# Ein System zur schnellen Entwicklung von Bildverarbeitungsalgorithmen

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Dipl.-Ing., Dipl.-Inform. Helmut Herrmann
aus Hanau

Mannheim, 2004

| | |
|---|---|
| Dekan: | Professor Dr. Jürgen Potthoff, Universität Mannheim |
| Referent: | Professor Dr. Bernd Jähne, Universität Heidelberg |
| Korreferent: | Professor Dr. Reinhard Männer, Universität Mannheim |

Tag der mündlichen Prüfung: 12. Juli 2004

# Zusammenfassung

Die automatische Auswertung von Bildern ist zu einem wichtigen Instrument sowohl in der Wissenschaft als auch in der Industrie geworden. Durch die benötigte Flexibilität einerseits und die gesteigerte Leistungsfähigkeit moderner Mikroprozessoren andererseits geschieht Bildverarbeitung heute verstärkt auf Multifunktionsrechnern statt auf spezieller Bildverarbeitungshardware. Als Folge dessen werden Software-Werkzeuge zur Entwicklung von Bildverarbeitungsanwendungen benötigt. Die vorliegende Arbeit beschäftigt sich mit einem Softwaresystem zur schnellen Erstellung von Bildverarbeitungsalgorithmen. Mit der Software *heurisko* wird ein System entwickelt, das dem Anwender durch die Kombination einer problemorientierten Skriptsprache und einer graphischen Entwicklungsumgebung mit einer Bibliothek mit Bildverarbeitungsoperatoren ein effizientes Werkzeug an die Hand gibt. Dazu wurden eine universelle Datenstruktur für die multidimensionale Low- und High-Level-Bildverarbeitung und ein hierarchisches Operatorkonzept eingeführt, das datentypabhängige und -unabhängige Operatoren sorgfältig trennt. Der interne hierarchische Aufbau und die sorgfältige Klassenbildung von Bildverarbeitungsalgorithmen erlauben einen hohen Grad an Wiederverwendung von Code und erhöhen die Wirksamkeit von partiellen hardwarenahen Optimierungen. Die Modularität, insbesondere die klare Trennung zwischen Benutzerschnittstelle und Algorithmik, und offene Schnittstellen des Systems für Bildverarbeitungsfunktionen, Datenakquisition und Datenein- und –ausgabe eröffnen dem Anwender die Möglichkeit, eigene Erweiterungen vorzunehmen. Die Portabilität sorgt dafür, dass das System auf vielen Plattformen lauffähig ist. Durch die Erzeugung eines binären Zwischencodes wird der Geschwindigkeitsnachteil des interpretierenden Systems gegenüber einem kompilierenden System praktisch aufgehoben. Mit Anwendungsbeispielen von der Echtzeitbilderfassung bis zur Restaurierung von Bildfolgen wird demonstriert, wie mit dem System verschiedene Aufgaben effektiv gelöst werden können.

# Abstract

Automatic analysis of images has become an important instrument both in science and industry. The demand for flexible solutions in combination with the ever increasing computing power of modern microprocessors made it possible to solve more and more image processing problems on PCs instead of dedicated hardware. Therefore software tools for fast and flexible development of image algorithms and applications are required. This dissertation deals with a software system for fast development of image processing algorithms combining a problem-oriented script language with a library of image processing functions. A universal data structure for multidimensional low level and high level image processing has been developed and a hierarchical operator concept that carefully separates data type-dependent from data type-independent operations. By a hierarchical design of the image processing algorithms and the operator classes a high degree of code reusability has been achieved. Furthermore hand coded optimization has become feasible because it is limited to a small fraction of the code. The system is highly modular. There is a strict separation between the interchangeable user interface and the image processing kernel. Open interfaces have been designed for algorithms, for data input and output and for acquisition of images and time series that allow users to extend the system. The portable code assures that the system runs on many platforms. The generation of a binary intermediate code by the interpreter overcomes the speed disadvantage of interpreting systems. A number of applications including such diverse tasks as real-time image recording up to restoration of image sequences demonstrate the power and flexibility of the system.

# Inhalt

# 1 Einleitung

## 1.1 Hintergrund

„Ein Bild sagt mehr als tausend Worte“ ist sicherlich eines der bekanntesten Sprichworte und weist hin auf die Bedeutung von Bildern für uns Menschen. Tatsächlich benutzen wir im täglichen Leben immer wieder Bilder. Aus dem Urlaub senden wir eine Ansichtskarte und zeigen den Daheimgebliebenen, in welcher schönen Umgebung wir uns erholen. Wenn wir eine neue Einbauküche planen, erstellen wir eine Maßskizze unserer Küche, bevor wir ins Möbelhaus gehen. Bei Autofahrten in unbekannte Gegenden orientieren wir uns mit Hilfe von Landkarten. Und unvergessliche Momente unseres Lebens halten wir in Photos fest. Da wundert es nicht, dass Bilder auch in der Industrie, in der Wissenschaft und in anderen Bereichen wie z. B. in der Medizin eine wichtige Rolle spielen. Dabei ist festzustellen, dass Bilder zum Erkennen, Überwachen und Messen eingesetzt werden und zunehmend andere Sensoren ersetzen, so dass man das zitierte Sprichwort mit dem Zusatz „… und mehr als herkömmliche Sensoren.“ versehen könnte. Bei einem *herkömmlichen* Sensor kann man sich etwa ein Thermometer zur Temperaturmessung, einen mechanischen Taster zur Abtastung einer Objektoberfläche oder die Finger eines Arztes bei einer medizinischen Untersuchung vorstellen, die man heute durch eine Wärmebildkamera, durch ein Kamerasystem zur Erfassung von 3-D-Daten bzw. durch ein Ultraschalldiagnosegerät ersetzen oder ergänzen kann. Wie man aus diesen drei Beispielen schon sieht, ist die Vielfalt der bildgebenden Sensoren sehr groß. Zudem wurden viele Verfahren entwickelt, bei denen ein primärer, eigentlich strahlungsloser Vorgang durch einen Trick in einem sekundären Vorgang Strahlung erzeugt, welche von einem passenden Detektor erfasst und zur Beobachtung des primären Vorgangs benutzt wird.

Der Begriff *herkömmlicher Sensor* sollte hier nicht allzu streng verstanden werden, da Bilder nicht erst seit der Erfindung der Photographie oder seit der Entdeckung der Röntgenstrahlung für Messungen im weitesten Sinne zur Hilfe genommen werden. Vergleichsweise neu ist jedoch die automatische, maschinelle Auswertung von Bildern, wobei man heute auf Grund der verfügbaren hohen Rechenleistung auf Standardrechnern immer mehr von speziellen Hardware- auf reine Software-Lösungen übergeht. Diese automatische Auswertung von Bildern mit Computern soll im Folgenden mit dem Begriff *Bildverarbeitung* gemeint sein. Die Entwicklung einer Bildverarbeitungslösung ist in diesem Sinne die Erstellung eines Programms. Bildverarbeitung umfasst dabei sowohl die Umformung eines Bildes in ein anderes Bild als auch die Gewinnung nicht bildhafter Information durch Analyse des Bildinhaltes. Ein anderer, international gebräuchlicher Begriff für Bildverarbeitung ist beispielsweise *Computer Vision* [Köthe 2000]. Um den Unterschied zum menschlichen Sehen herauszustellen, wird auch von *maschinellem Sehen* oder von *Machine Vision* gesprochen, wobei diese Begriffe eher im industriellen Umfeld üblich sind.

Das oben zitierte Sprichwort sagt zum einen etwas aus über die Bedeutung von Bildern, zum anderen aber auch über die in einem Bild mögliche Informationsmenge. Während ein übliches Thermometer zu einem Zeitpunkt nur einen einzigen Temperaturwert liefert, kann das Bild einer Infrarotkamera eine räumliche Temperaturverteilung messen. Nicht immer ist mit Bildern, auf welch raffinierte Weise auch immer sie erzeugt wurden, eine Bildverarbeitung im hier definierten Sinne nötig. So kann durch die Erkennungsleistung des menschlichen Sehsystems der Zahnarzt allein bei Betrachtung einer Röntgenaufnahme seine Schlüsse ziehen oder die Feuerwehr mit Hilfe einer Wärmebildkamera einen Brandherd orten. Die

Bildverarbeitung kommt aber immer dann ins Spiel, wenn Bilder automatisch ausgewertet werden sollen, wenn der Informationsgehalt von Bildern zu umfangreich wird oder von Menschen naturbedingt nicht extrahiert werden kann. Einige interessante Anwendungen für Bildverarbeitung in der Wissenschaft findet man im ersten Kapitel von [Jähne 2004] und Beispiellösungen aus der Industrie in [Jähne et. al. 1995].

Aus den bisherigen Überlegungen wird schon deutlich, dass Bildverarbeitung komplex und divers ist. Komplex ist sie, weil Bilder eine Fülle von Information enthalten, aus der die interessierende Information erst extrahiert werden muss. Divers ist sie, weil es so weit gefächerte Bilderzeugungsmöglichkeiten und so unterschiedliche Anwendungsgebiete gibt, dass viele Wissensgebiete gleichzeitig involviert sind. Bildverarbeitung ist außerdem ein hierarchischer Prozess, wie die aus [Jähne 2002] stammende Abbildung 1.1 zeigt. Demnach beginnt die Lösung einer Bildverarbeitungsaufgabe mit der Bilderzeugung. Die Digitalisierung im nächsten Schritt ist der Einstieg in die Bildverarbeitungssoftware. Auf der ersten Verarbeitungsstufe findet die Bildvorverarbeitung statt. Es folgend mehrere Schritte zur Segmentierung der interessierenden Objekte von ihrer Umgebung. Danach werden diese Objekte analysiert und klassifiziert. Nicht jede Bildverarbeitungsaufgabe erfordert alle oder die gleichen Verarbeitungsstufen, aber trotzdem wiederholen sich die einzelnen Schritte immer wieder. Die entsprechenden Softwaremodule lassen sich also bei geeigneter Programmierung wieder verwenden.

Zur Verdeutlichung der obigen Feststellungen seien beispielhaft drei Bildverarbeitungsaufgaben skizziert. Betrachtet sei als erstes die Aufgabe, bei einem Kamerahersteller die Qualitätskontrolle mit Hilfe der Bildverarbeitung zu verbessern. Die gesteckten Ziele sind unter anderen, durch objektive Messwerte Justierungen unabhängig vom subjektiven Empfinden eines Menschen mit konstanter Qualität vorzunehmen, dem menschlichen Auge versteckte Mängel sichtbar und damit auch behebbar zu machen und eine objektive Protokollierung von Kennwerten während der gesamten Lebensdauer der Kamera zu ermöglichen. Um die gestellte Bildverarbeitungsaufgabe erledigen zu können, muss sich der Entwickler der Algorithmen mit Kameratechnik auseinander setzen. Denn nur dann kann er verstehen, welche Bilder er unter welchen Bedingungen aufnehmen muss und wonach er in ihnen suchen soll. Eine interessante Teilaufgabe ist, die Modulationstransferfunktion (MTF) einer Kamera zu messen. Die MTF ist die Übertragungsfunktion der Amplitude des Ausgangssignals in Abhängigkeit von der Wellenlänge der räumlichen Verteilung des einfallenden Lichtsignals. Es ist klar, dass auf Grund der geometrischen Gegebenheiten der Bildpunkte eines Bildsensors nach dem Abtasttheorem die theoretische Obergrenze bei der Wellenlänge liegt, die zwei Bildpunkten entspricht. Praktisch erfahren jedoch bereits Signale mit größerer Wellenlänge eine Abschwächung der Amplitude. Um die MTF möglichst genau und auch schnell messen zu können, muss man sich ein geeignetes Testmuster überlegen, am besten eines, in dem alle interessierenden Wellenlängen vorkommen.

Eine andere Beispielaufgabe ist die Rekonstruktion von Bildern in der Forensik oder im Rahmen polizeilicher Ermittlungstätigkeit. Es könnte darum gehen, aus Videoaufnahmen bewegter Objekte mit Bewegungsunschärfe Schriften, Gesichter oder anderes zu rekonstruieren. Hier muss zunächst grundsätzlich festgestellt werden, dass einmal verlorene Information durch kein noch so raffiniertes Verfahren wieder gewonnen werden kann. Es muss also darum gehen, aus dem vorliegenden Bildmaterial Information so geschickt zusammenzusuchen, dass zumindest ein Teil der ursprünglichen Bildinformation wieder erkennbar wird. Bei Bildsequenzen kann man sich zunutze machen, dass aus der Beziehung zwischen den einzelnen Bildern die Schätzung der Bewegung möglich ist. Mit Hilfe dieser Information kann man versuchen, eine durch die Bewegung verursachte Bildunschärfe zu verringern.

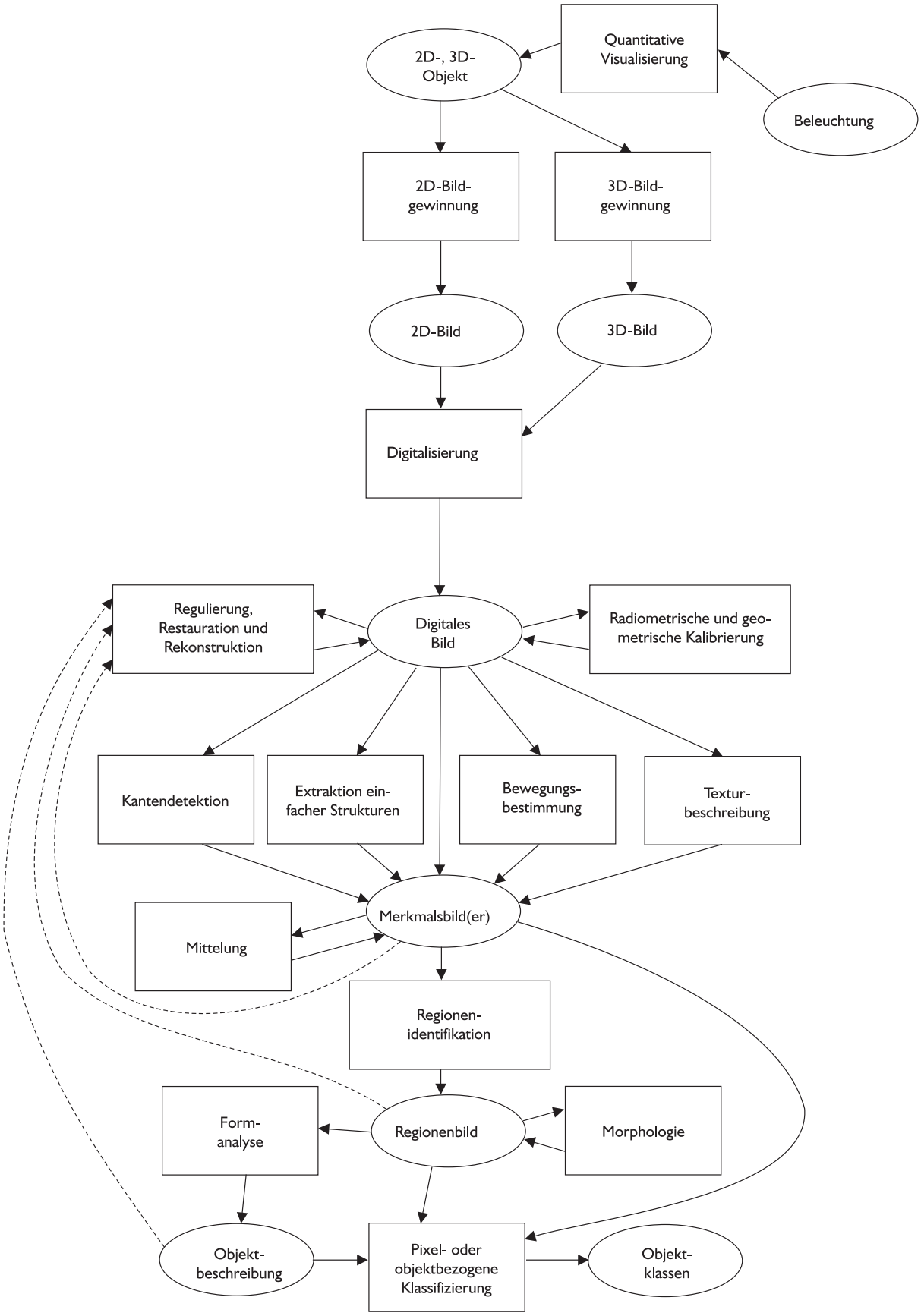


***Abbildung 1.1:*** *Hierarchie der Bildverarbeitungsoperationen von der Bilderzeugung bis zur Bildauswertung (mit freundlicher Genehmigung des Autors von [Jähne 2002])*

Ein drittes Beispiel ist die Simulation des menschlichen Sehens durch das Betrachten von projizierten Bildern in Fahrzeugen mit einer Stereokameraanordnung, um von menschlichen Betrachtern berichteten Phänomenen auf die Spur zu kommen. Eines der Phänomene ist, dass die Betrachter nicht mehr in der Lage sind, die Bilder der beiden Augen zu fusionieren; sie sehen Doppelbilder. Mit dem Stereosystem kann man gezielt die Unterschiede und Verzerrungen zwischen linkem und rechtem Bild messen und feststellen, unter welchen Bedingungen Doppelbilder gesehen werden. Hier besteht für die Bildverarbeitung das Hauptproblem, korrespondierende Bildpunkte in den beiden Stereobildern zu finden.

Diese wenigen Beispiele zeigen, wie weit gefächert die Aufgaben in der Bildverarbeitung sind. Eine aktuelle systematische Übersicht über Bildverarbeitungssysteme und Anwendungen in der Industrie mit vielen Literaturhinweisen wird in dem Artikel [Malamas et. al. 2003] geboten.

## 1.2 Aufgabenstellung und Überblick

Zielsetzung dieser Arbeit ist es, einen möglichst umfassenden Anforderungskatalog an ein Bildverarbeitungssystem zu formulieren und daraus die Bildverarbeitungssoftware heurisko zu entwickeln, so dass sie diesen Anforderungen genügt. Da Bildverarbeitung heute weit verbreitet auf PCs geschieht und auch alle Implementierungen und Tests im Rahmen dieser Arbeit auf diesem Rechnertyp stattgefunden haben, beschäftigt sich Kapitel 2 mit ein paar für das Folgende relevanten Aspekten der PCs. Darüber hinaus wird im gleichen Kapitel schon vorab ein wenig in die Bildverarbeitungssoftware heurisko eingeführt, weil sie als Plattform für alle aufgeführten Messungen gedient hat. Kapitel 3 gibt dann einen Querschnitt von Bildverarbeitungssystemen aus der Literatur wieder und extrahiert daraus die dort zu findenden Lösungskonzepte. In Kapitel 4 werden diese Lösungskonzepte wieder aufgegriffen und als Grundlage für die Entwicklung von heurisko diskutiert. Anschließend stellt Kapitel 5 Optimierungen einiger Teilaspekte von heuriskos bezüglich der zuvor formulierten Anforderungen vor, bevor in Kapitel 6 anhand von ausgewählten Beispielen die Tauglichkeit des entwickelten Softwarepaktes demonstriert wird. Abgeschlossen wird die Arbeit mit einem Ausblick in Kapitel 7.

# 2 Bildverarbeitung mit PC-Systemen

Mit dem Begriff *PC-System* sollen hier allgemein alle Allzweckrechner mit Mikroprozessoren gemeint sein. Es soll hier nicht nach PCs, IBM-PCs und Workstations und nicht nach Herstellern oder Betriebssystemen unterschieden werden. Ein PC ist deshalb ein *personal computer*, im Wesentlichen gedacht als Einbenutzersystem, von seiner Architektur her ohne Ausrichtung auf eine einzige *bestimmte* Anwendung, aufgebaut aus kostengünstigen Massenkomponenten und betrieben mit einem Standardbetriebssystem.

Wie vieles andere auch findet Bildverarbeitung heute sicherlich zum Großteil auf solchen Systemen statt. Außerdem ist bekannt, dass die Betriebssysteme der Firma Microsoft den Markt dominieren. Deshalb nimmt es nicht Wunder, dass Bildverarbeitungssoftware gerade für diesen Bereich angeboten wird und dass man sich mitunter mit Eigenheiten solcher Systeme beschäftigen muss. In diesem Kapitel wird auf Teilaspekte der Rechnerarchitektur eingegangen, so weit sie später eine Rolle spielen. Vorher wird der für alle Messungen und Tests benutzte Rechner vorgestellt und in die Bildverarbeitungssoftware heurisko eingeführt.

## 2.1 Testrechner

Im Weiteren werden ab und zu Laufzeiten für Bildverarbeitungsoperationen angegeben. Wenn nicht ausdrücklich anderes erwähnt ist, wurden diese Laufzeiten auf dem PC mit der Ausstattung gemäß Tabelle 2-1 ermittelt.

***Tabelle 2-1:*** *Testrechner mit aktuellem Prozessor*

| Testrechner | | |
|---|---|---|
| Hardware | Hauptplatine | Asus P4P800 |
| | Prozessor | Intel Pentium 4, Taktfrequenz 2,6 GHz, Sekundärer Cache 512 kB, Systembus 800 MHz |
| | RAM | 1 GB |
| Software | Betriebssystem | Microsoft Windows XP SP1 |
| | Entwicklungsumgebung | Microsoft Visual C++ 6.0 SP5 |
| | Bildverarbeitung | heurisko 5, erzeugt mit C-Compiler-Option /O2 für Geschwindigkeitsoptimierung |

***Tabelle 2-2:*** *Zweiter Testrechner mit Prozessor der vorigen Generation*

| Zweiter Testrechner | | |
|---|---|---|
| Hardware | Hauptplatine | Asus CUV4X |
| | Prozessor | Intel Pentium III, Taktfrequenz 800 MHz, Sekundärer Cache 256 kB, Systembus 133 MHz |
| | RAM | 512 MB |
| Software | Betriebssystem | Microsoft Windows 2000 SP4 |
| | Entwicklungsumgebung | auf erstem Testrechner, siehe oben |
| | Bildverarbeitung | siehe oben |

Der im Testsystem verwendete Prozessor Pentium 4 ist mit Hyper-Threading-Technologie ausgestattet [Intel]. Durch diese stellt sich der Prozessor einer Software mit mehreren parallelen Threads wie zwei CPUs dar. Die Ressourcen des Prozessors werden dadurch besser genutzt und die Leistungsfähigkeit entsprechend gesteigert. Das Hyper-Threading bringt allerdings keinerlei Vorteile beim Betreiben eines Programms ohne parallele Threads, sondern eher sogar kleine Nachteile. Deshalb war das Hyper-Threading in der Regel ausgeschaltet. Zum Vergleich mit einer etwas älteren Prozessorgeneration wurden teilweise auch Tests mit einem zweiten Rechner durchgeführt (siehe Tabelle 2-2).

## 2.2 Bildverarbeitungssoftware heurisko

Alle Implementierungen und Tests im Rahmen dieser Arbeit erfolgten in dem Bildverarbeitungspaket heurisko®, das als kommerzielles Produkt bei der Firma AEON in Hanau erhältlich ist [heurisko]. heurisko wird seit 1992 unter der wissenschaftlichen Betreuung von Prof. Bernd Jähne von der Universität Heidelberg entwickelt. An dieser Stelle soll heurisko nur kurz vorgestellt werden; viele weitere interessante Einzelheiten werden im weiteren Verlauf entsprechend der jeweils behandelten Thematik vertieft.

heurisko besteht im Kern aus einer modularen Bildverarbeitungsbibliothek, die über einen Interpreter dem Benutzer verfügbar gemacht ist. Das Programm wird mit einer graphischen Benutzeroberfläche geliefert (Abbildung 2.1), über die man interaktiv mit dem System arbeiten oder auch Skripte für automatische Abläufe erstellen kann. Lässt man die graphische Schnittstelle weg, kann man heurisko über die in C geschriebene Interpreterschnittstelle in andere Programmsysteme integrieren und so z. B. in der Automatisierung einsetzen.

Die Skripte werden in der heurisko-eigenen Skriptsprache, konzipiert für die Bedürfnisse der Bildverarbeitung, verfasst. Die wesentlichen Elemente dieser Sprache sind

- Anweisungen zur Erzeugung von Objekten entsprechend den Variablen in einem C-Programm,
- Operatoraufrufe entsprechend den Funktionsaufrufen in C-Programmen,
- Erstellung eigener Operatoren entsprechend den Funktionen in C-Programmen durch den Benutzer und
- Anweisungen zur Programmflusskontrolle wie bedingte Verzweigungen oder Programmschleifen.

Ein simples Beispiel soll zeigen, wie ein heurisko-Skript aussehen kann:

```
ubyte im1[480][640], im2[480][640]; # Bildobjekte im1 und im2
string file;                        # Zeichenkettenobjekt
file = RequestFiles();  # Dateidialog
if(file);               # wenn Dialog mit Auswahl verlassen
    im1 = Read(file);   # Einlesen der Datei in im1
    im2 = Bin4(im1);    # glätten und speichern in im2
endif;
```

Dieses Programm definiert zwei Bildobjekte `im1` und `im2` und ein Zeichenkettenobjekt `file`. Mit dem Operator `RequestFiles()` wird der Benutzer aufgefordert, über einen Standarddateidialog eine Bilddatei auszuwählen. Wurde eine Datei gewählt, wird sie in `im1` eingelesen und dann nach einer Glättung in `im2` gespeichert.

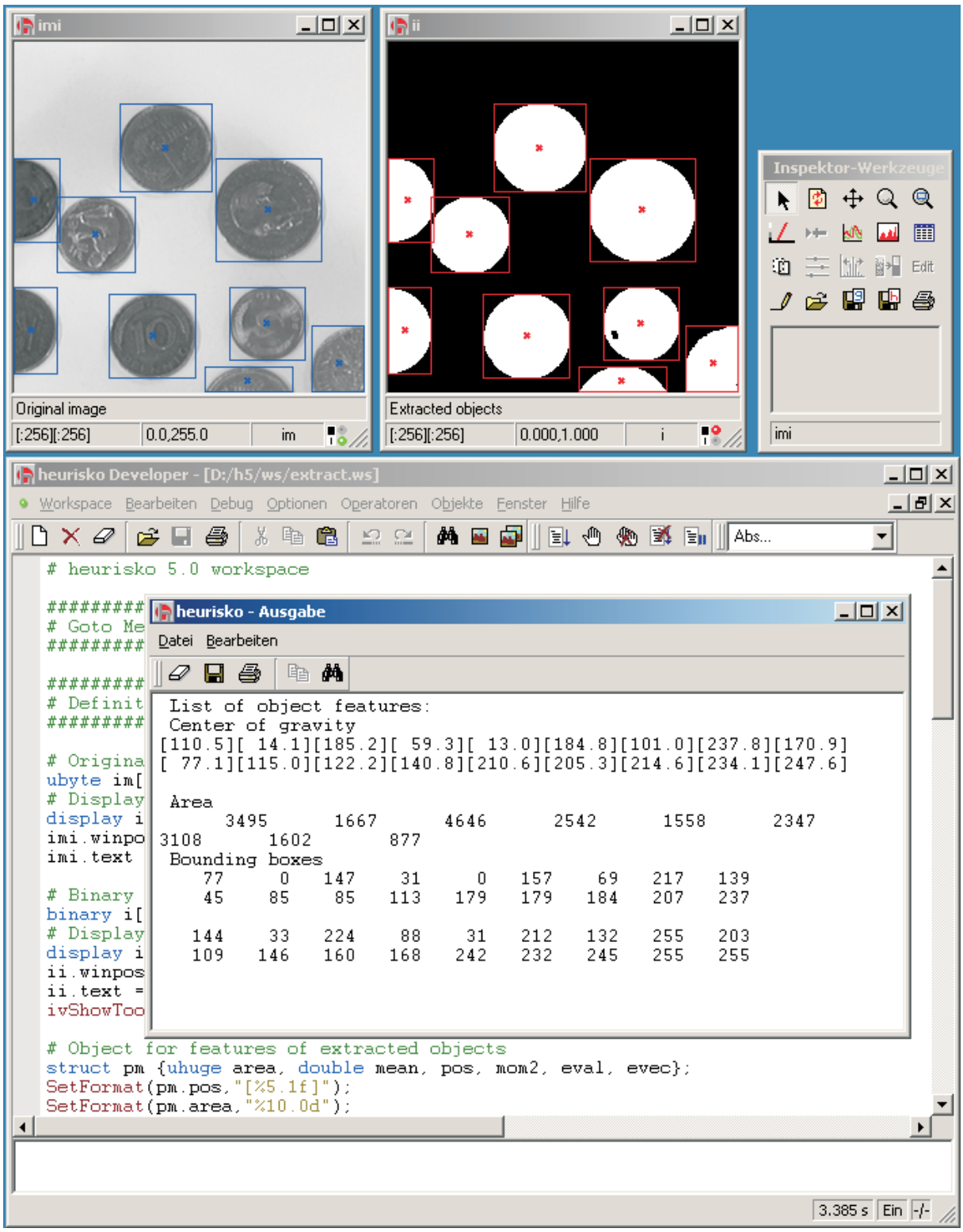


***Abbildung 2.1:*** *Die graphische Benutzeroberfläche von heurisko mit Hauptfenster, Ausgabefenster, zwei Inspektorfenstern und Inspektor-Werkzeugen*

Allgemein hat ein Operatoraufruf folgende Form:

$obj_1, \ldots, obj_n$ = Operator($obj_{n+1}, \ldots, obj_m$);

wobei $m \geq n$ gilt. Eine Besonderheit ist, dass links vom Gleichheitszeichen mehrere Objekte stehen können. Die links stehenden Objekte dürfen vom Operator verändert werden, die rechts stehenden dagegen nicht; sie sind reine Eingabeobjekte. Der Interpreter prüft für einen Operatoraufruf nur die korrekte Anzahl an Parameterobjekten, aber nicht die Art der verwendeten Objekte. Erst zur Laufzeit wertet die Implementierung des Operators die Objekte aus und prüft, ob sie tatsächlich verarbeitet werden können. Je nach Implementierung können Parameterobjekte viele Formen und Datentypen haben. So ist der Operator `Bin4()` im obigen Beispiel zurzeit für ein- bis dreidimensionale Objekte und für ganze Zahlen mit 8 und 16 Bit je Bildpunkt und für einfachgenaue Fließkommanzahlen implementiert.

Maßgeblich für die Entwicklung von heurisko waren und sind folgende Prinzipien:

- Plattformunabhängigkeit: Der gesamte Quellcode soll so weit wie möglich keine Plattformabhängigkeiten aufweisen. Insbesondere verfügt heurisko trotz mancher unvermeidlicher plattformabhängiger Ergänzungen über eine Kernversion mit der gesamten Bildverarbeitungsalgorithmik, die sich auf jedem beliebigen System mit einem ANSI/ISO-C-Compiler übersetzen und binden lässt.

- Erweiterbarkeit durch den Anwender: Kein Bildverarbeitungsprogramm wird je für sich beanspruchen können, vollständig bezüglich der gewünschten Funktionalität zu sein. Deshalb ist es unerlässlich, komfortable Erweiterungsmöglichkeiten für den Benutzer vorzusehen.

- Modularität: Eine modulare Architektur unterstützt die beiden zuvor genannten Anforderungen.

- effiziente Algorithmen: Das Leitmotiv bei jeder Implementierung soll neben der unverzichtbaren Korrektheit auch eine hohe Ausführungsgeschwindigkeit sein.

Am Schluss dieses groben Überblicks soll noch erwähnt sein, dass heurisko im Wesentlichen in ANSI/ISO C implementiert ist. Ausnahmen bilden die viel später nach dem Umstieg auf die plattformunabhängige Bibliothek Qt für die Programmierung von Benutzeroberflächen neu begonnenen Module für die Visualisierung und die graphische Bedienoberfläche, die in C++ programmiert sind. Der Grund der Entscheidung für C liegt darin, dass beim Startschuss von heurisko im Jahr 1992 im Hinblick auf Plattformunabhängigkeit C die beste Wahl schien. C-Compiler gab es fast für jede Plattform, und das mit Gnu C sogar frei. Auch für Bildverarbeitung wichtige Hilfsbibliotheken wie beispielsweise Treiber für Geräte zur Datenerfassung wurden damals mit C-Schnittstellen angeboten. Für C++ war dagegen noch kein Standard verabschiedet. Das soll an dieser Stelle erst einmal genügen. Weitere Einzelheiten folgen bei den behandelten Themen.

## 2.3 Einige Bemerkungen zur PC-Hardware

Es sei voraus geschickt, dass es hier nicht um eine erschöpfende Beschreibung einiger Aspekte der PC-Hardware geht, sondern vielmehr um eine kurze Darstellung, die es erlaubt, einige in dieser Arbeit erwähnten Ideen, Messungen und Beobachtungen besser zu verstehen.

### 2.3.1 Mikroprozessoren

Für die Bearbeitung von Instruktionen benötigen Mikroprozessoren im Allgemeinen mehrere Taktzyklen. Moderne Prozessoren sind jedoch mit einer Art Pipeline ausgestattet, die es ihnen erlaubt, mit jedem Takt eine neue Instruktion zu beginnen und eine andere zu beenden. Die im vorigen Takt übernommene Instruktion ist in ihrer Bearbeitung innerhalb der Pipeline schon bei einer nächsten Stufe angekommen und gibt somit die Hardwareressourcen am Beginn der Pipeline frei. Noch früher begonnene Instruktionen befinden sich in späteren Stadien der Pipeline. Ist die Pipeline einmal gefüllt, sieht es so aus, als ob der Prozessor je Takt eine Instruktion ausführt. Der Prozessor ist damit optimal genutzt, es ist nur eine Zeitverzögerung zwischen Bearbeitungsbeginn und Bearbeitungsende zu beachten. Man kann sich einen heutigen Prozessor also wie ein Miniaturfließband vorstellen [Hennessy und Patterson 1996], [Müller-Schloer 1991]).

Es gibt Vorkommnisse, welche die Pipeline außer Takt bringen. Ein Grund ist, dass ein Prozessor bei der Vorbereitung seiner Schritte gewisse Annahmen treffen muss, die möglicherweise nicht zutreffen. Man stelle sich eine Abfrage vor, deren Ergebnis über den weiteren Verlauf eines Programms entscheidet. Während sich die Abfrage noch in der Pipeline befindet, müssen für einen reibungslosen Ablauf schon die nächsten Instruktionen begonnen werden. Dazu nimmt der Prozessor einen bestimmten Ausgang der Abfrage an. Entspricht die Annahme des Prozessors dem tatsächlichen Ergebnis der Abfrage, füllt er die Pipeline mit den richtigen Instruktionen; der Programmablauf ist optimal. Liefert die Abfrage aber ein anderes Ergebnis als vom Prozessor angenommen, hat er die falschen Folgeinstruktionen begonnen, muss die Pipeline leeren und stattdessen die richtigen Instruktionen anfordern. Das kostet verhältnismäßig viel Zeit; der Programmablauf ist empfindlich gestört.

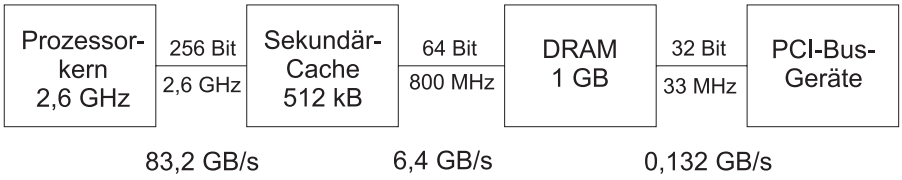


***Abbildung 2.2:*** *Datentransportwege mit theoretischen Transferraten im Testrechner mit einem Pentium 4 als Prozessor.*

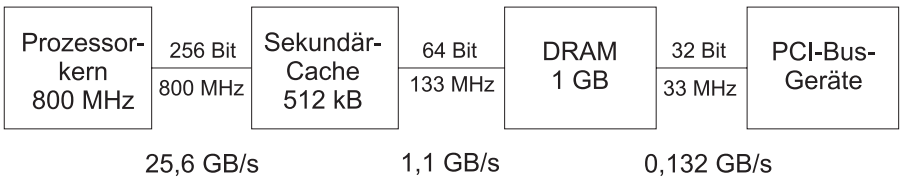


***Abbildung 2.3:*** *Datentransportwege mit theoretischen Transferraten im zweiten Testrechner mit einem Pentium III als Prozessor.*

Ein zweiter Grund für die Ausbremsung des Prozessors ist gegeben, wenn die Bandbreite für den Zugriff auf den Speicher zu gering ist. Genauso schnell, wie die Instruktionen vom Prozessor verarbeitet werden sollen, müssen auch die Instruktionen selbst und die Daten aus dem Speicher herangeschafft und die Ergebnisse wieder in den Speicher transportiert werden. Da normales, kostengünstiges RAM zu langsam ist, hat man in der Rechnerarchitektur schon lange einen Kompromiss vorgenommen. Zwischen den vergleichsweise langsamen Hauptspeicher und den schnellen Prozessor fügt man eine oder mehrere Stufen mit schnellem, teurerem Speicher ein. Solch einen Speicher nennt man *Cache*. Man macht sich hier zunutze, dass häufig mehrere Operationen nacheinander auf den gleichen Daten oder auf Daten in deren Umgebung ausgeführt werden. Lädt man aus dem Hauptspeicher einen ganzen Bereich in den Cache und benötigt der Prozessor eine Weile Daten aus genau diesem Bereich, kann

der Prozessor mit großer Geschwindigkeit arbeiten. Sobald jedoch Daten angefordert werden, die sich nicht im Cache befinden, muss der Prozessor eventuell eine Zwangspause einlegen und auf die Beschaffung der Daten aus dem Hauptspeicher warten. Abbildung 2.2 zeigt die Verhältnisse schematisch für den Testrechner und Abbildung 2.3 für den zweiten, langsameren Rechner. Die theoretische Transferrate zwischen Cache und Prozessorkern unterscheidet sich von derjenigen zwischen Hauptspeicher und Cache um eine Größenordnung. Zum Vergleich ist außerdem die wiederum um eine Größenordnung kleinere Datenrate über den PCI-Bus in den Hauptspeicher angegeben, die beispielsweise für die Datenakquisition mit externen Geräten maßgeblich ist. Die hohe Transferrate zwischen sekundärem Cache und Prozessorkern wird dadurch erleichtert, dass sich beide auf dem Prozessor-Chip befinden.

In Tabelle 2-3 ist die Verarbeitungsrate für die Addition von Bildern verschiedener Datentypen nach folgender C-Funktion, hier für den Datentyp `float` angegeben, aufgeführt:

```
void vf_add_vvv(float* w1, const float* x1,
                           const float* y1, hUINT_PTR n) {
    while (n >= 2) {
        w1[0] = x1[0] + y1[0];
        w1[1] = x1[1] + y1[1];
        w1 += 2; x1 += 2; y1 += 2; n -= 2;
    }
    if (n) w1[0] = x1[0] + y1[0];
    return;
}
```

Der Funktionsparameter `n` gibt die Anzahl der zu addierenden Datenpunkte an. In der `while`-Schleife werden jeweils zwei Datenpunkte addiert statt nur einer, weil dies insgesamt etwas schneller ist.

Bei den Typen `float` und `unsigned long` besteht jeder Bildpunkt aus einem 32-Bit-Wort, was der Registerbreite des Prozessors entspricht. Betrachtet man die Ergebnisse des Testrechners, läuft die Addition für diese Datentypen schneller als für diejenigen, bei denen ein Bildpunkt mehr oder weniger als 32 Bit belegt. Dies ist beim Zweitrechner jedoch nicht zu beobachten. Der gemessenen Verarbeitungsrate des Hauptrechners von $0{,}171\times10^9$ Pixel/s entspricht bei `float` eine Datenrate von $0{,}171\times10^9\times8$ Byte/s = 1,37 GB/s in den Prozessor, was weit unter der in Abbildung 2.2 angegebenen theoretischen Datentransferrate, wie sie sich aus Busbreite und Busfrequenz ergibt, liegt.

***Tabelle 2-3:*** *Verarbeitungsrate der Testrechner bei Addition von Bildern der Größe 1024×1024*

| Datentyp | Verarbeitungsrate [$10^9$ Pixel/s] | | Verarbeitungsrate [GB/s] | |
|---|---|---|---|---|
| | Hauptrechner | Zweitrechner | Hauptrechner | Zweitrechner |
| `double` | 0,105 | 0,0182 | 1,68 | 0,29 |
| `float` | 0,171 | 0,0225 | 1,37 | 0,18 |
| `unsigned long` | 0,175 | 0,0207 | 1,40 | 0,17 |
| `unsigned short` | 0,112 | 0,0362 | 0,45 | 0,15 |
| `unsigned char` | 0,105 | 0,0646 | 0,21 | 0,13 |

Um eine Vorstellung von tatsächlich erreichbaren Datentransferraten zu erhalten, sei das Kopieren von einem Bild in ein anderes betrachtet. Die Vektorfunktionen des Kopieroperators sind für einige Datentypen in Assembler programmiert. Somit ist die Anzahl der zum Kopieren benötigten Prozessorinstruktionen genau bekannt. Nimmt man an, dass alle Instruktionen

nacheinander und nicht teilweise parallel ausgeführt werden, kennt man auch die Anzahl der höchstens benötigten Taktzyklen. Daraus lässt sich die theoretische Kopierrate berechnen. Wird diese Rate nicht erreicht, ist dies im Wesentlichen durch die reale Bandbreite der Busse bedingt. Die Verhältnisse für die beiden Testrechner sind in Tabelle 2-4 und Tabelle 2-5 aufgeführt.

***Tabelle 2-4:*** *Kopierrate des Testrechners mit Pentium 4 bei Bildern der Größe 1024×1024. Die Kopierfunktionen für die Datentypen* `float`, `unsigned short` *und* `unsigned char` *sind assembler-optimiert. Für diese Typen ist angegeben, wie viele Bildpunkte pro Instruktion kopiert werden.*

| Datentyp | theoretische Kopierrate | | gemessene Kopierrate | |
|---|---|---|---|---|
| | [Pixel/Instrukt.] | [GB/s] | [$10^9$ Pixel/s] | [GB/s] |
| `double` | - | - | 0,139 | 1,11 |
| `float` | 8/12 = 0,67 | 6,9 | 0,288 | 1,12 |
| `unsigned long` | - | - | 0,271 | 1,08 |
| `unsigned short` | 16/12 = 1,33 | 6,9 | 0,534 | 1,07 |
| `unsigned char` | 32/12 = 6,67 | 6,9 | 1,186 | 1,19 |

***Tabelle 2-5:*** *Kopierrate des zweiten Testrechners mit Pentium III bei Bildern der Größe 1024×1024. Die Kopierfunktionen für die Datentypen* `float`, `unsigned short` *und* `unsigned char` *sind assembler-optimiert. Für diese Typen ist angegeben, wie viele Bildpunkte pro Instruktion kopiert werden.*

| Datentyp | theoretische Kopierrate | | gemessene Kopierrate | |
|---|---|---|---|---|
| | [Pixel/Instrukt.] | [GB/s] | [$10^9$ Pixel/s] | [GB/s] |
| `double` | - | - | 0,0201 | 0,161 |
| `float` | 8/12 = 0,67 | 6,9 | 0,0357 | 0,143 |
| `unsigned long` | - | - | 0,0249 | 0,100 |
| `unsigned short` | 16/12 = 1,33 | 6,9 | 0,0630 | 0,126 |
| `unsigned char` | 32/12 = 2,67 | 6,9 | 0,1324 | 0,132 |

Die Messergebnisse zeigen, dass eine Vorhersage, wie der Durchsatz in Abhängigkeit von der Objektgröße, des Datentyps und der Taktfrequenz des Prozessors sein wird, nicht oder nur schwer möglich ist. Eine nähere Untersuchung hierzu ist nicht Thema dieser Arbeit. Der nächste Abschnitt beschäftigt sich jedoch noch ein wenig mit den Eigenheiten bei Laufzeitmessungen auf einem PC.

### 2.3.2 Laufzeitmessungen

Moderne PCs sind komplexe Systeme, die darauf ausgelegt sind, das Zusammenwirken von Hard- und Software selbsttätig zu optimieren. Die Optimierungskriterien richten sich nach dem, was die Hersteller als hauptsächlichen Bedarf bei der Kundschaft ausmachen. Die breite Masse bilden die Benutzer von PCs im Büro oder zu Hause. PCs für industrielle Aufgaben oder erst recht für Bildverarbeitung spielen eher eine untergeordnete Rolle. Deshalb geschehen große Entwicklungen auch nicht für diesen Minderheitsmarkt. Die Erweiterungen zur Parallelverarbeitung mehrerer Datenpunkte mit einer einzigen Instruktion wurden nicht für die wissenschaftliche und industrielle Bildverarbeitung geschaffen, sondern für Multimedia-Anwendungen, wie es der Name MMX (multimedia extension) eines dieser Instruktionssätze schon sagt.

Eine Konsequenz ist, dass das Verhalten eines auf einem PC laufenden Programms nicht genau vorausgesagt werden kann. Für viele industrielle Anwendungen fehlt beispielsweise die Fähigkeit eines PCs, innerhalb einer garantierten kurzen Zeit auf ein äußeres Ereignis zu reagieren (Echtzeitfähigkeit). Wie später mehrfach deutlich wird, kann man nicht einmal davon ausgehen, dass der Algorithmus mit der geringsten Anzahl von Operationen auch tatsächlich der schnellste ist. Außerdem kann man feststellen, dass ein Algorithmus nicht immer die gleiche Zeit benötigt. Das sollen die Messungen der Laufzeiten von drei benutzerdefinierten heurisko-Operatoren auf dem Testrechner demonstrieren.

```
operator op1();
    t = GetTimeDiff.second(); // interne Referenzzeit setzen
    repeat(50);
        z = SqrDiff(x,y);     // Quadrieren der Differenz
    endrepeat;
    t = GetTimeDiff.second(); t; // Zeitdifferenz messen
endoperator;

operator op2();
    repeat(10);
        t = GetTimeDiff.second(); // interne Referenzzeit setzen
        z = SqrDiff(x,y);         // Quadrieren der Differenz
        t = GetTimeDiff.second(); t; // Zeitdifferenz messen
    endrepeat;
endoperator;

operator op3();
    z = SqrDiff(x,y);
    repeat(10);
        t = GetTimeDiff.second();//interne Referenzzeit setzen
        z = SqrDiff(x,y);        // Quadrieren der Differenz
        t = GetTimeDiff.second(); t; // Zeitdifferenz messen
    endrepeat;
endoperator;
```

In allen drei Operatoren gehören je zwei Aufrufe von `GetTimeDiff()` zusammen. Der erste setzt die interne Referenzzeit auf die aktuelle Systemzeit; der zurückgelieferte Wert ist hier uninteressant. Der zweite Aufruf ermittelt dann die Zeitdifferenz zum ersten Aufruf. Diese Differenz beinhaltet allerdings auch den Zeitaufwand des Aufrufs von `GetTimeDiff()`, der jedoch vernachlässigbar ist. Die Zeit wird intern in Anzahl Prozessorzyklen gemessen und dann mit der Taktfrequenz des Prozessors in Sekunden umgerechnet und mit der Anweisung `t;` auf dem Bildschirm ausgegeben. Der erste Operator misst die Zeit für eine Schleife, die 50-mal den eingebauten Operator `SqrDiff()` aufruft. Für die Messungen wurde der Operator 10-mal ausgeführt. Der zweite Operator misst 10-mal die Ausführungszeit eines einzigen Aufrufs von `SqrDiff()`. Der dritte Operator unterscheidet sich vom zweiten nur dadurch, dass vor der Schleife die Operation `SqrDiff()` einmal ohne Zeitmessung ausgeführt wird. Damit unterscheiden sich die Vorgeschichten der beiden Schleifen.

Tabelle 2-6 enthält die Messwerte selbst, dann den Mittelwert, die Standardabweichung und das Verhältnis des Maximalwertes zum Minimalwert für jede Messwertspalte. Die Maximal- und Minimalwerte sind fett gedruckt. Die drei linken Spalten wurden bei ausgeschaltetem, die drei rechten bei eingeschaltetem Hyper-Threading gemessen. Wie man an den miteinander korrespondierenden Mittelwerten erkennt, sind die Zeiten mit Hyper-Threading im Durchschnitt zwischen 6 und 9 % größer als ohne. Auch die Standardabweichung und das Verhältnis der Extremwerte sind mit Hyper-Threading tendenziell höher. Diese Tendenz wird allerdings durch die dritte Spalte unterbrochen. Sie enthält nämlich einen auffälligen

Ausreißer nach oben. Lässt man aber den Ausreißer weg, bestätigen auch die jeweils dritten Spalten die Tendenz.

