Diplomarbeit

– Entwicklung einer 64 bit-Software zur Separation morphologischer Strukturen in dreidimensionalen Bildern –

an der
Fachhochschule für Wirtschaft (FHW) Berlin
im Bereich Berufsakademie

Vorgelegt von:	Stefan Kirste
Studienrichtung:	Informatik
Jahrgang:	2001

Betreuer/Gutachter:	Dipl.-Phys. Alexander Rack Prof. Dr.-Ing. Michael Greiff
Zeitraum der Diplomarbeit:	13. April 2004 – 13. Juli 2004
Ausbildungsbetrieb:	Hahn–Meitner–Institut Berlin GmbH
letzte Änderung:	12. Juli 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Ich erkläre ehrenwörtlich:

1. dass ich meine Diplomarbeit selbstständig verfasst habe,
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe,
3. dass ich meine Diplomarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Zusammenfassung

Diese Arbeit dient dem Erreichen des Grades *Dipl.-Ing. (BA) Fachrichtung Informatik* an der Fachhochschule für Wirtschaft Berlin im Fachbereich Berufsakademie. Die Durchführung erfolgte durch Stefan Kirste in der Arbeitsgruppe „Synchrotron-Tomographie“ in der Abteilung Werkstoffe (SF3) im Bereich Strukturforschung am Hahn-Meitner-Institut Berlin. Die Arbeit wurde am 13.7.2004 in der Berufsakadmie Berlin eingereicht.

Die vorliegende Diplomarbeit beschäftigt sich mit der Segmentierung von 3D-Bildern. Die Spezialisierung liegt dabei in der Separation von morphologischen Strukturen in tomographischen Aufnahmen, welche mittels Synchrotron-Tomographie erstellt werden.

Die Hauptaufgabe dieser Arbeit liegt in der Erstellung einer Bibliothek, welche Funktionen zur parametrisierten Segmentierung bereitstellt. Auf Grund der Beschaffenheit der Datensätze ergeben sich zwei besondere Anforderung bei der Umsetzung der Software:

1. Durch die extreme Größe der Tomogramme soll die Entwicklung der Bibliothek neben den zur Zeit geläufigen 32-Bit Architekturen auch für 64-Bit-Prozessoren und -Betriebssysteme zur erweiterten Arbeitsspeichernutzung implementiert werden. Die Ressourcenverwaltung muss durch eine optimale Speicherausnutzung geprägt sein, um auch möglichst große Bilder analysieren zu können.

2. Tomographische Abbildungen sind durch eine starke Inhomogenität gekennzeichnet. Deshalb müssen Algorithmen zur rauschfreien Segmentierung entwickelt und bei der Umsetzung mit eingebracht werden.

Neben der Implementierung der Software zur parametrisierten Separation werden zusätzlich erste Verfahren zur (semi-)automatischen Segmentierung beschrieben. Diese können im Anschluss an die Diplomarbeit zur Erweiterung der Bibliothek genutzt werden.

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Inhalt der Diplomarbeit ist die Erstellung einer Software-Bibliothek („a4ibool“), die sowohl für 32-Bit- als auch 64-Bit-Systeme geeignet sein soll.

Die Funktion der Bibliothek besteht in der parametrisierten Separation von Objekten innerhalb von dreidimensionalen Bildern, welche wiederum als Graustufenwerte (8-Bit Charakterwerte) vorliegen. Es gilt dabei einen Algorithmus zu realisieren, der ein minimales Binarisierungsrauschen garantiert.

Vorrang bei der Umsetzung gegenüber einer minimalen Rechenzeit hat eine optimale Ausnutzung des Arbeitsspeichers, um möglichst große Bilder analysieren zu können. Dies gilt vor allem im Hinblick auf die Speicherlimitierung auf 32-Bit-Systemen.

Im Anschluss sollen erste Konzepte zur Semi-Automatisierung des Binarisierungsvorganges entwickelt und getestet werden.

1.2 Grundlegende Begriffe

CCD CCD steht für „Charge Coupled Device“ (ladungsgekoppeltes Bauelement). Ein CCD-Chip ist ein Bauelement der Halbleitertechnik, welches u.a. bei modernen Digitalkameras eingesetzt wird. Ein CCD-Chip besteht aus Lichtsensoren (Pixeln), die beim Eintreffen von (Licht-) Photonen freie Ladungsträger erzeugen. Die Ladungsträger werden als äquivalenter Strom ausgelesen. Der digitalisierte Strom (Counts pro Zeiteinheit) entspricht der Intensität des eingefallenen Lichts. (vgl. [17])

Histogramm Das Histogramm repräsentiert die Helligkeitsverteilung eines Bildes. In ihm werden die einzelnen Helligkeitswerte (im Graustufenbild: Graustufenwerte) über ihre Anzahl aufgetragen. Das Histogramm kann in vielen Bereichen der Bildverarbeitung verwendet werden. Beispiele sind die optimale Anpassungen von Kontrast und Helligkeit sowie die Schwellwertbestimmung bei der Binarisierung (vgl. Kapitel 3 und 4).

Morphologie/Morphologische Strukturen/Morphologische Operationen
„Die Morphologie (zu Deutsch Formenlehre) ist die Wissenschaft von der

> Form, Gestalt und Organisation. In der Biologie bezeichnet die Morphologie etwa die Lehre vom Aufbau von Pflanzen und Tieren. In der Bildverarbeitung versteht man unter morphologischen Operationen solche, die sich auf die äußere Form eines Objektes beziehen.“ [4]

Elementare morphologische Operationen bzw. Transformationen sind die Erosion (lat.: erodere, abnagen/abtragen) und Dilatation (lat.: dilatare, ausbreiten/dehnen), bei denen die Morphologie von Strukturen gezielt manipuliert wird. Diese werden im Abschnitt 3.2.2 näher beschrieben. Aus den Basisoperationen ergeben sich weitere morphologische Transformationen wie zum Beispiel deren Verknüpfung (Öffnen und Schließen) oder der „Fill“- und „Connected“-Algorithmus (s. auch 3.2.2).

Pixel/Voxel Ein Pixel ist die kleinste, darstellbare Einheit auf einem Bildschirm, auf dem Papier (Drucker) oder die kleinste Zelle eines Bildsensors. Erweitert man die Definition auf den dreidimensionalen Raum, so erhält man den Voxel. Diese Bezeichnung dient zur theoretischen Beschreibung der kleinsten Einheit zum Beispiel bei Tomogrammen. Die Darstellung am Bildschirm bzw. auf dem Papier erfolgt dann wieder durch Projektion auf den 2D-Raum.

Radiogramm/Tomogramm Bei einem Radiogramm handelt es sich um eine durch eine Röntgenaufnahme erstellte 2D-Abbildung. Die im Bild gespeicherten Informationen stellen den durch ein Objekt abgeschwächten Röntgenstrahl dar. Das Tomogramm beinhaltet einen rekonstruierten 3D-Datensatz aus einzelnen Radiogrammen, welche unter verschiedenen Winkeln aufgenommen wurden. [9]

1.3 Ausgangspunkt

1.3.1 Das Partnerunternehmen

Das Hahn-Meitner-Institut Berlin (HMI) ist eine naturwissenschaftliche Forschungseinrichtung. Die Schwerpunkte der wissenschaftlichen Arbeiten liegen im Bereich Solarenergie- und Strukturforschung. Zusätzlich werden zwei Großanlagen für verschiedene Zwecke betrieben: ein Forschungsreaktor (BER II) zur Erzeugung von Neutronenstrahlen sowie ein Teilchenbeschleuniger (ISL). Die Anwendungsgebiete reichen von der Untersuchung verschiedener Materialien über die Verifizierung der Echtheit von antiken Kunstwerken bis hin zur Augentumortherapie. Des Weiteren besteht eine Kooperation mit der BESSY GmbH, der Berliner Elektronenspeicherring-Gesellschaft für Synchrotronstrahlung mbH, deren Anlage sich in Berlin-Adlershof befindet.

In der Abteilung Werkstoffe (SF 3) werden moderne Materialsysteme hinsichtlich ihrer Struktur und den daraus folgenden Eigenschaften untersucht. Da die Gewinnung entscheidender Ergebnisse im Wesentlichen im Micro- und Nanometerbereich liegt, müssen entsprechende Verfahren zur Untersuchung eingesetzt werden. Die hier entstandenen Erkenntnisse fließen dann in die Entwicklung von zukünftigen Materialien ein.

Eine Arbeitsgruppe der Abteilung Werkstoffe beschäftigt sich mit der Untersuchung von Materialien mittels Synchrotron-Tomographie. In Kooperation mit der Bundesanstalt für Materialforschung und -prüfung (BAM) steht dafür ein Messplatz am Elektronenspeichering

(BESSY II) in Berlin-Adlershof zur Verfügung [19]. Dieser wird durch das HMI in Gemeinschaft mit der BAM betreut. Dabei wird nicht nur eigene Forschung betrieben, sondern es werden auch Messungen für Industrie, Hochschulen und anderen Forschungseinrichtungen durchgeführt.

Neben der Synchrotron-Tomographie werden ab Mitte dieses Jahres auch Untersuchungen mittels Neutronen-Tomographie durchgeführt. Dazu befindet sich ein Messplatz am Forschungsreaktor BER II im Aufbau. Die wissenschaftlichen Instrumenten am BER II werden u.a. durch das Berliner Zentrum für Neutronenstreuung (BENSC) entwickelt und betreut.

1.3.2 Synchrotronstrahlung

Die Synchrotronstrahlung stellt eine „besondere“ Art von Licht dar. Im normalen Sprachgebrauch wird der Begriff „Licht“ dazu verwendet, um den für das menschliche Auge sichtbaren Anteil des elektromagnetischen Spektrums zu charakterisieren. Dieses umfasst allerdings einen viel größeren Teil – angefangen von den Radiowellen, über den Infrarotbereich bis hin zu den Ultravioletten- und Gamma-Strahlen. Dabei werden die spektralen Eigenschaften der elektromagnetischen Strahlen durch ihre Energie und Wellenlängen spezifiziert. Dabei gilt die Plancksche Formel:

$$E = h * \nu, \qquad \text{mit} \quad \nu = \frac{c}{\lambda}$$

$E =$ Energie, $\quad h =$ Plack'sches Wirkungsquantum, $\quad \nu =$ Frequenz,
$c =$ Lichtgeschwindigkeit $\quad$ und $\quad \lambda =$ Wellenänge

Das heißt, je kleiner die Wellenlänge ist, desto größer ist die Energie der Strahlung (vgl. Abbildung 1.1, [2] und [8]).

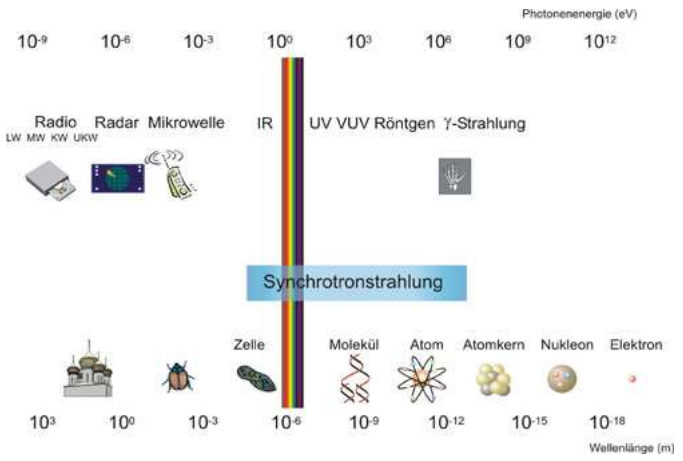


Abbildung 1.1: Elektromagnetisches Spektrum: Das elektromagnetische Spektrum reicht von den Radiowellen über die Infrarot- und Ultraviolettstrahlung bis hin zu den Gammastrahlen. Die für den Menschen sichtbaren Wellenlängen liegen im Bereich von 720 nm (Rot) bis 380 nm (Violett). Synchrotronstrahlung kann über ein Spektrum von 10^{-5} bis 10^{-12} m erzeugt werden.[2]

Erzeugt wird die Synchrotronstrahlung durch Beschleunigung von geladenen Teilchen in elektrischen Feldern. Dabei werden Elektronen benutzt, da sie auf Grund ihrer geringen Ladung und Masse am einfachsten zu beschleunigen sind. Die Beschleunigung erfolgt bei modernen Anlagen durch Mehrfachbeschleuniger, in denen die gewünschte Geschwindigkeit der Elektronen durch wiederholtes Zuführen von Energie erreicht wird. Eine Möglichkeit dazu stellt der Linearbeschleuniger dar. In ihm werden mehrere elektrische Felder hintereinander in Reihe geschaltet. Bei diesem Verfahren ist jedoch die Beschleunigungsstrecke begrenzt, sodass lediglich ein Strahl mit vergleichsweise geringer Energie entsteht. Eine Alternative dazu stellt das Synchrotron bzw. der Ringbeschleuniger dar. Durch Umlenken der Elektronen auf eine Kreisbahn mittels Ablenkmagneten können diese beliebig oft durch die Beschleunigerstrecke geführt werden und somit höhere Energien erreichen.

Beim Elektronenspeicherring BESSY II werden die Elektronen über eine Kombination aus beiden Varianten – Linearschleuniger und Synchrotron – beschleunigt. Dazu wird zunächst durch eine Elektronenkanone ein Elektronenstrahl mit einer Energie von 70 keV erzeugt. Dieser wird dann durch einen Linearbeschleuniger (Mikrotron) vorbeschleunigt und danach mit einen Zyklotron auf seine endgültige Geschwindigkeit gebracht. Beim Verlassen des Zyklotrons haben die Elektronen eine Energie von 1,7 GeV und erreichen eine Geschwindigkeit, welche der des Lichtes nahe kommt. Um den Strahl für Messungen über einen längeren Zeitraum zur Verfügung zu stellen, wird dieser nach der Beschleunigung in einen Speicherring gegeben. In diesem werden die Elektronen mittels Ablenkmagneten auf der Kreisbahn gehalten. Zur Aufrechterhaltung der Strahlenergie sind ein Vakuum sowie weitere elektrische Felder vorhanden. [2]

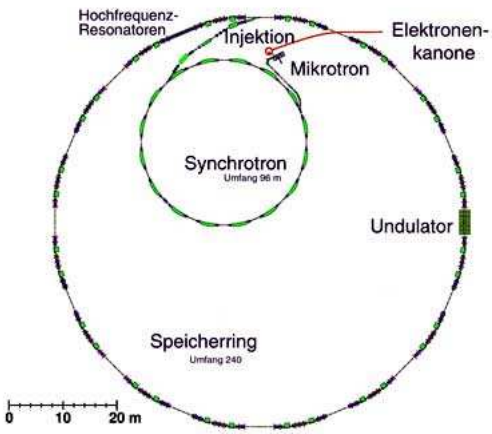


Abbildung 1.2: Elektronenspeicherring BESSY II - Berlin Adlershof [2]: Am BESSY werden die Elektronen über eine Elektronenkanone erzeugt, über eine Linearbeschleuniger (Mikrotron) vorbeschleunigt und mittels eines Synchrotrons auf die endgültige Geschwindigkeit und Energie gebracht. Danach wird der Elektronenstrahl zum Bereitstellen für die Messplätze in den eigentlichen Speicherring gegeben.

1.3.3 Computer-Tomographie

Die Computer-Tomographie (CT) funktioniert prinzipiell wie die Aufnahme von Röntgenbildern. Röntgenstrahlen haben die Eigenschaft, dass sie verschiedene Materialien unterschiedlich stark durchdringen. Aus diesem Grund werden die zu untersuchenden Objekte in einen Röntgenstrahl gehalten. Der aus der Probe austretende, abgeschwächte Strahl enthält dann Information über das Innere des 3D-Objektes, projiziert auf ein 2D-Bild.

Bekannt ist das Prinzip insbesondere aus der Medizin. Bei der Computer-Tomographie wird ein Röntgenstrahl erzeugt, welcher wiederum 2D-Abbilder von Körperregionen erstellt. Durch Veränderung der Position der Röntgenquellen werden Aufnahmen aus verschiedenen Winkeln gemacht, aus denen dann eine 3D-Abbildung des Körpers rekonstruiert werden kann.

Das Verfahren der Synchrontron-Tomographie entspricht dem des CTs. Auch hier wird die Eigenschaft der Abschwächung von Röntgenstrahlen durch unterschiedliche Materialien genutzt. Der große Unterschied zur Computer-Tomographie liegt in der hohen Auflösung. Bei der Synchrotron-Tomographie ist eine Auflösung von bis zu 250 Pixel pro Millimeter (3,5 μm pro Pixel) und mehr möglich. Des Weiteren können durch die hohe Qualität des Strahles auch sehr kleine Systeme sowie verschiedene Materialien innerhalb von Proben und schnell ablaufende Ereignisse aufgenommen werden. [9]

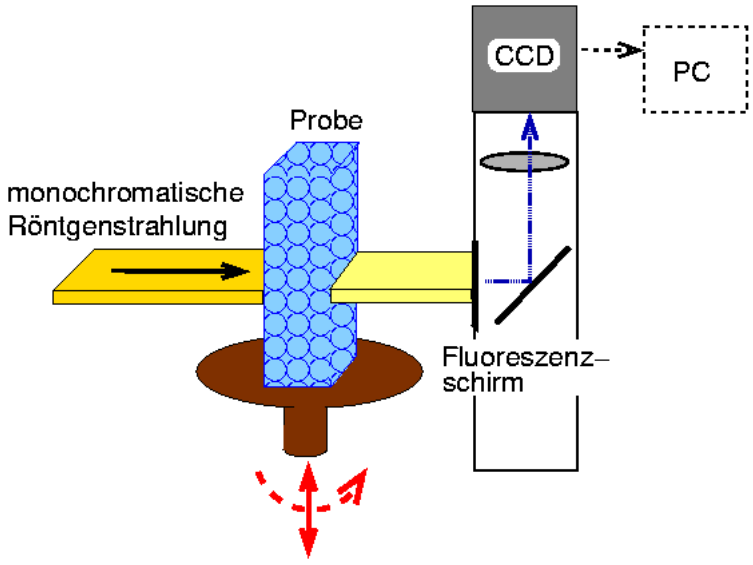


Abbildung 1.3: Schematische Darstellung des Messplatzes am BESSY II: Die Probe ist auf einem drehbaren Teller gelagert. Sie wird von Röntgenstrahlen durchdrungen, welche durch einen fluoreszierenden Schirm sichtbar gemacht werden. Die dauerhafte digitale Speicherung der Aufnahmen, welche zur späteren Rekonstruktion eines 3D-Datensatzes aus unterschiedlichen Winkel gemacht werden, erfolgt durch eine CCD-Kamera. [19]

Zur Untersuchung von verschiedenen Problemen wurden durch das HMI und die Bundesanstalt für Materialforschung und -prüfung ein Messplatz am Elektronenspeicherring BESSY II eingerichtet, welcher die Voraussetzungen für den soeben beschriebenen Vorgang der Computer-Tomographie schafft: Die zu untersuchende Probe befindet sich auf einem drehbaren Teller, welcher eine Rotation in der XY-Ebene erlaubt. Zusätzlich kann der Teller in

X-, Y- und Z-Richtung verschoben werden. Dies wird zur Justierung, zur Minimierung von Aufnahmefehlern sowie zum Abbilden von großen Proben benötigt. Der Röntgenstrahl durchdringt die Probe in der Z-Ebene und wird anschließend mittels eines fluoreszierenden Schirms sichtbar. Um die Abbildung (Radiogramm) dauerhaft digital zu speichern, wird eine CCD-Kamera verwendet, welche über eine Vergrößerungsoptik eine Aufnahme von dem Schirm macht. Um später einen dreidimensionalen Datensatz zu erhalten, müssen durch Drehung des Probentellers verschiedene Radiogramme aus unterschiedlichen Winkeln erstellt werden.[19]

1.3.4 Vom Tomogramm zur 3D-Bildanalyse

Nachdem die Radiogramme, wie oben beschrieben, erstellt wurden, müssen diese zur Informationsgewinnung weiterverarbeit werden. Dabei muss insbesondere auf die Größe der Daten geachtet werden, damit die Bearbeitung auf der einen Seite in endlicher Zeit möglich ist und auf der anderen Seite die Datenmenge nicht die Bearbeitungsmöglichkeit des Computers übersteigt. Weitere Details zu dieser Problematik werden in den Abschnitten 2.1 und 2.2 diskutiert.

Der erste Schritt besteht in der Rekonstruktion der 3D-Tomogramme aus den Radiogrammen. Mit Hilfe der „Gefilterten Rückprojektion“ [9] wird aus den 2D-Radiogrammen das 3D-Tomogramm berechnet. Mit den so gewonnenen Informationen können bereits erste Auswertungen, insbesondere durch „einfaches“ Betrachten des 3D-Objekts mit 3D-Rendering-Programmen[1] und dem Erstellen von Histogrammen erworben werden.

Sowohl beim ersten als auch bei allen folgenden Schritten wird versucht, die Datenmenge zu begrenzen. Die geschieht vor allem durch Festlegen einer „Region of Interest“ (ROI) im Tomogramm, ein bestimmter Bereich auf den sich weitere Untersuchungen konzentrieren sollen. Des Weiteren wird an dieser Stelle der Datensatz, der nach der Rekonstruktion in Form von Float-Werten (4 Byte) vorliegt, in Charakter-Werte (1 Byte) umgewandelt. Dies ist möglich, da sich die Dynamik der Graustufenwerte auf einen kleinen Bereich konzentriert. Skaliert man den wesentlichen Float-Graustufenbereich auf 8 Bit, so wird die Dynamik bei dieser Speichertiefe hinreichend genau abgebildet.

Das weitere Vorgehen verlangt eine Trennung verschiedener morphologischer Strukturen, zum Beispiel von Rissen innerhalb eines Verbundwerkstoffes, in der Abbildung des Probenobjekts. Dazu wird ein Verfahren angewendet, welches als Binarisierung (ausführliche Beschreibung in den Kapiteln 3 und 4) bezeichnet wird. Nach dem Binarisieren können Größenverteilungen bestimmt werden, welche quantitative Aussagen über das Vorkommen verschieden großer Objekte machen.

Der letzte Schritt, welcher sich wiederum in beliebig viele Unterschritte unterteilen lässt, beinhaltet das Anwenden von morphologischen Transformationen auf den Datensatz zur Bestimmung verschiedener Eigenschaften des Testobjektes. Eine Möglichkeit besteht zum Beispiel in der Bestimmung der minimalen Zellwandstärke von schaumähnlichen Strukturen [14].

[1] Bespiele für 3D-Rendering Programme: VGStudio Max (Volume Graphics GmbH) oder AVS (Advanced Visual Systems Inc.)

Kapitel 2

Vorbetrachtungen

Zunächst sollen einige Vorbetrachtungen zu den vorhandenen Daten und der Programmierumgebung in Bezug zur Aufgabenstellung durchgeführt werden. Des Weiteren wird ein konkretes Anwendungsbeispiel vorgestellt, um verschiedene Schritte anschaulicher darstellen zu können. Sämtliche bei den Vorbetrachtungen gewonnenen Erkenntnisse liefern eine Grundlage für spätere Entscheidungen hinsichtlich der Algorithmenwahl sowohl bei der parametrisierten als auch bei der dynamischen Segmentierung und der Implementierung der Bibliothek.

2.1 32- vs. 64-Bit Architektur

In erster Linie gilt die Diplomarbeit der Erstellung einer 64-Bit-Software (vgl. Abschnitt 1.1). Neben dem Einsatzgebiet in einem reinen 64-Bit-Programm soll die Bibliothek auch auf 32-Bit-Systemen arbeiten. Dieser Zusatz liegt vor allem in der zur Zeit noch weiten Verbreitung von 32-Bit-Architekturen begründet. Durch die unabhängige Programmierung können die zur Zeit aktuellen Rechner (vgl. Testrechner Anhang C) für den Einsatz der zu erstellenden Bibliothek verwendet werden.

