Fakultät für Informatik
Professur Graphische Datenverarbeitung und Visualisierung

Diplomarbeit

Verwaltung sehr großer volumetrischer Datensätze unter Verwendung der hierarchischen Lauflängenkodierung

Holger Gerth

Chemnitz, den 25. September 2007

Prüfer: Prof. Dr. Guido Brunnett
Betreuer: Dipl.-Inf. David Brunner

Gerth, Holger
Verwaltung sehr großer volumetrischer Datensätze unter Verwendung der hierarchischen Lauflängenkodierung
Diplomarbeit, Fakultät für Informatik
Technische Universität Chemnitz, September 2007

Inhaltsverzeichnis

Abbildungsverzeichnis **v**

1 Einleitung **1**

1.1 Motivation 1
1.2 Ziel der Arbeit 1
1.3 Gliederung 1

2 Grundlagen **3**

2.1 Morphologische Bildverarbeitung 3
2.1.1 Dilation 4
2.1.2 Erosion 5
2.1.3 Opening und Closing 6
2.2 Topologische Ausdünnung / Skelettierung 7
2.2.1 Topologie 7
Topologische Eigenschaften 8
Eulerzahl 9
Topologische Klassifikation 10
2.2.2 Skelettierungsalgorithmen im 3D 13
Erhaltung der Topologie 13
Verbundskomponenten in kleinen Nachbarschaften 14
Arten von Algorithmen 16
Algorithmus von Tsao und Fu 17
Algorithmus von Gong und Bertrand 18
2.3 Hierarchische Lauflängenkodierte Niveaumenge 21
2.3.1 Aufbau der Datenstruktur 22
2.3.2 Hierarchische Datenstruktur 23
2.3.3 Direktzugriff 25
2.3.4 Sequenzieller Zugriff 26
2.3.5 Schneller Zugriff auf Nachbarn 27
2.3.6 Dilation 27
1D Dilation 28
Union-Algorithmus 28

3 Aufbau der Datenstruktur 31
3.1 Vorverarbeitung der Daten . . . 31
3.2 Änderungen an der H-LLK Datenstruktur . . . 32
3.3 Dateiformat . . . 32
3.4 Erzeugung der Daten . . . 34
3.5 Laden der Daten zur Bearbeitung . . . 35

4 Algorithmen auf der Datenstruktur 37
4.1 Dilation . . . 37
4.1.1 1D Dilation . . . 37
4.1.2 N-D Dilation . . . 38
4.2 Erosion . . . 39
4.2.1 1D Erosion . . . 40
4.2.2 N-D Erosion . . . 40
4.2.3 Berechnung der Schnittmenge . . . 42
4.3 Morphologische Operationen . . . 44
4.4 Ausdünnung/Skelettierung . . . 45
4.4.1 Ermittlung der Nachbarschaften . . . 46
Ermittlung der $\mathbf{N_6}$ Nachbarschaft . . . 46
Ermittlung der $\mathbf{N_{26}}$ Nachbarschaft . . . 46
4.4.2 Ermittlung der definierten Punkte . . . 48
4.4.3 Gerichtete Randpunkte . . . 48
4.4.4 Finale Punkte . . . 49
4.4.5 Simple Punkte . . . 49
4.4.6 Simplizität in der Ebene . . . 49
4.4.7 Löschen von Punkten . . . 50
4.4.8 Partielle Skelettierung . . . 51

5 Ergebnisse 54
5.1 Speicherbedarf . . . 56
5.2 Laufzeiten der Algorithmen . . . 57

6 Zusammenfassung 60
6.1 Diskussion . . . 60
6.2 Ausblick . . . 61

Glossar 62

Literaturverzeichnis 63

A Das Testprogramm 67

B Algorithmen **68**
B.1 Allgemeine Datentypen 68
B.2 Algorithmen der H-LLK Niveaumenge 69
B.2.1 Direktzugriff 69
B.2.2 Sequenzieller Zugriff 69
B.2.3 Dilationsalgortithmus 70
1D Dilation 70
Union 70
ND Dilation 72
B.2.4 Erosionsalgorithmus 73
1D Erosion 73
Schnittmenge 74
ND Erosion 75
B.2.5 Skelettierung 76
Mediale Fläche / mediale Achse 76
Simplizität in der Ebene 77
Löschen von Punkten 77
Berechnung eines Ausdünnungsschritts 79
B.3 Algorithmen der erweiterten H-LLK Niveaumenge 80
B.3.1 Morphologische Operatoren 80
B.3.2 Skelettierung 80

Abbildungsverzeichnis

2.1 Beispiel für die Dilation im 2D 4
2.2 Beispiel für die Erosion im 2D 5
2.3 Beispiel für ein Opening im 2D 6
2.4 Beispiel für ein Closing im 2D 7
2.5 Von links: ein 3D Objekt sowie seine mediale Fläche und die mediale Achse 8
2.6 Topologische Klassifikation 11
2.7 Reihenfolge der Nachbarn in einer $3 \times 3 \times 3$ Nachbarschaft 16
2.8 Ein Objekt, welches ausschließlich aus simplen Punkten besteht 17
2.9 Simplizität in der Ebene 18
2.10 1D LLK Niveaumenge 22
2.11 Ein 2D Objekt und seine klassifizierten Voxel 23
2.12 Kodierung eines 2D Objekts in eine 2D H-LLK Niveaumenge 24
2.13 Die 1D Dilation veranschaulicht als Grafik 28

3.1 Aufbau der Hauptdatei 33
3.2 Aufbau der Segment-Dateien 34

4.1 Schnittmenge zweier LLK Segmente 42
4.2 Reihenfolge der gefundenen Nachbarn bei der N_{26} Nachbarschaft 47
4.3 Beispiel für die Rückgabe der Punkte beim sequenziellen Zugriff 48

5.1 Das Kuh-Modell in Voxeldarstellung 54
5.2 Speicherbedarf der erzeugten Voxeldatensätze 55
5.3 Speicherbedarf der erweiterten H-LLK Datenstruktur 56
5.4 Laufzeiten der Dilation 57
5.5 Laufzeiten der Erosion 58
5.6 Laufzeiten der Skelettierung 59

A.1 Screenshot des Testprogramms 67

1 Einleitung

1.1 Motivation

Moderne Aufnahmetechnologien, wie die Computertomographie in der Medizin oder das Elektronenmikroskop in der Biochemie, sind heutzutage in der Lage, Aufnahmen in hohen Auflösungen und damit sehr großem Detailreichtum zu erzeugen. Werden die Daten in Form eines einfachen kartesischen Gitters gespeichert, dann wird allerdings für diese Aufnahmen mit Auflösungen von bis zu 5000^3 Voxeln sehr viel Speicher benötigt.

Da nicht alle Voxel brauchbare Informationen enthalten, ist es sinnvoller lediglich die Voxel zu speichern, die interessante Daten enthalten. Allerdings besteht bei solch einer Komprimierung das Problem, daß die Algorithmen für die benötigten Operationen, wie zum Beispiel Dilation oder Erosion, auf solchen speichereffizienten Datenstrukturen sowie der Zugriff auf die einzelnen Daten unter Umständen sehr langsam sein kann.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist der Entwurf einer speichereffizienten Datenstruktur zur Verwaltung komplexer Datensätze. Die Datenstruktur soll auf der Grundlage der Hierarchischen Lauflängenkodierten Niveaumenge von Housten u.a. [HBN+06] aufgebaut werden, welche sich im Vergleich von mehreren speichereffizienten Datenstrukturen für volumetrische Datensätze in einer Studienarbeit ([HG07]) als bester Kompromiß in den Bereichen Speichereffizienz, Flexibilität bei Änderungen an den Daten sowie Zugriffszeiten gezeigt hat.

Desweiteren sollen grundlegende Algorithmen der Bildverarbeitung, wie die Dilation, die Erosion und die Skelettierung auf der Datenstruktur effizient implementiert werden. Der Anspruch liegt vorallem in der Anpassung an die Struktur der Hierarchischen Lauflängenkodierten Niveaumenge und in einer schnellen Verarbeitung, ohne daß das komplette Objekt in den Speicher geladen werden muß.

1.3 Gliederung

In Kapitel 2 werden die Grundlagen erläutert, welche die morphologische Bildverarbeitung, die Skelettierung und die zu Grunde liegende Hierarchische Lauflängenkodierte Niveaumenge umfassen. Diese drei Themengebiete sowie die darin enthaltenen Begriffe sind notwendig

für das weitere Verständnis der Arbeit.

In Kapitel 3 wird der Aufbau der Datenstruktur erklärt.Desweiteren wird aufgezeigt, wie die als Eingabe vorliegenden Grauwertbilder binärisiert und zu einem dreidimensionalen Objekt zusammengefaßt werden. Schließlich wird noch die Verwaltung der Daten erläutert, was hauptsächlich das Speichern auf dem Datenträger und das Laden der Daten für die Bearbeitung umfaßt.

Das Kapitel 4 umfaßt die Beschreibung der auf der Datenstruktur implementierten Algorithmen. Es werden die grundlegende Funktionsweise sowie die Änderungen zu einigen originalen Algorithmen der Hierarchischen Lauflängenkodierten Niveaumenge beschrieben. Die exakten Algorithmen in Pseudo-Code findet man im Anhang.

Im Kapitel 6 werden die Ergebnisse der vorliegenden Arbeit zusammengefaßt. Desweiteren werden noch Überlegungen zur Verbesserung und Erweiterung der Datenstruktur aufgezeigt.

2 Grundlagen

2.1 Morphologische Bildverarbeitung

Unter der mathematischen Morphologie[1] versteht man in der Bildverarbeitung die Verarbeitung von Rasterbildern. Sie dient primär zur Formanalyse.
Die Anwendungsgebiete für die mathematische Morphologie sind vielfältig. Man braucht sie z.B. in der medizinischen Bildverarbeitung oder der computergestützten Qualitätskontrolle.

Sei $\mathbb{Z}$ die Menge der ganzen Zahlen und

$$\mathbb{Z}^{\ltimes} = \underbrace{\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \ldots \times \mathbb{Z}}_{n-mal}.$$

Jedes Rasterbild ist als Teilmenge von $\mathbb{Z}^n$ darstellbar. Binärbilder werden als Teilmengen von $\mathbb{Z}^2$, bestehend aus Ortskoordinaten der Bildpunkte beschrieben.
Dagegen werden Grauwert- oder zeitinvariante Binärbilder in $\mathbb{Z}^3$ aus Ortskoordinaten und der Zeit bzw. dem Grauwert gebildet. Viele komplexe Bildverarbeitungsalgorithmen lassen sich auf die folgenden elementaren Operationen zurückführen.

Als Beispiel sei hier die Translation als morphologische Operation beschrieben. Es sei $X \subseteq \mathbb{Z}^n$ ein Rasterbild und $h \in \mathbb{Z}^n$ ein beliebiger Vektor, um den das Objekt verschoben werden soll. Das neue Bild wird nun nach folgender Vorschrift erzeugt:

$$X_h = \{d \in \mathbb{Z}^n : \exists x \in X \text{ mit } d = x + h\}.$$

Elementare Operatoren der mathematischen Morphologie, auf die sich komplexe Bildverarbeitungsaufgaben zurückführen lassen, sind die Dilation (auch Dilatation), die Erosion sowie das Opening und das Closing, welche nachfolgend beschrieben werden. Komplexe Bildverarbeitungsaufgaben lassen sich auf diese einfachen Basisoperationen zurückführen.

[1] Lehre von der Form

2.1.1 Dilation

Seien $X, B \subseteq \mathbb{Z}^n$. Die Menge

$$X \oplus B = \{d \in \mathbb{Z}^n : \exists x \in X, \exists b \in B \text{ mit } d = x + b\}$$

heißt *Dilation* der Mengen X und B.

Bei der Dilation kommt es zu einer Verstärkung bzw. Verdickung des Bildes X bezüglich des Strukturelements B. Durch die Dilation können Teilchengruppen vereint und Löcher sowie Risse im Objekt geschlossen werden. Abbildung 2.1 zeigt ein Beispiel für die Dilation im 2D.

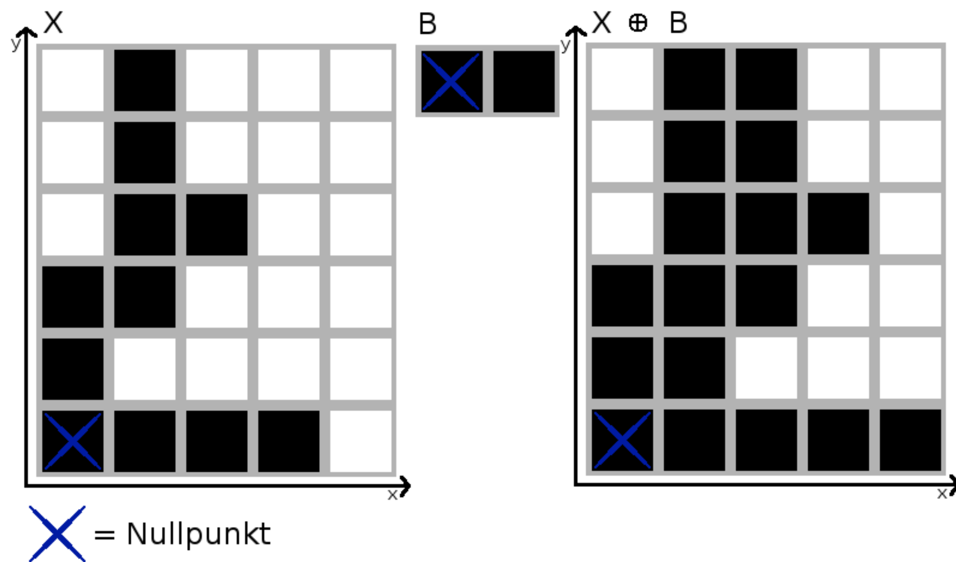


Abbildung 2.1: Beispiel für die Dilation im 2D

2.1.2 Erosion

Seien $X, B \subseteq \mathbb{Z}^n$. Die Menge

$$X \ominus B = \{d \in \mathbb{Z}^n : \forall b \in B \Rightarrow d + b \in X\}$$

heißt *Erosion* der Mengen X und B.

Bei der Erosion kommt es demnach zu einer Abtragung des Bildes X bezüglich des Strukturelementes B. Schmale Stellen und kleine Objekte, deren geometrische Ausdehnungen kleiner als die des Strukturelements sind, werden bei der Erosion vollständig eliminiert. Abbildung 2.2 zeigt ein Beispiel für die Erosion im 2D.

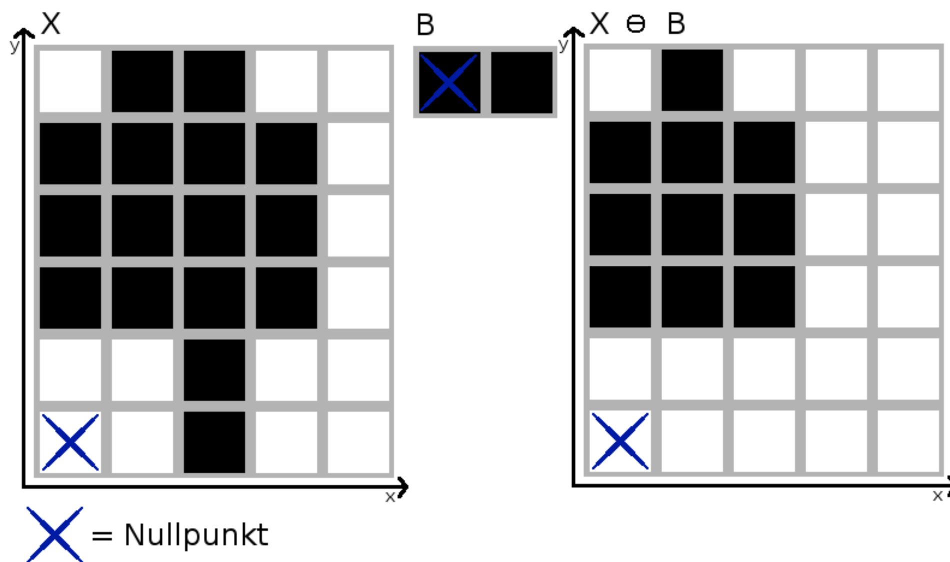


Abbildung 2.2: Beispiel für die Erosion im 2D

2.1.3 Opening und Closing

Seien $X, B \subseteq \mathbb{Z}^n$. Die Menge

$$X \circ B = (X \ominus B) \oplus B$$

heißt *Opening* der Mengen X und B und die Menge

$$X \bullet B = (X \oplus B) \ominus B$$

heißt *Closing* der Mengen X und B.

Beim Opening wird also zuerst eine Erosion auf das Bild angewandt und danach eine Dilation auf dem Ergebnis ausgeführt. Das Opening bewirkt eine Eliminierung von im Verhältnis zum Strukturelement B kleinen Teilmengen des Bildes X, d.h. schmale Verbindungen oder auch alleinstehende Mengenelemente werden gelöscht.
Beim Closing werden Erosion und Dilation in umgekehrter Weise auf das Bild angewandt und kleine Einschnitte und Zwischenräume geschlossen.
Die Abbildungen 2.3 und 2.4 zeigen Beispiele für die beiden Operationen in 2D.

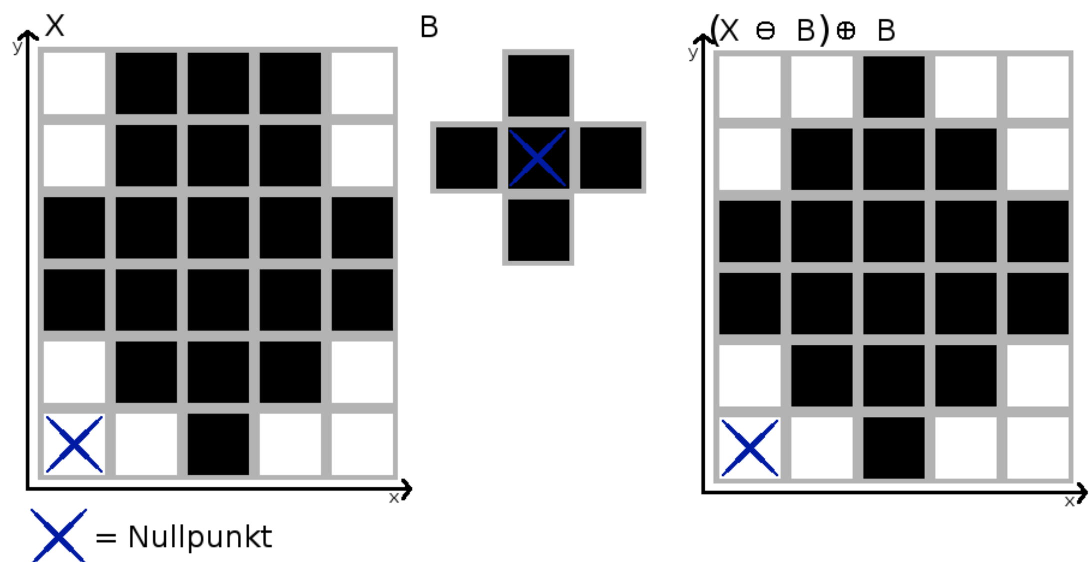


Abbildung 2.3: Beispiel für ein Opening im 2D

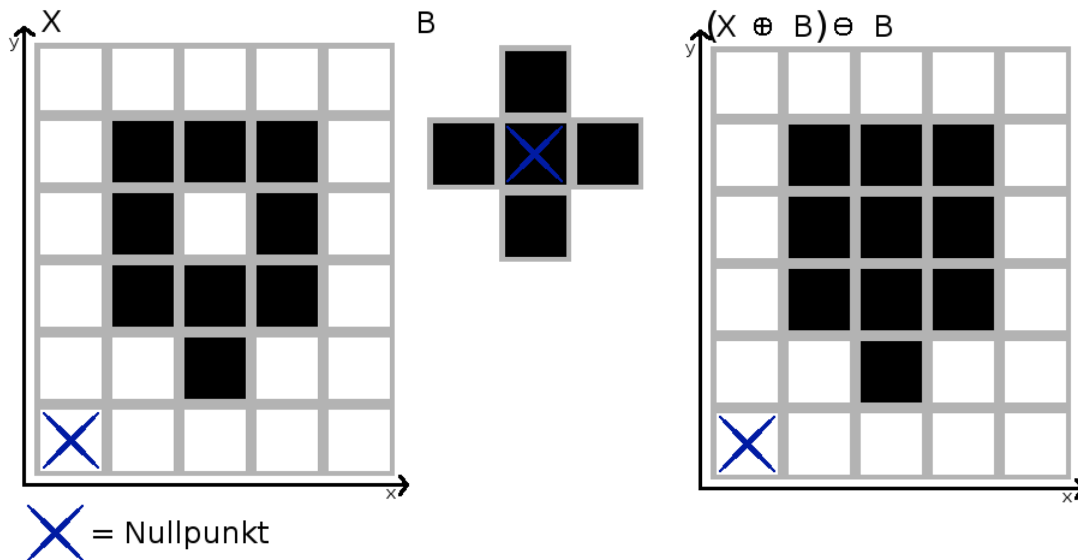


Abbildung 2.4: Beispiel für ein Closing im 2D

2.2 Topologische Ausdünnung / Skelettierung

In einigen Anwendungen ist es notwendig, aus flächenhaften Objekten mittels Ausdünnung linienhafte Objekte zu erzeugen. Diese Ausdünnung wird auch als *Skelettierung* bezeichnet und wird für die Formbeschreibung komplexer Objekte, für die Kurvendarstellung und für die Trennung von Objekten benötigt.

Unter Skelettierung versteht man das Abtragen von Randpunkten (Randvoxeln) bis schließlich nur noch ein Skelett des originalen Bildes übrig ist. Im 3D unterscheidet man dabei zwischen zwei verschiedenen Arten von Skeletten: zum einen wäre das die *mediale Fläche* und zum anderen die *mediale Achse* (siehe Abbildung 2.5).

2.2.1 Topologie

Unter der *Topologie* eines 3D Objekts versteht man dessen Beschreibung über diverse Charakteristika, welche invariant gegenüber Homöomorphismen (bijektive, stetige Abbildungen zwischen zwei Objekten wie Dehnen, Stauchen, Verbiegen oder Verzerren) sind. Man kann sich das Objekt als eine Art elastisches Material vorstellen, welches man dehnen, verbiegen oder verdrehen kann, ohne daß es dabei auseinanderbricht, Löcher bekommt oder zwischen den Objektgrenzen eine Verbindung entsteht, wo vorher keine war. Eigenschaften wie die Größe und die Krümmung des Objekts sowie dessen Orientierung würden sich bei Transformationen verändern. Für die Topologie benötigt man also andere Eigenschaften.

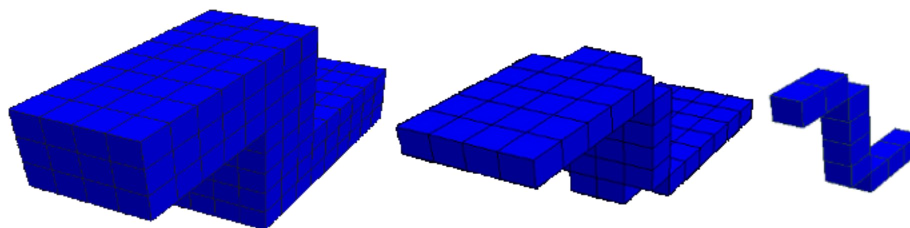


Abbildung 2.5: Von links: ein 3D Objekt sowie seine mediale Fläche und die mediale Achse

Topologische Eigenschaften

Eine gegenüber Homöomorphismen invariante Eigenschaft ist die Anzahl der *Verbundskomponenten*. Als Verbundskomponenten bezeichnet man Teile der Vordergrundvoxel, die einen zusammenhängenden Datenblock bilden. Welche Voxel Verbundskomponenten bilden, wird anhand der gewählten Nachbarschaftsbeziehung bestimmt.
Im 3D gibt es drei verschiedene Arten von Nachbarschaftsbeziehungen. Bei der *Sechser Nachbarschaft* (N_6) werden nur die sechs Nachbarn des Voxels betrachtet, die sich in maximal einer Koordinate unterscheiden. Bei der *Achtzehner Nachbarschaft* (N_{18}) kommen zu diesen sechs Nachbaren noch die hinzu, die sich in maximal zwei Koordinaten unterscheiden. Bei der *Sechsundzwanziger Nachbarschaft* (N_{26}) werden schließlich alle Nachbarn des Voxels betrachtet. Die entsprechenden Pendants im 2D sind die *Vierer Nachbarschaft* (N_4) und die *Achter Nachbarschaft* (N_8).
Formal lassen sich die Nachbarschaften folgendermaßen beschreiben. Sei $S \subset \mathbb{Z}^n$ die Menge der Vordergrundvoxel und $v \in S$ ein Vordergrundvoxel sowie $w \in \mathbb{Z}^n$ ein beliebiges anderes Voxel.

$$
\begin{aligned}
N_6(v): \quad & \{w: \quad |v_x - w_x| + |v_y - w_y| + |v_z - w_z| = 1\} \\
N_{18}(v): \quad & \{w: \quad max(|v_x - w_x|, |v_y - w_y|, |v_z - w_z|) = 1 \quad \text{und} \\
& \qquad\quad |v_x - w_x| + |v_y - w_y| + |v_z - w_z| \leq 2\} \\
N_{26}(v): \quad & \{w: \quad max(|v_x - w_x|, |v_y - w_y|, |v_z - w_z|) = 1\}
\end{aligned} \tag{2.1}
$$

Eine weitere invariante geometrische Eigenschaft des Objekts ist die Anzahl der *Löcher*. Als Loch bezeichnet man eine von dem Objekt vollständig eingeschlossene Komponente des Hintergrunds.

