Rekonstruktion von Oberflächenmorphologien und Merkmalskeletten aus dreidimensionalen Daten unter Verwendung hochparalleler Rechnerarchitekturen

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich 12

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Daniel Jungblut

aus Heidelberg

Frankfurt am Main 2012

(D30)

Vom Fachbereich 12 (Informatik und Mathematik) der
Johann Wolfgang Goethe - Universität als Dissertation angenommen.

Dekan: Prof. Dr. Thorsten Theobald

Gutachter: Prof. Dr. Gabriel Wittum, Prof. Dr. Gillian Queisser

Datum der Disputation: 22.11.2012

„Unkraut wächst in zwei Monaten, eine rote Rose braucht dafür ein ganzes Jahr.“

Dschalal ad-Din Muhammad Rumi
Islamischer Mystiker (1207-1273)

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Einleitung

1.1.1 Der klassische NeuRA-Algorithmus

Eine Verbindung zwischen den unterschiedlichen wissenschaftlichen Disziplinen Biologie, Neurowissenschaften, Mathematik und wissenschaftlichem Rechnen zu schaffen, stellt eine große Herausforderung dar. Der am interdisziplinären Zentrum für wissenschaftliches Rechnen der Universität Heidelberg entwickelte Algorithmus *NeuRA* (NEUron Reconstruction Algorithm) [20] zur automatischen Rekonstruktion von Oberflächengeometrien aus dreidimensionalen Mikroskopbildern schlägt eine Brücke zwischen diesen Disziplinen. In einschlägigen Vorarbeiten wurden durch von *NeuRA* rekonstruierten Geometrien Neuronenzellkerne vermessen [92] und die Ausbreitung von Signalprozessen in diesen Zellkernen untersucht [91]. Entgegen der lange vorherrschenden Meinung, dass die Zellkerne von Neuronenzellen kugelförmig sind, wurde mit dem Einsatz von *NeuRA* gezeigt, dass diese Zellkerne Einstülpungen aufweisen [90], und systematisch untersucht, wie diese Einstülpungen das Verhältnis von Oberfläche zu Volumen der Neuronenzellkerne beeinflussen [92]. Der klassische *NeuRA*-Algorithmus (Abbildung 1.1) aus dem Jahr 2004 arbeitet in drei Schritten:

1. Vorfilterung durch trägheitsbasierte anisotrope Diffusion
2. Segmentierung
3. Berechnung der Oberflächengeometrie durch den *Marching-Tetrahedron-Algorithmus* [91, 111] oder Extraktion eines Merkmalskeletts durch ein *Thinning-Verfahren* [19, 104]

1.1.2 Parallelisierung des Rekonstruktions-Algorithmus

Die bisherige serielle Implementierung von *NeuRA* benötigte mehrere Tage um Datensätze der Dimension 256^3 mit ca. 16 Millionen Bildpunkten, was einer Auflösung von älteren konfokalen und Zwei-Photonen-Mikroskopen (Kapitel 2) entspricht, zu verarbeiten. In den letz-

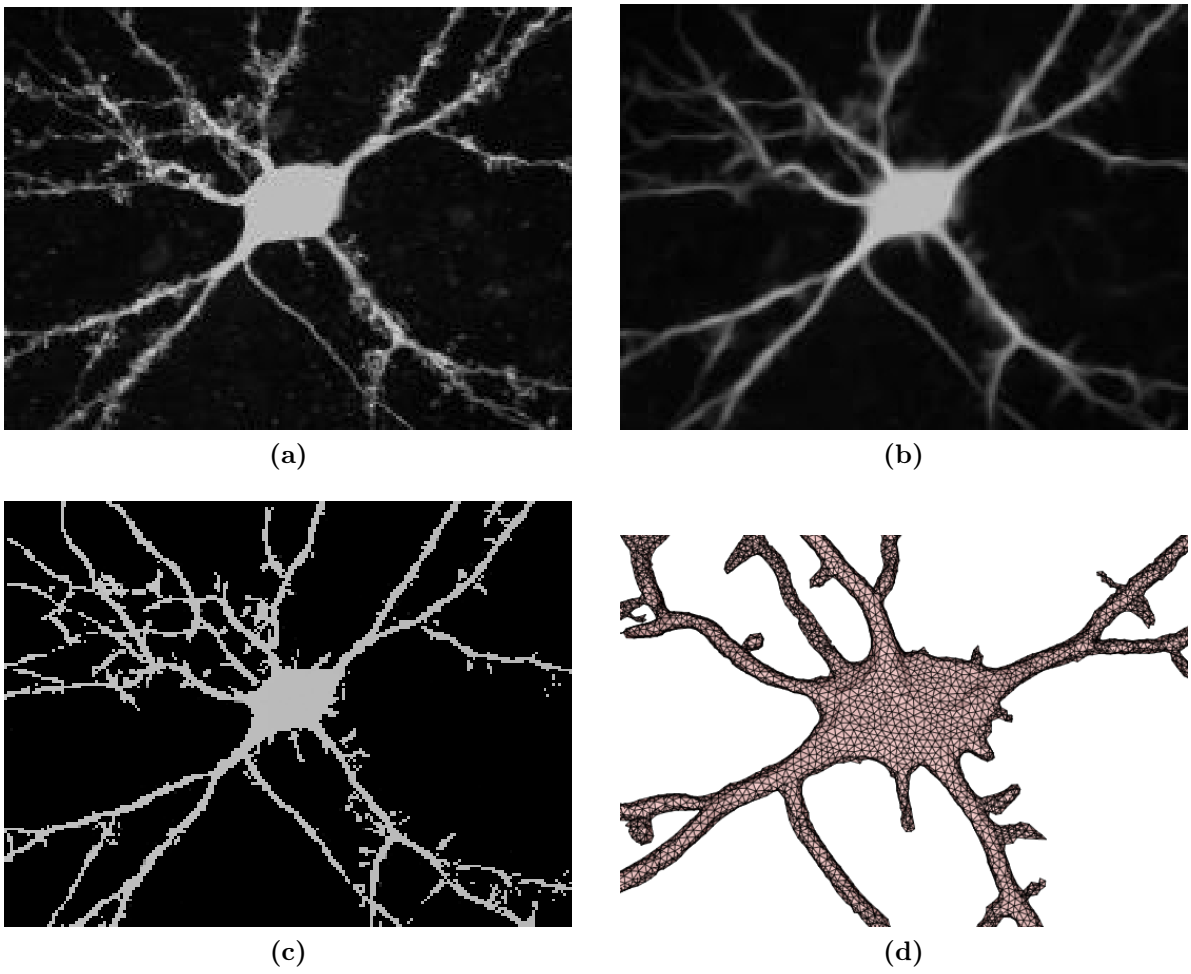
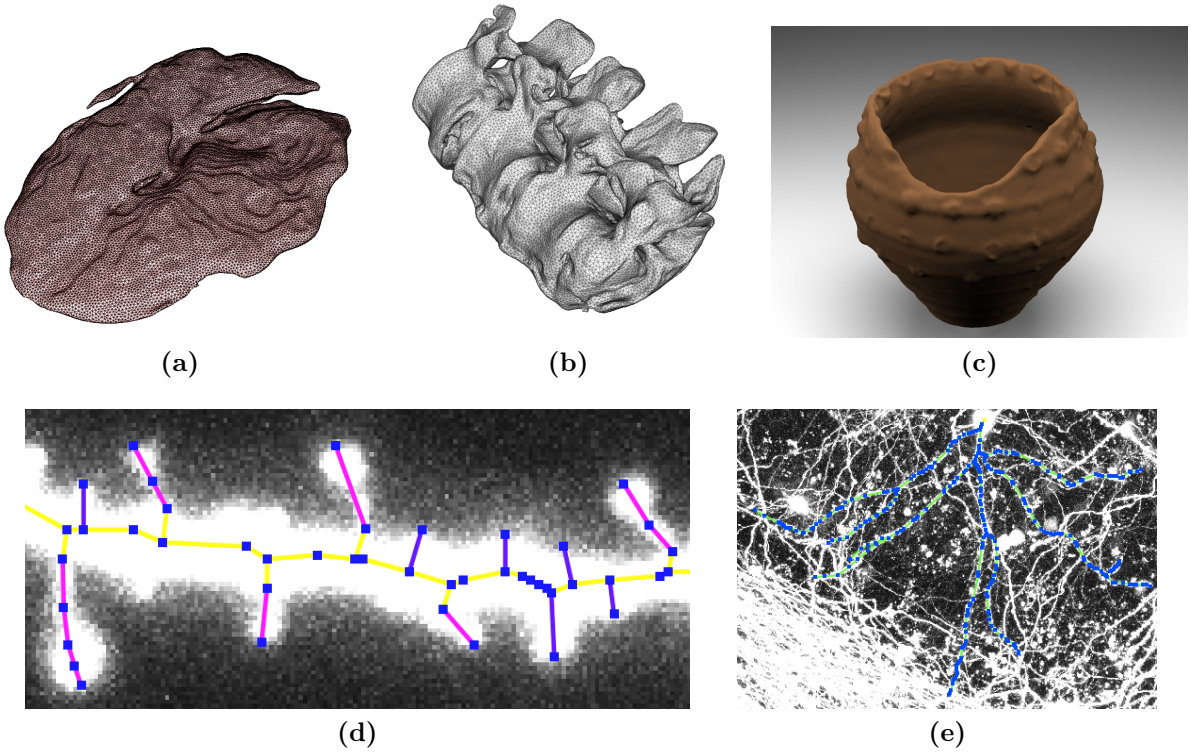


Abbildung 1.1: *Ablauf des klassischen* NeuRA-*Algorithmus: (a) Original Mikroskopaufnahme. (b) Gefilterte Aufnahme. (c) Segmentierte Aufnahme. (d) Rekonstruierte Oberflächengeometrie. Diese Rekonstruktion wurde mit Hilfe der im Rahmen dieser Arbeit entstandenen Neuimplementierung* NeuRA2 *erstellt.*

ten Jahren hat sich die Technik dieser Mikroskope deutlich weiterentwickelt, sodass mittlerweile Aufnahmen von einzelnen Neuronenzellen mit einer Auflösung von $2048 \times 2048 \times 400$, d.h. mit über 1,5 Milliarden Bildpunkten, erzeugt werden können. An eine Anwendung der seriellen Implementierung auf solche Datensätze ist trotz in dieser Zeit schneller gewordener Computer kaum zu denken. Im Rahmen dieser Arbeit wurde daher der *NEUron Reconstruction Algorithm* unter dem Namen *NeuRA2* weiterentwickelt, optimiert und neu implementiert. Hierbei wird die parallele Architektur moderner Hochleistungsgrafikkarten (GPUs) verwendet und gezeigt, welche Typen von Algorithmen sich zur Umsetzung auf dieser Architektur eignen und wie diese Algorithmen effizient GPU-basiert implementiert werden können.



(a) **(b)** **(c)**

(d) **(e)**

Abbildung 1.2: *Verschiedene mit* NeuRA2 *erzeugte Rekonstruktionen: (a) Innere Membran eines Neuronenzellkerns. (b) Halswirbelsäule. (c) Fotorealistische Darstellung eines rekonstruierten antiken Keramikgefäßes. (d) Merkmalskelett eines Dendritensegmentes mit dendritischen Dornfortsätzen. (e) Merkmalskelett einer Neuronenzelle. Weitere Rekonstruktionen und deren vielfältige Anwendungen sind in Kapitel 14 aufgeführt.*

1.1.3 Verallgemeinerung des Rekonstruktions-Algorithmus

Zudem wird in dieser Arbeit das Konzept des klassischen *NeuRA*-Algorithmus verallgemeinert. Durch eine datenabhängige Auswahl des optimalen Operators in jedem der Verarbeitungsschritte lassen sich die Oberflächen oder Merkmalskelette von nahezu beliebigen Objekten aus dreidimensionalen Bildern rekonstruieren (Abbildung 1.2). Zu den drei verallgemeinerten Schritten stehen in *NeuRA2* folgende Operatoren zur Verfügung:

1. Vorfilterung zur Verminderung des Rauschsignals:
   - Background-Noise-Subtraction (Kapitel 4.3.1)
   - Medianfilter (Kapitel 4.3.2)
   - Trägheitsbasierter Diffusionsfilter (Kapitel 5)
   - Multiskalen-Scharfzeichner (Kapitel 6)

2. Bildsegmentierung:

   - Globale Segmentierung (Kapitel 7.1)
   - Hysterese-Segmentierung (Kapitel 7.1)
   - Lokale Segmentierung nach *Otsu* (Kapitel 7.3)
   - Regionenwachstum-Segmentierung (Kapitel 7.4)

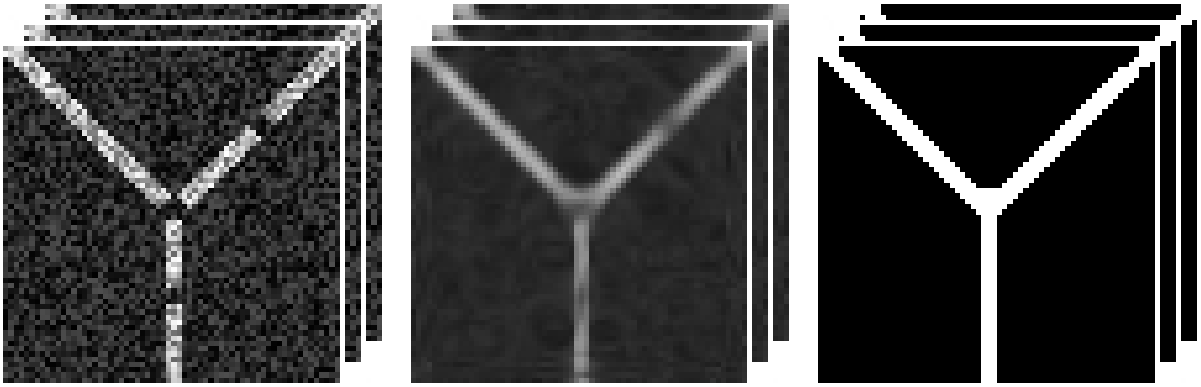
3. Rekonstruktion als Oberflächengeometrie oder Merkmalskelett:

   - Marching-Cubes-Gittergenerator (Kapitel 8)
   - Gitterzerlegung (Kapitel 9)
   - Gitteroptimierung (Kapitel 10)
   - Skelettierung (Kapitel 13)

1.1.4 Parallelisierung des trägheitsbasierten Diffusionsfilters

Die zentrale Rolle nimmt hierbei der trägheitsbasierte anisotrope Diffusionsfilter, auch trägheitsbasierter Diffusionsfilter genannt, ein. Wie in den Vorarbeiten [50, 90, 101] gezeigt, reduziert er das Rauschen in den Mikroskopaufnahmen drastisch, wobei die Größe der im Bild vorhandenen Strukturen erhalten bleibt und Lücken in diesen Strukturen geschlossen werden (Abbildung 1.3). Hinter dem trägheitsbasierten Diffusionsfilter steht die Idee, die Grauwerte des Bildes als Temperaturverteilung zu interpretieren und die Wärme entlang von Strukturen eine kurze Zeit fließen zu lassen, während Wärmefluss senkrecht zu Kanten im Bild nicht zugelassen wird. Demnach ist eine nichtlineare Wärmeleitungsgleichung auf den Bilddaten zu lösen, wobei das durch die Diskretisierung der Differentialgleichung entstehende lineare Gleichungssystem eine Gleichung und eine Unbekannte für jeden Bildpunkt aufweist. Für hochaufgelöste Mikroskopaufnahmen ist daher ein dünnbesetztes Gleichungssystem mit über 1,5 Milliarden Unbekannten aufzustellen und zu lösen. Da für eine hinreichende Rauschverminderung, wie in [50, 90, 101] gezeigt, mehrere Zeitschritte erforderlich sind, ist zur Anwendung dieses Algorithmus die mehrfache Lösung eines extrem hochdimensionalen Gleichungssystems nötig. Rechnet man die Laufzeit der seriellen Implementierung des Filters auf solche Datenmengen hoch, beträgt sie unter Verwendung eines aktuellen Computers (Stand 2010) knapp zwei Wochen. Eine massive Reduktion der Laufzeit ist durch den in *NeuRA2* gewählten parallelen Ansatz möglich. Hierzu sei die Funktionsweise des trägheitsbasierten Filters detailliert betrachtet, der in den folgenden wesentlichen Schritten arbeitet:

1. Strukturerkennung mittels lokaler Trägheitstensoren
2. Assemblierung der Systemmatrix
3. Lösen des Gleichungssystems



(a) **(b)** **(c)**

Abbildung 1.3: *Trägheitsbasierte anisotrope Diffusion erhält die Größe von Strukturen und kann Lücken innerhalb von Strukturen schließen. (a) Verrauschte Modellgeometrie eines Dendriten mit Lücken. (b) Nach Anwendung von trägheitsbasierter Diffusion. (c) Nach anschließender Segmentierung.*

In meiner Diplomarbeit [50] ist gezeigt, dass diese Teile des Algorithmus auf elementarster Ebene hochgradig parallel berechnet werden können. Eine GPU-basierte Implementierung des trägheitsbasierten Diffusionsfilters unter Verwendung eines reduzierten Diskretisierungsschemas mit Hilfe von *Cg-Shadern* [35] ist ebenfalls beschrieben. Das reduzierte Diskretisierungsschema führt allerdings zu Artefakten, die eine Weiterverarbeitung der Daten teilweise erschweren. Daher ist in dieser Arbeit die Genauigkeit des ursprünglichen Diskretisierungsschemas [101] mit der Geschwindigkeit moderner Grafikkarten kombiniert und in Kapitel 5 eine in *CUDA* [78] implementierte Variante des trägheitsbasierten Diffusionsfilters beschrieben. Aufgrund des lokalen Verhaltens der dem Filter zugrunde liegenden nichtlinearen Wärmeleitungsgleichung können große Bilder in sich überlappende Teilbilder zerschnitten werden, die einzeln gefiltert und dank der Überlappung anschließend zu einem stetigen Ergebnisbild zusammengesetzt werden können. Dieses Vorgehen ermöglicht zudem eine parallele Verarbeitung der einzelnen Teilbilder, die auf verschiedenen Grafikkarten gleichzeitig gefiltert werden können. Die Verwendung des GPU-basierten Parallelrechners *Scout*, der über 96 Tesla-C1060 Prozessoren [79] verfügt, ermöglicht die Bearbeitung hochaufgelöster Mikroskopaufnahmen innerhalb weniger Minuten.
Parallelisierung findet hierbei auf zwei verschiedenen Ebenen statt: Zum Einen werden die einzelnen Teilbilder parallel auf verschiedenen Grafikprozessoren gleichzeitig gefiltert, zum Anderen finden die einzelnen Filterprozesse hochgradig parallel statt, da jeder der Tesla-C1060 Prozessoren über 240 Rechenwerke verfügt. Schon die Verwendung eines einzigen Tesla-C1060 Prozessors beschleunigt die Laufzeit des trägheitsbasierten Diffusionsfilters fast um den Faktor 100 verglichen mit einem Kern eines modernen Standardprozessors und zeigt so das massive Potential moderner Grafikkarten bei speziellen Anwendungen.

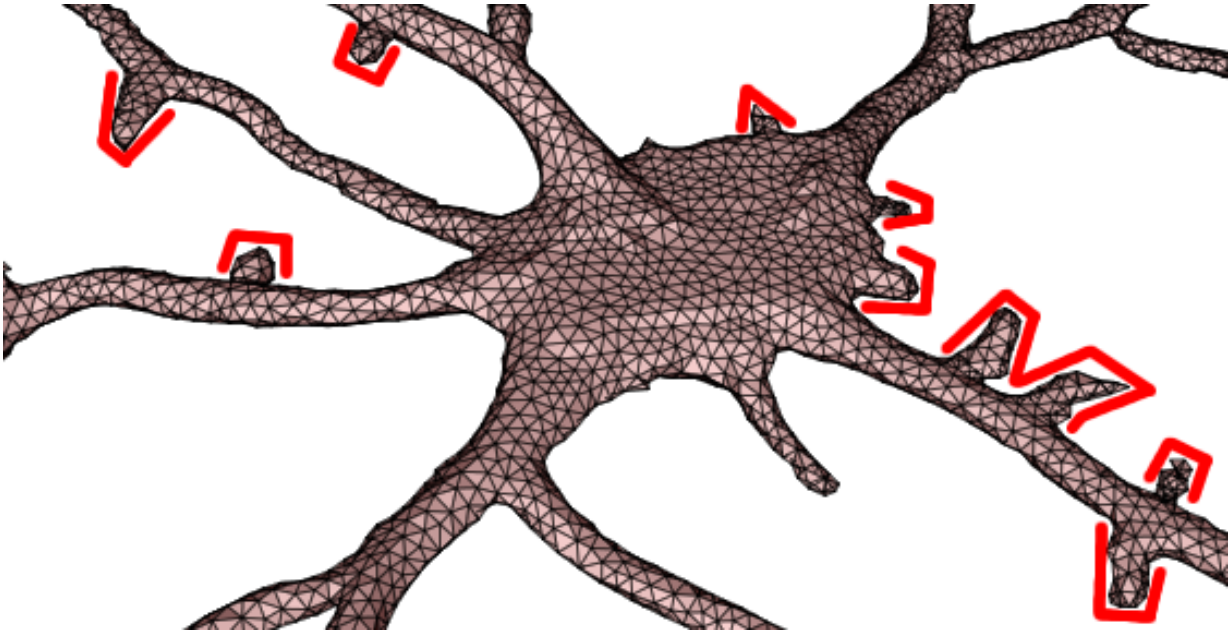


Abbildung 1.4: *Die Erweiterung von* NeuRA2 *durch den Multiskalen-Scharfzeichner (Kapitel 6) ermöglicht die korrekte Rekonstruktion von dendritischen Dornfortsätzen (rot markiert), auch* Spines *genannt.*

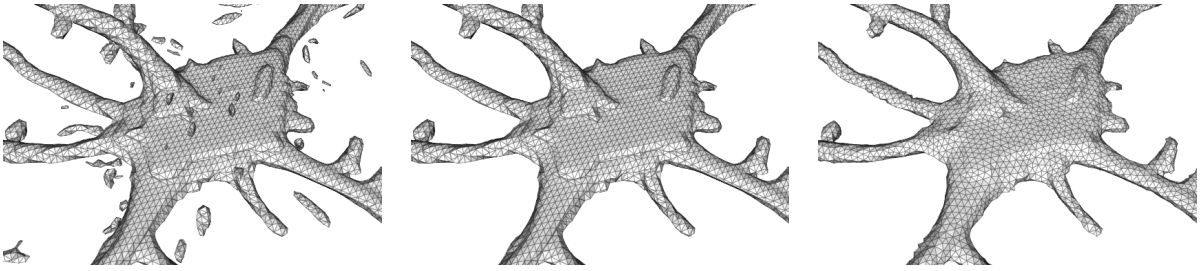
1.1.5 Rekonstruktion hochaufgelöster Aufnahmen

Die Verarbeitung hochaufgelöster Mikroskopaufnahmen ermöglicht die Rekonstruktion von sehr feinen Strukturen einzelner Neuronenzellen. Die nur wenigen Nanometer großen dendritischen Dornfortsätze, sogenannte *Spines* [119], müssen von dem Rekonstruktionsverfahren erkannt und korrekt rekonstruiert werden. Die Spines sind zwar in den Mikroskopaufnahmen sichtbar, weisen verglichen mit den dickeren Dendritenästen aber einen schlechten Kontrast auf. Die ursprünglich in *NeuRA* verwendeten Segmentierungsmethoden sind daher nicht in der Lage diese Spines korrekt zu segmentieren. Aus diesem Grund ist in *NeuRA2* mit dem *Multiskalen-Scharfzeichner* (Kapitel 6) ein weiterer Vorfilterprozess integriert, der linienartige Strukturen unterschiedlicher Dicke scharfzeichnet. Die eher rundlichen Zellkerne werden durch die ursprüngliche Variante des Scharfzeichners [37] allerdings teilweise zerstört. Da diese in den Mikroskopbildern glücklicherweise sehr hell sind, kann der Scharfzeichnungsprozess mit einer globalen Schwellwertsegmentierung (Kapitel 7.1) und einer Histogramm-Normalisierung (Kapitel 4.2.2) kombiniert werden, um dieses Manko auszugleichen. Eine effiziente parallele Implementierung des Multiskalen-Scharfzeichners ist leicht möglich, da bei diesem Operator keinerlei Interaktion zwischen Berechnungen auf unterschiedlichen Bildpunkten stattfindet.

1.1.6 Erweiterung der Segmentierungsverfahren

Bei der sogenannten *Segmentierung* (Abbildung 1.1c und Kapitel 7) wird für jeden Bildpunkt des vorverarbeiteten Bildes entschieden, ob er zum im Bild befindlichen Objekt oder zum Hintergrund gehört. Hierbei wird ein Binärbild erzeugt, das lediglich aus weißen und schwarzen Bildpunkten besteht. Die ursprüngliche Implementierung von *NeuRA* beinhaltet neben der *globalen Schwellwertsegmentierung* (Kapitel 7.1) die lokale Segmentierung nach *Otsu* [85], die auf einer statistischen Überlegung beruht. Die Segmentierung nach *Otsu* (Kapitel 7.3) wurde in den Arbeiten [90, 91] erfolgreich zur Rekonstruktion von Neuronenzellkernen eingesetzt. Dieses sehr rechenintensive Verfahren lässt sich ebenfalls effizient auf Grafikkarten implementieren, wodurch Beschleunigungen von über einem Faktor 50 gegenüber Standardprozessoren der aktuellen Generation zu beobachten sind. Zur Rekonstruktion von Daten aus Computertomographie-Aufnahmen wurde in *NeuRA2* zudem eine *Regionenwachstum-Segmentierung* (Kapitel 7.4) integriert. Hierbei werden ausgehend von Bildpunkten, die mit absoluter Sicherheit zum Objekt gehören, nach und nach alle Nachbarbildpunkte dieser Bildpunkte überprüft und entsprechend dem Objekt oder dem Hintergrund zugeordnet. Der Rand des zu segmentierenden Objekts wird hierbei sehr exakt durch die Verwendung des Grauwertgradienten (Kapitel 4.4.1) detektiert, was sich vor allem für die Segmentierung von Computertomographie-Aufnahmen eignet, da diese meist eine sehr homogene Grauwertverteilung aufweisen. Zudem wurde *NeuRA2* durch die *Hysterese-Segmentierung* (Kapitel 7.1) erweitert.

1.1.7 Gittererzeugung und Gitteroptimierung



(a) **(b)** **(c)**

Abbildung 1.5: *(a) Durch den Marching-Cubes-Algorithmus generiertes Oberflächengitter. (b) Nach Entfernung der Blobs. (c) Nach Gitteroptimierung.*

Aus den segmentierten Bildern wird nach einer weiteren Anwendung des trägheitsbasierten Diffusionsfilters die Oberflächengeometrie der Objekte durch den *Marching-Cubes-Algorithmus* (Abbildung 1.5a und Kapitel 8) rekonstruiert. Dieses Verfahren wurde ebenfalls in CUDA implementiert, wobei sich zeigt, dass hierbei keine Beschleunigungsfaktoren wie bei der trägheitsbasierten Diffusion oder der Segmentierung nach *Otsu* erreicht werden können, da bei jedem parallelen Ansatz des Marching-Cubes-Algorithmus ein hoher Overhead produziert wird, der den durch die Parallelisierung gewonnenen Laufzeitgewinn zum

größten Teil wieder zunichte macht. Zwei Nachverarbeitungsschritte auf den generierten Gittern schließen den Rekonstruktionsprozess ab:
Die durch den Marching-Cubes-Algorithmus generierten Oberflächengitter enthalten häufig sogenannte *Blobs*, kleine disjunkte Teilgitter, die durch das Rauschsignal der ursprünglichen Mikroskopaufnahme Eingang in die Rekonstruktion erhalten haben. Betrachtet man das Oberflächengitter als Graphen und bestimmt dessen Zusammenhangskomponenten durch eine Tiefensuche, können die störenden Blobs von der eigentlichen Geometrie der rekonstruierten Zelle separiert werden (Abbildung 1.5b und Kapitel 9). Zudem bestehen die rekonstruierten Geometrien aus sehr vielen Dreiecken und besitzen teilweise sehr scharfe Kanten. Durch Anwendung einer modifizierten Version des in [45] beschriebenen Algorithmus zur Gitteroptimierung wird ein topologisch gleiches Gitter erzeugt, das deutlich weniger Vertices enthält und bei dem die Kanten geglättet sind (Abbildung 1.5c und Kapitel 10).

1.1.8 Oberfläche, Volumen und Krümmung

In den Kapiteln 11 und 12 ist erklärt, wie die Oberfläche, das Volumen und die lokale diskrete Krümmung eines durch ein Dreiecksgitter approximierten Objekts berechnet werden kann und die Genauigkeit dieser Berechnungen analysiert. Unter der Voraussetzung, dass ein Dreiecksgitter eine geschlossene zweidimensionale Fläche im dreidimensionalen Raum beschreibt, lässt sich deren Oberfläche durch Aufsummieren aller Dreiecksflächen berechnen (Kapitel 11.1). Das Volumen kann hingegen auf zwei unterschiedliche Weisen bestimmt werden: Entweder man überführt das Oberflächengitter in ein aus Tetraedern bestehendes Volumengitter und summiert die Volumina aller Tetraeder auf, oder man überführt das Volumenintegral über das Innere des Gitters mit Hilfe des Integralsatzes von Gauß in ein Oberflächenintegral, das wiederum durch Summieren über alle Dreiecke des Gitters berechnet werden kann (Kapitel 11.2). Es ist gezeigt, dass beide Verfahren praktisch dieselben Ergebnisse liefern (Kapitel 11.3.1). Des Weiteren ist der Einfluss der Gitteroptimierung auf die berechnete Oberfläche und das Volumen einer Kugelgeometrie untersucht und gezeigt, dass bereits grobe Gitter für eine sehr exakte Bestimmung dieser Größen ausreichen (Kapitel 11.3.2). Das Ergebnis des Marching-Cubes-Algorithmus hängt vom sogenannten *Isoflächen-Schwellwert* ϑ ab, in dessen Abhängigkeit eine lineare Interpolation der Vertexkoordinaten des rekonstruierten Gitters durchgeführt wird und der somit die Oberfläche und das Volumen der rekonstruierten Geometrien stark beeinflusst. Daher ist in Abschnitt 11.3.3 gezeigt, dass der Wert $\vartheta = 157$ optimal zur Berechnung von Oberfläche und Volumen eines Objekts ist. Dieser Wert wurde bei der Gitterrekonstruktion für alle weiteren Anwendungen in dieser Arbeit verwendet. Darüber hinaus ist die Genauigkeit des Verfahrens für die Volumenberechnung bei archäologischen Anwendungen evaluiert (Kapitel 11.4). Hierbei zeigt sich, dass das Volumen von Keramiken und die Kapazität von hochaufgelöst aufgenommenen Keramikgefäßen exakter bestimmt werden kann, als dies bei den üblichen archäologischen Methoden der Fall ist (vgl. Abschnitt 1.1.10). Der Einfluss der Eingabeparameter des Rekonstruktionsprozesses und der Auflösung der Aufnahmen ist hierbei ebenfalls detailliert untersucht, sowie gezeigt, dass der in Kapitel 10

beschriebene Gitteroptimierungs-Algorithmus keinen signifikanten Einfluss auf die berechneten Volumenwerte hat. In Kapitel 12 ist schließlich erklärt, wie der Begriff der *Krümmung* auf einem diskreten Dreiecksgitter definiert werden kann. Ferner ist anhand einer Modellgeometrie gezeigt, dass die diskrete lokale Krümmung quadratisch in der maximalen lokalen Gittermaschenweite h gegen die kontinuierliche Krümmung konvergiert und das im Falle einer Kugel mit Radius r eine maximale Gittermaschenweite von $h = \frac{r}{10}$ ausreicht um die lokale Krümmung mit einem Fehler von maximal einem Prozent zu approximieren. Dies entspricht einer Triangulierung mit etwa 3000 Dreiecken.

1.1.9 Extraktion von Merkmalskeletten

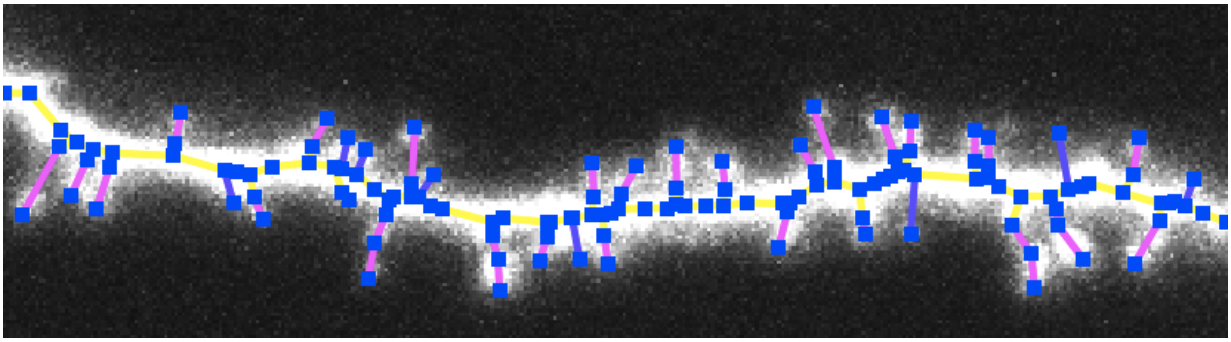
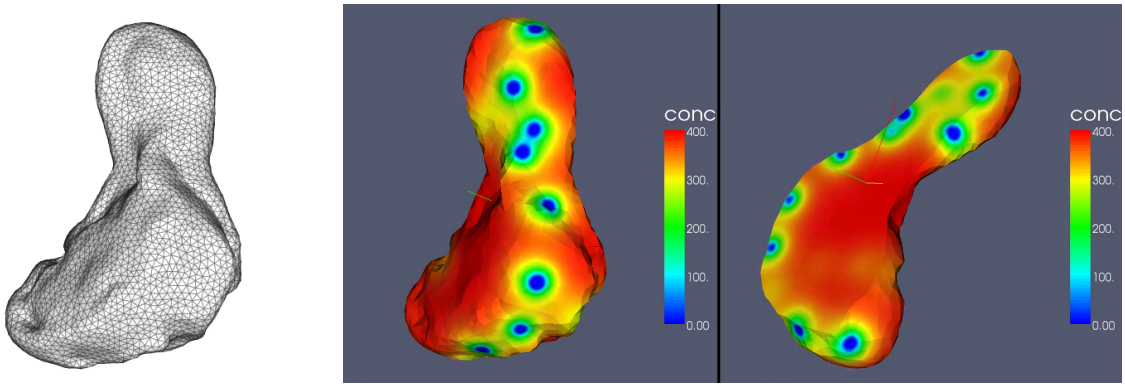


Abbildung 1.6: *Merkmalskelett eines Dendritensegments mit Spines.*

Alternativ kann ein Merkmalskelett des im dreidimensionalen Bild enthaltenen Objekts erzeugt werden (Abbildung 1.6 und Kapitel 13). Hierbei wird das vorgefilterte Bild entlang der z-Achse mittels Maximumnorm auf eine Ebene projiziert, aus der dann durch das *Augmented-Fast-Marching-Verfahren* [110] ein zunächst zweidimensionales Skelett erzeugt wird. Jedem Knoten dieses zweidimensionalen Skeletts wird anschließend eine Koordinate entlang der z-Achse zugeordnet, indem der Schwerpunkt der Grauwertverteilung entlang der z-Achse bestimmt wird (Kapitel 13.2.1), sodass insgesamt ein dreidimensionales Merkmalskelett entsteht. Da die in dieser Arbeit betrachteten Aufnahmen von lebenden Zellen stammen, sind die Aufnahmen meist sehr verrauscht oder enthalten Teile von weiteren Zellen, die nicht rekonstruiert werden sollen. Daher ermöglicht die im Rahmen dieser Arbeit entstandene Software *SpineLab* eine manuelle Nachbearbeitung dieser Skelette. Anschließend kann eine Skelettanalyse durchgeführt werden, welche die Messung der Anzahl an Spines, deren durchschnittliche Länge, sowie die Länge und das Volumen einzelner Spines beinhaltet. Bei den Skelettrekonstruktionen von kompletten Neuronenzellen kann ebenfalls die Länge der einzelnen Dendritensegmente sowie der Winkel zwischen Verzweigungen automatisch analysiert werden. Solche Analysen ermöglichen eine Untersuchung morphologischer Veränderungen von Nervenzellen unter dem Einfluss pharmakologischer Präparate oder mechanischer Eingriffe (Kapitel 14.4.2).

1.1.10 Anwendungen und Ergebnisse



(a) **(b)**

Abbildung 1.7: *(a) Rekonstruktion eines präsynaptischen Boutons. (b) Simulation der Vesikeldynamik innerhalb des Boutons. Details zu dieser und weiteren Simulationen auf rekonstruierten Geometrien finden sich in Kapitel 14.2.*

Die vielfältigen Anwendungen der Rekonstruktionen sind in Kapitel 14 ausführlich beschrieben. Neben den bereits angesprochenen Längen und Volumenanalysen können die Rekonstruktionen zur Visualisierung von Objekten aus dreidimensionalen Bildern eingesetzt werden (Kapitel 14.1). Zudem können die generierten Gitter als Grundlage für Simulationen verwendet werden (Abbildung 1.7 und Kapitel 14.2). Ferner wurden mit *NeuRA2* Computertomographie-Aufnahmen von mehr als zwanzig antiken Keramikgefäßen und Fragmenten antiker Keramikgefäße rekonstruiert (Kapitel 14.3). Durch die Berechnung des Volumens (Kapitel 11) der rekonstruierten Geometrien lässt sich die Rohdichte der Keramiken ermitteln. Hierbei zeigt sich eine Abhängigkeit der Rohdichte vom jeweils verwendeten Keramiktyp. Zudem lässt sich mit Hilfe von *NeuRA2* das Füllvolumen der Gefäße sehr exakt bestimmen, was bei einigen Gefäßen aufgrund der Gefahr der Beschädigung, die von herkömmlichen archäologischen Methoden ausgeht, bisher nicht möglich war. Weitere rekonstruierte CT-Datensätze, die nicht zu den bisher beschriebenen Anwendungsfeldern gehören, finden sich in Kapitel 14.5 und 14.6. Die Vielfalt der in dieser Arbeit behandelten Anwendungen, denen alle morphologische Rekonstruktionen zugrunde liegen, zeigt die Flexibilität des Rekonstruktions-Algorithmus *NeuRA2* und schlägt Brücken zwischen verschiedensten wissenschaftlichen Disziplinen.

1.1.11 Erweiterbarkeit

Die Ergebnisse dieser Arbeit gehen weit über theoretische Machbarkeitsstudien hinaus. Sie zeigen vielmehr die weitläufige Anwendbarkeit der entwickelten Bildverarbeitungs-, Rekonstruktions- und Analyseverfahren, die auf einfache Weise auf weitere Daten angewandt und somit leicht auf zukünftige Anwendungsfelder übertragen werden können. Aus diesem Grund wurde bei den im Rahmen dieser Arbeit entstandenen Implementierungen

NeuRA2 (Anhang A) und *SpineLab* (Anhang B) gezielt Wert auf leichte Erweiterbarkeit gelegt, um weitere Algorithmen, die für zukünftige Anwendungen benötigt werden, in die bestehenden Verfahren zu integrieren.

1.2 Gliederung der vorliegenden Arbeit

Nach einigen Vorbemerkungen über bereits veröffentlichte Teilergebnisse dieser Arbeit (Kapitel 1.3.1), den in dieser Arbeit dokumentierten Programmcode (Kapitel 1.3.2) und die in dieser Arbeit dargestellten dreidimensionalen Bilder (Kapitel 1.3.3) folgen einführende Kapitel:
In Kapitel 2 sind die für die Aufnahmen verwendeten Techniken der konfokalen und Zwei-Photonen-Mikroskope und der Computertomographie erklärt. Kapitel 3 enthält die wichtigsten Informationen und Techniken zur parallelen Programmierung mit CUDA, die verwendet wurden um einige in dieser Arbeit behandelten Bildverarbeitungsoperatoren parallel zu implementieren. Die Grundlagen und elementaren Operatoren der digitalen Bildverarbeitung, die in den späteren Kapiteln zur Konstruktion komplexer Bildfilter und der Rekonstruktionsverfahren verwendet werden, werden in Kapitel 4 erklärt. Die in *NeuRA2* und *SpineLab* implementierten Algorithmen *trägheitsbasierter Diffusionsfilter*, *Multiskalen-Scharfzeichner*, *Segmentierung*, *Gittergenerierung*, *Gitterzerlegung*, *Gitteroptimierung*, *Oberflächen- und Volumenberechnung* und *Skelettierung* mit detaillierter Beschreibung der entsprechenden Algorithmen, ihrer Implementierungen und der numerischen Optimierung ihrer Eingabeparameter folgen in den Kapiteln 5 bis 11 und 13. In Kapitel 12 ist die Genauigkeit der Krümmung der durch Dreiecksgitter rekonstruierten Objekte thematisiert und gezeigt, wie grob ein Gitter maximal sein darf, um die Krümmung von Oberflächen hinreichend exakt zu approximieren. Kapitel 14 erläutert ausführlich die interdisziplinären Anwendungen der durch *NeuRA2* und *SpineLab* gewonnenen Rekonstruktionen. Die abschließende Diskussion in Kapitel 15 gibt Ausblicke auf mögliche Verbesserungen und Optimierungen der Rekonstruktions-Algorithmen und zukünftigen Anwendungen. Im Anhang der Arbeit finden sich Details zu den im Rahmen dieser Arbeit implementierten Programmen *NeuRA2* (Anhang A) und *SpineLab* (Anhang B), sowie eine Referenztabelle der rekonstruierten Keramikgefäße (Anhang C) und detaillierte Tabellen zur Genauigkeitsanalyse aus Kapitel 11.4.4 (Anhang D).

1.3 Vorbemerkungen

1.3.1 Bisherige Veröffentlichungen

Die Ergebnisse der vorliegenden Arbeit wurden bereits teilweise bei Konferenzen vorgestellt bzw. in wissenschaftlichen Fachzeitschriften veröffentlicht. Im Einzelnen sind das:

- Die Ergebnisse zur GPU-basierten Implementierung des trägheitsbasierten anisotropen Diffusionsfilters wurden in [50] veröffentlicht.

- Das Verfahren zur Rekonstruktion von archäologischen CT-Daten wurde auf der Konferenz *Scientific Computing and the Cultural Heritage 2009* in Heidelberg vorgestellt und ist in [51] publiziert.

- Die Validierung und Ergebnisse zu *SpineLab* sind in [53] abgedruckt.

- Weitere Ergebnisse zu den Anwendungen der Archäologie befinden sich derzeit (Stand: Dezember 2012) im Druck [56, 57].

1.3.2 Quellcodebeispiele

In einigen späteren Kapiteln, sowie im Kapitel 3 über CUDA, werden die Implementierungen der in dieser Arbeit vorgestellten und diskutierten Methoden und Verfahren durch Quellcodebeispiele verdeutlicht. Diese Quellcodebeispiele folgen einheitlichen Konventionen:

1. Quellcode hebt sich von beschreibendem Text durch die Verwendung von `Schreibmaschinenschrift` ab.

2. Die Dimensionen der bearbeiteten Daten werden entsprechend der Koordinatenrichtungen mit `size_x`, `size_y` und `size_z` bezeichnet.

3. Die Gesamtgröße von Daten wird als `size = size_x * size_y * size_z` bezeichnet.

4. Um eigens definierte Funktionen und Variablen von welchen aus eingebundenen Bibliotheken deutlich abzugrenzen, verwenden entgegen der üblichen Konvention alle selbst definierten Bezeichner Unterstriche zur Bildung zusammengesetzter Namen, wie beispielsweise:

   ```
   int number_of_vertices;
   void print_vector(float * v);
   ```

5. Zeigervariablen vom Host, die auf Speicherbereiche des Device-Speichers zeigen, sind mit dem Präfix `cuda_` versehen:

   ```
   float * cuda_x;
   ```

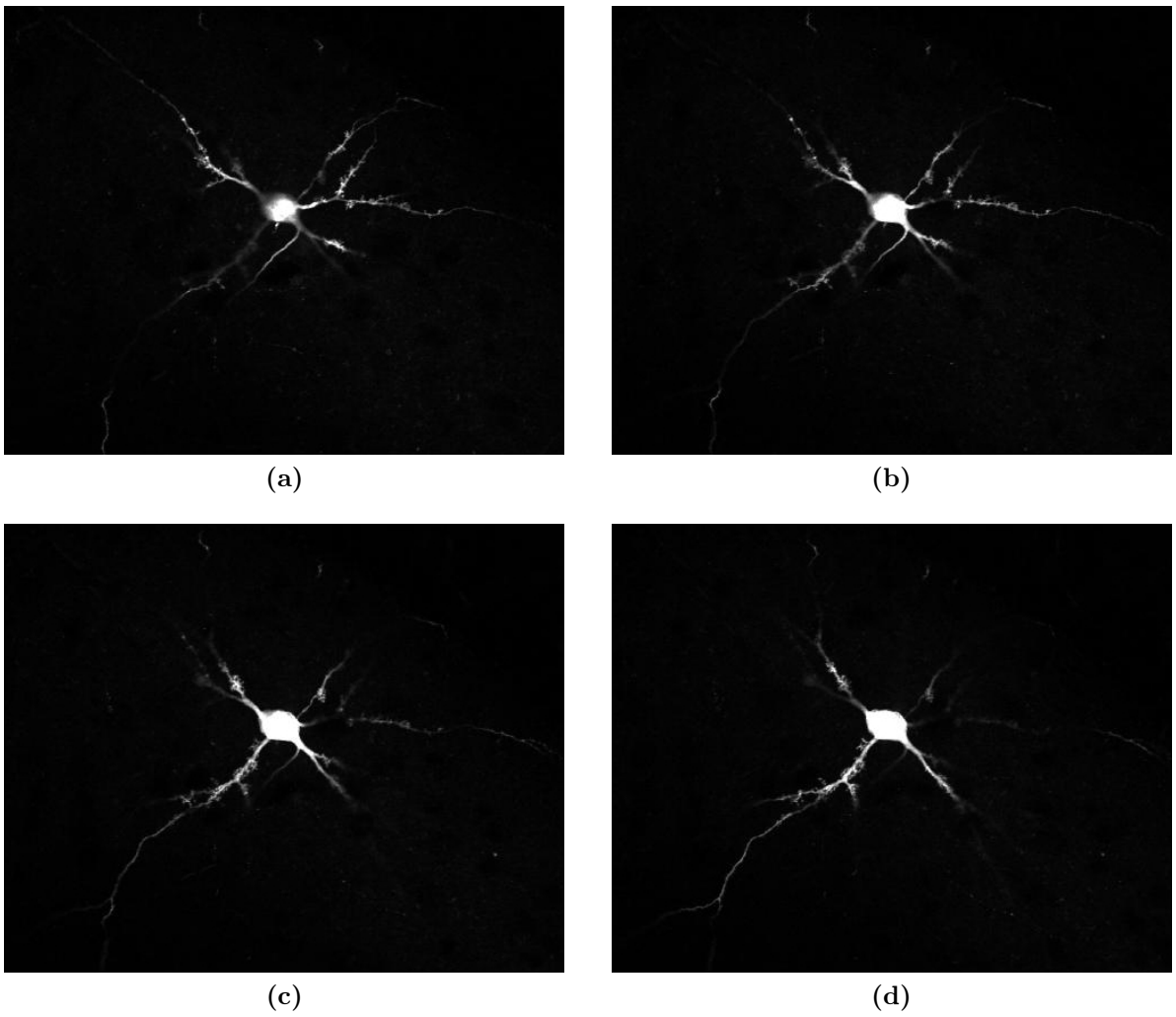
   Eine Erklärung zu den Begriffen Host und Device findet sich in Kapitel 3.3.

6. Funktionen, die vom Host aufgerufen werden und auf dem Device laufen, sind mit dem Suffix `_kernel` versehen.

1.3.3 Darstellung dreidimensionaler Bilder

Der Rekonstruktions-Algorithmus *NeuRA2* verarbeitet dreidimensionale Bilder, die man sich als Bildstapel zweidimensionaler Bilder vorstellen kann. Um diese dreidimensionalen Bilder und die Ergebnisse der einzelnen Verarbeitungsschritte auf einem Computermonitor anzuzeigen oder einem Blatt Papier auszudrucken, müssen die dreidimensionalen Bilder zweidimensional dargestellt werden. Die in dieser Arbeit abgedruckten Visualisierungen von dreidimensionalen Bildern wurden auf zwei verschiedene Arten gewonnen:

- Darstellung einer Schnittebene oder Volumenprojektion



(a) **(b)** **(c)** **(d)**

Abbildung 1.8: *Darstellungen von verschiedenen Schnittebenen eines dreidimensionalen Bildstapels.*

Bei der *Darstellung einer Schnittebene* wird das dreidimensionale Bild mit einer Ebene geschnitten und alle diese Ebene schneidenden Bildpunkte dargestellt. Zeigt die Normale der

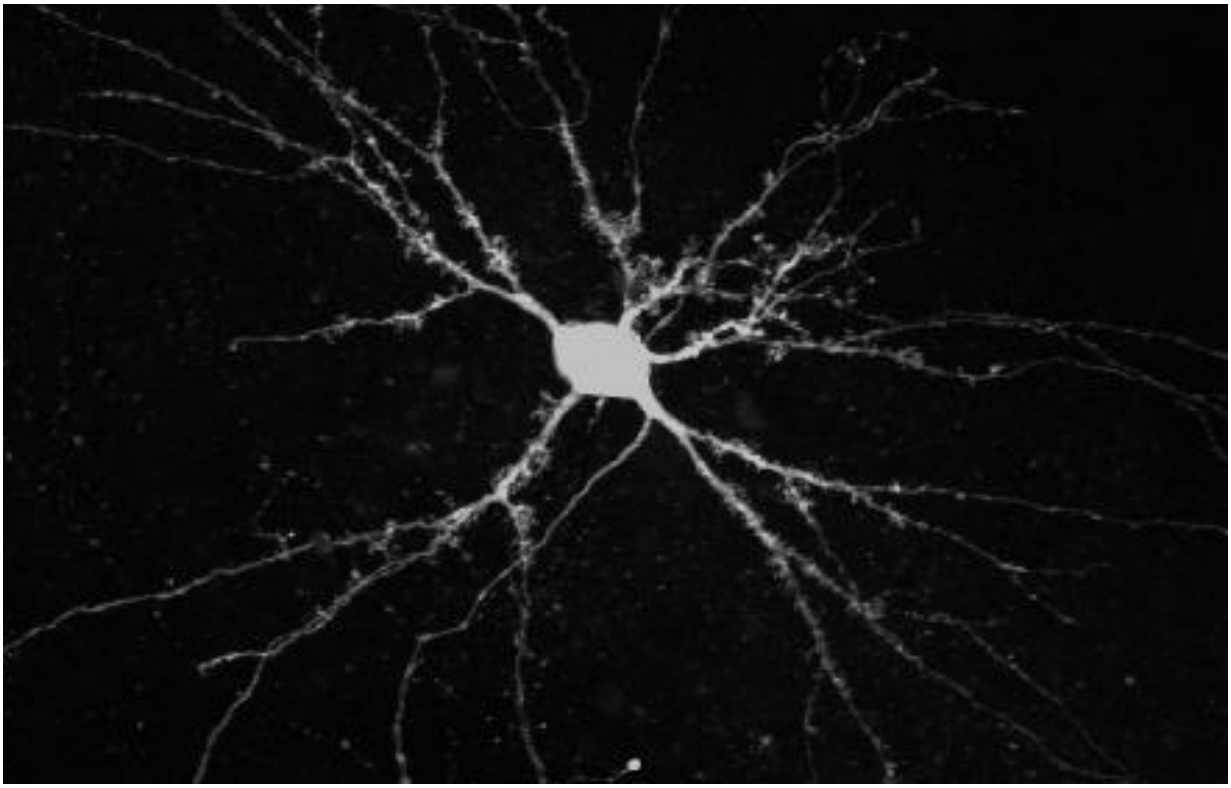


Abbildung 1.9: *Volumenprojektion des Bildstapels aus Abbildung 1.8.*

Schnittebene in z-Richtung, so ist das resultierende Schnittbild eines der sich im Bildstapel befindlichen Einzelbilder (Abbildung 1.8).

Bei der *Volumenprojektion* (Abbildung 1.9) wird ein zweidimensionales Bild erzeugt, indem von jedem zu generierenden Bildpunkt ein Strahl durch das dreidimensionale Bild geschickt wird. Abhängig davon, wo dieser Strahl auf das sich im Bildstapel befindliche Objekt trifft, wird das Pixel des zu generierenden Bildes entsprechend eingefärbt. Dieses Verfahren wird als *Raycasting* bezeichnet und kann detailliert in [32, 33] nachgelesen werden.

Alle in dieser Arbeit abgedruckten Schnittebenendarstellungen und Volumenprojektionen dreidimensionaler Bilder wurden mit der Software *ImageJ* [48] und den Benutzeroberflächen von *NeuRA2* (Anhang A) und *SpineLab* (Anhang B) erstellt. Bei den einzelnen Abbildungen ist jeweils gekennzeichnet, ob es sich um Schnittebenendarstellungen oder Volumenprojektionen handelt.

Kapitel 2

Bildaufnahme durch konfokale Mikroskopie, Zwei-Photonen-Mikroskopie und Computertomographie

2.1 Lichtmikroskopie im Allgemeinen

Normale Lichtmikroskope, sogenannte *Transmissionsmikroskope*, durchstrahlen das zu mikroskopierende Objekt, im Folgenden als *Probe* bezeichnet, mit für das menschliche Auge sichtbarem Licht. Der Anteil des Lichts, der von der Probe nicht absorbiert wird, wird auf einer Projektionsebene aufgefangen, wodurch ein Bild der Probe entsteht. Durch den Einsatz von entsprechenden Linsen kann dieses transmittierte Licht so gebündelt werden, dass ein scharfes Bild einer Schnittebene, der sogenannten *Fokusebene* der Probe, entsteht. Durch Verwendung verschiedener Fokusebenen kann ein Bildstapel, d.h. ein dreidimensionales Bild, der Probe gewonnen werden.

2.2 Epifluoreszenz-Mikroskopie

Bei der *Epifluoreszenz-Mikroskopie* werden die Bereiche der Probe, die aufgenommen werden sollen, mit einem fluoreszierenden Farbstoff präpariert, der von Licht einer bestimmten Wellenlänge zur Fluoreszenz angeregt wird. Dabei werden die Moleküle des Fluoreszenzfarbstoffes durch Aufnahme einer bestimmten Energiemenge auf ein höheres Energieniveau angehoben und strahlen beim Zurückfallen auf das Ausgangsniveau die aufgenommene Energie als Fluoreszenzlicht wieder ab. Anstatt sichtbaren Lichts wird bei der Epifluoreszenz-Mikroskopie Licht mit exakt der das Fluoreszenzmittel anregenden Wellenlänge zum Durchleuchten der Probe verwendet. Das Bild der fluoreszierenden Probe wird dabei Ebene für Ebene aufgenommen. Um nicht fälschlicherweise das anregende Licht aufzunehmen, wird das transmittierte Licht durch einen Filter geleitet, der das vom Mi-

kroskop ausgesandte Licht abfängt und nur das vom Fluoreszenzmittel ausgesandte Licht durchlässt. Ein Nachteil der bisher vorgestellten Techniken ist, dass das Bildsignal der Fokusebene massiv durch unscharfe Bereiche benachbarter Ebenen überlagert wird.

2.3 Konfokale Mikroskopie

Bei der konfokalen Mikroskopie wird die Probe nicht mehr schichtweise als Ganzes abgebildet, sondern Punkt für Punkt abgetastet. Hierbei wird der aufzunehmende Punkt in der Probe mit einem Laser, der die verwendete Fluoreszenz zum Leuchten anregt, fokussiert. Der Laserstrahl regt nicht nur das Fluoreszenzmittel in diesem Punkt, sondern auch das Fluoreszenzmittel auf dem Weg, den er zum fokussierten Punkt zurücklegt, an. Daher wird vor die Projektionsebene eine Lochblende gesetzt, die von außerhalb des Fokuspunktes herrührendes Licht abblockt.

2.4 Zwei-Photonen-Mikroskopie

Bei der *Zwei-Photonen-Mikroskopie* (Abbildung 2.1) wird ein Laser verwendet, der Licht der doppelten Wellenlänge, die zur Anregung des Farbstoffes benötigt wird, ausstrahlt. Einzelne Photonen können so das Fluoreszenzmittel nicht zum Leuchten anregen. Wenn aber zwei Photonen nahezu gleichzeitig auf ein Molekül des Fluoreszenzmittels treffen, addieren sich die Energien der beiden Photonen, sodass es zum Leuchten angeregt wird. Da dies nur im Fokuspunkt des Lasers möglich ist, wird das Fluoreszenzmittel exakt an der aufzunehmenden Stelle angeregt. Die bei der konfokalen Mikroskopie verwendete Lochblende ist dann überflüssig. Außerdem wird durch die geringere Laserenergie eine Schädigung der Probe vermindert, was vor allem bei der Aufnahme von lebenden Proben vorteilhaft ist.

2.5 Computertomographie

Eine weitere Möglichkeit dreidimensionale Bilder aufzunehmen ist die Computertomographie (CT). Hierbei werden Röntgenbilder des aufzunehmenden Objekts aus vielen unterschiedlichen Richtungen erstellt, die anschließend zu zweidimensionalen Bildern zusammengesetzt werden. Dieser Vorgang wird wiederholt nachdem das Objekt ein Stück in z-Richtung bewegt wurde, wodurch ein Bildstapel, d.h. ein dreidimensionales Bild des aufzunehmenden Objekts, erzeugt wird. In dieser Arbeit werden medizinische Computertomographie-Aufnahmen, sowie industrielle CT-Aufnahmen von antiken Keramikgefäßen betrachtet. Weitere Details zu den Aufnahmen mit archäologischem Inhalt finden sich in [56].

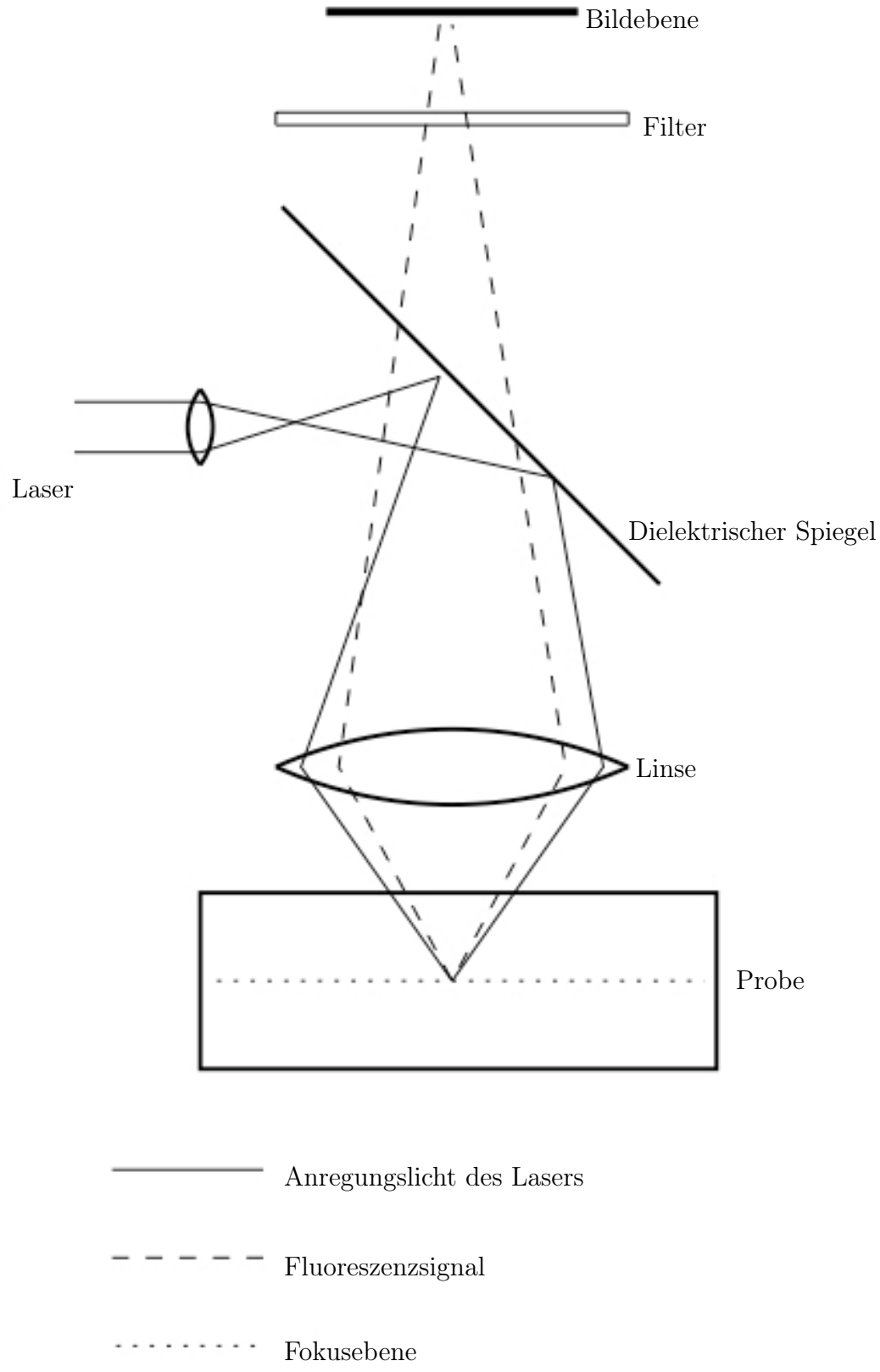
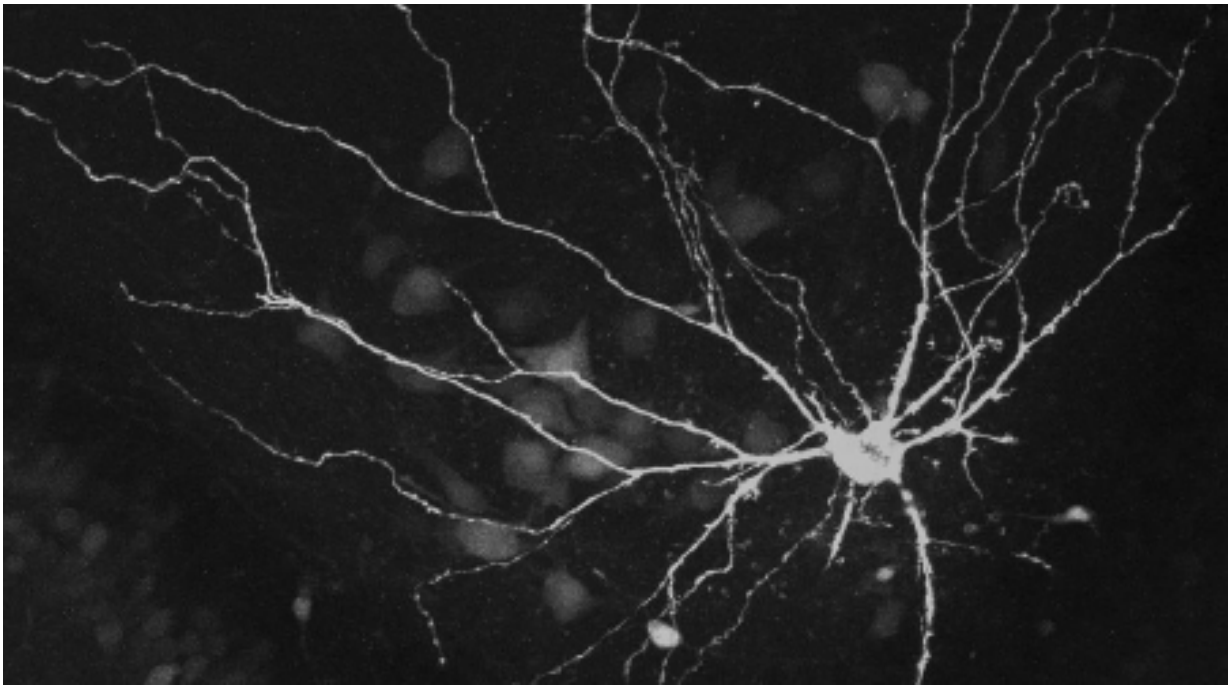


Abbildung 2.1: *Schematischer Aufbau eines Zwei-Photonen-Mikroskops. Quelle:* [101]

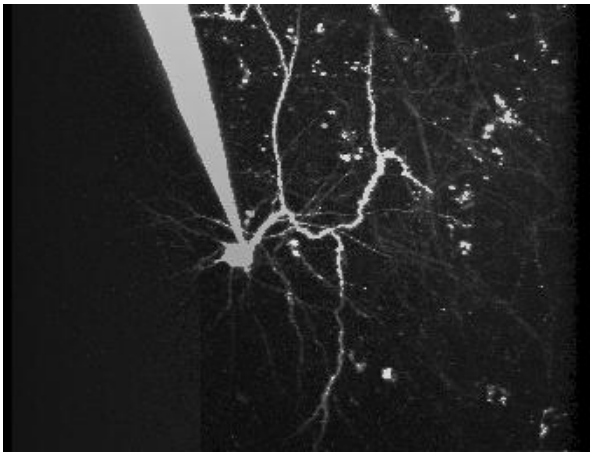
2.6 Resultierende Aufnahmen

Die durch konfokale (Abbildung 2.2a,b) und Zwei-Photonen-Mikroskopie (Abbildung 2.2c) gewonnenen Aufnahmen weisen in x-Richtung und y-Richtung meist die gleiche Auflösung auf, während sich die Auflösung in z-Richtung von der in x-Richtung und y-Richtung üblicherweise unterscheidet. Hierbei entspricht die z-Richtung den unterschiedlichen Fokusebenen (Kapitel 2.1). Üblicherweise werden die Aufnahmen auf das ganzzahlige Grauwertintervall $[0, 255]$ skaliert, was einem Speicherbedarf von einem Byte pro aufgenommenem Bildpunkt, auch *Voxel* genannt, entspricht. Moderne Zwei-Photonen-Mikroskope ermöglichen Aufnahmen einzelner Neuronenzellen mit einer Auflösung von $2048 \times 2048 \times 400$ Voxeln, wobei ein Voxel 200 bis 300 Nanometer in x- und y-Richtung und etwa 1000 Nanometer in z-Richtung groß ist. Bei der in Abbildung 2.2 gezeigten Neuronenzelle ist deutlich ein schlechtes *Signal-zu-Rausch-Verhältnis* zu erkennen, was es schwierig macht, die Bildinformation vom Hintergrundrauschen zu trennen. Zur erfolgreichen Rekonstruktion der Zellen ist daher die Vorverarbeitung der Mikroskopaufnahmen mit einem rauschvermindernden Bildfilter zwingend erforderlich (Kapitel 5 und 6).

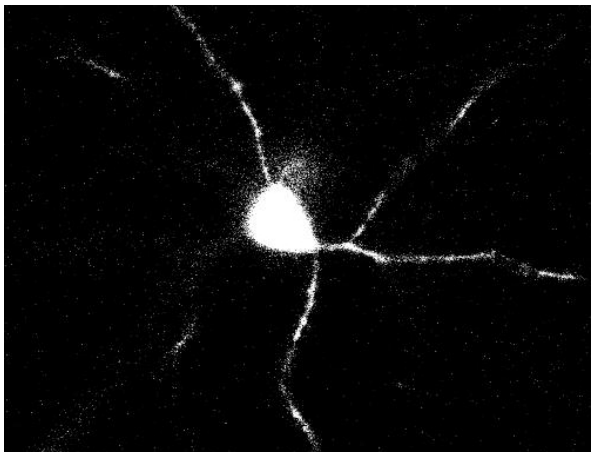
Die Grauwertskala der Ergebnisbilder von Computertomographen ist üblicherweise mit 12-Bit pro Bildpunkt festgelegt, der sogenannten *Hounsfield-Skala* [47]. Zur Verarbeitung in *NeuRA2* werden diese Daten auf eine 8-Bit Auflösung skaliert, wobei Informationen verloren gehen. Bei den in dieser Arbeit behandelten Computertomographie-Aufnahmen von Keramikgefäßen (Abbildung 2.3) stört diese Einschränkung allerdings nicht. Weitere Details zur Aufnahme archäologisch interessanter Objekte mit Hilfe von Computertomographen ist [40] zu entnehmen. Es ist zu beachten, dass die durch Röntgenstrahlung verursachte hohe Strahlendosis die Altersbestimmung der Gefäße beeinträchtigen kann [55], weshalb der Scanvorgang im Protokoll der gescannten Objekte festgehalten werden muss.



(a)



(b)



(c)

Abbildung 2.2: *Verschiedene konfokale und Zwei-Photonen-Mikroskopaufnahmen von Neuronenzellen. Man erkennt deutlich das schlechte* Signal-zu-Rausch-Verhältnis. *(a) und (b) Volumenprojektionen konfokaler Aufnahmen. (c) Schnittebenenbild einer Zwei-Photonen-Aufnahme.*

(a)

(b)

Abbildung 2.3: *Schnittebenenbilder entlang verschiedener Koordinatenachsen eines durch Computertomographie aufgenommenen Keramikgefäßes mit Metallbeschlägen. Zur besseren Darstellung wurde der Kontrast dieser Abbildungen erhöht.*

Kapitel 3

GPU-Programmierung mit CUDA

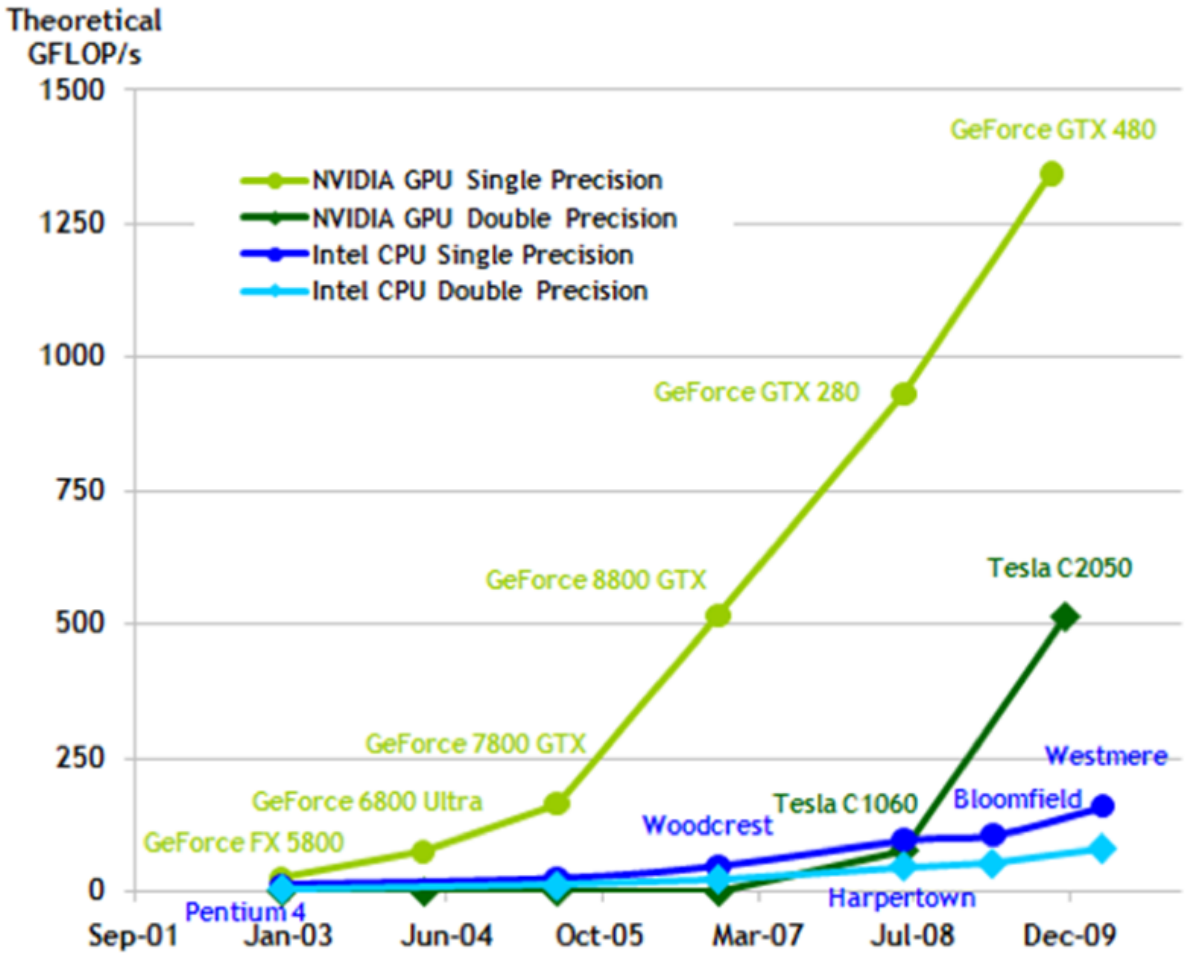


Abbildung 3.1: *Entwicklung der Rechenleistung von* Intel *Standardprozessoren und* Nvidia *Grafikkarten seit 2003. Quelle:* [83]

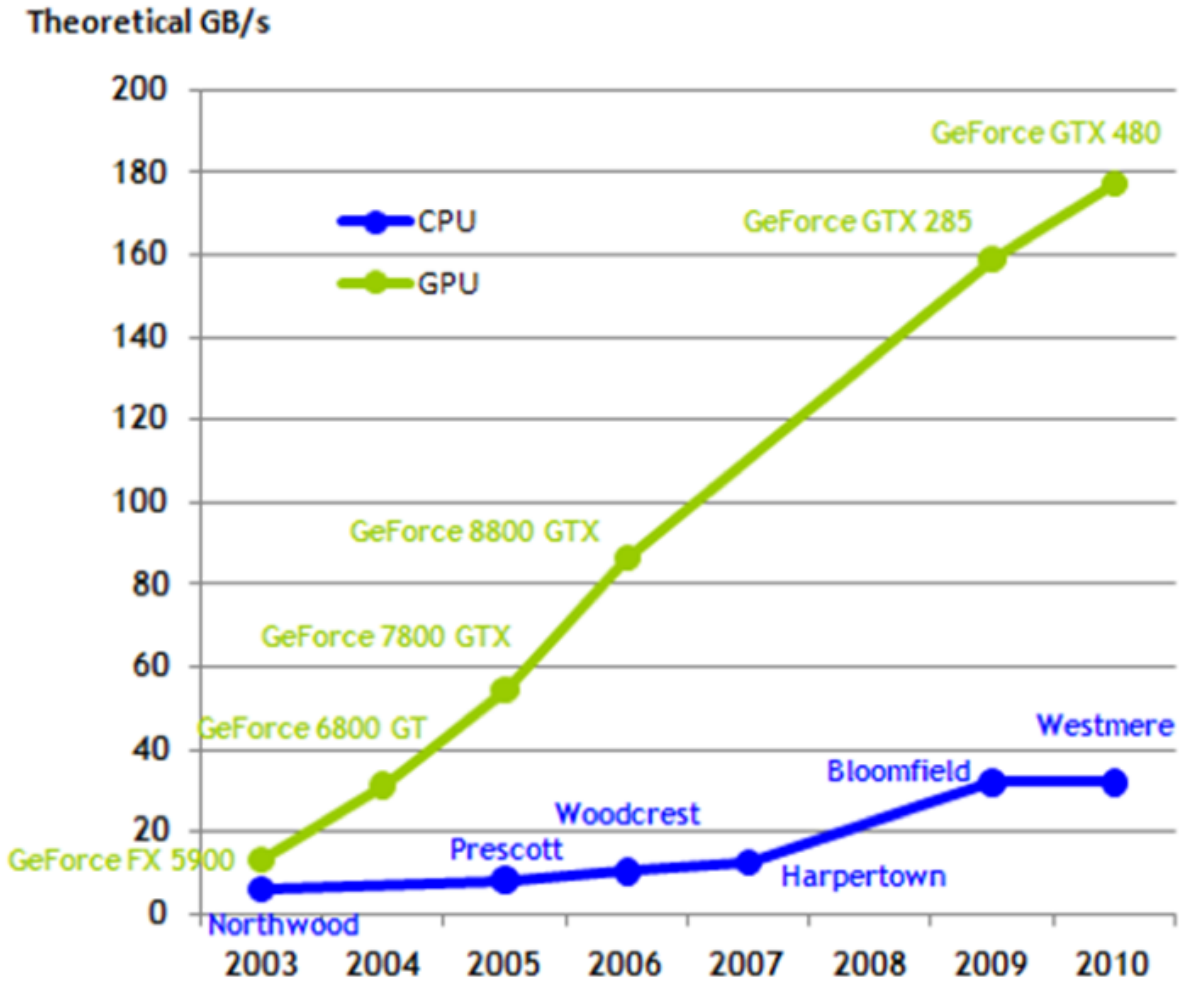


Abbildung 3.2: *Entwicklung der Speicherbandbreite von* Intel *Standardprozessoren und* Nvidia *Grafikkarten in den letzten Jahren. Quelle:* [83]

Da die Taktraten moderner Prozessoren aus physikalischen Gründen kaum mehr zu erhöhen sind, besteht die einzige Möglichkeit moderne Rechner weiter zu beschleunigen darin, Rechnungen parallel auszuführen. Ein Ansatz hierfür sind Multikernprozessoren, die im Prinzip wie ein kleines Cluster vernetzter Hauptprozessoren (CPUs) funktionieren, wobei sich die einzelnen Kerne Bereiche des Hauptspeichers und eventuell des Caches teilen. Ein weiterer Ansatz besteht darin, die Rechenleistung moderner Grafikkarten (GPUs) zu nutzen, die in Sachen Geschwindigkeit moderne CPUs bei weitem übertreffen (Abbildungen 3.1 und 3.2). Die *Nvidia* CUDA Technologie erlaubt die Nutzung der massiven Rechenleistung moderner Grafikkarten, ohne detaillierte Kenntnisse über die Architektur dieser Karten zu besitzen. *Nvidia* bietet hierzu einen C Compiler für *Nvidia* GPUs, einen Laufzeittreiber, eine Laufzeitbibliothek, sowie die zwei High-Level-Bibliotheken *cuBLAS* und *cuFFT* an. *Nvidia* CUDA ist aktuell (Juli 2010) verfügbar für *MacOSX*, *Linux* und

Microsoft Windows XP / Vista / 7 in der Version 3.0. In diesem Kapitel werden das CUDA Programmiermodell sowie einige Subtilitäten der GPU-Programmierung erläutert. Zudem werden am Ende des Kapitels (Abschnitt 3.10) einige Alternativen zur parallelen Programmierung von Grafikhardware und Multikernprozessoren und deren Unterschiede zum CUDA-Programmiermodell vorgestellt.

3.1 Prinzipieller Unterschied zwischen CPU und GPU

Moderne GPUs arbeiten auf elementarster Ebene parallel: Viele arithmetisch-logische Einheiten (ALUs) führen ein Programm auf verschiedenen Daten gleichzeitig aus (Abbildung 3.3). Im Gegensatz zu herkömmlichen Prozessoren verfügen Grafikprozessoren nur über einen kleinen Daten- und einen kleinen Programmcache. Da auf einer GPU jedes Programm parallel auf vielen Datenelementen ausgeführt wird, ist eine komplizierte Ablaufsteuerung wie *Branch-Prediction* [108] nicht vorgesehen. Ebenso wie die Speicherlatenz beim Zugriff auf Daten können solche Verzweigungen im Programmablauf hinter den Berechnungen, die auf anderen Datenbereichen operieren, versteckt werden. Moderne CPUs verfügen hingegen über einen großen Cache und eine aufwändige Programmsteuerung, die versucht durch Umsortieren von Instruktionen und dem richtigen Vorhersagen von Sprungbefehlen, die Ausführung von Programmen zu beschleunigen.

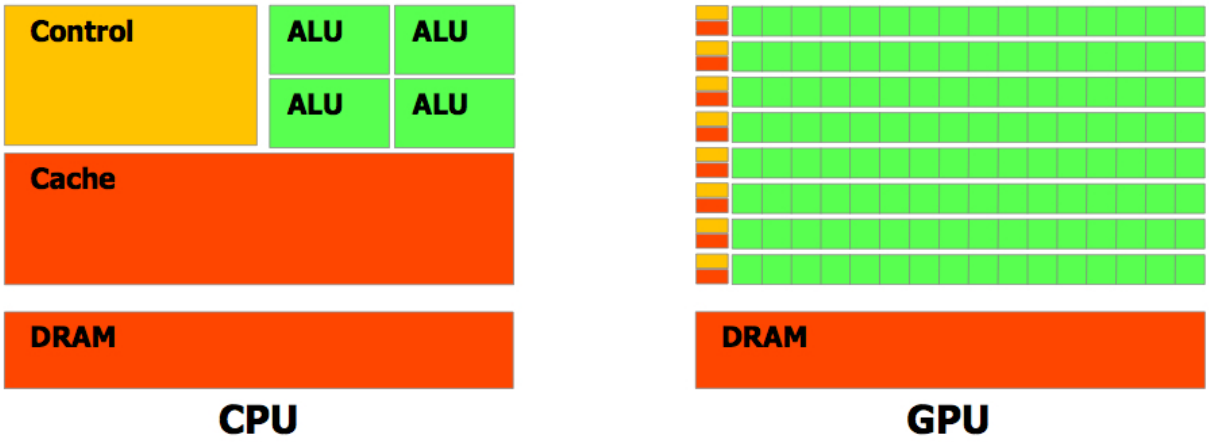


Abbildung 3.3: *Schematische Darstellung des prinzipiellen Unterschiedes zwischen CPU und GPU. Quelle:* [78]

3.2 Historische Entwicklung von GPUs

Grafikprozessoren werden seit Mitte der 1990er Jahre dazu verwendet, dreidimensionale Computergrafiken in Echtzeit zu realisieren. Die Grafikkarten berechnen hierbei aus Geometriedaten (Vertices, Normalen, Farben, Texturkoordinaten, Lichtquellen usw.) und

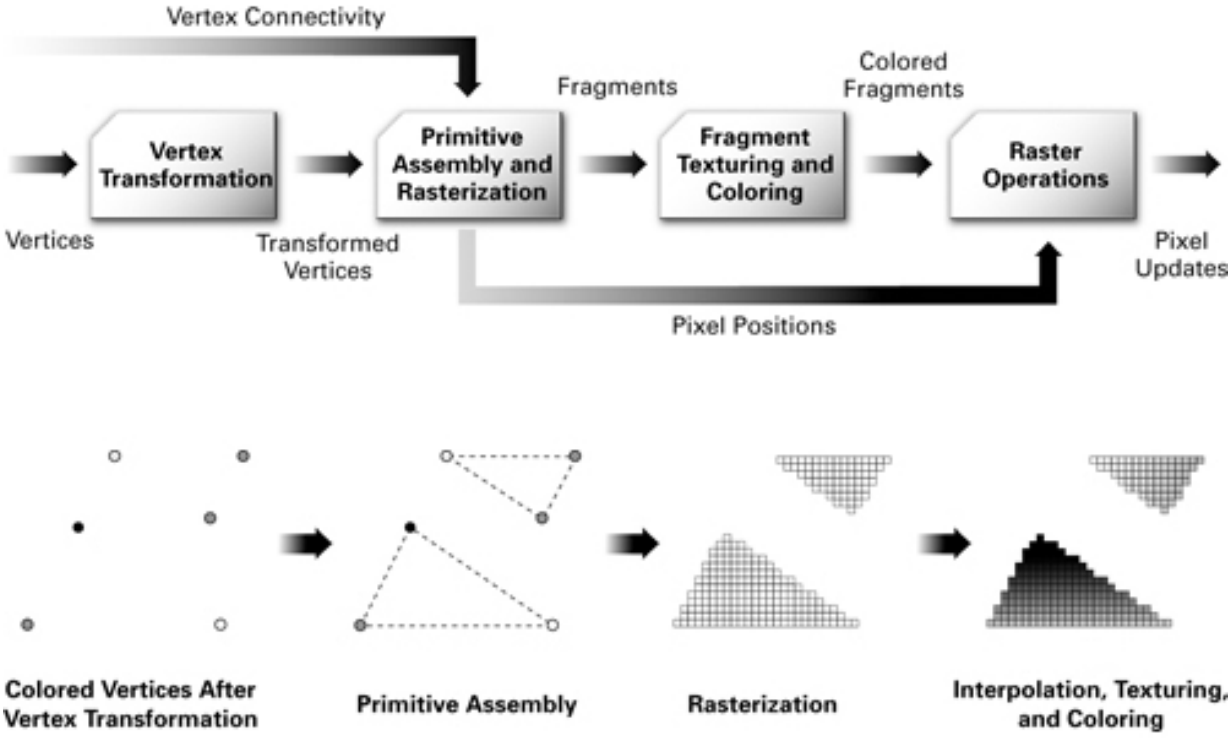


Abbildung 3.4: *Statische Grafikpipeline (bis ca. 2003). Quelle:* [35]

Pixeldaten (z.B. Texturen) das Bild einer virtuellen Szene aus dem Blickwinkel einer sich in der Szene befindlichen virtuellen Kamera [44]. Diese Bilderzeugung war zunächst statisch angelegt und nur durch Festlegen weniger Einstellungsmöglichkeiten manipulierbar. Die wichtigsten Schritte der sogenannten *Grafikpipeline* (Abbildung 3.4) sind die Folgenden:

- *Hauptprogramm:*
  Das auf der CPU laufende *Hauptprogramm* sendet die Geometrie- und Bilddaten an die GPU. Der *Rendervorgang*, wie die automatische Erzeugung eines Bildes genannt wird, wird vom Hauptprogramm gesteuert.

- *Vertexprozessor:*
  Der *Vertexprozessor* berechnet die Monitorkoordinaten der Vertices, ausgehend von ihren Koordinaten im Objektraum und der Position der virtuellen Kamera.

- *Primitive berechnen:*
  Die einzelnen Vertices werden zu primitiven Oberflächen, den sogenannten *Primitiven*, zusammengefügt. Diese Oberflächenstücke werden anschließend in Dreiecke zerlegt.

- *Rasterisierung und Interpolation:*
  Die Dreiecke werden in *Fragmente* zerteilt, deren Größe mit dem Pixelraster des

Monitors übereinstimmt. Farben und Texturkoordinaten werden für jedes einzelne Fragment interpoliert.

- *Fragmentprozessor:*
  Der *Fragmentprozessor* berechnet die Farbe jedes Fragments anhand der im vorherigen Schritt interpolierten Farbe und auf dem Fragment liegender Texturen.

- *Colorbuffer:*
  Das gerenderte Bild wird im *Colorbuffer* gespeichert und kann von dort aus auf dem Monitor angezeigt werden.

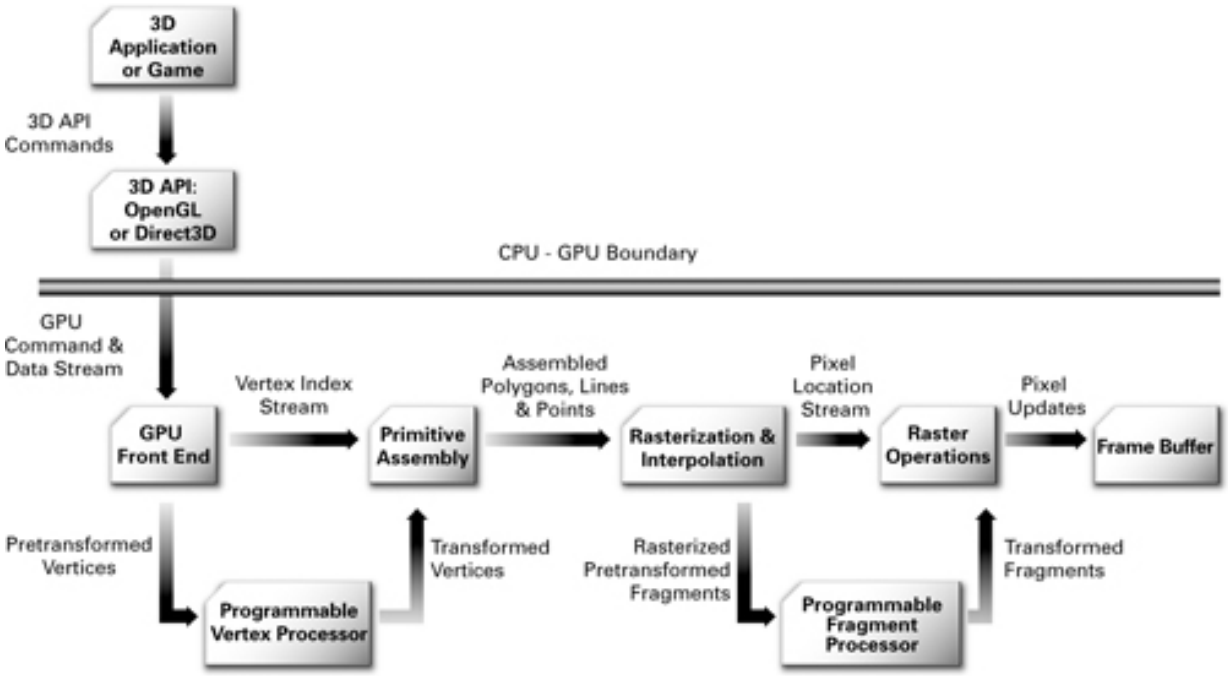


Abbildung 3.5: *Programmierbare Grafikpipeline (ab ca. 2003). Quelle:* [35]

Bei genauerer Betrachtung der Grafikpipeline fällt auf, dass die vom Vertexprozessor ausgeführten Vertexoperationen unabhängig voneinander für alle Vertices gleichzeitig berechnet werden können. Ebenso verhält es sich mit dem Färben der einzelnen Fragmente. Vertexprozessor und Fragmentprozessor lassen sich daher trivial parallelisieren, was bereits seit der ersten Grafikkartengeneration ausgenutzt wird. Um komplexere grafische Effekte in Echtzeit berechnen zu können sind Vertexprozessor und Fragmentprozessor seit 2003, der Geburtsstunde der GPU-Programmierung, frei programmierbar (Abbildung 3.5). Die auf der Grafikkarte ausgeführten Programme werden als *Shader* bezeichnet. Zunächst waren nur einfache *Vertex-Shader* möglich, die in Assembler programmiert werden mussten. Inzwischen gibt es an Hochsprachen angelehnte Programmiersprachen zur Shader Programmierung, wie beispielsweise das von *Nvidia* und *Microsoft* entwickelte *C for graphics*

oder kurz *Cg* [35]. Die hohe Parallelität der Grafikkarten veranlasste zahlreiche Wissenschaftler dazu, diese Programmierbarkeit für vielfältige Berechnungen außerhalb der Computergrafik zu nutzen. In [50] ist beispielsweise eine modifizierte Version des in Kapitel 5 beschriebenen *trägheitsbasierten anisotropen Diffusionsfilters* und dessen Realisierung mit Hilfe von *Fragment-Shadern* zu finden. Weitere Details zur Shaderprogrammierung in Grafikprogrammen und wissenschaftlichen Anwendungen sind [2, 34, 35, 88] zu entnehmen. Das CUDA Programmiermodell abstrahiert von den Konzepten der Grafikarchitektur und bietet die Möglichkeit Programme für moderne GPUs zu entwickeln, ohne detaillierte Kenntnisse über die Grafikpipeline zu besitzen.

3.3 Das CUDA Programmiermodell

Die GPU *(Device)* dient als Koprozessor für die CPU *(Host)*, wobei die GPU über einen eigenen Speicher, den sogenannten *Device Memory*, verfügt. Als *Kernel* werden Programme bezeichnet, die auf dem Device ausgeführt werden. Diese parallelen Kernels werden in vielen konkurrierenden *Threads* gleichzeitig ausgeführt. Hierbei werden bis zu 512 einzelne Threads in *Thread-Blöcken* zusammengefasst. Innerhalb eines Blocks können die einzelnen Threads über den sogenannten *Shared Memory* auf sehr schnelle Weise miteinander kommunizieren. Außerdem können die Threads eines Blocks mit Hilfe des `__syncthreads()` Befehls synchronisiert werden. Die einzelnen Blöcke werden im sogenannten *Gitter* oder *Grid* zusammengefasst (Abbildung 3.6). Damit die Kernels auf eindimensionalen, zweidimensionalen oder dreidimensionalen Daten operieren können, ist die maximale Dimension von Blöcken mit drei, die des Gitters jedoch nur mit zwei festgelegt. Ein dreidimensionales Gitter kann durch eine einfache `for`-Schleife um den Kernel-Aufruf simuliert werden. Jeder Thread und jeder Block erhalten eine eindeutige ID, auf die im Kernel als spezielle eingebaute Variable zugegriffen werden kann. Mit Hilfe von *Thread-ID* und *Block-ID* wird in jedem Thread berechnet, auf welchem Datenfragment er operiert. Ein kleines Beispiel, in dem ein Vektor v mit einem Skalar s multipliziert wird, verdeutlicht die bisher eingeführten Begriffe:

```
// Kernel werden mit dem Bezeichner __global__ versehen:
__global__ void scale_vector(float * v, float s) {

  // Berechnen des Index idx auf dem der aktuelle Thread
  // operiert mit Hilfe der eingebauten Variablen
  // blockIdx, blockDim und threadIdx:
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  // Ausführen der Rechnung:

  v[idx] *= s;
}
```

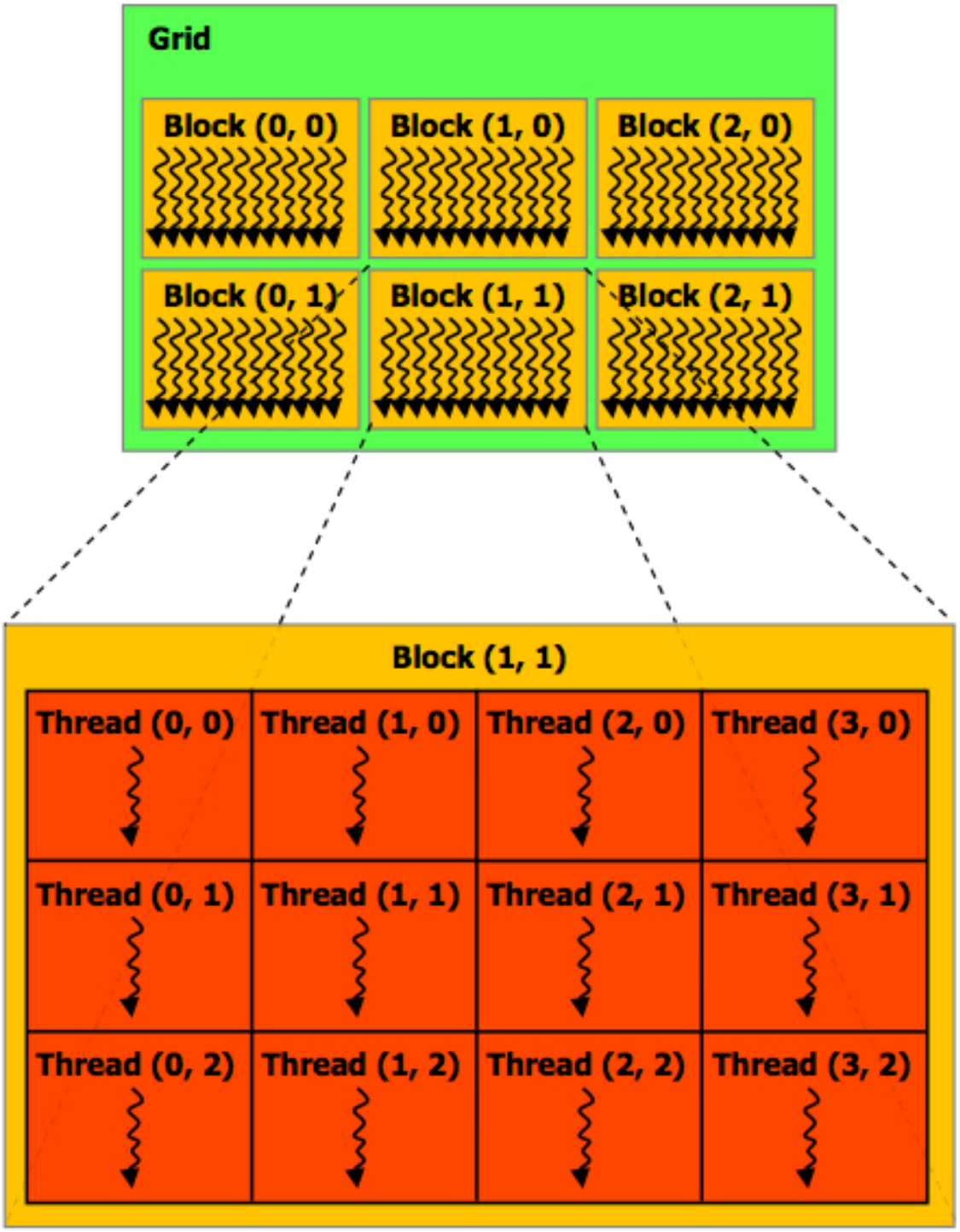


Abbildung 3.6: *Einzelne* Threads *werden in* Blöcken *zusammengefasst, die wiederum im* Gitter *zusammengefasst sind. Quelle:* [78]

```
int main() {

  ...

  // Dimension des Vektors:

  const int N = 1024;


  // Anzahl Threads pro Block:

  const int threads_per_block = 256;


  // Aufruf des Kernels. In den spitzen Klammern <<< , >>>
  // werden die Dimension des Gitters und
  // die Dimension der einzelnen Blöcke angegeben:

  scale_vector <<< N / threads_per_block, threads_per_block >>>
                                                        (v, 0.5);

}
```

Zu beachten gilt, dass die Kernels so gestaltet sein müssen, dass die Threads der einzelnen Blöcke, sowie die einzelnen Blöcke selbst, unabhängig voneinander in beliebiger Reihenfolge oder parallel ausgeführt werden können. Die Dimension des Gitters, d.h. die Anzahl der Blöcke, wird typischerweise durch die Größe der zu bearbeitenden Daten festgelegt, wobei die Dimension der einzelnen Blöcke, d.h. die Anzahl der Threads pro Block, durch die verwendete Hardware limitiert ist.

3.4 Speicherverwaltung

Abbildung 3.7 zeigt die Speicherhierarchie des CUDA Programmiermodells. Wie bereits erwähnt, können die einzelnen Threads innerhalb eines Blocks Daten über den *Shared Memory* schnell austauschen. Einzelne Threads speichern ihre Zwischenergebnisse in den ebenfalls sehr schnellen Registern. Reichen die Register nicht aus, werden Daten in den langsamen *Local Memory* ausgelagert. Ebenfalls langsam ist der *Global Memory*, der allerdings die einzige Möglichkeit zur Kommunikation von Threads aus verschiedenen Blöcken darstellt. Außerdem werden sämtliche Berechnungsergebnisse in den globalen Speicher geschrieben, da nur dieser vom Host gelesen werden kann. Benötigt ein Kernel Eingabedaten, die selbst nicht verändert werden, so bietet sich die Verwendung des *Constant Memorys* und des *Texture Memorys* an. Diese sind schreibgeschützt und können durch Kernel-Aufrufe

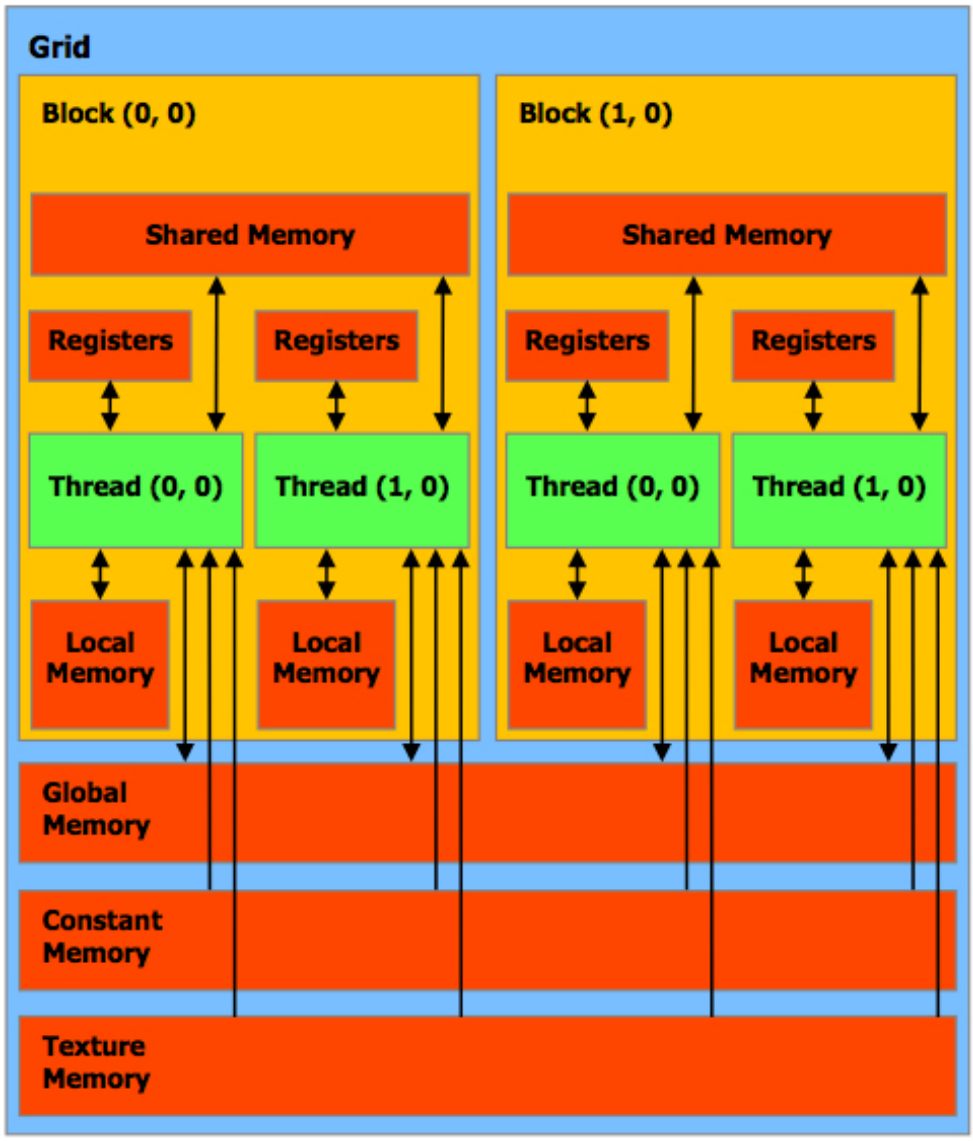


Abbildung 3.7: *CUDA-Speicherhierarchie. Quelle:* [73]

nicht verändert werden. Daher können die dort gespeicherten Daten gecacht werden und sind somit schnell verfügbar, falls benachbarte Threads benachbarte Daten aus diesen Speicherbereichen anfordern. Der Host kann den globalen Speicher sowohl schreiben als auch lesen und in den konstanten Speicher und in den Texturspeicher schreiben (Abbildung 3.8). Auf die anderen Speicherbereiche hat der Host keinen Zugriff. Es können maximal acht Threads pro Block gleichzeitig ausgeführt werden (vgl. Kapitel 3.5). Daher kann ein Großteil der Zeit, der für zeitaufwändige Zugriffe auf den globalen Speicher benötigt wird (Tabelle 3.1), dadurch überdeckt werden, dass Threads, die einen Zugriff auf den Speicher gestartet haben, zunächst nicht weiter beachtet und Instruktionen anderer Threads in der Zwischenzeit ausgeführt werden.

