auf Volumen der realen Welt eignen, da sie eine sehr gute Zeitkomplexität besitzen. Dies ist besonders wichtig, da diese Operationen dazu verwendet werden, das Segmentierungsergebnis zu bearbeiten. Würden sie zu viel Zeit in Anspruch nehmen, so wäre ihre Eignung für die Praxis in Frage gestellt. Dabei ist der geringe Zeitbedarf bei Verfahren, welche auf Volumendaten arbeiten, keinesfalls als normal zu bezeichnen und kann deshalb an dieser Stelle abschließend als sehr positiv hervorgehoben werden.

9.3 Beurteilung von Regionen

Der „OpenGL XGMAP3D Viewer“ bietet nicht nur die Möglichkeit der Darstellung einer XGMAP3D, sondern kann ebenfalls Informationen zu den einzelnen Regionen, zu Volumenschalen oder zu Flächen geben. Dies wurde bereits in Kapitel 8.2 kurz angedeutet, aber noch nicht näher ausgeführt.

Diese Informationen können dem Benutzer dabei helfen, die Segmentierung zu verbessern oder Regionen besser zu klassifizieren. In diesem Unterkapitel werden diese Möglichkeiten des Programms anhand von Beispielen vorgestellt.

In Abschnitt 9.2.2 und 9.2.3 wurde bereits die Möglichkeit erwähnt, DARTFACES nach ihrem durchschnittlichen Gradienten auszuwählen. Die Information des durchschnittlichen Gradienten kann vom Benutzer auch aus dem ListView abgelesen werden. In der Spalte „Mean Gradient“ eines DARTFACES steht dieser Wert. Wie in Abbildung 95 zu sehen ist, besitzt die ausgewählte Fläche „Face 75“ einen durchschnittlichen Gradienten von 83,2754.

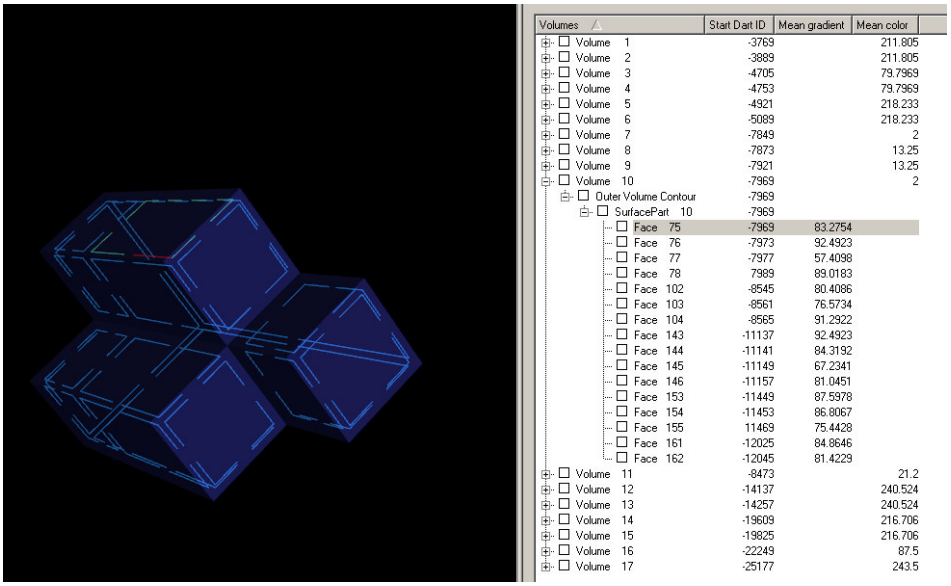


Abbildung 95: Das bearbeitete Volumen besteht aus 17 Regionen. Die Region "Volume 10" besteht aus einem Oberflächenteil, welches sich aus 16 Flächen zusammensetzt. Für jede Struktur sind die verfügbaren Informationen angegeben.

Jede Struktur der XGMAP3D besitzt einen Anker-Dart. Dieser wird durch seine DartID in der Spalte „Start Dart ID“ angegeben. Des Weiteren gibt es noch die Spalte „Mean color“, die die durchschnittliche Farbe einer Region (wird im ListView mit „Volume“ bezeichnet) angibt. Zu Flächen („Face“) und Oberflächenteilen („SurfacePart“) wird keine mittlere Farbe angegeben.

Sollte eine Fläche einen sehr geringen durchschnittlichen Gradienten aufweisen, so kann dies ein Indiz dafür sein, dass die Trennung der beiden anliegenden Regionen durch die Übersegmentierung zustande gekommen ist. In diesem Fall kann es sinnvoll sein, die Fläche zu entfernen, da mit hoher Wahrscheinlichkeit beide Regionen an besagter Stelle Teile eines gleichen Objekts repräsentieren.

Auch die durchschnittliche Farbe der Regionen kann dem Benutzer helfen Regionen eines Objekts zu finden. Meist haben alle Regionen, in die ein Objekt zerfallen ist, ähnliche Farben. Um also die Segmentierung von Objekten zu verbessern, kann der Benutzer anhand der durchschnittlichen Farbe der Regionen relativ schnell alle Regionen finden, zu denen das Objekt zerfallen ist. Sind diese Regionen gefunden, so kann der Benutzer diese verschmelzen (siehe 9.2.3).

Regionen können allerdings nicht nur anhand ihrer Farbe beurteilt werden, sondern auch durch ihre Euler-Charakteristik (siehe Abschnitt 3.4.6 und 7.5.2). Diese Information kann der Benutzer für alle Schalen („Outer Volume Contour“ und „Inner Volume Contur“) durch ein Anklicken mit der rechten Maustaste abrufen. In Abbildung 96 ist ein Beispiel für eine solche Information zu sehen. Die ausgewählte Kontur besitzt 46 Knoten, 66 Kanten und 22 Flächen. Daher ergibt sich eine Euler-Charakteristik von 2. Man kann also relativ schnell erkennen, dass die gewählte Kontur homöomorph zu einer Kugel ist.

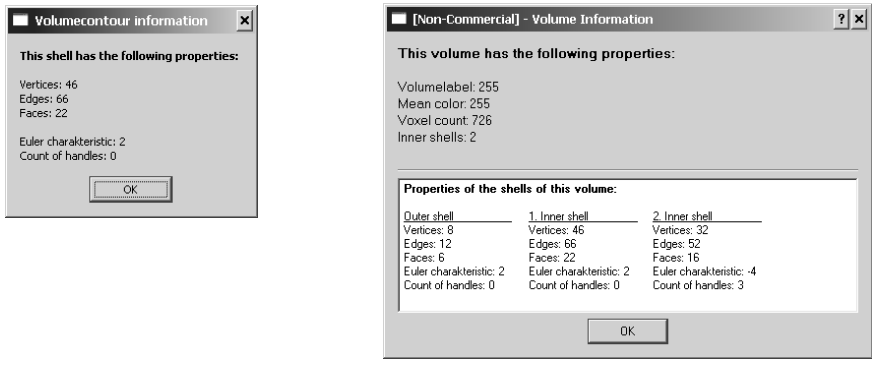


Abbildung 96: Informationen einer Regionenkontur werden in Informationsfenstern ausgegeben (links). Alle Informationen zu einer Region werden in anderen Fenstern (rechts) ausgeben.

Beim Anklicken einer Region („Volume“) werden die Informationen dieser Region und der untergeordneten inneren Schalen zusammengefasst. Rechts in Abbildung 96 ist auch dafür ein Beispiel zu sehen.

In Abbildung 97 sind zwei Regionen inklusive einiger Informationen abgebildet. Man kann schnell erkennen, dass das obere Beispiel ein Würfel ist, da die Region aus sechs Flächen besteht und eine Euler-Charakteristik von zwei besitzt.

Das untere Beispiel stellt einen Torus dar, da die Euler-Charakteristik null beträgt. Obwohl dieser aus recht vielen Voxeln (140), vielen Knoten (208), Kanten (318) und Flächen (110) besteht, kann dieses durch die automatischen Zählungen schnell herausgefunden werden.

