Generische Programmierung für die Bildverarbeitung

Ullrich Köthe

Hamburg, 2000

Ullrich Köthe
Email: koethe@informatik.uni-hamburg.de

Universität Hamburg, Fachbereich Informatik
Arbeitsbereich Kognitive Systeme
Vogt-Kölln-Str. 30
22527 Hamburg

Dissertation zur Erlangung des Doktorgrades
am Fachbereich Informatik der Universität Hamburg

Gutachter:	Prof. Dr. H. Siegfried Stiehl, Hamburg
	Prof. Dr. Bernd Neumann, Hamburg
	Prof. Dr. Stefan Jähnichen, Berlin
Tag der Disputation:	29. Februar 2000

Herstellung: Libri Books on Demand

ISBN 3-8311-0239-2

Kurzfassung

Nach wie vor ist unklar, wie man im Computer Vision-Kontext flexible Software entwickeln kann. Dies ist bedauerlich, weil gerade dieses Fachgebiet hiervon besonders profitieren würde. Flexible Softwarerealisierungen würden es wesentlich erleichtern, komplexe Algorithmen immer wieder im Detail an neue Aufgaben anzupassen, neue Erkenntnisse schnellstmöglich in die Praxis zu überführen und die entwickelten Komponenten in einen größeren Systemzusammenhang zu integrieren, um nur einige Gründe zu nennen.

Neue Methoden der generischen Programmierung – insbesondere abstrakte Iteratoren, Zugriffsobjekte und Funktoren – eröffnen vielversprechende Möglichkeiten zur Verbesserung der Flexibilität. Mit Hilfe dieser Methoden gelingt es uns in der vorliegenden Arbeit erstmalig, ein umfassendes System von generischen Schnittstellen für Computer Vision zu definieren. Dies versetzt uns in die Lage, unabhängige Bausteine für verbreitete Bildverarbeitungs- und Bildanalyseverfahren zu implementieren, die flexibel miteinander kombiniert und leicht an die verschiedensten Aufgaben angepaßt werden können. Im Rahmen der vom Autor entwickelten Computer Vision-Bibliothek VIGRA wurden alle diese Konzepte erfolgreich umgesetzt und überprüft.

Unter den neu entwickelten Bausteinen kommt denjenigen zur Realisierung von zellulären Komplexen eine besondere Bedeutung zu. Diese Bausteine erlauben es erstmals, eine universelle Repräsentation für Segmentierungsergebnisse zu schaffen. An mehreren Beispielen demonstrieren wir, wie zelluläre Komplexe verwendet werden können, um Segmentierungsalgorithmen zu vereinheitlichen, zu verbessern und zu kombinieren. Hier zeigt sich auf besonders überzeugende Weise das große Potential der generischen Programmierung für Computer Vision.

Danksagung

Diese Arbeit wäre nicht möglich gewesen ohne die Unterstützung vieler Menschen, denen ich dafür zu großem Dank verpflichtet bin. An erster Stelle muß ich hier Prof. H. Siegfried Stiehl nennen, dessen Angebot zur externen Betreuung dieser Dissertation mir ebenso überraschend wie willkommen war. Seine tiefgehenden Kommentare haben wesentlich zum Gelingen der Arbeit beigetragen.

Zum größten Teil ist diese Dissertation am Fraunhofer-Institut für Graphische Datenverarbeitung, Institutsteil Rostock entstanden. Meinen dortigen Abteilungsleitern, Prof. Bodo Urban und Dr. Erhard Berndt, danke ich für die großen Freiräume, die das gründliche Nachdenken über die dargestellten Probleme erst ermöglicht haben. Dr. Wolfgang Luth danke ich dafür, daß er mich mit dem äußerst interessanten Forschungsgebiet Computer Vision bekannt gemacht und dafür begeistert hat.

Vielen Kollegen habe ich für ausgiebige Diskussionen zu den verschiedensten Fragen zu danken: Holger Diener, Klaus Hartenstein, Steffen Nowacki, Uwe von Lukas, Andreas Schlempp und Dr. Karsten Weihe. Diese Diskussionen haben mir zahlreiche Anregungen gegeben, die mein Verständnis der behandelten Probleme wesentlich vorangebracht haben.

Am Schluß und vor allem möchte ich meiner Familie danken für die große Geduld und Unterstützung – nicht zuletzt beim Korrekturlesen der Arbeit – während der Zeit des nächtelangen Schreibens. Ohne diesen sicheren Rückhalt hätte ich womöglich mehrmals vorzeitig das Handtuch werfen müssen.

Inhalt

Verzeichnis der Abbildungen und Tabellen

Kapitel 1

Einleitung

1.1 Hintergrund und Motivation

In den letzten Jahren haben Standardworkstations ein Leistungsniveau erreicht, das sie zunehmend für Computer Vision-Anwendungen interessant werden ließ.[1] Immer häufiger geht man von traditionellen Lösungen auf der Basis von spezieller Hardware zu reinen Softwarelösungen über. Softwarelösungen haben den entscheidenden Vorteil, daß sie sich relativ leicht an veränderte Anforderungen anpassen lassen. In der Tat sagt schon die Bezeichnung *Soft*ware, daß Flexibilität zu den entscheidenden Charakteristika von Computerprogrammen gehört. Flexibilität kann in der Praxis aus verschiedenen Gründen gefordert werden:

Inkrementelle Entwicklung: Ein größeres System entsteht schrittweise. Wenn man neue Funktionalität hinzufügt, ist es oft erforderlich, auch Teile des vorhandenen Systems zu modifizieren und anzupassen. Insbesondere müssen Datenstrukturen erweitert werden, um neue Informationen aufzunehmen.

Laufende Verbesserung: Änderungen aufgrund von Programmierfehlern oder wegen neuer Lösungsideen müssen laufend oder periodisch in das System

[1] Zur Terminologie: Der Begriff Computer Vision dient in dieser Arbeit als Oberbegriff für Systeme und Verfahren, die aus Bildern Informationen über die dargestellten Objekte extrahieren. Dies umfaßt sowohl die Bildverarbeitung (die Transformation von Bildern in andere Bilder) als auch die Bildanalyse (die Gewinnung anderer Repräsentationsformen aus Bildern).

eingearbeitet werden. Dies ist gerade bei Computer Vision-Anwendungen sehr wichtig, wo ständig neue Verfahren entwickelt und veröffentlicht werden.

Wechselnde Anforderungen: Häufig ändern sich im Laufe der Entwicklung eines Systems (oder nach dessen Fertigstellung) die Anforderungen an seine Funktionalität. Häufig liegt dies daran, daß die Anforderungen erst dann endgültig formuliert werden können, wenn ein Teil des Systems bereits in der Praxis eingesetzt oder zumindest erprobt wurde.

Veränderte Umgebung: Ein Softwaresystem steht nicht allein, sondern hängt von verschiedenen anderen Hard- und Softwarekomponenten ab. Diese Komponenten können sich unabhängig voneinander ändern, so daß das zu entwickelnde Softwaresystem gegebenenfalls an diese Änderungen angepaßt werden muß.

Wiederverwendung: Aus verschiedenen Gründen ist es wünschenswert, einzelne Softwarekomponenten in anderen Projekten wiederzuverwenden. Dies spart Entwicklungskosten, weil Einarbeitungs- und Testaufwand eingespart werden können, und führt zu Lösungen von höherer Qualität. Entscheidend ist auch hier, daß die vorhandene (wiederverwendbare) Funktionalität möglichst einfach an die Anforderungen des neuen Projektes angepaßt werden kann.

Es kommt also darauf an, Software von vornherein so zu gestalten, daß sie Änderungen unterstützt und sich diesen nicht in den Weg stellt. Leider passiert dies nicht von selbst, wie das bekannte Sprichwort "Never change a running program!" drastisch belegt. Das Hauptproblem besteht darin, daß eine Änderung häufig Folgeänderungen erzwingt, so daß sich Änderungen im Extremfall lawinenartig über das ganze System ausbreiten. Dies kann einen so großen Aufwand verursachen, daß man von der ursprünglichen Änderung lieber Abstand nimmt. Außerdem ist es sehr schwierig, die Korrektheit des geänderten Programms zu garantieren: oft wird eine notwendige Folgeänderung übersehen, und unerwartete Fehler treten auf. Je komplizierter die Software wird, desto stärker treten diese Probleme in den Vordergrund. Nur mit gezielten Gegenmaßnahmen im Sinne eines Design for Change kann auch bei komplizierten Softwaresystemen die gewünschte Flexibilität gesichert und die Komplexität der gegenseitigen Abhängigkeiten reduziert und beherrscht werden.

Software ist nur dann flexibel, wenn Änderungen mit angemessenem Aufwand möglich bleiben. Dies bedeutet vor allem, daß jede Änderung auf einen kleinen Bereich des Systems beschränkt bleiben muß. Folgeänderungen müssen, wenn irgend möglich, vermieden werden. Mit anderen Worten, verschiedene Aspekte der Funktionalität müssen in jeweils eigenständigen *Bausteinen* konzentriert werden, die unabhängig voneinander modifizierbar sind. Die Abhängigkeiten zwischen den Bausteinen müssen über *Schnittstellen* (Interfaces) sorgfältig beschrieben und insge-

samt gezielt minimiert werden. Im Idealfall besteht ein komplexes System aus vielen unabhängigen Bausteinen, die entsprechend den Anforderungen der jeweiligen Applikation maßgeschneidert zusammengesetzt werden.

Die Softwaretechnik befaßt sich seit langem intensiv mit den methodischen Voraussetzungen hoher Flexibilität, angefangen von Wirths Prinzip des top-down-Entwurfs durch schrittweise Verfeinerung [WIRTH71] und Parnas' Geheimnisprinzip [PARNAS72], über die Bemühungen der strukturierten [DAHL+72] und objekt-orientierten Programmierung [BOOCH94, MEYER97] bis hin zu den neuen Ideen der komponentenbasierten Softwareentwicklung [SZYPERSKI97] und der generischen Programmierung [MUSSTEP94, AUSTERN98], um nur einige Ansätze zu nennen. Trotz ihrer Verschiedenheit gehen alle diese Methoden von drei Grundproblemen aus, die bei der Entwicklung flexibler Software gelöst werden müssen:

- die geeignete Zerlegung der Aufgabenstellung in Teilaufgaben,
- die abstrakte Beschreibung der Beziehungen zwischen den Bausteinen, die diese Aufgaben lösen, sowie
- die Entwicklung von leistungsfähigen technischen Mitteln, um einen abstrakten Entwurf tatsächlich flexibel implementieren zu können.

Gleichzeitig dürfen Performanz und einfache Verwendbarkeit der Bausteine nicht dem Streben nach Flexibilität zum Opfer fallen.

Selbstverständlich hängt die angemessene Lösung dieser Probleme sehr stark von der jeweiligen Anwendung bzw. den jeweiligen Klassen von Anwendungen ab. Im Anwendungsbereich Computer Vision finden wir eine Reihe besonderer Anforderungen:

- Computer Vision ist in stärkerem Maße als viele andere Anwendungsgebiete von *Algorithmen* geprägt. Der weitaus größte Teil der einschlägigen Veröffentlichungen beschreibt Algorithmen bzw. die dahinter stehenden Berechnungstheorien. Eine Entwurfsmethode für Computer Vision muß also neben leistungsfähigen Datenabstraktionen auch und vor allem algorithmische Abstraktionen unterstützen.
- Computer Vision ist selten Selbstzweck, sondern fast immer Bestandteil eines umfassenderen Systems, wie z.B. eines medizinischen Diagnose- oder Therapieunterstützungsgeräts, einer industriellen Produktionsanlage oder eines autonomen Roboters. Deshalb muß sich der Entwurf der Computer Vision-Komponente den Erfordernissen und der Architektur des Gesamtsystems unterordnen, und nicht umgekehrt.
- Computer-Vision-Bausteine haben in der Regel sehr große Datenmengen zu verarbeiten. Ein kontinuierlicher Videodatenstrom generiert z.B. etwa 40 MBytes pro Sekunde, Luft- und Satellitenbilder umfassen meist um die 250

MBytes, und 3-dimensionale Computertomogramme können bereits mit mehreren Gigabytes zu Buche schlagen. Deshalb ist es wichtig, die Flexibilität nicht zu Lasten der erforderlichen Verarbeitungsgeschwindigkeit zu erhöhen.

- Durch die aktive Forschung auf allen Teilgebieten von Computer Vision werden die Verfahren ständig weiterentwickelt. Solche Weiterentwicklungen sollten mit geringem Aufwand in die jeweilige Umgebung integrierbar sein. Dies wäre gerade hinsichtlich eines schnellen Austauschs sowie einer sorgfältigen Validierung und Evaluierung neuer Verfahren wünschenswert [VIERGEVER+99].

Es hat selbstverständlich in der Vergangenheit nicht an Versuchen gefehlt, flexible Computer Vision-Komponenten zu entwickeln. Zunächst wurden diese Komponenten auf dem Prinzip der *prozeduralen Programmierung* aufgebaut, das von verbreiteten Programmiersprachen wie C und Pascal unterstützt wird.

Die grundlegenden Bausteine der prozeduralen Programmierung sind die *Unterprogramme* (Prozeduren und Funktionen). Ein Unterprogramm kapselt einen Algorithmus oder einen Teil davon und ermöglicht es dadurch, diese Funktionalität an beliebiger Stelle aufzurufen. Dies führt natürlicherweise zu einer algorithmischen Zerlegung eines Problems, und komplexere Algorithmen werden hierarchisch aus einfacheren Unterprogrammen aufgebaut. Dieses Vorgehen hat sich in der Computer Vision sehr gut bewährt und wird immer noch sehr breit eingesetzt, wie Systeme wie Candela [CANDELA98], SCILImage [KOELSMEUL95] und Tina [POLLARD+97] zeigen. Auch der ISO-Standard PIKS (Programmer's Imaging Kernel System, [PIKS94, PRATT95]), der sich allerdings nicht durchsetzen konnte, verwendet dieses Paradigma.

Der Datenaustausch zwischen den Unterprogrammen eines prozeduralen Systems erfolgt mit Hilfe von (möglicherweise verschachtelten) Datenstrukturen, die jedoch rein passive Datenbehälter sind. Dies hat sich als Nachteil erwiesen, weil dadurch einerseits die Entwicklung von sehr komplexen Datenstrukturen, wie Graphen und Netzwerken, erschwert wird, und andererseits durch fehlende Kapselung die Unterprogramme vollkommen von den konkreten Datenstrukturen abhängen. Das hat zur Folge, daß nach jeder Änderung einer Datenstruktur auch sämtliche Unterprogramme, die diese Datenstruktur benutzen, angepaßt werden müssen.

Man versucht dem durch Standardisierung entgegenzuwirken, zum Beispiel mit standardisierten Datenformaten wie dem MPEG-Standard [HASKELL+96] und den Bildformaten des DICOM-Standards [DICOM93]. Standardisierung ist aber nur dort möglich, wo die Anforderungen gut verstanden sind und sich nicht ändern. Dies gilt in der Computer Vision nur selten. Die Abhängigkeit der Algorithmen von den Datenstrukturen und generell die Abhängigkeit allgemeiner Verfahren von den Details der Implementation hat sich daher hier, wie auch in anderen Anwendungs-

gebieten, als ernstes Problem der prozeduralen Programmierung erwiesen (vergleiche z.B. [MARTIN95]).

Man hat sich deshalb schon seit langem um leistungsfähigere Abstraktionsmechanismen bemüht. Im Computer Vision-Kontext führte die Weiterentwicklung der bisherigen Ideen zu Systemen, bei denen Algorithmen und Datenstrukturen als unabhängige *Komponenten* gekapselt werden. Beispiele für diesen Ansatz sind die Systeme Khoros [RASKUB94] und Halcon/Horus [HORUS97]. Mit Hilfe einer (meist textuellen, bei Khoros visuellen) Skript- oder Makrosprache können die Komponenten sehr flexibel miteinander kombiniert werden. Inkompatibilitäten zwischen verschiedenen Komponenten können durch Zwischenschalten von Konvertierungskomponenten behoben werden. Um eine hohe Flexibilität zu erreichen, muß man allerdings die Funktionalität feinkörnig auf sehr viele Komponenten verteilen. Dies führt einerseits dazu, daß zur Verwendung eines komponentenbasierten Systems erhebliche Computer-Vision-Kenntnisse nötig sind. Andererseits entsteht wegen des Aufwands beim Aufruf der Komponenten, beim Datentransfer und bei der eventuell notwendigen Konvertierung eine erhebliche Geschwindigkeitseinbuße. Deshalb werden diese Systeme hauptsächlich für die Prototypentwicklung und in wissenschaftlichen Anwendungen verwendet.

Ein vollkommen anderer Weg zur Verbesserung der Flexibilität wird von der *objekt-orientierten Programmierung* eingeschlagen. Sie legt den Schwerpunkt auf die Datenstrukturen, die zu aktiven, gekapselten Objekten weiterentwickelt werden. Bei der strengen Objektorientierung können Algorithmen nur noch als Methoden bzw. member functions der Objekte auftreten, sie werden gleichsam zu „Diensten", die die Datenstrukturen anbieten. Bei hybriden Programmiersprachen, wie C++, können allerdings nach wie vor auch Subroutinen definiert werden. Beispiele für objektorientierte Computer Vision-Systeme sind TargetJr [TARGETJR96] und das Image Understanding Environment (IUE) [IUE98].

Durch die objekt-orientierte Programmierung erhält man hervorragende Möglichkeiten, komplexe Datenstrukturen, wie sie für Computer Vision typisch sind, zu modellieren und in Vererbungshierarchien zu organisieren. Abstrakte Schnittstellenklassen gestatten es in Verbindung mit virtuellen Funktionen und abgeleiteten Implementationsklassen, Datenstrukturen in gewissen Grenzen austauschbar, also polymorph, zu gestalten. Wie wir in Kapitel 2 zeigen werden, hat dieser Mechanismus jedoch einige Schwächen, die ihn für viele Computer Vision-Probleme weniger attraktiv machen. Darüber hinaus zeigt sich, daß die Betonung der Datenstrukturen (Objekte) gegenüber den Algorithmen bei der Problemzerlegung im Bereich Computer Vision nicht die natürlichste Herangehensweise ist.

Insgesamt kann man das Problem der Entwicklung von flexibler Computer Vision-Software nach wie vor nicht als gelöst betrachten. Immer noch ist viel zuviel Handarbeit nötig, um neue Bausteine aus vorhandenen zusammenzusetzen, sie den Anforderungen in einem neuen Kontext anzupassen und alles in ein einheitliches

Anwendersystem zu integrieren. A. Smeulders (in [VIERGEVER+99]) schätzt, daß über 80 Prozent der Ressourcen bei der Entwicklung von Computer Vision-Anwendungen durch Programmierung in Anspruch genommen werden, anstatt für die Entwicklung und Evaluierung neuer Algorithmen und Konzepte zur Verfügung zu stehen – ein zweifellos unbefriedigender Zustand.

Wir wollen im Verlauf dieser Arbeit Gründe hierfür erörtern und Lösungsmöglichkeiten erarbeiten. Dabei wird mit der *generischen Programmierung* eine neue Entwurfsmethode im Mittelpunkt stehen, die in den letzten Jahren durch A. Stepanov und D. Musser entwickelt wurde [MUSSTEP89, MUSSTEP94] und die inzwischen Grundlage der C++-Standardbibliothek geworden ist. Generische Programmierung widmet ihre Aufmerksamkeit bei der Problemzerlegung gleichermaßen den Algorithmen und den Datenstrukturen. Wichtigstes Ziel ist dabei die Vermeidung aller unnötigen Kopplungen zwischen den Bausteinen: hierarchisch aufgebaute Datenstrukturen und Algorithmen sollen weitestgehend frei aus einfacheren Bestandteilen kombinierbar sein, und Algorithmen sollen unabhängig von konkreten Datenstrukturen implementiert werden.

Die generische Programmierung analysiert Algorithmen und Datenstrukturen zunächst unabhängig voneinander. Im Zentrum dieser Methode stehen Fragen wie: Welche minimalen Anforderungen müssen Algorithmen an Datenstrukturen stellen, um ihre Aufgabe effizient erfüllen zu können? Welche Operationen können von verschiedenen Datenstrukturen effizient angeboten und implementiert werden? Auf dieser Grundlage wird eine konzeptuelle Taxonomie der Anforderungen entwickelt, die in abstrakten Kategorien standardisiert werden. Bei der Umsetzung in eine Programmiersprache kommt es nun darauf an, daß die Bausteine nur auf der abstrakten Ebene von anderen Bausteinen abhängen. Die Anpassung eines Bausteins an eine konkrete Implementation dieser „Zulieferer" muß weitgehend automatisch, ohne Änderungen des Quellcodes, erfolgen.

Natürlich muß die Programmiersprache die notwendigen Mittel zur Umsetzung dieser Technik bereitstellen. Es ist häufig schwierig, Bausteine zu entwickeln, die einerseits die Konzepte des Anwendungsgebiets elegant umsetzen und gleichzeitig in der gegebenen Programmiersprache effizient implementiert werden können. Zur Zeit besitzt C++ mit dem template-Mechanismus das im Hinblick auf Abstraktionsfähigkeit und Performanz leistungsfähigste Instrumentarium zur Unterstützung der generischen Programmierung [STEPANOV95], ihre Umsetzung in anderen Sprachen wie Ada oder Lisp ist jedoch ebenfalls möglich. Die vorliegende Dissertation zeigt, wie die neuen Möglichkeiten der generischen Programmierung für den Entwurf flexibler Computer Vision-Bausteine nutzbar gemacht werden können.

1.2 Zielstellung und Aufbau der Arbeit

Die vorliegende Dissertation beschreibt ein System generischer Konzepte für Computer Vision-Anwendungen. Ziel dieser Bemühungen ist eine wesentliche Erhöhung der Flexibilität von Computer Vision-Software sowie die Überwindung einer Reihe von Schwierigkeiten mit herkömmlichen Ansätzen. Dabei handelt sich es sowohl um softwaretechnische Probleme, vor allem die Abhängigkeit der Algorithmen von den unterliegenden Datenstrukturen, als auch um Probleme der Computer Vision, wie beispielsweise die Frage nach einer einheitlichen Repräsentation von Segmentierungsergebnissen. Zur Entwicklung der generischen Konzepte für diesen Anwendungsbereich werden wir uns mit so grundsätzlichen Fragen wie „Was ist ein Bild?“, „Welche Eigenschaften hat ein Pixel?“ oder „Wie definieren wir eine Segmentierung?“ auseinandersetzen müssen. In die Beantwortung dieser Fragen werden Ergebnisse aus der Mathematik, der Softwaretechnik und der Computer Vision einfließen.

Schwerpunkt der Arbeit ist die erstmalige Definition eines umfassenden und konsistenten *Systems von generischen Schnittstellenkonzepten* für Computer Vision. Die Leistungsfähigkeit dieses Systems wird durch zahlreiche Einzellösungen demonstriert und in vielen neuartigen Bausteinen verwirklicht. Dank der einheitlichen Behandlung aller aufgeworfenen Fragen sind diese Bausteine in weiten Grenzen frei kombinierbar. Dabei reicht die Bandbreite der Lösungen von der Definition grundlegender Pixeltypen (wie z.B. RGB-Werten) und Bilddatentypen über die Implementation wichtiger Bildverarbeitungsalgorithmen bis hin zur Beschreibung von Graphdatenstrukturen und Segmentierungsverfahren. Insbesondere gelingt es uns erstmalig, eine universelle Repräsentation für Segmentierungsergebnisse zu verwirklichen.

Wir werden dabei nicht bei theoretischen Überlegungen zu diesen Problemen stehenbleiben, sondern sämtliche Konzepte durch konkreten Quellcode demonstrieren. Der Quellcode ist im wesentlichen der generischen Computer Vision-Bibliothek VIGRA entnommen, die vom Autor dieser Arbeit entwickelt wurde und in der alle hier vorgestellten Konzepte verwirklicht sind. Im Anhang (Seite 251) geben wir einen Überblick über die Funktionalität dieser Bibliothek.[2] Dank der neuen Konzepte können mit Hilfe dieser Bibliothek viele Programmieraufgaben, die bisher einen großen Aufwand erforderten, durch Änderung nur weniger Quellcodezeilen realisiert werden. Wie verschiedene Benchmark-Tests zeigen, wird diese Flexibilität nicht durch einen großen Geschwindigkeitsverlust erkauft.

---

[2] Unter der Adresse `http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/` ist ein großer Teil der VIGRA-Funktionalität frei zugänglich.

Die vorliegende Arbeit gliedert sich wie folgt: In Kapitel 2 wollen wir wichtige Prinzipien für die Entwicklung flexibler Software herausarbeiten. Auf dieser Basis werden wir analysieren, warum existierende Computer Vision-Software nicht das wünschenswerte Maß an Flexibilität bietet. In Kapitel 3 werden wir die generische Programmierung als vielversprechende Alternative einführen und deren Grundlagen erläutern. Kapitel 4 ist der Entwicklung generischer Schnittstellenkonzepte für grundlegende Bildverarbeitungsbausteine und der Definition entsprechender Datenstrukturen (Bilder und Pixeltypen) und Algorithmen (Punktoperationen und Faltung) gewidmet. Kapitel 5 befaßt sich mit der Definition von Iterator-Adaptern, die die Verwendung unterschiedlicher Navigationsmuster auf ein und derselben Bilddatenstruktur erlauben, z.B. die Auswahl linearer Untermengen, den Zugriff auf die Nachbarn eines Punkts und die Verfolgung einer Objektkontur. In Kapitel 6 zeigen wir, daß sich auch traditionelle Segmentierungsverfahren, insbesondere Kantendetektion und Regionenwachstum, generisch implementieren lassen, wodurch ihre Flexibilität stark verbessert wird. Der zentrale Abschnitt der Arbeit ist Kapitel 7, in welchem wir auf der Grundlage des mathematischen Konzepts des *zellulären Komplexes* erstmalig eine universelle Repräsentation für Segmentierungsergebnisse sowie die damit verbundenen generischen Schnittstellen definieren werden. Schließlich demonstrieren wir in Kapitel 8 an mehreren Beispielen, wie diese Repräsentation implementiert werden kann und wie sich bekannte Algorithmen an die neue Repräsentation anpassen lassen.

Kapitel 2

Flexible Software für Computer Vision – eine Bestandsaufnahme

2.1 Grundlegende Prinzipien für den Entwurf von flexibler Software

Auch wenn die Werbung häufig das Gegenteil zu suggerieren versucht, führt keine der heute bekannten Entwurfsmethoden automatisch zu flexibler und wiederverwendbarer Software. Vielmehr muß der Designer einer Softwarekomponente gezielt Mechanismen einbauen, die eine hohe Flexibilität gewährleisten. Ohne das explizite "Design for Change" kann sehr leicht die Situation eintreten, daß eine Modifikation zu Folgeänderungen im ganzen System führt. Dadurch können selbst einfache Änderungen sehr aufwendig werden. Außerdem besteht die Gefahr, daß man nicht alle notwendigen Folgeänderungen erkennt, so daß unerwartete Fehler in Programmteilen auftreten, die man gar nicht geändert hat.

Viele Autoren (vergleiche die nachfolgende Diskussion) haben sich zu Entwurfsprinzipien geäußert, die die einfache Modifizierbarkeit von Software sichern helfen. Grundlage aller Methoden ist die geschickte Zerlegung des Systems in einzelne Bausteine[3]. Die Zerlegung verfolgt im allgemeinen drei wesentliche Ziele:

[3] Zur Terminologie: Wir verwenden den Begriff *Baustein* als allgemeinen Oberbegriff für die Bestandteile einer Systemzerlegung, weil dieser Begriff neutral bezüglich der verwendeten Entwicklungsmethode ist. Andere mögliche Begriffe (wie Modul, package, Subsystem, Objekt/Klasse)

Vereinfachung: Auf der konzeptuellen Ebene dient die Zerlegung dazu, ein komplexes Problem in leichter zu lösende Teilprobleme aufzuteilen.

Abgrenzung: Auf der Implementationsebene sollen die Teile eines Systems so voneinander abgegrenzt werden, daß man die Bausteine einzeln bearbeiten und testen kann.

Flexibilität: Auf beiden Ebenen verfolgt die Zerlegung das Ziel, daß veränderte Anforderungen an die Systemfunktionalität mit möglichst geringem Aufwand realisiert werden können.

Jeder Baustein stellt eine bestimmte Funktionalität bereit, die von anderen Teilen des Systems abgerufen werden kann. Aus der Sicht eines Bausteins ist die Welt also in ein "Inneres" und eine "Umgebung" geteilt, und dazwischen befindet sich eine sichtbare Grenze, die "Schnittstelle".

Eine gute Zerlegung ist einerseits dadurch gekennzeichnet, daß die Problemstellungen eines bestimmten Anwendungsbereichs in einfacher und effizienter Art und Weise modelliert werden. Andererseits muß die Zerlegung gewährleisten, daß kleine Änderungen der Problemstellung durch entsprechend kleine Änderungen der Bausteine realisierbar sind. Das heißt vor allem, daß kleine Änderungen auf möglichst wenige Bausteine beschränkt bleiben müssen und sich nicht in einer Art Kettenreaktion über das ganze System erstrecken dürfen. Hierzu sind Bausteine mit *hoher Kohäsion und schwacher Kopplung* erforderlich, bei denen das Innere jedes Bausteins wesentlich enger miteinander zusammenhängen sollte als die Bausteine untereinander.

2.1.1 Hohe Kohäsion

Hohe Kohäsion bedeutet zunächst, daß jeder Baustein *genau eine* wohldefinierte Funktionalität implementiert [STEVENS+74]. Allerdings ist "genau eine Funktionalität" kein vollkommen feststehender Begriff, denn eine solche Einschätzung hängt natürlich vom Betrachtungsmaßstab ab: eine Funktionalität, die man auf hoher Ebene als nur eine betrachtet, muß auf tieferer Ebene in Teilschritte zerlegt werden. Daraus ergibt sich sofort, daß wir auf unterschiedlichen Ebenen auch unterschiedliche Arten von Bausteinen benötigen, damit wir die Granularität der Zerlegung dem

implizieren meist eine bestimmte Methode, Technologie und/oder Granularität der Zerlegung. Auch der Begriff *Komponente*, der sich als Alternative anbietet, kann zu Mißverständnissen Anlaß geben, weil er neuerdings im Zusammenhang mit Component Technologies häufig in wesentlich spezieller Bedeutung benutzt wird (nämlich als Server-Baustein, der mit Hilfe von middleware-Technologien wie CORBA und COM/ActiveX angesprochen wird, vgl. z.B. [SZYPERSKI97]).

jeweiligen Umfang der Problemstellung anpassen können.[4] Auf der Ebene feiner Granularität benötigen wir kleine, effiziente Bausteine wie z.B. einzelne Prozeduren und Datenstrukturen. Auf höherer Ebene steht dagegen die Koordination komplexer Abläufe und die Flexibilität zur Laufzeit im Vordergrund, wofür sich beispielsweise objekt-orientierte frameworks und Software-Komponenten eignen.

Wir können hohe Kohäsion daran erkennen, daß auf der gegebenen Betrachtungsebene die Teile eines Bausteins (also z.B. die Methoden eines Objekts oder die Objekte eines Moduls) nur gemeinsam einen Sinn ergeben und deshalb normalerweise auch gemeinsam benutzt werden. Allerdings ist die Bewertung der Kohäsion aufgrund dieses Kriteriums nicht unproblematisch. Häufig wird beispielsweise bei einem Baustein zwischen read-only und read-write Funktionalität unterschieden, deren getrennte Verwendung gerade beabsichtigt ist. Das Problem wird noch deutlicher, wenn ein Baustein je nach Kontext mehrere Rollen spielen kann. Wir werden im Zusammenhang mit dem Problem der Subjektivität auf diese Fragen zurückkommen (siehe Seite 17).

Martin [MARTIN95] weist darauf hin, daß das Kriterium der common closure für die Beurteilung der Kohäsion noch wichtiger ist. Nach Martin hat ein Baustein hohe Kohäsion, wenn seine Teile den gleichen Arten von Änderungen unterworfen sind. Gegenüber den meisten Änderungen im System ist der Baustein als Ganzes immun. Wenn aber ein Teil des Bausteins geändert werden muß, müssen seine übrigen Teile ebenfalls geändert werden. Ungenügende Kohäsion besitzt ein Baustein hingegen dann, wenn eine Teilfunktionalität unabhängig vom Rest des Bausteins variiert – diese sollte besser in einen eigenen Baustein ausgelagert werden. Hohe Kohäsion ist demnach dadurch charakterisiert, daß bestimmte Arten von Änderungen in jeweils einem Baustein konzentriert werden.

Eine gute Regel für die Erkennung hoher Kohäsion stellt das Prinzip *„Einmal und nur einmal“* dar [BECK+98]. Es besagt, daß jede Funktionalität im System durch einen und nur einen Baustein realisiert werden sollte. Eine selbständige Funktionalität, die nicht in einem eigenen Baustein gekapselt ist, wird immer das Prinzip „eine Funktionalität pro Baustein“ verletzen. Schlimmer noch ist der Fall, wenn die ungekapselte Funktionalität in vielen Bausteinen wiederkehrt: dann erfordert jede Änderung dieser Funktionalität die Modifikation vieler verschiedener Bausteine – dies ist der Grund dafür, daß das vermeintlich simple „Jahr 2000“-Problem so riesige Kosten verursacht hat.

---

[4] Zwar beanspruchen manche Vertreter einer Methode, daß ihre Bausteine für alle Ebenen optimal sind. Dies äußert sich in solchen Behauptungen wie "everything is an object" oder neuerdings "object-orientation has failed, but component software is succeeding" [UDELL94]. Die großen Unterschiede der verschiedenen Anwendungsbereiche und Granularitätsebenen lassen dies jedoch als wenig plausibel erscheinen.

Der Vorteil des Prinzips *„Einmal und nur einmal“* liegt in seiner Einfachheit. Es ist im Prinzip sehr leicht festzustellen, ob eine Funktionalität in mehreren Bausteinen wiederkehrt, auch wenn es dabei natürlich einen gewissen Spielraum bei der Frage gibt, ob zwei Varianten einer Funktionalität gleich oder nur ähnlich sind. Zudem berichtet [BECK+98] von einem überraschenden Nebeneffekt: wenn man das Prinzip *„Einmal und nur einmal“* konsequent umsetzt, verbessert sich die Systemstruktur automatisch auch hinsichtlich anderer wünschenswerter Eigenschaften. In einem Prozeß der Selbstorganisation verringert sich insbesondere die Kopplung der Bausteine, weil sich zeigt, daß das Prinzip *„Einmal und nur einmal“* in einem „Spaghettisystem“ nicht verwirklicht werden kann. Sollten sich diese Beobachtungen in weiteren empirischen Untersuchungen bestätigen, könnte sich *„Einmal und nur einmal“* in Zukunft zu einer der wichtigsten praktischen Entwurfsregeln entwickeln.

2.1.2 Schwache Kopplung

Dem Ziel, die Auswirkungen von Änderungen lokal zu begrenzen, dient auch die Forderung nach *geringer Kopplung*. Generell heißt dies, daß jeder Baustein nur minimale Annahmen über seine Umgebung (die übrigen Bausteine im selben System) machen sollte. Aus Sicht jedes Bausteins müssen wir dabei die Umgebung in zwei Gruppen einteilen: diejenigen anderen Bausteine, die die Dienste des betrachteten Bausteins in Anspruch nehmen (clients)[5] und diejenigen, deren Dienste der betrachtete Baustein benötigt (server). [MARTIN95] bezeichnet Kopplungen der ersten Gruppe als *afferente Kopplungen*, die der zweiten als *efferente Kopplungen*. Zur Vermeidung von Verwechslungen werden wir die erste Gruppe (Abhängigkeit eines Server von einem Client) als *exportierte* und die zweite (Abhängigkeit eines Client von einem Server) als *importierte Kopplungen* bezeichnen. Beide Gruppen verlangen unterschiedliche Methoden zur Reduzierung der Kopplung.

Weitgehend etabliert haben sich die Methoden zur Verminderung exportierter Abhängigkeiten. Das wichtigste Prinzip ist zweifellos das *Geheimnisprinzip* [PARNAS72]: jeder Baustein muß die Implementation seiner Funktionalität kapseln und darf seine Dienste nur über eine Schnittstelle zur Verfügung stellen. Folgeänderungen bei den Nutzern der Dienste werden vermieden, solange interne Änderungen eines Bausteins hinter der Schnittstelle verborgen bleiben. Wir nennen diese Schnittstelle die *exportierte* (exported interface) oder *angebotene Schnittstelle* (offered interface).

[5] Als Client bezeichnen wir in dieser Arbeit ganz allgemein einen Softwarebaustein, der die Dienste eines anderen Bausteins (des Server) in Anspruch nimmt, um seine eigene Aufgabe zu erfüllen.

Offered Interface: Ein offered interface ist eine Beschreibung der Dienste eines Bausteins sowie der Mechanismen zu deren Aktivierung, die Implementationsdetails so weit wie möglich verbirgt.

Die angebotene Schnittstelle eines Bausteins sollte *vollständig* sein. Das heißt, die Clients sind in der Lage festzustellen, in welchem Punkt seines Zustandsraumes sich ein Baustein befindet (observability), und sie können seinen Zustand ändern, wobei in mehreren Schritten jeder Punkt des Zustandsraumes erreichbar ist (controllability, vgl. [WEIDE+96]). Dabei bleiben jedoch die grundlegenden Invarianten des Bausteins erhalten. Öffentlich zugängliche Funktionen können also den Baustein nicht in einen unerlaubten (inkonsistenten) Zustand versetzen.

Gleichartige Dienste von verschiedenen Bausteinen lassen sich in einer verallgemeinerten Schnittstelle zusammenfassen (*Generalisierung*). Dadurch werden diese Bausteine aus Sicht der Schnittstelle austauschbar. Überdies können wir Meta-Schnittstellen definieren, die die Eigenschaften eines Bausteins beschreiben oder sogar sein Verhalten zu manipulieren gestatten, so daß der Baustein an die Anforderungen seiner Clients besser angepaßt werden kann.

Stärker noch als die exportierten bestimmen die *importierten Kopplungen*, wie flexibel ein System gegenüber Änderungsanforderungen ist und wie gut Änderungen lokal begrenzt bleiben. Zunächst einmal muß jeder Baustein seine importierten Kopplungen und alle damit verbundenen Annahmen *explizit deklarieren*. Wir nennen eine solche Deklaration die *importierte Schnittstelle* (imported interface). Noch besser wird der Sachverhalt durch den englischen Begriff required interface ausgedrückt.

Required Interface: Ein required interface ist eine explizite Beschreibung aller Schnittstellen und der damit verbundenen funktionalen Annahmen, von denen ein Baustein abhängt.

Jeder Baustein definiert demnach offered interfaces zur Beschreibung seiner Dienste und required interfaces zur Beschreibung seiner Abhängigkeiten. Entscheidend ist dabei die Feststellung, daß beide Beschreibungen nur dem Baustein "gehören", der sie definiert. Sie können und müssen nur dann geändert werden, wenn sich die Eigenschaften dieses einen Bausteins ändern. Es sei bemerkt, daß diese interfaces auf jeden Fall explizit, aber nicht notwendigerweise abstrakt[6] sein müssen.

Durch die Unterscheidung exportierter und importierter Kopplungen können wir eine neue Antwort auf die alte Frage nach *minimalen Schnittstellen* geben. Es zeigt sich nämlich, daß required interfaces minimal sein sollten, während diese Forderung für offered interfaces nicht aufrecht erhalten werden kann. Um festzustellen, welche minimale Funktionalität ein Server bieten sollte, müßte man die Anforderungen aller

[6] Eine abstrakte Schnittstelle bezieht sich nicht auf einen bestimmten Server und kann deshalb durch verschiedene Server implementiert werden. Ein konkrete Schnittstelle benennt hingegen einen bestimmten Server, der nicht ohne weiteres ausgetauscht werden kann.

Clients vollständig kennen, bevor man ein offered interface definiert. In der Praxis ist dies jedoch unmöglich, denn es können jederzeit neue Clients hinzugefügt werden. Folglich wird jeder Server versuchen, möglichst viele Clients zufriedenzustellen, indem er von vornherein mehrere Möglichkeiten bietet, seine Dienste zu verwenden (z.B. sogenannte convenience functions). Dadurch ist seine Schnittstelle jedoch nicht mehr minimal.

Ganz anders sieht es dagegen bei required interfaces aus. Hier kennen wir genau die Anforderungen des jeweiligen Client, denn das required interface wird von diesem Client selbst spezifiziert und „gehört" nur ihm. Deshalb ist es hier möglich, eine minimale Schnittstelle zu definieren. Minimale required interfaces sind sogar in höchstem Maße wünschenswert, denn je weniger Anforderungen ein Client an seine Server stellt, desto einfacher wird es sein, in einem bestimmten Kontext einen passenden Server zu finden. Gerade die Bestimmung minimaler Anforderungen von Algorithmen an ihre Datenstrukturen erweist sich als ein Grundpfeiler der generischen Programmierung und leistet einen wesentlichen Beitrag zur hohen Flexibilität generischer Algorithmen [MUSSTEP89, MUSSTEP94].

Mit der Einführung von expliziten Beschreibungen für importierte und exportierte Abhängigkeiten haben wir bereits einen wichtigen Schritt zum Management der Kopplungen getan. Wir können jetzt zumindest in jedem Falle feststellen, welche Folgen eine Änderung hat. Dies ist sehr wichtig, denn es sind gerade die impliziten Annahmen und Abhängigkeiten, die in vielen Systemen zu unerwarteten und unverständlichen Fehlern in vermeintlich nicht geänderten Programmteilen führen.

Prinzipiell gibt es zwei Möglichkeiten, importierte Kopplungen explizit zu beschreiben: durch Angabe des Namens einer geeigneten Spezifikation oder als Liste der erforderlichen Operationen. Im Rahmen unserer Diskussion wollen wir diese Varianten als named interface bzw. syntactic interface bezeichnen.

Mit der Angabe des Namens einer Spezifikation verweist der Baustein auf die in dieser Spezifikation definierten Eigenschaften. Bausteine kommen als Server in Frage, wenn sie eine Schnittstelle mit dem betreffenden Namen anbieten. Der Name impliziert im Idealfall, daß der Baustein die erforderliche Spezifikation auch tatsächlich implementiert. In traditionellen prozeduralen Systemen entspricht jedem Namen in der Regel genau ein Baustein. Wenn beispielsweise in C eine Funktion als Argument ein `struct Image` verlangt oder wenn sie den Ausdruck `abs(a)` aufruft, dann wird die betreffende Funktionalität durch genau eine Datenstruktur `struct Image` bzw. genau eine Funktion `abs` implementiert. Damit entsteht zwangsläufig eine feste Kopplung zwischen dem Client und seinem Server, die Möglichkeiten zur Anpassung der Bausteine an unterschiedliche Erfordernisse sind dadurch sehr eingeschränkt.

Wir können eine geringere Kopplung und damit höhere Flexibilität erreichen, wenn dem Namen eine *abstrakte Spezifikation* zugeordnet werden kann. Einige solche Techniken gibt es bereits in der prozeduralen Programmierung, wie z.B. das function overloading (derselbe Funktionsname wird durch mehrere Funktionen mit jeweils

unterschiedlichen Argumenten implementiert; dies wird von C++, jedoch nicht von C unterstützt) und der function pointer (ein Name legt nur eine Funktionssignatur fest, jede Funktion mit entsprechender Signatur kann benutzt werden).

Inzwischen gibt es weit leistungsfähigere Mittel, abstrakte Spezifikationen in einer Programmiersprache oder einer formalen Beschreibungssprache auszudrükken, beispielsweise mit Hilfe abstrakter Basisklassen in objekt-orientierten Sprachen oder mit den interfaces der verschiedenen middleware-Technologien (z.B. CORBA [OMG98] und DCOM [REDMOND97]). Der entscheidende Punkt besteht dabei darin, daß die Abstraktion während der Implementierung nicht verlorengeht. Auf dem Papier waren abstrakte Spezifikationen schon immer möglich. Jedoch erforderte die Implementation lange Zeit konkrete Bausteine, die weit weniger flexibel waren als die ursprüngliche Spezifikation auf Papier. Wir werden in Abschnitt 2.2 an konkreten Code-Beispielen zeigen, wie ernst dieses Problem ist.

Anstelle der Angabe eines Namens kann ein Baustein sein required interface auch als Liste der erforderlichen Operationen beschreiben, d.h. in Form einer *syntaktischen Spezifikation*. Im Grunde geschieht dies bereits dadurch, daß der Baustein die Operationen in seiner Implementation aufruft. Allerdings wäre diese Beschreibung nicht, wie gefordert, *explizit*. Der Benutzer des Bausteins müßte das required interface erst durch Inspektion des Quellcodes herausfinden, was zumindest erheblichen Aufwand erfordert (sofern der Quellcode überhaupt zur Verfügung steht). Auf der anderen Seite bietet im Moment keine der gängigen Programmiersprachen ein Sprachkonstrukt zur Formulierung einer syntaktischen Spezifikation. Zur Zeit müssen solche Spezifikationen als Bestandteil der Dokumentation geliefert werden.

Eine Liste notwendiger abstrakter Operationen definiert implizit einen formalen Typ (oder mehrere). Da aber für diesen Typ nicht ausdrücklich ein bestimmter Name gefordert wird (wie bei der Verwendung eines named interface), können leichter passende Server gefunden werden, weil die Entscheidung später getroffen werden kann: bei einem named interface muß bereits der Server-Programmierer sichern, daß der Server die mit dem Namen verbundene Spezifikation exakt erfüllt. Bei einem syntactic interface hingegen muß die Konformität erst dann überprüft werden, wenn ein bestimmtes Client-Server-Paar tatsächlich verbunden werden soll. Der Programmierer, der diese Verbindung herstellt, hat wegen seiner besseren Kenntnis der konkreten Situation größeren Spielraum bei der Entscheidung, auf welche Weise der Vertrag zwischen Client und Server zustande kommen kann.

Allerdings ist die Realisierung einer abstrakten Schnittstelle keineswegs einfach: selbst wenn ein Baustein auf einer abstrakten Beschreibungsebene nur allgemeine Anforderungen stellt, geht die Allgemeinheit bei der Implementation häufig verloren. Viele Programmiersprachen, besonders die älteren prozeduralen Sprachen wie C, erlauben es nicht, Bausteine so allgemein zu implementieren, wie dies im Prinzip möglich wäre. Bessere Möglichkeiten bieten objekt-orientierte und generische Sprachen. Wir werden diese Fragen in Abschnitt 2.2 weiter vertiefen.

Auch wenn man die Abhängigkeiten zwischen Bausteinen minimiert, kann und will man sie nicht ganz beseitigen, denn die Bausteine sollen zur Lösung eines komplexen Problems zusammenwirken. Wir müssen deshalb die verbleibenden Abhängigkeiten so gestalten, daß sie zu nicht-lokalen Änderungen wenig Anlaß geben. Es gibt hierfür drei grundlegende Strategien, die sich in der einen oder anderen Form in den klassischen Arbeiten finden:

Abhängigkeit von stabilen Bausteinen: Wenn ein Baustein nur von solchen Bausteinen abhängt, die sich nie ändern, werden seine required interfaces immer erfüllt sein. Da man niemals sämtliche Aspekte eines Bausteins flexibel gestalten kann, ergibt sich die Forderung nach stabilen Annahmen als praktische Notwendigkeit – auf bestimmte Dinge muß man sich verlassen können, wenn man überhaupt ein System fertigstellen will. So ist es z.B. unproblematisch, wenn ein Baustein direkt von den Konstrukten der Programmiersprache oder den Funktionen einer Standardbibliothek abhängt. Auch innerhalb jedes Projekts wird man meist einige grundlegende Bausteine fixieren.

Allerdings ist die Annahme der Stabilität zweischneidig, denn der Mechanismus wirkt auch in umgekehrter Richtung: wenn ein Baustein erklärt, von einem bestimmten anderen Baustein unlösbar abzuhängen, so ist der letztere damit effektiv an seine derzeitige Funktionalität gebunden, gleichsam "gefesselt". Jede Änderung des Servers kann dazu führen, daß die Annahmen des Client nicht mehr erfüllt sind, so daß Folgeänderungen notwendig werden. Dies ist besonders in prozeduralen Sprachen zu beobachten: eine Datenstruktur, von der viele Algorithmen abhängen, kann effektiv kaum noch geändert werden, weil alle Algorithmen an die Änderungen angepaßt werden müßten. Wir müssen deshalb vermeiden, daß Bausteine direkt von denjenigen Bausteinen abhängen, die wir möglicherweise ändern müssen.

Abstrakte Abhängigkeiten: Änderungen des Server werden möglich, wenn der Client ein abstraktes required interface definiert. Jeder Baustein, der zu dem required interface konform ist, kann als Server eingesetzt werden. Dazu ist es notwendig, einen abstrakten Entwurf so in einer Programmiersprache umzusetzen, daß die Abstraktion nicht verlorengeht. Hierzu benötigen wir die Unterstützung der Programmiersprache, damit einerseits die Abstraktion möglichst leicht ausgedrückt werden kann und andererseits die Rechengeschwindigkeit durch zusätzliche Flexibilität nicht über Gebühr leidet.

Konfigurierbare Bausteine: Anstelle eines abstrakten required interface oder zusätzlich zu diesem kann ein Baustein auch mehrere alternative required interfaces angeben. Auf diese Weise kann sich der Baustein aktiv an den jeweiligen Kontext anpassen. Mit Hilfe von Metainformationen, die den aktuellen Kontext beschreiben, wird jeweils das geeignete required interface ausgewählt. Die Konfiguration kann durch *Auswahl* geschehen, das heißt durch

Aktivieren einer von mehreren vorgefertigten Möglichkeiten. Weit flexibler ist allerdings die Konfiguration durch *Generierung*: aus einer formalen Spezifikation des aktuellen Kontexts wird ein Baustein erzeugt, der die gewünschte Funktionalität in diesem Kontext realisiert. Code-Generatoren (zu denen wir auch den template-Mechanismus von C++ rechnen dürfen) sind hierfür ebenso Beispiele wie die unter der Bezeichnung Computational Reflection bekannten Techniken [MAES88].

An dieser Stelle erkennt man auch, warum offered und required interfaces getrennt betrachtet werden müssen. Es könnte ja zunächst so scheinen, daß diese Trennung nur unnötige Komplexität ins System bringt. Abbildung 1 zeigt ein Beispiel für einen Entwurf, bei dem Client und Server nur durch jeweils eine Schnittstelle verbunden sind.

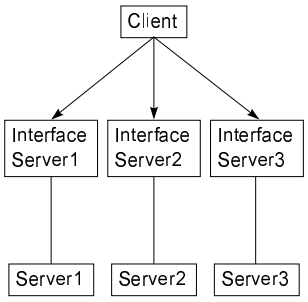


Abbildung 1: Entwurf, der offered und required interfaces gleichsetzt.

Obwohl dies auf den ersten Blick einfach aussieht, bringt es eine wesentliche Einschränkung mit sich. Wir haben uns nämlich mit diesem Entwurf auf stabile Annahmen festgelegt: da Client und Server von derselben Schnittstelle abhängen, muß diese Schnittstelle *zwingend standardisiert sein*. Es ist dabei nicht entscheidend, ob es sich um einen offiziellen ISO-Standard oder einen projektinternen Standard handelt, wichtig ist nur die Stabilität der Schnittstelle. Eine Standardisierung ist jedoch nur dann sinnvoll, wenn sie auf stabilen, allgemein akzeptierten Annahmen beruht. Gerade im Computer Vision-Kontext ist dies aber häufig nicht der Fall.

Noch schwerer wiegt allerdings ein zweiter Nachteil, der sich aus dem Problem der Subjektivität ergibt [HAROSSH93]:

Problem der Subjektivität: Kein einzelnes offered interface kann die Anforderungen aller potentiellen Clients gleich gut abdecken.

Auch wenn verschiedene Clients die Dienste desselben Servers benötigen, können sich ihre Anforderungen im Detail stark unterscheiden. Deshalb neigen offered interfaces dazu, mehr als das absolute Minimum an Funktionalität anzubieten, denn dies macht sie für eine größere Anzahl von Clients attraktiv. Werden nun die

required interfaces der Clients mit der „fetten" Server-Schnittstelle gleichgesetzt, geht deren erforderliche Minimalität verloren: Clients beginnen von Funktionalität abzuhängen, die sie gar nicht benötigen, und selbst die benötigte Funktionalität ist möglicherweise nicht in optimaler Form präsent, weil die Server-Schnittstelle auf viele Clients Rücksicht nehmen muß. Zwar könnte man verlangen, daß die Abhängigkeit dadurch reduziert wird, daß der Client nur die absolut notwendige Funktionalität *benutzt*. Aber dies funktioniert in der Praxis nicht, weil Programmierer in der Regel eine Schnittstelle vollständig ausnutzen, es sei denn, das erlaubte Subset wäre *explizit* spezifiziert – und genau das ist der Sinn eines separaten required interface.

Zur Abgrenzung gegenüber der bottom-up-Abstraktion, die durch Kapselung und Generalisierung auf der Server-Seite erreicht wird, wollen wir einen Entwurf, bei dem jeder Client seine Anforderungen in einem separaten, minimalen required interface beschreibt, als *top-down*-Abstraktion bezeichnen:

Prinzip der top-down-Abstraktion: Flexible Clients beschreiben in Form eines abstrakten required interface minimale Anforderungen an akzeptable Server.

Abbildung 2 verdeutlicht, wie wir durch die Trennung von offered und required interfaces, also durch top-down-Abstraktion, ein hohes Maß an Flexibilität gewinnen können. Dank dieser Trennung können wir je nach Situation entscheiden, welche Art der Verbindung zwischen Client und Server jeweils die zweckmäßigste ist: wir können jedes required interface auf ein anderes Serverinterface abbilden, wir können einen Server wählen, der mehrere required interfaces gleichzeitig abdeckt, und wir können Adapter dazwischen schalten, die die Anforderungen auf das tatsächlich Gebotene abbilden. Ist das required interface genügend abstrakt definiert, können diese Optionen ohne Modifikation des Client realisiert werden.

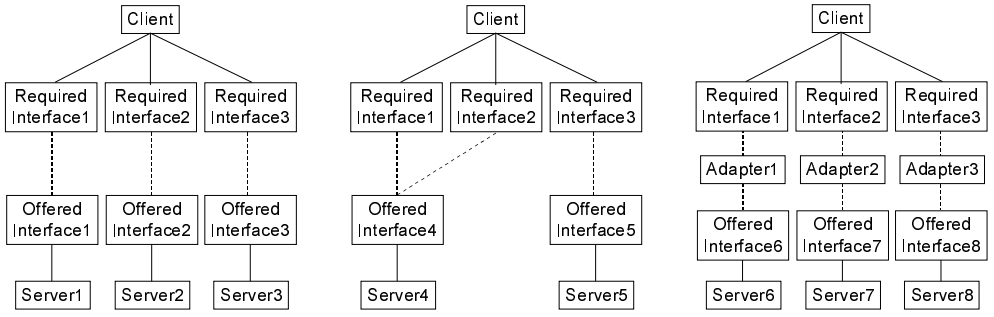


Abbildung 2: Die Trennung von offered und required interfaces eröffnet viele verschiedene Optionen, Client und Server zu verbinden.

Wenn required interfaces und offered interfaces konzeptionell voneinander unabhängig sind, besteht die Gefahr, daß sie nicht immer direkt zusammenpassen. Mögliche Konflikte können auf zwei Arten gelöst werden:

Adaption: Wir definieren Schnittstellen-Adaption als die *Übersetzung* eines required interface in ein nicht vollständig zu diesem konformes offered interface. Wir schalten zwischen Client und Server einen Adapter, der dem Client suggeriert, daß es den von ihm geforderten Server tatsächlich gibt. Hinter der Fassade des Adapters kann die geforderte Funktionalität auf verschiedene Arten implementiert werden. Einige bewährte Varianten sind beispielsweise als Entwurfsmuster in [GHJV94] beschrieben: im *Adapter*-Muster werden die Anforderungen direkt in die Angebote eines Servers übersetzt, beim *Decorator*-Muster fügt der Adapter zusätzliche Funktionalität hinzu, die der Server nicht selbst anbietet, und beim *Mediator*- sowie beim *Facade*-Muster koordiniert ein Adapter das Zusammenwirken mehrerer Server.

Dies unterstreicht die Wichtigkeit separater required interfaces: da sie explizit und formal sind, ist das Ziel der Adaption exakt bestimmt, und da sie minimal sind, muß der Adapter niemals komplizierter sein als nötig.

Standardisierung: Trotz der vielfältigen Möglichkeiten zur Adaption ist es von Vorteil, wenn Schnittstellen von vornherein zusammenpassen. Wir definieren Schnittstellen-Standardisierung als den Prozeß der *Abstimmung* von required interfaces und offered interfaces, so daß entweder kein Adapter notwendig ist, oder ein solcher bereits mit standardisiert wird. Man beachte, daß Standardisierung die Trennung zwischen required und offered interfaces nicht aufhebt – auch wenn sie ohne Adapter zusammenpassen, bleiben sie doch separiert (das required interface könnte z.B. eine Untermenge des offered interface sein).

Neben den Schnittstellen selbst können wir auch Schnittstellenbeschreibungen auf einer Meta-Ebene standardisieren. Dadurch gewinnen wir größeren Spielraum bei der Gestaltung alternativer Schnittstellenvarianten, denn mit Hilfe der Metainformationen kann der jeweils benötigte Adapter automatisch generiert werden.

Gerade die Kombination beider Möglichkeiten, also die Verwendung von *standardisierten Adaptern*, erweist sich zur Lösung des Problems der Subjektivität als besonders zweckmäßig: sie gestattet, daß ein Server vielen verschiedenen Clients dienen kann, ohne daß seine Schnittstelle dadurch unnötig aufgebläht wird. Ein bekanntes Beispiel für einen standardisierten Adapter ist der *Iterator*: viele verschiedene Datenstrukturen mit ganz unterschiedlichen Eigenschaften können einen Iterator mit standardisierter Schnittstelle anbieten, so daß Algorithmen sich nicht um die Besonderheiten jeder Datenstruktur kümmern müssen.

Die Idee der top-down-Abstraktion ist nicht neu. Es fehlt aber bisher ein übergreifender Begriff für jene Techniken, die diese Grundidee in jeweils sehr unterschiedlicher Weise verwirklichen. Wir haben hier die Begriffe der top-down-Abstraktion und des required interface eingeführt, um das Wesentliche und Gemeinsame hinter

den individuellen Methoden besser sichtbar werden zu lassen. Einige Beispiele sollen dies verdeutlichen.

Ansätze der top-down-Abstraktion finden wir zum Beispiel seit langem in Form von Gerätetreibern in Betriebssystemen. Das Betriebssystem spezifiziert ein required interface für Druckertreiber, und jeder Server, der eine entsprechende Schnittstelle offeriert, kann als Druckertreiber angesprochen werden. Dabei spielt es gar keine Rolle, ob der jeweilige Server wirklich einen Drucker repräsentiert. So wird der Adobe Acrobat Distiller zwar als Druckertreiber installiert, er erzeugt aber in Wirklichkeit ein File im Portable Document Format (pdf). Eine analoge Idee liegt auch den callbacks gängiger Fenstersysteme (wie z.B. OSF Motif) sowie den Plug-In Interfaces zugrunde, die in Graphikprogrammen und Internetbrowsern häufig anzutreffen sind.

Top-down Abstraktion gehört zu den wesentlichen Triebkräften bei der zur Zeit sehr intensiv erforschten Rollenmodellierung [REENS+96, RIEHLEGRO98]. Eine *Rolle* definiert eine Sicht, die ein Objekt auf ein anderes Objekt hat. Unterschiedliche Clients können unterschiedliche Sichten auf dasselbe Objekt haben, das Objekt kann dadurch sehr verschiedene Rollen spielen. Eine Rolle beschreibt also die erforderliche Minimalfunktionalität, die ein Server *im Kontext eines bestimmten Clients* zur Verfügung stellen muß. Damit entspricht der Rollenbegriff weitgehend unserem Begriff des required interface.

Auch *abstrakte Basisklassen* innerhalb der objekt-orientierten Programmierung sowie die Anforderungen an template-Parameter in der generischen Programmierung können wir als required interfaces interpretieren. Diese Varianten werden wir weiter unten anhand von Beispielen aus dem Bereich Computer Vision näher beleuchten.

2.1.3 Variationspunkte und iterative Entwicklung

Im Verlauf der bisherigen Diskussion sind wir noch nicht darauf eingegangen, wieviel Flexibilität in ein System eingebaut werden sollte und auf welche Weise dies geschieht. Diese Entscheidung wird wesentlich von den Kosten beeinflußt, die Flexibilität verursacht: die Entwicklung eines flexiblen Systems ist derzeit aufwendiger als die eines inflexiblen, die Einarbeitung für die Benutzer ist schwieriger, und die Performanz kann durch Flexibilität beeinträchtigt werden. Zuviel Flexibilität verursacht deshalb unnötige Kosten. Auf der anderen Seite können aber auch durch fehlende Flexibilität erhebliche Kosten entstehen, nämlich wenn dadurch größere Änderungen oder eine Neuentwicklung erforderlich werden. Man muß deshalb sehr genau abwägen, wieviel Flexibilität ein System anbieten sollte und mit welchen Mitteln. Zur Klärung dieser Frage wollen wir Flexibilität genauer definieren:

Flexibilität: Wir bezeichnen ein System als flexibel, wenn die *wahrscheinlichen* Änderungsanforderungen durch lokale Modifikationen des Systems realisiert werden können.

Das heißt, daß wir zunächst die Wahrscheinlichkeit möglicher Änderungen abschätzen müssen. Ist die Wahrscheinlichkeit für eine bestimmte Änderung gering, oder kann die Art dieser Änderung nicht genau genug vorhergesagt werden, sollte auf Flexibilität verzichtet werden. Baut man Flexibilität nur „auf Verdacht" ein, kann es sehr leicht passieren, daß die betreffende Änderung später gar nicht oder in anderer Form als erwartet notwendig wird. Dadurch entsteht die berüchtigte „Übergeneralisierung", bei der der Aufwand für den Einbau von Flexibilität ihren Nutzen weit übersteigt. So wird unnötigerweise die Systemkomplexität erhöht, und für die Implementierung der eigentlichen Funktionalität bleibt nicht mehr genügend Zeit. Leider findet man solche Systeme nicht selten – das Image Understanding Environment ist ein typisches Beispiel aus dem Computer-Vision-Bereich (vergl. Seite 29).

Am sichersten läßt sich notwendige Flexibilität erkennen, wenn die betreffende Änderung tatsächlich eingetreten ist. Dann wissen wir genau, an welcher Stelle und auf welche Weise ein neuer *Variationspunkt* in das System eingebaut werden muß. Als Variationspunkt (variation point oder hot spot) bezeichnet man einen explizit gekennzeichneten, wohl-isolierten Teil eines Softwaresystems, der die Änderung eines bestimmten Aspekts der Funktionalität erlaubt bzw. erleichtert [PREE97]. Flexible Systeme sind dadurch gekennzeichnet, daß die am häufigsten benötigten Variationspunkte bereits bestehen und neue relativ leicht eingebaut werden können. Bei unflexiblen Systemen kann dies hingegen sehr großen Aufwand erfordern. Der Einbau eines Variationspunktes erfolgt in zwei Schritten (die Schritte sind eine Art „konstruktive Definition" der Idee des Variationspunktes):

1. Wenn wir feststellen, daß sich ein Aspekt mit genügend hoher Wahrscheinlichkeit ändern wird (oder bereits geändert hat), müssen wir zunächst für diesen Aspekt die *Möglichkeit der Flexibilität* schaffen. Wir identifizieren dafür zuerst alle Bausteine, die von Folgeänderungen betroffen sind, die also von (vermeintlich stabilen) Annahmen über den geänderten Aspekt abhängen. Sodann identifizieren wir *aus Sicht jedes betroffenen Bausteins* die Invarianten dieses Aspekts, also diejenigen Annahmen, die allen Alternativen gemeinsam sind. Diese Invarianten formuliert jeder Baustein in Form eines abstrakten required interface. Schließlich wird jeder betroffene Baustein so modifiziert, daß er nur noch auf seinem neuen required interface aufbaut. Dadurch wird er gleichsam gegen zukünftige Änderungen dieses Aspekts „immunisiert".
2. Nach dem ersten Schritt, der die Möglichkeit der Flexibilität geschaffen hat, folgt nun die Definition und Implementation von Alternativen für die verschiedenen Situationen. Auf Änderungen des betreffenden Aspekts kann dann einfach dadurch reagiert werden, daß die jeweils gerade angemessene

Alternative aktiviert wird. Wenn nötig, kann auch eine neue Alternative hinzugefügt werden, ohne daß dies Folgeänderungen in den Clients verursachen würde.

Wir können uns dies so veranschaulichen, daß wir im ersten Schritt eine neue *Dimension der Flexibilität*, eine neue *Abstraktionsachse*, schaffen (jeder Variationspunkt entspricht einer solchen Achse), während wir im zweiten Schritt Punkte auf dieser Achse definieren und auswählen. Das heißt, die lokale Änderbarkeit ist für diejenigen Anforderungen gewährleistet, für die bereits eine Abstraktionsachse existiert: in diesem Fall muß nur noch eine Alternative ausgewählt bzw. eine neue implementiert werden. Nicht-lokale Änderungen werden hingegen notwendig, wenn der betreffende Aspekt noch nicht durch eine Abstraktionsachse repräsentiert wird, so daß der Code umstrukturiert werden muß, um eine neue Achse zu schaffen.

Umstritten ist allerdings nach wie vor, wann der richtige Zeitpunkt zur Einführung eines neuen Variationspunktes ist. Traditionell herrschte die Auffassung vor, daß das nachträgliche Einfügen eines Variationspunktes in ein existierendes System mit sehr hohem Aufwand verbunden ist. Deshalb wurde empfohlen, in einer sorgfältigen Analyse die notwendigen Variationspunkte vorab zu identifizieren und sie von vornherein in das System einzubauen (Wasserfallmodell, vgl. [BALZERT96]). Allerdings ist eine solche Analyse ebenfalls sehr aufwendig. Zudem besteht die Gefahr, daß nicht alle notwendigen Variationen erkannt werden oder die Lösung für die später tatsächlich eintretende Änderung nicht optimal ist.

Inzwischen hat sich mehr und mehr die Erkenntnis durchgesetzt, daß flexible Systeme nur *iterativ* entstehen können, weil viele für die Flexibilität wichtige Fragen erst dann beantwortet werden können, wenn ein Teil des Systems bereits implementiert ist (vgl. z.B. [COCKBURN98]). Deshalb beginnt man bei der iterativen Vorgehensweise mit einer Systemversion, die zunächst nur wenige Variationspunkte besitzt. Anhand dieser ersten Version wird dann analysiert und entschieden, wo und mit welchen Mitteln bei der nächsten Iteration weitere Variationspunkte eingebaut werden müssen. Diese Schritte werden mehrmals wiederholt (Spiralmodell, iterative Entwicklung, vgl. [BALZERT96]).

Auch die Ergebnisse der vorliegenden Arbeit sind das Resultat eines iterativen Entwicklungsprozesses.[7] Durch die Notwendigkeit, neue Funktionalität zu integrieren, entstanden immer wieder neue Anforderungen, die durch die bestehenden Variationspunkte nicht abgedeckt waren. Dies konnte auch durch vorherige Analyse nicht vollständig vermieden werden, weil häufig die besonderen Detailanforderungen eines Verfahrens erst während der Implementierung zutage traten. Interessanterweise hat sich gezeigt, daß die Struktur der Schnittstellen dank fortgesetzter

[7] Dies wird zum Teil auch bei der Präsentation der Ergebnisse in dieser Arbeit sichtbar. Beispielsweise befassen wir uns insgesamt viermal (in den Anschnitten 2.2, 4.1, 4.4 und 4.7) mit Punktoperationen auf Bildern und entwickeln jeweils neue, verbesserte Versionen dieser Algorithmen.

Anpassung im Laufe der Zeit wesentlich *einfacher* geworden ist. Die auf reiner Analyse beruhenden Konzepte neigten dagegen eher zu unnötiger Komplexität. Insgesamt war eine Tendenz zu einer immer feinkörnigeren Zerlegung zu beobachten, wodurch sich die Kombinationsmöglichkeiten der einzelnen Bausteine stark verbessert haben.

2.1.4 Unabhängige Abstraktionsachsen und das Problem des kartesischen Produkts

Die im vorigen Abschnitt eingeführte Veranschaulichung von Flexibilität durch Abstraktionsachsen macht deutlich, daß Abstraktionen einen mehrdimensionalen Raum aufspannen. Die jeweils aktive Alternative auf jeder Achse kann mit einer Koordinate verglichen werden, und die augenblickliche Systemkonfiguration entspricht dem durch die Koordinaten festgelegten Punkt des Raumes. Mit zunehmender Zahl der Achsen wird dieser Raum immer komplexer, und die Zahl der möglichen Kombinationen (also die Zahl der möglichen Punkte des Raums) wächst exponentiell an: bei n Achsen mit je m Varianten gibt es m^n Möglichkeiten. Daraus erwächst das sogenannte library scaling problem oder, wegen der offensichtlichen mathematischen Analogie, das *Problem des kartesischen Produkts* [SZYPERSKI96]:

> **Problem des kartesischen Produkts:** In einem flexiblen System wächst die Anzahl der möglichen Konfigurationen exponentiell mit der Anzahl der Abstraktionsachsen. Mit steigender Systemgröße ist es unmöglich, alle Konfigurationen explizit zu implementieren.

Ein sehr bekanntes Beispiel für dieses Problem ist die Komponentenbibliothek von Booch [BOOCH87]. Boochs Komponentenbibliothek unterstützt vier grundlegende Abstraktionsachsen: data organization, concurrency, ordering und memory management. Die data organization-Achse umfaßt 17 verschiedene abstrakte Datenstrukturen, darunter stack, queue, tree usw. Die concurrency-Achse bietet drei Alternativen für Nebenläufigkeit: sequential (keine Nebenläufigkeit), guarded (der Programmierer muß Synchronisationssignale explizit aufrufen), und synchronized (automatische Synchronisation). Die ordering-Achse bietet nur zwei Möglichkeiten (geordnet oder ungeordnet), und die memory management-Achse wiederum drei. Daneben gibt es noch eine Reihe lokaler Abstraktionen. Um alle sinnvollen Kombinationen abzudekken, mußte Booch etwa 400 Bausteine implementieren (darunter zum Beispiel 18=3*2*3 verschiedene Varianten von queues).

Da den meisten Entwicklern nicht genügend Ressourcen zur Verfügung stehen, um alle Varianten ihres Anwendungsgebiets vollständig abzudecken, muß man nach Lösungen für das Problem des kartesischen Produkts suchen. Eine offensichtliche Idee besteht in dem Versuch, die Zahl der Kombinationen so weit wie möglich

zu reduzieren. Ein wichtiger Schritt in diese Richtung ist bereits die Forderung, Flexibilität nur dann einzubauen, wenn sie wirklich benötigt wird, um die Zahl der Achsen gering zu halten.

Ein weitere, besonders in der Computer Vision verbreitete Möglichkeit (siehe Abschnitt 2.2), ist die Verwendung sogenannter bottleneck interfaces [SZYPERSKI96]. Bei diesem Ansatz versucht man, eine Achse dadurch zu neutralisieren, daß man auf ihr nur eine Variante zur Verfügung stellt – solche Achsen tragen nicht zum exponentiellen Wachstum bei. Allerdings steckt man meist in einem Dilemma, welche Variante man wählen sollte. Wählt man die allgemeinste Variante, so wird das resultierende System im *typischen* Fall unnötig komplex und damit wenig effizient sein. Wählt man dagegen eine Variante, die näher am typischen Fall liegt, sind nicht mehr alle Fälle abgedeckt.

Eine leistungsfähigere Methode zur Reduzierung der Zahl der Achsen stellt die Verwendung *unabhängiger Abstraktionen* dar. Aus der Statistik ist bekannt, daß man die Zahl der Variablen oft drastisch reduzieren kann, wenn man unkorrelierte Variablen einführt (z.B. durch Hauptachsentransformation). Das gleiche gilt auch in der Softwareentwicklung: wir können ein flexibles System wesentlich vereinfachen, wenn wir unabhängige (*„orthogonale“*) Abstraktionsachsen einführen. Die Identifikation solcher Achsen ist ein wesentliches Ziel der Standardisierung.

Unabhängige Achsen allein reichen jedoch nicht aus. Sie reduzieren zwar die Anzahl der Möglichkeiten, aber der exponentielle Charakter des Wachstums bleibt trotzdem bestehen. Wir können dies jedoch auch positiv sehen – eine exponentielle Zahl von Möglichkeiten ist ein Zeichen großer Flexibilität. Voraussetzung für diesen neuen Blickwinkel ist allerdings, daß wir *nicht alle Kombinationen explizit programmieren müssen*. Wenn es gelänge, mit $m \times n$ Bausteinen m^n Kombinationsmöglichkeiten zu realisieren, wäre dies in der Tat eine sehr effiziente Verwendung der Ressourcen. Das heißt, wir müssen danach trachten, jede beliebige Kombination *bei Bedarf* durch Zusammenfügen geeigneter Basisbausteine *automatisch* zu generieren.

Don Batory [BATORY+93] faßt Methoden, die dies leisten, unter dem Begriff der *parametrisierten Bausteine* zusammen. Dieser Begriff meint, daß Bausteine nicht nur Dienste exportieren, sondern daß sie auch bestimmte Dienste importieren. Jeder importierte Dienst wird durch einen Typparameter repräsentiert. Durch Instanziierung der Parameter mit unterschiedlichen Servern ändert sich das Verhalten des parametrisierten Bausteins, er repräsentiert jeweils andere Punkte des Abstraktionsraumes. Damit entsprechen die Begriffe „exportierte und importierte Dienste“ im wesentlichen unseren offered und required interfaces.

Batory bezeichnet alle Bausteine, die die gleiche Schnittstelle exportieren, als Mitglieder eines realm. Die Parameter eines Bausteins verweisen ebenfalls auf jeweils ein realm, aus dem der Baustein stammen muß, der für den jeweiligen Parameter eingesetzt wird. Im folgenden Beispiel (zitiert aus [BATORY98]) gibt es drei verschiedene realms (`S`, `T`, `W`) mit jeweils drei Bausteinen, die zum Teil Parameter aus einem der realms importieren (in eckigen Klammern):

```
S = { a, b, c }

T = { d[S], e[S], f[S] }

W = { n[W], p, q[S,T] }
```

Eine bestimmte Realisierung dieser Bausteine bezeichnet Batory als Typgleichung (type equation). Deren Typ ist durch das realm des äußeren Bausteins bestimmt. Zum Beispiel sind

```
A1 = d[b]

A2 = n[ q[c,e[a]] ]
```

Typgleichungen des realms T bzw. W. Die Syntax der Typgleichungen ähnelt der Syntax eines Funktionsaufrufs, aber es handelt sich nicht um Ausdrücke auf Werten, sondern um Ausdrücke auf Typen.

Die realms können wir nun mit den Abstraktionsachsen identifizieren. Die Bausteine in einem realm sind somit die möglichen alternativen Realisierungen dieser Abstraktion. Jede Typgleichung beschreibt einen Punkt im Raum der möglichen Konfigurationen, der durch die Abstraktionsachsen aufgespannt wird. Daraus folgt, daß wir jede gewünschte Konfiguration *bei Bedarf* durch Kombination geeigneter Grundbausteine erzeugen können. Weder müssen alle Kombinationen vorfabriziert werden, wie in Boochs Bibliothek, noch müssen wir uns auf eine relativ kleine Untermenge aller Möglichkeiten beschränken, wie beim bottleneck interface.

Batory hat mit seiner Arbeitsgruppe verschiedene Implementationsstrategien für diesen Ansatz untersucht. Mit einer Kombination von Vererbung und templates (in C++) ist es ihnen gelungen, einen Teil von Boochs Komponentenbibliothek, der ursprünglich 82 Bausteine (ca. 11 000 Codezeilen C++) enthielt, mit nur 22 Bausteinen (ca. 2 700 Codezeilen) zu realisieren, wobei die neue Bibliothek zudem flexibler und schneller war als Boochs Original (vergleiche [BATORY98]).

Parametrisierte Bausteine sind auch die Grundlage der generischen Programmierung, wie wir in Kapitel 3 sehen werden. Im Unterschied zu Batorys Terminologie haben sich hier die Begriffe *Konzept* (für realm) und *Modell eines Konzepts* (für alternative Realisierungen innerhalb eines realm) eingebürgert. Überdies ist die generische Programmierung noch flexibler, da die exportierten Schnittstellen wiederum von den Typen der importierten Bausteine abhängen können. Zwei Instanziierungen desselben Bausteins mit verschiedenen importierten Bausteinen exportieren somit nicht notwendigerweise identische Schnittstellen.

Damit diese Flexibilität für den Benutzer noch handhabbar bleibt, benötigen wir leistungsfähige Methoden der Selbstkonfiguration, woraus sich die große Bedeutung der Metainformationen innerhalb der generischen Programmierung ergibt. Metainformationen beschreiben in maschinenlesbarer Form, welche der möglichen Varianten einer exportierten Schnittstelle jeweils realisiert wurden und welche Anforderungen für den Import weiterer Schnittstellen sich daraus ergeben.

2.2 Analyse bestehender Computer Vision-Software

Wir können die existierende Computer Vision-Software in drei große Gruppen einteilen: prozedurale, objekt-orientierte und interaktive Systeme.

Die Systeme der letzen Gruppe verwenden durchweg einen Schichtansatz: die interaktive Funktionalität baut auf einer unterliegenden prozeduralen oder objekt-orientierten Schicht auf. Die Flexibilität dieser Systeme wird also durch die Eigenschaften beider Schichten bestimmt. Mit Hilfe von interpretierten Kommandosprachen (wie z.B. bei Halcon/Horus [HORUS97]) oder visueller Programmierung (wie bei Khoros [RASKUB94]) kann auf der interaktiven Ebene sogar dann ein hohes Maß an Flexibilität erzielt werden, wenn die unterliegenden Algorithmen wenig flexibel sind.

Nehmen wir an, daß in Khoros zwei Bilder gegeben sind, die die partiellen Ableitungen eines Grauwertbildes in x- und y-Richtung enthalten. Wir können dann mit wenigen Mausklicks einen Algorithmus zusammenstellen, der in jedem Pixel aus den beiden partiellen Ableitungen erster Ordnung den Gradientenbetrag `sqrt(gx*gx + gy*gy)` berechnet. Wir benötigen hierfür zwei Bausteine für die Multiplikation, einen für die Addition und einen für das Radizieren. Diese elementaren Funktionsbausteine werden in Khoros glyph genannt. Wird eine andere Betragsnorm benötigt, z.B. die L1-Norm `abs(gx) + abs(gy)`, so löscht man wieder mit wenigen Mausklicks den Baustein für das Radizieren und ersetzt die Multiplikatoren durch Bausteine für den Absolutbetrag.

Allerdings hat dieses Vorgehen sehr negative Auswirkungen auf die Rechengeschwindigkeit: jeder glyph muß zunächst seine Eingabebilder von der Festplatte laden, dann seine Berechnungen ausführen und schließlich die Ergebnisse wieder speichern. Entwickelt man komplexere Algorithmen mit Hilfe der interaktiven Oberfläche, ist die Performanz für viele Anwendungen zu gering. Zwar kann man effizientere interaktive Systeme implementieren als Khoros, aber die Geschwindigkeit compilierter Sprachen ist nicht erreichbar. Hinzu kommt, daß interaktive Sprachen in erster Linie dazu entwickelt wurden, um andere Bausteine zu steuern. Deshalb eignen sie sich weniger zur direkten Implementation von Computer Vision-Funktionalität. Das heißt, auch wenn interaktive Systeme durchaus interessante Eigenschaften aufweisen, liegt der eigentliche Knackpunkt der Flexibilität auf der unteren Ebene. Wir werden uns deshalb in dieser Arbeit auf die Bausteine der unteren, compilierten Ebene konzentrieren.

Objekt-orientierte Computer Vision Systeme sind fast immer in C++ implementiert, prozedurale heute meist in C. Auch die letzteren werden nach wie vor breit eingesetzt. Bei unserer Analyse mußten wir uns auf frei zugängliche Systeme beschränken, da über kommerzielle Produkte keine hinreichend detaillierten

Informationen verfügbar waren, insbesondere nicht über Details der Implementation. Wir haben die folgenden Systeme in unsere Analyse einbezogen:

Prozedurale Systeme:

- *Khoros*: entwickelt an der University of New Mexico [KHOROS91, RASKUB94],[8]
- *Candela*: Computer Vision Labor (CVAP) der Königlichen Technischen Hochschule Stockholm [CANDELA98],
- *Tina*: Electronic Systems Group der University of Sheffield [POLLARD+97],
- *IPRS*: Machine Vision Laboratory der University of Melbourne [IPRS93],

Objekt-orientierte Systeme:

- *TargetJr*: ursprünglich am Image Understanding Lab der Firma General Electric entwickelt, wird es inzwischen von Computer Vision Gruppen weltweit weiterentwickelt [TARGETJR96],
- *Image Understanding Environment* (IUE): ein Projekt des amerikanischen Verteidigungsministeriums zur Schaffung eines de-facto-Standards für militärische Computer Vision Anwendungen, von der Firma Amerinex Applied Imaging implementiert und von verschiedenen Forschungsgruppen, vor allem in den USA und in Großbritannien, weiterentwickelt [DOLAN+96, IUE98].

Ebenfalls prozedural spezifiziert ist der internationale Bildverarbeitungsstandard PIKS (Programmer's Imaging Kernel System [PIKS94, PRATT95]). Dieser definiert einen 5-dimensionalen Bilddatentyp (3 Raumkoordinaten, Zeit, spektrale Dimension) sowie die üblichen Punktoperationen und Filter, wie wir sie auch in den anderen Computer Vision Systemen finden. Da dieser Standard keine Akzeptanz gefunden hat und es meines Wissens nur eine (kommerzielle) Implementation gibt, werden wir ihn im folgenden nur kurz analysieren.

Tabelle 1 auf Seite 28 gibt einen Überblick über die Eigenschaften der betrachteten Systeme. Man erkennt eine ganze Reihe von Übereinstimmungen zwischen den Systemen, aber auch wesentliche Unterschiede. Die Gemeinsamkeiten betreffen vor allem die grundlegende Bildverarbeitungsfunktionalität (Punktoperationen, Filter), aber auch eine Reihe von Analysefunktionen wie zum Beispiel Kantendetektion. Die Unterschiede bei den komplexeren Analysefunktionen sind dagegen relativ groß.

[8] Wir betrachten hier Khoros Version 1. Inzwischen liegt Khoros in der Version 2.3 vor. Khoros 2.x realisiert objekt-orientierte Konzepte, ohne eine objekt-orientierte Programmiersprache zu verwenden. Die damit verbundenen komplizierten Probleme können wir im Rahmen dieser Arbeit nicht diskutieren.

