Universität Heidelberg
Hochschule Heilbronn
Medizinische Informatik

Diplomstudiengang Medizinische Informatik

Diplomarbeit

Entwicklung einer visuell interaktiven Segmentierungskomponente unter Verwendung von OpenCL für das Operationsplanungssystem MOPS 3D

Marius Wirths

31.01.2012

Referent: Prof. Dr-Ing. Hartmut Dickhaus
Korreferent: Dr. med., Dipl. Phys., M. Sc. Roland Metzner
Betreuer: Dipl.-Inform. Med. Urs Eisenmann

Zusammenfassung

Das neurochirurgische Operationsplanungssystem MOPS 3D wurde am Institut für Medizinische Biometrie und Informatik der Universität Heidelberg entwickelt. Es unterstützt den Chirurgen mit einer Vielzahl von Methoden bei der Operationsplanung und der Durchführung. Dabei hat sich der Funktionsumfang von MOPS 3D im Laufe der Zeit stetig erweitert. Seit neuem gehört hierzu die Möglichkeit der Volumenvisualisierung, diese wurde mit dem Framework Voreen, welches auf OpenGL basiert, realisiert. Eine weiter Neuerung, die für diese Diplomarbeit von maßgeblicher Bedeutung ist, ist die Entwicklung des OpenCL-Standards. Durch diesen wird es nicht nur möglich Anwendungen hochgradig parallel auf der Grafikkarte auszuführen, es ist des Weiteren möglich mit OpenCL direkt auf Darstellungen, die durch OpenGL generierten wurden, zu operieren. Hierdurch ergeben sich vielversprechende Perspektiven bei der Entwicklung neuer Werkzeuge für das Planungssystem MOPS 3D.

So wird in der nachfolgenden Arbeit eine neue unabhängige Segmentierungskomponente für das Planungssystem MOPS 3D konzipiert und implementiert. Dabei wird auf die Interoperabilität zwischen OpenGL und OpenCL gesetzt, um ein konstantes visuelles Feedback über den Segmentierungsverlauf an den Anwender in Form von 3D-Volumendarstellungen zu gewährleisten.

Zunächst werden Segmentierungsverfahren auf ihre Eignung zur Umsetzung mit OpenCL analysiert. Aufgrund dieser Analyse sind ein Regiongrow- und ein Snakeverfahren für eine GPU-beschleunigte Umsetzung ausgewählt worden. Die Anforderungen an die ausgewählten Verfahren ergeben sich dabei aus einer Analyse der bereits in MOPS 3D bestehenden Segmentierungsverfahren, durch die Schwächen dieser vorhandenen Verfahren aufgedeckt wurden. Anschließend wird die Umsetzbarkeit der ausgewählten Verfahren mit OpenCL prototypisch ermittelt.

Die hieraus gewonnen Erkenntnisse werden genutzt um ein Konzept für die in dieser Arbeit entstandenen Segmentierungskomponente zu entwickeln. Darauf folgend werden die ausgewählten Segmentierungverfahren durch die Implementierung einer unabhängigen Segmentierungskomponente umgesetzt. Anschließend wird die Integration in das Planungssystem MOPS 3D durchgeführt. Dabei benötigte Funktionalitäten vonseiten der integrierenden Anwendung wurden als Teil der Diplomarbeit von Herrn Sascha Diatschuk umgesetzt.

Zwar sind in der OpenCL-Realisierung vonseiten der Hersteller noch kleinere Schwachstellen bzw. Fehler vorhanden, insgesamt erweist sich die Verwendung von OpenCL für die Entwicklung von Segmentierungsverfahren dennoch als Erfolg. So konnte durch eine

Evaluierung der Ergebnisse gezeigt werden, dass eine im hohen Maße performante Segmentierungskomponente entwickelt wurde, die durch ihr visuelles Feedback eine komfortable Nutzung seitens des Anwenders erlaubt. Darüber hinaus ist ein großes Potential an möglichen zukünftigen Erweiterungen gegeben.

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen und mich bei allen Personen bedanken, die mich bei der Durchführung dieser Diplomarbeit unterstützt haben.

Herrn Prof. Dr.-Ing. Hartmut Dickhaus möchte ich dafür danken, dass er mir die Möglichkeit gegeben hat, diese Diplomarbeit durchzuführen.

Herrn Dr. med., Dipl. Phys., M. Sc. Roland Metzner danke ich für die Übernahme des Korreferats. Des Weiteren danke ich ihm für die immer hilfreiche Beratung in medizinischen Fragestellungen sowie für die Bereitstellung verschiedener Bilddatensätze.

Besonderer Dank gilt Herrn Dipl. Inform. Med. Urs Eisenmann für die Betreuung dieser Diplomarbeit. Er stand mir jederzeit bei technischen und inhaltlichen Fragen helfend und fachlich kompetent zur Seite.

Auch möchte ich meinem Kommilitonen Sascha Diatschuk für die gute Zusammenarbeit bei der Verwirklichung unserer beider Diplomarbeiten sowie für die kompetente Beratung, was Musikgeschmack angeht, danken.

Genereller Dank geht an meine Mitbewohner. Sie haben es geschafft, dass ich Heidelberg mein Zuhause nenne.

Ganz besonders danke ich meiner Familie. Sie standen sowohl im Studium als auch während der Durchführung meiner Diplomarbeit immer hinter mir. Meiner Schwester und meinem Bruder möchte ich hier für die vielen schönen Abende danken. Nicht zuletzt möchte ich meinen Eltern für die moralische Unterstützung danken. Zusätzlich danke ich meiner Mutter für die zahllosen langen Abende des Korrekturlesens.

Inhaltsverzeichnis

1 Einleitung

Bildgebende Verfahren haben sich zu einem wichtigen Hilfsmittel der medizinischen Praxis entwickelt. Sie werden in allen medizinischen Bereichen, also sowohl der Diagnostik als auch der Therapie, eingesetzt. Dabei besteht eine Fülle verschiedener Verfahren bzw. Geräten, die verwendet werden, um Bilder für die medizinische Nutzung zu generieren. Beispielhaft seien hier das Röntgen, die Magnetresonanztomographie und die Computertomographie genannt. Dabei hat sich der Stand der Technik derart weiterentwickelt, dass nicht nur einzelne Schichtbilder betrachtet werden können, sondern aus diesen ganze Volumen erzeugt werden können, so dass eine räumliche Darstellung möglich ist. Dies bietet den Vorteil, dass der Anwender sich leicht in vorhandenem Bildmaterial orientieren kann.

Dabei ergibt sich die Problematik der Extraktion von Strukturen, um diese weiter verarbeiten bzw. nutzen zu können. Vor allem in der Neurochirugie ist das Erfassen verschiedener Objekte wie z.B. dem Cortex oder der Ventrikel von besonderem Interesse. Durch die im Hirn vorhandenen hoch sensiblen Strukturen erfahren Werkzeuge bzw. Methoden, die es dem Anwender erlauben sich möglichst präzise zu orientieren, bei der Operationsplanung eine besondere Gewichtung. Neue hierfür notwendige Verfahren werden aus dem stetig wachsenden Forschungsbereich der Bildverarbeitung bereitgestellt.

Insbesondere die Segmentierung, welche einen Teilbereich der Bildverarbeitung ist, spielt dabei eine besondere Rolle. Sie hat die Aufgabe vorhandenes Bildmaterial in disjunkte Regionen einzuteilen. Hierdurch wird es also möglich Objekte in Bildern bzw. Volumen von einander abzugrenzen. Um solche Ergebnisse erzielen zu können werden häufig komplexe Algorithmen verwendet. Dies führt dazu, dass die Ausführung solcher Algorithmen viel Zeit beansprucht. Besondere Probleme entstehen so bei Verfahren der interaktiven Segmentierung. Dabei handelt es sich um Verfahren, die durch stetige Eingaben des Anwenders zusätzliche Informationen über das zu erfassende Objekt erhalten und anhand dieser eine Segmentierung vornehmen. Die Komplexität dieser Verfahren wird jedoch eingeschränkt, da dem Anwender keine zu großen Wartezeiten bei der Berechnung des Segmentierungsergebnisses zugemutet werden können. Bei diesen Verfahren besteht der Vorteil darin, dass es mit ihnen, Dank der Interaktion mit dem Anwender, möglich ist auch komplexe zu segmentieren.

Durch die steigende Performance heutiger Systeme hat die interaktive Segmentierung an Relevanz gewonnen. Vor allem durch die seit kurzem entstandenen Techniken, durch die es möglich wird sich die Rechenleistung von Grafikkarten zu nutze zu machen, ist es möglich geworden selbst komplexe Aufgaben innerhalb kürzester Zeit durchzuführen. Dies steigert im Bezug auf die interaktive Segmentierung die realisierbaren Interaktionsmöglichkeiten

mit dem Anwender erheblich.

1.1 Motivation

Interaktive Segmentierungsverfahren werden auch in dem neurochirurgischen Operationsplanungssystem MOPS 3D eingesetzt. Jedoch sind diese nicht auf dem aktuellen Stand der Technik und lassen daher einige Wünsche in der Anwendung bzw. im Bedienungskomfort offen. So muss für die 3D-Darstellung von Segmentierungsergebnissen eine aufwendige Oberflächentriangulierung durchgeführt werden. Neben der relativ langen Zeit, die benötigt wird diese durchzuführen, gehen bei dieser Art der Darstellung alle Informationen über Strukturen innerhalb des segmentierten Objektes verloren, da lediglich ‚wie der Name schon sagt, eine Oberfläche berechnet wird und so ein hohler Körper entsteht. Auch werden Zwischenergebnisse der Segmentierung nur als 2D-Darstellungen innerhalb der jeweiligen Schichten der verwendeten Bilddaten angezeigt. Dies bereitet Probleme bei der Beurteilung, ob ein momentaner Stand der Segmentierung ein zu segmentierendes Objekt ausreichend und fehlerfrei erfasst. Relativ neu ist in MOPS 3D die Möglichkeit der Volumenvisualisierung. Mit ihr werden tatsächlich ganze Volumen und nicht nur Oberflächen angezeigt. Realisiert wurde die Volumenvisualisierung mit Hilfe des Frameworks Voreen, welches auf OpenGL basiert. Bei OpenGL handelt es sich um eine Bibliothek für graphische Darstellungen.

Dies ist von besonderem Interesse, da es mit OpenCL - einem Standard der das ausführen hochgeradig paralleler Programme erlaubt - möglich ist, direkt auf durch OpenGL generierte Darstellungen zu operieren. So könnte durch den Einsatz von OpenCL bei der Segmentierung direkt auf der Volumenvisualisierung operiert werden, was die bereits angesprochene Oberflächentriangulierung überflüssig macht, da alle Segmentierungergebnisse samt ihrer Zwischenergebnisse in der Volumendarstellung angezeigt werden können. Durch das so generierte konstante visuelle Feedback wird es dem Anwender ermöglicht jederzeit Überblick über den Segmentierungsverlauf zu behalten und gegebenenfalls in diesen einzugreifen. Zusätzlich verspricht die Tatsache, dass die Rechenleistung heutiger Grafikkarten die von handelsüblichen Hauptprozessoren bei weitem übersteigt, einen enormer Performancezuwachs bei der Umsetzung von Segmentierungsverfahren mit OpenCL.

Wie zu sehen ist, stellt sich die Implementierung von Segmentierungsverfahren unter Verwendung von OpenCL als lohnendes Ziel dar. Daher Soll im Verlauf dieser Diplomarbeit eine unabhängige Segmentierungskomponente erstellt und in das Planungssystem MOPS 3D eingebunden werden.

1.2 Zielsetzung

Wie in Kapitel 1.1 erläutert ist der Einsatz von OpenCL für die Umsetzung von interaktiven Segmentierungsverfahren vielversprechend. Gegenstand dieser Diplomarbeit ist daher die Entwicklung einer unabhängigen Segmentierungskomponente. Besonderes Augenmerk

wird hierbei auf die Verwendung von OpenCL gelegt, um sich die Rechenleistung heutiger Grafikkarten zu nutze zu machen. Insbesondere soll dabei die Möglichkeit der Interoperation zwischen OpenCL und OpenGL berücksichtigt werden, um die Darstellung von Volumen durch die zu entwickelnden Segmentierungsverfahren direkt auf der Grafikkarte beeinflussen zu können. Folgend wird das Vorgehen beschrieben, welches ausgewählt wurde, um die eben beschriebenen Ziele zu erreichen.

Als erstes gilt es eine Auswahl über die umzusetzenden Segmentierungsverfahren zu treffen. Diese werden aufgrund ihrer Eignung im Hinblick auf eine Umsetzung mit OpenCL, also im Hinblick auf ihre Parallelisierbarkeit, ausgewählt. Des Weiteren sollen anhand der in MOPS 3D vorhandenen Segmentierungsverfahren neue Anforderungen an die umzusetzenden Verfahren ermittelt und deren Lösung konzipiert werden. Anschließend ist ein Lösungsansatz zu erarbeiten, der es erlaubt, eine anwendungsunabhängige Segmentierungskomponete zu entwickeln.

Ist ein generelles Konzept erstellt, soll mittels der Methodik des Prototyping ermittelt werden, wie sich die ausgewählten Segmentierungsverfahren in Bezug auf die in der Konzeption erarbeiteten Anforderungen und der Verwendung von OpenCL umsetzen lassen. Vor allem soll so nötiges Wissen für den Umgang mit OpenCL erarbeitet werden.

Anschließend wird das Ziel verfolgt, anhand des erarbeiteten Konzepts und der aus dem Prototyping gewonnenen Erkenntnissen, eine unabhängige Segmentierungskomponente zu implementieren und diese in das multimodale Operationsplanungssystem MOPS 3D einzubinden. Darauf folgend werden die erreichten Ergebnisse evaluiert und einer kritischen Betrachtung unterzogen.

2 Grundlagen

2.1 OpenCL

Um die Rechenleistung von Systemen zu erhöhen, wird vermehrt auf Parallelität gesetzt. So besitzen zum Entstehungszeitpunkt dieser Diplomarbeit durchschnittliche CPUs vier Kerne und können somit auch vier Berechnungen gleichzeitig ausführen. Im Bereich der graphischen Darstellung wird dieser Weg schon länger verfolgt und ist dem entsprechend weiter fortgeschritten. So können mit heutigen GPUs weit über 100 Berechnungen gleichzeitig ausgeführt werden. Um sich diese Rechenleistung zu Nutze machen zu können, wurde OpenCL (Open Computing Language) entwickelt ([1], S.1). Hierbei handelt es sich um einen frei verfügbaren Standard zur Erstellung hoch paralleler Programme, bei denen der Entwickler auf viele verschiedene Ressourcen (z.B. CPUs und GPUs) zum Ausführen eines Programmes zugreifen kann. Das besondere hierbei ist, dass jegliche Art von Programm ausgeführt werden kann und nicht wie z.B. bei OpenGL nur Grafik-bezogene Anweisungen durchgeführt werden. Des Weiteren wird durch diesen Standard gewährleistet, dass ein mit OpenCL geschriebenes Programm auf jeglicher OpenCL unterstützender Hardware ohne Änderungen ausgeführt werden kann. Veröffentlicht und weiter entwickelt wird OpenCL durch die Khronos Group.

2.1.1 OpenCL Aufbau

OpenCl setzt sich hauptsächlich aus drei verschiedenen Bestandteilen zusammen. Hierbei handelt es sich um eine Sprachspezifikation, eine Plattform-API und eine Laufzeit-API ([2], S.8).

2.1.1.1 Sprachspezifikation

In der Sprachspezifikation werden die Syntax und das Programmierinterface zum Implementieren von so genannten *Kernel* festgelegt. *Kernel* beinhalten denjenigen Code, der später auf der gewünschten Hardware, also z.B. GPU oder CPU, laufen soll. Die OpenCL-Programmiersprache (*OpenCL C*) basiert auf der ISO C99 Spezifikation mit einigen Erweiterungen und Einschränkungen. Zu den Erweiterungen gehört das Hinzufügen von Vektortypen und Vectoroperationen, verbesserter Zugriff auf Bilder, sowie das Hinzufügen von Adressraumbezeichnern. Zu den Einschränkungen gehört, dass es keine Funktionszeiger gibt, dynamisches Allokieren von Speicher untersagt ist und Rekursion nicht unterstützt wird ([2], S.8).

2.1.1.2 Plattform-API

Durch die Plattform-API erhält der Entwickler Zugriff auf Methoden, um das vorhandene System nach Geräten zu durchsuchen, die OpenCL unterstützen (*OpenCL Devices*). Des Weiteren lässt sich durch diese API das von OpenCL verwendete Konzept von Geräten, *Kontext* und Befehlswarteschlangen (*Command Queue's*) umsetzen (genauere Informationen hierzu sind im Kapitel 2.1.2.1 »Das Plattform-Modell« enthalten). So können OpenCL-Geräte ausgewählt und initalisiert werden. Des Weiteren kann diesen Arbeit zugewiesen und lesend bzw. schreibend auf sie zugegriffen werden. ([2], S.8).

2.1.1.3 Laufzeit-API

Die Laufzeit-API von OpenCL kümmert sich um die Verwaltung von benötigten Ressourcen, sowie der Ausführungsreihenfolge von anstehenden Aufgaben (z.B. das Ausführen von *Kernel* oder von Lese- und Schreibzugriffen) in Befehlswarteschlangen (*Command Queue's*) und ist somit für den Programmablauf zuständig ([2], S.8).

2.1.2 OpenCL Architektur

Der grundlegende Aufbau bzw. die Funktionsweise von OpenCL unterteilt sich in vier verschiedene Modelle. Hierbei handelt es sich um das Plattform-Modell, das Speicher-Modell, das Ausführungs-Modell und das Programmier-Modell ([3], S.21 + [4], S.15-16). Diese werden in den folgenden Kapiteln beschrieben.

2.1.2.1 Das Plattform-Modell

Das Plattform-Modell beschreibt einen *Host*, der auf ein oder mehrere *OpenCL Devices* (bzw. Compute Devices) zugreift. Dies ist in Abbildung 2.1 dargestellt.
Hierbei stellt ein *Host* einen Computer, wie er im täglichen Gebrauch zu finden ist, dar. Ein *OpenCL Device* ist z.B. ein GPU oder ein CPU. Ein solches Device setzt sich wiederum aus mehreren Compute Units zusammen, die wiederum aus mehreren Processing Elements (PEs) bestehen ([3], S.21-22 + [2], S.9). In diesen PEs werden die Berechnungen auf dem *OpenCL Device* durchgeführt. Für den Programmablauf bedeutet dies, dass eine OpenCL-Anwendung auf einem *Host* ausgeführt wird und von diesem dann Befehle an die *OpenCL Devices* bzw. an die PEs gesendet werden. Der genauere Ablauf wird in dem Kapitel 2.1.2.2 beschrieben.

2.1.2.2 Das Ausführungs-Modell

Das Ausführungs-Modell besteht aus zwei Komponenten. Diese sind *Kernel* und *Host-Programme* . *Kernel* sind C-Funktionen sehr ähnlich und stellen die auf einem *OpenCl Device* auszuführenden Einheiten dar. Das *Host-Programm*, welches wie in Kapitel 2.1.2.1 beschrieben auf dem *Host* ausgeführt wird, stellt einen *Kontext* zu den *OpenCL Devices* her und reiht *Kernel* zur Ausführung in Befehlswarteschlangen (*Command Queues*) ein

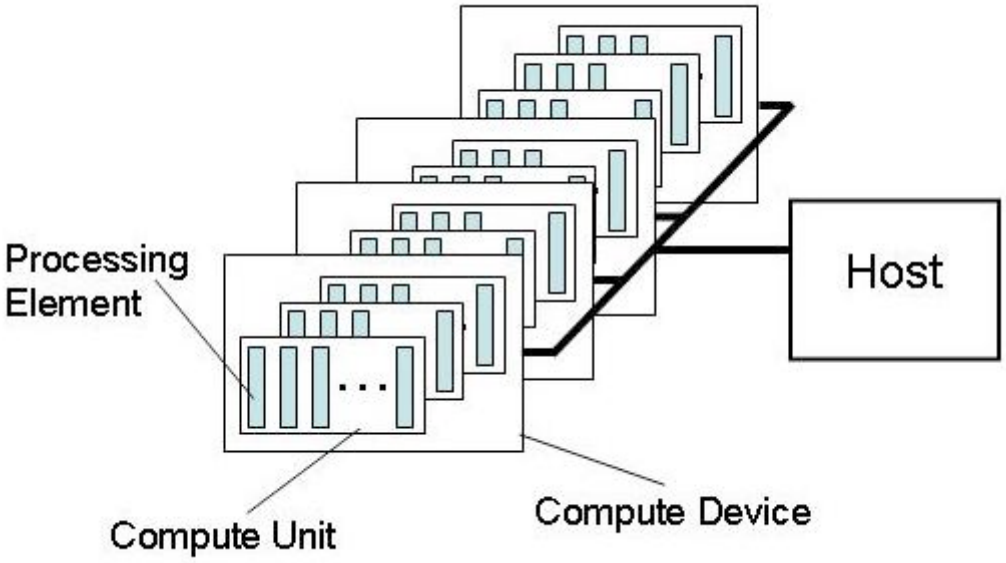

Abbildung 2.1: Das OpenCL Plattform-Modell. Quelle: The OpenCL Specification Version: 1.1; Von: Khronos OpenCL Working Group

([3], S.23).
Um parallele Berechnungen durchzuführen, wird in OpenCL beim Einreihen eines *Kernels* in eine *Command Queue* ein N-dimensionaler *Index-Raum* aufgespannt. Dieser *Index-Raum* wird in OpenCL als NDRange bezeichnet, wobei N die Werte zwischen eins und drei einschließlich annehmen kann (also ein ein- bis dreidimensonaler Raum aufgespannt werden kann). Des Weiteren wird der Wertebereich für jede Dimension festgelegt, die nur ganzzahlige Werte annehmen kann. Für jeden Punkt in diesem Raum wird eine eigene Instanz des *Kernels* ausgeführt. Eine solche Instanz wird in OpenCL als *Work Item* bezeichnet und erhält über die Position im *Index-Raum* eine eindeutige globale ID ([2], S.11). Jedes *Work Item* führt den selben Code aus, wobei der Ausführungsweg durch den Code und die Daten auf denen operiert wird bei den einzelnen *Work Items* unterschiedlich sein kann ([3], S.23). Wenn z.B. durch einen *Kernel* eine Punktoperation realisiert wurde und damit ein 512x512 Pixel großes Bild bearbeitet werden soll, würde es sich anbieten, einen zweidimensionalen Raum aufzuspannen, wobei jede Achse dieses Raumes Werte von 0 bis 511 annimmt, so dass für jeden Bildpunkt ein *Work Item* gestartet wird. Über die globale ID lässt sich dann bestimmen auf welchem Pixel das jeweilige *Work Item* operieren soll. Hierbei würde dann also jedes *Work Item* den gleichen Code in gleicher Weise aber auf unterschiedlichen Daten ausführen.
Des Weiteren werden *Work Items* in Arbeitsgruppen (*Work Groups*) zusammengefasst. Jede *Work Group* hat die gleiche Anzahl an Dimensionen wie die, die beim Einreihen des *Kernels* angegeben wurde. Darüber hinaus besitzt jede *Work Group* eine eigene *Work Group ID* sowie jedes *Work Item* in einer *Work Group* eine eigene lokale ID ([3], S.23 + [2], S.11-12). Der beispielhafte Aufbau eines *Index-Raumes* ist in Abbildung 2.2 veran-

schaulicht. So lässt sich also jedes *Work Item* entweder über seine global ID oder über eine Kombination aus *Work Group ID* und lokaler ID eindeutig identifizieren. Sinn dieser *Work Groups* ist, dass jedes *Work Item* innerhalb einer *Work Group* gleichzeitig auf dem gleichen *OpenCL Device* ausgeführt wird. So können Daten innerhalb einer *Work Group* ausgetauscht werden und außerdem sind Synchronisationsmethoden vorhanden. Eine Möglichkeit zur Synchronisation zwischen verschiedenen *Work Groups* ist nicht gegeben.

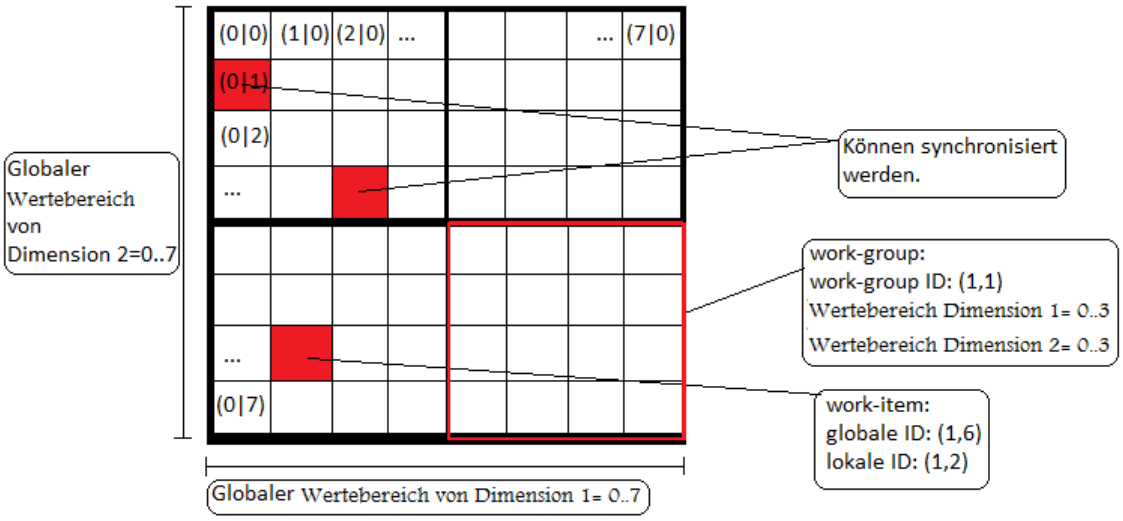

Abbildung 2.2: Aufbau des *Index-Raumes*. Jedes Kästchen steht hierbei für ein *Work Item*. In den Kästchen sind beispielhaft die jeweiligen globalen IDs angegeben.

Wie eingangs in diesem Kapitel erwähnt, stellt das *Host-Programm* einen *Kontext* und mindestens eine *Command Queue* zur Verfügung. Dies sind gleichzeitig die beiden Hauptbestandteile, aus denen sich das *Host-Programm* zusammensetzt. Diese beiden werden im Folgenden genauer erläutert.

Der *Kontext* setzt sich aus 4 Komponenten zusammen ([3], S.25 + [2], S.13). Die erste Komponente ist eine Liste der vom *Host* zu verwendenden *OpenCL Devices*. Zweite Komponente sind die vorhandenen *Kernel*. Die dritte Komponente setzt sich aus mindestens einem Programmobjekt zusammen. Diese Programmobjekte beinhalten den Code, sowie die zugehörigen ausführbaren Dateien, die die einzelnen *Kernel* implementieren. Die letzte Komponente sind Speicherobjekte (memory objects). Diese stellen Objekte im Speicher dar, auf die vom *Host* zugegriffen werden kann und die von den vorhandenen *Kernel* bearbeite werden können. Um den *Kontext* zu erstellen und zu manipulieren, werden von der Plattform-API Methoden zur Verfügung gestellt.

Die *Command Queue* wird ebenfalls mittels Methoden aus der Plattform-API erstellt und dient zur Koordination bei der Ausführung von *Kernel* ([4], S.22-23). Hierbei lassen sich 3 verschiedene Arten von Anweisungen unterscheiden mit denen dies bewerkstelligt wird ([3], S.25). Zunächst sind Anweisungen zum Ausführen von *Kernel* auf einem bestimmten *OpenCL Device* vorhanden. Darüber hinaus werden Anweisungen zum Bearbeiten der

memory objects zur Verfügung gestellt. Mit diesen lassen sich Daten aus memory objects lesen, so wie in momery objects schreiben. Des Weiteren können Daten zwischen memory objects ausgetauscht werden. Die dritte Art von Anweisungen dient zur Synchronisation von Anweisungen. Durch diese kann z.B. das *Host-Programm* dazu gebracht werden, so lange zu warten, bis alle vorhandenen Anweisungen in der commnd queue ausgeführt sind. Die *Command Queue* plant und organisiert die Ausführung aller Anweisungen. Zu beachten ist, dass diese asynchron zwischen *Host* und *OpenCL Device* ablaufen. Hierbei gibt es zwei verschiedene Möglichkeiten wie Anweisungen untereinander ausgeführt werden ([3], S.25 + [2], S.14). Erste Möglichkeit ist, dass sie der Reihe nach abgearbeitet werden, wie sie an die *Command Queue* übergeben wurden, so dass sicher gestellt ist, dass jede Anweisung erst durchgeführt wird wenn die vorangegangene abgeschlossen ist. Diese Verfahren wird in OpenCL In-order Execution genannt. Zweite Möglichkeit ist, dass sie mit dem so genannten Out-of-order Execution Verfahren ausgeführt werden. Dabei werden Anweisungen zwar auch in der Reihenfolge gestartet wie sie an die *Command Queue* übergeben wurden, jedoch wird nicht darauf geachtet, dass Anweisungen abgeschlossen sind, bevor die nächste ausgeführt wird. Bei diesem Verfahren muss der Entwickler selbst auf nötige Synchronisierung achten.

2.1.2.3 Das Speicher-Modell

Das Speicher-Modell beschreibt 4 verschiedene Speicherregionen ([3], S.26). Auf jede von diesen Regionen kann ein *Work Item* zugreifen, während das *Host-Programm* nur auf bestimmte Speicher Zugriff hat.

- Globaler Speicher : Jedes *Work Item* kann auf diesen Speicher lesend und schreibend zugreifen. Hierbei wird jedoch nicht garantiert, dass der verwendete Speicher bei Zugriffen durch die *Work Items* konsistent ist bzw. bleibt. Er muss mittels der Plattform-API vom *Host-Programm* dynamisch allokiert werden. Des Weiteren ist es auch vom *Host* aus möglich, mittels Lese- und Schreibmethoden aus der Plattform-API, auf diesen Speicher zuzugreifen.

- Konstanter Speicher : Dieser Speicher bleibt während der Ausführung eines *Kernels* unverändert. Jedes *Work Item* kann auf diesen Speicher nur lesend zugreifen. Dieser Speicher kann entweder statisch durch den *Kernel* oder dynamisch durch das *Host-Programm* allokiert werden und kann ebenfalls mittels der Plattform-API vom *Host* gelesen und geschrieben werden.

- Lokaler Speicher : Dieser Speicher dient dem Datenaustausch zwischen *Work Items* einer *Work Group* und kann dem entsprechend nur von diesen gelesen und geschrieben werden. Hierbei sind in *OpenCL C* Methoden vorhanden, um zu gewährleisten, dass Zugriffe auf diesen Speicher konsistent sind. Allokiert werden kann er entweder statisch durch den *Kernel* (also mit fester Größe im *Kernel-Code* festgeschrieben)oder dynamisch durch das *Host-Programm* (also zur Laufzeit des Programms mit variabler

Größe), wobei dieses weder lesenden noch schreibenden Zugriff auf diesen Speicher hat.

- Privater Speicher : Jedes *Work Item* besitzt seinen eigenen privaten Speicher und kann als einziger auf diesen zugreifen. Dies ist sowohl lesend als auch schreibend möglich. Er wird ausschließlich vom jeweiligen *Work Item* statisch allokiert.

Ein Überblick, wie die verschiedenen Speicherarten konzeptionell auf einem *OpenCL Device* angeordnet sind, ist in Abbildung 2.3 gegeben. Ergänzend ist noch zu erwähnen, dass globaler Speicher sowohl im Speicher des devices als auch im Arbeitsspeicher des *Hostes* allokiert werden kann (was aber aufgrund der erhöhten Zugriffslatenz zu Performanceeinbußen führt).

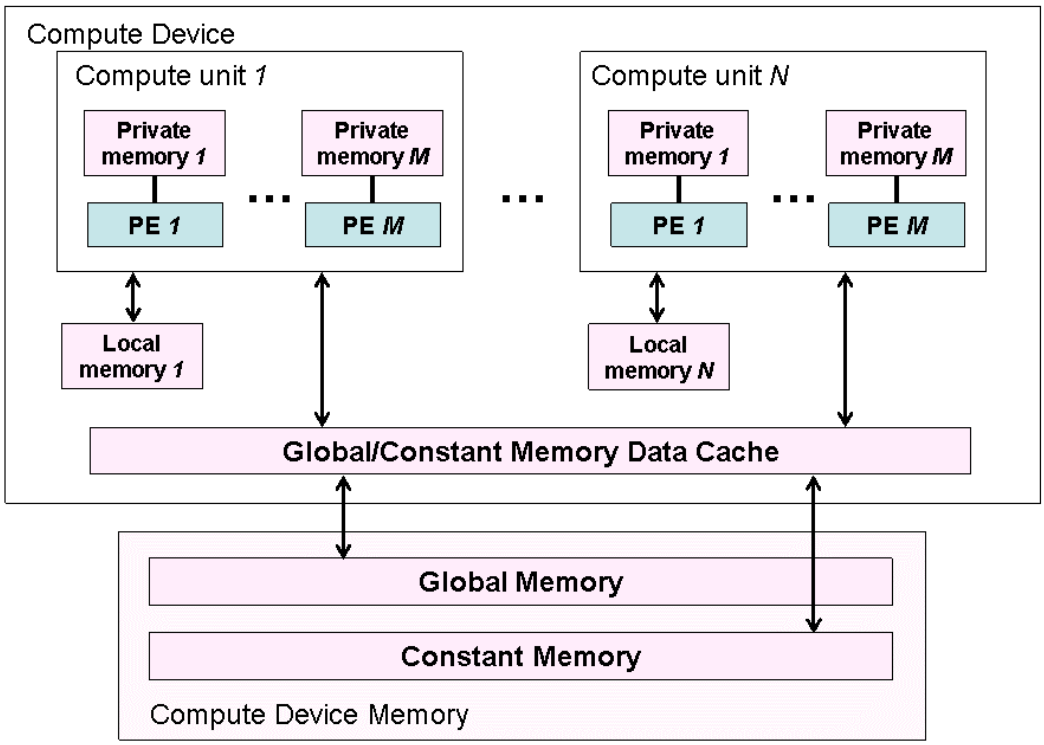

Abbildung 2.3: Konzeptioneller Aufbau eines *OpenCL Devices* mit den verschiedenen compute units, den dazu gehörigen processing elements und den verschiedenen vorhandenen Speicherarten. Quelle: The OpenCL Specification Version: 1.1; Von: Khronos OpenCL Working Group

2.1.2.4 Das Programmier-Modell

OpenCL unterstützt zwei verschiedene Programmiermodelle. Diese sind das datenparallele (data parallel) und das aufgabenparallele (task parallel) Modell ([3], S.28). Des Weiteren werden Mischformen zwischen diesen beiden zugelassen. Ausschlaggebend für das Design

von OpenCL ist jedoch das datenparallele Modell.
Im datenparallelen Modell wird eine Reihe von Anweisungen auf mehrere Elemente eines Speicherobjektes angewandt, so dass, wenn sich strickt an das Modell gehalten wird, für jedes Element des Speicherobjektes die gleichen Anweisungen separat durchgeführt werden. Dieses Verhalten wird, wie in Kapitel 2.1.2.2 beschrieben, durch den aufgespannten *Index-Raum* erreicht. Wobei OpenCL hier das datenparallele Modell etwas lockert, indem nicht für jedes Element ein *Work Item* angelegt werden muss.
Im aufgabenparallelen Modell werden verschiedene Anweisungen parallel ausgeführt. OpenCL bietet hierfür die Möglichkeit *Kernel* unabhängig zum *Index-Raum* und damit unabhängig zu anderen *Kernel* zu starten. Dadurch können verschiedene *Kernel* gleichzeitig gestartet werden.

2.1.3 Speicherobjekte in OpenCL

Wie bereits in Kapitel 2.1.2.2 erwähnt, lassen sich mittel Plattform-API Memory Objects erstellen. OpenCL unterscheidet dabei zwischen *Buffer Objects* und *Image Objects* ([3], S.30). *Buffer Objects* werden zum Halten von eindimensionalen Daten verwendet. Jedes Element eines *Buffer Objects* kann ein skalarer Datentyp wie z.B. int oder float sein. Des Weiteren werden Vektortypen oder auch vom Benutzer spezifizierte Strukturen unterstützt. *Image Objects* hingegen sind dazu ausgelegt, verschiedene Bildformate aufnehmen zu können. Dabei werden sowohl zwei- als auch dreidimensionale Datenstrukturen unterstützt. Generell gilt, dass jedes memory object mindestens ein Element besitzt. Abgesehen davon, wofür die beiden verschiedenen Arten von memory objects gedacht sind, unterscheiden sie sich in zwei grundlegenden Punkten.

- Die Elemente in einem *Buffer Object* werden sequenziell hintereinander im Speicher gehalten, so dass im *Kernel* durch einen Zeiger auf sie zugegriffen werden kann. Elemente in einem *Image Object* hingegen werden in einem Format gehalten, auf das nicht direkt durch den Nutzer bzw. durch den *Kernel* zugegriffen werden kann. *OpenCL C* definiert daher Methoden, um lesend und schreibend auf *Image Objects* zugreifen zu können ([3], S.30).

- In einem *Buffer Object* werden Daten immer im gleichen Format gespeichert wie auf sie durch einen *Kernel* zugegriffen wird. In einem *Image Objects* ist dies nicht der Fall. Dies begründet sich darauf, dass *Image Objects* in einem für die Darstellung optimierten Speicherbereich der Grafikkarte abgelegt werden. So werden einzelne Elemente eines *Image Objects* im *Kernel* immer als Vektor mit 4 Komponenten behandelt (der Datentyp der Komponenten kann z.B. float oder aber auch integer sein). Dies muss nicht das Format sein, in dem die Daten im Speicher gehalten werden. Um dennoch mit *Image Objects* arbeiten zu können, werden in *OpenCL C* Methoden zum Lesen und Schreiben angeboten, die die Daten in das jeweils benötigte Format umformen (als 4 Komponenten-Vektor im *Kernel* und dem internen Format im Speicher) ([3], S.30).

2.1.4 Interoperation zwischen OpenCL und OpenGL

Ähnlich wie bei OpenCL ist OpenGL ein Standard, der durch die Khronos Group Veröffentlicht und Weiterentwickelt wird. Auch er ist ein Standard, der sich die Rechenleistung von GPUs zu nutze macht ([5], S.6). Jedoch wird OpenGL hauptsächlich zur Darstellung von Objekten und Bildern verwendet. So wird OpenGL viel in der Spielindustrie, aber auch z.B. in der Medizin zum Entwickeln von Programmen eingesetzt.
In OpenCL ist die Möglichkeit gegeben auf Objekte aus OpenGL direkt zuzugreifen und diese zu manipulieren. Dadurch kann die durch OpenGL erzeugte Darstellung beeinflusst werden, ohne Daten zwischen Hauptspeicher und Grafikkartenspeicher austauschen zu müssen. Des Weiteren müssen sich von Seiten der OpenCL Entwicklung keinerlei Gedanken gemacht werden, wie die manipulierten Daten dargestellt werden, da dies von OpenGL übernommen wird.

Um dies zu bewerkstelligen muss in OpenCL zunächst ein mit OpenGL geteilter *Kontext* erzeugt werden ([3], S.308). Dieser *Kontext* unterscheidet sich grundsätzlich nicht zu dem in Kapitel 2.1.2.2 beschriebenen *Kontext*. Er erweitert ihn jedoch, sodass durch diesen *Kontext* z.B. aus Objekten von OpenGL (z.B. Texturen) geteilte memory objects in OpenCL erstellt werden können ([3], S.316). Dadurch arbeiten sowohl OpenGL als auch OpenCL auf dem gleichen Speicher, so dass in OpenCl durchgeführte Änderungen sofort in OpenGL vorhanden sind. Um zu verhindern, dass OpenCL und OpenGL gleichzeitig auf ein Objekt zugreifen, werden von der Plattform-API Methoden bereitgestellt, die einen Zugriff auf ein geteiltes memory object einleiten und Methoden, die dieses memory object wieder freigeben. Innerhalb eines solchen Blocks dürfen keine OpenGL Methoden auf dem geteilten Objekt ausgeführt werden. Sollte dies doch geschehen, ist der Zustand des memory objects nicht sicher bestimmbar.