Die letzte Eigenschaft wird als *Henkel* des Objekts bezeichnet. Oftmals werden die Henkel

auch als Tunnel durch das Objekt hindurch bezeichnet, also als ein Loch mit zwei Ausgängen auf der Oberfläche. Diese Definition ist aber nicht immer eindeutig. Deshalb definiert man Henkel auch über die Anzahl der Jordankurven, also die maximale Anzahl von nichttrennenden Schnitten, die man durch das Objekt machen kann, ohne zusätzliche Verbundskomponenten zu erzeugen. Die Anzahl der Henkel wird auch als das *Geschlecht* des Objekts bezeichnet.

Eulerzahl

Die drei vorher genannten invarianten geometrischen Eigenschaften, also die Anzahl der Löcher, die Anzahl der Verbundskomponenten und das Geschlecht des Objekts vereint, ergeben die *Eulerzahl*, welche über die Euler-Charakteristik χ folgendermaßen definiert ist. Sei c die Anzahl der Verbundskomponenten, h die Anzahl der Löcher und g das Geschlecht eines dreidimensionalen Objekts $X \subseteq \mathbb{Z}^3$:

$$\chi(X) = c + h - g \tag{2.2}$$

Zwei Objekte sind *topologisch äquivalent*, wenn sie die selbe Eulerzahl haben.

Die Eulerzahl ist allerdings eine globale Eigenschaft und verlangt deshalb auch nach globalen Algorithmen, die aber viel zu zeitaufwendig und daher uninteressant sind. Da sich die Anzahl der Verbundskomponenten nicht lokal berechnen läßt, liegt es nahe, daß man auch die Eulerzahl nicht lokal berechnen kann. Glücklicherweise läßt sich aber die Änderung der Eulerzahl sowie auch des Geschlechts des Objekts trotzdem lokal durch eine Untersuchung der Nachbarschaften bestimmen. Als mathematischer Hintergrund dient die Tatsache, daß sich das Geschlecht eines digitalen Objekts durch die Untersuchung der Oberflächen, welche das Objekt und dessen Löcher umschließen, berechnen läßt. Solche Oberflächen bestehen aus einem Gefüge von Polyedern und werden als *netzförmige Oberflächen* bezeichnet. Sie bestehen aus Vertizes, Kanten und Flächen von Voxeln. Das Geschlecht einer solchen netzförmigen Oberfläche kann nun durch Zählen der Vertizes, Kanten und Flächen bestimmt werden. Sei v die Anzahl der Vertizes, k die Anzahl der Kanten und f die Anzahl der Flächen:

$$2 - 2g = v - k + f \tag{2.3}$$

Mit Hilfe der in Formel 2.3 beschrieben Tatsache, läßt sich das Geschlecht durchaus mittels lokaler Operatoren berechnen.
Jede Verbundskomponente und jedes Loch wird von einer netzförmigen Oberfläche eingeschlossen. Bei k Komponenten und l Löchern gibt es somit $k + l$ dieser netzförmigen Oberflächen. Das Geschlecht einer netzförmigen Oberfläche kann mit der Formel 2.3 berechnet werden. Jede netzförmige Oberfläche enthält entweder eine einzige Verbundskomponente oder ein Loch. Es gilt:

$$\chi_i = 1 - g_i \tag{2.4}$$

und somit

$$2 - 2g_i = 2 - 2 * (1 - \chi_i) = 2\chi_i \tag{2.5}$$

Mit χ_i wird hier die Eulerzahl einer Verbundskomponente bezeichnet. Daraus ergibt sich folgendes:

$$\begin{aligned} 2\chi &= \sum_{i=1}^{k+l} 2\chi_i \\ &= \sum_{i=1}^{k+l} (2 - 2g_i) \\ &= \sum_{i=1}^{k+l} v_i - k_i + f_i \end{aligned} \tag{2.6}$$

An Formel 2.6 sieht man, daß sich die Eulerzahl durch Zählen der Vertizes, Kanten und Flächen berechnen läßt. Allerdings ist dies nicht so einfach, wie es auf den ersten Blick erscheint, da es nicht ausreicht, Oberflächenvoxel zu ermitteln und deren Kanten und Vertizes zu zählen. Einige Kanten und Vertizes würden so doppelt gezählt werden.

Topologische Klassifikation

Die Punkte im euklidischen Raum $\mathbb{E}^n$ lassen sich nach ihrer Lage klassifizieren.

Definition 2.1 *Sei ein Punkt $p \in \mathbb{E}^n$, $\varepsilon > 0$.*
*Die **ε-Umgebung** $U_\varepsilon(p)$ ist die Menge aller Punkte q mit euklidischem Abstand $d(p,q) < \varepsilon$.*

Ausgehend von dieser Definition, können Punkte im $\mathbb{E}^n$ folgendermaßen klassifiziert werden. Sei $p \in \mathbb{E}^n$, $S \subset \mathbb{E}^n$ und $\bar{S} = \mathbb{E}^n \setminus S$.

1. p ist *innerer Punkt* von S, wenn eine ε-Umgebung von p existiert mit $U_\varepsilon(p) \subset S$
2. p ist *Randpunkt* von S, wenn jede ε-Umgebung von p sowohl einen Punkt von S als auch einen Punkt von $\bar{S}$ enthält
3. p ist *Berührpunkt* von S, wenn jede ε-Umgebung von p einen Punkt von S enthält
4. p ist *Häufungspunkt* von S, wenn p Berührpunkt von $S \setminus \{p\}$ ist
5. p ist ein *isolierter Punkt* von S, wenn eine ε-Umgebung von p existiert mit $U_\varepsilon(p) \cap S = \{p\}$

Noch einige Bemerkungen dazu: Ein isolierter Punkt ist offensichtlich ein spezieller Randpunkt. Innere und isolierte Punkte sind stets Elemente von S, während alle anderen Punkte nicht zu S gehören müssen. Ein Berührpunkt von S, der nicht zu S gehört, ist Randpunkt von S. Jeder Punkt von S ist Berührpunkt von S. Ein Punkt von S, der nicht Häufungspunkt von

S ist, ist ein isolierter Punkt.

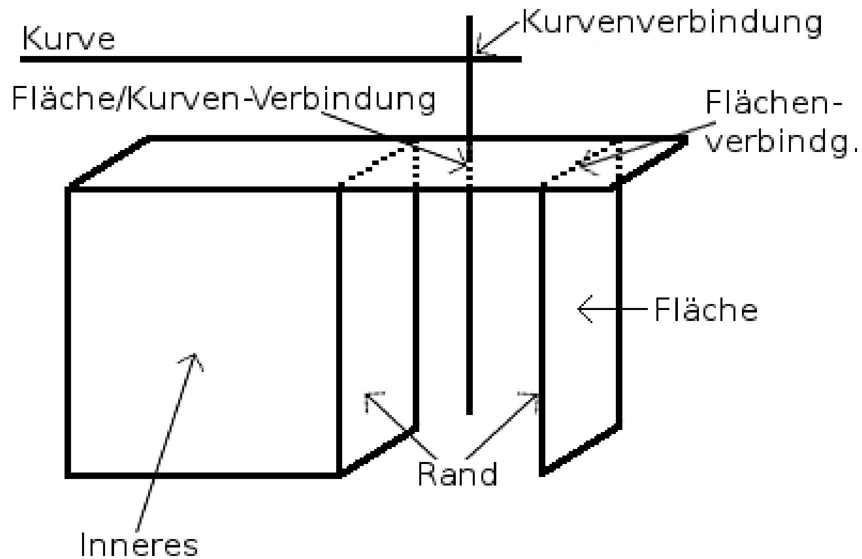


Abbildung 2.6: Topologische Klassifikation

In der Praxis entspricht die ε-Umgebung der Nachbarschaften eines Punktes. Wurde eine ε-Umgebung für einen Punkt gefunden, die eine der oben genannten Bedingungen erfüllt, so erfüllen auch alle kleineren ε-Umgebungen diese Bedingungen. Um Punkte zu klassifizieren, reicht es demnach, ihre kleinen Nachbarschaften zu betrachten. Unterscheidet man, wie in Abbildung 2.6 ersichtlich, zusätzlich, ob der Punkt zu einer Kurve oder Fläche gehört, ergeben sich folgende neun Klassifizierungen:

1. innerer Punkt,
2. isolierter Punkt,
3. Randpunkt,
4. Kurvenpunkt,
5. Kurvenverbindung,
6. Flächenpunkt,

7. Fläche/Kurve(n)-Verbindung,
8. Flächenverbindung,
9. Flächen/Kurve(n)-Verbindung.

Sei nun $\bar{K}$ die Anzahl Verbundskomponenten des Hintergrunds und K^* die Anzahl Verbundskomponenten des Vordergrunds in einer $3 \times 3 \times 3$ Nachbarschaft. Dann lassen sich die Punkte entsprechend der Anzahl Verbundskomponenten in ihrer Nachbarschaft klassifizieren.

1	innerer Punkt	$\bar{K} = 0$	
2	isolierter Punkt		$K^* = 0$
3	Randpunkt	$\bar{K} = 1$	$K^* = 1$
4	Kurvenpunkt	$\bar{K} = 1$	$K^* = 2$
5	Kurvenverbindung	$\bar{K} = 1$	$K^* > 2$
6	Flächenpunkt	$\bar{K} = 2$	$K^* = 1$
7	Fläche/Kurve(n)-Verbindung	$\bar{K} = 2$	$K^* \geq 2$
8	Flächenverbindung	$\bar{K} > 2$	$K^* = 2$
9	Flächen/Kurve(n)-Verbindung	$\bar{K} > 2$	$K^* \geq 2$

Für den Zusammenhang gilt dabei die 26-Verbundenheit für die Vordergrundkomponenten und die 6-Verbundenheit für die Hintergrundkomponenten. Um einen Punkt zu klassifizieren, reicht es also, seine Zusammenhangskomponenten für Vorder- und Hintergrund zu zählen.

2.2.2 Skelettierungsalgorithmen im 3D

Erhaltung der Topologie

Die Skelettierungsalgorithmen sollen das Objekt nun so ausdünnen, ohne daß dessen topologischen Eigenschaften verändert werden. Es sollen also die Anzahl der Komponenten, der Löcher und das Geschlecht des Objekts im resultierenden Objekt unverändert erhalten bleiben. Neben der Topologie soll aber auch noch die grundlegende Gestalt des Objekts erhalten bleiben, so daß das Ergebnis einem Skelett ähnelt. Um dies zu erreichen, dürfen demnach zum einen keine verbundenen Teile getrennt oder nicht verbundene Komponenten vereint werden. Auch dürfen keine Löcher oder Henkel geschlossen oder neue erzeugt werden.
Aus Performanzgründen soll nun die Eulerzahl aber nicht jedes Mal neu berechnet werden, wenn für einen Voxel entschieden werden soll, ob er gelöscht werden kann, ohne das dabei die Topologie des Objekts geändert wird. Es wird daher ein lokales Kriterium benötigt, welches in einer relativ kleinen Nachbarschaft von $3 \times 3 \times 3$ Voxeln überprüft werden kann.

Definition 2.2 *Sei $I \subset \mathbb{Z}^3$ ein 3-dimensionales Binärbild. Ein Punkt $x \in I$ wird als* ***simpler Punkt*** *bezeichnet, wenn sein Löschen die Eulerzahl von I nicht verändert.*

Dies ist genau dann der Fall, wenn die Anzahl der Verbundskomponenten in der N_{26} Nachbarschaft des Punktes für die Vordergrundvoxel und die Hintergrundvoxel genau Eins ist. Wenn man die Definition eines simplen Punktes nun mit der Definition eines Randpunkts aus dem vorangegangenen Abschnitt vergleicht, so wird ersichtlich, daß jeder Randpunkt einem simplen Punkt entspricht und umgekehrt.

Die Erhaltung der Topologie stellt die Minimalanforderung dar, um Gestaltsinformationen bewahren zu können. Sie ist aber normalerweise nicht ausreichend, um die geometrischen Eigenschaften eines Objekts zu charakterisieren. So ist z.B. eine Verbundskomponente ohne Löcher und Henkel topologisch äquivalent zu einem einzigen Voxel. Demnach würde jede Verbundskomponente ohne Hohlräume zu einem Voxel degenerieren, wenn man sich nur auf die Erhaltung der Topologie beschränkt. Es werden also noch zusätzliche Charakteristika benötigt.

Verbundskomponenten in kleinen Nachbarschaften

Um zu bestimmen, ob ein Punkt simpel ist, ist es notwendig, die Anzahl an Verbundskomponenten innerhalb seiner N_{26} Nachbarschaft zu ermitteln. Dazu eignet sich ein einfacher in [BNSdB97] vorgestellter Algorithmus zum Ermitteln der Verbundskomponenten in kleinen $3 \times 3 \times 3$ Nachbarschaften und wird nachfolgend in leichter Abwandlung erläutert.

Sei $N'_{26}(p) = N_{26}(p) \cup p$. Der Algorithmus erhält als Eingabe die N'_{26} Nachbarschaft eines Punktes $p \in \mathbb{Z}^3$ als binäres Bild und eine vorberechnete LookUp-Tabelle L mit den n-verbundenen Nachbarn jedes Punktes $q \in N'_{26}(p)$, $n \in \{6, 18, 26\}$. Desweiteren sei V ein eindimensionales Array mit einem Eintrag für jeden Punkt aus $N'_{26}(p)$, welche zu Beginn mit Null initialisiert werden, sowie ein eindimensionales Ausgabe-Array, welches die Markierungen für jeden Punkt enthält. Markiert wird mit Zahlenwerten beginnend bei Eins.

Es wird nun über alle Punkte der Eingabe iteriert. Wenn der aktuelle Punkt q in der Eingabe Eins ist, aber noch nicht markiert wurde, bekommt er eine neue Markierung g. Der entsprechende Eintrag in V wird auf Eins gesetzt. Anschließend wird über alle Einträge von V iteriert. Wird ein Eintrag mit Eins gefunden, werden alle Nachbaren von q mit Hilfe von L untersucht. Ist ein Nachbar in der Eingabe 1 und wurde noch nicht markiert, so erhält er dann ebenfalls die Markierung g und die entsprechende Position in V wird auf Eins gesetzt. Wurden alle Nachbarn von q betrachtet, wird an dessen Eintrag in V eine Zwei geschrieben, was bedeutet das der Punkt fertig ist. Anschließend wird die Iteration über V so lange wiederholt, bis keine Eins mehr gefunden werden kann, und es wird mit dem nächsten Punkt aus der Eingabe fortgefahren.

Das Ausgabe-Array enthält schließlich die Markierungen für die einzelnen Punkte. Die Anzahl der Verbundskomponenten entspricht dabei der höchsten Markierung. Um die Verbundskomponenten der Hintergrundvoxel zu ermitteln, muß lediglich die Eingabe negiert werden.

Der Algorithmus setzt voraus, daß die Punkte der Eingabe in einer festen Reihenfolge vorliegen (siehe Abbildung 2.7). Desweiteren werden für die verschiedenen Arten der Verbundenheit verschiedene LookUp-Tabellen benötigt. Die Tabelle für die 26-Verbundenheit wird in Tabelle 2.1 dargestellt. Die entsprechenden LookUp-Tabellen für 18- und 6-Verbundenheit sind Untermengen dieser Tabelle und die zugehörigen Indizes werden kursiv und fett dargestellt.

Voxel	**Nachbarn**												
0	**1**	**3**	4	**9**	10	12	*13*						
1	**0**	**2**	3	**4**	5	9	**10**	11	*12*	13	*14*		
2	**1**	4	**5**	10	**11**	*13*	14						
3	**0**	1	**4**	**6**	7	9	*10*	**12**	13	15	*16*		
4	0	**1**	2	**3**	**5**	6	**7**	8	*9*	10	*11*	12	**13**
	14	*15*	16	*17*									
5	1	**2**	**4**	7	**8**	*10*	11	13	**14**	*16*	17		
6	**3**	4	**7**	12	*13*	**15**	16						
7	3	**4**	5	**6**	**8**	*12*	13	*14*	15	**16**	17		
8	4	**5**	**7**	*13*	14	16	**17**						
9	**0**	1	2	*4*	**10**	*12*	13	**18**	19	21	*22*		
10	0	**1**	2	*3*	4	*5*	**9**	**11**	12	**13**	14	18	**19**
	20	*21*	22	*23*									
11	1	**2**	*4*	5	**10**	13	**14**	19	**20**	*22*	23		
12	0	*1*	**3**	4	6	*7*	**9**	10	**13**	**15**	16	18	*19*
13	*0*	1	*2*	3	**4**	5	*6*	7	*8*	**9**	**10**	11	**12**
	14	15	**16**	17	*18*	19	*20*	21	**22**	23	*24*	25	*26*
14	*1*	2	4	**5**	7	8	10	**11**	**13**	16	**17**	*19*	20
	22	**23**	*25*	26									
15	3	*4*	**6**	7	**12**	13	**16**	21	*22*	**24**	25		
16	*3*	4	*5*	6	**7**	8	12	**13**	14	**15**	**17**	*21*	22
	23	24	**25**	26									
17	*4*	5	7	**8**	13	**14**	**16**	*22*	23	25	**26**		
18	**9**	10	12	*13*	**19**	**21**	22						
19	9	**10**	11	*12*	13	*14*	**18**	**20**	21	**22**	23		
20	10	**11**	*13*	14	**19**	22	**23**						
21	9	*10*	**12**	13	15	*16*	**18**	19	**22**	**24**	25		
22	*9*	10	*11*	12	**13**	14	*15*	16	*17*	18	**19**	20	**21**
	23	24	**25**	26									
23	*10*	11	13	**14**	*16*	17	19	**20**	**22**	25	**26**		
24	12	*13*	**15**	16	**21**	22	**25**						
25	*12*	13	*14*	15	**16**	17	21	**22**	23	**24**	**25**		
26	*13*	17	16	**17**	22	**23**	**25**						

Tabelle 2.1: Adjazenztabelle für die Nachbarschaftbeziehungen

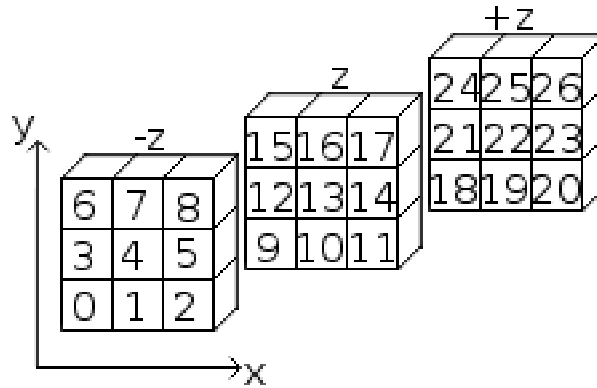


Abbildung 2.7: Reihenfolge der Nachbarn in einer $3 \times 3 \times 3$ Nachbarschaft

Arten von Algorithmen

Zusammenfassend kann gesagt werden, daß Skelettierungsalgorithmen aus dem iterativen Löschen von simplen Punkten bestehen, welche zusätzlich noch einige Nebenbedingungen erfüllen, so daß geometrische Eigenschaften erhalten bleiben. Man unterscheidet dabei grundlegend zwischen zwei Klassen von Algorithmen nach der Reihenfolge, in der sie Punkte löschen.

Bei *sequenziellen Skelettierungsalgorithmen* wird in jeder Iteration nur ein Punkt auf einmal gelöscht. Dies garantiert zwar die Erhaltung der Topologie, trotzdem kann das Objekt in bestimmten Fällen ohne entsprechende Nebenbedingungen bis auf einen Voxel degenerieren.

Die zweite Klasse sind die *parallelen Skelettierungsalgorithmen*, welche in einem Schritt simultan mehrere Punkte auf einmal löschen. Diese Algorithmen haben allerdings das Problem, daß unter Umständen das komplette Objekt verschwinden kann. Als Beispiel dient hier Abbildung 2.8, welche ein Objekt zeigt, das vollständig aus simplen Punkten besteht. Ein wahlloses simultanes Löschen von simplen Punkten würde zum vollständigen Verschwinden des Objekts führen. Trotzdem werden parallele Skelettierungsalgorithmen häufiger verwandt, da sie im Allgemeinen die besseren Ergebnisse liefern. Das Problem des Verschwindens wird dabei mit Hilfe von zusätzlichen Vorkehrungen vermieden. Ein Beispiel für solch eine Vorkehrung ist die Klassifizierung der Voxel in verschiedene Typen. Anschließend wird dann das simultane Löschen nur für einen Typ von Voxeln je Iteration erlaubt.

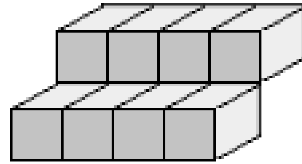


Abbildung 2.8: Ein Objekt, welches ausschließlich aus simplen Punkten besteht

Definition 2.3 *Sei $I \subset \mathbb{Z}^3$ ein 3-dimensionales Binärbild und $p \in I$ ein Vordergrundvoxel, dann wird p als* ***gerichteter Randpunkt*** *der Richtung N(orden), O(sten), W(esten), S(üden), Ob(en), U(nten) bezeichnet, wenn das zu p adjazente Voxel in Richtung N,O,W,S,Ob,U zum Hintergrund gehört.*

Viele parallele Skelettierungsalgorithmen erlauben das simultane Löschen von gerichteten Randpunkten eines Typs. Das Löschen wird dabei in Form von Unterzyklen, einen für jede Richtung, realisiert. Dies geschieht in zwei Phasen. In einem ersten Scan werden alle löschbaren Randpunkte markiert. Ist die Markierung abgeschlossen, werden die Punkte in einem zweiten Scan simultan gelöscht. Dies wird solange wiederholt, bis kein weiterer Punkt gelöscht werden kann.

Dies sichert zwar, daß ein Objekt nicht verschwindet, allerdings kann es immer noch zu einem einzigen Punkt degenerieren. Wenn ein Objekt Verzweigungen enthält, so sollen diese Verzweigungen im Ergebnis in ausgedünnter Form erhalten bleiben. Es sind also noch weitere Kriterien notwendig, um die grundlegende Form des Objekts zu erhalten und exzessives Schrumpfen zu verhindern. Die Grundidee dabei ist, die Endpunkte dieser Verzweigungen zu identifizieren und deren Löschen zu verbieten, auch wenn es sich um simple Punkte handelt.

Definition 2.4 *Ein simpler Punkt $x \in I$ wird als* ***finaler Punkt*** *bezeichnet, wenn die Anzahl n an Vordergrundvoxeln in seiner N_{26}-Nachbarschaft kleiner als 2 ist.*

Existieren keine adjazenten Vordergrundvoxel, so handelt es sich um einen isolierten Punkt. Beträgt $n = 1$, so handelt es sich um einen Endpunkt einer Verzweigung oder Kurve. In beiden Fällen soll der Punkt dann nicht gelöscht werden.

Algorithmus von Tsao und Fu

Der *Algorithmus von Tsao und Fu* [TF81] gehört zur Klasse der parallelen Skelettierungsalgorithmen. Die Berechnung des Skeletts erfolgt dabei in zwei Schritten. Im ersten Schritt wird die mediale Fläche des Objekts berechnet und anschließend davon ausgehend die mediale Achse.
Der Algorithmus löscht in Unterzyklen parallel simple Punkte, die nicht final sind. Desweiteren wird durch eine zusätzliche Bedingung zur “Simplizität in der Ebene” sichergestellt,

daß das resultierende Skelett glatt ist. Für jeden Unterzyklus gibt es dabei zwei zu überprüfende Ebenen, welche die 3×3-Ebenen orthogonal zur Richtung des aktuellen Unterzyklus sind. Die "Simplizität in der Ebene" Bedingung ist erfüllt, wenn für diese beiden Ebenen gilt, daß das aktuelle Voxel simpel bezüglich der jeweiligen Ebene ist, sprich sein Löschen die Verbindung zwischen den restlichen Vordergrundvoxeln in dieser Ebene nicht auflöst, und die verbleibende Anzahl Vordergrundvoxel größer 2 ist. Abbildung 2.9 zeigt ein Beispiel für den löschbaren und den nicht löschbaren Fall. Vordergrundvoxel werden durch Einsen representiert, Hintergrundvoxel durch Nullen.

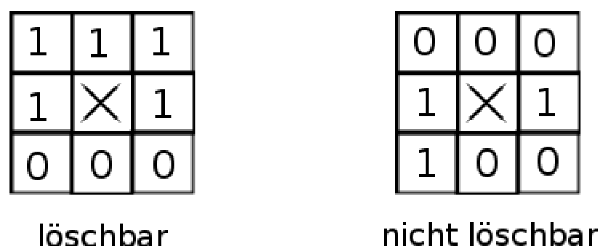


Abbildung 2.9: Simplizität in der Ebene

Wurde die mediale Fläche ermittelt, kann man in ähnlicher Weise fortfahren, um davon ausgehend die mediale Achse zu berechnen. Der einzige Unterschied liegt in einer Lockerung der "Simplizität in der Ebene" Bedingung. So muß diesmal die Anzahl der verbleibenden Vordergrundvoxel in einer Ebene nur noch größer 1 sein.

Algorithmus von Gong und Bertrand

Ein weiterer paralleler Skelettierungsalgorithmus ist der *einfache parallene 3D Ausdünnungsalgorithmus* [GB90] von Gong und Bertrand. Dabei kritisieren sie an Skelettierungsalgorithmen wie dem von Tsao und Fu, die den Pfadzusammenhang in der N'_{26} Nachbarschaft ($N'_{26}(p) = N_{26}(p) \cup p$) verwenden, um die Topologie zu erhalten, daß diese Bedingung nicht ausreichend ist, da trotzdem ein Loch entstehen kann, obwohl der Pfadzusammenhang in N'_{26} gewährleistet ist. Tsao umgeht das Problem durch die zusätzliche "Überprüfung der Ebene" Bedingung.
Dieses Problem versuchen Gong und Bertrand nun durch eine Neudefinition der simplen Punkte in ihrem Algorithmus vollständig auszuschließen. Das Ergebnis des Algorithmus ist die mediale Fläche.