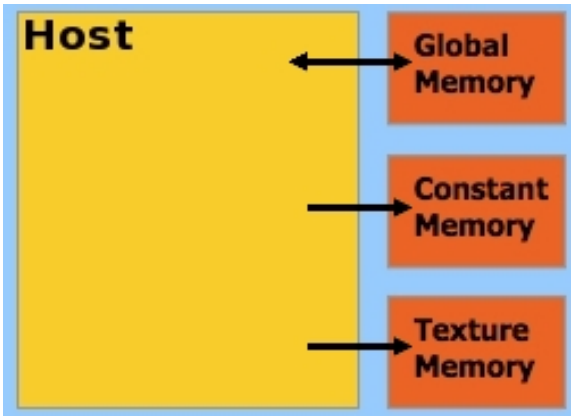


Abbildung 3.8: *Der Host kann den globalen Speicher schreiben und lesen, sowie konstanten Speicher und Texturspeicher schreiben. Quelle:* [73]

Speichertyp	Speicherzugriff	Zugriffsgeschwindigkeit
Register	Lesen und schreiben pro Thread	Ein Taktzyklus
Shared Memory	Lesen und schreiben pro Block	Ein Taktzyklus
Local Memory	Lesen und schreiben pro Thread	400 bis 600 Taktzyklen
Global Memory	Lesen und schreiben pro Grid	400 bis 600 Taktzyklen
Constant Memory	Lesen pro Grid	1 bis 100 Taktzyklen
Texture Memory	Lesen pro Grid	1 bis 100 Taktzyklen

Tabelle 3.1: *Zugriffsmöglichkeiten und Zugriffsgeschwindigkeiten der verschiedenen Speichertypen.*

3.5 Hardwareimplementierung

Jedes Device besteht aus N Multiprozessoren, die jeweils M Threads gleichzeitig ausführen können (Abbildung 3.9). Jeder Multiprozessor bearbeitet einen oder mehrere Blöcke. Hierbei wird die Verteilung der Blöcke und der Threads auf die einzelnen Prozessoren automatisch von der CUDA Laufzeitbibliothek bzw. dem CUDA Treiber übernommen (Abbildung 3.10). Aktuell gilt bei allen CUDA-fähigen Grafikkarten $M = 8$. Die Anzahl der Multiprozessoren N variiert jedoch von 1 (GeForce 9200) bis 30 (GTX 285).

3.6 CUDA-API

Das CUDA-API (Application Programming Interface) erweitert die Sprache C um verschiedene Bezeichner von Funktionen und Variablen, sowie die Möglichkeit Kernels aufzurufen. Zudem enthält das CUDA-API eine Laufzeitbibliothek, die Funktionen für den Host und das Device sowie die zwei High-Level-Bibliotheken *cuBLAS* und *cuFFT* bereitstellt, und den CUDA-Debugger [75].

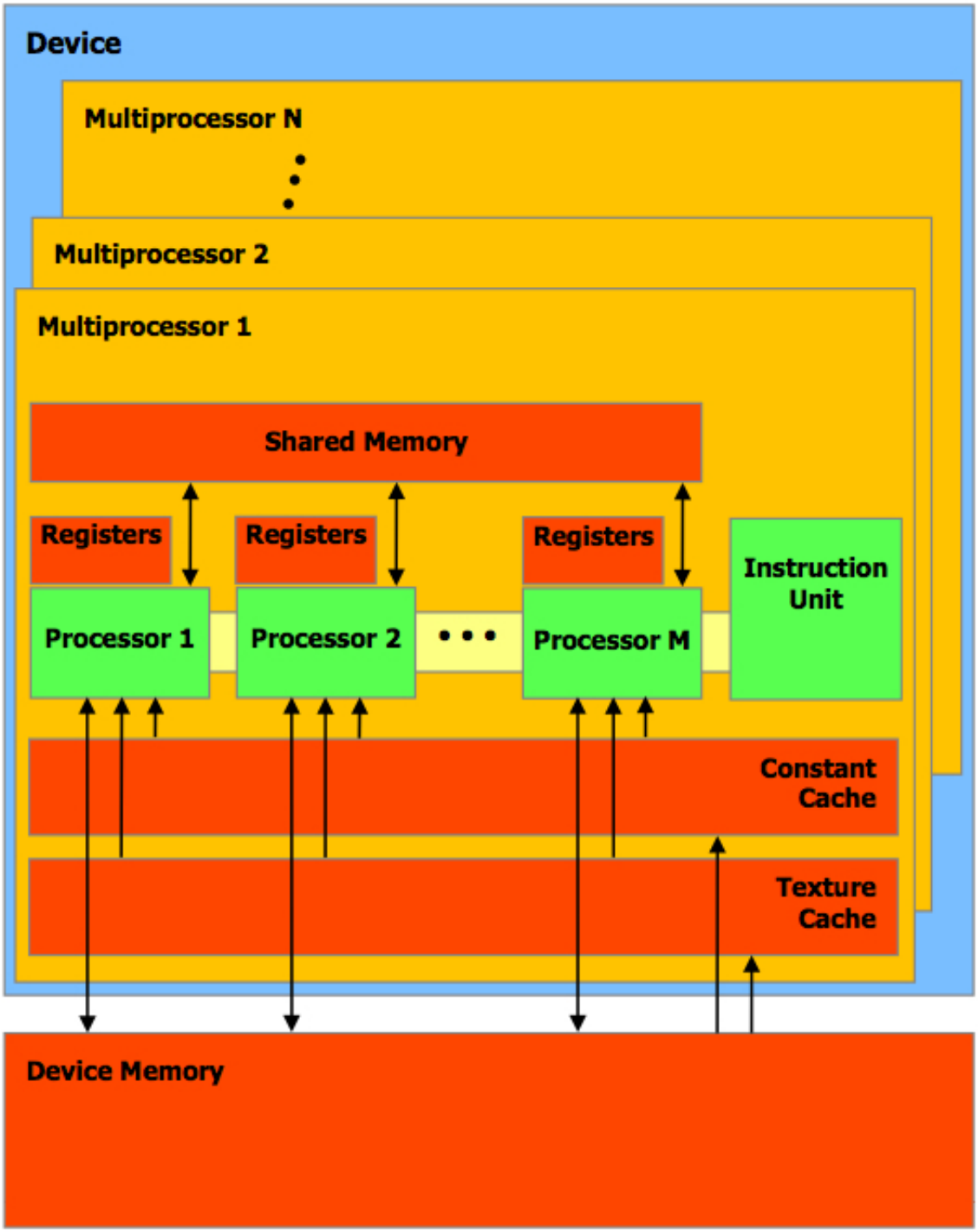


Abbildung 3.9: *CUDA-Hardwareimplementierung. Quelle:* [78]

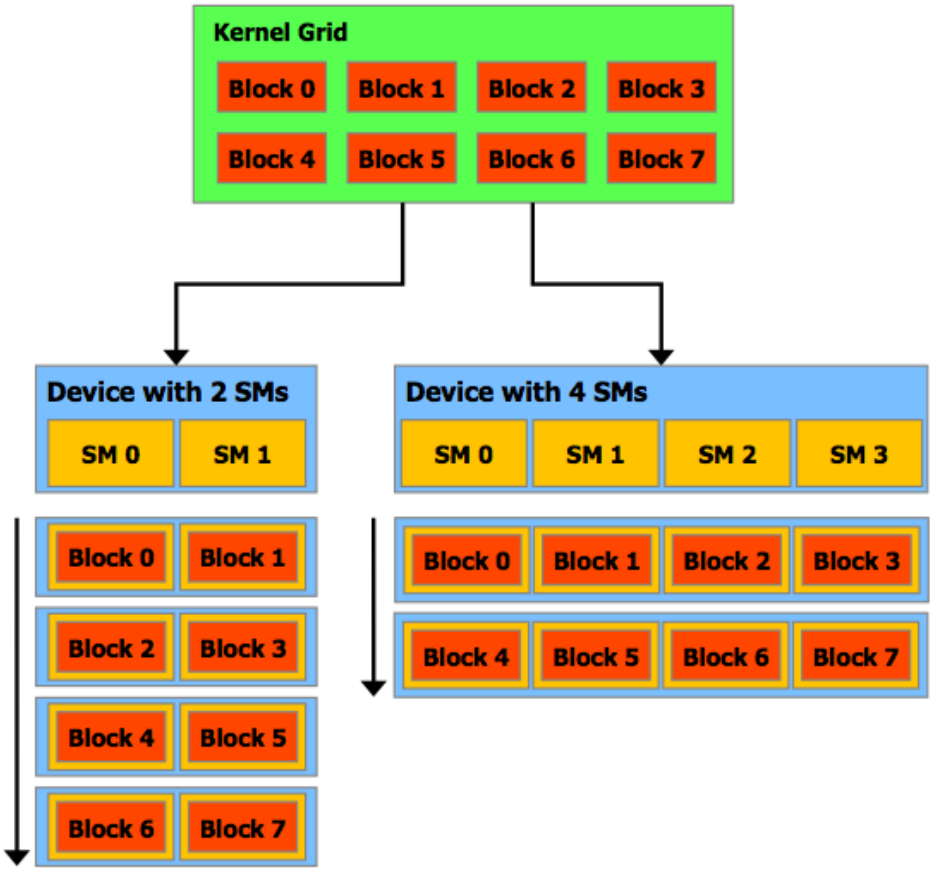


Abbildung 3.10: *Die CUDA Laufzeitbibliothek und der CUDA Treiber verteilen die zu bearbeitenden Blöcke automatisch auf die Multiprozessoren. Je mehr Multiprozessoren vorhanden sind, desto schneller ist die Ausführung des CUDA Programms. Quelle:* [81]

3.6.1 Bezeichner von Funktionen

Kennzeichner	Ausgeführt auf	Aufrufbar von
`__global__`	Device	Host
`__device__`	Device	Device
`__host__`	Host	Host

Tabelle 3.2: *Bezeichner von Funktionen.*

Der Bezeichner `__global__` kennzeichnet Funktionen, die auf dem Device ausgeführt und vom Host aufgerufen werden: Die *Kernels*. Der Bezeichner `__host__` erscheint auf den ersten Blick überflüssig, da er Funktionen kennzeichnet die vom Host aufgerufen und vom Host ausgeführt werden. Er kann allerdings mit dem Bezeichner `__device__` kombiniert werden,

um Code zu erzeugen, der sowohl auf dem Host, als auch auf dem Device ausgeführt und entsprechend aufgerufen wird. Zu beachten ist, dass `__global__`-Funktionen immer vom Typ `void` sind und `__device__`-Funktionen automatisch als `inline` interpretiert werden.

3.6.2 Bezeichner von Variablen

Kennzeichner	Speicherort	Zugreifbar von	Lebensdauer
`__device__`	Global Memory	Device & Host	Hauptprogramm
`__constant__`	Constant Memory	Device & Host	Hauptprogramm
`__shared__`	Shared Memory	Device	Kernel-Aufruf

Tabelle 3.3: *Bezeichner von Variablen.*

Tabelle 3.3 listet die Bezeichner für Variablen, ihren Speicherort, ihre Zugreifbarkeit und ihre Lebensdauer auf.

3.6.3 Kernel Aufrufe

Bei jedem Aufruf eines Kernels muss die Dimension des Gitters, sowie die Dimension der einzelnen Blöcke übergeben werden. Im obigen Beispiel ist dieses Vorgehen für ein eindimensionales Gitter mit eindimensionalen Blöcken aufgeführt. Mehrdimensionale Gitter und Blöcke werden mit Hilfe des CUDA eigenen Datentyps `dim3` erzeugt.

```
__global__ void some_kernel(...) {

  // Berechnen der Indizes der zu bearbeitenden Daten:
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;

  ...

}

int main() {

  // Zweidimensionale Daten der Größe data_size pro Dimension
  // sollen bearbeitet werden:
  const int data_size = 1024;

  // Jeder Block enthält 16 Threads in jeder Dimension:
  const int block_size = 16;
```

```
  // Erzeugen der CUDA eigenen Datentypen mit Informationen über
  // Block- und Gitterdimension:
  dim3 block_dimension(block_size, block_size, 1);
  dim3 grid_dimension(data_size / block_dimension.x,
                      data_size / block_dimension.y, 1);

  // Aufruf des Kernels:
  some_kernel <<< grid_dimension, block_dimension >>> (...);

}
```

Bezeichnung	Inhalt
gridDim	Dimension des Grids
blockIdx	Block Index im Grid
blockDim	Dimension der Blöcke
threadIdx	Thread Index im Block

Tabelle 3.4: *Spezielle eingebaute Variablen zur Indizierung der einzelnen Threads.*

Wie aus den beiden Beispielen ersichtlich, wird die Indizierung der zu bearbeitenden Daten innerhalb der Threads durch spezielle, eingebaute Variablen berechnet (Tabelle 3.4). Alle diese eingebauten Variablen sind vom Typ `dim3`. Wertzuweisungen und Zugriffe auf die Adressen dieser Variablen sind nicht erlaubt. Beim Aufruf eines Kernels können in den spitzen Klammern weitere Argumente, wie beispielsweise die Größe von dynamisch alloziertem Shared Memory übergeben werden. Für weitere Details sei auf [78] verwiesen.

3.6.4 Laufzeitbibliothek

Die CUDA Laufzeitbibliothek gliedert sich in Host Komponente, Device Komponente und gemeinsame Komponente.

Host Komponente

Die Host Komponente enthält Funktionen zur Verwaltung des Devices und des Device Speichers. Diese Funktionen sind eng an die C–Syntax angelegt, was durch ein Beispiel verdeutlicht werden soll:

```
// Host Code:

float * x = get_data_from_somewhere();
float * cuda_x = NULL;
```

```
int size = sizeof(float) * 1024;

// Speicher auf dem Device allozieren:
cudaMalloc((void **) & cuda_x, size);

// Daten vom Host zum Device kopieren:
cudaMemcpy(cuda_x, x, size, cudaMemcpyHostToDevice);

// Kernel-Aufruf, der die unter cuda_x gespeicherten Daten
// manipuliert:
call_some_kernel <<< ... , ... >>> (cuda_x, ...);

// Zurückkopieren der manipulierten Daten zum Host:
cudaMemcpy(x, cuda_x, size, cudaMemcpyDeviceToHost);

// Freigeben des Device-Speichers:
cudaFree(cuda_x);
```

Erwähnenswert ist zudem die Funktion `cudaGetDeviceProperties()`, mit deren Hilfe Eigenschaften des verwendeten Device, wie beispielsweise der maximal verfügbare Speicher, ausgelesen werden können. Ebenfalls wichtig sind die Funktionen `cudaGetDeviceCount()`, `cudaGetDevice()` und `cudaSetDevice()`, mit deren Hilfe mehrere Devices von einem Host angesprochen werden können. In Kapitel 3.8.3 findet sich eine detaillierte Beschreibung zur *Multi-GPU-Programmierung*.

Device Komponente

Die Device Komponente beinhaltet mathematische Funktionen verschiedener Genauigkeit, den Befehl `__syncthreads()` zur Synchronisierung aller Threads innerhalb eines Blocks sowie atomare Funktionen, die Operationen ohne Beeinflussung anderer Threads ausführen. Falls möglich, sollte aus Effizienzgründen auf atomare Funktionen verzichtet werden (vgl. Kapitel 5.7.3). Eine detaillierte Auflistung aller Funktionen der Device Komponente der CUDA Laufzeitbibliothek ist in [78] zu finden.

Gemeinsame Komponente

Die gemeinsame Komponente enthält eingebaute Vektordatentypen wie `dim3`, `float2`, `float3`, `float4`, `int2`, `int3` und `int4`. Nicht initialisierte Komponenten von Variablen des Typs `dim3` werden automatisch auf den Wert 1 gesetzt. Eine komplette Auflistung aller eingebauten Datentypen findet sich in [78]. Die einzelnen Komponenten dieser eingebauten Datentypen können mit den Suffixen

`.x` `.y` `.z` `.w`

angesprochen werden. Ein aus den Shadersprachen bekanntes *Swizzling* [35] ist nicht vorgesehen. Für jeden dieser Datentypen existiert mit `make_<typename>` ein entsprechender Konstruktor. Die gemeinsame Komponente stellt mit der Funktion `clock_t clock()` ebenfalls eine Möglichkeit zur Messung von benötigten Taktzyklen von Host und Device zur Verfügung. Vektorwertige Rechenoperationen sind in CUDA selbst nicht vorgesehen. Die dem CUDA SDK beiliegende Datei `cutil_math.h` enthält allerdings einige der aus *Cg* [35] bekannten Funktionen zum Rechnen mit Vektordatentypen.

3.6.5 High-Level-Bibliotheken

Das CUDA-API stellt die zwei High-Level-Bibliotheken *cuBLAS* und *cuFFT* zur Verfügung. Diese sind Implementierungen von *BLAS* (Basic Linear Algebra Subprograms) und *FFT* (Fast Fourier Transformation) auf Ebene des CUDA APIs. Allerdings ist die Implementierung auf einzelne Devices beschränkt. Die beiden Bibliotheken werden in dieser Arbeit nicht verwendet und sind nur der Vollständigkeit halber aufgeführt. Für eine detaillierte Beschreibung sei daher auf [74, 77] verwiesen.

3.7 Entwickeln mit CUDA

Zur leichten Einarbeitung in CUDA wird von *Nvidia* ein Software Development Kit (SDK) mit vielen Programmbeispielen zur Verfügung gestellt. Die bereitgestellten Makefiles erleichtern das Debuggen der GPU-Programme, wenn man den Code mit den Flags `dbg=1` für den Debugmodus bzw. `emu=1` für den Emulationsmodus übersetzt. Im Debugmodus übersetzte Programme erleichtern die Suche nach Fehlern bei Kernel-Aufrufen und Speicherzugriffsverletzungen. Im Emulationsmodus übersetzte Programme werden auf dem Host ausgeführt, wodurch ihre semantische Korrektheit durch Verwendung von `printf(...)` und anderen Befehlen, die nicht auf dem Device ausgeführt werden können, geprüft werden kann. Ab Version 2.2 stellt *Nvidia* zudem einen Debugger zum Debuggen der Kernels zur Verfügung [75].

3.8 CUDA in der Praxis

Guter paralleler GPU-Code kann einen Laufzeitunterschied von bis zu zwei Größenordnungen gegenüber seriellem CPU-Code erreichen. Ein schlechtes Programmdesign kann diesen Laufzeitgewinn jedoch schnell zunichte machen.

3.8.1 Datentransfer zwischen Host und Device

Optimalerweise sollten die Eingangsdaten zu Beginn einer Rechnung auf das Device geladen, dort bearbeitet und anschließend die Zieldaten zurück in den Hauptspeicher kopiert werden (Abbildung 3.11). Die eigentliche Rechnung wird dabei vom Host gesteuert, ein

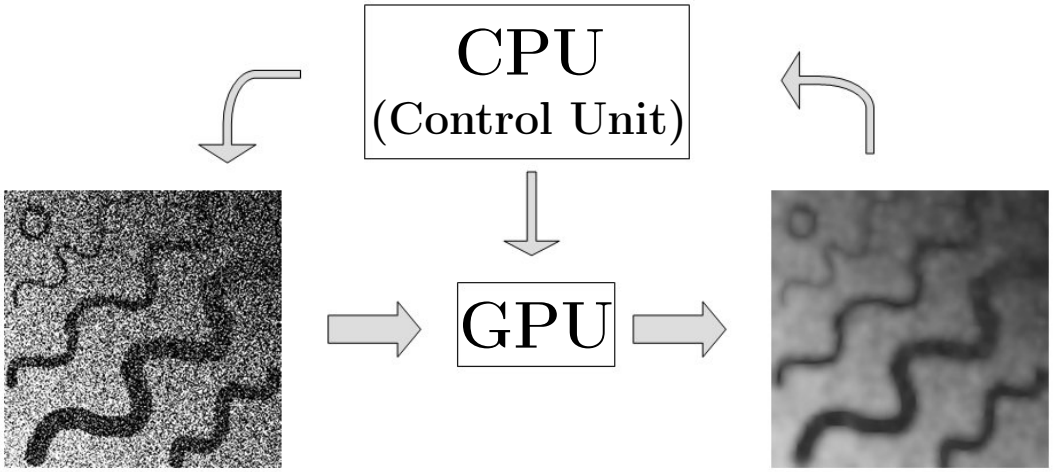


Abbildung 3.11: *Der Datentransfer zwischen Host und Device sollte so gering wie möglich sein. Quelle:* [50]

weiterer Datenaustausch zwischen Host und Device findet nicht statt. Auf dem Device selbst können Daten problemlos hin- und herkopiert werden, da Grafikkarten über eine sehr hohe Speicherbandbreite verfügen (Abbildung 3.2). Bevor große Datenmengen mehrfach zwischen Host und Device hin- und herkopiert werden ist es ratsam, auch serielle Codeteile auf dem Device auszuführen. Selbstverständlich ist solch ein Vorgehen nicht für jeden Algorithmus möglich, wodurch die Performance der Grafikkarte dann nicht optimal ausgenutzt wird (vgl. Kapitel 8).

3.8.2 Speichernutzung

Der Zugriff auf den globalen Speicher sollte so gering wie möglich gehalten werden. Optimal ist eine Schreiboperation pro Thread oder höchstens eine Lese- und eine Schreiboperation pro Thread. Redundante Berechnungen sind oftmals einem globalen Speicherzugriff vorzuziehen. Die Benutzung von konstantem Speicher und Texturspeicher kann die Laufzeit drastisch verbessern. Ebenso kann es sinnvoll sein, Daten im Shared Memory zu cachen, falls benachbarte Threads auf benachbarte Daten zugreifen müssen. Alle diese Techniken werden im späteren Verlauf dieser Arbeit genauer betrachtet und entsprechende Laufzeituntersuchungen durchgeführt.

Konstanter Speicher

Die Größe des benötigten konstanten Speichers muss bereits zur Kompilierzeit des Programms bekannt sein. Diese Speicherbereiche können dabei direkt mit Daten gefüllt werden

```
__constant__ int constant_array[8] = {0, 1, 2, 3, 4, 5, 6, 7};
```

oder zunächst nur deklariert werden

```
__constant__ float constant_data[1024];
```

und zur Laufzeit vom Host durch den Befehl

```
cudaMemcpyToSymbol(constant_data,
                   (void *) data,
                   1024 * sizeof(float),
                   0,
                   cudaMemcpyHostToDevice);
```

mit den im Feld `data` gespeicherten Daten initialisiert werden. Ein Nachteil des konstanten Speichers ist, dass er über die gesamte Programmlaufzeit alloziert bleibt und diese Bereiche des Device Speichers nicht anderweitig verwendet werden können. Zudem muss die Größe des benötigten Speichers bereits zur Kompilierzeit bekannt sein. Möchte man die Vorteile des gecachten Zugriffs mit dynamischer Speicher-Allokation verbinden, bietet sich die Verwendung des Texturspeichers an.

Texturspeicher

Sämtliche Eingabedaten, die in den Kernels nur gelesen aber nicht verändert werden, sollten als Texturen gebunden werden, damit ein gecachter Zugriff auf diese Daten möglich ist. Die einzelnen Bildpunkte einer Textur werden als *Texel* bezeichnet. Zunächst muss die Textur mittels

```
texture<float, 1, cudaReadModeElementType> tex;
```

global deklariert werden. Der Host kann dann durch

```
cudaBindTexture(0, tex, cuda_data, size * sizeof(float));
```

die unter `cuda_data` gespeicherten Daten der Größe `size` im Device Speicher an die eindimensionale Textur `tex` binden. Mit dem Befehl

```
cudaUnbindTexture(tex);
```

kann der Host die gebundene Textur wieder freigeben. Das Device kann durch den Befehl

```
tex1Dfetch(tex, index);
```

auf das durch `index` indizierte Datenfeld der als Textur gebundenen Daten zugreifen. Texturen, die auf diese Weise den Zugriff auf lineare Speicherbereiche beschleunigen, unterliegen allerdings einigen Einschränkungen:

- Es sind nur eindimensionale Texturen möglich.
- Es ist keine Texturfilterung möglich.

- Die Texturen können nur mit nicht normalisierten, ganzzahligen Texturkoordinaten adressiert werden (siehe unten).
- Die verschiedenen Adressierungsmodi (siehe unten) werden nicht unterstützt.

Um diese Eigenschaften beim Texturzugriff dennoch nutzen zu können, müssen durch CUDA-Arrays allozierte Speicherbereiche als Texturen gebunden werden. Zunächst wird global mittels

```
texture<float, 2, cudaReadModeElementType> tex;
```

eine zweidimensionale Textur mit Einträgen vom Typ `float` deklariert. Mit

```
cudaArray * array = NULL;
cudaChannelFormatDesc cf = cudaCreateChannelDesc<float>();
cudaMallocArray(&array, &cf, width, height);
```

kann Speicher mit Hilfe eines CUDA-Arrays auf dem Device alloziert werden. Die Variablen `width` und `height` bezeichnen hierbei Länge und Höhe des zweidimensionalen Datenfelds. Mit

```
cudaFreeArray(array);
```

wird der von einem CUDA-Array allozierte Speicher wieder freigegeben. Der Befehl

```
cudaMemcpyToArray(array,
                  0,
                  0,
                  data,
                  width * height * sizeof(float),
                  cudaMemcpyHostToDevice);
```

kopiert die im Feld `data` abgelegten Daten in das CUDA-Array. Mittels

```
cudaBindTextureToArray(tex, array);
```

kann die oben deklarierte Textur an das zweidimensionale Feld gebunden werden. Über die Attribute

```
tex.normalized = 1;
tex.addressMode[0] = cudaAddressModeClamp;
tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear;
```

können nun die verschiedenen Zugriffsmodi der Textur gesteuert werden, die im Folgenden erläutert sind:

- *Normalisierte Texturkoordinaten:*
  Die Texel lassen sich mit ganzzahligen Werten aus dem Bereich $[0, \texttt{width}) \times [0, \texttt{height})$ adressieren. Bei der Verwendung von normalisierten Texturkoordinaten werden die Texel durch Fließkommazahlen aus dem Bereich $[0, 1) \times [0, 1)$ adressiert. Dies hat den Vorteil, dass die verwendeten Texturkoordinaten unabhängig von der eigentlichen Texturgröße sind und ermöglicht so die Verwendung von Texturen unterschiedlicher Detailstufen bei gleichbleibender Adressierung.

- *Adressierungsmodus:*
  Der Adressierungsmodus kann auf die Werte `clamp` und `wrap` eingestellt werden. Hierbei bedeutet `clamp`, dass Texturkoordinaten außerhalb des Bereichs von $[0, N)$ auf 0 für kleinere Werte und $N - 1$ für größere Werte gesetzt werden. Entsprechend werden bei der Verwendung von normalisierten Texturkoordinaten diese auf das Intervall $[0, 1)$ gesetzt. Der `wrap`-Modus ermöglicht die Verwendung von periodischen Texturen und kann nur für normalisierte Texturkoordinaten verwendet werden. Im `wrap`-Modus wird beispielsweise eine Texturkoordinate mit Wert 1.25 auf 0.25 und eine Texturkoordinate mit Wert -1.25 auf 0.75 gesetzt.

- *Texturfilterung:*
  Texturfilterung ist ebenfalls nur für normalisierte Texturkoordinaten möglich. Bei der linearen Filterung (`cudaFilterModeLinear`) werden die um die Texturkoordinaten liegenden Texel linear interpoliert. Schaltet man die Filterung aus (`cudaFilterModePoint`), wird jeweils das Texel ausgewählt, das am nächsten an den eingegebenen Texturkoordinaten liegt.

Texturen, die an über CUDA-Arrays allozierten Speicher gebunden sind, können entsprechend der Dimension der Texturen mit den Befehlen

```
float a = tex1D(tex_1d, x);
float b = tex2D(tex_2d, x, y);
float c = tex3D(tex_3d, x, y, z);
```

von dem Device ausgelesen werden. Bei der Verwendung von CUDA-Arrays ist zu beachten, dass Kernels diese Speicherbereiche nicht schreiben können. Soll das Ergebnis eines Kernel-Aufrufs über ein CUDA-Array als Textur gebunden werden, so muss dieses Ergebnis von dem Kernel zunächst in einem linearen Speicherbereich des globalen Speichers abgelegt werden und anschließend mit Hilfe der Befehle `cudaMemcpyToArray` bzw. `cudaMemcpy2DToArray` in das vorher allozierte Array kopiert werden. Natürlich können nicht nur Texturen mit Fließkommawerten verwendet werden. Neben ganzzahligen Werten ist die Verwendung der oben eingeführten Vektorvariablen möglich.

3.8.3 Multi-GPU-Programmierung

Von einem Host können mehrere Devices gleichzeitig verwendet werden. Ein Datenaustausch zwischen den Devices ist allerdings nur über den Speicher des Hosts möglich. Jedes

Device muss durch einen separaten Thread des Hauptprogramms angesprochen werden. Folgender, an das Multi-GPU Beispiel in [76] angelehnter Beispielcode, verdeutlicht das Vorgehen bei der Multi-GPU-Programmierung. Zunächst werden mit dem Befehl

```
#include <multithreading.h>
```

die entsprechenden Hilfsfunktionen aus dem CUDA Software Development Kit [76] eingebunden. Durch

```
int device_count = 0;
cudaGetDeviceCount(&device_count);
```

wird die Anzahl verfügbarer Devices ausgelesen. Anschließend wird mit

```
CUTThread * threads =
          (CUTThread *) malloc (sizeof(CUTThread) * device_count);
```

Speicherplatz für die verschiedenen CPU-Threads alloziert, die nötig sind, um die verschiedenen Devices anzusteuern. Die einzelnen CPU-Threads werden durch

```
CUT_THREADPROC cpu_thread(int * device) {

  int device_id = *device;
  cudaSetDevice(device_id);

  // Hier wird der Code eingefügt, der auf dem Device
  // mit der Nummer device_id ausgeführt wird.

  ...

  CUT_THREADEND;
}
```

definiert. Aufgerufen werden die einzelnen CPU-Threads wie folgt:

```
int thread_id[device_count];

for(int i = 0; i < device_count; i++) {
  thread_id[i] = i;
  threads[i] =
      cutStartThread((CUT_THREADROUTINE)cpu_thread,
                                  (void *) & thread_id[i]);
}
```

Abschließend wartet der Host bis alle CPU-Threads fertig bearbeitet wurden

```
cutWaitForThreads(threads, device_count);
```

und gibt den für die CPU-Threads allozierten Speicherbereich mittels

```
free(threads);
```

wieder frei. Sinn der Multi-GPU-Programmierung ist es, große Datenmengen in kleine Portionen zu zerteilen, die dann separat auf verschiedenen Devices bearbeitet werden können. Übersteigt die Anzahl der Datenpakete die Anzahl der verfügbaren Devices, müssen einige Devices mehrere der Datenpakete bearbeiten. Diese Datenpakete werden dann entsprechend der Verfügbarkeit der Devices auf diese verteilt. Folgender Code überprüft, ob noch Datenpakete zu bearbeiten sind und ob Devices verfügbar sind und verteilt gegebenenfalls die nächsten zu bearbeitenden Daten auf das freie Device. Zunächst werden die einzelnen CPU-Threads angepasst, sodass sie über ein Flag steuern, welche Devices von ihnen aktuell verwendet werden.

```
// Speichert, welches Device frei und welches beschäftigt ist:
int multigpu_working[MAX_GPUS];

// Zuordnung von CPU-Threads und Devices:
int thread_id[MAX_GPUS];

// Speichert, welche Daten von welchem Device bearbeitet werden:
int multigpu_data_index[MAX_GPUS];

CUT_THREADPROC cpu_thread(int * device) {

  int device_id = *device;

  cudaSetDevice(device_id);

  // Hier wird der Code eingefügt, der auf dem Device mit Nummer
  // device_id ausgeführt wird. Das zu bearbeitende Datenpaket
  // ist über multigpu_data_index[device_id] indiziert.

  ...

  // Freigeben des Device:
  multigpu_working[device_id] = 0;

  CUT_THREADEND;
}
```

Der aufrufende Code überprüft die Verfügbarkeit der vorhandenen Devices und startet die einzelnen CPU-Threads, denen jeweils ein Device und die zu bearbeitenden Daten zugeteilt werden.

```
// Anzahl der zu bearbeitenden Datenpakete:
int data_packages = ... ;

// Zu Beginn sind alle Devices verfügbar:
for(int i = 0; i < MAX_GPUS; i++)
  multigpu_working[i] = 0;

// Der i-te CPU-Thread verwendet das i-te Device:
for(int i = 0; i < MAX_GPUS; i++)
  thread_id[i] = i;


// Alle Datenpakete müssen bearbeitet werden:
for(int i = 0; i < data_packages; i++) {

  // Das aktuelle Datenpaket wurde noch nicht bearbeitet:
  int processed = 0;

  // Zähler, um verschiedene Devices durchzugehen:
  int device_id = 0;

  // Suche solange nach einem freien Device, bis das
  // Datenpaket einem Device zugewiesen wurde:
  while(!processed) {

    // Wenn aktuelles Device frei ist ...
    if(!multigpu_working[device_id]) {

      // ... ist es jetzt belegt ...
      multigpu_counter[device_id] = 1;

      // ... und bearbeitet Datenpaket Nummer i:
      multigpu_data_index[device_id] = i;

      // Starten des CPU-Threads:
      threads[device_id] =
         cutStartThread((CUT_THREADROUTINE) cpu_thread,
                        (void *) & thread_id[device_id];
```

```
      // Ab jetzt wird das Datenpaket bearbeitet:
      processed = 1;

    } else {

      // Falls das überprüfte Device nicht frei ist wird bei
      // dem nächsten Device weitergesucht:
      device_id ++;

      // Wieder bei Null anfangen, wenn device_id zu groß ist:
      if(device_id == device_count) device_id = 0;
    }
  }
}
// Warten bis alle CPU-Threads fertig sind:
cutWaitForThreads(threads, device_count);
```

3.9 Grenzen von CUDA

Die hochparallele Architektur moderner Grafikkarten zieht einige Einschränkungen in deren Programmierbarkeit nach sich: Generell kann auf einer GPU nur einfach strukturierter C-Code ausgeführt werden. Dieses Manko wurde durch die neue *Fermi-Architektur* verbessert, dessen Programmiermodell ebenfalls objektorientierte Konstrukte zulässt [80]. Da Kernels keinen Speicher allozieren können, ist eine dynamische Speicherverwaltung auf der GPU nicht möglich. Ebenso ist ein rekursiver Aufruf von Funktionen auf der GPU nicht möglich. Zudem rechnen die meisten Grafikkarten nur in einfacher Genauigkeit (*Single-Precision*). Die für die Rechnungen in dieser Arbeit verwendeten Chipsätze (GT 200, Tesla 10XX) unterstützen zwar doppelte Genauigkeit (*Double-Precision*), dies aber aktuell um den Faktor 8 langsamer, was sich mit Fermi ebenfalls verbessert hat. Für die in dieser Arbeit behandelten Bildverarbeitungsoperatoren sind Rechnungen in einfacher Genauigkeit allerdings ausreichend. Bei rundungsfehleranfälligen Rechnungen kann dies jedoch zu Problemen führen. Außerdem ist anzumerken, dass durch die von außen nicht einsichtige Verteilung der Blöcke und einzelnen Threads auf die einzelnen Prozessoren der Grafikkarten, das CUDA Programmiermodell nicht deterministisch ist. Es obliegt dem Programmierer seine in CUDA entwickelten parallelen Programme deterministisch zu gestalten. Trotz der Einschränkung in der Programmierbarkeit ist das CUDA Programmiermodell Turing-vollständig.

3.10 Alternativen zu CUDA

Im abschließenden Abschnitt dieses Kapitels sind die Alternativen der Hersteller *AMD* und *Intel* aufgeführt, sowie zwei Ansätze, das auf Grafikhardware basierte Hochleistungsrechnen plattform- und herstellerunabhängig zu gestalten.

3.10.1 Brook GPU

Brook GPU ist ein an der Stanford University entwickelter Compiler, der die Verwendung moderner Grafikhardware für beliebige parallele Programme ermöglicht. Die Verwendung von Grafikkarten für allgemeine Aufgaben wird in diesem Zusammenhang häufig mit *GPGPU* (*General-Purpose Computation on Graphics Hardware*) [39] abgekürzt. Der Vorteil des *Brook* GPU Compilers ist, dass er sowohl plattform- als auch hardwareunabhängig ist. *Brook* ist eine Erweiterung von Standard-C und stellt so eine Verknüpfung der weit verbreiteten Programmiersprache C mit einfachen Ideen der parallelen Programmierung her. Zentral ist dabei der neu eingeführte Datentyp *Stream*. Beispielsweise bezeichnet

```
float s<10, 10>;
```

einen Stream der Dimensionen 10×10 vom Datentyp `float`. GPU Programme, die in diesem Zusammenhang ebenfalls als Kernels bezeichnet werden, operieren dann parallel auf den Streamdaten, die im Texturspeicher der Grafikkarten abgelegt werden. Weitere Details zu *Brook* GPU sind [23] zu entnehmen.

3.10.2 AMD Firestream

AMD Firestream Prozessoren stellen das Konkurrenzprodukt zu den *Tesla* Prozessoren von *Nvidia* dar. Ihre grundlegende Architektur ist ähnlich und Firestream Prozessoren können mit einem Derivat des *Brook* GPU Compilers (Kapitel 3.10.1) oder in *OpenCL* (siehe unten) programmiert werden. Weitere Details finden sich in [4].

3.10.3 Intel Ct

Das als *Tera-Scale Computing* oder *Larrabee* bezeichnete Forschungsprogramm von *Intel* hat zum Ziel, massiv parallele Hardware unter Verwendung gewöhnlicher CPU Kerne zu realisieren. Das hierzu von Intel entwickelte Programmiermodell *Ct* [38] ist im Gegensatz zu CUDA und *Brook* deterministisch. Obwohl bereits 2007 ein Prototyp eines Prozessors mit 80 Kernen und einer Spitzenleistung von etwa einem Teraflop präsentiert wurde, hat dieses Projekt keine Marktreife erlangt. Ende 2009 verkündete Intel das Ende des Projekts *Larabee* für den Anwendermarkt, führt das Projekt aber zu Forschungszwecken weiter.

3.10.4 OpenCL

Der Standard *OpenCL 1.0* (*Open Computing Language*) [71] wurde am 8. Dezember 2008 veröffentlicht. Ziel ist es, einen Programmier-Standard bereitzustellen, der sowohl

plattform- als auch hardwareunabhängiges Entwickeln paralleler Programme ermöglicht. Ähnlich wie in CUDA besteht ein *OpenCL* kompatibler Rechner aus einem Host und einer beliebigen Anzahl von *OpenCL* Devices. Der Host verteilt wiederum, die ebenfalls als Kernels bezeichneten, *OpenCL* Programme auf den einzelnen Devices. Hierbei werden die Kernels erst zur Laufzeit vom *OpenCL* Compiler übersetzt, was eine optimierte Übersetzung für die verwendete Hardware ermöglicht. Die drei großen Hersteller *AMD*, *Nvidia* und *Intel* haben angekündigt eine *OpenCL* Unterstützung für ihre jeweiligen Programmiermodelle zur Verfügung zu stellen. Diese angekündigte Kompatibilität stellt sicher, dass derzeit in CUDA oder *Brook* entwickelte Programme später mit relativ wenig Aufwand in *OpenCL* portiert und dann mit beliebiger Hardware verwendet werden können. Eine erste hardwareunabhängige Implementierung von *OpenCL* ist von *Apple* im Betriebssystem *Snow Leopard* (Mac OS X 10.6) realisiert [6]. *Nvidia* hat zudem eine *OpenCL* Version veröffentlicht, welche die *OpenCL* Programme in CUDA übersetzt und anschließend den üblichen CUDA Compiler verwendet [82, 83]. Ebenso stellt *AMD* mittlerweile einen *OpenCL* Compiler für die *Firestream*-Produktserie zur Verfügung.

Kapitel 4

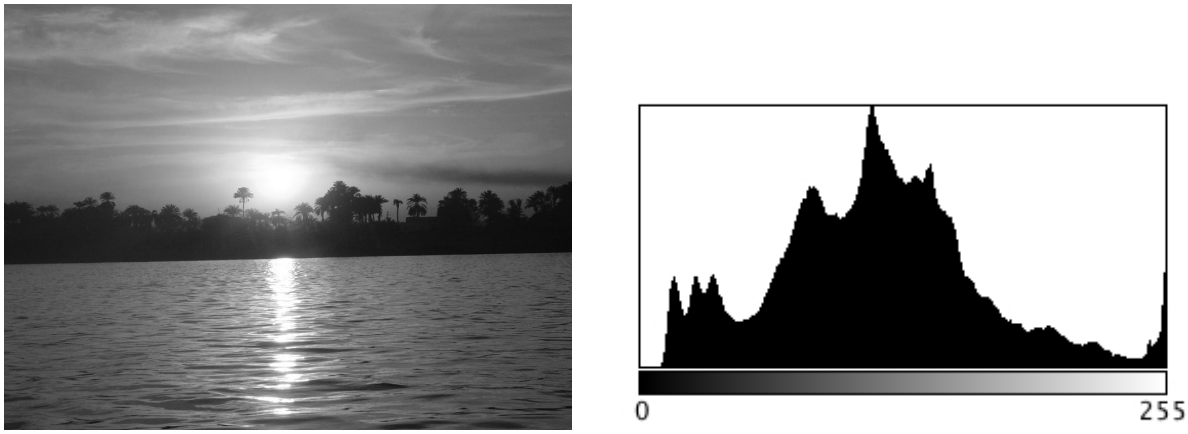
Elementare Operatoren der digitalen Bildverarbeitung

In diesem Kapitel werden einige elementare Operatoren der digitalen Bildverarbeitung vorgestellt, die in den Kapiteln 6 und 7 zur Konstruktion komplexer Bildverarbeitungsoperatoren benötigt werden und bei einigen Anwendungen direkt zur rauschvermindernden Vorverarbeitung eingesetzt werden können. Zudem werden Hinweise auf effiziente Implementierungen dieser Operatoren gegeben. Die hier vorgestellten Operatoren sind allgemein auf n-dimensionalen Grauwertbildern u definiert. Der Wert der diskreten Abbildung $u(x_1, ..., x_n)$ bezeichnet dabei den Grauwert des Bildes u an der Position $(x_1, ..., x_n)$. Im weiteren Verlauf dieser Arbeit werden die Operatoren speziell für die Dimensionen $n = 2$ und $n = 3$ angewandt. Ein einzelner Bildpunkt eines n-dimensionalen Bildes wird, wie bei zweidimensionalen Bildern üblich, als *Pixel* bezeichnet. Die Grauwerte sind ganzzahlig und entstammen dem Intervall $[0, G]$. Im Abschnitt über Merkmalsextraktion (Kapitel 4.4) wird von einem kontinuierlichen Grauwertintervall $[0, 1]$ ausgegangen, das durch eine einfache Skalierung mit anschließender Rundung auf das ganzzahlige Intervall $[0, G]$ abgebildet werden kann. Das durch einen Operator T transformierte Bild u wird als $u^* = Tu$ bezeichnet. Der Grauwert des Ergebnisbildes in einem Pixel p wird als *Antwort* des Filters in p bezeichnet. Eine detailliertere Einführung in die Thematik der digitalen Bildverarbeitung und weitere Details zu den hier behandelten Operatoren findet sich in [49].

4.1 Grauwert-Histogramm

Definition 4.1 (Grauwert-Histogramm)
Sei u ein diskretes Grauwertbild mit Grauwerten aus dem Intervall $[0, G]$. Das Grauwert-Histogramm *(Abbildung 4.1) oder kurz* Histogramm *ist der Graph der Funktion*

$$h(g) = \sum_{(x_1,...,x_n)} \delta_{g,u(x_1,...,x_n)}. \tag{4.1}$$



(a) **(b)**

Abbildung 4.1: *(a) Grauwertbild eines Sonnenuntergangs mit $G = 255$. (b) Zugehöriges Grauwert-Histogramm.*

Hierbei bezeichnet $\delta_{i,j}$ die Kroneckerfunktion*. Die Histrogramm-Funktion $h(g)$ weist somit jedem Grauwert seine Häufigkeit im Bild u zu.*

4.2 Punktoperatoren

Definition 4.2 (Punktoperatoren)
Ein Punktoperator *T weist jedem Pixel p eines gegebenen Bildes u in Abhängigkeit von seiner Position im Bild und seinem Farb- oder Grauwert einen neuen Farb- bzw. Grauwert zu:*

$$u^*(x_1, ...x_n) \;=\; T_{x_1,...,x_n}(u(x_1, ..., x_n)) \tag{4.2}$$

Bei dieser Zuweisung wird keinerlei Information aus der Nachbarschaft von p verwendet. Hierbei heißt der Punktoperator T inhomogen*, falls er explizit von der Lage $(x_1, ..., x_n)$ der einzelnen Pixel abhängt und* homogen*, falls er nur von dem Grauwert des jeweiligen Pixels abhängt.*

4.2.1 Lookup-Tabellen

In der Praxis werden Punktoperatoren meist auf diskrete Grauwertbilder angewandt. Sei u ein Grauwertbild der Dimension 512×512. Jeder Bildpunkt sei mit 8-Bit kodiert, was einem ganzzahligen Grauwert aus dem Intervall $[0, 255]$ entspricht und T sei ein homogener Punktoperator. Bei einer naiven Implementierung von T berechnet man den Punktoperator in jedem Bildpunkt separat, wodurch $512 \cdot 512 = 262144$ Auswertungen von T nötig sind. Viele dieser Berechnungen werden auf diese Weise hochgradig redundant ausgeführt, da das ursprüngliche Bild nur aus 256 verschiedenen Grauwerten besteht. Es ist daher sinnvoll, die Ergebnisse zunächst für alle 256 möglichen Grauwerte zu berechnen, diese in einer

sogenannten *Lookup-Tabelle* zu speichern und anstatt T in jedem Bildpunkt auszuwerten, in der Lookup-Tabelle nachzuschlagen. Lediglich bei sehr einfachen Punktoperatoren, wie dem unten definierten *Negativoperator*, ist die direkte Berechnung schneller als die Verwendung einer Lookup-Tabelle.

4.2.2 Homogene Standardtransformationen

Homogene Transformationen hängen nach Definition nur vom Grauwert des zu transformierenden Pixels ab. Daher genügt es, diese Abbildungen auf jedem Grauwert g des Eingabebildes zu definieren.

Negativtransformation

Die *Negativtransformation* N ist durch die Abbildungsvorschrift

$$N(g) = G - g \tag{4.3}$$

definiert.

Potenztransformation

Die *Potenztransformation* P_γ, auch als *Gammakorrektur* bezeichnet, ist definiert durch

$$P_\gamma(g) = \left\lfloor G \left(\frac{g}{G}\right)^\gamma \right\rfloor . \tag{4.4}$$

Hierbei bezeichnet $\lfloor . \rfloor$ die *Gaußklammer* oder *Abrundungsfunktion*.

Logarithmische Transformation

Die *logarithmische Transformation* L ist gegeben durch

$$L(g) = \left\lfloor G \log_{G+1}(g+1) \right\rfloor . \tag{4.5}$$

Exponentialtransformation

Die *Exponentialtransformation* E ist durch

$$E(g) = \left\lfloor (G+1)^{\frac{g}{G}} - 1 \right\rfloor \tag{4.6}$$

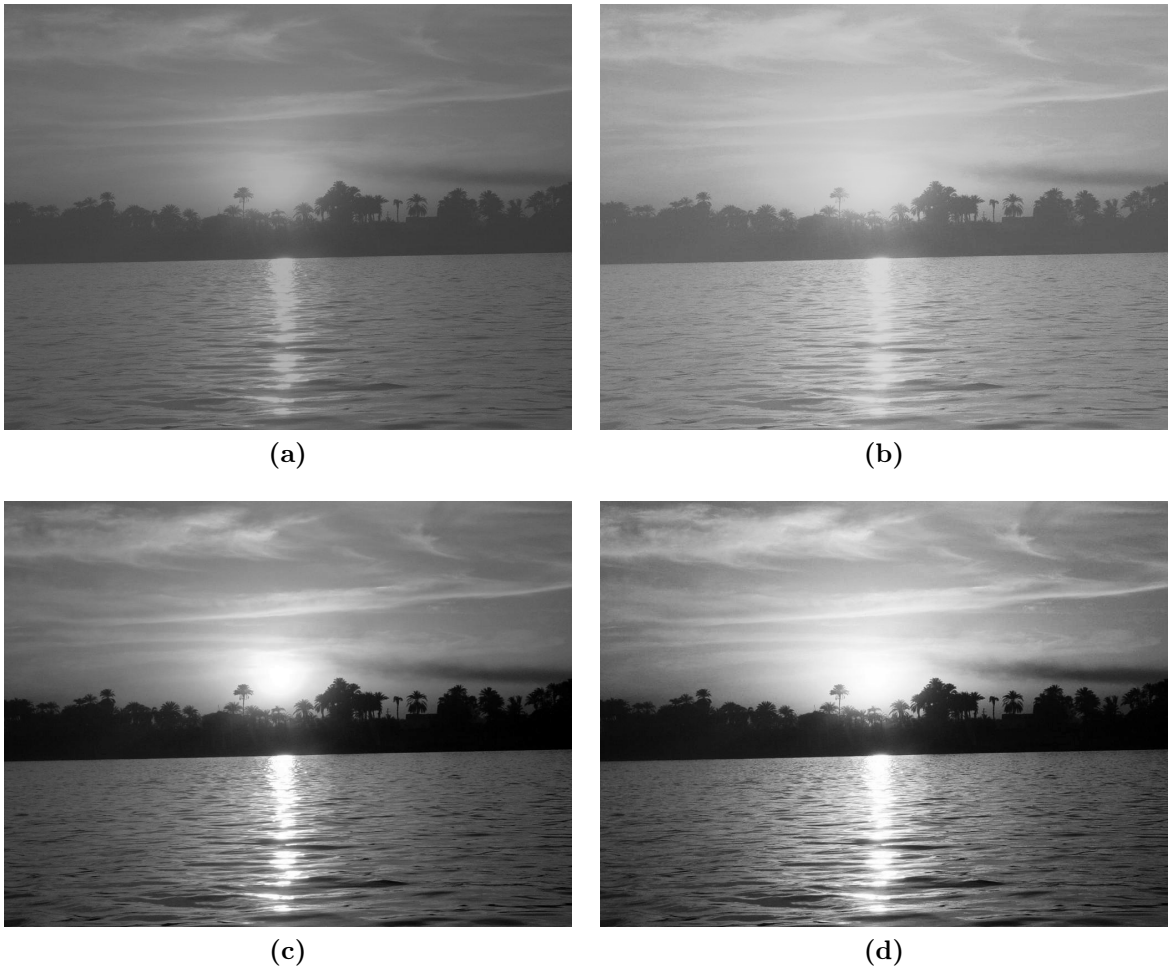
gegeben. Betrachtet man die logarithmische Transformation und die Exponentialtransformation auf einem kontinuierlichen Grauwertspektrum, so sind die beiden Funktionen zueinander invers. Wie die bisher vorgestellten homogenen Standardtransformationen auf ein Grauwertbild wirken, ist Abbildung 4.2 zu entnehmen.

(a) **(b)**

(c) **(d)**

(e) **(f)**

Abbildung 4.2: *Resultate, der in diesem Abschnitt eingeführten homogenen Standardtransformationen: (a) Originalbild. (b) Negativtransformation. (c) Logarithmische Transformation. (d) Exponentialtransformation. (e) Potenztransformation mit* $\gamma = 0.5$*. (f) Potenztransformation mit* $\gamma = 2$*.*



(a) **(b)**

(c) **(d)**

Abbildung 4.3: *(a) Sonnenuntergangsbild mit schlechtem Kontrast. (b) Nach Anwendung des Aufhellungs-Operators A_{128}. (c) Nach Anwendung von Histogramm-Normalisierung. (d) Nach Anwendung von Histogramm-Ausgleich.*

Aufhellung

Der Aufhellungs-Operator A hängt von der Eingabegröße $t \in [0, G]$ ab und ist definiert durch

$$A_t(g) \;=\; \min\left\{ g + \left\lfloor \frac{t \cdot g}{G} \right\rfloor, G \right\}. \tag{4.7}$$

Abbildung 4.3b zeigt die Anwendung des Aufhellungsoperators A_{128}. Dunkle Bereiche werden weniger aufgehellt, helle Bereiche werden stärker aufgehellt. Dieser Operator findet in der semiautomatischen Nachbearbeitung von Dendritensegmenten in der im Rahmen dieser Arbeit entstandenen Software *SpineLab* Anwendung (vgl. Kapitel B.1.4).

Histogramm-Normalisierung

Sei u ein Grauwertbild mit Grauwerten aus dem Intervall $[0, G]$. Häufig werden nicht alle möglichen Grauwerte in dem Bild angenommen. Zudem ist es oftmals der Fall, dass nur Grauwerte aus dem Intervall $[g_{min}, g_{max}]$ mit $0 \leq g_{min} < g_{max} \leq G$ tatsächlich angenommen werden. Um die optische Qualität solcher Bilder aufzuwerten, kann man den Kontrast erhöhen, indem die vorhandenen Grauwerte auf das volle Intervall $[0, G]$ skaliert werden. Dies leistet die *Histogramm-Normalisierung* (Abbildung 4.3c)

$$HN(g) \;=\; \left\lfloor G\,\frac{g - g_{min}}{g_{max} - g_{min}} \right\rfloor. \tag{4.8}$$

Zu beachten gilt, dass die Histogramm-Normalisierung lediglich eine optische Verbesserung des Ausgangsbildes bedingt, die statistischen Eigenschaften der zugrunde liegenden Grauwertverteilung jedoch nicht verändert [49].

Histogramm-Ausgleich

Beim *Histogramm-Ausgleich*, auch *Histogramm-Äqualisierung* oder *Histogramm-Egalisierung* (Abbildung 4.3d) genannt, wird ebenfalls der Kontrast des Eingabebildes verstärkt, wobei eine Gleichverteilung der Grauwerte im Histogramm des Zielbildes berechnet wird, um so den zu Verfügung stehenden Wertebereich optimal ausnutzen zu können. Zunächst sei unter Verwendung der Histogramm-Funktion $h(g)$ (Gleichung 4.1) das *kumulative Histogramm* $h_k(g)$ definiert durch

$$h_k(g) \;=\; \sum_{i=0}^{g} h(i). \tag{4.9}$$

Der Histogramm-Ausgleich ist dann durch

$$HA(g) \;=\; \left\lfloor \frac{G\,h_k(g)}{M} \right\rfloor \tag{4.10}$$

definiert, wobei M die Anzahl der Pixel des zu transformierenden Bildes bezeichnet.

Globale Schwellwertverfahren

Globale Schwellwertverfahren zur Bildsegmentierung fallen ebenfalls in die Kategorie der homogenen Standardtransformationen und sind in dieser Arbeit in Kapitel 7.1 ausführlich beschrieben.

4.2.3 Gradationskurven

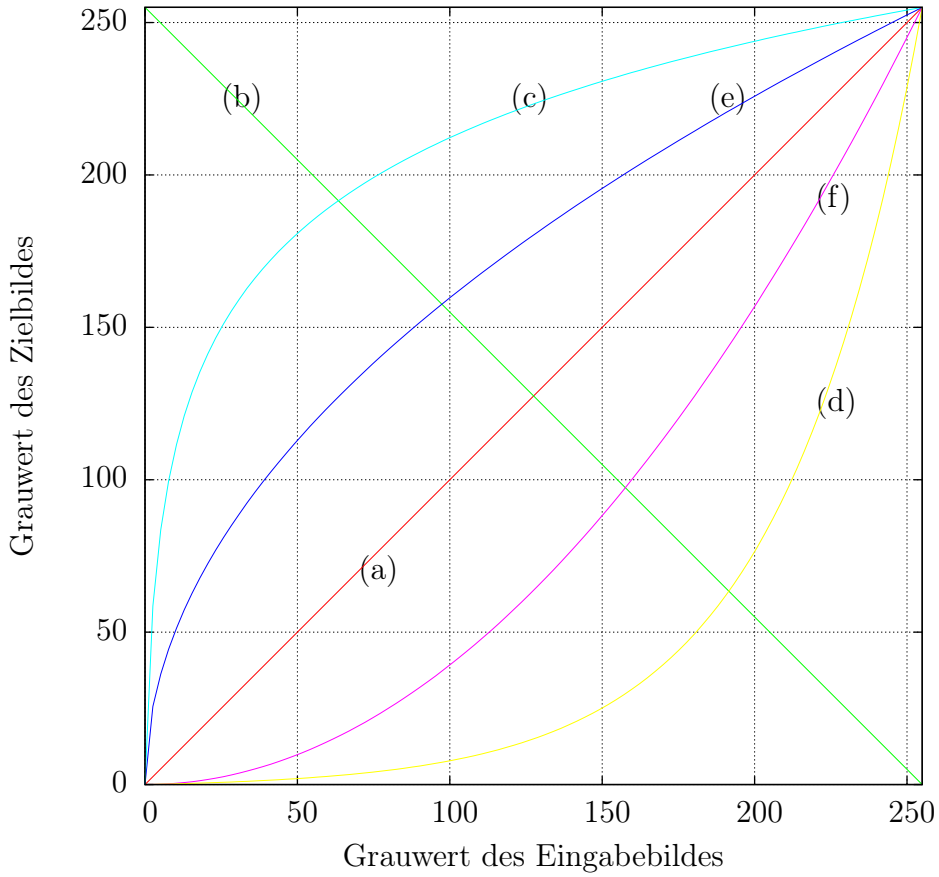


Abbildung 4.4: *Gradationskurven verschiedener homogener Standardtransformationen (Kapitel 4.2.2). (a) Identität. (b) Negativtransformation. (c) Logarithmische Transformation. (d) Exponentialtransformation. (e) Potenztransformation mit* $\gamma = 0.5$*. (f) Potenztransformation mit* $\gamma = 2$*.*

Definition 4.3 (Gradationskurve)
Die Gradationskurve*, auch* Transformationskennlinie *genannt, beschreibt die Transformation der Grauwerte eines Eingabebildes u auf die Grauwerte des Ergebnisbildes u^* unter einem homogenen Punktoperator T.*

Bei einer in der Praxis üblichen diskreten Grauwertverteilung ist die Gradationskurve durch die Lookup-Tabelle gegeben. Die Gradationskurve ermöglicht ein einfaches Ablesen, welcher Grauwert g auf welchen Grauwert g^* unter dem homogenen Punktoperator T abgebildet wird. Die Gradationskurven verschiedener homogener Standardtransformationen, die im vorherigen Abschnitt eingeführt wurden, sind in Abbildung 4.4 skizziert.

4.3 Nachbarschaftsoperatoren

Bei den sogenannten *Nachbarschaftsoperatoren* hängt die Antwort des Filters in einem Pixel p von dem Grauwert dieses Pixels, sowie den Grauwerten der Pixel in einer Umgebung von p ab. Diese Umgebung wird als *Nachbarschaft* von p bezeichnet. Einige wichtige Nachbarschaftsoperatoren, die zur Rauschverminderung in Bildern verwendet werden, sind im Folgenden beschrieben.

4.3.1 Gaußscher Weichzeichner

Der *Gaußsche Weichzeichner* oder *Gauß Filter* ist als diskrete Faltung des Grauwertbildes u mit dem *Gauß-Kern*

$$g(x,\sigma) = \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}}\, e^{\frac{-\|x\|^2}{2\sigma^2}} \tag{4.11}$$

definiert, wobei σ^2 der Varianz der Gaußschen Normalverteilung entspricht. Der Gaußsche Weichzeichner ist ein *Tiefpassfilter*, der niedrige Frequenzen des Bildsignals durchlässt und hohe Frequenzen des Bildsignals auslöscht. Die Antwort des Gaußschen Weichzeichners kann als Lösung der *linearen Wärmeleitungsgleichung* oder *Diffusionsgleichung*

$$\partial_t u = \Delta u \tag{4.12}$$

betrachtet werden [50]. Diese Beobachtung führt zu der Klasse der *anisotropen Diffusionsfilter*, die den Wärmefluss über eine Orts- und grauwertabhängige Strukturerkennung regeln. Der sehr wirksame *trägheitsbasierte anisotrope Diffusionsfilter* wird in Kapitel 5 detailliert behandelt. Obwohl die Filterantwort in einem Pixel durch die Faltung mit dem Gauß-Kern (4.11) von allen anderen Pixel des Bildes abhängt, genügt es, eine kleine Nachbarschaft jedes Pixels zu betrachten, da diese Abhängigkeit aufgrund des exponentiellen Terms im Gauß-Kern sehr schnell vernachlässigbar klein wird. Daher berechnet man zunächst eine diskrete Maske, die ausdrückt wie die Antwort des Filters von einem Pixel und seiner Nachbarschaft abhängt. Die Maske wird anschließend so normiert, dass ihre Einträge aufsummiert Eins ergeben, sodass der Filter masseerhaltend wirkt. Eine effiziente

Implementierung des Gaußschen Weichzeichners nutzt aus, dass das Ergebnis dasselbe ist, wenn man zunächst mit einem eindimensionalen Gauß-Kern in x-Richtung und anschließend mit demselben Gauß-Kern in y-Richtung faltet [49]. Dieses Vorgehen lässt sich auf Bilder beliebiger Dimension verallgemeinern.

Filterung von Hintergrundrauschen

Der Gaußsche Weichzeichner kann dafür verwendet werden das Hintergrundrauschen bei Aufnahmen mit variierendem Kontrast zu vermindern. Hierbei wird das Eingabebild mit dem Gauß-Kern gefaltet und anschließend dieses Ergebnis vom Originalbild abgezogen. Das Resultat wird dann normiert. Dieses Verfahren wird als *Background-Noise-Subtraction* bezeichnet.

Definition 4.4 (Background-Noise-Subtraction)

$$BS(x, \sigma) := HN(u(x) - g(x, \sigma) * u(x)). \tag{4.13}$$

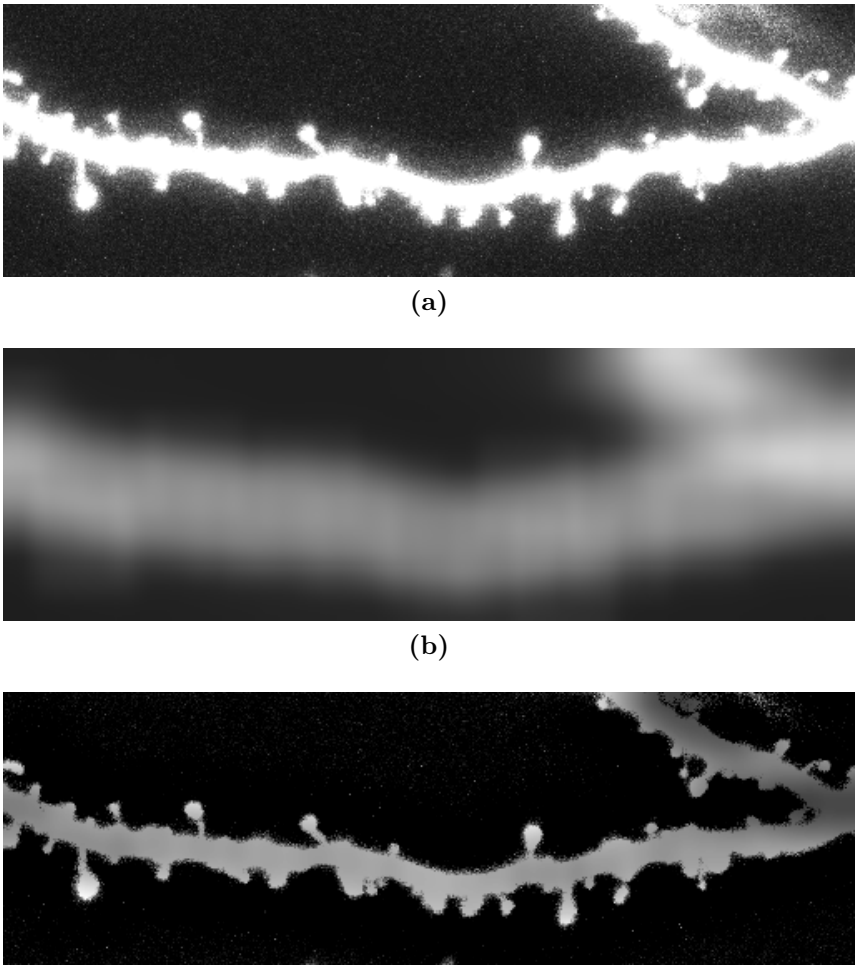
Typischerweise wird hierbei ein großes σ im Bereich von 60 bis 100 verwendet. Ein Ergebnis der Reduktion des Hintergrundrauschens ist Abbildung 4.5 zu entnehmen.

4.3.2 Medianfilter

Der *Medianfilter* kann ebenfalls zur Unterdrückung von Rauschen in Bilddaten verwendet werden. Beim Medianfilter werden alle Grauwerte der Nachbarschaft eines Pixels der Größe nach sortiert. Die Antwort des Filters ist dann der mittlere Grauwert dieser Liste, der sogenannte *Median*. Die Größe der betrachteten Nachbarschaft ist der relevante Eingabeparameter des Filters. Der Medianfilter zeichnet sich dadurch aus, dass er extreme Ausreißer in Grauwertbildern eliminiert, während Kanten weitestgehend erhalten bleiben. Allerdings werden sehr feine Strukturen durch Anwendung des Median-Filters zerstört. Bei der Rekonstruktion von Dendritensegmenten kann der Medianfilter als Vorverarbeitungsschritt eingesetzt werden, da er das in diesen Aufnahmen vorhandene feinkörnige Rauschen stark vermindert (Abbildung 4.6). Die Wahl einer zu großen Nachbarschaft zerstört allerdings relevante Bildinformation (Abbildung 4.6d). Der Medianfilter eignet sich ebenfalls zur Reduktion des Rauschens in Computertomographie-Aufnahmen.

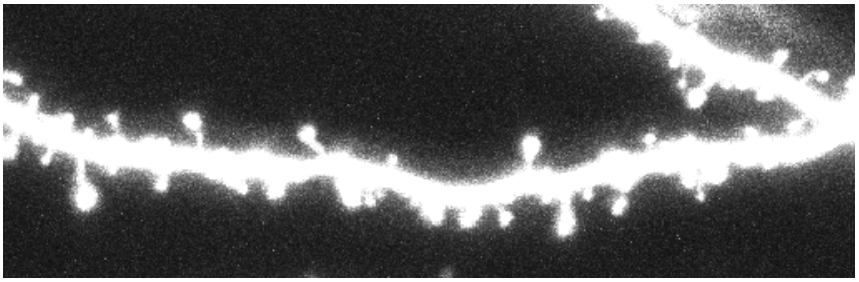
4.3.3 Mittelwertfilter

Der *Mittelwertfilter* funktioniert ähnlich wie der Medianfilter. Die Antwort des Mittelwertfilters ist jedoch nicht der Median der betrachteten Grauwerte, sondern deren arithmetisches Mittel, weshalb er anfällig gegenüber starken Ausreißern ist. Daher findet er in dieser Arbeit keine Anwendung.

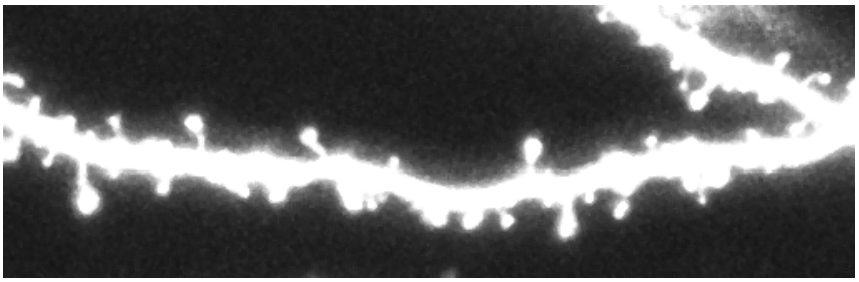


(a) **(b)** **(c)**

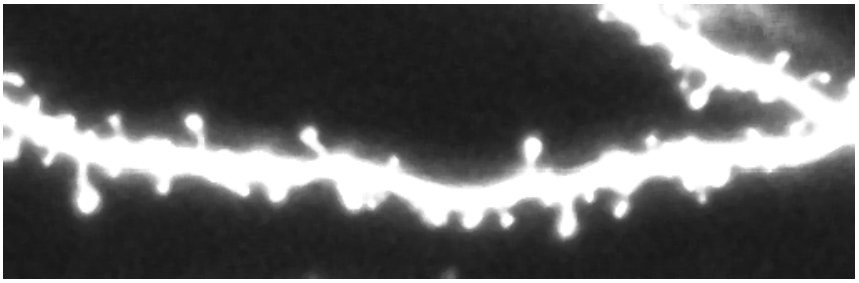
Abbildung 4.5: *Reduktion von Hintergrundrauschen. (a) Volumenprojektion eines Dendritensegments. (b) Gaußscher Weichzeichner mit* $\sigma = 100$*. (c) Resultat von* $BS(x, 100)$*.*



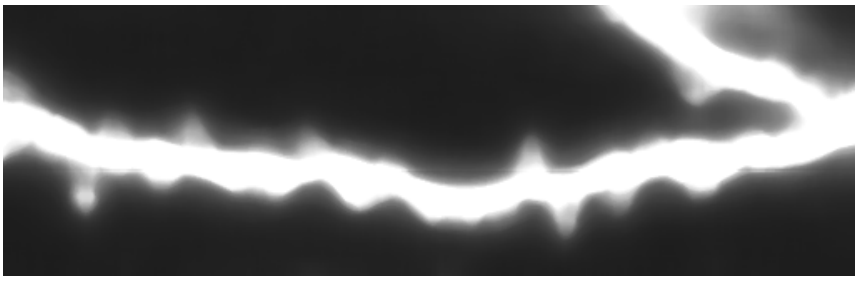
(a)



(b)



(c)



(d)

Abbildung 4.6: *Der Medianfilter verringert feinkörniges Rauschen in Mikroskopaufnahmen, wie es beispielsweise in den konfokalen Aufnahmen von Dendritensegmenten häufig vorkommt. (a) Originalaufnahme (Volumenprojektion). Nach Anwendung eines Medianfilters mit Nachbarschaftsgröße (b)* 1*, (c)* 2 *und (d)* 9*. (d) Bei Wahl einer zu großen Nachbarschaft kann relevante Bildinformation verloren gehen.*

4.4 Merkmalsextraktion

Kanten in einem Bild entsprechen einer großen lokalen Änderung des Grauwerts. Mathematisch formuliert entspricht dies einer großen Ableitung der Grauwerte des Bildes. Bilder sind eine endliche Ansammlung von Pixel unterschiedlicher Grauwerte, d.h. mathematisch gesehen eine diskrete Funktion

$$u : \Omega_h \to [0, 1] \tag{4.14}$$

mit

$$\Omega_h \subset \mathbb{Z}^n, \tag{4.15}$$

die nicht differenzierbar ist. Betrachtet man die Funktion u jedoch als diskreten Repräsentanten der kontinuierlichen, hinreichend glatten Bildsignalfunktion

$$U : \Omega \to [0, 1] \tag{4.16}$$

mit

$$\Omega \subset \mathbb{R}^n, \tag{4.17}$$

so ist aus der Analysis wohlbekannt, wie die Ableitungen von U gebildet werden. Da in der Regel für ein gegebenes diskretes Bild u keine kontinuierliche Darstellung U bekannt ist, können diese Ableitungen nur numerisch in einer diskreten Anzahl von Punkten $x \in \Omega$ berechnet werden. Mit Ω_h ist jedoch bereits eine Diskretisierung des Gebiets Ω gegeben, weshalb es sich anbietet die Ableitungen in genau diesem Gitter, d.h. in jedem Bildpunkt des diskreten Bildes u, numerisch zu berechnen. Diese diskrete Ableitung wird als *Ableitung des Bildes* u bezeichnet.

4.4.1 Grauwertgradient

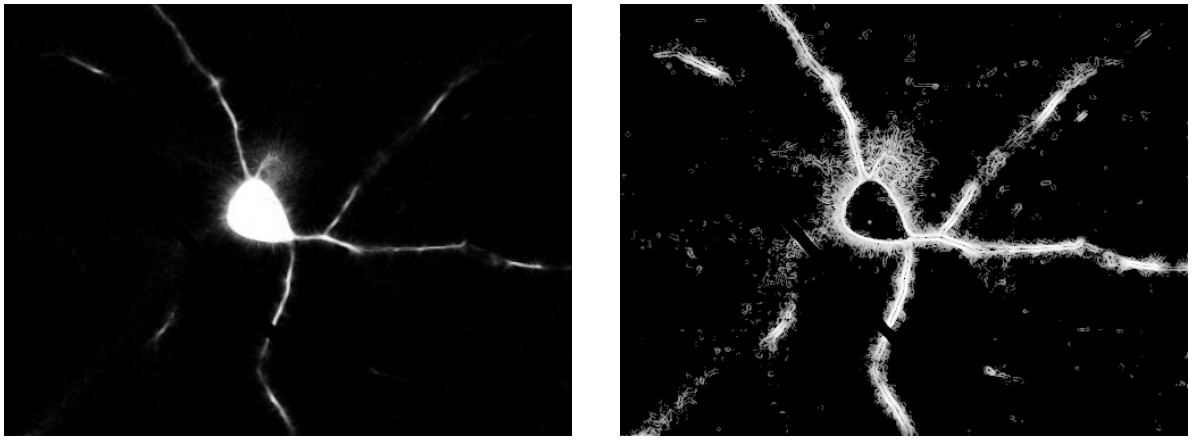
Die erste Ableitung eines Bildes wird als *Grauwertgradient* bezeichnet. Der Grauwertgradient ∇_h ist eine Abbildung

$$\nabla_h : \Omega_h \to [-1, 1]^n \tag{4.18}$$

mit

$$\nabla_h u(x_1, ..., x_n) = \begin{pmatrix} \partial_{h,x_1} u(x_1, ..., x_n) \\ . \\ . \\ . \\ \partial_{h,x_n} u(x_1, ..., x_n) \end{pmatrix}. \tag{4.19}$$

Die zur Berechnung des Gradienten benötigten partiellen Ableitungen werden durch finite Differenzen ∂_{h,x_i} approximiert [102]. Das Ergebnis ist ein Vektor mit n Elementen. Zur Detektion von Kanten reicht es häufig aus den Betrag des Gradienten zu kennen, weshalb



(a) **(b)**

Abbildung 4.7: *(a) Mit trägheitsbasierter Diffusion (Kapitel 5) vorgefilterte Mikroskopaufnahme (Schnittebenendarstellung). (b) Normalisierter Grauwertgradient.*

nur dieser gespeichert wird. Bei der Betragsbildung kann ein Wert > 1 entstehen, der auf 1 abgeschnitten wird, damit der Betrag des Grauwertgradienten wiederum als Grauwertbild interpretiert werden kann (Abbildung 4.7). An diesem Beispielbild wird bereits deutlich, dass zu einer adäquaten Detektion der Strukturen in den Aufnahmen der Neuronenzellen aufwändigere Verfahren als das simple Extrahieren von Kanteninformationen durch den Grauwertgradienten eingesetzt werden müssen.

4.4.2 Hesse-Matrix

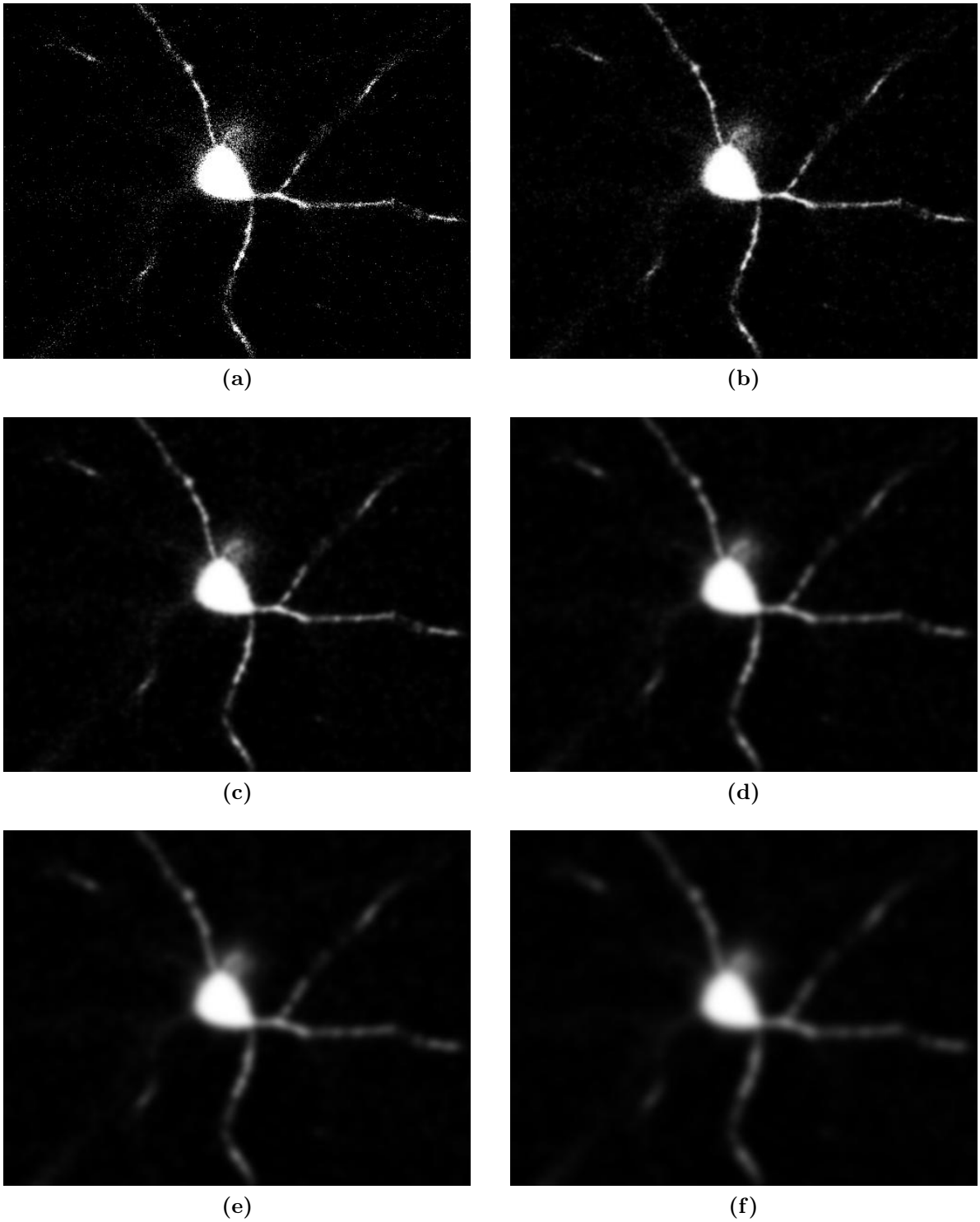
Die Nulldurchgänge der zweiten Ableitung entsprechen betragsmäßigen Maxima des Grauwertgradienten. Die zweite Ableitung eines Bildes u ist gegeben durch die *Hesse-Matrix*

$$\mathcal{H}(u) = \begin{pmatrix} \frac{\partial_h^2 u}{\partial_h x_1 \partial_h x_1} & \cdots & \frac{\partial_h^2 u}{\partial_h x_1 \partial_h x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial_h^2 u}{\partial_h x_n \partial_h x_1} & \cdots & \frac{\partial_h^2 u}{\partial_h x_n \partial_h x_n} \end{pmatrix}. \tag{4.20}$$

Setzt man die kontinuierliche Bildfunktion U als hinreichend glatt voraus, so vertauschen die partiellen Ableitungen und die Hesse-Matrix wird symmetrisch. Die Eigenwerte sind dann nach dem Spektralsatz über normale Operatoren [36] sämtlich reell. Speziell für den zwei- und dreidimensionalen Fall können die Eigenwerte der Hesse-Matrix leicht durch die Nullstellen ihres charakteristischen Polynoms durch die geschlossenen Formeln zur Lösung quadratischer bzw. kubischer Gleichungen bestimmt werden [14]. Durch Ordnen und differenziertes Betrachten der Eigenwerte können unterschiedlich ausgeprägte Strukturen in Bildern detektiert werden. Weitere Details sind in Kapitel 6 zu finden.

4.5 Multiskalen-Repräsentation

Die im vorherigen Abschnitt diskutierte Merkmalsextraktion ist nur dazu geeignet sehr fein aufgelöste Strukturen zu detektieren. Um größere Strukturen und vor allem verschieden große Strukturen in einem Bild erkennen zu können, müssen Filtermasken unterschiedlicher Größe zum Einsatz kommen. Dies würde den Rechenaufwand allerdings erheblich erhöhen. Eine andere Möglichkeit ist, das Bild in verschiedenen Auflösungen zu betrachten. Hierzu wird eine neue Dimension in der Betrachtung des Bildes eingeführt, die sogenannte *Skala*. Die Koordinate dieser Dimension wird mit σ bezeichnet. Für die Repräsentation eines Pixels $u(x_1, ..., x_n)$ des Bildes u im sogenannten *Skalenraum* schreibt man dann $u(\sigma, x_1, ..., x_n)$. Typischerweise wird die Darstellung eines Bildes im Skalenraum durch Anwendung des Gaußschen Weichzeichners (Kapitel 4.3.1) gewonnen. Die Repräsentation einer Mikroskopaufnahme im Skalenraum ist in Abbildung 4.8 dargestellt. Die Wichtigkeit der Verwendung einer Multiskalenanalyse bei der Merkmalsextraktion wird bei dem in Kapitel 6 behandelten Multiskalen-Scharfzeichner deutlich. Weitere Details und Eigenschaften des Skalenraums sind ebenfalls [49] zu entnehmen.



(a) **(b)**

(c) **(d)**

(e) **(f)**

Abbildung 4.8: *Darstellung einer Mikroskopaufnahme im Skalenraum (Schnittebenenbild): (a) $\sigma = 0$. (b) $\sigma = 1$. (c) $\sigma = 2$. (d) $\sigma = 3$. (e) $\sigma = 4$. (f) $\sigma = 5$.*

Kapitel 5

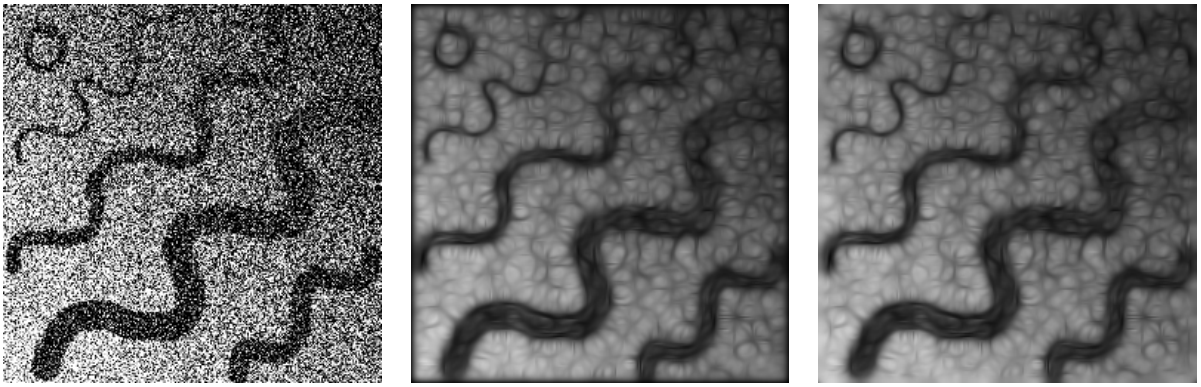
Trägheitsbasierte anisotrope Diffusion

Wie in den Vorarbeiten [21, 50, 90, 92, 101] ausführlich beschrieben, eignet sich das Schema der *trägheitsbasierten anisotropen Diffusion* hervorragend dafür, Rauschen aus den aufgenommen Mikroskopbildern zu entfernen, während die in den Bildern vorhandenen Strukturen erhalten bleiben. Zudem können aufnahmebedingte Lücken, die beispielsweise durch Schattenwürfe entstehen können, durch den trägheitsbasierten Diffusionsfilter geschlossen werden. Der Schwerpunkt in diesem Kapitel liegt darin, zu zeigen, wie das Schema der trägheitsbasierten Diffusion effizient parallel implementiert werden kann, um es auf große Datenmengen anwenden zu können. Hierbei findet die Parallelisierung auf zwei unterschiedlichen Ebenen statt: Erstens wird die massiv parallele Architektur moderner Grafikprozessoren bestmöglich ausgenutzt, indem bei einzelnen Bildern möglichst viele Bildpunkte gleichzeitig verarbeitet werden. Zweitens werden große Datenmengen in kleine Blöcke aufgeteilt, die dann parallel auf vielen Grafikkarten gleichzeitig bearbeitet werden können. Hierbei zeigt sich einerseits, dass die hier beschriebene Implementierung die Möglichkeiten moderner Grafikkarten fast optimal ausnutzt und andererseits selbst riesige Datenmengen durch den Einsatz hinreichend vieler Grafikkarten innerhalb weniger Minuten gefiltert werden können. Das Schema der trägheitsbasierten Diffusion eignet sich allerdings nicht für alle Arten von Daten: Enthalten die Mikroskopaufnahmen zu viel feinkörniges Rauschen, das durch Streulicht verursacht wird, besteht die Gefahr, dass der trägheitsbasierte Filter dieses Rauschsignal mit dem Bildsignal vermischt. Ebenso können sehr feine Objektstrukturen durch den trägheitsbasierten Diffusionsfilter zerstört werden. Die in diesem Kapitel beschriebenen Ergebnisse sind teilweise bereits in [52] veröffentlicht worden.

5.1 Formulierung als Differentialgleichung

5.1.1 Die Diffusionsgleichung

Das Schema von Diffusionsfiltern ist durch die *Wärmeleitungsgleichung* oder *Diffusionsgleichung*



(a) **(b)** **(c)**

Abbildung 5.1: *Vergleich zwischen Neumann-Null und Dirichlet-Null-Randbedingung. Die schwarzen Ränder bei Verwendung der Dirichlet-Null-Randbedingung sind nicht erwünscht. (a) Zweidimensionales Testbild (Quelle:* [28]*). (b) Nach Anwendung des trägheitsbasierten Diffusionsfilters mit Dirichlet-Null-Randbedingung. (c) Nach Anwendung des trägheitsbasierten Diffusionsfilters mit Neumann-Null-Randbedingung.*

$$\partial_t u = \nabla \cdot (D(u)\nabla u) \text{ auf } \mathbb{R}^+ \times \Omega \tag{5.1}$$

$$u(x,0) = u_0 \text{ auf } \bar{\Omega} \tag{5.2}$$

$$(D(u)\nabla(u)) \cdot \vec{n} = 0 \text{ auf } \mathbb{R}^+ \times \partial\Omega \tag{5.3}$$

definiert [87, 114]. Hierbei bezeichnet Ω eine offene und beschränkte Teilmenge des $\mathbb{R}^2$ bzw. $\mathbb{R}^3$, abhängig von der Dimension des zu filternden Bildes. Entscheidend ist die Wahl des lokalen Diffusionsparameters $D(u)$. Hierbei entspricht $D(u) = 1$ der *isotropen Diffusion*, d.h. dem Gaußschen Weichzeichner (Kapitel 4.3.1). Verschiedene Ansätze, diesen als skalare Funktion zu definieren finden sich in [87, 114] und sind in [68] eingehend diskutiert. Dieser Ansatz wird als *nichtlineare Diffusion* bezeichnet. Wählt man $D(u)$ als tensorwertige Abbildung, so erhält man das Schema der *anisotropen nichtlinearen Diffusion*. Vorschläge für zweidimensionale anisotrope nichtlineare Diffusionsfilter sind in [114] beschrieben. Die Verwendung von physikalischen Trägheitsmomenten zur Bestimmung des Diffusionstensors $D(u)$ definiert schließlich das Schema der *trägheitsbasierten anisotropen nichtlinearen Diffusion*, das etwas kürzer als *trägheitsbasierte anisotrope Diffusion* [90] bezeichnet wird. Wie eingangs erwähnt, ist dieses Filterschema in den Arbeiten [21, 50, 90, 92, 101] bereits ausführlich untersucht worden. Da die Berechnung des trägheitsbasierten Diffusionsfilters jedoch sehr zeitaufwändig ist, liegt der Schwerpunkt dieses Kapitels in der Entwicklung einer hochparallelen Implementierung des Filters, der die Anwendung auf sehr große Datensätze ermöglicht.

Die Wärmeleitungsgleichung (5.1) ist eine *Startwert-Randwertaufgabe* [102]. Als Startwert wird die Wärmeverteilung zum Zeitpunkt $t_0 = 0$ festgelegt (Gleichung 5.2). In

der Anwendung des Bildfilters ist dies das zu filternde Bild. Als Randbedingung bietet sich die *Neumann-Null-Randbedingung*, auch *natürliche Randbedingung* genannt, an (Gleichung 5.3). Diese besagt, dass keine Wärme über den Rand des Gebietes fließen kann, was ein nach außen isoliertes System beschreibt. Alternativ könnte eine *Dirichlet-Null-Randbedingung* verwendet werden, die besagt, dass außerhalb des Gebietes die Temperatur Null vorherrscht. Dies führt allerdings einerseits Wärme aus dem Bild ab und andererseits entstehen dadurch unschöne schwarze Ränder in den gefilterten Bildern (Abbildung 5.1).

5.1.2 Herleitung der Diffusionsgleichung

Die Diffusionsgleichung (5.1) wird aus dem *Fick'schen Gesetz*, sowie dem *Prinzip der Massenerhaltung* hergeleitet. Das Fick'sche Gesetz

$$\phi = -D\,\nabla u \tag{5.4}$$

besagt, dass durch Diffusion Unterschiede in der Konzentrationsverteilung ausgeglichen werden. Hierbei generiert der Konzentrationsgradient ∇u einen Fluss ϕ, der diesem Gradienten entgegenwirkt. Kombiniert mit dem Prinzip der Massenerhaltung

$$\partial_t u = -\nabla \cdot \phi \tag{5.5}$$

ergibt sich die Diffusionsgleichung (5.1). Formel (5.5) ist einsichtiger, wenn man mit Hilfe des Gauß'schen Integralsatzes [58] die integrale Formulierung

$$\int_\Omega \partial_t u \, dV = -\int_\Omega \nabla \cdot \phi \, dV = -\int_{\partial\Omega} \phi \cdot \vec{n} \, dS \tag{5.6}$$

betrachtet, die besagt, dass die zeitliche Veränderung der gesamten Masse innerhalb des Gebietes Ω gerade dem Fluss über den Rand $\partial\Omega$ entspricht.

5.2 Strukturerkennung

5.2.1 Diskreter Trägheitstensor

Der Diffusionstensor $D(u)$ wird bei der trägheitsbasierten anisotropen Diffusion durch die Eigenwerte und Eigenvektoren des physikalischen Trägheitstensors D_I berechnet. Dieser wird durch die Formel für eine diskrete Massenverteilung [60]

$$D_I = \frac{1}{M} \sum_i \left(u(i) \begin{pmatrix} y_i^2 + z_i^2 & -x_i y_i & -x_i z_i \\ -y_i x_i & x_i^2 + z_i^2 & -y_i z_i \\ -z_i x_i & -z_i y_i & x_i^2 + y_i^2 \end{pmatrix} \right) \tag{5.7}$$

mit

$$M = \sum_i u(i) \tag{5.8}$$

in jedem Bildpunkt direkt berechnet. Hierbei wird in den Formeln (5.7) und (5.8) über alle Voxel v_i mit $\|v - v_i\|_\infty \leq \rho$ summiert und $u(i)$ bezeichnet den Grauwert des Bildes im Bildpunkt v_i. Die Größe des Integrationsgebietes wird durch den Parameter ρ bestimmt, der entsprechend der Auflösung der zu erkennenden Strukturen passend gewählt werden muss (Kapitel 5.9). Die Verwendung von Formel (5.7) zur Berechnung des Trägheitstensors bietet sich für die Implementierung in CUDA an, da die Berechnung für jedes Voxel unabhängig erfolgt und somit trivial parallelisiert werden kann. Es ist allerdings offensichtlich, dass einige Teilsummen aus den Formeln (5.7) und (5.8) in diesem Fall redundant berechnet werden. In einer seriellen Implementierung des Filters können einige dieser redundanten Berechnungen vermieden werden [101].

5.2.2 Steuerung der Diffusion

Durch die Eigenwerte und Eigenvektoren des Trägheitstensors können Beschaffenheit und Ausrichtung von Strukturen in einem Bild erkannt werden [21, 50, 90, 92, 101]. Da der Trägheitstensor D_I eine reelle, symmetrische und positiv definite Matrix ist [60], besitzt er nach dem Spektralsatz für normale Operatoren [36] drei reelle Eigenwerte $\lambda_1 \leq \lambda_2 \leq \lambda_3$ mit den dazugehörigen orthonormalen Eigenvektoren μ_1, μ_2, μ_3. An dieser Stelle sei darauf hingewiesen, dass in den vorherigen Arbeiten die Eigenwerte umgekehrt sortiert sind, da in [101] die Strukturerkennung durch Betrachtung der Funktion $1 - u$ beschrieben ist. Die konsistente Ordnung der Eigenwerte ist in Abschnitt 5.2.3 beschrieben. Der Diffusionstensor $D(u)$ ist dann definiert durch

$$D(u) = S \begin{pmatrix} \eta_1 & 0 & 0 \\ 0 & \eta_2 & 0 \\ 0 & 0 & \eta_3 \end{pmatrix} S^T \tag{5.9}$$

mit

$$S = (\mu_1, \quad \mu_2, \quad \mu_3), \tag{5.10}$$

bestehend aus den Eigenvektoren μ_1, μ_2, μ_3 und den *Diffusionskoeffizienten*

$$\eta_1 = 1, \ \eta_2 = \eta_3 = \epsilon \tag{5.11}$$

für lineare Strukturen,

$$\eta_1 = \eta_2 = 1, \ \eta_3 = \epsilon \tag{5.12}$$

für planare Strukturen und

$$\eta_1 = \eta_2 = \eta_3 = 1 \tag{5.13}$$

für isotrope Strukturen [21, 50, 90, 92, 101]. Durch die in [61] vorgeschlagenen Größen, die an dieser Stelle an die korrekte Ordnung der Eigenwerte der Trägheitstensoren angepasst sind (vgl. Kapitel 5.2.3),

$$c_l = \frac{\lambda_3 - \lambda_2}{\sum_{i=1}^{3} \lambda_i} \tag{5.14}$$

$$c_p = \frac{2\lambda_2 - 2\lambda_1}{\sum_{i=1}^{3} \lambda_i} \tag{5.15}$$

$$c_i = \frac{3\lambda_1}{\sum_{i=1}^{3} \lambda_i} \tag{5.16}$$

mit den Eigenschaften

$$c_l + c_p + c_i = 1 \tag{5.17}$$

und

$$c_l, c_p, c_i \in [0, 1] \tag{5.18}$$

kann dynamisch ermittelt werden, ob lokal lineare, planare oder isotrope Strukturen vorliegen. Es ist aber entscheidend die Diffusionskoeffizienten η_1, η_2 und η_3 je nach erkannter Struktur nach den Formeln (5.11), (5.12) und (5.13) zu wählen und nicht eine nichtlineare Funktion, wie in [87, 114] vorgeschlagen und in [68] ausführlich diskutiert, zur Regelung der Stärke der Diffusion zu verwenden, da sonst nicht gewährleistet ist, dass die Größe der im Originalbild enthaltenen Strukturen erhalten bleibt [92].

5.2.3 Korrekte Ordnung der Eigenwerte

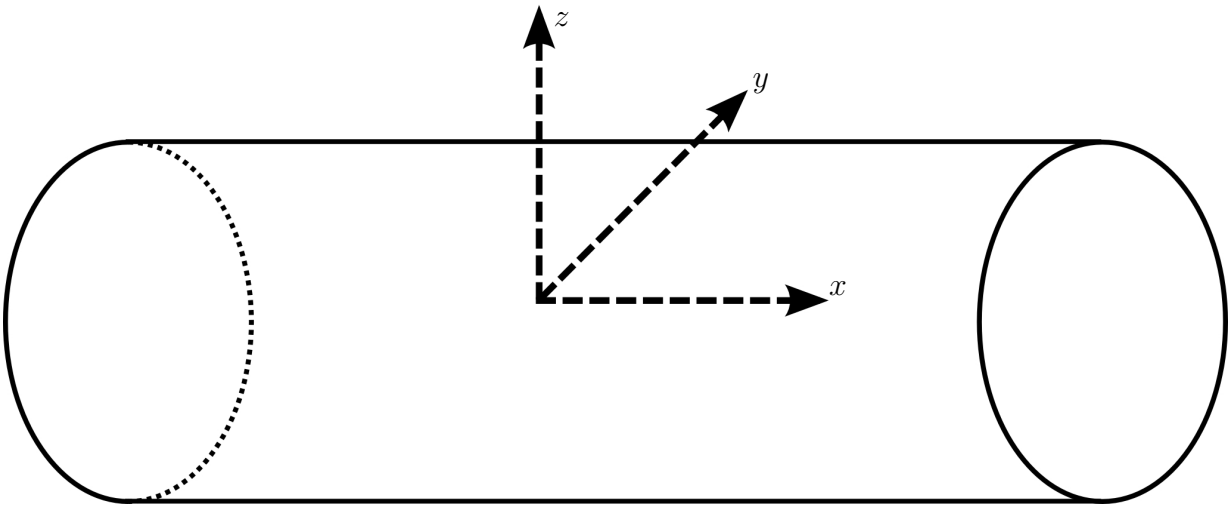


Abbildung 5.2: *Hauptträgheitsachsen eines Zylinders. Das Trägheitsmoment entlang der x-Achse ist klein gegenüber den Trägheitsmomenten entlang der y-Achse und der z-Achse.*

Um die Eigenwerte des Trägheitstensors korrekt zu ordnen ist zunächst wichtig, klar zwi-

schen dem *Diffusionstensor* und dem *Trägheitstensor* zu unterscheiden. Die Eigenwerte η_1, η_2 und η_3 des Diffusionstensors D (5.9), d.h. die *Diffusionstensoren*, steuern die Stärke der Diffusion entlang der ihnen zugehörigen Eigenvektoren. Wie der Name D schon vermuten lässt, ist dies der Tensor, der in die zugrunde liegende nichtlineare Wärmeleitungsgleichung (5.1) eingesetzt wird. Der Trägheitstensor D_I (5.7) ist hingegen ein Hilfskonstrukt, mit dessen Hilfe die Eigenvektoren μ_1, μ_2, μ_3 des Diffusionstensors berechnet werden. Verwirrenderweise stimmen die Eigenvektoren von Diffusionstensor und Trägheitstensor nach Konstruktion des Diffusionstensors überein. Entscheidend ist daher die Frage, wie aus den Eigenwerten des Trägheitstensors die Diffusionskoeffizienten η_i und damit die Eigenwerte des Diffusionstensors bestimmt werden. Dafür muss zunächst die Frage beantwortet werden, ob ein hoher oder ein niedriger Grauwert einer hohen Masse entspricht. Intuitiv sind die Begriffe Masse und Grauwertintensität gleichzusetzen. Helle Bereiche des Bildes entsprechen somit einer dichten Masseverteilung, was konsistent mit den Formeln (5.7) und (5.8) ist. Als nächstes stelle man sich eine lineare Struktur in Form eines Zylinders (Abbildung 5.2) mit Radius r, Länge l und Masse m, d.h. einen idealisierten Dendriten, vor. Diffusion soll entlang des Zylinders, aber nicht senkrecht dazu möglich sein. Die Eigenwerte des Trägheitstensors entsprechen gerade den Hauptträgheitsmomenten des Zylinders [60]. Diese betragen

$$I_1 = \frac{1}{2}mr^2, \tag{5.19}$$

falls der Zylinder um seine Mittelachse rotiert und

$$I_2 = I_3 = \frac{1}{4}mr^2 + \frac{1}{12}ml^2, \tag{5.20}$$

falls der Zylinder senkrecht zu seiner Mittelachse, d.h. um eine der anderen beiden Hauptträgheitsachsen rotiert [60]. Unter der Annahme, dass $l \gg r$, gilt folglich

$$I_1 \ll I_2 = I_3. \tag{5.21}$$

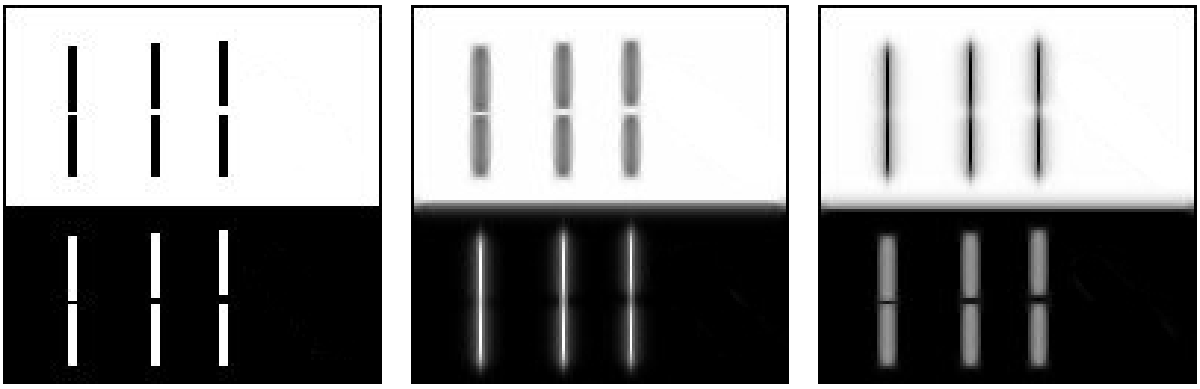
Der Eigenvektor des Trägheitstensors des Zylinders in Richtung der Mittelachse des Zylinders korrespondiert somit zum kleinsten Eigenwert, da das Trägheitsmoment am kleinsten ist, wenn der Zylinder um seine Mittelachse rotiert. Bei der Detektion von linearen Strukturen ist daher Diffusion entlang des kleinsten Eigenwertes des Trägheitstensors erwünscht, was mit der oben eingeführten Ordnung der Eigenwerte des Trägheitstensors und der Diffusionskoeffizienten übereinstimmt. Genauso verhält es sich bei der Detektion von planaren Strukturen, wie beispielsweise Membranoberflächen. Betrachtet man diese als dünne Scheiben, d.h. als Zylinder mit $r \gg l$, so ergibt sich für die Trägheitsmomente senkrecht zur Mittelachse

$$I_1 = I_2 = \frac{1}{4}mr^2 \tag{5.22}$$

und für das Trägheitsmoment entlang der Symmetrieachse wiederum

$$I_3 = \frac{1}{2}mr^2. \tag{5.23}$$

Diffusion ist entlang der Membranoberfläche, d.h. entlang der Eigenvektoren zu den Trägheitsmomenten I_1 und I_2 gewünscht. Somit müssen wiederum die Diffusionskoeffizienten, die den kleinen Eigenwerten des Trägheitstensors entsprechen, groß gewählt werden. Auf dieses Problem wird bereits in [50] hingewiesen. Dort wird eine zweidimensionale Implementierung des trägheitsbasierten Filterschemas detailliert diskutiert und festgestellt, dass dieser Ansatz zur Strukturerkennung nicht ohne Weiteres gleichzeitig zur Detektion von hellen Strukturen auf dunklem Grund und dunkler Strukturen auf hellem Grund eingesetzt werden kann (Abbildung 5.3). Es ist deutlich zu erkennen, dass je nach Filtereinstellung und Hintergrundfarbe, Diffusion entlang der Strukturen oder senkrecht dazu ermöglicht wird. Betrachtet man die 5×5 Pixel großen Ausschnitte eines zweidimensionalen Bildes (Abbildung 5.4) und berechnet man den diskreten zweidimensionalen Trägheitstensor [50] jeweils im mittleren Pixel mit Integrationsgröße $\rho = 2$, so ergeben sich die Trägheitstensoren



(a) **(b)** **(c)**

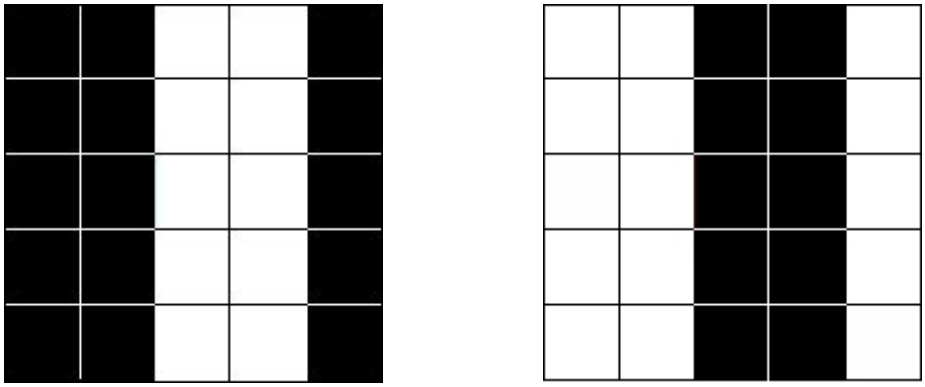
Abbildung 5.3: *Anwendung des trägheitsbasierten Diffusionsfilters aus [50] auf ein zweidimensionales Testbild. (a) Testbild. (b) Filterung mit der Einstellung, dass helle Bereiche einer hohen Masse entsprechen. (c) Filterung mit der Einstellung, dass dunkle Bereiche einer hohen Masse entsprechen. Quelle:* [50]

$$D_I = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.125 \end{pmatrix} \tag{5.24}$$

für die helle Struktur auf dunklem Grund und

$$D_I = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.75 \end{pmatrix} \tag{5.25}$$

für die dunkle Struktur auf hellem Grund. Offenbar ist die Struktur in beiden Bildern in vertikale Richtung ausgerichtet und Diffusion entlang dieser Richtung erwünscht. Die



(a) **(b)**

Abbildung 5.4: *Zwei* 5×5 *Pixel große Ausschnitte aus einem zweidimensionalen Bild mit (a) einer hellen Struktur auf dunklem Grund und (b) einer dunklen Struktur auf hellem Grund. Quelle:* [50]

Eigenwerte und Eigenvektoren von (5.24) und (5.25) lassen sich leicht ablesen, sodass deutlich wird, dass im ersten Fall der kleine Eigenwert zum Eigenvektor $\mu_2 = (0, 1)$ gehört, im zweiten Fall jedoch der größere Eigenwert. Dieses [50] entnommene Beispiel verdeutlicht das Problem der korrekten Ordnung der Eigenwerte des Trägheitstensors. Da bei den vorliegenden Aufnahmen jeweils helle Strukturen auf dunklem Hintergrund detektiert werden sollen, liefert diese Ordnung der Eigenwerte des Trägheitstensors das gewünschte Ergebnis.