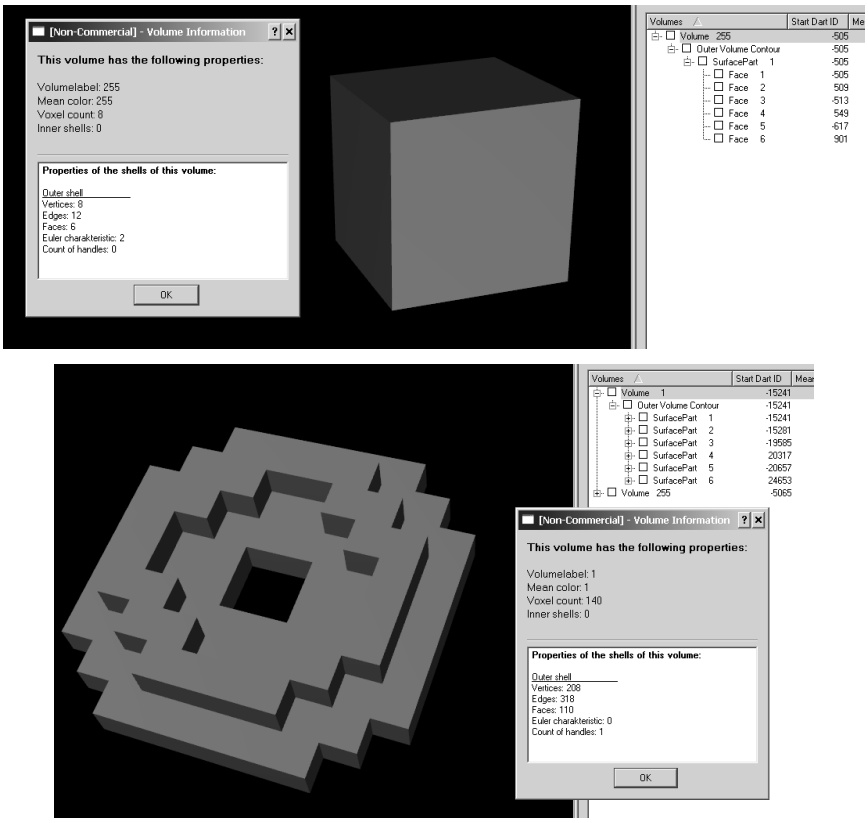


Abbildung 97: Regionen können automatisch hinsichtlich ihrer Größe und Euler-Charakteristik untersucht werden. Somit lässt sich schnell die Aussage treffen, dass das obige Beispiel homöomorph zu einem Würfel und das untere Beispiel homöomorph zu einem Torus ist.

Weitere topologische Merkmale, wie die Nachbarschaft von Regionen, lassen sich derzeit noch nicht anzeigen. Die Information der Nachbarschaft ist allerdings vorhanden und kann vom Benutzer durch Wechseln des β_3-Orbits gewonnen werden. Durch diese Vorgehensweise lässt sich zumindest bestimmen, welche Regionen an einer Fläche zusammenliegen.

9.4 Experimente mit Nicht-Mannigfaltigkeiten

Durch die Anwendung verschiedener Segmentierungsalgorithmen und Volumendaten können die verschiedensten Konfigurationen von Voxeln auftreten, die zusammen eine Region bilden. Bei einigen solcher gefundenen Regionen scheint es auf den ersten Blick relativ unverständlich zu sein, weshalb eine dieser Regionen durch eine entsprechende 3-G-Map beziehungsweise 3-XG-Map repräsentiert wird. In Abschnitt 3.4.3 in der Definition der G-Map wurden Quasi-Mannigfaltigkeiten (Definition 3.3.6, Abschnitt 3.3.1) zugelassen. Da Zellkomplexe in dieser Arbeit durch die Repräsentation von Voxelkanten zustande kommen, bedeutet dies, dass repräsentierte Regionen, die quasi-mannigfaltig sind auch mannigfaltig sind. Daher ist es interessant zu betrachten, wie Fälle von Nicht-Mannigfaltigkeiten behandelt werden.

Die Repräsentation dieser Fälle in einer 3-G-Map hat außerdem Auswirkungen auf die 3-XG-Map. Anhand von Beispielen sollen daher in diesem Unterkapitel einige Erläuterungen zu besonderen Fällen erfolgen.

Zunächst stellt sich die Frage, wie Nicht-Mannigfaltigkeiten in einer 3-G-Map auftreten können. Wie bereits in Abschnitt 3.3.1 definiert, ist ein Zellkomplex dann nicht mehr mannigfaltig, wenn an eine (*i-1*)-Zelle[67] mehr als zwei *i*-Zellen anliegen. Dieser Sachverhalt lässt sich auf die 3-G-Map dahingehend anwenden, indem man die Orbits entsprechend als *i*-Zellen betrachtet. Das Orbit $\beta_0 \circ \beta_2 \circ \beta_3$ beschreibt beispielsweise eine 1-Zelle und das Orbit $\beta_0 \circ \beta_1 \circ \beta_3$ eine 2-Zelle. Dieser Sachverhalt wurde bereits in Abschnitt 3.4.3 in Definition 3.4.6 erwähnt.

Bildhaft vorgestellt, entspricht eine 1-Zelle einer Kante. Zum Wechseln einer Kante in einer 3-G-Map ist das β_1 -Orbit erforderlich. Aus diesem Grund erreicht man über das Orbit $\beta_0 \circ \beta_2 \circ \beta_3$ genau die Darts, die zusammen eine Kante bilden. Entsprechende Beispiele kann man für allen anderen *i*-Zellen aufstellen.

Eine 3-G-Map ist also beispielsweise dann nicht mehr mannigfaltig, wenn es eine Kante gibt, an der mehr als zwei Flächen anliegen. In Abbildung 98 ist ein Beispiel einer solchen 3-G-Map abgebildet.

67 In diesem Abschnitt gilt $i \in \{0,1,2,3\}$.

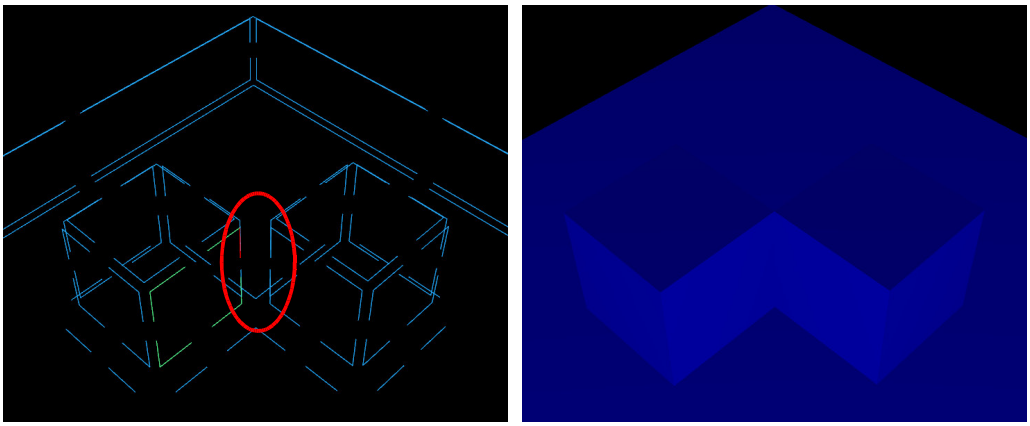


Abbildung 98: An der markierten Kante liegen vier Flächen an. Aus diesem Grund liegt an dieser Stelle eine Nicht-Mannigfaltigkeit vor. Rechts befindet sich die gerenderte Darstellung der Szene.

Vergleicht man diese Nicht-Mannigfaltigkeit mit der Definition der Quasi-Mannigfaltigkeit (siehe Abschnitt 3.3.1), so erkennt man, dass der dargestellte Fall nicht regulär adjazent ist, da er dieser Definition nicht entspricht.

Solche Nicht-Mannigfaltigkeiten treten immer dann auf, wenn Voxel einer Region nicht direkt 6-benachbart sind, aber trotzdem eine gemeinsame Kante haben. Im obigen Beispiel sind die Voxel über die darunterliegenden Voxel miteinander 6-verbunden.[68]

Wechselt man von einem Dart auf dieser Kante mit dem β_2-Orbit, so erreicht man auf dem selben Voxel die zweite Fläche, die an der Kante anliegt. Man kann aber über das Orbit $\beta_0 \circ \beta_2 \circ \beta_3$ alle Darts der Kante erreichen, da dieser Zusammenhalt über die umgebende Region gegeben ist. Wechselt man hier über das β_2-Orbit von einem Dart der Kante zu einem anderen, so erreicht man einen Dart der selben Kante auf dem anderen Voxel. In Abbildung 99 ist die Kontur der umgebenden Region dargestellt. Zudem sind die vier interessanten Flächen der innen liegenden Region noch einmal eingezeichnet. So sind alle Darts, die die nicht-mannigfaltige Kante bilden, abgebildet. An diesem Beispiel lässt sich die Behandlung von Nicht-Mannigfaltigkeiten gut demonstrieren.

Andere Möglichkeiten, in denen Nicht-Mannigfaltigkeiten vorkommen, treten auf, wenn zwei Voxel einer Region an einem Knoten zusammenliegen. Wiederum sind diese beiden Voxel nicht direkt 6-benachbart, sondern über andere Voxel 6-verbunden. Eine solche Konfiguration ist allerdings am betrachteten Knoten auch nicht quasi-mannigfaltig. Ein solcher 3-Zellkomplex kann in zwei 3-Zellkomplexe zerlegt werden, die nur durch eine 0-Zelle (den Knoten) miteinander verbunden sind. Aus diesem Grund sind auch nicht alle Darts, die an diesem Knoten anliegen, über das Orbit $\beta_1 \circ \beta_2 \circ \beta_3$ zu erreichen.