Sei $S \in \mathbb{Z}^3$ ein 3-dimensionales volumetrisches Objekt: S wird durch Einsen representiert,

$\bar{S}$ durch Nullen. Sei weiterhin NK die Anzahl der Verbundskomponenten und NL die Anzahl Löcher. Nach [Mor81] erfüllt ein simpler Punkt folgende Bedingungen:

$$
\begin{array}{lll}
1a & NK\left(S\cap N'_{26}(p)\right) & = NK\left(S\cap N_{26}(p)\right) \\
1b & NK\left(\bar{S}\cap N_{26}(p)\cup p\right) & = NK\left(\bar{S}\cap N_{26}(p)\right) \\
1c & NL\left(S\cap N'_{26}(p)\right) & = NL\left(S\cap N_{26}(p)\right) \\
1d & NL\left(\bar{S}\cap N_{26}(p)\cup p\right) & = NL\left(\bar{S}\cap N_{26}(p)\right)
\end{array}
$$

Der direkte Weg um zu bestimmen, ob ein Punkt simpel ist, wäre die Anzahl der Löcher und Verbundskomponenten zu zählen. Diese Berechnung läßt sich allerdings nicht so einfach parallel implementieren. Unter der Annahme, daß die 26er Verbindung für S und die 6er Verbindung für $\bar{S}$ verwendet wird, werden zwei neue Bedingungen eingeführt:

$$
\begin{array}{lll}
1a' & \bar{K}_6 & = 1 \\
1b' & \bar{K}_{26} & = 1
\end{array}
$$

$\bar{K}_6$ stellt dabei die Anzahl der entsprechend der 6er Verbindung adjazenten Verbundskomponenten in $\bar{S}\cap N_{26}(p)$ dar, $\bar{K}_{26}$ die Anzahl der entsprechend der 26er Verbindung adjazenten Verbundskomponenten in $S\cap N_{26}(p)$. Es ist offensichtlich, daß $\bar{K}_{26}$ gleich $NK(S\cap N_{26}(p))$ ist. Mit diesen neuen Bedingungen kann eine Neudefinition eines simplen Punktes gegeben werden:

Definition 2.5 *Ein Punkt ist dann und nur genau dann simpel, wenn er die Bedingungen 1a', 1b', 1c und 1d erfüllt.*

Diese Neudefinition reicht aber immer noch nicht für eine einfache Bestimmung, ob ein Punkt simpel ist, aus, da immer noch die Anzahl der Löcher gezählt werden muß. Es sind also noch weitere Bedingungen notwendig. Zunächst noch ein paar Erklärungen zum besseren Verständnis.

Im 3D kann ein Punkt als elementarer Würfel betrachtet werden. Besitzt ein Punkt p_i genau eine gemeinsame Kante mit einem anderen Punkt p, so ist p_i 12-adjazent zu p. Teilen sich die beiden Punkte lediglich einen Vertex, so ist p_i 8-adjazent zu p.
Ähnlich wie bei Tsaos Algorithmus, wird eine Iteration in sechs Unterzyklen entsprechend der sechs Richtungen (N,S,O,W,Ob,U) unterteilt, um zu verhindern, daß ein Objekt verschwindet. Sei d die Richtung im aktuellen Unterzyklus. Es werden nun ausschließlich Punkte aus S betrachtet, die in Richtung von d Nachbarn in $\bar{S}$ haben.

Sei $p \in S$, p_d der Nachbar von p in Richtung d entsprechend der 6er Nachbarschaft und $p_{\bar{d}}$ der entsprechende Nachbar in der Gegenrichtung. Außerdem sei $N_d(p) = N_6(p) - \{p_d\} - \{p_{\bar{d}}\}$. Es gelten folgende neue Bedingungen:

2a	$p_d = 0$
2b	$p_{\bar{d}} = 1$
2c	wenn $p_i = 0$ dann ist $p_l = 0$
	$\forall p_i \in N_d(p)$ und für den Punkt p_l 6-adjazent zu p_i und p_d
2d	wenn p_i, p_j und $p_k = 0$ dann ist $p_m = 0$
	$\forall p_i \in N_d(p), p_j \in N_d(p), p_j$ 12-adjazent zu p_i,
	p_k 6-adjazent zu p_i und p_j, p_m 12-adjazent zu p_d und 6-adjazent zu p_k

Erfüllt ein Punkt die Bedingungen in 2, so erfüllt er auch die Bedingungen in 1 und ist damit simpel. Ein Beweis dazu findet man in [GB90].

Sei T1 nun eine topologische Ausdünnungsoperation, welche aus dem simultanen Entfernen von Randpunkten besteht, welche Bedingung 2 innerhalb eines Unterzyklus erfüllen. Die Topologie des Objekts bleibt nach dieser Operation erhalten. Allerdings kann ein Objekts bei der iterativen Anwendung von T1 bis auf einen Voxel degenerieren, falls das Objekt keine Löcher oder Henkel besitzt. Zur Erhaltung der medialen Oberfläche sollten also Randpunkte dieser Oberfläche nicht entfernt werden.
Sei n die Anzahl Einsen in $N_{26}(p)$ und n_i die Anzahl Einsen 6-adjazent zu p im i-ten Oktanten von $N_{26}(p)$. Es gilt eine weitere Bedingung:

$$3 \quad n \geq 8 \vee (4 \leq n \leq 7 \wedge (\exists i \in \{1, ..., 8\}, n_i = 3))$$

Erfüllt ein Punkt Bedingung 3, so wird er nicht als Randpunkt der medialen Oberfläche betrachtet. Die Ausdünnungsoperation T2 besteht nun aus dem simultanen Entfernen von Randpunkten, welche in einem Unterzyklus die Bedingungen 2 und 3 erfüllen.

Der Algorithmus läßt sich nun zusammenfassend durch die iterative Anwendung von T2 beschreiben. In jeder Iteration wird T2 auf einen Typ von Randpunkten in einer festen Reihenfolge angewandt: Ob, N, O, U, S, W. Der Algorithmus stoppt, wenn kein weiterer Punkt gelöscht werden kann.

Gegenüber Tsaos Algorithmus bietet der von Gong und Bertrand eine schnellere Berechnung der medialen Oberfläche, da die Berechnung nicht so aufwendig ist. So besteht der Algorithmus in jedem Unterzyklus nur aus einem Schritt anstatt aus zweien wie bei Tsao. Der Nachteil des Algorithmus liegt darin, daß er lediglich die mediale Fläche, nicht aber die mediale Achse berechnet.

2.3 Hierarchische Lauflängenkodierte Niveaumenge

Die *Hierarchischen Lauflängenkodierten Niveaumenge* (H-LLK Niveaumenge, [HBN+06]) ist eine effiziente Datenstruktur zur Speicherung volumetrischer Datensätze. Zunächst folgen ein paar Erklärungen zum Begriff der lauflängenkodierten Niveaumenge.

Unter einem Lauf versteht man die Zusammenfassung von Daten der selben Art, indem man nur Typ, Startpunkt und Anzahl (Länge des Laufs) speichert. Das Verfahren der Kompression bezeichnet man dann als *Lauflängenkodierung* (LLK).
Zum Beispiel würde aus dem Text *aaabbccccddd* die Lauflängenkodierung *a3b2c4d3* entstehen.

Bei einer *Niveaumenge* (engl. Level Set) wird in einem n-dimensionalen Raum der $(n-1)$-dimensionale Rand Γ eines n-dimensionalen Objekts als Nullstellenmenge einer n-dimensionalen Hilfsfunktion φ beschrieben. Die Hilfsfunktion wird auf dem ganzen betrachteten Gebiet definiert, und zwar mit positiven Werten auf der einen und negativen Werten auf der anderen Seite von Γ.
Der Vorteil einer Niveaumenge liegt darin, daß man Kurven und Oberflächen auf einem räumlich festen (Eulerschen) Koordinatensystem berechnen kann, ohne Parametrisierungen dieser Objekte verwenden zu müssen. So muß die Topologie des Objekts nicht unbedingt bekannt sein bzw. kann sich während der Berechnung ändern. Damit ist eine einfache Verfolgung der Ränder beweglicher Objekte möglich.
Ein Beispiel für eine Niveaumenge wäre ein Kreis in 2D. Der Rand Γ ist gegeben durch die Kreisfunktion $(x-x_M)^2+(y-y_M)^2=r^2$, wobei $p=(x,y)\in\Gamma$ ein Randpunkt ist. Der Mittelpunkt des Kreises ist gegeben durch (x_M,y_M) und sein Radius durch r. Die Hilfsfunktion φ ist in diesem Beispiel eine vorzeichenbehaftete Abstandsfunktion, welche für Punkte $p\in\Gamma$ mit Null, für $(x-x_M)^2+(y-y_M)^2>r^2$ mit positiven Abstand und für $(x-x_M)^2+(y-y_M)^2<r^2$ mit negativen Abstand definiert ist.

Der grundlegende Lösungsvorschlag beruht auf der dimensionsweisen Anwendung der Lauflängenkodierung auf das Objekt. Die Lauflängenkodierung wird dabei für jede Dimension unabhängig entlang der jeweiligen Kodierungsachse durchgeführt, wobei Bereiche außerhalb des Begrenzungsbereichs kompakt gespeichert werden. In Abschnitt 2.3.2 wird die Erzeugung der Datenstrukutur noch genauer beschrieben.

Kodierung der Läufe: Wie bereits erwähnt, nutzt die H-LLK Niveaumenge die Lauflängenkodierung, um das Objekt in einer Serie von Läufen zu kodieren. Es werden dabei folgende drei Kodierungen verwendet: *positive Läufe* beschreiben den Bereich außerhalb des Objekts, *negative Läufe* den Bereich im Inneren und *definierte Läufe* den Bereich innerhalb des Begrenzungsbereichs.

2.3.1 Aufbau der LLK Datenstruktur

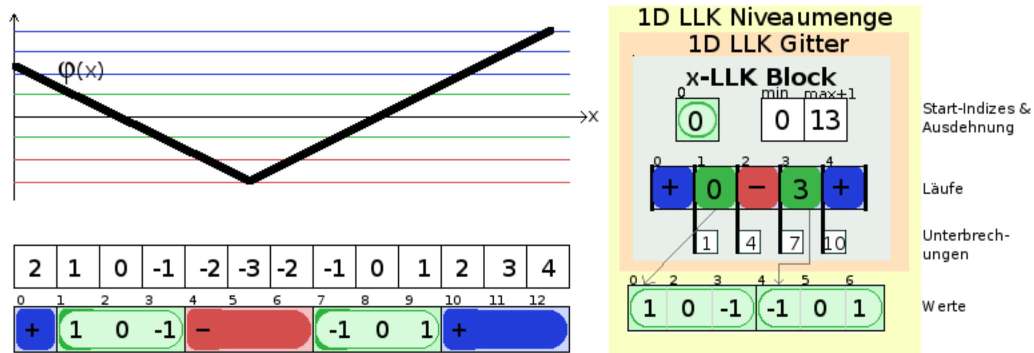


Abbildung 2.10: 1D LLK Niveaumenge

Die H-LLK Datenstruktur ist eine rekursive Datenstruktur. Sie besteht aus einem LLK Block, welcher die Läufe einer bestimmten Dimension kodiert, und einem Verweis auf einen LLK Block der nächst niedrigeren Dimension. Die niedrigste Dimension enthält außerdem noch ein Array für die Werte der innerhalb der definierten Läufe liegenden Gitterpunkte (Werte-Array). Diese Werte entsprechen dem euklidischen Abstand der Voxel zur Objektgrenze.

Ein LLK Block setzt sich aus folgenden Arrays zusammen:

- *Start Indizes:* beschreibt die Unterteilung des LLK Blocks in verschiedene Segmente. Das Array enthält Indizes, die als Zeiger auf den ersten Lauf des Segments im Läufe-Array dienen.

- *Ausdehnung:* die Werte *Min* und *Max* beschreiben die minimalen und maximalen (plus eins) Coordinaten entlang der Kodierungsachse. Durch diese Werte wird außerdem eine explizite Bounding Box des Objekts definiert.

- *Läufe:* beschreiben inwiefern jedes Segment in verschiedene Läufe unterteilt ist. Negative und positive Läufe bezeichnen komprimierte Läufe, während definierte Läufe einen Index enthalten, welcher in der niedrigsten Dimension als Zeiger in das Werte-Array dient und bei höheren Dimensionen auf das entsprechende Segment in den Start Indizes des nächsten hierarchisch untergeordneten LLK Blocks zeigt.

- *Unterbrechungen:* enthalten in geordneter Reihenfolge entsprechend der Läufe die absoluten Koordinaten entlang der Kodierungs Achse, an denen jeder Lauf beginnt, mit Ausnahme des ersten Laufs eines Segments, dessen Startkoordinate durch *Min* gegeben ist.

2.3.2 Aufbau der hierarchischen Datenstruktur

Durch den hierarchischen Aufbau der H-LLK Datenstruktur läßt sie sich für beliebige Dimensionen benutzen. Für jede Dimension wird immer nur in eine Koordinatenrichtung kodiert. Bei einem Objekt der Dimension n werden die Voxel zuerst in Richtung der ersten Kodierungsachse (z.B. entlang der x-Achse) kodiert. Das Ergebnis wird dann auf den Unterraum projeziert, der durch die verbliebenen $n-1$ Koordinatenachsen aufgespannt wird. Aus den Kodierungsreihen, welche definierte Daten enthalten, werden bei dieser Projektion definierte Gitterpunkte erzeugt. Vollständig innerhalb oder außerhalb liegende Kodierungsreihen werden in negative oder positive Gitterpunkte projeziert. Anschließend wird dieses neue Voxelgitter der Dimension $n-1$ entlang der nächsten Kodierungsachse kodiert. Auf diese Art wird weiter fortgefahren, bis man in der niedrigsten Dimension angekommen ist.

Der hierarchische Aufbau der Datenstruktur soll an einem Beispiel in 2D aufgezeigt werden. Die Datenstruktur besteht in diesem Beispiel aus zwei LLK Blöcken entsprechend der beiden Kodierungsachsen. Der obere Block kodiert das Objekt in Richtung der y-Achse und der untere Block in Richtung der x-Achse.

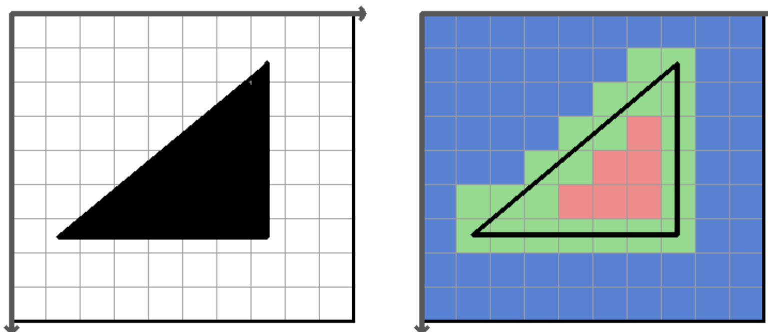
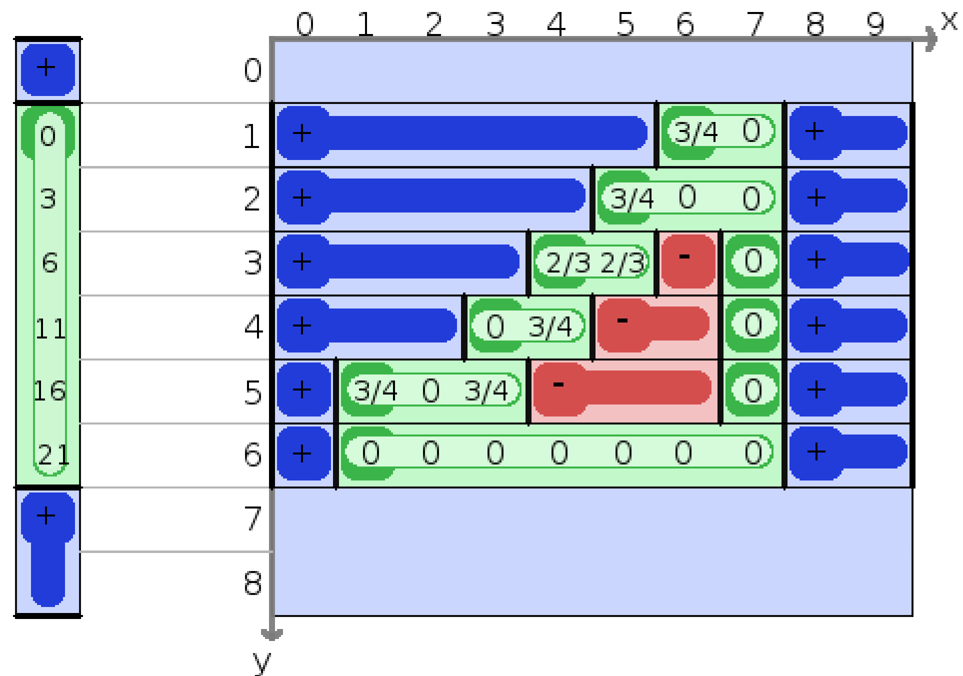


Abbildung 2.11: Ein 2D Objekt und seine klassifizierten Voxel

In Abbildung 2.11 ist ein Dreieck über einem Gitter abgebildet, direkt daneben die entsprechenden bereits klassifizierten Voxel. Außerhalb liegende Voxel werden Blau dargestellt, innerhalb liegende Rot. Die Voxel im Begrenzungsbereich sind Grün dargestellt.



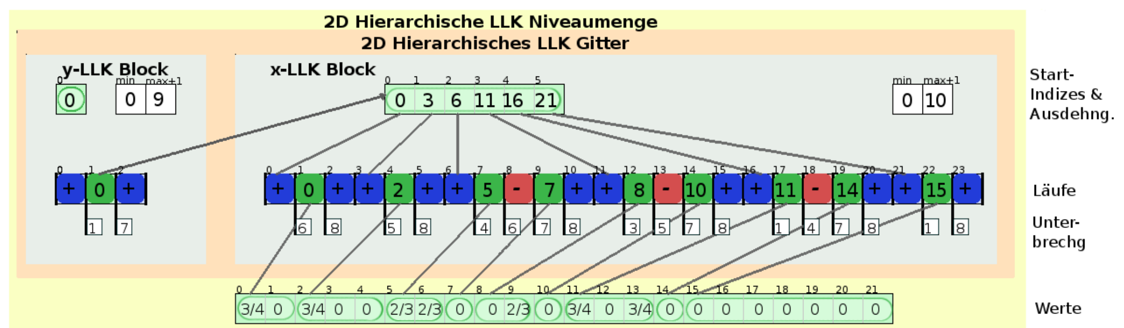


Abbildung 2.12: Kodierung eines 2D Objekts in eine 2D H-LLK Niveaumenge

Abbildung 2.12 zeigt eine schematische Darstellung der Kodierung dieses Objekts in eine H-LLK Niveaumenge. Die Kodierung entlang der x-Achse ist auf dem 12×12 Gitter und die Kodierung entlang der y-Achse auf dem 1×12 Gitter dargestellt. Jeder lineare Durchlauf der Lauflängenkodierung entlang der x-Achse wird im x-Block gespeichert. Ausgenommen sind davon jedoch Durchläufe, die keine definierten Läufe enthalten. Im Beispiel sind das

die Bereiche 0 und 7-8 bei der Kodierung der y-Achse. Diese wurden als komplett außerhalb klassifiziert und werden daher bei der Kodierung entlang der x-Achse nicht mit betrachtet. Somit werden Segmente die entlang einer bestimmten Kodierungsachse keinen definierten Lauf enthalten, also entweder vollständig innerhalb oder vollständig außerhalb sind, an die darüber liegende Dimension weitergegeben. Der höhere LLK Block wird entlang der y-Achse kodiert und ergibt den dargestellten y-LLK Block.
Die definierten Werte der y-Achsen Kodierung fungieren als Offsets für die entsprechenden Segmente der x-Achsen Kodierung. Die definierten Werte der x-Achsen Kodierung sind die eigentlichen Werte der Voxel und entsprechen dem euklidischen Abstand der Voxel zur Objektgrenze.

Das untere Bild in Abbildung 2.12 zeigt nun die aus der schematischen Darstellung resultierende 2D H-LLK Niveaumenge. Die definierten Werte der y-Achsen Kodierung werden dabei im Start Indizes Array des x-Blocks gepeichert. Die definierten Werte der x-Achsen Kodierung, also die eigentlichen Werte der Voxel, werden in einem speziellen Werte Array gespeichert.

Wie man sieht, speichern die LLK Blöcke die Läufe sequentiell. Die definierten Werte des x-Blocks sind Indizes in das Werte-Array und verweisen auf den Wert des Voxels eines definierten Laufs im Werte-Array. Die definierten Werte im y-Block hingegen verweisen auf die Start Indizes des x-Blocks. Somit ist erkennbar, daß der niedrigste LLK Block die Werte der Voxel kodiert, während alle höheren Blöcke das Ergebnis der vorangegangenen Lauflängenkodierung kodieren. Dadurch läßt sich die Datenstruktur sehr einfach für höhere Dimensionen verallgemeinern.

2.3.3 Direktzugriff

Ein schneller Zugriff auf einzelne Elemente ist für viele Applikationen der Computergrafik wichtig, wie zum Beispiel bei der Kollisionsdetektion oder beim Ray Tracing. Der Direktzugriff auf die H-LLK Datenstruktur arbeitet rekursiv:

(1) Der Direktzugriff auf dem obersten LLK Block wird mit zwei Parametern gestartet: einem Vektor mit den Koordinaten des gefragten Gitterpunkts $qv = (q_1, ..., q_n)$ und einem Segment Index *si*, welcher zu Beginn mit Null initialisiert wird. Von dem Koordinatenvektor wird nun die oberste Koordinate q_n abgespalten.

(2) Über den gegebenen Segment Index läßt sich nun der Index des ersten Laufs in dem Segment, die erste Unterbrechung sowie die Anzahl der Läufe in dem Segment (die Segmentlänge) bestimmen. Der Lauf, welcher die gesuchte Koordinate q_i enthält, läßt sich nun mittels Binärer Suche innerhalb der Grenzen des Segments bestimmen. Falls der Lauf negativ oder positiv ist, wird dessen Lauf-Kodierung zurückgeliefert. Ansonsten berechnet man den Index für die definierten Daten. Dabei dient der von dem

Lauf gelieferte Index als Offset und man erhält den gesuchten Index, indem man dieses Offset mit q_i addiert und die Koordinate der letzten Unterbrechung vor dem Lauf subtrahiert.

(3) Falls noch ein LLK Block in einer hierarchisch untergeordneten Dimension existiert, wird der Direktzugriff für diesen mit dem restlichen Koordinatenvektor und dem vorher berechneten Index als Segment-Index aufgerufen. Ansonsten wird der Index für die definierten Daten, der dann als Zeiger in das Werte-Array dient, zurückgegeben.

2.3.4 Sequenzieller Zugriff

Da viele Schemata zur Deformation von Oberflächen sequenziell auf eine Niveaumenge zugreifen, wurde die H-LLK Niveaumenge besonders für diese Anwendungen optimiert. Der sequenzielle Zugriff arbeitet ebenfalls rekursiv auf der Datenstruktur und iteriert entsprechend der Kodierungsachsen über die Daten:

Der sequenzielle Zugriff wird ähnlich wie der Direktzugriff als Prozeduraufruf im obersten LLK Block gestartet. Dabei werden zwei Parameter übergeben: ein Segment Index si initialisiert mit Null und ein leerer Vektor für Eltern-Koordinaten.

Die Prozedur beginnt nun, die Läufe des gegebenen Segments zu traversieren. Nicht definierte Läufe werden ignoriert. Wird ein definierter Lauf gefunden, werden seine Länge und seine Startkoordinate entlang der aktuellen Kodierungsachse bestimmt. Anschließend wird über alle Koordinaten des Laufs iteriert:

Zu jeder Koordinate wird der Index für die definierten Daten berechnet (siehe Abschnitt 2.3.3 Direktzugriff). Desweiteren wird ein Vektor der aktuellen Koordinaten erzeugt, welcher aus den jeweiligen Koordinatenvektoren besteht. Diesen Koordinatenvektor erhält man, indem der Vektor der Eltern-Koordinaten mit der jeweils aktuellen Koordinate vereint wird. Falls der aktuelle LLK Block der unterste ist, wird zum Ergebnis der aktuelle Koordinatenvektor und dessen zugehöriger Index der definierten Daten hinzu gefügt.
Ansonsten wird die Prozedur rekursiv für die nächst niedrigere Dimension unter Verwendung des Index für die definierten Daten als Segment-Index und dem aktuellen Koordinatenvektor als Eltern-Koordinatenvektor aufgerufen und der dadurch erhaltene Rückgabewert zum Ergebnis hinzu gefügt.

Der sequenzielle Zugriff liefert schließlich eine Liste aller definierten Gitterpunkte sowie ihre zugehörigen Indizes in das Werte-Array zurück.

2.3.5 Schneller Zugriff auf Nachbarn

Für zum Beispiel Skelettierungsalgorithmen wird ein effizienter Zugriff auf die Nachbarschaften eines Punktes benötigt. Die rekursive Struktur der H-LLK Niveaumenge läßt sich für eine effiziente Suche nach Nachbarn gut ausnutzen. In der niedrigsten Dimension können Nachbarn in konstanter Zeit ermittelt werden, da dazu nur das entsprechende Segment, in dem sich der gegebene Gitterpunkt befindet, durchsucht werden muß.
Bei höheren Dimensionen gestaltet sich die Suche ähnlich einfach. Die Nachbarn werden mittels Binärer Suche nach der Koordinate der niedrigsten Dimension in den durch die anderen Koordinaten angegebenen benachbarten Segmente ermittelt.