5.3 Lösung der partiellen Differentialgleichung

5.3.1 Diskretisierung der Zeitableitung

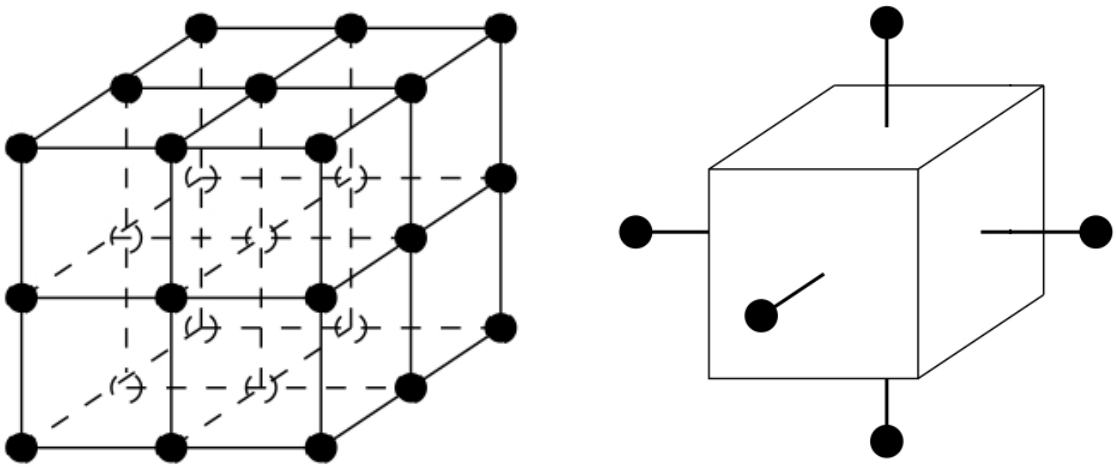
Die Zeitableitung wird durch ein *semi-implizites Eulerverfahren* [50, 101] diskretisiert

$$\frac{u^{t+1} - u^t}{\tau} = \nabla \cdot \left(D\left(u^t\right) \nabla u^{t+1} \right). \tag{5.26}$$

Die Diffusionstensoren $D(u)$ werden hierbei durch die aktuell gegebene Grauwertverteilung u^t berechnet. Durch den Implizität erzeugenden Term ∇u^{t+1} garantiert das Schema unbedingte Stabilität für beliebig große Zeitschrittweiten τ [101, 102].

5.3.2 Diskretisierung der Raumableitung

Die Raumableitungen werden durch ein *Finites Volumen Verfahren* diskretisiert, bei dem *Finite Elemente Ansatzfunktionen* verwendet werden. Integration von Gleichung (5.1) über das Gebiet Ω liefert



(a) **(b)**

Abbildung 5.5: *(a) Uniformes Gitter. (b) Einzelne Integrationsbox. Quelle:* [101]

$$\int_{\Omega} \partial_t u \, dV = \int_{\Omega} \nabla \cdot (D(u) \nabla u) \, dV. \tag{5.27}$$

Das Gebiet Ω wird zunächst durch ein uniformes Gitter (Abbildung 5.5a) überzogen, wobei die Gitterpunkte den Mittelpunkten der Voxel des zu filternden Bildes entsprechen. Anschließend werden k quaderförmige Integrationsboxen Ω_i (Abbildung 5.5b) über das Gitter gelegt, die das Gitter komplett abdecken und deren Mittelpunkte genau den Gitterpunkten entsprechen. Formel (5.1) integriert über die einzelnen Boxen Ω_i ergibt

$$\sum_{i=1}^{k} \int_{\Omega_i} \partial_t u \, dV = \sum_{i=1}^{k} \int_{\Omega_i} \nabla \cdot (D(u) \nabla u) \, dV. \tag{5.28}$$

Nach dem Integralsatz von Gauß [58] ergibt sich für die Summanden der rechten Seite von Gleichung (5.28)

$$\int_{\Omega_i} \nabla \cdot (D(u) \nabla u) \, dV = \int_{\partial\Omega_i} D(u) \nabla u \cdot \vec{n} \, dS \quad \forall\, i = 1, ..., k. \tag{5.29}$$

Aus den Gleichungen (5.28) und (5.29) folgt

$$\sum_{i=1}^{k} \int_{\Omega_i} \partial_t u \, dV = \sum_{i=1}^{k} \int_{\partial\Omega_i} D(u) \nabla u \cdot \vec{n} \, dS. \tag{5.30}$$

Diese Gleichheit ist insbesondere dann erfüllt, wenn

$$\int_{\Omega_i} \partial_t u \, dV = \int_{\partial\Omega_i} D(u) \frac{\partial u}{\partial \vec{n}} \, dS \quad \forall\, i = 1, ..., k \tag{5.31}$$

auf allen Integrationsboxen Ω_i gilt. Gleichung (5.31) wird auch als Erhaltungssatz bezeichnet, der besagt, dass die zeitliche Änderung der Masse in jeder Integrationsbox gleich dem Fluss über den Rand der jeweiligen Integrationsbox ist (vgl. Formel (5.6)).

5.3.3 Resultierende Diskretisierung

Kombination von Zeitdiskretisierung (5.26) und Raumdiskretisierung (5.31) ergibt

$$u_i^{t+1} = u_i^t + \frac{\tau}{|\Omega_i|} \left(\int_{\partial\Omega_i} D(u) \frac{\partial u}{\partial \vec{n}} \, dS \right) \quad \forall\, i = 1, ..., k. \tag{5.32}$$

Hierbei bezeichnet u_i^t den Wert im i-ten Gitterpunkt, d.h. im Mittelpunkt der i-ten Integrationsbox zum Zeitpunkt t und $|\Omega_i|$ das Volumen der i-ten Integrationsbox. Auf jeder Integrationsbox Ω_i des Gitterpunktes x_i wird eine trilineare Basisfunktion φ_i mit

$$\varphi_i(x_j) = \delta_{ij} \tag{5.33}$$

definiert. Hierbei bezeichnet δ_{ij} die *Kroneckerfunktion*. Die Gitterfunktion u zum Zeitpunkt t lässt sich dann darstellen als Linearkombination der Basisfunktionen φ_j mit skalaren Koeffizienten u_j^t

$$u^t(x) = \sum_{j=1}^{k} u_j^t \, \varphi_j(x). \tag{5.34}$$

Aus den Gleichungen (5.32) und (5.34) folgen die k linearen Gleichungen

$$u_i^{t+1} = u_i^t + \frac{\tau}{|\Omega_i|} \left(\sum_{j=1}^{k} u_j^{t+1} \int_{\partial\Omega_i} \left(D\left(u^t(x)\right) \nabla\varphi_j(x) \right) \cdot \vec{n} \, dS \right). \tag{5.35}$$

Approximation der auftretenden Flächenintegrale über die Seitenflächen der Integrationsboxen durch eine einfache Mittelpunktsregel mit Integrationspunkten ip_k führt zu dem linearen Gleichungssystem

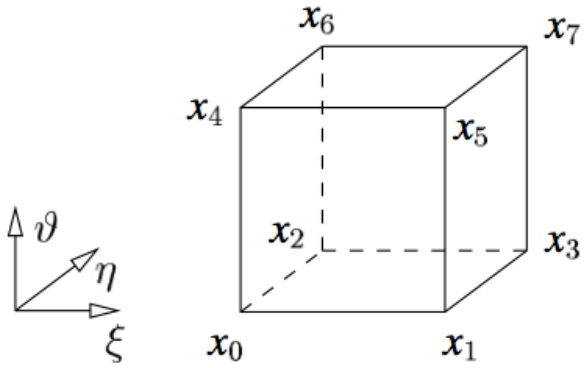
$$Mu^{t+1} = (I - \tau A)u^{t+1} = u^t \tag{5.36}$$

mit Koeffizienten

$$a_{ij} = \frac{1}{|\Omega_i|} \sum_k \left(D\left(u^t(ip_k)\right) \nabla\varphi_j\left(ip_k\right) \right) \cdot \vec{n_k} \, |S_k| \,. \tag{5.37}$$

Hierbei bezeichnet $|S_k|$ den Flächeninhalt der k-ten Seitenfläche der entsprechenden Integrationsbox und n_k die äußere Normale dieser Seitenfläche. Die ip_k bezeichnen die Integrationspunkte, deren Wahl weiter unten ausführlicher diskutiert wird. Die Maschenweite h des Gitters kann als konstant angenommen werden, da bei dem Bildfilter das zugrunde liegende Gitter durch das zu filternde Bild fest vorgegeben ist und wird daher auf $h = 1$ festgesetzt [50]. Da sich bei Mikroskopaufnahmen die Auflösung in z-Richtung von der Auflösung in x- und y-Richtung meist unterscheidet, ist bei einer Implementierung des Filters darauf zu achten, dass diese Anisotropie durch eine mögliche Wahl der Gittergröße bedacht wird. Für ein isotropes Gitter mit Maschenweite $h = 1$ vereinfacht sich Gleichung (5.37) zu

$$a_{ij} = \sum_k \left(D\left(u_i^t(ip_k)\right) \nabla\varphi_j\left(ip_k\right)\right) \cdot \vec{n_k} \left|S_k\right| . \tag{5.38}$$



(a)

Abbildung 5.6: *Referenzwürfel. Quelle:* [101]

Es bietet sich an, die Koeffizienten der Matrix auf sogenannten Referenzwürfeln (Abbildung 5.6) zu berechnen. Die aus finite Elemente Verfahren bekannten trilinearen Ansatzfunktionen [43] müssen dann nur auf dem Referenzelement definiert werden:

$$
\begin{aligned}
\Phi_0(\xi,\eta,\vartheta) &= (1-\xi)\,(1-\eta)\,(1-\vartheta)\\
\Phi_1(\xi,\eta,\vartheta) &= \xi\,(1-\eta)\,(1-\vartheta)\\
\Phi_2(\xi,\eta,\vartheta) &= (1-\xi)\,\eta\,(1-\vartheta)\\
\Phi_3(\xi,\eta,\vartheta) &= \xi\,\eta\,(1-\vartheta)\\
\Phi_4(\xi,\eta,\vartheta) &= (1-\xi)\,(1-\eta)\,\vartheta\\
\Phi_5(\xi,\eta,\vartheta) &= \xi\,(1-\eta)\,\vartheta\\
\Phi_6(\xi,\eta,\vartheta) &= (1-\xi)\,\eta\,\vartheta\\
\Phi_7(\xi,\eta,\vartheta) &= \xi\,\eta\,\vartheta
\end{aligned}
$$

Die Gradienten $\nabla\Phi_{\iota(j,e)}$ sind dann definiert durch

$$\nabla\Phi_{\iota(j,e)}(ip(\xi,\eta,\vartheta)) = \nabla\varphi_j(ip(x,y,z)). \tag{5.39}$$

Hierbei berechnet die *Indizierungsfunktion* $\iota(j,e)$ den entsprechenden Index der Ansatzfunktionen des Referenzwürfels in Abhängigkeit des Index j der globalen Ansatzfunktion und des Index e des betrachteten Integrationsvolumens. Die oben eingeführten Integrationspunkte ip_k müssen auf dem Referenzwürfel definiert werden. Hierfür ergeben sich nach [101] zwei Möglichkeiten (Abbildungen 5.7 und 5.8). Nach [101] ist jedoch kein merklicher Qualitätsunterschied des Filters bei einer unterschiedlichen Wahl der Integrationspunkte festzustellen.

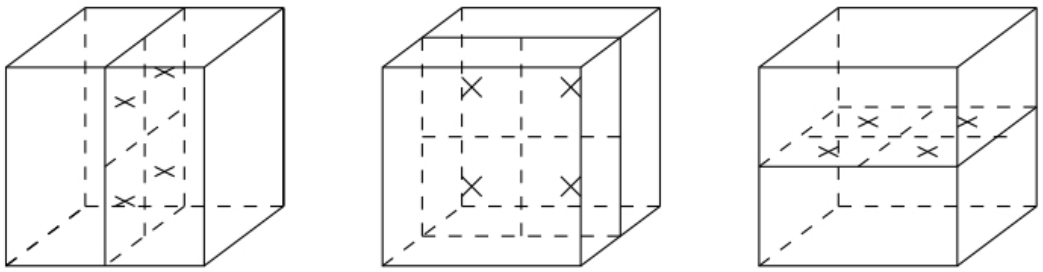


Abbildung 5.7: *Mögliche Wahl der Integrationspunkte ip_k im Referenzwürfel. Quelle:* [101]

5.3.4 Diskretisierung der Randbedingung

Um die *Neumann-Null-Randbedingung* (5.3) oder *natürliche Randbedingung* korrekt zu diskretisieren, muss für Integrationspunkte, die auf dem Rand $\partial\Omega$ liegen,

$$D(u)\nabla u = 0 \tag{5.40}$$

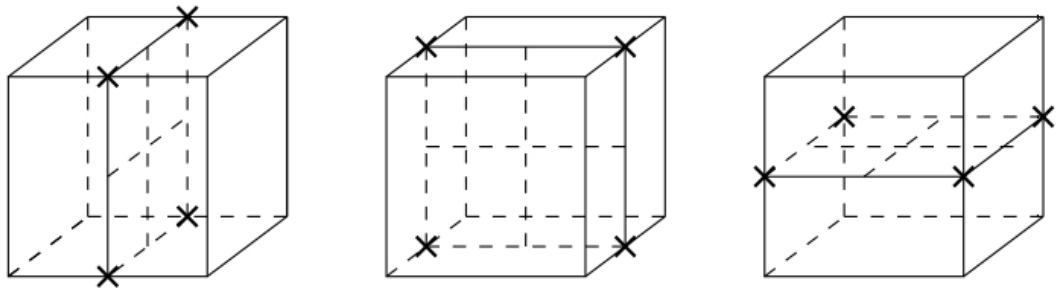


Abbildung 5.8: *Alternative Wahl der Integrationspunkte ip_k im Referenzwürfel. Bei dieser Möglichkeit liegen die zwölf Integrationspunkte auf den Mitten der Kanten des Referenzwürfels. Diese Variante wird in der Implementierung von* NeuRA2 *verwendet. Quelle:* [101]

gelten. Diese Forderung ist insbesondere dann erfüllt, wenn

$$\nabla u = 0 \tag{5.41}$$

für alle auf dem Rand liegenden Integrationspunkte gilt. Hierfür werden bei der Berechnung der Matrixeinträge die Beiträge von Summanden, bei denen der Integrationspunkt auf dem Rand des Gebietes liegt, einfach weggelassen [43].

5.3.5 Assemblierung der Matrix

Die Assemblierung der Matrix kann prinzipiell auf zwei verschiedene Weisen geschehen:

- Elementweise Assemblierung
- Knotenweise Assemblierung

Bei der *elementweisen Assemblierung* wird für jeden einzelnen Referenzwürfel die Kopplung aller acht Ecken des Referenzwürfels mit allen anderen Ecken des Würfels berechnet. Vorteil dieser Methode ist, dass die Kopplung von sogenannten *interagierenden Knoten* vom Betrage her innerhalb einzelner Elemente gleich ist. Diese Koeffizienten müssen daher nur einmal berechnet werden [101]. Nachteil ist, dass verschiedene Würfel Summanden zu Einträgen aus verschiedenen Matrixzeilen beisteuern. Dies erschwert eine parallele Implementierung der elementweisen Assemblierung. Bei der *knotenweisen Assemblierung* wird die Kopplung jedes Knoten mit sich selbst und mit allen 26 Nachbarknoten berechnet. Hierfür müssen in jedem Knoten Kopplungskoeffizienten von jeweils acht Referenzwürfeln berechnet werden. Die Summanden der Matrixeinträge, die von den interagierenden Knoten herrühren, werden auf diese Weise doppelt berechnet. Dafür kann die knotenweise Assemblierung trivial parallelisiert werden, da die Matrixeinträge jedes einzelnen Knoten unabhängig von den anderen Knoten berechnet werden können. Details zur elementweisen Assemblierung sind [101] zu entnehmen. Die knotenweise Assemblierung wird im Folgenden

genauer beleuchtet, da sie sich leicht parallelisieren lässt und daher in *NeuRA2* eingesetzt wird.

Knotenweise Assemblierung der Systemmatrix

Referenzwürfel	Integrationspunkte
w_0	ip_5, ip_7, ip_{11}
w_1	ip_6, ip_7, ip_{10}
w_2	ip_4, ip_5, ip_9
w_3	ip_4, ip_6, ip_8
w_4	ip_1, ip_3, ip_{11}
w_5	ip_2, ip_3, ip_{10}
w_6	ip_0, ip_1, ip_9
w_7	ip_0, ip_2, ip_8

Tabelle 5.1: *Bei der knotenweisen Assemblierung liefern jeweils drei Integrationspunkte pro Referenzwürfel einen Beitrag zu den Kopplungen des betrachteten zentralen Knotens (vgl. Abbildungen 5.9 und 5.10).*

Für jeden Knoten des Gitters muss eine Matrixzeile mit 27 Einträgen, welche die Kopplungen des Knotens mit sich selbst und seinen 26 Nachbarn enthält, erzeugt werden. Die Matrixeinträge können in einem Feld abgelegt werden, wobei die Reihenfolge der Nachbarn durch die Sternschablonen (5.42), (5.43) und (5.44) fest vorgegeben ist.

$$\begin{bmatrix} 13 & 12 & 11 \\ 14 & 9 & 10 \\ 15 & 16 & 17 \end{bmatrix} \tag{5.42}$$

$$\begin{bmatrix} 4 & 3 & 2 \\ 5 & 0 & 1 \\ 6 & 7 & 8 \end{bmatrix} \tag{5.43}$$

$$\begin{bmatrix} 22 & 21 & 20 \\ 23 & 18 & 19 \\ 24 & 25 & 26 \end{bmatrix} \tag{5.44}$$

Für jeden Knoten müssen die acht Referenzwürfel w_i betrachtet werden (Abbildung 5.9), wobei in jedem Referenzwürfel die Kopplung über drei Integrationspunkte mit den Nachbarknoten läuft (Tabelle 5.1). Bei der knotenweisen Assemblierung werden daher für jeden Referenzwürfel w_i die Kopplungen über die entsprechenden Integrationspunkte ip_k (Abbildung 5.10) mit den über den Referenzwürfel verbundenen Nachbarknoten nach Gleichung (5.38) berechnet.

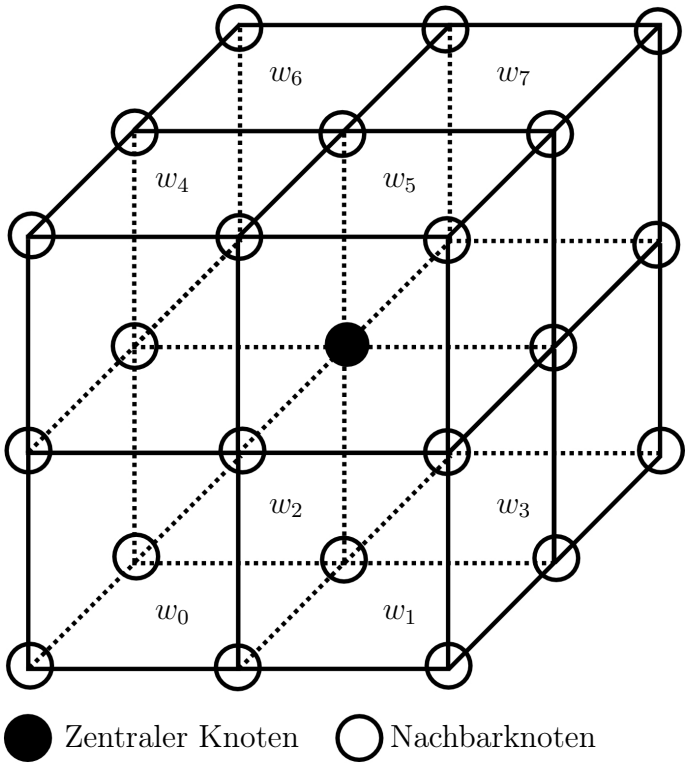


Abbildung 5.9: *Die Kopplungen des zentralen Knotens mit sich selbst und seinen* 26 *Nachbarknoten werden über die acht Referenzwürfel w_i berechnet.*

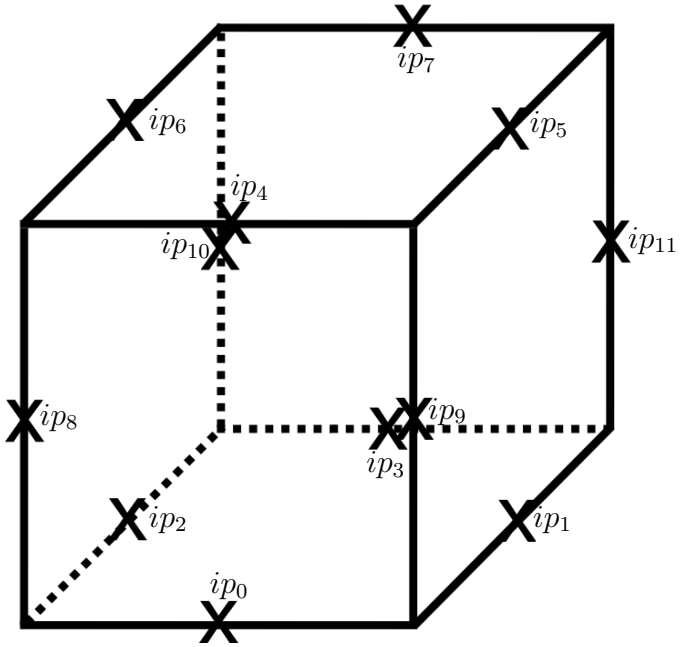


Abbildung 5.10: *Referenzwürfel mit durchnummerierten Integrationspunkten.*

5.3.6 Lösen des resultierenden Gleichungssystems

Durch die Diskretisierung reduziert sich das Problem, die partielle Differentialgleichung (5.1) zu lösen, auf das Problem ein lineares Gleichungssystem der Form

$$Mx = b \tag{5.45}$$

zu lösen. Die Matrix M des linearen Gleichungssystems ist hierbei *dünn besetzt* und enthält maximal 27 von Null verschiedene Einträge pro Zeile [101]. Es bietet sich daher an, ein iteratives Lösungs-Verfahren anzuwenden. In der ältesten Arbeit zur trägheitsbasierten anisotropen Diffusion [101] ist ein *bistabiles Verfahren konjugierter Gradienten* [54] als schnellstes Lösungs-Verfahren evaluiert. Ebenfalls wird in [101] beobachtet, dass das einfache Schema von *Gauß-Seidel* [54] sehr schnell funktioniert. In [50] ist aufgezeigt, dass unter Verwendung des dort eingeführten *projizierten Diffusionstensors*, das resultierende Gleichungssystem mit linearer Abhängigkeit der Zeitschrittweite τ durch die Verfahren von *Jacobi* [54], *Gauß-Seidel* und dem *SOR-Verfahren* [54] gelöst werden kann. Die in [50] gezeigte Konvergenzabschätzung verwendet die Annahme, dass alle Nichtdiagonaleinträge der Matrix M kleiner als Null sind. Dies ist für die Verwendung des vollen Diffusionstensors (5.9) nicht der Fall [101]. Da die positiven Nebendiagonaleinträge, verglichen mit den negativen Nebendiagonaleinträgen und den Diagonalelementen, vom Betrage her sehr klein sind, fallen sie allerdings wenig ins Gewicht. Ein gutes Konvergenzverhalten dieser Verfahren bleibt zumindest für kleine Zeitschrittweiten τ zu erwarten, was sich experimentell bestätigen lässt. Aufgrund der hohen Kosten zur Erstellung der Grobgittermatrizen scheidet ein Mehrgitterverfahren [42, 54] zur effizienten Lösung aus [50, 101].

5.3.7 Iterative Lösungs-Verfahren

Im Folgenden sei das lineare Gleichungssystem $Mx = b$ mit $M \in \mathbb{R}^{n\times n}$ und $x, b \in \mathbb{R}^n$ zu lösen.

Definition 5.1 (Iterative Löser)

1. *Ein* Iterativer Löser *ist definiert durch einen Startwert x_0 und eine Iterationsvorschrift $x_{i+1} := \phi\, x_i$.*
2. *Der Löser heißt* linear*, wenn ϕ linear ist.*
3. *Der Löser heißt* konsistent*, wenn alle Fixpunkte x^* von ϕ die Gleichung $Mx^* = b$ erfüllen.*
4. *Das Verfahren heißt* konvergent*, falls $\forall\, b\, \exists\, x^* : x^* = \lim_{i\to\infty} x_i$ gilt und x^* vom Startwert x_0 unabhängig ist.*

Satz 5.2 (Konsistentes lineares Lösungs-Verfahren)
Ein konsistentes lineares Lösungs-Verfahren *ist gegeben durch*

$$x_{i+1} := x_i - \tilde{M}^{-1}(Mx_i - b). \tag{5.46}$$

Beweis: *Die Linearität des Verfahrens ist offensichtlich. Sei* $x^* = x^* - \tilde{M}^{-1}(Mx^* - b)$, *so folgt* $\tilde{M}^{-1}(Mx^* - b) = 0$. *Da die Matrix* $\tilde{M}^{-1}$ *invertierbar ist, besitzt sie vollen Rang. Somit folgt* $(Mx^* - b) = 0$, *d.h. die Konsistenz des Verfahrens.*

Bemerkung 5.3 (Ursprüngliches Problem)
Für $M = \tilde{M}$ *ergibt sich das ursprüngliche Problem* $x_{i+1} = x_i - M^{-1}(Mx_i - b) = x_i - M^{-1}Mx_i + M^{-1}b = M^{-1}b$. *Die Lösung folgt dann in einem Iterationsschritt.*

Bemerkung 5.4 (Angenäherte Inverse)
$\tilde{M}^{-1}$ *bezeichnet die* angenäherte Inverse *von* M. *Die Matrix* $\tilde{M}$ *ist an die ursprüngliche Matrix* M *angelehnt, aber mit Komplexität O(n) zu invertieren.*

Definition 5.5 (Jacobi-Verfahren und Gauß-Seidel-Verfahren)
Die Matrix M *sei zerlegt in* $M = D - L - R$. *Hierbei bezeichne* D *die Diagonale,* $-L$ *den linken unteren Teil und* $-R$ *den rechten oberen Teil von* M. *Nach Formel* (5.46) *ist das* Jacobi-Verfahren *und das* Gauß-Seidel-Verfahren *definiert durch*

- $\tilde{M}_J = D$ *(Jacobi-Verfahren)*
- $\tilde{M}_{GS} = D - L$ *(Gauß-Seidel-Verfahren).*

Definition 5.6 (Iterationsmatrix)
Der Operator, der in der Iterationsvorschrift (5.46) *auf* x_i *wirkt, heißt* Iterationsmatrix. *Für Jacobi-Verfahren und Gauß-Seidel-Verfahren ergeben sich die Iterationsmatrizen*

- $S_J = D^{-1}(L + R)$
- $S_{GS} = (D - L)^{-1}R$.

Bemerkung 5.7 (Iterationsvorschrift)
Nach [102] *ergeben sich für die* k*-te Iteration folgende Iterationsvorschriften für das Jacobi-Verfahren*

$$x_i^{k+1} := -\frac{1}{m_{ii}} \left[\sum_{j=1, j\neq i}^{n} m_{ij} x_j^k - b_i \right], \; i = 1, 2, ...n, \; k = 0, 1, 2, ... \tag{5.47}$$

und das Gauß-Seidel-Verfahren

$$x_i^{k+1} := -\frac{1}{m_{ii}} \left[\sum_{j=1}^{i-1} m_{ij} x_j^{k+1} + \sum_{j=i+1}^{n} m_{ij} x_j^k - b_i \right], \; i = 1, 2, ...n, \; k = 0, 1, 2, ... \; . \tag{5.48}$$

Die beiden Iterationsvorschriften stimmen fast überein. Der einzige Unterschied besteht darin, dass die in der k-ten Iteration berechneten Einträge von x mit Index $< i$ bereits zur

Berechnung des Eintrags mit Index i verwendet werden. Neben einer üblicherweise besseren Konvergenzgeschwindigkeit des Gauß-Seidel-Verfahrens [117] ist der Speicheraufwand im Vergleich mit dem Jacobi-Verfahren geringer, da für x^k und x^{k+1} kein separater Speicher verwendet werden muss. Das Jacobi-Verfahren kann trivial parallelisiert werden, da jeder Eintrag von x^{k+1} unabhängig voneinander berechnet werden kann. Eine Parallelisierung des Gauß-Seidel-Verfahren ist ungleich schwerer. Sind aber beide Verfahren für ein Problem als konvergent bekannt, lassen sich die Vorteile des geringeren Speicherbedarfs und der Parallelisierbarkeit kombinieren [50], indem man die Iterationsvorschrift des Jacobi-Verfahrens (5.47) verwendet, für x^k und x^{k+1} dasselbe Feld im Speicher benutzt und die Berechnungen der einzelnen Koeffizienten von x^{k+1} in beliebiger Reihenfolge und teilweise parallel ausführen lässt. Auf diese Weise lässt sich ein schnelles paralleles Lösungs-Verfahren für das Gleichungssystem konstruieren.

Definition 5.8 (Lösungs-Verfahren)
Das Lösungs-Verfahren *verwendet die Iterationsvorschrift des Jacobi-Verfahrens* (5.47)*, wobei für die Vektoren x^k und x^{k+1} derselbe Speicher verwendet wird und die Koeffizienten von x^{k+1} in beliebiger Reihenfolge, parallel oder seriell berechnet werden.*

Bemerkung 5.9 (Eigenschaften des Lösungs-Verfahrens)
Bei rein serieller Berechnung der Koeffizienten von x^{k+1} ergibt sich das Gauß-Seidel-Verfahren. Werden alle Koeffizienten von x^{k+1} gleichzeitig berechnet, so entspricht das Lösungs-Verfahren dem Jacobi-Verfahren. Man spricht daher von einem Hybrid-Verfahren.

Ein lineares Gleichungssystem gilt als gelöst, wenn der Defekt d^k nach k Iterationen kleiner als die Abbruchschwelle ϵ ist

$$d^k = \left\| Mx^k - b \right\|_2 < \epsilon. \tag{5.49}$$

Der Defekt wird dabei häufig nach jedem Iterationsschritt berechnet. Da die Berechnung des Defekts allerdings so aufwändig ist, wie das Berechnen einer einzelnen Iteration [50], ist dieses Vorgehen wenig effizient. In [50] ist ferner gezeigt, dass unter Verwendung des dort definierten *projizierten Diffusionstensors*, die benötigten Iterationen für Jacobi- Gauß-Seidel- und SOR-Verfahren linear mit der Zeitschrittweite τ ansteigen, und ein Aufwandschätzer zur Vorhersage der benötigen Iterationen unter dem SOR-Verfahren evaluiert. Für diesen Aufwandschätzer ist eine viermalige Berechnung des Defekts d^k nötig. Bei einer Implementierung in CUDA sollte nach Möglichkeit komplett auf die Berechnung des Defekts verzichtet werden, da es sich bei dem Defekt um eine skalarwertige Rechenoperation handelt, die auf der GPU nur teuer durchzuführen ist. Bei der in [50] vorgeschlagenen Implementierung des dort eingeführten Filterschemas mit Hilfe von *Cg-Shadern* [35] wurde dieses Problem gelöst, indem unabhängig von den Eingabedaten $n\tau$ Iterationen durchgeführt werden, wobei $n = 120$ als *Worst-Case-Konstante* evaluiert ist. Die Effizienz und die Qualität der dort vorgestellten Ergebnisse rechtfertigen dieses Vorgehen. In Kapitel 5.6 wird n für die Implementierung des Lösungs-Verfahrens zur Lösung des Filterschemas unter Verwendung des vollbesetzten Diffusionstensors neu evaluiert.

Bemerkung 5.10 (Weitere Iterationsmethoden)
Weitere Iterationsmethoden sind

- *ILU-Zerlegung (unvollständige LU-Zerlegung)*
- *Krylow-Unterraum-Methoden (z.B. Verfahren konjugierter Gradienten)*
- *Mehrgitterverfahren.*

Da das durch die Diskretisierung des Schemas der trägheitsbasierten anisotropen Diffusion entstandene lineare Gleichungssystem sehr effizient durch das oben beschriebene Hybrid-Lösungs-Verfahren gelöst werden kann, wird an dieser Stelle nicht weiter auf andere Lösungs-Methoden eingegangen. Für eine genauere Diskussion dieser Verfahren sei daher auf [42, 54, 117] verwiesen.

5.4 Implementierung in CUDA

In diesem Abschnitt ist die Implementierung des trägheitsbasierten anisotropen Diffusionsfilters in CUDA für eine GPU beschrieben, die bereits vorab in [52] veröffentlicht wurde. Da man im Rahmen des Diffusionsfilters an einer lokalen Rauschglättung interessiert ist, ermöglicht das lokale Verhalten der zugrunde liegenden Wärmeleitungsgleichung für kleine Zeitschritte eine triviale Parallelisierung des Filters, in dem das Bild in einzelne, überlappende Würfel zerschnitten und auf mehreren Grafikkarten gleichzeitig gefiltert wird. Die Überlappung der einzelnen Würfel ermöglicht eine anschließende Zusammensetzung der einzeln gefilterten Teilbilder, in der keine Schnittkanten sichtbar sind (vgl. Kapitel A.2.1). Obwohl der Speicherbedarf des Filters sehr hoch ist, können auf diese Weise große Datensätze auf einer oder mehreren Grafikkarten in kurzer Zeit gefiltert werden. Im Folgenden wird zunächst davon ausgegangen, dass der zu filternde Datensatz komplett von der verwendeten Grafikkarte gefiltert werden kann.

5.4.1 Eingabeparameter

Das Resultat des Filterprozesses hängt von folgenden Parametern ab:

- *Strukturerkennungsparameter ρ:*
  Der Strukturerkennungsparameter ρ legt die Größe des Integrationsgebietes bei der Berechnung der Diffusionstensoren fest (Abschnitt 5.2.1) und bestimmt damit die Größe der im Bild erkannten Strukturen.

- *Zeitschrittweite τ:*
  Die Zeitschrittweite τ bestimmt die Länge eines Zeitschritts bei der semiimpliziten Zeitdiskretisierung (Abschnitt 5.3.1).

- *Anzahl Zeitschritte n_t:*
  Der Parameter n_t legt die Anzahl der zu berechnenden Zeitschritte fest (Abschnitt 5.3.1).

- *Diffusionskoeffizienten η_1, η_2, η_3:*
  Die Diffusionskoeffizienten werden zur Detektion linearer Strukturen auf $\eta_1 = 1$, $\eta_2 = \eta_3 = \epsilon$ und zur Detektion planarer Strukturen auf $\eta_1 = \eta_2 = 1$, $\eta_3 = \epsilon$ gesetzt.

Die Stärke der Filterung wird dabei durch das Produkt $T = \tau n_t$ festgelegt. Außerdem enthält die hier vorgeschlagene Implementierung des Filters weitere technische Parameter, die im Folgenden eingeführt, diskutiert und optimiert werden.

5.4.2 Arbeitsablauf

Der Filterprozess wird in fünf Arbeitsschritten durchgeführt:

1. *Einlesen der Daten:*
   Der zu filternde Datensatz wird eingelesen und seine Einträge in Fließkommazahlen einfacher Genauigkeit aus dem Intervall $[0, 1]$ konvertiert. Anschließend werden die Daten in den Device Speicher geladen und als eindimensionale Textur gebunden, um einen gecachten Zugriff zu ermöglichen.

2. *Berechnen der Diffusionstensoren:*
   Das Device berechnet für jeden Gitterpunkt, d.h. jeweils für die Mittelpunkte der Voxel, die lokalen Diffusionstensoren.

3. *Assemblierung der Matrix:*
   Die Matrix wird knotenweise auf dem Device assembliert und dort gespeichert.

4. *Lösen des Gleichungssystems:*
   Das resultierende Gleichungssystem wird mit Hilfe eines hochoptimierten parallelen Lösungs-Verfahrens (Kapitel 5.3.7) von dem Device gelöst. Die Schritte zwei bis vier werden wiederholt, falls der Filter mehrere Zeitschritte durchführen soll.

5. *Ausschreiben der Daten:*
   Die gefilterten Daten werden auf den Host kopiert, in das ursprüngliche Grauwertintervall konvertiert und anschließend gespeichert.

Die einzelnen Arbeitsschritte werden im Folgenden detailliert beschrieben. Der generelle Arbeitsablauf zeigt bereits, dass eine effiziente Implementierung des Filters in CUDA möglich ist, da der Datentransfer zwischen Host und Device sehr gering ist (vgl. Kapitel 3.8.1) und die einzelnen Rechenoperationen weitestgehend parallel ausgeführt werden können.

5.4.3 Einlesen der Daten

Die Daten werden eingelesen, in einfach genaue Fließkommazahlen aus dem Intervall $[0, 1]$ konvertiert und für einen gecachten Zugriff an die Textur

```
texture<float, 1, cudaReadModeElementType> texture_b;
```

gebunden.

5.4.4 Berechnen der Diffusionstensoren

Bei der Berechnung der Diffusionstensoren muss zunächst die Frage beantwortet werden, in welchen Punkten diese berechnet werden sollen. Nach der Diskretisierung (5.38) müssen die Diffusionskoeffizienten jeweils in den Integrationspunkten ip_k ausgewertet werden. Je nach Wahl der Integrationspunkte müssten dann bis zu zwölf Diffusionstensoren pro Kontrollelement berechnet werden, was einen extremen Rechen- und Speicheraufwand bedeuten würde [50, 101]. Alternativ können die Tensoren in jedem Gitterpunkt berechnet und dann in den Integrationspunkten interpoliert werden, indem die Eigenwerte und Eigenvektoren interpoliert und der Tensor dann nach Gleichung (5.9) in jedem Gitterpunkt berechnet wird. Dabei geht einerseits die Normierung der Eigenvektoren verloren, die unter einem großen Aufwand neu berechnet werden muss, andererseits ist die Interpolation aufgrund der Gefahr von Auslöschung von Vektoren numerisch nicht akzeptabel. Beide Aspekte sind in [101] eingehend diskutiert. Somit bleibt die Möglichkeit, die Tensoren als konstant auf jedem Element oder jedem Knoten anzunehmen und sie entsprechend zu berechnen. Wie in [101] vorgeschlagen werden bei der in dieser Arbeit diskutierten Implementierung des trägheitsbasierten Diffusionsfilters die Diffusionstensoren als konstant auf jedem Knoten angenommen. Ein großer numerischer Einfluss dieses Vorgehens ist nicht zu erwarten, da dies lediglich einer geringfügigen Verschiebung des Integrationsgebietes bei der Integration der Trägheitstensoren, die wie oben beschrieben zur Berechnung der Diffusionstensoren verwendet werden, entspricht. Interessant wäre allerdings eine genauere Untersuchung der resultierenden Systemmatrix im Falle von elementweise konstanten Diffusionstensoren, da diese aufgrund der symmetrischen Wahl der Integrationspunkte und der symmetrischen Ansatzfunktionen ebenfalls symmetrisch wird. In der in dieser Arbeit verwendeten Variante der knotenweise konstanten Diffusionstensoren ist im Allgemeinen die Systemmatrix jedoch nicht symmetrisch. Zur knotenweisen Berechnung der Diffusionstensoren wird das Feld `cuda_tensor` auf dem Device alloziert, das sechs Einträge pro Knoten, die lexikographisch angeordnet sind, speichern kann. Hierbei wird die Symmetrie der Tensoren ausgenutzt. Die Berechnung erfolgt in einem Kernel, der für jeden einzelnen Gitterknoten ausgeführt wird. Der Kernel besteht aus zwei Teilen:

- Integration des diskreten Trägheitstensors D_I nach Formel (5.7).
- Berechnen der Eigenwerte und Eigenvektoren des Trägheitstensors D_I und anschließendes Berechnen des Diffusionstensors D nach Formel (5.9).

```
__global__ void tensor_kernel(float * tensor,
                              const int block_z,
                              const int size_x,
                              const int size_y,
                              const int size_z,
                              const int rho) {


  // Berechnen der Koordinaten des aktuellen Knotens:
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  int idz = block_z * blockDim.z + threadIdx.z;

  // Diskreter Trägheitstensor und Initialisierung:
  float D_I[3][3];

  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++)
      D_I[i][j] = 0.0f;

  // Gesamtmasse im Integrationsgebiet:
  float mass = 0.0f;

  // Größe des Integrationsgebietes in den sechs Raumrichtungen:
  // Kleinbuchstaben: negative Richtung
  // Großbuchstaben: positive Richtung
  int rho_x = rho;
  int rho_X = rho;
  int rho_y = rho;
  int rho_Y = rho;
  int rho_z = rho;
  int rho_Z = rho;

  // Abschneiden des Integrationsgebietes am Rand des Bildes:
  while((idx - rho_x) < 0) rho_x --;
  while((idx + rho_X) >= size_x) rho_X --;
  while((idy - rho_y) < 0) rho_y --;
  while((idy + rho_Y) >= size_y) rho_Y --;
  while((idz - rho_z) < 0) rho_z --;
  while((idz + rho_Z) >= size_z) rho_Z --;

  // Berechnung der diskreten Integration:
```

```
  for(int z = -rho_z; z <= rho_Z; z++)  {
    for(int y = -rho_y; y <= rho_Y; y++) {
      for(int x = -rho_x; x <= rho_X; x++) {

        // Auslesen der Grauwerte der einzelnen Voxel:
        float m = tex1Dfetch(texture_b,
          (idz+z) * size_x * size_y + (idy+y) * size_x + (idx+x));

        // Berechnung der Gesamtmasse im Integrationsgebiet:
        mass += m;

        // Berechnen der Koeffizienten des
        // diskreten Trägheitstensors:
        D_I[0][0] += m * (y * y + z * z);
        D_I[1][1] += m * (x * x + z * z);
        D_I[2][2] += m * (x * x + y * y);
        D_I[0][1] -= m * x * y;
        D_I[0][2] -= m * x * z;
        D_I[1][2] -= m * y * z;

      } // for z
    } // for y
  } // for x

  // Berechnen des Diffusionstensors:

  ...

}
```

Die Berechnung der Eigenwerte und Eigenvektoren des diskreten Trägheitstensors wird von einer leicht modifizierten Version des in [89] vorgeschlagenen Codes zur Hauptachsentransformation symmetrischer Matrizen durchgeführt, der dort nachgelesen werden kann.

5.4.5 Assemblierung der Matrix

Wie oben beschrieben, wird die Matrix knotenweise assembliert. Dies hat den Vorteil, dass der Assemblierungsalgorithmus unabhängig voneinander auf jedem Knoten operiert und somit trivial parallelisiert werden kann. Es wird daher für jeden Knoten ein Thread ausgeführt, der in dem entsprechenden Knoten alle Kopplungskoeffizienten mit den Nachbarknoten berechnet und dann in dem Feld `cuda_matrix` ablegt. Die knotenweise Assemblierung hat dabei zudem den Vorteil, dass jeder Matrixkoeffizient komplett berechnet und nur einmal in den globalen Speicher geschrieben werden muss, was eine minimale Anzahl von

Schreiboperationen in den globalen Speicher bedeutet. Bei einer elementweisen Assemblierung, bei der auf jedem Element ein Thread operiert, steuern verschiedene Threads einzelne Summanden zu bestimmten Matrixeinträgen bei. Somit sind einerseits mehrere Schreiboperationen pro Matrixkoeffizient auf den globalen Speicher nötig und andererseits müssen diese Schreiboperationen atomar ausgeführt werden, um sicherzustellen, dass nicht mehrere Threads zur gleichen Zeit auf dieselbe Speicheradresse schreiben. Ein Laufzeitvergleich der beiden Assemblierungsmethoden findet sich in Kapitel 5.7.3. Bei der knotenweisen Assemblierung für ein isotropes Gitter mit Maschenweite $h = 1$ wird in jedem Knoten die Kopplung mit sich selbst und seinen 26 Nachbarknoten über die angrenzenden acht Referenzwürfel nach Formel (5.38) berechnet.

5.4.6 Lösen des Gleichungssystems

Das resultierende Gleichungssystem wird durch das oben beschriebene parallele Hybrid-Lösungs-Verfahren gelöst (Definition 5.8). Hierbei wird jeder Eintrag des Vektors x^{k+1} in einem separaten Thread berechnet. Die Reihenfolge sowie die Parallelität der Ausführung der einzelnen Threads hängt hierbei vom CUDA-Treiber und der bei der Ausführung verwendeten Hardware ab.

```
// DEVICE CODE:

 __global__ void solver_kernel(float * matrix,
                               float * x,
                               float * b,
                               const int block_z,
                               const int size_x,
                               const int size_y,
                               const int size_z) {


  // Berechnen der Koordinaten des aktuellen Gitterknotens:
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  int idz = block_z * blockDim.z + threadIdx.z;

  // Berechnen der eindimensionalen Adressen, um die Daten
  // des aktuellen Knotens im Feld x und im Feld matrix auslesen
  // zu können:
  int index_x = idz * size_x * size_y + idy * size_x + idx;
  int index_matrix = 27 * index_x;

  // Temporärer Speicher, um Werte der Nachbarn des aktuellen
  // Knotens, multipliziert mit den entsprechenden
```

```
  // Matrix-Koeffizienten, zu speichern:
  float temp[27];

  // Werte aller Nachbarn um entsprechende Matrixkoeffizienten
  // auszulesen. Die Funktion get_index bestimmt dabei den
  // eindimensionalen Index des i-ten Nachbarn und gibt -1
  // zurück, falls der aktuelle Knoten am Rand liegt und der
  // i-te Nachbar dadurch nicht existiert:
  for(int i = 1; i < 27; i++) {
     int index = get_index_x(i,idx,idy,idz,size_x,size_y,size_z);
     if(index != -1) temp[i] = x[index] * matrix[index_matrix+i];
     else temp[i] = 0.f;
  }

  // Speichern des Diagonaleintrags der Matrix:
  temp[0] = matrix[index_matrix];

  // In der Variable result wird das Ergebnis des
  // Iterationsschritts berechnet:
  float result = b[index_x];

  // Iterationsvorschrift:
  for(int i = 1; i < 27; i++)
    result -= temp[i];

  // Dividieren durch das Diagonalelement:
  result /= temp[0];

  // Speichern des berechneten Wertes:
  x[index_x] = result;

}
```

Bei genauerer Betrachtung dieses Kernels fällt auf, dass für jeden Iterationsschritt sehr viele Speicheroperationen notwendig sind, die in CUDA teuer sind und damit die Ausführung des Kernels sehr langsam machen. Folgende Beobachtungen helfen die Implementierung des Lösers in CUDA bezüglich der Laufzeit zu optimieren:

- Threads, die auf benachbarten Knoten operieren, greifen teilweise auf dieselben Daten zu.
- Bei jedem Iterationsschritt wird die komplette Matrix aus dem globalen Speicher geladen.

Es ist daher sinnvoll alle Daten, die innerhalb eines Thread-Blocks bearbeitet werden, nur einmal aus dem globalen Speicher zu laden und über den Shared-Speicher den einzelnen Threads zur Verfügung zu stellen und zudem mehrere Iterationsschritte in einem Kernel-Aufruf durchzuführen. Der Versuch, den Zugriff auf die Matrixkoeffizienten durch den Texturcache zu beschleunigen hatte keinen Laufzeitgewinn zur Folge. Dies liegt daran, dass bei jedem Kernel-Aufruf jedes Matrixelement genau einmal aus dem globalen Speicher geladen wird. Die Verwendung eines Caches bringt dadurch keinen Nutzen. In der folgenden Implementierung des Lösers werden in jedem Thread-Block alle benötigten Werte der Unbekannten `x` in den Shared-Speicher geladen. Die global deklarierten Konstanten `THREADS_X`, `THREADS_Y` und `THREADS_Z` legen die Dimensionen der Thread-Blöcke in die entsprechenden Koordinatenrichtungen fest. Anschließend werden die Threads innerhalb eines Blocks synchronisiert und α Iterationsschritte berechnet.

```
// DEVICE CODE:

__global__ void solver_kernel(const float * matrix,
                              float * x,
                              const int block_z,
                              const int size_x,
                              const int size_y,
                              const int size_z,
                              const int alpha) {

  // Shared-Memory in Größe des Blocks + Randwerte des Blocks:
  __shared__ float t[THREADS_X + 2][THREADS_Y + 2][THREADS_Z + 2];

  // Berechnen der Koordinaten des aktuellen Gitterknotens:
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  int idz = block_z * blockDim.z + threadIdx.z;

  // Indizierung des Shared-Memory +1 wegen Rändern:
  int ii = threadIdx.x + 1;
  int jj = threadIdx.y + 1;
  int kk = threadIdx.z + 1;

  // Entsprechender Index im eindimensionalen Feld x:
  int index_x = idz * size_x * size_y + idy * size_x + idx;

  // Index im Matrixfeld:
  int index_matrix = 27 * index_x;
```

```
 // Die rechte Seite bleibt immer konstant und wird aus der
 // entsprechenden Textur ausgelesen:
 float bb = tex1Dfetch(texture_b, index_x);

// Die Matrixzeile ändert sich ebenfalls nicht, daher wird
// sie lokal gecacht:
 float M[27];
 for(int i = 0; i < 27; i++)
   M[i] = matrix[index_matrix + i];

 // Jeder Thread lädt seinen zugehörigen Eintrag aus dem
 // Feld x in den Shared-Speicher:
 t[ii][jj][kk] = x[index_x];
 // Threads die am Rand liegen, laden zudem die Werte auf dem
 // Rand in den Shared-Memory. Werte außerhalb des Bildes
 // werden auf 0.f gesetzt:

 ...


 // Warten bis alle Threads die Daten geladen haben:
 __syncthreads();

 // Nun werden in dem Block alpha Iterationen ausgeführt:
 for(int i = 0; i < alpha; i++) {

    float result = bb
                - t[ii+1][jj][kk] * M[1]
                - t[ii+1][jj+1][kk] * M[2]
                - t[ii][jj+1][kk] * M[3]
                - t[ii-1][jj+1][kk] * M[4]
                - t[ii-1][jj][kk] * M[5]
                - t[ii-1][jj-1][kk] * M[6]
                - t[ii][jj-1][kk] * M[7]
                - t[ii+1][jj-1][kk] * M[8]
                - t[ii][jj][kk+1] * M[9]
                - t[ii+1][jj][kk+1] * M[10]
                - t[ii+1][jj+1][kk+1] * M[11]
                - t[ii][jj+1][kk+1] * M[12]
                - t[ii-1][jj+1][kk+1] * M[13]
                - t[ii-1][jj][kk+1] * M[14]
                - t[ii-1][jj-1][kk+1] * M[15]
                - t[ii][jj-1][kk+1] * M[16]
```

```
                - t[ii+1][jj-1][kk+1] * M[17]
                - t[ii][jj][kk-1] * M[18]
                - t[ii+1][jj][kk-1] * M[19]
                - t[ii+1][jj+1][kk-1] * M[20]
                - t[ii][jj+1][kk-1] * M[21]
                - t[ii-1][jj+1][kk-1] * M[22]
                - t[ii-1][jj][kk-1] * M[23]
                - t[ii-1][jj-1][kk-1] * M[24]
                - t[ii][jj-1][kk-1] * M[25]
                - t[ii+1][jj-1][kk-1] * M[26];

    t[ii][jj][kk] = result / M[0];

    // Synchronisierung der Threads innerhalb eines Blocks:
    __syncthreads();
  }

  // Nach alpha Iterationen werden die Daten in den globalen
  // Speicher zurückgeschrieben:
  x[index_x] = t[ii][jj][kk];

}
```

5.5 Speicherbedarf

Der Speicherbedarf des trägheitsbasierten anisotropen Diffusionsfilters ist sehr hoch. Für jedes Voxel des zu filternden Bildes müssen

- 6 Einträge für den Diffusionstensor
- 27 Matrixeinträge
- 1 Eintrag für die rechte Seite des Gleichungssystems
- 1 Eintrag für das Ergebnis

gespeichert werden. Bedenkt man, dass der Speicherplatz jedes Voxel ein Byte beträgt und die in einfacher Genauigkeit ausgeführte Rechnung vier Byte pro Eintrag benötigt, beträgt der Speicherbedarf 140mal der Größe des zu filternden Bildes. Aktuelle Zwei-Photonen-Mikroskope liefern Bilder in einer Auflösung von $2048 \times 2048 \times 368$ Voxel. Für diese, fast 1.5GByte großen Bilder, benötigt der trägheitsbasierte Diffusionsfilter etwa 210GByte Speicher. Eine Möglichkeit solche riesigen Datenmengen zu verarbeiten ist, die Matrix nicht explizit abzuspeichern, sondern die benötigten Elemente bei Bedarf neu zu berechnen. Da

das Berechnen der Diffusionstensoren und der Matrixeinträge allerdings sehr zeitaufwändig ist [50, 101], ist dieses Vorgehen nicht empfehlenswert. Es ist daher besser, das Bild in viele überlappende Teilbilder zu zerschneiden, diese separat zu filtern und mit einer geeigneten Interpolation an den Bildrändern wieder zusammenzusetzen (Kapitel 3.8.1). Aufgrund des lokalen Verhaltens, der dem Filter zugrunde liegenden nichtlinearen Wärmeleitungsgleichung für wenige kleine Zeitschritte, die zur Rauschglättung ausreichend sind, beeinträchtigt dieses Vorgehen das Resultat des Filters nicht. Dieses Vorgehen erlaubt zudem eine hochgradig parallele Implementierung des Filters, da die einzelnen Teilbilder auf viele Grafikkarten verteilt werden können.

5.6 Bestimmung der benötigten Lösungs-Iterationen

Wie oben beschrieben gilt ein lineares Gleichungssystem als gelöst, wenn der Defekt (5.49) kleiner als die Abbruchschwelle ϵ ist. Die Berechnung des Defekts ist hierbei so aufwändig, wie eine Matrix-Vektor-Multiplikation, zusammen mit einem Skalarprodukt [50]. Vor allem die Berechnung des Skalarprodukts ist auf Grafikhardware wenig effizient. Daher ist es sinnvoll, unabhängig von dem zu lösenden Gleichungssystem eine bestimmte Anzahl $\nu(\tau)$ Lösungs-Iterationen durchzuführen und dann das Gleichungssystem als hinreichend exakt gelöst zu betrachten. An dieser Stelle sei zudem in Erinnerung gerufen, dass die Aufgabe nicht darin besteht eine partielle Differentialgleichung möglicht exakt zu lösen, sondern einen schnellen Bildverarbeitungsoperator zu konstruieren. Die Abhängigkeit von der Zeitschrittlänge τ ist in [50] ausführlich diskutiert. Unter Verwendung des dort definierten projizierten Diffusionstensors ergibt sich

$$\nu(\tau) = 120\tau \tag{5.50}$$

als sehr großzügige Abschätzung für die benötigte Anzahl Lösungs-Iterationen. An dieser Stelle sei die Anzahl benötigter Lösungs-Iterationen für den Löser unter Berücksichtigung des Synchronisationsparameters α neu evaluiert. Hierzu sei das Konvergenzverhalten des Lösers mit Zeitschrittweise $\tau = 1$ zunächst auf zwei verschiedenen Datensätzen betrachtet, wobei der Filter einmal auf lineare (Abbildung 5.11) und einmal auf planare Strukturerkennung (Abbildung 5.12) eingestellt ist. Aufgrund der Einschränkung, dass die Grafikkarte in einfacher Genauigkeit rechnet, ist das Gleichungssystem nicht sonderlich exakt zu lösen. Wie exakt hängt offenbar von den zu bearbeitenden Daten und den Einstellungen des Filters ab (Abbildungen 5.11 bis 5.14). Ziel ist es unter Minimierung der Laufzeit eine Worst-Case Konstante ν und den Synchronisationsparameter α so zu bestimmen, dass unter der Durchführung von ν Lösungs-Iterationen das Gleichungssystem so exakt, wie es die arithmetische Genauigkeit der verwendeten Grafikprozessoren zulässt, gelöst wird. Zur Bestimmung dieser Worst-Case Konstante sei ein dreidimensionaler Datensatz mit Schachbrettmuster der Dimension 256^3 gewählt, der mit isotropen Filtereinstellungen und unterschiedlicher Zeitschrittweite gefiltert wird. Das Konvergenzverhalten des Lösers ist

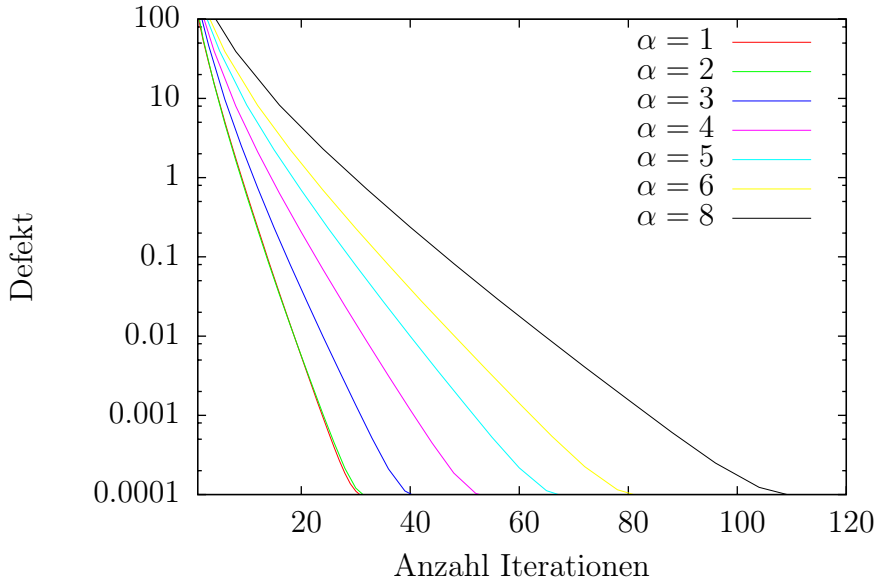


Abbildung 5.11: *Benötigte Lösungs-Iterationen zur Reduktion des Defekts bei einem Testdatensatz mit Zeitschrittlänge $\tau = 1$ und linearer Filtereinstellung unter Verwendung verschiedener Synchronisationsparameter $\alpha = 1, 2, 3, 4, 5, 6, 8$. Man beachte die logarithmisch aufgetragene y-Achse.*

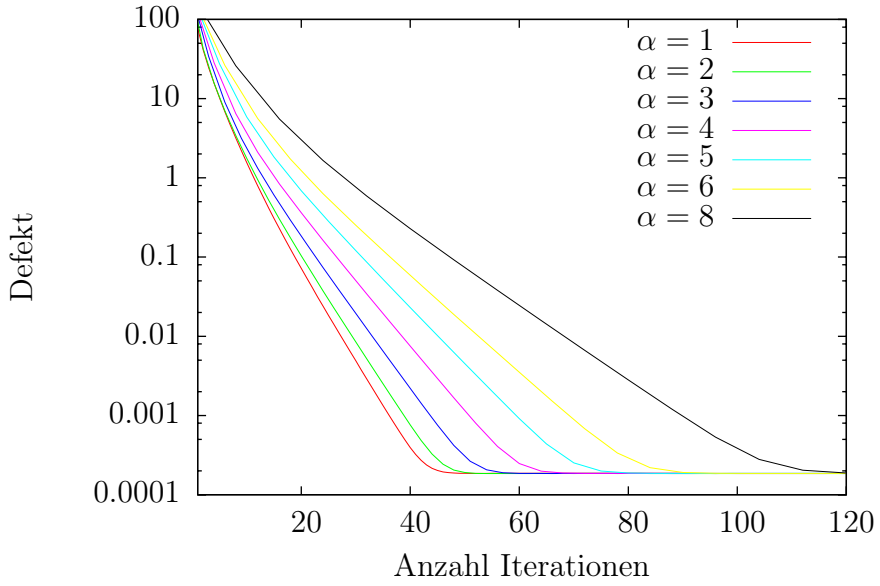


Abbildung 5.12: *Benötigte Lösungs-Iterationen zur Reduktion des Defekts bei einem Testdatensatz mit Zeitschrittlänge $\tau = 1$ und planarer Filtereinstellung unter Verwendung verschiedener Synchronisationsparameter $\alpha = 1, 2, 3, 4, 5, 6, 8$. Man beachte die logarithmisch aufgetragene y-Achse.*

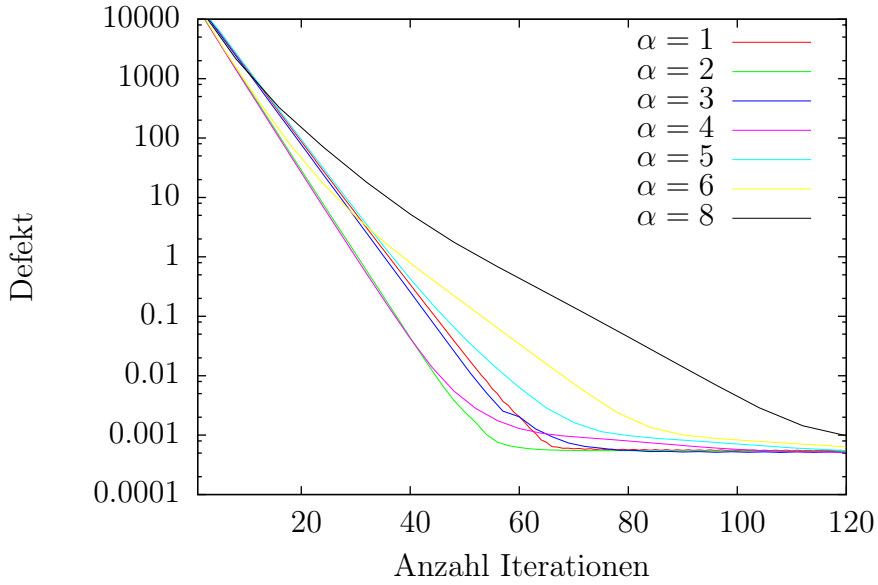


Abbildung 5.13: *Benötigte Lösungs-Iterationen zur Reduktion des Defekts bei dem Testdatensatz mit Schachbrettmuster mit Zeitschrittlänge $\tau = 1$ und isotroper Filtereinstellung unter Verwendung verschiedener Synchronisationsparameter $\alpha = 1, 2, 3, 4, 5, 6, 8$. Man beachte die logarithmisch aufgetragene y-Achse.*

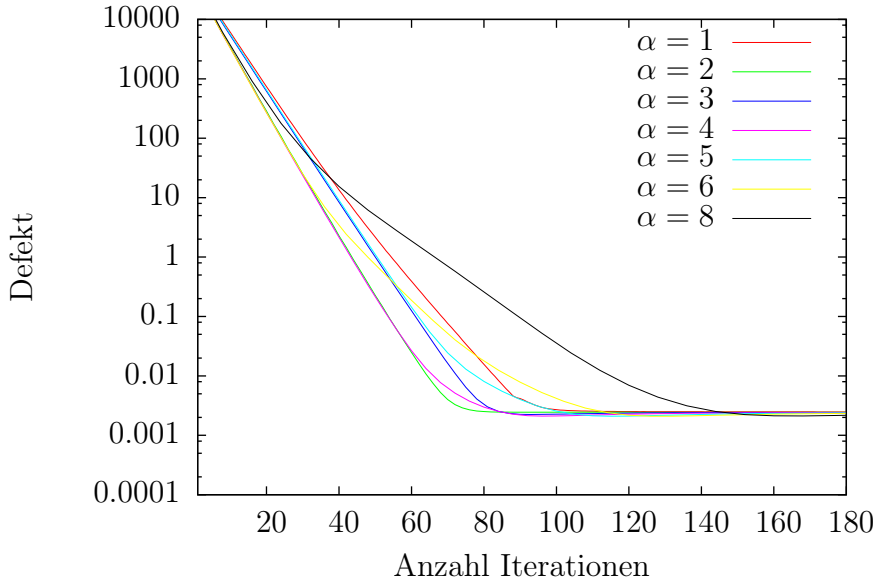


Abbildung 5.14: *Benötigte Lösungs-Iterationen zur Reduktion des Defekts bei dem Testdatensatz mit Schachbrettmuster mit Zeitschrittlänge $\tau = 2$ und isotroper Filtereinstellung unter Verwendung verschiedener Synchronisationsparameter $\alpha = 1, 2, 3, 4, 5, 6, 8$. Man beachte die logarithmisch aufgetragene y-Achse.*

für die Zeitschrittweiten $\tau = 1$ und $\tau = 2$ den Abbildungen 5.13 und 5.14 zu entnehmen. Im Gegensatz zu dem in [50] beschriebenen Filterschema, das den projizierten Diffusionstensor verwendet, hängt die Anzahl der benötigten Iterationen bei der Verwendung des voll besetzten Diffusionstensors nicht linear von der Zeitschrittweite τ ab. Zudem wird die beste Konvergenz des Lösers für $\alpha = 2$ erreicht, wobei ungefähr 70 Iterationen nötig sind, um den Defekt maximal stark zu reduzieren. Um Rundungsfehler an den Grenzen der Thread-Blöcke zu minimieren, werden anschließend noch vier voll synchronisierte Lösungs-Iterationen auf das Gleichungssystem angewandt.

5.7 Laufzeitanalyse

5.7.1 Laufzeitvergleich zwischen CPU und GPU Implementierung

Zunächst sei die CPU Implementierung des Filters mit der GPU Implementierung verglichen. Beide Implementierungen sind hochgradig optimiert und somit deutlich schneller als die ursprüngliche *NeuRA* Implementierung [20]. Zudem sei erwähnt, dass die CPU Implementierung ein SOR-Verfahren mit Aufwandschätzer [50] zur Lösung des Gleichungssystems verwendet, das deutlich weniger Lösungs-Iterationen erfordert als das Lösungs-Verfahren, das in der GPU Implementierung zum Einsatz kommt. Dennoch benötigt die GPU Implementierung für den in [21] betrachteten Datensatz der Dimension $256 \times 256 \times 216$ mit Eingabeparametern $\rho = 5$, $\tau = 1$, $n_t = 4$ lediglich 70 Sekunden auf einer Tesla-C1060 Karte [79]. Die CPU Implementierung des Filters benötigt bei gleichen Eingabeparametern auf einem Kern eines Intel Xeon 3GHz Prozessors etwa 113 Minuten zum Filtern dieses Datensatzes. Die GPU Implementierung ist somit knapp 100-mal schneller, als die CPU Implementierung, was zwei Größenordnungen entspricht. Bedenkt man, dass eine Tesla-C1060 Karte über 240 Rechenwerke verfügt, die mit 1.3GHz getaktet sind [79], ergibt sich, verglichen mit einem Kern eines 3GHz Prozessors, maximal ein Faktor von 104, den die GPU realistischerweise schneller sein kann, als der einzelne Prozessorkern. Dies bedeutet, dass die hier beschriebene Implementierung des trägheitsbasierten Filters die Kapazität moderner Grafikkarten nahezu optimal ausnutzt.

5.7.2 Verwendung des Texturcaches bei der Integration der Trägheitstensoren

Der Zugriff auf Daten aus dem globalen Speicher der Grafikkarte ist nicht gecacht. Daher dauert es 400 bis 600 Taktzyklen, um ein Datum aus dem globalen Speicher zu lesen [78]. Der Zugriff auf den schreibgeschützten Texturspeicher ist mit einer Zugriffszeit von 100 Taktzyklen deutlich schneller. Zudem werden die dort hinterlegten Daten gecacht. Der Zugriff auf ein gecachtes Datum ist innerhalb eines Taktes möglich [78]. Bei der Integration der Trägheitstensoren greifen benachbarte Threads auf die gleichen Daten zu. Die Verwendung des Texturcaches verringert die Laufzeit für diesen Teil des Algorithmus erheblich, vor

ρ	**Textur Speicher**	**Globaler Speicher**
3	$45s$	$47s$
5	$70s$	$76s$
8	$156s$	$180s$
10	$260s$	$306s$

Tabelle 5.2: *Laufzeit zum Filtern eines Datensatzes der Dimension* $256 \times 256 \times 216$ *auf einem Tesla-C1060 Prozessor, mit Parametern* $\tau = 1$, $n_t = 4$ *und variierender Integrationsgröße* ρ *unter Verwendung von Textur Speicher und globalem Speicher.*

allem für große Integrationsgebiete (Tabelle 5.2), da hier die Laufzeit zur Tensorintegration die Laufzeit zum Lösen des Gleichungssystems deutlich übersteigt.

5.7.3 Elementweise und Knotenweise Assemblierung

Elementweise Assemblierung	**Knotenweise Assemblierung**
$16.3s$	$1.9s$

Tabelle 5.3: *Laufzeitvergleich zum Assemblieren der Systemmatrix eines Datensatzes der Dimension* $256 \times 256 \times 216$ *auf einem Tesla-C1060 Prozessor zwischen elementweiser und knotenweiser Assemblierung.*

Wie in den Kapiteln 5.3.5 und 5.4.5 beschrieben reduziert die knotenweise Assemblierung die Anzahl der Speicherzugriffe bei der Generierung der Systemmatrix auf ein Minimum. Zudem sind keine atomaren Speicheroperationen nötig, da jeder Thread genau eine Matrixzeile erzeugt und diese dann in den globalen Speicher schreibt. Verblüffend ist allerdings, dass durch diese effiziente Speichernutzung die knotenweise Assemblierung fast zehnmal schneller ist als die elementweise Assemblierung (Tabelle 5.3), obwohl bei der knotenweisen Assemblierung redundante Berechnungen ausgeführt werden (Kapitel 5.3.5).

5.7.4 Asynchrone Lösungs-Iterationen

Synchronisationsparameter α	**Zeit zum Lösen**
1	$16s$
2	$8.2s$
4	$4.3s$
8	$2.4s$

Tabelle 5.4: *Laufzeit für 70 Lösungs-Iterationen (vergleiche Abschnitt 5.6) auf einem Gleichungssystem mit* 14155776 *Unbekannten, bei variierendem Synchronisationsparameter* α.

Jeder Aufruf des Löser-Kernels führt α Lösungs-Iterationen aus, bei denen nur die Daten innerhalb eines Thread-Blocks über den Shared-Speicher synchronisiert werden. Wie oben gezeigt, wird das Gleichungssystem für $\tau = 1$ nach

$$\frac{70}{\alpha} \tag{5.51}$$

Aufrufen des Löser-Kernels als hinreichend exakt gelöst betrachtet. Nach jedem Kernel-Aufruf findet implizit eine Synchronisation aller Daten statt. Am Ende des Iterationsvorgangs werden vier abschließende, voll synchronisierte Lösungs-Iterationen ausgeführt, um Rundungsfehler an den Rändern der Thread-Blöcke zu unterbinden. Wie oben gezeigt, besitzt der Löser für $\alpha = 2$ das beste Konvergenzverhalten, wobei für eine höhere Wahl des Synchronisationsparameters die Laufzeit erheblich gedrückt werden kann (Tabelle 5.4). Es zeigt sich aber, dass für eine Wahl von $\alpha = 4$ oder $\alpha = 8$ Rundungsfehler an den Grenzen der Thread-Blöcke auftreten, weshalb in der aktuellen Implementierung des GPU-beschleunigten trägheitsbasierten Diffusionsfilters der Synchronisationsparameter $\alpha = 2$ zum Einsatz kommt.

5.7.5 Filtern großer Datenmengen

ρ	**Laufzeit**
3	$133s$
5	$166s$
8	$288s$
10	$434s$

Tabelle 5.5: *Laufzeit zum Filtern eines Bildes der Dimension* $2048 \times 2048 \times 368$ *mit den üblichen Filterparametern* $\tau = 1$, $n_t = 4$ *und variierender Integrationsgröße* ρ, *unter Verwendung des Clusters* Scout *mit 96 Tesla-C1060 Karten.*

Aktuelle Zwei-Photonen-Mikroskope können einzelne Aufnahmen mit einer Auflösung von $2048 \times 2048 \times 368$ erzeugen. Solch ein Bild wird in 96 Teilbilder der Dimension $360 \times 272 \times 200$, inklusive einem Überlapp von 16 Voxel in jede Raumrichtung, zerschnitten und auf die 96 Tesla-C1060 Karten des Clusters *Scout* via MPI [41] verteilt und parallel gefiltert. Der Speicherbedarf jeder einzelnen Karte beträgt dabei knapp 2.8GByte. Die Laufzeit des Filters, inklusive Laden und Speichern des Bildes und Interpolation in den überlappenden Bereichen der Teilbilder, ist Tabelle 5.5 zu entnehmen.

5.8 Vergleich mit der ursprünglichen Implementierung

Abgesehen von Rundungsfehlern, die durch die Verwendung der einfach genauen Arithmetik herrühren, sind die Resultate des ursprünglich in *NeuRA* [20] implementierten Version und der GPU-basierten Implementierung des trägheitsbasierten Diffusionsfilters gleich.

(a) **(b)** **(c)**

Abbildung 5.15: *Vergleich zwischen ursprünglicher Implementierung des trägheitsbasierten Diffusionsfilters und der GPU-basierten Implementierung. Das Beispiel zeigt, dass die trägheitsbasierte Diffusion sowohl Lücken schließt, als auch die Größe von Strukturen erhält. (a) Schnittebenenaufnahme des Testbilds. (b) Nach Anwendung der ursprünglichen Implementierung. (c) Nach Anwendung der GPU-basierten Implementierung.*

Hierzu sei die in den Vorarbeiten [50, 101] eingeführte Geometrie eines Ypsilons mit Lücken betrachtet. Das weiße Ypsilon befindet sich in schwarzem Hintergrund und besitzt eine Ausdehnung von jeweils drei Voxel in jede Raumrichtung. Zudem enthält es an drei Stellen jeweils Lücken, die ebenfalls drei Voxel groß sind. Der Datensatz hat die Dimension 64^3. Der trägheitsbasierte Filter wurde mit Parametern $\rho = 5$, $\tau = 2$ und $n_t = 2$ auf das Bild angewandt (vgl. [101]), wobei deutlich wird, dass die trägheitsbasierte Diffusion sowohl Lücken schließt, als auch die Größe von Strukturen erhält (Abbildung 5.15).

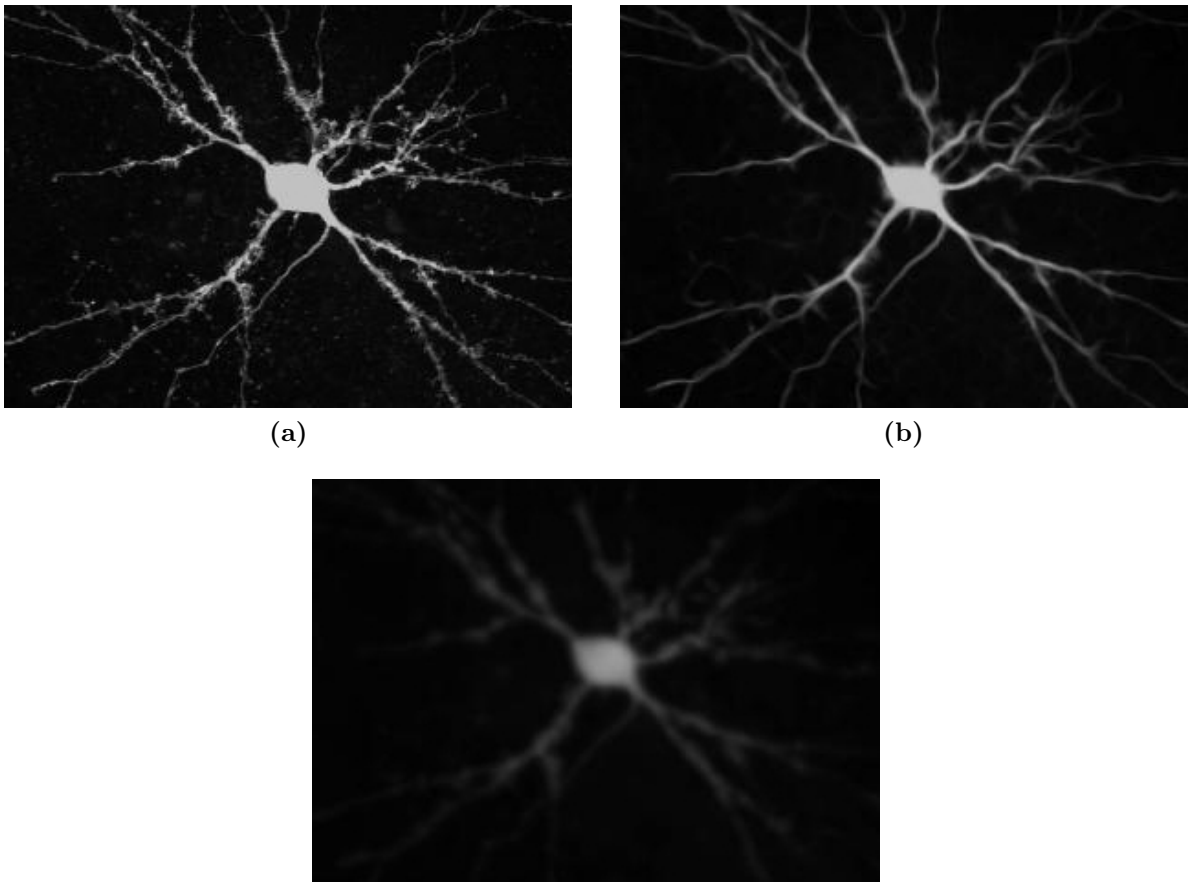
5.9 Numerische Optimierung der Parameter

Das Resultat des trägheitsbasierten Filters hängt von den Eingabeparametern

- ρ – Größe des Integrationsgebietes zur Strukturerkennung
- τ – Länge eines Zeitschritts
- n_t – Anzahl zu berechnender Zeitschritte
- η_1, η_2, η_3 – Diffusionskoeffizienten

ab. Je nachdem, ob lineare oder planare Geometrien zu filtern sind, werden die Diffusionsparameter auf $\eta_1 = 1$, $\eta_2 = \eta_3 = \epsilon$ bzw. $\eta_1 = \eta_2 = 1$, $\eta_3 = \epsilon$ eingestellt, was in den Vorarbeiten [50, 90, 101] eingehend diskutiert ist. Für Anzahl und Länge der Zeitschritte wurde in allen Vorarbeiten $\tau = 1$ und $n_t = 4$ evaluiert. Ebenso wurde der Strukturerkennungsparameter ρ in den besagten Vorarbeiten auf den Wert 5 optimiert, was sich bei Betrachtung der in dieser Arbeit vorgestellten Ergebnisse bestätigt.

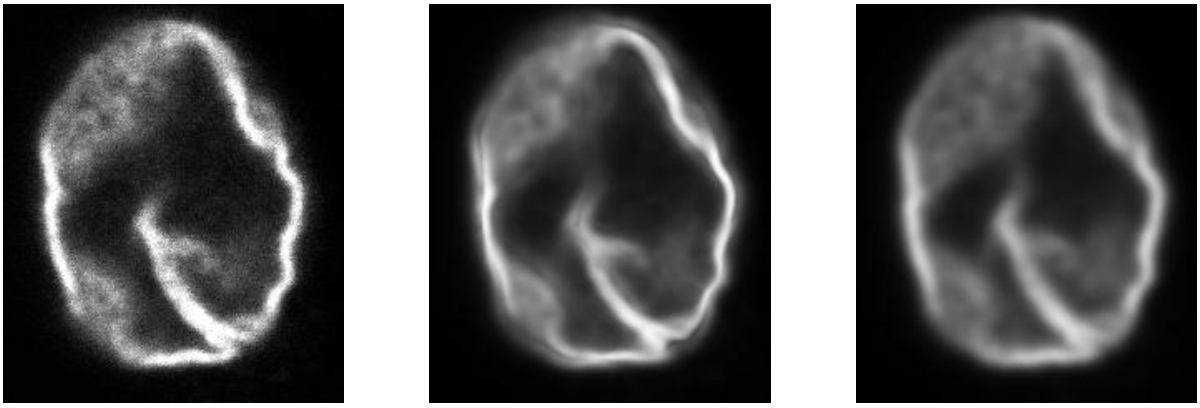
5.10 Ergebnisse



(a) **(b)** **(c)**

Abbildung 5.16: *(a) Volumenprojektion einer Neuronenzelle. (b) Nach Anwendung von trägheitsbasierter Diffusion mit Parametern $\rho = 5$, $\tau = 1$, $n_t = 4$, $\eta_1 = 1$, $\eta_2 = \eta_3 = \epsilon$. (c) Nach Anwendung von isotroper Diffusion. Quelle:* [52]

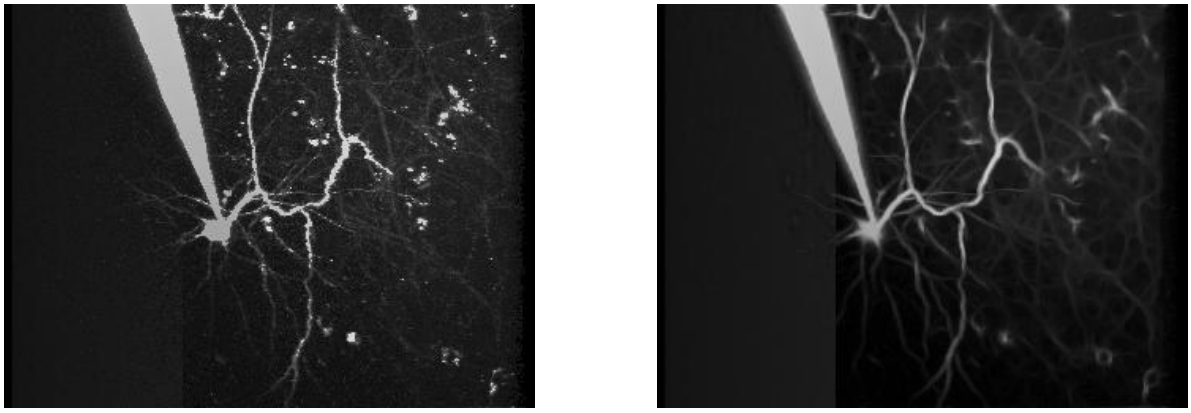
Im abschließenden Abschnitt dieses Kapitels finden sich die Resultate des trägheitsbasierten Diffusionsfilters, angewandt auf verschiedene Mikroskopdaten, wieder. Zunächst sei der Filter mit dem Prozess der isotropen Diffusion, jeweils für lineare und planare Strukturerkennung, verglichen. Hierzu wurde der Filter auf die Aufnahme einer Neuronenzelle mit linearer Strukturerkennung ($\eta_1 = 1$, $\eta_2 = \eta_3 = \epsilon$, Abbildung 5.16) und auf die Aufnahme eines Neuronenzellkerns mit planarer Strukturerkennung ($\eta_1 = \eta_2 = 1$, $\eta_3 = \epsilon$, Abbildung 5.17) angewandt. Zudem wurden beide Bilder mit isotroper Diffusion gefiltert. Bei Betrachtung der Abbildungen 5.16 und 5.17 ist eine deutliche Rauschverminderung des trägheitsbasierten Bildes zur Rohaufnahme zu erkennen. Verglichen mit der isotropen Diffusion gelingt es dem trägheitsbasierten Filter jedoch gleichzeitig die Größe der im Bild vorhandenen Strukturen und den im Bild vorhandenen Kontrast zu erhalten. Abbildung 5.18 zeigt die Volumenprojektion einer Neuronenzelle einer lebenden Maus, sowie die



(a) **(b)** **(c)**

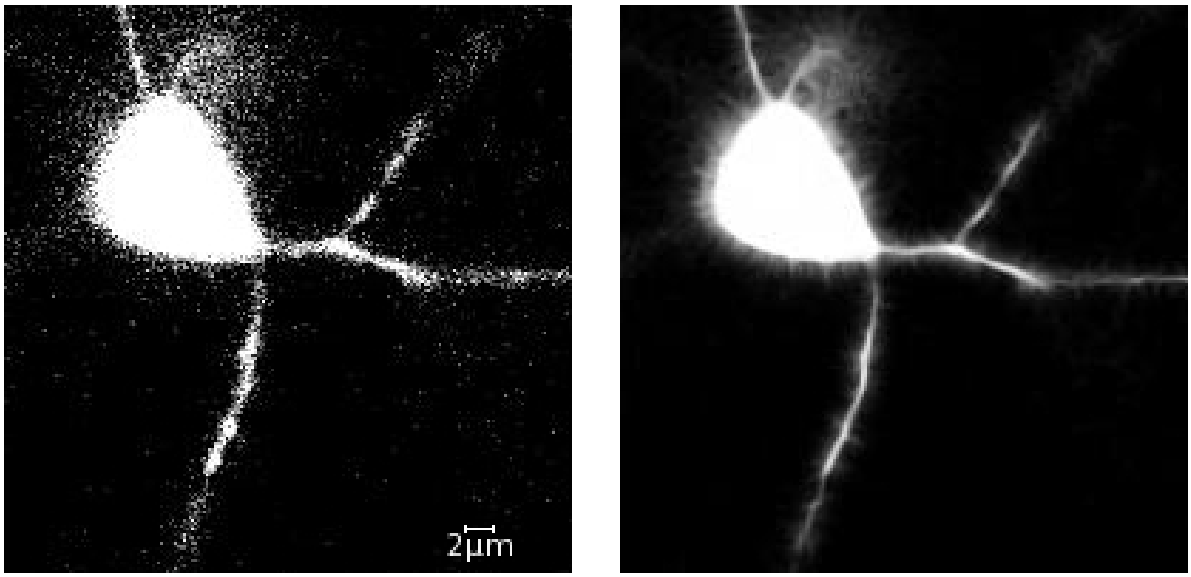
Abbildung 5.17: *(a) Schnittebenenbild eines Neuronenzellkerns. (b) Nach Anwendung von trägheitsbasierter Diffusion mit Parametern $\rho = 5$, $\tau = 1$, $n_t = 4$, $\eta_1 = \eta_2 = 1$, $\eta_3 = \epsilon$. (c) Nach Anwendung von isotroper Diffusion.*

Aufnahme nach Anwendung von trägheitsbasierter Diffusion. Bei dieser Aufnahme ist das Signal-zu-Rausch-Verhältnis besonders schlecht, da sich die Maus während der Aufnahme bewegen kann. Zudem ist deutlich die Pipette zu erkennen mit der das zur Mikroskopie notwendige Kontrastmittel gespritzt wurde. In Abbildung 5.19 ist das Schnittebenenbild einer hochaufgelösten Mikroskopaufnahme der Dimension $2048 \times 2048 \times 368$ sowie die Anwendung des Filters unter variierendem Strukturerkennungsparameter ρ aufgezeigt. Bei genauer Betrachtung fällt auf, dass bei der Parametereinstellung $\rho = 5$ der beste Kompromiss zwischen korrektem Filtern filigraner Strukturen und Rauschverminderung getroffen wird. Dies entspricht dem Resultat der einschlägigen Vorarbeiten.



(a) **(b)**

Abbildung 5.18: *(a) Volumenprojektion einer Neuronenzelle einer lebenden Maus. Es ist deutlich die Pipette zu sehen mit der das Kontrastmittel gespritzt wurde. (b) Nach Anwendung von trägheitsbasierter Diffusion mit Parametern $\rho = 5$, $\tau = 1$, $n_t = 4$, $\eta_1 = 1$, $\eta_2 = \eta_3 = \epsilon$.*



(a) **(b)**

(c) **(d)**

Abbildung 5.19: *Schnittebenenbild einer hochaufgelösten Aufnahme einer Neuronenzelle. (a) Rohaufnahme. (b), (c), (d) Nach Anwendung von trägheitsbasierter Diffusion mit Parametern* $\tau = 1$, $n_t = 4$, $\eta_1 = 1$, $\eta_2 = \eta_3 = \epsilon$ *und (b)* $\rho = 3$, *(c)* $\rho = 5$ *und (d)* $\rho = 8$.

Kapitel 6

Multiskalen-Scharfzeichnung

Ausgehend von der Arbeit *Multiscale Vessel Enhancement Filtering* [37] ist in diesem Kapitel ein *Multiskalen-Scharfzeichner* vorgestellt, der lineare Strukturen, wie die in Neuronenaufnahmen vorhandenen Dendritenäste und dendritischen Dornfortsätze, hervorhebt und dabei Rauschen unterdrückt. Dieser weitere Vorfilterungsprozess ermöglicht eine anschließende Segmentierung hochaufgelöst mikroskopierter Neuronenzellen. Da die Zellkerne dieser Neuronenzellen nicht einer linearen Struktur entsprechen, wird der Scharfzeichnungsprozess mit einer globalen Schwellwertsegmentierung (Kapitel 4.2.2 und 7.1) kombiniert, sodass die Zellkerne erhalten bleiben. Zudem muss das Histogramm des resultierenden Bildes normalisiert werden (Kapitel 4.2.2), damit dieses einen vernünftigen Kontrast aufweist. Diese beiden Erweiterungen des ursprünglichen Scharfzeichnungsfilters sind in Kapitel 6.5 detailliert erklärt. Der Scharfzeichner ist verglichen mit dem trägheitsbasierten Diffusionsfilter (Kapitel 5) sehr schnell und lässt sich leicht parallelisieren, da die Operationen in den einzelnen Bildpunkten unabhängig voneinander ausgeführt werden.

6.1 Multiskalen-Analyse

Um Strukturen verschiedener Größe in einem Bild adäquat zu detektieren, bietet sich eine Multiskalen-Analyse der zu extrahierenden Bildinformation an (vgl. Kapitel 4.5). Die verschiedenen Skalen σ eines n-dimensionalen Bildes sind dabei als diskrete Faltung der Bildinformation mit dem Gauß-Kern (vgl. Kapitel 4.3.1)

$$g(x,\sigma) = \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}}\, e^{\frac{-\|x\|^2}{2\sigma^2}} \tag{6.1}$$

definiert. Hierbei entspricht σ^2 der Varianz der Gaußschen Normalverteilung.

6.2 Lokale Merkmalsextraktion

Ein typischer Ansatz das lokale Verhalten eines Bildes u zu untersuchen, ist die Betrachtung der Taylorreihe

$$u_\sigma(x + \Delta x) \approx u_\sigma(x) + \Delta x^T \nabla u_\sigma(x) + \Delta x^T \mathcal{H} u_\sigma(x) \Delta x \tag{6.2}$$

von u_σ auf der Skala σ um den Entwicklungspunkt x bis zur zweiten Ordnung. Hierbei bezeichnen $\nabla u_\sigma(x)$ den Grauwertgradienten (Kapitel 4.4.1) und $\mathcal{H} u_\sigma(x)$ die Hesse-Matrix (Kapitel 4.4.2) des Bildes u_σ im Entwicklungspunkt x. Die zur Berechnung des Grauwertgradienten und der Hesse-Matrix notwendigen partiellen Ableitungen $\frac{\partial}{\partial x_i}$ sind dabei als Faltung der Bilddaten u mit den Ableitungen des Gauß-Kerns (6.1)

$$\frac{\partial}{\partial x_i} u_\sigma(x) = u(x) * \frac{\partial}{\partial x_i} g(x, \sigma) \tag{6.3}$$

definiert. Gleichung (6.3) kann [63] folgend mit einer von der Skala σ abhängigen Gewichtung $\gamma(\sigma)$ kombiniert werden, um die Merkmale bestimmter Skalen zu bevorzugen oder zu benachteiligen. Die Analyse der zweiten Ableitung $\mathcal{H}$ auf der Skala σ misst den Kontrast zwischen den Bereichen innerhalb und außerhalb des Intervalls $(-\sigma, \sigma)$ in Richtung des Grauwertgradienten (Abbildung 6.1).

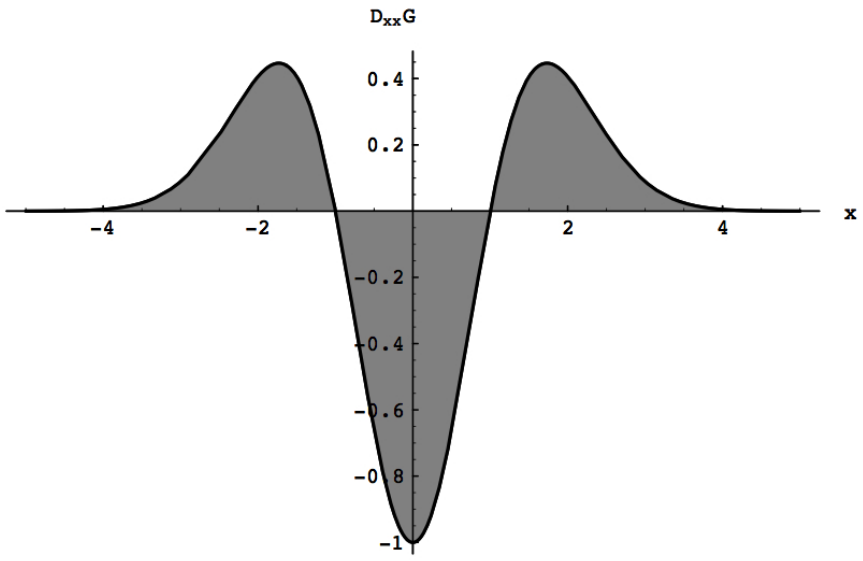


Abbildung 6.1: *Die zweite Ableitung des Gauß-Kerns misst den Kontrast innerhalb und außerhalb des Intervalls* $(-\sigma, \sigma)$*. Hier:* $\sigma = 1$*. Quelle:* [37]

λ_1	λ_2	λ_3	**Orientierung**
0	0	$-$	Helle planare Struktur
0	$-$	$-$	Helle lineare Struktur
$-$	$-$	$-$	Helle punktartige Struktur
0	0	$+$	Dunkle planare Struktur
0	$+$	$+$	Dunkle lineare Struktur
$+$	$+$	$+$	Dunkle punktartige Struktur
Andere			Keine Struktur (Rauschen)

Tabelle 6.1: *Lokale Strukturen in Abhängigkeit der Eigenwerte λ_i (0: klein; $+$: betragsmäßig groß, positiv; $-$: betragsmäßig groß, negativ).*

6.3 Eigenwertanalyse der Hesse-Matrix

Die Hesse-Matrix $\mathcal{H}$ (vgl. Kapitel 4.4.2) in drei Dimensionen ist definiert durch

$$\mathcal{H}(u) = \begin{pmatrix} \frac{\partial^2 u}{\partial x_1 \partial x_1} & \frac{\partial^2 u}{\partial x_1 \partial x_2} & \frac{\partial^2 u}{\partial x_1 \partial x_3} \\ \frac{\partial^2 u}{\partial x_2 \partial x_1} & \frac{\partial^2 u}{\partial x_2 \partial x_2} & \frac{\partial^2 u}{\partial x_2 \partial x_3} \\ \frac{\partial^2 u}{\partial x_3 \partial x_1} & \frac{\partial^2 u}{\partial x_3 \partial x_2} & \frac{\partial^2 u}{\partial x_3 \partial x_3} \end{pmatrix} \tag{6.4}$$

und besitzt die drei reellen Eigenwerte λ_1, λ_2 und λ_3, die mit Hilfe der Cardanischen Formeln [14] als Nullstellen des charakteristischen Polynoms von $\mathcal{H}$ direkt berechnet werden können und dem Betrag nach sortiert werden:

$$|\lambda_1| \leq |\lambda_2| \leq |\lambda_3| \tag{6.5}$$

Ähnlich wie die Eigenvektoren des diskreten Trägheitstensors (5.7) charakterisieren die Eigenvektoren der Hesse-Matrix die Richtung von lokalen Strukturen innerhalb des Bildes. Im Gegensatz zur trägheitsbasierten anisotropen Diffusion (Kapitel 5) verzichtet die Methode der Multiskalen-Scharfzeichnung jedoch auf die Verwendung dieser Richtungsinformation und verwendet zur Strukturerkennung lediglich die lokale Krümmung, d.h. die Eigenwerte der Hesse-Matrix. Das Schema des Scharfzeichners ist in der Lage, sowohl dunkle Strukturen auf hellem Hintergrund, als auch helle Strukturen auf dunklem Hintergrund zu erkennen und entsprechend scharfzuzeichnen [37]. Die Detektion der unterschiedlichen Strukturen ist in Tabelle 6.1 aufgeführt. Eine ideale lineare Struktur wird bei der Eigenwertanalyse der Hesse-Matrix durch

$$|\lambda_1| \approx 0 \tag{6.6}$$

$$|\lambda_1| \ll |\lambda_2| \tag{6.7}$$

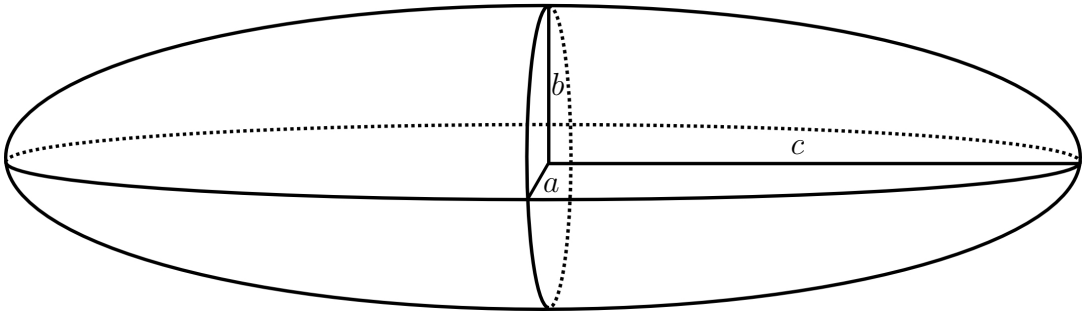


Abbildung 6.2: *Der aus den Eigenwerten der Hesse-Matrix gewonnene Ellipsoid mit den Halbachsen* $a = |\lambda_1| \leq b = |\lambda_2| \leq c = |\lambda_3|$.

$$\lambda_2 \approx \lambda_3 \tag{6.8}$$

charakterisiert. Diese sollen systematisch detektiert werden. Bei vorliegenden linearen und punktförmigen Strukturen ist der Ausdruck

$$R_a = \frac{|\lambda_2|}{|\lambda_3|} \tag{6.9}$$

groß, ansonsten ist er klein. Der Ausdruck

$$R_b = \frac{|\lambda_1|}{\sqrt{|\lambda_2 \lambda_3|}} \tag{6.10}$$

ist hingegen nur für punktförmige Strukturen groß und sonst klein. Folglich liegt eine lineare Struktur genau dann vor, wenn R_a groß und R_b klein ist. Die beiden Größen R_a und R_b lassen sich geometrisch interpretieren: Betrachtet man den Ellipsoiden E (Abbildung 6.2), der von den Eigenvektoren der Hesse-Matrix aufgespannt wird und deren Halbachsen genau so lang sind wie Beträge der zugehörigen Eigenwerte groß, so sind R_a und R_b geometrisch definiert durch

$$R_a = \frac{A/\pi}{c^2} \tag{6.11}$$

und

$$R_b = \frac{V/(4\pi/3)}{(A/\pi)^{\frac{3}{2}}}. \tag{6.12}$$

Hierbei bezeichnet $V = \frac{4}{3}\pi abc$ das Volumen des Ellipsoiden mit den Halbachsen $a \leq b \leq c$ und $A = \pi bc$ den Flächeninhalt, der von den beiden längsten Halbachsen aufgespannten Ellipse. Die beiden geometrischen Quotienten R_a und R_b sind unabhängig von einer Skalierung der Grauwerte des Bildes [37], um sicherzustellen, dass sie nur ein Maß für die geometrische Information des Bildes sind. Um das Hintergrundrauschen deutlich von den im Bild vorhandenen Strukturen abzugrenzen, ist allerdings die Betrachtung des Betrages

der zweiten Ableitung, beispielsweise in Form der *Frobeniusnorm* der Hesse-Matrix

$$S = \|\mathcal{H}\|_F = \sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2} \tag{6.13}$$

notwendig. Die Frobeniusnorm der Hesse-Matrix ist in Hintergrundregionen aufgrund von fehlendem Kontrast in den Bilddaten klein, in Regionen mit Strukturinformation ist sie jedoch groß. Legt man die Größen R_a, R_b und S zur Detektion der hellen linearen Strukturen entsprechenden Dendritenäste zugrunde, so liefert die Funktion

$$\nu_\sigma(x) = \begin{cases} 0, & \text{falls } \lambda_2 > 0 \text{ oder } \lambda_3 > 0 \\ \left(1 - \exp\left(-\frac{R_a^2}{2\alpha^2}\right)\right) \exp\left(-\frac{R_b^2}{2\beta^2}\right) \left(1 - \exp\left(-\frac{S^2}{2\gamma^2}\right)\right), & \text{sonst} \end{cases} \tag{6.14}$$

die Antwort des Multiskalen-Scharfzeichners auf der Skala σ im Bildpunkt x. Mit Hilfe der Parameter α, β und γ wird die Empfindlichkeit des Scharfzeichners auf die entsprechenden Größen R_a, R_b und S gesteuert. Der Einfluss dieser Parameter auf das Verhalten des Scharfzeichners ist in Abschnitt 6.8 diskutiert.