2.2 Verwendete Filter

In der Bildverarbeitung werden Filter für verschiedene Zwecke eingesetzt. Sie eigenen sich z.B. zum Entfernen von Rauschen, dem Glätten von Kanten oder dem Schließen von Lücken im Bild vorhandenen Strukturen ([6], S.269). Genau diese Eigenschaften machen Filter für die Segmentierung attraktiv. So können Daten vor dem Segmentieren aufbereitet oder vorhandene Segmentierungsergebnisse überarbeitet werden. Darüber hinaus gibt es noch weit mehr Anwendungsgebiete sowie Eigenschaften von verschiedenen Filtern, die für diese Diplomarbeit jedoch belanglos sind und nicht weiter behandelt werden. Folgend werden die Filter erläutert, die bei der Umsetzung der Diplomarbeit Einsatz gefunden haben. Des Weiteren wird davon ausgegangen, dass die beschriebenen Filter nur auf Graustufenbilder angewandt werden.

2.2.1 Mittelwertfilter

Der Mittelwertfilter (bzw. Rechteckfilter) gehört zu den linearen Filtern. Das bedeutet, dass ein Zielpixel aus der gewichteten Summe von Quellpixeln berechnet wird. Dies lässt sich auch als Faltung des Bildes mit einer Filterfunktion darstellen. Welche Pixel bei der Berechnung mit einbezogen werden und wie diese gewichtet sind, kann durch eine Matrix dargestellt werden. Diese Matrix wird auch Filtermaske genannt. Welche Werte in der Filtermaske stehen ergibt sich aus der Filterfunktion. Beim Mittelwertfilter bedeutet dies, dass der Mittelwert über den Quellpixel samt seiner benachbarten Werte aus dem Eingangsbild gebildet wird ([6], S.314). Dieser Mittelwert wird dann an die Stelle des Eingangswertes in das Ausgangsbild geschrieben. Wie viele benachbarte Werte in die Berechnung mit eingezogen werden wird über die Filtermaske bestimmt. Veranschaulicht ist dies in Abbildung 2.4.

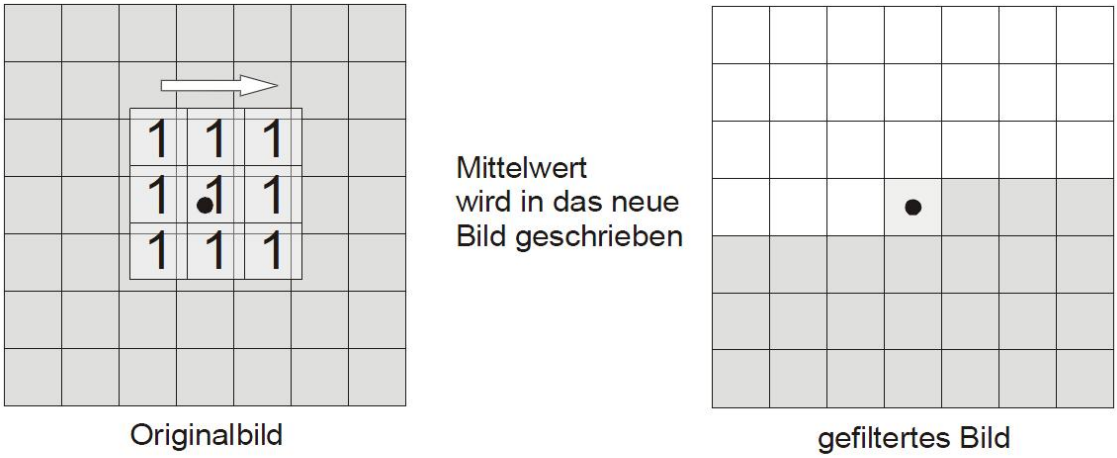

Abbildung 2.4: Mittelwertfilter mit einer 3x3 Filtermaske. Quelle: http://www.techfak.uni-bielefeld.de/~cbauckha/MuVdBData/04-DanielS-Filter.pdf

Der Mittelwertfilter eignet sich zum Entfernen von Rauschen oder dem Entfernen von irrelevanten Strukturen. Nachteilig ist jedoch, dass auch Kanten verwischt werden, da diese sich durch einen hohen Grauwertübergang auszeichnen, der beim Bilden des Mittelwerts verringert wird.

2.2.2 Gauß Filter

Der Gauß Filter gehört genauso wie schon der Mittelwertfilter zu den linearen Filtern. Somit setzt sich auch hier ein Zielpixel aus einer gewichteten Summe von Quellpixeln zusammen. Dabei ergeben sich die Gewichte der Filtermaske im eindimensionalen Fall aus der Gaußfunktion (bzw. der Normalverteilung) wie sie durch Formel 2.1 beschrieben ist

([7], Kapitel 3.1, S.1).

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2.1}$$

Hierbei entspricht σ der Standardabweichung. Theoretisch wird die Gaußfunktion nie 0 und würde zur korrekten Filterung eine unendlich große Filtermaske benötigen. Um dieses Problem zu lösen, wird die Filtermaske in ihren Ausmaßen beschnitten. Hierbei entsteht bei der Anwendung des Filters ein Fehler. Daher muss σ möglichst so gewählt werden, dass der Fehler bei den gewählten Maßen der Filtermaske möglichst klein wird und dennoch ein gutes Filterergebnis entsteht. Um dies zu erreichen kann die Halbwertsbreite benutzt werden. Die Halbwertsbreite einer Funktion ergibt sich aus der Differenz zwischen den beiden Argumentwerten, die die Hälfte des Wertes des Maximums der Funktion besitzen. Veranschaulicht ist dies in Abbildung 2.5. Die Halbwertsbreite m für Formel 2.1 lässt sich

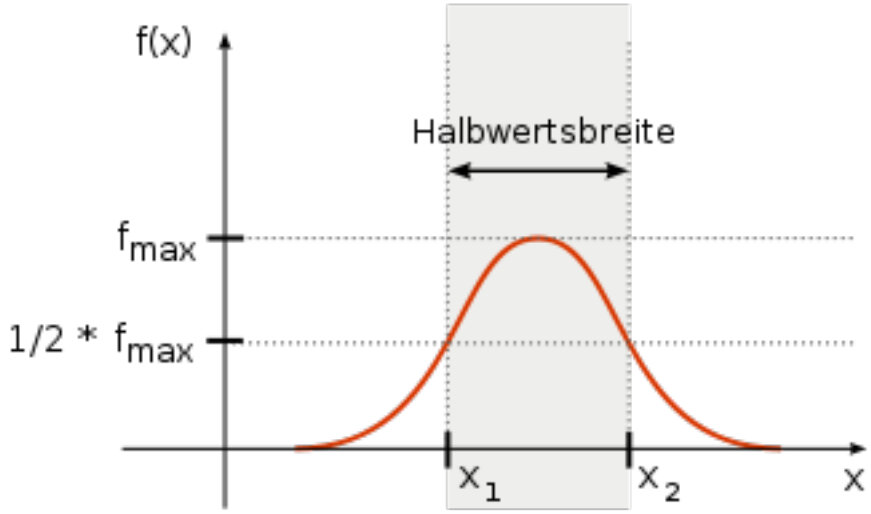

Abbildung 2.5: Berechnung der Halbwertsbreite. Quelle:
http://de.wikipedia.org/wiki/Halbwertsbreite

daher durch Formel 2.2 berechnen ([8]).

$$m = 2\sqrt{2ln2}\sigma \tag{2.2}$$

Wählt man nun eine bestimmte Breite für die Filtermaske, lässt sich durch Umformung der Formel 2.2 ein passendes σ berechnen. Dieses Verfahren lässt sich auch auf den mehrdimensionalen Fall anwenden. Die Werte der entsprechenden Filtermaske im allgemeinen Fall sind durch die Formel 2.3 gegeben; n entspricht dabei der Anzahl der Dimensionen von $\vec{x}$. Wird davon ausgegangen, dass alle Seiten der Filtermaske gleich lang sind ergibt sich daraus, dass das σ für jede Dimension gleich ist und sich somit genauso wie im eindimensionalen Fall berechnen lässt.

$$g(\vec{x}) = \frac{1}{(\sqrt{2\pi}\sigma)^n} e^{-\frac{|\vec{x}|^2}{2\sigma^2}} \tag{2.3}$$

2.2.3 Median Filter

Der Median Filter ist ein Rangordnungsfilter und gehört deshalb zu den nichtlinearen Filtern. Das bedeutet, dass er sich nicht durch eine Faltung beschreiben lässt. Rangordnungsfilter im Allgemeinen sammeln die benachbarten Pixelwerte eines Quellpixels und sortieren diese in eine Liste. Aus dieser Liste wird dann ein Grauwert ausgewählt und an die Stelle des Zielpixels geschrieben. Im Falle des Median Filters wird dabei der Median der sortierten Liste ausgewählt und in den Zielpixel geschrieben ([9], S. 81 + 82). Verwendet wird der Median Filter vor allem um Rauschen zu entfernen. Verglichen mit dem Mittelwertfilter besteht hier der Vorteil, dass Kanten kaum verwischt werden. Dies ist in Abbildung 2.6 dargestellt. Ein Nachteil des Median Filters ist jedoch der hohe Rechenaufwand. Da für

Abbildung 2.6: Anwendung eines Median Filters auf ein stark verrauschtes Bild. Quelle: http://en.wikipedia.org/wiki/File:Medianfilterp.png

jedes Quellpixel dessen Nachbarschaft sortiert werden muss, steigen bei größeren Bildern die nötigen Rechenoperationen stark an. Um den Rechenaufwand zu verringern bietet es sich an, ein passendes Selektionsverfahren zu implementieren.

2.3 Interaktive Segmentierung

In der Bildverarbeitung wird die Segmentierung dazu verwendet ein vorhandenes Bild in disjunkte Regionen einzuteilen, so dass jedes Pixel (oder im dreidimensionalen Fall jedes Voxel) genau einer Region angehört ([10], S.3). In der Bearbeitung von medizinischen Bilddaten spielt dies sowohl bei der Diagnostik als auch bei der Therapie eine besondere Rolle ([11], S.95). So kann z.B. vor Beginn einer Behandlung ein Tumor segmentiert werden, um Aussagen über Form und Volumen machen zu können. Generell können durch die Segmentierung also interessante Objekte wie Gefäße, Gewebe und Tumoren erfasst werden . Um

Pixel einer Region bzw. einem Objekt zuteilen zu können, werden verschiedene Kriterien einbezogen. Hierbei handelt es sich um Kriterien wie z.B. ähnliche Farbwerte oder die Entfernung zwischen den Pixeln, sowie Strukturmerkmale. Dabei lässt sich das Gebiet der Segmentierung in zwei Bereiche unterteilen: der überwachten (supervised) Segmentierung und der unüberwachten (unsupervised) Segmentierung ([10], S.6 + 12). In der unüberwachten Segmentierung werden keinerlei vorherige Annahmen über das zu segmentierende Bild gestellt und es wird versucht, nur durch im Bild enthaltenen Informationen die Pixel in Regionen einzuteilen. Im Gegensatz dazu wird in der überwachten Segmentierung in zwei verschiedenen Phasen segmentiert. In der ersten Phase werden Modelle aller zu erwartenden Regionen anhand einer Datenbank ausgewählt. In der zweiten Phase wird dann jedes Pixel einer passenden Region zugeteilt.
Zwischen diesen beiden Bereichen der Segmentierung bzw. im Übergang dieser Bereiche befindet sich die interaktive - oder auch semi-überwachte Segmentierung. Hier werden zusätzliche Informationen zu einem Bild nicht aus einer Datenbank, sonder durch Benutzereingaben zur Verfügung gestellt ([10], S.14). Durch die Benutzereingaben werden die Objekte markiert, die segmentiert werden sollen. Nach dem dies geschehen ist, werden die Eingaben durch Segmentierungsalgorithmen interpretiert und es wird versucht eine geeignete Segmentierung zu erstellen. Ist diese erstellt hat der Nutzer die Möglichkeit neue Eingaben vorzunehmen, die dann wiederum interpretiert werden, um eine passendere Segmentierung zu erzeugen. Dieser Vorgang kann so oft wiederholt werden bis eine Segmentierung erzeugt wurde, die den Vorstellungen des Nutzers entspricht. Vorteil dieses Verfahrens ist, dass jegliche Art von Bildern segmentiert werden können, ohne dass vorher durch aufwändige Algorithmen Annahmen erstellt werden oder geeignete Modelle in einer Datenbank vorhanden sein müssen. Diese Art der Segmentierung stellt zwei verschiedene Forderungen an die Segmentierungsalgorithmen. Zum einen müssen verwendete Algorithmen möglichst zeitnahe Ergebnisse liefern um eine akzeptable Interaktion mit dem Nutzer zu ermöglichen. Dies hat zur Folge, dass genutzte Algorithmen eine möglichst geringe Rechenkomplexität besitzen. Zum anderen sollte die Anzahl der Benutzereingaben, die nötig sind um eine geeignete Segmentierung zu erzeugen, so gering wie möglich sein. Dies hat wiederum zur Folge, dass die Komplexität und damit die Rechenintensivität der verwendeten Algorithmen steigt je besser Benutzereingaben interpretiert werden können ([10], S.14). Wie zu sehen ist sind dies zwei sich ausschließende Eigenschaften, so dass beim Erstellen einer interaktiven Segmentierung ein geeigneter Mittelweg gefunden werden muss.
In den folgenden Kapiteln 2.3.1 und 2.3.2 werden die beiden interaktiven Segmentierungsverfahren erläutert, die in dieser Diplomarbeit Verwendung finden.

2.3.1 Regiongrow

Beim Regiongrow-Verfahren wird versucht Pixel (bzw. Voxel) in Regionen zusammen zu fassen, so dass die entstandenen Regionen hinsichtlich eines betrachteten Bildmerkmals homogen sind ([10], S.19). Bildmerkmale können hier z.B. Intensitätswerte, Farbwerte oder

auch Texturierung (also Struktureigenschaften) sein. Neben dem Homogenitätskriterium ist eine weitere Bedingung des Regiongrow-Verfahrens, dass alle Pixel einer Region miteinander verbunden sind. Dies bedeutet, dass jeder Pixel einer Region in direkter Nachbarschaft mit den umliegenden Pixeln der Region steht. Vereinfacht ausgedrückt bedeutet dies, dass innerhalb einer Region keine Lücken vorkommen dürfen. Genereller Ablauf des Regiongrow-Verfahrens ist, dass der Benutzer mindestens einen so genannten Saatpunkt (Seedpoint) bestimmt. Dieser Seedpoint stellt die initiale Region dar. Ausgehend von diesem Seedpoint werden die benachbarten Pixel auf ein Homogenitätskriterium überprüft. Wird dieses erfüllt, werden die untersuchten Pixel zur vorhandenen Region hinzugefügt und selbst als Seedpoints behandelt. So werden im nächsten Schritt alle benachbarten Pixel der neuen Seedpoints wiederum auf Homogenität überprüft. Dieser Vorgang wird so oft iteriert, bis keine neuen Seedpoints mehr hinzugefügt werden können ([11], S.100). Dieser Zustand tritt ein, wenn es keinen weiteren mit der Region benachbarten Pixel gibt, der das Homogenitätskriterium erfüllt.
Ausschlaggebend für das Segmentierungsergebnis ist somit zum einen die Wahl der Seedpoints durch den Benutzer, zum anderen das verwendete Homogenitätskriterium und des Weiteren die verwendete Nachbarschaft zwischen den einzelnen Pixeln([11], S.103;[12], S.408; [13], S. 153). Im 2D-Bereich bieten sich entweder die vierer- oder die achter-Nachbarschaft

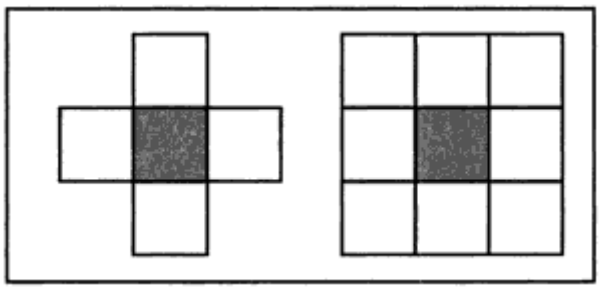

Abbildung 2.7: Vierer- und achter-Nachbarschaft in 2D. Quelle: Medizinische Bildverarbeitung: Bildanalyse, Mustererkennung und Visualisierung für die computergestützte ärztliche Diagnostik und Therapie, S.:101, von: Heinz Handels

an. Bei der vierer-Nachbarschaft zeichnen sich die Menge der benachbarten Pixel N eines Pixels p dadurch aus, dass jeder Pixel aus N eine gemeinsame Kante mit p besitzt. Veranschaulicht ist dies in Abbildung 2.7 links. Bei der achter-Nachbarschaft wird die Menge der Benachbarten Pixel aus N eines Pixels p so beschrieben, dass sie entweder eine gemeinsame Kante oder eine gemeinsame Ecke mit p besitzen. Zu sehen ist dies in Abbildung 2.7 rechts. Im dreidimensionalen Bereich bietet sich die Verwendung der secher- oder der sechsundzwanziger-Nachbarschaft an ([11], S.101 + 102). Bei der sechser-Nachbarschaft werden die benachbarten Voxel N eines Voxels V dadurch bestimmt, dass sie eine gemeinsame Fläche mit V besitzen. Dargestellt ist dies in Abbildung 2.8 rechts. Im Fall der sechsundzwanziger-Nachbarschaft bildet sich die Menge N der benachbarten Voxel von Voxel V so, dass jeder Voxel aus N mindestens eine gemeinsame Ecke, Kante oder Fläche mit

V besitzt. Veranschaulicht ist dies in Abbildung 2.8 links.

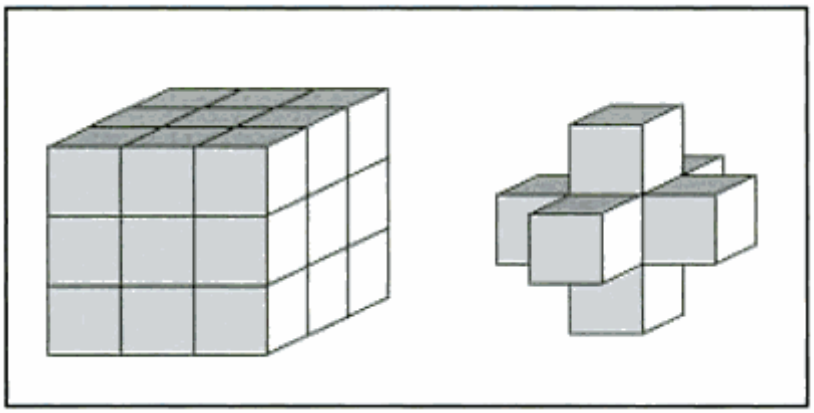

Abbildung 2.8: Secher- und sechsundzwanziger-Nachbarschaft in 3D. Quelle: Medizinische Bildverarbeitung: Bildanalyse, Mustererkennung und Visualisierung für die computergestützte ärztliche Diagnostik und Therapie, S.:102, von: Heinz Handels

Welchen Einfluss die Wahl des Nachbarschaftsverhältnisses hat wird in Abbildung 2.9 verdeutlicht. Der obere Teil der Abbildung stellt den durch den Benutzer definierten seed point dar, wobei jedes Feld einen Pixel repräsentiert. Im unteren Teil werden zwei mögliche Segmentierungsergebnisse dargestellt. Jedes Feld, das eine Zahl beinhaltet gehört zum segmentierten Objekt. Durch den Zahlenwert wird angegeben in welchem Iterationsschritt der jeweilige Pixel der Region hinzugefügt wurde. Im unteren linken Bereich der Abbildung ist das Segmentierungsergebnis unter Verwendung der vierer-Nachbarschaft dargestellt. Im unteren rechten Bereich ist das Ergebnis unter Verwendung der achter-Nachbarschaft abgebildet. Wie zu sehen ist unterscheiden sich die beiden Segmentierungsergebnisse stark. Welche Implementierung des Nachbarschaftverhältnisses besser für die Segmentierung geeignet ist, muss je nach Anwendung entschieden werden. So können durch die achter-Nachbarschaft z.B. feinere Strukturen segmentiert werden, wobei jedoch die Gefahr größer ist dass der Regiongrower ausläuft und Strukturen erfasst, die nicht zum eigentlichen Objekt gehören.

2.3.2 Segmentierung durch Snakes

Snakes stellen Kurven dar, durch die versucht wird Kanten in einem Bild zu erfassen ([11], S.127), um so Objekte einzugrenzen und findet daher auch bei der Segmentierung Einsatz ([14]). Generelles Vorgehen beim Verwenden von Snakes zur Segmentierung von Objekten besteht darin, dass durch den Benutzer eine initiale Kurve vorgegeben wird. Diese Kurve hat maßgeblichen Einfluss auf das Segmentierungsergebnis und sollte möglichst nahe an dem zu segmentierenden Objekt liegen. Ausgehend von dieser initialen Kurve wird versucht die Kurve zu finden, die den Umriss des gesuchten Segments am besten beschreibt. Um diese Kurve zu finden wird jeder Snake eine Energiefunktion zugeordnet. Diese Funktion ist so aufgebaut, dass sie minimal wird, wenn die Snake das Segment am besten beschreibt.

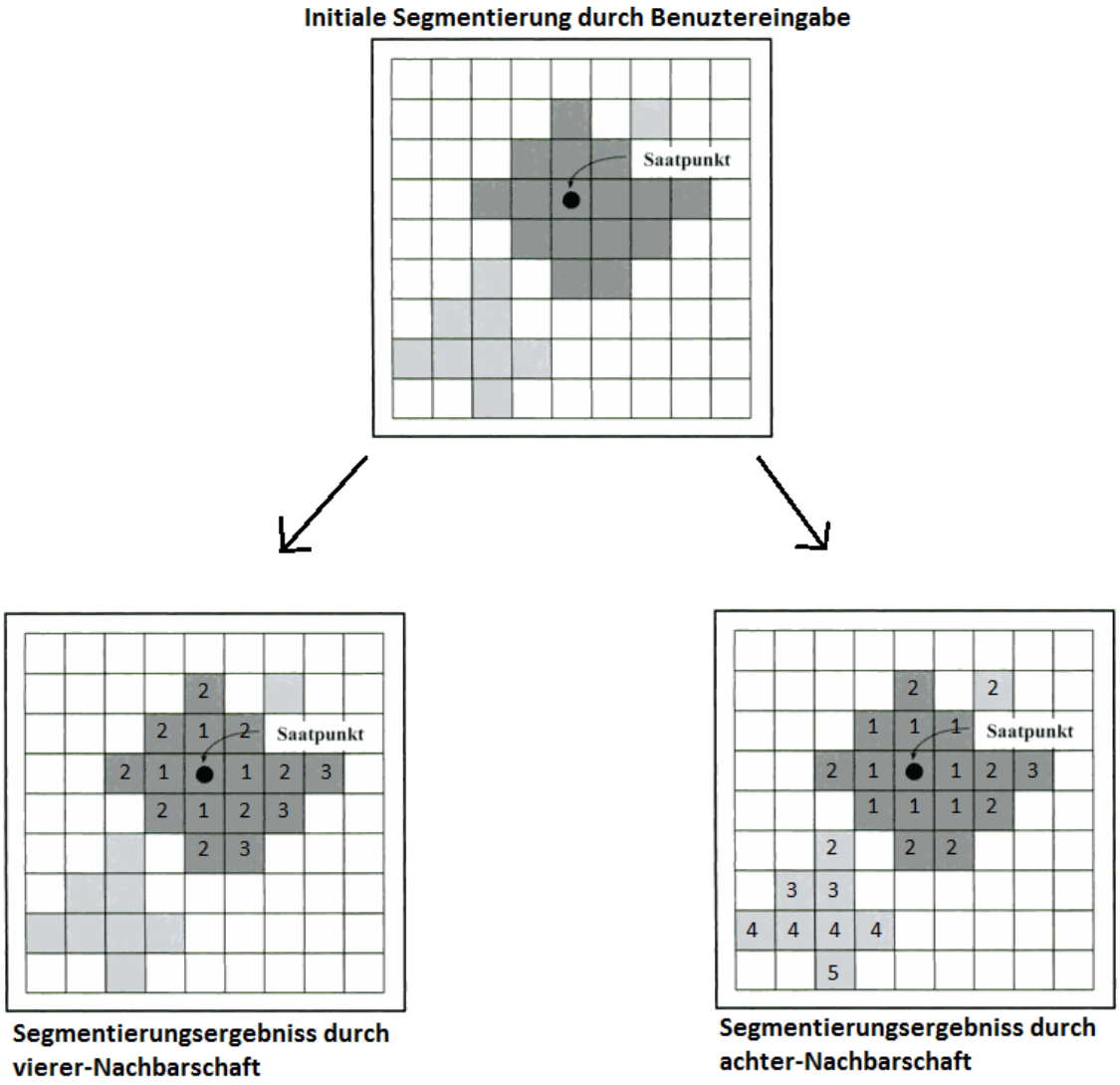

Abbildung 2.9: Segmentierungsergebnisse unter Verwendung einer vierer-Nachbarschaft und unter Verwendung einer achter-Nachbarschaft. Quelle: Medizinische Bildverarbeitung: Bildanalyse, Mustererkennung und Visualisierung für die computergestützte ärztliche Diagnostik und Therapie, S.:105, von: Heinz Handels

Um also durch Snakes zu segmentieren gilt es ein Minimierungsproblem zu lösen ([10], S.20; [11], S.130). Znächst muss aber erst eine passende Energiefunktion gefunden werden. Wenn man die durch die Snake repräsentierte Kurve durch Formel 2.4 beschreibt, lässt sich eine passende Energiefunktion wie in 2.5 darstellen. Wobei $E_{int}(c)$ eine innere Engergie und $E_{ext}(c)$ eine externe Energie beschreiben. Die innere Energie geht allein aus der Form der Kurve hervor und bezieht keinerlei Bildinformationen mit ein. Die externe Energie hingegen setzt sich aus Informationen aus dem Bild zusammen. Genauer beschrieben werden diese beiden Energien in Kapitel 2.3.2.1 und 2.3.2.2.

$$c : [0,1] \rightarrow \mathbb{R} \; mit \; c(s) = \begin{pmatrix} x(s) \\ y(s) \end{pmatrix} \tag{2.4}$$

$$E(c) = E_{int}(c) + E_{ext}(c) \tag{2.5}$$

2.3.2.1 Innere Energie

Die innere Energie wird durch zwei Eigenschaften der Kurve bestimmt. Dies ist zum einen die Länge der Kurve und zum anderen deren Glätte (([11], S.127; [10], S.21). Die Länge der Kurve lässt sich durch das Integral über die erste Ableitung der Kurve c(s) bestimmen. Daher gibt Formel 2.6 ein geeignetes Maß über die Länge wieder und wird größer je länger die Kurve ist.

$$\int_0^1 |c'(s)|^2 ds \tag{2.6}$$

Die Glätte der Kurve lässt sich wiederum durch das Integral über die zweite Ableitung der Kurve c(s) bestimmen. Wiedergegeben ist dies in Formel 2.7. Auch hier gilt je mehr Krümmungen in der Kurve enthalten sind, desto größer wird 2.7.

$$\int_0^1 |c''(s)|^2 ds \tag{2.7}$$

Daher lässt sich die innere Energie an einem Punkt der Snake wie in Formel 2.8 berechnen oder allgemein für die gesammte Snake mit Formel 2.9. Hierbei sind λ_1 und λ_2 reelle positive Konstanten, die zur Gewichtung der einzelnen Energiebestandteile dienen. Hierdurch ist die Möglichkeit gegeben, Vorinformationen über das zu segmentierende Objekt in die Berechnung einfließen zu lassen. Ist z.B. bekannt, dass das Objekt wenige Krümmungen aufweist, bietet es sich an, ein großes λ_2 zu wählen, da hierdurch die Snake dazu veranlasst wird nur wenige Richtungswechsel zuzulassen.

$$E_{int}(x,y) = \frac{1}{2}(\lambda_1 |c'(x,y)|^2 + \lambda_2 |c''(x,y)|^2) \tag{2.8}$$

$$E_{int}(c) = \frac{1}{2} \int_0^1 \lambda_1 |c'(s)|^2 + \lambda_2 |c''(s)|^2 ds \tag{2.9}$$

2.3.2.2 Externe Energie

Setzt man voraus, dass es sich bei dem zu segmentierenden Bild um ein Graustufenbild handelt, so kann dieses durch I(x,y) beschrieben werden. Wobei I den Grauwert an der Stelle (x|y) wiedergibt. Da sich die externe Energie durch Bildinformationen bildet und die Snake sich Kanten annähern soll, sind genau diese von Interesse. Kanten zeichnen sich durch hohe Veränderungen des Grauwertes aus und lassen sich daher durch die erste Ableitung also $\nabla I(x, y)$ berechnen. Daraus ergibt sich für die externe Energie der Snake an einem bestimmten Punkt die Formel 2.10 bzw. für die gesamte Snake die Formel 2.11 ([15], S.360). Das negative Vorzeichen ergibt sich daraus, dass die externe Energie in der Nähe von starken Kanten möglichst klein werden soll. λ_3 ist wie in der inneren Energie eine reelle positive Konstante, um den Einfluss der externen Energie gewichten zu können. Durch diese Gewichtung kann bestimmt werden, wie stark die Snake auf Kanten im Bild reagiert. Sind z.B. in der Nähe des zu segmentierenden Objekts viele schwache Kanten vorhanden, die nicht zum Objekt gehören, bietet es sich an, ein kleines λ_3 zu wählen, damit diese von der Snake ignoriert werden.

$$E_{ext}(x, y) = -\lambda_3 |\nabla I(x, y)| \tag{2.10}$$

$$E_{ext}(c) = -\lambda_3 \int_0^1 |\nabla I(c(s))| ds \tag{2.11}$$

Nachteilig bei dieser Definition der externen Energie ist, dass die Snake nur auf Kanten in unmittelbarer Nähe reagieren kann. Um dies zu verbessern kann Formel 2.11 so erweitert werden, dass das Bild mit einer Gauß-Funktion gefaltet wird ([15], S.360), was bedeutet das Bild mit einem Gauß Filter zu behandeln (siehe Kapitel 2.2.2). Daraus ergibt sich Formel 2.12 wobei G(x,y) einer Gauß-Funktion entspricht und * die Faltung darstellt.

$$E_{ext}(c) = -\lambda_3 \int_0^1 |\nabla [G(c(s)) * I(c(s))]| ds \tag{2.12}$$

Hierdurch werden Kanten verbreitert und die Snake kann somit auch aus größerer Entfernung auf diese reagieren. Jedoch entsteht dabei der Nachteil, dass feinere Bildstrukturen verloren gehen und somit eine exakte Annäherung an das Objekt durch die Snake nicht mehr möglich ist.
Des Weiteren besteht das Problem, dass sich eine Snake mit den beiden vorgestellten Definitionen der externen Energie nur schlecht an konkave Strukturen anpassen kann ([16], S.65). Dargestellt ist dies in Abbildung 2.10. Eine Lösung zu diesen Problemen stellen so genannte Gradient Vector Flow Fields (*GVF-Fields*) dar wie sie von Chenyang Xu und Jerry L. Prince entwickelt wurden. Die genauere Funktionsweise von *GVF-Fields* wird im Kapitel 2.3.2.4 behandelt.

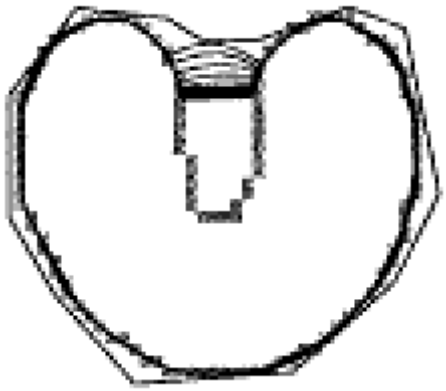

Abbildung 2.10: Problem von konventionellen Snakes bei der Annäherung an konkave Regionen. Quelle: Snake, Shapes, and Gradient Vector Flow; von Cheyang Xu und Jerry L. Prince

2.3.2.3 Minimierung der Snake-Energie

Aus den Kapiteln 2.3.2.1 und 2.3.2.2 erhalten wir für die Energie der Snake die Formeln 2.13 und 2.14. Hierbei beschreibt 2.13 die Energie in einem Punkt der Snake und 2.14 die Energie der gesamten Snake.

$$E(x,y) = \frac{1}{2}(\lambda_1|c'(x,y)|^2 + \lambda_2|c''(x,y)|^2) - \lambda_3|\nabla I(c(x,y))| \tag{2.13}$$

$$E(c) = \frac{1}{2}\int_0^1 \lambda_1|c'(s)|^2 + \lambda_2|c''(s)|^2 ds - \lambda_3 \int_0^1 |\nabla I(c(s))| ds \tag{2.14}$$

Bei dieser Darstellung der Snake-Energie wird davon ausgegangen, dass c(s) als auch I(x,y) stetig differenzierbar sind. Tatsächlich besteht ein Bild und auch eine Snake jedoch aus einer Menge von diskreten Punkten. Daher muss 2.14 in eine diskrete Darstellung umgeformt werden. Es ergibt sich daraus die Formel 2.15. Wobei $\vec{v}_i$ einen Punkt aus der Menge $\{\vec{v}_0, \vec{v}_1, ..., \vec{v}_n\}$ mit $\vec{v}_i = (x_i, y_i)$ darstellt, die die Snake bzw. die Kurve bilden.

$$E(c) \approx \sum_{i=0}^{n} E(\vec{v}_i) \tag{2.15}$$

Dieses Energiefunktional gilt es zu minimieren, wobei darauf zu achten ist, dass nur ein lokales und kein globales Minimum gesucht wird. Ein geeignetes Verfahren um ein lokales Minimum zu finden ist das Gradientenverfahren ([17], S. 126). Hier wird ausgehend von einem Startpunkt iterativ immer entgegen des Gradienten an der Stelle des Punktes gegangen. Im zweidimensionalen Fall bedeutet dies, dass sich ein Punkt (x_{t+1}, y_{t+1}) ausgehend von dem Punkt (x_t, y_t) durch Formel 2.16 und Formel 2.17 berechnen lässt. Wobei f die Gleichung darstellt, deren lokales Minimum gesucht wird. γ ist eine reelle Konstante und

bestimmt die Schrittweite, die entgegen des Gradienten gegangen wird.

$$x_{t+1} = x_t - \gamma \frac{df}{dx}(x_t) \tag{2.16}$$

$$y_{t+1} = y_t - \gamma \frac{df}{dy}(y_t) \tag{2.17}$$

Um die Snake-Energie minimieren zu können muss für das Gradientenverfahren also zunächst eine Ableitung für E(c) bestimmt werden. Da E(c) näherungsweise als Summe dargestellt werden kann und die Ableitung einer Summe gleich der Summe der Ableitungen ist, ergibt sich für die Ableitung von E(c) :

$$\nabla E(c) = \sum_{i=0}^{n} \nabla E(\vec{v}_i) \tag{2.18}$$

Hieraus wird ersichtlich, dass um die gesamte Snake zu minimieren jeder einzelne Punkt der Snake separat mit dem Gradientenverfahren minimiert werden kann. Wird jedes $\vec{v}_i$ als Funktion von t betrachtet folgt daraus:

$$\vec{v}_i(t+1) = \vec{v}_i(t) - \nabla E(\vec{v}_i(t)) \tag{2.19}$$

wobei t den jeweiligen Iterationsschritt darstellt und $\vec{v}_i(0)$ einen Punkt der initial vorhandenen Kurve beschreibt. Wie sich der Gradient der Energie in einem Punkt der Snake genau berechnet, ist in Formel 2.21 wiedergegeben.

$$\nabla E(x,y) = \frac{1}{2}(\lambda_1 \nabla |c'(x,y)|^2 + \lambda_2 \nabla |c''(x,y)|^2) - \lambda_3 \nabla |\nabla I(c(x,y))| \tag{2.20}$$

$$\nabla E(x,y) = \frac{1}{2}(\lambda_1 |c''(x,y)|^2 + \lambda_2 |c''''(x,y)|^2) - \lambda_3 \nabla |\nabla I(c(x,y))| \tag{2.21}$$

Da die externe Energie sich nur aus Informationen aus dem Bild herleitet und dem entsprechend unabhängig von der Kurve ist, muss diese bei der Implementierung für jeden Punkt nur einmal berechnet werden und kann dann in jedem Iterationsschritt wieder verwendet werden. Die innere Energie hingegen hängt von der Form der Kurve ab, die sich mit jedem Iterationsschritt ändert und muss daher für jede Iteration neu berechnet werden.
Im folgenden Kapitel wird eine Erweiterung der externen Energie vorgestellt, die die in Kapitel 2.3.2.2 dargestellten Probleme einer konventionellen Snake behandelt.

2.3.2.4 Gradient Vector Flow

Eine Snake die E(c) minimiert, muss auch die Euler-Gleichung wie sie durch Formel 2.22 gegeben ist erfüllen ([15], S.360).

$$\lambda_1 c''(s) - \lambda_2 c''''(s) - \nabla E_{ext} = 0 \tag{2.22}$$

Diese Euler-Gleichung kann auch als eine Formel für ein Kräftegleichgewicht betrachtet werden.

$$F_{int} + F_{ext} = 0 \tag{2.23}$$

Wobei $F_{int} = \lambda_1 c''(s) - \lambda_2 c''''(s)$ ist und $F_{ext} = -\nabla E_{ext}$ ist. F_{int} ist dabei die Kraft, die die Snake je nach Gewichtung von λ_1 und λ_2 möglichst kurz und kurvenfrei hält. F_{ext} ist diejenige Kraft, die die Snake in die Richtung von Kanten treibt. Dieses F_{ext} hat nach bisher vorgestellter Definition von E_{ext} noch einige Mängel.Veranschaulicht ist dies in Abbildung 2.11. Abschnitt a zeigt eine Snake die sich schrittweise an eine u-förmige Region anpasst bis sie in einem stabilen Zustand ist. Aus den Abschnitten b und c geht hervor warum die Region nur schlecht durch die Snake wiedergegeben werden kann. Hier ist die externe Kraft (F_{ext}) in Form von Vektoren wiedergegeben. Zu sehen ist, dass das F_{ext} nur in unmittelbarer Nähe der Region wirkt (b) und dass eine Snake nicht in konkave Strukturen gezogen werden kann (c).

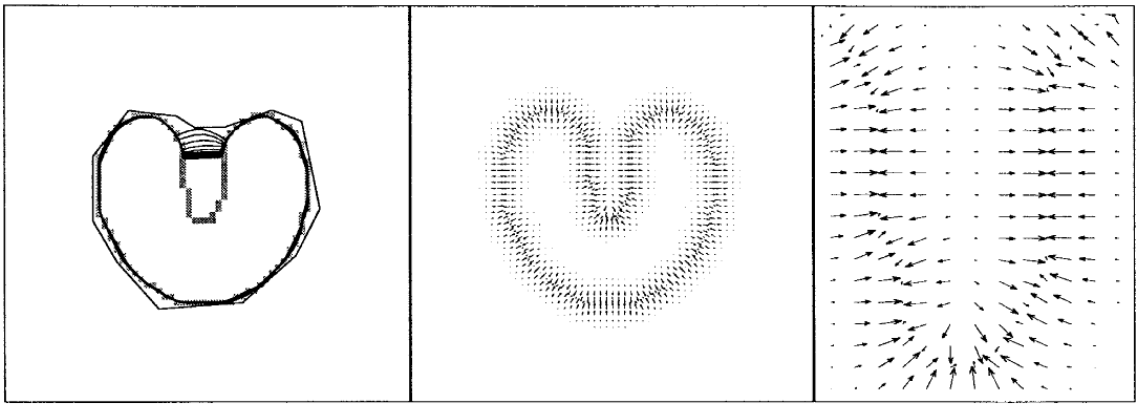

Abbildung 2.11: Problem von konventionellen Snakes bei der Annäherung an konkave Regionen. Quelle: Snake, Shapes, and Gradient Vector Flow; von Cheyang Xu und Jerry L. Prince

Von Chenyang Xu und Jerry L. Prince wurde eine neuere Art von externer Kraft vorgestellt, die die eben erwähnten Probleme behandelt. Ausgehend von der Kräftegleichgewichtsgleichung wird hier F_{ext} durch ein Vektorfeld $\mathrm{v}(x, y)$ gegeben, mit $\mathrm{v}(x, y) = [u(x, y), v(x, y)]$ wobei v das durch Formel 2.24 gegebene Energiefunktional minimiert ([15], S.362). f stellt eine Funktion für das Kantenbild des zu segmentierenden Bildes dar. Chenyang Xu und Jerry L. Prince benennen v als Gradient Vector Flow (GVF) field.

$$E = \int\int \mu(u_x^2 + u_y^2 + v_x^2 + v_y^2) + \nabla f^2 v - \nabla f^2 dxdy \tag{2.24}$$

Zu sehen ist, dass wenn ∇f klein ist die Energie von der Summe der Quadrate der partiellen Ableitungen des Vektorfeldes dominiert wird, was ein gleichmäßiges Erscheinungsbild des Vektorfeldes entfernt von Kanten zur Folge hat. Hingegen bei großem $|\nabla f|$ dominiert der zweite Teil der Gleichung die Energie und wird dadurch minimiert $\mathrm{v} = \nabla f$ zu setzen.