	Khoros	Tina	IPRS	Candela	TargetJr	IUE
Bilddatentypen	3D (Höhe, Breite, spektr. Bänder)	2D (Höhe, Breite)	nD (Höhe, Breite, ... nutzerdefin.)	2D (Höhe, Breite)	4D (Höhe, Breite, Tiefe, Zeit)	2D, 3D (Höhe, Breite, Tiefe)
Kennzeichnung des Pixeltyps	Typ-ID (RGB = 3 Bänder)	Typ-ID (kein RGB)	explizit (RGB = Pixeltyp)	Typ-ID (RGB = Array von Bildern)	Typ-ID (RGB = Pixeltyp)	Subklassen (RGB = Pixeltyp)
Kapselung	nein	Zugriffsfkt.	nein	Zugriffsfkt.	Pufferobj. Zugriffsfkt.	Pufferobj. Zugriffsfkt.
Punktoperationen	unäre, binäre Arithmetik, algebraische Funktionen, logische Op.	unäre, binäre Arithmetik, algebraische und nutzer-definierte Funktionen	beliebige unäre und binäre Operationen	unäre, binäre Arithmetik, algebraische Funktionen	unäre, binäre Arithmetik, algebraische und nutzer-definierte Funktionen	keine
Implementation	explizit für alle Typen	explizit für verschiedene Typen	Codegenerator	Interpreter, nur Typ float	C-Präprozessor (byte, short, int) explizit (float)	entfällt (da keine Operationen implementiert)
Filter	lineare, morphologische, Fourier	lineare, morphologische, Gabor, Fourier	lineare, Fourier	lineare, Fourier	lineare, nicht-lineare, Wavelet	keine
Implementation	nur Typ float (linear Filter) bzw. byte (morphologische Filter)	nur Typ float	explizit für alle Typen	nur Typ float	explizit (float) C-Präprozessor (einige Funktionen und Typen)	entfällt (da keine Filter implementiert)
Randbehandlung (vgl. Abschnitt 5.1)	Abschneiden des Kerns (äquivalent zu konstant 0)	konstant 0	undokumentiert	diverse	keine Berechnung am Rand	diverse durch Konfiguration des Pufferobjekts
Datenstrukturen für Bildanalyse (vgl. Kapitel 7)	keine (Bild wird für verschiedene Zwecke "mißbraucht")	Edgel, Geometrie (Punkt, Linie, Ebene u.a.)	Edge, Region, Junction, FeatureSpace	EdgeChain, Statistik	Geometrie, Topologie, Bildmerkmale Koordinaten-systeme, Statistik	Geometrie, Topologie, Bildmerkmale Koordinaten-systeme, Statistik
Bildanalyse-algorithmen	Kanten-, Linienerken., labeling, Klassifikation, Regionen-statistik	Merkmals-detektion matching model fitting (2D und 3D)	Merkmals-detektion shape from shading, focus, stereo Statistik	Kantenerk., Eckenerken., labeling, Distanztransf., Skelett	Merkmals-detektion grouping shape from stereo fitting (2D und 3D)	Merkmals-detektion fitting (weitere in Entwicklung)

Tabelle 1: Überblick über die Eigenschaften wichtiger Computer Vision Systeme (zusammengestellt durch den Autor anhand der Dokumentation und des Quellcodes der Systeme)
Erklärungen:
- 2D, 3D etc.: Bilder sind Arrays der angegebenen Dimension
- Typ-ID: ein Bilddatentyp kann verschiedene Pixeltypen enthalten, die durch ein Datenfeld in der Bilddatenstruktur (type tag) unterschieden werden
- explizit: für jeden Pixeltyp gibt es einen eigenen Bilddatentyp bzw. eine eigene Funktionsimplementation

(weitere Erklärungen im Text, siehe auch Abschnitt 7.1)

Noch gravierender sind allerdings die Unterschiede bei den Datenstrukturen. Dies beginnt bereits bei den Bilddatenstrukturen: Tina und Candela betrachten Bilder als 2-dimensionale Gitter mit skalaren Pixeltypen, in *Khoros* wird eine dritte Dimension hinzugefügt, um vektorielle Pixeltypen (z.B. RGB-Farben, komplexe Zahlen) repräsentieren zu können. TargetJr definiert vier Dimensionen (3 Raumdimensionen und die Zeit), und in IPRS können Bilder sogar beliebig hohe Dimension haben. Das Image Understanding Environment schließlich bietet eine umfangreiche Bibliothek von Bildklassen an, bei der die verschiedensten Bilddatentypen hierarchisch strukturiert und klassifiziert werden. Eine noch größere Variationsbreite findet man bei den Datenstrukturen für die Repräsentation von Bildanalyseergebnissen.

Die unmittelbare Folge der Verwendung so unterschiedlicher Datentypen ist die *völlige Inkompatibilität* der Algorithmen der verschiedenen Systeme. Es ist nicht möglich, einen Algorithmus beispielsweise aus dem Khoros-System herauszulösen und mit den Bilddatenstrukturen und Algorithmen von Tina zu verknüpfen. Man muß stets die entsprechenden Datenstrukturen und weitere unterstützende Funktionen ebenfalls in die neue Umgebung transferieren. Benutzt man Funktionalität aus verschiedenen Systemen, so existieren diese nebeneinander und lassen sich nicht ohne umfangreiche Quellcodeänderungen integrieren.

Die traditionelle Antwort auf derartige Inkompatibilitätsprobleme ist eine *Standardisierung der Datenstrukturen*. Dies ist das erklärte Entwurfsziel des Image Understanding Environment. Es bietet eine riesige Sammlung von Datenstrukturen für die unterschiedlichsten Anforderungen der Computer Vision. (Aus dieser Zielstellung heraus erklärt sich auch, warum das IUE nur relativ wenige Algorithmen anbietet.) Bei der Standardisierung des IUE wurden umfangreiche Erfahrungen aus früheren Entwicklungen berücksichtigt (insbesondere TargetJr), so daß die Klassenbibliothek umfassendes Computer Vision know-how widerspiegelt.

Jedoch hat die Standardisierung einen wesentlichen Nachteil: da ein Standard für längere Zeit festgeschrieben wird, müssen die standardisierten Datenstrukturen alle möglicherweise anfallenden Anforderungen von vornherein abdecken, denn nachträgliche Änderungen eines Standards sind sehr schwierig (bei ISO-Standards z.B. frühestens nach 5 Jahren). Die Anforderungen im Computer Vision-Bereich sind aber äußerst vielfältig, weil es bisher kaum kanonische, allgemein akzeptierte Lösungen gibt. Daher müssen die Datenstrukturen in PIKS und im IUE sehr leistungsfähig und allgemein sein, um breit einsetzbar zu bleiben.

Die Kehrseite des Versuchs, allen Anwendern gerecht zu werden, ist eine extrem hohe Komplexität der Bausteine. Dies erkennt man bereits an einer so einfachen Datenstruktur wie einem 2-dimensionalen Punkt. Die IUE-Klasse `IUE_point_2d` speichert nicht nur zwei Koordinatenwerte, sondern sie bietet umfangreiche Funktionalität, die mit Punkten nicht unmittelbar zu tun hat, darunter Referenzzählung (Speichermanagement), dynamische Typisierung (run-time type information über das

im C++-Standard Verfügbare hinaus), dynamischen Methodenaufruf aufgrund des Laufzeittyps mehrerer Argumente (engl. multi-methods bzw. multiple dispatch, vergl. [LEAVMILL98])[9], Ein- und Ausgabe in Dateien (Serialisierung der internen Repräsentation) und dynamische Attribute (Algorithmen können einem `IUE_point_2d` zur Laufzeit neue Attribute hinzufügen). Dazu kommt noch die für Punkte definierte Vektorarithmetik sowie verschiedene Varianten von Abstandsbestimmungen zwischen Objekten des Typs `IUE_point_2d` und anderen Punktklassen.

Da diese Funktionalität zum größten Teil über Vererbung realisiert wird, muß die Klasse `IUE_point_2d` zu ihrer Deklaration 70 weitere C++ Headerfiles inkludieren. Das sind 235 kBytes für eine so einfache Funktionalität wie das Verwalten eines Koordinatenpaars! Da die Vererbung eine statische Beziehung zwischen Klassen ist, kann die zusätzliche Funktionalität auch nicht ohne weiteres entfernt werden, wenn sie nicht benötigt wird. Es wären umfangreiche Änderungen des Quellcodes notwendig, die man nur durchführen kann, wenn der Quellcode zur Verfügung steht, neu kompiliert werden kann und man ihn außerdem ausreichend verstanden hat. Aber selbst dann ist der Aufwand in der Praxis meist zu hoch.

Eine solche Übergeneralisierung ist auch beim PIKS-Standard [PIKS94, PRATT95] zu beobachten. Verständlichkeit und Akzeptanz dieses Standards leiden darunter, daß alle Datenstrukturen extrem allgemein formuliert wurden, damit der Standard für alle Eventualitäten gewappnet ist. Die Definition eines 5-dimensionalen Bilddatentyps ist ein typisches Beispiel für den bottleneck-Ansatz, bei dem man einen Universalbaustein definiert, um zu vermeiden, daß eine Abstraktionsachse mehrere Alternativen bieten muß. Doch dadurch wird das Problem nur verlagert – komplexe Algorithmenimplementationen und umständliche Benutzung sind die Folgen. Es stellt sich die Frage, ob man hohe Flexibilität und Wiederverwendbarkeit nicht anders erreichen kann als durch das Überladen der Bausteine mit Fähigkeiten für jede Besonderheit eines jeden denkbaren Anwendungsfalls.

Die allgemeine Diskussion in Abschnitt 2.1 hat Wege in diese Richtung angedeutet: wir müssen einfache Grundbausteine definieren, die je nach Kontext immer wieder neu kombiniert werden können. Für jeden Anwendungsfall wählt der Entwickler angemessene Bausteine aus, deren Kombination die notwendige Systemfunktionalität bereitstellt. Dazu sind jedoch Bausteine erforderlich, die voneinander unabhängig sind und sich weitgehend automatisch integrieren lassen. Insbesondere müssen wir abstrakte *Algorithmen definieren, die von konkreten Datenstrukturen unabhängig* sind. Wir werden im folgenden anhand einer genaueren Analyse der grundlegenden Punktoperationen zeigen, daß die bisherigen Ansätze für Computer Vision Software dies nicht leisten. Die bestehenden Systeme sind entweder inflexibel oder

[9] Das Konzept der multi-methods wird z.B. von den Programmiersprachen CLOS [PAEPCKE93] und Dylan [SHALIT96] direkt unterstützt.

die Flexibilität mußte durch einen sehr großen Programmieraufwand (exponentielle Codeexplosion aufgrund des Problems des kartesischen Produkts) erkauft werden.

Betrachten wir als Beispiel die einfache unäre Punktoperation, einen Algorithmus, der mittels einer unären Funktion den Wert jedes Pixels[10] transformiert. Wenn wir (der Einfachheit halber) die alten Werte durch die neuen überschreiben, können wir dies durch folgenden Pseudocode beschreiben:

Algorithmus unäre Punkttransformation:

0) gegeben seien ein Bild `b` und eine unäre Funktion `f`
1) für jedes Pixel `pb` aus `b`:
   wende `f` auf `pb` an und schreibe das Ergebnis in `pb` zurück

Dieser Pseudocode könnte als Zwischenergebnis einer top-down-Zerlegung im Sinne der von Niklaus Wirth entwickelten Methode der funktionalen Zerlegung durch schrittweise Verfeinerung entstanden sein [WIRTH71]. Bei dieser Entwurfsmethode wird die zu lösende Aufgabe zunächst in groben Zügen skizziert und danach schrittweise in immer feinere Arbeitsschritte zerlegt, bis am Ende der Zerlegung ausführbare Instruktionen der Programmiersprache stehen.

Der obige Pseudocode beschreibt eine bereits sehr feinkörnige Funktionalität, aber die Anforderungen an die darunterliegenden Bausteine sind immer noch ganz abstrakt spezifiziert: es muß ein Bild geben, auf dessen Pixel wir in einer bestimmten Reihenfolge zugreifen können, sowie eine Funktion, die auf jedes Pixel angewendet werden kann. Wenn wir jedoch die Zerlegung weiter verfeinern und schließlich in einer prozeduralen Sprache wie C implementieren wollen, müssen wir uns für eine konkrete Datenstruktur und eine konkrete Funktion entscheiden, denn der Algorithmus muß die Funktion aufrufen und auf die Pixel zugreifen. Die einfachste Wahl wäre die folgende Datenstruktur: [11]

```
typedef unsigned char byte;  // wir werden ´byte´ im folgenden weiterbenutzen

struct SimpleByteImage {
    int width, height;
    byte * data;
};
```

[10] Zur Terminologie: Jedes Pixel repräsentiert einen Abtastpunkt einer 2-dimensionalen kontinuierlichen Bildfunktion (z.B. der Intensität). Anschaulich gesprochen, repräsentiert ein Pixel Eigenschaften eines kleinen Gebiets um den jeweiligen Gitterpunkt. Pixel sind die Elemente einer Bilddatenstruktur.

[11] Alle Beispiele sind in C++ implementiert. C++ ist eine *Multiparadigmen-Sprache*, die in gleicher Weise prozedurale, objekt-orientierte und generische Programmierung unterstützt. Wir können die Beispiele zu den verschiedenen Paradigmen somit in einer einheitlichen Programmiersprache formulieren, was den Vergleich vereinfacht.

Außerdem müssen wir uns für eine bestimmte Operation entscheiden, z.B. die Multiplikation der Pixelwerte mit einer Konstanten (Skalierung). Dann können wir eine Funktion implementieren, die genau diese Variante des Algorithmus realisiert:

```
void scaleImage1(struct SimpleByteImage * image, int scale)
{
    int i;
    for(i=0; i < image->width * image->height; ++i)
    {
        image->data[i] = scale * image->data[i];
    }
}
```

Man erkennt, daß der Algorithmus bereits sehr detaillierte Angaben über das Bild machen muß: er bestimmt, daß ein Bild ein `struct` vom Typ `SimpleByteImage` ist, welches die Bildgröße in Feldern mit den Namen `width` und `height` sowie die Pixeldaten in einem eindimensionalen Array mit Namen `data` speichert. Außerdem ist die eigentliche Punktoperation (die Skalierung der Pixelwerte) "fest verdrahtet". Für die anderen benötigten Operationen (z.B. Addieren einer Konstanten, Berechnen der Wurzel oder des Logarithmus usw.) müssen jeweils analoge Funktionen implementiert werden.

Damit macht der Algorithmus wesentlich genauere Angaben über die von ihm benutzten Bausteine als nötig. Die Abstraktion der ursprünglichen Beschreibung ist verloren gegangen. Beispielsweise kann der Algorithmus in der vorliegenden Form nur den Pixeltyp `byte` verarbeiten, während es sich im Verlauf der weiteren Systementwicklung wahrscheinlich als notwendig erweisen wird, auch andere Pixeltypen zu unterstützen. Da die vorliegende Implementation dies nicht erlaubt, müssen wir die obigen Definitionen für alle gewünschten Pixeltypen wiederholen.

Wir definieren also `SimpleIntImage`, `SimpleFloatImage`, `SimpleRGBImage` usw. und reimplementieren die Funktion `scaleImage1()` und alle übrigen Funktionen für alle diese Typen. Alternativ können wir einen Bilddatentyp `UniversalImage` einführen, der ein Datenfeld enthält, das den gerade aktuellen Pixeltyp angibt. Die Algorithmen müssen dann in einer umfangreichen `case`-Anweisung diesen Typ erfragen und die Daten entsprechend interpretieren. Gegenüber separaten Bilddatentypen für jeden Pixeltyp hat die letztere Variante den Vorteil, daß wir den Pixeltyp leichter zur Laufzeit konvertieren können. Die Algorithmen hingegen werden komplexer, da eine falsche Interpretation des Pixeltyps leicht zum Programmabsturz führt. Beiden Varianten gemeinsam ist jedoch eine sehr große Redundanz des Quellcodes.

Wir sehen hier ein erstes Beispiel für das Problem des kartesischen Produkts: der Algorithmus definiert die unabhängigen Abstraktionsachsen "Pixeltyp" und "Funktion", und die Anzahl der notwendigen Implementationen steigt quadratisch mit der Anzahl der Grundbausteine an ($p \times f_1$ Varianten für p Pixeltypen und f_1 unäre Funktionen). Da alle Implementationen vom Wesen her gleich sind, besitzt der Code eine hohe Redundanz. Das Problem verschärft sich noch weiter, wenn wir Operatio-

nen mit mehreren Quellbildern (wie die binären Punktoperationen) implementieren und dabei zulassen wollen, daß die Quellbilder unterschiedliche Pixeltypen haben: dann wächst die Anzahl bereits mit p^2 oder einer noch höheren Potenz.

Daß dies ein ernstes Problem ist, zeigt sich sehr schnell, wenn sich die Annahmen ändern, die der Algorithmus über seine Datenstrukturen macht. Stellen wir uns z.B. vor, wir müssen im weiteren Verlauf der Systementwicklung den Bilddatentyp dahingehend ändern, daß er die Daten als zweidimensionales Array speichert. Dies zwingt uns, sämtliche $p \times f_1$ Varianten des Transformationsalgorithmus zu ändern, z.B.:

```
void scaleImage2(struct NewSimpleByteImage * image, int scale)
{
    int x, y;
    for(y=0; y < image->height; ++y)
    {
        for(x=0; x < image->width; ++x)
        {
            image->data[y][x] = scale * image->data[y][x];
        }
    }
}
```

Selbst bei einem so einfachen Algorithmus wie der unären Punkttransformation entsteht durch eine triviale Änderungen der Datenstruktur ein hoher Arbeitsaufwand, weil viele Varianten dieses Algorithmus in gleicher Weise geändert werden müssen. Zudem schleichen sich während der Anpassungsarbeiten sehr leicht Fehler ein, z.B. kann man leicht die Reihenfolge der Indizes `x` und `y` verwechseln. Bei komplizierten Algorithmen wird der Aufwand zum Ausführen und Testen der Änderungen schnell so groß, daß bestimmte Änderungen in der Praxis gar nicht mehr angegangen werden.

Es hat nicht an Versuchen gefehlt, dieses Problem zu lösen oder zumindest zu entschärfen. Im obigen Fall wäre die beste Lösung die Einführung von Zugriffsfunktionen `getPixel()` und `setPixel()`, so daß die eigentliche Datenorganisation gekapselt wird. Konzeptionell entspricht dies der Definition einer neuen Abstraktionsachse "Datenorganisation". Mit der objekt-orientierten Programmierung sind Zugriffsfunktionen zur Selbstverständlichkeit geworden, aber unter den prozeduralen Systemen wird diese Möglichkeit nur von Candela und Tina, und hier ausschließlich für Bilddatenstrukturen, angeboten.

Durch Einführung einer neuen Achse haben wir jedoch die Probleme mit den beiden bereits vorhandenen Achsen nicht gelöst: die Anzahl der notwendigen Algorithmenimplementationen entspricht immer noch dem Produkt $p \times f_1$. Um diese Zahl zu reduzieren, verwenden die existierenden Systeme ganz unterschiedliche Strategien. Einige setzen auf der Achse "Funktionen" an. Tina und TargetJr bieten

beispielsweise einen allgemeinen Punktoperator an, dem die auszuführende Funktion als Funktionszeiger übergeben wird:

```
void transformImage1(struct FloatImage * image, float (*f)(float));
```

Auf diese Weise können wir sofort auch komplexe, nutzerdefinierte Funktionen auf ein Bild anwenden, ohne dafür die Schleife neu programmieren zu müssen. Allerdings hat das Verfahren Grenzen. Erstens kann eine C Funktion keinen inneren Zustand besitzen, so daß wir z.B. die Funktion `scaleImage` nicht auf diese Weise implementieren können, weil sie einen zusätzlichen Parameter benötigt. Zweitens führt ein Funktionsaufruf immer zu einer merklichen Geschwindigkeitseinbuße.

Das erste Problem wird in Candela dadurch beseitigt, daß statt einer einfachen Funktion ein *Interpreter* für beliebige Ausdrücke zur Verfügung gestellt wird. In Candela kann man schreiben:

```
/* Umwandlung eines RGB-Bildes in ein Grauwertbild */
grayimage = pointop("(r+g+b)/3", rgbimage, 3 /* colors */, flags);
```

Der als String übergebene Ausdruck `"(r+g+b)/3"`, der arithmetische und algebraische Funktionen enthalten kann, wird zur Laufzeit in einen internen Bytecode umgewandelt und ausgeführt. Damit ist dieser Ansatz flexibler als derjenige auf der Basis von Funktionszeigern, aber er ist auch noch langsamer.

Das System IPRS verwendet dagegen einen Ansatz, der bereits zur Übersetzungszeit wirkt, nämlich einen Codegenerator. Hier wird in einem einfachen Konfigurationsfile beschrieben, welche Operation für welchen Pixeltyp ausgeführt werden soll:

```
TYPES
  byte
  float
  int
  rgb
FUNCTIONS
  *dp = sqrt(*sp);
  *dp = sqrt(*sp);
  *dp = sqrt(*sp);
  dp->r = sqrt(sp->r); dp->g = sqrt(sp->g); dp->b = sqrt(sp->b);
```

Der Codegenerator erzeugt für jeden angegebenen Pixeltyp eine Funktion und setzt die angegebenen Ausdrücke in die Schleife ein. Da dies im Quellcode geschieht, ist die generierte Funktion zur Laufzeit nicht langsamer als eine per Hand programmierte Version. Allerdings ist der IPRS Codegenerator sehr speziell: über die Erzeugung von Punktoperationen hinaus ist er nicht anwendbar.

Somit bieten zwar alle drei Ansätze (Funktionszeiger, Interpreter, Codegenerator) interessante Ideen zur Lösung des Problems des kartesischen Produkts, aber keiner

kann voll befriedigen. Wir werden in Kapitel 4 zeigen, wie sich mit der generischen Programmierung wesentliche Vorteile aller drei Ansätze verbinden lassen.

Neben der Achse "Funktion" kann man zur Verringerung der Redundanz auch auf der Achse "Bilddatentyp" ansetzen. In existierenden Systemen läuft dies stets auf einen bottleneck-Ansatz hinaus: man wählt einen repräsentativen Pixeltyp (meist `float`) und implementiert jede Funktion nur noch für diesen Typ. Alle anderen Pixeltypen müssen vorher in den bottleneck-Typ umgewandelt werden. Dies geht am einfachsten mit dem `UniversalImage`:

```
void scaleImage3(struct UniversalImage * image, float scale)
{
    int i;

    // Umwandlung des Bilddatentyps, falls nicht bereits FLOAT
    if(image->pixeltype != FLOAT) convertImage(image, FLOAT);

    for(i=0; i < image->width * image->height; ++i)
    {
        image->data[i] = scale * image->data[i];
    }
}
```

Auf diese Weise benötigen wir für jeden Algorithmus nur noch eine Funktion. Allerdings kostet jede Umwandlung (und eventuelle Rückwandlung) zusätzliche Zeit. Auch ist die Verwendung des Typs `float` ineffizient, wenn in einem bestimmten Kontext `byte` ausgereicht hätte. Zudem kann `float` nicht für alle Pixeltypen stehen: bereits bei einer Konvertierung von `double` geht Information verloren, und bei `complex` gelten sogar ganz andere Rechenregeln. Viele Systeme verwenden deshalb einen gemischten Ansatz: einfache, häufig benötigte Funktionen werden explizit für alle Typen programmiert, und die übrigen nutzen den bottleneck-Ansatz.

In den letzten Jahren hat sich die *objekt-orientierte Programmierung* allgemein durchgesetzt. Viele Vertreter dieser Methode empfehlen, Algorithmen und Datenstrukturen als selbständige Bausteine aufzugeben und beide in Objekten zu vereinigen (vergl. z.B. [AUPPERLE97]). Man könnte also vermuten, daß die Implementation der Punktoperationen als member function des Bilddatentyps das Problem des kartesischen Produkts lösen könnte. Die Funktion `scaleImage` können wir als member function wie folgt implementieren:

```
class ByteImageWithMemberFunctions
{
    int width, height;
    byte * data;

  public:
```

```
    void scale(float scale)
    {
        for(int i=0; i<width*height; ++i)   data[i] = scale * data[i];
    }

    // weitere member functions, Konstruktoren, Destruktor usw.
};

class FloatImageWithMemberFunctions { ... }; // analog für andere Pixeltypen
```

Wenn wir diesen Code vom Standpunkt des Aufwands bei der Implementierung kritisch anschauen, erkennen wir, daß er gegenüber der Lösung der prozeduralen Programmierung kaum Vorteile bietet: die Implementierung der member function hängt direkt von den Datenattributen des Objekts ab, genau wie vorher die Funktionen von den Datenstrukturen abhängig waren. Die Redundanz hat sich nicht verringert, denn nach wie vor müssen sämtliche Punktoperationen für sämtliche Bildklassen erneut implementiert werden.

Zudem gibt es hunderte verschiedene Algorithmen, und es ist nicht klar, welche wir als member functions anbieten sollten. Bieten wir eine möglichst vollständige Auswahl, wird kaum eine Anwendung die Möglichkeiten vollständig nutzen. Das heißt, die Bilddatentypen werden wesentlich komplexer als nötig sein, da sie unbenutzte Funktionalität enthalten. Wir finden deshalb in objekt-orientierten Computer Vision-Systemen kaum Algorithmen, die als member functions einer Datenstruktur implementiert sind.

Bevor wir weitere Lösungsideen analysieren, wollen wir versuchen, die Ursache der Schwierigkeiten besser zu verstehen. Wir beobachten hier ein Phänomen, das Robert Martin als *Abhängigkeitsparadoxon der funktionalen Zerlegung* [MARTIN95] beschreibt: obwohl konzeptionell die Datenstrukturen von den Anforderungen der Algorithmen (also die tieferliegenden Bausteine von den Anforderungen der höheren) abhängen, kehrt sich diese Abhängigkeit im Code um: wenn man die höheren Bausteine implementieren will, muß man Details festlegen, für die diese Bausteine konzeptionell gar nicht zuständig sind. Sie werden damit unnötigerweise von solchen Details abhängig, und bei der Implementation der tieferliegenden Bausteine ist man gezwungen, den „Vorgaben von oben“ zu folgen. Das heißt, die tieferliegenden Bausteine lassen sich nur noch mit sehr großem Aufwand ändern. Wirth selbst hat bereits auf diese Schwierigkeit aufmerksam gemacht, wenn er feststellt, daß bei der top-down-Zerlegung bestimmte Entscheidungen bereits getroffen werden müssen, bevor die eigentlich notwendigen Detailinformationen vorliegen (siehe [WIRTH71]). In traditionellen prozeduralen Programmiersprachen kann man diese Entscheidungen nicht verschieben, weil man sonst mit der Zerlegung nicht weiter fortfahren könnte.

Dies führt zu der paradoxen Situation, daß sich allgemeine Bausteine, die grundlegende Lösungsstrategien festlegen, leicht ändern lassen, während die Modifikation von Implementationsdetails sehr schwierig sein kann. Erforderlich ist genau das

Gegenteil: auch wenn wir das allgemeine Verfahren bereits festgelegt haben, sollten die Details noch in weiten Grenzen variabel sein. Die dazu notwendige Abstraktion geht allerdings bei einer traditionellen Implementation verloren.

Heute sind jedoch Methoden verfügbar, mit denen sich dies vermeiden läßt: durch die Definition eines required interface können Bausteine ihre Anforderungen exakt und dennoch abstrakt spezifizieren. Die Bausteine der höheren Schichten werden dadurch nicht von denen der tieferen Schichten abhängig, denn das abstrakte required interface kann mit vielen verschiedenen Servern verbunden werden. Dies gilt auch und gerade für Algorithmen. Wir wollen dies als Spezialfall des Prinzips der top-down-Abstraktion explizit festhalten:

Abstrakte Algorithmenimplementation: Ein abstrakter Algorithmus sollte so implementiert werden, daß er in Form eines formalen und abstrakten required interface minimale Anforderungen an akzeptable Datenstrukturen und verwendete Unterfunktionen beschreibt.

Betrachten wir zur Verdeutlichung der abstrakten Algorithmenimplementation und damit zur Motivation der nachfolgenden Kapitel eine extrem einfache Funktion, die *Minimum-Funktion*. Eine abstrakte Beschreibung dieser Funktion im Rahmen einer top-down-Zerlegung könnte etwa lauten:

Funktion Minimum:

0) gegeben: zwei Werte eines Typs, auf dem eine Ordnungsrelation definiert ist
1) liefere als Ergebnis den kleineren der beiden Werte

Wenn wir diese Funktion in einer prozeduralen Sprache wie C implementieren, müssen wir den Typ der Funktionswerte festlegen. Wählen wir beispielsweise `int`, so erhalten wir die folgende Version:

```
int min1(int v1, int v2) {
    if (v1 < v2) return v1; else return v2;
}
```

Die abstrakte Forderung, daß eine Ordnungsrelation definiert ist, wird durch die Wahl von `int` implizit erfüllt, denn die Operation '<' ist für `int`-Argumente Bestandteil der Sprache C. Leider geht jedoch auf diese Weise die Abstraktion völlig verloren, denn die abstrakte Definition von `Minimum` galt für beliebige Typen, nicht nur für `int`.

Eine flexiblere Variante können wir in C++ mit Hilfe einer abstrakten Basisklasse `Comparable` angeben:

```
class Comparable {
  public:
    virtual bool operator<(Comparable const &) const = 0;
};
```

```
Comparable & min2(Comparable & v1, Comparable & v2) {
    if (v1 < v2) return v1; else return v2;
}
```

Die Funktion `min2` können wir für alle Klassen benutzen, die von `Comparable` abgeleitet sind und den `operator<` entsprechend implementieren. Damit kommen wir der abstrakten Formulierung von `Minimum` bereits wesentlich näher, denn es kann beliebig viele solche Klassen geben. Allerdings gibt es immer noch Einschränkungen: erstens können wir einer Klasse, deren Quellcode wir nicht ändern können oder dürfen, nicht nachträglich die neue Basisklasse `Comparable` hinzufügen, und zweitens sind die eingebauten C++ Typen (wie z.B. `int`) keine Klassen.
Ein noch flexiblere Lösung bietet der template-Mechanismus von C++:

```
template <class Ordered>
Ordered & min3(Ordered & v1, Ordered & v2) {
    if (v1 < v2) return v1; else return v2;
}
```

Diese Variante kann in der Tat für sämtliche Typen, für die der Vergleichsoperator '<' definiert ist, benutzt werden, darunter auch für `int` und für `Comparable`. Damit entspricht sie beinahe der abstrakten Beschreibung von `Minimum`. Es gibt allerdings noch einen kleinen Unterschied: die abstrakte Beschreibung spricht von *einer* (beliebigen) Ordnung, wir haben uns jedoch auf die von `operator<` bestimmte Ordnung festgelegt. Es kann aber für denselben Datentyp in unterschiedlichen Kontexten mehrere sinnvolle Ordnungen geben. Deshalb können wir die Funktion `min` weiter verallgemeinern, indem wir ihr als weiteres Argument einen (austauschbaren) *Funktor* übergeben, der die gewünschte Ordnungsrelation realisiert:

```
template <class T, class OrderingRelation>
T & min4(T & v1, T & v2, OrderingRelation lessThan) {
    if (lessThan(v1, v2)) return v1; else return v2;
}
```

Aus diesen Beispielen sollte deutlich geworden sein, daß abstrakte Anforderungen, die sich aus einer funktionalen Zerlegung ergeben, auf ganz unterschiedliche Weise in eine Implementation übersetzt werden können. Die entscheidende Frage ist: *Wie sichern wir, daß abstrakte Anforderungen bei der Implementation erhalten bleiben*? Daß wir uns auf einer tiefen Schicht der Zerlegung befinden, rechtfertigt keineswegs die vorzeitige Festlegung von Implementationentscheidungen. Auch low-level-Anforderungen können durchaus abstrakt sein – man betrachte nur die aus der Mathematik bekannte axiomatische Definition der Additionsoperation, die den Typ der zu addierenden Objekte völlig offen läßt.

Wir wollen in einem weiteren Beispiel den Algorithmus `scaleImage` mit Hilfe objekt-orientierter Konzepte abstrahieren. Das heißt, wir werden das required

interface durch abstrakte Basisklassen repräsentieren. Für die Implementation von scaleImage benötigen wir zwei Basisklassen: AbstractPixel und AbstractImage:

```
class AbstractPixel {
  public:
    virtual void scale(float) = 0;
    ... (weitere Funktionen)
};

class AbstractImage {
  public:
    virtual int width() const = 0;
    virtual int height() const = 0;
    virtual AbstractPixel & getPixel(int x, int y) = 0;
};
```

Mit Hilfe dieses required interface können wir die objekt-orientierte Version von scaleImage wie folgt implementieren:

```
void scaleImage4(AbstractImage * image, float scale)
{
    int x, y;
    for(y=0; y < image->height(); ++y)
    {
        for(x=0; x < image->width(); ++x)
        {
            image->getPixel(x, y).scale(scale);
        }
    }
}
```

Dieser Algorithmus stellt wesentlich geringere Anforderungen an die Datenstruktur als die bisherigen Varianten der Skalierungsfunktion: die tatsächliche Bildklasse muß von der Basisklasse AbstractImage und die Pixelklasse von AbstraktPixel abgeleitet sein. Beide Klassen müssen die angegebenen virtuellen Funktionen implementieren. Somit ist diese Lösung sehr viel flexibler als unsere bisherigen Varianten. Dennoch wird sie in keinem der objekt-orientierten Computer Vision-Systeme benutzt, denn sie ist viel zu ineffizient.

Dies liegt daran, daß in jedem Schleifendurchlauf mehrere virtuelle Funktionen aufgerufen werden müssen. Durch jeden zusätzlichen Funktionsaufruf geht zunächst direkt Zeit verloren, darüber hinaus werden aber auch bestimmte Compileroptimierungen verhindert, die für hohe Rechengeschwindigkeit wesentlich sind. In obiger Funktion betrifft dies vor allem die Adreßrechnung beim Zugriff auf die Bilddaten: bei direktem Zugriff auf die Daten, wie in scaleImage2, kann der Compiler den mit der Variable y verbundenen Teil der Adreßrechnung aus der inneren

Schleife entfernen, weil y dort konstant ist. In `scaleImage4` wird diese Optimierung durch die virtuelle Funktion verhindert, denn der Compiler hat keine Kenntnis davon, was sich hinter dem Funktionsaufruf verbirgt. Der resultierende Geschwindigkeitsverlust ist erheblich und liegt in der Größenordnung von 400-500% (vergleiche den Geschwindigkeitstest in Abschnitt 4.11). Wäre die Funktion hingegen nicht virtuell, könnte sie `inline` expandiert und dadurch normal optimiert werden. Kapselung und Optimierung widersprechen einander nicht prinzipiell, nur dynamische Funktionsaufrufe (wie z.B. virtuelle Funktionen) wirken sich stets negativ auf die Geschwindigkeit aus.

Bei binären Funktionen wird der Geschwindigkeitsverlust noch größer. Nehmen wir an, daß wir bei der Definition einer Funktion `addImages` einen Pixeltyp mit einer virtuellen Funktion `add` angefordert haben:

```
virtual AbstractPixel::add(AbstractPixel const &);
```

Aufgrund des Liskovschen Substitutionsprinzips [LISKOV88] müssen alle von `AbstractPixel` abgeleiteten Klassen die Funktion `add()` mit demselben abstrakten Argumenttyp ´`AbstractPixel const &`´ implementieren. In Wirklichkeit können wir jedoch nicht beliebige Ausprägungen von abstrakten Pixeln zueinander addieren. Beispielsweise wäre die Addition eines RGB-Werts zu einem Grauwert undefiniert. Wir müssen daher zur Laufzeit prüfen, ob die Argumente der Operation `add()` miteinander verträglich sind. Das kostet wiederum zusätzliche Zeit.

Um den hohen Geschwindigkeitsverlust der „reinen“ objekt-orientierten Konzepte zu vermeiden, geht man in der Computer Vision-Praxis Kompromisse ein. Eine Möglichkeit ist die Definition von spezielleren Zugriffsfunktionen. Die Klasse `IUE_scalar_image` des IUE implementiert beispielsweise je 6 verschiedene Funktionen, um einzelne Pixel zu lesen bzw. zu schreiben: als `byte`, `int` und `double` sowie als Arrays mit einem Element derselben Typen (die Array-Funktionen werden benötigt, um mit Multispektralbildern kompatibel zu sein – die Zahl der Elemente des Arrays entspricht der Anzahl der spektralen Kanäle.) Es handelt sich dabei aber nach wie vor um virtuelle Funktionen, so daß die Effizienz immer noch nicht befriedigt.

Sowohl IUE als auch TargetJr empfehlen deshalb die Verwendung von Pufferobjekten für den Zugriff auf die Bilddaten. Wir wollen dies am Beispiel der Klasse `BufferXY` von TargetJr kurz erläutern. Wenn ein Algorithmus auf die Pixel eines Bildes zugreifen will, fordert er von der Bilddatenstruktur ein Objekt der Klasse `BufferXY` an. Die Bilddatenstruktur kopiert ihre Daten in dieses Objekt. Der Algorithmus kann über schnelle Zugriffsfunktionen direkt auf die kopierten Pixeldaten zugreifen. Nach Beendigung der Operation werden die geänderten Daten aus dem Puffer in das Bild zurückgeschrieben. Die Klasse `BufferXY` ist keine abstrakte Basisklasse und auch nicht von einer solchen abgeleitet, sie ähnelt eher den Bild-

datenstrukturen der prozeduralen Systeme. Wir kommen deshalb zu folgender Feststellung:

> *In den analysierten objekt-orientierten* Computer Vision*-Systemen ist die Schnittstelle zwischen Algorithmen und Bilddatenstrukturen nicht objekt-orientiert definiert.*

Wir gründen diese Feststellung auf die übliche Definition der objekt-orientierten Programmierung, die ausdrücklich die Verwendung von Vererbung und Polymorphie (virtuellen Funktionen) fordert. Werden zwar Objekte verwendet, aber ohne Vererbung und Polymorphie, spricht man von *objekt-basierter Programmierung*. Abgesehen von der besseren Kapselung der Datenstrukturen unterliegen objekt-basierte Ansätze den gleichen Problemen wie prozedurale.

Wir halten fest, daß die objekt-orientierte Programmierung im Kontext der Punktoperationen auf einer zu hohen Ebene ansetzt: sie bietet (teure) Laufzeitflexibilität auf einer Ebene, wo Geschwindigkeit entscheidend ist. Wir benötigen einen weiteren Flexibilitätsmechanismus, der unterhalb der objekt-orientierten Programmierung ansetzt und der genauso effizient ist wie eine herkömmlich implementierte, inflexible Lösung. Die *generische Programmierung* bietet hierfür geeignete Mittel an. Dies haben auch die Entwickler von TargetJr und IUE erkannt: die Klasse `BufferXY` wurde inzwischen zu einem template `ImageBuffer` weiterentwickelt, das für beliebige Pixeltypen parametrisiert ist. Ein analoges Konzept findet sich mit den verschiedenen templates der Familie `image_accessor` im IUE. Darüber hinaus gibt es erste Versuche, auch Algorithmen generisch zu implementieren. Es fehlt aber bisher ein sorgfältig ausgearbeitetes Konzept, wie die generische Programmierung am effektivsten in der Computer Vision eingesetzt werden soll. In der vorliegende Arbeit wird erstmalig ein solches Konzept entwickelt und seine Tragfähigkeit exemplarisch nachgewiesen.

2.3 Zusammenfassung des Kapitels

Wir haben in diesem Kapitel folgendes gezeigt:

- Für flexible Software ist es notwendig, ein System in unabhängige Bausteine zu zerlegen, die hohe Kohäsion und geringe Kopplung aufweisen. Insbesondere dürfen Algorithmen nicht von Datenstrukturen abhängen.
- Die existierenden Computer Vision-Systeme erfüllen diese Forderung nicht. Flexibilität kann deshalb in diesen Systemen nur mit großem Aufwand, durch explizite Implementation aller möglichen Varianten, erreicht werden (Problem des kartesischen Produkts). Änderungen an den Datenstrukturen ziehen Änderungen in allen sie benutzenden Algorithmen nach sich.
- Zur Überwindung dieser Probleme ist es notwendig, bessere Abstraktionen und damit bessere Schnittstellen zu definieren. Angesichts der großen Bedeutung von Algorithmen für Computer Vision kommt insbesondere der top-down Abstraktion große Bedeutung zu.
- Die objekt-orientierte Programmierung bietet im Prinzip geeignete Mechanismen hierfür an. Diese sind jedoch im Vergleich zu prozeduralen Lösungen zu ineffizient, so daß sie in objekt-orientierten Computer Vision-Systemen nicht genutzt werden können.
- Zur Überwindung dieser Probleme ist es notwendig, bessere Abstraktionen und damit bessere Schnittstellen zu definieren. Angesichts der großen Bedeutung von Algorithmen für Computer Vision kommt insbesondere der top-down Abstraktion große Bedeutung zu.

Kapitel 3

Generische Programmierung

3.1 Grundideen

Wie wir im weiteren Verlauf der Arbeit zeigen werden, bietet die *generische Programmierung* zur Zeit die besten Möglichkeiten, im Rahmen der Computer Vision die Anforderungen an flexible Software zu erfüllen. Der Ausdruck „generische Programmierung“ sowie die dahinter stehende Methode wurden im Verlauf der letzten 15 Jahre von Alexander Stepanov und David Musser entwickelt [MUSSTEP89, MUSSTEP94]. Ihr Ziel bestand darin, die Software- und insbesondere Algorithmenentwicklung auf eine solidere theoretische Basis nach dem Vorbild der Axiomatisierung der Mathematik zu stellen. Gleichzeitig suchten sie nach Wegen, ihre Ideen in einer Programmiersprache adäquat, also ohne Verlust der Abstraktion, auszudrücken. Zur Zeit halten sie den template-Mechanismus der Programmiersprache C++ für die beste existierende Technologie zur Implementierung ihrer Konzepte.

Die Vorteile der generischen Programmierung haben auch das C++ Standardisierungskomitee überzeugt. Die von Stepanov und Musser entwickelte *Standard Template Library* (*STL*) wurde in verbesserter Form in den C++ Standard aufgenommen und bildet den Kern der Standardbibliothek dieser Sprache [C++98, AUSTERN98]. Dadurch wurden die Erkenntnisse der generischen Programmierung einem breiten Nutzerkreis zugänglich und bekannt.

Interessant ist nun die Frage, ob mit Hilfe dieser überzeugenden Ideen bekannte Defizite bei der Softwareerstellung in verschiedenen wissenschaftlichen Anwendungsgebieten, wie wir sie in Abschnitt 2.2 für Computer Vision herausgearbeitet

haben, überwunden werden können. Dadurch ließe sich einerseits der Aufwand für Programmierung und Fehlersuche stark verringern, wodurch Ressourcen für die eigentliche Algorithmenentwicklung freigesetzt würden. Andererseits würden dadurch Voraussetzungen geschaffen, eine große Zahl von Lösungsideen in kurzer Zeit zu implementieren und zu evaluieren (rapid prototyping). Darüber hinaus besteht die Hoffnung, daß ein Kern generischer Computer Vision-Bausteine geschaffen werden kann, die sorgfältig nach international akzeptierten Kriterien überprüft und bewertet worden sind (vgl. [VIERGEVER+99]). Eine solche Sammlung wäre sehr hilfreich bei der Überführung von Computer Vision-Methoden aus der Forschung in die industrielle Praxis.

Eine der wichtigsten Säulen der generischen Programmierung ist die starke Betonung der algorithmischen Abstraktion, die einen gleichberechtigten (oder sogar wichtigeren) Platz neben den abstrakten Datentypen erhält. Die generische Programmierung entkoppelt Algorithmen mit Hilfe abstrakter required interfaces von den unterliegenden Datenstrukturen, d.h. sie definiert *abstrakte Algorithmen*. Stepanov und Musser bezeichnen dies als algorithm-oriented approach. In [MUSSTEP94, S. 624] erklären sie ihren Ansatz wie folgt:

> "Start with the most efficient known algorithms and data structures, identify container access operations [...] on which the algorithms depend, and abstract [...] those operations by determining the minimal behavior they must exhibit in order for the algorithms to perform a useful operation."

Komplementär zur Definition minimaler required interfaces wird analysiert, welche Zugriffsoperationen bei unterschiedlicher Datenorganisation effizient implementiert werden können. Danach werden die Anforderungen der Algorithmen und die Möglichkeiten der Datenstrukturen einander gegenübergestellt und die gemeinsamen Eigenschaften mit Hilfe von standardisierten *Konzepten* systematisiert. Der Begriff des *Konzepts* entstammt ursprünglich der Mathematik, wo er eine Gruppe von Axiomen bezeichnet. Eine Menge von gleichartigen Objekten, die die Axiome erfüllen, wird *Modell* des Konzepts genannt. Zu den bekanntesten Konzepten der Mathematik gehören die algebraischen Gruppen mit ihren Modellen *natürliche, rationale* und *reelle Zahlen*.

In der generischen Programmierung steht der Begriff *Konzept* für eine Liste genau definierter funktionaler und syntaktischer Anforderungen an einen Datentyp oder eine Funktion. Wir können die Anforderungen eines required interface ebenso wie die Angebote eines offered interface kompakt beschreiben, indem wir einfach auf ein entsprechendes Konzept verweisen. Konzepte sind Ergebnis einer Standardisierung, die Anforderungen und Angebote systematisiert und gegeneinander abgleicht.

Konzepte können auf mehrere Arten zueinander in Beziehung stehen. Einerseits bemüht man sich um unabhängige Konzepte, damit die Zahl unterschiedlicher Abstraktionsachsen gering bleibt. Die Konzepte „Iterator“ und „Funktor“ (siehe Abschnitt 3.3) repräsentieren beispielsweise orthogonale Abstraktionsachsen.

Zweitens kann ein Konzept ein anderes *spezialisieren*. Ein Modell dieses Konzepts ist damit automatisch auch Modell des allgemeinen Konzepts. Beispielsweise ist der Bidirectional Iterator eine Spezialisierung des Forward Iterator (siehe Abschnitt 3.3.3), denn er fügt zu den Anforderungen des letzteren die Möglichkeit der Rückwärtsiteration hinzu. Schließlich können Konzepte auch *Alternativen* auf der gleichen Abstraktionsachse beschreiben, wie es zum Beispiel bei den unterschiedlichen Ausprägungen einer Bilddatenstruktur mit verschiedenen Pixeltypen (`byte`, `int`, `float` usw.) der Fall ist.

Die Umsetzung der generischen Programmierung mit Hilfe von C++ templates hat verschiedene Vorteile. Wenn wir uns templates entsprechend unserer Diskussion von Flexibilität in Abschnitt 2.1 anschauen, finden wir die folgenden wichtigen Eigenschaften:

- Templates definieren ihre Anforderungen als abstraktes required interface.
- Das required interface hat die Form einer syntaktischen Spezifikation, zulässige Server müssen also nicht von einer bestimmten Basisklasse abgeleitet sein.
- Template-Bausteine sind konfigurierbar, das heißt, sie passen sich daran an, durch welchen Baustein das required interface implementiert wird.
- Template-Bausteine werden automatisch generiert, wenn sie benötigt werden. Das Problem des kartesischen Produkts kann dadurch gelöst werden.
- Die Konfiguration und Instanziierung von templates erfolgt weitgehend automatisch, da der Compiler auf ein leistungsfähiges Verfahren zur Auswahl der richtigen Implementation und zusätzliche Metainformationen (z.B. traits-Klassen, siehe Abschnitt 3.3.5) zurückgreifen kann.
- Die Flexibilitätsmechanismen wirken zur Kompilierungszeit, so daß Geschwindigkeitsverluste zur Laufzeit vermieden werden können.

Dem stehen zwei Nachteile gegenüber: erstens überprüft zwar der Compiler, ob ein Server das required interface seines Client erfüllt (zumindest syntaktisch), aber es gibt innerhalb der Sprache C++ kein Konstrukt, mit dem man das required interface explizit notieren kann. Das required interface muß also Bestandteil der Dokumentation sein. Zweitens ist die Beschränkung der Flexibilität auf Compilezeitmechanismen nicht immer ausreichend. Wenn wir außerdem Flexibilität zur Laufzeit benötigen, müssen wir die templates mit anderen Mechanismen, wie zum Beispiel virtuellen Funktionen, kombinieren. C++ hat den Vorteil, daß es solche Mechanismen ebenfalls bereitstellt, so daß wir für unterschiedliche Anforderungen die jeweils geeignete Vorgehensweise auswählen können.

Insgesamt hat sich im Verlauf der Entwicklung der in dieser Arbeit vorgestellten Ansätze gezeigt, daß die Vorteile gegenüber den Nachteilen weit überwiegen.

3.2 Fundamentale Konzepte

Die C++-Standardbibliothek definiert eine Reihe fundamentaler Konzepte, die man auch in der Computer Vision häufig benötigt. Wir wollen hier die entsprechenden Definitionen aus [AUSTERN98] wiedergeben. Die sehr strenge, mathematische Definition der folgenden Konzepte sichert dabei, daß Bausteine, die die entsprechenden Operationen nutzen, absolut verläßliche Annahmen über ihre Semantik machen können.

Es ist zu bemerken, daß die Konzepte und ihre Anforderungen in Form von ausführbaren Ausdrücken der Sprache C++ spezifiziert werden. Diese Ausdrücke demonstrieren, wie man Objekte *benutzen* kann, die Modelle des betreffenden Konzepts sind. Im allgemeinen gibt es in C++ mehrere Möglichkeiten, die in den Ausdrücken verwendeten Funktionen zu implementieren. Beispielsweise kann der Vergleichsoperator `operator<` sowohl als member function als auch als freie Funktion implementiert werden. Durch Beschränkung auf die Angabe ausführbarer Ausdrücke vermeidet man, daß die Konzeptdefinition unnötigerweise eine bestimmte Implementation erzwingt.

Die eingebauten numerischen Typen von C++ (also z.B. `int`, `float` usw.) sind übrigens Modelle jedes der folgenden fundamentalen Konzepte.

Assignable

Ein Typ `X` wird als Modell des Konzepts Assignable bezeichnet, wenn man Instanzen dieses Typs ineinander kopieren kann. In C++ wird dies durch die Forderung ausgedrückt, daß der Benutzer des Typs `X` den Kopierkonstruktor und die Zuweisungsfunktion aufrufen kann (`y` ist eine Instanz des Typs `X`):

```
X x(y);  // falls X Assignable ist, sind Kopierkonstruktor
x = y    // und Zuweisungsfunktion definiert
```

Default Constructible

Ein Typ `X` erfüllt die Anforderungen an das Konzept Default Constructible, wenn man Instanzen dieses Typs ohne Angabe von Parametern erzeugen kann. In C++ heißt dies, daß der Standardkonstruktor definiert sein muß:

```
X x;   // Erzeuge eine Variable des Typs X bzw.
X()    // ein namenloses Objekt des Typs X mittels Standardkonstruktor
```

Equality Comparable

Die Modelle des Konzepts Equality Comparable haben die Eigenschaft, daß man feststellen kann, ob zwei Objekte gleich sind oder nicht (x und y sind Instanzen des Typs X):

```
x == y // Gleichheit
x != y // Ungleichheit, äquivalent zu !(x == y)
```

Der Identitätsoperator muß außerdem die mathematischen Axiome einer Äquivalenzrelation erfüllen (wiederum in C++ Syntax ausgedrückt):

Identität: &x == &y impliziert x == y (identische Objekte sind äquivalent)

Reflexivität: x == x (x ist sich selbst äquivalent)

Symmetrie: x == y impliziert y == x (Argumente sind vertauschbar)

Transitivität: x == y und y == z implizieren x == z

Wenn ein Typ gleichzeitig Modell von Assignable ist, fordert man außerdem, daß die Zuweisung die Identität impliziert, also:

```
X x = y;  impliziert x == y
X x(y);   impliziert x == y
```

LessThan Comparable

Ein Typ X ist Modell der Kategorie LessThan Comparable, falls seine Instanzen die Vergleichsoperation operator< und ihre Varianten implementieren:

```
x < y  // kleiner als
x > y  // größer als, äquivalent zu y < x
x <= y // kleiner oder gleich, äquivalent zu !(y < x)
x >= y // größer oder gleich, äquivalent zu !(x < y)
```

Die Vergleichsoperation muß außerdem die mathematischen Axiome einer irreflexiven Halbordnung erfüllen:

Irreflexivität: x < x ist falsch

Asymmetrie: x < y impliziert !(y < x)

Transitivität: x < y und y < z implizieren x < z

Strict Weakly Comparable

Dieses Konzept ist eine Verschärfung von LessThan Comparable. Ein Typ ist Strict Weakly Comparable (*streng geordnet*), wenn die Ordnungsrelation "<" eine Äquiva-

lenzrelation impliziert [AUSTERN98]. Das heißt, zwei Werte x und y können als äquivalent betrachtet werden, wenn weder x < y noch y < x gelten. Die Äquivalenzrelation muß transitiv sein:

!(x<y) && !(y<x) && !(y<z) && !(z<y) impliziert !(x<z) && !(z<x)

Diese Äquivalenzrelation kann sich von der Identität, die das Konzept Equality Comparable beschreibt, unterscheiden. Zwei Werte sind identisch, wenn alle ihre Teile identisch sind. Das Konzept Strict Weakly Comparable hingegen läßt schwächere Äquivalenzrelationen zu, bei der zwei Werte äquivalent sein können, obwohl sie nicht identisch sind. Man erhält eine solche Vergleichsoperation zum Beispiel, wenn man beim Vergleich nicht den gesamten Zustand der Objekte berücksichtigt. Daraus folgt die praktisch wichtige Möglichkeit, daß man in unterschiedlichen Kontexten unterschiedliche Äquivalenzrelationen definieren kann, die nur die jeweils relevanten Teile des Objektzustands vergleichen.

3.3 Wichtige Programmiertechniken

3.3.1 Generische Datentypen

Die einfachste Technik der generischen Programmierung ist die Definition generischer Datentypen. Dabei handelt es sich meist um container-Bausteine, bei denen der Typ der enthaltenen Objekte durch einen frei wählbaren Parameter repräsentiert wird. Alle modernen C++-Bibliotheken definieren ihre container als generische Bausteine, auch wenn sie ansonsten dem objekt-orientierten Paradigma folgen (dies gilt auch für die Computer Vision-Bibliotheken). Ein einfaches generisches Array[12] kann wie folgt aussehen:

```
template <class ItemType>
class GenericArray
{
    ItemType * data;
    int itsSize;

public:
    GenericArray(int size, ItemType const & initial); // Konstruktor mit
                                                      // explizitem Initialwert
                                                      // für die Elemente
```

[12] Wir verwenden in dieser Arbeit den Begriff Array für eine Datenstruktur, die eine Sequenz von Datenelementen speichert, wobei auf jedes Element in *konstanter* Zeit zugegriffen werden kann (random access). Die übliche deutsche Übersetzung „Feld“ ist leider mehrdeutig (z.B. Felder einer relationalen Datenbank).

```
    GenericArray(int size);                          // Konstruktor ohne
                                                     // expliziten Initialwert
                                                     // (default wird verwendet)

    // Lesen und Schreiben des n-ten Elements in konstanter Zeit
    ItemType getItem(int n)                { return data[n]; }
    void setItem(ItemType new_value, int n) { data[n] = new_value; }
    ...
};
```

Mit einer solchen Definition können Arrays für beliebige Elementtypen automatisch erzeugt werden:

```
typedef GenericArray<byte>  ByteArray;
typedef GenericArray<float> FloatArray;
...
```

Auf diese Weise etablieren wir die Abstraktionen „Elementtyp“ und „Array“ als orthogonale Abstraktionsachsen, ohne daß wir dem Problem des kartesischen Produkts erliegen. Wir können neue Elementtypen definieren, ohne daß dies Einfluß auf das Array hat (`GenericArray` muß nicht modifiziert werden). Ebenso können wir weitere generische Containertypen implementieren und können diese bei Bedarf automatisch für jeden Elementtyp instanziieren.

Das required interface für den jeweiligen Elementtyp hängt davon ab, welchen Konstruktor man verwenden will. In jedem Fall muß der Elementtyp den Anforderungen an das Konzept Assignable genügen. Wir können das Array dann mit dem Konstruktor `GenericArray(int size, ItemType const & initial)` initialisieren, der den in der Variable `initial` angegebenen Wert in jedes Element des Arrays kopiert. Ist der Pixeltyp hingegen ein Modell des Konzepts Default Constructible, ist es nicht notwendig, einen initialen Wert explizit anzugeben. Wir können deshalb für solche Pixeltypen den Konstruktor `GenericArray(int size)` verwenden, der jedes Element des Arrays mit Hilfe des Standardkonstruktors initialisiert.

3.3.2 Generische Algorithmen

Im zweiten Schritt der generischen Programmierung werden auch die Algorithmen in parametrisierte Bausteine umgewandelt. Beispielsweise können die Parameter die Datenstrukturen repräsentieren, auf denen der Algorithmus arbeitet:

```
template <class SourceArray, class DestinationArray>
void copyArray(SourceArray const & src, DestinationArray & dest)
{
    for(int i=0; i < src.size(); ++i)   dest.setItem(src.getItem(i), i);
}
```

Jedem Parameter ist ein required interface für die jeweilige Datenstruktur zugeordnet. Wir spezifizieren das required interface als syntaktische Schnittstelle, d.h. durch eine Liste von Ausdrücken, die auf dem jeweiligen Datentyp ausführbar sein müssen:

```
src.size();  // return-Wert dieser Funktion muß LessThanComparable mit int sein

int i;
src.getItem(i);     // return-Wert muß in den ItemType von
                    // DestinationArray konvertierbar sein

dest.setItem(newvalue, i); // newvalue ist der neue Wert des Elements i
```

Im Fall der einfachen Funktion `copyArray()` ist das required interface länger als die Funktion selbst. Dies wird aber bei komplexeren Funktionen nicht mehr so sein. Außerdem hat man die Möglichkeit, typische Anforderungen als *Konzepte* zu verallgemeinern, so daß die Anforderungen kompakt durch Verweisen auf das entsprechende Konzept angegeben werden können.

Wiederum ist die entscheidende Eigenschaft der obigen Funktion `copyArray`, daß sie das Problem des kartesischen Produkts für eine bestimmte Klasse von Bilddatentypen löst: wir können mit *einer Funktionsimplementation* sämtliche Arrays kopieren, deren Elementtypen ineinander konvertiert werden können, und die die erforderlichen Funktionen `get/setItem` anbieten. Fehlt hingegen die Möglichkeit der Parametrisierung von Algorithmen, gibt es also keine abstrakten Algorithmen, so wächst die Anzahl der notwendigen Kopierfunktionen quadratisch mit der Anzahl der Elementtypen (es sei denn, man entscheidet sich für die Verwendung von bottleneck type mit den bereits diskutierten Nachteilen).

3.3.3 Iteratoren

Bei der Analyse zahlreicher fundamentaler Algorithmen haben Stepanov und Musser festgestellt [MUSSTEP94], daß die obige Form generischer Algorithmen noch nicht optimal ist. Die Funktion `copyArray()` greift erstens direkt auf die Arrays zu und impliziert damit das Vorhandensein der Funktionen `get/setItem`, obwohl andere Zugriffsfunktionen möglicherweise genauso verwendet werden könnten. Außerdem sind diese Funktionen als Indexfunktionen spezifiziert: das gewünschte Element wird durch den Indexparameter `i` angegeben. Im Prinzip könnte der Algorithmus mit diesen Funktionen die Elemente in einer beliebigen Reihenfolge kopieren (random access). In Wirklichkeit tut er dies nicht, sondern greift auf die Elemente in aufsteigender Reihenfolge zu. Das required interface enthält somit speziellere Anforderungen als nötig, ist also nicht minimal.

In der C++-Standardbibliothek [AUSTERN98] spezifiziert man die Schnittstelle zwischen Algorithmen und Datenstrukturen deshalb durch einen Adapter, nämlich

einen *linearen Iterator*. Dies hat den Vorteil, daß das required interface durch wesentlich mehr verschiedene container erfüllt werden kann als wenn die container-Operationen direkt aufgerufen würden. Man kann kompatible lineare Iteratoren für so verschiedene Datenstrukturen anbieten wie Arrays, Listen, Mengen, Bäume, Graphen usw. Selbst auf ein und derselben Datenstruktur können wir gleichzeitig mehrere Navigationsmuster (also alternative Reihenfolgen, in denen die Elemente der Datenstruktur ausgewählt werden) unterstützen, wie z.B. Vorwärts- und Rückwärtsiteration. Die Navigationsfunktionen sind natürlich in jedem Falle anders implementiert, aber der Algorithmus sieht immer die gleiche Iteratorschnittstelle, die transparent auf die Möglichkeiten des jeweiligen Datentyps abgebildet wird.

Da nicht alle Algorithmen gleiche Anforderungen an einen Iterator haben und nicht jeder Datentyp sämtliche Iterationsmuster effizient unterstützen kann, definiert die C++-Standardbibliothek verschiedene Iteratorkategorien, die die unterschiedlichen Varianten systematisieren. Insbesondere werden die Konzepte Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, und Random Access Iterator unterschieden. Diese Konzept werden dadurch charakterisiert, welche Navigationsoperationen besonders effizient, nämlich *in konstanter Zeit* $O(1)$, ausgeführt werden. Je weiter unten ein Iterator in Abbildung 3 steht, desto mehr solcher Operationen bietet er an. Die leistungsfähigeren Iteratorkonzepte sind dabei Spezialisierungen der einfachen und erfüllen auch deren Anforderungen (Abbildung 3).

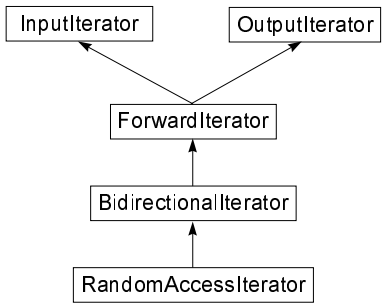


Abbildung 3: Spezialisierungsbeziehungen zwischen den Iteratorkategorien der C++-Standardbibliothek

Wir wollen auf die Eigenschaften der drei letzten Konzepte näher eingehen. Natürlich sind diese Kategorien im C++ Standard wesentlich genauer definiert als es hier möglich ist. Einzelheiten findet man z.B. in [AUSTERN98]. Die Anforderungen an Iteratoren sind dabei so formuliert, wie man es von der traditionellen Zeigerarithmetik in C gewohnt ist. Dadurch sind Zeiger automatisch Modelle aller Iteratorkonzepte.

Forward Iterator: Iteratoren, die diesem Konzept entsprechen, definieren Operationen, um zum nächsten Element der Sequenz zu gelangen (Pre-Inkrement `++iterator` und Post-Inkrement `iterator++`). Außerdem erlauben sie den

Zugriff auf das aktuelle Element durch die Operation `*iterator`. Sie sind zudem Modelle von Assignable, Default Constructible und Equality Comparable. Aufgrund der dritten Eigenschaft kann das Ende der Sequenz durch einen Iterator angezeigt werden, der hinter das letzte gültige Element zeigt (past-the-end bzw. beyond). Wenn der Ausdruck `iterator == beyond` den Wert `true` liefert, ist die Iteration beendet. (Einige weniger wichtige Anforderungen müssen wir hier aus Platzgründen weglassen, siehe jedoch [AUSTERN98].)

Für die Implementierung des Kopieralgorithmus aus dem vorigen Abschnitt reichen Iteratoren der Konzepts Forward Iterator bereits aus. Die Implementation eines Forward Iterator ist auf eine oder mehrere Arten für praktisch alle container-Datentypen möglich. Die folgende Variante von `copy` kann somit wesentlich breiter eingesetzt werden als das ursprüngliche `copyArray`:

```
template <class SourceIterator, class DestinationIterator>
void copy(SourceIterator src, SourceIterator beyond, DestinationIterator dest)
{
    for(; src != beyond; ++src, ++dest)  *dest = *src;
}
```

Bidirectional Iterator: Bidirektionale Iteratoren unterstützen zusätzlich Operationen, um von einem gegebenen Element auf das *vorhergehende* Element der Sequenz zu gelangen (Pre-Dekrement `--iterator` und Post-Dekrement `iterator--`). Mit Hilfe eines solchen Iterators können wir beispielsweise einen Algorithmus `reverseCopy` implementieren, der die Elemente in umgekehrter Reihenfolge kopiert:

```
template <class SourceIterator, class DestinationIterator>
void reverseCopy(SourceIterator src, SourceIterator beyond,
                 DestinationIterator dest)
{
    if(src == beyond) return;
    do {
        --beyond;
        *dest = *beyond;
        ++dest;
    }
    while( src != beyond);
}
```

Random Access Iterator: Random Access Iteratoren bieten zusätzlich zu den Operationen der beiden vorigen Kategorien die Möglichkeit, jederzeit zu einem beliebigen Element der Sequenz zu springen. Insbesondere können wir einen Offset addieren (`iterator += n`, `iterator -= n`, `iterator + n`, `iterator - n`), den Abstand zwischen zwei Iteratoren feststellen (`iter1 - iter2`) und zwei Iteratoren mit Hilfe der „kleiner als“ Operation vergleichen (`iter1 < iter2`). Außerdem gibt es eine neue Zugriffsfunktion, nämlich den Indexoperator

`iterator[n]`, der auf das *n*-te Element relativ zur aktuellen Position zugreift. Man erkennt, daß diese Anforderungen genau der üblichen Syntax der Zeigerarithmetik in C/C++ entsprechen. Daher sind Zeiger auf die Elemente eines Arrays Modelle des Konzepts Random Access Iterator. Random Access Iteratoren werden beispielsweise von der Standardimplementation des Quick Sort Algorithmus benötigt.

Aus den angegebenen Implementationen der Funktionen `copy()` und `reverseCopy()` erkennt man außerdem eine wichtige Konvention der C++-Standardbibliothek: der Bereich der Iteration wird durch ein Paar von Iteratoren gegeben, die auf das erste sowie hinter das letzte Element des gewünschten Bereichs zeigen. Der Bereich ist also durch ein halboffenes Intervall `[first, beyond)` gegeben. Jeder container hat Funktionen `begin()` und `end()`, die Iteratoren auf das erste und hinter das letzte Element der Iteration zurückgeben.

3.3.4 Funktoren

Funktoren stellen eine Verallgemeinerung von Funktionszeigern dar: sie kapseln eine Funktion, die mit der üblichen Syntax eines Funktionsaufrufes ausgeführt werden kann. Die Funktorkonzepte sind dabei so formuliert, daß sie einerseits traditionelle Funktionszeiger einschließen, andererseits aber auch Objekte jeder Klasse, die einen geeigneten Funktionsaufrufoperator `operator()` definieren. Die C++-Standardbibliothek unterscheidet nach der Anzahl der Funktionsargumente drei grundlegende Funktorkonzepte:

Generator: Funktor ohne Argument, Aufrufsyntax `result = f()`. Generatoren dienen zum Initialisieren eines Datenbereichs. Ein Generator könnte beispielsweise eine Konstante oder eine Zufallszahl zurückgeben. Sinnvoll sind auch Generatoren, wo der zurückgegebene Wert von der bisherigen Zahl der Funktionsaufrufe abhängt.

Unary Function: Funktor mit einem Argument, Aufrufsyntax `result = f(arg)`. Zu dieser Gruppe zählen alle unären Transformationen wie z.B. die algebraischen Funktionen.

Binary Function: Funktor mit zwei Argumenten, Aufrufsyntax `result = f(arg1, arg2)`. Hierzu zählen alle binären Funktionen, insbesondere auch die arithmetischen Operationen.

Zusätzlich zu diesen bei [AUSTERN98] definierten Typen wollen wir den Analyser einführen. Austern behandelt diesen Typ als Spezialfall von Unary bzw. Binary Function, bei dem der Rückgabewert ignoriert wird. Wegen der großen Bedeutung

des Konzepts für Computer Vision halten wir einen expliziten Namen für angebracht.

Unary Analyser: Funktor mit einem Argument, ohne Rückgabewert, Aufrufsyntax `f(arg)`. Ein Unary Analyser dient dazu, Statistiken über alle bisherigen Aufrufe des Funktors zu speichern, z.B. den kleinsten und größten bisherigen Argumentwert, den Mittelwert der bisherigen Argumente usw.

Binary Analyser: Funktor mit zwei Argumenten, ohne Rückgabewert, Aufrufsyntax `f(arg1, arg2)`. Ein Binary Analyser dient dazu, Statistiken zu gewinnen, die von zwei Argumenten abhängen, wie z.B. die Korrelation.

Man beachte, daß die Ergebnisse der Analyse über eine beliebige Anzahl von Aufrufen akkumuliert werden. Das heißt, ein Analyser speichert intern die gewünschten Informationen über alle Datenelemente, die ihm bisher übergeben wurden. Nach Beendigung der Iteration können die akkumulierten Informationen mit Hilfe von weiteren Funktionen des Analyser abgefragt werden.

Funktoren werden eingesetzt, um aus einem Algorithmus bestimmte, unabhängig variierende Teilberechnungen herauszulösen. Zum Beispiel ist die Schleife über alle Pixel bei allen unären Punktoperationen identisch, während sich die Art der Transformation jeweils unterscheidet. Durch Einführen eines Funktors können wir die Schleife von der aktuellen Transformation entkoppeln. Aus Sicht der Schleife wird auf diese Weise eine neue Abstraktionsachse für die eigentliche Berechnung definiert. Die C++-Standardbibliothek macht sich dies beispielsweise in der Funktion `transform()` zunutze, die mit Hilfe von abstrakten Funktoren für jede beliebige Punktoperation eingesetzt werden kann:

```
template <class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator src, InputIterator beyond,
                         OutputIterator dest, UnaryFunction f);

vector<float> v1, v2;
... // Initialisierung der Vektoren

transform(v1.begin(), v1.end(), v2.begin(),   sqrt);  // Wurzel aus jedem Element

transform(v1.begin(), v1.end(), v2.begin(), negate<float>());  // Negation
```

Die Verwendung von Funktoren (Funktionsobjekten) hat gegenüber den Funktionszeigern den Vorteil, daß Objekte einen inneren *Zustand* besitzen können. Wir haben deshalb die Möglichkeit, in einem Funktor Informationen über mehrere Aufrufe hinweg zu speichern, so daß wir z.B. die Anzahl der Aufrufe zählen oder die Summe aller bisher übergebenen Argumentwerte berechnen können. Dies wird insbesondere zur Implementierung der Analyser benötigt, wie wir am Beispiel des `MinMaxFunctor` in Abschnitt 4.1 zeigen werden.

Darüber hinaus können wir im Konstruktor des Funktionsobjekts zusätzliche Funktionsparameter initialisieren, die bei jedem Aufruf intern abgefragt werden.

Dadurch können wir die auf Seite 34 beschriebene Funktion `transformImage1()` so verallgemeinern, daß sie auch die Funktionalität von `scaleImage()` einschließt, was in der ursprünglichen Form (vgl. Seite 34) nicht möglich war. Wir definieren dafür einen Funktor `ScaleFunctor` wie folgt:

```
template <class ValueType>
struct ScaleFunctor
{
    ValueType theScale;

    ScaleFunctor(ValueType scale)    // initialisiere Faktor im Konstruktor und
    : theScale(scale)                // speichere ihn in Datenfeld des Funktors
    {}

    ValueType operator()(ValueType arg) const {
        return theScale * arg;               // benutze Parameter zur
        Skalierung
    }
};

// Skalierung mit dem Faktor 2.0
transform(v1.begin(), v1.end(), v2.begin(), ScaleFunctor<float>(2.0));
```

Der konstante Skalierungsparameter ist im Funktor gespeichert und muß nicht bei jedem Aufruf von `operator()` übergeben werden. Man erkennt, daß dies mit einer einfachen Funktion nicht möglich wäre.

Wir werden in Abschnitt 4.1 zeigen, wie wir mit Hilfe von Funktoren das Problem des kartesischen Produkts für Punktoperationen lösen können, dessen unbefriedigende Lösung wir als wichtigen Schwachpunkt bisheriger Softwareansätze in der Computer Vision identifiziert hatten.

3.3.5 Traits

Das Konzept der *traits* ist die in C++ wichtigste Möglichkeit, im Rahmen der generischen Programmierung *Metainformationen* zu verwalten [MYERS95]. Eine traits-Klasse ist ein template, das eine Abbildung *von Typen auf Typen* realisiert. Traits ähneln damit herkömmlichen Funktionen, aber letztere haben als Argumente *Werte* (eines oder mehrerer Typen), während die Argumente einer traits-Klasse Typen sind.

Wir wollen dies an einem Beispiel verdeutlichen. Einige generische Algorithmen, denen als Argument ein Iterator übergeben wurde, müssen den Wertetyp des Iterators kennen, also den Typ des Resultats der Operation `*iterator`. Dieser Typ wird beispielsweise benötigt, wenn der Algorithmus eine Variable zur Speicherung von Zwischenergebnissen des betreffenden Typs definieren möchte. Mit Hilfe der traits-Klasse `iterator_traits` aus der C++-Standardbibliothek kann der Algorithmus den benötigten Typ für einen gegebenen Iterator abfragen:

```
template <class Iterator>
void anAlgorithm(Iterator i)
{
    typedef typename iterator_traits<Iterator>::value_type ValueType;

    ValueType intermediate = *i;   // benutze ValueType als Typ von Zwischenerg.
    ...
}
```

Das template iterator_traits muß für alle vorhandenen Iteratoren überladen werden, so daß der geschachtelte Typname value_type dem gewünschten Typ entspricht. Ist der Iterator beispielsweise ein Zeiger auf ein C++-Array, so definiert man die traits-Klasse so, daß sie auf den Wertetyp des Arrays verweist:

```
template <class Iterator> struct iterator_traits; // forward-declaration

template <class ValueType>
struct iterator_traits<ValueType *>   // partielle Spezialisierung für Zeiger
{
    typedef ValueType value_type;
};
```

Solche und ähnliche traits-Klassen spielen eine große Rolle bei der Selbstkonfiguration von generischen Bausteinen. Selbstkonfigurierende Bausteine sind in der Lage, ihre interne Funktionalität daran anzupassen, auf welche Weise ihr required interface instanziiert wurde. Traits-Klassen ermöglichen es, während der Konfiguration die notwendigen Informationen über den aktuellen Server, im obigen Beispiel also den Wertetyp des Iterators, abzufragen.

Traits-Klassen haben den großen Vorteil, daß sie auch nachträglich implementiert werden können, um bereits existierende Server zu charakterisieren. Wenn ein Baustein Metainformationen benötigt, die bisher noch nicht in rechneradäquater Form vorliegen, kann eine entsprechende Traits-Klasse jederzeit hinzugefügt werden.

3.4 Zusammenfassung des Kapitels

In diesem Kapitel haben wir einige grundlegende Ideen der generischen Programmierung und wichtige Programmiertechniken beschrieben, insbesondere generische Datenstrukturen und Algorithmen sowie eine Reihe von Schnittstellenkonzepten (Iteratoren, Funktoren, Traits). Es wurde anhand von einführenden Beispielen gezeigt, daß die Umsetzung der generischen Programmierung mit Hilfe des template-Mechanismus von C++ zu sehr flexiblen Lösungen führt, die dennoch so effizient wie traditionelle, inflexible Lösungen sind [AUSTERN98].

Kapitel 4

Grundlegende generische Konzepte für Computer Vision

In diesem Kapitel wollen wir beginnen, die Ideen der generischen Programmierung auf Probleme der Computer Vision anzuwenden. Das Kapitel verfolgt dabei zwei Ziele: einerseits wollen wir zeigen, daß die in Kapitel 2 aufgeworfenen Probleme durch die generische Programmierung tatsächlich gelöst werden. Andererseits wollen wir grundlegende generische Konzepte entwickeln, die für Computer Vision Anwendungen benötigt werden.

Wir werden in diesem Kapitel grundlegende abstrakte Bildverarbeitungsalgorithmen studieren und daraus generische Schnittstellen für Computer Vision-Bausteine ableiten. Insbesondere wollen wir die Anforderungen der Punktoperationen und der Faltung analysieren und sie den Möglichkeiten verschiedener Bilddatenstrukturen gegenüberstellen. Daraus werden wir verschiedene generische Konzepte entwickeln, die die grundlegenden Anforderungen der Bildverarbeitung abdecken. Im Sinne eines iterativen Vorgehensmodells passen wir Algorithmen und Datenstrukturen schrittweise an die neuen Konzepte an (daher erscheinen einige Überschriften mehrmals, z.B. „Punktoperationen auf Bildern (1), (2) bzw. (3)“). Am Schluß des Kapitels vergleichen wir unsere generischen Lösungen mit anderen Ansätzen.

4.1 Punktoperationen auf Bildern (1)

Die Punktoperationen gehören zu den grundlegenden Algorithmen der Computer Vision. Aufrund ihrer Einfachheit eignen sie sich sehr gut zur Einführung von generischen Konzepten für diesen Anwendungsbereich. Punktoperationen bearbeiten jeden Punkt eines Bildes oder mehrerer Bilder unabhängig von den übrigen Punkten. Wir unterscheiden die folgenden wesentlichen Punktoperationen:

Initialisierung: Algorithmen, die einen Generator verwenden, um allen Pixeln einen definierten Anfangswert zuzuweisen.

Transformation: Algorithmen, die auf jedes Pixel des Quellbildes eine Unary Function anwenden und das Ergebnis dem örtlich korrespondierenden Pixel des Zielbildes zuweisen.

Kopie: Spezialfall der Transformation, bei der die Werte ohne Änderung kopiert werden (automatische Typumwandlungen sind hierbei jedoch zulässig).

Kombination: Algorithmen, die eine Binary Function auf korrespondierende Pixelwerte von zwei Quellbildern anwenden und das Ergebnis in das entsprechende Pixel des Zielbildes schreiben (ternäre und höhere Kombinationen können ebenfalls definiert werden).

Analyse: Algorithmen, die mit Hilfe eines Analysers statistische Informationen über ein oder mehrere Quellbilder gewinnen.

Bei unserem ersten Schritt, diese Funktionalität bereitzustellen, wollen wir die Algorithmen verwenden, die die C++-Standardbibliothek bereits enthält. Für unseren Zweck kommen die folgenden Algorithmen infrage:

```
// Initialisierung
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator beyond, T const & value);

// Transformation
template <class InputIterator, class OutputIterator, class UnaryFunction >
OutputIterator transform(InputIterator src, InputIterator beyond,
                         OutputIterator dest, UnaryFunction f);

// Kopie
template <class InputIterator, class OutputIterator >
OutputIterator copy(InputIterator src, InputIterator beyond,
                    OutputIterator dest);

// Kombination
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class BinaryFunction >
OutputIterator transform(InputIterator src1, InputIterator beyond1,
                         InputIterator src2,
                         OutputIterator dest, BinaryFunction f);
```

```
// Analyse
template <class InputIterator, class UnaryAnalyser >
UnaryAnalyser for_each(InputIterator src, InputIterator beyond, UnaryAnalyser f);
```

Die Semantik dieser Funktionen entspricht genau den Anforderungen, die wir oben aufgelistet haben (vgl. [AUSTERN98]). Wollen wir diese Funktionen auf Bilder anwenden, müssen wir für die Bilder eine Schnittstelle definieren, die den Anforderungen des required interface der Funktionen entspricht. Insbesondere müssen wir lineare (1-dimensionale) Iteratoren definieren. Es gibt natürlich viele verschiedene Möglichkeiten, die Pixel eines zweidimensionalen Bildes in eine lineare Reihenfolge zu bringen, wichtig ist nur, daß alle Bilder dieselbe Reihenfolge benutzen. Wir definieren deshalb:

ScanOrderIterator: Der ScanOrderIterator ist ein ForwardIterator, der alle Pixel eines Bildes so abtastet, daß die Pixel jeder Zeile von links nach rechts und die Zeilen von oben nach unten angesteuert werden (engl. row-major order).

Diese Definition hat den Vorteil, daß sie einerseits sehr einfach implementiert werden kann (sind die Bilddaten intern als lineares Array gespeichert, kann ein Zeiger auf dieses Array als ScanOrderIterator verwendet werden), und andererseits dieselbe Reihenfolge auch bei anderen Operationen häufig verwendet wird (z.B. Einlesen der Bilddaten aus einem File, Kompression mit arithmetischer oder Lauflängenkodierung).

Nehmen wir an, wir haben einen Bilddatentyp definiert, bei dem ScanOrderIteratoren über die Funktionen `begin()` (erstes Pixel) und `end()` (hinter dem letzten Pixel) erfragt werden können. Dann können wir ein solches Bild folgendermaßen initialisieren:

```
GenericImage1< byte > image(100, 200);  // erzeuge ein Bild der Größe 100x200

fill(image.begin(), image.end(), 0);    // setze alle Pixel auf den Wert 0
```

Ähnlich werden die übrigen Funktionen benutzt. Sind an einem Algorithmus mehrere Bilder beteiligt, müssen die Bilder natürlich die gleiche Größe haben, damit ihre ScanOrderIteratoren synchron laufen. Wir wollen hier exemplarisch einige weitere Varianten herausgreifen:

```
// erzeuge einige gleichgroße Bilder
GenericImage1< float > img1(100, 200), img2(100,200), image3(100,200);

... // Initialisierung der Bilder

// negiere ein Bild mittels negate-Funktor
transform(img1.begin(), img1.end(), img2.begin(), negate<float>());

// ziehe die Wurzel aus jedem Pixel mittels sqrt-Funktion
transform(img1.begin(), img1.end(), img2.begin(), sqrt);
```

```
// addiere zwei Bilder mittels plus-Funktor
transform(img1.begin(), img1.end(), img2.begin(), img3.begin(), plus<float>());

// finde den jeweils kleineren Pixelwert mittels min-Funktion
transform(img1.begin(), img1.end(), img2.begin(), img3.begin(), min<float>);
```

Komplexere Funktoren können aus einfachen zusammengesetzt werden, indem man Kombinatoren dazwischen schaltet, wie es aus der funktionalen Programmierung bekannt ist [C++98, AUSTERN98]:

```
// tansformiere ein Bild mit der Funktion exp(-beta*x)
float beta = ...;
transform(img1.begin(), img1.end(), img2.begin(),
          compose1(ptr_fun(exp), bind1st(multiplies<float>(), -beta)));
```

Der Kombinator `bind1st()` bindet das erste Argument der gegebenen Funktion (hier: `multiplies<float>`) an die gegebene Konstante (`-beta`). Der Ausdruck `bind1st(multiplies<float>(), -beta)` realisiert also die Funktion *–beta * x*. Der zweite Kombinator `compose1()` bewirkt die Hintereinanderausführung der angegebenen Funktionen, der gesamte Ausdruck berechnet somit *exp(-beta * x)*. Mit Hilfe von sogenannten expression templates kann man die Erzeugung zusammengesetzter Funktoren syntaktisch vereinfachen und gleichzeitig noch leistungsfähiger gestalten. Entsprechende Mechanismen sind in [KÖTHE99A] beschrieben.

Natürlich ist es auch möglich, komplexe Funktoren explizit zu programmieren. Ein Beispiel hierfür ist die Transformation eines Bildes mit Hilfe einer vorher berechneten Wertetabelle (engl. look-up table). Ein solcher Funktor könnte für Argumente vom Typ byte etwa so aussehen:

```
struct LookUpTableFunctor
{
    byte lookUpTable[256];

    byte operator()(byte const & old_vlaue) const {
        return lookUpTable[old_value];   // Transformation mit Hilfe der Tabelle
    }
};
```

Dieser Funktor kann zum Beispiel benutzt werden, um ein Grauwertbild zu invertieren:

```
GenericImage1<byte> img1(100,200), img2(100,200);
...

LookUpTableFunctor lut;

// Initialisierung der Tabelle für Invertierung
for(int i=0; i<256; ++i)   lut.lookUpTable[i] = 255 - i;

transform(img1.begin(), img1.end(), img2.begin(), lut);
```

Sehr interessant ist auch die Möglichkeit der Gewinnung statistischer Informationen mit Hilfe von Funktoren. Der folgende Funktor kann beispielsweise eingesetzt werden, wenn man sich für den kleinsten und größten Pixelwert eines Bildes interessiert:

```
template <class T>
struct MinMaxFunctor
{
    T min, max;   // Minimum und Maximum der bisher "gesehenen" Werte
    int count;    // Anzahl der bisher "gesehenen" Werte

    MinMaxFunctor() : count(0) {}   // intialisiere Anzahl mit 0

    void operator()(T const & v) {
        if(count == 0)
        {
            min = max = v;       // initialisiere min und max beim ersten
            Aufruf
        }
        else
        {
            if(v < min) min = v; // aktualisiere min und
            if(max < v) max = v; // max, wenn nötig
        }
        ++count;                 // zähle die Anzahl der Aufrufe
    }
};
```

In Kombination mit dem Algorithmus `for_each()` können wir diesen Funktor wie folgt verwenden:

```
GenericImage1<float> img(100,200);
...

MinmaxFunctor<float> minmax;
minmax = for_each(img.begin(), img.end(), minmax);

cout << "Minimum: " << minmax.min << ", Maximum: " << minmax.max << endl;
```

Auf dieselbe Art kann man auch den Mittelwert, die Varianz oder Statistiken höherer Ordnung für ein Bild berechnen.