6.4 Verwendung der Multiskalen Information

Die Funktion $\nu_\sigma(x)$ wird unter Variation des Skalenparameters σ für jeden Bildpunkt x mehrfach ausgewertet. Auf der Skala σ, die bestmöglich mit der Größe der zu detektierenden Struktur übereinstimmt, wird $\nu_\sigma(x)$ maximal [37]. Die Antwort $\nu(x)$ des Scharfzeichners im Bildpunkt x ist definiert durch

$$\nu(x) = \max_\sigma \nu_\sigma(x). \tag{6.15}$$

Die Notwendigkeit der Verwendung verschiedener Skalen σ ist ebenfalls in Kapitel 6.8 diskutiert.

6.5 Anpassung des Verfahrens zur Scharfzeichnung von Neuronenzellen

Der Kontrast der Ergebnisbilder ist teilweise sehr schlecht. Daher wird an die Scharfzeichnung eine Histogramm-Normalisierung (Kapitel 4.2.2) angeschlossen. Zudem entspricht der Zellkern der Neuronenzellen nicht den scharfzuzeichnenden linearen Strukturen. Die Ränder des Zellkerns werden zwar scharfgezeichnet, das Innere wird jedoch als isotrope Fläche erkannt und die Filterantwort ist somit klein. Dieses Problem kann mit einer globalen Schwellwertsegmentierung (Kapitel 4.2.2 und 7.1) auf einfache Weise gelöst werden. Hierbei wird ausgenutzt, dass der Zellkern in den Mikroskopaufnahmen sehr hell ist. Die Filterantwort (6.15) wird erweitert durch

$$\bar{\nu}(x) = \max\left\{\max_{\sigma} \nu_\sigma(x),\, s_\vartheta(x)\right\} \tag{6.16}$$

mit

$$s_\vartheta(x) = \begin{cases} 1 \textit{ für } u(x) \geq \vartheta \\ 0 \textit{ sonst} \end{cases} \tag{6.17}$$

bei einem einstellbaren Schwellwertparameter ϑ.

6.6 Implementierung

Der Multiskalen-Scharfzeichner kann in jedem Bildpunkt separat berechnet werden. Abhängigkeiten zwischen verschiedenen Voxeln bestehen nicht. Daher lässt sich das CUDA Programmiermodell optimal für die Implementierung des Scharfzeichners verwenden. Für jedes Voxel wird ein Thread generiert, der unabhängig von allen anderen Threads die Rückgabe des Scharfzeichners in diesem Voxel berechnet. Die auftretenden partiellen Ableitungen werden mit Hilfe von finiten Differenzen [102] approximiert. Eine Laufzeitanalyse der Implementierung wurde an dieser Stelle ausgespart, da der Filter für große Datenmengen sehr schnell berechnet werden kann.

6.7 Anwendung auf Testdatensätze

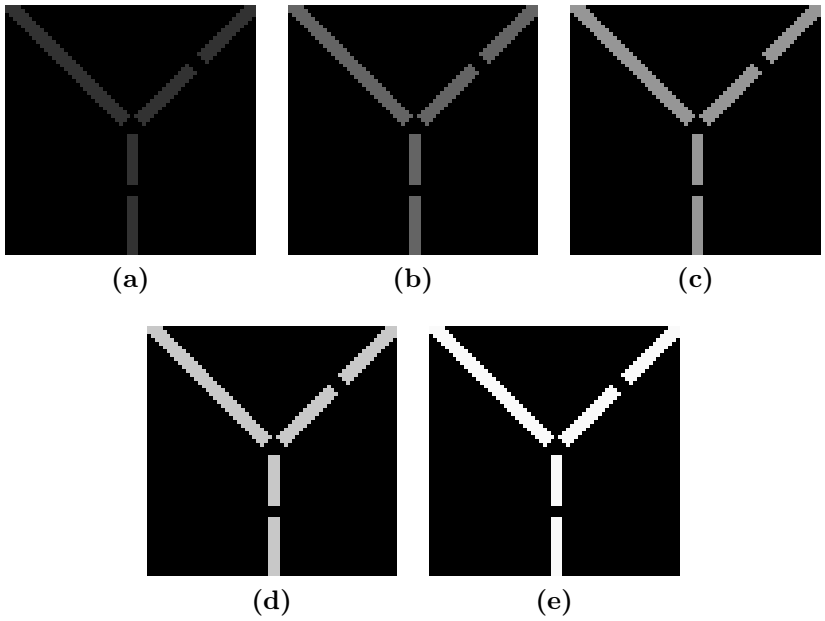


Abbildung 6.3: *Der aus dem vorherigen Kapitel bekannte Testdatensatz mit unterschiedlichem Grauwertkontrast (Schnittebenenbilder). (a) Grauwert von 50. (b) Grauwert von 100. (c) Grauwert von 150. (d) Grauwert von 200. (e) Grauwert von 250.*

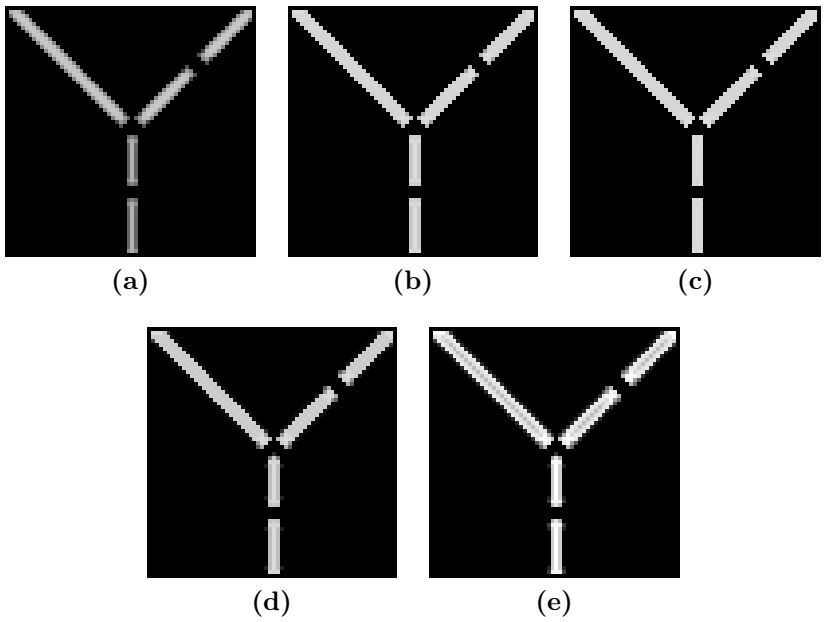
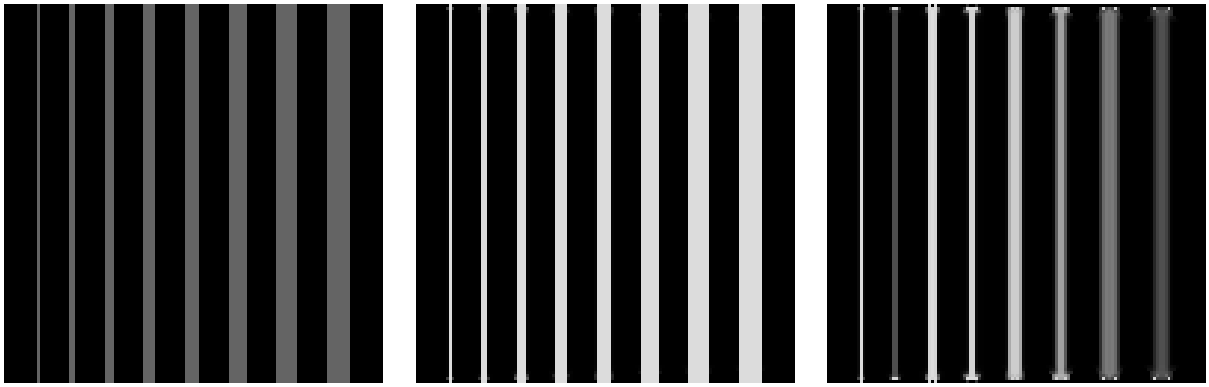


Abbildung 6.4: *Die Testdaten aus Abbildung 6.3 nach Anwendung des Scharfzeichners mit Parametern* $\alpha = 0.2$, $\beta = 2$, $\gamma = 0.05$ *und* $\sigma \in \{0.5, 1, 1.5\}$ *(Schnittebenenbilder).*



(a) **(b)** **(c)**

Abbildung 6.5: *Testdaten zur Validierung, dass der Multiskalen-Scharfzeichner Strukturen unterschiedlicher Größe korrekt behandeln kann (Schnittebenenbilder). (a) Querschnitt durch das Testbild mit Quadern unterschiedlicher Größe. (b) Scharfgezeichnet mit Parametern* $\alpha = 0.5$*,* $\beta = 2$*,* $\gamma = 0.1$ *und* $\sigma \in \{0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4\}$*. (c) Scharfgezeichnet unter Verwendung zu weniger Skalen.*

Bei der Anwendung des Multiskalen-Scharfzeichners auf verschiedene Testdatensätze zeigt sich, dass bei dem Scharfzeichenprozess die Größe vorhandener Strukturen erhalten bleibt und gleichzeitig Strukturen mit unterschiedlichem Grauwertkontrast gleichermaßen scharfgezeichnet werden. Hierzu wird zunächst der aus dem vorherigen Kapitel bekannte Datensatz des Ypsilons in verschiedenen Grauwertstufen betrachtet (Abbildung 6.3). Diese mit den Parametern $\alpha = 0.2$, $\beta = 2$, $\gamma = 0.05$ und $\sigma \in \{0.5, 1, 1.5\}$ scharfgezeichneten Testdatensätze sind Abbildung 6.4 zu entnehmen. Hierbei wurde weder die globale Schwellwertsegmentierung noch die Histogramm-Normalisierung verwendet. Diese Testdaten zeigen, wie der Scharfzeichner Strukturen mit unterschiedlichem Kontrast bei Verwendung der gleichen Eingabeparameter gleichmäßig scharfzeichnet. Ein Schließen von Lücken, wie bei der trägheitsbasierten anisotropen Diffusion beobachtet wurde, kann der Multiskalen-Scharfzeichner allerdings nicht leisten. Zudem sei angemerkt, dass der Scharfzeichner unabhängig von der Orientierung der Strukturen, diese scharfzeichnet und die Größe der Strukturen nicht erhält. Es bleibt zu untersuchen, dass der Scharfzeichner Strukturen unterschiedlicher Größe erkennen und deren Kontrast erhöhen kann. Hierfür wurde ein Testdatensatz, der Quader mit dem Grauwert 100 in unterschiedlichen Größen enthält, mit dem Scharfzeichner bearbeitet (Abbildung 6.5). Bei der Scharfzeichnung mit ausreichend vielen Skalen werden alle Strukturen korrekt scharfgezeichnet. Auch hierbei ist deutlich zu erkennen, dass die Größe der Strukturen unverändert bleibt. Werden jedoch zu wenige Skalen verwendet, wird nur der Kontrast derjenigen Strukturen verbessert, die auf eine der verwendeten Skalen passen (Abbildung 6.5c). Die Notwendigkeit der Verwendung vieler Skalen wird im nächsten Abschnitt und in Abbildung 6.8 nochmals verdeutlicht.

6.8 Eingabeparameter

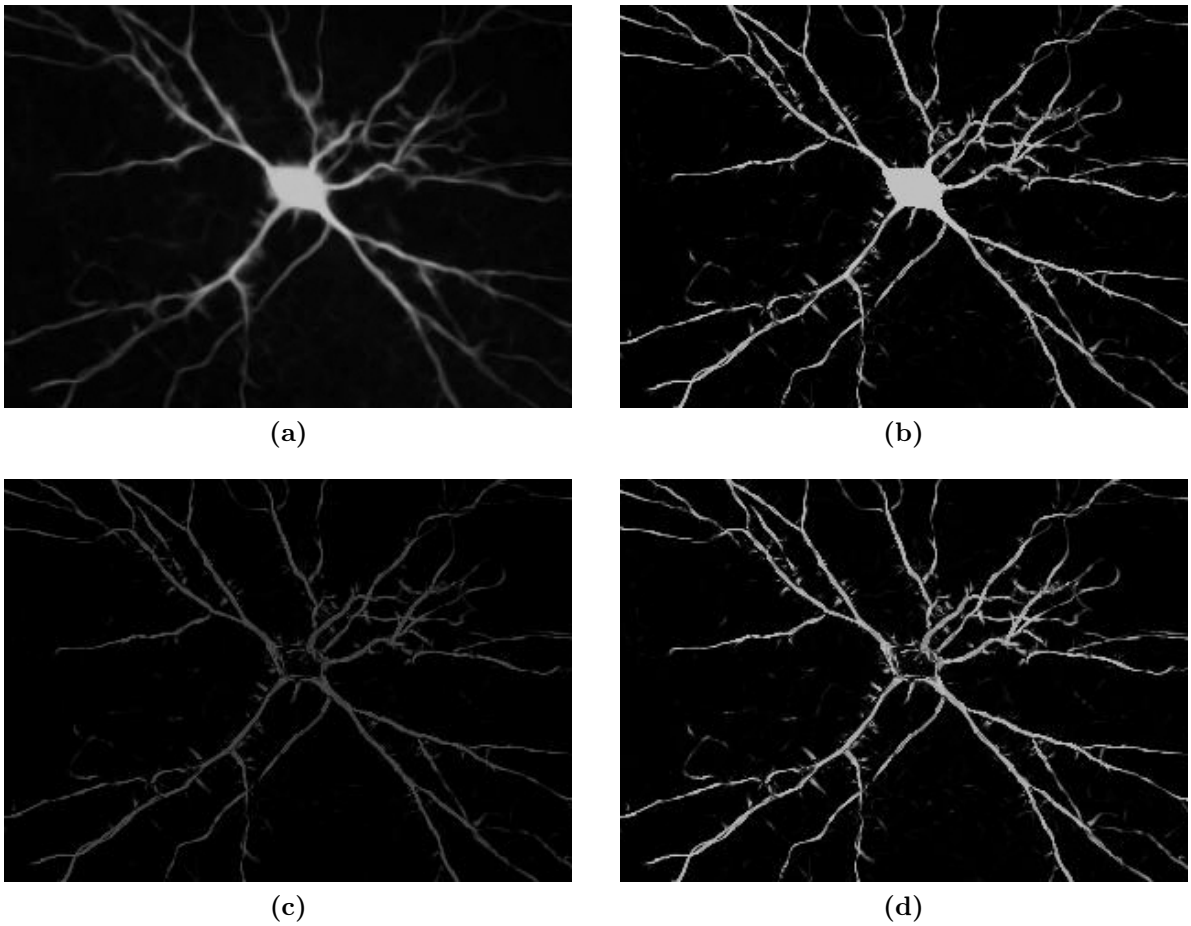


Abbildung 6.6: *(a) Volumenprojektion einer durch trägheitsbasierte Diffusion vorgefilterten dreidimensionalen Mikroskopaufnahme. (b) Nach Anwendung des Multiskalen-Scharfzeichner mit Parametern* $\alpha = 1$, $\beta = 1$, $\gamma = 0.1$ *und dem Skalenraum* $\sigma \in \{0, 0.5, 1, 1.5, 2, 2.5, 3\}$ *mit anschließender Histogramm-Normalisierung und globaler Schwellwertsegmentierung mit* $\vartheta = 0.8$ *zur Detektion des Zellkerns. (c) Multiskalen-Scharfzeichner ohne anschließende Histogramm-Normalisierung und ohne kombinierte Schwellwertsegmentierung. (d) Multiskalen-Scharfzeichner mit anschließender Histogramm-Normalisierung, aber ohne kombinierte Schwellwertsegmentierung.*

Im Folgenden sei der Multiskalen-Scharfzeichner auf eine durch trägheitsbasierte Diffusion vorgefilterte dreidimensionale Mikroskopaufnahme einer Neuronenzelle angewandt (Abbildung 6.6a). Diese und die weiteren Abbildungen in diesem Abschnitt sind jeweils Volumenprojektionen dieser Aufnahme. Es ist eine deutliche Scharfzeichnung der Dendritenäste und des Zellkerns zu erkennen (Abbildung 6.6b). Dieses gute Ergebnis lässt sich nur durch die Kombination des Scharfzeichners mit einer globalen Schwellwertsegmentierung sowie einer anschließenden Histogramm-Normalisierung erreichen. Die Filterantwort ohne

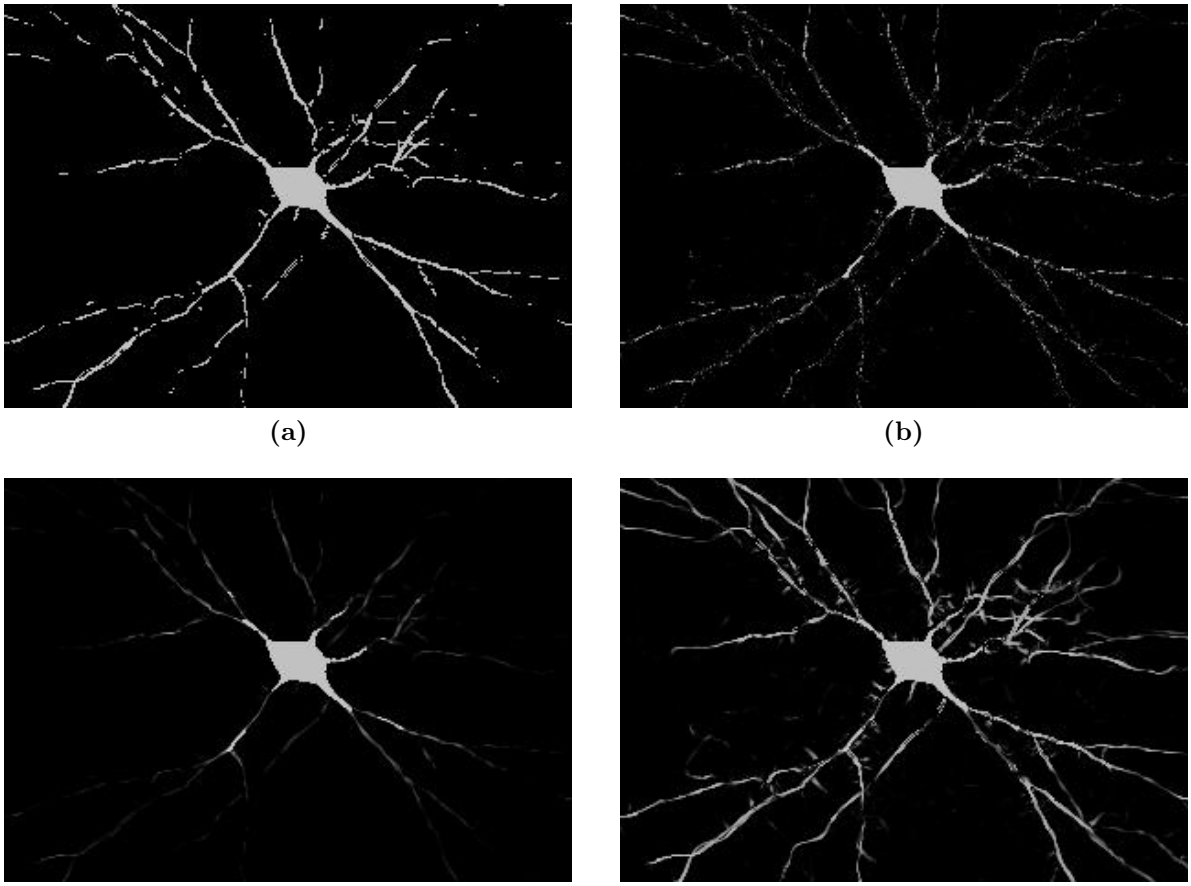
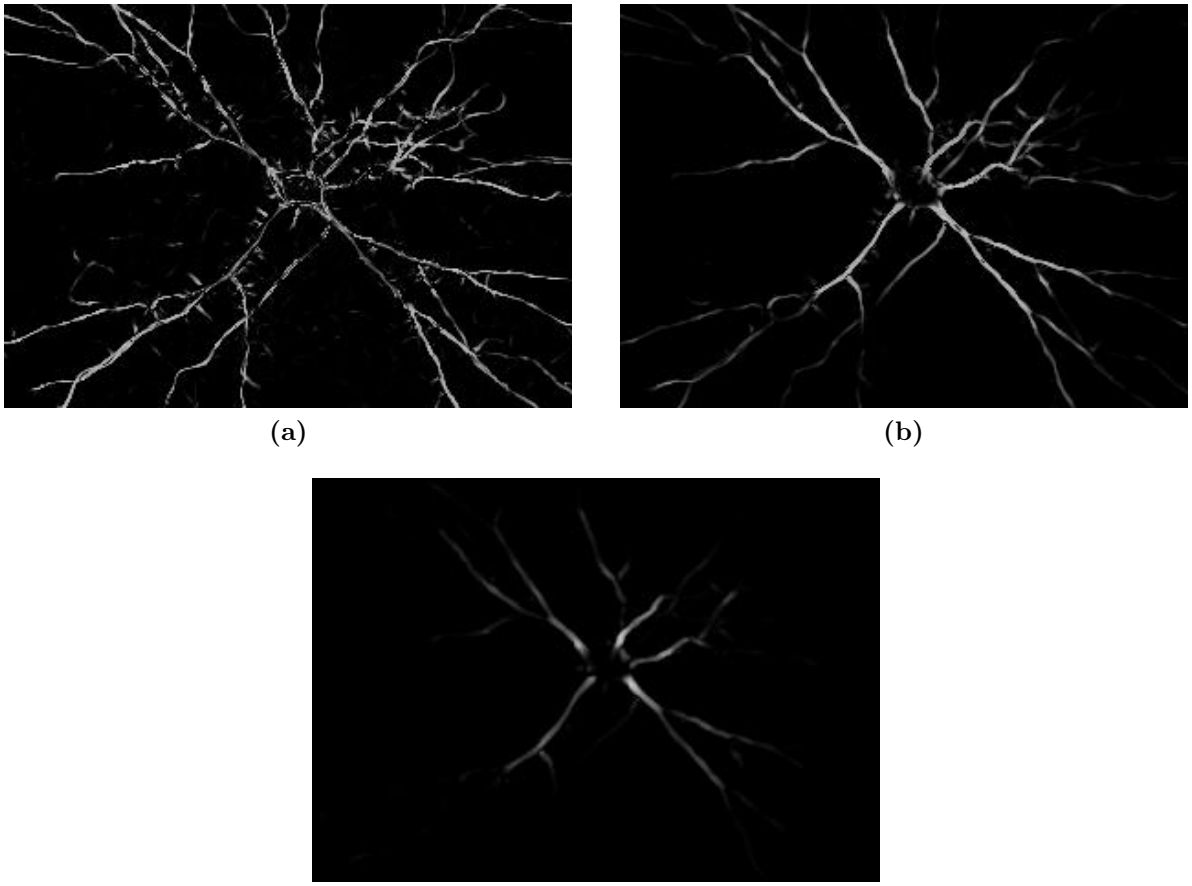


Abbildung 6.7: *Bei verschiedenen Fehleinstellungen der Parameter des Multiskalen-Scharfzeichners enthalten die Dendritenäste anschließend große Lücken oder besitzen einen schlechten Kontrast. (a) α zu groß. (b) β zu klein. (c) γ zu groß. (d) Verwendung von zu wenigen Skalen.*

diese zusätzlichen Anpassungen ist den Abbildungen 6.6c und 6.6d zu entnehmen. Die Parameter α, β und γ müssen so gewählt werden, dass die Exponentialterme aus Formel (6.14) nicht zu schnell abklingen, da sonst Lücken in den Dendritenästen auftreten können (Abbildungen 6.7 a,b,c). Dies ist ebenfalls der Fall, falls zu wenige Skalen verwendet werden (Abbildung 6.7d). Dass die Verwendung von vielen unterschiedlichen Skalen notwendig ist, wird zudem durch Abbildung 6.8 verdeutlicht, welche die Filterantwort der einzelnen Skalen zur Detektion von Strukturen unterschiedlicher Größe enthält. Hierbei ist zu beachten, dass zu einer besseren Darstellung bei den Bildern aus Abbildung 6.8 eine Histogramm-Normalisierung durchgeführt wurde.



(a) **(b)**

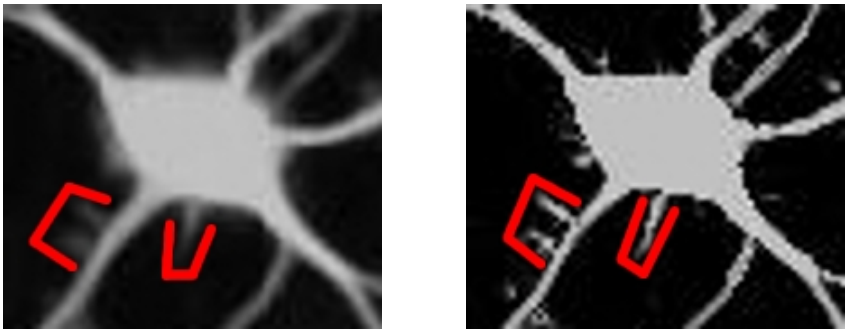
(c)

Abbildung 6.8: *Antwort des Scharfzeichners auf verschiedenen Skalen (Volumenprojektionen). (a) $\sigma = 0$. (b) $\sigma = 1$. (c) $\sigma = 2$. Für eine bessere Darstellung wurde eine Histogramm-Normalisierung auf die eigentliche Antwort des Scharfzeichners angewandt.*

6.9 Ergebnisse

Durch den Multiskalen-Scharfzeichner werden *Dendritische Dornfortsätze*, sogenannte *Spines*, die in hochaufgelösten Mikroskopaufnahmen zu sehen sind, scharfgezeichnet. Die Spines besitzen in den ursprünglichen und in den vorgefilterten Aufnahmen einen schlechten Kontrast und werden daher bei einer Segmentierung (Kapitel 7) ohne vorherige Scharfzeichnung nicht berücksichtigt. Abbildung 6.9 zeigt einen Ausschnitt der in Abbildung 6.6a gezeigten vorgefilterten Neuronenzelle sowie den Ausschnitt der scharfgezeichneten Zelle. Die scharfgezeichneten rot markierten Spines sind deutlich zu erkennen.

Als nächstes sei die durch trägheitsbasierte Diffusion vorgefilterte Aufnahme einer Neuronenzelle einer lebenden Maus scharfgezeichnet (Abbildungen 5.18 und 6.10). Wie in Kapitel 5.10 beschrieben leidet diese Aufnahme unter einem besonders schlechten Signal-zu-Rausch-Verhältnis. Durch geeignete Wahl der kleinsten Skala bei der Multiskalenanalyse



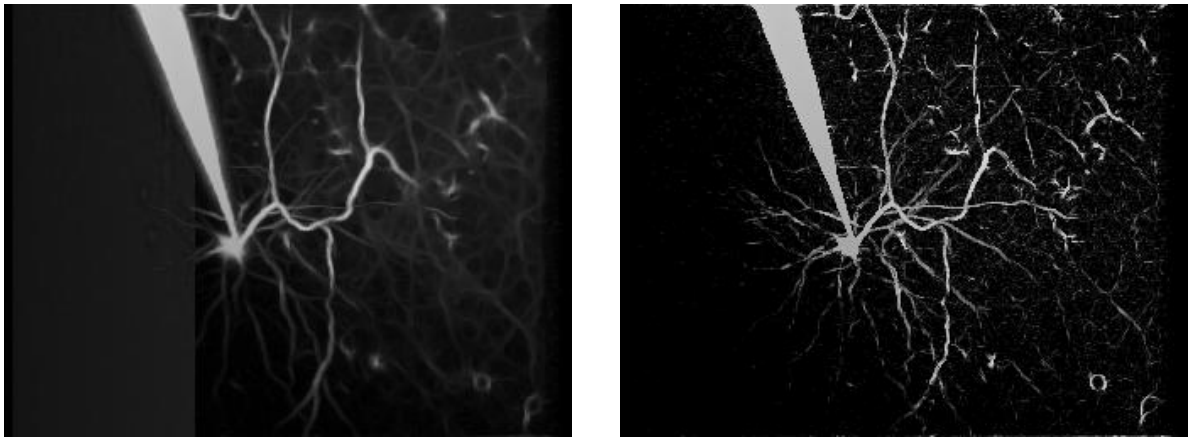
(a) **(b)**

Abbildung 6.9: *Durch den Multiskalen-Scharfzeichner wird der Kontrast der Spines (rot markiert) erhöht, die dann anschließend vernünftig segmentiert und rekonstruiert werden können (vgl. Kapitel 14.4.3). (a) Ausschnitt der vorgefilterten Aufnahme aus Abbildung 6.6a. (b) Nach Anwendung des Multiskalen-Scharfzeichners (Volumenprojektionen).*

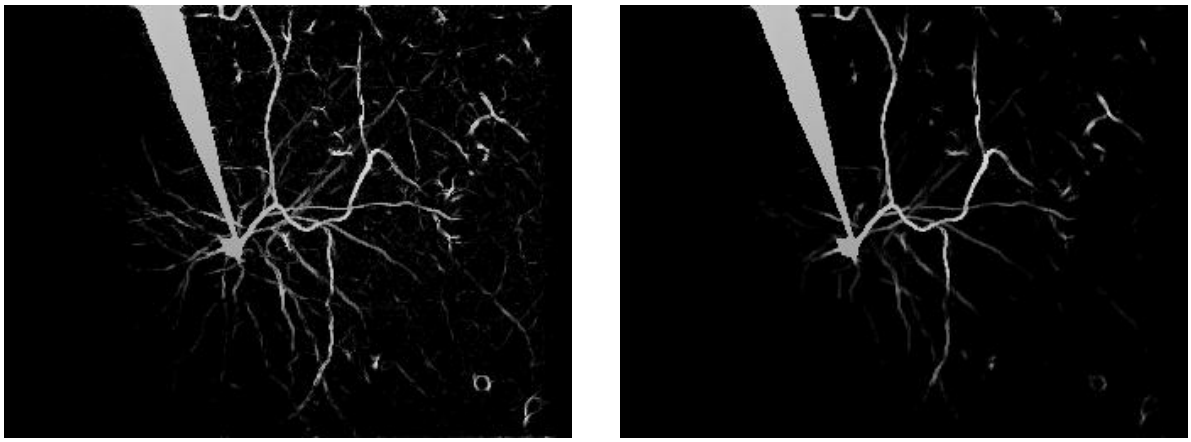
wird der Großteil des Rauschens unterdrückt und nur die relevante Bildinformation scharfgezeichnet (Abbildungen 6.10c und d). Hierbei ist allerdings zu beachten, dass auch sehr filigrane Strukturen, wie die dendritischen Dornfortsätze, nicht mehr scharfgezeichnet, sondern ausgelöscht werden. Dieses Defizit stört an dieser Stelle allerdings nicht, da die betrachtete Lebendaufnahme über eine zu geringe Auflösung verfügt, um überhaupt Spines sehen zu können.

Der Multiskalen-Scharfzeichner kann ohne Vorfilterung direkt auf Mikroskopaufnahmen angewandt werden. Hierbei wird ein Großteil des Rauschens unterdrückt. Allerdings können, verglichen mit dem trägheitsbasierten Diffusionsfilter, Lücken in den Dendritenästen durch den Scharfzeichner nicht geschlossen werden. Ein Vergleich zwischen gefilterter, scharfgezeichneter und sowohl gefilterter als auch scharfgezeichneter Mikroskopaufnahme findet sich in Abbildung 6.11.

Sehr gute Ergebnisse liefert der Scharfzeichner auch bei der Kontrasterhöhung von Dendritensegmenten (Abbildung 6.12). Um das in den Aufnahmen vorhandene feinkörnige Rauschen großteils zu eliminieren wird zunächst ein Medianfilter (Kapitel 4.3.2) angewandt. Anschließend folgt der Scharfzeichner mit Parametern $\alpha = 2$, $\beta = 1$, $\gamma = 0.1$, $\sigma \in \{1, 1.5, 2, 2.5, 3, 3.5, 4\}$. Hierbei gilt für die kleinste Skala $\sigma_{min} > 0$, um das feinkörnige Rauschen komplett zu unterdrücken, wie den scharfgezeichneten Aufnahmen (Abbildungen 6.12b und d) zu entnehmen ist.

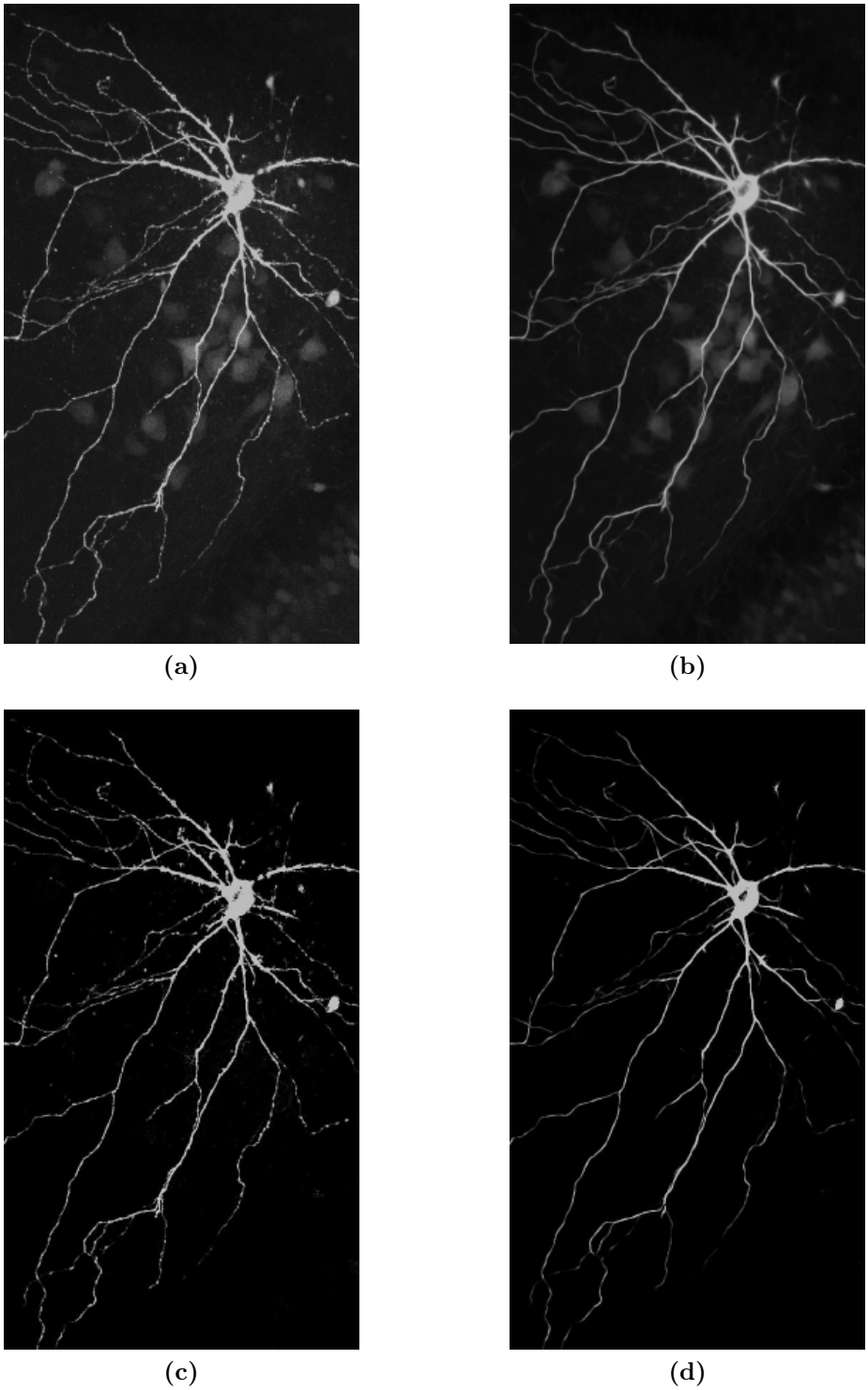


(a) **(b)**



(c) **(d)**

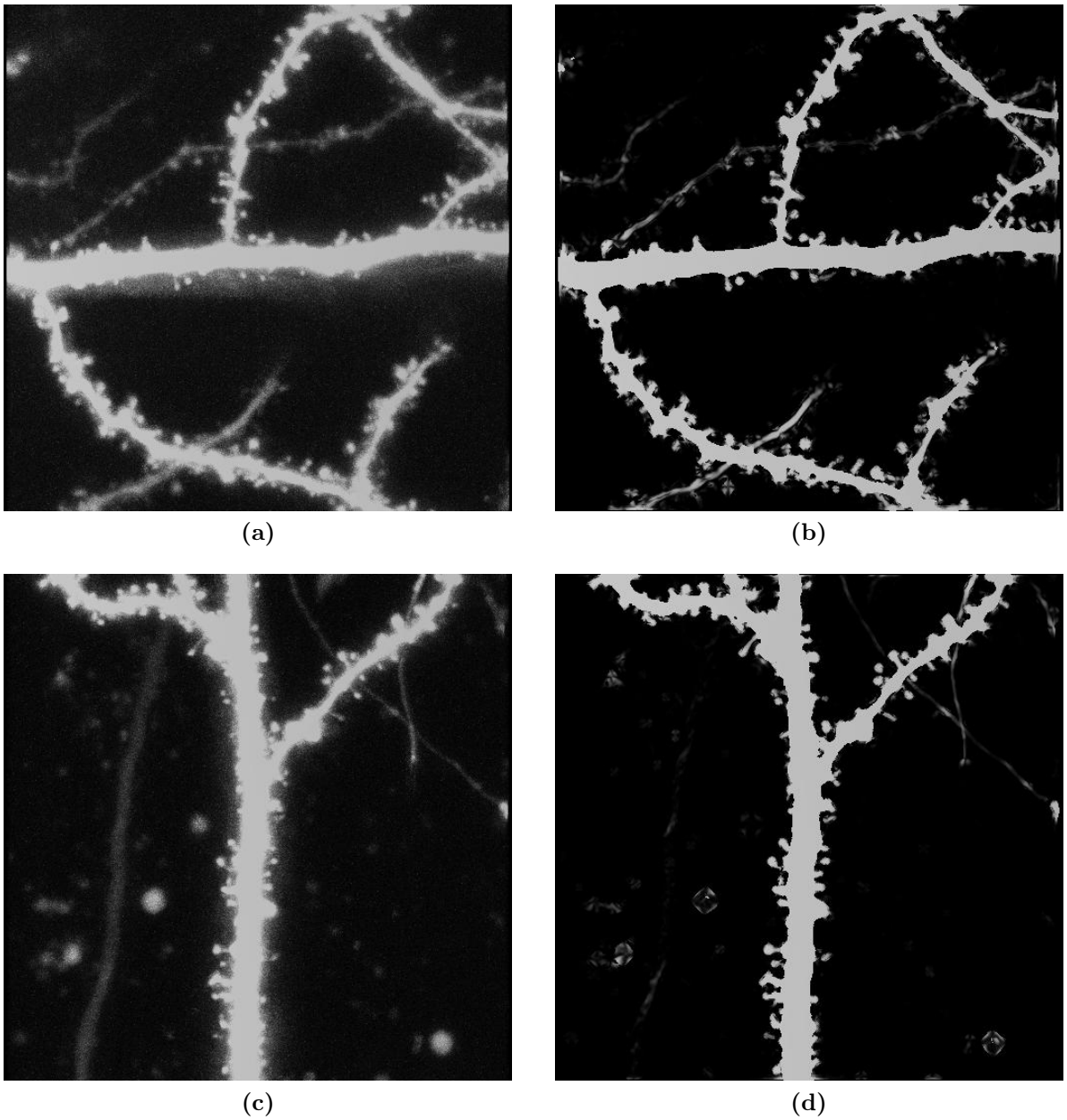
Abbildung 6.10: *(a) Volumenprojektion einer vorgefilterten Aufnahme einer Neuronenzelle einer lebenden Maus. (b), (c), (d) Nach Anwendung des Multiskalen-Scharfzeichners mit Parametern (b)* $\alpha = 3$, $\beta = 1$, $\gamma = 0.1$, $\sigma \in \{0, 0.5, 1, 1.5, 2, 2.5, 3\}$, *(c)* $\alpha = 2$, $\beta = 1$, $\gamma = 0.1$, $\sigma \in \{1, 1.5, 2, 2.5, 3, 3.5, 4\}$ *und (d)* $\alpha = 3$, $\beta = 1$, $\gamma = 0.1$, $\sigma \in \{1, 1.5, 2, 2.5, 3, 3.5, 4\}$.



(a) **(b)**

(c) **(d)**

Abbildung 6.11: *Aufnahme einer Neuronenzelle (Volumenprojektion) und Vergleich von trägheitsbasierter Diffusion mit Parametereinstellung $\rho = 5$, $\tau = 1$, $n_t = 4$, $\eta_1 = 1$, $\eta_2 = \eta_3 = \epsilon$ und Multiskalen-Scharfzeichnung mit Parametern $\alpha = 2$, $\beta = 1$, $\gamma = 0.1$, $\sigma \in \{1, 1.5, 2, 2.5, 3, 3.5, 4\}$. (a) Originalaufnahme. (b) Gefilterte Aufnahme. (c) Scharfgezeichnete Aufnahme. (d) Gefilterte und scharfgezeichnete Aufnahme.*



(a) **(b)** **(c)** **(d)**

Abbildung 6.12: *Multiskalen-Scharfzeichnung von zwei Dendritensegmenten (Volumenprojektion) mit Parametern $\alpha = 2$, $\beta = 0.5$, $\gamma = 0.01$, $\sigma \in \{1, 1.5, 2, 2.5, 3, 3.5, 4\}$. Das feinkörnige Rauschen der Originalaufnahmen (a), (c) wurde hierbei zunächst durch Anwendung eines Medianfilters (Kapitel 4.3.2) verringert.*

Kapitel 7

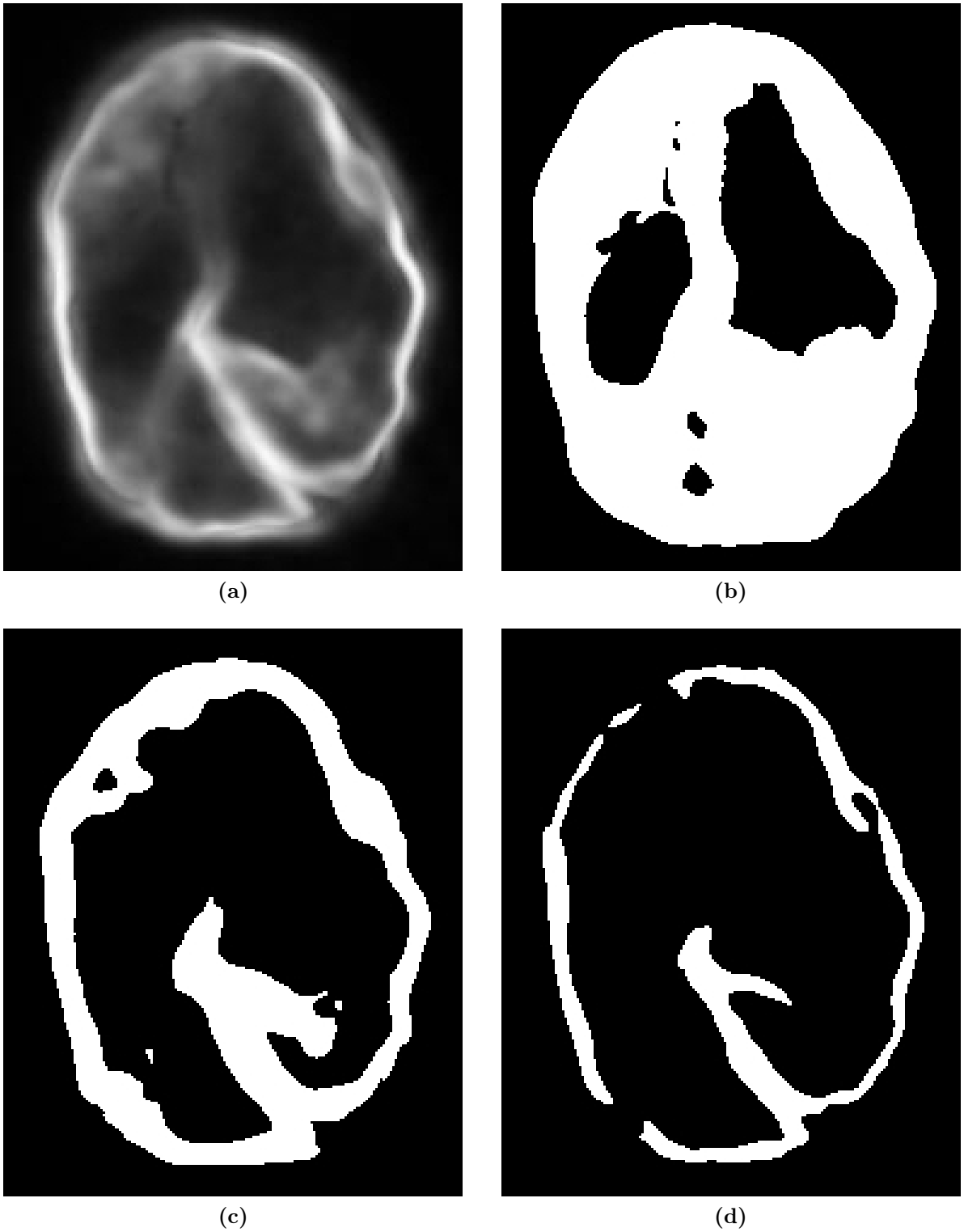
Segmentierung

Nachdem das Rauschen in den Bildern durch trägheitsbasierte Diffusion, Multiskalen-Scharfzeichnung, oder im Falle von Computertomographie-Aufnahmen durch Anwendung eines Median-Filters hinreichend reduziert wurde, wird eine Segmentierung des Bildes durchgeführt. Hierbei wird für jedes Voxel entschieden, ob es zum Objekt oder zum Hintergrund gehört und entsprechend auf weiß (Grauwert 255, bzw. Helligkeit 1.0) oder schwarz (Grauwert 0, bzw. Helligkeit 0.0) gesetzt wird. Nach einer weiteren Anwendung des trägheitsbasierten Diffusionsfilters zur Glättung der Kanten kann anschließend die Oberflächengeometrie durch den *Marching-Cubes-Algorithmus* (Kapitel 8) rekonstruiert werden. Zunächst werden mit der *Schwellwertsegmentierung*, der *Hysterese-Segmentierung* und der *statistischen Segmentierung nach Otsu* einige einfache Segmentierungsalgorithmen vorgestellt, die häufig ausreichend sind, wenn das im Bildsignal vorhandene Rauschen durch entsprechende Vorverarbeitung hinreichend reduziert werden konnte. Bei der Segmentierung von Keramiken, die mittels Computertomographie aufgezeichnet wurden, hat sich eine spezielle *Regionenwachstum-Segmentierung* bewährt, die ebenfalls in diesem Kapitel erläutert wird.

7.1 Globale Schwellwertsegmentierung

Die *globale Schwellwertsegmentierung* ist ein Bildverarbeitungsoperator vom Typ der homogenen Standardtransformationen (Kapitel 4.2.2). Für die globale Schwellwertsegmentierung wird zunächst ein Schwellwert ϑ festgelegt. Alle Voxel, die heller oder gleichhell sind als dieser Schwellwert, werden dem Objekt zugeordnet und auf den Wert *weiß* gesetzt. Alle Voxel die dunkler als dieser Schwellwert sind werden dem Hintergrund zugeordnet und auf den Wert *schwarz* gesetzt.

Definition 7.1 (Globale Schwellwertsegmentierung) *Sei u eine Bildfunktion, die einem Tripel (x, y, z) den Grauwert $u(x, y, z)$ zuordnet. So ist die globale Schwellwertsegmen-*



(a) **(b)**

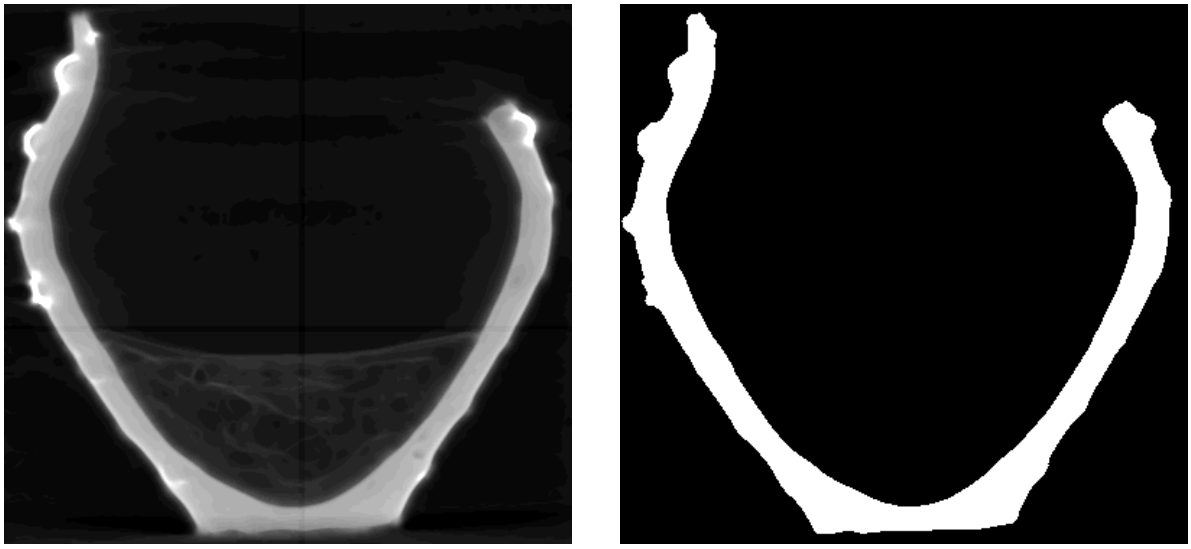
(c) **(d)**

Abbildung 7.1: *(a) Schnittebenendarstellung eines mit trägheitsbasierter Diffusion gefilterten Neuronenzellkerns. (b) Nach globaler Schwellwertsegmentierung mit* $\vartheta = 50$ *(c)* $\vartheta = 100$ *und (d)* $\vartheta = 150$*. Eine Größenänderung des segmentierten Objekts in Abhängigkeit des Schwellwertparameters ist deutlich zu erkennen.*

tierung $s_\vartheta(x, y, z)$ definiert durch

$$s_\vartheta(x, y, z) := \begin{cases} 1 \text{ für } u(x, y, z) \geq \vartheta \\ 0 \text{ sonst.} \end{cases} \tag{7.1}$$

Die Anwendung der globalen Schwellwertsegmentierung ist Abbildung 7.1 zu entnehmen. Hierbei wird die Abhängigkeit des Ergebnisses von der Parametereinstellung deutlich. Die globale Schwellwertsegmentierung ist sehr einfach und kann nur dann erfolgreich eingesetzt werden, wenn sich Bildsignal und Rauschen durch eine scharfe Linie im Grauwerthistogramm voneinander trennen lassen, wie es beispielsweise bei guten Computertomographie-Aufnahmen der Fall ist. Eine erste Verbesserung ist die *Hysterese-Segmentierung*, die ebenfalls zur Klasse der globalen Schwellwertsegmentierer gehört, im Sinne der Bildverarbeitung aber ein Nachbarschaftsoperator ist (Kapitel 4.3). Hierbei werden zwei Schwellwerte ϑ_{min} und ϑ_{max} festgelegt. Ein Voxel wird dem Objekt zugeordnet, wenn entweder sein Grauwert größer oder gleich ϑ_{max} ist, oder sein Grauwert größer oder gleich ϑ_{min} ist und der Grauwert eines Nachbarvoxels größer oder gleich ϑ_{max} ist. Hierbei werden zwei Voxel als Nachbarn bezeichnet, wenn sich ihre Indizierung in genau einer Koordinatenrichtung um den Wert 1 unterscheidet.



(a) **(b)**

Abbildung 7.2: *(a) Schnittebenendarstellung einer vorgefilterten Computertomographie-Aufnahme eines Keramikgefäßes. Zur besseren Darstellung wurde das Histogramm normalisiert. (b) Nach Anwendung von $s_{25,30}$.*

Definition 7.2 (Hysterese-Segmentierung) *Sei u eine Bildfunktion, die einem Tripel $X = (x, y, z)$ den Grauwert $u(x, y, z)$ zuordnet. So ist die Hysterese-Segmentierung*

$s_{\vartheta_{min},\vartheta_{max}}(x,y,z)$ *definiert durch*

$$s_{\vartheta_{min},\vartheta_{max}}(x,y,z) := \begin{cases} 1 \text{ für } u(x,y,z) \geq \vartheta_{max} \\ 1 \text{ für } u(x,y,z) \geq \vartheta_{min} \text{ und } (*) \\ 0 \text{ sonst} \end{cases} \tag{7.2}$$

mit

$$(*)\ \ u(\tilde{X} = (\tilde{x},\tilde{y},\tilde{z})) \geq \vartheta_{max} \text{ mit } \|X - \tilde{X}\|_1 = 1. \tag{7.3}$$

Das Ergebnis der Hysterese-Segmentierung angewandt auf eine Computertomographie-Aufnahme eines Keramikgefäßes ist Abbildung 7.2 zu entnehmen. Die Nachteile beider bisher erwähnten Methoden sind offensichtlich: Parameter, die entscheidend auf die Qualität der Rekonstruktionen Einfluss nehmen, müssen vom Benutzer frei gewählt werden. Eine Änderung dieser Parameter zieht eine unerwünschte Änderung der Größe der Geometrie mit sich (Abbildung 7.1). Zudem kann dieses Verfahren nur schlecht auf Bilder mit unterschiedlichen Kontrastverhältnissen angewandt werden [49]. Da das bei der konfokalen und Zwei-Photon-Mikroskopie verwendete Kontrastmittel mit der Zeit ausbleicht, ist das Signal in den später aufgenommenen Schichten des Bildes schwächer, wodurch ein Kontrastverlust entsteht [90]. Zur Segmentierung der Mikroskopaufnahmen ist es daher nötig eine lokale Segmentierung durchzuführen, d.h. durch Betrachtung kleiner Umgebungen der einzelnen Voxel für jedes Voxel einen separaten Schwellwert zu bestimmen (Kapitel 7.2). Die globale Schwellwertsegmentierung hingegen ist häufig ausreichend um Computertomographie-Aufnahmen zu segmentieren, da diese üblicherweise einen gleichmäßigen Kontrast aufweisen. Die in Kapitel 7.4 beschriebene *Regionenwachstum-Methode* eignet sich jedoch noch besser zur Segmentierung von Bildern mit homogener Grauwertverteilung, da hier die Kanten in Form des Grauwertgradienten in die Segmentierung mit einfließen und die Segmentierung von der Einstellung des Benutzers unabhängig ist.

7.2 Lokale Schwellwertsegmentierung

Bei der lokalen Schwellwertsegmentierung wird für jeden einzelnen Bildpunkt oder für kleine Bildbereiche jeweils automatisch ein Schwellwert bestimmt. Der Bildpunkt bzw. die Bildbereiche werden dann nach der oben beschriebenen Schwellwertmethode segmentiert. Ein einfacher statistischer Ansatz hierfür ist die *Segmentierung nach Otsu* [85].

7.3 Segmentierung nach Otsu

Dieses, auch *Clustering* genannte Verfahren, berechnet in jedem Voxel einen Schwellwert und segmentiert dieses Voxel dann nach der Schwellwertmethode. Die lokalen Schwellwerte für die einzelnen Voxel werden in einer kubusförmigen Umgebung mit Radius ρ um das jeweilige Voxel durch eine statistische Überlegung berechnet: Die Voxel innerhalb einer

solchen Umgebung werden in zwei Klassen, auch *Cluster* genannt, aufgeteilt. Die beiden Klassen entsprechen gerade dem im Bild befindlichen Objekt und dem Hintergrund. Der *Otsu-Schwellwert* maximiert die Varianz zwischen den beiden Klassen und ordnet so die Voxel mit der größtmöglichen Wahrscheinlichkeit der richtigen Klasse zu.

7.3.1 Mathematische Formulierung

Sei $p(g)$ die Auftrittswahrscheinlichkeit des Grauwerts $g \in [0, G]$. Dann berechnet sich die Auftrittswahrscheinlichkeit eines Voxels in den beiden Klassen durch

$$P_0(\vartheta) = \sum_{g=0}^{\vartheta-1} p(g) \tag{7.4}$$

und

$$P_1(\vartheta) = \sum_{g=\vartheta}^{G} p(g) = 1 - P_0(\vartheta). \tag{7.5}$$

Sei $\bar{g}$ das arithmetische Mittel der Grauwerte aller Voxel und $\bar{g_0}$ und $\bar{g_1}$ die arithmetischen Mittel der Grauwerte innerhalb der beiden Klassen, so ergeben sich die Varianzen innerhalb der einzelnen Klassen durch

$$\sigma_0^2(\vartheta) = \sum_{g=0}^{\vartheta-1} (\bar{g_0} - \bar{g})^2 p(g) \tag{7.6}$$

und

$$\sigma_1^2(\vartheta) = \sum_{g=\vartheta}^{G} (\bar{g_1} - \bar{g})^2 p(g). \tag{7.7}$$

Die Gesamtvarianz innerhalb der Klassen ist dann definiert durch

$$\sigma_{in}^2(\vartheta) = P_0(\vartheta)\, \sigma_0^2(\vartheta) + P_1(\vartheta)\, \sigma_1^2(\vartheta), \tag{7.8}$$

während die Varianz zwischen den beiden Klassen durch

$$\sigma_{zw}^2(\vartheta) \;=\; P_0(\vartheta)(\bar{g_0} - \bar{g})^2 \;+\; P_1(\vartheta)(\bar{g_1} - \bar{g})^2 \tag{7.9}$$

definiert ist. Unter Verwendung der Relationen $P_0 + P_1 = 1$ und $\bar{g} = P_0\bar{g_0} + P_1\bar{g_1}$ lässt sich Gleichung (7.9) vereinfachen zu

$$\sigma_{zw}^2(\vartheta) \;=\; P_0(\vartheta)P_1(\vartheta)(\bar{g_0} - \bar{g_1})^2. \tag{7.10}$$

Insgesamt gilt es den Quotienten

$$q(\vartheta) = \frac{\sigma_{zw}^2}{\sigma_{in}^2} = \frac{P_0(\vartheta)P_1(\vartheta)(\bar{g_0} - \bar{g_1})^2}{P_0(\vartheta)\,\sigma_0^2(\vartheta) + P_1(\vartheta)\,\sigma_1^2(\vartheta)} \tag{7.11}$$

zu maximieren, wobei alle möglichen Werte von ϑ getestet werden müssen. Der Algorithmus zur Bestimmung des lokalen *Otsu*-Schwellwerts im Voxel v lässt sich dann wie folgt beschreiben:

1. Berechne die Wahrscheinlichkeiten $p(g)$ für jeden Grauwert g in der Umgebung des Voxels v.

2. Berechne für alle möglichen Schwellwerte $\vartheta \in [\vartheta_{min}, \vartheta_{max}]$ die Variablen P_0, P_1, $\bar{g_0}$ und $\bar{g_1}$, und daraus die Werte für σ_{in}^2, σ_{zw}^2 und q.

3. Wähle als lokalen *Otsu*-Schwellwert denjenigen, für den q maximal wird.

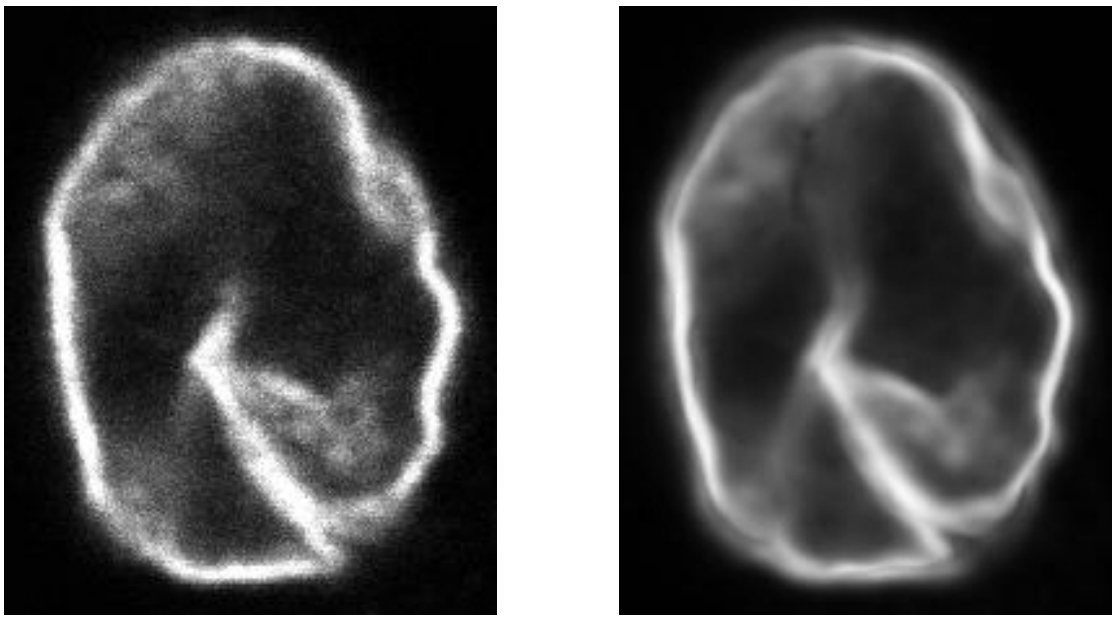
Üblicherweise wird als Eingabeparameter des Verfahrens ein minimaler und ein maximaler Schwellwert $[0 < \vartheta_{min} < \vartheta_{max} \leq G]$ vom Benutzer vorgegeben. Zudem ist der Radius ρ der jeweils betrachteten Umgebungen als Eingabeparameter festzulegen.

7.3.2 Eingabeparameter

Das Resultat der Segmentierung mit der lokalen Schwellwertmethode nach *Otsu* hängt von minimalem und maximalem Schwellwert ϑ_{min} und ϑ_{max} sowie von der betrachteten Größe ρ der Umgebung ab. Zur Untersuchung der Abhängigkeit der Parameter des *Otsu*-Verfahrens sei der Zellkern einer Neuronenzelle betrachtet (Abbildung 7.3), der [90] entnommen ist. Ein typischer Fehler, der bei der Wahl der Parameter gemacht werden kann, ist die falsche Wahl von minimalem und maximalem Schwellwert: Bei einer zu kleinen Wahl von ϑ_{min} werden Teile des Hintergrundrauschens fälschlicherweise dem Objekt zugeordnet (Abbildung 7.4a) und bei einer zu kleinen Wahl von ϑ_{max} wird das Objekt nicht korrekt segmentiert, sondern durch die Segmentierung vergrößert (Abbildung 7.4b). Die Umgebungsgröße ρ sollte weder zu klein noch zu groß gewählt werden, da bei einer zu kleinen Wahl das Rauschen nicht hinreichend vom Objekt getrennt werden kann (Abbildung 7.5a) und bei einer zu großen Wahl filigrane Strukturen nicht adäquat segmentiert werden können (Abbildung 7.5c). Im Gegensatz zur globalen Schwellwertsegmentierung ist die Segmentierung nach *Otsu* jedoch deutlich robuster gegenüber Änderungen in den Eingabeparametern.

7.3.3 Implementierung in CUDA

Die lokale *Otsu*-Segmentierung ist ein weiteres Beispiel für einen sehr effizient in CUDA zu implementierenden Algorithmus, da einerseits die Segmentierung der einzelnen Voxel völlig unabhängig voneinander geschieht und der Algorithmus andererseits sehr wenige Speicherzugriffe benötigt, aber sehr rechenintensiv ist. Die Implementierung in CUDA besteht aus drei Schritten:

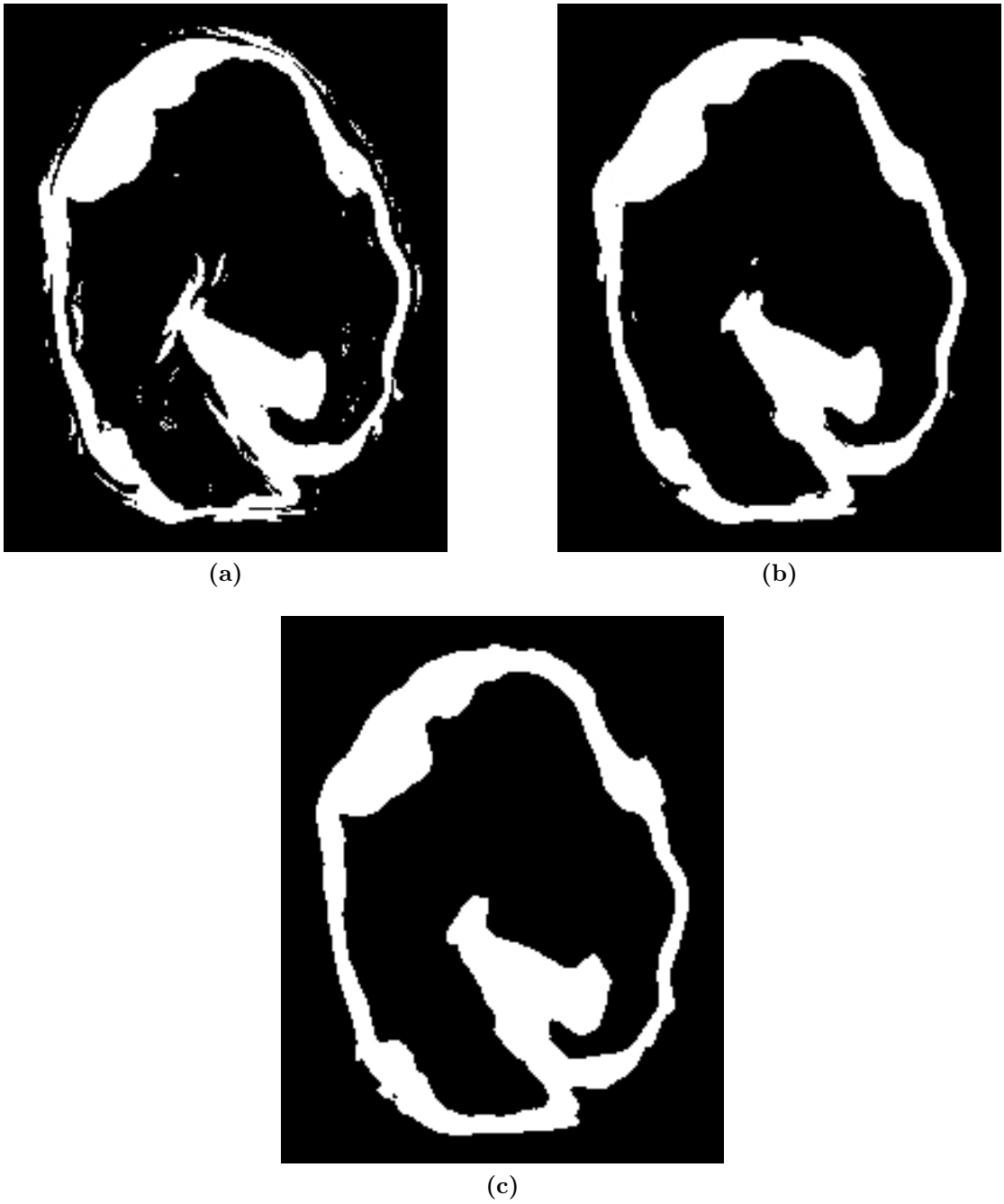


(a) **(b)**

Abbildung 7.3: *(a) Mikroskopaufnahme eines Neuronenzellkerns (Schnittebenendarstellung, Quelle:* [90]*). (b) Mit trägheitsbasierter Diffusion gefilterte Mikroskopaufnahme.*

(a) **(b)**

Abbildung 7.4: *Mit dem* Otsu-*Verfahren segmentierte und zuvor gefilterte Mikroskopaufnahme aus Abbildung 7.3. (a) Wenn ϑ_{min} zu klein gewählt wird, wird zuviel Rauschen als Objekt erkannt. (b) Wählt man ϑ_{max} zu klein, werden Teile des Hintergrundes der Zelle zugeordnet, wodurch diese größer wird.*



(a) **(b)**

(c)

Abbildung 7.5: *Mit dem* Otsu-*Verfahren segmentierte und zuvor gefilterte Mikroskopaufnahme aus Abbildung 7.3 mit Parametern* $\vartheta_{min} = 64$, $\vartheta_{max} = 180$ *und (a)* $\rho = 1$, *(b)* $\rho = 3$ *und (c)* $\rho = 5$.

1. Das zu segmentierende Bild wird in den Device-Speicher kopiert und dort als Textur gebunden, damit ein gecachter Zugriff auf die Daten möglich ist. Zudem wird Speicher für das Zielbild auf der Grafikkarte alloziert.

2. Für jeden einzelnen Bildpunkt wird in einem separaten Thread der lokale *Otsu*-Schwellwert berechnet und das Ergebnis der Segmentierung in dem Zielbild gespeichert.

3. Das Zielbild wird in den Hauptspeicher kopiert.

Aufgrund der völligen Unabhängigkeit der Operationen auf den einzelnen Bildpunkten ist eine Parallelisierung des *Otsu*-Verfahrens unter Verwendung mehrerer Grafikkarten trivial.

7.3.4 Laufzeitanalyse

Größe der Daten	Laufzeit GPU	Laufzeit CPU	Beschleunigung
11 MByte	$1.4s$	$40s$	28.6
14.5 MByte	$1.7s$	$52s$	30.6
71 MByte	$4.9s$	$253s$	51.6
250 MByte	$16.4s$	$849.5s$	51.8
1440 MByte	$101.4s$	$5272s$	52

Tabelle 7.1: *Laufzeit des* Otsu *Verfahrens auf GPU (Tesla-C1060 Prozessor) und CPU (AMD Phenom X9950 mit 2.6GHz) bei Anwendung auf Daten unterschiedlicher Größe mit typischen Parametern* $\vartheta_{min} = 10$, $\vartheta_{max} = 240$ *und* $\rho = 2$.

Wie oben beschrieben eignen sich Grafikkarten hervorragend für die Berechnung der *Otsu*-Segmentierung. Beschleunigungsfaktoren von über 50 (Tabelle 7.1) bestätigen diese Behauptung.

7.4 Regionenwachstum-Segmentierung

Eine weitere Möglichkeit zur Segmentierung der vorverarbeiteten Volumenbilder ist die *Regionenwachstum-Segmentierung*. Hierbei werden ausgehend von einem Saatpunkt, der mit absoluter Sicherheit zum Objekt gehört, nach und nach alle weiteren Nachbarvoxel entweder dem Objekt oder dem Hintergrund zugeordnet, bis keine weiteren Objektvoxel mehr gefunden werden. Viele Regionenwachstum-Methoden setzen Saatpunkte simultan in Objekt- und Hintergrundregionen und lassen die Regionen so lange wachsen, bis das komplette Bild segmentiert wurde. Dieses Vorgehen ist bei den in dieser Arbeit betrachteten Aufnahmen allerdings wenig effizient, da der Hintergrund einen deutlich größeren Teil des Bildes ausmacht, als die sich darin befindlichen Objekte. Das hier vorgestellte und in *NeuRA2* implementierte Verfahren ist an [97] angelehnt. Ausgehend von einem

Saatpunkt, der vom Benutzer frei gewählt werden kann, werden benachbarte Voxel, die einen kleinen Grauwertgradienten aufweisen, unter weiterer Berücksichtigung eines einfachen lokalen Schwellwerts ϑ, dem Objekt zugeordnet. Voxel, die einen großen Grauwertgradienten aufweisen, werden hingegen dem Objekt genau dann zugewiesen, wenn sich ihr Grauwert nicht weiter als eine Standardabweichung von dem lokalen Grauwertmittel der bereits zum Objekt gehörigen Voxel unterscheidet. Es ist offensichtlich, dass dieser Ansatz anfällig gegenüber Kontrastunterschieden innerhalb eines Bildes ist, weshalb sich diese Segmentierungsmethode für konfokale Aufnahmen schlecht eignet. Allerdings lässt sie sich hervorragend für die Segmentierung von Computertomographie-Aufnahmen, insbesondere für die Segmentierung der in dieser Arbeit betrachteten Keramikgefäße, verwenden.

7.4.1 Prinzipieller Ablauf des Verfahrens

1. Der Betrag des Grauwertgradienten wird berechnet und das Histogramm des entstehenden Gradientenbildes ausgeglichen und normalisiert.

2. Bestimmung eines Saatpunktes durch den Benutzer. Der Saatpunkt wird dem Objekt zugeordnet.

3. Betrachte alle Nachbarn von Voxeln, die dem Objekt neu zugeordnet werden. Diese werden ebenfalls dem Objekt zugeordnet, falls eine der folgenden Bedingungen erfüllt ist:

   - Falls der Betrag des Grauwertgradienten in diesem Punkt größer als der globale Schwellwert ϑ_∇ ist und der Grauwert in diesem Punkt nicht mehr als eine Standardabweichung vom lokalen Mittelwert, der bereits zum Objekt zugeordneten Voxel, abweicht.
   - Falls der Betrag des Grauwertgradienten in diesem Punkt kleiner als der globale Schwellwert ϑ_∇ und der Grauwert in dem Punkt größer ist als der lokale Schwellwert ϑ.

4. Fahre mit Punkt 3 fort, bis keine weiteren Voxel dem Objekt zugeordnet werden.

Der in Schritt 3 verwendete globale Gradientenschwellwert ϑ_∇ ist hierbei ein Eingabeparameter des Verfahrens. Im ursprünglichen Verfahren [97], wie auch in der Arbeit [67], ist dieser Wert mit $0.95G$ bei einer Grauwertskala von $[0, G]$ fixiert, was sich als vernünftige Wahl zur Detektion der Kanten eignet. Der einzige echte Eingabeparameter des Verfahrens ist dabei der lokale Schwellwert ϑ, der einer ungefähren globalen Segmentierung entspricht, und daher von den zu segmentierenden Daten abhängt. Durch die spezielle Betrachtung der Kanten des Objekts können auf diese Weise allerdings viel exaktere Ergebnisse erzielt werden.

7.4.2 Berechnung des Grauwertgradienten

Die Berechnung des Grauwertgradienten, Histogramm-Ausgleich und Histogramm-Normalisierung sind in Kapitel 4 ausführlich beschrieben.

7.4.3 Setzen des Saatpunktes

Der Benutzer setzt einen Saatpunkt in die Mitte des zu segmentierenden Objekts. Enthält ein Bild mehrere Objekte, kann auf diese Weise komfortabel eines dieser Objekte ausgewählt werden. Sollen mehrere disjunkte Teilbereiche eines Bildes mit Hilfe der Regionenwachstum-Segmentierung gefunden werden, kann die Segmentierung auf den Hintergrundbereich des invertierten Bildes angewandt werden [65]. Zudem sollte der Saatpunkt mittig im Objekt platziert werden, um ein korrektes Wachsen des Gebietes, zunächst im Inneren und später in den Randbereichen des Bildes, zu gewährleisten. Das Ergebnis dieses Segmentierungsverfahrens hängt unter Umständen von der Wahl des Saatpunktes ab. Es haben sich bei den damit segmentierten Keramikgefäßen praktisch keine Unterschiede durch eine unterschiedliche Wahl der Saatpunkte gezeigt, da sich das Verfahren ohnehin nur zur Segmentierung homogener Grauwertbereiche eignet.

7.4.4 Auswahlkriterien

Wie oben beschrieben, werden weitere Nachbarvoxel von Voxeln, die bereits dem Objekt zugeordnet werden, unter der Erfüllung einer von zwei Kriterien ebenfalls dem Objekt zugeordnet. Beide Kriterien stellen sowohl eine Bedingung an den Betrag des Grauwertgradienten als auch eine Bedingung an den Grauwert in dem jeweiligen Voxel. Es ist naheliegend, die beiden Kriterien nach dem Betrag des Grauwertgradienten zu kategorisieren, da diese Bedingungen disjunkt sind.

Kriterium im Bereich großer Grauwertgradienten

Bereiche großer Grauwertgradienten entsprechen scharfen Kanten im zu segmentierenden Bild. Um diese Kanten in der Segmentierung möglichst gut anzunähern wird ein Voxel im Kantenbereich dem Objekt zugeordnet, wenn sich sein Grauwert höchstens eine Standardabweichung vom Mittelwert der bereits segmentierten Region unterscheidet.

Kriterium im Bereich kleiner Grauwertgradienten

Falls ein kleiner Grauwertgradient vorliegt, befindet sich die Suche nach Objektvoxeln innerhalb des Objekts und entfernt vom Rand. Somit ist eine einfache Schwellwertentscheidung an dieser Stelle ausreichend: Das untersuchte Voxel wird dem Objekt zugeordnet, falls er heller ist, als der eingestellte lokale Schwellwert ϑ.

7.4.5 Ergebnisse

Die Ergebnisse der Regionenwachstum-Segmentierung ähneln denen der globalen Schwellwertsegmentierung mit dem Unterschied, dass die Ränder der Objekte sehr exakt segmentiert werden. In dieser Arbeit wurde die Regionenwachstum-Methode hauptsächlich zur Segmentierung der Keramikgefäße verwendet. In Abbildung 7.6 sind einige der damit segmentierten Keramiken zu sehen.

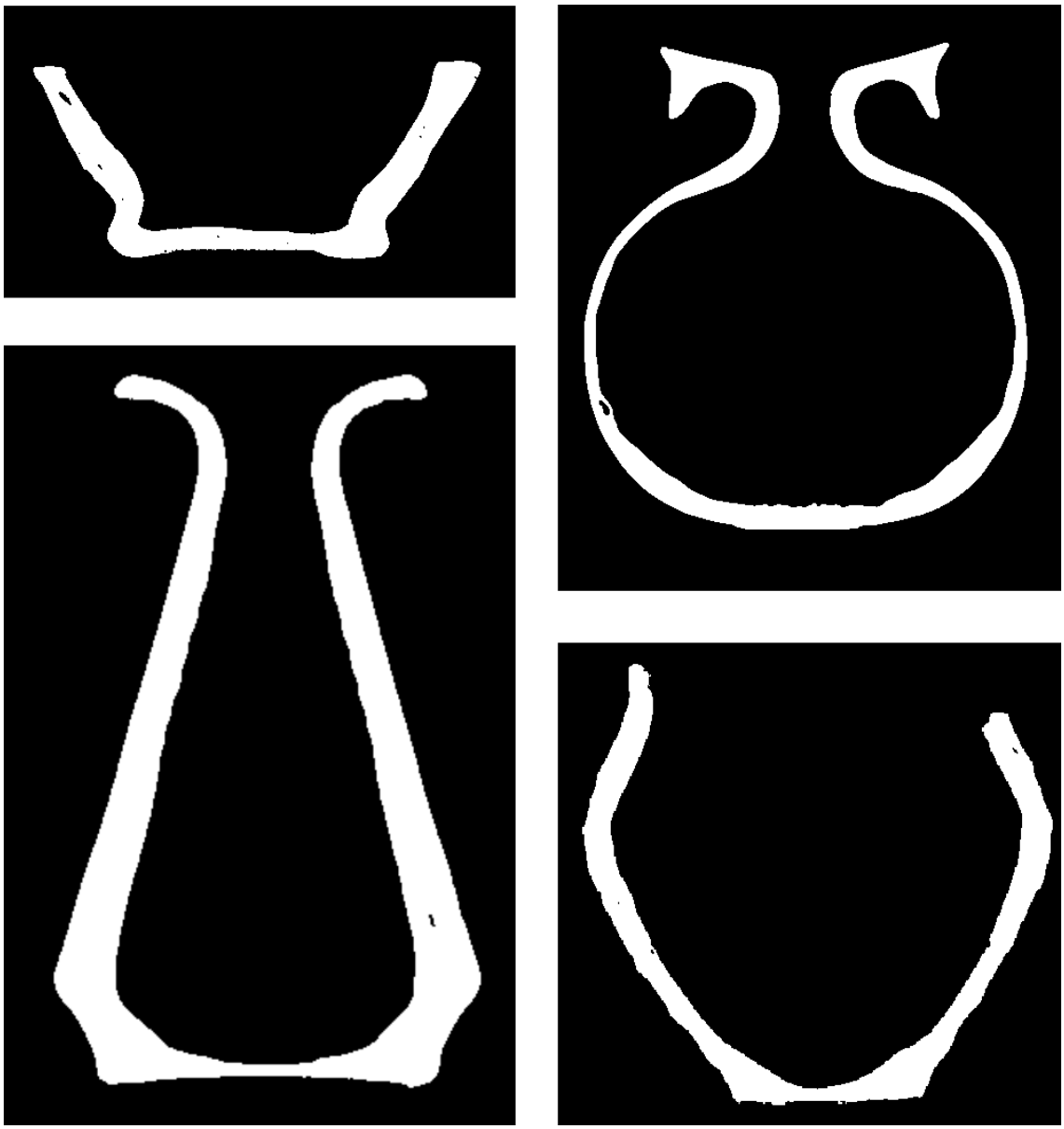


Abbildung 7.6: *Schnittebenendarstellungen unterschiedlicher durch die Regionenwachstum-Methode segmentierter Keramikgefäße. Größere Lufteinschlüsse werden dabei nicht dem Objekt zugeordnet.*

7.5 Modell basierte Segmentierung

Mit Ausnahme der Regionenwachstum-Methode, bei der zusammenhängende Bildbereiche segmentiert werden, verwenden die hier vorgestellten Segmentierungsmethoden keine Information über die zu segmentierenden Objekte. Ihre allgemeine Arbeitsweise ermöglicht zwar einerseits das Verarbeiten von Bildern mit beliebigem Inhalt, hat andererseits aber den Nachteil, dass Bilder, die über ein zu schlechtes Signal-zu-Rausch-Verhältnis verfügen, nicht vernünftig segmentiert werden können. Hier können modellbasierte Segmentierungsverfahren Abhilfe leisten, die Informationen über die zu segmentierenden Objekte für deren Segmentierung verwenden. Der Nachteil solcher Verfahren ist, dass sie sich nur auf sehr spezielle Probleme anwenden lassen. In *NeuRA2* sind solche Methoden zwar nicht vorgesehen, die Software kann aber leicht um weitere problemspezifische Verfahren erweitert werden.

7.6 Wahl der Segmentierung

Die Wahl der richtigen Segmentierungsmethode hängt entscheidend von der Qualität des zu segmentierenden Bildes ab. Während Computertomographie-Aufnahmen, die üblicherweise über einen konstanten Kontrast und scharfe Kanten verfügen, sehr gut mit der oben beschriebenen Regionenwachstum-Methode segmentiert werden können, ist bei konfokalen Mikroskopaufnahmen aufgrund des ausbleichenden Kontrastmittels die statistische Methode nach *Otsu* vorzuziehen.

Kapitel 8

Gittererzeugung

Die Oberflächengeometrie der zu rekonstruierenden Objekte wird in *NeuRA2* durch den *Marching-Cubes-Algorithmus* [64] erzeugt. In der ursprünglichen Version von *NeuRA* [20] wurde zur Gittergenerierung der *Regularisierte Marching-Tetrahedra-Algorithmus* [111] aus zwei Gründen eingesetzt: Einerseits war der Marching-Cubes-Algorithmus bis 2006 durch ein Softwarepatent geschützt, sodass die Verwendung des Marching-Cubes-Algorithmus bis dahin nur kostenpflichtig möglich war, und da andererseits die höhere Genauigkeit des *Marching-Tetrahedra-Algorithmus* für ältere Mikroskopaufnahmen nötig war, um die in diesen Aufnahmen nur wenige Voxel dicken Dendriten adäquat zu rekonstruieren. In *NeuRA2* kommt nun der Marching-Cubes-Algorithmus zur Gittererzeugung zum Einsatz, da das Softwarepatent inzwischen ausgelaufen ist und hochaufgelöste Aufnahmen von Neuronenzellen verfügbar sind. Zudem ist dieses Verfahren leichter zu implementieren und schneller als der Regularisierte Marching-Tetrahedra-Algorithmus.

Da zu Beginn der Gittergenerierung nicht klar ist, aus wie vielen Dreiecken die erzeugte Geometrie bestehen wird, sind Techniken der dynamischen Speicherallokation nötig, die in CUDA nicht ohne weiteres implementiert werden können. Im Folgenden wird daher neben einer Beschreibung des Algorithmus detailliert auf seine parallele Implementierung in CUDA eingegangen. Da der Marching-Cubes-Algorithmus allerdings sehr effizient seriell implementiert werden kann und eine parallele Implementierung einen großen Overhead produziert, zeigt sich, dass die Implementierung in CUDA nur etwa zweimal schneller ist, als eine entsprechend optimierte serielle Implementierung. Dies ist ein Beleg dafür, dass nicht alle Algorithmen durch eine parallele Implementierung massiv beschleunigt werden können. Da die Laufzeit des Marching-Cubes-Algorithmus im Vergleich mit den anderen in *NeuRA2* verwendeten Bildverarbeitungsoperatoren sehr gering ist, lohnen sich weitere Optimierungen an dieser Stelle nicht.

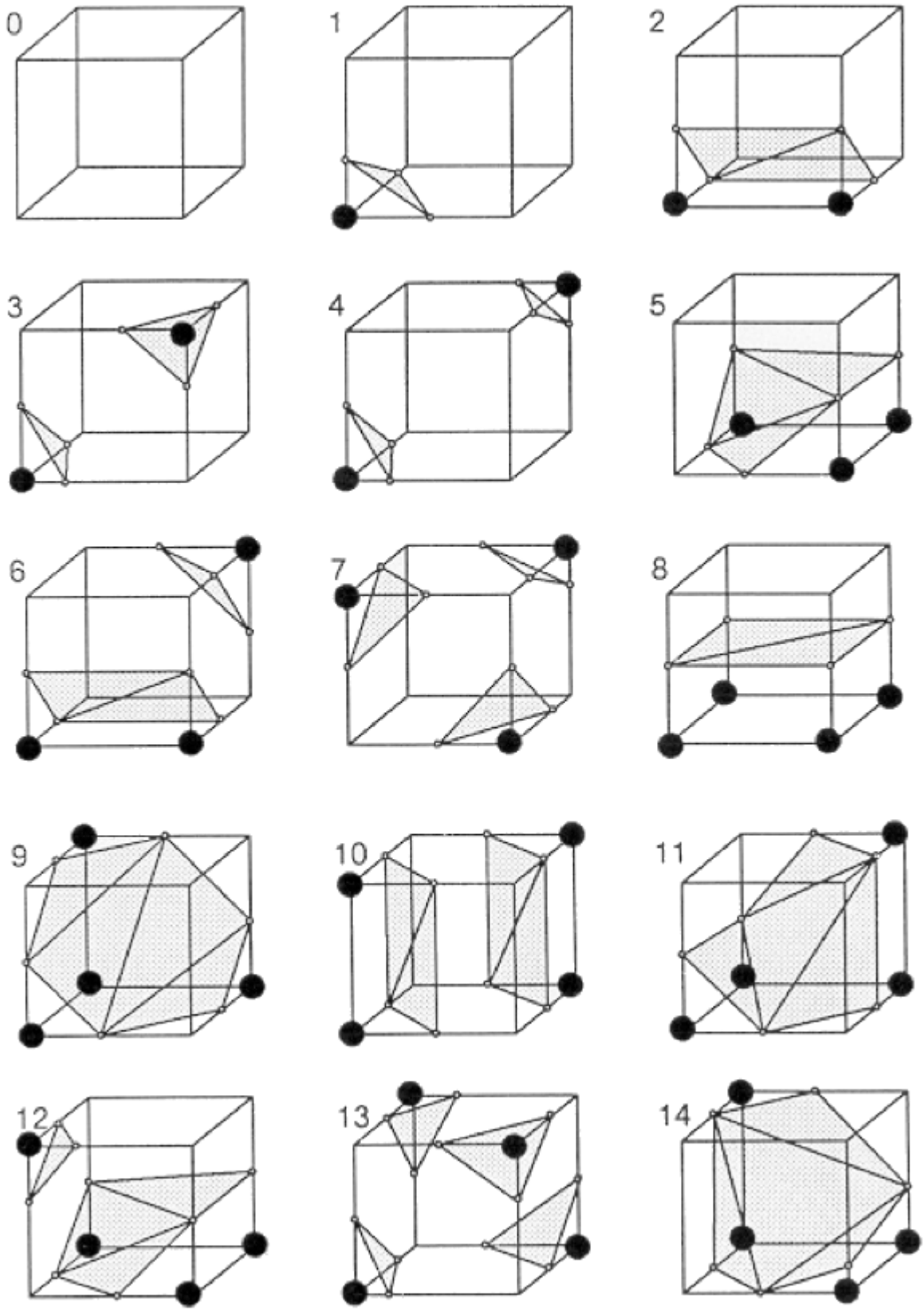


Abbildung 8.1: *Die 15 Grundkonfigurationen des Marching-Cubes-Algorithmus. Quelle:* [64]

8.1 Marching-Cubes-Algorithmus

Der Marching-Cubes-Algorithmus nähert die Oberfläche eines durch ein dreidimensionales Bild beschriebenes Objekt durch ein Dreiecksgitter an. Hierbei wird das ursprüngliche Bild in kleine Würfel zerlegt und anschließend für jeden Würfel bestimmt, wie er das Objekt innerhalb des Bildes schneidet. Diese Schnitte ergeben die Dreiecke des Gitters. Jeweils acht Voxel (zwei in jede Raumrichtung) werden zu einem solchen Würfel zusammengefasst. Als Eingabe des Algorithmus dient der sogenannte *Isoflächen-* oder *Isolevel-Schwellwert* ϑ, mit dessen Hilfe nach dem Prinzip der Schwellwertsegmentierung (Kapitel 7.1) entschieden wird, welche Voxel dem Objekt und welche dem Hintergrund zugeordnet werden.

8.1.1 Klassifikation der Würfel

Jeweils acht Voxel bilden einen Würfel. Abhängig vom Schwellwert ϑ gehört jedes dieser Voxel entweder zum Objekt oder zum Hintergrund. Folglich gibt es 256 verschiedene Möglichkeiten, wie die Voxel angeordnet sein können, die sogenannten *Konfigurationen*. Die Triangulierung dieser 256 Konfigurationen ist sehr aufwändig und fehleranfällig. Betrachtet man jedoch komplementäre Konfigurationen und Konfigurationen, die durch einfache Drehungen auseinander hervorgehen, erhält man 15 Basiskonfigurationen (Abbildung 8.1). Mit Hilfe dieser Basiskonfigurationen wird eine Lookup-Tabelle generiert, welche die relevanten Informationen über die Triangulierung enthält. Jeder der 256 Konfigurationen wird ein Index zugeordnet, durch den der entsprechende Eintrag in der Lookup-Tabelle nachgeschlagen werden kann.

8.1.2 Erzeugung der Geometrie

Auf jeder Kante eines Würfels, die einen Voxel der Klasse Hintergrund mit einem Voxel der Klasse Objekt verbindet, wird ein Vertex erzeugt. Die exakte Position dieses Vertex wird durch eine lineare Interpolation bestimmt, die durch die Grauwerte der beiden beteiligten Voxel gewichtet wird. Für jeden Vertex kann noch die Einheitsnormale durch das Gradientenfeld des ursprünglichen Bildes berechnet werden [64]. Die Normalen können alternativ als Mittelung aller Normalen der zum Vertex angrenzenden Dreiecke bestimmt werden, nachdem das gesamte Dreiecksgitter konstruiert wurde [44]. Werden die einzelnen Würfel in sequenzieller Reihenfolge abgearbeitet, kann die Rechenzeit durch Einsparen redundanter Berechnungen optimiert werden. Bei den nicht am Rand des Bildes liegenden Würfeln müssen jeweils nur Vertices, die auf den Kanten e_5, e_6 und e_{10} liegen (Abbildung 8.2), neu interpoliert werden. Alle anderen Vertices sind bereits durch Nachbarwürfel erzeugt worden.

8.1.3 Ablauf des klassischen Algorithmus

Der klassische Marching-Cubes-Algorithmus lässt sich zusammengefasst in sieben Schritten formulieren. Als Eingabeparameter dient der *Isolevel-Schwellwert* ϑ, der das Bild in die

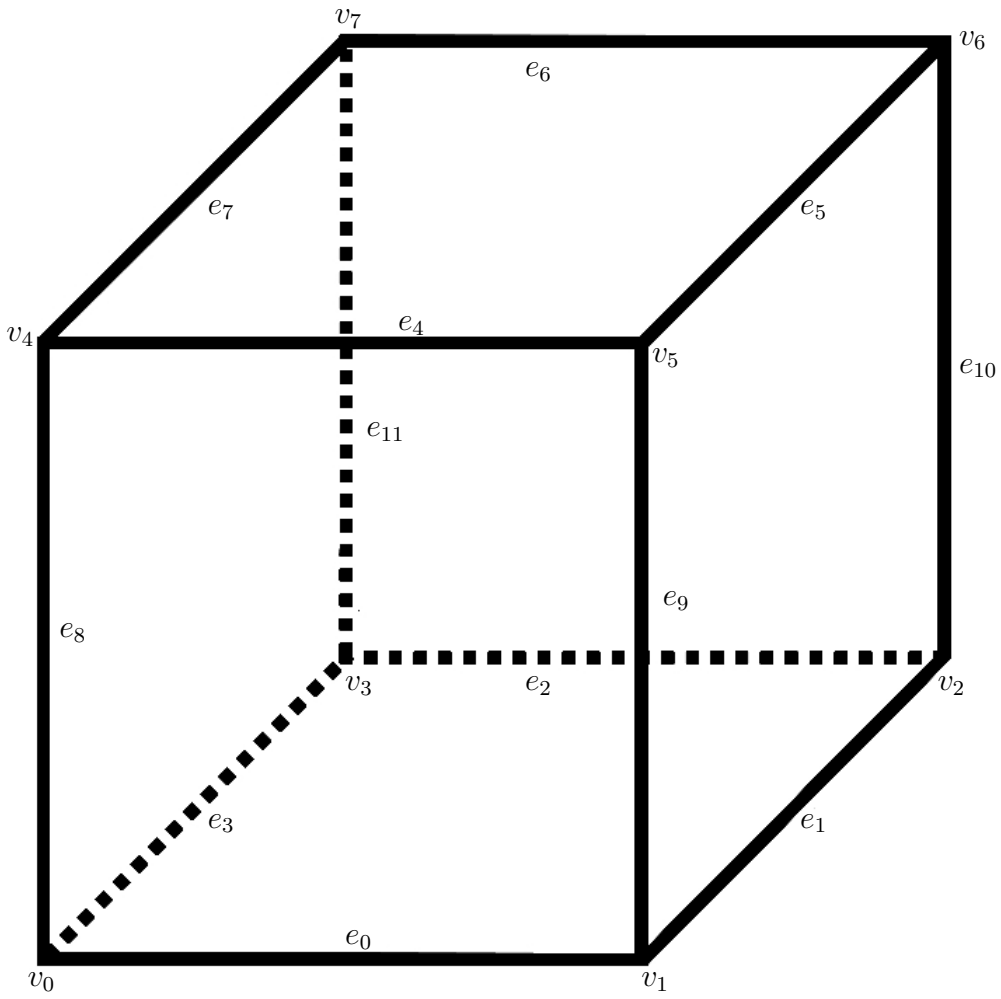


Abbildung 8.2: *Nummerierung der Kanten und Vertices eines Würfels.*

Klassen Objekt und Hintergrund aufteilt (vgl. Kapitel 7.1).

1. *Daten einlesen:*
   Die Bilddaten werden eingelesen.

2. *Würfel bilden:*
   Aus jeweils acht Voxel wird, wie oben beschrieben, ein Würfel gebildet. Die Ecken und die Kanten des Würfels werden durchnummeriert.

3. *Index des Würfels bestimmen:*
   Aus den Grauwerten der acht beteiligten Voxel wird der Index i des Würfels bestimmt. Hierbei wird in der Binärdarstellung von i die j-te Ziffer auf 0 gesetzt, falls die j-te Ecke des Würfels zum Hintergrund gehört und auf 1 gesetzt, falls die j-te Ecke des Würfels zum Objekt gehört. Resultat ist ein ganzzahliger Index $i \in [0, 255]$.

4. *Kanten erzeugen:*
   Die Triangulierung wird in der Lookup-Tabelle nachgeschlagen und anschließend die Kanten aller Dreiecke erzeugt.

5. *Vertexkoordinaten interpolieren:*
   Die Koordinaten der einzelnen Vertices werden linear bezüglich der Grauwerte der anliegenden Ecken des Würfels interpoliert.

6. *Einheitsnormalen berechnen:*
   Die Einheitsnormalen werden durch den Grauwertgradienten an allen Ecken des Würfels berechnet und zwischen den Ecken interpoliert.

7. *Daten ausgeben:*
   Die berechneten Vertices, Normalen und die Triangulierung werden in entsprechende Listen eingefügt. Es wird mit dem nächsten Würfel fortgefahren.

8.2 Serielle Implementierung

Größe der Daten	Vertices	Dreiecke	Laufzeit
11 MByte	120983	241384	$0.8s$
14.5 MByte	210852	396312	$1s$
71 MByte	61038	118352	$5.2s$
506 MByte	219660	436042	$37.4s$

Tabelle 8.1: *Anzahl generierter Dreiecke und Vertices und Laufzeit der seriellen Marching-Cubes-Implementierung auf einem Kern eines AMD Phenom X9950 mit 2.6GHz unter Betrachtung verschiedener Datensätze.*

Bei der seriellen Implementierung wird das dreidimensionale Feld, in dem die Grauwerte der Voxel gespeichert sind, einmal durchlaufen. Hierbei werden die Konfigurationen jedes Würfels berechnet, die entsprechenden Vertices interpoliert und die zugehörigen Dreiecke erzeugt. Die einzelnen Vertices werden über eine *Hashtabelle* [86] verwaltet, damit Indizes bereits berechneter Vertices schnell bestimmt werden können. Welche Vertices von welcher Konfiguration erzeugt werden und wie die dazugehörigen Triangulierungen aussehen, ist in entsprechenden Lookup-Tabellen [18] gespeichert. Die serielle Implementierung des Marching-Cubes-Algorithmus ist sehr schnell (Tabelle 8.1) und die Laufzeit verhält sich linear bezüglich der Datengröße. Das Verfahren besitzt daher optimale Komplexität.

8.3 Implementierung in CUDA

Die parallele Implementierung des Marching-Cubes-Algorithmus in CUDA gestaltet sich ungleich schwerer und birgt einige Schwierigkeiten: Dem CUDA Programmiermodell ent-

sprechend sollte für jeden Würfel ein Thread erzeugt werden, der alle Dreiecke dieses Würfels generiert. Ein offensichtliches Problem ist, dass zunächst nicht bekannt ist, welcher Würfel wie viele Dreiecke erzeugt. Man müsste daher jedem Würfel genügend Speicherplatz zur Verfügung stellen, um maximal fünf Dreiecke zu generieren. Da die Objekte im Vergleich zum Hintergrund meist nur einen kleinen Teil des Bildes ausmachen, ist allerdings zu erwarten, dass man auf diese Weise viel mehr Speicher zur Verfügung stellen muss, als das erzeugte Gitter benötigt. Eine weitere Schwierigkeit an dieser Methode ist, dass viele Vertices redundant erzeugt werden. Angelehnt an die Implementierung des Marching-Cubes-Algorithmus in [76] wird im Folgenden eine Implementierung in CUDA beschrieben, die große Datenmengen verarbeiten kann und deren Ergebnisgeometrie keine doppelten Vertices enthält. Dennoch generiert dieser Ansatz einen großen Overhead, der eine schnelle Laufzeit der parallelen Implementierung verhindert.

8.3.1 Paralleler Marching-Cubes-Algorithmus

1. *Daten einlesen:*
   Die Bilddaten werden eingelesen, in den Device-Speicher kopiert und dort als Textur gebunden.

2. *Klassifizierung der Würfel:*
   Für jeden Würfel wird berechnet, ob er auf dem Schnitt zwischen Objekt und Hintergrund liegt und wenn ja, wie viele Vertices für diesen Würfel generiert werden müssen. Alle Würfel, die einen Beitrag zur generierten Geometrie leisten, werden im Weiteren als *aktive Würfel* bezeichnet.

3. *Anzahl aktiver Würfel:*
   Die Anzahl aktiver Würfel α wird bestimmt.

4. *Indizierung aktiver Würfel:*
   Im Weiteren sollen nur noch Threads für aktive Würfel gestartet werden. Daher wird ein Indexfeld der Größe α erstellt, das die Adressen der aktiven Würfel enthält.

5. *Anzahl generierter Vertices und Berechnung der Startadressen:*
   Die Anzahl generierter Vertices, sowie die Adressen, an denen die einzelnen aktiven Würfel die von ihnen generierten Vertices abspeichern, werden bestimmt.

6. *Erzeugen der Dreiecke:*
   Für alle aktiven Würfel werden die entsprechenden Dreiecke generiert. Für jedes Dreieck werden dabei alle drei Vertices separat erzeugt.

7. *Zusammenlegen redundanter Vertices:*
   Die meisten Vertices werden auf diese Weise mehrfach generiert. Daher wird für jeden Vertex das erste Vorkommen in der Vertexliste bestimmt und diese Adresse gespeichert.

8. *Daten ausschreiben:*
   Die Vertexdaten und das zuletzt berechnete Adressfeld werden auf den Host kopiert. Dieser erzeugt daraus die Triangulierung.

8.3.2 Laufzeitanalyse

Betrachtet man die parallele Implementierung genauer, fallen einige Stellen auf, an denen sich Optimierungen realisieren lassen: Vor allem die redundante Erzeugung der Vertices kann vermieden werden, wobei dann die Triangulierung bereits auf der Grafikkarte generiert werden muss und nicht vom Host erzeugt werden kann. Dadurch sind einige zusätzliche Schreiboperationen auf den globalen Speicher nötig, weshalb sich die Laufzeit wiederum erhöht. Insgesamt ergibt sich so keine Beschleunigung. Erzeugt man die Geometrie des häufig in dieser Arbeit verwendeten Testdatensatzes der Dimension $256 \times 256 \times 216$ mit der parallelen Marching-Cubes-Implementierung, so beträgt die Laufzeit des Algorithmus auf einer GeForce 8600M mit lediglich $N = 4$ Multiprozessoren etwa 1.35 Sekunden. Dabei werden zunächst über eine Million Vertices erzeugt, die anschließend auf etwa 200000 Vertices reduziert werden. Die maximale Speicherauslastung beträgt 233 Megabyte, wobei dieser Datensatz ca. 200000 aktive Würfel erzeugt. Die Verwendung einer Tesla-C1060 Karte verringert die Laufzeit bei diesem Testdatensatz auf 0.48 Sekunden, obwohl hier $N = 30$ Multiprozessoren zum Einsatz kommen und die Taktrate der Prozessoren fast doppelt so hoch ist wie bei der GeForce 8600M. Dies macht deutlich, dass bei dem Marching-Cubes-Algorithmus die Laufzeit nicht durch die Berechnungen, sondern im Wesentlichen durch die Speicherzugriffe bestimmt wird. Daher werden die parallelen Streamingprozessoren der Grafikkarten nicht optimal ausgenutzt. Zudem muss in einer seriellen Implementierung das dreidimensionale Bild nur einmal durchlaufen werden, um das Oberflächengitter zu generieren. Bei einer Implementierung in CUDA sind - wie oben beschrieben - mehrere Durchläufe nötig, da zunächst nicht bekannt ist, welcher Würfel wie viele Dreiecke erzeugt. Bei der Verwendung einer Tesla-C1060 Karte läuft die hier beschriebene Implementierung in CUDA etwa zweimal schneller als die entsprechende serielle Implementierung. Die parallele Verwendung mehrerer GPUs verbessert diesen Faktor entsprechend. Es ist anzumerken, dass die Laufzeit der Gittergenerierung gering ist, verglichen mit der Laufzeit der sonstigen in *NeuRA2* verwendeten Bildverarbeitungsoperatoren. Somit lohnen sich weitere Optimierungen an dieser Stelle nicht.