***Tabelle 2-6:*** *Variationen der Ausführungszeiten für heurisko-Operatoren auf dem Testrechner unter dem Betriebssystem Windows XP*

| | Ausführungszeiten in s für Objekte `float obj[16384][512]` | | | | | |
|---|---|---|---|---|---|---|
| | ohne Hyper-Threading | | | mit Hyper-Threading | | |
| Operator | `op1()` | `op2()` | `op3()` | `op1()` | `op2()` | `op3()` |
| Messung 1 | 5.0137 | **0.1181** | 0.1007 | 5.4414 | 0.1025 | 0.1185 |
| Messung 2 | 4.9991 | **0.0994** | 0.1003 | **5.5442** | **0.1261** | 0.1186 |
| Messung 3 | 5.0167 | 0.1000 | **0.0994** | 5.4614 | 0.1192 | 0.1175 |
| Messung 4 | 5.0018 | 0.1001 | 0.0994 | 5.4183 | 0.1065 | 0.1023 |
| Messung 5 | 5.0055 | 0.0997 | 0.0997 | 5.5132 | 0.1067 | 0.1115 |
| Messung 6 | **5.0233** | 0.0994 | 0.1003 | 5.4470 | 0.0998 | **0.1266** |
| Messung 7 | 5.0132 | 0.1002 | 0.0997 | 5.5341 | **0.0994** | 0.1196 |
| Messung 8 | 5.0089 | 0.1007 | 0.1197 | 5.4350 | 0.1054 | 0.1131 |
| Messung 9 | 5.0161 | 0.0995 | **0.1433** | 5.4851 | 0.1036 | **0.1018** |
| Messung 10 | **4.9923** | 0.0994 | 0.1001 | **5.3801** | 0.1088 | 0.1056 |
| Mittelwert | 5.0091 | 0.1017 | 0.1063 | 5.4660 | 0.1078 | 0.1135 |
| Stdabw. | 0.0094 | 0.0058 | 0.0144 | 0.0527 | 0.0085 | 0.0082 |
| Max/Min | 1.0062 | 1.1881 | 1.4417 | 1.0305 | 1.2686 | 1.2436 |

Man kann für eine Software ohne parallele Threads wie heurisko bei Betrieb auf dem Testrechner zusammenfassend feststellen:

- Mit Hyper-Threading verlängert sich die Ausführungszeit.
- Ebenso nehmen die Schwankungen zwischen einzelnen Testläufen zu.
- Misst man vergleichsweise kurze Zeiten, können extreme Ausreißer auftreten.
- Die Vorgeschichte wirkt sich auf die gemessenen Zeiten aus.
- Das Messen zur Optimierung von Ausführungszeiten muss sorgfältig durchdacht sein.

Da die Operatoren in heurisko nur in einem Thread laufen, wurde Hyper-Threading auf dem Testrechner im Normalfall abgeschaltet.

# 3 Lösungskonzepte aus der Literatur

In der Einleitung wurde die Komplexität und Vielfalt der digitalen Bildverarbeitung diskutiert. In diesem Kapitel wird nun untersucht, mit welchen Konzepten ein System zur Lösung der vielfältigen Aufgabenstellungen entwickelt werden kann. Diese Analyse wird die Basis für die Struktur des in der vorliegenden Arbeit vorgestellten Bildverarbeitungssystems heurisko sein.

Der klassische Lösungsansatz, Programmbibliotheken in Programmiersprachen der dritten Generation (third generation languages, 3GLs [FOLDOC]) wie Pascal, C oder C++, werden in Abschnitt 3.1 diskutiert. Höhere Programmiersprachen (fourth generation languages, 4GLs wie MATLAB oder heurisko) erlauben problemspezifischere Ansätze und beschleunigen zudem die Entwicklung durch die Vermeidung der Kompilier-Ausführungszyklen (Abschnitt 3.2). Graphische Programmierschnittstellen versprechen dagegen die vollständige Vermeidung textbasierter Programmierung (Abschnitt 3.2.2). Als Optimum erscheinen schließlich Systeme, die automatisch für eine Klasse von Problemen eine optimale Lösung finden und damit dem Benutzer die Programmierung insgesamt abnehmen (Abschnitt 3.3).

## 3.1 Programmbibliotheken

In der numerischen Mathematik haben einige wenige Programmbibliotheken wie *LINPACK* und *LAPACK* große Verbreitung gefunden ([LINPACK], [LAPACK]). Außerdem gibt es Bücher wie *Numerical Recipes* [Press et. al. 1992], in denen numerische Algorithmen ausführlich beschrieben werden. Numerische Algorithmen wie z. B. zur Matrixinversion oder zur Singularwertzerlegung sind komplex und bedürfen einer sorgfältigen Implementierung, um korrekte Ergebnisse zu liefern und numerisch stabil zu sein. Deswegen ist es nicht verwunderlich, dass sich die überwältigende Mehrheit der Benutzer auf wenige bewährte und getestete Bibliotheken verlässt und diese sich dadurch in einem selbst stabilisierenden Prozess durchsetzen.

In der Bildverarbeitung ist eine ähnliche Entwicklung nicht zu beobachten. Es gibt eine große Fülle von Programmbibliotheken, von denen sich aber keine wirklich durchgesetzt hat. In der Literatur findet man dementsprechend eine Menge von Büchern, die Programmbibliotheken beschreiben, darunter [Voss und Süße 1991], [Bässmann und Besslich 1991], [Klette und Zamperoni 1992], [Pitas 1993], [Sid-Ahmed 1995], [Parker 1997], [Umbaugh 1998], [Ritter und Wilson 2001] und [Nikolaidis und Pitas 2001].

Von allen existierenden Bibliotheken werden ein paar typische Vertreter herausgegriffen, um deren Nutzen und Anwendungsmöglichkeiten für die Bildverarbeitung zu diskutieren. Es handelt sich um

1. *OpenCV*, eine von Intel entwickelte C-Bibliothek für Bildverarbeitung und Computer Vision,

2. *VXL* (die „Vision *something* Libraries“), eine template-basierte C++-Bibliothek,

3. *VIGRA*, das intensiv auf generischem Programmieren beruht, und

4. *HALCON*, eine auf mehreren Plattformen verbreitete, nicht im Quellcode verfügbare kommerzielle Bildverarbeitungsbibliothek der deutschen Firma MVTec.

Nach der oben gegebenen Definition bedeutet die Lösung einer Aufgabe mit Bildverarbeitung die Erstellung von Software. Das entscheidende Charakteristikum von Software ist - der Name sagt es ja schon - ihre Flexibilität. Da die Entwicklung von Software zeitraubend und damit teuer ist, ist man sehr daran interessiert, Software wieder verwenden, also flexibel einsetzen zu können. Dies gelingt umso besser, je flexibler Software ist. Flexible Software erleichtert aber nicht nur die Wiederverwendung, sondern auch die Wartung eines vorhandenen Software-Systems. Es kommt sicher nur selten vor, dass Software nicht im Laufe ihres Lebens an neue Anforderungen angepasst und um erwünschte Funktionalität erweitert werden muss. Man denke beispielsweise an den Übergang von 16- auf 32- oder von 32- auf 64-Bit-Prozessoren.

Leider ist Software jedoch nicht von sich aus schon so flexibel, wie man sich das erhofft; zwischen Anspruch und Wirklichkeit klafft eine Lücke. Deswegen soll in diesem Abschnitt an den herausgegriffenen prominenten Beispielen die Frage untersucht werden, welche Flexibilität Programmbibliotheken zur Lösung von Bildverarbeitungsproblemen bringen, welches Spektrum an Problemen sich damit lösen lässt und wie einfach dies ist.

Das Teilgebiet Software Engineering der Informatik beschäftigt sich schon lange damit, wie man gute Software erstellt und was gute Software überhaupt ist. Nach weit verbreiteter Auffassung lässt sich mit objektorientierten Methoden am leichtesten gute Software entwickeln. Aus diesem Grund sind neuere Bibliotheken häufig in C++ implementiert.

### 3.1.1 OpenCV

OpenCV ist eine der bekanntesten Open-Source-Programmbibliotheken [OpenCV]. Sie wurde primär von Intel entwickelt, ist frei verwendbar und steht als C-Bibliothek im Quellcode zur Verfügung. Die Bibliothek ist kompatibel mit den kommerziellen und nur binär erhältlichen Intel Performance Primitives (IPP) mit hoch optimiertem Code für die Prozessoren von Intel. OpenCV ist für sich verwendbar, nutzt aber die IPP-Dateien, wenn es diese vorfindet. Hinter Intels Engagement für ein Open-Source-Projekt darf man sicherlich ein kommerzielles Interesse vermuten, da anspruchsvolle Bildverarbeitungsanwendungen auch leistungsstarke Prozessoren benötigen.

OpenCV bietet zwei Besonderheiten:

1. Für Bilddaten existiert die einheitliche Datenstruktur `IplImage`, die allerdings nur 2-D-Daten kennt. Die unterstützten Basisdatentypen sind binäre Zahlen, vorzeichenlose und vorzeichenbehaftete ganze Zahlen mit 8 Bit, vorzeichenbehaftete ganze Zahlen mit 16 und 32 Bit und einfachgenaue Gleitkommazahlen. Zusätzlich können für Farbbilder und andere Mehrkanalbilder mehrere Werte an einem Bildpunkt entweder als einzelne Kanäle abgespeichert oder zu einem vektorwertigen Bildpunkt (Pixel) zusammengefasst werden. Leider werden für Faltungskerne, Regions-of-Interest und anderes eigene Datenstrukturen benutzt, die darüber hinaus auch noch datentypabhängig sind.

2. Die Bibliothek hat eine Fülle von Routinen für die Low-Level-Bildverarbeitung mit einfacher Bildarithmetik, logischen Operatoren, Filteroperationen, linearen Transformationen, morphologischen Operationen, Farbraumkonversionen, einfachen geometrischen und einigen statistischen Operationen. Zu diesen Operationen gibt es in der IPP-Bibliothek für diverse Prozessoren viele unter Einsatz von SIMD-Instruktionssätzen wie MMX in mühevoller Handarbeit optimierte Routinen. Beim Aufruf von OpenCV

prüft dieses, welcher Prozessor vorhanden ist und lädt dann, falls vorhanden, die für diesen Prozessor optimierte IPP-Bibliothek.

3. Die OpenCV-Bibliothek fügt den Basisstrukturen wie `IplImage` eine Fülle neuer Datenstrukturen hinzu. Dies liegt daran, dass OpenCV außer den Low-Level-Funktionen eine ganze Reihe von High-Level-Funktionen enthält, angefangen von Bildpyramiden und Kamerakalibrierung bis hin zur Bildfolgenanalyse mit Methoden des optischen Flusses und der Gestenerkennung, um nur einige zu nennen. Dadurch ist die Bibliothek sehr heterogen und es ist aufwändig, sie zu benutzen oder für sie eine Schnittstelle zu einem anderen Programm zu schreiben. Weiterhin fällt auf, dass die Funktionen nur für einen kleinen Teil der möglichen Datentypen implementiert sind.

Schon aus diesen wenigen Bemerkungen ist zu schließen, dass die OpenCV-Bibliothek keinen klaren Aufbau hat, sondern eher ein zufälliges Gebilde ist, das im Laufe der Zeit weniger gewachsen als gewuchert ist. Einige der implementierten Routinen sind zwar hervorragend und werden wie z. B. die Kamerakalibrierroutinen oft genutzt, aber die Bibliothek selbst hat aus den oben beschriebenen Gründen vergleichsweise nur wenige Anwender. Auch im Hause Intel selbst hat sie laut Aussage eines Mitarbeiters der Computational Nano-Vision Research Group keine weite Verbreitung gefunden (persönliche Mitteilung von Dr. Horst Haußecker an Prof. Bernd Jähne).

Es ist sehr bedauerlich, dass die Bibliothek durch ihre ersichtlichen Schwächen nicht mehr Anwendung gefunden hat, zumal in die Entwicklung der hoch optimierten Routinen der IPP-Bibliothek sehr viele Mannjahre Arbeit geflossen ist. Allerdings ist diese hardwarenahe Optimierung auch gleichzeitig eine Schwäche, denn es bedarf eines großen und ständigen Aufwands, die Routinen an die aktuellen Prozessoren anzupassen, und sie nutzt nichts auf anderen Architekturen.

### 3.1.2 VXL

VXL (Vision *something* Libraries, [VXL]) ist eine Sammlung von C++-Bibliotheken, die für die universitäre Forschung im Bereich der Bildverarbeitung entwickelte wurde. Die Bibliothekssammlung ist noch relativ jung (die Definition des Kerns fand im Februar 2000 statt) und entstand aus den zwei Bibliotheken *TargetJr* [TargetJr] und *Image Understanding Environment* [IUE] mit der Absicht, diese kompakter, schneller und konsistenter zu machen. Diese interessante Feststellung ist ein Hinweis darauf, dass viele Bibliotheken für die digitale Bildverarbeitung letztlich ihr Ziel nicht erreicht haben, ein akzeptiertes, effizient nutzbares Werkzeug zu sein. VXL ist in ANSI/ISO C++ geschrieben, um das System möglichst portabel zu halten.

Wie schon erwähnt, ist VXL modular aufgebaut. Neben den Kernbibliotheken, die keinerlei Abhängigkeiten voneinander haben, gibt es eine ganze Reihe von speziellen Bibliotheken mit möglichst wenigen Abhängigkeiten von anderen Bibliotheken. Dies hat den Vorteil, dass Anwendungen nicht mit allen, sondern nur mit den tatsächlich benötigten Bibliotheken gebunden werden müssen. Die Kernbibliotheken fasst die folgende Aufstellung zusammen.

- *vnl* (n = numerics) enthält numerische Container und Algorithmen für z. B. Vektoren, Matrizen und Matrizenzerlegungen.

- *vil* (i = imaging) ermöglicht das Lesen und Speichern von Bildern, auch von sehr großen, in vielen gängigen Dateiformaten.

- *vgl* (g = geometry) besitzt Funktionalität für ein- bis dreidimensionale elementare geometrische Objekte wie Punkte und Kurven.

- *vsl* (s = streaming I/O) bietet plattformunabhängige binäre Ein- und Ausgabefunktionalität.

- *vbl* (basic templates) enthält nützliche Template-Klassen für beispielsweise zwei- und dreidimensionale und dünn besetzte Datenfelder.

- *vul* (u = utilities) hält vielfach nützlichen, nicht-numerischen Code bereit für Dinge wie den Umgang mit Dateinamen und URL-Adressen, das Parsen von Kommandozeileneingaben, Zeichenkettenoperationen und Zeitfunktionen.

Weitere Bibliotheken beinhalten zusätzliche numerische Algorithmen, Bildverarbeitung, Koordinatensysteme, Kamerageometrie, Stereobildverarbeitung, Bestimmung räumlicher Strukturen aus Bewegung (structure from motion), statistische Methoden, Klassifizierung, robuste Schätzung, Objektverfolgung, 3-D-Bildverarbeitung, auf OpenGL basierende Elemente für graphische Benutzeroberflächen und vieles andere mehr. Die Beschreibungen finden sich im so genannten VXL-Buch [VXL Book].

Aus dieser Zusammenstellung geht schon hervor, dass diese Bibliothek sehr forschungsorientiert aufgebaut ist. Es fehlen Komponenten zur Einbindung von Bilderfassungshardware und Bildverarbeitungsfunktionalität, wie sie z. B. in Inspektionssystemen in der Industrie benötigt wird. Im Gegensatz zur OpenCV-Bibliothek gibt es allerdings eine einheitlichere Datenstruktur.

### 3.1.3 VIGRA

Die bisherigen Betrachtungen in diesem Abschnitt haben deutlich gezeigt, dass objektorientierte Programmierung allein nicht zu wirklich nützlichen Programmbibliotheken für die Bildverarbeitung führt. Man kann nun immer behaupten, dies liege an der mangelhaften Umsetzung. Besser ist es jedoch kritisch zu fragen, ob es nicht tiefer liegende Gründe gibt. Der Autor von VIGRA, Ullrich Köthe, hat sich speziell mit dem Problem fehlender Flexibilität und damit Wiederverwendbarkeit der bestehenden Software im Bereich der Bildverarbeitung beschäftigt. Für ihn sind es hauptsächlich zwei Gründe, die durch objektorientierte Methoden alleine nicht beseitigt werden können:

- Leistungsfähigkeit kontra Flexibilität: In der Bildverarbeitung hat man es mit großen Datenmengen zu tun, die in einer bestimmten Zeit verarbeitet werden müssen. Das führt allzu oft dazu, dass die Flexibilität der Ausführungsgeschwindigkeit geopfert wird.

- Kopplung zwischen Software-Modulen: Module von Softwarepaketen sind miteinander gekoppelt, so dass Teile nicht losgelöst vom Ganzen in anderen Software-Systemen verwendet werden können.

Was demnach gewünscht wird, ist zum ersten eine Technik zur Erstellung flexibler Software ohne nachteiligen Einfluss auf die Geschwindigkeit und sind zum zweiten unabhängige, einzeln wieder verwendbare Software-Bausteine. Köthe setzt genau an diesen beiden Punkten an und propagiert *generisches Programmieren* als Schlüssel zur Lösung der Probleme. Das erklärt auch den Namen der Bibliothek, denn VIGRA ist eine Abkürzung für *Vision with Generic Algorithms*. Das Prinzip der generischen Programmierung wurde von Musser und Stepanov [Musser und Stepanov 1989, 1994] unabhängig von einer Programmiersprache eingeführt und später als Grundlage der Standard *Template Library* (STL) in C++ übernommen. C++ ist auch die Sprache, in der Köthe mit dem Template-Mechanismus die ausgeprägteste Unterstützung der generischen Programmierung fand. Die folgende Erläuterung der

Grundideen in VIGRA orientiert sich in sehr gekürzter Form an der Darstellung in [Köthe 1999].

Als erster Versuch ohne generische Programmierung zur Überwindung der Abhängigkeit der Algorithmen vom Datentyp sei die folgende abstrakte Klasse eines Bildes betrachtet, von der konkrete Bilder abgeleitet werden können:

```
class AbstractImage
{
    public:
    virtual unsigned char getPixel(int x, int y) const = 0;
    virtual void setPixel(unsigned char val, int x, int y) = 0;
    virtual int width() const = 0;
    virtual int height() const = 0;
    // …
}
```

Das Konzept mit abstrakten Klassen und virtuellen Funktionen hat zwei entscheidende Nachteile. Virtuelle Funktionsaufrufe können von Compilern nicht durch ihre konkrete Implementierung ersetzt werden (*inlining*) und führen laut Messungen von Köthe zu einer zeitlichen Mehrbelastung von bis zu 500 %. Der zweite, schwerwiegendere Nachteil besteht im festgelegten Typ des Rückgabewertes. Für Bilder anderer Datentypen müssten weitere abstrakte Klassen definiert und eigene Versionen von Algorithmen geschrieben werden. Abhilfe schaffen Templates:

```
template <typename PIXELTYPE>
class GenericImage
{
    public:
    PIXELTYPE getPixel(int x, int y) const {
        return lines_[y][x];
    }

    void setPixel(PIXELTYPE val, int x, int y) {
        lines_[y][x] = val;
    }
    int width() const { return width_; }
    int height() const { return height_; }
    // …
    private:
    PIXELTYPE * data_;
    PIXELTYPE ** lines_;
    int width_, height_;
}
```

Damit könnte ein generischer Kopieralgorithmus wie folgt aussehen:

```
template <typename SRCIMAGE, typename DESTIMAGE>
void copyImage(SRCIMAGE const & src, DESTIMAGE & dest)
{
    for(int y=0, y<src.height(); ++y)
        for(int x=0, x<src.width(); ++x)
            dest.setPixel(src.getPixel(x, y), x, y);
}
```

Der Template-Mechanismus entfernt also in den Algorithmen die Abhängigkeit vom Datentyp und erlaubt dem Compiler wegen des so genannten *compile-time polymorphism*, den Code

zu expandieren (inline) und damit Programme zu erzeugen, die genauso schnell laufen wie traditionelle zeigerbasierte Implementierungen. Die geringen gemessenen Laufzeitunterschiede führt Köthe darauf zurück, dass die Optimierungsstufe der meisten bestehenden Compiler noch nicht genügend gut mit generischem Code umgehen kann. Es gebe aber keinen zwingenden Grund, dass generischer Code langsamer sein müsse.

Damit ist die Grundidee erläutert. Für die praktische Arbeit werden aber neben den generischen Datenstrukturen und Algorithmen noch weitere Elemente benötigt. *Iteratoren* erlauben das Navigieren durch Objekte und entkoppeln Datenstrukturen und Algorithmen voneinander. *Funktoren* kapseln einen Teil eines Algorithmus und wirken wie Funktionszeiger, können aber einen internen Status speichern und vom Compiler durch den konkreten Code ersetzt werden. Darüber hinaus werden spezielle Datenzugriffsobjekte benötigt und so genannte *Trait*-Objekte zur Speicherung maschinenlesbarer Zusatzinformation wie beispielsweise über automatische Typkonversionen.

VIGRA ist im Quellcode frei zugänglich und steht unter *The VIGRA Artistic License*, die mehr Freiheiten als die weithin bekannte GPL (GNU Public License) gewährt. In seiner Dissertation diskutiert [Köthe 2000] ausführlich, welche Probleme bei der Software-Entwicklung auftreten können und wie sie speziell für die Bildverarbeitung in VIGRA gelöst sind.

### 3.1.4 HALCON

HALCON ist eine für Windows, Linux und einige UNIX-Systeme erhältliche Programmbibliothek mit den Sprachschnittstellen HALCON/C, HALCON/C++ und HALCON/COM. Entsprechend lässt es sich mit den Programmiersprachen C, C++ und in Microsoft Windows mit Visual Basic, C# und Delphi verwenden. Als kommerzielles Produkt wird HALCON allerdings nicht mit Quellcode geliefert. Aus dem gleichen Grund existiert auch über die Bedienungsanleitungen hinaus keine Literatur zur Bibliothek. Dass HALCON hier trotzdem erwähnt wird, hat neben seinem Bekanntheitsgrad – zumindest in Deutschland - und seiner umfangreichen Funktionalität einen anderen Grund. Mit *HDevelop* verfügt HALCON nämlich über eine interaktive Programmierumgebung, so dass HALCON sowohl als 3GL- als auch als 4GL-Sprache angesehen werden kann [HALCON].

HDevelop ermöglicht auf der einen Seite interaktives Arbeiten und Ausprobieren von Algorithmen und auf der anderen Seite das Erstellen von Programmen. Die Programme werden in einer proprietären Sprache gespeichert und können jederzeit so ausgeführt werden. Die Programme können nicht direkt editiert, sondern müssen Anweisung um Anweisung über Menüs und Dialoge interaktiv erstellt werden. Auf diese Weise erreicht HDevelop einen hohen Grad an impliziter Hilfestellung bei der Programmentwicklung und damit eine geringe Fehlerquote. Hat man die Entwicklung einer Anwendung abgeschlossen, kann man interaktiv erstellte Programme in Code der Sprachen C, C++ oder Visual Basic exportieren.

Ansonsten behauptet HALCON von sich, zurzeit die umfassendste auf dem Weltmarkt erhältliche Bibliothek für Bildverarbeitung zu sein. Wenn man sich den angebotenen Funktionsumfang ansieht, scheint das nicht nur aus der Luft gegriffen zu sein. Die über 1100 Operatoren betreffen alle Ebenen der Bildverarbeitung von der Vorverarbeitung bis zur Analyse. Der mutmaßliche wirtschaftliche Erfolg von HALCON zeigt, dass offensichtlich die Kombination aus einer umfangreichen, klar strukturierten und sich auch an den Bedürfnissen der Industrie orientierenden Programmbibliothek und aus der Möglichkeit, interaktiv damit arbeiten zu können, ein guter Weg zur Lösung von Bildverarbeitungsproblemen ist.

### 3.1.5 Fazit

Zusammenfassend kann festgestellt werden, dass Programmbibliotheken zur erfolgreichen Lösung der breiten Fülle von Bildverarbeitungsaufgaben nicht ausreichen. Dies liegt aber nur zum Teil an mangelhaft aufgebauten Bibliotheken. Dahinter stecken tiefer gehende Probleme. In der digitalen Bildverarbeitung ist man mit folgenden Tatsachen konfrontiert:

1. Es gibt eine Fülle an unterschiedlichen Objekten, die bearbeitet werden müssen. Man hat es nicht nur mit Bilddaten zu tun, sondern mit vielen anderen Datenstrukturen, darunter Histogrammvektoren, Faltungskernen, Punktlisten, Objektparameterlisten, Konturbeschreibungen und Graphen. Das ist eine große Herausforderung an den Entwurf einer Programmbibliothek. Darin besteht auch der große Unterschied zu numerischen Programmbibliotheken, die mit einer weit geringeren Zahl von Objekttypen auskommen können. Wie an den Beispielen diskutiert wurde, werden Programmbibliotheken dadurch schnell unübersichtlich und schwer benutzbar.

2. Daneben gibt es auch eine Fülle an Datentypen. In einer numerischen Programmbibliothek hat man es dagegen mit nur wenigen Datentypen zu tun, in der Regel mit einfach- oder doppeltgenauen Gleitkommazahlen, vielfach sogar nur mit einem der beiden Datentypen. Lediglich bei Bibliotheken, die komplexe Größen verarbeiten, benötigt man noch die entsprechenden komplexen Gleitkommazahlen. In der Bildverarbeitung kommen noch Datentypen für ganze Zahlen mit verschiedener Wortlänge hinzu. In der Regel sind das alle Datentypen, die eine Programmiersprache anzubieten hat, inklusive binärer Zahlen. In der Forschung wird dieses Problem oft umgangen, indem alle Bilddaten in einfachgenaue Gleitkommazahlen (Typ `float` in C) umgewandelt und alle Berechnungen mit diesen durchgeführt werden. Dies ist aber keine Lösung, wenn wie z. B. bei Bildfolgen große Datenmengen vorliegen. Die Umwandlung von 8-Bit-Bildern in solche vom Typ `float` vervierfacht den Speicherbedarf.

3. Programmbibliotheken alleine sind offensichtlich nicht genügend flexibel. Der Aufwand, um damit eine Lösung für ein komplexes Bildverarbeitungsproblem zu entwickeln, ist durch den Kompilier-Ausführungszyklus immer noch zu hoch.

## 3.2 Höhere Programmiersprachen und graphische Programmentwicklung

Skriptsprachen bieten den Vorteil der schnellen und flexiblen Änderung. Das bereits bei den Bibliotheken beschriebene Paket HALCON besitzt eine 4GL-Schale, so dass es auch wie eine höhere Programmiersprache benutzt werden kann. Dieser Ansatz soll anhand der Software MATLAB näher diskutiert werden (Abschnitt 3.2.1). Weiterhin soll in Abschnitt 3.2.2 betrachtet werden, welche Vor- und Nachteile eine graphische Programmentwicklung mit sich bringt.

### 3.2.1 MATLAB

MATLAB ist eine 4GL-Sprache der Firma The MathWorks aus den USA, die in Forschung und Technik häufig zur Verkürzung der Entwicklungszeit statt einer 3GL-Sprache eingesetzt wird [MATLAB]. Die Geschwindigkeit der erstellten Programme spielt eher eine untergeordnete Rolle. MATLAB ist weit verbreitet – von über 500.000 Anwendern weltweit ist die Rede - und kann sich deshalb den Aufwand leisten, mit vielfältigen Schnittstellen zur Datenerfassung und –ausgabe ausgestattet zu sein. Umgekehrt trägt die Verbreitung dazu bei, dass

andere Software-Hersteller Schnittstellen zu MATLAB in ihre Produkte integrieren. Die Hauptmerkmale sind:

- interaktiver und automatischer Betrieb
- mehr als 600 mathematische, statistische und technische Funktionen
- reichhaltige Visualisierungsmöglichkeiten
- Werkzeuge zur Erstellung graphischer Benutzeroberflächen
- Datenimport und -export
- große Palette von Erweiterungspaketen unter anderem für Simulation, Bilderfassung und Bildverarbeitung
- Integration in andere Umgebungen (C, C++, Fortran, Java, COM-Komponenten und Excel) und Einbettung von Programmen in C, C++, FORTRAN oder Java
- Übersetzung von MATLAB-Anwendungen in C oder C++

Viele der mathematischen Funktionen von MATLAB arbeiten direkt mit Vektoren und Matrizen. Aufgaben, die in C Dutzende von Zeilen lang und durch Schleifen sehr rechenintensiv wären, können daher in MATLAB mit nur einem einzigen Funktionsaufruf erledigt werden. Der kürzere MATLAB-Code ist außerdem einfacher zu pflegen. In MATLAB sind mit LAPACK, BLAS und FFTW angesehene Bibliotheken anderer Hersteller integriert.

MATLAB ist zwar keine eigens für Bildverarbeitung entwickelte Software, bietet aber nach der obigen Aufzählung auch Spezialpakete für diesen Bereich. MATLAB wird von vielen Bildverarbeitern zur schnellen Entwicklung von Algorithmen benutzt. Für die Routineauswertungen und Anwendungen in der Industrie ist MATLAB dagegen weniger geeignet. Dies liegt im Wesentlichen daran, dass fast alle Berechnungen mit doppeltgenauen Gleitkommazahlen durchgeführt werden und daher zum einen viel Speicherplatz benötigen und zum anderen deutlich langsamer rechnen.

Die Tatsache, dass Befehle wie in MATLAB interpretiert werden müssen, bedeutet keinen nennenswerten Geschwindigkeitsverlust, solange ein zu interpretierender Befehl viele Datenpunkte auf einmal verarbeitet. Das ist der Fall, wenn eine Operation ein ganzes Bild mit größenordnungsmäßig einer Million Datenpunkten verarbeitet. Selbst bei zeilen- oder spaltenweiser Vorgehensweise werden pro Befehl noch hunderte von Datenpunkte mit einem einzigen Operationsaufruf verarbeitet, so dass sich der zeitliche Zusatzaufwand durch die Interpretation nur unwesentlich bemerkbar macht. Bei bildpunktweisem Aufruf erhält man aber gravierende Laufzeiterhöhungen. Man denke an ein Bild, dessen Bildpunkte einzeln in zwei verschachtelten Schleifen über Bildzeilen und Bildpunkte innerhalb der Zeilen verarbeitet werden. Für die Bildgröße 640×480 erhält man so 307.200 Anweisungen, die interpretiert werden müssen, bevor sie zur Ausführung kommen. Die Zeit zur Interpretation übersteigt hier bei weitem die eigentliche Datenverarbeitungszeit.

Ein Ausweg ist, solche ungeschickten Schleifen schlicht zu vermeiden. Das ist allerdings dann nicht möglich, wenn eine komplexere Funktion in der Bibliothek nicht existiert, aber aus mehrere anderen zusammengesetzt werden kann. MATLAB führt in solchen Fällen mit dem *JIT-Accelerator* eine automatische Optimierung durch [MathWorks 2002]. Dabei kommen zwei Methoden zum Einsatz, *Just-In-Time Code Generation* und *Run-Time Type Analysis*. Nach der ersten Methode wird zur Laufzeit für MATLAB-Anweisungen maschinennäherer

und damit schnellerer Code erzeugt. Nach der zweiten Methode prüft MATLAB die beteiligten Objekte und speichert speziellen Code, der bei wiederholter Verwendung der Objekte schneller ausgeführt werden kann, solange sich Typen und Formen der Objekte nicht ändern. In Kombination ergeben diese beiden Methoden für Programmschleifen im Extremfall Beschleunigungsfaktoren von bis zu einigen Tausend. Dies ist MATLABs erster Schritt zur Überbrückung der Leistungslücke zwischen 3-GL- und 4-GL-Systemen.

### 3.2.2 Graphische Programmiersysteme

Mit dem Apple Macintosh-Rechner kamen graphische Benutzerschnittstellen auf. So ist es auch nicht verwunderlich, dass die erste Software zur graphischen Programmierung in der Messdatenverarbeitung für den Mac angeboten wurde. Die Software *LabView* der Firma National Instruments wies eine einfache datenflussorientierte graphische Benutzerschnittstelle auf [LabView]. Es ist offensichtlich, dass eine solche Schnittstelle besonders in der Messdatenverarbeitung sinnvoll ist, da sich durch die Integration der Datenaufnahme eine datenflussorientierte Betrachtungsweise anbietet. Benutzt man allerdings eine graphische Benutzerschnittstelle für komplexe Auswertungsalgorithmen und Steuerabläufe, so wird sie schnell weniger übersichtlich als eine klassische prozedurale Vorgehensweise. Das zeigt, dass eine graphische Benutzerschnittstelle das Erlernen einer Software sicher erheblich erleichtert. Es ist aber eine Illusion zu glauben, ein komplexes Programm damit wesentlich leichter und ohne Mühe erstellen zu können. Bei der Wahl der Benutzerschnittstelle kann man vielfach heftige Diskussionen beobachten, die manches Mal fast “religiöse“ Züge tragen. Tatsache ist aber, dass vieles reine Gewohnheitssache ist und dass keine Benutzerschnittstelle die inhärente Komplexität eines Problems aus der Welt schafft.

Für den Bereich der Bildverarbeitung bieten beispielsweise die Entwicklungspakete *Khoros* der Firma Khoral mit *Cantata* und *Common Vision Blox* der Firma Stemmer Imaging mit *iTUITION* eine graphische Programmierung ([Khoros] und [CVB]).

## 3.3 Automatisch lernende Programmierung

Die optimale Lösung einer Aufgabenstellung ist wohl die, dem System nur die Aufgabenstellung zu geben und es selbst eine Lösung, womöglich sogar die optimale Lösung, finden zu lassen. Es ist offensichtlich, dass dieser Ansatz noch illusorisch ist für eine beliebige allgemeine Bildverarbeitungsaufgabe. Denkbar ist sie allerdings für begrenzte Aufgabenstellungen.

### 3.3.1 Selbstlernende schnelle Fouriertransformation

Ein herausragendes Beispiel dieser Art ist das Paket FFTW (*fastest Fourier transform in the west*) von Frigo und Johnson ([Frigo und Johnson 1998] und [FFTW]), das die Laufzeit der Fouriertransformation optimiert. Vorgegeben ist also die Aufgabe, von einem bestimmten Datenfeld bekannter Größe auf einem bestimmten Rechner möglichst schnell die Fouriertransformierte zu berechnen. Dies bedarf eines Lernschritts und eines Formats zur Speicherung des gelernten Wissens. FFTW scheint äußerst beliebt zu ein, denn es ist in viele Softwarepakete integriert, so auch in MATLAB und heurisko. Wesentlicher Grund für die Beliebtheit ist neben der Schnelligkeit der Algorithmen die Tatsache, dass FFTW die Transformation für Daten beliebiger Größe $N$ berechnen kann, während andere Lösungen sich aus algorithmischen Gründen auf solche $N$ beschränken, die eine Potenz von 2 sind.

Die Fouriertransformation (FT) ist ein wichtiges Werkzeug in Wissenschaft und Technik und in ihrer diskreten Variante [Bracewell 1986], der diskreten Fouriertransformation (DFT), auch

in der Bildverarbeitung. Würde man die DFT für ein Bild $G$ der Größe $M \times N$ mit den Grauwerten $g_{m,n}$ geradewegs gemäß ihrer Definition

$$f_{u,v} = \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} g_{m,n} \exp\left(-\frac{2\pi i m u}{M}\right) \exp\left(-\frac{2\pi i n v}{N}\right)$$

implementieren, hätte sie eine Zeitkomplexität von $O(N^4) = O(M^4)$ mit $M = cN$ für eine reelle Zahl $c$. Dies bedeutet, dass die Anzahl der benötigten Rechenschritte mit der vierten Potenz von der Problemgröße $N$ abhängt. Für ein Bild der Größe 512×512 müssten genauer etwa $5 \times 10^{11}$ Fließkommaoperationen ausgeführt werden, was bei einer Rechenleistung von 1 GFLOPS ($10^9$ Fließkommaoperationen pro Sekunde) etwa 500 s dauern würde. (Der Testrechner erreicht nach den bereits vorgestellten Messergebnissen bei Fließkommaadditionen etwa 0,2 GFLOPS.) Diese wäre mit Sicherheit nur in seltenen Fällen akzeptabel. Für die folgenden Überlegungen ist es praktisch, die Problemgröße $N$ als die Anzahl der Bildpunkte zu definieren. Dann hat der naive Algorithmus die Zeitkomplexität $O(N^2)$. Die gleichen Überlegungen gelten übrigens auch für die inverse FT, denn deren Definition ist

$$g_{m,n} = \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} f_{u,v} \exp\left(\frac{2\pi i m u}{M}\right) \exp\left(\frac{2\pi i n v}{N}\right).$$

Die FT wird erst durch schnelle Algorithmen praktikabel. Die meisten von ihnen beruhen auf der rekursiven Anwendung des Prinzips *Divide and Conquer* (Cooley und Tukey 1965, zitiert in [Blahut 1985]). Dabei wird das gestellte Problem dadurch gelöst, dass man es in Teilprobleme zerlegt, dann die Teilprobleme löst und schließlich die Teillösungen zur Gesamtlösung zusammensetzt. Man kann das Prinzip rekursiv weiterführen, indem man auch die Teilprobleme unterteilt, bis man schließlich auf Probleme stößt, die nicht mehr weiter unterteilbar sind oder eine triviale Lösung haben. Diese Technik führt nicht für jedes Problem zu einer geringeren Zeitkomplexität, aber bei der FT erhält man auf diesem Weg einen wesentlich schnelleren und damit praktikablen Algorithmus, weshalb man dann von *fast Fourier transform* (abgekürzt FFT) spricht.

Es gibt verschiedene FFT-Algorithmen, die sich durch die Art der Problemzerlegung unterscheiden. Eine Zerlegung ist immer dann möglich, wenn die Problemgröße keine Primzahl ist. Dies ist bei Bildern nie der Fall, da die Gesamtanzahl der Bildpunkte immer das Produkt aus Anzahl Zeilen und Spalten $N = N_x N_y$ ist. Für eine gegebene Problemgröße $N$ entspricht die Anzahl der Zerlegungsmöglichkeiten der Anzahl der Möglichkeiten, diese Zahl in Faktoren zu zerlegen. Für die FFT gibt es eine Reihe von Zerlegungsprinzipien [Besslich und Lu 1990]. Man kann die Zerlegung z. B. entweder im Orts- oder im Fourierraum vornehmen. Bei höheren Dimensionen kann man direkt zerlegen oder unter Ausnutzung der Separabilität der FFT eine Folge von 1-D-Transformationen ausführen [Blahut 1985]. Für alle Zerlegungsmöglichkeiten ist aber der Aufwand zum Zusammenfügen $O(N)$ [Besslich und Lu 1990]. Nach dem Master-Theorem [Cormen et. al. 2001] gilt dann mit einer rekursiven Zerlegung in $a > 1$ Teilprobleme der Größe $N/a$, d. h. mit der Rekursion

$$T(N) = aT\left(\frac{N}{a}\right) + \Theta(N),$$

für den FFT-Algorithmus eine Laufzeit von

$$T(N) = \Theta(N \log N).$$

Analoges gilt für die inverse Transformation. Verglichen mit dem Zeitbedarf des naiven FT-Algorithmus ist das eine enorme Zeitersparnis.

Damit haben zwar alle Varianten eines FFT-Algorithmus die gleiche Zeitkomplexität von $\Theta(N \log N)$, aber die Schreibweise $\Theta(f(N))$ für eine Funktion *f(N)* bedeutet lediglich, dass für die Anzahl Rechenschritte *t* eines Algorithmus gilt:

$$\exists c_1, c_2 : t(N) = c_1 f(N) + c_2 .$$

Die Konstanten $c_1$ und $c_2$ bestimmen mit, welchen tatsächlichen Rechenaufwand eine konkrete FFT-Variante für ein gegebenes *N* aufweist. Hinzu kommt, dass auch die Art der Speicherzugriffe die Ausführungszeit entscheidend beeinflusst (siehe Abschnitt 2.3.1).

Leider kann man auf Grund der Komplexität heutiger Rechnerarchitekturen Laufzeiten kaum vorhersagen. Deshalb ist eine sinnvolle Alternative, einen optimalen Algorithmus zu lernen, indem in einem Lernschritt mehrere Varianten ausprobiert und die Laufzeiten gemessen werden. FFTW besteht aus sorgfältig optimierten Blöcken von C-Code, so genannte *Codelets*, die einen Teil einer Transformation berechnen und miteinander kombiniert werden können. Eigentlich müsste FFTW zur Optimierung alle Varianten testen, was allerdings wegen der exponentiellen Zunahme der Kombinationsmöglichkeiten unvertretbaren Zeitaufwand zur Folge hätte. Deshalb wendet der Planer (*planner*) von FFTW das Prinzip des *dynamic programming* an und reduziert so den Suchraum. Das Resultat ist ein Plan, der für die gegebene Aufgabe im Sinne des angewandten Algorithmus die optimale Zusammenstellung von Codelets enthält. Dieser Plan wird gespeichert und bei der Ausführung einer FFT von der Steuereinheit (*executor*) abgerufen.

Tabelle 3-1 vergleicht die Laufzeiten der FFT und der inversen FFT für Objekte verschiedener Größen und für zwei unterschiedliche Implementierungen auf dem Testrechner. Der Vergleich zwischen heurisko 5 mit FFTW und heurisko 4 ohne FFTW zeigt, dass die FFTW-Implementierung mehr als doppelt so schnell ist. Die Ergebnisse zeigen außerdem, dass es ungünstige Objektgrößen gibt. Obwohl ein Bild der Größe 513×513 nur unwesentlich größer als eines mit 512×512 Bildpunkten ist, steigt die Laufzeit auf über das Dreifache an. Bei [FFTW] gibt es ausführliche Testberichte mit Laufzeitmessungen auf vielen Plattformen im Vergleich mit über 40 anderen FFT-Implementierungen. Die Ergebnisse zeigen, dass FFTW allen Implementierungen ohne hardware-spezifische Optimierungen überlegen und den für eine bestimmte Hardware optimierten Implementierungen ebenbürtig ist.

***Tabelle 3-1:*** *Ausführungszeiten in ms auf dem Testrechner für FFT und inverse FFT für Objekte verschiedener Größe mit einfachgenauen Fließkommazahlen. In heurisko 5 wird FFTW für beliebige Objektgrößen benutzt, in heurisko 4 ein eigener Basis-2-Algorithmus, bei dem die Objektgröße eine Potenz von 2 sein muss. Die Spalten sind mit den zugehörigen Operatornamen überschrieben. Alle Implementierungen in C ohne hardware-spezifische Optimierungen. Angegeben sind auch die Lernzeiten in s von FFTW, wenn noch keinerlei Vorwissen existiert.*

| | heurisko 5 mit FFTW | | | | heurisko 4 | |
|---|---|---|---|---|---|---|
| | `ftLearn` | `iftLearn` | `ft` | `ift` | `FT` | `FTinvers` |
| Objektgröße | [s] | [s] | [ms] | [ms] | [ms] | [ms] |
| 512×512 | 43 | 45 | 15 | 15 | 36 | 35 |
| 513×513 | 26 | 20 | 54 | 51 | - | - |
| 512×1024 | 47 | 49 | 33 | 32 | 80 | 79 |
| 1024×1024 | 50 | 51 | 71 | 68 | 173 | 170 |

### 3.3.2 Selbstlernende Texturanalyse

In der Bildverarbeitung gibt es bisher nur für Teilaufgaben Ansätze, selbstlernende Algorithmen einzusetzen. Als Anschauung für die gängige Vorgehensweise sei eines der wenigen publizierten Beispiele herausgegriffen. Wagner beschreibt ein sich automatisch konfigurierendes Bildverarbeitungssystem zur Texturanalyse und Objekterkennung, das in Abbildung 3.1 gezeigt ist [Wagner 2000]. Die Systemoptimierung besteht aus der Optimierung der Filter in der Vorverarbeitung, der Merkmalsauswahl, der Klassifikatorauswahl und gegebenenfalls einer Optimierung der Lernmusterauswahl. Die gleichzeitige Optimierung aller Systemparameter ist mit den heutigen Rechenleistungen nicht möglich. Wagner beschränkt sich deshalb auf die erwähnten Teiloptimierungen. Als Optimierungskriterien werden sowohl die Erkennungsleistung als auch die Rechenzeit herangezogen. Zum Lernen und Testen werden jeweils eigene Stichproben verwendet. Nach Abschluss der Optimierung muss die entstandene Systemkonfiguration objektiv bewertet werden. Dazu wird eine dritte Stichprobe, die Verifikationsstichprobe, benutzt. Sie soll zeigen, wie sich das System bei neuen, bisher noch nicht gesehenen Daten verhält. Wenn das System nämlich nicht genügend generalisiert, tritt ein Effekt auf, der mit *Auswendiglernen* bezeichnet wird und die bloße Anpassung des Klassifikators an das Lernmaterial bedeutet.

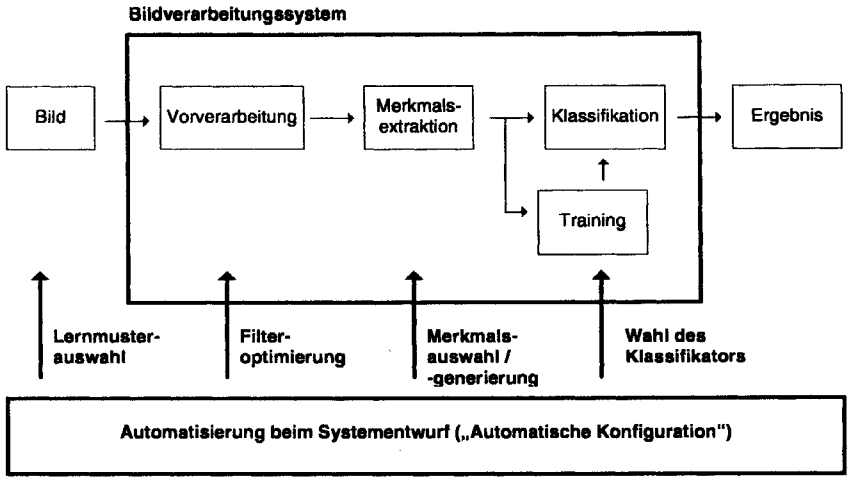


***Abbildung 3.1:*** *Automatische Konfiguration eines Bildverarbeitungssystems [Wagner 2000]*

Wagner führt aus, dass unter den Optimierungsproblemen bei der automatischen Konfiguration von Bildverarbeitungssystemen häufig drei Kategorien zu finden sind:

1. *Auswahlprobleme* treten bei der Suche nach geeigneten Merkmalsmengen oder einem optimalen Klassifikator auf. Die Anzahl der Kombinationsmöglichkeiten ist durch die Größe der Menge, aus der gewählt wird, beschränkt.

2. Eine *Parameteroptimierung* ist die Bestimmung optimaler Parameter. Dies können z. B. die Koeffizienten einer Faltungsmaske oder Schwellwerte für eine Segmentierung sein.

3. Bei der *Funktionengenerierung* geht es darum, aus Funktionsbausteinen neue Funktionen zu erzeugen, die ein Teilproblem optimal lösen. Ein Beispiel ist die Generierung neuer Texturmerkmale mit genetischer Programmierung.

Da die Parameteroptimierung auf Standardverfahren zurückgreift und weniger starke Auswirkungen auf die Leistung des Gesamtsystems hat, beschäftigt sich Wagner näher mit Auswahlproblemen und diskutiert ein Verfahren zur Funktionengenerierung.

Bei der Auswahl von Texturmerkmalen tritt das Problem auf, dass eine vollständige Suche auf Grund der großen Anzahl der möglichen Merkmale (mehrere Merkmalssätze aus der Literatur mit insgesamt 318 Merkmalen hat Wagner untersucht) nicht möglich ist. Da beliebig viele Merkmale aus $n$ Merkmalen ausgewählt werden können, gibt es $2^n$ Möglichkeiten. Deshalb exitieren eine ganze Reihe suboptimaler Suchverfahren, die so lange fortgeführt werden, bis ein Abbruchkriterium erreicht ist. Ein deterministisches Verfahren, das *Sequential Forward Search*, soll hier kurz skizziert werden. Von allen Merkmalen wird im ersten Schritt dasjenige mit der besten Erkennungsleistung gewählt. In weiteren Schritten wird aus den bisher nicht gewählten Merkmalen jeweils dasjenige hinzugenommen, das mit den bereits gewählten das beste Ergebnis erzielt. Ein Abbruchkriterium dieses Verfahrens kann sein, dass sich die Erkennungsleistung über mehrere Schritte hinweg nicht mehr signifikant ändert. Nichtdeterministische Verfahren ergeben sich bei Anwendung *genetischer Algorithmen*. Hier werden Bitstrings verwendet, deren Länge der Anzahl der zur Verfügung stehenden Merkmale entspricht. Ein Bitstring enthält für jedes gewählte Merkmal eine 1, ansonsten eine 0 und wird ein Individuum genannt. Das Verfahren beginnt mit einer Startpopulation von Individuen. Neue Individuen entstehen durch Mutation und Crossover. Bei einer Mutation werden Bitpositionen zufällig modifiziert und bei einem Crossover werden zwei Individuen an derselben Stelle zerschnitten und über Kreuz wieder zusammengefügt. Die neuen Individuen werden zu den alten hinzugefügt. Da die neue Population jedoch nicht größer sein soll als die alte, werden nach einer Bewertung die schlechtesten Individuen aussortiert. Auch dieses Verfahren läuft bis zum Erreichen eines Abbruchkriteriums.