68 siehe Kapitel 3.2 zu Nachbarschaft und Verbundenheit von Voxeln

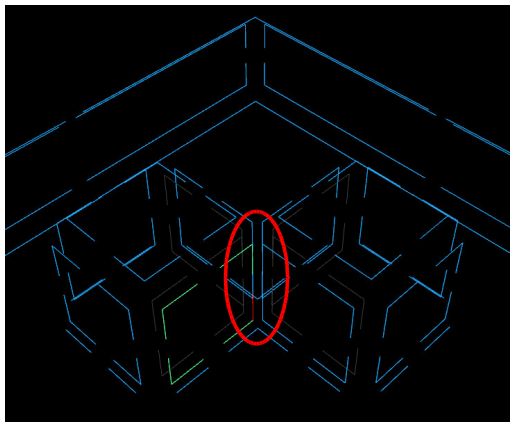


Abbildung 99: Die markierte Kante stellt ein $\beta_0 \circ \beta_2 \circ \beta_3$ -Orbit dar.

Liegen Voxel verschiedener Regionen auf die oben genannte Arten aneinander, also haben entweder nur eine gemeinsame Kante oder einen gemeinsamen Knoten, und sind zusätzlich von einer gemeinsamen Region umgeben, so ergibt sich ein analoges Verhalten zu den obigen Beispielen. Sind die Voxel kantenverbunden, so gibt es nur eine umgebende Hülle um die Regionen beider Voxel. Sind die Regionen der Voxel nur über einen Knoten verbunden, so besitzt jede Region eine eigene umgebende Hülle. Mit der umgebenden Hülle ist jeweils die innere Kontur der umgebenden Region gemeint (in Abbildung 100 dunkelblau dargestellt).

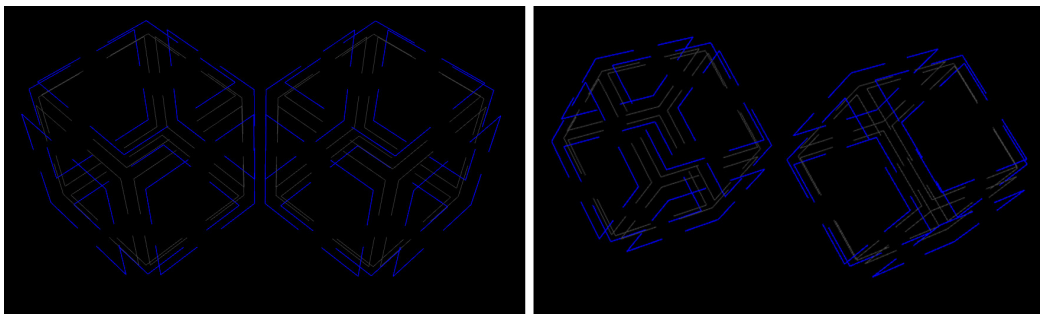


Abbildung 100: Im linken Bild stellen die dunkelblauen Darts eine gemeinsame Oberfläche dar. Im linken Bild werden durch die dunkelblauen Darts zwei getrennte Oberflächen repräsentiert.

Die hier beschriebenen Fälle können durchaus auch bei der Bearbeitung von Real-Welt-Volumen auftreten. Aus diesem Grund wurden sie an dieser Stelle beschrieben. Bei der weiteren Bearbeitung der 3-G-Map oder auch bei deren Beurteilung durch den Benutzer, muss eindeutig sein, warum die entsprechenden Konfigurationen von Darts und Orbits auftreten.

Ein Beispiel für den Umgang mit Nicht-Mannigfaltigkeiten, zeigt die folgende Abbildung 101.

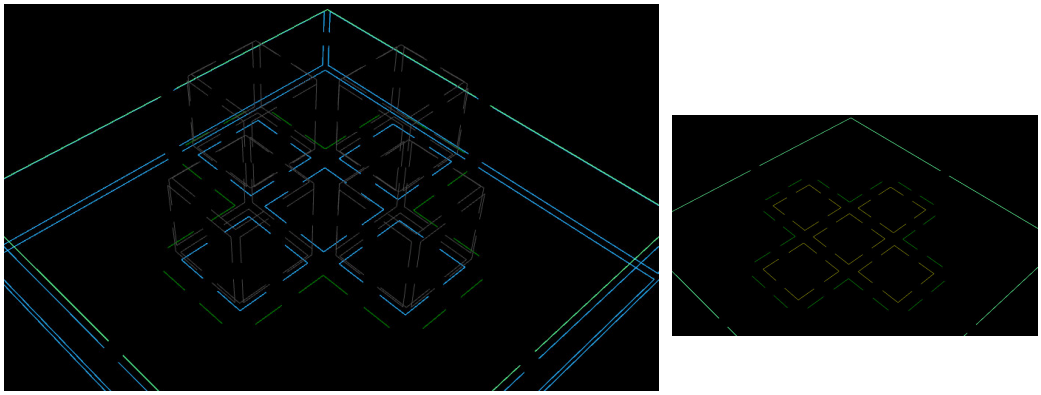


Abbildung 101: Links: Auf einer ebenen Region liegen vier Voxel unterschiedlicher Regionen (grau). Die innere Flächenkontur (dunkelgrün) verläuft um alle diese aufliegenden Regionen herum. Rechts: Die innere Flächenkontur besitzt einen DartSurfacePart mit insgesamt fünf DartFaces.

Wie in der Abbildung zu sehen ist, führen die vier unterschiedlichen Regionen (als graue Würfel dargestellt) zu einer inneren Kontur der grün dargestellten DartFace. Man beachte, dass es nicht vier innere Konturen jeweils um eine der Würfel-Regionen gibt, sondern nur eine um alle zusammen. Dieses Beispiel zeigt Parallelen zu dem in Kapitel 3.2 erwähnten Verbundenheits-Paradoxon. Beachtenswert ist die mittlere DartFace, die zum DartSurfacePart innerhalb der inneren Kontur gehört. Würde um jeden aufliegenden Voxel eine Kontur verlaufen, so gäbe es diese DartFace nicht. Der Bereich, den sie umschließt, würde dann zur grün hervorgehobenen DartFace gehören.

Die hier beschriebene Behandlung solcher Fälle ist aber sinnvoll. Denn wie man anhand der Abbildung 102 sehen kann, lässt sich die untere (blaue) Region leicht verändern, so dass der Ansatz mit den einzelnen Konturen keinen Sinn mehr ergibt. Für eine solche Veränderung wurde zwischen die vier unterschiedlichen Regionen ein weiterer Voxel der unteren (blauen) Region gelegt.

Die innere Kontur der oberen (grünen) DartFace ändert sich nicht. Nur der DartSurfacePart, der innerhalb der inneren Kontur liegt, hat sich durch die Hinzunahme eines weiteren Voxels zur unteren Region verändert. Dieses zweite Beispiel verdeutlicht, warum es auch im ersten Beispiel schon sinnvoll war, dass die innere Kontur um alle aufliegenden Regionen herumläuft. In diesem Beispiel ist es nicht mehr möglich, dass vier einzelne innere Konturen um die aufliegenden Regionen verlaufen, da dafür die Darts fehlen. Des Weiteren hat sich die Region topologisch betrachtet nicht verändert, sodass in beiden Fällen eine gleiche Behandlung stattfinden sollte.

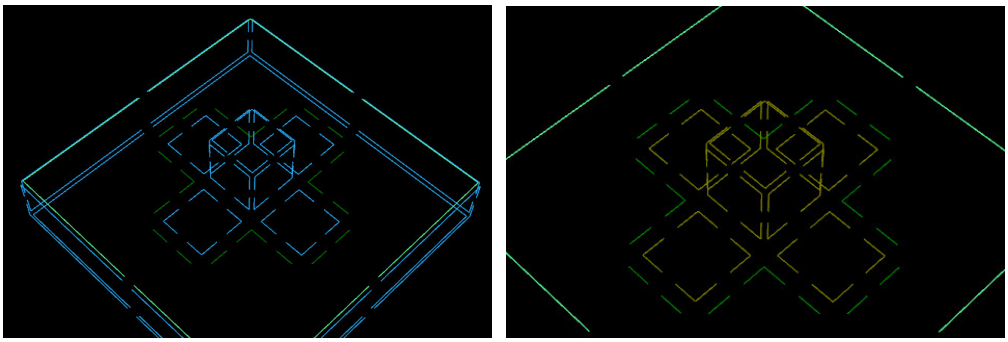


Abbildung 102: Die untere (blaue) Region wird gegenüber dem obigen Beispiel um einen „Voxel“ erweitert (rechts daneben der DARTSURFACEPART in der inneren Kontur).

Die obigen Beispiele haben gezeigt, dass Nicht-Mannigfaltigkeiten Auswirkungen auf eine 3-G-Map haben und somit auch die Strukturen der 3-XG-Map beeinflussen.

Durch diese Beispiele konnten also die Mächtigkeit und auch die Grenzen der G-Map-Repräsentation aufgezeigt werden.

10 Zusammenfassung und Ausblick

Dieses Kapitel resümiert diese Diplomarbeit und die Leistung, die in ihr erbracht wurde. Aus diesem Grund werden die Ergebnisse, die im Rahmen dieser Arbeit entstanden sind, zusammengefasst und im Bezug auf die definierten Ziele bewertet.