Um eine Kompatibilität zwischen beiden Architekturen (32- und 64-Bit) zu gewährleisten, müssen verschiedene Aspekte berücksichtigt werden: Das Hauptproblem stellt die unterschiedliche Darstellung der Datentypen dar. Je nach Architektur, Betriebssystem und Compiler können diese in der Größe variieren. Als Beispiel soll hier der Variablentyp „integer“ betrachtet werden. Bei 32-Bit-Betriebssystemen belegen Integervariablen im Allgemein einen Speicherplatz von 32 Bit bzw. 4 Byte. Im Gegensatz dazu können diese bei 64-Bit-Systemen 64 Bit bzw. 8 Byte groß sein. Bei dieser Unterscheidung muss jedoch berücksichtigt werden, dass die soeben getroffenen Verallgemeinerungen nicht auf alle Rechnerarchitekturen zutreffen. Ein Gegenbeispiel ist durch Testrechner 3 – dem Compaq SC45 Compute Cluster – gegeben. Bei den Alpha-Prozessoren EV 68, auf denen ein Tru64 Unix läuft, haben Integervariablen lediglich ein Länge von 4 Byte. Dies entspricht der Speicherbelegung von 32-Bit-Systemen.

Auf Grund der unterschiedlichen Repräsentation von Variablentypen in unterschiedlichen Betriebssystemen ergeben sich verschiedene Probleme, welche bei der Umsetzung des Programms berücksichtigt werden müssen: Eine einfache und meist auch relativ schnelle Variante Variablen zu prüfen bzw. zu verändern sind sogenannte Bitoperationen. Bei diesen Operationen werden Variablen auf unterster Ebene, der Bitebene, manipuliert oder abgefragt. Die

bekanntesten und am häufigsten eingesetzten Operationen sind Bitshiftoperationen und Bitmasken. Beide Operationen werden meist dazu verwendet, um einzelne Bits aus Variablenwerten zu separieren und auf Bedingungen bzw. Zustände hin zu überprüfen. Ein weiteres Problem stellen direkte oder indirekte Type-Cast-Operationen (Typenumwandlungen) dar, bei denen Variablen von unterschiedlichen Typen aufeinander abgebildet werden. [10]

Zur Betrachtung dieser Problematik soll ein Beispielprogramm (s. Anhang A.1) analysiert werden: Bei der Ausführung auf verschiedenen Rechnerarchitekturen ergaben sich folgende Ausgaben:

Listing 2.1: Ausgabe des Testprogramms 64bit-test.c auf einem Alpha EV5/EV68 Prozessor, 400/1001MHz, mit Tru64 Unix 5.1B

```
1 Vor Bitshifting:         FFFFFFFF
2 Zwischendurch:           FFFFFFFF00
3 Nach Bitshifting:        FFFFFFFF
4
5 Original: FFFFFFFF
6 Kompliment: FFFFFFFF00000000
7
8 15
9 Segmentation fault
```

Listing 2.2: Ausgabe des Testprogramms 64bit-test.c auf einem Xeon Prozessor, 2600MHz, mit S.u.S.E. 9.0 32-Bit

```
1 Vor Bitshifting:         FFFFFFFF
2 Zwischendurch:           FFFFFF00
3 Nach Bitshifting:        FFFFFF
4
5 Original: FFFFFFFF
6 Kompliment: 0
7
8 15
9 17
```

In den Listings ist zu erkennen, dass die Ausführung des gleichen Quellcodes bei den Testrechnern mit einer 64-Bit CPU und Betriebssystem ein anderes Ergebnis der durchgeführten Operationen hervorrufen als bei den 32-Bit-Rechner. Alle Unterschiede sind auf eine verschiedene Repräsentation von Datentypen zurückzuführen.

Bei der ersten Operation – dem Bitshifting – werden die Bits der Variable „l“ um acht Stellen nach links, in Richtung der höherwertigen Bits, verschoben. Da der Variablentyp „long“ unter Tru64 Unix durch acht Byte repräsentiert wird, wird die Verschiebung des 4. Bytes (acht Bits) in dem 5. Byte gespeichert. Beim Zurückschieben in Richtung der niederwertigen Bits bleibt somit die Information des ursprünglichen 4. Bytes vorhanden, so dass der Wert nach der doppelten Verschiebung mit Richtungsänderung gleich bleibt. Bei der Ausführung auf einem 32-Bit-Betriebssytem (S.u.S.E 9.0) gehen die Information des 4. Bytes verloren, da hier die Repräsentation eines „long“-Datentyps auf vier Bytes begrenzt ist. Beim Zurückschieben in Richtung der niederwertigen Bits werden die oberen Bits durch Nullen aufgefüllt, so dass der neue Wert entsprechend kleiner wird.

Beim Verwenden einer weiteren Bitoperation – dem Komplement – kommt es bei der Ausführung ebenfalls zu unterschiedlichen Ergebnissen. Die Bildung des Einer-Komplements, welches in diesem Fall berechnet wird, erfolgt durch bitweise Negation. Die Begründung für

die verschiedenen Ausgaben liegt ebenso in der größeren Speichertiefe des „long“-Datentyps in einem 64Bit-Betriebssystem. Da die oberen 4 Byte im Beispiel Null sind, ergibt das Ergebnis nicht wie bei 32 Bit-Betriebssystemen Null sondern $2^{64} - 2^{32}$ oder FFFFFFFF00000000hex.

Das letzte Beispiel im Testprogramm (siehe Anhang A.1) zeigt eine verkettete Liste, deren Elemente durch zwei unterschiedliche Strukturen dargestellt werden. Bei der ersten Struktur „_xy“ wird die Referenz auf das nächste Element durch einen Zeiger vom Typ der Struktur gespeichert. In der zweiten Struktur wird die Adresse des nächsten Wertes durch einen „unsigned int“ abgelegt. Die zweite Variante stellt eine einfache Lösung dar, funktioniert aber nur, wenn die Länge des Datentyps „int“ der Länge eines Zeigers auf der entsprechenden Hardwarearchitektur entspricht. Bei den 64-Bit-Prozessoren (Alpha EV5 und EV68) ist die Länge der Zeiger im Betriebssystem 64-Bit, da diese benötigt wird, um den kompletten theoretischen Adressraum anzusprechen. Wie jedoch bereits am Anfang dieses Abschnittes beschrieben, haben Variablen dieses Typs unter Tru64 Unix lediglich eine Länge von 4 Byte. Als Konsequenz ergibt sich, dass beim Casten der ersten Struktur auf die zweite die Zuordnung der einzelnen Variablenwerte nicht mehr übereinstimmt, da die Gesamtlänge der einzelnen Strukturen nicht identisch ist. Beim Zugriff auf die Werte der gecasteten Struktur werden somit Speicheradressen ausgelesen, deren Werte ungültig sind, so dass es zu nicht erwarteten Ausgaben kommt. Im ungünstigsten Fall wird über die verkettete Liste auf Speicheradressen zugegriffen, deren Inhalte Teil des Betriebssystems oder anderer Programme sind. Dies kann zu nicht kontrollierbaren Reaktionen führen – im schlimmsten Fall zum Absturz des Betriebssystems. Aus diesem Grund verfügen moderne Betriebssysteme über Schutzmechanismen, sodass das fehlerhafte Programm mit einer Speicherzugriffsverletzung beendet wird.

Um ein Programm zu erstellen, das sowohl auf einem 32-Bit-Computer/Betriebssystem als auch unter 64-Bit läuft, müssen soeben beschriebene Randeffekte vermieden werden. Eine Möglichkeit besteht in dem Umgehen entsprechender Operationen. Da dies jedoch nicht in jedem Fall möglich ist, müssen andere Wege gefunden werden. Abhilfe bietet das Benutzen von Datentypen wie Charakter, deren Länge fest vorgeschrieben ist, und somit unter allen Betriebssystemen gleich ist. Der Nachteil hier liegt in der geringen Speichertiefe, da Charakterwerte lediglich eine Länge von acht Bit besitzen. Alternativ besteht die Möglichkeit, ein Array von Charakterwerten zu benutzen, um auch größere Werte darstellen zu können. Nachteilig ist hier jedoch die Ausführen von einfachen Operationen wie zum Beispiel den Grundrechenarten.

2.2 Bilddaten und -größe

Eines der wichtigsten Probleme bei der Bildanalyse von Computer-Tomogrammen stellt die extreme Größe der Bilddaten dar. Diese Eigenschaft ist nicht nur störend hinsichtlich der Bearbeitungsgeschwindigkeit sondern insbesondere auch in Bezug auf den zur Verfügung stehenden Arbeitsspeicher. Die Datengröße kann dabei relativ schnell Werte erreichen, welche von 32-Bit Computersystemen nicht, oder ggf. nur in Teilen, bearbeitet werden können.

Um dieses Problem zu verdeutlichen, soll an dieser Stelle eine Beispielrechnung durchgeführt werden: Die aktuelle CCD-Kamera am Messplatz kann Fotografien bis zu einer maximalen Auflösung von 2048x2048 Pixel erstellen. Pro Pixel werden die Farbinformationen als Integer-Werte mit 16 Bit Farbtiefe (Graustufenwerte) gespeichert. Führt man eine Tomographie mit einer Auflösung von 2048x1024 Pixeln und 900 Drehungen bzw. Projektionen durch, so erhält

man Rohdaten in der Größenordnung von 3600 MByte. Bei der Rekonstruktion werden die 2D-Radiogramme in 2D-Sinogramme[1] überführt. Die „Gefilterte Rückprojetion“ ([9]) wandelt diese mittels Fourier-Transformation und unter Einberechnung des Drehwinkels in die einzelnen Schichten des 3D-Tomogramms um. Um eine möglichst hohe Genauigkeit zu bekommen, wird der Datensatz dabei auf 32 Bit Floatwerte erweitert. Dadurch ergibt sich eine Größe des Tomogramms von $2048 * 1024 * 1024 * 32Bit = 8GByte$.

Bevor es jedoch zur Binarisierung kommt, werden die Bilddaten durch Festlegen einer „Region of Intrest“ und das Umwandeln in Charakter-Werte (vgl. 1.3.4) reduziert. So bleibt, betrachtet man das Beispiel weiter, eine Größe von ca. 1-2 GByte für die Bilddaten übrig. Dies erscheint zunächst ein überschaubarer Wert, da bereits mit 32-Bit-Prozessoren ein maximaler Speicher von 4 GByte angesprochen werden kann. Zieht man jedoch mit in Betracht, dass für Bearbeitungsvorgänge zusätzlich eine oder mehrere Kopien der Daten vorhanden sein müssen, oder durch rekursive Aufrufe der Stackspeicher erheblich anwachsen kann, so erkennt man schnell die Speicherproblematik. Zusätzlich stehen je nach Betriebssystem pro Prozess lediglich maximal zirka 2 GByte Speicher zur Verfügung. Alternativ besteht die Möglichkeit einzelne Teile der Bilddaten zu laden, um dass Speicherproblem zu umgehen. Dies hat aber auf der einen Seite eine Verlangsamung der Berechnung zur Folge oder kann unter Umständen nicht bei allen 3D-Algorithmen angewandt werden.

Der Speichermangel tritt bei 64-Bit-Prozessoren in den Hintergrund, da hier theoretisch bis zu $1,7 * 10^{10}$ GByte adressierbar sind. Allerdings wird hier der Speicher durch physikalische Beschränkungen wie Anzahl der Speicherbänke, die maximale Größe eines Speicherriegels und des Auslagerungsspeichers begrenzt.

Ein weiteres großes Problem bei der Verarbeitung liegt in der Beschaffenheit der Tomogramme. Die darin abgebildeten Strukturen eines Objektes besitzen in fast allen Fällen eine extrem breite, nicht-homogene Graustufenverteilung. Dies bedeutet nicht nur, dass die zur Darstellung einer Struktur gehörenden Pixel nicht denselben oder annähernd denselben Graustufenwert ausweisen. Zusätzlich ist keine klare Verteilung der Graustufenwerte – von zum Beispiel dunklen Bereichen in der Mitte hin zu hellen Bereichen am Rand oder andersherum – erkennbar.

Diese negativen Erscheinungen in Bezug auf die Segmentierung haben verschiedene Ursprünge. Das erste Problem liegt in den zu untersuchenden Proben. Die einzelnen Strukturen, auch Phasen genannt, unterliegen Dichteschwankungen, so dass der Synchrotronstrahl innerhalb einer Phase unterschiedlich stark abgeschwächt wird. Weitere Probleme sind auf das abbildende System zurückzuführen. Auf der einen Seite sind die Röntgenstrahlen nicht komplett parallel. Zusätzlich kommt es durch Beugungseffekte und Streuung am Rand der Materialgrenzen zu nicht erwünschten Änderungen der Flugbahn von Photonen. Ein Großteil der Fehler entsteht des Weiteren auf Grund der Kameraoptik und bei der Rekonstruktion. Beim Belichten der einzelnen Photozellen der Kamera kommt es zu einem Übersprechen auf benachbarte Zellen, so dass das Bild unscharf werden kann. Bei der Rekonstruktion entstehen sogenannte Diskretisierungsprobleme, da die Pixel- bzw. Voxelgröße auf einen endlich kleinen Wert begrenzt ist.

Während die Datengröße besondere Maßnahmen bei der Implementierung verlangt, müssen die Segmentierungsalgorithmen entsprechend der ungünstigen Bildeigenschaften im Sinne der

[1] Im Sinogramm werden alle zu einer tomographischen Schnittebene gehörenden Daten aus den Radiogrammen zusammengefasst.

Darstellung angepasst werden. Auf Grund der zuletzt diskutierten Problematik muss ein entsprechend komplizierter Binarisierungsalgorithmus verwendet werden (vgl 3.1). Inbesondere die Überlegungen hinsichtlich einer dynamischen Segmentierung im Kapitel 4 sind stark von der Darstellung der Strukturen abhängig.

2.3 Anwendungsbeispiel: Metallschaum

Im Hahn-Meitner-Institut Berlin in der Abteilung Werkstoffe (SF3) beschäftigt sich eine Arbeitsgruppe mit dem Schäumen von Metallen. Das Ziel ist dabei – ähnlich wie bereits bei Kunststoffen oder Glas – poröse Strukturen aus Metallen zu erstellen.

Die Herstellung der Metallschäume kann auf unterschiedliche Weise vollzogen werden. Eine Methode, die am HMI näher untersucht wird, verwendet zur Schäumung des entsprechenden Metalls (Aluminium, Zink) ein Treibmittel (Titanhydrid) in Pulverform. Die Ausgangsmaterialien werden gemischt und zu einem Vormaterial (Rohling oder Halbzeug) zusammengepresst. Danach erfolgt die Erhitzung des Rohlings auf eine Temperatur, die um den Schmelzpunkt des verwendeten Metalls liegt. Bei dieser Temperatur setzt das Treibmittel Gas frei, so dass sich Blasen bilden, welche die Poren formen. Hat das Material seine endgültige Form erreicht, wird der Schäumungsvorgang durch schnelles Abkühlen beendet. Durch das zusätzliche Auftragen einer nicht-schäumenden Deckschicht vor dem Schäumen entsteht eine sogenannte Sandwichstruktur.

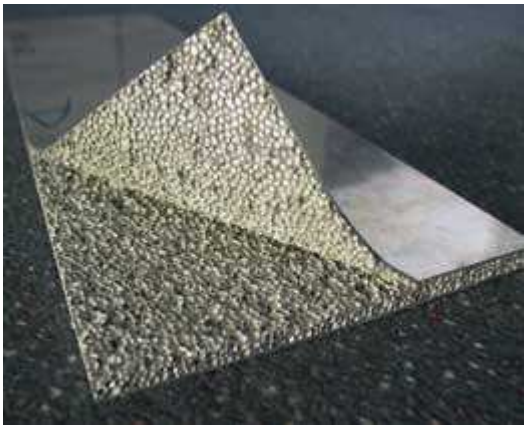


Abbildung 2.1: Abbildung einer Sandwichstruktur: Zur Erstellung der Sandwichstruktur wird auf das zu schäumende Material vor dem Erhitzen ein nicht-schäumbares Metall aufgewalzt. (Quelle: Fraunhofer-Institut für Fertigungstechnik und Angewandte Materialforschung)

Der Vorteil des Metallschaums liegt in seiner leichten und dabei auch festen Beschaffenheit. Auf der einen Seite zeichnet er sich durch ein sehr geringes Gewicht aus, kann jedoch auf der anderen Seite auf Grund seiner extrem hohen Stabilität großen Belastungen ausgesetzt werden. Zusätzlich besitzt er hohe mechanische und akustische Dämmeigenschaften und ist nicht brennbar. Daraus ergeben sich verschiedene Anwendungsgebiete wie Wärmedämmung, Brandschutz oder der Einsatz zur Energieabsorption. Verwendet werden können Metallschäume beim Flugzeugbau und in der Automobilindustrie. [1], [13]

Kapitel 3

Parametrisierte Segmentierung

Die vorliegende Diplomarbeit beschäftigt sich mit dem dritten großen Schritt bei der Datenauswertung (vgl. 1.3.4 Vom Tomogramm zum 3D-Bildanalyse) – dem Segmentieren bzw. Binarisieren. Dabei werden verschiedene Strukturen, zum Beispiel verschiedene Materialen eines Werkstoffes, aus dem Datensatz zur weiteren Analyse gefiltert. Eine detailierte Beschreibung der Binarisierung, wie sie in der Bibliothek umgesetzt wurde, erfolgt im Abschnitt 3.1. Überlegungen hinsichtlich einer dynamischen Segmentierung werden im Folgen Kapitel diskutiert.

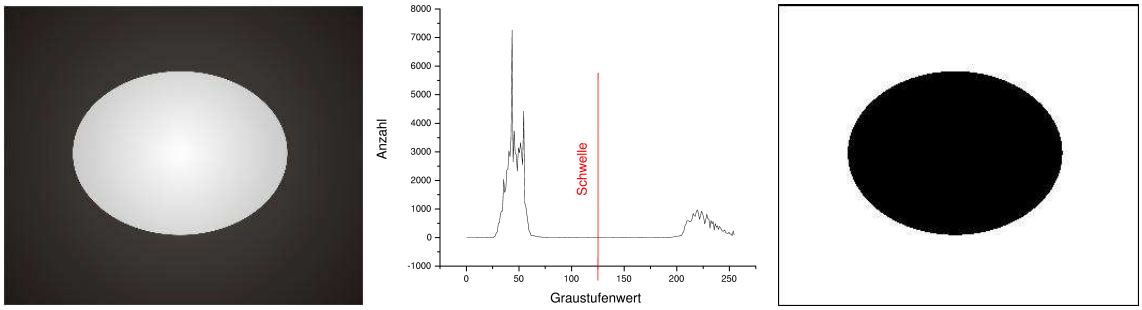
Nachdem die genaue Vorgehensweise der Binarisierung für den Anwendungsfall des Diplomthemas erklärt wurde, folgt der praktische Teil der Diplomarbeit. Dazu sollen zunächst verschiedene Algorithmen und Verfahren zur bestmöglichen Berechnung diskutiert werden. Danach wird die Implementierung des entsprechenden Programms beschrieben.

3.1 Binarisierung mittels Regionenwachstum und Schwellwerthysterese

Die Binarisierung ist, einfach gesprochen, das Umwandeln von Graustufenbildern in Schwarz-Weiß-Bilder. Mit diesem Vorgang können durch zum Beispiel Festlegen von verschiedenen Grenz- oder Schwellwerten innerhalb des Graustufenbereiches unterschiedliche Strukturen aus dem Bild gefiltert werden.

Die Graustufenbilder werden im Allgemeinen als 8 oder 16-Bit Werte pro Pixel gespeichert. Dabei bedeutet eine Vergrößerung der Speichertiefe eine Erhöhung der Graustufenanzahl und somit eine detailiertere Darstellung. Die Speicherung von Werten über 32-Bit pro Pixel wird jedoch in den wenigsten Fällen verwendet, da diese Detailgenauigkeit vom Menschen entweder gar nicht oder nur noch minimal wahrgenommen wird.

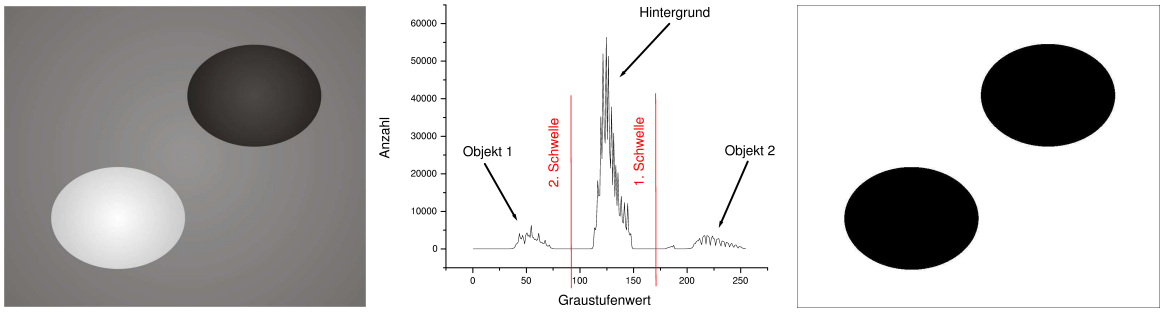
Die einfachste Möglichkeit einer Binarisierung ist die mit einem Grenzwert. Dieser Wert wird so aus dem Bereich des Graustufenbildes festgelegt, dass eine gewünschte Struktur aus dem Bild separiert wird. Ein einfaches Beispiel ist das Trennen eines nahezu einfarbigen Hintergrundes von den Vordergrundobjekten (3.1(a)). Dazu wird der Schwellwert ausgewählt, der zwischen den Grausstufenwert des Hintergrunds und denen der Vordergrundobjekte liegt. Dieser lässt sich am einfachsten mittels eines Histogramms bestimmen (3.1(b)). Je nach Helligkeit des Hintergrundes werden dann alle Pixel die unterhalb (dunkler Hintergrund) bzw.



(a) Original (b) Histogramm mit Schwellwert (c) Binarisiertes Bild

Abbildung 3.1: Binarisierung mit einem Grenzwert: Die Binarisierungsschwelle wird so gewählt, dass sie zwischen den Graustufenwerten des Hinter- und Vordergrunds liegt (b). Im dargestellten Beispiel wurden dem Hintergrund und dem Vordergrundobjekt ein Farbverlauf gegeben, damit diese im Histogramm besser sichtbar sind.

die oberhalb (heller Hintergrund) des Schwellwertes liegen mit Eins und alle anderen mit Null markiert. Als Ergebnis werden alle Vordergrundobjekte weiß und der Hintergrund schwarz dargestellt (Abbildung 3.1(c)).



(a) Original (b) Histogramm mit Schwellwerten (c) Binarisiertes Bild

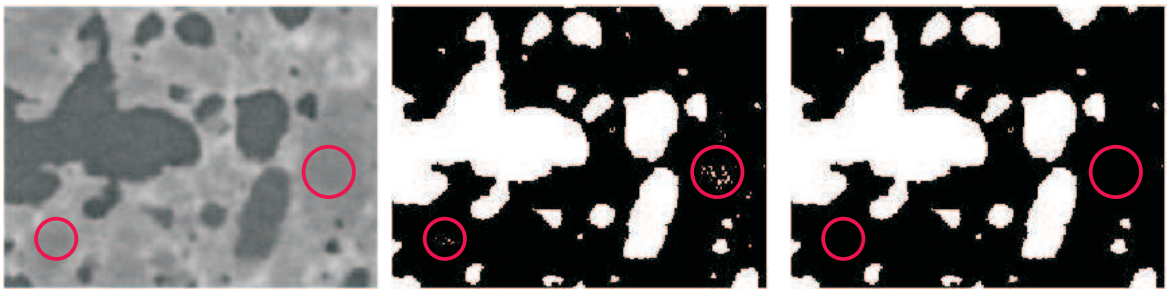
Abbildung 3.2: Binarisierung mit zwei Grenzwerten: Die Binarisierungsschwellen werden so gewählt, dass diese zwischen den Graustufenwerten des Hintergrunds und denen der Objekte liegen liegt (b). Im dargestellten Beispiel wurden dem Hintergrund und dem Vordergrundobjekt wie in Abbildung 3.1 ein Farbverlauf gegeben, damit diese im Histogramm besser sichtbar sind.