Wir halten fest, daß der hier skizzierte generische Ansatz das Problem des kartesischen Produkts löst, das wir in Kapitel 2 an eben diesen Punktoperationen aufgezeigt hatten: wir benötigen nur 5 grundlegende Funktionen und eine Reihe von Funktoren (von denen die wichtigsten bereits in der C++-Standardbibliothek definiert sind), um eine große Menge unterschiedlicher Punktoperationen zu realisieren. Diese Grundbausteine sind unabhängig von den verwendeten Bild- und Pixeltypen. Für jede konkrete Situation können die Grundbausteine nach Bedarf zusammengesetzt werden, und der Compiler generiert automatisch den notwendi-

gen Code. Man muß also nicht, wie bei Khoros oder Tina, eine große Zahl von Varianten einzeln implementieren.

Damit die genannten templates instanziiert werden können, müssen die beteiligten Typen das required interface des jeweiligen Algorithmus erfüllen. Für die Punktoperationen bedeutet das, daß der gewünschte Funktor auf den jeweiligen Pixeltyp angewendet werden kann. Beispielsweise setzt der `MinMaxFunctor` voraus, daß für den Pixeltyp die Operation "<" definiert ist. Pixeltypen, für die dies nicht gilt, wie zum Beispiel komplexe Zahlen, können nicht verwendet werden.

Für die Bilddatentypen gilt, daß sie einen ScanOrderIterator anbieten müssen, und daß sie *gleich groß* sein müssen, damit korrespondierende Punkte in der Iterationssequenz die gleiche Position haben. In der Praxis zeigt sich, daß dies eine relativ starke Einschränkung ist: häufig soll ein Algorithmus nur auf überlappende Bereiche unterschiedlicher Bilder angewendet werden. Allgemeiner ausgedrückt, wollen wir in der Lage sein, Algorithmen auf beliebige Bildausschnitte zu beschränken. Dies ist mit den bisher verwendeten linearen Iteratoren nicht ohne weiteres möglich, denn die 2-dimensionale Struktur eines Bildes ist in diesen Iteratoren nicht repräsentiert und kann somit von den Algorithmen auch nicht berücksichtigt werden. Wir müssen also Algorithmen definieren, deren required interface die 2-dimensionale Bildstruktur explizit berücksichtigt.

4.2 Die Faltungsoperation auf Bildern (1)

Neben den Punktoperationen bilden die Faltungsalgorithmen die zweite Gruppe der grundlegenden Bildverarbeitungsalgorithmen. Faltungsalgorithmen sind die Basis für viele andere Algorithmen der Computer Vision, wie z.B. Kantendetektion, Pyramidenberechnung, Texturanalyse und die Berechnung des optischen Flusses [JÄHNE91]. An dieser Stelle ist die Faltung vor allem deshalb interessant, weil sie im Unterschied zu den Punktoperationen eine *Umgebung* jedes Punktes benötigt. Die 2-dimensionale Faltung ist folgendermaßen definiert [JÄHNE91]:

$$(f * g)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x+u, y+v) g(-u,-v)\, du\, dv \tag{4.1}$$

Dabei ist $f(.)$ die Bildfunktion und $g(.)$ der sogenannte Faltungskern. Im Diskreten gehen die Integrale in eine Summe über. Für den typischen Fall eines Faltungskerns endlicher Größe erhalten wir endliche Summationsgrenzen:

$$(f * g)(x, y) = \sum_{i=-I}^{K} \sum_{j=-J}^{M} g(-i,-j) f(x+i, y+j) \tag{4.2}$$

Der Ausdruck auf der rechten Seite muß für jeden Punkt berechnet werden. Wir erkennen, daß wir dafür auf die Pixelwerte in einer *Umgebung* um jeden Punkt zugreifen müssen. Wenn man nur über einen ScanOrderIterator verfügt, ist es unmöglich, die Nachbarpunkte in allen Richtungen zu bestimmen. Um beispielsweise festzustellen, welcher Punkt über einem gegebenen Punkt liegt (welcher also eine um 1 kleinere y-Koordinate hat), benötigt man neben dem ScanOrderIterator die Breite des Bildes.

Wir stellen auch hier – und in noch überzeugenderer Weise als bei den Punktoperationen – fest, daß die 1-dimensionalen Iteratorkonzepte der C++-Standardbibliothek für die Implementierung von Algorithmen auf Bildern nicht ausreichen. Wir wollen uns deshalb im nächsten Abschnitt damit befassen, diese Konzepte auf zwei Dimensionen zu erweitern.

4.3 Zweidimensionale Iteratoren

Wir haben festgestellt, daß die Iteratorkonzepte, die in der C++-Standardbibliothek definiert werden, die 2-dimensionale Struktur eines Bildes nicht widerspiegeln: sie interpretieren ein Bild nur als lineare Folge von Werten. Für viele Anwendungen reicht dies nicht aus. Bereits bei den Punktoperationen benötigen wir Informationen über die 2-dimensionale Struktur des Bildes, wenn wir Bilder unterschiedlicher Größe verknüpfen oder auf Teilbildern arbeiten wollen. Erst recht benötigen wir die 2-dimensionale Struktur, wenn wir, wie bei der Faltung, auf eine Umgebung des aktuellen Punkts zugreifen müssen.

Wir könnten das Problem dadurch lösen, indem wir jedem Algorithmus einen ScanOrderIterator *und* die Breite des Bildes übergeben würden. Die Algorithmen wären dann in der Lage, intern die notwendigen Indexberechnungen auszuführen. Dies widerspricht aber dem Prinzip der Kohäsion: neben ihrer eigentlichen Aufgabe müßten die Algorithmen nun außerdem wissen, wie man aus einem linearen Iterator und der Breite des Bildes die 2-dimensionale Bildgeometrie ableitet. Es wäre besser, wenn diese Information in einem eigenen Baustein gekapselt wäre. Dies ist die Aufgabe des *2-dimensionalen Iterators*.

Wir entwickeln hier erstmals ein Iteratorkonzept, das die Entwurfsprinzipien der Iteratoren der C++-Standardbibliothek auf 2-dimensionale Strukturen erweitert. Aus den Erfahrungen mit existierenden Systemen ergeben sich hierfür zwei Ansätze: Indexoperationen und Navigationsoperationen. Bei der Verwendung des ersten Ansatzes verwaltet der Algorithmus eine Differenz (eine ganze Zahl in 1D, ein Paar von ganzen Zahlen in 2D), und der Iterator liefert den zugehörigen Wert, wenn ihm die Differenz übergeben wird. Beim Navigationsansatz hingegen gibt der Iterator

immer den Wert zurück, auf dem er sich gerade befindet. Um einen anderen Wert abzufragen, muß der Iterator vorher auf die betreffende Position bewegt werden.

Je nach Problemstellung können Algorithmen die eine oder die andere Variante bevorzugen. Da Bilddatenstrukturen in den meisten Fällen beide Ansätze effizient implementieren können, werden wir unseren Bilditerator so definieren, daß er beide Ansätze unterstützt. Der Bilditerator verallgemeinert damit den Random Access Iterator der C++-Standardbibliothek, der im 1-dimensionalen ebenfalls beide Varianten anbietet.

Indexoperation

Wenn wir den Index auf zwei Dimensionen erweitern wollen, haben wir hierzu zwei Möglichkeiten. Wir können einerseits die beiden Koordinatenwerte explizit übergeben. Dafür bietet sich in C++ die Funktion `operator()` an. Ist ein `ImageIterator` gegeben, kann man mit folgender Syntax auf das Pixel `(x,y)` relativ zur aktuellen Iteratorposition zugreifen:

```
ImageIterator iterator = ...;
int x = 10, y = 20;
ValueType value = iterator(x, y);
```

Allerdings ist diese Syntax nicht mit der Syntax im 1-dimensionalen kompatibel. Im 1-dimensionalen wird zur Indexierung die Funktion `operator[]` verwendet, die in C++ nur ein Argument akzeptiert. Wir können allerdings auch diesen Operator auf zwei Dimensionen erweitern, indem wir einen 2-dimensionalen Differenzvektor einführen. Wir wollen diesen Vektor `Diff2D` nennen und ihn wie folgt definieren:

```
struct Diff2D
{
    Diff2D(int dx, int dy) : x(dx), y(dy) {}

    int x, y;
};
```

Mit Hilfe dieser 2-dimensionalen Differenz können wir den Zugriff folgendermaßen implementieren:

```
ImageIterator iterator = ...;
Diff2D diff(10,20);
ValueType value = iterator[diff];
```

Wir werden im folgenden beide Varianten zulassen, da die erste in Algorithmen häufig einfacher zu handhaben ist, während die zweite es gestattet, Zugriffsobjekte zu definieren, die für ein- und zweidimensionale Iteratoren gleichermaßen verwendbar sind (siehe Abschnitt 4.6).

Navigation

Beim Navigationsansatz muß der Algorithmus den Iterator an die gewünschte Position bewegen, ehe er die korrespondierenden Daten abfragen kann. Dafür muß der Algorithmus in der Lage sein, die gewünschte Bewegungsrichtung festzulegen. In zwei Dimensionen gibt es vier prinzipielle Richtungen: rechts, links, nach oben, nach unten. Wir könnten also in der Bilditeratorklasse Methoden einfügen, die den Iterator in die gewünschte Richtung bewegen, wie z.B. `iterator.right()`, `iterator.up()` usw.

Dies beraubt uns aber der Vorteile der Operatorschreibweise in C++. Die Operatorschreibweise ist nicht nur kompakter, sondern sie impliziert eine im C++-Standard klar definierte Semantik, die mit Methodenaufrufen nicht ohne weiteres ersetzt werden kann. Beispielsweise ist der Unterschied zwischen Pre-Inkrement (`++iterator`) und Post-Inkrement (`iterator++`) bei der Operatorschreibweise klar definiert, während dies bei Methoden wie `iterator.right()` weit weniger klar ist.

Es erweist sich deshalb als günstiger, für die Navigation die Operatorschreibweise auf 2 Dimensionen zu verallgemeinern. Wir erreichen dies, indem wir in die Bilditeratoren zwei Objekte einbetten, die unterschiedliche Blickwinkel auf die Navigationsdaten definieren: beide Objekte unterstützen die aus der C++-Standardbibliothek bekannten Navigationsoperationen, aber beziehen sie jeweils nur auf eine Koordinate. Zur Veranschaulichung betrachte man die folgende Iteratordefinition:

```
class ImageIterator {
  public:
    ...

    class MoveX {
        ... // Daten, die zur Navigation in x-Richtung notwendig sind

      public:
        // die Operatoren beziehen sich nur auf die x-Achse, z.B.
        MoveX & operator++();        // gehe nach rechts
        MoveX & operator--();        // gehe nach links
        MoveX & operator+=(int dx);  // springe um angegebenen Betrag horizontal
        ...
    };

    class MoveY {
        ... // Daten, die zur Navigation in y-Richtung notwendig sind

      public:
        // die Operatoren beziehen sich nur auf die y-Achse, z.B.
        MoveY & operator++();         // gehe nach unten
        MoveY & operator--();         // gehe nach oben
```

```
        MoveY & operator+=(int dy);   // springe um angegebenen Betrag vertikal
        ...
    };

    MoveX x;   // x-Sicht auf die Navigationsdaten
    MoveY y;   // y-Sicht auf die Navigationsdaten
};
```

Wir können jetzt die eingebetteten Objekte benutzen, um die Bewegungsrichtung in zwei Dimensionen mit derselben Syntax zu spezifizieren, wie sie bei linearen Iteratoren benutzt wird, wobei die Angabe von x bzw. y den jeweiligen Operator auf die horizontale bzw. vertikale Koordinate bezieht (wie üblich wächst x von links nach rechts und y von oben nach unten, der Koordinatenursprung liegt in der linken oberen Ecke des Bildes):

```
ImageIterator iterator = ...;

++i.x;   // gehe nach rechts
--i.x;   // gehe nach links
++i.y;   // gehe nach unten
--i.y;   // gehe nach oben
```

In gleicher Weise werden die übrigen Operatoren (`operator+=`, `operator-=`, `operator==`) verallgemeinert. Mit Hilfe des 2-dimensionalen Differenzvektors `Diff2D` definieren wir außerdem eine Reihe von Operationen, die sich auf beide Koordinatenrichtungen gleichzeitig beziehen. Diese werden direkt auf den Iterator angewandt, da hier die Navigationsobjekte nicht notwendig sind:

```
iterator += Diff2D(10, -15);        // gehe 10 Pixel nach rechts und 15 nach oben
ImageIterator iterator2 = ...;      // definiere zweite Iteratorvariable
Diff2D diff = iterator - iterator2; // berechne Differenz der Iteratoren
```

Eine vollständige Liste der Operationen, die ein Bilditerator unterstützen muß, findet sich in Tabelle 2. Diese Tabelle beschreibt im Detail sämtliche Anforderungen an einen random access ImageIterator.

Tabelle 2 (rechte Seite): Vollständige Liste der Anforderungen an den `ImageIterator`

Bedeutung der Symbole:	`ImageIterator i = ..., j = ...;` (müssen sich auf dasselbe Bild beziehen)
	`int dx, dy;`
	`Diff2D diff;`

Bemerkungen:	[1] in der inneren Schleife zu bevorzugen [2] Kombination von Zugriff und Navigation (d.h. `*i.x++`) nicht erlaubt [3] nur bei nicht-konstanten Iteratoren [4] nur bei konstanten Iteratoren

Operation	Resultat	Semantik
`ImageIterator::MoveX`		Typ des x-Navigator
`ImageIterator::MoveY`		Typ des y-Navigator
`++i.x; i.x++`	`void`	Inkrementieren der x-Koordinate[1,2]
`--i.x; i.x--`	`void`	Dekrementieren der x-Koordinate[1,2]
`i.x += dx`	`ImageIterator::MoveX &`	Addieren von `dx` zur x-Koordinate[1]
`i.x -= dx`	`ImageIterator::MoveX &`	Subtrahieren von `dx` von der x-Koordinate[1]
`i.x - j.x`	`int`	Differenz der x-Koordinaten
`i.x = j.x`	`ImageIterator::MoveX &`	`i.x += j.x - i.x`
`i.x == i.y`	`bool`	`j.x - i.x == 0` [1]
`i.x < j.x`	`bool`	`j.x - i.x > 0`
`++i.y; i.y++`	`void`	Inkrementieren der y-Koordinate[2]
`--i.y; i.y--`	`void`	Dekrementieren der y-Koordinate[2]
`i.y += dy`	`ImageIterator::MoveY &`	Addieren von `dy` zur y-Koordinate
`i.y -= dy`	`ImageIterator::MoveY &`	Subtrahieren von `dy` von der y-Koordinate
`i.y - j.y`	`int`	Differenz der y-Koordinaten
`i.y = j.y`	`ImageIterator::MoveY &`	`i.y += j.y - i.y`
`i.y == j.y`	`bool`	`j.y - i.y == 0`
`i.y < j.y`	`bool`	`j.y - i.y > 0`
`ImageIterator k(i)`		Kopierkonstruktor
`k = i`	`ImageIterator &`	Zuweisung
`i += diff`	`ImageIterator &`	`(i.x += diff.x, i.y += diff.y)`
`i -= diff`	`ImageIterator &`	`(i.x -= diff.x, i.y -= diff.y)`
`i + diff`	`ImageIterator`	`{ ImageIterator tmp(i); tmp += diff; return tmp; }`
`i - diff`	`ImageIterator`	`{ ImageIterator tmp(i); tmp -= diff; return tmp; }`
`i - j`	`Diff2D`	`{ Diff2D tmp(i.x - j.x, i.y - j.y); return tmp; }`
`i == j`	`bool`	`i.x == j.x && i.y == j.y`
Optionale Operationen (siehe Abschnitt 4.6)		
`ImageIterator::value_type`		Typ der Pixel
`*i`	`ImageIterator::value_type &`	Lese-/Schreibzugriff auf das aktuelle Pixel[3]
`*i`	`ImageIterator::value_type`	Lesezugriff auf das aktuelle Pixel[4]
`i[diff]`	`ImageIterator::value_type &`	Lese-/Schreibzugriff auf Pixel mit Offset `diff`[3]
`i[diff]`	`ImageIterator::value_type`	Lesezugriff auf Pixel mit Offset `diff`[4]
`i(dx, dy)`	`ImageIterator::value_type &`	Lese-/Schreibzugriff auf Pixel mit Offset `(dx, dy)`[3]
`i(dx, dy)`	`ImageIterator::value_type`	Lesezugriff auf Pixel mit Offset `(dx, dy)`[4]

Obwohl es ohne weiteres möglich wäre, auch Vorwärts- und bidirektionale (in 2-D besser als unidirektional zu bezeichnende) Bilditeratoren zu definieren, hat sich dies in der Praxis bisher nicht als notwendig erwiesen. Die Zugriffsoperationen (`operator*`, `operator[]`, `operator()`) sind als optional gekennzeichnet, da sich im Abschnitt 4.5 zeigen wird, daß nicht alle Bilddatenstrukturen sie effizient implementieren können. Wir werden den Datenzugriff deshalb in Abschnitt 4.6 in speziellen Zugriffsobjekten kapseln.

Wie in der C++-Standardbibliothek müssen alle aufgeführten Operationen in konstanter Zeit ausgeführt werden. Dennoch sind im allgemeinen nicht alle Operationen gleich schnell. Deshalb ist in der Tabelle zusätzlich festgelegt, daß die Operationen, die sich auf die x-Koordinate beziehen, nicht langsamer als andere implementiert werden sollten und daher in inneren Schleifen zu bevorzugen sind. Der Vollständigkeit halber sei noch hinzugefügt, daß sämtliche Navigationsoperationen miteinander kommutieren müssen, so daß die gleiche Position erreicht wird, wenn eine bestimmte Navigationssequenz (z.B. `++i.x; ++i.y;`) in einer anderen Reihenfolge (also `++i.y; ++i.x`) ausgeführt wird.

4.4 Punktoperationen auf Bildern (2)

Nachdem wir im vorigen Abschnitt 2-dimensionale Iteratoren eingeführt haben, wollen wir diese nun benutzen, um Algorithmen für die Punktoperationen zu definieren, die die 2-dimensionale Struktur der Bilder berücksichtigen. Dadurch verallgemeinern wir einerseits die C++-Standardbibliothek, andererseits ist dies der erste Schritt, generische Techniken in die Computer Vision einzuführen und die Flexibilität der Softwarebausteine wesentlich über das aus anderen Computer Vision-Systemen Bekannte hinaus zu steigern.

Hierzu ist es zunächst notwendig zu klären, wie ein Algorithmus mit Hilfe von 2-dimensionalen Iteratoren die Iterationsgrenzen bestimmen kann. In der C++-Standardbibliothek werden die Iterationsgrenzen durch Iteratoren angezeigt, die sich hinter der letzten gültigen Position befinden (past-the-end). In einer linearen Sequenz gibt es zwei solche Positionen (vor dem ersten und hinter dem letzten Element), in einem Bild dagegen wesentlich mehr: wenn wir alle Punkte außerhalb des Bildes, die mindestens einen 8-Nachbarn[13] im Bild haben, als past-the-border Positionen bezeichnen, hat ein Bild der Größe $w \times h$ genau $2(w+h+2)$ past-the-border Positionen. Entsprechend viele Bilditeratoren benötigen wir, wenn wir alle Begrenzungen explizit repräsentieren wollen.

[13] Die 8-Nachbarn eines Bildpunktes sind die horizontal, vertikal und diagonal an diesen Punkt angrenzenden Bildpunkte.

Es ist jedoch nicht notwendig, alle diese Grenzmarkierungen explizit zu speichern und an Algorithmen zu übergeben, solange wir gewährleisten, daß auch auf Grenzmarkierungen bestimmte Navigationsoperationen erlaubt sind. Dies gestattet es uns, aus einer kleinen Zahl von Markierungen andere je nach Bedarf zu generieren. Insbesondere fordern wir z.B. für eine rechte Grenzmarkierung (also das Ende einer Zeile), daß die Operation `++end_of_row.y` die rechte Grenzmarkierung der folgenden Zeile generiert. Analog gilt dies für die anderen Navigationsoperationen und die anderen Grenzen.

Mit dieser Forderung und den Operationen aus Tabelle 2 genügen zwei Bilditeratoren, um einen rechteckigen Bereich eines Bildes zu spezifizieren: einer für die linke obere Ecke und einer für die rechte untere Ecke des Bereichs. Analog zur C++-Standardbibliothek folgen wir der Konvention, daß der erste Iterator, `upperleft`, in der gewünschten Region liegt, während der zweite, `lowerright`, den ersten Punkt diagonal *außerhalb* der Region markiert. Diese Konvention ist in Abbildung 4 veranschaulicht. Wir werden im folgenden stets solche Iteratorpaare benutzen, um rechteckige Regionen im Bild zu definieren. Man beachte, daß der zweite Iterator nur die Grenze der Region markiert, er darf nicht zum Lesen oder Schreiben von Pixeldaten verwendet werden, da er außerhalb des Bildes liegen kann. Wir werden uns in Abschnitt 5.1 ausführlicher mit der Behandlung des Regionenrands befassen.

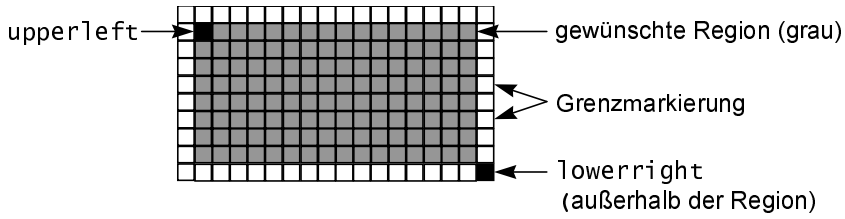


Abbildung 4: Festlegung einer rechteckigen Region of Interest durch zwei Iteratoren

Mit Hilfe dieser Konvention können wir nun die fünf in Abschnitt 4.1 beschriebenen Grundoperationen (Initialisierung, Transformation, Kopie, Kombination und Analyse) in 2-dimensionaler Form implementieren. Wir werden diese Algorithmen durch das Suffix "`Image`" im Funktionsnamen (z.B. `initImage`, `transformImage`) kennzeichnen. Um zu zeigen, daß Algorithmen die Iteratoren auf verschiedene Art benutzen können, werden wir in jeder der folgenden Operationen eine andere Variante benutzen:

```
// Initialisierung
template <class ImageIterator, class T>
void initImage(ImageIterator upperleft, ImageIterator lowerright,
               T const & value)
{
    Diff2D size = lowerright - upperleft;  // Bildgröße

    // Variante 1: Benutzung des Indexoperators operator()
    for(int y=0; y < size.y; ++y)
    {
        for(int x=0; x < size.x; ++x)
        {
            upperleft(x, y) = value;  // schreibe Initialwert in Pixel (x,y)
        }
    }
}

// Transformation
template <class ImageIterator1, class ImageIterator2, class UnaryFunction >
void transformImage(ImageIterator1 srcupperleft, ImageIterator1 srclowerright,
                    ImageIterator2 destupperleft, UnaryFunction f)
{
    Diff2D size = srclowerright - srcupperleft;  // Bildgröße
    Diff2D current(0,0);

    // Variante 2: Benutzung des Indexoperators operator[]
    for(; current.y < size.y; ++current.y)
    {
        for(current.x=0; current.x < size.x; ++current.x)
        {
            // Transformation des aktuellen Punktes
            destupperleft[current] = f(srcupperleft[current]);
        }
    }
}

// Kopie
template <class ImageIterator1, class ImageIterator2 >
void copyImage(ImageIterator1 srcupperleft, ImageIterator1 srclowerright,
               ImageIterator2 destupperleft)
{
    // Variante 3: Verwendung von Navigationsoperationen und operator*
    for(; srcupperleft.y < srclowerright.y; ++srcupperleft.y, ++destupperleft.y)
    {
        ImageIterator1 scurrent = srcupperleft;
        ImageIterator2 dcurrent = destupperleft;

        for(; scurrent.x < srclowerright.x; ++scurrent.x, ++dcurrent.x)
        {
            *dcurrent = *scurrent;  // Kopieren des aktuellen Punktes
        }
    }
}
```

```
// Kombination
template < class ImageIterator1, class ImageIterator2,
           class ImageIterator3, class BinaryFunction >
void combineTwoImages(ImageIterator1 src1_ul, ImageIterator1 src1_lr,
                      ImageIterator2 src2_ul, ImageIterator3 dest_ul,
                      BinaryFunction f)
{
    int width  = src1_lr.x - src1_ul.x;  // Bildbreite
    int height = src1_lr.y - src1_ul.y;  // Bildhöhe

    // Variante 4: Navigation, Grenzmarkierung am Ende der ersten Spalte und
    // der aktuellen Zeile
    ImageIterator1 beyond_first_column = src1_ul + Diff2D(0, height);

    for(; src1_ul != beyond_first_column; ++src1_ul.y, ++src2_ul.y, ++dest_ul.y)
    {
        ImageIterator1 s1cur = src1_ul;
        ImageIterator2 s2cur = src2_ul;
        ImageIterator3 destcur = dest_ul;
        ImageIterator1 beyond_current_row = s1cur + Diff2D(width, 0);
        for(; s1cur != beyond_current_row; ++s1cur.x, ++s2cur.x, ++destcur.x)
        {
            *destcur = f(*s1cur, *s2cur);  // Kombination der aktuellen Werte
        }
    }
}

// Analyse
template <class ImageIterator, class UnaryAnalyser >
void inspectImage(ImageIterator ul, ImageIterator lr, UnaryAnalyser & f)
{
    // Variante 5: Navigation, aber horizontale und vertikale Iteration sind
    // vertauscht (dies zeigt, daß x und y symmetrisch sind)
    for(; ul.x < lr.x; ++ul.x)
    {
        ImageIterator current = ul;
        for(; current.y < lr.y; ++current.y)
        {
            f(*current);  // Analysieren des aktuellen Punktes
        }
    }
}
```

Mit diesen Definitionen entspricht die Funktionalität der 2-dimensionalen Algorithmen genau der ihrer 1-dimensionalen Gegenstücke. Wir sind jetzt aber in der Lage, auf beliebigen rechteckigen Regionen zu arbeiten. Angenommen, der Bilddatentyp bietet Methoden `upperLeft()` und `lowerRight()`, die die entsprechenden Iteratoren zurückgeben. Dann können wir die neuen Algorithmen wie folgt benutzen:

```
GenericImage2<float> img1(100, 200), img2(100, 200);

// kopiere das gesamte Bild
// (dies ist äquivalent zu copy(img1.begin(), img1.end(), img2.begin()); )
copyImage(img1.upperLeft(), img1.lowerRight(), img2.upperLeft());

// kopiere eine Region aus img1 in die Mitte von img2
ImageIterator<float> iter1 = img1.upperLeft();
copyImage(iter1 + Diff2D(25, 25), iter1 + Diff2D(75, 75),
          img2.upperLeft() + Diff2D(25, 75));

// initialisiere einen 2 Pixel breiten Rand von img2 mit dem Wert 1000
initImage(img2.upperLeft(), img2.upperLeft() + Diff2D(100,2), 1000);
initImage(img2.upperLeft(), img2.upperLeft() + Diff2D(2, 200), 1000);
initImage(img2.lowerRight() - Diff2D(100, 2), img2.lowerRight(), 1000);
initImage(img2.lowerRight() - Diff2D(2, 200), img2.lowerRight(), 1000);

// subtrahiere ein Bild von einer um 1 Pixel verschobenen Version seiner selbst
// (dies realisiert einen einfachen Operator für die erste Ableitung)
combineTwoImages(img1.upperLeft() + Diff2D(1,0), img1.lowerRight(),
                 img1.upperLeft(), img2.upperLeft(), minus<float>());
```

Diese vier Beispiele verdeutlichen eine Reihe wichtiger Eigenschaften der neu definierten Algorithmen und Iteratoren:

- Alle Operationen, die wir mit ScanOrderIteratoren ausführen konnten, können wir auch mit ImageIteratoren durchführen.
- Wir können ohne Änderung der Algorithmen auf rechteckigen Unterregionen arbeiten.
- Arbeitet ein Algorithmus auf mehreren Bildern, können diese Unterregionen in jedem Bild anders positioniert werden, solange sie gleiche Größe haben. Aufgrund dieser Eigenschaft besteht z.B. keine Notwendigkeit mehr für zusätzliche Algorithmen, um Teilbilder aus einem größeren Bild zu extrahieren und in einem anderen Bild an beliebiger Stelle wieder einzufügen – diese Funktionalität ist jetzt ein Spezialfall von `copyImage()`.
- Die Funktoren sind durch die Einführung der neuen Algorithmen nicht betroffen. Alle Funktoren, die mit den 1-dimensionalen Algorithmen verwendet werden konnten, arbeiten ebenso mit den 2-dimensionalen zusammen. Dies zeigt erneut, wie effektiv die generische Programmierung das Problem des kartesischen Produkts löst.

Um darüber hinaus auf beliebig geformten (nicht rechteckigen) Regionen arbeiten zu können, definieren wir zu jeder der obigen fünf Funktionen eine weitere Variante, die ein Masken-Bild zur Festlegung der Region benutzt. Das Masken-Bild enthält nur innerhalb der gewünschten Region Pixelwerte ungleich null. Nur bei diesen Pixeln wird die normale Funktion des Algorithmus ausgeführt. Alle übrigen Punkte werden übersprungen. Die Algorithmen mit Masken-Bild sind in ihrem Namen

durch das Suffix "If" gekennzeichnet. Die Funktion `initImageIf()` hat beispielsweise folgende Gestalt:

```
template <class ImageIterator1, class ImageIterator2, class T>
void initImageIf(ImageIterator1 upperleft, ImageIterator1 lowerright,
                 ImageIterator2 mask_upperleft, T const & value)
{
    Diff2D size = lowerright - upperleft;  // Bildgröße

    for(int y=0; y < size.y; ++y)
    {
        for(int x=0; x < size.x; ++x)
        {
            if(mask_upperleft(x, y) == 0) continue;  // ignoriere Punkte
                                                     // außerhalb der ROI
            upperleft(x, y) = value;  // schreibe Initialwert in Pixel (x,y)
        }
    }
}
```

Die Implementation von Masken-Varianten der übrigen Punktoperatoren erfolgt analog. Auch von dieser Erweiterung sind die Funktoren nicht betroffen, ebensowenig wie die Bilddatentypen und ihre Iteratoren.

Dennoch reichen die bis jetzt definierten Algorithmen noch nicht aus, um sämtliche Anforderungen an Punktoperationen abzudecken. Angenommen, wir wollen einen der Algorithmen auf ein RGB-Bild, jedoch nur auf einem Farbkanal anwenden. Dies kann erforderlich sein, wenn eine Operation nicht für ganze RGB-Tupel, wohl aber für jeden einzelnen Farbkanal definiert werden kann (man denke z.B. an die Bestimmung von Minima oder Maxima). Da die Zugriffsfunktionen eines Iterators für RGB-Bilder (`*iterator` bzw. `iterator(x,y)`) stets das ganze RGB-Tupel zurückgeben, können wir keinen der vorhandenen Funktoren für diesen Spezialfall benutzen. Zwar wäre es möglich, spezielle Funktoren zu schreiben, die ihre Operation nur auf einen Farbkanal anwenden, dies widerspricht aber wieder dem Prinzip der Kohäsion: ein Funktor wäre für mehrere Aufgaben zuständig (Auswahl eines Farbkanals *und* Transformation eines Werts). Daher ist diese Lösung nicht optimal. Bevor wir allerdings eine bessere Lösung für dieses Problem erarbeiten, wollen wir die Bilddatenstrukturen und ihre Pixeltypen genauer studieren.

4.5 Bilddatenstrukturen und ihre Iteratoren

Nachdem wir bisher im wesentlichen einfache Algorithmen und ihre Anforderungen untersucht haben, wollen wir uns jetzt den Bilddatenstrukturen zuwenden. Wir

werden dabei zunächst einen Bilddatentyp und die zugehörigen Iteratoren entwikkeln, der alle bisher definierten Anforderungen erfüllt. Sodann zeigen wir am Beispiel der Khoros-Bilddatenstruktur, wie man die erforderlichen Iteratoren auch für vorhandene Datenstrukturen bereitstellen kann. Dabei werden wir auf Probleme mit den Zugriffsoperationen der Iteratoren (operator* etc.) stoßen, die erklären, warum wir diese Operationen im Abschnitt 4.3 als optional gekennzeichnet haben.

4.5.1 Der Datentyp BasicImage und der BasicImageIterator

Der Datentyp BasicImage ist gedacht als Beispiel dafür, wie man mit minimalem Aufwand alle bisher definierten Anforderungen erfüllen kann. Der Typ BasicImage wird mit Hilfe des templates vector aus der C++-Standardbibliothek implementiert. Dies ist eine große Vereinfachung, denn vector übernimmt alle Aufgaben, die mit der Speicherverwaltung zusammenhängen. Wir verwenden hier die bekannte Technik, die Bildmatrix durch zwei geschachtelte Arrays (Objekte des Typs vector) zu implementieren, so daß wir mit doppelter Indexierung ([y][x]) auf jedes Pixel zugreifen können. Der innere vector (data_) enthält die eigentlichen Pixeldaten als 1-dimensionales Array, während der äußere (rows_) Zeiger auf das erste Pixel jeder Zeile speichert (Abbildung 5).

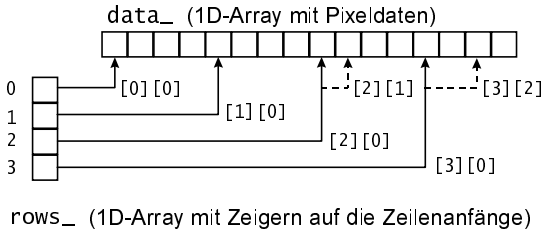


Abbildung 5: Implementierung einer Bildmatrix durch zwei geschachtelte Arrays. data_ enthält 1-dimensionale Pixeldaten, rows_ speichert Zeiger auf den Anfang jeder Zeile, so daß jedes Pixel durch zwei Indizes [y][x] referenziert wird. (Bildgröße: 4×4)

Dies gibt BasicImage die folgende Gestalt:

```
template <class PixelType>
class BasicImage
{
    typedef vector< PixelType >            DataVector;
    typedef vector< DataVector::iterator > RowStartVector;

    DataVector     data_;
    RowStartVector  rows_;
    int width_, height_;
```

```
  public:
    // der Iterator von vector erfüllt die Anforderungen des ScanOrderIterator
    typedef typename DataVector::iterator  ScanOrderIterator;

    // der 2-dimensionale Bilditerator wird im Anschluß definiert
    typedef BasicImageIterator<PixelType>  ImageIterator;

    // Konstruktor initialisiert die Bildgröße und den initialen Pixelwert
    BasicImage(int width, int height, PixelType init = PixelType() )
    : width_(width), height_(height),
      data_(width*height, init),
      rows_(height)
    {
        DataIterator d = data_.begin();
        // speichere einen Iterator auf das erste Pixel von Zeile i in rows_[i]
        for(int i=0; i<height; ++i, d+=width)    rows_[i] = d;
    }

    // Abfrage der Bildgröße
    int width() const  { return width_; }
    int height() const { return height_; }

    // Zugriff auf das Pixel (x, y)
    PixelType & getPixel(int x, int y) {  return rows_[y][x];  }

    // ScanOrderIteratoren auf Anfang und hinter das Ende der Pixelsequenz
    ScanOrderIterator begin() { return data_.begin(); }
    ScanOrderIterator end()   { return data_.end(); }

    // ImageIteratoren auf die linke obere Ecke des Bildes und
    // auf die Position rechts unterhalb der rechten unteren Ecke
    ImageIterator upperLeft()  { return ImageIterator(0, rows_.begin()); }
    ImageIterator lowerRight()  { return ImageIterator(width_, rows_.end()); }
};
```

Im Konstruktor erkennt man, wie die Daten für die doppelte Indizierung organisiert werden: der Vektor `data_` alloziert `width*height` Datenelemente und initialisiert sie mit dem in `init` übergebenen Anfangswert. Der Vektor `rows_` hingegen hat die Länge `height` und speichert in Element `i` jeweils den Iterator, der auf das Element `data_[i*width]`, also das erste Element der `i`-ten Zeile verweist. Dadurch können wir in der Funktion `getPixel()` die Daten über die Zweifachindexierung `rows_[y][x]` referenzieren.

Bei der Implementation der Iteratoren haben wir uns die Tatsache zunutze gemacht, daß die Klasse `vector` bereits einen Iterator anbietet. Da wir die Daten zeilenweise (engl. row-major) organisiert haben, erfüllt dieser Iterator die Anforderungen an einen `ScanOrderIterator`, und wir können die Funktionen `begin()` und `end()` der Bilddatenstruktur direkt an den Vektor `data_` delegieren. Auch die Implementation des 2-dimensionalen Iterators ist sehr einfach, da wir das vertikale Navigationsobjekt `MoveY` mit einem Iterator auf den Vektor `rows_` identifizieren

können, während als horizontales Navigationsobjekt MoveX einfach ein Integer verwendet werden kann. Das folgende Listing zeigt, wie auf diese Weise alle in Tabelle 2 aufgeführten Operationen implementiert werden können:

```
template <class PIXELTYPE >
class BasicImageIterator
{
    typedef vector< typename vector< PixelType >::iterator >  RowStartVector;

  public:
    typedef PIXELTYPE                          value_type;

    typedef int                                MoveX;
    typedef typename RowStartVector::iterator  MoveY;

    // durch die obige Wahl für MoveX und MoveY stehen die für x und y
    // geforderten Operationen automatisch zur Verfügung
    MoveX x;
    MoveY y;

    // Initialisierung an der Position (xinit, yinit)
    BasicImageIterator(MoveX xinit, MoveY yinit)
    : x(xinit), y(yinit)
    {}

    // Navigationsfunktionen mit zweidimensionalen Offsets
    BasicImageIterator & operator+=(Diff2D const & offset) {
        x += offset.width;
        y += offset.height;
        return *this;
    }

    BasicImageIterator & operator-=(Diff2D const & offset) {
        x -= offset.width;
        y -= offset.height;
        return *this;
    }

    BasicImageIterator operator+(Diff2D const & offset) const {
        return (BasicImageIterator(*this) += offset);
    }

    BasicImageIterator operator-(Diff2D const & offset) const {
        return (BasicImageIterator(*this) -= offset);
    }

    // Differenz von zwei ImageIteratoren
    Diff2D operator-( BasicImageIterator const & o) const {
        return Diff2D(x - o.x, y - o.y);
    }
```

```
        // Vergleich von zwei ImageIteratoren
        bool operator==( BasicImageIterator const & o) const {
            return x == o.x && y == o.y;
        }

        // Zugriffsfunktionen auf die Pixeldaten
        value_type & operator*() {
            return (*y)[x];
        }

        value_type & operator[](Diff2D const & d) {
            return y[d.height][x+d.width];
        }

        value_type & operator()(int dx, int dy) {
            return y[dy][x+dx];
        }
    };
```

Mit dieser Definition von `BasicImageIterator` wird die oben angegebene Implementation der Funktionen `BasicImage::upperLeft()` und `BasicImage::lowerRight()` verständlich, welche genau die im vorigen Abschnitt entwickelte Konvention für die Angabe von Regionenbegrenzungen realisieren. Mit diesen Definitionen können wir nun sofort Bilddatentypen für die skalaren Pixeltypen als template-Instanzen von `BasicImage` generieren:

```
typedef  BasicImage< unsigned char >   ByteImage;
typedef  BasicImage< int >             IntImage;
typedef  BasicImage< float >           FloatImage;
...
```

Durch Verwendung nutzerdefinierter Pixeltypen können wir ohne zusätzlichen Aufwand beliebig komplexe Bilddatentypen als Instanzen des allgemeinen templates `BasicImage` definieren, wie wir im nächsten Abschnitt anhand von RGB-Bildern demonstrieren wollen.

4.5.2 RGB-Werte und RGB-Bilder

Auch für RGB-Bilder läßt sich das soeben eingeführte template `BasicImage` verwenden. Allerdings benötigen wir vorher einen leistungsfähigen Typ für RGB-Werte. Wir wollen uns dabei nicht mit einer einfachen Struktur zufriedengeben, die nur die Farbwerte speichert, sondern wir wollen von vornherein auch einige arithmetische Operationen anbieten, so daß möglichst viele Punktoperationen auch auf RGB-Werte angewendet werden können.

Wir werden die arithmetischen Operationen komponentenweise definieren, d.h. jede Operation wird für den roten, grünen und blauen Kanal unabhängig voneinander ausgeführt. Auf diese Weise können Addition, Subtraktion und Multiplikation sowie eine Reihe unärer Operationen (Negation, Absolutbetrag, algebraische Operationen) definiert werden. Die Definition einer Division ist nicht sinnvoll, weil es Probleme mit der Division durch Null gibt: das Nullelement für RGB-Werte ist dasjenige Tupel, bei dem alle drei Farben den Wert Null haben. Eine verbotene Division durch Null tritt hingegen bereits auf, wenn nur eine der Komponenten Null ist. Dies widerspricht den Anforderungen an eine Divisionsalgebra, wo Division durch Null nur mit dem Nullelement auftreten darf (vergl. [BARTNACK94]). Wir definieren deshalb keine Division.

Wie bei den Bildern definieren wir die Klasse für RGB-Werte als template eines Komponententyps. Wir können dadurch ohne zusätzlichen Aufwand RGB-Werte mit unterschiedlichen Basistypen instanziieren. Durch Verwendung von template member functions können die verschiedenen Varianten von `RGBValue` automatisch ineinander umgewandelt werden:

```
template <class COMPONENTTYPE>
class RGBValue
{
    COMPONENTTYPE rgb_[3];  // die Farbwerte

  public:
    typedef COMPONENTTYPE  value_type;

    // Standardkonstruktor
    RGBValue() {}  // impliziert automatische Initialisierung des Array rgb_

    // Konstruktor mit expliziter Initialisierung der Farbkanäle
    RGBValue(COMPONENTTYPE r, COMPONENTTYPE g, COMPONENTTYPE b)
    {
        rgb_[0] = r;  rgb_[1] = g;  rgb_[2] = b;
    }

    // Kopierkonstruktur (gleichzeitig Typumwandlung)
    template <class OTHERTYPE>
    RGBValue( RGBValue<OTHERTYPE> const & o)
    {
        rgb_[0] = o.rgb_[0];  rgb_[1] = o.rgb_[1];  rgb_[2] = o.rgb_[2];
    }

    // Zuweisung
    template <class OTHERTYPE>
    RGBValue & operator=( RGBValue<OTHERTYPE> const & o)
    {
        rgb_[0] = o.rgb_[0];  rgb_[1] = o.rgb_[1];  rgb_[2] = o.rgb_[2];
        return *this;
    }
```

```
        // Vergleich
        bool operator==( RGBValue const & o) const
        {
            return (rgb_[0] == o.rgb_[0] &&
                    rgb_[1] == o.rgb_[1] &&  rgb_[2] == o.rgb_[2]);
        }

        // komponentenweise Addition
        RGBValue & operator+=( RGBValue const & o)
        {
            rgb_[0] += o.rgb_[0];  rgb_[1] += o.rgb_[1];  rgb_[2] += o.rgb_[2];
            return *this;
        }

        RGBValue operator+(RGBValue const & o) const
        {
            return (RGBValue(*this) += o);
        }

        ... // Subtraktion und Multiplikation analog

        // Zugriff auf die Rotkomponente (analog für die anderen Farben)
        COMPONENTTYPE & red()       {  return rgb_[0];  }
        COMPONENTTYPE   red() const {  return rgb_[0];  }
        void setRed(COMPONENTTYPE & red) {  rgb_[0] = red;  }
        ...
    };
```

Mit dieser Definition können RGB-Bilder ebenfalls als Instanzen des templates `BasicImage` definiert werden:

```
typedef  BasicImage< RGBValue< byte > >    ByteRGBImage;
typedef  BasicImage< RGBValue< float > >   FloatRGBImage;
...
```

Ganz analog können wir für andere strukturierte Pixeltypen vorgehen, wie z.B. 2D-Vektoren, die wir unter anderem für die Definition eines Typs „Gradientenbild“ verwenden können: zunächst definieren wir den neuen Pixeltyp `Vector2D` sowie die für diesen Typ sinnvollen Operationen, und danach instanziieren wir das template `BasicImage` für diesen Pixeltyp. Wir erkennen, daß es sich bei „Pixeltypen“ und „Bilddatentypen“ um zwei Abstraktionsachsen handelt, die unabhängig voneinander erweitert werden können und deren Realisierungen man beliebig miteinander kombinieren kann. Zwar haben wir auf der Achse „Bilddatentypen“ bisher nur eine Variante (nämlich `BasicImage`) implementiert, aber auch hier können wir ohne weiteres weitere Varianten hinzufügen, wie z.B. ein 3-dimensionales Bild (Datenvolumen). Der Code für jede benötigte Kombination wird vom Compiler automatisch generiert, das Problem des kartesischen Produkts ist somit auch in diesem Fall gelöst.

4.5.3 Adaption existierender Bilddatentypen

Eine der größten Stärken der generischen Programmierung besteht darin, daß sie einen allmählichen Übergang von anderen Programmiermethoden ermöglicht. Man kann also für eine gewisse Zeit oder dauerhaft eine andere Programmiermethode neben der generischen Programmierung weiter benutzen. Dies erfordert unter anderem, daß neue generische Algorithmen auch auf alte, schon vorhandene Datenstrukturen anwendbar sind. Wir wollen deshalb hier einige Beispiele dafür angeben, wie man für vorhandene Bilddatenstrukturen die notwendigen Iteratoren bereitstellen kann.

Als erstes wollen wir uns einen Bilddatentyp ansehen, der seine interne Implementation vollständig kapselt, wie beispielsweise die folgende abstrakte Basisklasse für Bilder:

```
class AbstractByteImage {
    virtual int width() const = 0;
    virtual int height() const = 0;
    virtual byte & getPixel(int x, int y) = 0;
};
```

Wir haben hier nicht die Möglichkeit, wie beim `BasicImageIterator` auf interne Details der Bildklasse (wie den `RowStartVector`) zuzugreifen. Statt dessen müssen wir uns auf den Aufruf der Zugriffsfunktion `getPixel()` beschränken. Die Definition eines 2-dimensionalen Iterators für diesen Fall ist dennoch sehr einfach: der Iterator muß einen Zeiger auf das Bild speichern, und die x- und y-Navigatoren werden einfach als `int` implementiert:

```
class AbstractByteImageIterator {
    AbstractByteImage * image_;

  public:
    typedef  int  MoveX;
    typedef  int  MoveY;

    MoveX x;
    MoveY y;

    AbstractByteImageIterator(AbstractByteImage * i)
    : image_(i), x(0), y(0)
    {}

    byte & operator*() { return image->getPixel(x, y); }

    byte & operator()(int dx, int dy) { return image->getPixel(x+dx, y+dy); }

    byte & operator[](Diff2D const & d) {
        return image->getPixel(x+d.width, y+d.width);
    }
    ... // übrige Funktionen sind mit BasicImageIterator identisch
};
```

Dieses Design bietet die einzigartige Möglichkeit, generische Algorithmen auch dann noch anwenden zu können, wenn die unterliegenden Datenstrukturen ursprünglich nicht dafür vorgesehen waren und auch nicht entsprechend geändert werden können.

Ein anderer wichtiger Fall ist die Adaption traditioneller, strukturierter Datentypen. Wir greifen hier als Beispiel den Bilddatentyp des Systems Candela heraus [CANDELA98]. Dieses Bildformat ist eine C-Struktur. Der Datenzugriff ist jedoch in Zugriffsfunktionen gekapselt. Der hier relevante Teil der Schnittstelle ist folgendermaßen deklariert:

```
typedef struct { ... } * canPic;  // Zeiger auf ein Candela-Bild

// Kennung für die verschiedenen unterstützten Pixeltypen
typedef enum {
    canUnknown,                   // unbekannter Pixeltyp
    canI1,                        // boolsche Pixel     (1 bit)
    canI8, canI16, canI32,        // ganzzahlige Pixel  (byte, short, int)
    canF32, canF64                // reellwertige Pixel (float, double)
} canPicFormat;

canPicFormat can_format(canPic);  // Abfrage des Pixeltyps

char * canpic_pixels(canPic);     // Zeiger auf die Bilddaten (muß noch in Zeiger
                                  // auf korrekten Pixeltyp konvertiert werden)

int can_xsize(canPic);            // Breite und
int can_ysize(canPic);            // Höhe des Bildes
```

Auch für diesen Typ läßt sich der 2-dimensionale Iterator sehr leicht implementieren. Wir definieren die x- und y-Navigatoren wieder als `int` und verwalten intern einen Zeiger auf die Bilddaten, den wir im Konstruktor aus dem übergebenen `canPic` extrahieren. Um alle Pixeltypen mit einer Implementation abdecken zu können, definieren wir den Iterator als template des Pixeltyps:

```
template <class PIXELTYPE>
struct CandelaImageIterator {
    PIXELTYPE * data_;
    int width_;

  public:
    typedef  int  MoveX;
    typedef  int  MoveY;

    MoveX x;
    MoveY y;
```

```
    CandelaImageIterator(canPic image)
    : x(0), y(0),
      data_((PIXELTYPE *) canpic_pixels(image)), // erfrage Zeiger auf Bilddaten
      width_( can_xsize(image) )                 // erfrage Breite des Bildes
    {}

    PIXELTYPE & operator*() {  return data_[x+y*width_];  }
    PIXELTYPE & operator()(int dx, int dy) { return data_[x+dx+(y+dy)*width_]; }

    PIXELTYPE & operator[](Diff2D const & d)
    {
        return data_[x + d.x + (y + d.y)*width_];
    }

    ... // übrige Funktionen sind mit BasicImageIterator identisch
};
```

Die Auswahl des richtigen Iterators hängt dann vom jeweiligen Pixeltyp der Bilddatenstruktur ab, die wir mit der Funktion can_format() zur Laufzeit abfragen können. In einer entsprechenden Fallunterscheidung können wir jeweils den richtigen Iterator instanziieren:

```
canPic image;
...
switch(can_format(image))
{
    case canI8:
        CandelaImageIterator<byte> upperleft(image),
            lowerright = upperleft + Diff2D(can_xsize(image), can_ysize(image);
        ...
        break;

    case canF32:
        CandelaImageIterator<float> upperleft(image),
            lowerright = upperleft + Diff2D(can_xsize(image), can_ysize(image);
        ...
        break;

    ... // übrige Fälle
}
```

Allerdings beobachten wir hier wieder ein exponentielles Anwachsen der Variantenzahl aufgrund des Problems des kartesischen Produkts. Man kann die Variantenzahl durch einen bottleneck-Ansatz verringern und zunächst alle Bilder in ein einheitliches Format transformieren. In Candela steht dafür die Funktion convert_to_f32() zur Verfügung:

```
canPic image;
...
```

```
image = convert_to_f32(image);

CandelaImageIterator<float> upperleft(image),
    lowerright = upperleft + Diff2D(can_xsize(image), can_ysize(image);
```

Wir sehen also, daß auch in diesem Fall die Anpassung eines vorhandenen Bilddatentyps an die Anforderungen generischer Algorithmen nicht schwierig ist. Es gibt jedoch ein Problem, wenn wir von skalaren Pixeltypen auf multispektrale Bilder übergehen. In Candela werden diese Bilder als Array von skalaren Bildern dargestellt, z.B. bei RGB-Bildern:

```
typedef canPic RGBPic[3];
```

Ein RGB-Bild ist hier ein Array der Länge 3 von skalaren Bildern. Das bedeutet, daß kein expliziter Typ für einzelne RGB-Tupel (ähnlich dem oben definierten `RGBValue`) existiert – ein RGB-Wert wird implizit durch die an der selben Position gespeicherten skalaren Werte der drei Bänder[14] repräsentiert. Es zeigt sich nun, daß dieses Vorgehen nicht damit verträglich ist, wie der C++ Standard die Zugriffsfunktionen der Iteratoren definiert: es ist nicht möglich, die geforderten Zugriffsfunktionen (`operator*`, `operator[]`) zu implementieren, wenn der Rückgabetyp dieser Funktionen nicht explizit existiert. Wir werden deshalb im nächsten Abschnitt die Methode des Datenzugriffs über Iteratoren verbessern.

4.6 Zugriffsobjekte

Wir hatten in den letzten Abschnitten Probleme mit den Zugriffsfunktionen der Iteratoren (`operator*`, `operator[]`, `operator()`) angedeutet, die wir hier genauer erläutern und lösen wollen.

Den Zugriffsfunktionen der Iteratoren, wie sie die C++-Standardbibliothek definiert, liegen zwei wichtige Annahmen zugrunde:

[14] Zur Terminologie: Wir verwenden den Begriff „Farbkanal“ zur Beschreibung der abstrakten Struktur eines Farbbildes. Der Begriff „Farbband“ soll hingegen der speziellen Implementationstechnik vorbehalten bleiben, bei der jeder Farbe ein separates Array bzw. Bild zugeordnet wird, wo es also keine expliziten RGB-Tupel gibt.

1. Die Zugriffsfunktionen geben eine Referenz auf ein in der unterliegenden Datenstruktur gespeichertes Objekt zurück. Dies hat den Vorteil, daß man die Zugriffsfunktionen auf der rechten *und* auf der linken Seite einer Zuweisung verwenden kann. Andererseits folgt daraus die Forderung, daß die Datenstruktur Objekte exakt des Typs, der zurückgegeben wird, speichern muß.
2. Die Algorithmen sind stets für das vollständige Objekt definiert. Werden nur bestimmte Attribute des Objekts benötigt, muß dies im Algorithmus oder in einem Funktor explizit kodiert werden.

Beide Annahmen lassen sich bzgl. Computer Vision nicht aufrechterhalten. Dies läßt sich am einfachsten mit Hilfe von RGB-Bildern zeigen. Wir beginnen mit der ersten Annahme und betrachten dazu noch einmal das RGB-Bild aus dem Candela-System:

```
typedef canPic RGBPic[3];
```

Ein RGB-Bild ist ein Array von drei skalaren Bilder. Diese Struktur ist logisch äquivalent mit einem einzelnen Bild, das RGB-Tupel enthält, aber eine explizite Datenstruktur für RGB-Werte gibt es hier nicht. Wir können dennoch einen 2-dimensionalen Iterator gemäß Tabelle 2 definieren, er muß aber andere Zugriffsfunktionen als die Iteratoren der C++-Standardbibliothek erhalten. Dies ist der Grund dafür, daß wir die Zugriffsfunktionen in Tabelle 2 als *optional* markiert hatten. Der Iterator für Candelas RGB-Bilder hat folgende Gestalt:

```
template <class PIXELTYPE>
struct CandelaRGBImageIterator {
    PIXELTYPE * data_[3];        // data_ jetzt Array mit drei Spektralkanälen
    int width_;

  public:
    typedef  int  MoveX;
    typedef  int  MoveY;

    MoveX x;
    MoveY y;

    CandelaRGBImageIterator(RGBPic image)
    : x(0), y(0),
    width_( can_xsize(image) )                // erfrage Breite des Bildes
    {
        // speichere Zeiger auf die Bilddaten der drei Farbkanäle
        data_[0] = (PIXELTYPE *) canpic_pixels(image[0]);
        data_[1] = (PIXELTYPE *) canpic_pixels(image[1]);
        data_[2] = (PIXELTYPE *) canpic_pixels(image[2]);
    }

    // Referenzen auf die einzelnen Farbkanäle
    PIXELTYPE & red()   {  return data_[0][x+y*width_];  }
    PIXELTYPE & green() {  return data_[1][x+y*width_];  }
    PIXELTYPE & blue()  {  return data_[2][x+y*width_];  }
```

```
    // Rückgabe eines RGB-Tupels als Wert (nicht als Referenz)
    RGBValue<PIXELTYPE> get() {
        return RGBValue<PIXELTYPE>(red(), green(), blue());
    }

    // schreiben eines RGB-Tupels über explizite set()-Funktion
    void set(RGBValue<PIXELTYPE> const & v) {
        red()   = v.red();
        green() = v.green();
        blue()  = v.blue();
    }

    ... // weitere Funktionen
};
```

Da der logische Datentyp „RGB-Tupel“ nicht als Tupel gespeichert ist, haben wir keine Zugriffsfunktion definiert, die ein RGB-Tupel *per Referenz* zurückgibt. Es gibt statt dessen drei Funktionen, die Referenzen auf die einzelnen Farbwerte zurückgeben, sowie zwei Funktionen, die separate Rot-, Grün- und Blau-Werte in ein Objekt vom Typ `RGBValue` konvertieren und umgekehrt. Die Definition eines `operator*`, der einen `RGBvalue` per Referenz zurückgibt, ist nicht möglich. Daher kann der Ausdruck `*iterator` *nicht auf der linken Seite einer Zuweisung benutzt werden*, wie es für die bisher definierten Algorithmen erforderlich ist.

Auch die zweite der oben erwähnten Annahmen ist nicht immer erfüllt. Nehmen wir an, wir haben die Aufgabe, den roten Kanal eines RGB-Bildes in den blauen Kanal eines anderen zu kopieren. Obwohl es nahe läge, kann die Funktion `copyImage()` dafür nicht benutzt werden, denn sie kopiert immer die gesamte Datenstruktur. Wir müssen also `transformImage()` verwenden und einen Funktor für diese Operation schreiben. Wird die Aufgabe so modifiziert, daß die Farbwerte außerdem noch invertiert werden sollen, so benötigen wir einen weiteren Funktor. Ebenso benötigen wir einen neuen Funktor, wenn wir statt des roten Kanals den grünen kopieren müssen. Die Konzepte „Auswahl eines Farbkanals“ und „Ausführen einer Operation“ sind also nicht unabhängig voneinander. Auch bei der Faltung treten ähnliche Probleme auf. Mit den bisher definierten Konzepten ist es beispielsweise nicht möglich, auf jeden Farbkanal einen anderen Faltungskern anzuwenden.

Analysieren wir den Grund für diese Schwierigkeiten, so erkennen wir, daß die Iteratordefinitionen der C++-Standardbibliothek das Kohäsionsprinzip (vgl. Abschnitt 2.1.1) verletzen. Ein Iterator hat zwei Aufgaben zu erfüllen: er muß auf einer Datenstruktur navigieren, und er muß Zugriffsfunktionen auf das aktuelle Element bereitstellen. Die obigen Beispiele zeigen, daß diese Aufgaben unabhängig voneinander variieren können. Darüber hinaus müssen die Zugriffsfunktionen selbst auch zwei Funktionen erfüllen: sie dienen sowohl zum Lesen als auch zum Schreiben der Daten (Verwendung von `*iterator` auf der rechten bzw. linken Seite einer Zuwei-

sung). Es ist ungünstig, die Vereinigung dieser Aufgaben in einem Baustein zu erzwingen, wie es die C++-Standardbibliothek tut. Wir können unseren Entwurf verbessern, indem wir den Datenzugriff in eigene Bausteine auslagern. Wir wollen diese Bausteine *Zugriffsobjekte* (engl. data accessor) nennen. Wir folgen damit einem Vorschlag von [KÜHLWEIHE97] und passen deren Design an die Computer Vision-Anforderungen an. Insbesondere erweitern wir die Zugriffsobjekte um Funktionen zum Zugriff an einer anderen als der aktuellen Position, so daß wir nicht nur den `operator*`, sondern auch den `operator[]` kapseln können.

Ein Zugriffsobjekt hat getrennte Methoden `get()` und `set()` für das Lesen und Schreiben der Daten, sowie eine lokale Typdefinition `value_type`, die den Rückgabetyp der Funktion `get()` angibt. Der Iterator wird diesen Methoden als Argument übergeben und bestimmt somit, an welcher Position einer Datenstruktur die aktuellen Daten gelesen bzw. geschrieben werden. Das Zugriffsobjekt liest oder schreibt dann an dieser Stelle die gewünschten Attribute. Dadurch wird eine bessere Aufgabenteilung erreicht: der Iterator verliert die Zuständigkeit für den Datenzugriff und ist nur noch für die Navigation verantwortlich. Beide Funktionalitäten lassen sich nun unabhängig voneinander variieren.

Jedes Zugriffsobjekt kapselt eine bestimmte Vorgehensweise, mit Hilfe eines Iterators auf die Daten zuzugreifen. Wir können deshalb in allen Fällen, in denen auf die Daten in gleicher Weise zugegriffen wird, dasselbe Zugriffsobjekt benutzen. Wir implementieren folglich die Zugriffsfunktionen als template member functions. Das einfachste Zugriffsobjekt ruft intern einfach die Zugriffsfunktionen des Iterators auf:

```
template <class PIXELTYPE>
struct StandardAccessor
{
    typedef PIXELTYPE value_type;

    // Lesen der Daten an der aktuellen Position über operator*
    template <class ITERATOR>
    value_type const & get(ITERATOR & i) const {  return *i;  }

    // Schreiben der Daten an der aktuellen Position operator*
    template <class ITERATOR>
    void set(value_type const & v, ITERATOR & i) const {  *i = v;  }

    // Lesen der Daten an einer anderen Position über operator[]
    template <class ITERATOR, class Difference >
    value_type get(ITERATOR & i, Difference const & d) const {
        return i[d];
    }

    // Schreiben der Daten an einer anderen Position über operator[]
    template <class ITERATOR, class Difference >
    void get(value_type const & v, ITERATOR & i, Difference const & d) const {
        i[d] = v;
    }
};
```

Man erkennt, daß wir auch die Indexoperation operator[] gekapselt haben: die zweite Variante von get() und set() akzeptiert ein Argument vom Typ Difference, der ein template-Parameter ist. Bei linearen Iteratoren verlangt der operator[] ein Argument von Typ int, in diesem Falle wird Difference also mit int instanziiert. Bei 2-dimensionalen Iteratoren hingegen müssen wir ein Objekt vom Typ Diff2D übergeben. Durch die Definition von Difference als template-Parameter haben dennoch beide Varianten die gleiche Schnittstelle und die gleiche Implementation.

Wir verwenden das Zugriffsobjekt StandardAccessor immer dann, wenn der Iterator die Standardzugriffsfunktionen anbietet und wenn der Algorithmus auf das gesamte Datenobjekt zugreifen will. Dies wird sicherlich der Normalfall sein. Der StandardAccessor funktioniert hingegen nicht für den CandelaRGBImageIterator. Für diesen Iterator definieren wir ein anderes Zugriffsobjekt:

```
template <class COMPONENTTYP>
struct CandelaRGBAccessor
{
    typedef RGBValue< COMPONENTTYP > value_type;

    // Lesen der Daten an der aktuellen Position über ITERATOR::get()
    template <class ITERATOR>
    value_type const & get(ITERATOR & i) const {  return i.get();  }
    // Schreiben der Daten an der aktuellen Position über ITERATOR.set()
    template <class ITERATOR>
    void set(value_type const & v, ITERATOR & i) const {  i.set(v);  }

    // Lesen der Daten an einer anderen Position über ITERATOR::get()
    template <class ITERATOR, class Difference >
    value_type get(ITERATOR & i, Difference const & d) const {
        return i.get(d);
    }

    // Schreiben der Daten an einer anderen Position über ITERATOR.set()
    template <class ITERATOR, class Difference >
    void get(value_type const & v, ITERATOR & i, Difference const & d) const {
        i.set(v, d);
    }
};
```

Beide Zugriffsobjekte haben, im Gegensatz zu den Iteratoren, eine identische Schnittstelle. Der Unterschied zwischen den zugrundeliegenden Bilddatentypen bleibt den Algorithmen somit verborgen.

Um die Verwendung der Zugriffsobjekte zu vereinfachen, ordnen wir jedem Iteratortyp ein Standard-Zugriffsobjekt (DefaultAccessor) zu. Dieses Objekt sollte so gewählt werden, daß seine Funktionen sich auf das gesamte logische Datenobjekt der zugrundeliegenden Datenstruktur beziehen. Die beiden soeben eingeführten

Zugriffsobjekte erfüllen diese Forderung. Das zu jedem Iterator gehörende Zugriffsobjekt kann mit Hilfe einer traits-Klasse `IteratorTraits` erfragt werden:

```
template <class Iterator> struct IteratorTraits;  // forward-Deklaration

// partielle template-Spezialisierung für BasicImageIterator
template <class PIXELTYPE >
struct IteratorTraits < BasicImageIterator<PIXELTYPE> >
{
    typedef StandardAccessor< PIXELTYPE > DefaultAccessor;
};

// partielle template-Spezialisierung für CandelaRGBImageIterator
template <class PIXELTYPE >
struct IteratorTraits < CandelaRGBImageIterator <PIXELTYPE> >
{
    typedef CandelaRGBAccessor< PIXELTYPE > DefaultAccessor;
};

... // usw. für die übrigen Iteratoren
```

Diese traits-Klassen bestimmen, daß der DefaultAccessor eines `BasicImageIterator` die Klasse `StandardAccessor` ist, während im Zusammenhang mit dem `CandelaRGBImageIterator` die Klasse `CandelaRGBAccessor` verwendet werden soll.

Über die Standard-Zugriffsobjekte hinaus eröffnet sich eine Reihe neuer Möglichkeiten durch die Einführung spezialisierter Zugriffsobjekte. Wir können `get()` und `set()` zum Beispiel so implementieren, daß nur ein Farbkanal an den Algorithmus übergeben wird. Der Algorithmus „sieht“ somit ein skalares Bild, obwohl die unterliegende Datenstruktur ein RGB-Bild ist. Für den roten Kanal sieht dies beispielsweise folgendermaßen aus:

```
template <class COMPONENTTYP>
struct RedChannelAccessor
{
    typedef COMPONENTTYP value_type;

    // Lesen der Daten an der aktuellen Position über operator*
    template <class ITERATOR>
    value_type const & get(ITERATOR & i) const {
        return (*i).red();     // extrahiere den roten Kanal
    }

    // Schreiben der Daten an der aktuellen Position über operator*
    template <class ITERATOR>
    void set(value_type const & v, ITERATOR & i) const {
        (*i).setRed(v);        // schreibe den roten Kanal
    }

    ... // analog mit operator[]
};
```

Dieses Zugriffsobjekt kann verwendet werden, wenn ein Algorithmus nur auf dem roten Kanal eines RGB-Bildes arbeiten soll, und außerdem das Bild die Pixel als `RGBValue` speichert und der Iterator die Standardzugriffsfunktionen anbietet. Für andere Bilddatentypen muß die Implementation entsprechend geändert werden (der `CandelaRGBImageIterator` verlangt z.B. `i.red()` anstelle von `(*i).red()`), die Schnittstelle des Zugriffsobjekts bleibt jedoch erhalten.

Ebenso interessant ist die Möglichkeit, in den Funktionen des Zugriffsobjektes aus gespeicherten Attributen andere logische Attribute zu berechnen, die nicht explizit gespeichert sind. Wir können so beispielsweise zwischen RGB- und Grauwerten transformieren. Die `set()`-Funktion setzt den gegebenen Grauwert in allen Farbkanälen, und die `get()`-Funktion berechnet "on-the-fly" die Luminanz eines RGB-Wertes. Wenn wir hierfür einfach den Mittelwert der Farbintensitäten bilden, erhalten wir folgendes Zugriffsobjekt:

```
template <class COMPONENTTYPE>
struct TransformBetweenRGBAndGray {
    typedef COMPONENTTYPE value_type;

    template <class RGBITERATOR>
    value_type get(RGBITERATOR & i) const {
        return ((*i).red() + (*i).green() + (*i).blue()) / 3;
    }

    template <class RGBITERATOR>
    void set(value_type const & v, RGBITERATOR & i) const {
        (*i).setRed(v); (*i).setGreen(v); (*i).setBlue(v);
    }
};
```

Im allgemeinen führt der Einsatz von Zugriffsobjekten nicht zu einem Geschwindigkeitsverlust, weil die meisten Compiler die `inline` implementierten Zugriffsfunktionen so optimieren, als ob kein Zugriffsobjekt vorhanden wäre. Der konsequente Einsatz von Zugriffsobjekten rechtfertigt die in Tabelle 2 vorgenommene Kennzeichnung der Zugriffsoperationen der Iteratoren (`operator*`, `operator[]` und `operator()`) als optional, da diese Funktionen nicht von allen Datenstrukturen implementiert werden können und die Algorithmen sie nicht mehr direkt aufrufen müssen.

4.7 Punktoperationen auf Bildern (3)

In diesem Abschnitt wollen wir anhand der Punktoperationen zeigen, wie man Algorithmen für die Benutzung von Zugriffsobjekten modifizieren muß, und wie man die modifizierten Algorithmen einsetzt. Die Idee besteht dabei darin, zu jedem Iteratorargument ein zugehöriges Zugriffsobjekt zu übergeben. Unsere fünf grundlegenden Punktoperationen erhalten damit die folgende Schnittstelle:

```
// Initialisierung
template <class ImageIterator, class Accessor, class T>
void initImage(ImageIterator upperleft, ImageIterator lowerright, Accessor a,
               T const & value);

// Transformation
template <class ImageIterator1, class Accessor1,
          class ImageIterator2, class Accessor2, class UnaryFunction >
void transformImage(ImageIterator1 src_ul, ImageIterator1 src_lr, Accessor1 srca,
                    ImageIterator2 dest_ul, Accessor2 desta, UnaryFunction f);

// Kopie
template <class ImageIterator1, class Accessor1,
          class ImageIterator2, class Accessor2 >
void copyImage(ImageIterator1 src_ul, ImageIterator1 src_lr, Accessor1 srca,
               ImageIterator2 dest_ul, Accessor2 desta);

// Kombination
template < class ImageIterator1, class Accessor1,
           class ImageIterator2, class Accessor2,
           class ImageIterator3, class Accessor3, class BinaryFunction >
void
combineTwoImages(ImageIterator1 src1_ul, ImageIterator1 src1_lr, Accessor1 src1a,
                 ImageIterator2 src2_ul, Accessor2 src2a,
                 ImageIterator3 dest_ul, Accessor3 desta,
                 BinaryFunction f);

// Analyse
template <class ImageIterator, class Accessor, class UnaryAnalyser >
void
inspectImage(ImageIterator ul, ImageIterator lr, Accessor a, UnaryAnalyser & f);
```

In analoger Weise ändert man die Schnittstellen aller Algorithmen, wie z.B. der Punktoperationen mit Maskenbild (`...ImageIf`) und der Faltungsoperationen. Allerdings haben sich damit die Schnittstellen stark verkompliziert, `combineTwoImages` benötigt beispielsweise bereits acht Parameter. Wir wollen im nächsten Abschnitt einen Weg zeigen, wie die Schnittstelle vereinfacht werden kann, ohne ihre Flexibilität aufzugeben.

Als Beispiel für die Implementation der modifizierten Funktionen wollen wir den Quellcode der Funktionen `transformImage()` und `copyImage()` angeben. Die notwendigen Modifikationen der übrigen Algorithmen sind daraus leicht erkennbar:

```
// Kopie
template <class ImageIterator1, class Accessor1,
          class ImageIterator2, class Accessor2 >
void copyImage(ImageIterator1 src_ul, ImageIterator1 src_lr, Accessor1 srca,
               ImageIterator2 dest_ul, Accessor2 desta)
{
    for(;src_ul.y < src_lr.y; ++ src_ul.y, ++ dest_ul.y)
    {
        ImageIterator1 scurrent = src_ul;
        ImageIterator2 dcurrent = dest_ul;

        for(; scurrent.x < src_lr.x; ++scurrent.x, ++dcurrent.x)
        {
            // Kopieren des aktuellen Punktes:
            // Lesen des aktuellen Wertes über srca, Schreiben über desta
            desta.set( srca.get(scurrent), dcurrent);
        }
    }
}

// Transformation
template <class ImageIterator1, class Accessor1,
          class ImageIterator2, class Accessor2, class UnaryFunction >
void transformImage(ImageIterator1 src_ul, ImageIterator1 src_lr, Accessor1 srca,
                    ImageIterator2 dest_ul, Accessor2 desta, UnaryFunction f);
{
   Diff2D size = src_lr - src_ul;  // Bildgröße
   Diff2D current(0,0);

   for(; current.y < size.y; ++current.y)
   {
       for(current.x=0; current.x < size.x; ++current.x)
       {
           // Transformation des aktuellen Punktes

           // anstelle des operator[] der Iteratoren werden jetzt die
           // Funktionen des Zugriffsobjekts aufgerufen, die eine Difference
           // akzeptieren (also "get(iterator, current)" und
           // "set(value, iterator,current)")
           desta.set(f(srca.get(src_ul, current)), dest_ul, current);
             //              ^ Lesen der Quelldaten
             //      ^ Aufruf des Funktors
             //   ^ Schreiben des Ergebnisses
       }
   }
}
```

Die modifizierten Funktionen bieten nun weitere Möglichkeiten, Bilder flexibel zu manipulieren, zum Beispiel:

```
ByteRGBImage rgb1(100, 200), rgb2(100, 200);
ByteImage    gray(100, 200);
RGBPic       candelaRGB;
for(int i=0; i<3; ++i)  candelaRGB[i] = can_palloc(100, 200, canI8);
...

// Erzeugung einiger Zugriffsobjekte
IteratorTraits< ByteRGBImage::Iterator >::DefaultAccessor rgbaccessor;
IteratorTraits< ByteImage::Iterator >::DefaultAccessor    grayaccessor;
CandelaRGBAccessor<byte>                   canrgbaccessor;
RedChannelAccessor<byte>                   redaccessor;
BlueChannelAccessor<byte>                  blueaccessor;
TransformBetweenRGBAndGray<byte>           rgb2gray, gray2rgb;

// Transformiere RGB-Bild in Graubild
copyImage(rgb1.upperLeft(), rgb1.lowerRight(), rgb2gray,
          gray.upperLeft(), grayaccessor);

// Transformiere Graubild in RGB-Bild
copyImage(gray.upperLeft(), gray.lowerRight(), grayaccessor,
          rgb2.upperLeft(), gray2rgb);

// Addiere zwei RGB-Bilder und speichere Resultat in einem Candela-RGB-Bild
combineTwoImages(rgb1.upperLeft(), rgb1.lowerRight(), rgbaccessor,
                 rgb2.upperLeft(), rgbaccessor,
                 CandelaRGBImageIterator<byte>(candelaRGB), canrgbaccessor,
                 plus<RGBVAlue<byte> >());

// Addiere nur die roten Kanäle der RGB-Bilder
combineTwoImages(rgb1.upperLeft(), rgb1.lowerRight(), redaccessor,
                 rgb2.upperLeft(), redaccessor, rgb2.upperLeft(), redaccessor,
                 plus<RGBVAlue<byte> >());

// Kopiere den roten Kanal von rgb1 in den blauen Kanal von rgb2
copyImage(rgb1.upperLeft(), rgb1.lowerRight(), redaccessor,
          rgb2.upperLeft(), blueaccessor);

// setze alle Pixel zu schwarz, deren roter Kanal den Wert 0 hat
// (das Bild ist hier glechzeitig Maskenbild)
RGBValue<byte> black(0,0,0);
initImageIf(rgb1.upperLeft(), rgb1.lowerRight(), rgbaccessor,
            rgb1.upperLeft(), redaccessor, black);
```

Viele Algorithmen, die bisher eine eigene Funktion erforderten (wie z.B. die Konvertierung zwischen RGB und Grauwerten) können nun als Spezialfall einer der fünf Grundfunktionen mit einem geeigneten Zugriffsobjekt ausgedrückt werden. Die höhere Flexibilität wird allerdings mit einer etwas komplexeren Handhabung

erkauft. Wir werden in Abschnitt 4.9 mit den Parameterobjekten eine Methode vorstellen, diese zusätzliche Komplexität zu kapseln, falls sie nicht benötigt wird.

4.8 Die Faltungsoperation auf Bildern (2)

Nach der Einführung der 2-dimensionalen Iteratoren und der Zugriffsobjekte sind wir nunmehr auch in der Lage, die 2-dimensionale Faltungsoperation zu implementieren. Wir erinnern daran, daß wir dazu den folgenden Ausdruck in jedem Punkt des Bildes berechnen müssen:

$$(f * g)(x,y) = \sum_{i=-I}^{K} \sum_{j=-J}^{M} g(-i,-j) f(x+i, y+j) \tag{4.3}$$

Anschaulich gesprochen bedeutet dies, daß wir an jedem Punkt den Faltungskern wie ein Fenster über das Bild legen, die jeweils übereinander liegenden Werte von Kern und Bild miteinander multiplizieren und die Ergebnisse aufsummieren. Bei der Implementation ergibt sich jedoch eine Schwierigkeit: bei Punkten nahe des Bildrands liegt der Faltungskern teilweise außerhalb des Bildes. Zu den außerhalb liegenden Werten des Kerns sind keine korrespondierenden Bildwerte definiert, mit denen man multiplizieren könnte. Dies ist das sogenannte *Randproblem*.

Wir werden das Randproblem in den Abschnitten 5.1 und 5.5 ausführlich behandeln. Hier wollen wir zunächst annehmen, daß der zu bearbeitende Bereich so gewählt wird, daß der Kern noch vollständig im Bild liegt. Wir können dies stets garantieren, wenn wir nicht das gesamte Bild bearbeiten, sondern den Arbeitsbereich entsprechend verkleinern: wir verschieben die Iteratoren, die die Ecken des Arbeitsbereiches angeben, weit genug ins Innere des Bildes. Dadurch existieren auch außerhalb des neuen Arbeitsbereichs gültige Pixelwerte, die bei der Faltung verwendet werden können. Um dies kenntlich zu machen, wollen wir die Funktion `convolveImageValidOutside()` nennen.

Diese Funktion benötigt ein Quell- und ein Zielbild sowie den Faltungskern. Anders als bei Bildern (wo ein Iteratorpaar genügt, um das Iterationsgebiet zu bestimmen) benötigen wir für den Faltungskern drei Punkte: die obere linke und die untere rechte Ecke sowie das Zentrum des Kerns. Zwar sind in der Praxis viele Kerne symmetrisch, so daß auch zwei Punkte genügen würden, aber wir wollen dies nicht in der Algorithmenschnittstelle erzwingen. Unter diesen Voraussetzungen erhalten wir die folgende Implementation:

```
template <class SrcImageIterator, class SrcAccessor,
          class DestImageIterator, class DestAccessor, class KernelIterator>
void convolveImageValidOutside(SrcImageIterator src_ul,
                               SrcImageIterator src_lr, SrcAccessor srca,
                               DestImageIterator dest_ul, DestAccessor desta,
                               KernelIterator kcenter, Diff2D k_ul, Diff2D k_lr)
{
    // zwei äußere Schleifen über alle Punkte
    for(; src_ul.y < src_lr.y; ++src_ul.y, ++dest_ul.y)
    {
        SrcImageIterator src_x = src_ul;
        DestImageIterator dest_x = dest_ul;

        for(; src_x.x < src_lr.x; ++src_x.x, ++dest_x.x)
        {
            // temporäre Variable für die Summe (die Klasse NumericTraits wird
            // in Abschnitt 4.10.2 erläutert)
            typedef typename
               NumericTraits< typename SrcAccessor::value_type >::ScalarPromote
               SumType;
            SumType sum = NumericTraits<SumType>::zero();

            // zwei innere Schleifen über das durch den Kern bestimmte Fenster
            for(int j = k_ul.height; j <= k_lr.height; ++j)
            {
                for(int i = k_ul.width; i <= k_lr.width; ++i)
                {
                    sum += kcenter(-i, -j) * srca.get(src_x, Diff2D(i, j));
                }
            }
            // Schreiben des Ergebnisses
            desta.set(sum, dest_x);
        }
    }
}
```

Wenn wir diese Funktion nutzen wollen, um die Faltung mit einem 3x3-Binomialfilter (vgl. z.B. [JÄHNE91]) zu berechnen, müssen wir die Bilditeratoren um den Vektor (1,1) nach innen verschieben:[15]

```
float raw_kernel[] = {1.0/16.0, 1.0/8.0, 1.0/16.0,
                      1.0/8.0,  1.0,4.0, 1.0,8.0,
                      1.0/16.0, 1.0/8.0, 1.0/16.0};

FloatImage binom3x3(3,3);                          // Erzeugen und
copy(raw_kernel, raw_kernel+9, binom3x3.begin());  // Initialisieren des Kerns
```

[15] Im Abschnitt 5.5 werden wir eine bessere Möglichkeit der Randbehandlung implementieren, die das Verkleinern des Arbeitsbereichs vermeidet.

```
Diff2D kernel_upperleft(-1, -1), kernel_lowerRight(1, 1);
FloatImage::ImageIterator kernel_center = binom3x3.upperLeft() + Diff2D(1, 1);

FloatImage src(w,h), dest(w,h);
StandardAccessor<float> accessor;

convolveImageValidOutside(src.upperLeft() - kernel_upperleft,
                          src.lowerRight() - kernel_lowerright, accessor,
                          dest.upperLeft() - kernel_upperleft, accessor,
                          kernel_center, kernel_upperleft, kernel_lowerright);
```

Natürlich wird man in der Praxis für häufig benötigte Kerne factory-Funktionen [GHJV94] schreiben, die die Initialisierung der Kerne kapseln. Darauf wollen wir aber hier nicht eingehen. Mit Hilfe der Zugriffsobjekte können wir diesen Faltungsalgorithmus auch auf RGB-Bilder anwenden, indem wir ihn dreimal aufrufen, jeweils mit einem Zugriffsobjekt für den roten, grünen bzw. blauen Kanal.

4.9 Parameterobjekte

Wir hatten in den letzten Abschnitten gesehen, daß die Verwendung von Zugriffsobjekten zu mitunter recht komplizierten Schnittstellen bei den Algorithmen führt. Algorithmen mit acht und mehr Parametern sind schwer zu handhaben, und die Reihenfolge der Parameter kann leicht verwechselt werden. Wir müssen deshalb Maßnahmen ergreifen, die Schnittstellen wieder zu vereinfachen, ohne daß dadurch die Flexibilität beeinträchtigt wird.

