Hochschule für Technik, Wirtschaft und Kultur Leipzig (FH)
Fachbereich Informatik, Mathematik und Naturwissenschaften
Studiengang Informatik

Konzeption und Implementierung eines Systems zur Visualisierung und Bearbeitung von 3D–Volumendaten auf Consumer–Hardware

Diplomarbeit

eingereicht von

Enrico Reimer

Datum:	28.1.2005
Matrikelnummer:	29936 (00IN)
eMail:	enni_@t-Online.de
Gutachter (HTWK):	Prof. Dr.-Ing. Frank Jaeger
Betreuer (CBS):	PD Dr. Gabriele Lohmann

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Alle Stellen, die den Quellen wörtlich oder inhaltlich entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Leipzig, den 28.1.2005 Enrico Reimer

Danksagung

Diese Arbeit entstand in Kooperation mit dem Max-Planck-Institut für Kognitions- und Neurowissenschaften. An dieser Stelle möchte ich mich bei allen Kollegen für die Bereitstellung des Themas und der Arbeitsmittel sowie für die fortwährende Unterstützung während der Bearbeitung bedanken!

Außerdem gilt mein Dank Herrn Prof.Dr. Jaeger von der HTWK, der sich die Zeit nahm, Vorabversionen des Textes zu lesen, und der mit fundierten Hinweisen zur Verbesserung der Diplomarbeit beitrug.

Eine große Hilfe sind auch meine Beta-Leser, vor allem Nico und Johannes gewesen. Ohne ihre ausführliche Fehlersuche und die zahlreichen Tipps sähe meine Arbeit heute mit Sicherheit anders aus.

Enrico Reimer

Erkennen heißt: zu der halben Wirklichkeit der Sinnenerfahrung die Wahrnehmung des Denkens hinzufügen, auf daß ihr Bild vollständig werde.

J.W. von Goethe

Neben der Genauigkeit des Messinstruments ist die schnelle und korrekte Interpretation des Gemessenen von zentraler Bedeutung für das Messen selbst. Moderne bildgebende Messsysteme erzeugen heute große Mengen oft dreidimensionaler graphischer Informationen die von Fachleuten interpretiert werden müssen. Diese Diplomarbeit beschäftigt sich mit dem Entwurf und der Implementierung eines Systems zur Visualisierung dreidimensionaler Messdaten, wie sie beispielsweise Kernspintomografen liefern.

Inhaltsverzeichnis

1 Einleitung

Der Nutzen einer Messmethode hängt in großem Maße davon ab, wie gut die erhaltenen Daten interpretiert und angewandt werden können. In der modernen Medizin liefern Kernspintomografen nicht nur punktgenaue Informationen über anatomische Details unseres Körpers, sie lassen auch Erkenntnisse über natürliche Vorgänge in ihm zu. So ist durch die funktionelle Kernspintomografie inzwischen messbar, welcher äußere Reiz in welchem Ausmaß zu erhöhten Aktivitäten in bestimmten Teilen des menschlichen Gehirns führt. Obwohl sich die Messtechnik und die Methoden zur automatischen Verbesserung der Daten rasant weiterentwickeln, müssen diese Informationen noch immer von Menschen interpretiert werden, welche oft ihre Probleme mit den komplexen Informationen haben. Es ist daher eine zugleich schwierige, aber auch wichtige Aufgabe, die dreidimensionalen Messdaten (Volumendaten) für den menschlichen Betrachter entsprechend aufzuarbeiten und geeignet darzustellen.

Deshalb befasst sich diese Diplomarbeit mit dem Entwurf und der Entwicklung eines sowohl einfachen, als auch flexiblen Systems zur Visualisierung und Bearbeitung von Volumendaten.

1.1 Motivation

Die Bildgebungsverfahren der Kernspintomografie (MRI) und der funktionellen Kernspintomografie (fMRI) haben in den letzten Jahren große technische Fortschritte gemacht. Die Auflösung von Kernspintomografen steigt ständig, so dass immer genauere Daten ermittelt, und Aktivitäten im Gehirn immer genauer gemessen werden können. Auch die mathematischen Modelle zur Erzeugung und Aufbereitung der ermittelten Daten werden rapide weiterentwickelt. Ferner stehen im Bereich der interaktiven Visualisierung und Verarbeitung der Daten einige umfangreiche, professionelle Softwaresysteme zur Verfügung.

1.1.1 Verfügbare Mittel der Visualisierung

Diese Programme sind jedoch oft überdimensioniert und bieten meist mehr Funktionalitäten als erforderlich. Neben der unnötigen Erschwerung für den Anwender erhöht diese übertriebene Komplexität den Bedarf an Ressourcen und verteuert die Softwareprodukte gleichzeitig. Außerdem kommt es oft vor, dass trotz des hohen Funktionsumfanges gerade das eine Feature, das dringend gebraucht wird, fehlt. Eine Anpassung ist meist nicht nur aus lizenzrechtlichen Gründen problematisch, sondern oft auch aufgrund des Umfangs dieser Systeme schlicht unmöglich. Was fehlt, ist ein einfaches, flexibles und effektives System, das eine möglichst direkte und intuitive Visualisierung der Volumendaten erlaubt, und sich auf die wesentlichsten Funktionen beschränkt. Gleichzeitig sollte es auf gängiger PC-Hardware flüssig und stabil betrieben werden können und sich leicht anpassen bzw. erweitern lassen.

1.1.2 Objekterkennung durch Segmentierung

Die Erkennung von Strukturen ist eine der üblichsten Aufgaben bei der Interpretation der Daten bildgebender Messsysteme. Die durch die Kernspintomografie generierten Volumendaten bilden in diesem Zusammenhang keine Ausnahme. Aus diesem Grunde finden beim Max-Planck-Institut für Kognitions- und Neurowissenschaften (CBS) mehrere Algorithmen zur Objekterkennung Anwendung. Diese hoch spezialisierten Algorithmen lassen jedoch keinen Nutzereingriff zu. Etwaige Erkennungsfehler müssen im Nachhinein mühsam von Hand korrigiert werden. Es wäre effektiver diese Fehler im Entstehen zu unterbinden, d.h. den Rechner schon im Ansatz zu korrigieren, bevor sich der gemachte Fehler ausweitet.

1.2 Zielsetzung

Inhalt dieser Diplomarbeit ist der Entwurf und die Realisierung eines Programms zur gemeinsamen Visualisierung anatomischer und funktioneller Volumendaten mittels einer geeigneten Darstellungsstrategie. Das dabei entwickelte Programm soll neben der Visualisierung der Volumendaten Werkzeuge für eine interaktive Objekterkennung innerhalb dieser Daten bieten. Es wird daher nötig sein, Methoden zur interaktiven Segmentierung zu entwerfen, bei der sich Anwender und Rechner gegenseitig unterstützen. Dieses

System soll auf zurzeit gängigen Büro-PCs möglichst ohne zusätzliche Kosten einsetzbar sein. Daraus folgt, dass seine Anforderungen an die Hardware möglichst gering sein sollten und es vor allem ohne Spezialhardware auskommen muss. Die Erweiterung und Anpassung dieses Programms soll möglichst leicht, und eine Portierung auf andere Plattformen mit geringem Aufwand möglich sein.

1.3 Aufbau der Arbeit

- Kapitel 2 beschreibt die allgemeinen Grundlagen dreidimensionaler Messungen und der Visualisierung der so ermittelten dreidimensionalen Messdaten auf zweidimensionalen Anzeigesystemen wie z.B. Bildschirmen.
- Kapitel 3 behandelt den Entwurf, und die Implementierung eines auf *OpenGL* basierenden allgemeinen Basissystems zur Visualisierung dreidimensionaler Szenen. Es werden auch die nötigen mathematischen Grundlagen dieser Visualisierung besprochen.
- Kapitel 4 beschreibt den Aufbau einer, auf den in Kapitel 3 angestellten Überlegungen basierenden Basisbibliothek. Diese Bibliothek wird die Implementierung und spätere Pflege des eigentlichen Programms erleichtern.
- Kapitel 5 schildert die theoretischen Grundlagen einer auf "Wasserscheiden" basierenden Segmentierung im Raum, während Kapitel 6 zwei Implementierungen dieses Verfahrens behandelt.
- Kapitel 7 behandelt die auf den vorherigen Arbeiten basierende Implementation des eigentlichen Visualisierungs- und Bearbeitungssystems und Kapitel 8 schließt diese Arbeit mit einer Zusammenfassung ab.

2 Grundlagen und Vorüberlegungen

2.1 Die Magnetresonanztomografie

Der folgende kurze Abriss der Geschichte der Magnetresonanztomografie basiert hauptsächlich auf einer Zusammenstellung von Hornak [8] und Informationen aus der Wikipedia [22].

Felix Bloch und Edward Purcell entdeckten 1946 unabhängig voneinander das Prinzip der kernmagnetischen Resonanz (NMR). Wie so oft in der Grundlagenforschung wurde anfangs keine Anwendung für diesen physikalischen Effekt gesehen, und die Entdeckung blieb relativ unbeachtet. Erst ab 1950 wurde die NMR wieder aufgegriffen und für chemische und physikalische Analysemethoden weiterentwickelt. 1952 wurde Felix Bloch und Edward Purcell für ihre Entdeckung den Nobelpreis für Physik verliehen.

Eine Anwendung in der Medizin fand die NMR erst 1971, als Raymond Damadian nachweisen konnte, dass sich die Magnetresonanzen von gesundem Gewebe und von Tumoren unterscheiden. 1972 wurde die auf Röntgenstrahlen basierende Computertomografie (CT) eingeführt. Das hatte zwar nichts mit der NMR zu tun, zeigte aber, dass Krankenhäuser durchaus gewillt waren, große Geldmengen für medizinische Untersuchungsgeräte auszugeben. Noch im selben Jahr leitete Paul Lauterbur von der CT ein auf NMR basierendes Verfahren ab und konnte erste Bilder von kleinen Gewebeproben erzeugen. Der Grundstein der heutigen MRI wurde jedoch erst 1975 von Richard Ernst gelegt, als dieser ein auf der Fourier-Transformation basierendes NMR-Bildgebungsverfahren vorstellte. Anders als die CT hat die MRI keine bekannten Nebenwirkungen und kann relativ problemlos, auch in kurzen Zeitabständen, an jedem Teil des Körpers durchgeführt werden.

Aufgrund des hohen technischen und mathematischen Aufwands konnten lange Zeit nur kleine Bilder angefertigt werden. Erst die rasante Entwicklung der Rechentechnik ermöglichte es ab 1980 mittels Ernsts Technik innerhalb von 5 Minuten komplette Schnittbilder des Körpers zu erzeugen. Bis 1986 wurde diese Zeit weiter auf 5 Sekunden

reduziert. Außerdem konnten erstmals mehrere Schnitte nacheinander abgebildet und zu den heute üblichen Volumendaten zusammengefasst werden. 1991 erhielt Richard Ernst für seine Arbeit den Nobelpreis für Chemie.

Die ein Jahr später entwickelte funktionelle MRI (fMRI) verwendet das von Ogawa [17] publizierte BOLD-Verfahren um die Sauerstoffanreicherung von Blut kernspintomografisch zu erfassen. Da sich das Verhältnis von oxigeniertem zu deoxigeniertem Blut zusammen mit der Erhöhung der Aktivität in bestimmten Gehirnregionen ändert, lässt sich diese Aktivität zeitnah messen. Diese gemessenen Aktivitäten lassen sich dann in direkten zeitlichen Zusammenhang mit äußeren Reizen bringen. Mit Hilfe der fMRI kann somit eine dreidimensionale Aktivitätskarte des Gehirns angefertigt werden.

2.2 Visualisierung dreidimensionaler Daten

In den letzten Jahren ist die Computergrafik zu einem alltäglichen Werkzeug in der Wissenschaft geworden. Vor allem die computergestützte Visualisierung, im Folgenden allgemein als *Renderings* bezeichnet, ist für die Darstellung großer und komplexer Datenmengen ein gern verwendetes Mittel. Dies trifft besonders auf die Visualisierung dreidimensionaler Daten zu. Die zwei wichtigsten Methoden des *Renderings* dreidimensionaler Daten sind das *volume rendering* und der *Schnitt*.

2.2.1 Volume rendering

Da dreidimensionale Datensätze meist das Ergebnis einer Messung an einem natürlichen dreidimensionalen Objekt sind, also dieses Objekt in einer bestimmten Weise repräsentieren, liegt es nahe dieses Objekt auch in seiner natürlichen Form darzustellen. Zu diesem Zweck wird jedes Datum an dem Punkt im Anzeigeraum abgebildet, an dem es im reellen Raum gemessen wurde. Diese nur in Bereich der computergestützten Visualisierung verfügbare Methode wird als *volume rendering* bezeichnet. Sie liefert ein naturgetreues Abbild des untersuchten Objektes, das für den Betrachter leicht zu erfassen ist.

2.2.2 Der Schnitt

Bei der optischen Mikroskopie wird oft aus dem zu untersuchenden Objekt eine dünne Scheibe herausgeschnitten. Diese Scheibe wird stellvertretend für das Objekt untersucht,

da sich eine dünne Scheibe leichter durchleuchten lässt als das ganze Objekt. Innerhalb der Probe kommt es außerdem kaum zu Überlagerungen, da sie so dünn ist.

In der Regel werden mehrere Schnitte angefertigt, um so die innere Struktur des Objektes zu bestimmen. Um den mathematischen Aufwand für Rückschlüsse aus der Position im Schnitt auf die Position in der Probe klein zu halten, werden Schnitte in der Regel parallel zu einer der Koordinatenebenen durchgeführt, sie stehen also senkrecht auf einer der Koordinatenachsen. Auf dieser Achse lässt sich die Tiefe des untersuchten Schnittes leicht ablesen. Die anderen beiden Achsen entsprechen genau denen aus dem zweidimensionalen Koordinatensystem des Schnittes.

2.3 Visualisierung mittels OpenGL

Der Einsatz von *OpenGL* auf herkömmlichen PCs spielt in der Umsetzung des Visualisierungssystems eine zentrale Rolle. Es zahlt sich daher aus diesen Bereich der Computergrafik etwas genauer zu betrachten. Kommende Überlegungen und Entscheidungen in Entwurf und Implementierung basieren auf diesen Informationen.

2.3.1 OpenGL als universelle Visualisierungsschnittstelle

OpenGL ist eine Softwareschnittstelle zu einem beliebigen grafischen Anzeigesystem, im Folgenden *Renderer* genannt. Das System folgt dabei einem Client-Server-Konzept, wobei der *Renderer* den Server darstellt und die Grafikanwendung den Client. Für die meisten gebräuchlichen Sprachen sind *OpenGL*-Funktionsbibliotheken verfügbar. Die Funktionen, die diese Bibliotheken zur Verfügung stellen, bilden die standardisierte Programmierschnittstelle (API) von *OpenGL*. Diese *OpenGL-API* wird oft vereinfachend als *OpenGL* bezeichnet. Da *OpenGL* in hohem Maße standardisiert ist, lassen sich verschiedenste *Renderer* zur Darstellung verwenden, ohne dass die Anwendung neu kompiliert werden muss.

Der *OpenGL*-Standard wird von einen Standardisierungsgremium, dem "Architecture Review Board" (ARB) gepflegt und überwacht, und die *OpenGL-API* steht auf allen üblichen Plattformen zur Verfügung. Die Referenzdokumentation, das "Bluebook" wird regelmäßig aktualisiert vom ARB herausgegeben. Zur Zeit ist die von Mark Segal und Leech [12] überarbeitete Version 1.5 am gebräuchlichsten. Anders als z.B. bei *Direct3D* wird durch die Verwendung von *OpenGL* sowohl auf Client- als auch auf Serverseite

die Bindung an einen Hersteller oder an eine Plattform vermieden. Das ARB spezifiziert genau die Vorbedingungen und die Nachbedingungen jeder API-Funktion. Es wird jedoch bewusst auf Aussagen zu ihrer Implementierung verzichtet. Inzwischen ist ein großer Teil der Funktionalität von *OpenGL* direkt in die Hardware von Grafikkarten übernommen worden. Einige Anbieter verwenden auch spezielle Multimediabefehlssätze moderner Prozessoren. *OpenGL* stellt somit eine in vielen Fällen hardwarebeschleunigte, jedoch in jedem Fall verlässliche und standardisierte Schnittstelle bereit.

2.3.2 Der Renderer

Gezeichnet wird bei der *OpenGL*-Architektur immer durch den *Renderer*, der Hauptbestandteil des Servers ist. Server und *Renderer* werden deshalb oft vereinfachend gleich gesetzt. Die meisten Renderer sind mit Grafikkarten verwoben, da das Gezeichnete in den meisten Fällen auf Bildschirmen angezeigt werden soll. Durch die Übernahme von möglichst vielen Bestandteilen eines Renderers direkt in die Hardware der Grafikkarte soll eine möglichst hohe Leistung beim Zeichnen erreicht werden. Aufgrund der hohen Komplexität der umfangreichen *OpenGL-API* ist dies sehr aufwendig, und vollständig hardwareimplementierte Renderer sind nur in wenigen teuren Spezialgrafikkarten zu finden. Die meisten Hersteller gehen statt dessen einen Kompromiss ein, indem sie seltener verwendete oder zu komplexe Teile des *Renderers* in den Treiber verlagern. Diese vor allem im Consumer-Bereich übliche Lösung hat auch den Vorteil, dass die Grafikkarte relativ leicht weitere Grafikschnittstellen unterstützen kann. Alle Grafikschnittstellen basieren auf sehr ähnlichen Grundkonzepten, so dass sie viele Funktionalitäten gemeinsam haben. Dieser gemeinsame Nenner wird von den Herstellern in ihre Hardware implementiert, während der Treiber als Mittler fungiert und Unterschiede ausgleicht.

Allgemein lassen sich *Renderer* in die zwei Gruppen *Softwarerenderer* und *Hardwarerenderer* unterteilen. Nur komplett in Software implementierte *Renderer* werden gemeinhin als *Softwarerenderer* bezeichnet. Als *Hardwarerenderer* gelten dagegen üblicherweise schon Renderer die nur den wichtigsten Teil ihrer Funktionalität in Hardware ausführen. Da die Implementation komplexer geometrischer Algorithmen in Hardware sehr viel aufwendiger ist als in Software, galt der Preis einer Grafikkarte lange Zeit als Maß des Anteils hardwareimplementierter Funktionalität in dieser Grafikkarte.

Mit der Massenproduktion im Consumer-Markt ist der Anteil der Implementationskosten für den *Renderer* und somit der Anteil der Entwicklungskosten einer Grafikkarte gemessen an den Produktkosten jedoch erheblich geschrumpft. Consumer-3D-Grafikkarten kann so trotz hoher Entwurfskosten immer mehr Funktionalität direkt in die

Hardware integriert werden. Gleichzeitig entstehen neue Prioritäten, welche Funktionalität in Hardware verfügbar sein sollte. Da Consumer-3D-Grafikkarten praktisch ausschließlich auf Spiele ausgelegt sind, wird vielen Funktionen professioneller *Renderer* zugunsten anderer, für Spiele wichtigerer Funktionen weniger Bedeutung beigemessen. Zum Beispiel werden im professionellen Bereich oft Drahtgitter und einfarbige Flächen dargestellt. Im Gegensatz dazu liegt der Fokus bei Spielen vor allem bei texturierten Flächen. Dies zeigt sich im Verhalten der verschiedenen Grafikkarten. Während Consumer-*Renderer* zum Beispiel beim Zeichnen von Drahtgittermodellen extrem schlechte Leistungen zeigen, zeichnen sie die eigentlich aufwändigeren texturierten Flächen mit enormer Geschwindigkeit.

Hardwarerenderer für den Consumer-Markt werden zur Zeit fast ausschließlich von nVidia [16] oder ATI [1] geliefert. Die von nVidia zentral angebotenen Treiber sind schon längere Zeit für verschiedene 32Bit und 64Bit Plattformen verfügbar. Im Unix-Bereich waren daher bisher vor allem nVidia-*Renderer* verbreitet. ATI verbessert jedoch seit kurzem seine bisher schwache Unterstützung von nicht-Windows-Betriebssystemen und bietet inzwischen auch 64-Bit-Treiber an.

Softwarerenderer sind meist mit dem Betriebssystem gelieferte Ersatzrenderer, die die Funktionsfähigkeit *OpenGL*-basierter Anwendungen sicherstellen sollen. Neben den herstellerspezifischen *Softwarerendereren* ist vor allem auf Unix-Systemen das unter der MIT-Lizenz [15] stehende Mesa [14] stark verbreitet.

Mesa zeichnet sich vor allem dadurch aus, dass es den Renderer nocheinmal in eine Schnittstelle (auf die die *OpenGL-API* aufsetzt) und einen ausführenden Teil unterteilt. Die softwareimplementierte Funktionalität des ausführenden Teils steht dabei als Satz leicht austauschbarer Funktionszeiger zur Verfügung. Einige "Treibermodule" für Grafikkarten, die selbst entweder keine oder nur eine unzureichende *OpenGL*-Schnittstelle haben, machen sich das zu nutze. Ist es bei einer Grafikkarte möglich, eine bestimmte Funktionalität hardwarebeschleunigt zu implementieren, dann ersetzen sie den Zeiger auf die entsprechende reine Software-Funktion durch den Zeiger auf ihre eigene, ganz oder teilweise hardwarebeschleunigte Funktion. Dies ist z. B. oft bei den für das Rastern zuständigen Funktionen möglich, da hier nur noch zweidimensional gezeichnet wird. In der Regel kann jede moderne Grafikkarte zweidimensionale Zeichenoperationen, wie z.B. das Zeichnen von Polygonen u.Ä., hardwarebeschleunigt ausführen. Mesa ist also nicht einfach nur ein *Softwarerenderer*, sondern kann auch ein Mittel sein, "schwacher" oder inkompatibler Hardware mittels Kompatibilitätsschicht entgegen zu kommen.

2.3.3 Die Architektur von OpenGL

Die *OpenGL-API* abstrahiert den *Renderer* zu einer Zustandsmaschine, die auf gegebenen Ortsvektoren aus R^3 Transformationen mittels mehrerer Matrizen anwendet. Auf diesem mathematischen Wege werden jegliche Tranformationen des zu zeichnenden Ortsvektors ($\mathbb{R}^3 \mapsto \mathbb{R}^3$) sowie perspektivische Verzerrung und Projektion auf die Projektionsfläche ($\mathbb{R}^3 \mapsto \mathbb{R}^2$) realisiert. Die Projektionsfläche kann vereinfachend mit der Oberfläche des Bildschirms gleichgesetzt werden. Da die meisten Darstellungsmedien diskrete Koordinatensysteme verwenden, werden die Ergebnisse der Projektion gegebenenfalls noch gerastert ($\mathbb{R}^2 \mapsto \mathbb{N}^2$). Die so ermittelten Vektoren aus $\mathbb{N}^2$ werden dann verwendet, um Farbinformationen in das entsprechende Darstellungsmedium zu schreiben. Diese Farbinformationen können entweder direkter oder indirekter Natur sein. Direkte Farbinformationen werden aus einem vorbereiteten n-dimensionalen Puffer, einer Textur, gelesen. Mittels, durch Texturkoordinaten bestimmten Matrixoperationen ($\mathbb{R}^n \mapsto \mathbb{R}^3$) werden diese Farbwerte dann in den Raum projiziert. Indirekte Farbinformationen sind durch lineare Funktionen interpolierte Farbwerte, wobei diese Funktionen durch den Ortsvektor parametrisiert werden. Dazu kommen noch die so genannten Shader, kleine Programme, die ähnlich einer Textur den Farbwert für jedes einzelne Pixel bestimmen. Da diese Shaderprogramme als Algorithmus für jeden einzelnen dargestellten Punkt ausgeführt werden müssen, sind sie sehr rechenzeitaufwendig und werden deshalb üblicherweise direkt vom Grafikchip ausgeführt. Die durch diese drei Methoden bestimmten Farbwerte lassen sich mittels *Multitexturing* beliebig kombinieren.

3 Allgemeiner Entwurf

3.1 Auswahl der Visualisierungsmethode

Die Visualisierung der Volumendaten ist zentraler Bestandteil des zu entwerfenden Systems, daher spielt die Auswahl der Methode der Visualisierung eine entscheidende Rolle. Im Folgenden sollen die Vor- und Nachteile der beiden vorgestellen Visualisierungsmethoden *Schnitt* und *volume rendering* betrachtet, und daraus die Geeignetere ermittelt werden.

3.1.1 Das volume rendering

Wie bereits erwähnt, werden beim *volume rendering* die dreidimensional bestimmten Daten auch wieder dreidimensional abgebildet ($\mathbb{R}^3_{reel} \mapsto \mathbb{R}^3_{virt}$). Diese "naturgetreue" Abbildung ist sehr intuitiv, da der Betrachter seine, aus der reellen Welt stammenden, Begrifflichkeiten und Denkweisen ohne Abstraktion übernehmen kann und sich so schnell "zurechtfindet".

Es existieren zwei übliche Arten des *volume renderings*. Beide werden auf Volumendaten angewendet, wobei jedem Datum ein Voxel in der dreidimensionalen Szene entspricht.

Das direkte *volume rendering* (DVR) wie es z.B. von Levoy [10] beschrieben wird, basiert auf dem *Raytracing*. Der Vorteil dieses Verfahrens ist, dass es keine Flächen benötigt, sondern die Voxelhaufen der Szene direkt darstellen kann. Es ist somit möglich transparente, oder nicht scharf abgegrenzte Körper natürlich darzustellen. Direktes *volume rendering* ist jedoch sehr aufwendig, da prinzipiell jedes einzelne Datum bzw. jeder einzelne Voxel berücksichtigt werden muss. Es gibt zwar Ansätze dies hardwarebeschleunigt auszuführen (Roettger et al. [19]), jedoch ist interaktives Arbeiten selbst dann nur mit extrem leistungsfähiger Hardware möglich. Der in dieser Arbeit geplante breite Einsatz am Arbeitsplatz wäre derzeit nicht realisierbar.

Ein weiterer Aspekt ist, dass sich bei diesem Verfahren, unabhängig vom Darstellungsmedium, immer mehrere Daten überlagern werden. Auch wenn verdeckte Daten (z.B. bei Transparenten Körpern) mit in die Darstellung einfließen findet ein Informationsverlust bzw. eine Verfälschung statt. Es kann nur "in das Objekt hineingeblickt" werden, wenn irrelevante Daten ganz oder teilweise ausgeblendet werden. Wenn also die Informationen selektiv reduziert werden, und so der "Blick" auf die relevanten Daten freigeben wird. Dazu muss selbstverständlich erst einmal bekannt sein, welche Daten irrelevant sind. Dass relevante Daten relevante Daten überlagern kann jedoch auch so nicht verhindert werden.

Die zweite Methode, das Oberflächenrendering (SF), basiert auf einer binären Segmentierung des Datenraums. Für jedes Voxel wird bestimmt, ob es Teil des Objektes ist, oder nicht. Transparente oder diffuse Objekte sind nicht möglich. Auch hier findet eine selektive Reduzierung der Daten statt. Zwischen den auf diese Weise getrennten Voxelmengen wird durch den *marching cubes*-Algorithmus [11] eine Polygonfläche erzeugt. Diese Fläche entspricht der Oberfläche des Körpers. Sie kann konventionell, also auch hardwarebeschleunigt gerendert werden. In dieser Arbeit geht es jedoch um Daten von Messungen innerer Strukturen, während die Oberfläche gänzlich uninteressant ist. Auch die zweite Art des *volume renderings* ist daher ungeeignet.

Letztendlich gibt es praktisch gesehen kein Medium für wirkliche dreidimensionale Darstellung. Auch die menschliche Netzhaut, quasi als letzte Instanz, ist nur zweidimensional. Jegliche räumliche Informationen werden vom Gehirn "interpoliert". Bei der üblichen Darstellung auf einem Bildschirm oder auf Papier gehen ohnehin sämtliche Tiefeninformation verloren. Eine verlustfreie direkte Darstellung von Volumendaten ist daher kaum möglich. Zusammenfassend kann gesagt werden, dass das *volume rendering* für die genannte Aufgabe der interaktiven Visualisierung von Volumendaten entweder zu langsam oder zu verlustreich ist.