Bei dieser Vorgehensweise wird natürlich vorausgesetzt, dass es keine Vordergrundobjekte gibt, von denen Graustufenwerte oberhalb (dunkler Hintergrund) bzw. unterhalb (heller Hintergrund) der Hintergrundwerte liegen. Ist dies der Fall, gibt es zwei Möglichkeiten, um den Hintergrund vom Vordergrund zu separieren. Im ersten Fall wird die Binarisierung mit einem Grenzwert zweimal durchgeführt. Dabei werden zunächst alle Vordergrundobjekte herausgefiltert, die von ihren Graustufenwerten unterhalb der des Hintergrundes liegen. Im nächsten Schritt wird das Ursprungsbild erneut mit entsprechendem Schwellwert zur Separation der Vordergrundobjekte mit Graustufenwerten oberhalb der des Hintergrundes binarisiert. Um zum Schluss ein korrektes Bild zu erhalten, müssen die beiden Teilbilder über eine Oder-

Verknüpfung zusammengefügt werden.

Eine weitaus einfachere Variante zur Lösung dieses Problems stellt die Binarisierung mit zwei Schwellwerten dar. Bei dieser Methode werden ein oberer und ein unterer Grenzwert so festgelegt, dass diese den Graustufenbereich des Hintergrundes einschließen. Bei dieser als auch bei der zuvor beschriebenen Möglichkeit gilt als Voraussetzung, dass innerhalb des Graustufenbereiches des Hintergrundes keine Vordergrundobjekte und umgekehrt liegen dürfen. Sollte dies der Fall sein, muss die Binarisierung mit zwei Schwellwerten doppelt oder eine Binarisierung mit vier Schwellwerten durchgeführt werden. Dies setzt sich entsprechend bei weiteren Vordergrundobjekten, deren Graustufenwerte innerhalb der des Hintergrundes liegen und deren Graustufenmenge disjunkt zu den vorhergehenden Objekten sind, fort. Da die Beschaffenheit der vorliegenden Daten (2.3) in den meisten Fällen Strukturen enthält, deren Graustufenbereiche nicht von anderen gekreuzt werden, wird bei den folgenden Überlegungen auf das Modell mit zwei Schwellwerten aufgebaut.

Bei der Synchrotron-Tomographie (1.3.3) und der dazugehörenden Rekonstruktion (1.3.4) entsteht ein 3D-Datensatz – das Tomogramm. Dieses liegt in 8-Bit Graustufenwerten vor. Bei der Synchrotron-Tomographie kommt es durch das Messverfahren zu systematischen Fehlern. Diese äußern sich in (Ring-)Artefakten oder Rauschen. Bei der Rekonstruktion wird durch das Anwenden verschiedener Filter ein Großteil dieser Fehler korrigiert. Trotzdem führt dieses Verfahren zu einer örtlich inhomogenen Graustufenverteilung.

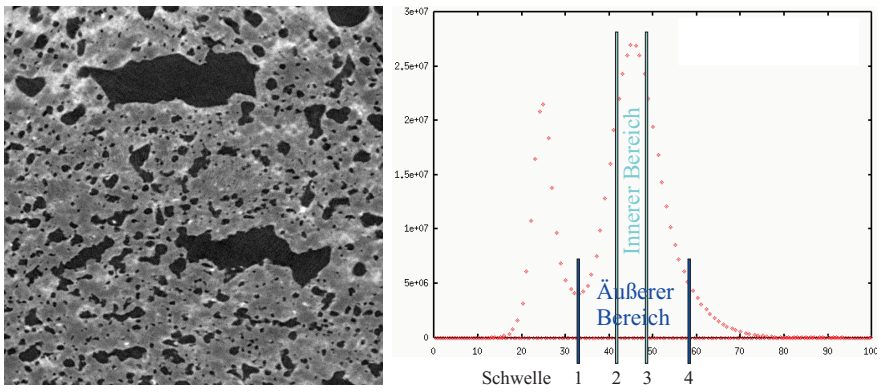


(a) Original (b) Einfacher Grenzwert (c) Doppelter Grenzwert

Abbildung 3.3: Binarisierung der Poren eines Metallschaums (Material: AlSi6Cu10) mit unterschiedlicher Anzahl von Grenzwerten: Bei Verwendung einer inneren und äußeren Grenze (c) werden Pixel mit Graustufenwerte, die auf Grund von Messfehlern falsche Werte aufweisen (b), ignoriert. (Siehe markierte Bereiche)

Versucht man das Tomogramm auf die oben beschriebene Art zu separieren, so erhält man ein Ergebnis, welches ein starkes Binarisierungsrauschen aufweist. Als Beispiel soll hier ein binarisiertes Tomogramm von einer Metallschaumstruktur (2.3) dienen, bei der die Poren separiert wurden. Die im Bild 3.3(b) dargestellte, binarisierte Porenstruktur des Metallschaums enthält viele einzelne weiße Pixel (markierter Bereich), welche nicht zu einer Pore gehören. Der Ursprung dieser Fehler sind einzelne Voxel, deren Graustufenwerte auf Grund von Messfehlern einen zu niedrigen Wert aufweisen und somit unterhalb des Grenzwertes liegen. Beim Betrachten der Schicht kann das menschliche Auge sofort die eigentlichen Strukturen erkennen, in dem es die Pixelfehler ignoriert. Kritisch wird diese Binarisierung jedoch bei der Weiterverarbeitung durch den Computer. Die einzelnen weißen Pixel stellen in ihrer Gesamtanzahl einen

so großen Wert da, dass es bei weiteren Auswertungen zu starken Abweichungen kommen kann.



(a) Original (b) Histgramm mit Schwellerten

Abbildung 3.4: Binarisierung mit Schwellhysterese: Um ein minimales Binarisierungsrauschen zu erreichen, werden die Schwellwerte verdoppelt. Dadurch kann in einen inneren und einen äußeren Bereich (siehe (b)) unterschieden werden. Bei der Binarisierung werden alle Pixel/Voxel die innerhalb des inneren Schwellwertes liegen als sicherer Bestandteil des zu separierenden Volumens – in diesem Fall die Matrix in (a) – betrachtet. Von diesem ausgehend wird so lange in einer vordefinierten Nachbarschaft gewachsen, wie Pixel/Voxel mit Graustufenwerten innerhalb des äußeren Bereichs liegend gefunden werden.

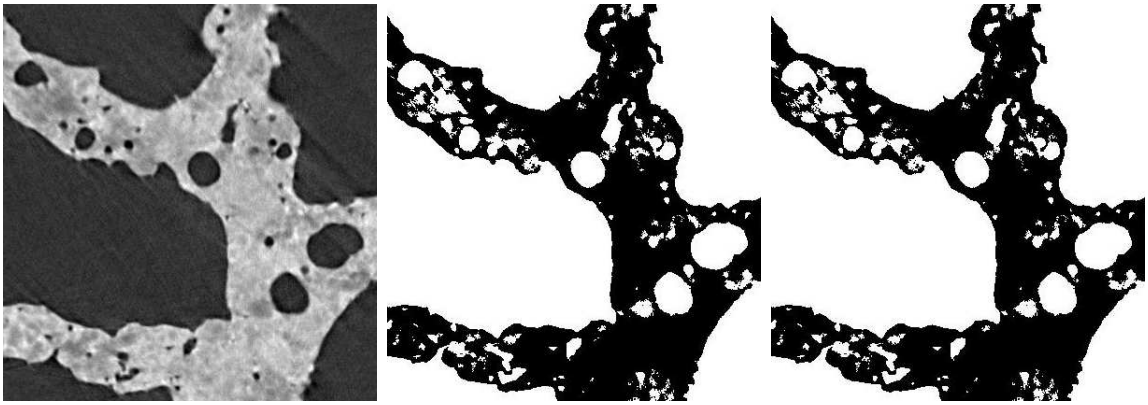
Die Einführung einer Schwellwerthysterese (vgl. 4.4 Canny-Operator) soll dieses Problem minimieren. Dazu wird die Anzahl der Grenzwerte, von zwei auf vier, verdoppelt (siehe Abbildung 3.4). Das Ziel dieser Änderung liegt im Finden bzw. Ignorieren von Voxeln, deren Graustufenwerte durch Messfehler in benachbarte Strukturen[1] abweichen. Durch die Verdoppelung der Zahl der Grenzwerte kann nun zwischen einem inneren und einem äußeren Bereich in der Graustufenverteilung unterschieden werden. Dabei werden alle Voxel mit Graustufenwerten, die innerhalb des inneren Bereiches liegen, als sicherer Bestandteil des Volumens der zu separierenden Struktur betrachtet. Anschließend werden sukzessive alle Nachbarn untersucht, ob diese innerhalb der äußeren Grenzen liegen. Ist dies der Fall, verfügen diese über eine Verbindung zur Keimzelle (sicheres Volumen) und können somit auch der zu separierenden Struktur zugeordnet werden.

Der Binarisierungsalgorithmus ändert sich dementsprechend dahin, dass im Tomogramm Voxel gesucht werden, deren Graustufenwert zwischen den inneren Grenzen liegt. Nach dem Finden eines solchen Voxels (Keimzelle), wird um diesen solange gewachsen, wie benachbarte Voxel gefunden werden, deren Graustufenwert zwischen den beiden äußeren Grenzen liegt (Ergebnis s. Bild 3.3(c)). Diese Vorgehensweise vergrößert den Aufwand für die Binarisierung zwar erheblich, liefert jedoch ein weit besseres Ergebnis.

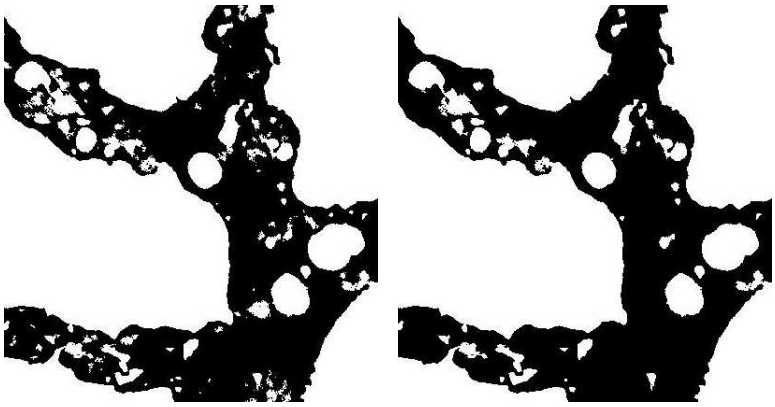
Bei der soeben beschriebenen Erweiterung spielt die Definition der Nachbarschaft eine wichtige Rolle. Im 3D-Raum wird diese im Allgemeinen mit 26 Nachbarn pro Voxel definiert. Eine Nachbarschaft mit 18 oder mit 6 Nachbarn ist jedoch auch denkbar. Bei ersterer werden

[1] Benachbarte Strukturen in Hinblick auf die Graustufenwerte und nicht des Ortes

die Ecken des „Nachbarschaftswürfel“ wegelassen. Die zweite Definition enthält lediglich die Hauptachsen der einzelnen Ebenen. Neben einer 3D-Nachbarschaft kann auch eine Nachbarschaft im Zweidimensionalen gewählt werden. Da es sich aber um einen 3D-Datensatzsatz handelt, führt dies zu extrem schlechten Ergebnissen (s. Bild 3.5(e)). Bei einer 3D-Nachbarschaft, die nicht alle 26 Nachbarn mit einbezieht (3.5(c) und 3.5(d)), führt die Binarisierung immer noch zu relativ guten Ergebnissen. Diese sind u.a. von der Morphologie der Struktur abhängig. Vorteilig bei der Verringerung der Nachbarn ist die schnellere Ausführungszeit des Algorithmus.



(a) Original (b) 26 Nachbarn (3D) (c) 18 Nachbarn (3D)



(d) 6 Nachbarn (3D) (e) 4 Nachbarn (3D)

Abbildung 3.5: Binarisierung mit unterschiedlicher Nachbarschaftsdefinition: Beim Verringern der Nachbarn im 3D-Raum können weiterhin gute Ergebnisse erzielt werden. Zusätzlich steigt die Verarbeitungsgeschwindigkeit. Beim Definieren einer Nachbarschaft, welche nur Nachbarn in einer Ebene (2D) enthält, verschlechtert sich die Ausgabe für ein 3D-Bild rapide. Daraus folgt, dass die binären Tomogramme nicht mehr weiter genutzt werden können. (Zur besseren optischen Hervorhebung der Unterschiede wurden bei der Binarisierung der Porenstruktur größere Schwellwerte für die äußeren Grenzen verwendet.

3.2 Wachstumsalgorithmen

In diesem Abschnitt werden Algorithmen zur Umsetzung der Binarisierung beschrieben. Da sich das Auffinden der Keimzellen (vgl. 3.1) auf ein lineares Durchsuchen des 3D-Datensatzes (Punktoperation) beschränkt, soll darauf nicht näher eingegangen werden. Ein interessantes Thema ist dagegen das Finden von Algorithmen zum Wachsen um die Keimzelle und das damit verbundene Füllen der zu separierenden Struktur. Da es sich hier im Gegensatz zum Finden der Keimzellen um lokale Operatoren, welche in Abhängigkeit einer vordefinierten Nachbarschaft arbeiten, handelt, müssen komplexere Überlegungen hinsichtlich der Bearbeitungsgeschwindigkeit und der Speichernutzung auf Grund der Größe der Tomogramme durchgeführt werden.

Im Folgenden sollen zwei Algorithmen – „Boundary Fill“ und „Fill“ – vorgestellt werden. Im Anschluss erfolgen einige Überlegungen zur Parallelisierungsmöglichkeit der beiden Verfahren. Bei der endgültigen Implementierung der Bibliothek wird dann auf den effektiveren der beiden Algorithmen mit Blick auf Speichernutzung und Geschwindigkeit zurückgegriffen.

3.2.1 „Boundary Fill“

Einer der bekanntesten Algorithmen zum Füllen von Objekten ist der sogenannte „Boundary Fill“-Algorithmus. Dieser wird im Allgemeinen dazu verwendet, Flächen mit einer bestimmten Farbe zu füllen. Als Voraussetzung dafür muss ein Punkt gegeben sein, der innerhalb der zu füllenden Fläche liegt. Zusätzlich müssen die Grenzen des Objektes durch eine weitere Farbe gekennzeichnet sein.

Der Ablauf des „Boundary Fill“-Algorithmus sieht wie folgt aus: Ausgehend von dem Pixel in dem zu füllenden Bereich werden alle Nachbarn untersucht, ob diese noch innerhalb der Fläche liegen. Ist dies der Fall, werden diese mit der zu füllenden Farbe markiert und danach die Nachbarn des neuen Pixels untersucht. Diese Prozedur wird solange wiederholt, bis der Rand der Fläche erreicht ist.

Der „Boundary Fill“-Algorithmus kann auf Grund seiner Vorgehensweise am einfachsten rekursiv implementiert werden. Das Programmlisting für eine Nachbarschaft von vier Pixeln ist in Listing 3.1[2] dargestellt.

Listing 3.1: Boundary Fill-Algorithmus (Pseudocode)

```
1  BoundaryFill(x, y, Füllfarbe, Grenzfarbe)
2  BEGIN
3          IF ( (getPixel(x,y)<>Fuellfarbe) AND (getPixel(x,y)<>Grenzfarbe) )
4          BEGIN
5                  BoundaryFill(x+1,y  ,Füllfarbe, Grenzfarbe);
6                  BoundaryFill(x  ,y+1,Füllfarbe, Grenzfarbe);
7                  BoundaryFill(x-1,y  ,Füllfarbe, Grenzfarbe);
8                  BoundaryFill(x  ,y-1,Füllfarbe, Grenzfarbe);
9          END;
10 END;
```

Betrachtet man die Binarisierung eines Tomogramms, so sind nur kleine Änderungen am Algorithmus notwendig, um diesen zur Binarisierung nutzen zu können. Der erste Unterschied

[2]Der hier dargestellte Algorithmus funktioniert nur, wenn sich in der Füllfläche kein Kreis von Pixeln mit der zu füllenden Farbe befindet.

zu einem herkömmlichen Bild liegt in dem Vorhandensein eines 3D-Datensatzes. Daraus ergibt sich eine neue Nachbarschaftsbeziehung. Diese muss auf den 3D-Raum ausgeweitet werden, so dass sich die Anzahl der rekursiven Aufrufe vergrößert. Des Weiteren wird der Rand der zu füllenden Struktur nicht durch einen Farbwert bzw. Graustufenwert definiert, sondern ergibt sich aus dem vorgegebenen Schwellwert.

Ein weitaus größeres Problem im 3D-Raum tritt durch die große Anzahl der auftretenden rekursiven Funktionsaufrufe und dem damit verbundenen Speicherverbrauch auf. Lösungen zur Umgehung dieses Problems werden in Abschnitt 3.3.2 diskutiert.

3.2.2 „Fill“

Der „Fill“-Algorithmus ist eine weiterführende morphologische Transformation zum Füllen von Strukturen. Dieser greift auf eine morphologische Basistransformation – die Dilatation – zurück.

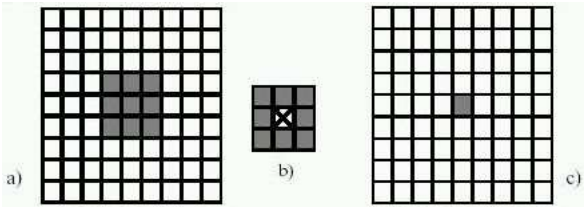


Abbildung 3.6: Erosion [11]: Das Strukturelement (b) wird über jeden Pixel des Ursprungsbild (a) gelegt. Ein Pixel unter dem Maskenmittelpunkt bleibt erhalten, wenn alle Maskeneinträge mit allen korrespondierenden Pixeln des Ursprungbildes übereinstimmen (c).

Die Dilatation sowie die verwandte Operation Erosion verwenden ein sogenanntes Strukturobjekt, welches zur Separation eines bestimmten Pixelmusters dient. Beim Auffinden des entsprechenden Musters werden bei der Erosion Pixel vom Rand des Objektes abgetragen bzw. bei der Dilatation hinzugefügt. Sowohl Erosion als auch Dilatation sind Transformationen, die auf Binärbilder angewandt werden. Die Bedingungen zur Änderung eines Bildes sind, dass bei der Erosion sämtliche Einträge des Strukturelements mit den entsprechenden Pixeln im Originalbild übereinstimmen müssen. Bei der Dilatation verhält es sich umgekehrt: Wenn der Maskenmittelpunkt mit dem Ursprungsbild übereinstimmt, wird dilatiert. Sind die Bedingungen erfüllt, wird der Pixel unter dem Mittelpunkt der Maske bei der Erosion beibehalten. Bei der Dilatation werden entsprechend die im Strukturelement um den Mittelpunkt beschriebenen Pixel dem Originalbild hinzugefügt. [4]

Die mengentheoretische Aussage sieht wie folgt aus:

Erosion: $B \oplus S = \{x | B_X \subseteq S\}$

Dilatation: $B \ominus S = \{x | \overline{B}_X \cap S \neq 0\}$

Die Menge S und B steht für alle grau markierten Pixel, wobei S das Strukturelement und B das Bild repräsentiert und x ∈ S, B gilt. Anschauliche Bespiele sind in Bild 3.6 und 3.7 dargestellt. [11]

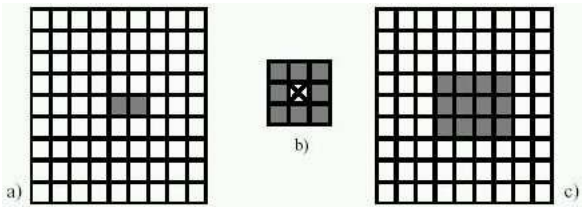


Abbildung 3.7: Dilatation [11]: Bei der Dilatation wird wie bei der Erosion das Strukturelement (b) Punkt für Punkt über das Ursprungsbild (a) gelegt. Entspricht der Maskenmittelpunkt dem Pixel im Originalbild, so werden alle restlichen Punkte der Maske diesem hinzugefügt (c).

Der „Fill"-Algorithmus beinhaltet eine iterative Vorgehensweise zum Ausfüllen von Strukturen. Ausgehend von einem Punkt (Füllmenge F_0, der in der Struktur liegt (Abbildung 3.8(c)), wird eine Dilatation auf diesem durchgeführt (3.8(d)). Danach wird das Ergebnis mit der Komplementärmenge des Bildes geschnitten (3.8(e)). Im nächsten Schritt wird die Dilatation auf die neu entstandene Ergebnismenge (Füllmenge F_n) angewandt. Im Anschluss erfolgt erneut die Bildung der Schnittmenge mit der Komplementärmenge des Bildes (3.8(f)). Dies wird solange fortgesetzt, bis die Füllmenge F_n sich nicht mehr vergrößert (3.8(h)). [4]

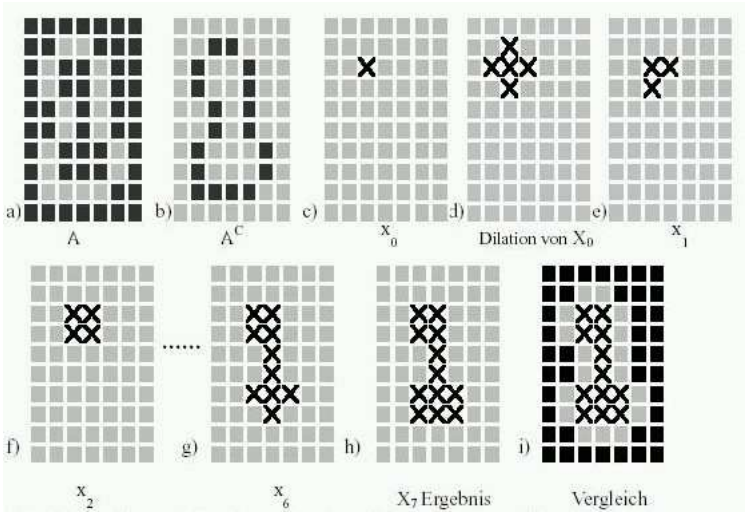


Abbildung 3.8: Fill: Der Fill-Algorithmus beinhaltet die iterative Vereinigung der Dilatation der Füllmenge mit der Komplementärmenge des Bildes. Der Vorgang endet, wenn sich die Füllmenge nach Anwendung eines Iterationsschrittes nicht mehr vergrößert. [11]

In der Mengentheorie ergibt dies folgende Darstellung [11]:

$$\textbf{Fill:} \quad F_n = (F_{n-1} \ominus S) \cap \overline{B}, \qquad n = 1, 2, 3, ...$$

$$F = Füllmenge, \quad S = Strukturelement, \quad B = Bild$$

Damit der „Fill"-Algorithmus beim Binarisieren eingesetzt werden kann, müssen auch hier

einige Anpassungen vorgenommen werden. Das Strukturelement wird an die festgelegte Nachbarschaft im 3D-Raum angepasst. Die Bedingung zur Durchführung der Dilatation wird an die äußeren Schwellwerte (vgl. Abschnitt 3.1) für die Binarisierung geknüpft. Das heißt, sobald ein Nachbarvoxel des zu untersuchenden Raumpunktes innerhalb der äußeren Grenzwerte liegt, wird die Dilatation ausgeführt. Die nachfolgende Vereinigung mit der Komplementärmenge des Bildes wird so geändert, dass nur die Voxel weiter in der Füllmenge verbleiben, deren Graustufenwerte auch innerhalb der äußeren Grenzen liegen.