8.4 Verwendung mehrerer Grafikkarten

Um Oberflächengitter aus großen Datensätzen zu erzeugen, werden einzelne Teilgitter zunächst parallel auf mehreren Grafikkarten berechnet, um diese anschließend zu einem Gitter ohne mehrfache Vertices zu vernetzen. Hierbei wird der zu bearbeitende Datensatz mit einer Überlappung von einem Voxel in z-Richtung aufgeschnitten und die jeweiligen Teilgitter erzeugt. Anschließend werden die Teilgitter an den Unter- bzw. Oberkanten miteinander vernetzt und die doppelt vorhandenen Vertices zusammengefasst. Die Vernetzung

Größe der Daten	Vertices	Dreiecke	Teilbilder	Laufzeit
71 MByte	61038	118352	3	1.1s
506 MByte	219660	436042	17	7.5s
1440 MByte	647277	1180282	62	29.8s

Tabelle 8.2: *Anzahl generierter Dreiecke und Vertices, sowie Anzahl verwendeter Teilbilder und Laufzeit der parallelen Marching-Cubes-Implementierung auf drei Tesla-C1060 Karten unter Betrachtung verschiedener Datensätze.*

ist sehr schnell, da jeweils nur die Vertices mit minimaler bzw. maximaler z-Koordinate der einzelnen Teilgitter betrachtet werden müssen. Vergleicht man die Laufzeiten der seriellen Implementierung (Tabelle 8.1) mit denen der parallelen Implementierung (Tabelle 8.2), ist bei der Verwendung von drei Tesla-C1060 Prozessoren ein Laufzeitgewinn von etwa einem Faktor fünf gegenüber der seriellen Implementierung zu beobachten.

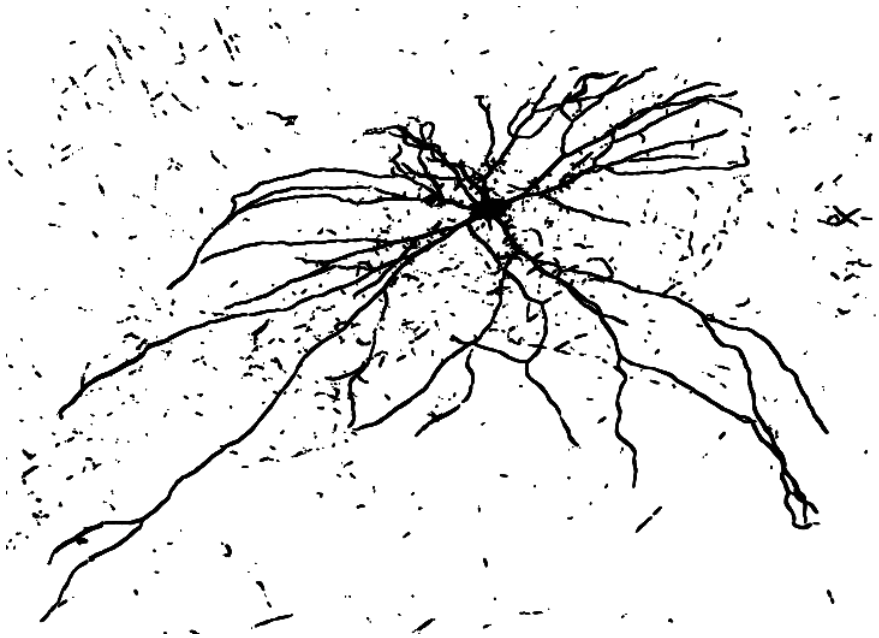
8.5 Eingabeparameter

Der einzige zu optimierende Eingabeparameter bei der Marching-Cubes-Gittererzeugung ist der *Isoflächen-Schwellwert* ϑ, der einen signifikanten Einfluss auf die Größe der rekonstruierten Geometrien hat. In Kapitel 11 ist beschrieben, wie Oberfläche und Volumen der rekonstruierten Geometrien berechnet werden können. Hierbei ist entscheidend, den *Isoflächen-Schwellwert* so einzustellen, dass das erzeugte Gitter die Oberfläche des zu rekonstruierenden Objekts optimal approximiert. Die numerische Optimierung des *Isoflächen-Schwellwerts* ist daher in Kapitel 11.3.3 zu finden.

8.6 Ergebnisse

Da im späteren Verlauf dieser Arbeit noch viele Abbildungen rekonstruierter Oberflächengeometrien zu finden sind, seien an dieser Stelle nur zwei durch den Marching-Cubes-Algorithmus erzeugte Geometrien dargestellt und dabei auf die Defizite des Marching-Cubes-Algorithmus hingewiesen: Der in Abbildung 8.3a zu sehende Neuronenzellkern besitzt eine sehr kantige Geometrie. Die rekonstruierte Neuronenzelle in Abbildung 8.3b enthält neben der eigentlichen Neuronenzelle viele kleine *Blobs*: Bereiche, die nicht mit der Neuronenzelle verbunden sind und durch Rauschen in der Mikroskopaufnahme hervorgerufen werden. Um diese Defizite zu beheben und zudem die Anzahl Dreiecke in den rekonstruierten Oberflächengeometrien zu vermindern, ist der Einsatz von Algorithmen zur Gitterzerlegung und Gitteroptimierung notwendig. Diese Verfahren werden in den nächsten beiden Kapiteln beschrieben.

(a)

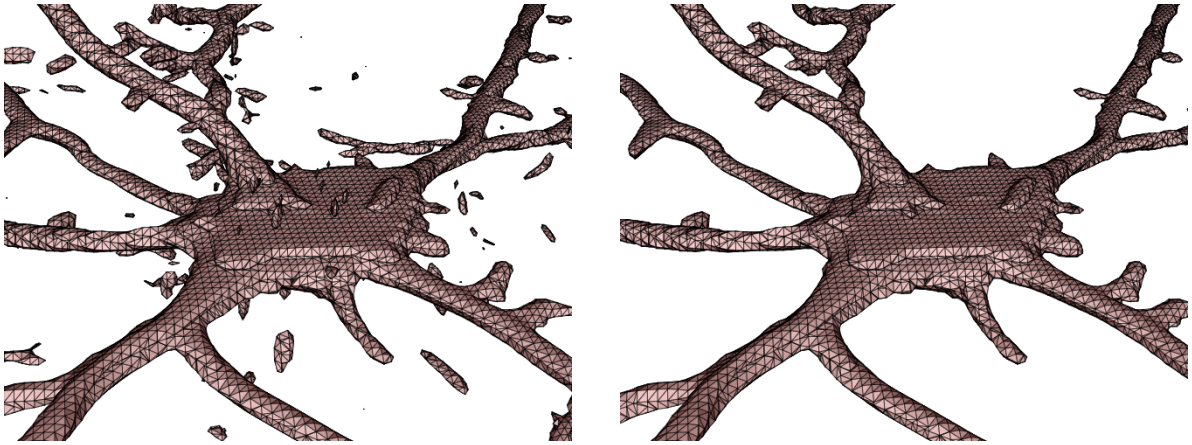


(b)

Abbildung 8.3: *Ergebnisse des Marching-Cubes-Algorithmus: (a) Rekonstruktion eines Neuronenzellkerns. (b) Rekonstruktion einer Neuronenzelle.*

Kapitel 9

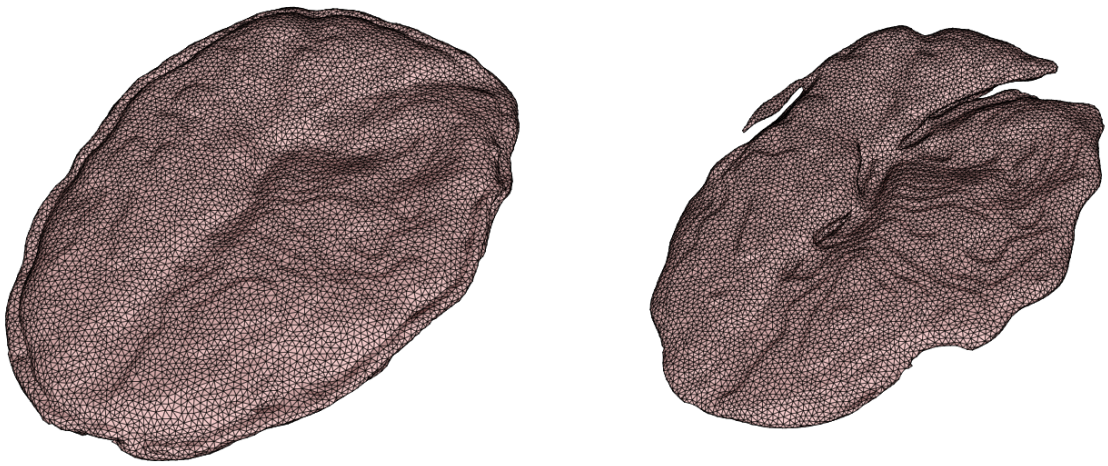
Gitterzerlegung



(a) **(b)**

Abbildung 9.1: *(a) Rekonstruierte Geometrie einer Neuronenzelle mit* Blobs. *(b) Rekonstruierte Geometrie nach Entfernung der durch Rauschen verursachten Blobs.*

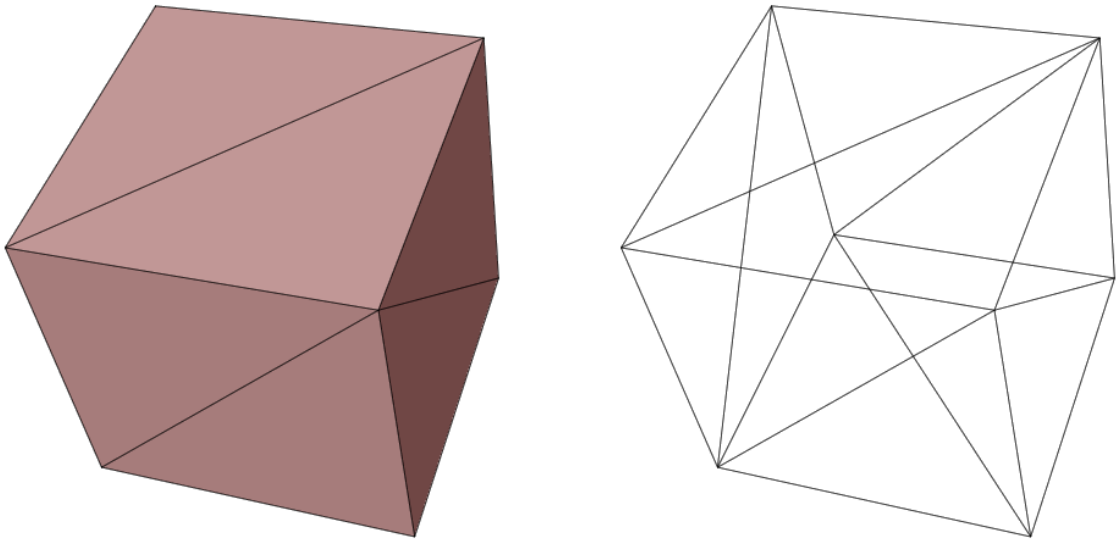
Durch die verwendeten Segmentierungsmethoden (Kapitel 7) können fälschlicherweise Bereiche zum Objekt zugeordnet werden, die eigentlich dem Rauschanteil der ursprünglichen Aufnahme entstammen und die nicht mit dem zu rekonstruierenden Objekt verbunden sind. Diese unerwünschten Bereiche werden als *Blobs* bezeichnet (Abbildung 9.1). Bei der Rekonstruktion der Geometrie durch den Marching-Cubes-Algorithmus (Kapitel 8) entstehen so disjunkte Teilgitter, die voneinander getrennt werden müssen. Das Oberflächengitter wird dabei als Graph betrachtet, dessen Zusammenhangskomponenten durch eine Tiefensuche bestimmt werden können. Jede Zusammenhangskomponente entspricht dabei einem der disjunkten Teilgitter. Auf diese Weise lässt sich zudem bei der Rekonstruktion von Neuronenzellkernen die innere Membran und die äußere Membran voneinander trennen (Abbildung 9.2, Kapitel 14.4.1).



(a) **(b)**

Abbildung 9.2: *(a) Äußere Membran eines rekonstruierten Neuronenzellkerns. (b) Innere Membran eines rekonstruierten Neuronenzellkerns. Diese wurden durch* Gitterzerlegung *voneinander getrennt.*

9.1 Oberflächengitter als Graph



(a) **(b)**

Abbildung 9.3: *(a) Einheitswürfel. (b) Einheitswürfel als Drahtgittermodell.*

Ein Oberflächengitter besteht aus einer nummerierten Liste von *Vertices*, die durch ihre kartesischen Koordinaten im dreidimensionalen Raum gegeben sind, sowie der *Triangulie-*

rung - eine Liste aller Dreiecke. Hierbei ist jedes Dreieck durch die Indizes der drei Vertices gegeben, welche die Ecken des Dreiecks beschreiben. Der Einheitswürfel (Abbildung 9.3a) wird beispielsweise durch die acht Vertices

```
0:      0.0     0.0     0.0
1:      1.0     0.0     0.0
2:      1.0     1.0     0.0
3:      0.0     1.0     0.0
4:      0.0     0.0     1.0
5:      1.0     0.0     1.0
6:      1.0     1.0     1.0
7:      0.0     1.0     1.0
```

und die zwölf Dreiecke

```
0:      0       2       1
1:      0       3       2
2:      0       1       5
3:      0       5       4
4:      1       2       6
5:      1       6       5
6:      2       3       7
7:      2       7       6
8:      3       0       4
9:      3       4       7
10:     4       5       6
11:     4       6       7
```

beschrieben. Bei der Triangulierung ist darauf zu achten, dass die einzelnen Dreiecke die gleiche Orientierung aufweisen, d.h. die Reihenfolge der begrenzenden Vertices muss für alle Dreiecke entweder im Uhrzeigersinn oder im Gegenuhrzeigersinn gegeben sein. Dies ist wichtig, da zur korrekten Darstellung der Oberflächen durch *OpenGL* die Normalen der einzelnen Dreiecke berechnet werden und diese konsistent entweder ein inneres oder ein äußeres Normalenfeld der darzustellenden Fläche bilden müssen [44].

Anschaulich ist der Graph dieses Oberflächengitters durch das Drahtgittermodell (Abbildung 9.3b) gegeben: Die Knoten des Graphen (für eine Einführung in die mathematische Disziplin der Graphentheorie sei auf [115] verwiesen) entsprechen den Vertices des Gitters und die Kanten des Graphen entsprechen den Dreieckskanten im Gitter. Die sogenannten *Zusammenhangskomponenten* des Graphen entsprechen den einzelnen disjunkten Teilgittern des Oberflächengitters und werden durch eine *Tiefensuche* separiert.

9.2 Bestimmung der Zusammenhangskomponenten durch Tiefensuche

Die Tiefensuche zur Bestimmung von Zusammenhangskomponenten innerhalb eines Graphen wird durch folgende rekursive Funktion beschrieben:

```
Tiefensuche(v) {

  Markiere den Knoten v;

  Falls ein nicht markierter Nachbarknoten w von v existiert:
    Tiefensuche(w);

}
```

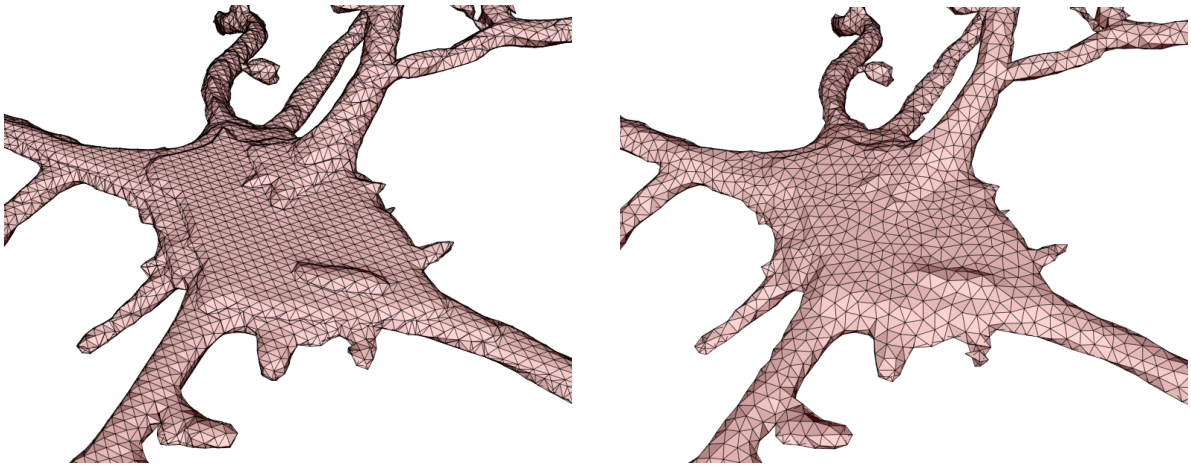
Gestartet wird mit dem Aufruf `Tiefensuche(`v_0`)` mit einem beliebigen Startknoten v_0. Nach Terminierung der Tiefensuche sind alle Knoten der Zusammenhangskomponente von v_0 markiert. Weitere Zusammenhangskomponenten lassen sich berechnen, indem die Tiefensuche mit einem bisher nicht markierten Startknoten erneut aufgerufen wird. Weitere Details zur Tiefensuche und dazugehörige Komplexitätsüberlegungen sind [93] zu entnehmen.

9.3 Implementierung

Die von Sebastian Reiter entwickelte Implementierung *Splitter* [94] ermöglicht die Zerlegung von Oberflächengittern in einzelne disjunkte Teilgitter und speichert diese in separaten Dateien ab. Hierbei können Teilgitter, die nur aus wenigen Dreiecken bestehen, gänzlich ignoriert werden.

Kapitel 10

Gitteroptimierung



(a) **(b)**

Abbildung 10.1: *(a) Durch den Marching-Cubes-Algorithmus erzeugtes und von Blobs befreites Dreiecksgitter. Überflüssige Dreiecke und scharfe Kanten sind deutlich zu erkennen. (b) Im optimierten Gitter haben alle Dreiecke in etwa die gleiche Fläche, außerdem sind die Übergänge weicher.*

Die durch den Marching-Cubes-Algorithmus erzeugten Geometrien sind teilweise sehr kantig und enthalten einige Dreiecke mit sehr kleinen Oberflächen, die prinzipiell überflüssig sind (Abbildung 10.1). Der in *NeuRA2* verwendete Algorithmus zur Gitteroptimierung ist größtenteils [45] entnommen. Mathematisch lässt sich das Problem der Gitteroptimierung wie folgt formulieren:

Gegeben sei eine Menge von Punkten $X \subset \mathbb{R}^3$ und ein *Dreiecksgitter* G_0, auch *Mesh* oder *Gitter* genannt, das diese Punkte verbindet. Gesucht ist ein Dreiecksgitter G derselben Topologie wie G_0, das G_0 möglichst gut approximiert und aus möglichst wenigen Vertices besteht.

10.1 Gitterrepräsentation

Ein Dreiecksgitter ist eine stückweise lineare Fläche, deren Dreiecksflächen an Kanten zusammengefügt sind. Die Eckpunkte der Dreiecke werden hierbei als *Vertices* bezeichnet. Im Rahmen des hier diskutierten Optimierungs-Algorithmus ist es wichtig, zwischen der geometrischen Position der Vertices und der *Triangulierung*, d.h. welche Vertices durch Dreiecke verbunden sind, zu unterscheiden (vgl. Kapitel 9.1). Daher ist ein Dreiecksgitter G formal durch das Tupel (V, T) gegeben, wobei $V \subset \mathbb{R}^3$ die Menge der Raumkoordinaten aller Vertices und T die Triangulierung des Dreiecksgitters beschreibt. Die einzelnen Dreiecke werden dabei durch ihre Kanten $\{j, k\} \in T$ beschrieben, welche die Vertices v_j und v_k verbinden.

10.2 Formulierung als Minimierungsproblem

Das oben beschriebene Optimierungsproblem, welches das Gitter $G_0 = (V_0, T_0)$ in das Gitter $G = (V, T)$ transformiert, lässt sich durch Minimierung des Energiefunktionals

$$E(V, T) = E_{dist}(V, T) + E_{rep}(V) + E_{spring}(V, T) \tag{10.1}$$

lösen [45]. Hierbei beschreibt

$$E_{dist}(V, T) = \sum_{i=1}^{n} d(x_i, G)^2 \tag{10.2}$$

die Summe der quadrierten Abstände vom Gitter G zu festzulegenden Kontrollpunkten $X = \{x_1, ..., x_n\}$. Die Anzahl der Kontrollpunkte, die vom Benutzer einzustellen ist, bestimmt wie exakt das optimierte Gitter das ursprüngliche Gitter approximiert. Hierbei werden die Kontrollpunkte zufällig aus den Vertices des zu optimierenden Gitters generiert [94]. Der Term

$$E_{rep}(V) = c_{rep} \, |V| \tag{10.3}$$

gewichtet die Anzahl Vertices $|V|$ im optimierten Gitter G. Durch das Entfernen von Vertices zu G wird der Term $E_{dist}(V, T)$ kleiner, jedoch wird das Entfernen von Vertices durch den Term $E_{rep}(V)$ bestraft. Das Verhältnis zwischen Anzahl Vertices und der Genauigkeit des optimierten Gitters wird dabei durch den Parameter c_{rep} gesteuert. Beim Minimum der Summe beider Terme würde man intuitiv ein optimiertes Gitter mit wenigen Vertices, welches das ursprüngliche Gitter möglichst gut approximiert, erwarten. Dieses Minimum existiert allerdings nicht notwendigerweise [66], was zu seltsamen Spitzen, auch *Spikes* genannt, in den optimierten Geometrien führen kann (Abbildung 10.2). Um die Existenz eines Minimums von dem Optimierungsproblem (10.1) zu garantieren, wird der Federungsterm

$$E_{spring}(V, T) = \sum_{\{j,k\} \in T} \kappa \, \|v_j - v_k\|_2^2 \tag{10.4}$$

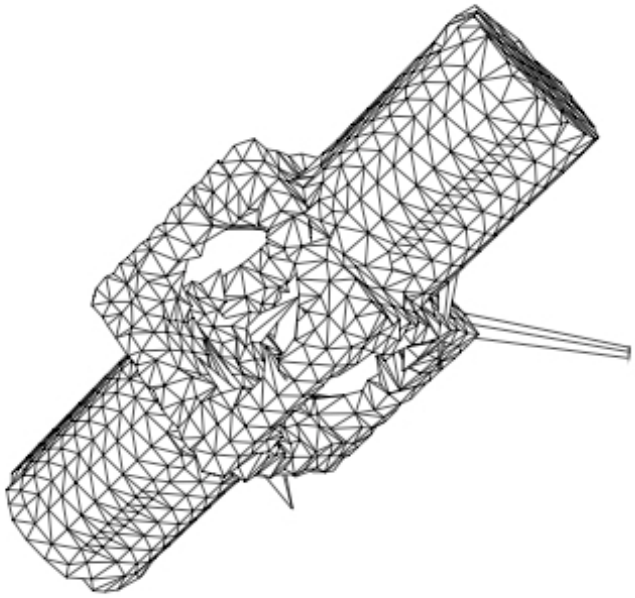


Abbildung 10.2: *Mögliche Entstehung von sogenannten Spikes bei der Lösung des Optimierungsproblems* (10.1) *ohne Berücksichtigung des Federungsterms* $E_{spring}(V,T)$. *Quelle:* [45]

eingeführt [46]. Anschaulich wird auf jeder Kante des Gitters eine Feder mit Federstärke κ und Ruhelänge 0 gespannt, die versucht den Abstand zwischen den einzelnen Vertices zu minimieren. Dieser Term garantiert zudem, dass keine entarteten Dreiecke entstehen, bei denen einzelne Kanten deutlich länger als die übrigen Kanten sind. Im Gegenteil - der Federungsterm garantiert, dass die resultierenden Dreiecke möglichst gleichseitig werden [45].

10.3 Lösung des Minimierungsproblems

Ziel ist es, das Energiefunktional (10.1) über der Menge $\mathcal{T}$ aller möglichen Dreiecksgitter, die zum ursprünglichen Gitter homöomorph sind, zu minimieren. Die Arbeitsweise, des in [45] vorgestellten Algorithmus sei an dieser Stelle skizziert:

Um $E(V,T)$ zu minimieren wird das Minimierungsproblem in zwei verschachtelte Teilprobleme zerlegt: Eine innere Minimierung über die Vertexmenge V, für eine feste Triangulierung T, und eine äußere Minimierung über die Triangulierung T.

Der folgende Pseudo-Code, der den prinzipiellen Ablauf des Lösungsalgorithmus beschreibt, ist [45] entnommen:

```
OptimizeMesh(V_0, T_0) {

   V := OptimizeVertexPositions(V_0, T_0)
    T := T_0

   // Löse äußeres Minimierungsproblem:
    repeat {

      (V', T') := GenerateLegalMove(V, T)
       V' := OptimizeVertexPositions(V', T')
       if E(V', T') < E(V, T) then (V, T) := (V', T')

   } until convergence
    return  (V, T)
}
```

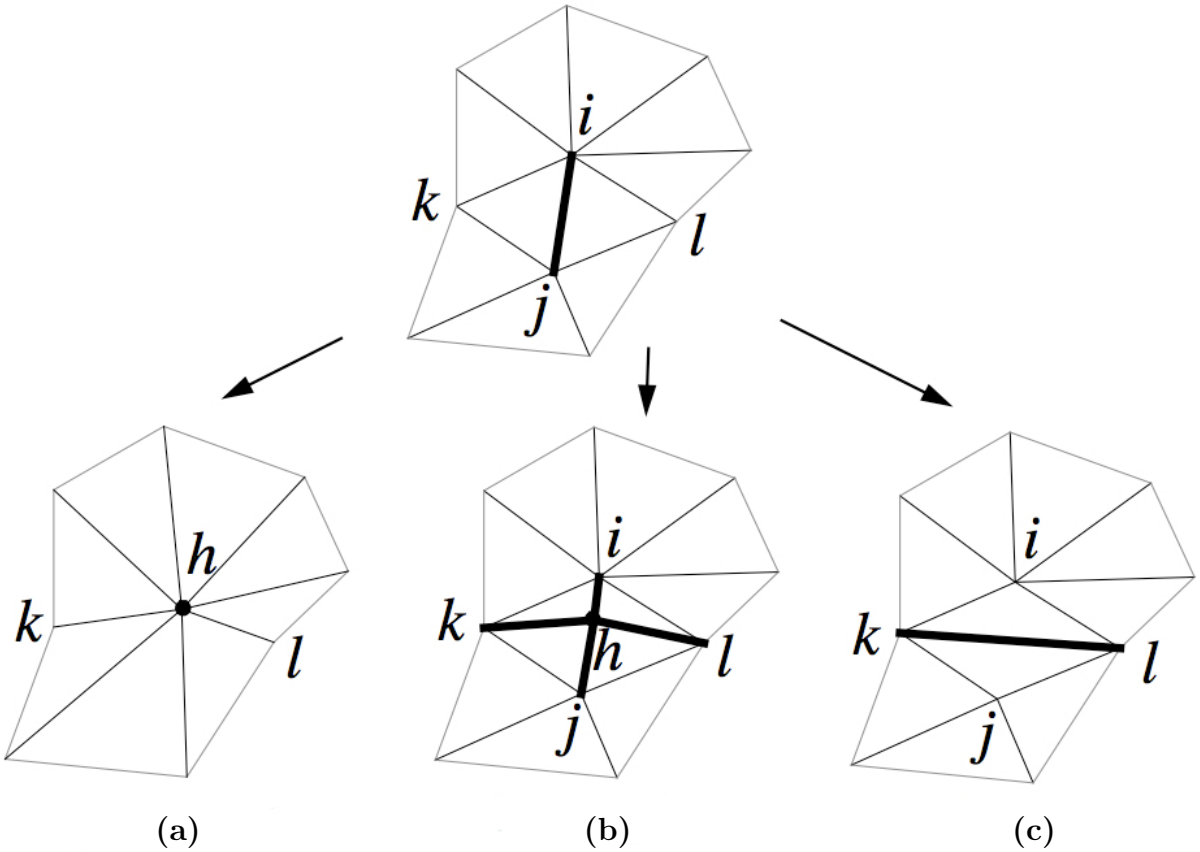


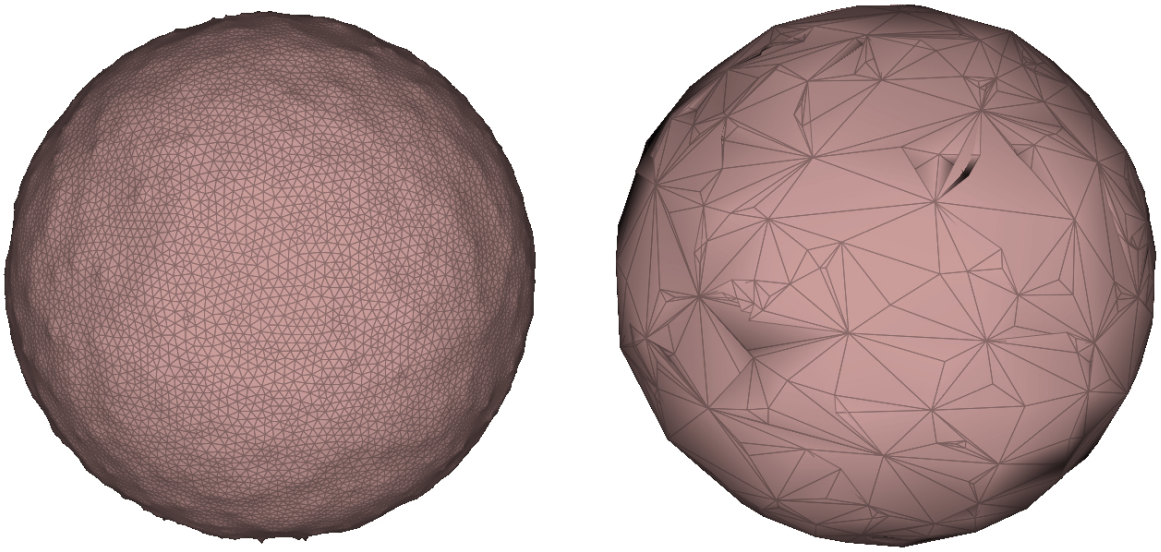
Abbildung 10.3: *Lokale Optimierung: (a) Zusammenziehen (Edge Collapse), (b) Teilen (Edge Split) und (c) Vertauschen von Dreieckskanten (Edge Swap). Quelle:* [45]

Hierbei löst die Funktion `OptimizeVertexPositions`(V, T) das innere Minimierungsproblem

$$E(T) = \min_V E(V, T) \tag{10.5}$$

für eine feste Triangulierung T, wobei das auftretende lineare Gleichungssystem mit dem Verfahren konjugierter Gradienten [54] effizient gelöst werden kann. Die Funktion `GenerateLegalMove` optimiert das Gitter lokal durch Zusammenziehen, Teilen oder Vertauschen von einzelnen Kanten der Triangulierung (Abbildung 10.3). Beim Zusammenziehen oder Vertauschen von Kanten muss dabei überprüft werden, dass sich die Topologie des Gitters nicht verändert.

10.4 Eingabeparameter



(a) **(b)**

Abbildung 10.4: *(a) Bei zu wenigen Kontrollpunkten weicht die optimierte Geometrie stark von dem ursprünglichen Gitter ab. (b) Bei einer zu kleinen Wahl der Federkonstante κ entstehen stark entartete Dreiecke. Die ursprüngliche Geometrie in Form einer Kugel ist in Abbildung 10.6a zu finden.*

Das Resultat der Gitteroptimierung hängt im Wesentlichen von drei festzulegenden Parametern ab, die jeweils die Gewichtung der drei Energieterme des zu minimierenden Funktionals (10.1) bestimmen:

- Die Anzahl der Kontrollpunkte bestimmt, wie exakt das optimierte Gitter das ursprüngliche Gitter approximiert.

- Über den Parameter c_{rep} wird gesteuert, ob das optimierte Gitter aus vielen oder wenigen Dreiecken bestehen soll.
- Die Federkonstante κ bestimmt die Qualität der Dreiecke, d.h. ob diese nahezu gleichseitig oder teilweise entartet sein dürfen.

Um Gitter in unterschiedlichen Auflösungsstufen einfacher zu erhalten, kann zudem noch ein Parameter eingeführt werden, der angibt aus wie vielen Dreiecken das optimierte Gitter mindestens bestehen soll [94]. Damit werden die Anzahl Kanten, die in der lokalen Gitteroptimierung zusammengezogen werden, begrenzt. Die konkrete Wahl der Parameter muss für jedes zu optimierende Gitter per Hand eingestellt werden. Bei der Verwendung zu weniger Kontrollpunkte weicht das optimierte sehr stark vom ursprünglichen Gitter ab (Abbildung 10.4a). Bei einer zu kleinen Wahl der Federkonstante κ entstehen hingegen stark entartete Dreiecke (Abbildung 10.4b). Tritt einer dieser Effekte auf, muss der entsprechende Parameter angepasst werden. Gitter mit zu wenigen Dreiecken approximieren das ursprüngliche Gitter ebenfalls nur unzureichend (Abbildung 10.5f). Um letzteres Problem zu vermeiden, kann entweder der Parameter c_{rep} verringert, oder die minimale Anzahl Vertices im optimierten Gitter erhöht werden. Es ist sinnvoll, ein zu optimierendes Gitter mit unterschiedlichen Parametereinstellungen auf verschiedene Qualitätsstufen zu reduzieren, um anschließend das für den entsprechenden Zweck best geeignete optimierte Gitter auswählen zu können. Weitere Informationen zur passenden Wahl der Parameter sind [45, 94] zu entnehmen.

10.5 Implementierung

Die in *NeuRA2* verwendete Implementierung des hier vorgestellten Algorithmus zur Gitteroptimierung stammt von Sebastian Reiter, dem an dieser Stelle herzlich für seine hervorragende Arbeit gedankt sei. Details zur Implementierung und Benutzung sind [94] zu entnehmen. Einstellungen an dem Ergebnis der Gitteroptimierung können über die in Kapitel 10.4 beschriebenen Parameter vorgenommen werden.

10.6 Ergebnisse

Im Folgenden sind die Ergebnisse der Anwendung der Gitteroptimierung auf Geometrien, die vom Marching-Cubes-Algorithmus (Kapitel 8) generiert wurden, gezeigt. Hierbei werden ein Modell eines Dendriten (Abbildung 10.5), eine kugelförmige Testgeometrie (Abbildung 10.6), eine rekonstruierte Neuronenzelle (Abbildung 10.7) und ein Keramikgefäß (Abbildung 10.8) betrachtet. Bei genauer Betrachtung der optimierten Gitter ist zu erkennen, dass die Dreiecke, die der Marching-Cubes-Algorithmus erzeugt, deutlich reduziert werden, ohne die Topologie des Gitters zu verändern. Im Gegenteil: Die vom Marching-Cubes-Algorithmus erzeugten scharfen Kanten werden merklich geglättet. Wird die Anzahl der Dreiecke allerdings zu stark reduziert, ist mit bloßem Auge zu erkennen, dass sich ursprüngliches Gitter und optimiertes Gitter unterscheiden können (Abbildung 10.5f). In

Kapitel 11.3.2 ist gezeigt, dass sich Oberfläche und Volumen der optimierten Geometrien kaum von denen der ursprünglichen Geometrien unterscheiden, wenn bei der Optimierung hinreichend viele Kontrollpunkte verwendet werden.

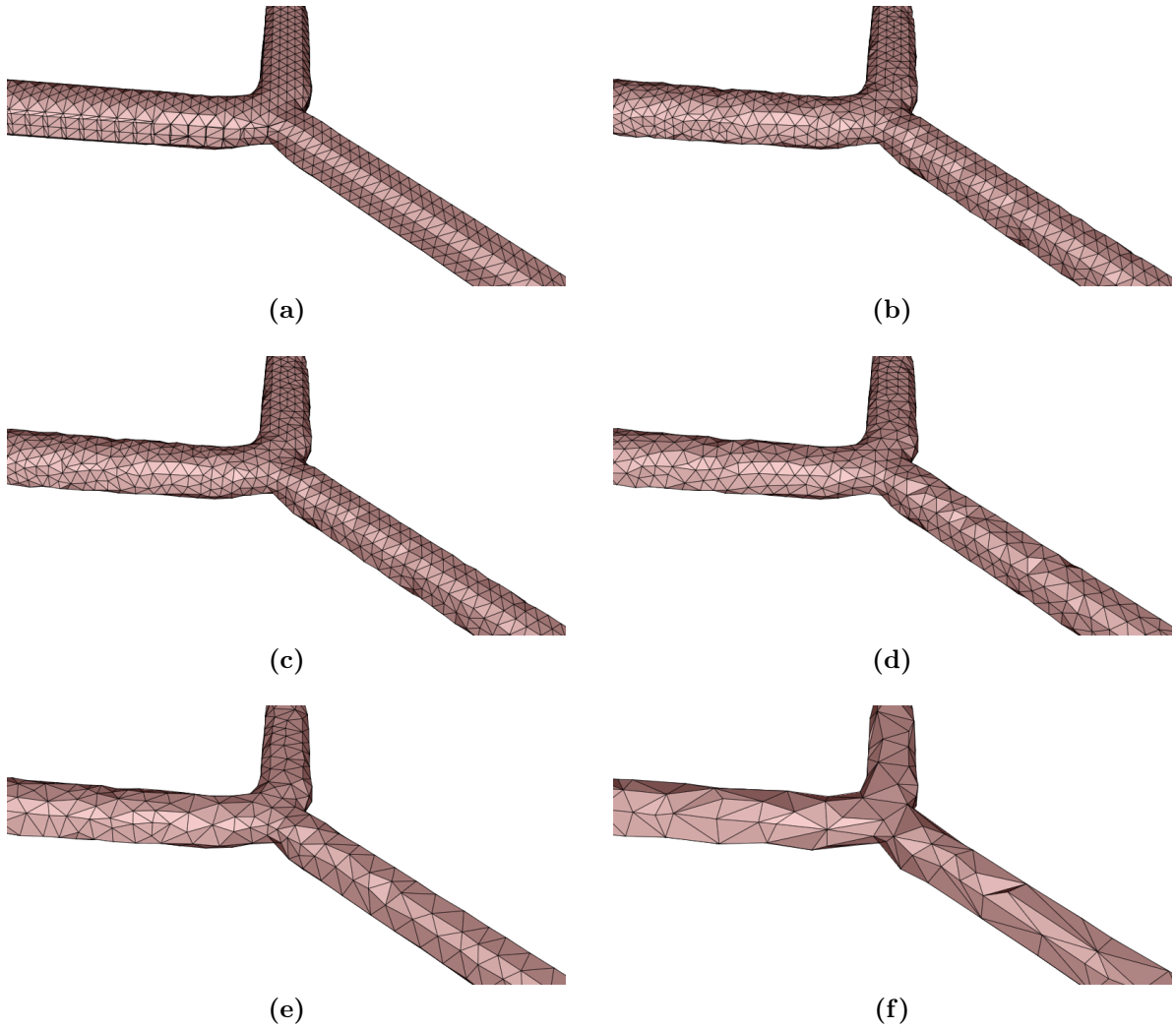


Abbildung 10.5: *Mit 3000 Kontrollpunkten optimierte Modellgeometrie einer Dendritenverzweigung. (a) Durch den Marching-Cubes-Algorithmus erzeugte Geometrie mit 3672 Dreiecken. (b) bis (f) optimierte Geometrien bestehend aus (b) 3000, (c) 2800, (d) 2000, (e) 1200 und (f) 500 Dreiecken.*

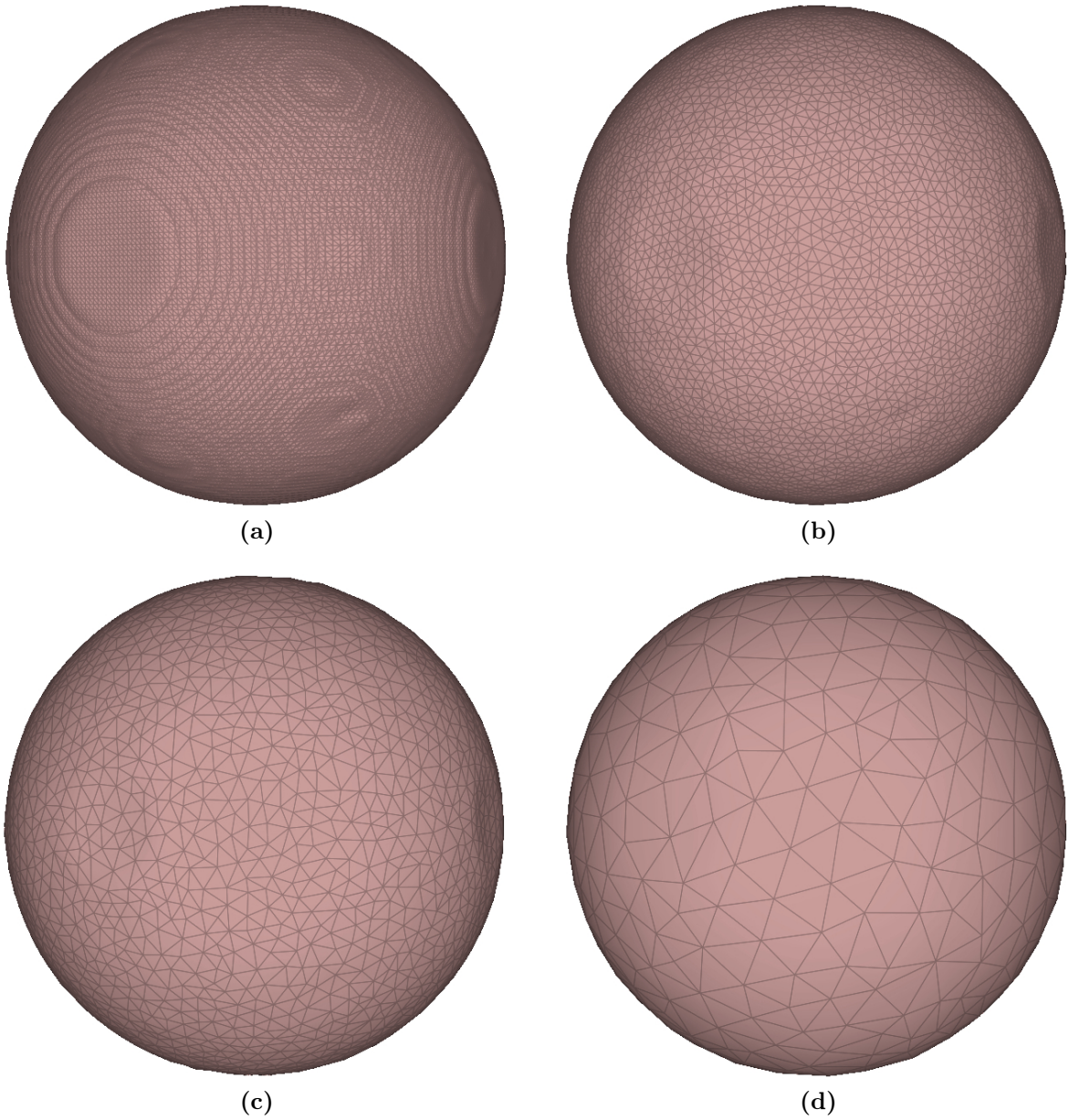
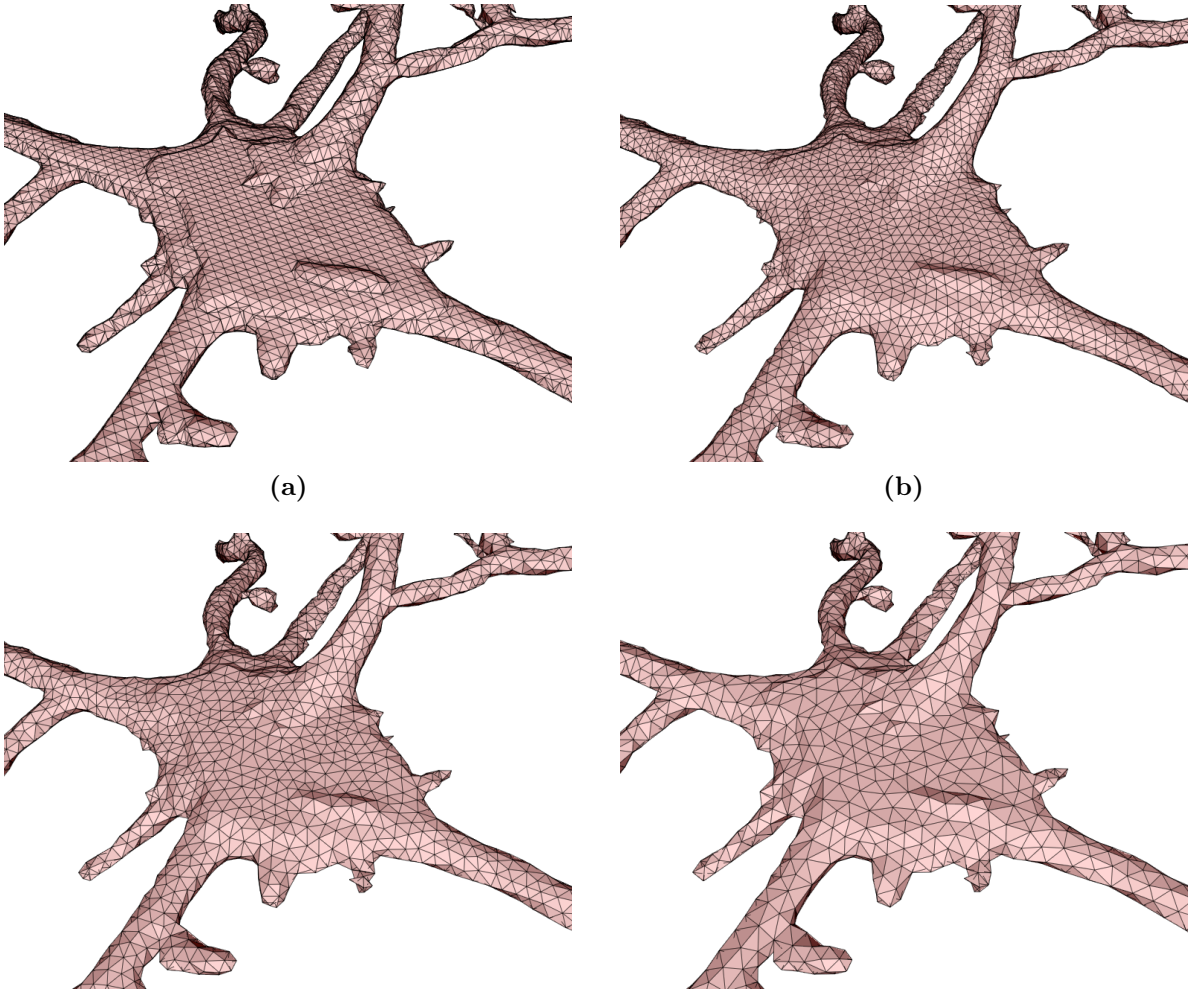
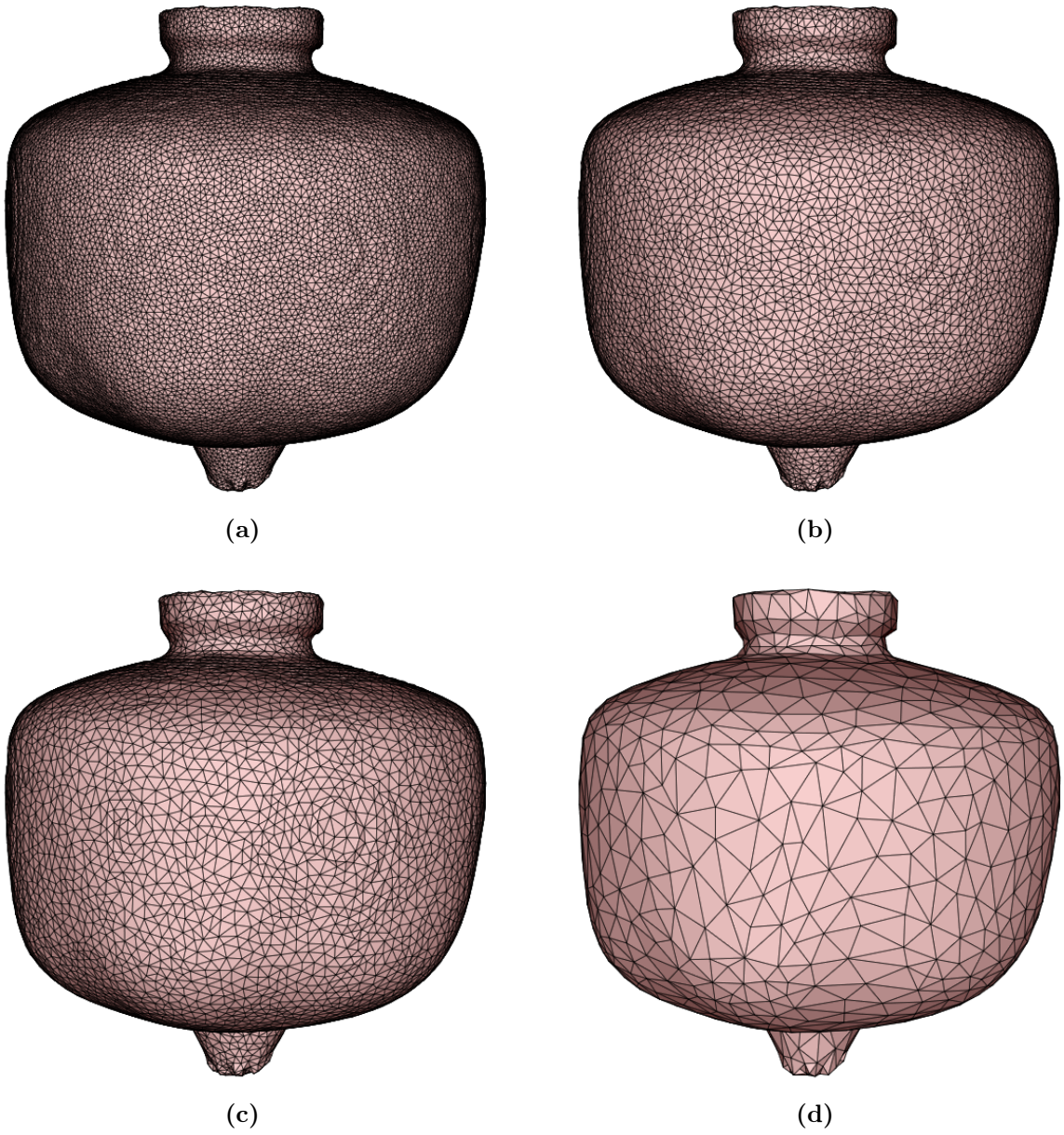


Abbildung 10.6: *Mit 20000 Kontrollpunkten optimierte Geometrie einer Kugel. (a) Durch den Marching-Cubes-Algorithmus erzeugte Geometrie mit 225000 Dreiecken. (b), (c) und (d) optimierte Geometrien bestehend aus (b) 40000, (c) 12500 und (d) 2000 Dreiecken.*



(a) **(b)** **(c)** **(d)**

Abbildung 10.7: *Mit 100000 Kontrollpunkten optimierte Geometrie einer Neuronenzelle. (a) Durch den Marching-Cubes-Algorithmus erzeugte Geometrie mit 80000 Dreiecken. Optimierte Geometrien mit (b) 78000, (c) 60000 und (d) 40000 Dreiecken.*



(a) **(b)**

(c) **(d)**

Abbildung 10.8: *Geometrie eines rekonstruierten Keramikgefäßes mit unterschiedlicher Detailstufe bestehend aus (a) 170000, (b) 130000, (c) 50000 und (d) 4000 Dreiecken.*

Kapitel 11

Berechnung von Oberflächen und Volumina

Das Berechnen von Oberflächen und Volumina der rekonstruierten Objekte ist aus mehreren Gründen interessant: Bei archäologischen Rekonstruktionen lassen sich einerseits durch die Berechnung des Füllvolumens der Gefäße Rückschlüsse über die verwendeten Maßeinheiten verschiedener Epochen gewinnen und andererseits lässt sich durch die Kenntnis über das Volumen der Keramik selbst ihre physikalische Dichte errechnen. In [90] wurde mit Hilfe der Oberflächen- und Volumenberechnung von *NeuRA* der Zusammenhang zwischen Einfaltungen in Neuronenzellkernen und deren Oberfläche und Volumina aufgezeigt (Kapitel 14.4.1). Daher ist in diesem Kapitel die Methodik zur Berechnung von Oberfläche und Volumen der rekonstruierten Geometrien erklärt und gezeigt, dass der Einfluss der Gitteroptimierung (Kapitel 10) auf Oberfläche und Volumen rekonstruierter Objekte gering ist. Des Weiteren ist gezeigt, dass der Fehler durch die Approximation der Objekte durch Dreiecksgitter bei richtiger Einstellung des Isoflächen-Schwellwerts des Marching-Cubes-Gittergenerators (Kapitel 8) sehr klein ausfällt. Abschließend ist in diesem Kapitel evaluiert, dass die Bestimmung von Volumina rekonstruierter Keramikgefäße genauer ist, als herkömmliche archäologische Methoden.

11.1 Berechnung von Oberflächen

Das Berechnen der Oberflächen der rekonstruierten Dreiecksgitter ist sehr einfach: Es wird der Flächeninhalt jedes einzelnen Dreiecks berechnet und diese Flächeninhalte aufsummiert. Die Fläche A des Dreiecks mit den Randpunkten x, y, z in dreidimensionalen kartesischen Koordinaten lässt sich durch die Formel

$$A = 0.5\,|(y - x) \times (z - x)| \tag{11.1}$$

berechnen.

11.2 Berechnung von Volumina

Für die Berechnung der Volumina von Dreiecksgittern finden in *NeuRA2* zwei unterschiedliche Verfahren Anwendung: Einerseits kann das Oberflächengitter in ein aus Tetraedern bestehendes Volumengitter überführt werden, oder das Volumen direkt mit Hilfe des Gaußschen Integralsatzes durch eine Summation über alle Dreiecke berechnet werden.

11.2.1 Berechnung von Volumina durch Volumengitter

Aus dem Oberflächengitter wird zunächst ein Volumengitter generiert. In *NeuRA2* kommt hierzu der Algorithmus *TetGen* [105] zum Einsatz, der aus geschlossenen Dreiecksgittern Volumengitter aus Tetraedern erzeugt, welche die Delaunay-Bedingung erfüllen. Hierbei dient die Summe der Volumina der einzelnen Tetraeder wiederum als Approximation des Volumens des rekonstruierten Objekts. Das Volumen V eines Tetraeders mit Eckpunkten $x = (x_0, x_1, x_2), y = (y_0, y_1, y_2), z = (z_0, z_1, z_2), w = (w_0, w_1, w_2)$ errechnet sich durch die Formel

$$V = \frac{1}{6} \det(x - y, x - z, x - w) \tag{11.2}$$

bzw. in Koordinatenform durch

$$V = \frac{1}{6} \det \begin{pmatrix} x_0 - y_0 & x_1 - z_0 & x_2 - w_0 \\ x_0 - y_1 & x_1 - z_1 & x_2 - w_1 \\ x_0 - y_2 & x_1 - z_2 & x_2 - w_2 \end{pmatrix}. \tag{11.3}$$

Die durch *TetGen* generierten Volumengitter können zudem dazu verwendet werden, Simulationen auf rekonstruierten Geometrien durchzuführen (Kapitel 14.2).

11.2.2 Berechnung von Volumina durch den Satz von Gauß

Wesentlich schneller lässt sich das Volumen mit Hilfe des Integralsatzes von Gauß berechnen. Hierbei wird das Volumen V des, durch eine geschlossene stückweise lineare Fläche berandeten, Gebietes Ω folgendermaßen berechnet

$$V = \int_\Omega 1 \, dV = \int_\Omega \nabla \circ (x, 0, 0)^T \, dV. \tag{11.4}$$

Die Konstante 1 wird dabei durch $\nabla \circ (x, 0, 0)^T$ ersetzt. Durch den Integralsatz von Gauß [58] wird das Volumenintegral über die Divergenz des Vektorfelds $(x, 0, 0)^T$ in das Oberflächenintegral

$$V = \int_{\partial\Omega} (x, 0, 0)^T \circ \vec{n} \, dS \tag{11.5}$$

umgeformt, wobei $\vec{n}$ die äußere Einheitsnormale von $\partial\Omega$ bezeichnet. Das resultierende Oberflächenintegral kann numerisch durch

$$V = \sum_{\Delta} |\Delta| \cdot (x_{\Delta}, 0, 0)^T \circ \vec{n_{\Delta}} \tag{11.6}$$

berechnet werden, wobei über alle Dreiecke Δ des Gitters summiert wird und $|\Delta|$ den Flächeninhalt, x_{Δ} die x-Koordinate des Schwerpunktes und $\vec{n_{\Delta}}$ die äußere Normale des Dreiecks Δ bezeichnet. Die Formel ist exakt, da zur Auswertung des Integrals für jedes einzelne Dreieck die Mittelpunktregel angewandt wird [116]. Weitere Details und Pseudocode zur Volumenbestimmung nach Gauß finden sich in [31].

11.3 Fehlerabschätzung

In diesem Abschnitt wird zunächst der Unterschied zwischen den beiden Methoden zur Berechnung von Volumina betrachtet. Hierbei zeigt sich, dass beide Verfahren zur Volumenbestimmung praktisch die gleichen Ergebnisse liefern. Anschließend wird der Einfluss der einzustellenden Parameter bei der Gittererzeugung und der Gitteroptimierung auf Oberfläche und Volumen der rekonstruierten Geometrien untersucht. Zudem wird gezeigt, dass die Gitteroptimierung Oberfläche und Volumen der Geometrien nicht nennenswert verändert und der für die Oberflächen- und Volumenberechnung optimale Marching-Cubes-Isoflächen-Schwellwert bestimmt. Für die letzten beiden Untersuchungen dienen jeweils Kugeln, da diese eine gekrümmte Oberfläche besitzen und sich Oberfläche und Volumen exakt errechnen lassen.

11.3.1 Unterschied der Volumenberechnungsverfahren

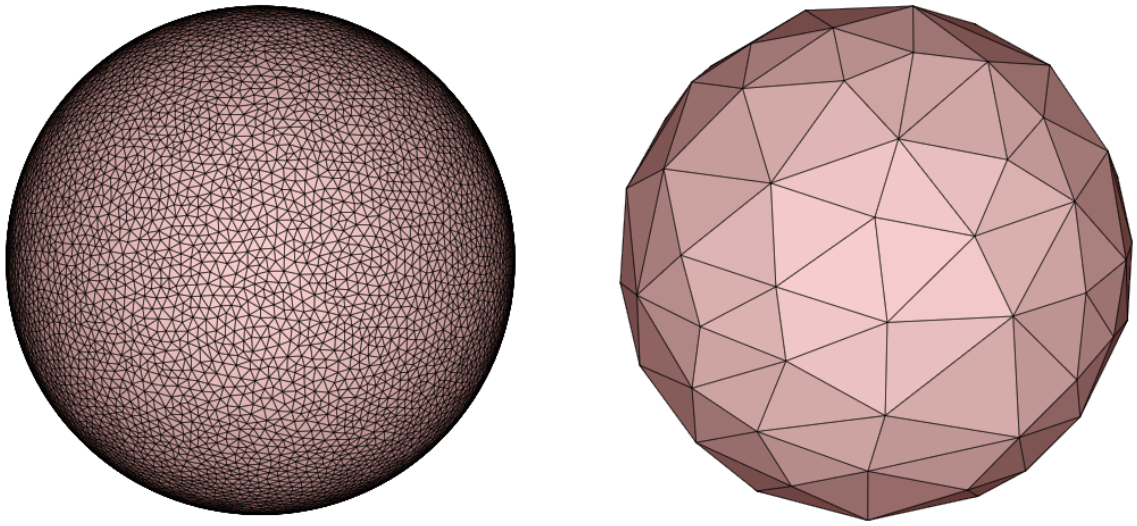
Objekt	Dreiecke	Tetraeder	Gauß	Abweichung
Einheitswürfel	12	1	1	0%
Keramikgefäß	17778	4257024.582467	4257024.569773	$3 \cdot 10^{-7}$ %
Kugel	26114	856912.50465	856912.50406	$7 \cdot 10^{-8}$ %
Bouton	53524	21756.409757	21756.409787	$1.4 \cdot 10^{-7}$ %
Zellkern	100000	241371.895365	241371.895654	$1.2 \cdot 10^{-7}$ %

Tabelle 11.1: *Unterschied der beiden Volumenberechnungsverfahren.*

Zur Bestimmung des Unterschiedes der oben erklärten Methoden zur Volumenberechnung eines geschlossenen Oberflächengitters wurden von fünf unterschiedlich komplexen Objekten jeweils das Volumen durch die beiden oben beschriebenen Verfahren berechnet. Die Werte der Volumen sind hierbei dimensionslose Größen. Tabelle 11.1 ist zu entnehmen, dass die, durch Rundungsfehler bedingten, Abweichungen der beiden Verfahren so gering sind, dass die beiden Verfahren als gleichwertig zu betrachten sind. Es ist hierbei zu beachten, dass die Rechnungen in einfacher Genauigkeit durchgeführt wurden. Wird das

Volumengitter nicht für weitere Anwendungen benötigt ist in der Praxis aufgrund seiner höheren Geschwindigkeit das Verfahren von Gauß zur Volumenberechnung vorzuziehen.

11.3.2 Einfluss der Gitteroptimierung



(a) **(b)**

Abbildung 11.1: *Die Testgeometrie der Einheitskugel bestehend aus (a)* 20156 *Dreiecken und (b)* 202 *Dreiecken.*

Um die Methoden zur Berechnung von Oberfläche und Volumen der rekonstruierten Geometrien zu evaluieren und den Einfluss des Gitteroptimierungs-Algorithmus (Kapitel 10) zu untersuchen, sei eine Einheitskugel (Abbildung 11.1) betrachtet, da deren Oberfläche und Volumen exakt berechnet werden kann. Die Geometrie der Einheitskugel wurde mit der Software *Blender* [98] erzeugt und besteht aus 327680 Dreiecken. Exakte Oberfläche A und Volumen V der Kugel lassen sich durch die Formeln

$$A = 4\pi r^2 \tag{11.7}$$

und

$$V = \frac{4}{3}\pi r^3 \tag{11.8}$$

berechnen. Bei der Einheitskugel ($r = 1$) ergibt sich für die Oberfläche $A = 12.56637$ und für das Volumen $V = 4.18879$. Die Anzahl der Dreiecke der Einheitskugel wurde mit Hilfe der Gitteroptimierung schrittweise auf 20156, 1790, 474 und 202 verringert. Hierfür wurden

Anzahl Dreiecke	c_{rep}	κ
20156	10^{-4}	0.16
1790	10^{-3}	0.13
474	10^{-2}	0.1
202	10^{-1}	0.08

Tabelle 11.2: *Parametereinstellung bei der Gitteroptimierung.*

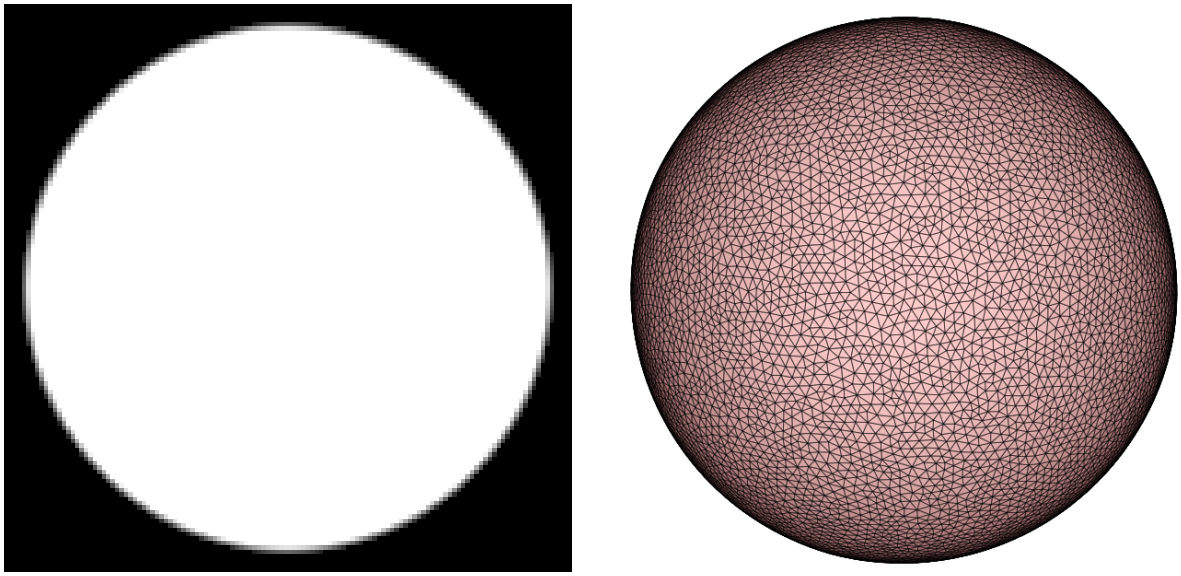
Anzahl Dreiecke	Oberfläche	Rel. Fehler	Volumen	Rel. Fehler
Exakter Wert	12.56637	0	4.18879	0
327680	12.5661	0.0019%	4.18865	0.0034%
20156	12.5659	0.004%	4.18808	0.017%
1790	12.5763	0.079%	4.18818	0.015%
474	12.6075	0.33%	4.18829	0.012%
202	12.666	0.79%	4.18932	0.013%

Tabelle 11.3: *Parametereinstellung bei der Gitteroptimierung.*

jeweils 163840 Kontrollpunkte, d.h. die halbe Anzahl der Dreiecke des ursprünglichen Gitters, verwendet, damit die optimierte Geometrie die ursprüngliche Geometrie hinreichend gut approximiert [46]. Die Parametereinstellungen für c_{rep} und κ sind Tabelle 11.2 zu entnehmen. Es zeigt sich, dass selbst bei stark optimierten Gittern, d.h. bei Gittern die nur noch aus wenigen Dreiecken bestehen, der relative Fehler in der Oberflächen- und Volumenberechnung sehr klein ist (Tabelle 11.3). Selbst bei einer Reduktion auf 202 Dreiecke ist der Fehler noch unter einem Prozent. Erwähnenswert ist, dass bei der Volumenberechnung der Fehler nahezu konstant bleibt, während er bei der Oberflächenberechnung anwächst. Bei glatten Oberflächen hat die Verwendung der Gitteroptimierung somit keinen signifikanten Einfluss auf die Berechnung von Oberfläche und Volumen. Es bleiben bei der Oberflächen- und Volumenberechnung noch zwei weitere Aspekte zu betrachten: Einerseits der Einfluss der Gitteroptimierung auf durch den Marching-Cubes-Algorithmus erzeugte Geometrien und andererseits der Einfluss des Isoflächen-Schwellwerts bei der Marching-Cubes-Gittergenerierung.

11.3.3 Optimierung des Isoflächen-Schwellwerts des Marching-Cubes-Algorithmus

Es sei ein Volumenbild der Dimension 128^3, das eine Kugel mit Radius 60 enthält, betrachtet (Abbildung 11.2a). Berechnet man das Volumen der in dem Volumenbild enthaltenen Testkugel als gewichtete Summe über alle Voxel des Bildes, ergibt sich ein relativer Fehler von 0.08% gegenüber dem Volumen einer exakten Kugel mit Radius 60. Durch



(a) **(b)**

Abbildung 11.2: *(a) Schnittebenenbild der Testkugel mit Radius* 60*. (b) Aus* 25270 *Dreiecken bestehendes optimiertes Gitter der Testkugel. Zur Gittererzeugung wurde der in diesem Kapitel als optimal evaluierte Isoflächen-Schwellwert* 157 *bei der Marching-Cubes-Gittererzeugung verwendet.*

den Marching-Cubes-Algorithmus wird aus dem Volumenbild der Testkugel eine Oberflächengeometrie erzeugt, wobei der Isoflächen-Schwellwert im ganzzahligen Intervall $[1, 254]$ variiert wird. Die erzeugten Gitter bestehen, abhängig vom Isoflächen-Schwellwert, aus etwa 130000 bis 140000 Dreiecken. Da, wie oben gesehen, bei Kugelgeometrien, die aus ungefähr 20000 Dreiecken bestehen, Volumen und Oberfläche sehr exakt berechnet werden können, werden die erzeugten Gitter zunächst mit den Parametern $c_{rep} = 0.0001$, $\kappa = 0.2$ und 70000 Kontrollpunkten zu Kugelgeometrien, die aus ungefähr 25000 Dreiecken bestehen, optimiert (Abbildung 11.2b). Von allen auf diese Weise erzeugten Geometrien wird die Oberfläche und das Volumen auf die oben beschriebene Weise berechnet (Abbildungen 11.3 und 11.4). Den beiden Schaubildern ist zu entnehmen, dass Oberfläche und Volumen mit Ausnahme der Randbereiche linear vom Isoflächen-Schwellwert abhängen. Dies ist nicht verwunderlich, da der Marching-Cubes-Algorithmus bei der Erzeugung der Vertexkoordinaten eine lineare Interpolation in Abhängigkeit des Isoflächen-Schwellwerts durchführt (Kapitel 8.1.2). Die Abweichung von Oberfläche und Volumen ist bei Verwendung des Isoflächen-Schwellwerts 157 mit 0.004% bzw. 0.018% minimal. Folglich können Oberfläche und Volumen von Objekten auf diese Weise sehr exakt bestimmt werden, da dieser Fehler unterhalb der Genauigkeit des Volumenbildes liegt.

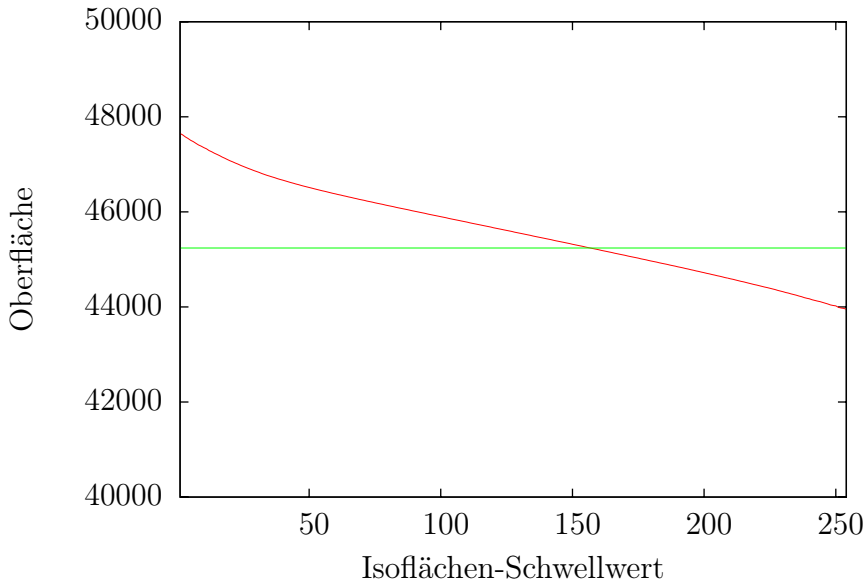


Abbildung 11.3: *Abhängigkeit der Oberfläche vom Isoflächen-Schwellwerts des Marching-Cubes-Algorithmus. Der exakte Wert ist durch die grüne Gerade dargestellt.*

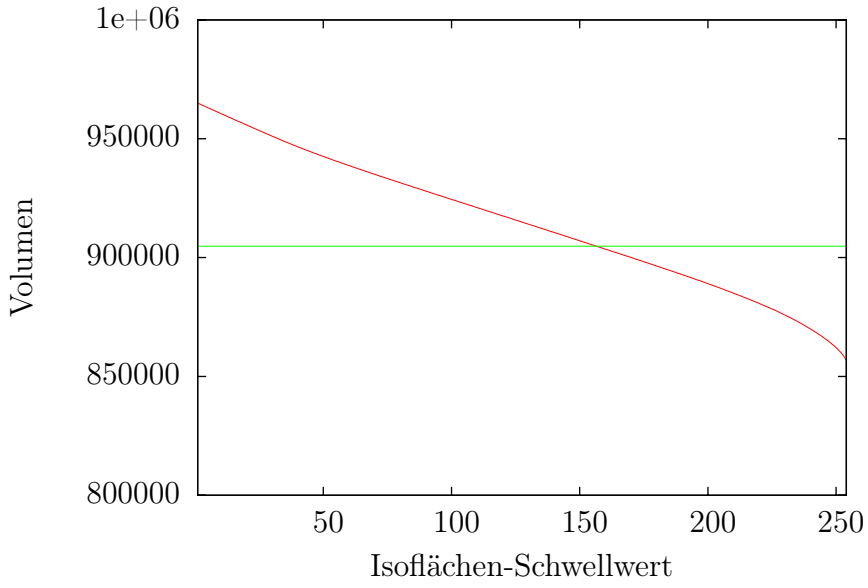
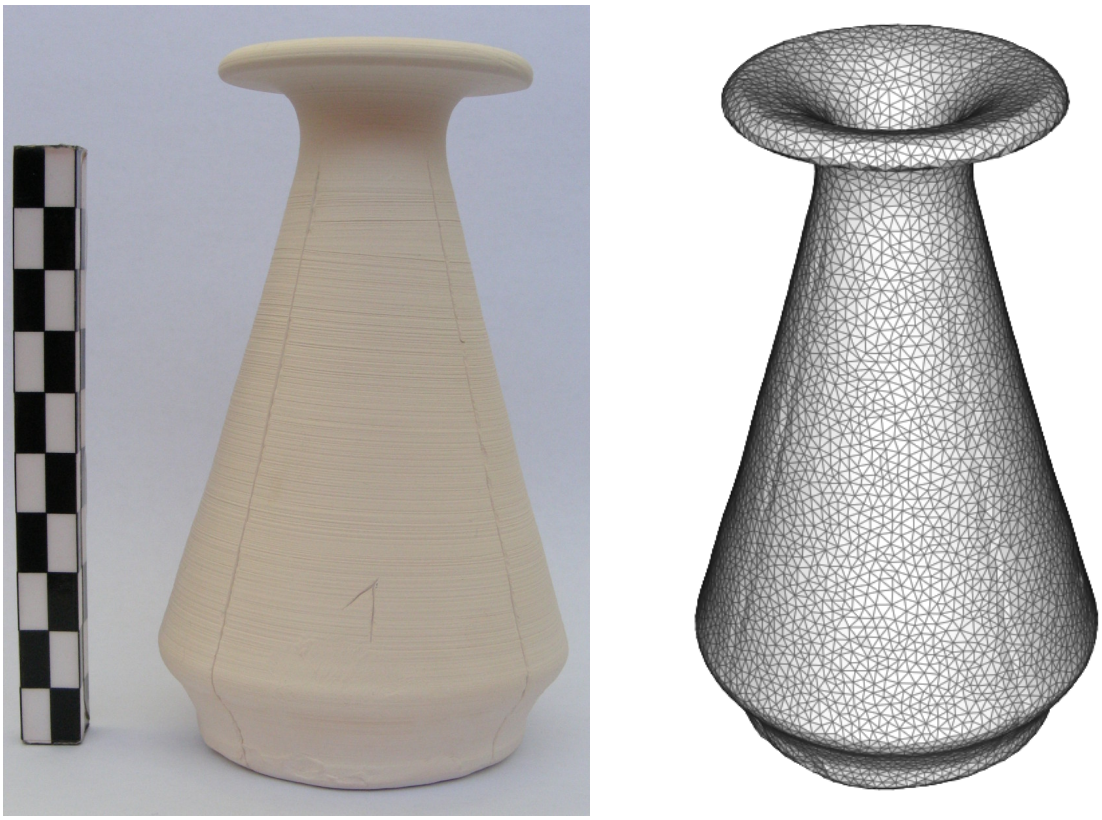


Abbildung 11.4: *Abhängigkeit des Volumens vom Isoflächen-Schwellwerts des Marching-Cubes-Algorithmus. Der exakte Wert ist durch die grüne Gerade dargestellt.*



(a) **(b)**

Abbildung 11.5: *Modernes Testgefäß zur Bestimmung des relativen Fehlers in der Volumenbestimmung. (a) Alabastron. (b) Rekonstruktion.*

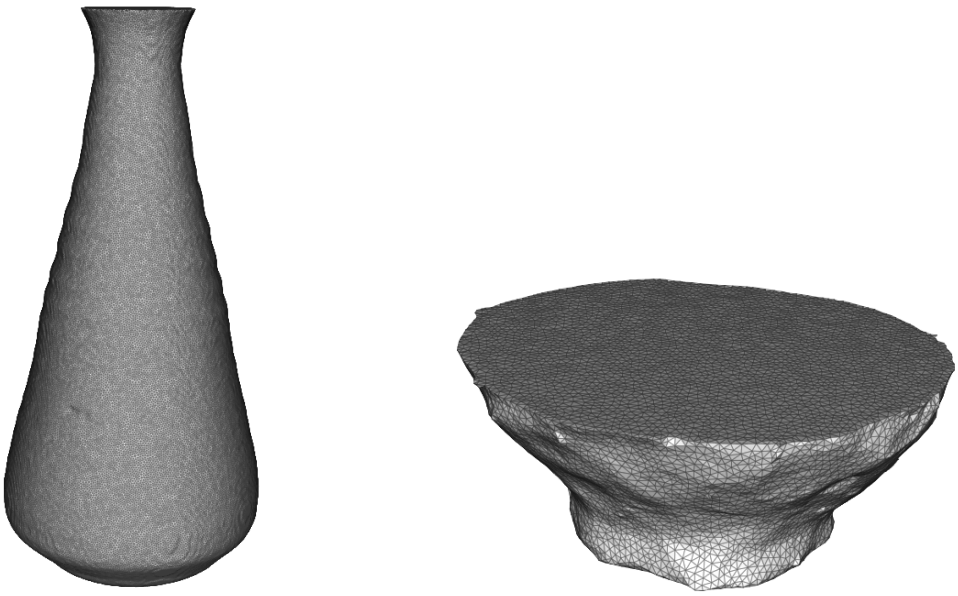
(a) **(b)**

Abbildung 11.6: *Modernes Testgefäß zur Bestimmung des relativen Fehlers in der Volumenbestimmung. (a) Handgetöpferte Schale. (b) Rekonstruktion.*

11.4 Volumenbestimmung in der Archäologie

Die hier vorgestellte Technik zur Oberflächenrekonstruktion und anschließender Volumenbestimmung kann dazu verwendet werden das Füllvolumen (Kapazität) von Keramikobjekten zu berechnen. In der Archäologie wird üblicherweise das Volumen solcher Gefäße bestimmt, indem Sand, getrocknete Erbsen, Linsen, Styroporkügelchen oder Reiskörner in die Gefäße geschüttet werden und anschließend das Volumen des verwendeten Materials gemessen wird [13]. Ein Fehler von mehreren Prozent ist hierbei durchaus üblich [13]. Eine Volumenbestimmung durch das Einfüllen von Flüssigkeiten ist zwar genauer, kann die antiken Gefäße allerdings beschädigen und wird aus konservatorischen Gründen in einem Museum strikt abgelehnt. Um abzuschätzen wie genau die hier vorgestellte *CT-Methode* ist, wurden zwei moderne Gefäße (Abbildungen 11.5a und 11.6a) mit Hilfe eines Computertomographen gescannt [56] und rekonstruiert (Abbildungen 11.5b und 11.6b). Die Rekonstruktionen sind mit *NeuRA2* erstellt, wobei folgende Operatoren angewandt wurden:

1. Medianfilterung mit Nachbarschaftsgröße 1
2. Regionenwachstum-Segmentierung mit datenabhängigem Schwellwert
3. Marching-Cubes-Gittergenerierung
4. Gitterzerlegung und Gitteroptimierung



(a) **(b)**

Abbildung 11.7: *Rekonstruktion des Inneren der Testgefäße.*

Bei der Regionenwachstum-Segmentierung muss ein Parameter eingestellt werden. Da die Kanten der Objekte über den Grauwertgradienten detektiert werden, ist diese Segmentierung gegenüber der Parametereinstellung allerdings sehr robust (vgl. Kapitel 7.4). Zur Berechnung der Kapazität der Gefäße wird das segmentierte Gefäß in der Höhe, in der das Volumen berechnet werden soll, mit einer Ebene abgeschlossen. Das Innere des Gefäßes wird anschließend segmentiert und rekonstruiert (Abbildung 11.7) und das zugehörige Volumen berechnet.

11.4.1 Genauigkeit des Verfahrens

Zur Untersuchung der Genauigkeit des hier beschriebenen Verfahrens wurde die Kapazität der Testgefäße auf zwei unterschiedliche Weisen gemessen: Das Füllvolumen wurde sowohl durch das Befüllen der Gefäße mit Wasser als auch durch das Befüllen mit Reis bestimmt. Das Alabastron wurde dabei nicht komplett befüllt, was bei der Berechnung des Volumens zu beachten ist. Die Schale wurde komplett befüllt, was vor allem bei der Befüllung mit Reis fehleranfällig ist, da die Öffnung der Schale sehr groß ist. Beide Messungen wurden mehrfach durchgeführt und anschließend der Mittelwert gebildet. Der durch die Wassermethode bestimmte Wert kann somit als exaktes Volumen angenommen werden. Tabelle 11.4 zeigt, dass die hier vorgestellte *CT-Methode* zur Berechnung von Volumina sehr genau ist und bessere Werte als die ungenaue Reismethode [13] liefert.

Objekt	**Wassermethode**	**Reismethode**	**CT-Methode**
Alabastron	92ml	92ml	92ml
Schale	65ml	67ml	65ml

Tabelle 11.4: *Volumenbestimmung der Testgefäße.*

11.4.2 Einfluss der Füllhöhe

Das Einstellen der Füllhöhe nimmt einen signifikanten Einfluss auf die berechnete Kapazität, was am Beispiel des Alabastrons systematisch untersucht wurde. Die Füllhöhe kann mit einer Genauigkeit eingestellt werden, die der Auflösung eines Voxels entspricht, was bei dem Alabastron 0.27mm entspricht. Im Schaubild 11.8 ist die berechnete Kapazität des Alabastrons in Abhängigkeit der Füllhöhe dargestellt, wobei die Füllhöhe zwischen komplett leer und komplett voll variiert wird. Die grüne Linie markiert das mit der Wassermethode bestimmte Füllvolumen des Alabastrons. Im Bereich dieser Markierung ändert sich das berechnete Füllvolumen um etwa 0.05ml bei einer Füllhöhenänderung von einem Voxel. Verglichen mit der Genauigkeit der Volumenbestimmung mit Hilfe von Wasser oder Reis ist damit eine hohe Robustheit des CT-Verfahrens gegenüber Veränderungen der Eingabeparameter gewährleistet. Eine systematische Volumen- und Dichteanalyse von den im Rahmen dieser Arbeit rekonstruierten Keramikgefäßen findet sich in Kapitel 14.3.

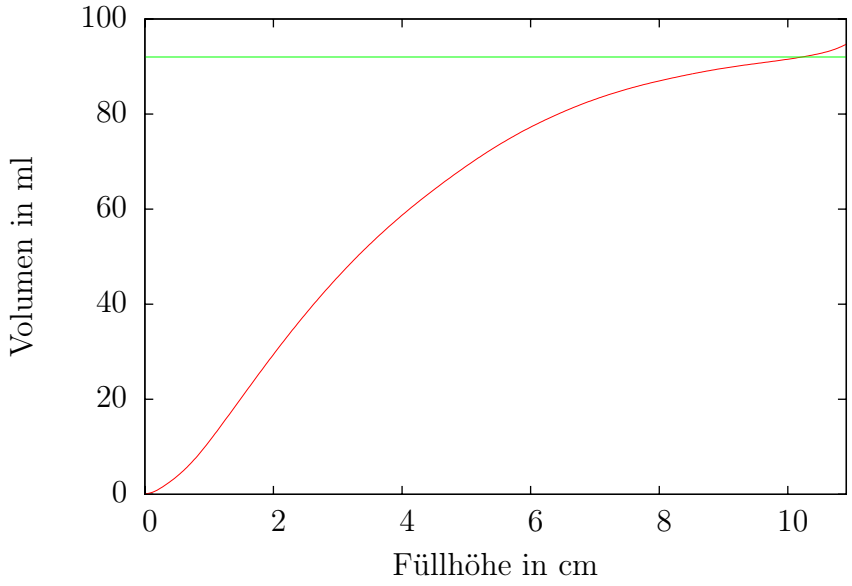


Abbildung 11.8: *Kapazität des Alabastrons in Abhängigkeit der Füllhöhe. Der exakte Wert ist durch die grüne Gerade dargestellt.*

11.4.3 Einfluss der Auflösung

Darüber hinaus stellt sich die Frage, wie fein die Auflösung der CT-Scans sein muss, damit das Volumen mit Hilfe des CT-Verfahrens hinreichend exakt bestimmt werden kann. Alle vorliegenden Scans wurden mit einer isotropen Auflösung zwischen 0.15mm und 0.28mm pro Voxel aufgenommen (vgl. Kapitel 14.3 und [56]). Das Alabastron wurde mit einer isotropen Auflösung von 0.27mm pro Voxel und die Schale mit einer isotropen Auflösung von 0.2mm pro Voxel gescannt. Die 512^3 Voxel großen Bilder wurden auf die Größen 256^3, 128^3 usw. skaliert, rekonstruiert und die Kapazität der rekonstruierten Gefäße berechnet (Abbildung 11.9). Es zeigt sich, dass die berechnete Kapazität bei gröberen Auflösungen geringer wird, und bei Auflösungen, die feiner als 2mm pro Voxel sind, näher als 5% am exakten Wert liegt, was im Rahmen solcher Untersuchungen hinreichend exakt ist [13]. Die in dieser Arbeit betrachteten CT-Aufnahmen sind in jede Raumrichtung 7- bis 13-mal besser aufgelöst und damit hinreichend genau um die Kapazitäten der darin befindlichen Objekte sehr exakt zu berechnen.

11.4.4 Einfluss der Gitteroptimierung

In Kapitel 11.3.2 ist gezeigt, dass die Verwendung des Gitteroptimierungs-Algorithmus kaum einen Einfluss auf die berechnete Oberfläche und das berechnete Volumen einer Kugel hat. Um dieses Ergebnis zu untermauern, wurden die für das Füllvolumen der Testgefäße berechneten Gitter entsprechend optimiert und die Volumina aller Gitter der beiden Gitterhierarchien berechnet (Tabellen 11.5 und 11.6, Abbildung 11.10 und Anhang D).

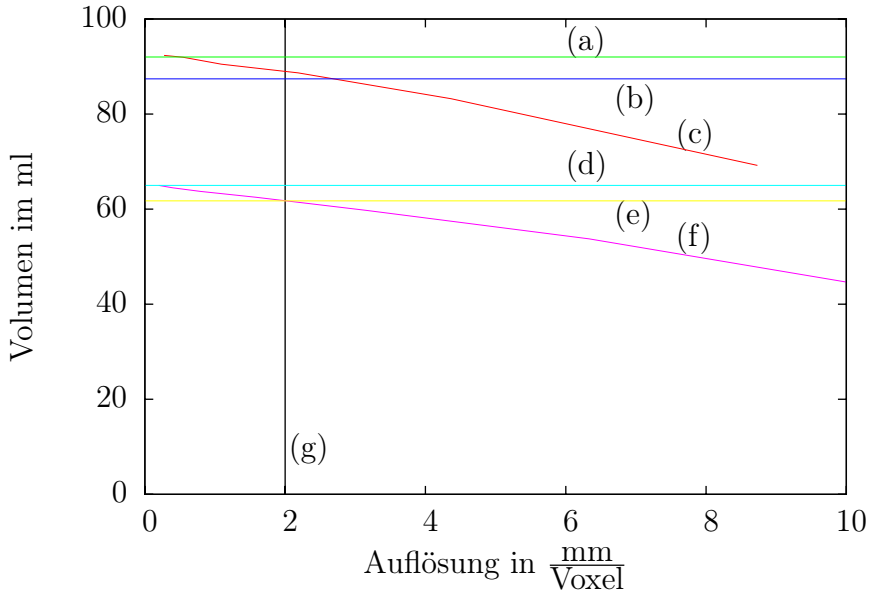


Abbildung 11.9: *Kapazität der Testgefäße in Abhängigkeit der Auflösung. (a) Exaktes Volumen V_A des Alabastrons. (b) 5%ige Abweichung von V_A. (c) Volumen des Alabastrons in Abhängigkeit der Auflösung. (d) Exaktes Volumen V_S der Schale. (e) 5%ige Abweichung von V_S. (f) Volumen der Schale in Abhängigkeit der Auflösung. (g) Auflösung von 2mm pro Voxel.*

Hierbei bestätigt sich das Ergebnis aus Kapitel 11.3.2. In Abbildung 11.11 ist gezeigt, weshalb schon grobe Gitter das Volumen eines Objekts sehr gut approximieren: In rot ist das Innere des Alabastrons bestehend aus 18380 Dreiecken und in grün ist das Innere des Alabastrons bestehend aus 1676 Dreiecken dargestellt. Da der Gitteroptimierungs-Algorithmus die Oberfläche des ursprünglichen Gitters sehr exakt approximiert, zeigt sich, dass sich die rote und die grüne Fläche häufig schneiden. Teilweise liegt das grobe Gitter somit außerhalb des feinen Gitters und teilweise liegt das grobe Gitter innerhalb des feinen Gitters, wodurch sich Fehler in der Volumenberechnung großteils ausgleichen.

Anzahl Dreiecke	**Volumen in ml**	**Abweichung**
455924	92.035	0%
232270	92.036	0.0005%
18380	92.059	0.025%
1676	92.072	0.04%
318	92.095	0.065%

Tabelle 11.5: *Einfluss der Gitteroptimierung – Alabastron.*

Anzahl Dreiecke	Volumen in ml	Abweichung
726512	64.996	0%
347870	64.944	0.08%
34288	64.954	0.065%
2788	64.962	0.051%
362	64.986	0.015%

Tabelle 11.6: *Einfluss der Gitteroptimierung – Schale.*

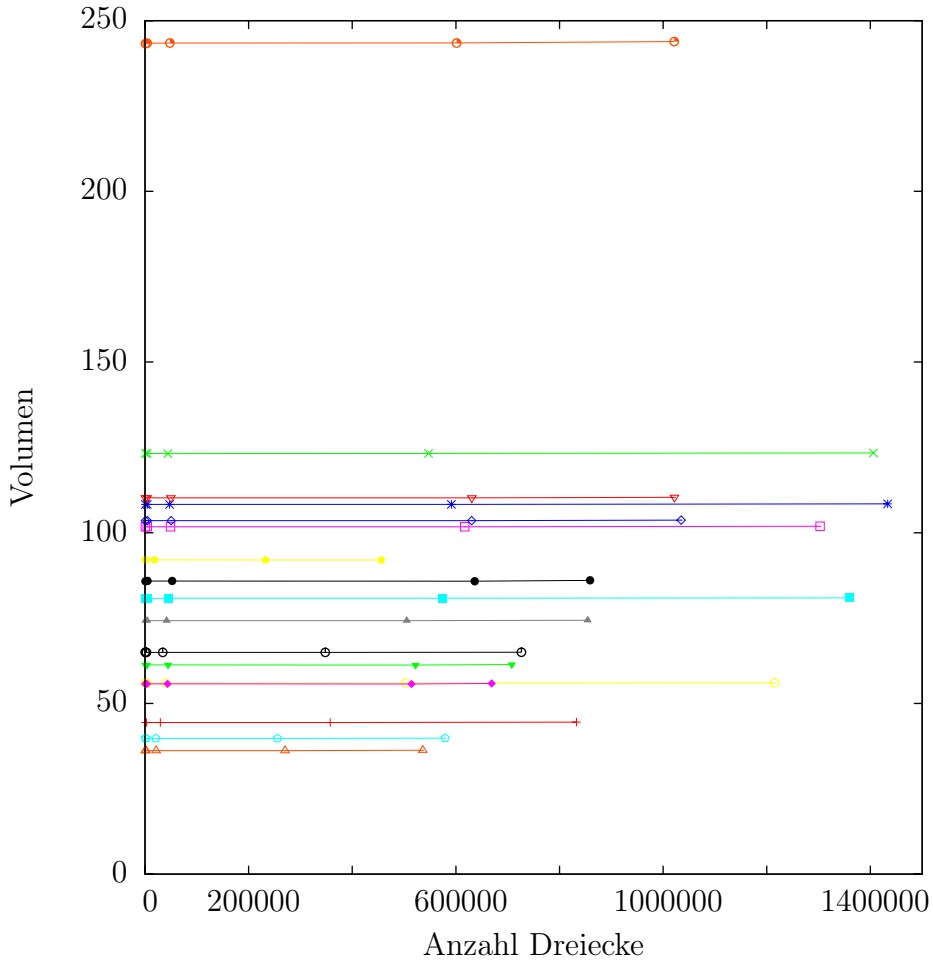


Abbildung 11.10: *Das Volumen der rekonstruierten Keramikgefäße in Abhängigkeit der Gitterauflösung. Es zeigt sich, dass der Gitteroptimierungs-Algorithmus keinen Einfluss auf das berechnete Volumen hat. Jede Gerade stellt das Volumen eines rekonstruierten Gefäßes dar. Die Tabellen mit den zugehörigen Werten finden sich in Anhang D.*

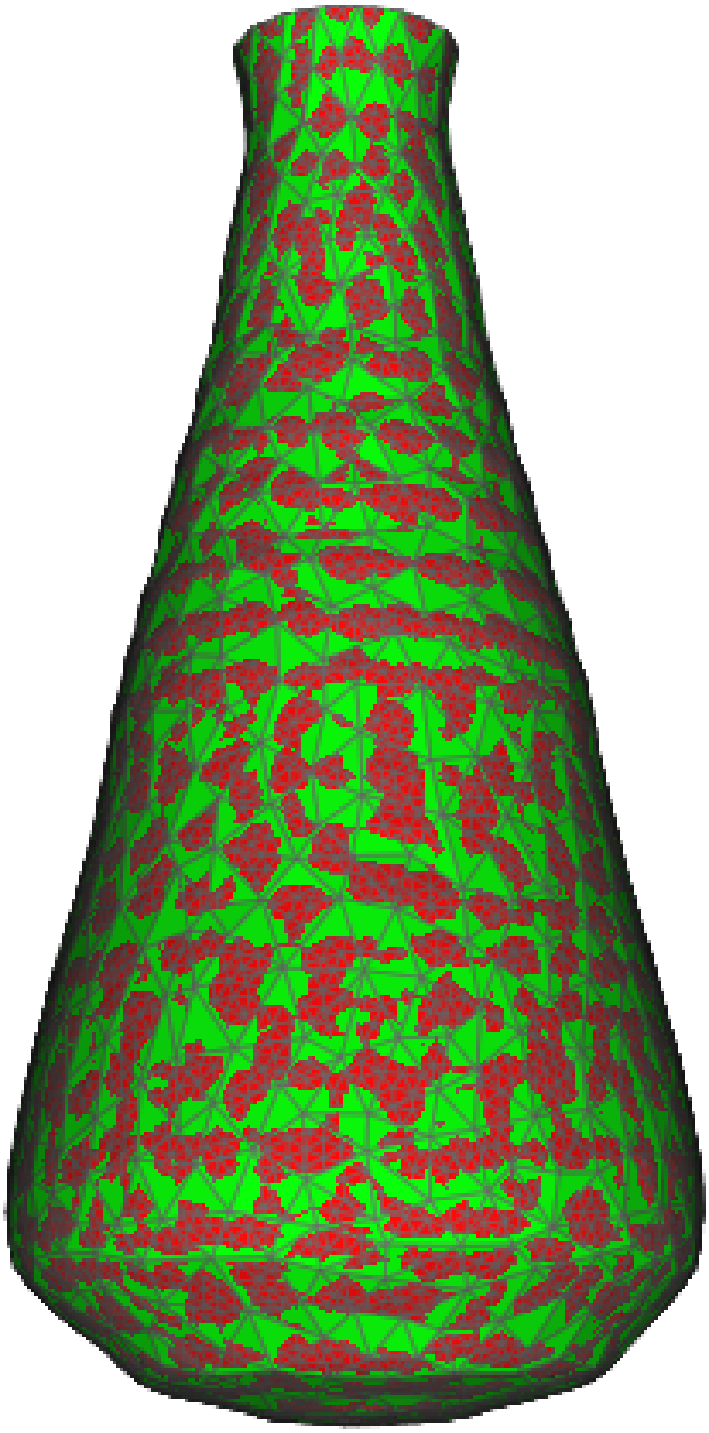


Abbildung 11.11: *Das Innere des Alabastrons approxmiert durch* 18380 *Dreiecke (rot) und* 1676 *Dreiecke (grün). Da sich beide Oberflächen häufig schneiden, gleichen sich Fehler in der Volumenberechnung großteils aus.*

Kapitel 12

Krümmung

Nachdem im vorherigen Kapitel gezeigt wurde, wie Oberflächen und Volumina von rekonstruierten Gittern berechnet werden können, stellt sich die Frage, wie exakt die Krümmung von Oberflächen durch Dreiecksgitter approximiert werden kann. Hierzu werden die aus der Differentialgeometrie [9] bekannten Begriffe *Gaußsche Krümmung* und *mittlere Krümmung* für *reguläre Flächen* eingeführt und [29] folgend auf Dreiecksgeometrien übertragen, für die sich diskrete Formeln zur Berechnung der lokalen Krümmung herleiten lassen. Anschließend ist gezeigt, dass die diskrete Gaußsche Krümmung einer *lokal homogen triangulierten* Einheitskugel quadratisch in der maximalen lokalen Maschenweite h gegen die exakte Krümmung der Einheitskugel konvergiert. Diese Aussage wird durch in *Double-Precision-Arithmetik* durchgeführte numerische Experimente bestätigt, die darüber hinaus zeigen, dass die diskrete mittlere Krümmung ebenfalls quadratisch in der maximalen lokalen Maschenweite gegen die exakte mittlere Krümmung konvergiert. Weitere numerische Experimente in der für Oberflächengitter üblichen *Single-Precision-Arithmetik* ergeben, dass eine Kugel mit Radius r durch mindestens 3000 Dreiecken approximiert werden muss, damit die Abweichung der diskreten Gaußschen Krümmung und der diskreten mittleren Krümmung maximal ein Prozent von den entsprechenden exakten Werten beträgt. Dies entspricht einer maximalen Gittermaschenweite von $h = \frac{r}{10}$.

12.1 Kontinuierliche Krümmung

Die Krümmung einer Fläche wird auf einer sogenannten *regulären Fläche* S definiert, die sich dadurch auszeichnet, dass an jedem ihrer Punkte eine Tangentialebene angelegt werden kann.

Definition 12.1 (Reguläre Fläche)
Eine Teilmenge $S \subset \mathbb{R}^3$ heißt reguläre Fläche*, falls für jeden Punkt $p \in S$ eine Umgebung $V \subset \mathbb{R}^3$, eine offene Menge $U \subset \mathbb{R}^2$ und ein stetig differenzierbarer Homöomorphismus $\phi : U \to V \cap S$ existieren, sodass für jeden Punkt $q \in U$ das Differential $\frac{d\phi}{dx}(q) : \mathbb{R}^2 \to \mathbb{R}^3$ injektiv ist.*

Bemerkung 12.2 (Parametrisierung)
Die Abbildung ϕ heißt Parametrisierung. *Die Forderung an das Differential $\frac{d\phi}{dx}(q)$ stellt sicher, dass an jedem Punkt der Fläche ein Tangentialvektor angeheftet werden kann. Insbesondere ist jede reguläre Fläche eine differenzierbare Untermannigfaltigkeit [58].*

Betrachtet man nun in einem Punkt $p \in S$ den Schnitt, der von dem Normalenvektor von S in p und einem Tangentialvektor von S in p aufgespannten Ebene mit S, so ergibt sich eine ebene Kurve, deren Krümmung mit den üblichen Methoden der Analysis [58] bestimmt werden kann und als *Normalkrümmung* bezeichnet wird.

Definition 12.3 (Normalkrümmung)
Sei $S \subset \mathbb{R}^3$ eine reguläre Fläche, n_p der Normalenvektor von S im Punkt $p \in S$ und t_p ein beliebiger Tangentialvektor in p an S. Die Normalkrümmung *ist definiert, als die Krümmung der ebenen Kurve, die durch den Schnitt von S mit der von den beiden Vektoren n_p und t_p aufgespannten Ebene durch den Punkt p entsteht.*

Auf Basis der minimalen und maximalen Normalkrümmungen k_1 und k_2 in einem Punkt p, den sogenannten *Hauptkrümmungen*, werden die *Gaußsche* Krümmung als Produkt von k_1 und k_2 und die mittlere Krümmung als arithmetisches Mittel von k_1 und k_2 definiert. Sie bilden die wichtigsten Krümmungsbegriffe der Differentialgeometrie. Zum Studium weiterer Krümmungsbegriffe oder für äquivalente Definitionen der hier eingeführten Krümmungen sei auf [9] verwiesen. Ein weiterer wichtiger Begriff ist der *Krümmungsradius*, der dem Radius des Kreises entspricht, der eine Kurve lokal optimal approximiert und dem Kehrwert der Normalkrümmung entspricht.

Definition 12.4 (Hauptkrümmung)
Der Minimalwert k_1 und der Maximalwert k_2 aller möglichen Normalkrümmungen im Punkt $p \in S$ heißen Hauptkrümmungen.

Definition 12.5 (Krümmungsradius)
Die Kehrwerte der Hauptkrümmungen werden als Krümmungsradien $r_1 = \frac{1}{k_1}$ *und $r_2 = \frac{1}{k_2}$ bezeichnet.*

Definition 12.6 (Gaußsche Krümmung)
Die Gaußsche Krümmung *K im Punkt p einer regulären Fläche $S \subset \mathbb{R}^3$ ist definiert als Produkt der beiden Hauptkrümmungen*

$$K = k_1 \cdot k_2 = \frac{1}{r_1} \cdot \frac{1}{r_2}. \tag{12.1}$$

Definition 12.7 (Mittlere Krümmung)
Die mittlere Krümmung *H im Punkt p einer regulären Fläche $S \subset \mathbb{R}^3$ ist definiert als arithmetisches Mittel der beiden Hauptkrümmungen*

$$H = \frac{1}{2}(k_1 + k_2) = \frac{1}{2}\left(\frac{1}{r_1} + \frac{1}{r_2}\right). \tag{12.2}$$

Bemerkung 12.8 (Krümmung einer Kugeloberfläche)
Die Gaußsche und die mittlere Krümmung der Oberfläche einer Kugel mit Radius r sind gegeben durch

$$K = \frac{1}{r^2} \tag{12.3}$$

und

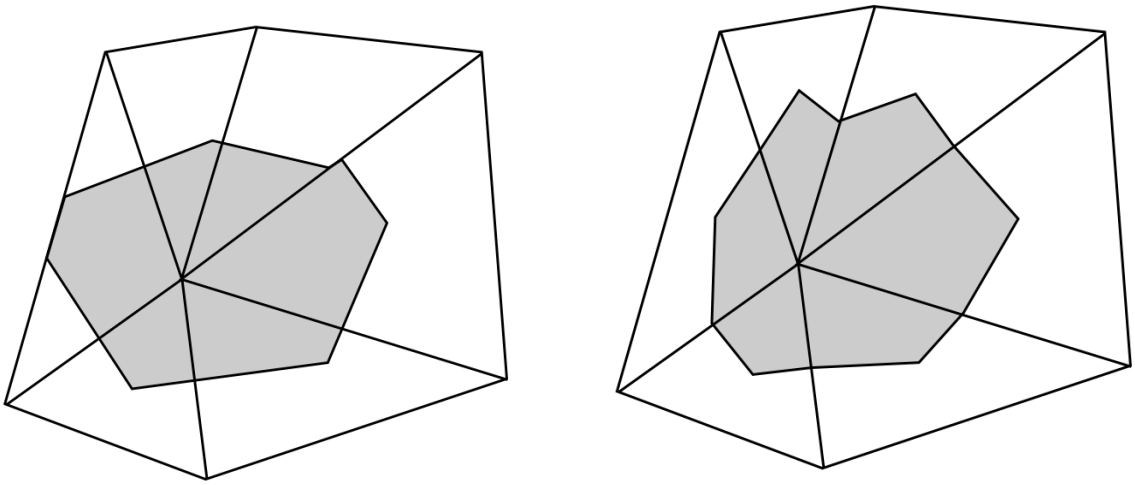
$$H = \frac{1}{r}, \tag{12.4}$$

da für die Hauptkrümmungen k_1 und k_2 einer Kugel mit Radius r in jedem Punkt

$$k_1 = k_2 = \frac{1}{r} \tag{12.5}$$

gilt.