Wir können dies durch die Einführung von Parameterobjekten erreichen. Anstatt alle Parameter einzeln an einen Algorithmus zu übergeben, gruppieren wir zusammengehörende Parameter und übergeben sie gemeinsam in einem Parameterobjekt. Wir drücken damit logische Beziehungen zwischen den Parametern explizit aus. Aufgrund ihrer logischen Zusammengehörigkeit werden wir folglich die Iteratoren und Zugriffsobjekte, die sich auf dasselbe Bild beziehen, zusammenfassen. Wir benötigen also ein `pair`, um ein Iterator/Accessor-Paar zu repräsentieren, und ein `triple`, um zwei Iteratoren für die Festlegung der Iterationsgrenzen sowie den zugehörigen Accessor zu übergeben. Die Klassen `pair` und `triple` haben folgende Definition:

```
template <class First, class Second>
struct pair
{
    First  first;
    Second second;
```

```
    pair(First const & f, Second const & s)
    : first(f), second(s)
    {}

    ... // weitere Funktionen (siehe [AUSTERN98])
};


template <class First, class Second, class Third>
struct triple {
    First  first;
    Second second;
    Third  third;

    triple(First const & f, Second const & s, Third const & t)
    : first(f), second(s), third(t)
    {}

    ... // weitere Funktionen
};
```

Diese Klassen können wir als Parameterobjekte verwenden, um damit eine neue Schnittstelle für die Algorithmen zu definieren. Wir zeigen dies hier beispielhaft anhand von `copyImage()`, die anderen Algorithmen werden analog behandelt:

```
template <class SrcIterator, class SrcAccessor,
          class DestIterator, class DestAccessor>
void copyImage(triple<SrcIterator, SrcIterator, SrcAccessor> src,
               pair<DestIterator, DestAccessor> dest)
{
    // entpacke die Parameterobjekte und rufe das ursprüngliche copyImage() auf:
    copyImage(src.first /* src_upperleft */, src.second /* src_lowerright */,
              src.third /* src_accessor */,
              dest.first /* dest_upperleft */, src.second /* dest_accessor */);
}
```

Dieses Interface enthält die gleichen Informationen wie vorher, aber Parameter für Quell- und Zielbild sind in einem `triple` bzw. `pair` zusammengefaßt. Dies allein führt allerdings noch nicht zu einer vereinfachten Handhabung. Wir haben aber jetzt die Möglichkeit, *Factory-Funktionen* [GHJV94] zu schreiben, die die Generierung von Parameterobjekten kapseln. Innerhalb der Factory-Funktionen können wir Standardwerte für alle jene Parameter einsetzen, für die wir nicht explizit eine abweichende Belegung wünschen. Die folgende Factory-Funktion beispielsweise generiert ein `triple` direkt aus einem Objekt von Typ `BasicImage`. Auf diese Weise drücken wir aus, daß die Iteration sich über das gesamte Bild erstrecken soll und dabei das Standard-Zugriffsobjekt verwendet wird:

```
template <class PixelType>
triple<BasicImageIterator<PixelType>, BasicImageIterator<PixelType>,
       StandardAccessor< PixelType> >
srcImageRange(BasicImage<PixelType> & image)
{
    return triple<BasicImageIterator<PixelType>, BasicImageIterator<PixelType>,
                  StandardAccessor< PixelType> >(
        image.upperLeft(), image.lowerRight(), StandardAccessor< PixelType>() );
}
```

Die folgende Variante verwenden wir, wenn wir die Iteration zwar auf das gesamte Bild erstrecken, jedoch ein spezielles Zugriffsobjekt verwenden wollen:

```
template <class PixelType, class Accessor>
triple<BasicImageIterator<PixelType>, BasicImageIterator<PixelType>, Accessor>
srcImageRange(BasicImage<PixelType> & image, Accessor const & accessor)
{
    return
    triple<BasicImageIterator<PixelType>, BasicImageIterator<PixelType>,
           Accessor> (image.upperLeft(), image.lowerRight(), accessor);
}
```

Die nächste Variante hingegen erlaubt die explizite Übergabe eines speziellen Iterationsgebiets, wobei das Standard-Zugriffsobjekt verwendet wird:

```
template <class Iterator>
triple<Iterator, Iterator, typename IteratorTraits<Iterator>::DefaultAccessor>
srcIterRange(Iterator upperleft, Iterator lowerright)
{
    return triple<Iterator, Iterator,
                  typename IteratorTraits<Iterator>::DefaultAccessor>(
                      upperleft, lowerright,
                      typename IteratorTraits<Iterator>::DefaultAccessor());
}
```

Analoge Factory-Funktionen definieren wir zur Erzeugung eines `pair` aus `upperleft` und `accessor` sowie für die entsprechenden Parameterobjekte des Zielbildes. Mit Hilfe der factory-Funktionen können wir die Beispiele aus dem vorigen Abschnitt wesentlich übersichtlicher schreiben:

```
ByteRGBImage rgb1(100, 200), rgb2(100, 200);
ByteImage    gray(100, 200);
RGBPic       candelaRGB;
for(int i=0; i<3; ++i)  candelaRGB[i] = can_palloc(100, 200, canI8);
...

// Erzeugen einiger Zugriffsobjekte
RedChannelAccessor<byte>                    redaccessor;
BlueChannelAccessor<byte>                   blueaccessor;
TransformBetweenRGBAndGray<byte>            rgb2gray, gray2rgb;
```

```
// Transformiere RGB-Bild in Graubild
copyImage(srcImageRange(rgb1, rgb2gray), destImage(gray));

// Transformiere Graubild in RGB-Bild
copyImage(srcImageRange(gray), destImage(rgb2, gray2rgb));

// Transformiere 10X10 Pixel aus der linken oberen Ecke des Graubilds in RGB-Bild
copyImage(srcIterRange(gray.upperLeft(), gray.upperLeft() + Diff2D(10,19)),
          destImage(rgb2, gray2rgb));

// Addiere zwei RGB-Bilder und speichere Resultat in einem Candela-RGB-Bild
combineTwoImages(srcImageRange(rgb1), srcImage(rgb2),
                 destImage(candelaRGB), plus<RGBVAlue<byte> >());

// Addiere nur die roten Kanäle der RGB-Bilder
combineTwoImages(srcImageRange(rgb1, redaccessor), srcImage(rgb2, redaccessor),
                 destImage(rgb2, redaccessor), plus<RGBVAlue<byte> >());

// Kopiere den roten Kanal von rgb1 in den blauen Kanal von rgb2
copyImage(srcImageRange(rgb1, redaccessor), destImage(rgb2, blueaccessor));

// setze alle Pixel zu schwarz, deren roter Kanal den Wert 0 hat
// (das Bild ist hier glechzeitig Maskenbild)
RGBValue<byte> black(0,0,0);
initImageIf(srcImageRange(rgb1), srcImage(rgb1, redaccessor), black);
```

Damit haben wir eine Form für die Schnittstellen gefunden, die einfache Benutzbarkeit mit hoher Flexibilität vereint. Die Komplexität, die mit der Verwendung von Iteratoren und Zugriffsobjekten verbunden ist, wird hinter den Factory-Funktionen versteckt, solange die volle Flexibilität nicht benötigt wird. Wenn jedoch der Standardfall nicht ausreicht, ist die volle Flexibilität sofort verfügbar.

4.10 Generische Konzepte und Metainformationen für arithmetische Operationen

Wir haben bisher Funktionen für arithmetische Operationen sowie die Faltung benutzt, ohne explizit zu fragen, auf welche Pixeltypen diese Operationen überhaupt angewendet werden können. Solange wir uns auf die eingebauten Typen der Sprache C++ beschränken (`int`, `float` usw.), ist dies nicht notwendig, weil diese Typen die notwendigen Operationen standardmäßig unterstützen. Wenn wir jedoch nutzerdefinierte Pixeltypen zulassen, wie z.B. `RGBValue`, müssen wir die Anforderungen der Algorithmen hinsichtlich arithmetischer Operationen genauer definieren.

Wir wollen einige illustrative Beispiele geben. Die Definition der Faltung hatten wir bereits angegeben:

$$(g * f)(x,y) = \sum_{i,j} g(-i,-j) f(x+i, y+j) \tag{4.4}$$

wobei g ein Faltungskern, f die Bildfunktion und * der Faltungsoperator ist. Um diese Operation ausführen zu können, müssen wir die Pixelwerte mit den Werten des Faltungskerns multiplizieren können und die Ergebnisse dieser einzelnen Operationen addieren. Auf der Grundlage der Faltung kann man den sogenannten Strukturtensor definieren [JÄHNE91]:

$$S = \begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix} = \begin{pmatrix} g*(f_x f_x) & g*(f_x f_y) \\ g*(f_x f_y) & g*(f_y f_y) \end{pmatrix} \tag{4.5}$$
$$\text{mit} \quad f_x = a_x * f \quad \text{und} \quad f_y = a_y * f$$

Zur Berechnung des Strukturtensors wird die Bildfunktion f zunächst mit je einem Ableitungskern in x- und y-Richtung (a_x bzw. a_y) gefaltet, danach werden die so berechneten Ableitungen des Bildes punktweise multipliziert ($f_x f_x$ usw.), und die Ergebnisse dieser Operation werden nochmals mit einem Glättungsfilter g gefaltet. Hier kommt somit die Anforderung hinzu, Produkte von Pixelwerten bilden zu können.

Der Strukturtensor ist Grundlage von zwei wichtigen Detektoren für Grauwertecken (vgl. z.B. [ROHR94]). Der Plessey-Detektor [HARSTEV88] identifiziert Grauwertecken mit den Maxima der sogenannten Corner Response Function. Diese Funktion ist definiert als gewichtete Differenz aus Determinante und Quadrat der Spur des Strukturtensors:

$$\text{CornerResponse} = \det S - k\,(\text{tr}\,S)^2 = (S_{11}S_{22} - S_{12}S_{21}) - k(S_{11}+S_{22})^2 \tag{4.6}$$

Hier benötigen wir als zusätzliche Operationen die Subtraktion von Pixelwerten sowie die Multiplikation mit einer reellen Zahl (der Gewichtungskonstanten k). Im Unterschied dazu detektiert der Förstner-Operator [FÖRSTNER91] Grauwertecken als Maxima des Quotienten aus Determinante und Spur des Strukturtensors:

$$\text{FörstnerOperator} = \frac{\det S}{\text{tr}\,S} \tag{4.7}$$

Dieser Operator ist nur anwendbar, wenn der Datentyp der Pixelwerte auch die Division unterstützt. Ähnliche Überlegungen muß man bei anderen Algorithmen anstellen. Es liegt deshalb nahe, typische Anforderungen zu systematisieren und zu standardisieren. Wir wollen in diesem Kapitel generische Konzepte hierfür entwikkeln.

4.10.1 Konzepte für arithmetische Datentypen

Die Mathematik beschäftigt sich im Rahmen der Theorie algebraischer Gruppen seit langem mit der Klassifikation von Zahlentypen. Es ist zweckmäßig, auf dieser Klassifikation aufzubauen, auch wenn hier nicht deren volle Breite benötigt wird. Wir wollen zunächst einen Überblick über wichtige algebraische Strukturen der Mathematik geben (vergl. [BARTNACK94]):

Körper: Ein algebraischer Körper ist eine Menge von Objekten, auf denen die arithmetischen Grundoperationen Addition und Multiplikation sowie deren Umkehrungen Subtraktion und Division mit den üblichen Rechenregeln definiert sind. Insbesondere müssen die Addition zumindest kommutativ und die Multiplikation zumindest assoziativ sein. Außerdem gibt es ein neutrales Element der Addition als eindeutige Lösung der Gleichung $a + x = a$ (Nullelement) und ein neutrales Element der Multiplikation als eindeutige Lösung der Gleichung $a\,x = a$ (Einselement). Beispielsweise bilden die rationalen Zahlen einen Körper, ebenso die reellen Zahlen.

Ring: Ein Ring ist eine Menge von Objekten, in der man in der üblichen Weise addieren, subtrahieren und multiplizieren, aber nicht notwendigerweise dividieren darf. Da hierbei eine der Forderungen an einen Körper fehlt, ist jeder Körper ein Ring. Es gibt aber auch Ringe, die keine Körper sind, wie z.B. die Menge der ganzen Zahlen.

Linearer Raum: Ein linearer Raum (auch Vektorraum) ist eine Menge von Objekten, die man wie üblich addieren kann (es gilt das Kommutativgesetz und es exisitert ein Nullelement), und die man mit Objekten aus einem Körper (z.B. einer reellen Zahl) multiplizieren kann. Dabei müssen Addition und Multiplikation die üblichen Linearitätsbedingungen erfüllen. Man bezeichnet die Multiplikation als *äußere Multiplikation*, da sie Objekte unterschiedlichen Typs verknüpft. Die Vektoren einer gegebenen Dimension bilden beispielsweise einen Vektorraum über dem Körper der reellen Zahlen.

Lineare Algebra: Eine lineare Algebra (oder kurz Algebra) ist ein linearer Raum, bei dem zusätzlich eine *innere Multiplikation* definiert ist (nicht zu verwechseln mit dem Skalarprodukt: das Ergebnis der inneren Multiplikation ist wieder ein Objekt der Grundmenge). Die innere Multiplikation verknüpft zwei Objekte der Grundmenge mit den üblichen Rechenregeln (insbesondere gilt das Assoziativgesetz und es existiert ein Einselement). Eine lineare Algebra ist stets ein Ring. Die quadratischen Matrizen einer gegebenen Größe bilden eine lineare Algebra über dem Körper der reellen Zahlen.

Divisionsalgebra: Eine Divisionsalgebra erweitert die lineare Algebra um die Division als Umkehrung der inneren Multiplikation. Jede Divisionsalgebra ist somit selbst ein Körper. Wir können andererseits einen Körper als Spezialfall

einer Divisionsalgebra auffassen, bei dem äußere und innere Multiplikation identisch sind.

Darüber hinaus muß in allen Fällen eine Äquivalenzrelation definiert sein, damit wir feststellen können, ob zwei Objekte der Grundmenge gleich sind. Die weitaus meisten für Computer Vision relevanten Pixeltypen kann man in eine der erwähnten Klassen einordnen.

Die integralen Datentypen der Sprache C++ (`byte`, `short`, `int`) sind Modelle des Konzepts „Ring“. Genauer gesagt sind sie Restklassenringe zum Modul 2^8 (`byte`), 2^{16} (`short`) bzw. 2^{32} (`int`) (andere Module sind in C++ erlaubt, werden aber selten verwendet). Die Klassifikation als Restklassenring ergibt sich aus der Tatsache, daß die Arithmetik der integralen Typen zyklisch definiert ist: tritt während einer arithmetischen Berechnung ein Überlauf auf, wird das Ergebnis modulo der angegebenen Zweierpotenzen berechnet. Tritt hingegen kein Überlauf auf, erhält man das gleiche Ergebnis wie im Ring der ganzen Zahlen. Ein integraler Typ mit genügend großem Modul (so daß kein Überlauf auftritt) verhält sich also wie eine ganze Zahl. Aus der Klassifikation als Ring ergibt sich, daß wir die integralen Typen addieren, subtrahieren und multiplizieren dürfen. Zwar definiert C++ auch eine Division für integrale Typen, aber diese berechnet den ganzzahligen Anteil des Quotienten und genügt damit nicht den mathematischen Anforderungen an einen Körper: sie ist nicht die Umkehrung der Multiplikation, denn für die integralen Typen gilt im allgemeinen `(a / b) * b` $\neq$ `a`. Wenn ein Algorithmus die Division benötigt, können wir die integralen Typen nur eingeschränkt verwenden.

Die Fließkommatypen (`float` und `double`) approximieren die Körper der rationalen und der reellen Zahlen. Durch die begrenzte Genauigkeit der Zahlendarstellung gelten die Körperaxiome nicht exakt, aber für die meisten Anwendungen ist die erreichte Genauigkeit ausreichend. Bei höheren Genauigkeitsanforderungen kann man einen nutzerdefinierten Typ für Zahlen beliebiger Genauigkeit einführen (vergl. z.B. [BURNKÖN96]). Da die eingebauten Typen der Sprache C++ die mathematischen Konzepte nur approximieren, erscheint es zweckmäßig, die Konzepte für generische Software weniger streng zu fassen als die entsprechenden mathematischen Axiome.

Die Sprache C++ definiert außerdem ein Konzept, das in den algebraischen Strukturen nicht vorgesehen ist, nämlich die *Typkonvertierung*. Dadurch können wir alle arithmetischen Operationen auch mit Argumenten gemischter Typen ausführen. Insbesondere können wir jeden eingebauten Typ mit einem `double` multiplizieren. Damit können wir die integralen Typen näherungsweise als Algebren über dem Körper der reellen Zahlen interpretieren.

Aufgrund des Gesagten wollen wir die generischen Konzepte für arithmetische Operationen wie folgt definieren:

AlgebraicRing: Ein Datentyp genügt den Anforderungen an das Konzept AlgebraicRing, wenn er folgende Operationen unterstützt:

Vergleich von Datenelementen auf Identität: Erfüllung der Anforderungen an EqualityComparable gemäß [AUSTERN98].

Addition und Subtraktion von Datenelementen: Anwendung der üblichen Operatoren für diese Operationen. Die Subtraktion muß die Addition umkehren (zumindest näherungsweise, wobei die erforderliche Güte der Näherung von der Anwendung abhängt). Es existiert ein eindeutiges Nullelement.

```
AlgebraicRing a, b, c, zero = ...;
c += a;
c -= a;
c = a + b;
c = a - b;
c = -a;
assert(a + zero == a);
assert(a + b == b + a);
```

Multiplikation von Datenelementen: Anwendung der üblichen Operatoren für diese Operation. Es existiert ein eindeutiges Einselement.

```
AlgebraicRing a, b, c, one = ...;
c *= a;
c = a * b;
assert(a * one == a);
assert((a * b) * c == a * (b * c));
```

AlgebraicField: Ein Datentyp genügt den Anforderungen an das Konzept AlgebraicField (algebraischer Körper), wenn er die Anforderungen an AlgebraicRing erfüllt und zusätzlich folgende Operationen unterstützt:

Division von Datenelementen: Anwendung der üblichen Operatoren für diese Operation. Die Division kehrt die Multiplikation um (zumindest näherungsweise, wobei die erforderliche Güte der Näherung von der Anwendung abhängt). Die Division durch das Nullelement der Addition ist nicht definiert:

```
AlgebraicField a, b, c, zero = ...;
if(a != zero) c /= a;
if(a != zero) c = a + b;
```

LinearSpace: Ein Datentyp genügt den Anforderungen an das Konzept LinearSpace, wenn er folgende Operationen unterstützt:

Vergleich von Datenelementen auf Identität: Erfüllung der Anforderungen an EqualityComparable gemäß [AUSTERN98].

Addition und Subtraktion von Datenelementen: Anwendung der üblichen Operatoren für diese Operationen. Die Subtraktion muß die Addition umkehren (zumindest näherungsweise, wobei die erforderliche Güte der Näherung von der Anwendung abhängt). Es existiert ein eindeutiges Nullelement.

```
LinearSpace a, b, c, zero = ...;
c += a;
c -= a;
c = a + b;
c = a - b;
c = -a;
assert(a + zero == a);
assert(a + b == b + a);
assert(a - b == a + (-b));
```

Multiplikation und Division mit Elementen eines Körpers (Modellen von AlgebraicField):

```
LinearSpace a, b;
AlgebraicField f, f_zero = ...;
b *= f;
if(f != f_zero) b /= f;
b = f * a;
b = a * f;
if(f != f_zero) b = a / f;
```

Addition und Multiplikation müssen, zumindest näherungsweise, die Linearitätsbedingungen erfüllen:

```
LinearSpace a, b;
AlgebraicField e, f;
assert( e*(a + b) == e*a + e*b );
assert( (e + f)*a == e*a + f*a );
```

LinearAlgebra: Ein Datentyp genügt den Anforderungen an das Konzept LinearAlgebra, wenn er gleichzeitig die Anforderung an die Konzepte AlgebraicRing und LinearSpace erfüllt.

DivisionAlgebra: Ein Datentyp genügt den Anforderungen an das Konzept DivisionAlgebra, wenn er gleichzeitig die Anforderung an die Konzepte AlgebraicField und LinearSpace erfüllt.

Diese Konzepte decken die wichtigsten Anforderungen ab, die bei Computer Vision-Algorithmen auftreten. Wir erkennen beispielsweise, daß die Anforderungen des Faltungsalgorithmus auf zwei Arten erfüllt werden können: haben die Werte des Faltungskerns den gleichen Typ wie die Pixel, so muß dieser Typ ein AlgebraicRing sein. Erfüllt der Typ der Pixelwerte hingegen nur die Anforderungen an LinearSpace, so müssen die Werte des Kerns aus dem Körper (AlgebraicField) stammen,

über dem der lineare Raum definiert ist. Praktisch folgt daraus, daß wir mit einem Kern mit dem Elementtyp `double` die Faltung ausführen können für (1) sämtliche eingebauten Typen von C++ sowie (2) für alle linearen Räume, deren äußere Multiplikation mit dem Typ `double` definiert ist. Ähnlich finden wir für den Plessey-Eckendetektor, daß er auf Bilder angewendet werden kann, deren Pixeltyp eine LinearAlgebra ist, während der Förstner-Operator eine DivisionAlgebra benötigt.

4.10.2 Metainformationen über arithmetische Datentypen

Im vorigen Abschnitt hatten wir bereits erwähnt, daß C++ die Möglichkeit der *automatischen Typkonvertierung* bietet. Dies gestattet dem Programmierer, für jedes Datenelement den jeweils am besten passenden Typ zu verwenden. Werden mehrere verschiedene Typen in einem arithmetischen Ausdruck gemischt, fügt der Compiler automatisch die notwendigen Typkonvertierungen ein. Im Kontext von Computer Vision ist dies sehr verbreitet: beispielsweise werden für die Bildschirmdarstellung und Archivierung üblicherweise `byte`- oder `short`-Bilder benutzt, während für viele Bildverarbeitungsverfahren Pixel vom Typ `float` oder `double` zweckmäßiger sind, um eine hohe Genauigkeit zu sichern.

Folglich treten in Computer Vision-Algorithmen häufig verschiedene Datentypen gemischt auf. Generische Algorithmen müssen in der Lage sein, in einer solchen Situation die angemessenen Typen der Zwischen- und Endergebnisse automatisch zu bestimmen. Als einfaches Beispiel betrachte man die Addition von zwei Datenelementen, die wir für beliebige Datentypen verallgemeinern möchten. Dies ist im Prinzip sehr einfach, wenn wir, wie in C++ üblich, die binäre Addition (`operator+`) auf die Addition mit Zuweisung (`operator+=`) zurückführen:

```
template <class ArithmeticType>
ArithmeticType
operator+(ArithmeticType const & l, ArithmeticType const & r)
{
    ArithmeticType result(l);
    result += r;
    return result;
}
```

Diese generische Addition ist noch nicht in Ausdrücken mit gemischten Typen anwendbar. Wollen wir sie entsprechend verallgemeinern, müssen wir einen Weg finden, den Typ von `result` korrekt festzulegen. Dies geschieht zweckmäßig durch eine traits-Klasse `PromoteTraits`, die die Typumwandlungsregeln kodiert [MYERS95, VELD97]:

```
template <class ArithmeticType1, class ArithmeticType2>
typename PromoteTraits< ArithmeticType1, ArithmeticType2 >::Promote
operator+( ArithmeticType1 const & l, ArithmeticType2 const & r)
```

```
{
    typedef typename PromoteTraits< ArithmeticType1, ArithmeticType2 >::Promote
            ResultType;
    ResultType result(l);
    result += static_cast<ResultType>(r);
    return result;
}
```

Die Metaklasse `PromoteTraits<...>` muß für alle Paare von Typen definiert werden, die in einem Ausdruck gemeinsam auftreten können. Der eingebettete Typ `Promote` gibt den gemeinsamen Obertyp an, in den die beiden ursprünglichen Typen konvertiert werden sollen (die entsprechenden Konvertierungsoperatoren bzw. Konstruktoren müssen natürlich definiert sein). Beispielsweise wäre in einem gemischten Ausdruck mit `int`- und `float`-Argumenten die Konvertierung nach `float` angebracht, während die Addition eines `ByteImage` und eines `FloatImage` zweckmäßigerweise ein `FloatImage` ergeben sollte. Die entsprechenden traits-Klassen sehen demnach folgendermaßen aus:

```
template <class T1, class T2> struct PromoteTraits;    // forward declaration

template <> struct PromoteTraits< int, float > {   // explizite Spezialisierung
    typedef float Promote;                          // für int <-> float
};

template <> struct PromoteTraits< ByteImage, FloatImage > {
    typedef FloatImage Promote;  // Spezialisierung für ByteImage <-> FloatImage
};
```

Bei der weiteren Analyse stellt sich heraus, daß die Frage der Typumwandlung nicht nur bei binären Operationen eine Rolle spielt, sondern auch dann, wenn nur ein Datentyp vorhanden ist. Man betrachte zum Beispiel die Aufgabe, den Mittelwert eines Array von byte zu bestimmen. Der folgende Algorithmus führt in den meisten Fällen zu fehlerhaften Ergebnissen:

```
byte wrongAverage(byte * begin, byte * end)
{
    int count = end - begin;
    byte sum = 0;                                   // Fehler: Summe als byte

    for(; begin != end; ++begin)    sum += *begin;  // Gefahr des Überlaufs !
    return sum / count;
}
```

Es ist natürlich ein Fehler, die Zwischensumme sum als byte zu speichern, weil wahrscheinlich nach wenigen Iterationen ein Überlauf eintritt. Obwohl der Fehler hier offensichtlich ist, wäre er möglicherweise nicht so schnell aufgefallen, wenn der Algorithmus generisch, also als template, implementiert worden wäre. Um auch in

diesem Falle richtige Ergebnisse zu garantieren, müssen wir einen geeigneten Typ definieren, der solche Zwischenergebnisse aufnehmen kann. Wir nennen die hierfür zuständige Metaklasse NumericTraits. Die korrekte generische Variante der Mittelwertbildung würde damit folgendermaßen implementiert (wir verwenden außerdem die in Abschnitt 4.6 eingeführten Zugriffsobjekte):

```
template <class Iterator, class Accessor>
typename NumericTraits< typename Accessor::value_type >::Promote
average(Iterator begin, Iterator end, Accessor a)
{
    int count = 0;
    typedef typename NumericTraits< typename Accessor::value_type >::Promote
            SumType;
    SumType sum = SumType::zero();

    for(; begin != end; ++begin, ++count)    sum += a.get(*begin);
    return sum / (double)count;
}
```

Wir sehen, wie mit Hilfe der traits-Klasse der korrekte Summentyp indirekt ermittelt wird. Daneben illustriert das Codebeispiel eine weitere Aufgabe der Klasse NumericTraits: sie enthält außerdem eine factory-Funktion zero(), die das Nullelement des betreffenden Datentyps erzeugt, das für die Initialisierung von sum benötigt wird. Daneben muß eine Funktion nonzero() definiert werden, die ein beliebiges Element ungleich zero() erzeugt. Auch das Einselement muß, falls es existiert, in gleicher Weise durch eine factory-Funktion one() innerhalb von NumericTraits zugänglich sein. Falls one() definiert ist, muß nonzero() das Einselement zurückliefern. Schließlich enthält die Metaklasse noch zwei weitere Typen, ScalarMultiplier und ScalarPromote, die angeben, mit welchem Typ die skalare (äußere) Multiplikation definiert ist und welchen Typ das Ergebnis dieser Multiplikation hat. Die vollständige Metaklasse NumericTraits für den Typ byte sieht folglich so aus:

```
template <class T> struct NumericTraits;  // forward declaration

template <> struct NumericTraits< byte > { // explizite Spezialisierung für byte
    typedef int    Promote;
    typedef double ScalarMultiplier;
    typedef double ScalarPromote;

    static byte zero() { return 0; }
    static byte one()  { return 1; }
    static byte nonzero()  { return 1; }
};
```

Entsprechende Metaklassen müssen für sämtliche anderen Datentypen, die eines der arithmetischen Konzepte erfüllen, definiert werden.

4.10.3 Anwendung: arithmetische Operationen für RGB-Werte

Wir hatten bereits in Abschnitt 4.5 einen einfachen Typ für RGB-Werte angegeben. Mit Hilfe der inzwischen eingeführten Konzepte für arithmetische Operationen und den zugehörigen Metaklassen können wir diesen Typ wesentlich verbessern. Wir wollen dabei erreichen, daß der RGB-Typ möglichst viele der geforderten Anforderungen erfüllt, so daß man ihn bei möglichst vielen Algorithmen unmittelbar verwenden kann.

Da RGB-Tupel Vektoren der Länge 3 sind, liegt es nahe, ihnen die Eigenschaften eines LinearSpace zu geben. Dabei werden alle Operationen komponentenweise ausgeführt, daß heißt, wir addieren Rot zu Rot, Grün zu Grün, Blau zu Blau. Das Nullelement ist dann der Nullvektor (0,0,0). Die äußere Multiplikation wird bezüglich des Typs `double` definiert.

Wir können aber noch weiter gehen und eine komponentenweise Multiplikation einführen. Damit erfüllt der RGB-Wert auch die Anforderungen an LinearAlgebra, wobei das Einselement der Vektor (1,1,1) ist. Die Bedingungen für eine Divisionsalgebra sind allerdings nicht erfüllt, denn eine Division durch Null kann auch dann auftreten, wenn nur eine der Komponenten den Wert Null hat: (1,1,1) / (0,1,1) ist undefiniert, obwohl der Divisor nicht das Nullelement ist.

Da es in Computer Vision-Anwendungen üblich ist, RGB-Werte mit unterschiedlichen Komponententypen zu verwenden, werden diese zwangsläufig in Ausdrücken gemischt auftreten. Wir müssen deshalb auch Operationen für die Typumwandlung sowie die zugehörigen Metaklassen definieren.

Die Sprache C++ erlaubt uns, diese Operationen auf Basis der in Abschnitt 4.5.2 bereits beschriebenen Klasse `RGBValue` zu definieren, ohne daß wir diese Klasse modifizieren müssen. Alle Operationen werden so definiert, daß sie gemischte Argumenttypen akzeptieren:

```
// Anforderungen an LinearAlgebra: Addition
template <class T1, class T2>
RGBValue<T1> & operator+=(RGBValue<T1> & v1, RGBValue<T2> const & v2)
{
    v1.red() += v2.red(); v1.green() += v2.green(); v1.blue() += v2.blue();
    return v1;
}

template <class T1, class T2>
typename PromoteTraits<RGBValue<T1>, RGBValue<T2> >::Promote
operator+(RGBValue<T1> const & v1, RGBValue<T2> const & v2)
{
    typedef typename PromoteTraits<RGBValue<T1>, RGBValue<T2> >::Promote Result;

    return (Result(v1) += v2);
}
```

```
// Anforderungen an LinearAlgebra: Subtraktion
template <class T1, class T2>
RGBValue<T1> & operator-=(RGBValue<T1> & v1, RGBValue<T2> const & v2) {
    v1.red() -= v2.red(); v1.green() -= v2.green(); v1.blue() -= v2.blue();
    return v1;
}

template <class T1, class T2>
typename PromoteTraits<RGBValue<T1>, RGBValue<T2> >::Promote
operator-(RGBValue<T1> const & v1, RGBValue<T2> const & v2) {
    typedef typename PromoteTraits<RGBValue<T1>, RGBValue<T2> >::Promote Result;

    return (Result(v1) -= v2);
}

template <class T>
RGBValue<T> operator-(RGBValue<T> const & v) {
    return RGBValue<T>(-v.red(), -v.green(), -v.blue());
}


// Anforderungen an LinearAlgebra: Multiplikation
template <class T1, class T2>
RGBValue<T1> & operator*=(RGBValue<T1> & v1, RGBValue<T2> const & v2) {
    v1.red() *= v2.red(); v1.green() *= v2.green(); v1.blue() *= v2.blue();
    return v1;
}

template <class T1, class T2>
typename PromoteTraits<RGBValue<T1>, RGBValue<T2> >::Promote
operator*(RGBValue<T1> const & v1, RGBValue<T2> const & v2) {
    typedef typename PromoteTraits<RGBValue<T1>, RGBValue<T2> >::Promote Result;

    return (Result(v1) *= v2);
}


// Anforderungen an LinearAlgebra: äußere Multiplikation
template <class T >
RGBValue<T> & operator*=(RGBValue<T> & v,
                         typename NumericTraits<RGBValue<T> >::ScalarMultiplier d) {
    v.red() *= d; v.green() *= d; v.blue() *= d;
    return v;
}

template <class T >
RGBValue<T> & operator/=(RGBValue<T> & v,
                         typename NumericTraits<RGBValue<T> >::ScalarMultiplier d) {
    v.red() /= d; v.green() /= d; v.blue() /= d;
    return v;
}
```

```
template <class T>
typename NumericTraits<RGBValue<T> >::ScalarPromote
operator*(RGBValue<T> const & v,
          typename NumericTraits<RGBValue<T> >::ScalarMultiplier d) {
    return (typename NumericTraits<RGBValue<T> >::ScalarPromote(v) *= d);
}

template <class T>
typename NumericTraits<RGBValue<T> >::ScalarPromote
operator*(typename NumericTraits<RGBValue<T> >::ScalarMultiplier d,
          RGBValue<T> const & v) {
    return (typename NumericTraits<RGBValue<T> >::ScalarPromote(v) *= d);
}

template <class T>
typename NumericTraits<RGBValue<T> >::ScalarPromote
operator/(RGBValue<T> const & v,
          typename NumericTraits<RGBValue<T> >::ScalarMultiplier d) {
    return (typename NumericTraits<RGBValue<T> >::ScalarPromote(v) /= d);
}
```

Die Metaklassen `NumericTraits` und `PromoteTraits` für RGB-Werte werden soweit wie möglich auf die entsprechenden Metaklassen der Komponententypen zurückgeführt. Dank der in C++ möglichen partiellen `template`-Spezialisierung genügt eine Implementation, um diese Metaklassen für alle möglichen RGB-Werte zu definieren:

```
template <class COMPONENTTYPE>
struct NumericTraits< RGBValue< COMPONENTTYPE > >
{
    typedef NumericTraits< COMPONENTTYPE > BaseTraits;

    typedef RGBValue< typename BaseTraits::Promote >        Promote;
    typedef typename BaseTraits::ScalarMultiplier           ScalarMultiplier;
    typedef RGBValue< typename BaseTraits::ScalarPromote >  ScalarPromote;

    static RGBValue< COMPONENTTYPE > zero() {
        return RGBValue< COMPONENTTYPE >( BaseTraits::zero(),
                                          BaseTraits::zero(),
                                          BaseTraits::zero() );
    }

    static RGBValue< COMPONENTTYPE > one() {
        return RGBValue< COMPONENTTYPE >( BaseTraits::one(),
                                          BaseTraits::one(),
                                          BaseTraits::one() );
    }

    static RGBValue< COMPONENTTYPE > nonzero() { return one(); }
};
```

```
template <class COMPONENTTYPE1, COMPONENTTYPE2 >
struct PromoteTraits< RGBValue< COMPONENTTYPE1 >, RGBValue< COMPONENTTYPE2 > >
{
    typedef PromoteTraits<COMPONENTTYPE1, COMPONENTTYPE2>  BaseTraits
    typedef RGBValue< typename BaseTraits::Promote >       Promote;
};
```

Durch die Definition einer Arithmetik für RGB-Werte können wir nun den Faltungsalgorithmus direkt auf RGB-Werte anwenden und müssen ihn nicht mehr für jeden Farbkanal getrennt aufrufen. Die Verwendung von RGB-Bildern wird somit hinsichtlich dieser und anderer Operationen transparent. Solange ein Algorithmus nur die hier definierten arithmetischen Operationen benötigt, sind keine getrennten Funktionen für skalare Bilder und RGB-Bilder mehr notwendig.

4.11 Vergleich des generischen Ansatzes mit anderen Ansätzen

Wir haben nunmehr einen Grundstock an generischen Konzepten für Computer Vision-Anwendungen geschaffen, mit denen die wichtigsten wiederkehrenden Anforderungen an Flexibilität abgedeckt werden:

- variierende Bilddatenstrukturen und Pixeltypen
- Typkonvertierungen
- unterschiedliche Iterationsgebiete und -reihenfolgen
- austauschbare Teilberechnungen.

Wir wollen diese neuen Konzepte nun mit herkömmlichen Ansätzen hinsichtlich Flexibilität, Code-Umfang und Geschwindigkeit vergleichen. Zunächst wollen wir am Beispiel einiger Punktoperationen typische Implementationen aus dem Khoros-System [KHOROS91] den entsprechenden generischen Varianten gegenüberstellen. Khoros bietet sich zum Vergleich an, weil es das Problem des kartesischen Produkts in sehr reiner Form zeigt – die meisten Punktoperationen sind explizit für alle unterstützten Pixeltypen implementiert. Dadurch ist der Khoros-Code hochgradig redundant.[16] Unsere generischen Algorithmen vermeiden Redundanz und benötigen deshalb bei gleicher oder umfangreicherer Funktionalität nur etwa ein Zehntel

[16] Andere Systeme verringern die Redundanz durch eine der in Abschnitt 2.2 beschriebenen Techniken, wie z.B. bottleneck-Typen oder einfache Codegeneratoren. Jedoch führt dies, wie wir gezeigt hatten, nicht zu befriedigenden Lösungen des Problems des kartesischen Produkts.

des Quellcodes, ohne dadurch wesentlich langsamer zu arbeiten als die entsprechenden Khoros-Funktionen.

Bildkonvertierung

In Khoros ist für die Bildkonvertierung die Funktion `lvconvert()` zuständig, die ein Bild eines beliebigen Typs in ein äquivalentes Bild jedes anderen Typs umwandelt. Um ein Bild mit Pixeln vom Typ `byte` in eines vom Typ `float` umzuwandeln, wird beispielsweise der folgende Aufruf verwendet (drei Punkte repräsentieren weitere Funktionsparameter, die in unserem Zusammenhang nicht interessieren):

```
// byte-Bild erzeugen
xvimage * khoros_image = createimage(height, width, VFF_TYP_1_BYTE, ...);
...
// in float-Bild konvertieren
lvconvert(khoros_image, VFF_TYP_FLOAT, ...);
```

Der entsprechende Aufruf in unserem generischen Ansatz lautet:

```
ByteImage  bimage(height, width);   // Quellbild erzeugen
FloatImage fimage(height, width);   // Zielbild erzeugen
...
copyImage(srcImageRange(bimage), destImage(fimage));  // Konvertieren
```

Damit Khoros alle Typen von Bildern ineinander umwandeln kann, muß die Konversionsfunktion intern für jede mögliche Kombination von Typen explizit implementiert werden. Da `lvconvert()` sechs Pixeltypen unterstützt (1 bit, 1/2/4 byte, float und komplex), gibt es 36 mögliche Kombinationen. Aus diesem Grunde werden für `lvconvert()` über 1200 Codezeilen benötigt (gemessen als Anzahl der C-statements im originalen Quellcode; Messung durch den Autor). Bei der Funktion `copyImage()` hingegen benötigen wir nur ein template für sämtliche Kombinationen, da der Compiler den benötigten Code hieraus automatisch generieren kann. Die Funktion `copyImage()` besteht deshalb nur aus 10 Codezeilen, und die Menge der unterstützten Pixeltypen ist nicht von vornherein beschränkt.

Allerdings ist der Vergleich in dieser Form noch nicht ganz fair, denn die Funktion `lvconvert()` übernimmt noch eine weitere Aufgabe: sie kann während des Kopierens eine lineare Transformation auf die Pixelwerte anwenden. Dies wird vor allem dazu verwendet, um Werte aus einem beliebigen Wertebereich auf den für die Bildschirmdarstellung notwendigen Bereich 0...255 abzubilden. Das ist beispielsweise bei der Konvertierung von `float` nach `byte` wichtig:

```
// float-Bild erzeugen
xvimage * khoros_image = createimage(height, width, VFF_TYP_FLOAT, ...);
...
```

```
// in byte-Bild konvertieren und auf Bereich 0...255 transformieren
lvconvert(khoros_image, VFF_TYP_1_BYTE, true, false, 255.0, 0, false);
            //  obere Grenze des neuen Wertebereichs ^^^^^
```

Mit unseren generischen Funktionen können wir diese Aufgabe in zwei Schritten lösen. Zunächst werden mit Hilfe von `inspectImage()` der minimale und der maximale Pixelwert gesucht. Wir verwenden hierzu den Funktor `MinMaxFunctor` aus Abschnitt 4.1:

```
FloatImage fimage(height, width);   // Quellbild erzeugen
ByteImage  bimage(height, width);   // Zielbild erzeugen
...
MinMaxFunctor<float> minmax;

inspectImage(srcImageRange(fimage), minmax);
```

Die hierbei gefundenen Extremwerte können wir nun verwenden, um eine lineare Transformation nach folgender Formel durchzuführen:

$$I_{neu} = \frac{255}{I_{\max} - I_{\min}}(I_{alt} - I_{\min}) \tag{4.8}$$

Dazu müssen wir einen neuen Funktor implementieren, der eine solche lineare Transformation realisiert:

```
template <class PixelType>
struct LinearIntensityTransform
{
    typedef typename NumericTraits< PixelType >::Promote         Offset;
    typedef typename NumericTraits< Offset >::ScalarMultiplier   Scale;
    typedef typename NumericTraits< Offset >::ScalarPromote      Result;

    Scale  scale;
    Offset offset;

    LinearIntensityTransform(Scale s, Offset o)
    : scale(s), offset(o)
    {}

    Result operator()(PixelType const & v) const {
        return Result(scale * (v + offset));
    }
};
```

Mit Hilfe dieses Funktors können wir nun im zweiten Schritt die Funktion `transformImage()` aufrufen, um die Pixelwerte umzurechnen:

```
LinearIntensityTransform<float>
                toVisibleRange(255.0/(minmax.max - minmax.min), - minmax.min);

transformImage(srcImageRange(fimage), destImage(bimage), toVisibleRange);
```

Dieser Aufruf ersetzt den Aufruf der Funktion `copyImage()`. Addieren wir den Aufwand, der bei der zweiten generischen Variante entsteht (die der Funktion `lvconvert()` entspricht), so kommen wir auf insgesamt etwa 40 Codezeilen. Der Unterschied ist immer noch beträchtlich: Khoros benötigt etwa die 30-fache Anzahl an Codezeilen, und seine Funktionalität ist dennoch weniger allgemein, denn die möglichen Pixeltypen sind von vornherein festgelegt.

Extrahieren und Einfügen von Teilbildern

Für das Extrahieren und Einfügen von Teilbildern stellt Khoros die Funktionen `lvextract()`, `lvinsert()` und `lvreplace()` zur Verfügung. Typische Aufrufe sehen beispielsweise so aus:

```
xvimage * img1 = createimage(200, 200, VFF_TYP_FLOAT, ...);
xvimage * img2;
...
// extrahiere 100x100 Pixel ab der Position (50, 50) des Quellbildes
lvextract(img1, &img2, 100, 100, 50, 50, ...);

// füge die extrahierten Pixel ab der Position (100, 100) wieder ein
lvinsert(img2, img1, 100, 100, ...);

xvimage * img3 = createimage(200, 200, VFF_TYP_FLOAT, ...);
xvimage * mask = createimage(200, 200, VFF_TYP_FLOAT, ...);
...

// überschreibe die Pixel von img3 mit den entsprechenden Werten von img1
// wenn das korrespondierende Maskenpixel einen Wert != 0 hat
lvreplace(img3, img1, mask);
```

Die entsprechenden generischen Aufrufe lauten:

```
FloatImage img1(200, 200);
FloatImage img2(100, 100);
...
// extrahiere 100x100 Pixel ab der Position (50, 50) des Quellbildes
copyImage(srcIterRange(img1.upperLeft() + Diff2D(50, 50),
                       img1.upperLeft() + Diff2D(150, 150)), destImage(img2));

// füge die extrahierten Pixel ab der Position (100, 100) wieder ein
copyImage(srcImageRange(img2), destIter(img1.upperLeft() + Diff2D(100, 100)));

FloatImage img3(200, 200), mask(200, 200);
...
// überschreibe die Pixel von img3 mit den entsprechenden Werten von img1
// wenn das korrespondierende Maskenpixel einen Wert != 0 hat
copyImageIf(srcImageRange(img1), srcImage(mask), destImage(img3));
```

Die Khoros-Funktionen schlagen mit über 300 Codezeilen zu Buche, während wir für die generische Implementation nur 10 zusätzliche Codezeilen für `copyImageIf()` benötigen. Man könnte einwenden, daß die notwendige Funktionalität bereits in den Iteratoren enthalten ist, deren Größe in den Vergleich nicht einfließt. Der Vergleich fällt aber auch dann zugunsten des generischen Ansatzes aus, wenn wir die Bilddatenstrukturen und die damit assoziierten Typen vergleichen: der Khoros-Bilddatentyp `struct xvimage` sowie die zugehörigen Funktionen `createimage()` und `freeimage()` benötigen etwas über 200 Codezeilen. Die in dieser Arbeit beschriebenen Klassen `BasicImage`, `BasicImageIterator`, `StandardAccessor` sowie die Parameterobjekte (einschließlich aller member functions und factory functions) benötigen zusammen nur etwa 120 Codezeilen.

Arithmetische und algebraische Punktoperationen

Für die binären arithmetischen Punktoperationen stehen in Khoros die Funktionen `lvadd()`, `lvsub()`, `lvmul()`, `lvdiv()` und `lvabsdiff()` (Absolutbetrag der Differenz) zur Verfügung, als unäre Operationen werden unter anderem `lvabs()` (Absolutbetrag), `lvexp()` (Exponentialfunktion zur Basis 10), `lvlog()` (natürlicher und dekadischer Logarithmus), `lvnot()` (Negation), `lvoffset()` (Addieren einer Konstanten) und `lvsqrt()` (Quadratwurzel) angeboten. In unserem generischen Ansatz können wir diese Funktionen durch die Kombination eines geeigneten Funktors mit den Algorithmen `combineTwoImages()` bzw. `transformImage()` implementieren. Die binären Khoros-Funktionen haben eine durchschnittliche Größe von 150 Codezeilen, die unären kommen auf etwa 100 Codezeilen, während im generischen Ansatz ein neuer Funktor mit unter 10 Codezeilen anzusetzen ist. Zudem sind die wichtigsten Funktoren in der C++-Standardbibliothek bereits enthalten.

Insgesamt reduziert sich durch die generische Programmierung die Codegröße auf weniger als ein Zehntel bei gleichzeitig wesentlich höherer Flexibilität. Dieser Vorteil wird sich auch bei den komplexeren Algorithmen, die wir in den folgenden Kapiteln behandeln werden, zeigen. Der Unterschied rührt in erster Linie daher, daß Khoros alle Algorithmen mehrmals implementieren muß, weil sie stark von den Datenstrukturen abhängen. Bei generischer Programmierung hingegen bleibt die abstrakte Beziehung zwischen Algorithmen und Datenstrukturen erhalten.

Geschwindigkeitsvergleich

Auch bezüglich der Geschwindigkeit schneidet die generische Programmierung gut ab. Um diese Aussage zu belegen, haben wir entsprechende Tests mit verschiedenen Versionen der Faltungsoperation durchgeführt:

Variante 1: traditionelle prozedurale Implementation

Variante 2: Implementation mit `BasicImageIterator` (Abschnitt 4.5.1) , Verwendung der Navigationsfunktionen und `operator*`

Variante 3: Implementation mit `BasicImageIterator` (Abschnitt 4.5.1), Verwendung der Indexfunktion `operator()`

Variante 4: Implementation mit einer von `AbstractByteImage` (Abschnitt 4.5.3) abgeleiteten Klasse und der virtuellen Funktion `getPixel()`

	Variante des Algorithmus			
System	1	2	3	4
1	2.75 s (100%)	3.3 s (120%)	2.9 s (105%)	12.6 s (460%)
2	9.9 s (100%)	18.5 s (190%)	12.9 s (130%)	42.2 s (425%)
3	3.7 s (100%)	4.5 s (120%)	3.7 s (100%)	22.2 s (600%)
4	11.5 s (100%)	21.0 s (180%)	46.0 s (400%)	64.8 s (560%)
5	8.1 s (100%)	9.1 s (112%)	9.0 s (111%)	38.4 s (470%)
	System 1: SGI O2, IRIX 6.2, SGI C++ 7.2 System 2: SGI INDY, IRIX 5.3, SGI C++ 4.0 (veraltet) System 3: Sun UltraSparc 2, Solaris 5.6, GNU g++ 2.95.1 System 4: Sun SparcServer 1000, Solaris 5.5, Sun C++ 4.1 (veraltet) System 5: PC Pentium 90, Windows NT 4.0, Microsoft VisualC++ 5.0			

Tabelle 3: Geschwindigkeitsvergleich verschiedener Implementationen der Faltung

Die entsprechenden Algorithmen wurden mit einem Kern der Größe 7x7 auf ein Bild der Größe 2000x1000 angewendet. Die Werte des Bildes und des Kerns hatten den Typ `double`. Die Ergebnisse sind in Tabelle 3 zusammengefaßt. Man erkennt, daß die Geschwindigkeit der generischen Implementationen (Varianten 2 und 3) derjenigen der prozeduralen Variante (Nr. 1) nahe kommt. Dagegen ist Variante 4, die eine virtuelle Funktion benutzt, wesentlich langsamer. Der Geschwindigkeitsverlust bei den generischen Varianten erklärt sich daraus, daß die Codeoptimierung gegenwärtiger Compiler bei generischen Konstrukten noch nicht so ausgereift ist wie bei der traditionellen Zeigerarithmetik. Dies erkennt man insbesondere an den veralteten Systemen 2 und 4. Bei moderneren Compilern (Systeme 1, 3 und 5) hingegen liegt die Rechenzeit der generischen Varianten nur noch wenig über der Rechenzeit der traditionellen Implementation. Durch weitere Verbesserungen der Optimierung läßt sich prinzipiell identische Geschwindigkeit erreichen. Dies gilt jedoch nicht für Variante 4: bei Verwendung einer virtuellen Funktion ist ein zusätzlicher Funktionsaufruf, der entscheidende Codeoptimierungen verhindert, unvermeidlich.

4.12 Zusammenfassung des Kapitels

In diesem Kapitel haben wir folgendes gezeigt:

- Die generische Programmierung ist für Computer Vision-Anwendungen geeignet. Für einfache Punktoperationen können die Konzepte des C++-Standards direkt übernommen werden.
- Die 2-dimensionale Bildstruktur erfordert die Einführung von Konzepten für 2-dimensionale Iteratoren. Wir haben erstmalig einen Bilditerator definiert, der die Konzepte des C++-Standards auf diesen Fall verallgemeinert.
- Die Vielfalt der möglichen Pixeltypen und Bilddatenstrukturen erfordert die Trennung von Navigations- und Zugriffsfunktionalität. Der Datenzugriff muß deshalb in Zugriffsobjekten gekapselt werden.
- Auf der Basis dieser neuen Konzepte wurden erstmalig grundlegende Bildverarbeitungsverfahren (Punktoperationen, Faltung) generisch implementiert. Gleichzeitig haben wir einen neuen Bilddatentyp vorgestellt, der alle Anforderungen erfüllt, und bekannte Bilddatentypen aus anderen Systemen angepaßt. Die neuen generischen Bausteine sind weitgehend frei miteinander kombinierbar.
- Da algebraische Operationen für Computer Vision von grundlegender Bedeutung sind, haben wir hierfür abstrakte Konzepte einschließlich der notwendigen traits-Klassen definiert. Ein template für RGB-Tupel, das die Anforderungen an eine lineare Algebra erfüllt, demonstriert die Anwendung dieser Konzepte.
- Im Vergleich mit herkömmlichen Ansätzen konnten wir zeigen, daß die generischen Lösungen trotz ihres wesentlich geringeren Quellcodeumfangs höhere Flexibilität bzgl. Datentypen, Iterationsgebieten und Teilberechnungen besitzen. Dennoch sind sie nur wenig langsamer als explizit programmierte, inflexible Lösungen.

Kapitel 5

Iterator-Adapter

Iterator-Adapter sind Iteratoren, deren required interface ebenfalls einen Iterator anfordert. Sie bieten somit eine neue Navigationsfunktionalität auf Basis derjenigen, die durch den importierten Iterator realisiert wird. In der C++-Standardbibliothek gibt es einen solchen Iterator-Adapter, den `reverse_iterator`, der die Bedeutung von Vorwärts- und Rückwärtsiteration (`operator++` und `operator--`) vertauscht. Übergibt man einen `reverse_iterator` an einen Algorithmus, so wird die Bearbeitungsreihenfolge umgekehrt, ohne daß der Algorithmus oder die Datenstruktur deswegen umgeschrieben bzw. angepaßt werden müßten.

Im Rahmen seiner Theorie der parametrisierten Bausteine bezeichnet Batory solche Bausteine als *symmetrisch* ([BATORY98]). Ein symmetrischer Baustein aus dem realm W hat mindestens einen Parameter aus demselben realm W (vergleiche Abschnitt 2.1). Symmetrische Bausteine haben die interessante Eigenschaft, daß sie beliebig miteinander kombiniert werden können. Sie bieten die Möglichkeit, vorhandenen Bausteinen neue Funktionalität hinzuzufügen, indem man sie in einen Adapter mit der gleichen Schnittstelle einbettet. Gamma et al. bezeichnen dies als Decorator-Muster [GHJV94].

Wir werden in diesem Kapitel zeigen, daß den Iterator-Adaptern zur Lösung von Computer Vision-Problemen eine große Bedeutung zukommt. Symmetrische Iterator-Adapter ermöglichen uns beispielsweise eine sehr elegante Lösung des Randproblems: auf der Basis des gewöhnlichen 2-dimensionalen Iterators, der nur innerhalb des Bildes gültige Werte liefert, definieren wir einen Adapter mit der gleichen Schnittstelle, der auch außerhalb des Bildes abgefragt werden kann.

Neben den symmetrischen Adaptern spielen auch solche Iterator-Adapter eine große Rolle, die aus einem 2-dimensionalen Bild eine 1-dimensionale Untermenge, z.B. eine Linie, einen Kreis oder eine Kontur, extrahieren. Solche Iteratoren importieren einen Bilditerator und exportieren einen linearen Iterator, der eines der Iteratorkonzepte der C++-Standardbibliothek modelliert. Dies erlaubt uns, bestimmte Bildbearbeitungsalgorithmen so zu verallgemeinern, daß sie nur noch einen linearen Iterator anfordern, der die jeweils gewünschte Untermenge des Bildes mit Hilfe eines Adapters auswählt. Wir können beispielsweise einen Algorithmus zum Zeichnen geometrischer Figuren implementieren, der an jedem Punkt einer linearen Sequenz die gewünschte Farbe schreibt. Die gewünschte Figur (Linie, Kreis, Polygon, Spline usw.) wird durch den Adapter repräsentiert und ist dem Algorithmus selbst nicht bekannt.

Der Grund für die Wichtigkeit der Iterator-Adapter liegt in der Tatsache, daß viele Computer Vision-Algorithmen aus zwei Teilen bestehen: einem Navigationsteil und einem Arbeitsteil. Dasselbe Arbeitsteil kann mit verschiedenen Navigationsmustern kombiniert werden, ebenso wie dasselbe Navigationsmuster im Zusammenhang mit verschiedenen Aufgaben verwendet werden kann. Die oben erwähnten Iterator-Adapter für geometrische Figuren könnten beispielsweise auch dazu dienen, aus einem Bild ein Intensitätsprofil entlang der betreffenden geometrischen Figur zu extrahieren.

Iterator-Adapter führen also zu einer verbesserten Zerlegung des Systems in Bausteine und damit zu höherer Flexibilität und geringerem Programmieraufwand. Unter bestimmten Bedingungen können wir auf diese Weise sogar Algorithmen in Iteratoren verwandeln, wie wir in Abschnitt 5.8 am Beispiel der Konturverfolgung zeigen werden. Durch die einheitliche Schnittstelle der Iteratoren kann man die umgewandelten Algorithmen, also die Iterator-Adapter, sehr viel einfacher rekursiv kombinieren und ineinander schachteln als die ursprünglichen Algorithmen. Wir werden im folgenden einige interessante Iterator-Adapter entwickeln und dadurch eine Reihe von eleganten Algorithmenimplementationen ermöglichen.

5.1 Lösung des Randproblems mit Iterator-Adaptern

Wir hatten bei der Implementation der Faltung in Abschnitt 4.8 gesehen, daß der Algorithmus in der Nähe des Bildrands modifiziert werden muß. Dies gilt auch für viele andere Algorithmen, wie z.B. das Auffinden lokaler Extrema der Grauwertfunktion und das Markieren zusammenhängender Regionen (engl. connected components labeling). Oftmals hat der Code für die Sonderfälle in der Nähe des Randes einen größeren Umfang als der für den Normalfall im Inneren des Bildes. Dies kostet nicht nur zusätzliche Zeit bei der Programmentwicklung, sondern kann

auch die Lesbarkeit des Codes so verschlechtern, daß die Intention des Algorithmus nicht mehr erkennbar ist. Dazu kommt, daß in den meisten Fällen verschiedene Methoden der Randbehandlung eingesetzt werden können oder müssen. Wenn wir die Randbehandlung fest mit dem Code des Algorithmus verbinden, muß der Algorithmus für jede Art der Randbehandlung erneut implementiert werden.

Es ist daher wünschenswert, für die Randbehandlung eine unabhängige Abstraktionsachse einzuführen. Die eleganteste Möglichkeit besteht darin, dem Algorithmus ein größeres Bild vorzutäuschen als in Wirklichkeit vorhanden ist, so daß die Sonderbehandlung des Randes entfallen kann. Dafür gibt es mehrere Varianten (vergleiche z.B. das Candela-System [CANDELA98]), von denen wir die wichtigsten hier aufzählen wollen:

Konstanter Wert: Die einfachste Lösung besteht in der Annahme, daß alle Punkte außerhalb des Bildes einen konstanten Wert haben.

Nächstliegender Wert: Sehr verbreitet ist die Variante, an jeder außerhalb des Bildes liegenden Koordinate den Wert des nächstgelegenen gültigen Punktes zu verwenden. Dies ist äquivalent damit, die beiden Randzeilen und -spalten bis unendlich zu wiederholen. Diagonal außerhalb des Bildes wird der jeweils am nächsten liegende Eckpunkt verwendet.

Periodische Randbedingungen: Hier wird das Bild in allen Richtungen periodisch fortgesetzt, das heißt, nach der letzten Zeile bzw. Spalte folgt wieder die erste und umgekehrt. Genauer gewinnt man das scheinbar vergrößerte Bild g aus dem Originalbild f (mit Höhe h, Breite w) wie folgt:

$$g(x,y)=\begin{cases} f(x,y) & \text{falls} \quad 0\le x<w,\ 0\le y<h \\ f(x-kw,y-lh) & \text{falls} \quad kw\le x<(k+1)w,\ lh\le y<(l+1)h \\ & \text{mit} \quad k,l=0,\pm1,\pm2,\cdots \end{cases} \tag{5.1}$$

Reflektive Randbedingungen: Bei dieser Variante wird das Bild am Rand gespiegelt. Die Werte der äußeren Punkte entsprechen denen ihres jeweiligen Spiegelbildes im Inneren. Genauer definiert man:

$$g(x,y)=\begin{cases} f(|x|,|y|) & \text{falls} \quad -w<x<w \ \text{ und } \ -h<y<h \\ f(|x-kw_2|,|y-lh_2|) & \text{falls} \quad kw_2-w<x<kw_2+w,\ lh_2-h<y<lh_2+h \\ & \text{mit} \quad w_2=2w-1,\ h_2=2h-1 \ \text{ und } \ k,l=0,\pm1,\pm2,\cdots \end{cases} \tag{5.2}$$

Man kann alle diese Varianten beispielsweise implementieren, indem man jedes Bild vor der Bearbeitung in einen größeren Puffer kopiert. Die zusätzlichen Pixel des Puffers werden dann entsprechend der gewählten Randbehandlung mit Werten gefüllt. Mit Hilfe von Iterator-Adaptern können wir dasselbe erreichen, ohne das Bild kopieren zu müssen. Betrachten wir zunächst den Adapter für einen konstanten

Wert außerhalb des Bildes (der Anschaulichkeit wegen verwenden wir keine Zugriffsobjekte):

```
template <class ImageIterator>  // der zu adaptierende Iterator
class ConstantOutsideIterator
{
  public:
    typedef typename ImageIterator::value_type value_type;
    typedef int MoveX;
    typedef int MoveY;

    MoveX x;
    MoveY y;

    // Initialisierung mit Originaliteratoren und Wert für die Außenpunkte
    ConstantOutsideIterator(ImageIterator upperleft, ImageIterator lowerright,
                            value_type outside_value)
    : adaptee_(upperleft),
      size_(lowerright - upperleft),
      outside_value_(outside_value)
      x(0), y(0)
    {}

    // Test, ob sich der Iterator außerhalb des Bildbereichs befindet
    bool isOutside() const {
        return (x < 0 || x >= size_.x || y < 0 || y >= size_.y);
    }

    // Zugriff auf Bilddaten (innerhalb des Bildes) bzw. die Konstante
    value_type & operator*()
    {
        // Zurückgeben der Konstanten, wenn außerhalb des Bildbereichs
        if(isOutside()) return outside_value_;

        // sonst Delegation zum adaptierten Iterator
        return adaptee_(x, y);
    }
    ... // weitere Funktionen

  private:
    ImageIterator adaptee_;
    Diff2D        size_;
    value_type    outside_value_;
};
```

Die Schnittstelle dieses Iterators entspricht genau den Anforderungen an einen 2-dimensionalen Iterator gemäß Tabelle 2 (abgesehen von der zusätzlichen Funktion `isOutside()`). Wir können ihn also stets anstelle des ursprünglichen, adaptierten Iterators verwenden. Ähnlich können wir Adapter für die anderen Varianten der

Randbehandlung implementieren. Diese unterscheiden sich im wesentlichen durch die Implementation des operator*, die wir hier nur angeben wollen:

```
template <class ImageIterator>
value_type & NearestValueIterator<ImageIterator>::operator*()
{
    // Clipping der Koordinaten am Rand
    int xx = (x < 0) ? 0 : (x >= size_.x)  ? size_.x  - 1 : x;
    int yy = (y < 0) ? 0 : (y >= size_.y) ? size_.y - 1 : y;

    // gib den nächstgelegenen Wert zurück
    return adaptee_(xx, yy);
}

template <class ImageIterator>
value_type & ReflectiveBoundaryIterator<ImageIterator>::operator*()
{
    // Spiegeln der Koordinaten am Rand
    int xx = (x < 0) ? -x : (x >= size_.x)  ? 2*size_.x  - 2 - x : x;
    int yy = (y < 0) ? -y : (y >= size_.y) ? 2*size_.y - 2 - y : y;

    // gib den Wert des Spiegelbilds zurück
    return adaptee_(xx, yy);
}

template <class ImageIterator>
value_type & PeriodicBoundaryIterator<ImageIterator>::operator*()
{
    // Abbildung der Koordinaten auf die erste Periode
    int xx = (x + size_.x)  % size_.x;
    int yy = (y + size_.y) % size_.y;

    // gib den entsprechenden Wert der ersten Periode zurück
    return adaptee_(xx, yy);
}
```

Auch diese Iterator-Adapter sind symmetrisch - offered und required interface sind identisch. Die Randbehandlung ist damit für Algorithmen völlig transparent. Wir können den Faltungsalgorithmus convolveImageValidOutside() aus Abschnitt 4.8 ohne Änderung benutzen und dennoch das gesamte Bild bearbeiten. Die Annahme gültiger Werte außerhalb des Bildes wird gewährleistet, indem wir den SrcImageIterator in einen der Randadapter einbetten. Allerdings hat diese Lösung einen Schönheitsfehler: sie ist relativ langsam. Dies liegt daran, daß die Randadapter auf der Annahme beruhen, daß der Algorithmus keine Informationen über den Bildrand besitzt. Deshalb wird in operator* jedesmal eine vollständige Randüberprüfung vorgenommen. Der Faltungsalgorithmus hingegen „weiß“ im Prinzip ganz genau, wann und wo er auf den Außenbereich zugreift. Wenn wir dieses Wissen ausnutzen, können wir eine effizientere Randbehandlung definieren, die dennoch sehr flexibel ist. Wir zeigen dies in Abschnitt 5.5.

5.2 Über- und Unterabtastung

Die Über- und Unterabtastung eines Bildes ist eine andere häufig gestellte Aufgabe, die mit Hilfe von Iterator-Adaptern gelöst werden kann. Betrachten wir beispielsweise die Berechnung einer Burt-Pyramide [BURT84]. In einer Burt-Pyramide wird jedem Bild eine Folge von Bildern zugeordnet, wobei jedes Bild der Folge sich aus dem vorhergehenden durch Verkleinerung auf die halbe Bildgröße ergibt (eine präzisere Definition folgt in Abschnitt 5.3). Für die Berechnung des nächsten Bildes der Folge benötigen wir einen Iterator, der aus dem aktuellen Bild nur jede zweite Zeile und Spalte auswählt. Wir implementieren hierfür einen Iterator-Adapter, der seine aktuelle Koordinate um einen gewissen Wert vervielfacht (dies ist äquivalent dazu, jeweils eine gewisse Anzahl von Zeilen und Spalten zu überspringen):

```
template <class ImageIterator>  // der zu adaptierende Iterator
class SubsamplingIterator
{
  public:
    typedef typename ImageIterator::value_type value_type;
    typedef int MoveX;
    typedef int MoveY;

    MoveX x;
    MoveY y;

    // Initialisierung mit Originaliteratoren und Offsets
    SubsamplingIterator(ImageIterator upperleft, float xoff, float yoff)
    : adaptee_(upperleft), xoffset(xoff), yoffset(yoff),
      x(0), y(0)
    {}

    value_type & operator*()
    {
        // Zurückgeben des Wertes mit der vervielfachten Koordinate
        return adaptee_(xoffset*x, yoffset*y);
    }

    ... // weitere Funktionen

  private:
    ImageIterator adaptee_;
    float xoffset, yoffset;
};
```

In einigen Algorithmen, wie z.B. affinen Transformationen der Bildkoordinaten [WOLBERG90], benötigen wir außerdem das Umgekehrte: Funktionswerte zwischen den originalen Abtastpunkten, also an Positionen, die keine ganzzahligen, sondern

reellwertige (bzw. rationale) Koordinaten haben. In diesem Fall verwendet man oft eine bi-lineare Interpolation zwischen den benachbarten ganzzahligen Koordinaten, um einen Funktionswert an der gewünschten Stelle zu berechnen. Die bi-lineare Interpolation an der Position (x, y) ist wie folgt definiert (der Ausdruck $\lfloor x \rfloor$ steht für die nächstkleinere ganze Zahl seines Arguments):

$$\begin{aligned} x_0 &= \lfloor x \rfloor, \quad a_x = x - x_0 \\ y_0 &= \lfloor y \rfloor, \quad a_y = y - y_0 \\ f(x,y) &= (1-a_x)(1-a_y) f(x_0, y_0) + a_x (1-a_y) f(x_0+1, y_0) + \\ &\quad (1-a_x) a_y f(x_0, y_0+1) + a_x a_y f(x_0+1, y_0+1) \end{aligned} \tag{5.3}$$

Auch diesen Ausdruck kann man in einen Iterator-Adapter einbetten. Man beachte, daß die Navigationsobjekte `MoveX` und `MoveY` jetzt den Datentyp `float` haben:

```
template <class ImageIterator>  // der zu adaptierende Iterator
class BilinearInterpolatingIterator
{
  public:
    typedef typename ImageIterator::value_type value_type;
    typedef float MoveX;
    typedef float MoveY;

    MoveX x;
    MoveY y;

    // Initialisierung mit Originaliteratoren und Offsets
    BilinearInterpolatingIterator(ImageIterator upperleft)
    : adaptee_(upperleft)
      x(0), y(0)
    {}

    value_type operator*()
    {
        // nächstkleinere ganzzahlige Koordinate
        int x0 = (int)x;
        int y0 = (int)y;
        // Interpolationsfaktoren
        float ax = x - x0;
        float ay = y - y0;

        // bilineare Interpolation an der aktuellen Koordinate
        return (1.0 - ax) * (1.0 - ay) * adaptee_(x0, y0) +
               ax * (1.0 - ay) * adaptee_(x0 + 1, y0) +
               (1.0 - ax) * ay * adaptee_(x0, y0 + 1) +
               ax * ay * adaptee_(x0 + 1, y0 + 1);
    }
```

```
      ... // weitere Funktionen

  private:
    ImageIterator adaptee_;
};
```

Im Unterschied zu den bisherigen Iteratoren ist dieser Iterator im Sinne der C++-Standardbibliothek ein konstanter Iterator [AUSTERN98], weil wir mit ihm die existierenden Werte nicht überschreiben können. Für interpolierte (also berechnete) Werte existiert kein Speicherplatz im Bild, sie können deshalb nur gelesen werden.

5.3 Anwendung: Burt-Pyramide

Mit Hilfe je eines Iterator-Adapters für Randbehandlung und Unterabtastung können wir nun sehr einfach eine Burt-Pyramide erzeugen [BURT84]. Die Burt-Pyramide ordnet dem gegeben Bild eine Folge von Bildern zu, bei denen jedes gegenüber dem vorigen auf die Hälfte verkleinert wurde (Abbildung 6). Hat Bild i der Folge die Größe $w_i \times h_i$, so hat das nächste Bild (die nächste *Ebene der Pyramide*) die Größe $(w_{i+1} = \lceil w_i / 2 \rceil) \times (h_{i+1} = \lceil h_i / 2 \rceil)$, wobei halbzahlige Größen aufgerundet werden (bezeichnet durch $\lceil x \rceil$). Man bezeichnet diese Folge auch als *Auflösungspyramide*. Vor der Reduzierung der Bildgröße muß allerdings noch eine Bandbegrenzung (Glättung) erfolgen, für die Burt eine Familie von 5x5 Filtern vorgeschlagen hat, auf die wir hier nicht eingehen wollen. Burt verwendet dabei reflektive Randbedingungen:

```
FloatImage::Iterator burtfilter = ...;

int w0 = ...,      h0 = ...;       // Originale Bildgröße
int w1 = (w0+1)/2, h1 = (h0+1)/2;  // neue Bildgröße (mit Aufrunden)

FloatImage level0(w0, h0);         // Originalbild
FloatImage level1(w1, h1);         // nächstkleinere Kopie
FloatImage tmp(w0, h0);

// Kapseln des Quelliterators in Adapter für reflektive Randbedingungen
ReflectiveBoundaryIterator<FloatImage::Iterator>
                        ul_0(level0.upperLeft(), level0.lowerRight()),
                        lr_0 = ul_0 + Diff2D(w0, h0);
StandardAccessor<float> accessor;

// Glättung zur Bandbegrenzung
convolveImageValidOutside(ul_0, lr_0, accessor, tmp.upperleft(), accessor,
                          burtfilter, Diff2D(-2, -2), Diff2D(2, 2));
```

Abbildung 6: Drei Ebenen einer Burt-Pyramide

```
 // Kapseln des Iterators für das bandbegrenzte Bild in SubsamplingIterator
SubsamplingIterator<FloatImage::Iterator>  ul_tmp(tmp.upperLeft(), 2.0, 2.0),
                                           lr_tmp = ul_tmp + Diff2D(w1, h1);

// Kopieren der ausgewählten Zeilen und Spalten in level1
copyImage(ul_tmp, lr_tmp, accessor, level1.upperLeft(), accessor);
```

Das Vergrößern erfolgt in genau umgekehrter Reihenfolge: die Werte des jeweils kleineren Bilds werden in jede zweite Zeile und Spalte des größeren kopiert (die übrigen Punkte müssen zuvor mit 0 initialisiert werden, wofür man z.B. `initImage()` verwenden kann). Danach wird das vergrößerte Bild mit dem `burtfilter` geglättet, wodurch die fehlenden Zeilen und Spalten interpoliert werden.

5.4 Iteratoren für lineare Untermengen des Bildes

Wir wollen uns nun Adaptern zuwenden, die 2-dimensionale Iteratoren in lineare Iteratoren transformieren, d.h. in Iteratoren, die eine 1-dimensionale Untermenge der Punkte eines Bildes adressieren. Solche Adapter sind für alle diejenigen Algorithmen interessant, die nur auf eine lineare Untermenge angewendet werden sollen: diese Algorithmen werden davon entlastet, die gewünschte Untermenge selbst auswählen zu müssen. Dies führt nicht nur zu einer Vereinfachung, sondern gestattet auch die Verallgemeinerung der Algorithmen auf beliebige Untermengen, die jeweils durch entsprechende Adapter festgelegt werden.

Die wichtigsten 1-dimensionalen Untermengen sind zweifellos die Zeilen und Spalten eines Bildes. Der transparente Zugriff auf Zeilen und Spalten eines Bildes ermöglicht uns beispielsweise eine elegante Implementation der separierbaren

Faltung, wie wir im nächsten Abschnitt zeigen werden. Das Prinzip von Iterator-Adaptern für Zeilen und Spalten ist einfach: wir implementieren die Navigationsfunktionen und die Indexoperation der Adapter so, daß sie nur auf eine Koordinate des Bilditerators wirken. Der operator* kann ohne Änderung übernommen werden. Der ColumnIterator beispielsweise navigiert auf einer bestimmte Spalte des Bildes (der Anschaulichkeit halber verwenden wir keine Zugriffsobjekte):

```
template <class ImageIterator>
class ColumnIterator
{
    ImageIterator adaptee_;
  public:
    typedef typename ImageIterator::value_type  value_type;

    // Initialisierung mit Bilditerator
    ColumnIterator(ImageIterator const & i) : adaptee_(i)  {}

    //  Navigationsoperation wirkt nur auf Y
    ColumnIterator & operator++()  {  ++ adaptee_.y;  }

    // ColumnIteratoren sind gleich, wenn ihre Bilditeratoren gleich sind
    bool operator==(ColumnIterator const & o) const  {
        return adaptee_ == o.adaptee_;
    }

    // Zugriffsoperator wird an Bilditerator weitergereicht
    value_type & operator*()  {  return  *adaptee_;  }

    // Indexoperator wirkt nur auf y-Koordinate (x-Offset ist 0)
    value_type & operator[](int dy) { return *adaptee_(0, y); }

    ... // weitere Funktionen
};
```

Entsprechend wird der RowIterator für Zeilen implementiert. Gelegentlich werden darüber hinaus anders orientierte Linien benötigt, etwa wenn wir eine Strecke zeichnen oder das Grauwertprofil entlang einer beliebigen Geraden aufnehmen möchten. Wir definieren hierfür den LineIterator, der eine leicht modifizierte Version des inkrementellen scan conversion-Algorithmus gemäß [FOLEY+93] realisiert. Dieser Algorithmus ermittelt diejenigen Gitterpunkte im $\mathbb{Z}^2$, die eine im $\mathbb{R}^2$ gegebene Linie optimal approximieren. Seien p_1 und p_2 Anfangs- und Endpunkt der gewünschten Linie und $d_x = x_2 - x_1$ und $d_y = y_2 - y_1$ deren Abstände in x- bzw. y-Richtung. Der Algorithmus bestimmt dann die Anzahl der Schritte, um von p_1 nach p_2 zu gelangen, durch die L∞-Distanz der beiden Punkte: $d = \max(|d_x|,|d_y|)$. Die notwendigen Inkremente in x- und y-Richtung ergeben sich als Quotienten der Abstände und der Schrittzahl: $i_x = d_x / d$ und $i_y = d_y / d$. Nach jedem Schritt erhält man damit eine reellwertige Koordinate, die auf den nächstliegenden Gitterpunkt gerundet wird (siehe Abbildung 7).

Wir initialisieren diesen Iterator, indem wir Bilditeratoren auf den Anfangs- und Endpunkten der gewünschten Linie übergeben. Daraus werden im Konstruktor die Inkremente berechnet. Die Navigationsoperationen des `LineIterator` inkrementieren bzw. dekrementieren dann die x- und y-Koordinaten des eingebetteten Bilditerators entsprechend:

```
template <class ImageIterator>
class LineIterator
{
    ImageIterator iter_;   // eingebetteter Iterator
    float x_, y_;          // aktuelle (reellwertige) Koordinate des Iterators
    float ix_, iy_;        // Inkremente in x- und y-Richtung
  public:
    typedef typename ImageIterator::value_type  value_type;

    LineIterator(ImageIterator start, ImageIterator end)
    : iter_(start), x_(0.0), y_(0.0), ix_(0.0), iy_(0.0)
    {
        int dx = end.x - start.x;           // x-Distanz
        int dy = end.y - start.y;           // y-Distanz
        float dd = max(abs(dx), abs(dy));   // L∞-Distanz
        if(dd > 0.0)  {
            ix_ = dx / dd;                   // x-Inkrement
            iy_ = dy / dd;                   // y-Inkrement
        }
    }

    // inkrementieren
    LineIterator & operator++() {
        x_ += ix_;
        y_ += iy_;
        return *this;
    }

    bool operator==(LineIterator const & o) const {
        return x_ == o.x_ && y_ == o.y_;
    }

    // Zugriffsoperation: gib den Wert des nächstgelegenen Gitterpunktes zurück
    // (hierfür gibt es effizientere Implementierungen, die jedoch nicht so
    //  leicht lesbar sind, vgl. [FOLEY+93])
    value_type & operator*()  {  return iter_(round(x_), round(y_));  }

    static int round(float f) { return (int)( (f < 0) ? f - 0.5 : f + 0.5); }
    ...
};
```

Nach dem gleichen Muster können wir Iteratoren für andere eindimensionale Strukturen, wie Kreise und Polygone, definieren. Mit dieser Art von „geometrischen

Iteratoren“ ist es möglich, die Beschreibung der geometrischen Strukturen von den Operationen zu trennen, die auf diesen Strukturen ausgeführt werden sollen. Dadurch erhalten wir z.B. die Möglichkeit, einen generischen Algorithmus zum Zeichnen von geometrischen Figuren zu definieren:

```
template <class ShapeIterator, class Color>
void drawShape(ShapeIterator i, ShapeIterator end, Color color)
{
    do {  *i = color; } while(++i != end);
}
```

Ganz analog lassen sich andere Algorithmen, wie beispielsweise die Gewinnung eines Grauwertprofils entlang einer beliebig geformten und orientierten Schnittlinie, verallgemeinern. Wesentlich ist dabei die Beobachtung, daß auch viele andere Algorithmen in einen navigierenden und einen manipulierenden Teil zerlegbar sind. Iterator-Adapter erlauben dann eine saubere Trennung der beiden Teile in zwei unabhängige Bausteine und führen somit zu einer eleganten und flexiblen Lösung.

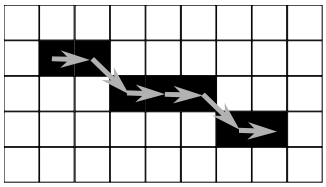


Abbildung 7: Weg eines `LineIterator` vom Punkt (1,1) zum Punkt (7,3)

5.5 Anwendung: Separierbare Faltung

Viele Faltungskerne, darunter die in der Computer Vision-Praxis wichtigen Kerne der Binomial- und Gaußfilter sowie deren Ableitungen, sind *separierbar*. Das heißt, wir können die Faltung mit einem dieser 2-dimensionalen Kerne durch die Konkatenation von je einer Faltung mit einem 1-dimensionalen Kern in x- und y-Richtung ersetzen [JÄHNE91]:

$$(g * f)(x,y) = ((g_2 \cdot g_1) * f)(x,y) = (g_2 * (g_1 * f))(x,y) \tag{5.4}$$

Dabei sind g_1 und g_2 die 1-dimensionalen Kerne in x- bzw. y-Richtung, deren Euklidisches Produkt $(g_2 \cdot g_1)$ dem 2-dimensionalen Kern g entspricht. Die Verwendung der 1-dimensionalen Kerne führt zu einer erheblichen Verringerung des Berechnungsaufwandes. Hat der originale Kern die Größe mxn, so haben die 1-dimensionalen Kerne nur noch die Größe mx1 bzw. 1xn. Da die Anzahl der auszu-

führenden Operationen der Größe des Kerns proportional ist, verringert sich der Aufwand von *O(mn)* auf *O(m+n)* pro Bildpunkt.

Bevor wir die Implementation der 1-dimensionalen Faltung angeben, wollen wir jedoch noch einmal auf das Randproblem zurückkommen. Wir hatten festgestellt, daß die in Abschnitt 5.1 angegebene Lösung des Randproblems mit Iterator-Adaptern im Falle der Faltung nicht sehr effizient ist, weil der Adapter mehr Tests durchführt als notwendig sind. Wir wollen uns bei der Entwicklung einer besseren Lösung auf die im Kontext der Faltung besonders geeigneten Randbehandlungsvarianten *nächstliegendes Element* sowie *reflektive* und *periodische Randbedingungen* beschränken.

Diese Varianten haben eine wichtige Gemeinsamkeit: am Rand und außerhalb des Randes geht die Iteration wie gewohnt weiter, nur die *Inkremente ändern sich*. Bei reflektiven Randbedingungen kehrt sich das Vorzeichen des Inkrements um. Bei periodischen Randbedingungen müssen wir, anschaulich gesprochen, einmal einen Sprung auf die gegenüberliegende Seite des Bildes machen, danach geht die Iteration wie gewohnt weiter. Auch den Zugriff auf das nächstliegende Element können wir so interpretieren, daß am Rand die Inkremente einfach 0 gesetzt werden, so daß wir auf dem letzten gültigen Element stehenbleiben. Um jede dieser drei Methoden zu charakterisieren, müssen wir nur zwei Zahlen angeben: das erste Inkrement nach Erreichen des Randes, und das dann folgende Inkrement. Wir werden diese Informationen in einem `pair<int,int>` an den Algorithmus übergeben. Für die verschiedenen Varianten müssen die Inkremente wie folgt gesetzt werden:

```
pair<int, int>   nearestValidPixel(0, 0);

pair<int, int>   reflectAtBorder(-1, -1);

pair<int, int>   periodicBorder(-size+1, 1);  // size ist die Länge des Signals
```

Die übrigen Argumente des Faltungsalgorithmus bleiben bestehen, sie sind aber jetzt 1-dimensional. Wir wollen die Funktion deshalb `convolveSignal()` nennen:

```
template <class SrcIterator, class SrcAccessor,
          class DestIterator, class DestAccessor,
          class KernelIterator>
void convolveSignal(SrcIterator src_begin, SrcIterator src_end, SrcAccessor srca,
                    DestIterator dest_begin, DestAccessor desta,
                    KernelIterator kcenter, int k_left, int k_right,
                    pair<int, int> border_increments)
{
    SrcIterator s = src_begin;
    DestIterator d = dest_begin;

    // äußere Schleife über alle Punkte
    for(; s < src_end; ++s, ++d)
    {
```

```
            // temporäre Variable für die Summe
            typedef typename
                NumericTraits< typename SrcAccessor::value_type >::ScalarPromote
                SumType;
            SumType sum = NumericTraits<SumType>::zero();

            // feststellen, wieviel des Kerns innerhalb des Signals liegt
            int left = max(k_left, src_begin - s);
            int right = min(k_right, src_end - s - 1);
            int i, k;

            // erste Schleife: alle Punkte, wo Kern im Innern der Signals liegt
            for(i = left; i <= right; ++i)
            {
                sum += kcenter[-i] * srca.get(s, i);
            }

            // zweite Schleife: weiter nach links mit Hilfe der border_increments
            i = left - 1;
            k = left - border_increments.first;
            for(; i >= k_left; --i, k -= border_increments.second)
            {
                sum += kcenter[-i] * srca.get(s, k);
            }

            // dritte Schleife: weiter nach rechts mit Hilfe der border_increments
            i = right + 1;
            k = right + border_increments.first;
            for(; i <= k_right; ++i, k += border_increments.second)
            {
                sum += kcenter[-i] * srca.get(s, k);
            }

            // Schreiben des Ergebnisses
            desta.set(sum, d);
        }
    }
```

Diese Funktion kann nicht nur auf 1-dimensionale Untermengen eines Bildes, z.B. Zeilen, Spalten und beliebig orientierte Linien, angewendet werden, sondern auf alle Datenstrukturen, die Random Access Iteratoren nach der Definition der C++-Standardbibliothek anbieten, wie z.B. die Iteratoren der Standardklasse `vector` und gewöhnliche Zeiger auf ein C-Array.

Für die separierbare Faltung in x-Richtung müssen wir die 1-dimensionale Faltung auf jede Zeile des Bildes anwenden. Dazu erzeugen wir für jede Zeile `RowIteratoren` und rufen mit ihnen `convolveSignal()` auf:

```
template <class SrcImageIterator, class SrcAccessor,
          class DestImageIterator, class DestAccessor,
          class KernelIterator>
void separableConvolveImageX(
    SrcImageIterator src_ul, SrcImageIterator src_lr, SrcAccessor srca,
    DestImageIterator dest_ul, DestAccessor desta,
    KernelIterator kcenter, int k_left, int k_right, pair<int,int> border_incr)
{
    int w = src_lr.x - src_ul.x;   // Breite des Bildes

    // Schleifen die erste Spalte abwärts
    for(; src_ul.y < src_lr.y; ++src_ul.y, ++dest_ul.y)
    {
        // RowIteratoren für die aktuelle Zeile erzeugen
        RowIterator<SrcImageIterator>  src_begin(src_ul);
        RowIterator<DestImageIterator> dest_begin(dest_ul);

        convolveSignal(src_begin, src_begin+w, scra, dest_begin, desta,
                       kcenter, k_left, k_right, border_incr);
    }
}
```

Analog implementieren wir das Filter in y-Richtung mit `ColumnIteratoren`.