3.1.2 Der Schnitt

Der *Schnitt* ist eine in der wissenschaftlichen Praxis bewährte und beliebte Methode. Er bietet auf relativ einfache Weise Einblick in das untersuchte Objekt. Die dabei stattfindende Reduzierung der sichtbaren "Daten" auf einen kleinen Ausschnitt kann durchaus als Reduktion der Informationen verstanden werden. Anders als beim *volume rendering* ist diese Reduktion jedoch nicht selektiv, sondern unterliegt einer einfachen linearen Formel. Das ist kein Nachteil, da, wie bereits erörtert, die selektive Reduktion ohnehin nicht zweckmäßig ist. Durch das Anfertigen mehrerer Schnitte, lassen sich die vorher

"aussortierten" Daten gezielt sukzessiv ergänzen. Hierbei werden Überlagerungen durch die Ausschließlichkeit der Methode von vornherein vermieden. Elemente, die bereits Teil eines vorher angefertigten Schnittes sind, können nicht mehr Teil des Objektes sein. Eine entsprechend geordnete Menge von Schnitten lässt sich leicht auch dreidimensional interpretieren, obwohl der einzelne Schnitt nur zweidimensionale Informationen liefert.

Das Konzept des Schnittes lässt sich leicht in die Computergrafik übertragen, indem eine gedachte Ebene durch den Datensatz und damit durch das Objekt gelegt wird. Werden anschließend alle Daten angezeigt, die auf dieser Ebene liegen, entspricht das einem virtuellen *Schnitt* durch das Objekt. Dabei kann immer nur ein Datum an einer bestimmten Stelle der Ebene liegen. Überlagerungen, und damit ein Verlust von Information bei der Darstellung werden auf diese Weise von vornherein ausgeschlossen. Der *Schnitt* eignet sich zudem besser für die Darstellung auf zweidimensionalen Medien wie Bildschirmen oder Papier.

Der Zeichenaufwand für einen *Schnitt* ist erheblich geringer als beim *volume rendering*, da nur ein Bruchteil der Daten gezeichnet wird. Zur Bestimmung der zu zeichnenden Punkte einer Ebene reicht die einfache lineare Gleichung $\vec{r} = \vec{r_0} + \lambda\vec{u} + \mu\vec{v}$ aus. Dabei beschreiben $\vec{r_0} \in \mathbb{R}^3$ den Ortsvektor eines Punktes auf der Ebene, $\vec{u} \in \mathbb{R}^3$ und $\vec{v} \in \mathbb{R}^3$ Vektoren auf der Ebene, und $\lambda \in \mathbb{R}$ und $\mu \in \mathbb{R}$ beliebige skalare Faktoren. Sämtliche Punkte, deren Ortsvektor $\vec{r} \in \mathbb{R}^3$ diese Gleichung erfüllt, gehören zu der Ebene und müssen somit gezeichnet werden.

Zwar geht auch bei diesem Verfahren die direkte Information über die Tiefe verloren, sie lässt sie sich aber relativ leicht ermitteln. Die Tiefe eines jeden angezeigten Datums entspricht genau der Tiefe seines Ortsvektors $\vec{r} \in \mathbb{R}^3$, der sich wiederum leicht mittels obiger Gleichung bestimmen lässt. Analog zum physischen Anfertigen mehrerer Schnitte, lässt sich die Schnittebene durch Anpassung der Parameter $\vec{r_0}$, $\vec{u}$ und $\vec{v}$ verlagern. Es lassen sich auch beliebig viele *Schnitte* gleichzeitig beliebig in den Datenraum legen.

Wie bereits erwähnt, ist der Schnitt in der wissenschaftlichen Praxis bekannt. Für die Zielgruppe (Mediziner) ist der *Schnitt* eine vertraute Darstellungsform. Er kann deshalb innerhalb dieses Umfeldes als mindestens so intuitiv wie *volume rendering* angesehen werden und ist gleichzeitig übersichtlicher. Folglich ist der *Schnitt* sowohl aus ergonomischer Sicht, als auch aus Sicht der technischen Umsetzung besser geeignet.

3.2 Die Nutzerschnittstelle (GUI)

Der Entwurf der Nutzerschnittstelle ist entscheidend für die praktische Anwendbarkeit und Akzeptanz eines Programms. Dies gilt besonders für Visualisierungstools. Neben der bereits besprochenen Intuitivität und Effektivität der Visualisierung an sich, ist auch die Intuitivität und Effektivität des Programms in das sie eingebettet ist, von großer Bedeutung.

3.2.1 Anforderungen

Es sollen Schnitte aus Volumendaten dargestellt werden. Aus der Praxis ist bekannt, dass meist mehrere Schnitte zur Anwendung kommen. Es ist daher sinnvoll, mehrere Schnitte gleichzeitig darzustellen. Daraus ergibt sich jedoch das Problem, dass sich mehrere Schnittebenen gegenseitig überlagern könnten, was wieder Informationsverlust zur Folge hat. Außerdem schränkt die gleichzeitige Anzeige mehrerer *Schnitte* die Übersichtlichkeit stark ein. Aus diesem Grunde ist es notwendig, dass jeder *Schnitt* seine eigene Sicht also sein eigenes gesondertes Fenster hat. Diese Sichten sollen die Möglichkeit bieten, andere Schnitte aus- bzw. einzublenden, was jedoch keinen Einfluss auf deren eigene Sicht haben darf. Eine zentrale Verwaltung aller Schnitte sowie ein Übersichtsfenster, das die Lage aller Schnitte im Raum und untereinander veranschaulicht, verbessert die Übersichtlichkeit zusätzlich. Gleichzeitig muss das gesamte System angemessen schnell sein und möglichst geringe Anforderungen an die Hardware stellen, um flüssiges interaktives Arbeiten zu ermöglichen.

3.2.2 Umsetzung des Schnittkonzeptes

Die Schnitte selbst sind zweidimensional und können damit theoretisch ohne Informationsverlust (abgesehen von Rastering) auf dem Bildschirm abgebildet werden. Dies wird erreicht, indem die Sichtlinie, also die Linie an der entlang der Nutzer auf den Schnitt blickt, als Normale der Schnittebene verwendet wird. Die Folge dieser Ausrichtung ist, dass die Schnittfläche parallel zur Projektionsfläche ausgerichtet wird.

Bestimmung der Schnittebene

Die Schnittfläche wird durch Berechnung der Ortsvektoren ihrer vier Ecken bestimmt. Dazu sind außer der Sichtlinie noch zwei weitere Parameter notwendig: der Aufspannwinkel und die Sichtsenkrechte. Der Aufspannwinkel der Sicht ist der Winkel zwischen dem oberen und dem unteren Rand des Sichtfeldes. Er beträgt in der Regel ca. 30° und ist vergleichbar mit der Brennweite einer Kamera. Die Sichtsenkrechte definiert Rollbewegungen der Sicht um die Sichtlinie. Sie zeigt aus Sicht des Betrachters per Definition immer senkrecht nach oben. Während die Rollbewegungen direkt vom Nutzer bestimmt werden, leitet sich der Aufspannwinkel von der Größe und Form des Sichtfensters sowie dem Abstand der Kamera zur Schnittfläche ab. Aus diesen drei Parametern lassen sich die Eckvektoren der Schnittfläche wie folgt berechnen:

1. Der Sichtvektor $\vec{s} \in \mathbb{R}^3$ (steht senkrecht auf der Schnittfläche und zeigt zur Kamera) hat die gleiche Richtung wie die Sichtlinie, und seine Länge wird vom Nutzer indirekt als Entfernung der Kamera vom Schnitt bestimmt.

2. Die direkt vom Nutzer durch Rollbewegungen der Kamera bestimmte Sichtsenkrechte ist der Vektor $\vec{u} \in \mathbb{R}^3$. Sie steht senkrecht auf dem Sichtvektor.

3. Der Winkel $\delta \in \mathbb{R}$ der Diagonalen des Sichtfeldes zu seiner Senkrechten wird nach der Formel $\delta = \arctan(b/h)$ berechnet. Die Breite b und die Höhe h des Sichtfeldes sind die in das Koordinatensystem des Raumes umgerechneten Maße des Sichtfensters.

4. Die Länge der Hypotenuse des dabei Aufgespannten Dreiecks $c \in \mathbb{R}$ ergibt sich aus $c = \frac{\sin(\alpha)}{\cos(\delta)} * |s|$, wobei $\alpha \in \mathbb{R}$ der Aufspannwinkel der Sicht ist.

5. Die Eckvektoren der Schnittfläche lassen sich nun durch Multiplikation der normierten Sichtsenkrechte mit c und der entsprechenden Rotation um die Sichtlinie ermitteln. Dies geschieht nach folgender Vorschrift.

$$\vec{c} = \begin{pmatrix} \cos(360° - \delta) \\ \cos(\delta) \\ \cos(180° - \delta) \\ \cos(180° + \delta) \end{pmatrix}, \vec{c'} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} - \vec{c} \text{ und } \vec{s} = \begin{pmatrix} \sin(360° - \delta) \\ \sin(\delta) \\ \sin(180° - \delta) \\ \sin(180° + \delta) \end{pmatrix}$$

mit $\vec{c} \in \mathbb{R}^4$, $\vec{c'} \in \mathbb{R}^4$ und $\vec{s} \in \mathbb{R}^4$ in

$$\vec{e_n} = \begin{pmatrix} s_x^2 c'_n + c_n & s_y s_x c'_n + s_z s_n & s_z s_x c'_n - s_y s_n \\ s_x s_y c'_n - s_z s_n & s_y^2 c'_n + c_n & s_z s_y c'_n - s_x s_n \\ s_x s_z c'_n + s_y s_n & s_y s_z c'_n - s_x s_n & s_z^2 c'_n + c_n \end{pmatrix} \cdot \vec{u}$$

Dabei bestimmt $\vec{e_n} \in \mathbb{R}^3$ den Ortsvektor der n-ten Ecke der Schnittfläche (es gilt $0 \leq n < 4$ mit $n \in \mathbb{N}$). Zu beachten ist, dass aus dem Sichtfeld nur die indirekten Größen Länge und Winkel der (Fenster-)Diagonalen zur Bestimmung der Diagonalen der Schnittfläche verwendet werden. Die Senkrechte des Sichtfensters stimmt nicht gezwungenermaßen mit der Sichtsenkrechten überein, sie dient nur der Bestimmung des Winkels δ. Die Schnittfläche wird wie beschrieben ausschließlich durch die Rotation der Sichtsenkrechten um den Sichtvektor geformt. Die Werte c und δ sind dabei lediglich Parameter, die dafür sorgen, dass Sichtfenster und Schnittfläche die gleiche Form haben.

Abb. 3.1 verdeutlicht die Wirkung dieses Vorgehens. Das linke Bild zeigt den *Schnitt*, wie er in der Schnittsicht dargestellt wird. Das rechte Bild zeigt den Überblick, der den *Schnitt* selbst, und seine Lage im (Daten)Raum visualisiert.

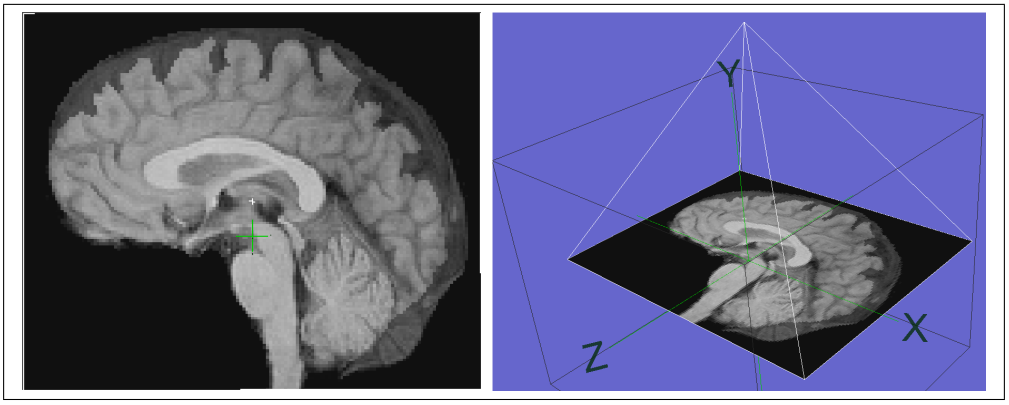

Abb. 3.1: Einzelner *Schnitt* mit Übersicht

Der Betrachter blickt, wie im linken Bild zu erkennen, immer senkrecht auf den *Schnitt* beziehungsweise der Bildschirm liegt zu jedem Zeitpunkt parallel zur Projektionsfläche. Jegliche perspektivische Verzerrungen werden auf diese Weise vermieden. Die physische Analogie dazu ist eine Kamera, vor die ein Schirm montiert ist, der genau das Sichtfeld der Kamera abdeckt. Wird die Kamera bewegt, bewegt sich der Schirm mit. Die relative Lage der Kamera zum Schirm ändert sich daher nicht. Die Schnittebene übernimmt die Rolle des Schirms. Wenn der Betrachter seine Sicht mittels Maus bewegt, bewegt er die Schnittebene, also den *Schnitt* durch den Raum der Volumendaten.

Die Schnittsicht erscheint immer zweidimensional, lediglich die Übersicht verdeutlicht die Lage des *Schnittes* im Datenraum. Es findet eine strenge Trennung zwischen den quasi zweidimensionalen primären Informationen des *Schnittes* und seinen dreidimensionalen sekundären Eigenschaften, wie zum Beispiel seiner Lage, statt. Dies erhöht die

Übersichtlichkeit, da der Anwender beim Arbeiten in der Schnittsicht nicht durch "unwesentliche" Informationen abgelenkt wird. Um ggf. die Lage des *Schnittes* oder andere sekundäre Informationen in Erfahrung zu bringen, reicht ein Blick auf das Übersichtsfenster.

Abbildung der Volumendaten auf dem Schnitt

Welches Datum aus dem Datenraum an welcher Position auf dem *Schnitt* gezeichnet wird, hängt von dessen Lage ab und wird durch die schon erwähnte Formel $\vec{r} = \vec{r_0} + \lambda\vec{u} + \mu\vec{v}$ bestimmt. Der *Schnitt* wird in Wirklichkeit nicht durch eine unendliche Schnittebene, sondern durch ein rechteckiges Schnittpolygon realisiert. Die Skalierungsfaktoren λ und μ der obigen Ebenengleichung sind also nicht beliebig. Ihre Wertebereiche werden durch die Abmessungen des Polygons bestimmt. Dieses Abmessungen wiederum werden so angepasst, dass das Polygon die Schnittsicht genau ausfüllt.

Bewegt sich der Betrachter im GL-Raum, so ändert er nicht seine Sicht auf die Schnittebene, denn die bewegt sich immer entsprechend mit. Stattdessen ändert er die Lage des Schnittpolygons im Datenraum, und damit die Parameter der obigen Formel. Auf diesem Wege bestimmt er welche Daten aus dem Datenraum, d.h. dem Volumendatensatz, dargestellt werden. Die Berechnung ist zwar schon um einiges einfacher als beim *volume rendering* aber immernoch recht umfangreich. Sie lässt sich jedoch auf den *Renderer* übertragen, der sie meist effektiver bewältigen kann.

OpenGL zeichnet Flächen polygonorientiert. Nur die Punkte des GL-Raumes, die zum zu zeichnenden Polygon gehören, für die also obige Gleichung zutrifft, werden überhaupt beim Zeichnen berücksichtigt. Wird einer dieser Punkte gezeichnet, kann die verwendete Farbe wie schon erklärt unter Anderem aus einem n-dimensionalen Puffer bestimmt werden. Es bietet sich also an, die dreidimensionalen Messdaten direkt in einem solchen Puffer abzulegen. Aus den Texturkoordinaten des zu zeichnenden Punktes $\vec{r} \in \mathbb{R}^3$ bestimmt der *Renderer* mittels einer einfachen Transformation $\mathbb{R} \mapsto \mathbb{N}$ den Index des Farbwerteintrages im Texturpuffer. Die Texturkoordinaten eines Punktes werden Analog zu seinen Raumkoordinaten über die Formel $r_t\vec{e}x = r_{0_t}\vec{e}x + \lambda\vec{u} + \mu\vec{v}$ bestimmt. Das Schnittpolygon existiert daher nicht nur im GL-Raum, sondern gleichzeitig auch im Texturraum des *Renderers* und damit im Datenraum der in den Texturspeicher geladenen Messdaten. Um die Lage des Schnittpolygons im Texturraum zu bestimmen, werden seinen Eckpunkten zusätzlich zu den Raumkoordinaten Texturkoordinaten zugewiesen. Je nach Lage des Schnittpolygon im Texturraum werden die Daten bzw. Farbwerte aus

dem Daten- bzw. Texturraum auf dem Schnittpolygon abgebildet, die es gerade darin "schneidet".

Durch Gleichsetzen des Koordinatensystems des GL-Raumes mit dem des Texturraumes lassen sich Texturkoordinaten ohne Transformation im GL-Raum abbilden und anders herum. Es existiert daher ein virtueller Datenraum innerhalb des GL-Raumes. Dieser Datenraum wird in der Übersicht von Abb. 3.1 auf Seite 21 durch den schwarzen Quader angedeutet, wirklich sichtbar können aber nur die Volumendaten werden, die auf einer Schnittebene liegen.

Sämtliche angeführten Rechnungen werden vom *Renderer*, also im Idealfall hardwarebeschleunigt, ausgeführt. Der *Renderer* bestimmt die zu zeichnenden Punkte. Er ermittelt den Index der Farbwerte für das Zeichnen und greift direkt auf diese Werte (die Messwerte) im Texturspeicher zu. Nur die Raum- und Texturkoordinaten der Eckpunkte des Schnittes müssen von der Anwendung bestimmt werden. Da GL- und Texturraum gleichgesetzt sind, lassen sich die Raumkoordinaten auch als Texturkoordinaten werwenden. Die vier GL-Raum-Koordinaten des Schnittpolygons können wie in Abschnitt 3.2.2 auf Seite 20 beschrieben, ermittelt werden. Durch die Reduzierung der nötigen anwendungsseitigen Operationen auf die Bestimmung dieser Eckpunkte lässt sich die Darstellung des *Schnittes* sehr schnell und ressourcenschonend ausführen.

Mehrere Schnitte

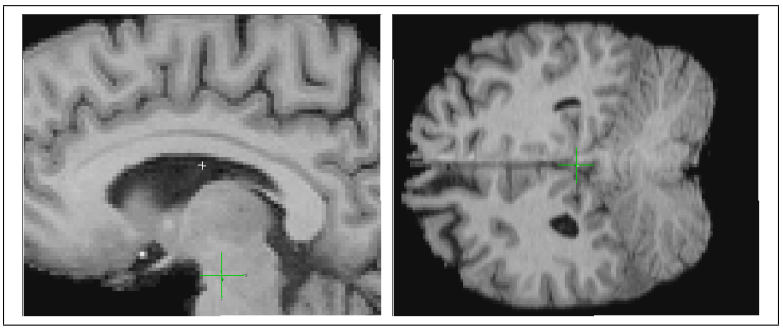

Abb. 3.2: Schnitte A und B

Abb. 3.1 auf Seite 21 stellt mit zwei Fenstern nur die einfachste Variante der Visualisierung dar. Wie in den Anforderungen beschrieben, wird das System beliebig viele Schnitte gleichzeitig in mehreren Sichten darstellen können. Abb. 3.2 zeigt zum Beispiel die Verwendung zweier Schnitte. Jedem Schnitt wird dabei eine Schnittsicht zugeordnet.

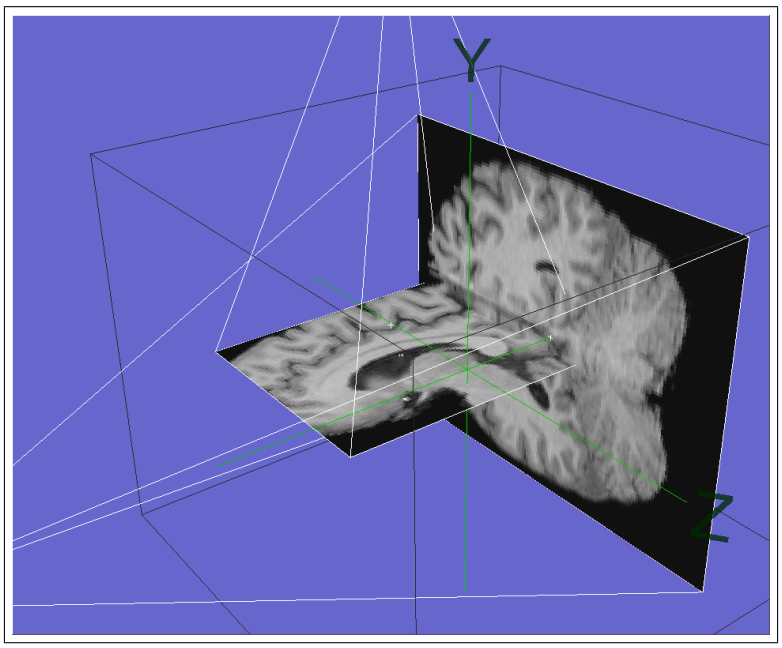

Abb. 3.3: Übersicht zu den zwei Schnitten A und B

Die Schnitte (Abb. 3.2 auf der vorherigen Seite) können unabhängig voneinander positioniert werden. Jede Schnittsicht zeigt nur ihre eigene Schnittebene, andere Schnitte sind ausgeblendet. Die dazugehörige Übersicht (Abb. 3.3) zeigt dagegen beide Schnitte, ihre Lage zueinander, sowie ihre Lage innerhalb des Datenraumes.

Aus der Verwendung mehrerer Fenster, und damit mehrerer Instanzen des Renderers ergibt sich zusätzlicher Verwaltungsaufwand. Die Änderung der Lage einer Schnittfläche muss allen Instanzen des Renderers, die diese Schnittebene darstellen, mitgeteilt werden. Dies betrifft mindestens die ihr zugeordnete Sicht und das Übersichtsfenster.

Zudem müssten sich alle Instanzen die Volumendaten teilen, damit diese große Datenmenge nicht unnötig vervielfacht werden muss. Auch die Schnittfläche selbst darf bei der Anzeige in einem zusätzlichen Fenster möglichst nicht kopiert werden. Ihre Parameter werden bei der Anzeige in einer zusätzlichen Instanz nicht verändert. Eine Kopie, die als weitere Schnittfläche an den Renderer übermittelt werden müsste, wird daher eigentlich nicht benötigt. *OpenGL* bietet dazu die Möglichkeit, dass sich mehrere Instanzen des Renderers interne Daten wie Texturen und zu zeichnende Objekte untereinander teilen. Ihre anwendungsseitigen Container fallen aber wie alle anderen gemeinsam verwendeten Daten in den Verantwortungsbereich der Anwendung. In der Implementation müssen daher Mittel zum Teilen von Ressourcen und zur Kommunikation integriert werden.

3.2.3 Der Cursor

Als Cursor wird in der IT allgemein die Bearbeitungsposition innerhalb einer Menge von Daten bezeichnet. Dies gilt auch für die hier verwendeten Volumendaten. Der Cursor liegt naturgemäß meist im Zentrum der Betrachtung, und kann so effektiv besonders veränderliche Informationen übermitteln. Dabei sollte er aber auch nicht überladen werden.

Positionsbestimmung

Die wichtigste Information, die ein Cursor darstellt ist die der Position. Visualisierungssysteme verwenden meist die Position des Mauszeigers im Darstellungsfenster, um daraus die Position des Cursors zu ermitteln. Die Übertragung von Bildschirmkoordinaten ist jedoch bei dreidimensionalen Systemen nicht so trivial wie bei zweidimensionalen Systemen. Das Problem liegt dabei darin begründet, dass die bei der Projektion des Raumes auf den Bildschirm ($\mathbb{R}^3 \mapsto \mathbb{N}^2$) verloren gegangene Tiefeninformation bei der umgekehrten Projektion der Bildschirmkoordinaten in den Raum ($\mathbb{N}^2 \mapsto \mathbb{R}^3$) fehlt. Aus den Bildschirmkoordinaten lässt sich die Raumkoordinate daher nicht eindeutig bestimmen, vielmehr liefert $\mathbb{N}^2 \mapsto \mathbb{R}^3$ einen senkrecht auf der Projektionsfläche stehenden Strahl von möglichen Koordinaten. *OpenGL* bietet zwar eine entsprechende Funktion zur umgekehrten Projektion an, dieser muss aber eine Tiefeninformation in der Form $z = \frac{a}{b}$ übergeben werden. Dabei sind a der Abstand des gesuchten Punktes von der Projektionsebene, und b der Abstand des Horizonts von der Projektionsebene. Die Maßzahl $z \in \mathbb{R}$ hat dabei nichts mit der z-Achse des Raumes zu tun, sie gibt lediglich ein Längenverhältnis an, und bestimmt so einen Punkt auf dem erwähnten Strahl. Für z sind also nur Werte zwischen 0 und 1 sinnvoll. Ein $z > 1$ würde bedeuten, dass der gesuchte Punkt hinter dem Horizont liegt, und ein $z < 0$, dass er vor der Projektionsfläche liegt. In beiden Fällen ist er nicht sichtbar und damit nutzlos. Die Bestimmung dieses Verhältnisses bleibt Aufgabe der Anwendung, und stellt damit das eigentliche Problem bei der Übertragung von Bildschirmkoordinaten in den Raum dar.

Durch die Verwendung des Schnittkonzeptes ist die Bestimmung der Verhältnisszahl z relativ einfach. Der Nutzer bewegt sich zwar im Grunde frei im Raum, jedoch bleibt er immer auf der Schnittebene. Da nur die auf der Schnittebene liegende Volumendaten sichtbar werden, sind nur die auf der Schnittebene liegende Koordinaten für den Anwender von Interesse. Der gesuchte Wert a (b ist vorgegeben) entspricht also genau dem Abstand der Projektionsebene am ausgewählten Punkt zur Schnittebene. Auch a

ist weitestgehend konstant, da die Schnittebene, wie bereits dargelegt, parallel zur Projektionsebene liegt, und nur selten ihren Abstand zu dieser ändert. Auf diese Weise kann auch bei dieser Operation die eigentliche Rechenlast (die Projektion $\mathbb{N}^2 \mapsto \mathbb{R}^3$) auf den Renderer übertragen werden, während die dazu nötige Verhältniszahl z weitestgehend von Konstanten abhängt, und deshalb nur selten bestimmt werden muss.