Als Beispiel wird gezeigt, wie die Achternachbarschaft N_8 im 2D anhand des Punktes (6,2) in Abbildung 2.12 ermittelt wird. Zunächst wird mittels Binärer Suche im Segment mit der y-Koordinate 2 nach dem Lauf gesucht, der die x-Koordinate 6 enthält. Auf die zwei Nachbarn in diesem Segment kann nun in konstanter Zeit mittels Inkrementieren bzw. Dekrementieren des Index in das Werte-Array auf deren definierte Werte zugegriffen werden, da sie ebenfalls im definierten Lauf von (6,2) liegen.
Für die benachbarten Segmente mit den y-Koordinaten 1 und 3 wird ähnlich verfahren. Für die Koordinate 1 wird festgestellt, daß in diesem Segment keine definierten Läufe existieren. Der Punkt (6,2) hat also dort keinen Nachbarn. Mittels Binärer Suche nach der x-Koordinate 6 im Segment wird der Punkte (6,3) als direkter Nachbar ermittelt. Für die Achternachbarschaft werden nun noch die Nachbarn dieses Punktes entlang der x-Achse benötigt. Auf diese kann wiederum in konstanter Zeit ausgehend von (6,3) zugegriffen werden.

2.3.6 Dilationsalgorithmus

Da sich eine direkte Dilation als ineffizient und langsam erwiesen hat, haben die Autoren der H-LLK Niveaumenge einen neuen Algorithmus entworfen, welcher den rekursiven Aufbau der Datenstruktur besser ausnutzt. Dieser Algorithmus arbeitet ebenfalls rekursiv und läßt sich folgendermaßen beschreiben.

Zu Beginn wird auf dem obersten LLK Block eine 1D Dilation ausgeführt. Um nun den nächst tieferen LLK Block zu dilatieren, werden die Koordinaten der definierten Läufe des dilatierten LLK Blocks iterativ durchgegangen.

Die aktuelle Koordinate sei y_m. Das korrespondierende LLK Segment der hierarchisch untergeordneten Dimension wird nun erzeugt, indem man zuerst alle Segmente im Bereich $y_m - n$ bis $y_m + n$ (wobei n die Anzahl der Dilations-Punkte ist) im originalen LLK Block dieser Dimension dilatiert.
Im zweiten Schritt wird nun aus diesen individuell erzeugten LLK Segmenten durch eine Vereinigung (Union) ein einzelnes dilatiertes Segment erzeugt, welches dann zum neuen di-

latierten LLK Block der hierarchisch untergeordneten Dimension hinzu gefügt wird.

Dieses Verfahren wird nun rekursiv auf alle LLK Blöcke in der hierarchisch untergeordneten Dimension angewandt und endet, wenn alle definierten Elemente des obersten LLK Blocks abgearbeitet sind.

Zum Abschluß der Dilation wird nun ein neues Werte-Array erzeugt. Bereits existierende Gitterpunkte erhalten ihren ursprünglichen Wert. Neue Gitterpunkte werden initialisiert.

Zum besseren Verständnis der Funktionsweise des Dilationsalgorithmus folgt nun eine genauere Beschreibung der 1D Dilation sowie des verwendeten Union-Algorithmus zum Vereinigen zweier H-LLK Datenstrukturen gleicher Dimension.

1D Dilation

Der 1D Dilationsalgorithmus ist recht einfach aufgebaut und besteht aus zwei Schritten:

Im ersten Schritt werden alle definierten Läufe des LLK Segments unabhängig dilatiert. Es wird also die Unterbrechung vor dem Lauf und die Unterbrechung nach dem Lauf (als dessen Grenzen) um die Anzahl der Dilationspunkte n verkleinert bzw. vergrößert.

Anschließend wird ein neues LLK Segment geformt, indem die dilatierten Läufe vereint werden. Die verbleibenden negativen oder positiven Läufe werden dabei konsistent mit dem originalen Segment gehalten. Wurde die minimale oder maximale Koordinate durch einen dilatierten definierten Lauf überschritten, gibt es zwei Möglichkeiten. Falls der erste bzw. letzte Lauf definiert war, wird Min bzw. Max einfach angepaßt. War der Lauf nicht definiert, wird ein entsprechender Lauf der Länge n an die entsprechende Stelle eingefügt.

Abbildung 2.13 zeigt ein Segment vor und nach der 1D Dilation bei $n = 1$. Die bereits existierenden definierten Punkte sind mit e gekennzeichnet, die neu hinzugefügten mit n.

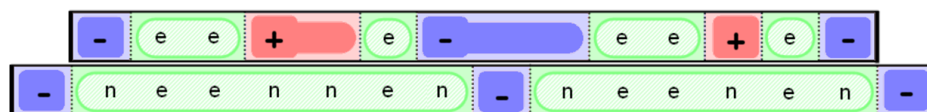


Abbildung 2.13: Die 1D Dilation veranschaulicht als Grafik

Union-Algorithmus

Die von der Dilation benötigte Vereinigung zweier H-LLK Datenstrukturen unter Berücksichtigung der korrespondierenden Segmente tieferer Dimensionen hat sich als recht kom-

plex erwiesen und wird daher zum besseren Verständnis hier beschrieben. Vom Prinzip her funktioniert eine Vereinigung von zwei H-LLK Datenstrukturen folgendermaßen.

Als Eingabe erhält der Algorithmus eine Liste von H-LLK Datenstrukturen der selben Dimension, die vereint werden sollen. Falls die Liste nur ein Element enthält, endet der Algorithmus und dieses eine Element wird unverändert zurückgegeben.

Es werden nun die ersten zwei Elemente aus der Liste betrachtet. Zu Beginn werden Min und Max dieser beiden LLK Segmente verglichen und das kleinere Min bzw. größere Max als Ausdehnung für das neue Segment gewählt. Nun werden die einzelnen Läufe der beiden Segmente iterativ miteinander verglichen. Endet der erste Lauf des einen Segments nach seinen Koordinaten noch bevor der erste Lauf des anderen Segments beginnt, werden dessen Läufe unverändert zum Ergebnis hinzugefügt, bis ein Lauf erreicht wurde, dessen Unterbrechung nach der Startkoordinate des ersten Laufs im anderen Segment liegt. Im Weiteren wird das neue Segment folgendermaßen gebildet.

Folgende drei Fälle müssen unterschieden werden:

1. Sind die beiden aktuellen Läufe der zwei Segmente definiert, wird ein definierter Lauf und die größte der beiden Unterbrechungen zum Rückgabe-Segment hinzu gefügt. Falls ein LLK Block hierarchisch untergeordneter Dimension existiert, werden die einzelnen Koordinaten zwischen der letzten und der aktuellen Unterbrechung des Rückgabesegments betrachtet. Dabei kann die Unterbrechung des kürzeren Laufs überschritten werden, so daß bei dem betroffenen Eingabesegment zum nächsten Lauf gewechselt werden muß.
   Falls beide Läufe der Eingabe-Segmente definiert sind, werden die zu der aktuellen Koordinate korrespondierenden hierarchisch untergeordneten LLK Segmente ermittelt und mittels der Union Operation vereint. Das Ergebnis wird dann zum LLK Block des Rückgabesegments hinzu gefügt. Ist einer der beiden Läufe nicht definiert, werden die korrespondierenden hierarchisch untergeordneten LLK Segmente des definierten Laufs unverändert ins Rückgabe-Segment übernommen.

2. Ist der Lauf des einen Segments definiert, der des anderen aber nicht, wird zum Rückgabe-Segment ein definierter Lauf sowie die Unterbrechung des LLK Segments, in dem sich dieser definierte Lauf befindet, hinzugefügt. Falls ein LLK Block in einer hierarchisch untergeordneten Dimension existiert, wird für diesen das korrespondierende Untersegment ermittelt und unverändert zum hierarchisch untergeordneten LLK Block des Rückgabesegments hinzugefügt.

3. Ist keiner der beiden Läufe definiert, wird die kleinste Unterbrechung zum Rückgabesegment hinzu gefügt. Ist ein Lauf davon negativ (innerhalb des Objekts), wird zum Rückgabe-Segment ein negativer Lauf hinzu gefügt, ansonsten ein positiver Lauf.

Es wird nun in den beiden Segmenten jeweils zum nächsten Lauf gesprungen, falls deren Unterbrechung kleiner oder gleich der letzten Unterbrechung im Rückgabe-Segment ist.
Ist die letzte Unterbrechung des Rückgabe-Segments größer als der Max-Wert eines der Eingabesegmente aber kleiner als der Max-Wert des anderen, so werden die verbleibenden Läufe von diesem größeren Segment zum Rückgabesegment unverändert hinzugefügt.

Existieren in der Liste noch weitere Elemente, wird nun das nächste LLK Segment aus der Liste mit dem durch den Algorithmus neu erzeugtem LLK Segment vereint. Dies wird solange fortgesetzt, bis die Liste leer ist.

3 Aufbau der erweiterten H-LLK Datenstruktur

In diesem Kapitel wird nun der Aufbau der erweiterten hierarchisch lauflängenkodierten Niveaumenge vorgestellt. Der Hauptunterschied zur normalen H-LLK Niveaumenge ist die Speicherung der Daten auf dem Datenträger und das zur Bearbeitung und Visualisierung nur Teile der Daten in den Hauptspeicher geladen werden müssen. Dadurch eignet sich die Datenstruktur besonders für sehr große Voxelmengen, deren Daten sonst nicht verwaltbar wären. Desweiteren ist die Datenstruktur besonders auf die Verwaltung von binären Voxelmengen optimiert. Im weiteren wird von der hier vorgestellten Datenstruktur als *erweiterte H-LLK Datenstruktur* gesprochen.

3.1 Vorverarbeitung der Daten

Da die zu verarbeitenden Daten zu Beginn meist in Form von Graustufenbildern vorliegen, müssen sie vor der Lauflängenkodierung noch binärisiert werden. Dies geschieht mittels eines *Schwellwertverfahrens*. Dabei unterscheidet man zum einen zwischen globalen und lokalen Schwellwertverfahren. Während bei einem globalen Schwellwertverfahren ein Wert für das komplette Bild gilt, wird es bei einem lokalen Schwellwert in kleine Teilbereiche aufgeteilt, für die dann unterschiedliche Schwellwerte verwendet werden können. Desweiteren verwenden einige Verfahren feste (meist vom Nutzer wählbare) Schwellwerte. Andere, sogenannte adaptive Schwellwertverfahren, verwenden optimierte Schwellwerte, die sich zum Beispiel nach dem am häufigsten auftretenden Grauwert richten.

An sich kann jedes Schwellwertverfahren verwendet werden. Der Einfachheit halber wird für die Vorverarbeitung ein globaler fester Schwellwert verwendet, welcher vorher vom Nutzer festgelegt wird. Das Grauwertbild wird daher zunächst in ein binäres kartesisches Gitter mit so vielen Einträgen, wie das Eingabebild Punkte besitzt, umgewandelt, an dessen jeweilige Positionen für Punkte, die einen Grauwert unterhalb des Schwellwertes haben, Null geschrieben wird und für die mit einem Grauwert über dem Schwellwert eine Eins. Dieses binäre kartesische Gitter wird dann an die Lauflängenkodierung weitergegeben.

Die Anzahl der Einsen wird während der Vorverarbeitung gezählt. Ist sie Null oder unterschreitet einen vorher festgelegten Wert, so wird ein entsprechender Status-Wert an die Lauflängenkodierung gegeben, daß sich in dem Bild keine relevanten Daten befunden haben.

3.2 Änderungen an der originalen H-LLK Datenstruktur

Gegenüber der originalen Datenstrukur gibt es aufgrund der in der Vorverarbeitung binärisierten Daten eine grundlegende Änderung, die im Wegfall des expliziten Werte-Array liegt. Die Voxel in den definierten Läufen können lediglich den Wert Eins annehmen, somit ist keine Speicherung der Werte für jeden einzelnen Voxel mehr notwendig und der dafür aufgewandte Speicherplatz kann eingespart werden.

Eine weitere Änderung ergibt sich in der Definition der drei Lauf-Typen. So wird als Datentyp für einen Lauf lediglich ein 4 Byte Integer verwendet. Die drei Lauftypen sind demnach wie folgt definiert.

$$\begin{array}{lcl} \text{Negativ} & = & -1 \\ \text{Positiv} & = & 2.147.483.647 \\ \text{Definiert} & = & [0, 2.147.483.647] \end{array}$$

Es sind demnach Indizes zwischen 0 und 2.147.483.646 für definierte Läufe möglich. Positive Läufe werden durch den beim verwendeten Datentyp, hier also den 4 Byte Integer, größt möglichen positiven Ganzzahlwert gekennzeichnet.

3.3 Dateiformat

Bei der Entwicklung eines passenden Formats zeigte sich, daß es viel zu unflexibel gegenüber Änderungen an den Daten ist, wenn das vollständige Objekt in einer Datei gespeichert wird. Unter bestimmten Umständen, zum Beispiel wenn bei der Erosion neue Läufe hinzugefügt werden müssen, reicht so der belegte Speicherbereich für einen Datensatz nach der Änderung nicht mehr aus, so daß zusätzliche Daten mittels Verweis am Ende der Datei gespeichert werden müßten, was den Zugriff auf die Daten verlangsamen würde. Auch gestaltet sich dadurch der Zugriff auf einzelne Teile des Objekts als umständlich und langsam, da der Dateizeiger zuerst zu der entsprechenden Stelle bewegt werden muß, an der der gesuchte Datensatz liegt, welches ein Iterieren durch die komplette Datei bis dahin erfordert.

Es wurde daher eine flexiblere Datenstruktur entwickelt, welche jedes y-Segment, also die lauflängenkodierten 2D Binärbilder, in einer seperaten Datei speichert. Eine Hauptdatei enthält dann die Kodierung entlang der z-Achse und verweist auf die entsprechenden Datensätze. Im Folgenden ist nun der Aufbau dieser zwei Dateitypen beschrieben.

Die Dateien enthalten die Daten blockweise gegliedert nach ihrem Typ in binär kodierter Form. Die einzelnen Blöcke enthalten den Bezeichner des Blocks als 4 Byte Zeichenmenge, die Anzahl der in diesem Block enthaltenen Daten als 2 Byte Integer und die eigentlichen Daten, alle als 2 Byte Integer. Den Aufbau der Hauptdatei veranschaulicht Abbildung 3.1.

MinZ <short int>
MaxZ <short int>
MinY <short int>
MaxY <short int>
MinX <short int>
MaxX <short int>
Runs <short int> (Anzahl)
<short int> (* Anzahl)
....
Brks <short int> (Anzahl)
<short int> (* Anzahl)
....

Abbildung 3.1: Aufbau der Hauptdatei

Wie man in Abbildung 3.1 sieht, enthält die Hauptdatei neben den bereits genannten Datenblöcken noch sechs weitere Bezeichner *Min* und *Max* für je x, y und z. Diese enthalten die Ausdehnung des Objekts entlang der entsprechenden Koordinatenrichtungen. Der Block mit dem Bezeichner *Runs* enthält die Läufe entlang der z-Achse, der Block mit dem Bezeichner *Brks* die entsprechenden Unterbrechungen.
Die Segmente der y-Kodierung werden in einer solchen Form als Dateien gespeichert, daß man sie über den Index des jeweiligen y-Segments aus der z-Kodierung adressieren kann. Sie enthalten das jeweilige kodierte y-Segment und die zu diesem Segment gehörigen Segmente aus der x-Kodierung. Ein Unterschied zur Hauptdatei liegt in der Verwendung von 4 Byte Integern für die Indizes der Läufe und der Anzahl der in jedem Array enthaltenen Daten. Den Aufbau dieser Segment-Dateien zeigt Abbildung 3.2.

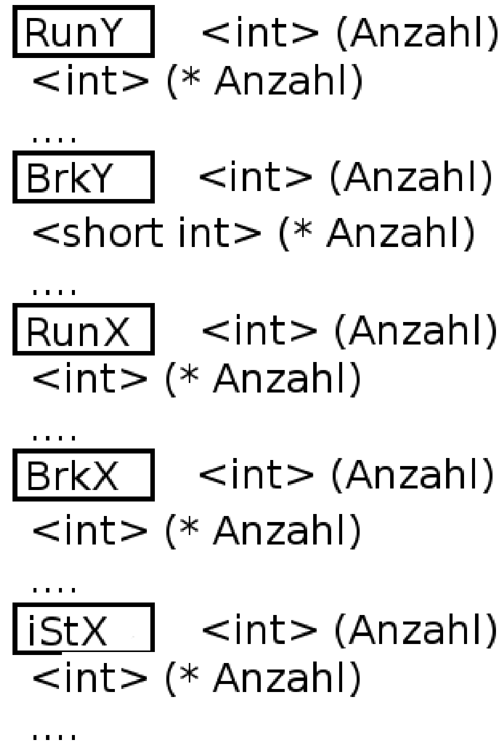


Abbildung 3.2: Aufbau der Segment-Dateien

Die Läufe der jeweiligen Kodierungen werden in den Blöcken *RunY* und *RunX* gepeichert, die Unterbrechungen in den Blöcken *BrkY* und *BrkX*. Außerdem enthalten diese y-Segment-Dateien noch einen Block mit dem Bezeichner *iStX*. Dieser enthält die Start-Indizes der enthaltenen x-Segmente.

3.4 Erzeugung der Daten

Um aus den einzelnen Graustufenbildern eine hierarchisch lauflängenkodierte Voxelmenge zu erzeugen, werden die in der Vorverarbeitung binarisierten Einzelbilder in der Reihenfolge ihrer Aufnahme lauflängenkodiert und zu der hierarchischen Struktur hinzugefügt.

Als Eingabe erhält der Algorithmus einen Verzeichnisnamen, in dem die Daten gespeichert werden sollen, sowie den Dateinamen des Bildes, welches zu der Datenstruktur hinzugefügt werden soll. Zunächst wird überprüft, ob sich in dem angegebenen Verzeichnis eine Hauptdatei befindet. Ist dies der Fall, wird sie in den Speicher geladen, ansonsten wird die z-Kodierung initialisiert: die Ausdehnungen *min* und *max* werden für alle drei Koordina-

tenachsen mit Null initialisiert und zum Array für die Segmentindizes wird ebenfalls Null hinzugefügt.

Anschließend wird die Vorverarbeitung für das gegebene Bild gestartet. Liefert diese zurück, daß das Bild keine relevanten Daten enthält, wird zur z-Kodierung ein nicht definierter Lauf hinzugefügt, falls der letzte Lauf definiert war, und *max* um Eins inkrementiert.
Ansonsten wird das Bild hierarchisch lauflängenkodiert und eine neue zweidimensionale H-LLK Datenstruktur erzeugt.

Zunächst werden die Ausdehnungen entlang der x-Achse und der y-Achse festgelegt. Dazu wird für beide Koordinatenachsen *min* mit Null initialisiert und *max* auf die Anzahl Punkte in der jeweiligen Koordinatenrichtung gesetzt. Die Lauflängenkodierung wird zunächst für jede y-Koordinate entlang der x-Achse durchgeführt. Enthält das so entstandene LLK Segment definierte Läufe, wird es zum x-Block der H-LLK Struktur hinzugefügt. Der Index des ersten Laufs dieses Segments wird zum Startindizes-Array des x-Blocks hinzugefügt.
Desweiteren wird für die Kodierung ein zusätzliches Array *yArray* für die y-Kodierung erzeugt. Es besitzt Einträge für jede mögliche y-Koordinate. Wurde nun ein neues Segment zum x-Block hinzugefügt, wird der entsprechende Index auf den Startindex dieses Segments zu *yArray* hinzugefügt. Ansonsten wird der Eintrag als nicht definiert gekennzeichnet.
Wurde die Kodierung entlang der x-Achse abgeschlossen, so wird nun das dabei erzeugte *yArray* lauflängenkodiert und daraus der entsprechende y-Block erzeugt.

Es werden nun noch die Ausdehnungen der neuen H-LLK Struktur entlang der y-Achse und der x-Achse mit den globalen Ausdehnungen aus der Hauptdatei verglichen und entsprechend angepaßt, falls sie von den Ausdehnungen des lauflängenkodierten Bildes überschritten werden. Zum Schluß wird aus den neuen Daten der z-Kodierung eine neue Hauptdatei erzeugt und in das angegebene Verzeichnis geschrieben. Wurde aus dem gegebenen Bild eine 2D H-LLK Struktur erzeugt, wird diese in Form einer neuen Segmentdatei in das angegebene Verzeichnis geschrieben.

3.5 Laden der Daten zur Bearbeitung

Der Algorithmus zum Laden der Daten erhält den Namen des Verzeichnisses, in dem der zu bearbeitende Datensatz liegt, als Eingabe. Zunächst werden nur die Daten aus der Hauptdatei in den Speicher geladen, die im Prinzip die Kodierung der z-Achse und die Ausdehnungen entlang der drei Koordinatenachsen umfassen.

Die Daten lassen sich zur Bearbeitung in zusammenhängenden Teilstücken laden. Daher ist es nicht notwendig, das komplette Objekt für die Bearbeitung im Speicher zu halten, was vorallem bei sehr großen Objekten von Vorteil ist. Es werden dazu nur die Teile des Daten-

satzes geladen, welche in einem vorher von der jeweiligen Operation festgelegten Bereich an z-Koordinaten $minZ \leq i \leq j \leq maxZ$ liegen.
Es wird nun eine völlig unabhängige H-LLK Datenstruktur aus den geladenen Daten erzeugt, auf der dann die jeweilige Operation angewandt wird. Dazu werden über die Indizes der definierten Läufe der z-Kodierung zwischen den Koordinaten $minZ \leq i \leq j \leq maxZ$ die Indizes der jeweiligen Segment-Dateien ermittelt, welche nacheinander geladen werden und zu dem bereits geladenen Teil des Datensatzes hinzugefügt werden. Das somit geladene Teilstück des Datensatzes wird im weiteren als *z-Block* bezeichnet.

4 Algorithmen auf der erweiterten H-LLK Datenstruktur

In diesem Kapitel wird die Implementation grundlegender Algorithmen der Bildverarbeitung auf der H-LLK Datenstruktur erläutert. Der Pseudo-Code zu sämtlichen hier vorgestellten Algorithmen befindet sich im Anhang.

4.1 Dilation

Wie in Kapitel 2.3.6 ersichtlich, weist der originale Dilationsalgorithmus der H-LLK Niveaumenge einen großen Nachteil auf: er erlaubt lediglich quadratische, vollständig belegte Strukturelemente, die das Objekt entlang jeder der sechs Koordinatenrichtungen (im 3D) um n Voxel verdicken. Da in der Praxis auch beliebig belegte Strukturelemente verwendet werden, war es notwendig, einen Dilationsalgorithmus für die Datenstruktur zu entwickeln, welcher beliebige Strukturelemente erlaubt, aber trotzdem optimal an die rekursive Struktur der H-LLK Niveaumenge angepaßt ist, um die Dilation effizient und in möglichst kurzer Zeit durchführen zu können.

Die Idee dabei ist, das Strukturelement ebenfalls durch hierarchische Lauflängenkodierung zu komprimieren. Dadurch läßt es sich einfacher für die einzelnen Untersegmente zerlegen, so daß ein ähnliches Vorgehen wie im originalen Algorithmus möglich ist. Nachfolgend werden die Anpassungen der 1D Dilation und der N-D Dilation beschrieben.

4.1.1 1D Dilation

Der 1D Dilationalgorithmus erhält als Eingabe das passende Segment aus der Kodierung des Strukturelements, die Koordinate des Bezugspunkts in diesem Segment sowie einen Segment-Index auf das Segment in der H-LLK Datenstruktur, das dilatiert werden soll. Die Koordinate des Bezugspunkts dient dabei zur Anpassung des Strukturelements auf den aktuellen definierten Punkt, der während der Dilation gerade betrachtet wird.

Zu Beginn des Algorithmus wird überprüft, ob das Strukturelement an der Koordinate des Bezugspunkts definiert ist. Ist dies der Fall, werden in das Ergebnis der Dilation die definierten Punkte des originalen Segments mit aufgenommen, ansonsten werden nur die durch die Dilation neu hinzugefügten Punkte als Ergebnis zurückgeliefert.

Es werden nun alle Läufe im originalen Segment betrachtet. Ist der Lauf definiert, wird das Strukturelement für jede definierte Koordinate angepaßt, so daß der Bezugspunkt die gleiche Koordinate wie die aktuell betrachtete Koordinate hat. Dabei werden die Ausdehnungen *min* und *max* und die Unterbrechungen der Läufe des Strukturelements passend zur neuen Koordinate des Bezugspunkts verändert, so daß das Strukturelement konsistent bleibt.
Anschließend wird das geänderte Strukturelement entweder mit dem originalen Segment oder mit einem leeren Segment[1] vereint, falls der Bezugspunkt im Strukturelement definiert war. Das Ergebnis wird schließlich zur Rückgabe, dem dilatierten 1D Segment, hinzugefügt. Dieses Vorgehen wird nun für jede definierte Koordinate mit dem jeweils angepaßten Strukturelement durchgeführt und das jeweilige Ergebnis mit dem dilatierten 1D Segment vereint. Wurden alle definierten Koordinaten des Segments bearbeitet, wird das dilatierte 1D Segment zurückgegeben.

4.1.2 N-D Dilation

Bei der N-D Dilation wird, beginnend mit dem obersten LLK Block, zuerst mittels der 1D Dilation das entsprechende dilatierte Segment für diesen Block gebildet. Anschließend werden die dimensional niedrigeren Segmente 1D dilatiert und die Dilation rekursiv auf den einzelnen Segmenten fortgesetzt, falls dies noch nicht der hierarchisch niedrigste Block der H-LLK Niveaumenge war. Nachfolgend ist der Dilationaalgorithmus im Detail beschrieben.