*Genetische Programmierung* arbeitet nicht mit Bitstrings, sondern mit baumartigen Datenstrukturen. Ansonsten wird auch hier die genetische Entwicklung durch Mutation und Crossover vorangetrieben. Baumstrukturen eignen sich besonders dazu, Funktionen zu beschreiben. Da Merkmale mit Hilfe von Funktionen berechnet werden, kann man die genetische Programmierung dazu verwenden, neue Merkmale zu gewinnen. Umfangreiche Tests mit automatischer Auswahl optimaler Merkmale zeigen, dass bei Auswahl aus einem bekannten Merkmalssatz eine relative Verbesserung der Erkennungsleistung von bis zu 10 % und bei Auswahl aus einer Vielzahl von Merkmalssätzen von bis zu 20 % möglich ist. Die genetische Programmierung konnte Wagner wegen des hohen Rechenleistungsbedarfes trotz Parallelisierung nur eingeschränkt testen. Deshalb kann bisher nur festgestellt werden, dass es sich um einen viel versprechenden Ansatz handelt, den es sich lohnt bei weiter steigender Rechenleistung zu verfolgen.

Die Klassifikatorauswahl spielt nach Wagner keine so große Rolle bei der Leistungsfähigkeit eines Mustererkennungssystems. Da jedoch nur *ein* Klassifikator zu wählen ist, lässt sich mit wenig Aufwand durch Vergleich der zur Verfügung stehenden Klassifikatoren eine optimale Wahl vornehmen.

# 4 Bausteine für ein Bildverarbeitungssystem

## 4.1 Entwurfsprinzipien

Die Analyse der Lösungskonzepte für ein flexibel einsatzfähiges und gleichzeitig leistungsstarkes Bildverarbeitungssystem im vorangegangenen Kapitel ergab ein klares Anforderungsprofil. Dies wird nun in diesem Kapitel der Ausgangspunkt für den Entwurf des Bildverarbeitungssystems heurisko sein. Dessen Entwurfskriterien werden in diesem Abschnitt einleitend diskutiert.

**Bibliotheksfunktionen und Skriptsprache:** Bedingt durch die Komplexität und Vielfältigkeit der Bildverarbeitungsfunktionen ist ein sorgfältiger Entwurf der Bildverarbeitungssoftware notwendig, damit sie für den Benutzer überschaubar bleibt. Das ist nur möglich, indem die Bildverarbeitungsfunktionen systematisch analysiert und in Klassen eingeteilt werden. Darüber hinaus ist zu prüfen, wie durch einen hierarchischen internen Aufbau der Bibliothek eine optimale Wiederverwendbarkeit von Code gewährleistet werden kann. Dies ist auch wichtig unter dem Gesichtspunkt der Optimierung einzelner Funktionen.

Weiterhin sollten die Funktionsbausteine der Bibliothek so gebaut sein, dass sich daraus in möglichst kompakter, flexibler und zugleich übersichtlicher Art komplexe Algorithmen erstellen lassen. Es erscheint sinnvoll zu prüfen, ob die Bibliothekbausteine mit einer Skriptsprache verknüpft werden können, um eine schnelle Entwicklung und Änderung von Algorithmen zu ermöglichen. Das ist nur sinnvoll, wenn das Interpretieren der Skriptsprache im Vergleich zu den aufgerufenen Bildverarbeitungsfunktionen keine nennenswerte Rechenzeit kostet. Nur dann kann das System auch effizient für praktische Aufgaben in Forschung und Industrie eingesetzt werden. Wichtig ist weiterhin, dass die Skriptsprache Elemente enthält, die wesentlich für eine effektive Bildverarbeitung sind. Dazu gehören der einfache und schnelle Zugriff auf Datenausschnitte (so genannte Areas-of-Interest), die übliche Programmflusssteuerung und auch effektive Möglichkeiten für Schleifen.

**Einheitliche Datenstruktur für multidimensionale Bildverarbeitung:** Die Komplexität einer Bildverarbeitungsbibliothek wird auch entscheidend durch die Datenstrukturen geprägt. Der gesamte Entwurf der Bibliothek ließe sich vereinfachen, wenn es möglich wäre, eine einheitliche Datenstruktur für alle Ebenen der Bildverarbeitung zu benutzen. Es ist offensichtlich, dass dieser Ansatz auch die Entwicklung einer Skriptsprache entscheidend vereinfacht.

Ein langfristig angelegtes System darf nicht auf die Bearbeitung zweidimensionaler Bilder beschränkt bleiben, sondern sollte von vorneherein auch für die Bearbeitung höherdimensionaler Daten wie Bildfolgen, Volumenbilder und spektroskopische Bilder gleichermaßen geeignet sein. Dieser Ansatz ist besonders hervorzuheben, da fast alle gängigen Bildverarbeitungspakete für die Analyse zweidimensionaler Bilder konzipiert wurden. Wenn überhaupt multidimensionale Daten verarbeitet werden können, dann geschah dies durch nachträgliche Änderungen. Das ist mit entsprechenden Brüchen im Konzept verbunden und führt zu einer schlechteren Unterstützung für höherdimensionale Daten, d. h. nur ein deutlich geringerer Funktionsumfang steht für sie zur Verfügung.

**Datentypen:** Im Gegensatz zu numerischen Bibliotheken werden in der Bildverarbeitung eine Fülle von Datentypen benötigt. Wie in der Einleitung diskutiert, werden neben Gleitkommazahlen auch diverse ganze Zahlen gebraucht. Auch dieser Umstand muss beim Entwurf der

Bildverarbeitungsbibliothek sorgfältig berücksichtigt werden. Es erscheint sinnvoll, den datentypunabhängigen Teil der Algorithmik sorgfältig vom datentypabhängigen zu trennen. Dadurch kann der unabhängige Teil der Algorithmik für alle Datentypen gemeinsam verwendet werden. Gleichzeitig ist es leichter, den abhängigen Codeteil mit im Idealfall deutlich geringerem Umfang gezielt zu optimieren, z. B. durch Benutzung der Multimedia-Instruktionssätze. Ein solches Vorgehen erfordert natürlich eine systematische Analyse und Klassifizierung der möglichen Algorithmen für die Bildverarbeitung.

**Datenein- und -ausgabe:** Die Unterstützung der Bildakquisition ist ein zentrales Element jedes Bildanalysesystems, wie es in Forschung und Industrie benötigt wird. Dabei ist eine offene Schnittstelle wegen der Vielfältigkeit der Bildakquisitionssysteme unerlässlich. Da die Bildaufnahme mehr und mehr integraler Bestandteil von Messdatenerfassungssystemen wird, müssen auch entsprechende Schnittstellen zur Akquisition analoger und digitaler Messdaten und zur Steuerung und Regelung von Prozessen vorhanden sein. Dies gilt nicht nur für industrielle Anwendungen, sondern zunehmend auch für den Einsatz in der Forschung.

**Benutzerschnittstelle:** Die Art der Benutzerschnittstelle hängt sowohl von der Aufgabenstellung als auch von den Vorlieben und Erfahrungen des Benutzers ab. Daher ist es ein wichtiges Entwurfsziel, die eigentlichen Bildverarbeitungsroutinen streng von der Benutzerschnittstelle zu trennen und eine offene und einfache Schnittstelle zwischen dem Bildverarbeitungskern und der Benutzerschnittstelle zu definieren. Es sollte möglich sein, das System mit unterschiedlichen Benutzerschnittstellen einzusetzen.

**Modularität und Portabilität:** In die Entwicklung eines Bildverarbeitungssystems müssen viele Mannjahre investiert werden. Von daher ist sehr genau zu überlegen, welche Plattformabhängigkeiten man sich einhandeln möchte. Im Idealfall ist das gesamte System plattformunabhängig gestaltet, so dass es bei Bedarf auf beliebige Plattformen portiert werden kann. Insbesondere für graphische Benutzeroberflächen ist das nicht so einfach zu erreichen. Ein modular aufgebautes System erlaubt, von Plattformen unabhängige und abhängige Code-Bereiche voneinander zu trennen. Das wiederum ermöglicht in übersichtlicher Weise, auf einer Plattform entweder auf ein abhängiges, aber nicht zwingend benötigtes Modul zu verzichten oder ein plattformspezifisches Modul einzusetzen. Die Austauschbarkeit der Module ist auch für die Wartung und lokale Optimierung eine Erleichterung. Nicht zuletzt sind modulare Systeme auch leicht erweiterbar, wenn entsprechende Schnittstellen vorhanden sind. Dies ist nicht nur für den Entwickler des Systems ein Vorteil, sondern auch für den Anwender, der eigene Erweiterungen in das System einbringen möchte.

Zur Orientierung für die weiteren Kapitel zeigt Abbildung 4.1 in einem vereinfachten Blockdiagramm den modularen Aufbau von heurisko. In der oberen Hälfte ist der Interpreter zu sehen, der aus den ihm übergebenen Kommandos einen Binärcode erzeugt. Diesen Code leitet er an die Ausführungseinheit im Kernmodul weiter. Der Interpreter ist mit Hilfe der angedeuteten Kommunikationsmodule in der Lage, mit mehreren Kernen zusammenzuarbeiten, wobei sich die Kerne nicht im gleichen Rechner befinden müssen. Der Kern enthält neben der Ausführungssteuerung Hilfsfunktionen und Schnittstellen zu den so genannten Erweiterungsmodulen, in denen der größte Teil der Funktionalität der heurisko-Operatoren implementiert ist. Erweiterungsmodule heißen sie, weil sie den Kern um ihre Funktionalität erweitern und weil der gleiche Mechanismus dazu verwendet werden kann, um andere Software in heurisko einzubinden und in Form von Operatoren dem Anwender zugänglich zu machen. Für die Datenerfassung, für die Visualisierung von Daten und für die Ein- und Ausgabe über Dateien verfügt heurisko zudem über drei erweiterte Schnittstellen, welche jeweils die spezielle Art der Funktionalität berücksichtigen und die Erstellung entsprechender Erweiterungsmodule erleichtern.

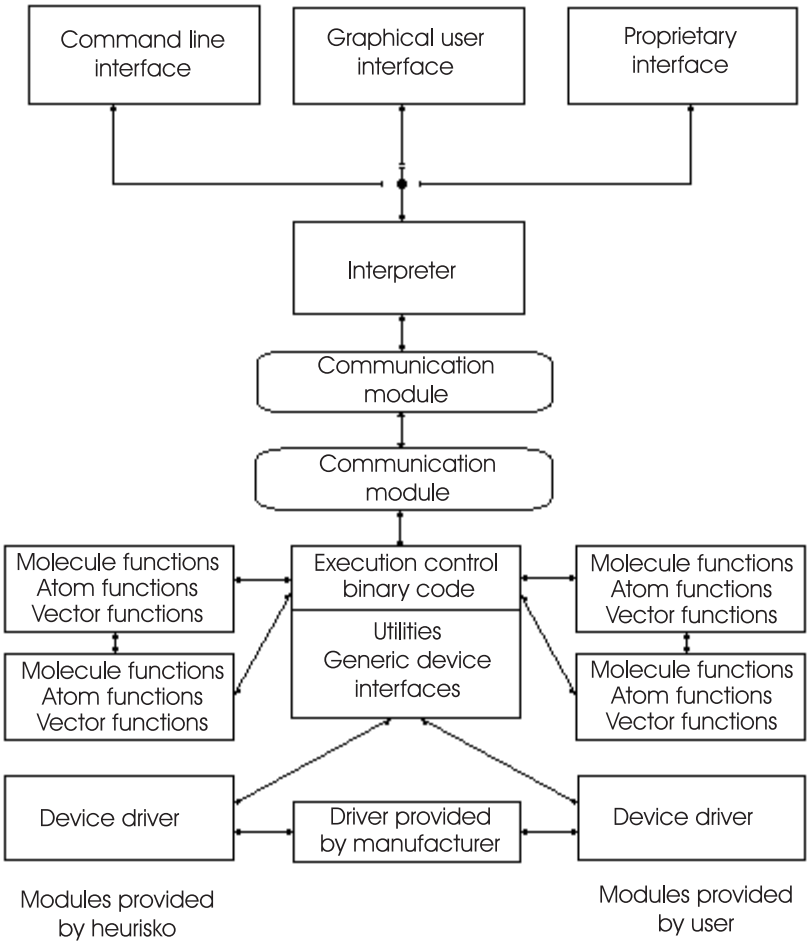


***Abbildung 4.1:*** *Blockdiagramm der Architektur heuriskos*

Für die Steuerung des Interpreters benötigt man ein Steuerprogramm. Solch ein Steuerprogramm kann, muss aber keine graphische Benutzeroberfläche enthalten. Wichtig ist, dass jedes Steuerprogramm ein separates Modul ist und, im Diagramm durch den Drehschalter angedeutet, leicht ausgetauscht werden kann. Heurisko wird mit zwei Steuerprogrammen geliefert, eines mit einer simplen Kommandozeileneingabe, das andere mit einer komfortablen graphischen Benutzeroberfläche. Der Anwender kann den Interpreter stattdessen beispielsweise mit dem Programm einer Fertigungszelle in einer Fabrik steuern.

Weitere Details zu den heurisko-Komponenten werden in den nächsten Abschnitten beschrieben. Mit den heurisko-Objekten und deren Datenstrukturen für die Bildverarbeitung beschäftigt sich Abschnitt 4.2. Daran anschließend werden die Operatoren, die auf diesen Objekten arbeiten, näher betrachtet (4.3). Die Implementierung der Operatoren befindet sich teils in Erweiterungsmodulen, weshalb diese in Abschnitt 4.4 vorgestellt werden. Der darauf folgende Abschnitt 4.5 zeigt, wie in speziellen Erweiterungsmodulen die Unterstützung für Dateien, Datenerfassung und Visualisierung erfolgt. Der letzte Abschnitt behandelt den Interpreter und die graphische Benutzeroberfläche (4.6).

## 4.2 Effektive Datenstrukturen für die Bildverarbeitung

### 4.2.1 Ausgangspunkt

Bilddaten liefernde Quellen reichen von Zeilen- und Flächenkameras über Röntgengeräte bis zu Sensoren, deren Signale nichts mit einer optischen Abbildung zu tun haben, die aber trotz-

dem zu Bildern zusammengesetzt werden. So entstehen Bilddaten unterschiedlicher Dimensionen. Dreidimensionale Daten erhält man unter anderem bei der Aneinanderreihung von zu unterschiedlichen Zeitpunkten aufgenommenen 2-D-Bildern oder durch Aufnahmen von Körpern in verschiedenen Schnittebenen. Ein schon heute realisierstes Beispiel für vierdimensionale Daten sind von einem Satelliten in vielen Längenwellenbereichen aufgenommene Bildsequenzen mit der Wellenlänge als vierte Dimension. Auch zur Verarbeitung der Bilder werden Datenstrukturen benötigt. Für Faltungen braucht man Faltungskerne, deren maximale Dimension derjenigen der zu faltenden Bilddaten gleich ist. Histogramme sind eindimensionale Bildverarbeitungsobjekte. Zusätzlich besteht Bedarf an homogenen und heterogenen Verbundobjekten. Mehrkanalbilder sind homogene Verbundobjekte, bei denen alle Komponenten des Verbunds den gleichen Datentyp und die gleiche Gestalt haben. Dagegen benötigt man für die Merkmalsbeschreibung heterogene Verbundobjekte. Interessierende Eigenschaften eines Bildes seien beispielsweise für segmentierte Objekte deren Schwerpunktpositionen, die Größe der Objektflächen und statistische Werte. Verbundobjekte sorgen hier für eine bessere Ordnung und sind außerdem eine gute Möglichkeit, komplexe Daten effizient als einen einzigen Parameter einer Funktion zu übergeben. Für eine hierarchische Anordnung von Teilobjekten braucht man schließlich auch noch Bäume.

Wie immer bei komplexeren Daten muss es möglich sein, auf beliebige Teilbereiche der Daten zugreifen zu können. Für Bilder haben sich die englischen Ausdrücke *region of interest* (ROI) oder *area of interest* (AOI) eingebürgert. Dies sind im einfachsten Fall rechteckige Bereiche innerhalb eines 2-D-Bildes. Im Folgenden wird *AOI* allgemeiner als Abkürzung eines Begriffs für einen Teilbereich eines Objektes verwendet. In diesem Sinne ist dann auch ein Bild aus einer Bildsequenz eine AOI. (AOI soll feminin sein, da A für die englische Übersetzung *area* des deutschen Wortes *Fläche* steht.)

Eine weitere Vielfalt der Bilddaten entsteht durch die Notwendigkeit und Zweckmäßigkeit unterschiedlicher Datentypen. Monochrome Kameras liefern ganze Zahlen mit 8, 10 oder 12 Bit pro Bildpunkt. Dabei werden 10- und 12-Bit-Grauwerte im Rechner in 16-Bit-Worten gespeichert. Viele Farbkameras legen die Daten eines Bildpunktes in 32-Bit-Worten ab, je 8 Bit für die drei Farbkanäle, 8 Bit bleiben unbenutzt. Andere benutzen aus Speichereffizienzgründen und zur Begrenzung der Datenrate nur 24 Bit je Bildpunkt. Dann wiederum gibt es Geräte wie Computer-Tomographen, die Fließkommadatentypen liefern. Bei der Verarbeitung von Bilddaten und bei der Darstellung von Ergebnissen kommen noch andere Datentypen hinzu. Binärwerte eignen sich für die Segmentierung von Objekten, komplexe Zahlen braucht man für die Fouriertransformation, Fließkommazahlen doppelter Genauigkeit dienen zu genaueren Berechnungen, 64-Bit-Zahlen können für große Zahlen sinnvoll sein und ohne Zeichenketten kommt man auch nicht aus.

### 4.2.2 Atome und Moleküle

Vielleicht der wichtigste Aspekt von Datenstrukturen ist die Effizienz der darauf realisierbaren Algorithmen. Ein bekanntes Beispiel sind binäre Bäume, bei denen das Suchen eines Elements unter $n$ Elementen von der Ordnung $O(\log n)$ ist. In einer Liste würde der Aufwand für das Suchen dagegen von der Ordnung $O(n)$ sein. Dies ist für genügend große $n$ ein enormer Unterschied. Es lohnt sich also, über geschickte Datenstrukturen nachzudenken.

In heurisko gibt es zur Erfüllung aller erwähnten Anforderungen für die interne Verwaltung eines Objektes eine *einzige*, etwas komplexere und von der Gestalt des Objekts unabhängige Datenstruktur. Der Benutzer der Skriptsprache merkt von dieser Struktur nichts. Für ihn ist ein Skalar trotzdem nur ein einfaches Objekt mit einem einzigen Datenpunkt, und ein Verbundobjekt besteht für ihn aus genau den Komponenten, die er in der Strukturdefinition

angegeben hat, obwohl beide Objekte intern mit Hilfe der gleichen Datenstruktur gespeichert sind.

Um nun auch heterogene Objekte verwalten zu können, lassen sich beliebig viele Instanzen der erwähnten Datenstruktur über Verweise baumartig miteinander verknüpfen. Eine Instanz wird in heurisko mit dem der Atomphysik entlehnten Begriff *Molekül* bezeichnet. Die Gesamtheit aller möglichen Verknüpfungen lässt sich am besten mit Begriffen der Graphentheorie ausdrücken:

> Jedes interne *heurisko-Objekt* bildet einen *Baum* mit einer ausgezeichneten *Wurzel* und gerichteten Kanten. Jeder Knoten des Baumes ist ein Molekül. Alle internen Knoten, und wenn es sich nicht um den trivialen Baum mit nur einem Knoten handelt, auch die Wurzel, enthalten keine Daten, sondern nur Verweise auf Kindknoten. Lediglich die Blätter des Baumes speichern beliebig-dimensionale, aber homogene Daten.

Deshalb werden die Blattknoten auch als *Atome* bezeichnet. Die Daten eines Atoms bilden also ein Feld beliebiger Dimension. (Nur zur Begrenzung des Speicherplatzes für Verwaltungsdaten ist die Dimension zurzeit mit einem einstellbaren Schalter für den Compiler auf fünf beschränkt.) Die internen Datenstrukturen für Moleküle und Atome heißen `hMOLECULE` und `hATOM`.

```
// Molekül:

typedef struct hMOLECULE {
    hSINT_PTR n;         // Contains <n> sub molecules
    hBITS32 flag;        // Flags
    hSINT32 handle;      // Handle to acquisition and
                         // control device or inspector
    void* d1;            // Pointer to auxiliary data
    struct hIV_DEV* iv1; // Pointer to inspector device
    struct hMOLECULE* down2[1]; // Pointer to <n> sub molecules
                                // or atoms
} hMOLECULE;
```

Es soll nicht auf alle Einzelheiten von `hMOLECULE` eingegangen werden. Wesentlich ist, dass anhand der ersten Komponente festgestellt werden kann, wie viele Submoleküle das Molekül hat. Die Zeiger auf diese Submoleküle stehen in dem Feld in der letzten Komponente.

```
// Atom:

typedef struct hATOM {
    hSINT_PTR n;    // Contains <n> sub molecules (always
                    // negative)
    hBITS32 flag;   // Flags
    hSINT32 handle; // Handle to acquisition and control device
                    // or inspector
    void* d1;       // Pointer to allocated data
    struct hATOM* parent1; // Pointer to atom carrying pointer
                    // list and data from which this atom is AOI
    hATOM_PROP* aprop1; // Pointer to properties of object
                        // (shared by all AOI objects)
    void* aux1;     // Pointer to auxiliary properties and data
    void** ptr2;    // Pointer to list of pointers to data
                    // (shared by all AOI and alias objects)
    hUINT_PTR pn;   // Number of vector pointers
```

```
        hUINT_PTR vn;   // Length of vectors
        hUINT_PTR vo;   // Offset in vectors
        hUINT_PTR dim;  // Dimension of object
        hUINT32 type;   // Type of data
        hUINT32 bits;   // Number of bits (including sign bit!)
        hUINT32 usage;  // Counter of multiple usage (remove only
                        // if usage == 0)
        hUINT32 childs; // Counter for children
        char* format1;  // Pointer to format string
        hATOM_STRUCT* astruct1; // Pointer to actual dimension
        hATOM_STRUCT astruct[H_MAX_DIM]; // Structure for
                        // description of H_MAX_DIM dimensions
    } hATOM;
```

Die Atomstruktur gleicht in ihren ersten Komponenten der Molekülstruktur. Deshalb können Atome auch als Submoleküle fungieren, wie in der Baumdefinition eines heurisko-Objekts angegeben. Wenn die erste Komponente eines Submoleküls negativ ist, weiß man, dass man ein Atom, also ein Blatt des Baumes, vor sich hat. Hinter der Komponente `ptr2` verbirgt sich ein Zeiger auf die Daten (siehe unten), deren Typ in der Komponente `type` gespeichert ist.

***Tabelle 4-1:*** *Die Datentypen in heurisko. Die Typen* `huge` *und* `uhuge` *passen sich an die Rechnerarchitektur an.*

| heurisko-Typ Interpreter | heurisko-Typ Def. im Kern | Bits je Datenpunkt | Beschreibung | zusätzliche gepackte Typen |
|---|---|---|---|---|
| `binary` | `hBITS32, hBITS64` | 1 | vorzeichenlose ganze Zahl | immer gepackt, 32 oder 64 Bit |
| `byte` | `hSINT8` | 8 | vorzeichenbehaftete ganze Zahl | `byte2, byte3, byte4` |
| `ubyte` | `hUINT8` | 8 | vorzeichenlose ganze Zahl | `ubyte2, ubyte3, ubyte4` |
| `short` | `hSINT16` | 16 | vorzeichenbehaftete ganze Zahl | `short2, short3, short4` |
| `ushort` | `hUINT16` | 16 | vorzeichenlose ganze Zahl | `ushort2, ushort3, ushort4` |
| `long` | `hSINT32` | 32 | vorzeichenbehaftete ganze Zahl | `long2, long3, long4` |
| `ulong` | `hUINT32` | 32 | vorzeichenlose ganze Zahl | `ulong2, ulong3, ulong4` |
| `huge` | `hSINT64` | 32 oder 64 | vorzeichenbehaftete ganze Zahl | `huge2, huge3, huge4` |
| `uhuge` | `hUINT64` | 32 oder 64 | vorzeichenlose ganze Zahl | `uhuge2, uhuge3, uhuge4` |
| `float` | `float` | 32 | einfachgenaue reelle Zahl | `float2, float3, float4` |
| `double` | `double` | 64 | doppeltgenaue reelle Zahl | |
| `fcomplex` | `hcFlOAT` | $2 \times 32$ | zwei einfachgenaue reelle Zahlen | |
| `dcomplex` | `hcDOUBLE` | $2 \times 64$ | zwei doppeltgenaue reelle Zahlen | |
| `string` | `char` | 8 | Zeichenkette beliebiger Länge | |

Tabelle 4-1 zeigt die von heurisko unterstützten Datentypen. Die erste Spalte enthält die Typbezeichnungen, die im Skript Verwendung finden. Die zugehörigen Definitionen im heurisko-Kern sind in der zweiten Spalte aufgeführt. Mit Aufkommen von 64-Bit-Prozessoren wurde ein neuer ganzzahliger Datentyp mit 64 Bit möglich. Damit man schon heute auf 32-Bit-Systemen Programme für 64-Bit-Systeme schreiben kann, wurden dazu neue Typen für ganze Zahlen eingeführt, die der Compiler entsprechend dem Zielsystem in ganze Zahlen mit 32 oder mit 64 Bit umsetzt. Die zugehörigen heurisko-Typen sind `huge` und `uhuge`. Eine Besonderheit sind binäre Zahlen. Aus Effizienzgründen packt heurisko je nach System 32 oder 64 binäre Datenpunkte in eine einzige ganze Zahl. In der letzten Tabellenspalte sind weitere gepackte Datentypen aufgeführt. Diese Datentypen sind sinnvoll zur Behandlung von Mehrkanalbildern. Das bekannteste Anwendungsbeispiel sind Farbbilder, bei denen alle drei Farbwerte für einen Bildpunkt hintereinander im Speicher liegen. Eine andere Möglichkeit für Farbbilder wäre, sie in drei getrennten Bildern zu speichern. Mit den Operatoren `Pack()` und `Unpack()` kann man in heurisko Objekte von der einen in die andere Darstellung umkopieren.

Die aktuelle Realisierung des Interpreters lässt noch gar nicht zu, das interne Objektverwaltungskonzept vollständig auszuschöpfen. Ein paar Beispiele sollen zeigen, was bislang möglich ist. In der ersten Zeile ist jeweils angegeben, wie das Objekt im Skript zu deklarieren ist. Danach folgen Angaben über Moleküle und Atome im Innern von heurisko, von denen der Skriptentwickler jedoch nichts wissen muss. Die Syntax für die Objektdefinition ist an die C-Syntax angelehnt. In der Dimensionsangabe steht links die am langsamsten laufende Koordinate, rechts die am schnellsten laufende (häufig mit x bezeichnet).

| | |
|---|---|
| **Skalar:** | `ulong size;` |
| Molekül und Atom: | `size` |
| **Bildsequenz** (Grauwerte): | `ubyte seq[100][480][640];` |
| Molekül und Atom: | `seq` |
| **Bildliste** (Grauwerte): | `ushort list/100/[480][640];` |
| Molekül: | `list` |
| Atome (100): | `list/0/, list/1/, …,list/99/` |
| **Bildliste** (3 Farbwerte): | `struct collist/100/[480][640] {ubyte b, g, r};` |
| Wurzelmolekül: | `collist` |
| Submoleküle (100): | `collist/0/, clolist/1/, …, collist/99/` |
| Atome (100×3): | `collist/i/.b, collist/i/.g, collist/i/.r` |

Beim ersten Beispiel handelt es sich um einen Skalar mit Namen `size` vom Typ `ulong`. Das zweite Beispiel ist ein dreidimensionales Objekt mit Namen `seq`, das eine Bildsequenz mit 100 Bildern der Größe 640×480 vom Typ `ubyte` darstellt. Das dritte Beispiel ist ein Molekül namens `list` mit 100 gleichen Atomen. Jedes Atom ist ein Bild der Größe 640×480 vom Typ `ushort`. Der Unterschied zwischen dem zweiten und dem dritten Beispiel ist die Dimension. Während für `seq` dreidimensionale Operationen durchgeführt werden können, ist

das bei `list` nicht möglich. Das letzte Beispiel ist nochmals eine Bildliste, dieses Mal aber mit Farbbildern. Jedes der 100 Farbbilder hat drei Farbkomponenten der Größe 640×480. Die Molekülstruktur besitzt drei Ebenen mit dem Objekt `collist` als Wurzel. Diese hat 100 Submoleküle mit jeweils drei Atomen.

### 4.2.3 Daten im Speicher

Solange im Laufe der Analyse eines Bildes noch mit dem Bild selbst gearbeitet wird (*ikonische* im Gegensatz zur *symbolischen* Bildverarbeitung), sind die Algorithmen dadurch geprägt, dass sie einen Bildpunkt nach dem anderen verarbeiten. Hierfür eignet sich die lückenlose Speicherung der Bildpunkte. Üblicherweise liefern bildgebende Systeme die Bilder zeilenweise immer von links nach rechts, beginnend mit dem Punkt in der linken oberen Ecke und endend in der rechten unteren Ecke. Dies ist auch für Bildquellen nach Videonorm so. Obwohl die beiden Halbbilder zu verschiedenen Zeitpunkten aufgenommen werden, stehen sie nicht getrennt im Speicher. Die Treiber der Bilderfassungshardware sorgen für die Speicherung als ein Gesamtbild. Für die Verarbeitung solcher Bilddaten wäre es ausreichend zu wissen, an welcher Speicheradresse die Bilder beginnen, wie viele Bildpunkte sie haben und wie die Farbwerte einzelner Bildpunkte im Speicher angeordnet sind (z. B. 8 Bit pro Bildpunkt für monochrome Bilder oder 32 Bit je Bildpunkt für Farbbilder mit 8 Bit je Farbe in der Reihenfolge B, G, R und 8 unbenutzten Bit). Oft wird dafür gesorgt, dass Bildzeilen an Adressen beginnen, die durch eine bestimmte Zahl, etwa 256, ohne Rest teilbar sind. Dann kommt es auf die Anzahl Punkte je Bildzeile an, ob am Ende jeder Zeile unbenutzter Speicher übrig bleibt. In diesen Fällen muss man zur Verarbeitung der Bilddaten außer der tatsächlichen Bildzeilenlänge noch die Zeilenlänge im Speicher (*pitch*) wissen. Insgesamt ist die zusätzlich zu den eigentlichen Bilddaten gespeicherte Information also unabhängig von der Bildgröße. Für den Zugriff auf einen bestimmten Bildpunkt lässt sich daraus in konstanter Zeit (*O(1)*) dessen Speicheradresse berechnen.

Es ist allerdings vorteilhaft, bei der Datenorganisation etwas mehr Aufwand zu treiben. Statt sich nur die Anfangsadresse eines Bildes zu merken, kann man in einem eindimensionalen Feld die Adressen aller Zeilenanfänge speichern. Dieses Konzept ist sehr leicht auf beliebige Dimensionen ausdehnbar. Jedes Objekt mit einer Dimension größer als zwei lässt sich in 2-D-Bildscheiben zerlegen. Man verlängert dann einfach das Zeigerfeld und speichert nacheinander die Zeilenanfänge dieser Bildscheiben und merkt sich außerdem die Ausdehnung des Objektes in jeder Richtung. Bezeichnet man die Koordinaten eines *n*-dimensionalen Objektes mit $x_0$, $x_1$, … , $x_{n-1}$ und die Koordinatenlängen mit $m_0$, $m_1$, … , $m_{n-1}$, so ergibt sich für die Zeilenzeiger das Speicherschema in Abbildung 4.2. Die Zeile $i_1$, … , $i_{n-1}$ hat demnach den Index

$$i = i_{n-1} m_{n-2} m_{n-3} \cdots m_1 + i_{n-2} m_{n-3} \cdots m_1 + \cdots + i_2 m_1 + i_1$$

in der Zeigerliste, und der Wert des Bildpunktes mit den Indices $i_0$, $i_1$, … , $i_{n-1}$ ist `*(type*)p2[i+i0]`, wobei `type` der Datentyp der Bildpunkte ist.

Weiter unten, unter anderem in Abschnitt 4.3.4, wird dargestellt, welche Vorteile das Speichern der Zeilenadressen für die Algorithmik hat. Ein anderer positiver Nebeneffekt ist, dass fragmentierter Speicher gut ausgenutzt werden kann. Im Testrechner mit Windows XP ist 1 GB RAM installiert. Es war in vielen Versuchen bereits direkt nach dem Rechnerstart nicht möglich, (mit der C-Funktion `malloc()`) mehr als 600 MB an einem Stück zu belegen, obwohl noch mehr Speicher frei war. Das liegt daran, dass der Speicher-Manager selbst entscheidet, wohin er benötigte Bibliotheken lädt. Schon alleine hierdurch entsteht eine Fragmentierung des zur Verfügung stehenden Speichers. Man könnte nun trickreich versuchen, den Speicher-Manager zur Verschiebung der Bibliotheken zu veranlassen und gleich nach

dem Start den gewünschten Speicher zu belegen. Dies ist jedoch aus mehreren Gründen völlig unbefriedigend. Erstens ist eine solche Lösung plattformabhängig. Zweitens ist der Speicher-Manager nicht dazu geschaffen, ihm ins Handwerk zu pfuschen, weshalb die möglichen Tricks auch nicht dokumentiert sind. Drittens kann man sich nicht darauf verlassen, dass die angewandten Tricks in der nächsten Version des Betriebssystems auch noch funktionieren. Die Vektorzeigerliste legt den Gedanken nahe, große Speicheranforderungen transparent für den Anwender durch mehrere kleinere Stücke zu befriedigen. Die Strategie könnte sein, es zuerst mit einem Block der gesamten angeforderten Menge zu versuchen. Schlägt der Versuch fehl, probiert man es mit zweimal der halben Größe oder mit einer anderen Aufteilung.

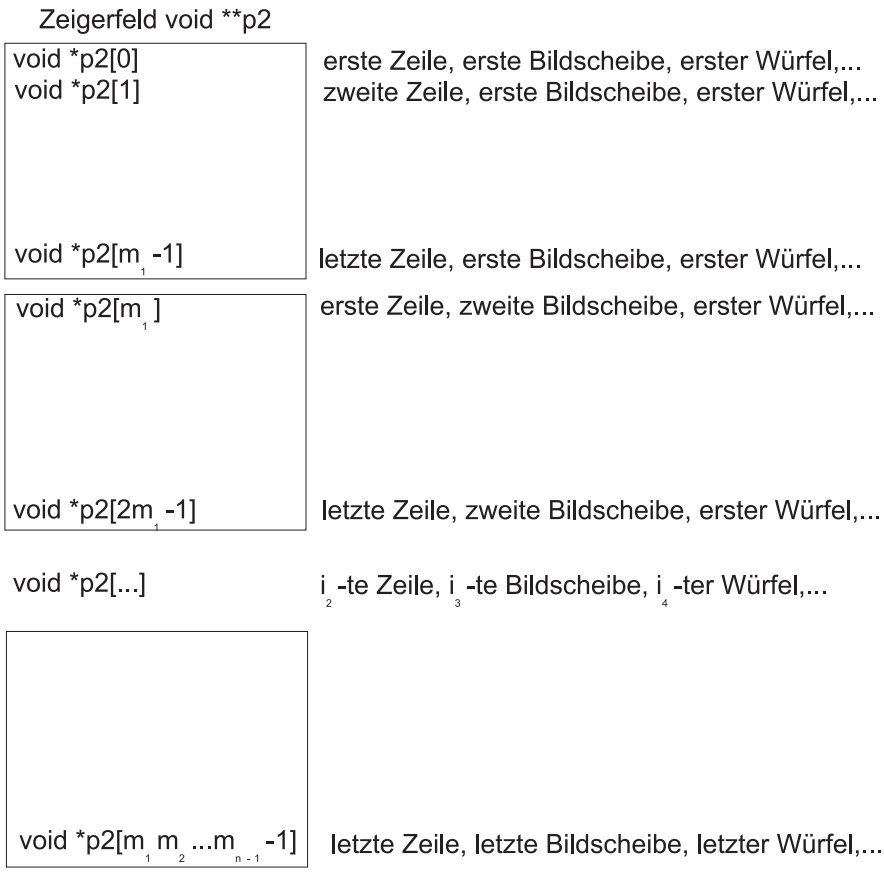


***Abbildung 4.2:*** *Speicherschema eines Atoms*

### 4.2.4 AOIs und andere Kindobjekte

Weiter oben wurde die AOI eingeführt als ein Teilbereich eines Objektes. Benötigt wird eine Syntax, die es erlaubt, eine Operation auf eine AOI zu beschränken. Welche Form die AOI haben muss, hängt von der Operation ab. In heurisko sind zurzeit vier Möglichkeiten implementiert, die anhand der Objekte

```
uybte seq[100][480][640];  # Sequenz mit 100 Bildern
uybte list/100/[480][640]; # Liste mit 100 Bildern
struct colim[480][640] {ubyte b, g, r}; # Farbbild (3 Komp.)
struct aoi3d[3] {ushort point, length}; # Quader
```

demonstriert werden sollen:

- `seq[start:length]:` Durch den Startindex `start` und die Länge der AOI `length` in einfachen eckigen Klammern wird ein Bereich bezüglich einer Richtung eines Atoms selektiert. In allen anderen Richtungen wird die Operation auf das

gesamte Objekt abgewandt, es sei denn, auch für sie ist eine AOI spezifiziert. Das angegebene Beispiel selektiert einen Teil der 100 Bilder in der Sequenz. Wollte man einen Zeilenbereich in allen Bildern auswählen, müsste man `seq[][start:length]` angeben. Die leeren Klammern (hier für z) besagen, dass in dieser Richtung das vollständige Objekt gemeint ist. Leere Klammern hinter der letzten Auswahl lässt man weg (hier für x). Es existieren ein paar weitere abkürzende Schreibweisen. Fehlt der Startindex, beginnt die AOI beim Index 0. Bei fehlender Längenangabe reicht die AOI bis zum Ende. Wird nur ein Zahlenwert ohne Doppelpunkt in die Klammern geschrieben, wird der dem Zahlenwert entsprechende Index selektiert.

- `seq[[aoi3d]]`: Mit doppelten eckigen Klammern spezifiziert man eine Verallgemeinerung der AOI mit einfachen Klammern auf beliebige Dimensionen und damit auf einen Hyperwürfel. Die Komponente `point` im Objekt `aoi3d` gibt den Startpunkt der AOI an und die Komponente `length` die Ausdehnung der AOI in allen Richtungen des Atoms. Diese Schreibweise ist insbesondere dann praktisch, wenn das die AOI spezifizierende Objekt das Ergebnis eines Operators ist und direkt in den nächsten Operator eingegeben werden kann. In heurisko kommt das beispielsweise bei der interaktiven Bereichsselektion mit der Maus in einem Inspektorfenster vor.

- `list/start:length/`: Vorwärtsgeneigte Schrägstriche haben eine ähnliche Bedeutung wie das einfache Klammernpaar. Im Unterschied dazu selektiert man damit jedoch einen Bereich aus einer Liste und damit Objektmoleküle.

- `colim.b`: Die Punktnotation ist auch von anderen Sprachen her bekannt und lässt die Selektion einer Komponente eines Verbundobjektes zu. In heurisko sind die Komponenten Moleküle. Man kann auch mehrere Moleküle gleichzeitig auswählen, indem man weitere Komponenten mit Doppelpunkten wie in `colim.b:g` hinzufügt.

- Die Selektion von Molekülen und die Spezifikation von AOIs in Atomen sind in heurisko kombinierbar. So erhält man mit `aoi3d.point[0]` die x-Koordinate des Punktes.

Für jede AOI erzeugt der heurisko-Kern ein temporäres Molekül oder Atom. Alle Eigenschaften werden vom Elternmolekül oder –atom übernommen. Jedoch werden keine neuen Datenspeicher und Zeigerlisten angelegt, weil sich Eltern und AOI-Kindobjekte diese teilen. Für die weitere Verarbeitung im Kern erhalten also alle Funktionen normale Moleküle und Atome weitergereicht. Sie müssen nicht wissen, dass es sich um AOIs handelt. Nachteilig ist, dass die AOI-Erzeugung ein wenig Zeit kostet. Dieser Nachteil kann jedoch für AOIs, die mehrmals benötigt werden, durch so genannte *Alias*-Objekte umgangen werden.

```
ubyte slice = seq[10];
```

erzeugt ein Objekt, das von Bild 10 der oben definierten Sequenz eine permanente AOI darstellt. Mit Kindobjekten, die wie für AOIs aus bereits vorhandenen Objekten gebildet werden, ist es sogar möglich, die Daten eines Objekts in einem zweiten Objekt als Daten eines anderen Datentyps anzusehen:

```
float x[512][512];
fcomplex xc = x;
```

Das Objekt `xc` sieht die Daten als komplex an und hat nur noch halb so lange Zeilen wie `x`, weil je eine `float`-Zahl für die reelle und imaginäre Komponente benötigt werden. Ebenso ist es zur Bequemlichkeit möglich, durch einen senkrechten Strich mehrere Moleküle zu *einem* temporären Molekül zusammenzufassen:

```
float a, b, c, d, e, f;
a|b = Add(c|d, e|f);
```

Nach der Diskussion der heurisko-Objekte und ihrer Datenstrukturen beschäftigt sich der folgende Abschnitt mit den Operationen auf diesen Objekten.

## 4.3 Generische Operatoren

### 4.3.1 Grundprinzip

Zu jedem im Interpreter verfügbaren Operator

$obj_1, \ldots, obj_n$ = OpXYZ($obj_{n+1}, \ldots, obj_m$);

mit *n* Ausgabe- und *m-n* Eingabeobjekten enthält heurisko eine Molekülfunktion

hERR mFnXYZ(hMOLECULE* $obj_1$, …, hMOLECULE* $obj_m$).

Die Parameter des Operators `OpXYZ()` tauchen also alle in der gleichen Reihenfolge wieder auf als Funktionsparameter der Funktion `mFnXYZ()`, welche einen Fehlercode vom Typ `hERR` zurückgibt. Die Zuordnung der Funktion zum Operator geschieht über eine Attributliste, welcher der heurisko-Kern die Eigenschaften des Operators entnimmt. Da es sich bei den Molekülen um komplexe Objekte handeln kann, enthält `mFnXYZ()` meist nicht die gesamte Operatorimplementierung. Stattdessen verteilt sich die Implementierung über eine dreistufige Hierarchie von Molekül-, Atom- und Vektorfunktionen. Zur Erläuterung sei die Addition zweikanaliger Bilder betrachtet:

```
struct stereo1[480][640] {ubyte left, right};
                    # erstes Molekül mit 2 Atomen
stereo2 := stereo1; # zweites Molekül nach Muster stereo1
stereo3 := stereo1; # drittes Molekül nach Muster stereo1
…
stereo1 = Add(stereo2, stereo3);
```

Zu `Add()` gehören die Molekülfunktion `m3Add()`, die Atomfunktion `a3Add()` und die Vektorfunktion `v3ub_Add()` für den verwendeten Typ `ubyte`:

```
hERR m3Add(hMOLECULE* m1, hMOLECULE* n1, hMOLECULE* o1);

hERR a3Add(hATOM* a1, hATOM* b1, hATOM* c1);

void v3ub_Add(hUINT8* w1, const hUINT8* x1,
              const hUINT8* y1, const hCCC* ccc1);
```

Die Molekülfunktion `m3Add()` sucht die Atome der drei Moleküle auf und delegiert die Addition für jeden Satz zugehöriger Atome an die Atomfunktion `a3Add()`. Diese wiederum durchläuft die Zeilenzeigerlisten der Atome und ruft für jeden Satz korrespondierender Zeilenvektoren die Vektorfunktion `v3ub_Add()` auf. Für den gegebenen Fall mit Stereobildern wird die Atomfunktion zweimal aufgerufen, einmal für die linken Bilder, dann für die

rechten. Die Vektorfunktion wird insgesamt 840-mal durchlaufen, da linke und rechte Teilbilder je 480 Zeilen haben.

Nach dieser Vorstellung des Grundprinzips befasst sich der Abschnitt 4.3.2 mit der Interpreterebene, bevor es mit Details zur Implementierung auf der Kernebene im Abschnitt 4.3.3 weitergeht.

### 4.3.2 Operatoren im Interpreter

Operatoren seien zunächst auf der Interpreterebene betrachtet. Interpretierte Sprachen haben als Hauptziel, es dem Anwender möglichst einfach zu machen. Eine Möglichkeit ist, generische Operatoren anzubieten, deren Parameterobjekte in weiten Grenzen variabel sind und die erst zur Laufzeit bestimmen, welche internen Funktionen entsprechend den aktuellen Objekten aufzurufen sind. Dem Benutzer stellt sich auf diese Weise eine bestimmte Operation als ein einziger Operator dar, gleich, auf welchen Objekten er arbeiten soll. Als Beispiel sei die punktweise Addition betrachtet, bei der die Summe korrespondierender Punkte zweier Eingabeobjekte im korrespondierenden Punkt des Ausgabeobjektes abgelegt wird. Es liegt nahe zu fordern, dass die Objekte gleiche Gestalt haben. Denn nur dann gibt es für alle Punkte der Objekte korrespondierende Punkte in allen anderen Objekten. Weiter kann man verlangen, dass die beteiligten Objekte den gleichen Datentyp haben. Selbst mit diesen Einschränkungen bietet der Operator eine große Vielfalt, weil er auf Skalare, Vektoren, Bilder, auf Objekte beliebiger Gestalt angewendet werden kann, falls die Implementierung dies wie im Fall des heurisko-Operators `Add()` unterstützt. Für Skalare, Vektoren und Bilder demonstriert dies das folgende Skript:

```
# Addition von Objekten gleicher Gestalt und gleichen Datentyps

float x1, x2, x3;                          # Skalare
float y1[256], y2[256], y3[256];           # Zeilenvektoren
float y4[256][1], y5[256][1], y6[256][1]; # Spaltenvektoren
float z1[480][640], z2[480][640], z3[480][640]; # Bilder

x1 = Add(x2, x3); # Add für Skalare
y1 = Add(y2, y3); # Add für Zeilenvektoren
y4 = Add(y5, y6); # Add für Spaltenvektoren
z1 = Add(z2, z3); # Add für Bilder
```

Man kann nun einen Schritt weiter gehen und unterschiedliche Datentypen zulassen:

```
# Division von Objekten gleicher Gestalt,
# aber unterschiedlichen Datentyps

float x, y, z; # Gleitkommzahlen mit 32 Bit
long a, b, c;  # ganze Zahlen mit 32 Bit
y = 7.9; z = 5.0;
b = 8; c = 5;

x = Div(b, c); x; # Eingabe long, Ausgabe: float
a = Div(y, z); a; # Eingabe float, Ausgabe: float
x = Div(y, c); x; # Eingabe float und long, Ausgabe: float
a = Div(b, z); a; # Eingabe long und float, Ausgabe: long
```

Die Kommandos `x;` und `a;` sind Kurzformen für `List(x);` und `List(a);` und schreiben die aktuellen Werte der Objekte `x` bzw. `a` ins Ausgabefenster. Im Beispiel ist die Ausgabe

```
1.600000
       2
1.580000
       2
```

Die unterschiedlichen Datentypen erfordern, dass für die Ausführung eine automatische Typanpassung vorgenommen wird, wenn die Operation für eine Datentypkombination nicht implementiert ist. Dies kann so geschehen, dass die Typen der beteiligten Objekte selbst nicht verändert werden, sondern intern mit temporären Objekten gearbeitet wird. Es ist aber auch möglich, Objekttypen dynamisch anzupassen, wobei heurisko Änderungen grundsätzlich nur für Ausgabeobjekte erlaubt. Beide Methoden besitzen gewisse Nachteile. Automatische Typanpassungen haben eventuell numerische Folgen, die nicht toleriert werden dürfen, aber vom Anwender möglicherweise nicht bemerkt werden. Ein Beispiel wäre die Division zweier Ganzzahlobjekte, deren Ergebnis eigentlich in einem Objekt mit reellen Zahlen gespeichert werden soll, das aber automatisch in ein Objekt mit ganzen Zahlen konvertiert wird, um die ganzzahlige Division des Systems zu nutzen. Dem Anwender wird es also mit der automatischen Typanpassung auf der einen Seite bequem gemacht, auf der anderen Seite wird ihm aber auch die Verantwortung für die numerischen Folgen auferlegt. Temporäre Objekte dagegen vermeiden numerische Folgen, wenn sie so gewählt werden, dass mit dem genaueren Datentyp gerechnet wird. Im Beispiel würden also zwei temporäre Fließkommazahlenobjekte für die beiden ganzzahligen Eingabeobjekte erzeugt und die Division für Gleitkommazahlen ausgeführt. Nachteilig sind der zusätzliche Speicherbedarf und der Kopieraufwand. Im obigen Beispiel wurden bei der ersten Division für die Eingabeobjekte vom Typ `long` intern solche vom Typ `float` benutzt. Im zweiten Fall wurde eine Division mit `float`-Objekten durchgeführt und abschließend das Ergebnis in das Objekt vom Typ `long` kopiert. Im dritten Fall wurde intern mit einer Kopie des Objekts `c` mit Datentyp `float` gearbeitet und für die vierte Division wurde *z* in ein temporäres Objekt vom Typ `long` kopiert. Es wurde also die Methode der temporären Kopien angewandt.