Darstellung

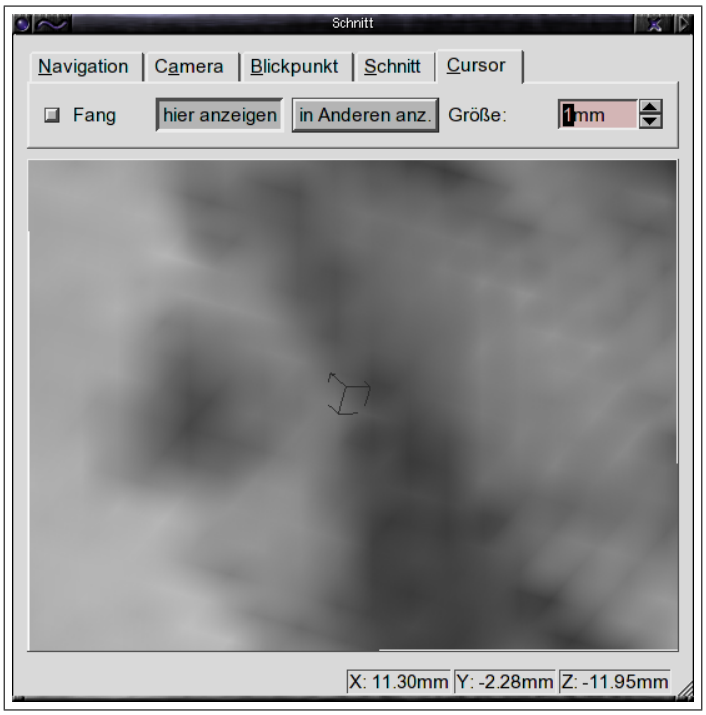

Abb. 3.4: Beispiel eines Cursors

Cursor werden in der Regel als Trennmarkierung zwischen einzelnen Dateneinheiten, oder als Markierung einer einzelnen Dateneinheit dargestellt. Bei Visualisierungssystemen ist der Cursor oft ein schlichtes Fadenkreuz. Das hier entworfene System verwendet jedoch eine räumliche Darstellung, bei der weder die Betrachtungsrichtung, noch die Skalierung offensichtlich sind. Es liegt daher nahe, dem Cursor noch zusätzlich zur Position Informationen über Lage und Skalierung mitzugeben. Dazu wird der Cursor als Würfel entworfen, dessen Kantenlänge einen bekannten Wert, z.B. eine Einheit im Datenraum, hat. Auf diese Weise fungiert er gleichzeitig als Maßstab, als Markierung des aktuell ausgewählten Datenelementes und auch als Abgrenzung zu den Nachbarn dieses

Elementes. Wird der Würfel fest an den Koordinatenachsen des Raumes ausgerichtet, kann er außerdem die eventuelle "Schieflage" des betrachteten Schnittes durch seine eigene "Schieflage" (im Verhältnis zum Schnitt) andeuten. Abb. 3.4 auf der vorherigen Seite zeigt einen solchen Cursor in der Mitte der Anzeige eines schief im Raum liegenden Schnittes.

Die Position des Cursors im Raum, und damit auch im Datenraum) wird in diesem Beispiel in der Statuszeile angezeigt. Der Cursor selbst liegt schief und verdeutlicht so die "Schieflage" des Schnittes. Der Schnitt selbst ist stark vergrößert dargestellt, was auch durch die relative Größe des mit einem Millimeter Kantenlänge eigentlich sehr kleinen Cursors erkennbar ist.

3.3 Segmentierung der Volumendaten

Wie in der Zielsetzung beschrieben, soll das Visualisierungssystem die Möglichkeit zur interaktiven Objekterkennung bieten. Dabei sollen sich Anwender und Rechner gegenseitig unterstützen. Indem dem Rechner höhere Erkennungsaufgaben abgenommen werden, lässt sich die notwendige Rechenzeit reduzieren. Außerdem wird das Erkennungssystem auf diese Weise flexibler. Es lässt sich somit auf eine breitere Auswahl von Eingabedaten anwenden und reagiert toleranter auf unerwartete Werte. Die, der Interaktiven Segmentierung vorangestellte automatische Aufbereitung besteht in einer Vorsegmentierung der Daten. Dabei sucht der Rechner kleinere Strukturen, die auch automatisch sicher zu erkennen sind, und markiert sie. Bei der, darauf folgenden Interaktiven Segmentierung sollen diese vorher erkannten Unterobjekte vom Nutzer sinnvoll zusammengesetzt werden.

3.3.1 Die automatische Vorsegmentierung

Für die Vorsegmentierung der Daten wird die von Digabel und Lantuéjoul [6] entwickelte *watershed*-Transformation verwendet werden. Sie interpretiert zweidimensionale Bilder als zweidimensionale Graphen, und trennt in den daraus gebildeten Wertgebirgen die "Senken" und "Gipfel". Soll die *watershed*-Transformation auf dreidimensionale Bilder, also Volumendaten, angewendet werden, müssten lediglich die verwendeten zweidimensionalen Graphen durch eine dritte Dimension erweitert werden. Obwohl sie ursprünglich für zweidimensionale Bilddaten entworfen wurde, lässt sich *watershed*-Transformation auf diese Weise leicht auf dreidimensionale Bilder übertragen. Es existiert ein große

Anzahl von Algorithmen, die eine *watershed*-Transformation umsetzen. Für diese Arbeit wurde ein rekursiver Wurzelsuchalgorithmus nach Meijster und Roerdink [13] ausgewählt. Dieser Algorithmus bietet eine Kostenfunktion, die benötigt wird, wenn der Graph eine erhöhte Konnektivität erhalten soll. Diese Problematik wird in Kapitel 6 auf Seite 54 ausführlich behandelt werden. Die Wurzelsuche lässt sich außerdem leicht parallelisieren, und kann somit von zukünftigen *dual-core*-PC-Systemen profitieren. Das Ergebnis dieser Transformation sind dreidimensionale Bilder, in denen alle Punkte, die zu einem Unterobjekt gehören, den gleichen Wert (Index) haben. Diese Strukturen müssen im Rahmen der folgenden interaktiven Segmentierung vom Nutzer zusammengefügt werden.

3.3.2 Visualisierung der interaktiven Segmentierung

Bei der Interaktiven Segmentierung spielt das Visualisierungssystem eine wichtige Rolle, denn der Nutzer muss interaktiv mitverfolgen können, welches Unterobjekt er gerade ausgewählt hat, wo dieses Objekt liegt, und wie es aussieht. Daher muss das ausgewählte Unterobjekt dreidimensional dargestellt werden. Der Schnitt als Primäre "Arbeitsfläche" des Nutzers und Domäne des Cursors ist jedoch prinzipiell zweidimensional. Das gewählte Unterobjekt ließe sich zwar trotzdem dreidimensional darin zeichnen, jedoch würde es einen Teil des Schnittes verdecken. Ein Unterobjekt wird aktiv, wenn der Cursor sich in ihm befindet. Folglich währe dieser zu jeden Zeitpunkt von der Oberfläche des aktiven Unterobjektes verdeckt. Aus diesen Gründen wird die Darstellung des aktiven Unterobjektes in die Übersicht ausgelagert. In der Schnittansicht wird das Unterobjekt dagegen lediglich angedeutet. Die vom Nutzer zusammengesetzten Objekte werden ausschließlich in der Übersicht dargestellt.

Diese Visualisierung der Segmentierung muss ausreichend schnell erfolgen, um flüssiges Arbeiten zu ermöglichen. Aufgrund der Vorbereitung der Daten durch die Vorsegmentierung muss der Rechner während der eigentlichen Segmentierung nur noch das vom Nutzer ausgewählte Unterobjekt aus dem Ergebnisspool der Vorsegmentierung auswählen und darstellen. Werden die Darstellungen der Unterobjekte gepuffert (die Unterobjekte ändern sich nie), lässt sich die Visualisierung der Segmentierung lässt sich daher problemlos mit der geforderten Geschwindigkeit umsetzen. Dessenungeachtet ist auch hier ein effektives System für den Datenaustausch zwischen den Sichten notwendig ist.

4 Implementation der Basisbibliothek

4.1 Vorbemerkungen zur Basisbibliothek

Ziel dieser Diplomarbeit ist die Entwicklung eines möglichst praxisnahen Visualisierungstools. Da sich in der Praxis die Anforderungen oft ändern, sollte dieses Programm leicht zu pflegen und zu erweitern sein.

Visualisierungssysteme sind naturgemäß äußerst komplex und umfangreich. Das erschwert deren Pflege und jegliche nachträgliche Anpassungen wird zu einer potenziellen Fehlerquelle. Jedoch fällt beim Betrachten der Anforderungen an solche Systeme auf, dass viele grundlegende Probleme und Aufgaben regelmäßig in gleicher, oder leicht abgewandelter Form wiederkehren. So benötigen z.B. alle 3D-Visualisierungssysteme eine Kamera, Texturen und grundlegende geometrische Objekte wie etwa Flächen. All diese Programme müssen außerdem sowohl mit dem Nutzer, als auch mit ihrer Umgebung interagieren. Es ist daher sinnvoll, zuerst ein möglichst allgemeines Zeichensystem zu entwerfen, das diese Grundvoraussetzungen erfüllt. Dieses System wird in einer Basisbibliothek implementiert, und steht auf diese Weise darauf aufbauenden Systemen zur Verfügung. Ausgehend von dieser Grundlage lassen sich dann leicht Lösungen für einen konkreten Fall ableiten. Die Basisbibliothek wird objektorientiert implementiert. Spätere Spezialisierung sind daher problemlos über das Mittel der Vererbung zu erreichen. Da die Basisbibliothek die Grundlage dieser und zukünftiger Visualisierungsarbeiten darstellen und dabei möglichst breit einsetzbat sein soll, muss ihr Entwurf besonders gründlich und umfassend erfolgen.

4.1.1 Gemeinsam genutzte Ressourcen

Im Entwurf hat sich der Bedarf nach einer effektiven Ressourcenteilung unter den verschiedenen Objekten gezeigt. Einerseits, um Speicherplatz zu sparen, und andererseits, um Änderungen an einem mehrfach verwendeten Objekt nicht unnötig bei dessen Kopien

reproduzieren zu müssen. Die einfachste Methode dies zu erreichen, ist die Verwendung ein und des selben Speicherbereiches für mehrere Objekte.

Das Zeigerproblem

In den Sprachen C und C++ werden zum gleichzeitigen Verwenden des selben Speicherbereiches üblicherweise *Zeiger* verwendet. *Zeiger* speichern statt der eigentlichen Daten nur die Adresse, an der sich die Daten im Speicher befinden. Auf diesem Wege kann ohne weiteren Aufwand von jedem beliebigen Teil eines Programms aus auf die selben Daten zugegriffen werden. Dieses Konzept ist zwar sehr effektiv, da es sich an der Hardware orientiert, führt aber unter Umständen zu großen Problemen. Wird der Gültigkeitsbereich einer Variablen verlassen, wird der für sie reservierte Speicherbereich wieder freigegeben. Die Variable gilt damit als gelöscht. Im Normalfall stellt dies kein Problem dar, da die Variable per Definition die Einzige war, die auf diese Daten zugriff. Da die Variable nicht mehr existiert, werden die gelöschten Daten garantiert nicht mehr benötigt. Verlässt ein Zeiger seinen Gültigkeitsbereich, wäre es theoretisch kein Problem den Speicherbereich auf den er "zeigt" freizugeben. Zu diesem Zeitpunkt ist aber nicht bekannt, ob dieser Speicherbereich nicht noch von einem anderen Zeiger, oder einer regulären Variablen verwendet wird. Der von einem Zeiger verwendete Speicher kann deshalb nur von Hand freigegeben werden, wenn sichergestellt wurde, dass er nicht mehr verwendet wird. Es ist offensichtlich, dass das sehr schwierig werden kann. Gerade in komplexen Systemen geht die Übersicht über die Objekte im Speicher schnell verloren. Oft ist es für den Entwickler schlicht unmöglich zu garantieren, dass ein bestimmter von Zeigern verwendeter Speicherbereich gelöscht werden kann. Dabei kann das Freigeben noch verwendeten Speichers zu schweren Fehlern führen.

Intelligente Zeiger

Der Vorteil regulärer Variablen ist die automatische Freigabe des belegten Speichers bei Verlassen des Gültigkeitsbereiches. Der Vorteil von Zeigern liegt dagegen in der Möglichkeit einen Speicherbereich mehrfach zu verwenden. Diese scheinbar gegensätzlichen Eigenschaften vereinen intelligente Zeiger.

Intelligente Zeiger sind Objekte, die als reguläre Variable erzeugt werden. Wie andere Variablen werden sie bei Verlassen ihres Kontextes automatisch gelöscht. Den eigentlichen Zeiger halten sie als Parameter. Wird ein solcher intelligenter Zeiger kopiert, wird nicht nur der eigentliche Zeiger, also die Speicheradresse, kopiert. Es wird außerdem ein

interner Referenzzähler für diese Speicheradresse erhöht. Verlässt einer der Beiden seinen Gültigkeitsbereich, so kann sein Destruktor über diesen Referenzzähler leicht ermitteln, ob der von diesem intelligenten Zeiger verwendete Speicherbereich noch von Anderen intelligenten Zeigern verwendet wird. Ist dies der Fall, wird lediglich der Referenzzähler um eins decrementiert. Damit dieses System funktioniert, müssen sich die intelligenten Zeiger den Referenzzähler teilen. Ist der zu löschende intelligente Zeiger dagegen der Einzige, der diesen Speicherbereich verwendet, gibt er ihn frei. Zudem existieren noch einige syntaktische Mittel, welche die Arbeit mit intelligenten Zeigern einfacher und sicherer gestalten. Die für normale Zeiger in C und C++ üblichen Operatoren zur Dereferenzierung (`*` und `->`) können durch entsprechende eigene Funktionen der Klasse *intelligenter Zeiger* überladen werden, um die "Dereferenzierung" der intelligenten Zeiger zu vereinfachen. Desweiteren besteht das Interface dieser Klasse aus zwei Konstruktoren. Der Erste erwartet einen konventionellen Zeiger und ihm sollte ausschließlich der Rückgabewert einer Objekterzeugung (`new`-Operation) übergeben werden. Der zweite Konstruktor ist der Kopierkonstruktor. Auch er muss überladen werden, da beim Kopieren der Referenzzähler erhöht werden muss. Natürlich darf nicht von außen auf interne Parameter des intelligenten Zeigers zugegriffen werden können. Die einzige Stelle des Interfaces, an der noch konventionelle Zeiger auftreten, ist beim Erzeugen eines neuen referenzierten Objektes, also dem ersten Konstruktor. Die `new`-Operation, und damit das Erzeugen eines neuen Referenzierten Objektes im Speicher ließe sich auch direkt in den Konstruktor des intelligenten Zeigers verlagern. Dies wird jedoch dadurch erschwert, dass die Parameter, die der Konstruktor des referenzierten Objektes erwartet, unbekannt sind. Aus diesem Grund wird auf diese Möglichkeit verzichtet. Es reicht an dieser Stelle auch aus, einfach "aufzupassen". Die genannten Anforderungen, und das hier entworfene Verhalten basieren auf den Arbeiten von Ellis und Detlefs [7] sowie Colvin [5]. Sie werden von dem Datentyp `shared_ptr` umgesetzt. Diese generative Objektklasse aus den Boost-Bibliotheken [3] wird helfen im Verlauf dieser Arbeit ein robustes und leicht handhabbares System zur gemeinsamen Benutzung von Speicherbereichen zu implementieren.

4.1.2 Kommunikation

Eine weitere, beim Entwurf deutlich gewordene Anforderung ist die Möglichkeit zur Kommunikation zwischen einzelnen Teilen des Programms. Darunter wird allgemein die Möglichkeit eines Programmteiles verstanden, unter bestimmten Umständen die Ausführung von Code in einem "entfernten" anderen Teil des Programms auszulösen. Also den Prozessor zu veranlassen, an die Entsprechende Stelle im Programm und nach

Ausführung des Codes wieder zurück zu springen. Auch wenn die verwendete Programmiersprache C++ kein explizit dafür ausgelegtes Mittel kennt, gibt es unter Anderem die folgenden Möglichkeiten das beschriebene Verhalten mit allgemeineren Mitteln zu modellieren.

Polling gemeinsamer Variablen

Die einfachste Methode zur Kommunikation zwischen Programmteilen besteht darin, die zu übermittelnde Botschaft in einem gemeinsam verwendeten Speicherbereich abzulegen. Hat der Empfänger die Nachricht gelesen, löscht er sie in der Regel, um zu verhindern, dass er sie versehentlich doppelt liest. Das Problem dabei ist, dass eine Nachricht dadurch nur einmal gelesen werden kann. die Versendung an mehrere Empfänger ist somit ausgeschlossen. Weiterhin können die potentiellen Empfänger nicht wissen, ob eine Nachricht für sie vorliegt. Sie müssen dies in regelmäßigen Zeitintervallen selbst prüfen. Dies ist meist so implementiert, dass alle potentiellen Empfänger von einem zentralen System regelmäßig dazu veranlasst werden, die Prüfung und ggf. den der Nachricht entsprechenden Code auszuführen. Diese als *Polling* bezeichneten Operationen kosten zusätzlich Prozessorzeit und verursachen auch zusätzlichen Verwaltungsaufwand, da eine Zentrale Verwaltungsstelle für die Kommunikation nötig wird.

Triggering mittels Callbacks

Die Programmiersprache C ermöglicht neben Zeigern auf Daten auch Zeiger auf Funktionen. Durch dieses System ist es beispielsweise möglich einer Funktion den Zeiger auf eine Nachrichtenfunktion mitzugeben, die diese dann nach Belieben aufrufen kann. Auf diese Weise kann sie bei ihrem Aufrufer "zurückrufen". Die dabei verwendeten Funktion werden deshalb auch als *Callback*-Funktionen bzw. als *Callbacks* bezeichnet. Übergibt ein potentieller Empfänger einem Sender eine vom Empfänger bestimmte Nachrichtenfunktion, kann der Sender sie nach Belieben aufrufen. Der Sender führt daher vom Empfänger bestimmten Code aus. Im Gegensatz zum *Polling* von Nachrichten bestimmt hier der Sender, wann der Code ausgeführt wird. Da der Sender "weiß", wann eine Nachricht vorliegt, wird nur dann Code ausgeführt, wenn es auch wirklich nötig ist. Unter C++ werden *Callbacks* oft als Objektklassen mit überladenem `()`-Operator realisiert. Dadurch verhalten sie sich syntaktisch wie Funktionen, obwohl sie im Sinne der Sprache Objekte sind. Aus diesem Grunde können *Callbacks*-Objekte Parameter wie zum Beispiel einen Zeiger auf den Empfänger halten. Obwohl *Callbacks* sehr effektiv sind, werden sie bei

größeren Systemen selten eingesetzt, denn Zeiger auf Funktionen leiden unter ähnlichen Problemen wie Zeiger auf Daten. Die Callbackfunktionen selbst werden zwar nur äußerst selten gelöscht (Programmcode wird nur unter besonderen Bedingungen freigegeben), aber es kann nicht garantiert werden, dass der Empfänger bzw. seine Daten noch existieren. Die Aufrufe des *Callbacks* selbst könnten demnach ins "Leere" laufen und u. U. sogar Schutzverletzungen auslösen. Das gleiche gilt für *Callback*-Objekte wenn diese ihre Empfänger nicht überwachen. Zudem muss der Sender die *Callback*-Objekte verwalten. Er muss also über sämtliche *Callback*-Objekt-Klassen informiert sein und ihre Instanzen sinnvoll verwalten können.

Triggering mittels Signal-Slot-Architektur

Die *Signal-Slot*-Architektur stellt eine formalisierte Erweiterung von *Callback*-Objekten dar. Der *Slot* ist dabei ein *Callback*-Objekt mit einigen zusätzlich vereinbarten Eigenschaften. Er wird üblicherweise als Parameter-Objekt des Empfängers implementiert, so dass er automatisch gelöscht wird, wenn der Empfänger gelöscht wird. Durch einen überladenen Destruktor kann der *Slot* sich so bei seinem *Signal* abmelden. Das verhindert, dass Empfänger angesprochen werden, die nicht mehr existieren. Aufgebaut wird eine *Signal-Slot*-Verbindung indem der Empfänger seinen entsprechenden *Slot* bei dem *Signal*-Objekt des Senders anmeldet. *Signal*-Objekte werden ähnlich wie *Slot*-Objekte in der Regel als Parameter des Senders implementiert, um auch hier sicherzustellen, dass der Sender sich gegebenenfalls bei den, mit ihm verbundenen, *Slots* abmeldet. Ausgelöst wird das *Signal* über den Aufruf seines überladenen `()`-Operators. Dieser ruft wiederum die `()`-Operatoren aller registrierten *Slots* auf. Diese lösen dann die gewünschten Aktionen im Empfänger aus. Die Vorteile diese Konzeptes liegen darin, dass Empfänger und Sender nichts voneinander wissen müssen, die Verbindung aber dennoch stabil ist. Wie bei einfachen *Callbacks* kann die Kommunikationsverbindung zur Laufzeit beliebig hergestellt und wieder getrennt werden. Dabei fungieren die *Signal*- bzw. *Slot*-Objekt als standardisierte Vermittler, so dass beliebige Objekte "verbunden" werden können. Auch die Verbindung mehrerer Empfänger bzw. deren *Slots* mit einem *Signal* stellt kein Problem dar, da das *Signal*-Objekt die ihm zugeordneten *Slots* in einer dynamischen Liste halten kann.

Neben der Quasi-Standardimplementation der Boost-Bibliotheken [4] gehört die Umsetzung in der *Qt-Bibliothek* zu den verbreitetsten *Signal-Slot*-Systemen für C++. *Qt* verwendet allerdings ein weniger formalisiertes Konzept, das auf einem *Precompiler*, Makros und einer zentralen Verwaltung von Sendern und Empfängern basiert. Außer

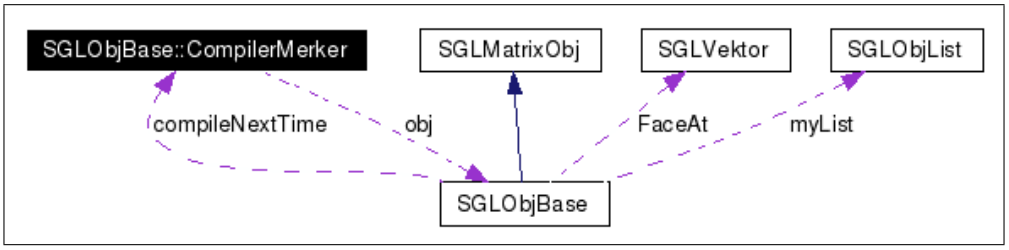

Abb. 4.1: Der *Slot* `SGLObjBase::CompilerMerker` und seine Beziehungen

für die Implementation der Nutzerschnittstelle wird in dieser Arbeit ausschließlich die Boost-Implementation Anwendung finden. Abb. 4.1 zeigt zum Beispiel die *Slot*-Klasse `CompilerMerker`, deren einzige Instanz das zu *SGLObjBase* gehörende Funktionsobjekt `compileNextTime` ist.

4.2 Die wichtigsten Objektklassen

Die Basisbibliothek wird, wie im Entwurf festgelegt, objektorientiert implementiert. Die dabei verwendete Hierarchie und Strukturierung lehnt sich an die eines "natürlichen" Raumes an.

4.2.1 Die Raumklasse SGLSpace

SGLSpace, die Objektklasse zur Abstraktion des Raumes, stellt die zentrale Schnittstelle und Verwalungsinstanz des Systems dar. Sie erzeugt einen *OpenGL*-Kontext, und initialisiert wenn nötig den *OpenGL-Renderer*. Sie ist ebenso für die Kommunikation mit den Widgetsystem zuständig. Zur Anpassung an verschiedene Widgetsysteme werden spezialisierte Raumklassen von der allgemeinen Raumklasse abgeleitet. Diese Ableitungen sind Teil gesonderter Widget-Adapter-Bibliotheken.

Sämtliche zu zeichnende Objekte müssen sich bei der entsprechenden Instanz der Raum-Klasse registrieren, um in diesem "Raum" gezeichnet zu werden. Die Raumklasse speichert intelligente Zeiger auf diese Objekte in speziellen Listen. Neben den "normalen" Objekten verwaltet jede Rauminstanz noch einige spezielle Objekte die es nur einmal pro Raum geben kann. Zum Beispiel eine Konsole für die Ausgabe von Meldungen im Grafikfenster, Koordinatengitter mit ihren Achsenbeschriftungen und natürlich eine Kamera. Auch die Modi des Renderers werden zentral in der Raumklasse verwaltet.

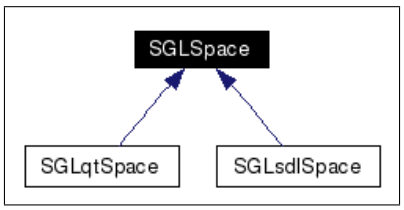

Abb. 4.2: Klassendiagramm für SGLSpace

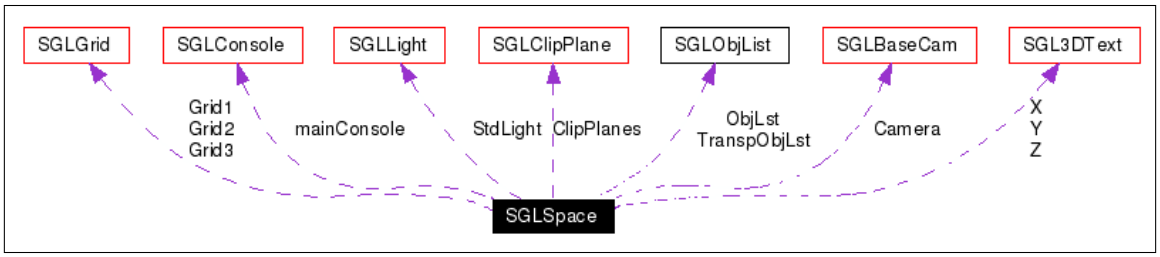

Abb. 4.3: wichtige Member von SGLSpace

Jegliche interne und externe Ereignisse werden von ihr interpretiert und die entsprechenden Operationen, wie zum Beispiel die Umpositionierung der Kamera, ausgeführt. Bei Bedarf zeichnet sie die Szene danach komplett neu. Dazu wird der Renderer in den entsprechenden Modus versetzt und anschließend die Objektlisten aufgefordert, bei allen registrierten Objekten die Funktion zum Zeichnen aufzurufen.

Jede Instanz der Raumklasse hat einen eigenen *OpenGL*-Kontext und ein eigenes Fenster, in das dieser Kontex *rendert*. Die einzelnen Kontexte verschiedener Instanzen der Raumklasse können sich aber trotzdem Daten im *Renderer* teilen. Objekte, die in einer Instanz erzeugt wurden, stehen so automatischen allen anderen *OpenGL*-Kontexten zum Darstellen zur Verfügung. Unabhängig davon muss ein Objekt in jeder Instanz der Raumklasse registriert sein, in der es gezeichnet werden soll. Ist ein Objekt in mehreren Instanzen der Raumklasse registriert, können diese sich seine Daten teilen. Auf diese Weise wird ein und das selbe Objekt mit den entsprechenden Transformationen der verschiedenen Instanzen in den verschiedemem Kontexten gezeichnet. In solch einem Fall kann jede Instanz der Raumklasse als eigenständige Sicht auf ein und denselben Raum betrachtet werden. Wie bereits erwähnt muss das Teilen der Container der Zeichenobjekte durch die Anwendung selbst realisiert werden, da die Kontexte sich nur *Renderer*-interne Daten teilen können.

4.2.2 Die Basisklasse für Zeichenobjekte `SGLObj`

`SGLObj` ist die Basisklasse aller Container für Zeichenobjekte. Sie hält sämtliche allgemeine Parameter des Objektes wie Farbinformationen und die Transformationsmatrizen zur Positionierung des Objektes im Raum. Sie stellt auch die Schnittstelle zum Zeichnen dieses Objektes dar, und verwaltet seinen Operationspuffer.