5.6 Anwendung: Rekursive Filter

Die Einführung von Iterator-Adaptern für Zeilen und Spalten ermöglicht auch eine elegante Implementation der sogenannten *rekursiven Filter* auf Bildern, die von Deriche in die Computer Vision-Forschung eingeführt wurden [DERICHE90]. Rekursive Filter haben diesen Namen erhalten, weil sie das Filterergebnis an einem Punkt mit Hilfe der Ergebnisse an bereits bearbeiteten Punkten berechnen. Dadurch sind sie sehr effizient: unabhängig von der Skala des Operators[17] werden (bei Filtern 1. Ordnung) pro Punkt nur drei Multiplikationen und drei Additionen benötigt. Allerdings sind rekursive Filter nur im 1-dimensionalen definiert, aber wir können sie, wie die separierbaren Filter, in beiden Richtungen nacheinander ausführen. Zwar ist das resultierende 2-dimensionale Filter nicht vollkommen isotrop, man kann jedoch die Abweichung häufig vernachlässigen bzw. sie nach einem Vorschlag von Lanser und Eckstein [LANSECK92] nachträglich korrigieren.

[17] Als Skala eines Filters bezeichnet man einen Parameter, mit dem man die Größe des Filters einstellen kann. Die Skala bestimmt also den Einzugsbereich des Filters, was bei Glättungsfiltern der Stärke der Glättung entspricht. Beispielsweise wird die Skala von Gaußfiltern durch die Standardabweichung angegeben (vgl. z.B. [WITKIN83, LINDEHAAR94]).

Man kann die rekursive Filtertechnik beispielsweise verwenden, um das Exponentialfilter zu implementieren. Der Kern des Exponentialfilters ist gegeben durch

$$g(x) = a\, e^{\frac{-|x|}{\sigma}} = a\, b^{|x|} \quad \text{mit } b = e^{-\frac{1}{\sigma}} = \text{const} < 1 \tag{5.5}$$

wobei a eine Normierungskonstante und $\sigma>0$ die Skala des Operators ist. Um die Möglichkeit der rekursiven Implementation zu erkennen, wollen wir das Faltungsprodukt als unendliche Reihe ausschreiben:

$$\begin{aligned}(g * f)(x) &= \sum_{i=-\infty}^{\infty} g(-i) f(x+i) = \sum_{i=-\infty}^{\infty} a\, b^{|i|} f(x+i) \\ &= a\Big[f(x) + \big(b f(x-1) + b^2 (f(x-2) + \cdots) + \big(b f(x+1) + b^2 f(x+2) + \cdots\big)\Big] \\ &= a\Big[f(x) + b\Big(f(x-1) + b\Big(f(x-2) + \cdots\Big)\Big) + b\Big(f(x+1) + b\Big(f(x+2) + \cdots\Big)\Big)\Big]\end{aligned} \tag{5.6}$$

Durch das wiederholte Ausklammern von b in der letzten Formulierung wird klar, wie der Ausdruck rekursiv implementiert werden muß: wir benötigen zwei rekursive Filter, je einen für die rechte und linke Flanke der Exponentialfunktion. Wir nennen diese Filter F_R und F_L. Die Werte von F_R und F_L an der Position x können dann stets aus dem aktuellen Funktionswert an der Stelle x und den Werten von F_R und F_L an den Positionen $x\pm1$ berechnet werden:

$$F_R(x) = f(x) + b\, F_R(x+1) \qquad F_L(x) = f(x) + b\, F_L(x-1) \tag{5.7}$$

Damit ergibt sich als rekursive Implementation des Exponentialfilters:

$$(g * f)(x) = \sum_{i=-\infty}^{\infty} a\, b^{|i|} f(x+i) = a\Big(f(x) + b\Big(F_L(x-1) + F_R(x+1)\Big)\Big) \tag{5.8}$$

Die Normierungskonstante a können wir mit Hilfe der Bedingung bestimmen, daß ein konstantes Signal durch das Filter nicht verändert werden soll. Nehmen wir beispielsweise an, daß das Signal überall den Wert 1 hat und setzen dies in Formel (5.8) ein, erhalten wir:

$$1 = (g * [f = 1])(x) = a \sum_{i=-\infty}^{\infty} b^{|i|} = a\left(2 \sum_{i=0}^{\infty} b^i - 1\right) = 1 \tag{5.9}$$

Da $b<1$ ist, hat die unendliche geometrische Reihe $\Sigma\, b^i$ den Grenzwert $1/(1-b)$. Wir erhalten also für die Normierungskonstante a:

$$a = \frac{1-b}{1+b} \tag{5.10}$$

Als Randbedingung wählt man bei rekursiven Filtern am einfachsten den Modus, der den jeweiligen Randwert bis unendlich wiederholt. Für die linke Flanke heißt

das, wir nehmen an, daß alle Pixel von -∞ bis 0 den Wert von $f(0)$ hätten. Damit ergibt sich als Initialisierung $F_L(-1)=f(0)/(1-b)$. Analog erhalten wir $F_R(w)=f(w-1)/(1-b)$ für die rechte Flanke. Wir erhalten somit folgende generische Implementation des 1-dimensionalen Exponentialfilters:

```
template <class SrcIterator, class SrcAccessor,
          class DestIterator, class DestAccessor>
void exponentialFilter(SrcIterator src, SrcIterator end, SrcAccessor sa,
                       DestIterator dest, DestAccessor da, double scale)
{
    SrcIterator begin = src;

    typedef typename
        NumericTraits<typename SrcAccessor::value_type>::ScalarPromote TmpType;

    // generiere ein temporäres Array für F_L
    vector<TmpType>  F_L(end - src);
    vector<TmpType>::iterator  f = F_L.begin();

    double b = exp(-1.0 / scale);
    double a = (1.0 - b) / (1.0 + b);

    // Berechnung von F_L
    *f = sa.get(src) / (1.0 - b);
    for(++src, ++f; src != end; ++src, ++f)
    {
        *f = sa.get(src) + b * f[-1];
    }

    // Berechnung von F_R und des Filterergebnisses
    --src; --f;  dest += src - begin;
    TmpType F_R = sa.get(src) * b / (1.0 - b);
    for(; src != begin; --src, --dest, --f)
    {
        da.set(a * (*f + F_R), dest);
        F_R = b * (sa.get(src) + F_R);
    }

    da.set(a * (*f + F_R), dest);
}
```

Man erkennt, daß der Datentyp des Quellbildes die Anforderungen an einen linearen Raum über dem Körper `double` erfüllen muß, so daß dieser Algorithmus auf alle bisher betrachteten Pixeltypen, auch auf RGB-Werte, anwendbar ist. Um die Funktion `exponentialFilter()` auf Bilder anzuwenden, kapseln wir die Bilditeratoren in den Adaptern `RowIterator` und `ColumnIterator`, wie wir dies im vorigen Abschnitt am Beispiel der separierbaren Faltung gezeigt hatten.

5.7 Adapter für Punktnachbarschaften

Bei einigen Algorithmen besteht die Notwendigkeit, auf die Nachbarpunkte eines gegebenen Punkts zuzugreifen. Dies gilt beispielsweise für die Ermittlung lokaler Extrema der Bildfunktion und für die Bestimmung zusammenhängender Regionen (engl. connected components labeling) [HORN86]. Dabei wird entweder die 4-Nachbarschaft (als Nachbarn gelten die vier horizontal und vertikal direkt angrenzenden Punkte) oder die 8-Nachbarschaft (die diagonal angrenzenden Punkte gelten ebenfalls als benachbart) verwendet.

Wir wollen zunächst einen Adapter für die 8-Nachbarschaft entwickeln. Dafür ist es üblich, die 8 Nachbarn mit Hilfe eines Richtungscodes (des sogenannten Freeman-Codes) zu numerieren [FREEMAN61, HABER91]:

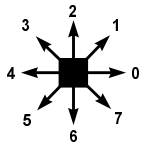


Abbildung 8: Schema der Numerierung der 8-Nachbarn eines Punktes (Richtungscodierung)

Wir wollen den Iterator-Adapter als Modell des Konzepts BidirectionalIterator implementieren. Das heißt, der Adapter referenziert stets einen der acht Nachbarn, und durch Anwenden von `operator++` und `operator--` wird er auf den nach der Richtungscodierung jeweils folgenden bzw. vorhergehenden Nachbarn gesetzt.

Zusätzlich führen wir eine neue Navigationsoperation `jumpToNeighbor()` ein, die das Zentrum des Adapters auf den aktuellen Nachbarn verschiebt und gleichzeitig die Richtung um 180 Grad dreht. Anschaulich gesprochen heißt dies, daß die Rollen von Zentrum und Nachbar vertauscht werden: der Adapter folgt einem Pfeil gemäß Abbildung 8 und kehrt dann seine Richtung um.

Unter diesen Voraussetzungen erhalten wir die folgende Implementation (wir werden im Anschluß erklären, warum wir den Adapter als `Circulator` bezeichnen):

```
template <class ImageIterator>
class Neighborhood8Circulator
{
    ImageIterator center_;
    byte direction_;

public:
    // Initialisierung mit ImageIterator für das Zentrum und Anfangsrichtung
    Neighborhood8Circulator(ImageIterator center, int direction = 0)
    : center_(center), direction_(direction)
    {}
```

```
    // Inkrementierung des Richtungscodes modulo 8
    Neighborhood8Circulator & operator++() {
        (direction_ + 1) % 8; return *this;
    }

    // Dekrementierung des Richtungscodes modulo 8
    Neighborhood8Circulator & operator--() {
        (direction_ + 7) % 8; return *this;
    }

    // Verschiebung des Zentrums und Umkehrung der Richtung
    void jumpToNeighbor() {
         center_ += differenceVector();
         direction_ = (direction_ + 4) % 8;
    }

    // Abfagen des aktuellen Richtungscodes
    int directionCode() const { return direction_; }

    // Dekodierung der Richtungscodes in Differenzvektoren
    Diff2D differenceVector() const {
         static int dx[] = {1, 1, 0, -1, -1, -1, 0, 1};
         static int dy[] = {0, -1, -1, -1, 0, 1, 1, 1};

         return Diff2D(dx[direction_], dy[direction_]);
    }

    // Abfagen des Zentrums
    ImageIterator center() const { return center_; }

    // Zugriff auf den aktuellen Nachbarn
    ImageIterator operator*() {
         return (center_ + differenceVector());
    }

    // Vergleich (nur Adapter mit dem gleichen Zentrum dürfen verglichen werden)
    bool operator==(Neighborhood8Circulator const & o) const {
         return direction_ == o.direction_;
    }
};
```

Ein Iterator für die 4-Nachbarschaft wird ebenso implementiert, mit dem Unterschied, daß nur die geradzahligen Richtungscodes verwendet werden. Der Inkrementierungsoperator muß also statt zur nächsten Richtung zur übernächsten weitergehen:

```
Neighborhood4Circulator & Neighborhood4Circulator::operator++() {
    (direction_ + 2) % 8; return *this;
}              // ^ Weitergehen zur übernächsten Richtung
```

Entsprechend wird die Dekrementierungoperation modifiziert. Weitere Änderungen sind gegenüber dem `Neighborhood8Circulator` nicht notwendig.

An der Implementierung der Operation zum Inkrementieren und Dekrementieren der Iteratoren erkennen wir eine wichtige Besonderheit: die Iteratoren zeigen *zyklisches Verhalten*. Wenn sie in Richtung 7 (8-Nachbarschaft) bzw. 6 (4-Nachbarschaft) zeigen, folgt als nächstes wieder Richtung 0. Dies entspricht der erwünschten Semantik, denn die Festlegung der Richtungscodes in Abbildung 8 ist vollkommen willkürlich geschehen. Aus Sicht der Algorithmen gibt es normalerweise keine „erste" oder „letzte" Richtung, sondern es werden relative Richtungen wie die „folgende", die „vorherige" oder die „gegenüberliegende" verwendet.

Allerdings erfüllt dieses Verhalten nicht die Anforderungen der Konzepte der C++-Standardbibliothek. Die Standardkonzepte setzen voraus, daß jede Sequenz ein ausgezeichnetes "past-the-end"-Element hat, welches zur Festlegung der Iterationsgrenzen benutzt werden kann. Eine zyklische Struktur besitzt kein "past-the-end"-Element, d.h. alle erreichbaren Elemente der Struktur sind gültig. Deshalb ist es notwendig, für zyklische Strukturen neue Konzepte, die sogenannten *Zirkulatoren*, einzuführen.

5.7.1 Zirkulatoren

In Computer Vision haben wir es häufig mit zyklischen Datenstrukturen zu tun. Wir haben soeben mit den Nachbarschaft-Zirkulatoren ein erstes Beispiel dafür gefunden. Das Gleiche gilt für Iteratoren, die Polygone oder Kreise repräsentieren: diese Strukturen sind ihrem Wesen nach zyklisch, auch wenn man aus praktischen Gründen meist einen Punkt als Anfangspunkt auszeichnet.

Formell beschrieben wird zyklisches Verhalten durch das Konzept des *Zirkulators* [KETTNER98]. Wie bei den Iteratoren unterscheidet man Forward-, Bidirectional- und RandomAccess Zirkulatoren. Es gilt folgende Definition:

Zirkulatoren: Zirkulatoren der Konzepte Forward-, Bidirectional- und RandomAccessCirculator unterstützen dieselbe Navigationsfunktionalität wie die entsprechenden Iteratoren, aber die Semantik der Operationen spiegelt die zyklische Struktur der unterliegenden Sequenz: während bei Iteratoren fortgesetztes Inkrementieren schließlich zum Ende der Sequenz führt, kommt man bei einem Zirkulator stets wieder zum Ausgangspunkt zurück. D.h., es gibt stets eine Zahl $n>0$, so daß gilt:

```
Circulator i = ...;
advance(i, n);
assert(i == i);
```

Allen Zirkulatorpositionen entsprechen gültige Elemente der Sequenz, es gibt kein "past-the-end" Element.

Zyklisches Verhalten geht über die Konzeption des C++ Standards hinaus, weil die Voraussetzung, daß jede Sequenz ein wohldefiniertes Ende hat, nicht mehr erfüllt ist. Das hat unter anderem zur Folge, daß ein Iteratorpaar (`begin, end`) nicht mehr ausreicht, um die Iterationsregion eindeutig zu bestimmen. Man sieht dies am einfachsten an dem Paar (`begin, begin`), d.h. wenn Anfang und Ende der Sequenz identisch sind. Bei Iteratoren bezeichnet dies stets die leere Sequenz. Bei zyklischen Datenstrukturen ist man hingegen geneigt, einen vollständigen Umlauf für die sinnvollste Interpretation zu halten [KETTNER98]. Eine leere Sequenz wird statt dessen durch einen *singulären Zirkulator* ausgedrückt. Ein Zirkulator benötigt deshalb eine zusätzliche Funktion `isSingular()`, mit der Algorithmen feststellen können, ob eine leere Sequenz übergeben wurde.

Man könnte die Definition von Zirkulatoren vermeiden, wenn man in jede zyklische Sequenz zwangsweise ein "past-the-end"-Element einfügt. Dies würde jedoch die natürliche Symmetrie der unterliegenden Struktur zerstören, denn es gibt keine ausgezeichnete Position, an der man das zusätzliche Element plazieren sollte. Algorithmen, die einen vollen Umlauf ausführen, beginnen nicht immer an derselben Startposition (vor der man die Endmarkierung natürlicherweise einfügen würde). Statt dessen beginnt und endet die Iteration bei einem beliebigen Element. In diesem Falle stört ein künstliches Endelement an einer beliebigen Stelle mehr als es nützt, denn es muß immer übersprungen werden, wenn die Iteration an einer anderen Stelle enden soll. Es ist deshalb günstiger, die Zirkulatoren so zu definieren, wie es ihr logisches Verhalten erfordert.

Wenn ein Algorithmus unbedingt eine Endmarkierung benötigt, kann man diese auch nachträglich mittels eines Zirkulator-Adapters einfügen. Ein solcher Adapter speichert zusätzlich eine *Windungszahl* Die Windungszahl zählt die Anzahl der vollen Umläufe des gekapselten Zirkulators. Sie wird jedesmal um eins erhöht, wenn der Adapter sein Ausgangselement wieder erreicht. Zwei Zirkulator-Adapter sind gleich, wenn sie auf dasselbe Element zeigen *und* ihre Windungszahlen gleich sind. Der Adapter muß zusätzlich einen Zirkulator auf sein Anfangselement speichern, so daß er in der Lage ist, einen vollständigen Umlauf zu erkennen. Dies führt zu folgender Implementation:

```
template <class Circulator>
class CirculatorAdapter
{
    Circulator begin_, current_;
    int winding_number_;

  public:
    CirculatorAdapter(Circulator begin, int winding_number)
    : begin_(begin), current_(current), winding_number_(winding_number)
    {}
```

```
    CirculatorAdapter & operator++() {
        ++current_;
        if(current_ == begin_) ++winding_number_;
        return *this;
    }

    bool operator==( CirculatorAdapter const & o) const {
        return (current_ == o.current_) &&
               (winding_number_ == o.winding_number_);
    }
    ...
};
```

Auf diese Weise ist es sehr einfach, jeden beliebigen Ausschnitt einer zyklischen Struktur, einschließlich mehrfacher Umläufe, durch ein zu den Standardkonzepten konformes Adapterpaar zu spezifizieren. Ein vollständiger Umlauf, beginnend bei einem beliebigen Element, wird dadurch gekennzeichnet, daß Anfangs- und Enditerator auf dasselbe Element zeigen, ihre Windungszahlen sich jedoch um eins unterscheiden:

```
Circulator i = ...;
CirculatorAdapter<Circulator> begin(i, 0), end(i, 1);
```

5.7.2 Anwendung: Erkennung lokaler Minima der Bildfunktion

Eine einfache Anwendung des `Neighborhood8Circulator` ist ein Algorithmus zur Detektion lokaler Minima in einem Bild. An jedem Punkt testen wir, ob der aktuelle Wert kleiner ist als alle seine Nachbarn. Die Nachbarn finden wir dabei mit Hilfe des Zirkulators. Wir können diesen Algorithmus folgendermaßen implementieren:

```
template <class SrcImageIterator, class SrcAccessor,
          class DestImageIterator, class DestAccessor,
          class MarkerType>
void localMinimaOfImage(
    SrcImageIterator src_ul, SrcImageIterator src_lr, SrcAccessor srca,
    DestImageIterator dest_ul, DestAccessor desta, MarkerType minimum_marker)
{
    // Schleifen über alle Punkte
    for(;src_ul.y < src_lr.y; ++ src_ul.y, ++ dest_ul.y)
    {
        ImageIterator1 scurrent = src_ul;
        ImageIterator2 dcurrent = dest_ul;

        for(; scurrent.x < src_lr.x; ++scurrent.x, ++dcurrent.x)
        {
            // Nachbarschaftzirkulator an der aktuellen Position
            Neighborhood8Circulator<SrcImageIterator> neighbors(scurrent);
```

```
                // Schleife über alle Nachbarn (wir wissen, daß es 8 Nachbarn gibt)
                int i;
                for(i=0; i<8; ++i, ++neighbors) {
                    // teste, ob Zentralwert kleiner als Nachbarwert ist
                    if(!(srca(scurrent) < srca(*neighbors)) break;
                }

                // wenn i==8 => Minimum
                if( i==8) desta.set(minimum_marker, dcurrent);
            }
        }
    }
```

Man beachte, daß der Zirkulator nicht prüft, ob der aktuelle Nachbar noch innerhalb des Bildes liegt. Wir können dies sichern, indem wir den Arbeitsbereich um ein Pixel verkleinern. Alternativ können wir auch von vornherein Iterator-Adapter für reflektive Randbedingungen an die Funktion `localMinimaOfImage()` übergeben, um das gesamte Bild bearbeiten zu können. Die Verwendung des Nachbarschaftszirkulators vereinfacht den Algorithmus, weil er nicht mehr wissen muß, wie man zu einem gegebenen Punkt die Nachbarn findet. Wir könnten z.B. einen Zirkulator für die 4-Nachbarschaft einführen, ohne daß weitere Änderungen notwendig werden.

5.8 Kontur-Zirkulator

Zum Abschluß dieses Kapitels wollen wir einen Adapter definieren, mit dem wir die Kontur eines Objekts in einem Binärbild extrahieren können. Da eine Kontur zyklisch ist, wird es sich dabei um einen Zirkulator handeln. Dieser Zirkulator ist aus zwei Gründen interessant: erstens demonstriert er sehr gut, wie man Iterator-Adapter mehrfach ineinander schachteln kann, und zweitens ist er ein erstes Beispiel für Konzepte, die wir im Zusammenhang mit der Bildsegmentierung in den folgenden Kapiteln ausführlich diskutieren wollen.

Wir verwenden in diesem Abschnitt die Definition von Pavlidis [PAVLIDIS82].[18] Pavlidis definiert die Kontur einer Region allgemein als die Menge derjenigen Punkte (Pixel) der Region, die Nachbarn sowohl innerhalb als auch außerhalb der Region besitzen. Je nach Festlegung der Nachbarschaft ergeben sich daraus zwei Varianten: die 4-Kontur einer Region enthält alle Punkte der Region, von denen wenigstens ein 8-Nachbar außerhalb der Region liegt. Entsprechend enthält die 8-Kontur alle Punkte, von denen wenigstens ein 4-Nachbar außerhalb liegt. Die Bezeichnungen 4-Kontur und 8-Kontur ergeben sich daraus, daß die Konturpunkte

[18] Eine bessere Definition werden wir in Kapitel 7 angeben.

untereinander jeweils über 4- bzw. 8-Nachbarschaft verbunden sind. Daher sollte man die 4-Kontur verwenden, wenn man bei der Bestimmung der Regionen 4-Nachbarschaft zugrunde gelegt hatte und umgekehrt.

Die Kontur einer Region kann, wenn die Region Löcher besitzt, aus mehreren Teilkonturen bestehen. Die Punkte jeder Teilkontur kann man durch *Konturverfolgung* finden, wenn ein Punkt der Teilkontur bekannt ist. Dies wird durch den folgenden Algorithmus realisiert (zur Vereinfachung nehmen wir hier an, daß jede Region aus mindestens 2 Punkten besteht):

Konturverfolgung:

0) gegeben: ein Punkt der Kontur (Startpunkt) sowie eine Richtung, so daß der zu dieser Richtung korrespondierende Nachbar außerhalb der Region liegt
1) Initialisierung: inkrementiere den Richtungscode solange, bis der korrespondierende Nachbarpunkt innerhalb der Region liegt (Startrichtung)
2) Wiederhole, bis Startpunkt und Startrichtung wieder erreicht werden:
   2.1) gehe entlang der aktuellen Richtung zum nächsten Punkt der Kontur und kehre die Richtung um
   2.2) suche den nächsten Punkt der Kontur: inkrementiere den Richtungscode solange, bis der zur neuen Richtung korrespondierende Nachbar innerhalb der Region liegt

Verwenden wir bei der Suche die 4-Nachbarschaft (geradzahlige Richtungscodes), erhalten wir die 4-Kontur, bei Verwendung der 8-Nachbarschaft (sämtliche Richtungscodes) entsprechend die 8-Kontur. Abbildung 9 zeigt ein Beispiel für beide Varianten. Gegeben waren als Startpunkt das obere linke Pixel der Region sowie der Richtungscode 4 (nach links).

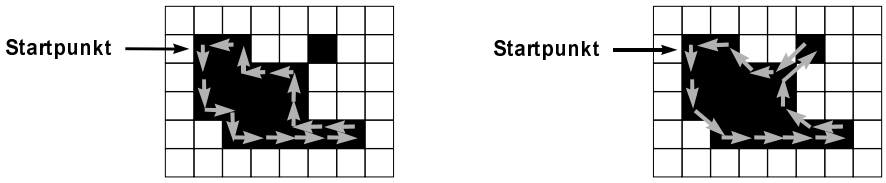


Abbildung 9: Konturen, die wir durch Konturverfolgung von den angegebenen Startpunkten aus erhalten (links: 4-zusammenhängende Region und 4-Kontur, rechts: 8-zusammenhängende Region und 8-Kontur)

Diesen Algorithmus können wir sehr einfach mit Hilfe unseres Nachbarschaftszirkulators implementieren. Wir initialisieren den Nachbarschaftszirkulator so, daß sein Zentrum der erste Punkt der Kontur ist und er auf einen Nachbarn verweist, der nicht zur Region gehört. Zur Bestimmung der Startrichtung bzw. der Richtung zum nächsten Punkt der Kontur inkrementieren (drehen) wir den Nachbarschaftszirkulator so lange, bis der aktuelle Nachbarpunkt wieder ein Punkt der Region ist.

Das Weitergehen zum nächsten Punkt und das Umkehren der Richtung wird durch einen Aufruf der Funktion `jumpToNeighbor()` realisiert. Der `ContourCirculator` kapselt diese Funktionalität wie folgt:

```
template <class NeighborhoodCirculator>
class ContourCirculator
{
    NeighborhoodCirculator current_;

  public:
    // Initialisierung mit einem NeighborhoodCirculator, der den
    // Startpunkt der Kontur markiert
    ContourCirculator(NeighborhoodCirculator start)
    : current_(start)
    {
        // Richtung zum nächsten Konturpunkt suchen (Startrichtung)
        do { ++current_ } while (**current_ != *(current_.center()));
    }

    // Inkrementierung: Finden des nächsten Punkts der Kontur
    ContourCirculator & operator++()
    {
        // zum nächsten Punkt der Kontur springen
        current_.jumpToNeighbor();
        // Richtung zum nunmehr nächsten Konturpunkt suchen
        do { ++current_ } while (**current_ != *(current_.center()));
    }

    // Vergleich
    bool operator==( ContourCirculator const & o) const {
        return current_ == o.current_;
    }

    // Zugriff: zurückgeben des aktuellen NeighborhoodCirculator
    NeighborhoodCirculator operator*() { return current_; }
};
```

Wir können nun schrittweise einen Konturverfolgungsalgorithmus aufbauen. Zuerst suchen wir den ersten Punkt der Kontur derjenigen Region, die durch `regionlabel` markiert ist:

```
IntImage regionimage(w,h);
... // Markieren der Region

int x,y;
for(y=0; y<h; ++y) {
    for(x=0; x<w; ++x){
        if(regionimage(x,y) == regionlabel) break;
    }
    if(x != w) break;
}
```

Als nächstes erzeugen wir einen Bilditerator, der auf der eben gefundenen Koordinate plaziert wird. Damit wir im folgenden keine Rücksicht auf die Bildbegrenzung nehmen müssen (am Rand liegen einige Nachbarpunkte außerhalb des Bildes), kapseln wir den Bilditerator in einem `ConstantOutsideIterator`. Die Verwendung dieses Adapters bietet sich an, weil wir dem Außenbereich ein anderes label als der bearbeiteten Region geben können, so daß die Kontur auch dann richtig weiterverfolgt wird, wenn die Region den Bildrand berührt.

```
typedef ConstantOutsideIterator<IntImage::Iterator> RegionImageIterator;

RegionImageIterator
  first_point(regionimage.upperLeft(), regionimage.lowerRight(), regionlabel-1);

first_point += Diff2D(x,y); // first_point referenziert den ersten Konturpunkt
```

Nun initialisieren wir den Nachbarschaftszirkulator. Da wir den ersten Punkt von links nach rechts gesucht haben, kann der linke Nachbar des ersten Punkts kein Konturpunkt sein. Wir initialisieren den Nachbarschaftszirkulator deshalb mit Richtungscode 4. Außerdem überprüfen wir, ob die Region aus mindestens 2 Punkten besteht, indem wir testen, ob der Zirkulator bei einem vollen Umlauf (8 Schritte) wenigstens einen weiteren Punkt der Region findet:

```
typedef Neighborhood8Circulator<RegionImageIterator> NeighborCirculator;

NeighborCirculator neighbors(first_point, 4);

// enthält die Region mindestens einen weiteren Punkt ?
bool secondPointFound = false;
for(int i=0; i<8; ++i, ++neighbors)
{
     if(**neighbors == regionlabel) secondPointFound = true;
}
precondition(secondPointFound);
```

Wenn die Region aus mindestens zwei Punkten besteht, können wir den `NeighborCirculator` verwenden, um einen `ContourCirculator` zu initialisieren:

```
ContourCirculator<NeighborCirculator> contour_begin(neighbors);
```

Wir können diesen Zirkulator nun benutzen, um die verschiedensten Informationen über die Kontur zu gewinnen. Beispielsweise können wir die aufeinanderfolgenden Richtungscodes in einer Liste speichern. Eine solche Liste heißt *Kettencode* der Kontur [FREEMAN61]. Zusammen mit den Koordinaten des Anfangspunktes läßt sich mit Hilfe des Kettencodes die gesamte Kontur rekonstruieren:

```
list<int> chain_code;
ContourCirculator<NeighborCirculator> contour_current(contour_begin);
```

```
do {
    // Speichern des aktuellen Richtungscodes
    chain_code.push_back((*contour_current).directionCode());

    // weiter zum nächsten Konturpunkt
    ++contour_current;
}
while(contour_current != contour_begin);
```

Interessant an diesem Beispiel ist vor allem die Tatsache, daß wir vier Iteratoren ineinander geschachtelt haben: auf der untersten Ebene liegt der ursprüngliche Bilditerator. Er wurde in einen Adapter für Randbehandlung eingebettet. Dieser wiederum ist eingebettet in den Zirkulator zum Auffinden der Nachbarn eines Punktes, und auf der höchsten Ebene finden wir schließlich den Kontur-Zirkulator.

Der Vorteil dieser Vorgehensweise liegt in der klaren Aufgabentrennung: jede Ebene hat eine ganz spezifische Aufgabe. Dadurch können Teilaufgaben sehr leicht ausgetauscht werden. Wir können z.B. zur 4-Kontur übergehen, wenn wir den `Neighborhood8Circulator` durch den entsprechenden Adapter für die 4-Nachbarschaft ersetzen. Keine weiteren Änderungen sind dafür nötig. Diese hohe Flexibilität verdanken wir dem generischen Entwurf mit Iteratoren, die trotz unterschiedlicher Funktionalität einheitliche Schnittstellen anbieten.

5.9 Zusammenfassung des Kapitels

In diesem Kapitel haben wir Iterator-Adapter als eine besonders interessante und leistungsfähige Technik der generischen Programmierung vorgestellt:

- Iterator-Adapter erlauben die Änderung des Navigationsmusters, ohne daß dies für Algorithmen sichtbar wird. Auf diese Weise konnten wir elegante Lösungen für das Randproblem, die Über- und Unterabtastung sowie die Auswahl linearer Untermengen des Bildes entwickeln. Daraus ergaben sich zweckmäßige Implementierungen für wichtige Algorithmen, wie z.B. die separierbare und die rekursive Filterung.
- Insbesondere ist es uns mit dieser Technik erstmalig gelungen, in größerem Umfang Algorithmen in Form von Iteratoren zu implementieren, darunter das Abtasten einer beliebigen geraden Linie, den Zugriff auf die 4- und 8-Nachbarschaften eines Punktes sowie die Konturverfolgung.
- Die Implementation von Algorithmen als Iteratoren (bzw. Iterator-Adapter) führt zu einer besseren Systemzerlegung, weil einerseits der Navigationsteil vom Arbeitsteil eines Algorithmus abgetrennt werden kann und sich andererseits die entstandenen Iteratoren, dank ihrer einheitlichen Schnittstelle, leicht miteinander und mit anderen Bausteinen kombinieren lassen.
- Das Auftreten zyklischer Strukturen erfordert die Einführung neuer generischer Konzepte, der Zirkulatoren, die sich wie Iteratoren verhalten, jedoch am Ende der Sequenz zum Anfang zurückkehren.

Kapitel 6

Generische Implementation grundlegender Segmentierungsverfahren

Die Segmentierung ist allgemein definiert als die Zerlegung eines Bildes in zusammengehörende Regionen und deren Konturen (eine genauere Definition entwickeln wir in Abschnitt 7.3.3). Wir werden in diesem Kapitel generische Algorithmen für einige grundlegende Segmentierungsverfahren entwickeln. Damit wollen wir einerseits belegen, daß die generische Programmierung auch in diesem Kontext angewendet werden kann. Andererseits sollen die Beispiele häufig benutzte Möglichkeiten illustrieren, Segmentierungsergebnisse in Form von Bildern darzustellen. Die in diesem Kapitel verwendeten Binärbilder, Kantenbilder und Regionenbilder werden wir im Abschnitt 7.1 hinsichtlich ihrer topologischen Eigenschaften weiter analysieren.

6.1 Schwellwertbildung

Die Schwellwertbildung ist eines der ältesten Segmentierungsverfahren. Wir wollen es wegen seiner Einfachheit hier an den Anfang stellen, auch wenn es nur bei sehr speziellen Problemstellungen anwendbar ist, nämlich dann, wenn Objekte und Hintergrund durch einen globalen Schwellwert voneinander unterschieden werden können. Ergebnis der Schwellwertbildung ist ein Binärbild, bei dem der Hinter-

grund den Wert 0 und die Objekte den Wert 1 haben. Wir erkennen, daß dies eine einfache Punkttransformation ist. Wir definieren deshalb einen Schwellwertfunktor:

```
template <class ValueType>
struct ThresholdFunctor
{
    ValueType threshold, zero, one;

    // Initialisieren des Schwellwerts
    ThresholdFunctor(ValueType thresh)
    : threshold(thresh) , zero(NumericTraits<ValueType>::zero()),
      one(NumericTraits<ValueType>::one())
    {}

    // Binarisiering mit Hilfe des Schwellwerts
    ValueType operator()(ValueType const & v) const {
        return (v < threshold) ? zero : one;
    }
};

// Anwendung: Schwellwertbildung beim Wert 100
ByteImage src(w,h), dest(w,h);
...
transformImage(srcImageRange(src), destImage(dest), ThresholdFunctor<byte>(100));
```

Damit dieser Funktor angewendet werden kann, muß der Pixeltyp ein Modell des Konzepts LessThan Comparable sein. Die Schwellwertbildung erfordert keine neue Funktion, sondern kann mit einem Funktor als Anwendung von `transformImage()` ausgeführt werden.

6.2 Kantendetektion

Kantendetektoren gehören zu den am häufigsten verwendeten Segmentierungsverfahren. Sie bestimmen die Konturen zwischen verschiedenen Regionen anhand eines Diskontinuitätskriteriums. Das heißt, man interpretiert einen Punkt als Bestandteil der Kontur, wenn sich in seiner Umgebung ein Bildmerkmal (z.B. der Grauwert) sehr stark ändert. Aus der Vielzahl unterschiedlicher Möglichkeiten dies umzusetzen, wollen wir beispielhaft ein Verfahren herausgreifen, das die Nullstellen der zweiten Ableitung eines Bildes auswertet. Im 1-dimensionalen korrespondieren die Nullstellen der zweiten Ableitung zu den Extrema der ersten Ableitung, und näherungsweise gilt dies auch im 2-dimensionalen. Allerdings gibt es in 2D verschiedene Möglichkeiten, eine zweite Ableitung zu berechnen. Bei der Kantendetektion hat sich besonders der *Laplaceoperator* bewährt [MARR82], weil er isotrop (richtungsunabhängig) und einfach realisierbar ist:

$$\nabla^2 f = \frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f \tag{6.1}$$

Nach Anwendung des Laplaceoperators auf das Bild f markiert man alle Nulldurchgänge des Resultats als Konturpunkte. Zusätzlich kann man die Signifikanz der Kontur bewerten, indem man die Größe des Gradientenbetrags an der Position des Nulldurchgangs mißt. Der Gradientenbetrag ist wie folgt definiert:

$$|\nabla f| = \sqrt{\left(\frac{\partial}{\partial x} f\right)^2 + \left(\frac{\partial}{\partial y} f\right)^2} \tag{6.2}$$

Wir wollen die Bilder, die durch Anwendung dieser Operatoren entstehen, als *Laplacebild* bzw. *Gradientenbild* bezeichnen. Der Kantendetektionsalgorithmus gliedert sich damit in folgende Schritte:

Kantendetektion (Nullstellen der zweiten Ableitung):

1) Berechnen von Laplacebild und Gradientenbild
2) Auffinden der Nulldurchgänge des Laplacebildes
3) Optional als Erweiterung von [Marr82]: Markieren derjenigen Nulldurchgänge, bei denen der Gradientenbetrag größer als ein Schwellwert ist

Die Berechnung von Ableitungen eines Bildes kann nur dann sinnvoll definiert werden, wenn man sich dabei auf eine bestimmte Skala bezieht [KOENDER90]. Vor der Bestimmung einer Ableitung muß man das Bild also mit einem Glättungsfilter der gewünschten Skala falten. Die Bestimmung der Ableitung des geglätteten Bildes ist mathematisch äquivalent zur Faltung des Originalbildes mit der Ableitung des Glättungsfilters (L ist ein beliebiger linearer Differentialoperator, g ein Glättungsfilter und f das Originalbild):

$$L(g * f)(x, y) = (Lg * f)(x, y) \tag{6.3}$$

In der Literatur wurden verschiedene Glättungsfilter hinsichtlich ihrer Eignung für die Kantendetektion untersucht (siehe z.B. [CANNY86, SHENCAST92]). Zur Implementation des Laplaceoperators werden am häufigsten das Laplacean-of-Gaussian–Filter (g ist ein Gaußfilter) und das Laplacean-of-Exponential–Filter (g ist ein Exponentialfilter) verwendet. Wir wollen hier die zweite Variante wählen.

Shen und Castan [SHENCAST92] haben das Laplacean-of-Exponential–Filter sehr genau untersucht und gezeigt, daß es Kanten genauer lokalisiert und weniger empfindlich gegenüber Bildrauschen ist als das Laplacean-of-Gaussian–Filter. Da man es mit Hilfe von rekursiven Filtern implementieren kann, ist es zudem extrem schnell. Shen und Castan schlagen außerdem vor, die Berechnungen noch weiter zu beschleunigen und zu vereinfachen, indem der Laplaceoperator durch das Differ-

ence-of-Exponential-Filter approximiert wird. Dieses Filter berechnet die Differenz zwischen dem mit einem Exponentialfilter geglätteten Bild und dem Originalbild.[19]

Nach der Anwendung des Laplaceoperators müssen wir die Nulldurchgänge suchen und markieren. Wir gehen dazu punktweise über das Bild und vergleichen jeden Punkt mit seinen 4-Nachbarn. Wir markieren den aktuellen Punkt als Kantenpunkt, wenn an diesem Punkt der Gradientenbetrag größer ist als der gegebene Schwellwert *und* entweder (a) die zweite Ableitung den Wert Null hat, oder (b) zwischen dem aktuellen Punkt und einem seiner Nachbarn ein Nulldurchgang liegt, wobei der aktuelle Punkt dem Nulldurchgang näher ist als sein Nachbar.

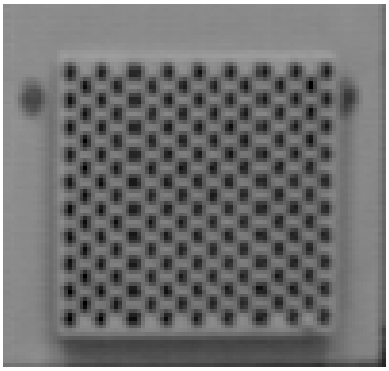
```
template <class SrcIterator, class SrcAccessor,
          class DestIterator, class DestAccessor, class Value1, class Value2>
void differenceOfExponentialEdges(
           SrcIterator src_upperleft, SrcIterator src_lowerright, SrcAccessor sa,
           DestIterator dest_upperleft, DestAccessor da,
           double scale, Value1 gradient_threshold, Value2 edge_marker)
{
    Diff2D size = src_lowerright - src_upperleft;

    // generiere temporäre Bilder für erste und zweite Ableitung
    typedef typename
            NumericTraits<typename SrcAccessor::Value_type>::ScalarPromote
            TmpType;
    BasicImage<TmpType> grad(size.x, size.y), laplace(size.x, size.y);

    // glätte das Bild mit Exponentialfilter
    recursiveSmoothX(srcIterRange(src_upperleft, src_lowerright, sa),
                     destImage(laplace), scale);
    recursiveSmoothY(srcImageRange(laplace), destImage(laplace), scale);

    // bestimme Differenz zwischen geglättetem Bild und Original
    combineTwoImages(srcImageRange(laplace), srcIter(src_upperleft, sa),
                     destImage(laplace), minus<TmpType>());

    // bestimme Gradientenbetrag des Originalbildes
    gradientMagnitude(srcIterRange(src_upperleft, src_lowerright, sa),
                      destImage(grad), scale);

    static Diff2D right(1, 0), down(0, 1), current;
    TmpType zero = NumericTraits<TmpType>::zero();

    // Detektieren und Markieren der Nullstellen
    for(current.y = 0; current.y < size.y; ++current.y)
    {
        for(current.x = 0; current.x < size.x; ++current.x)
        {
            if(grad[current] < gradient_threshold) continue;
```

[19] Diese Approximation wird auch beim Gaußfilter häufig verwendet (Difference-of-Gaussian).

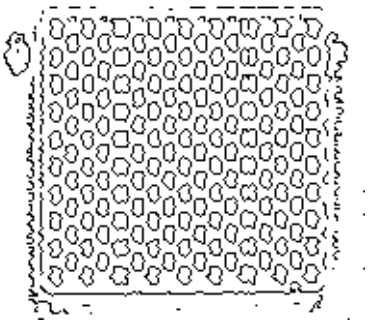
```
                if(laplace[current] == zero)  // Punkt ist exakt Nullstelle
                {
                    da(edge_marker, dest_upperleft, current);
                    continue;
                }

                if(current.x != size.x - 1 &&
                   laplace[current] * laplace[current + right] < zero)
                {
                    // Nullstelle zwischen aktuellem Punkt und rechten Nachbarn,
                    // markiere Punkt mit betragsmäßig kleinerer zweiter Ableitung
                    if(abs(laplace[current]) < abs(laplace[current+right])
                    {
                        da(edge_marker, dest_upperleft, current);
                    }
                    else
                    {
                        da(edge_marker, dest_upperleft, current + right);
                    }
                }

                if(current.y != size.y - 1 &&
                   laplace[current] * laplace[current + down] < zero)
                {
                    // Nullstelle zwischen aktuellem Punkt und unterem Nachbarn,
                    // markiere Punkt mit betragsmäßig kleinerer zweiter Ableitung
                    if(abs(laplace[current]) < abs(laplace[current+down])
                    {
                        da(edge_marker, dest_upperleft, current);
                    }
                    else
                    {
                        da(edge_marker, dest_upperleft, current + down);
                    }
                }
            }
        }
    }
```

Damit der Algorithmus in der angegebenen Form tatsächlich funktioniert, muß man sichern, daß die Nullstellen 1-dimensionale Gebilde sind und keine lokal ausgedehnten Regionen mit dem konstanten Wert Null. Solche Gebiete könnten entstehen, wenn die Bildfunktion ein Gebiet mit konstanten oder linear ansteigenden Grauwerten (Rampe) enthält. In der Praxis haben wir allerdings (bei Verwendung des Difference-of-Exponential-Filters) nie einen solchen Fall beobachtet. Es wäre daher zu prüfen, ob das Difference-of-Exponential-Filter aufgrund seiner mathematischen Eigenschaften stets 1-dimensionale Nulldurchgänge garantiert.

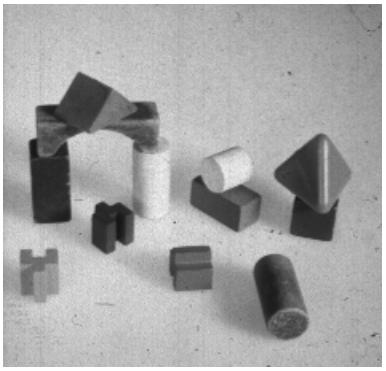
Abbildung 10 zeigt zwei Beispiele für die Anwendung dieses Algorithmus. Die vergleichsweise schlechten Ergebnisse beim zweiten Bild resultieren daraus, daß dieses Bild zahlreiche diffuse Kanten (z.B. Schattenkanten) enthält. Um auch hier befriedigende Kanten zu erhalten, reicht die Anwendung des Operators auf einer Skala (wie bei `differenceOfExponentialEdges()`) nicht aus. Bessere Ergebnisse kann man mit einem Multiskalenverfahren erzielen, siehe z.B. [BEILSTIEHL95, LINDEBERG98]. Natürlich reichen zwei Bilder nicht aus, um einen Algorithmus hinreichend bewerten zu können. Eine umfassende Validierung und Evaluierung eines Segmentierungsalgorithmus ist jedoch sehr aufwendig und kann hier nicht durchgeführt werden (vgl. hierzu [VIERGEVER+99]).



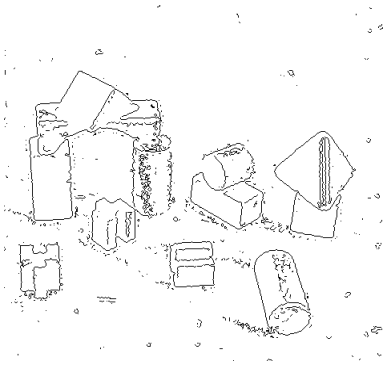
Originalbild (Größe 199x187)



Kantenbild bei Skala 0.8



Originalbild (Größe 473x449)
(Quelle: CVAP, KTH Stockholm)



Kantenbild bei Skala 1.8

Abbildung 10: Kantendetektion mit dem Difference-Of-Exponential-Algorithmus

6.3 Regionenwachstum

Im Gegensatz zu kantenorientierten Segmentierungsverfahren, die Diskontinuitäten detektieren, verwenden regionenorientierte Verfahren ein Homogenitätskriterium. Wir wollen uns hier mit dem sogenannten Seeded Region Growing nach Adams und Bischof [ADAMBISCH94] befassen, bei dem gegebene Startregionen (engl. seed regions) nach und nach vergrößert werden, indem die noch freien Punkte in optimaler Weise an die vorhandenen Regionen angelagert werden. Der Anlagerungsprozeß wird durch eine Kostenfunktion gesteuert, die bewertet, wie gut ein Kandidatenpixel zu einer Region paßt (gute Übereinstimmung verursacht geringe Kosten). Kann ein Kandidat an verschiedene Regionen angelagert werden, wird diejenige Anlagerung mit geringsten Kosten gewählt. Das Verfahren minimiert damit die Summe der Kosten aller Anlagerungen.

Die Liste der Kandidaten verändert sich dynamisch. Zu jeder Zeit sind alle die Punkte Kandidaten, die mindestens einer der bestehenden Regionen direkt benachbart sind, aber noch nicht angelagert wurden. Nach dem Anlagern an eine Region wird der Kandidat aus der Liste entfernt, und seine noch freien (d.h. noch nicht angelagerten) Nachbarpunkte werden in die Liste eingetragen. Ist ein Kandidat zu mehreren Regionen benachbart, ist er auch mehrmals, mit möglicherweise jeweils anderen Kosten, in der Liste enthalten.

Wir können nun die Forderung nach globaler Kostenminimierung erfüllen, indem wir Kandidaten, deren Anlagerung geringe Kosten verursacht, zuerst bearbeiten – die Anlagerungen mit hohen Kosten werden nur dann ausgeführt, wenn sich vorher keine bessere Möglichkeit geboten hat. Um schnell auf den Kandidaten mit den geringsten Kosten zugreifen zu können, ist es zweckmäßig, die Kandidatenliste als Prioritätswarteschlange zu organisieren, die nach ansteigenden Kosten sortiert ist. Wir erhalten somit die folgende Algorithmenbeschreibung:

Seeded Region Growing:

1) Initialisiere die Prioritätswarteschlange der Kandidatenpunkte: berechne die Kosten für alle freien Punkte, die 4-Nachbarn einer bestehenden Region sind, und sortiere diese Punkte nach ansteigenden Kosten in die Prioritätswarteschlange ein. Punkte mit gleichen Kosten werden nach der Methode "First In - First Out" behandelt.
2) Solange noch Kandidaten in der Prioritätswarteschlange sind:
   2.1) Entferne den Kandidaten mit den geringsten Kosten aus der Warteschlange

2.2) Wenn dieser Kandidat immer noch frei ist: (diese Abfrage ist notwendig, weil ein Kandidat mehrmals in der Warteschlange enthalten sein kann)

2.2.1) Lagere ihn an die Nachbarregion an, für die minimale Kosten entstehen

2.2.2) Suche die freien 4-Nachbarn des soeben angelagerten Punktes, berechne ihre Kosten und ordne sie in die Prioritätswarteschlange ein

Wir setzen bei der Implementation dieses Algorithmus voraus, daß jede Keimregion durch eine eindeutige positive ganze Zahl (engl. label) identifiziert wird. Die zu den Keimregionen gehörenden Punkte sind in einem Bild durch die jeweilige Zahl repräsentiert, die freien Punkte hingegen sollen die Markierung Null erhalten. Nach Abschluß des Algorithmus sind alle Punkte mit der Zahl markiert, die ihre Region kennzeichnet. Wir wollen ein solches Bild, bei dem der Wert jedes Punktes die Zugehörigkeit zu einer Region anzeigt, ein *Regionenbild* nennen. Der Algorithmus erhält als Argument ein Regionenbild, bei dem alle freien Punkte den Wert Null haben, und transformiert es in ein Regionenbild, wo diese Punkte einer der Keimregionen zugeordnet sind.

Das Verhalten des vorliegenden Algorithmus kann durch unterschiedliche Wahl der Kostenfunktion in weiten Grenzen variiert werden. Es ist deshalb wichtig, daß bei der Implementation des Algorithmus die Kostenfunktion leicht austauschbar bleibt. Wir werden sie deshalb als Funktor (unter dem Namen `cost`) implementieren. Ein zweiter Funktor `merge` übernimmt das Anlagern eines Punktes an eine Region. Wenn nötig aktualisiert `merge` außerdem nach jeder Anlagerung die Regionenmerkmale, die in spätere Kostenberechnungen einfließen (z.B. kann sich der mittlere Grauwert einer Region mit jeder Anlagerung ändern). Wir erhalten folgende Implementation:

```
template <class ImageIterator, class CostFunctor, class MergeFunctor>
void
seededRegionGrowing(ImageIterator regionimage_ul, ImageIterator regionimage_lr,
                    CostFunctor cost, MergeFunctor merge)
{
    Diff2D size = regionimage_lr - regionimage_ul;

    // Einbetten des Iterators in einen ConstOutsideIterator, der sich um die
    // Randbehandlung kümmert:
    typedef ConstOutsideIterator< Iterator > CheckedImageIterator;
    CheckedImageIterator checked_ul(regionimage_ul, regionimage_lr, 0);

    // die Klasse CandidatePoint wird im Anschluß erklärt
    typedef CandidatePoint<typename CostFunctor::value_type> Candidate;

    // Erzeugen der Kandidatenliste
    // (priority_queue, vector und greater sind C++ Standardklassen)
    std::priority_queue<Candidate, std::vector<Candidate>,
                                   std::greater<Candidate> > candidateList;
```

```
        int count = 0;  // Zähler für die Kandidaten

        // Schleife über alle Punkte
        Diff2D current;
        for(current.y = 0; current.y < size.y; ++current.y)
        {
            for(current.x = 0; current.x < size.x; ++current.x)
            {
                // ist der aktuelle Punkt Bestandteil einer Keimregion ?
                if(regionimage_lr[current] != 0)
                {
                    // ja -> suche in der Nachbarschaft nach freien Punkten
                    // (die Funktion wird im Anschluß erklärt)
                    findCandidatesInNeighborhood(checked_ul + current, current,
                                                 candidateList, cost, count);
                }
            }
        }

        // solange Kandidaten in candidateList sind
        while(!candidateList.empty())
        {
            // Koordinate und gewünschtes Label des Kandidaten erfragen
            Diff2D location = candidateList.top().location;
            int label = candidateList.top().label;

            // Kandidaten aus der Liste entfernen
            candidateList.pop();

            // Iterator erzeugen, der anzulagernden Kandidaten markiert
            ImageIterator candidatePoint = regionimage_ul + location;

            // Kandidaten der Region zuordnen und evtl. statistische
            // Informationen der Region aktualisieren
            bool merge_succeeded = merge(candidatePoint, location, label);

            // wenn die Zuordnung erfolgreich durchgeführt wurde
            if(merge_succeeded)
            {
                // neue Kandidaten in der Nachbarschaft suchen
                findCandidatesInNeighborhood(checked_ul + location, location,
                                             candidateList, cost, count);
            }
        }
    }
```

In der Prioritätswarteschlange müssen wir nicht nur die Koordinaten und die Kosten für jeden Kandidaten speichern, sondern auch, auf welche Region sich diese Kosten beziehen, damit wir auch dann die korrekte Verschmelzung ausführen, wenn der Kandidat mehreren Regionen benachbart ist. Aus technischen Gründen

muß außerdem ein Zähler gespeichert werden, der festhält, in welcher Reihenfolge die Kandidaten in die Warteschlange eingefügt wurden. Kandidaten mit gleichen Kosten werden stets in dieser Reihenfolge bearbeitet. Andernfalls würde die verwendete priority_queue die geforderte "First in - first out" Eigenschaft nicht realisieren:

```
template < class Cost>
struct CandidatePoint
{
    Diff2D location;     // Koordinaten des Kandidaten
    Cost cost_;          // Kosten der Anlagerung
    int label;           // Label der Region, auf die sich die Kosten beziehen
    int count_;          // Zähler für First in - fist out bei gleichen Kosten

    CandidatePoint(Diff2D loc, int l, Cost cost, int count)
    : location(loc), label(l), cost_(cost), count_(count)
    {}

    // Vergleich der Kosten (vergleich der Zählervariablen bei gleichen Kosten)
    bool operator<( CandidatePoint const & o) const
    {
        return (cost_ == o.cost_) ? count_ < o.count_ : cost_ < o.cost_;
    }
};
```

In der Funktion findCandidatesInNeighborhood() prüfen wir, welche 4-Nachbarn des übergebenen Punktes noch frei sind und erzeugen gegebenenfalls einen neuen Eintrag in der Kandidatenliste. Wir benutzen den Neighborhood4Circulator (Abschnitt 5.7), um die Nachbarpunkte zu finden:

```
template <class CheckedImageIterator, class CandidateList, class CostFunctor>
void
findCandidatesInNeighborhood(CheckedImageIterator center, Diff2D location,
           CandidateList & candidateList, CostFunctor const & cost, int & count)
{
    typedef CandidatePoint<typename CostFunctor::value_type> Candidate;

    int label = *center;

    // Einbetten des aktuellen Punkts in Nachbarschafts-Adapter
    Neighborhood4Circulator< CheckedImageIterator > neighbors(center);

    // Schleife über alle Nachbarn
    for(int i=0; i < 4; ++i, ++neighbors)
    {
        if((*neighbors).isOutside()) continue; // außerhalb der Bildregion
        if(**neighbors != 0)         continue; // Nachbar nicht mehr frei

        Diff2D new_location = location + neighbors.differenceVector();
```

```
            // Kosten berechnen, Candidate erzeugen und in candidateList einfügen
            Candidate new_candidate(new_location, label,
                                    cost(new_location, label), ++count);
            candidateList.push(new_candidate);
        }
    }
```

Der Vorteil unserer generischen Implementation besteht darin, daß keine Annahmen über die Berechnung der Kosten gemacht werden – dies ist vollkommen in den Funktoren `cost` und `merge` gekapselt. Wir haben deshalb die Möglichkeit, durch geeignete Wahl dieser Funktoren verschiedene Wachstumsstrategien zu implementieren. Wir wollen als Beispiele das *Wasserscheidenverfahren* sowie ein *Regionenwachstum aufgrund statistischer Merkmale* realisieren.

6.3.1 Das Wasserscheidenverfahren

Das Wasserscheidenverfahren [VINCSOILLE91] hat seinen Ursprung im geographischen Begriff der Wasserscheide, der die Grenzlinie zwischen den Einflußbereichen verschiedener Wassersysteme bezeichnet. Wasser, das auf einer Seite der Wasserscheide niedergeht, fließt zum tiefsten Punkt des einen Einzugsbereichs (Auffangbecken), Wasser auf der anderen Seite zum tiefsten Punkt eines anderen Einzugsbereichs. Wasserscheiden separieren also die Einzugsbereiche der Auffangbecken. Wasser, das direkt auf einer Wasserscheide niedergeht, kann in beide benachbarten Auffangbecken abfließen. Die Bedeutung der Wasserscheiden für das Problem der Bildsegmentierung ergibt sich daraus, daß man ein Diskontinuitätsmaß, wie zum Beispiel den Gradientenbetrag des Bildes, als Gebirge auffassen kann, dessen Einzugsbereiche als Regionen hoher Homogenität (geringer Diskontinuität) und dessen Wasserscheiden als Konturen dieser Regionen interpretiert werden können.

Vincent und Soille haben gezeigt, daß man Wasserscheiden effizient mit dem sogenannten Flutungsverfahren finden kann [VINCSOILLE91]. Dieses Verfahren beginnt bei den lokalen Minima bzw. minimalen Plateaus[20] jedes Einzugsbereichs und „füllt" das Gebirge von dort aus mit Wasser. Trifft in einem Punkt Wasser aus verschiedenen Einzugsbereichen zusammen, wird der betreffende Punkt als Wasserscheidenpunkt markiert. Etwas vereinfacht dargestellt, geht der Algorithmus von Vincent und Soille wie folgt vor:

[20] Minimale Plateaus sind zusammenhängende Gebiete gleichen Grauwerts, die niedriger liegen als ihre Umgebung, d.h., alle Nachbarpunkte des Gebiets haben einen höheren Grauwert.

Wasserscheidenalgorithmus:

1) Zunächst werden die Punkte des Gradientenbildes nach aufsteigenden Werten sortiert. Dabei wird vorausgesetzt, daß die Gradientenbeträge auf ganze Zahlen quantisiert sind und im Bereich h_{min} bis h_{max} liegen.
2) Man detektiert nun einzelne Punkte und zusammenhängende Regionen, die den global minimalen Gradientenbetrag h_{min} besitzen. Dies sind die initialen Auffangbecken.
3) Für jeden Gradientenwert von h_{min}+1 bis h_{max}:

   Punkte bzw. Regionen, die den aktuellen Gradientenwert haben, werden an das jeweils benachbarte Auffangbecken angelagert. Grenzen Punkte an zwei verschiedene Einzugsbereiche an, markiert man sie als Wasserscheiden. Punkte und Regionen, die den aktuellen Gradientenwert haben, aber keinem der vorhandenen Auffangbecken benachbart sind, bilden neue, zusätzliche Auffangbecken. Dieser Vorgang wird für alle Werte von h_{min}+1 bis h_{max} wiederholt., bis sämtliche Punkte bearbeitet sind.

Die Anlagerung der Punkte wird mit Hilfe einer "First in - first out"-Datenstruktur gesteuert, in die jeder Punkt mit dem aktuellen Gradientenwert eingetragen wird, sobald er Nachbar eines der Auffangbecken geworden ist. Das Flutungsverfahren kann deshalb als Spezialfall eines Regionenwachstums betrachtet werden. Daraus ergibt sich die Idee, Seeded Region Growing zur Implementation des Wasserscheidenalgorithmus zu verwenden. Um diese Idee zu verwirklichen, benötigen wir eine Kostenfunktion, die genau die Wachstumsstrategie des Flutungsverfahrens realisiert.

Wir erreichen dies, indem wir die Kosten jedes Punkts mit dem lokalen Gradientenbetrag gleichsetzen. Dadurch haben Punkte mit niedrigem Gradientenbetrag hohe Priorität und werden zuerst bearbeitet. Somit wird das Gradientenbild, wie gefordert, von unten nach oben geflutet. Punkte mit gleichem Gradientenwert verursachen identische Kosten und werden nach der "First in - first out"-Regel behandelt. Auch dies entspricht der Reihenfolge beim Wasserscheidenalgorithmus von Vincent und Soille. Der Kostenfunktor für Seeded Region Growing muß folglich für jeden Kandidaten den zugehörigen Gradientenbetrag zurückgeben. Wir übergeben ihm zu diesem Zweck im Konstruktor einen Iterator auf die linke obere Ecke des Gradientenbildes:

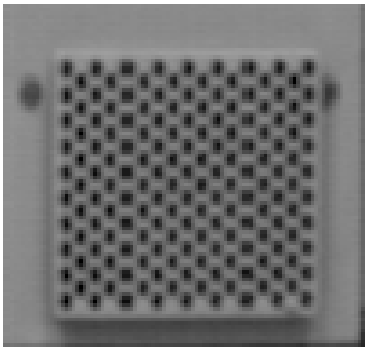
```
template <class ImageIterator>
struct WatershedCostFunctor
{
     typedef typename ImageIterator::value_type value_type;

     ImageIterator gradientimage;  // Iterator auf das Gradientenbild

     WatershedCostFunctor(ImageIterator gul) : gradientimage(gul) {}
```

```
    // Berechne, welche Kosten entstehen würden, wenn der Punkt 'location' mit
    // der Region 'label' vereinigt wird
    value_type operator()(Diff2D const & location, int label) const {
        return gradientimage[location];   // Kosten entsprechen Gradientenbetrag
    }
};
```

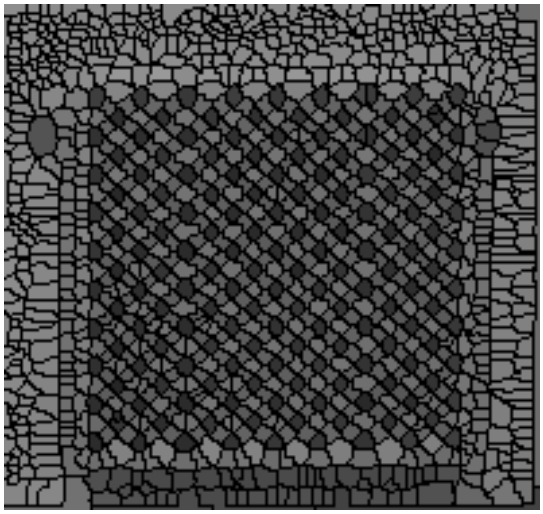
Das Anlagern von Punkten an eine Region wird durch den Funktor merge übernommen. Dies geschieht ganz einfach dadurch, daß dem betreffenden Punkt die Markierung (label) dieser Region zugewiesen wird. Dabei muß allerdings geprüft werden, ob der betreffende Punkt noch frei ist. Wenn nicht, unterscheiden wir zwei Fälle: soll dem Punkt dieselbe Markierung nochmals zugewiesen werden, wird dies einfach ignoriert. Unterscheiden sich hingegen gewünschte und bereits gesetzte Markierung, so handelt es sich um einen Wasserscheidenpunkt. Der Funktor muß in diesem Fall eine spezielle Wasserscheidenmarkierung setzen, die ihm bei der Initialisierung übergeben wird:

```
struct WatershedMergeFunctor
{
    int watershed_label;

    WatershedMergeFunctor(int ws_label) : watershed_label(ws_label) {}

    // Anlagern des Punktes 'current_point' an die Region 'label'
    template <class ImageIterator>
    bool operator()(ImageIterator current_point, Diff2D loc, int label) const {

        if(*current_point == 0) {         // wenn Punkt noch frei
            *current_point = label;       // -> Anlagern
            return true;                  // Anlagern erfolgreich
        }
        if(*current_point != label) {    // zwei verschiedene Nachbarregionen
            *current_point = watershed_label; // -> Wasserscheide
        }
        return false;  // Anlagern nicht erfolgreich
    }
};
```

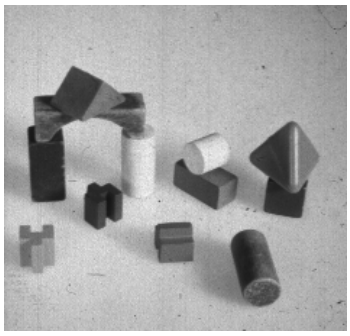
Unsere Realisierung unterscheidet sich in zwei wesentlichen Punkten vom Algorithmus von Vincent und Soille. Erstens müssen bei unserem Verfahren die Auffangbecken (lokale Minima einschließlich minimaler Plateaus) zu Beginn des Algorithmus bekannt sein und als Keimregionen übergeben werden, während Vincent und Soille die Auffangbecken im Verlaufe ihres Algorithmus finden. Zweitens erfolgt das Sortieren in unserem Algorithmus im Verlauf des Wachstums durch die Prioritätswarteschlange, während Vincent und Soille alle Punkte zu Beginn des Verfahrens sortieren.



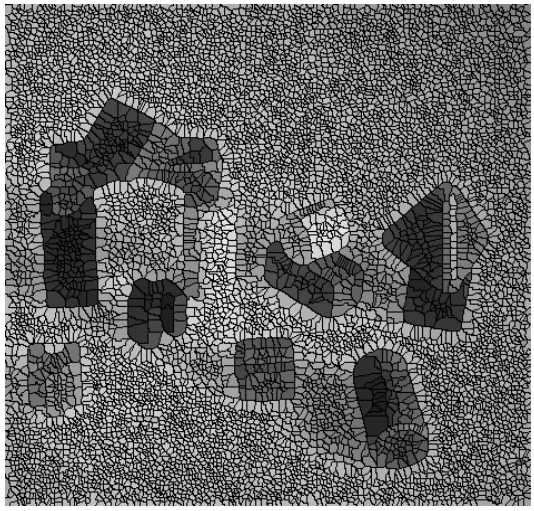
Originalbild (Größe 199x187)



Wasserscheiden des Gradienten bei Skala 1.2



Originalbild (Größe 473x449)



Wasserscheiden des Gradienten bei Skala 2

Abbildung 11: Ergebnisse des Wasserscheidenverfahrens, implementiert als Spezialisierung von Seeded Region Growing

Da wir zum Sortieren die Klasse `priority_queue` des C++-Standards verwenden, können bei unserem Verfahren die Kosten (also die Grauwerte des Gradientenbilds) beliebigen Typ haben, solange dieser Typ ein Modell des Konzepts LessThan Comparable ist. Der Algorithmus von Vincent und Soille ist hingegen auf Werte angewiesen, die auf ganze Zahlen in einem vorher festgelegten Intervall (typisch 0...255) quantisiert sind. Die höhere Flexibilität unseres Verfahrens wird mit einer etwas höheren Laufzeitkomplexität erkauft. Der Algorithmus von Vincent und Soille benötigt lineare Zeit ($O(N)$, wenn N die Anzahl der Punkte ist). Bei Seeded Region Growing ist die Zeit zum Einfügen und Löschen eines Kandidaten in `priority_queue` proportional zum Logarithmus der Größe der Warteschlange. Daraus ergibt sich die Komplexität unseres Algorithmus zu $O(N \log U)$, wenn U die mittlere Zahl von Kandidaten in der Warteschlange ist. Die Zahl U ist proportional zur mittleren Kantenlänge aller Regionen, sie hängt also stark vom jeweiligen Bild ab. Im ungünstigsten Fall gilt $O(U) = O(N)$, unser Algorithmus verhält sich also im ungünstigsten Fall wie $O(N \log N)$ und ist damit ebenfalls relativ schnell. Darüber hinaus können wir eine optimierte Warteschlange verwenden, wenn wir uns auf eine festgelegte Zahl von Gradientenwerten beschränken. Dann wird zum Einfügen und Entfernen eines Elements nur konstante Zeit benötigt, so daß sich die Komplexität des gesamten Algorithmus auf $O(N)$ reduziert. Unter gleichen Voraussetzungen ist Seeded Region Growing demnach ebenso schnell wie der Algorithmus von Vincent und Soille.

Abbildung 11 zeigt zwei Beispiele für die Anwendung des Wasserscheidenverfahrens. Man erkennt die typische Übersegmentierung, die dieser Algorithmus verursacht (vgl. auch [TIECKGERL97]). Auch hier würde die wünschenswerte umfassende Evaluierung des Algorithmus den Rahmen dieser Arbeit sprengen.

6.3.2 Regionenwachstum aufgrund statistischer Merkmale

Als zweite Beispielanwendung für Seeded Region Growing wollen wir ein Regionenwachstum aufgrund statistischer Merkmale beschreiben, wie es in [ADAMBISCH94] vorgestellt wird. Als Kostenfunktion für die Anlagerung des Punktes p an Region i wollen wir den Betrag der Differenz zwischen dem mittleren Grauwert $\overline{f_i}$ der Region und dem Grauwert $f(p)$ des Punktes verwenden:

$$\text{cost}(p,i) = \left| f(p) - \overline{f_i} \right| \tag{6.4}$$

Um dies verwirklichen zu können, benötigen wir zunächst eine Datenstruktur, die den mittleren Grauwert aller Regionen berechnet und im Verlaufe des Wachstums aktualisiert. Wir machen uns dabei zunutze, daß der Mittelwert als Quotient zwischen der Summe aller Grauwerte der Region und der Regionengröße berechnet

wird. Diese beiden Meßwerte kann man sehr leicht inkrementell berechnen. Damit ergibt sich folgende Datenstruktur:

```
template <class ValueType>
struct RegionAverages
{
    std::vector<ValueType> sum_;         // Summe der Grauwerte für jede Region
    std::vector<int>       size_;        // Größe jeder Region

    // Initialisierung für bestimmte Zahl von Regionen
    RegionAverages(int max_region_label)
    : sum_(max_region_label+1, NumericTraits<ValueType>::zero()),
      size_(max_region_label+1, 0)
    {}

    // Aktualisieren der Region 'label' mit Wert 'v'
    void update(ValueType v, int label)
    {
        sum_[label]  += v;
        size_[label] += 1;
    }

    // Berechnen des aktuellen Mittelwerts von Region 'label'
    ValueType average(int label) const
    {
        return sum_[label] / size_[label];
    }
};
```

Die Funktoren `cost` und `merge` benötigen einen Zeiger auf eine solche Struktur, damit sie die Statistiken auswerten und aktualisieren können. Wir erhalten somit die folgenden Implementierungen:

```
template <class ImageIterator, class RegionAverages>
struct AverageCostFunctor
{
    typedef typename ImageIterator::value_type value_type;

    ImageIterator upperleft;  // Iterator auf das Originalbild
    RegionAverages * region_averages;

    AverageCostFunctor(ImageIterator ul, RegionAverages * ra)
    : upperleft(ul),
      region_averages(ra)
    {}

    // Berechne, welche Kosten entstehen würden, wenn der Punkt 'location' mit
    // der Region 'label' vereinigt wird
    value_type operator()(Diff2D const & location, int label) const
    {
        return abs(region_averages->average(label) - upperleft[location]);
    }
};
```

```
template <class ImageIterator, class RegionAverages>
struct AverageMergeFunctor
{
    ImageIterator upperleft;  // Iterator auf das Originalbild
    RegionAverages * region_averages;

    AverageMergeFunctor(ImageIterator ul, RegionAverages * ra)
    : upperleft(ul),
      region_averages(ra)
     {}

    // Anlagern des Punktes 'current_point' an die Region 'label'
    template <class Iterator>
    bool operator()(Iterator current_point, Diff2D location, int label) const
    {
        if(*current_point != 0)  return false;  // Punkt war schon vergeben

        // Statistik aktualisieren
        region_averages->update(upperleft[location], label);

        *current_point = label;       // -> Anlagern
        return true;                  // Anlagern erfolgreich
    }
};
```

Anders als beim Wasserscheidenverfahren markieren wir hier keine Konturen zwischen den Regionen, sondern ordnen jeden Punkt einer Region zu (vollständige Zerlegung des Bildes in Regionen). Das Setzen einer bestimmten Konturmarkierung entfällt deshalb in AverageMergeFunctor. Die Berechnung der Kosten mit Hilfe von Mittelwerten und Differenzen ist natürlich nur eine Möglichkeit, statistische Informationen zu verwerten. Nach dem gleichen Muster können wir Funktoren schreiben, die Statistiken höherer Ordnung auswerten (z.B. die Grauwertvarianz oder Texturmerkmale) oder andere Distanzmaße verwenden, um die Kosten zu bestimmen (z.B. eine Distanz für RGB-Werte). Dies unterstreicht noch einmal die Vielseitigkeit von Seeded Region Growing. Erst durch die generische Programmierung gelingt es, diese Vielseitigkeit mit geringem Aufwand praktisch umzusetzen und nutzbar zu machen.

6.4 Zusammenfassung des Kapitels

In diesem Kapitel haben wir erstmalig generische Funktionen für Segmentierungsverfahren implementiert. Diese Funktionen spezifizieren ihre Anforderungen vollständig durch generische Konzepte und sind dadurch von speziellen Bilddatenstrukturen und Pixeltypen unabhängig. Zudem waren wir in der Lage, Seeded Region Growing soweit zu verallgemeinern, daß wir durch Austausch zweier einfacher Funktoren sehr verschiedene Algorithmen (Wasserscheidenverfahren, Regionenwachstum auf der Basis statistischer Merkmale) implementieren konnten, ohne dafür die Kernfunktion ändern zu müssen.

Kapitel 7

Konzepte für eine universelle Repräsentation von Segmentierungsergebnissen

Wenn wir die bildhaften Repräsentationen der Segmentierungsergebnisse aus dem vorigen Kapitel anschauen, erkennen wir , daß jeder Algorithmus seine Ergebnisse auf andere Weise darstellt: Die Schwellwertoperation produziert ein binäres Bild, in welchem Vorder- und Hintergrund durch unterschiedliche Werte markiert sind. Auch bei der Kantendetektion entsteht ein binäres Bild, das jedoch Objekte und deren Kanten kennzeichnet. Das Regionenwachstum auf der Basis statistischer Merkmale erzeugt ein Regionenbild, das heißt eine vollständige Zerlegung des Bildes in Regionen, wobei jede Region durch eine spezifische ganze Zahl markiert wird. Der Wasserscheidenalgorithmus markiert ebenfalls jede Region durch eine ganze Zahl, aber er kennzeichnet zusätzlich die Kantenpunkte zwischen den Regionen. Diese Aufzählung ließe sich weiter fortsetzen. Etwas überspitzt kann man sagen, daß zur Zeit jeder Segmentierungsalgorithmus eine andere Repräsentation für seine Ergebnisse benutzt.

Eine solche Vielfalt an Repräsentationen ist aus Sicht der Nutzer und Entwickler von Computer Vision-Anwendungen äußerst nachteilig. Sie verhindert nämlich, daß die Bausteine zur Weiterverarbeitung von Segmentierungsergebnissen unabhängig sein können von den Bausteinen, die diese Ergebnisse produzieren. Wenn wir einen Segmentierungsalgorithmus austauschen, werden die nachfolgenden Algorithmen häufig nicht mehr funktionieren, weil sie die neue Repräsentation nicht verarbeiten können. Um das System wieder lauffähig zu machen, müssen wir entweder die

nachfolgenden Bausteine ebenfalls ändern oder zusätzliche Konvertierungsbausteine einfügen. Die erste Möglichkeit erfordert großen Aufwand sowie fundierte Kenntnisse der zu ändernden Algorithmen und widerspricht grundsätzlich unserem Ziel, unabhängige Bausteine zu entwickeln. Auch die zweite führt nicht zu befriedigenden Lösungen, denn jede Konvertierung kostet zusätzlich Zeit und Speicher und verkompliziert die Struktur des Systems.[21] Häufig, und gerade im Kontext der Segmentierung, ist zudem eine Konvertierung gar nicht möglich, weil verschiedene Repräsentationen nicht unbedingt äquivalente Informationen enthalten.

Darüber hinaus wird, wenn jedes Segmentierungsverfahren eine andere Repräsentation verlangt, die Kombination mehrerer Verfahren erschwert. Bei vielen Aufgabenstellungen erwartet man eine Verbesserung der Segmentierung, wenn man die Ergebnisse mehrerer Algorithmen gegeneinander abgleicht oder gemeinsam optimiert. Zwingende Voraussetzung hierfür ist jedoch, daß man die Ergebnisse der verschiedenen Verfahren überhaupt miteinander vergleichen kann. Eine einheitliche Repräsentation für Segmentierungsergebnisse wäre hierbei zumindest eine große Hilfe.

Wir wollen deshalb in diesem Kapitel eine universelle Repräsentation für Segmentierungsergebnisse vorstellen und dafür die notwendigen theoretischen Grundlagen sowie abstrakte generische Schnittstellenkonzepte entwickeln.

7.1 Analyse häufig verwendeter Repräsentationen

Die Frage der Datenrepräsentation wurde im Computer Vision-Kontext von verschiedenen Autoren intensiv diskutiert. Einen Überblick hierüber gibt z.B. [STIEHL90]. Traditionell wird die Computer Vision in drei Ebenen eingeteilt: low-level, intermediate-level, und high-level-Verarbeitung. Jeder dieser Ebenen werden jeweils spezifische Datenrepräsentationen zugeordnet. Ein typisches Beispiel hierfür ist das Visions-System der University of Massachusetts [HANSRISE88]:

Low-level Repräsentationen: Die Daten liegen als digitale Bilder (2-dimensionale Matrizen) vor. Hierzu gehören Sensordaten von Kameras und anderen bildgebenden Geräten sowie Ergebnisse von Algorithmen, soweit sie in bildlicher

[21] Die zusätzliche Komplexität erschwert zunächst die Systementwicklung: um Konvertierungen überhaupt zu ermöglichen, muß man weitgehend redundante Bausteine für die unterschiedlichen Varianten entwickeln und pflegen. Redundante Bausteine hatten wir als wichtige Ursache für das Problem des kartesischen Produkts identifiziert. Aber zur Laufzeit tritt eine weitere Schwierigkeit auf: wenn mehrere Repräsentationen einer Information gleichzeitig existieren, muß deren Konsistenz explizit gesichert werden. Insgesamt setzt der zusätzliche Aufwand bei der Konvertierung von vornherein Grenzen für die Anwendbarkeit dieses Ansatzes.

Form abgespeichert werden, wie z.B. Tiefenbilder, die sich aus der Analyse von Stereo-Bildpaaren ergeben.

Intermediate-level Repräsentationen: Die Daten liegen in Form einer intermediate symbolic representation vor. Eine solche Struktur besteht aus tokens, die grundlegende Bildprimitive wie Linien und Regionen sowie deren Gruppierungen und Zerlegungen in symbolischer Form beschreiben.

High-level Repräsentationen: Die Daten liegen in Form einer wissensbasierten Szenenbeschreibung, z.B. als semantisches Netz, vor.

Ballard und Brown hingegen unterteilen den intermediate-level Bereich weiter und unterscheiden vier Kategorien [BALLBROWN82]:

verallgemeinerte Bilder: Ausgangsdaten der Analyse, die als rechteckiges Gitter dargestellt sind; entspricht den low-level Repräsentationen von [HANSRISE88],

segmentierte Bilder: Ergebnis der Einteilung der Bildelemente in sinnvolle Gruppen, die den abgebildeten Objekten entsprechen,

geometrische Repräsentationen: Beschreibungen der Objektform, die entweder aus segmentierten Bildern oder aus Vorwissen über die abgebildeten Objekte abgeleitet werden,

relationale Modelle: komplexe Repräsentationen für die high-level-Analyse, insbesondere symbolische Repräsentationen wie z.B. semantische Netzwerke; entspricht damit den high-level Repräsentationen von [HANSRISE88].

Eine etwas andere Verfeinerung der grundlegenden Kategorien gibt Marr an [MARR82]. Diese Einteilung ist relativ stark auf Marrs speziellen Ansatz der Bilderkennung zugeschnitten. Auch hier wird das intermediate-level in zwei Kategorien unterteilt:

Images: Intensitätsbilder (Im Gegensatz zu den anderen Autoren bezieht Marr hier keine weiteren Sensormodalitäten ein.),

Primal Sketch: bottom-up Segmentierung zur Bestimmung der Intensitätskanten im Bild einschließlich deren Lage und Beziehungen,

2½-D Sketch: Beschreibungen der Orientierung und des ungefähren Abstands zur Kamera der im Bild dargestellten Oberflächen,

3-D Model: Beschreibung der 3-dimensionalen Form und Lage der dargestellten Objekte mit Hilfe von Volumen- und Oberflächenmodellen.

Wir wollen im folgenden insbesondere Repräsentationen für Segmentierungsergebnisse ausführlicher diskutieren. Da diese Repräsentationen von allen zitierten Autoren einer (bzw. der) Kategorie des intermediate-level zugeordnet werden, werden wir auf low-level und high-level-Repräsentationen nicht näher eingehen. Um

Gemeinsamkeiten und Unterschiede verschiedener Repräsentationen leichter herausarbeiten zu können, wollen wir drei Untergruppen unterscheiden:

Ikonische Repräsentationen: Die Segmentierungsergebnisse werden in Form eines digitalen Bildes (also auf einem 2-dimensionalen Rechteckgitter) dargestellt. Dies entspricht in etwa den „segmentierten Bildern" von Ballard und Brown.

Geometrische Repräsentationen: Jedes Objekt (bzw. jeder Teil eines Objekts) wird durch ein parametrisiertes geometrisches Modell beschrieben, wobei die Parameter aufgrund von Messungen in einem oder mehreren Bildern festgelegt werden. Diese Kategorie entspricht der gleichnamigen Kategorie von Ballard und Brown, soweit diese Autoren geometrische Repräsentationen zur Bildsegmentierung verwenden.

Topologisch-geometrische Repräsentationen: Bei dieser Gruppe steht die Darstellung der topologischen Struktur, also der räumlichen Beziehungen von (einfachen) symbolischen Bildmerkmalen im Vordergrund. Der Zusatz „geometrisch" deutet an, daß den Bildmerkmalen meist auch geometrische Informationen zugeordnet werden. Diese Kategorie verallgemeinert die intermediate symbolic representation von Hanson und Riseman.

7.1.1 Ikonische Repräsentationen

Als ikonisch bezeichnen wir eine Repräsentation, wenn die Segmentierungsergebnisse in einem digitalen Bild, also einem 2-dimensionalen rechteckigen Gitter, dargestellt werden. Beispiele für Algorithmen, die ikonische Repräsentationen liefern, haben wir in Kapitel 6 angegeben. Der Vorteil der ikonischen Darstellung liegt in ihrer Einfachheit: es wird keine neue Datenstruktur benötigt, und die räumlichen Beziehungen der gefundenen Merkmale können direkt aus der Darstellung abgelesen werden.

Leider ist diese Einfachheit nur eine scheinbare. Sobald man sich tiefgründiger mit ikonischen Repräsentationen befaßt, stößt man auf eine ganze Reihe ernster Probleme. Die Tatsache, daß es sehr viele verschiedene Varianten solcher Repräsentationen gibt (Kapitel 6 erwähnte vier Beispiele), ist äußerer Ausdruck dieses Zustands: je nach Aufgabenstellung versucht man eine Variante zu wählen, deren jeweilige Einschränkungen sich möglichst nicht negativ auf das Endergebnis auswirken.

Grundlage der ikonische Repräsentationen ist die von Rosenfeld [ROSENFELD74] eingeführte Interpretation eines Bildes als Graph: die Pixel werden mit den Knoten des Graphen identifiziert, und die Kanten des Graphen kodieren die Nachbarschaft der Pixel. Ein Pixel wird üblicherweise mit seinen horizontalen und vertikalen Nachbarn (4-Nachbarschaft) oder seinen horizontalen, vertikalen und diagonalen

Nachbarn (8-Nachbarschaft) verbunden.[22] Die Schwierigkeiten der ikonischen Repräsentationen erwachsen nun daraus, daß sich aus diesen Definitionen der elementaren Nachbarschaft nicht ohne weiteres widerspruchsfreie Definitionen für den Zusammenhang, die Nachbarschaft und den Rand (die Kanten) von Regionen ableiten lassen, wie wir an einigen Beispielen zeigen wollen.