Operatoren werden noch allgemeiner verwendbar, wenn man auch die Beschränkung auf Objekte gleicher Gestalt lockert. Dazu seien drei Anwendungsfälle erwähnt. Im ersten Fall sollen zwei Eingabeobjekte addiert und das Ergebnis im Ausgabeobjekt abgelegt werden. Ein Quellobjekt und das Ergebnisobjekt haben gleiche Gestalt, das zweite Quellobjekt ist ein Skalar. Dann wird der Skalar zu jedem Punkt des anderen Eingabeobjekts addiert; das Skalarobjekt wird also gedanklich zu einem Objekt derselben Gestalt wie die beiden übrigen Objekte expandiert, wobei alle Datenpunkte den gleichen Wert haben. In heurisko wird ein solches Verhalten eines Operators *expandierend* genannt. Im folgenden Skript werden nacheinander ein Skalar, ein Zeilenvektor und ein Spaltenvektor expandiert.

```
# Expandierende Addition von Objekten ungleicher Gestalt

ubyte s;              # Skalar
ubyte row[640];       # Zeilenvektor
ubyte col[480][1];    # Spaltenvektor
ubyte im1[480][640], im2[480][640]; # Bilder

im1 = Add(im2, s);    # im1[i][j] = im2[i][j] + s
im1 = Add(im2, row); # im1[i][j] = im2[i][j] + row[j]
im1 = Add(im2, col); # im1[i][j] = im2[i][j] + col[i]
```

Im zweiten Fall ungleichartiger Objekte könnte man die Schnittmenge der Objekte bezüglich korrespondierender Punkte ermitteln und nur für diese Punkte die Addition durchführen. In heurisko wird dieses Verhalten *minimierend* genannt. Es sind durchaus Fälle denkbar, wo dies sinnvoll ist. Weil dadurch aber vom Entwickler unbemerkte Fehler entstehen konnten, wurde

minimierendes Operatorverhalten zu Gunsten der expliziten Angabe von AOIs wieder abgeschafft. Einzig das Kopieren blieb als Ausnahme minimierend. Auch hier wieder ein Beispiel zur Illustration:

```
# Operationen mit Objekten ungleicher Gestalt

ubyte im1[480][640], im2[240][320];          # Bilder

# Die Addition ist nicht minimierend:
im1 = Add(im1, im2);                         # nicht erlaubt!
im1[:240][:320] = Add(im1[:240][:320], im2); # in Ordnung
# oder in Kurzschreibweise:
im1[:240][:320] = Add(im2);                  # in Ordnung

# Das Kopieren ist minimierend:
im1 = Copy(im2);                 # im1[0:240][0:320] = im2
# oder in Kurzschreibweise:
im1 = im2;                       # im1[0:240][0:320] = im2
```

Der dritte Fall ungleichartiger Objekte eines Operators macht aus der Addition zusätzlich eine Summation. Wenn nämlich die beiden Eingabeobjekte gleiche Gestalt haben, das Ausgabeobjekt aber hinsichtlich mindestens einer Dimension nur die Länge eins hat, werden alle Punkte der anderen Objekte in dieser Richtung aufsummiert. Da diese Operation in heurisko für die Addition mit dem Operator `Add()` nicht vorgesehen ist, sei stattdessen der Operator `SumProduct()` betrachtet, der die Eingabeobjekte punktweise miteinander multipliziert und im Ausgabeobjekt aufsummiert. Wenn die beiden Eingabeobjekte Bilder der Größe $n \times m$ sind und das Ausgabeobjekt ein Spaltenvektor mit $m$ Elementen, so enthält das Element $m_i$ nach der Operation die Summe aller Produkte der Zeile $m_i$ der Bilder. Analoges gilt, wenn das Ausgabeobjekt ein Zeilenvektor ist. Ist es dagegen ein Skalar, erhält dieser die Summe der Produkte aller Bildpunkte. Dieses Prinzip lässt sich auf beliebige Dimensionen anwenden und wird in heurisko *reduzierend* genannt. Entscheidend ist nur, dass das Ausgabeobjekt bezüglich einer Dimension die Länge eins hat, in der die anderen Objekte eine Ausdehnung größer als eins haben. Beispiele:

```
# Reduzierende Operationen mit Objekten ungleicher Gestalt

float mean, var;                       # Mittelwert und Varianz
float row[640];                        # Zeilenvektor
float col[480][1];                     # Spaltenvektor
float x[480][640];                     # Bild

mean = Sum.clear&mean(x);              # Mittelwert
var = SumSqrDiff.clear&mean(x, mean); # Varianz

row = Sum.clear(x);       # row[j] = Summe über i von x[i][j]
col = Sum.clear(x);       # col[i] = Summe über j von x[i][j]
```

Die Beispiele zeigen noch einen weiteren Aspekt der heurisko-Operatoren. Durch einen Punkt vom Operatornamen getrennt können Operatormodi angegeben werden, wenn ein Operator mehrere Modi kennt. Lässt ein Operator mehrere Modi gleichzeitig zu, kombiniert man diese durch ein `&` wie in `clear&mean`. Die Angabe des Standardmodus eines Operators kann unterbleiben. Da `Sum` und `SumSqrDiff` im Standardmodus die neu ermittelten Summen zu den bereits in den Ausgabeobjekten bestehenden Werten addieren, muss der Modus `clear` gewählt werden, wenn das Ausgabeobjekt vorher gelöscht werden soll. Der Modus `mean` gibt an, dass der Operator während der Summation die Anzahl der Summenelemente zählt und anschließend das Ergebnis durch diese Anzahl dividiert. Hierzu noch ein Beispiel:

```
# Mittelwert mehrerer Bilder

float mean;                          # Mittelwert
float x1[480][640], x2[480][640];    # Bilder

mean = Sum.clear(x1); # mean löschen und erstes Bild summieren
mean = Sum.mean(x2);  # zweites Bild summieren, dann Mittelwert
```

### 4.3.3 Operatoren im Kern

Nach der Interpreterebene sei nun die Implementierungsebene der Operatoren betrachtet. In der ikonischen Bildverarbeitung kann man Klassen von Operationen ausmachen, die jeweils, unabhängig vom Datentyp, nach dem gleichen Muster ablaufen. Wohl die einfachste Gruppe von Operationen sind die so genannten *Punktoperationen*, die jeden Bildpunkt losgelöst von seiner Umgebung betrachten und nur die Bildpunkte an der gleichen Position in allen beteiligten Objekten miteinander verknüpfen. Die bekanntesten Vertreter dieser Operationen sind die vier arithmetischen Grundfunktionen Addition, Subtraktion, Multiplikation und Division. Sie gehören genauer zur Klasse der *dyadischen* Punktoperationen mit zwei Operanden. Die trigonometrischen Funktionen zählen zur Klasse der *monadischen* Punktoperationen mit nur einem Operanden. Selbst von komplexeren Bildverarbeitungsoperationen als den Punktoperationen kann man Klassen mit gleichem Ablaufschema bilden, so auch die Klasse der Faltungsoperationen, zu denen viele Arten von Filtern und morphologische Operationen gehören.

Für eine Operationsklasse mit gleichem Ablaufschema kann man eine *generische* Funktion schreiben, die außer den zu verarbeitenden Objekten die auszuführende Datenoperation als Parameter hat. Man kann sich nun überlegen, was man in der gemeinsamen Funktion erledigt und auf welcher Ebene man in operationsspezifische Funktionen verzweigt. Für Punktoperationen etwa könnte man alle Prüfungen und die Schleifen durch die Objekte bis zu einem einzelnen Bildpunkt in einer generischen Funktion unterbringen. Für ein Bild mit 640×480 Bildpunkten würde dann die Funktion mit der eigentlichen, spezifischen Operation 307.200-mal aufgerufen. Da jeder Funktionsaufruf Zeit kostet, ist diese Lösung nicht günstig. Man kann sich aber als Kompromiss die im letzten Abschnitt vorgestellte Zeilenzeigerliste zunutze machen und statt Bildpunktfunktionen so genannte *Vektorfunktionen* schreiben, die auf einem eindimensionalen Feld beliebiger Länge von Bildpunkten arbeiten. Die Zahl der Funktionsaufrufe reduziert sich dadurch auf die Quadratwurzel der Anzahl Bildpunkte (bei quadratischen Bildern) oder sogar weniger (bei Bildern mit weniger Zeilen als Bildpunkten pro Zeile), im erwähnten Beispiel also auf 480. Bei dieser Lösung fällt die für die Funktionsaufrufe benötigte Zeit bei Bildern mit genügender Zeilenlänge nicht mehr ins Gewicht.

Für die bereits in Abschnitt 4.3.2 diskutierte Addition soll hier nun beschrieben werden, wie sie mit Hilfe generischer Funktionen tatsächlich implementiert ist. Die erste von der Ausführungseinheit aufgerufene Funktion ist die Molekülfunktion `m3Add()`:

```
H_IMEXPORT hERR m3Add(hMOLECULE* m1, hMOLECULE* n1,
                      hMOLECULE* o1, hUINT32 mode) {
    return mg3PopExpand(m1, n1, o1, a3PopExpand, v3Add, mode);
}
```

Wie zu sehen ist, delegiert diese Funktion alles Weitere sofort an die generische Funktion `mg3PopExpand()`, die alle expandierenden Punktoperationen mit drei Objekten behandelt. Der letzte Parameter `mode` enthält eine Kennzahl für den im Abschnitt 4.3.2 beschriebenen Operatormodus. Die Parameter `a3PopExpand` und `v3Add` bestimmen die Typen der auszuführenden generischen Funktion auf der Atom- bzw. der Vektorebene.

```
H_IMEXPORT hERR mg3PopExpand(hMOLECULE* m1, hMOLECULE* n1,
        hMOLECULE* o1, hFNERR_A3V1 afn3, hFNVEC_LIST vfn3,
        hUINT32 mode) {
    hSINT_PTR ms = mGetN(m1), ns = mGetN(n1), os = mGetN(o1);
    if (ms >= 0) {
        // Go down to next layer of submolecules
        hMOLECULE** md2 = mGetDown(m1);
        hMOLECULE** nd2 = &n1, **od2 = &o1;
        // If atom level has not been reached for molecule n1,
        // go down one level as well
        if (ns >= 0) {
            if (ns != ms) return X_ERR_MOL_NO_MATCH;
            nd2 = mGetDown(n1);
            ns = 1;
        } else {
            ns = 0;
        }
        // If atom level has not been reached for molecule o1,
        // go down one level as well
        if (os >= 0) {
            if (os != ms) return X_ERR_MOL_NO_MATCH;
            od2 = mGetDown(o1);
            os = 1;
        } else {
            os = 0;
        }
        // Loop through all submolecules
        while (ms) {
            hERR error = mg3PopExpand(*md2++, *nd2, *od2,
                                      afn3, vfn3, mode);
            if (error) return error;
            nd2 += ns; od2 += os;
            ms--;
        }
    } else {
        // Atom level has been reached: apply atomic
        // vector function
        if (ns >= 0 || os >= 0) return X_ERR_MOL_SOURCE; // Atom
            // level has not been reached for molecule n1 or o1
        return afn3((hATOM*)m1, (hATOM*)n1, (hATOM*)o1,
                                            vfn3, mode);
    }
    return NO;
}
```

Die vorstehende Funktion demonstriert, wie auf der Molekülebene parallel in den beteiligten Molekülen ein Aufsuchen aller Atome vor sich geht und für jeden Satz zugehöriger Atome die im vierten Parameter vorgegebene generische Atomfunktion `a3PopExpand()` aufgerufen wird. Ein Fehler tritt auf, wenn die Molekülstrukturen nicht zueinander passen. Es ist aber möglich, Moleküle zu expandieren:

```
# heurisko-Skript

float a/2/; # 2 Submoleküle (Liste von 2 Skalaren)
float b/3/; # 3 Submoleküle (Liste von 3 Skalaren)
float c;    # 1 Submolekül (1 Skalar)
```

```
a = Add(a, b); # Fehler: a und b passen nicht zueinander
a = Add(a, c); # OK: c wird expandiert
```

Die Funktion a3PopExpand() nimmt eine Fallunterscheidung bezüglich der Datentypen der Objekte vor. Nur für den einfachsten Fall gleicher Typen zeigt das folgende Listing den vollständigen Code der Funktion. Man erkennt, dass dem Datentyp eine Kennzahl entspricht, die als Index in das mit dem Parameter vl3pop (dessen Wert ist hier v3Add, wie das obige Listing von m3Add zeigt) übergebene Feld von Vektorfunktionen verwendet wird. Existiert für diesen Typ die Vektorfunktion nicht, wird ein Fehler zurückgegeben. Ansonsten werden ein paar Steuerparameter gesetzt, die das eventuelle Expandieren steuern, und danach die Vektorfunktionen ausgeführt.

```
H_IMEXPORT hERR a3PopExpand(hATOM* a1, hATOM* b1, hATOM* c1,
                        hFNVEC_LIST vl3pop, hUINT_PTR scl) {
    hUINT_PTR dim_a = aGetDim(a1), dim_b = aGetDim(b1),
                                   dim_c = aGetDim(c1);
    hUINT32 typa = aGetType(a1), typb = aGetType(b1),
                                 typc = aGetType(c1);
    hUINT32 id[3];
    int dst_is_src = (vl3pop.flag&H_VOP_DST_IS_SRC);

    id[0] = hID(typa); id[1] = hID(typb); id[2] = hID(typc);

    // Expanding operation requires that
    // dim_a <= dim_b and dim_c
    if (dim_b > dim_a || dim_c > dim_a) return K_ERR_DIM;

    if (typb == typa && typc == typa) {
        // All data types are equal, no type conversion required
        hFNVEC3Cp v3c;
        v3c.fn = (hFNVEC3p)vl3pop.fnl[id[0]];
        if (v3c.fn == NULL) return X_ERR_VEC_FN;
        {   // Set control structure for vector function
            hERR error =
                    aSetEqExpandCCC(&v3c.ccc, a1, b1, c1, id);
            if (error) return error;
        }
        v3c.scl = scl;      // Copy scale factor
        if (dim_a == 1) {   // Call vector function directly
                            // for vectors
            v3c.fn(aGetPtr1(a1), aGetPtr1(b1), aGetPtr1(c1),
                                               &v3c.ccc, scl);
        } else {      // Recursive execution for 2D+ objects
            hACCC ccc[H_MAX_DIM-1];
            hERR error = aSetEqExpandACCC(&ccc[0],
                        aGetStruct(a1), dim_a, aGetStruct(b1),
                        dim_b, aGetStruct(c1), dim_c);
            if (error) return error;
            a3v1Pop(aGetPtr2(a1), aGetPtr2(b1),
                    aGetPtr2(c1), ccc, &v3c);
        }
        return NO;

    } else if (typb == typa && typc != typa) {
        …
    } else if (typb != typa && typc == typa) {
        …
```

```
        } else if (typb != typa && typc == typb
          && hPrecision(id[0]) < hPrecision(id[1]) && !dst_is_src) {
            …
        } else {
            …
        }
    }
```

Angenommen, der Datentyp der Objekte, die addiert werden sollen, sei `ubyte`. Dann ist `v3ub_Add` die zugehörige Vektorfunktion. Diese wertet die zuvor in der Atomfunktion gesetzten Steuerparameter aus und führt dann die Vektoraddition durch. Für vier der fünf Fälle wird dazu in eine weitere Funktion verzweigt, welche die normale Variante in C-Code und eine Alternative in Assembler enthält.

```
H_EXPORT void v3ub_Add(hUINT8* w1, const hUINT8* x1, const
hUINT8* y1, const hCCC* ccc1) {
    hUINT_PTR n = iGetLength(ccc1);
    w1 += iGetOffset1(ccc1);
    x1 += iGetOffset2(ccc1);
    y1 += iGetOffset3(ccc1);
    switch (iGetMode(ccc1)) {
    case H_VVV: vub_adds_vvv(w1, x1, y1, n); break;
    case H_VVS: vub_adds_vvs(w1, x1, n, y1[0]); break;
    case H_VSV: vub_adds_vvs(w1, y1, n, x1[0]); break;
    case H_VSS:
        {
            hUINT_PTR t = x1[0] + y1[0];
            if (t > UCHAR_MAX) t = UCHAR_MAX;
            vb_set(w1, n, (hSINT8)t);
            break;
        }
    default:
        {
            hUINT_PTR ix = 0,
                nx = (iGetInc2(ccc1))? n : iGetElems2(ccc1);
            hUINT_PTR iy = 0,
                ny = (iGetInc3(ccc1))? n : iGetElems3(ccc1);
            do {
                hUINT_PTR t = x1[ix] + y1[iy];
                if (t > UCHAR_MAX) t = UCHAR_MAX;
                *w1++ = (hUINT8)t;
                if (++ix == nx) ix = 0;
                if (++iy == ny) iy = 0;
            } while (--n);
        }
    }
    return;
}
```

Alle generischen Funktionen sind im Kernmodul von heurisko untergebracht, während die Vektorfunktionen von separaten Modulen bereitgestellt werden.

Das Vektorfunktionskonzept hat übrigens noch andere Vorteile als den erwähnten zur Reduktion der Funktionsaufrufe. Zum einen ist die Wahrscheinlichkeit groß, dass alle bei einem Funktionsaufruf beteiligten Vektoren in den Cache des Prozessors passen. Man optimiert so die Speicherzugriffe und erhöht die Leistungsfähigkeit des Prozessors (siehe 5.1 und 5.2 für weitere Erläuterungen). Zum anderen lassen sich Vektorfunktionen vielfältig wieder verwen-

den. So wird die Addition unter anderem auch für Faltungsoperationen benötigt und Kopierfunktionen werden noch viel häufiger benutzt. Es handelt sich also um ein schönes Beispiel für die Wiederverwendung von Code. Dies führt zum vierten Vorteil der Vektorfunktionen: Es lohnt sich, Vektorfunktionen von Hand zu optimieren. Insbesondere kann man die modernen Multimedia-Instruktionssätze zur Parallelverarbeitung ausnutzen, um eine erhebliche Beschleunigung der Bildverarbeitung zu erreichen (siehe 5.2).

Eine nach den Punktoperatoren weitere große Klasse von Operatoren, für die generische Funktionen sinnvoll sind, stellen die Nachbarschaftsoperatoren dar, die im nächsten Abschnitt behandelt werden.

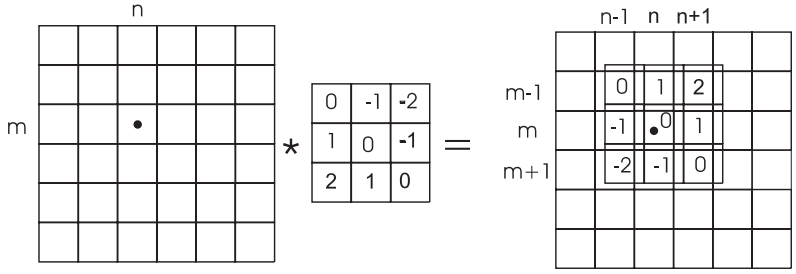


***Abbildung 4.3:*** *Faltung eines Bildes mit einer Maske der Größe 3×3*

## 4.3.4 Nachbarschaftsoperationen

Im Folgenden sei beispielhaft die diskrete Faltung eines Bildes mit einer Faltungsmaske dargestellt. Die Bedeutung dieser Operation liegt unter anderem darin, dass viele Filteroperationen Faltungsoperationen sind. Sehr ausführlich wird die Faltung in [Jähne 2004] und [Jähne 2002] behandelt. Der Einfachheit halber sei hier von einer Maske ungerader Größe ausgegangen. Wird der Grauwert eines Bildpunktes in Zeile $m$ und Spalte $n$ durch $g_{mn}$ bezeichnet, ein Element der Faltungsmaske entsprechend mit $h_{mn}$ und habe die Maske $(2r+1)(2r+1)$ Elemente, so ergibt sich der Grauwert $g'_{mn}$ nach der Faltung gemäß

$$g'_{mn} = \sum_{m'=-r}^{r} \sum_{n'=-r}^{r} g_{m+m',n+n'} h_{-m',-n'} .$$

Für die Faltung eines Bildes mit einer Maske der Größe 3×3 ($r$=1) ist dies in Abbildung 4.3 gezeigt. Man kann sich das Vorgehen so vorstellen, dass man die Faltungsmaske derart über das Bild legt, dass sich der gerade betrachtete Bildpunkt und das Zentrum der (ungeraden) Maske decken und die Maskenelemente über denjenigen Bildpunkten liegen, mit denen sie zu multiplizieren sind. Nach den Berechnungen für einen Bildpunkt wird die Maske zum nächsten Bildpunkt geschoben. Pro Bildpunkt sind $(2r+1)(2r+1)$ Multiplikationen und $(2r+1)(2r+1)$-1 Additionen auszuführen. Soll das Ausgabebild im gleichen Objekt wie das Eingabebild gespeichert werden, muss man dafür sorgen, dass man die Originalgrauwerte nicht überschreibt, solange sie noch für die Berechnungen bei Nachbarbildpunkten gebraucht werden. Die einfachste Lösung wäre, das Eingabebild in einem temporären Objekt aufzuheben. Für große Objekte wäre dies jedoch wegen des zusätzlichen Speicherbedarfs ungeschickt, möglicherweise sogar nicht akzeptabel. Deshalb soll hier ein anderer Weg aufgezeigt werden.

Betrachtet man die Faltung einer Bildzeile (man stelle sich das oben beschriebene Verschieben der Faltungsmaske entlang der Zeile vor), so fällt auf, dass alle für die Berechnungen benötigten Bildpunkte in den gleichen $2r+1$ Zeilen liegen. Des Weiteren erkennt man, dass beim Übergang zur nächsten Bildzeile nur eine neue Bildzeile hinzukommt, während eine

nicht mehr benötigt wird. Diese Beobachtungen führen zu folgendem Verfahren, das in Abbildung 4.4 für eine Maske der Größe 5×5 illustriert ist: Für die Faltung der Zeile *m* sorgt man dafür, dass sich Kopien der Zeilen *m*-*r* bis *m*+*r* in einem Puffer für 2*r*+1 Zeilen befinden. Punkt für Punkt wird dann die Faltung mit Hilfe der Zeilenpuffer berechnet und die Originalzeile im Bildpuffer mit dem Ergebnis überschrieben. Für die Faltung der Zeile *m*+1 wird die Zeile *m*-*r* im Zeilenpuffer nicht mehr benötigt. In deren Puffer kann stattdessen die nun benötigte Zeile *m*+*r*+1 kopiert werden. Führt man die Zeilenadressierung im Puffer modulo 2*r*+1 aus, haben jeweils die nicht mehr und die neu gebrauchte Zeile die gleiche Pufferadresse. Natürlich hätte man statt der Zeilenpuffer auch einen Puffer von lediglich der Maskengröße bereitstellen und nach den Berechnungen für einen Bildpunkt die dem linken und oberen Maskenrand entsprechenden Bildpunkte mit den nächsten am rechten und unteren Maskenrand überschreiben können. Aber das vektorisierte Verfahren hat bezüglich der Speicherzugriffe und Funktionsaufrufe deutliche Vorteile.

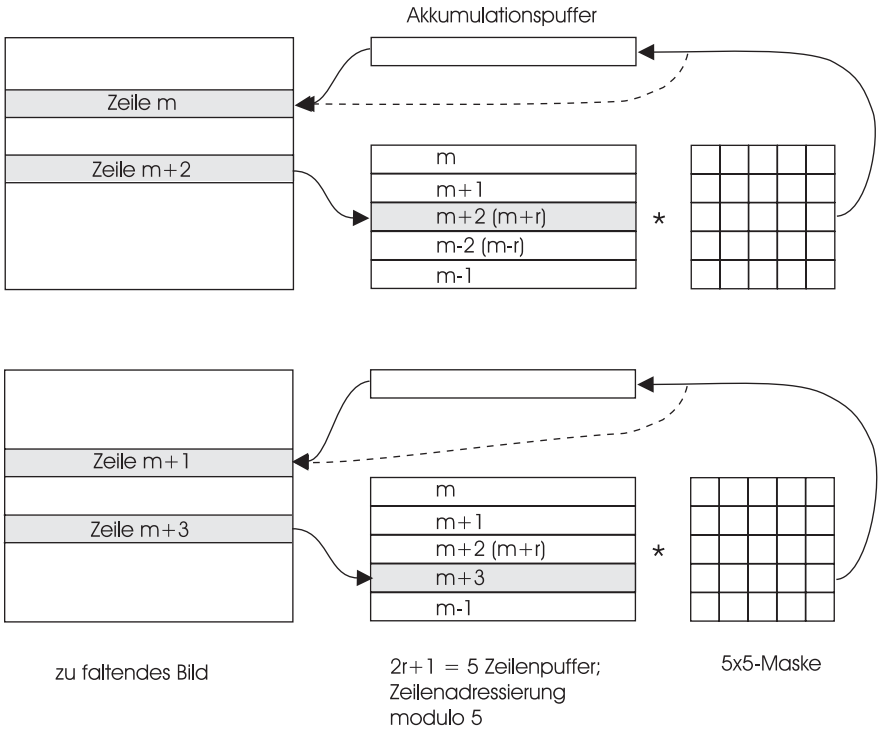


***Abbildung 4.4:*** *Schematische Darstellung der Faltung eines Bildes mit einer Maske der Größe 5×5 unter Verwendung von fünf zusätzlichen Zeilenpuffern und eines Akkumulationspuffers*

Das Vektorverfahren lässt sich noch weiter optimieren. Sieht man einen zusätzlichen Akkumulationspuffer von der Länge einer Bildzeile vor, kann man die Faltung Koeffizient für Koeffizient vornehmen. Man multipliziert also eine gesamte Bildzeile mit dem zugehörigen Maskenkoeffizienten und addiert das Ergebnis zum Inhalt des Akkumulators. Wenn man so vorgeht, lohnt sich auch für jeden Koeffizienten die Abfrage, ob er 0 oder 1 ist. Im ersten Fall kann die Zeile komplett übersprungen werden, im zweiten Fall kann die Multiplikation entfallen; die Zeile wird nur zum Akkumulator addiert.

Besondere Vorkehrungen sind an den Bildrändern nötig. Dies ist jedoch kein spezielles Problem des geschilderten Verfahrens, sondern tritt bei Faltungen immer auf und soll deshalb hier nicht behandelt werden.

Die Faltung ist eine lineare Operation. Folglich gilt für sie das Superpositionsprinzip. Dies kann man ausnutzen, wenn sich eine mehrdimensionale Faltungsmaske in mehrere eindimensionale separieren lässt. Als Beispiel sei die Glättung eines Bildes mit der 5×5-Binomialmaske

$$\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix}$$

betrachtet, die sich wie angegeben in zwei identische eindimensionale Masken separieren lässt, eine für eine horizontale Faltung, die andere für eine vertikale. Die Ausführung der zwei 1-D-Faltungen benötigt für jeden Bildpunkt insgesamt $2(2r+1)$ Multiplikationen und $4r$ Additionen. Im Fall der 5×5-Maske sind das 10 Multiplikationen und 8 Additionen gegenüber 25 Multiplikationen und 24 Additionen, die man für die 2-D-Faltung benötigen würde. Die Separierung spart also Rechenoperationen ein. Für den Fall der Identität von Ein- und Ausgabeobjekt kann ein gegenüber dem 2-D-Fall vereinfachtes Verfahren, das in Abbildung 4.5 dargestellt ist, angewandt werden. Da die Masken jeweils nur noch eindimensional sind, reduziert sich die Zahl der zusätzlichen Zeilen- bzw. Spaltenpuffer auf eins.

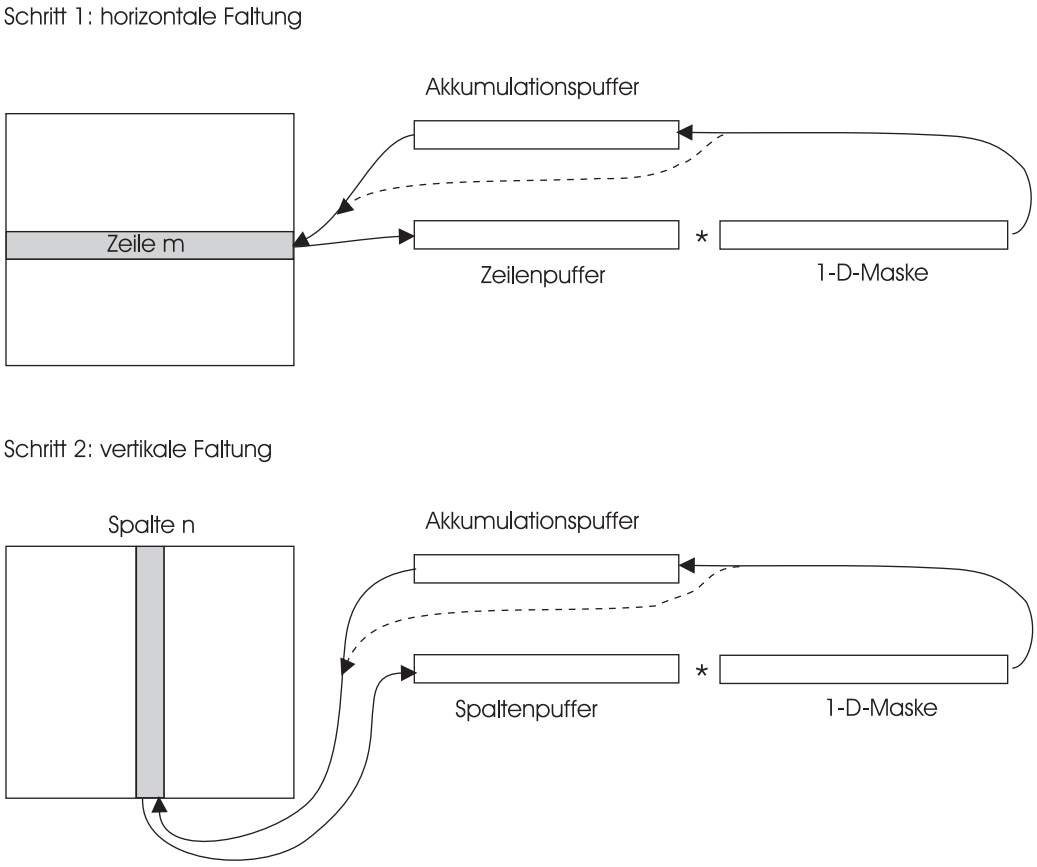


***Abbildung 4.5:*** *Schematische Darstellung der separierten Faltung eines Bildes mit einem Zeilenpuffer für die horizontale und einem Spaltenpuffer für die vertikale Faltung und mit einem Akkumulationspuffer*

Durch eine Kombination der Ideen gelangt man zu einem dritten Verfahren, welches die Speicherzugriffe der separierten Faltung reduziert. Die horizontale Faltung wird nicht für das gesamte Bild auf einmal ausgeführt, sondern immer nur für eine Zeile. Dann folgt die vertikale Faltung für diese Zeile. Der Vorteil ist, dass die Daten nur einmal in den Cache geladen werden müssen und dort so lang bleiben, bis die Faltungen in beiden Richtungen durchgeführt wurden. Die Prinzipien der separierten Faltung lassen sich einfach auf beliebige Dimensionen erweitern.

***Tabelle 4-2:*** *Verarbeitungszeiten in ms für verschiedene Implementierungen in C der Faltung eines Bildes mit der 3×3-Binomialmaske. Über jeder Spalte mit den gemessenen Zeiten ist die Anzahl der Objekte angegeben. Bei 1 sind Ein- und Ausgabeobjekt identisch, bei 2 unterschiedlich.* `Bin2N()`*: nicht separierte Faltung;* `Bin2S()`*: separierte Faltung;* `Bin2()`*: separierte Faltung mit weiteren Optimierungen (siehe Text).*

| Bildgröße | Datentyp | `Bin2N()` | | `Bin2S()` | | `Bin2()` | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 1 | 2 | 1 | 2 |
| 512×512 | `float` | 3,0 | 3,3 | 6,5 | 6,5 | 2,1 | 2,3 |
| | `short` | 2,1 | 2,0 | 2,6 | 2,7 | 1,7 | 1,6 |
| | `byte` | 10 | 8,0 | 2,6 | 2,6 | 4,4 | 3,7 |
| 512×1024 | `float` | 6,1 | 6,4 | 13 | 13 | 4,3 | 4,1 |
| | `short` | 5,0 | 4,0 | 8,7 | 9,0 | 3,0 | 3,2 |
| | `byte` | 16 | 16 | 5,6 | 5,8 | 7,1 | 7,6 |
| 1024×1024 | `float` | 12 | 13 | 28 | 27 | 8,0 | 8,2 |
| | `short` | 7,8 | 7,9 | 18 | 18 | 6,0 | 6,2 |
| | `byte` | 41 | 32 | 14 | 14 | 18 | 15 |
| 2048×1024 | `float` | 23 | 26 | 58 | 55 | 19 | 17 |
| | `short` | 16 | 16 | 37 | 38 | 12 | 11 |
| | `byte` | 63 | 63 | 31 | 28 | 29 | 29 |
| 2048×2048 | `float` | 48 | 52 | 115 | 114 | 37 | 35 |
| | `short` | 31 | 31 | 75 | 72 | 25 | 23 |
| | `byte` | 128 | 128 | 65 | 57 | 57 | 57 |

In heurisko ist die Glättung mit der 3×3-Binomialmaske

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

für die drei vorgestellten Verfahren implementiert. Der Operator `Bin2N()` führt die nicht-separierte Faltung durch, im Operator `Bin2S()` ist die separierte Faltung realisiert und in `Bin2()` werden sowohl die Separierung zur Reduktion der Rechenoperationen als auch die Optimierung der Speicherzugriffe genutzt. In Tabelle 4-2 sind die auf dem Testrechner gemessenen Verarbeitungszeiten dieser drei in C implementierten Operatoren für verschiedene Bildgrößen und Datentypen zusammengestellt. Dabei wurde jeder Operator einmal mit identischen Ein- und Ausgabeobjekten und ein zweites Mal mit unterschiedlichen Objekten ausgeführt. Die Messergebnisse zeigen, dass man nicht leicht vorhersagen kann, welcher Operator im aktuellen Fall die kürzeste Laufzeit haben wird. Außerdem bedeutet die geringere Zahl von Rechenoperationen nicht zwangsläufig auch die kürzere Ausführungsdauer. Deutlich wird dies beispielsweise beim Vergleich der Ergebnisse der Operatoren `Bin2N()` und `Bin2S()` für den Datentyp `float`. Eigentlich würde man für die separierte Faltung mit `Bin2S()` wegen der geringeren Anzahl von Rechenoperationen eine kürzere Rechenzeit erwarten. Offensichtlich machen jedoch die doppelten Speicherzugriffe für die horizontale und vertikale Faltung diesen Vorteil wieder zunichte.

## 4.4 Erweiterungsmodule

In Abschnitt 4.1 ist ein Blockdiagramm für heurisko angegeben. Dort kann man sehen, dass die Bildverarbeitungsfunktionalität in Form von Molekül-, Atom- und Vektorfunktionen in eigenen Blöcken außerhalb des Zentralmoduls untergebracht ist. Grob betrachtet gibt es zwei Arten von solchen Modulen, die in Windows als DLLs und in anderen Betriebssystemen als Shared Libraries vorliegen. Die eine Modulart enthält die nach Datentyp geordneten, allgemein verwendbaren Vektorfunktionen, die mit heurisko ausgeliefert und vom Kern bei Programmstart automatisch gesucht und eingebunden werden. Die andere Modulart enthält die Implementierung für jeweils einen bestimmten Satz von Operatoren. Für solche optionalen Module muss dem Kern über einen Importbefehl explizit mitgeteilt werden, dass er sie einbinden soll. Da ein optionales Modul die Funktionalität heuriskos erweitert, wird es *Erweiterungsmodul* genannt. Die Schnittstelle zwischen dem Kern und Erweiterungsmodulen ist offen gelegt und kann von jedem Anwender für eigene Erweiterungen benutzt werden. Im Folgenden soll gezeigt werden, wie ein Erweiterungsmodul und heurisko zusammen arbeiten. Es handelt sich um das Modul `hk_ft`, welches in heurisko die auf FFTW basierende Operatorgruppe zur Fouriertransformation bereitstellt.

Um etwas über die von einem Erweiterungsmodul angebotenen Operatoren zu erfahren, ruft der Kern nach dem Laden eines Moduls dessen Funktion `hmInit()` auf.

```
H_EXPORT hERR hmInit(hOPMODULE* hm1, hUINT32 id) {
    …
    // Register module parameters
    {
        hm1->mopgr1 = FTModule;
        …
        hm1->close = hmClose;
        hm1->copyright = NULL;
        hm1->modtype = H_MOD_EXTENSION; // Mark as general
                                        // extension module
        // Version parameters
        hm1->version = H_VERSION;
        hm1->release = H_RELEASE;
        …
    }

    // Import wisdom collected in previous sessions
    {
        FILE* wisdom_file;
        wisdom_file = fopen("fftw.wis", "r");
        if (wisdom_file) {
            fftw_status state =
                ftw_import_wisdom_from_file(wisdom_file);
            fclose(wisdom_file);
        }
    }
    return NO;
};
```

In dieser Funktion wird über das Setzen der Komponenten der beim Aufruf übergebenen Struktur vom Typ `hOPMODULE` ein Teil der Information über das Modul an den Kern übergeben. Die wichtigste Information steckt in der Komponente `mopgr1`, die ein Zeiger auf eine Struktur des Typs `hMOPGR` mit einer Liste der vom Modul angebotenen Operatorgruppen

darstellt. Eine andere interessante Komponente ist `close`, der Zeiger auf eine Funktion, die heurisko beim Freigeben des Moduls aufruft und Gelegenheit zu Aufräumarbeiten gibt. Nach dem Füllen der Struktur `hm1` hat das Modul in `hmInit()` noch die Möglichkeit zu optionalen Initialisierungen.

Die Liste mit den im Modul `hk_ft` implementierten Operatorgruppen ist kurz:

```
H_EXPORT hMOPGR FTModule[] = {
    {"Fourier transform",   mopFT},
    {NULL,                  NULL}
};
```

Die einzige Gruppe heißt „Fourier transform“. Deren Operatoren sind in einer weiteren Liste mit Einträgen vom Typ `hMOP` beschrieben; der Zeiger darauf ist `mopFT`. Einen Ausschnitt dieser Liste zeigt das folgende Listing:

```
hMOP mopFT[] = {

//    NAME          CLASS         IMPLEM. TYPES  PROPS   VECFN
//                    OPERATOR LIST

    {"ftSetWindow", H_OP_WINDOW0, 0x01, 0, 0,
                     {m0ftSetWindow}},

    {"ftLearn",     H_OP_FT,      0x03, 0, 0,
                     {m1ftLearn, m2ftLearn}},

    {"ftiLearn",    H_OP_FTI,     0x03, 0, 0,
                     {m1ftiLearn, m2ftiLearn}},

    {"ft",          H_OP_FT,      0x03, 0, 0,
                     {m1ft, m2ft}},

    {"fti",         H_OP_FTI,     0x03, 0, 0,
                     {m1fti, m2fti}},

    …

    {NULL,          H_OP_NO_CLASS,0x00, 0, 0,      {NULL}}
};
```

Jeder Eintrag in der Liste beschreibt einen Operator. Die erste Komponente gibt an, mit welchem Namen der Operator in einem heurisko-Skript aufrufbar sein soll. Die zweite Komponente gibt die Klasse an, zu welcher der Operator gehört. Obwohl es bei dem gewählten Beispiel durchweg etwas kompliziertere Klassen sind, steht dort meistens eine Kennung dafür, wie viele Ausgabe- und Eingabeobjekte der Operator hat. So könnte der Operator `ft()` die Kennung `H_OP_OUT1_IN01` haben, die besagt, dass der Operator ein Ausgabeobjekt hat und dass es zwei Varianten gibt, wovon die eine kein Eingabeobjekt (Ziffer 0) und die andere ein Eingabeobjekt (Ziffer 1) hat. Der Operator kann demnach in der ersten Form

```
xc = ft(); # Ein- und Ausgabeobjekt identisch
```

oder in der zweiten Form

```
xc = ft(x); # Ein- und Ausgabeobjekt verschieden
```

aufgerufen werden. In der tatsächlichen Kennung `H_OP_FT` steckt zusätzlich die Information, welche Modi für `ft()` gewählt werden können, wobei die Modi in diesem Fall zur Richtungsauswahl dienen. Die nächste Komponente der Operatorbeschreibung gibt bit-codiert an, welche durch die Kennung eigentlich vorgesehenen Operatorvarianten auch tatsächlich implementiert sind. Für `ft()` gibt es zwei Varianten, die beide implementiert sind, so dass die Bits 0 und 1 gesetzt sind und der Wert `0x3` in die Liste eingetragen ist. Es folgen zwei Kennzahlen, die hier nicht erklärt werden sollen. Die letzte Komponente schließlich enthält die Zeiger auf die Molekülfunktionen aller implementierten Operatoren. Für die zweite Variante von `ft()` sei hier die Funktion `m2ft()` mit Verweis auf die Erläuterungen zu den generischen Molekülfunktionen in Abschnitt 4.3.3 wiedergegeben:

```
H_EXPORT hERR m2ft(hMOLECULE* m1, hMOLECULE* n1, hBITS32 dir) {
    hUINT_PTR i = 0;
    // Preoder traversal of molecule
    mStack2[i++] = m1; // Push root molecules on stack
    mStack2[i++] = n1;
    do {
        hMOLECULE *t1 = mStack2[--i], *s1 = mStack2[--i];
        hSINT_PTR n = mGetN(s1);
        if (mGetN(t1) != n) return X_ERR_MOL_NO_MATCH;
        if (n >= 0) {
            hMOLECULE **m2 = mGetDown(s1) + n;
            hMOLECULE **n2 = mGetDown(t1) + n;
            if (i + n + n > H_MAX_MOL_STACK)
                              return X_ERR_MOL_STACK;
            while (n) {
                mStack2[i++] = *--m2;   // Push all sub
                    // molecules on stack in reverse order
                mStack2[i++] = *--n2;
                n--;
            }
        } else {
            hERR error = a2ft((hATOM*)s1, (hATOM*)t1,
                                    FT.window, dir);
            if (error) return error;
        }
    } while (i);
    return NO;
}
```

Die Funktion enthält die für Molekülfunktionen übliche Traversierung der Moleküle und den Aufruf der Atomfunktion `a2ft()`.

Neben den eben beschriebenen allgemeinen Erweiterungsmodulen kennt heurisko drei spezielle Arten von Erweiterungsmodulen. In Abschnitt 4.5.1 wird ein Modul für das Arbeiten mit Dateien und in Abschnitt 4.5.2 ein zweites für Datenerfassung und Kommunikation vorgestellt. Abschnitt 4.5.3 diskutiert das dritte Modul, welches zur Inspektion und Visualisierung von Objekten dient.

## 4.5 Spezielle Erweiterungsmodule

### 4.5.1 Dateien

Kein Bildverarbeitungssystem kommt ohne *Dateneingabe* und meistens auch nicht ohne *Datenausgabe* aus. Eine Möglichkeit des Datenaustauschs bilden Dateien. Ein Problem entsteht durch die vielen unterschiedlichen Dateiformate, denn ein Bildverarbeitungssystem wird wegen des Aufwands niemals alle Formate direkt unterstützen können. Deshalb kommt es darauf an, durch Möglichkeiten zur Anpassung eine gute indirekte Unterstützung zu bieten.

Für das Lesen und Schreiben von Dateien enthält heurisko einen Satz von formatneutralen Operatoren (siehe Tabelle 4-3), deren formatspezifische Low-Level-Implementierung in separaten Modulen untergebracht ist. Für jede Dateioperation muss ein Format spezifiziert werden. Dies kann implizit über die Dateinamenerweiterung oder explizit über die Angabe eines Operatormodus geschehen. Jedem Format ist eine eindeutige Kennung zugeordnet. Ist diese `xyz`, erwartet heurisko die zugehörigen Low-Level-Funktionen im Modul `io_xyz`. Die Schnittstelle zu den I/O-Modulen unterscheidet sich von der eines Erweiterungsmoduls, wie es in Abschnitt 4.4 beschrieben wurde. Details sollen hier jedoch nicht erörtert werden. Die unterstützten Dateiformate zeigt Tabelle 4-4. Weitere Formate können leicht durch Erstellung neuer `io_*`-Module hinzugefügt werden. Eine andere Möglichkeit, mit Hilfe der Low-Level-Dateioperatoren ein beliebiges Format zu benutzen, demonstriert folgendes Skript:

```
# Lesen eines Volumenbildes aus einer Datei
# zur Kürzung des Beispiels ohne Auswertung des Headers

ubyte vol[166][242][214];
ubyte bheader[44];
long handle;
handle = FileOpenRead.raw("volumetric/mr_head/head_t1.dat");
bheader = FileRead(handle);
vol = FileRead(handle);
FileClose(handle);
```

***Tabelle 4-3:*** *Die wichtigsten heurisko-Operatoren zum Lesen und Schreiben von Dateien*

| Operator | Beschreibung |
|---|---|
| | High-Level |
| `Read, Write` | Lesen und Schreiben einer Datei |
| `ReadSeq, WriteSeq` | Lesen und Schreiben einer Bildsequenz, eine Datei je Bild mit automatischer Dateinummerierung |
| | Low-Level |
| `FileOpenRead, FielOpenWrite` | Öffnen einer Datei |
| `FileClose` | Schließen einer Datei |
| `FileRead, FileWrite` | Lesen und Schreiben in geöffneter Datei |
| `FileGetPosition, FileSetPosition, FileIncPosition` | Dateizeigeroperationen |
| `FileGetSize` | Dateigröße |

***Tabelle 4-4:*** *Die wichtigsten in heurisko unterstützten Dateiformate*

| Format | Modul | Beschreibung |
|---|---|---|
| ASCII | `io_asc` | vom Menschen lesbare Form der Daten, beschränkt geeignet zum Datenaustausch mit anderen Programmen, speicherplatzintensiv, für alle heurisko-Objekte |
| RAW | `io_raw` | speicherplatzsparendes Rohformat, sehr beschränkt geeignet zum Datenaustausch mit anderen Programmen, für alle heurisko-Objekte |
| TIFF | `io_tiff` | TIFF-Format, sehr gut geeignet zum Datenaustausch mit anderen Programmen, nur Bilder und Bildsequenzen |
| JPEG | `io_jpeg` | JPEG-Format, sehr gut geeignet zum Datenaustausch mit anderen Programmen, nur 8-Bit-Bilder |
| BMP | `io_bmp` | Bitmap-Format, sehr gut geeignet zum Datenaustausch mit anderen Programmen, in heurisko nur 8-Bit-Bilder |

### 4.5.2 Datenerfassung und Kommunikation

Stammen die Eingabedaten nicht aus Dateien, sondern von Bilderfassungs-, Mess- oder sonstigen speziellen Geräten, spricht man von *Datenerfassung*. Ist es darüber hinaus auch noch möglich, über Ausgabekanäle Daten wieder nach außen zu geben, kann man von *Kommunikation* sprechen. Während die Vielfalt beim Datenaustausch über Dateien von den unterschiedlichen Dateiformaten herrührt, ist es bei der Datenerfassung und Kommunikation die Verschiedenartigkeit der Geräte. Ein Bildverarbeitungssystem wird niemals alle Geräte und Schnittstellen direkt unterstützen können. Deshalb kommt es auch hier darauf an, durch Möglichkeiten zur Anpassung eine gute indirekte Unterstützung zu bieten. Im Folgenden ist öfter von *Datenblock*-Akquisition anstatt von *Bild*-Akquisition die Rede. Damit wird der Tatsache Rechnung getragen, dass nicht nur *Bild*erfassungsgeräte unterstützt werden.