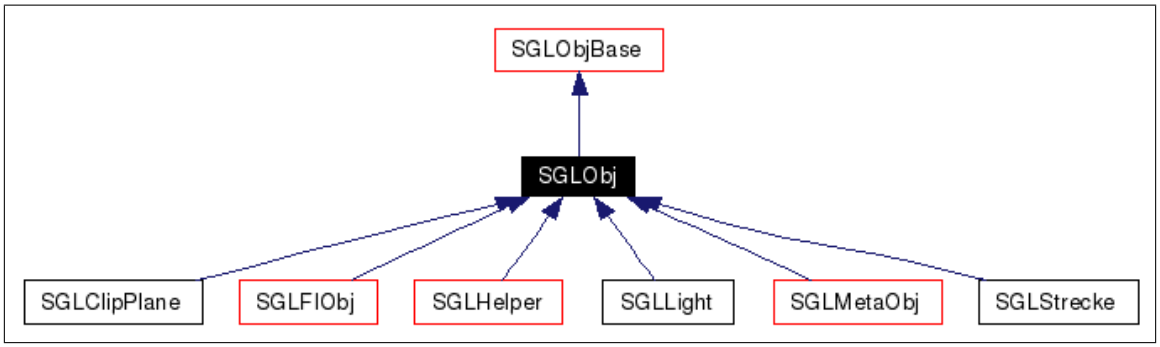

Abb. 4.4: Klassendiagramm für SGLObj

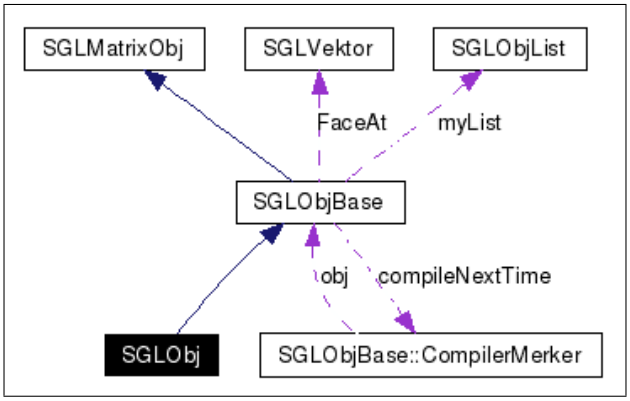

Abb. 4.5: SGLObj und ihre Beziehungen zu Anderen Klassen

Puffern von Zeichenoperationen in OpenGL

OpenGL ist in der Lage, sich durch API-Aufrufe ausgelöste interne Operationen des *Renderers* zu merken und im *Renderer* zu puffern. Muss das Objekt neu gezeichnet werden, entscheidet die Zeichenschnittstelle von `SGLObj`, ob es nötig ist, das Objekt mittels API-Aufrufen komplett neu zu generieren. Dies wird im Folgenden als Kompilierung des

Objektes bezeichnet. Hat sich an einem Objekt seit der letzten Kompilierung Nichts geändert, würde sich auch an den API-Aufrufen zum Zeichnen Nichts im Vergleich zum vorherigen Aufruf dieser Zeichenfunktionen ändern. In solch einem Fall reicht es aus, nur den entsprechenden Operationspuffer aufzurufen. Der Effekt dieses Vorgehens entspricht dem eines Caches für Zeichenoperationen und vereinfacht nebenbei das gleichzeitige Anzeigen eines Objektes in mehreren Sichten, denn die entsprechenden Operationspuffer liegen im *Renderer*. Sie stehen somit allen Kontexten gleichermaßen zur Verfügung.

Tritt ein Ereignis auf, welches das Objekt möglicherweise verändert, muss der Operationspuffer dieses Objektes als ungültig markiert werden. Gleichzeitig werden alle Sichten, in denen es angezeigt wird, aufgefordert neu zu zeichnen. Ruft eine solche Sicht (Instanz der Raumklasse) über ihre Objektliste die Zeichenfunktion dieses Objekts auf, kompiliert diese das Objekt neu, da sie erkennt, dass der Operationspuffer nicht mehr aktuell ist. Danach markiert die Zeichenfunktion den Operationspuffer des Objektes wieder als gültig und zeichnet das Objekt. Wenn möglich, werden das Zeichnen und das Kompilieren zusammengefasst, im Ergebnis unterscheidet es sich aber nicht von der Hintereinanderausführung.

Operationspuffer beim Zeichnen in mehreren Sichten

Wie beschrieben, teilen sich alle Sichten sowohl interne Daten des *Renderers*, in diesem Fall den Operationspuffer des Objektes, als auch die dazugehörigen anwendungsseitigen Informationen. Sie erkennen daher bei einem kürzlich kompilierten Objekt, dass der Operationspuffer gültig ist. Sie können das Objekt also direkt zeichnen, ohne es erneut kompilieren zu müssen. In einer *Multithreading*-Umgebung kann es theoretisch vorkommen, dass mehrere Instanzen gleichzeitig die Ungültigkeit der Operationspuffer feststellen. Das hat zur Folge, dass das Objekt unter Umständen unnötigerweise mehrfach kompiliert wird. Dies kostet zwar etwas Zeit, hat aber keine weiteren Folgen, da der *Renderer* dafür sorgt, dass der Operationspuffer auch unter diesen Umständen korrekt geschrieben wird. Es lohnt es sich daher nicht, ein System zu implementieren, das parallele Kompilierungen verhindert. Ein solches System würde selbst zusätzlich Rechenzeit beanspruchen, und wäre nicht zuletzt eine weitere Fehlerquelle.

Die Funktion, die den Operationspuffer eines Objektes als ungültig markiert, ist als *Slot* implementiert (`CompilerMerker compileNextTime` zu sehen auf Abb. 4.4 auf der vorherigen Seite). Dadurch ist es möglich, den Operationspuffer eines Objektes mittels eines *Signals*, z. B. von einem anderen Objekt aus, als ungültig zu markieren.

Gleichzeitig ist es auch möglich den *Slot* konventionell wie eine Funktion aufzurufen ("`compileNextTime()`"), da der *Slot* ein Funktionsobjekt ist.

SGLObj als abstrakte Klasse

Welche API-Funktionen beim Zeichnen bzw. Kompilieren des Objektes in welcher Form aufgerufen werden, hängt davon ab, was gezeichnet werden soll. Die eigentliche Zeichenfunktion, die beim Kompilieren des Operationspuffers aufgerufen wird, ist deshalb rein virtuell (`SGLObj::generate()`). Sie wird erst durch die eigentliche Objektklasse implementiert. So haben die verschieden Objekte eine gemeinsame Verwaltung und trotzdem volle Kontrolle über ihre Zeichenoperationen. Neue Objektklassen können somit auch außerhalb der Basisbibliothek implementiert bzw. spezialisiert werden.

Positionierung mittels Transformationsmatrix

Neben der Schnittstelle zum Zeichnen des Objektes und zur Verwaltung des Operationspuffers bietet `SGLObj` außerdem von `SGLMatrixObj` geerbte Funktionen zur Manipulation der Transformationsmatrix des Objektes. In Kombination mit den Matrizen der Sicht bestimmen sie Lage und Skalierung des Objektes. Es stehen Funktionen zum Rotieren, Verschieben und Skalieren zur Verfügung. Sie nehmen die entsprechenden Änderungen an der Transformationsmatrix vor. Bei Aufruf dieser Funktionen muss auf die Reihenfolge geachtet werden, da sie sich auch gegenseitig beeinflussen. Jegliche Operationen sind als nicht umkehrbar definiert. Der Aufruf der Transformationsmatrix wird mit in den Operationspuffer aufgenommen, er wird daher bei allen Zeichenaufrufen von allen Sichten aus implizit zusammen mit den restlichen Zeichenoperationen aufgerufen. Das Objekt hat dadurch für alle Sichten die selbe Lage im Raum, während jede Sicht ihre eigenen Sichttransformationen darüberlegt. Das entspricht dem entworfenen Konzept von mehreren Sichten die aus unabhängigen Richtungen den selben Raum betrachten. Sie "sehen" das Objekt zwar an der gleichen Stelle im Raum, aber aus verschiedenen Perspektiven.

4.2.3 Basisklasse zur Manipulation der Sicht (`SGLBaseCam`)

Die Instanzen der Raumklasse verwalteten ihre Sichttransformationen nicht selbst, sondern überlassen dies dem Kameraobjekt, das ihnen zugeordnet ist. Dabei bestimmt die Kamera ob und wie sie auf Ereignisse wie zu Beispiel einen Mausklick reagiert.

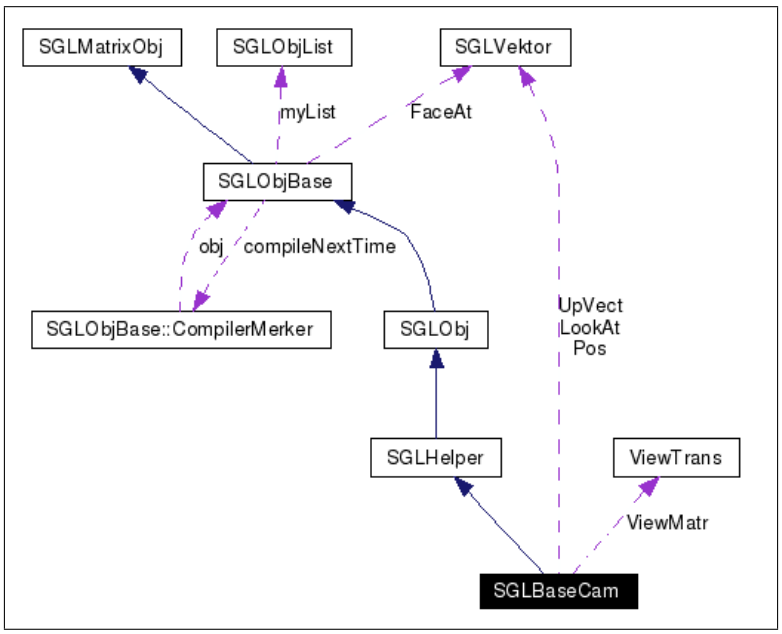

Abb. 4.6: `SGLBaseCam` und ihre Beziehungen zu anderen Klassen

Durch das Austauschen des Kameraobjektes lässt sich somit die Interaktion der Sicht mit dem Anwender beeinflussen. Gleichzeitig erbt `SGLBaseCam` von `SGLObj`, Kameras lassen sich also auch zeichnen. Das wurde z.B. in den Beispielbildern der *GUI* genutzt, um die Lage und die Parameter der Kameras anzudeuten.

Abstraktion der Sichttransformationen

Die Lage der Kamera im Raum wird im Gegensatz zur Lage der anderen Objekte nicht durch ihre Transformationsmatrix bestimmt. Wird eine Kamera bewegt, ändert sie nicht ihre Transformationsmatrix, sondern die Transformationsmatrizen der Sicht und bestimmt so zentral wie der Raum in der Sicht gezeichnet wird, der die Kamera zugeordnet ist. Wird die Kamera virtuell nach rechts bewegt, ändert dies die Transformationsmatrizen derart, dass alles im Raum entsprechend weiter links gezeichnet wird. Das gleiche gilt auch für alle anderen virtuellen Bewegungen der Kamera im Raum. Die meisten dazu nötigen Matrixoperationen werden wieder vom *OpenGL-Renderer* ausgeführt. Die *OpenGL-API* bietet Funktionen, die auf der Basis direkter Angaben zur Lage der Kamera und des Punktes an den sie "blickt" entsprechende Änderungen an den Transformationsmatrizen der Sicht vornimmt. Die Kamera ist somit das einzige Objekt, dessen Lage im Raum nicht durch seine Transformationsmatrix, sondern durch direkte

Koordinaten bestimmt wird.

Das Zoomen kann auf mehrere Arten erfolgen. Die einfachste und robusteste Methode ist, die Kamera einfach näher an das betrachtete Objekt heran zu bewegen. Die Zoomoperation wäre damit auch nur eine Bewegung der Kamera. Die zweite Möglichkeit besteht in einer Verengung des Sichtfeldes der Kamera. Da so weniger dargestellt wird, aber die Größe der Anzeige selbst nicht verändert wird, erscheint das Angezeigte größer. Dies ist vergleichbar mit dem Ändern der Brennweite einer realen Kamera. Die Dritte Möglichkeit besteht darin, den Zoom als zweidimensionale Skalierung direkt auf den Anzeigepuffer anzuwenden. Der Zoomfaktor ist dabei allerdings auf ganze Zahlen beschränkt, und darf nicht kleiner als eins sein. Diese dritte Methode wird in dieser Arbeit nicht zur Anwendung kommen.

Anpassung an Form und Größe des Sichfensters

Eine Änderung von Form und Größe der Anzeige hat tiefgreifende Folgen auf die Darstellung. Der *Renderer* muss den Anzeigepuffer entsprechend anpassen und auch die Kamera selbst muss unter Umständen einige Anpassungen vornehmen. Die Information, über eine Änderung des Anzeigefensters kommt vom Widgetadapter und wird von der Raumklasse an die aktuelle Kamera weitergegeben. Wie diese darauf reagiert, hängt von ihrem Verhaltensprofil ab. In jeden Fall informiert sie den Renderer über die Änderung der Anzeige. Unternimmt sie nichts weiteres, muss z. B. bei einer Verkleinerung der Anzeige das in seiner Größe unveränderte Sichtfeld der Kamera auf eine kleinere Fläche projiziert werden. Das Angezeigte wird dadurch unweigerlich verkleinert. Die Kamera kann diesem Effekt durch Zoomen entgegenwirken. Welche der drei möglichen Methoden des Zoomens sie dabei anwendet hängt wiederum von ihrem Profil ab. Die dritte Methode steht dabei aber nicht zur Wahl, da sich mit ihr nicht stufenlos zoomen lässt. Für die beiden Anderen muss der Abstand der Kamera zum betrachteten Punkt bzw. der Winkel des Sichtfeldes bestimmt werden. Zu diesem Zweck werden die Eckpunkte des Fensters in den Raum projiziert. Dazu wird die in Abschnitt 3.2.3 auf Seite 25 beschriebene Positionsbestimmung angewendet, wobei die aktuelle Entfernung der Kamera zu dem betrachteten Punkt als Tiefeninformation fungiert. Die neue Entfernung bzw. der Winkel des Sichtfeldes der Kamera lässt sich dann aus der Höhe dieses projizierten "Fensters" wie folgt bestimmen.

$$\alpha = \arctan\left(\frac{h}{|\vec{s}|}\right) * 2$$

$$\vec{s'} = \vec{s} * \frac{h}{\tan(\alpha/2)|\vec{s}|}$$

Dabei sind $\vec{s} \in \mathbb{R}^3$ der Vektor von der Kamera zu dem betrachteten Punkt, $h \in \mathbb{R}$ die Höhe des projizierten Fensters und $\alpha \in \mathbb{R}$ der Winkel des Sichtfeldes der Kamera. Es ist zu beachten, dass der betrachtete Punkt in beiden Fällen gleich bleibt. Während im ersten Fall das Sichtfeld erweitert wird, ändert die Kamera im zweiten Fall ihre Position entlang der Sichtlinie. In beiden Fällen wird nur die Höhe, nicht die Breite des Anzeigefensters berücksichtigt.

4.2.4 Basisklassen für Flächenobjekte

Allgemeine geometrische Objekte (`SGLFlObj`)

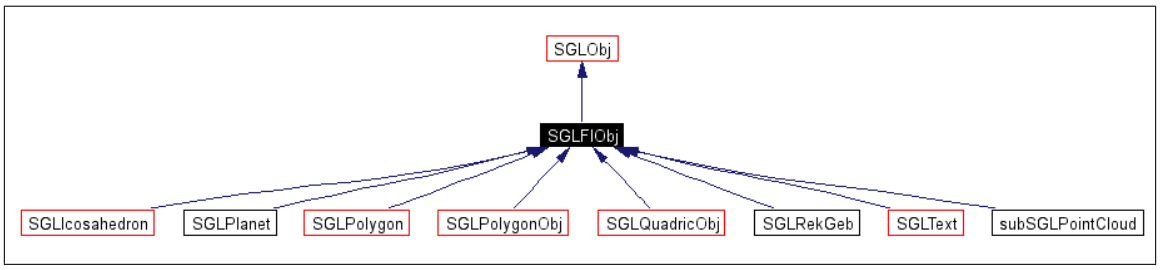

Abb. 4.7: Klassendiagramm für `SGLFlObj`

Ein Großteil der zu zeichnenden Objekte sind geometrische Objekte, die eine Oberfläche haben. Für sie werden beim Zeichnen weitere Informationen darüber benötigt, wie ihre einzelne Grenzflächen zu zeichnen sind. Die Informationen die das Zeichnen der Flächen direkt bestimmen, werden in der eigenständigen Objektklasse `SGLMaterial` gehalten. Allgemeinere Informationen, wie z. B., ob das betreffende Objektes überhaupt mit Oberflächen, oder nur als Drahtgittermodell gezeichnet werden soll, liegen im Objekt selbst.

Polygonobjekte und Polygone (`SGLPolygonObj` / `SGLPolygon`)

Eine Spezialisierung der Flächenobjekte stellen Polygonobjekte dar. Polygonobjekte unterscheiden sich untereinander nur in Lage und Anzahl der Polygone, die ihre Grenzflächen darstellen. Gezeichnet werden Polygonobjekte, indem alle ihre Grenzflächen ge-

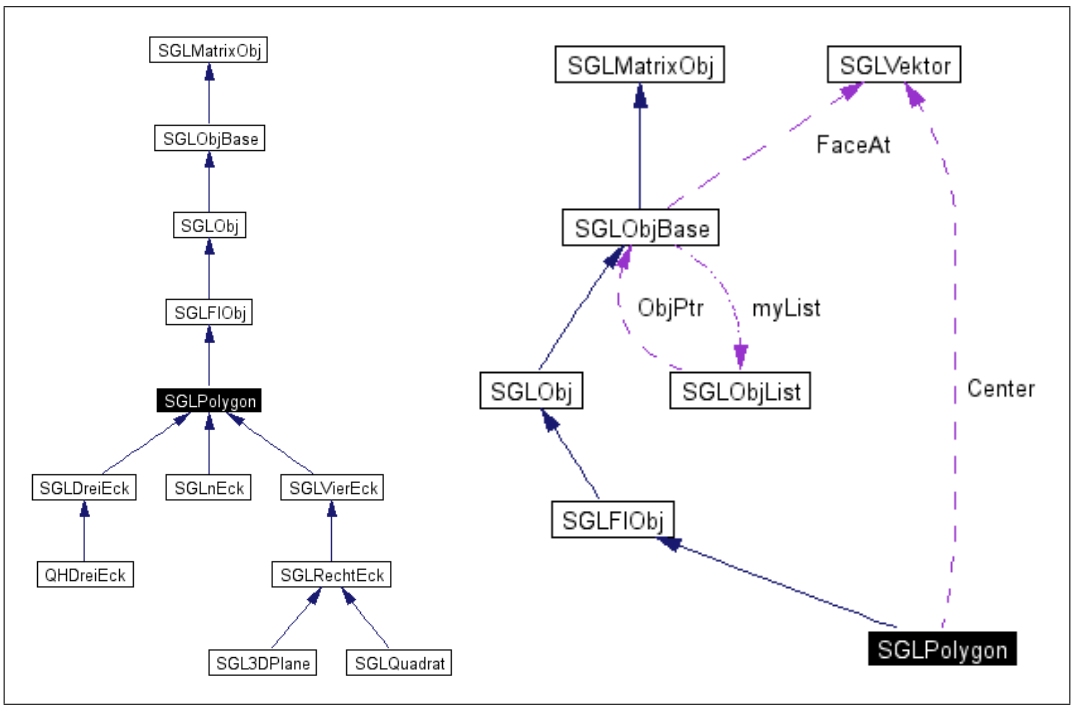

Abb. 4.8: Klassendiagramm und Beziehungen von `SGLPolygon`

zeichnet werden. Diese Grenzflächen sind eingenständige Instanzen der Klasse `SGLPolygon`, die auch einzeln gezeichnet werden können.

Sie erben von `SGLObj` und können daher wie alle anderen Objekte unter Einbeziehung des Operationspuffers gezeichnet werden. Wie schon beschrieben, sind die allgemeinem Zeichnenfunktionen (Pufferverwaltung, Setzen globaler Parameter, Laden der Transformationsmatrix) und das eigentliche Zeichnen (Kompilieren) in gesonderten Funktionen implementiert. Die Polygonobjekte können ihre allgemeinen Zeichenfunktionen inklusive dem Verwalten des Operationspuffers wie andere Objekte auch ausführen. Statt der eigentlichen Zeichnenoperation, d.h der Kompilierung rufen sie aber sequentiell die Kompilierung ihrer Grenzflächen-Polygone auf. Die Lage des Polygonobjektes im Raum wird nur durch die Transformationsmatrix des Polygonobjektes selbst bestimmt. Die Transformationsmatrizen der Polygone werden ignoriert, da der Aufruf der Transformationsmatrix nicht Teil der eigentlichen Zeichenoperation ist. Das ist auch sinnvoll, da sich die Lage der Grenzflächen einfacher und zuverlässiger durch die Bestimmung ihrer Eckpunkte relativ zum Mittelpunkt des Polygonobjektes festlegen lässt. Mit Hilfe intelligenter Zeiger können sich diese Polygone auch programmtechnisch Eckpunkte teilen, die sie geometrisch gemeinsam haben. Verschiebt sich der Eckpunkt eines Polygons,

verschieben sich damit automatisch auch alle Eckpunkte anliegender Polygone. Eine normalerweise Kommunikation zwischen den Polygonen zur "Meldung" der Änderung ist bei Polygonen innerhalb von Polygonobjekten nicht erforderlich, da das Polygonobjekt sie ohnehin nur als Verbund kompiliert. Bei einer Änderung muss daher nur das Übergeordnete Polygonobjekt "informiert" werden. Diese Änderungen werden aber ohnehin in der Regel von diesem Objekt ausgeführt, somit erübrigt sich auch diese Kommunikation.

4.3 Die Widgetadapter

Für die Basisbibliothek ist es unerheblich, welches Widgetsystem für das eigentliche Programm verwendet wird. Zwischen beiden existieren nur zwei Berührungspunkte, das Fenstermanagement und die Eingabebehandlung. Das Fenstermanagement umfasst das Anlegen von Fenstern inklusive eines dazugehörigen *OpenGL*-Kontextes, sowie den Umgang mit dieses Fenster betreffenden Ereignissen. Zum Beispiel eine Änderung der Fenstergröße. Währenddessen ist die Eingabebehandlung für sämtliche Eingabeereignisse, sowohl gedrückte Tasten, als auch Mausbewegungen zuständig. Das Zeichnen an sich ist unabhängig vom verwendeten Widgetsystem, da der *OpenGL-Renderer* direkt in den Anzeigepuffer, den *OpenGL-Kontext* schreibt und so das Widgetsystem umgeht.

Widgetadapter sind gesonderte Bibliotheken, die als Vermittler zwischen dem jeweiligen Widgetsystem und der Basisbiliothek dienen. Sie leiten die vom Widgetsystem (und damit vom der Plattform) kommenden Ereignisse an die entsprechenden Stellen in der Basisbibliothek weiter. Implementiert werden Adapter mittels Ableitungen der entsprechenden Objektklassen aus der Basisbibliothek. Sie sind damit das erste Beispiel einer Spezialisierung aus den allgemeinen Klassen der Basisbibliothek.

Im Rahmen dieser Arbeit werden Adapter für *QT* und *SDL* implementiert. Die *QT*-Bibliothek bietet ein mächtiges Widgetsystem, das für zahlreiche Plattformen verfügbar ist. Es hat in jeder Umgebung die gleiche API und reduziert so den Portierungsaufwand. Im Gegensatz dazu ist *SDL* eigentlich eine auf Spiele spezialisierte Multimediaschnittstelle. Es ist daher kein Widgetsystem in eigentlichen Sinne, sondern eher mit *DirectX* vergleichbar. Wie auch *Qt* ist es, im Gegensatz zu *DirectX*, Plattformunabhängig. *SDL* verfügt über einfache Funktionen zum Anlegen und Verwalten von Fenstern, sowie über eine Eingabebehandlung und erfüllt damit die gestellten Anforderungen.

Bei der Entwicklung der Basisbibliothek dient *SDL* als alternative Testumgebung, da es viel einfacher als *Qt* und damit leichter zu überblicken ist. Das eigentliche Visualisierungstool wird jedoch ausschließlich auf *Qt* ausgelegt sein.

5 Die watershed-Transformation

Voraussetzungen für eine interaktive Objekterkennung in einem Grauwertbild ist die Aufteilung des Bildes in Unterobjekte. Diese Vorsegmentierung gruppiert nach dem *watershed*-Verfahren [6, 9] Punkte, die mit größter Wahrscheinlichkeit dem gleichen Objekt angehören. Das heißt, dass die Wahrscheinlichkeit, dass diese Punkte einem anderen beliebig kleinen Objekt angehören kleiner ist.

Die *watershed*-Transformation wurde von [6] eingeführt, und später von Beucher und Lantuéjoul [2] verbessert. Sie ist die üblichste Methode zur Segmentierung zweidimensionaler Grauwertbilder. Sowohl die theoretischen Überlegungen in diesem Kapitel, als auch die Implementationen im nächsten Kapitel basieren auf der weiterführenden Ausarbeitung von Roerdink und Meijster [18] zu diesem Thema.

Ist die Vorsegmentierung abgeschlossen, kann der Anwender die entstehenden Unterobjekte daraufhin interaktiv zusammenfügen, um daraus die korrekten Objekte zu formen. Der Vorteil besteht darin, dass der Nutzer die Kontrolle über die Objekterkennung behält. Er kann außerdem bei einem Irrtum seiner seinerseits jederzeit ein beliebiges Unterobjekt entfernen, denn die Hauptobjekte sind lediglich Listen der Unterobjekte die sie bilden.

5.1 Mathematische Grundlagen

5.1.1 Grauwertbilder als Graphen

Allgemeine Graphen

Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten ($V \subset \mathbb{D}$), sowie einer Menge von Knotenpaaren ($E \subseteq V \times V$). Dabei ist $\mathbb{D}$ ein n-dimensionaler Datenraum in dem die Punkte V liegen. Die Knotenpaare E werden im Folgenden als Kanten bezeichnet. In gerichteten Graphen besteht E aus geordneten Paaren, während in ungerichteten

Graphen die Reihenfolge der Knoten in den Paaren nicht festgelegt ist. Die Menge der Nachbarn eines Knotens p $N_g(p)$ ergibt sich aus der Menge aller Knoten q, für die es eine Kante (p,q) oder (q,p) gibt. Es gilt:

$$N_g(p) = \{\forall q \in V | \exists k \in E : k = (p,q) \vee k = (q,p)\}$$

Ein Pfad $\pi(p,q)$ der Länge ℓ, in einem Graphen $G = (V,E)$ von einem Knoten p zu einem weiteren Knoten q ist eine Sequenz von Knoten $(p_0, p_1, \ldots, p_{\ell-1}, p_\ell)$, für die $p_0 = p, p_\ell = q$ sowie $\forall i \in [0,\ell) : (p_i, p_{i+1}) \in E$ gilt. Existiert ein Pfad $\pi(p,q)$, wird q als von p erreichbar bezeichnet $(p \rightsquigarrow q)$. Ist dabei $p = q$, ist $\pi(p,q)$ ein Kreis.

Geodätischer Abstand

Der Geodätische Abstand zwischen zwei Knoten $p, q \in V$ ist die Länge $d_V(p,q)$ des kürzesten Pfades zwischen p und q innerhalb der Menge V. Der Geodätische Abstand zwischen einem Knoten p und einer Knotengruppe $P \subset V$ lässt sich daraus ableiten als $d_V(p,P) = min_{q \in \mathbb{P}}(d_V(p,q))$.

Geodätische Einflusszone

$B \subseteq V$ sei unterteilt in k Gruppen zusammenhängender Knoten B_i wobei $(i = 1 \ldots k)$. Dann ist die Geodätische Einflusszone $iz_V \subseteq V$ wie folgt definiert.

$$iz_V(B_i) = \{p \in \mathbb{V} | \forall i \in [1 \ldots k] \backslash \{i\} : d_V(p,B_i) < d_V(p,B_j)\}$$

$iz_V(B_i)$ sind also alle Punkte p für die gilt, dass alle $d_V(p,B_j)$ größer sind als $d_V(p,B_i)$.

Die Menge $IZ_V(B) \subset V$ ist die Vereinigungsmenge aller $iz_V(B_i)$ für $i = 1...k$, d.h.

$$IZ_V(B) = \bigcup_{i=1}^{k} iz_V(B_i)$$

Punkte, die zwar Element von B, aber nicht von $IZ_V(B)$ sind, konnten folglich keiner Einflusszone innerhalb von B eindeutig zugeordnet werden. Dies tritt ein, wenn geodätischer Abstand zu mindestens zwei verschiedenen $B_i \subseteq B$ gleich ist.