Hierdurch bleiben ursprüngliche Kanten nahezu erhalten mit dem gewollten Nebeneffekt, dass das Feld sich in homogenen Regionen ein wenig verändert. μ ist eine reelle Konstante und dient als Regulator zwischen dem ersten und dem zweiten Term.
Es kann gezeigt werden, dass durch die Lösung der Gleichungen 2.25 und 2.26 ein passendes *GVF-Field* gefunden werden kann ([15], S.363).

$$\mu\nabla^2 u - (u - f_x)(f_x^2 + f_y^2) = 0 \tag{2.25}$$
$$\mu\nabla^2 v - (v - f_y)(f_x^2 + f_y^2) = 0 \tag{2.26}$$

Chenyang Xu und Jerry L. Prince stellen hierfür eine nummerische Lösung vor, bei der sie v und somit auch u und v als Funktion der Zeit behandeln ([18], S.68). Daraus ergibt sich als iterative Lösung die Formeln 2.27 und 2.28

$$\begin{aligned} u(x,y,t+1) = (1 - b(x,y)\triangle t)u(x,y,t) + r(u(x+1,y,t) + u(x,y+1,t) + \\ u(x-1,y,t) + u(x,y-1,t) - 4u(x,y,t)) + c_1(x,y)\triangle t \end{aligned} \tag{2.27}$$

$$\begin{aligned} v(x,y,t+1) = (1 - b(x,y)\triangle t)v(x,y,t) + r(v(x+1,y,t) + v(x,y+1,t) + \\ v(x-1,y,t) + v(x,y-1,t) - 4v(x,y,t)) + c_1(x,y)\triangle t \end{aligned} \tag{2.28}$$

Mit

$$b(x,y) = f_x(x,y)^2 + f_y(x,y)^2$$

$$c_1(x,y) = b(x,y)f_x(x,y)$$

$$c_2(x,y) = b(x,y)f_y(x,y)$$

Und

$$r = \frac{\mu\triangle t}{\triangle x \triangle y}$$

Wobei $\triangle t$ zur Diskretisierung der Zeit dient und $\triangle x$ und $\triangle y$ das Pixelspacing im zu segmentierenden Bild wiedergeben. t gibt den jeweiligen Iterationsschritt an mit $\mathrm{v}(x,y,0) = \nabla f$ als Iterationsstart.

3 Anforderungsanalyse und Konzeption

In diesem Kapitel wird zunächst auf das Multimodale Operationsplanungssystem 3D (MOPS 3D) eingegangen, in das die zu entwickelnde Segmentierungskomponente integriert werden soll. In diesem Zusammenhang werden zunächst bereits vorhandene Segmentierungsmöglichkeiten erläutert, um daraus die Anforderungen der zu verwirklichenden Segmentierungskomponente abzuleiten.

3.1 MOPS 3D

Das Operationsplanungssystem MOPS 3D wurde an der Hochschule Heilbronn und der Universität Heidelberg im Studiengang Medizinische Informatik entwickelt. Es unterstützt den Anwender bei der Operationsplanung durch eine Vielzahl von Anwendungsmöglichkeiten. Im Groben lassen sich diese Anwendungsmöglichkeiten in die Bereiche Visualisierung, Segmentierung und Registrierung einteilen. Von besonderem Interesse für diese Diplomarbeit sind die vorhandenen Segmentierungsmöglichkeiten, welche im Kapitel 3.1.1 genauer erläutert werden. Folgend wird daher nur ein kurzer Überblick über die Visualisierung vorhandener Daten gegeben. So ist es möglich sich verschiedene 2D-Schichten (Coronal-, Transversal- und Sagitalebene) so wie eine 3D Ansicht anzeigen zu lassen ([19]). Wiedergeben sind diese verschieden Anzeigemöglichkeiten beispielhaft in Abbildung 3.1. Neben der Oberflächendarstellung segmentierter Objekte kann der aktuelle Datensatz auch mittels Volumenrendering, also einer tatsächlichen Volumendarstellung, angezeigt werden. Sämtliche Darstellungen erfolgen mittels OpenGL bzw. Voreen. Bei Voreen handelt es sich um eine Open Source Bibliothek zum Darstellen von Volumen, welche in C++ geschrieben wurde und auf OpenGL basiert ([20] Kapitel 2.3).

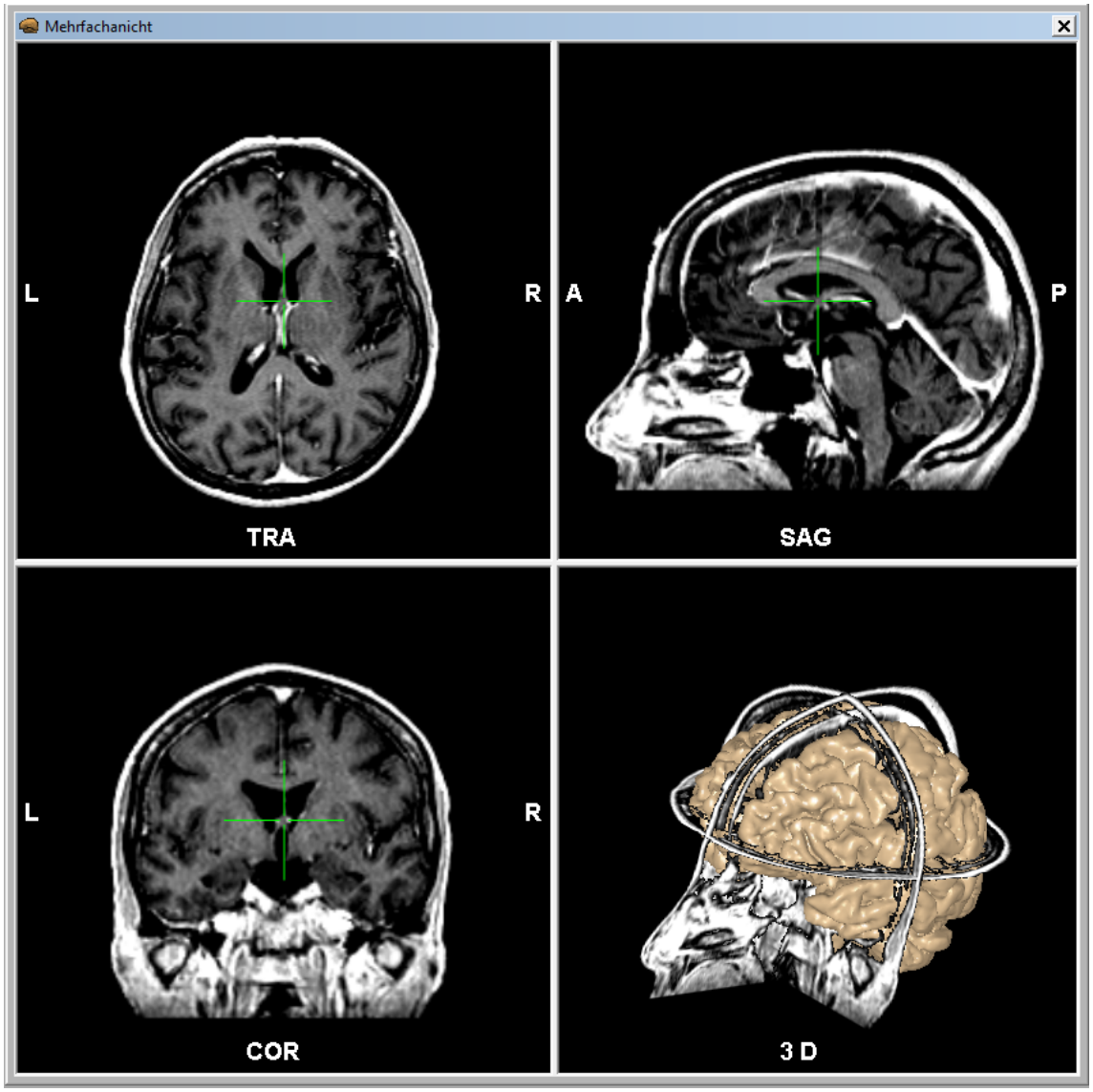

Abbildung 3.1: Oben links: Transversale 2D-Ansicht; Oben rechts: Sagitale 2D-Ansicht; Unten links: Coronale 2D-Ansicht; Unten Rechts: 3D-Ansicht. Zu sehen sind die verschiedenen 2D-Schichten sowie eine Oberflächendarstellung des segmentierten Cortexes.

3.1.1 Vorhandene Segmentierungsmöglichkeiten

Die Segmentierungsverfahren in MOPS 3D bestehen aus drei unterschiedlichen Verfahren:

- Manuelle Segmentierung durch Markieren des Objekts und des Hintergrunds.
- Segmentieren durch ein Histogramm basiertes Schwellwertverfahren.
- Segmentieren mittels eines Regiongrowverfahrens.

Darüber hinaus sind dem Nutzer verschiedene Möglichkeiten zum Nachbearbeiten der Segmentierungsergebnisse gegeben. Hierbei handelt es sich um Operatoren zur Dilatation und Erosion, so wie zum Medianfiltern der Segmentierungsergebnisse. Des Weiteren steht dem Nutzer ein Scrapping-Algorithmus zur Verfügung. Als Benutzerinterface sind die verschiedenen Verfahren auf einem Panel zusammengefasst (siehe Abbildung 3.2). Die beispielhafte

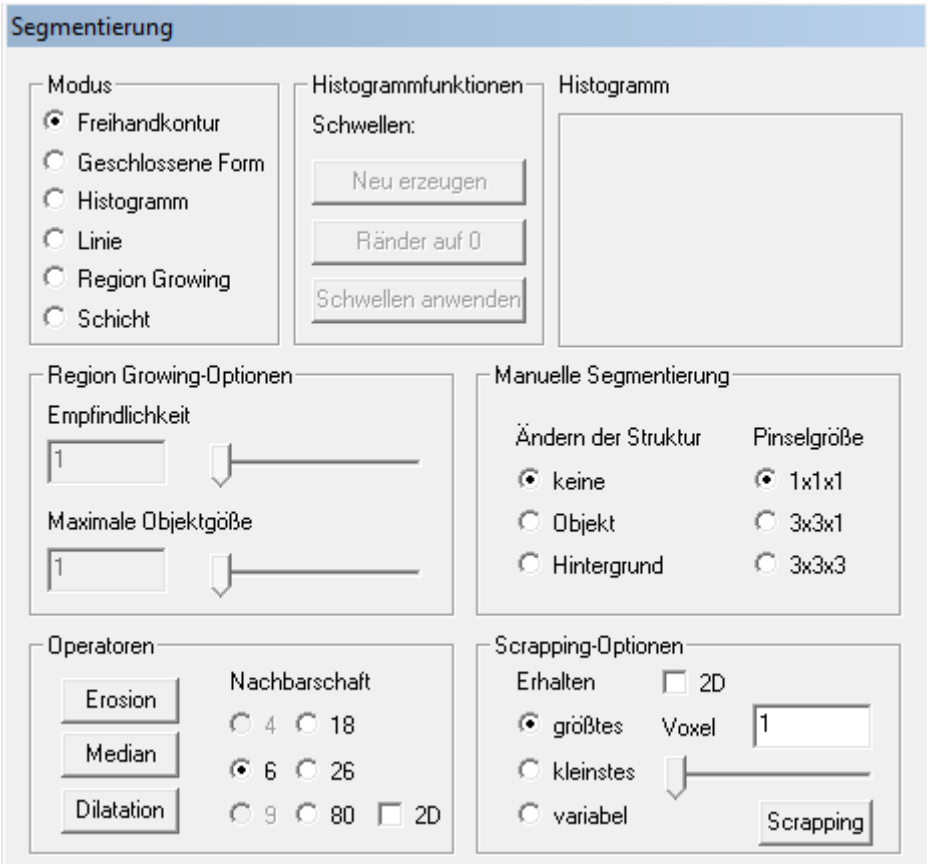

Abbildung 3.2: Ausschnitt des Segmentierungsfensters in MOPS 3D, in dem die vorhandenen Segmentierungsverfahren und die Operatoren zur Nachbearbeitung zusammengefasst sind.

Anwendung des vorhanden Regiongrowverfahrens und des Histogrammverfahrens werden in den folgenden Kapiteln 3.1.1.1 und 3.1.1.2 erläutert, um daraus anschließend die Anforderungen an eine neue Segmentierungskomponente zu erarbeiten.

3.1.1.1 Anwendung des Histogrammverfahrens

In diesem Kapitel wird beispielhaft darauf eingegangen wie mittels des Histogrammverfahrens der Cortex segmentiert werden kann. Natürlich sind noch weit mehr Anwendungsfälle möglich, die hier jedoch nicht behandelt werden, da nur der generelle Segmentierungsablauf dargestellt werden soll. Abbildung 3.3 fasst die Arbeitsschritte zusammen, die nötig sind um eine geeignete Segmentierung zu erzeugen. Zunächst muss die Fensterung der angezeig-

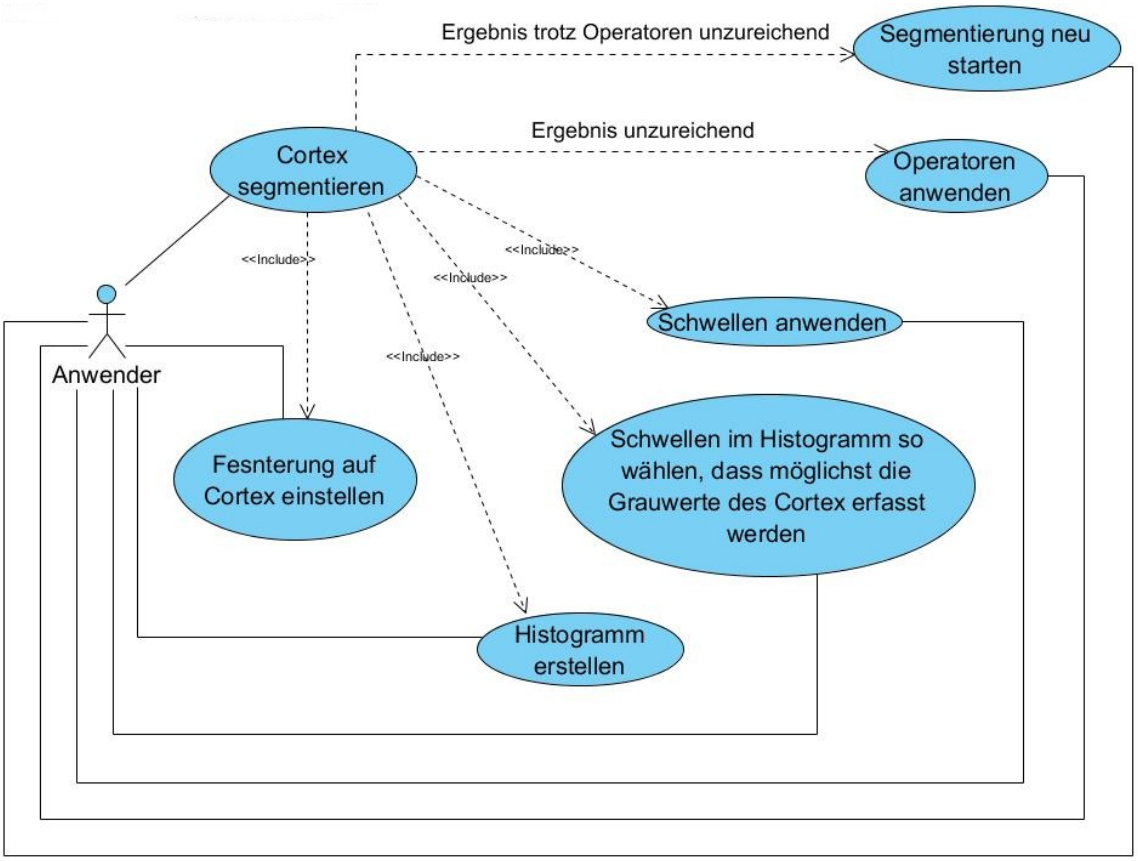

Abbildung 3.3: Anwendungsfalldiagramm für das Histogrammverfahren.

ten Grauwerte so angepasst werden, dass der Cortex möglichst gut sichtbar wird. Danach kann anhand der gefensterten Werte ein Histogramm erzeugt werden. Diese beiden Schritte können so oft wiederholt werden, bis sich die Grauwerte des Cortex im Histogramm gut abzeichnen. Das bedeutet, dass in einem kleinen Grauwertebreich eine große Häufung im Histogramm sichtbar wird. Anschließend muss im Histogramm eine untere und eine obere Schwelle so gewählt werden, dass sie die Grauwerte des Cortex möglichst gut erfassen. Im nächsten Schritt werden dann die Schwellen angewendet. Hierbei werden alle Grauwerte die zwischen den beiden Schwellen liegen zu einem Segmentierungsergebnis zusammengefasst. Ein solches unüberarbeitetes Ergebnis ist in Abbildung 3.4 links zu sehen. In der Abbildung 3.4 rechts ist das Segmentierungsergebnis zu sehen, welches nach Durchführung eines Opening Verfahrens (Durchführung einer Erosion und darauf folgend einer Dilatation) und anschließender Anwendung des Scrapping Algorithmus, mit der Einstellung nur das größte

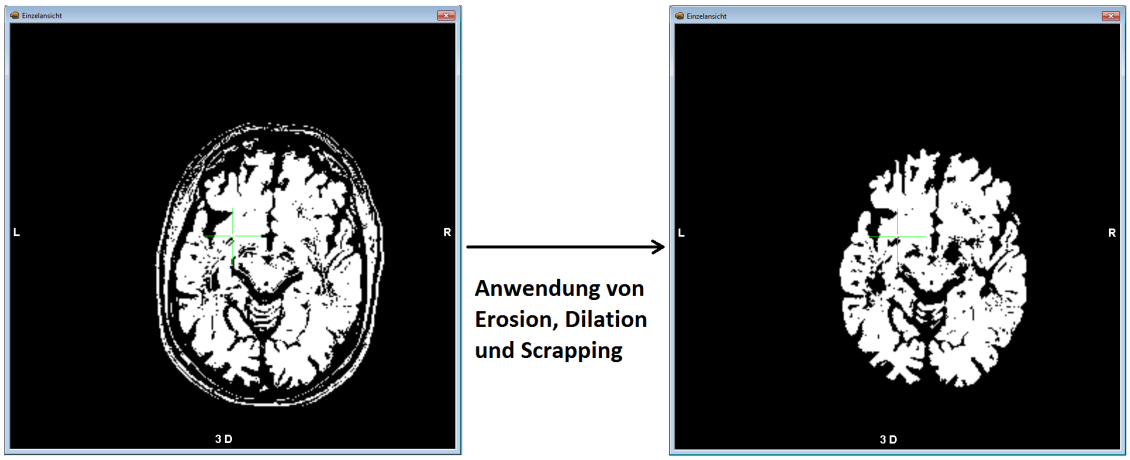

Abbildung 3.4: Segmentierungsergebnis mittels Histogrammverfahrens.
Links: Segmentierungsergebnis ohne Nachbearbeitung;
Rechts: Segmentierungsergebnis mit Nachbearbeitung

Objekt in der Segmentierung zu belassen, entsteht. Um sich das Ergebnis auch in 3D anzeigen lassen zu können, muss als letzter Schritt eine Oberflächentriangulierung durchgeführt werden. Erst hiernach ist es möglich sich einen Gesamteindruck über die Segmentierung zu verschaffen. Die Triangulierung der in Abbildung 3.4 dargestellten Segmentierung ist in Abbildung 3.5 wiedergegeben. Dies beinhaltet den Nachteil, dass Fehler im Segmentierungsergebnis, die in der zweidimensionalen Darstellung nicht aufgefallen sind, jedoch durch die 3D-Darstellung erkannt werden, nicht mehr bearbeitet werden können, da es nicht möglich ist Zwischenergebnisse zu verändern, um ein besseres Endergebnis zu erzielen. Die einzige Möglichkeit einen solchen Fehler zu beheben, ist die Segmentierung neu zu starten und das bisherige Ergebnis zu verwerfen.

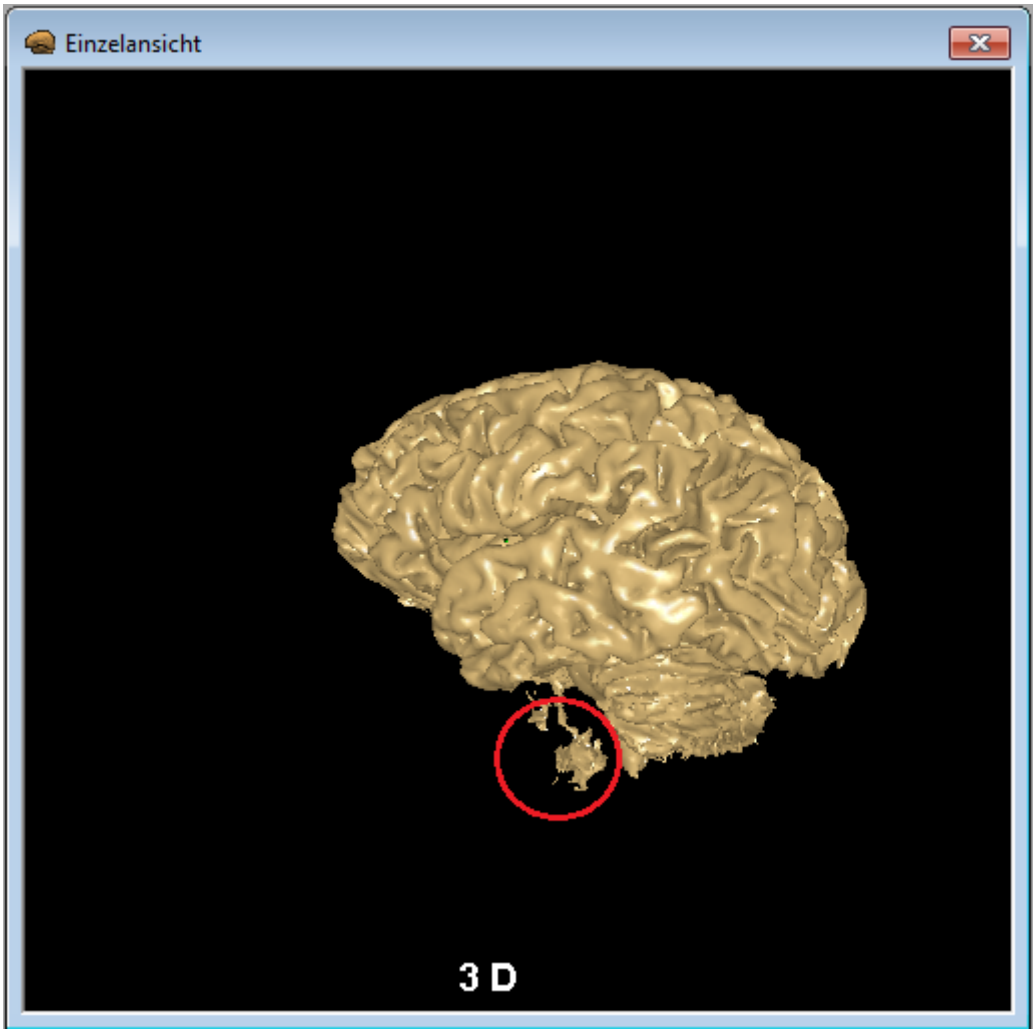

Abbildung 3.5: Trianguliertes Ergebnis der Histogrammsegmentierung. Rot markiert ist ein Teil des Segmentierungsergebnisses, welches nicht zum Cortex gehört, jedoch durch die Segmentierung mit erfasst wurde.

3.1.1.2 Anwendung des vorhandenen Regiongrowverfahrens

Ähnlich wie in Kapitel 3.1.1.1 wird auch hier ein Anwendungsbeispiel für das vorhandene Regiongrowverfahren dargestellt. Es geht dabei nicht um die Vollständigkeit aller Anwendungsmöglichkeiten, sondern um die Veranschaulichung des generellen Vorgehens bei der Segmentierung mittels Regiongrow. Als Anwendungsbeispiel werden die lateralen Ventrikel eines Tumorpatientens mittels Regiongrow segmentiert. Die nötigen Arbeitsschritte für die Segmentierung sind in Abbildung 3.6 zusammengefasst. Die ersten Schritte, um mit

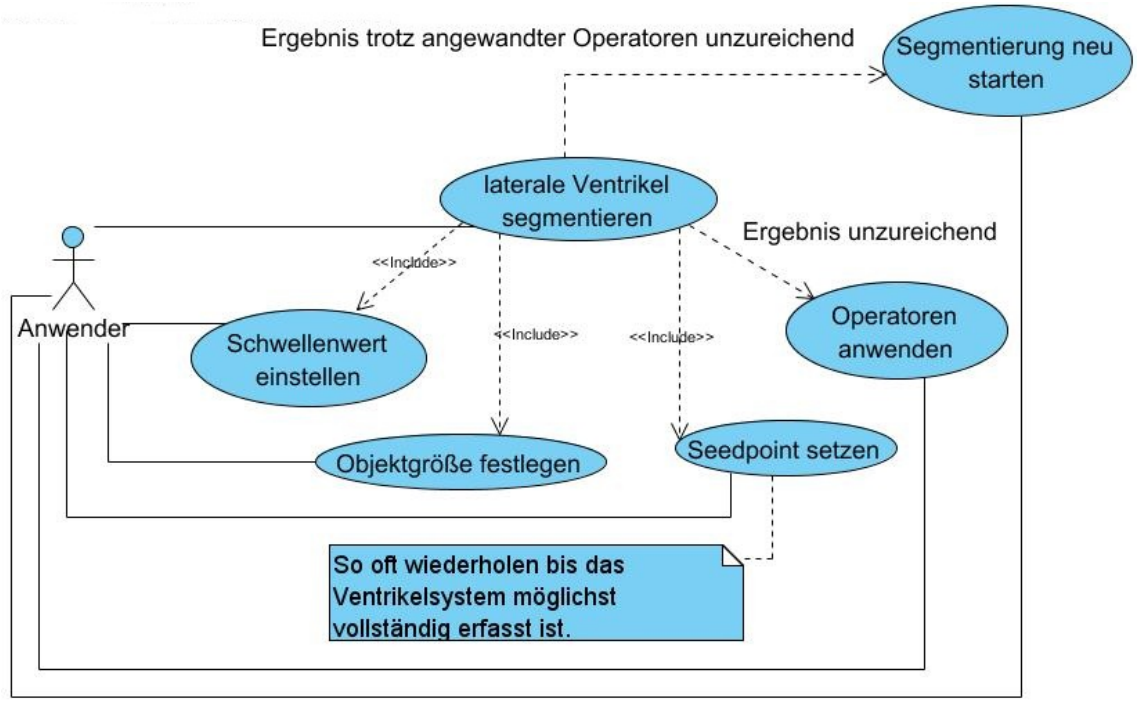

Abbildung 3.6: Anwendungsfalldiagramm für das vorhandene Regiongrowverfahren.

dem Regiongrower zu arbeiten bestehen darin, einen Schwellwert und eine maximale Objektgröße festzulegen. Der Schwellwert gibt an, in welchem Bereich um die Grauwerte der gewählten Seedpoints benachbarte Pixel bzw. Voxel zum Segmentierungsergebnis hinzugefügt werden. Hierüber wird also das Homogenitätskriterium bestimmt. Da die lateralen Ventrikel einen sehr einheitlichen Grauwertebereich besitzen, bietet es sich an, einen geringen Schwellwert einzustellen. Über die maximale Objektgröße wird ein relatives Maß dafür angegeben wie groß das zu segmentierende Objekt maximal sein kann. Da das Ventrikelsystem im Vergleich zum gesamten Schädel relativ klein ist, bietet es sich auch hier an einen kleinen Wert für die maximale Objektgröße zu wählen. Anschließend beginnt die eigentliche Segmentierung durch das Hinzufügen von *Seedpoints*. Hierbei wird nach jedem Hinzufügen eines Seedpoints die zugehörige Region berechnet und danach angezeigt. Wiedergegeben ist dies in Abbildung 3.7 (a), (b) und (c). Nach dem das Ventrikelsystem durch die Seedpoints möglichst gut erfasst ist, kann das Segmentierungsergebnis mittels der in Kapitel 3.1.1 genannten Operatoren nachbearbeitet werden. Im Falle der Segmentierung, wie sie in Abbildung 3.7 (c) wiedergegeben ist, bietet es sich an, eine Dilatation anzuwenden, da

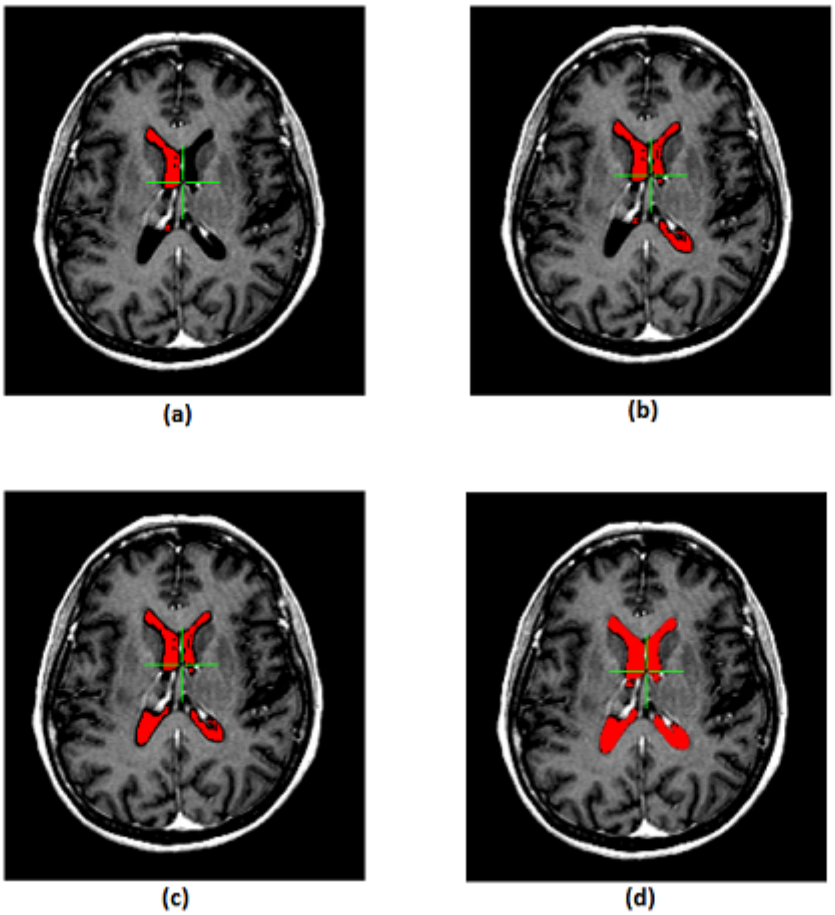

Abbildung 3.7: Ventrikelsegmentierung mittels Regiongrowverfahrens.
(a) Segmentierungsergebnis nach Bestimmung eines Seedpoints.
(b) Segmentierungsergebnis nach Bestimmung eines 2. Seedpoints. (c) Unüberarbeitetes Segmentierungsergebnis nach Bestimmung weiterer Seedpoints.
(d) Segmentierungsergebnis nach angewandter Dilatation.

zu erkennen ist, dass das bisherige Segmentierungsergebnis im Hinblick auf das Segmentierungsziel gleichmäßig zu klein ist (dunkler Rand um die bisher erfasste Segmentierung). Hierdurch wird es möglich ein Ergebnis wie es in Abbildung 3.7 (d) wiedergegeben ist zu erreichen. Gleich wie bei dem Histogrammverfahren in Kapitel 3.1.1.1 muss das endgültige Segmentierungsergebnis trianguliert werden um eine 3D-Obeflächendarstellung zu erzeugen (wiedergegeben in Abbildung 3.8). Auch hier bedeutet dies, dass die Segmentierung abgeschlossen ist. Sollten zu diesem Zeitpunkt noch Fehler in der Segmentierung auffallen, die in der 2D-Darstellung übersehen wurden, ist die einzige Möglichkeit, um eine fehlerfreie Segmentierung zu erhalten, das vorhandene Ergebnis zu verwerfen und die Segmentierung von Neuem zu beginnen.

Wie in diesem Kapitel und in Kapitel 3.1.1.1 zu sehen ist, besteht ein maßgeblicher Mangel bei den vorhandenen Segmentierungsverfahren in den fehlenden Eingriffsmöglichkeiten für den Anwender. Dies basiert unter anderem darauf, dass der Anwender den Ablauf der

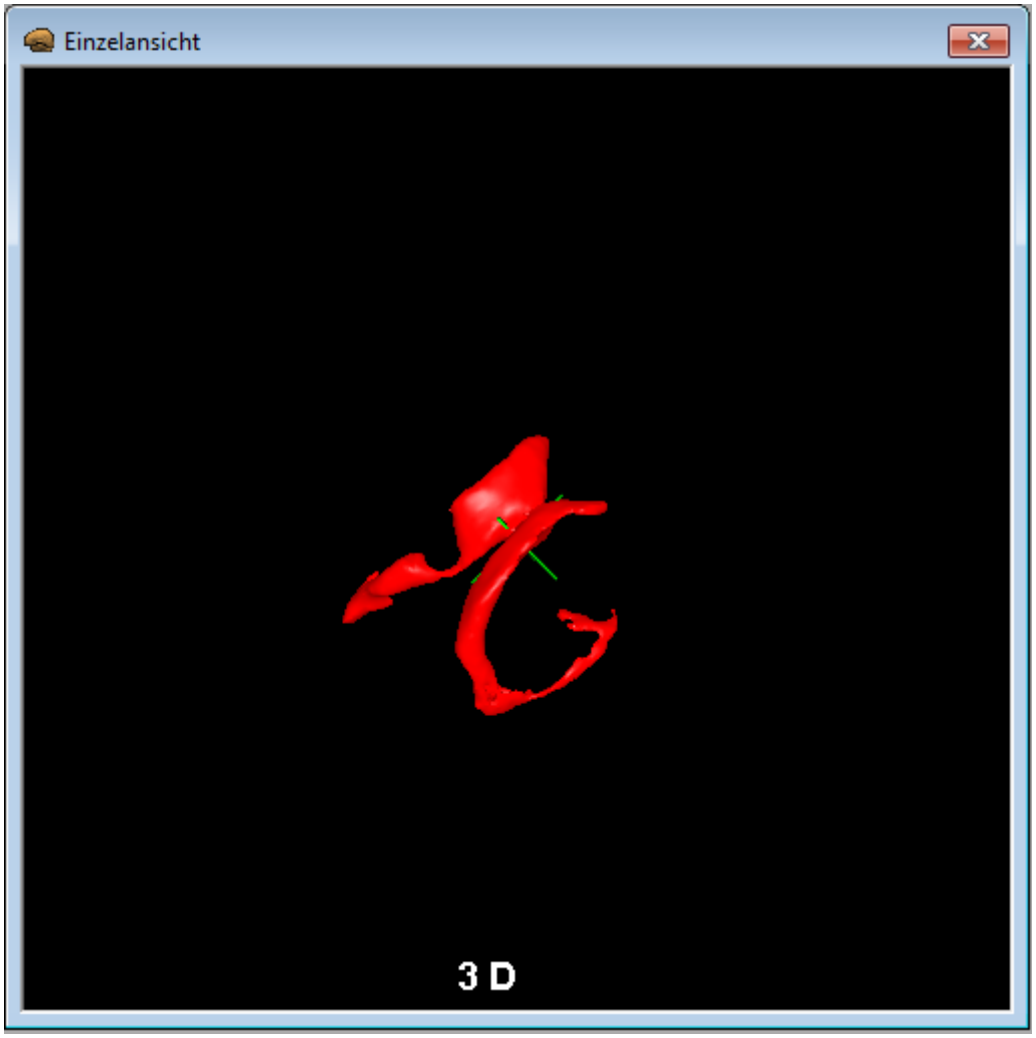

Abbildung 3.8: Triangulierte Oberflächendarstellung der durch das Regiongrowverfahren segmentierten lateralen Ventrikel.

Segmentierung nicht optisch nachvollziehen kann, sondern immer erst nach Abschluss eines Segmentierungsalgorithmus' das resultierende Ergebnis angezeigt bekommt. Auch ist die zweidimensionale Darstellung der Segmentierungsergebnisse nur bedingt ausreichend um sich einen Gesamteindruck über den momentanen Segmentierungsstand verschaffen zu können.

3.2 Anforderungen an die neue Segmentierungskomponente

In diesem Kapitel wird ein Überblick über die grundsätzlichen Anforderungen der neu zu entwickelnden Segmentierungskomponente gegeben. Diese lassen sich grob in 4 Punkten zusammenfassen:

- Entwicklung einer vom MOPS 3D unabhängigen Segmentierungskomponente.
- Gewährleistung eines konstanten graphischen Feedbacks an den Nutzer während der

Segmentierung.

- Gewährleistung, dass der Nutzer jederzeit auf die momentane Segmentierung Einfluss nehmen kann.
- Aufbereitung vorhandener Daten mittels Filter vor der Segmentierung, sowie Nachbearbeitung vorhandener Segmentierungsergebnisse.

Nachfolgend werden diese einzelnen Punkte genauer erläutert, um dann in Kapitel 3.3 einen Entwurf für die Anforderungen zu präsentieren.

3.2.1 Entwicklung einer unabhängigen Segmentierungskomponente

Um eine hohe Wiederverwendbarkeit zu erreichen, ist ein Ziel dieser Diplomarbeit die entwickelten Verfahren in eine unabhängige Komponente zusammenzufassen. Hierdurch wird es möglich die gewonnenen Erkenntnisse nicht nur in MOPS 3D, sondern auch in anderen Anwendungen, die sich mit der Volumendarstellung von Bilddaten beschäftigen zu verwenden. Um dies zu ermöglichen ist es nötig Schnittstellen bereitzustellen, die eine möglichst einfache Einbindung der zu entwickelnden Segmentierungskomponente in bestehende Anwendungen erlauben.

3.2.2 Konstantes Feedback an den Nutzer

Wie in Kapitel 3.1.1.1 und Kapitel .3.1.1.2 zu sehen ist besteht ein hauptsächliches Problem der vorhandenen Segmentierungsverfahren darin, dass der Nutzer sich erst nach Abschluss der Segmentierung einen Gesamteindruck über das Segmentierungsergebnis verschaffen kann. Hieraus leitet sich eine grundlegende Anforderung an die neu zu entwickelnde Segmentierungskomponete ab.
Es soll dem Nutzer möglich sein, sich jeder Zeit einen umfassenden Überblick über die momentane Segmentierung verschaffen zu können. Um dies zu gewährleisten muss eine konstante graphische Rückmeldung (im Folgenden Feedback genannt) an den Nutzer erfolgen. Da durch eine 2D-Visualisierung immer nur ein Teil des momentanen Segmentierungsstandes dargestellt werden kann, muss das Feedback in Form einer 3D-Darstellung erfolgen. Eine zusätzliche Anforderung an das zu gebende Feedback ist, dass es so performant erfolgt, dass ein flüssiger Programmablauf gewährleistet bleibt, um dem Nutzer unzumutbar lange Wartezeiten zu ersparen.

3.2.3 Jederzeitiger Eingriff in die laufende Segmentierung

Um aus dem in Kapitel 3.2.2 beschriebenen Feedback einen praktischen Nutzen ziehen zu können, soll es dem Nutzer möglich sein, jeder Zeit in die laufende Segmentierung eingreifen zu können. So kann der Nutzer durch das vorhandene Feedback entstehende Fehler frühzeitig erkennen und soll dann in der Lage sein, diese durch Anpassung entsprechender Segmentierungsparamenter zu vermeiden, bzw. zu beseitigen.