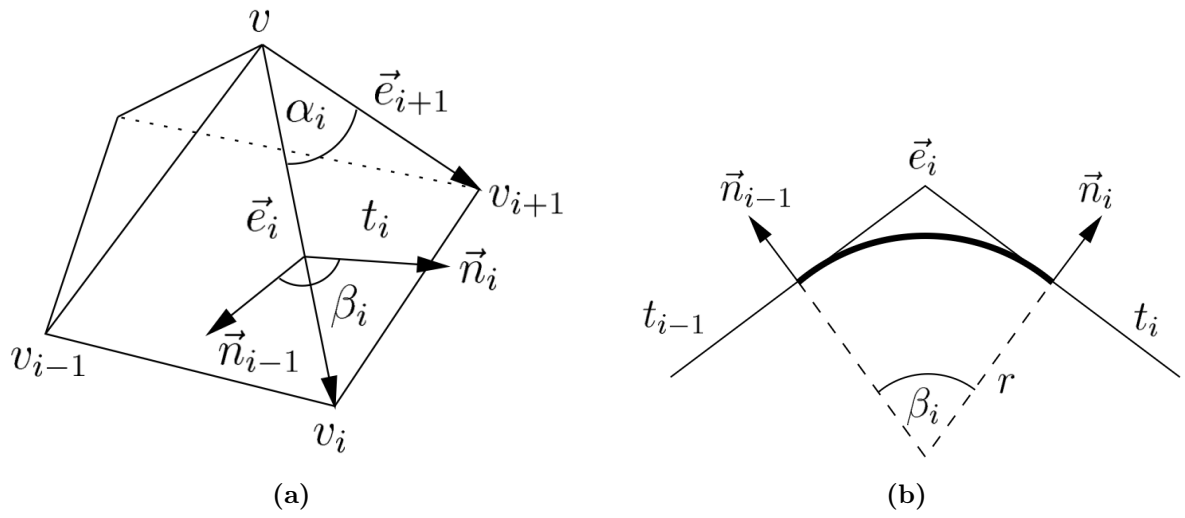
12.2 Diskrete Krümmung



(a) **(b)**

Abbildung 12.1: *(a) Voronoi-Fläche S_V um einen Vertex v. (b) Baryzentrische Fläche S_B um einen Vertex v. Quelle:* [29]

Um die diskrete Krümmung eines Dreiecksgitters zu berechnen, betrachtet man dieses als diskreten Repräsentanten einer regulären Fläche im $\mathbb{R}^3$. Hierzu stelle man sich die nicht differenzierbaren Kanten des Gitters als Zylinder und die Ecken als Kugeln mit jeweils kleinem Radius r vor, welche die einzelnen Seitenflächen glatt miteinander verbinden. Berechnet man das Integral der Krümmung über die Voronoi-Fläche S_V (Abbildung 12.1a) um den Vertex v, so ergeben sich nach [3, 26] die Formeln



(a) **(b)**

Abbildung 12.2: *(a) Vertex v und seine Nachbarschaft. (b) Winkel β_i zwischen den Normalen zweier benachbarter Dreiecke. Quelle:* [29]

$$\bar{K}_h = \int_S K \, dS = 2\pi - \sum_{i=1}^{n} \alpha_i \tag{12.6}$$

und

$$\left|\bar{H}_h\right| = \int_S dS \, |H| = \frac{1}{4} \sum_{i=1}^{n} \|\vec{e_i}\| \, |\beta_i| \tag{12.7}$$

zur Berechnung der integralen Gaußschen Krümmung $\bar{K}$ und der integralen mittleren Krümmung $\left|\bar{H}\right|$. Hierbei ist der Vertex v Eckpunkt von n Dreiecken t_i mit den Normalen $\vec{n}_i$ und ist über jeweils eine Dreieckskante $\vec{e}_i$ mit den Vertices v_i verbunden. Die für die diskrete Gaußsche Krümmung benötigten Winkel α_i bezeichnen jeweils die Winkel zwischen den Kanten $\vec{e}_i$ und $\vec{e}_{i+1}$ (Abbildung 12.2a) und die für die diskrete mittlere Krümmung benötigten Winkel β_i bezeichnen die Winkel zwischen den Flächennormalen $\vec{n}_{i-1}$ und $\vec{n}_i$ (Abbildung 12.2b). In diesen Definitionen wird der Index $n+1$ mit 1 und der Index 0 mit n identifiziert. Die diskrete Gaußsche Krümmung K_h und die diskrete mittlere Krümmung H_h im Vertex v ergeben sich durch die Quotienten

$$K_h = \frac{\bar{K}_h}{S_V} \tag{12.8}$$

und

$$|H_h| = \frac{\left|\bar{H}_h\right|}{S_V}. \tag{12.9}$$

Alternativ können die Integrale über die baryzentrische Fläche (Abbildung 12.1b) berechnet

werden, die gerade einem Drittel der Fläche aller angrenzenden Dreiecke von v entspricht. Dabei müssen die Formeln (12.6) und (12.7) entsprechend angepasst werden [29]. Beide Flächendefinitionen erfüllen die Bedingung, dass die Summe der Flächen aller Vertices genau die Oberfläche des Dreiecksgitters ergibt.

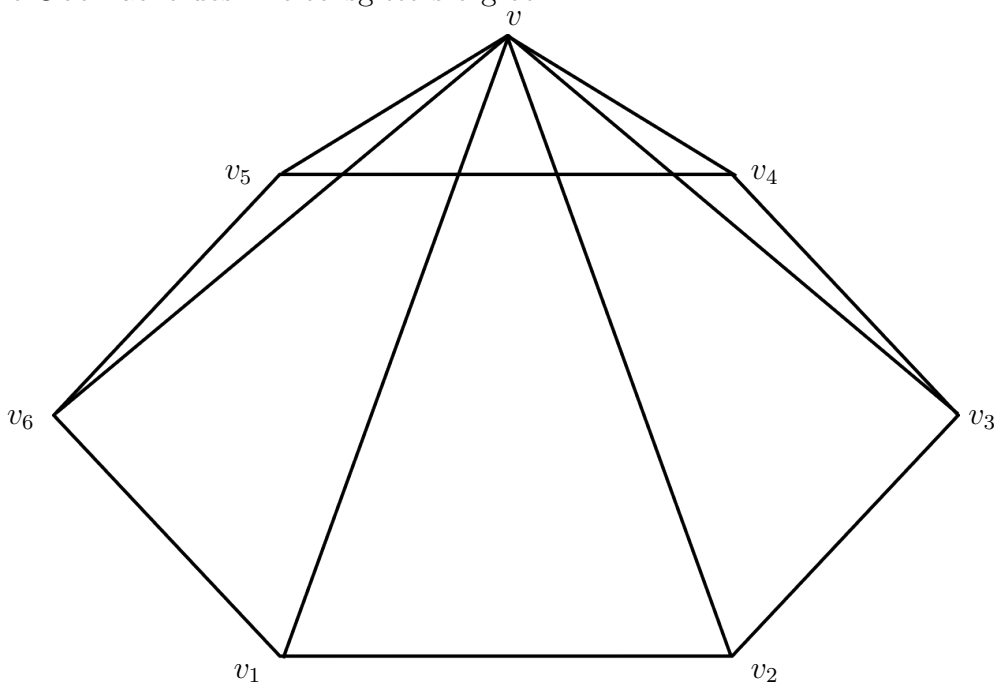


Abbildung 12.3: *Lokal homogene Triangulierung.*

Definition 12.9 (Lokal homogene Triangulierung)
Die Triangulierung einer Kugel heißt lokal homogen im Vertex v*, falls genau sechs gleiche gleichschenklige Dreiecke in v zusammenlaufen, sodass alle an v anliegenden Kanten die Länge h haben und v und die Nachbarvertices v_i exakt auf der Kugeloberfläche liegen (Abbildung 12.3).*

Satz 12.10 (Konvergenz der diskreten Gaußschen Krümmung)
Sei B_1 die Einheitskugel, die im Punkt v durch eine lokal homogene Triangulierung durch ein Dreiecksgitter mit Maschenweite h approximiert ist. Dann konvergiert die in Formel (12.8) *definierte diskrete Gaußsche Krümmung K_h quadratisch in h gegen die kontinuierliche Gaußsche Krümmung K.*

Beweis: Nach Bemerkung 12.8 gilt für die kontinuierliche Krümmung der Einheitskugel in jedem Punkt ihrer Oberfläche $K = 1$. Aufgrund der Symmetrie der lokal homogenen Triangulierung genügt es den Winkel α von einem der an v angrenzenden Dreiecke t zu bestimmen (Abbildung 12.4). Das gleichschenklige Dreieck t hat zweimal die Seitenlänge h und einmal die Seitenlänge

$$k = \sqrt{h^2 - \frac{h^4}{4}}. \tag{12.10}$$

Die Höhe d des Dreiecks ist daher gegeben durch

$$d = \sqrt{h^2 - \left(\frac{k}{2}\right)^2} = \sqrt{h^2 - \frac{h^2}{4} + \frac{h^4}{16}} = \frac{1}{2}\sqrt{3h^2 + \frac{h^4}{4}}. \tag{12.11}$$

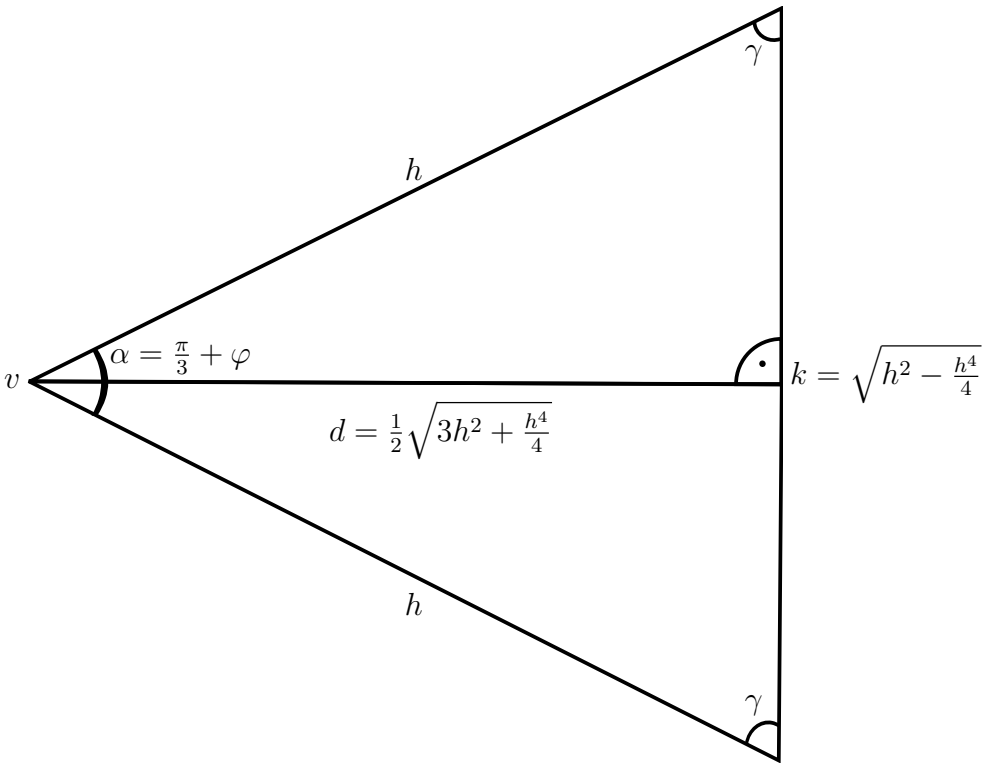


Abbildung 12.4: *Einzelnes Dreieck einer lokal homogenen Triangulierung.*

Der Winkel α weicht für kleine h nur wenig von $\frac{\pi}{3}$ ab, weshalb

$$\alpha = \frac{\pi}{3} + \varphi \tag{12.12}$$

mit $\varphi \to 0$ für $h \to 0$ gilt und folglich

$$\frac{\alpha}{2} = \frac{\pi}{6} + \frac{\varphi}{2} \tag{12.13}$$

gilt. Ferner gilt

$$\sin\left(\frac{\alpha}{2}\right) = \frac{k}{2} \cdot \frac{1}{h} = \frac{1}{2}\sqrt{1 - \frac{h^2}{4}} \tag{12.14}$$

und nach den bekannten Additionstheoremen

$$\sin\left(\frac{\alpha}{2}\right) = \sin\left(\frac{\pi}{6} + \frac{\varphi}{2}\right) = \sin\left(\frac{\pi}{6}\right)\cos\left(\frac{\varphi}{2}\right) + \cos\left(\frac{\pi}{6}\right)\sin\left(\frac{\varphi}{2}\right)$$

$$= \frac{1}{2}\cos\left(\frac{\varphi}{2}\right) + \frac{\sqrt{3}}{2}\sin\left(\frac{\varphi}{2}\right). \tag{12.15}$$

Für hinreichend kleines h und damit hinreichend kleines φ gilt nach den Taylorentwicklungen von cos und sin

$$\cos\left(\frac{\varphi}{2}\right) \approx 1 \tag{12.16}$$

und

$$\sin\left(\frac{\varphi}{2}\right) \approx \frac{\varphi}{2}. \tag{12.17}$$

Folglich gilt

$$\sin\left(\frac{\alpha}{2}\right) \approx \frac{1}{2} + \frac{\sqrt{3}}{4}\varphi \tag{12.18}$$

und es folgt mit Gleichung (12.14)

$$\varphi = \frac{2}{\sqrt{3}}\left(\sqrt{1 - \frac{h^2}{4}} - 1\right). \tag{12.19}$$

Die Voronoi-Fläche S_V für das gleichschenklige Dreieck t berechnet sich für $\varphi \leq \frac{\pi}{6}$ (d.h. t ist spitzwinklig) nach [29] durch die Formel

$$S_V = \frac{1}{4}h^2\cot(\gamma) = \frac{1}{4}h^2\frac{k}{2d} = \frac{h^2}{4}\frac{\sqrt{h^2 - \frac{h^4}{4}}}{\sqrt{3h^2 + \frac{h^4}{4}}}. \tag{12.20}$$

Für die diskrete Gaußsche Krümmung gilt daher

$$K = \frac{2\pi - \sum_{i=1}^{6}\alpha}{\sum_{i=1}^{6} S_V} = \frac{2\pi - 6(\frac{\pi}{3} + \varphi)}{6 \cdot S_V} = -\frac{\varphi}{S_V}$$

$$= -\frac{\frac{2}{\sqrt{3}}\left(\sqrt{1 - \frac{h^2}{4}} - 1\right)}{\frac{h^2}{4}\frac{\sqrt{h^2 - \frac{h^4}{4}}}{\sqrt{3h^2 + \frac{h^4}{4}}}} = \frac{8}{\sqrt{3}} \cdot \frac{1 - \sqrt{1 - \frac{h^2}{4}} \cdot \sqrt{3h^2 + \frac{h^4}{4}}}{h^2 \cdot \sqrt{h^2 - \frac{h^4}{4}}}. \tag{12.21}$$

Die entsprechende Taylorentwicklung von K im Punkt $h = 0$ liefert

$$K = 1 + \frac{11}{48}h^2 + O(h^4) \tag{12.22}$$

und somit quadratische Konvergenz. □

12.3 Numerische Experimente

12.3.1 Modellgeometrie in doppelter Genauigkeit

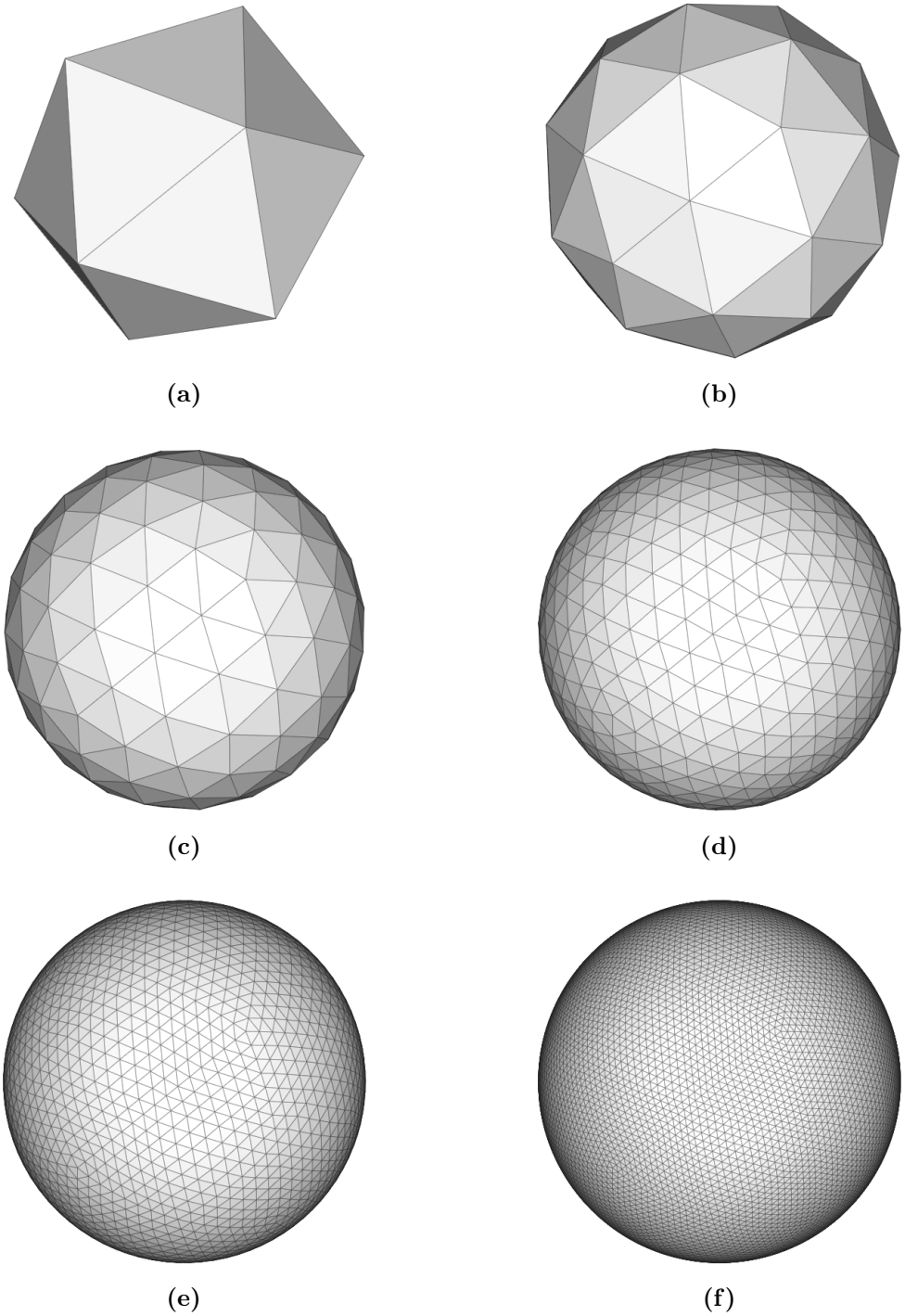
Gitter	Anzahl Vertices	Anzahl Dreiecke
(a)	12	20
(b)	42	80
(c)	162	320
(d)	642	1280
(e)	2562	5120
(f)	10242	20480
(g)	40962	81920
(h)	163842	327680
(i)	655362	1310720
(j)	2621442	5242880

Tabelle 12.1: *Anzahl Vertices und Dreiecke der einzelnen Gitter.*

Das Ergebnis von Satz 12.10 lässt sich durch numerische Experimente bestätigen. Ausgehend von einem Ikosaeder, dessen Eckpunkte auf der Einheitskugel liegen, wird nach dem Subdivisions-Schema von *Loop* [59, 120] eine immer feinere Gitterhierarchie erzeugt, deren neu generierten Gitterpunkte jeweils auf die Einheitskugel projiziert werden. Die Triangulierungen dieser Gitter sind näherungsweise lokal homogen und weisen nur wenige Vertices mit fünf angrenzenden Dreiecken auf. Für die weiteren Experimente wurden die Tabelle 12.1 bzw. Abbildung 12.5 zu entnehmenden Gitter verwendet. Hierbei entspricht Gitter (a) dem Ikosaeder. Alle Gitterpunkte eines Gitters l sind dabei in der nächsten Verfeinerungsstufe $l+1$ enthalten. Es wurde die diskrete Krümmung in jedem der Gitterpunkte v berechnet und gegen das Maximum der Maschenweite h aller von v benachbarten Dreiecke aufgetragen (Abbildung 12.6). Diese Testrechnungen wurden in *Double-Precision-Arithmetik* durchgeführt. Das gleiche numerische Experiment lässt sich für die diskrete mittlere Krümmung durchführen, bei der sich dasselbe Konvergenzverhalten zeigt (Abbildung 12.7).

12.3.2 Realistische Geometrie in einfacher Genauigkeit

Datenformate zum Speichern von Dreiecksgittern unterstützen üblicherweise nur *Single-Precision-Arithmetik*. Zudem werden die in dieser Arbeit betrachteten Rekonstruktionen durch die *Gitteroptimierung nach Hoppe* (Kapitel 10) erzeugt. Für die weiteren Testrechnungen sei daher auf die in Kapitel 11.3.2 verwendeten Testgeometrien der Einheitskugel zurückgegriffen, die durch den Gitteroptimierungs-Algorithmus erzeugt wurden. Für alle Vertices v wurde die Abweichung der diskreten Gaußschen Krümmung und der mittleren



(a) **(b)**

(c) **(d)**

(e) **(f)**

Abbildung 12.5: *(a) Ikosaeder bestehend aus 12 Vertices und 20 Dreiecken. (b)-(f) Verfeinerungen nach dem Subdivisions-Schema von* Loop. *(b) 42 Vertices und 80 Dreiecke. (c) 162 Vertices und 320 Dreiecke. (d) 642 Vertices und 1280 Dreiecke. (e) 2562 Vertices und 5120 Dreiecke. (f) 10242 Vertices und 20480 Dreiecke.*

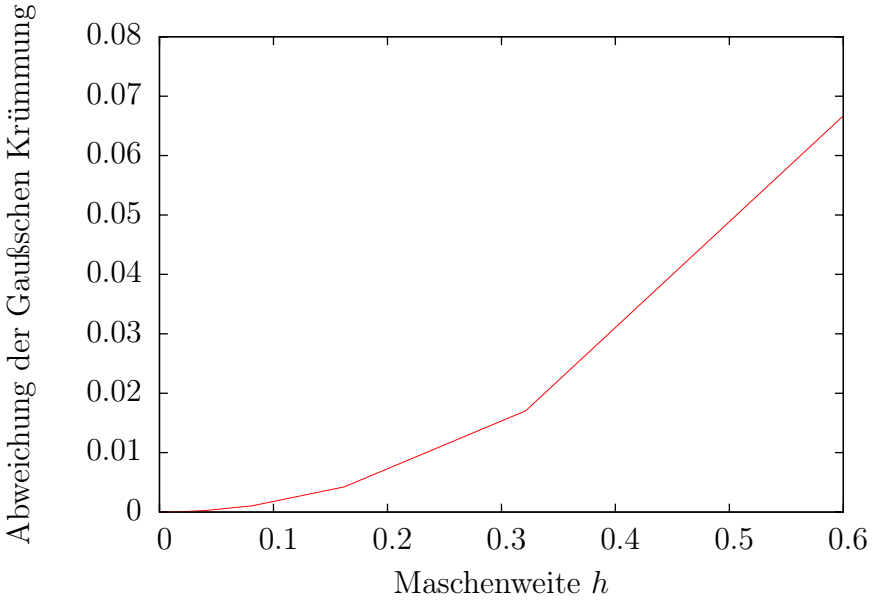


Abbildung 12.6: *Die Abweichung der diskreten Gaußschen Krümmung konvergiert in Abhängigkeit der lokalen Maschenweite h quadratisch gegen* 0.

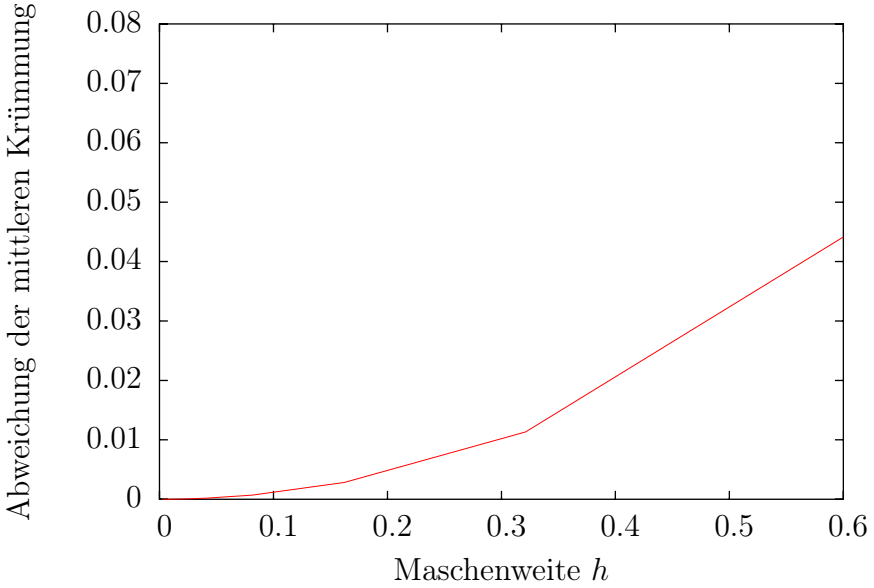


Abbildung 12.7: *Die Abweichung der diskreten mittleren Krümmung konvergiert in Abhängigkeit der lokalen Maschenweite h ebenfalls quadratisch gegen* 0.

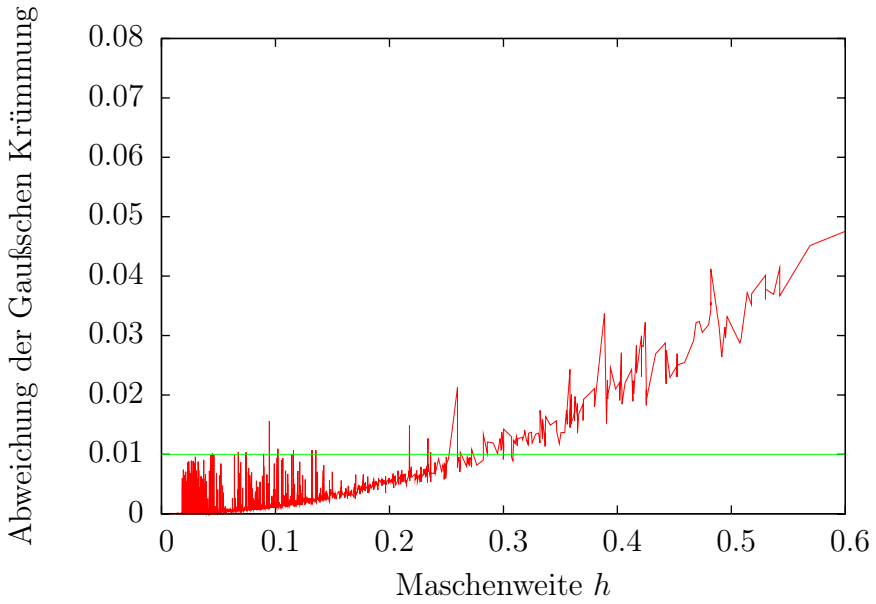


Abbildung 12.8: *Die Abweichung der diskreten Gaußschen Krümmung konvergiert in Abhängigkeit der lokalen Maschenweite h quadratisch gegen 0. Alle Werte unterhalb der grünen Linie weichen weniger als ein Prozent von der exakten Krümmung ab.*

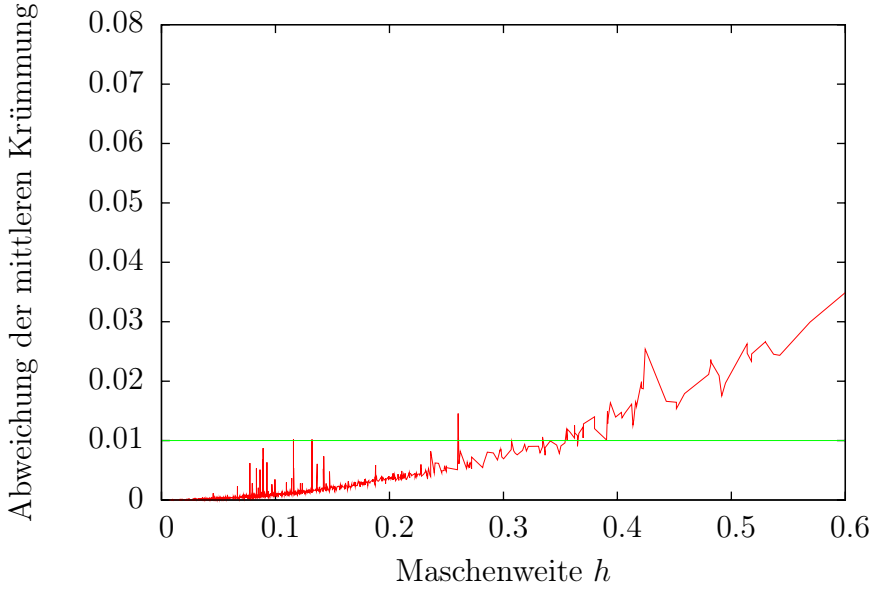


Abbildung 12.9: *Die Abweichung der diskreten mittleren Krümmung konvergiert in Abhängigkeit der lokalen Maschenweite h ebenfalls quadratisch gegen 0. Alle Werte unterhalb der grünen Linie weichen weniger als ein Prozent von der exakten Krümmung ab.*

Krümmung von der exakten Krümmung sowie die maximale Maschenweite h aller benachbarten Dreiecke von v berechnet (Abbildungen 12.8 und 12.9). Obwohl die Testrechnungen statistisch verteilte Abweichungen aufweisen, ist die quadratische Konvergenz gut zu erkennen. Darüber hinaus ergibt sich unter Berücksichtigung der statistischen Ausreißer für lokale Triangulierungen mit der Maschenweite $h \leq 0.1$ eine Abweichung von maximal einem Prozent gegenüber der exakten Krümmung der Kugeloberfläche. Der Wert $h \leq 0.1$ bezieht sich auf die Einheitskugel. Um die Krümmung der Oberfläche eines kugelförmigen Objekts mit beliebigem Radius r mit einer Abweichung von höchstens einem Prozent anzunähern, muss folglich die maximale Maschenweite des Gitters mit

$$h = 0.1 \cdot r \tag{12.23}$$

fixiert werden. Unter Verwendung von Gleichung (12.22) würde sogar $h = 0.2 \cdot r$ ausreichen, um die Krümmung einer Oberfläche mit einem maximalen Fehler von einem Prozent zu approximieren, allerdings setzt die Herleitung dieser Gleichung eine lokal homogene Triangulierung, sowie eine exakte Arithmetik voraus. Unter Verwendung von gleichseitigen Dreiecken Δ mit Seitenlänge h und Fläche

$$A_\Delta = \frac{\sqrt{3}}{4} h^2, \tag{12.24}$$

ergibt sich die Anzahl benötigter Dreiecke N aus

$$N \cdot A_\Delta = 4\pi r^2 \tag{12.25}$$

mit dem Ergebnis

$$N \approx 3000. \tag{12.26}$$

Zur Berechnung der Kapazitäten und Keramikvolumina der rekonstruierten Keramikgefäße (Kapitel 11.4 und 14.3) wurden jeweils Geometrien mit mehreren Zehntausend Dreiecken verwendet, die aufgrund dieser Untersuchungen die Krümmung der rekonstruierten Objekte sehr exakt approximieren. Des Weiteren bestätigt dieses Ergebnis die Beobachtung aus Kapitel 11.3.2, dass bereits sehr grobe Dreiecksgitter ausreichen, um rekonstruierte Objekte gut zu approximieren.

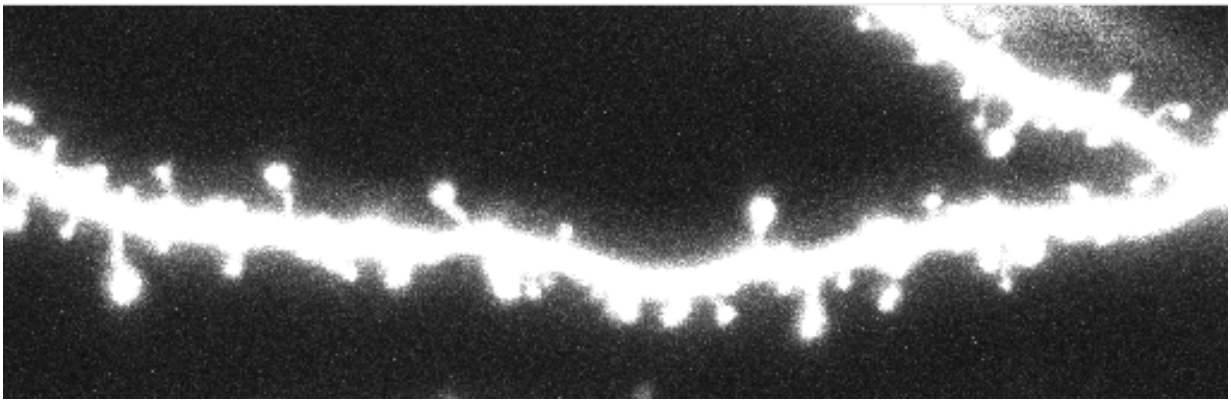
Kapitel 13

Erzeugung von Merkmalskeletten

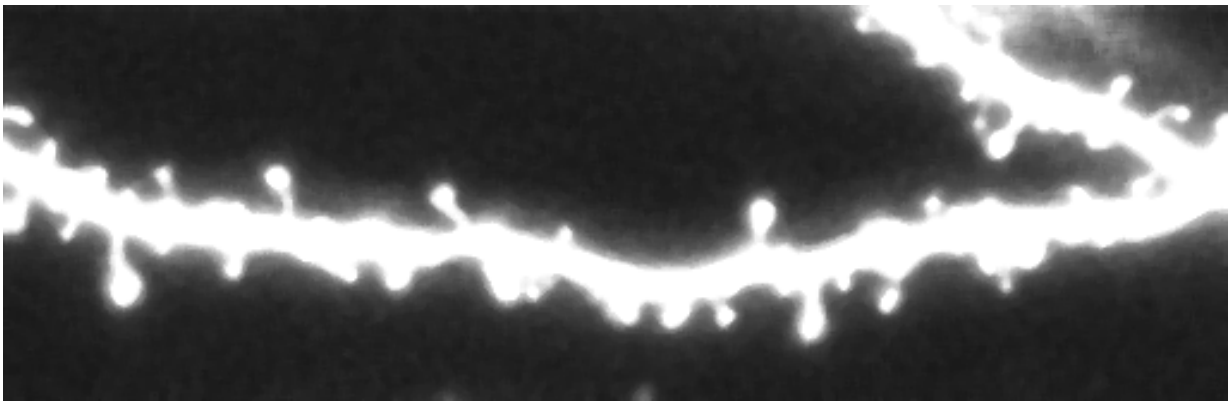
Der Begriff *Skelettierung* bezeichnet eine Klasse von Bildverarbeitungsoperatoren, die aus in Bildern enthaltenen Objekten eine Mittelachse mit einer Breite eines Bildpunktes extrahieren. Der in der ursprünglichen Version von *NeuRA* verwendete Skelettierer, der auf einem *Thinning*-Verfahren beruht [19, 104], hat sich für stark verrauschte *in vivo* Aufnahmen als nicht brauchbar erwiesen. Daher ist das in *SpineLab* zur Mittelachsen-Erzeugung von Dendritensegmenten oder ganzen Neuronenzellen verwendete *Erweiterte Fast Marching Verfahren* [110] (kurz *AFM-Verfahren* für *Augmented Fast Marching Method*) in diesem Kapitel ausführlich beschrieben. Es wird ausgenutzt, dass die Auflösung in x-Richtung und y-Richtung der betrachteten konfokalen Mikroskopaufnahmen gegenüber der Auflösung in z-Richtung deutlich höher ist: Das vorgefilterte Bild wird in Richtung der z-Achse mittels Maximum-Kriterium auf eine Ebene projiziert. Von diesem zweidimensionalen Bild wird anschließend durch das *AFM*-Verfahren eine Mittelachse erzeugt, aus der ein Graph, das sogenannte *Merkmalskelett*, generiert wird. Anschließend wird jedem Skelettknoten eine z-Koordinate zugewiesen, indem der Schwerpunkt des dreidimensionalen Bildes entlang der z-Richtung in den x- und y-Koordinaten des Skelettknotens berechnet wird. Trotz der zwischenzeitlichen Projektion auf eine Ebene entsteht auf diese Weise ein dreidimensionales Skelett, das durch einen Graphen repräsentiert wird. Die erzeugten Merkmalskelette sind häufig unvollständig, da die Morphologie der auf diese Weise rekonstruierten Mikroskopdaten meist sehr komplex und stark verrauscht ist. Die im Rahmen dieser Arbeit entwickelte Software *SpineLab* stellt daher alle notwendigen Methoden zur Verfügung, um die automatisch erzeugten Skelette beliebig nachzubearbeiten und anschließend automatisch zu analysieren. Das in *SpineLab* implementierte Rekonstruktionsverfahren wurde bereits vorab in [53] veröffentlicht.

13.1 Erzeugung von Mittelachsen

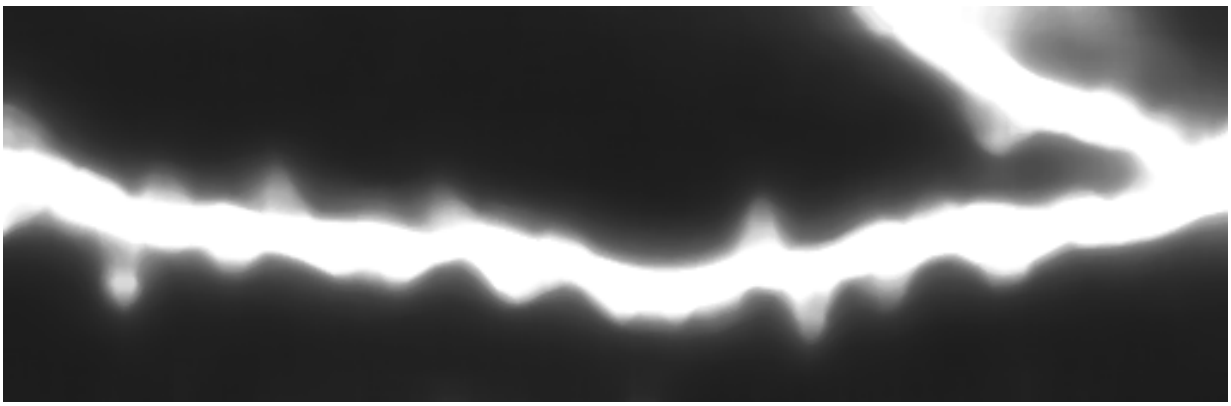
Die Generierung von Mittelachsen ist ein weiteres komplexes Themengebiet der digitalen Bildverarbeitung und kann systematisch in den Arbeiten [11, 22] studiert werden. Zur Generierung von Mittelachsen in Aufnahmen mit neurobiologischem Inhalt wurden sowohl



(a)

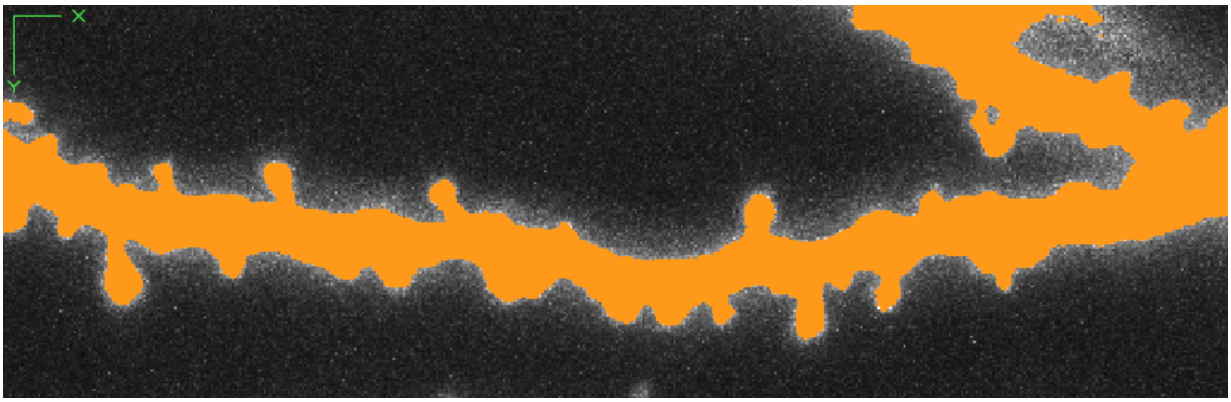


(b)

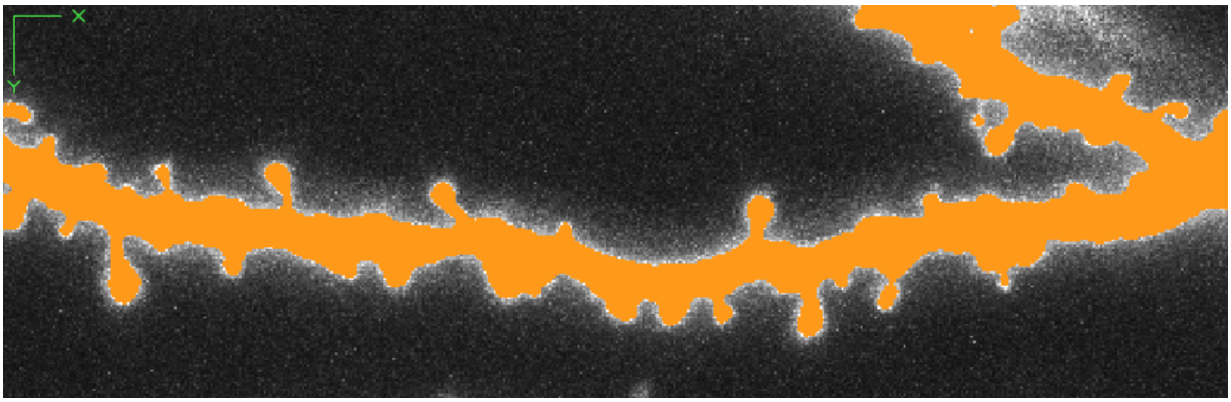


(c)

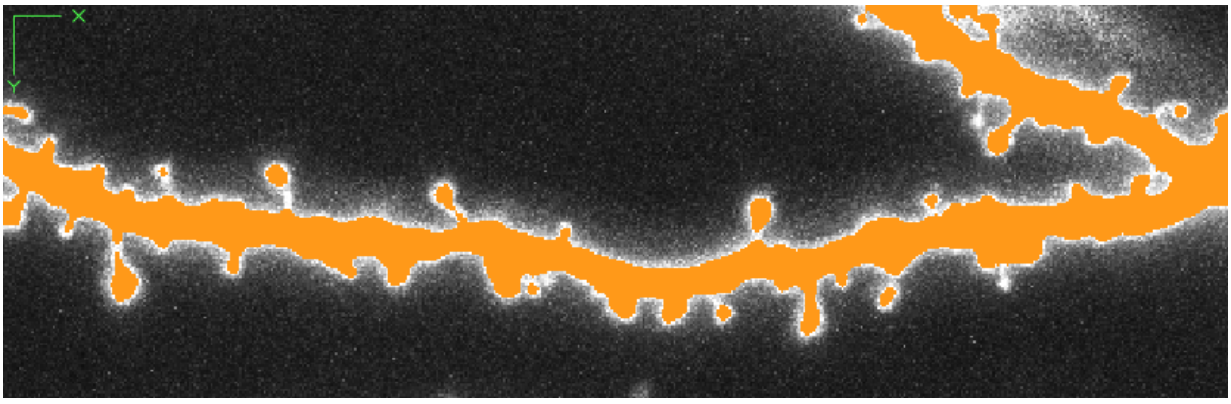
Abbildung 13.1: *(a) Entlang der z-Achse projiziertes Dendritensegment. (b) Nach Anwendung eines Medianfilters mit Nachbarschaftsgröße 2. (c) Nach Anwendung eines Medianfilters mit Nachbarschaftsgröße 9. Bei einer zu großen Nachbarschaftsgröße geht relevante Bildinformation verloren.*



(a)

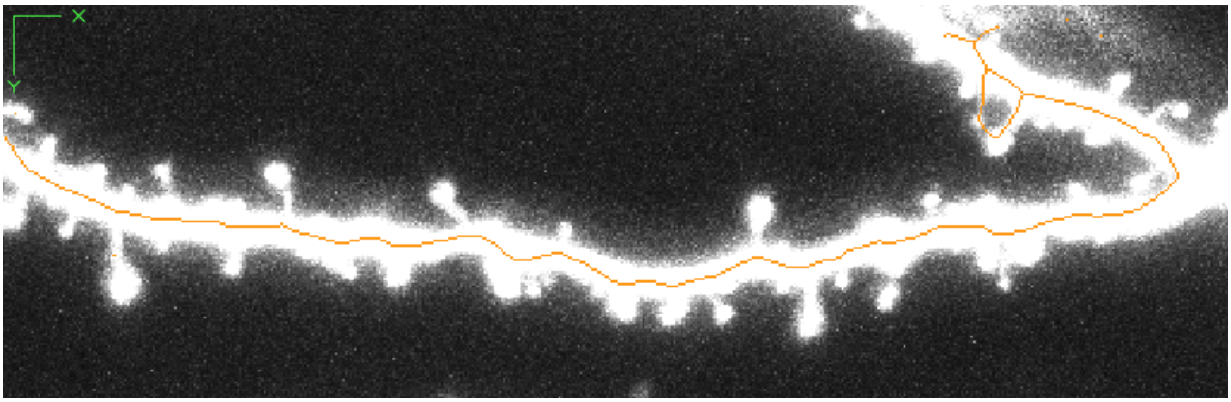


(b)

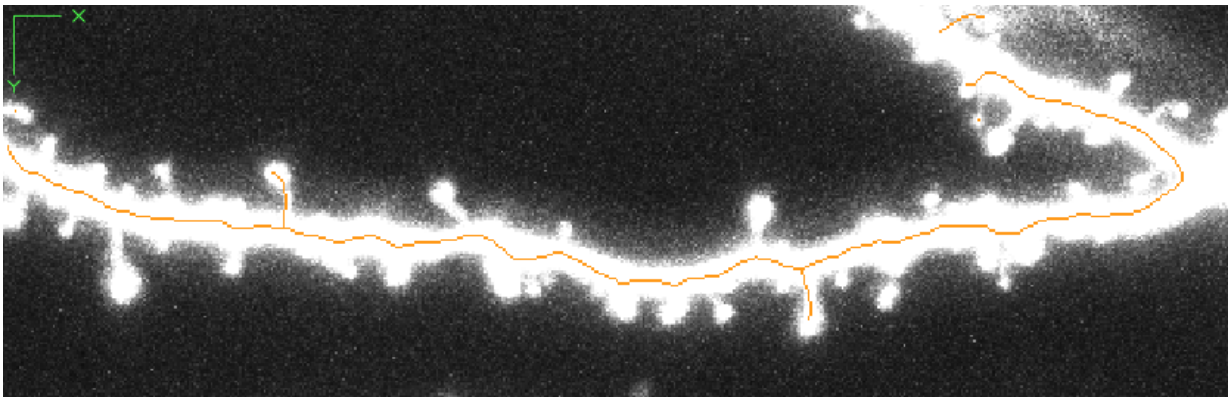


(c)

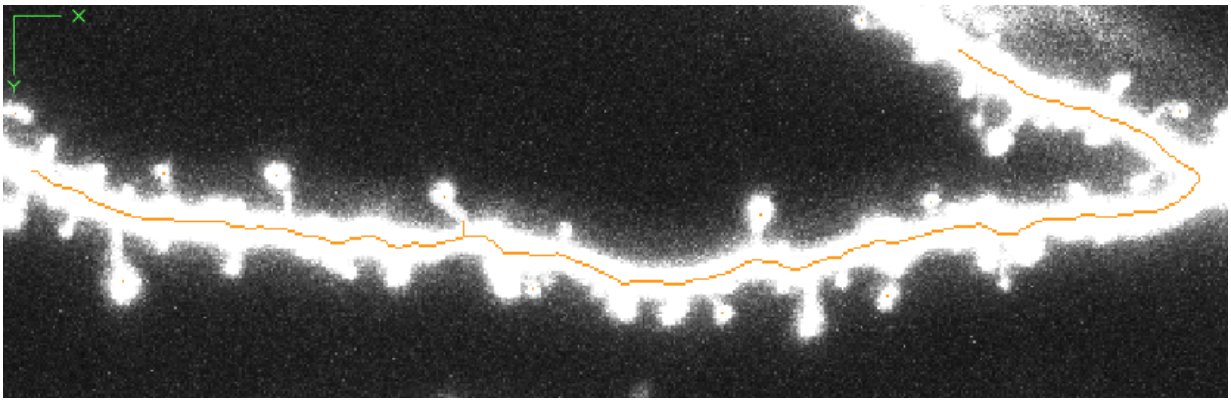
Abbildung 13.2: *Projiziertes und gefiltertes Bild (Abbildung 13.1) nach Anwendung des globalen Segmentierungsverfahrens (orange) mit (a) Schwellwert* $\vartheta = 150$*, (b) Schwellwert* $\vartheta = 200$ *und (c) Schwellwert* $\vartheta = 250$*.*



(a)



(b)



(c)

Abbildung 13.3: *Mittelachse (orange) des projizierten Bildes. (a) Segmentierungsschwellwert $\vartheta = 150$. (b) Segmentierungsschwellwert $\vartheta = 200$. (c) Segmentierungsschwellwert $\vartheta = 250$. Die Mittelachse im quer liegenden Dendriten ist nahezu gleich, während die Mittelachse im rechten oberen Bereich und in den Spines voneinander abweicht.*

Verfahren getestet, die auf dreidimensionalen Bildern basieren [99, 109], als auch Verfahren, die Skelette aus rekonstruierten Oberflächengittern erzeugen [7] (vgl. Kapitel 14.4.4 und Abbildung 14.40). Aufgrund der geringen z-Auflösung der konfokalen Mikroskope und des hohen Rauschanteils in den Aufnahmen liefern diese Algorithmen jedoch keine befriedigenden Ergebnisse. Die entscheidende Neuerung, hochwertige Skelettierungen dieser Aufnahmen zu erzeugen, ist die Projektion entlang der z-Achse.

13.1.1 Projektion entlang der z-Achse

Definition 13.1 (Maximum-Projektion)
Sei $u(x, y, z)$ ein dreidimensionales Grauwertbild der Größe $n_x \times n_y \times n_z$. So ist die zweidimensionale Maximum-Projektion *$u_\pi(x, y)$ entlang der z-Achse durch*

$$u_\pi(x, y) := \max_{z=1,..,n_z} u(x, y, z) \tag{13.1}$$

definiert.

Nach der Projektion entlang der z-Achse ist die Anwendung eines Medianfilters (Kapitel 4.3.2) mit kleiner Nachbarschaftsgröße (1 oder 2) sinnvoll, um den Rand des relevanten Objekts von restlichem Rauschen zu befreien (Abbildung 13.1). Wird die Nachbarschaftsgröße des Medianfilters allerdings zu groß gewählt, geht relevante Bildinformation verloren (Abbildung 13.1c).

13.1.2 Segmentierung des projizierten Bildes

Zur Bestimmung der Mittelachse muss das projizierte Bild segmentiert werden (Kapitel 7). Es hat sich gezeigt, dass eine globale Schwellwertsegmentierung (Kapitel 7.1) an dieser Stelle ausreichend ist, da für die Mittelachsenextraktion keine exakte Erkennung der Kanten notwendig ist. Wird der Schwellwert größer oder kleiner gewählt, wächst bzw. schrumpft das segmentierte Objekt. Dies hat aber nur einen geringen Einfluss auf die Lage der Mittelachse, da dieses Wachsen oder Schrumpfen in alle Richtungen relativ gleichmäßig stattfindet. In Abbildung 13.3 ist die Mittelachse in Abhängigkeit des Segmentierungsparameters ϑ dargestellt, der stark variiert wird. Die Mittelachse im quer liegenden Dendriten ist aber bei allen Schwellwerten fast identisch. Unterschiede sind im rechten oberen Teil des Bildes sowie bei der Mittelachse innerhalb der Spines erkennbar. Der einzustellende Schwellwert hängt im Allgemeinen von den Daten ab und die Qualität der Mittelachse lässt sich leicht optisch beurteilen. Daher stellt die Software *SpineLab* eine einfache Möglichkeit zur Einstellung des Schwellwerts zur Verfügung, dessen Ergebnis leicht kontrolliert und gegebenenfalls korrigiert werden kann.

13.1.3 AFM-Verfahren

Aus dem projizierten und segmentierten Bild wird die Mittelachse mit Hilfe des *AFM-Verfahrens* [110] extrahiert. Dieses Verfahren ist, wie in Abbildung 13.2 zu sehen, robust

gegenüber der Wahl des Schwellwertparameters und hängt nur vom Parameter ζ ab, der die Größe von Teilobjekten angibt, die durch eine eigene Mittelachse und damit durch einen eigenen Skelettarm charakterisiert werden. Die in *SpineLab* verwendete effiziente Implementierung ermöglicht ebenfalls eine durch den Benutzer an die Daten angepasste Einstellung dieses Parameters in Echtzeit. Grundlage für das *Erweiterte Fast Marching Verfahren* (AFM) ist das *Fast Marching Verfahren* (FMM) [103].

Fast Marching Verfahren

Die Mittelachse wird durch das Zusammenziehen des Randes des segmentierten Objekts erzeugt. Hierbei beschreibt das über die *Eikonal-Gleichung*

$$|\nabla T| = 1 \tag{13.2}$$

mit $T = 0$ auf dem Objektrand definierte skalare Feld T die Distanz eines Bildpunktes vom Rand des Objekts [103]. Ausgehend vom Rand des Objekts werden sukzessiv alle Werte des Feldes T innerhalb des Objekts berechnet, wobei der Rand des Objekts als *Band* betrachtet wird, das nach und nach immer enger gezogen wird, bis alle Pixel des Objekts untersucht wurden. Anders ausgedrückt beschreibt der Rand des Objekts eine *Pixelfront*, die sich in das Objekt hinein bewegt, bis nur noch die Mittelachse des Objekts übrig bleibt. Für jedes Pixel (x, y) des zweidimensionalen Bildes wird der Wert $T(x, y)$ und ein zugehöriger Wert $f(x, y)$ gespeichert, der folgende Werte annehmen kann:

- `BAND`: Das Pixel gehört zur sich bewegenden Pixelfront. Der zugehörige T-Wert wird aktuell berechnet.
- `INSIDE`: Das Pixel liegt innerhalb der sich bewegenden Front. Der zugehörige T-Wert ist noch nicht bekannt.
- `KNOWN`: Das Pixel liegt hinter der sich bewegenden Pixelfront. Der zugehörige T-Wert wurde bereits berechnet.

Die Felder f und T werden durch

$$f(x,y) := \begin{cases} \texttt{BAND} \text{ für } (\texttt{x},\texttt{y}) \text{ auf dem Rand des Objekts} \\ \texttt{INSIDE} \text{ für } (\texttt{x},\texttt{y}) \text{ innerhalb des Objekts} \\ \texttt{KNOWN} \text{ sonst} \end{cases} \tag{13.3}$$

und

$$T(x,y) := \begin{cases} 0 \text{ für } (x,y) \text{ auf dem Rand des Objekts} \\ \texttt{MAX\_VALUE} \text{ für } (\texttt{x},\texttt{y}) \text{ innerhalb des Objekts} \\ 0 \text{ sonst} \end{cases} \tag{13.4}$$

initialisiert. Hierbei bezeichnet `MAX_VALUE` einen Wert, der größer ist, als alle Werte, die praktisch von T angenommen werden. Zudem werden alle Bildpunkte, die auf dem Rand des Objekts liegen, in dem Container `NarrowBand` gespeichert. Dieser Container wird im weiteren Verlauf des Verfahrens alle Punkte speichern, die zu der kollabierenden Front von Pixeln gehören, aus der die Mittelachse entsteht. Die Werte werden dabei aufsteigend nach ihrem jeweiligen T-Wert sortiert. Unter Verwendung der Datenstruktur `multimap` aus der *C++-Standard-Template-Library* kann der `NarrowBand`-Container sehr effizient implementiert werden. Nach der Initialisierung breitet sich das Band nach und nach soweit aus, bis alle T-Werte des Objekts berechnet wurden. Hierzu wird die *Eikonal-Gleichung* (13.2) mittels der finiten Differenzen

$$\begin{aligned} &\max(T(x,y) - T(x-1,y),\ T(x,y) - T(x+1,y),\ 0)^2 \\ +\ &\max(T(x,y) - T(x,y-1),\ T(x,y) - T(x,y+1),\ 0)^2 = 1 \end{aligned} \tag{13.5}$$

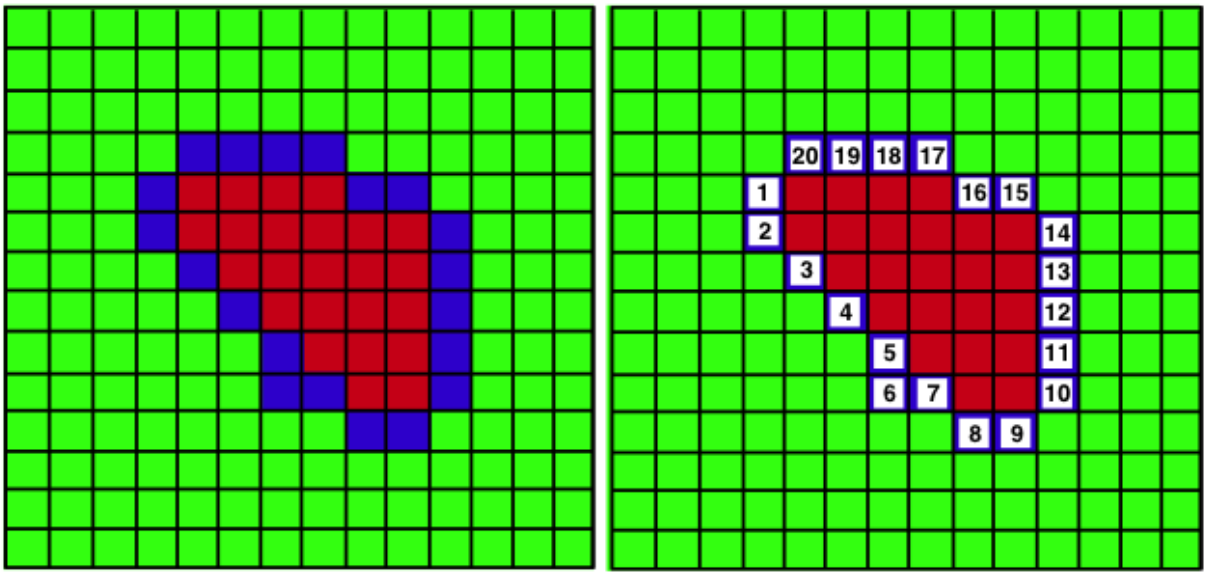
diskretisiert. Durch ein in [103] vorgeschlagenes Upwind-Verfahren, das jeweils die Einschränkung von Gleichung (13.5) auf alle vier Nachbarn des Pixels (x, y) berechnet und als Lösung für T den kleinsten dieser vier Werte verwendet, kann die diskretisierte *Eikonal-Gleichung* sehr effizient gelöst werden. Weitere Erläuterungen [103] und Pseudocode [110] zur Lösungsroutine sind den angegebenen Quellen zu entnehmen.

Bei der Ausbreitung des Bandes werden folgende Schritte abgearbeitet, solange sich Elemente im `NarrowBand` Container befinden:

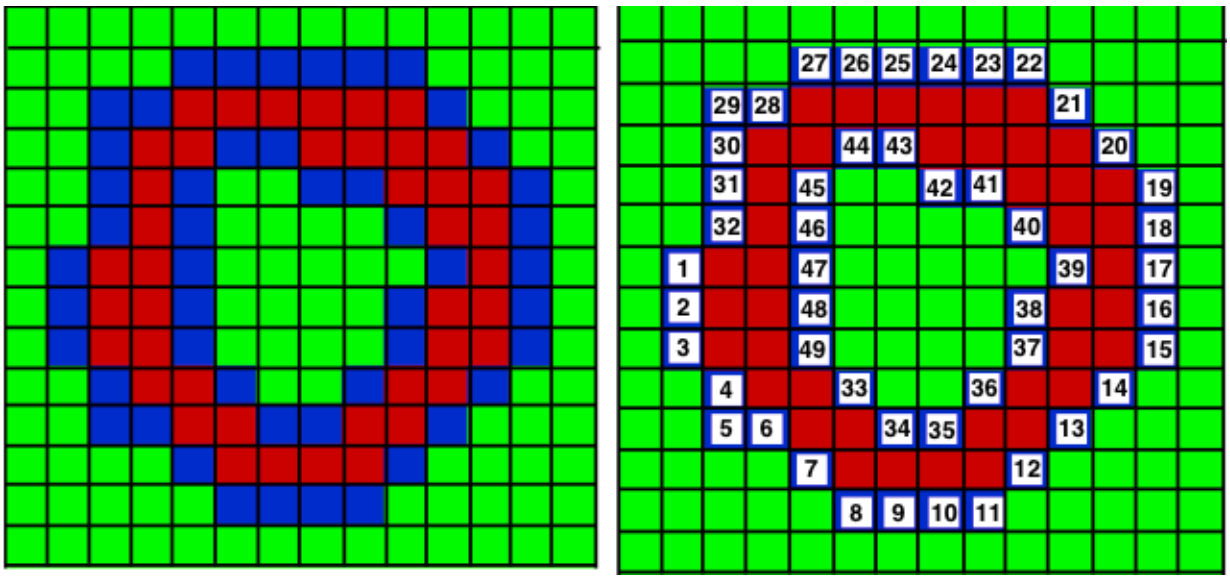
- Der Punkt (x, y) mit dem kleinsten T-Wert wird aus dem `NarrowBand` extrahiert und $f(x, y)$ auf den Wert `KNOWN` gesetzt.

- Alle Nachbarpixel von (x, y), deren f-Wert nicht `KNOWN` ist, werden in das `NarrowBand` eingefügt. Der zugehörige Wert von T wird durch das oben beschriebene Upwind-Schema berechnet.

Zwei Pixel werden dabei als benachbart bezeichnet, wenn sich genau eine ihrer Koordinaten um den Wert 1 unterscheidet. Das Feld T enthält nach Abschluss der Berechnung für alle Pixel, die innerhalb des Objekts liegen, den Abstand des jeweiligen Pixels zum Rand des Objekts.



(a)

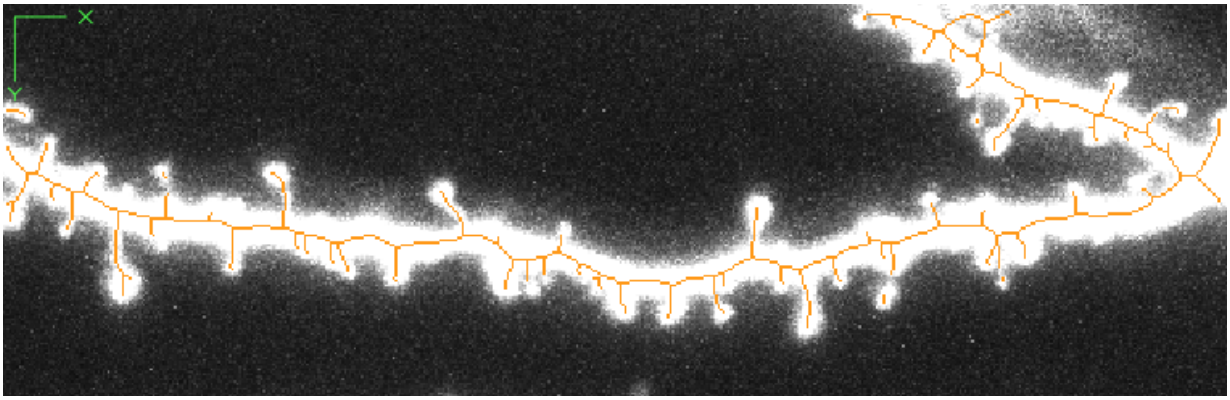


(b)

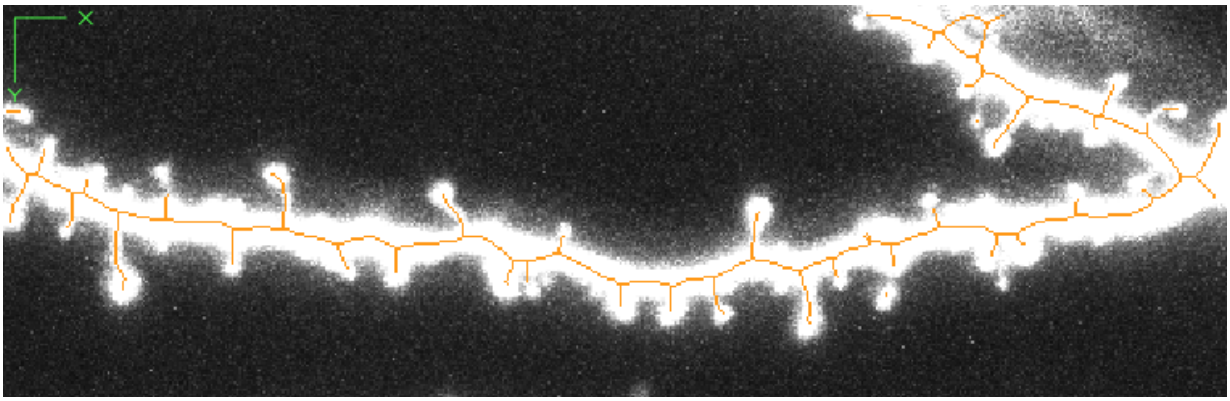
Abbildung 13.4: *Initialisierung des Feldes U bei Objekten mit (a) einem Rand und (b) mehreren Rändern. Quelle:* [110]

Erweitertes Fast-Marching Verfahren

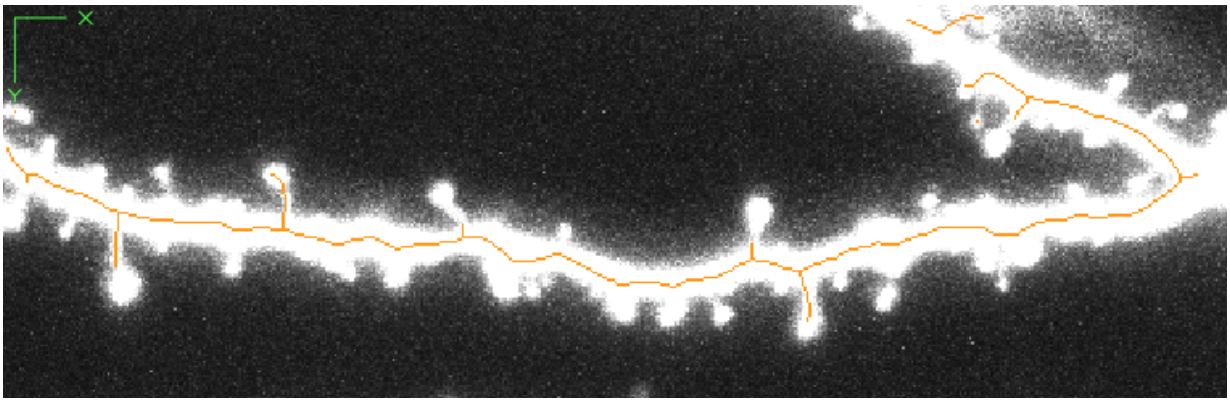
Das Fast-Marching Verfahren wird nun dahingehend erweitert, dass jedem Pixel ein weiterer Wert U zugewiesen wird, der die Länge zu dem Randpixel angibt, das in diesem Punkt kollabiert ist [110]. Ausgehend von einem beliebigen Randpunkt des Objekts, der mit dem Wert $U = 1$ initialisiert wird, werden monoton aufsteigend die weiteren Randpunk-



(a)



(b)



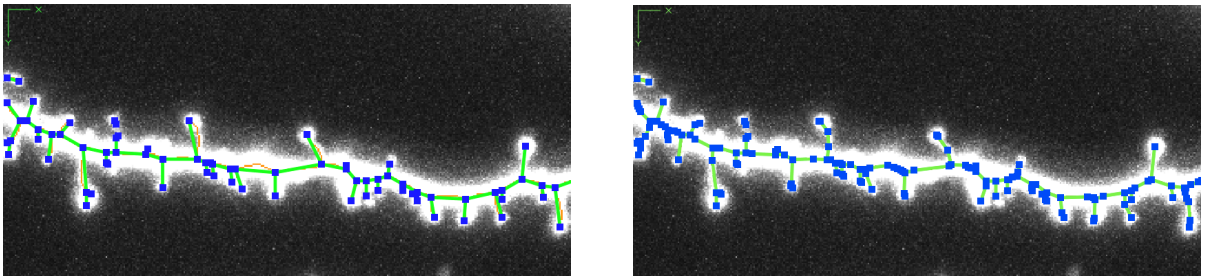
(c)

Abbildung 13.5: *Durch den Parameter ζ des AFM-Verfahrens kann die Sensitivität der Skelettierung in Bezug auf die Größe von Teilobjekten (z.B. Spines) eingestellt werden. (a) $\zeta = 5$, (b) $\zeta = 10$, (c) $\zeta = 50$.*

te entsprechend ihres Abstandes entlang des Randes zum Ausgangspixel initialisiert (Abbildung 13.4). Besteht ein Objekt aus mehreren Rändern, wird der weitere Rand ebenso initialisiert, wobei die zusätzlichen Ausgangspixel nicht mit 1 sondern mit $\max(U) + 1$ initialisiert werden (Abbildung 13.4b). Das Feld U wird ebenso wie das Feld T während der Ausbreitung des Pixelbandes für alle zum Objekt gehörenden Pixel berechnet. Hierzu wird für jedes dem `NarrowBand` neu zugeordneten Pixel nicht nur der Wert für T, sondern auch der Wert für U berechnet:
Sei (x, y) das letzte dem `NarrowBand` entnommene Pixel und $(\tilde{x}, \tilde{y})$ das aktuell betrachtete Nachbarpixel von (x, y), dessen f-Wert noch nicht bekannt ist. Unterscheiden sich die bereits bekannten U-Werte aller Nachbarn von $(\tilde{x}, \tilde{y})$ um mehr als den Wert 2, so bedeutet dies, dass nicht benachbarte Randpunkte in dem Pixel $(\tilde{x}, \tilde{y})$ kollabiert sind [110]. Der U-Wert von $(\tilde{x}, \tilde{y})$ wird gleich dem U-Wert von (x, y) gesetzt. Andernfalls wird der U-Wert von $(\tilde{x}, \tilde{y})$ als arithmetisches Mittel aller bekannten U-Werte von Nachbarn von $(\tilde{x}, \tilde{y})$ berechnet. Dieses Vorgehen ist detailliert und durch Pseudocode veranschaulicht in [110] nachzulesen. Nachdem das Feld U für alle dem Objekt zugehörigen Pixel berechnet wurde, können die Skelettpunkte bestimmt werden. Hierbei werden alle Pixel betrachtet, die Nachbarn haben, deren U-Wert sich um mindestens ζ unterscheidet, wobei ζ den einzustellenden Parameter bezeichnet. Aufgrund der Definition von U hat ζ eine exakte geometrische Bedeutung: Jedes Teilobjekt, dessen Rand länger als ζ Pixel ist, wird mit einem eigenen Skelettarm versehen (Abbildung 13.5). Da der Parameter ζ signifikant von den Daten und der Größe der zu detektierenden Strukturen abhängt, muss er für jedes Bild individuell eingestellt werden. Es bleibt anzumerken, dass durch die Wahl der initialen Randpunkte Skelettarme berechnet werden, die eigentlich keine sind, da sich die U-Werte des Start- und des Endpunktes jedes Randes um einen großen Wert unterscheiden. Daher wird der komplette Algorithmus zweimal durchlaufen, wobei die Ränder bei den beiden Durchläufen unterschiedlich initialisiert werden. Die resultierende Mittelachse ist dann der Schnitt der beiden erzeugten Mittelachsen. Ein weiteres Artefakt des *AFM*-Verfahrens ist, dass die Mittelachsen nie bis zum Rand des Objekts laufen. Die Endpunkte der Skelettarme müssen daher in einem weiteren Verarbeitungsschritt an das Ende der Spines versetzt werden, damit die auf diese Weise erzeugten Skelette für die Längenvermessung solcher Spines verwendet werden können. Im nächsten Abschnitt wird beschrieben, wie aus der Mittelachse des projizierten Bildes ein dreidimensionales Skelett von Dendritensegmenten oder von ganzen Neuronenzellen generiert werden kann und auf welche Weise diese Skelette semi-automatisch nachbearbeitet werden können.

13.2 Generierung des Merkmalskeletts

Auf der Basis der Mittelachse wird das dreidimensionale *Merkmalskelett* erzeugt, ein Graph, der die Morphologie des zu rekonstruierenden Objekts beschreibt und aus sogenannten *Skelettknoten* und *Skelettkanten* besteht. Hierbei werden zunächst die Verzweigungspunkte der Mittelachse sowie deren Endpunkte bestimmt. Dort werden Skelettknoten eingefügt und diese, der Topologie der Mittelachse entsprechend, mit Skelettkanten verbunden (Ab-



(a) **(b)**

Abbildung 13.6: *Generierung des Merkmalskeletts. (a) Zunächst werden an allen Verzweigungs- und Endpunkten Skelettknoten (blau) eingefügt und diese der Topologie der Mittelachse entsprechend verbunden (grün). (b) Anschließend werden so viele zusätzliche Skelettpunkte eingefügt, bis die Mittelachse (orange) durch das Merkmalskelett komplett überdeckt ist, d.h. das Merkmalskelett die Mittelachse optimal approximiert.*

bildung 13.6a). Anschließend werden die Mittelachsenstücke entlang dieser Verzweigungspunkte abgefahren und solange zusätzliche Skelettknoten auf der Mittelachse eingefügt, bis alle Skelettkanten auf der Mittelachse liegen. Dies gewährleistet eine optimale Approximation der Mittelachse durch die Skelettknoten (Abbildung 13.6b).

13.2.1 Berechnung der z-Koordinate

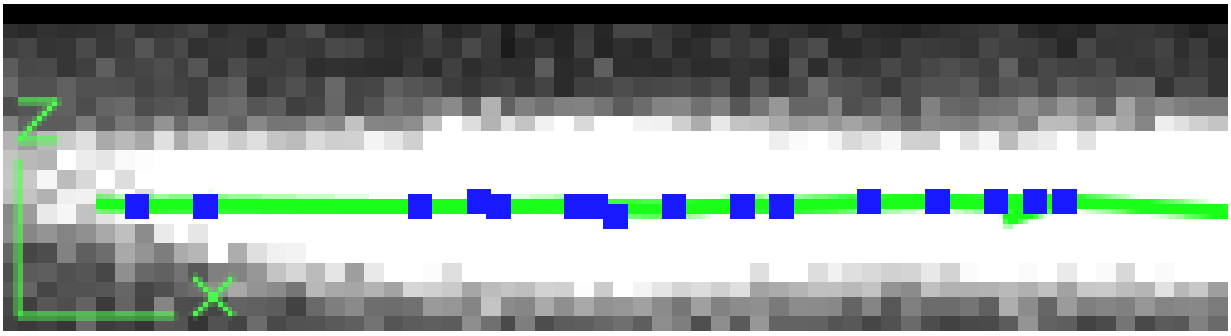


Abbildung 13.7: *Berechnung der z-Koordinate nach Formel* (13.6)*. Das erzeugte Merkmalskelett liegt in z-Richtung mittig im Dendriten.*

Da die Mittelachse durch Projektion des Bildes entlang der z-Achse generiert wurde, fehlt dem daraus gewonnenen Skelett zunächst diese Tiefeninformation. Jeder Skelettknoten, der auf Basis der Mittelachse berechnet wurde, kann einem Punkt auf der Mittelachse zugeordnet werden und besitzt daher ganzzahlige Koordinaten (x, y). Sei $u(x, y, z)$ das dreidimensionale Bild der Dimension $n_x \times n_y \times n_z$, so berechnet sich die z-Koordinate s_z des Skelettknotens an der Position (x, y) durch

$$s_z = \frac{\sum_{z=1}^{n_z} z \cdot u(x,y,z)}{\sum_{z=1}^{n_z} u(x,y,z)}. \tag{13.6}$$

Dies entspricht dem Schwerpunkt der Grauwertverteilung des Bildes entlang der z-Achse. Um diese Berechnung robust gegenüber Rauschen im Bildsignal zu gestalten, kann anstatt des originalen Bildes u ein vorgefiltertes Bild $\tilde{u}$ verwendet werden. Hierbei hat sich ein Medianfilter mit kleiner Nachbarschaftsgröße als sehr brauchbar erwiesen. Abbildung 13.7 zeigt das Merkmalskelett eines Dendriten, das durch die hier beschriebene Berechnung der z-Koordinate mittig im Dendriten liegt.

13.3 Spezielle Merkmalskelette

Die aus dreidimensionalen Bildern rekonstruierten Merkmalskelette eignen sich sehr gut zur Analyse der Länge der in den Bildern enthaltenen Objekten und Teilobjekten. In der Software *SpineLab* sind mit der Analyse von Dendritensegmenten und der Rekonstruktion von ganzen Neuronenzellen zwei Anwendungen realisiert, die im Folgenden diskutiert werden. Weitere Details zu neurobiologischen Anwendungsfeldern sind Kapitel 14.4.2 zu entnehmen.

13.3.1 Analyse von Dendritensegmenten

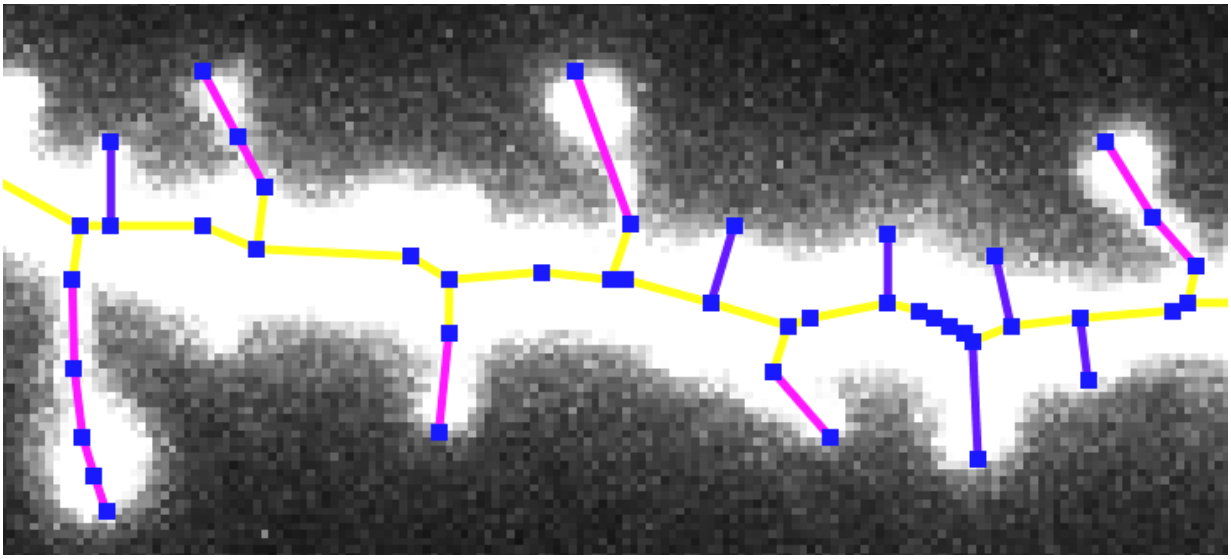


Abbildung 13.8: *Skelett zur Analyse von Dendritensegmenten. Der Dendrit ist gelb eingefärbt, Spines, die bei der Längenanalyse berücksichtigt werden, sind magenta und Spines, die bei der Längenanalyse nicht berücksichtigt werden, sind violett eingefärbt. Die Länge eines Spines entspricht der Summe der Länge der magentafarbenen Kanten.*

Bei der Analyse von Dendritensegmenten sind folgende Fragestellungen in den Neurowissenschaften interessant [112]:

- Bestimmung der Anzahl an Spines eines Dendritensegments
- Automatische Vermessung von Spines
- Bestimmung der Volumina einzelner Spines

Grundlage für diese in *SpineLab* implementierten Analyseverfahren ist eine spezielle Skelettform:

Definition 13.2 (Skelett zur Analyse von Dendritensegmenten)
Ein Merkmalskelett zur Analyse von Dendritensegmenten ist ein spezieller Graph $G = (V, K)$*, bestehend aus den Vertices* V *und den Kanten* K*, mit folgenden Eigenschaften:*

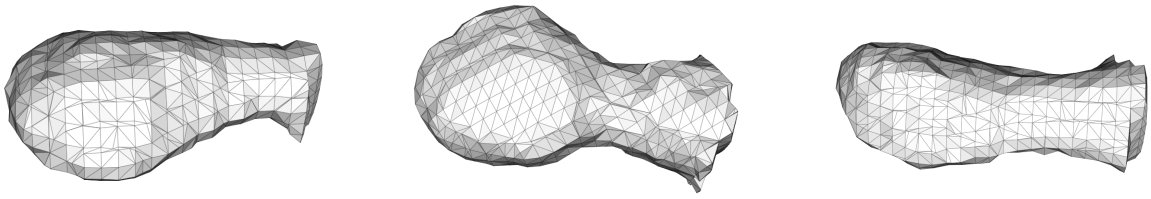
- G *ist ein Baum, d. h.* G *ist zusammenhängend und enthält keine Zyklen* [115].
- G *ist gewichtet, wobei die Gewichte der Kanten die reale Länge in Mikrometern symbolisieren.*
- *Der längste Pfad von* G *repräsentiert den Dendriten.*
- *Am Dendriten angehängte Äste repräsentieren Spines.*

Die Spines werden dabei in zwei Kategorien eingeteilt: Einige der Spines sollen bei der automatischen Längenanalyse nicht berücksichtigt werden, beispielsweise wenn sie hauptsächlich in z-Richtung orientiert sind und dadurch die Länge aufgrund der anisotropen Auflösung nicht hinreichend exakt bestimmt werden kann. Bei den übrigen Spines soll die Länge automatisch berechnet werden.

Definition 13.3 (Spine)
Sei G *ein Graph entsprechend Definition* (13.2)*.* S *heißt Spine, falls* $S = (V_s, K_s) \subset G = (V, K)$ *mit* $|V_s| \geq 1$ *gilt,* S *zusammenhängend ist und alle Vertices von* S *höchstens Grad* 2 *haben.*

- *Falls* $|V_s| > 1$ *wird der Spine bei der Längenanalyse berücksichtigt.*
- *Falls* $|V_s| = 1$ *wird der Spine bei der Längenanalyse nicht berücksichtigt.*

Definition 13.4 (Länge eines Spines)
Die Länge eines bei der Längenanalyse berücksichtigten Spines S *ist die Summe der Länge aller seiner Kanten, d.h. die Summe der Gewichte aller Kanten von* S*.*



(a) **(b)** **(c)**

Abbildung 13.9: *Oberflächenrekonstruktionen einzelner Spines.*

Bemerkung 13.5
Ein bei der Längenanalyse nicht berücksichtigter Spine S hat nach der obigen Definition die Länge 0*.*

Das Beispiel eines solchen Merkmalskeletts und die zugrunde liegende Mikroskopaufnahme ist Abbildung 13.8 zu entnehmen. Wurden die Spines auf diese Weise identifiziert, kann ihr Volumen mit Hilfe des *NeuRA2*-Verfahrens berechnet werden. Hierbei wird auf der Ebene des dreidimensionalen Bildes mit Hilfe des vorhandenen Merkmalskeletts der zu untersuchende Spine vom Dendriten abgetrennt. Seine Oberfläche wird daraufhin als Dreiecksgitter rekonstruiert (Abbildung 13.9) und anschließend das Volumen berechnet. Dabei findet folgende Konfiguration bei der Rekonstruktion Anwendung:

1. Rauschverminderung durch Median-Filterung mit Nachbarschaftsgröße 1
2. Globale Segmentierung mit dem gleichen Schwellwert, der zur automatischen Mittelachsen-Bestimmung (Kapitel 13.1) verwendet wurde
3. Marching-Cubes-Gittergenerierung

Anschließend wird das Volumen des generierten Gitters durch das Verfahren nach Gauß (Kapitel 11.2) berechnet.

13.3.2 Rekonstruktion von Neuronenzellen

SpineLab ermöglicht das Merkmalskelett einzelner Neuronenzellen zu rekonstruieren, obwohl die zugrunde liegenden Mikroskopdaten über eine sehr schlechte Qualität verfügen, wie es bei *in vivo* Aufnahmen oftmals der Fall ist (Abbildungen 13.11, 13.12). Zur automatischen Längenanalyse der rekonstruierten Zellen dient wiederum die Repräsentation des Skeletts als Graph:

Definition 13.6 (Rekonstruktion einer Neuronenzelle)
Die Rekonstruktion einer Neuronenzelle ist ein Graph $G = (V, K)$ mit folgenden Eigenschaften:

- *G ist ein Baum.*

- *G ist ein gewichteter Graph, wobei die Gewichte den realen Längen der Zelle in Mikrometern entsprechen.*
- *Eines der Blätter von G ist als Zellkern (Soma) gekennzeichnet.*

Das Soma wird dabei vom Benutzer per Hand ausgewählt. Anschließend kann die Zelle automatisch vermessen werden, indem ausgehend vom Soma die Entfernung der einzelnen Verzweigungen und der Blätter des Baumes als Summe der Gewichte der Kanten, die durchlaufen werden, um diese Punkte zu erreichen, berechnet wird. Aus dieser Information wird ein sogenanntes *Dendrogramm* (Abbildung 13.13) erzeugt. Zudem wird der Winkel zwischen den Verzweigungspunkten automatisch berechnet. Damit dieser Winkel eindeutig definiert ist, wird hierbei angenommen, dass nur Bifurkationen als Verzweigungen in Neuronenzellen auftreten können. Der Winkel an einem Verzweigungspunkt ist dann als Winkel zwischen dem Verzweigungspunkt und den beiden nächsten Verzweigungspunkten höherer Ordnung definiert (Abbildung 13.10).

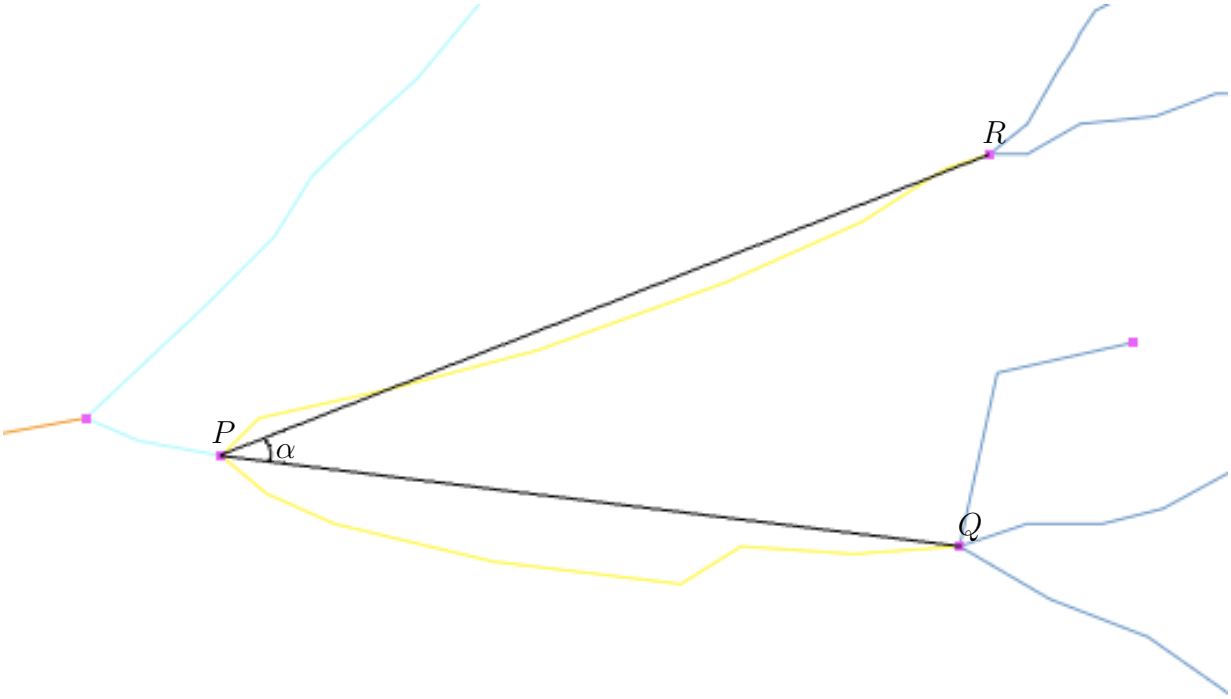


Abbildung 13.10: *Der Winkel α am Verzweigungspunkt P ist definiert als der Winkel zwischen dem Verzweigungspunkt und den beiden nächsten Verzweigungen Q und R.*

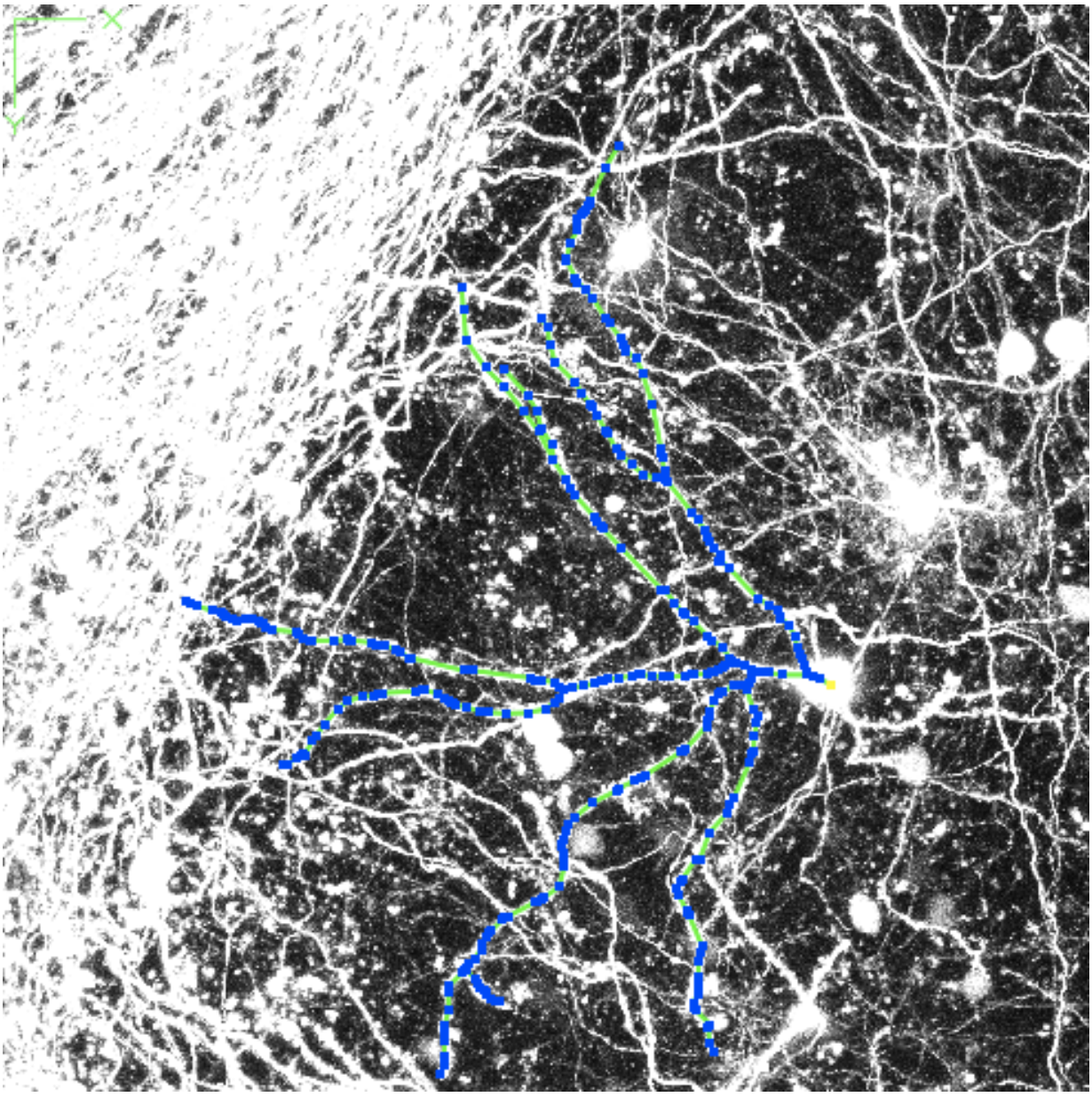


Abbildung 13.11: *Mit Hilfe von* SpineLab *lassen sich einzelne Neuronenzellen aus einem Verbund von Zellen rekonstruieren.*

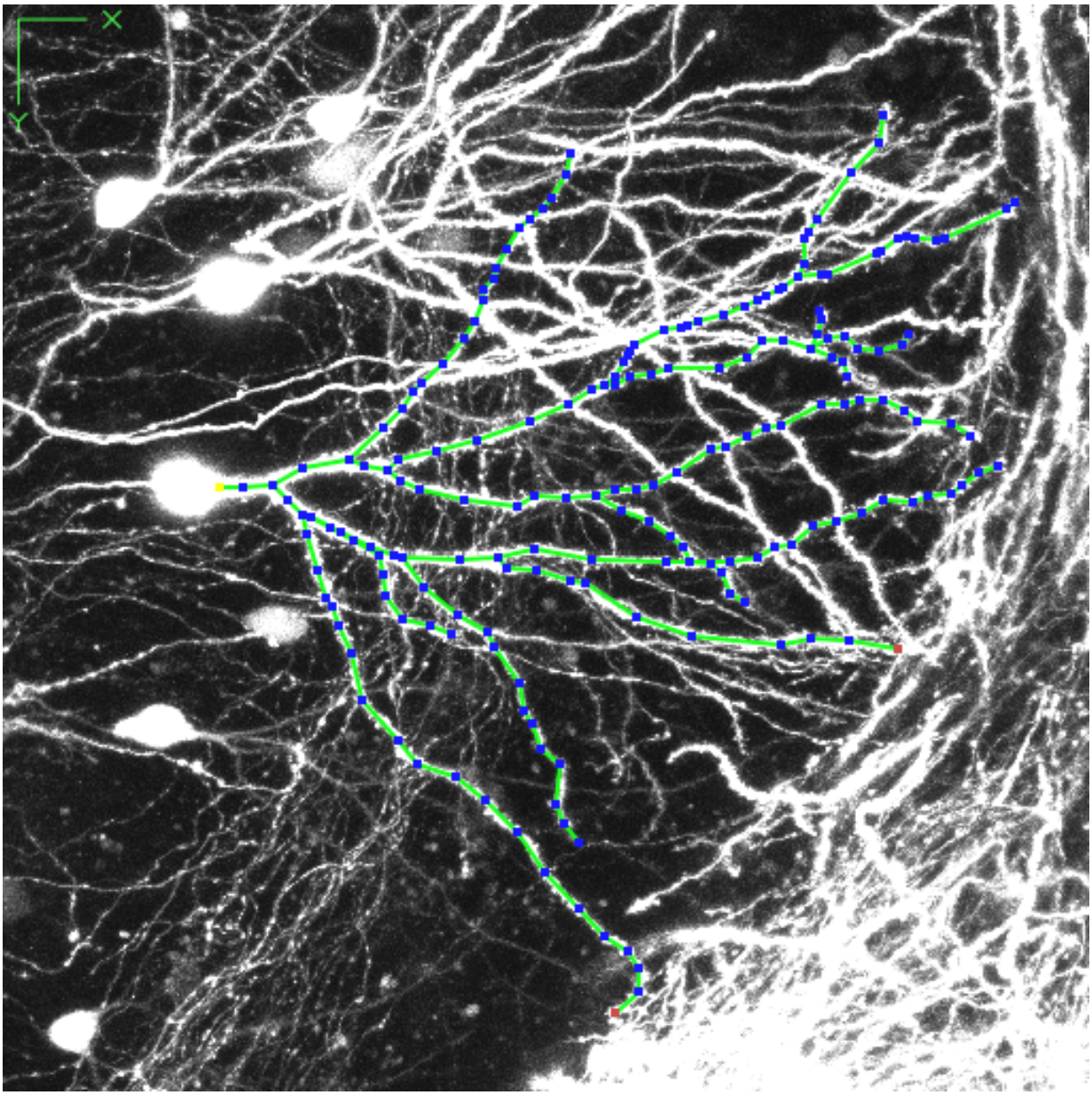
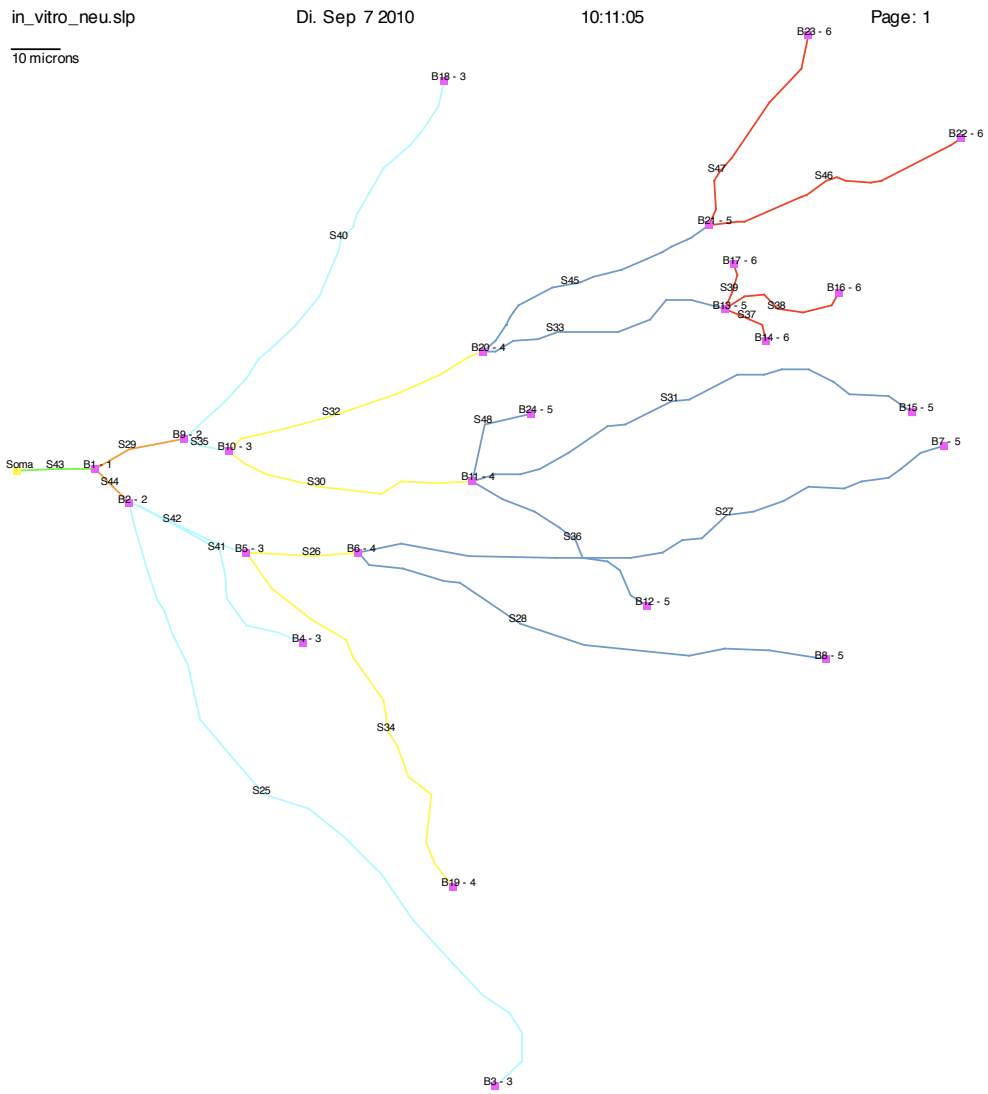


Abbildung 13.12: *Rekonstruktion einer weiteren Neuronenzelle. Die braunen Punkte markieren das Ende von Dendriten, die nicht in die Längenanalyse einfließen, da ihre Endpunkte nicht exakt festgelegt werden können.*



VoxelSizeX = 0.5	VoxelSizeY = 0.5	VoxelSizeZ = 3	TotalDendriticBranchLength = 1448.47		
B1 - 1: 18.0902	Angle: 71.299	B11 - 4: 100.17	Angle: 48.1382	S25: 170.639	S35: 10.4378
B2 - 2: 28.8279	Angle: 59.9941	B12 - 5: 166.859		S26: 23.7956	S36: 66.6882
B3 - 3: 199.467		B13 - 5: 177.346	Angle: 71.682	S27: 136.696	S37: 16.6786
B4 - 3: 91.1774		B14 - 6: 194.024		S28: 105.809	S38: 30.1937
B5 - 3: 65.1254	Angle: 71.6947	B15 - 5: 201.348		S29: 20.1782	S39: 20.9002
B6 - 4: 88.921	Angle: 22.8825	B16 - 6: 207.539		S30: 51.4642	S40: 102.288
B7 - 5: 225.617		B17 - 6: 198.246		S31: 101.178	S41: 62.3495
B8 - 5: 194.73		B18 - 3: 140.556		S32: 60.0654	S42: 36.2975
B9 - 2: 38.2684	Angle: 72.6616	B19 - 4: 164.992		S33: 68.574	S43: 18.0902
B10 - 3: 48.7063	Angle: 28.394	B20 - 4: 108.772	Angle: 42.1738	S34: 99.8663	S44: 10.7377

Abbildung 13.13: *Das Dendrogramm der rekonstruierten Neuronenzelle aus Abbildung 13.12 mit automatisch berechneten Längen der Dendriten und Winkeln zwischen den Verzweigungen.*

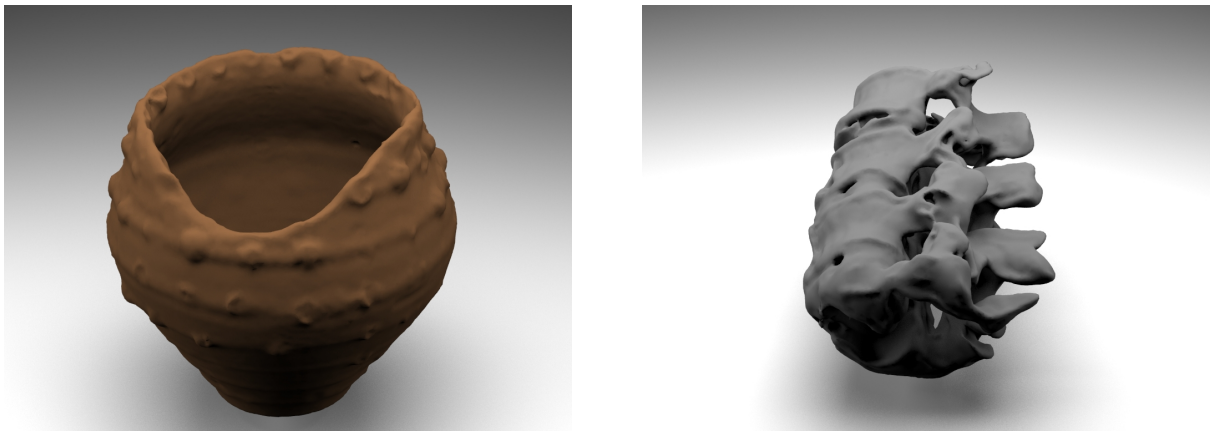
Abbildung 13.14: *Das Dendrogramm der rekonstruierten Neuronenzelle aus Abbildung 13.12 (Fortsetzung).*

Kapitel 14

Anwendungen und Ergebnisse

Im abschließenden Kapitel werden die mit *NeuRA2* und *SpineLab* im Rahmen dieser Arbeit rekonstruierten Objekte gezeigt, sowie deren vielfältige Anwendungen in den verschiedensten Wissenschaftsbereichen diskutiert.