Die übliche Definition einer zusammenhängenden Region besagt, daß zwei Pixel zur selben Region gehören, wenn sie durch einen Pfad verbunden werden können, der vollständig in der Region liegt. Ein Pfad ist dabei eine Folge von direkt benachbarten Pixeln, wobei entweder 4- oder 8-Nachbarschaft verwendet wird. Diese Definition führt jedoch auf einen Widerspruch, der unter der Bezeichnung „Zusammenhangsparadoxon" bekannt geworden ist. Wir wollen dies anhand der Abbildung 12 verdeutlichen (vergleiche [LATECKI+95]). Angenommen wir interpretieren Abbildung 12 mit der 8-Nachbarschaft. Dann bilden die schwarzen Pixel eine geschlossene Kurve (Jordan-Kurve) und die weißen ein zusammenhängendes Gebiet. Dies steht jedoch im Widerspruch zum Jordanschen Kurventheorem, welches besagt, daß jede geschlossene Kurve die Ebene in genau zwei zusammenhängende Regionen teilt. Mit der 4-Nachbarschaft erhalten wir den umgekehrten Widerspruch: die weißen Pixel zerfallen in zwei getrennte Regionen, obwohl die schwarzen Pixel keine geschlossene Kurve mehr bilden.

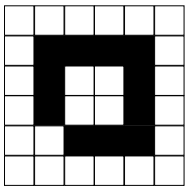


Abbildung 12: Zusammenhangsparadoxon (siehe Text)

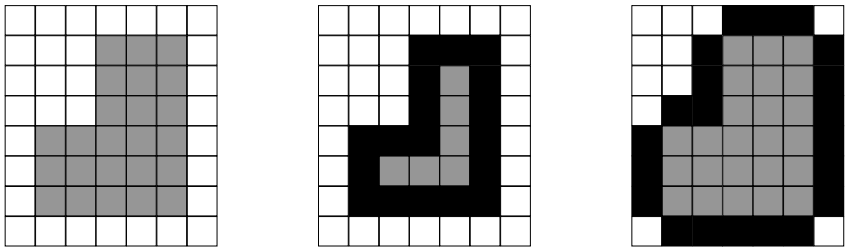


Abbildung 13: Kontur einer Region nach [PAVLIDIS82]: Punkte der Region (links), Innenkontur mit 8-Nachbarschaft (Mitte), Außenkontur mit 4-Nachbarschaft (rechts)

[22] Dies gilt für rechteckige Gitter. Auf hexagonalen Gittern ergibt sich die 6-Nachbarschaft, auf die wir aber nicht eingehen wollen.

Man kann das Paradoxon auf verschiedene Arten beseitigen. Einige Autoren, z.B. [ROSENFELD74], schlagen vor, für schwarze Pixel 8-Nachbarschaft, für weiße hingegen 4-Nachbarschaft zu benutzen. Dies ist jedoch nur bei Binärbildern anwendbar und führt zu einer im allgemeinen nicht akzeptablen Asymmetrie zwischen Vordergrund und Hintergrund. Latecki el al. [LATECKI+95, LATECKI98] definieren eine besondere Klasse von *wohlgeformten Segmentierungen*, die unter 4- und 8-Nachbarschaft identische Zusammenhangskomponenten enthalten. Das heißt, wenn zwei diagonal benachbarte Pixel derselben Region angehören, gehört auch mindestens einer ihrer gemeinsamen 4-Nachbarn zu dieser Region. Somit kann die Konfiguration aus Abbildung 12 nicht auftreten. Obwohl dies das Zusammenhangsparadoxon beseitigt, führt es noch nicht zu einer vollständigen Lösung der Probleme mit ikonischen Repräsentationen, wie die folgende Überlegung zur Definition des Randes einer Region zeigt.

Eine häufig verwendete Definition für die Kontur einer Region betrachtet alle diejenigen Punkte als Konturpunkte, bei denen jede Umgebung mindestens einen Punkt enthält, der zur Region gehört und mindestens einen, der nicht zur Region gehört. Pavlidis [PAVLIDIS82] zeigt, daß man daraus vier verschiedene Definitionen für die Kontur einer Region in einem Bild gewinnen kann, je nachdem ob die Konturpunkte innerhalb oder außerhalb der Region liegen sollen und ob 4- oder 8-Nachbarschaft zugrundegelegt wird.

Abbildung 13 verdeutlicht dies an einem Beispiel (vergleiche [KOVALEVSKY89]). Definieren wir die Kontur als die Menge der Punkte innerhalb der Region, die mindestens einen 8-Nachbarn außerhalb der Region besitzen, erhalten wir die Punkte, die in Abbildung 13 (Mitte) schwarz markiert sind. Diese Konturdefinition wird auch von dem in Abschnitt 5.8 beschriebenen `ContourCirculator` realisiert. Definieren wir hingegen die Kontur als die Menge der Punkte außerhalb der Region, die mindestens einen 4-Nachbarn in der Region besitzen, erhalten wir die in Abbildung 13 rechts markierten Punkte, also eine völlig andere Kontur. Diese Kontur ist zudem nicht 4-zusammenhängend, obwohl wir bei seiner Definition die 4-Nachbarschaft vorausgesetzt haben. Noch problematischer ist allerdings die Tatsache, daß keine dieser Konturdefinitionen symmetrisch ist: bestimmen wir die Konturen der grauen und der weißen Region nach der gleichen Regel, ergeben sich jeweils verschiedene Mengen, obwohl beide Regionen aneinander angrenzen und damit eine gemeinsame Kontur haben müßten. Dies gilt auch dann, wenn die Regionen wohlgeformt im Sinne von Lateckis Definition sind.

Wir können eine eindeutige Grenze zwischen den Regionen finden, wenn wir bestimmte Punkte explizit als Grenzpunkte markieren, die zu keiner Region gehören. Wir haben dies zum Beispiel bei dem Kantendetektor aus Abschnitt 6.2 und beim Wasserscheidenalgorithmus aus Abschnitt 6.3 getan. Zwei Punkte aus verschiedenen Regionen dürfen dann nicht mehr direkt benachbart sein, es muß immer ein Grenzpunkt dazwischen liegen. Grenzpunkte haben also die Eigenschaft, daß mindestens zwei ihrer Nachbarn zu verschiedenen Regionen gehören.

Um bei diesem Ansatz das Zusammenhangsparadoxon zu vermeiden, bietet es sich an, unterschiedliche Nachbarschaften für Regionenpunkte und Grenzpunkte zu verwenden. Dies ist möglich, weil wir ein Kantenbild als Binärbild interpretieren können, in welchem die Kanten den Wert null und die Regionen den Wert eins haben (oder umgekehrt). Der Wasserscheidenalgorithmus aus Abschnitt 6.3 erzeugt beispielsweise 4-zusammenhängende Regionen und 8-zusammenhängende Kanten. Die umgekehrte Möglichkeit wird von Tieck und Gerloff verwendet [TIECKGERL97]. Dies führt zu einer wesentlichen Verbesserung der Situation, aber immer noch zu keiner vollständigen Lösung, wie eine Betrachtung der Kreuzungspunkte von Kanten zeigt.

Kreuzungspunkte sind definiert als diejenigen Kantenpunkte, die an mindestens drei Regionen angrenzen. Wie Abbildung 14 verdeutlicht, gibt es Kantenpunkte, die diese Definition nicht erfüllen, aber die wir aus der Problemstellung heraus trotzdem als Kreuzungspunkte klassifizieren möchten. Die Kanten in Abbildung 14 enthalten keine Kreuzungspunkte, denn keiner der schwarz markierten Kantenpunkte grenzt an mehr als zwei Regionen an. Wir können aber auch keinen Kantenpunkt aus der Konfiguration entfernen, ohne daß sich der Zusammenhang der Kanten und Regionen ändert. Wie [TIECKGERL97] erstmals gezeigt haben, müssen zur korrekten Behandlung solcher und ähnlicher Konfigurationen zahlreiche Sonderfälle betrachtet werden, wobei den Autorinnen allerdings kein Beweis für die Vollständigkeit der von ihnen behandelten Menge von Sonderfällen gelungen ist.

Abbildung 14: Beispiele für Kantenkreuzungen, bei denen kein Kantenpunkt an mehr als zwei Regionen grenzt (links für 4-verbundene, rechts für 8-verbundene Kanten)

Wie sich zeigen wird, ist eine vollständige Lösung des Problems erst möglich, wenn man die Kanten nicht auf den Gitterpunkten, sondern dazwischen markiert, z.B. durch Verwendung der Khalimsky-Ebene ([KHALIMSKY+90], siehe Abschnitt 7.3.1) oder durch Kantenrepräsentationen mit sub-pixel Genauigkeit (siehe z.B. [BEILSTIEHL95]). Dies führt jedoch über die rein ikonische Repräsentation hinaus, da Strukturen unterhalb der Pixelauflösung nicht mehr direkt im Bild dargestellt werden können. Zusammenfassend halten wir fest, daß die zur Zeit üblichen ikonischen Repräsentationen unter einer Reihe von Problemen leiden, die die Weiterverarbeitung sehr erschweren.

7.1.2 Geometrische Repräsentationen

Geometrische Repräsentationen stellen die Merkmale durch geeignet parametrisierte geometrische Prototypen dar, deren Parameter mit Hilfe von im Bild enthaltenen Informationen an konkrete Objektinstanzen angepaßt werden. Im folgenden wollen wir kurz einige verbreitete Varianten angeben.

Punkte werden normalerweise durch ihre Koordinaten identifiziert. Bei geraden Linien gibt man Anfangs- und Endpunkt oder eine Parameterdarstellung der Geradengleichung an. Für gekrümmte Linien werden am häufigsten Kreis-, Ellipsen- und Hyperbelsegmente sowie Spline-Kurven verwendet. Regionen können beispielsweise als Polygone, durch eine oder mehrere gekrümmte Konturen oder durch die Vereinigung einfacher geometrischer Flächen (Dreiecke, Ellipsen) beschrieben werden. In einigen Arbeiten werden sogar komplexe 3-dimensionale Modelle direkt an die Bilddaten angepaßt. Beispielsweise verwenden Koller et al. parametrisierte Drahtgittermodelle zur Erkennung und Beschreibung von Fahrzeugen [KOLLER+93].

Die Anpassung eines Modells an die Bildinformation erfolgt im allgemeinen in der Weise, daß man das Modell in die Bildebene projiziert und danach ein geeignetes Kriterium maximiert, welches die Übereinstimmung des Modells mit der unterliegenden Bildinformation mißt. Bei komplexen Modellen ist hierfür meist ein iteratives Optimierungsverfahren notwendig. Einfache Modelle (Geraden, Kreise, Ellipsen) können mit Hilfe der Methode der kleinsten Quadrate angepaßt oder durch eine Hough-Transformation direkt lokalisiert werden [HARSHAP92]. Die Koordinaten von Punkten können als Schnittpunkte von Geraden, durch parametrisierte Modelle der erwarteten Intensitätsverteilung in der Umgebung der gesuchten Punkte [ROHR92, LUHMANN96] oder direkt mit Hilfe von Eckendetektoren gemessen werden [ROHR94].

Verschiedene Computer Vision-Systeme unterscheiden sich sehr stark darin, welche geometrischen Beschreibungsformen sie anbieten. Khoros 1.0 beispielsweise unterstützt nur die Angabe von Linien durch ihre Endpunkte [KHOROS91]. Tina bietet Punktkoordinaten, gerade Linien und Kurven zweiter Ordnung, jeweils in 2-D und 3-D [POLLARD+97]. Besonders reich an Datenstrukturen für geometrische Merkmalsbeschreibungen ist das Image Understanding Environment, das Klassen für sämtliche oben erwähnten Beschreibungsformen anbietet [IUE98].

Verfahren zur Bestimmung einer geometrischen Repräsentation haben den Vorteil, daß sie sehr genau sind. Im Extremfall kann man die Lage einer Bildstruktur mit weniger als 0.01 Pixel Abweichung bestimmen, wie z.B. [LUHMANN96] berichtet. Außerdem sind geometrische Repräsentationen relativ kompakt, d.h. die Zahl der Parameter ist klein im Vergleich zur Zahl der Bildpunkte. Dies spart nicht nur Speicher, sondern erhöht auch die Robustheit geometrischer Segmentierungsalgorithmen: durch die starke Überbestimmung bei der Ermittlung der Parameter reagieren diese Algorithmen weit weniger empfindlich auf Bildrauschen, Beleuch-

tungseffekte und Verdeckungen als lokale Verfahren wie Kantendetektoren und Regionenwachstum.

Auf der anderen Seite haben rein geometrische Verfahren auch gravierende Nachteile. Erstens ist eine hinreichend selektive Vorauswahl derjenigen Bildpunkte notwendig, deren Werte in die Optimierung eingehen. Verwendet man zu viele Punkte, die nicht zum gesuchten Objekt gehören, erhält man oft falsche oder zumindest ungenaue Lösungen. Bei iterativen Verfahren kommt hinzu, daß die Parameter des Modells bereits hinreichend genau initialisiert werden müssen, damit die Optimierung gegen die gesuchte Lösung konvergiert. Man benötigt deshalb häufig ein zweites Segmentierungsverfahren zur Vorverarbeitung und Initialisierung der Modelle. In vielen Fällen ist die Bandbreite möglicher Objekte außerdem so groß, daß verschiedene Prototypen zur Auswahl stehen müssen. Das Problem, jeweils zu entscheiden, welcher Prototyp an eine gegebene Bildstruktur angepaßt werden soll, konnte bisher nur in Spezialfällen automatisch gelöst werden.

Ein weiterer Nachteil rein geometrischer Repräsentationen ergibt sich daraus, daß jedes Objekt isoliert betrachtet wird. Deshalb sind Informationen zu Nachbarschaft, Überschneidungen oder ähnlichen topologischen Beziehungen nicht direkt verfügbar. Zwar können diese Informationen im Prinzip aus den Parametern der Modelle berechnet werden, dies ist jedoch mit zusätzlichem Aufwand verbunden und erfordert häufig auch den Wechsel in eine andere Repräsentation.

Man verwendet geometrische Beschreibungen deshalb nicht isoliert, sondern verknüpft sie mit einer anderen Repräsentationsform. Beispielsweise verwendet man zusätzlich ikonische Beschreibungen, um die Beziehung zwischen einer geometrischen Repräsentation und den dazu korrespondierenden Strukturen im Bild herzustellen. Topologische Repräsentationsformen, wie wir sie im nächsten Unterabschnitt diskutieren, eignen sich besonders, um die Zusammenhangs- und Nachbarschaftsbeziehungen zwischen geometrischer Objekten und deren Teilen zu beschreiben.

Insgesamt ist festzuhalten, daß die gesuchte universelle Repräsentation von Segmentierungsergebnissen nicht primär auf geometrischen Konzepten aufbauen kann, auch wenn diese zweifellos eine wichtige ergänzende Rolle spielen werden.

7.1.3 Topologisch-geometrische Repräsentationen

Topologische Repräsentationen abstrahieren von den konkreten Eigenschaften der Objekte und stellen in erster Linie deren gegenseitige Beziehungen dar. Sie sind damit allgemeiner als die bisher beschriebenen Repräsentationen, denn sie treffen keine einschränkenden Voraussetzungen über die dargestellten Strukturen, im Gegensatz zu ikonischen Repräsentationen, die eine Gitterstruktur voraussetzen, oder geometrischen, die eine bestimmte geometrische Form darstellen. Da topologi-

sche Strukturen meist durch geometrische Beschreibungen ergänzt werden, wollen wir hier den Begriff „topologisch-geometrische Repräsentationen" verwenden.

Eine verbreitete Datenstruktur ist die sogenannte „Kette von Kantenelementen" (engl. edgel chain), die eine Weiterentwicklung des in Abschnitt 5.8 beschriebenen Kettencodes darstellt. Wir finden eine solche Datenstruktur unter anderem in Tina, IPRS, TargetJr und dem IUE. Ein edgel chain ist eine Folge von abstrakten elementaren Kantenstücken (engl. edgel), die den Verlauf einer digitalen Kurve beschreiben. Jedes edgel hat etwa die Länge der Gitterweite und repräsentiert ein kleines Stück der Kurve. Meist werden den Kantenstücken weitere Parameter zugeordnet, beispielsweise eine lokale Richtung und eine lokale Kantenposition, die oft mit einer Abweichung deutlich unterhalb der Gitterauflösung angegeben werden kann. Die edgel chain wird deshalb häufig im Zusammenhang mit subpixel-genauen Kantendetektoren verwendet.

Repräsentiert man neben den Kanten selbst auch deren Kreuzungen, so gelangt man zum *Kantengraphen*, der den Zusammenhang aller Kanten im Bild beschreibt. Die Kreuzungen sind die Knoten dieses Graphen, und jede (kreuzungsfreie) Kette von Kantenelementen entspricht einer Kante.

Ebenfalls weit verbreitet ist der duale Graph[23] des Kantengraphen, der *Regionennachbarschaftsgraph* [PAVLIDIS77]. Ausgehend von einer Regionensegmentierung, wie sie beispielsweise Seeded Region Growing (Abschnitt 6.3) liefert, wird jede Region als Knoten des Graphen repräsentiert, und eine Kante verbindet zwei Knoten, wenn die zugehörigen Regionen im Bild benachbart sind.

Weder der Kantengraph noch der Regionennachbarschaftsgraph allein reichen jedoch aus, um die topologischen Verhältnisse in einem Bild vollständig zu beschreiben, wie insbesondere [KOVALEVSKY89] klarstellt. Kovalevsky führt das Konzept der Zelltopologien in das Forschungsgebiet Computer Vision ein und zeigt dabei, daß für eine konsistente topologische Beschreibung eines Bildes drei Arten von Primitiven (Punkte, Kanten, Regionen, in diesem Zusammenhang auch als 0-, 1- und 2-Zellen bezeichnet) gleichzeitig betrachtet werden müssen. Diesen Ansatz, der in Abschnitt 7.3 ausführlich behandelt wird, haben sich vor allem TargetJr und das IUE zu eigen gemacht, die Klassen für 0-, 1- und 2-Zellen sowie deren Verknüpfungen anbieten. Auf der Grundlage derselben Überlegungen definieren Fuchs und Förstner [FUCHSFÖRST95] einen Merkmalsgraphen (engl. feature adjacency graph), dessen Knoten Instanzen der Primitive der drei Typen sind und dessen Kanten die Nachbarschaftsrelationen der Primitivinstanzen ausdrücken. Tieck und Gerloff [TIECKGERL97] lösen das Repräsentationsproblem, indem sie gleichzeitig einen Kanten- und einen Regionengraphen verwenden, wodurch sich ebenfalls die

[23] Der duale Graph eines ebenen Graphen ist folgendermaßen definiert: jede Fläche des Originalgraphen wird durch einen Knoten des dualen Graphen repräsentiert. Zwei Knoten des dualen Graphen werden durch eine Kante verbunden, wenn die entsprechenden Flächen des Originalgraphen eine gemeinsame Kante besitzen.

benötigten Nachbarschaftsbeziehungen aller drei Typen von Primitiven ableiten lassen.

Allerdings ist die Benutzung der genannten topologischen Strukturen nicht unproblematisch. Obwohl sie auf einer gemeinsamen mathematischen Idee beruhen, wird diese auf sehr unterschiedliche Weise praktisch umgesetzt. Algorithmen können nicht ohne weiteres von den Zellstrukturen des Image Understanding Environment auf einen Merkmalsgraphen nach Förstner oder auf die Kombination von Kanten- und Regionengraph nach Tieck/Gerloff übertragen werden. Eine Konvertierung zwischen den verschiedenen Varianten erscheint zwar möglich, wäre aber mit erheblichem Aufwand verbunden. Dazu kommt, daß die genannten Datenstrukturen gleichsam nur das Rohmaterial für die Bildung einer topologischen Repräsentation bereitstellen – für die korrekte und konsistente Verknüpfung der einzelnen Strukturen ist der Programmierer selbst verantwortlich. Die Datenstrukturen sind deshalb relativ schwer zu handhaben, was zweifellos dazu beiträgt, daß viele Entwickler vor der Verwendung topologischer Repräsentationen zurückschrecken. Insgesamt fehlen bisher ausgereifte Konzepte, die die mathematischen Ideen der Zelltopologie in einfacher und nutzerfreundlicher Weise in adäquate Software übersetzen.

7.2 Anforderungen an eine universelle Repräsentation von Segmentierungsergebnissen

Idealerweise würde eine universelle Repräsentation von Segmentierungsergebnissen die Vorteile aller oben diskutierten Darstellungsformen vereinen: die mathematische Exaktheit und Flexibilität der topologischen, die Genauigkeit und Robustheit der geometrischen und die Einfachheit und bildhafte Anschaulichkeit der ikonischen Repräsentationen. Wir wollen daher einige grundlegende Anforderungen formulieren, die wir bei der Annäherung an das Ideal erfüllen müssen. In den folgenden Kapiteln werden wir diese Anforderungen im einzelnen analysieren und umsetzen.

Ziel jeder Segmentierung ist die Zusammenfassung von einfachen Bildstrukturen (z.B. Pixeln) zu größeren Einheiten. Die beschriebenen Probleme mit ikonischen Repräsentationen verdeutlichen, daß wir dazu zunächst die Begriffe der Nachbarschaft und des Randes von Bildstrukturen exakt und widerspruchsfrei definieren müssen. Die Bedingungen, unter denen dies möglich ist, werden in der Mathematik durch die Axiome eines topologischen Raumes beschrieben (vgl. Abschnitt 7.3). Wir müssen also fordern:

Korrekte Darstellung topologischer Relationen: Die universelle Repräsentation muß die Axiome eines topologischen Raumes erfüllen, so daß wir den Zusammenhang, die Nachbarschaft und den Rand von Strukturen definieren und beschreiben können.

Wir stellen diese Forderung an den Anfang, weil sie nach [KOVALEVSKY89] eine sehr wichtige Konsequenz hat (vgl. Abschnitt 7.3): zur Bildung eines endlichen topologischen Raumes in zwei Dimensionen sind drei Arten von Primitiven notwendig: Punkte (0-Zellen), Linien (1-Zellen) und Regionen (2-Zellen). Deshalb muß die gesuchte Repräsentation alle drei Arten von Primitiven unterstützen:

Repräsentation von drei Primitivtypen: Die universelle Repräsentation muß in der Lage sein, gleichzeitig Instanzen der drei Primitivtypen Punkt, Linie und Region aufzunehmen und deren Nachbarschaft darzustellen.

Da wir im Verlaufe der Segmentierung die Primitive zu größeren Einheiten zusammenfassen wollen, muß eine universelle Repräsentation dies unterstützen. Es spielt dabei keine Rolle, wie wir im Einzelfall entscheiden, welche Gruppierungen sinnvoll sind – hierfür könnten sich Regionenwachstumsprozesse ebenso eignen wie Kantendetektion, Gruppierung aufgrund von Gestaltmerkmalen [MOHNEVAT92, SARKBOY93] oder semantischen Interpretationen und viele andere Algorithmen. Wir fordern daher:

Möglichkeit der Gruppierung: Die Primitive innerhalb der universellen Repräsentation müssen in unterschiedlicher Weise zu größeren Einheiten zusammengefaßt werden können.

Bis hierher sind die Primitive ebenso wie deren Gruppierungen zu größeren Einheiten nur abstrakte Strukturen, deren Eigenschaften allein durch die Anforderungen der Topologie bestimmt werden. Für viele Anwendungen reichen diese abstrakten Informationen nicht aus, wir benötigen reichhaltigere Beschreibungen. Diese gilt insbesondere auch bei der Bestimmung sinnvoller Gruppierungen der Primitive, bei der Segmentierung und beim Übergang zur höheren Bildanalyse:

Zuordnung problemspezifischer Beschreibungen: Es muß möglich sein, den Primitiven sowie ihren Gruppierungen je nach Problemstellung verschiedene Eigenschaften und unterschiedliche Beschreibungen zuzuordnen.

Die Bandbreite möglicher Beschreibungen ist dabei sehr groß. Sie umfaßt geometrische Eigenschaften (z.B. die Koordinaten eines Punktes oder die Parameter einer geraden Linie) ebenso wie statistische Beschreibungen (z.B. den mittleren Grauwert einer Region) bis hin zu komplexen semantische Interpretationen („Wand eines Gebäudes“). Man beachte, daß diese Beschreibungen die topologischen Relationen ergänzen, sie aber nicht ersetzen, wie es bei rein geometrischen Repräsentationen der Fall wäre.

Um den Anspruch der Universalität einlösen zu können, muß die gesuchte Repräsentation mit pixel-basierten Segmentierungsalgorithmen, die bisher eine einfache ikonische Repräsentation verwendet haben, ebenso zusammenarbeiten können wie mit komplexen, graphbasierten Verfahren. Dies wird sich nur erreichen lassen, wenn die universelle Repräsentation auf viele verschiedene Arten implementiert werden kann. Das heißt:

Abstrakte Schnittstelle: Die universelle Repräsentation muß in Form einer abstrakten Schnittstelle spezifiziert werden, die sich auf vielfältige Weise implementieren läßt.

Diese Anforderungen stellen eine minimale Menge dar, die wir in den folgenden Abschnitten so weit wie möglich in konkreten Datenstrukturen umsetzen wollen.

7.3 Topologische Definition der Segmentierung

Auf der Grundlage der Analyse der Anforderungen und des Stands der Technik in den vorherigen Abschnitten wollen wir nun generische Softwarekonzepte für eine universelle Repräsentation von Segmentierungsergebnissen entwickeln. Zuvor müssen wir jedoch einige theoretische Betrachtungen anstellen. Es hat sich im Zusammenhang mit den ikonischen Repräsentationen gezeigt, daß es sehr kompliziert ist, einer vorhanden Struktur nachträglich topologische Eigenschaften hinzuzufügen. Wir wollen deshalb hier topologische Konzepte an den Anfang stellen. Da die Topologie nur solche Annahmen über die dargestellten Objekte trifft, die für die konsistente Definition von Nachbarschaften absolut notwendig sind, bleibt dennoch großer Spielraum, die Repräsentation je nach Anwendung auf unterschiedliche Weise zu implementieren (z.B. auf Basis eines Gitters oder eines Graphen) sowie durch weitere Merkmale zu ergänzen (z.B. geometrische Eigenschaften der Objekte).

In Rahmen der Untersuchung der gegenseitigen Lage- und Nachbarschaftsbeziehungen von Punkten und Mengen hat die Topologie gezeigt, daß eine konsistente Beschreibung dieser Beziehungen nur möglich ist, wenn die Axiome eines topologischen Raumes erfüllt sind. Ein topologischer Raum beschreibt die Grundeigenschaften von offenen Mengen:

Definition 7.1 (Topologischer Raum): Ein Paar (E, OM) aus einer Menge E von abstrakten Elementen (der *Trägermenge* des topologischen Raumes) und einer Teilmenge OM der Potenzmenge von E (der Menge der *offenen Teilmengen* von E) wird topologischer Raum genannt, wenn es die folgenden Axiome erfüllt:

(1) Die leere Menge $\varnothing$ und E selbst sind offene Mengen: $\varnothing \in OM, E \in OM$.

(2) Die Vereinigung von zwei offenen Mengen ist eine offene Menge:
$O_1, O_2 \in OM \Rightarrow O_1 \cup O_2 \in OM$

(3) Der Durchschnitt von zwei offenen Mengen ist offen:
$O_1, O_2 \in OM \Rightarrow O_1 \cap O_2 \in OM$

Der topologische Raum wird *trennbar* (genauer T_0-trennbar) genannt, wenn er außerdem das folgende Trennungsaxiom erfüllt:

(4) Für zwei beliebige Elemente e_1, e_2 aus E gibt es stets eine offene Menge $U \in OM$ so daß genau eins der beiden Elemente in U liegt.

Wir werden unsere Definition einer Segmentierung und die dazugehörigen Datenstrukturen auf einem topologischen Raum aufbauen, der diesen Axiomen entspricht. Dabei ist zu beachten, daß ein diskretes Bild stets eine endliche Anzahl von Elementen enthält und folglich eine endliche Topologie benötigt wird.

7.3.1 Definition des zellulären Komplexes

Kovalevsky [KOVALEVSKY89] hat gezeigt, daß jeder endliche trennbare topologische Raum die Struktur eines *zellulären Komplexes* besitzt. Ein zellulärer Komplex ist folgendermaßen definiert:

Definition 7.2 (zellulärer Komplex): Ein zellulärer Komplex $C = (Z, B, dim)$ besteht aus einer Menge Z von Zellen, einer Funktion *dim*, die jeder Zelle eine ganzzahlige, nicht-negative Dimension zuordnet, und einer binären Relation $B \subset Z \times Z$ über den Zellen, der sogenannten *Begrenzungsrelation*. Die Begrenzungsrelation darf nur Paare (z_1, z_2) enthalten, bei denen die Dimension der ersten Zelle kleiner als die der zweiten ist: $dim(z_1) < dim(z_2)$. Wir sagen dann „z_1 begrenzt z_2". Außerdem muß die Begrenzungsrelation transitiv sein, d.h. wenn die Paare (z_1, z_2) und (z_2, z_3) in B enthalten sind, so muß auch (z_1, z_3) in B enthalten sein.

Ist n die maximale vorkommende Dimension, nennen wir den Komplex *n-Komplex*. Zellen mit $dim(z) = n$ begrenzen keine andere Zelle.

Enthält die Begrenzungsrelation ein Paar (z_1, z_2), nennen wir z_1 und z_2 *inzident*. Inzidenz ist folglich eine symmetrische Relation, Begrenzung eine asymmetrische. Um zu einem topologischen Raum zu gelangen, identifizieren wir die Menge der Zellen Z mit der Trägermenge E. Darüber hinaus müssen wir die offenen Mengen in einem zellulären Komplex definieren. Dazu benötigen wir zunächst den Begriff des Teilkomplexes:

Definition 7.3 (Teilkomplex): Ein *Teilkomplex* $C' = (Z', B', dim)$ eines zellulären Komplexes C wird durch eine Teilmenge $Z' \subseteq Z$ der Zellen von C gebildet,

wobei die Dimension der Zellen sich nicht ändert und die Begrenzungsrelation B' der Durchschnitt von $Z' \times Z'$ mit B ist.

Anschaulich gesprochen werden einfach einige Elemente aus Z weggelassen und alle Einträge der Begrenzungsrelation gelöscht, die eins der weggelassenen Elemente enthalten. Jeder zelluläre Komplex erfüllt die Axiome eines endlichen trennbaren topologischen Raums, wenn offene Mengen wie folgt definiert werden [KOVALEVSKY89]:

Definition 7.4 (offene Menge): Ein Teilkomplex CO eines zellulären Komplexes C wird *offene Teilmenge* von C genannt, wenn er die folgende Eigenschaft besitzt: gehört eine Zelle z zu CO, so gehören auch alle Zellen zu CO, die in C von z begrenzt werden.

Eine offene Menge, die die Zelle z enthält, wird als *offene Umgebung von z* bezeichnet. Die Umgebung, die nur z und alle direkt von z begrenzten Zellen enthält, ist die *minimale* offene Umgebung von z. Die Gestalt der minimalen offenen Umgebung einer Zelle wird entscheidend von der Dimension dieser Zelle bestimmt. Insbesondere gilt, daß die minimale offene Umgebung einer Zelle mit der höchsten vorkommenden Dimension die Zelle selbst ist, denn eine solche Zelle begrenzt keine anderen Zellen.

Definition 7.5 (abgeschlossene Menge): Das Komplement einer offenen Menge wird als *abgeschlossene Menge* bezeichnet.

Die kleinste abgeschlossene Obermenge einer Menge bezeichnen wir als deren *abgeschlossene Hülle*. Umgekehrt ist die größte offene Untermenge einer Menge deren *offener Kern*. Daraus ergibt sich die übliche Definition des Randes einer Menge:

Definition 7.6 (Rand einer Menge): Die Differenz aus abgeschlossener Hülle und offenem Kern einer Menge wird als *Rand dieser Menge* bezeichnet.

Für die Definition einer Segmentierung benötigen wir außerdem noch die folgenden Begriffe:

Definition 7.7 (Pfad und Zyklus): Ein *Pfad* innerhalb eines zellulären Komplexes ist eine Folge von Zellen mit der Eigenschaft, daß aufeinander folgende Zellen stets inzident sind. Enthält der Pfad nur Zellen der Dimension null und eins (0-und 1-Zellen), bezeichnen wir ihn als *1-dimensionalen Pfad* oder kurz *1-Pfad*. Ein *Zyklus* ist ein 1-Pfad, dessen Anfangs- und Endelement gleich sind.

Definition 7.8 (zusammenhängende Teilmenge): Eine Teilmenge $S \subseteq C^n$ eines n-Komplexes heißt *zusammenhängend*, wenn es für alle Paare von Zellen $z, z' \in S$ in dieser Teilmenge einen Pfad gibt, der die beiden Zellen verbindet und der vollständig in S liegt.

An dieser Stelle erkennen wir den Grund für die topologischen Inkonsistenzen der traditionellen ikonischen Repräsentationen. Das Problem liegt darin, daß dort alle Bildpunkte gleich behandelt werden. Da es folglich nur eine Art von Zellen gibt, muß man ikonische Repräsentationen topologisch als 0-Komplex betrachten. In einem 0-Komplex besitzt jede Zelle die höchstmögliche Dimension (nämlich Dimension Null). Deshalb kann keine Zelle eine andere begrenzen, d.h. es gibt keine inzidenten Zellen, folglich existieren auch keine zusammenhängenden Teilmengen, und der Begriff der Segmentierung als Aufteilung des Bildes in zusammenhängende Regionen verliert seinen Sinn. Wird dennoch – wie traditionell üblich – für alle Pixel eine Nachbarschaft festgelegt (z.B. die 4- oder 8-Nachbarschaft), können widersprüchliche Resultate nicht ausgeschlossen werden, weil diese Festlegungen die Axiome eines topologischen Raumes nicht erfüllen [PTAK+97].

Besonders deutlich wird der Unterschied zwischen beiden Ansätzen bei der Definition der Kontur einer Region (vgl. Abschnitt 7.1 und Abbildung 13): wie Pavlidis gezeigt hatte, kann man im Rahmen des traditionellen Ansatzes je nach Wahl der Definition mindestens vier verschiedene Konturen erhalten. Betrachten wir ein Bild hingegen als 0-Komplex, ergibt sich für den Rand von Teilmengen ein völlig anderes Ergebnis: in einem 0-Komplex begrenzt keine Zelle eine andere. Folglich ist jeder beliebige Teilkomplex eines 0-Komplexes eine offene Menge. Damit ist aber jeder Teilkomplex gleichzeitig abgeschlossen, denn er läßt sich als Komplement eines offenen Teilkomplexes darstellen. Daraus folgt, daß sowohl die abgeschlossene Hülle als auch der offene Kern jeder Menge mit der Menge übereinstimmen. Die Differenz von Hülle und Kern ist stets die leere Menge, d.h. Teilmengen eines 0-Komplexes besitzen keinen Rand. Daher sind 0-Komplexe zwar widerspruchsfrei, aber nicht besonders nützlich für die Bildsegmentierung.

Um sinnvolle Definitionen für Nachbarschaften, zusammenhängende Teilmengen und Konturen angeben zu können, müssen wir zelluläre Komplexe *höherer Dimension* betrachten. Für Computer Vision-Fragestellungen sind insbesondere diejenigen zellulären Komplexe von Interesse, die sich in die (Bild-)Ebene einbetten lassen, weil man dadurch eine Korrespondenz zwischen Bildmerkmalen (Punkten, Kanten und Regionen) und topologischen Primitiven (Zellen) herstellen kann. Bei diesen sogenannten *planaren Zellkomplexen* handelt es sich um spezielle Zellkomplexe der Dimension 2.

Um einen planaren Zellkomplex ohne Überlappungen in die Ebene einzubetten, bildet man jede 0-Zelle eineindeutig auf einen Punkt der Ebene ab. Die 1-Zellen stellt man als (nicht notwendig gerade) Linien dar, die die Punkte ohne Überschneidungen verbinden, und die 2-Zellen identifiziert man mit den maximalen zusammenhängenden Gebieten zwischen den Punkten und Linien. Zusätzlich wird eine 2-Zelle dem Gebiet zugeordnet, das sich bis ins Unendliche erstreckt. Wir bezeichnen sie als *Außenzelle* des Zellkomplexes und den Rand der Außenzelle als *äußeren Rand*.

Ein planarer Zellkomplex läßt sich stets auf unterschiedliche Arten in die Ebene einbetten. Ein Zellkomplex, der bereits in die Ebene eingebettet wurde, heißt *ebener*

Zellkomplex Im Computer Vision-Kontext verfügt man normalerweise über geometrische Zusatzinformationen (z.B. Bildpunktkoordinaten), aus denen sich zwangsläufig eine bestimmte Einbettung in die Ebene ergibt. Daher werden wir im folgenden ausschließlich ebene Zellkomplexe betrachten, d.h. Zellkomplexe, deren Einbettung in die Ebene bereits bekannt ist.

Aus der Definition der Einbettung folgt, daß jede 1-Zelle von zwei 0-Zellen begrenzt wird, die nicht notwendigerweise verschieden sein müssen. Sind Anfangs- und Endzelle identisch, nennen wir die 1-Zelle *Schlinge*. Jede 1-Zelle begrenzt wiederum zwei 2-Zellen, die ebenfalls nicht notwendigerweise verschieden sein müssen. Sind die 2-Zellen verschieden, nennen wir die 1-Zelle *Grenzelement*, andernfalls *Brückenelement*. Jede 0-Zelle kann beliebig viele 1- und 2-Zellen begrenzen, ebenso wie der Rand einer 2-Zelle beliebig viele 1- und 0-Zellen enthalten kann. Wir nennen die Anzahl der 1-Zellen, die von einer 0-Zelle begrenzt werden, den *Grad* dieser 0-Zelle. Eine 0-Zelle vom Grad null wollen wir als *isolierten Punkt* bezeichnen und eine vom Grad eins als *Endpunkt*. Hat die 0-Zelle einen Grad von drei oder größer, sprechen wir von einem *Kreuzungspunkt*. Abbildung 15 zeigt ein Beispiel für einen ebenen Zellkomplex.

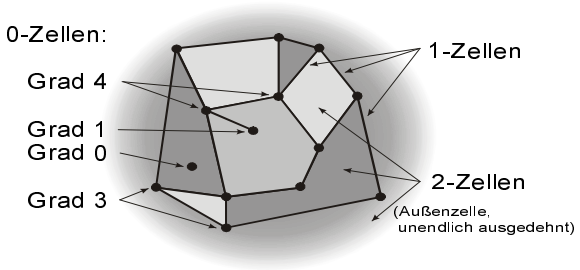


Abbildung 15: Beispiel für einen ebenen Zellkomplex

Durch die Einbettung des Zellkomplexes in die Ebene ergibt sich eine enge Beziehung zu planaren Graphen. Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten, die die Knoten verbinden. Man kann die Relationen „zwei 0-Zellen begrenzen eine 1-Zelle“ eines Zellkomplexes auf die Relation „eine Kante verbindet zwei Knoten“ eines Graphen abbilden und damit die 0-Zellen mit den Knoten und die 1-Zellen mit den Kanten identifizieren. Das heißt, durch Weglassen der 2-Zellen können wir jedem ebenen Zellkomplex einen korrespondierenden ebenen *Kantengraphen* zuordnen.

Da dieser Graph eben ist, besitzt er einen dualen Graphen, den wir als *Flächengraphen* bezeichnen. Jedem Knoten des Flächengraphen entspricht eine 2-Zelle des ursprünglichen Zellkomplexes und jeder Kante eine 1-Zelle. Dabei gilt: begrenzt im Zellkomplex eine 1-Zelle zwei 2-Zellen, verbindet im Flächengraphen die korre-

spondierende Kante die entsprechenden Knoten. Jedes Brückenelement verursacht folglich im Flächengraphen eine Schleife. Abbildung 16 zeigt den Kantengraphen und den Flächengraphen, die zu dem Zellkomplex aus Abbildung 15 korrespondieren.

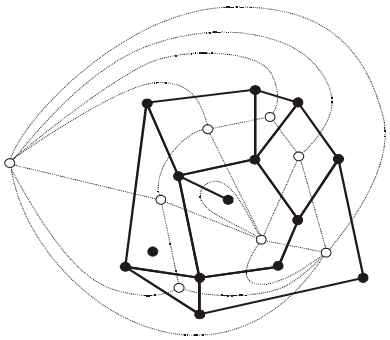


Abbildung 16: zum Zellkomplex aus Abbildung 15 korrespondierende Graphen: Kantengraph (schwarze Knoten und durchgehende Kanten) und Flächengraph (weiße Knoten und gestrichelte Kanten)

Umgekehrt können wir einen ebenen Zellkomplex als ebenen Graphen auffassen, bei dem explizit auch die Flächen zwischen den Knoten und Kanten betrachtet werden. Diese Struktur wird in der Graphentheorie auch als *Landkarte* oder *ebene Karte* (engl. planar map) bezeichnet. Aufgrund der engen Beziehung zur Graphentheorie werden wir die Begriffe Knoten, Kante und Fläche (engl. node, edge, face) im folgenden synonym zu den Begriffen 0-, 1-, bzw. 2-Zelle verwenden.

Im Spezialfall des quadratischen Rasters können wir jedem Pixel eine 2-Zelle zuordnen, und die Außenzelle entspricht dem Gebiet außerhalb des Bildes. 1-Zellen und 0-Zellen werden zwischen den Pixeln bzw. zwischen den Pixeln und der Außenzelle eingefügt, wie es in Abbildung 17 für ein 2×2 Raster gezeigt ist. Mit Ausnahme der 0-Zellen auf dem äußeren Rand haben alle 0-Zellen den Grad vier. Mit Ausnahme der Außenzelle werden alle 2-Zellen von je vier 0- und 1-Zellen begrenzt. Diese Struktur wird von manchen Autoren als Khalimsky-Ebene [KHALIMSKY+90, BRAQDOM96] bzw. als Hyperraster [WINTER95] bezeichnet. Wir halten fest:

Jedem Bild ist eindeutig eine Khalimsky-Ebene zugeordnet, die man durch Einfügen von 0- und 1-Zellen zwischen den Pixeln erhält.

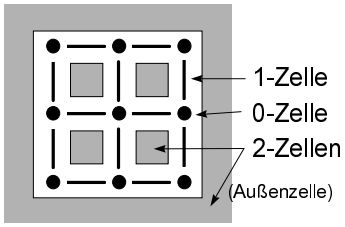


Abbildung 17: Zelltopologie auf dem quadratischen Raster (Khalimsky-Ebene)

Es ist in diesem Zusammenhang interessant zu bemerken, daß der zu einer Khalimsky-Ebene korrespondierende Flächengraph der traditionellen Interpretation der Bildebene nach [ROSENFELD74] entspricht: die Pixel korrespondieren zu den Knoten des Graphen, und diese sind über 4-Nachbarschaft verbunden. Dies reicht jedoch nicht aus, um die komplette topologische Information darzustellen. Dazu wird der vollständige Zellkomplex benötigt, oder man muß Flächengraphen und Kantengraphen kombinieren (vgl. [TIECKGERL97] und [KROPATSCH95]).

Die Einbettung eines Zellkomplexes in die Ebene hat eine sehr wichtige Konsequenz: die zu jeder 0-Zelle inzidenten 1-Zellen haben eine eindeutige Reihenfolge, die sogenannte *Inzidenzordnung*. Bei der Einbettung in die Ebene werden 0- und 1-Zellen auf Punkte bzw. Linien der Ebene abgebildet. Wir gewinnen daraus die Inzidenzordnung durch Analyse der Richtungen, in der die Linien die Punkte erreichen. Messen wir die Winkel, die diese Richtungen mit der x-Achse einschließen, erhalten wir für die Linien, die sich in einem Punkt treffen, eine zyklische Ordnung (Abbildung 18). Wenn wir die Winkel stets in mathematisch positiver Richtung angeben, entspricht die Reihenfolge der 1-Zellen der mathematisch positiven Orientierung, und die Inzidenzordnungen sind an allen 0-Zellen gleich orientiert.

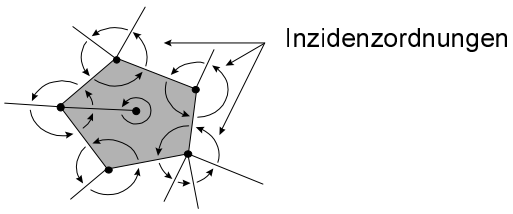


Abbildung 18: Inzidenzordnungen der 1-Zellen in der Umgebung der 0-Zellen

7.3.2 Konturen in zellulären Komplexen

Mit Hilfe der bisherigen Definitionen können wir nun die Konturen einer 2-Zelle formal definieren:

Definition 7.9 (Kontur): Jede Zusammenhangskomponente des Randes einer 2-Zelle wird *Kontur* dieser Zelle genannt.

Der Begriff der Kontur charakterisiert den Rand einer 2-Zelle näher. Der Rand kann aus mehreren Zusammenhangskomponenten bestehen, wenn die 2-Zelle „Löcher" besitzt, d.h. wenn weitere Zellen im Inneren der 2-Zelle liegen. Abbildung 19 zeigt einen Zellkomplex, bei dem der Rand der dunkelgrauen 2-Zelle aus drei Konturen besteht. Jede 2-Zelle hat genau eine *äußere Kontur* und kann beliebig viele *innere Konturen* besitzen.

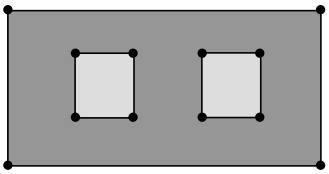


Abbildung 19: Beispiel für einen Zellkomplex, bei dem der Rand einer 2-Zelle (dunkelgrau) aus drei Konturen besteht

Jede Kontur ist ein Zyklus. Wir können uns diese Eigenschaft zunutze machen, um Konturen durch *Konturverfolgung* zu identifizieren. Um eine Konturverfolgung durchführen zu können, müssen wir zunächst eine Reihenfolge für die Zellen der Kontur festlegen. Wir definieren diese Reihenfolge mit Hilfe der Inzidenzordnungen an den 0-Zellen, wie Abbildung 20 verdeutlicht. Wenn die Kante $\mathbf{E_2}$ in der Inzidenzordnung des Knotens $\mathbf{N_1}$ Nachfolger von Kante $\mathbf{E_1}$ ist, so folgen diese Zellen in der Kontur in der umgekehrten Reihenfolge aufeinander: (…,$\mathbf{E_2}$, $\mathbf{N_1}$, $\mathbf{E_1}$,…). Entsprechend ergibt sich die Reihenfolge der Konturelemente am anderen Ende der Kante $\mathbf{E_1}$. Wir können auf diese Weise die Kontur iterativ verfolgen, bis wir wieder an einer bereits bearbeiteten Zelle ankommen. Wir wollen die Reihenfolge, die sich daraus ergibt, als *Konturordnung* bezeichnen.

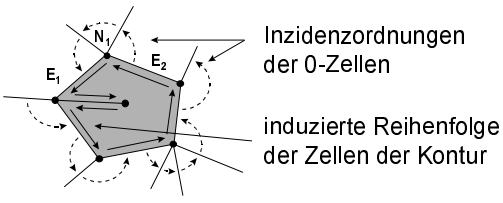


Abbildung 20: Die Inzidenzordnung der 0-Zellen definiert die Konturordnung (d.h. die Reihenfolge der Zellen in der Kontur).

Wir erkennen in Abbildung 20, daß Brückenelemente bei der Konturverfolgung zweimal durchlaufen werden. Grenzelemente kommen in der Konturordnung einer Fläche hingegen genau einmal vor. Bei der Verfolgung der Konturen der benachbarten Flächen werden diese Grenzelemente jeweils in umgekehrter Richtung durchlaufen.

Jede Kontur besitzt eine eindeutige Orientierung, die aus der Orientierung der Inzidenzordnung an den 0-Zellen folgt. Sind die Inzidenzordnungen mathematisch positiv orientiert, ist die äußere Kontur jeder 2-Zelle ebenfalls positiv orientiert, während die inneren Konturen negativ orientiert sind. Die Kontur der Außenzelle, also der äußere Rand der Zellstruktur, ist negativ orientiert. Abbildung 21 zeigt Beispiele für unterschiedlich orientierte Konturen.

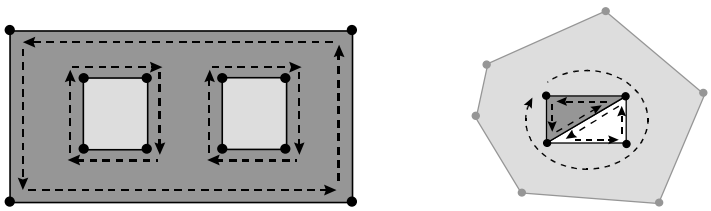


Abbildung 21: Beispiele für die Orientierung von Konturen.
links: Die dunkelgraue Fläche hat eine positiv orientierte äußere und zwei negativ orientierte innere Konturen.
rechts: Die äußeren Konturen der dunkelgrauen bzw. weißen Fläche sind positiv, die innere Kontur der hellgrauen ist negativ orientiert.

Wenn wir die Einbettung des Zellkomplexes in die Ebene kennen, können wir leicht feststellen, ob eine bestimmte Kontur eine äußere oder innere Kontur ist. Wir bestimmen den Wert eines geeigneten Kurvenintegrals über der Kontur, z.B. des Integrals $\frac{1}{2}\oint(x\,dy - y\,dx)$, das den Flächeninhalt der eingeschlossenen Fläche ermittelt und das für Polygone einfach der Summe $\frac{1}{2}\sum_i(x_i\,y_{i+1} - y_i\,x_{i+1})$ entspricht. Der Wert dieses Integrals wechselt sein Vorzeichen, wenn wir die Kontur in umgekehrter Richtung durchlaufen. Wir vergleichen also das Vorzeichen des Integralwerts über einer bestimmten Kontur mit dem Vorzeichen des entsprechenden Integralwerts über dem äußeren Rand des Zellkomplexes. Sind die Vorzeichen gleich oder hat das erste Integral den Wert Null, handelt es sich um eine innere Kontur, andernfalls um eine äußere.

7.3.3 Definition der Segmentierung in zellulären Komplexen

Auf der Basis der zellulären Komplexe können wir nun erstmalig eine formale, widerspruchsfreie Definition der Segmentierung angeben. Wir wollen mit dieser

Definition erreichen, daß Zellen nach bestimmten Regeln zu größeren Einheiten zusammengefaßt werden können. Hierzu definieren wir zunächst drei spezielle Teilmengen, die die Grundlage der Segmentierung bilden sollen:

Definition 7.10 (Region): Eine offene, zusammenhängende Teilmenge eines 2-Komplexes heißt *Region.*

Definition 7.11 (Linie): Ein 1-dimensionaler Pfad heißt *Linie*, wenn alle enthaltenen 0-Zellen den Grad zwei haben und er mit je einer 1-Zelle beginnt und endet. Eine Linie kann folglich keinen Zyklus enthalten.

Definition 7.12 (Vertex): Eine abgeschlossene, zusammenhängende Teilmenge eines 2-Komplexes heißt *Vertex*, wenn sie keine 2-Zellen und keine Zyklen enthält.

Diese Definitionen sind so gewählt, daß wir die Untermengen eindeutig in einzelne 2-, 1- bzw. 0-Zellen transformieren können, wie wir Abschnitt 7.4.2 zeigen werden. Die einfachsten Regionen, Linien bzw. Vertizes sind die 2-, 1- bzw. 0-Zellen selbst. Komplexere Beispiele zeigt Abbildung 22.

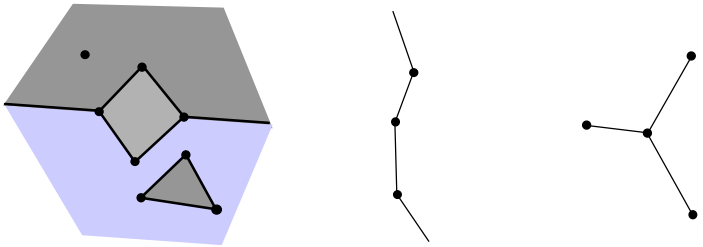


Abbildung 22: Beispiele für Untermengen eines Zellkomplexes: Region (links), Linie (Mitte), Vertex (rechts)

Die klassische Definition einer Bildsegmentierung nach [PAVLIDIS77] fordert die vollständige Zerlegung der Bildes in nicht überlappende Regionen. Diese Definition kann jedoch nicht auf Zellkomplexe übertragen werden: wenn wir nur Regionen zulassen würden, gäbe es nur eine erlaubte „Segmentierung", nämlich die Zusammenfassung des gesamten Zellkomplexes in eine einzige Region. Sobald wir mehrere (nicht überlappende) Regionen bilden, bleiben immer 0- und 1-Zellen übrig, die keiner Region zugewiesen werden können, weil diese dann keine offenen Mengen mehr wären. Deshalb müssen wir bei der Segmentierung von Zellkomplexen alle drei Typen von Untermengen zulassen:

Definition 7.13 (segmentierter Zellkomplex): Ein ebener zellulärer Komplex, der vollständig in Vertizes, Linien und Regionen zerlegt wurde, wird als segmentierter Zellkomplex bezeichnet. Die Region, die die Außenzelle enthält, darf keine weiteren Zellen enthalten.

Jede Zelle gehört somit genau einer Untermenge an, 2-Zellen können jedoch nicht Bestandteil von Vertizes und Linien sein. Anschaulich können wir die Linien als Kanten von Regionen und die Vertizes als deren Ecken interpretieren. Da die Außenzelle eine Region für sich bildet, können Zellen des äußeren Randes des Zellkomplexes nicht in Regionen enthalten sein. Dadurch bleibt der Rand als Grenze zwischen der Außenzelle und dem „Inneren" des Zellkomplexes bei der Segmentierung erhalten. Abbildung 23 zeigt ein Beispiel für eine segmentierte Khalimsky-Ebene.

Unsere Definition eines segmentierten Zellkomplexes ist sehr allgemein. Es ist beispielsweise ohne weiteres möglich, Regionen zu repräsentieren, in deren Innerem weitere Regionen liegen („Löcher"). Ebenso können Kanten dargestellt werden, die keine geschlossenen Kurven sind, wie es bei der Kantendetektion häufig vorkommt. Wir werden im folgenden die generischen Konzepte zur Definition einer einheitlichen Repräsentation für Segmentierungsergebnisse auf dieser Definition aufbauen. Es wird sich zeigen, daß wir bekannte Segmentierungsalgorithmen so modifizieren können, daß ihr Ergebnis der obigen Definition entspricht.

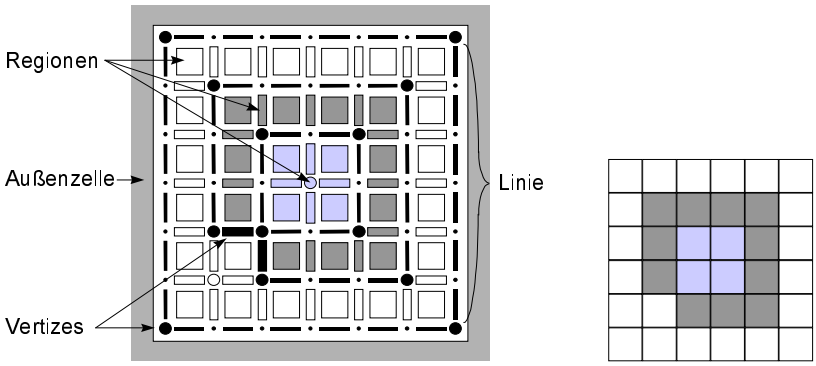


Abbildung 23 links: Beispiel für eine Segmentierung in der Khalimsky-Ebene. Die Segmentierung teilt die Ebene in vier Regionen (weiß, hell-, dunkelgrau, Außenzelle), 12 Linien (dünn: ▬·▬) und 11 Vertizes (fett: ●). Man beachte besonders die Konfiguration ●▬┐ ●, die einen einzelnen, ausgedehnten Vertex (bestehend aus drei 0-Zellen und zwei 1-Zellen) darstellt.
rechts: zum Vergleich das entsprechende Regionenbild. Informationen über Linien und Vertizes können hier nicht dargestellt werden und gehen verloren.

Ein segmentierter Zellkomplex enthält als Teilstrukturen verschiedene der bekannten topologischen Repräsentationen, die wir in Abschnitt 7.1 beschrieben haben. Eine Linie in einem segmentierten Zellkomplex entspricht einer „Kette von Kantenelementen" (edgel chain). Das Analogon eines Kantenbildes erhalten wir, wenn wir die Regionen ignorieren und nur die Linien und Vertizes betrachten. Auch der

Regionennachbarschaftsgraph kann aus einem segmentierten Zellkomplex gewonnen werden. Allerdings können laut Definition eines Zellkomplexes Regionen niemals direkt benachbart sein. Wir müssen deshalb zunächst die Regionennachbarschaft definieren:

Definition 7.14: Zwei Regionen heißen *benachbart* (*adjazent*), wenn es mindestens eine Linie gibt, die jede der beiden Regionen begrenzt, also dazwischen liegt.

Mit Hilfe dieser Definition erhalten wir den Regionennachbarschaftsgraphen, wenn wir jeder Region einen Knoten des Graphen zuordnen und zwei Knoten durch eine Kante verbinden, wenn die betreffenden Regionen benachbart sind. In der Khalimsky-Ebene entspricht diese Definition der 4-Nachbarschaft, denn es kann keine Linie geben, die zwei nur diagonal aneinandergrenzende Regionen begrenzt.

Die Schwierigkeiten, die wir im Zusammenhang mit traditionellen ikonischen Repräsentationen beschrieben haben, treten bei Verwendung von Zellkomplexen nicht auf. Die Definition des Randes einer Region als Differenz zwischen der abgeschlossenen Hülle und dem offenen Kern der Region liefert immer ein eindeutiges, widerspruchsfreies Ergebnis. Auch das Zusammenhangsparadoxon wird gelöst, weil man beweisen kann, daß jede Jordan-Kurve (also jeder Zyklus) den Zellkomplex in genau zwei Teilkomplexe teilt [KOVALEVSKY89].

7.4 Transformationen zwischen Zellkomplexen und die Zellpyramide

7.4.1 Euler-Operatoren

Bisher haben wir Zellkomplexe als statische, unveränderliche Gebilde betrachtet. Häufig reicht dies jedoch nicht aus, zum Beispiel wenn eine fehlerhafte Szenenbeschreibung korrigiert werden soll oder wenn für verschiedene Fragestellungen unterschiedlich detaillierte Beschreibungen benötigt werden (siehe Abschnitt 7.4.3). In solchen Fällen ist es notwendig, verschiedene Zellkomplexe ineinander zu transformieren.

Es zeigt sich nun, daß man alle möglichen Transformationen durch die wiederholte Anwendung von wenigen elementaren Operationen ausdrücken kann. Diese Operationen werden als *Euler-Operatoren* bezeichnet, weil die Eulersche Gleichung (7.1) eine notwendige Bedingung für ihre Korrektheit aufstellt. Euler-Operatoren wurden vor allem in der Computergeometrie intensiv untersucht, da man mit ihrer Hilfe beliebige Polyeder konstruieren und ineinander überführen kann [MÄNTYLÄ88]. Wir wollen diese Überlegungen auf ebene Zellkomplexe übertragen.

Die Eulersche Gleichung (auch als Eulersche Polyederformel bekannt) setzt die Anzahl der Zellen eines Zellkomplexes mit der Anzahl der Zusammenhangskomponenten des korrespondierenden Kantengraphen in Beziehung [WILSON72]. Es gilt:

$$v - e + f = k + 1\,, \tag{7.1}$$

wobei v, e und f die Anzahl der Knoten, Kanten bzw. Flächen des Zellkomplexes und k die Anzahl der Zusammenhangskomponenten des Kantengraphen sind. Diese Gleichung gilt in jedem ebenen Zellkomplex. Folglich ist eine Transformation nur dann zulässig, wenn sie die Anzahl der Primitive und Zusammenhangskomponenten so ändert, daß die Eulersche Gleichung erfüllt bleibt.

Um dies für eine gegebene Transformation zu prüfen, interpretieren wir die Eulersche Gleichung als Gleichung einer Hyperebene in einem 4-dimensionalen Raum (in $\mathbb{Z}^4$) mit den Koordinatenachsen (v, e, f, k):

$$(1,-1,1,-1)(v,e,f,k)^T = 1 \tag{7.2}$$

Der Normalenvektor der Hyperebene lautet (1, -1, 1, -1). Wir schreiben nun die Änderungen, die eine Transformation in den Koordinatenwerten bewirkt, ebenfalls als 4-dimensionalen Vektor (Δv, Δe, Δf, Δk). Wir wollen diesen Vektor als *charakteristischen Vektor* der Transformation bezeichnen. Es muß dann gelten:

$$(1,-1,1,-1)(v+\Delta v, e+\Delta e, f+\Delta f, k+\Delta k)^T = 1 \tag{7.3}$$

Diese Bedingung ist äquivalent damit, daß der Vektor (Δv, Δe, Δf, Δk) innerhalb der Hyperebene liegen muß. Das heißt, das Skalarprodukt dieses Vektors mit dem Normalenvektor muß Null sein. Die Bedingung für erlaubte Transformationen lautet also:

$$(1,-1,1,-1)(\Delta v, \Delta e, \Delta f, \Delta k)^T = \Delta v - \Delta e + \Delta f - \Delta k = 0 \tag{7.4}$$

Da es sich um eine Hyperebene im 4-dimensionalen Raum handelt, kann man alle Vektoren, die Gleichung (7.4) erfüllen, als Linearkombination von drei linear unabhängigen Basisvektoren ausdrücken. Theoretisch reichen also drei geeignet gewählte Transformationen und deren Umkehrungen aus, um sämtliche möglichen Transformationen zu beschreiben. In der Praxis kommt man allerdings nicht mit drei Transformationen aus, da dies zu sehr ineffizienten Algorithmen führen würde.

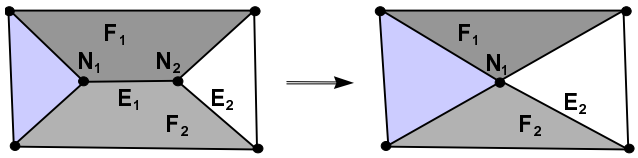
Wir wollen im folgenden fünf elementare Operatoren definieren, die im Zusammenhang mit der Segmentierung in ebenen Zellkomplexen besonders nützlich sind: Verschmelzen benachbarter 0-, 1- bzw. 2-Zellen, Entfernen eines Brückenelements und Entfernen einer isolierten 0-Zelle. Bei der Beschreibung der Transformationen werden wir 0-, 1- und 2-Zellen stets durch die Buchstaben **N**, **E** bzw. **F** (für node, edge, und face) kennzeichnen.

Da es sich bei den fünf grundlegenden Transformationen um lokale Operatoren handelt, müssen wir nur den jeweils betroffenen Ausschnitt des Zellkomplexes betrachten. Wir können deshalb explizit zeigen, wie die Menge der Zellen und die Begrenzungsrelation modifiziert werden müssen, damit ein Zellkomplex wiederum in einen konsistenten Zellkomplex überführt wird. Wir nehmen außerdem an, daß eine Einbettung des ursprünglichen Zellkomplexes in die Ebene vorliegt, aus der sich die Einbettung des transformierten Zellkomplexes ergibt.

Verschmelzen benachbarter 0-Zellen (Kantenkontraktion)

Als benachbart (adjazent) bezeichnen wir zwei 0-Zellen, wenn es eine 1-Zelle gibt, die von jeder der beiden 0-Zellen begrenzt wird. Bei der Kantenkontraktion wird die betreffende 1-Zelle gelöscht und die beiden inzidenten 0-Zellen miteinander verschmolzen. Die Verschmelzung drücken wir so aus, daß wir eine der beiden 0-Zellen löschen und die von ihr begrenzten 1- und 2-Zellen an die überlebende 0-Zelle „anhängen" (Abbildung 24). Es werden also je ein Knoten und eine Kante gelöscht, die Zahl der Flächen und Zusammenhangskomponenten bleibt gleich. Der charakteristische Vektor dieser Operation lautet (-1, -1, 0, 0), Bedingung (7.4) ist damit erfüllt.

Abbildung 24 zeigt, wie die Menge der Zellen und die Begrenzungsrelation modifiziert werden müssen, um zwei 0-Zellen zu verschmelzen. Die neue Inzidenzordnung an der überlebenden 0-Zelle N_1 ergibt sich aus den Inzidenzordnungen vor der Verschmelzung: wenn die gelöschte Kante E_1 in der Inzidenzordnung von N_2 einen Nachfolger E_2 hatte, so nimmt E_2 in der Inzidenzordnung von N_1 den Platz der gelöschten Kante E_1 ein. Die auf E_2 folgenden Kanten werden in ihrer ursprünglichen Reihenfolge in die neue Inzidenzordnung der Zelle N_1 übertragen.



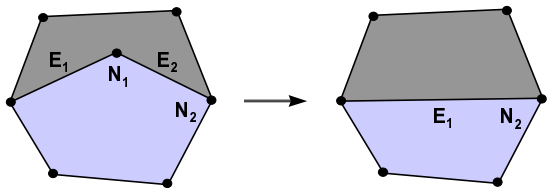
Vorbedingungen	Modifikationen der Zellenmenge	Modifikationen der Begrenzungsrelation
N_1 und N_2 begrenzen E_1	lösche N_2 und E_1	lösche Paare, die E_1 enthalten
E_1 begrenzt F_1 und F_2		lösche die Paare (N_2, F_1) und (N_2, F_2)
		ersetze in den übrigen Paaren N_2 durch N_1

Abbildung 24: Verschmelzen benachbarter 0-Zellen

Verschmelzen benachbarter 1-Zellen

Als benachbart bezeichnen wir zwei 1-Zellen, wenn sie von derselben 0-Zelle begrenzt werden. Die Verschmelzung benachbarter 1-Zellen ist allerdings nur dann möglich, wenn die gemeinsame 0-Zelle den Grad zwei hat. Bei dieser Transformation wird die betreffende 0-Zelle sowie eine der beiden 1-Zellen gelöscht, und die überlebende 1-Zelle ersetzt den vorher existierenden Pfad. Mathematisch ist diese Operation ein Spezialfall der Kantenkontraktion. Wegen ihrer eigenständigen semantischen Bedeutung betrachten wir sie jedoch separat. Wie bei der Kantenkontraktion werden je eine 0- und 1-Zelle gelöscht. Mit dem charakteristischen Vektor (-1, -1, 0, 0) ist auch hier die Bedingung (7.4) erfüllt.

Die notwendigen Modifikationen des Zellkomplexes zeigt Abbildung 25. Da die zu löschende 0-Zelle den Grad zwei hat, ist das Verschmelzen von zwei 1-Zellen etwas einfacher als die Kantenkontraktion. In der Inzidenzordnung der überlebenden 0-Zelle $\mathbf{N_2}$ nimmt $\mathbf{E_1}$ den Platz der gelöschten Kante $\mathbf{E_2}$ ein.



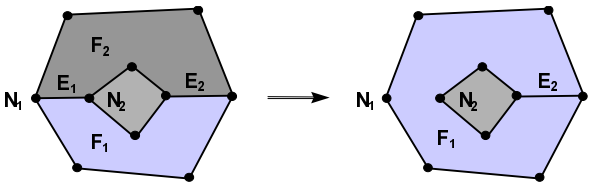
Vorbedingungen	Modifikationen der Zellenmenge	Modifikationen der Begrenzungsrelation
$\mathbf{N_1}$ begrenzt nur $\mathbf{E_1}$ und $\mathbf{E_2}$	lösche $\mathbf{N_1}$ und $\mathbf{E_2}$	Ersetze das Paar ($\mathbf{N_2}$, $\mathbf{E_2}$) durch ($\mathbf{N_2}$, $\mathbf{E_1}$)
$\mathbf{N_2}$ begrenzt $\mathbf{E_2}$		lösche alle Paare, die $\mathbf{N_1}$ oder $\mathbf{E_2}$ enthalten

Abbildung 25: Verschmelzen benachbarter 1-Zellen

Verschmelzen benachbarter 2-Zellen

Zwei 2-Zellen sind benachbart, wenn sie von einer gemeinsamen 1-Zelle begrenzt werden. Beim Verschmelzen dieser 2-Zellen wird die betreffende 1-Zelle sowie eine der beiden 2-Zellen gelöscht, und die Zellen, die die gelöschte 2-Zelle begrenzt haben, begrenzen nun die überlebende 2-Zelle. Wir können die Verschmelzung von 2-Zellen als „Regionenwachstum“ interpretieren. Bei dieser Operation wird die Anzahl der Kanten und Flächen um je Eins verringert, der charakteristische Vektor (0, -1, -1, 0) erfüllt ebenfalls die Bedingung (7.4).

Die notwendigen Modifikationen des Zellkomplexes werden durch Abbildung 26 illustriert. In den Inzidenzordnungen der Endknoten der zu löschenden Kante E_1 wird diese Kante einfach entfernt. Man beachte, daß bei der Verschmelzung von 2-Zellen *nur eine 1-Zelle* gelöscht wird. Eventuell vorhandene weitere Grenzelemente zwischen den beiden 2-Zellen (in unserem Beispiel E_2) bleiben erhalten, verwandeln sich jedoch in Brückenelemente, denn sie begrenzen nur noch eine 2-Zelle.

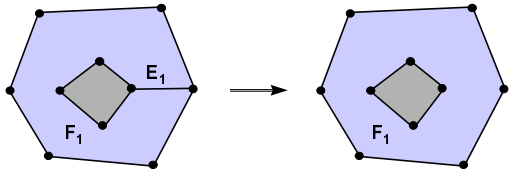


Vorbedingungen	Modifikationen der Zellenmenge	Modifikationen der Begrenzungsrelation
E_1 begrenzt F_1 und F_2	Lösche E_1 und F_2	lösche alle Paare, die E_1 enthalten
N_1 und N_2 begrenzen E_1		lösche die Paare (N_1, F_2) und (N_2, F_2)
		ersetze in den übrigen Paaren F_2 durch F_1

Abbildung 26: Verschmelzen benachbarter 2-Zellen

Entfernen eines Brückenelements

Als Brückenelement bezeichnen wir eine 1-Zelle, wenn sie nur eine 2-Zelle begrenzt. Wie oben beschrieben, entstehen Brückenelemente bei der Verschmelzung von 2-Zellen, wenn die 2-Zellen durch mehrere Grenzelemente verbunden waren. Auch bei der Kantendetektion können Brückenelemente entstehen, z.B. wenn der Kontrast einer Kante allmählich unter die Detektionsgrenze fällt („lose Enden"). Das Entfernen eines Brückenelements ist sehr einfach, weil außer dem Löschen der 1-Zelle keine weiteren Modifikationen des Zellkomplexes nötig sind. Bei der Bestimmung des charakteristischen Vektors ist zu beachten, daß sich die Anzahl der Zusammenhangskomponenten des Kantengraphen um 1 erhöht, wie in Abbildung 27 deutlich wird. Der charakteristische Vektor lautet also (0, -1, 0, 1) und erfüllt Bedingung (7.4). Abbildung 27 zeigt die notwendigen Modifikationen des Zellkomplexes. Die zu entfernende Kante E_1 wird in den Inzidenzordnungen ihrer Endknoten gelöscht.

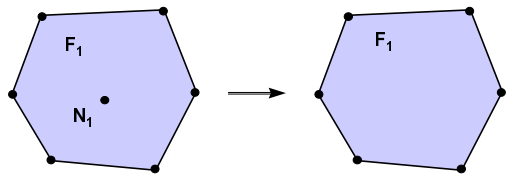


Vorbedingungen	Modifikationen der Zellenmenge	Modifikationen der Begrenzungsrelation
E_1 begrenzt nur F_1	lösche E_1	lösche alle Paare, die E_1 enthalten

Abbildung 27: Entfernen eines Brückenelements

Entfernen einer isolierten 0-Zelle

Als isolierte 0-Zelle bezeichnen wir eine 0-Zelle, wenn sie nur eine 2-Zelle begrenzt, wenn sie also den Grad null hat. Isolierte 0-Zellen entstehen aus Endpunkten (0-Zellen vom Grad eins) durch das Entfernen der inzidenten 1-Zelle (die stets ein Brückenelement ist). Dies wird in Abbildung 28 verdeutlicht. Da hierdurch die Anzahl der Zusammenhangskomponenten des Kantengraphen um 1 sinkt, lautet der charakteristische Vektor (-1, 0, 0, -1). Auch diese Operation erfüllt Bedingung (7.4).



Vorbedingungen	Modifikationen der Zellenmenge	Modifikationen der Begrenzungsrelation
N_1 begrenzt nur F_1	lösche N_1	lösche das Paar (N_1, F_1)

Abbildung 28: Entfernen einer isolierten 0-Zelle

7.4.2 Kontraktion eines segmentierten Zellkomplexes

Mit Hilfe der oben definierten Grundoperatoren wollen wir nun eine spezielle Klasse von Transformationen definieren, die einen segmentierten Zellkomplex in einen neuen, einfacheren Zellkomplex überführen:

Definition 7.15 (Kontraktion eines segmentierten Zellkomplexes): Als Kontraktion eines segmentierten Zellkomplexes bezeichnen wir eine Transformation, die jede Region in eine einzelne 2-Zelle, jede Linie in eine einzelne 1-Zelle und jeden Vertex in eine einzelne 0-Zelle überführt. Den neuen Zellkomplex, der im Ergebnis einer Kontraktion entsteht, bezeichnen wir als *kontrahierten Zellkomplex*.

Die Definition des segmentierten Zellkomplexes wurde absichtlich so gewählt, daß die Kontraktion stets ausgeführt werden kann. Wir zeigen dies im folgenden, indem wir jede Kontraktion auf die wiederholte Anwendung der oben definierten elementaren Euler-Operatoren zurückführen. Daraus ergibt sich einerseits ein Algorithmus zur Implementation der Kontraktion. Andererseits wird dadurch bewiesen, daß ein ebener Zellkomplex durch Segmentierung und anschließende Kontraktion wiederum in einen ebenen Zellkomplex überführt wird, das heißt, daß die Menge der ebenen Zellkomplexe abgeschlossen ist bezüglich der Operationen „Segmentierung + Kontraktion". Dies liefert die Voraussetzung für die Definition von Zellpyramiden im nächsten Abschnitt.

Am einfachsten läßt sich eine Linie in eine 1-Zelle transformieren. Eine Linie hatten wir definiert als 1-dimensionalen Pfad, bei dem jede 0-Zelle den Grad 2 hat und der mit je einer 1-Zelle beginnt und endet. Wir können deshalb auf zwei aufeinanderfolgende 1-Zellen die Operation „Verschmelzen benachbarter 1-Zellen" anwenden und dadurch je eine 0- und 1-Zelle löschen. Es entsteht eine Linie, die um zwei Zellen kürzer ist. Auf diese Linie wenden wir die Operation erneut an und so fort, bis schließlich nur noch eine 1-Zelle übrig bleibt. Dieser Endzustand wird immer erreicht, da die Zahl der 0-Zellen in jeder Linie um eins geringer ist als die Zahl der 1-Zellen. Die überlebende 1-Zelle repräsentiert die ursprüngliche Linie im kontrahierten Zellkomplex.

Die Transformation von Vertizes ist nur wenig komplizierter. Ein Vertex wurde definiert als abgeschlossene Teilmenge eines Zellkomplexes, welche keine Zyklen und keine 2-Zellen enthält. Im graphentheoretischen Sinne ist ein Vertex also ein Baum, dessen Blätter und innere Knoten 0-Zellen sind, die durch 1-Zellen verbunden werden. Wir wählen nun eine 0-Zelle des Vertex, die genau eine zum Vertex gehörende 1-Zelle begrenzt – d.h. ein Blatt des Baumes – und verschmelzen sie mit Hilfe der Operation „Kantenkontraktion" mit der anderen inzidenten 0-Zelle dieser 1-Zelle. Dadurch verringert sich die Zahl der Zellen im Vertex um zwei, aber die Baumeigenschaft bleibt erhalten. Wir wiederholen dies solange, bis nur noch eine 0-Zelle übrigbleibt. Da die Anzahl der Knoten in einem Baum um eins größer ist als die Zahl der Kanten, wird dieser Endzustand stets erreicht. Die überlebende 0-Zelle repräsentiert den Vertex im kontrahierten Zellkomplex.

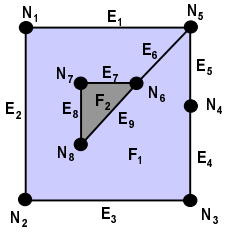
Für die Zuordnung einer 2-Zelle zu einer Region werden die übrigen drei Euler-Operationen benötigt. Wir löschen zunächst alle 1-Zellen, die in der Region enthalten sind. Dies ist stets möglich, weil laut Definition einer offenen Menge die von den

1-Zellen begrenzten 2-Zellen ebenfalls Bestandteil der Region sind. Wir dürfen deshalb die Operation „Verschmelzen benachbarter 2-Zellen" anwenden und zunächst alle Grenzelemente (1-Zellen, die zwei verschiedene 2-Zellen begrenzen) entfernen. Da die Region per Definition zusammenhängend ist, werden dadurch alle 2-Zellen der Region bis auf eine gelöscht. Alle jetzt noch in der Region vorhandenen 1-Zellen sind Brückenelemente, die mit der Operation „Entfernen eines Brückenelements" entfernt werden können. Danach kann die Region, neben der überlebenden 2-Zelle, nur noch isolierte 0-Zellen enthalten, die mit der Operationen „Entfernen einer isolierten 0-Zelle" gelöscht werden. Als einzige Zelle der Region bleibt eine 2-Zelle übrig, welche die Region im kontrahierten Zellkomplex repräsentiert.

Daraus folgt auch, daß unser System von Euler-Operatoren vollständig ist. Das heißt, wir können jede beliebige Transformation zwischen Zellkomplexen durch eine Folge der fünf Euler-Operatoren und deren Rücktransformationen ausdrücken.[24] Wir können nämlich in jedem ebenen Zellkomplex eine Region definieren, die sämtliche Zellen des Zellkomplexes enthält. Mit dem oben angegeben Verfahren können wir jede Region und damit jeden beliebigen ebenen Zellkomplex in eine einzelne 2-Zelle überführen. Durch Anwendung der entsprechenden Rücktransformationen können wir umgekehrt jeden Zellkomplex aus einer einzelnen 2-Zelle erzeugen. Daraus folgt, daß wir über den Umweg einer einzelnen 2-Zelle sämtliche ebenen Zellkomplexe ineinander transformieren können. Natürlich führt dies in der Praxis nicht zu sehr effizienten Algorithmen, aber es beweist die Vollständigkeit des vorgeschlagenen Operatorsystems (genau genommen reichen sogar die drei Operatoren aus, die bei der Transformation von Regionen verwendet werden).

Abbildung 29 zeigt ein Beispiel für die Kontraktion eines segmentierten Zellkomplexes. Ausgangspunkt ist ein (willkürlich gewählter) Zellkomplex, der einem Rechteck entspricht, dessen Inneres in zwei 2-Zellen sowie einige 0- und 1-Zellen aufgespalten ist und dessen rechte Kante in zwei 1-Zellen sowie eine 0-Zelle zerfällt. Diese Ausgangssituation könnte zum Beispiel Folge einer Übersegmentierung im vorhergehenden Verarbeitungsschritt sein. Durch ein geeignetes Segmentierungsverfahren, das den Zellkomplex analysiert, wurde die Übersegmentierung erkannt und das Innere des Rechtecks zu einer Region sowie die rechte Kante zu einer Linie zusammengefaßt. Die Abbildung demonstriert nun, wie der zu dieser Segmentierung korrespondierende kontrahierte Zellkomplex erzeugt wird, indem die zusammengehörenden Zellen sukzessive miteinander verschmolzen werden.

[24] Die Rücktransformationen werden analog zu den Hintransformationen definiert. Da wir jene jedoch im weiteren Verlauf der Arbeit nicht benötigen, wollen wir die Definitionen nicht explizit angeben.



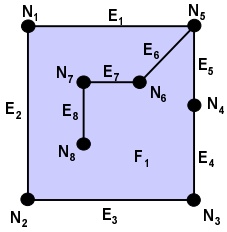
Ausgangssituation:

Zellkomplex mit den Zellen N_1...N_8, E_1..E_9, F_1, F_2, F_∞ (Außenzelle, nicht dargestellt), segmentiert in folgende Regionen, Linien und Vertizes:

Regionen: R_1: 2-Zellen F_1, F_2, 1-Zellen E_6, E_7, E_8, E_9, 0-Zellen N_6, N_7, N_8
R_2: Außenzelle F_∞ (nicht dargestellt)

Linien: L_1: 1-Zelle E_1
L_2: 1-Zelle E_2
L_3: 1-Zelle E_3
L_4: 1-Zellen E_4, E_5 , 0-Zelle N_4

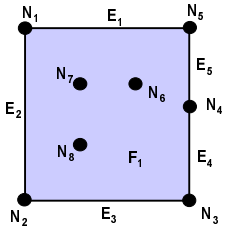
Vertizes: V_1: 0-Zelle N_1, V_2: 0-Zelle N_2, V_3: 0-Zelle N_3, V_4: 0-Zelle N_5



Schritt 1:

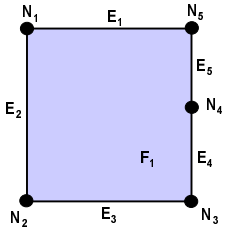
Verschmelzen der 2-Zellen F_1 und F_2 an der 1-Zelle E_9

(F_2 und E_9 werden dadurch gelöscht)



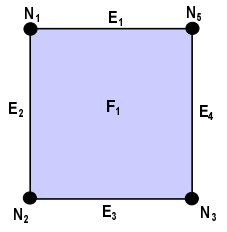
Schritt 2:

Entfernen der Brückenelemente E_6, E_7, E_8



Schritt 3:

Entfernen der nunmehr isolierten 0-Zellen N_6, N_7, N_8



Schritt 4: Verschmelzen der 1-Zellen E_4 und E_5 an der 0-Zelle N_4

(E_5 und N_4 werden dadurch gelöscht)

Endsituation: eindeutige Beziehung zwischen Regionen, Linien und Vertizes der Segmentierung zu den Zellen des kontrahierten Zellkomplexes:

Regionen: $R_1 = F_1$, R_2 = Außenzelle

Linien: $L_1 = E_1$, $L_2 = E_2$, $L_3 = E_3$, $L_4 = E_4$

Vertizes: $V_1 = N_1$, $V_2 = N_2$, $V_3 = N_3$, $V_4 = N_5$

Abbildung 29: Kontraktion einer segmentierten Zellkomplexes

7.4.3 Zellpyramide und Segmentierungshierarchie

Wir haben im vorigen Abschnitt gezeigt, daß man durch die Operationen „Segmentierung“ und „Kontraktion“ einen gegebenen Zellkomplex in einen anderen überführen kann. Wir können diese beiden Operationen deshalb wiederholt anwenden und dadurch eine Folge von immer weiter vereinfachten Zellkomplexen erzeugen. In Analogie zu den bekannten Bildpyramiden wollen wir eine solche Folge als *Zellpyramide* bezeichnen:

Definition 7.16 (Zellpyramide): Eine Folge von Zellkomplexen wird als Zellpyramide bezeichnet, wenn auf jedem Zellkomplex der Folge eine Segmentierung definiert ist, aus der das nächste Glied der Folge durch Kontraktion hervorgeht. Die Glieder der Folge werden auch als Ebenen der Pyramide bezeichnet.

Da die Transformation jeder Ebene in die nächste durch eine Segmentierung bestimmt wird, bezeichnen wir die Zellpyramide auch als *Segmentierungshierarchie*. Am Fuße der Zellpyramide steht eine Khalimsky-Ebene, die zum Originalbild korrespondiert. Die höheren Ebenen beschreiben immer gröbere Bildstrukturen. Den Primitiven auf höheren Ebenen sind eindeutig Gruppen von Primitiven jeder tieferen Ebene zugeordnet, bis hinunter zu einzelnen Pixeln in der Khalimsky-Ebene.