Wertegraphen

Die Funktion $f : V \mapsto \mathbb{X}$ des Wertegraphen $G = (V, E, f)$ bestimmt für jeden Knoten $p \in V$ einen Wert $f(p) \in \mathbb{X}$. Dabei ist $\mathbb{X}$ ein meist durch die Anwendung bestimmter Werteraum.

Als Pegelgruppe auf dem Pegel h in einem Wertegraph $G = (V, E, f)$ wird eine Gruppe von, durch Kanten verbundenen, Knoten ($p \in V$) mit dem selben Wert $f(p)$ bezeichnet. Die Grenze einer Pegelgruppe setzt sich aus den Knoten der Pegelgruppe zusammen, deren Nachbarn nicht zur Pegelgruppe gehören. Die abfallende Grenze einer Pegelgruppe sind Grenzknoten deren Nachbarn niedrigere Werte $f(p)$ haben, während Nachbarn von Knoten der Ansteigenden Grenze höhere $f(p)$, als die Knoten der Pegelgruppe aufweisen. Als innere Pegelgruppe werden alle Knoten der Pegelgruppe bezeichnet, die nicht zur Grenze gehören.

Abfallende Pfade sind Pfade für deren Knoten $\forall i \in [0, \ell) : f(p_i) \geq f(p_{i+1})$ gilt, während für ansteigende Pfade $\forall i \in [0, \ell) : (f(p_i) \leq f(p_{i+1}))$ gilt. $\Pi_f^{\downarrow}(p)$ bezeichnet die Menge aller abfallenden Pfade, die in p beginnen. Analog dazu ist $\Pi_f^{\uparrow}(p)$ die Menge aller ansteigenden Pfade, die in p beginnen.

Als lokales Minimum wird eine Pegelgruppe P bezeichnet, die keine abfallende Grenze hat, d.h. $\forall p \in P : \Pi_f^{\downarrow}(p) = \emptyset$.

Ein Wertegraph ist Plateaufrei, wenn jeder Knoten, der nicht Teil eines lokalen Minima ist, einen "niedrigeren" Nachbarn hat. In einem solchen Graphen $G = (V, E, f)$ gilt daher $\forall p \in V : \Pi_f^{\downarrow}(p) \neq \emptyset$.

digitale Grauwertbilder als Wertegraphen

Ein digitales Grauwertbild ist ein Graph $G = (\mathbb{D}, E, f)$, bei dem $\mathbb{D} \subseteq \mathbb{Z}^n$ ein Satz von Knoten ist, welche jeweils zu ihren Nachbarn Kanten haben können. $n \in \mathbb{Z}$ ist dabei Anzahl der Dimensionen des Bildes. Diese Knoten sollen im Folgenden als Punkte bezeichnet werden. Es sei $E \in [\mathbb{Z}^n \times \mathbb{Z}^n]$ die Menge aller Kanten zwischen benachbarten Punkten. Die Funktion $f : \mathbb{D} \mapsto \mathbb{N}$ weist dabei jedem Punkt $p \in \mathbb{D}$ einen Ganzzahlwert $f(p)$ als Grauwert zu.

5.2 Definitionen der watershed-Transformation

5.2.1 Topografischer Abstand in stetigen Bildern

In einem stetigen Bild existieren keine diskreten Punkte. Der "Pfad" von einer Position, zu einer Anderen innerhalb dieses Wertegebirges ist daher eine stetige Kurve. Sind die Punkte p und q Anfang bzw. Ende eines solchen Pfades, dann beschreibt die Funktion $\gamma : \mathbb{R} \mapsto \mathbb{D}$ einen Pfad zwischen beiden wenn $\gamma(0) = p$ und $\gamma(1) = q$ sind. Für die *watershed*-Transformation sind nur stetig absteigende Pfade interessant. Es gilt somit:

$$\forall i \in R : 0 \leq i \leq 1 \rightarrow \int (\gamma(i)) < 0$$

Das Minimum aller Ableitungen der Pfade γ innerhalb des Wertegebirges $\mathbb{D}$, für die $\gamma(0) = p$ und $\gamma(1) = q$ gilt, beschreibt den steilste Absteig von p nach q. Die Topografische Distanz zwischen p und q ist also definiert als

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

5.2.2 watershed-Transformation in stetigen Bildern

Das Bild $G = (\mathbb{D}, E, f)$ habe einen Satz lokaler Minima $\{m_k\}_{k \in I}$ (I sei Menge von Indexen). Der Einzugsbereich $CB(m_i) \subseteq \mathbb{D}$ des lokalen Minimums m_i ist definiert als die Punkte, die topografisch dichter an m_i sind als zu jedem anderen lokalen Minimum m_j (vergl. Abschnitt 5.1.1 auf Seite 46)

$$CB(m_i) = \{x \in \mathbb{D} | \forall j \in I \backslash \{i\} : f(m_i) + T_f(x, m_i) < f(m_j) + T_f(x, m_j)\}$$

Eine Wasserscheide W_{shed} ist die Menge der Punkte, die zu keinem $CB(m_i)$ gehören. Sie besteht aus Punkten, welche die gleiche Entfernung zu mindestens zwei m_i haben. (vergl. Abschnitt 5.1.1 auf Seite 46)

$$W_{shed}(G) = \mathbb{D} \cap \left(\bigcup_{i \in I} CB(m_i) \right)$$

Die *watershed*-Transformation des Bildes G ist eine Abbildung $\lambda : \mathbb{D} \mapsto I \cup \{W\}$, sodass $\lambda(p) = i$, wenn $p \in CB(m_i)$ und $\lambda(p) = W$ mit $p \in W_{shed}(f)$ wobei $W \notin I$ ein frei

bestimmter Index ist. Die *watershed*-Transformation eines Bildes G weist daher allen Punkten dieses Bildes so Indexe zu, dass Punkte eines Einzugsbereiches den gleichen Index bekommen während in zwei verschiedenen Einzugsbereichen nie der gleiche Index vorkommt. Punkte, die nicht eindeutig einem Einzugsbereich zugeordnet werden können, werden gesondert behandelt werden. Diesen "Bergkuppen" wird ein Index $W \notin I$ zugewiesen.

5.2.3 watershed-Transformation in diskreten Bildern

Wird die *watershed*-Transformation für stetige Bilder auf diskrete Bilder übertragen, tritt folgendes Problem auf: In diskreten Bildern können Gebiete mit konstantem $f(p)$, d.h. mit konstantem Grauwert auftreten. Sind diese Pegelgruppen von "höheren" Punkten umgeben, lassen sich ihre Punkte noch einfach dem jeweiligen lokalen Minima zuordnen. Jedoch lassen sich Pegelgruppen, die ausschließlich von "niedrigeren" Punkten umgeben sind, nicht eindeutig zuordnen (die *watershed*-Transformation kennt keine "lokalen Maxima"). Diese Punkte müssen gesondert behandelt werden, was im Abschnitt 5.3.2 auf Seite 52 genauer besprochen wird.

watershed-Transformation durch Absenken

Der klassische Algorithmus zur *watershed*-Transformation durch Absenken nach Vincent und Soille [20] setzt ein digitales Grauwertbild mit $f : \mathbb{D} \mapsto \mathbb{N}$ vorraus. Das Maximum h_{max} und das Minimum h_{min} der Grauwerte $f(p)$ aller Punkte p seien bekannt und es gelte $h_{max} < \infty$ bzw. $h_{min} > -\infty$. Da das Bild aus diskreten Punkten besteht, und da $f : \mathbb{D} \mapsto \mathbb{N}$ eine diskrete Funktion ist, gibt es eine endliche Anzahl Werte für $h = f(p)$. Im Bild kommen endlich viele Grauwerte vor. Es lässt sich somit über diese Werte iterieren. $P_h \subseteq \mathbb{D}$ sei die Menge der Punkte deren Grauwert kleiner oder gleich h ist, d.h. $P_h = \{\forall p \in \mathbb{D} : f(p) \leq h\}$. Jede Iteration betrachtet lediglich die Menge der Punkte aus ihrem P_h. In jeder Iteration werden die betrachteten Punkte die innerhalb der geodätischen Einflusszone der in der vorherigen Iteration entstandenen Einflusszone liegen dieser zugerechnet. Tritt ein neues lokales Minimum auf, wird es als neue Geodätische Einflusszone vermerkt. Lokale Minima sind immer vollständig Teil eines P_h, da alle Punkte eines lokalen Minima den selben Grauwert haben. Beginnt die Iteration bei h_{min} lässt sich zudem ausschließen, dass Punkte aus dem Einzugsgebiet eines noch nicht bekannten Minima betrachtet werden. Für den Algorithmus lässt sich folgende Rekursion definieren.

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

Eine Geodätische Einflusszone in $h+1$ kann eine komplett Neue sein, oder sie ist ein eine Stufe höher liegender Teil einer Geodätische Einflusszone aus X_h. In beiden Fällen fließen ihre Punkte in X_{h+1} ein. MIN_h ist dabei die Vereinigung aller lokalen Minima mit $f(p) = h$. So sammeln sich alle Punkte, die entweder einem lokalen Minima angehören, oder in der Geodätische Einflusszone der einen Pegel tiefer liegenden "Sammlung" liegen.

Die Wasserscheide $W_{sched}(G)$ ist die Menge der Punkte, die am Ende der Iteration ($h = h_{max}$) weder einem bekannten lokalem Minimum angehören, noch einer Geodätischen Einflusszone zugeordnet wurden. Es gilt $W_{shed}(G) = \mathbb{D} \backslash X_{h_{max}}$

5.3 watershed-Transformation durch Wurzelsuche

$G = (\mathbb{D}, E, f)$ sei ein diskretes plateaufreies Grauwertbild. Jeder Punkt p, der nicht Teil eines lokalen Minimums ist, muss daher mindestens einen Nachbarn q haben, so dass $f(p) > f(q)$ gilt. Der steilste Abhang von p ist definiert als

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

$N_G(p)$ ist dabei die Menge der Nachbarn von p in G. Wenn $p = q$ gilt, oder p Teil eines lokales Minima ist, ist $LS(p)$ als 0 definiert.

5.3.1 Topografischer Abstand in diskreten Bildern

Die Menge der Nachbarn q von p, deren "Höhendifferenz" maximal, also gleich $LS(p)$ ist, wird als $\Gamma(p)$ bezeichnet. Analog dazu heißt die Menge der Punkte q, für die $p \in \Gamma(q)$ gilt, $\Gamma^{-1}(p)$.

Die Kosten für den Weg von p zu einem Nachbarn q sind definiert als:

$$cost(p,q) = \begin{cases} LS(p) * d(p,q) & f(p) > f(q) \\ LS(q) * d(p,q) & f(p) < f(q) \\ \frac{1}{2}(LS(p) + LS(q)) * d(p,q) & f(p) = f(q) \end{cases}$$

Darauf aufbauend ist die topografische Länge des Pfades $\pi = (p_0, \ldots, p_\ell)$ zwischen $p_0 = p$ und $p_\ell = q$ definiert als:

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

Der topografische Abstand zwischen p und q ist das Minimum der topografischen Längen aller Pfade zwischen p und q

$$T_f(p,q) = \min_{\pi \in [p \rightsquigarrow q]} T_f^h(p,q)$$

und der topografische Abstand zwischen p und einer Menge $A \subseteq \mathbb{D}$ ist analog zur geodätischen Distanz $T_f(p,A) = \min_{a \in A} T_f(p,a)$

Wir nennen $(p_0, p_1, \ldots, p_n)$ den steilsten Abstieg von $p_0 = p$ nach $p_n = q$ wenn $p_{i+1} \in \Gamma(p_i)$ für alle $i = 0, \ldots, n-1$ ist. Ein Punkt q gehört zum *downstream* von p wenn es einen steilsten Abstieg von p nach q gibt. Ein Punkt q gehört zum *upstream* von p, wenn p zum *downstream* von q gehört.

Definition

Es sei $f(p) > f(q)$. Der Pfad $\pi(p,q)$ ist genau dann der steilste "Abhang", wenn $T_f^\pi(p,q) = f(p) - f(q)$ ist. Ansonsten gilt $T_f^\pi(p,q) > f(p) - f(q)$.

$$\forall \pi \in [p \rightsquigarrow q] : \pi = Abstieg_{\max} \leftrightarrow T_f^\pi(p,q) = f(p) - f(q)$$
$$\forall \pi \in [p \rightsquigarrow q] : \pi \neq Abstieg_{\max} \rightarrow T_f^\pi(p,q) > f(p) - f(q)$$

Mit der Einführung der topografischen Distanz für diskrete Bilder ist die restliche Definition für Einzugsgebiete von lokalen Minima analog zum stetigen Fall.

Es folgt, dass $CB(m_i)$ eine Menge von Punkten im *upstream* des lokalen Minima m_i ist. Die Wasserscheide besteht aus den Punkten p, die im *upstream* von mindestens zwei lokalen Minima liegen. Es gibt also mindestens zwei π_i, für die $T_f^{\pi_i}(p, q_i) = f(p) - f(q_i)$ gilt, wobei q_i lokale Minima sind.

Jeder Punkt p im *upstream* eines Wasserscheide-Punktes q ist durch die obige Definition selbst ein Wasserscheide-Punkt, denn q liegt gezwungenermaßen im *downstream* von p, und mit ihm die lokalen Minima von q.

Breite Wasserscheiden

"Breite" Wasserscheiden treten bei Anwendung der *watershed*-Transformation mittels Wurzelsuche immer dann auf, wenn sich der am steilsten absteigende Pfad von einem Punkt in seiner "Steilheit" nur geringfügig von von anderen absteigenden Pfaden unterscheidet. Der Grund dafür liegt darin, dass aufgrund der endlichen Zahl der Nachbarn eines Punktes nicht die Optimale (steilste) Richtung für den Abstieg gewählt werden kann. Auf Ursachen und Eindämmung dieses Problem wird später noch genauer eingegangen werden. Obwohl dies kein für die Methode der Wurzelsuche spezifisches Problem ist, kommt es hier häufiger als z.B. bei der Absenken-Methode vor.

5.3.2 Das Plateauproblem

Bis jetzt wurden Plateaus explizit ausgeschlossen, denn mit der bisherigen Definition kann die topologische Distanz zwischen zwei Punkten p und q eines Plateaus nicht korrekt bestimmt werden. Es wären $f(p) = f(q)$, was zu $LS(p,q) = 0$, $cost(p,q) = 0$ und schließlich zu $T_f^{\pi}(p,q) = 0$ führt. Daher muss die Kostenfunktion entweder erweitert, oder die Plateaus entfernt werden. Für die Wurzelsuche werden Plateaus explizit vor der eigentlichen *watershed*-Transformation entfernt. Die Methode des Absenkens behandelt Plateaus direkter. Im Abschnitt 6.3.1 auf Seite 65 wird genauer darauf eingegangen werden.

Beseitigung von Plateaus

Es sei $\Pi_f^{\downarrow}$ die Menge aller absteigenden Pfade von p ($p \in \mathbb{D}[\exists q \in \mathbb{D} | \forall \pi(p,q) \in \Pi_f^{\downarrow}(p) : f(q) < f(p)]$) und $len(\pi)$ sei die geodätische Länge des Pfades π. $G = (\mathbb{D}, E, f)$ sei ein diskretes Bild, und die Funktion $d : \mathbb{D} \mapsto \mathbb{N}$ sei wie folgt definiert:

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

Unter der Annahme $L_C = \max_{p \in \mathbb{D}} d(p)$ wird die Wertefunktion f_{LC} des plateaubereinigten Bildes $G_{LC} = (\mathbb{D}, E_{LC}, f_{LC})$ basierend auf der Wertefunktion f des Originalbildes bestimmt durch:

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

Die Funktion d hat bei lokalen Minima den Wert 0, bei allen Anderen entspricht $d(p)$ der Länge des Pfades zu Punkten mit geringerem Grauwert. Die Relation zwischen den Punkten ist definiert durch $x \sqsubset y \leftrightarrow f_{LC}(x) < f_{LC}(y)$. Desweiteren muss sich auch der Graph des Bildes ändern. G_{LC} stellt einen gerichteten Graphen dar, in dem nur Bögen von "höheren" Punkten zu "niedrigeren" Punkten führen. Für $E_{LC} \subseteq \mathbb{D} \times \mathbb{D}$ gilt also

$$(p, q) \in E_{LC} \leftrightarrow q \in \Gamma(p)$$

Auf dem Gebiet eines Plateaus wird eine Kante von p nach q erzeugt, wenn die geodätische Distanz zur Kante des Plateaus für p größer ist, als für q. Wenn also $q \sqsubset p$ gilt. Der Graph G_{LC} ist durch diese Definition implizit azyklisch.

Die Bestimmung des Einzugsbereichs verhält sich in einem solchen Bild wie folgt:

$$CB(m_i) = \{p \in \mathbb{D} | \forall j \in I \backslash \{i\} : f_{LC}(m_i) + T_{f_{LC}}(p, m_i) < f_{LC}(m_j) + T_{f_{LC}}(p, m_j)\}$$

6 Implementierungen der watershed-Transformation

6.1 Allgemeine Datenstrukturen

Die Struktur des Graphen eines Bildes, d.h. die Liste der Punkte (D) und die Liste ihrer Kanten bzw. Bögen untereinander (E) sind relativ unabhängig von den Grauwerten dieses Bildes (f). Die Menge der Punkte hat sich in den bisherigen Ausführungen nie geändert. Es wurden stets die selben Punkte verwendet. Lediglich ihre Beziehungen zueinander haben sich einmal geändert. Aus Kanten zwischen geodätischen Nachbarn wurden gerichtete Bögen zu Punkten mit niedrigerem Funktionswert. Dem stehen mehrere Werte entgegen, die eine Funktion f für einen bestimmten Punkt aus D liefern kann. Es ist daher sinnvoll, Daten und Struktur eines Bildes in der Implementation zu trennen.

6.1.1 Der Container für Bilddaten `Bild`

Die Bilddaten sind die, jedem Punkt aus der Menge D zugeordneten Grauwertwerte in den verwendeten digitalen Bildern. Ist F das Grauwertgebirge eines Bildes, so bildet die Funktion f aus der Menge der Punkte in die Menge der Grauwerte dieses Bildes ab ($f : D \mapsto F$). Die Klasse `Bild` bietet einen Container für die Bilddaten und gleichzeitig eine Funktion, die für einen übergeben Punkt $p \in D$ aus diesen Daten das entsprechende von $f(p)$ liefert. Im Nachfolgenden werden für verschiedene Funktionen f zur Abbildung in verschiedene Grauwertgebirge verschiedene Instanzen dieser Klasse verwendet werden. Diese Instanzen werden vereinfachend als "Bild X" bezeichnet werden, wobei "X" der Bezeichnung der Funktion f entspricht (z.B. "Bild f" oder "Bild f_{LC}").

```
1 template <class T> class Bild
2 {
3     T *data;
4     unsigned int xsize,ysize;
5     public:
```

```
6       inline unsigned int size(){return xsize*ysize;}
7       Bild(unsigned int x,unsigned int y,T initVal):xsize(x),ysize(y)
8       {
9           if(initVal==0)data=(T*)calloc(size(),sizeof(T));
10          else
11          {
12              data=(T*)malloc(size()*sizeof(T));
13              for(int i=size()-1;i>=0;i--)data[i]=initVal;
14          }
15      }
16      ~Bild(){free(data);}
17      inline T &operator[](Punkt &p)
18      {
19          assert((p.posx+p.posy*xsize)<size());
20          return data[p.posx+p.posy*xsize];
21      }
22      inline T &operator[](PunktRef &p)
23      {
24          assert((p.posx+p.posy*xsize)<size());
25          return data[p->posx+p->posy*xsize];
26      }
27  };
```

Der Codeausschnitt zeigt, dass `Bild` eine parametrisierte Klasse ist, deren Instanzen an den Wertebereich angepasst werden können, in den die verwendete Funktion abbilden soll. Der Konstruktor legt mit den Informationen über die Dimensionen des Bildes einen entsprechend großen Speicherbereich an. Dabei greift er je nach Voraussetzungen auf die günstigste Systemfunktion zurück. Der überladene `[]`-Operator stellt die Funktionalität von $f(p)$ zur Verfügung. Das heißt er bildet die übergebenen Punkte in den Wertebereich des Bildes ab. Dabei gibt er Referenzen zurück Es ist somit auch möglich Werte zu schreiben ($f(p) \leftarrow x$).

6.1.2 Der Container für Graphendaten `Punkt`

Die Struktur des Graphen eines Bildes wird indirekt durch die einzelnen Knoten des Graphen gespeichert. Jeder dieser Knoten stellt einen Punkt des Bildes dar. Neben Informationen über seine Position innerhalb des Bildes hält jede Punkt eine Liste mit Referenzen auf Punkte, zu denen er Bögen bzw. Kanten hat.

```
1   struct Punkt
2   {
3       unsigned short posx,posy;
4       Punkt():posx(numeric_limits<unsigned short>::max()),posy(numeric_limits<unsigned short>::max()){}
5       deque<PunktRef> neighboursA; /*senkreche/waagerechte Nachbarn im ürsprnglichen Graphen*/
6       deque<PunktRef> neighboursB; /*diagonale Nachbarn im ürsprnglichen Graphen*/
7       multimap<double,PunktRef>   sortetNeighbours; /*Nachbarn(aus neighbours*) mit niedrigerem Wert*/
8       template <class FT> bool hasLowerNeighbour(Bild<FT> &value_list){
9           foreach(PunktRef,neighboursA,p)if(value_list[*p]<value_list[*this])
10              return true;
11          foreach(PunktRef,neighboursB,p)if(value_list[*p]<value_list[*this])
12              return true;
13          return false;
14      }
15
16  };
```

Die Punktlisten `neighboursA` und `neighboursB` halten Referenzen auf senkrechte und waagerechte bzw. auf diagonale Nachbarn der Punkte. Diese zwei Listen bilden somit die Kantenmenge E des Graphen des ursprünglichen Bildes. Die Bögen aus E_{LC} werden durch die Liste `sortetNeighbours` gehalten. Diese Liste speichert zusätzlich zu der Referenz des Punktes, zu dem der jeweilige Bogen geht, den Wert von $T_f^{\pi}(p,q)$. Sie sortiert die Einträge nach diesem Wert, so dass die steilsten Bögen am Ende der Liste stehen. Ist die Liste vollständig, ergibt `sortetNeighbours.rbegin()->first` dadurch immer den maximalen Abstieg und alle Einträge ab `sortetNeighbours.lower_bound(LS)` haben mindestens den Abstieg LS.

6.2 Watershed-Transformation mittels Wurzelsuche

Die auf der Wurzelsuche basierende Implementation der *watershed*-Transformation ist rekursiv und jede Verzweigung der Rekursion läuft isoliert ab. Sie is somit unabhängig von allen anderen Verzweigungen des selben Rekursionsschrittes. Die Wurzelsuche lässt sich daher bei Bedarf leicht parallelisieren.

Dieser von Meijster und Roerdink [13] publizierte Algorithmus zur Umsetzung der *watershed*-Transformation setzt ein plateaubereinigtes Bild mit dem Graphen G_{LC} vorraus. Jeder Punkt p des Graphen G_{LC} hat dabei einen Bogen zu seinem Nachbarn $q \in N_G(p)$ ($(p,q) \in E_{LC}$), wenn $f_{LC}(q) < f_{LC}(p)$ gilt. Diese Bögen können also nur "von oben nach unten" gehen, was Kreise ausschließt. Für die Wurzelsuche muss das Bild plateaufrei sein. Der erste Schritt besteht somit darin, das ursprüngliche Bild in ein plateaufreies Bild zu transformieren.

6.2.1 Von $G = (D, E, f)$ zum plateaufreien $G'_{LC} = (D, E, f_{LC})$

Den Überlegungen in Abschnitt 5.2.3 auf Seite 49 folgend erzeugt die folgende Funktion aus den Strukturinformationen (D und E) und den Grauwerten (f) des originalen Bildes einen neuen Satz Grauwerte f_{LC}, indem sie für alle Punkte $p \in D$ den Wert von $f_{LC}(p)$ bestimmt.

```
1 void t_dist::lowerCompletion
2 (deque<pixel> &D, bild<pixelval> &f, bild<unsigned short> &f_LC, bild<short> &idx)
3 {
4     ptr_fifo fifo;
5     unsigned short dist=1;
6     foreach(pixel,D,p)
7     {
8         if(p->hasLowerNeighbour(f))
9         {
```

```
10              fifo.push(*p);
11              f_LC[p]=numeric_limits<unsigned short>::max();
12          }
13          else f_LC[p]=0;
14      }
15
16      fifo.push_null();
17      while(!fifo.empty())
18      {
19          pixel *p=fifo.pop();
20          if(p==NULL)
21          {
22              if(!fifo.empty())
23              {
24                  fifo.push_null();
25                  assert(dist< numeric_limits<unsigned short>::max());
26                  dist++;
27              }
28          }
29          else
30          {
31              f_LC[p]=dist;
32              foreach(pixel*,p->neighbours,q)
33              {
34                  if(f[p]==f[*q] && f_LC[*q]==0)
35                  {
36                      fifo.push(q);
37                      f_LC[*q]=numeric_limits<unsigned short>::max();
38                  }
39              }
40          }
41      }
42      foreach(pixel,D,p)
43      {
44          assert(f_LC[p]< numeric_limits<unsigned short>::max());
45          if(f_LC[p]!=0)
46              f_LC[p]+=dist*f[p]-1;
47          else
48          {
49              f_LC[p]=dist * f[p];
50              idx[p]=numeric_limits<unsigned short>::max();
51          }
52      }
53  }
```

Die Variable `dist` wird mit 1 initialisiert. Die erste Schleife [Zeilen 6-14] trägt für alle Punkte p für die $\exists q \in N_{G_E}(p) : f(q) < f(p)$ gilt in f_{LC} einen ungültigen Wert ein, und hängt diese Punkte hinten an eine FIFO-Liste an. Alle anderen Punkte sind entweder lokale Minima oder Teil eines Plateaus. Für sie wird in $f_{LC}(p) \leftarrow 0$ gesetzt. Nach Abschluss der Schleife wird abschließend eine Trennmarkierung an die Liste angefügt.

Die zweite Schleife [Zeilen 17-41] prüft, ob einer der Punkte aus der FIFO-Liste gleichwertige Nachbarn hat. In der Liste können zwar ausschließlich Punkte vorkommen, die mindestens einen niederwertigeren Nachbarn haben. Dennoch kann ein weiterer Nachbar eines dieser Punkte gleichwertig sein. Ein solcher Punkt, der sowohl gleichwertige als auch niederwertigere Nachbarn besitzt, ist der Rand eines Plateaus. All seine gleichwertigen Nachbarn, die selbst keinen niederwertigeren Nachbarn haben, werden am Ende der FIFO-Liste angefügt (hinter der Markierung). Es ist zu beachten, dass aufgrund der ersten Schleife an dem, für einen Punkt p in f_{LC} eingetragenen Wert leicht zu erkennen ist, ob dieser Punkt niederwertigere Nachbarn hat. Dies muss nicht wiederholt geprüft

werden. Nachdem sich die gefundenen Plateaupunkte in der Liste befinden, wird für sie in dem Bild f_{LC} ein ungültiger Wert eingetragen. Das verhindert, dass sie ein zweites mal von einem anderen Randpunkt des Plateaus aus an die Liste angehängt werden können. Die Schleife setzt zudem für jeden anderen Punkt der Liste den Wert in f_{LC} auf die Entfernung, die dieser Punkt zum nächsten niederwertigeren Punkt hat. Die Entfernung beträgt anfangs 1, da sich nur Punkte mit niederwertigeren Nachbarn in der Liste befinden. Stößt die Schleife jedoch zum ersten Mal auf die Trennmarkierung, müssen alle folgenden Punkte Plateaupunkte sein. Sie sind daher mindestens zweit Schritte von einem niederwertigeren Punkt entfernt. Diese Punkte müssen einen Schritt über den Nachbarn vornehmen, der sie in die Liste eingetragen hat, um zu einem niederwertigeren Punkt zu gelangen. Die Nachbarn dieser Punkte können nur noch gleichwertig sein, da der Rand des Plateaus bereits bearbeitet wurde. Die Prüfung auf Gleichwertigkeit ist aus diesem Grund eigentlich nicht mehr notwendig. Alle Nachbarn, die noch nicht untersucht wurden, werden wiederum hinter einer neuen Trennmarkierung in die Liste aufgenommen. Ihre Entfernung zu einem niederwertigeren Punkt ist abermals um einen Schritt größer geworden. Auf diese Weise "wachsen" alle Plateaus von außen zu. Nach dem Ende der Schleife entspricht der für jeden Punkt p in f_{LC} eingetragene Wert $d(p)$.