3.3 Implementierung

3.3.1 Allgemeines

3.3.1.1 Betriebssystem

Die Programmierung der Bibliothek für die Binarisierung erfolgt für Unix-Betriebsysteme. Dies liegt an dem weit verbreiteten Einsatz von Unix-Systemen auf Hochleistungsrechnern. Da die Anzahl der Unix-Derivate und deren Aufteilung in verschiedene Distributionen eine extrem große Anzahl von unterschiedlichen Betriebssystemen darstellt, kann die reibungslose Erstellung einer Software für alle Unix-Varianten nicht garantiert werden. Aus diesem Grund wurde bei der Programmierung insbesondere auf die Lauffähigkeit der im HMI zur Verfügung stehenden Unix-Systemen geachtet. Diese sind S.u.S.E. Linux 9.0/9.1, Solaris 9 und Tru64 Unix 5.1B (vgl. Anhang C).

Zum Testen der Anforderungen einer 64-Bit Software steht somit das Tru64 Unix zur Verfügung. Da die Rechner der Arbeitsgruppe „Synchrotron-Tomographie“ mit einer 32-Bit-Architektur mit einem 32-Bit Linux von S.u.S.E. laufen, musste der Einsatz der Bibliothek auch unter diesen Voraussetzungen möglich sein (vgl. 2.1). Der Einsatz eines 64-Bit-Rechners mit S.u.S.E. 9.1 64-Bit ist für dieses Jahr geplant. Der entsprechende Rechner stand jedoch während der Diplomzeit noch nicht zur Verfügung.

3.3.1.2 Programmiersprache

Die zur Umsetzung gewählte Programmiersprache ist C. Die Entscheidung dafür hat verschiedene Gründe: Wie bereits im Abschnitt zuvor beschrieben, ist der Einsatz der Bibliothek auf einem Unix-System geplant. Die quasi „Standardprogrammiersprache“ für Unix ist C bzw. C++. Dies liegt u.a. historisch begründet, da das Betriebssystem Unix auch in der Programmiersprache C geschrieben wurde.

Ein weiterer wichtiger Grund ist das Vorhandensein weiterer Bibliotheken zur Bildanalyse, welche auch in C geschrieben sind. Im Bereich der Grundlagenforschung, wie sie am HMI und auch in der Abteilung Materialforschung betrieben wird, müssen oft neue Programme und Algorithmen auf Grund neuer Entwicklungen und Erkenntnisse implementiert werden. Um diese nach Möglichkeit zu vereinheitlichen, hilft das Programmieren in einer festgelegten Sprache. Des Weiteren soll in naher Zukunft ein Projekt gestartet werden, welches das Erstellen einer Oberfläche zur anwenderfreundlichen Verarbeitung vom Radiogramm bis hin zur Bildanalyse enthält. Dazu sollen sämtliche bereits vorhandenen Bibliotheken verwendet werden. Mit

Hinsicht auf dieses Projekt liegt es nahe, die Programmiersprache für alle Teilprojekte bzw. -programme zu vereinheitlichen.

Als letzter Grund soll die Beschaffenheit von C dienen. C ist eine Middle-Class Sprache. Sie vereint die Vorteile eine hardwarenahen Programmierung, wie zum Beispiel mit Assembler, und die einer High-Level-Sprache wie Pascal oder Basic. Eine Middle-Class-Sprache erlaubt auf der einen Seite die Manipulation von Bits, Bytes und Adressen, bietet auf der anderen Seite aber auch den Umgang mit Datentypen wie in High-Level-Sprachen. Erstere Eigenschaft ist besonders wichtig für die Verwaltung der Tomogramme im Arbeitsspeicher. Durch dynamische Speicherreservierung sowie Verwendung und Manipulation von Zeigern können einzelne Werte relativ einfach bestimmt und verändert werden. Der letzte große Vorteil von C ist die hohe Portabilität. Bei allen Testrechnern stand ein C-Compiler zur Verfügung. Der Quellcode ließ sich bis auf wenige kleine Anpassungen auf allen Systemen in Maschinen-Code übersetzen. Die Ausnahmen bezogen sich lediglich auf Bibliotheksfunktionen, welche nicht bei allen Betriebssystemen zur Verfügung standen.

3.3.1.3 Entwicklungsumgebung

Als Entwicklungsumgebung zur Erstellung der Bibliothek wurde das Programm KDevelop (`www.kdevelop.org`) in der Version 2.1.5 verwendet. KDevelop bietet eine graphische Oberfläche zum Erstellen und Debuggen von Programmen. Neben C und C++ Software können auch Projekte zur Oberflächenprogrammierung mittels KDE- oder QT-Bibliotheken oder bei neueren Versionen bzw. mit Erweiterungen auch Software mit anderen Sprachen erstellt werden.

Beim Anlegen eines neuen Projektes muss ein dem Programm entsprechendes Template gewählt werden. Bei der Erstellung der Bibliothek wurde ein einfaches „Hello-World“-Programm in der Programmiersprache C gewählt. Auch wenn dies zunächst nicht dem eigentlichen Ziel entspricht, hat diese Vorgehensweise folgenden Vorteil:

KDevelop erstellt nicht nur die entsprechende .c-Datei, sondern auch einen Ordner mit allen Dateien, welche für eine möglichst betriebssystemunabhängige Kompilierung (s. oben) notwendig sind. Dabei werden die GNU-Programme „autoconf“, „automake“ und „autoheader“ verwendet. „Autoconf“ erzeugt aus der Überprüfung der vorhandenen Quellcodedateien eine Datei „configure.scan“, welche verschiedene Anweisungen für das später generierte „configure“-Skript enthält. Aus der umbenannten Datei „configure.ac“ („configure.scan“) und dem Makefile („makefile.am“) für das Programm, welches durch KDevelop erstellt wird, erzeugen die oben genannten Programme die drei Dateien „configure“, „config.h.in“ und „makefile.in“. Mit diesen Dateien kann später der Benutzer bzw. Administrator das eigentliche Programm kompilieren. Dazu ruft er zunächst das „configure“-Skript auf, welches das Unix-Derivat auf seine Eigenschaften hin überprüft. Dieses erzeugt ein angepasstes Makefile, dass über den Aufruf von „make“ die entsprechenden Binaries erstellt. [3]

Um das Projekt den eigentlichen Anforderungen anzupassen, wurde als erstes der entsprechende Quellcode für das „Hello-World“-Programm gelöscht. Ein größeres Problem war zunächst die Erstellung eines Projektes zur Erzeugung einer Bibliothek, da dieses standardmäßig bei KDevelop nicht mit angeboten wird. Beim Durchsuchen der KDevelop-Dokumentation konnte jedoch festgestellt werden, dass Quellcodedateien von Unterordnern generell zu einer Bibliothek zusammengefasst werden. Aus diesem Grund wurde ein entsprechender

Ordner mit dem Namen der Bibliothek – a4ibool-[Version] – angelegt. Beim Anlegen der ersten Quellcodedateien erfolgt durch KDevelop eine Abfrage, ob eine statische oder dynamische Bibliothek erstellt werden soll. An dieser Stelle wurde eine statische Bibliothek gewählt, damit diese später bei der Erstellung von Programmen, welche die Bibliothek benutzen, mit dazugelinkt werden kann. Der Vorteil gegenüber einer dynamischen Bibliothek liegt darin, dass diese nicht erst in das entsprechende Betriebssystem installiert werden muss. Da der Quellcode mit großer Wahrscheinlichkeit nur selten oder gar nicht von mehreren oder doppelt ausgeführten Programmen auf einem Rechner gleichzeitig gebraucht wird, entfällt der speichersparende Vorteil einer dynamischen Bibliothek.

Auf Grund der Erstellung der Bibliothek im Unterordner des Projektes kann gleichzeitig im Stammordner des Projektes ein Programm zur Demonstration bzw. Anwendung der Bibliothek erstellt werden. Dies wurde zu Testzwecken als auch nach der Fertigstellung im Einsatz benutzt.

3.3.1.4 Datenverwaltung im Arbeitsspeicher

Die Verwaltung der Daten zur Laufzeit ist eines der wichtigsten Kriterien zur effektiven Speicherausnutzung. Zusätzlich kann durch verschiedene Techniken die Berechnungsgeschwindigkeit deutlich gesteigert werden.

Das zu bearbeitende Tomogramm ist in einer RAW-Datei gespeichert. Dies bedeutet, dass sämtliche Graustufenwerte für die einzelnen Voxel byteweise direkt hintereinander angeordnet sind. Dabei werden in der Datei die Werte jeweils zeilenweise aufsteigend in X-Richtung, in der entsprechenden Größe der Y-Dimension, gespeichert. Diese Anordnung entspricht einer XY-Ebene. Nachfolgend setzt sich der Aufbau für die Anzahl der XY-Ebenen (Z-Dimension) fort.

Die erste Möglichkeit der Speicherung des Tomogramms ist ein dreidimensionales Array. Nach Übergabe der Dimensionen durch den Benutzer an das Programm kann der entsprechende Speicher für das Array reserviert werden:

Listing 3.2: Dynamische Speicherreservierung für ein dreidimensionales Array

```
1  unsigned char ***pucTomogram;
2
3  if( !(pucTomogram = (unsigned char***)malloc(sizeof(char**)*uiDimX)) )
4    { perror("binarization"); return 1; }
5  for(x = 0; x < uiDimX; x++)
6  {
7    if( !(pucTomogram[x] = (unsigned char**)malloc(sizeof(char*)*uiDimY)) )
8      { perror("binarization"); return 1; }
9    for(y = 0; y < uiDimY; y++)
10     if( !(pucTomogram[x][y] = (unsigned char*)malloc(sizeof(char)*uiDimZ)) )
11       { perror("binarization"); return 1; }
12 }
```

Durch die verkettete Speicherreservierung ergibt sich die in Bild 3.9 abgebildete Anordnung im Speicher. Für die X-Dimension wird ein Array von Zeigern angelegt, die auf entsprechende Felder für die Y-Dimension zeigen. Diese wiederum adressieren Arrays für die Z-Dimension, welche dann die eigentlichen Graustufenwerte enthalten.

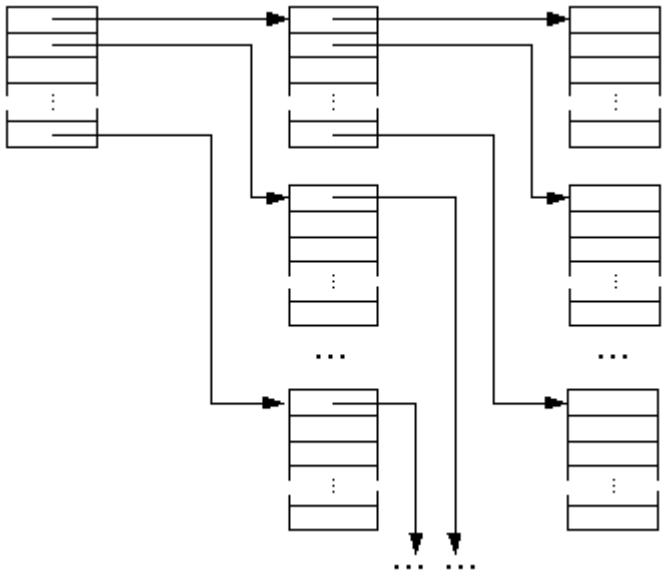


Abbildung 3.9: Verteilung eines dynamisch reserviertes, dreidimensionalen Arrays im Speicher: Die ersten beiden Derefferenzierungen zeigen auf Arrays, die wiederum auf weitere Felder verweisen. Erst die Arrays der dritten Ebene enthalten die eigentlichen Daten.

Ist der Speicher reserviert, können die einzelnen Graustufenwerte eingelesen werden. Der Nachteil, der in Listing 3.2 dargestellten Möglichkeit der Speicherreservierung ist, dass die einzelnen Werte lediglich byteweise geladen werden können. Bei der Größe der Dateien kann dies eine bis mehrere Minuten dauern. Eine Verbesserung in dieser Hinsicht ist die Vertauschung der X- und Z-Dimensionen bei der Speicherreservierung. Da nun die eigentlichen Graustufenwerte in Arrays der X-Dimension gespeichert werden, kann eine komplette Zeile (vgl. RAW-Dateiaufbau) aus der Eingabedatei gelesen werden. Durch das Benutzen einer Block-Read-Funktion erhöht sich die Einlesegeschwindigkeit erheblich. Dabei ist jedoch zu beachten, dass die X- und Z-Dimension des Arrays danach vertauscht sind.

Eine weitere Variante der Datenhaltung ist das Benutzen eines eindimensionalen Arrays. Die Vorteile hier liegen in einer einfacheren Speicherreservierung und in einer noch kürzeren Einlesezeit. Da hier die Graustufenwerte der einzelnen Voxel genau wie in der RAW-Datei angeordnet sind, können diese mittels einer einzigen Block-Read-Operation eingelesen werden.

Die Adressierung eines 3D-Datensatzes in einem eindimensionalen Feld ist zwar komplizierter, kann jedoch unter Umständen einen Geschwindigkeitsvorteil bringen. Um einen bestimmten Voxel zu adressieren, muss die entsprechende XY-Ebene durch Multiplikation mit den X- und Y-Dimensionen ermittelt werden. Danach werden die Offsets für die entsprechende Y-Zeile (X-Dimension*Y-Zeile) und den X-Wert (X-Dimension) hinzuaddiert.

Der Grund dafür liegt in den auszuführenden Adressierungsoperationen. Bei einem dreidimensionalen dynamischen Array muss eine dreifach indirekte Adressierung mit Offset im Hauptspeicher zum Ermitteln des eigentlichen Wertes durchgeführt werden. Bei einem eindimensionalen Array mit bekanntem Offset verringert sich der Aufwand dahin, dass lediglich

eine indirekte Adressierung mit Offset notwendig ist. Bei gut optimiertem Code befindet sich in beiden Fällen die Startadresse des Arrays bereits in einem Register, da ansonsten ein zusätzlicher Speicherzugriff notwendig ist.

Nachteilig bei der Datenhaltung in einem eindimensionalen Array ist die möglicherweise geringere Effektivität bei der Speicherausnutzung. Da bei einem dreidimensionalen Feld (s. Bild 3.9) die einzelnen Teilfelder in unterschiedlichen Speicherbereichen liegen können, ist es wahrscheinlich, dass beim Reservieren trotz möglicher starker bzw. ungünstig verteilter Speichernutzung noch freier Speicher durch das Betriebssystem zur Verfügung gestellt werden kann. Beim eindimensionalen Feld ist dies dann meist nicht mehr möglich, da hier alle Einträge direkt hintereinander liegen. Bei den Ausführungen in diesem Absatz ist zu beachten, dass hier nicht direkt vom Hauptspeicher gesprochen wird, sondern sich die Beschreibungen auf den virtuellen Speicher beziehen. Die Reihenfolge eines 1D-Arrays im realen Hauptspeicher kann durch das Laden verschiedener Speicherseiten durchaus nicht linear sein (vgl. dazu [18]).

Ein weiteres Kriterium zur optimalen Speichernutzung ist die Anzahl der Kopien der Tomogramme, die gleichzeitig im Speicher gehalten werden müssen. Die einfachste Möglichkeit ist die Verwendung von zwei Arrays. Dabei dient eines zum Lesen der Graustufenwerte und das zweite zum Schreiben der Zwischen- bzw. Endergebnisse der Binarisierung.

Bei dieser Vorgehensweise beträgt die Speichernutzung mehr als das doppelte der Größe des Tomogramms. Auf Grund des knappen Speichers sollte diese Lösung nur bei Algorithmen eingesetzt werden, bei denen diese unbedingt erforderlich sind. Betrachtet man den Aufbau des Tomogrammdatensatzes näher, so ergeben sich Möglichkeiten Speicher zu sparen. Ziel dieser Überlegungen ist es, die Kodierung (8-Bit Charakterwerte) für die Graustufenwerte im 3D-Datensatz zusätzlich für weitere Informationen zu nutzen.

Wie bereits in Abschnitt 3.1 „Binarisierung“ beschrieben, werden durch den Benutzer bei der manuellen Schwellwertangabe zur Separation einer Struktur vier Grenzen vorgegeben. Diese verteilen sich über den Graustufenbereich von 8 Bit, das heißt von 0 bis 255. Da beim Finden der Keimzellen sowie beim Wachsen um diese lediglich auf größer oder kleiner gleich der inneren bzw. äußeren Grenzwerte geprüft wird, ergeben sich freie Bereiche zwischen den einzelnen Grenzen bzw. zwischen den Grenzwerten und dem Minimum (0) oder dem Maximum (255). Liegen die Grenzwerte zum Beispiel alle unter 100, so können u.a. die Bits über 1100100b (100d) der Charakterwerte zur zusätzlichen Kodierung für die einzelnen Voxel benutzt werden. Damit diese Kodierung nicht mit den ursprünglichen Graustufenwerten kollidiert, müssen diese, wenn sie innerhalb der Grenzen liegen, auf die nächst niedrigere (bei größer gleich) bzw. nächst höhere (kleiner gleich) Grenze verschoben werben. Neben den Grenzwerten müssen zusätzlich zwei weitere Werte die unterhalb des ersten und oberhalb des letzten Grenzwertes liegen als Sammelbereich aller ebenfalls unterhalb des ersten bzw. oberhalb des letzten Grenzwertes liegenden Voxel reserviert werden. Bei vier Grenzwerten ergeben sich somit maximal $255 - 4 - 2 = 249$ zusätzliche Kodierungsmöglichkeiten pro Voxel.

Die genaue Anwendung ist in den folgenden Abschnitten beschrieben.

3.3.2 „Boundary Fill“

3.3.2.1 Rekursive Implementation

Wie bereits in Abschnitt 3.2.1 beschrieben, lässt sich der „Boundary Fill“-Algorithmus am einfachsten rekursiv implementieren. Der Pseudocode aus Listing 3.1 kann ohne Probleme in wirklichen Quellcode umgesetzt werden. Lediglich die Nachbarschaft muss über die Anzahl der eigenen Aufrufe, mittels Angabe des jeweiligen Nachbarn, festgelegt werden. Wird das Programm übersetzt und ausgeführt, so erhält man schnell korrekte Ergebnisse.

Der große Nachteil dieser Variante ist, dass durch mehrfache rekursive Aufrufe der Stackspeicher extrem schnell anwächst. Um die mögliche Größe des Stacks zu verdeutlichen, soll der „Boundary Fill“-Algorithmus an einem Beispiel untersucht werden: Gegeben sei ein Tomogramm mit einer Größe von $512 * 512 * 512$ Voxeln. Im Extremfall kann die zu separierende Struktur das komplette Volumen einnehmen. Startet der Algorithmus in einer der Ecken des Tomogramms, muss er sich für jeden Voxel erneut selber aufrufen, bevor sich die Rekursionstiefe wieder verringert. Dabei wird für jeden Funktionsaufruf ein neuer Stackframe mit der Rücksprungadresse, den Übergabeparametern und den lokalen Variablen der Funktion angelegt. Da letztere nicht vorhanden sind, enthält der Stackframe die Rücksprungadresse und die Koordinaten des zu untersuchenden Punktes. Die Grenzwerte können zur Speichereinsparung außerhalb der Funktion definiert werden, da diese immer gleich bleiben. Der Zugriff auf das Tomogramm kann ebenfalls über eine globale Variable erfolgen.

Betrachtet man eine 32-Bit Architektur, so müssen für die Rücksprungadresse 32Bit bzw. 4Byte gespeichert werden. Die Koordinaten können als „short“-Werte (16Bit/ 2Byte) übergeben werden, da dies für die maximale Tomogrammgröße (vgl. 2.2) ausreicht. Aus diesen Werten ergibt sich eine Größe von 10 Byte pro Stackframe. Bei $255^3 = 134.217.728$ Aufrufen ergibt dies eine Stackgröße von 1,25 GByte. Die Größe des Stackframes kann bei einer 32-Bit-Architektur durch Benutzung eines eindimensionalen Arrays auf 8 Byte (4Byte für die Rücksprungadresse und 4 Byte für die Array-Adressierung) reduziert werden. Bei einer maximalen Größe von 1 GByte, auch wenn diese bei großen Strukturen lediglich zur Hälfte erreicht wird, muss trotzdem mit Speichermangel gerechnet werden. Dies gilt zumindest für 32-Bit-Architekturen, bei denen in den meisten Fällen nur 2GByte Speicher pro Benutzerprozess vom Betriebssystem zugeteilt werden.

Das eigentliche Problem bei der Speicherverwaltung liegt in der Tatsache, dass die Größe des Stacks zur Programmierzeit noch nicht bekannt ist. Des Weiteren gibt es keine Möglichkeit eine feste, maximale vom Betriebssystem zuweisbare Stackgröße zur Laufzeit zu ermitteln. Zwar kann die Stackgröße durch den Benutzer bzw. dem Administrator auf eine vordefinierte Größe – die auch unendlich sein kann – festgelegt werden. Es kann jedoch nicht mit Sicherheit daraus ein Schluss über die tatsächliche maximale Größe gezogen werden. Als Konsequenz führt der Stacküberlauf zum Absturz des Programms.

3.3.2.2 Iterative Implementation

Aus den soeben gezogenen Überlegungen wurde die rekursive Implementation nicht in die endgültige Version übernommen. Da der „Boundary Fill“-Algorithmus in seiner Beschaffenheit jedoch relativ einfach umzusetzen ist, sollte dieser nicht komplett verworfen, sondern auf

eine andere, möglichst sichere Art und Weise implementiert werden. Aus diesem Grund wurde nach einer Möglichkeit gesucht, diesen nicht rekursiv bzw. iterativ zu programmieren.

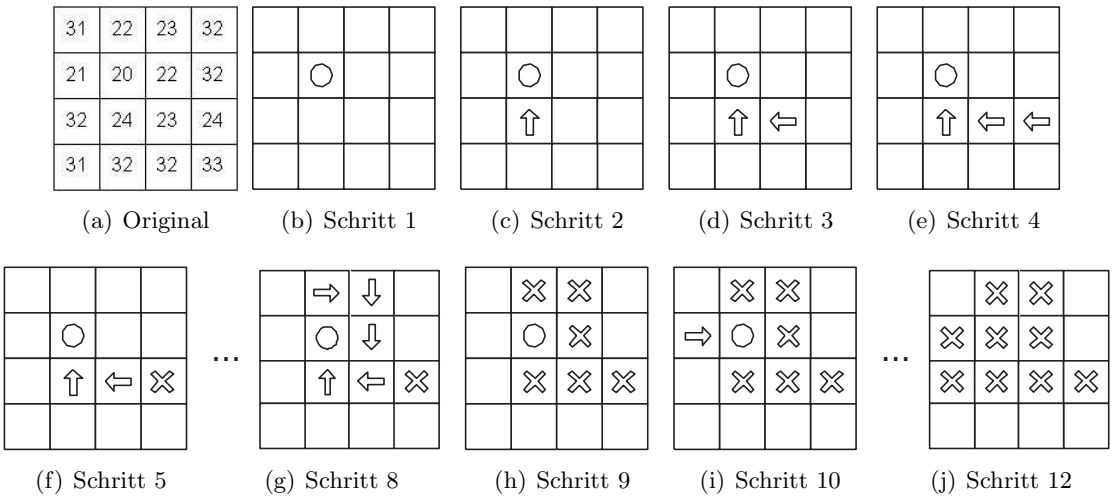
Bei der rekursiven Implementation wird durch den Aufbau des Stacks, der jeweilige vorhergehende untersuchte Voxel indirekt gespeichert. Beim Finden eines Voxels, bei dem keine Nachbarn die Wachstumskriterien erfüllen, verringert sich bei dieser Implementation die Rekursionstiefe und es wird zur aufrufenden Funktion (mit Hilfe der Rücksprungadresse) zurückgesprungen. Da die aufrufende Funktion in ihrem Stackframe die ihr übergebenen Parameter (Adresse des Voxels im Array) gespeichert hat, kann an dieser Stelle für den entsprechenden Voxel die Suche nach potentiellen Wachstumsnachbarn fortgesetzt werden.