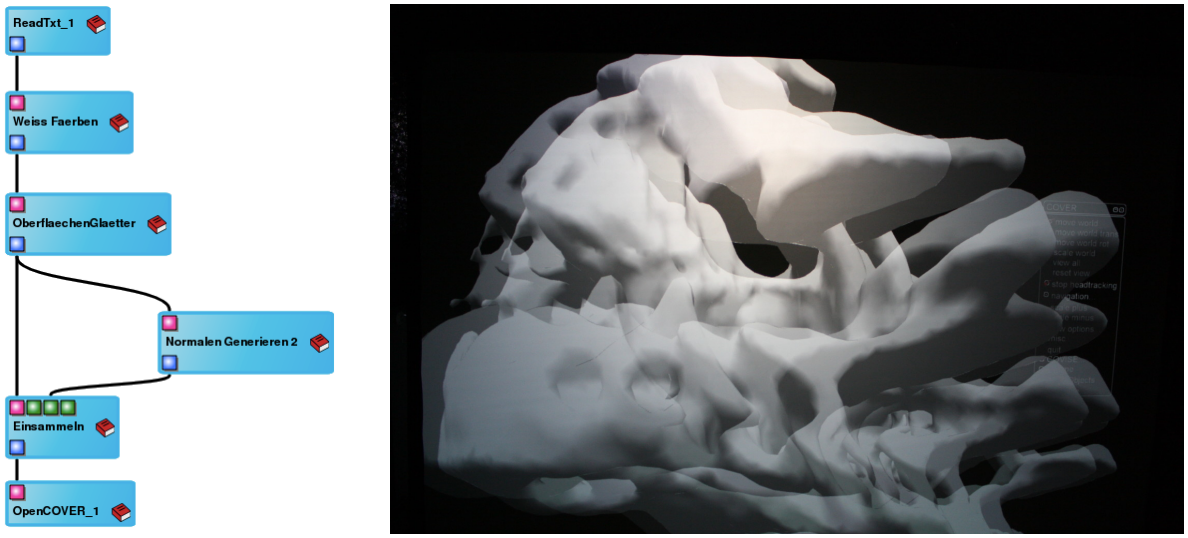
14.1 Visualisierung rekonstruierter Geometrien



(a) **(b)**

Abbildung 14.1: *Fotorealistische Darstellung eines rekonstruierten Keramikgefäßes (a) und einer rekonstruierten Halswirbelsäule (b).*

Eine interaktive Darstellung von Volumenbildern in Echtzeit ist zwar prinzipiell möglich, aber auf kleine Datenmengen beschränkt [32, 33]. Mikroskopaufnahmen mit einer Größe von mehreren Hundert Megabyte können auf diese Weise nicht mit einer akzeptablen Geschwindigkeit visualisiert werden. Die auf Dreiecken basierenden rekonstruierten Geometrien können hingegen in Echtzeit dargestellt und bewegt werden. Zudem können die erzeugten Geometrien in einem 3D-Modellierungsprogramm wie *Blender* [98] durch Raytracing und Radiosity Techniken [59] fotorealistisch gerendert werden (Abbildung 14.1).



(a) **(b)**

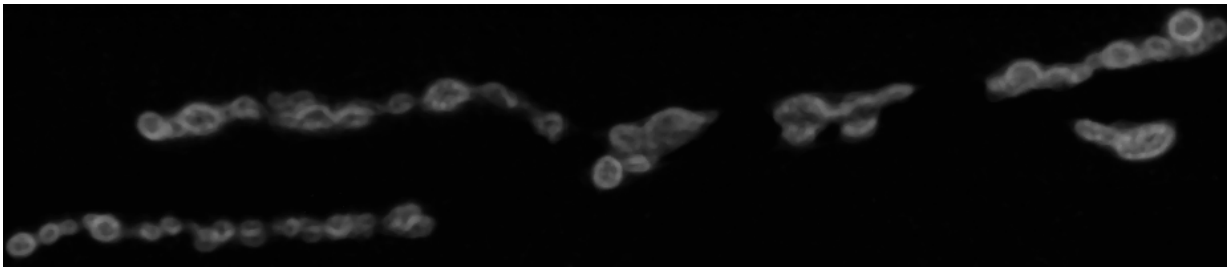
Abbildung 14.2: *(a)* Covise-*Modul zur dreidimensionalen Darstellung der rekonstruierten Geometrien. (b) Dreidimensionale Visualisierung einer rekonstruierten Geometrie auf einer Powerwall.*

Darüber hinaus wurde im Rahmen dieser Arbeit ein Modul für die Software *Covise* [8] entwickelt (Abbildung 14.2a), mit dessen Hilfe die rekonstruierten Geometrien dreidimensional dargestellt und interaktiv betrachtet werden können (Abbildung 14.2b).

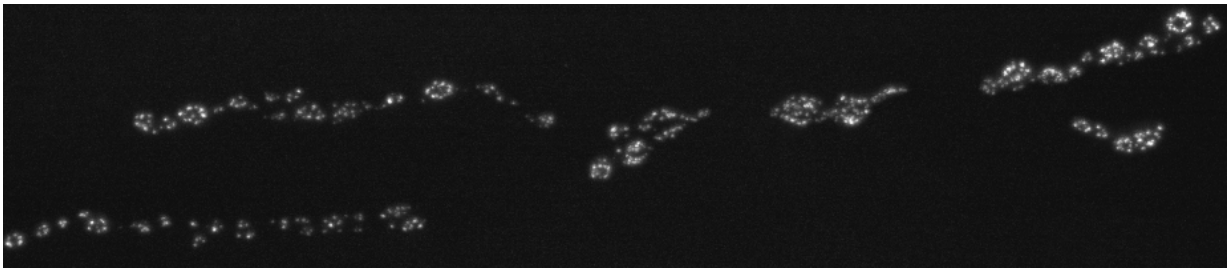
14.2 Rekonstruierte Geometrien als Grundlage für Simulationen

Die rekonstruierten Geometrien können verwendet werden, um verschiedenste Simulationen mit Simulationswerkzeugen wie *UG* [10] auf „echten“ Daten durchzuführen. Hierzu wird das durch *NeuRA2* erzeugte Oberflächengitter zunächst in ein aus Tetraedern bestehendes Volumengitter überführt, das dann als Ausgangsgitter für die Simulationen verwendet werden kann. Zur Generierung des Volumengitters ist die Software *TetGen* [105] an *NeuRA2* angekoppelt, deren Autor Hang Si an dieser Stelle recht herzlich gedankt sei. In einer früheren Arbeit [91] wurde bereits der Einfluss der Morphologie von Neuronenzellkernen auf die Ausbreitung von Calciumsignalen erfolgreich analysiert, wobei mit Hilfe von *NeuRA* rekonstruierte Geometrien eingesetzt wurden. Weitere durchgeführte und geplante Simulationen auf durch *NeuRA2* erzeugten Geometrien werden im Folgenden beschrieben.

14.2.1 Rekonstruktion von präsynaptischen Boutons



(a)



(b)

Abbildung 14.3: *(a) Axon eines Motoneurons mit seinen charakteristischen Boutons und (b) ihren aktiven Zonen (Volumenprojektionen). Quelle: AG Schuster, IZN Heidelberg*

Präsynaptische Boutons neuromuskulärer Synapsen bilden die Schnittstelle zwischen Nervensystem und Muskeln. Erreicht ein elektrisches Potential, ein sogenanntes Aktionspotential, am Ende eines Axons einen solchen Bouton, schüttet dieser in sogenannten Vesikeln befindliche Neurotransmitter in den synaptischen Spalt aus, die wiederum ein Aktionspotential in den Muskelzellen auslösen, das den Muskel kontrahiert. Um diese Vesikeldynamik

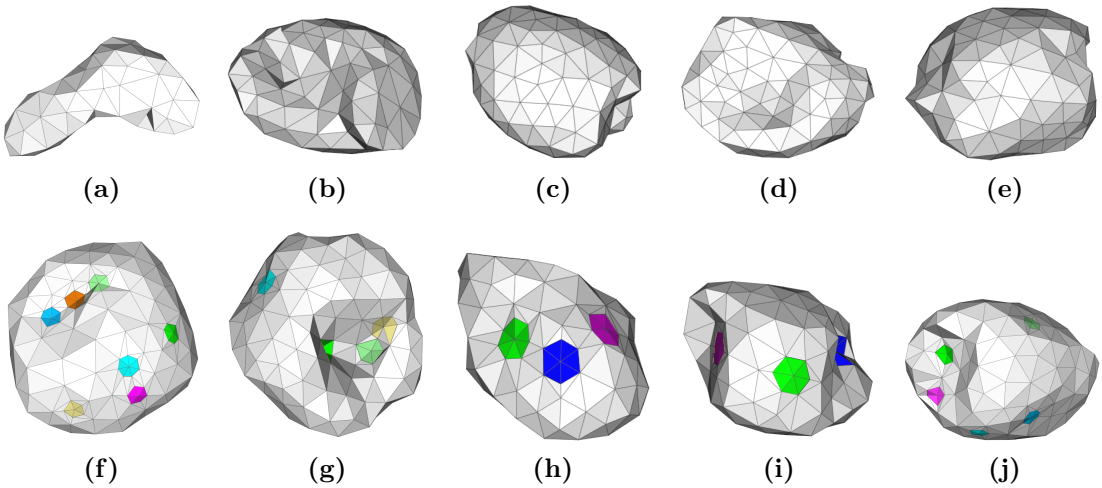


Abbildung 14.4: *Rekonstruktionen von zehn präsynaptischen Boutons. Bei den Rekonstruktionen (f) bis (j) sind die aktiven Zonen farbig markiert.*

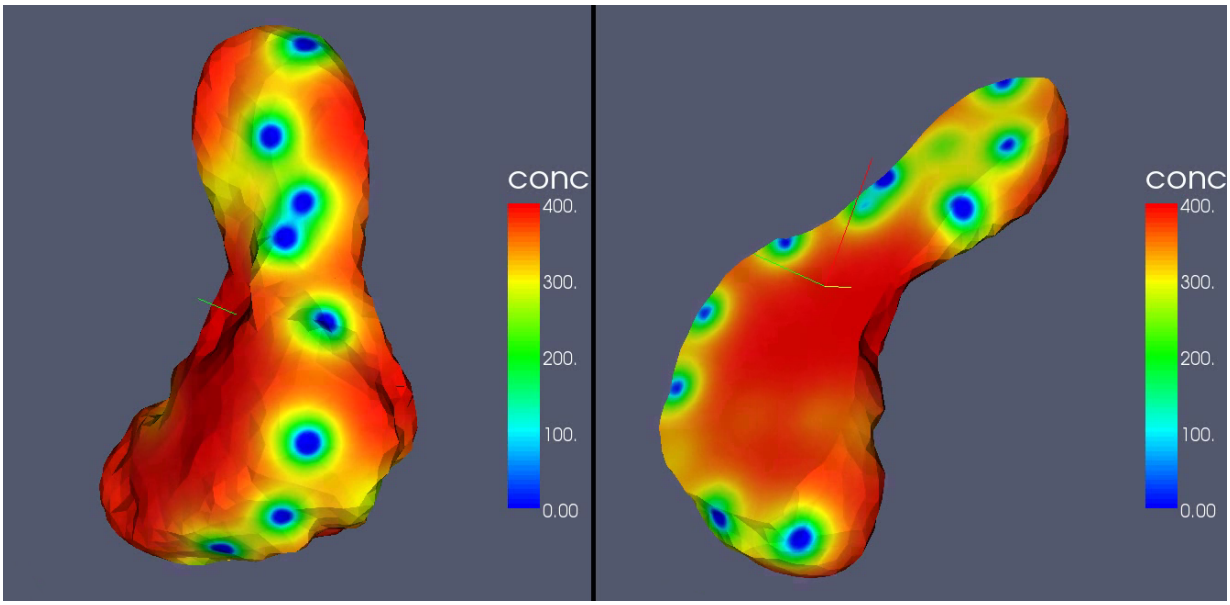


Abbildung 14.5: *Ergebnis der Simulation auf einem rekonstruierten Bouton (Abbildung 14.4a). Über die aktiven Zonen (blaue Kreise) werden die Vesikel ausgeschüttet. Quelle: G-CSC Frankfurt*

quantitativ untersuchen zu können wurden mit Hilfe von *NeuRA2* zehn solcher Boutons rekonstruiert (Abbildung 14.4), um die Simulationen auf echten Daten durchführen zu können. Ebenfalls wurden die aktiven Zonen, in denen die Vesikel mit der Zellmembran fusionieren, rekonstruiert und entsprechend in die Geometrien eingefügt (Abbildungen 14.4f bis j). Hierbei wurden Boutons einer Drosophilalarve mit Hilfe eines konfokalen Mikroskops aufgenommen. Das Ergebnis einer Simulation auf einem der rekonstruierten Boutons ist Abbildung 14.5 zu entnehmen. Die blauen Kreise entsprechen dabei den aktiven Zonen, über welche die Vesikel ausgeschüttet werden.

14.2.2 Rekonstruktion von Fibroblasten

In der Systembiologie wird die Funktionsweise von Signalwegen untersucht. Besonders bei Krebserkrankungen können gestörte Signalübertragungsmechanismen eine entscheidende Rolle spielen. Die dortige Forschung beschäftigt sich damit, wie aktivierte Signalmoleküle an Zielstrukturen in der Zelle (z.B. DNA im Zellkern) gelangen und wie die Konzentration dieser Signalmoleküle innerhalb der Zelle verteilt ist. Fibroblasten sind Zellen des Bindegewebes, die eine wichtige Rolle bei der Synthese der Interzellularsubstanz spielen. Die Zellen ändern bis zu ihrer Reifung die Form, die von Zelle zu Zelle sehr unterschiedlich sein kann (Abbildung 14.6). Besonders die langen zytoplasmatischen Fortsätze der Zellen beeinflussen die Konzentrationsverteilung der Signalmoleküle in der Zelle. Simulationen auf Rekonstruktionen solcher Zellen ermöglichen qualitative Aussagen über den Einfluss der Form der Zellen auf die Konzentrationsverteilung innerhalb der Zellen. Hierfür wurde mit Hilfe von *NeuRA2* die Oberfläche solcher Zellen und der zugehörigen Zellkerne als Dreiecksgitter rekonstruiert (Abbildung 14.7). Da die für diese Simulationen verwendete Software *Gascoigne* [12] auf Hexaedergittern arbeitet, wurde aus den durch *NeuRA2* gewonnenen Gitterpunkten mit Hilfe der Software *Ansys ICEM CFD* [5] ein Vierecks-Oberflächengitter (Abbildung 14.8a) und anschließend ein Hexaeder-Volumengitter (Abbildung 14.8b) eines Fibroblasten konstruiert. Die in [72] auf einer vereinfachten Modellgeometrie durchgeführten Simulationen zur Diffusion von *Stat5-Molekülen* aus dem *Jak2-Stat5-Signalweg*, der die Zellentwicklung kontrolliert und unter anderem für die Bildung roter Blutkörperchen verantwortlich ist, konnte mit Hilfe von *NeuRA2* auf realitätsnahen Geometrien (Abbildung 14.9) durchgeführt werden. Somit kopnnte in [24] der Einfluss der Morphologie solcher Zellen auf die Konzentrationsverteilungen der Signalmoleküle systematisch untersucht werden.

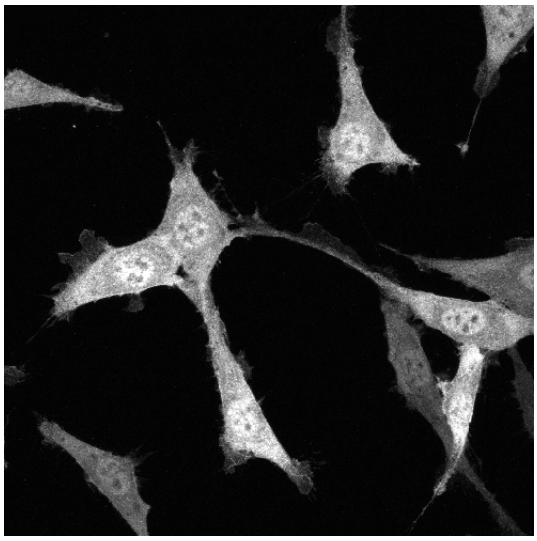
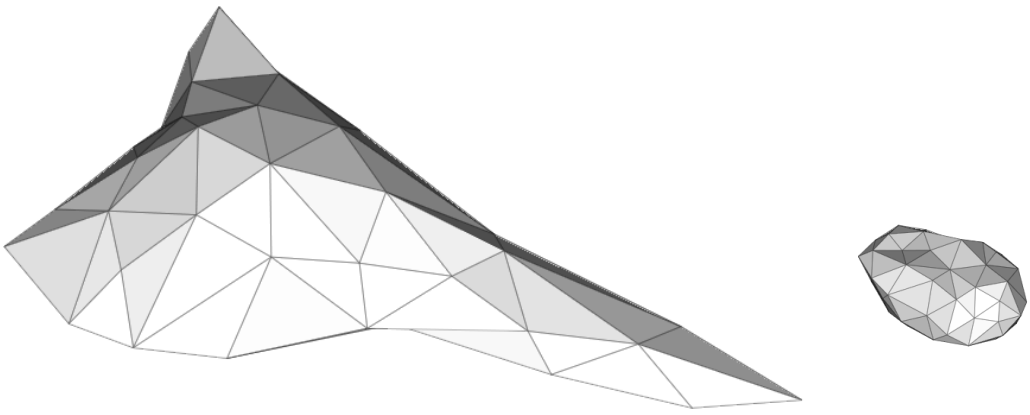
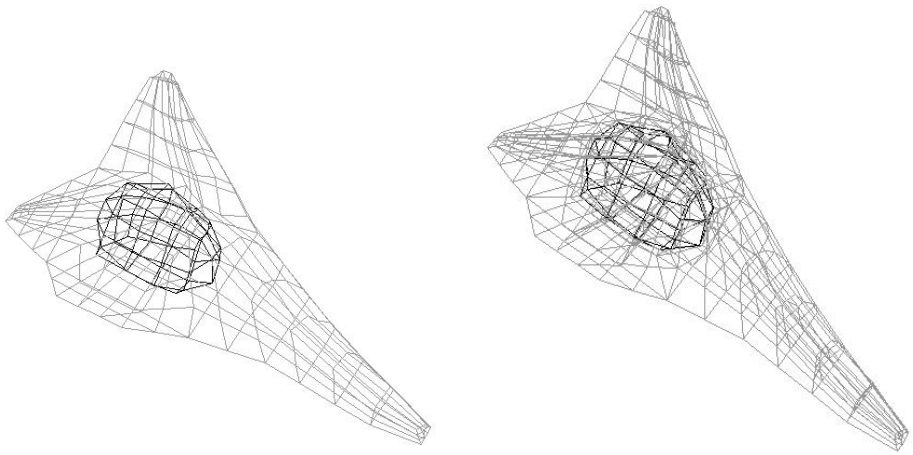


Abbildung 14.6: *Fibroblasten in verschiedenen Formen (Volumenprojektion). In der Mitte der Zellen sind etwas heller die Zellkerne zu erkennen. Quelle: DKFZ Heidelberg*



(a) **(b)**

Abbildung 14.7: *Rekonstruktion eines Fibroblasten. (a) Zelle. (b) Zellkern.*



(a) **(b)**

Abbildung 14.8: *Oberflächengitter (a) und Volumengitter (b) des rekonstruierten Fibroblasten mit innen liegendem Zellkern.*

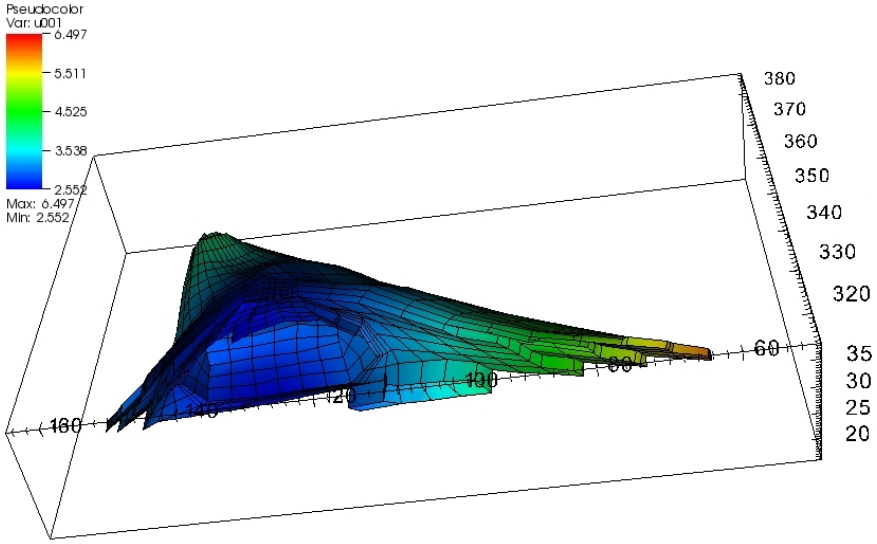
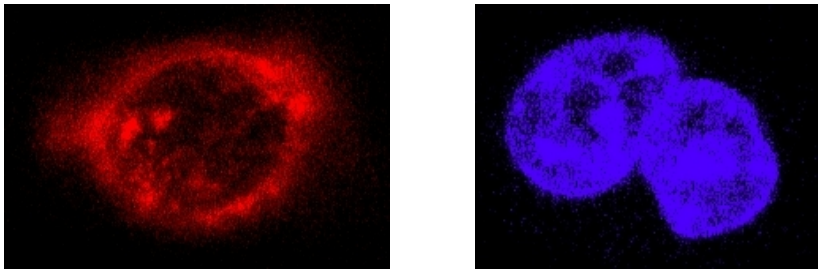


Abbildung 14.9: *Simulation auf einem rekonstruierten Fibroblasten: Räumliche Verteilung von pStat in der Fibroblastenzelle nach $t = 30min$. In den Ausläufern liegt eine Konzentration von 6.5 Moleküle/μm^3 pStat vor, bis zum Zellkern sinkt die Konzentration auf 2.5 Moleküle/μm^3. Quelle: Simulation AG Rannacher, IWR Heidelberg*

14.2.3 Rekonstruktion von Leberzellen

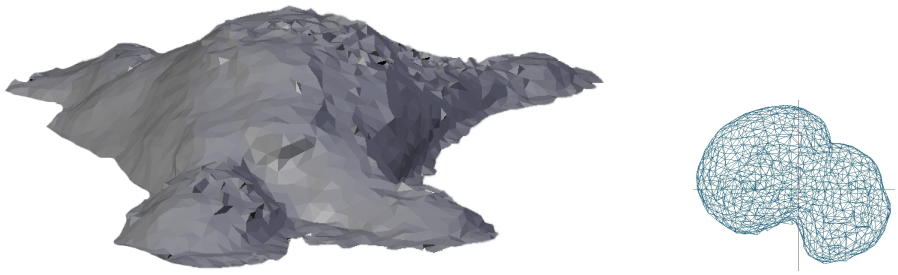
Die Leber ist das zentrale Organ des Stoffwechsels und vor allem für den Abbau und die Ausscheidung von Stoffwechselprodukten zuständig. Für die Modellierung von räumlicher und zeitlicher Anreicherung und dem Abbau dieser Stoffe ist es von großer Bedeutung, Organellen und Membranen von Leberzellen zu rekonstruieren, um diese Prozesse auf echten Zellen untersuchen zu können.



(a) **(b)**

Abbildung 14.10: *Konfokale Aufnahme einer Leberzelle (Schnittebenenbilder). (a) Äußere Zellmembran. (b) Zellkern. Quelle: Institute of Materials Science, TU Dresden*

Die zu rekonstruierenden Bereiche können mit Hilfe verschiedener Farbstoffe unter einem konfokalen Mikroskop sichtbar gemacht (Abbildung 14.10) und anschließend mit *NeuRA2* rekonstruiert (Abbildung 14.11) werden.

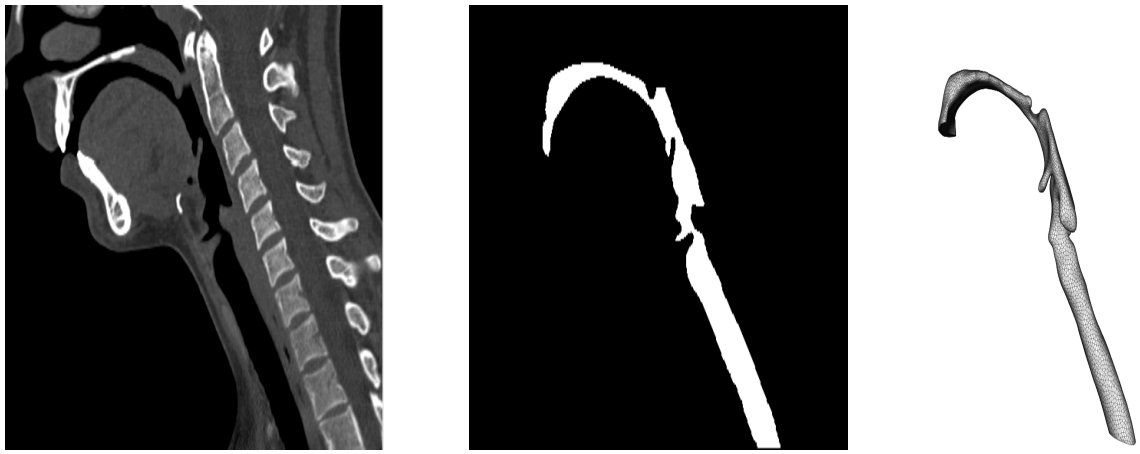


(a) **(b)**

Abbildung 14.11: *Rekonstruktion der Leberzelle. (a) Äußere Zellmembran. (b) Zellkern. Quelle: Dep. Computational Hydrosystems, Helmholtz Center for Environmental Research, Leipzig*

Zusätzliche Parameter wie Diffusions- und Reaktionsraten können mittels Fluorescence Recovery After Photobleaching (FRAP) Experimenten [70, 106, 107] bestimmt werden. Diese Konstanten und die zuvor rekonstruierten Gittergeometrien können nun verwendet werden, um Simulationen mit Hilfe von *OpenGeoSys (OGS)* [15] durchzuführen, was eine sowohl räumlich als auch zeitlich hochaufgelöste Visualisierung und Analyse solcher biochemischen Prozesse ermöglicht.

14.2.4 Rekonstruktion einer Luftröhre



(a) **(b)** **(c)**

Abbildung 14.12: *(a) Computertomographie-Aufnahme des unteren Schädelbereichs und des Halses eines Menschen. Quelle: Universitätsklinikum Heidelberg. (b) Segmentierung der Luftröhre und des unteren Rachenbereichs (Schnittebenenaufnahmen). (c) Rekonstruktion des unteren Rachenbereichs und der Luftröhre als Oberflächengitter.*

Bei der Behandlung von Asthma und anderen Lungenkrankheiten werden pharmazeutische Wirkstoffe häufig mit Hilfe von Aerosolen über die Luftröhre in die Lunge transportiert. Durch die Software *OpenFOAM 1.5* [84] ist in [25] die Ausbreitung solcher Pharmazeutika auf einer Modellgeometrie von Mund- und Rachenraum simuliert worden. Die vom Universitätsklinikum Heidelberg bereitgestellte Computertomographie-Aufnahme des unteren Schädelbereichs und des Halses eines Menschen (Abbildung 14.12a) wurde mit Hilfe von *NeuRA2* segmentiert (Abbildung 14.12b) und rekonstruiert (Abbildung 14.12c). Unter Verwendung des rekonstruierten Gitters kann diese Simulation zukünftig auf einer realistischen Geometrie durchgeführt werden.

14.3 Anwendungen in der Archäologie

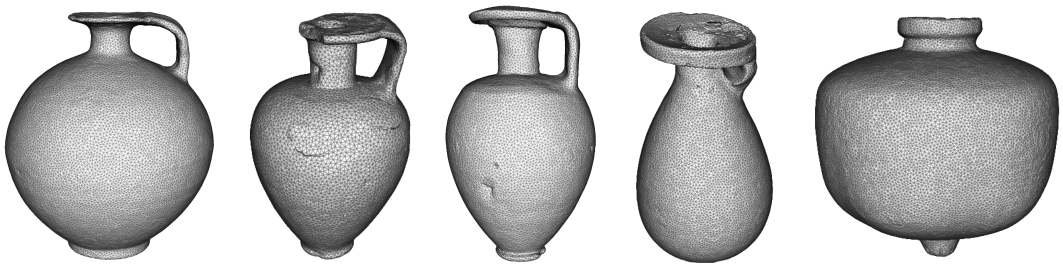


Abbildung 14.13: *Rekonstruktionen der protokorinthischen Keramikgefäße (a) CT03155, (b) CT01184, (c) CT03145, des korinthischen Alabastrons (d) CT03214 und des ostgriechischen Granatapfelgefäßes (e) CT03047.*

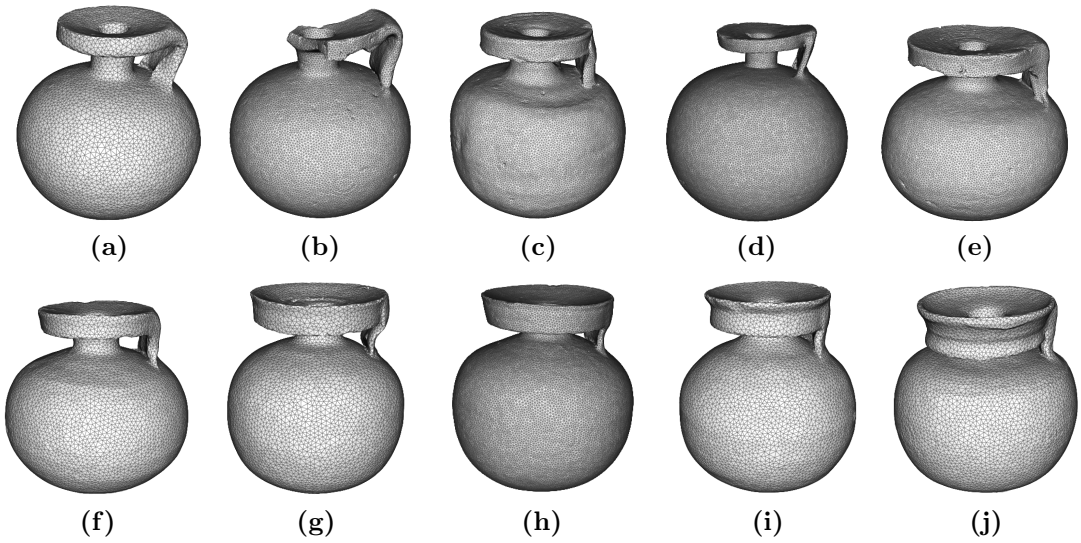
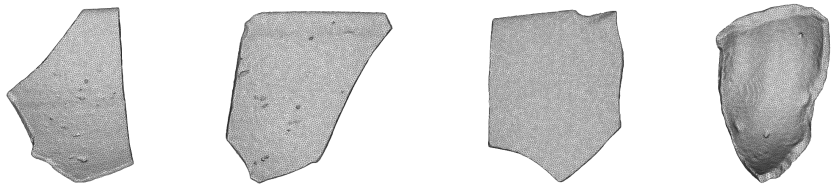


Abbildung 14.14: *Dem Alter nach sortierte Rekonstruktionen korinthischer Keramikgefäße (a) CT02187, (b) CT03211, (c) CT03212, (d) CT04136, (e) CT03213, (f) CT02186, (g) CT02185, (h) CT03210, (i) CT02183, (j) CT02184.*

Die hier aufgeführten Ergebnisse sind bereits teilweise in [51, 56, 57] veröffentlich worden bzw. befinden sich im Druck. Die im Rahmen dieser Arbeit rekonstruierten und untersuchten Keramikgefäße (Abbildungen 14.13 und 14.14) und Keramikfragmente (Abbildung 14.15) stammen aus den Sammlungen des *Universalmuseums Joanneum* in Graz und dem *Institut für Archäologie* der *Karl-Franzens-Universität* in Graz. Sie wurden mit Hilfe eines industriellen Computertomographen am *Österreichischen Gießerei Institut* (ÖGI) in Leoben digitalisiert und mit Scan-Nummern (CTXXXXX) versehen. Alle Scans wurden mit einer isotropen Auflösung zwischen 0.15mm und 0.28mm pro Voxel gescannt [56]. Zudem wurden die zwei modernen Testgefäße (CT03244 und CT03245, vgl. Kapitel 11.4) gescannt,



(a) **(b)** **(c)** **(d)**

Abbildung 14.15: *Rekonstruktionen verschiedener Keramikfragmente (a) CT02188-1, (b) CT02188-4 nordionischer, (c) CT02188-2 attischer und (d) CT02188-3 korinthischer Herkunft.*

um die Volumenbestimmung der Keramiken zu kalibrieren. Die Verknüpfung zwischen den Scan-Nummern und den Objektidentifizierungsnummern der archäologischen Sammlungen ist in Tabelle C.1 im Anhang dieser Arbeit zu finden. Die mit Hilfe von *NeuRA2* rekonstruierten Computertomographie-Aufnahmen dieser Keramikgefäße können auf vielfältige Weise verwendet werden: Unter anderem können Untersuchungen an den Objekten vorgenommen werden ohne die Originalexponate zu berühren oder die digitalisierten Objekte können über webbasierte Datenbanken weltweit zur Verfügung gestellt werden.

Objekt	**Füllvolumen CT**	**Füllvolumen Reis**	**Abweichung**
CT01184	45ml	44ml	2%
CT02183	123ml	nicht möglich	–
CT02184	108ml	112ml	4%
CT02185	102ml	106ml	4%
CT02186	81ml	84ml	4%
CT02187	56ml	58ml	4%
CT03047	86ml	nicht möglich	–
CT03145	36ml	36ml	0%
CT03155	74ml	76ml	3%
CT03210	110ml	114ml	4%
CT03211	61ml	62ml	2%
CT03212	104ml	107ml	3%
CT03213	56ml	58ml	4%
CT03214	40ml	39ml	3%
CT03244	92ml	92ml	0%
CT03245	65ml	67ml	3%
CT04136	244ml	252ml	3%

Tabelle 14.1: *Volumenbestimmung in der Archäologie: Das Füllvolumen der rekonstruierten Gefäße.*

Objekt	Masse	Keramikvolumen	Rohdichte
CT01184	47.1g	27ml	1.7g/ml
CT02184	82g	44ml	1.9g/ml
CT02185	71.6g	40ml	1.8g/ml
CT02186	70.9g	42ml	1.7g/ml
CT02187	63.4g	37ml	1.7g/ml
CT02188-1	18.6g	10ml	1.9g/ml
CT02188-2	7.8g	4ml	2g/ml
CT02188-3	11.0g	6ml	1.8g/ml
CT02188-4	21.2g	11ml	1.9g/ml
CT03047	72.7g	55ml	1.3g/ml
CT03145	45.7g	25ml	1.8g/ml
CT03155	37.8g	23ml	1.6g/ml
CT03210	68.6g	37ml	1.9g/ml
CT03211	52.9g	30ml	1.8g/ml
CT03212	97.6g	54ml	1.8g/ml
CT03213	59g	32ml	1.8g/ml
CT03214	47.6g	27ml	1.8g/ml
CT03244	214.1g	110ml	1.9g/ml
CT03245	96g	49ml	2.0g/ml
CT04136	128.8g	71ml	1.8g/ml

Tabelle 14.2: *Volumenbestimmung in der Archäologie: Keramikvolumen und Rohdichte der rekonstruierten Gefäße.*

Keramiktyp	Rohdichte
Ostgriechisch	ca. 1.3g/ml
Protokorinthisch	ca. $1.6 - 1.7$g/ml
Korinthisch	ca. $1.7 - 1.9$g/ml
Nordionisch	ca. 1.9g/ml
Attisch	ca. 2g/ml
Modern	ca. $1.9 - 2$g/ml

Tabelle 14.3: *Die Rohdichte der rekonstruierten Keramiken ist je nach Keramiktyp unterschiedlich.*

(a) **(b)**

Abbildung 14.16: *Die digitale Entfernung von mitgebrannten Tonklumpen ermöglicht die Berechnung des Füllvolumens von Gefäß CT02183. (a) Schnittebenendarstellung der segmentierten CT-Aufnahme. (b) Nach Entfernung der Tonklumpen.*

14.3.1 Volumenberechnung

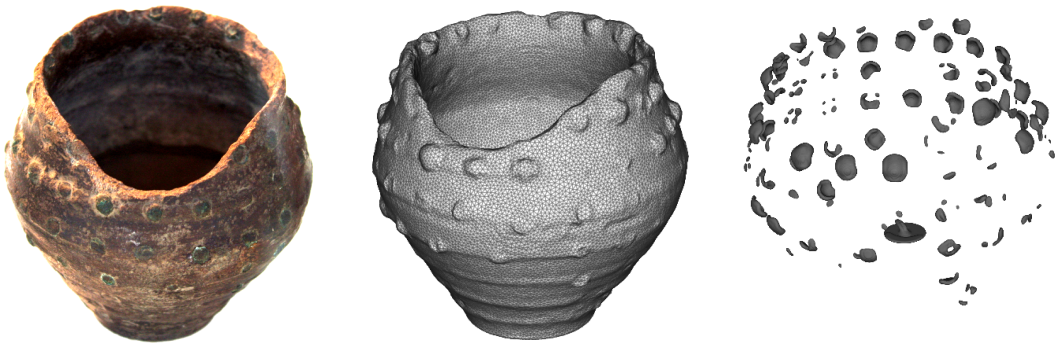
Viel wichtiger ist es jedoch Keramikvolumen und Kapazität der rekonstruierten Gefäße automatisch bestimmen zu können. Die Berechnung des Volumens der Keramik ermöglicht die Berechnung der physikalischen Dichte der Objekte, woraus sich Rückschlüsse auf die verwendeten Materialien, unterschiedliche Töpfertechniken und den Keramiktyp der Gefäße ziehen lassen. Zudem lassen sich durch die Berechnung der Kapazität der Gefäße Aussagen über die verwendeten Maßeinheiten in unterschiedlichen Zeitepochen treffen. Mit Hilfe des in Kapitel 11.4 kalibrierten Verfahrens zur Volumenbestimmung in der Archäologie wurden mit *NeuRA2* siebzehn durch Computertomographie aufgenommene Keramikgefäße rekonstruiert und jeweils ihr Füllvolumen (Tabelle 14.1) und ihr Keramikvolumen einschließlich Luftporen (Tabelle 14.2) berechnet. Da die Gefäße unter Normalbedingungen sehr exakt gewogen werden können, lässt sich mit Hilfe des Keramikvolumens die sogenannte *Rohdichte*

$$\text{Rohdichte} = \frac{\text{Masse der Keramik}}{\text{Volumen inklusive Lufteinschlüsse}} \tag{14.1}$$

berechnen, die ebenfalls in Tabelle 14.2 aufgeführt ist. Dabei zeigt sich eine Abhängigkeit der Rohdichte vom verwendeten Keramiktyp. Diese in Tabelle 14.3 aufgeführten Zahlen sind allerdings mit einer gewissen Vorsicht zu betrachten, da mit Ausnahme von korinthischen Gefäßen (die Typen der einzelnen Keramiken findet sich in Tabelle C.1 im Anhang

dieser Arbeit) nur wenige Rekonstruktionen und damit Messwerte für die Rohdichte vorliegen. Für die Zukunft wäre es interessant diesen Zusammenhang weiter zu untersuchen. Dennoch lassen sich somit weitere Informationen zu antiken Gefäßen sammeln und unbekannte Gefäße mit Hilfe weiterer Charakteristika Zeitepochen oder einem bestimmten Keramiktyp zuweisen. Bei genauer Betrachtung von Tabelle 14.1 fällt auf, dass bei Gefäß CT02183 der mit der Reismethode bestimmte Volumenwert fehlt. Dieses Gefäß enthält mehrere mitgebrannte Tonklumpen, die wahrscheinlich beim Töpferprozess in das Gefäß gefallen sind (Abbildung 14.16a). Da die Entfernung dieser Tonklumpen das Gefäß beschädigen würde, lässt sich das Volumen dieser Keramik mit den üblichen Methoden aus der Archäologie nicht bestimmen. Entfernt man diese Tonklumpen mit Hilfe von *NeuRA2* digital (Abbildung 14.16b), lässt sich durch das in dieser Arbeit vorgestellte CT-Verfahren das Volumen des Gefäßes jedoch bestimmen. Da eine übliche archäologische Volumenbestimmung des ostgriechischen Granatapfelgefäßes CT03047 ebenfalls nur unter Beschädigung des Gefäßes möglich wäre, fehlt der entsprechende Wert in Tabelle 14.2 ebenfalls. Durch Anwendung der CT-Methode lässt sich das Volumen dieses Gefäßes berechnen. Abschließend bleibt zu bemerken, dass der durch die Reismethode bestimmte Wert häufig den CT-Volumenwert übertrifft. Dieses Phänomen wurde bereits bei dem schalenförmigen Testgefäß (vgl. Kapitel 11.4) beobachtet und lässt sich darauf zurückführen, dass sich der Reis in einem Gefäß mit großer Grundfläche besser verdichtet, als in dem Messzylinder zur Volumenbestimmung.

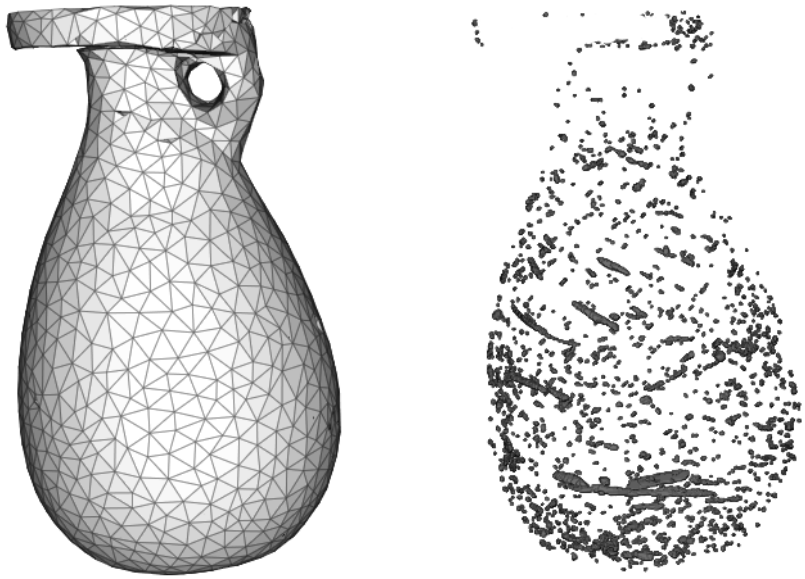
14.3.2 Visualisierung von Charakteristika



(a) **(b)** **(c)**

Abbildung 14.17: *(a) Foto von Gefäß CT03046 mit grünen Bronzeapplikationen. (b) Rekonstruktion. (c) Rekonstruktion der Metallbeschläge.*

Mit Hilfe der rekonstruierten CT-Aufnahmen können verschiedene Charakteristika der Keramiken visualisiert werden. Beispielsweise ist Gefäß CT03046, ein der Este-Kultur entstammender Becher (Abbildung 14.17a), mit grünen Bronzeapplikationen verziert. Bei äußerer Betrachtung des Gefäßes ist nicht erkennbar, ob diese Beschläge flache, gewölbte Blättchen sind oder die Form von Nägeln aufweisen. Durch geeignete Schwellwertsetzung



(a) **(b)**

Abbildung 14.18: *(a) Rekonstruktion des Alabastrons CT03214. (b) Rekonstruktion der in der Keramik enthaltenen Luftporen, die im Uhrzeigersinn spiralförmig nach oben gehen.*

bei der Segmentierung können die Metallbeschläge separat rekonstruiert werden, wodurch ihre flache Form erkennbar wird (Abbildung 14.17c). Darüber hinaus können, ebenfalls durch geeignete Schwellwertsetzung bei der Segmentierung, die in den Keramiken vorhandenen Luftporen sichtbar gemacht werden, was exemplarisch für das Alabastron CT03214 in Abbildung 14.18 zu sehen ist. Hierbei wird die beim Töpferprozess verwendete Richtung der Drehscheibe gegen den Uhrzeigersinn sichtbar und legt, wie bei allen auf diese Weise untersuchten Gefäßen, einen Rechtshänder als Töpfer nahe.

14.4 Anwendungen in den Neurowissenschaften

14.4.1 Vermessung von Neuronenzellkernen

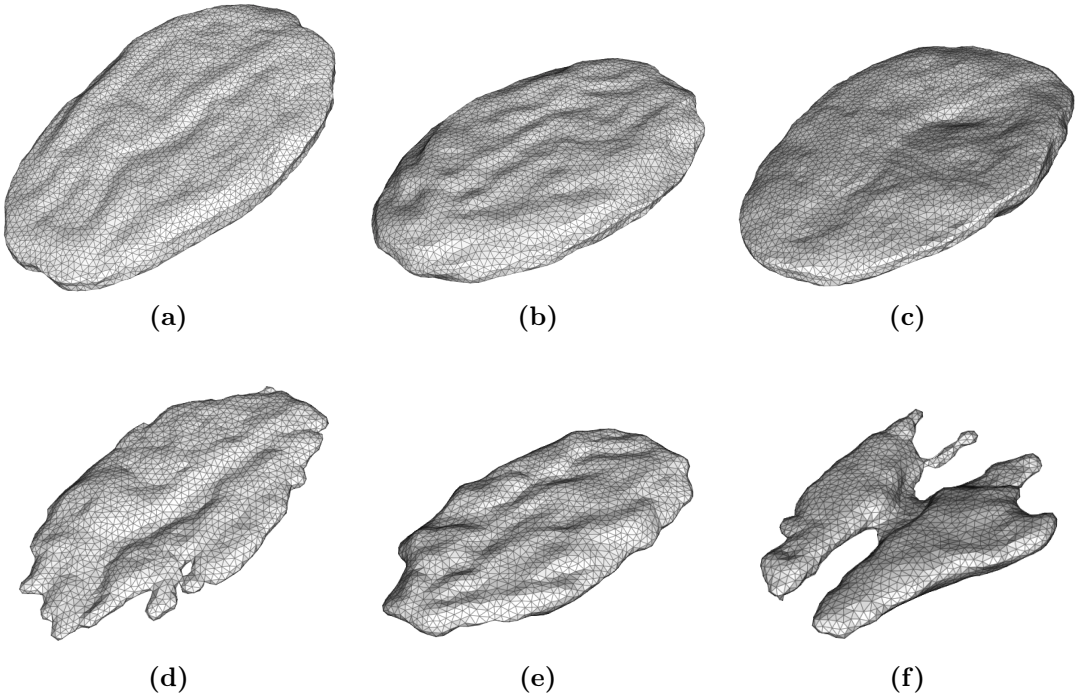


Abbildung 14.19: *Mit* NeuRA2 *rekonstruierte Zellkerne: (a), (b), (c) Äußere Membran. (d), (e), (f) Innere Membran.*

Durch die mit *NeuRA* systematisch gewonnenen Rekonstruktionen von Neuronenzellkernen konnte in [90, 92] entgegen der vorher üblichen Meinung, diese besäßen eine kugelförmige Struktur, gezeigt werden, dass die Zellkerne von Neuronenzellen *Invaginationen* oder *Einstülpungen* der Zellkernmembran besitzen. In [90] wurde zudem der Einfluss dieser Einstülpungen auf das Verhältnis zwischen Oberfläche und Volumen der Neuronenzellkerne ausführlich untersucht. Ergebnis dieser Untersuchungen ist, dass das Volumen-zu-Oberflächen-Verhältnis bei Zellkernen mit Einstülpungen um etwa zwanzig Prozent kleiner ist, als bei Zellkernen, die keine Einstülpungen aufweisen. Mit Hilfe der in *NeuRA2* enthaltenen parallelen Implementierung des klassischen Neuronen-Rekonstruktions-Algorithmus sind Rekonstruktion und Untersuchung solcher Zellkerne nun in wenigen Minuten möglich. Abbildung 14.19 zeigt exemplarisch drei mit *NeuRA2* rekonstruierte Zellkerne.

14.4.2 Untersuchung morphologischer Veränderungen in Neuronenzellen

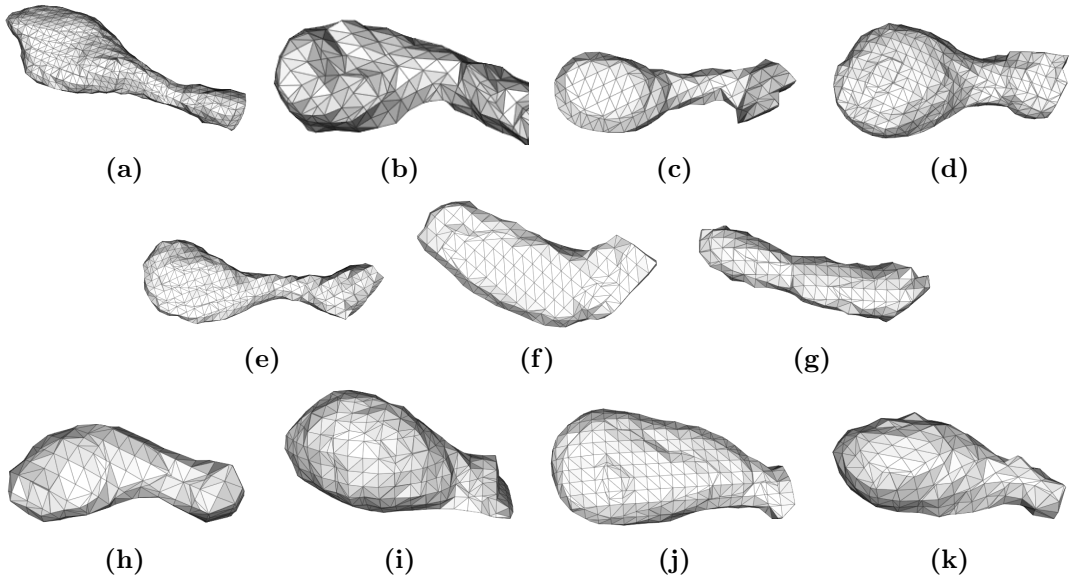


Abbildung 14.20: *Die Oberflächenrekonstruktion einzelner Spines ermöglicht die Bestimmung deren Volumina.*

Durch die Analyse morphologischer Veränderungen in lebenden Neuronenzellen kann der Einfluss von pharmakologischen Präparaten oder mechanischen Eingriffen in das Nervensystem von Versuchstieren systematisch untersucht werden. Durch solche Manipulationen an Mäusegehirnen können sowohl Umgestaltungen von Dendriten als auch von dendritischen Dornfortsätzen beobachtet werden [69, 112]. Mit Hilfe von *SpineLab* kann die Morphologie von mikroskopierten Neuronenzellen auf diesen beiden Ebenen folgendermaßen analysiert werden:

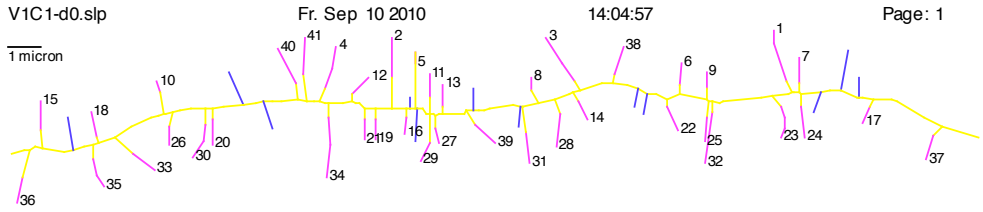
1. Semiautomatische Rekonstruktion einzelner Dendritensegmente
   (Kapitel 13.3.1 und Anhang B.1):
   - Automatische Analyse der Spinepopulation
   - Automatische Längenanalyse der rekonstruierten Spines
   - Automatische Bestimmung der durchschnittlichen Spinelänge
   - Automatische Bestimmung der aufsummierten Spinelänge
   - Semiautomatische Volumenanalyse einzelner Spines
2. Semiautomatische Rekonstruktion ganzer Neuronenzellen
   (Kapitel 13.3.2 und Anhang B.2):

- Automatische Analyse der Länge einzelner Dendriten
- Automatische Bestimmung der Länge von Dendritenabschnitten
- Automatische Analyse der Winkel bei Dendritenverzweigungen
- Automatische Bestimmung der summierten Dendritenlänge

Wie in den vorhergehenden Kapiteln beschrieben, ist die Bildqualität der Lebendaufnahmen häufig sehr schlecht, weshalb eine vollautomatische Rekonstruktion dieser Daten kaum möglich ist. Daher ist in *SpineLab* eine manuelle Nachbearbeitung der automatisch generierten Merkmalskelette integriert. Um die Volumenanalyse einzelner Spines durchführen zu können, wird deren Oberfläche, wie in Kapitel 13.3.1 beschrieben, als Dreiecksgitter rekonstruiert. Einige der auf diese Weise rekonstruierten Spines sind Abbildung 14.20 zu entnehmen.

Durch *SpineLab* wurden einige Dendritensegmente semiautomatisch rekonstruiert, sowie die Anzahl an Spines und deren Länge bestimmt. Den Abbildungen 14.21 bis 14.25 sind exemplarisch fünf solcher Rekonstruktionen zu entnehmen. Hierbei werden durchschnittlich etwa 60% bis 70% aller Spines automatisch erkannt, vor allem solche, die in x- und y-Richtung ausgeprägt sind. In die schlechter aufgelöste z-Richtung orientierte Spines werden vom Rekonstruktions-Algorithmus hingegen häufig übersehen und müssen manuell hinzugefügt werden.

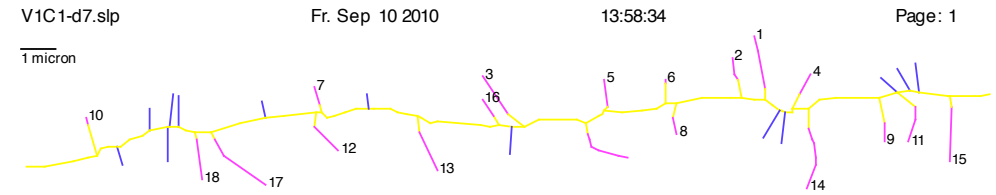
In den Abbildungen 14.26 bis 14.34 sind die Dendrogramme von fünf rekonstruierten Neuronenzellen zu sehen. Alle fünf Zellen wurden mit Hilfe von *SpineLab* und den beiden Softwarepaketen *Neurolucida* [16] und *NeuronStudio* [96, 113] (Abbildung 14.35) rekonstruiert. Die Längen der einzelnen rekonstruierten Dendritensegmente und die insgesamte Länge aller Dendriten einer einzelnen Zelle wurden zwischen den verschiedenen Rekonstruktionen verglichen. Hierbei zeigt sich, dass *SpineLab* und *NeuronStudio* bei einer statistischen Abweichung von maximal fünf Prozent identische Ergebnisse liefern. Die von *Neurolucida* berechneten Dendritenlängen sind allerdings zwischen sieben und fünfzehn Prozent kürzer, als die mit *SpineLab* oder *NeuronStudio* berechneten. Vermutlich wird bei *SpineLab* und *NeuronStudio* der Verlauf der Dendriten in der schlecht aufgelösten z-Richtung exakter rekonstruiert, weshalb die einzelnen Dendriten insgesamt länger werden. Bei der systematischen Untersuchung morphologischer Veränderungen spielen diese Abweichungen allerdings eine untergeordnete Rolle, wenn alle auszuwertenden Daten mit derselben Software rekonstruiert werden. Eine vollständig per Hand durchgeführte Rekonstruktion einer Neuronenzelle dauert bei allen getesteten Softwarepaketen ungefähr zehn Minuten. Die in *SpineLab* integrierte semi-automatische Rekonstruktionsmethode (Kapitel B.2) verringert den Aufwand jedoch um einige Minuten. Weitere Ergebnisse, die mit *SpineLab* erzielt wurden sowie eine ausführliche Validierung der Ergebnisse sind in [53] beschrieben.



VoxelSizeX = 0.0714102 VoxelSizeY = 0.0714102 VoxelSizeZ = 0.464 TotalSpineLength = 38.8755 AverageSpineLength = 0.948182
NumberOfSpines = 53 SpinesWithLength = 41

1: 2.09969
2: 1.37684
3: 1.62473
4: 1.4605
5: 0.35423
6: 0.730804
7: 0.984649
8: 0.430092
9: 0.727009
10: 0.355256
11: 0.81865
12: 0.770565
13: 0.782732
14: 0.794569
15: 1.0694
16: 0.594665
17: 0.522567
18: 0.802988
19: 0.71018
20: 0.836069
21: 0.71012
22: 0.885297
23: 1.00295
24: 0.991804
25: 0.746302
26: 0.579793
27: 0.498985
28: 1.00422
29: 0.770506
30: 1.13671
31: 1.20868
32: 1.55853
33: 0.819044
34: 1.87509
35: 0.987254
36: 0.809951
37: 0.930435
38: 0.911968
39: 0.851774
40: 1.51631
41: 1.23358

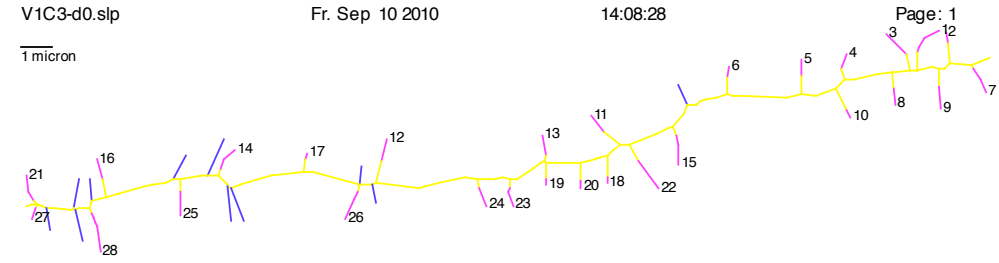
Abbildung 14.21: *Dendrogramm eines rekonstruierten Dendritensegments.*



VoxelSizeX = 0.0714102 VoxelSizeY = 0.0714102 VoxelSizeZ = 0.531294 TotalSpineLength = 20.3316 AverageSpineLength = 1.12953
NumberOfSpines = 31 SpinesWithLength = 18

1: 2.52235
2: 0.660857
3: 1.7748
4: 0.83286
5: 0.77755
6: 0.125371
7: 0.582881
8: 0.538393
9: 0.532866
10: 0.535257
11: 1.12758
12: 1.07886
13: 1.39743
14: 2.12896
15: 1.76791
16: 0.647576
17: 2.04527
18: 1.25481

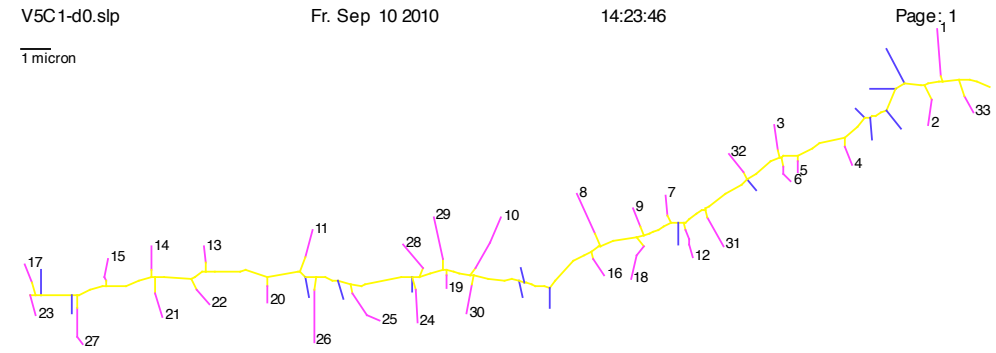
Abbildung 14.22: *Dendrogramm eines rekonstruierten Dendritensegments.*



VoxelSizeX = 0.0714102 VoxelSizeY = 0.0714102 VoxelSizeZ = 0.464 TotalSpineLength = 18.2954 AverageSpineLength = 0.653407
NumberOfSpines = 39 SpinesWithLength = 28

1: 1.24698
2: 0.416593
3: 0.860603
4: 0.49765
5: 0.490023
6: 0.373869
7: 0.821407
8: 0.638186
9: 0.719489
10: 0.291698
11: 0.632605
12: 0.796541
13: 0.664334
14: 0.88634
15: 0.923208
16: 0.649791
17: 0.16425
18: 0.197913
19: 0.198085
20: 0.232782
21: 0.849439
22: 1.20457
23: 0.55379
24: 0.594703
25: 0.749215
26: 0.914539
27: 0.439797
28: 1.28699

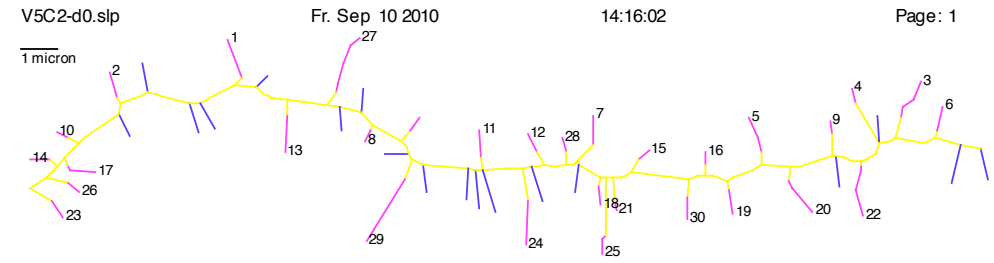
Abbildung 14.23: *Dendrogramm eines rekonstruierten Dendritensegments.*



VoxelSizeX = 0.0714102 VoxelSizeY = 0.0714102 VoxelSizeZ = 0.448 TotalSpineLength = 32.0262 AverageSpineLength = 0.970491
NumberOfSpines = 48 SpinesWithLength = 33

1: 1.57191
2: 0.818436
3: 0.813137
4: 0.69059
5: 0.391995
6: 0.617812
7: 0.660264
8: 1.43486
9: 0.812354
10: 1.9198
11: 1.11012
12: 1.09816
13: 0.631394
14: 0.719629
15: 0.849239
16: 0.630228
17: 0.673603
18: 1.51418
19: 0.428285
20: 0.489347
21: 0.834567
22: 0.689244
23: 0.66321
24: 1.09197
25: 1.43449
26: 1.90299
27: 1.20215
28: 1.166
29: 1.44904
30: 1.06884
31: 1.18447
32: 0.826292
33: 0.637592

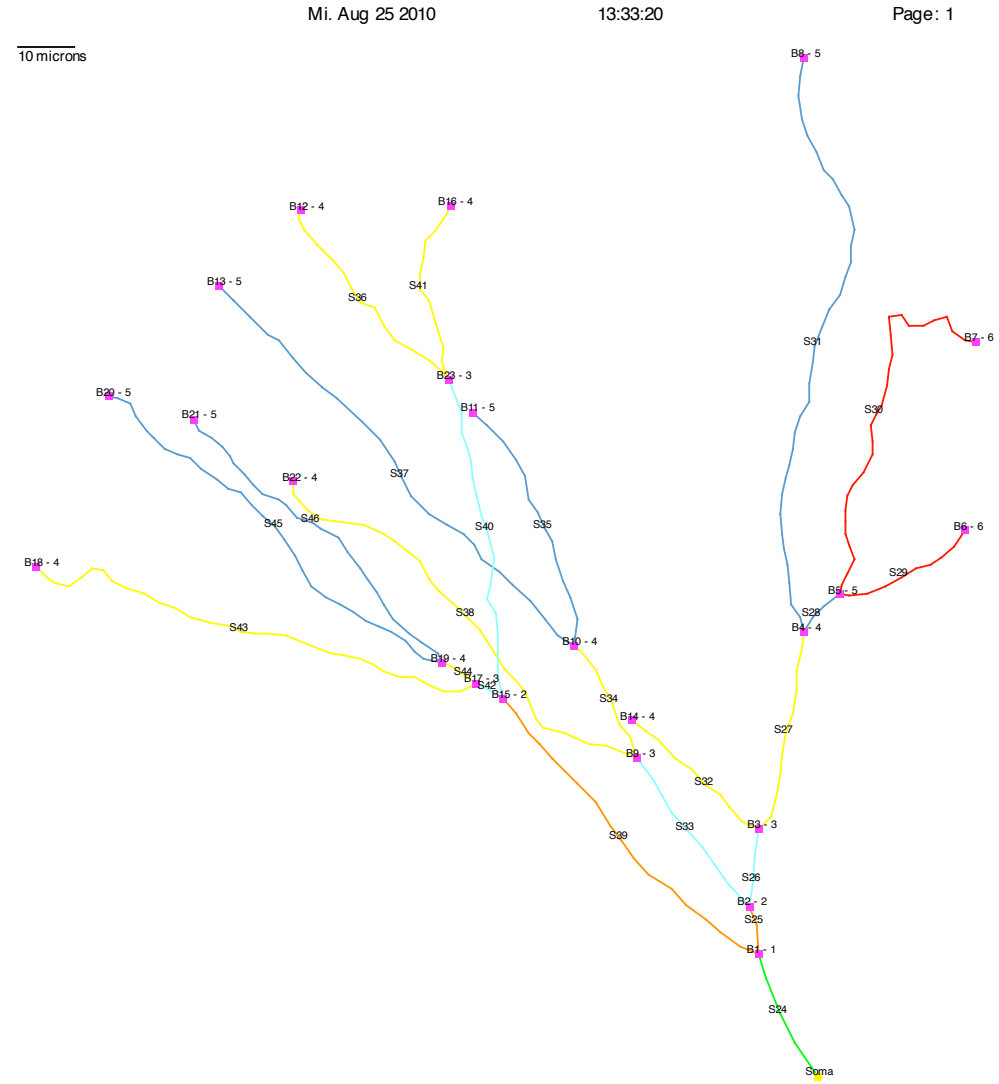
Abbildung 14.24: *Dendrogramm eines rekonstruierten Dendritensegments.*



VoxelSizeX = 0.0714102 VoxelSizeY = 0.0714102 VoxelSizeZ = 0.448 TotalSpineLength = 25.153 AverageSpineLength = 0.838435
NumberOfSpines = 48 SpinesWithLength = 30

1: 1.38914
2: 0.75884
3: 1.21358
4: 0.428449
5: 1.30523
6: 0.648761
7: 1.44167
8: 0.348793
9: 0.438917
10: 0.413559
11: 0.784044
12: 0.519216
13: 1.01832
14: 0.554614
15: 0.434845
16: 0.356672
17: 0.957419
18: 0.605767
19: 0.668739
20: 1.0637
21: 0.293804
22: 1.44227
23: 0.581363
24: 1.27714
25: 0.506081
26: 0.450211
27: 2.05374
28: 0.401659
29: 2.01645
30: 0.780071

Abbildung 14.25: *Dendrogramm eines rekonstruierten Dendritensegments.*



VoxelSizeX = 0.636232	VoxelSizeY = 0.636232	VoxelSizeZ = 3	TotalDendriticBranchLength = 1164.71		
B1 - 1: 24.2355	Angle: 33.9659	B11 - 5: 140.839		S24: 24.2355	S34: 26.1512
B2 - 2: 32.8105	Angle: 44.3942	B12 - 4: 199.142		S25: 8.57498	S35: 48.3808
B3 - 3: 46.6066	Angle: 66.879	B13 - 5: 186.493		S26: 13.7961	S36: 46.9406
B4 - 4: 82.5447	Angle: 45.5662	B14 - 4: 87.5863		S27: 35.9381	S37: 94.0354
B5 - 5: 94.1345	Angle: 30.2115	B15 - 2: 90.5681	Angle: 51.4513	S28: 11.5899	S38: 103.689
B6 - 6: 141.738		B16 - 4: 188.174		S29: 47.6036	S39: 66.3326
B7 - 6: 182.663		B17 - 3: 96.0773	Angle: 17.1201	S30: 88.5287	S40: 61.6331
B8 - 5: 197.113		B18 - 4: 186.698		S31: 114.569	S41: 35.9725
B9 - 3: 66.3068	Angle: 51.0884	B19 - 4: 103.046	Angle: 33.2389	S32: 40.9797	S42: 5.50918
B10 - 4: 92.458	Angle: 22.4053	B20 - 5: 187.462		S33: 33.4964	S43: 90.6208

Abbildung 14.26: *Dendrogramm einer rekonstruierten Neuronenzelle.*

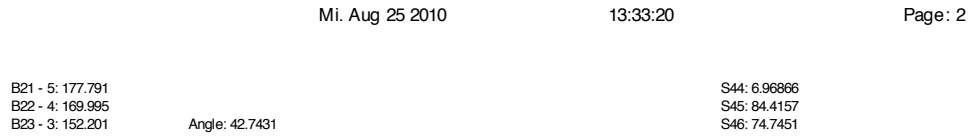


Abbildung 14.27: *Dendrogramm einer rekonstruierten Neuronenzelle (Fortsetzung).*

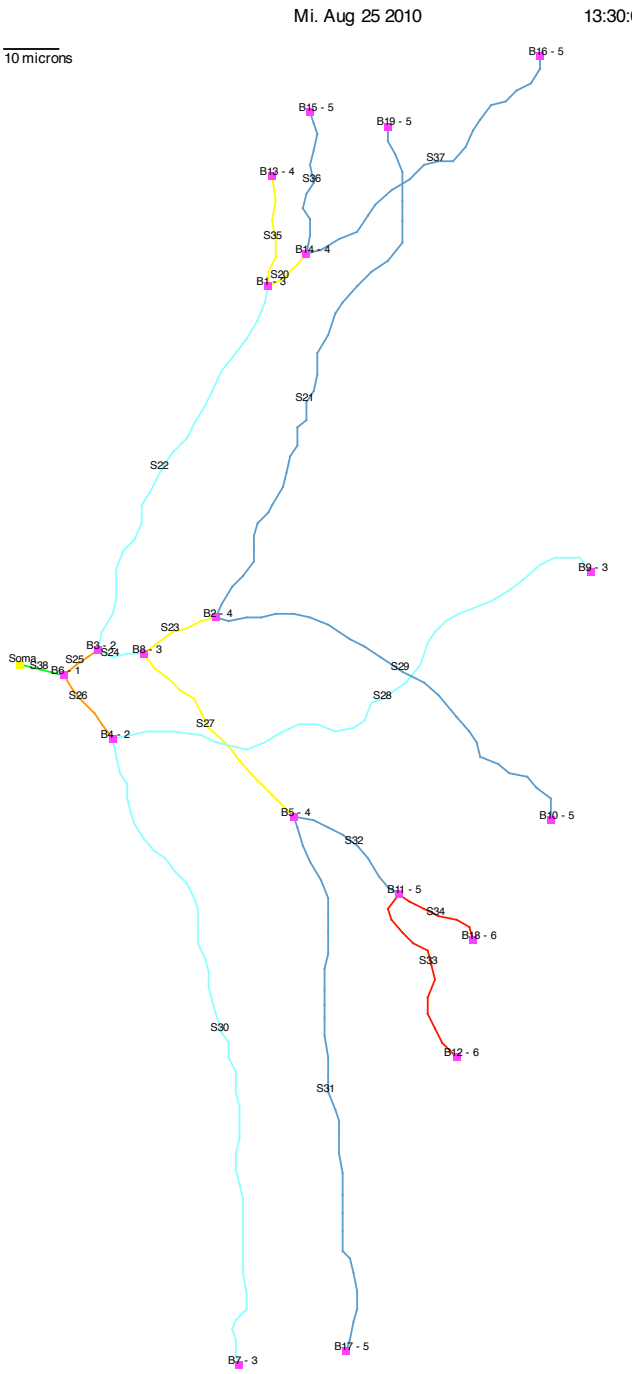


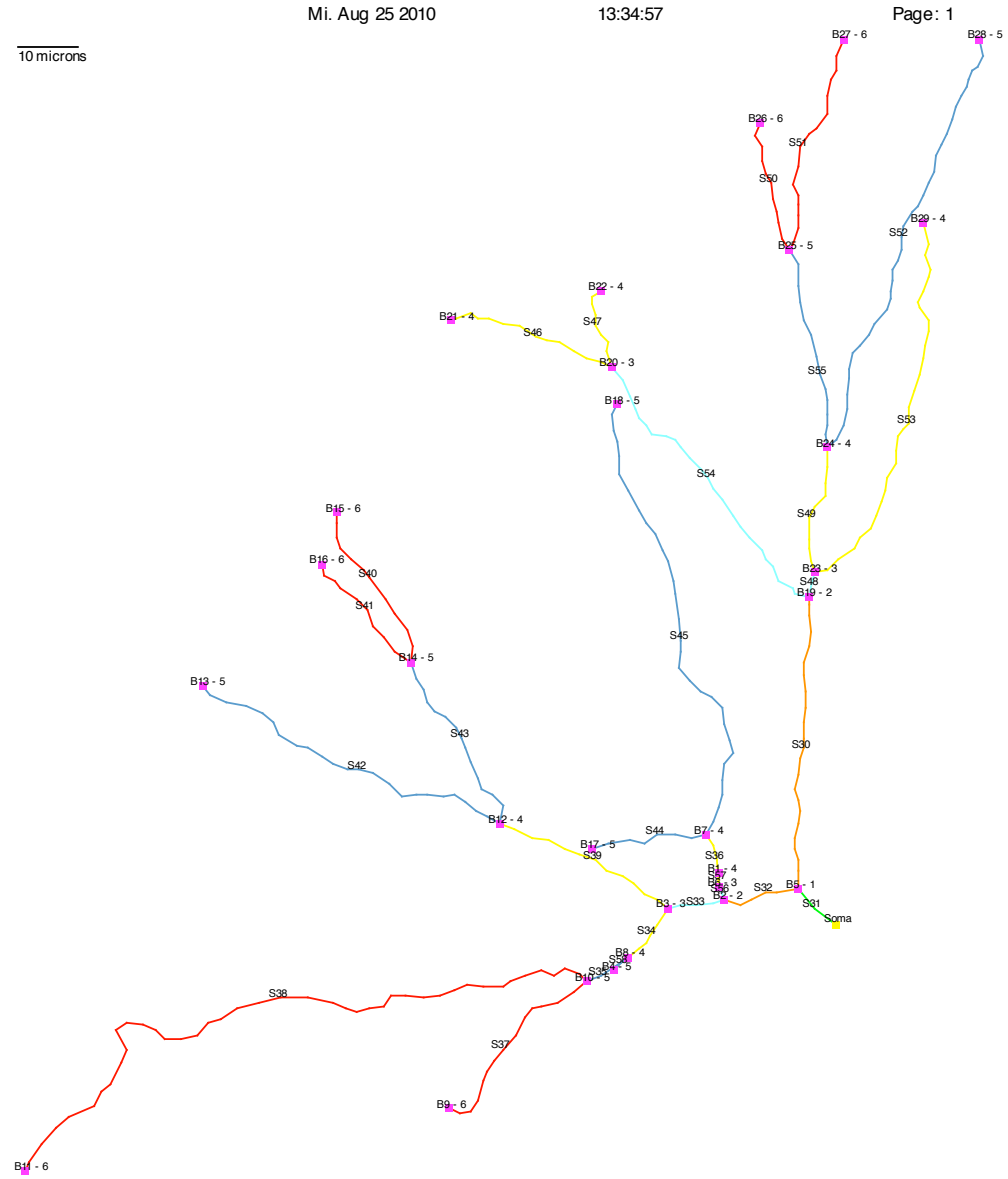
Abbildung 14.28: *Dendrogramm einer rekonstruierten Neuronenzelle.*

Mi. Aug 25 2010 13:30:05 Page: 2

VoxelSizeX = 0.636232 VoxelSizeY = 0.636232 VoxelSizeZ = 3 TotalDendriticBranchLength = 983.854

B1 - 3: 95.0365	Angle: 68.2912		S20: 14.0252
B2 - 4: 42.658	Angle: 101.398		S21: 110.515
B3 - 2: 15.6389	Angle: 75.2443		S22: 79.3976
B4 - 2: 24.6969	Angle: 97.7466		S23: 17.0075
B5 - 4: 68.0955	Angle: 50.2268		S24: 10.0116
B6 - 1: 7.86976	Angle: 87.336		S25: 7.7691
B7 - 3: 162.112			S26: 16.8272
B8 - 3: 25.6505	Angle: 75.4407		S27: 42.445
B9 - 3: 149.73			S28: 125.033
B10 - 5: 128.719			S29: 86.0607
B11 - 5: 97.6634	Angle: 52.7869		S30: 137.415
B12 - 6: 142.682			S31: 109.901
B13 - 4: 120.117			S32: 29.5678
B14 - 4: 109.062	Angle: 47.6779		S33: 45.0189
B15 - 5: 139.545			S34: 24.439
B16 - 5: 174.047			S35: 25.081
B17 - 5: 177.997			S36: 30.4829
B18 - 6: 122.102			S37: 64.9858
B19 - 5: 153.173			S38: 7.86976

Abbildung 14.29: *Dendrogramm einer rekonstruierten Neuronenzelle (Fortsetzung).*



VoxelSizeX = 0.449884 VoxelSizeY = 0.449884 VoxelSizeZ = 3 TotalDendriticBranchLength = 1198.3

B1 - 4: 39.2273		B7 - 4: 55.2353	Angle: 74.3374	S30: 57.2139	S36: 18.2574
B2 - 2: 30.5072	Angle: 68.7468	B8 - 4: 53.704	Angle: 12.2449	S31: 9.55509	S37: 46.7365
B3 - 3: 41.416	Angle: 77.2738	B9 - 6: 110.997		S32: 20.9521	S38: 143.382
B4 - 5: 58.0766		B10 - 5: 64.2604	Angle: 26.3264	S33: 10.9088	S39: 37.2291
B5 - 1: 9.55509	Angle: 88.4632	B11 - 6: 207.643		S34: 12.288	S40: 39.4972
B6 - 3: 36.9779	Angle: 60.5965	B12 - 4: 78.6451	Angle: 58.0467	S35: 10.5564	S41: 32.7023

Abbildung 14.30: *Dendrogramm einer rekonstruierten Neuronenzelle.*

Abbildung 14.31: *Dendrogramm einer rekonstruierten Neuronenzelle (Fortsetzung).*

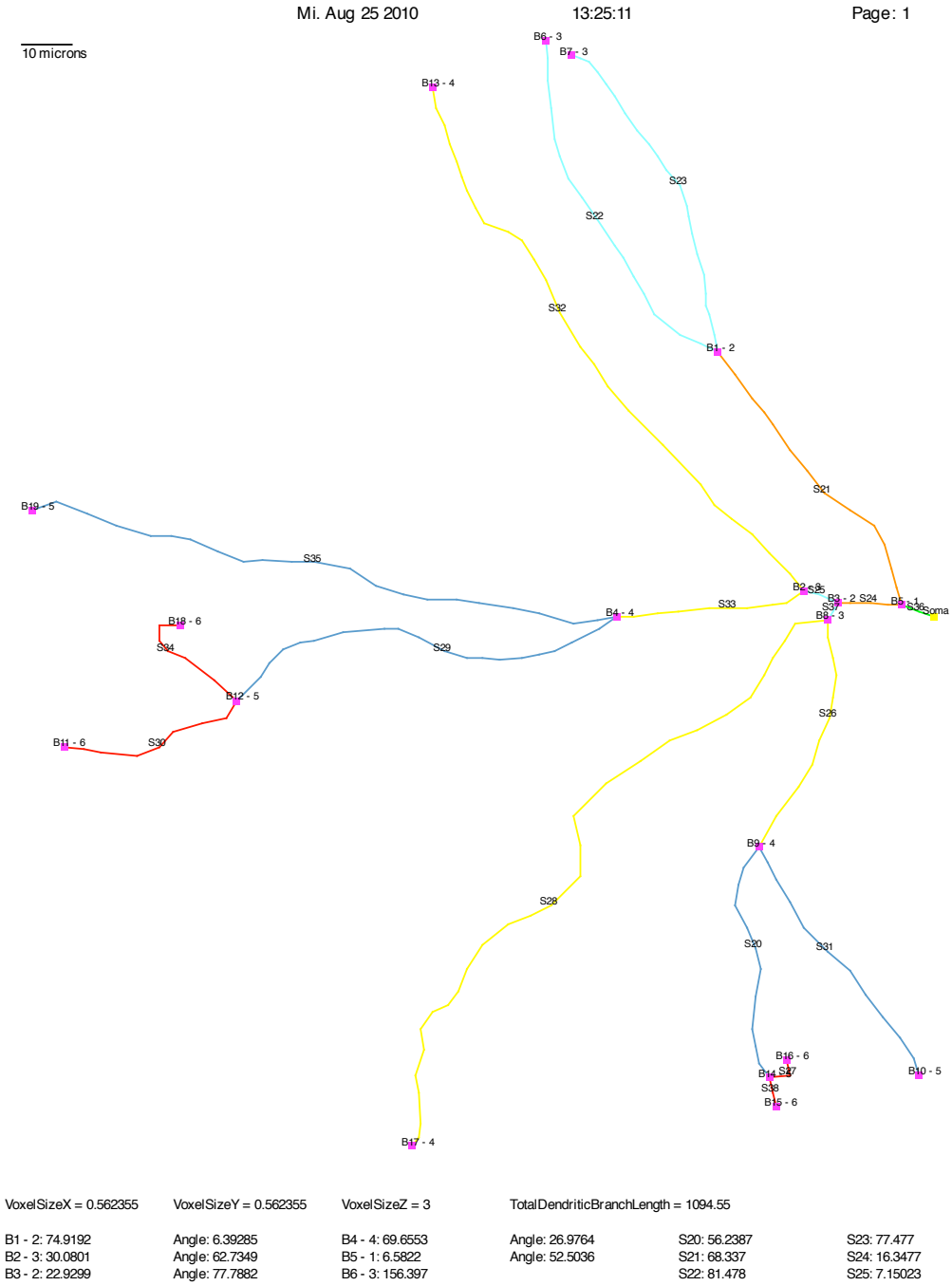
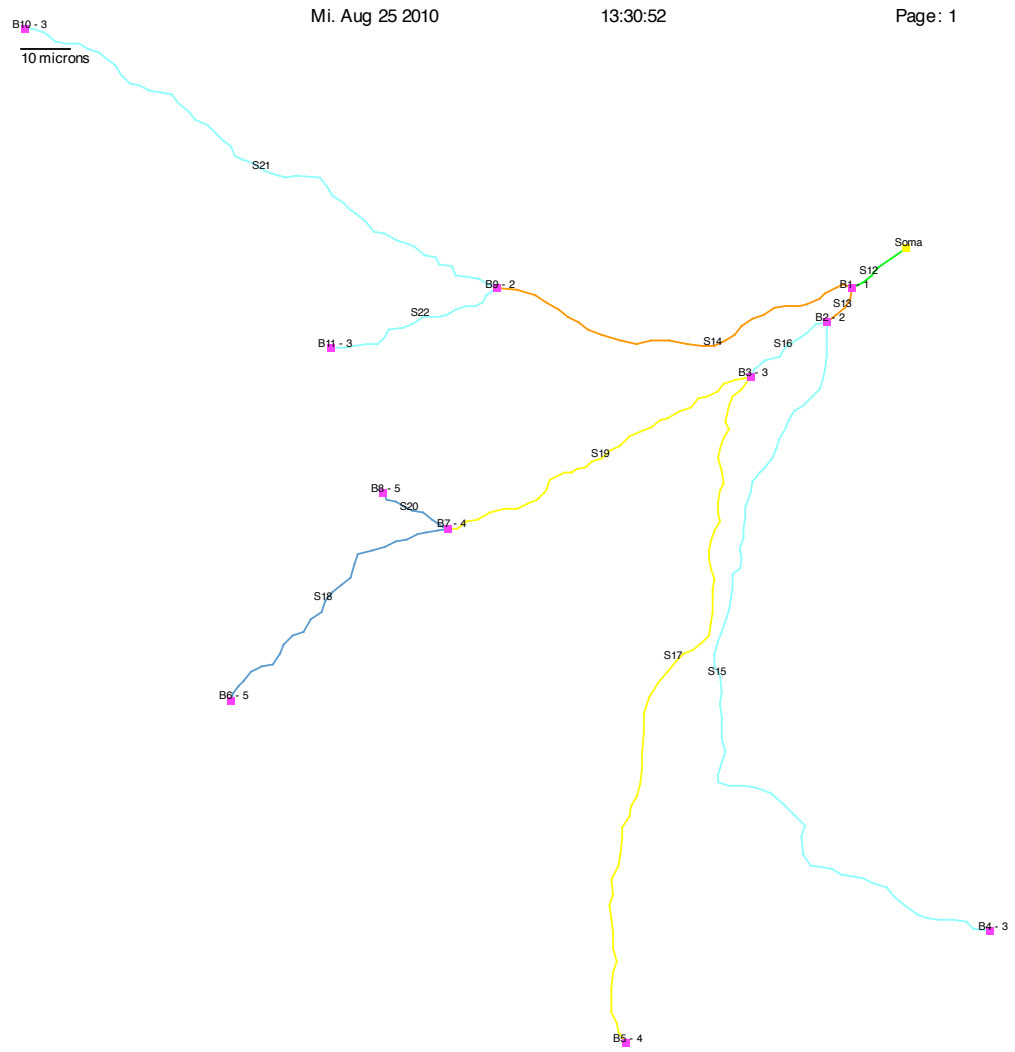


Abbildung 14.32: *Dendrogramm einer rekonstruierten Neuronenzelle.*

B7 - 3: 152.396
B8 - 3: 27.9742 Angle: 25.333
B9 - 4: 87.5538 Angle: 30.2413
B10 - 5: 164.352
B11 - 6: 203.45
B12 - 5: 162.432 Angle: 67.8168
B13 - 4: 162.503
B14 - 5: 143.793 Angle: 119.44
B15 - 6: 150.265
B16 - 6: 152.924
B17 - 4: 186.642
B18 - 6: 198.889
B19 - 5: 192.656

S26: 59.5796
S27: 9.13116
S28: 158.668
S29: 92.7772
S30: 41.0176
S31: 76.7979
S32: 132.423
S33: 39.5752
S34: 36.457
S35: 123
S36: 6.5822
S37: 5.04427
S38: 6.4722

Abbildung 14.33: *Dendrogramm einer rekonstruierten Neuronenzelle (Fortsetzung).*



VoxelSizeX = 0.44988 VoxelSizeY = 0.44988 VoxelSizeZ = 3 TotalDendriticBranchLength = 842.493

B1 - 1: 13.8945	Angle: 53.7821		S12: 13.8945
B2 - 2: 22.6992	Angle: 69.7821		S13: 8.80467
B3 - 3: 45.2072	Angle: 53.0033		S14: 92.6497
B4 - 3: 217.337			S15: 194.638
B5 - 4: 214.049			S16: 22.508
B6 - 5: 191.344			S17: 168.842
B7 - 4: 123.357	Angle: 70.6099		S18: 67.9864
B8 - 5: 140.558			S19: 78.15
B9 - 2: 106.544	Angle: 47.6479		S20: 17.2004
B10 - 3: 237.046			S21: 130.502
B11 - 3: 153.863			S22: 47.3184

Abbildung 14.34: *Dendrogramm einer rekonstruierten Neuronenzelle.*

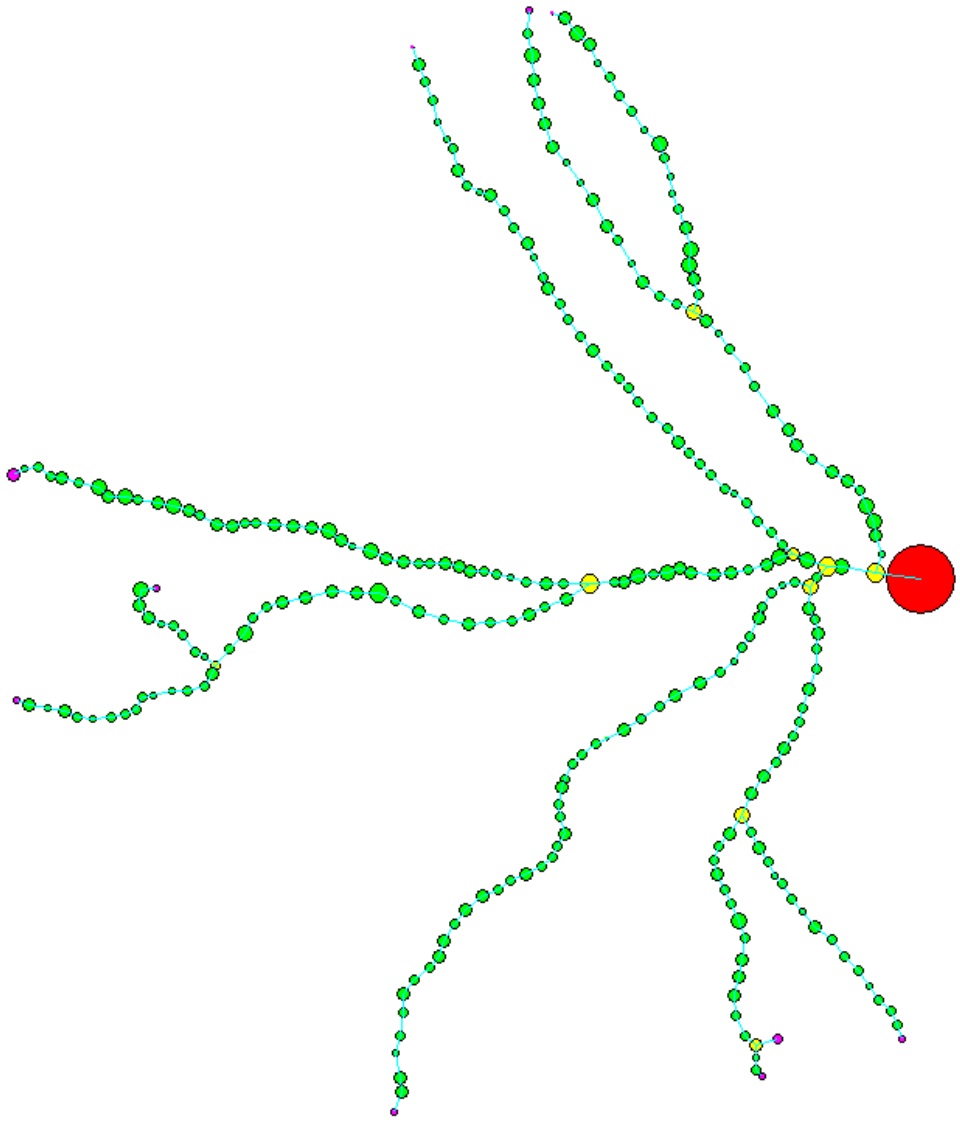


Abbildung 14.35: *Rekonstruktion der Neuronenzelle aus Abbildung 14.32 mit* NeuronStudio.

14.4.3 Oberflächenrekonstruktion von Neuronenzellen

In diesem Kapitel sind einige mit Hilfe von *NeuRA2* rekonstruierten Oberflächengeometrien von Neuronenzellen dargestellt (Abbildungen 14.36 und 14.37). Zur Rekonstruktion dieser Zellen wurde neben dem trägheitsbasierten Diffusionsfilter (Kapitel 5) der Multiskalen-Scharfzeichner (Kapitel 6) zur Rauschreduktion und Kontrasterhöhung angewandt. Solche Rekonstruktionen können zukünftig zur Untersuchung der Signalübertragung in Neuronen, wie sie in [118] auf Modellgeometrien durchgeführt wurde, verwendet werden. Abbildung 14.38 zeigt die Rekonstruktion einer Neuronenzelle einer lebenden Maus (vgl. Abbildungen 5.18 und 6.10).

Abbildung 14.36: *Oberflächenrekonstruktion einer hochaufgelöst mikroskopierten Neuronenzelle.*

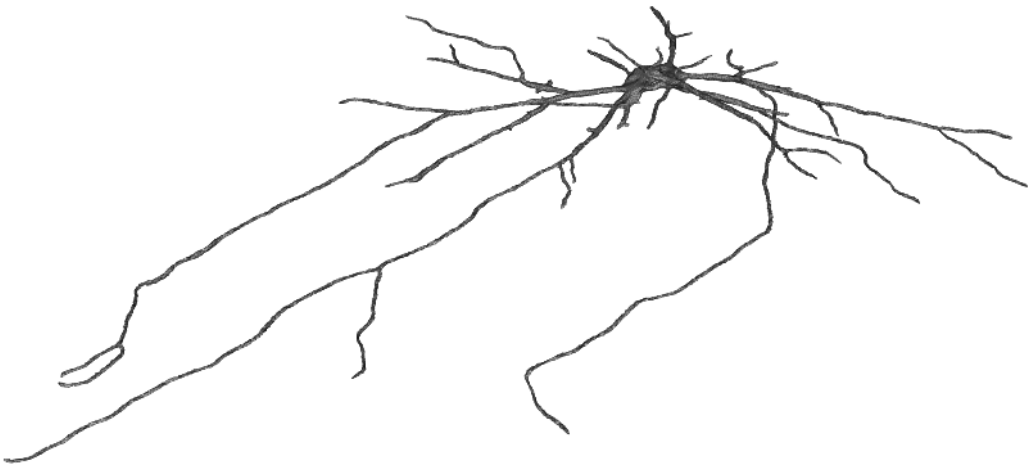
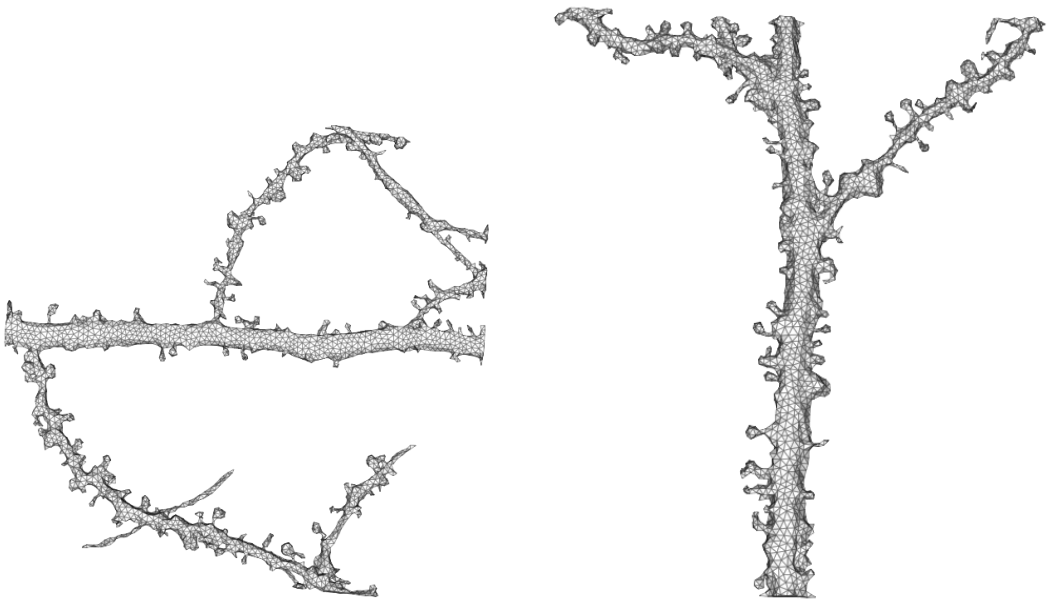


Abbildung 14.37: *Weitere Rekonstruktion einer hochaufgelöst mikroskopierten Neuronenzelle.*

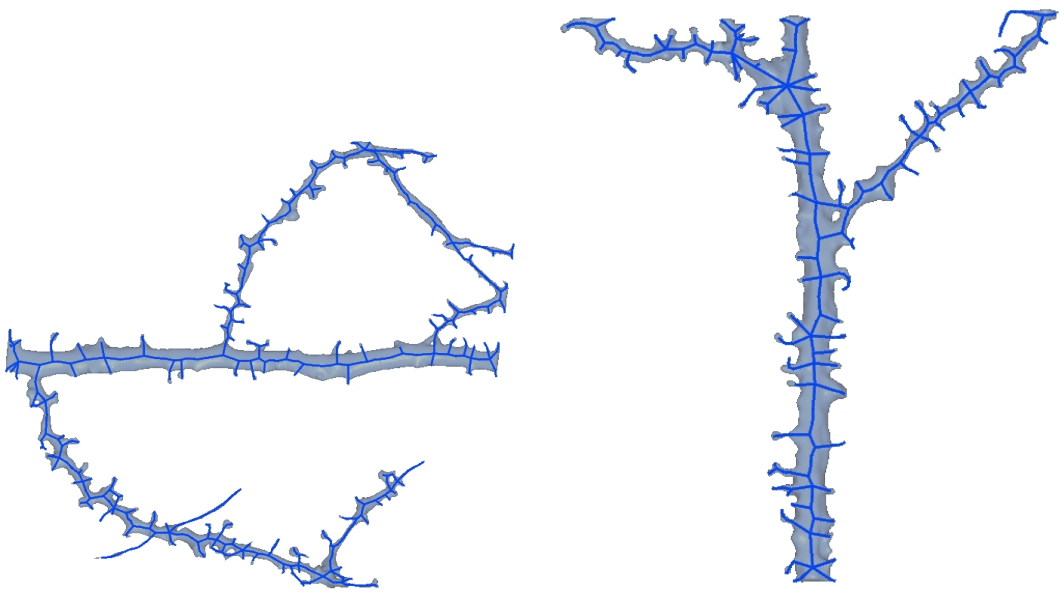
Abbildung 14.38: *Rekonstruktion einer Neuronenzelle aus lebendem Gewebe. Die geringe Auflösung der Aufnahme führt zu teilweise zackigen Rändern an den Dendriten.*

14.4.4 Oberflächenrekonstruktion von Dendritensegmenten



(a) **(b)**

Abbildung 14.39: *Oberflächenrekonstruktion von Dendritensegmenten und Spines.*

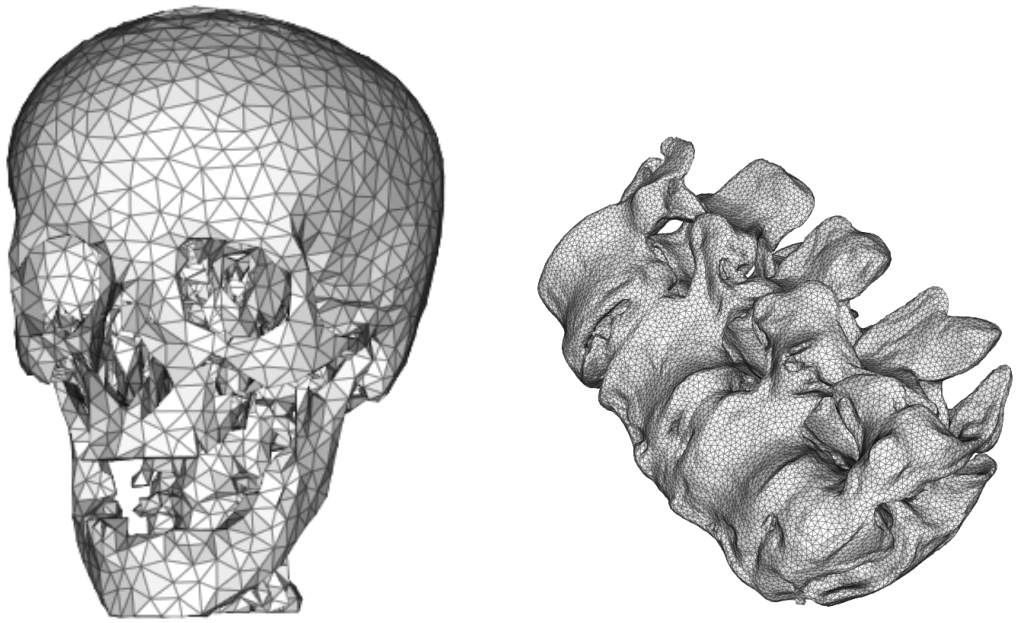


(a) **(b)**

Abbildung 14.40: *Mit Hilfe von* Gitter-Kontraktion [7] *extrahierte Merkmalskelette von Dendritensegmenten.*

Hochaufgelöst mikroskopierte Dendritensegmente mit anhängenden Spines können ebenfalls mit *NeuRA2* rekonstruiert werden (Abbildung 14.39). Da die Rohaufnahmen solcher Dendritensegmente häufig stark verrauscht sind, wird hierbei nicht der trägheitsbasierte Diffusionsfilter, sondern ein Medianfilter und der Multiskalen-Scharfzeichner zur Rauschreduktion angewandt. Nach einer anschließenden Segmentierung wird die Oberflächengeometrie rekonstruiert. Solche Rekonstruktionen können ebenfalls als Grundlage für numerische Simulationen oder zur Extraktion von Merkmalskeletten durch *Gitter-Kontraktion* [7] (Abbildung 14.40) verwendet werden. Es zeigt sich allerdings, dass aufgrund des hohen Rauschanteils in solchen Aufnahmen einige Spines bei der Gittergenerierung und bei der Gitter-Kontraktion verloren gehen können. Daher ist das in Kapitel 13 beschriebene Verfahren zur Skeletterzeugung vorzuziehen. Außerdem ist ein direkt auf den Bilddaten operierender Skelettierer deutlich schneller, da die Algorithmen zur Gittergenerierung und Gitteroptimierung nicht ausgeführt werden müssen.

14.5 Medizinische Rekonstruktionen

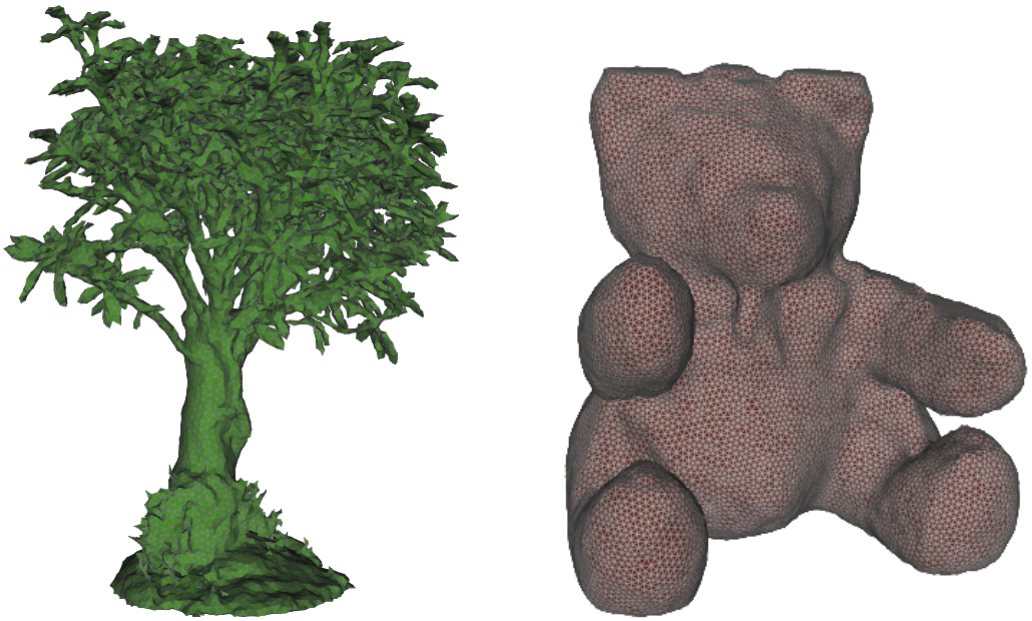


(a) **(b)**

Abbildung 14.41: *Medizinische Rekonstruktionen: (a) Schädel. (b) Halswirbelsäule.*

Medizinische CT-Aufnahmen verfügen üblicherweise über einen schlechteren Kontrast als industrielle CT-Aufnahmen oder die in dieser Arbeit behandelten Aufnahmen von Keramikgefäßen, da die Strahlendosis bei einem lebenden Patienten nicht sehr hoch gewählt werden darf. Dennoch lassen sich diese Aufnahmen hervorragend mit Hilfe von *NeuRA2* rekonstruieren (Abbildung 14.41 und Kapitel 14.2.4). Auch wenn im Rahmen dieser Arbeit keine konkreten Anwendungen mit diesen weiteren medizinischen Rekonstruktionen verknüpft sind, zeigt sie die hohe Flexibilität und Vielfältigkeit des Rekonstruktions-Algorithmus und dessen Verwendbarkeit in weiteren Anwendungsgebieten.