Die letzte Schleife [Zeilen 42-52] ermittelt nur noch für jeden Punkt p $f_{LC}(p)$ aus seinem nun bekannten $d(p)$ und trägt diesen Wert in f_{LC} ein. An dieser Stelle lassen sich auch leicht lokale Minima identifizieren. Alle Punkte p, für die noch immer $f_{LC}(p) = 0$ gilt können nur zu einem Minimum gehören. Diese nebenbei erhaltene Information wird in dem Bild *idx* abgelegt, da sie noch beim Aufbau des Graphen gebraucht wird. Der entstandene Graph $G'_{LC} = (D, E, f_{LC})$ ist plateaufrei. Es gibt neben den lokalen Minima keinen Punkt ohne nach f_{LC} niederwertigeren Nachbarn mehr. Für alle Punkte $p \in D \cap \bigcup_{i \in I} m_i$ (I sei die Menge der Minima-Indexe) gilt $\Pi^{\downarrow}_{f_{LC}(p)} \neq \emptyset$. Das Bild ist plateaufrei.

6.2.2 Von $G'_{LC} = (D, E, f_{LC})$ nach $G_{LC} = (D, E_{LC}, f_{LC})$

Die Bögen für E_{LC} können nun relativ einfach erzeugt werden. Der Abstieg von einem Punkt p zu einem anderen Punkt q wird durch $\frac{f_{LC}(p) - f_{LC}(q)}{d(p,q)}$ bestimmt. Es handelt sich um ein diskretes Bild in dem der geodätische Abstand zweier Nachbarn immer 1 ist. Daher gilt somit $\forall p \in D[\forall q \in N_{G_{LC}} : d(p, q) = 1]$, denn p und q sind Nachbarn. Es wird also ein Bogen von einen Punkt p auf seinen Nachbarn q in E eingetragen, wenn $f_{LC}(p) - f_{LC} = LS(p)$ maximal ist. Werden diagonale Nachbarn mit einbezogen, muss für diese $d(p, q) = \sqrt{2}$ bzw. $d(p, q) = \sqrt{3}$ bei Volumenbildern angenommen werden.

```
1  void buildGraph(deque<Punkt> &D, Bild<unsigned int> &lc)
2  {
3      foreach(Punkt,D,p)
4      {
5          foreach(PunktRef,p->neighboursA,q)if(lc[p]>lc[*q])//senkrechte und waagerechte Nachbarn
6              //lc[p]>lc[*q] => lc[p]-lc[*q] kann nicht negativ sein - keine Pruefung noetig
7              p->sortetNeighbours.insert(pair<double,PunktRef>(lc[p]-lc[*q],*q));
8
9          foreach(PunktRef,p->neighboursB,q)if(lc[p]>lc[*q])//diagonale Nachbarn
10             //lc[p]>lc[*q] => lc[p]-lc[*q] kann nicht negativ sein - keine Pruefung noetig
11             p->sortetNeighbours.insert(pair<double,PunktRef>((lc[p]-lc[*q])/sqrt(2.),*q));
12     }
13 }
```

Da $LS(p)$ unbekannt ist, weicht diese Implementierung leicht von der obigen Definition ab. Um $LS(p)$ zu bestimmen müssten alle Nachbarn, die mindestens um 1 niederwertiger sind untersucht werden um das Maximum, also $LS(p)$ zu finden. Stattdessen nimmt die Schleife alle Nachbarn q mit $f_{LC}(p) - f_{LC} \geq 1$ zusammen mit diesem Wert $(f_{LC}(p) - f_{LC})$ in eine Liste auf. Die Liste wird dabei nach diesem Wert sortiert, daher sammeln sich die Punkte, für die $f_{LC}(p) - f_{LC}$ maximal, und damit gleich $LS(p)$ ist automatisch am Ende der Liste.

Eine Eintragung in dieser Liste wird als Bogen von dem Besitzer dieser Liste zu dem eingetragenen Punkt betrachtet. Nach Abschluss dieser Funktion gibt es für jeden Punkt, der nicht zu einem lokalen Minimum gehört, einen Pfad zu einem solchen. Da ein Punkt eine konstante maximale Anzahl von Nachbarn hat, bleibt maximale Länge der sortierten Liste konstant. Der Aufwand für das Einfügen in diese Liste ist daher ebenfalls konstant bzw. unabhängig von der Größe des Bildes. Gleichzeitig hat jeder Punkt mindestens ein Bogen, der zu einen niederwertigeren Punkt führt, so dass in jeder dieser Listen mindestens ein Eintrag steht.

Die einzigen Punkte, für die das nicht zutrifft, sind Punkte, die zu einem lokalen Minimum gehören. Für diese Punkte p gilt immernoch $\Pi^{\downarrow}_{f_{LC}(p)} = \emptyset$. Sie können somit keinen Bogen zu einem niederwertigeren Punkt besitzen. Um dem Algorithmus garantieren zu können, dass von jedem Punkt ein Bogen zu einen Punkt führt, müssen diese Punkte ein Bogen zu sich selbst besitzen. In ihrer Liste für niederwertigere Punkte müssen sie folglich selbst als einziger Eintrag mit dem Abstand 0 eingetragen werden. Diese Punkte können auf diese Weise als Wurzel erkannt werden. Jedoch muss davon ausgegangen werden, dass es für jedes lokale Minimum m mehr als einen Punkt $p \in m$ gibt. Diese Punkte p mit $\Pi^{\downarrow}_{f}(p) = \emptyset$ gelten als Wurzel des Graphen G_{LC} (Der Graph kann mehrere Wurzeln haben). Jeder Wurzel in G_{LC} wird in der nächsten Funktion ein eindeutiger Index zugewiesen werden. Jeder dieser eindeutigen Indexe steht wiederum für ein lokales Minimum. Für alle anderen Punkte wird in der eigentlichen Wurzelsuche über die bereits erstellten Bögen die am nächsten liegende Wurzel gesucht, und ihnen der Index dieser

Wurzel zugewiesen. Besteht ein lokales Minimum aus mehreren Punkten, ist es möglich, dass Punkten, die im Einzugsbereich dieses Minimums liegen, verschiedene Wurzeln zugewiesen werden. Sie werden also fälschlicherweise als zu verschiedene Minima gehörend angesehen. Um das zu verhindern, darf ein Minimum nur aus einer Wurzel bestehen. Zu diesem Zweck wird für jedes Minimum m ein beliebiger Punkt $p_m \in m$ ausgewählt, der stellvertretend als Wurzel für das Minimum steht. Allen Punkte aus m (auch p_m) wird daraufhin statt dem Bogen auf sich selbst einen Bogen auf p_m eingetragen. Dabei bleibt die topographische Länge dieser Bögen zu p_m 0, da alle Punkte aus m gleichwertig sind. Folglich hat der ausgewählte Punkt p_m als einziger Punkt des entsprechenden lokalen Minimums einen Bogen auf sich selbst und ist somit die einzige Wurzel des Minimums m.

Die Punkte eines lokalen Minimums haben per Definition den gleichen Wert wie die Wurzel, die dieses Minimum repräsentiert. Es wird um sie herum immer Punkte geben, die höher liegen, sonst wären sie nicht Teil eines Minimums. Folglich können solche Punkte höchstens einem Minimum angehören, sie sind nur von einem lokalen Minimum aus erreichbar, ohne dass der Pfad über höhere Punkte verlaufen muss. Ist ein lokales Minimum m bekannt und durch einen beliebigen Punkt $p_m \in m$ repräsentiert, lassen sich alle gleichwertigen Punkte in Nachbarschaft von p_m diesem Minimum durch einfache Wertevergleiche zuordnen. Diese Operation wird rekursiv für jeden gleichwertigen Nachbarn von p wiederholt, indem dieser wiederum gleichwertige Nachbarn sucht, die noch nicht eingeordnet wurden. Diese Rekursion kann auch genutzt werden, um jeweils jedem Minimum bzw. seinen Punkten einen eindeutigen Index zuzuweisen. Dies ist möglich, da jeder obersten Verzweigung der Rekursion ein lokales Minimum entspricht. Alle Punkte, die zu diesem Minimum gehören, werden in dem entsprechenden Zweig der Rekursion behandelt. Auf diesem Wege kann ihnen der Index des Minimums zugewiesen werden. Der folgende Quelltextausschnitt zeigt die zwei Funktionen, die diese Aufgaben erfüllen.

```
1  bool setMinLabel(Punkt &p,PunktRef root,int label,Bild<int> &lab)
2  {
3      if(lab[p]==-1)
4      {
5          lab[p]=label;
6          p.sortetNeighbours.insert(pair<double,PunktRef>(0,root));
7          foreach(PunktRef,p.neighboursA,q)//Direkte Nachbarn
8              setMinLabel(**q,root,label,lab);
9          foreach(PunktRef,p.neighboursB,q)//Diagonale Nachbarn
10             setMinLabel(**q,root,label,lab);
11         return true;
12     }
13     else return false;
14 }
15
16 void labelMinima(deque<Punkt> &D,Bild<int> &lab)
17 {
18     int label=1;
19     foreach(Punkt,D,p)
20         if(setMinLabel(*p,p,label,lab))label++;
21 }
```

Die äußere Funktion `labelMinima` ruft für jeden Punkt die Rekursion `setMinLabel` auf. Wurde eine Rekursion erfolgreich abgeschlossen, geht die äußere Funktion davon aus, dass eine Menge m vollständig ist, und erhöht den Index. Die Rekursion nutzt die bei der Plateaubeseitigung gesammelte Information über lokale Minima, um zu erkennen, ob für den behandelten Punkt p $p \in m$ gilt. Ein direkter Wertevergleich ist nicht notwendig. Es reicht zu prüfen, ob der behandelte Punkt teil eines Minimums ist. Ist dies der Fall, bekommt er einen Bogen zu dem übergebenen Ursprungspunkt. Der Vermerk, dass er einem Minimum angehört, wird durch den Index des Minimums überschrieben, dem er angehört. Dieser Punkt wird ab diesem Zeitpunkt also nicht mehr als Teil eines Minimums angesehen und dadurch auch nicht wiederholt betrachtet. Danach wird die Rekursion für jeden seiner Nachbarn ($N_G(p)$) mit dem gleichen Index und dem gleichen Ursprungspunkt ausgeführt. Ist der behandelte Punkt jedoch kein Teil eines Minimums bricht dieser Zweig der Rekursion erfolglos ab.

Wie bereits in diesem Abschnitt erklärt, kann kein Punkt mehreren Minima angehören. Es wird also kein Punkt mehrfach betrachtet, zumal dieser nach der ersten Betrachtung ohnehin nicht mehr als Teil eines Minimums erkannt wird. Da bereits bekannt ist, welche Punkte einem Minimum angehören, müssen sie nicht mehr durch eine Suche ermittelt werden. Der Aufwand dieser Operation übersteigt somit nie $O(n)$ wobei n die Anzahl der Punkte im Bild ist. In der Praxis liegt der Aufwand meist weit unter O(n), denn zusammenhängende Gruppen gleichwertiger Punkte in einem lokalen Minimum stellen meist nur einen kleinen Teil aller Punkte des Bildes dar.

Damit ist der Graph $G_{LC} = (D, E_{LC}, f_{LC})$ komplett, und es kann mit der eigentlichen Wurzelsuche begonnen werden.

6.2.3 Die Wurzelsuche in G_{LC}

Die *watershed*-Transformtion des ursprünglichen Bildes findet über eine Wurzelsuche in G_{LC} statt. Sie weist jedem Punkt, der noch keinen Index hat, den Index der Wurzel zu, zu der der Pfad am steilsten ist. Dadurch wird eine Zuordnung aller Punkte zu einem Einzugsbereich und damit eine sinnvolle Gruppierung erreicht. Die Punkte, die auf diese Weise mindestens zwei Wurzeln (d.h. Minima) zugeordnet wurden, werden wie spezifiziert als Wasserscheide markiert. Diese Markierung erfolgt über einen speziellen Index, der sonst nicht vorkommt. Wie im folgenden Ausschnitt zu sehen ist, ist auch diese Funktion wieder zweigeteilt und rekursiv.

```
1 PunktRef Resolve(PunktRef p,Bild<Punktval> &im)
2 {
3     assert(p->sortetNeighbours.size());
```

```
4       double LS=p->sortetNeighbours.rbegin()->first;//Der hoechste Key ist maximaler Anstieg == LS(p)
5       PunktRef rep= PunktRef()+1;//Der Nachf. einer leeren Punktref. ist wiederum leer (aber ungleich)
6       for(multimap<double,PunktRef >::iterator i=p->sortetNeighbours.lower_bound(LS);
7           i!=p->sortetNeighbours.end() && rep!=PunktRef();
8           i++)
9       {
10          #define q    i->second
11          if(q!=p && q!=PunktRef())
12              q=Resolve(q,im,mark);
13          if(i==p->sortetNeighbours.lower_bound(LS))//erster Nachbar
14              rep=q;
15          else if(q!=rep)// Wurzel eines anderen Nachbarn ist verschieden => Wasserscheide
16              rep=PunktRef();
17          #undef q
18      }
19      assert(rep!=PunktRef()+1);
20      return rep;
21  }
22
23  void unionFind(deque<Punkt> &D,Bild<unsigned int> &lc,Bild<int> &lab,Bild<Punktval> &im)
24  {
25      foreach(Punkt,D,p)
26      {
27          PunktRef rep;
28          rep=Resolve(p,im,false);
29          if(rep!=PunktRef())
30              lab[p]=lab[rep];
31          else
32              lab[p]=WSHED;
33      }
34  }
```

Die äußere Funktion `findRoot` verwendet die rekursive Funktion `resolve`, um die eindeutige Wurzel jedes Punktes zu ermitteln. Ist die ermittelte Wurzel gültig, wird in dem Bild `idx` den Index seiner Wurzel vermerkt. Ist die Wurzel jedoch ungültig, geht die äußere Funktion davon aus, dass die Wurzel nicht eindeutig ermittelt werden konnte. In diesem Fall, wird der Punkt in `idx` als Wasserscheide eingetragen. Die innere Funktion [Zeilen 1-21] sucht für jeden übergeben Punkt `p` die Wurzel. Dazu untersucht sie alle Punkte `q` zu denen von `p` aus ein Bogen existiert und deren gemerkter Grauabstand $(f_{lc}(p) - f_{lc}(q))$ maximal ist $(q \in \Gamma(p))$ [Schleifenkopf Zeilen 6-8]. Ist `q` ein gültiger und normaler Punkt, wird seine Wurzel rekursiv bestimmt. Das Ergebnis ersetzt den LC-Nachbarn in der Variablen `q` [Zeilen 11-12]. Die folgenden Iterationen (weitere LC-Nachbarn) laufen bis Zeile 13 gleich ab. Es wird eine Wurzel für den LC-Nachbarn in dem Eintrag `i` gesucht, und in `q` abgelegt. In Zeile 15 wird nun jedoch die neu gefundene Wurzel mit der Wurzel verglichen, die der Vorherige durchlauf ergeben hat. Sind beide verschieden wird der Rückgabewert der Funktion ungültig. Die Schleife bricht in diesem Fall ab und gibt den ungültigen Wert zurück. Daraufhin wird die äußere Funktion noch den betrachteten Punkt wie beschrieben als Wasserscheide markieren.

Im Ergebnis besitzen alle Punkte entweder einen gültigen Index oder sind eine Wasserscheide.

6.2.4 Zusammenfassung

Die Umsetzung der *watershed*-Transformation mittels Wurzelsuche interpretiert ein plateaufreies Bild als Graphen auf einem plateaufreien Wertegebirge. Sie führt in diesem Graphen für jeden Knoten eine rekursive Wurzelsuche durch, wobei jedes lokale Minima eine Wurzel ist.

Hat ein als Punkt interpretierter Knoten niedrigere Nachbarn, bildet er Bögen zu den Nachbarn, zu denen der Unterschied des Grauwertes maximal ist. Knoten ohne niedrigere Nachbarn, gehören einem lokalen Minimum an. Sie bilden einen Bogen zu dem Punkt aus, der stellvertretend für das Minimum steht, dem sie angehören. Dies gilt ebenfalls für den stellvertretenden Punkt selbst. Damit sind alle Punkte eines lokalen Minimums (einer Senke) diesem Minimum zugeordnet. In dem nun vollständigen Graphen versuchen während der Wurzelsuche die restlichen Punkte über einen möglichst steilen Pfad einen Punkt zu erreichen, der bereits einem Minimum zugeordnet ist. Es wird nicht gefordert, dass dieser gefundene Punkt selbst Teil eines Minimums ist. Auf diese Weise können schon bekannte steilste Pfade wiederverwendet werden. Hat ein Punkt ersteinmal einen kürzesten Pfad gefunden, besteht eine gute Chance, dass Punkte in seiner Nähe diesen Pfad mitbenutzen können. Dabei zeigt sich jedoch ein Problem der rekursiven Definition des kürzesten Pfades zur Wurzel. Sie bewirkt, dass alle Punkte die im *upstream* einer Wasserscheide liegen selbst eine Wasserscheide sind. Allerdings ist gerade bei falschen Wasserscheiden die Wahrscheinlichkeit groß, dass Punkte in ihrem *upstream* liegen.

Eigentlich können Wasserscheiden nach ihrer Definition nur auf "Bergspitzen" vorkommen. Das gilt aber nur, wenn ein Punkt unendlich viele Nachbarn hat, denn nur dann kann er wirklich den steilsten Weg nach unten wählen. Je weniger Nachbarn ein Punkt hat (Konnektivität), umso weiter muss er mit seinem "Abstieg" vom Optimalen (steilsten) Weg abweichen. Unter ungünstigen Bedingungen kann das zu mehreren Pfaden zu mehreren Minima führen. Keiner dieser Pfade ist dabei wirklich der steilste, sie sind nur zufällig gleich weit vom eigentlichen Optimum entfernt. Der entsprechende Punkt wird trotzdem als Wasserscheide interpretiert. Diese falschen Wasserscheiden liegen, wie gesagt, nicht gezwungenermaßen auf einer Bergspitze, folglich gibt es mit großer Wahrscheinlichkeit Punkte in ihrem *upstream*.

Aus 5.3.1 auf Seite 51 ergibt sich, dass jeder Punkt, der bei seiner Wurzelsuche auf eine Wasserscheide trifft, selbst zu einer wird. Sein bisheriger Weg bis zu dem Wasserscheide-Punkt, war so steil wie möglich. Es ist auch bekannt, dass der von dem gefundenen Punkt ausgehende Pfad maximal steil ist, weshalb der gesamte Pfad der steilstmöglichste Pfad zu der entsprechenden Wurzel ist. Folglich kann der suchende Punkt die Zuordnung

des aktuellen Punktes übernehmen, denn aufgrund der Definition würde er wieder den gleichen Pfad finden. Durch die Übernahme der Zuordnung spart er sich den weiteren Weg nach unten, und damit Rechenzeit. Ist der aktuelle Punkt aber eine falsche Wasserscheide, ist er also fälschlicherweise mindestens zwei Minima zugeordnet, pflanzt sich dieser Fehler nun unkontrolliert fort. Falsche Wasserscheiden "wachsen" so zusätzlich an. Der Fehler dabei liegt nicht in der "Abkürzung", sondern in der rekursiven Definition der Wurzelsuche. Er lässt sich daher nicht vermeiden.

Die Form der Senken in einem Bild und die Konnektivität beeinflussen wieviele Punkte fälschlicherweise als Wasserscheide deklariert werden. Die Abbildungen 6.1 und 6.2 auf der nächsten Seite illustrieren dies (schwarze Bereiche sind Wasserscheiden).

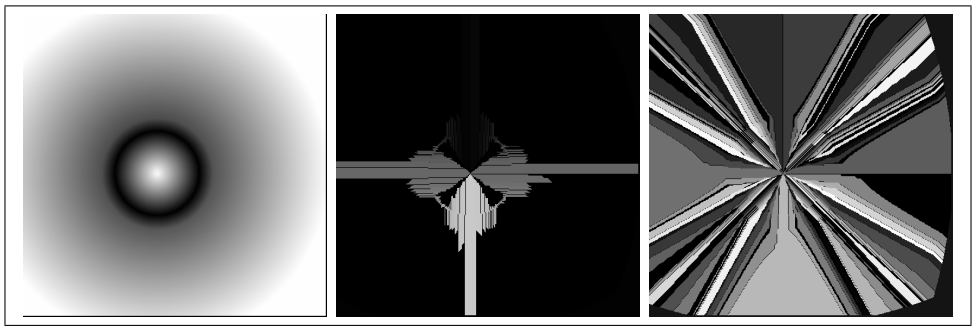

Abb. 6.1: *watershed*-Transformation eines ungünstigen Bildes mittels Wurzelsuche

Die Abb. 6.1 zeigt sehr deutlich, wie ein ungünstiges Bild zu großen Fehlern in den *watershed*-transformierten Bildern führen kann. In dem gezeigten Fall sorgt die Form der Senke im Original (der schwarze Ring im linken Bild) für Probleme bei der Zuordnung der Punkte. Im mittleren Bild können nur senkrechte und waagerechte Pfade eindeutig als kürzeste Pfade erkannt werden. Die Punkte kennen dort nur die Nachbarn neben, und über ihnen. Im rechten Bild kommen noch die diagonalen Nachbarn hinzu. Wie man sieht entschärft diese Erhöhung der Konnektivität das Problem falscher Wasserscheiden zwar merklich, kann ihre Entstehung aber nicht verhindern.

Die Abb. 6.2 auf der nächsten Seite zeigt den umgekehrten Fall. Durch die günstige Form der Senke in der Mitte entstehen dort keine falschen Wasserscheiden. Lediglich die tiefen Ränder des Bildes verursachen waagerecht bzw. senkrecht verlaufende Wasserscheiden wo keine sein dürften. Im Gegensatz zur vorherigen Abbildung kommen sie hier aber sehr viel seltener vor. Die Erhöhung der Konnektivität hat bei diesem Beispiel keinen Einfluss.

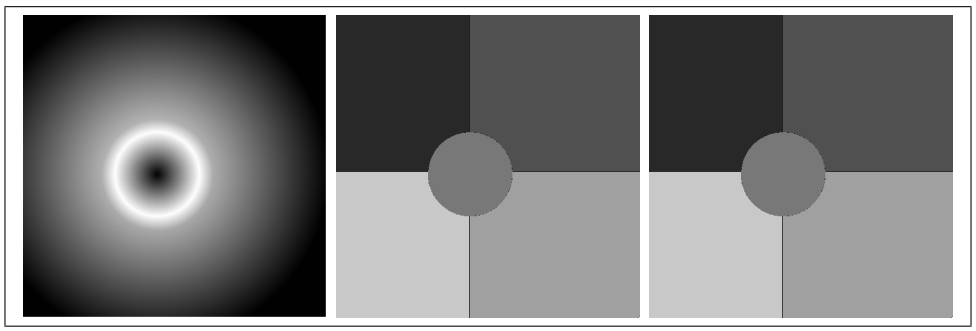

Abb. 6.2: *watershed*-Transformation eines günstigen Bildes mittels Wurzelsuche

Die Vorteile der *watershed*-Transformation mittels Wurzelsuche liegen in ihrer relativ hohen Geschwindigkeit und ihrer einfacheren Parallelisierbarkeit. Sie hat jedoch den Nachteil, dass sie unter ungünstigen Bedingungen eine hohe Fehlerrate aufweist.

6.3 Watershed-Transformation mittels Absenken

Da die Implementation der *watershed*-Transformation als Wurzelsuche [13] unter bestimmten Umständen zu großen Erkennungsfehlern führt, sollte die Anwendung eine Alternative Implementierung bieten, die dieses Problem nicht hat. Die dazu ausgewählte Implementierung nach Vincent und Soille [20] basiert statt der Wurzelsuche auf dem Versenken des Grauwertgebirges (in Wasser). Die Pfade zu den Minima werden nicht für jeden Punkt einzeln, sondern Schritt für Schritt für alle Punkte gleichzeitig gebildet. Die Vervielfachung falsch erkannter Wasserscheiden wird auf diese Weise vermieden. Da außerdem Plateaus korrekt behandelt werden, ist eine Vortransformation des Bildes nicht nötig. Dieser Algorithmus besteht jedoch aus mehreren teilweise ineinander verschachtelten Schleifen. Sie besitzt daher theoretisch eine höhere Zeitkomplexität als die *watershed*-Transformation mittels Wurzelsuche.