Die Zellpyramide ist damit eine Verallgemeinerung der bekannten Bildpyramiden, wie z.B. der Burt-Pyramide (vgl. Abschnitt 5.3). Bildpyramiden verwenden jedoch eine starre Regel, um zur nächsten Stufe der Pyramide zu gelangen. Bei der Burt-Pyramide werden z.B. stets 2x2 Pixel zu einem neuen Pixel zusammengefaßt. Im Gegensatz dazu kann man die Zellen in einer Zellpyramide beliebig zusammenfassen, solange die Definition eines segmentierten Zellkomplexes erfüllt ist. Durch geeignete Wahl von Segmentierungsverfahren lassen sich die verschiedensten Reduktionsschemen implementieren. Eine zur Burt-Pyramide analoge Regel kann man als möglichen Spezialfall auf einer Khalimsky-Ebene realisieren. Interessanter sind jedoch irreguläre Reduktionen, weil man damit Form und Nachbarschaft der Primitive an den tatsächlichen Bildinhalt anpassen kann.

Damit ähnelt eine Zellpyramide einem Atlas, der ein bestimmtes Gebiet auf mehreren Karten mit unterschiedlichem Maßstab darstellt. Je gröber der Maßstab einer Karte ist, desto stärker werden die dargestellten Geländemerkmale generalisiert. Die Art der Generalisierung hängt dabei von Typ und Form der Merkmale ab und kann sich von Fall zu Fall stark unterscheiden. Genau wie ein Atlas sind Zellpyramiden dann nützlich, wenn für verschiedene Fragestellungen unterschiedlich detaillierte Darstellungen der Szene benötigt werden. Je nach Bedarf kann man Details ignorieren oder hervorheben, oder mit Hilfe einer coarse-to-fine-Analyse das Wissen über eine bestimmte Struktur nach und nach verfeinern, indem man Merkmale von groben bis zu feinen Auflösungen verfolgt.

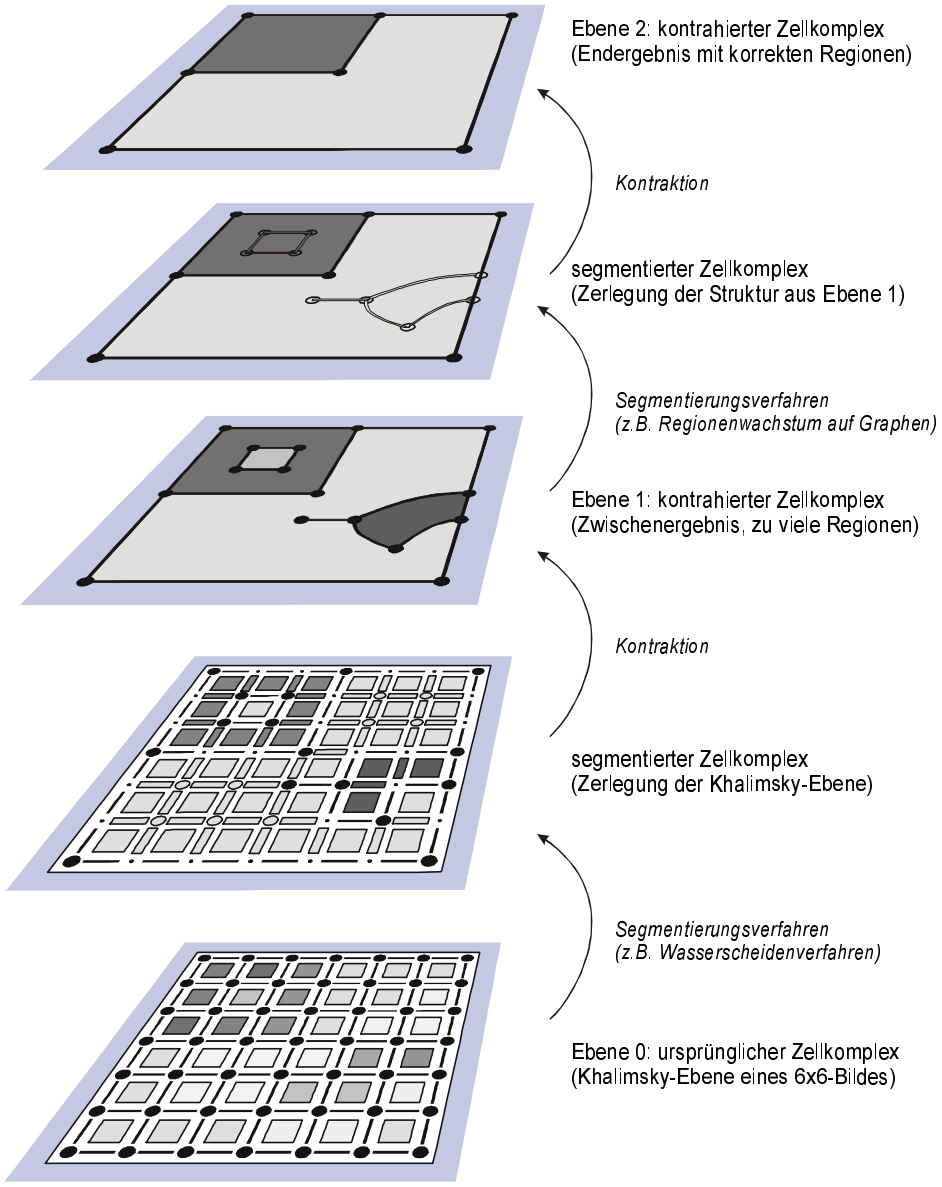


Abbildung 30: Beispiel für eine Zellpyramide, die durch mehrmalige Segmentierung eines Bildes auf aufeinanderfolgenden Ebenen der Pyramide gewonnen wird

Zellpyramiden sind auch dann sehr nützlich, wenn man die gesuchte Segmentierung einer Szene nicht in einem Schritt erreichen kann. Nicht wenige Segmentierungsalgorithmen neigen zum Beispiel dazu, eine Übersegmentierung zu produzieren, d.h. das Bild in zu viele (und damit zu kleine) Regionen aufzuteilen. Um diese Fehler zu korrigieren, muß man in einem oder mehreren Nachbearbeitungsschritten kleine Regionen zu größeren zusammenfassen. Dafür bietet sich eine Zellpyramide an: man beginnt mit einer Segmentierung der Khalimsky-Ebene, wie man sie zum Beispiel mit dem modifizierten Wasserscheidenverfahren aus Abschnitt 8.4.1 bestimmen kann. Diese initiale Segmentierung besteht meist aus zu vielen Regionen. Daher kontrahieren wir sie zu einem neuen Zellkomplex und segmentieren mit einem anderen Segmentierungsverfahren erneut. Im Idealfall werden dadurch alle fälschlicherweise getrennten Regionen zusammengefaßt, andernfalls muß das Verfahren iterativ wiederholt werden. Abschnitt 8.4.3 beschreibt einen entsprechenden Algorithmus.

Abbildung 30 zeigt ein Beispiel für eine solche iterative Segmentierung. Auf der untersten Ebene (Ebene 0) sehen wir die Khalimsky-Ebene, die zu einem Bild der Größe 6x6 korrespondiert. Mit Hilfe eines geeigneten Verfahrens segmentieren wir die Khalimsky-Ebene. Im Beispiel werden dadurch unter anderem 4 Regionen (neben der Außenzelle) gefunden. Die segmentierte Khalimsky-Ebene wird danach kontrahiert, so daß sich der Zellkomplex der Pyramidenebene 1 ergibt. Aufrund seiner unregelmäßigen Struktur stellen wir ihn als Graphen dar. Ein zweites Segmentierungsverfahren erkennt nun, daß die zwei kleinen 2-Zellen fälschlicherweise von ihrer Umgebung abgetrennt wurden (Übersegmentierung). Diese 2-Zellen werden deshalb mit den beiden größeren 2-Zellen und den dazwischen liegenden 0- und 1-Zellen zu je einer neuen Region zusammengefaßt. Die folgende Kontraktion dieser zweiten Segmentierung liefert schließlich den Zellkomplex auf Ebene 2, der hier in Bezug auf ein anwendungsspezifisches Kriterium als korrekt angesehen wird. Wäre das Abbruchkriterium noch nicht erfüllt, könnten weitere Ebenen der Pyramide berechnet werden.

Man bezeichnet Pyramiden, die keine starren Reduktionsregeln verwenden, als *irreguläre Pyramiden*. Einige Autoren haben bereits irreguläre Pyramiden in der Bildanalyse verwendet. Montanvert, Meer und Rosenfeld [MONTAN+91] schlagen eine irreguläre Pyramide auf der Basis des Regionennachbarschaftsgraphen vor. Das Ziel besteht dabei insbesondere in der Entwicklung paralleler Algorithmen zur Analyse von Zusammenhangskomponenten und zur Bildsegmentierung. Kropatsch [KROPATSCH95] stellt fest, daß die Betrachtung des Regionennachbarschaftsgraphen allein nicht ausreicht und erweitert das Verfahren durch Hinzunahme des dualen Graphen. Außerdem verallgemeinert er die Reduktionsregel durch die Einführung von Reduktionsparametern, die die Zuordnung von Zellen zur nächsten Ebene beschreiben. Dieses Verfahren ist ein Spezialfall der hier vorgestellten Zellpyramiden. Auch das Verfahren von Tieck und Gerloff [TIECKGERL97], bei dem eine initiale

Wasserscheidensegmentierung iterativ durch Zusammenfassen von Regionen verbessert wird, läßt sich mit Zellpyramiden realisieren. Die bei [TIECKGERL97] besonders wichtige Funktion des Rückgängigmachens einer Verschmelzung wird durch die Rücktransformationen der Euler-Operatoren realisiert und entspricht dem Übergang von einer höheren auf eine niedrigere Ebene der Zellpyramide.

Unser Verfahren zum Aufbau einer irregulären Pyramide auf der Basis von Zellkomplexen hat eine Reihe von Vorteilen gegenüber bisherigen Verfahren:

- Durch die Definitionen des Zellkomplexes und der Zellpyramide ist gesichert, daß die Axiome eines topologischen Raumes auf jeder Pyramidenebene erfüllt sind. Damit sind stets konsistente Definitionen von Nachbarschaften, Zusammenhangskomponenten und Konturen möglich.
- Die Vollständigkeit der betrachteten Transformationen wurde explizit bewiesen. Es gibt keine Spezialfälle.
- Die Definition eines segmentierten Zellkomplexes ist sehr allgemein, so daß sich viele verschiedene Reduktionsverfahren realisieren lassen, wenn nötig auch auf jeder Ebene ein anderes.
- Jede Segmentierung kann wieder in einen Zellkomplex kontrahiert werden. Deshalb ist es nicht notwendig, eine „Segmentierung der Segmentierung“ zu definieren. Dies vereinfacht die Implementation von Datenstrukturen und Algorithmen für Zellpyramiden.

7.5 Schnittstellenkonzepte für Zellkomplexe

Nachdem wir im vorigen Abschnitt zelluläre Komplexe als abstrakte mathematische Konzepte definiert haben, wollen wir nun entsprechende Schnittstellenkonzepte entwickeln. Zur vollständigen Beschreibung eines Zellkomplexes benötigen wir Iteratoren, die die topologische Struktur des Zellkomplexes wiedergeben, und Zugriffsobjekte, mit denen man die Iteratoren erzeugen und ineinander umwandeln kann. Gleichzeitig dienen die Zugriffsobjekte auch zur Abfrage der anwendungsspezifischen Zusatzinformationen, die mit jeder Zelle verknüpft werden können. Außerdem werden wir Funktionsobjekte für Euler-Operatoren definieren, um auch die Schnittstellen für Transformationen zwischen Zellkomplexen zu vereinheitlichen.

7.5.1 Iteratoren für Zellkomplexe

Da Zellkomplexe Datenstrukturen sind, die eine große Zahl gleichartiger Elemente enthalten, bietet es sich an, die Schnittstelle in Form von Iteratoren zu definieren. Wir benötigen zunächst Iteratoren, die uns sämtliche Instanzen der drei Zellentypen liefern, die in einem Zellkomplex enthalten sind. Wir werden dabei die Bezeichnungen Node für 0-Zelle, Edge für 1-Zelle und Face für 2-Zelle verwenden.

Node Iterator: Ein Node Iterator ist ein linearer Forward Iterator, der sämtliche 0-Zellen eines Zellkomplexes liefert.

Edge Iterator: Ein Edge Iterator ist ein linearer Forward Iterator, der sämtliche 1-Zellen eines Zellkomplexes liefert.

Face Iterator: Ein Face Iterator ist ein linearer Forward Iterator, der sämtliche 2-Zellen eines Zellkomplexes liefert.

Der Benutzer dieser Iteratoren kann dabei im allgemeinen keine Annahmen über eine bestimmte Reihenfolge machen, in der die Zellen besucht werden. Allerdings muß die Reihenfolge bei mehrfachem Durchlaufen der Sequenzen gleich bleiben. Spezielle Implementationen von Zellkomplexen können weitergehende Garantien über eine bestimmte Reihenfolge abgeben.

Um Informationen über die Verknüpfung des Zellkomplexes zu erfragen, benötigen wir weitere Schnittstellen. Die wichtigste Struktur ist die Menge der von einer 0-Zelle begrenzten 1- und 2-Zellen. Aufgrund der Einbettung in die Ebene haben diese 1-Zellen (und damit die dazwischen liegenden 2-Zellen) eine eindeutige Ordnung, die als Inzidenzordnung bezeichnet wird. Da die Inzidenzordnung zyklisch ist, definieren wir die Schnittstelle als Zirkulator (vergleiche Abschnitt 5.7.1):

Node Incidence Circulator: Der Node Incidence Circulator ist ein Bidirectional Circulator, der die zu einer gegebenen 0-Zelle inzidenten 1-Zellen in der durch die Inzidenzordnung festgelegten Reihenfolge liefert. Wir bezeichnen diesen Zirkulator im folgenden kurz als **Ray Circulator**.

Obwohl dieser Zirkulator so definiert ist, daß er die 1-Zellen abarbeitet, können wir ihn auch für die 2-Zellen verwenden, die von der umlaufenen 0-Zelle begrenzt werden. Anstatt auf die 1-Zelle zuzugreifen, auf die der Zirkulator gerade zeigt, greifen wir auf die 2-Zelle zu, die jeweils links von der aktuellen 1-Zelle liegt. Dies können wir, wie weiter unten gezeigt wird, mit Hilfe von Zugriffsobjekten erreichen, so daß kein weiterer Zirkulator notwendig wird.

Zusätzlich zur normalen Zirkulatorfunktionalität muß der Ray Circulator eine Funktion `jumpToOpposite()` anbieten, mit der wir die Richtung des Zirkulators umkehren können: der neue Zirkulator referenziert nach wie vor die gleiche 1-Zelle, umläuft aber nun die 0-Zelle, die vorher Endpunkt der betreffenden 1-Zelle war.

Diese Vertauschung von Anfangs- und Endpunkt erweist sich als sehr hilfreich bei der Implementation von Navigationsalgorithmen auf Zellkomplexen.

Etwas komplizierter ist der Zugriff auf den Rand einer 2-Zelle, weil dieser Rand mehrere Zusammenhangskomponenten haben kann. Wir benötigen deshalb zunächst einen Iterator, der diese Zusammenhangskomponenten liefert:

Boundary Components Iterator: Der Boundary Components Iterator ist ein Forward Iterator, der die Zusammenhangskomponenten des Randes einer gegebenen 2-Zelle liefert.

Die Länge der Sequenz entspricht der Anzahl der Zusammenhangskomponenten des Randes. Hat eine 2-Zelle keine Löcher, so besteht dieses Sequenz nur aus einem Element. In Abschnitt 7.3.2 hatten wir für die Zusammenhangskomponenten des Randes den Begriff *Kontur* eingeführt. Jede Kontur ist ein Zyklus. Wir können deshalb für jede Kontur einen Zirkulator definieren, der die Kontur in der Reihenfolge umläuft, die sich aufgrund der Konturordnung ergibt:

Contour Nodes Circulator: Der Contour Nodes Circulator ist ein Bidirectional Circulator, der die 0-Zellen einer gegebenen Kontur in der durch die Konturordnung festgelegten Reihenfolge durchläuft.

Besteht die Kontur nur aus einer isolierten 0-Zelle, so hat die Sequenz die Länge eins. Einen analogen Zirkulator definieren wir für die 1-Zellen der Kontur:

Contour Edges Circulator: Der Contour Edges Circulator ist ein Bidirectional Circulator, der die 1-Zellen einer gegebenen Kontur in der durch die Konturordnung festgelegten Reihenfolge durchläuft.

Wir können dafür nicht den Contour Nodes Circulator verwenden, weil dieser für isolierte Punkte die Länge eins hat, während der Contour Edges Circulator in diesem Fall die Länge null hat (eine Zirkulatorsequenz der Länge null bezeichnet man als *singulär*).

Auch der Contour Edges Circulator muß die Funktion `jumpToOpposite()` unterstützen, die die Richtung des Zirkulators umkehrt. Nach Anwendung dieser Operation referenziert der Zirkulator nach wie vor die gleiche 1-Zelle, umläuft aber jetzt eine Kontur der benachbarten 2-Zelle. Diese Vertauschung von rechter und linker 2-Zelle erweist sich bei der Navigation auf Zellkomplexen als sehr nützlich.

Es sei an dieser Stelle darauf hingewiesen, daß Ray Circulator und Contour Edges Circulator *dual* zueinander sind: der Ray Circulator arbeitet auf den 1-Zellen, die zu einer 0-Zelle inzident sind, und der Contour Edges Circulator auf denen, die zu einer 2-Zelle inzident sind. In beiden Fällen handelt es sich um bidirektionale Zirkulatoren mit identischer Schnittstelle. Diese Übereinstimmung spiegelt die Tatsache, daß Kantengraph und Flächengraph duale Graphen sind (siehe Abschnitt 7.3.1).

Viele Algorithmen interessieren sich nicht dafür, ob der Rand einer 2-Zelle aus mehreren Zusammenhangskomponenten besteht oder nicht. Sie wollen nur auf

sämtliche 1-Zellen des Randes zugreifen, ohne zusätzlichen Aufwand für die Verwaltung der einzelnen Konturen einer Fläche treiben zu müssen. Für diesen Fall definieren wir den folgenden Iterator:

Boundary Edges Iterator: Der Boundary Edges Iterator ist ein Forward Iterator, der sämtliche 1-Zellen abarbeitet, die zu einer gegebenen 2-Zelle inzident sind.

In der Praxis kann man diesen Iterator einfach als Adapter implementieren: der Adapter enthält einen Boundary Components Iterator und führt für jede von diesem referenzierte Kontur einen vollen Umlauf mit dem jeweiligen Contour Edges Circulator durch. Zwar könnten Algorithmen dies auch selbst erledigen, doch verbessert sich die Zerlegung der Systemfunktionalität, wenn wir diese Aufgabe in einem eigenen Baustein kapseln.

Einige Algorithmen benötigen nicht den vollständigen Zellkomplex, sondern nur den Regionennachbarschaftsgraphen, wie wir ihn in Abschnitt 7.1 definiert hatten. Diese Algorithmen greifen direkt auf die Nachbarflächen zu, ohne sich für die dazwischen liegenden Kanten zu interessieren. Dies läßt sich vereinfachen, wenn der folgende Iterator benutzt wird:

Face Adjacency Iterator: Der Face Adjacency Iterator liefert für eine gegebene 2-Zelle sämtliche benachbarten 2-Zellen.

Auch diesen Iterator kann man als Adapter implementieren, und zwar auf Basis des Boundary Edges Iterator. Der Adapter kann beispielsweise so implementiert werden, daß er bei der Initialisierung einmal die Sequenz des Boundary Edges Iterator abarbeitet und alle dabei gefundenen benachbarten 2-Zellen in einer Liste speichert. Diese Liste wird dann während der Iteration abgearbeitet.

Eine andere Möglichkeit besteht darin, eine spezielle, für diesen Anwendungsfall optimierte Datenstruktur für Regionennachbarschaftsgraphen zu definieren. Diese Datenstruktur implementiert den Face Adjacency Iterator direkt, also nicht als Adapter, und ist deshalb effizienter. Die Schnittstelle des Face Adjacency Iterator ändert sich jedoch nicht, so daß Algorithmen mit beiden Varianten arbeiten können. Dies unterstreicht einmal mehr die Flexibilität der generischen Programmierung.

7.5.2 Zugriffsobjekte für Zellkomplexe

Die Zugriffsobjekte für Zellkomplexe haben zwei Aufgaben. Einerseits werden sie benutzt, um die im vorigen Abschnitt definierten Iteratoren bzw. Zirkulatoren zu erzeugen und ineinander umzuwandeln. Andererseits dienen sie dazu, anwendungsspezifische Informationen abzufragen bzw. zu schreiben, die mit den Zellen verknüpft sind (z.B. Koordinaten für 0-Zellen, eine Geradenanpassung für 1-Zellen oder der mittlere Grauwert einer 2-Zelle). Die Konzentration dieser Funktionalität in

Zugriffsobjekten befreit die Iteratoren davon, anwendungsspezifische Besonderheiten zu kennen und zu berücksichtigen und dient damit wiederum einer besseren Systemzerlegung und einer höheren Flexibilität.

Wir werden hier allgemeine Zugriffsobjekte für die erste Aufgabe definieren. Zur Erfüllung der zweiten Aufgabe können diese Objekte um anwendungsspezifische Funktionen erweitert werden. Darüber hinaus kann man zusätzliche Zugriffsobjekte definieren. Jedes Zugriffsobjekt kann Informationen über eine bestimmte Art von Zellen liefern, wenn ihm ein geeigneter Iterator übergeben wird.

Zugriffsobjekte für 0-Zellen

Der Node Accessor liefert Informationen über eine 0-Zelle, wenn ihm ein Node Iterator oder ein Contour Nodes Circulator übergeben wird. Er hat die folgende Schnittstelle:[25]

```
struct NodeAccessor
{
    // erzeuge Ray Circulator (Node Incidence Circulator) für aktuelle 0-Zelle
    RayCirculator rayCirculator(NodeIterator);

    // erzeuge Ray Circulator für aktuelle 0-Zelle, der die gleiche 1-Zelle
    // referenziert wie der übergebene ContourNodesCirculator
    RayCirculator rayCirculator(ContourNodesCirculator);

    // wandle ContourNodesCirculator in NodeIterator für die gleiche 0-Zelle um
    NodeIterator nodeIterator(ContourNodesCirculator);

    // ermittle den Grad der aktuellen 0-Zelle
    int degree(NodeIterator);
    int degree(ContourNodesCirculator);

    // Beispiel für Zugriff auf anwendungsspezifische 0-Zellen-Informationen
    NodeInfo operator()(NodeIterator);
    NodeInfo operator()(ContourNodesCirculator);
};
```

Ähnlich definieren wir den Zugriff auf 0-Zellen, wenn Iteratoren gegeben sind, die auf 1-Zellen verweisen. Wir müssen hier allerdings beachten, daß jede 1-Zelle von zwei 0-Zellen begrenzt wird. Wir benötigen deshalb zwei Zugriffsobjekte, eins für den Anfangs- und eins für den Endpunkt der 1-Zelle. Welche der beiden 0-Zellen Anfang und Ende ist, ergibt sich aus der Semantik des Iterators. Bei einem Ray Circulator ist der Anfangspunkt stets der, den der Zirkulator umläuft. Bei einem Contour Edges Circulator ist der Anfangspunkt diejenige 0-Zelle, die in der Zellen-

[25] Bei der Umsetzung dieser Schnittstelle kann man die Flexibilität erhöhen, indem man variable Anwendungsinformationen zuläßt: `template <class NodeInfo> struct NodeAccessor {...};`

folge der Kontur vor der aktuellen 1-Zelle steht. Bei einem Edge Iterator können Anfangs- und Endpunkt beliebig oder nach anwendungsspezifischen Kriterien gewählt werden. Wir erhalten folgende Zugriffsobjekte:

```
struct NodeAtStartAccessor
{
    // wandle Iteratoren/Zirkulatoren in RayCirculator für den Startpunkt um
    // (RayCirculator referenziert gleiche 1-Zelle wie der übergebene Iterator)
    RayCirculator rayCirculator(EdgeIterator);
    RayCirculator rayCirculator(ContourEdgesCirculator);
    RayCirculator rayCirculator(BoundaryEdgesIterator);

    // wandle Iteratoren/Zirkulatoren in NodeIterator für die 0-Zelle um
    NodeIterator nodeIterator(EdgeIterator);
    NodeIterator nodeIterator(RayCirculator);
    NodeIterator nodeIterator(ContourEdgesCirculator);
    NodeIterator nodeIterator(BoundaryEdgesIterator);

    // ermittle den Grad des Startpunktes der übergebenen 1-Zelle
    int degree(EdgeIterator);
    int degree(RayCirculator);
    int degree(ContourEdgesCirculator);
    int degree(BoundaryEdgesIterator);

    // Zugriff auf anwendungsspezifische Informationen des Startknotens
    NodeInfo operator()(EdgeIterator);
    NodeInfo operator()(RayCirculator);
    NodeInfo operator()(ContourEdgesCirculator);
    NodeInfo operator()(BoundaryEdgesIterator);
};

struct NodeAtEndAccessor {
    ... // analog
};
```

Als zusätzliche Informationen über eine 0-Zelle benötigt man beispielsweise die Koordinaten des zugehörigen Punktes im Bild. Diese Koordinaten können in einem anwendungsspezifischen `NodeInfo`-Objekt gespeichert werden. Ähnlich können wir bei den folgenden Zugriffsobjekten geometrische und andere Beschreibungen in entsprechenden `EdgeInfo`- und `FaceInfo`-Objekten speichern.

Zugriffsobjekte für 1-Zellen

Ein Algorithmus kann Informationen über eine 1-Zelle abfragen, wenn ihm einer der Iteratoren zur Verfügung steht, die eine 1-Zelle eindeutig identifizieren, d.h. ein Edge Iterator, ein Ray Circulator, ein Contour Edges Circulator oder ein Boundary

Edges Iterator. Die Schnittstellendefinition der Zugriffsobjekte folgt dem gleichen Prinzip wie bei den 0-Zellen:

```
struct EdgeAccessor
{
    // wandle Iteratoren/Zirkulatoren in EdgeIterator für die 1-Zelle um
    EdgeIterator edgeIterator(RayCirculator);
    EdgeIterator edgeIterator(ContourEdgesCirculator);
    EdgeIterator edgeIterator(BoundaryEdgesIterator);

    // Zugriff auf anwendungsspezifische Informationen der 1-Zelle
    EdgeInfo operator()(EdgeIterator);
    EdgeInfo operator()(RayCirculator);
    EdgeInfo operator()(ContourEdgesCirculator);
    EdgeInfo operator()(BoundaryEdgesIterator);
};
```

Zugriffsobjekte für 2-Zellen und ihre Konturen

Nach dem bereits bewährten Muster definieren wir nun auch Zugriffsobjekte für 2-Zellen. Eine bestimmte 2-Zelle wird identifiziert durch einen Face Iterator oder einen Boundary Components Iterator. Diese benutzen das folgende Zugriffsobjekt:

```
struct FaceAccessor
{
    // Erzeugen der zur 2-Zelle gehörenden BoundaryComponentsIteratoren
    BoundaryComponentsIterator boundaryComponentsIteratorBegin(FaceIterator);
    BoundaryComponentsIterator boundaryComponentsIteratorEnd(FaceIterator);

    // Erzeugen der zur 2-Zelle gehörenden BoundaryEdgesIteratoren
    BoundaryEdgesIterator boundaryEdgesIteratorBegin(FaceIterator);
    BoundaryEdgesIterator boundaryEdgesIteratorEnd(FaceIterator);

    // Erzeugen der zur 2-Zelle gehörenden FaceAdjacencyIteratoren
    FaceAdjacencyIterator faceAdjacencyIteratorBegin(FaceIterator);
    FaceAdjacencyIterator faceAdjacencyIteratorEnd(FaceIterator);

    // Erzeugen eines ContourNodesCirculator für die aktuelle Kontur
    ContourNodesCirculator contourNodesCirculator(BoundaryComponentsIterator);

    // Erzeugen eines ContourEdgesCirculator für die aktuelle Kontur
    ContourEdgesCirculator contourEdgesCirculator(BoundaryComponentsIterator);

    // Umwandeln eines BoundaryComponentsIterator in einen FaceIterator
    FaceIterator FaceIterator(BoundaryComponentsIterator);

    // Beispiel für die Abfrage anwendungsspezifischer Informationen der 2-Zelle
    FaceInfo operator()(FaceIterator);
    FaceInfo operator()(FaceAdjacencyIterator);
};
```

Bei der Definition von Zugriffsobjekten auf die Informationen der 2-Zelle für die übrigen Iteratoren bzw. Zirkulatoren müssen wir beachten, daß eine 1-Zelle stets zwei 2-Zellen begrenzt. Wir müssen deshalb getrennte Zugriffsobjekte für die linke und die rechte 2-Zelle definieren. Welche die linke oder rechte 2-Zelle ist, ergibt sich wieder aus der Einbettung des Zellkomplexes in die Ebene und der Orientierung des übergebenen Iterators. Bei einem Ray Circulator ist die *linke* 2-Zelle immer die, die in mathematisch *positiver* Richtung auf die referenzierte 1-Zelle folgt. Die linke 2-Zelle liegt also zwischen der aktuellen und der nachfolgenden Position des Zirkulators. Bei einem Contour Edges Circulator, Boundary Edges Iterator oder einem Face Adjacency Iterator ist die linke 2-Zelle immer diejenige, die der Iterator/Zirkulator gerade umläuft. Bei einem Edge Iterator kann die linke und rechte 2-Zelle beliebig oder nach anwendungsspezifischen Kriterien gewählt werden. Wir erhalten die folgenden Zugriffsobjekte:

```
struct FaceAtLeftAccessor
{
    // Erzeugen eines ContourEdgeIterators für die linke 2-Zelle
    ContourEdgeIterator contourEdgeIterator(EdgeIterator);
    ContourEdgeIterator contourEdgeIterator(RayCirculator);
    ContourEdgeIterator contourEdgeIterator(BoundaryEdgesIterator);

    // Umwandeln des übergebenen Iterators in einen FaceIterator
    FaceIterator faceIterator(RayCirculator);
    FaceIterator faceIterator(ContourEdgesCirculator);
    FaceIterator faceIterator(BoundaryEdgesIterator);
    FaceIterator faceIterator(FaceAdjacencyIterator);

    // Beispiel für Abfrage anwendungsspezifischer Informationen der 2-Zelle
    FaceInfo operator()(EdgeIterator);
    FaceInfo operator()(RayCirculator);
    FaceInfo operator()(ContourEdgesCirculator);
    FaceInfo operator()(BoundaryEdgesIterator);
    FaceInfo operator()(FaceAdjacencyIterator);
};

struct FaceAtRightAccessor {
    ... // analog
};
```

Zusammen mit den Iteratoren und Zirkulatoren reichen diese Zugriffsobjekte aus, um die Struktur eines Zellkomplexes vollständig zu beschreiben. Algorithmen, die nur lesend auf einen Zellkomplex zugreifen wollen, benötigen keine weiteren Informationen als diejenigen, die durch diese Objekte repräsentiert werden.

Es gehört zu den überraschenden und erfreulichen Ergebnissen der generischen Programmierung, daß eine so komplizierte Datenstruktur wie ein zellulärer Komplex durch eine vergleichsweise einfache und weitgehend einheitliche Schnittstelle

(neun Iteratoren bzw. Zirkulatoren sowie sieben Zugriffsobjekte) beschrieben werden kann, wobei diese Schnittstelle noch nicht einmal minimal ist (Boundary Edges Iterator und Face Adjacency Iterator können auf der Basis der übrigen Iteratoren/Zirkulatoren implementiert werden).

7.5.3 Funktionsobjekte für Euler-Operatoren

Die bisher vorgestellten Teile der Schnittstelle für Zellkomplexe unterstützen noch keine Euler-Operatoren und erlauben damit noch nicht die Transformation eines Zellkomplexes in einen anderen, wie sie z.B. bei der Kontraktion erforderlich ist. Wir definieren für jede der in Abschnitt 7.4.1 eingeführten Euler-Operatoren ein Funktionsobjekt. D.h., die Transformation wird bei Aufruf des operator() ausgeführt. An den jeweiligen operator() übergeben wir einen Iterator, der die zu transformierenden Zellen eindeutig festlegt. Wo angemessen, wird ein Iterator auf die „überlebende“ Zelle der betreffenden Transformation zurückgegeben:

```
struct MergeNodes
{
    // Verschmelzung der Endknoten der gegebenen 1-Zelle
    NodeIterator operator()(EdgeIterator);
};

struct MergeEdges
{
    // Verschmelzung der von der gegebenen 0-Zelle begrenzten 1-Zellen
    // (die 0-Zelle muß den Grad 2 haben und darf keine Schlinge begrenzen)
    EdgeIterator operator()(NodeIterator);
};

struct MergeFaces
{
    // Verschmelzung der von der gegebenen 1-Zelle begrenzten 2-Zellen
    // (die 1-Zelle darf kein Brückenelement sein)
    FaceIterator operator()(EdgeIterator);
};

struct RemoveBridge
{
    // Entfernen eines Brückenelements (Iterator muß ein solches referenzieren)
    void operator()(EdgeIterator);
};

struct RemoveIsolatedNode
{
    // Entfernen einer isolierten 0-Zelle
    // (Iterator muß eine solche referenzieren)
    void operator()(NodeIterator);
};
```

Da bei der Verschmelzung von 2-Zellen neue Brückenelemente und isolierte 0-Zellen entstehen können, die separat entfernt werden müssen, ist es zweckmäßig, die dazu notwendigen Schritte in einem eigenen Funktionsobjekt zu kapseln:

```
struct MergeFacesCompletely
{
    // Verschmelzung von zwei 2-Zellen und Entfernung aller dabei eventuell
    // enstehenden Brückenelemente und isolierten 0-Zellen
    FaceIterator operator()(EdgeIterator);
};
```

Mit Hilfe dieser Euler-Operatoren implementieren wir die Kontraktion eines segmentierten Zellkomplexes wie folgt: jede Teilmenge der Segmentierung (d.h. jede Region, jede Linie, jeder Vertex) wird durch ein Objekt repräsentiert, das angibt, welche Zellen zu dieser Teilmenge gehören. Diese Objekte speichern Referenzen auf die Zellen in Form von Iteratoren. Im Falle eines Vertex-Objekts erfragen wir nun alle Iteratoren, die auf 1-Zellen verweisen, und rufen mit ihnen den Funktor `MergeNodes` auf. Im Ergebnis bleibt nur noch eine 0-Zelle übrig, die den Vertex in kontrahierten Zellkomplex repräsentiert. Analog verfahren wir mit den 0-Zellen eines Linien-Objekts: wir wenden den Funktor `MergeEdges` an, bis nur noch eine 1-Zelle übrig ist. Bei einem Regionen-Objekt fragen wir zuerst alle 1-Zellen ab und löschen sie entweder mit `MergeFaces` oder `RemoveBridge`, je nachdem ob es sich um ein Grenzelement oder ein Brückenelement handelt. Danach löschen wir alle 0-Zellen mittels `RemoveIsolatedNode`.

7.6 Zusammenfassung des Kapitels

In diesem Kapitel haben wir folgendes gezeigt:

- Es wird eine einheitliche Repräsentation für Segmentierungsergebnisse benötigt, die von möglichst vielen Algorithmen gleichermaßen benutzt werden kann. Diese Repräsentation muß Punkte, Kanten und Regionen sowie deren Merkmale und Relationen enthalten.
- Bisher übliche Repräsentationen erfüllen diese Forderungen nicht vollständig. Die Repräsentationen der Ergebnisse verschiedener Algorithmen sind inkompatibel, sie enthalten nicht immer alle Merkmalstypen, und es gibt Probleme bei der widerspruchsfreien Definition von Nachbarschaften, Konturen und anderen topologischen Eigenschaften.
- Ebene Zellkomplexe eignen sich als Repräsentationsform für Segmentierungsergebnisse. Erstmals haben wir den Begriff „Segmentierung“ auf der Basis von Zellkomplexen exakt definiert. Aus dieser Definition folgen widerspruchsfreie Definitionen von Merkmalsnachbarschaften und Konturen. Außerdem kann jeder Segmentierung durch Kontraktion wiederum ein Zellkomplex zugeordnet werden. Die notwendigen Transformationen hierfür wurden in Form von elementaren Euler-Operatoren angegeben. Durch iterative Wiederholung von Segmentierung und Kontraktion kann eine Zellpyramide aufgebaut werden.
- Wir haben uns nicht auf die abstrakte mathematische Definition ebener Zellkomplexe beschränkt, sondern außerdem Schnittstellenkonzepte definiert, die generische Implementierungen von Zellkomplexen erlauben. Die Schnittstelle besteht aus einer Menge von abstrakten Iteratoren, Zirkulatoren, Zugriffsobjekten und Funktoren.

Kapitel 8

Datenstrukturen und Algorithmen für zelluläre Komplexe

8.1 Implementation von Zellkomplexen als Graphen

Aufgrund der engen Beziehung zwischen ebenen Zellkomplexen und ebenen Graphen bietet es sich an, zur Implementation von Zellkomplexen Techniken zu verwenden, die sich bereits bei Graphen bewährt haben. Insbesondere können wir auf Erfahrungen zur Repräsentation der Oberflächen von Polyedern zurückgreifen, denn Polyeder sind ebenfalls Graphen aus Punkten, Kanten und Flächen. Dieses Problem wurde im Rahmen der Computational Geometry intensiv erforscht [MÄNTYLÄ88, KETTNER98], und wir können grundlegende Implementationsideen übernehmen.

Zunächst definieren wir für jeden Zelltyp eine interne Klasse, die Informationen zur internen Verwaltung der Zellinstanzen speichert. Für jede Zelle erzeugen wir eine Instanz der entsprechenden Klasse, und alle Instanzen jeder Klasse werden in einem Array (Klasse `vector` aus der C++-Standardbibliothek) abgespeichert. Die Iteratoren dieser Arrays erfüllen die Anforderungen an den Face Iterator, Edge Iterator bzw. Node Iterator, so daß wir bereits die ersten drei geforderten Iteratoren realisiert haben:

```
struct CellComplexFace { ... };
struct CellComplexEdge { ... };
struct CellComplexNode { ... };
```

```
typedef vector< CellComplexFace >::iterator CellComplexFaceIterator;
typedef vector< CellComplexEdge >::iterator CellComplexEdgeIterator;
typedef vector< CellComplexNode >::iterator CellComplexNodeIterator;
```

Als nächstes müssen wir die Inzidenzordnung jedes Knotens bestimmen und daraus den Ray Circulator ableiten. [KETTNER98] beschreibt drei Methoden, die sich in der Computational Geometry für diesen Zweck bewährt haben. Jede der drei Methoden kann für die Implementation eines Zellkomplexes verwendet werden, der unseren Anforderungen entspricht. Wir wählen hier die Methode der *Halbkanten* (siehe auch [MÄNTYLÄ88]). Jeder 1-Zelle werden zwei Halbkanten zugeordnet. Jede Halbkante stellt die Beziehung zu einer der beiden 0- und 2-Zellen her, zu denen die 1-Zelle inzident ist. Außerdem bilden alle Halbkanten, die zu einer bestimmten 0-Zelle inzident sind, eine doppelt verkettete zyklische Liste, die die Inzidenzordnung an dieser 0-Zelle repräsentiert: [26]

```
struct HalfEdge
{
    HalfEdge * previous, * next;  // doppelt verkettete Liste der
                                  // Inzidenzordnung
    HalfEdge * opposite;          // die andere Halbkante der 1-Zelle
    CellComplexEdge * edge;       // 1-Zelle, zu der die Halbkante gehört
    CellComplexNode * node;       // inzidente 0-Zelle
    CellComplexFace * face;       // inzidente 2-Zelle
};
```

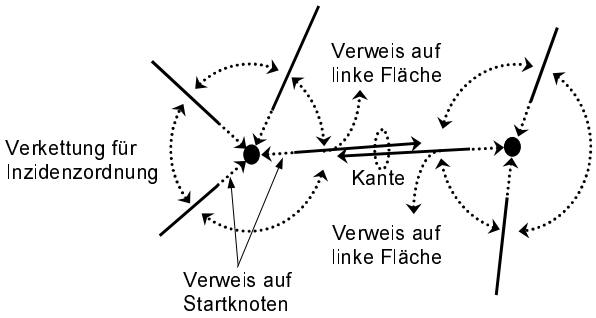


Abbildung 31: Darstellung der Struktur eines Zellkomplexes durch Halbkanten (schwarze Linien) und deren Verknüpfungen (gestrichelte Linien)

Wie der Quellcode sowie Abbildung 31 zeigen, verweist jede Halbkante auf eine 1-Zelle (`edge`), auf die andere Halbkante dieser 1-Zelle (`opposite`), auf eine 2-Zelle,

[26] Zur Vereinfachung der Darstellung wollen wir in dieser Beispielimplementation keine isolierten 0-Zellen zulassen. Eine entsprechende Erweiterung kann man beispielsweise dadurch realisieren, daß man isolierte 0-Zellen intern mit Hilfs-Halbkanten verbindet, die nach außen nicht sichtbar sind.

die von dieser 1-Zelle begrenzt wird (face), und auf eine 0-Zelle, die die 1-Zelle begrenzt (node). Außerdem speichert sie einen Zeiger auf die in der Inzidenzordnung vorausgehende (previous) und nachfolgende (next) Halbkante. Die Halbkanten enthalten damit den größten Teil der Information über die Struktur des Zellkomplexes.

Mit Hilfe der Halbkanten können wir jetzt den Ray Circulator implementieren. Dieser iteriert über die zyklische Liste, die die Halbkanten an einem gegebenen Knoten bilden. Die Funktion jumpToOpposite() ersetzt die aktuelle Halbkante durch die andere Halbkante der aktuellen 1-Zelle:

```
class CellComplexRayCirculator
{
  public:
    HalfEdge * current_;

    // Initialisierung mit einer Halbkante der aktuellen 0-Zelle
    CellComplexRayCirculator(HalfEdge * begin) : current_(begin) {}

    // Inkrementierung
    CellComplexRayCirculator & operator++() {
        current_ = current_->next;      // zum nächsten Element der Liste
    }

    // Dekrementierung
    CellComplexRayCirculator & operator--() {
        current_ = current_->previous;  // zum vorigen Element der Liste
    }

    void jumpToOpposite() {
        current_ = current_->opposite;  // zur anderen Halbkante der 1-Zelle
    }

    // Zugriff auf die 1-Zellen-Information
    CellComplexEdge * edge() { return current_->egde; }

    // Zugriff auf die 0-Zellen-Information
    CellComplexNode * node() { return current_->node; }

    // Zugriff auf die 2-Zellen-Information
    CellComplexFace * face() { return current_->face; }

    ... // weitere Funktionen
};
```

In den Objekten zur Verwaltung von 1-Zellen speichern wir einen Verweis auf eine der beiden Halbkanten, die der betreffenden 1-Zelle zugeordnet sind. Ebenso speichern wir im Verwaltungsobjekt jeder 0-Zelle einen Verweis auf eine der inzidenten Halbkanten:

```
struct CellComplexEdge {
    HalfEdge * firstHalfedge;
};

struct CellComplexNode {
    HalfEdge * startOfIncidenceList;
};
```

Als nächstes müssen wir den Contour Nodes Circulator und den Contour Edges Circulator implementieren. Diese Zirkulatoren umlaufen eine bestimmte Kontur einer 2-Zelle in der durch die Konturordnung festgelegten Reihenfolge. Wie wir in Abschnitt 7.3.2 erläutert hatten, können wir diese Reihenfolge durch Konturverfolgung aus den Inzidenzordnungen an jedem Knoten bestimmen. Dazu benutzen wir den folgenden Konturverfolgungsalgorithmus:

Konturverfolgung in Zellkomplexen:

0) Voraussetzung: `r` ist ein Ray Circulator, der auf eine (beliebige) 1-Zelle der Kontur verweist (die beiden inzidenten 0-Zellen der 1-Zelle sind dann ebenfalls Bestandteil der Kontur).
1) Setze den Ray Circulator zur nächsten 0-Zelle der Kontur: `r.jumpToOpposite();`
2) Dekrementiere den Ray Circulator, so daß er auf die nächste 1-Zelle der Kontur zeigt: `--r;`

Da der Konturverfolgungsalgorithmus mit Hilfe des Ray Circulator formuliert wurde, wird der `CellComplexContourCirculator`, der diesen Algorithmus in seinem Inkrementierungsoperator verwendet, als Adapter des Ray Circulator implementiert:

```
struct CellComplexContourCirculator
{
    CellComplexRayCirculator adaptee_; // adaptierter Ray Circulator

    // Initialisierung mit einer Halbkante aus der Kontur
    CellComplexContourCirculator(HalfEdge * he)
    : adaptee_(he)
    {}

    // Inkrementierung (ein Schritt der Konturverfolgung)
    CellComplexContourCirculator & operator++()
    {
        // Weitergehen zur nächsten 0-Zelle
        adaptee_.jumpToOpposite();

        // RayCirculator auf die nächste Halbkante der Kontur drehen
        --adaptee_;
        return *this;
    }
```

```
    // Dekrementierung (ein Schritt zurück bei der Konturverfolgung)
    CellComplexContourCirculator & operator--() {
        // RayCirculator auf die vorhergehende Halbkante der Kontur drehen
        ++adaptee_;

        // Zurückgehen zur vorigen 0-Zelle
        adaptee_.jumpToOpposite();
        return *this;
    }

    // Zugriff auf aktuelle 0-Zelle
    CellComplexNode * node() { return adaptee_.node(); }

    // Zugriff auf aktuelle 1-Zelle
    CellComplexEdge * edge() { return adaptee_.edge(); }

    // Zugriff auf die 2-Zellen, deren Kontur wir umlaufen
    CellComplexFace * face() { return adaptee_.face(); }

    ... // weitere Funktionen
};
```

Dieser Iterator kann sowohl die Aufgaben des Contour Nodes Circulator als auch die des Contour Edges Circulator erfüllen. Das Verhalten des ersteren realisieren wir, indem wir den `CellComplexContourCirculator` mit einem `NodeAtStartAccessor` verknüpfen, das des letzteren durch Verknüpfung mit einem `EdgeAccessor` (vgl. Abschnitt 7.5.2):

```
// zur Benutzung mit NodeAtStartAccessor
typedef CellComplexContourCirculator CellComplexContourNodesCirculator;

// zur Benutzung mit EdgeAccessor
typedef CellComplexContourCirculator CellComplexContourEdgesCirculator;
```

In jedem Objekt, das eine 2-Zelle verwaltet, müssen wir Verweise auf sämtliche Konturen dieser 2-Zelle speichern. Wir realisieren dies durch Listen von Zeigern auf Halbkanten. Eine Halbkante aus jeder Kontur wird in der Liste, die zur jeweiligen 2-Zelle gehört, gespeichert (Klasse `list` aus der C++-Standardbibliothek). Mit Hilfe der gespeicherten Zeiger können wir für jede Kontur den zugehörigen `CellComplexContourCirculator` instanziieren. Die Iteratoren der Listen erfüllen die Anforderungen an den Boundary Components Iterator:

```
struct CellComplexFace {
    list<HalfEdge *> contours;
};

typedef list<HalfEdge *>::iterator  CellComplexBoundaryComponentsIterator;
```

Es fehlen nun noch der Boundary Edges Iterator und der Face Adjacency Iterator. Wie wir in Abschnitt 7.5.1 erläutert hatten, werden diese als Adapter auf der Basis der

bereits definierten Iteratoren implementiert. Da sich dabei keine wesentlichen neuen Einsichten ergeben, wollen wir hierauf verzichten.

Als nächstes müssen die Zugriffsobjekte implementiert werden. Wir wollen als Beispiel Teile des Node At Start Accessor vorstellen, die Implementation der übrigen Zugriffsobjekte erfolgt analog. Der `CellComplexNodeAtStartAccessor` hat erstens die Aufgabe, einen Ray Circulator zu erzeugen, wenn ein anderer Iterator gegeben ist:

```
struct CellComplexNodeAtStartAccessor
{
    // Ray Circulator für den Anfangsknoten der gegebenen Edge
    CellComplexRayCirculator rayCirculator(CellComplexEdgeIterator i)
    {
        return CellComplexRayCirculator(i->firstHalfedge);
    }

    CellComplexRayCirculator rayCirculator(CellComplexContourEdgesCirculator i)
    {
        return i.adaptee_;
    }
};
```

In analoger Weise implementiert man die Erzeugung eines Node Iterator. Weiterhin kann dieses Zugriffsobjekt benutzt werden, um den Grad des Startknotens der jeweiligen Kante zu berechnen. Wir implementieren dies, indem wir zählen, wie viele Schritte ein vollständiger Umlauf des Ray Circulator erfordert (falls dies in einem bestimmten Kontext zu lange dauert, kann man den Knotengrad natürlich auch im Verwaltungsobjekt `CellComplexNode` zwischenspeichern):

```
struct CellComplexNodeAtStartAccessor  // Fortsetzung
{
    // Bestimmung des Grads des Startknotens der gegebenen Kante
    int degree(CellComplexRayCirculator i)
    {
        int count = 0;
        if(!i.isSingular())            // kein isolierter Knoten
        {
            CellComplexRayCirculator end = i;
            do { ++count; ++i; } while(i != end);
        }
        return count;
    }

    // sind andere Iteratoren gegeben, wird zuerst ein Ray Circulator erzeugt
    // und dann die soeben definierte Funktion aufgerufen, z.B.
    int degree (CellComplexContourEdgesCirculator i)
    {
        return degree(rayCirculator(i));
    }
};
```

Schließlich müssen wir noch Zugriffsfunktionen für die anwendungsspezifischen Daten implementieren. Es liegt jedoch in der Natur der Sache, daß es hierfür keine allgemeingültige Implementation gibt. Eine Möglichkeit besteht darin, solche Daten in einem der Verwaltungsobjekte, z.B. CellComplexNode, zu speichern. Dies geht sehr einfach, wenn wir das Verwaltungsobjekt als template einer anwendungsspezifischen Klasse definieren:

```
Template <class UserNodeData>
struct CellComplexNode
{
    HalfEdge * startOfIncidenceList;
    UserNodeData userData;
};
```

Mit dieser Definition muß auch das Zugriffsobjekt als template definiert werden. Dann kann der Zugriff auf Nutzerdaten wie folgt implementiert werden:

```
template <class UserNodeData>
struct CellComplexNodeAtStartAccessor
{
    // Abrufen anwendungsspezifischer Daten, die in CellComplexNode
    // gespeichert sind
    UserNodeData operator()(CellComplexEdgeIterator i) {
        return i->firstHalfedge->node->userData;
    }
};
```

Gerade hier wird der Vorteil von Zugriffsobjekten besonders deutlich: jede Anwendung verknüpft andere spezielle Informationen mit den Zellen. In den meisten Fällen wird auch der Mechanismus, wie man an diese Daten jeweils herankommt, unterschiedlich sein. Durch die Verwendung von Zugriffsobjekten sind die Iteratoren vor solchen Änderungen geschützt. Dadurch bleibt der Aufwand gering, anwendungsspezifische Daten jederzeit hinzuzufügen und zu entfernen.

Als letztes wollen wir noch ein Beispiel für die Implementation eines Funktors für Euler-Operatoren angeben. Wir wählen als Beispiel die Transformation „Verschmelzen von benachbarten 0-Zellen“. Der Funktor, der diese Transformation implementiert, erhält als Argument den Ray Circulator, der die zu entfernende 1-Zelle referenziert. Der Algorithmus wird durch Kommentare im folgenden Code erklärt:

```
struct MergeNodes
{
    CellComplexNode * operator()(CellComplexRayCirculator r)
    {
        // die beiden Halbkanten der 1-Zelle extrahieren
        HalfEdge * first  = r.current_;
        HalfEdge * second = first->opposite;
```

```
            // die Information der überlebenden 0-Zelle auf die Halbkanten der zu
            // löschenden 0-Zelle übertragen
            CellComplexNode * survivor = first->node;
            HalfEdge * current = second->next;
            while (current != second) {
                 current->node = survivor;
                 current = current->next;
            }

            // Inzidenzordnungen der beiden 0-Zellen verbinden und zu löschende
            // Kante herauslösen (die Behandlung des
            // Sonderfalls "0-Zelle vom Grad 1" wurde hier weggelassen)
            first->previous->next  = second->next;
            second->next->previous = first->previous;
            first->next->previous  = second->previous;
            second->previous->next = first->next;

            // Löschen der nicht mehr benötigten Objekte
            deleteNode(second->node);
            deleteEdge(second->edge);
            deleteHalfEdge(first);
            deleteHalfEdge(second);

            return survivor;
        }
    };
```

Diese Ausschnitte aus der Implementation der Klasse `CellComplex` und ihrer Hilfsklassen sollen genügen, um die Prinzipien der graph-basierten Realisierung eines Zellkomplexes zu verdeutlichen. Der vollständige Abdruck der Implementation würde keine weiteren wesentlichen Einsichten vermitteln, so daß wir hier darauf verzichtet haben. Wir wollen an dieser Stelle vor allem deutlich machen, daß wir mit diesem Ansatz in der Lage sind, die in Abschnitt 7.5 definierte Schnittstelle mit relativ geringem Aufwand zu implementieren. Die vorgestellte Implementation ist außerdem sehr schnell: insbesondere gelangen wir stets in konstanter Zeit zur nächsten Kante in der Inzidenz- bzw. Konturordnung, unabhängig von der Größe des Zellkomplexes. Dadurch hängt die Zeit, um beispielsweise alle Nachbarflächen einer gegebenen Fläche zu finden, nur von der jeweiligen Anzahl dieser Nachbarflächen ab, nicht aber von der Gesamtzahl der Flächen.

Auf der anderen Seite benötigt die Implementation mittels Halbkanten relativ viel Speicher: wenn ein Zeiger 4 Byte belegt, benötigt jede Halbkante 24 Byte, jede Kante (2 Halbkanten) bereits 48 Byte. Dazu kommt noch der Speicherbedarf für die Verwaltungsobjekte (je 4 Byte) sowie die Arrays (4 Byte pro Eintrag) und Listen (12 Byte pro Eintrag). Aus der Graphentheorie ist bekannt [WILSON72], daß ein ebener Graph mit n Knoten höchstens $3n$-6 Kanten und $2n$-4 Flächen haben kann, wenn man keine Schleifen und keine Mehrfachkanten zuläßt. Daraus ergibt sich ein Speicherbedarf von maximal ca. 220 Byte pro Knoten, für einen Graphen mit 100 000

Knoten also maximal 22 MByte. In der Praxis wird die erlaubte Maximalzahl der Kanten allerdings selten erreicht, so daß man im typischen Fall deutlich weniger Speicher benötigt. Dennoch können für sehr große Zellkomplexe andere Implementationen notwendig werden.

8.2 Das Zellgitter als Implementation der Khalimsky-Ebene

In diesem Abschnitt wollen wir mit dem Zellgitter eine generische Datenstruktur zur Implementation der Khalimsky-Ebene vorstellen. Die Khalimsky-Ebene ist definiert als Zellkomplex, den man in Form eines quadratischen Gitters in die Ebene einbetten kann. Ohne Beschränkung der Allgemeinheit können wir die 0-Zellen einer Khalimsky-Ebene (bis auf die Außenzelle) auf Punkte abbilden, deren x- und y-Koordinaten ungerade ganze Zahlen sind. Die 1-Zellen verbinden diese Punkte durch gerade Linien, deren Mittelpunkte eine gerade und eine ungerade Koordinate haben (entweder x gerade und y ungerade oder umgekehrt). Den 2-Zellen entsprechen dann die Quadrate, die durch je vier 0- und 1-Zellen eingeschlossen werden. Der Schwerpunkt jedes dieser Quadrate hat gerade Koordinatenwerte.

Aus dieser Einbettung in die Ebene folgt, daß wir eine Khalimsky-Ebene als Bild darstellen können, bei dem wir alle Pixel mit ungeradzahligen Koordinaten als 0-Zellen, mit geradzahligen Koordinaten als 2-Zellen und mit gemischten Koordinaten als 1-Zellen interpretieren. Die Außenzelle muß separat repräsentiert werden. Wir wollen diese Datenstruktur als *Zellgitter* (`Cellgrid`) bezeichnen. Unter der Bezeichnung super grid wurde eine solche Datenstruktur bereits 1970 von Brice und Fennema verwendet [BRICEFEN70]; allerdings waren die weitreichenden topologischen Konsequenzen damals noch nicht bekannt. Das Zellgitter ist eine sehr einfache Realisierung einer Khalimsky-Ebene. Es vereint Eigenschaften einer topologischen und einer ikonischen Repräsentation und bringt dadurch die enge Beziehung zwischen der Khalimsky-Ebene und dem Originalbild sehr gut zum Ausdruck. Wenn das Zellgitter mit 1 Byte pro Pixel implementiert wird, benötigt man gegenüber der graph-basierten Implementation eines Zellkomplexes (Abschnitt 8.1) nur etwa 1/35 des Speichers.[27] Diese Einsparung wird durch die starre Gitterstruktur,

[27] Da das Zellgitter viermal so viele Pixel enthält wie das zugehörige Originalbild, ist der Speicherbedarf immer noch erheblich. Wie [BRAQDOM96] feststellen, reichen für bestimmte Anwendungen 2 Bit pro Pixel aus, so daß man die Größe der Datenstruktur nochmals auf ein Viertel reduzieren kann, indem man Blöcke von 2x2 Pixeln in einem Byte repräsentiert. Allerdings ist in diesem Ansatz der Zugriff auf die Zellen komplizierter, und die Flexibilität der Datenstruktur wird weiter eingeschränkt.

also geringere Flexibilität erkauft. Abbildung 32 zeigt ein Beispiel für die Darstellung einer Khalimsky-Ebene als Zellgitter, wobei der oberen linken 0-Zelle die Koordinate (-1, -1) zugeordnet wurde. Die Außenzelle ist in diesem Zellgitter nicht explizit dargestellt.

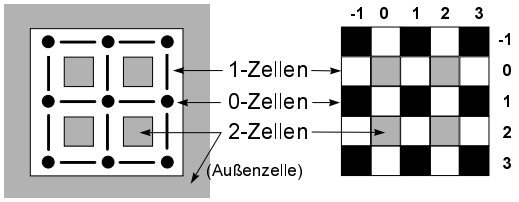


Abbildung 32: Repräsentation einer Khalimsky-Ebene (links) als Zellgitter (rechts). Im Zellgitter sind 0-Zellen schwarz, 1-Zellen weiß und 2-Zellen grau markiert. Die Werte der x-Koordinate sind oben und der y-Koordinate rechts angegeben.

Im folgenden wollen wir die in Abschnitt 7.5 definierte Schnittstelle für eine als Zellgitter repräsentierte Khalimsky-Ebene implementieren. Wir hatten dort zunächst gefordert, daß Iteratoren angeboten werden, mit denen wir sämtliche Zellen als lineare Sequenz abfragen können. Wir können diese Iteratoren beispielsweise als Adapter des `ImageIterator` des Bildes implementieren.

Wir wollen den Iterator für 2-Zellen explizit angeben, die Implementation der Iteratoren für die beiden anderen Zellentypen erfolgt analog. Der `CellgridFaceIterator` kapselt einen `ImageIterator`, der zunächst auf der ersten 2-Zelle (0,0) plaziert wird. Wird der `CellgridFaceIterator` inkrementiert, so wird der eingebettete Iterator zur nächsten 2-Zelle weitergesetzt. Innerhalb einer Zeile wird also jeweils ein Pixel übersprungen, und nach der letzten 2-Zelle einer Zeile muß die ganze nächste Zeile übersprungen werden. Man bezeichnet einen solchen Adapter, der nach einer bestimmten Regel gewisse Elemente einer Sequenz auswählt und andere überspringt, als *Filteriterator*. Eine mögliche Implementation sieht folgendermaßen aus:

```
template <class ImageIterator>
class CellgridFaceIterator
{
    ImageIterator current_, lowerright_;
    int width_;

    // Initialisierung mit dem Rechteck (upperleft, lowerright)
    CellgridFaceIterator(ImageIterator upperleft, ImageIterator lowerright)
    {
        width_ = lowerright.x - upperleft.x;  // Breite des Zellgitters
        current_ = upperleft + Diff2D(1, 1);  // erste 2-Zelle des Zellgitters
        lowerright_ = lowerright;
    }
```

```
    // Inkrementierung
    CellgridFaceIterator & operator++()
    {
        current_.x += 2;    // zur nächsten 2-Zelle (Überspringen der 1-Zelle)

        if(current_.x == lowerright_.x)    // wenn Zeilenende
        {
          current_ += Diff2D(1-width_, 2); // zum Anfang der übernächsten Zeile
        }
    }

    // Vergleich
    bool operator==(CellgridFaceIterator const & o) const {
        return current_ == o.current_;
    }

    // Zugriff auf aktuelle Zelle
    ImageIterator & operator*() { return current_; }

    ... // weitere Operationen
};
```

Wir definieren den Zugriffsoperator `operator*` so, daß er den eingebetteten Iterator auf die aktuelle 2-Zelle zurückgibt. Weitere anwendungsspezifische Daten der aktuellen 2-Zelle kann man mit einem im jeweiligen Kontext geeigneten Zugriffsobjekt (FaceAccessor) abfragen. Die Implementation von `CellgridEdgeAccessor` und `CellgridNodeAccessor` erfolgt analog, mit entsprechend modifizierten Regeln, wieviele Zellen jeweils übersprungen werden müssen, um zur nächsten Zelle des gewünschten Typs zu gelangen.

Ebenso wichtig sind die Zirkulatoren, die die Struktur des Zellkomplexes wiedergeben – Ray Circulator, Contour Nodes Circulator und Contour Edges Circulator. Da die Khalimsky-Ebene eine regelmäßige Gitterstruktur besitzt, haben diese Zirkulatoren bei allen Zellen dasselbe Verhalten: sie müssen stets nacheinander die vier Nachbarn der aktuellen 0- bzw. 2-Zelle referenzieren. Einen ähnlichen Zirkulator, den `Neighborhood4Circulator`, hatten wir bereits in Abschnitt 5.7 beschrieben. Um diesen Zirkulator im neuen Kontext verwenden zu können, müssen wir nur die Funktion `jumpToOpposite()` hinzufügen. Diese Funktion unterscheidet sich von der ursprünglich definierten Funktion `jumpToNeighbor()` dadurch, daß wir den Zirkulator jetzt um das *Doppelte* des aktuellen Differenzvektors verschieben müssen, weil die Nachbarzelle des gleichen Typs immer zwei Pixel entfernt ist. Die Implementation des so angepaßten Zirkulators lautet wie folgt:

```
template <class IMAGEITERATOR>
class CellgridNeighborhood4Circulator
{
    IMAGEITERATOR center_;
    byte direction_;
```

```
  public:
    // Initialisierung mit ImageIterator für das Zentrum und Anfangsrichtung
    CellgridNeighborhood4Circulator(ImageIterator center, int direction = 0)
    : center_(center), direction_(direction)
    {}

    // Inkrementierung des Richtungscodes modulo 8
    CellgridNeighborhood4Circulator & operator++() {
        (direction_ + 2) % 8; return *this;
    }

    // Dekrementierung des Richtungscodes modulo 8
    CellgridNeighborhood4Circulator & operator--() {
        (direction_ + 6) % 8; return *this;
    }

    // Verschiebung des Zentrums zur benachbarten 0- bzw. 2-Zelle
    // und Umkehrung der Richtung (Sprung um den doppelten Differenzvektor!)
    void jumpToOpposite() {
        center_ += (differenceVector() + differenceVector());
        direction_ = (direction_ + 4) % 8;
    }

    // Abfragen des aktuellen Richtungscodes
    int directionCode() const { return direction_; }

    // Dekodierung des Richtungscodes in einen Differenzvektor
    Diff2D differenceVector() const {
        static int dx[] = {1, 1, 0, -1, -1, -1, 0, 1};
        static int dy[] = {0, -1, -1, -1, 0, 1, 1, 1};

        return Diff2D(dx[direction_], dy[direction_]);
    }

    // Abfragen des Zentrums
    ImageIterator center() const { return center_; }

    // Zugriff auf den aktuellen Nachbarn (das ist stets eine 1-Zelle)
    ImageIterator operator*() {
        return (center_ + differenceVector());
    }

    // Vergleich (nur Adapter mit dem gleichen Zentrum dürfen verglichen werden)
    bool operator==( CellgridNeighborhood4Circulator const & o) const {
        return direction_ == o.direction_;
    }
};
```

Wenn wir diesen Zikulator auf einer 0-Zelle plazieren, zeigt er das geforderte Verhalten des Ray Circulator. Plazieren wir ihn hingegen auf einer 2-Zelle, implementiert er den Contour Edges Circulator bzw. den Contour Nodes Circulator: der erste verwendet die geradzahligen Richtungscodes, der zweite hingegen die ungeradzahligen. Das heißt, wenn wir im Konstruktor mit dem Richtungscode 0 initiali-

sieren, erhalten wir den Contour Edges Circulator, bei Initialisierung mit Richtungscode 1 wird das Verhalten des Contour Nodes Circulator realisiert:

```
typedef CellgridNeighborhood4Circulator<SomeImageIterator> CellgridRayCirculator;

typedef CellgridNeighborhood4Circulator<SomeImageIterator>
                                                    CellgridContourEdgesCirculator;

typedef CellgridNeighborhood4Circulator<SomeImageIterator>
                                                    CellgridContourNodesCirculator;
```

Es fehlen nun noch der Boundary Components Iterator, der Boundary Edges Iterator und der Face Adjacency Iterator. Der Boundary Components Iterator für ein Zellgitter ist trivial, weil jede 2-Zelle genau eine Kontur hat. Für diese Kontur muß er den auf der entsprechenden 2-Zelle plazierten `CellgridNeighborhood4Circulator` zurückgeben:

```
template <class ImageIterator>
struct CellgridBoundaryComponentsIterator
{
    typedef
        CellgridNeighborhood4Circulator<ImageIterator> ContourEdgesCirculator;

    ContourEdgesCirculator contour;
    bool is_end;

    // Initialisierung für gegebene 2-Zelle
    CellgridBoundaryComponentsIterator(ImageIterator face)
    : contour(face),
      is_end(false)
    {}

    // Inkrementierung
    CellgridBoundaryContourIterator & operator++()
    {
        is_end = true;   // Sequenz hat die Länge 1 => Ende sofort erreicht
    }

    // Vergleich
    bool operator==( CellgridBoundaryContourIterator const & o) const
    {
        return is_end == o.is_end;
    }

    // Zugriff: Zurückgeben des Konturzirkulators
    ContourEdgesCirculator & operator*() { return contour; }
};
```

Da jede 2-Zelle nur eine Kontur und genau vier verschiedene benachbarte 2-Zellen hat, sind Boundary Edges Iterator und Face Adjacency Iterator in einem Zellgitter

identisch. Ihr Verhalten entspricht darüber hinaus fast dem Verhalten des Contour Edges Circulator, mit dem Unterschied, daß es sich um Iteratoren handelt und nicht um Zirkulatoren. Wir hatten in Abschnitt 5.7.1 beschrieben, wie man mit Hilfe eines Adapters einen Zirkulator in einen Iterator umwandeln kann. Das heißt, wir können Boundary Edges Iterator und Face Adjacency Iterator für das Zellgitter als `CirculatorAdapter` des `CellgridNeighborhood4Circulator` implementieren.

Wir haben damit gezeigt, daß ein Zellgitter hinsichtlich der geforderten Iteratoren und Zirkulatoren die Anforderungen an einen Zellkomplex erfüllt. Auch die Zugriffsobjekte lassen sich problemlos implementieren. Wir zeigen hier als Beispiel einen Ausschnitt aus dem `FaceAccessor`:

```
template <class ImageIterator>
struct FaceAccessor
{
    typedef CellgridFaceIterator<ImageIterator>     FaceIterator;
    typedef CellgridBoundaryComponentsIterator<ImageIterator>
                                                    BoundaryComponentsIterator;
    typedef CellgridNeighborhood4Circulator<ImageIterator>
                                                    ContourEdgesCirculator;
    typedef CirculatorAdapter< ContourEdgesCirculator> BoundaryEdgesIterator;

    // Erzeugen der zur 2-Zelle gehörenden BoundaryComponentsIteratoren
    BoundaryComponentsIterator boundaryComponentsIteratorBegin(FaceIterator i)
    {
        return BoundaryComponentsIterator(*i);
    }

    BoundaryComponentsIterator boundaryComponentsIteratorEnd(FaceIterator i)
    {
        return ++boundaryComponentsIteratorBegin(i);
    }

    // Erzeugen der zur 2-Zelle gehörenden BoundaryEdgesIteratoren
    BoundaryEdgesIterator boundaryEdgesIteratorBegin(FaceIterator i)
    {
        BoundaryEdgesIterator(*boundaryComponentsIteratorBegin(i), 0);
    }

    BoundaryEdgesIterator boundaryEdgesIteratorEnd(FaceIterator i)
    {
        BoundaryEdgesIterator(*boundaryComponentsIteratorBegin(i), 1);
    }

    // Erzeugen eines ContourEdgesCirculator für die aktuelle Kontur
    ContourEdgesCirculator contourEdgesCirculator(BoundaryComponentsIterator i)
    {
        return *i;
    }

    ... // weitere Funktionen
};
```

Dieser Ausschnitt illustriert, daß auch die Implementation der Zugriffsobjekte keine Schwierigkeiten bereitet. Wir wollen dies hier nicht weiter ausführen, da es sich dabei um eine reine Fleißarbeit handelt und sich keine interessanten neuen Einsichten ergeben.

8.3 Segmentierte Zellkomplexe auf Basis des Zellgitters

Nachdem wir im vorigen Abschnitt das Zellgitter eingeführt haben, wollen wir nun beschreiben, wie man auf dem Zellgitter eine Segmentierung darstellen kann. Es geht dabei zunächst nur um die Entwicklung einer Datenstruktur, mit der man ein solches segmentiertes Zellgitter darstellen und die enthaltenen Informationen abfragen kann. Algorithmen zur Erzeugung eines segmentierten Zellgitters behandeln wir in Abschnitt 8.4.

Zunächst müssen wir festlegen, wie man für jede Zelle erkennt, ob sie zu einer Region, einer Linie oder einem Vertex gehört. Um dabei nicht eine bestimmte Implementationstechnik zu erzwingen, verwenden wir dazu ein Zugriffsobjekt. Wenn wir diesem Zugriffsobjekt einen Iterator übergeben, der auf eine Zelle (ein Pixel) in einem Zellgitter verweist, erhalten wir als Antwort, zu welcher Art von Untermenge die betreffende Zelle gehört:

```
template <class ImageIterator>
struct SegmentedCellgridAccessor
{
    int label(ImageIterator);
    bool isRegion(ImageIterator);
    bool isLine(ImageIterator);
    bool isVertex(ImageIterator);
};
```

Zusätzlich enthält dieses Objekt eine Funktion `label()`, die für jede Untermenge eine eindeutige Identifikationsnummer (engl. label) zurückgibt. Alle Zellen, die zur selben Untermenge gehören, haben die gleiche Identifikationsnummer. Dies entspricht dem üblichen Vorgehen, Regionen in einem Regionenbild durch Nummern zu kennzeichnen, nur daß wir hier drei verschiedene Arten von Untermengen betrachten. Wir wollen ein so markiertes Zellgitter als *segmentiertes Zellgitter* (`SegmentedCellgrid`) bezeichnen. Damit bei den labels kein Überlauf auftritt, muß man ein segmentiertes Zellgitter normalerweise mit 2 oder 4 Byte pro Pixel implementieren, wobei zwei Bits den Untermengentyp und die übrigen das label kodieren.

Die Zuordnung der Zellen zu Untermengen allein ist jedoch noch nicht sehr nützlich. Wichtig ist vielmehr, daß wir auch die Struktur des segmentierten Zellkomplexes analysieren können. Typische Fragestellungen wären zum Beispiel, welche Regionen einander benachbart sind, welche Linien zur Kontur einer Region gehören, ob eine Region Löcher hat usw. Um solche Fragen zu beantworten, ist es am einfachsten, den segmentierten Zellkomplex zunächst zu kontrahieren. Die gesuchten topologischen Informationen sind dann direkt in der Struktur des kontrahierten Zellkomplexes kodiert und werden über dessen normale Schnittstelle (also mit Hilfe der Iteratoren und Zugriffsobjekte aus Abschnitt 7.5) abgefragt. Dieses Vorgehen hat den großen Vorteil, daß man keine neue Schnittstelle für die Analyse der Struktur der Segmentierung definieren muß.

In einem Zellkomplex, der als Graph implementiert wurde, kann die Kontraktion mit Hilfe der Euler-Operatoren sehr einfach realisiert werden. Bei einem segmentierten Zellgitter hingegen ist eine regelmäßige Gitterstruktur fest vorgegeben. Die Verschmelzung benachbarter Zellen ist in dieser starren Datenstruktur nicht möglich, d.h. wir können die Euler-Operatoren nicht direkt implementieren. Um diese Schwierigkeit zu vermeiden, könnten wir das Zellgitter zuerst in einen Graphen umwandeln, so daß wir nicht mehr an die regelmäßige Struktur gebunden sind. Dieses Vorgehen wäre jedoch sehr ineffizient.

Zu einer besseren Lösung gelangen wir, wenn wir uns klarmachen, daß wir nur die *Schnittstelle* des kontrahierten Zellkomplexes benötigen. Welche Implementation hinter dieser Schnittstelle steht, ist unerheblich. Dies führt uns zu der Idee, daß die Kontraktion nicht physisch ausgeführt werden muß, solange wir die notwendigen Iteratoren und Zugriffsobjekte des kontrahierten Zellkomplexes auf andere Weise implementieren können. Wir wollen im folgenden einen Weg dafür aufzeigen.

Die Aufgabe besteht also darin, einen kontrahierten Zellkomplex auf der Basis des segmentierten Zellgitters zu implementieren. Wir wollen diese Datenstruktur kürzer *kontrahiertes Zellgitter* (`ContractedCellgrid`) nennen. Ein kontrahiertes Zellgitter behandelt die Regionen, Linien und Vertizes in einem segmentierten Zellgitter so, als wären sie einzelne 2-, 1- bzw. 0-Zellen. Die Iteratoren und Zugriffsobjekte des kontrahierten Zellgitters realisieren die Schnittstelle eines unregelmäßigen Zellkomplexes, obwohl sie intern auf einem regelmäßigen Zellgitter aufsetzen.

Wir beginnen mit der Implementation der Iteratoren für die reine Aufzählung aller 0-, 1- bzw. 2-Zellen. Dazu definieren wir wiederum interne Objekte zur Verwaltung der Zellen jeden Typs. Diese Objekte speichern die Identifikationsnummer der jeweiligen Untermenge der Segmentierung, so daß die korrespondierenden Pixel des Zellgitters eindeutig zugeordnet werden können:

```
struct ContractedCellgridFace
{
    int label;   // ID der korrespondierenden Region
};
```

```
struct ContractedCellgridEdge
{
    int label;   // ID der korrespondierenden Linie
};

struct ContractedCellgridNode
{
    int label; // ID des korrespondierenden Vertex
};
```

Für jede dieser Klassen definieren wir ein Array (vector aus der C++-Standardbibliothek), die sämtliche Objektinstanzen des betreffenden Untermengentyps enthält. Die Iteratoren dieser Listen erfüllen alle Anforderungen an einen Face Iterator, Edge Iterator bzw. Node Iterator für den kontrahierten Zellkomplex:

```
typedef vector<ContractedCellgridFace>::iterator  ContractedCellgridFaceIterator;
typedef vector<ContractedCellgridEdge>::iterator  ContractedCellgridEdgeIterator;
typedef vector<ContractedCellgridNode>::iterator  ContractedCellgridNodeIterator;
```

Die Implementation des Ray Circulator ist etwas schwieriger. Um das Verständnis zu erleichtern, wollen wir zunächst eine eingeschränkte Klasse von segmentierten Zellgittern betrachten, für die es einen einfachen Ray Circulator gibt. Danach kehren wir zum allgemeinen Fall zurück.

Wir nehmen im folgenden an, daß sämtliche Vertizes und Linien der Segmentierung aus nur einer 0- bzw. 1-Zelle bestehen. Mit anderen Worten, jede 0- und 1-Zelle, die nicht einer Region zugeordnet wurde, bildet einen Vertex bzw. eine Linie. Regionen können jedoch weiterhin beliebig komplex sein. Diese Einschränkung hat den Vorteil, daß es in der Umgebung eines Vertex bis auf Rotation nur sechs verschiedene Konfigurationen geben kann, die in Abbildung 33 gezeigt sind. Da es sich dabei um topologische Konfigurationen in der Khalimsky-Ebene handelt, sind diese Konfigurationen auch nicht vom Maßstab oder der Auflösung eines Bildes abhängig.

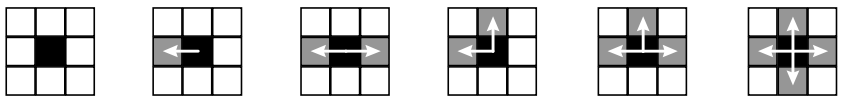


Abbildung 33: Mögliche Konfigurationen (bis auf Rotation) in der Umgebung eines Vertex der eingeschränkten Klasse von segmentierten Zellgittern. (a) isolierter Punkt, (b) Endpunkt, (c) Durchgangspunkt, (d) Eckpunkt, (e) und (f) Kreuzungspunkte vom Grad 3 bzw. 4. Der Vertex ist schwarz, Regionen sind weiß und Linien grau markiert, die Pfeile deuten die möglichen Positionen des Ray Circulator für den Vertex an.

Durch die Einschränkung auf diese sechs Konfigurationen kann der Ray Circulator sehr einfach als Adapter auf der Basis des `CellgridNeighborhood4Circulator` des Zellgitters implementiert werden. Unter den vier möglichen Richtungen, die der eingebettete Zirkulator einnehmen kann, wählt der Adapter mit Hilfe des `SegmentedCellgridAccessor` diejenigen aus, bei denen die inzidente 1-Zelle als Linie markiert wurde. Die übrigen Richtungen werden übersprungen. Bei einem isolierten Punkt gibt es keine inzidenten Linien, der Zirkulator ist in diesem Falle singulär:

```
template <class ImageIterator>
struct SimpleContractedCellgridRayCirculator
{
    typedef
       CellgridNeighborhood4Circulator<ImageIterator> NeighborhoodCirculator;
    typedef SegmentedCellgridAccessor<ImageIterator> Accessor;

    NeighborhoodCirculator adaptee_;
    bool is_singular;
    Accessor cellinfo;

    // Initialisierung mit Ray Circulator des unterliegenden Zellgitters
    SimpleContractedCellgridRayCirculator(NeighborhoodCirculator c)
    : adaptee_(c),
      is_singular(false)
    {
        // finden der ersten Richtung, für die die 1-Zelle eine Linie ist
        int i=0;
        for(; !cellinfo.isLine(*adaptee_) && i < 4; ++i) ++adaptee_;

        if(i == 4) is_singular = true;  // keine Linie gefunden
    }

    bool isSingular() const
    {
        return is_singular;
    }

    // Inkrementierung
    SimpleContractedCellgridRayCirculator & operator++()
    {
        if(!isSingular())
        {
            // Weiterrücken bis zur nächsten Linie
            // (Überspringen der 1-Zellen, die keine Linien sind)
            do { ++adaptee_; } while(!cellinfo.isLine(*adaptee_));
        }
        return *this;
    }
```

```
        // Dekrementierung
        SimpleContractedCellgridRayCirculator & operator--()
        {
             if(!isSingular())
             {
                  // rückwärts Weiterrücken bis zur vorigen Linie
                  // (Überspringen der 1-Zellen, die keine Linien sind)
                  do { --adaptee_; } while(!cellinfo.isLine(*adaptee_));
             }
             return *this;
        }

        void jumpToOpposite() {
             if(!isSingular()) adaptee_.jumpToOpposite();
        }

        // Zugriff auf aktuellen Vertex wird an adaptierten Zirkulator delegiert
        ImageIterator vertex() { return adaptee_.center(); }

        // Zugriff auf aktuelle Linie wird an adaptierten Zirkulator delegiert
        ImageIterator line() { return *adaptee_; }

        ... // weitere Funktionen
    };
```

Bevor wir jetzt wieder zum allgemeinen Fall, also zu beliebigen segmentierten Zellgittern, zurückkehren, wollen wir noch einmal an einem Beispiel demonstrieren, warum man nicht mit Vertizes auskommt, die aus nur einer 0-Zelle bestehen. Das Problem der Vereinfachung besteht darin, daß ein Vertex dann *höchstens vier Linien* begrenzen kann. In der Praxis gibt es aber Objekte, deren Repräsentation Vertizes (d.h. Kreuzungspunkte von Kanten) von höherem Grad erfordert. Abbildung 34 zeigt links ein Objekt, dessen Zentrum ein Vertex vom Grad sechs bildet. Dies kann in einer vereinfachten Segmentierung nicht dargestellt werden.

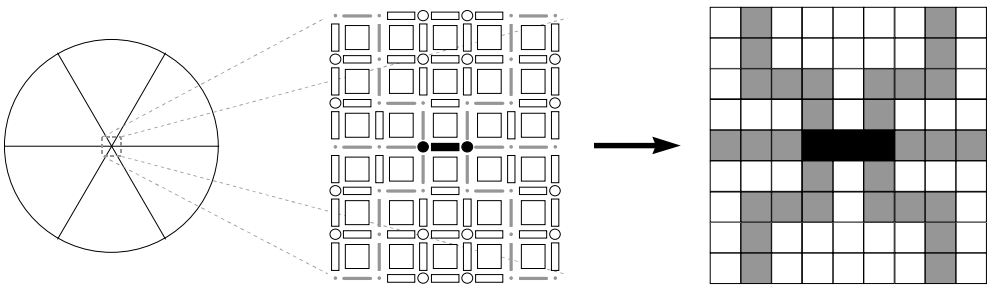


Abbildung 34: links: Knoten vom Grad 6, Mitte: Ausschnittsvergrößerung der Umgebung diese Knotens in der Khalimsky-Ebene (idealisiert), rechts: Repräsentation der Khalimsky-Ebene als segmentiertes Zellgitter (Regionen sind weiß, Linien grau, der Vertex schwarz dargestellt)

Die allgemeine Definition der Zellsegmentierung erlaubt nun die Zusammenfassung mehrerer benachbarter 0-Zellen (mit den dazwischen liegenden 1-Zellen) zu einem einzigen Vertex. Ein solcher „ausgedehnter" Vertex kann beliebigen Grad haben, auch wenn der Grad der einzelnen 0-Zellen, aus denen er besteht, beschränkt ist. Abbildung 34 Mitte und rechts zeigt, wie der Knoten vom Grad sechs in der Khalimsky-Ebene bzw. in einem Zellgitter durch Zusammenfassen von *zwei* benachbarten 0-Zellen zu einem Vertex dargestellt wird. Es mag zunächst befremdlich erscheinen, daß ein Vertex im Zellgitter eine Ausdehnung besitzen kann. Es sei jedoch daran erinnert, daß es sich hier um eine topologische Repräsentation handelt, bei der die geometrische Veranschaulichung beliebig gewählt werden kann, solange sie die notwendigen topologischen Eigenschaften besitzt. Dies ist hier gewährleistet, denn wir erhalten stets einen korrekten Zellkomplex, wenn wir ausgedehnte Vertizes kontrahieren.

Der Ray Circulator eines kontrahierten Zellgitters muß folglich in der Lage sein, einen ausgedehnten Vertex so zu behandeln, als ob er kontrahiert worden wäre, ohne daß wir die Kontraktion tatsächlich ausführen können. Abbildung 35 zeigt die möglichen Positionen des Ray Circulator für einen Vertex vom Grad sechs, wie er bei der in Abbildung 34 gezeigten Konfiguration auftritt, sowie für einen Vertex vom Grad fünf.