3.2.4 Bereitstellung verschiedener Filter

Wie in Kapitel 2.2 beschrieben eignen sich Filter zum Entfernen von Rauschen, dem Glätten von Kanten oder dem Schließen von Lücken. Ziel ist es, sich genau diese Eigenschaften auch in der zu implementierenden Komponente zunutze zu machen. Hier gibt es zwei verschiedene sinnvolle Anwendungsmöglichkeiten:

- Filtern der Bilddaten vor der Segmentierung.
- Filtern des vorhandenen Segmentierungsergebnisses.

Filtern der Bilddaten vor der Segmentierung macht vor allem Sinn um störendes Rauschen zu unterdrücken, was sonst leicht zu unerwünschten Segmentierungsergebnissen führen kann. Auch ist es durchaus sinnvoll Lücken in vorhandenen Strukturen mittels Filterung vor der Segmentierung zu schließen. Das Filtern der Segmentierungsergebnisse dient zur Verbesserung der graphischen Darstellung. Hierdurch werden gleichmäßigere Grauwertübergänge generiert, was zu weicheren Kanten führt und somit eine harmonischere Darstellung erzeugt.

3.3 Entwurf

In diesem Kapitel wird für die in Kapitel 3.2 dargestellten Anforderungen ein Entwurf präsentiert. Dieser dient als Grundlage der in Kapitel 4 dargestellten Implementierung. Zunächst wird auf die ausgewählten Segmentierungsverfahren im Hinblick auf die Anwendung und den Aufbau dieser eingegangen. Anschließend wird ein Konzept für die zu realisierenden Filter vorgestellt. Zum Schluss wird dargestellt wie diese Komponenten in eine übergeordnete unabhängige Komponente zusammengefasst werden.

3.3.1 Die Segmentierungsverfahren

Folgend werden die Entwürfe der beiden Segmentierungsverfahren vorgestellt, die für eine GPU-Realisierung ausgewählt wurden. Zum einen wurde das vorhandene Regiongrowverfahren aufgegriffen und anhand der erarbeiteten Anforderungen neu konzipiert. Des Weiteren wurde ein Snakeverfahren als zusätzliches Segmentierungsverfahren ausgewählt. Diese beiden Verfahren eignen sich auf Grund ihrer hohen Parallelisierbarkeit besonders für eine Umsetzung in OpenCL. Des Weiteren kann durch das wieder aufgegriffene Regiongrowverfahren ein direkter Vergleich zwischen dem bereits vorhandenen Verfahren und der GPU-beschleunigten Lösung gezogen werden. Mit der Auswahl des Snakeverfahrens wird es außerdem möglich zu demonstrieren, dass auch komplexere Verfahren mit Hilfe von OpenCL umgesetzt werden können.

3.3.1.1 Regiongrow

Um generelle Unterschiede zwischen dem neuen GPU-basierten Regiongrowverfahren und dem bereits vorhandenen Regiongrowverfahren bei der Nutzung zu verdeutlichen, wird auf

das gleiche Anwendungsbeispiel wie in Kapitel 3.1.1.2 eingegangen. Die Anwendungsschritte, die durch das neue Verfahren zur Verfügung gestellt werden sollen, sind in Abbildung 3.9 wiedergegeben. Ähnlich wie beim vorhandenen Regiongrowverfahren müssen auch im neu-

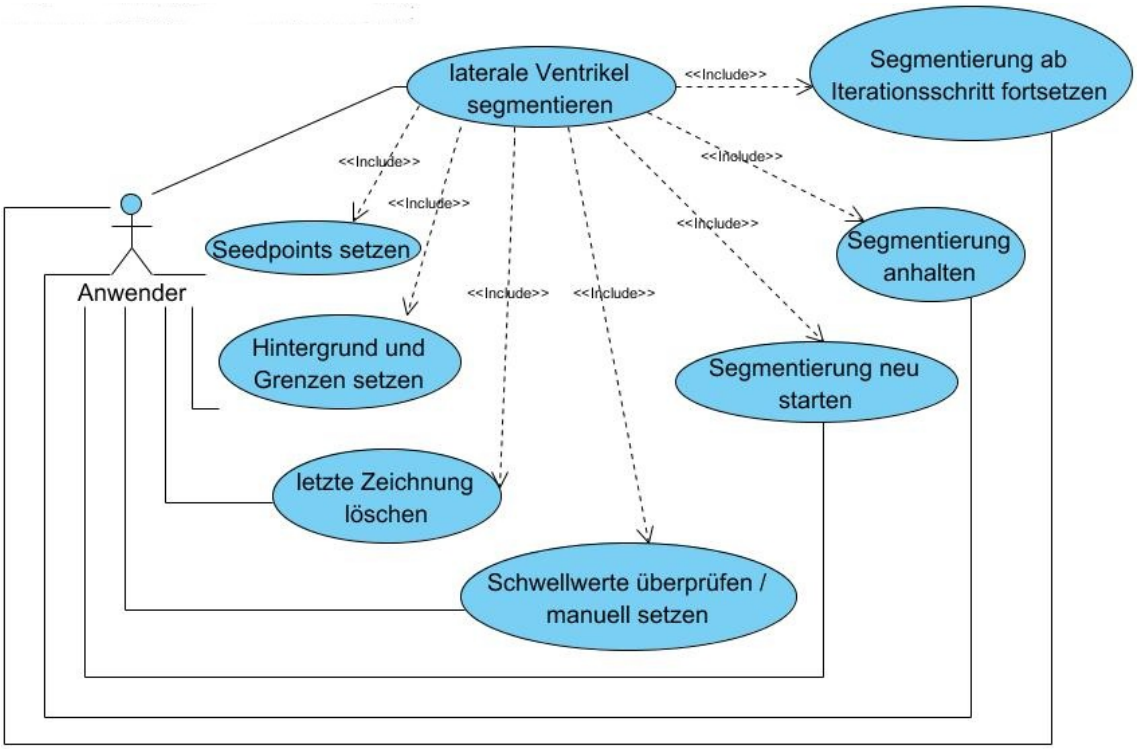

Abbildung 3.9: Anwendungsfalldiagramm des neuen Regiongrowverfahrens. Dargestellt sind die einzelnen Anwendungsschritte, die für eine Segmentierung mittels Regiongrow zur Verfügung stehen.

en Verfahren zunächst *Seedpoints* bestimmt werden. Jedoch soll es hier möglich sein direkt mehrere Punkte zu bestimmen, ohne dass nach jedem einzelnen hinzugefügten Punkt auf ein Zwischenergebnis gewartet werden muss. Im konkreten Anwendungsbeispiel könnten dadurch direkt mehrere Bereiche der lateralen Ventrikel erfasst werden. Außerdem soll der Anwender die Möglichkeit haben dem Regiongrowalgorithmus zusätzliche Informationen zu geben. Es sollen Grenzen gesetzt werden können, so dass der Regiongrower an dieser Stelle nicht weiter laufen kann. Auch soll es möglich sein einen expliziten Hintergrund festzulegen, um bestimmte Grauwertbereiche aus dem Homogenitätskriterium ausschließen zu können. Im Anwendungsbeispiel soll es dem Anwender dadurch möglich sein den Regiongrower daran zu hindern in falsche Strukturen, wie z.B. dem Kortex, auszulaufen. Sowohl das Setzen von *Seedpoints*, sowie das Bestimmen von Grenzen und Hintergrund stellen das Hinzufügen von Points of Interest (deutsch: Punkten von Interesse; kurz: POI) dar. Durch die gesetzten POI soll automatisch ein Grauwertebereich für die zu segmentierende Struktur, sowie ggf. ein Grauwertebereich für den gewählten Hintergrund berechnet werden. So wird es dem Anwender erspart, selbst einen Bereich bestimmen zu müssen. Dennoch soll es dem Anwender möglich sein, die berechneten Grauwertbereiche manuell

zu verändern, um z.B. einen zu groß berechneten Bereich verkleinern zu können. Nach den beschriebenen Anwendungsschritten kann die Segmentierung gestartet werden. Hierbei soll dem Anwender kontinuierlich Rückmeldung über den momentanen Segmentierungsstand gegeben werden. Des Weiteren ist vorgesehen, dass der Anwender die laufende Segmentierung zu jedem Zeitpunkt anhalten kann. Ist die Segmentierung angehalten, soll es möglich sein zu einem beliebigen Iterationsschritt in der Segmentierung zu springen, um dann z.B. neue POI zu setzen oder die Grauwertebereiche anzupassen, um danach die Segmentierung fortzusetzen. So kann z.B. im Anwendungsbeispiel frühzeitig erkannt werden, ob Fehler bei der Segmentierung der Ventrikel entstehen und direkt auf diese reagiert werden. Der Funktionsumfang des zu implementierenden Regiongrowers lässt sich also in folgenden Punkten zusammenfassen :

- Setzen von POI.
- Entfernen von POI.
- Automatische Bestimmung der Grauwertbereiche, sowohl für die zu segmentierende Struktur, als auch ggf. für einen markierten Hintergrund mit der Möglichkeit diese manuell zu verändern.
- Starten des Regiongrowverfahrens.
- Stoppen der laufenden Segmentierung.
- Anzeige eines beliebigen Iterationsschrittes.
- Segmentierung ab einem beliebigen Iteratisschritt fortsetzen.

3.3.1.2 Snakeverfahren

Um den Funktionsumfang des zu entwickelnden Snakeverfahrens zu verdeutlichen, wird in diesem Kapitel der allgemeine Segmentierungsablauf beschrieben. Ein Überblick über die zur Verfügung stehenden Anwendungsschritte bei der Segmentierung ist in Abbildung 3.10 gegeben. Zunächst soll es dem Anwender möglich sein, eine initiale Snake in Form einer geschlossene Kurve zu bestimmen. Diese Kurve sollte die zu segmentierende Struktur möglichst eng umschließen, um schnell zu einem guten Segmentierungsergebnis gelangen zu können. Des Weiteren sollen in der Snake Fixpunkte definiert werden können. Hierbei handelt es sich um Punkte, die nicht durch den Snakealgorithmus verschoben werden können. Mit diesen Fixpunkten wird es dem Anwender möglich z.B. die Form der Snake in kontrastarmen Bereichen einer Struktur vorzugeben, um zu verhindern, dass die Snake durch andere stärkere Kanten angezogen werden kann. Auch können so starke Richtungswechsel markiert werden, welche sonst eventuell durch die Snake nicht genau wiedergeben werden könnten. Als nächstes soll der Anwender die Gewichtung der internen und externen Energie einstellen können. Dies ermöglicht es dem Anwender die Snake individuell auf die

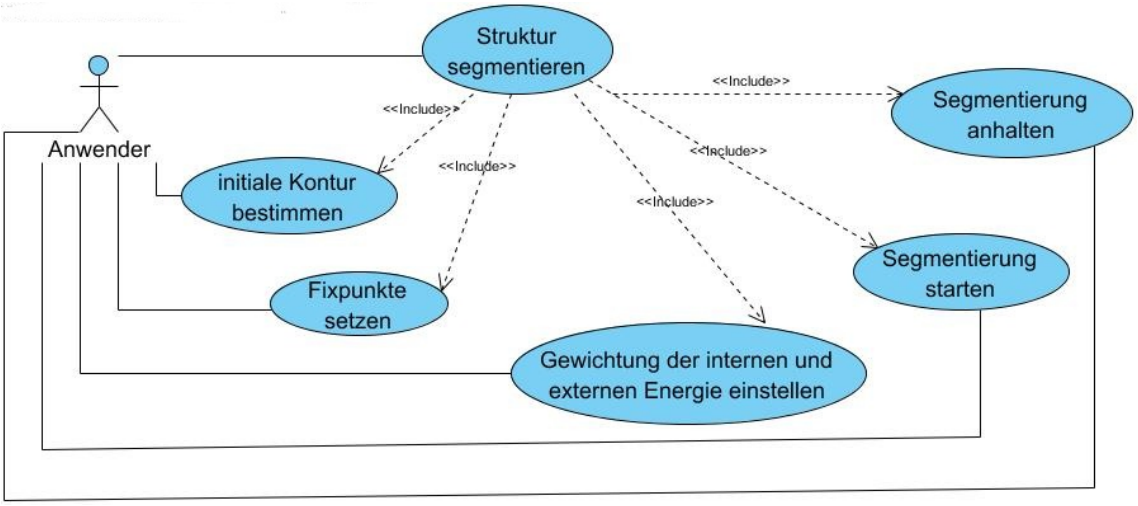

Abbildung 3.10: Anwendungsfalldiagramm des Snakeverfahrens. Dargestellt sind die einzelnen Anwendungsschritte, die für eine Segmentierung mittels Snakes zur Verfügung stehen.

Eigenschaften der zu segmentierenden Struktur einstellen zu können. Sind alle beschriebenen Eingaben vorgenommen worden, soll der Anwender den Snakealgorithmus starten. Hierbei soll kontinuierlich graphisch Rückmeldung über den momentanen Segmentierungsstand gegeben werden. Ähnlich wie bei dem in Kapitel 3.3.1.1 beschriebenen Regiongrower soll es auch in dem zu implementierenden Snakeverfahren möglich sein die laufende Segmentierung jederzeit zu stoppen. Ist die Segmentierung angehalten, soll der Anwender die Gewichtung der internen und externen Energie verändern können. Dies ermöglicht es dem Anwender frühzeitig auf in der Segmentierung entstehende Fehler reagieren zu können. Sollte z.B. die Snake sich an einer Stelle an eine Kante anpassen, die nicht zur gewollten Struktur gehört, könnte die Gewichtung der internen Energie vergrößert werden, um diesem Verhalten entgegen zu wirken. Die Funktionalitäten des zu implementierenden Snakeverfahrens lassen sich also in folgenden Punkten zusammenfassen:

- Bestimmen einer initialen Kurve.
- Setzen von Fixpunkten.
- Gewichtung der internen und externen Energie festlegen.
- Starten des Snakeverfahrens.
- Stoppen der laufenden Segmentierung.

3.3.1.3 Vorgehen bei der Umsetzung

Um zu ermitteln, wie sich Arbeitsschritte durch den Einsatz von OpenCL parallelisieren lassen, wird bei der Umsetzung der in Kapitel 3.3.1.1 und 3.3.1.2 beschriebenen Komponenten nach dem Entwicklungsmodell des Prototyping vorgegangen. So wird zunächst ein

Prototyp außerhalb von MOPS 3D entwickelt, um anschließend durch die so gewonnenen Erkenntnisse eine unabhängige Komponente für den Einsatz in MOPS 3D implementieren zu können.

3.3.2 Bilddaten- und Segmentierungsfilter

Kapitel 3.2.4 beschreibt die Notwendigkeit Bilddaten vor der Segmentierung filtern zu können, sowie den Nutzen, der durch die Filterung von Segmentierungsergebnissen entsteht. Filter, die auf den Bilddaten operieren, werden folgend als Bildfilter bezeichnet und Filter, die auf den Segmentierungsergebnissen operieren als Segmentierungsfilter. Da sich die Bildfilter in der Funktionsweise nicht von den Segmentierungsfiltern unterscheiden, müssen die Filteralgorithmen nur jeweils einmal implementiert werden. Um einen Filter zu verwenden, soll der Anwender lediglich die gewünschte Filterart, die Größe der Filtermaske und den Ausführungszeitpunkt (also vor oder nach der Segmentierung) bestimmen können. Diese Anwendungsschritte sind in Abbildung 3.11 wiedergeben. Die Ausführungsreihenfolge der

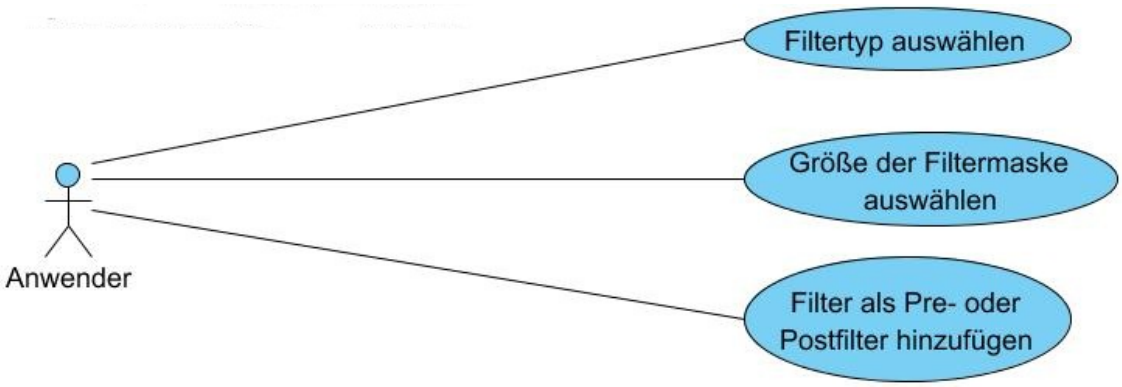

Abbildung 3.11: Anwendungsfalldiagramm für das Hinzufügen der Filter als Bild- oder Segmentierungsfilter.

Filter soll der Reihenfolge entsprechen, wie die einzelnen Filter der Segmentierung hinzugefügt wurden, wobei natürlich alle Bildfilter vor den Segmentierungsfilteren ausgeführt werden. Neben dem Hinzufügen von Filtern soll es des Weiteren möglich sein Filter frei wählbar wieder zu entfernen. Hierdurch sollen Filter, die sich für die aktuelle Segmentierung als überflüssig oder störend erwiesen haben, flexibel gelöscht werden können.

3.3.3 Entwurf der Segmentierungskomponente

In den vorangegangen Kapiteln wurden die einzelnen Bestandteile der zu entwickelnden Segmentierungskomponente aus Anwendersicht erläutert. In diesem Kapitel wird nun ein Entwurf beschrieben, durch den es möglich ist, eine unabhängige Segmentierungskomponente zu entwickeln. Daher werden hier auf geplante Programmabläufe eingegangen.

3.3.3.1 Die Schnittstellen

Für die Umsetzung der unabhängigen Segmentierungskomponente sollen zwei Schnittstellen definiert werden. Die erste Schnittstelle dient dazu, der Anwendung Zugriff auf Funktionen der Segmentierungskomponente zur Segmentierung zu verschaffen. Die zweite Schnittstelle wiederum dient der Segmentierungskomponete dazu, auf nötige Funktionen der Anwendung zugreifen zu können. Im Folgenden werden diese beiden Schnittstellen Segmentierungsschnittstelle und Anwendungsschnittstelle genannt. Die Segmentierungsschnittstelle steht hierbei für die erste Schnittstelle und die Anwendungsschnittstelle für die zweite. Zu beachten ist, dass der Entwurf und die Umsetzung der Anwendungsschnittstelle in einer separaten Diplomarbeit von Herrn Sascha Diatschuk durchgeführt wurde ([20]). Da einige Funktionalitäten der Anwendungsschnittstelle für das Verständnis der Interoperation zwischen Anwendungsschnittstelle und Segmentierungsschnittstelle unabdingbar sind, werden diese im weiteren Verlauf dennoch beschrieben. So wird die Funktionalität der Segmentierungsschnittstelle von Seiten der Segmentierungskomponente implementiert. Die Anwendungsschnittstelle wird jedoch lediglich in der Segmentierungskomponente deklariert und von Seiten der Anwendung implementiert. Hierdurch wird der zu entwickelnden Komponente, unabhängig von der einbindenden Anwendung, immer der gleiche benötigte Funktionsumfang zur Verfügung gestellt. Dieser Funktionsumfang lässt sich grundlegend in drei Punkten zusammenfassen, die nachfolgend erläutert werden:

- Zugriff auf *OpenGL*- und *Device Kontext*.
- Zugriff auf die *Textur*.
- Methode zum Zeichnen der *OpenGL Textur*.

Der Zugriff auf den *OpenGL*- und *Device Kontext* wird benötigt, um einen geteilten *Kontext* zwischen OpenGL und OpenCL erstellen zu können, wie er in Kapitel 2.1.4 beschrieben ist. Hierdurch wird es möglich in OpenCL mit OpenGL geteilte *Memory Objects* zu erstellen. Um ein solches *Memory Object* erstellen zu können, muss zusätzlich zum *OpenGL Kontext* Zugriff auf die *Textur ID* der *OpenGL Textur* gewährt werden, die segmentiert werden soll. Dadurch soll es möglich werden, die Darstellung direkt auf der Grafikkarte durch die Segmentierung zu beeinflussen. Um Segmentierungsergebnisse oder Zwischenschritte an geeigneten Zeitpunkten anzeigen zu können, muss des Weiteren eine Methode zum Zeichen der *OpenGL Textur* von der Anwendungsschnittstelle zur Verfügung gestellt werden.
Die von der Segmentierungsschnittstelle bereitgestellten Methoden sollen die in den Kapiteln 3.3.1.1, 3.3.1.2 und 3.3.2 beschriebenen Funktionalitäten umfassen und lassen sich daher grob in den folgenden Punkten zusammenfassen:

- Hinzufügen und Entfernen von Bild- und Segmentierungsfiltern.
- Auswählen des Segmentierungsverfahrens.
- Konfiguration der einzelnen Segmentierungsverfahren.

- Bereitstellung von Gettern und ggf. Settern für verschiedene Werte, wie z.B. die ermittelten Schwellwerte nach dem Hinzufügen von *Seedpoints*.
- Starten und Stoppen der Segmentierung.

Um die Funktionsweise der beiden Schnittstellen im Zusammenhang mit einer einbindenden Anwendung zu verdeutlichen, werden im Folgenden zwei Programmabläufe beispielhaft beschrieben. Als erstes Beispiel wird auf das Erstellen der Segmentierungskomponente eingegangen. Der dafür benötigte Programmablauf ist in Abbildung 3.12 wiedergegeben. Zunächst erfolgt ein Konstruktoraufruf von Seiten der Anwendung an die Segmentierungs-

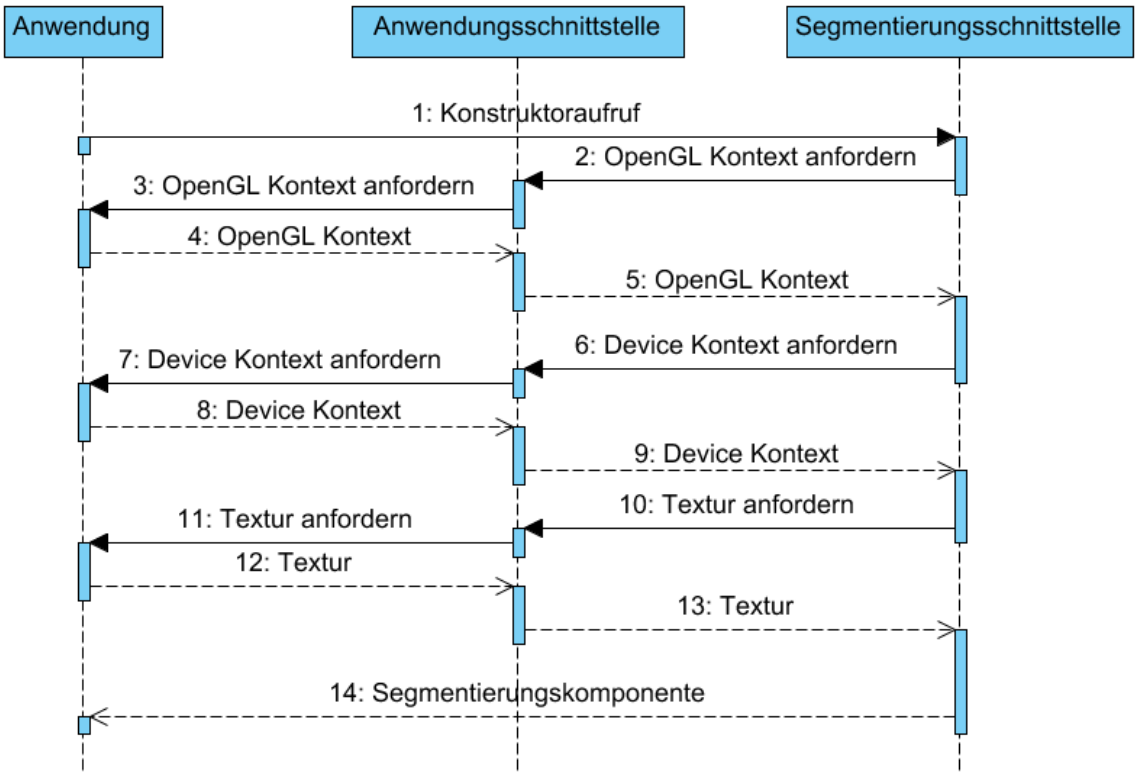

Abbildung 3.12: Sequencediagramm für das Erstellen der Segmentierungskomponente.

komponente (Schritt 1:Konstuktoraufruf in Abbildung 3.12). Innerhalb des Konstruktors sollen dann der geteilte *Kontext* und das geteilte *Memory Object* erstellt werden. Hierzu soll durch die Anwendungsschnittstelle zunächst auf den *OpenGl Kontext* und dann auf den *Device Kontext* zugegriffen werden, um anhand dieser den geteilten *Kontext* erstellen zu können. Dies ist in Abbildung 3.12 durch die Schritte »2: OpenGL Kontext anfordern« bis »9: Device Kontext» wiedergegeben. Ist der geteilte *Kontext* erstellt, soll das geteilte *Memory Object* erzeugt werden. Dafür wird von der Segmentierungskomponente mittels der Anwendungsschnittstelle die *Textur ID* erfragt und anhand dieser das *Memory Object* erzeugt (Abbildung 3.12 Schritte »10: Textur anfordern« bis »13: Textur«). Sind die beschriebenen Schritte bearbeitet ist die Segmentierungskomponente einsatzbereit und wird als Objekt an die Anwendung übergeben.
Als zweites Beispiel wird ein möglicher Programmablauf bei der Segmentierung mittels des

Regiongrowverfahrens dargestellt. Wiedergegeben ist dieser Ablauf in Abbildung 3.13. Zu

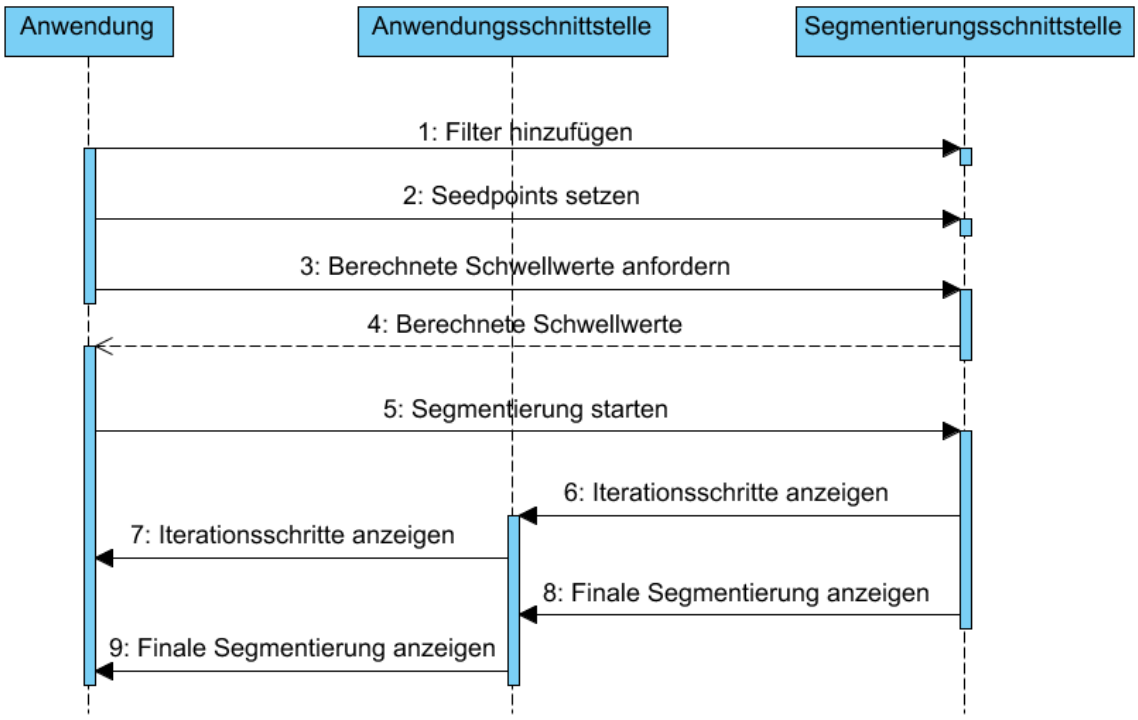

Abbildung 3.13: Sequencediagramm für einen beispielhaften Programmablauf bei der Segmentierung.

beachten ist, dass hierfür zuvor von der Anwendung das entsprechende Segmentierungsverfahren mittels der Segmentierungsschnittstelle ausgewählt werden muss. Wie in den Kapitel 3.3.2 beschrieben, soll es dem Nutzer möglich sein verschiedene Filter der Segmentierung hinzuzufügen. Dies ist aus Anwendungssicht in Abbildung 3.13 in Schritt »1: Filter hinzufügen« wiedergegeben. Hier werden von der Anwendung die zu verwendenden Filter mittels der Komponnentenschnittstelle bestimmt. Gleiches gilt für die durch den Nutzer bestimmten *Seedpoints*. In den Schritten »3: Berechnete Schwellwerte anfordern« und »4: Berechnete Schwellwerte« wird von der Anwendung auf die ermittelten Schwellwerte zugegriffen, um diese z.B. dem Nutzer anzeigen zu können. In Schritt »5: Segmentierung starten« wird dann von der Anwendung die eigentliche Segmentierung mittels der Segmentierungsschnittstelle angestoßen. Im Verlauf der Segmentierung sollen immer wieder Zwischenergebnisse angezeigt werden, welches durch die Anwendungsschnittstelle ermöglicht wird. Hier wiedergegeben durch die Schritte 6 und 7. Nach abgeschlossener Segmentierung soll dann die finale Segmentierung angezeigt werden (Schritte 8 und 9). Zu beachten ist, dass Anwendung und Segmentierungskomponente bei der Segmentierung (Schritte 5 bis 9) parallel laufen sollen. Hierdurch wird es trotz laufender Segmentierung möglich, weiter mit der Anwendung zu interagieren. Dies ist notwendig, um z.B. auf Nutzereingaben wie das Anhalten der Segmentierung reagieren zu können.

4 Implementierung

In diesem Kapitel werden wichtige technische Details der Implementierung erläutert. Dazu wird zunächst auf den Klassenentwurf für die entwickelte Segmentierungskomponente eingegangen. Anschließend werden die Kernfunktionalitäten der einzelnen Klassen erläutert. Hierbei liegt das Augenmerk nicht darauf einen vollständigen detaillierten Überblick über die entstandene Implementierung zu vermitteln. Es werden vielmehr wichtige Aspekte der Implementierung im Zusammenhang mit OpenCL erläutert. An der kompletten Implementierung interessierte Leser seien daher auf die Dokumentation des entstandenen Codes verwiesen.

4.1 Klassenentwurf

Dieses Kapitel soll eine grobe Übersicht über die erstellten Klassen, deren Beziehung untereinander, sowie der allgemeinen Funktionalitäten der einzelnen Klassen verschaffen. Zunächst wird auf den Entwurf der Klassen eingegangen, die die in Kapitel 3.3.3.1 beschriebenen Schnittstellen realisieren. Diese sind in Abbildung 4.1 wiedergegeben. Die *SegmentationManager*-Klasse entspricht dabei der Segmentierungsschnittstelle und die *GPUSegmentationBase*-Klasse der Anwendungsschnittstelle. Zu beachten ist, dass hier lediglich die wichtigsten Methoden wiedergegeben sind und z.B. etwaige Getter und Setter außer Acht gelassen werden. Folgende Auflistung gewährt einen kurzen Überblick über die einzelnen Methoden der *SegmentationManager*-Klasse:

- SegmentationManager: Konstruktor der *SegmentationManager*-Klasse. Ihm muss ein Objekt übergeben werden, welches die abstrakte Klasse *GPUSegmentationBase* realisiert, so dass auf benötigte Funktionen der Anwendung zugegriffen werden kann. Im Konstruktor selbst wird dann ein mit OpenGL geteilter *Kontext* sowie ein mit der zu segmentierenden Textur geteiltes *Memory Object* erzeugt.

- ~SegmentationManger: Destruktor der *SegmentationManager*-Klasse. Hier werden alle beanspruchten Ressourcen wieder freigegeben.

- setMode: Setzt das zu verwendende Segmentierungsverfahren. Hierzu wird von der *SegmentationManager*-Klasse ein entsprechender Aufzählungstyp bereitgestellt, welcher als Argument für die Methode erwartet wird.

- addPreFilter, addPostFilter, removePreFilter, removePostFilter: Setzt das in Kapitel 3.3.2 beschriebene Konzept von Bild- und Segmentierungsfiltern um. Durch den

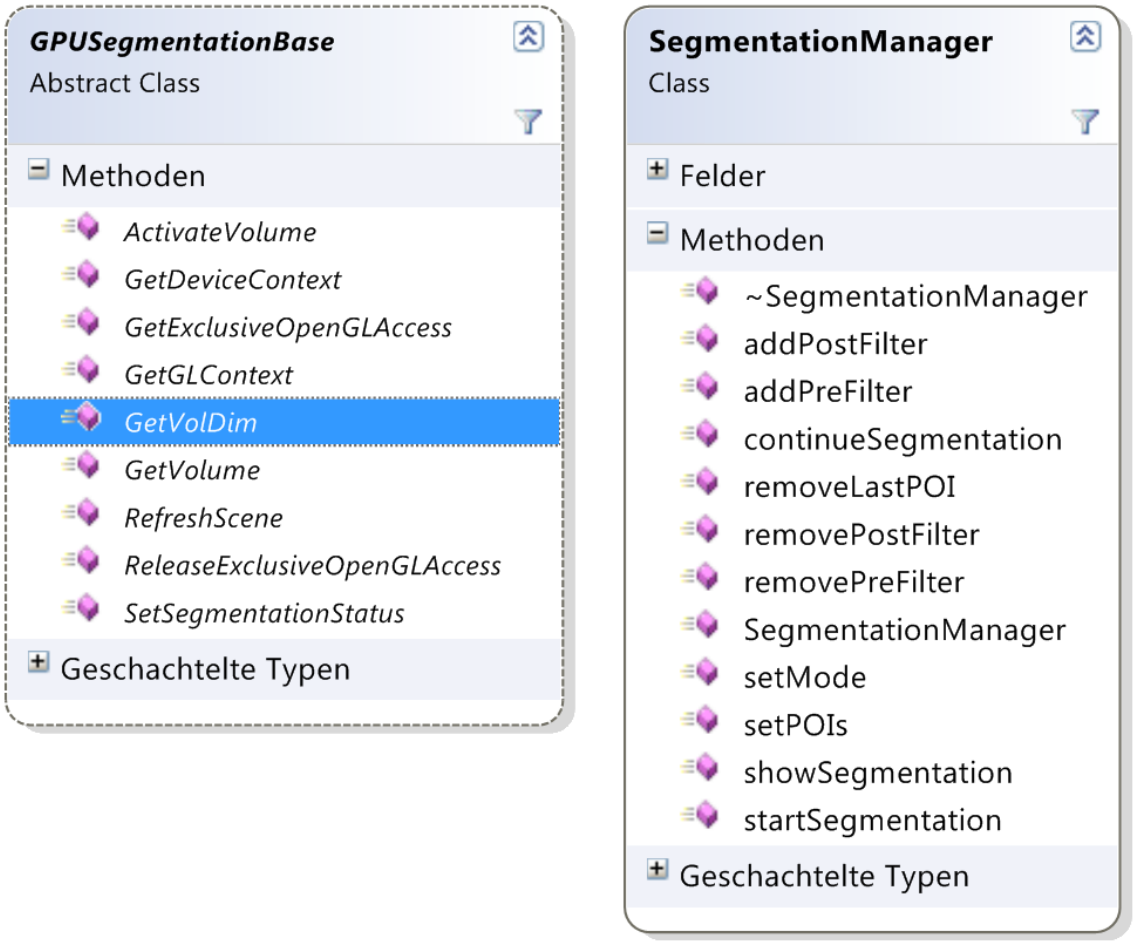

Abbildung 4.1: Wiedergegeben ist das Klassendiagramm der beiden Klassen, die die in Kapitel 3.3.3.1 beschriebenen Schnittstellen realisieren.

Aufruf von addPrefilter bzw. addPostFilter wird dem entsprechend ein Bild- bzw. Segmentierungsfilter der Segmentierung hinzugefügt. Die Methoden removePreFilter und removePostFilter ermöglichen es einen bestimmten Bild- bzw. Segmentierungsfilter von der Segmentierung zu entfernen.

- setPOIs: Ermöglicht das Hinzufügen von POI zum Markieren von *Seedpoints*, Grenzen und Hintergrund, wie sie in Kapitel 3.3.1.1 beschrieben wurden.
- removeLastPOI: Entfernt die zuletzt hinzugefügten POI. Mehrfacher Aufruf dieser Methode führt dazu, dass nach und nach Weiter POI entfernt werden, bis keine POI mehr vorhanden sind.
- startSegmentation: Startet die eigentliche Segmentierung. Erneuter Aufruf dieser Methode während einer laufenden Segmentierung führt dazu, dass die Segmentierung angehalten wird.
- continueSegmentation: Setzt die Segmentierung ab dem momentan angezeigten Iterationsschritt fort.

- showSegmentation: Zeigt einen bestimmten Iterationsschritt an, wobei die Nummer des anzuzeigenden Iterationsschrittes als Argument an die Methode übergeben wird.

Folgende Auflistung fasst die von Seiten der Segmentierungskomponente erwarteten Funktionalitäten der einzelnen Methoden der *GPUSegmentationBase*-Klasse zusammen:

- GetDiveceContext, GetCLContext: Gewährt Zugriff auf den *Device*- und *OpenGL Kontext*, um im Konstruktor des SegmentationManagers einen geteilten *Kontext* erstellen zu können.
- GetVolume, GetVolDim: Gewährt Zugriff auf die zu segmentierende Textur, sowie auf die Maße dieser, um ein geteiltes *Memory Object* erzeugen zu können.
- GetExclusiveOpenGLAccess, ReleaseExclusiveOpenGLAccess: Ermöglicht es OpenGL- und OpenCL-Aufrufe an benötigten Stellen zu synchronisieren.
- ActivateVolume: Da eine 3D-Anzeige des zu segmentierenden Volumens vor der Segmentierung nicht sinnvoll ist (es würde einfach ein Block angezeigt), wird es durch diese Methode ermöglicht, die Anzeigte des Volumens zu einem geeigneten Zeitpunkt zu aktivieren.
- RefreshScene: Zeichnet die zu segmentierende Textur neu, um Zwischenschritte bei der Segmentierung, sowie das Endergebnis anzeigen zu können.
- SetSegmentationStatus: Teilt der Anwendung mit, ob eine Segmentierung momentan durchgeführt wird.