Die N-D Dilation erhält als Eingabe ein Strukturelement der Dimension N, einen Segment Index *si*, welcher zu Beginn der Dilation mit Null initialisiert ist, sowie den Koordinatenvektor *qv*, welcher die Koordinaten des Bezugspunkts im Strukturelement enthält.
Es wird nun auf das Segment mit dem Index *si* die 1D Dilation angewandt. Anschließend werden die Läufe des dilatierten Segments betrachtet. Ist der Lauf definiert, werden die einzelnen Koordinaten des Laufs betrachtet und die Dilation rekursiv für alle durch das Strukturelement einbezogenen Koordinaten, ausgehend von der aktuellen Koordinate, fortgeführt.

Sei q_i die Koordinate des Bezugspunkts im aktuellen Segment und y_m die aktuell betrachtete definierte Koordinate im dilatierten Segment. Desweiteren sei min_{SE} die minimale und max_{SE} die maximale Ausdehnung des Strukturelements sowie *min* die minimale und *max* die maximale Ausdehnung des originalen H-LLK Segments entlang der Koordinatenrichtung der Dimension N, dann ist

$$D_S = \begin{cases} y_m - max_{SE} - q_i - 1 & \text{wenn } y_m - max_{SE} - q_i - 1 \geq min \\ min & \text{sonst} \end{cases}$$

$$D_E = \begin{cases} y_m + q_i - min_{SE} + 1 & \text{wenn } y_m + q_i - min_{SE} + 1 \leq max \\ max & \text{sonst} \end{cases}$$

[1] Damit ist ein 1D Segment gemeint, welches lediglich einen positiven Lauf enthält und die Ausdehnungen des originalen Segments besitzt.

Es werden nun die Koordinaten im Bereich D_S bis D_E betrachtet. Befindet sich die aktuell betrachtete Koordinate k im originalen Segment in einem definierten Lauf, wird das zugehörige Segment in der nächst niedrigeren Dimension zuerst 1D-dilatiert und anschließend die Dilation auf diesem Untersegment rekursiv fortgesetzt. Für die Dilation der dimensional niedrigeren Segmente dient dabei das entsprechende Untersegment mit der Koordinate k_{se} im Strukturelement. Die Koordinate wird zu Beginn mit

$$k_{se} = \begin{cases} q_i + y_m - min & \text{wenn } y_m - max_{SE} - q_i - 1 < min \\ max & \text{sonst} \end{cases}$$

initialisiert und in jeder Iteration um Eins dekrementiert. Das Untersegment mit der Koordinate k wird aber nur dann dilatiert, wenn im Strukturelement der Lauf, in dem die Koordinate k_{se} liegt, definiert ist.
Wenn die Iteration über die einzelnen Koordinaten abgeschlossen ist, werden die einzelnen dilatierten Untersegmente vereint und das Ergebnis wird zum hierarchisch untergeordneten LLK Block des dilatierten Segments hinzugefügt.

Der Algorithmus endet, wenn alle definierten Koordinaten des dilatierten Segments betrachtet wurden. Es wird schließlich die dilatierte H-LLK Datenstruktur zurückgegeben.

4.2 Erosion

Die Erosion läßt sich ähnlich wie die Dilation gestalten, also optimal an die hierarschiche Struktur der H-LLK Niveaumenge anpassen. Der grundlegende Unterschied ist dabei, daß das Strukturelement dazu dient zu entscheiden, ob ein Punkt beibehalten werden kann, oder ob er aus der Datenstruktur gelöscht werden muß. Ein Punkt wird genau dann beibehalten, wenn, ausgehend vom aktuell betrachteten Punkt in der Datenstruktur und dem Bezugspunkt im Strukturelement, alle definierten Punkte im Strukturelement auch im originalen Segment definiert sind. Ist dies nicht der Fall, wird der Punkt gelöscht.

Ebenso wie bei der Dilation, ist das Strukturelement bei der Erosion hierarchisch lauflängenkodiert. Der Algorithmus unterteilt sich demnach in eine 1D Erosion, welche ein einzelnes eindimensionales Segment erodiert, und eine N-D Erosion, welche dann das komplette Objekt erodiert. Desweiteren werden die einzelnen erodierten Nachbarsegmente, welche vom Strukturelement, ausgehend vom aktuell betrachteten Segment, mit einbezogen werden, nicht wie bei der Dilation vereint, sondern es wird die Schnittmenge daraus gebildet. Im folgenden werden nun die Algorithmen für die 1D Erosion, die N-D Erosion sowie die Bildung der Schnittmenge beschrieben.

4.2.1 1D Erosion

Der Algorithmus für die 1D Erosion erhält, wie sein Pendant bei der Dilation, das passende Segment aus der Kodierung des Strukturelements, die Koordinate des Bezugspunkts für die aktuelle Kodierungsachse q_i sowie den Index si auf das zu bearbeitende Segment. Ist q_i im Strukturelement nicht definiert, wird nun über sämtliche Koordinaten des Segments iteriert, wobei das Strukturelement mit Hilfe des Bezugspunkts an die aktuelle Koordinate angepaßt wird. Ansonsten wird lediglich über die Koordinaten der definierten Läufe iteriert.

Sei k die während der Iteration aktuell betrachtete Koordinate. Das entsprechend angepaßte Segment des Strukturelements wird nun mit dem Segment mit dem Index si verglichen. Die Anpassung des Strukturelements erfolgt dabei genauso wie bei der 1D Dilation. Sind alle im Strukturelement definierten Koordinaten auch im Segment definiert, so wird im Ergebnis die Koordinate k als definiert gesetzt, also ein definierter Lauf mit Unterbrechung $k+1$ hinzugefügt bzw. falls der letzte Lauf im Ergebnissegment definiert war, die Unterbrechnung angepaßt. Ansonsten gehört die Koordinate zu einem nicht definierten Lauf.

Ist die Koordinate nicht definiert, wird folgendermaßen entschieden, ob der entsprechende Lauf negativ oder positiv ist. War der letzte Lauf im Ergebnissegment nicht definiert, wird einfach dessen Unterbrechnung angepaßt. Ansonsten wird überprüft, in was für einem Lauf sich die Koordinate im originalen Segment befindet. War der Lauf definiert oder negativ, wird ein negativer Lauf mit Unterbrechung an Position $k+1$ hinzugefügt. War er positiv, wird ein positiver Lauf mit Unterbrechung an Position $k+1$ hinzugefügt.

Ist die Iteration über die Koordinaten abgeschlossen, wird noch überprüft, ob der letzte Lauf im Segment negativ ist. Ist dies der Fall, wird er durch einen positiven Lauf ersetzt. Das somit erzeugte erodierte Segment wird schließlich zurückgegeben.

4.2.2 N-D Erosion

Die N-D Erosion arbeitet rekursiv auf allen Segmenten der H-LLK Datenstruktur. Es werden, beginnend mit dem einen Segment des obersten LLK Block der Datenstruktur, die einzelnen Segmente der LLK Blöcke betrachtet. Befindet sich das aktuell betrachtete Segment nicht in der niedrigsten Dimension der Datenstruktur, so wird über alle Koordinaten des Segments iteriert. Dabei werden für jede Koordinate alle durch des Strukturelement mit einbezogenen Nachbarkoordinaten betrachtet und aus der Schnittmenge ihrer entsprechenden erodierten Untersegmente ein neues erodiertes Untersegment für die aktuell betrachtete Koordinate erzeugt. Enthält das neu erzeugte Untersegment definierte Daten, so wird die aktuell betrachtete Koordinate für das erodierte Segment als definiert gesetzt, ansonsten als nicht definiert. Befindet sich das aktuell betrachtete Segment hingegen in der niedrigsten Dimension, so wird das entsprechende erodierte Segment mit Hilfe des 1D Erosionsalgorithmus erzeugt

und dessen Ergebnis zurückgegeben.

Der N-D Erosionsalgorithmus bekommt als Eingabe ein Strukturelement der Dimension N, die Koordinaten des Bezugspunkts im Strukturelement in Form des Koordinatenvektors qv sowie einen Segment-Index auf das aktuell zu bearbeitende Segment si, welcher zu Beginn mit Null initialisiert ist.

Sei $q_i \in qv$ die Koordinate des Bezugspunkts im aktuellen Segment. Es wird nun das Segment mit dem Index si betrachtet. Befindet sich der Algorithmus im dimensional niedrigstem LLK Block der Datenstruktur, wird von dem Segment unter Verwendung des angegebenen Strukturelements die 1D Erosion berechnet und das entsprechende erodierte Segment zurückgegeben.
Ansonsten wird über die einzelnen Läufe des Segments iteriert. Handelt es sich bei dem aktuell betrachteten Lauf um einen nicht definierten Lauf, so wird dieser einfach zum erodierten Segment hinzugefügt, falls q_i im Strukturelement definiert ist. Ist dies nicht der Fall oder es handelt sich um einen definierten Lauf, wird über dessen Koordinaten iteriert.

Sei y_m die aktuell betrachtete Koordinate im aktuellen definierten Lauf des Segments. Desweiteren seien min_{SE} die minimale und max_{SE} die maximale Ausdehnung des Strukturelements sowie min die minimale und max die maximale Ausdehnung des originalen H-LLK Segments entlang der Koordinatenrichtung der Dimension N, dann ergeben sich die erste Koordinate E_S und die letzte Koordinate E_E, die durch das Strukturelement einbezogen werden, folgendermaßen:

$$E_S = \begin{cases} y_m - q_i - min_{SE} & \text{wenn } y_m - q_i - min_{SE} \geq min \\ min & \text{sonst} \end{cases}$$

$$E_E = \begin{cases} y_m + max_{SE} - q_i & \text{wenn } y_m + max_{SE} - q_i \leq max \\ max & \text{sonst} \end{cases}$$

Überschreiten E_S und E_E die minimale bzw. maximale Ausdehnung des Segments, so werden die Läufe für die entsprechenden außerhalb liegenden Koordinaten als nicht definiert angenommen.

Es wird nun über die Koordinaten im Bereich E_S bis E_E iteriert. Die entsprechende Koordinate k_{se} im Strukturelement, welche zu Beginn der Iteration mit min_{SE} initialisiert ist, wird mit jedem Schritt um Eins inkrementiert. Man unterscheidet zwischen zwei zu betrachtenden Fällen.

Ist die k_{se} im Strukturelement definiert, die aktuell betrachtete Koordinate k im originalen Segment hingegen nicht, so wird die Iteration abgebrochen und die Koordinate y_m im erodierten Segment als nicht definiert gekennzeichnet. Wenn der letzte Lauf im erodierten Seg-

ment bereits nicht definiert war, wird lediglich dessen Unterbrechung angepaßt. Ansonsten wird ein negativer Lauf hinzugefügt, falls $y_m - 1$ im erodierten Segment definiert war, bzw. ein positiver Lauf, falls noch kein Lauf zum erodierten Segment hinzugefügt wurde, falls y_m im originalen Segment definiert war. Ansonsten wird der entsprechende Lauf aus dem originalen Segment hinzugefügt. Anschließend wird mit der Koordinate $y_m + 1$ fortgefahren.

Ist k_{se} im Strukturelement definiert und die aktuelle Koordinate k im betrachteten Segment ebenfalls, so wird die N-D Erosion mit dem entsprechenden Untersegment und dem Untersegment der Koordinate k_{se} aus dem Strukturelement aufgerufen und das entsprechende erodierte Untersegment erzeugt. Enthält das Untersegment nach der Erosion keine definierten Koordinaten mehr, so wird es verworfen und zum erodierten Segment wird wie vorher beschrieben ein nicht definierter Lauf hinzugefügt.

Wurde die Iteration über die Koordinaten von E_S bis E_E abgeschlossen, so wird die Koordinate y_m im erodierten Segment als definiert gesetzt. Das entsprechende erodierte Untersegment, welches noch zum dimensional niedrigeren LLK Block der erodierten H-LLK Datenstruktur hinzugefügt wird, ergibt sich aus der Schnittmenge der einzeln berechneten erodierten Untersegmente der vorangegangenen Iteration.

Der Algorithmus endet, wenn alle Koordinaten des Segments betrachtet wurden. Zum Schluß wird noch überprüft, ob der letzte Lauf im Segment negativ ist. Ist dies der Fall, wird er durch einen positiven Lauf ersetzt. Die erodierte H-LLK Datenstruktur wird schließlich zurückgegeben.

4.2.3 Algorithmus für die Berechnung der Schnittmenge mehrerer LLK Segmente

Bei der Schnittmenge zweier H-LLK Datenstrukturen beliebiger Dimension, werden in das Ergebnis-Segment nur die Koordinaten der jeweiligen Segmente zu einem definierten Lauf hinzugefügt, die in beiden Segmenten definiert sind. Siehe dazu als Beispiel die Schnittmenge zweier eindimensionaler Segmente in Abbildung 4.1.

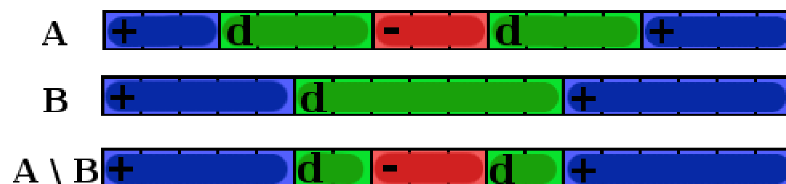


Abbildung 4.1: Schnittmenge zweier LLK Segmente

Der Algorithmus zur Berechnung der Schnittmenge gestaltet sich ähnlich wie der Algo-

rithmus für die Union zweier H-LLK Datenstrukturen. Als Eingabe erhält der Algorithmus ebenfalls eine Liste von mehreren N-dimensionalen H-LLK Datenstrukturen.

Es werden zu Beginn die ersten zwei Elemente der Liste betrachtet. Als minimale Ausdehnung *min* und maximale Ausdehnung *max* für das Rückgabesegment wird die größte minimale Ausdehnung der beiden Eingabe-Segmente bzw. die kleinste maximale Ausdehnung gewählt. Anschließend wird über die Läufe der beiden Segmente zwischen den Koordinaten *min* und *max* iteriert. Das Rückgabesegment wird dabei aufgebaut, indem immer zwei Läufe aus den beiden aktuellen Eingabesegmenten verglichen werden. Dabei müssen folgende zwei Fälle unterschieden werden.

Haben beide aktuell betrachteten Läufe unterschiedliche Typen oder sind beide nicht definiert, so wird ein nicht definierter Lauf hinzugefügt. Als Unterbrechnung wird die kleinste der beiden Unterbrechungen hinzugefügt, falls diese kleiner als *max* ist. Der Typ des nicht definierten Laufs wird dabei nach folgender Priorität entschieden. Ist einer der beiden Läufe definiert, so wird der Typ des nicht definierten Laufs verwendet. Sind beide nicht definiert, wird ein negativer Lauf hinzugefügt, falls mindestens einer der beiden Läufe negativ ist. Ansonsten wird ein positiver Lauf hinzugefügt.

Sind dagegen beide Läufe definiert, wird ein definierter Lauf und die kleinste der beiden Unterbrechungen zum Rückgabesegment hinzugefügt, falls diese kleiner als *max* ist. Falls ein LLK Block hierarchisch niedrigerer Dimension existiert, werden die einzelnen Koordinaten zwischen der letzten und der aktuellen Unterbrechung des Rückgabesegments betrachtet. Für jede Koordinate werden die jeweiligen zwei Untersegmente der Eingabesegmente ermittelt und die Schnittmenge daraus gebildet. Das so gebildete neue Untersegment wird dann zum hierarchisch untergeordneten LLK Block des Rückgabesegments hinzugefügt.

Ist die zuletzt hinzugefügte Unterbrechung im Rückgabesegment größer oder gleich der Unterbrechung des aktuell betrachteten Laufs in einem der Eingabesegmente, so wird von dem jeweiligen Segment der nächste Lauf betrachtet. Die Iteration endet, wenn eines der beiden Eingabesegmente den letzten Lauf erreicht hat und sich demnach keine weiteren Koordinaten überschneiden können. Zum Schluß wird noch überprüft, ob der letzte Lauf im Rückgabesegment negativ ist. Ist dies der Fall, wird er durch einen positiven Lauf ersetzt.

Wurden alle sich überschneidenden Koordinaten der beiden Segmente betrachtet, wird das neu gebildete Rückgabesegment mit dem nächsten Element der Liste aus der Eingabe verglichen. Der Algorithmus endet, wenn die Liste keine weiteren Elemente enthält.

4.3 Morphologische Operationen auf der erweiterten H-LLK Datenstruktur

Opening und Closing bestehen beide lediglich aus dem Nacheinanderausführen der Erosion und Dilation auf die Daten. Beide Algorithmen sind demnach ähnlich aufgebaut, es unterscheidet sich lediglich die Reihenfolge, in der Erosion und Dilation aufgerufen werden. Daher sei hier nun das allgemeine Vorgehen bei der Anwendung der morphologischen Operatoren auf die erweiterte H-LLK Datenstruktur beschrieben.

Als Eingabe erhalten die Algorithmen ein lauflängenkodiertes Strukturelement der Dimension Drei und einen Koordinatenvektor qv. Zur Bearbeitung der Daten wird die Kodierung der z-Achse in Blöcke mit einer vorher vom Nutzer festgelegten Anzahl z-Koordinaten aufgeteilt, welche nachfolgend als *z-Blöcke* bezeichnet werden. Über die Anzahl dieser z-Blöcke wird nun iteriert. Bei jedem Block, außer beim ersten und letzten, muß außerdem noch ein ausreichend breiter Überlappungsbereich aus dem vorangegangenen und dem nachfolgenden Block für die Bearbeitung geladen werden, welcher sich nach dem Strukturelement richtet. Bevor die jeweiligen z-Blöcke allerdings geladen werden, wird ein Backup der gesamten Datenstruktur erzeugt.

Sei $zBlock_k$ der k-te Block in der Iteration. Seine erste Koordiante sei min_k und die letzte plus Eins max_k sowie min die minimale und max die maximale Ausdehnung des Datensatzes in z-Richtung. Desweiteren sei q_z die z-Koordinate des Bezugspunkts und $min_{SE}(z)$ die minimale und $max_{SE}(z)$ die maximale Ausdehnung des Strukturelements entlang der z-Koordinate. Die erste zu ladende z-Koordinate wird mit Z_S und die letzte mit Z_E bezeichnet. Dann sind

$$Z_S = \begin{cases} min_k - q_z - min_{SE}(z) & \text{wenn } q_z - min_{SE}(z) \geq min \\ min & \text{sonst} \end{cases}$$

$$Z_E = \begin{cases} max_k + max_{SE}(z) - q_z & \text{wenn } max_{SE}(z) - q_z \leq max \\ max & \text{sonst} \end{cases}$$

Es wird nun mittels der in Abschnitt 3.5 beschriebenen Methode zum Laden der Daten der z-Block mit den Koordinaten Z_S bis $Z_E - 1$ in den Speicher geladen und auf diesem die jeweilige Operation ausgeführt. Für das Opening wird nun zuerst mit dem gegebenen Strukturelement die Erosion mit dem geladenen Teil der Daten durchgeführt und anschließend mit dem selben Strukturelement die Dilation auf das Ergebnis der Erosion angewandt. Für das Closing erfolgt dies in umgekehrter Reihenfolge.

Ist die Bearbeitung abgeschlossen, werden die im z-Block enthaltenen bearbeiteten Segmente der y-Kodierung zurückgeschrieben, bis auf den Überlappungsbereich. Es werden also nur die y-Segmente im Bereich min_k bis max_k geschrieben. Anschließend wird der nächste

z-Block betrachtet, bis alle Blöcke betrachtet wurden. Außerdem wird in jeder Iteration ein Teil des bearbeitenden z-Segments erzeugt, welcher dem obersten LLK Block des gerade betrachteten z-Blocks innerhalb der Koordinaten min_k max_k entspricht. Dieses neu erzeugte z-Segment wird nach Abschluß der Iteration als neue Hauptdatei zurückgeschrieben. Zu letzt wird dann diese neue Hauptdatei in den Speicher geladen.

4.4 Ausdünnung/Skelettierung

Der implementierte Skelettierungsalgorithmus orientiert sich am Algorithmus von Tsao und Fu (siehe Kapitel 2.2.2). Es handelt sich demnach um ein Zweipaßverfahren, wobei im ersten Paß die mediale Oberfläche und im zweiten Paß die mediale Achse erzeugt wird. Beide Pässe unterscheiden sich nur in der Form der “Simplizität in der Ebene”-Bedingung wie in Kapitel 2.2.2 beschrieben. Der zweite Paß zur Berechnung der medialen Achse erhält als Eingabe die mediale Oberfläche.
Der Algorithmus löscht nach folgenden Kriterien solange Punkte aus der Datenstruktur, bis das jeweilige Skelett erzeugt wurde und keine weiteren Punkte gelöscht werden können. Zu Beginn des Algorithmus werden alle definierten Punkte in der H-LLK Datenstruktur ermittelt. Anschließend werden in einer festgelegten Reihenfolge von Unterzyklen entsprechend der sechs Koordinatenrichtungen alle gerichteten Randpunkte der jeweils aktuellen Richtung betrachtet. Die Reihenfolge sei Unten (-z), Oben (+z), Osten(-x), Westen(+x), Süden(-y), Norden(+y). Dazu wird das zu Beginn ermittelte Array mit den Koordinaten der definierten Punkte durchlaufen.

Für den aktuell betrachteten Punkt wird zunächst anhand eines *gelöscht*-Flags überprüft, ob er bereits aus der Datenstruktur gelöscht wurde. Falls der Punkt noch nicht gelöscht wurde, wird anschließend überprüft, ob der Punkt die notwendigen Kriterien erfüllt, damit er überhaupt aus der Datenstruktur gelöscht werden darf. Dazu muß der Punkt zunächst ein Randpunkt der aktuellen Richtung sein. Desweiteren darf er nicht final, also kein Kurvenendpunkt, sein und muß der Definition eines simplen Punktes entsprechen. Erfüllt er nun auch noch die “Simplizität in der Ebene”-Bedingung, so wird seine Koordinate zu einem Array mit den zu löschenden Punkten hinzugefügt und sein *gelöscht*-Flag gesetzt.
Wurden alle definierten Punkte betrachtet, werden die ermittelten zu löschenden Punkte aus der Datenstruktur entfernt, und ein Flag gesetzt, daß Punkte während diesem Unterzyklus gelöscht wurden. Anschließend wird mit dem nächsten Unterzyklus fortgefahren.

Wurden alle sechs Unterzyklen durchlaufen, beginnt der Algorithmus von neuem, wenn Punkte in einem der Unterzyklen entfernt wurden. Ansonsten endet der Algorithmus und das jeweilige Skelett wurde erzeugt.

Bei der Skelettierung in der erweiteten H-LLK Datenstrukur reicht es, einen kleinen Über-

lappungsbereich von einem Voxel Breite aus den benachbarten z-Blöcken zu laden, damit man die Nachbarschaften der im Grenzbereich liegenden Punkte berechnen kann. Ebenso wie in Abschnitt 4.3 beschrieben, wird zu Beginn ein Backup der Daten erzeugt. Anschließend werden die einzelnen z-Blöcke einer vorher festgelegten Größe zur Bearbeitung geladen, der Skelettierungsalgorithmus auf jeden Block separat angewandt und die bearbeiteten Daten zurückgeschrieben.

4.4.1 Ermittlung der Nachbarschaften

Wie aus den vorangegangenen Erläuterungen ersichtlich, werden für die Berechnung der medialen Oberfläche und der medialen Achse zwei Nachbarschaftsbeziehungen benötigt, nämlich die N_6 Nachbarschaft und die N_{26} Nachbarschaft. Beide Algorithmen erhalten als Eingabe die Koordinaten des Punktes, von dem die Nachbarn ermittelt werden sollen, in Form einen Koordinatenvektors qv sowie einen Segmentindex si. Als Rückgabe erhält man ein Array mit Boolean-Werten, welche angeben, ob der Punkt an der jeweiligen Position einen definierten Nachbarn besitzt. Zu beachten ist bei dem Algorithmus für die N_{26} Nachbarschaft noch, daß eigentlich die N'_{26} Nachbarschaft ermittelt wird (siehe Kapitel 2.2.2), also das Rückgabe-Array auch den gegebenen Punkt mit enthält.

Ermittlung der N_6 Nachbarschaft

Der Algorithmus startet auf dem obersten LLK Block. Sei q_i die oberste Koordinate des Koordinatenvektors qv. Es wird nun ermittelt, ob sich die Koordinaten $q_i - 1$ und $q_i + 1$ in definierten Läufen befinden. Ist dies der Fall und es gibt weitere dimensional untergeordnete LLK Blöcke, so wird der entsprechende Segmentindex der jeweiligen Koordinate für den nächst niedrigeren LLK Block ermittelt und mit Hilfe dieses Index und den verbliebenen Koordinaten von qv mittels Direktzugriff überprüft, ob der jeweilige Nachbarpunkt definiert ist. Ist der Punkt definiert, wird zum Ergebnis Eins hinzu gefügt, ansonsten Null.
Existieren weitere dimensional niedrigere LLK Blöcke, wird anschließend der Segmentindex im nächst niedrigeren LLK Block für die Koordinate q_i ermittelt und die Suche nach Nachbarn rekursiv auf dem nächsten LLK Block fortgesetzt. Das Ergebnis der Rekursion wird dann zur Rückgabe hinzugefügt. Im 3D werden die Nachbarn schließlich in folgender Reihenfolge zurückgegeben: $-z$, $+z$, $-y$, $+y$, $-x$ und $+x$.