***Tabelle 4-5:*** *Operatoren des heurisko-Moduls* `hk_ac` *für Datenerfassung und Kommunikation. In der letzten Spalte ist angegeben, welcher Operator gerätespezifisch implementiert werden muss.*

| Operator | Beschreibung | spezifisch? |
|---|---|---|
| `acReset` | Neuinitialisierung eines Gerätes | nein |
| `acInit` | Neuinitialisierung der Datenblockzähler | nein |
| `ac` | einmalige Akquisition | ja |
| `acStart` | Start einer kontinuierlichen Akquisition im Hintergrund | ja |
| `acStop` | Ende einer kontinuierlichen Akquisition im Hintergrund | ja |
| `acCnt` | Anzahl der erfassten Datenblöcke in der laufenden Akquisition | nein |
| `acPosition` | Index des aktuell erfassten Datenblocks | nein |
| `acTest` | Status der letzten Akquisition | nein |
| `acWait` | Warten, bis eine bestimmte Anzahl von Datenblöcken erfasst ist | ja |
| `acDisplayStart` | Live-Bild in einem zugeordneten Inspektorfenster einschalten | nein |
| `acDisplayStop` | Live-Bild in einem zugeordneten Inspektorfenster ausschalten | nein |

heurisko verfügt mit `hk_ac` über ein Erweiterungsmodul mit einer allgemeinen Schnittstelle für Datenerfassungs- und Kommunikationsgeräte. Dieses Modul ist ein Standarderweite-

rungsmodul und erfordert für jedes konkrete Gerät ein weiteres, spezielles Erweiterungsmodul. Die Operatoren von `hk_ac` sind teilweise abstrakt, da die eigentliche Implementierung dieser Operatoren im gerätespezifischen Erweiterungsmodul geschieht, wobei nicht die gesamte Funktionalität zur Verfügung gestellt werden muss. Während der Funktionsumfang der bereits mit heurisko ausgelieferten speziellen Erweiterungsmodule umfangreicher ist, weil die konkrete Anwendung noch nicht bekannt ist und ein weiter Anwendungsbereich abgedeckt sein soll, kann sich der Anwender, der ein eigenes Modul entwickelt, auf das für seinen Fall Erforderliche beschränken.

Eine besondere Unterstützung für die Datenerfassung gibt es in heurisko nicht nur auf der Operatorseite, sondern auch auf der Objektseite. Und zwar wird ein Gerät in heurisko durch ein Objekt von einer speziellen Art repräsentiert. Während sich allgemeine Objekte vom Anwender innerhalb gewisser Grenzen frei definieren lassen, wird ein Geräteobjekt automatisch mit Hilfe von Information, die heurisko aus dem zugehörigen Erweiterungsmodul bezieht, erzeugt. Vor weiteren Erläuterungen soll ein Beispiel gegeben werden.

```
device cam, type xyz, config cfgstr, buffers buf;
```

Reservierte Worte sind aufrecht gedruckt, Variable dagegen schräg. Das Kommando `device` bewirkt, dass ein Objekt erzeugt wird, welches ein Gerät repräsentiert. Das Objekt erhält den Namen *cam*. Mit dem Schlüsselwort `type` wird der Parameter *xyz* übergeben, der das Gerät spezifiziert. Dabei ist *xyz* Bestandteil des Dateinamens des Erweiterungsmoduls; der vollständige Modulname muss nach der Konvention für heurisko-Erweiterungsmodule `acq_`*xyz* sein. Der erste Teil `acq` des Namens kennzeichnet das Modul als eines, das mit `hk_ac` zusammenarbeitet. Der zweite Teil des Namens kennzeichnet das Gerät selbst. Heurisko lädt das gewünschte Modul und führt eine Konfigurationsroutine aus. Die Konfiguration kann durch den mit dem Schlüsselwort `config` übergebenen String *cfgstr* gesteuert werden. Viele Geräte werden mit einem komfortablen Konfigurationsprogramm geliefert und erlauben das Speichern von Konfigurationen in Dateien. In diesen Fällen ist es praktisch, wenn das Erweiterungsmodul im Konfigurationsparameter den Namen einer Konfigurationsdatei erwartet. Der mit dem Kennwort `buffer` übergebene Parameter *buf* schließlich ist ein Objekt mit vorgegebener Struktur, welches den Speicher für die zu akquirierenden Daten beschreibt. Das Objekt *buf* könnte wie folgt definiert sein:

```
# Objekt zur Beschreibung zweier Puffer

struct buf/2/{
    string name,  # Name für späteren Zugriff
    long dim,     # Dimension
    long size[3], # Größe, hier für bis zu 3-D
    long offs[3]; # Offset, hier für bis zu 3-D
    string format # Format für Ausgabe
};

buf/0/ = ”buf1”.{2}.{-1,-1}.{-1,-1}.”%4i”;
buf/1/ = ”buf2”.{3}.{-1,-1,100}.{-1,-1,0}.”%4i”;
```

Das mit dem `device`-Befehl erzeugte Objekt *cam* könnte dann so aussehen:

```
struct cam
    string type,
    float start,
    float end,
    ulong eventpos[4],
    string event[4],
```

```
        long inoffset[2],
        long insize[2],
        long inbits[2],
        long binning,
        long mode,
        long trigger,
        long timeout,
        long gain,
        float integrate,
        float tempsensor,
        ushort buf1[480][640],
        ushort buf2[100][480][640]
    };
```

Es würde hier zu weit führen, sämtliche Einzelheiten zu erläutern. Alle Komponenten bis auf die beiden letzten stammen aus einer obligatorischen, aber flexiblen Struktur im Erweiterungsmodul und bestimmen, welche Eigenschaften des Gerätes in einem Skript abgefragt oder von einem Skript gesetzt werden können. Die letzten beiden Komponenten des Geräteobjekts werden durch das mit dem Schlüsselwort `buffer` übergebene Objekt spezifiziert. Im obigen Beispiel werden gleich zwei Puffer für die akquirierten Daten vorgesehen. Der erste Puffer mit dem Namen `buf1` ist für Einzelbilder gedacht, der zweite, `buf2`, für Sequenzen mit 100 Bildern. Mit der Wertzuweisung von -1 für die Komponenten `size` und `offs` bestimmt man, dass für die Bildgrößen und die Position des Bildfensters die Standardwerte des Gerätes oder die durch eine Konfigurationsdatei gesetzten Werte übernommen werden sollen.

Einer der Akquisitionsoperatoren heißt `ac` (siehe Tabelle 4-5) und bewirkt ein einmaliges Füllen des angegebenen Puffers. So würde mit `ac(cam.buf1)` ein Einzelbild und mit `ac(cam.buf2)` eine Sequenz mit 100 Bildern aufgenommen. Über die Abfrage der Komponente `tempsensor` könnte man vorher erfahren, ob die Betriebstemperatur der Kamera im zulässigen Bereich ist. Und mit dem Setzen der Komponente `integrate` könnte man die Belichtungszeit der Kamera einstellen. Die Abfrage einer Komponente bewirkt den Aufruf einer Lesefunktion und das Setzen einer Komponente den Aufruf einer Setzfunktion des Gerätetreibers. Was mit einem Erweiterungsmodul möglich ist, wird also nicht alleine durch die angebotenen Operatoren bestimmt, sondern auch durch das Objekt, welches das Gerät repräsentiert. Obwohl die Signaturen der Operatoren in der abstrakten Schnittstelle festgelegt sind, belässt ihre Implementierung dem Entwickler weitgehende Freiheiten für die Semantik. Darüber hinaus kann der Entwickler die Schnittstelle zum Gerät über das Geräteobjekt beliebig ausgestalten und eine operatorlose Zusatzfunktionalität schaffen.

In der wissenschaftlichen und industriellen Bildverarbeitung ist häufig eine Online-Datenverarbeitung nötig. Im Gegensatz zu manch anderem Online-System hat man es in der Bildverarbeitung im Allgemeinen mit einer vergleichsweise hohen Datenrate zu tun, die sich aus der räumlichen Auflösung der Bilder, der Speichertiefe je Bildpunkt und aus der Bildrate zusammensetzt. Für die direkte (Online-) Datenverarbeitung muss in einem Bildverarbeitungssystem für Synchronisation gesorgt werden. Hinzu kommt, dass Datenerfassung und -verarbeitung eventuell parallel geschehen müssen. Während der Verarbeitung eines im Speicher liegenden Bildes wird in einem solchen Fall bereits das nächste Bild erfasst. Aber selbst bei indirekten (Offline-) Systemen kann die Datenerfassung eine Herausforderung sein. Ein typisches Beispiel sind Bildsequenzen, die auf Grund der Datenfülle nicht mehr im Hauptspeicher gehalten werden können, sondern Schritt haltend mit der Datenankunft auf einem Massenspeicher abgelegt werden müssen. Zur Illustration sei ein kleines Zahlenbeispiel gegeben. Der Sensor einer monochromen Kamera habe eine räumliche Auflösung von 640×480, die Bildaufnahmefrequenz betrage 100 Hz und pro Bildpunkt werden 8 Bit ausgegeben. Jedes Bild belegt dann 300 kB Speicher und der Datenstrom beträgt knapp 30 MB/s. Solange die Daten

im Hauptspeicher abgelegt werden können, ist das für einen modernen PC, dessen PCI-Bus eine Datentransferrate von bis zu 132 MB/s unterstützt (siehe 2.3.1), kein Problem. Nimmt man an, dass im Hauptspeicher für die Bilddaten 1 GB reserviert sind, lassen sich Sequenzen von etwas 35 s Länge im RAM aufnehmen. Bei längeren Sequenzen müssen die Daten parallel zur Akquisition aus dem Hauptspeicher auf einen Massenspeicher verschoben werden. Ohne besondere Vorkehrungen bezüglich der Hardware ist jedoch die geforderte Datenrate von 30 MB/s nicht garantiert.

Die abstrakte Schnittstelle für die Datenakquisition beinhaltet Möglichkeiten zur Synchronisation und Echtzeitverarbeitung. Viele Geräte benötigen für eine kontinuierliche Datenerfassung nur einen Anstoß und erledigen dann den Datentransfer ohne die CPU per DMA. Zum Starten der Akquisition existiert eine Funktion, die sofort zurückkehrt, ohne auf das Ende zu warten. Die CPU kann so während der Datenerfassung für andere Zwecke eingesetzt werden, z. B. zur Verarbeitung bereits im Hauptspeicher befindlicher Bilder oder zum Transfer der Daten auf einen Massenspeicher. Zur Synchronisation startet das gerätespezifische Modul einen zweiten Thread, in dem entweder ein Zähler des Gerätes abgefragt oder auf eine Meldung des Gerätes gewartet werden kann. Der zweite Thread ermöglicht das Warten, ohne die CPU zu blockieren. Darüber hinaus bietet heurisko den synchronen Warteoperator `acWait()`, der erst zurückkehrt, wenn der gewünschte Zählerstand erreicht oder die spezifizierte Anzahl von Datenblöcken erfasst worden ist.

```
# Akquisition einer Sequenz
# buf2 sei ein Puffer für 100 Bilder

i = 0;
ac(cam.buf2); # Akquisition starten ohne zu warten
repeat(100);
    acWait.position(cam.buf1,i);
    …        # Bild i verarbeiten, z. B. speichern
    i = Inc();
endrepeat;
```

Im Beispiel wird die Akquisition von der durch das Objekt `cam` repräsentierten Kamera mit `ac()` in Gang gesetzt. Die Objektkomponente `buf1` ist der für die Kamerabilder reservierte Speicher; in diesem Fall soll es Speicher für eine Bildsequenz mit 100 Bildern sein. In einer Schleife wird jeweils auf das Ende der Akquisition eines Bildes gewartet; dann wird das Bild verarbeitet. Anschließend wird der Bildzähler für den nächsten Schleifendurchgang inkrementiert. Im nächsten Beispiel beginnt mit `acStart()` eine kontinuierliche, zyklische Akquisition in `buf3`. Dieses Mal habe der Speicher Raum für zwei Bilder. Wenn das zweite Bild im Speicher ist, wird das dritte Bild wieder über das erste Speicherbild geschrieben, danach das vierte über das zweite Speicherbild usw. Dies geschieht so lange, bis die Akquisition mit `acStop()` beendet wird. In einer Schleife wird jeweils auf das Ende der Akquisition eines Bildes gewartet; dann wird das Bild verarbeitet. Anschließend wird der Bildindex für das nächste Bild gesetzt.

```
# Zyklische Akquisition in Ringspeicher
# buf3 sei ein Puffer für zwei Bilder

loop = 1; i = 0;
acStart(cam.buf3); # kontinuierliche Akquisition starten
while(loop);
    acWait.position(cam.buf3,i);
    …             # Bild i verarbeiten, z. B. speichern
    if (i); i = Dec();
    else; i = Inc(); endif;
```

```
endwhile;
acStop(cam.buf3);
```

Die beiden Beispielskripte weisen ein Manko auf, denn es gibt keine Kontrolle darüber, ob während der Verarbeitung eines Bildes mehr Zeit verbraucht wird als die Erfassung des nächsten Bildes benötigt. Die Folge im ersten Beispiel wäre, dass im nächsten Schleifendurchlauf mit `acWait()` auf ein Ereignis gewartet wird, das schon vorbei ist und niemals mehr eintreten wird. Im zweiten Beispiel würde zwar so lange gewartet, bis das Ereignis auf Grund der zyklischen Akquisition wieder eintritt, aber zwischendurch würden Bilder unbemerkt nicht verarbeitet. Abhilfe schafft hier der Operator `acCnt()`, der zu jeder Zeit die Anzahl der bis dahin akquirierten Datenblöcke zurückgibt. Hier ist noch einmal das zweite Beispiel, dieses Mal aber ergänzt durch eine Überwachung auf Vollständigkeit der bearbeiteten Bilder:

```
# Zyklische Akquisition in Ringspeicher mit Überwachung
# buf3 sei ein Puffer für zwei Bilder

short cnt, counter;
counter = 1;       # Sollzählerstand
loop = 1; i = 0;
acStart(cam.buf3); # kontinuierliche Akquisition starten
while(loop);
    acWait.position(cam.buf3,i);
    cnt = acCnt(cam.buf3);
    if(cnt > counter);
        Pause.error("Fehler: Bild ausgelassen.");
        return;
    endif;
    …              # Bild i verarbeiten, z. B. speichern
    if (i); i = Dec();      # nächstes Bild ist Bild 0
    else; i = Inc(); endif; # nächstes Bild ist Bild 1
    counter = Inc();
endwhile;
acStop(cam.buf3);
```

Das Akquisitionsmodul ist nicht auf Bildquellen beschränkt. Denkt man sich statt einer Kamera irgendein Daten lieferndes Gerät, ist es in den Code-Beispielen unerheblich, welcher Art die Daten sind. Obwohl bisher nur von Datenerfassung die Rede war, lässt sich das Akquisitionsmodul ebenso für die Ausgabe von Daten verwenden. So wurden für die abstrakte Datenerfassungsschnittstelle von heurisko unter anderem Erweiterungsmodule für Spektrometer, für die serielle und parallele Schnittstelle eines PC und für Karten mit analogen und digitalen Eingängen erstellt.

### 4.5.3 Visualisierung

Wie kann ein Entwickler einen besseren Zugang zu seiner Aufgabe bekommen als durch ein Bild seiner Daten? Und wie kann man einem Anwender in vielen Fällen das Ergebnis einer Bildverarbeitung klarer präsentieren als wieder durch ein Bild? Das schon einmal am Anfang zitierte Sprichwort „Ein Bild sagt mehr als tausend Worte“ trifft eben auch hier zu. Sowohl auf der Entwicklerseite als auch auf der Anwenderseite ist Bildverarbeitung im Allgemeinen ohne Darstellung von Bildern nicht denkbar. Auf Grund der Verschiedenartigkeit der Daten ist es allerdings genauer, von Visualisierung der Daten als von Bilddarstellung zu sprechen.

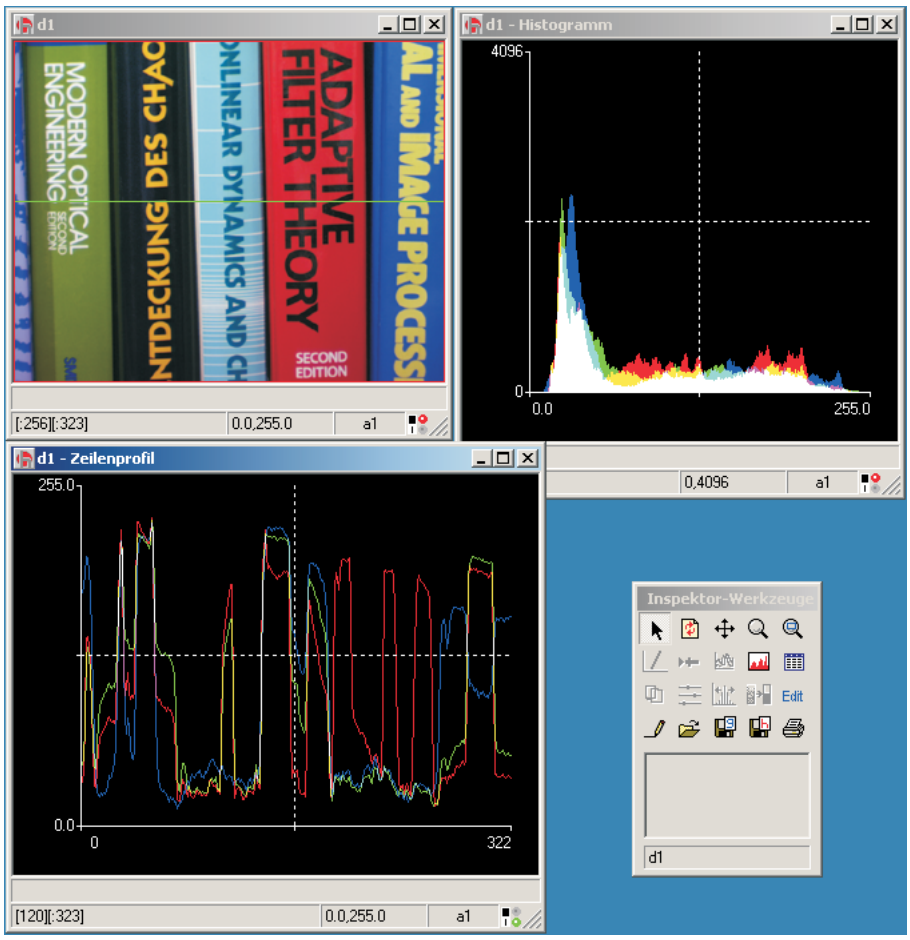


***Abbildung 4.6:*** *heurisko-Inspektor; links oben: 2-D-Inspektor mit Originalfarbbild; rechts oben: 1-D-Inspektor mit Histogramm des Originalbildes; links unten: 1-D-Inspektor mit Zeilenprofil zu mit grüner Linie markierter Zeile des Originalbildes; rechts unten: Inspektorwerkzeuge mit Aktivierung für Zeilenprofilfenster*

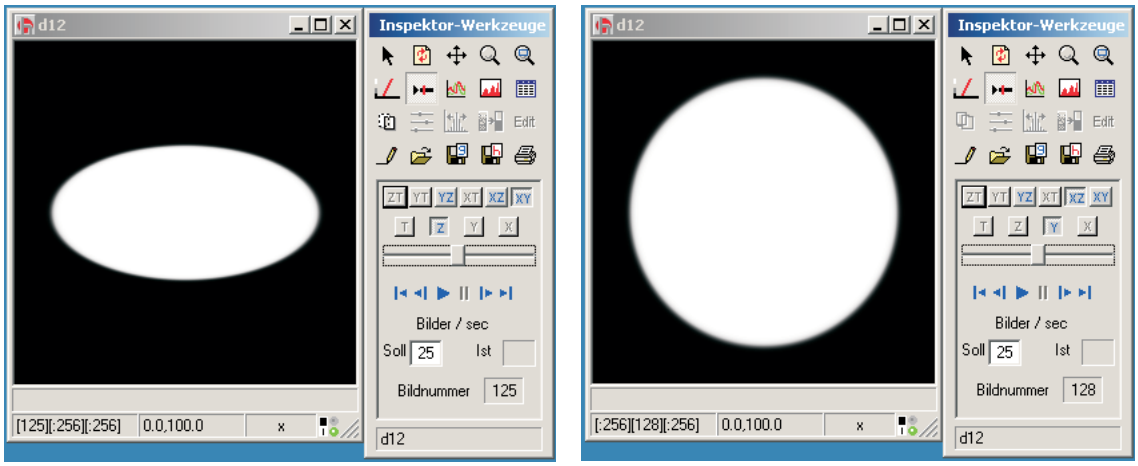


***Abbildung 4.7:*** *heurisko-Inspektor für eiförmiges 3-D-Objekt; links: 3-D-Inspektor mit xy-Scheibe aus der Mitte des Objekts und zugehörige Inspektorwerkzeuge; rechts: 3-D-Inspektor mit xz-Scheibe aus der Mitte des Objekts und zugehörige Inspektorwerkzeuge*

Bildverarbeitungsfunktionalität muss also mit Visualisierungsfunktionalität kombiniert werden. Bildverarbeitungsbibliotheken verfügen meistens nicht über Möglichkeiten zur Visualisierung. Es gibt aber Visualisierungsbibliotheken, mit denen sie zusammen eingesetzt werden können. Interpretierende Systeme, die ohnehin die Einfachheit in der Verwendung als ihr Hauptziel haben, integrieren dagegen in der Regel Möglichkeiten zur Visualisierung. Da der Schwerpunkt dieser Softwarepakete aber nicht auf der Visualisierung liegt, ist mit

Einschränkungen zu rechnen. Ein typisches Beispiel ist die Darstellung von 3-D-Daten, für die es mächtige Spezialpakete (z. B. [VTK] und [VGL]) gibt. Leider ist die Kombination verschiedener Software-Systeme auf Interpreterebene nicht möglich. Anbieter solcher Systeme müssen deshalb durch ausreichende Funktionalität dafür sorgen, dass die Akzeptanz ihrer Produkte gegeben ist.

Im Abschnitt 4.5.2 über die Datenerfassung wurde das abstrakte Erweiterungsmodul `hk_ac` beschrieben. Ein weiteres solches Modul existiert für die Visualisierung. Ganz ähnlich wie für die Datenerfassung definiert das Visualisierungsmodul `hk_iv` Operatoren, die für die graphische Darstellung von Bilddaten nützlich sind. Die konkrete Implementierung erfolgt in speziellen Modulen. Das Standardmodul heuriskos zur Inspektion und Visualisierung von Objekten ist `iv_inspector` und wird *Inspektor* genannt. Zum objektorientierten Entwurf des Inspektors gibt es eine Diplomarbeit [Gleisinger 2000]. Ebenso wie für die Datenerfassung ist mit dem Modul nicht nur ein Operatorsatz verknüpft, sondern auch ein besonderes Inspektionsobjekt, welches über die Operatoren hinaus zusätzliche Funktionalität bietet. Will man das heurisko-Objekt *`obj`* inspizieren, erzeugt man mit

```
display dname, type "inspector", obj;
```

ein assoziiertes Inspektionsobjekt, auf das man mit dem Namen `dname` Zugriff bekommt. Der Typparameter bestimmt, dass das Erweiterungsmodul `hk_iv` das Modul `iv_inspector` laden soll. Genauso gut könnte jedes andere zur Verfügung stehende Visualisierungsmodul geladen werden, z. B. das, welches MATLAB für Plots zur Hilfe nimmt. Außer dem Inspektionsobjekt wird ein Fenster zur Darstellung der Daten des inspizierten Objektes `obj` erzeugt und auf dem Bildschirm angezeigt. Die Art des Fensters richtet sich nach dem Objekt (Abbildung 4.6 und Abbildung 4.7). Für ein dreidimensionales Objekt stellt der Inspektor einen 2-D-Schnitt dar und aktiviert in der Inspektorwerkzeugleiste Elemente zur Navigation durch das Objekt. Angenommen, die Koordinaten des Objekts seien mit X, Y und Z bezeichnet. Initial stellt der Inspektor den XY-Schnitt mit Z=0 dar. Es kann aber auch auf XZ- oder YZ-Schnitte umgeschaltet werden. Mit einer Steuerung, wie man sie von Ton- oder Filmabspielgeräten kennt, kann man entlang der dritten Koordinate navigieren, wobei die Abspielgeschwindigkeit einstellbar ist. Die Besonderheit des Inspektors ist die Assoziierung mit dem inspizierten Objekt. Der Benutzer erhält automatisch eine sinnvolle Darstellung der Daten, kann aber bei Bedarf die Darstellung beeinflussen. Beispielsweise beginnt ein Inspektor mit eingeschaltetem Live-Modus, was bedeutet, dass jede Datenänderung eine automatische Auffrischung der Darstellung bewirkt. Ist dies unerwünscht, lässt sich der Live-Modus abschalten und eine gezielte Aktualisierung der Darstellung durch Aufruf des Operators `ivView()` erreichen.

Das mit obigem `display`-Befehl erzeugte Inspektionsobjekt hat folgende Gestalt:

```
struct dname {
    long objtype,
    long dispsize[2],
    float range[2],
    long rangemode,
    long winpos[2],
    long viewpos[2],
    string text,
    long zoom,
    long ipoltype,
    long live,
    long aoi[4],
    long objnum,
```

```
    long lutused,
    ubyte lutr[256],
    ubyte lutg[256],
    ubyte lutb[256]
}
```

Es würde zu weit führen, an dieser Stelle alle Komponenten dieses Objektes zu erläutern. Jedoch lassen die Namen der Komponenten schon erahnen, wozu die einzelnen Komponenten dienen. So liefert eine Abfrage von `winpos` die aktuelle Position des zugehörigen Inspektorfensters auf dem Bildschirm und das Setzen von `winpos` hat die Verschiebung des Fensters zur Folge. Es wird also zur Positionierung kein eigener Operator benötigt. Da das Inspektionsobjekt vom speziellen Inspektionsmodul selbst bestimmt wird, kann die durch den festen abstrakten Operatorsatz bereitgestellte Funktionalität beliebig ergänzt werden. Die Operatoren von `hk_iv` zeigt Tabelle 4-6.

***Abbildung 4.8:*** *Operatoren des heurisko-Moduls* `hk_iv`*.*

| Operator | Beschreibung |
|---|---|
| `ivView` | Auffrischung des Inspektorfensters inklusive Abspielen einer Sequenz mit vorgegebener Frequenz |
| `ivDraw` | Zeichnen und Schreiben im Inspektorfenster |
| `ivDrawClr` | Löschen der gezeichneten und geschriebenen Elemente |
| `ivSetFont` | Fontauswahl für das Schreiben |
| `ivSetPen` | Stiftauswahl für das Zeichnen und Schreiben |
| `ivWrite` | Speichern des Fensterinhalts in einer Datei |
| `ivPrint` | Drucken des Fensterinhalts |
| `ivShowToolbox` | Werkzeugdialog an spezifizierter Position anzeigen |
| `ivHideToolbox` | Werkzeugdialog verstecken |
| `ivSelect` | AOI-Auswahl per Maus im Inspektorfenster |

Neben `iv_inspector` wurden zwei weitere Module zu `hk_iv` erstellt. Das eine ist `iv_matlabr13` zur Nutzung der Plot-Möglichkeiten mit MATLAB (Release 13), das andere `iv_vgl` zum Rendern von Volumendaten mit der VGL-Bibliothk der deutschen Firma Volume Graphics [VGL]. Für MATLAB müssen die Daten der heurisko-Objekte umkopiert werden, da MATLAB die Daten spaltenweise und mit dem Datentyp `double` erwartet. Bei VGL, das die Möglichkeit bietet, externe Daten rendern zu lassen, muss dagegen nicht umkopiert werden. Es ist nur dafür zu sorgen, dass die heurisko-Objekte mit dem Attribut `noalign` definiert werden. Zwei kleine Skripte demonstrieren, wie einfach mit `iv_matlabr13` (Abbildung 4.9) und `iv_vgl` (Abbildung 4.10) eine Darstellung von Daten aus heurisko heraus möglich ist. Über Komponenten der speziellen Objekte, die mit dem `display`-Befehl erzeugt werden, können das Plotten und Rendern beeinflusst werden. Gezeigt ist das für die Beschriftung eines Plots und eine Materialeigenschaft des gerenderten Objekts. Im MATLAB-Fenster können weitere MATLAB-Funktionen aufgerufen werden und im VGL-Fenster ist beispielsweise Rotieren des Objektes per Maus möglich.

```
# 1. Plots von Spektren mit MATLAB

# Struktur mit 3 Spalten variabler Größe
struct spectrum[*911] {float wl, abs1, abs2};
```

```
# MATLAB-Fenster:
display m1, type "matlabR13:plot", spectrum.wl,
                                   spectrum.abs1.abs2;
m1.live = 0; # keine automatische Auffrischung

... # Lesen der Spektren

ivView(m1); # gezielte Auffrischung der Darstellung
m1.xlabel = "\lambda [nm]";
m1.ylabel = "transmittance";



# 2. Rendern von Volumendaten eines Kopfes mit VGL

noalign ubyte head1[166][242][214]; # Attribut noalign beachten!
ubyte t[242][214];

display vgl1, type "vgl", head1;
vgl1.live = 0; # kein automatisches Rendern bei Datenänderung

... # Setzen der Volumendaten

vgl1.matisosurface = 0.2;
ivView(vgl1);  # gezieltes Rendern
```

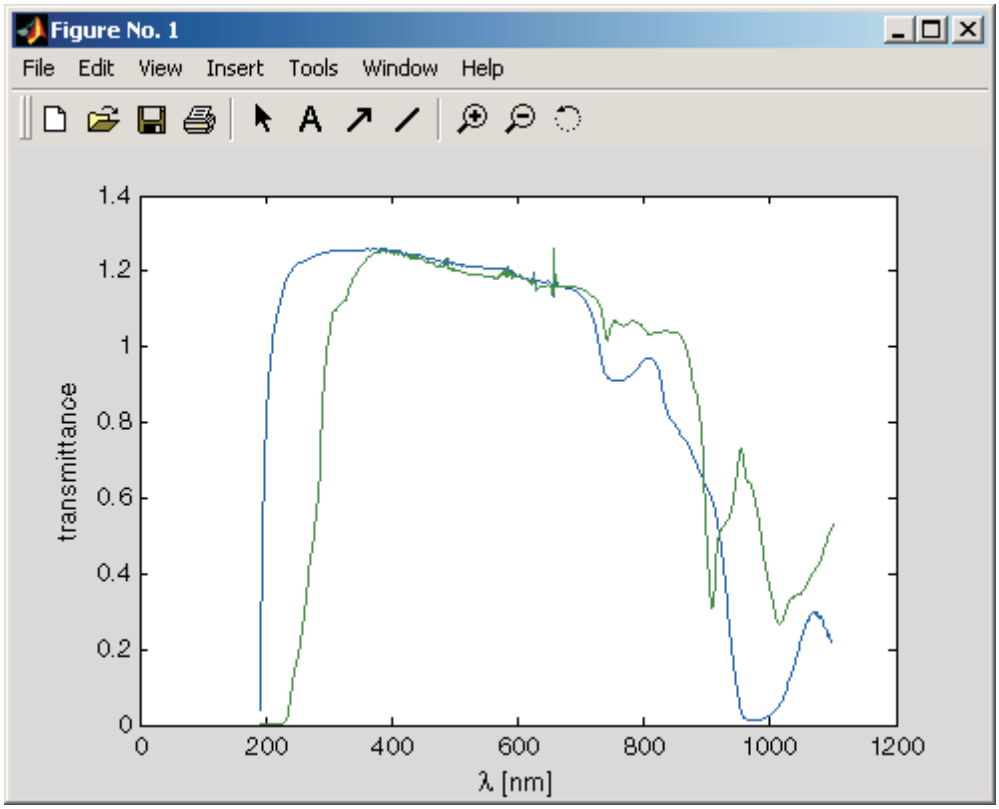


***Abbildung 4.9:*** *Plots von heurisko-Objekten mit MATLAB*

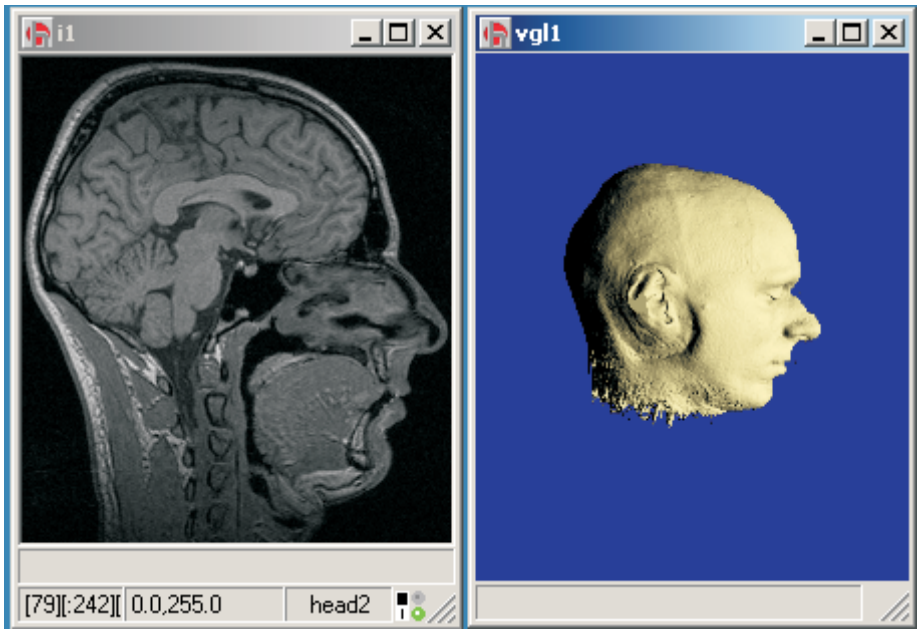


***Abbildung 4.10:*** *Links: Schnitt durch den Volumendatensatz eines menschlichen Kopfes; rechts: mit VGL gerenderte Darstellung des Kopfes*

## 4.6 Interpretieren versus Kompilieren

Wenn hier Interpretieren und Kompilieren einander gegenübergestellt werden, so sind damit im Grunde die beiden Möglichkeiten gemeint, eine Bildverarbeitungsanwendung entweder in einer 4GL- oder in einer 3GL-Sprache zu programmieren. Den ersten Fall hat man sich so vorzustellen, dass eine Anwendung als Skript in der 4GL-Programmiersprache abgelegt ist und auch so zur Ausführung kommen kann. Das Skript muss dabei zur Laufzeit von einem Interpreter in Aufrufe von Funktionen einer darunter liegenden Funktionsbibliothek umgesetzt werden. Der Benutzer eines Programms benötigt das Skript, d. h. den Quellcode, und ein separates Programm mit dem Interpreter und der Funktionsbibliothek. Im zweiten Fall muss der 3GL-Quellcode von einem Compiler aus der Hochsprache in ein Programm, das direkt gestartet und ausgeführt werden kann, übersetzt werden. Der Anwender braucht keinen Quellcode. Die Grenzen zwischen interpretierten und übersetzten Sprachen sind dabei durchaus fließend (siehe z. B. *Ousterhout's dichotomy* in [FOLDOC]). Bekannte Skriptsprachen sind unter anderem Tcl, Perl, Python, Java Script, Visual Basic, IDL und MATLAB. Eine Übersicht zum Thema Skriptsprachen findet man in [Ousterhout 1998].

Die Vorteile einer Skriptsprache sind:

- Einfachheit: Es sind keine zusätzlichen Werkzeuge wie ein Compiler notwendig. Befehle können direkt ausgeführt werden.

- Problemorientiertheit: Einige Skriptsprachen sind für bestimmte Anwendungsgebiete geschaffen und erlauben die Programmierung auf weit höherer Ebene als Standardhochsprachen.

- Plattformunabhängigkeit: Das Skript läuft unverändert auf allen Plattformen, auf denen der Interpreter angeboten wird.

Eine Skriptsprache hat jedoch auch einige Nachteile:

- Interpretierte Programme haben eine geringere Geschwindigkeit als kompilierte Programme.
- Interpretierende Software-Pakete lassen sich nicht auf Skriptsprachenebene kombinieren.
- Für Skriptsprachen stehen weniger komfortable Benutzeroberflächen und Entwicklungswerkzeuge zur Verfügung.

Es folgt eine Diskussion der einzelnen Vor- und Nachteile. Dabei wird gelegentlich abkürzend vom *Interpreter* gesprochen, obwohl eigentlich das Gesamtsystem aus Interpreter und darunter liegender Funktionsbibliothek gemeint ist.

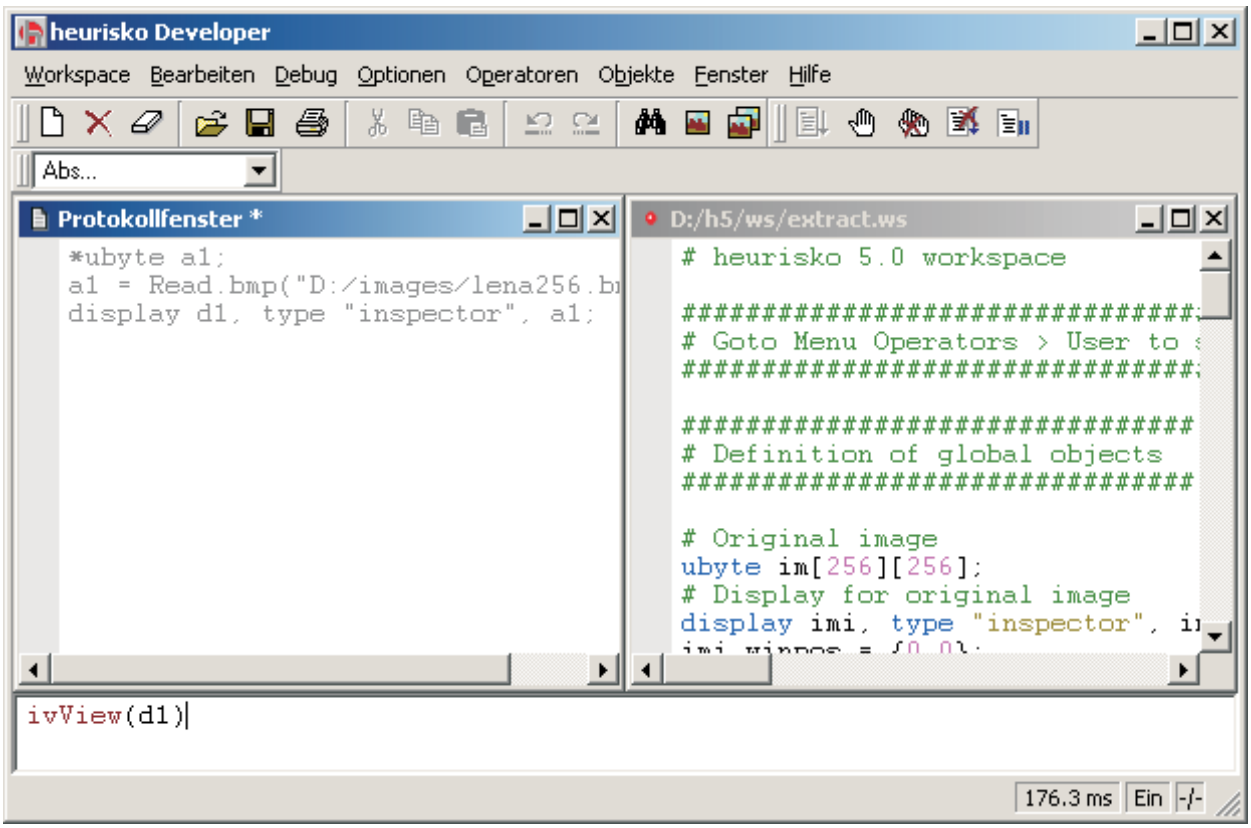


***Abbildung 4.11:*** *Hauptfenster von heurisko mit einem Editorfenster, dem Protokollfenster und darunter angeordnetem Kommandozeilenbereich*

### 4.6.1 Diskussion der Vorteile

**Einfachheit**

Das Erstellen eines interpretierten Programms lässt einen schnellen Wechsel zwischen Quellcodeerstellung und Testen zu. Man kann ein paar Zeilen schreiben und diese dann unmittelbar zur Ausführung bringen. Man muss sich dazu vor Augen halten, dass die so erstellten Programme meistens keine großen Programmsysteme sind, sondern eher kleine Einheiten mit wenigen hundert Codezeilen. Abbildung 4.11 zeigt das heurisko-Hauptfenster mit einem Editorfenster, dem Protokollfenster und dem darunter angeordneten Kommandozeilenbereich. Programme in einem Editorfenster lassen sich editieren und mit einem Befehl ausführen, wieder editieren und erneut ausführen. Außerdem können einzelne Befehle oder kurze Befehlssequenzen über die Kommandozeile eingegeben werden. Fehlerfrei ausgeführte Kommandos werden zur späteren Verwendung im Protokollfenster mitgeschrieben. Sämtliche

Befehle können natürlich auch über Menüs und Dialoge mit direkter Hilfestellung eingegeben werden. Diese Möglichkeiten bedeuten ein großes Maß an Flexibilität und Komfort beim Erstellen eines Programms.

**Problemorientiertheit**

Skriptsprachen stützen sich auf die darunter liegenden problemspezifischen Funktionsbibliotheken und haben damit komplexe Befehle zur Verfügung, für die man in einer Standardhochsprache mehrere Codezeilen bräuchte. In heurisko gibt es beispielsweise die Anweisung *scan*, mit der eine komplexe Schleife mit wenigen Zeilen geschrieben werden kann:

```
scan(im1|0, im2|0:::-1);
    im2 = im1;
endscan;
```

Diese Schleife erzeugt zeilenweise in `im2` eine auf den Kopf gestellte Kopie von `im1`. Die Zeichen hinter den senkrechten Strichen in der Parameterliste von `scan` geben an, wie das jeweilige Objekt durchlaufen werden soll. Es sei nebenbei bemerkt, dass man in heurisko für dieses Beispiel besser den vorhandenen Operator `Mirror()` verwenden würde. Eine problemorientierte Sprache verfügt über viele Funktionen ähnlich `Mirror()`, welche die Arbeit enorm erleichtern. Zudem handelt es sich bei interpretierten Sprachen zumeist um dynamisch getypte Sprachen, die den Umgang mit ihnen weiter vereinfachen.

**Plattformunabhängigkeit**

Ein ausführbares Programm ist an eine Plattform gebunden. Dagegen ist ein Skript unverändert auf jeder Plattform einsetzbar, auf der ein Interpreter zu der Skriptsprache existiert. Natürlich ist damit die Plattformabhängigkeit nur auf den Interpreter verlagert, aber entscheidend ist, dass der Anwender der Programmiersprache davon nichts merkt. Die Plattformunabhängigkeit der Anwenderprogramme ist besonders dann interessant, wenn sich der Interpreter leicht portieren lässt und der Anwender so eine hohe Sicherheit hat, dass seine Investition auch für zukünftige Plattformen nicht verloren ist. In der Architektur von heurisko sind alle Elemente der graphischen Benutzeroberfläche streng vom Rest getrennt. So kann heurisko ohne die Benutzeroberfläche in kürzester Zeit auf jedes System, für das ein ANSI/ISO-kompatibler C-Compiler angeboten wird, portiert werden. Dies wurde bereits mehrfach so praktiziert, wenn auch manche der Portierungen heute ohne Bedeutung sind. Heurisko wurde anfangs unter NextStep auf einem Next Computer entwickelt und unter MS-DOS mit einer 32-Bit-Erweiterung und dem Koprozessor Intel i860 eingesetzt. Kurz darauf erfolgte die Ergänzung einer graphischen Benutzeroberfläche unter MS Windows 3.1, dann die Portierung auf die 32-Bit-Betriebssystem MS Windows 95, MS Windows NT und Linux. Zwischenzeitlich wurde heurisko jeweils ohne graphische Benutzeroberfläche auch auf Macintosh-Systemen, auf IBMs OS/2, auf OS/9-Hardware der Firma ELTEC und auf UNIX-Systemen (IBM, HP, SGI) demonstriert. Überdies erfolgte die Portierung auf die IA64-Architektur unter MS Windows 2000. Heute basiert die Benutzeroberfläche von heurisko auf der plattformunabhängigen C++-Bibliothek Qt der norwegischen Firma Trolltech [QT], so dass auch dieser Teil heuriskos auf vielen wichtigen Plattformen problemlos übersetzt werden kann.

Die Distribution eines Programms einer 4GL-Sprache in Form eines Skriptes hat den Nachteil, dass jeder das Programm lesen kann und dass damit möglicherweise ungewollt Know-how preisgegeben wird. In heurisko gibt es dazu als Abhilfe optional die Möglichkeit, das Skript vor der Auslieferung über einen Dongle sicher zu verschlüsseln. Das Programm wird dann automatisch beim Einladen in den Interpreter für den Anwender unsichtbar wieder entschlüsselt. Der Zeitverlust in der Größenordnung einer Sekunde spielt in der zugehörigen Programmphase keine Rolle.

### 4.6.2 Diskussion der Nachteile

**Geschwindigkeit**

Interpretieren kostet Zeit. Deshalb werden Skripte von einigen Interpretersystemen in einen anderen Code übersetzt, der dann sehr schnell interpretiert werden kann. Diese Methode wird beispielsweise seit der Version 8.0 bei dem Skriptsprachentandem Tcl und Tk angewendet [Tkl/Tk]. Andere Pakete wie HALCON verfügen neben dem Interpreter auch über einen Compiler, der ein Skript in eine Standardhochsprache übersetzt. Hier sind die Vorteile beider Methoden miteinander verknüpft: eine bequeme Programmentwicklung in der Skriptsprache und eine hohe Ausführungsgeschwindigkeit der endgültigen, kompilierten Anwendung.

In heurisko erzeugt der Interpreter einen binären Zwischencode für die Ablaufsteuerung im Kern. Die Basis der Code-Erzeugung bildet eine Reihe von Feldern, wobei der Index im Feld die Kennzahl (ID) des dort eingetragenen Elements bildet. Die Elemente des ersten Feldes entsprechen Operatorgruppen und enthalten Zeiger auf weitere Felder, deren Einträge Zeiger auf die Molekülfunktionen der Operatoren sind. Objekte werden nach ihrer Art unterschieden. Es gibt globale und lokale Objekte, Aliasobjekte, spezielle Objekte zur Datenerfassung und Visualisierung und andere Objektarten. Zu jeder Art existiert ein Feld mit Zeigern auf die Moleküle der Objekte. Das erste Byte des Befehlscodes kennzeichnet den Typ der durchzuführenden Anweisung, denn die Deklaration eines neuen Objektes etwa, der Beginn einer Programmschleife oder der Aufruf eines Operators müssen in der Ablaufsteuerung des Kerns unterschiedlich behandelt werden. Entsprechend unterscheidet sich auch der Binärcode. Handelt es sich um den Aufruf eines Operators, enthalten die folgenden Code-Bytes Kennzahlen für die Operatorgruppe und den Operator, eine Typkennzeichnung des Operators (die hier nicht weiter erläutert werden soll), den Operatormodus, die Anzahl der Parameterobjekte insgesamt und die Anzahl der Ausgabeobjekte. Dann folgen je Parameterobjekt der Anweisung zwei Bytes mit der erwähnten Kennzahl und mit zusätzlicher Information. Man erkennt, dass die Codeerzeugung und die schnelle Decodierung im Kern wesentlich damit zusammenhängen, dass der Zugriff auf Objekte und Operatoren effizient über ihre Kennzahlen erfolgt.

Übergibt man an den Interpreter interaktiv Kommandos, die sofort ausgeführt werden sollen, ergibt sich aus dem Zwischencode kein Geschwindigkeitsvorteil. (Allerdings spielt bei interaktiven Eingaben, die vergleichsweise immer langsam sind, die Zeit zum Interpretieren sowieso keine Rolle.) Für den automatischen Betrieb arbeitet man jedoch in der Regel mit mehr oder weniger großen Skripten, in denen Funktionen definiert werden, die in heurisko *benutzerdefinierte* oder *selbstdefinierte* Operatoren heißen. Außer diesen Operatoren enthält ein Skript Objektdeklarationen und andere sofort ausführbare Anweisungen. Ein Skript wird zunächst in seiner Gesamtheit dem Interpreter übergeben. Nachdem dieser daraus den internen Code erzeugt hat, ruft man im weiteren Verlauf jeweils nur noch einen der benutzerdefinierten Operatoren über den Interpreter auf. Gegenüber der Laufzeit der dadurch angestoßenen komplexen Operation kann die Zeit zum Interpretieren des Aufrufs vernachlässigt werden. Hier ist ein simples Beispiel eines heurisko-Skriptes:

```
// globale Objekte:
ubyte in[640][480];
float out[640][480];

// selbst definierte Operatoren:
operator init();
    string file;
    file = RequestFiles();
    if (!file); return; endif;
    long ok;
```

```
    in, ok = Read(file;
    if (!ok); Pause.error("Fehler beim Einlesen"); endif;
endoperator;

operator go();
    out = Bin2(in); // Glätten
    out = D1_3();   // Differenzieren
endoperator;

// Initialisierungen:
init();
```

Bei der Übergabe des Skriptes an den Interpreter werden die globalen Objekte sofort angelegt, danach die Codes für die Operatoren erzeugt and anschließend gleich der Operator `init()` ausgeführt. Der Hauptoperator `go()` steht nun zum Abruf bereit. Aber auch `init()` könnte zu jedem Zeitpunkt wieder ausgeführt werden.