Nachdem die Funktionalitäten der Schnittstellen erläutert wurden, wird nun auf die Bestandteile der Segmentierungskomponente eingegangen, die mittels *Kernel* auf der Grafikkarte operieren. Hierbei handelt es sich um das Regiongrowverfahren, das Snakeverfahren sowie die verschiedenen in Kapitel 2.2 beschriebenen Filter. Ein Entwurf dieser Klassen ist in Abbildung 4.2 wiedergegeben. Wie zu sehen ist, werden alle Klassen von der übergeordneten Klasse GPUTool abgeleitet. Hierdurch werden grundlegende Funktionalitäten an die einzelnen Bestandteile vererbt, so dass diese nicht mehrfach in den einzelnen Klassen implementiert werden müssen. Dabei handelt es sich vor allem um eine Methode zum Laden und Kompilieren des *OpenCL Programmcodes*. Des Weiteren werden Felder zur Verfügung gestellt, die Referenzen auf den verwendeten geteilten *Kontext*, sowie auf die verwendete *Command Queue* enthalten. Das Erstellen der einzelnen benötigten *Kernel* sowie die Verwendung dieser durch geeignete Methoden muss dann in den abgeleiteten Klassen implementiert werden. In Abbildung 4.2 ist des Weiteren zu sehen, dass die verschiedenen zu implementierenden Filter durch eine einzige Klasse repräsentiert werden. Dies ist daher möglich, da der einzige Unterschied der einzelnen Filter in ihre Implementierung des *Kernels* zum Filtern des Volumens besteht. Einzige Ausnahme ist hier der Gauß Filter, da dessen Filtermaske aus Performancegründen nicht im *Kernel*, sondern im *Host-Programm* berechnet wird und als zusätzliches Argument an den *Kernel* übergeben wird. Weitere

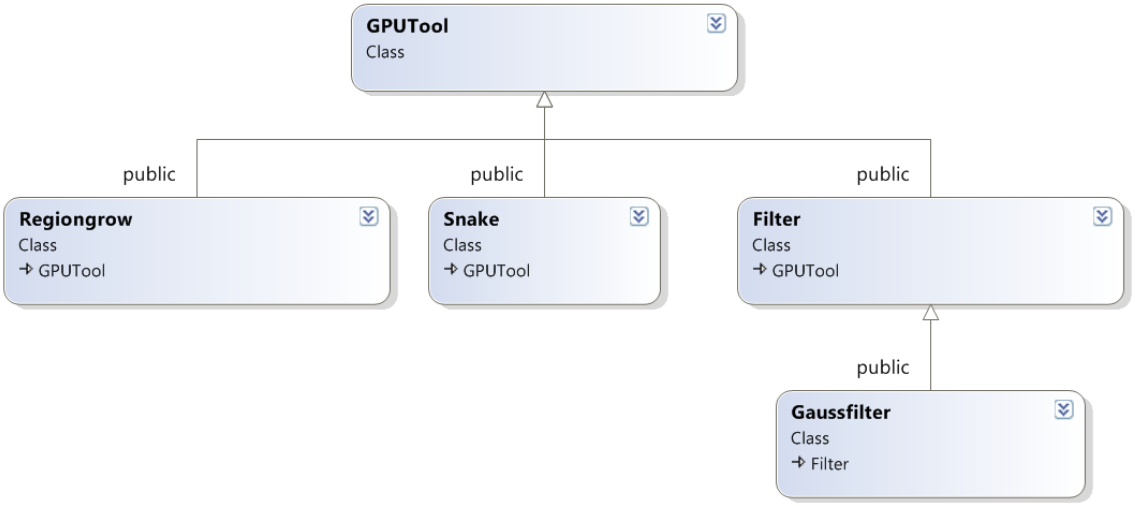

Abbildung 4.2: Das dargestellte Klassendiagramm zeigt die Konzeption der einzelnen Bestandteile der Segmentierungskomponente, die mittels *Kernel* auf der Grafikkarte operieren.

Details der Filter sind in dem Kapitel 4.2.4 beschrieben.
Zu beachten ist, dass die Snake-Klasse hier nur der Vollständigkeits halber wiedergegeben ist. Bei der in dieser Diplomarbeit entstandenen Implementierung wurde das Snakeverfahren nicht über einen prototypischen Stand hinaus entwickelt. Es soll hierdurch lediglich veranschaulicht werden, wie ein Snakeverfahren in die Segmentierungskomponente eingebunden werden kann.

4.2 Realisierung

In diesem Kapitel werden Details der in dieser Diplomarbeit entstandenen Implementierung erläutert. Es wird dargestellt, wie Probleme und Herausforderungen, die sich bei der Implementierung ergaben, gelöst wurden. Dabei wird vor allem auf Codestellen des *Host-Programms* eingegangen, die Funktionen von OpenCL verwenden, um die Interoperation zwischen *Host* und *OpenCL Devices* zu verdeutlichen. Des Weiteren werden wichtige Detail der verwendeten *Kernel* beschrieben. Der Ablauf der folgenden Kapitel orientiert sich dabei an dem Klassengerüst wie es in Kapitel 4.1 beschrieben wurde. Hierdurch soll verdeutlicht werden welche Klasse sich mit welchen Problemstellungen befasst.

4.2.1 SegmentationManager

Zentrale Aufgabe der *SegmentationManager*-Klasse ist es den Zugriff durch OpenCL auf das zu segmentierende OpenGL-Volumen herzustellen. Um dies zu bewerkstelligen müssen verschiedene Vorbereitungen getroffen werden. Sie lassen sich in drei Punkte zusammenfassen:

- Erstellen eines mit OpenGL geteilten Kontextes.

- Erstellen einer passenden *Command Queue*.
- Erstellen eines geteilten *Memory Objects*.

In den nachfolgenden Kapiteln 4.2.1.1, 4.2.1.2 und 4.2.1.3 wird darauf eingegangen wie und warum diese einzelnen Arbeitsschritte durchgeführt werden.

4.2.1.1 Geteilter Kontext

Eine grundlegende Voraussetzung für die Interoperation zwischen OpenCL und OpenGL ist das Erstellen eines geteilten *Kontextes*. Durch diesen wird es möglich mit OpenCL auf den gleichen *Devices* (im konkreten Fall also der gleichen Grafikkarte) wie OpenGL zu operieren. Um diesen erstellen zu können müssen zunächst die *OpenCL Devices* ermittelt werden, für die ein mit OpenGL geteilter *Kontext* erstellt werden soll. Von OpenCL wird dafür die Methode *clGetCLContextInfoKHR* zur Verfügung gestellt. Durch sie ist es möglich alle *OpenCL Devices* zu ermitteln, die mit einem bestimmten OpenGL Kontext assoziiert werden. In der Implementierung wird der *OpenGL Kontext*, mit dem ein geteilter *Kontext* erstellt werden soll, durch die entsprechenden Methoden der *GPUSegmentationBase* ermittelt.

Da es sich bei der Interoperation zwischen OpenGL und OpenCL um eine Erweiterung des OpenCL Standards handelt, kann die Methode *clGetCLContextInfoKHR* jedoch nicht ohne weiteres im *Host-Programm* aufgerufen werden. Hierzu muss zunächst die Adresse der Methode ermittelt werden. Dazu wird von OpenCL die Methode *clGetExtensionFunctionAddress* bereitgestellt. Ihr wird als Parameter der Name der gesuchten Methode übergeben. Zurückgegeben wird ein Void Pointer, der dann noch in das entsprechende Methodenformat umgewandelt werden muss. Sind die beiden beschriebenen Methoden ausgeführt, kann dann letztendlich mittels *clCreateContext* der geteilte *Kontext* erzeugt werden. Der vollständige Programmablauf der eben beschriebenen Methoden um einen geteilten *Kontext* zu erzeugen, ist in Listing 4.1 wiedergegeben.

```
1 //Ermitteln der Methodenadresse von clGetGLContextInfoKHR.
  //(P1) wandelt den ermittelten Void Pointer in das passende Format um.
3 myclGetGLContextInfo = (P1)clGetExtensionFunctionAddress(
    "clGetGLContextInfoKHR");
5 //clGetGLCOntextInfoKHR ausführen, um alle mit dem OpenGL Kontext
  //assoziierten OpenCL Devices zu ermitteln.
7 myclGetGLContextInfo(..., devices, ...);
  //Geteilten Kontext für die gefundenen Devices erzeugen.
9 context = clCreateContext(..., devices, ...);
```

Listing 4.1: Erstellen eines geteilten Kontextes. Zunächst werden alle mit dem OpenGL Kontext assoziierten Devices ermittelt und dann für diese ein geteilter Kontext erstellt.

4.2.1.2 Command Queue

Nach dem, wie in dem in Kapitel 4.2.1.1 beschrieben Vorgehen, ein geteilter *Kontext* für alle mit OpenGL assoziierten Devices erzeugt wurde, muss um mit OpenCL arbeiten zu können, eine *Command Queue* (beschrieben in Kapitel 2.1.2.2) für das passende *OpenCL Device* erstellt werden. Die *Command Queue* organisiert und regelt die Ausführung von OpenCL Aufrufen auf einem bestimmten *OpenCL Device*. Da auf der Grafikkarte operiert werden soll, muss genau dieses *Device* ermittelt werden. Um dies zu bewerkstelligen kann die von OpenCL bereitgestellte Methode *clGetDeviceInfo* verwendet werden. Durch sie ist es möglich zu ermitteln, um welchen Gerätetyp es sich bei einem bestimmten *Device* handelt.

In Listing 4.2 ist wiedergegeben wie in der Implementierung der *SegmentationManager*-Klasse die eben beschriebene Methode verwendet wird, um das passende *OpenCL Device* zu ermitteln. Des Weiteren wird anschließend mittels *clCreateCommandQueue* eine *Command Queue* für dieses Device erstellt.

```
1  //Liste aller gefundenen OpenCL Devices durchgehen.
   for(i=0; i< count;i++)
3  {
     //Devicetyp an der Stelle devices[i] ermitteln.
5    clGetDeviceInfo(devices[i],
                     CL_DEVICE_TYPE,
7                    sizeof(cl_device_type),
                     &device_type,
9                    &ret_size);
     //Handelt es sich um ein GPU Device?
11   if(device_type == CL_DEVICE_TYPE_GPU)
     {
13     //ID des GPU Devices eintragen
       gpuDevice = devices[i];
15     //und For-Schleife verlassen.
       i = count;
17   }
   }
19
   //Warteschlange zum Ausführen von OpenCL Anweisungen erstellen.
21 command_queue = clCreateCommandQueue(context, gpuDevice, ...);
```

Listing 4.2: Erstellen der Command Queue

So können mittels einer Schleife alle gefundenen *OpenCL Devices* (siehe Kapitel 4.2.1.1) durchgegangen werden, um mittels *clGetDeviceInfo* den Typ des jeweiligen Devices zu ermitteln. Ist das *Device* gefunden worden, welches die Grafikkarte repräsentiert, kann für dieses eine *Command Queue* erstellt werden.

4.2.1.3 Geteiltes Memory Objekt

In Kapitel 4.2.1.1 wurde ein mit OpenGL geteilter *Kontext* erzeugt. Hierdurch ist die generelle Möglichkeit zur Interoperation zwischen OpenGL und OpenCL gewährleistet. In Kapitel 4.2.1.2 wurde dann eine *Command Queue* erstellt, durch die OpenCL auf der gleichen Grafikkarte, auf der auch OpenGL ausgeführt wird, arbeitet. Was noch fehlt ist der Zugriff auf die zu segmentierende *OpenGL Textur*, um direkt auf dieser operieren zu können. Mit diesem Zugriff wird es möglich, die Darstellung von OpenGL mit OpenCL zu beeinflussen, um z.B. einzelne Iterationsschritte bei der Segmentierung anzeigen zu können. Dieser Zugriff wird durch das Erstellen eines geteilten *Memory Object* hergestellt. Wie dieses erzeugt wird ist in Listing 4.3 wiedergegeben.

```
1 //OpenCL/GL Memory Object wird erstellt
  //Textur ID der zu segmentierenden OpenGL Textur mittels
3 //der GPUSegmentationBase ermitteln.
  GLuint textureID = gpuB->GetVolume();
5 //Geteiltes Memory Object erzeugen.
  mem_inputImage = clCreateFromGLTexture3D(context, CL_MEM_READ_WRITE,
      GL_TEXTURE_3D, 0, textureID, 0);
```

Listing 4.3: Erzeugen des mit OpenGL geteilten Memory Objects.

Zunächst muss die ID der zu segmentierenden Textur ermittelt werden. Hierfür wird die Methode *GetVolume*, welche durch die *GPUSegmentationBase*-Klasse bereitgestellt wird, verwendet. Durch die *Textur ID* kann dann mittels *clCreateFromGLTexture3D* ein geteiltes *Memory Object* für die zu segmentierende Textur erzeugt werden.

Nach dem dies geschehen ist, ist die Grundlage für die Interoperation zwischen OpenGL und OpenCL hergestellt. So wird der hier erstellte Zugriff auf das zu segmentierende Volumen (bzw. der *Textur*) in sämtlichen umgesetzten Verfahren verwendet, um Ergebnisse berechnen und anzeigen zu können.

4.2.2 GPUTool

Da sämtliche Bestandteile der Segmentierungskomponete, die mittels *Kernel* auf der Grafikkarte operieren, von der GPUTool-Klasse abgeleitet werden, wird hier auf die wichtigsten Funktionalitäten dieser Klasse eingegangen.

Die Methoden der GPUTool-Klasse verfolgen zwei verschiedene Ziele. Diese sind in der folgenden Auflistung zusammengefasst und werden anschließend erläutert:

- Bereitstellen eines einheitlichen Konzepts, um OpenCL-Sourcecode zu kompilieren.
- Bereitstellen einer Arbeitskopie, um innerhalb von *Kerneln* auf den Daten des geteilten *Memory Objects* operieren zu können.

Zunächst wird hier auf das Kompilieren von OpenCL-Sourcecode eingegangen. So muss in OpenCL um *Kernel* verwenden zu können, zunächst aus einem in *OpenCL C* geschriebenen

Sourcecode ein *OpenCL Programm* erstellt werden, welches dann für die zum *Kontext* gehörenden *OpenCL Devices* kompiliert wird. Die GPUTool-Klasse stellt dafür die Methode *getProgram* zur Verfügung. Ihr wird als Parameter der Dateiname der Datei übergeben, die den zu kompilierenden Sourcecode enthält. Innerhalb dieser Methode wird der Sourcecode eingelesen und daraus ein *OpenCL Programmobjekt* erstellt. Zum Erstellen und Kompilieren des *Programmobjekts* werden dabei die Methoden *clCreateProgramWithSource* und *clBuildProgram* verwendet, welche OpenCL bereitstellt. In Listing 4.4 ist dieser Vorgang wiedergegeben.

```
//Programm erstellen und kompilieren.
  //Programm wird erstellt.
  program = clCreateProgramWithSource(//Der geteilte Kontext.
                                      *context,
                                      //Anzahl der Zeiger in
                                      //&Kernel_src.
                                      1,
                                      //Der Programmcode.
                                      (const char **)&kernel_src,
                                      NULL,0);
  //Programm wird kompiliert.
  clBuildProgram(program,...);
```

Listing 4.4: Ein OpenCl Programm wird erstellt und kompiliert, um OpenCL-Anweisungen auf der Grafikkarte ausführen zu können.

Nach dem Kompilieren können dann in den abgeleiteten Klassen anhand des *Programmobjekts* die benötigten Kernel erzeugt werden. Genaueres dazu ist den Kapiteln 4.2.3 und 4.2.4 zu entnehmen.

Nachfolgend wird erläutert, warum es nötig ist, eine Arbeitskopie von dem geteilten *Memory Object* zu erstellen und wie dies bewerkstelligt wird.

In Kapitel 4.2.1.3 wurde beschrieben, wie es möglich ist, mit OpenCL einen Zugriff auf die zu segmentierende *OpenGL Textur* zu erzeugen. Bei dem dabei erstellten *Memory Object* handelt es sich um ein *Image Object*, wie es in Kapitel 2.1.3 beschrieben ist. Daraus ergibt sich ein Problem, da Methoden zum Schreiben in 3D *Image Object* in *OpenCL C* zu einer optionalen Erweiterung gehören. Diese Erweiterung wird von den, zum Entstehungszeitpunkt dieser Diplomarbeit aktuellen Nvidia Grafikkartentreibern (Vers. 280.26 und ältere) nicht unterstützt. Hierdurch wird es unmöglich Segmentierungsergebnisse direkt während der Ausführung eines Kernels in die zu segmentierende Textur zu schreiben. So kann lediglich durch das *Host-Programm* in das geteilte *Memory Object* geschrieben werden.

Um dieses Problem zu umgehen, muss daher eine Arbeitskopie in Form eines *Buffer Objects* erzeugt werden, auf das dann mittels der Kernel sowohl lesend als auch schreibend zugegriffen werden kann. Vonseiten des *Host-Programms* kann nun an geeigneten Stellen der Inhalt der Arbeitskopie in das geteilte *Memory Object* geschrieben werden. Um diese

Arbeitskopie zu erzeugen und deren Inhalt auf den Inhalt der zu segmentierenden *Textur* zu setzen, wird von der GPUTool-Klasse die Methode *updateImage* bereitgestellt. Listing 4.5 zeigt wie innerhalb dieser Methode die Arbeitskopie mit entsprechendem Inhalt erstellt wird.

```
//Erstellen der Arbeistkopie (noch ohne Inhalt).
mem_orginalPixelValues = clCreateBuffer(*context,
                                        CL_MEM_READ_WRITE,
                                        size,...);
//Zugriff auf das geteilte Memory Object starten.
clEnqueueAcquireGLObjects(..., mem_inputImage, ...);
//Inhalt des geteilten Memory Object in die Arbeistkopie schreiben.
clEnqueueCopyImageToBuffer(*command_queue ,
                           *mem_inputImage,
                           mem_orginalPixelValues, ...);
//Zugriff auf das geteilte Memory Object beenden.
clEnqueueReleaseGLObjects(..., mem_inputImage, ...);
```

Listing 4.5: Erstellen einer Arbeitskopie mit gleichem Inhalt wie das geteilte Memory Object.

Nachteilig an diesem Vorgehen ist, dass wesentlich mehr Speicher als theoretisch benötigt auf der Grafikkarte allokiert werden muss. Auch drosseln die benötigten Kopiervorgänge vonseiten des *Host-Programms* die Geschwindigkeit der entwickelten Verfahren. Sobald das Schreiben in 3D *Image Objects* vonseiten der Grafikkartentreiber unterstützt wird, sollte daher die Implementierung entsprechend nachgerüstet werden. Dies stellt kein Problem dar, da lediglich auf die Kopiervorgänge im *Host-Programm* verzichtet werden müsste und die entwickelten Kernel nur minimal anzupassen sind, um direkt in das geteilte *Memory Object* zu schreiben.

4.2.3 Regiongrow

In diesem Kapitel werden die zentralen Bestandteile der Regiongrow-Klasse beschrieben. Dazu werden zunächst die benötigten *Kernel* und die wichtigsten *Memory Objects* erläutert. Außerdem wird dargestellt, wie das Setzen von POI (siehe Kapitel 3.3.1.1) in der Regiongrow-Klasse realisiert wurde. Anschließend wird auf den Ablauf einer Iteration, wie sie bei einer laufenden Segmentierung durchgeführt wird, eingegangen. Abschließend wird noch der Mechanismus zum Anzeigen einzelner Iterationsschritte dargestellt.

4.2.3.1 Benötigte Kernel und Memory Objects

Um die *Kernel*, welche sich im kompilierten *OpenCL Programm* befinden nutzen zu können, müssen im *Host-Programm* zunächst *Kernel Objekte* erzeugt werden. Hierzu wird von OpenCL die Methode *clCreateKernel* bereitgestellt. Ihr wird das kompilierte *OpenCL Programm*, sowie der Name des *Kernels* übergeben. Die so erstellten *Kernel* können dann zur Ausführung an eine *Command Queue* übergeben werden. In der Regiongrow-Klasse wer-

den zwei *Kernel* verwendet. Dabei handelt es sich um einen *Kernel*, der den eigentlichen Regiongrowalgorithmus implementiert und einen *Kernel*, der dazu verwendet wird, um den Segmentierungsstand anzeigen zu können. Des Weiteren spielt ein zusätzliches *Buffer Object* eine zentrale Rolle in der Regiongrow-Klasse. Dieses *Buffer Object* wird folgend Regionsmaske genannt.

Die Regionsmaske wird dazu verwendet, den momentanen Segmentierungsstand abzuspeichern. Dabei handelt es sich um ein short-Array, wobei für jeden Bildpunkt der zu segmentierenden *OpenGL Textur* ein Element in dem Array vorhanden ist. Über die Werte der Elemente der Regionsmaske wird der momentane Segmentierungsstand wiedergegeben. So werden z.B. durch den Anwender im Volumen eingetragene Grenzen mit dem Wert -1 in den entsprechenden Elementen der Regionsmaske repräsentiert. Hintergrund, bzw. durch den Regiongrowalgorithmus noch nicht zum Segmentierungsobjekt hinzugefügte Bildpunkte, werden in der Regionsmaske mit dem Wert null ausgedrückt. Werte größer null beschreiben die durch den Regiongrowalgorithmus erfasste Region. Zur Veranschaulichung ist ein vereinfachtes Beispiel einer Regionsmaske in Abbildung 4.3 dargestellt. Weiter

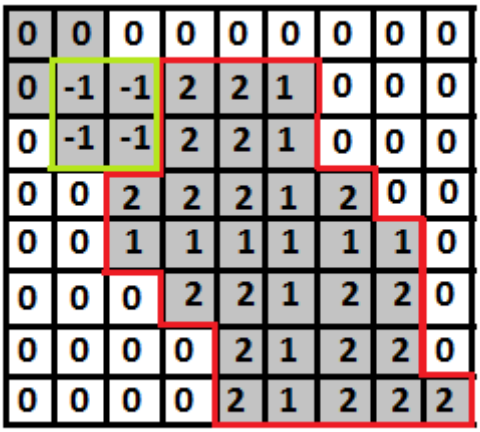

Abbildung 4.3: Dargestellt ist ein vereinfachtes 2D Beispiel der Regionsmaske, wie sie im Regiongrower verwendet wird. Jedes Kästchen des dargestellten Gitters repräsentiert dabei einen Bildpunkt. Die eingetragenen Zahlen wiederum repräsentieren den zum jeweiligen Bildpunkt enthaltenen Wert in der Regionsmaske. Rot umrandet ist dabei die durch den Regiongrowalgorithmus erfasste Region. Der grün umrandetet Bereich stellt eine eingetragene Grenze dar.

ist anzumerken, dass über die Zahlenwerte der Regionsmaske auch bestimmt werden kann in welchem Iterationsschritt ein Bildpunkt zur Region hinzugefügt wurde. So wären z.B. in dem in Abbildung 4.3 wiedergegeben Beispiel alle Kästchen, die eine eins enthalten im ersten Iterationsschritt erfasst worden. Alle Kästchen, die eine zwei enthalten, wären dem entsprechend im zweiten Iterationsschritt der Region hinzugefügt worden. Diese Eigenschaft der Regionsmaske ist nötig, um zu ermöglichen, dass bestimmte Iterationsschritte

angezeigt werden können (siehe Kapitel 4.2.3.4).

4.2.3.2 Setzen von POIs

Als interaktives Segmentierungsverfahren stellen durch den Anwender zur Verfügung gestellte Informationen einen grundlegenden Bestandteil des Regiongrowalgorithmus' dar. In diesem Kapitel wird daher der Mechanismus der Regiongrow-Klasse erläutert, welcher es ermöglicht POIs (siehe Kapitel 3.3.1.1) zu setzen.

Wie bereits in Kapitel 4.2.3.1 beschrieben wird zum Halten der verschiedenen POIs eine Regionsmaske verwendet. Daher ist es erforderlich, dass neue POIs, die von der Anwendung an die Segmentierungskomponente übergeben wurden, in diese eingetragen werden. Hierfür wird eine Kopie der Regionsmaske im Hauptspeicher gehalten, in die dann neue POIs eingetragen werden. Sinn dieser Kopie ist es, dass nicht für jeden hinzugefügten POI auf den Grafikkartenspeicher zugegriffen werden muss. Statt dessen wird die Kopie der Regionsmaske nur einmal vor dem Durchführen des Regiongrowalgorithmus' in die Regionsmaske im Grafikkartenspeicher geschrieben (erläutert in Kapitel 4.2.3.3). Weiter zu beachten ist, dass je nach dem, um was für einen POI es sich handelt, weitere Berechnungen durchgeführt werden müssen. Beim Setzen von Seedpoints müssen z.B. die Schwellwerte für das Homogenitätskriterium überprüft und eventuell neu gesetzt werden.

Zunächst wird der Seedpoint in die Hauptspeicherkopie der Regionsmaske eingetragen. Hierzu wird das entsprechende Element der Regionsmaske auf eins gesetzt. Anschließend wird der Grauwert an der Stelle des Seedpoints aus der Arbeitskopie des geteilten *Memory Objects* gelesen, um dann bei Bedarf die Schwellwerte für das Homogenitätskriterium anpassen zu können. Zur Veranschaulichung ist dieses Vorgehen nochmals in Abbildung 4.4 wiedergegeben. Das Vorgehen beim Setzen von Hintergrund und Grenzen (siehe Kapitel 3.3.1.1) ist ähnlich. Wobei beim Setzen von Grenzen das Berechnen neuer Schwellwerte entfällt und lediglich an die entsprechende Position der Regionsmaske eine -1 eingetragen werden muss. Im Gegensatz dazu wird beim Setzen von Punkten, die den Hintergrund festlegen, die Regionsmaske nicht verändert und nur die Schwellwerte für den Hintergrund bei Bedarf angepasst.

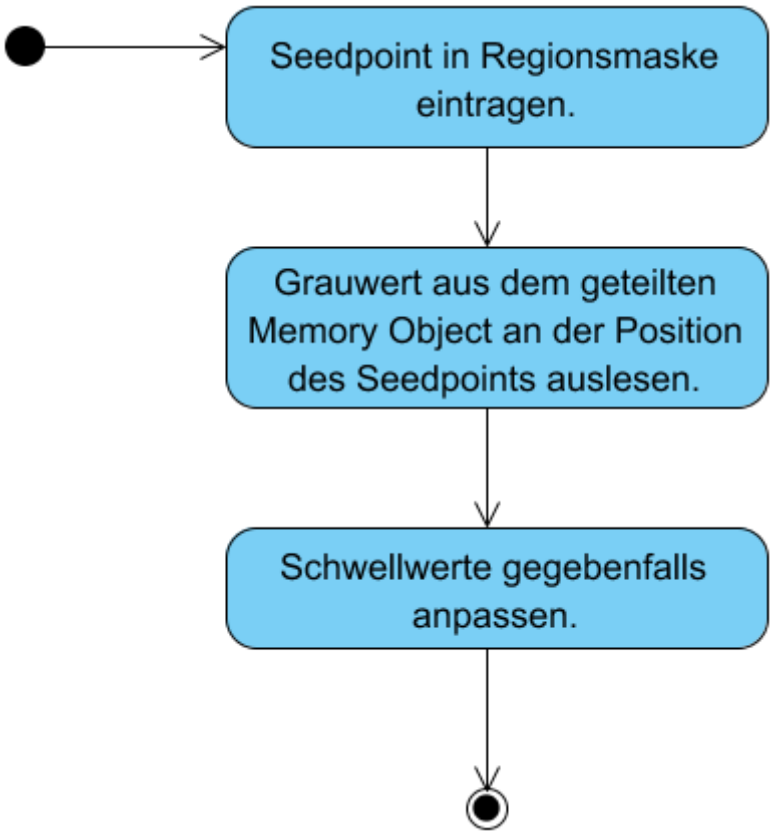

Abbildung 4.4: Programmablauf beim Hinzufügen eines Seedpoints.

4.2.3.3 Durchführung der Iterationen

Das iterative Durchführen des Regingrowalgorithmuses bildet das Kernstück der Regiongrow-Implementierung. Hierdurch wird die tatsächliche Segmentierung ausgeführt. Zunächst wird der Inhalt der Regionsmaske aus dem Hauptspeicher in die Regionsmaske, welche sich im Speicher der Grafikkarte befindet, geschrieben. Zu beachten ist, dass dieser Kopiervorgang nur einmal vor dem Durchführen der Iterationen vorgenommen werden muss. Anschließend wird eine While-Schleife gestartet, die so lange ausgeführt wird, bis sich nach einer Iteration die Regionsmaske und damit das momentane Segmentierungsergebnis nicht mehr verändert hat. Um zu erfassen, ob es eine Änderung gab, wird die bool Variable *rChanged* und das *Memory Object mem_regionChanged* verwendet. Zu Anfang einer jeden Iteration wird *rChanged* zunächst auf *false* gesetzt. Danach wird der Inhalt von *rChanged* in *mem_regionChanged* kopiert. Anschließend wird der Kernel, der den Regiongrowalgorithmus implementiert, durch Aufruf der Methode *clEnqueueNDRangeKernel* ausgeführt. Durch entsprechende Paramenter wird durch den Aufruf von *clEnqueueNDRangeKernel* festgelegt, dass für jeden Bildpunkt der zu segmentierenden *Textur* ein *Work Item* gestartet wird. Dies führt dazu, dass der Regiongrowalgorithmus hochgradig parallel ausgeführt wird. Innerhalb der *Work Items* wird für den jeweiligen Bildpunkt anhand der Regionsmaske überprüft, ob es sich um einen Punkt der segmentierten Region handelt. Ist dies der Fall, werden die benachbarten Bildpunkte auf Ähnlichkeit im Hinblick auf das Ho-

mogenitätskriterium überprüft und falls dieses erfüllt ist, wird ein entsprechender Eintrag in die Regionsmaske vorgenommen. Sollten neue Bildpunkte dem Segmentierungsergebnis hinzugefügt worden sein, wird von dem entsprechenden *Work Item* der Inhalt des *Memory Object mem_regionChanged* auf *true* gesetzt. Nach dem der *Kernel* vollständig ausgeführt wurde, wird der Inhalt von *mem_regionChanged* in die Varibale *rChanged* kopiert. Ist *rChanged* gleich *true* wird eine erneute Iteration mittels der While-Schleife gestartet. Zur Verdeutlichung ist der eben beschriebene Ablauf noch einmal in Abbildung 4.5 wiedergegeben. Der Mechanismus, wie die einzelnen Iterationsschritte angezeigt werden, ist in dem folgenden Kapitel 4.2.3.4 beschrieben.

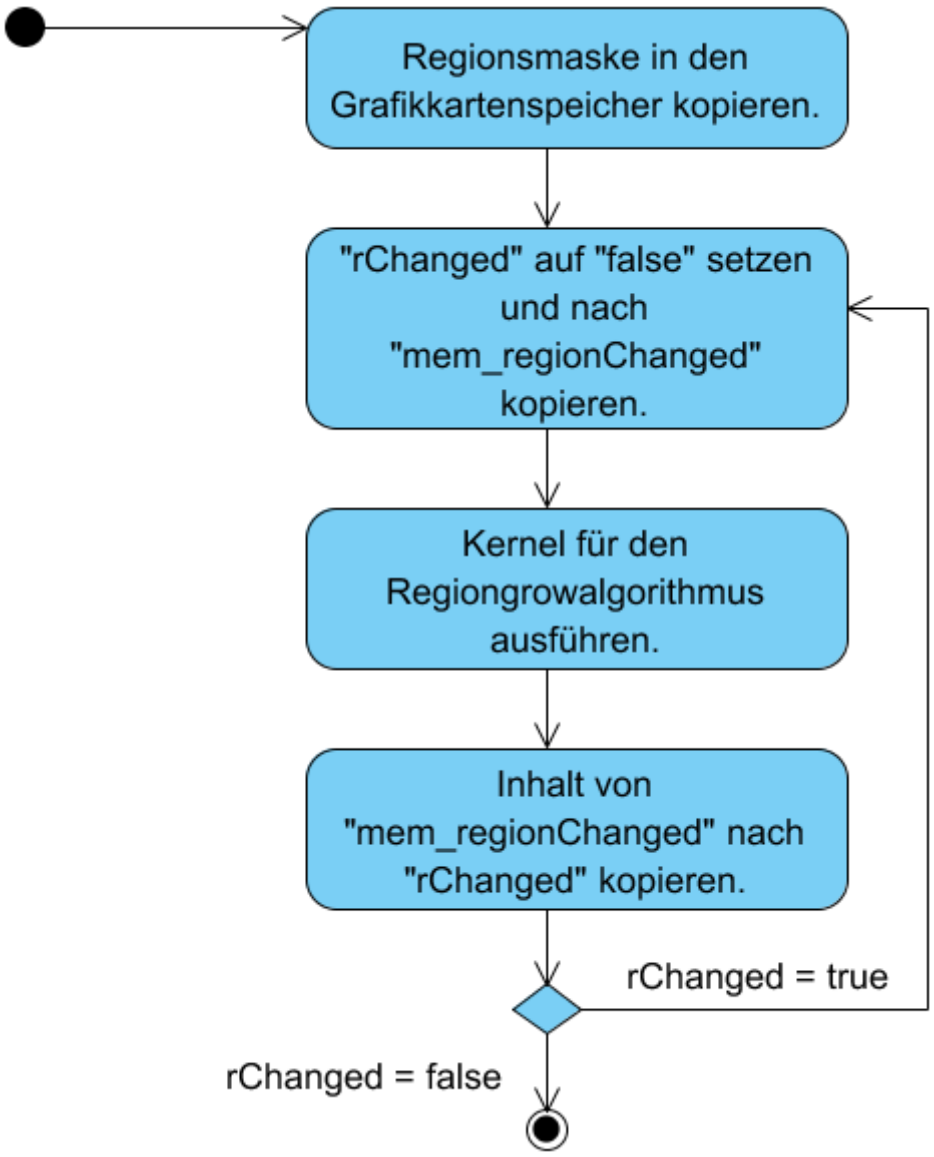

Abbildung 4.5: Ablauf der Iterationen bei der Segmentierung.

4.2.3.4 Anzeigen eines Iterationsschrittes

In Kapitel 4.2.2 wurde erläutert, dass die ausgeführten *Kernel* nicht direkt in das geteilte *Memory Object* schreiben können. Daher ist es nötig durch das *Host-Programm* entsprechende Schreibvorgänge vorzunehmen. Hierzu werden die anzuzeigenden Daten zunächst durch die Ausführung eines *Kernels* in ein *Buffer Object* geschrieben. Der Inhalt dieses *Buffer Objects* kann dann mittels der Methode *clEnqueueCopyBufferToImage* in das geteilte *Memory Object* geschrieben werden. Dabei gilt es zu beachten, dass während OpenCL auf dem geteilten *Memory Object* operiert nicht durch OpenGL auf dieses zugegriffen werden darf. Dies könnte sonst zu einem undefinierten Zustand der *Textur* führen. Da wie in Kapitel 3.3.3.1 erwähnt die Segmentierung parallel zur einbindenden Anwendung ausgeführt wird, könnte jedoch genau dies vorkommen. Um dieses Problem zu handhaben werden von der *GPUSegmentationBase* die Methoden *GetExclusiveOpenGLAccess* und *ReleaseExclusiveOpenGLAccess* zur Verfügung gestellt. Durch sie kann ein exklusiver Zugriff auf das geteilte *Memory Object* gestartet und beendet werden. Dem entsprechend muss *GetExclusiveOpenGLAccess* vor dem Schreiben in das *Memory Object* und *ReleaseExclusiveOpenGLAccess* nach abgeschlossenem Schreiben aufgerufen werden, so dass sichergestellt wird, dass keine OpenGL-Aufrufe auf dem geteilten *Memory Object* ausgeführt werden während durch OpenCL auf dieses zugegriffen wird. Anschließend muss zur Anzeige des Segmentierungsstandes die *OpenGL Textur* neu gezeichnet werden. Das eben beschriebene Vorgehen ist in Listing 4.6 wiedergegeben.

```
//Exklusiven Zugriff auf das geteilte Memory Object sichern,
//um Aufrufe von OpenCL und OpenGL zu synchronisieren.
gpuB->GetExclusiveOpenGLAccess();

//Anzuzeigendes Bild berechnen.
clEnqueueNDRangeKernel(*command_queue, k_getImage, 3, ..., global, ...);
//Zugriff auf das geteilte Memory Object starten.
clEnqueueAcquireGLObjects(..., mem_inputImage, ...);
//Anzuzeigende Bilddaten in das geteilte Memory Object kopieren.
clEnqueueCopyBufferToImage(*command_queue,mem_tmpPixelBuffer,*
    mem_inputImage, ...);
//Das geteilte Memory Object wieder freigeben.
clEnqueueReleaseGLObjects(..., mem_inputImage, ...);
//Alle noch ausstehenden OpenCL-Aufrufe ausführen.
clFinish(*command_queue);

//Exklusiven Zugriff auf das Memory Object wieder freigeben.
gpuB->ReleaseExclusiveOpenGLAccess();
//Textur neu zeichnen.
gpuB->RefreshScene();
```

Listing 4.6: Anzeige eines Iterationsschrittes.

Durch die Übergabe der entsprechenden Parameter wird auch hier, ähnlich wie in Kapitel 4.2.3.3, für jeden Bildpunkt der *OpenGL Textur* ein *Work Item* gestartet. Innerhalb des

Kernels wird nun mittels der Regionsmaske die anzuzeigende Iteration berechnet und die entsprechenden Bilddaten in das *Buffer Object* geschrieben. Nachdem der Inhalt des *Buffer Objects* in das geteilte *Memory Object* kopiert wurde, kann durch die Methode *RefreshScene*, welche durch die *GPUSegmentationBase* zur Verfügung gestellt wird, die *OpenGL Textur* neu gezeichnet werden.

4.2.4 Filter

Das Funktionsprinzip der *Filter*-Klasse basiert darauf, dass alle *Kernel*, die einen Filteralgorithmus implementieren, drei Argumente erwarten. Diese bestehen aus einem Buffer für das Eingangsbild, einem Buffer für das Ausgangsbild und einem Parameter, der die Größe der zu verwendenden Filtermaske festlegt. Daher ergibt sich für jeden Filter der gleiche Methodenkopf, wie er in der OpenCL-Implementierung der einzelnen Filter verwendet werden kann. Wiedergegeben ist dieser Methodenkopf in Listing 4.7.

```
1 //Genereller Methodenkopf für einen Filterkernel.
__kernel void filter(__global ushort *inputImage,
3                    __global ushort *filteredImage,
                     int kernelsize)
```

Listing 4.7: Genereller Methodenkopf wie er von einem Kernel erwartet wird, der einen Filter realisiert.

Da durch diese Definition jeder Filter sich nur noch in der Implementierung der Filteralgorithmen innerhalb des *Kernels* unterscheidet, kann vonseiten des *Hosts* jeder Filter durch die *Filter*-Klasse umgesetzt werden. Es muss lediglich durch die von der Klasse *GPUTool* zur Verfügung gestellten Methode *getProgram* der entsprechende *OpenCL Programmcode* geladen und kompiliert werden. Danach kann der Kernel für den Filteralgorithmus immer auf die gleiche Weise erzeugt werden. Zunächst wird der Kernel für den Filteralgorithmus erzeugt und anschließend wird diesem das zu filternde Bild, also die Arbeitskopie des geteilten *Memory Objects*, übergeben.

Des Weiteren ist der Ablauf der Filterung vonseiten des *Hosts* immer gleich. Er lässt sich in zwei Bereiche unterteilen. Bei dem ersten Bereich handelt es sich um die Durchführung des Filteralgorithmuses. Der dafür verantwortliche Programmcode ist in Listing 4.8 wiedergegeben. Der zweite Bereich ist für die Anzeige des Filterergebnisses zuständig.

Um die Filterung durchführen zu können, wird zunächst ein *Buffer Object* für das Ergebnis der Filterung erzeugt. Dieses und die zu verwendende Filtermaskengröße werden dann an den *Kernel*, welcher den Filteralgorithmus implementiert, übergeben.

```
//Buffer für das gefilterte Bild erstellen.
mem_filteredImage = clCreateBuffer(...);
//Buffer für das gefilterte Bild an den Kernel übergeben.
clSetKernelArg(k_filter, 1, sizeof(cl_mem), (void*)&mem_filteredImage);
//Filtermaskengröße an den Kernel übergeben.
clSetKernelArg(k_filter, 2, sizeof(cl_int), (void*)&param);
//Filterung durchführen.
clEnqueueNDRangeKernel(*command_queue,k_filter, 3, ..., global, ...);
```

Listing 4.8: Programmcode, der bei der Filterung des geteilten Memory Objects durchlaufen wird.

Mit *clEnqueueNDRangeKernel* wird der *Kernel* zur Ausführung in die *Command Queue* (*command_queue*) eingereiht, wobei festgelegt wird, dass für jeden Bildpunkt des zu filternden Bildes ein *Work Item* gestartet wird. Der für die Anzeige des Filterergebnisses verantwortliche Code ist in Listing 4.9 dargestellt.

```
//Exklusiven Zugriff auf das geteilte Memory Object sichern.
gpuB->GetExclusiveOpenGLAccess();
//Zugriff auf das geteilte Memory Object starten.
clEnqueueAcquireGLObjects(..., mem_inputImage, ...);
//Inhalt des gefilterten Bildes in das geteitle Memory Object
//kopieren.
clEnqueueCopyBufferToImage(*command_queue, mem_filteredImage, *
    mem_inputImage, ...);
//Zugriff auf das geteitle Memory Object beenden.
clEnqueueReleaseGLObjects(..., mem_inputImage, ...);
//Exklusiven Zugriff wieder freigeben.
gpuB->ReleaseExclusiveOpenGLAccess();
//Geteiltes Memory Object neu zeichnen.
gpuB->RefreshScene();
```

Listing 4.9: Programmcode zur Anzeige des gefilterten Bildes.

Hier wird ähnlich wie bei der Anzeige eines Iterationsschrittes in der *Regiongrow*-Klasse vorgegangen (erläutert in Kapitel 4.2.3.4). Um möglichen Fehlern vorzubeugen wird auch hier der exklusive Zugriff auf das geteilte *Memory Object* sichergestellt. Anschließend wird der Inhalt des gefilterten Bildes (*mem_filteredImage*) in das geteilte *Memory Object* (*mem_inputImage*) geschrieben. Um das Ergebnis der Filterung anzuzeigen, muss das *Memory Object* nach abgeschlossenem Kopiervorgang neu gezeichnet werden. Gleich wie in Kapitel 4.2.3.4) wird hierfür die entsprechende Methode der *GPUSegmentationBase* verwendet.