Im nächsten Unterkapitel wird zunächst noch einmal kurz auf die Ziele dieser Diplomarbeit eingegangen werden. Diese Ziele wurden bereits in der Einleitung dieser Arbeit vorgestellt. Wege, die zum Erreichen dieser Ziele führen, wurden ebenfalls erläutert. Nachdem in den letzten Kapiteln Lösungen entworfen und umgesetzt wurden, sollen diese jetzt kurz mit den Zielen in Bezug gesetzt werden. Des Weiteren wird darauf eingegangen werden, wie die Ziele erreicht werden konnten und was über die Ziele hinaus erreicht wurde.

Das letzte Unterkapitel dieser Diplomarbeit liefert einige interessante Anregungen der Erweiterung. Es soll insbesondere aufzeigen, wie vielfältig die in dieser Diplomarbeit vorgestellte Datenstruktur angewandt werden kann. Die Anregungen ergaben sich im Rahmen dieser Diplomarbeit und sollen nicht vorenthalten werden. Auch werden diese möglichen Erweiterungen nicht nur aufgezählt, es wird vielmehr gezeigt werden, wie sie sich in die bestehende Anwendung eingliedern lassen.

10.1 Ergebnisse dieser Arbeit

Diese Zusammenfassung entspricht einer kurzgefassten Bewertung des Geleisteten, da ausführlichere Bewertungen der einzelnen Teile bereits im Verlauf dieser Arbeit durchgeführt wurden. Dazu sei auf die jeweiligen Kapitel der entsprechende Themen verwiesen.

Das Ziel dieser Arbeit bestand darin, Segmentierungen beliebiger Volumendatensätze so zu repräsentieren, dass die enthaltenen Regionen hinsichtlich ihrer Geometrie und Topologie untersucht und bearbeitet werden können.

Auf dem Weg zu diesem Ziel ergaben sich weitere Teilziele, die zunächst erreicht werden mussten. Dies führte zu den folgenden Aufgaben:

1. Implementation einer Voxel-Nachbarschaft
2. Segmentierung beliebiger Volumen zur Erlangung einer ikonischen Regionenrepräsentation
3. Implementation der G-Map-Datenstrukturen
4. Entwicklung der 3-XG-Map zur Speicherung von in der G-Map nicht speicherbaren topologischen Informationen
5. Erstellung einer Benutzungsoberfläche, um das Arbeiten mit den umgesetzten Datenstrukturen übersichtlich und effizient zu ermöglichen

Die Implementation der benötigten Voxel-Nachbarschaft wurde in Kapitel 4 beschrieben. Es wurde allerdings nicht nur die 6er-Nachbarschaft, die für die weiteren Aufgaben dieser Arbeit ausgereicht hätte, sondern ebenfalls die 26er-Nachbarschaft in der VoxelNeighborhood umgesetzt. Da sich bei der Implementation der Nachbarschaften eng an der PixelNeighborhood orientiert wurde, welche universell einsetzbar ist, gilt dies auch für die VoxelNeighborhood. Ein Beispiel der Anwendbarkeit stellt der ebenfalls implementierte Wasserscheiden-Algorithmus dar. Die VoxelNeighborhood erfüllt alle an sie gestellten Anforderungen und eignet sich aufgrund ihrer Struktur ebenfalls für den Einsatz in anderen Bereichen der Bildverarbeitung. Durch eine Eingliederung in die VIGRA wird sie einem großen Nutzerkreis zugänglich gemacht werden.

Bei der Wahl des verwendeten Segmentierungsalgorithmus' stand die Zeitkomplexität im Vordergrund. Das Ergebnis sollte zudem einer ikonischen Regionenrepräsentation entsprechen, in der alle Voxel einer Region 6-verbunden sind. Zu diesem Zweck bot sich eine Umsetzung des in Kapitel 5 beschriebenen Union-Find-Wasserscheiden-Verfahrens an. Wie die Diskussion in Abschnitt 5.4.2 bereits ergeben hat, liefert der Algorithmus die geforderten Ergebnisse. Die Zeitkomplexität ist dabei sehr gering. Aus diesen Gründen war die Entscheidung zugunsten dieses Algorithmus' die richtige, zumal er nicht auf die Segmentierung einer bestimmten Klasse von Volumendaten beschränkt ist.

Nachdem ein Volumen in eine ikonische Regionenrepräsentation umgewandelt wurde, kann aus dieser eine G-Map erzeugt werden. Während dieser Diplomarbeit wurde der Datentyp der G-Map implementiert. Da bei diesem Schritt die Möglichkeit der Bearbeitung auch großer Volumendatensätze im Vordergrund stand, war die Effizienz des

Erstellungsvorgangs der GMAP3D besonders wichtig. Durch die Einführung der Volume-Map, die ein neuartiges Konzept der Erstellung ermöglicht, konnte eine Speicher sparende und schnelle Generierung der GMAP3D erreicht werden. Mit Hilfe der Volume-Map kann ein Segmentierungsergebnis schnell in eine GMAP3D überführt werden.

Wie bereits erwähnt, ist es nicht möglich, alle topologischen Informationen in einer G-Map explizit zu repräsentieren. Beispiele für Fälle, in denen topologische Informationen verloren gehen, wurden in Abschnitt 3.4.1 genannt. Aus diesem Grund wurde die Erweiterung der GMAP3D, die XGMAP3D, realisiert. Sie repräsentiert die verlorene topologische Information explizit und verknüpft diese auf elegante Weise mit den Darts der zugrunde liegenden GMAP3D. Der Aufwand der Erstellung der XGMAP3D konnte so weit minimiert werden, dass diese in akzeptabler Zeit ausgeführt wird.

Die erstellten Datenstrukturen sind allerdings relativ nutzlos, wenn sie nicht angemessen und übersichtlich präsentiert werden können. Aus diesem Grund wurde eine Anwendung erstellt (siehe Kapitel 8), die sowohl eine grafische, als auch eine baumartige Ansicht auf die Strukturen der XGMAP3D bietet. Durch Interaktion beider Darstellungsarten wird die Übersicht während der Navigation durch die Strukturen verbessert. Auch die farbigen Hervorhebungen der einzelnen Elemente erleichtern das Verständnis für die Strukturen. Des Weiteren konnten Operationen in die Anwendung integriert werden. Sie ermöglichen die Bearbeitung der Segmentierungsergebnisse, wie in Abschnitt 7.5.3 beschrieben. Auch topologische und geometrische Informationen können, wie in Kapitel 9.3 erläutert, über die Anwendung ausgegeben werden. Diese Informationen erlauben die objektive Beurteilung des zugrunde liegenden Segmentierungsergebnisses. Die topologische Information der Nachbarschaft ist explizit in der XGMAP3D enthalten.

Die Erfüllung der oben genannten Teilziele führte zum Erreichen des Ziels, das für diese Arbeit gesteckt wurde. Durch die zahlreich durchgeführten Optimierungen konnte sogar erreicht werden, dass eine Bearbeitung von Real-Welt-Volumen in kurzer Zeit möglich ist. Im Vergleich zu den in der verfügbaren Literatur angegebenen Beispielen, ist die Zeitkomplexität der in dieser Arbeit vorgestellten Methode geringer.

Zusätzlich zum eigentlichen Ziel dieser Diplomarbeit wurden einige Operationen implementiert, die die Durchführung von Untersuchungen und Bearbeitungen der Geometrie und Topologie erlauben. Diese Operationen verwenden die Struktur der XGMAP3D und stellen erste Beispiele der Verwendung der erstellten Datenstrukturen dar. Wie diese weiter ausgebaut werden kann, folgt im nächsten Unterkapitel.

10.2 Möglichkeiten der Erweiterung

Während des Arbeitens mit den implementierten Datenstrukturen ergaben sich noch einige weitere Ideen, die im Zuge einer zukünftigen Erweiterung umgesetzt werden könnten. Nachdem die Basis für aufbauende Operationen in dieser Arbeit geschaffen wurde, steht nun der Weg offen, die Anwendung zu erweitern und eventuell zu spezialisieren.

Im Folgenden werden einige Anregungen aufgezählt:

- Einbinden weiterer Segmentierungsverfahren

  Je nach zu verarbeitendem Volumendatensatz kann es sinnvoll sein, spezielle Segmentierungsverfahren anzuwenden, die auf Volumendatensätze der jeweiligen Art spezialisiert sind. Stehen verschiedene Segmentierungsverfahren zur Wahl, ist auch ein Vergleich der Ergebnisse bezüglich der Geometrie und Topologie möglich.

- Erstellung der XGMap3d aus Subvoxel-Repräsentationen

  Eine ikonische Regionenrepräsentation kann in unterschiedlicher Weise vorliegen. Neben der hier genannten voxelbasierten Darstellung gibt es beispielsweise noch die der Marching-Cubes. Zur Erstellung der XGMap3d müsste dazu ein neuer XGMap3dCreator implementiert werden.

- Erweiterung der Euler-Operationen

  Die bereits vorhandenen Verschmelzungs- und Löschoperationen können durch weitere ergänzt werden. Eine weitere sinnvolle Operation wäre die Zerteilung von Regionen. Auch die Möglichkeit, einen Schritt rückgängig machen zu können, wäre eine denkbare Erweiterung.