Ermittlung der N_{26} Nachbarschaft

Der Algorithmus zur Ermittlung der N_{26} Nachbarschaft arbeitet ähnlich. Beginnend mit dem obersten LLK Block wird zunächst überprüft, ob sich die Koordinaten $q_i - 1$ und $q_i + 1$ sich in definierten Läufen befinden. Ist dies der Fall, wird mit dem jeweiligen berechneten Segmentindex dieser beiden Koordinaten und dem verbleibenden Koordinatenvektor qv der Algorithmus rekursiv für den nächst niedrigeren LLK Block aufgerufen. Ist dies nicht

der Fall, so wird das Rückgabe-Array mit entsprechend vielen Nullen aufgefüllt. Ansonsten wird das Ergebnis der Rekursion zum Rückgabe-Array hinzugefügt.
Desweiteren wird noch für den der Koordinate q_i entsprechenden Segmentindex die Rekursion gestartet und das Ergebnis zur Rückgabe hinzugefügt. Diese Rekursion wird gestartet, nachdem $q_i - 1$ betrachtet wurde und bevor $q_i + 1$ betrachtet wird.

Ist der Algorithmus im untersten LLK angelangt, wird für die drei Koordinaten $q_i - 1$, q_i und $q_i + 1$ überprüft, ob sie definiert sind oder nicht, und entsprechend Null oder Eins zur Rückgabe hinzugefügt. Abbildung 4.2 zeigt die Reihenfolge der Punkte aus der N_{26} Nachbarschaft, wie sie von dem Algorithmus im 3D zurückgegeben werden. Im 2D würde der Algorithmus die N_8 Nachbarschaft zurückgeben.

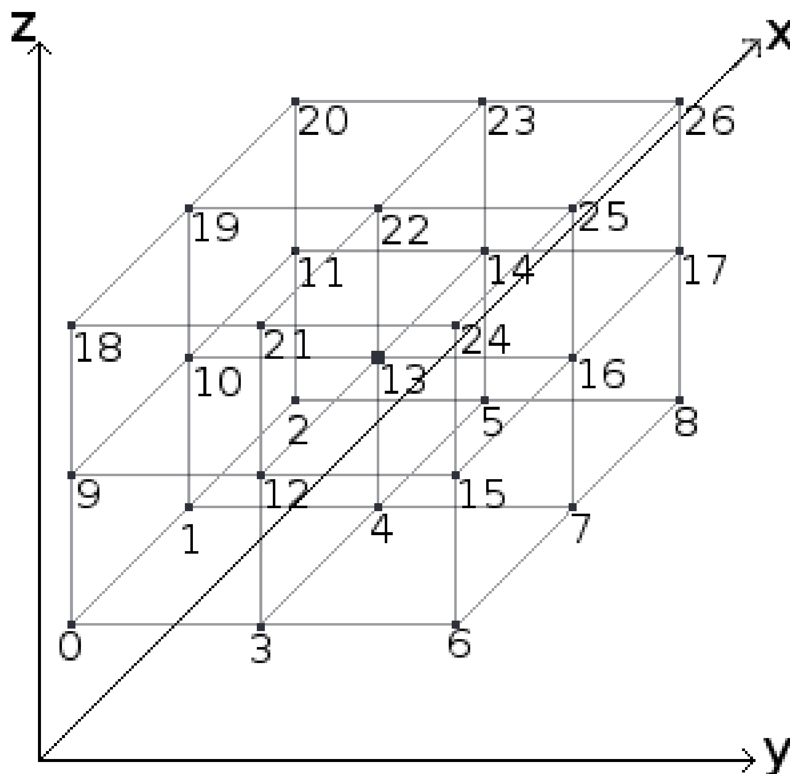


Abbildung 4.2: Reihenfolge der gefundenen Nachbarn bei der N_{26} Nachbarschaft

4.4.2 Ermittlung der definierten Punkte

Damit nicht in jeder Iteration des Skelettierungsalgorithmus das komplette Objekt erneut nach Randpunkten durchsucht werden muß, werden zu Beginn alle definierten Punkte der H-LLK Datenstruktur mittels des sequenziellen Zugriffs ermittelt. Dabei werden in einem Array die Koordinaten der Punkte inklusive eines Flags für jeden einzelnen Punkt gespeichert, welches angibt, ob der Punkt bereits aus der Datenstruktur gelöscht wurde.

Der Algorithmus für den sequenziellen Zugriff bietet den Vorteil, daß die Punkte entsprechend ihrer Koordinaten in geordneter Reihenfolge zurückgegeben werden. Das ist für das Löschen der Punkte nützlich (siehe dazu Abschnitt 4.4.7). So folgen zum Beispiel zuerst alle Punkte der selben z-Koordinate nacheinander. Diese sind dann entsprechend ihrer y-Koordinaten und schließlich noch nach ihren z-Koordinaten sortiert. Siehe dazu auch Abbildung 4.3, in der ein einfaches Beispiel für die Rückgabe des Algorithmus in 2D gegeben ist. Definierte Voxel sind Grün markiert, negative Rot und positive Blau.

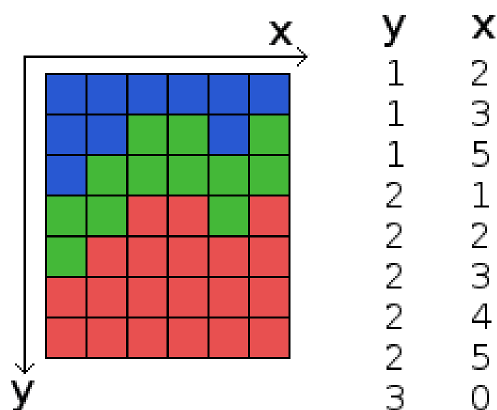


Abbildung 4.3: Beispiel für die Rückgabe der Punkte beim sequenziellen Zugriff

4.4.3 Gerichtete Randpunkte

Wie in der Beschreibung zu Tsaos Algorithmus in Kapitel 2.2.2 erläutert, besteht jede Iteration des Skelettierungsalgorithmus aus sechs Unterzyklen, in denen jeweils die Randpunkte der aktuellen Richtung gelöscht werden. Die Bedingung für gerichtete Randpunkte überprüft nun, ob der aktuell betrachtete Punkt ein Randpunkt der aktuell gültigen Richtung ist. Dazu wird die N_6 Nachbarschaft des Punkts ermittelt und überprüft, ob der Punkt in der angegebenen Richtung einen definierten Nachbarn besitzt. Besitzt er in dieser Richtung keinen definierten Nachbarn, so ist er ein gerichteter Randpunkt der angegebenen Richtung.

4.4.4 Finale Punkte

Finale Punkte bzw. Kurvenendpunkte sind wichtig, damit die Form des Objekts im Skelett erhalten bleibt. Sie dürfen daher nicht gelöscht werden. Da ein Punkt zum finalen Punkt werden kann, wenn Punkte aus der Datenstruktur gelöscht wurden, kann erst bei der Betrachtung des jeweiligen Punkts überprüft werden, ob er final ist.

Dazu wird die N_{26}-Nachbarschaft des Punkts ermittelt. Sind darin weniger als zwei Nachbarn definiert, so ist der Punkt final und darf nicht gelöscht werden.

4.4.5 Simple Punkte

Ein wichtiges Kriterium für einen Randpunkt ist, daß er der Definition eines simplen Punktes entspricht. Wie in Abschnitt 2.2.1 beschrieben, müssen dazu sowohl die Anzahl der Verbundskomponenten der Vordergrundvoxel als auch die Anzahl der Verbundskomponenten der Hintergrundvoxel in der N_{26} Nachbarschaft des Punktes jeweils genau Eins sein. Ist dies der Fall, ist der Punkt ein simpler Randpunkt.

Zur Ermittlung der Verbundskomponenten wird der in Abschnitt 2.2.2 beschriebene Algorithmus verwendet. Als Eingabe erhält der Algorithmus die N'_{26} Nachbarschaft des zu überprüfenden Punktes, welche den gegebenen Punkt mit enthält. Für die Vordergrundkomponenten wird dabei der zentrale Punkt in der Nachbarschaft auf den Hintergrund gesetzt und die 26-Verbundenheit verwendet. Für die Hintergrundkomponenten wird der zentrale Punkt auf den Vordergrund gesetzt und das Inverse der N'_{26} Nachbarschaft benutzt. Die Hintergrundkomponenten werden dabei Anhand der 6-Verbundenheit ermittelt.

4.4.6 Simplizität in der Ebene

Der Algorithmus für die Überprüfung der Simplizität in der Ebene erhält als Eingabe einen Koordinatenvektor qv des zu überprüfenden Punkts sowie die minimal notwendige Anzahl definierter Voxel darstellt, die in den beiden zu überprüfenden Ebenen nach dem Löschen des gegeben Punkts noch vorhanden sein muß. Für die Berechnung der medialen Fläche ist die minimale Anzahl verbleibender Voxel zwei, für die mediale Achse eins.

Um zu überprüfen, ob der zu löschende Punkt simpel bezüglich der zwei zur Richtung des aktuellen Unterzyklus orthogonalen Ebenen ist, wird zunächst die N_{26} Nachbarschaft des Punkts ermittelt, aus der dann die jeweils passenden zwei 3×3 Ebenen extrahiert werden. Da die Rückgabe des Algorithmus für die N_{26} Nachbarschaft auch den gegeben Punkt enthält, muß die entsprechende Position vor der weiteren Bearbeitung noch auf Null gesetzt werden.

Zunächst wird nun für beide Ebenen überprüft, ob die Anzahl definierter Punkte nicht kleiner als die geforderte minimale Anzahl Voxel ist. Sollte dies für eine der Ebenen der Fall sein, so ist die Bedingung zur Simplizität in der Ebene nicht erfüllt und der Punkt kann nicht gelöscht werden.
Anschließend werden in beiden Ebenen die Verbundskomponenten gezählt. Sollten durch das Löschen des Punktes neue Verbundskomponenten in mindestens einer der beiden Ebenen entstehen, so ist der Punkt bezüglich der entsprechenden Ebene nicht simpel und darf daher ebenfalls nicht gelöscht werden.
Für das Zählen der Verbundskomponenten wird eine 2D Version des in Abschnitt 2.2.2 vorgestellten Algorithmus verwendet. Es wird dabei eine Adjazenz-Tabelle für die 8-Verbundenheit verwendet.

Die "Simplizität in der Ebene" Bedingung ist erfüllt, wenn für beide Ebenen gilt, daß der Punkt simpel bezüglich der jeweiligen Ebene ist und nach dem Löschen sich noch mindestens die geforderte Anzahl definierter Punkte in dieser Ebene befinden.

4.4.7 Löschen von Punkten aus der H-LLK Datenstruktur

Beim Löschen der Punkte offenbart die H-LLK Niveaumenge ihre große Schwäche. So ist es notwendig, die Datenstruktur komplett neu aufzubauen, wenn ein Punkt oder mehrere Punkte gelöscht werden sollen, da man nicht wie in einem kartesischen Gitter den Wert des zu löschenden Punkts einfach auf Null setzen kann, sondern die Läufe angepaßt werden müssen, was im Normalfall dazu führt, daß neue Läufe entstehen oder Läufe komplett entfernt werden. Dieses Vorgehen ist sehr aufwendig, aber die einzige Möglichkeit, um Punkte effektiv aus der Datenstruktur zu löschen.

Ziel ist es daher, in einem Löschvorgang möglichst viele Punkte aus der Datenstruktur zu entfernen, um das Kosten/Nutzen-Verhältnis möglichst günstig zu halten. Dies geschieht am effektivsten, in dem man die hierarchische Strukur der H-LLK Niveaumenge ausnutzt. Dazu wird zunächst ein veränderter Koordinatenvektor eingeführt, dessen Datentyp in Pseudo-Code folgendermaßen aussieht:

```
1 struct punktVektor {
2   short int                  koord;
3   Vektor(punktVektor)        subKoord;
4 };
```

Zu jeder Koordinate *koord* werden in dem Koordinatenvektor die zugehörigen dimensional untergeordneten Koordinaten in einem weiteren Koordinatenvektor *subKoord* gespeichert.

Der Algorithmus zum Löschen der Punkte erhält als Eingabe einen nach seinen Koordinaten *koord* aufsteigend sortierten Koordinatenvektor vom Typ *Vektor(punktVektor)* und startet auf dem einen Segment des obersten LLK Blocks. Es wird nun eine neue H-LLK Datenstruktur

aufgebaut, die die zu löschenden Punkte nicht mehr enthält. Dazu werden nun alle Läufe des Segments betrachtet.

Nicht definierte Läufe werden unverändert übernommen. Es wird dabei lediglich überprüft, ob der zu letzt zur neuen H-LLK Datenstruktur hinzugefügte Lauf nicht definiert war. In dem Fall wird lediglich die Unterbrechung angepaßt. Sollte der letzte hinzugefügte Lauf außerdem negativ gewesen sein und der aktuelle Lauf positiv, dann wird dieser Lauf noch durch einen positiven Lauf ersetzt.

Ist der Lauf hingegen definiert, werden die einzelnen Koordinaten des Laufs betrachtet. Es sei qv ein Koordinatenvektor vom Typ Vektor(punktVektor) und i ein Index auf die Elemente dieses Koordinatenvektors, initialisiert mit Null. Desweiteren sei $nCoord$ die aktuell betrachtete Koordinate des definierte Laufs. Wenn ein hierarchisch untergeordnetet LLK Block existiert, wird zunächst das zu $nCoord$ gehörige Untersegment aus der Datenstruktur extrahiert. Als nächstes wird überprüft, ob $qv[i].koord = nCoord$ ist. Ist dies nicht der Fall (oder es wurden bereits alle Elemente von qv betrachtet), bleibt die Koordinate im neuen Segment definiert. Bei höherer Dimension wird das zugehörige Untersegment unverändert zum dimensional niedrigeren LLK Block der neuen H-LLK Datenstruktur hinzugefügt.

Besteht hingegen Gleichheit zwischen $nCoord$ und $qv[i].koord$, wird die k im neuen Segment zu einem nicht definierten Lauf hinzugefügt. War der letzte Lauf im neuen Segment definiert, wird ein negativer Lauf hinzugefügt. War er nicht definiert, wird die Unterbrechung des letzten Laufs angepaßt. Wurde noch kein Lauf zum neuen Segment hinzugefügt, wird ein positiver Lauf hinzugefügt. Als neue Unterbrechung wird $k+1$ hinzugefügt.
Existiert ein hierarchisch untergeordneter LLK Block, so wird der Algorithmus zum Löschen auf dem vorher extrahierten Untersegment mit $qv[i].subKoord$ als Eingabe gestartet. Enthält das Untersegment nach dem Löschen weiterhin definierte Daten, wird es zum nächst niedrigeren LLK Block der neuen H-LLK Datenstruktur hinzugefügt und k bleibt definiert. Ansonsten wird k zu einem nicht definierten Lauf hinzugefügt.

Der Algorithmus endet, wenn die neue H-LLK Datenstruktur komplett aufgebaut wurde. Zum Schluß wird noch die orignale H-LLK Datenstruktur mit der neu erzeugten überschrieben.

4.4.8 Partielle Skelettierung

Ein Problem bei der Skelettierung auf der erweiterten H-LLK Datenstruktur liegt in der Unterteilung des Datensatzes bei der Bearbeitung entlang der z-Achse. So wird, wenn man den Algorithmus nur auf einen Teil des Datensatzes anwendet, ein zu diesem Teil passendes Skelett berechnet, welches aber nicht dem entsprechenden Teilstück des Skeletts entspricht, das bei der Skelettierung des gesamten Datensatzes entsteht und somit aus den einzeln berechne-

ten Skeletten kein gültiges Skelett für das gesamte Objekt entsteht. Es werden nun mehrere Ansätze zur Lösung dieses Problems erläutert.

Eine denkbarer Ansatz ist es, einen ein Voxel dicken Bereich aus den benachbarten z-Blöcken des aktuell betrachteten z-Blocks zu laden. Wenn der z-Block die minale Ausdehnung z_k und die maximale Ausdehnung z_m hat, dann werden noch die xy-Ebenen mit den z-Koordinaten z_{k-1} und z_{m+1} geladen, falls der z-Block in diese Richtung Nachbarn besitzt. Zurückgeschrieben wird aber nur das Ergebnis zwischen den Koordinaten z_k und z_m. Um die Übergänge zu erhalten, werden die definierten Voxel mit der Koordinate z_{k-1} für den Unterzyklus der Richtung $-z$ und die mit z_{m+1} für den Unterzyklus $+z$ gesperrt. Allerdings führt dieses Vorgehen zu einer Streckung oder Stauchung des Skeletts entlang der z-Achse. Um dies zu verhindern, müßten die Voxel im Übergangsbereich zwischen den z-Blöcken, die zum korrekten Skelett gehören, vor dem Löschen geschützt werden. Allerdings kann man diese Voxel nur durch die Berechnung des vollständigen Skeletts ermitteln.

Bei einem zweiten Ansatz wird ein größerer Bereich der benachbarten z-Blöcke zusätzlich zum aktuellen z-Block geladen, davon das Skelett berechnet und der aktuelle z-Block mit den Koordinaten z_k und z_m wieder zurückgeschrieben. Dies scheitert aber ebenfalls daran, daß das so berechnete Teilskelett zwischen den Koordinaten z_k und z_m nicht dem Skelett entspricht, welches für diesen Bereich bei der Skelettierung des kompletten Datensatzes entstehen würde.

Das Problem bei der Skelettierung besteht vorallem darin, daß jeder Ausdünnungsschritt[2] auf dem Objekt vom vorhergehenden Ausdünnungsschritt abhängt. Da bei der teilweisen Skelettierung weder Informationen über die aktuell nicht geladenen Teilstücke des Datensatzes vorliegen, noch wie diese Teilstücke sich bei jedem Ausdünnungsschritt des aktuell geladenen Teilstücks mit verändern würden, lassen sich keine Übergangsbedingungen zwischen den Teilstücken finden, mit denen ein korrektes Skelett erzeugt wird.

Eine dritte aufwendige aber gangbare Möglichkeit zur Aufteilung der Skelettierung besteht trotzdem. Dabei werden mit einem z-Block die jeweils ein Voxel dicken angrezenden xy-Ebenen der benachbarten z-Blöcke geladen. Allerdings ist die Ausführung eines kompletten Ausdünnungszyklus über alle sechs Koordinatenrichtungen auch nicht möglich, da sich bei jedem Löschen von gerichteten Randpunkten einer Richtung die Nachbarschaften der Punkte ändern können und so die Punkte in den Übergangsbereichen zu den benachbarten z-Blöcken keine, im Vergleich zur Skelettierung des kompletten Objekts, korrekten Nachbarschaften mehr haben können, was dann wegen den anderen Nebenbedingungen wie finale Punkte oder "Simplizität in der Ebene" zu einem Fehlverhalten beim Löschen der gerichteten Randpunkte der verbleibenden Richtungen führen kann.

[2] Damit ist ein Unterzyklus gemeint, in dem Randpunkte einer Richtung gelöscht werden.

Der Algorithmus wird daher in sechs Unterzyklen gemäß der sechs Koordinatenrichtungen unterteilt. In jedem Schritt werden alle z-Blöcke betrachtet. Für den jeweils aktuellen z-Block werden die gerichteten Randpunkte der aktuell gültigen Richtung ermittelt. Nun wird für die ermittelten Punkte überprüft, ob sie nicht final sind, es sich um simple Punkte handelt und sie die "Simplizität in der Ebene"-Bedingung erfüllen. Ist dies der Fall, werden die Punkte gelöscht und der geänderte z-Block zurückgeschrieben. Anschließend wird der geänderte z-Block zurückgeschrieben und mit dem nächsten fortgefahren. Wurden Punkte in einem z-Block für eine der sechs Richtungen gelöscht, startet der Algorithmus erneut, bis keine Punkte weiter gelöscht werden können. Die zusätzlich geladenen Punkte an den Übergangsbereichen zu den benachbarten z-Blöcken dienen dabei nur zur korrekten Bestimmung der Nachbarschaften und werden beim Löschen nicht mit betrachtet.

5 Ergebnisse

Aus Mangel an geeigneten großen Datensätzen, wurden geeignete Datensätze durch die Konvertierung des Kuh-Modells (Abbildung 5.1) in eine Voxeldarstellung erzeugt. Der Speicheraufwand ist während der Konvertierung allerdings so hoch, daß aufgrund technischer Begrenzungen lediglich Auflösungen bis maximal $698 \times 471 \times 303$ erzeugt werden konnten. Es wurden Datensätze mit folgenden Auflösungen erzeugt:

ID	Auflösung	Gesamtzahl Voxel
1	$31 \times 21 \times 13$	7.812
2	$69 \times 47 \times 30$	97.290
3	$149 \times 101 \times 64$	963.136
4	$323 \times 218 \times 140$	9.857.960
5	$698 \times 471 \times 303$	99.613.674

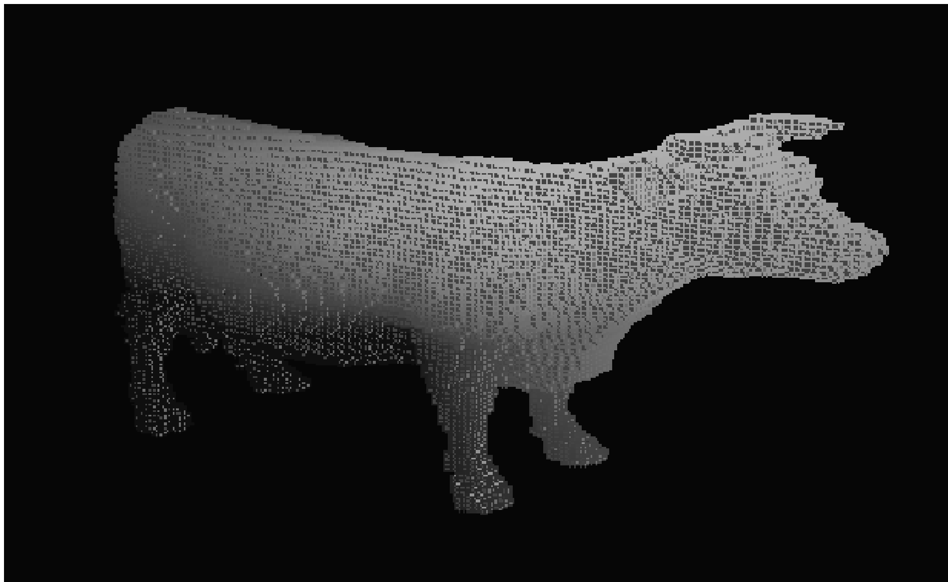


Abbildung 5.1: Das Kuh-Modell in Voxeldarstellung

Abbildung 5.2 zeigt den Speicherbedarf der erzeugten Volumendatensätze in MiByte, welche nach der Konvertierung jeweils in Form eines dreidimensionalen Arrays als kartesisches Gitter im Speicher vorlagen. Wie man sieht, benötigt das Kuh-Modell in der höchsten der

fünf Auflösungen bereits über zwei GiB Speicher.

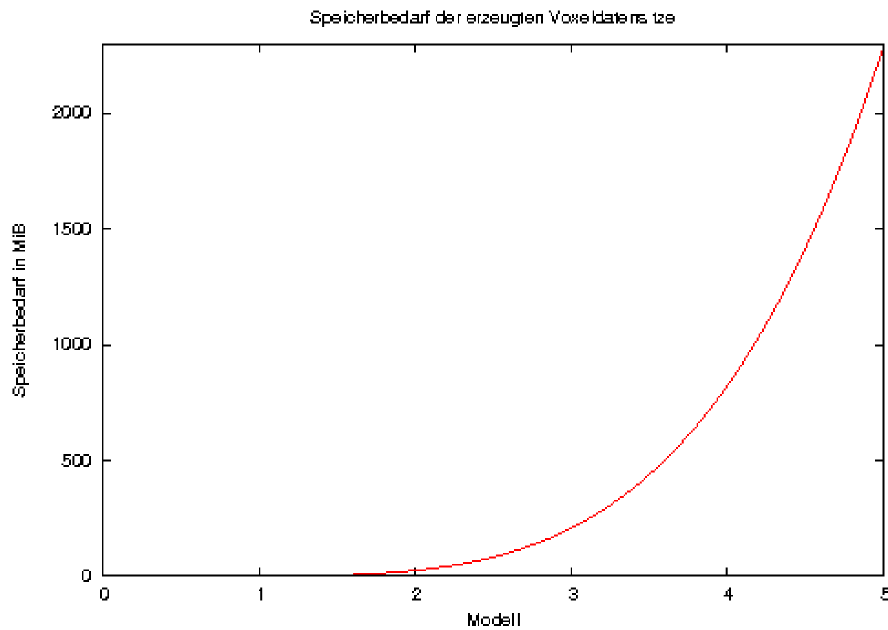


Abbildung 5.2: Speicherbedarf der erzeugten Voxeldatensätze

Bei Technologien wie der Computertomographie werden allerdings direkt volumetrische Datensätze erzeugt, so daß eine Umwandlung entfällt und deutlich größere Auflösungen erreicht werden können.

Im folgenden wird noch zwischen hohlen Modellen und soliden Modellen, also Modelle ohne Hohlräume, unterschieden. Für die Ermittlung der Laufzeiten wurden ausschließlich die soliden Modelle verwendet. Die Laufzeiten wurden auf einem Intel Core 2 Duo mit 2,4 GHz ermittelt.