4.2.4.1 Gauß Filter

Bei der Umsetzung des Gauß Filters wurde sich dazu entschlossen von dem allgemeinen Ansatz für die Implementierung von verschiedenen Filteralgorithmen, wie er in Kapitel

4.2.4 vorgestellt wurde, abzuweichen. Zwar wäre es möglich einen Gauß Filter so umzusetzen, dass er sich an die Vorgaben in Bezug auf den Methodenkopf des *Kernels*, welcher den Filteralgorithmus implementiert, hält, jedoch müsste dann bei der Filterung jedes ausgeführte *Work Item* die Berechnung der Gewichte in der Filtermaske separat durchführen. Dies führt zu einem enormen unnötigen Rechenaufwand. Daher wurde sich dazu entschlossen, die Gewichte der Filtermaske lediglich einmal im *Host-Programm* zu berechnen und diese als weiteres Argument an den, für die Filterung verantwortlichen Kernel zu übergeben. Wie die berechneten Gewichte an den *Kernel* für die Filterung übergeben werden ist in Listing 4.10 abgebildet.

```
1  //Buffer für die zu verwendende Filtermaske erstellen.
   mem_filterMask = clCreateBuffer(...,
3                                  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                   ...,
5                                  filterMask, ...);
   //Filtermaske an den Kernel, welcher den
7  //Filteralgorithmus implementiert, übergeben.
   clSetKernelArg(k_filter, 3, sizeof(cl_mem), (void*)&mem_filterMask);
9  Filter::filterImage();
```

Listing 4.10: Erzeugen eines Buffer Objects, um die Gewichte der zu verwendenden Filtermaske an den für die Filterung verantwortlichen Kernel übergeben zu können.

Nachdem die Gewichte an den *Kernel* übergeben wurden kann der restliche Ablauf der Filterung analog wie in Kapitel 4.2.4 durchgeführt werden. Daher muss anschließend nur die Methode *filterImage* der Superklasse *Filter* aufgerufen werden.

4.2.5 Snakeverfahren Prototyp

Wie bereits in Kapitel 4.1 erwähnt, wurde das Snakeverfahren im Verlauf dieser Diplomarbeit nicht über einen prototypischen Stand entwickelt. Jedoch kann durch den Prototypen die Realisierbarkeit eines Snakeverfahrens mit Hilfe von OpenCL gezeigt werden. Daher wird hier auf grundlegende Elemente der Implementierung des Prototypen in Bezug auf OpenCL eingegangen. In Abbildung 4.6 ist der generelle Programmablauf bei der Segmentierung durch den Snakeprototypen wiedergegeben. Während das Bestimmen der initialen Kurve sowie das Einstellen der Gewichtung von interner und externer Energie durch den Anwender erfolgt, fassen die beiden Aktivitäten »GVF-Field berechnen« und »Iterationen zur Minimierung der Snakeenergie starten« die Hauptbestandteile der Implementierung zusammen. Diese wurden mit Hilfe von OpenCL realisiert und werden in den Kapiteln 4.2.5.1 und 4.2.5.2 näher erläutert.

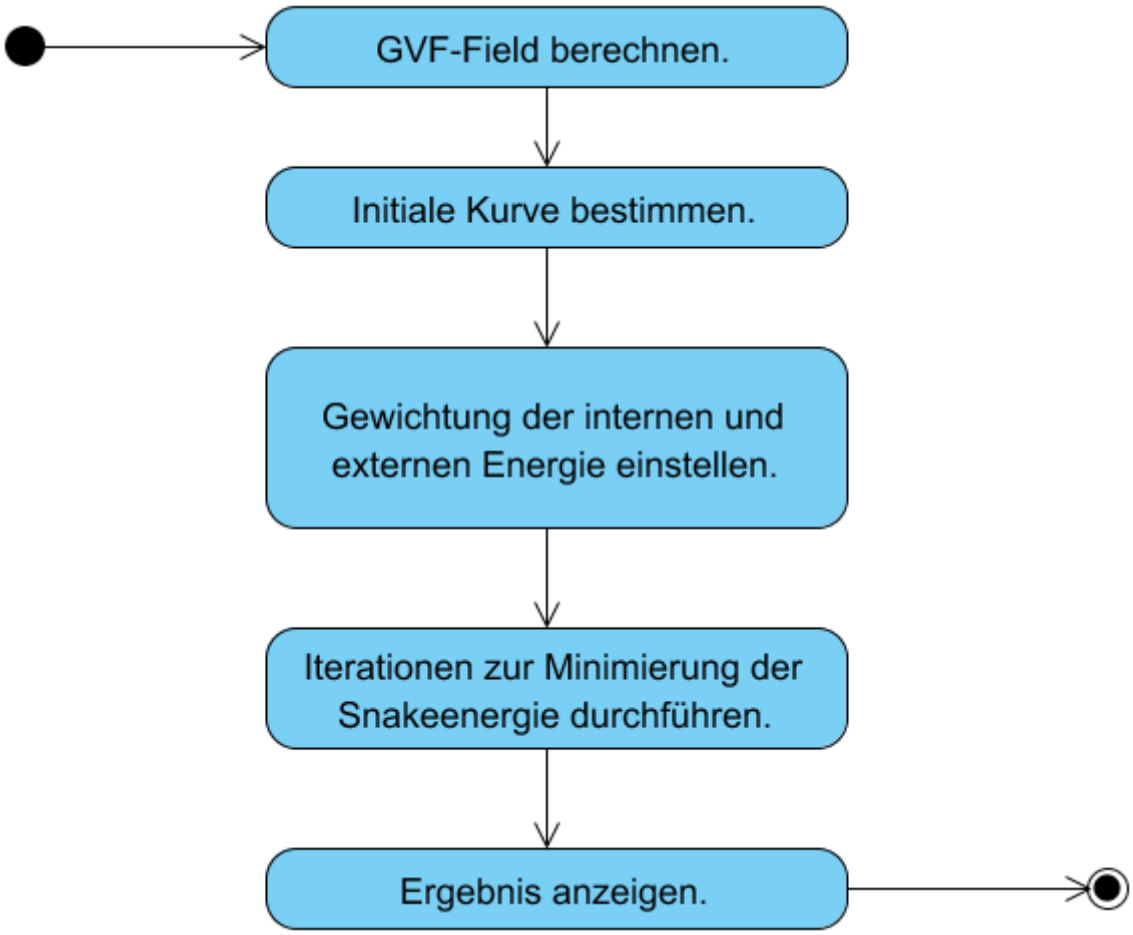

Abbildung 4.6: Programmablauf bei der Segmentierung mittels Snakes.

4.2.5.1 Berechnung des GVF-Feldes

Wie bereits in den Grundlagen erwähnt, hängt die externe Energie einer Snake nur von dem zu segmentierenden Bild ab. Daher muss diese nur für jeden Bildpunkt einmal berechnet werden. Im Folgenden wird beschrieben wie die externe Energie durch die Berechnung eines *GVF-Feldes* für jeden Bildpunkt bestimmt wird.
Um ein *GVF-Feld* erstellen zu können müssen zunächst die Betragswerte der Gradienten des zu segmentierenden Bildes berechnet werden. Im Prototypen werden diese durch den *Kernel k_getGradAbsolute* berechnet. Da der entwickelte Snakeprototyp nur auf zweidimensionalen Bildern operiert, wird hier ein zweidimensionaler *Index-Raum* aufgespannt, so dass für jeden Bildpunkt ein *Work Item* gestartet wird. Innerhalb der *Work Items* wird dann für den jeweiligen Bildpunkt durch die Verwendung eines Sobel-Operators die Ableitung in X- und Y-Richtung bestimmt. Mit Hilfe der beiden Ableitungen wird dann der Betragswert des Gradienten berechnet.
Nach dem die Beträge der Gradienten berechnet wurden, wird anhand dieser das initiale *GVF-Feld* erstellt. Dieses dient als Iterationsstart bei der Berechnung des *GVF-Feldes*. Auch hier wird durch die entsprechenden Parameter für jeden Bildpunkt ein *Work Item*

gestartet. Im Anschluss werden die diskreten Werte der Funktionen $b(x, y)$, $c_1(x, y)$ und $c_2(x, y)$, wie sie Ende Kapitel 2.3.2.4 angegeben sind, berechnet. Diese werden ebenfalls durch einen *Kernel* ermittelt. Nach dem alle bisher beschriebenen *Kernel* ausgeführt wurden, kann die iterative Berechnung des *GVF-Feldes* durchgeführt werden. Generell ist zu beachten, dass durch jede durchgeführte Iteration die Vektoren der einzelnen Kanten weiter gestreut werden, also einen weiterreichenden Einfluss erhalten. Dem entsprechend können in einem Bild, in dem viele schwache und nur wenige starke Kanten enthalten sind, bei der Ausführung zu vieler Iterationen die Vektoren der schwachen Kanten von denen der starken überlagert werden. Es ist also vom Bild abhängig wie viele Iterationen bei der Berechnung des *GVF-Feldes* tatsächlich durchlaufen werden sollten. Bei der Durchführung der einzelnen Iterationen werden zwei *Buffer Objects* verwendet, um die jeweiligen Iterationsschritte bestimmen zu können. Sie dienen als Eingangs- und Ausgangswerte für die Iterationen, wobei die Ausgangswerte der vorangegangenen Iteration die Eingangswerte für die folgende Iteration bilden.

Mit Abschluss der For-Schleife ist das finale GVF-Feld berechnet und kann bei der Minimierung der Snakeenergie als externe Energie verwendet werden.

4.2.5.2 Durchführung einer Iteration

Um die Funktionsweise des *Kernels*, welcher die Berechnung einer Iteration für die Minimierung der Snakeenergie durchführt, zu erläutern, wird auf die wichtigsten Argumente dieses *Kernels* eingegangen. Der Methodenkopf dieses Kernels ist in Listing 4.11 wiedergegeben.

```
1  __kernel void iteration(//Kurve, welche die Snake repräsentiert.
                           __global float2* curve,
3                          //Array, dass festlegt bei welchen Punkten
                           //der Kurve es sich um einen Fixpunkt handelt.
5                          __global char* fixpoints,
                           //Das GVF-Feld.
7                          __global float2* gvf,
                           //Gewichtung der internen Energie.
9                          float weightInternal,
                           //Gewichtung der externen Energie.
11                         float weightExternal, ...)
```

Listing 4.11: Methodenkopf des Kernels, der die Berechnung einer Iteration bei der Minimierung der Snakeenergie durchführt.

Das erste Argument beinhaltet die Punkte der Kurve, für die im Zuge der Minimierung neue Koordinaten ermittelt werden sollen. Durch das zweite Argument wird das Konzept von Fixpunkten, wie es in Kapitel 3.3.1.2 erläutert wurde, realisiert. Hiermit wird es dem *Kernel* möglich zu ermitteln, ob es sich bei einem Punkt der Kurve um einen Fixpunkt handelt. Sollte dies der Fall sein, wird für diesen Punkt keine neue Position berechnet. Als drittes Argument wird das *GVF-Feld* des zu segmentierenden Bildes erwartet. Dieses wird bei der Berechnung als externe Energie verwendet. Über das vierte und fünfte Argument

wird dann noch die Gewichtung der internen und externen Energie festgelegt. Sind alle Argumente an den *Kernel* übergeben, kann die Berechnung einer Iteration ausgeführt werden.

Da die Punkte der Kurve in einem eindimensionalen Array gehalten werden, wird auch bei der Ausführung des *Kernels* (*k_iteration*) ein eindimensionaler *Index-Raum* aufgespannt, so dass für jeden Punkt der Kurve genau ein *Work Item* ausgeführt wird. Dies entspricht der Erkenntnis aus Kapitel 2.3.2.3, dass die Minimierung der gesamten Snakeenergie durch die Minimierung der Energie in den einzelnen Punkten der Snake berechnet werden kann. Zur Anzeige der neu berechneten Kurve muss nach abgeschlossener Iteration die Kurve aus dem Grafikkartenspeicher in den Hauptspeicher kopiert werden und auf der Anwendungsoberfläche neu gezeichnet werden.

5 Ergebnisse

In diesem Kapitel werden die Verfahren, der in dieser Diplomarbeit entstandenen Segmentierungskomponente, dargestellt. Dazu werden verschiedene Anwendungsbeispiele erläutert. Des Weiteren wird auf die Performace bzw. Effektivität der entwickelten Verfahren eingegangen.

5.1 Anwendungsbeispiele

In diesem Kapitel wird die Anwendung der entwickelten Verfahren anhand verschiedener Anwendungsbeispiele demonstriert. Hierdurch soll die Nutzbarkeit der Verfahren erläutert werden. In Kapitel 5.2 werden die hier durchgeführten Anwendungsbeispiele wieder aufgegriffen und im Hinblick auf ihre Effektivität und Performance näher erläutert.

5.1.1 Segmentierung des Cortex

In diesem Kapitel wird dargestellt, wie durch die vorhandenen Segmentierungsmöglichkeiten der entwickelten Segmentierungskomponente der Cortex erfasst werden kann. In der Praxis kann ein solches Segmentierungsergebnis vor allem zur räumlichen Orientierung verwendet werden. Außerdem kann so die konkrete Lage der Gyri ermittelt werden, um z.B. Aussagen über die genaue Lage von motorischen Arealen machen zu können. Auch kann ein solches Segmentierungsergebnis bei der Planung tiefer gehender Eingriffe verwendet werden, um zu bestimmen zwischen welchen Gyri der Eingriff erfolgen soll. In Abbildung 5.1 ist der verwendete Datensatz wiedergegeben wie er sich dem Anwender vor der Segmentierung präsentiert. Dabei handelt es sich um eine T1-gewichtete Magnetresonanztomographie. Sie besteht aus 192 aufgenommenen Schichten, die jeweils ein Millimeter dick sind. Jede Schicht besteht aus 320*270 Pixeln, wobei die Kantenlänge jedes Pixels in der Realität einem Millimeter entspricht.

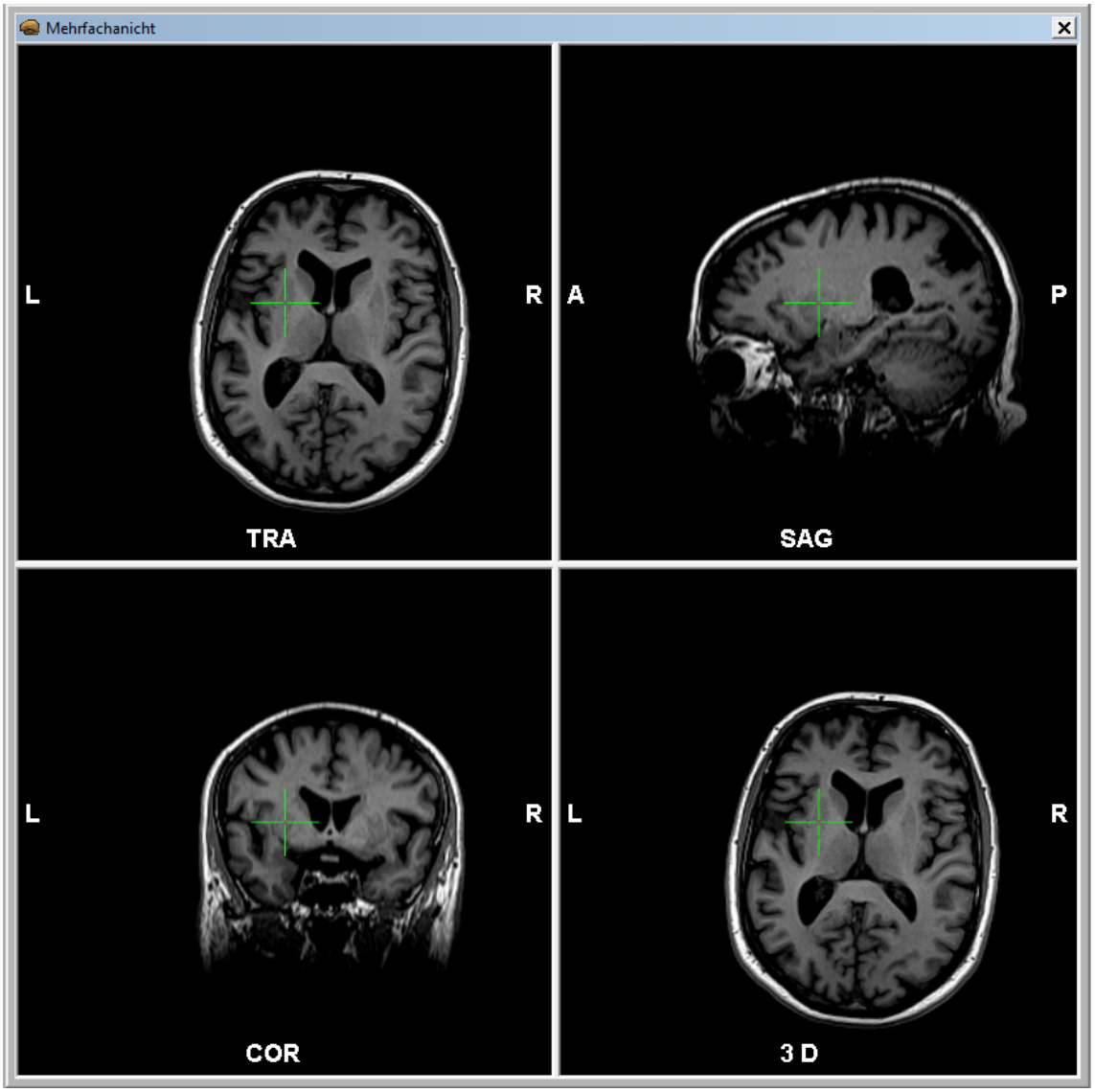

Abbildung 5.1: Zu sehen sind die verschiedenen 2D-Schichten des vorliegenden Datensatzes vor der Segmentierung.

Der Cortex besteht aus einer zwei bis 5 mm dicke Schicht, die das Großhirn umfasst und ist somit ein Teil der grauen Substanz. Für eine möglichst korrekte Segmentierung müsste also nur diese Ummantelung erfasst werden. Da jedoch nur die Oberfläche des Cortex von Interesse ist, stellt es kein Problem dar, wenn weitere Areale des Großhirns erfasst werden, wie z.B. die weiße Substanz. Dies ist insofern hilfreich, da es so möglich ist, sich vom Inneren des Hirns nach und nach an eine optimale Segmentierung heranzutasten. Dem entsprechend bietet es sich an die ersten Seedpoints innerhalb der weißen Substanz zu setzen. Die für die Segmentierung initial gesetzten Seedpoints sind in Abbildung 5.2 wiedergegeben. Nach dem die initialen Seedpoints gesetzt wurden, werden dem Nutzer die ermittelten Schwellwerte für das Homogenitätskriterium angezeigt (siehe Abbildung 5.3). Der Anwender hätte nun die Möglichkeit die ermittelten Schwellwerte weiter anzupassen. Um sich einen Eindruck verschaffen zu können, wie gut die gewünschte Segmentierung

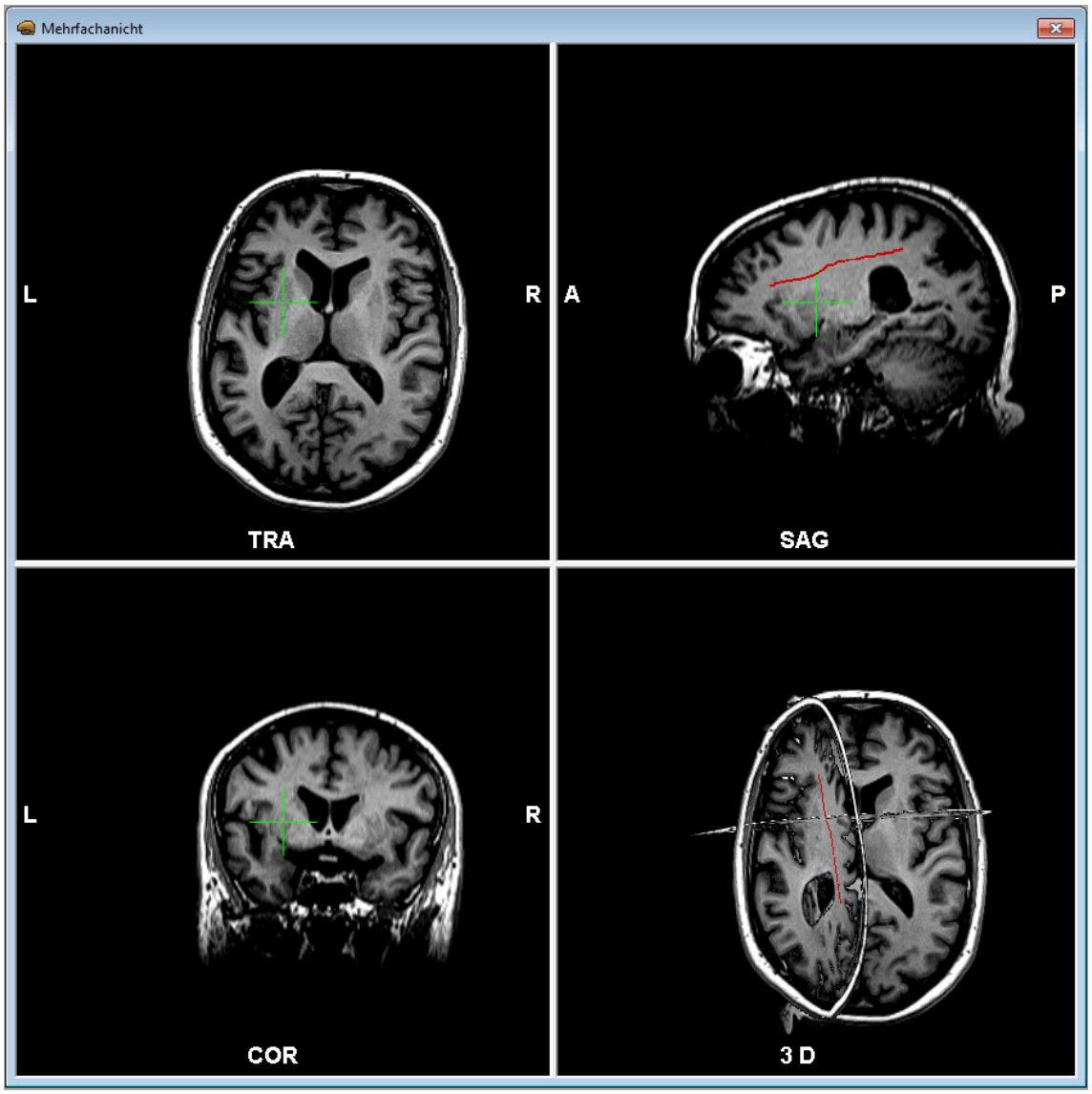

Abbildung 5.2: In der Abbildung sind durch die rote Einfärbung die initial gesetzten Seedpoints zu sehen.

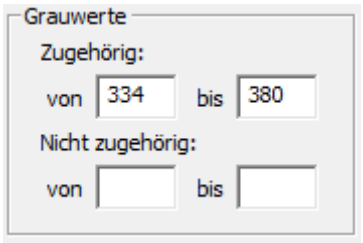

Abbildung 5.3: Wiedergeben sind die Schwellwerte, wie sie nach dem Hinzufügen der initialen Seedpoints durch die Segmentierungskomponente berechnet wurden.

durch die gesetzten Seedpoints erfasst wurde, bietet es sich jedoch an, zunächst den Regiongrowalgorithmus zu starten. Während der Ausführung des Algorithmus' wird dann jeder durchlaufene Iterationsschritt angezeigt. In dem hier dargestellten Anwendungsbeispiel umfasste dies 42 Iterationen. Das Zwischenergebnis, das so erzeugt wurde, ist in Abbildung 5.4 rechts unten zu sehen. Zusätzlich zu dem erzeugten Zwischenergebnis werden in den

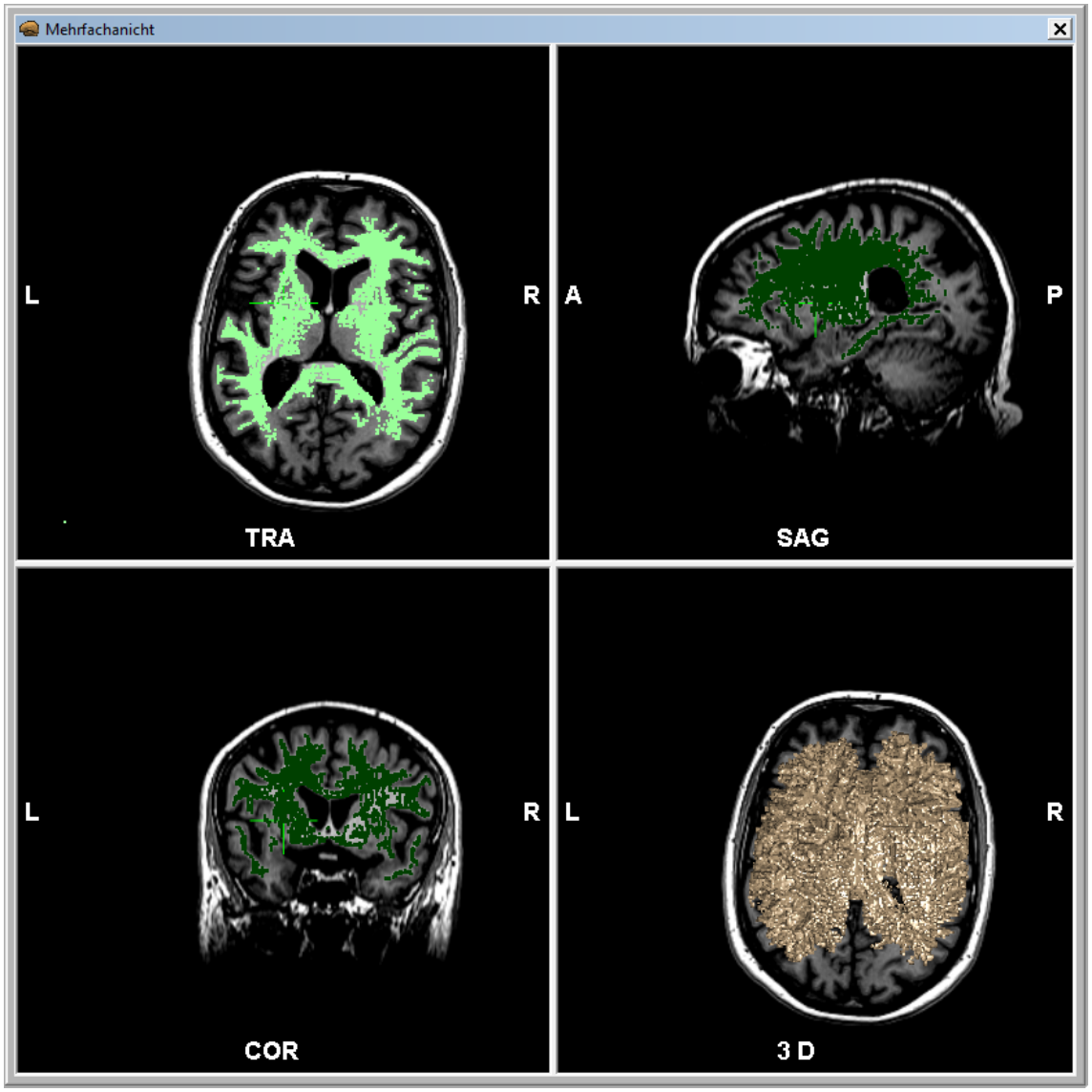

Abbildung 5.4: Zwischenergebnis wie es durch die berechneten Schwellwerte mit Hilfe der initalen Seedpoints erzeugt wurde. Insgesamt wurden 42 Iterationen benötigt.

2D-Ansichten die durch die Segmentierung erfassten Bildpunkte in der jeweils dargestellten Schicht angezeigt. Zu sehen ist dies in Abbildung 5.4 durch die grün eingefärbten Bereiche. So ist leicht zu erkennen, dass der momentane Segmentierungsstand noch nicht ausreichend für das gewünschte Endergebnis ist. Welche Bereiche bei der Segmentierung noch fehlen lässt sich dank der Einfärbung der Bildpunkte in den 2D-Ansichten leicht erkennen. Dem Anwender stehen nun vor allem zwei Möglichkeiten offen um die Segmentierung zu

verfeinern. Die erste Möglichkeit besteht darin die ermittelten Schwellwerte anzupassen. Dieses Vorgehen eignet sich besonders um Feinabstimmungen bei einer nahezu vollständigen Segmentierung vorzunehmen und ist daher bei dem momentanen Segmentierungsstand weniger interessant. Die zweite Möglichkeit besteht darin weitere Seedpoints zu setzen. Für dieses Vorgehen wurde sich bei dem durchgeführten Anwendungsbeispiel entschieden. Die zusätzlich hinzugefügten Seedpoints sind in Abbildung 5.5 zu sehen. Neben diesen Seed-

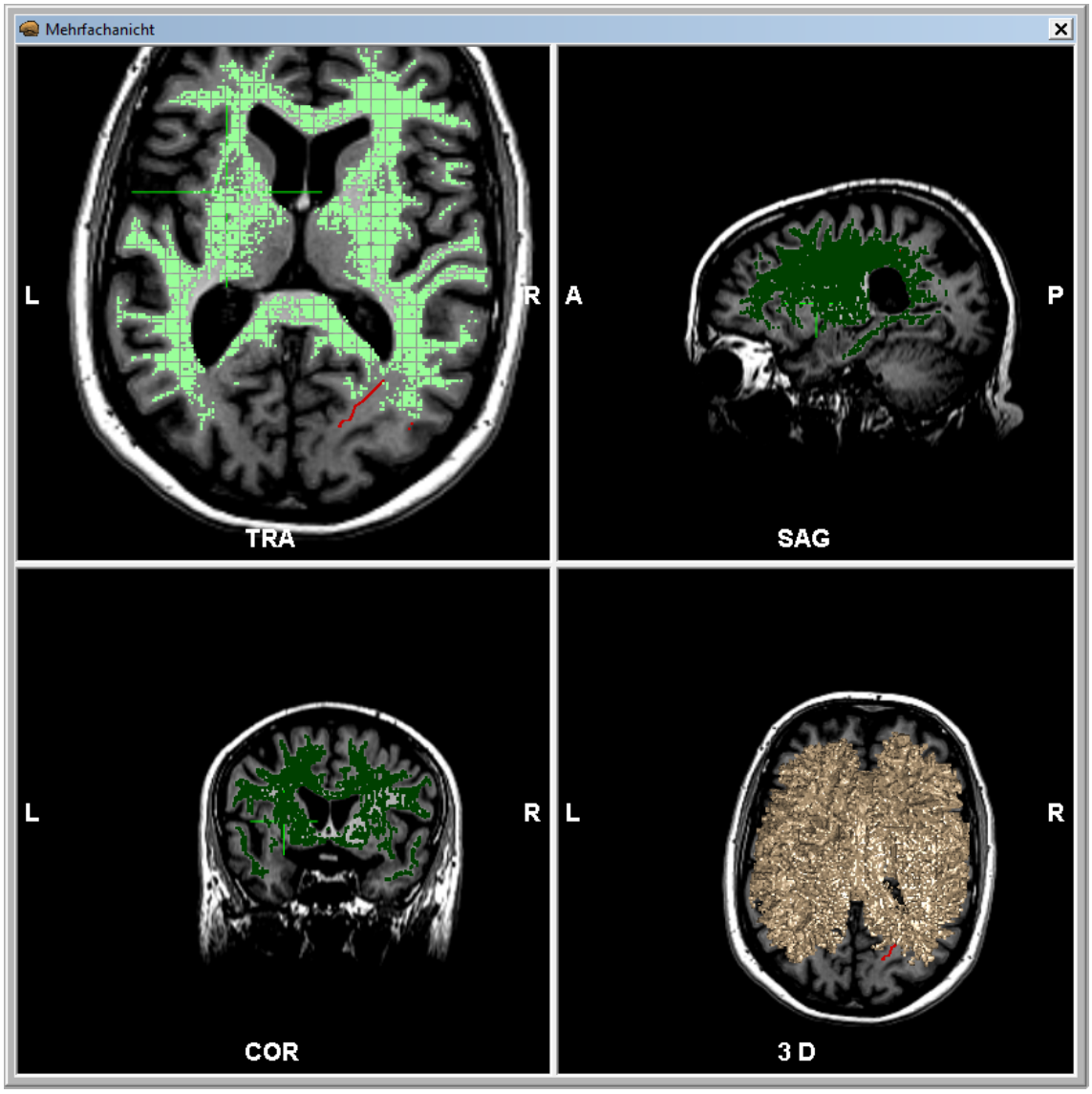

Abbildung 5.5: Mit Rot dargestellt sind weitere hinzugefügte Seedpoints, um das Segmentierungsergebnis zu verbessern.

points war es nötig noch weitere Seedpoints zu bestimmen und anschließend geringfügige Änderungen an den verwendeten Schwellwerten vorzunehmen. Hierdurch ist es möglich, ein finales Ergebnis wie in Abbildung 5.6 zu erzeugen. Rot umrandet ist dabei ein Bereich, der bei der Segmentierung durch ein vorhandenes Ödem gestört wurde. Dieses Ödem stellt sich in den verwendeten Bilddaten durch geringe Grauwerte dar (zu sehen in Abbildung

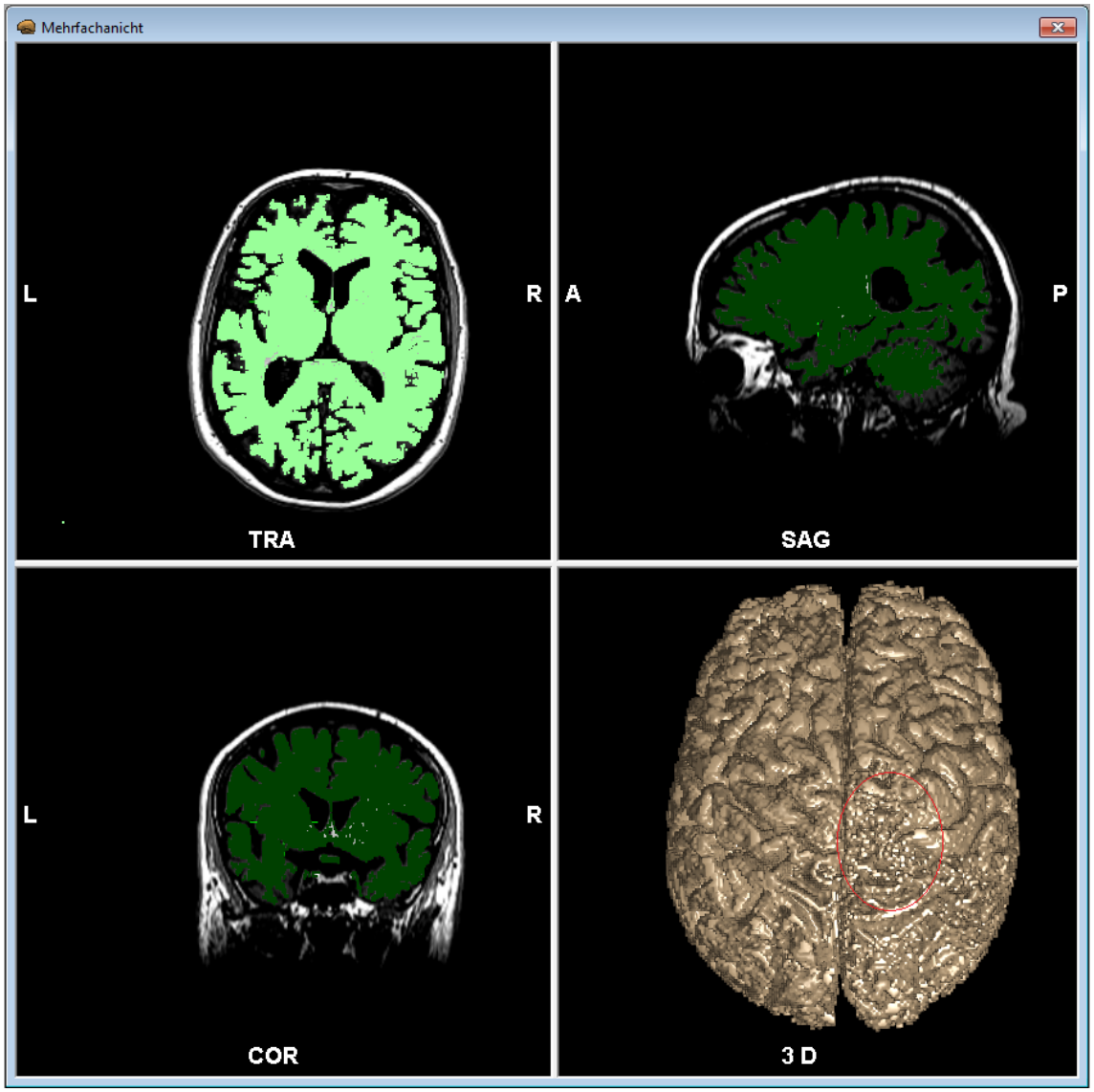

Abbildung 5.6: Finales Segmentierungsergebnis wie es durch die zusätzlichen Seedpoints und Anpassung der Schwellwerte erzielt wurde. Rot markiert ist ein Bereich, der durch ein in den Bilddaten vorhandenes Ödem von dem Segmentierungsalgorithmus nicht erfasst wurde.

5.7). Um auch diesen Bereich vollständig zu erfassen, müsste der untere Schwellwert sehr weit herabgesetzt werden. Hierdurch würde der Regiongrowalgorithmus jedoch an mehreren Stellen in Strukturen auslaufen, die nicht zum gewünschten Segmentierungsergebnis gehören. Des Weiteren werden bei der Segmentierung große Grauwertübergänge zwischen

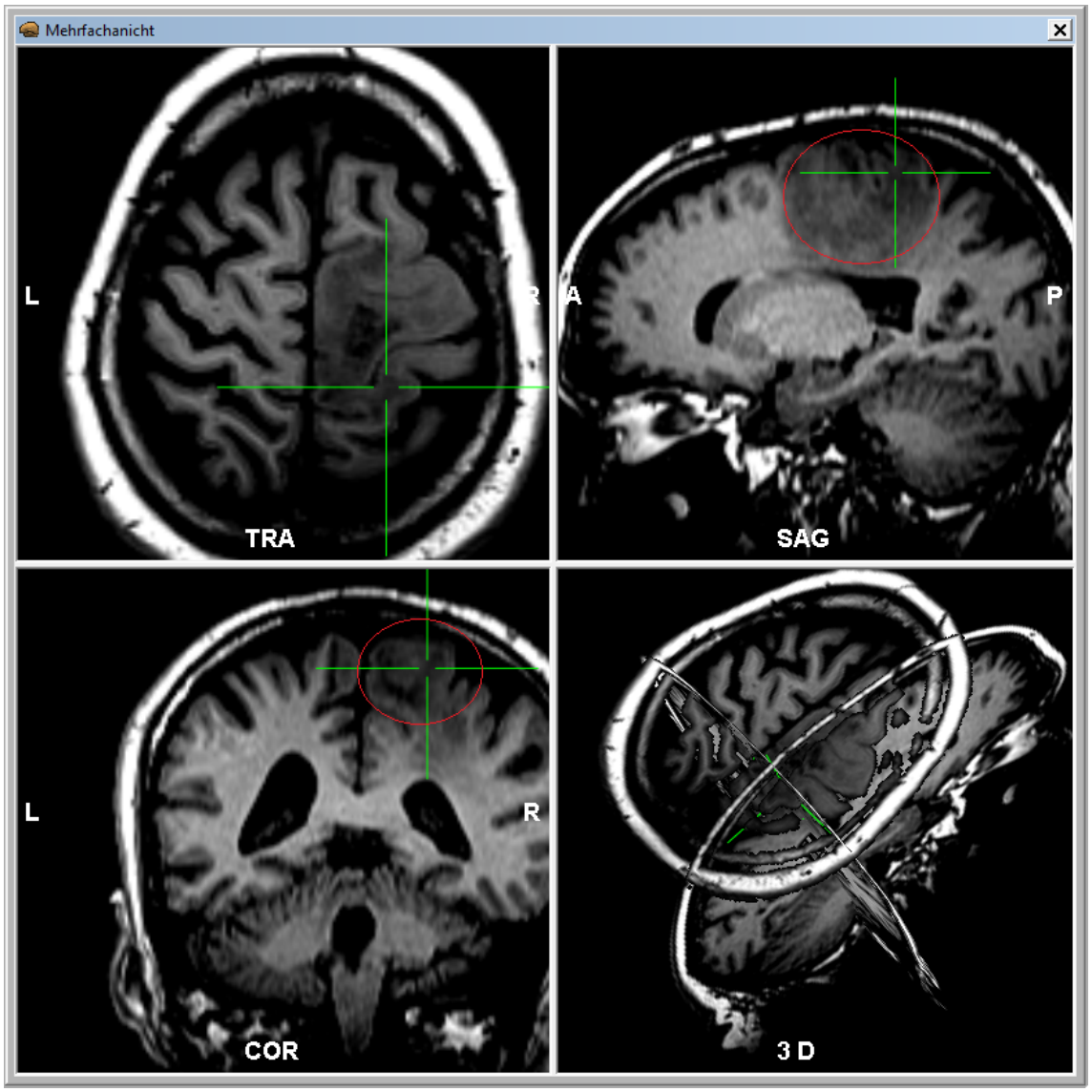

Abbildung 5.7: Zu sehen ist ein Ödem, welches bei der Segmentierung des Cortex direkten Einfluss auf das Ergebnis hat.

der erfassten Region und dem Hintergrund erzeugt. Dies hat keinen Einfluss auf die Vollständigkeit des Segmentierungsergebnisses führt jedoch zu einer grobkörnigen Darstellung. Um weichere Grauverläufe zwischen Region und Hintergrund zu generieren, können die bereitgestellten Segmentierungsfilter verwendet werden. Im Anwendungsbeispiel wurde dazu ein Gauß Filter mit einer $3 * 3 * 3$ Filtermaske auf das finale Segmentierungsergebnis angewandt. Die hierdurch generierte Darstellung ist in Abbildung 5.8 wiedergegeben. Zu beachten ist, dass das Anwenden von Filtern immer zu einem Informationsverlust führt.