Bei der iterativen Umsetzung muss somit ein ähnliches Konzept entwickelt werden. Da hier kein Stackframe vorhanden ist, besteht eine Möglichkeit im Aufbau eines „künstlichen“ Stacks. Auf diesen können die zum Teil bereits untersuchten Voxel gespeichert werden. Zusätzlich muss durch einen weiteren Eintrag vermerkt werden, welche Nachbarn des aktuellen Voxels schon untersucht wurden. Der Rumpf des Algorithmus besteht aus einer Schleife, die solange ausgeführt wird, bis kein neuer Voxel mehr hinzugekommen und der Stack nicht leer ist. Findet der Algorithmus einen neuen Nachbarn, der die Vorgaben erfüllt, werden seine Adresse sowie die Kodierung für die bereits untersuchten Nachbarn auf den Stack gelegt und die Schleife beginnt mit der Adresse des neuen Voxels von vorn. Können keine neuen Nachbarn mehr gefunden werden, wird die Adresse des zuletzt untersuchten Voxels vom Stack geholt und die Schleife setzt an der Stelle des zuletzt untersuchten Nachbarn (ebenfalls vom Stack) für diesen Voxel fort.

Betrachtet man den Speicherbedarf bei diesem Vorgehen, so stellt man fest, dass dieser ungefähr, je nach Kodierung der bereits untersuchten Nachbarn, genau so groß wie bei der rekursiven Variante werden kann. Der Vorteil bei dieser Umsetzung ist jedoch, dass das Programm nicht abstürzt, da der „künstliche“ Stack überwachbar ist. Nachteilig äußert sich die unbekannte maximale Speichergröße. Sollte das Betriebsystem während des Wachsens keinen Speicherplatz mehr zur Verfügung stellen, kann der Algorithmus zwar kontrolliert beendet werden, das Ergebnis ist jedoch nur unvollständig. Eine Alternative dazu wäre das Anlegen einer Bitmap vor dem Starten des „Boundary Fill“-Algorithmus, in dem die Rücksprungadressen und die bereits untersuchten Nachbarn gespeichert werden. Dadurch kann von vornherein bestimmt werden, ob der Speicher ausreichen wird.

Um den Speicherbedarf nachhaltig zu verringern, kann der zuletzt untersuchte Voxel nicht durch seine absolute Adresse im Array, sondern durch die Richtung in der er sich befindet, gespeichert werden. Die Richtung lässt sich durch den relativen Adresswert vom zu untersuchenden Voxel beschreiben. Da die Nachbarschaft im 3D-Raum auf maximal 26 Nachbarn begrenzt ist, reicht ein Charakterwert zum Speichern der Richtung aus. Legt man die Reihenfolge, in der die Nachbarn untersucht werden, bereits vorher fest, so kann aus dem relativen Adresswert auch zusätzlich die Information über die bereits untersuchten Nachbarn gewonnen werden. Bei dieser Vorgehensweise reicht eine Array von der Größe des Tomogramms zur Kodierung aus, so dass der zum Tomogramm zusätzliche benötigte Speicher, in dem im vorherigen Abschnitt beschriebenen Beispiel, auf $255 * 255 * 255 * 1Byte = 128MByte$ sinkt. In diesem Fall wird wie in 3.3.1.4 beschrieben, dass Tomogramm-Array zum Lesen und die Kopie zum Bearbeiten verwendet.

Bei der Umsetzung des Programms (Quellcode im Anhang B.1) wurde auf die soeben beschriebenen Überlegungen zurückgegriffen. Eine schematische Darstellung der Implementie-



(a) Original (b) Schritt 1 (c) Schritt 2 (d) Schritt 3 (e) Schritt 4

(f) Schritt 5 (g) Schritt 8 (h) Schritt 9 (i) Schritt 10 (j) Schritt 12

Abbildung 3.10: Schematische Darstellung der Implementation des „Boundary Fill"-Algorithmus anhand eines 2D-Beispiels: Die Pfeile, welche auf den zuvor untersuchten Voxel verweisen, der Kreis, welcher die Keimzelle beschreibt, und das Kreuz, welches auf eine fertige Bearbeitung verweist, besitzen bei der Umsetzung jeweils eine bestimmte Kodierung. Der hier skizzierte Fall basiert auf einer Vierer-Nachbarschaft der Hauptachsen. Die Grenzwerte für den äußeren Bereich liegen bei 0 und 25, die für den inneren bei 0 und 21.

rung des „Boundary Fill"-Algorithmus ist in Abbildung 3.10 dargestellt.

Bevor es zur Ausführung des „Boundary Fill"-Algorithmus kommt, werden alle Nachbarn mittels Offset von dem zu untersuchenden Voxel in einem Array gespeichert. Die Nachbarschaften werden aufsteigend durch eine zusammenhängende Reihe definiert. Die Pfeile in den unterschiedlichen Ausrichtungen beschreiben in Abbildung 3.10 die Reihe zur Richtungskodierung. Jede Ausrichtung entspricht einem Wert der Reihe. Die Reihenfolge ist dabei wie folgt festgelegt: Zunächst wird der Nachbar unterhalb des zu untersuchenden Pixels (Pfeil nach oben) überprüft, danach der Rechte (Pfeil nach links), der Linke (Pfeil nach rechts) und zuletzt der Nachbar oberhalb des zu untersuchenden Pixels (Pfeil nach unten). Dadurch ergibt sich eine Vierer-Nachbarschaft mit allen angrenzenden Pixeln auf den Hauptachsen (X und Y). Zusätzlich existieren zwei weitere Kodierungen: a) der Voxel ist die Keimzelle (Kreis in 3.10) und b) der Voxel ist bereits binarisiert (Kreuz in 3.10).

Der Rumpf des Algorithmus ist eine Schleife (Quellcode im Anhang B.1), die beim Erreichen der Keimzelle terminiert. Vor dem Starten der Schleife wird die der Funktion übergebene Keimzelle kodiert. Danach wird die Schleife mit dem entsprechenden Nachbarschaftswert, welcher zu Beginn auf den niedrigsten Wert der Reihe initialisiert wird, durchlaufen. Findet sich ein Nachbar, der die Kriterien erfüllt, wird in dem „Arbeits-Array" an der Adresse des Nachbarn die Richtung des zuvor untersuchten Voxel gespeichert und dieser zur neuen Keimzelle. Ist kein Nachbar zu finden, erhöht sich der Nachbarschaftszähler solange, bis er seinen Maximalwert erreicht hat. Ist dies der Fall, wird der Inhalt der Adresse des gerade untersuchten Voxel ausgelesen. Enthält er die Kodierung für die Keimzelle, endet die Schleife. Ansonsten wird über die Richtungskodierung der zuvor untersuchte Voxel ermittelt und der Nachbarschaftszähler auf den nächsten zu untersuchenden Nachbar gesetzt.

Um die Verarbeitungsgeschwindigkeit zu erhöhen wurde bei der Implementation für den Bilddatensatz ein eindimensionales Array verwendet. Die Verringerung der Ausführung ist auf die Array-Adressierung (vgl. Abschnitt Datenverwaltung im Arbeitsspeicher) zurückzuführen. Da durch das „Nachbarschafts-Array“ die Offsets zu den Nachbarn von vornherein bekannt sind, erfolgt die Adressierung schneller als mit einem 3D-Array. Der große Nachteil dabei ist, dass die Dimensionen des Bildes bei der Verwendung eines 1D-Arrays verloren gehen. Aus diesem Grund wurde das „Nachbarschafts-Array“ doppelt erstellt: Zusätzlich neben dem 1D-Array mit entsprechenden Offsets für das eindimensionale Bilddaten-Array auch ein 3D-„Nachbarschafts-Array“ mit Offsets in X-, Y- und Z-Richtung. Dieses dient zur Überprüfung, ob der durch Addition des Offsets zur momentanen Keimzelle, zu untersuchende Nachbarvoxel noch innerhalb der Dimensionen des Bildes liegt. Damit die X-, Y- und Z-Koordinaten nicht bei jedem Schritt neu bestimmt werden müssen, wird die momentane Keimzelle nicht nur als Index im 1D-Array, sondern auch in ihren X-, Y- und Z-Koordinaten bei jedem Schritt mitgeführt (vgl. Quellcode Anhang B.1)

Die letzte Möglichkeit um erneut Speicher zu sparen, welche auch bei der Implementation mit verwendet wurde, ist die Verwendung eines einzigen Arrays von der Größe des Tomogrammdatensatzes. Dazu soll auf die Überlegungen im Abschnitt Datenverwaltung im Arbeitsspeicher (3.3.1.4) zurückgegriffen werden. Da bei der soeben vorgestellten Implementierung lediglich 30 Kodierungsmöglichkeiten notwendig sind, kann durch das Umkodieren von Graustufenbereichen genug Platz für diesen Zweck erzeugt werden. Die Implementation (vgl. Anhang B.1) erfolgt in der gleichen Weise wie mit zwei Feldern. Nur die Überprüfung der Nachbarschaftskriterien muss entsprechend angepasst werden. Zusätzlich werden die Graustufenwerte des Tomogrammdatensatzes wie in 3.3.1.4 angepasst und der „Boundary Fill“-Funktion ein Arbeitsbereich übergeben, der für die Codierung benutzt werden kann.

Die zuletzt dargestellte Umsetzung enthält die für den „Boundary Fill“-Algorithmus die beste Variante in Hinsicht auf die Speichernutzung. Ein positiver Randeffekt bei der Benutzung eines Nachbarschafts-Array ist, dass der Nutzer der Bibliothek dieses frei wählen kann. Zusätzlich wurden einige Standardnachbarschaften bereits vordefiniert. Die Ausführungsgeschwindigkeit kann lediglich durch den Einsatz von Parallelverarbeitung zusätzlich erhöht werden. Umsetzungen dazu werden in Abschnitt 3.3.4 beschrieben.

3.3.3 „Fill“

Die Implementierung des „Fill“-Algorithmus folgt ähnlichen Überlegungen wie bei der Umsetzung des „Boundary Fill“-Algorithmus. In erster Linie soll eine möglichst speicherschonende Variante programmiert werden. Da es sich beim „Fill“-Algorithmus um ein iteratives Verfahren handelt (siehe 3.2.2), treten keine Probleme im Zusammenhang mit dem Stackspeicher auf. Zur Codierung von Zwischenergebnissen wird wie beim „Boundary Fill“-Algorithmus ein „Arbeitsbereich“ innerhalb von nicht benötigten Graustufenwerten festgelegt, sodass zur Bearbeitung ebenso nur Speicher in der Größe des Tomogramms benötigt wird. Näheres dazu ist den folgenden Absätzen beschrieben.

Um eine in Bezug auf die Rechenzeit effektive Umsetzung des „Fill“-Algorithmus zu erreichen, muss von der eigentlichen Vorgehensweise, wie sie in Abschnitt 3.2.2 beschrieben ist, stark abgewichen werden. Der Grund dafür liegt in der aufwendigen Berechnung der einzelnen Iterationsschritte, welche eine Dilatation des vorläufigen Ergebnisses und den Schnitt mit

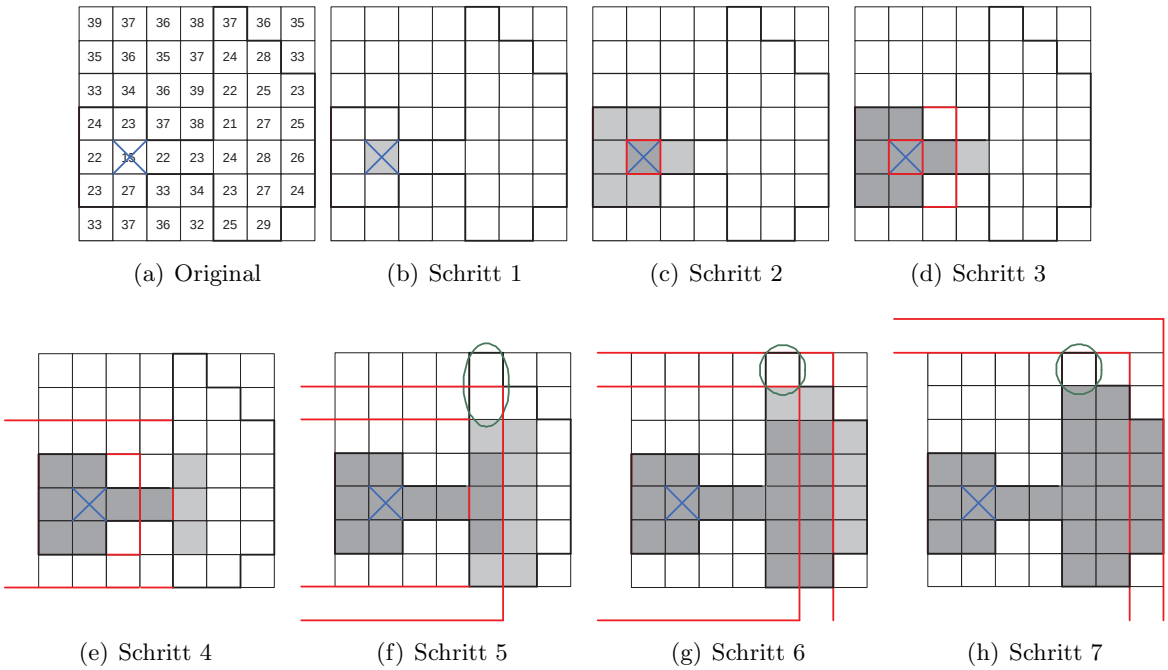
dem Originalbild enthalten. Insbesondere die Dilatation der Ergebnismenge, die mit steigendem Iterationsschritt erheblich anwächst, ist mit einem großen Rechenaufwand im 3D-Raum verbunden.

Aus diesem Grund wurden in erster Linie Überlegungen zur Reduktion des Rechenaufwands der Dilatation durchgeführt. Die einfachste Möglichkeit besteht in einer unterschiedlichen Codierung von Voxeln. Die durch die vorherigen Dilatationen hinzugewonnen Pixel, werden mit einem bestimmten Wert als Füllmenge codiert. Im Unterschied dazu erhalten Voxel, die im aktuellen Iterationsschritt hinzukommen sind, eine andere Codierung. Der Vorteil dabei ist, dass im nächsten Schritt die Dilatation lediglich auf die zuvor neu hinzugewonnenen Voxel beschränkt werden kann. Zusätzlich zur doppelten Codierung wird der anschließende Schnitt mit der Komplementärmenge des Originalbildes im selben Schritt wie die Dilatation durchgeführt. Beim Dilatieren werden die entsprechenden Nachbarvoxel überprüft, ob ihre Graustufenwerte im Originalbild innerhalb der äußeren beiden Grenzen liegen. Nur wenn dies der Fall ist, wird der Voxel der Ergebnismenge hinzuaddiert.

Die nächste wichtige Thematik beschäftigt sich mit dem zu untersuchenden Raumvolumen pro Iterationsschritt. Eine Möglichkeit, die ohne großen Aufwand umsetzbar ist, ist das Durchsuchen des kompletten Bildvolumens. Dies führt jedoch zu einem erheblichen Aufwand, sodass die Berechnungszeit extrem groß ist. Eine weitaus bessere Variante ist die Verknüpfung der zu untersuchenden Voxel an das Strukturelement bzw. die Nachbarschaft und den jeweiligen Iterationsschritt. Dies ist durch die Charakteristik der Dilatation gegeben. Beim Wachsen von einem bestimmten Punkt aus, vergrößert sich die Struktur in der entsprechenden Form des Strukturelements. Im Fall einer Nachbarschaftsbeziehung mit 26 angrenzenden Voxeln, die jeweils eine Fläche (Nachbarn auf den Hauptachsen), ein Gerade oder Punkt (Nachbarn auf den Diagonalen) mit dem zu untersuchenden Raumpunkt gemeinsam haben, wächst die Struktur würfelförmig mit dem Ursprungsvoxel als Mittelpunkt. Nimmt man dagegen lediglich die 6 auf den Hauptachsen angrenzenden Voxel als Nachbarn, so entsteht ein dreidimensionales Kreuz.

Aus diesen Überlegungen wäre die einfachste Variante bei einer 26-iger Nachbarschaft das zu untersuchende Raumvolumen auf einen mit jedem Iterationsschritt wachsenden Würfel zu beschränken. Der Mittelpunkt des Würfels ist die Keimzelle, die Kantenlänge beträgt zweimal den Iterationsschritt plus eins, wobei mit dem nullten Schritt begonnen wird. Diese Implementation funktioniert auch für beliebige Nachbarschaften, wobei ggf. die Kantenlänge des Würfels bei Nachbarn die mehr als ein Voxel von dem zu untersuchenden Bildpunkt entfernt sind, vergrößert werden muss. Alternativ kann das Volumen exakt dem Strukturelement in der entsprechenden Größe des jeweiligen Iterationsschritts angepasst werden. Dies hat jedoch einen deutlichen Mehraufwand zur Folge.

Trotz dieser Verbesserung bei der Umsetzung des „Fill“-Algorithmus beträgt die Rechnenzeit immer noch das Zehn- bis Fünfzehnfache der des „Boundary-Fill“-Algorithmus. Aus diesem Grund wurde eine weitere Geschwindigkeitssteigerung bei der Implementation versucht. Da neue Voxel lediglich an der pro Iterationsschritt aktuellen „Oberfläche“ der gewachsenen Struktur hinzugewonnen werden, besteht die Möglichkeit nur diese zu untersuchen. Die Oberflächenform entspricht der des Strukturelements (siehe oben). Damit lediglich sechs Ebenen untersucht werden müssen, wurde bei dieser Variante die 26-iger Nachbarschaft (Würfel mit den zu untersuchenden Voxeln als Mittelpunkt) festgelegt. Als Nachteil dieser Begrenzung ergibt sich, dass der Benutzer nach dieser Geschwindigkeitsoptimierung die Nachbarschaft



(a) Original (b) Schritt 1 (c) Schritt 2 (d) Schritt 3

(e) Schritt 4 (f) Schritt 5 (g) Schritt 6 (h) Schritt 7

Abbildung 3.11: Begrenzung der Dilatation beim „Fill“-Algorithmus durch Optimierung: Wird beim „Fill“-Algorithmus lediglich die Kante/Oberfläche des Quadrats/Würfels (rot eingeschlossener Bereich) auf neu hinzugewonnene Pixel/Voxel zur Füllmenge untersucht, muss auf dieser solange dilatiert werden, bis keine neuen Pixel/Voxel mehr hinzugekommen sind. Ist dies nicht der Fall kann es zu Fehlern, wie sie in Bild (f), (g) und (h) im grün markierten Bereich zu sehen sind, kommen. Im Schritt 5 wurde die Kante (Pixel am Rand der roten Linie in Richtung Keimzelle/blaues Kreuz) nur einmal von oben nach unten durchlaufen, sodass die beiden oberen Pixel nicht mit zur Füllmenge hinzugekommen sind. Bei der Dilatation im 6. Schritt wird durch 8-er Nachbarschaftsbeziehung zwar einer der beiden fehlenden Pixel noch der Ergebnismenge hinzugefügt. Der Bildpunkt in der obersten Spitze kann jedoch in keinem weiteren Schritt mehr zur Füllmenge hinzukommen.

nicht mehr frei wählen kann.

Bei der Begrenzung der Untersuchung auf die Oberfläche muss jedoch beachtet werden, dass es durch die bedingte Dilatation entsprechend den Wachstumskriterien zu einer unvollständigen Ausdehnung auf den Oberflächen kommen kann. Aus diesem Grund muss pro zu untersuchender Oberfläche solange dilatiert werden, bis kein neuer Voxel mehr dazukommt. Ein Beispiel dazu ist in Abbildung 3.11 dargestellt: Dieses wurde zur einfacheren Darstellung für ein Bild im 2D-Raum durchgeführt. Die Nachbarschaft/das Strukturelement ist dementsprechend durch acht Nachbarn definiert, sodass im Gegensatz zu einem 3D-Datensatz lediglich vier Kanten pro Iterationsschritt untersucht werden müssen. Diese werden durch die roten Markierungslinien begrenzt. Die hier abgebildete Struktur besitzt eine ein Pixel breite Verbindung zwischen zwei größeren Teilstücken. In den einzelnen Schritten wird dargestellt, wie neue Pixel (hellgrau) zu der Füllmenge aus den letzten Iterationsschritt (dunkelgrau) hinzugewonnen werden.

In diesem Beispiel wird davon ausgegangen, dass die vertikalen Kanten von oben nach unten durchsucht werden. Beim Erreichen des 5. Schrittes (3.11(f)) tritt zum ersten Mal ein Fehler auf (grüner Bereich), welcher sich in den folgenden Schritten fortsetzt. Auf Grund der einen Pixel breiten Verbindung sind bei der Dilatation im 4. Schritt nicht alle Pixel auf der zu untersuchenden Kante des 5. Schrittes zur Füllmenge hinzugekommen. Aus diesem Grund wird beim einmaligen Durchsuchen der Kante der oberste Pixel nicht mit zur Ergebnismenge hinzugerecht. Erst beim zweiten Mal würden auch dieser Bildpunkt mit erkannt werden. Nach der dritten Dilatation im Bereich der Kante würde zum nächsten Iterationsschritt übergegangen werden, da nun keine neuen Pixel mehr hinzukommen würden.

Neben der speziellen Betrachtung der zu untersuchenden Kanten/Oberflächen fällt in diesem Beispiel eine weitere Möglichkeit zur Geschwindigkeitsoptimierung auf. Pro Iterationsschritt werden alle vier Kanten in die entsprechenden Richtungen vergrößert. Dies ist auch der Fall, wenn in der jeweiligen Dimension kein neuer Bildpunkt mehr zur Füllmenge hinzugekommen ist oder bereits die Grenzen des Bildes erreicht sind. Aus diesem Grund wurde als eine weitere Optimierung das Ausdehnen der Kanten/Oberflächen an die Bedingung geknüpft, ob ein neuer Bildpunkt in der entsprechenden Richtung zur Ergebnismenge hinzugekommen ist. Beim Treffen auf den Rand erfolgt die Begrenzung ebenso, da hier keine neuen Bildpunkte mehr gefunden werden können.

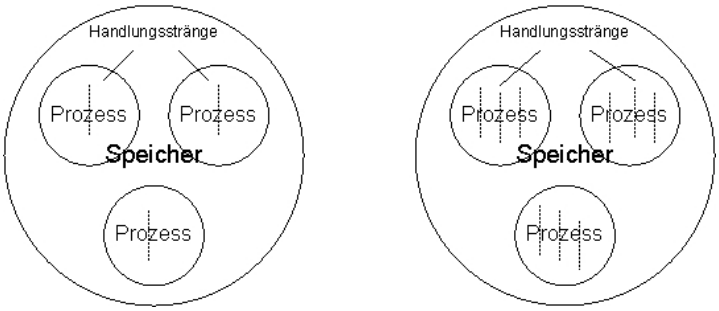
Auch die letzten beiden Verbesserungen der „Fill“ Implementation konnten die Bearbeitungszeit nicht entscheidend verkürzen. Bei der Binarisierung des Tomogramms mit dem „Fill“- und „Boundary Fill“-Algorithmus musste festgestellt werden, dass ersterer im Mittel eine vier bis fünffach so große Rechenzeit benötigte. Zusätzlich kann mit der letzten Verbesserung des „Fill“-Algorithmus lediglich eine vordefinierte Nachbarschaft mit 26 Voxeln zur Segmentierung benutzt werden. Die Berechnung der aktuellen Oberfläche bei einem beliebigen Strukturelement ist relativ kompliziert und würde mit großer Wahrscheinlichkeit mehr Rechenzeit in Anspruch nehmen als die Untersuchung eines entsprechend großen Würfelvolumens. Aus diesen Gründen wurde in der endgültigen Bibliotheksversion als Wachstumsverfahren der „Boundary Fill“-Algorithmus verwendet.