5.1 Speicherbedarf

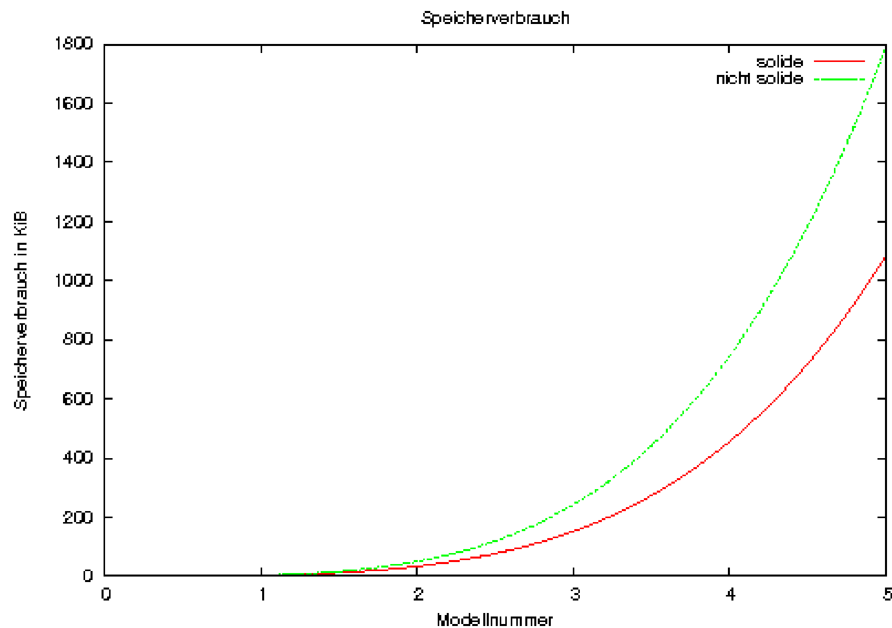


Abbildung 5.3: Speicherbedarf der erweiterten H-LLK Datenstruktur

Wie aus Abbildung 5.3 ersichtlich, ist der Speicherbedarf der erweiterten H-LLK Niveaumenge im Vergleich zu dem benötigten Speicher der erzeugten Testmodelle minimal, da lediglich die Läufe, Unterbrechungen und ein paar Indizes gespeichert werden müssen. Bei dem Kuh-Modell vervierfacht sich in etwa der Speicherbedarf, wenn sich die Gesamtanzahl der Voxel verzehnfacht. Demnach läßt sich abschätzen, daß für eine Auflösung von 5000^3 Voxeln für das Kuh-Modell weniger als 100 MiB notwendig wären.
In Abbildung 5.3 wird auch deutlich, wie der Speicherbedarf bei der erweiterten H-LLK Datenstruktur von der Anzahl der kodierten Läufe abhängt. So benötigt ein hohles Modell deutlich mehr Speicher als ein Modell ohne Hohlräume, da bei einem hohlen Modell viel mehr Läufe benötigt werden, um es zu kodieren. Dagegen benötigt ein solides Modell trotz größerer Anzahl definierter Voxel weniger Speicher. Das begründet sich in der Tatsache, daß die Daten binarisiert vorliegen und kein Werte-Array benötigt wird, da jeder definierte Voxel den gleichen Wert hat.

5.2 Laufzeiten der Algorithmen

Für die nachfolgend ermittelten Laufzeiten wurden, wenn nicht anders vermerkt, ausschließlich die soliden Versionen des Kuh-Modells verwendet. Die Strukturelemente für die morphologischen Operationen sind voll belegt und quadratisch. Die Größe der z-Blöcke betrug für alle Laufzeitermittlungen zehn z-Koordinaten. Dargestellt werden die Laufzeiten im Verhältnis zur Gesamtzahl Voxel im Datensatz.

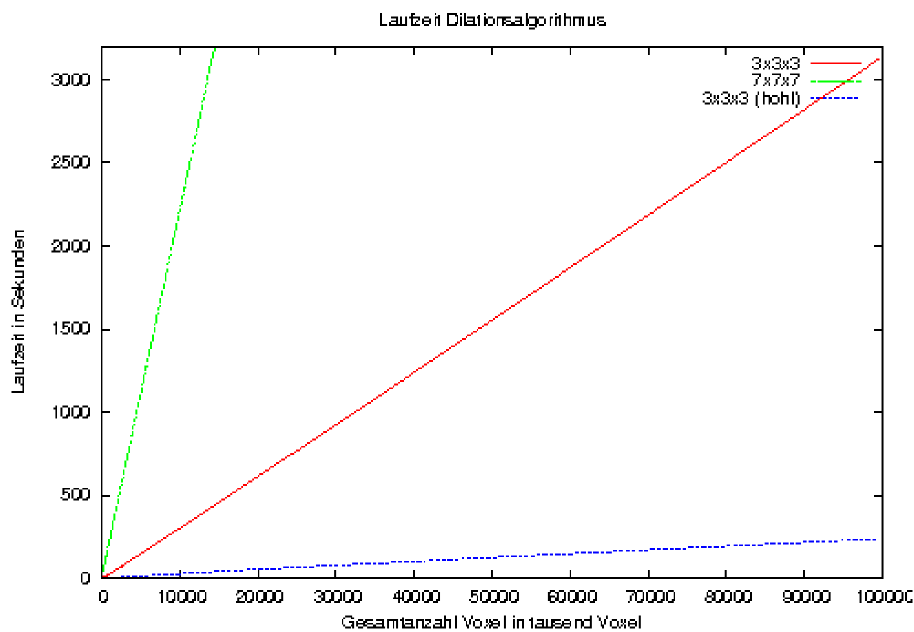


Abbildung 5.4: Laufzeiten der Dilation

In Abbildung 5.4 sind die Laufzeiten des Algorithmus für zwei verschieden große Strukturelemente sowie zum Vergleich die Laufzeit auf einem hohlen Testobjekt mit einem $3 \times 3 \times 3$ Strukturelement dargestellt. Wird für alle Auflösungen ein Strukturelement mit der selben Ausdehnung verwendet, so steigt die Laufzeit des Dilationsalgorithmus nahezu linear entsprechend der Gesamtanzahl Voxel an. Vergrößert sich das Strukturelement auf ungefähr die doppelte Größe, so kann die Laufzeit bis um das Achtfache ansteigen. So benötigte die Dilation mit einem $7x7x7$ Strukturelement für das Testobjekt 5 bereits über sechs Stunden. Da bei dem Algorithmus jede definierte Koordinate betrachtet werden muß, sind die Laufzeiten bei Objekten mit Hohlräumen etwas besser, wie auch aus Abbildung 5.4 ersichtlich ist.

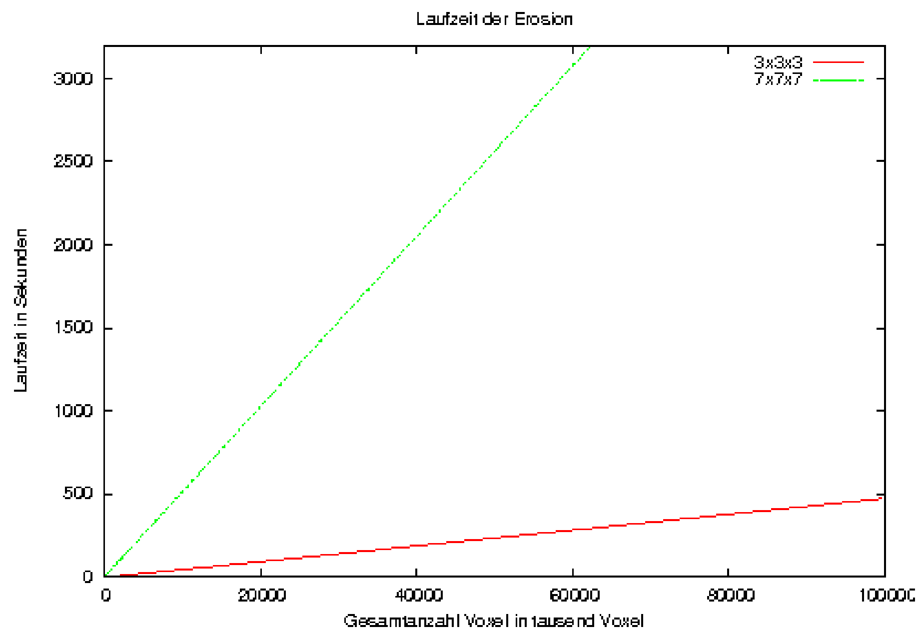


Abbildung 5.5: Laufzeiten der Erosion

Da der Algorithmus für die Erosion nicht so aufwendig ist wie bei der Dilation, zeigen sich allgemein bessere Laufzeiten. Ebenso wie bei der Dilation steigt die Laufzeit annähernd linear zur Gesamtzahl Voxel an. Allerdings ist die Laufzeit der Erosion noch stärker von der Größe des Strukturelements abhängig, wie man an dem Auseinanderdriften der beiden Kurven in Abbildung 2.2 sieht.

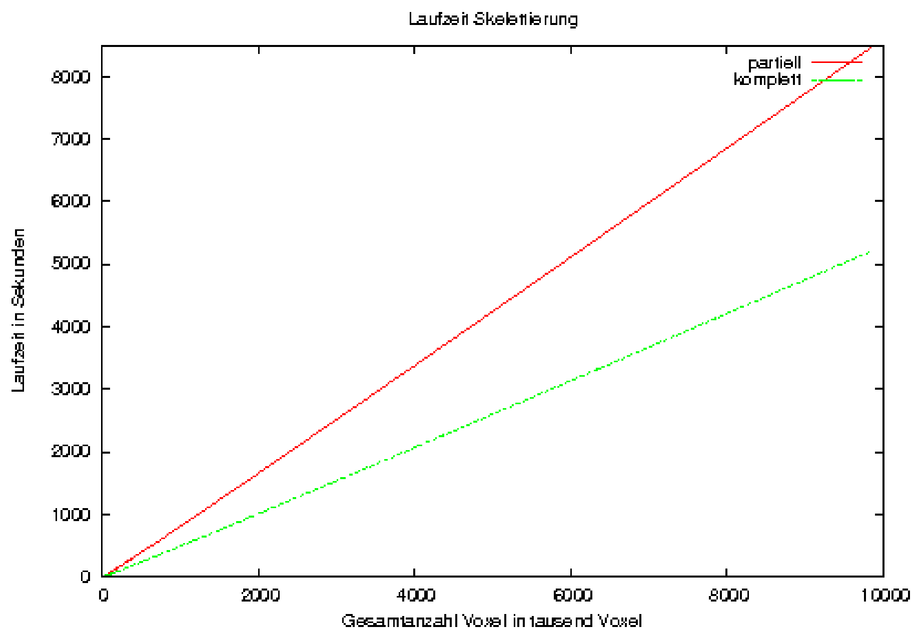


Abbildung 5.6: Laufzeiten der Skelettierung

Abbildung 5.6 zeigt die Laufzeiten zur vollständigen Berechnung der medialen Achse, einmal mit dem normalen Skelettierungsalgorithmus, wenn der Datensatz vollständig im Speicher liegt, und einmal mittels partieller Skelettierung. Da bei jedem Schritt, in dem Punkte aus der Datenstruktur gelöscht werden, die Datenstruktur komplett neu aufgebaut werden muß, ist die Berechnung der medialen Achse allgemein sehr aufwendig. Die Laufzeit steigt dabei sehr stark linear in Bezug zur Gesamtzahl Voxel an. So wird bereits für das noch recht kleine Testobjekt 3 mehr als eine Stunde für die Berechnung der medialen Achse benötigt. Da bei der partiellen Skelettierung in jedem Teilschritt alle z-Blöcke geladen und wieder zurückgeschrieben werden, steigt dabei die Laufzeit deutlich stärker als bei der Skelettierung des kompletten Objekts.

6 Zusammenfassung

6.1 Diskussion

Der große Vorteil der erweiterten H-LLK Datenstruktur zeigt sich in dem sehr geringen Speicherbedarf für die Daten, welcher relativ unabhängig von der Ausdehnung des Objekts ist, sondern hauptsächlich von der Anzahl der Läufe und damit entstehenden Segmente in den unteren Dimensionen abhängt. So benötigen volumetrische Datensätze, die unkomprimiert mehrere GiByte Speicher belegen, als erweiterte H-LLK Niveaumenge lediglich ein paar MiByte.

Dem großen Vorteil der speichereffizienten Verwaltung der Datensätze folgt der große Nachteil in Form der sehr langen Laufzeiten der Algorithmen für die Bearbeitung der Datensätze. Da bei jeder Änderung des Datensatzes die komplette Datenstruktur neu aufgebaut werden muß, arbeiten die morphologischen Operatoren auf der Datenstruktur nur sehr langsam.
Ein besonderes Problem stellt dabei auch die Laufzeit des Skelettierungsalgorithmus dar, welcher zum einen durch das aufwendige Löschen von Punkten in jedem Teilschritt sehr langsam wird, und sich die Erzeugung des Skeletts nicht effizient auf Teildatensätze aufteilen läßt. Da aktuelle Rechentechnik über deutlich mehr Hauptspeicher verfügt, als selbst Datensätze mit sehr großen Auflösungen als erweiterte H-LLK Datenstruktur benötigen, empfiehlt es sich, zu Gunsten deutlich besseren Laufzeiten den Datensatz für die Skelettierung komplett in den Hauptspeicher zu laden.

Mit diesen Ergebnissen eignet sich die erweiterte H-LLK Niveaumenge hervorragend, um große Datensätze effizient zu speichern. Für die Bearbeitung der Datensätze ist die Datenstruktur allerdings weniger geeignet, da bereits die im Vergleich zu den anvisierten Auflösungen relativ kleinen Datensätze wie das Kuh-Modell 5 (siehe Abschnitt 5) Stunden zur Berechnung der medialen Achse oder der Dilation benötigen.

Ein Problem bei der Erzeugung der erweiterten H-LLK Datenstruktur ergibt sich außerdem für die Erhaltung der Niveaumengeneigenschaft, der Klassifizierung der Hintergrundvoxel in negativ (innerhalb des Objekts) und positiv (außerhalb des Objekts). So läßt sich zwar für jedes einzelne Bild entscheiden, welche Voxel in der Ebene positiv oder negativ sind. Allerdings werden somit auch Voxel als negativ klassifiziert, die zwar in der 2D Ebene komplett von definierten Voxeln umschlossen sind, aber im 3D eben nicht komplett eingeschlossen sind. Es ist deshalb noch ein zusätzliches aufwendiges Verfahren notwendig, welches nach

dem Konvertieren der Daten überprüft, ob dabei als negativ klassifizierte Voxel im 3D in direkter Nachbarschaft zu positiv klassifizierten Voxeln stehen und die Voxel entsprechend neu gelabelt werden.

6.2 Ausblick

Ein begrenzender Faktor für die maximal mögliche Auflösung der Datensätze liegt in der Verwendung von 2 Byte Integern für die Unterbrechungen der Läufe sowie die Ausdehnungen des Objekts und 4 Byte Integern für die Indizes in den Segmentdateien. Damit liegt die maximale mögliche (theoretische) Auflösung derzeit bei 65.535^3 Voxeln mit maximal 214.748.3647 Läufen in y und x-Richtung. Diese Begrenzung kann, falls erforderlich, natürlich durch größere Datentypen erhöht werden. Allerdings steigt damit auch der Speicherbedarf entsprechend an. So wäre zum Beispiel für den gleichen Datensatz bei der Verwendung von 4 Byte Integern für die Koordinaten und 8 Byte Integern für die Indizes bereits der doppelte Speicher notwendig, als bei der aktuellen Verwendung von 2 und 4 Byte Integern.

Die erweiterte H-LLK Datenstruktur wurde auf die Speicherung von binarisierten Voxelmengen optimiert, welches den Wegfall des separaten Werte-Arrays ermöglicht. Sie läßt sich aber auch wieder um das Werte-Array erweitern. Der Speicherbedarf steigt dann je nach Menge der definierten Voxel sowie der Art des für ihre definierten Daten verwendeten Datentyps an. Auf die Algorithmen hat dies allerdings kaum einen Einfluß. Es sind allerdings zwei wichtige Aspekte zu beachten. Zum einen müssen die Werte der definierten Voxel unter Umständen aktualisiert werden bzw. neu erzeugte definierte Voxel, wie sie bei der Dilation entstehen, müssen initialisiert werden. Desweiteren empfiehlt es sich, für das Laden der Daten die Größe der z-Blocks dynamisch anzupassen, damit der Bedarf an Hauptspeicher während der jeweiligen Operation möglichst konstant gehalten wird, da die einzelnen y-Segmente durch das Werte-Array dann auch deutlich mehr Hauptspeicher benötigen. Eine Möglichkeit wäre, nur soviele y-Segmente je z-Block zu laden, bis ein bestimmter Grenzwert an zulässigen definierten Voxeln überschritten wird. Dies bietet sich an, da die Anzahl definierter Voxel je y-Segment durch die Anzahl Einträge im Werte-Array explizit gegeben ist. Für die Berechnung des Skeletts kann zu dem auf das Laden des Werte-Arrays verzichtet werden, da es für den Algorithmus nicht benötigt wird.

Literaturverzeichnis

[BB04] David Brunner und Guido Brunnett: *Mesh Segmentation Using the Objekt Skeleton Graph*, *Proc. of the seventh IASTED International Conf. on Computer Graphics and Imaging, CGIM*, S. 48–55, 2004.

[BNSdB97] G. Borgefors, I. Nystrom und G. Sanniti di Baja: *Connected Components in 3D Neighbourhoods*, S. xx–yy, 1997.

[Bru05] Guido Brunnett: *Skript Vorlesung Solid Modelling*, Technische Universität Chemnitz, Chemnitz, Germany, 2005.

[GB90] W. Gong und G. Bertrand: *A Simple Parallel 3D Thinning Algorithm*, S. I: 188–190, 1990.

[HBN+06] Ben Housten, Christopher Batty, Ola Nilsson, Michael Nielsen und Ken Museth: *Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation*, *ACM Transactions on Graphics*, Bd. 25, No. 1:S. 1–24, Jan. 2006.

[HG07] Holger Gerth: *Vergleichende Studie zur Verwaltung großer volumetrischer 3D-Datensätze mit dem Schwerpunkt auf den hierarchischen und lauflängenkodierten Datenstrukturen*, Studienarbeit, Technische Universität Chemnitz, 2007.

[Loh98] Gabriele Lohmann: *Volumetric Image Analysis*, B.G. Teubner, Stuttgart, Germany, 1998, ISBN 3-519-06447-2.

[Mor81] D. Morgenthaler: *Three-dimensional simple points: serial erosion, parallel thinning, and skeletonization*, Technical report tr-1005, University of Maryland, Computer Vision Laboratory, Computer Science Center, Febr. 1981.

[Ste05] Johannes Steinmüller: *Skript Vorlesung Bildverarbeitung*, Technische Universität Chemnitz, Chemnitz, Germany, 2005.

[TF81] Y.F. Tsao und K.S. Fu: *A Parallel Thinning Algorithm for 3D Pictures*, Bd. 17(4):S. 315–331, December 1981.

Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 25. September 2007

Gerth Holger

Anhang

A Bedienung des Testprogramms

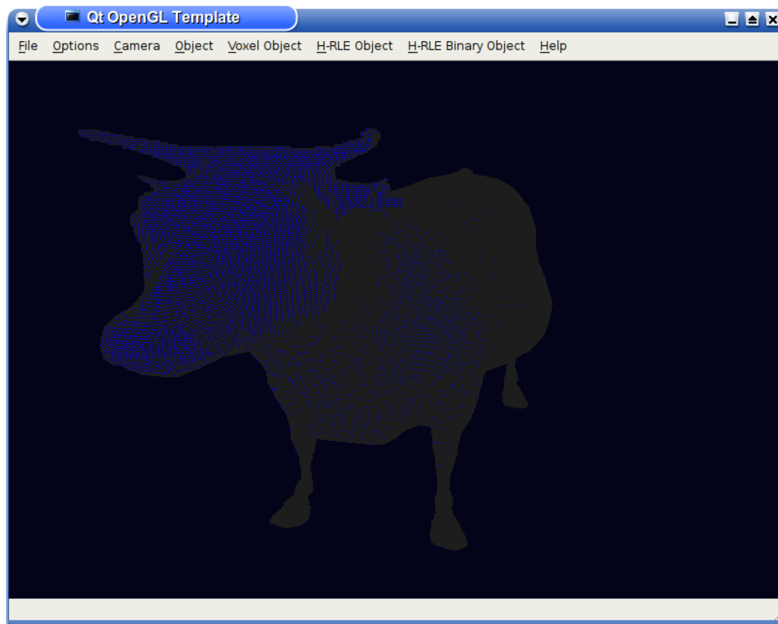


Abbildung A.1: Screenshot des Testprogramms

Die Steuerung der Kammera erfolgt über die Maus. Durch Drücken der linken Maustaste und bewegen der Maus wird die Kamera rotiert, durch Drücken der rechten Maustaste und vor und zurück bewegen der Maus wird die Kamera rein und raus gezoomt.

Die restliche Bedienung des Programms erfolgt komplett über die verschiedenen Menüs des Programms. Unter *File* kann über *Open* eine BRep-Datei im Format .smf (Simple Model Format) oder über *Open HRLEB File* eine erweiterte H-LLK Datenstruktur geladen werden. Die Menüs *Options* und *Camera* ermöglichen verschiedene Einstellungen für OpenGL. Die Menüs *Object*, *Voxel Object*, *HRLE Object* und *HRLE Binary Object* ermöglichen das Ausführen verschiedener Algorithmen auf den jeweiligen Datenstrukturen.

B Algorithmen

In diesem Abschnitt werden alle wichtigen Algorithmen für die H-LLK Niveaumenge und die erweiterte H-LLK Niveaumenge in einem C++-ähnlichen Pseudo-code beschrieben.

B.1 Allgemeine Datentypen

```
1 enum borderPointType {
2     BP_NONE,          // kein Randpunkt
3     BP_BOTTOM,        // z-1
4     BP_TOP,           // z+1
5     BP_EAST,          // y+1
6     BP_WEST,          // y-1
7     BP_SOUTH,         // x+1
8     BP_NORTH          // x-1
9 };
```

```
1 enum morphType {
2     EROSION,
3     DILATION,
4     OPENING,
5     CLOSING
6 };
```

```
1 #define RT_NEGATIVE        -1
2 #define RT_POSITIVE        2147483647
3 #define RT_DEFINED(x)      ((RT_NEGATIVE < (x)) && ((x) < RT_POSITIVE))
```

```
1 typedef struct structCoordValue {
2     short int                          coord;
3     vector<struct structCoordValue>    subCoords;
4 } coordValue;
```

```
1 typedef struct struct_DefinedPoint {
2     vector<int>        coordinate;
3     bool               isDeleted;
4 } definedPoint;
```

B.2 Algorithmen der H-LLK Niveaumenge

B.2.1 Direktzugriff

Der Methode für den Direktzugriff wird ein Vektor qv mit Koordinaten des gewünschten Gitterpunkts übergeben sowie ein Segment Index si, initialisiert mit Null. Die Methode gibt den entsprechenden Lauf des Gitterpunkts zurück bzw. den entsprechenden definierten Index.

```
1  int randomAccess( vector<int> qv, int si = 0 ) {
2
3    qi = qv.popBack();
4
5    Ermittle firstRun, firstBreak und Segment Länge des gegebenen Segments
6
7    int run = BinarySearch( qi, firstRun, firstBreak );
8
9    if ( RT_DEFINED(run) ) {
10
11     definedDataIndex = run + qi - letzter Break vor dem Run;
12
13     if ( lower_hrle != NULL )
14       return lower_hrle->randomAccess( qv, definedDataIndex )
15     else return definedDataIndex;
16   }
17
18   return run;
19 }
```

B.2.2 Sequenzieller Zugriff

Dem sequenziellen Zugriff wird ein zu Beginn leerer Vektor von Eltern-Koordinaten übergeben sowie ein Segment Index si, welcher wieder mit Null initialisiert wird. Es wird eine Liste aller definierten Gitterpunkte der H-LLK Niveaumenge zurückgeliefert.

```
1  vector<definedPoint> sequentialAccess( vector<int> parentCoord, int si = 0 ) {
2
3    vector<definedPoint>      returnValue;
4    vector<int>               currentCoordinate;
5
6    Ermittle First Run und Segment Länge des gegebenen Segments
7
8    for ( i = alle Runs des Segments ) {
9
10     short int run = runCodeArray.at(i);
11
12     Ermittle Start- und Endkoordinate des aktuellen Runs
13
14     for ( qi = alle Koordinaten des Runs ) {
15
16       currentCoordinate = parentCoord;
17       currentCoordinate.push_back( qi );
18
19       if ( lower_hrle == NULL ) {
20         definedPoint newCoordValue;
21         // Reihenfolge der Koordinaten in currentCoordinate vorher umdrehen
22         newCoordValue.coordinate = currentCoordinate;
23         newCoordValue.deleted = false;
24         returnValue.push_back( newCoordValue );
25       }
26       else {
27         definedDataIndex = run + qi - letzter Break vor dem Run;
28         vector<definedPoint> tmp = lower_hrle->sequentialAccess( currentCoordinate, definedDataIndex );
29         returnValue.union( tmp );
30       }
31
```

```
32          }
33
34      }
35
36      return returnValue;
37
38  }
```

B.2.3 Dilationsalgortithmus

Der Dilationsalgortithmus besteht im Prinzip aus drei Methoden: einer 1D Dilation, welches ein einzelnes LLK Segment unabhängig von der restlichen Datenstruktur dilatiert, einem Union-Algorithmus, welcher zwei LLK Segmente inklusive ihrer zugehörigen Segmente in hierarchisch untergeordneten Dimensionen vereint, der Dilation an sich, welche mit Hilfe der ersten beiden Methoden die Dilation des kompletten Objekts durchführt. Als Eingabe erhält die Dilation das Strukturelement und die Koordinaten des Anlegepunktes darin.