Tabelle 4-6 zeigt den Einfluss des Interpreters auf die Ausführungszeiten ausgewählter Operationen mit je einem Eingangs- und einem Ausgangsobjekt. `Bin2()` ist eine Faltungsoperation mit einer 3×3-Maske, die vergleichsweise schnell ausgeführt werden kann. `Tan()` ist die Tangensfunktion, die auf Grund ihrer Transzendenz sehr viel mehr Zeit benötigt und überdies nur auf Fließkommadaten arbeitet. `Nop()` ist eine spezielle Funktion zur Messung des Zeitbedarfs für einen Funktionsaufruf in heurisko. Beim Aufruf dieser Funktion läuft vor und nach der eigentlichen Operation alles genauso ab wie bei den anderen beiden Funktionen, aber in `Nop()` selbst geschieht nichts; die Funktion kehrt sofort zurück. Die Messungen der ersten Zeile in der Tabelle erfolgten durch Eingabe der Anweisung über den Interpreter. Man erkennt, dass 1550 µs für den Aufruf der eigentlich interessierenden Funktion vergeudet werden. Im Falle der Faltung mit `Bin2()` ist das ein Anteil von etwa 60 %, bei der komplexeren Tangensfunktion aber von nur noch knapp 6 %. Die Zeiten in der zweiten Zeile wurden beim Aufruf eines kompilierten Unterprogramms gemessen. Hier beträgt die Zeit für die interne Ablaufsteuerung noch 21 µs. Dies sind ca. 2,5 % der Ausführungszeit für die Faltung und weniger als 1 ‰ beim Tangens. Bei größeren Datenfeldern, wie sie sich auch bei multidimensionalen Daten ergeben, reduziert sich der unerwünschte Zeitanteil weiter.

***Tabelle 4-6:*** *Einfluss des Interpreters auf die Laufzeit*

| | Ausführungszeiten in µs | | |
|---|---|---|---|
| | `y = Nop(x)` | `y = Bin2(x)` | `b = Tan(a)` |
| Interpretiert | 1550 | 2500 | 27000 |
| Kompiliert | 21 | 850 | 25000 |

Nop: leere Operation
Bin2: Binomialfilter mit Maske der Größe 3x3
Tan: Tangensfunktion
`x, y:` Bilder der Größe 640x480, 8-Bit-Zahlen (`ubyte`)
`a, b:` Bilder der Größe 640x480, 64-Bit-Fließkommazahlen (`double`)

Das führt auf einen weiteren Aspekt, warum ein interpretiertes Programm langsamer als ein kompiliertes Programm ist. Wie die Rechenzeit für `Nop`() in der zweiten Zeile der Tabelle zeigt, ist selbst der vom Interpreter erzeugte Binärcode noch vergleichsweise langsam, wenn die aufgerufene Funktion einfach ist. Das liegt daran, dass sich Operatoren auf einer hohen abstrakten Ebene befinden. Ebenso würde man mit einem kompilierten Programm eine große Laufzeit erhalten, wenn man z. B. für jeden Bildpunkt eine Bibliotheksfunktion benutzen würde, die eigentlich für vollständige Bilder gedacht ist. Damit wird deutlich, dass man die

Anzahl von einfachen Operationen, bei denen der Zeitaufwand für den Aufruf groß gegenüber dem der eigentlichen Operation ist, gering halten muss. Eine Programmschleife über alle Punkte eines Bildes ist durch Ersetzung der Pixeloperation durch eine Bildoperation nach Möglichkeit zu vermeiden. Der Entwickler muss hier mitdenken. Wie bereits erwähnt, setzt MATLAB zur Abhilfe den JIT-Accelerator ein (siehe Abschnitt 3.2.1). In heurisko arbeiten so gut wie alle Operatoren auf allen Objekten, unabhängig von ihrer Komplexität. Programmschleifen sind deshalb selten. Wenn sie aber einmal unumgänglich sind, bietet der schon zitierte `scan`-Operator eine gewisse Entschärfung des Problems. Im folgenden Beispiel benötigt die erste Schleife mit `repeat()` 7,7 s und die zweite mit `scan()` 4,3 s. Man beachte, dass bei Verwendung von `scan()` die Indizierung implizit vorhanden ist und deshalb die Notation mit den Klammern `[]` wegfällt.

```
# Schleifen über alle Bildpunkte

nx = 480; ny = 640;

# mit repeat:
m = Clr();
repeat(ny);
    n = Clr();
    repeat(nx);
        y[m][n] = Nop(x[m][n]);
        n = Inc();
    endrepeat;
    m = Inc();
endrepeat;

# mit scan:
scan(x|0, y|0);
    scan(x|0, y|0);
        y = Nop(x);
    endscan;
endscan;
```

**Kombinierbarkeit:**

Kein Software-Paket deckt alle Wünsche ab. Es wird deshalb immer Anwender geben, die gezwungen sind, entweder auf eigentlich gewünschte Funktionalität zu verzichten oder aber verschiedene Pakete miteinander zu kombinieren. Da Interpretersprachen proprietär sind, lassen sich nur Komponenten in der gleichen Sprache gut kombinieren. Bei unterschiedlicher Sprache lassen sie sich nicht gemeinsam auf der Skriptsprachenebene einsetzen. Wenn man auf einen kombinierten Einsatz nicht verzichten möchte, sind zwei Auswege denkbar. Beim ersten entscheidet man sich für eine Interpretersprache als die Hauptsprache und integriert das andere Paket auf der Ebene der Implementierungssprache. Beim zweiten Ausweg schafft sich der Anwender eine übergeordnete Anwendung, die beide Interpretersprachen zugänglich macht. Dies setzt voraus, dass die Software-Systeme entsprechende Schnittstellen aufweisen.

Diese Schnittstelle erlaubt es beliebigen Programmen, sich an Stelle der Originalbenutzeroberfläche an den Interpreter anzudocken. So wird mit heurisko ein Austauschmodul geliefert, das heurisko zu einer simplen Konsolenapplikation macht. Die Interpreterschnittstelle kommt im Wesentlichen mit drei Funktionen aus, mit einer zur Initialisierung, einer zweiten zur Kommandoübergabe und einer dritten zum Aufräumen vor Beendigung. Ein einfaches C-Steuerprogramm, das heurisko nutzt, könnte so aussehen:

```
char cmd[128];
hERR error;

error = hiInit(…);
if (error) {…}

strcpy(cmd, "WsRun xyz.ws");
error = hiDo(cmd, T_INSTRUCT_CL);
if (error) {…}

strcpy(cmd, "…");
error = hiDo(cmd, T_INSTRUCT_CL);
if (error) {…}

error = hiClose();
if (error) {…}
```

Etwas aufwendiger wird es, wenn Daten, auf denen heurisko arbeiten soll, vom Steuerprogramm kommen, oder wenn das Steuerprogramm mit den Daten von heurisko weiterarbeiten will.

**Benutzeroberflächen und Entwicklungswerkzeuge**

Für Standardprogrammiersprachen steht eine Vielzahl von ausgereiften kommerziellen und freien Werkzeugen zur Verfügung: Für Skriptsprachen existieren sie teilweise auch, insbesondere für solche mit weiter Verbreitung. Bei Produkten mit geringerer Verbreitung fehlen solche Werkzeuge aber oder sind nicht so komfortabel, wie man es sonst gewohnt ist.

Editoren für Programmiersprachen bieten häufig folgende Möglichkeiten:

- Farbliche Markierung: Dies ist das wohl wichtigste Hilfsmittel zur Sicherstellung der syntaktischen Korrektheit und auch zur Übersichtlichkeit. So werden unter anderem Kommentare, Schlüsselworte, Strings und Zahlen in speziellen Farben dargestellt.

- automatischer Texteinzug: Viele Editoren bieten einen automatischen Einzug für Programmzeilen. Dabei richten sie sich nach der bei vielen Programmierern üblichen Praxis. Der automatische Einzug ist eher ein zusätzlicher Komfort als ein wichtiges Hilfsmittel.

- Klammernpaartest: Beim Schreiben eines Programms kommt es häufig vor, dass man Klammern vergisst. Beispielsweise funktioniert der Test derart, dass man sich bei der Positionierung der Einfügemarke direkt neben einer öffnenden oder direkt neben einer schließenden Klammer die korrespondierende Klammer anzeigen lassen kann.

- spezielle Textunterlegung: Beim Editieren von Text werden häufig die Funktionen *Kopieren*, *Einfügen*, *Ersetzen* und *Löschen* verwendet. Elegant ist es, wenn man dazu Text z. B. mit einem Mausdoppelklick markieren kann. Programmiersprachen lassen jedoch in Worten andere Zeichen zu als gesprochene Sprachen, z. B. den Unterstrich. Deshalb ist ein an die Programmiersprache angepasster Editor praktisch. Ein weiterer Komfort ist, wenn man ein beliebiges Rechteck aus Zeilen und Spalten markieren und bearbeiten kann.

Damit wird solch ein Editor von der unterstützen Sprache abhängig. Das bedeutet weiter, dass die Wahrscheinlichkeit, einen komfortablen Editor zu erhalten, für wenig verbreitete Sprachen geringer als für weit verbreitete Sprachen ist.

Das Hauptfenster der graphischen Benutzeroberfläche von heurisko ist in Abbildung 4.11 zu sehen. Die Flexibilität durch den jederzeit möglichen Wechsel zwischen Editieren und Ausführen eines Skriptes, Eingaben in der Kommandozeile und Arbeiten mit den Menüs und Dialogen wurde in diesem Kapitel bereits erwähnt. Der Editor basiert auf einem Standardtexteditor der Qt-Bibliothek, der um die farbliche Markierung der Syntax ergänzt wurde. Weitere programmiersprachenspezifische Fähigkeiten wie einen Klammerpaartest oder einen automatischen Texteinzug bietet der Editor zurzeit nicht.

So gut wie kein Programm ist auf Anhieb fehlerfrei. Deshalb gibt es keine Programmerstellung ohne *Debugging*. Ein Debugger erhöht die Effizienz beim Debugging erheblich. Übliche Eigenschaften eines Debuggers sind:

- Unterbrechungspunkte: Man kann im Programmtext Unterbrechungspunkte markieren, an denen ein Programm bei Betrieb im Debug-Modus angehalten, aber nicht beendet wird.

- schrittweise Programmausführung: Ab einem Unterbrechungspunkt kann das Programm entweder normal oder schrittweise fortgesetzt werden. Oft gibt es noch weitere Möglichkeiten, die Fortführung feinfühliger zu steuern.

- Inspektion von Objekten: Im angehaltenen Zustand eines Programms können die Objekte inspiziert werden, ein wesentliches Hilfsmittel zur Funktionsüberprüfung eines Programms.

- Textausgabe: Ein Debugger erlaubt die Ausgabe von Text. Dies kann man nutzen, den Zustand von Objekten anzuzeigen, ohne das Programm anzuhalten.

Leider sind auch Debugger sprachabhängig. Der Debugger heuriskos befindet sich noch in der Entwicklungsphase. Zwar gab es schon immer Kommandos, die das Anhalten an einem Unterbrechungspunkt und danach schrittweises Vorangehen ermöglichen, aber die graphische Unterstützung im Editor und die notwendige Kommunikation zwischen Steuerprogramm und Kern ist noch nicht fertig gestellt.

Sobald Programme einigermaßen fehlerfrei sind, kann man sich dem *Profiling* widmen. In der Bildverarbeitung, in der man es mit einer großen Menge von Daten zu tun hat, kommt es häufig darauf an, dass Algorithmen genügend schnell laufen. Dann ist das Hauptziel des Profiling die Optimierung der Laufzeit. In der Statuszeile des Hauptfensters in Abbildung 4.11 sieht man unten rechts ein Feld, das eine Zeitdauer in s oder ms anzeigt. Dies ist die Dauer des letzten ausgeführten Kommandos. Zusammen mit den Operatoren zur Zeitmessung kann man diese Anzeige für das Profiling nutzen.

Neben dem Hauptfenster gibt es noch das Ausgabefenster. In Abbildung 4.12 sind als Beispiel die Ergebnisse einer Merkmalsextraktion zu sehen. Man könnte dort auch die gemessenen Zeiten beim Profiling ausgeben. Eine Ausgabe erfolgt jeweils mit

```
List(obj);
```

wobei `obj` ein beliebiges heurisko-Objekt ist. Der Operator gibt jeden Datenpunkt des Objekts mit einer für das jeweilige Atom geltenden Formatierung aus. Die Formatierung kann mit

```
SetFormat(obj, formatstring);
```

gezielt gesetzt werden, wobei `formatstring` ähnliche Möglichkeiten wie die `printf`-Funktion in C erlaubt. Ausgaben im Ausgabefenster können skriptgesteuert gespeichert oder gelöscht werden. Das kann man für automatische Protokolle nutzen.

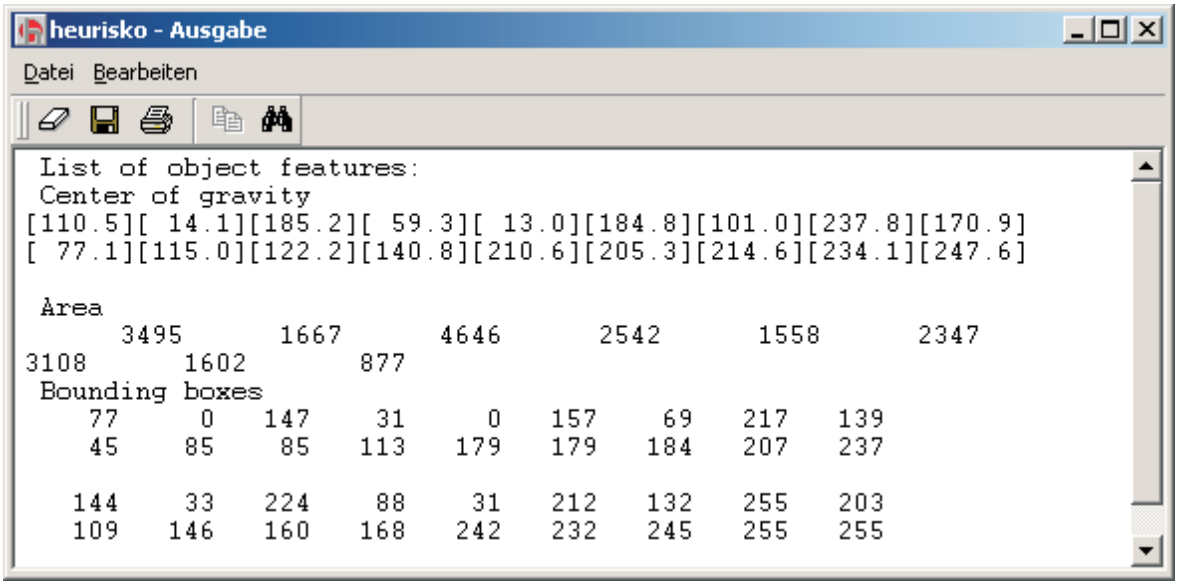


***Abbildung 4.12:*** *Ausgabefenster von heurisko mit Ergebnissen einer Merkmalsextraktion*

Ein weiteres Detail der Benutzeroberfläche ist in Abbildung 4.13 zu sehen. Der Operatordialog soll die Benutzung eines Operators erleichtern. Der Dialog zeigt erst einmal an, welche Varianten eines Operators es gibt. Dann bekommt man die Syntax und ein Textfeld mit einer Kurzbeschreibung gezeigt und darunter für jeden Parameter ein Auswahlfeld mit den zurzeit existierenden Objekten angeboten. Über eine Schaltfläche gelangt man optional in den Dialog zur Definition eines neuen Objektes, das automatisch in das Parameterfeld übernommen wird. Für andere Operatoren kann an dieser Stelle beispielsweise auch ein Schieberegler zur Einstellung des Parameters erscheinen. Zur Unterstützung bei der Objektauswahl wird im unteren Textfeld jeweils zum aktuell ausgewählten Objekt die Definition angezeigt. Des Weiteren enthält der Dialog auch die möglichen Operatormodi. All diese Informationen zu einem Operator muss sich die Benutzeroberfläche übrigens nicht merken. Sie erfragt sie vielmehr bei jedem Operatoraufruf neu vom Kern. Die Benutzeroberfläche weiß von sich aus nicht einmal, welche Operatoren der Kern zu bieten hat. Stattdessen meldet der Kern jeden ihm bekannten Operator an die Benutzeroberfläche, die den Operator in ihre Menüs einbaut. Das gilt auch für die in einem Skript definierten Operatoren.

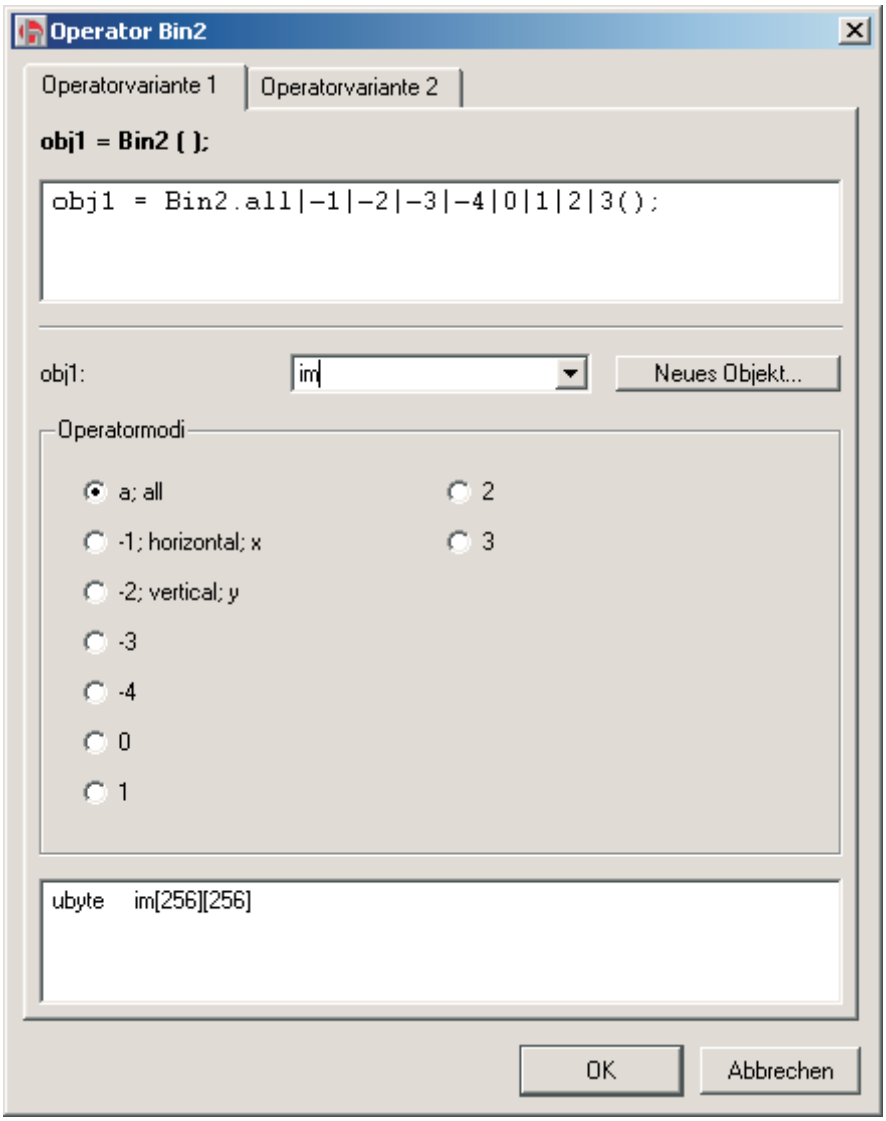


***Abbildung 4.13:*** *Operatordialog in heurisko*

# 5 Optimierungen

Abbildung 1.1 hat gezeigt, dass Bildverarbeitung ein hierarchischer und komplexer Prozess ist. Deshalb ist es auch nicht verwunderlich, dass es nicht nur *einen* Lösungsweg für ein gegebenes Problem gibt. Das gilt sowohl im Groben als auch im Feinen, d. h. bei der Auswahl der prinzipiellen Komponenten eines Lösungswegs und bei der Realisierung der Komponenten selbst. Für den Fall der schnellen Fouriertransformation wurde in Abschnitt 3.3 erläutert, dass es verschiedene Möglichkeiten der Zerlegung eines Problems in Teilprobleme gibt, welche unterschiedliche Laufzeiten zur Folge haben. In diesem Zusammenhang wurde die FFTW-Bibliothek beschrieben, welche zu einem gegebenen Objekt auf einem gegebenen Rechner den schnellsten Algorithmus bestimmt. Es wurde festgestellt, dass es nicht ausreicht, den Algorithmus mit der geringsten Anzahl von Operationen zu berechnen, weil die Laufzeit auch vom verwendeten System beeinflusst wird. Der optimale Algorithmus bezüglich Geschwindigkeit muss stattdessen durch Laufzeitmessungen experimentell ermittelt werden. Dies ist nur ein Beispiel dafür, dass eine Optimierung nicht trivial ist und deshalb automatisch von der Bildverarbeitungssoftware statt manuell vom Benutzer durchgeführt werden sollte. Als Optimierungskriterien kommen unter anderem in Betracht:

- **Geschwindigkeit:** Diese spielt wegen der großen Datenmengen sehr oft eine entscheidende Rolle. Beispielsweise müssen Taktzeiten eingehalten werden. Oder es hängt die Akzeptanz einer interaktiven Anwendung von der Ausführungszeit ab. Die Geschwindigkeit kann damit über die Anwendbarkeit eines Verfahrens überhaupt entscheiden, über das Ausmaß dessen, was in der zur Verfügung stehenden Zeit machbar ist, oder über den technischen Aufwand, der nötig ist, um die erforderliche Rechenleistung zur Verfügung zu stellen.

- **Genauigkeit:** Genauigkeit steht hier für ein zur Aufgabe passendes Gütekriterium. Das kann eine Richtung sein, eine Position, eine Fläche oder irgendeine in Bildern gemessene Größe. Das können aber auch der Anteil nicht erkannter fehlerhafter Ereignisse und der Anteil fälschlicherweise als Fehler erkannter Ereignisse bei einer Qualitätskontrolle sein. Auch die Genauigkeit kann darüber entscheiden, ob eine Aufgabe durch Bildverarbeitung zufrieden stellend lösbar ist.

- Im Prinzip sind auch **Kombinationen** der Kriterien denkbar. Von zwei Texturanalyseverfahren mit gleich gutem Erkennungsvermögen wird man die Auswahl nach einem zusätzlichen Kriterium treffen (siehe Abschnitt 3.3 und [Wagner 2000]).

Bei der Optimierung kann es um mehr gehen, als unter mehreren akzeptablen Lösungen die beste zu finden. Es geht dann um die notwendige Konfiguration, die ein System überhaupt sinnvoll anwendbar macht. Man kann also auch sagen, dass ein System *konfiguriert* statt *optimiert* wird. Wenn die Konfiguration, aus welchen Gründen auch immer, nicht durch eine einfache Einstellprozedur nach Anweisung bewerkstelligt werden kann, ist ein automatisches Verfahren gefragt. Und wenn diese automatische Einstellung nicht in einem einzigen Schritt vorgenommen werden kann, sondern durch Wiederholung immer gleicher oder ähnlicher Schritte und durch Vergleich erzielter Ergebnisse, handelt es sich um eine *lernende* Optimierung. FFTW z. B. lernt durch Probieren vieler Zerlegungen, welche die im gegebenen Fall optimale ist. Für die FFT spielen die eigentlichen Daten keine Rolle; die optimale Zerlegung ist auf einem gegebenen Rechner durch die Objektgröße alleine bestimmt. Das Lernen muss allerdings auf jedem Rechner durchgeführt werden. Bei anderen Bildverarbeitungsaufgaben

kann das ganz anders sein. Das Lernen für die Erkennung von Texturen z. B. benötigt Daten, die den später zu verarbeitenden Bildern ähnlich sind, muss aber in der Regel nur einmal auf einem beliebigen Rechner vorgenommen werden, wenn die Erkennungsleistung optimiert wird.

Eine Optimierung kann ebenso bezüglich des Umfangs sehr unterschiedlicher Natur sein. Während die Optimierung der FFT für ein bestimmtes Objekt nur einen Teilaspekt eines Bildverarbeitungssystems betrifft, kann eine Konfiguration auch das gesamte System umfassen. Eine Konfiguration dieser Art ist in [Wagner 2000] beschrieben (siehe Abschnitt 3.3). Entsprechend unterscheidet sich auch der Aufwand für eine automatische Konfiguration. Während FFTW nur einige Sekunden bis wenige Minuten für eine Optimierung benötigt, ist eine Systemkonfiguration mit heutigen Rechenleistungen häufig noch gar nicht oder nicht in vertretbarer Zeit realisierbar.

Die Art der Konfiguration hängt vom Bildverarbeitungssystem ab. Ein System, welches für eine bestimmte Aufgabe entworfen wurde und nur diese lösen soll, kann und soll möglichst in seiner Gesamtheit konfiguriert werden. Ein Bildverarbeitungssystem, das viel allgemeiner zur Entwicklung von mehr oder weniger beliebigen Bildverarbeitungsanwendungen gedacht ist, kann nur automatische Optimierungen für Teilaspekte anbieten. Als Beispiel wurde FFTW bereits ausführlich diskutiert.

Im Folgenden werden drei Typen von Optimierungen in heurisko vorgestellt. Abschnitt 5.1 beschreibt, wie heurisko bei verschachtelt angegebenen Operatoraufrufen die Anzahl der Speicherzugriffe und damit die Laufzeit reduzieren kann. Die zweite Art von Optimierung, vorgestellt in Abschnitt 5.2, geschieht schon vor der Erzeugung des Programms durch hardwarenahe Programmierung unter Ausnutzung der Multimedia-Instruktionssätze für Parallelverarbeitung. Das dritte Beispiel betrifft die optimale Auswahl eines aus mehreren heurisko-Operatoren (Abschnitt 5.3).

## 5.1 Operatorverschachtelung

Eine Bildverarbeitungsbibliothek bietet wegen der erwünschten Flexibilität eher viele elementare Operationen an als komplexe Operatoren zur Lösung eines vollständigen Problems. Deshalb ist Bildverarbeitung durch die Hintereinanderausführung vieler Operatoren gekennzeichnet. Das aber hat eine große Zahl von Speicherzugriffen zur Folge, die nach den Erläuterungen in Abschnitt 2.3.1 den Bearbeitungsfluss hemmen. Die Frage ist nun, ob man durch Geschick Speicherzugriffe einsparen kann. Das Vektorfunktionsprinzip in heurisko bietet eine Möglichkeit zur Optimierung des Datentransfers zwischen Speicher und CPU. Angenommen, es sei gemäß dem Pythagorassatz

$$c = \sqrt{a^2 + b^2}$$

zu berechnen. Da es dafür keinen direkten Operator gibt, muss man das in heurisko z. B. so programmieren:

```
c = Sqr(a);
d = Sqr(b);
c = Add(c, d);
c = Sqrt(c);
```

Es sind also vier Operationen auszuführen, zweimal Quadrieren, einmal Addieren und schließlich Ziehen der Quadratwurzel. Es wird zusätzlich das Objekt `d` zum Zwischenspeichern eines Teilergebnisses benötigt, wobei `d` die gleiche Größe wie die anderen Objekte hat.

Für die erste Operation werden die Eingabedaten in `a` vom Speicher in die CPU und die Ergebnisdaten in `c` von der CPU in den Speicher transportiert. Bei der zweiten Operation geschieht Analoges mit den Objekten `b` und `d`. Bei den folgenden beiden Operationen muss `c` in jeder Richtung transportiert werden, weil es sowohl ein Eingabe- als auch das Ausgabeobjekt darstellt. Der Datentransport bremst den Prozessor, weil die Daten nicht so schnell heran- und abtransportiert werden, wie der Prozessor sie verarbeiten kann. Moderne PCs sind deshalb mit einem schnellen Zwischenspeicher, dem so genannten Cache ausgestattet. Wenn sich der gerade bearbeitete Datenbereich im Cache befindet, kann die CPU vergleichsweise schnell arbeiten, bis andere Daten benötigt werden, die sich noch nicht im Cache befinden. Der Cache des Testrechners hat eine Größe von 512 kB. Wenn die Beispielobjekte Bilder der Größe 1024×1024 mit dem Datentyp `float` sind, belegen sie je 4 MB, passen also nicht in den Cache. So kann das Objekt `c` nach der ersten Operation nicht für die dritte Operation und nach der dritten nicht für die vierte im Cache verbleiben, sondern muss ungünstigerweise mehrmals hin und her transportiert werden.

Heurisko bietet die Möglichkeit, die obige Rechenvorschrift auch in der verschachtelten Form

```
c = Sqrt(Add(Sqr(a), Sqr(b)));
```

oder übersichtlicher mit

```
c = Sqrt(Sqr(a) + Sqr(b));
```

als eine einzige Anweisung anzugeben. Diese komplexe Anweisung zerlegt der Interpreter wieder, weil er dem Kern nur eine Folge von Elementaranweisungen übergeben kann. Nach der Literatur über Compiler-Bau geschieht diese Zerlegung beim Parsen üblicherweise durch die Bildung eines Baumes, aus dem dann die Folge der Elementaroperationen abgelesen werden kann [Holub 1990]. Als Ergebnis erhält man z. B.

```
t1 = Sqr(a);
t2 = Sqr(b);
t3 = Add(t1, t2);
c  = Sqrt(t3);
```

Man erkennt, dass der Interpreter drei temporäre Objekte einführen musste, um sich die Zwischenergebnisse aufzuheben. Ein intelligenter Interpreter könnte noch erkennen, dass `t1` und `t2` nach der Addition nicht mehr benötigt werden, und `t3` einsparen:

```
t1 = Sqr(a);
t2 = Sqr(b);
t1 = Add(t1, t2);
c  = Sqrt(t1);
```

Trotzdem wird immer noch ein temporäres Objekt mehr benutzt als bei der ersten Variante, die vom Benutzer ohne Verschachtelung eingegeben wurde. Die Verschachtelung bringt demnach dem Benutzer mehr Übersicht und Bequemlichkeit, verschärft aber durch die temporären Objekte das Speicherzugriffdilemma und erhöht überdies den Speicherbedarf. Für heurisko konnte jedoch für bestimmte Fälle eine Gegenmaßnahme gefunden werden, die nicht nur die geschilderten Nachteile verhindert, sondern auch noch die Zahl der Speicherzugriffe gegenüber der nicht verschachtelten Schreibweise verringert. Grundlage der Lösung bildet die Einführung einer Kennung für den Typ eines Operators. Erkennt der Interpreter, dass die an dem verschachtelten Ausdruck beteiligten Operatoren ausnahmslos die Kennung tragen, die sie als geeignete, mit Vektorfunktionen implementierte Operatoren ausweisen, erzeugt er für

den Kern einen speziellen Befehlscode. Dieser veranlasst den Kern, die Kette von Operationen vektorweise so auszuführen, als hätte man etwa

```
# Pseudocode

for all vectors av, bv, cv in a, b, c do;
    t1v = Sqr(av);
    t2v = Sqr(bv);
    t3v = Add(t1v, t2v);
    cv = Sqrt(t3v);
enddo;
```

geschrieben. Tatsächlich kennt der heurisko-Interpreter das Kommando `do` nicht. Es wurde absichtlich gewählt, um hervorzuheben, dass es sich hier um eine spezielle Operationsfolge unterhalb der Operatorebene handelt. Anstatt eine Operation nach der anderen auf jeweils den gesamten Objekten auszuführen, werden in einer Schleife alle Operationen auf jeweils einem Vektor der Objekte vorgenommen. Für Zwischenergebnisse werden drei temporäre Vektoren erzeugt. Mit den oben angegebenen Daten belegt jeder Zeilenvektor 4 kB. Für einen Schleifendurchlauf finden demnach alle benötigten Daten Platz im Cache, so dass beispielsweise das Zwischenergebnis der ersten Operation für die dritte Operation im Cache verbleiben kann.

***Tabelle 5-1:*** *Häufigkeiten der Zeilentransporte für jede Teiloperation und jedes Objekt für das Pythagoras-Rechenbeispiel nach dem Standardverfahren*

| Operation | Objekt | | | |
|---|---|---|---|---|
| | a | b | c | d |
| `c = Sqr(a)` | 1+0 | 0 | 0+1 | 0 |
| `d = Sqr(b)` | 0 | 1+0 | 0 | 0+1 |
| `c = Add(c,d)` | 0 | 0 | 1+1 | 1+0 |
| `c = Sqrt(c)` | 0 | 0 | 1+1 | 0 |
| gesamt | 1 | 1 | 5 | 2 |

***Tabelle 5-2:*** *Häufigkeiten der Zeilentransporte für jede Teiloperation und jedes Objekt für das Pythagoras-Rechenbeispiel nach dem Zeilenvektorverfahren*

| Operation | Objekt | | |
|---|---|---|---|
| | a | b | c |
| `t1 = Sqr(a)` | 1+0 | 0 | 0 |
| `t2 = Sqr(b)` | 0 | 1+0 | 0 |
| `t3 = Add(t1,t2)` | 0 | 0 | 0 |
| `c = Sqrt(t3)` | 0 | 0 | 0+1 |
| gesamt | 1 | 1 | 1 |

Um den möglichen Zeitgewinn ein wenig abzuschätzen, kann man vergleichen, wie häufig jede Bildzeile nach dem Standardverfahren mit vollständigen Objekten und wie häufig sie nach dem vektororientierten Verfahren zwischen Hauptspeicher und Cache transportiert werden muss. In Tabelle 5-1 ist die Bestimmung der Zeilentransporthäufigkeiten für das Standardverfahren, nach dem jede Teiloperation immer auf dem vollständigen Objekt durchgeführt wird, bevor die nächste Teiloperation folgt, aufgeführt. Tabelle 5-2 zeigt die Häufigkeiten für das vektororientierte Verfahren. Bei der Angabe *m*+*n* bezieht sich *m* auf den Vektortransport vom Speicher in den Cache und *n* auf den Transport vom Cache in den Speicher.

Demnach wird gemäß Tabelle 5-1 insgesamt jeder Vektor von `a` und `b` einmal, von `d` zweimal und von *c* fünfmal transportiert. Handelt es sich um zweidimensionale Objekte und hat jedes Objekt *y* Zeilen, finden 9*y* Zeilentransporte statt. Geht man für die spezielle Zeilenverarbeitung davon aus, dass alle für einen Schleifendurchlauf benötigten Zeilen im Cache Platz finden, reduziert sich nach Tabelle 5-2 der Transportaufwand für `c` auf die Speicherung am Ende. Für `a` und `b` ändert sich nichts gegenüber dem Standardverfahren, das Objekt *d* existiert nicht und die temporären Vektoren verbleiben immer im Cache. Insgesamt gibt es bei dieser Vorgehensweise also nur noch 3*y* Zeilentransporte.

Welchen Zeitgewinn man durch das Vektorverfahren tatsächlich erreicht, hängt unter anderem von der Komplexität der durchgeführten Operationen ab. Eine trigonometrische Operation benötigt im Verhältnis zum Datentransport wesentlich mehr Zeit als eine Operation, die der Prozessor mit *einem* Maschinenbefehl ausführen kann. Folglich kann man in einem solchen Fall keine Zeitersparnis erwarten. In Tabelle 5-4 und in Tabelle 5-5 sind die Rechenzeiten für die beiden Verfahren einander gegenübergestellt. Ein S in der zweiten Spalte bedeutet, dass die Zeiten dieser Zeile nach dem Standardverfahren ermittelt wurden; ein V zeigt an, dass das Vektorverfahren angewendet wurde. Wie die Beispiele in heurisko programmiert wurden, ist in Tabelle 5-3 erläutert. Für die mit V gekennzeichneten Zeilen ist zunächst die kompaktere Schreibweise aufgeführt und dann nach einem Pfeil die ebenso mögliche verschachtelte Schreibweise ohne Infix-Notation angegeben, in die der Interpreter die in der ersten Notation eingegebenen Kommandos überführt.

***Tabelle 5-3:*** *Beispiele, für die im Folgenden die Rechenzeiten nach dem Standardverfahren (S in der zweiten Spalte) und nach dem Vektorverfahren (V in der zweiten Spalte) gemessen werden*

| Operation | | heurisko-Kommandos |
|---|---|---|
| $c = \sqrt{a^2 + b^2}$ | S | `c=Sqr(a); d=Sqr(b); c=Add(c,d); c=Sqrt(c);` |
| | V | `c=Sqrt(a*a+b*b);` → `c=Sqrt(Add(Sqr(a),Sqr(b)));` |
| $c = a^2 + 2ab - b^2$ | S | `c=Sqr(a); d=Sqr(b); c=Sub(c,d); d=Mul(a,b); d=Mul(d,2.0); c=Add(c,d);` |
| | V | `c=Sqr(a)+2.0*a*b-Sqr(b);` → `c=Add(Sub(Sqr(a),Sqr(b)),Mul(2.0,Mul(a,b)));` |
| $c = a^{10}$ | S | `c=Mul(a,a); c=Mul(c,a); c=Mul(c,a); c=Mul(c,a); c=Mul(c,a); c=Mul(c,a); c=Mul(c,a); c=Mul(c,a); c=Mul(c,a);` |
| | V | `c=a*a*a*a*a*a*a*a*a*a;` → `c=Mul(Mul(Mul(Mul(Mul(Mul(Mul(Mul(Mul(a,a),a),a),a),a),a),a),a),a);` |
| $c = \left(a^2 + b^2\right)^2$ | S | `c=Sqr(a); d=Sqr(b); c=Add(c,d); c=Sqr(c);` |
| | V | `c=Sqr(a*a+b*b);` → `c=Sqr(Add(Sqr(a),Sqr(b)));` |
| $c = \ln\left(\sqrt{a}/\sqrt{b}\right)$ | S | `c=Sqrt(a); d=Sqrt(b); c=Div(c,d); c=Ln(c);` |
| | V | `c=Ln(Sqrt(a)/Sqrt(b));` → `c=Ln(Div(Sqrt(a),Sqrt(b)));` |

Die willkürlich gewählten ersten drei Beispiele in Tabelle 5-4 zeigen, dass deutliche Zeitgewinne möglich sind. Das Verhältnis der Zeiten in den mit S gekennzeichneten Zeilen zu den Zeiten in den mit V gekennzeichneten Zeilen ist in etwa konstant für unterschiedliche Zeilenzahlen. Die in der letzten Spalte ablesbare Ersparnis der Zeilentransporte lässt keine direkte Vorhersage des Ausmaßes des zu erwartenden Zeitgewinns zu. Die beiden letzten Beispiele wurden gezielt so gewählt, dass sich beide aus der gleichen Anzahl von Grundoperationen zusammensetzen. Das vierte Beispiel besteht dabei ausschließlich aus einzelnen Maschinen-

befehlen entsprechenden Bildpunktoperationen, während das fünfte Beispiel ausnahmslos komplexe Bildpunktoperationen enthält. Man erkennt, dass in diesem Fall durch das Vektorverfahren so gut wie kein Zeitgewinn möglich ist. Es bleibt aber wenigstens der nicht unerhebliche Vorteil einer übersichtlichen Schreibweise ohne die Nachteile durch temporäre Objekte. In Tabelle 5-5 sind abschließend die Messergebnisse für die ersten drei Bespiele aus Tabelle 5-4 bei verdoppelter Zeilenlänge zusammengefasst.

***Tabelle 5-4:*** *Vergleich der Rechenzeiten für ein paar Beispiele, die nach dem Standardverfahren (S in der zweiten Spalte) und nach dem Vektorverfahren (V in der zweiten Spalte) verarbeitet werden. Die Objekte sind zweidimensional mit einer Zeilenlänge von 1024. Die vorletzte Spalte gibt den mittleren Beschleunigungsfaktor an, die letzte Spalte die Anzahl Transporte je Objektzeile.*

| Operation | | Rechenzeit in ms für Zeilenlänge 1024, Datentyp `float`, Zeilenzahl wie angegeben | | | | | $t_S/t_V$ | Zeilen-transporte |
|---|---|---|---|---|---|---|---|---|
| | | 1024 | 2048 | 4096 | 8192 | 16384 | | |
| $c = \sqrt{a^2 + b^2}$ | S | 35 | 68 | 136 | 277 | 553 | 1,57 | 9 |
| | V | 22 | 44 | 88 | 176 | 351 | | 3 |
| $c = a^2 + 2ab - b^2$ | S | 30 | 58 | 114 | 231 | 463 | 3,41 | 15 |
| | V | 9 | 17 | 34 | 67 | 134 | | 5 |
| $c = a^{10}$ | S | 51 | 101 | 202 | 404 | 804 | 2,17 | 27 |
| | V | 23 | 46 | 92 | 192 | 380 | | 2 |
| $c = \left(a^2 + b^2\right)^2$ | S | 22 | 43 | 86 | 172 | 345 | 2,50 | 9 |
| | V | 9 | 17 | 34 | 68 | 140 | | 3 |
| $c = \ln\left(\sqrt{a} \,/\, \sqrt{b}\right)$ | S | 651 | 1301 | 2607 | 5220 | 10450 | 1,01 | 9 |
| | V | 647 | 1300 | 2602 | 5190 | 10400 | | 3 |

***Tabelle 5-5:*** *Vergleich der Rechenzeiten für ein paar Beispiele, die nach dem Standardverfahren (S in der zweiten Spalte) und nach dem Vektorverfahren (V in der zweiten Spalte) verarbeitet werden. Die Objekte sind zweidimensional mit einer Zeilenlänge von 2048. Die vorletzte Spalte gibt den mittleren Beschleunigungsfaktor an, die letzte Spalte die Anzahl Transporte je Objektzeile.*

| Operation | | Rechenzeit in ms für Zeilenlänge 2048, Datentyp `float`, Zeilenzahl wie angegeben | | | | | $t_S/t_V$ | Zeilen-transporte |
|---|---|---|---|---|---|---|---|---|
| | | 1024 | 2048 | 4096 | 8192 | 16384 | | |
| $c = \sqrt{a^2 + b^2}$ | S | 69 | 136 | 274 | 548 | 1210 | 1,59 | 9 |
| | V | 44 | 88 | 177 | 349 | 702 | | 3 |
| $c = a^2 + 2ab - b^2$ | S | 57 | 113 | 230 | 465 | 920 | 3,43 | 15 |
| | V | 17 | 33 | 66 | 132 | 275 | | 5 |
| $c = a^{10}$ | S | 101 | 204 | 403 | 804 | 1620 | 2,18 | 27 |
| | V | 46 | 91 | 190 | 370 | 750 | | 11 |

Das beschriebene Verfahren zur Beschleunigung durch Zusammenfassung von Vektorfunktionen lässt sich nur für diejenigen Bildverarbeitungsoperatoren anwenden, bei denen die Grundoperation auf einen Datenpunkt oder zumindest auf den Vektor beschränkt ist. Globale Transformationen wie die FFT oder globale Operationen wie die Berechnung eines Histogramms müssen erst für das gesamte Objekt abgeschlossen sein, bevor die nächste Operation beginnen kann. Bei Nachbarschaftsoperationen stimmt das zwar so nicht, aber im allgemeinen Fall hängt ein Datenpunkt von seiner Umgebung in allen Richtungen ab, so dass auch die zugehörigen Vektoren zur Verfügung stehen müssen. Das Vektorverfahren dieses Abschnitts und das in Abschnitt 4.3.4 beschriebene Verfahren für die Faltung lassen sich

deshalb nicht einfach kombinieren. Ob das Vektorverfahren anwendbar ist, erkennt der Interpreter an der erwähnten Operatorkennung.

## 5.2 Parallelverarbeitung

### 5.2.1 SIMD-Programmierung

Prozessoren sind für eine bestimmte Speicherwortlänge ausgelegt. Die heute am weitesten verbreiteten 32-Bit-Prozessoren arbeiten z. B. mit 32-Bit-Operanden in 32-Bit-Registern. Selbst wenn in einem Programm nur ganze Zahlen mit je 16 Bit benutzt werden, verarbeitet sie ein 32-Bit-Prozessor intern immer als 32-Bit-Daten; deren obere 16 Bit sind lediglich auf 0 gesetzt. In der häufig vorkommenden Bildverarbeitung mit Grauwertbildern werden meistens sogar nur 8 Bit je Bildpunkt benötigt. Der Prozessor ist in solchen Fällen nur schlecht genutzt. Das brachte die Hersteller auf den Gedanken, die vorhandenen Hardwareressourcen eines Prozessors mit vergleichsweise geringem Aufwand so umzurüsten, dass mit speziellen neuen Instruktionen in den vorhandenen Registern entweder ein Wort mit voller Registerbreite oder aber zwei Worte mit halber Breite oder vier Worte mit Viertelbreite usw. parallel verarbeitet werden können. Da in einem solchen Modus für jedes Teilwort die gleiche Operation ausgeführt wird, sagt man hierzu *SIMD*-Verarbeitung, wobei SIMD eine Abkürzung von *single instruction, multiple data* ist. Bei Intel hieß der erste zugehörige Instruktionssatz *MMX*. MM steht für Multimedia und zeigt, dass nicht die Bildverarbeitung die treibende Kraft für diese Entwicklung war, sondern, wie so häufig, ein Massenmarkt. Mittlerweile gibt es bei Intel mehrere Weiterentwicklungen des MMX und auch andere Prozessorhersteller haben ähnliche, teilweise kompatible Instruktionssätze in ihre Prozessoren integriert. Eine Übersicht über verschiedene Multimedia-Instruktionssätze und einiges zur Historie findet man in [Jähne und Herrmann 1999].

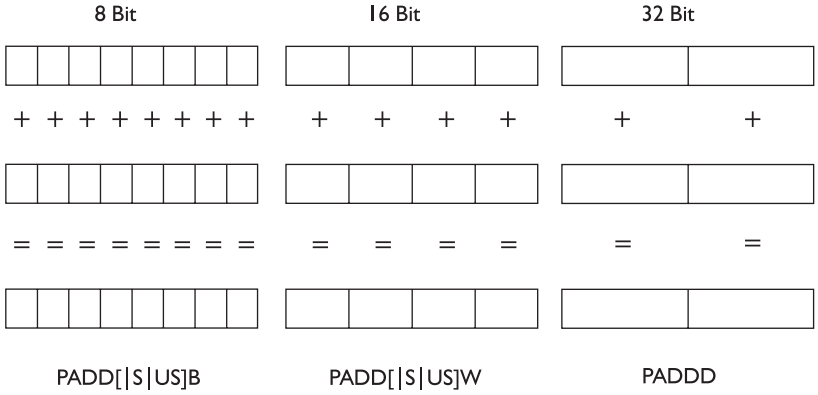


***Abbildung 5.1:*** *SIMD-Addition am Beispiel von Intels MMX*

Die SIMD-Register eines 32-Bit-Prozessors von Intel haben sogar eine Größe von 64 Bit, so dass parallel entweder acht 8-Bit-Pixel oder vier 16-Bit-Pixel oder zwei 32-Bit-Pixel verarbeitet werden können. Für die Addition ist dies in Abbildung 5.1 schematisch dargestellt. Der Nachteil der SIMD-Technik ist, dass sie bisher kein Compiler effektiv nutzen kann und dass sie im Gegensatz zu Standardprogrammiersprachen plattformabhängig ist. Man muss sich die Vorteile durch Assembler-Programmierung erkaufen. Für ein Bildverarbeitungssystem, welches wie heurisko weitgehend auf Vektorfunktionen basiert, lässt sich allerdings schon mit vergleichsweise geringem Aufwand eine merkliche Leistungssteigerung erzielen. Da ein großer Anteil der Prozessorzeit von den Vektorfunktionen benötigt wird, reicht es, nur jene zu optimieren. Wenn man sich dann noch vor Augen hält, dass viele Vektorfunktionen oft wieder verwendet werden und, ganz entscheidend, dass einmal erstellter Code auch noch auf den

nächsten Prozessorgenerationen läuft, lohnt sich der Programmieraufwand. Sorgt man dafür, dass jede benötigte Funktion zuerst einmal in einer Standardprogrammiersprache implementiert ist, kann man die Optimierung später bei Bedarf sukzessive als Alternative vornehmen und behält gleichzeitig ein zwar langsameres, aber plattformunabhängiges und portierbares Grundsystem in der Hinterhand. In heurisko sind Vektorfunktionen mit und ohne SIMD-Optimierung in separaten Modulen untergebracht. Beim Start ermittelt heurisko, welche Instruktionssätze vom Prozessor unterstützt werden und lädt dann entweder das Grundmodul ohne Optimierung oder das Modul mit dem zugehörigen SIMD-Code.