So kann mit der hier durchgeführten Filterung zwar eine gleichmäßigere Darstellung erzeugt werden, jedoch wird dabei die Abgrenzung zwischen den einzelnen Gyri verwischt. Ob ein Filter zur Glättung verwendet werden soll liegt also im Ermessen des Anwenders und den Ansprüchen, die er an die Segmentierung stellt.

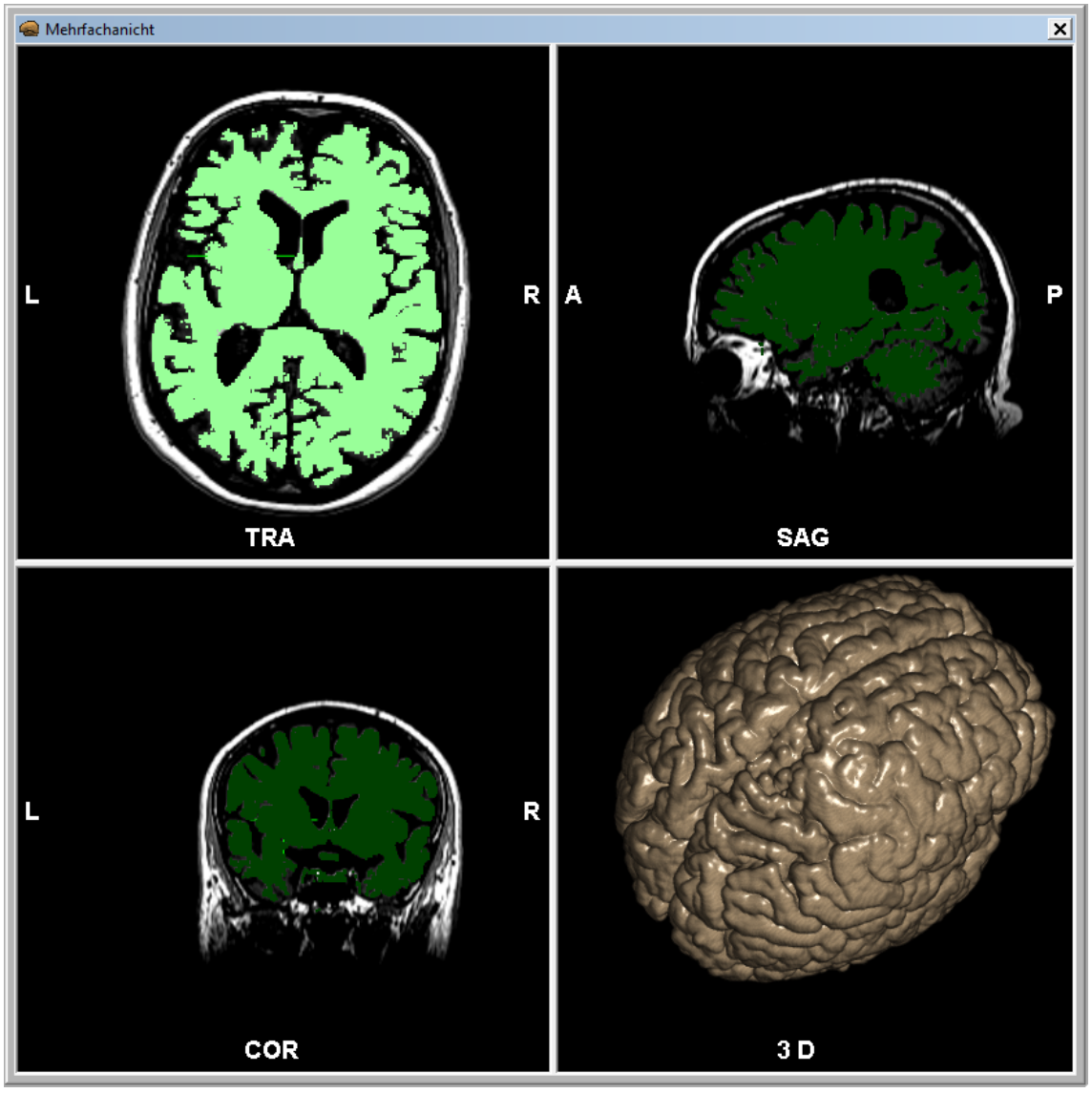

Abbildung 5.8: Mit einem Gauß Filter gefiltertes Segmentierungsergebnis um Kanten zu glätten und so eine einheitlichere Darstellung zu erzeugen.

5.1.2 Weiße Substanz

In dem folgenden Anwendungsbeispiel soll die weiße Substanz des Hirns, im gleichen Datensatz wie in Kapitel 5.1.1, erfasst werden. Hierbei handelt es sich um die Leitungsbahnen der Nervenzellen des Gehirns. Praktische Relevanz hat dies, um z.B. erkennen zu können, ob durch einen Tumor die Übertragung von Signalen eines funktionalen Areals gestört werden. Darüber hinaus kann sich ein Überblick über die Faserverläufe im Hirn verschafft

werden. Die Segmentierung der weißen Substanz kann mit Hilfe des Regiongrowalgorithmus sehr schnell durchgeführt werden, da sie von der grauen Substanz umgeben ist, die sich in den Bilddaten durch geringere Grauwerte auszeichnet. Dies macht ein Auslaufen des Algorithmus' nahezu unmöglich. In Abbildung 5.9 sind die Seedpoints zu sehen, die für eine initiale Segmentierung gewählt wurden. Die hierdurch ermittelten Schwellwerte

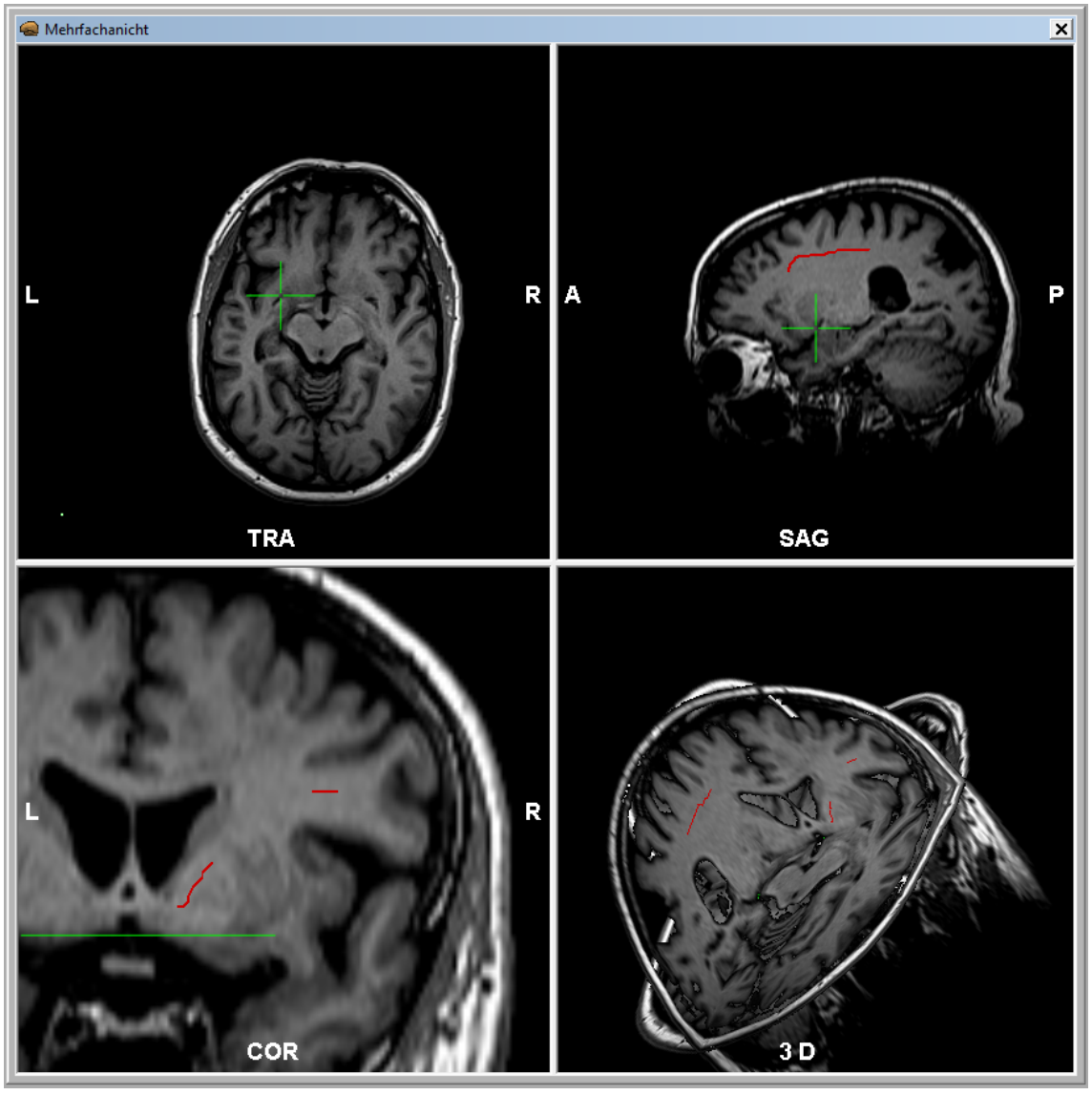

Abbildung 5.9: Initial gesetzte Seedpoints für die Segmentierung der weißen Substanz.

für das Homogenitätskriterium umfassten die Werte 300 (untere Schwelle) und 379 (obere Schwelle). Nach dem diese Werte auf 290 und 400 angepasst wurden, konnte eine Segmentierung entsprechend dem gesetzten Ziel erzeugt werden. Dieses ist in Abbildung 5.10 wiedergegeben.

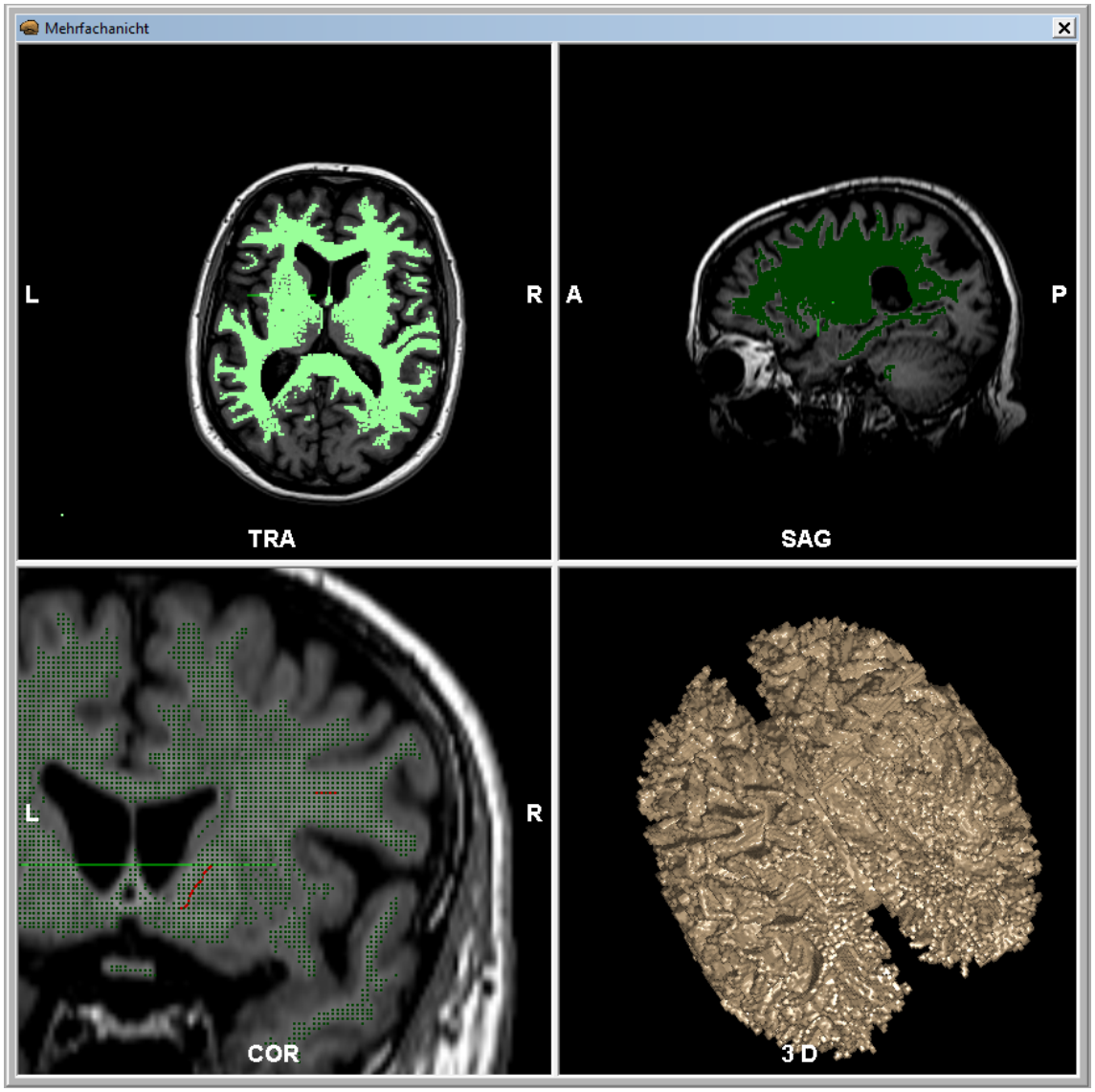

Abbildung 5.10: Segmentierungsergebnis der weißen Substanz.

5.1.3 Segmentierung der lateralen Ventrikel

Der in dem folgenden Anwendungsbeispiel verwendete Datensatz wurde von einem Tumorpatienten mittels Magnetresonanztomographie (T1-gewichtet) aufgenommen. Er besteht aus 256 Schichtaufnahmen mit einer Dicke von 1,3 mm. Jede Schicht besteht dabei aus 256*256 Pixel mit einer Kantenlänge von 1*1 mm. Um zu sehen welche Auswirkungen der Tumor auf die Morphologie der lateralen Ventrikel hat, wurde hier das Ziel gesetzt diese zu segmentieren. Dazu wurden zunächst mehrere Seedpoints gesetzt, um die Ventrikel möglichst schnell vollständig erfassen zu können. Zu sehen sind diese in der Abbildung 5.11. Anschließend wurde der Regiongrowalgorithmus gestartet. Durch das kontinuierlichen vi-

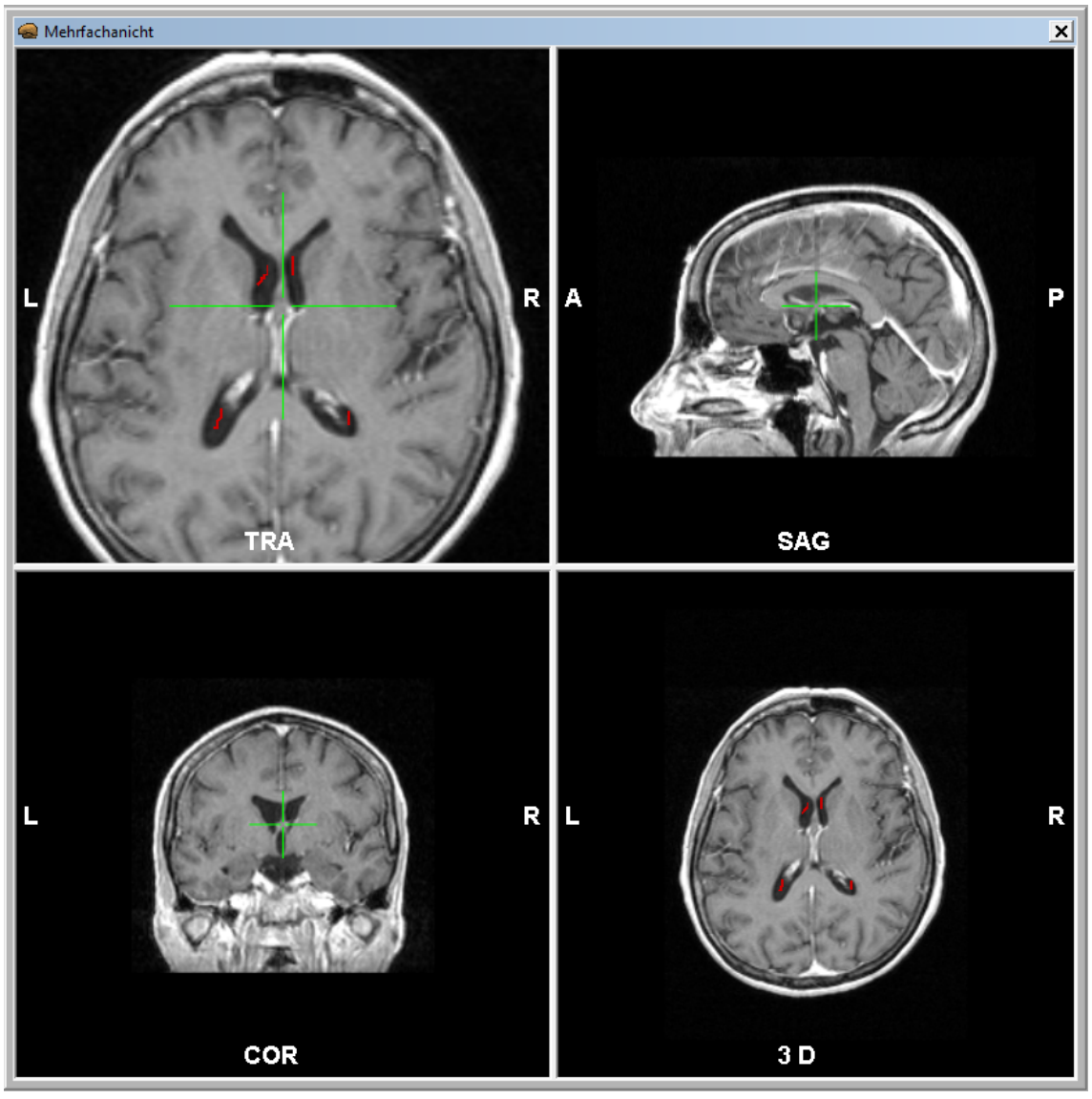

Abbildung 5.11: Zu sehen sind die Seedpoints wie sie initial gesetzt wurden, um die lateralen Ventrikel zu segmentieren.

suellen Feedback an den Nutzer, also das Anzeigen der einzelnen Iterationsschritte, war es möglich frühzeitig zu erkennen, dass der Algorithmus in unerwünschte Strukturen auslief.

Daraufhin wurde die laufende Segmentierung angehalten und zu einem Iterationsschritt gewechselt, in dem sich der Ursprung für die fehlerhafte Segmentierung gut abzeichnete. Dieser Iterationsschritt ist in Abbildung 5.12 wiedergegeben. Rot umrandet ist hierbei der durch die Segmentierung erfasste, jedoch unerwünschte Bereich des momentanen Segmentierungsstandes. Da es sich um einen kleine Bereich handelt, in dem der Algorithmus ausläuft, bietet es sich an, den Regiongrowalgorithmus durch das Setzen einer Grenze am Auslaufen zu hindern. Um dies zu bewerkstelligen stehen dem Anwender mehrere Möglichkeiten zur Verfügung. Generell ist es möglich entweder direkt in der 3D-Darstellung Grenzen einzufügen oder diese in den 2D-Ansichten zu setzen. In den 2D-Ansichten können so z.B. Ebenen eingezeichnet werden, welche dann als Grenze verwendet werden. Sie eignen sich vor allem um auch größere Bereiche abzugrenzen. Jedoch ist es aufwändi-

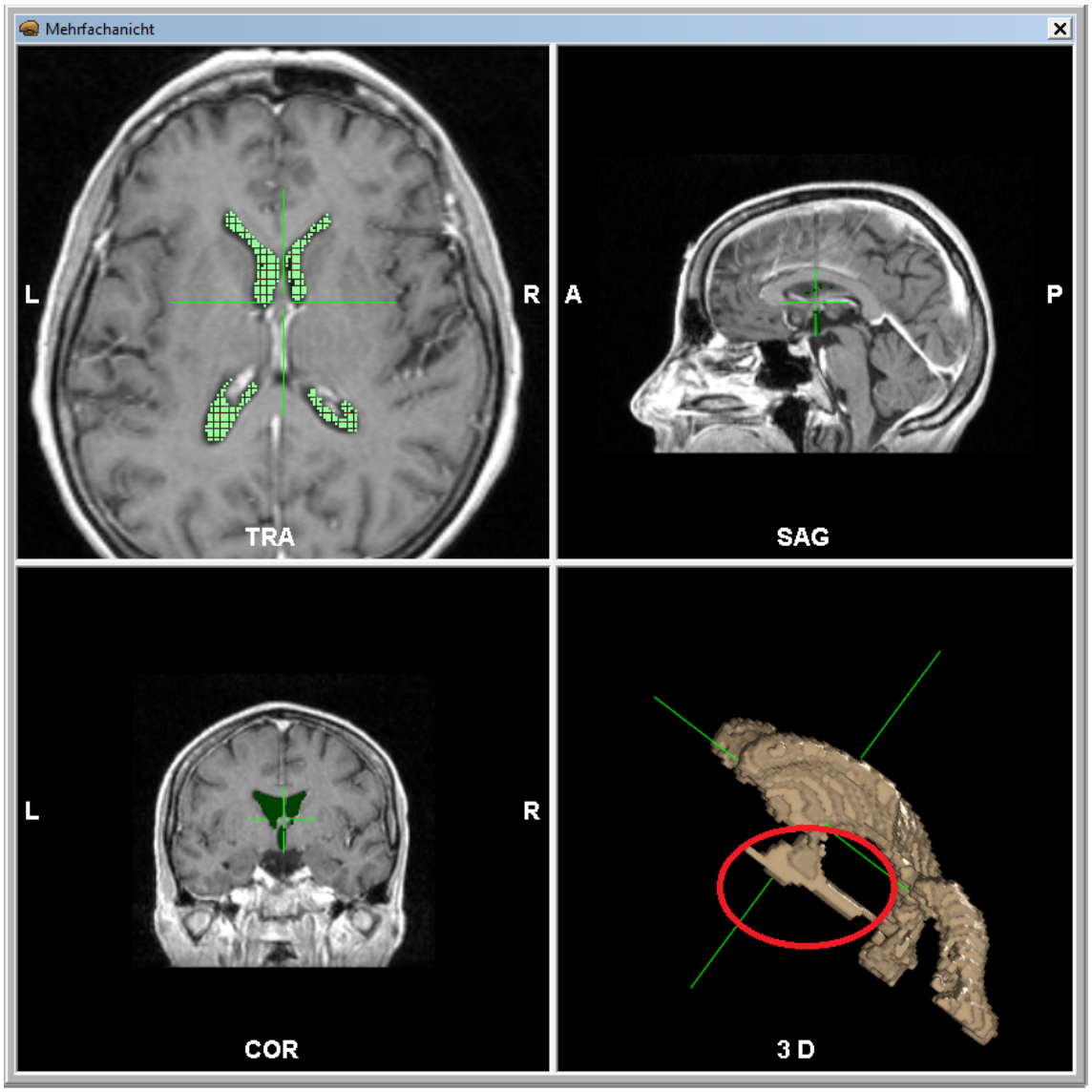

Abbildung 5.12: Segmentierungsstand in dem zu sehen ist, an welcher Stelle der Regiongrowalgorithmus in unerwünschte Strukturen ausläuft.

ger diese zu nutzen, da zunächst eine geeignete 2D-Schicht ermittelt werden muss, in die dann die Grenze eingefügt wird. Im Anwendungsbeispiel wurde daher die Grenze direkt in der 3D-Darstellung gesetzt. Eine so hinzugefügte Grenze stellt sich als Kugel dar. Der Durchmesser der Kugel kann nach dem Hinzufügen noch so angepasst werden, dass der gewünschte Bereich abgedeckt ist. Das Segmentierungsergebnis, welches mit der verwendeten Grenze entsteht ist in Abbildung 5.13 zu sehen. Nach dem die Schwellwerte weiter

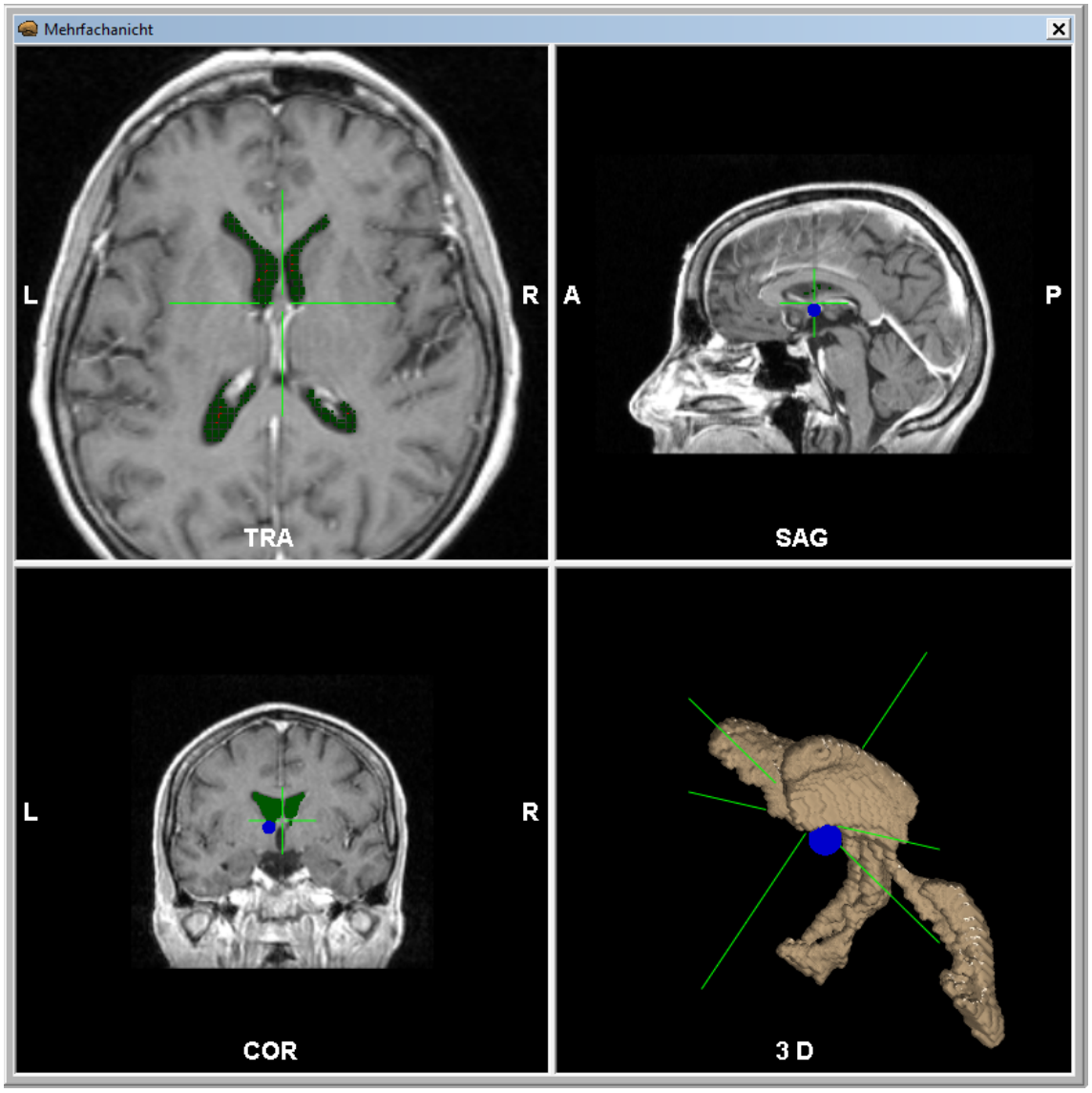

Abbildung 5.13: Das Segmentierungsergebnis wie es unter Verwendung einer durch den Anwender gesetzten Grenze erzielt werden konnte.

angepasst wurden, konnten die lateralen Ventrikel so erfasst werden wie sie in Abbildung 5.14 zu sehen sind. Rot umrandet ist oben links der Querschnitt des vorhandenen Tumors. Unten rechts wurde durch die rote Umrandung der Bereich markiert, in dem der Tumor die Form der Ventrikel direkt beeinflusst hat.

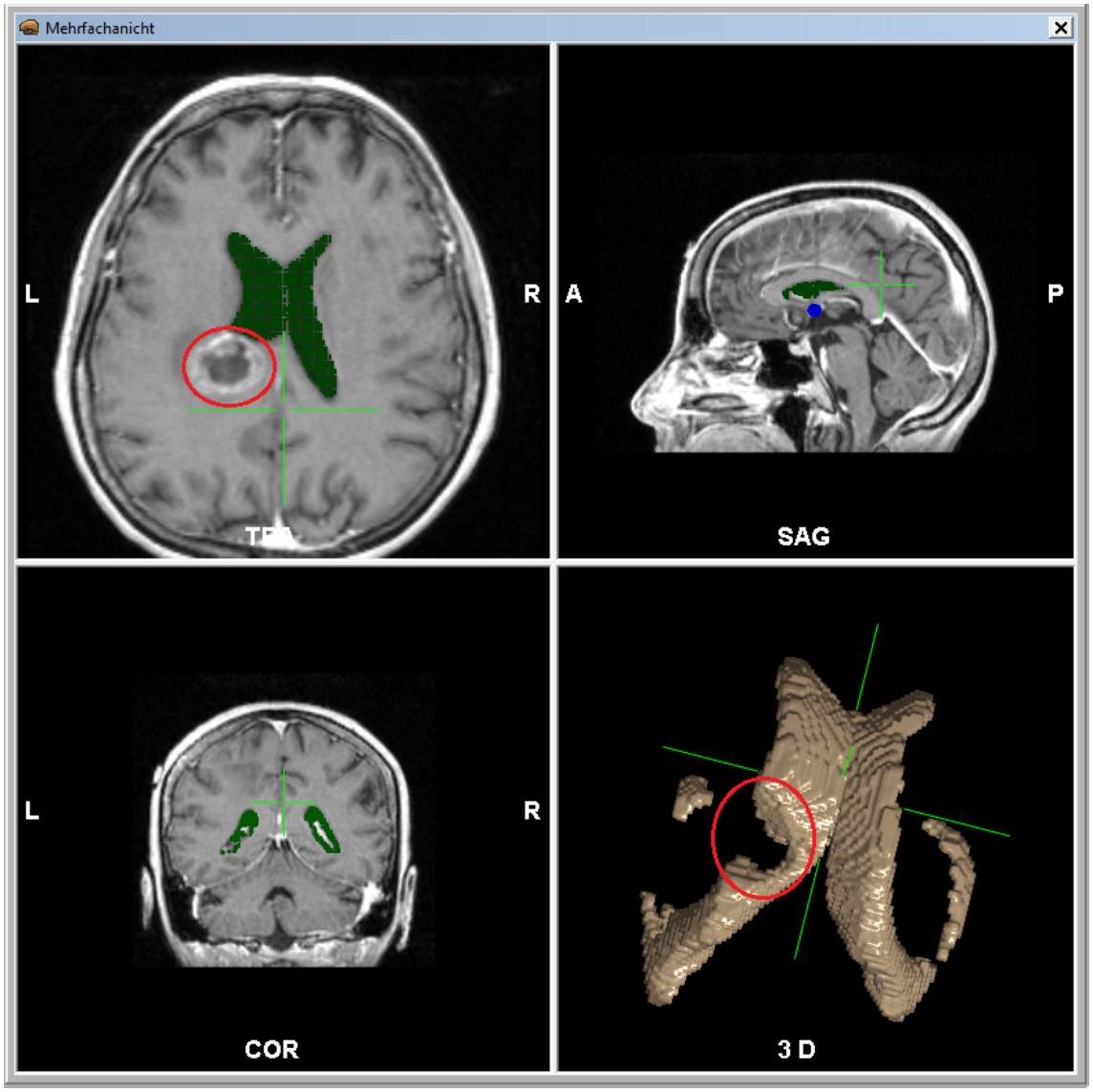

Abbildung 5.14: Das finale Segmentierungsergebnis der Ventrikel. Oben links ist ein Querschnitt des vorhandenen Tumors rot umrandet. Unten rechts ist die Auswirkung des Tumors auf die Morphologie der Ventrikel rot markiert.

5.1.4 Segmentierung eines Tumors

Im vorangegangen Beispiel wurden die lateralen Ventrikel eines Tumorpatienten segmentiert, um die Auswirkungen des Tumors auf diese zu verdeutlichen. Um die Auswirkungen des Tumors auf die lateralen Ventrikel weiter zu verdeutlichen, soll in diesem Anwendungsbeispiel als Segmentierungsziel der Tumor erfasst werden. Das Vorgehen ist hierbei analog zu den vorangegangen Beispielen. Zunächst müssen Seedpoints definiert werden, um den Tumor mit Hilfe des Regiongrowalgorithmus' erfassen zu können. Die dafür ausgewählten Seedpoints sind in Abbildung 5.15 wiedergegeben. Um eine vollständige Segmentierung

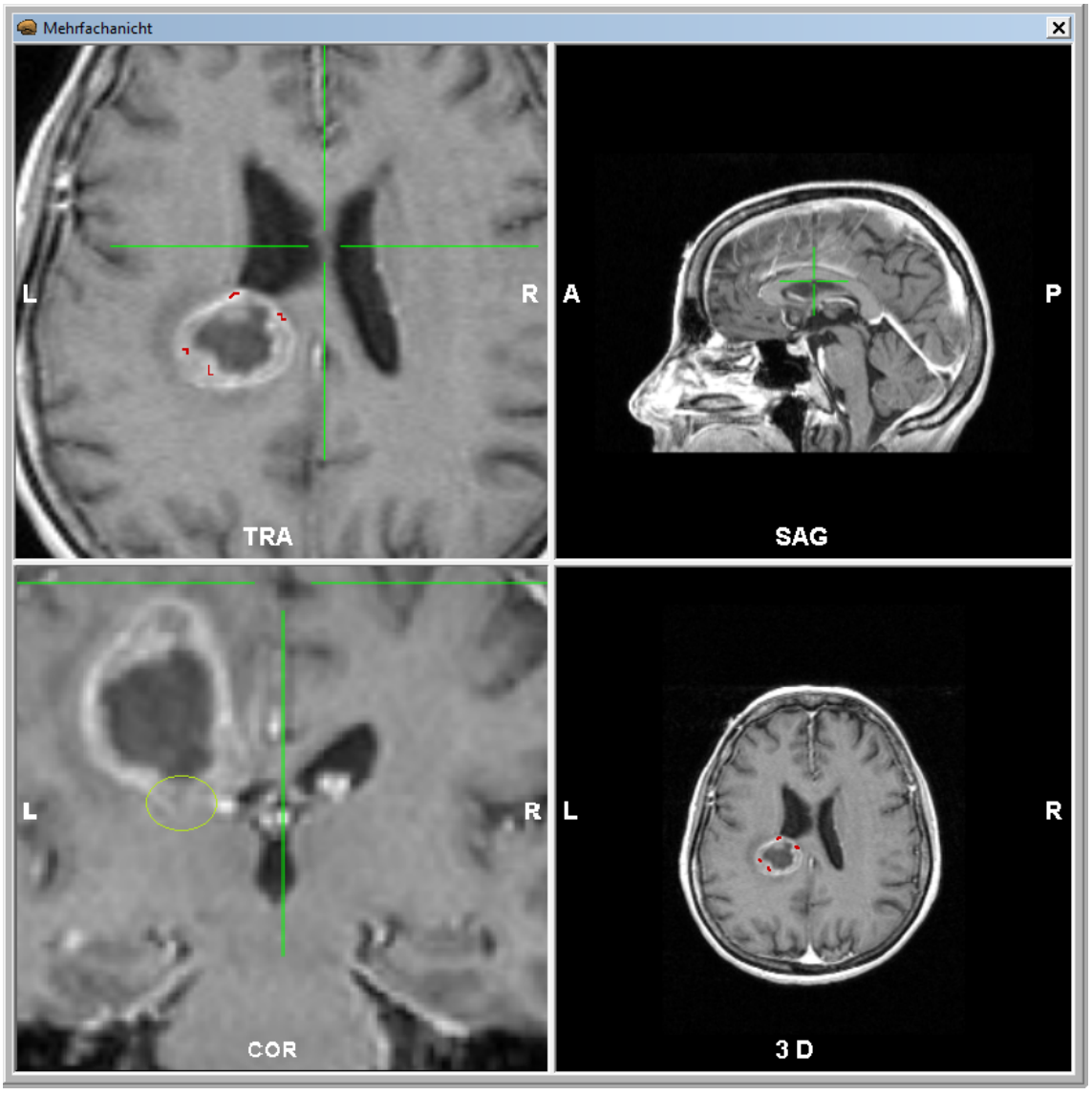

Abbildung 5.15: Initiale Seedpoints wie sie für die Segmentierung des Tumors verwendet wurden. Grün umrandet ist ein Bereich des Tumors, der sich kaum vom Hintergrund unterscheidet

des Tumors zu erzielen, war es zusätzlich nötig, die berechneten Schwellwerte anzupassen. So wurde die untere Schwelle von 217 auf 207 herabgesetzt. Da der Tumor sich in einem

bestimmten Bereich kaum von der Umgebung absetzt, also die gleichen Grauwerte wie die Umgebung besitzt (in Abbildung 5.15 grün umrandet), war es des Weiteren nötig, eine Grenze ähnlich wie in Kapitel 5.1.3 zu setzen. Das Ergebnis, das so erzielt werden konnte, ist in Abbildung 5.16 zu sehen. Um die Ergebnisse aus diesem Anwendungsbeispiel und

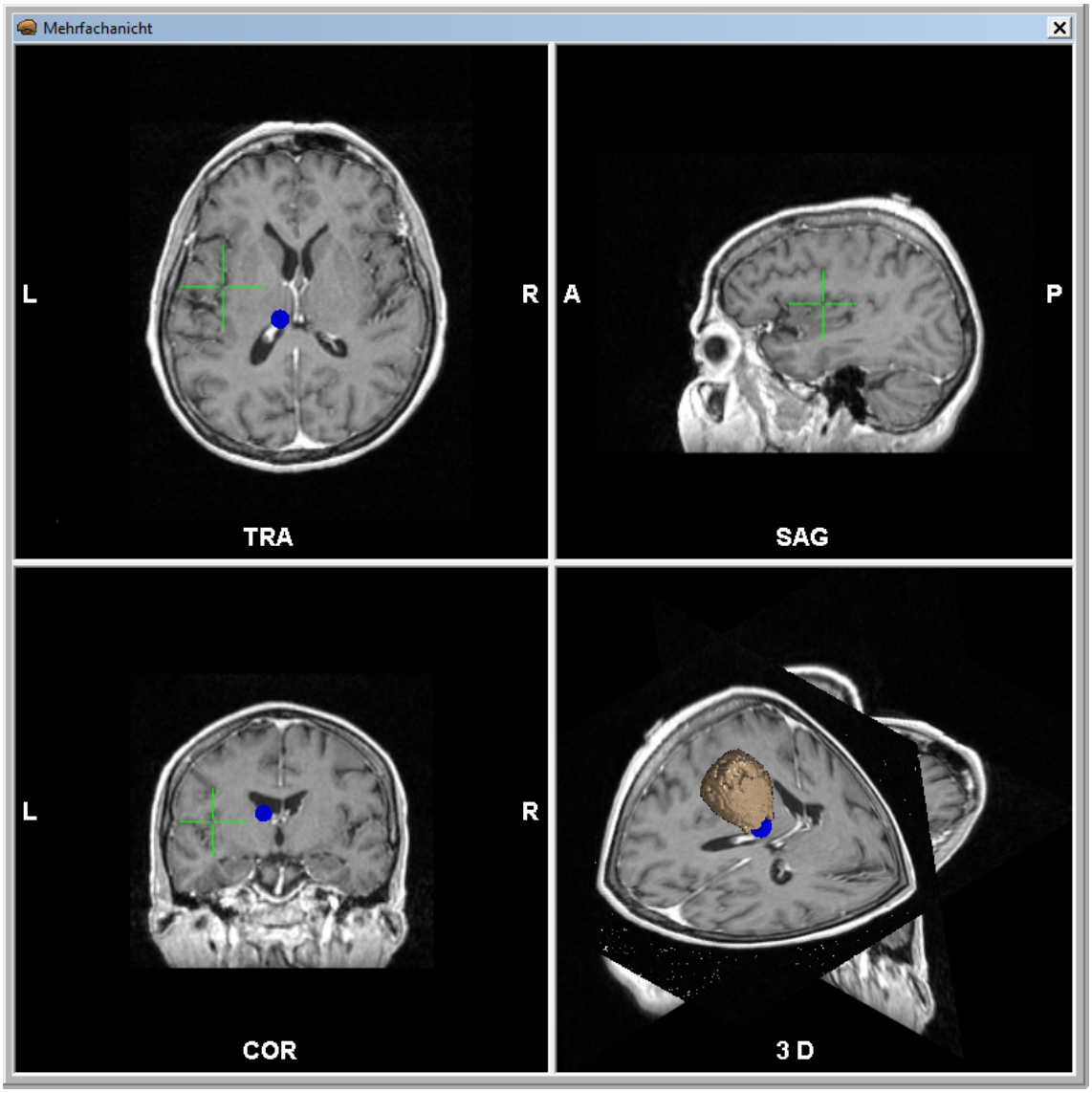

Abbildung 5.16: Segmentierungsergebnis des Tumors wie es durch Anpassung der Schwellwerte und mit der Verwendung einer Grenze erreicht werden konnte.

dem Anwendungsbeispiel aus Kapitel 5.1.3 zusammen zu führen, können diese gemeinsam angezeigt werden. Zu sehen ist dies in Abbildung 5.17. Hierdurch kann der Einfluss des Tumors auf die Ventrikel weiter verdeutlicht werden. Um den Tumor gut von den Ventrikeln optisch abzugrenzen, wurde eine unterschiedliche Farbgebung gewählt.