- Hinzufügen von weiteren Auswahlkriterien

  Bisher können Regionen mittels ihrer durchschnittlichen Farbe und Flächen anhand ihres mittleren Gradienten automatisch ausgewählt werden. Weitere auswählende Kriterien könnten von Nutzen sein.

- Ermittlung weiterer topologischer Eigenschaften

  Zur ausführlicheren topologischen Beschreibung kann die Ermittlung der Homologiegruppen anhand von Betti-Zahlen dienlich sein.

- Weitere Renderoptionen

  Um die Darstellung noch übersichtlicher zu machen, könnten Oberflächen unterschiedlicher Regionen in verschiedenen Farben gerendert werden. Damit bei der gerenderten Darstellung die Darts und auch im Hintergrund liegende Regionen sichtbar sind, wäre ein Alpha-Blending möglich. Die Transparenz der Oberflächen kann beispielsweise in Abhängigkeit des Gradienten gewählt werden.

- Weitere Navigationsmöglichkeiten

  Mit Hilfe eines dreidimensionalen haptischen Eingabegeräts (zum Beispiel PHANTOM®) kann die direkte Auswahl von einzelnen Elementen im OpenGL-Fenster erfolgen. Steht nur eine normale Maus zur Verfügung, wäre die Auswahl aus einer Liste von Elementen, die auf einem Strahl ausgehend von der Position des Mauszeigers liegen, eine weitere Möglichkeit.

Die entwickelte Anwendung stellt eine gute Basis für die genannten Erweiterungen dar. Daher wäre es sehr wünschenswert, wenn sie innerhalb weiterführender Arbeiten umgesetzt werden könnten.

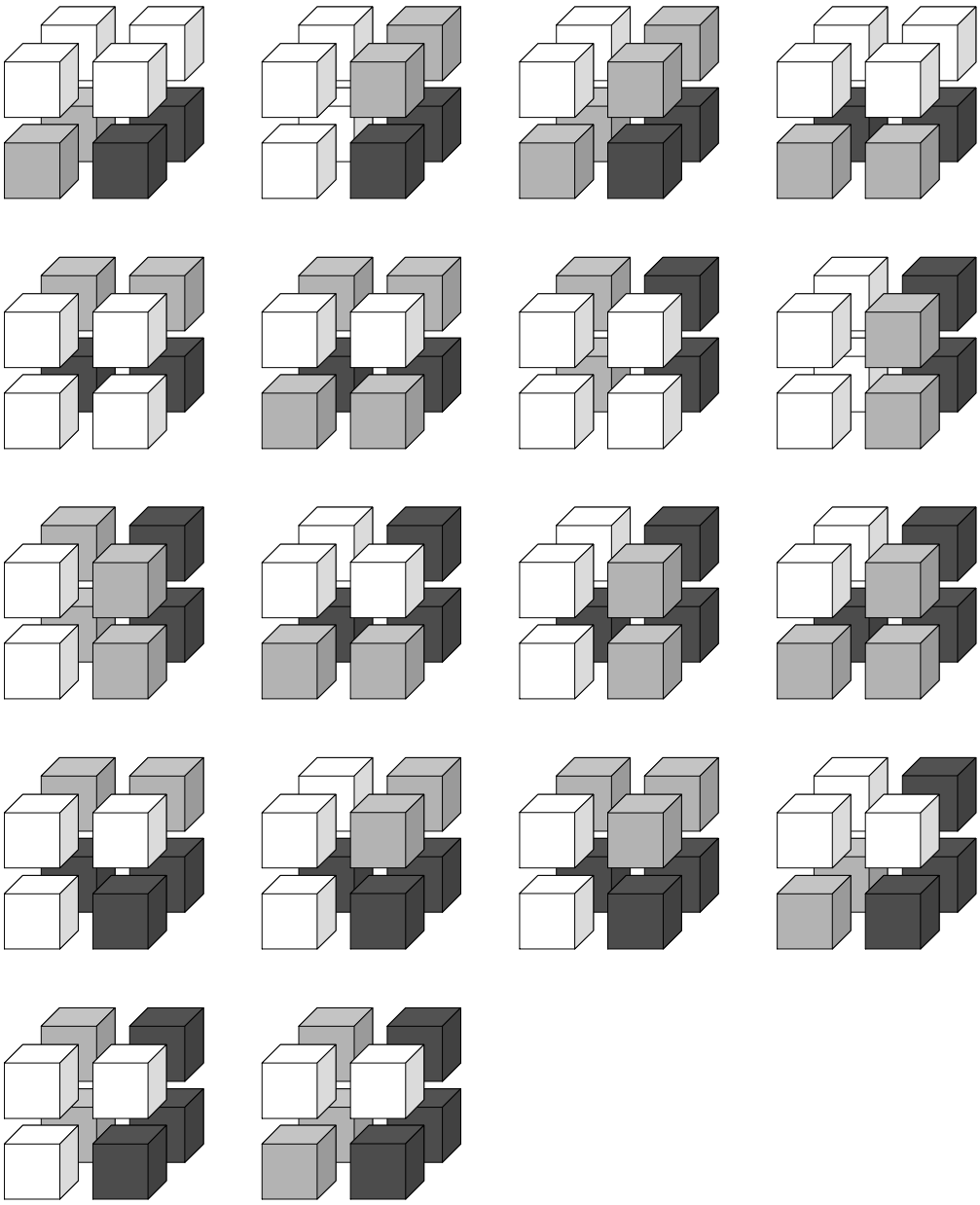
Anhang

A Tabelle der Richtungen des NeighborCode3DTwentySix

Richtungen im NeighborCode3DTwentySix		
Richtungen	in Bit-Form	Differenzvektor vom Zentrum
InFrontNorthWest	1	(-1, -1, -1)
InFrontNorth	2	(0, -1, -1)
InFrontNorthEast	3	(1, -1, -1)
InFrontWest	4	(-1, 0, -1)
InFront	5	(0, 0, -1)
InFrontEast	6	(1, 0, -1)
InFrontSouthWest	7	(-1, 1, -1)
InFrontSouth	8	(0, 1, -1)
InFrontSouthEast	9	(1, 1, -1)
NorthWest	10	(-1, -1, 0)
North	11	(0, -1, 0)
NorthEast	12	(1, -1, 0)
West	13	(-1, 0, 0)
East	14	(1, 0, 0)
SouthWest	15	(-1, 1, 0)
South	16	(0, 1, 0)
SouthEast	17	(1, 1, 0)
BehindNorthWest	18	(-1, -1, 1)
BehindNorth	19	(0, -1, 1)
BehindNorthEast	20	(1, -1, 1)
BehindWest	21	(-1, 0, 1)
Behind	22	(0, 0, 1)
BehindEast	23	(1, 0, 1)
BehindSouthWest	24	(-1, 1, 1)
BehindSouth	25	(0, 1, 1)
BehindSouthEast	26	(1, 1, 1)
CausalFirst = InFrontNorthWest		
CausalLast = West		
Error = -1		
AntiCausalFirst = BehindSouthEast		
AntiCausalLast = East		
OppositeDirPrefix = -1		
OppositeOffset = 25		

B Precodes einer Level-2-Karte

In diesem Teil des Anhangs werden alle Precodes aufgezeigt, die zum Erstellen einer Level-2-Karte notwendig sind. Sie wurden im Rahmen dieser Arbeit zusammengestellt, da in der vorhandenen Literatur lediglich einige Precodes, nicht aber alle 18 abgebildet sind.



C Kurzanleitung der Anwendung

In diesem Abschnitt wird eine kurze Anleitung des „OpenGL XGMap3d Viewer“ gegeben. Diese kann ebenfalls angerufen werden, wenn man im Hauptfenster der Anwendung die Taste F1 drückt oder im Menü „Help“ den Menüpunkt „Quick reference guide“ auswählt.

Inhalt der Kurzanleitung

1. Laden eines Volumens und Erstellen einer XGMap
2. Bedeutung der verschiedenen Farben
3. Drehen und Verschieben der angezeigten XGMap
4. Auswahl des Drehursprungs
5. Hineinzoomen in die XGMap
6. Drehung, Verschiebung und Zoom rückgängig machen
7. Auswahl von Elementen der XGMap
8. Wechseln zwischen den Elementen der XGMap
9. Anpassung der angezeigten aktiven Elemente
10. Anpassung des Anzeigemodus
11. Oberflächen-Rendering der einzelnen Regionen
12. Bearbeiten der angezeigten XGMap
13. Speichern der geänderten XGMap
14. Anzeige von topologischen Informationen der XGMap-Elemente
15. Auswahl anhand der mittleren Farbwerte der Volumen
16. Programmdokumentation in neuem Fenster öffnen

1. Laden eines Volumens und Erstellen einer XGMap

Geladen werden können alle Volumendaten, deren einzelne Schichten als einzelne Bilder gespeichert sind. Die Schichtenbilder müssen fortlaufend nummeriert sein (Beispiel bild_xxx.png, wobei xxx die Bildnummer angiebt).