6.3.1 Implementation nach Vincent-Soille

Für die *watershed*-Transformation mittels Absenken müssen sämtliche Punkte des Bildes ($p \in D$) aufsteigend nach ihrem Grauwert $f(p)$ sortiert sein. Der Vincent-Soille-Algorithmus besteht hauptsächlich aus einer großen Schleife, die den Pegel $h \in \mathbb{N}$ sukzessive von h_{min} nach h_{max} durchläuft. Das entspricht einem "Versenken" des Grauwertgebirges und

sollte nicht mit einem "Volllaufen" des selbigen verwechselt werden. Die "Pegelhöhe" ist im gesamten Gebirge zu jedem Zeitpunkt gleich. Der folgende Codeausschnitt zeigt den Funktionskopf, einige nötige Initialisierungen und den Anfang dieser Schleife.

```
1  #define INIT -1
2  void calc_cassic(
3       deque<Punkt> &D,Bild<int> &lab,Bild<Punktval> &im,unsigned int size_x,
4       unsigned int size_y, VImage &src,VImage &dst,VAttrList &out_list,bool inv
5       )
6  {
7       init(size_x,size_y,src,im,D,inv);
8       Bild<unsigned int> dist(size_x,size_y,0);
9
10      foreach(Punkt,D,p)lab[p]=INIT;
11
12      ptr_fifo fifoA;
13      unsigned int curlab=0;
14      unsigned int curdistA;
15
16      list< PunktRef>::iterator aktP=D.begin();
17      unsigned short h_min=im[**D.begin()];
18      unsigned short h_max=im[**D.rbegin()];
19
20      for(Punktval h=h_min;h<=h_max;h++)
21      {
22          curdistA=1;
```

Wie im nächsten Ausschnitt zu sehen, werden innerhalb der Schleife zuerst alle Punkte p mit $f(p) = h$ untersucht. Dabei bekommt jedes neue lokale Minima einen eindeutigen Index, den es später an alle Punkte in seinem Einzugsbereich weitergibt. Außerdem werden alle Punkte mit $f(p) = h$ markiert. Da die Punkte in der Liste nach $f(p)$ sortiert sind, muss die Liste nicht durchsucht werden, um alle Punkte mit $f(p) = h$ zu finden. Hat einer dieser Punkte einen Nachbarn, der bereits mindestens einer Einflusszone angehört, wird er an eine FIFO-Liste angehängt. In dieser Liste stehen nun alle "Uferpunkte" des aktuellen Pegelstandes. Als Abschluss wird an die FIFO-Liste eine Trennmarkierung angehängt.

```
23          for(list< PunktRef>::iterator p=aktP;p!=D.end() && im[*p]==h;p++)
24          {
25              lab[*p]=MASK;    //Alle Punkte mit f(p)=h markieren
26              bool cont=false;
27              //Alle ''Uferpunkte'' (Punkte an Grenze zu Basins) in fifo-Listen ablegen
28              foreach(PunktRef,(*p)->neighboursA,q)if(lab[*q]>0 || lab[*q]==WSHED)
29              /*wenn p einen Nachbar q hat, der zu mindestens einem erkannten Basin gehoert
30                -zu einem Unbekannten kann er nich gehoeren
31                -gehoert der Nachbar zu einem Basin, hat per def ein niedrigeres h, da er schon vorher
32                 bearbeitet worden sin muss
33                -steilster anstieg ist Implizit, da niedrigste Pkt zuerst gelabelt werden
34              */
35              {
36                  dist[*p]=1;//Abstand zum naechsten Punkt, der schon "unter wasser" steht
37                  fifoA.push(*p);
38                  break;
39              }
40          }
41          fifoA.push_null();
```

Diese "Uferpunkte", d.h. alle Punkte $p \in fifoA$ müssen nun indiziert werden. Das heißt, sie müssen entweder eindeutig einem Bassin zugeordnet, oder als Wasserscheide identifiziert werden. Diese Aufgabe wird von der im nächsten Ausschnitt gezeigten Schleife

erfüllt, indem sie jeweils einen Eintrag aus der FIFO-Liste der "Uferpunkte" herausnimmt und dessen Umgebung untersucht. Ist der aktuelle Punkt `p` eine Trennmarkierung, liegen für die aktuelle Ausbreitung des Wassers keine (weiteren) "Uferpunkte" vor. Das Wasser muss sich in diesem Fall einen Schritt auf Plateaus ausbreiten, wenn in dieser Pegelhöhe Plateaus existieren. Die Schleife wird dann neu gestartet, um die neuen "Uferpunkte" zu bearbeiten [Zeilen 45-52]. Auf diese Weise wird das Plateauproblem gelöst. Ist der aktuelle Punkt keine Trennmarkierung werden seine Nachbarn in den Zeilen 53 bis 77 untersucht. Ist einer seiner Nachbarn schon indiziert worden, und ist er selbst noch keinem Bassin zugeordnet, übernimmt er dessen Zuordnung. Hat er dagegen schon einen anderen Index, ist er folglich gleich weit von zwei Bassins entfernt und wird als Wasserscheide markiert. Trifft nichts davon zu, übernimmt der untersuchte Punkt ggf. die Wasserscheide-Markierung seines Nachbarn. Nachbarn, für die zwar $f(p) = h$ gilt, die aber nicht zum aktuellen "Ufer" gehören (erkennbar an `dist[*q]==0`) werden mit erhöhtem Abstand hinter der Trennmarkierung an die FIFO-Liste angefügt. Sie werden verarbeitet werden, sobald die Ausbreitung des Wassers ausreichend erhöht wurde.

```
43          for(PunktRef p=fifoA.pop();fifoA.size();p=fifoA.pop())
44          {
45              if(p==PunktRef())
46              /* wenn keine ''Uferpunkte'' mehr da, lassen wir das Wasser IN DER EBENE weiterlaufen
47                 (die Pegelhoehe wird hier NICHT erhoeht)*/
48              {
49                  fifoA.push_null();
50                  curdistA++;
51                  continue;
52              }
53              foreach(PunktRef,p->neighboursA,q) // p entsprechend seinen Nachbarn Indizieren
54              {
55                  if(dist[*q]<curdistA && (lab[*q]>0 || lab[*q] == WSHED))
56                  //q ist schon indiziert (gehoert zu mind. einem Basin) und erreichbar
57                  {
58                      if(lab[*q]>0)//q gehoert eindeutig zu EINEM Basin
59                      {
60                          //wenn p nich eindeutig einem Basin zugeordnet ist
61                          if(lab[p] == MASK || lab[p] == WSHED)
62                              lab[p]=lab[*q];//p wird dem Basin von q zugeordnet
63                          else
64                              //wenn Basin von p nicht das Selbe wie das von q ist.
65                              if(lab[p]!=lab[*q])lab[p]=WSHED;
66                      }
67                      else //Wenn q eine Wasserscheide ist
68                          //wird p auch eine, wenn es noch keinem Basin zugeordnet ist
69                          if(lab[p]==MASK)lab[p]=WSHED;
70                  }
71                  else if(lab[*q]==MASK && dist[*q]==0)
72                  //q hat selbe Hoehe wie p, ist aber nicht Teil des aktuellen ''Ufers'' => Plateau
73                  {
74                      dist[*q]=curdistA+1;//Punkte eines Plateaus sind weiter weg vom Wasser
75                      fifoA.push(*q);
76                  }
77              }
78          }
```

Alle Punkte p mit $f(p) = h$ die nach dieser Schleife noch nicht indiziert wurden, haben nur höhere Nachbarn, sie sind somit Minima. Der folgende letzte Ausschnitt zeigt die Schleife, die nocheinmal über alle Punkte p mit $f(p) = h$ geht und sie daraufhin unter-

sucht. Dabei wird auch der Puffer für die Abstände zurückgesetzt. In den Zeilen 85 bis 98 bekommen jedes neue Minima und seine Nachbarn gleicher Höhe einen eindeutigen Index. Im Gegensatz zu der ersten inneren Schleife [Zeilen 23-40] ändert diese Schelife den globalen Iterator `aktP` direkt, sodass dieser nach ende der Schleife den ersten Punkt p mit $f(p) > h$ referenziert.

```
80          //neue lokale Minima erkennen
81          for(;aktP!=D.end() && im[*aktP]==h;aktP++)
82          {
83              dist[*aktP]=0;
84              //Alle Punkte, die jetzt noch keinem min zugeordnet werden konnten, sind lokale Minima
85              if(lab[*aktP]==MASK)
86              {
87                  lab[*aktP]=++curlab;
88                  fifoA.push(*aktP);
89                  while(fifoA.size())
90                  {
91                      PunktRef q=fifoA.pop();
92                      foreach(PunktRef,q->neighboursA,r)if(lab[*r]==MASK)
93                      {
94                          fifoA.push(*r);
95                          lab[*r]=curlab;
96                      }
97                  }
98              }
99          }
100     }
101 }
```

Damit ist der Block der Hauptschleife zu Ende. Der Pegel h wird um eins erhöht und alle Operationen beginnen von vorn. Existieren für einen Pegel h keine Punkte mit $f(p) = h$, wird die erste Schleife nicht ausgeführt, da $f(p) \neq h$. Dadurch steht in der FIFO-Liste nur eine Trennmarkierung, die sofort bei der Initialisierung der zweiten Schleife [Zeile 43] herunter genommen wird. Die Bedingung `fifoA.size()` kann d.h. nicht mehr erfüllt werden, und die Ausführung der Schleife wird abgewiesen. Für die letzte Schleife gilt das Gleiche wie für die Erste, sie wird nicht ausgeführt, da bereits für den ersten betrachteten Punkt p $f(p) \neq h$ gilt. Für dieses h mit $\forall p \in D : f(p) \neq h$ führt die Hauptschleife daher keine Operationen aus, sondern erreicht sofort das Ende der Schleife und erhöht h.

Ist der Pegel $h = h_{max}$ erreicht, wurden alle Punkte $p \in D$ behandelt, alle lokalen Minima des Gebirges wurden gefunden und indiziert sowie alle anderen Punkte ihrem nächsten Minima zugeordnet oder als Wasserscheide markiert.

6.3.2 Erhöhung der Konnektivität

Im Gegensatz zur Wurzelsuche kennt die Implementation nach Vincent-Soille keine Kostenfunktion. Stattdessen wird der Geodätische Abstand zweier Nachbarn hier fest als 1 angenommen. Der Abstand diagonaler Nachbarn, wie sie bei einer Konnektivität von 8

auftreten, beträgt jedoch $\sqrt{2}$. Daraus folgt, dass der Algorithmus, wie er hier implementiert wurde, nur für eine Konnektivität von 4 geeignet ist. Das Fehlen der Kostenfunktion zeigt sich beim Ausbreiten des Wassers in der Ebene. Es lässt sich umgehen, indem diese Ausbreitung in mehreren Schritten ausgeführt wird. Jeder dieser Schritte ist dann für je eine "Abstandsart" ausgelegt. Zu diesem Zweck sind gesonderte FIFO-Listen und `curdistX`-Variablen notwendig.

6.3.3 Ergebnisse

Da auch die Implementation nach Vincent-Soille weiterhin auf diskreten Bildern bzw. diskreten Graphen angewendet wird, werden auch weiterhin falsche Wasserscheiden erkannt. Es kommt auch hier vor, dass statt dem eigentlich kürzesten Pfad zwei suboptimale Pfade gefunden werden. Dieses grundlegende Problem wurde schon besprochen und lässt sich schwer unterbinden, da die Diskretität eine grundsätzliche Eigenschaft der Eingabedaten ist. Bei Vincent-Soille können sich diese Fehler allerdings nicht rückwärts über die Wurzelsuche fortpflanzen, da es hier keine Wurzelsuche gibt. Auch Plateaus müssen nicht künstlich in Hänge verwandelt werden, sie werden bei dieser Implementation intuitiver über das Ausbreiten des Wassers in der Ebene behandelt.

Eine Transformation nach Vincent-Soille erweist sich daher als relativ unempfindlich gegenüber ungünstigen Bildern. Die Abb. 6.3 auf der nächsten Seite zeigt die Ergebnisse einer Anwendung auf das gleiche ungünstige Bild, in dem die Wurzelsuche über 45000 Wasserscheiden fand. Analog zu Abb. 6.1 auf Seite 64 und Abb. 6.2 auf Seite 65 wurde für das mittlere Bild eine vierfache Konnektivität verwendet, während das rechte Bild das Ergebnis einer Transformation mit achtfacher Konnektivität ist. Das Linke Bild ist wie immer das Original. Vor allem am mittleren Bild erkennt man die Robustheit von Vincent-Soille, beziehungsweise im Vergleich dazu das totale Versagen der Wurzelsuche. Während die Wurzelsuche fast das gesamte Bild als Wasserscheide "erkennt" (ca. 45000 Pixel eines 256x256 Pixel großen Bildes wurden als Wasserscheide markiert), zeigt Vincent-Soille ein ähnliches Verhalten wie bei der achtfachen Konnektivität. Es werden nur an den "üblichen" Punkten falsche Wasserscheiden gefunden (ca. 13000 Wasserscheiden in einen 256x256 Pixel großen Bild). Es fällt außerdem auf, dass die Schwarzen Linien, die die Wasserscheiden darstellen hier nie mehr als einen Pixel breit sind.

Abb. 6.4 auf der nächsten Seite zeigt zum Vergleich das Verhalten von Vincent-Soille auf einem günstigen Bild. Hier unterscheiden sich beide Implementationen kaum. Sowohl bei einer vierfachen, als auch bei einer achtfachen Konnektivität fand die Wurzelsuche ca. 1000 Wasserscheiden. Vincent-Soille liegt mit 989 Wasserscheiden nur knapp darunter.

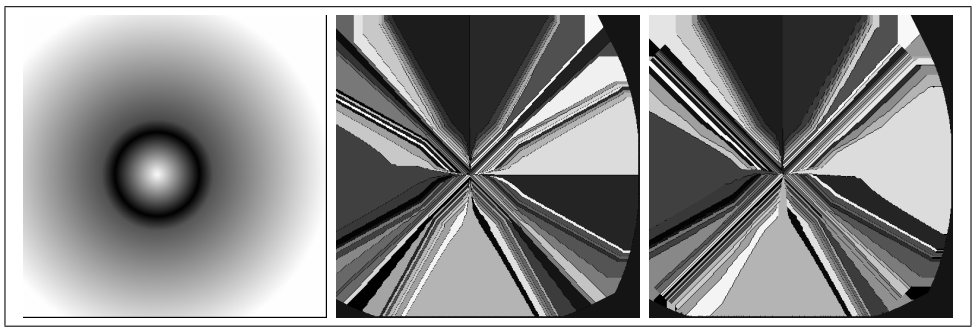

Abb. 6.3: *watershed*-Transformation eines ungünstigen Bildes mittels Vincent-Soille

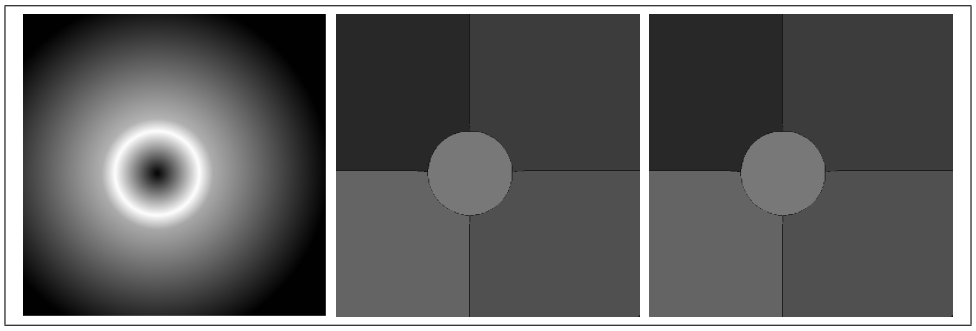

Abb. 6.4: *watershed*-Transformation eines günstigen Bildes mittels Vincent-Soille

6.4 Watershed in dreidimensionalen Bildern

Bisher wurde der Einfachheit halber in den Betrachtungen zu den Implementationen implizit von zweidimensionalen Bildern ausgegangen. Die Messdaten liegen dagegen dreidimensional vor. Die Implementierungen müssen daher noch an dreidimensionale Bilder angepasst werden.

6.4.1 Anpassung der Algorithmen

Die Anpassung der Definition der *watershed*-Transformation erweist sich als relativ unkompliziert. Die *watershed*-Transformation basiert grundsätzlich auf Graphen von Punkten als Eingabedaten. Um die *watershed*-Transformation auf dreidimensionale Bilder auszulegen reicht es daher für den Ausgangsgraphen $G = (D, E, f)$ festzulegen, dass $D \subseteq \mathbb{Z}^3$ gilt. In einem solchen Graphen hat ein Punkt sechs direkte Nachbarn, der Abstand zu diesen Nachbarn beträgt 1. Liegt der Punkt im Zentrum eines, innerhalb dieses

Graphen gedachten, Würfels mit der Kantenlänge von 2, so liegen diese Nachbarn jeweils im Mittelpunkt einer seiner Seitenflächen. Weitere acht Nachbarn kommen als diagonale Nachbarn auf den Koordinatenebenen dazu. Sie liegen in der Mitte der Würfelkanten und ihre Entfernung zum zentralen Punkt beträgt $\sqrt{2}$. Die acht Eckpunkte des Würfels stellen mit einem Abstand von $\sqrt{3}$ die am weitesten vom zentralen Punkt entfernten Nachbarn dar. Statt vier bzw. acht Nachbarn sind bei dreidimensionalen Graphen Folglich sechs, vierzehn oder zweiundzwanzig Nachbarn mit drei verschiedenen Abständen zu berücksichtigen. Jede Implementierung der *watershed*-Transformation stützt sich auf diese Datenstruktur, und ist daher theoretisch wie die mathematische Definition der Transformation unabhängig von der Dimensionalität der Daten.

Dementsprechend beschränkt sich die konkrete Anpassung der Wurzelsuche [13] an dreidimensionale Daten auf die Verwendung einer, der Lage des Nachbarn entsprechenden Kostenfunktion. Weiterhin müssen lediglich entsprechend mehr Nachbarn eines Punktes berücksichtigt werden. Die bei der Implementierung für die Speicherung der Nachbarn verwendeten Listen passen ihre Länge automatisch den Anforderungen an, sodass an dieser Stelle keine Änderung vorgenommen werden muss. Es erweist sich jedoch als praktischer die Nachbarn von vornherein entsprechend ihrer Lage in verschiedene Listen einzusortieren. Damit ist die Art ihrer Nachbarschaft bekannt, und sie muss bei Bedarf nicht neu ermittelt werden. Im Ansatz ist dieses Vorgehen auch schon in der gezeigten zweidimensionalen Implementation der Wurzelsuche zu sehen. Für die diagonalen Nachbarn in einem zweidimensionalen Graphen mit achtfacher Konnektivität wurde dort eine gesonderte Liste `neighboursB` eingeführt. Die darin liegende Information über die Entfernungen wurde in der Funktion `buildGraph` für die Kostenberechnung verwendet. Für die Verarbeitung dreidimensionaler Graphen mit zweiundzwanzigfacher Konnektivität reicht es daher analog eine weitere Liste `neighboursC` hinzuzufügen, und entsprechend zu verwenden.

Wie in Abschnitt 6.3.2 auf Seite 68 beschrieben sind Entfernungen zwischen Nachbarn, die einen anderen Wert als 1 haben für die Implementation nach Vincent-Soille [20] problematisch. Für die Verarbeitung dreidimensionaler Graphen muss daher analog zu der dort beschriebenen Lösung ein weiterer Schritt für die Ausbreitung des Wassers eingeführt werden. Die Alternative wäre, sich bewusst auf eine sechsfache Konnektivität zu beschränken. Eine solche Implementation ist merklich schneller als eine Implementation, die zweiundzwanzig Nachbarn berücksichtigen muss. Die Implementation nach Vincent-Soille hat sich auch bei geringer Konnektivität als robust erwiesen. Unter Umständen ist es daher sinnvoll einen kleinen zusätzlicher Fehler zugunsten der einfacheren Implementation und der Geschwindigkeit in Kauf zu nehmen.

6.4.2 Reduzierung des Speicherbedarfs

Die in den Abschnitten 6.2 auf Seite 56 und 6.3.1 auf Seite 65 beschriebenen Implementationen sind bewusst naiv gehalten worden, sie sind somit gleichzeitig leicht verständlich, und robust. Werden sie unverändert für die *watershed*-Transformation dreidimensionaler Daten übernommen wächst mit der Anzahl der Punkte auch der Speicherbedarf exponentiell an. Zum Beispiel belegt die im Abschnitt 6.1.2 auf Seite 55 beschriebene Datenstruktur zur Repräsentation eines Punktes 0.00015259 MByte. Das erscheint zunächst wenig, jedoch besteht eine typische MRT-Aufname des Kenrspintomografen des CBS derzeit aus 5.12 Millionen Punkten ($160 \times 160 \times 200$ Punkte), was einen Speicherbedarf 781.26 MByte allein für die Punkte ergibt. Dazu kommen der *overhead* durch die Struktur der verwendeten Listen sowie der Speicherbedarf der Bilddaten und Zwischenergebnisse. Der Speicherbedarf des gesamten Systems kann so schnell mehrere GByte betragen. Dies steht klar im Wiederspruch zu dem Anspruch des Programms auch auf einfachen Arbeitsplatzrechnern anwendbar zu sein. Der Speicherbedarf der *watershed*-Implementierungen muss also für den praktischen Einsatz drastisch verringert werden.

Wie schon beschrieben "verschwendet" die Datenstruktur `Punkt` den meisten Speicher. Instanzen dieser Datenstruktur kommen in sehr großer Zahl vor. Jedes hier eingesparte Byte kann daher mehrere MByte Gesamtersparnis bringen. Es fällt auch auf, dass sowohl die Nachbarn, als auch die Koordinaten eines Punktes jederzeit ermittelt aus seiner Position in der Liste der Punkte und den Dimensionen des Datensatzes ermittelt werden kann. Ein Punkt wird somit nur noch durch seinen Index in der Liste der Punkte repräsentiert. Er ist daher nur noch 4 Byte groß, und die 5.12 Millionen Punkte eines $160 \times 160 \times 200$-Bildes belegen nur noch knapp 20 MByte. Die somit nötigen Integeroperationen zur Bestimmung der fehlenden Daten verlangsamen die Algorithmen zwar etwas, dies ist jedoch angesichts einer Einsparung von fast 95% vertretbar. Die Speicherung der Punkte in dynamischen Listen ist oft unnötig, da sich die Länge der meisten in den Implementationen vorhandenen Listen kaum ändert. Ein einfaches Array reicht in diesen Fällen aus, und bringt kaum *overhead* mit. Die Datenstruktur `Bild` wird dagegen beibehalten, denn sie ist aus Speichersicht bereits optimal. Ihre Schnittstelle muss auch nicht angepasst werden, denn der Container `Punkt` besteht weiterhin, und kann benutzt werden. Lediglich die Nachbarlisten werden entfernt. Statt Punkten werden wie beschrieben nur deren Indexe in den Punktlisten gespeichert.

Auf die Implementation der *watershed*-Transformation nach Vincent-Soille [20] angewendet haben diese Anpassungen folgende Auswirkungen. Ein einfaches Array hält die Indexe der Punkte von 0 bis $n-1$, wobei n die Anzahl der Punkte ist. Diese Liste wird

nach den Grauwerten sortiert. Dabei werden aus jedem Index die Koordinaten bestimmt. Die Koordinaten werden verwendet, um den Grauwert des Entsprechenden Punktes mittels *vista*-Funktionen ([21] siehe auch Abschnitt 7 auf Seite 75) direkt aus den Messdaten zu bestimmen. Sind die Punkt-Indexe in der Liste sortiert, kann der eigentliche Algorithmus beginnen. Er arbeitet die Liste der Indexe sukzessive ab, und bestimmt bei Bedarf die Koordinaten der Punkte wie beschrieben. Die in der ursprünglichen Implementation verwendete FIFO-Liste bleibt als einzige in dieser Form erhalten. Sie kann nie mehr Punkte beinhalten, als in einer Ebene der Volumendaten vorkommen. Außerden werden regelmäßig Punkte zu ihr hinzugefügt bzw. entfernt, sodass an dieser Stelle ein einfaches Array ungeeignet ist. Die verwendeten *vista*-Funktionen [21] können direkt verwendet werden, um den Grauwert eines Punktes zu ermitteln. Da sie die Volumendaten schon von sich aus puffern, ist es nicht nötig diese zwischen zu speichern. Der Container für die Farbwerte (im Beispiel `im`) kann daher entfallen. Zur Optimierung des Containers für die Markierungen (im Beispiel `lc`) wird die Konstante `INIT` zur Markierung unbearbeiteter Punkte von -1 auf den größtmöglichen Wert des verwendeten Datentyps geändert. Dies Spart das Vorzeichenbit ein, und ermöglicht eine effektivere Ausnutzung des Wertebereichs. Zeigt sich in der Praxis, dass nie mehr als $2^16 - 2$ Minima vorkommen, kann der verwendete Datentyp auf 16 Bit reduziert werden. Vorerst wird jedoch darauf verzichtet. Die einzigen größeren durch die *watershed*-Transformation angelegten Objekte im Speicher sind somit die sortierte Punktliste und das Ergebnissbild mit den indizierten Punkten. Der Speicherbedarf der Implementation der *watershed*-Transformation nach Vincent-Soille reduziert sich durch diese Optimierungen auf weniger als 50 MByte. Die Voraussetzungen für die Anwendbarkeit auf üblichen Desktop-Rechnern sind daher für diesem Fall wieder erfüllt.

Die Implementation der Wurzelsuche [13] kann ähnlich angepasst werden. Die Punktliste wird hier nicht sortiert, die Reihenfolge der Punkte ist irrelevant. Eine sortierte Punktliste ist daher überflüssig, die Punkte können direkt über ihre Koordinaten im Bild angesprochen werden. Für jeden Punkt muss jedoch eine Liste aller Punkte in seinem *downstream* angelegt werden. Diese Listen können je nach verwendeter Konnektivität eine Maximallänge von 6, 14 oder 22 Punkten erreichen. In der Regel werden sie aber weit weniger Punkte beinhalten, da es nur selten vorkommt, daß ein Punkt im *upstream* all seiner Nachbarn liegt. Eine feste Ausage über die Länge dieser Listen ist daher nicht möglich. Ist eine solche Liste allerdings einmal gefüllt, ändert sich ihre Länge nicht mehr. Die Listen der Nachbarn im *downstream* eines Punktes können daher als Arrays implementiert werden, die bei der Erkennung der Nachbarn im *downstream* mit maximaler Länge (6,14 oder 22) initialisiert werden. Die erkannten Nachbarn im *downstream* des Punktes werden bei der Erkennung in diesem Array abgelegt. Ist die nötige Länge

der Liste am Ende der Operation bekannt, wird das Array entsprechend gekürzt. Desweiteren speichert die Wurzelsuche Zwischenergebnisse einiger Operationen als Bild ab, diese Bilder sind in den meisten Fällen temporär. Sie können daher gelöscht bzw. überschrieben werden, wenn sie nicht mehr benötigt werden. Das Überschreiben der Bilder ist bei gleichem Datentyp problemlos möglich, da die Anzahl der Punkte eines Bildes in jedem Fall konstant bleibt. Der Speicherbedarf dieser Implementaion der *watershed*-Transformation kann vor allem aufgrund der *downstream*-Listen lediglich auf unter 300 MByte reduziert werden. Es wird angenommen, dass übliche Desktop-Rechnern über 512 MByte Arbeitsspeicher verfügen. Die Voraussetzungen für die Anwendbarkeit auf solchen Rechnern kann somit knapp erfüllt werden. Ein Speicherbedarf von unter 50 MByte wäre realisierbar, wenn die Nachbarn im *downstream* nicht gespeichert würden. Da sie in diesem Fall bei Bedarf immer wieder neu ermittelt werden müssten, würde diese "Optimierung" die Transformation enorm verlangsamen. Die Listen bleiben deshalb vorerst erhalten.

7 Implementierung des Visualisierungstools

Bei der Implementierung des eigentlichen Visualisierungstools wird auf die Basisbibliothek, den *QT*-Widgetadapter sowie die, an die Anwendung auf Volumendaten angepassten Implementierungen der *watershed*-Transformation zurückgegriffen. Die grafische Schnittstelle basiert, wie bereits erklärt auf *QT* und die für das Programm verwendeten Daten liegen im *vista*-Format vor. Zum Lesen und Schreiben dieser Daten greift das Programm daher auf die Funktionen der frei verfügbaren *vista*-Bibliothek [21] zurück.

7.1 Handhabung der Volumendaten

Das *vista*-Format organisiert grafische Daten als sequentielle Datensätze beliebiger Anzahl in Blöcken innerhalb einer *vista*-Datei. Jeder Eintrag eines solchen Blockes entspricht dabei einem Voxel des Datenraumes und damit einem Messpunkt im reellen Raum. Die Kantenlängen des gemessenen Quaders sowie die Kantenlängen der einzelnen Voxel sind in zusätzlichen Informationsfeldern innerhalb der entsprechenden Blockes gespeichert.

Gelesen werden diese Daten direkt von einer Instanz der Klasse `GLvlVolumeTex`. Wie Abb. 7.1 auf der nächsten Seite zeigt, ist `GLvlVolumeTex` abgeleitet von `SGLBaseTex`, der Basisklasse für Texturcontainer aus der Basisbibliothek.