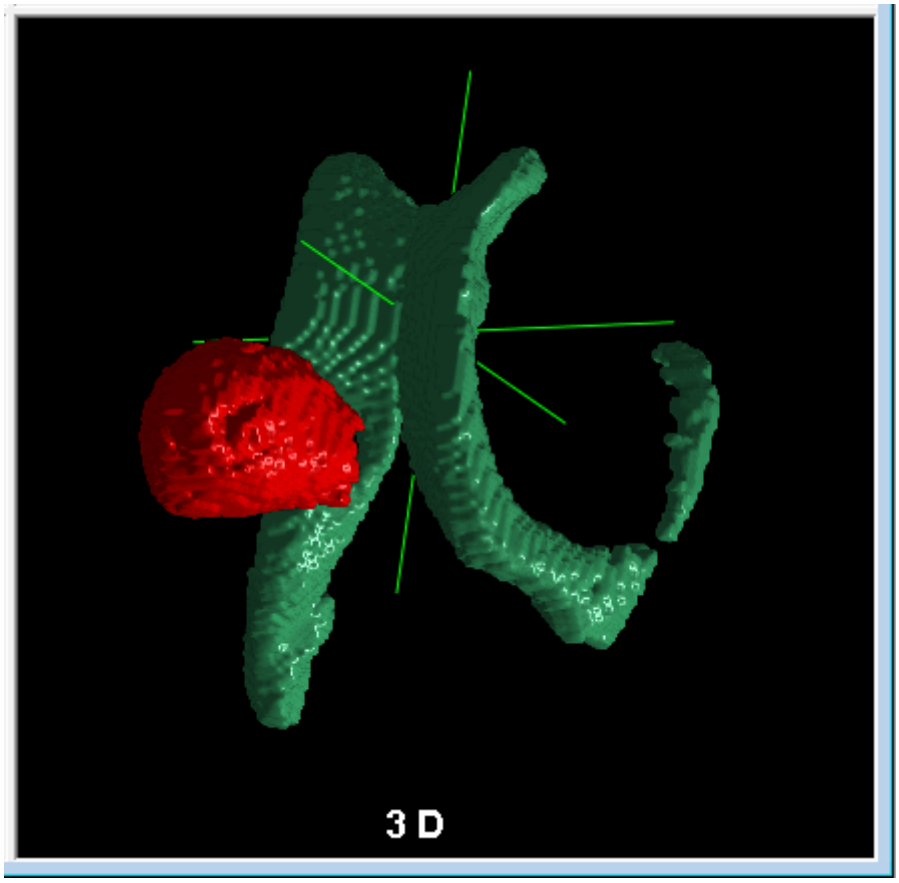

Abbildung 5.17: Gemeinsame Darstellung des segmentierten Tumors und der lateralen Ventrikel.

5.1.5 Segmentierung des Schädels

Da die bisherigen Anwendungsbeispiele immer auf Datensätzen operierten, die unter Verwendung der Magnetresonanztomographie erzeugt wurden, wird im folgenden Anwendungsbeispiel ein Datensatz verwendet, der mit Hilfe eines Computertomographen des Herstellers Siemens mit einem Pixelspacing von 0,3125 mm in der Breite und 0,3125 mm in der Höhe, erzeugt wurde. Er besteht aus 141 Schichtaufnahmen mit einer jeweiligen Dicke von 1,5 mm. Weil sich knöcherne Strukturen innerhalb eines Computertomographie-Datensatzes durch einen sehr hohen Kontrast abzeichnen und hier gezeigt werden soll, dass generell Bilddaten verschiedener Modalitäten verwendet werden können, wurde die Segmentierung des Schädels als Ziel des Anwendungsbeispiels ausgewählt. Dem entsprechend wurden initiale Seedpoints, wie sie in Abbildung 5.18 links zu sehen sind, bestimmt. Durch diese war es möglich das gewünschte Segmentierungsergebnis nahezu vollständig zu erfassen. Es mussten lediglich geringfügige Änderungen an den Schwellwerten für das Homogenitätskriterium vorgenommen werden. Das so resultierende Ergebnis ist in Abbildung 5.18 rechts wiedergegeben. Wie zu sehen ist, können durch den entwickelten Regiongrowalgorithmus ohne Probleme Bilddaten aus verschiedenen Modalitäten für die Segmentierung verwendet werden.

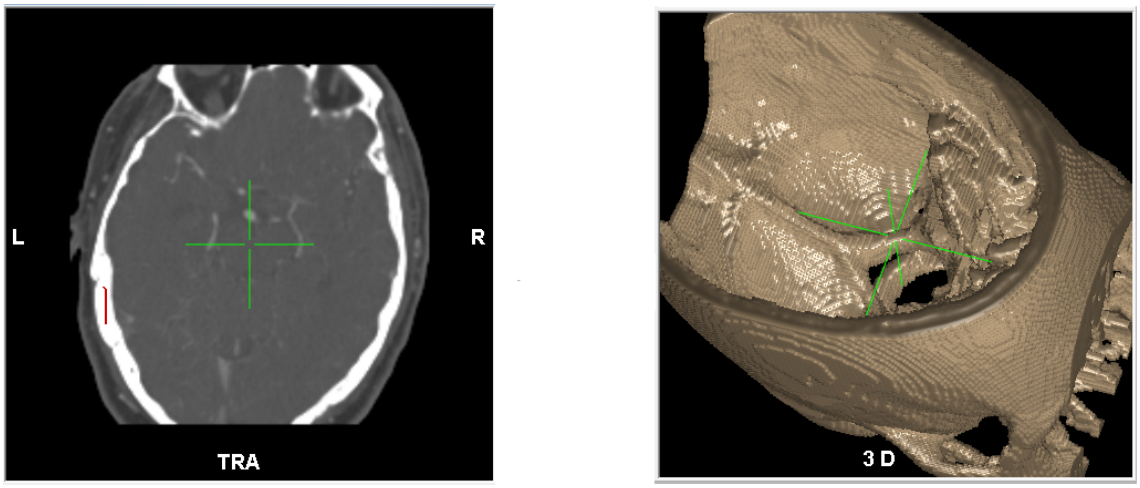

Abbildung 5.18: Links: Die für die Segmentierung verwendeten Seedpoints.
Rechts: Das Segmentierungsergebnis nach dem die Schwellwerte angepasst wurden.

5.1.6 Anwendung des Snakeverfahrens

Wie bereits in Kapitel 4.1 erläutert, wurde das Snakeverfahrens nicht über einen prototypischen Stand hinaus entwickelt. Durch diesen Prototyp wurde die generelle Realisierbarkeit des Snakeverfahrens mit Hilfe von OpenCL ermittelt. Daher wird in den folgenden Kapiteln 5.1.6.1 und 5.1.6.2 die Anwendung des Prototypen dargestellt. Zunächst wird hier ein einfaches Beispiel erläutert, um die korrekte Arbeitsweise des Verfahrens zu demonstrieren. Anschließend wird auf ein Anwendungsbeispiel unter Verwendung von medizinischen Bilddaten eingegangen. Zu beachten ist, dass der Prototyp nur auf 2D-Bildern operieren kann. Die Erweiterung des Verfahrens, um auf 3D-Bilddaten operieren zu können, sollte in einem weiterführenden Projekt jedoch kein Problem darstellen, da alle benötigten Rechenoperationen für jede Dimension separat durchgeführt werden können.

5.1.6.1 Generelle Funktionalität

In diesem Kapitel wird die generelle Funktionalität des Snakeverfahrens demonstriert. Um mittels des Snakeverfahrens die Kanten eines Objektes erfassen zu können, muss durch den Anwender zunächst eine initiale Kontur, bzw. Snake bestimmt werden. Eine solche Snake ist in Abbildung 5.19 (a) zu sehen. Zu beachten ist, dass die Snake in den meisten Bereichen sehr nah am zu segmentierenden Objekt eingezeichnet wurde. Die einzige Ausnahme besteht links unten. Hier wurde absichtlich die Snake weiter entfernt vom Objekt eingezeichnet. Dies soll das Verhalten des Alorithmus' auf eine solche ungünstige Benutzereingabe demonstrierten. In Abbildung 5.19 (b) ist die Snake wiedergegeben, wie sie nach 10 Iterationsschritten bei der Minimierung der Snakeenergie berechnet wurde. Leicht zu erkennen ist, dass das zu segmentierende Objekt fast vollständig erfasst wurde. Lediglich der Teil der Snake, der weit entfernt vom Objekt eingezeichnet wurde, konnte noch nicht an das Objekt angepasst werden. Dies liegt daran, dass die Vektoren des GVF-Feldes in

größerer Distanz zu vorhandenen Kanten sehr klein sind. So wird die Snake zwar auch in großer Entfernung von Kanten angezogen, erfährt aber pro Iterationsschritt nur geringfügige Änderungen. In Abbildung (c) ist die finale Snake wiedergeben, die das Objekt vollständig erfasst. Hierzu waren insgesamt 127 Iterationen nötig. In Abbildung 5.19 (d) ist das Betragsbild des verwendeten GVF-Feldes wiedergegeben. Dabei gilt, dass die Snake stärker angezogen wird je heller ein Bildpunkt ist. Zu beachten ist, dass das GVF-Feld weiter streut als es in der Abbildung zu sehen ist, da die Vektoren, wie bereits beschrieben, die weiter entfernt sind von Kanten einen verhältnismäßig kleinen Wert besitzen und daher nur mit einem sehr geringen Grauwert wiedergegeben werden.

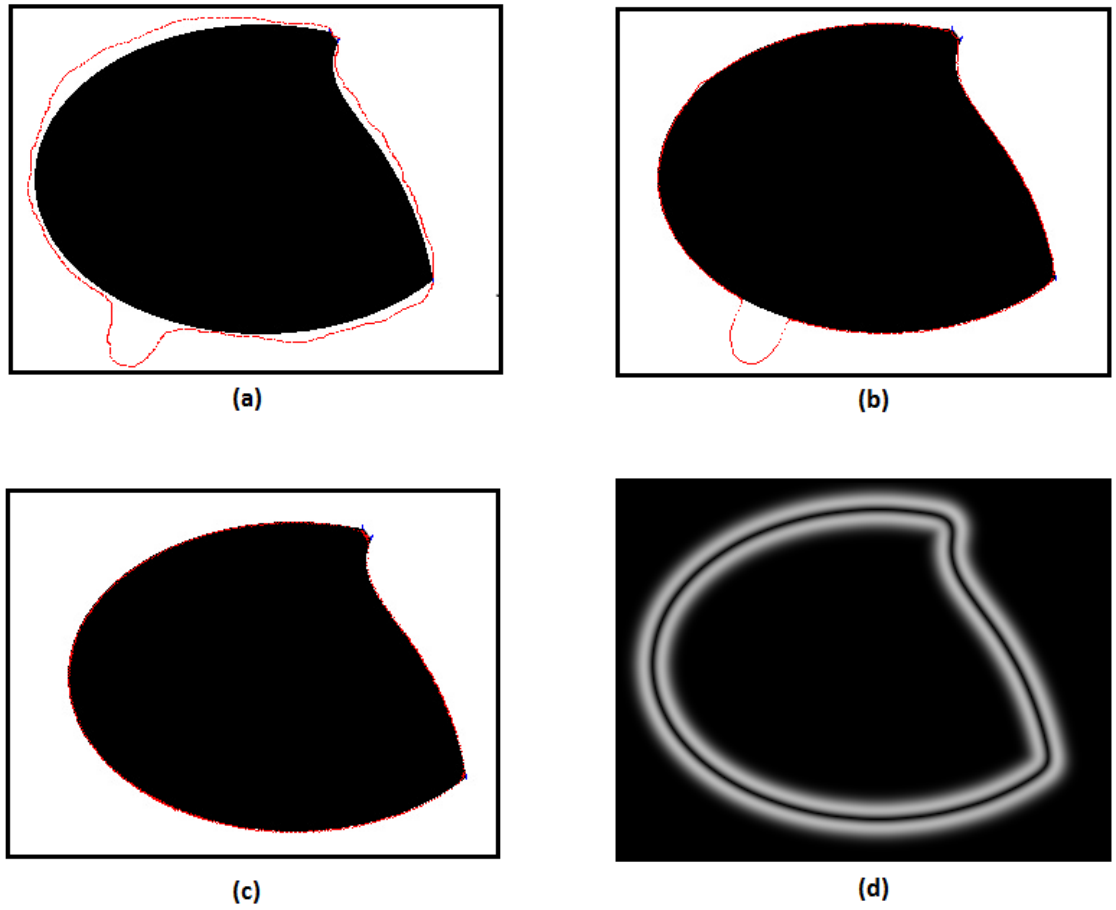

Abbildung 5.19: (a): Die initial eingezeichnete Snake.
(b): Die Snake nach 10 Iterationen.
(c): Finale Snake nach 127 Iterationen.
(d): Verwendetes GVF-Feld.

5.1.6.2 Beispielhafte Anwendung

In dem folgenden Anwendungsbeispiel soll der Bezug zur medizinischen Nutzbarkeit des implementierten Snakeverfahrens verdeutlicht werden. Die Bilddaten, auf denen in diesem Beispiel operiert wird, stellen einen Querschnitt durch den Kopf eines Menschen dar.

Sie wurden durch einen Magnetresonanztomographen des Herstellers Siemens mit einer T1-Gewichtung erzeugt. Die Maße des dabei entstandenen Bildes betragen 320*270 Pixel (in Höhe und Breite) wobei die Kantenlänge eines Pixel in der Realität einem Millimeter entspricht. Als Segmentierungsziel soll die Kontur der lateralen Ventrikel erfasst werden. Dazu wurde zunächst eine initiale Snake bestimmt, wie sie in Abbildung 5.20 (a) zu sehen ist. Anschließend wurde der Snakealgorithmus gestartet. Innerhalb von 98 Iterationen war es möglich eine vollständige Annäherung der Snake an die Kontur zu bestimmen. Wiedergegeben ist dies in Abbildung 5.20 (b). Anzumerken ist, dass aus dem hier darge-

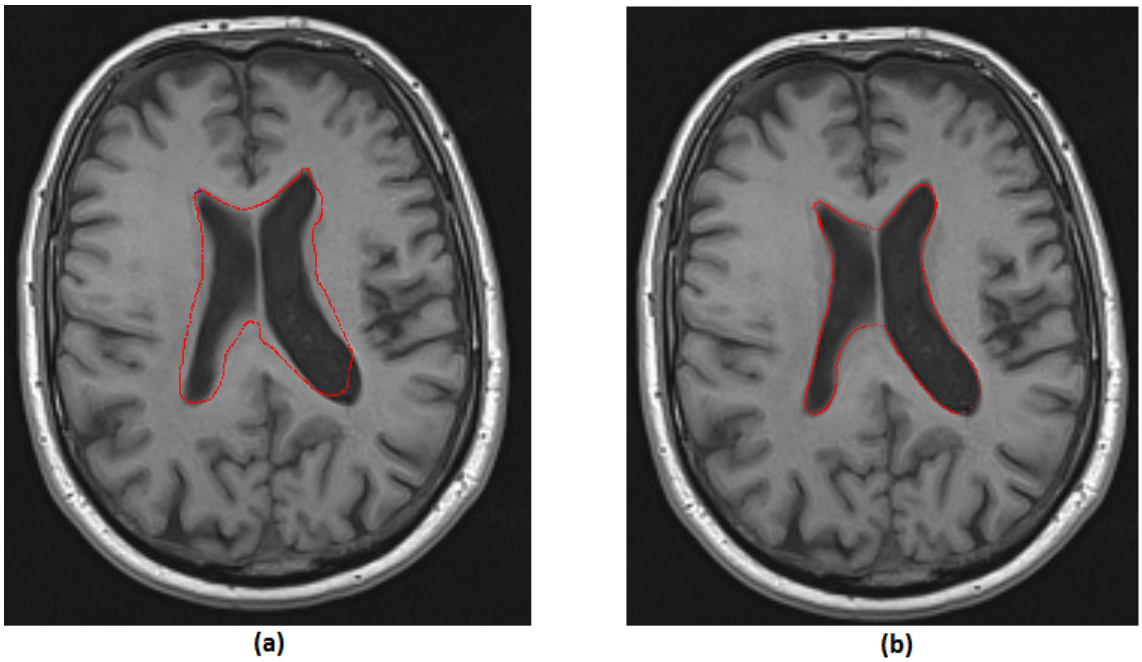

Abbildung 5.20: (a): Die initial eingezeichnete Snake.
(b): Die finale Snake nach 98 Iterationen.

stellten Segmentierungsergebnis noch kein großer Nutzen gezogen werden kann. Es sind aber weitere verschiedene Verwendungsmöglichkeiten auf Grundlage der bisherigen Funktionalität denkbar. So könnte ein 3D-Bilddatensatz Schicht für Schicht mittels des Snakeverfahrens segmentiert werden, um eine Oberflächendarstellung zu erzeugen. Auch sind im 3D-Bereich weitere Anwendungsmöglichkeiten denkbar, um z.B. Verläufe von Gefäßen zu erfassen.

5.2 Effektivität der Verfahren

Nach dem in Kapitel 5.1 die Funktionalität und die Anwendung der entwickelten Verfahren erläutert wurde, soll in diesem Kapitel auf die Effektivität der Verfahren eingegangen werden. Hierzu wird ein Vergleich zwischen den bereits vorhandenen Segmentierungsverfahren und den neu entwickelten durchgeführt. Dabei geht es um Vollständigkeit, Fehlerfreiheit und Geschwindigkeit bei der Segmentierung.

Ein genereller Gewinn in der Anwendung der GPU-beschleunigten Verfahren besteht in dem kontinuierlichen Feedback an den Nutzer. Es ermöglicht dem Anwender auf Anhieb eine fehlerfreie Segmentierung zu erzeugen. So kann schnell auf Fehler in der Segmentierung reagiert werden, um diese zu beseitigen. In den in MOPS 3D vorhandenen Standartsegmentierungsverfahren ist diese Möglichkeit nicht gegeben. Zwar können auch hier gute Ergebnisse erzielt werden, jedoch ist nicht ausgeschlossen, dass dazu mehrere Anläufe nötig sind. Auch in der Zeit, die benötigt wird, um ein vollständiges Ergebnis zu erreichen, ergeben sich in der GPU-Implementierung Vorteile. Diese lassen sich vor allem darauf zurückführen, dass bei den auf der Grafikkarte operierenden Segmentierungsverfahren keine aufwändige Oberflächendarstellung berechnet werden muss. So war es unter Optimalbedingungen möglich mit Hilfe des Regiongrowverfahrens ein Ergebnis, wie es in Kapitel 5.1.1 veranschaulicht wurde, innerhalb von 10,6 Sekunden zu erzeugen. Ein vergleichbares Ergebnis konnte mittels des bereits vorhandenen Histogrammverfahrens im Optimalfall nur innerhalb von 95,9 Sekunden hergestellt werden. Dabei wurden 72 Sekunden dazu verwendet eine Oberflächendarstellung zu berechnen und anzuzeigen. Unter Optimalbedingungen wird hier verstanden, dass der Anwender genau weiß, wie er bei der Segmentierung vorzugehen hat, um das gewünschte Ergebnis zu erzielen (also z.B. welche Schwellwerte verwendet werden müssen usw.). Die Zeiten, die als geübter Anwender für die einzelnen Segmentierungsziele benötigt wurden, wie sie in Kapitel 5.1 dargestellt sind, sind in Tabelle 5.1 wiedergegeben. Dabei sind in der Spalte »Normal« die Zeiten wiedergegeben wie

Segmentierungsziel	Normal	Optimal	Standardseg.
Cortex	1:52.0 Min.	0:10.6 Min.	1:35.9 Min.
Weisse Substanz	0:48.6 Min.	0:18.7 Min.	3:55.0 Min.
Ventrikel	2:43.5 Min.	0:48.4 Min.	2:48.5 Min
Tumor	2:12.3 Min.	1:42.6 Min.	3:39.4 Min.
Schädel	1:12.3 Min.	0:18.6 Min.	1:44.6 Min.

Tabelle 5.1: Tabelle, die die benötigte Zeit für die einzelnen Segmentierungsziele zusammenfasst.
Normal: Benötigte Zeit bei der Segmentierung ohne Vorwissen.
Optimal: Benötigte Zeit bei der Segmentierung wenn zu verwendende Parameter bekannt waren.
Standartseg.: Benötigte Zeit um durch die bereits vorhandenen Segmentierungsverfahren aus MOPS 3D ein vergleichbares Segmentierungsergebniss zu erzielen.

sie ohne Vorwissen bei der ersten Segmentierung des jeweiligen Segmentierungsziels benötigt wurden. In der Spalte »Optimal« ist wiedergegeben welche Zeit für eine erfolgreiche Segmentierung benötigt wurde, wenn alle zu verwendenden Parameter bekannt sind. Die Spalte »Standardseg.« fasst die Zeiten zusammen, die für ein mit der GPU-Segmentierung vergleichbaren Ergebnis benötigt wurden, wenn die bereits in MOPS 3D vorhandenen Segmentierungsverfahren angewandt wurden. Wie zu sehen ist, können nicht nur in kurzer Zeit gute Segmentierungsergebnisse erzielt werden, es ist darüber hinaus ein deutlicher Zeitge-

winn im Vergleich zu den Standardsegmentierungsverfahren vorhanden. Natürlich hängt die Qualität der Ergebnisse stark von der Qualität der verwendeten Bilddaten ab. Bestehen in einem verwendeten Datensatz zu viele schwache Kanten, ist ein erfolgreiches Arbeiten mit dem Regiongrowverfahren nicht mehr gewährleistet. Auch spielt die Hardware eine große Rolle bei der Segmentierung. Sie beeinflusst vor allem die für die Segmentierung benötigte Zeit. So wurden die in Tabelle 5.1 dargestellten Zeiten unter Verwendung der Grafikkarte *GeForce GTX 260* erreicht. Mit einer leistungsfähigeren Grafikkarte sollte es möglich sein noch schneller zum gewünschten Ergebnis zu gelangen. Generell war es mit der verwendeten Hardware möglich ca. 5 Iterationen pro Sekunde anzeigen zu lassen. Wurde nur das Endergebnis bei der Segmentierung durch das Regiongrowverfahren angezeigt, konnten sogar ca 70 Iterationen pro Sekunde berechnet werden. Hieraus wird deutlich, dass bei einer Weiterentwicklung in der Interoperablilität zwischen OpenGL und OpenCL deutliches Potential zur Steigerung der Performance vorhanden ist.

6 Diskussion und Ausblick

Im Verlauf dieser Diplomarbeit wurde eine unabhängige Komponente zum Segmentieren dreidimensionaler Bilddaten entwickelt. Hierfür wurden verschiedene Segmentierungsverfahren analysiert, konzipiert und prototypisch auf ihre Eignung für eine Umsetzung mit OpenCL getestet. Anschließend wurden die entwickelten Verfahren in einer unabhängigen Komponente umgesetzt, welche ihren Einsatz in MOPS 3D gefunden hat. In diesem Kapitel werden verschiedene Aspekte der erreichten Ergebnisse zunächst kritisch beleuchte. Anschließend wird ein Ausblick auf mögliche zukünftige Entwicklungen gegeben.

6.1 Diskussion

Die Entwicklung einer unabhängigen Segmentierungskomponente, die durch GPU-Beschleunigung ein neues Maß an Interaktion mit dem Anwender erlaubt, hat sich Dank der Verwendung von OpenCL als Erfolg erwiesen. Die hohe Parallelisierbarkeit der implementierten Verfahren erbrachte ein Leistungsniveau, das es ermöglicht, dem Anwender jederzeit Rückmeldung über den aktuellen Segmentierungsstand zu geben. Hierdurch wird ein komfortabler Umgang mit den entwickelten Verfahren möglich.

Entscheidend war dabei der Einsatz des OpenCL-Standards. Neben der hohen Parallelität bei der Ausführung der verwendeten Algorithmen konnte durch die Verwendung von OpenCL weitestgehend eine Platformunabhängigkeit erreicht werden, da OpenCL von vielen namhaften Herstellern unterstützt wird (z.B. NVIDIA, AMD und Intel). Dies war einer der Hauptgründe warum bei der Implementierung OpenCL der Vorzug vor der *Compute Unified Device Architecture* (CUDA) von Nvidia gegeben wurde. Bei CUDA handelt es sich ähnlich wie bei OpenCL um eine Plattform, die es ermöglicht, Programme direkt auf der Grafikkarte auszuführen. Jedoch wurde CUDA zu Beginn dieser Diplomarbeit ausschließlich von Grafikkarten des Herstellers NVIDIA unterstützt. OpenCL hat daher große Chancen Marktführer bei der Entwicklung von GPU-beschleunigten Anwendungen zu werden. Dies sichert die Zukunftsperspektive bei der Verwendung von OpenCL. Diese Aussicht wird jedoch dadurch getrübt, dass gegen Ende dieser Diplomarbeit veröffentlicht wurde, dass CUDA in Zukunft auch von anderen Grafikkarten (z.B. von AMD) unterstützt werden soll ([21]). Welchen Einfluss dies auf die weitere Entwicklung von OpenCL hat wird sich in der Zukunft zeigen.

Auch bereitete die Entwicklung mit OpenCL einige Probleme. So fehlt es an Tools, die es dem Programmierer erleichtern Anwendungen, die OpenCL verwenden, zu implementieren. Ein besonderes Problem ergibt sich hier bei der Fehlersuche (dem Debugging) in dem

entstandenen Code. So gibt es während der Umsetzung dieser Diplomarbeit kein Tool, welches es erlaubt, den Programmablauf auf der Grafikkarte bei der Ausführung befriedigend nachzuvollziehen. Dieser Mangel lässt sich darauf zurückführen, dass es sich bei OpenCL um einen sehr jungen Standard handelt. So sind z.B. für CUDA solche Tools vorhanden. Diese wurden jedoch auch erst im Laufe der bereits längeren Entstehungsgeschichte von CUDA entwickelt, was darauf hoffen lässt, dass es eine ähnliche Entwicklung im Bereich von OpenCL geben wird.

Des Weiteren lässt der Entwicklungsstand von OpenCL vonseiten der Hersteller noch Wünsche offen. So ist es, wie in Kapitel 4.2.2 beschrieben, mit aktuellen Treibern von NVIDIA noch nicht möglich innerhalb eines *Kernels* schreibend auf eine 3D-Textur zuzugreifen. Um dennoch Ergebnisse anzeigen zu können, war ein Workaround auf der *Hostseite* der Segmentierungskomponente nötig. Hierdurch ergeben sich unnötige Performance-Einbußen. Mit der Weiterentwicklung von OpenCL sollten solche Zugriffe in Zukunft jedoch möglich sein. Diese ließen sich dann leicht in die vorhandene Implementierung integrieren. Hier besteht also großes Potential, um die vorhandene Segmentierungskomponente weiter zu verbessern. Auch erscheint das Speichermanagement der verwendeten Treiber von NVIDIA unter kleineren Fehlern zu leiden. So führte das wiederholte allokieren und freigeben größerer Speichervolumen (in der Größenordnung von 200MB und größer) zu undefiniertem Auftreten von Fehlern in der Anwendung, so dass trotz genügender freier Ressourcen kein weitere Speicher allokiert werden kann. Dem wurde dadurch entgegengewirkt, dass verwendeter Speicher nur an unbedingt nötigen Stellen freigegeben wird und dem entsprechend dieser erst wieder bei Bedarf allokiert wird. Hierdurch konnte das Auftreten von Fehlern weitestgehend beseitigt werden. Dabei handelt es sich jedoch nur um eine Symptombehandlung, die nicht die Ursache für das beschriebene Verhalten beseitigen kann. Auch hier bleibt abzuwarten, ob zukünftige Treiberversionen, in Bezug auf die Speicherverwaltung, Abhilfe schaffen werden.

Trotz der beschriebenen Mängel hat sich der Einsatz von OpenCL gelohnt. Vor allem der umgesetzte Regiongrowalgorithmus konnte allen Anforderungen gerecht werden. So konnte in Kapitel 5 ausführlich die komfortable Anwendbarkeit des Verfahrens demonstriert werden. Zu erwähnen ist hier, dass die Qualität der Segmentierungsergebnisse stark von der Qualität der verwendeten Bilddaten abhängt. Dies ist jedoch ein generelles Problem der Segmentierung und lässt sich nicht auf die entstandene Implementierung zurückführen. Auch die entwickelten Filteralgorithmen arbeiten hoch performant. Einzige Ausnahme bildet hier der Medianfilter. So stellte sich heraus, dass OpenCL für Sortieralgorithmen (welche nötiger Bestandteil eines Medianfilters sind) eher ungeeignet ist, da die Architektur von GPUs nicht darauf ausgelegt ist viele Vergleichsoperationen in kurzer Zeit durchführen zu müssen. Hier besteht weiterer Forschungsbedarf um performante Sortieralgorithmen für OpenCL zu ermitteln.

Der entwickelte Snakeprototyp konnte mit OpenCL ebenfalls sehr performant umgesetzt werden. Lediglich die Integration des Verfahrens in die entstandene Segmentierungskomponente konnte in dieser Diplomarbeit nicht durchgeführt werden. Dies begründet sich

auf die schwierige Handhabung von OpenCL und den damit entstandenen Zeitproblemen. Nichts desto trotz konnte die Funktionalität des Verfahrens unter Verwendung von OpenCL im 2D-Bereich erfolgreich demonstriert werden. Eine Weiterentwicklung des Verfahrens für die Anwendung in dreidimensionalen Datensätzen sollte kein Problem darstellen. Dies ergibt sich daraus, dass alle Berechnungen des Verfahrens für jede Dimension separat und auf gleiche Weise durchgeführt werden. So müssten beim Wechsel von 2D auf 3D lediglich bereits vorhandene Algorithmen auf eine zusätzliche Dimension angewandt werden. Das einzige Problem, welches sich hierbei ergibt, ist der hohe Speicherverbrauch bei der Berechnung des GVF-Feldes. Hier können Datenvolumen von der Größe eines Gigabytes überschritten werden. Dies könnte auf älteren Grafikkarten wie z.B. der GeForce GTX 260 (mit 800MB Speicher) dazu führen, dass die Berechnung in mehrere Schritte aufgeteilt werden muss. Jedoch besitzen Mittelklasse-Grafikkarten gegen Ende dieser Diplomarbeit bereits zwischen ein und zwei Gigabyte internen Speicher (z.B. die GeForce GTS 450), was darauf schließen lässt, dass dieses Problem in Zukunft nicht mehr besteht.

6.2 Ausblick

Wie bereits gezeigt wurde, ist mit der entwickelten Segmentierungskomponente eine solide Grundlage für den Einsatz von Segmentierungsalgorithmen geschaffen worden, die auf OpenCL basieren. So ist die Entwicklung weiterer Verfahren, die in die bestehende Komponente integriert werden sollen, ohne weiteres denkbar. Dies wird durch das Konzept der *GPUTools* sowie dem Design des *SegmentationManger* möglich. Ein weiteres Verfahren wäre z.B. die GPU-beschleunigte Segmentierung mittels des Histogrammverfahrens. Dies könnte im Vergleich zu dem bereits bestehenden Verfahren in dem Planungssystem MOPS 3D so erweitert werden, dass dem Anwender direkt die Auswirkung von gesetzten Schwellwerten angezeigt wird. Darüber hinaus sind der Verwirklichung weiterer Verfahren keine Grenzen gesetzt, insofern sich diese gut parallelisieren lassen.

Des Weiteren sind für das entwickelte Snakeverfahren viele Ausbaumöglichkeiten gegeben. So ist es denkbar statt einfacher Konturen ganze Ebenen für die Repräsentation von Snakes zu wählen. Damit wäre es theoretisch möglich direkt die Oberfläche von Objekten vollständig zu erfassen. Des Weiteren könnten Snakes auch in 4D-Datensätzen angewandt werden, um z.B. Bewegungsabläufe zu erfassen. Natürlich ist dies mit einem erheblichen Aufwand bei der Integrierung des Snakeprototypen in die bestehende Segmentierungskomponente verbunden, da einige Erweiterungen des Verfahrens implementiert werden müssten. Dennoch ist eine solche Nutzung des Verfahrens zukünftig denkbar.

Da das entwickelte Softwarepaket eine vom Planungssytem MOPS 3D unabhängige Komponente bildet, ist neben der Integration neuer Verfahren auch die Verwendung in anderen Anwendungen möglich. Eine besondere Flexibilität für die Wiederverwendung der entwickelten Komponente ist dadurch gegeben, dass auf die Interoperation von OpenCL und OpenGL gesetzt wurde. Wie bereits beschrieben wird OpenCL von vielen Herstellen unterstützt und auch OpenGL hat sich, vor allem angetrieben durch die Spieleindustrie, zu

einem quasi Industriestandard entwickelt.

Limitierend sind hier lediglich die relativ hohen Hardwarevoraussetzungen. So sollte ein System, in dem die Segmentierungskomponente eingesetzt werden soll, mindestens über eine Grafikkarte mit 800MB internen Speicher verfügen. Darüber hinaus muss natürlich OpenCL vonseiten der Grafikkarte unterstützt werden. Handelsübliche Mittelklasse-Grafikkarten erfüllen zum derzeitigen Stand der Diplomarbeit diese Anforderungen. Problematisch sind hier nur Grafikkarten älterer Systeme. Jedoch sind Grafikkarten, die die genannten Voraussetzungen nicht erfüllen ca. 4 Jahre alt und werden somit ihre Lebensdauer in den folgenden ein bis zwei Jahren überschritten haben, so dass auch diese in Zukunft nicht mehr zu einem Problem führen werden.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] John E. Stone, David Gohara, Guochun Shi: *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.* Comput Sci Eng. 2010 May.

[2] AMD: *Introduction to OpenCL Programming, Training Guide.* Mai 2010.

[3] Munshi Aaftab: *OpenCL Specification Version 1.1.* Juni 2011. http://www.khronos.org/registry/cl/

[4] Benedict Gaster, David R. Kaeli, Lee Howes, Perhaad Mistry, Dana Schaa: *Heterogeneous Computing with OpenCL.* Elsevier, August 2011.

[5] Jackie Neider, Tom Davis, Mason Woo: *OpenGL Programming Guide; The Official Guide to Learning OpenGL, Release 1.* Addison-Wesley Longman Publishing Co., 1994.

[6] Bernd Jähne: *Digitale Bildverarbeitung.* Springer, August 2005.

[7] Bart M. ter Haar Romeny: *Front-end vision and multi-scale image analysis: multi-scale computer vision theory and applications, written in Mathematica.* Springer, September 2003.

[8] Weisstein, Eric W.: *Gaussian Function.* MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GaussianFunction.html. ; Letzter Aufruf 13.12.2011.

[9] Gonzalo R. Arce: *Nonlinear signal processing: a statistical approach.* John Wiley and Sons, 2005

[10] Jakob Santner: *Interactive Multi-Label Segmentation.* Graz,Österreich, Oktober 2010.

[11] Heinz Handels: *Medizinische Bildverarbeitung: Bildanalyse, Mustererkennung und Visualisierung für die computergestützte ärztliche Diagnostik und Therapie (Studienbücher Medizinische Informatik).*

[12] Gonzalez Woods,Steven Eddins: *Digital Image Processing Using MATLAB.* Prentice Hall, Februar 2004.

[13] Wolfgang Birkfellner: *Applied Medical Image Porcessing, A Basic Course.* Taylor & Francis, September 2010.

[14] Frederic F. Leymarie: *Tracking deformable objects in the plane using an active contour model.* IEEE Trans. Pattern Anal. Machine Intell., vol. 15, pp. 617–634, 1993..

[15] Chenyang Xu and Jerry L. Prince: *Snakes, Shapes, and Gradient Vector Flow.* IEEE Transactions on image processing, vol. 7, no. 4, März 1998.

[16] C. Davatzikos und Jerry. L. Prince: *An active contour model for mapping the cortex.* IEEE Transactions on medical imaging, vol. 14, no. 1, März 1995.

[17] Meister Andreas: *Numerik linearer Gleichungssysteme. 4. Auflage.* Vieweg + Teubner, Februar 2011.

[18] Chenyang Xu and Jerry L. Prince: *Gradient Vector Flow: A New External Force for Snakes.* IEEE Proc. Conf. on Comp. Vis. Patt. Recog, Los Alamitos: Comp. Soc. Press, pp. 66-71, 1997.

[19] R. Metzner, U. Eisenmann, CR. Wirtz, H. Dickhaus: *Integration of Pre- and Intraoperative Multimodal Data Sources in Neurosurgical Operations.* IFMBE Proceedings Volume 11, 2005. Prague, Czech Republic 2005.

[20] S. Diatschuk: *Entwicklung einer performanten Volume Rendering Komponente für das Operationsplangunssystem MOPS 3D.* Universität Heidelberg, Januar 2012.

[21] Heise Online: *NVIDIAs CUDA-Technik bald auch für AMD-GPUs.* Quelle: http://www.heise.de/developer/meldung/NVIDIAs-CUDA-Technik-bald-auch-fuer-AMD-GPUs-1395220.html, letzter Zugriff: 22.01.2012.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und ist auch noch nicht veröffentlicht.

Ort, Datum	Marius Wirths