Eine Möglichkeit, mit SIMD-Code für einen Intel-Prozessor 8-Bit-Pixel zweier Objekte zu addieren und in einem dritten zu speichern, ist die folgende:

```
        …
        // Loop initialization
        movq        mm4, [ebx]
        movq        mm5, [ebx+8]
        movq        mm6, [ebx+16]
        movq        mm7, [ebx+24]
        movq        mm0, [esi]
        movq        mm1, [esi+8]
        movq        mm2, [esi+16]
        movq        mm3, [esi+24]
        add         esi, 32
        add         ebx, 32
        dec         ecx
        jz          m2

m1:     // Loop: Add 4*8 = 32 8-bit pixels per loop
        paddusb     mm0, mm4
        paddusb     mm1, mm5
        paddusb     mm2, mm6
        paddusb     mm3, mm7
        movq        mm4, [ebx]
        movq        mm5, [ebx+8]
        movq        mm6, [ebx+16]
        movq        mm7, [ebx+24]
        movq        [edi], mm0
        movq        [edi+8], mm1
        movq        [edi+16], mm2
        movq        [edi+24], mm3
        movq        mm0, [esi]
        movq        mm1, [esi+8]
        movq        mm2, [esi+16]
        movq        mm3, [esi+24]
        add         ebx, 32
        add         edi, 32
        add         esi, 32
        dec         ecx
        jg          m1
        …
```

Von dieser Vektorfunktion ist nur der wesentliche Teil gezeigt. In der aus 21 Instruktionen bestehenden Schleife werden jeweils 32 Bildpunkte addiert. Werden alle Instruktionen vom Prozessor sequentiell ausgeführt, werden demnach innerhalb von 21 Taktzyklen 32 Additionen vorgenommen. Für den Testrechner mit 2,6 GHz ergibt sich daraus eine theoretische Verarbeitungsrate von knapp 4 GPixel/s. Um diese Rate tatsächlich zu erreichen, müssten

während 21 Takten zweimal 32 Byte in den Prozessor transportiert werden, was einer Transportrate von knapp 8 GB/s entspräche.

Tabelle 5-6 zeigt, dass die Addition unter den angegebenen Bedingungen im SIMD-Modus für 16-Bit-Zahlen (Typ `ushort`) größenordnungsmäßig viermal und für 8-Bit-Zahlen (Typ `ubyte`) achtmal schneller läuft als im normalen Modus. Damit wird praktisch genau die Beschleunigung erreicht, die man sich aus der vier- bzw. achtfach erhöhten Anzahl von gleichzeitig bearbeiteten Bildpunkten erhofft. Die Messergebnisse zeigen auch, dass für 32-Bit-Fließkommazahlen vom Typ `float` und für ganze Zahlen mit 32 Bit vom Typ `ulong` kein nennenswerter Vorteil erreicht wird. Erwartet hätte man eine Verdoppelung der Verarbeitungsrate. Darüber hinaus belegen die gemessenen Raten, dass die theoretische Verarbeitungsrate des Prozessors nicht annähernd erreicht wird, denn bei der Taktrate von 2,6 GHz könnten $2{,}6\times10^9$ Bildpunkte vom Typ `float` addiert werden, wenn die Daten rechtzeitig zur Verfügung stünden. Dazu müssten $2\times4\times2{,}6\times10^9$ Byte/s = 20,8 GByte/s (zwei Operanden mit je 32 Bit) in den Prozessor und die Hälfte davon wieder in den Speicher transportiert werden, was schon die theoretische Datenrate von 6,4 GByte/s zwischen Hauptspeicher und Cache des Testrechners bei weitem übersteigt (Abschnitt 2.3.1).

***Tabelle 5-6:*** *Verarbeitungsrate für die Addition mit verschiedenen Datentypen ohne und mit SIMD auf dem Testrechner für Objekte der Größe 1024×1024*

| Datentyp | Verarbeitungsrate ohne SIMD | | Verarbeitungsrate mit SIMD | |
|---|---|---|---|---|
| | [$10^9$ Pixel/s] | [GB/s] | [$10^9$ Pixel/s] | [GB/s] |
| `double` | 0,105 | 0,840 | - | - |
| `float` | 0,171 | 0,684 | 0,189 | 0,756 |
| `ulong` | 0,175 | 0,700 | 0,187 | 0,748 |
| `ushort` | 0,112 | 0,224 | 0,372 | 0,744 |
| `ubyte` | 0,105 | 0,105 | 0,741 | 0,741 |

***Tabelle 5-7:*** *Verarbeitungsraten in $10^6$ Pixel/s für vier Prozessorgenerationen, gemessen mit heurisko 4. –S und +S bezeichnen Operationen ohne bzw. mit SIMD-Optimierung. Alle Objekte haben eine Größe von 512×512 Bildpunkten und den bei jeder Operation vermerkten Datentyp. Ausgabeobjekte und Eingabeobjekte sind nicht identisch.*

| Operation | Pentium 166 MHz | | Pentium II 400 MHz | | Pentium III 800 MHz | | Pentium 4 2,6 GHz | |
|---|---|---|---|---|---|---|---|---|
| | -S | +S | -S | +S | -S | +S | -S | +S |
| Kopieren | | | | | | | | |
| `ubyte` | - | - | - | - | 191 | 198 | 1748 | 2097 |
| `short` | - | - | - | - | 88 | 89 | 552 | 552 |
| Addition | | | | | | | | |
| `ubyte` | 11,0 | 49,3 | 77,5 | 143 | 101 | 157 | 655 | 807 |
| `short` | 7,4 | 23,4 | 24,3 | 32,4 | 47 | 58 | 131 | 388 |
| Multiplikation | | | | | | | | |
| `ubyte` | 6,0 | 38,8 | 65,4 | 121 | 97 | 119 | 157 | 807 |
| `short` | 5,5 | 23,3 | 23,7 | 32,5 | 45 | 59 | 96 | 388 |
| Faltung 5×5 | | | | | | | | |
| `ubyte` | 0,3 | 4,1 | 1,0 | 7,8 | 3 | 22 | 8 | 62 |

Tabelle 5-7 zeigt einen Vergleich der mit SIMD erreichbaren Leistungen über vier Generationen von Intel-Prozessoren hinweg. Die Messungen hierzu wurden der besseren Vergleichbarkeit wegen mit der Version 4 von heurisko durchgeführt, da die beiden älteren Systeme nicht mehr zur Verfügung standen und deshalb auf früher veröffentlichte Ergebnisse zurückgegriffen werden musste. Die Ergebnisse bestätigen die bereits in Abschnitt 2.3.1 getroffene Feststellung, dass Vorhersagen der Leistungssteigerung durch SIMD-Programmierung im konkreten Fall nur schwer möglich sind. Außer natürlich der Hardware spielen Datentyp, Objektgröße und die benachbarten Programmbefehle eine wichtige Rolle. Kann nämlich beispielsweise ein im Cache befindlicher Datenvektor gleich durch mehrere Funktionen bearbeitet werden, ist in dieser Zeit kein Transport über den langsameren Bus außerhalb des Prozessors nötig.

Ein letztes Experiment in diesem Abschnitt gilt dem Vergleich der bereits im Abschnitt 4.3.3 vorgestellten drei Implementierungen der Binomialglättung mit einer 3×3-Maske unter Berücksichtigung einer teilweisen Optimierung durch SIMD. Die Messergebnisse in Tabelle 5-8 zeigen keine Abhängigkeit von der Zeilenlänge. Systematisch beschleunigt durch SIMD-Code sind die Vektorfunktionen nur in den fett gedruckten Fällen. In den anderen Fällen mag die eine oder andere Kopierfunktion optimierten Code enthalten.

***Tabelle 5-8:*** *Verarbeitungsraten in $10^6$ Pixel/s für drei verschiedene Implementierungen der Binomialfilterung mit einer 3×3-Maske bei verschiedenen Zeilenlängen. –S und +S bezeichnen Operationen ohne bzw. mit SIMD-Optimierung. Alle Objekte haben 1024 Zeilen und den jeweils vermerkten Datentyp. Ausgabeobjekt und Eingabeobjekt sind nicht identisch. Fetter Druck bedeutet systematische Beschleunigung durch SIMD-Code.*

| Bildgröße | Datentyp | `Bin2N()` | | `Bin2S()` | | `Bin2()` | |
|---|---|---|---|---|---|---|---|
| | | -S | +S | -S | +S | -S | +S |
| 512×1024 | `float` | 82 | 81 | 40 | 42 | 128 | 128 |
| | `ushort` | 131 | 126 | 58 | 63 | 164 | 177 |
| | `ubyte` | 33 | 48 | 90 | **192** | 69 | **357** |
| 1024×1024 | `float` | 81 | 80 | 39 | 40 | 128 | 131 |
| | `ushort` | 133 | 125 | 58 | 61 | 169 | 201 |
| | `ubyte` | 33 | 48 | 75 | **129** | 70 | **367** |
| 2048×1024 | `float` | 81 | 73 | 38 | 37 | 123 | 114 |
| | `ushort` | 131 | 124 | 55 | 54 | 191 | 192 |
| | `ubyte` | 33 | 49 | 75 | **124** | 72 | **372** |

Für eine tiefer gehende Diskussion der mit Multimedia-Instruktionssätzen realisierbaren Bildverarbeitungsfunktionen sei wieder auf [Jähne und Herrmann 1999] verwiesen.

### 5.2.2 Daten-Alignment

Zwischen Cache und Prozessor wird aus Effizienzgründen immer eine Mindestmenge an Daten transportiert, die zudem an bestimmten Adressen im Speicher liegen. Für einen Prozessor mit einer Datenbusbreite von 256 Bit könnte die Mindestmenge eben diese 256 Bit betragen und die Anfangsadressen wären solche, die durch 256 ohne Rest teilbar sind. Weiter unten stellt sich heraus, dass es günstig ist, alle Vektoren von heurisko-Objekten an solchen Speicheradressen beginnen zu lassen. Dazu kann man davon ausgehen, dass das Betriebssystem angeforderten Speicher und damit den ersten Vektor eines Objekts selbsttätig an einer Adresse mit der genannten Eigenschaft beginnen lässt. Angenommen, es befinde sich ein Bild im Speicher. Dann kommt es auf die Länge der Bildzeile an, ob alle anderen Zeilen außer der ersten ebenso an Adressen mit der gewünschten Eigenschaft liegen. Ist dies nicht der Fall,

kann man künstlich dafür sorgen, indem man jede Zeile am Ende um die gleiche Anzahl Bits verlängert. Daten, die an bestimmten Adressen ausgerichtet sind, bezeichnet man im Englischen als *aligned*.

***Tabelle 5-9:*** *Vergleich der Ausführungszeiten auf dem Testrechner für die Addition mit "aligned" Objekten und "not aligned" Objekten vom Typ* `ubyte` *für verschiedene Zeilenlängen mit und ohne SIMD-Code. Alle Bilder hatten 512 Zeilen und wurden in einer Schleife 100-mal addiert.*

| Zeilenlänge | -SIMD | | | +SIMD | | |
|---|---|---|---|---|---|---|
| | „aligned“ t1 [ms] | „not aligned“ t2 [ms] | t2/t1 [%] | „aligned“ t1 [ms] | „not aligned“ t2 [ms] | t2/t1 [%] |
| 1024 | 483 | 480 | 99 | 71 | 73 | 103 |
| 1025 | 482 | 484 | 100 | 66 | 82 | 124 |
| 1026 | 482 | 499 | 104 | 63 | 79 | 125 |
| 1027 | 483 | 484 | 100 | 64 | 81 | 127 |
| 1028 | 484 | 486 | 100 | 66 | 71 | 109 |
| 1029 | 485 | 483 | 100 | 66 | 78 | 118 |
| 1030 | 487 | 495 | 102 | 67 | 73 | 109 |
| 1031 | 483 | 486 | 101 | 66 | 74 | 112 |
| 1032 | 486 | 486 | 100 | 67 | 70 | 105 |

***Tabelle 5-10:*** *Vergleich der Ausführungszeiten auf dem Testrechner für die Addition mit "aligned" Objekten und "not aligned" Objekten vom Typ* `float` *für verschiedene Zeilenlängen mit und ohne SIMD-Code. Alle Bilder hatten 512 Zeilen und wurden in einer Schleife 100-mal addiert.*

| Zeilenlänge | -SIMD | | | +SIMD | | |
|---|---|---|---|---|---|---|
| | aligned t1 [ms] | not aligned t2 [ms] | t2/t1 [%] | aligned t1 [ms] | not aligned t2 [ms] | t2/t1 [%] |
| 1024 | 300 | 301 | 100 | 282 | 293 | 104 |
| 1025 | 284 | 298 | 105 | 236 | 466 | 198 |
| 1026 | 282 | 292 | 104 | 233 | 252 | 108 |
| 1027 | 284 | 291 | 103 | 235 | 458 | 195 |
| 1028 | 284 | 284 | 100 | 237 | 235 | 99 |
| 1029 | 285 | 283 | 99 | 231 | 456 | 198 |
| 1030 | 286 | 300 | 105 | 231 | 237 | 102 |
| 1031 | 285 | 284 | 100 | 248 | 459 | 185 |
| 1032 | 286 | 285 | 100 | 232 | 239 | 103 |

In heurisko werden Objekte standardmäßig an durch 256 ohne Rest teilbaren Adressen ausgerichtet. Man kann aber mit dem Attribut `noalign` auch dafür sorgen, dass Daten nicht zwangsweise ausgerichtet werden. Das ist beispielsweise für die Verwendung des Paketes FFTW mit den FFT-Operatoren nötig. Betrachtet sei nun ein Bild vom Typ `ubyte` mit einer Zeilenlänge von 1024. Da jeder Bildpunkt 8 Bit belegt, entsprechen 256 Bit 32 Bildpunkten. Da zudem 32 Teiler von 1024 ist, liegen alle Zeilenanfänge des Bildes „aligned“ im Speicher, unabhängig davon, ob das Objekt in heurisko mit oder ohne `noalign` definiert wurde. Anders sieht es bei der Zeilenlänge 1025 aus. Hier fehlen jeder Zeile 31 Bildpunkte, damit die folgende Zeile im Speicher wie gewünscht ausgerichtet ist. Hat das Bild stattdessen den Datentyp `float`, entsprechen 256 Bit nur noch 8 Bildpunkten. Es gilt aber auch hier, dass das Bild bei einer Zeilenlänge von 1024 im Speicher immer ausgerichtet ist, bei 1025 jedoch

nicht. Wie sich das auf die Laufzeit der Addition auswirkt, ist in Tabelle 5-9 für den Typ `ubyte` und in Tabelle 5-10 für den Typ `float` zu sehen.

Während die Ausrichtung im Speicher für die C-Programmierung ohne SIMD-Instruktionen keinen Einfluss auf die Laufzeit hat, erkennt man, dass eine fehlende Ausrichtung eine Verlängerung der Laufzeit verursacht. Beim Datentyp `ubyte` fällt die Verschlechterung für die aktuelle Implementierung der Addition in heurisko mit 20 % noch moderat aus. Beim Datentyp `float` ist dagegen eine Verdoppelung der Laufzeit zu beobachten. Das heißt, die Laufzeit wird sogar schlechter als ohne SIMD-Code. Man erkennt, dass die Laufzeiterhöhung für `float` nur bei allen ungeradzahligen Zeilenlängen auftritt. Dass für die tatsächlichen Verhältnisse die konkrete Implementierung eine Rolle spielt, wurde bei früheren Messungen deutlich. Statt mit nur 20 % wurde die fehlende Ausrichtung beim Datentyp `ubyte` mit 500 % Laufzeiterhöhung bestraft. Wie sich die Datenausrichtung im Speicher für die aktuellen Implementierungen in heurisko beim Kopieren auswirkt, ist in Tabelle 5-11 für `ubyte` und in Tabelle 5-12 für `float` zu sehen.

***Tabelle 5-11:*** *Vergleich der Ausführungszeiten auf dem Testrechner für das Kopieren von "aligned" Objekten und "not aligned" Objekten vom Typ `ubyte` für verschiedene Zeilenlängen mit und ohne SIMD-Code. Alle Bilder hatten 512 Zeilen und wurden in einer Schleife 100-mal kopiert.*

| Zeilenlänge | -SIMD | | | +SIMD | | |
|---|---|---|---|---|---|---|
| | aligned t1 [s] | not aligned t2 [s] | t2/t1 [%] | aligned t1 [s] | not aligned t2 [s] | t2/t1 [%] |
| 1024 | 204 | 204 | 100 | 49 | 50 | 102 |
| 1025 | 205 | 208 | 101 | 47 | 50 | 107 |
| 1026 | 222 | 206 | 93 | 47 | 51 | 109 |
| 1027 | 206 | 206 | 100 | 47 | 51 | 108 |
| 1028 | 207 | 205 | 99 | 47 | 53 | 114 |
| 1029 | 207 | 206 | 100 | 46 | 52 | 113 |
| 1030 | 208 | 206 | 99 | 47 | 63 | 134 |
| 1031 | 206 | 209 | 101 | 46 | 53 | 114 |
| 1032 | 207 | 205 | 100 | 47 | 50 | 107 |

***Tabelle 5-12:*** *Vergleich der Ausführungszeiten auf dem Testrechner für das Kopieren von "aligned" Objekten und "not aligned" Objekten vom Typ `float` für verschiedene Zeilenlängen mit und ohne SIMD-Code. Alle Bilder hatten 512 Zeilen und wurden in einer Schleife 100-mal kopiert.*

| Zeilenlänge | -SIMD | | | +SIMD | | |
|---|---|---|---|---|---|---|
| | aligned t1 [s] | not aligned t2 [s] | t2/t1 [%] | aligned t1 [s] | not aligned t2 [s] | t2/t1 [%] |
| 1024 | 200 | 203 | 101 | 189 | 189 | 100 |
| 1025 | 173 | 198 | 114 | 170 | 191 | 112 |
| 1026 | 178 | 193 | 109 | 168 | 185 | 110 |
| 1027 | 176 | 193 | 110 | 169 | 189 | 112 |
| 1028 | 176 | 187 | 106 | 169 | 179 | 106 |
| 1029 | 177 | 187 | 106 | 169 | 183 | 108 |
| 1030 | 176 | 182 | 103 | 170 | 175 | 103 |
| 1031 | 177 | 184 | 104 | 169 | 179 | 106 |
| 1032 | 178 | 175 | 99 | 169 | 169 | 100 |

### 5.2.3 Mehrprozessorbetrieb

Während es im letzten Abschnitt im Prinzip darum ging, vorhandene Ressourcen besser zu nutzen, geht es hier darum, durch mehr Ressourcen die Leistung zu verbessern. Wenn man es schafft, die vorhandene Arbeit auf mehrere Prozessoren aufzuteilen und sie parallel ausführen zu lassen, kann sich die Rechenleistung eines Systems beträchtlich erhöhen. Es gibt zwei Arten des Mehrprozessorbetriebs. Bei der ersten befinden sich alle Prozessoren im gleichen Rechnersystem und es wird nur ein Prozess der Anwendung gestartet. Dieser Prozess verteilt die Arbeit auf mehrere Threads. Für die Zuteilung der verfügbaren Prozessorzeiten zu den Threads sorgt das Betriebssystem. Bei der zweiten Art der Parallelverarbeitung werden die Ressourcen in einem Netzwerk genutzt. Auf jedem beteiligten Rechner wird ein Prozess der Anwendung gestartet. Ein Rechner fungiert als Master, die anderen als Slaves. Eventuell sind auch andere Betriebsweisen möglich; darum soll es hier aber nicht gehen.

In jedem Fall muss die Anwendung beim Mehrprozessorbetrieb die Arbeit verteilen und synchronisieren. Es müssen aber auch die Daten zur Verfügung gestellt werden. Im Netzbetrieb heißt das, dass die Daten über das Netzwerk gesandt werden müssen. Mit heurisko ist ein Mehrprozessorbetrieb bisher noch nicht ernsthaft realisiert worden. In Abbildung 4.1 erkennt man aber, dass in der Architektur bereits Vorbereitungen getroffen sind. Der Kern ist strikt vom Interpreter getrennt. Der Interpreter ist in der Lage, mehrere Kerne zu starten und zu verwalten. Ausgenutzt wurde diese Trennung bisher in zweierlei Weise. Zu Beginn der Entwicklungsarbeiten für heurisko war ein normaler PC mit einem 80286-Prozessor zu langsam für Bildverarbeitung. Dass trotzdem PCs eingesetzt wurden, hatte im Wesentlichen zwei Gründe: Zum einen waren PCs vergleichsweise kostengünstig und zum anderen gab es für sie reichhaltige Datenerfassungs- und Datenverarbeitungs-Hardware. Deshalb wurde der PC mit einer Beschleunigerkarte ausgerüstet, die mit einem i860-Prozessor bestückt war. Der heurisko-Kern lief auf der Zusatzkarte, der Interpreter und die Benutzerschnittstelle residierten auf dem Host-PC. Die andere Nutzung der Trennung von Interpreter und Kern war in einem Testsystem, das ebenfalls aus einem PC mit einer Beschleunigerkarte bestand. In diesem Fall lief heurisko mit zwei Kernen, ein Kern auf dem Subsystem, der andere auf dem Host-Prozessor. Ziel dieses Testsystems war, herauszufinden, ob eine C-Portierung des Kerns von heurisko auf den Prozessor Imagine der holländischen Firma Arcobel sinnvoll ist. Die Aussagen des Prozessorherstellers ließen einen glauben, man könne mit dem vorhandenen C-Compiler aus C-Code ein leistungsfähiges Programm erzeugen, das man später nach Bedarf stückweise für den Prozessor mit Assembler optimieren könnte. Es erwies sich jedoch, dass ein C-Programm keine Chance gegen ein Programm für einen Standardprozessor hatte. Außerdem wurde die binäre Programmdatei etwa zehnmal größer als die Programmdatei auf dem Host. Dies hätte den Bedarf an statischem und damit sehr teurem RAM in die Höhe getrieben. Details findet man in [Herrmann 1996].

## 5.3 Optimale Filterung

### 5.3.1 Das Modul hk_opt

Wie bei der Fouriertransformation gibt es in der Bildverarbeitung selbst bei einfachen Operationen mehrere Implementierungsmöglichkeiten, die sich in ihrer Ausführungsgeschwindigkeit und eventuell auch ihrer Genauigkeit unterscheiden können. Da es aber nach den bisherigen Feststellungen, unter anderem in Abschnitt 2.3.1, unmöglich ist, für jeden Fall die optimale Implementierung vorherzusagen, empfiehlt es sich, die möglichen Implementierungen vorzunehmen und eine automatische Auswahl der im konkreten Fall optimalen Implementierung zu ermöglichen. Für solche Optimierungen in heurisko wurde das Erweiterungsmodul

hk_opt erstellt. Die Nutzung dieses Moduls ist ähnlich der des Moduls hk_ft für die FFT mit FFTW (siehe Abschnitt 3.3.1). Für einen zu optimierenden Operator stellt hk_opt einen Operator zur Verfügung, dessen Name sich aus dem des Originaloperators, einem Suffix für die Art der Optimierung und dem Suffix Learn zusammensetzt. Liegen in heurisko mehrere Implementierungen der gleichen Funktion vor, gibt es dazu eine entsprechende Anzahl von Operatoren mit ähnlichem Namen. Besteht die Optimierung dann aus der Auswahl eines der Operatoren, bietet hk_opt einen Lernoperator an, dessen Name den Kern der Originalnamen und die erwähnten Suffixe enthält. Als Beispiel wurden bereits die Operatoren Bin2N(), Bin2S() und Bin2() vorgestellt. Der zugehörige Lernoperator mit Geschwindigkeitsoptimierung heißt Bin2FastestLearn(). Nach dem Lernen kann der optimale Operator ohne das Suffix Learn verwendet werden. Die Originaloperatoren stehen damit unverändert weiter zur Verfügung.

Der Aufbau des Moduls hk_opt entspricht dem der anderen heurisko-Erweiterungsmodule:

- Die Datei hk_opt.c enthält die Funktionen, die jedes Erweiterungsmodul exportieren muss, z. B. zur Initialisierung des Moduls. Die zugehörige Header-Datei ist hk_opt.h.
- In mop_opt.c finden sich die Listen mit den Attributen der von hk_opt angebotenen Operatoren.
- Die Datei m_opt.c enthält die Implementierungen dieser Operatoren; die zugehörigen Definitionen stehen in mop_opt.h.

Alle weiteren Dateien sind modulspezifische Erweiterungen.

### 5.3.2 Alternative Operatoren

In heurisko gibt es für eine bestimmte Bildverarbeitungsoperation mitunter mehrere Operatoren, deren Implementierungen sich unterscheiden. Dadurch kann es sein, dass ein Operator schneller oder genauer als ein anderer ist. Als Beispiel wurde schon mehrmals die Glättung durch Faltung mit der Binomialmaske mit drei Maskenkoeffizienten in jeder Richtung betrachtet. Die Maske ist separierbar und hat in jeder Richtung die gleichen Koeffizienten $[1 \quad 2 \quad 1]$. heurisko bietet dazu drei Operatoren gemäß Tabelle 5-13 an. Die Auswahl von Richtungen, in denen die Filterung stattfindet, ist in Form von Operatormodi realisiert. Die Angaben zur Richtungswahl in der Tabelle zeigen, dass die Operatoren teils gemeinsame, teils unterschiedliche Modi anbieten. Je nach gewünschtem Modus stehen also alle oder nur ein Teil der Operatoren als Alternativen zur Verfügung. Eine weitere Einschränkung der Auswahlmöglichkeiten kann durch Unterschiede bezüglich der von den einzelnen Operatoren unterstützten Objektdimensionen und Datentypen bestehen.

***Tabelle 5-13:*** *Eigenschaften der Bin2*-Operatoren in heurisko*

| | Bin2N() | Bin2S() | Bin2() |
|---|---|---|---|
| Beschreibung | nicht-separable Faltung | separable Faltung | besonders schnelle separable Faltung |
| SIMD | nein | für ubyte | für ubyte |
| Dimension | ≤ 3-D | ≤ 4-D | ≤ 3-D |
| Richtungsauswahl | keine | beliebig viele Richtungen | alle Richtungen oder eine einzige beliebige |

### 5.3.3 Auswahl des optimalen Operators

Zur automatischen Auswahl des bezüglich eines oder mehrerer Kriterien optimalen Operators wird in die Initialisierungsphase des Anwenderprogramms ein Lernschritt durch Aufruf des zugehörigen Lernoperators eingebaut. Diesem Lernschritt müssen die Objekte und gegebenenfalls der gewünschte Modus übergeben werden, mit denen der ausgewählte Operator im späteren Anwendungsfall arbeiten soll. Einfach ist das, wenn die Optimierung nicht von den konkreten Daten abhängt. Dann reicht es, im Lernschritt die Struktur und die Datentypen der Objekte zu kennen. Nach dem Lernschritt speichert das Programm, dass für die gegebenen Bedingungen der gefundene Operator anzuwenden ist.

Ein wichtiger Aspekt beim Speichern der Lernergebnisse ist, dass bei einem Aufruf im eigentlichen Programm der optimale Operator schnell gefunden wird. Dies wird in `hk_opt` wie folgt erreicht: Jedes Lernergebnis wird in eine zum Lernoperator gehörige lineare Liste als erstes Element eingefügt. Diese Liste wird beim Verlassen von heurisko auf der Festplatte des Systems gespeichert und beim nächsten Start wieder eingelesen. Daneben existiert eine „heiße“ Liste, die anfangs leer ist. Wird jetzt ein Lernschritt ausgeführt, sieht `hk_opt` zuerst in der „heißen“ Liste nach, ob dort bereits ein Eintrag vorhanden ist. Wenn ja, ist der Lernschritt damit abgeschlossen. Wenn nicht, sucht `hk_opt` die andere Liste durch. Findet es dort schon ein Ergebnis, übernimmt es dies an den Anfang der „heißen“ Liste und beendet den Lernschritt. Nur wenn in beiden Listen noch kein Eintrag zu finden ist, wird ein Lernverfahren in Gang gesetzt, an dessen Ende der Eintrag des Ergebnisses an den Anfang der „heißen“ Liste steht. Beim Aufruf des optimalen Operators kann man also nach einem Lernschritt immer davon ausgehen, dass der gesuchte Operator in der kurzen „heißen“ Liste steht und schnell gefunden wird.

***Tabelle 5-14:*** *Ausführungszeiten in ms für drei verschiedene Implementierungen der Binomialfilterung mit einer 3×3-Maske im Vergleich zum optimalen Operator bei verschiedenen Zeilenlängen. Alle Objekte haben 1024 Zeilen und den jeweils vermerkten Datentyp.*

| Bildgröße | Datentyp | `Bin2N()` | `Bin2S()` | `Bin2()` | `Bin2Fastest()` |
|---|---|---|---|---|---|
| 512×1024 | `float` | 7 | 12 | 4 | 4 |
| | `ushort` | 4 | 9 | 3 | 3 |
| | `ubyte` | 11 | 3 | 2 | 2 |
| 1024×1024 | `float` | 77 | 27 | 8 | 8 |
| | `ushort` | 8 | 17 | 6 | 6 |
| | `ubyte` | 21 | 8 | 3 | 3 |
| 2048×1024 | `float` | 28 | 56 | 19 | 19 |
| | `ushort` | 16 | 36 | 11 | 11 |
| | `ubyte` | 43 | 17 | 5 | 5 |

Im Beispiel der oben erwähnten `Bin2*`-Opratoren wurde eine Optimierung der Ausführungszeit realisiert. Dazu enthält das Modul `hk_opt` den Operator `Bin2FastestLearn()`. Diesem Operator übergibt man genau die Objekte, - die konkreten Daten spielen hier keine Rolle - die man dem `Bin2`-Operator übergeben würde. Nach der Ausführung des Lernschritts führt heurisko bei Aufruf von `Bin2Fastest()` automatisch den für den gegebenen Fall schnellsten zur Verfügung stehenden `Bin2`-Operator aus. Da die `Bin2*`-Operatoren mehrere Modi offerieren, wurde `Bin2FastestLearn()` mit der Vereinigung aller `Bin2`-Modi ausgestattet. Verlangt der Benutzer dann einen bestimmten Modus, prüft `Bin2FastestLearn()` zunächst, für welche `Bin2`-Operatoren dieser

Modus auch tatsächlich implementiert wurde. Nur unter jenen Operatoren kann ausgewählt werden. Einige Testergebnisse zeigt Tabelle 5-14. Die Laufzeit von `Bin2Fastest()` ist immer gleich der geringsten Laufzeit der anderen drei Operatoren, wobei offensichtlich ausnahmslos `Bin2()` der schnellste Operator ist. Auch wenn man die Zeiten nicht rundet, fällt kein Unterschied zwischen den Laufzeiten von `Bin2Fastest()` und `Bin2()` auf; die Suche in der „heißen“ Liste fällt nicht ins Gewicht.

Dies ist nur ein erstes einfaches Beispiel. Es wird weitere Operationen geben, für die der optimale Operator ausgewählt werden kann. So gibt es in heurisko beispielsweise mehrere Implementierungen einer Ableitung erster Ordnung.

# 6 Synthese und Applikationsbeispiele

In diesem Abschnitt wird an fünf praktischen Applikationsbeispielen mit heurisko gezeigt, wie die zuvor beschriebenen Konzepte eingesetzt werden können. Die Beispiele werden in der Hauptsache aus der Sicht des Entwicklers der Bildverarbeitungsalgorithmik beschrieben. Die Beschreibung geht deshalb nur dann auf Details der Systeme außerhalb der Algorithmik ein, wenn es für die Bildverarbeitung von Bedeutung ist.

Das erste Beispiel zeigt, dass mit heurisko Volumendaten mit echten 3-D-Operatoren sehr schnell verarbeitet werden können (Abschnitt 6.2). Das zweite Anwendungsbeispiel beschreibt, wie man mit heurisko sehr lange Bildsequenzen in Echtzeit erfassen und speichern kann (Abschnitt 6.3). Eine Anwendung zur Restauration von Bildfolgen demonstriert, wie mit heurisko auch eine komplexe Anwendung beherrschbar ist (Abschnitt 6.4). Das vierte Projekt stellt zwei Module vor, welche die gleiche Codebasis wie heurisko haben, aber Teil eines anderen auf dem Markt erhältlichen Bildverarbeitungssystems sind (Abschnitt 6.5). Im letzten Beispiel geht es um Bildverarbeitung bei der Kameraherstellung (Abschnitt 1). Zuvor gibt Abschnitt 6.1 eine Übersicht über Architekturformen von Bildverarbeitungssystemen mit heurisko.

## 6.1 Architekturformen eines Systems mit heurisko

Das Paket heurisko lässt sich auf verschiedene Weise in Bildverarbeitungssysteme einbauen. Hier werden drei Grundformen der üblichen Architekturen, die sich so oder ähnlich in den einzelnen Projekten wieder finden, vorgestellt. In den folgenden Abbildungen werden Programmblöcke durch Kästen dargestellt, die sechs verschiedene, den Blockinhalt beschreibende Titelzeilen haben können. Der Titel *Steuerprogramm* bedeutet, dass dieser Block die Programmmodule umfasst, mit denen das Bildverarbeitungssystem gesteuert wird. Dieser Block kann eine, muss aber keine graphische Benutzerschnittstelle enthalten. Ein Block mit dem Titel Bildverarbeitung enthält den Interpreter und den Kern von heurisko. Der Kern stützt sich auf Blöcke, die Erweiterungsmodule zu heurisko symbolisieren. Es werden allgemeine Module mit *Operatoren* und spezielle für *Akquisition* und *Visualisierung* unterschieden. Im Inneren eines Blocks ist noch angegeben, ob er aus Standardmodulen von *heurisko* besteht oder aus *proprietären*, anwendungsspezifischen Modulen. Da hier nur Systeme mit heurisko beschrieben werden, besteht der Block *Bildverarbeitung* immer aus heurisko-Modulen. Zur Verdeutlichung, dass eine ganze Reihe von Operatoren unabhängig vom Steuerprogramm mit Dialogen verknüpft ist, ist der Block mit dem Titel *Dialoge* angegeben. Schließlich wird mit dem Kasten *Skript* angedeutet, dass zu einer Anwendung mit heurisko auch die Skripte gehören. Für das Folgende ist es angebracht, sich noch einmal Abbildung 4.1 und die zugehörige Beschreibung in Abschnitt 4.1 in Erinnerung zu rufen.

Abbildung 6.1 zeigt ein ausschließlich mit heurisko realisiertes System und ist im Grunde eine Vereinfachung der in Abbildung 4.1 abgebildeten modularen Struktur heuriskos. Solch eine Architektur ist immer dann möglich, wenn die Funktionalität in heurisko für die Anwendung ausreicht, die Anwendung nicht in ein übergeordnetes System eingebunden werden

muss und keine anwendungsspezifische Benutzeroberfläche benötigt wird. Damit ist ein solches System typisch für labormäßigen Betrieb und natürlich für Entwickler von Bildverarbeitungsanwendungen.

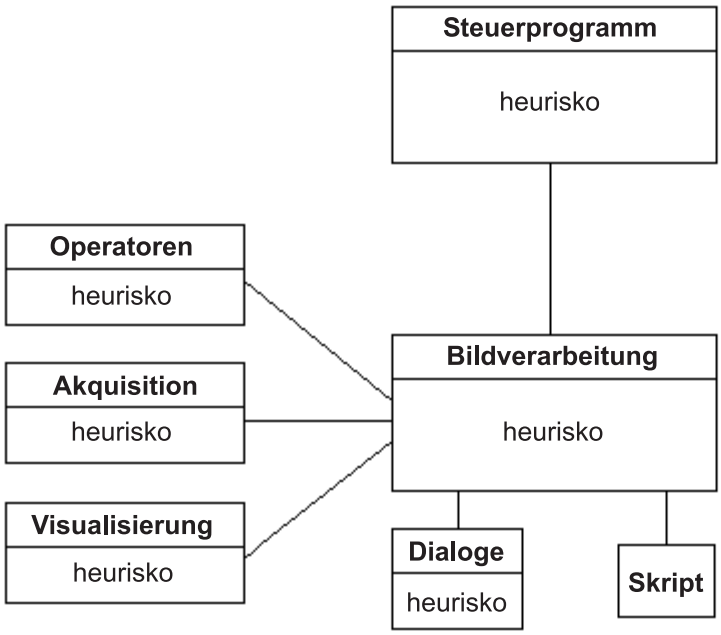


***Abbildung 6.1:*** *Architektur eines Bildverarbeitungssystems ausschließlich mit heurisko*

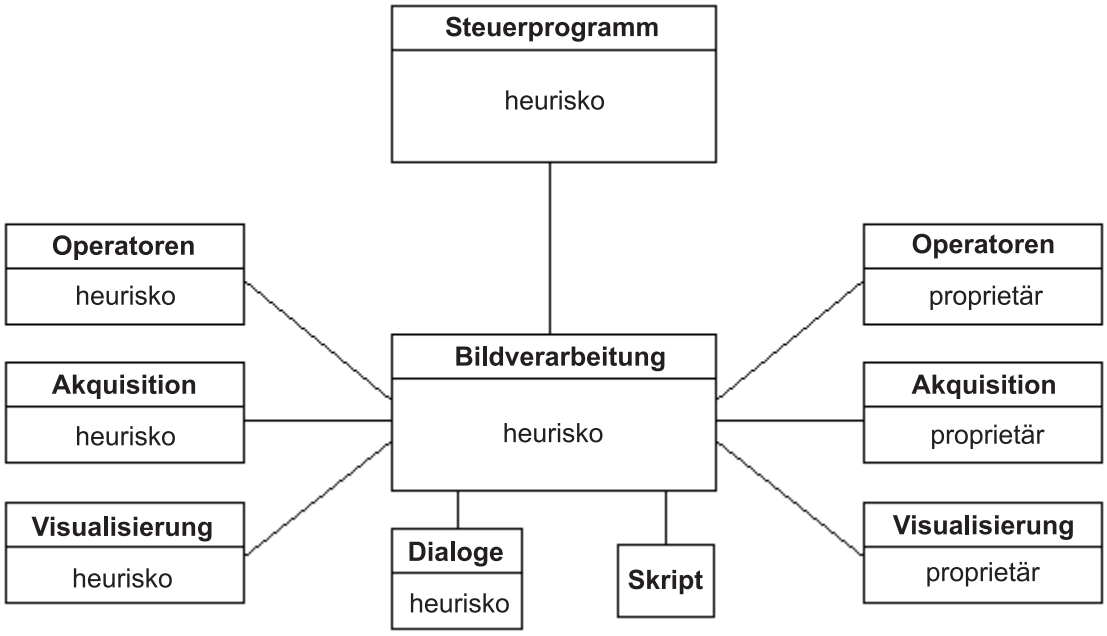


***Abbildung 6.2:*** *Architektur eines Bildverarbeitungssystems mit heurisko und proprietären Erweiterungen*

Die Architektur in Abbildung 6.2 zeigt ein System wie in Abbildung 6.1, das jedoch auf der rechten Seite um proprietäre Erweiterungsmodule ergänzt wurde. Obwohl alle drei verschiedenen Erweiterungsmodultypen gezeigt werden, soll diese Abbildung auch für alle Systeme stehen, die wenigstens ein Erweiterungsmodul enthalten. Da hier wieder das Originalsteuerprogramm von heurisko zum Einsatz kommt, ist auch diese Architektur typisch für Laborbetrieb und Anwendungsentwickler.

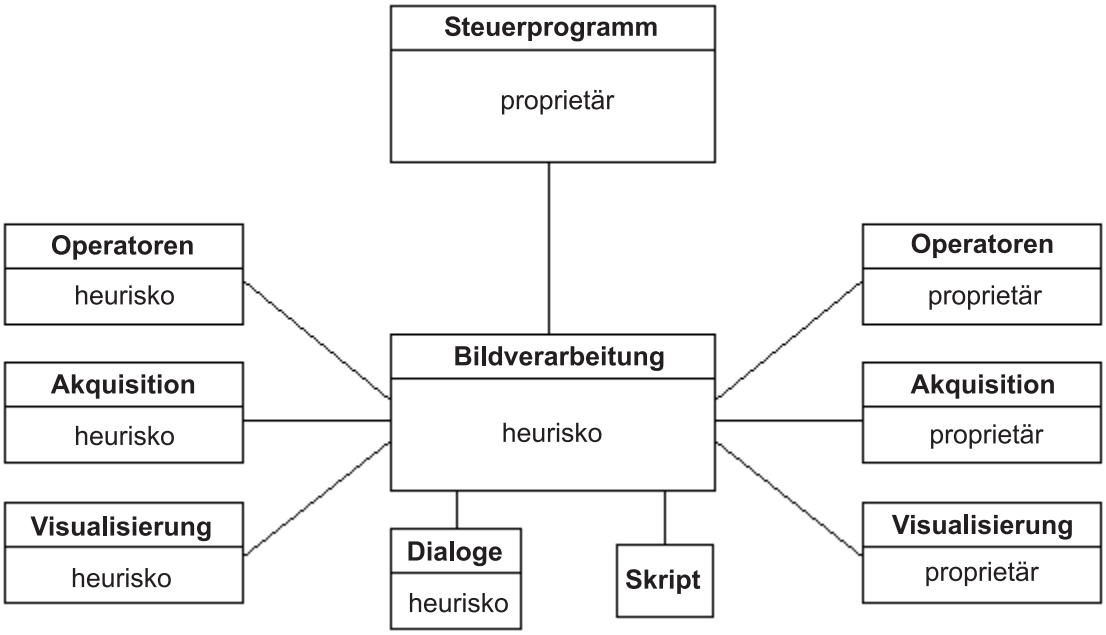


***Abbildung 6.3:*** *Architektur eines Bildverarbeitungssystems mit heurisko und proprietären Erweiterungen und proprietärem Steuerprogramm*

Die dritte Architekturgrundform ist in Abbildung 6.3 zu sehen. Hier wird das Bildverarbeitungssystem von einem proprietären Programm gesteuert. Das System kann, muss aber keine proprietären Erweiterungsmodule enthalten. Es ist typisch für industrielle Anwendungen oder für Anwendungen, die aus verschiedenen Gründen mit einer spezifischen Benutzeroberfläche ausgestattet sein sollen. Die mit einer ganzen Reihe von Operatoren verknüpften interaktiven Elemente wie z. B. Auswahldialoge können unabhängig vom Steuerprogramm verwendet werden, weil sie außerhalb des heurisko-eigenen Steuerprogramms in einem separaten heurisko-Modul implementiert sind.

## 6.2 Anwendungsbeispiel: Optimierungen für 3-D

### 6.2.1 Aufgabenstellung

In einer Firma werden CT-Volumenbilder von Spritzgussteilen auf Lunker und Risse untersucht [Eisele 2002]. Zur Version 4 von heurisko erstellt der Anwender ein eigenes Erweiterungsmodul mit einigen speziellen 3-D-Operatoren. Die Algorithmen sind jedoch so langsam, dass die Einsatzfähigkeit der Bildverarbeitungslösung nicht gegeben ist. Gefordert wird deshalb eine drastische Beschleunigung. Das in einer konkreten Anwendung zu verarbeitende Volumenbild ist durch

```
ubyte obj[449][511][511];
```

definiert und belegt knapp 112 MB Speicher. Es geht um folgende drei Operatoren des Erweiterungsmoduls:

1. `OptConv_1D:` Glättung mit der gleichen 1-D-Maske in jeder Richtung; spezielles Interesse an der Maske $\frac{1}{16}[1 \quad 4 \quad 6 \quad 4 \quad 1]$; Ein- und Ausgabeobjekt vom Typ `ubyte`

2. `Gradient3D:` Betrag des Grauwertgradienten mit frei wählbarer Faltungsmaske; spezielles Interesse an symmetrischer Differenz und Sobelfilter; Ein- und Ausgabeobjekt vom Typ `ubyte`

3. `GrowWhiteRegion:` Wachstum binärer Regionen; Ein- und Ausgabeobjekt sind identisch und vom Typ `binary`; ein Parameter gibt an, wie oft die Operation hintereinander auszuführen ist

Die Laufzeiten dieser Operatoren sind in Tabelle 6-1 angegeben. Die Messungen liegen schon etwas zurück und wurden deshalb nicht wie sonst in dieser Arbeit auf dem Testrechner durchgeführt. Die Zeiten zeigen deutlich, dass eine korrekte Implementierung gerade in der 3-D-Bildverarbeitung nicht ausreicht. Eine geringe Effizienz wird mit inakzeptablen Laufzeiten bestraft.

***Tabelle 6-1:*** *Laufzeiten dreier 3-D-Operatoren aus einem Erweiterungsmodul eines heurisko-Anwenders für Objekte der Größe 511×511×499 vom Typ* `ubyte` *bzw.* `binary` *auf den angegebenen Rechnern*

| Operator | Pentium III, 450 MHz, 1 GB RAM | Pentium 4, 1,4 GHz, 1 GB RAM |
|---|---|---|
| `OptConv_1D` | 237 s | 178 s |
| `Gradient3D` | 84 s | 36 s |
| `GrowWhiteRegion` | 123 s | 48 s |

Zusätzlich zur Laufzeitverbesserung der drei beschriebenen Operatoren wird eine schnelle Merkmalsberechnung für segmentierte Objekte benötigt. Als Merkmale sind statistische Werte, Fläche und Schwerpunkt, Momententensor und dessen Eigenwerte und Eigenvektoren gefragt. Eine wichtige Nebenbedingung ist die Vermeidung temporärer Objekte. Ein komplettes 3-D-Label-Objekt vom Typ `ushort`, benötigt für die Markierung der Bildpunktzugehörigkeit zu den segmentierten Objekten während der Merkmalsberechnung, würde 224 MB belegen, ein Gradient in nur *einer* Richtung vom Typ `float` gar 448 MB und die Gradienten in allen drei Richtungen vom Typ `ubyte` zusammen immerhin 336 MB.

### 6.2.2 Realisierung

Es wird auf Version 5 von heurisko übergegangen. Alle erforderlichen Operatoren können der Standardversion entnommen werden und sind sehr schnell. Deshalb erfordert die Aufgabenstellung des Kunden keine separaten Implementierungen in einem kundenspezifischen Erweiterungsmodul. Die Architektur der Lösung entspricht damit Abbildung 6.1, solange kein proprietäres Steuerprogramm, eventuell mit problemspezifischer Benutzeroberfläche, benötigt wird. Die Anforderungen werden wie folgt erfüllt:

**1. Glättung:** Die Glättung mit der gewünschten Maske wird vom Operator `Bin4()` geleistet.

**2. Betrag des Gradienten:** Es stehen in heurisko drei Operatoren für Ableitungen erster Ordnung zur Verfügung, die unter anderem den Modus `magnitude` oder kürzer `m` aufweisen, welcher den gewünschten Betrag des Gradienten berechnet. `D1_3()` ist ein symmetrischer Differenzenoperator, der die Ableitung mit der Maske $\frac{1}{2}[1 \quad 0 \quad -1]$ in jeder Richtung ermittelt. `Sobel3()` ist der Standard-Sobeloperator und `SobelD3()` eine Variante mit einer Minimierung des Fehlers bezüglich der Richtung. Die Masken dieser Operatoren sind für die x-Richtung im 2-D-Fall

$$\frac{1}{8}\begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix} \text{ bzw. } \frac{1}{32}\begin{bmatrix}3 & 0 & -3\\ 10 & 0 & -10\\ 3 & 0 & -3\end{bmatrix}$$

und müssen entsprechend in die dritte Dimension ausgedehnt gedacht und analog in die übrigen Richtungen angewendet werden. Allen Operatoren ist gemeinsam, dass der Betrag sofort berechnet wird, so dass die Gradienten nicht in temporären Objekten aufgehoben werden müssen.

**3. Regionenwachstum:** Das Regionenwachstum wird durch eine Dilatation mit dem Operator `Dilate3N()` vorgenommen. Es handelt sich dabei um eine Faltung mit den beiden Modi `corner` und `edge`, zu denen im 2-D-Fall die Maske

$$\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \text{ bzw. } \begin{bmatrix}0 & 1 & 0\\ 1 & 1 & 1\\ 0 & 1 & 0\end{bmatrix}$$

gehört. Die Erweiterung in die dritte Dimension ist offensichtlich. Die gewünschte mehrmalige Anwendung kann effizient in einer Schleife mit dem Steuerkommando `repeat` gemäß

```
repeat(n); obj = Dilate3N(); endrepeat;
```

realisiert werden.

**4. Merkmalsextraktion:** Für die Merkmalsextraktion steht eine ganze Reihe von Operatoren zur Verfügung. Je nachdem, an welchen Merkmalen man interessiert ist, kann man einen oder mehrere dieser Operatoren wählen. `ObjCnt()` zählt die Objekte, `ObjArea()` berechnet deren Flächen, `ObjMoment.1()` berechnet Momente erster Ordnung, `ObjMoment.2()` Momente zweiter Ordnung, `ObjStatistics()` statistische Werte und `ObjPosMoment()` Schwerpunkte, Momente zweiter Ordnung als Tensor und dessen Eigenwerte und Eigenvektoren. Alle Operatoren benötigen kein separates Label-Objekt.