Gelabelte Volumendaten (jede Region/Volumen wird durch eine einzigartige Farbe repräsentiert) können im Menü „File“ mit „Open labeled Volume“ geöffnet werden. Es ist lediglich ein einziges Bild der Bilderreihe abzugeben.

Nicht gelabelte Volumendaten können im Menü „File“ mit „Open Volume“ geöffnet werden. Zuvor kann es sinnvoll sein, den Wert des Gauß-Filters anzupassen (unter „Settings -> Options“).

Zudem gibt es die Möglichkeit Bilder per Drag-and-Drop direkt in den schwarzen Anzeigebereich der XGMap-Anzeige hineinzuziehen. Dabei kann der Benutzer entscheiden, ob das Volumen als gelabeltes oder ungelabeltes Bild angesehen werden soll.

2. Bedeutung der verschiedenen Farben

rot	Aktiver, also ausgewählter Dart
hellgrün	Äußere Kontur der aktiven Fläche, also die Fläche mit dem ausgewählten Dart
dunkelgrün	Innere Konturen der aktiven Fläche
hellblau	Äußere Kontur der aktiven Region
dunkelblau	Innere Konturen der aktiven Region
goldfarben	Aktives DartSurfacePart
grau	Inaktive, aber im ListView mit Haken versehene Darts

3. Drehen und Verschieben der angezeigten XGMap

Die angezeigte XGMap im OpenGL-Fenster lässt sich mit Hilfe der Maus drehen und verschieben. Zum Drehen bei gedrückter linker Maustaste den Mauszeiger über das OpenGL-Fenster bewegen.

Zum Verschieben ist zusätzlich zur linken Maustaste noch die Umschalttaste zu halten, um die XGMap nach links, rechts, oben und unten zu verschieben. Eine Verschiebung von vorne nach hinten ist mit der Alttaste möglich.

4. Auswahl des Drehursprungs

Der Drehursprung lässt sich im Menü „Settings – Rotate around“ einstellen. Der Drehursprung wird zentriert, soweit vorher noch nicht verschoben.

Die Option „Current Dart“ legt den Drehursprung auf den Startpunkt des aktiven Darts.

Die Option „Volume Center“ legt den Drehursprung auf den Mittelpunkt der XGMap.

Alle anderen Optionen legen den Drehursprung auf den Mittelpunkt der jeweiligen Bounding-Box.

5. Hineinzoomen in die XGMap

Ein Hinein- und Hinauszoomen der angezeigten XGMap ist mit dem Mausrad möglich. Das OpenGL-Fenster muss aktiv sein.

6. Drehung, Verschiebung und Zoom rückgängig machen

Durch Drücken der rechten Maustaste werden alle Veränderungen rückgängig gemacht. Das OpenGL-Fenster muss aktiv sein.

7. Auswahl von Elementen der XGMap

Die Elemente können im rechts angezeigten ListView direkt angewählt werden, indem sie mit der Maus angeklickt werden. Die Anzeige wird automatisch aktualisiert.

Werden Elemente mit einem Häkchen versehen, so werden sie auch dann angezeigt, wenn sie nicht mehr aktiv sind. Die Auswahl über ein Intervall kann über die entsprechenden Boxen unten rechts eingestellt werden. Steht die Zeichenfolge >> vor einem Volumeneintrag im ListView so bedeutet dies, dass einige Elemente des Volumens markiert sind, nicht aber alle Elemente. Nach einem Aufklappen sieht man die markierten bzw. nicht markierten Elemente.

8. Wechseln zwischen den Elementen der XGMap

Entweder Elemente direkt anwählen, siehe Auswahl von Elementen der XGMap, oder Darts mit Hilfe der mittleren Maustaste wechseln im OpenGL-Fenster.

mittlere Maustaste	Wechseln der Kante
mittlere Maustaste + Umschalttaste	Wechseln der Fläche
mittlere Maustaste + Alttaste	Wechseln der Region/Volumen

9. Anpassung der angezeigten aktiven Elemente

Es müssen nicht immer alle aktiven Elemente mit den entsprechenden Farben angezeigt werden. Im Menü „Settings – Highlight active Elements“ können die Strukturen an- oder abgewählt werden, die angezeigt werden sollen.

10. Anpassung des Anzeigemodus

Um die Übersichtlichkeit der dargestellten Elemente einer XGMap zu erhöhen, gerade wenn sehr viele Elemente gleichzeitig oder sehr große angezeigt werden, können die Darts auf unterschiedliche Weise gezeichnet werden. Im Menü „Settings – Dart rendering mode“ kann die gewünschte Option gewählt werden.

Zudem kann ausgewählt werden, ob der Rahmen des geladenen Volumens mit angezeigt werden soll.

11. Oberflächen-Rendering der einzelnen Regionen

Es gibt verschiedene Möglichkeiten, die Schalen der einzelnen Regionen zu rendern. Die erste Möglichkeit besteht darin, die Oberfläche der im ListView aktuell ausgewählten Region zu rendern.

Meist ist es jedoch sinnvoll, nicht eine einzelne Region, sondern eine Vielzahl ausgewählter Regionen zu rendern. Zu diesem Zweck gibt es einen weiteren Oberflächen-Rendering-Modus, der alle ausgewählten Volumenoberflächen rendert.

Besitzt eine Region einen Einschluss, so wird dieser im allgemeinen Fall durch die äußeren Oberflächen verdeckt. Um sich dennoch ein Bild von den Gegebenheiten im Inneren zu machen, existiert eine weitere Möglichkeit der Ansicht, die wahlweise an- oder ausgeschaltet werden kann: „Look inside“. Sie entfernt die in Blickrichtung äußeren Teile der Oberfläche und ermöglicht somit einen Blick ins Innere des Volumens.

12. Bearbeiten der angezeigten XGMap

Zum Bearbeiten der aktuellen XGMap stehen folgende Operationen bereit:

- Löschen der aktuell ausgewählten Face
- Löschen des aktuell ausgewählten Einschlusses
- Löschen aller im Listview selektierten Faces

Das Löschen von Faces dauert recht lange, das diese durch ein Umlabeln des Label-Volumens und anschließender Neuerstellung der kompletten XGMap realisiert sind. Es bietet sich daher der zweitgenannte obige Punkt an, da dort für alle zu löschenden Faces die XGMap nur einmal neu erstellt werden muss.

Das Löschen eines Einschlusses geschieht direkt auf den Datenstrukturen, und erfordert somit keine Neuerstellung der XGMap. Es entfernt eine isolierte Region komplett.

Des Weiteren gibt es noch folgende Möglichkeiten um Elemente der XGMap auszuwählen:

- Alles auswählen
- Auswahl aufheben
- Auswahl umkehren
- Faces mit niedrigem mittleren Gradienten auswählen

Die ersten drei Punkten sind selbst erklärend. Der letzte Punkt wählt alle die Faces aus, die unter einem mittleren Gradienten liegen. Der Schwellwert hierfür kann unter dem Menüpunkt Settings -> Options gesetzt werden.

13. Speichern der geänderten XGMap

Die manipulierte XGMap kann nicht direkt abgespeichert werden, es kann aber das veränderte zugrunde liegende Label-Volumen abgespeichert werden.

Diese Speicherung kann im Menü File->Save label-volume aufgerufen werden. Das Volumen wird dabei als Stapel von Schichtenbildern abgelegt.

14. Anzeige von topologischen Informationen der XGMap-Elemente

Wird mit der rechten Maustaste auf ein DartVolume-Objekt in dem ListView geklickt, so erscheint zu diesem ein Fenster, welches topologische Informationen dieses Volumens und aller Schalen anzeigt.

Klickt man hingegen auf eine einzelne Schale (Volumecontour-Element) im ListView, so werden die topologischen Informationen dieser einen Schale einzeln präsentiert.

15. Auswahl anhand der mittleren Farbwerte der Volumen

Die beiden Auswahlboxen unterhalb des ListViews können dazu verwendet werden, um eine Art Thresholding der Regionen durchzuführen. Dazu wählt man mit ihrer Hilfe ein Intervall aus, in dem die anzuzeigenden DartVolumes sich befinden sollen. Mit jeder Veränderung der Werte werden die Regionen, die innerhalb dieses Intervall liegen, grau dargestellt und im Listview mit einem Haken versehen.

D Glossar

3-XG-Map

Der Begriff 3-XG-Map beschreibt die formal definierte dreidimensionale erweiterte generalisierte Karte (siehe Kapitel 3.4.5).

ADT

ADT steht abkürzend für abstrakter Datentyp.

Anker

Ein Einstiegspunkt in ein Orbit, anhand dessen die Gleichheit zweier Orbits leicht feststellbar ist.

Bild

Unter einem Bild versteht man, soweit nicht anders angegeben, ein zwei- oder dreidimensionales digitales Bild.

Bildpunkt

Bildpunkt bezeichnet einen bestimmten Punkt in einem *Bild*.

Bounding Box

Die Bounding Box einer Region besteht aus dem minimalen nicht rotierten Quader, in den die Region eingebettet werden kann, und wird meist durch zwei Punkte angegeben.