1D Dilation

```
1   HierarchicalRLE* dilation1D( int si, CHierarchicalRLE* se, short int nCoord ) {
2
3       HierarchicalRLE* returnValue, temp;
4
5       HierarchicalRLE* segment = getSegmentAt(si);
6
7       for ( run = aktueller Lauf, über alle Läufe von segment ) {
8
9           if (RT_DEFINED(run) {
10              for ( aktKoord, über alle Koordinaten von run ) {
11
12                  passe Koordinaten von se an, so daß nKoord == aktKoord
13
14                  temp = union(segment, se);
15
16                  returnValue->add(temp);
17              }
18          }
19      }
20
21      return returnValue
22  }
```

Union

Als Eingabe erhält der Union-Algorithmus eine Liste von LLK Segmenten in Form von separaten H-LLK Datenstrukuren, welche mehrdimensional sein können. Diese Segmente werden während der Union zu einem einzigen zusammengefügt, welches dann zurück gegeben wird,

```
1   HierarchicalRLE* hrleUnion( vector<HierarchicalRLE*> hrleVector ) {
2
3     if ( hrleVector.size() == 1 ) return hrleVector.back();
4
5     HierarchicalRLE* segment1 = hrleVector.at(0);
6
7     for ( int i = 1; i < hrleVector.size(); i++ ) {
8
9       HierarchicalRLE* segment2 = hrleVector.at(i);
```

```
10  HierarchicalRLE* returnValue = new HierarchicalRLE();
11
12  Setze min und max von returnValue auf das grössere max bzw. kleinere min von segment1 und segment2
13  returnValue->start_indices.push_back( 0 );
14
15  Ermittle Start- und Endkoordinaten der ersten Läufe der beiden Segmente
16
17  if (startCoord2 > endCoord1)
18      füge Läufe von segment1 unverändert zu returnValue hinzu bis endCoord1 > startCoord2
19
20  if (startCoord1 > endCoord2)
21      füge Läufe von segment2 unverändert zu returnValue hinzu bis endCoord2 > startCoord1
22
23  // pos1 und pos2 sind Indices ins Run Type Array der beiden Segmente
24  int pos1 = 0, pos2 = 0;
25
26  while ( ( pos1 < segment1->run_types.size()) || (pos2 < segment2->run_types.size()) ) {
27
28    run1 = segment1->run_types.at(pos1);
29    run2 = segment2->run_types.at(pos2);
30
31    if ( run1 == run2 ) {
32
33      if ( RT_DEFINED(run1) ) {
34        Füge die grössere endCoord zu returnValue->run_breaks hinzu
35
36        for ( qi = alle Koordinaten des grösseren Laufs ) {
37          if ( qi > kleinere endCoord ) gehe zum nächsten Lauf des betroffenen Segments;
38          // wenn die kleinere endCoord überschritten wird, ist der betroffene
39          // Lauf eventuell nicht mehr RT_DEFINED
40          if ( RT_DEFINED(beide Läufe) )
41            Füge Union der beiden Subsegmente zu returnValue->lower_hrle hinzu
42          else
43            Füge das Subsegment des definierten Laufs zu returnValue->lower_hrle hinzu
44        }
45      }
46      else {
47        Füge die kleinere endCoord zu returnValue->run_breaks hinzu
48      }
49
50      if (returnValue->run_types.back != run1)
51        returnValue->run_types.push_back(run1);
52
53    }
54    else {
55
56      if ( RT_DEFINED(run1) ) {
57        returnValue->run_breaks.push_back(endCoord1);
58
59        for ( qi = startCoord1; qi < endCoord1; qi++ ) {
60          if ( qi > endCoord2 ) pos2++;
61          // bei Überschreiten von endcoord2 könnte run2 wieder RT_DEFINED werden
62          if ( RT_DEFINED(beide Läufe) )
63            Füge Union der beiden Subsegmente zu returnValue->lower_hrle hinzu
64          else
65            Füge das Subsegment von run1 zu returnValue->lower_hrle hinzu
66        }
67
68        if (returnValue->run_types.back != run1)
69          returnValue->run_types.push_back(run1);
70      }
71
72      else if ( RT_DEFINED(run2) ) {
73        returnValue->run_breaks.push_back(endCoord2);
74
75        for ( qi = startCoord2; qi < endCoord2; qi++ ) {
76          if ( qi > endCoord1 ) pos1++;
77          // bei Überschreiten von endcoord1 könnte run1 wieder RT_DEFINED werden
78          if ( RT_DEFINED(beide Läufe) )
79            Füge Union der beiden Subsegmente zu returnValue->lower_hrle hinzu
80          else
81            Füge das Subsegment von run2 zu returnValue->lower_hrle hinzu
82        }
83
84        if (returnValue->run_types.back != run2)
85          returnValue->run_types.push_back(run2);
86      }
87
```

```
88          else {
89            Füge die kleinere endCoord zu returnValue->run_breaks hinzu
90            Falls einer der Läufe RT_POSITIVE, füge einen positiven Lauf zum returnValue hinzu,
91            ansonsen einen negativen (returnValue->run_types.back != run)
92          }
93
94        }
95
96        Falls endCoord eines Runs kleiner als returnValue->run_breaks.back(), gehe solange zum
97        nächsten Lauf (pos1++ bzw. pos2++) im jeweiligen Segment, bis endCoord grösser ist
98
99      }
100
101     if ( ( pos1 < segment1->run_types.size()) || (pos2 < segment2->run_types.size()) ) {
102       Füge verbliebene Läufe des jeweiligen Segments zu returnValue hinzu
103     }
104
105     segment1 = returnValue;
106   }
107
108   return returnValue;
109 }
```

ND Dilation

```
1 HierarchicalRLE* dialtion( CHierarchicalRLE* se, vector<short int> qv  ) {
2
3   HierarchicalRLE* dilatedHRLE = dilation1D( 0, se, qv.back() );
4
5   dilatedHRLE = dilation( dilatedHRLE, se, qv, 0 );
6
7   return dilatedHRLE;
8
9 }
```

```
1 HierarchicalRLE* dilation( CHierarchicalRLE* dilatedHRLE, CHierarchicalRLE* se, vector<short int> qv, int si ) {
2
3   HierarchicalRLE* dilatedSubHRLE = new HierarchicalRLE();
4   HierarchicalRLE* tmpSegment, tmpSubSegment;
5   vector<HierarchicalRLE*> tmpSubsegments;
6   qv.pop_back() // entfernt letzten Wert von qv
7
8   if (lower_hrle != NULL ) {
9     for ( run = alle Runs von dilatedHRLE ) {
10
11      if ( RT_DEFINED(run) {
12
13        initialisiere sePos
14
15        for ( ym = alle Koordinaten des Runs ) {
16            tmpSubsegments.clear();
17
18            ermittle alle vom Strukturelement mir einbezogenen Koordinaten sowie aktuell
19            gültige Koordinate sePos im Strukturelement
20
21            for ( qi = vom Strukturelement einbezogenen Koordianten ) {
22
23                orgRun = Lauf im originalen Segment mit Koordinate qi
24
25                if (RT_DEFINED(orgRun)) {
26
27                    ermittle zu sePos zugehöriges Untersegment subSE von se
28                    ermittle definierten Index defIndex von orgRun
29
30                    tmpSegment = dilation1D(defIndex, subSE, qv.back());
31                    if ( lower_hrle != NULL )
32                        tmpSegment = dilation(tmpSegment,subSE,qv,defIndex);
33
34                    tmpSubsegments.push_back(tmpSegment);
35                }
36
37                sePos--;
38            }
```

```
39
40              tmpSubSegment = union(tmpSubsegments);
41
42              dilatedSubHRLE->add(tmpSubSegment);
43          }
44        }
45      }
46
47      dilatedHRLE->lower_hrle = dilatedSubHRLE;
48    }
49
50    return dilatedHRLE;
51  }
```

B.2.4 Erosionsalgorithmus

Wie die Dilation besteht die Erosion ebenfalls aus drei Teilen: einer 1D Erosion, die für die Koordinaten der Voxel in der untersten Dimension entscheidet, ob sie erhalten bleiben, einem Intersection-Algorithmus zur Berechnung der Schnittmenge zweier H-LLK Segmente und dem rekursiven Erosionsalgorithmus, der die Erosion auf den dimensional höheren Segmenten durchführt. Als Eingabe erhält die Erosion ebenfalls ein Strukturelement und die Koordinaten des Anlegepunktes darin.

1D Erosion

Die 1D Erosion erhält ein 1D Strukturelement mit der zugehörigen Koordinate des Anlegepunkts und einen Segment Index für das zu erodierende 1D Segment als Eingabe.

```
1   HierarchicalRLE* erosion1D( int si, CHierarchicalRLE* se, short int nCoord ) {
2
3       HierarchicalRLE* returnValue, temp;
4
5       HierarchicalRLE* segment = getSegmentAt(si);
6
7       for ( run = aktueller Lauf, über alle Läufe von segment ) {
8
9           if (RT_DEFINED(run) {
10              for ( aktKoord, über alle Koordinaten von run ) {
11
12                  passe Koordinaten von se an, so daß nKoord == aktKoord
13
14                  vergleiche das angepaßte Strukturelement mit segment
15
16                  if ( alle im Strukturelement definierten Punkte auch in segment definiert ) {
17                      füge für aktKoord einen RT_DEFINED Lauf zu returnValue hinzu
18                  }
19                  else {
20                      füge nicht definierten Lauf zu returnValue hinzu
21                  }
22              }
23          }
24          else {
25              füge Lauf zu returnValue hinzu
26          }
27      }
28
29      return returnValue
30  }
```

Schnittmenge

Der Algorithmus für die Schnittmenge erhält einen Vektor mit n H-LLK Segmenten als Eingabe und erzeugt aus der Schnittmenge dieser Segmente ein neues Segment.

```
1  HierarchicalRLE* hrleIntersection( vector<HierarchicalRLE*> hrleVector ) {
2
3    if ( hrleVector.size() == 1 ) return hrleVector.back();
4
5    HierarchicalRLE* segment1 = hrleVector.at(0);
6
7    for ( int i = 1; i < hrleVector.size(); i++ ) {
8
9      HierarchicalRLE* segment2 = hrleVector.at(i);
10     HierarchicalRLE* returnValue = new HierarchicalRLE();
11
12     Setze min und max von returnValue auf das kleinere max bzw. größere min von segment1 und segment2
13     returnValue->start_indices.push_back( 0 );
14
15     Ermittle Start- und Endkoordinaten der ersten Läufe der beiden Segmente
16
17     if (startCoord2 > endCoord1)
18         ignoriere Läufe von segment1 bis endCoord1 > startCoord2
19
20     if (startCoord1 > endCoord2)
21         ignoriere Läufe von segment2 bis endCoord2 > startCoord1
22
23     // pos1 und pos2 sind Indices ins Run Type Array der beiden Segmente
24     int pos1 = 0, pos2 = 0;
25
26     while ( ( pos1 < segment1->run_types.size()) || (pos2 < segment2->run_types.size()) ) {
27
28       if ( Ende eines Segments erreicht) break;
29
30       run1 = segment1->run_types.at(pos1);
31       run2 = segment2->run_types.at(pos2);
32
33       if ( RT_DEFINED(run1) && RT_DEFINED(run2) ) {
34
35         if ( RT_DEFINED(run1) ) {
36           returnValue->run_breaks.push_back(kleinste endCoord);
37
38           for ( qi = alle sich überschneidenden Koordinaten der beiden Läufe ) {
39             if ( lower_hrle != NULL ) {
40               returnValue->run_types.push_back(returnValue->lower_hrle->start_indices.size());
41               vector<HierarchicalRLE*> sv;
42               sv.push_back(segment1->getSegment(qi));
43               sv.push_back(segment2->getSegment(qi));
44               returnValue->add(hrleIntersection(sv));
45             }
46             else {
47               returnValue->run_types.push_back(definierten Lauf);
48             }
49           }
50         }
51
52       }
53       else {
54
55           returnValue->run_types.push_back(nicht definierten Lauf);
56           returnValue->run_breaks.push_back(größte endCoord);
57
58       }
59
60       Falls endCoord eines Runs kleiner als returnValue->run_breaks.back(), gehe solange zum
61       nächsten Lauf (pos1++ bzw. pos2++) im jeweiligen Segment, bis endCoord grösser ist
62
63     }
64
65     segment1 = returnValue;
66   }
67
68   return returnValue;
69 }
```

ND Erosion

Die ND Erosion erhält ein Strukturelement mit den Koordinaten des zugehörigen Anlegepunkts und einen Segment Index, initialisiert mit Null, als Eingabe.

```
HierarchicalRLE* erosion( CHierarchicalRLE* se, vector<short int> qv, int si ) {

  HierarchicalRLE* erodedHRLE = new HierarchicalRLE();
  HierarchicalRLE* tmpSegment, tmpSubSegment;
  vector<HierarchicalRLE*> tmpSubsegments;
  qv.pop_back() // entfernt letzten Wert von qv

  if (lower_hrle != NULL ) {
    for ( run = alle Runs von dilatedHRLE ) {

      if ( RT_DEFINED(run) {

        initialisiere sePos

        for ( ym = alle Koordinaten des Runs ) {
            tmpSubsegments.clear();

            ermittle alle vom Strukturelement mir einbezogenen Koordinaten sowie aktuell
            gültige Koordinate sePos im Strukturelement

            for ( qi = vom Strukturelement einbezogenen Koordianten ) {

                orgRun = Lauf im originalen Segment mit Koordinate qi
                seRun = se->getRun(sePos);

                if (RT_DEFINED(seRun) && RT_DEFINED(orgRun)) {

                    ermittle zu sePos zugehöriges Untersegment subSE von se
                    ermittle definierten Index defIndex von orgRun

                    tmpSegment = erosion(subSE, qv, defIndex );

                    tmpSubsegments.push_back(tmpSegment);
                }
                else {
                    tmpSubsegments.clear();
                    break;
                }

                sePos--;
            }

            if ( tmpSubsegments.size() > 0 ) {

                tmpSubSegment = hrleIntersection(tmpSubsegments);

                dilatedSubHRLE->add(tmpSubSegment);

                if (!RT_DEFINED(erodedHRLE->run_types.back()))
                    erodedHRLE->run_types.push_back(erodedHRLE->lower_hrle->start_indices.size());
            }
            else {
                erodedHRLE->run_types.push_back(nicht definierten Lauf);
                erodedHRLE->run_breaks.push_back(ym);
                erodedHRLE->run_breaks.push_back(ym+1);
            }
        }
        erodedHRLE->run_breaks.push_back( Unterbrechung des Laufs );
      }
      else {
          erodedHRLE->run_types.push_back(run);
          erodedHRLE->run_breaks.push_back( Unterbrechung des Laufs );
      }
    }

    dilatedHRLE->lower_hrle = dilatedSubHRLE;
  }
  else erodedHRLE = erosion1D( si, se, short qv.back() );

  return erodedHRLE;
}
```

B.2.5 Skelettierung

In diesem Kapitel sind die wichtigsten Algorithmen zur Berechnung der medialen Achse angegeben. Diese umfassen das Löschen der Punkte, die “Simplizität in der Ebene” und den eigentlichen Ausdünnungsalgorithmus

Mediale Fläche / mediale Achse

```
bool thinning( int checkingPlaneValue ) {

    int                       i, type;

    vector<definedPoint>      pointArray;
    vector<coordValue>        delateablePoints;
    bool                      pointsDeleted;

    pointArray = getDefinedPoints();

    do {

        pointsDeleted = false;

        // Reihenfolge: B, T, E, W, S, N
        for ( type = 1; type < 7; type++ ) {

            for ( i = 0; i < pointArray.size(); i++ ) {

              if ( pointArray.at(i).isDeleted == false ) {

                  if ( isBorderPointOfDirection( pointArray.at(i).coordinate,
                                                 (borderPointType)type ) ) {

                    if ( isFinalPoint( pointArray.at(i).coordinate ) == false ) {

                        if ( isSimplePoint(pointArray.at(i).coordinate) ) {

                            if ( checkingPlane( pointArray.at(i).coordinate,
                                 checkingPlaneValue, (borderPointType)type )) {

                                addToDeleteablePoints( pointArray.at(i).coordinate );

                                pointArray.at(i).isDeleted = true;
                            }

                        }
                    }
                  }

              }

            }

            if ( delateablePoints.size() > 0 ) {

                pointsDeleted = true;

                deleteBorderPoints( delateablePoints );

            }

            delateablePoints.clear();

        }

    } while ( pointsDeleted );

    return true;

}
```

Simplizität in der Ebene

Der Algorithmus erhält die Koordinaten des zu überprüfenden Punktes, einen Wert für die minimale Anzahl verbleibender definierter Punkte für die Ebenen und den Typ des Randpunkts zur Ermittlung der Ebenen als Eingabe.

```
1  bool CHierarchicalRLE::checkingPlane( vector<int> bp, int value, borderPointType type ) {
2
3      short int      numBlackPoints = 0;
4      vector<bool>  n26;
5      vector<bool>  firstPlane, secondPlane;
6
7      n26 = N26( bp );
8
9      ermittle die beiden Checking Planes firstPlane und secondPlane für
10     die bei type angegebene Richtung
11
12     firstPlane[4] = false;
13     secondPlane[4] = false;
14
15     numBlackPoints = 0;
16     for ( i = 0; i < firstPlane.size(); i++ )
17         if ( firstPlane.at(i) ) numBlackPoints++;
18     if ((numBlackPoints) < value) return false;
19
20     numBlackPoints = 0;
21     for ( i = 0; i < secondPlane.size(); i++ )
22         if ( secondPlane.at(i) ) numBlackPoints++;
23     if ((numBlackPoints) < value) return false;
24
25     if ( connectedComponets2D(firstPlane) != 1 ) return false;
26
27     if ( connectedComponets2D(secondPlane) != 1 ) return false;
28
29     return true;
30
31 }
```

Löschen von Punkten

Der Algorithmus zum Löschen der Punkte erhält eine Liste mit zu löschenden Punkten als Eingabe.

```
1  bool deleteBorderPoints( vector<coordValue> deleteablePoints ) {
2
3      CHierarchicalRLE* tmpHRLE = deleteBorderPoints( deleteablePoints );
4
5      if ( tmpHRLE == NULL ) return false;
6
7      copy( tmpHRLE );
8
9      return true;
10 }
```

```
1  CHierarchicalRLE* deleteBorderPoints( vector<coordValue> dpArray ) {
2
3    CHierarchicalRLE*    returnValue = new CHierarchicalRLE();
4    int dpPos = 0;
5
6    tmpHRLE->min = min;
7    tmpHRLE->max = max;
8    tmpHRLE->start_indices.push_back( 0 );
9
10   for ( run = alle Runs von dilatedHRLE ) {
11
12     if ( RT_DEFINED(run) ) {
13       for ( coord = alle Koordinaten des Laufs ) {
14         if ( (coord == dpArray.at(dpPos).coord) &&
15              (dpPos < dpArray.size() ) {
```

```
16
17              if (lower_hrle == NULL) {
18                if (RT_DEFINED(returnValue->run_types.back()))
19                  returnValue->run_types.push_back(nicht definierten Lauf);
20              }
21              else {
22
23                ermittle defIndex für Koordinate coord
24                CHierarchicalRLE* tmpHRLE = getSegment(defIndex);
25
26                bool deleted = tmpHRLE->deleteBorderPoints(
27                                deleteablePoints.at(dpPos]).subCoords );
28
29                if ( deleted ) {
30                  returnValue->run_types.push_back(nicht definierten Lauf);
31                }
32                else {
33                  if (!RT_DEFINED(returnValue->run_types.back()))
34                    returnValue->run_types.push_back(returnValue->lower_hrle->run_types.size());
35
36                  returnValue->lower_hrle->add(tmpHRLE);
37                }
38
39              }
40
41              if (letzter Lauf war ein anderer als aktuell hinzugefügter)
42                returnValue->run_breaks.push_back(coord);
43
44            }
45            else {
46              if (!RT_DEFINED(returnValue->run_types.back())) {
47
48                if (lower_hrle == NULL)
49                  returnValue->run_types.push_back(run);
50                else
51                  returnValue->run_types.push_back(returnValue->lower_hrle->start_indices.size());
52
53                if (letzter Lauf war ein anderer als aktuell hinzugefügter)
54                  returnValue->run_breaks.push_back(coord);
55
56              }
57            }
58          }
59        }
60        else {
61          returnValue->run_types.push_back(run);
62        }
63
64        returnValue->run_breaks.push_back( Unterbrechung des Laufs );
65
66    }
67  }
```

Berechnung eines Ausdünnungsschritts

Dieser Algorithmus wird benötigt, um für die Skelettierung der erweiterten H-LLK Niveaumenge einen Subzyklus zu berechnen. Er erhält als Eingabe einen Wert für die "Simplizität in der Ebene"-Bedingung und den Typ Randpunkt, der gelöscht werden soll.

```
1  bool thin( int checkingPlaneValue, borderPointType type ) {
2
3      vector<definedPoint>      pointArray;
4      vector<coordValue>        delateablePoints;
5      bool                      pointsDeleted;
6
7      pointArray = getBorderOfTypePoints(type);
8
9      pointsDeleted = false;
10
11     for ( i = 0; i < pointArray.size(); i++ ) {
12
13       if ( pointArray.at(i).isDeleted == false ) {
14
15         if ( isFinalPoint( pointArray.at(i).coordinate ) == false ) {
16
17             if ( isSimplePoint(pointArray.at(i).coordinate) ) {
18
19                 if ( checkingPlane( pointArray.at(i).coordinate,
20                       checkingPlaneValue, (borderPointType)type )) {
21
22                     if ((pointArray.at(i).coordinate.back() > min) &&
23                         (pointArray.at(i).coordinate.back() < (max-1)))
24                       addToDeleteablePoints( pointArray.at(i).coordinate );
25
26                     pointArray.at(i).isDeleted = true;
27                 }
28
29             }
30         }
31
32       }
33
34     }
35
36     if ( delateablePoints.size() > 0 ) {
37
38         pointsDeleted = true;
39
40         deleteBorderPoints( delateablePoints );
41
42     }
43
44     delateablePoints.clear();
45
46     return pointsDeleted;
47
48 }
```

B.3 Algorithmen der erweiterten H-LLK Niveaumenge

B.3.1 Morphologische Operatoren

Der Algorithmus zur Berechnung morphologischer Operationen erhält ein Strukturelement mit Koordinatenvektor für den Anlegepunkt sowie die maximale Anzahl z-Koordinaten für einen z-Block und die Art der morphologischen Operation als Eingabe.

```
1  bool morphOperation( CHierarchicalRLE* se, vector<short int> qv, int numZCoords, morphType type ) {
2
3    backupFiles();
4
5    short int           numZBlocks;
6    CHierarchicalRLE*   segmentZ = new CHierarchicalRLE();
7    short int           numZBlocks;
8    short int           startCoord, endCoord, firstLoadedCoord, lastLoadedCoord;
9    int                 defYIndex = 0;
10   short int           overlapLeft, overlapRight;
11
12   initialisiere segmentZ
13
14   overlapLeft = qv.back() - se->min;
15   overlapRight = se->max - qv.back() -1;
16
17   ermittle anhand numZCoords die Anzahl der z-Blöcke numZBlocks
18
19   ermittle startCoord und endCoord des ersten z-Blocks
20
21   for ( zBlock = alle z-Blöcke ) {
22
23     if ( (startCoord - overlapLeft) > minZ ) firstLoadedCoord = startCoord - overlapLeft;
24     else firstLoadedCoord = startCoord;
25     if ( (endCoord + overlapRight) < maxZ ) lastLoadedCoord = endCoord + overlapRight;
26     else lastLoadedCoord = endCoord;
27
28     if ( loadPartialObject( firstLoadedCoord, lastLoadedCoord ) ) {
29
30         switch (type) {
31           case EROSION:
32               partialHrleObject->erosion( structureElement, qv );
33               break;
34           case DILATION:
35               partialHrleObject->dilation( structureElement, qv );
36               break;
37           case OPENING:
38               partialHrleObject->opening( structureElement, qv );
39               break;
40           case CLOSING:
41               partialHrleObject->closing( structureElement, qv );
42               break;
43         }
44
45         rewriteLoadedParts( startCoord, endCoord, erodedSegmentZ, &defYIndex );
46     }
47
48     aktualisiere startCoord und endCoord
49   }
50
51   rewriteMasterFile( segmentZ );
52
53   return true;
54 }
```

B.3.2 Skelettierung

Der Skelettierungsalgorithmus der erweiterten H-LLK Datenstruktur nutzt die Funktion zur Berechnung eines Ausdünnungsschritts, um das Skelett zu erzeugen. Als Eingabe erhält er einen Wert, der für die "Simplizität in der Ebene"-Bedingung entscheidet, ob die mediale

Achse oder die mediale Fläche berechnet werden soll, sowie die maximal Anzahl z-Koordinaten für einen z-Block.

```
1  bool thinning( int checkingPlaneValue , int numZCoords ) {
2
3    short int            numZBlocks;
4    CHierarchicalRLE*    segmentZ = new CHierarchicalRLE();
5    short int            numZBlocks;
6    short int            startCoord , endCoord , firstLoadedCoord , lastLoadedCoord;
7    int                  defYIndex = 0;
8
9    do {
10
11     // Reihenfolge: B, T, E, W, S, N
12     for ( int type = 1; type < 7; type++ ) {
13
14     segmentZ = NULL;
15
16     ermittle anhand numZCoords
17
18     backupFiles();
19
20     ermittle startCoord und endCoord des ersten z-Blocks
21
22     initialisiere segmentZ
23
24     defYIndex = 0;
25
26     pointsDeleted = false;
27
28       for ( zBlock = 0; zBlock < numZBlocks; zBlock++ ) {
29
30           if ( (startCoord -1) > minZ ) firstLoadedCoord = startCoord - 1;
31           else firstLoadedCoord = startCoord;
32
33           if ( (endCoord+1) < maxZ ) lastLoadedCoord = endCoord + 1;
34           else lastLoadedCoord = endCoord;
35
36           if ( loadPartialObject( firstLoadedCoord , lastLoadedCoord , false , true ) ) {
37
38             if (pointsDeleted == false)
39                 pointsDeleted = partialHrleObject->thin( checkingPlaneValue , (borderPointType)type );
40             else
41                 partialHrleObject->thin( checkingPlaneValue , (borderPointType)type );
42
43             rewriteLoadedParts( startCoord , endCoord , segmentZ , &defYIndex );
44
45           }
46
47           aktualisiere startCoord und endCoord
48
49       }
50
51       rewriteMasterFile( segmentZ );
52     }
53
54   } while ( pointsDeleted );
55
56 }
```