3.3.4 Einbinden von Parallelverarbeitung

3.3.4.1 Umsetzungsmöglichkeiten von Parallelverarbeitung

Bei einer zeitgleichen Verarbeitung gibt es grundsätzlich zwei Möglichkeiten, um die Parallelität unter Unix zu erreichen: das Erstellen von mehreren Threads oder das Erstellen von mehreren Prozessen. Threads sind parallele Handlungsstränge, die innerhalb eines Prozesses laufen. Diese können, je nach Umsetzung im Betriebssystem, als User- oder Kernel-Threads implementiert werden. Im ersten Fall erfolgt das Scheduling durch ein Programm bzw. Dämon, der als Hintergrundprozess läuft. Bei einem Kernel-Thread wird das Scheduling direkt vom Betriebssystem übernommen. Da Threads innerhalb eines Prozesses laufen, können sie auf gemeinsamen Speicher zurückgreifen.

Im Gegensatz zum Thread erfolgt das Scheduling bei Prozessen generell durch das Betriebssystem. Da Prozesse komplett getrennt voneinander laufen, verfügen sie über den pro Prozess maximal durch das Betriebssystem zur Verfügung gestellten Speicher. Die Kommunikation untereinander kann nicht wie bei Threads über den Speicher des Prozessen (Heap) erfolgen. Hier sind komplexere Funktionen wie Semaphoren, Pipe, Signale, etc., welche über gemeinsamen, außerhalb der Prozesse reservierten Speicher laufen, notwendig. (Detailierte Ausführungen in [18] und [12].)



(a) Parallele Ausführung von Prozessen (b) Parallele Ausführung von Prozessen

Abbildung 3.12: Vergleich: Parallele Verarbeitung mit Prozessen und Threads: Bei paralleler Verarbeitung mit Prozessen bekommt jeder den maximal vom Betriebssystem zuweisbaren Speicher. Die Kommunikation der Handlungsstränge untereinander muss über den „shared memory“ mittels Semaphoren, Pipes oder Signalen erfolgen. Parallele Handlungsstränge, die durch Threads implementiert sind, können dagegen auf gemeinsamen Speicher zurückgreifen, da diese innerhalb eines Prozesses laufen.

Durch die soeben beschriebenen Eigenschaften ergeben sich verschiedene Vor- und Nachteile bzw. Einsatzgebiete für Threads und Prozesse. Während Threads durch Kommunikation über gemeinsamen Speicher relativ einfach synchronisiert werden können, haben Prozesse den Vorteil der Unabhängigkeit voneinander. Letzterer Umstand wird meist bei sicherheitsrelevanten Anwendungen genutzt, da hier für verschiedene Nutzer Prozesse mit unterschiedlichen Rechten gestartet werden können. Da dieses Kriterium bei der Binarisierung primär keine Rolle spielt, entfällt dieser Vorteil von Prozessen.

Das nächste wichtige Thema, das Scheduling, hat eine große Bedeutung hinsichtlich der Be-

rechnungsgeschwindigkeit. Die Parallelverarbeitung bei der Binarisierung soll zur Verkürzung der Ausführungszeit führen. Sind die Threads als User-Threads implementiert, können zwar mehrere parallele Handlungsstränge erzeugt werden, da diese jedoch innerhalb eines Prozesses laufen, wird ihnen lediglich die Rechenzeit für einen Prozess zur Verfügung gestellt. Da die nicht-parallelisierte Ausführung in einem Prozess bereits die CPU komplett auslastet, bringt dieses keinen Geschwindigkeitsvorteil. Insbesondere bei Computern mit mehreren CPUs können diese nicht genutzt werden. Bei modernen Threadimplementationen, wie die bei der Umsetzung eingesetzten Posix-Threads, handelt es sich jedoch um Kernel-Threads. Aus diesem Grund erfolgt das Scheduling genau wie bei Prozessen direkt durch das Betriebssystem, sodass es in dieser Beziehung keinen Unterschied zwischen Threads und Prozessen gibt.

Ein weiterer interessanter Punkt ist die Speicherverwaltung und Kommunikation zwischen verschiedenen Prozessen bzw. Threads. Da erstere den kompletten maximal vom Betriebssystem zugeteilten Speicher erhalten können, ist es möglich, bei einer Aufteilung des Tomogramms, wenn dies der entsprechende Algorithmus zulässt, größere Datenmengen zu verarbeiten. Bei Threads ist dies zwar nicht möglich, im Gegensatz zu Prozessen ist die Kommunikation durch den gemeinsam genutzten Speicher aber erheblich einfacher.

3.3.4.2 Parallelisierung der Keimzellensuche

Eine Möglichkeit zur Parallelverarbeitung, die für beide Algorithmen angewandt werden kann, ist das gleichzeitige Aufsuchen von Keimzellen, verbunden mit dem anschließenden Wachsen. Zur Umsetzung dieses Ansatzes gibt es zwei verschiedene Möglichkeiten: die Bearbeitung in einem (Tomogramm-)Datensatz und das Unterteilen des Tomogramms in verschiedene Abschnitte.

Bei der ersten Variante muss in jedem Fall berücksichtigt werden, dass parallel ablaufende Threads oder Prozesse zu einem bestimmten Zeitpunkt gleichzeitig auf Werte des Tomogramms zugreifen können. Das Lesen oder Schreiben im Tomogramm-Array erfolgt natürlich nacheinander, da jeweils nur ein Thread oder Prozess zu einem bestimmten Zeitpunkt die CPU haben kann. Auf Basis des gelesenen Wertes wird jedoch eine Entscheidung hinsichtlich des Fortlaufens des Algorithmus getroffen. Dementsprechend wird u.a. ein neuer Kodierungswert an die soeben ausgelesene Stelle geschrieben, die in diesem Fall dem „Boundary Fill“-Algorithmus signalisiert, dass er den Voxel bereits (zum Teil) untersucht hat. Ein Konflikt entsteht, wenn zuerst der eine Thread oder Prozess den Wert ausliest und dann der Scheduler einem anderen Thread oder Prozess die CPU zur Verfügung stellt. In diesem Fall kann es passieren, dass in beiden Handlungssträngen davon ausgegangen wird, dass der Voxel noch nicht untersucht wurde. Danach schreibt zunächst der eine Thread oder Prozess seine Codierung, die danach durch den anderen überschrieben wird. Die Folgen sind nicht zu erklärende Fehler bis hin zum Programmabsturz.

Aus diesem Grund müssen die verschiedenen Aktionspfade synchronisiert werden. Beim Einsatz von Threads besteht die Möglichkeit, kritische Codebereiche durch im gemeinsamen Speicher angelegte Mutexe zu sichern. Diese können vor Eintritt in den kritischen Bereich gesperrt und danach wieder freigegeben werden. Wichtig dabei ist, dass die Operationen „mutex_lock“ und „mutex_unlock“ atomar sind. Das heißt, während ihrer Ausführung kann der Scheduler nicht einem anderen Thread die CPU zur Verfügung stellen. Die entsprechende

Umsetzung ist im Betriebssystem implementiert. Eine Fehlerquelle, die beim Einsatz von Mutexen entstehen kann, ist das Sperren ohne nachfolgendes Freigeben. Dies führt zum Deadlock und muss in jedem Fall bei der Programmierung vermieden werden.

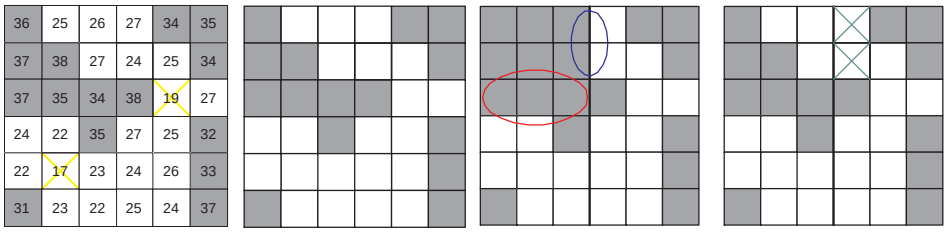
Der Test der sperrbasierten Synchronisation erfolgte anhand des „Boundary Fill"-Algorithmus. Dazu wurden zunächst zwei Threads erstellt, die gleichzeitig das Tomogramm von entgegengesetzten Seiten nach Keimzellen durchsuchen. An dieser Stelle muss bereits der erste Schutz eingesetzt werden, da die beiden Threads nicht gleichzeitig ein Voxel auf eine Keimzelle hin überprüfen dürfen. Die Sperre kann erst nach Aufruf des „Boundary Fill"-Algorithmus und dem Kodieren des Voxel als Keimzelle entfernt werden. Ist dies vorgenommen, wird der zweite Thread diesen Voxel nicht mehr untersuchen, da er bereits als „in Bearbeitung" gekennzeichnet ist. Der nächste Schutz muss bei der Überprüfung der Nachbarn eingebaut werden. Die Aufhebung kann wie bei der ersten Sperre erst nach der Kodierung des Nachbarn bzw. nach Feststellen des Nicht-Erfüllens der Nachbarschaftskriterien veranlasst werden.

Nach Sichern der beiden kritischen Codebereiche kann der Algorithmus gefahrlos parallel verarbeitet werden. Bei dem Einsatz der Mutexe müssen jedoch zuvor noch die Sperrbereiche festgelegt werden. Damit sich beide Threads möglichst wenig gegenseitig blockieren, muss pro Voxel eine Mutexvariable angelegt werden. Dies ist jedoch in Hinsicht auf den zusätzlichen Speicherverbrauch (4Byte pro Mutex) nicht sinnvoll. Zusätzlich kann das Betriebssystem eine so große Anzahl von Mutexen nicht verwalten und wird die Initialisierung der Mutexe verweigern. Eine bessere Variante ist das Sperren einer Ebene pro Mutex. Dies beinhaltet eine viel geringere weitere Speicherbelastung, führt in ungünstigen Fällen aber zum gegenseitigen Blockieren der Threads.

Beim Binarisieren mit Multithreading und sperrbasierter Synchronisation musste festgestellt werden, dass die Ausführungszeit extrem anstieg. Die Begründung hierfür liegt in der Zeit, die für das Sperren und Freigeben der Mutexe benötigt wird. Diese ist zwar relativ klein, betrachtet man jedoch die Häufigkeit der Aufrufe, so erkennt man die Problematik: Für jeden Voxel muss bei der Keimzellensuche pro Thread eine Lock- und eine Unlock-Operation ausgeführt werden. Des Weiteren wird bei der Nachbarschaftsprüfung bei jeder Untersuchung eines Nachbarn ein Lock und ein Unlock durchgeführt. Im Schnitt, je nach Größe der zu separierenden Strukturen, muss für die Hälfte aller Voxel für alle Nachbarn (26) ein Mutex einmal gesperrt und einmal entsperrt werden. Betrachtet man das bereits untersuchte Beispiel eines Tomogramms mit 512^3 Voxel so ergeben sich $512^3 * 2 * 2 + 512^3 / 2 * 2 * 26 = 4026531840 \approx 4$ Millarden Mutexoperationen. Diese Anzahl verdeutlicht den zusätzlichen Rechenaufwand. Eine Synchronisation mit Semaphoren kommt ebenfalls nicht in Frage, da entsprechende Operationen ähnlich lang dauern. Beim Einsatz von mehreren Prozessen würde sich die Ausführungszeit nochmals verlängern, da die Kommunikation mit Semaphoren über einen zusätzlichen geteilten Speicherbereich (shared memory) im Hauptspeicher führt. Auf Grund dieser Probleme musste eine Alternative gefunden werden. Diese verwendet das Aufteilen des Tomogramms in mehrere Teilabschnitte.

Diese Vorgehensweise – Aufteilung des Tomogramms in mehrere Abschnitte – begrenzt die Ausführung der Füllalgorithmen auf Teilbereiche des Tomogramms. Aus diesem Grund ist eine Überprüfung des Ergebnisses, da dadurch die beiden Algorithmen nicht korrekt – auf das ganze Tomogramm betrachtet – funktionieren, notwendig. Diese Tatsache ergibt sich aus den Nachbarschaftsbeziehungen. Da bei beiden Algorithmen auf die Nachbarn des zu untersuchenden Voxels zurückgegriffen werden muss, handelt es sich bei diesen um lokale

Operationen. Wird das Tomogramm an zum Beispiel einer Ebene geschnitten, kann beim Wachsen nicht mehr auf alle in der Nachbarschaft definierten angrenzenden Voxel zugegriffen werden. Dadurch können sich Fehler wie in Bild 3.13(c) dargestellt ergeben.



(a) Original (b) Verarbeitung innerhalb des kompletten Datensatzes (c) Parallele Verarbeitung ohne Korrektur (d) Parallele Verarbeitung mit Korrektur

Abbildung 3.13: Fehler bei räumlich begrenzter Ausführung des „Boundary Fill"-Algorithmus: Das zweite Bild (b) zeigt eine Struktur, wie sie durch das Ausführen des „Boundary Fill"-Algorithmus im kompletten Datensatz entsteht. Beim Aufteilen (c) werden Teile der Struktur nicht mit binarisiert, da diese in der einen Hälfte nicht über Nachbarn (roter Bereich) mit der Keimzelle (gelbe Kreuze) verbunden sind. Beim Wachsen in der zweiten Hälfte wird ein Teil ebenfalls durch Erreichen der Grenze (blauer Bereich) nicht erreicht. Erst durch das Untersuchen der Schnittkanten bzw. -ebenen (d) und das Durchlaufen des „Boundary Fill"-Algorithmus im ganzen Datensatz an positiv binarisierten Pixeln bzw. Voxeln wird dasselbe Ergebnis wie in (b) erzielt.

Um dieses Problem zu umgehen, werden nach Abarbeitung der Teilbereiche durch mehrere Threads oder Prozesse, die Übergangsregionen erneut untersucht. Befindet sich an den Schnittflächen der Teilbereiche ein Vogel, der positiv (zu einer zu separierenden Struktur gehörend) binarisiert wurde, wird dieser als Keimzelle betrachtet und der entsprechende Füllalgorithmus von diesem Punkt global im gesamten Tomogramm noch einmal ausgeführt.

Der große Vorteil bei der Aufteilung des Tomogramms ist, dass die parallelen Handlungsstränge nicht mehr synchronisiert werden müssen. Die anschließende Überprüfung wird dementsprechend ohne Parallelverarbeitung durchgeführt. Der letzte Umstand führt auch zur Entscheidung der Umsetzung der Parallelität. Der Einsatz von mehreren Prozessen, welcher ggf. auch zur Verarbeitung von größeren Tomogrammen führt (jeder Prozess bekommt den maximalen Speicher vom Betriebssystem), ist prinzipiell möglich. Da die Füllalgorithmen im Nachhinein jedoch global durchlaufen werden müssen, können lediglich Tomogramme von der Größe des maximal pro Prozess zuweisbaren Speichers bearbeitet werden. Deshalb wurde eine Implementierung mit Threads umgesetzt (s. Anhang B.2).

Die Ergebnisse, die mit diesen Verfahren gemacht wurden, entsprachen denen ohne Parallelverarbeitung. Die Ausführungszeit konnte je Beschaffenheit des Tomogramms bei zwei Threads nahezu halbiert werden.

3.3.4.3 Parallelverarbeitung der Algorithmen

Da die Implementierung des „Fill"-Algorithmus auf Grund der im Abschnitt 3.3.3 beschriebenen Probleme nicht weiter fortgesetzt wurde, beziehen sich nachfolgende Überlegungen auf

den „Boundary-Fill“ Algorithmus.

Der „Boundary Fill“-Algorithmus, wie er in Abschnitt 3.3.2 beschrieben ist, lässt sich gar nicht bzw. nur mit extremen Aufwand parallelisieren. Eine denkbare Variante ist das gleichzeitige Untersuchen der Nachbarn. Dies spart vor allen beim Vorhandensein von vielen Nachbarn, die nicht die Nachbarschaftskriterien erfüllen, eine Menge Zeit. Allerdings müsste das Ergebnis für eine weitere Verarbeitung gespeichert werden, da ansonsten durch erneutes Untersuchen der Nachbarn der Geschwindigkeitsvorteil verloren geht. Ein weiteres Problem stellt die geringe Anzahl der Operationen pro parallelem Handlungsstrang in diesem Fall dar. Die Systemzeit, die benötigt wird, um die entsprechenden Threads oder Prozesse zu erstellen, würde im Verhältnis zur eigentlichen Ausführungszeit zu groß sein, sodass der Vorteil der parallelen Verarbeitung nicht mehr vorhanden ist. Alternativ besteht die Möglichkeit eine der Anzahl der Nachbarn entsprechende Anzahl von Threads oder Prozessen zu erstellen, welche für die komplette Binarisierung zur Verfügung stehen. Diese müssen dann durch einen Kontroll-Thread oder -Prozess zur Verarbeitung der Nachbarn synchronisiert werden. Die Verwendung von Mutexen oder Semaphoren würde dann aber zu den gleichen Problemen wie in Abschnitt zuvor beschrieben führen.

Eine letzte Möglichkeit zur Parallelverarbeitung wäre das Beschränken des Wachsens auf verschiedene Teilbereiche. So könnten mehrere parallele Handlungsstränge erzeugt werden, die jeweils in unterschiedlichen Quadranten vom Ursprung (der Keimzelle) wachsen. Da hier jedoch der Algorithmus lokal beschränkt wird, müssten auch hier, wie bei der parallelen Keimzellensuche in Teilbereichen, im Nachhinein die Schnittebenen untersucht und ggf. korrigiert werden. Da die parallele Keimzellensuche – ein von der Vorgehensweise ähnliches Verfahren – für beide Algorithmen umsetzbar ist, wurde auf die Möglichkeit verzichtet. Zusätzlich kann beim Aufteilen des Wachsens, insbesondere bei kleinen Strukturen, ein zu großer Overhead erzeugt werden, sodass die parallele Verarbeitung nicht schneller werden würde.

Kapitel 4

Dynamische Segmentierung

Die Anwendung von dynamischen Segmentierungsmethoden ist ein Themengebiet, welches eine große Anzahl von möglichen Algorithmen und Verfahren beinhaltet. In diesem Kapitel werden einige grundlegende Vorgehensweisen beschrieben werden, auf denen komplexere Segmentierungsmethoden aufbauen können. Einige Verfahren wurden dabei zu Testzwecken implementiert und anhand von Beispieldatensätzen (Metallschäume) auf ihre Leistungsfähigkeit hin überprüft.

Der Hauptgrund für nachfolgende Überlegungen liegt in dem Aufwand bei der manuellen Grenzwertbestimmung, wie sie bei der Binarisierung im vorherigen Kapitel benötigt wird. Die Schwellwerte muss der Benutzer unter Zuhilfenahme von Bildbearbeitungsprogrammen wie VGStudio Max oder ImageJ[1] durch Anpassung der Farbpalette ermitteln. Leider bietet dieses Verfahren keine exakte Bestimmung des Grenzwertes, so dass das Tomogramm mehrmals mit unterschiedlichen Schwellen binarisiert werden muss, um die optimalen Grenzen zu finden. Des Weiteren wird dieses Verfahren durch die subjektive Entscheidungsfindung des Benutzers bestimmt.

Unter der Zuhilfenahme von dynamischen Segmentierungsmethoden sollen diese Nachteile umgangen werden. Die Suche zielt verstärkt auf das Finden von Verfahren zur dynamischen Schwellwertbestimmung. Dabei kann ebenso eine semi-automatische Lösung berücksichtigt werden, bei der der Nutzer näherungsweise bestimmte Grenzen ggf. anpasst.

Die Begründung für diese Fokussierung ist auf zwei Aspekte zurückzuführen: Neben den Mitarbeitern der Arbeitsgruppe „Synchrotrontomographie“ wird die Datenauswertung von Tomogrammen auch von Gastwissenschaftlern durchgeführt, denen in möglichst kurzer Zeit die verschiedenen Programme erklärt werden müssen. Eine (semi-)automatische Schwellwertbestimmung bei der Binarisierung mindert diesen Aufwand erheblich. Der zweite Grund liegt in den Eigenschaften von tomographischen Aufnahmen (vgl. 2.2). Erste Versuche haben gezeigt, dass eine komplett eigenständige Segmentierung auf Grund des starken Rauschens und der großen Inhomogenität relativ schwer oder sogar nicht erreichbar ist.

[1] National Institute of Mental Health, USA

4.1 Filter

Bevor erste Vorgehensweisen zur dynamischen Segmentierung erläutert werden, sollen als Einführung zwei elementare Filtertypen vorgestellt werden: Der Hochpass- und der Tiefpassfilter.

Beide Filterarten werden auf die fourier-transformierten Bilddaten, also die Frequenzen, angewandt. Während der Hochpassfilter tiefe Frequenzen entfernt, verhält es sich bei dem Tiefpassfilter genau umgekehrt. Das Wechseln von der Ortsdarstellung in die Frequenzdarstellung eines Bildes wird durch die auf Bilder angepasste „Diskrete Fourier-Transformation“ (DFT) oder „Fast Fourier-Transformation“ (FFT) erreicht. Die Fourier-Tranformation beruht auf der Tatsache, dass Funktionen als eine Überlagerung von Schwingungen, im einfachsten Fall Sinus- und Kosinusschwingungen, aufgefasst werden können. (Näheres zur Fourier-Tranformation in [7] und [16].)

In der Frequenzdarstellung entsprechen große Graustufenänderungen hohen Frequenzen. Dies führt dazu, dass beim Anwenden eines Tiefpassfilters starke Änderungen unterdrückt bzw. eliminiert werden. Das Bild wird dementsprechend auf der einen Seite geglättet, aber auf der anderen Seite auch unscharf. Der Hochpassfilter hebt dagegen starke Änderung der Grauwerte in der Ortsdarstellung hervor, sodass dieser zur Kantendetektion benutzt werden kann. (Ausführliches dazu im Abschnitt 4.3)

Obwohl Hoch- und Tiefpassfilter frequenzspezifische Eigenschaften des Bildes ausnutzen, können diese auch in der Ortsdarstellung durchgeführt werden. Dazu wird Punkt für Punkt eine Maske über das Bild gelegt (im mathematischen Sinne wird eine Faltung ausgeführt [7]), welche die Filterfunktion $\triangle$ repräsentiert. Meist wird dafür in 2D-Abbildungen folgende allgemeine Maske verwendet [16]:

$$\triangle f(x,y) = \begin{pmatrix} f(x-1,y-1) & f(x,y-y) & f(x+1,y-1) \\ f(x-1,y) & f(x,y) & f(x+1,y) \\ f(x-1,y+1) & f(x,y+1) & f(x+1,y+1) \end{pmatrix}$$

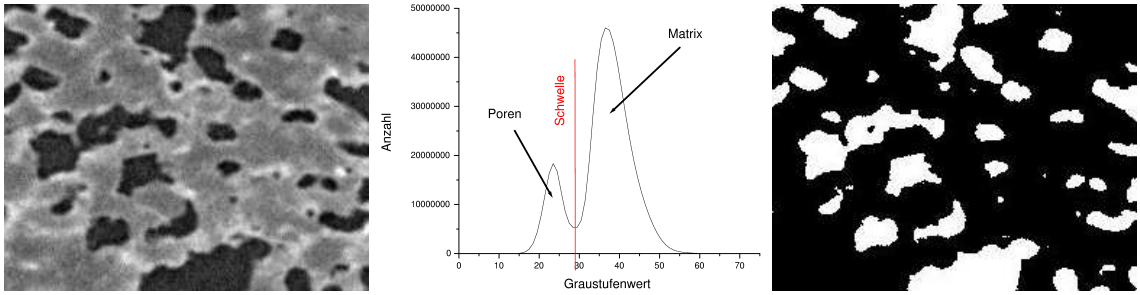
Die Größe der Maske ist jedoch variabel. Ihre Ausdehnung kann mit der in Abschnitt 3.1 beschriebenen Nachbarschaft verglichen werden

Als Beispiele für Tiefpassfilter sollen hier der Mittelwert und der Medianfilter erläutert werden. Ersterer berechnet den Durchschnittswert aller mit der Maske eingeschlossenen Pixel und ersetzt den mittleren Wert der Maske durch das Ergebnis. Beim Medianfilter hingegen werden alle Bildwerte die innerhalb der Maske liegen gespeichert und der Größe nach sortiert. Der mittlere Pixel der Maske bekommt dann den Wert, der in der Mitte der geordneten Bildpunkte liegt. Die Anwendung von Hochpassfiltern zur Rauschreduktion wird im Folgenden beschrieben.