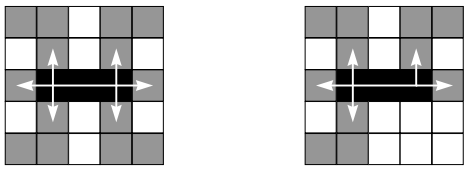


Abbildung 35: Mögliche Positionen des Ray Circulator an einem Vertex, der aus zwei 0-Zellen und einer 1-Zelle besteht, links: Grad 6 (vergl. Abbildung 34), rechts: Grad 5

Wie im vereinfachten Fall zeigt der Ray Circulator nacheinander auf jede Linie, die von dem betreffenden Vertex begrenzt wird, aber die Linien treffen sich nicht mehr alle in derselben 0-Zelle (in demselben Pixel) des unterliegenden Zellgitters. Dennoch können wir auch diesen Ray Circulator als Adapter auf der Basis des `CellgridNeighborhood4Circulator` implementieren. Es genügt jetzt aber nicht mehr, diesen adaptierten Zirkulator nur zu drehen, sondern wir müssen ihn auch zwischen den 0-Zellen des Vertex verschieben. Dies kann mit dem folgenden Algorithmus realisiert werden:

Inkrementieren des Ray Circulator für einen Knoten des kontrahierten Zellgitters:

0) Voraussetzung: der Ray Circulator steht auf einer gültigen Position
1) Der eingebettete `CellgridNeighborhood4Circulator` wird inkrementiert.

2) Zu welcher Art von Untermenge gehört die jetzt referenzierte 1-Zelle ?
   2.1) die 1-Zelle gehört zu einer Linie => nächste Position gefunden, Algorithmus beendet
   2.2) die 1-Zelle gehört zu einer Region => gehe zu 1)
   2.3) die 1-Zelle gehört zum Vertex => Aufrufen der Funktion `jumpToOpposite()`, dann gehe zu 1)

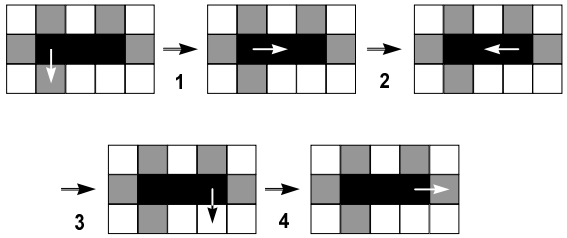


Abbildung 36: Zwischenschritte, die der `CellgridNeighborhood4Circulator`, der in einen `ContractedCellgridRayCirculator` eingebettet ist, ausführt, um von einer gültigen Position (erstes Bild) des Ray Circulator zur nächsten (letzes Bild) zu gelangen

Für den in Abbildung 36 dargestellten Vertex benötigt der Algorithmus von der gezeigten Ausgangsposition aus beispielsweise vier Schritte, um den Ray Circulator einmal zu inkrementieren:

Voraussetzung: der Zirkulator referenziert eine Linie, die durch den Vertex begrenzt wird (erstes Bild)

Schritt 1: Inkrementieren des eingebetteten `CellgridNeighborhood4Circulator`. Er referenziert nun eine 1-Zelle, die zum Vertex gehört (zweites Bild).

Schritt 2: Ausführen von `jumpToOpposite()` (drittes Bild)

Schritt 3: Inkrementieren des eingebetteten `CellgridNeighborhood4Circulator`. Er referenziert nun eine 1-Zelle, die zu einer Region gehört (viertes Bild).

Schritt 4: Inkrementieren des eingebetteten `CellgridNeighborhood4Circulator`.

Ergebnis: Der Zirkulator referenziert nun die 1-Zelle, die zur nächsten Linie der Inzidenzordnung gehört (letztes Bild in Abbildung 36).

Aus obigem Algorithmus ergibt sich folgende Implementation des `ContractedCellgridRayCirculator` (die Implementation enthält den Algorithmus zweimal: einmal in originaler Form in `operator++` und einmal mit einer kleinen Modifikation zur Detektion isolierter Vertizes im Konstruktor):

```
template <class ImageIterator>
struct ContractedCellgridRayCirculator
{
    typedef
       CellgridNeighborhood4Circulator<ImageIterator> NeighborhoodCirculator;
    typedef CellSegmentedCellgridAccessor<ImageIterator> Accessor;

    NeighborhoodCirculator adaptee_;
    bool is_singular;
    Accessor cellinfo;

    // Initialisierung mit Ray Circulator des unterliegenden Zellgitters
    ContractedCellgridRayCirculator(NeighborhoodCirculator c)
    : adaptee_(c),
      is_singular(false)
    {
        // Finden der ersten Richtung, für die die 1-Zelle eine Linie ist
        if(cellinfo.isLine(*adaptee_)) return;   // steht bereits richtig
        for(; ;)
        {
            ++adaptee_;
            if(adaptee_ == c)
            {
                // voller Umlauf ohne Linie zu finden => isolierter Punkt
                is_singular = true;
                return;
            }

            if(cellinfo.isLine(*adaptee_)) return;  // Linie gefunden

            // innere 1-Zelle des Vertex gefunden =>
            //                       springe zur benachbarten 0-Zelle
            if(cellinfo.isVertex(*adaptee_)) adaptee_.jumpToOpposite();
        }
    }

    bool isSingular() const { return is_singular; }

    // Inkrementierung
    ContractedCellgridRayCirculator & operator++()
    {
        if(!isSingular())
        {
            // Weiterrücken bis zur nächsten Linie
            for(;;)
            {
                ++adaptee_;
                if(cellinfo.isLine(*adaptee_)) break;  // Linie gefunden

                // innere 1-Zelle des Vertex gefunden =>
                //                       springe zur benachbarten 0-Zelle
                if(cellinfo.isVertex(*adaptee_)) adaptee_.jumpToOpposite();
            }
```

```
            }
            return *this;
        }

        ... // weitere Funktionen
    };
```

Die zweite wesentliche Erweiterung bei beliebig segmentierten Zellgittern gegenüber der vereinfachten Segmentierung betrifft die Linien. Auch Linien können jetzt beliebig viele 0- und 1-Zellen enthalten und müssen nicht mehr aus einer einzelnen 1-Zelle bestehen. Diese Erweiterung beeinflußt insbesondere die Implementation der Funktion `jumpToOpposite()` des `ContractedCellgridRayCirculator`. Wir können in dieser Funktion nicht mehr einfach zur benachbarten 0-Zelle der gerade referenzierten 1-Zelle springen, denn diese 0-Zelle könnte ebenfalls zur Linie gehören. Der Endvertex einer Linie ist vom Anfangsvertex aus im allgemeinen nicht in einem Schritt zu erreichen. Auch dieses Problem kann jedoch sehr einfach gelöst werden, denn es entspricht weitgehend dem Problem der Konturverfolgung. Der einzige Unterschied besteht darin, daß bei der Konturverfolgung sämtliche 0- und 1-Zellen einer Kontur besucht werden, während wir jetzt nur bis zur nächsten als Vertex markierten 0-Zelle gehen müssen. Die Implementation der Funktion `ContractedCellgridRayCirculator::jumpToOpposite()` erfolgt daher folgendermaßen:

```
template <class ImageIterator>
void ContractedCellgridRayCirculator::jumpToOpposite()
{
    if(!isSingular())
    {
        // Verfolgung der aktuellen Linie,
        //               bis gegenüberliegender Vertex erreicht
        for(;;)
        {
            adaptee_.jumpToOpposite();

            // Endvertex gefunden => Schleife beenden
            if(cellinfo.isVertex(adaptee_.center())) break;

            // innere 0-Zelle der Linie gefunden => Linie weiterverfolgen
            do { ++adaptee_; } while(!cellinfo.isLine(*adaptee_));
        }
    }
}
```

Damit ist die Implementation des Ray Circulator komplett. Einen solchen Ray Circulator speichern wir nun in jedem `ContractedCellgridNode`-Objekt, um schnell auf jeden Vertex zugreifen zu können:

```
struct ContractedCellgridVertex {
    int label;
    ContractedCellgridRayCirculator<ImageIterator>  rayCirculator;

    ... // weitere Informationen
};
```

Vom Standpunkt des Ressourcenverbrauchs ist die explizite Speicherung dieser Zirkulatoren unproblematisch: ein segmentiertes Zellgitter auf der Basis eines 256x256 Pixel großen Originalbildes enthält normalerweise höchstens einige tausend Vertizes. Mit der beschriebenen Implementation benötigt ein Ray Circulator 10 Byte, so daß wir typischerweise weniger als 50 kByte zusätzlichen Speicher benötigen. Dies ist wenig im Vergleich zum unterliegenden segmentierten Zellgitter, welches mindestens 513x513x2 Byte, also ca. 510 kBytes belegt.

Auf der Basis des `ContractedCellgridRayCirculator` können wir nun einen weiteren Adapter definieren, der das Verhalten des Contour Nodes Circulator für eine Region des segmentierten Zellgitters, d.h. für eine 2-Zelle des kontrahierten Zellgitters, implementiert. Erneut werden hier Iteratoren mehrfach ineinander eingebettet, um ein komplexes Verhalten zu realisieren (Abbildung 37).

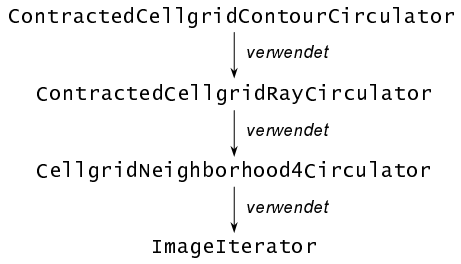


Abbildung 37: Schichtung der Iteratoren und Zirkulatoren des kontrahierten Zellgitters

Das Verhalten des `ContractedCellgridContourCirculator` ist, bis auf den Datenzugriff, identisch mit dem des `CellComplexContourCirculator` aus Abschnitt 8.1. Dies gilt übrigens ganz allgemein: hat man für einen Zellkomplex den Ray Circulator implementiert, können die meisten anderen Iteratoren als Adapter auf dessen Basis realisiert werden. Der `ContractedCellgridContourCirculator` verwendet ebenfalls den auf Seite 212 beschriebenen Algorithmus „Konturverfolgung in Zellkomplexen“ . Dieser Algorithmus wird in der folgenden Implementation zweimal verwendet: einmal in originaler Gestalt in der Inkrementierungsoperation `operator++` und zum zweiten Mal mit umgekehrter Reihenfolge der Schritte im Dekrementierungsoperator `operator--`:

```
template <class ImageIterator>
struct ContractedCellgridContourCirculator
{
    typedef ContractedCellgridRayCirculator<ImageIterator> RayCirculator;

    RayCirculator adaptee_;

    // Initialisierung mit einem RayCirculator auf einem Vertex der Kontur
    ContractedCellgridContourCirculator(RayCirculator c)
    : adaptee_(c)
    {}

    // Inkrementierung
    ContractedCellgridContourCirculator & operator++()
    {
        // Weitergehen zum nächsten Vertex
        adaptee_.jumpToOpposite();

        // RayCirculator auf die nächste Linie der Kontur drehen
        --adaptee_;
        return *this;
    }

    // Dekrementierung
    ContractedCellgridContourCirculator & operator--()
    {
        // RayCirculator auf die vorherige Linie der Kontur drehen
        ++adaptee_;

        // Zurückgehen zum vorigen Vertex
        adaptee_.jumpToOpposite();
        return *this;
    }

    // Zugriff auf aktuellen Vertex
    ImageIterator vertex() { return adaptee_.vertex(); }

    // Zugriff auf aktuelle Linie
    ImageIterator line()  { return adaptee_.line(); }

    ... // weitere Funktionen
};
```

Dieser Zirkulator kann sowohl als Contour Nodes Circulator als auch als Contour Edges Circulator verwendet werden, weil einerseits die zu den Vertizes gehörenden Informationen mit Hilfe der Funktion `vertex()` und andererseits die zu den Linien gehörenden durch `line()` abgefragt werden können. Die Aufrufe dieser Funktionen werden natürlich in einem `NodeAtStartAccessor` bzw. `EdgeAccessor` gekapselt. Ein Beispiel für den Weg des `ContractedCellgridContourCirculator` beim Umlaufen einer Region in einem segmentierten Zellgitter ist in Abbildung 38 gezeigt.

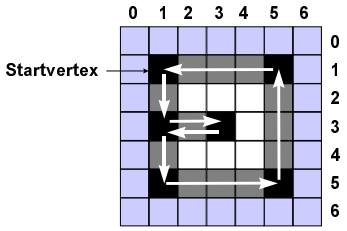


Abbildung 38: Weg eines `ContractedCellgridContourCirculator` beim Verfolgen der äußeren Kontur der weißen Region vom angegeben Startvertex aus (Vertizes sind schwarz, Linien dunkelgrau und Regionen hellgrau bzw. weiß markiert)

Die Implementation der übrigen Iteratoren erfolgt genau wie bei der graph-basierten Implementation von Zellkomplexen in Abschnitt 8.1. Wir wollen deshalb hier nicht weiter darauf eingehen.

Zum Schluß wollen wir noch einige Bemerkungen zur Implementation von Euler-Operatoren auf Zellgittern machen. Wir hatten weiter oben festgestellt, daß die starre Struktur eines Zellgitters die physische Verschmelzung von Zellen nicht zuläßt. Die Möglichkeit, dennoch Iteratoren für das kontrahierte Zellgitter zu realisieren, zeigt jedoch, daß die physische Verschmelzung gar nicht notwendig ist. Es reicht aus, wenn die Zellen entsprechend ihrer Zugehörigkeit zu Untermengen (Regionen, Linien, Vertizes) *markiert* werden.

Das heißt, wir können Euler-Operatoren auf segmentierten Zellgittern realisieren, indem wir die Markierung von Zellen ändern. Betrachten wir beispielsweise die Operation „Verschmelzen benachbarter 2-Zellen“. 2-Zellen des kontrahierten Zellgitters korrespondieren zu Regionen des segmentierten Zellgitters. Die labels der zu verschmelzenden Regionen seien r_1 und r_2, die dazwischen liegende Linie habe das label l. Dann können wir die Regionen verschmelzen, indem wir einfach allen Pixeln des Zellgitters, die label r_2 oder l haben, das neue label r_1 zuweisen. Außerdem muß den Pixeln der Linie der neue Untermengentyp „Region“ zugewiesen werden.[28] Auf die gleiche Weise werden die übrigen Euler-Operatoren implementiert. Es zeigt sich also, daß auch das kontrahierte Zellgitter alle Anforderungen an einen Zellkomplex erfüllt, die wir in Abschnitt 7.5 definiert hatten.

[28] Natürlich muß man außerdem die interne Information zur Verwaltung des kontrahierten Zellgitters aktualisieren. Dies ist jedoch unproblematisch.

8.4 Segmentierungsalgorithmen für Zellkomplexe

Zum Abschluß dieser Arbeit wollen wir nun noch einige Algorithmen angeben, die Zellkomplexe verwenden bzw. erzeugen. Da es sich dabei um einen aktuellen Forschungsgegenstand handelt, können wir hier nur exemplarisch einige grundsätzliche Ideen andeuten. Dabei wollen wir erstens zeigen, wie man traditionelle Segmentierungsverfahren (siehe Kapitel 6) so modifizieren kann, daß sie eine Segmentierung der Khalimsky-Ebene liefern. Zweitens wollen wir einen Segmentierungsalgorithmus vorstellen, den man iterativ auf Zellkomplexe anwenden kann, um zu einer Zellpyramide zu gelangen.

8.4.1 Überführen eines Regionenbildes in ein segmentiertes Zellgitter

Wir hatten in Abschnitt 7.1 beschrieben, daß traditionelle ikonische Repräsentationen unter topologischen Inkonsistenzen leiden, die deren Weiterverarbeitung erschweren. Bei Zellkomplexen hingegen treten solche Inkonsistenzen nicht auf. Es liegt deshalb nahe, traditionelle Segmentierungsalgorithmen so zu modifizieren, daß sie ihre Ergebnisse in Form eines Zellkomplexes darstellen. Diese Modifikation bewirkt nicht nur, daß die Segmentierungsergebnisse topologisch korrekt repräsentiert werden, sondern sie führt auch zu einer austauschbaren, einheitlichen Beschreibung der Ergebnisse. Bei bisherigen Algorithmen war dies nicht der Fall: die Vielfalt ikonischer Repräsentationen ist groß, und auch Konvertierungen sind nicht immer möglich. Auch in Kapitel 6 wurde diese Vielfalt deutlich: je nach Algorithmus hatten wir Binärbilder, Kantenbilder oder Regionenbilder mit und ohne Kanten erhalten.

Es zeigt sich nun, daß man ein Regionenbild sehr einfach in ein segmentiertes Zellgitter überführen kann, wenn die Regionen mit Hilfe der 4-Nachbarschaft definiert sind. Das gleiche gilt auch für Binärbilder, denn man kann diese durch Markierung der Zusammenhangskomponenten in Regionenbilder transformieren (um den folgenden Algorithmus anwenden zu können, muß diese Transformation allerdings nicht explizit ausgeführt werden). Wir nehmen also an, daß die Pixel des segmentierten Ausgangsbildes so markiert sind, daß zwei horizontal oder vertikal benachbarte Pixel genau dann zur selben Region gehören, wenn sie den gleichen Wert haben. Wir ordnen nun einem Ausgangsbild der Größe $w \times h$ ein Zellgitter der Größe $(2w+1) \times (2h+1)$ zu, wobei die 2-Zelle am Ort $(2x, 2y)$ jeweils zum Pixel am Ort

(x, y) korrespondiert. Der folgende Algorithmus liefert dann ein segmentiertes Zellgitter, dessen Regionen denen des Ausgangsbildes entsprechen:

Transformation eines Regionenbildes in ein segmentiertes Zellgitter:

1) Alle 2-Zellen des Zellgitters werden als „Region" markiert.
2) Für jede 1-Zelle des Zellgitters:
   2.1) Wenn die 1-Zelle zum äußeren Rand des Zellgitters gehört, wird sie als „Linie" markiert.
   2.2) Wenn die 1-Zelle zwei 2-Zellen begrenzt, deren korrespondierende Pixel gleiche Werte haben, wird sie als „Region" markiert.
   2.3) Andernfalls wird die 1-Zelle als „Linie" markiert.
3) Für jede 0-Zelle des Zellgitters:
   3.1) Wenn die 0-Zelle zum äußeren Rand des Zellgitters gehört, wird sie als „Vertex" markiert.
   3.2) Wenn die 0-Zelle nur 1-Zellen begrenzt, die als „Region" markiert wurden, wird sie als „Region" markiert.
   3.3) Andernfalls wird die 0-Zelle als „Vertex" markiert.
4) Durch Markieren der Zusammenhangskomponenten des Zellgitters werden allen Untermengen (Regionen, Linien und Vertizes) eindeutige label zugeordnet.

Anschaulich ausgedrückt, bildet dieser Algorithmus die Regionen auf das Zellgitter ab und fügt dazwischen Linien und Vertizes ein. In Abbildung 39 wird diese Vorgehensweise an einem einfachen Beispiel mit nur zwei Regionen illustriert. Zur besseren Veranschaulichung werden in der Abbildung die label der Linien und Vertizes nicht dargestellt.

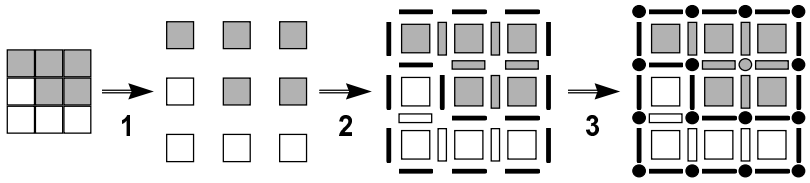


Abbildung 39: Überführung eines Regionenbildes (links) in ein segmentiertes Zellgitter (rechts). Schritt 1: Markieren der 2-Zellen, Schritt 2: Markieren der 1-Zellen, Schritt 3: Markieren der 0-Zellen (Regionen sind weiß und grau, Linien und Vertizes schwarz markiert)

Den obigen Algorithmus können wir nun als Nachbearbeitung auf die Ergebnisse der Schwellwertbildung (Abschnitt 6.1) und des Regionenwachstums aufgrund statistischer Merkmale (Abschnitt 6.3.2) anwenden. Auch die Modifikation des Wasserscheidenalgorithmus (Abschnitt 6.3.1) ist einfach möglich: um hier ein

Regionenbild zu erhalten, müssen wir nur das Markieren der Wasserscheiden weglassen. Dies geschieht durch eine einfache Änderung im `WatershedMergeFunctor` aus Abschnitt 6.3.1:

```
struct WatershedMergeFunctor // ohne Markieren der Wasserscheiden
{
    int watershed_label;

    WatershedMergeFunctor(int ws_label) : watershed_label(ws_label) {}

    // Anlagern des Punktes 'current_point' an die Region 'label'
    template <class ImageIterator>
    bool operator()(ImageIterator current_point, Diff2D loc, int label) const
    {
        if(*current_point != 0) return false; // Anlagern nicht erfolgreich

        // an dieser Stelle wurde das Markieren von Wasserscheiden weggelassen

        *current_point = label;       // -> Anlagern
        return true;                  // Anlagern erfolgreich
    }
};
```

Mit diesem Funktor produziert auch das Wasserscheidenverfahren ein Regionenbild, das wir in ein segmentiertes Zellgitter überführen können. Damit haben wir die Darstellung der Ergebnisse dieser drei Verfahren vereinheitlicht.

8.4.2 Erzeugung eines segmentierten Zellgitters durch Kantendetektion

Nur wenig komplizierter ist die Modifikation des Kantendetektors aus Abschnitt 6.2. Wir erinnern uns, daß der Kantendetektor zunächst mit einem Laplacefilter die zweite Ableitung des Originalbildes bestimmt. In diesem gefilterten Bild werden dann Nulldurchgänge gesucht, d.h. Positionen, wo ein Pixel einen positiven und das Nachbarpixel einen negativen Wert hat. Dies bedeutet aber, daß wir das gefilterte Bild als Binärbild interpretieren können: Pixel mit negativer zweiter Ableitung erhalten den Wert null, die übrigen Pixel den Wert eins. Auf dieses Binärbild können wir nun den Algorithmus aus dem vorigen Abschnitt anwenden, um ein segmentiertes Zellgitter zu erzeugen. Die Linien und Vertizes in diesem Zellgitter entsprechen dann den gesuchten Bildkanten.

Im Gegensatz zum Kantendetektor aus Abschnitt 6.2, der die Kanten *auf* den Pixeln markiert, markiert der modifizierte Algorithmus die Kanten *zwischen* den Pixeln. Letzeres entspricht genau der Definition eines Nulldurchgangs: Nulldurchgänge treten immer *zwischen* zwei Pixeln auf (abgesehen vom seltenen Fall, wo ein

Pixel genau den Wert null hat). Der modifizierte Algorithmus setzt die Definition also besser um als der ursprüngliche. Dies äußert sich unter anderem auch darin, daß die Kanten des neuen Algorithmus (d.h. die Kanten auf dem Zellgitter) stets 4-zusammenhängend und genau ein Pixel breit sind. Wie wir in Abschnitt 8.3 gesehen hatten, ist die Kantenverfolgung dadurch völlig unproblematisch.

Dies ist eine wesentliche Verbesserung gegenüber dem traditionellen Kantenbild, wo insbesondere in der Umgebung von Kreuzungspunkten relativ komplexe Algorithmen verwendet werden müssen, um den Kantenzusammenhang zu bestimmen [TIECKGERL97].

Bisher haben wir jedoch noch nicht berücksichtigt, daß der Kantendetektor aus Abschnitt 6.2 mit Hilfe des Gradientenbetrags die Kantenstärke bewertet und insignifikante Kantenpunkte löscht. Im modifizierten Algorithmus heißt das, daß wir eine 1-Zelle nur dann als Linie markieren, wenn der zugehörige Gradientenbetrag einen Schwellwert überschreitet. Dafür benötigen wir allerdings die Werte des Gradientenbetrags an den Positionen der 1-Zellen, also zwischen den Pixeln des Originalbildes. Diese Informationen müssen wir durch ein geeignetes Interpolationsverfahren berechnen. Die einfachste Möglichkeit besteht darin, das Maximum der Gradientenbeträge der benachbarten Pixel zu verwenden. Dies führt in der Praxis bereits zu sehr befriedigenden Resultaten (vgl. Abbildung 40). Wenn nötig, können selbstverständlich kompliziertere Interpolationsverfahren (z.B. Spline-Interpolation) verwendet werden.

In der folgenden Algorithmenbeschreibung wollen wir die einfache Interpolationsmethode zur Bestimmung der Signifikanz benutzen. Der neue Kantendetektionsalgorithmus benötigt somit ebenfalls das Gradientenbild und das mit dem Laplaceoperator gefilterte Bild („Laplacebild"). Das Zellgitter wird wieder so definiert, daß zur 2-Zelle am Ort $(2x, 2y)$ das Pixel am Ort (x, y) korrespondiert:

Erzeugung eines segmentierten Zellgitters durch Kantendetektion:

1) Erzeugung des Gradientenbildes und des Laplacebildes
2) Alle 2-Zellen des Zellgitters werden als „Region" markiert.
3) Für jede 1-Zelle des Zellgitters:
   - 3.1) Wenn die 1-Zelle zum äußeren Rand des Zellgitters gehört, wird sie als „Linie" markiert.
   - 3.2) Wenn die 1-Zelle zwei 2-Zellen begrenzt, deren korrespondierende Pixel im Laplacebild Werte mit gleichem Vorzeichen haben (Null werten wir dabei als positiv), wird sie als „Region" markiert.
   - 3.3) Wenn die 1-Zelle zwei 2-Zellen begrenzt, deren korrespondierende Pixel im Gradientenbild Werte haben, deren Maximum kleiner ist als ein Schwellwert, wird sie als „Region" markiert.
   - 3.4) Andernfalls wird die 1-Zelle als „Linie" markiert.

4) Für jede 0-Zelle des Zellgitters:
   4.1) Wenn die 0-Zelle zum äußeren Rand des Zellgitters gehört, wird sie als „Vertex“ markiert.
   4.2) Wenn die 0-Zelle nur 1-Zellen begrenzt, die als „Region“ markiert wurden, wird sie ebenfalls als „Region“ markiert.
   4.3) Andernfalls wird die 0-Zelle als „Vertex“ markiert.
5) Durch Markieren der Zusammenhangskomponenten des Zellgitters werden allen Untermengen (Regionen, Linien und Vertizes) eindeutige label zugeordnet.

Abbildung 40 stellt die Ergebnisse des originalen und des modifizierten Difference-of-Exponential Kantendetektors (einschließlich Kantenbewertung) einander gegenüber. Zur besseren Veranschaulichung sind im segmentierten Zellgitter die Linien und Vertizes schwarz, die Regionen weiß (und nicht mit ihren labels) markiert, wie es bei einem traditionellen Kantenbild üblich ist. Wie erwähnt, liefert der modifizierte Algorithmus – im Gegensatz zum originalen – Kanten, die stets 4-zusammenhängend und ein Pixel breit sind.

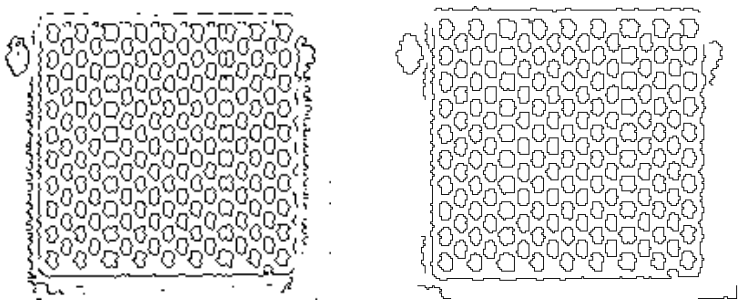


Abbildung 40: links: Kantenbild, das der originale Difference-Of-Exponential-Algorithmus liefert (vergleiche Abbildung 10),
rechts: vom modifizierten Difference-Of-Exponential-Algorithmus erzeugtes segmentiertes Zellgitter (Regionen sind weiß, Linien und Vertizes schwarz markiert)

8.4.3 Iteratives Regionenwachstum zur Gewinnung einer Zellpyramide

Viele Segmentierungsalgorithmen, wie z.B. der Wasserscheidenalgorithmus aus Abschnitt 6.3.1, neigen zur *Übersegmentierung* des Bildes. Das heißt, das segmentierte Bild enthält mehr Regionen als nach dem Inhalt der dargestellten Szene zu erwarten wäre. Die Übersegmentierung kann man unter anderem dadurch beseitigen, daß

man in einem oder mehreren Nachbearbeitungsschritten Regionen wieder zusammenfaßt, die fälschlicherweise getrennt wurden.

Wir wollen im folgenden annehmen, daß der initiale Algorithmus als Ergebnis einen (über-)segmentierten Zellkomplex liefert. Wie wir in Abschnitt 7.4.2 gezeigt haben, können wir diesen Zellkomplex in einen neuen Zellkomplex kontrahieren. Das bedeutet, daß die Beseitigung einer Übersegmentierung nur ein Spezialfall der allgemeineren Aufgabe ist, einen beliebigen Zellkomplex zu segmentieren. Wir wollen deshalb in diesem Abschnitt einen Segmentierungsalgorithmus vorstellen, der einen Zellkomplex in einen einfacheren Zellkomplex transformiert (die Schritte „Segmentierung“ und „Kontraktion“ werden dabei simultan ausgeführt).

Um dieses Ziel zu erreichen, wollen wir den Regionenwachstums-Algorithmus aus Abschnitt 6.3 so modifizieren, daß er nicht mehr auf Bildern arbeitet, sondern auf Zellkomplexen. Wenn man den ursprünglichen Algorithmus (bzw. die Funktion `seededRegionGrowing`) anschaut, macht man eine interessante Beobachtung: die Tatsache, daß der Algorithmus auf einer Bilddatenstruktur arbeitet, spielt bei der Implementation keine wesentliche Rolle. Der Bilditerator wird nur als eine Art Zeiger benutzt, der jedes Pixel des Bildes eindeutig identifiziert. Das Wissen über die 2-dimensionale Struktur des Bildes ist im Nachbarschaftszirkulator gekapselt und für den Algorithmus nicht sichtbar.

Daraus folgt, daß wir das Verhalten des Algorithmus grundlegend ändern können, wenn wir diese zwei Iteratoren austauschen. Um einen Zellkomplex zu verwenden, müssen wir 2-Zellen anstelle von Pixeln verwenden, und die Pixelnachbarschaft muß durch die Nachbarschaft der 2-Zellen ersetzt werden. Die Schnittstelle eines Zellkomplexes gemäß Abschnitt 7.5 bietet hierfür geeignete Iteratoren: mit einem Face Iterator können wir sämtliche 2-Zellen eines Zellkomplexes aufsuchen, und der Face Adjacency Iterator liefert die Nachbarn einer gegebenen 2-Zelle.

Wir können deshalb die Implementation des Regionenwachstums so ändern, daß sie diese Iteratoren verwendet. Die generelle Vorgehensweise des Algorithmus bleibt jedoch vollständig erhalten. Die folgende Algorithmenbeschreibung ist deshalb der ursprünglichen Beschreibung in Abschnitt 6.3 sehr ähnlich:

Seeded Region Growing auf Zellkomplexen:

1) Initialisiere die Prioritätswarteschlange der Kandidatenzellen:
   1.1) Finde alle 2-Zellen, die als Keimzellen markiert sind und suche deren benachbarte 2-Zellen auf
   1.2) Wenn die aktuelle Nachbarzelle frei (also keine Keimzelle) ist: erkläre diese Zelle zur Kandidatenzelle, berechne die Kosten für die Anlagerung der Kandidatenzelle an die aktuelle Keimzelle und sortiere die Kandidatenzellen nach ansteigenden Kosten in die Prioritätswarteschlange ein. Behandle Zellen mit gleichen Kosten nach der Methode “First in - first out”.

2) Solange noch Kandidaten in der Prioritätswarteschlange sind:
   2.1) Entferne die Kandidatenzelle mit den geringsten Kosten aus der Warteschlange
   2.2) Wenn diese Kandidatenzelle immer noch frei ist: verschmelze sie mit derjenigen benachbarten Keimzelle, für die minimale Kosten entstehen
   2.3) Stelle fest, ob durch die letzte Verschmelzung weitere freie Zellen zu Nachbarn der Keimzelle geworden sind. Wenn ja, erkläre diese zu Kandidatenzellen, berechne ihre Kosten und ordne sie in die Prioritätswarteschlange ein

Um diesen Algorithmus zu implementieren, benötigen wir noch einen zusätzlichen Funktor `MarkFreeCells`, mit dessen Hilfe wir freie 2-Zellen von den Keimzellen unterschieden können. Die Funktoren zum Berechnen der Kosten und zum Verschmelzen von Regionen haben die gleiche Bedeutung wie in Seeded Region Growing. Die Implementation des Regionenwachstums auf Zellkomplexen sieht dann folgendermaßen aus:

```
template <class CellComplex,
          class MarkFreeCells, class CostFunctor, class MergeFunctor>
void seededCellFaceMerging(CellComplex & cellComplex,
                    MarkFreeCells isFree, CostFunctor cost, MergeFunctor merge)
{
    // der Typ des Iterators für 2-Zellen
    typedef typename CellComplex::FaceIterator FaceIterator;

    // Zugriffsobjekt für die Faces
    typename CellComplex::FaceAccessor face;

    // interne Hilfsklasse für die Verwaltung von Kandidatenzellen
    typedef CandidateCell<FaceIterator, typename CostFunctor::value_type>
            Candidate;

    // Erzeugen der Kandidatenliste
    std::priority_queue<Candidate, std::vector<Candidate>,
                                std::greater<Candidate> > candidateList;
    int count = 0;  // Zähler für die Kandidaten

    // Schleife über alle 2-Zellen
    FaceIterator current = cellComplex.beginFaces();
    FaceIterator end     = cellComplex.endFaces();
    for(; current != end; ++current) {

        // ist die aktuelle 2-Zelle Bestandteil einer Keimregion ?
        if(!isFree(face(current))) {
            // ja -> suche in der Nachbarschaft nach freien Punkten
            findCandidatesInNeighborhood(current, face,
                                         candidateList, isFree, cost, count);
        }
    }
```

```
        // solange Kandidaten in candidateList sind
        while(!candidateList.empty())
        {
            // Iterator für Kandidaten erfragen
            FaceIterator candidateCell = candidateList.top().candidate;

            // Iterator für die 2-Zelle, an die der Kandidat angelagert werden soll
            FaceIterator absorbingCell = candidateList.top().absorber;

            // Kandidaten aus der Liste entfernen
            candidateList.pop();

            // Kandidaten verschmelzen
            bool merge_succeeded = merge(absorbingCell, candidateCell);

            // wenn die Verschmelzung erfolgreich durchgeführt wurde
            if(merge_succeeded)
            {
                // neue Kandidaten in der Nachbarschaft suchen
                findCandidatesInNeighborhood(absorbingCell, face,
                                             candidateList, isFree, cost, count);
            }
        }
    }
```

Auch die Klasse `CandidateCell` und die Funktion `findCandidatesInNeighborhood` erfahren gegenüber der ursprünglichen Implementation in Abschnitt 6.3 leichte Modifikationen, die wir hier aber nicht angeben wollen. Da `seededCellFaceMerging()` einen Zellkomplex wiederum in einen Zellkomplex transformiert, können wir diese Funktion iterativ aufrufen. Die jeweiligen Zwischenergebnisse entsprechen den einzelnen Stufen einer Zellpyramide. Der folgende Code realisiert dies:

```
CellComplex cellcomplex, cellPyramid[maxLevel+1];

initCellcomplex(cellcomplex); // Initialisierung
cellPyramid[0] = cellcomplex;

for(int level = 1; level <= maxLevel; ++level)
{
    markSeeds(cellcomplex);

    seededCellFaceMerging(cellcomplex, MarkFreeCellsFunctor(),
                          CostFunctor(), MergeFunctor());

    cellPyramid[level] = cellcomplex;
}
```

Durch geeignete Wahl der Keimzellen und des Kostenfunktors kann man das Verhalten des Regionenwachstums in weiten Grenzen variieren. Welche Entscheidungen zu guten Ergebnissen führen, ist zur Zeit noch offen. Zur Erforschung dieser

Frage versuchen wir einerseits, bewährte Ideen aus der Segmentierung von Bildern auf die Segmentierung von Zellkomplexen zu übertragen. Wünschenswert wäre zum Beispiel die Verallgemeinerung des Wasserscheidenverfahrens, so daß man es iterativ auf Zellkomplexe anwenden kann. Schwierigkeiten bereitet dabei die Bestimmung der Keimzellen: die auf Bildern verwendeten lokalen Minima des Gradientenbetrags haben in einem unregelmäßig strukturierten Zellkomplex keine direkte Entsprechung, so daß wir eine Alternative finden müssen.

Andererseits werden wir versuchen, bekannte graph-basierte Verfahren, wie zum Beispiel die Verfahren von Montanvert et al. [MONTAN+91], Kropatsch und Willersinn [WILLKROP94, KROPATSCH95] sowie von Tieck und Gerloff [TIECKGERL97], so zu modifizieren, daß sie Zellkomplexe verarbeiten können. Dies erscheint insbesondere bei den beiden letzten Verfahren aussichtsreich, weil diese bereits jetzt duale Graphen (Kanten- und Flächengraphen) verwenden. Darüber hinaus werden Kriterien der „Guten Gestalt" (engl. perceptual grouping, vergl. [MOHNEVAT92, SARKBOY93]) sowie die Verwendung von Vorwissen über die Szene zu sehr interessanten Algorithmen führen, besonders auf den höheren Ebenen der Zellpyramide. Hierzu besteht jedoch noch erheblicher Forschungsbedarf.

8.5 Zusammenfassung des Kapitels

In diesem Kapitel haben wir folgendes gezeigt:

- Die abstrakte Schnittstelle für ebene Zellkomplexe, die wir in Abschnitt 7.5 definiert haben, läßt sich auf verschiedene Arten effizient implementieren. Wir haben mit dem graph-basierten Zellkomplex, dem Zellgitter und dem kontrahierten Zellgitter drei Varianten hierfür vorgestellt, die jeweils unterschiedliche Eigenschaften hinsichtlich Speicherverbrauch und Flexibilität haben.
- Die Methode der Iterator-Adapter ist ein wesentliches Hilfsmittel bei der Implementation von generischen Zellkomplexen.
- Typische Segmentierungsverfahren lassen sich so modifizieren, daß sie ein segmentiertes Zellgitter (also eine Segmentierung der Khalimsky-Ebene) liefern.
- Das Regionenwachstum kann so verallgemeinert werden, daß es auf Zellkomplexen arbeitet. Durch iterative Anwendung dieses Algorithmus lassen sich Zellpyramiden erzeugen.

Kapitel 9

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden erstmalig in größerem Umfang Methoden der generischen Programmierung beim Entwurf und bei der Implementation von Computer Vision-Algorithmen eingesetzt. Es ist dadurch gelungen, ein umfassendes und konsistentes System generischer Konzepte für viele häufig auftretende Computer Vision-Probleme zu schaffen. Dieser Durchbruch erforderte die Integration neuer Erkenntnisse aus Computer Vision, Mathematik und Softwaretechnologie. Im Vergleich zu bisherigen prozeduralen und objekt-orientierten Lösungen konnten wir einen wesentlich höheren Grad an Flexibilität erreichen, ohne daß sich die Performanz signifikant verschlechterte. Gleichzeitig konnten wir die Redundanz des Codes und damit den Programmieraufwand stark verringern.

Die generische Programmierung gestattet es in weit höherem Maße als bisherige Ansätze, abstrakte Ideen ohne Verlust der Abstraktion in Softwarebausteine umzusetzen. Insbesondere gilt dies für Algorithmen, die nicht mehr von bestimmten Datenstrukturen abhängen müssen. Dies ermöglicht eine wesentlich bessere Zerlegung der Problemstellung in Bausteine, welche je nach Anwendung maßgeschneidert miteinander kombiniert werden können. Eine solche Zerlegung entspricht der Definition unabhängiger Abstraktionsachsen, so daß mit wenigen grundlegenden Abstraktionen eine große Zahl an Varianten erzeugt werden kann. Dank der ausgereiften Methoden der Selbstkonfiguration, die der template-Mechanismus der Sprache C++ bietet, müssen diese Varianten nicht explizit implementiert werden, sondern werden bei Bedarf automatisch generiert. Dadurch wird das „Problem des kartesischen Produkts“ gelöst, das bisher der Flexibilität schnell Grenzen gesetzt

hatte, weil der Implementationsaufwand exponentiell mit der Anzahl der Varianten anstieg.

Als entscheidendes Hilfsmittel bei der Definition der generischen Konzepte hat sich die starke Betonung der algorithmischen Abstraktion erwiesen. Dies steht im Gegensatz zu bisherigen Ansätzen, wo meist die Datenabstraktion im Vordergrund stand. Dank der Forderung, minimale Anforderungen von Algorithmen in Form von required interfaces explizit zu spezifizieren, waren wir in der Lage, bisher unbemerkte Gemeinsamkeiten zwischen Algorithmen aufzudecken und unnötige Details aus den Schnittstellendefinitionen zu eliminieren. Im Ergebnis konnten wir abstrakte Algorithmen für eine Reihe typischer Computer Vision-Verfahren, darunter Bildarithmetik, Filterung und Segmentierung, entwickeln, die mit beliebigen Datenstrukturen zusammenarbeiten können, solange diese die geforderten Schnittstellen implementieren.

Es hat sich dabei gezeigt, daß ein leistungsfähiger Server gegenüber verschiedenen Clients mit unterschiedlichen Schnittstellen auftreten kann und muß. Eine Datenstruktur kann im Kontext unterschiedlicher Algorithmen ganz verschiedene Rollen einnehmen, die jeweils eine andere Schnittstelle erfordern. Um den Server nicht mit zu vielen Schnittstellen zu überfrachten, hat es sich als zweckmäßig erwiesen, Schnittstellenkonzepte von vornherein als Adapter zu formulieren, insbesondere mit Hilfe von Iteratoren und Zugriffsobjekten. Auf diese Weise kann man die geforderten Schnittstellen bei einem geeigneten Server sehr leicht implementieren, wenn nötig auch nachträglich.

Im einzelnen haben wir in dieser Arbeit folgende wichtige Ergebnisse erzielt:

Analyse von Anforderungen an flexible Computer Vision-Systeme: Flexibilität wurde dabei als die Möglichkeit der lokalen Eingrenzung wahrscheinlicher Änderungen definiert. Insbesondere haben wir die Notwendigkeit herausgearbeitet, das „Problem des kartesischen Produkts“ (exponentielle Explosion der Variantenzahl) ohne wesentliche Einbußen bezüglich der Rechengeschwindigkeit zu lösen.

Definition von 2-dimensionalen Iteratoren und Zugriffsobjekten: Diese Iteratoren und Zugriffsobjekte dienen als Schnittstellen zwischen Bildern und Bildverarbeitungsfunktionen. Dadurch benötigen wir nur fünf grundlegende Funktionen sowie eine Reihe von Funktoren, um eine große Menge verschiedener Punktoperationen ausführen zu können. Auch Filter und Segmentierungsverfahren lassen sich als generische Algorithmen implementieren. Gegenüber einem herkömmlichen System wie Khoros sinkt der Quellcodeumfang auf ein Zehntel bei gleichzeitig wesentlich höherer Flexibilität und kaum verringerter Performanz.

Definition von generischen Konzepten und Metaklassen für arithmetische Operationen: Dies ermöglicht vielen Algorithmen ein großes Maß an Selbstkonfiguration, so daß sie sich ohne Änderung auf verschiedene Bild- und Pixeltypen anwen-

den lassen. Wir haben diese Techniken unter anderem genutzt, um arithmetische Operationen für RGB-Tupel zu definieren. Dies verleiht RGB-Tupeln die Eigenschaften einer linearen Algebra, so daß die Anforderungen zahlreicher Algorithmen abgedeckt werden.

Entwicklung von verschiedenen Iterator-Adaptern: Mit Hilfe von Iterator-Adaptern haben wir das Randproblem gelöst, wir konnten die verschiedensten linearen Untermengen aus Bildern extrahieren, und wir haben bestimmte Algorithmen in Form von Iteratoren formuliert. Algorithmen und Navigationsmuster lassen sich dadurch frei verknüpfen, das heißt, es ergibt sich eine wesentlich verbesserte Zerlegung des Systems in einzelne Bausteine.

Definition von ebenen Zellkomplexen und darauf aufbauenden Segmentierungen: Ebene Zellkomplexe erfüllen die Axiome eines topologischen Raumes und gestatten daher die widerspruchsfreie Definition von zusammenhängenden Regionen, Nachbarschaften und Konturen. Dies ist die Basis für unsere neue Definition der Bildsegmentierung. Mit Hilfe von elementaren Euler-Operatoren können Segmentierungen eines Zellkomplexes in einen neuen Zellkomplex kontrahiert werden. Die iterative Anwendung der Operationen Segmentierung und Kontraktion führt zu einer Zellpyramide, die bisherige Ansätze für irreguläre Pyramiden verallgemeinert.

Formulierung von Schnittstellenkonzepten für Zellkomplexe: Die Struktur eines Zellkomplexes kann durch wenige Iteratoren und Zugriffsobjekte vollständig beschrieben werden. Dadurch wird die Handhabung von Zellkomplexen entscheidend vereinfacht, und die Ergebnisse verschiedener Segmentierungsverfahren lassen sich in einheitlicher Form darstellen. Die Flexibilität der Schnittstelle wird dadurch unterstrichen, daß wir drei grundlegend verschiedene Implementationen angeben konnten. Auch die Kontraktion eines segmentierten Zellkomplexes läßt sich durch wenige abstrakte Funktoren realisieren.

Generische Implementation von Segmentierungsalgorithmen: Wir konnten den Nachweis erbringen, daß die neuen Konzepte der generischen Programmierung geeignet sind, wichtige Algorithmen der Computer Vision (darunter Kantendetektion und Regionenwachstum) zu implementieren. Die damit verbundene höhere Flexibilität verringert den Aufwand bei der Anpassung dieser Algorithmen an einen neuen Kontext. Durch die Verwendung zellulärer Komplexe werden die Repräsentationen von Segmentierungsergebnissen vereinheitlicht, bisher notwendige Konvertierungen eliminiert und ganz neue Kombinationsmöglichkeiten geschaffen.

Das wichtigste Einzelergebnis ist zweifellos die mathematisch exakte Definition einer Segmentierung als vollständige Zerlegung eines ebenen Zellkomplexes in Regionen, Linien und Vertizes. Damit ist es gelungen, eine allgemein verwendbare

Repräsentation für Segmentierungsergebnisse zu schaffen, die nachfolgenden Analysealgorithmen einheitliche Voraussetzungen bietet, ungeachtet dessen, mit welchem Verfahren die Segmentierung erzeugt wurde. Inkompatibilitäten zwischen verschiedenen Repräsentationen sowie Inkonsistenzen bei der Definition von Nachbarschaften und Konturen, die bei herkömmlichen Ansätzen häufig auftreten (wie beispielsweise das Zusammenhangsparadoxon), konnten dadurch überwunden werden.

Am Beispiel von drei verschiedenen Datenstrukturen (den graphbasierten Zellkomplexen, dem Zellgitter und dem kontrahierten Zellgitter) konnten wir zeigen, daß die abstrakten mathematischen Konzepte mit verhältnismäßig geringem Aufwand generisch implementiert werden können. Dies wurde nicht zuletzt durch die feinkörnige Zerlegung der Schnittstelle in Iteratoren, Zugriffsobjekte und Funktoren erreicht. Jedem Teil der Schnittstelle wird so eine genau umrissene, einfache Aufgabe zugewiesen. Hier erweist sich die separate Behandlung von Zugriffsobjekten als besonders hilfreich, weil nur so gewährleistet ist, daß jede Anwendung andere Zusatzinformationen mit den Merkmalen verbinden kann, ohne daß dafür der unterliegende Zellkomplex oder die navigierenden Teile der Schnittstelle geändert werden müssen.

Alle in dieser Arbeit beschriebenen Konzepte, Algorithmen und Datenstrukturen sind in der vom Autor implementierten Computer Vision-Bibliothek VIGRA (VIsion with GeneRic Algorithms) verwirklicht worden. Im Anhang (Seite 251) geben wir einen Überblick über die Funktionalität dieser Bibliothek. Ein großer Teil der Funktionalität ist in der public domain frei zugänglich.[29] Bei der Anwendung der VIGRA-Bibliothek in unserer Arbeitsgruppe haben sich die Erwartungen an eine verbesserte Flexibilität bestätigt. Es hat sich außerdem gezeigt, daß ein allmählicher Übergang zur generischen Programmierung möglich ist, weil einerseits C++ verschiedene Programmierparadigmen gleichzeitig unterstützt und andererseits für existierende Bausteine auch nachträglich generische Schnittstellen implementiert werden können. Ein solches allmähliches re-engineering existierender Lösungen ist auch und gerade für industrielle Anwendungen relevant, weil dadurch die bisherigen Investitionen geschützt und sogar aufgewertet werden. Sinnvoll wäre es dabei, den Übergang zur generischen Programmierung mit einer Algorithmen-Validierung zu verbinden, so daß sich generische Bausteine nicht nur durch hohe Flexibilität, sondern auch durch zertifizierte Leistungsdaten auszeichnen würden.

Trotz dieser Fortschritte mußten allerdings eine Reihe von Problemen offen bleiben. Die hier vorgestellten Konzepte stellen sicherlich nur einen ersten Schritt dar, der in verschiedene Richtungen weiterentwickelt werden muß.

Erstens ist es notwendig, weitere Algorithmen auf die neuen Konzepte umzustellen. Dabei werden zweifellos einige Aspekte zutage treten, wo diese Konzepte noch

[29] http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/

verfeinert werden müssen. Beispielsweise haben wir im Rahmen der Zellkomplexe nur diejenigen Transformationen behandelt, die den Zellkomplex vereinfachen. Die Definition der inversen Transformationen ist schwieriger, weil dabei zusätzliche Randbedingungen beachtet werden müssen. Die genaue Analyse der Anforderungen von Algorithmen, die diese Operationen verwenden wollen (z.B. Split-and-Merge-Algorithmus), wird hierfür sinnvolle Wege aufzeigen. Ähnliches gilt für die Schnittstellen zur Kennzeichnung einer Segmentierung, die sich durch die Analyse weiterer Algorithmen möglicherweise noch verbessern läßt.

Zweitens sollten die vorgestellten Konzepte auf drei Dimensionen (Volumendaten) erweitert werden, da der Auswertung von Volumendaten immer größere Bedeutung zukommt (besonders in der Medizin). Solange nur eine bestimmte eingeschränkte Klasse 3-dimensionaler Oberflächen benötigt wird (die sogenannten polyhedral complexes [KETTNER98]), lassen sich unsere Definitionen leicht verallgemeinern. Schwieriger wird es allerdings, sobald man sogenannte 3-Zellen hinzunehmen muß. Dann kann man keine eindeutige Inzidenzordnung der 1-Zellen an jeder 0-Zelle mehr festlegen und muß statt dessen eine Inzidenzordnung von 2-Zellen bezüglich einer 1-Zelle verwenden. Diese Fragen werden zukünftig eine sorgfältige Untersuchung erfordern.

Drittens schließlich haben wiederverwendbare Softwarekonzepte nur dann einen Sinn, wenn sie tatsächlich wiederverwendet werden. In diesem Zusammenhang kommt es zunächst vor allem darauf an, möglichst viele bekannte Algorithmen so zu modifizieren, daß sie einen Zellkomplex liefern bzw. anfordern. Besonders interessant erscheint uns die Möglichkeit, auf der Basis dieser einheitlichen Repräsentation verschiedene Algorithmen miteinander zu kombinieren, um die Schwächen einzelner Verfahren auszugleichen. Hierfür wäre eine Standardisierung generischer Konzepte innerhalb der Computer Vision Community wünschenswert.

Flexible, zertifizierte Implementationen wären außerdem eine große Hilfe bei der zur Zeit mehr und mehr geforderten Validierung und Evaluierung von Computer Vision-Algorithmen [VIERGEVER+99]. Frei verfügbare und getestete Referenzimplementationen bewährter Verfahren würden die Evaluierung neuer Algorithmen sowie die Erschließung neuer Anwendungsfelder für Computer Vision sehr erleichtern. Voraussetzung hierfür ist jedoch, daß sich diese Implementationen mit geringem Aufwand in einen neuen Kontext integrieren und an dessen besondere Anforderungen anpassen lassen. Die generische Programmierung bietet hierfür geeignete Möglichkeiten, wie beispielsweise mit der flexiblen Implementation des Regionenwachstums in Abschnitt 6.3 angedeutet wurde.

Ein ganz anderes Problem, das nicht für Computer Vision-Anwendungen spezifisch ist, ergibt sich aus der Art, wie die generische Programmierung umgesetzt wird. Der innerhalb der Sprache C++ verwendete template-Mechanismus ist zwar sehr leistungsfähig, aber er wirkt ausschließlich während der Compilierung. Dies ist zweifellos für viele Aufgaben die angemessene Lösung und dient außerdem der

Minimierung des Geschwindigkeitsverlustes, den Abstraktion verursacht. Häufig benötigt man jedoch darüber hinaus Flexibilität zur Laufzeit. Beispielsweise kann man in vielen Fällen erst nach dem Laden eines Bildes entscheiden, ob es sich um ein skalares Bild mit dem Pixeltyp `byte` oder `float` handelt, oder möglicherweise sogar um ein RGB-Bild. Demzufolge können wir auch erst zur Laufzeit entscheiden, welche Variante eines Verfahrens jeweils angewendet werden muß.

Bei der Realisierung von Laufzeitflexibilität bietet der template-Mechanismus keine Unterstützung. Wir sind hier auf traditionelle Mechanismen der Sprache C++ angewiesen. Es hat sich beispielsweise als gute Lösung erwiesen, generische Algorithmen in Objekten zu kapseln. Dadurch können wir virtuelle Funktionen nutzen, um Algorithmen zur Laufzeit auszuwählen. Man beachte, daß dies nicht heißt, daß wir die Algorithmen selbst als member functions implementieren. Die member functions haben ausschließlich die Aufgabe, Laufzeitflexibilität zu sichern und reichen die eigentliche Arbeit direkt an einen generischen Algorithmus weiter. Dadurch wird die Zerlegung der Systemfunktionalität sogar verbessert: polymorphe Objekte koordinieren mit Hilfe von virtuellen Funktionen zur Laufzeit die *Anwendung* der Algorithmen, während die *Implementation* dieser Algorithmen von unabhängigen und effizienten generischen Bausteinen bereitgestellt wird. Man kann sogar noch weiter gehen und die Objekte wiederum in einer interpretierten Sprache wie Java oder Python kapseln. Auf diese Weise kann man mit relativ geringem Aufwand ein interaktives Computer Vision-System auf der Basis generischer Algorithmen implementieren.

Für die Zukunft ist zu hoffen, daß die Compilertechnologie soweit verbessert wird, daß Flexibilität zur Laufzeit mit den gleichen syntaktischen und semantischen Sprachmitteln realisiert wird wie Flexibilität zur Übersetzungszeit. Ansätze in dieser Richtung findet man beispielsweise in der Sprache Dylan [SHALIT96], wo der Compiler automatisch entscheidet, ob ein Funktionsaufruf erst zur Laufzeit gebunden werden kann oder ob dies bereits zur Übersetzungszeit möglich ist. Interessant sind auch die Versuche, aktive Softwarebibliotheken zu entwickeln, die sich zur Übersetzungszeit partiell spezialisieren, soweit die bereits verfügbaren Informationen dies zulassen [JONES96, VELDGANN98]. In eine ähnliche Richtung zielen die zur Zeit viel diskutierten just-in-time compiler. Es bleibt abzuwarten, welche Konzepte sich letztendlich durchsetzen werden – die Entwicklung von generischen Konzepten für Systeme auf der Basis von "visual information technologies" (wozu neben der Computer Vision auch die Computergraphik gehört) muß jedoch bereits jetzt vorangetrieben werden.

Anhang:
Funktionalität der VIGRA-Bibliothek

Sämtliche Konzepte dieser Arbeit wurden im Rahmen der vom Autor entwickelten VIGRA-Bibliothek implementiert. Wir wollen hier einen kurzen Überblick über die Bestandteile dieser Bibliothek geben. Diejenigen Teile, die auch in der öffentlich zugänglichen Version der Bibliothek enthalten sind, kennzeichnen wir mit einem Stern *.

Datenstrukturen:

- RGB-Werte einschließlich der arithmetischen Operationen (Abschnitt 4.5.2)*
- generische Bilddatenstrukturen mit den notwendigen Iteratoren und Zugriffsobjekten (Abschnitte 4.3, 4.5 und 4.6) sowie Import/Exportfiltern für gängige Fileformate*
- Filterklassen für die Faltungsoperation: Gaußfilter und beliebige Ableitungen der Gaußfunktion, Binomial- und Boxfilter beliebiger Größe, symmetrische erste Ableitung, Sobelfilter, beliebige nutzerdefinierte Filter*
- Zellkomplexe mit den notwendigen Iteratoren und Zugriffsobjekten: graph-basierte Implementation (Abschnitt 8.1), Zellgitter (Abschnitt 8.2), kontrahiertes Zellgitter (Abschnitt 8.3)
- Traits-Metaklassen für arithmetische Operationen (Abschnitt 4.10.2)*

Bildverarbeitungsalgorithmen:

- Punktoperationen: Initialisieren, Kopieren, Ausschneiden von Bildern, unäre, binäre und ternäre Punkttransformationen, auch mit beliebiger region of interest (Abschnitt 4.7)*
- Funktoren für häufig benötigte Punkttransformationen (Abschnitt 4.1)*
- Expression templates zur automatischen Generierung von Funktoren [KÖTHE99A]
- Standardfaltung (Abschnitt 4.8) und separierbare Faltung (Abschnitt 5.5) mit verschiedenen Randbehandlungsverfahren (reflektive und periodische Randbedingungen, konstanter Wert, nächster Nachbar, Modifikation des Filters am Rand) unter Verwendung der oben definierten Filterklassen*
- rekursive Filter (Exponentialfilter und dessen erste und zweite Ableitung, Abschnitt 5.6)*

- morphologische Filter: Erosion, Dilatation, Medianfilter und Rangordnungsfilter mit kreisförmigen Masken beliebiger Größe*
- Bildvergrößerung und –verkleinerung mit verschiedenen Interpolationsverfahren (nächster Nachbar, lineare Interpolation, bi-kubische Spline-Interpolation)*
- Distanztransformation für verschiedene Distanzmetriken (city block, euklidisch, Schachbrett)

Bildanalysealgorithmen:

- Kantendetektion (Laplacean-of-Gaussian und Laplacean-of-Exponential, Abschnitt 6.2)*
- modifizierte Kantendetektion für Zellgitter (Abschnitt 8.4.2)
- Eckendetektion (Förstner-, Plessey-*, Rohr-, Beaudet- und Kitchen/Rosenfeld-Operatoren, Abschnitt 4.10)
- Schwellwertbildung (Abschnitt 6.1)*
- Markierung von Zusammenhangskomponenten (4- und 8-Nachbarschaft)*
- Regionenwachstum (seeded region growing, Abschnitt 6.3)*
- modifiziertes Wasserscheidenverfahren für Zellgitter (Abschnitt 8.4.1)
- iteratives Regionenwachstum für Zellpyramiden (8.4.3)
- Bestimmung lokal optimaler Skalen nach den Verfahren von [LINDEBERG98] und [KÖTHE96]
- statistische Regionenanalyse*, Konturverfolgung (Abschnitt 5.8)

Daneben gibt es eine Schnittstelle zur Anbindung der VIGRA-Bibliothek an die Programmiersprache Python, so daß die VIGRA-Funktionalität nicht nur in C++, sondern auch in Python-Skripten sowie interaktiv über die Kommandozeile aufgerufen werden kann. Die VIGRA-Bibliothek wird außerdem ergänzt durch ausführliche on-line Dokumentation. Der Quellcode und die Dokumentation der öffentlichen Teile sind unter `http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/` im Internet verfügbar.

Literaturverzeichnis

[ADAMBISCH94] R. Adams, L. Bischof: *"Seeded Region Growing"*, IEEE Trans. Pattern Analysis and Machine Intelligence, 16(6), pp. 641-647, 1994

[AUPPERLE97] M. Aupperle: *"Die Kunst der objektorientierten Programmierung mit C++"*, Braunschweig: Vieweg, 1997

[AUSTERN98] M. Austern: *"Generic Programming and the STL"*, Reading: Addison-Wesley, 1998

[BALLBROWN82] D. Ballard, C. Brown: *"Computer Vision"*, Englewood Cliffs: Prentice Hall, 1982

[BALZERT96] H. Balzert: *"Lehrbuch der Software- Technik"*, Band 1, Heidelberg: Spektrum Akademischer Verlag, 1996

[BARTNACK94] J. Barton, L. Nackman: *"Scientific and Engineering C++"*, Reading: Addison-Wesley, 1994

[BATORY+93] D. Batory, V. Singhal, M. Sirkin, J. Thomas: *"Scalable Software Libraries"*, in: D. Notkin (ed.): Proc. 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York: ACM Press, 1993

[BATORY98] D. Batory: *"Product-Line Architectures"*, Proc. 4. Fachkongreß Smalltalk und Java in Industrie und Anwendung, 1998

[BECK+98] K. Beck and others: *"Once and Only Once"*, *"Refactoring and Rewriting"*, open discussions at the Portland Pattern Repository WWW site (http://c2.com/cgi/wiki?OnceAndOnlyOnce, http://c2.com/cgi/wiki?RefactoringAndRewriting), 1998

[BEILSTIEHL95] W. Beil, H.S. Stiehl: *"Comparison of Two Multi-Scale Approaches to Edge Detection in Medical Images"*, in: N. Ayache (ed.): Computer Vision, Virtual Reality, and Robotics in Medicine, Proc. 1st Intl. Conf. CVRMed ´95, Lecture Notes in Computer Science vol. 905, Berlin: Springer, 1995

[BIGGER94] T. Biggerstaff: *"The Library Scaling Problem and the Limits of Concrete Component Reuse"*, in: W. Frakes (ed.): Proc. 3rd Intl. Conf. Software Reuse, Los Alamitos: IEEE Computer Society Press, 1994

[BOEHM88] B. Boehm: *"A Spiral Model of Software Development and Enhancement"*, IEEE Computer Magazine, 21(5), pp. 61-72, 1988

[BOOCH87] G. Booch: *"Software Components with Ada"*, Menlo Park: Benjamin/Cummings Publishing Company, 1987

[BOOCH94] G. Booch: *"Object-oriented Design With Applications"*, Second Edition, Reading: Addison-Wesley, 1994

[BRAQDOM96] J.-P. Braquelaire, J.-P. Domenger: *"Representation of Region Segmented Images with Discrete Maps"*, Université Bordeaux, Laboratoire Bordelais de Recherche en Informatique, Technical Report 1127-96, 1996

[BRICEFEN70] C. Brice, C. Fennema: *"Scene Analysis Using Regions"*, Artificial Intelligence, 1(3), pp. 205-226, 1970

[BROOKS75/95] F. Brooks: *"The Mythical Man-Month"*, Reading: Addison-Wesley, 1975, Extended Anniversary Edition 1995

[BURNKÖN96] C. Burnikel, J. Könemann: *"High-Precision Floating Point Numbers in LEDA"*, Max-Planck-Institut für Informatik, Saarbrücken, Technical Report 96-1-002, 1996

[BURT84] P. Burt: *"The Pyramid as a Structure for Efficient Computation"*, in: A. Rosenfeld (ed.): Multiresolution Image Processing and Analysis, Berlin: Springer, 1984

[C++98] International Organization for Standardization (ISO): *"International Standard 14882: Programming Language C++"*, 1998

[CANDELA98] Candela Development Team: *"Candela 2.1 Documentation"*, KTH Stockholm, Computational Vision and Active Perception Laboratory, 1998 (http://www.nada.kth.se/cvap/software.html)

[CANNY86] J. Canny: *"A Computational Approach to Edge Detection"*, IEEE Trans. Pattern Analysis and Machine Intelligence, 8(6), pp. 679-698, 1986

[CHAMP+93] D. de Champeaux, D. Lea, P. Faure: *"Object-Oriented System Development"*, Reading: Addison-Wesley, 1993

[COCKBURN98] A. Cockburn: *"Surviving Object-Oriented Projects"*, Reading: Addison-Wesley, 1998

[DAHL+72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: *"Structured Programming"*, London: Academic Press, 1972

[DERICHE90] R. Deriche: *"Fast Algorithms for Low-Level Vision"*, IEEE Trans. Pattern Analysis and Machine Intelligence, 12(1), pp. 78-87, 1990

[DOLAN+96] J. Dolan, C. Kohl, R. Lerner, J. Mundy, T. Boult, J.R. Beveridge: *"Solving Diverse Image Understanding Problems Using the Image Understanding Environment"*, Proc. 1996 ARPA Image Understanding Workshop, San Francisco: Morgan Kaufmann, 1996

[DICOM93] American College of Radiology, National Electrical Manufacturers Association: *"Digital Imaging and Communications in Medicine (DICOM): Version 3.0"*, Draft Standard, ACR-NEMA Committee, Working Group VI, Washington, DC, 1993.

[FOLEY+93] J. Foley, A. van Dam, S. Feiner, J. Hughes: *"Introduction to Computer Graphics"*, Reading: Addison-Wesley, 1993

[FOOTEYOD96] B. Foote, J. Yoder: *"Attracting Reuse"*, in: R. Martin, D. Riehle, F. Buschmann (eds.): Pattern Languages of Program Design 3, Reading: Addison-Wesley, 1996

[FÖRSTNER91] W. Förstner: *"Statistische Verfahren für die automatische Bildanalyse und ihre Bewertung bei der Objekterkennung und -vermessung"*, Habilitationsschrift, Fakultät für Bauingenieur- und Vermessungswesen der Universität Stuttgart, erschienen als: Veröffentlichungen der Deutschen Geodätischen Kommission, Reihe C, Heft 370, 1991

[FÖRSTNER94] W. Förstner: *"A Framework for Low-Level Feature Extraction"*, in: J. O. Eklundh (ed.): Computer Vision - European Conf. on Computer Vision ´94, Vol. II, pp. 383–394, Lecture Notes in Computer Science vol. 802, Berlin: Springer, 1994

[FREEMAN61] H. Freeman: *"On the encoding of arbitrary geometric configurations"*, IEE Trans. on Electronic Computers, Vol. EC-10, pp. 260-268, 1961

[FUCHSFÖRST95] C. Fuchs, W. Förstner: *"Polymorphic Grouping for Image Segmentation"*, in: ICCV '95, Proc. 5th Intl. Conf. on Computer Vision, Los Alamitos: IEEE Computer Society Press, 1995

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *"Design Patterns"*, Reading: Addison-Wesley, 1994

[GROSSLAT95] A. Gross, L. Latecki: *"Digitizations Preserving Topological and Differential Geometric Properties"*, J. Computer Vision and Image Understanding, 62(3), pp. 370-381, 1995

[HABER91] P. Haberäcker: *"Digitale Bildverarbeitung"*, München: Hanser, 1991

[HANSRISE88] A. Hanson, E. Riseman: *"The Visions Image-Understanding System"*, in: C. Brown: Advances in Computer Vision, vol. 1, Hillsdale: Lawrence Erlbaum Associates, 1988

[HAROSSH93] W. Harrison, H. Ossher: *"Subject-Oriented Programming: A Critique of Pure Objects"*, OOPSLA '93, Proc. Intl. Conf. Object-Oriented Programming Systems, Languages, and Applications, New York: ACM Press, 1993

[HARSHAP92] R. Haralick, L. Shapiro: *"Computer and Robot Vision"*, vol. 1, Reading: Addison-Wesley, 1992

[HARSTEV88] C. Harris, M. Stevens: *"A Combined Corner and Edge Detector"*, Proc. 4th Alvey Vision Conference, pp. 147-151, 1998

[HASKELL+96] B.G. Haskell, A. Puri, A.N. Netravali: *"Digital Video : An Introduction to MPEG-2"*, New York: Chapman & Hall, 1996

[HORN86] B.K.P. Horn: *"Robot Vision"*, Cambridge: MIT Press, 1986

[HORUS97] W. Eckstein, C.Steger: *"Architecture for computer vision application development within the HORUS system"*, J. Electronic Imaging, 6(2), pp. 244-261, 1997

[IPRS93] IPRS Development Team: *"Image and Pattern Recognition System (IPRS) User and Programmer Manual"*, University of Melbourne, Machine Vision Laboratory, 1993 (http://www.cs.curtin.edu.au/~cdillon/ftp.html)

[IUE98] *"Image Understanding Environment Documentation, Version 3.1"*, Amerinex Applied Imaging Inc., 1996-98 (http://www.aai.com/AAI/IUE/IUE.html)

[JÄHNE91] B. Jähne: *"Digitale Bildverarbeitung"*, Zweite Auflage, Berlin: Springer, 1991

[JONES96] N.D. Jones: *"An introduction to partial evaluation"*, ACM Computing Surveys, vol. 28, pp. 480-503, 1996

[KETTNER98] L. Kettner: *"Designing a Data Structure for Polyhedral Surfaces"*, Proc. 14th ACM Symp. on Computational Geometry, New York: ACM Press, 1998

[KHALIMSKY+90] E. Khalimsky, R. Kopperman, P. Meyer: *"Computer Graphics and Connected Topologies on Finite Ordered Sets"*, J. Topology and its Applications, vol. 36, pp. 1-27, 1990

[KHOROS91] The Khoros Group: *"Khoros 1.0 Programmer's Manual"*, Department of Electrical and Computer Engineering, University of New Mexico, 1991

[KOELSMEUL95] D. Koelma, A. Smeulders: *"An Image Processing Library based on Abstract Image Data-types in C++"*, in: C. Braccini et al. (eds.): Proc. of 8th Intl. Conf. on Image Analysis and Processing, Lecture Notes in Computer Science vol. 974, Berlin: Springer, 1995

[KOENDER90] J. Koenderink: *"Solid Shape"*, Cambridge: The MIT Press, 1990

[KOLLER+93] D. Koller, K. Daniilidis, H.H. Nagel: *"Model-based object tracking in monocular image sequences of road traffic scenes"*, Intl. J. of Computer Vision, 10(3), pp. 257-281, 1993

[KÖTHE95] U. Köthe: *"Primary Image Segmentation"*, in: G. Sagerer, S. Posch, F. Kummert (Hrsg.): Mustererkennung 1995, 17. DAGM-Symposium,
pp. 554-561, Berlin: Springer, 1995

[KÖTHE96] U. Köthe: *"Local Appropriate Scale in Morphological Scale-Space"*, in: B. Buxton, R. Cipolla: Computer Vision – ECCV '96, Proc. 4th European Conference on Computer Vision, vol. 1, Lecture Notes in Computer Science vol. 1064, pp. 219-228, Berlin: Springer, 1996

[KÖTHE98] U. Köthe: *"Design Patterns for Independent Building Blocks"*, in: J. Coldewey, P. Dyson (eds.): EuroPloP '98, Proc. of 3rd European Conf. on Pattern Languages of Programming and Computing 1998, pp. 143-165, Konstanz: Universitätsverlag Konstanz, 1999

[KÖTHE99] U. Köthe: *"Reusable Software in Computer Vision"*, in: B. Jähne, H. Haußecker, P. Geißler (eds.): Handbook of Computer Vision and Applications, Volume 3: Systems and Applications, pp. 103-132,
San Diego: Academic Press, 1999

[KÖTHE99A] U. Köthe: *"Expression Templates for Automatic Functor Creation"*, Fraunhofer-Institut für Graphische Datenverarbeitung Rostock, Technical Report 99I002-FEGD, 1999

[KOVALEVSKY89] V. Kovalevsky: *"Finite Topology as Applied to Image Analysis"*, Computer Vision, Graphics, and Image Processing, 46(2), pp. 141-161, 1989

[KROPATSCH95] W. Kropatsch: *"Building Irregular Pyramids by Dual Graph Contraction"*, IEE Proceedings / Vision, Image, and Signal Processing, 142(6), pp. 366-374, 1995

[KÜHLWEIHE97] D. Kühl, K. Weihe: *"Data Access Templates"*, C++ Report Magazine, July/August 1997

[LANSECK92] S. Lanser, W. Eckstein: *"A Modification of Deriche's Approach to Edge Detection"*, in: Proc. 11th IAPR Intl. Conf. Pattern Recognition, vol. 3, Los Alamitos: IEEE Computer Society Press, 1992

[LATECKI+95] L. Latecki, U. Eckhardt, A. Rosenfeld: *"Well-Composed Sets"*, Computer Vision and Image Understanding, 61(1), pp. 70-83, 1995

[LATECKI98] L. Latecki: *"Discrete Representation of Spatial Objects in Computer Vision"*, Dordrecht : Kluwer Academic Publishers, 1998

[LEAVMILL98] G. Leavens, T. Millstein: *"Multiple Dispatch as Dispatch on Tuples"*, in: OOPSLA '98, Proc. Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications, New York: ACM Press, 1998

[LIEBERH+88] K. Lieberherr, I. Holland, G. Lee, A. Riel: *"An Objective Sense of Style"*, IEEE Computer Magazine, 21(6), 1988

[LINDEBERG98] T. Lindeberg: *"Edge Detection and Ridge Detection with Automatic Scale Selection"*, Intl. Journal Computer Vision, 30(2), pp. 117-154, 1998

[LINDEHAAR94] T. Lindeberg, B. ter Haar Romeny: *"Linear Scale-Space"*, in: B. ter Haar Romeny (ed.): Geometry-Driven Diffusion in Computer Vision, Dordrecht: Kluwer Academic Publishers, 1994

[LISKOV88] B. Liskov: *"Data Abstraction and Hierarchy"*, SIGPLAN Notices, 23(5), 1988

[LUHMANN96] T. Luhmann: *"Results of the German Comparison Test for Digital Point Operators"*, in: Intl. Archives of Photogrammetry and Remote Sensing, vol. XXXI, Part B5, Vienna, 1996

[MAES88] P. Maes: *"Computational Reflection"*, The Knowledge Engineering Review, 3(3), pp. 1-19, 1988

[MÄNTYLÄ88] M. Mäntylä: *"An Introduction to Solid Modeling"*, Rockville: Computer Science Press, 1988

[MARR82] D. Marr: *"Vision"*, San Francisco: Freeman, 1982

[MARTIN95] R. Martin: *"Designing Object-Oriented C++ Applications Using the Booch Method"*, New York: Prentice Hall, 1995

[MARTIN96] R. Martin: *"The Interface Segregation Principle"*, C++ Report Magazine, August 1996

[MEYER97] B. Meyer: *"Object-Oriented Software Construction"*, Second Edition, New York: Prentice Hall, 1997

[MOHNEVAT92] R. Mohan, R. Nevatia: *"Perceptual organization for scene segmentation and description"*, IEEE Trans. Pattern Analysis and Machine Intelligence, 14(6), pp. 616-635, 1992

[MONTAN+91] A. Montanvert, P. Meer, A. Rosenfeld: *"Hierarchical image analysis using irregular tesselations"*, IEEE Trans. Pattern Analysis and Machine Intelligence, 13(4), pp. 307-316, 1991

[MUSSAINI96] D. Musser, A. Saini: *"STL Tutorial and Reference Guide"*, Reading: Addison-Wesley, 1996

[MUSSTEP89] D. Musser, A. Stepanov: *"Generic Programming"*, 1st Intl. Joint Conf. of ISSAC-88 (Intl. Symp. on Symbolic and Algebraic Computation) and AAECC-6 (Applied Algebra, Algebraic Algorithms, and Error Correcting Codes), Lecture Notes in Computer Science vol. 358, Berlin: Springer, 1989

[MUSSTEP94] D. Musser, A. Stepanov: *"Algorithm-Oriented Generic Libraries"*, Software - Practice and Experience, 24(7), pp. 623-642, 1994

[MYERS95] N. Myers: *"A New and Useful Template Technique: Traits"*, C++ Report Magazine, June 1995

[OMG98] Object Management Group: *"The Common Object Request Broker: Architecture and Specification"*, Revision 2.2, 1998

[PAEPCKE93] A. Paepcke: *"Object-Oriented Programming: The CLOS Perspective"*, Cambridge: MIT Press, 1993

[PARNAS72] D. Parnas: *"On the Criteria to be Used in Decomposing Systems into Modules"*, Communications of the ACM, 15(12), pp. 1053-1058, 1972

[PARNAS+85] D. Parnas, P. Clements, D. Weiss: *"The Modular Structure of Complex Systems"*, IEEE Trans. Software Engineering, 11(3), 1985

[PAVLIDIS77] T. Pavlidis: *"Structural Pattern Recognition"*, New York: Springer, 1977

[PAVLIDIS82] T. Pavlidis: *"Algorithms for Graphics and Image Processing"*, Berlin: Springer, 1982

[PIKS94] International Organization for Standardization (ISO): *"International Standard 12087: Image Processing and Interchange Standard"*, 1994

[POLLARD+97] S. Pollard, J. Porrill, N. Thacker: *"Tina Programmer's Guide"*, Electronic Systems Group, University of Sheffield, 1997 (http://www.shef.ac.uk/uni/academic/D-H/eee/esg/research/Tina.html)

[PRATT95] W. Pratt: *"PIKS foundation C programmer's guide"*, New York: Prentice Hall, 1995

[PREE97] W. Pree: *"Essential Framework Design Patterns"*, Object Magazine, 7(1), March 1997

[PTAK+97] P. Ptak, H. Kofler, W. Kropatsch: *"Digital Topologies Revisited: An Approach Based on the Topological Point-Neighborhood"*, in: E. Ahronovitz, C. Fiorio (eds.): Discrete Geometry for Computer Imagery, Proc. 7th Intl Workshop DGCI '97, Lecture Notes in Computer Science vol. 1347, Berlin: Springer, 1997

[RASKUB94] J. Rasure, S. Kubica: *"The Khoros Application Development Environment"*, in: H.I Christensen and J.L Crowley (eds.): Experimental Environments for Computer Vision and Image Processing, Singapore: World Scientific Press,1994

[REDMOND97] F. E. Redmond: *"DCOM: Microsoft Distributed Component Object Model"*, Foster City: IDG Books Worldwide, 1997

[REENS+96] T. Reenskaug, P. Wold, O.A. Lehne: *"Working with Objects"*, New York: Prentice Hall, 1996

[RIEHLEGRO98] D. Riehle, T. Gross: *"Role Model Based Framework Design and Integration"*, OOPSLA '98, Proc. Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications, New York: ACM Press, 1998

[ROHR92] K. Rohr: *"Recognizing Corners by Fitting Parametric Models"*, Intl. J. of Computer Vision, 9(3), pp. 213-230, 1992

[ROHR94] K. Rohr: *"Localization Properties of Direct Corner Detectors"*, J. of Mathematical Imaging and Vision, 4(2), pp. 139-150, 1994

[ROSENFELD74] A. Rosenfeld: *"Adjacency in Digital Pictures"*, Information and Control vol. 26, pp. 24-33, 1974

[SARKBOY93] S. Sarkar, K. Boyer: *"Perceptual Grouping in Computer Vision"*, IEEE Trans. On Systems, Man, and Cybernetics, 23(2), pp. 382-399, 1993

[SERRA82] J. Serra: *"Image Analysis and Mathematical Morphology"*, Vol. 1, London: Academic Press, 1982

[SHALIT96] A. Shalit: *"The Dylan Reference Manual"*, Reading: Addison-Wesley, 1996

[SHENCAST92] J. Shen, S. Castan: *"An Optimal Linear Operator for Step Edge Detection"*, CVGIP: Graphical Models and Image Processing, 54(2), pp. 112-133, 1992

[STEPANOV95] Al Stevens: *"Interview with Alexander Stepanov"*, Dr. Dobbs Journal, March 1995

[STEVENS+74] W. Stevens, G. Myers, L. Constantine: *"Structured Design"*, IBM Systems Journal, vol. 13, pp. 115-139, 1974

[STIEHL90] H.S. Stiehl: *"Issues of Spatial Representation in Computational Vision"*, in: C. Freksa, C. Habel (Hrsg.): Repräsentation und Verarbeitung räumlichen Wissens, pp. 83-98, Berlin: Springer, 1990

[SZYPERSKI96] C. Szyperski: *"Independently Extensible Systems – Software Engineering Potential and Challenges"*, in: K. Ramamohanarao (ed.): Proc. of 19th Australasian Computer Science Conference, Townsville: Australian Computer Science Communications, 1996

[SZYPERSKI97] C. Szyperski: *"Component Software"*, Harlow: Addison Wesley Longman, 1997

[TARGETJR96] *"TargetJr Documentation"* , The TargetJr Community, 1996-98, (http://www.targetjr.org/)

[TIECKGERL97] S. Tieck, S. Gerloff: *"Bildsegmentierung mittels Wasserscheidentransformation und interaktivem graphbasiertem Regionenwachstum"*, Diplomarbeit, Fachbereich Informatik der Universität Hamburg, 1997

[TOKBAT99] L. Tokuda, D. Batory: *"Evolving Object-Oriented Architectures with Refactorings"*, to appear in: Automated Software Engineering, 1999 (http://www.cs.utexas.edu/users/schwartz/pub.htm)

[UDELL94] J. Udell: *"ComponentWare"*, BYTE Magazine, 19(5), pp. 46-56, 1994

[VANBALEN+94] R. van Balen, D. Koelma, T.K. ten Kate, B. Mosterd, A. Smeulders: *"ScilImage: A Multi-layered Environment for Use and Development of Image Processing Software"*, in: I. Christensen, J.L. Cowley (eds.): Experimental Environments for Computer Vision and Image Processing, pp. 107-126, Singapore: World Scientific Press, 1994

[VELD97] T. Veldhuizen: *"User's Guide of the Blitz++ Numerical Library"*, 1997, (http://monet.uwaterloo.ca/blitz/manual/blitz.html)

[VELDGANN98] T. Veldhuizen, D. Gannon: *"Active Libraries: Rethinking the roles of compilers and libraries"*, in: Proc. SIAM Workshop on Object-Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), Yorktown Heights: SIAM Press, 1998

[VIERGEVER+99] M .Viergever, H.S. Stiehl, R. Klette, R.M. Haralick (eds.): *"Evaluation and Validation of Computer Vision Algorithms"*, Dordrecht: Kluwer Academic Publishers, 1999 (in press)

[VINCSOILLE91] L. Vincent, P. Soille: *"Watersheds in digital spaces: an efficient algorithm based on immersion simulations"*, IEEE Trans. Pattern Analysis and Machine Intelligence, 13(6), pp. 583-598, 1991

[WEIDE+96] B. Weide, S. Edwards, W. Heym, T. Long, W. Ogden: *"Characterizing Observability and Controllability of Software Components"*, in: M. Sitaraman (ed.): Proc. 4th Intl. Conf. Software Reuse, Los Alamitos: IEEE Computer Society Press, 1996

[WILLKROP94] D. Willersinn, W. Kropatsch: *"Dual graph contraction for irregular pyramids"*, in: Proceedings 12th Intl. Conf. Pattern Recognition, Conference D, pp. 251-256, Los Alamitos: IEEE Computer Society Press, 1994

[WILSON72] R. Wilson: *"Introduction to Graph Theory"*, Edinburgh: Oliver & Boyd, 1972

[WINTER95] S. Winter: *"Topological Relations between Discrete Regions"*, in: M. Egenhofer, J. Herring (eds.): Advances in Spatial Databases, pp. 310-327, Lecture Notes in Computer Science vol. 951, Berlin: Springer, 1995

[WIRTH71] N. Wirth: *"Program Development by Stepwise Refinement"*, Communications of the ACM, 14(4), pp. 221-227, 1971

[WITKIN83] A.P. Witkin: *"Scale Space Filtering"*, in: A. Bundy (ed.): Proc. 8th Intl. Joint Conf. on Artificial Intelligence, pp. 1019-1023, Los Altos: Kaufmann, 1983

[WOLBERG90] G. Wolberg: *"Digital Image Warping"*, Los Alamitos: IEEE Computer Society Press, 1990

Index