Von `SGLBaseTex` erbt `GLvlVolumeTex` die nötigen Methoden zur Verwaltung von *OpenGL*-Texturen. Das Laden der Volumendaten erfolgt durch die *vista*-Laderoutinen.
`GLvlVolumeTex` lädt beim Erzeugen einer Instanz der ihr übergeben Datei den Datensatz, der die gemessenen anatomischen Daten enthält. Zusätzlich zu diesen hoch aufgelösten Daten der Gewebestrukturen eines kompletten Gehirns, liegen den *vista*-Dateien meist noch funktionelle Daten bei. Diese Informationen über die Aktivitäten bestimmter Bereiche des selben Gehirns über eine bestimmte Zeitspanne hinweg liegen

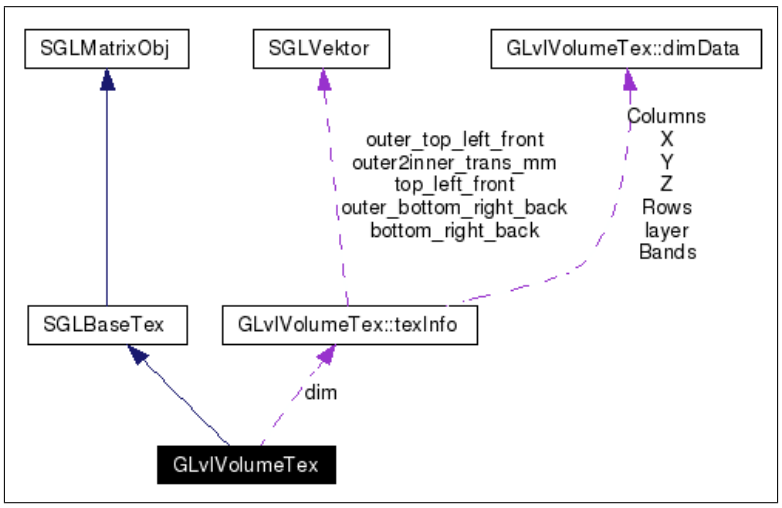

Abb. 7.1: `GLvlVolumeTex` und ihre Beziehungen

in einer geringeren Auflösung vor und umfassen meist auch nur einen Ausschnitt des Gehirns. Sie werden später unter Verwendung von *Multitexturing* beim Zeichnen der Schnittfläche "über" die Anatomischen Daten geblendet.

Die Volumendaten können in verschieden Datentypen vorliegen. `GLvlVolumeTex` muss dem Rechnung tragen und die jeweils optimale Laderoutine anwenden. Während anatomische Daten aus ganzzahligen Werten bestehen, sind funktionelle Daten in der Regel Fließkommazahlen. So werden anatomische Daten, die als Grauwerte zwischen 0 und 255 vorliegen, als Indexwerte einer vorgegeben Farbpalette geladen. Die Fließkommazahlen, welche die Aktivitäten im funktionellen Datensatz repräsentieren, lassen sich schlechter einem bestimmten Index zuweisen. Sie werden deshalb als Parameter einer vorgegebenen Funktion verwendet, die an den entsprechenden Stellen in der *OpenGL*-Textur Echtfarben zuweist.

Da die spätere Schnittfläche beliebig im Datenraum platziert werden kann, wird sie dort unter Umständen auch herausragen. In einem solchen Fall muss der Renderer beim Zeichnen der Textur auf die Schnittfläche ein definiertes Verhalten zeigen. Der Renderer wird deshalb so konfiguriert, dass er beim Zeichnen eines Pixels zu dem die Texturinformationen fehlen, die Informationen des am nächsten liegenden verfügbaren Voxels der Textur anwendet. Der Rand der Textur wird dadurch bei einer herausragenden Schnittfläche unendlich oft wiederholt. Aus diesem Grunde ist die Textur an allen Seiten um mindestens einen Voxel größer der entsprechende Datensatz. Dieser Voxel hat im Alpha-Kanal den Wert 0. Der *Renderer* ist dabei so konfiguriert, daß er derartige Punkte nicht

zeichnet.

Einfache *OpenGL*-Renderer können nur Texturen verarbeiten, deren Länge 2^n mit $n = \{1, 2, 3, \ldots\}$ entspricht. Deshalb kann die Größe der Textur nur eingeschränkt an die Größe der vorliegenden Daten angepasst werden. Die Daten werden stattdessen in eine Textur mit der nächstgrößeren Potenz von zwei gelegt. Wie schon erwähnt wird jeweils an einer Seite ein Voxel ausgespart, während auf der Gegenseite alle übrigen Voxel leer bleiben. Liegen Daten mit der Kantenlänge 500 vor, wird eine Textur der Kantenlänge $2^9 = 512$ angelegt. Daraus folgt, dass die eigentlichen Daten darin erst bei 1 beginnen und bei 501 enden. Die übrigen Voxel (0 und $501 - 511$) werden mit leeren Einträgen ($Alpha \leftarrow 0$) aufgefüllt. Diese Spezifikationen werden auf alle drei Dimensionen angewandt.

Die Einträge 1 bis 255 der Farbpalette für die anatomischen Daten werden mit grauen RGBA-Werten $((1, 1, 1, 255), \ldots, (255, 255, 255, 255)$ gefüllt. Die eigentliche Textur braucht daher wie bereits erwähnt aus nur 8 Bit großen Indexen bestehen, deren Werte $1 - 255$ in dem entsprechenden Grauwert dargestellt werden. Der ersten Eintrag der Farbpalette $((0, 0, 0, 0))$ ist ein Sonderfall, sein Alpha-Wert (vierter Wert) ist Null. Voxel mit dem Index 0 werden daher beim Zeichnen den Alpha-Wert ergeben. Sie werden somit nicht gezeichnet. Auf diese Weise können Voxel, die nicht gezeichnet werden sollen, markiert werden, ohne ein zusätzliches Bit dafür zu benötigen. Im Datensatz kommt der Wert 0 nur im "Hintergrund" vor, und dieser soll ohnehin nicht gezeichnet werden. Die anatomischen Daten stellen den mit Abstand größten Datensatz dar, diese Konzepte sparen Speicher bei der Ablage der Daten im Texturspeicher des Renderers.

7.2 Umsetzung der Nutzerschnittstelle

Die Nutzerschnittstelle des Programms besteht aus einer Anzahl Fenstern mit jeweils einer *OpenGL*-Instanz. Nach dem Start ist nur das Fenster mit der Übersicht vorhanden. Es zeigt zu diesem Zeitpunkt einen leeren Raum, denn es sind noch keine Schnitte angelegt. Für jeden neuen Schnitt wird ein weiteres Fenster mit einer Schnittansicht geöffnet. Diese Schnitte verhalten sich wie im Entwurf konzipiert. Wechselt der Nutzer zu einem Schnittfenster, kann er mit Hilfe der Maus die Lage des Schnittes verändern und der Cursor zeigt die in den Raum projizierte Position seiner Maus an. Im Übersichtsfenster wird gleichzeitig die Lage des Schnittes im Raum verdeutlicht und es lassen sich beliebig viele weitere Schnitte erzeugen.

Alle Schnittfenster, sowie das Übersichtsfenster teilen sich, wie im Entwurf vereinbart, sowohl *Renderer*interne Daten als auch die dazugehörigen programmseitigen Parameter. Um das System nicht unnötig zu belasten, wird jede Anzeige nur dann aktualisiert, wenn dies wirklich nötig ist. Es ist deshalb notwendig, dass ein Schnitt, der seine Lage ändert, dies selbstständig dem Übersichtsfenster mittels *Signal-Slot*-System mitteilt, damit dieses weiß, dass es neu zeichnen muss.

7.3 Darstellung der Volumendaten auf der Schnittfläche

Die Klasse für die eigentliche Schnittfläche `GLvlCutPlane` ist (über Zwischenschritte) von `SGLPolygon` aus der Basisklasse abgeleitet. Damit ist `GLvlCutPlane` ein schlichtes Viereck mit einer Instanz der Klasse `GLvlVolumeTex` als Textur. Diese Textur teilt sich die Schnittfläche mit Hilfe intelligenter Zeiger mit allen anderen Schnittflächen, da sie alle die selben Volumendaten darstellen sollen. Ihre Eckpunkte teilt sich die Schnittfläche mit der Kamera ihrer Schnittansicht. Jede Schnittansicht hat eine eigene Instanz der Klasse `GLvlPlaneCam`, einer Spezialisierung der Basisklasse `SGLBaseCam`. Diese Kamera bestimmt bei jeder Verlagerung alle vier Eckpunkte ihres Blickfeldes entsprechend den Beschreibungen aus Abschnitt 4.2.3 neu. Diese Eckpunkte teilt sie sich mit der Schnittfläche. Die Schnittfläche und die Projektionsfläche der Sicht sind damit identisch. Ändert sich ein Parameter der Kamera ändern sich auch die Eckpunkte der Schnittfläche. Lediglich die Mittelung über diese Änderung muss explitzit per `compileNextTime` an die Schnittfläche gesandt werden, damit sie beim nächsten Neuzeichnen neu kompiliert wird.

Die Lage der Schnittfläche im eigentlichen Raum ist damit bestimmt, ihre Lage im Datenraum fehlt aber noch. Auch hierfür müssen die Koordinaten der Eckpunkte bestimmt werden. Um deren Bestimmung möglichst einfach zu gestalten, benutzen der eigentliche Raum und der Datenraum ein identisches Koordinatensystem. Die Texturkoordinaten können daher fast direkt aus den Raumkoordinaten der Schnittfläche übernommen werden. Dabei ist jedoch zu beachten, dass die leeren Voxel an den Rändern der Textur im Renderer nicht gezeichnet werden. Da sie dennoch Teil des Texturraumes sind, ist der Datenraum eigentlich als eine Untermenge des Texturraumes zu betrachten. Die Texturkoordinaten der Schnittfläche werden beim Zeichnen auf den Texturraum und nicht auf den Datenraum angewendet. Sie müssen daher entsprechend angepasst werden, um

den Texturraum im Verhältnis zu der Schnittfläche so zu verschieben und zu skalieren, dass die Differenz zwischen Datenraum und Texturraum ausgeglichen wird.

7.4 Der Cursor

Der für den Cursor verwendete Objektklasse `GLvlPlaneCursor` ist von der Basisklasse `SGLMetaObj` abgeleitet. Sie vereint eine beliebige Anzahl von Instanzen der Basisklasse `SGLCube` in sich. Der Cursor besteht daher aus einer beliebigen Anzahl von Würfeln, die entsprechend Würfelförmig angeordnet werden. Die einzelnen Würfel haben eine Kantenlänge von 1 und werden als Drahtgittermodelle gezeichnet. Ihre Anzahl wird von Nutzer bestimmt, der auf diese Weise die Größe des Cursors anpassen kann. Dabei verdeutlichen die Einzelnen Würfel jederzeit den Maßstab. Die Position des Cursors im GL-Raum wird wie in Abschnitt 3.2.3 auf Seite 25 beschrieben, aus der Position des Mauszeigers ermittelt. Seine Koordinaten können dabei auf ganzzahlige Werte gerundet werden. Dieser "Fang" kann vom Nutzer aktiviert bzw. deaktiviert werden. Obwohl der Cursor den Mauszeiger prinzipiell ersetzt, wird dieser nicht ausgeblendet, denn vor allem bei aktiviertem Fang verwirrt ein fehlender Mauszeiger mehr als er nützt.

7.5 Integration der watershed-Transformation

Die in Kapitel 6 auf Seite 54 beschriebenen Implementationen der *watershed*-Transformation liefern Bilder $f_{wshed} : \mathbb{N}^3 \mapsto \mathbb{N}$ die die Gruppierung der Punkte nach ihrere zugehörigkeit zu lokalen Minima beschreiben. Für jeden Punkt p liefert $f_{wshed}(p)$ den Index des Minimas, dem er angehört. Die Visualisierung dieser Unterteilung wird in Kobination mit dem Cursor mittels Polygonen realisisiert. Jeder Punkt p, der einen Nachbar $q \in N_G(p)$ mit anderer Zugehörigkeit hat ($f_{wshed}(p) \neq f_{wshed}(q)$) legt dazu ein Polygon an der Grenze zwischen p und q an. Aus Effizienzgründen werden diese Polygone nicht als Instanzen der Basisklasse `SGLPolygon` angelegt. Stattdessen wird die Erkennung Grenzen und das anlegen der entsprechenden Polygone innerhalb der `generate()`-Routine der Objektklasse `GlvlMinima` ausgeführt, wobei das Objekt zwar kompiliert jedoch nicht gezeichnet wird. `GlvlMinima` wird zu diesem Zweck von der Basisklasse `SGLFlObj` abgeleitet. Im Zeichenpuffer des *Renderers* liegen somit durch ihre Grenzflächen, definierte *Minima-Körper* für jedes durch *watershed* erkannte Unterobjekt. Das für `GlvlMinima`-Objekte sehr aufwendige `generate()` muss lediglich einmal durchgeführt werden, da sich die durch sie repräsentierten *Minima-Körper* nicht verändern. (siehe "Caching von

Zeichenoperationen" in Abschnitt 4.2.2 auf Seite 36). Die Anwendungsseitigen Kontainer dieser Objekte (Instanzen der Klasse `GlvlMinima`) werden in einer Liste unter dem Index des lokalen Minimas mit dem entsprechenden Einzugsgebiet gespeichert. Bewegt sich der Cursor auf einen Punkt p im Texturraum, kann der *Minima-Körper* in dem sich p befindet direkt durch den Wert von $f_{wshed}(q)$ angesprochen werden. Ausschließlich dieser ausgewählte *Minima-Körper* wird gezeichnet. Die Auswahl und Darstellung eines *Minima-Körpers* für weitere Bearbeitung ist somit äußerst effizient, und kann daher problemlos interaktiv angewendet werden.

Die Aufgabe der Visualisierung nach der *watershed*-Transformation besteht somit darin, dem Nutzer die entsprechenden Unterobjekte anzuzeigen, und ihn beim "Einsammeln" dieser zu unterstüzen.

8 Zusammenfassung

8.1 Fazit der Arbeit

Im Rahmen dieser Arbeit wurde ein offenes und schlankes System zur Visualisierung dreidimensionaler Daten entworfen und entwickelt. Im Entwurf wurden einfache und effiziente Lösungen gesucht und gefunden. Dabei lag die Priorität primär auf der praktischen Anwendbarkeit im Wissenschaftlichen Umfeld, und sekundär auf der Flexibilität der Anwendung. Das System wurde von Grund auf Modular entworfen. Ein Basismodul, die *simpleGL*-Bibliothek, spielt in diesem Entwurf eine zentrale Rolle. Die *qt_glue*-Bibliothek entkoppelt als kleinstes Modul das Hauptmodul von der, für die Nutzerschnittstelle verwendeten *Qt*-Bibliothek. Die Nutzerschnittstelle greift als viertes Modul auf das Basismodul und auf das vierte Modul, die *watershed*-Transformation zurück. Sie bildet auf diese Weise die eigentliche Applikation. Weitere Module zur Erfüllung weiterer Aufgaben können diesem Verbund jederzeit hinzugefügt werden.

Der modulare Entwurf wurde konsequent objektorientiert umgesetzt. Während der Implementierung wurde großer Wert auf Wartbarkeit,Erweiterbarkeit und Wiederverwendbarkeit des Codes gelegt. Fehleranfällige Mittel der Programmierung, wie z.B. Zeiger wurden bewusst vermieden und durch moderne robustere Alternativen ersetzt. Lediglich während der Implementierung der *watershed*-Transformation wurden "traditionelle" Mittel bevorzugt, um die dort nötige Speichereffizienz zu erreichen.

8.2 Empfehlungen zur verwendeten Plattform

Das Visualisierungssystem wurde hauptsächlich unter Debian-Linux entwickelt. Seine Portabilität wurde durch die Verwendung portabler Bibliotheken und den weitgehenden Verzicht auf hardwarenahe Programmierung erhalten. Es sollte daher ggf. nach geringem Portierungsaufwand allen üblichen UNIX-Desktop-Systemen lauffähig sein. Auch eine

Portierung nach Windows wäre theoretisch möglich, scheitert aber unter Umständen an Lizenzkosten für *QT* und der *vista*-Bibliothek, die nicht für Windows verfügbar ist.

Eine Ausname der allgemeinen Verfügbarkeit auf üblichen Plattformen bilden in diesem Zusammenhang Rechner ohne kompatiblen *OpenGL*-Renderer. Das Visualisierungssystem kann zwar auf Softwarerenderern wie z.B. Mesa [14] betrieben werden, hier treten jedoch oft Probleme mit dem Grafikspeicher auf. Die Verlagerung der Volumendaten in den Speicher des Renderers entlasten auf Rechnern mit separatem Grafikspeicher die CPU und den Systemspeicher. Jedoch sind Systeme ohne echten Grafikspeicher (shared memory) aus diesem Grund nur eingeschränkt nutzbar, da hier der Grafikspeicher physikalisch im Systemspeicher liegt. Dieser ist für die Lagerung dreidimensionaler Texturen nicht geeignet. Ein Einsatz des entwickelten Visualisierungssystems auf Rechnern, bei denen die Grafikkarte lediglich als Chip auf dem *Mainboard* realisiert ist, ist daher zwar möglich, aber nicht empfehlenswert. Einfache 3D-Grafikkarten mit ausreichendem eigenen Speicher und *OpenGL*-kompatiblen Grafikprozessoren (GPUs) wie z.B. "ATI Radeon 9200", oder "nVidia GeForce2" sind sind dagegen vollkommen ausreichend. Besonders empfehlenswert sind GPUs nach Version 2.0 des *OpenGL*-Standards, denn dieser lässt einen effektiveren Umgang mit dem Texturspeicher zu. Der verfügbare Grafikspeicher sollte jedoch in jedem Fall groß genug sein, um die Volumendaten aufnehmen zu können. Für die am CBS üblichen MRT-Aufnahmen reicht ein Grafikspeicher von 32 MByte.

Verfügt das System wie empfohlen über separaten Grafikspeicher, reichen für die Visualisierung inklusive Segmentierung 128 MByte Systemspeicher. An die Festplatte werden keine besonderen Anforderungen gestellt. Ein großzügig bemessener Bildschirm ist jedoch von Vorteil.

Da *OpenGL* ein Client-Server-System ist, steht auch einer Anwendung auf einem Terminal-Server nichts im Wege, solange die Grafikkarten der Terminals die Anforderungen erfüllen. Dabei sollte jedoch beachtet werden, dass die Volumendaten beim Laden von der Anwendung (in diesem Fall auf dem Terminal-Server) zum Renderer (die Grafikkarte im Terminal) übertragen werden müssen. Daher wird eine ausreichend leistungsfähige Netzwerkverbindung benötigt.

8.3 Fortführende Überlegungen

Das im Rahmen dieser entwickelte eigentliche Visualisierungssystem (virgil) konnte dank der Auslagerung der meisten Funktionen in die entsprechenden Module einfach gehalten

werden. Daher bleibt der Quellcode sehr übersichtlich. Zukünftige Korrekturen, Anpassungen oder Erweiterungen lassen sich somit leicht realisieren. Die parallel entwickelte Basisbibliothek weist außer *OpenGL* keine weiteren Abhängigkeiten auf. Ihr Funktionsumfang deckt ein weites Spektrum von Visualisierungsaufgaben ab. Ihre Wiederverwendbarkeit auch außerhalb des aktuellen Rahmens ist daher ausgesprochen hoch.

Eine weitere Entwicklung sowohl des Basismoduls, als auch des Visualisierungssystems wird daher empfohlen.

Glossar

API	Application Programming Interface Die Schnittstelle, die ein Softwaresystem anderen Programmen zur Verfügung stellt.
ARB	Architecture Review Board Hier das Standardisierungsgremium des *OpenGL*-Standards. (siehe auch Kapitel 2.3 auf Seite 12)
BOLD	Blood Oxygen Level Dependency Von Ogawa [17] entwickeltes kernspintomografisches Verfahren, das die 1935 von Linus Pauling und Charles D. Coryell entdeckte paramagnetische Eigenschaft von Hämoglobin ausnutzt, um die Sauerstoffanreicherung im Blut zum messen. (siehe auch Abschnitt 2.1 auf Seite 10)
CBS	Max-Planck-Institut für Kognitions- und Neurowissenschaften (engl.: Max Planck Institute for Human Cognitive and Brain Sciences)
Direct3D	3D-Modul von DirectX *OpenGL* nachempfundene, aber objektorientierte Schnittstelle für die Programmierung dreidimensionaler Systeme. Stark an MS-Windows gebunden, und daher nur dort verfügbar.
dual-core	Prozessoren mit zwei Rechenwerken Dual-Core-Prozessoren werden von vielen Herstellern als Weg gesehen, die Leistung von Desktopsystemen ohne Erhöhung der Taktrate zu steigern. Sowohl AMD, als auch Intel haben Dual-Core-Prozessoren angekündigt oder bereits im Angebot.

DVR — direct volume rendering
Renderingmethode bei der Volumendaten direkt mittels *Raytracing* abgebildet werden. (siehe auch Abschnitt 3.1.1 auf Seite 16)

fMRI — funktionelle Magnetresonanztomografie oder Kernspintomografie
kernspintomografisches Verfahren das unter Verwendung des BOLD-Verfahrens über die Sauerstoffanreicherung Rückschlüsse auf die Aktivität des entsprechenden Gebietes zieht.
(siehe auch Abschnitt 2.1 auf Seite 10)

GL-Raum — virtueller, durch eine *OpenGL*-Sicht dargestellter Raum, in den zu zeichnende Objekte platziert werden. Der GL-Raum ist theoretisch unendlich, es wird jedoch nur ein, durch die Sicht definierter Teil, gezeichnet.

GPU — Graphic Processing Unit oder Visual Processing Unit (VPU)
Prozessor, der auf graphische Operationen wie z.B. *OpenGL*-Operationen spezialisiert ist. (siehe auch Abschnitt 2.3 auf Seite 12)

GUI — "graphical user interface" bzw. "Grafische Benutzungsschnittstelle"
System das grafisch mit dem Nutzer über Fenster, Schaltflächen, Dialogboxen u.Ä. interagiert. Wird allgemein als intuitiver als die Interaktion mittels Kommandos angesehen.

Kontext — Hier meist *OpenGL*-Kontext.
Identifiziert eine Instanz des *OpenGL*-Renderers. Der Kontext ist fest mit reservierten Ressourcen des Systems, sowie dem Fenster in das der Renderer zeichnet, verbunden.

mesa — auch Mesa3D
Vor allem im UNIX-Bereich verbreiteter *Softwarerenderer* mit modularer Architektur. (siehe auch Abschnitt 2.3.2 auf Seite 13 und Quellenverweis [14])

MRI — Magnetresonanztomografie, auch Kernspintomografie oder Kernspin
Ein bildgebendes Verfahren zur Darstellung von Strukturen im Inneren des Körpers, das auf dem magnetischen Moment von Atomkernen beruht. (siehe auch Abschnitt 2.1 auf Seite 10)

MRT — Magnetresonanztomografie
(siehe auch MRI)

OpenGL — Open Graphics Library
Ursprünglich von **?**] als IrisGL entwickelte Softwareschnittstelle für Vektorgrafik. Inzwischen als OpenGL standardisiert.
(siehe auch Kapitel 2.3 auf Seite 12)

overhead — Daten, die nicht primär zu den Nutzdaten zählen, sondern als Hilfsdaten zur Übermittlung oder Speicherung benötigt werden.

Qt — "The Qt application development framework"
Eine umfangreiche und sehr mächtige Systembibliothek mit Widgetsystem, die für zahlreiche Plattformen verfügbar ist. Die API ist plattformunabhängig.

rastern — Abbilden beliebiger Bilddaten in diskreten Puffern
Bezeichnet eine Operation, die meist stetige zweidimensionale Bilder in zweidimensionale Pixelpuffer (z.B. Bildpuffer einer Grafikkarte) abbildet, in diesem Fall ist $f : \mathbb{R}^2 \mapsto \mathbb{N}^2$.
(siehe auch Abschnitt 2.3.3 auf Seite 15)

Raytracing — auch Strahlverfolgung
Ein Rendering-Verfahren zur Erzeugung meist fotorealistisch wirkender Bilder. Für jedes Pixel der Zielbildes wird mindestens ein Strahl in die Szene gesandt. Die "Bedingungen", die der Strahl dort "vorfindet" bestimmen die Farbe des Zielpixels.

Renderer — vom englischen "to render" d.h. "machen"
Hier ist damit die Generierung von Pixelbildern aus abstrakten Eingabedaten gemeint. Als *Renderer* werden Systeme bezeichnet, diese Funktion ausführen.

SF — "surface fitting" bzw. Oberflächenrendering
Renderingmethode bei der Volumendaten indirekt mittels Polygonisierung dargestellt werden. (siehe auch Abschnitt 3.1.1 auf Seite 16)

Texturraum	virtueller Raum innerhalb des GL-Raumes, der durch eine dreidimensionale Textur geformt wird. Hier auch vereinfachend als Datenraum bezeichnet.
vista	Die Vista Bibliothek Eine Funktionsbibliothek für die wissenschaftliche Arbeit mit graphischen Daten im weitesten Sinne. Im CBS vor allem für den Datenaustausch verwendet. (siehe auch Abschnitt 7.1 auf Seite 75 und Quellenverweis [21])
voxel	Kurzform für Volumenpixel Dreidimensionale Version des Pixels. Ein Voxel repräsentiert ein Datum in einem Volumendatensatz. Voxel sind hier nicht gezwungenermaßen kubisch.
Widget	Kurzform von “Window Gadget” Frei übersetzt: “Dingsbums für Fenster”. Bezeichnet in der Programmierung allgemein Funktionen oder Objekte, die zum Darstellen in Grafischen Nutzerschnittstellen verwendet werden.

Literaturverzeichnis

[1] ATI. Ati technologies inc. URL `http://www.ati.com`.

[2] S. Beucher und C. Lantuéjoul. Sur l'utilisation de la ligne de partage des eaux en détection de contours. Technical Report N-598, Ecole des Mines de Paris, May 1979.

[3] boost::shared_ptr. shared_ptr. URL `http://www.boost.org/libs/smart_ptr/shared_ptr.htm`.

[4] boost::signals. signals. URL `http://www.boost.org/doc/html/signals.html`.

[5] Gregory Colvin. Exception safe smart pointers. *C++ committee document 94-168/N0555*, jul 1994.

[6] H. Digabel und C. Lantuéjoul. Iterative algorithms. In J.-L. Chermant, editor, *Quantitative analysis of microstructures in materials sciences, biology and medicine*, pages 85–99, Stuttgart, 1978. Dr. Riederer-Verlag GmbH.

[7] John R. Ellis und David L. Detlefs. Safe, efficient garbage collection for c++, usenix proceedings. *C++ committee document 94-168/N0555*, feb 1994.

[8] Joseph P. Hornak. The basics of mri. URL `http://www.cis.rit.edu/htbooks/mri/inside.htm`.

[9] C. Lantuéjoul. *La squelettisation et son application aux mesures topologiques des mosaïques polycristallines*. PhD thesis, Ecole des Mines de Paris, 1978.

[10] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990. ISSN 0730-0301. URL `http://doi.acm.org/10.1145/78964.78965`.

[11] William E. Lorensen und Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987. ISSN 0097-8930. URL http://doi.acm.org/10.1145/37401.37422.

[12] Chris Frazier Mark Segal, Kurt Akeley und Jon Leech. *The OpenGL Graphics System: A Specifikation*. The OpenGL Architecture Review Board, 1.5 edition. URL http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf.

[13] Arnold Meijster und Jos B. T. M. Roerdink. A disjoint set algorithm for the watershed transform, August 25 1998. URL http://citeseer.ist.psu.edu/114309.html;http://www.cs.rug.nl/~roe/psfiles/eusipco.ps.gz.

[14] Mesa. The mesa 3-d graphics library. URL http://www.mesa3d.org.

[15] MIT-Lizenz. The mit license. URL http://www.opensource.org/licenses/mit-license.php.

[16] nVidia. Nvidia corporation. URL http://www.nvidia.com.

[17] Seiji Ogawa. Oxygenation sensitive contrast in magnetic resonance image of rodent brain at high magnetic fields. *Magnetic Resonance Medcine*, 14, 1990.

[18] Jos B.T.M. Roerdink und Arnold Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundamenta Informaticae*, 0(41), 2001. URL http://www.cs.rug.nl/~roe/publications/parwshed.pdf.

[19] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, und Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 231–238. Eurographics Association, 2003. ISBN 1-58113-698-6.

[] SGI. Silicon graphics inc. URL http://www.sgi.com.

[20] Luc Vincent und Pierre Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(6): 583–598, 1991. ISSN 0162-8828. URL http://csdl.computer.org/comp/trans/tp/1991/06/i0583abs.htm.

[21] vista. vista software environment for computer vision research. URL http://sourceforge.net/project/showfiles.php?group_id=10180&package_id=10477.

[22] Wikipedia. Magnetresonanztomographie. URL http://de.wikipedia.org/wiki/Magnetresonanztomografie.