4.2 Schwellwertbasierte Verfahren

Schwellwertbasierte Verfahren versuchen durch direktes Finden von Grenzwerten das Bild zu segmentieren. Die Vorgehensweise bei der nachfolgenden Binarisierung wurde bereits ausführlich in Kapitel 3 beschrieben.

Die meisten schwellwertbasierten Verfahren arbeiten anhand des Histogramms des Bilddatensatzes zur Ermittlung der Schwellwerte. Die einfachste Vorgehensweise ist dabei ähnlich dem manuellen Finden der Grenzen, wie es im Abschnitt 3.1 beschrieben worden ist. Ein Beispiel, welches einen Metallschaum darstellt, ist in Abbildung 4.1 gegeben.



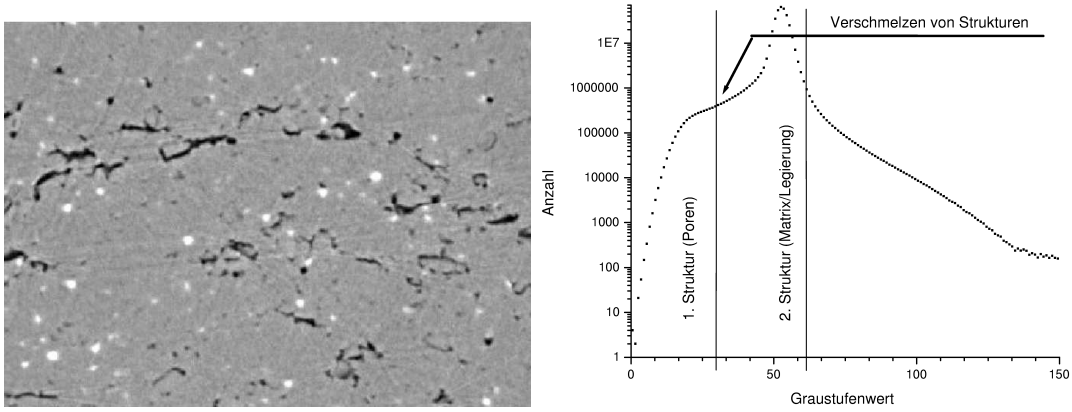
(a) Original (b) Histogramm (c) Binarisiert

Abbildung 4.1: Ermittlung von einfachen Schwellwerten mittels Histogramm: Die Grenzen werden durch Berechnung der Minima im Histogramm bestimmt. Im konkreten Beispiel (a) ist nur eine Grenze notwendig, da es sich um ein zweiphasiges Objekt handelt. Im Histogramm (b) sind die zwei Phasen (Matrix und Poren) durch die beiden Maxima gekennzeichnet. Die Grenze wird durch Ermittlung des lokalen Minimums zwischen den beiden Maxima bestimmt, sodass das Bild binarisiert werden kann (c).

Im Histogramm sind zwei lokale Maxima sichtbar, welche in ihren Graustufenwerten die Matrix (Material) bzw. die Poren repräsentieren. Die einfachste Möglichkeit zum Finden des Schwellwertes ist das Aufsuchen des Minimums zwischen den beiden Maxima. Die in diesem Tal vertretenen Graustufenwerte, repräsentieren den örtlichen Übergangsbereich zwischen der Matrix und den Poren im Bild. Dieses Minimum bietet sich zunächst als Grenzwert an. Die Vorgehensweise funktioniert allerdings nur, wenn die Matrix oder die Poren in allen Bildbereichen gleiche bzw. nur gering abweichende Graustufenwerte besitzen. Sollte dies nicht der Fall sein, kann es zu einer örtlich begrenzten Verschiebung der Schwelle in Richtung Matrix oder Poren kommen. Abhilfe schafft in diesem Fall die Aufteilung des Bildes in kleine Regionen, in denen jeweils das entsprechende Histogramm berechnet wird. Die daraus resultierenden Grenzwerte sind dann auf die zum Histogramm gehörende Region beschränkt.

Bei der Untersuchung von weiteren Tomogrammen wird jedoch schnell ersichtlich, dass eine klare Unterscheidung der einzelnen Strukturen innerhalb des Histogramms nur selten möglich ist. Den Grund hierfür stellt das Verschmelzen von Strukturen innerhalb des Histogramms dar (siehe Abbildung 4.2). Dies liegt in der starken Streuung (vgl. 2.2) der Graustufenwerte von einzelnen Strukturen begründet. Ein Ansatz zur Lösung des Problems stammt aus dem Jahr 1972. In ihm wird versucht, dass Histogramm als eine Summe von Normalverteilungen zu interpretieren. Jede Struktur im Bilddatensatz ist in ihren Graustufenwerten mit einer bestimmten Streuung (Varianz – die genormte Breite der Verteilungsfunktion) normalverteilt. Dabei wird jede Phase mit einer eigenen Verteilungskurve identifiziert. Addiert man die Verteilungskurven der einzelnen Strukturen, so ergibt sich das Histogramm über das Gesamtbild (siehe Abbildung 4.3). [16]

Bei den zu separierenden Tomogrammen kann im Gegensatz zur Theorie meist keine eindeutige Verteilung bestimmt werden. Dies liegt in der inhomogenen Graustufenverteilung



(a) Metallschaum (b) Histogramm

Abbildung 4.2: Verschmelzen von Strukturen im Histogramm: Im Histogramm des hier abgebildeten Metallschaums (Material: AlSi7) können die einzelnen Strukturen (Poren, Matrix und Treibmittel) nicht eindeutig erkannt werden, obwohl diese im Originalbild für das menschliche Auge eindeutig sichtbar sind. Auf Grund der starken Streuung innerhalb der Strukturen im Verhältnis zum Kontrast untereinander sind nicht alle Maxima eindeutig bestimmbar.

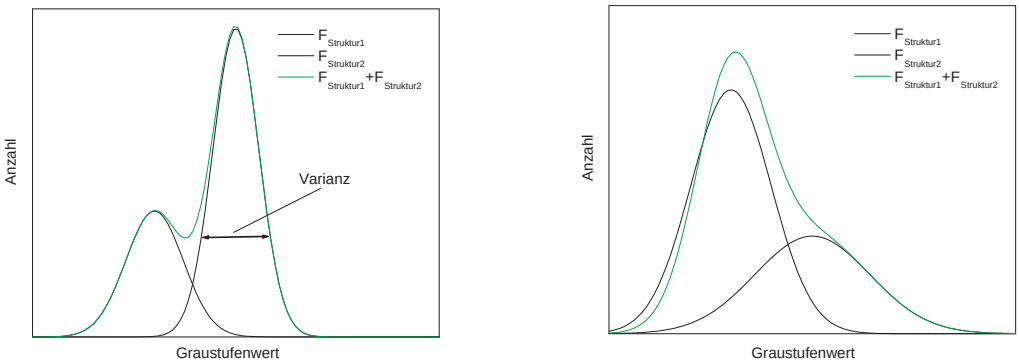
innerhalb der Strukturen begründet. Die Ursachen dafür werden in Abschnitt 2.2 „Bilddaten und -größe“ beschrieben. Selbst beim Vorliegen einer Normalverteilung fällt es daher schwer, die entsprechenden Verteilungsparameter zu bestimmen.

4.3 Ableitungsbasierte Verfahren

Neben den Segmentierungsmethoden, welche durch direktes Bestimmen von Schwellwerten die Binarisierung durchführen, gibt es weitere Möglichkeiten zur Separation von Bildern: die ableitungsbasierten Algorithmen. Diese versuchen nicht wie die schwellwertbasierten Methoden Bildbereiche in interne und externe Flächen bzw. Volumen zu zerlegen. Bei diesen Verfahren wird versucht über die Grenzen (Kanten, Konturen) des Objektes dieses zu separieren.

Wie bereits zuvor beschrieben, entsprechen Kanten in der Frequenzdarstellung hochfrequenten Anteilen. Diese müssen von den Kantendetektoren erkannt werden. Das Problem dabei ist, dass Rauschen ebenfalls durch hohe Frequenzen repräsentiert wird. Aus diesem Grund wird vor der Anwendung eines Kantenfilters das Bild geglättet. Dies geschieht durch einen Tiefpassfilter, wie er in Abschnitt 4.1 beschrieben ist. Im Anschluss an die Kantenerkennung wird das entstandene Bild binarisiert, sodass die Kanten eindeutig bestimmbar sind.

Im Folgenden sollen einige Kantendetektoren vorgestellt werden. Zur einfacheren Darstellung sind die folgenden Definitionen auf den 2D-Raum begrenzt. Eine Umsetzung für den 3D-Raum erfolgt in den meisten Fällen durch einfache Erweiterung um eine Dimension.



(a) Verteilung 1

(b) Verteilung 2

Abbildung 4.3: Histogramm durch Addition von Verteilungskurven der Strukturen: Ein Ansatz zur Bestimmung des Schwellwertes ist die Aufteilung des Histogramms in Normalverteilungen der einzelnen Strukturen. Damit kann das Problem der Verschmelzung von Strukturen im Histogramm durch starke Streuung (große Varianz) umgangen werden (vgl. (b)).

4.3.1 Ableitung erster Ordnung

Eine Möglichkeit zur Kantenerkennung ist die Erstellung des Ableitungsbildes eines Tomogramms. Die erste Ableitung hat bei großer Veränderung von Graustufenwerten im Bild ein lokales Maximum, welches zur Separation von Kanten verwendet werden kann (vgl. Bild 4.5). Berechnet wird die Veränderung durch partielle Ableitung der Bildfunktion in x- und y-Richtung. Multipliziert mit dem Eigenvektor in die jeweilige Richtung ergibt sich der Gradient eines Bildes:

$$\vec{\nabla} \cdot f(x,y) = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} \cdot \begin{pmatrix} f_x \\ f_y \end{pmatrix} = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{pmatrix}$$

Der Gradient zeigt in die Richtung des maximalen positiven Anstiegs. Der Betrag entspricht dem Wert des Anstieges in jedem Bildpunkt:

$$|\vec{\nabla} \cdot f(x,y)| = \sqrt{\left(\frac{\partial f(x,y)}{\partial x}\right)^2 + \left(\frac{\partial f(x,y)}{\partial y}\right)^2}$$

Mit der Berechnung des Betrages gehen die Richtungsinformationen verloren. Diese sind jedoch in diesem Fall nicht von Interesse. Aus diesem Grund kann der Anstieg zur schnelleren Berechnung annäherungsweise wie folgt bestimmt werden:

$$|\vec{\nabla} \cdot f(x,y)| = max\left[\left|\frac{\partial f(x,y)}{\partial x}\right|, \left|\frac{\partial f(x,y)}{\partial y}\right|\right],$$

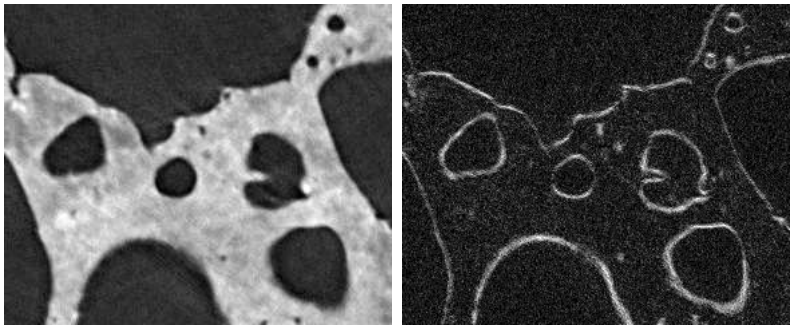
wobei sich $\frac{\partial f(x,y)}{\partial x}$ aus der Differenz

$$\triangle_x f(x,y) = f(x,y) - f(x-1,y) \qquad \text{(Rückwärtsgradient)},$$

$$\triangle_x f(x,y) = f(x+1,y) - f(x,y) \qquad \text{(Vorwärtsgradient) oder}$$

$$\triangle_x f(x,y) = \tfrac{f(x+1,y)-f(x-1,y)}{2} \qquad \text{(symmetrischer Gradient)}$$

berechnet. (Analog folgt die Definition für $\frac{\partial f(x,y)}{\partial y}$.) [7], [16] Das Ergebnis eines solchen Gradientenbildes ist in Abbildung 4.4 zu sehen.



(a) Original (b) Gradient

Abbildung 4.4: Kantenerkennung mittels Betragsgradienten: Das Gradientenbild in (b) wurden mittels Vorwärtsgradienten vom Originalbild (a) berechnet.

Die Berechnung eines Gradientenbildes kann wie bei der Anwendung von Filtern (siehe 4.1) durch Verwendung einer Maske berechnet werden. Setzt man den Rückwärtsgradienten in eine Maske um, so erhält man folgendes Ergebnis:

$$\triangle_x f(x,y) = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \triangle_y f(x,y) = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Nachteilig bei der Verwendung des Rück- oder Vorwärtsgradienten ist, dass das eigentliche Bild um einen halben Pixel verschoben wird. Um dies zu vermeiden, empfiehlt sich die Nutzung des symmetrischen Gradienten: [16]

$$\triangle_x f(x,y) = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad \triangle_y f(x,y) = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Der Roberts-Operator Der Roberts-Operator funktioniert ähnlich dem einfachen Differenz-Operator. Der Unterschied besteht darin, dass dieser die Differenz nicht in horizontaler und vertikaler Richtung bildet, sondern die Diagonalen verwendet. Aus diesem Grund wird er auch als „Roberts-Cross“ bezeichnet.

$$\triangle_x f(x,y) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \qquad \triangle_y f(x,y) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

Sowohl der einfache Differenz- als auch der Roberts-Operator verwenden keine Filterung. Deshalb sind sie relativ anfällig gegenüber Störungen und Rauschen. Aus diesem Grund werden diese in der Praxis relativ selten eingesetzt. [16]

Der Prewitt-Operator Der Prewitt-Operator enthält eine Glättung durch Mittelwertbildung senkrecht zur Differenzierungsrichtung und durch die Benutzung des symmetrischen Gradienten: [16]

$$\triangle_x f(x,y) = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \qquad \triangle_y f(x,y) = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Der Sobel-Operator Der Sobel-Operator verwendet bei der Glättung eine der Gauß-Funktion angenäherte Verteilung. Dabei ist zu beachten, dass die Approximationsgüte mit der Vergrößerung der Maske steigt. Für den Fall einer 3x3-Matrix sieht diese folgender Maßen aus: [16]

$$\triangle_x f(x,y) = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad \triangle_y f(x,y) = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

4.3.2 Ableitung zweiter Ordnung

Neben der Anwendung von approximierten Filtern der Ableitung erster Ordnung, kann auch ein Verfahren der zweiten Ableitung benutzt werden. Diesen Filter stellt der Laplace-Operator dar.

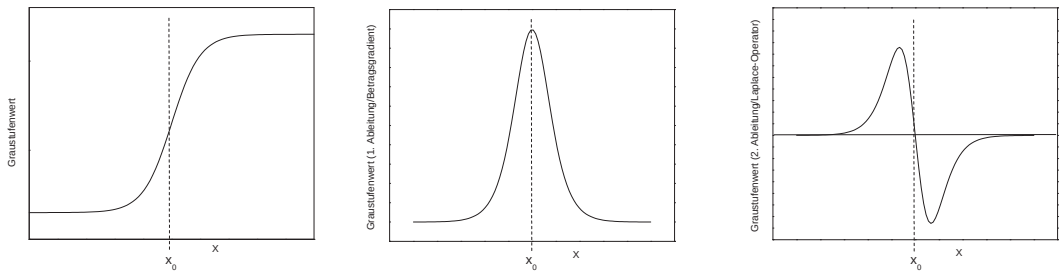
In der zweiten Ableitung besitzen starke Graustufenänderungen im Bild einen Nulldurchgang (vgl. Bild 4.5). Diese können bei der anschließenden Binarisierung durch Finden von Vorzeichenwechseln bestimmt werden. Alternativ besteht auch die Möglichkeit das Bild der zweiten Ableitung durch Betragsbildung und dann, wie bei den anderen Operatoren, mittels einfachen Schwellwert zu binarisieren.

Im kontinuierlichen Fall sieht der Laplace-Operator, angewandt auf eine Funktion f, wie folgt aus:

$$\vec{\nabla}^2 f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2}$$

Die Annäherung im diskreten Fall sowie die daraus resultierende Filtermaske kann folgendermaßen dargestellt werden:

$$\begin{aligned} \frac{\partial^2 f(x,y)}{\partial x^2} &\approx \frac{\partial (f(x+1,y) - f(x,y))}{\partial x} \\ &\approx (f(x+1,y) - f(x,y)) - (f(x,y) - f(x-1,y)) \\ &= f(x-1,y) - 2f(x,y) + f(x+1,y) \end{aligned}$$



(a) Ortsfunktion in X-Richtung (b) 1. Ableitung der Ortsfunktion in x-Richtung (Gradient) (c) 2. Ableitung der Ortsfunktion in x-Richtung (Laplace-Operator)

Abbildung 4.5: Ortsfunktion in X-Richtung mit 1. und 2. Ableitung: Die Kante bei x_0 ist in der Ortsfunktion in der jeweiligen Richtung durch eine starke Graustufenänderung geprägt (a). In der 1. Ableitung (Betragsgradient) befindet sich dementsprechend ein Maximum bei x_0 (b) und in der 2. Ableitung (Laplace-Operator) ein Nulldurchgang (c).

$$\triangle_{x,y} f(x,y) \quad = \quad \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

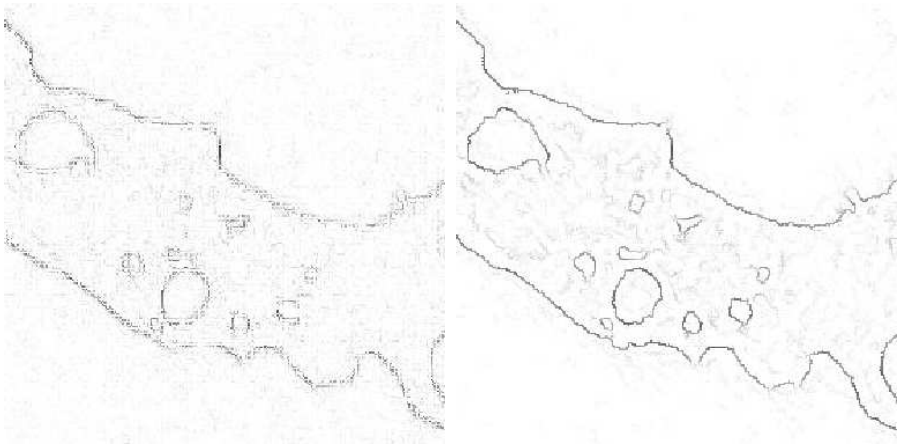
Neben dieser Maske gibt es weitere Möglichkeiten den Laplace-Operator als Filtermaske zu diskretisieren, die hier aber nicht weiter aufgeführt werden sollen. [7], [16]

Wie im Bild 4.6 zu erkennen ist, reagiert der Laplace-Filter sehr stark auf Störungen und Rauschen. Aus diesem Grund wird in der Praxis meist ein Gaußfilter (näheres zum Gaußfilter in [7]) davor angewandt. Beide Operationen können zum sogenannten „Laplacian-of-Gaussian“ Filter (LoG) zusammengefasst werden.

Das Ableitungsbild wird, wie bereits am Anfang des Abschnitts beschrieben, unter der Vorgabe eines Schwellwertes binarisiert. In [16] wird angegeben, dass ein allgemeingültiger Schwellwert unter Verwendung einer begrenzten Anzahl von Gradienten (Weglassen von Extremwerten) erhalten werden kann. Bei der Binarisierung von Tomogrammen konnte leider keine Verallgemeinerung hinsichtlich der Schwellwertwahl gefunden werden. Zwar liegen die Grenzen (prozentualer Wert in Abhängigkeit des größten Gradienten) bei der Binarisierung der drei Beispiele in Abbildung 4.7 relativ dicht zusammen. Es ist jedoch unklar, ob eine Generalisierung für alle Tomogramme getroffen werden können, da hier lediglich eine sehr kleine Auswahl getroffen wurde.

Die in Abbildung 4.7 zu sehenden Beispiele von tomographischen Aufnahmen zeigen jedoch, dass mit der Binarisierung ein weiteres Problem verbunden ist: Während einige Kantenlinien mehrere Pixel breit sind, gibt es an einigen Stellen Unterbrechungen der Kanten. Zum Segmentieren der verschiedenen Strukturen können diese Ergebnisse nicht verwendet werden. Um dieses Problem zu vermeiden, müssen alternative Vorgehensweisen – die Beschreibung folgt im nächsten Abschnitt – betrachtet werden.

Obwohl diese Gradientenbilder nicht zur direkten Segmentierung genutzt werden können, besteht die Möglichkeit, mit darin enthaltenen Informationen, annäherungsweise einen Grenz-

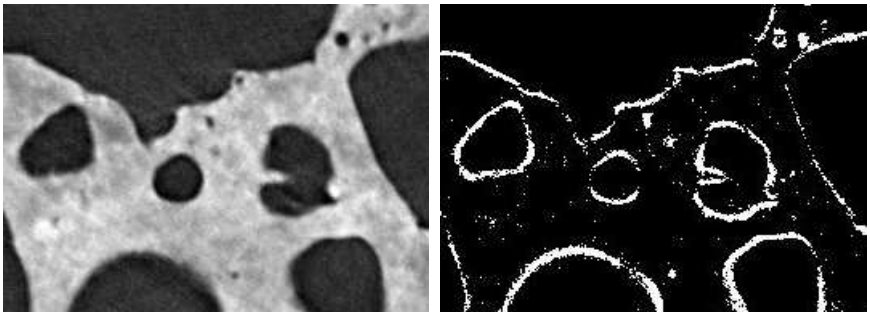


(a) Kantendetektion mit Laplace-Operator

(b) Kantendetektion mit Glättung und Laplace-Operator

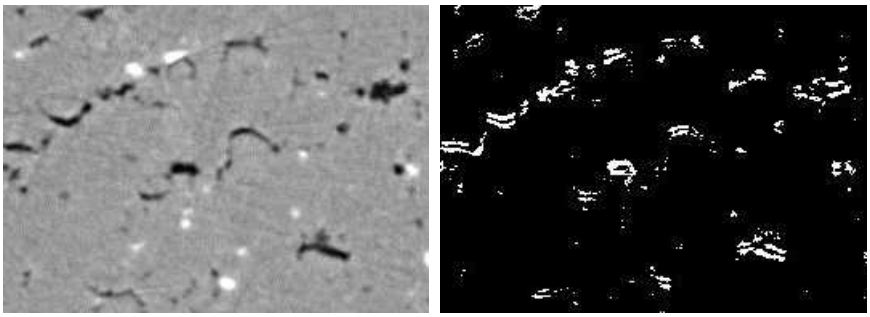
Abbildung 4.6: Kantenerkennung mit Laplace-Operator: Der Laplace-Operator ist relativ störanfällig gegenüber Rauschen (a). Aus diesem Grund wird das Eingangsbild vorher geglättet (b). Die Erstellung der Bilder erfolgte mit Gimp (GNU Image Manipulation Program)

wert zur Binarisierung mit einer schwellwertbasierten Methode zu finden. Die Idee bezieht sich auf das Vergleichen des binarisierten Gradientenbildes mit dem Originalbild. Dabei wird der Durchschnitt über diejenigen Graustufenwerte gebildet, deren Gradientenwert nach der Binarisierung positiv erkannt wird. Die Vorgehensweise funktioniert jedoch lediglich bei zweiphasigen Objekten, dass heißt, Proben, die nur zwei unterschiedliche Strukturen aufweisen. Beim Vorhandensein einer weiteren Phase werden auch diejenigen Kanten mit betrachtet, die nicht an der Binarisierungsgrenze der entsprechenden Struktur liegen.