DAG

DAG ist die Bezeichnung für einen gerichteten azyklischen Graphen.

Dart

Ein Dart ist das atomare Element einer kombinatorischen Karte.

Dartplatzhalter

Ein gesetztes Bit in der Volume-Map heißt Dartplatzhalter.

G-Map

Der Begriff G-Map beschreibt die formal definierte generalisierte Karte (siehe Abschnitt 3.4.3). *n*-G-Map bezeichnet eine G-Map der Dimension *n*.

GMAP3D

Der abstrakte Datentyp der Implementation einer 3-G-Map heißt in dieser Arbeit GMAP3D, der einer n-G-Map wird auch GMAP<N> oder abkürzend GMAP genannt.

GUI

Die Abkürzung für grafische Benutzungsoberfläche ist GUI.

Iterieren

Ein sequentielles Ablaufen einer Datenstruktur wird Iterieren genannt.

ListView

Ein Anzeigeelement, das eine baumartig untergliederte Liste anzeigt, heißt ListView.

Nachbarschaftskodierung

Die Kapselung einer Nachbarschaft wird auch als Nachbarschaftskodierung bezeichnet.

OpenGL

Eine Möglichkeit, dreidimensionale Szenen mit Hilfe von C++ in einer Anwendung zu visualisieren, ist die OpenGL-Bibliothek.

Pixel

Ein Bildpunkt eines zweidimensionalen Bildes wird Pixel genannt.

Plateau

Ein Plateau eines Bildpunktes in einer Nachbarschaft von Bildpunkten liegt dann vor, wenn dieser den gleichen Farbwert wie alle seine Nachbarn besitzt.

Precodes

Die Precodes stellen eine Repräsentation von Hash-Tabellen dar, die die Erstellung von kombinatorischen Karten vereinfachen und beschleunigen.

Qt

Qt ist eine Bibliothek zur Erstellung von grafischen Benutzungsoberflächen für C++, unabhängig vom Betriebssystem.

Real-Welt-Daten, Real-Welt-Bilder, Real-Welt-Volumen

Bilder, die reale Aufzeichnungen von Objekten sind, werden auch Real-Welt-Bilder genannt.

Rendering

Eine Projektion einer dreidimensionalen Szene auf eine Ebene wird als Rendering bezeichnet.

Richtungsvolumen

Das Volumen, das nach dem ersten Schritt des implementierten Union-Find-Wasserscheiden-Verfahrens entsteht, wird Richtungsvolumen genannt, da es die Richtungen in Bit-kodierter Form speichert.

Segmentierung

Der Prozess der Zerlegung eines *Bild*es in mehrere Regionen wird Segmentierung genannt.

Synthetische Daten

Daten, die künstlich hergestellt wurden, werden synthetische Daten genannt.

Traversieren

Ein eventuell ungeordnetes Ablaufen einer Datenstruktur wird auch als das Traversieren dieser bezeichnet.

Volume-Map

Ein Volumen heißt Volume-Map, wenn es in der Repräsentation von Definition 6.2.1 vorliegt.

Volumen, Volumendatensatz

Ein dreidimensionales digitales Bild heißt Volumen oder Volumendatensatz.

Voxel

Voxel wird als Bezeichner eines Volumen-Bildpunktes verwendet.

XGMAP3D

Der abstrakte Datentyp der Implementierung einer 3-XG-Map heißt in dieser Arbeit XGMAP3D.

E Literaturverzeichnis

[BD05] Braquelaire, A. and Domenger, J.-P. 2005. Representing and Segmenting 2D Images by Means of Planar Maps with Discrete Embeddings: From Model to Applications. In *Proceedings of 5th IAPR International Workshop on Graph-Based Representations in Pattern Recognition*, 2005, Poitiers, France, April 11-13, 2005, Luc Brun, Mario Vento (Eds.): Lecture Notes in Computer Science 3434, Springer 2005, 92-121

[BDDV03] Braquelaire, A. and Damiand, G. and Domenger, J.-P. and Vidil, F. 2003. Comparison and Convergence of Two Topological Models for 3D Image Segmenation. In *Proceedings of the 4th IAPR International Workshop on Graph Based Representations in Pattern Recognition*, 2003, York, UK, June 30 - July 2, 2003, Edwin R. Hancock and Mario Vento (Eds.): Lecture Notes in Computer Science 2726, Springer 2003, 59-70

[BDF00] Bertrand, Y., Damiand, G., and Fiorio, C. 2000. Topological Encoding of 3D Segmented Images. In *Proceedings of the 9th international Conference on Discrete Geometry For Computer Imagery* (December 13 - 15, 2000). G. Borgefors, I. Nyström, and G. S. Baja, Eds. Lecture Notes In Computer Science, vol. 1953. Springer-Verlag, London, 311-324.

[BFP99] Yves Bertrand , Christophe Fiorio , Yann Pennaneach, Border Map: A Topological Representation for nD Image Analysis, In: *Proceedings of the 8th International Conference on Discrete Geometry for Computer Imagery*, p.242-257, March 17-19, 1999

[Big89] Norman L. Biggs. Discrete Mathematics. Oxford University Press, Great Clarendon Street, Oxford, 1989. revised edition, original from 1985.

[BK99] Brun, L. and Kropatsch, W. *Dual Contraction of Combinatorial Maps*. Technical Report, Technische Universität Wien, 1999

[BK00] Brun, L. and Kropatsch, W. Irregular Pyramids with Combinatorial Maps In: Ferri et al. (Eds.): *SSPR&SPR 2000, LNCS 1876*, pp. 256, 2000. Springer-Verlag Berlin Heidelberg 2000

[BL79] Beucher, S., and Lantuéjoul, C. Use of watersheds in contour detection. In *Proc. International Workshop on Image Processing, Real-Time Edge and Motion Detection/Estimation,* Rennes, September 1979.

[Coh95] Cohn, A. G. *A Hierarchical Representation of Qualitative Shape based on Connection and Convexity*. International Conference on Spatial Information Theory COSIT-95 Semmering (Austria) September 21-23, 1995

[Cor75] Cori, R. *Un Code pour les Graphes Planaires et ses Applications*. Astrisque, Vol 27, Soc. Math. de France, Paris, 1975.

[DBF04] Damiand, G., Bertrand, Y., and Fiorio, C. 2004. Topological model for two-dimensional image representation: definition and optimal extraction algorithm. *Comput. Vis. Image Underst.* 93, 2 (Feb. 2004), 111-154.

[Die06] Reinhard Diestel. *Graphentheorie*. Springer-Verlag, Berlin, Heidelberg, New York. 2006. ISBN 3-540-21391-0

[DL02] G. Damiand, P. Lienhardt, Removal and contraction for n-dimensional generalized maps, In: H. Wildenauer, W. Kropatsch (Eds.): *Proceedings of the Computer Vision Winter Workshop*, Bad Ausse, Austria, 2002, pp. 208-221.

[DR02] Damiand, G. and Resch, P. Topological map based algorithms for 3d image segmentation. In: *Proceedings of 10th Discrete Geometry for Computer Imagery*, Bordeaux, France, April 3-5, 2002, LNCS 2301, pp. 220-231.

[DP88] Dörfler, W., Pescheck, W. Einführung in die Mathematik für Informatiker. 1988 Carl Hanser Verlag München Wien. ISBN 3-446-15112-5

[FMMP02] De Floriani, L., Mesmoudi, M. M., Morando, F. And Puppo, E. Non-manifold Decomposition in Arbitrary Dimensions. In A. Braquelaire, J.-O. Lachaud, and A. Vialard (Eds.): *DGCI 2002*, LNCS 2301, pp. 69–80, 2002. Springer-Verlag Berlin Heidelberg 2002

[GDL05] Grasset-Simon, C. and Damiand, G. and Lienhardt, P. 2005: Pyramids of n-Dimensional Generalized Maps. In *Proceedings of 5th IAPR International Workshop on Graph-Based Representations in Pattern Recognition*, 2005, Poitiers, France, April 11-13, 2005, Luc Brun, Mario Vento (Eds.): Lecture Notes in Computer Science 3434, Springer 2005, 142-152

[GW92] Gonzalez, R. C., Woods, R.E. 1992. *Digital Image Processing*. Addison-Wesley. ISBN 0-201-50803-6

[Jäh97] Jähne, B. 1997. Digitale Bildverarbeitung. 4. völlig neubearbeitete Auflage.Springer-Verlag, Berlin Heidelberg New York. ISBN 3-540-61379-X

[Köt00] Ullrich Köthe. *Generische Programmierung für die Bildverarbeitung*. PhD thesis, Dept. of Computer Science, University of Hamburg, 2000. (available via books-on-demand: `http://www.libri.de/`).

[Köt01] Ullrich Köthe. XPMaps and topological segmentation - a unified approach to finite topologies in the plane. Technical Report FBI-HH-M-308/01, Dept. of Computer Science, University of Hamburg, 2001. long version of DGCI paper with proofs, 14 pages.