14.6 Weitere Rekonstruktionen



(a) **(b)**

Abbildung 14.42: *Weitere Rekonstruktionen: (a) Bonsai-Baum. (b) Teddybär.*

Unter [95] stehen verschiedene CT-Datensätze zum Download zur Verfügung, unter denen sich auch ein Bonsai-Baum und ein Teddybär befinden, die sich selbstverständlich mit Hilfe von *NeuRA2* rekonstruieren lassen (Abbildung 14.42). Diese beiden Rekonstruktionen haben zwar keinen weiteren Nutzen im Sinne interessanter Anwendungen, sie unterstreichen aber die hohe Flexibilität von *NeuRA2*.

Kapitel 15

Diskussion

Die in dieser Arbeit beschriebenen Weiterentwicklungen *NeuRA2* und *SpineLab* des Neuronen-Rekonstruktions-Algorithmus *NeuRA* ermöglichen die Rekonstruktion von Oberflächenmorphologien und Merkmalskeletten von Neuronen und subzellulären Strukturen, z.B. den dendritischen Dornfortsätzen. Da der hier vorgestellte Algorithmus im Gegensatz zu anderen Arbeiten, wie beispielsweise [100], auf modellbasierte Ansätze verzichtet, geht das Anwendungsspektrum von *NeuRA2* weit über die Rekonstruktion von Neuronenzellen hinaus und löst vielmehr die Aufgabe Oberflächengeometrien und Merkmalskelette aus beliebigen dreidimensionalen Bildern zu extrahieren, sofern diese Bilder nicht durch ein zu schlechtes Rausch-zu-Signal-Verhältnis beeinträchtigt sind.

Um trotz der teilweise extrem rechenintensiven Bildverarbeitungsoperatoren große Datenmengen verarbeiten zu können, verwendet *NeuRA2* die hochparallele Architektur moderner Grafikkarten, die diese Algorithmen bis zu einhundert Mal schneller ausführen können als aktuelle Standardprozessoren. Durch den parallelen Einsatz mehrerer Grafikkarten werden weitere immense Laufzeitgewinne erzielt. Es zeigt sich hierbei, dass die Architektur solcher Grafikkarten hervorragend für eine schnelle Ausführung von Bildverarbeitungsoperatoren geeignet ist, bei denen häufig die gleiche Operation auf alle Bildpunkte angewandt wird. Allerdings zeigt sich ebenfalls, dass sich nicht alle Algorithmen mit Hilfe von Grafikprozessoren immens beschleunigen lassen: Als Beispiel ist in dieser Arbeit der für die Gittergenerierung verwendete *Marching-Cubes-Algorithmus* aufgeführt, der sehr effizient auf einem Standardprozessor ausgeführt werden kann.

Neben der Weiterentwicklung und Parallelisierung des Rekonstruktions-Algorithmus ist in dieser Arbeit seine hohe Flexibilität durch vielfältige Anwendungsmöglichkeiten beschrieben: Neben der Gittererzeugung zur Ausführung numerischer Simulationen auf „echten“ Daten wurden Computertomographiebilder von antiken Keramikgefäßen rekonstruiert, sowie deren Füllvolumina und Rohdichten berechnet. Bei diesen Ergebnissen sollte zukünftig der Einfluss des Keramiktyps auf die Rohdichte noch systematischer untersucht werden, da hierbei für einige Keramiktypen zu wenige gescannte Objekte vorlagen. Darüber hinaus ermöglicht die Erzeugung von Merkmalskeletten aus konfokalen Mikroskopdaten die sys-

tematische Untersuchung des Einflusses pharmakologischer Präparate oder mechanischer Eingriffe auf die Morphologie lebender Neuronenzellen. Bei diesen Anwendungen wurde jeweils die Qualität der Rekonstruktionsverfahren untersucht, indem gezeigt wurde, wie genau Oberfläche, Volumen oder Krümmung eines Objekts durch eine entsprechende Rekonstruktion approximiert wird.

Aufgrund dieser Vielfalt von Anwendungen aus den verschiedensten Wissenschaftsbereichen lassen sich die in dieser Arbeit beschriebenen Rekonstruktionsmethoden zukünftig auf weitere Anwendungen übertragen.

Des Weiteren verbinden die Ergebnisse und Anwendungsfelder dieser Arbeit unterschiedliche Wissenschaftszweige und verdeutlichen die Wichtigkeit des Einsatzes von Methoden der Mathematik und des Wissenschaftlichen Rechnens in den Natur- und Geisteswissenschaften.

Anhang A

Implementierung – *NeuRA2*

Die im Rahmen dieser Arbeit entstandene GPU-basierte Neuimplementierung des *Neuron Reconstruction Algorithm* trägt den Namen *NeuRA2*. Die Software *NeuRA2* ermöglicht die Verarbeitung großer dreidimensionaler Bilder und beinhaltet unter anderem folgende Bildverarbeitungsoperatoren, die in dieser Arbeit beschrieben sind:

- Histogramm-Normalisierung, Invertierung, Ausschneiden von Teilbereichen, Einfügen von Schnittebenen, Skalierung, Drehen und Spiegeln der Koordinatenachsen, manuelle Bearbeitung mit einem Pinsel
- Gauß-Filter (Kapitel 4.3.1), Background-Noise-Subtraction (Kapitel 4.3.1), Medianfilter (Kapitel 4.3.2), Trägheitsbasierter anisotroper Diffusionsfilter (Kapitel 5), Multiskalen-Scharfzeichner (Kapitel 6)
- Globale Schwellwertsegmentierung (Kapitel 7.1), Hysterese-Segmentierung (Kapitel 7.1), Lokale *Otsu*-Segmentierung (Kapitel 7.3), Regionenwachstum-Segmentierung (Kapitel 7.4), Segmentieren von Teilbildern, Auswahl einzelner segmentierter Komponenten
- Marching-Cubes-Gittergenerator (Kapitel 8), Berechnung von Oberfläche und Volumen von Gittern (Kapitel 11), Konvertierung von Oberflächengittern ins *obj*-Dateiformat

Zudem kann *NeuRA2* folgende Programme zur Weiterverarbeitung der erzeugten Gittergeometrien starten und die zugehörigen Parameter einstellen:

- Gitterzerlegung (Kapitel 9)
- Gitteroptimierung (Kapitel 10)
- Volumengittergenerator *TetGen* [105]

Einige der Bildverarbeitungsoperatoren (Median-Filter, Trägheitsbasierter anisotroper Diffusionsfilter, Multiskalen-Scharfzeichner, *Otsu*-Segmentierung) sind in CUDA implementiert und benötigen daher eine oder mehrere CUDA-fähige Grafikkarten, um ausgeführt zu

werden. Obwohl *NeuRA2* zur Verarbeitung dreidimensionaler Bilder konzipiert ist, können alle Bildverarbeitungsoperatoren auf Bildern der Dimension zwei angewandt werden, die dann intern als dreidimensionales Bild mit Dimension $z = 1$ behandelt werden. Zusätzlich enthält die aktuelle Implementierung von *NeuRA2* noch folgende Operatoren, die speziell für zweidimensionale Bilder entwickelt wurden:

- *AFM*-Skelettierungsverfahren (Kapitel 13)
- Auf *Fourier-Transformation* basierender Tiefpassfilter [49]
- Zählen von nicht zusammenhängenden segmentierten Regionen

A.1 Programmablauf

Prinzipiell besteht der Ablauf von *NeuRA2* aus den Schritten Laden, Bearbeiten und Speichern von Daten. Hierbei sind die Bestandteile Datenzugriff, Datenverarbeitung und Ablaufsteuerung objektorientiert implementiert und strikt voneinander getrennt (Kapitel A.2). Die Steuerung der einzelnen Schritte kann mit Hilfe einer komfortablen grafischen Benutzeroberfläche (Kapitel A.3) oder einer Skriptsprache (Kapitel A.3.4) gesteuert werden.

A.2 Programmstruktur

Die innere Struktur von *NeuRA2* besteht im Wesentlichen aus drei Klassen:

1. `C3DImage`
2. `CImagePrOp`
3. `CControl`

Dies ermöglicht eine strikte Kapselung von Datenverwaltung (`C3DImage`), Bildverarbeitung (`CImagePrOp`) und Ablaufsteuerung (`CControl`). Der Austausch von Daten zwischen der Datenverwaltung und der Datenverarbeitung findet über die Klasse `CSubimage` statt (Abbildung A.1).

A.2.1 Die Klasse C3DImage

Die Klasse `C3DImage` ist für die Verwaltung großer Datenmengen in Form dreidimensionaler Bilder konzipiert. Für jeden Bildpunkt wird ein Byte Speicher alloziert. Dies entspricht einer Grauwertauflösung von 256 Grauwertstufen. Sei u ein Bild der Dimension `size_x` $\times$ `size_y` $\times$ `size_z`, so wird u als `size_z`-dimensionaler Stapel von zweidimensionalen Bildern der Größe `size_x` $\times$ `size_y` gespeichert. Dieses Vorgehen ermöglicht ein problemloses

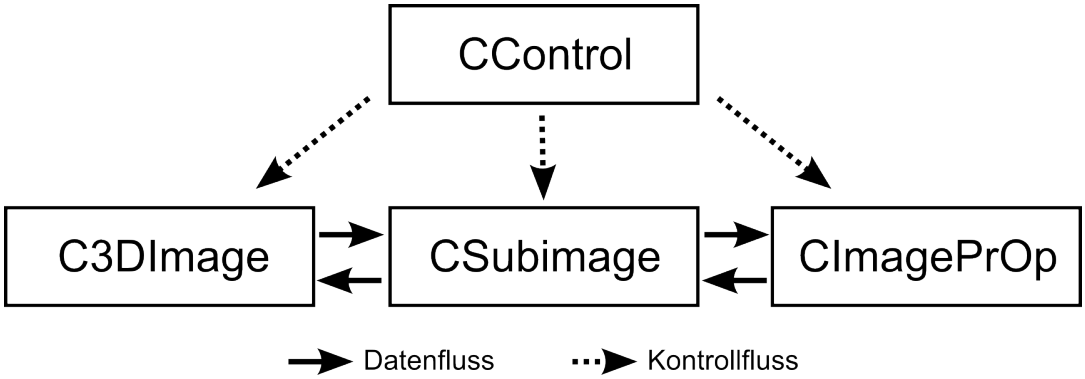


Abbildung A.1: *Programmstruktur von* NeuRA2*: Datenverwaltung, Datenverarbeitung und Ablaufsteuerung sind strikt gekapselt. Der Austausch von Daten findet über die Klasse* `CSubimage` *statt.*

Verwalten großer Datenmengen, da nur relativ kleine zusammenhängende Speicherbereiche alloziert werden müssen. Es besteht die Möglichkeit Bilder in drei verschiedenen Datenformaten zu laden und zu speichern:

1. `raw`-Format: Im `raw`-Format werden die Grauwerte der einzelnen Voxel mit einem Speicheraufwand von einem Byte pro Voxel hintereinander gespeichert. Dieses Format enthält keinerlei Informationen über die Dimension der gespeicherten Daten. Die Dimensionen müssen daher beim Lesen eines im `raw`-Format gespeicherten Bildes als zusätzliche Parameter bzw. durch ein entsprechendes Dialogfenster in der grafischen Benutzeroberfläche eingegeben werden.

2. `dat`-Format: Im `dat`-Format werden die Grauwerte der einzelnen Voxel ebenfalls byteweise hintereinander gespeichert. Das `dat`-Format enthält im Gegensatz zum `raw`-Format zusätzlich einen 24 Byte großen Header, der die Dimensionen des Bildes sowie die Größe der einzelnen Voxel enthält. Der Header besteht aus sechs ganzzahligen Werten, die jeweils mit vier Byte kodiert sind. Diese Werte sind im Einzelnen:

   - `size_x`: Dimension des Bildes in x-Richtung
   - `size_y`: Dimension des Bildes in y-Richtung
   - `size_z`: Dimension des Bildes in z-Richtung
   - `d_x`: Größe der Voxel in x-Richtung
   - `d_y`: Größe der Voxel in y-Richtung
   - `d_z`: Größe der Voxel in z-Richtung

   Beispielcode zum Erstellen von Dateien im `dat`-Format findet sich in [27]. In diesem Format gespeicherte Bilder können durch die am Interdisziplinären Zentrum für Wissenschaftliches Rechnen (IWR) der Universität Heidelberg entwickelten Programme

VRend [27] und *VVis* [30] mit Hilfe verschiedener Techniken zur Volumenvisualisierung [32, 33] dargestellt werden.

3. `tiff`-Format: Das `tiff`-Format ist ein sehr komplexes Dateiformat, das viele Möglichkeiten zum Speichern und Komprimieren von Bildern zur Verfügung stellt [1]. Durch die Verwendung sogenannter *Ordner* (englisch: Directories) ermöglicht das `tiff`-Format dreidimensionale Bilder in Form von Bildstapeln zu speichern. Hierbei wird jedes Einzelbild des Bildstapels in einem separaten Ordner abgelegt. *NeuRA2* und *SpineLab* verwenden die Bibliothek *Libtiff* [62] für das Lesen und Schreiben von Bildern im `tiff`-Format.

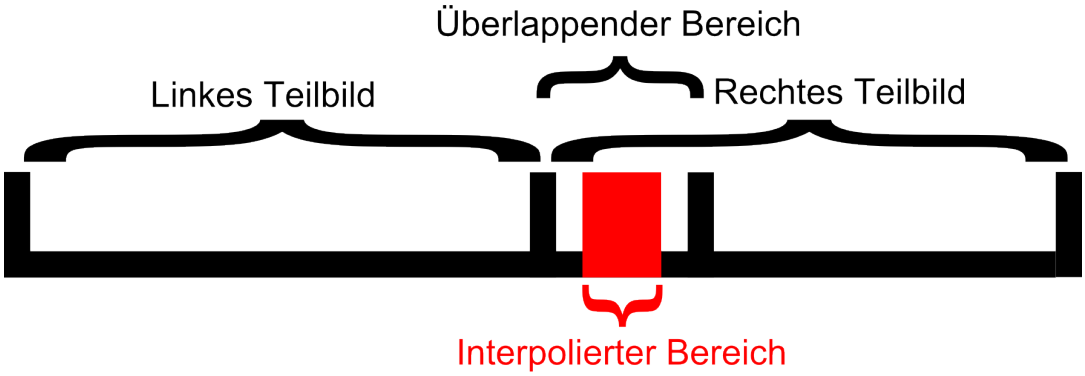
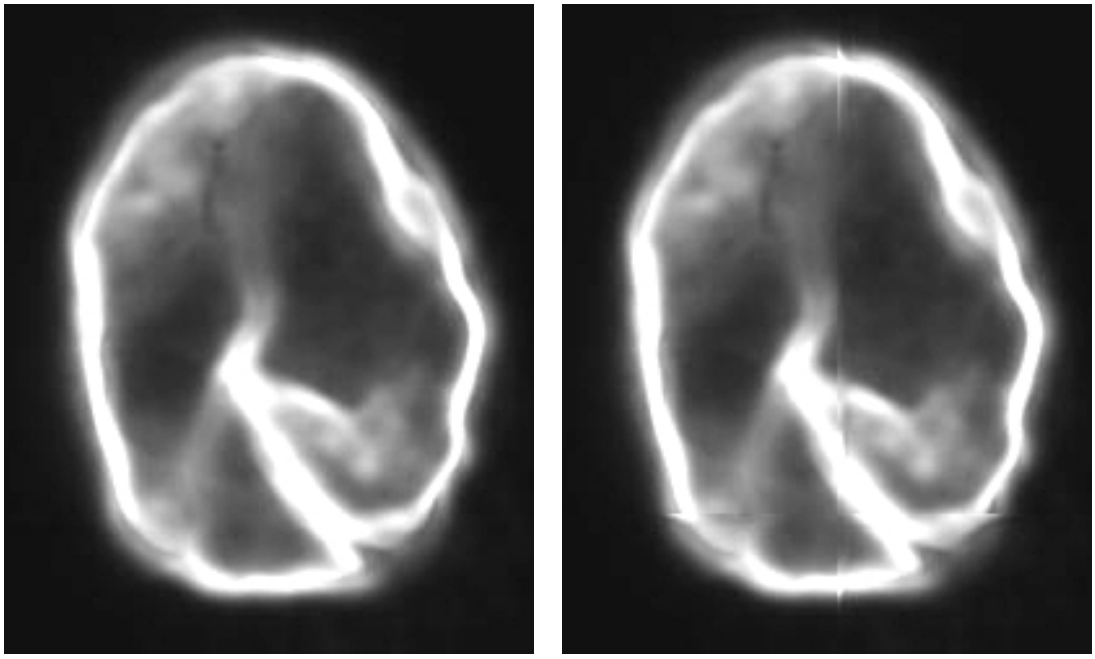


Abbildung A.2: *In der Mitte der überlappenden Bereiche werden die Grauwerte der einzelnen Teilbilder linear interpoliert.*

Die Klasse `C3DImage` stellt ferner die Möglichkeit zur Verfügung ein geladenes Bild in überlappende Teilbilder zu zerlegen, die dann separat - in beliebiger Reihenfolge oder parallel - bearbeitet und anschließend zurückgeschrieben werden können. Beim Zurückschreiben der Teilbilder werden die Grauwerte der überlappenden Regionen linear interpoliert, sodass ein stetiges Ergebnisbild entsteht (Abbildung A.2). Bei der Zerlegung von dreidimensionalen Bildern entstehen Bereiche, in denen durch die Interpolation bis zu acht Teilbilder Beiträge zu dem zusammengesetzten Bild liefern. In diesen Bereichen wird entsprechend eine bilineare, trilineare usw. Interpolation eingesetzt. In *NeuRA2* ist die Größe der Überlappung mit sechzehn Voxel und die Größe des interpolierten Bereichs mit vier Voxel festgelegt. Die einzelnen Teilbilder sind Objekte der Klasse `CSubimage`, die jeweils die Daten der Teilbilder sowie deren Dimension und Informationen über die Überlappung und ihre Position im Gesamtbild beinhalten. Zerlegt und bearbeitet man die Bilder ohne Überlappung und entsprechender Interpolation sind deutliche Unstetigkeitskanten in den Ergebnisbildern zu erkennen (Abbildung A.3).

A.2.2 Die Klasse CImagePrOp

Die Klasse `CImagePrOp` (Abkürzung für Image Processing Operator) ist eine abstrakte Basisklasse von der beliebige Bildverarbeitungsoperatoren abgeleitet werden können (Ab-



(a) **(b)**

Abbildung A.3: *Schnittebenendarstellung eines mit trägheitsbasierter Diffusion gefilterten Neuronenzellkerns (a) mit und (b) ohne Überlappung und entsprechender Interpolation. In der Mitte von Abbildung (b) ist deutlich eine Unstetigkeitskante zu erkennen.*

bildung A.4). Die Basisklasse stellt Funktionen, mit denen die Eingabeparameter der abgeleiteten Bildverarbeitungsoperatoren eingestellt werden können zur Verfügung, ebenso wie die rein virtuelle Funktion

```
virtual int process(CSubimage &subimage,
                    C3DImage* &image,
                    const int nArgInts,
                    int *argInts,
                    const int nArgFloats,
                    float *argFloats,
                    const int nArgStrings,
                    char **argStrings) = 0;
```

mit deren Hilfe der Aufruf des Operators auf einem Teilbild gestartet wird. Durch den Übergabeparameter `argInts` werden `nArgInts` ganzzahlige Parameter an den Bildverarbeitungsoperator übergeben. Entsprechend können Fließkommazahlen und Zeichenketten als Eingabeparameter an die Operatoren weitergereicht werden. Manche der Operatoren benötigen keine Zerlegung des Bildes in Teilbilder. Diese können über die Referenz `image` direkt auf den gespeicherten Daten operieren. Dies vermeidet einerseits das unnötige Kopieren großer Datenmengen und ermöglicht andererseits speziellen Operatoren die Größe des Bildes zu verändern. Mit Hilfe der ebenfalls virtuellen Funktion

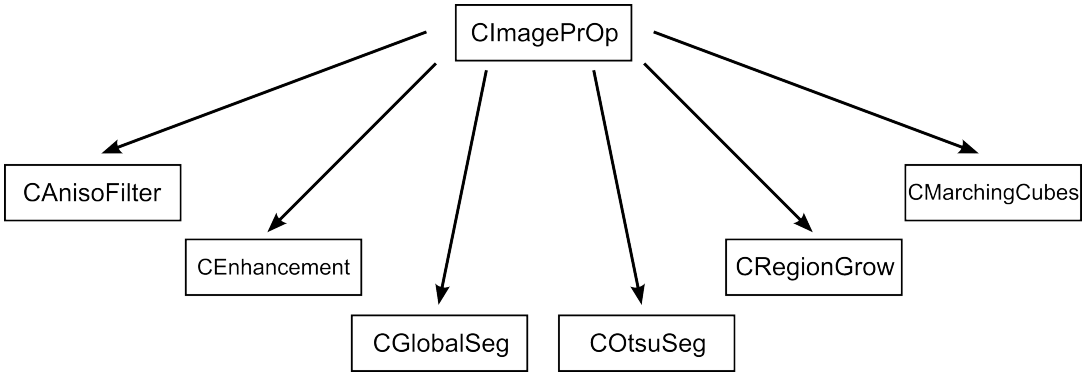


Abbildung A.4: *Von der abstrakten Basisklasse* `CImagePrOp` *werden die einzelnen Bildverarbeitungsoperatoren abgeleitet.*

```
virtual int getSubimageSize(int &size_x,
                            int &size_y,
                            int &size_z,
                            int &overlap_x,
                            int &overlap_y,
                            int &overlap_z,
                            int &interpolate_x,
                            int &interpolate_y,
                            int &interpolate_z,
                            const size_t availableGpuMemory) {
  size_x = size_x;
  size_y = size_y;
  size_z = size_z;
  overlap_x = 0;
  overlap_y = 0;
  overlap_z = 0;
  interpolate_x = 0;
  interpolate_y = 0;
  interpolate_z = 0;
  return 0;
}
```

hat jeder Bildverarbeitungsoperator die Möglichkeit zu bestimmen wie groß die einzelnen Teilbilder sein sollen. Hierbei kann die Überlappungsgröße `overlap_x, overlap_y, overlap_z` und ebenfalls ob beim Zusammensetzen der Teilbilder interpoliert werden soll (`interpolate_x, interpolate_y, interpolate_z`) eingestellt werden. Die Teilbildgröße kann dabei in Abhängigkeit des vorhandenen Grafikspeichers `availableGpuMemory` berechnet werden. Ist die Funktion `getSubimageSize` in der abgeleiteten Klasse nicht überladen, wird das komplette Bild an den Bildverarbeitungsoperator übergeben. Durch die Funktio-

nen

```
virtual int getSubimageSizePreview( ... ) { ... }
```

und

```
virtual int processPreview( ... ) { ... }
```

kann auf gleiche Weise eine Vorschauoption des Bildverarbeitungsoperators implementiert werden. Nach erfolgreichem Ausführen einer Operation wird diese in der Protokolldatei `log/logfile.log` gespeichert. Die Ausgabe der einzelnen Operatoren wird durch die virtuelle Funktion

```
virtual int log( ...) { ... }
```

gesteuert. Hierbei wird der verwendete Operator zusammen mit der Parametereinstellung in der Protokolldatei vermerkt.

A.2.3 Die Klasse CControl

Die Klasse `CControl` regelt den Programmablauf von *NeuRA2*. Beim Programmstart wird zunächst die verfügbare Grafikhardware untersucht und bestimmt welche Devices zum Berechnen der Bildverarbeitungsoperatoren verwendet werden können. Diese Analyse wird in der Datei `log/gpu.log` gespeichert. Falls keine CUDA-fähigen Devices vorhanden sind, beendet sich *NeuRA2* mit einer entsprechenden Meldung. Ferner steuert die Klasse `CControl` das Laden und Speichern von Bilddaten und die Ausführung der verschiedenen Bildverarbeitungsoperatoren. Hierbei werden, je nach verwendetem Operator und vorhandener Hardware, die zu bearbeitenden Daten in Teilbilder passender Größe zerlegt, bearbeitet und wieder zusammengesetzt. Sind mehrere CUDA-fähige Devices vorhanden, werden die einzelnen Teilbilder automatisch auf diese verteilt und parallel verarbeitet (Kapitel 3.8.3).

A.3 Benutzeroberfläche von *NeuRA2*

Die Benutzeroberfläche von *NeuRA2* bietet die Möglichkeit Daten zu laden und zu speichern, die eingebauten Bildverarbeitungsoperatoren aufzurufen und deren Eingabeparameter einzustellen. Die Benutzeroberfläche (Abbildung A.5) gliedert sich in Toolbox (oben), Voransicht (links), Auswahl der Bildverarbeitungsoperatoren (rechts) und Feedback-Fenster (unten). Die grafische Oberfläche wurde unter Verwendung von *Qt4* [17] implementiert.

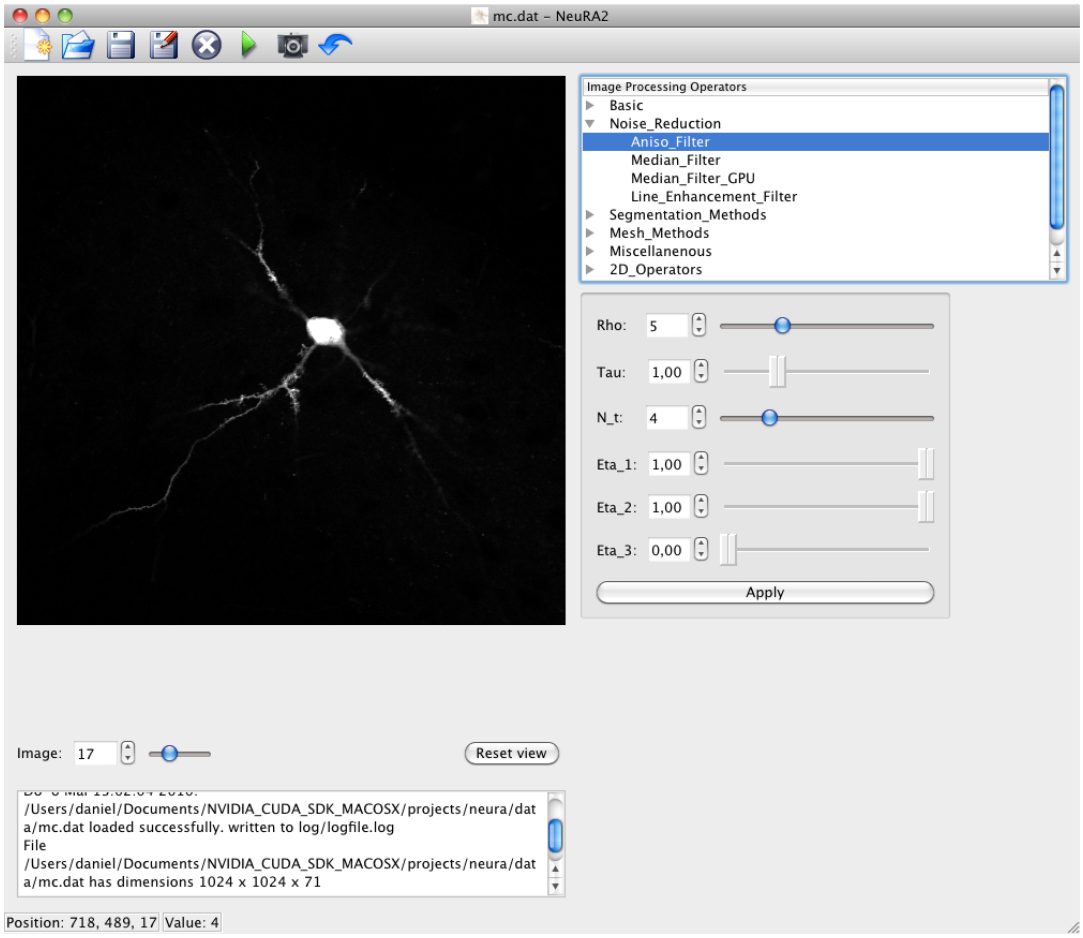


Abbildung A.5: *Die* NeuRA2 *Benutzeroberfläche.*

A.3.1 *NeuRA2* Toolbox

Die Toolbox enthält die einzelnen Funktionen zum Laden und Speichern der Bilder, sowie die Möglichkeit neue *NeuRA2* Fenster zu öffnen, Skripte auszuführen und die aktuelle Ansicht zu speichern. Die Möglichkeit *NeuRA2* zu skripten ist in Kapitel A.3.4 erklärt.

A.3.2 *NeuRA2* Ansichtsfenster

Im Ansichts-Fenster werden die geladenen Volumenbilder in Schnittebenendarstellung (vgl. Kapitel 1.3.3) angezeigt. Der Slider unterhalb des Ansichtsfensters ermöglicht die Auswahl der Schnittebenen in z-Richtung. Das Bild kann bei gedrückter mittlerer Maustaste bewegt werden. Durch Drücken der rechten Maustaste ist eine Vergrößerung bzw. Verkleinerung des Bildes möglich.

A.3.3 Steuerung der Bildverarbeitungsoperatoren

Im rechten Teil der Benutzeroberfläche können die in unterschiedlichen Abschnitten gegliederten Bildverarbeitungsoperatoren ausgewählt werden. Unterhalb der Auswahl der Operatoren können die Parameter des gewählten Operators eingestellt und dieser auf das Bild angewandt werden. Einige der Bildverarbeitungsoperatoren stellen eine Voransicht zur Verfügung. Ist die Voransicht aktiviert, so wird der ausgewählte Operator auf das aktuell eingestellte Schnittebenenbild angewandt, wodurch die Eingabeparameter leicht angepasst werden können. Die Verwendung der Voransicht bietet sich vor allem bei den unterschiedlichen Segmentierungsverfahren an.

A.3.4 *NeuRA2* Skriptsprache

NeuRA2 kann *NeuRA2*-Skriptdateien (*.nsc) direkt ausführen, ohne dass eine weitere Interaktion mit dem Benutzer notwendig ist. Mit Hilfe der Befehle

```
read_raw filename size_x size_y size_z
read_dat filename
read_tiff filename
write_raw filename
write_dat filename
write_tiff filename
```

können hierbei Bilder, die in den unterstützten Bildformaten vorliegen, geladen und gespeichert werden. Die Bildverarbeitungsoperatoren werden mit dem Befehl

```
improp name
```

oder

```
improp_index index
```

gestartet. Die Indizes der einzelnen Operatoren können dabei der Quelldatei `header/constants.h` entnommen werden. Die Eingabeparameter werden dann über die Schlüsselwörter `int`, `float` und `string` wie folgt eingestellt:

```
int nArgInts argInt0 argInt1 ...
```

Der Befehl `apply` startet den Operator. Bei der Wahl eines neuen Operators durch die Befehle `improp` oder `improp_index` werden die zuvor eingestellten Parameter gelöscht. Folgendes Beispiel verdeutlicht die Syntax eines *NeuRA2*-Skripts:

```
improp marching_cubes
  int      1  1
  float    6  0.0  0.0  0.0  1.0  1.0  1.0
  string   1  data/mesh.txt
  apply
```

Die Reihenfolge der Parameter kann aus der Reihenfolge ihrer Auflistung in der Benutzeroberfläche ermittelt werden. Nach Abarbeitung eines Skript beendet sich *NeuRA2* von selbst.

Anhang B

Implementierung - *SpineLab*

Die ebenfalls im Rahmen dieser Doktorarbeit entwickelte Software *SpineLab* vereinigt die behandelten Algorithmen zur Rekonstruktion und Analyse von Merkmalskeletten aus neurobiologischen Mikroskopaufnahmen. Darüber hinaus stellt *SpineLab* eine manuelle Nachbearbeitung der rekonstruierten Merkmalskelette zur Verfügung, die erforderlich ist, da solche *in vivo* Aufnahmen typischerweise stark verrauscht sind. Die objektorientierten Konzepte und die strikte Trennung zwischen Datenverwaltung, Datenverarbeitung und Programmsteuerung wurden von *NeuRA2* (Anhang A) übernommen. Im Gegensatz zu *NeuRA2* verzichtet *SpineLab* aus zwei Gründen auf eine GPU-basierte Beschleunigung: Einerseits ist *SpineLab* zur Verarbeitung von kleinen Datensätzen ausgelegt, andererseits funktioniert *SpineLab* dadurch unabhängig von der verwendeten Hardware. *SpineLab* ist ebenfalls in *Qt4* [17] implementiert, weshalb sich die Software leicht auf alle gängigen Betriebssysteme portieren lässt. Versionen für *MacOSX*, *Linux* und *Windows* sind verfügbar. Neben der Skelettierung und Analyse ganzer Neuronenzellen (Abschnitt B.2) oder einzelner Dendritensegmente (Abschnitt B.1) mit dendritischen Dornfortsätzen (Spines) kann die Oberfläche einzelner Spines rekonstruiert und dadurch das Volumen dieser Spines berechnet werden (Abschnitt B.1.4). In *SpineLab* sind folgende Bildverarbeitungsoperatoren direkt integriert:

- Background-Noise-Subtraction (Kapitel 4.3.1)
- Medianfilter (Kapitel 4.3.2)
- Globale Schwellwertsegmentierung (Kapitel 7.1)

Zur Rekonstruktion der Merkmalskelette findet das in Kapitel 13 beschriebene *Augmented-Fast-Marching* Verfahren Anwendung. Zur Oberflächenrekonstruktion der Spines ist zudem der Marching-Cubes-Gittergenerator (Kapitel 8) in *SpineLab* eingebaut. Eine rauschvermindernde Vorverarbeitung oder eine Segmentierung der Daten kann vorher mit *NeuRA2* durchgeführt werden.

B.1 Rekonstruktion von Dendritensegmenten

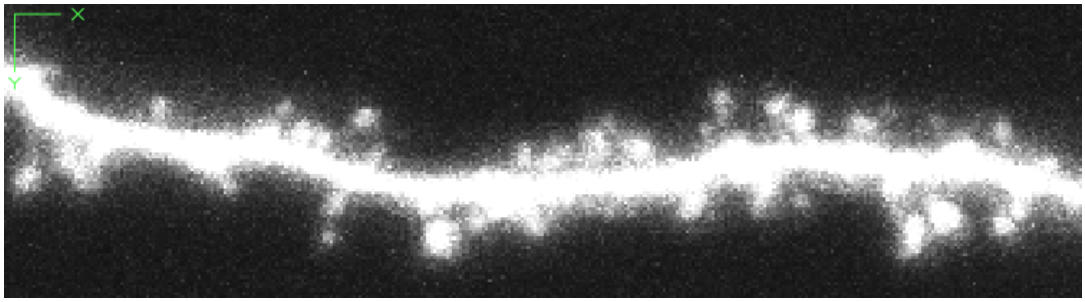
Die semiautomatische Rekonstruktion von Dendritensegmenten mit Spines ist im Folgenden erklärt. Hierbei dient eine stark verrauschte konfokale Aufnahme eines Dendritensegments als Grundlage (Abbildung B.1). Neben dem starken Rauschsignal sind einige der Spineköpfe nicht sichtbar mit dem Dendriten verbunden. Der in *SpineLab* realisierte Rekonstruktions- und Analyseprozess gliedert sich dabei in die folgenden Schritte, die in den nächsten Abschnitten detailliert beschrieben sind:

1. Automatische Generierung des Skeletts
2. Manuelle Nachbearbeitung des Skeletts
3. Automatische Längen- und Populationsanalyse der Spines, entsprechend Definition 13.3
4. Volumenanalyse einzelner rekonstruierter Spines

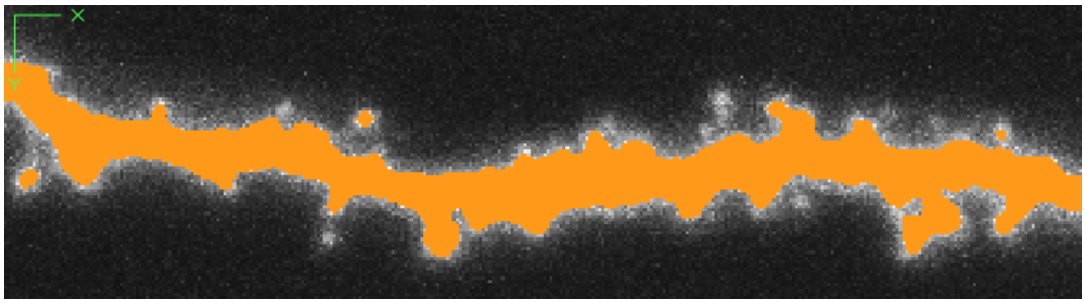
B.1.1 Automatische Generierung des Skeletts

Nach einer optionalen Vorverarbeitung des Bildes mit *NeuRA2* wird das Skelett automatisch wie folgt erzeugt:

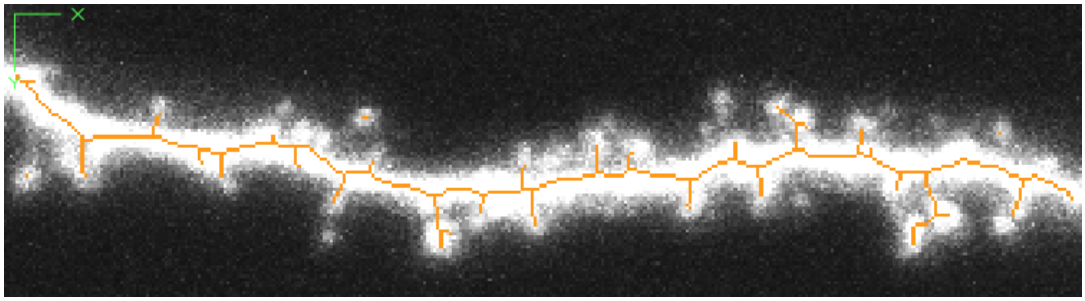
1. Das Bild wird entlang der z-Achse auf eine Bildebene projiziert. Auf das projizierte Bild wird ein Medianfilter angewandt und das Ergebnis anschließend mittels globaler Schwellwertsegmentierung segmentiert (Abbildung B.1a). Bei Bildern mit starken Kontrastunterschieden kann der Benutzer zudem den Background-Noise-Subtraction-Operator vor den Medianfilter schalten. Der Segmentierungsparameter kann durch den Benutzer in Abhängigkeit der Daten in Echtzeit eingestellt werden.
2. Mit Hilfe des *AFM*-Verfahrens wird die Mittelachse aller segmentierten Bereiche erzeugt. Kleine Segmente, die häufig Spineköpfen entsprechen, werden mit mindestens einem Mittelachsenpunkt versehen, auch wenn das *AFM*-Verfahren keine Mittelachse für diese Bereiche erzeugt (Abbildung B.1c). Die Sensitivität der Mittelachsenbestimmung kann ebenfalls durch den Benutzer in Echtzeit eingestellt werden.
3. Das Skelett wird wie in Kapitel 13.2 beschrieben erzeugt (Abbildung B.1d) und die z-Koordinaten der Skelettknoten als Schwerpunkt in z-Richtung berechnet (Kapitel 13.2.1).
4. Der Dendrit wird als längster zusammenhängender Pfad des Skelettgraphen identifiziert (Abbildung B.2a). Bei verzweigten Dendriten muss der Benutzer die Endpunkte und Verzweigungspunkte der Dendriten manuell identifizieren (vgl. Anhang B.1.2).



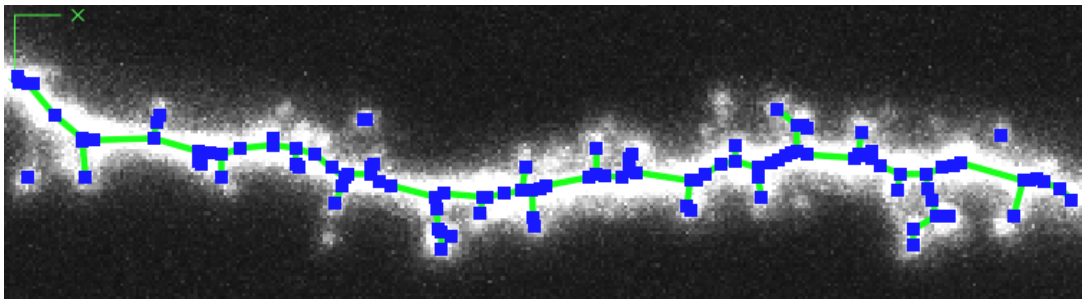
(a)



(b)

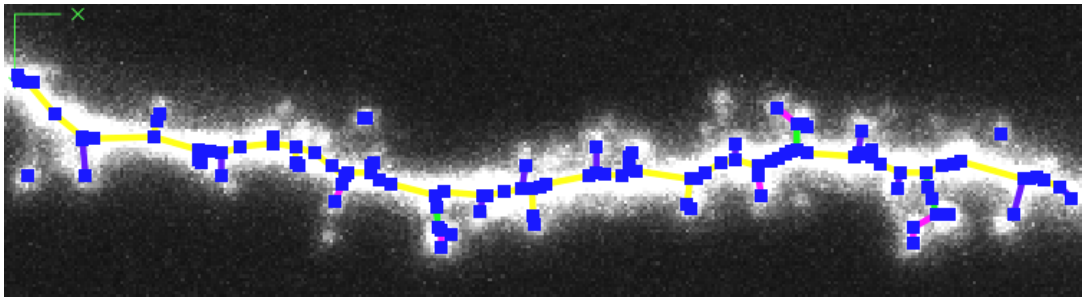


(c)

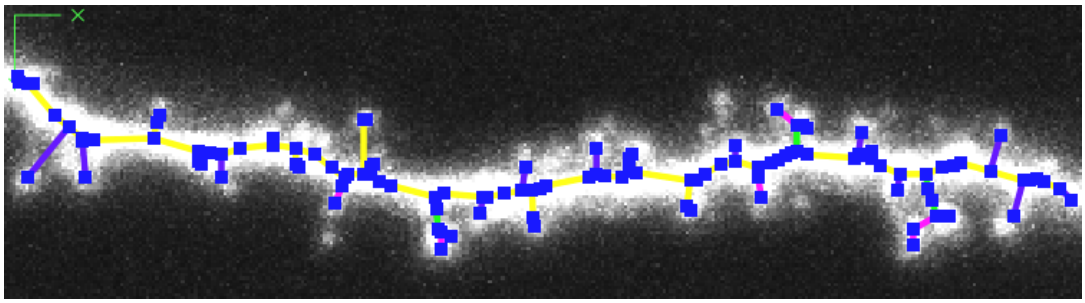


(d)

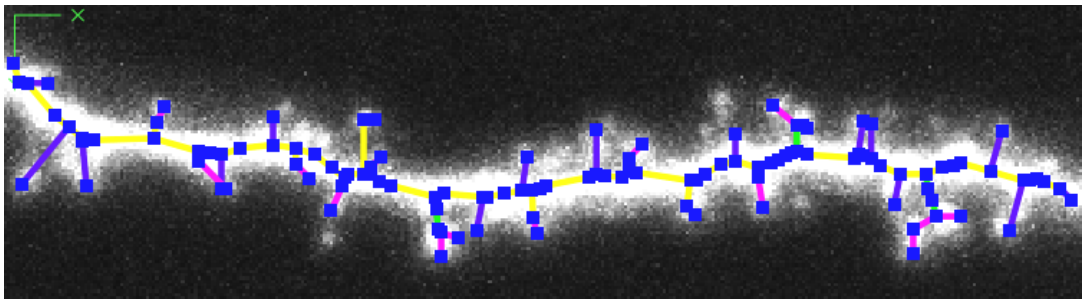
Abbildung B.1: *Ablauf der Rekonstruktion eines Dendritensegments mit* SpineLab*: (a) Volumenprojektion des Originalbildes. (b) Nach Segmentierung. (c) Nach Erzeugung der Mittelachse. (d) Automatisch generiertes Skelett.*



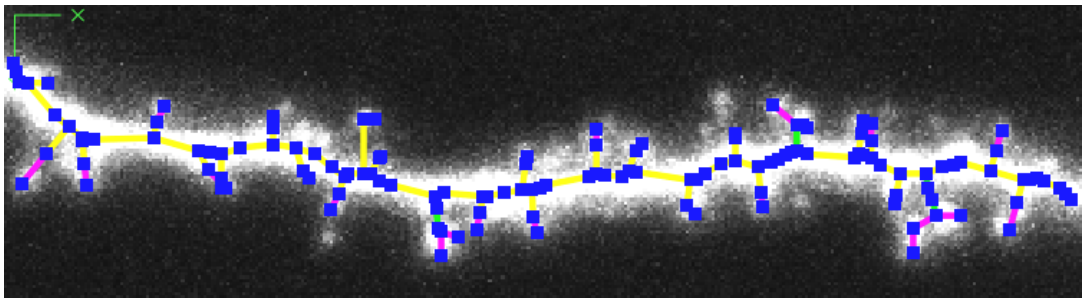
(a)



(b)



(c)



(d)

Abbildung B.2: *Ablauf der Rekonstruktion eines Dendritensegments mit* SpineLab *(Fortsetzung): (a) Identifizierung des Dendriten (gelb). (b) Isolierte Spines werden automatisch mit dem Dendriten verbunden. (c) Die Enden der Spines werden an das Ende des segmentierten Bereichs verschoben. (d) Die Startpunkte der Spines werden nachkorrigiert.*

5. Anschließend werden alle isolierten Vertices oder isolierten Teilskelette, die häufig Spineköpfe repräsentieren, mit dem Dendriten verbunden (Abbildung B.2b). Hierbei wird die kürzest mögliche Kante zum Dendriten eingefügt.

6. Die Enden der Spines, die noch mittig in den Spineköpfen liegen, werden auf den Rand des segmentierten Bildbereichs verschoben.

7. Mit Hilfe von Tangenten, die an den Rand des segmentierten Dendriten angelegt werden, werden die Startpunkte der Spines korrigiert. Dieser Teil des Verfahren ist allerdings fehleranfällig gegenüber Rauschen in den Bilddaten, weshalb die Startpunkte der Spines häufig manuell nachkorrigiert werden müssen.

B.1.2 Manuelle Nachbearbeitung des Skeletts

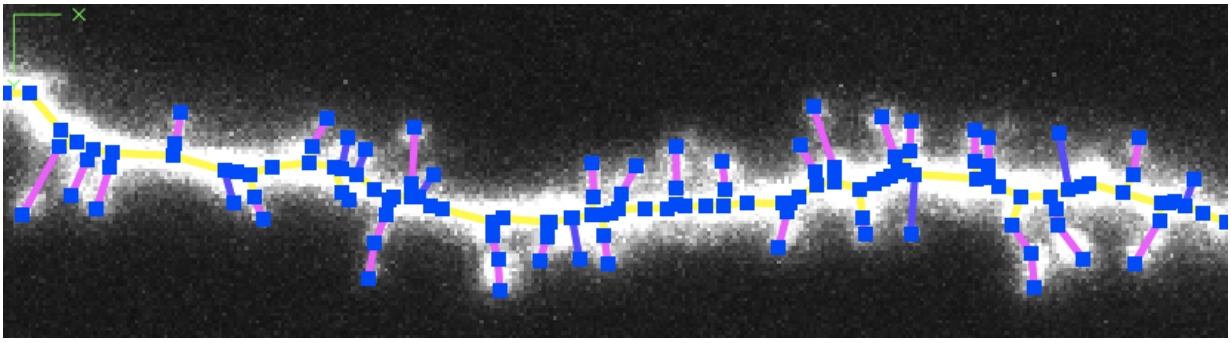


Abbildung B.3: *Das Dendritensegment aus den Abbildungen B.1 und B.2 nach manueller Nachbearbeitung.*

Das Automatisch generierte Skelett kann anschließend beliebig manuell nachbearbeitet werden, um eventuelle Fehler oder Unvollständigkeiten in der Rekonstruktion zu beheben (Abbildung B.3). *SpineLab* stellt dafür folgende Möglichkeiten zur Verfügung:

- Hinzufügen und Löschen von Vertices und Kanten
- Verschieben von Vertices in allen drei Raumrichtungen
- Verschieben und Löschen von markierten Bereichen
- Manuelle Auswahl des Dendriten
- Hinzufügen neuer Spines durch Markierung des Spineendpunktes

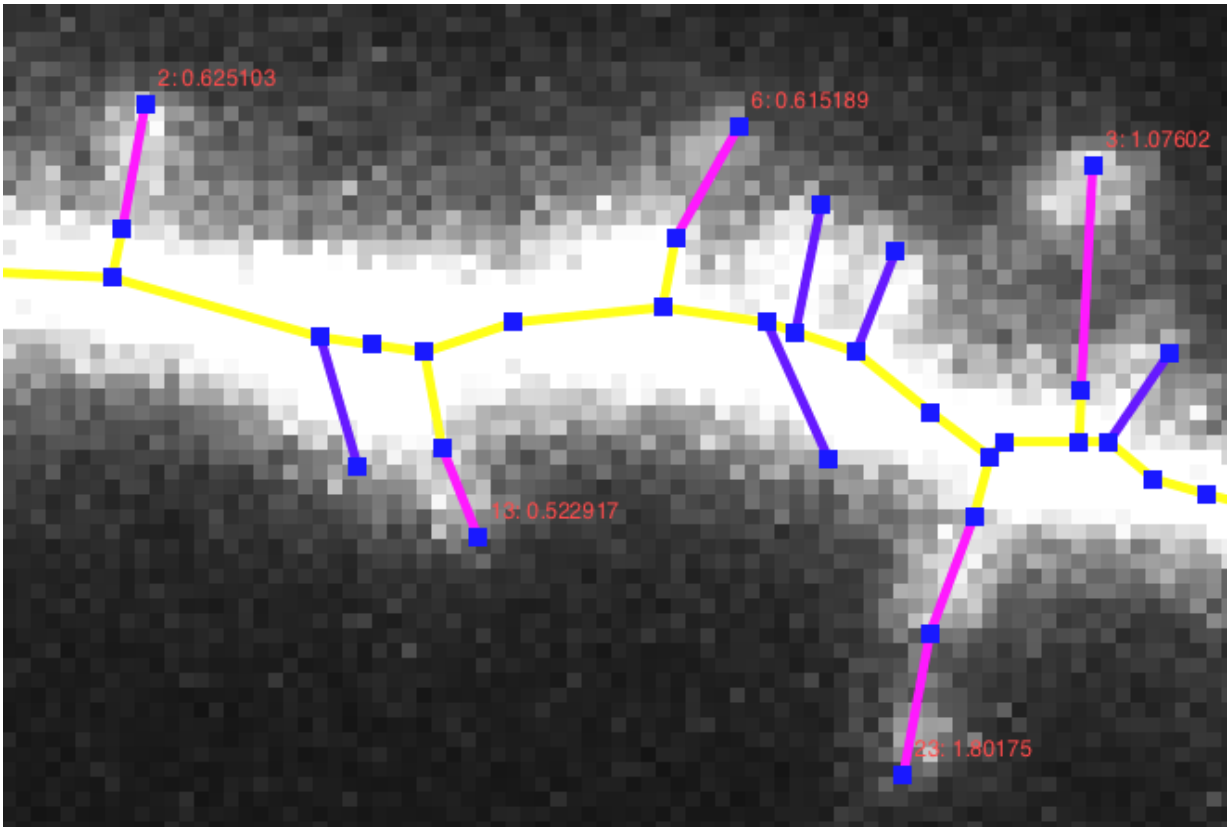


Abbildung B.4: *Automatische Analyse eines rekonstruierten Dendritensegments.*

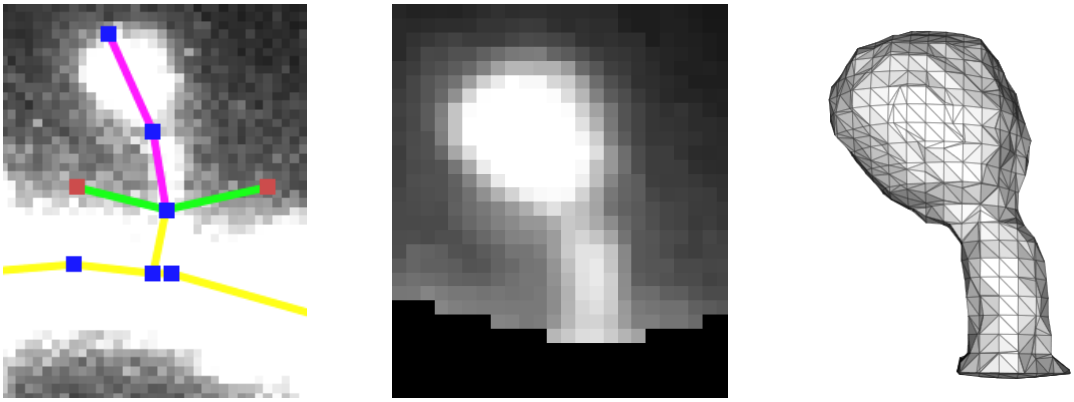
B.1.3 Automatische Längen- und Populationsanalyse

Nach der Erzeugung des Skeletts wird dieses automatisch analysiert: Hierbei wird die Anzahl an Spines und nach Definitionen 13.3 und 13.4 ihre Länge berechnet. Spines, deren Länge berechnet wurde, werden automatisch durchnummeriert. Es wird ihre Länge in Mikrometern und die Nummer des Spines angezeigt (Abbildung B.4). Zudem wird eine Auflistung aller Spines mit entsprechender Länge, die durchschnittliche Länge und die summierte Länge aller Spines generiert. Die Analyse kann als Dendrogramm abgespeichert oder direkt ausgedruckt werden.

B.1.4 Volumenanalyse einzelner Spines

Um das Volumen einzelner Spines berechnen zu können, werden diese vom Dendriten abgeschnitten und mit Hilfe des *NeuRA2*-Verfahrens mit folgender Konfiguration rekonstruiert:

- Medianfilter
- Globale Segmentierung
- Marching-Cubes-Gittererzeugung

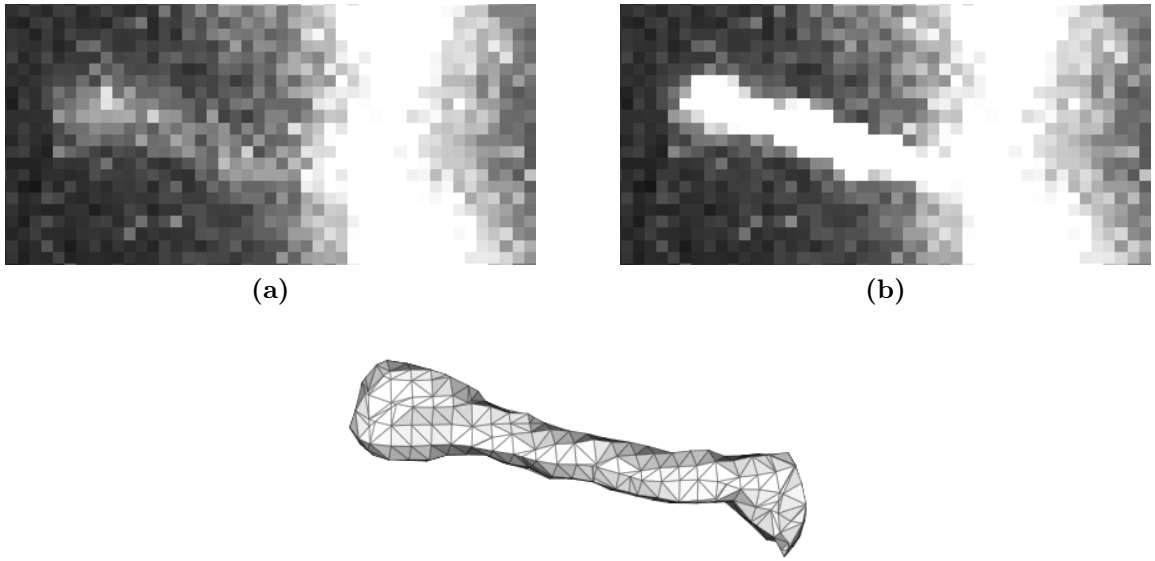


(a) **(b)** **(c)**

Abbildung B.5: *Volumenanalyse einzelner Spines: (a) Der Spine wird entlang der grünen Kanten vom Dendriten abgeschnitten. (b) Abgeschnittener und vorgefilterter Spine. (c) Rekonstruierter Spine.*

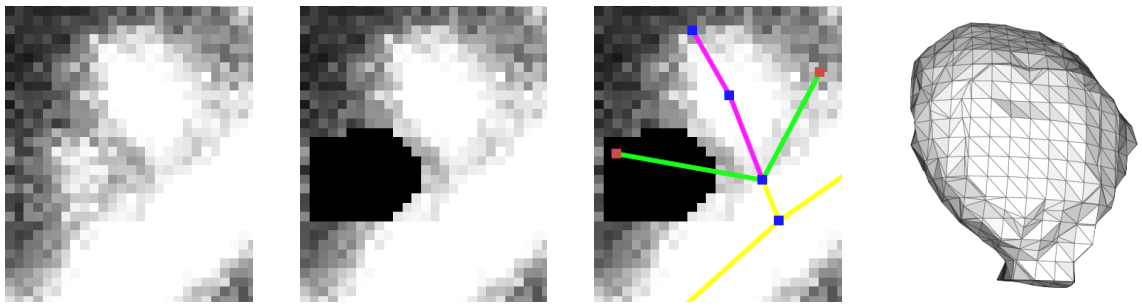
Der Medianfilter verringert das grobkörnige Rauschen, wenn das Originalbild nicht bereits vorverarbeitet wurde. Bei der globalen Segmentierung wird derselbe Parameter verwendet, der bei der Segmentierung zur Skeletterzeugung eingestellt wurde. Anschließend wird der Bildausschnitt, der den Spine enthält, mit dem Isoflächen-Schwellwert 157 (vgl. Kapitel 11.3.3) durch den Marching-Cubes-Algorithmus rekonstruiert und das Volumen nach dem Verfahren von Gauß berechnet (Kapitel 11.2).

Ein häufiges Problem bei der Oberflächenrekonstruktion einzelner Spines ist, dass diese im Vergleich zum Dendriten teilweise sehr dunkel sind (Abbildung B.6a). *SpineLab* stellt daher die Möglichkeit zur Verfügung solche Bildbereiche aufzuhellen. Dabei wird der entsprechende Bildausschnitt markiert und jedes Voxel mit dem, vom Benutzer einzustellenden, Aufhelloperator (Kapitel 4.2.2) bearbeitet (Abbildung B.6b). Anschließend ist die Rekonstruktion und die Volumenanalyse solcher Spines möglich (Abbildung B.6c). Ein weiteres Problem sind nahe beieinander liegende Spines (Abbildung B.7a). Das lokal rekonstruierte Oberflächengitter zur Volumenanalyse kann in einem solchen Fall aus zwei oder mehreren Spines bestehen. *SpineLab* bietet daher die Möglichkeit einzelne Spines auf Bildebene zu löschen (Abbildung B.7b), damit anschließend eine korrekte Volumenanalyse des gewünschten Spines möglich ist (Abbildungen B.7c, d). Bei einer anschließend durchgeführten Populationsanalyse ist natürlich zu beachten, dass einzelne Spines gelöscht wurden.



(a) **(b)** **(c)**

Abbildung B.6: *Aufhellen dunkler Spines: (a) Manche Spines sind sehr dunkel, was eine Oberflächenrekonstruktion erschwert. (b) Aufgehellter Spine. (c) Rekonstruierter Spine.*

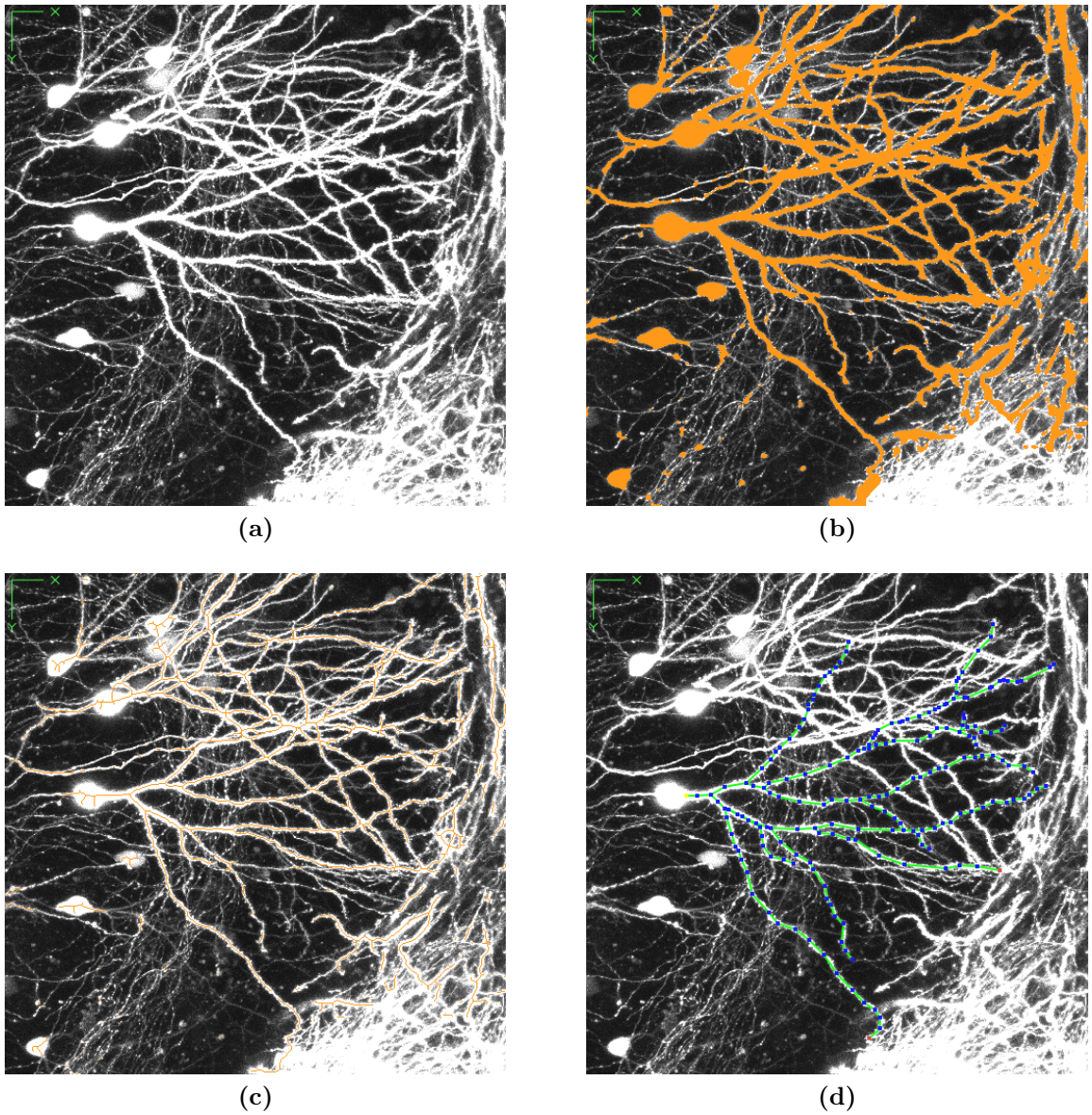


(a) **(b)** **(c)** **(d)**

Abbildung B.7: *Entfernen von Spines bei der Volumenanalyse: (a) Nahe beieinander liegende Spines können die Volumenanalyse erschweren. (b) Löschen des störenden Spines. (c) Skelett zur Volumenanalyse. (d) Rekonstruierter Spine.*

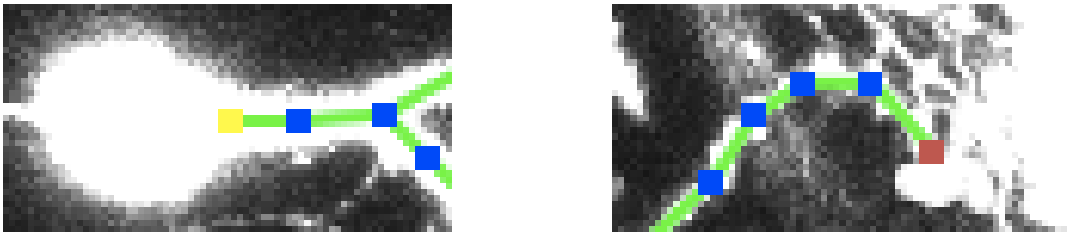
B.2 Rekonstruktion von Neuronenzellen

Die in Abschnitt B.1 beschriebene Rekonstruktion von Dendritensegmenten kann ebenfalls zur Rekonstruktion ganzer Neuronenzellen verwendet werden. Zur Rekonstruktion einzelner Zellen aus einer überlagerten Aufnahme mehrerer einzelner Zellen stellt *SpineLab* eine weitere semiautomatische Rekonstruktionsmethode zur Verfügung. Zunächst wird das Bild segmentiert und die Mittelachse der Dendriten mit Hilfe des *AFM*-Verfahrens wie oben beschrieben erzeugt. Anschließend kann der Benutzer die Mittelachsen der zu rekonstru-



(a) **(b)** **(c)** **(d)**

Abbildung B.8: *Rekonstruktion einzelner Neuronenzellen: (a) Volumenprojektion einer Aufnahme mit mehreren Neuronenzellen. (b) Segmentierung der Aufnahme. (c) Bestimmung der Mittelachsen. (d) Rekonstruktion als Merkmalskelett.*



(a) **(b)**

Abbildung B.9: *(a) Der Zellkern (Soma) wird vom Benutzer markiert und dadurch gelb gefärbt. (b) Dendritenenden, die durch andere Zellen oder Rauschen im Bild überlagert sind, können braun markiert werden und werden dadurch in der Längenanalyse nicht berücksichtigt.*

ierenden Dendriten abfahren, wodurch das Merkmalskelett dieser Dendriten erzeugt wird. An Stellen bei denen sich mehrere Dendriten von verschiedenen Zellen überlagern, kann der Benutzer sehr exakt den Verlauf der Dendriten nachbilden, die zur rekonstruierenden Zelle gehören. Das Merkmalskelett der Zelle entspricht graphentheoretisch [115] einem Baum (vgl. Definition 13.6). Die Wurzel des Baums symbolisiert das Soma der Zelle, die Blätter die Enden der Dendriten. Nachdem das Soma per Hand ausgewählt wurde (Abbildung B.9a) kann eine automatische Längenanalyse der rekonstruierten Zelle vorgenommen werden. Hierbei wird ausgehend vom Soma die Summe der Länge der Kanten zwischen den Verzweigungspunkten, sowie die Summe der Länge der Kanten bis zu den Enden der Dendriten berechnet. Das Ergebnis wird als Dendrogramm (vgl. Abbildung 13.13) gespeichert. Dendritenenden, die durch andere Zellen oder Rauschen im Bild überlagert werden, können vor der Längenanalyse vom Benutzer markiert werden (Abbildung B.9b) und werden nicht in der Längenanalyse berücksichtigt. Zudem wird der Winkel zwischen den Verzweigungspunkten berechnet (vgl. Kapitel 13.3.2).

B.3 Benutzeroberfläche von *SpineLab*

Die Benutzeroberfläche von *SpineLab* (Abbildung B.10) basiert auf der grafischen Oberfläche von *NeuRA2* und ist ebenfalls in *Qt4* [17] programmiert. Sie beinhaltet die in den vorherigen Abschnitten dieses Kapitels beschriebenen Algorithmen und Funktionen zur Rekonstruktion und Analyse neurobiologischer Mikroskopdaten.

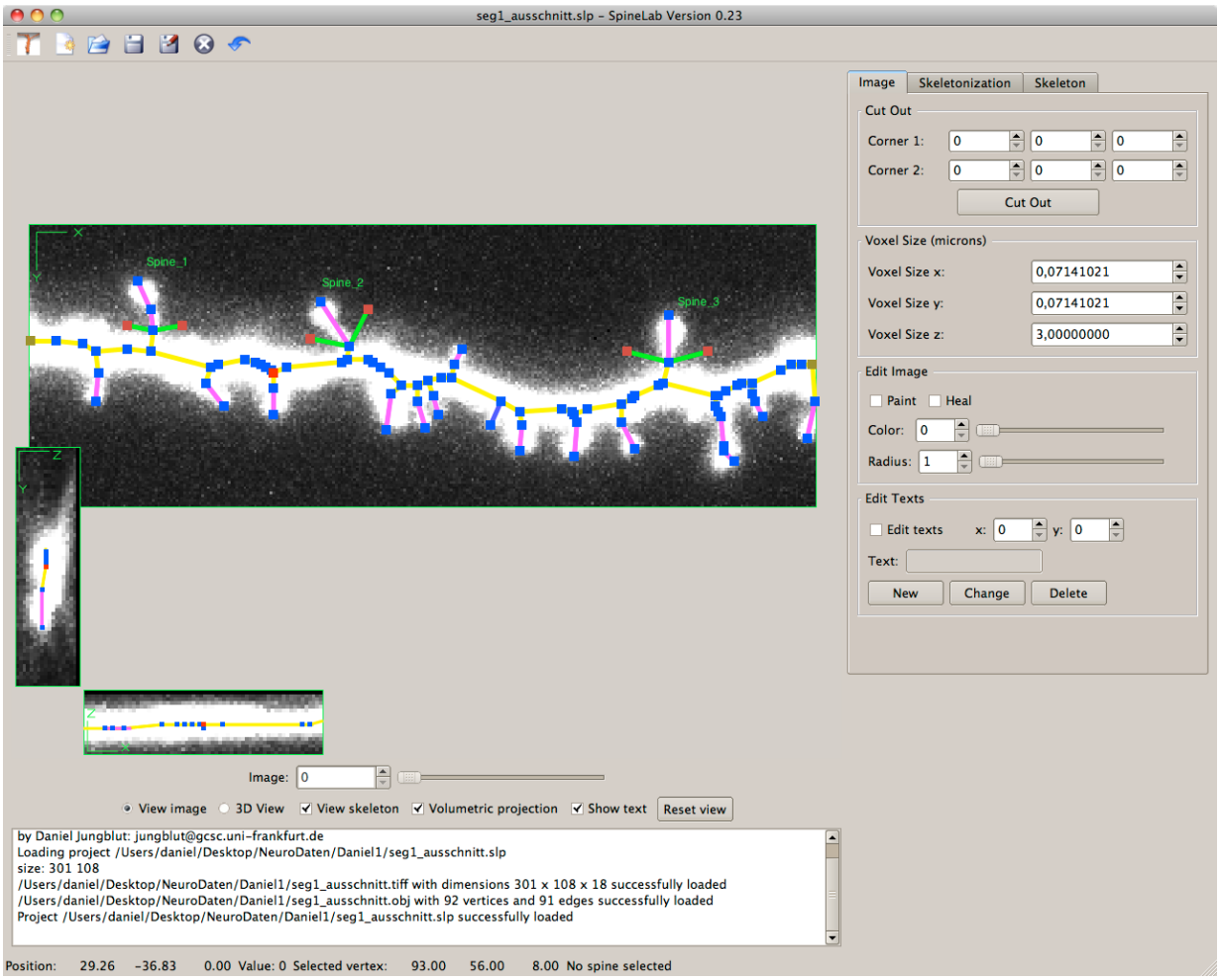


Abbildung B.10: *Die Benutzeroberfläche von* SpineLab.

Anhang C

Referenzen der archäologischen Objekte

CT Nummer	Objektidentifizierungsnummer	Keramiktyp	Gefäßform
CT01184	LMJ ARCH Inv. Nr. 4214	Protokorinthisch	Ovoider Aryballos
CT02183	LMJ ARCH Inv. Nr. 25306	Korinthisch	Kugelaryballos
CT02184	LMJ ARCH Inv. Nr. 25304	Korinthisch	Kugelaryballos
CT02185	LMJ ARCH Inv. Nr. 25278	Korinthisch	Kugelaryballos
CT02186	LMJ ARCH Inv. Nr. 4215	Korinthisch	Kugelaryballos
CT02187	LMJ ARCH Inv. Nr. 8738	Korinthisch	Kugelaryballos
CT02188-1	KFUG IA Inv. Nr. 804	Nordionisch	Kotyle
CT02188-2	KFUG IA Inv. Nr. 844	Attisch	Kyatos
CT02188-3	KFUG IA Inv. Nr. 788	Korinthisch	Kugelaryballos
CT02188-4	KFUG IA Inv. Nr. 808	Nordionisch	Kotyle
CT03046	UMJ ARCH Inv.-Nr. 25281	Este-Kultur	Becher
CT03047	KFUG IA Inv. Nr. G 56	Ostgriechisch	Granatapfelgefäß
CT03145	LMJ ARCH Inv. Nr. 25277	„Protokor. Stil“	Ovoider Aryballos
CT03155	LMJ ARCH Inv. Nr. 4190	Protokorinthisch	Kugeliger Aryballos
CT03210	LMJ ARCH Inv. Nr. 25301	Korinthisch	Kugelaryballos
CT03211	LMJ ARCH Inv. Nr. 25303	Korinthisch	Kugelaryballos
CT03212	LMJ ARCH Inv. Nr. 25305	Korinthisch	Kugelaryballos
CT03213	LMJ ARCH Inv. Nr. 25302	Korinthisch	Kugelaryballos
CT03214	Deutschfeistritz Privatslg.	Korinthisch	Alabastron
CT03244	Modernes Alabastron	Modern	Alabastron
CT03245	Moderne Schale	Modern	Schale
CT04136	KFUG IA Inv. Nr. G 26	Korinthisch	Kugelaryballos

Tabelle C.1: *Verknüpfung zwischen den CT-Nummern des Österreichischen-Gießerei-Instituts (ÖGI) und den Objektidentifizierungsnummern der Sammlungen, sowie Keramiktyp und Gefäßform.*

Anhang D

Einfluss der Gitteroptimierung auf das Gittervolumen

Anzahl Dreiecke	Volumen in ml	Abweichung
455924	92.035	0%
232270	92.036	0.0005%
18380	92.059	0.025%
1676	92.072	0.04%
318	92.095	0.065%

Tabelle D.1: *Einfluss der Gitteroptimierung – Alabastron (CT03244).*

Anzahl Dreiecke	Volumen in ml	Abweichung
726512	64.996	0%
347870	64.944	0.08%
34288	64.954	0.065%
2788	64.962	0.051%
362	64.986	0.015%

Tabelle D.2: *Einfluss der Gitteroptimierung – Schale (CT03245).*

Anzahl Dreiecke	Volumen in ml	Abweichung
832784	44.538	0%
357562	44.422	0.26%
29512	44.41	0.28%
3028	44.416	0.27%
672	44.403	0.3%

Tabelle D.3: *Einfluss der Gitteroptimierung – CT01184.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1405688	123.366	0%
547206	123.221	0.123%
44040	123.199	0.14%
4048	123.209	0.13%
854	123.232	0.11%

Tabelle D.4: *Einfluss der Gitteroptimierung – CT02183.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1433292	108.458	0%
591340	108.311	0.14%
47320	108.285	0.16%
4304	108.3	0.15%
854	108.292	0.15%

Tabelle D.5: *Einfluss der Gitteroptimierung – CT02184.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1302856	101.881	0%
617082	101.752	0.13%
49044	101.732	0.15%
4304	101.724	0.15%
824	101.733	0.15%

Tabelle D.6: *Einfluss der Gitteroptimierung – CT02185.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1359512	80.911	0%
574742	80.806	0.13%
46186	80.796	0.14%
4132	80.772	0.17%
842	80.808	0.13%

Tabelle D.7: *Einfluss der Gitteroptimierung – CT02186.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1215024	56.053	0%
503200	55.964	0.16%
40074	55.953	0.18%
3534	55.946	0.19%
730	55.955	0.18%

Tabelle D.8: *Einfluss der Gitteroptimierung – CT02187.*

Anzahl Dreiecke	Volumen in ml	Abweichung
859076	86.049	0%
636430	85.818	0.27%
52378	85.862	0.22%
4604	85.845	0.24%
848	85.814	0.28%

Tabelle D.9: *Einfluss der Gitteroptimierung – CT03047.*

Anzahl Dreiecke	Volumen in ml	Abweichung
536048	36.27	0%
270128	36.167	0.29%
21598	36.158	0.3%
1938	36.165	0.29%
426	36.152	0.32%

Tabelle D.10: *Einfluss der Gitteroptimierung – CT03145.*

Anzahl Dreiecke	Volumen in ml	Abweichung
854372	74.377	0%
505150	74.235	0.19%
41678	74.239	0.18%
4156	74.255	0.16%
912	74.239	0.19%

Tabelle D.11: *Einfluss der Gitteroptimierung – CT03155.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1021984	110.379	0%
630800	110.207	0.16%
49944	110.21	0.15%
4164	110.222	0.14%
728	110.207	0.16%

Tabelle D.12: *Einfluss der Gitteroptimierung – CT03210.*

Anzahl Dreiecke	Volumen in ml	Abweichung
707864	61.412	0%
522024	61.237	0.29%
44508	61.272	0.23%
3642	61.274	0.23%
676	61.284	0.21%

Tabelle D.13: *Einfluss der Gitteroptimierung – CT03211.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1034708	103.697	0%
630520	103.54	0.15%
50330	103.532	0.16%
4624	103.546	0.15%
854	103.538	0.15%

Tabelle D.14: *Einfluss der Gitteroptimierung – CT03212.*

Anzahl Dreiecke	Volumen in ml	Abweichung
669152	55.86	0%
514414	55.701	0.29%
43538	55.744	0.21%
3618	55.741	0.21%
642	55.741	0.21%

Tabelle D.15: *Einfluss der Gitteroptimierung – CT03213.*

Anzahl Dreiecke	Volumen in ml	Abweichung
579356	39.841	0%
255608	39.767	0.19%
20694	39.755	0.21%
2068	39.772	0.17%
470	39.822	0.05%

Tabelle D.16: *Einfluss der Gitteroptimierung – CT03214.*

Anzahl Dreiecke	Volumen in ml	Abweichung
1021296	243.862	0%
601392	243.47	0.16%
48158	243.444	0.17%
4380	243.414	0.18%
840	243.291	0.23%

Tabelle D.17: *Einfluss der Gitteroptimierung – CT04136.*

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Adobe: *Tiff Revision 6.0*. http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf, 1992. – Zuletzt besucht am 12.03.2009

[2] Akeley, K. ; Nguyen, H.: *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2007

[3] Alboul, L. ; Damme, R. van: Polyhedral metrics in surface reconstruction. In: *The Mathematics of Surfaces* VI (1996), S. 171–200

[4] AMD: *AMD Stream Computing User Guide*. http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf, 2009. – Zuletzt besucht am 08.07.2010

[5] Ansys: *Ansys ICEM CFD*. http://www.ansys.com/products/icemcfd.asp, 2009. – Zuletzt besucht am 10.11.2009

[6] Apple: *Apple Previews Mac OS X Snow Leopard to Developers*. http://www.apple.com/pr/library/2008/06/09snowleopard.html, 2008. – Zuletzt besucht am 15.01.2009

[7] Au, O. K. C. ; Tai, C. L. ; Chu, H. K. ; Cohen-Or, D. ; Lee, T. Y.: Skeleton Extraction by Mesh Contraction. In: *ACM Transactions on Graphics* 27 (2008), Nr. 3

[8] Aumüller, M. ; R. Lang, D. R. ; Schulze-Döbold, J. ; Werner, A. ; Wolf, P. ; Wössner, U.: *Covise Online Documentation*. http://vis.uni-koeln.de/covise/doc/html/tutorial/, 2008. – Zuletzt besucht am 01.10.2009

[9] Bär, C.: *Elementare Differentialgeometrie*. 2. Auflage. de Gruyter Lehrbuch, 2010

[10] Bastian, P. ; Birken, K. ; Lang, S. ; Johanssen, K. ; Neuss, N. ; Rentz-Reichert, H. ; Wieners, C.: UG - A flexible software toolbox for solving partial differential equations. In: *Computing and Visualization in Science* 1 (1997), S. 27–40

[11] Bauer, C.: *Extraktion und Modellherstellung von Gefäßbäumen aus medizinischen Volumendaten*, Universität Koblenz – Landau, Diplomarbeit, 2005

[12] Becker, R. ; Braack, M. ; Dunne, T. ; Meidner, D. ; Rannacher, R. ; Richter, T. ; Schmich, M. ; Wollner, W. ; Vexler, B.: *High Perfomance Adaptive Finite Element Toolkit Gascoigne*. http://gascoigne.uni-hd.de, 2010. – Zuletzt besucht am 13.04.2010

[13] Bentz, M. ; Böhr, E.: Zu den Maßen attischer Feinkeramik. In: *Beihefte zum Corpus Vasorum Antiquorum (CVA) Deutschland* 1 (2002), S. 73–80

[14] Bewersdorff, J.: *Algebra für Einsteiger*. 3. Auflage. Vieweg, 2007

[15] Bilke, L. ; Rink, K.: *OpenGeoSys*. http://www.ufz.de/index.php?en=18345, 2010. – Zuletzt besucht am 14.04.2010

[16] Bioscience mbf: *Neurolucida - advanced software for neuron reconstruction, 3D mapping, and morphometry*. http://www.mbfbioscience.com/neurolucida/, 2010. – Zuletzt besucht am 25.08.2010

[17] Blanchette, J. ; Summerfield, M.: *C++ GUI Programming with Qt 4*. Prentice Hall, 2006

[18] Bourke, P.: *Polygonising a scalar field*. http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/, 1994. – Zuletzt besucht am 10.11.2009

[19] Broser, P. J.: *Morphologische Bildoperatoren für die quantitative Neurobiologie*, Universität Heidelberg, Diss., 2006

[20] Broser, P.J. ; Eberhard, S. ; Heumann, H. ; Heusel, A. ; Jungblut, D. ; Queisser, G. ; Schulte, R. ; Vossen, C. ; Wittum, G.: *The Neuron Reconstruction Algorithm*. http://www.neura.org, 2005. – Zuletzt besucht am 01.10.2009

[21] Broser, P.J. ; Schulte, R. ; Roth, A. ; Helmchen, F. ; Waters, J. ; Lang, S. ; Sakmann, B. ; Wittum, G.: Nonlinear Anisotropic Diffusion Filtering of Three-Dimensional Image Data from 2-Photon-Microscopy. In: *Journal of Biomedical Optics* 9(6) (2004), S. 1253–1264

[22] Brunner, D.: *Segmentierung und Klassifizierung von 3d-Objekten: Skelettierung von 3d-Objekten auf kubisch-raumzentrierten Gittern und deren Anwendungen*. Vdm Verlag Dr. Müller, 2008

[23] Buck, I. ; Foley, T. ; Horn, D. ; Sugerman, J. ; Fatahalian, K. ; Houston, M. ; Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware. In: *ACM SIGGRAPH*, 2004, S. 19–26

[24] Claus, J. ; Friedmann, E. ; Klingmüller, U. ; Rannacher, R. ; Szekeres, T.: Spatial aspects in the SMAD signaling pathway. In: *J. Math. Biol.* (2012). – DOI 10.1007/s00285-012-0574-1

[25] Cui, X. G. ; Gutheil, E.: LES Study on the Flow-Field in an Idealized Human Mouth-Throat Model. In: *SPRAY 2010 - 9. Workshop über Sprays, Techniken der Fluidzerstäubung und Untersuchung von Sprühvorgängen*, 2010

[26] Damme, R. van ; Alboul, L.: Tight triangulations. In: *Mathematical Methods for Curves and Surfaces* (1995), S. 517–526

[27] Dartu, C. ; Krömker, S. ; Rings, J.: *Vrend Tutorial for Version 2.0.* `www.iwr.uni-heidelberg.de/groups/ngg/Vrend/vrend/VrendTutorial2.0.2.pdf`, 2004. – Zuletzt besucht am 10.11.2009

[28] Dressel, A.: *Die nichtlineare Diffusion in der Bildverarbeitung*, Universität Heidelberg, Diplomarbeit, 1999

[29] Dyn, N. ; Hormann, K. ; Kim, S.-J. ; Levin, D.: Optimizing 3D Triangulations Using Discrete Curvature Analysis. In: Lyche, T. (Hrsg.) ; Schumaker, L. L. (Hrsg.): *Mathematical Methods for Curves and Surfaces: Oslo 2000.* Nashville, TN : Vanderbilt University Press, 2001 (Innovations in Applied Mathematics), S. 135–146

[30] Eberle, T. ; Lampel, J.: *VVis Volume Visualisation.* `http://vvis.sourceforge.net/download/vvis-0.1-manual.pdf`, 2005. – Zuletzt besucht am 10.11.2009

[31] Eberly, D.: *Polyhedral Mass Properties.* `http://www.geometrictools.com/`, 2009. – Zuletzt besucht am 21.05.2010

[32] Fangerau, J.: *Volume Rendering auf Grafikkarten und parallele Implementierung in Cuda*, Universität Heidelberg, Diplomarbeit, 2009

[33] Fangerau, J. ; Krömker, S.: Parallel Volume Rendering Implementation on Graphics Cards using CUDA. In: *Heidelberg Academy of Sciences, Facing the Multicore-Challenge* Bd. 6310, Springer LNCS Series, 2010

[34] Fernando, R.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Addison-Wesley, 2004

[35] Fernando, R. ; Kilgard, M. J.: *The Cg Tutorial.* Addison-Wesley, 2003

[36] Fischer, G.: *Lineare Algebra.* 15. Auflage. Vieweg und Teubner, 2005

[37] Frangi, A. F. ; Niessen, W. J. ; Vincken, K. L. ; Viergever, M.A.: Multiscale Vessel Enhancement Filtering. In: *Lecture Notes in Computer Sciences* 1496 (1998), S. 130–137

[38] Ghuloum, A. ; Sprangle, E. ; Fang, J. ; Wu, G. ; Zhou, X.: *Ct: A Flexible Parallel Programming Model for Tera-scale Architectures.* Intel, 2007

[39] GPGPU: *General Purpose Computation Using Graphics Hardware.* `http://www.gpgpu.org`, 2002–2010. – Zuletzt besucht am 10.11.2009

[40] Greene, A. ; Hartley, C.: *From Analog to Digital: Protocols and Program for a Systematic Digital Radiography of Archaeological Pottery*. 2008

[41] Gropp, W. ; Skjellum, A. ; Lusk, E.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*. 2. Auflage. MIT Press, 1999

[42] Hackbusch, W.: *Iterative Lösung großer schwachbesetzter Gleichungssysteme*. 2. Auflage. Teubner, 1993

[43] Hackbusch, W.: *Theorie und Numerik elliptischer Differentialgleichungen*. 2. Auflage. Teubner, 1996

[44] Hill, F. S.: *Computer Graphics Using OpenGL*. 2. Auflage. Prentice Hall International, 2003

[45] Hoppe, H. ; DeRose, T. ; Duchamp, T. ; McDonald, J. ; Stuetzle, W.: Mesh Optimization. In: *ACM SIGGRAPH*, 1993, S. 19–26

[46] Hoppe, H. ; DeRose, T. ; Duchamp, T. ; McDonald, J. ; Stuetzle, W.: *Mesh Optimization*. Dept. of Computer Science and Engineering, University of Washington, 1993

[47] Hounsfield, G. N.: Computerized transverse axial scanning (tomography): Part I. Description of system. In: *British Journal of Radiology* 46 (1973), S. 1016–1022

[48] ImageJ: *ImageJ – Image Processing and Analysis in Java*. `http://rsb.info.nih.gov/ij/`, 2009. – Zuletzt besucht am 09.11.2009

[49] Jähne, B.: *Digitale Bildverarbeitung*. 6. Auflage. Springer, 2005

[50] Jungblut, D.: *Trägheitsbasiertes Filtern mikroskopischer Messdaten unter Verwendung moderner Grafikhardware*, Universität Heidelberg, Diplomarbeit, 2007

[51] Jungblut, D. ; Karl, S. ; Mara, H. ; Krömker, S. ; Wittum, G.: Surface Morphology Reconstruction of Volume Data for Archaeology. In: *Scientific Computing and Cultural Heritage – Contributions in Computational Humanities* Kapitel 5 (2012), S. 42–49

[52] Jungblut, D. ; Queisser, G. ; Wittum, G.: Inertia based filtering of high resolution images using a GPU cluster. In: *Computing and Visualization in Science* 14 (2012), S. 181–186

[53] Jungblut, D. ; Vlachos, A. ; Schuldt, G. ; Zahl, N. ; Deller, T. ; Wittum, G.: SpineLab: tool for three-dimensional reconstruction of neuronal cell morphology. In: *Journal of Biomedical Optics* 17(7) (2012)

[54] Kanzow, C.: *Numerik linearer Gleichungssysteme*. Springer, 2005

[55] Karl, S.: Durchleuchtungen griechischer Keramik - Industrielle 3D Röntgen-Computertomographie als archäometrische Methode. In: *Schriften der Archäologie und Archäometrie der Paris-Lodron-Universität Salzburg, Band 1*, 2009

[56] Karl, S. ; Jungblut, D. ; Mara, H. ; Wittum, G. ; Krömker, S.: Insights into manufacturing techniques of archaeological pottery: Industrial X-ray computed tomography as a tool in the examination of cultural material. In: *European Meeting on Ancient Ceramics*, 2009. – Im Druck befindlich.

[57] Karl, S. ; Jungblut, D. ; Rosc, J. ; Trinkl, E.: Berührungsfreie und nicht invasive Untersuchung antiker Keramik mittels industrieller Röntgen-Computertomografie. Mit einem Beitrag von Rudolf Erlach. In: *Interdisziplinäre Dokumentations- und Visualisierungsmethoden, CVA Österreich Beiheft 1.* – Erscheint 2013

[58] Königsberger, K.: *Analysis II.* Springer, 2005

[59] Krömker, S.: *Computergraphik II.* Vorlesungsskriptum, Universität Heidelberg, 2008

[60] Kuypers, F.: *Klassische Mechanik.* 5. Wiley-Vch, 1997

[61] Lenzen, F.: *3D-Rekonstruktion von DNA-Strukturen*, Universität Bonn, Diplomarbeit, 2001

[62] LibTIFF: *LibTIFF – TIFF Library and Utilities.* `http://www.libtiff.org/`, 2003. – Zuletzt besucht am 10.11.2009

[63] Lindeberg, T.: Edge detection and ridge detection with automatic scale detection. In: *IEEE Conference on Computer Vision and Pattern Recognition*, 1996, S. 465–470

[64] Lorensen, W. E. ; Cline, Harvey E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *Computer Graphics* 21 (1987), S. 163–169

[65] Lorenz, T.: *Mutational Analysis - A Joint Framework for Cauchy Problems* In *And* Beyond *Vector Spaces.* Springer, 2010

[66] Marin, S. ; Smith, P.: Parametric approximation of data using ODR splines. In: *Computer Aided Geometric Design* 11(3) (1994), S. 247–267

[67] Mat-Isa, N. A. ; Mashor, M. Y. ; Othman, N. H.: Seeded Region Growing Features Extraction Algorithm; Its Potential Use in Improving Screening for Cervical Cancer. In: *International Journal of the Computer, the Internet and Management* 13 (2005), S. 61–70

[68] Mrazek, P.: *Nonlinear Diffusion for Image Filtering and Monotonicity Enhancement*, Czech Technical University, Diss., 2001

[69] Müller, C. M. ; Vlachos, A. ; Deller, T.: Calcium homeostasis of acutely denervated and lesioned dentate gyrus in organotypic entorhino-hippocampal cocultures. In: *Cell Calcium* 47 (2010), S. 242–252

[70] Müller, F. ; Wach, P. ; McNally, J. G.: Evidence for a common mode of transcription factor interaction with chromatin as revealed by improved quantitative fluorescence recovery after photobleaching. In: *Biophys Journal* 94 (2008), Nr. 8, S. 3323–3339

[71] Munshi, A.: *The OpenCL Specification*. Khronos OpenCL Working Group, 2008

[72] Neumann, R.: *Räumliche Aspekte in der Signaltransduktion*, Universität Heidelberg, Diplomarbeit, 2009

[73] Nvidia: *Nvidia Cuda Programming Guide Version 1.1*. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf, 2007. – Zuletzt besucht am 08.07.2010

[74] Nvidia: *cuBLAS Documentation Version 2.0*. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf, 2008. – Zuletzt besucht am 08.07.2010

[75] Nvidia: *CUDA-GDB: The Nvidia CUDA Debugger Version 2.1 Beta*. http://developer.download.nvidia.com/compute/cuda/2_1/cudagdb/CUDA_GDB_User_Manual.pdf, 2008. – Zuletzt besucht am 08.07.2010

[76] Nvidia: *CUDA Software Development Kit Version 2.0*. http://developer.nvidia.com/object/cuda_2_0_downloads.html, 2008. – Zuletzt besucht am 08.07.2010

[77] Nvidia: *cuFFT Documentation Version 2.0*. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUFFT_Library_2.0.pdf, 2008. – Zuletzt besucht am 08.07.2010

[78] Nvidia: *Nvidia Cuda Programming Guide Version 2.0*. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, 2008. – Zuletzt besucht am 08.07.2010

[79] Nvidia: *Nvidia Tesla C1060 Specifications*. http://www.nvidia.com/docs/IO/56483/Tesla_C1060_boardSpec_v03.pdf, 2008. – Zuletzt besucht am 08.07.2010

[80] Nvidia: *Fermi Whitepaper Version 1.1*. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. – Zuletzt besucht am 08.07.2010

[81] Nvidia: *Nvidia Cuda Programming Guide Version 2.2*. http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf, 2009. – Zuletzt besucht am 08.07.2010

[82] NVIDIA: *OpenCL JumpStart Guide Version 0.9.* `developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf`, 2009. – Zuletzt besucht am 08.07.2010

[83] NVIDIA: *OpenCL Programming Guide for the CUDA Architecture Version 3.1.* `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf`, 2010. – Zuletzt besucht am 09.09.2010

[84] OPENFOAM: *OpenFOAM - The open source CFD toolbox.* `http://www.openfoam.com/`, 2010. – Zuletzt besucht am 20.05.2010

[85] OTSU, N.: A Threshold Selection Method from Gray-Level Histograms. In: *IEEE, Trans. Systems, Man, and Cybernetics* 9 (1979), S. 62–66

[86] OTTMANN, T. ; WIDMAYER, P.: *Algorithmen und Datenstrukturen.* 4. Auflage. Spektrum Akademischer Verlag, 2002

[87] PERONA, P. ; MALIK, J.: Scale-Space and Edge Detection Using Anisotropic Diffusion. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12 (1990), Nr. 7, S. 629–639

[88] PHARR, M.: *GPU Gems 2: Techniques for Graphics and Compute-Intensive Programming.* Addison-Wesley, 2005

[89] PRESS, W. H. ; FLANNERY, B. P. ; TEUKOLSKY, S. A. ; VETTERLING, W. T.: *Numerical Recipes in C.* 2. Auflage. Cambridge University Press, 1992

[90] QUEISSER, G.: *Rekonstruktion und Vermessung der Geometrie von Neuronen-Zellkernen*, Universität Heidelberg, Diplomarbeit, 2005

[91] QUEISSER, G.: *The Influence of the Morphology of Nuclei from Hippocampal Neurons on Signal Processing in Nuclei*, Universität Heidelberg, Diss., 2008

[92] QUEISSER, G. ; BADING, H. ; WITTMANN, M. ; WITTUM, G.: Filtering, reconstruction, and measurement of the geometry of nuclei from hippocampal neurons based on confocal microscopy data. In: *Journal of Biomedical Optics* 13(1) (2008)

[93] REINELT, G.: *Effiziente Algorithmen I.* Vorlesungsskriptum, Universität Heidelberg, 2007

[94] REITER, S.: *Tools zum Erstellen von glatten Isosurfaces aus Bildstacks.* 2007. – Universität Heidelberg

[95] REZK-SALAMA, C. R. ; ENGEL, K. ; HIGUERA, F. V.: *OpenQVis.* `http://openqvis.sourceforge.net/`, 2002. – Zuletzt besucht am 24.03.2010

[96] Rodriguez, A. ; Ehlenberger, D. B. ; Dickstein, D.L. ; Hof, P.R. ; Wearne, S.L.: Automated Three-Dimensional Detection and Shape Classification of Dendritic Spines from Fluorescence Microscopy Images. In: *PLoS ONE* 3(4) (2008)

[97] Romberg, J. ; Akram, W. ; Gamiz, C.: *Image Segmentation Using Seeded Region Growing*. `http://www.owlnet.rice.edu/~elec539/Projects97/WDEKnow`, 1997. – Zuletzt besucht am 10.11.2009

[98] Rosendaal, T. ; Selleri, S. ; et.al.: *The Official Blender Guide 2.3*. No Strach Press, 2004

[99] Rumpf, M. ; Telea, A.: A Continuous Skeletonization Method Based on Level Sets. In: *Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization*, 2002

[100] Schmitt, S. ; Evers, J. F. ; Duch, C. ; Scholz, M. ; Obermayer, K.: New Methods for the Computer-Assisted 3D Reconstruction of Neurons from Confocal Image Stacks. In: *Neuro Image* 23 (2004)

[101] Schulte, R.: *Filterung von 3D-Neuronenaufnahmen durch nichtlineare anisotrope Diffusion*, Universität Heidelberg, Diplomarbeit, 2003

[102] Schwarz, H. R. ; Köckler, N.: *Numerische Mathematik*. 5. Auflage. Teubner, 2004

[103] Sethian, J. A.: A Fast Marching Level Set Method for Monotonically Advancing Fronts. In: *Proceedings of the National Academy of Sciences of the United States of America* 93 (1996), S. 1591–1595

[104] Seul, M. ; O'Gorman, L. ; Sammon, M.: *Practical Algorithms for Image Analysis*. Cambridge University Press, 2000

[105] Si, H.: *TetGen - A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator*. `http://tetgen.berlios.de/`, 2009. – Zuletzt besucht am 01.10.2009

[106] Sprague, B. L. ; McNally, J. G.: FRAP analysis of binding: proper and fitting. In: *Trends in Cell Biology* 15 (2005), S. 84–91

[107] Sprague, B. L. ; Pego, R. L. ; Stavreva, D. A. ; McNally, J. G.: Analysis of Binding Reaction by Fluorescence Recovery after Photobleaching. In: *Biophysical Journal* 86 (2004), S. 3473–3495

[108] Tanenbaum, A. S.: *Computerarchitektur: Strukturen - Konzepte - Grundlagen*. 5. Auflage. Pearson Studium, 2005

[109] Telea, A. ; Vilanova, A.: A Robust Level-Set Algorithm for Centerline Extraction. In: *Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization*, 2003

[110] Telea, A. ; Wijk, J. van: An Augmented Fast Marching Method for Computing Skeletons and Centerlines. In: *Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization*, 2002

[111] Treece, G. M. ; Prager, R. W. ; Gee, A. H.: Regularised Marching Tetrahedra: Improved Iso-Surface Extraction. In: *Computers and Graphics* 23 (1998), S. 583–598

[112] Vlachos, A. ; Korkotian, E. ; Schonfeld, E. ; Copanaki, E. ; Deller, T. ; Segal, M.: Synaptopodin Regulates Plasticity of Dendritic Spines in Hippocampal Neurons. In: *The Journal of Neuroscience* 29 (2009), S. 1017–1033

[113] Wearne, S.L. ; Ehlenberger, D. B. ; Rocher, A. B. ; Hendersion, S. C. ; Hof, P.R.: New Techniques for imaging, digitization and analysis of three-dimensional neural morphology on multiple scales. In: *Neuroscience* 136 (2005), S. 661–680

[114] Weickert, J.: *Anisotropic Diffusion in Image Processing.* Teubner, 1998

[115] Wilson, R. J.: *Introduction to Graph Theory.* 4. Auflage. Prentice Hall, 1996

[116] Wittum, G.: *Einführung in die Numerische Mathematik.* Vorlesungsskriptum, Universität Heidelberg, 1998

[117] Wittum, G.: *Mehrgitterverfahren.* Vorlesungsskriptum, Universität Heidelberg, 1999

[118] Xylouris, K.: *Signalverarbeitung in Neuronen*, Universität Heidelberg, Diplomarbeit, 2008

[119] Yuste, R. ; Denk, W.: Dendritic spines as basic functional units of neuronal integration. In: *Nature* 375 (1995), S. 682–684

[120] Zorin, D. ; Schröder, P.: Subdivision for Modeling and Animation. In: *SIGGRAPH 2000 Course Notes*, 2000

Lebenslauf

Persönliche Daten

Name:	Daniel Jungblut
Adresse:	Mühlstraße 27 69226 Nußloch
Email:	mail@djungblut.de
Geburtsdatum:	23. April 1984
Geburtsort:	Heidelberg
Nationalität:	deutsch
Sprachkenntnisse:	Englisch (9 Jahre Schule, Sprachkurs in Southampton, Auslandssemester in Leeds) Französisch (5 Jahre Schule)

Bildungs- und Berufsweg

1990-1994:	Albert-Schweitzer-Grundschule in Heidelberg-Pfaffengrund
1994-2003:	Dietrich-Bonhoeffer-Gymnasium Eppelheim. Abitur 2003 mit den Leistungsfächern Mathematik und Physik (Note 1,0)
2003-2008:	Studium der Mathematik an der Universität Heidelberg

Mai 2005:	Vordiplom in Mathematik mit Durchschnittsnote 1,0 (Nebenfach Informatik)
2006-2007:	Auslandssemester in Leeds
Juni 2008:	Diplom in Mathematik mit Note 1,0 mit Auszeichnung (Nebenfach Informatik)
2004-2008:	Studienbegleitend verschiedene Hiwi-Stellen an der Universität Heidelberg
2008-2010:	Wissenschaftlicher Mitarbeiter des Goethe-Zentrums für wissenschaftliches Rechnen (G-CSC) an der Goethe-Universität Frankfurt
2009-2010:	Promotionsstudent an der Goethe-Universität Frankfurt
2011-2012:	Studienreferendar am Kepler-Gymnasium Freudenstadt bzw. Seminar Tübingen
Juli 2012:	Zweites Staatsexamen für den höheren Schuldienst am Gymnasium in den Fächern Mathematik und Informatik mit der Note 1,3 mit Auszeichnung
Seit Sept. 2012:	Studienrat am Privatgymnasium St. Leon-Rot

Stipendien und Studienförderungen

Während meines gesamten Studiums erhielt ich das „Online-Stipendium“ von e-fellows.net. Die Förderung wurde während meiner Promotionszeit weitergeführt.

Nach Vorschlag von Frau Professor Böge aus Heidelberg war ich von März 2005 bis September 2008 Stipendiat der „Studienstiftung des deutschen Volkes“.

Mein Auslandssemester (Wintersemester 2006 / 2007) an der Universität Leeds wurde vom Erasmus-Austausch-Programm und der Studienstiftung unterstützt.