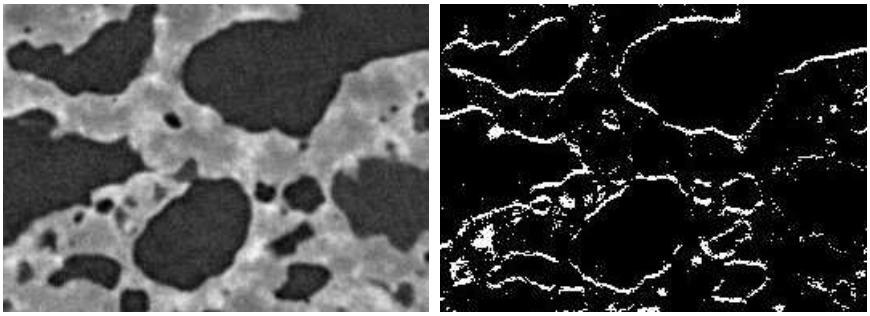
(a) Beispiel 1

(b) Binarisierungsgrenze bei 10% des größten Gradienten



(c) Beispiel 2

(d) Binarisierungsgrenze bei 11% des größten Gradienten



(e) Beispiel 3

(f) Binarisierungsgrenze bei 12% des größten Gradienten

Abbildung 4.7: Binarisierung von Gradientenbildern: Obwohl die prozentualen Werte für die Schwellwerte bei den Beispielen relativ dicht zusammen liegen, kann auf Grund der Vielzahl von tomographischen Aufnahmen keine Verallgemeinerung zur Findung des optimalen Schwellwertes gemacht werden.

4.4 Der Canny-Operator

Neben den bisher beschriebenen einfachen, lokalen Differenzoperationen existieren weitere Algorithmen zur Kantenerkennung, die sogenannten optimalen Operatoren. Im Gegensatz zu ersteren wird bei diesen versucht, zusätzlich Vorstellungen über die Bildeigenschaften im Hinblick auf Kantenmodelle und ein Maß für Fehler und Genauigkeiten bei der Detektion zu gewinnen. [16]

Weit verbreitete Operatoren dieser Art sind der Canny- und Marr-Hildreth-Operator. Im Folgenden soll der Canny-Operator näher beschrieben werden, da dieser interessante Verfahren enthält, die auch auf einfache Kantendetektoren angewandt werden können.

Der Canny-Operator benutzt zur Glättung und Kantendetektion die Richtungsableitung der Gauß-Funktion in x- und y-Richtung. Dabei können je nach Wahl der Varianz mehr Kanten erkannt oder übergangen werden. Anschließend erfolgt eine Kantenverfolgung nach dem „Non-Maxima-Supression“-Verfahren unter Benutzung einer Schwellwert-Hysterese. Mit der „Non-Maxima-Supression“-Methode sollen diejenigen Punkte entfernt werden, die nicht zu einer Kante gehören. Dazu werden die Nachbarn in Richtung und in entgegengesetzter Richtung des Gradienten untersucht. Besitzen diese einen höheren Gradientwert, wird der untersuchte Punkt von den weiteren Betrachtungen ausgeschlossen.

Im Anschluss erfolgt eine Kantenverfolgung mit einer Schwellwert-Hysterese. Dieses Verfahren wurde auch bei der parametrisierten Segmentierung (Kapitel 3) verwendet. In diesem Fall werden alle Punkte, die oberhalb des ersten Grenzwertes liegen, als sichere Konturen betrachtet. Von diesen aus werden weitere Pixel gesucht, die über der zweiten Schwelle liegen, um somit die Kanten zu schließen. Der erste Grenzwert ist dementsprechend größer als der zweite. Diese Vorgehensweise stellt eine weitere Filterung dar, da hier hohe, durch Rauschen bedingte Gradientenwerte (zwischen dem ersten und dem zweiten Schwellwert), die nicht zu einer Kante gehören, ignoriert werden.

Die beiden Verfahren, „Non-Maxima-Supression“ und Schwellwert-Hysterese, zur Kantennachbearbeitung können auch bei einfachen Operatoren, soweit diese zum Betrag auch die Richtung des Gradienten bestimmen, eingesetzt werden. Ein großer Nachteil dieser Vorgehensweise ist, dass – mit Blick auf die Vereinfachung für den Anwender – wieder mehrere Parameter zur Segmentierung vorgegeben werden müssen. Für eine Umsetzung dieser Verfahren müssten zusätzlich Möglichkeiten zur Eingrenzung der weiteren Parameter gefunden werden.

Neben dem Canny-Operator soll an dieser Stelle noch eine letzte Alternative erwähnt werden. Dazu müssen bei der Binarisierung des Gradientenbildes, welches mit einem beliebigen, oben erwähnten Verfahren erstellt wurde, alle Gradientenwerte, die größer als Null sind, mit binarisiert werden. Dadurch entsteht ein Bild, welches relativ breite Kanten, aber ein starkes Binarisierungsrauschen enthält. Letzteres kann durch Öffnen des Vordergrundes (Kanten) minimiert werden. Danach werden die Kanten mittels „Thinning“-Algorithmen auf ein Pixel breite Linien verdünnt. Weitere Details zu diesen Verfahren sind in [20] beschrieben.

Kapitel 5

Ergebnisse und Ausblick

5.1 Parametrisierte Segmentierung

Die Binarisierung von 3D-Bildern unter Angabe der Schwellwerte, wie sie in Kapitel 3 beschrieben ist, wurde erfolgreich in der zu erstellenden Bibliothek umgesetzt. Zusätzlich wurde ein Beispielprogramm implementiert, welches als Schnittstelle zum Anwender dient. Dieses benutzt die Bibliotheksfunktionen und die von dem Anwender übergegebenen Parameter zur Segmentierung von Tomogrammen.

Die Segmentierungsfunktion der Bibliothek liefert rauscharme, binäre Bilder. Um dieses zu erreichen, wurde ein Verfahren verwendet, welches zur Separation eine Schwellwerthysterese mit einem Regionenwachstum verknüpft. Unter der Angabe zweier Schwellwertbereiche werden in dem zu segmentierenden Tomogramm Voxel gesucht, die im inneren Schwellwertbereich liegen. Diese werden als sichere Voxel (Keimzellen) betrachtet, das heißt sie gehören auf jeden Fall zur zu separierenden Struktur. Von diesen aus wird in einer frei wählbaren Nachbarschaft sukzessive nach weiteren Bildpunkten gesucht, die innerhalb des äußeren Schwellwertbereiches liegen. Die Vereinigung dieser Voxel ergibt die zu separierende Struktur.

Die Binarisierung wurde mit größtmöglicher Effektivität in Hinblick auf Speicherverbrauch und Geschwindigkeit realisiert. Zum Wachsen um die Keimzellen wurde eine iterative Implementation des „Boundary Fill“-Algorithmus verwendet. Die Steuerung des Ablaufs des Algorithmus erfolgt durch verschiedene Kodierungen im zu untersuchenden Tomogramm. Deshalb wird zur Verarbeiten eines 3D-Bildes lediglich eine dem Datensatz äquivalente Größe an Hauptspeicher benötigt. Eine geringere Speicherbelastung ist bei dem verwendeten Algorithmus ohne extreme Verlängerung der Rechenzeit nicht möglich. Neben dem reinen Binarisierungsalgorithmus wurde in der Bibliothek auch eine Parallelisierung zur schnelleren Verarbeitung implementiert.

5.2 Dynamische Segmentierung – Ein Ausblick

Im Gegensatz zur parametrisierten Separation muss die dynamische Segmentierung noch in der Bibliothek umgesetzt werden. Die Aufgabe in diesem Teil der Diplomarbeit bestand im Finden erster möglicher Verfahren. Diese sind im letzten Kapitel beschrieben.

Ziel dieser Überlegungen ist es, dem Anwender – insbesondere unerfahrenen Gastwissenschaftlern – die Segmentierung zu erleichtern. Dies soll durch eine dynamische Segmentierung erreicht werden. Als Alternative besteht hier auch die Möglichkeit einer Semi-Automatisierung bei der dem Benutzer näherungsweise die Schwellwerte vorgegeben werden.

Bei der Suche nach neuen Möglichkeiten zur automatisierten Segmentierung musste festgestellt werden, dass durch die Erweiterungen oder Verfeinerungen bestehender Algorithmen sich meist zusätzliche Parameter ergeben. Da dies jedoch genau entgegengesetzt der eigentlichen Bestrebung läuft, müssen Verfahren gefunden werden, auch diese dynamisch zu bestimmen oder andere Algorithmen entworfen werden. Die zuletzt beschriebene Vorgehensweise – das Ausdünnen von breiten Kantenlinien – kann zu einer guten Möglichkeit der (semi-)automatischen Segmentierung führen. Dazu müssen verschiedene „Thinning“- und Skeletierungsalgorithmen auf ihre Effektivität überprüft werden.

Anhang A

Programme

A.1 64-Bit-Test

Listing A.1: Programm zur Darstellung Unterschieden bei 32- und 64-Bit Architekturen

```
1  #include <stdlib.h>
2
3  struct _xy
4  {
5          char cNumber;
6          struct _xy *pNext;
7  };
8
9  struct _ab
10 {
11         char cNumber;
12         unsigned pNext;
13 };
14
15 typedef struct _xy xy;
16 typedef struct _ab ab;
17
18 int main(int argc, int *argv[])
19 {
20         xy structXY;
21         ab structAB;
22
23         unsigned long l=0xFFFFFFFF,l1;
24         xy *AnfangXY;
25         ab *AnfangAB;
26         ab *pAB;
27
28         /*Bitshifting*/
29         printf("Vor Bitshifting:\t%lX\n",l);
30         l=l<<8;
31         printf("Zwischendurch:\t\t%lX\n",l);
32         l=l>>8;
33         printf("Nach Bitshifting:\t%lX\n\n",l);
34
35         /*Komplement*/
36         l=0xFFFFFFFF;
```

```
37          printf("Original:\t%lX\n",l);
38          printf("Komplement:\t%lX\n\n",~l);
39
40          /*Typecast*/
41          AnfangXY = (xy*)malloc(sizeof(xy));
42          (*AnfangXY).cNumber = 15;
43          (*AnfangXY).pNext = (xy*)malloc(sizeof(xy));
44          (*(*AnfangXY).pNext).cNumber = 17;
45          AnfangAB = (ab*)malloc(sizeof(ab));
46          AnfangAB = (ab*)AnfangXY;
47          printf("%ld\n",(*AnfangAB).cNumber);
48          pAB = (ab*)(*AnfangAB).pNext;
49          printf("%d\n",(*pAB).cNumber);
50
51
52  }
```

Anhang B

Quellcode

B.1 „Boundary Fill“-Algorithmus

Listing B.1: Umsetzung des Boundary Fill-Algortihmus

```
/***************************************************************************
                          boundaryfill.c  -  description
                             -------------------
    begin                : Mo Mai 17 2004
    copyright            : (C) 2004 by Stefan Kirste
    email                : kirste@hmi.de
 ***************************************************************************/

#include "boundaryfill.h"
#include <math.h>


void boundaryfill(unsigned char *tomogram, int dimx, int dimy, int dimzstart, int dimzend,
                  unsigned long tomostart, unsigned long tomoend, long initialpoint,
                  unsigned char lowerouterboundary, unsigned char upperouterboundary,
                  int neighbourhood1darray[], int **neighbourhood3darray, unsigned char neighbours,
                  unsigned char minwa, unsigned char maxwa)
{

  /*Richtungskodierung (Startwert wird auf den untersten Wert des Arbeitsbereiches gesetzt)*/
  int direction=minwa;
  /*Zu untersuchender Voxel als Referenz im 1D-Array und den dazugehörigen 3D-Werten*/
  long ic;
  int xc,yc,zc;
  /*3D-Werte für Keimzelle berechnen*/
  int initialpointx=initialpoint%(dimx*dimy)%dimx;
  int initialpointy=initialpoint%(dimx*dimy)/dimx;
  int initialpointz=initialpoint/(dimx*dimy);

  /*Keimzelle kodieren*/
```

```
31    tomogram[initialpoint]=maxwa-1;
32
33    while(1)
34    /*Überprüfen, ob schon alle Nacharn untersucht wurden*/
35      if (direction!=maxwa-1)
36      {
37        /*Zu untersuchenden Punkt durch Nachbarschaftsbeziehung berechnen*/
38        ic=initialpoint+neighbourhood1darray[direction-minwa];
39        xc=initialpointx+neighbourhood3darray[direction-minwa][0];
40        yc=initialpointy+neighbourhood3darray[direction-minwa][1];
41        zc=initialpointz+neighbourhood3darray[direction-minwa][2];
42
43        /*Prüfen, ob der zu untersuchende Punkt im Bild ist,  zwischen den
              beiden äußeren */
44        /*Grenzen liegt und noch nicht untersucht wurde*/
45        if ( (xc>=0) && (xc<dimx) && (yc>=0) && (yc<dimy) &&
46             (zc>=dimzstart) && (zc<dimzend) &&
47             (tomogram[ic] >= lowerouterboundary) && (tomogram[ic] <=
                 upperouterboundary) &&
48             ((tomogram[ic] < minwa) || (tomogram[ic] > maxwa)) )
49        {
50          /*Zu untersuchender Punkt wird zur neuen Keimzelle*/
51          initialpoint=ic;
52          initialpointx=xc; initialpointy=yc; initialpointz=zc;
53          /*Merken aus welcher Richtung man gekommen ist*/
54          tomogram[initialpoint]=direction;
55          /*Nachbarschaftszähler auf den ersten Wert zurücksetzen*/
56          direction=minwa;
57        }
58        else
59          /*Richtungszähler erhöhen (Nächsten Nachbarn untersuchen)*/
60          direction++;
61      }
62      else
63        /*Überprüfen, ob die Keimzelle wieder erreicht wurde*/
64        if (tomogram[initialpoint]==maxwa-1)
65        {
66          /*Voxel positiv binarisieren*/
67          tomogram[initialpoint]=maxwa;
68          /*Funktion beenden*/
69          return;
70        }
71        else
72        {
73          /*Richtung, aus der man gekommen ist, sichern*/
74          direction=++tomogram[initialpoint];
75          /*Voxel positiv binarisieren*/
76          tomogram[initialpoint]=maxwa;
77          /*Alte, zum Teil bereits untersuchte, Keimzelle neu festlegen*/
78          initialpoint-=neighbourhood1darray[direction-minwa-1];
79          initialpointx-=neighbourhood3darray[direction-minwa-1][0];
80          initialpointy-=neighbourhood3darray[direction-minwa-1][1];
81          initialpointz-=neighbourhood3darray[direction-minwa-1][2];
82        }
83  }
```

B.2 Parallele Keimzellensuche

Listing B.2: Umsetzung der parallelen Keimzellensuche mit anschließenden Wachsen

```
1  /***************************************************************************
2                            binarization.c  -  description
3                               -------------------
4      begin                : Di Mai 18 2004
5      copyright            : (C) 2004 by Stefan Kirste
6      email                : kirste@hmi.de
7   ***************************************************************************/
8
9  #include "binarization.h"
10 #include "neighbourhood.h"
11 #include "boundaryfill.h"
12 #include "fill.h"
13 #include <pthread.h>
14 #include <stdlib.h>
15
16
17 struct __parametersBoundaryFill
18 {
19   unsigned char *tomogram;
20   unsigned char lowerinnerboundary;
21   unsigned char upperinnerboundary;
22   unsigned char lowerouterboundary;
23   unsigned char upperouterboundary;
24   int *neighbourhood;
25   unsigned char neighbours;
26   unsigned char minwa;
27   unsigned char maxwa;
28   long tomostart;
29   long tomoend;
30 };
31
32 typedef struct __parametersBoundaryFill parametersBoundaryFill;
33
34
35 void BoundaryFill(void* params)
36 {
37   long i;
38   long ts;
39   long te;
40   unsigned char *tomogram;
41   unsigned char lowerinnerboundary, upperinnerboundary;
42   unsigned char maxwa;
43   parametersBoundaryFill *p = (parametersBoundaryFill*)params;
44
45   lowerinnerboundary=p->lowerinnerboundary;
46   upperinnerboundary=p->upperinnerboundary;
47   tomogram=p->tomogram;
48   maxwa=p->maxwa;
49   ts=p->tomostart;
50   te=p->tomoend;
51
52   /*vorgegebenen Bereich nach Keimzellen durchsuchen*/
53   for (i=ts; i<te; i++)
54   {
```

```
55      if (tomogram[i]>=lowerinnerboundary && tomogram[i]<=upperinnerboundary &&
56          tomogram[i]!=maxwa/*Codierung für bereits binarisiertes Voxel*/)
57        boundaryfill(tomogram, ts, te, i,
58                     p->lowerouterboundary, p->upperouterboundary,
59                     p->neighbourhood, p->neighbours, p->minwa, p->maxwa);
60    }
61  }
62
63
64  int binarizeBoundaryFill(unsigned char *tomogram,
65                           unsigned int x, unsigned int y, unsigned int z,
66                           unsigned char lowerouterboundary, unsigned char
                                 lowerinnerboundary,
67                           unsigned char upperinnerboundary, unsigned char
                                 upperouterboundary,
68                           int neighbourhood[], unsigned char neighbours, char
                                 processors)
69  {
70    unsigned long size = x * y * z;
71    unsigned char minwa, maxwa; /*Working Area*/
72    long i,j;
73    pthread_t *thread;
74    parametersBoundaryFill paramsBF;
75
76          .
77          .
78          .
79
80    /*Binarisieren mit "Boundary Fill"-Algorthmus*/
81
82    /*Übergabeparameter zuweisen*/
83    thread=(pthread_t*)malloc(sizeof(pthread_t)*processors);
84    paramsBF.tomogram=tomogram;
85    paramsBF.lowerinnerboundary=lowerinnerboundary;
86    paramsBF.upperinnerboundary=upperinnerboundary;
87    paramsBF.lowerouterboundary=lowerouterboundary;
88    paramsBF.upperouterboundary=upperouterboundary;
89    paramsBF.neighbourhood=neighbourhood;
90    paramsBF.neighbours=neighbours;
91    paramsBF.minwa=minwa;
92    paramsBF.maxwa=maxwa;
93    /*für jeden Thread Tomogram in entsprechende Bereiche zerlegen*/
94    for(i=0; i<processors; i++)
95    {
96      paramsBF.tomostart=(z/processors)*x*y*i;
97      if (i==processors-1) paramsBF.tomoend=size;
98      else paramsBF.tomoend=(z/processors)*x*y*(i+1);
99      if(pthread_create(thread+i, NULL, (void *)&BoundaryFill, &paramsBF))
100       perror("binarization");
101   }
102   /*Schnitteben untersuchen und ggf. "Boundary Fill"-Algorithmus global laufen
          lassen*/
103   for(i=0; i<processors; i++)
104     pthread_join(thread[i], NULL);
105   for (i=0; i<processors-1; i++)
106   {
107      for ( j=(z/processors+i)*x*y; j<(z/processors)*x*y+(x*y); j++)
```

```
108         if (tomogram[j] == maxwa)
109           boundaryfill(tomogram, 0, size, j,
110                        lowerouterboundary, upperouterboundary,
111                        neighbourhood, neighbours, minwa, maxwa);
112       for ( j=(z/processors+i)*x*y-1; j>(z/processors)*x*y-(x*y)-1; j--)
113         if (tomogram[j] == maxwa)
114           boundaryfill(tomogram, 0, size, j,
115                        lowerouterboundary, upperouterboundary,
116                        neighbourhood, neighbours, minwa, maxwa);
117     }
118
119           .
120           .
121           .
122
123 }
```

Anhang C

Testrechner

Typ	Prozessoren	Betriebssystem	Bemerkung
Delta computer Product GmbH, Hochleistungsserver Dual Pentium Xeon	Dual Xeon 2400Hz	S.u.S.E. 9.0	Arbeitsrechner der Arbeitsgruppe „Synchrotron-Tomographie“
LT Memory GmbH, Lentio PC Intel Xeon 3 GHz	Dual Xeon 3000Hz	S.u.S.E. 9.1	Arbeitsrechner der Arbeitsgruppe „Synchrotron-Tomographie“
Compaq SC45 Compute-Cluster	8 Knoten mit 4x1001MHz EV68 Alpha-CPUs	Tru64 Unix 5.1B	HMI-Rechner-Cluster für parallele High-Performance-Anwendungen
SUN-Fire-480R	2 UltraSPARC-III CPUs, je 900 MHz	Solaris 9	

Tabelle C.1: Testrechner

Literaturverzeichnis

[1] Banhart, J. und A. Haibel: *Metallschäume Produktion – Charakterisierung – Anwendung oder von der Mikrostruktur zu Anwendungen.* In: *37. Metallographie-Tagung mit Austellung.* Deutsche Gesellschaft für Materialkunde, September 2003.

[2] *Berliner Elektronenspeicherring - Gesellschaft für Synchrotronstarhlung m.b.H.*, Dezember 2002. Präsentationsheft.

[3] Brinkmann, Marcus: *Gemeinsame Quellen - Portibles Programmieren mit GNU Autoconf und Automake.* c't, Heft 21:208, 2003.

[4] Bräunl, Thomas, Stefan Feyrer, Wolfgang Rapf und Michael Reinhardt: *Parallele Bildverarbeitung.* Addision-Wesley (Deutschland) GmbH, 1995.

[5] Canny, John Francis: *Finding Edges and Lines in Images.* Technischer Bericht, MIT Artificial Intelligence Labratory, 1983.

[6] Canny, John Francis: *A Computational Approach to Edge Detection.* Technischer Bericht, MIT Artificial Intelligence Labratory, 1986.

[7] Castleman, Kenneth R.: *Digital Image Processing.* Prentice Hall, Inc., Upper Saddle River, NJ 07458, 1996.

[8] Gerthsen, Christian und Helmut Vogel: *Gerthsen Physik.* Springer, Heidelberg, 1997.

[9] Kalender, Willi A.: *Computed Tomography.* Publicis MCD Verlag, 2000.

[10] Kernigham und Ritchie: *Programmieren in C.* Carl Hanser Verlag München Wien, 1983.

[11] Lin, Yingjian: *Morphologische Bildverarbeitung.* Technischer Bericht, Universität Ulm, 2003.

[12] Mark Mitchell, Jeffrey Oldham, Alex Samuel: *Advanced Linux Programming.* New Riders Publishing, 201 West 103rd Street, Indianapolis, Indiana 46290, 1. Auflage, Juni 2001.

[13] *Metallschaum - eine Entwicklung aus der Grundlagenforschung.* Flyer, Hahn-Meitner-Institut (Abteilung SF3).

[14] Rack, A., B. Matijasevic A. Haibel, A. Bütow und J. Banhart: *Characterization of Metal Foams with Synchrotron Tomography and 3D Image Analysis.* In: *Proceedings of the 16. World Conference on Nondestructive Testing (WCNDT)*, Montreal, Canada, August/September 2004.

[15] Schildt, Herbert: *C++ ENT-PACKT.* 1. Auflage. mitp-Verlag, ein Geschäftsbereich der verlag moderne industrie Buch AG & CO.KG, Landsberg, 2001.

[16] Steinbrecher, Dr. Rainer: *Bildverarbeitung in der Praxis.* R. Oldenbourg Verlag, 2002.

[17] Sze, S. M.: *Physics of Semiconductor Devices.* John Wiley & Sons, Inc., 2. Auflage, 1981.

[18] Tanenbaum, Andrew S.: *Moderne Betriebssysteme.* Pearson Studium Prentice Hall Verlag, 2. Auflage, Juni 2002.

[19] Weidemann, G., J. Goebbels, Th. Wolk und H. Riesemeier: *First Computed Tomography Experiments.* In: *BAMline BESSY, Annual Report 2001, CD-ROM & WWW*, 0103-I-0017. Bundesanstalt für Materialforschung und -prüfung (BAM), 2001.

[20] Wenger, Susanne, Mauricio Seeberger, Philipp Horvath, David Jörg und Thomas Staub: *Praktikum Bildanalyse: Thinning Algorithms.* Technischer Bericht, Universität Bern, Februar 2003.