[Köt02] Ullrich Köthe. XPMaps and topological segmentation - a unified approach to finite topologies in the plane. In A. Braquelaire, J.-O. Lachaud, and A. Vialard, editors, *10th International Conference on Discrete Geometry for Computer Imagery (DGCI 2002)*, volume 2310 of *Lecture Notes in Computer Science*, pages 22–33, Berlin, 2002. Springer.

[Kov89] Kovalevsky, V. 1989. Finite Topology as Applied to Image Analysis. In: *Computer Vision, Graphics and Image Processing*, Vol. 46 (1989) pp. 141-161

[Kov99] Kovalevsky, V. 1999. A Topological Method of Surface Representation. In *Proceedings of the 8th international Conference on Discrete Geometry For Computer Imagery* (March 17 - 19, 1999). G. Bertrand, M. Couprie, and L. Perroton, Eds. Lecture Notes In Computer Science, vol. 1568. Springer-Verlag, London, 118-135.

[Kov03] Kovalevsky, V. 2003. Multidimensional cell lists for investigating 3-manifolds. *Discrete Appl. Math.* 125, 1 (Jan. 2003), 25-43.

[Kov04] Kovalevsky, V. 1999. Algorithms in Digital Geometry Based on Cellular Topology. In *Proceedings of the 10th International Workshop on Combinatorial Image Analysis, IWCIA 2004, Auckland, New Zealand,* (December 1-3, 2004). Klette, Reinhard; Zunic, Jovisa Eds. Lecture Notes In Computer Science, vol. 3322 . Springer-Verlag, London, 366-393.

[Lan78] Lantuéjoul, C. *La squelettisation et son application aux mesures topologiques des mosa¨ıques polycristallines*. PhD thesis, Ecole des Mines, Paris, 1978.

[LFB99] Lienhardt, P. and Fuchs, L. and Bertrand, Y. 1999. A course in Topology-based Geometric Modeling. In *Eurographics Workshop on Graphics and Visualization Education*, 39-44.

[Lie89] Lienhardt, P. 1989. Subdivisions of *n*-dimensional spaces and *n*-dimensional generalized maps. In *Proceedings of the Fifth Annual Symposium on Computational Geometry* (Saarbruchen, West Germany, June 05 - 07, 1989). SCG '89. ACM Press, New York, NY, 228-236.

[LM99] Lévy, B. and Mallet, J.-L. *Cellular Modeling in Arbitrary Dimension using Generalized Maps*, 1999

[Mat06] Weisstein, Eric W. "Homeomorphism." In: MathWorld - A Wolfram Web Resource. URL: http://mathworld.wolfram.com/Homeomorphism.html (Abgerufen: 6. Dezember 2006, 12:58 UTC)

[Mei03] Meine, H. 2003. *XPMap-Based Irregular Pyramids for Image Segmentation* Diplomarbeit am Fachbereich Informatik, Universität Hamburg.

[Meij05] Meijster, A. *Efficient Sequential and Parallel Algorithms for Morphological Image Processing*. Thesis Rijksuniversiteit Groningen. Printed by Universal Press (www.universalpress.nl). ISBN 90-367-1977-1

[NP01] Nikolaidis, N., Pitas, I. 3-D Image Processing Algorithms. 2001. John Wiley & Sons, Inc. ISBN 0-471-37736-8

[PGBM03] Prat, S. and Gioia, P. and Bertrand, Y. and Meneveaux, D. Connectivity Compression in Arbitrary Dimension. In *Pacific Graphics 2005, Macao, China*. Journal The Visual Computer, 01.09.2005, vol. 21, no. 8, Springer-Verlag, Berlin Heidelberg, 876-885.

[RK82] Rosenfeld, A., Kak, A.C.: *Digital picture Processing*. 2nd ed. Vol. I and II, Academic Press, Orlando, 1982.

[RP66] Rosenfeld, A., and Pfaltz, J. L. *Sequential operations in digital picture processing*. J. Ass. Comp. Mach. 13 (1966), 471–494.

[SDL05] Simon, C. and Damiand, G. and Lienhardt, P. nD Generalized Map Pyramids: Three Equivalent Representations. In Research report n° 2005-03, SIC, Université de Poitiers

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

[SWND06] Shreider, D.,Woo, M., Neider, J. and Davis, T. 2006 *OpenGL Programming Guide: the Official Guide to Learning OpenGL, Version 2*. 5th. Addison-Wesley. ISBN 0-321-33575-2

[Tar75] Tarjan, R. E. Efficiency of a good but not linear set union algorithm. In *Journal of the ACM*, 22(2):215-225, April 1975.

[Tar83] Tarjan, R. E. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series In Applied Mathematics, 1983, ISBN:0-89871-187-8

[TBB01] Sylvain Thery, Dominique Bechmann, Yves Bertrand: N-Dimensional Gregory-Bezier for N-Dimensional Cellular Complexes. In *WSCG (Short Papers)* 2001: 16-23.

[Vig06] *VIGRA Reference Manual*. URL: http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/vigra/ (Abgerufen: 6. Dezember 2006, 13:27 UTC)

[Vig06a] *Crack-Edge Beschreibung*. In VIGRA Reference Manual. URL: http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/vigra/CrackEdgeImage.html (Abgerufen: 6. Dezember 2006, 13:28 UTC)

[Vig06b] *Pixelneighborhood*. In VIGRA Reference Manual. URL: http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/vigra/group__PixelNeighborhood.html (Abgerufen: 10. Dezember 2006, 13:18 UTC)

[Vig06c] *MultiIterator-Konzept*. In VIGRA Reference Manual. URL: http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/vigra/classvigra_1_1MultiIterator.html (Abgerufen: 10. Dezember 2006, 13:46 UTC)

[Vig06d] *TinyVector*. In VIGRA Reference Manual. URL: http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/vigra/classvigra_1_1TinyVector.html (Abgerufen: 10. Dezember 2006, 20:25 UTC)

[Vig06e] *MultiArray*. In VIGRA Reference Manual. URL: http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/vigra/classvigra_1_1MultiArray.html (Abgerufen: 10. Dezember 2006, 20:35 UTC)

[VS91] Vincent, L. and Soille, P. 1991. Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations. *IEEE Trans. Pattern Anal. Mach. Intell.* 13, 6 (Jun. 1991), 583-598.

[Wik06a] Artikel *Kleinsche Flasche.* In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 4. August 2006, 10:04 UTC. URL: http://de.wikipedia.org/w/index.php?title=Kleinsche_Flasche&oldid=19777430 (Abgerufen: 22. August 2006, 14:14 UTC)

[Wik06b] Artikel *Möbiusband.* In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 1. August 2006, 20:39 UTC. URL: http://de.wikipedia.org/w/index.php?title=M%C3%B6biusband&oldid=19680801 (Abgerufen: 22. August 2006, 14:11 UTC)

[Wik06c] Artikel *Jordanscher Kurvensatz.* In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 23. September 2006, 14:16 UTC. URL: http://de.wikipedia.org/w/index.php?title=Jordanscher_Kurvensatz&oldid=21803722 (Abgerufen: 6. Dezember 2006, 12:54 UTC)

[Wik06d] Artikel *Toadmoor Tunnel*. (2006, August 28). In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 28 August 2006 13:03 UTC. URL: http://en.wikipedia.org/w/index.php?title=Toadmoor_Tunnel&oldid=72381204 (Abgerufen: 6. Dezember 2006, 13:34 UTC)

[Zom05] Zomorodian, A. J. Topology for Computing. Cambrigde University Press, 2005. ISBN 0-521-83666-2

F Abbildungsverzeichnis

G Aufteilung der Arbeit

Die Aufteilung des Schreibens dieser Arbeit unter den Diplomanden ergibt sich aus der folgenden Tabelle:

Kapitel	Autor(en)
1 – 2.1	Gemeinschaftsarbeit
2.2 – 3.4	Florian Heinrich
3.4.1	Gemeinschaftsarbeit
3.4.2 – 3.4.4	Florian Heinrich
3.4.5	Benjamin Seppke
3.4.6	Gemeinschaftsarbeit
4	Florian Heinrich
5 – 5.3.2	Benjamin Seppke
5.4 – 5.4.1	Gemeinschaftsarbeit
5.4.2 – 5.4.3.2	Benjamin Seppke
6 – 6.1.1	Florian Heinrich
6.1.2	Benjamin Seppke
6.2 – 6.2.2	Florian Heinrich
6.2.3 – 6.2.4	Gemeinschaftsarbeit
6.3 – 6.3.2	Benjamin Seppke
7 – 7.1	Florian Heinrich
7.2 – 7.2.4	Benjamin Seppke
7.3 – 7.3.3	Florian Heinrich
7.4 – 7.4.2	Gemeinschaftsarbeit
7.5 – 7.5.3.1	Florian Heinrich
7.5.3.2 – 7.5.3.3	Gemeinschaftsarbeit
8 – 8.1.3	Benjamin Seppke
8.2 – 9.2.1	Florian Heinrich
9.2.2 – 9.2.3	Benjamin Seppke
9.3 – 9.4	Florian Heinrich
10	Gemeinschaftsarbeit