***Tabelle 6-2:*** *Vergleich der Laufzeiten der angegebenen Operatoren für Objekte der Größe 511×511×499 vom Typ* `ubyte` *bzw.* `binary` *auf einem Pentium 4 mit 1,4 GHz*

| Operation | alter Operator | t1 [s] | neuer Operator | t2 [s] | Steigerung t1/t2 |
|---|---|---|---|---|---|
| Glättung | `OptConv_1D` | 178 | `Bin4` | 1,6 | 111 |
| Betrag des Gradienten | `Gradient3D` | 36 | `D1_3.m` | 3,4 | 11 |
| | | | `Sobel3.m` | 3,8 | 10 |
| | | | `Sobel3D.m` | 4,4 | 8 |
| Dilatation | `GrowWhiteRegion` | 48 | `Dilate.corner` | 0,7 | 69 |
| Merkmalsextraktion für 10 segmentierte Objekte | kein 3-D-Labeln in heurisko 4 | – | `ObjCnt` | 0,05 | – |
| | | | `ObjArea` | 0,1 | – |
| | | | `ObjMoment.1` | 0,1 | – |
| | | | `ObjMoment.2` | 0,1 | – |

In Tabelle 6-2 sind die Laufzeiten der neuen Operatoren in heurisko 5 im Vergleich zu denen im Erweiterungsmodul des Kunden angegeben. Die erreichten Geschwindigkeitssteigerungen

zeigen deutlich, wie viel Optimierungspotential vorhanden war. Für Faltungsoperationen, zu denen die Binomialglättung, die Gradientenberechnung und die Dilatation gehören, wurde bereits in Abschnitt 4.3.4 ausführlich diskutiert, wie effizient sie in heurisko implementiert sind. Die Erläuterungen zum Operator `Bin2()` im gleichen Abschnitt gelten analog auch für `Bin4()`, denn der Unterschied zwischen beiden ist nur die Faltungsmaske. Bezüglich der Geschwindigkeit profitieren die benötigten Operatoren gerade bei den verwendeten Datentypen `ubyte` und `binary` zusätzlich von der SIMD-Programmierung (siehe Abschnitt 5.2.1).

Der Betrag des Gradienten vom dreidimensionalen Grauwertbild $g(x,y,z)$,

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

erfordert die Zwischenspeicherung der partiellen Ableitungen. Für den Operator ist jedoch ein ebenenweises Vorgehen in z-Richtung implementiert. Nur die Ableitungen einer Bildebene müssen in einem temporären 2-D-Objekt aufgehoben werden. Für dieses 2-D-Objekt erfolgt dann vektorweise das Quadrieren, die Addition und die Wurzelberechnung zusammen in einer Vektorfunktion. Das vergleichsweise langwierige Ziehen der Quadratwurzel wird für den Datentyp `ubyte` mit Hilfe einer Look-up-Tabelle (LUT) noch zusätzlich beschleunigt. Für die LUT müssen einmalig nur 256 Wurzeln berechnet werden anstatt 117.243.329 Wurzeln für die 511×511×449 Bildpunkte des Beispielobjekts. Die Wurzelberechnung per LUT ist sehr schnell, da die Wurzel aus *i* im *i*-ten Eintrag der LUT steht.

Für die 3-D-Merkmalsextraktion existiert kein Vergleich mit einer Implementierung des Kunden oder in heurisko 4. Daher sei das implementierte Verfahren hier mit Standardverfahren verglichen. Die Extraktion von Objekten aus einem Grauwertbild geschieht mit Hilfe irgendeines passenden Verfahrens. Das Ergebnis ist ein Binärbild, in dem eine 1 anzeigt, dass der zugehörige Bildpunkt zu einem extrahierten Objekt gehört, und eine 0, dass der Bildpunkt Teil des Hintergrundes ist. Der nächste Schritt ist die Identifizierung der Objekte durch das Auffinden zusammenhängender 1-Regionen. Als Resultat erhält jeder 1-Bildpunkt eine Nummer derart, dass alle zu einer zusammenhängenden Region gehörenden Punkte die gleiche Nummer, die Objektnummer, aufweisen. Deshalb wird dieser Vorgang *Labeln* genannt. Üblicherweise wird dafür ein separates Label-Bild benutzt ([Pitas 1993] und [Jain et al. 1995]). Dieses Vorgehen ist bei 3-D-Daten allerdings problematisch, da ein komplettes weiteres 3D-Bild benötigt wird, das typischerweise vom Datentyp `ushort` ist, um die maximale Anzahl von Objekten nicht auf 255 zu beschränken. Bei der gegebenen Anwendung wären dies zusätzliche 224 MB.

Daher wurde nach einem Verfahren gesucht, das die Berechnung eines Labelbildes vermeidet. Der Trick besteht darin, Labeln und Merkmalsberechnung in einem Schritt durchzuführen und dabei wie gewohnt zeilenweise vorzugehen. Dazu wird zuerst die Vektorfunktion zur Erkennung der zusammenhängenden Objektbereiche aufgerufen. Wenn ein Objektpunkt (eine 1) in der Zeile auftaucht, wird zuerst geprüft, ob er mit einem Objekt in der vorangegangenen Zeile oder Ebene zusammenhängt. Ist dies der Fall, erhält er die Nummer dieses Objekts zugeordnet. Andernfalls wird eine neue Objektnummer vergeben. Für alle weiteren mit dem ersten Objektpunkt zusammenhängenden Pixel wird die gleiche Objektnummer vergeben. Dabei kann es allerdings vorkommen, dass ein Pixel mit einem Objekt zusammenhängt, das eine andere Nummer hat, da die Objektpunkte in der vorherigen Zeile getrennt vorlagen und erst in der aktuell untersuchten Zeile zu einem Objekt verschmelzen. Diese Äquivalenz muss in einer Äquivalenzliste gemerkt werden. Trotzdem kann man nach Abarbeiten der Zeile sofort die Merkmale der vorläufigen Objekte berechnen. Für die Fläche ist z. B. einfach die Menge der zu einer Nummer gehörenden Bildpunkte zu zählen. Nach Abarbeitung des gesamten Volu-

menbildes muss noch anhand der erstellten Äquivalenzliste die tatsächliche Zahl der Objekte berechnet und die Nummerierung korrigiert werden. Weiterhin müssen die für die vorläufigen Objektteile berechneten Merkmale zusammenfügt werden. Bei der Objektfläche ist dies eine einfache Addition. Aber auch für die anderen betrachteten Merkmale wie Objektschwerpunkt und Momente sind die Algorithmen zum Zusammenfügen nur unwesentlich komplizierter.

Eine weitere Beschleunigung konnte dadurch erreicht werden, dass alle Berechnungen in einer Lauflängencodierung durchgeführt wurden. Es hat sich gezeigt, dass die Beschleunigung der Berechnung der Merkmale im Lauflängencode deutlich größer ist als der zusätzliche Aufwand zur Lauflängenkodierung.

## 6.3 Anwendungsbeispiel: Echtzeitakquisition

### 6.3.1 Aufgabenstellung

Im Abschnitt 4.5.2 über das Erweiterungsmodul `hk_ac` für Datenerfassung mit heurisko wurde mit kleinen Beispielen demonstriert, wie Akquisition und Bildverarbeitung parallel geschehen können. In diesem Abschnitt soll ein ausführlicheres Beispielskript zeigen, wie mit heurisko auf einem PC eine lange Bildsequenz in Echtzeit unkomprimiert erfasst werden kann. Unter einer *langen* Bildsequenz ist hier zu verstehen, dass die Sequenz nicht mehr im Hauptspeicher Platz hat. Wegen der geforderten Echtzeit ist es daher notwendig, kontinuierlich bereits im Speicher liegende Bilder auf einen Massenspeicher zu transportieren und den zugehörigen Hauptspeicher wieder für die weitere Akquisition frei zu machen. Dies ist keine triviale Aufgabe, da die Bandbreite des PCI-Bus vom Datenerfassungsgerät in den Hauptspeicher größer ist als die Bandbreite für das Speichern auf einem Festplattensystem. Lange Bildsequenzen spielen bei wissenschaftlichen Experimenten und bei der Störfallursachenforschung in technischen Prozessen eine wichtige Rolle, um nur zwei Anwendungsmöglichkeiten zu nennen. Die Echtzeitdatenerfassung ist ein schönes Beispiel dafür, dass der Interpreter in heurisko genügend schnell ist, so dass auf eine komplexe Standardsprachen-Programmierung ohne Einbußen an Flexibilität und Kontrolle der Synchronisation verzichtet werden kann.

### 6.3.2 Realisierung

Für die Akquisition wird ein Puffer mit *n* Bildern angelegt. Dieser Puffer wird gedanklich in *m* gleich große Bereiche unterteilt. Dann startet eine zyklische Datenerfassung im Hintergrund, die unbeirrt läuft, bis sie explizit beendet wird. Wenn während eines Zyklus die Akquisition in den *i*-ten Bereich abgeschlossen ist, wird dieser Bereich im Rohformat in einer Datei auf der Festplatte gespeichert. Parallel dazu läuft die Akquisition in den Bereich *i*+1 weiter. Ist sie abgeschlossen, wird auch dieser Bereich in einer weiteren Datei gespeichert. Nach dem *n*-ten Bereich beginnt der Zyklus wieder mit dem ersten Bereich, und zwar so lange, bis ein Abbruchkriterium erreicht wird. Für die unten vorgestellten Operatoren seien folgende Definitionen in einem Skript vorgenommen worden:

```
# Definitionen für Echtzeitbilderfassung und –speicherung

# Akquisitionspuffer:
struct buffer {
    string name,
    long dim,
    long size[3],
    long offs[3],
    string format
```

```
};
buffer = "buf".{3}.{-1,-1,50}.{0,0}." %3i"; # 50 Bilder

# Akquisitionsobjekt für Framegrabber ELTEC p3i2:
device fg, type "pceye2p", config "ac_p3i2_mono.dat",
                           buffers buffer;

# weitere globale Objekte:
ubyte buf = fg.buf;     # Alias für buffer
ulong cnt, setcnt, pos; # Zähler
```

Der Puffer enthält also *n*=50 Bilder. Der folgende Operator zeigt für den Fall *m*=2 das geschilderte Verfahren. Die Parameter des Operators sind der Name der ersten Datei und die Anzahl der Schleifendurchläufe.

```
# Echtzeitbilderfassung und –speicherung

operator rtr1(fname, r);
    pos = Clr();
    acStart(buf); # zyklische Hintergrundakquisition starten
    repeat (r);   # 2r-mal 25 Bilder speichern
        while (pos != 24); pos = acPosition(buf); endwhile;
        time = GetTime.second(); time; # Protokoll
        Write(fname, buf[0:25]);       # erster Teilbereich
        fname = Inc(); # Nummer im Dateinamen hochzählen

        while (pos != 49); pos = acPosition(buf); endwhile;
        time = GetTime.second(); time; # Protokoll
        Write(fname, buf[25:25]);      # zweiter Teilbereich
        fname = Inc(); # Nummer im Dateinamen hochzählen
    endrepeat;
    acStop(buf);
endoperator;
```

Der Dateiname muss für das Funktionieren dieses Operators eine Nummer enthalten, die automatisch inkrementiert wird, so dass Dateien mit fortlaufenden Nummern entstehen, die nachher wieder automatisch verarbeitet werden können. Mit dem Befehl

```
rtr1("rt1_0001.raw", 100);
```

wird die Schleife 100-mal durchlaufen. Nach fehlerlosem Lauf befinden sich in dem Verzeichnis, das sich, weil kein absoluter Pfad angegeben wurde, aus dem in heurisko eingestellten Standardspeicherpfad ergibt, 200 Dateien mit Namen `rt1_0001.raw` bis `rt1_0200.raw`. Der Befehl `fname = Inc()` sorgt dafür, dass die in der `string`-Variablen `fname` gefundene Zahl inkrementiert wird.

Der Code in der `repeat`-Schleife besteht aus zwei nahezu identischen Abschnitten. Der Unterschied ist lediglich, dass im ersten Abschnitt der erste Teilbereich des Puffers und im zweiten Abschnitt der zweite Teilbereich gespeichert und der zugehörige Zählerstand abgewartet wird. Entsprechend wird in der ersten `while`-Schleife auf den Zählerstand 24 gewartet, in der zweiten dagegen auf 49. Denn `pos` als Rückgabewert von `acPosition()` gibt an, für welches letzte Bild im Puffer die Akquisition bereits abgeschlossen wurde. Problematisch an dieser Lösung könnte sein, dass nicht bemerkt wird, wenn das Schreiben auf die Festplatte oder eine andere vorübergehende Unterbrechung des Programms einmal zu lange dauert, so dass die Akquisition bereits über den Zählerstand, auf den gewartet werden soll,

hinaus vorangeschritten ist. Das würde bedeuten, dass erst bei Erreichen des erwarteten Zählerstandes im nächsten Akquisitionszyklus wieder gespeichert wird. Auf der Festplatte fehlt dann ein Teil der Sequenz.

Man kann dieses Risiko vermindern, indem man dafür sorgt, dass die oben erwähnten Parameter *m* und *n* günstig gewählt sind und dass das System nicht an der Grenze seiner dauerhaft möglichen (durchschnittlichen) Schreibrate arbeitet. Außerdem kann ein Raid-Festplattensystem eingesetzt werden, dass eine höhere Schreibrate garantiert. Darüber hinaus ist es sicher sinnvoll zu überprüfen, ob die Speicherung trotz aller anderen Maßnahmen nicht doch außer Takt gerät. Darauf kann man dann reagieren, beispielsweise mit einem Hinweis im Protokoll oder sogar mit Abbruch. Man kann allerdings auch noch ein wenig flexibler vorgehen. Wenn man annimmt, dass eine Störung nur vorübergehend und ausreichend kurz ist, ist es wahrscheinlich, dass sich das System wieder fängt. Die Flexibilität kann man dadurch erzielen, dass man nicht genau den Zählerstand 24 oder 49 verlangt, sondern nur, dass mindestens dieser Stand erreicht wurde. Als einen Fehler kann man dann ansehen, wenn der erwartete Zählerstand zu weit überschritten wurde. Der folgende Operator enthält die dazu nötigen Änderungen und Ergänzungen, wobei eine Zählerüberschreitung von 5 akzeptiert wird. Da das Schreiben von Dateien eine asynchrone Operation des Betriebssystems ist, so dass der Operator `Write()` schnell zurückkehrt, muss man gegenüber der ersten Lösung noch zusätzliche Abfragen beim Warten auf einen Zählerstand vornehmen.

```
# Echtzeitbilderfassung und –speicherung mit Überprüfung

operator rtr1(fname, r);
    ulong diff;
    pos = Clr(); setcnt = Clr();
    acStart(buf); # zyklische Hintergrundakquisition starten
    repeat (r);   # 2r-mal 25 Bilder speichern
        cnt = acCnt(buf);
        diff = Sub(cnt, setcnt); diff;
        if (diff > 5); "Warning!"; endif;
        pos = acPosition(buf);
        while (pos == 49); pos = acPosition(buf); endwhile;
        while (pos < 24);  pos = acPosition(buf); endwhile;
        time = GetTime.second(); time; # Protokoll
        Write(fname, buf[0:25]);       # erster Teilbereich
        fname = Inc(); # Nummer im Dateinamen hochzählen
        setcnt = Add(25);

        cnt = acCnt(buf);
        diff = Sub(cnt, setcnt); diff;
        if (diff > 5); "Warning!"; endif;
        pos = acPosition(buf);
        while (pos == 24); pos = acPosition(buf); endwhile;
        while (pos < 49);  pos = acPosition(buf); endwhile;
        time = GetTime.second(); time; # Protokoll
        Write(fname, buf[25:25]);      # zweiter Teilbereich
        fname = Inc(); # Nummer im Dateinamen hochzählen
        setcnt = Add(25);
    endrepeat;
    acStop(buf);
endoperator;
```

## 6.4 Anwendungsbeispiel: Restauration von Bildfolgen

### 6.4.1 Aufgabenstellung

Ein Kunde benötigt ein komplexes Bildanalysewerkzeug zur Restauration von Bildfolgen. Da das Bildmaterial alle Arten von Störungen aufweisen kann und der Bildinhalt nicht im Voraus bekannt ist, muss die Bildanalyse in labormäßigem Stil durchgeführt werden. Der Benutzer ist bei der Analyse wesentlich beteiligt, muss das konkrete Vorgehen aus vielen Möglichkeiten auswählen und vielfach mit mehreren Varianten experimentieren. Äußerst wichtig ist für den Kunden, dass einmal gefundene Lösungswege so abgelegt werden, dass sie jederzeit wieder nachvollziehbar sind.

Für ein Bildanalysewerkzeug, das so verschiedene Teilaufgaben wie Bildverbesserung, Bildvorverarbeitung, Bewegungsanalyse und Restaurierung zusammenfasst, ist es entscheidend, eine einfache und konsistente Benutzeroberfläche zu entwickeln. Ansonsten lassen sich die vielen Möglichkeiten der entwickelten Analysemethoden nicht effizient in der Praxis benutzen. Deswegen soll ein gemeinsames Konzept zur Bedienung der verschiedenen Bildverarbeitungsaufgaben entwickelt werden.

### 6.4.2 Realisierung

Schon die Algorithmikentwicklung dieser Aufgabe ist sehr aufwändig, so dass nicht auch noch eine große Summe in eine proprietäre Benutzeroberfläche investiert werden kann, zumal nur geschulte Anwender das System nutzen sollen. Die Anwendung besteht daher ausschließlich aus heurisko-Elementen und entspricht damit in seiner Architektur Abbildung 6.1.

Insgesamt umfasst die fertige Anwendung über 5000 Zeilen heurisko-Code und Hunderte von Operatordefinitionen, von denen rund 100 vom Benutzer direkt aufgerufen werden sollen. Um diese Komplexität beherrschen zu können, bedarf es einiger Ordnungsmöglichkeiten.

- Jeder heurisko-Operator ist einer Operatorgruppe zugeordnet, wobei in einem Skript beliebig viele neue Operatorgruppen definiert werden können. Eine interaktive Operatorauswahl per Menü besteht somit aus zwei Stufen. Zuerst wählt man aus der Liste aller Operatorgruppen eine Gruppe aus und dann aus dieser den gewünschten Operator. Teilt man die für eine Anwendung benötigten Operatoren in wenige Gruppen sinnvoll auf, erleichtert das die Suche des richtigen Operators deutlich.

- Eine Option der Benutzeroberfläche erlaubt das Verstecken aller in heurisko eingebauten Operatoren und Operatorgruppen. Das Operatormenü enthält dann nur noch die in einem Skript definierten Operatoren. Weiter können im Skript definierte Operatoren versteckt werden, indem man ihrem Namen einen Unterstrich voranstellt. Wenn folglich im Operatormenü dann nur noch die anwendungsspezifischen Gruppen mit den direkt benötigten Operatoren erscheinen, ist das sehr übersichtlich.

- Auch der Entwickler eines Skriptes kann seinen Code über die Operatorgruppen und Operatoren hinaus ordnen. Es ist nämlich möglich, innerhalb eines Skriptes andere Skripte zu importieren. Hiervon profitiert auch der Benutzer. Im vorliegenden Fall besteht das System aus vier Skripten. Eines von Ihnen stellt grundlegende Operatoren zum Einlesen, zum Speichern, zur interaktiven Darstellung und zur Verbesserung und Vorverarbeitung von Bildsequenzen zur Verfügung, die in allen anderen Skripten benötigt werden. Möchte sich der Benutzer manches Mal auf diese Funktionalität beschränken, braucht er auch nur dieses Skript zu laden und erhält dann eine noch übersichtlichere Anwendung. Genauso kann jedes der anderen drei Skripte als Aus-

gangspunkt gewählt werden, wobei dann gegebenenfalls andere benötigte Skripte selbsttätig importiert werden.

Zu einer Anwendung mit bedeutendem Anteil an interaktiver Beteiligung des Anwenders gehören Elemente, über die der Benutzer mit dem Programm kommunizieren kann. Üblicherweise werden Programme mit einer komfortablen programmspezifischen graphischen Benutzeroberfläche ausgestattet. Nicht selten ist der Aufwand für die Erstellung dieser Benutzerschnittstelle ebenso groß wie der für die Algorithmik zur Datenverarbeitung. Solch einen Aufwand kann man sich aber nicht immer leisten, insbesondere dann nicht, wenn die Benutzeroberfläche komplex ist und der Anwenderkreis klein. Zur schnellen, einfachen und flexiblen Erstellung von Kommunikationselementen bietet heurisko eine ganze Reihe von Operatoren, die allgemein verwendbare Dialoge anzeigen und eventuelle Rückgabewerte im Skript verfügbar machen (siehe Tabelle 6-3).

***Tabelle 6-3:*** *Operatoren in heurisko für Dialoge mit dem Benutzer*

| Operator | Beschreibung |
|---|---|
| `Pause` (1. Variante) | Meldebox mit OK-Schaltfläche |
| `Pause` (2. Variante) | Meldebox mit Schaltflächen für *Ja*, *Nein* und *Abbrechen* |
| `RequestDir` | Dialog zur Auswahl eines existierenden Verzeichnisses |
| `RequestFiles` | Dialog zur Auswahl existierender Dateien |
| `RequestNewFiles` | Dialog zur Auswahl eines Dateinamens |
| `Input.float` | Dialog zur Eingabe von *n* Gleitkommazahlen |
| `Input.integer` | Dialog zur Eingabe von *n* ganzen Zahlen |
| `Input.string` | Dialog zur Eingabe von *n* einzeiligen Texten |
| `Input.text` | Dialog zur Eingabe von *n* mehrzeiligen Texten |
| `Input.select` | Dialog mit 1-aus-*n*-Auswahl per Radioknöpfe |
| `Input.combobox` | Dialog mit 1-aus-*n*-Auswahl per aufklappbarer Liste |
| `Input.check` | Dialog mit *m*-aus-*n*-Auswahl |
| `OpenSlider` | *n* Regler mit je einem oder zwei einstellbaren Schiebern und sofortiger Rückkehr nach jeder Betätigung |
| `OpenInput` | *n* Regler mit je einem oder zwei einstellbaren Schiebern und Rückkehr erst nach Bestätigung durch *OK* |

Im Folgenden wird das prinzipielle Vorgehen beim Arbeiten mit der Anwendung skizziert. Vorausschickend sei festgestellt, dass Bildsequenzen im TIFF- oder JPEG-Format vorliegen müssen, wobei jedes Bild in einer eigenen Datei gespeichert ist. Die Dateien einer Sequenz müssen den gleichen Namensstamm und eine aufsteigende Nummerierung als weiteren Namensbestandteil haben. Solche Sequenzen können z. B. mit heurisko-Operatoren in einer anderen Anwendung leicht erzeugt werden.

Für den grundlegenden Umgang mit einer Bildsequenz stehen in dem Skript `sinitial.ws` die in Abbildung 6.4 zu sehenden Operatorgruppen zur Verfügung. Die Gruppe `SequenzIn` enthält unter anderem Operatoren zum Laden und mehrere Möglichkeiten zum Ansehen der Eingangsbildsequenz. Die Eingangsbildsequenz ist immer im mit `fb1` bezeichneten Fenster zu sehen (siehe Abbildung 6.5). Jede Bearbeitung der Eingangssequenz lässt diese unberührt und speichert das Ergebnis in der Ausgangsbildsequenz, die immer im Fenster `fb2` dargestellt ist (siehe ebenfalls Abbildung 6.5). Auch für die Ausgangsbildsequenz steht eine eigene Operatorgruppe zur Verfügung. Für Operationen mit beiden Sequenzen gleichzeitig hält die Gruppe `SequenzInOut` Operatoren bereit, beispielsweise zum gemeinsamen Abspielen

oder um die Ausgangsbildsequenz zur neuen Eingangsbildsequenz zu machen. Selbstverständlich können beide Sequenzen auch wieder gespeichert werden. Für manche Operationen werden Parameter benötigt, die vom Benutzer über Dialoge wie den in Abbildung 6.7 oder über graphische Eingaben per Maus wie in Abbildung 6.6 abgefragt werden. Die anderen Operatorgruppen dienen zur Bildverbesserung und zur Vorverarbeitung.

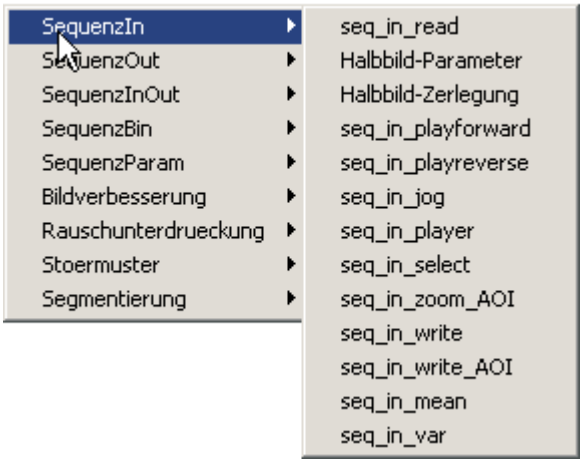


***Abbildung 6.4:*** *Operatorgruppen in* `sinitial.ws` *mit Operatoren in der Gruppe* `SequenzIn`


***Abbildung 6.5:*** *Dialog mit Regler zum Ansehen der Eingangsbildsequenz links; rechts in der Ausgangsbildsequenz ein vergrößerter Ausschnitt (heurisko 4)*

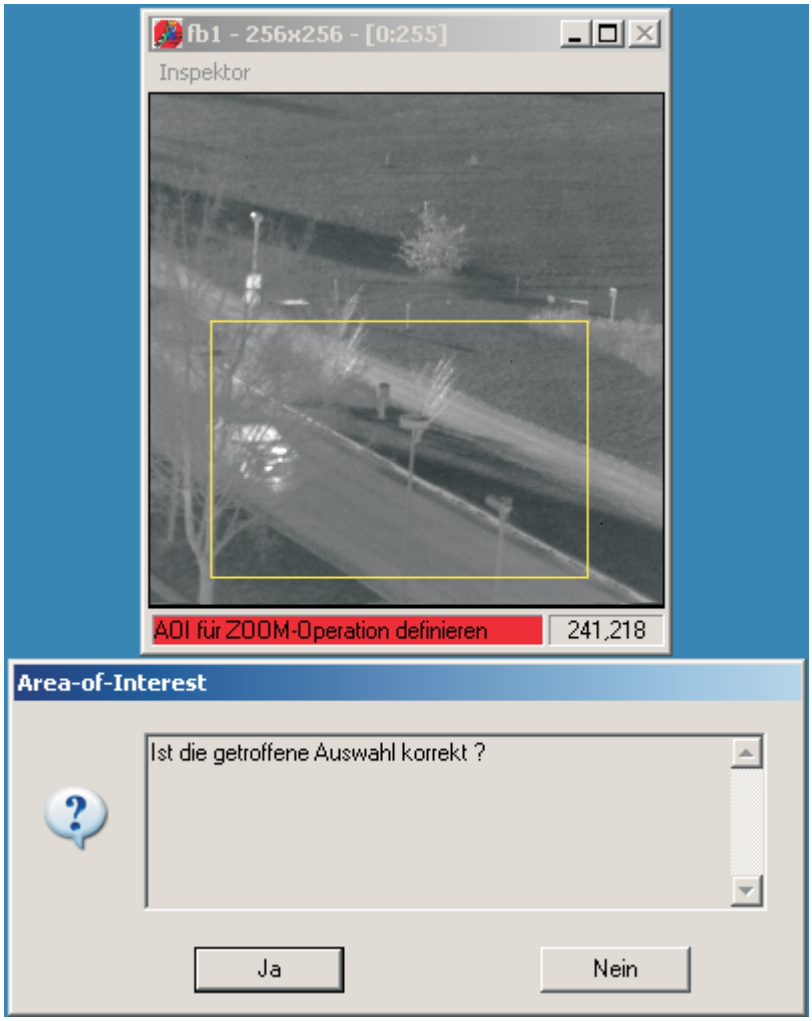


***Abbildung 6.6:*** *AOI-Auswahl per Maus in einem Fenster mit anschließender Bitte um Bestätigung (heurisko 4)*

Für die weitere Bildverarbeitung muss man eines der anderen drei Skripte laden, wobei durch Import immer auch die Funktionalität von `sinitial.ws` zur Verfügung steht. Hier ist das prinzipielle Vorgehen so, das ein gewähltes Verfahren zunächst über Dialoge parametriert wird. Als Beispiel ist in Abbildung 6.7 die Auswahl der Restaurierungsmethode gezeigt. Dann kann man das Verfahren an einem Teil der Sequenz testen. Im Fall der Restaurierung geschieht das durch den Operator `Manuell` im Menü `Restaurierung` (siehe Abbildung 6.8). Ist man mit den gewählten Parametern zufrieden, können diese in einer zum Verfahren gehörigen Konfigurationsdatei gespeichert und später statt einer manuellen Parametrierung wieder eingeladen werden (Operatorgruppe `RestaurierungAuswahl` in Abbildung 6.8). Sind alle Einstellungen abgeschlossen, kann schließlich die zeitraubende automatische Verarbeitung der gesamten Sequenz mit den eingestellten Parametern erfolgen. (Operatoren `Auto` oder `seq_Auto` in Abbildung 6.8).

Dieses Beispiel demonstriert, dass man mit heurisko eine komplexe Anwendung übersichtlich und mit einer ausreichend komfortablen Benutzeroberfläche ausschließlich auf Skriptbasis realisieren kann.


***Abbildung 6.7:*** *Dialog mit 1-aus-n-Auswahl (heurisko 4)*

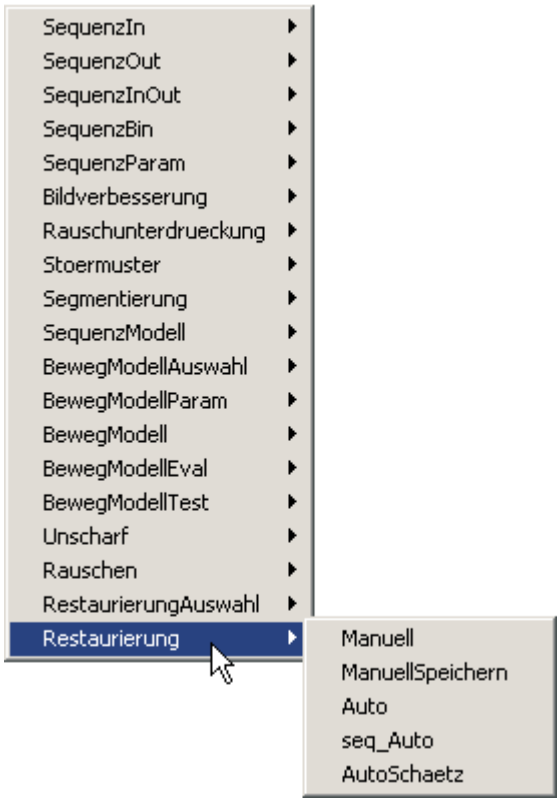


***Abbildung 6.8:*** *Operatorgruppen in* `srestore.ws` *mit Operatoren in der Gruppe* `Restaurierung`

## 6.5 Anwendungsbeispiel: Werkzeuge für CVB

### 6.5.1 Aufgabenstellung

Ein großes Problem in der Bildverarbeitung ist, dass es für die Vielfalt der Hardware wie z. B. Framegrabber eine genauso große Vielfalt der Software-Schnittstellen gibt, die keinem Standard gehorchen, und das sogar trotz einiger Hardware-Standards [Stelz 2003]. Möchte man in einem Projekt die Hardware wechseln, muss man allzu oft auch die Software anpassen, weil die Treiberschnittstelle der neu gewählten Hardware nicht gleich derjenigen der vorher benutzten Hardware ist. *Common Vision Blox* (CVB) der deutschen Firma Stemmer Imaging [CVB] will hier Abhilfe schaffen und bietet mit dem *CVB Image* eine Abstraktionsebene mit einheitlicher Programmierschnittstelle für eine breite Palette von in der Bildverarbeitung benutzter Hardware (siehe Abbildung 6.9). Der Name des Pakets deutet darauf hin, dass

Stemmer eigentlich einmal gehofft hatte, eine breitere Unterstützung in der Hardware-Industrie zu finden und damit so etwas wie einen Standard zu schaffen. Dies ist so jedoch nicht gelungen; die in CVB eingebundene Hardware umfasst lediglich die von Stemmer selbst vertriebene Hardware. Das ist allerdings nicht wenig, da Stemmer einer der größten Vertriebe für Bildverarbeitungskomponenten in Deutschland ist und damit eine breite Palette von Hardware im Programm hat.

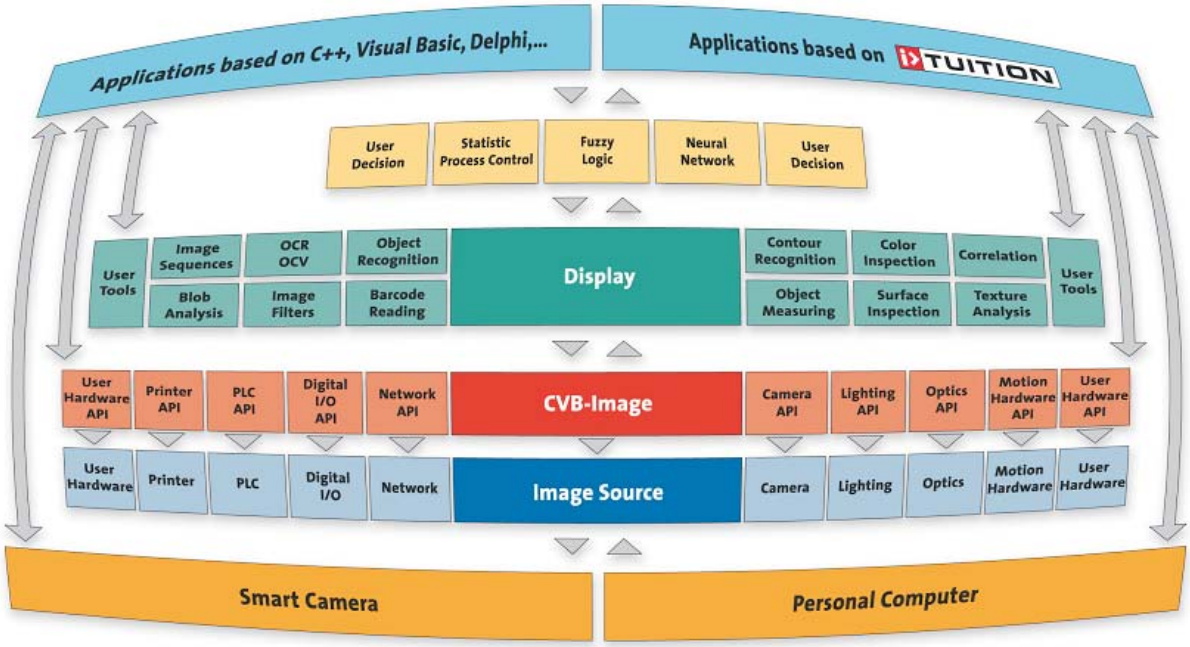


***Abbildung 6.9:*** *Common Vision Blox (aus CVB-Dokumentation)*

Mit CVB wollte Stemmer aber nicht nur eine einheitliche Software-Plattform für Bildverarbeitungshardware schaffen, sondern auch dem Anwendungsentwickler optional ein weit gefächertes Spektrum an Software-Werkzeugen bieten. Es wurden einige Hersteller gewonnen, Werkzeuge für CVB zur Verfügung zu stellen. Durch die enge Verknüpfung mit der Hardware ist CVB auf Windows-Systeme beschränkt, denn in der Regel liefern die Hersteller der Hardware auch nur für diese Plattformen Treiber. CVB kann man als Bibliothek mit den Sprachen C++ und mit den windows-spezifischen Sprachen Visual Basic und Delphi verwenden. Außerdem steht mit *iTUITION* eine Benutzeroberfläche zur graphischen Programmierung zur Verfügung.

Die meisten der CVB-Werkzeuge sind anwendungsorientiert, beispielsweise zum Lesen von Barcodes. Daraus entstand der Wunsch, daneben auch Werkzeuge mit bewährten allgemeinen Bildverarbeitungsfunktionen anbieten zu können. Die Aufgabe ist, aus vorhandenem heurisko-Code ohne großen Zusatzaufwand ein CVB-Werkzeug mit einer Sammlung nützlicher Faltungsoperationen und ein zweites Werkzeug mit FFT-Funktionen in Form von C- oder C++-Bibliotheken zu erstellen.

### 6.5.2 Realisierung

Es geht hier also nicht darum, dass ein Anwender mit heurisko solche CVB-Werkzeuge erstellt, sondern dass die heurisko-Entwickler selbst ihren bewährten und getesteten Code für die geplanten Werkzeuge effizient wieder verwenden. Da die Werkzeuge als Bibliotheken geplant sind, benötigen sie weder den Interpreter noch die Ablaufsteuerung im Kern. Folglich trifft hier keine der in Abschnitt 6.1 dargestellten Architekturen zu.

Hauptpunkt bei einer vorgeschalteten Machbarkeitsprüfung ist die Frage nach der Kompatibilität der internen Datenstrukturen und der unterstützten Datentypen. Da CVB-Bildobjekte aus vorzeichenlosen und vorzeichenbehafteten ganzen Zahlen oder aus einfachgenauen Gleitkommazahlen bestehen können, zeigt ein Vergleich mit Tabelle 4-1 sofort, dass heurisko-Funktionen dafür geeignet sind. Ein CVB-Bildobjekt ist in Abbildung 6.10 zu sehen. Das Objekt besteht aus einem Bildstapel mit *d* Bildebenen. Es handelt sich im Grunde um zweidimensionale Objekte, von denen aber mehrere als zusammengehörig betrachtet werden können. Die Anzahl der Bilder im Stapel ist in CVB die Dimension *d*. Mit $d > 1$ lassen sich beispielsweise Stereobilder, Farbbilder und Bildsequenzen verwirklichen. Im Vergleich zu heurisko handelt es sich aber bei solchen Bildsequenzen nicht um wirklich dreidimensionale Objekte. Während in heurisko Sequenzen mit dreidimensional wirkenden Algorithmen bearbeitet werden können, ist CVB auf sequentielle zweidimensionale Verarbeitung beschränkt. Damit ist klar, dass CVB-Objekte auf heurisko-Atome abgebildet werden können und dass heurisko ebenso von der funktionalen Seite her gerüstet ist.

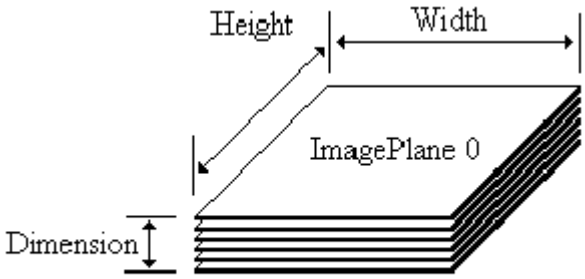


***Abbildung 6.10:*** *Das CVB-Bild ist zweidimensional. Mehrere Bilder können in einem Bildstapel angeordnet werden (aus CVB-Dokumentation).*

Es bleibt die Frage nach den Datenstrukturen. Jedes Bild ist in CVB so gespeichert, dass die Adresse *m* des Bildpunktes (*x*,*y*) mit Hilfe der Gleichung

$$m(x, y) = m(0,0) + vpatx(x) + vpaty(y)$$

berechnet werden kann. Dabei steht *vpat* für *virtual pixel access table*. *Vpatx* und *vpaty* sind Tabellen, die bei der Erzeugung eines Bildes mit Bezug auf die Basisadresse *m*(0,0) berechnet werden und einen schnellen Zugriff auf jeden beliebigen Bildpunkt ermöglichen.

Der allgemeine Fall dieses Konzepts entspricht nicht dem zeilenorientierten Speicherschema der heurisko-Objekte, erfordert also die Erzeugung temporärer Objekte und Umkopieren der Daten. Man kann jedoch mit CVB-eigenen Analysefunktionen feststellen, ob die Adresstabellen für ein Bild nach einem bestimmten einfachen Schema aufgebaut sind. Gibt `AnalyseXVPAT()` den Kennwert `XVPAT_LINEAR_WITH_DATATYPE` zurück, bedeutet dies, dass die Bildpunkte einer Bildzeile im Speicher direkt nebeneinander liegen. Das ist aber genau die gleiche Anordnung wie die der heurisko-Atome. Die Tabelle *vpaty* ist dann identisch mit der Zeilenzeigerliste in heurisko. Der folgende Code-Auszug zeigt die Erzeugung eines heurisko-Atoms zu einem CVB-Bild:

```
// heurisko-Atom zu CVB-Bild erzeugen

hERR CreateAtom(IMG img, hUINT32 type, long plane,
                hUINT_PTR n[2], BOOL srcobj, hATOM** atom) {
void** p2;
long incx, incy;
```

```
char* data = NULL;
PVPAT vpat;

GetImageVPA (img, plane, (LPVOID*)&data, &vpat);
if (AnalyseXVPAT(img, plane, &incx) ==
                                XVPAT_LINEAR_WITH_DATATYPE) {
    // nur Atom erzeugen und Zeigerliste füllen
    GetLinearAccess(img, plane, (LPVOID*)&data, &incx, &incy);
    *atom = aCreate(2, n, type, NULL, HMOD_NOALIGN);
    if (*atom == NULL) return AERR_ALLOCATION_FAILED;
    aSetProperties(aGetProp(*atom), type, 0, NULL, NULL);
                                     // Set image properties
    p2 = aGetPtr2(*atom);
    for (j = 0; j < n[0]; j++) {
        p2[j] = data + vpat[j].YEntry;
    }
} else {
    // Atom erzeugen und Daten kopieren, falls Eingabeobjekt
    *atom = aCreate(2, n, type, NULL, HMOD_NOALIGN|HA_DATA);
    if (*atom == NULL) return AERR_ALLOCATION_FAILED;
    aSetProperties(aGetProp(*atom), type, 0, NULL, NULL);
                        // Set image properties
    if (srcobj) { // wenn Eingabeobjekt, Daten kopieren
        p2 = aGetPtr2(*atom);
        switch (type) {
        case H_UINT8_TYPE:
        case H_SINT8_TYPE:
                    CopyCVB_BYTE  (p2,n,data,vpat); break;
        case H_UINT16_TYPE:
        case H_SINT16_TYPE:
                    CopyCVB_2BYTES(p2,n,data,vpat); break;
        case H_UINT32_TYPE:
        case H_SINT32_TYPE:
        case H_FLOAT_TYPE:
                    CopyCVB_4BYTES(p2,n,data,vpat); break;
        case H_DOUBLE_TYPE:
                    CopyCVB_8BYTES(p2,n,data,vpat); break;
        case H_FLOAT_TYPE | H_COMPLEX:
                    n[1] *= 2;
                    CopyCVB_4BYTES(p2,n,data,vpat); break;
        default: return AERR_INVALID_TYPE;
        }
        // wenn Inplace-Operation, müssen die Ergebnisse wieder
        // ins CVB-Eingabeobjekt zurückkopiert warden
        if (inplace) mustCopy = true;
    } else {
        // Ergebnisse müssen ins CVB-Ausgabeobjekt kopiert
        // werden
        mustCopy = true;
    }
}
return NO;
}
```

Diese Funktion wird für jedes beim Aufruf einer Werkzeugfunktion übergebene Bildobjekt vom Typ `IMG` ausgeführt. Handelt es sich bei dem CVB-Objekt um ein zu heurisko kompatibles Speicherschema, wird dazu nur ein Atom ohne eigene Daten benötigt. Lediglich die Datenzeiger werden mit Hilfe der Tabelle *vpaty* noch gesetzt. Im allgemeineren Fall wird ein

Atom mit eigenen Daten erzeugt und eine datentypabhängige Funktion aufgerufen, die mit Hilfe beider VPAT-Tabellen die Daten umkopiert.

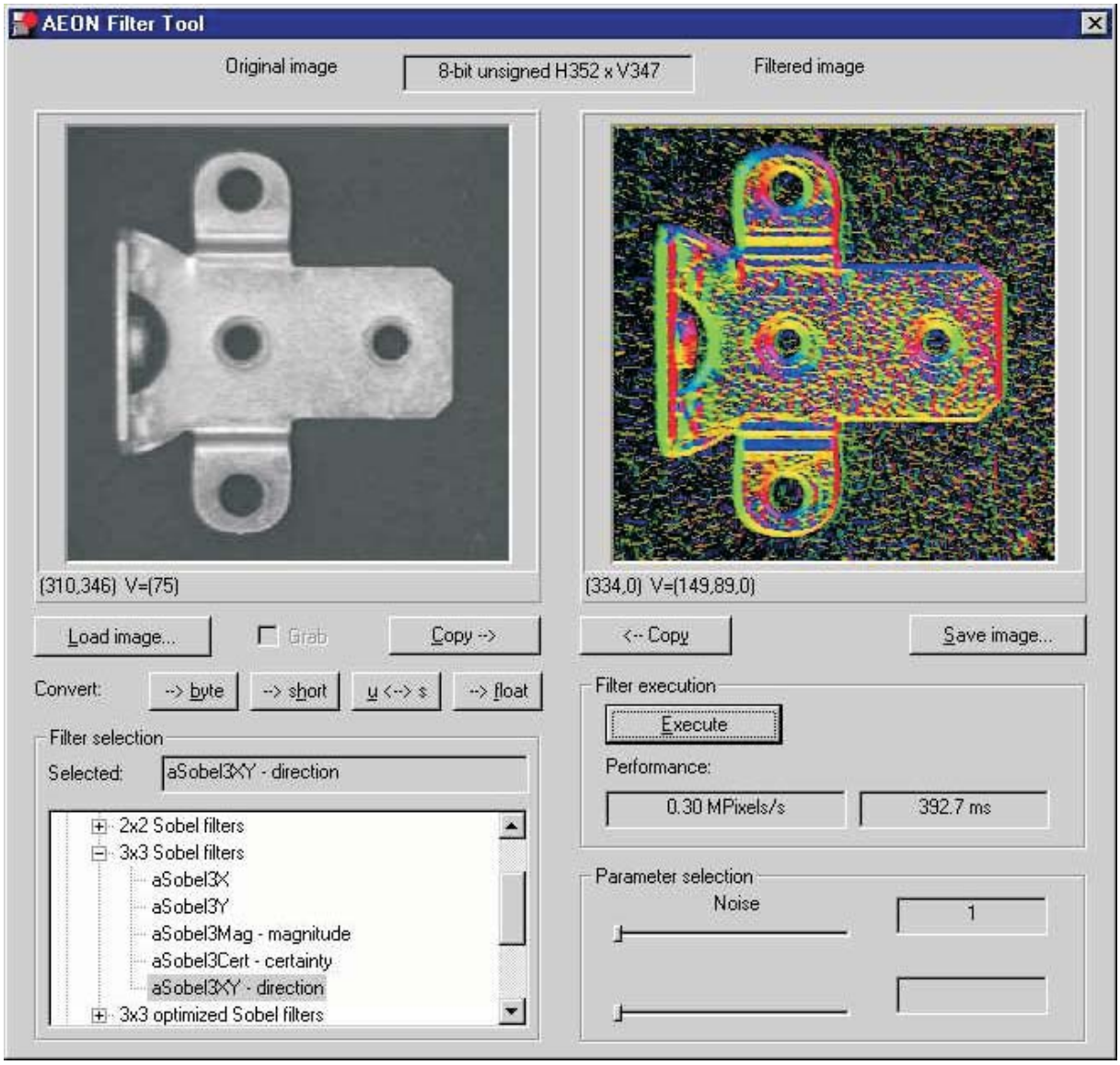


***Abbildung 6.11:*** *Evaluierungsprogramm für das Filtermodul zu Common Vision Blox*

Damit steht zwar fest, dass die zu erstellenden CVB-Werkzeuge für den allgemeinen Fall Funktionen zur Erzeugung temporärer Objekte und zum Umkopieren der Daten bereitstellen müssen, es hat sich aber herausgestellt, dass heurisko in den meisten Anwendungsfällen direkt auf kompatiblen CVB-Daten arbeiten kann. Dass CVB mit den beiden Tabellen überhaupt den allgemeineren Fall berücksichtigt, wird mit denkbaren exotischen Speicheranordnungen durch Treiber von Bilderfassungsgeräten begründet. Gemäß den Videonormen EIA und CCIR werden beispielsweise zwei Halbbilder nacheinander digitalisiert, einmal eines mit den geraden Zeilen, dann eines mit den ungeraden. Würde der Gerätetreiber die beiden Halbbilder so speichern, wie sie in den Rechner kommen, müsste man *vpaty* entsprechend füllen, um die für Algorithmen übliche Zeilenreihenfolge zu erhalten. Die Erfahrung zeigt jedoch, dass die meisten Treiber schon selbst die Umsortierung vornehmen und die Daten in der „richtigen“ Reihenfolge im Speicher ablegen.

Für jedes der beiden neuen CVB-Werkzeuge wird *ein* Bibliotheksmodul erzeugt. Da, wie oben festgestellt, nur Atom- und Vektorfunktionen aus heurisko gebraucht werden, besteht die Implementierungsarbeit aus den im Weiteren beschriebenen vier Schritten.

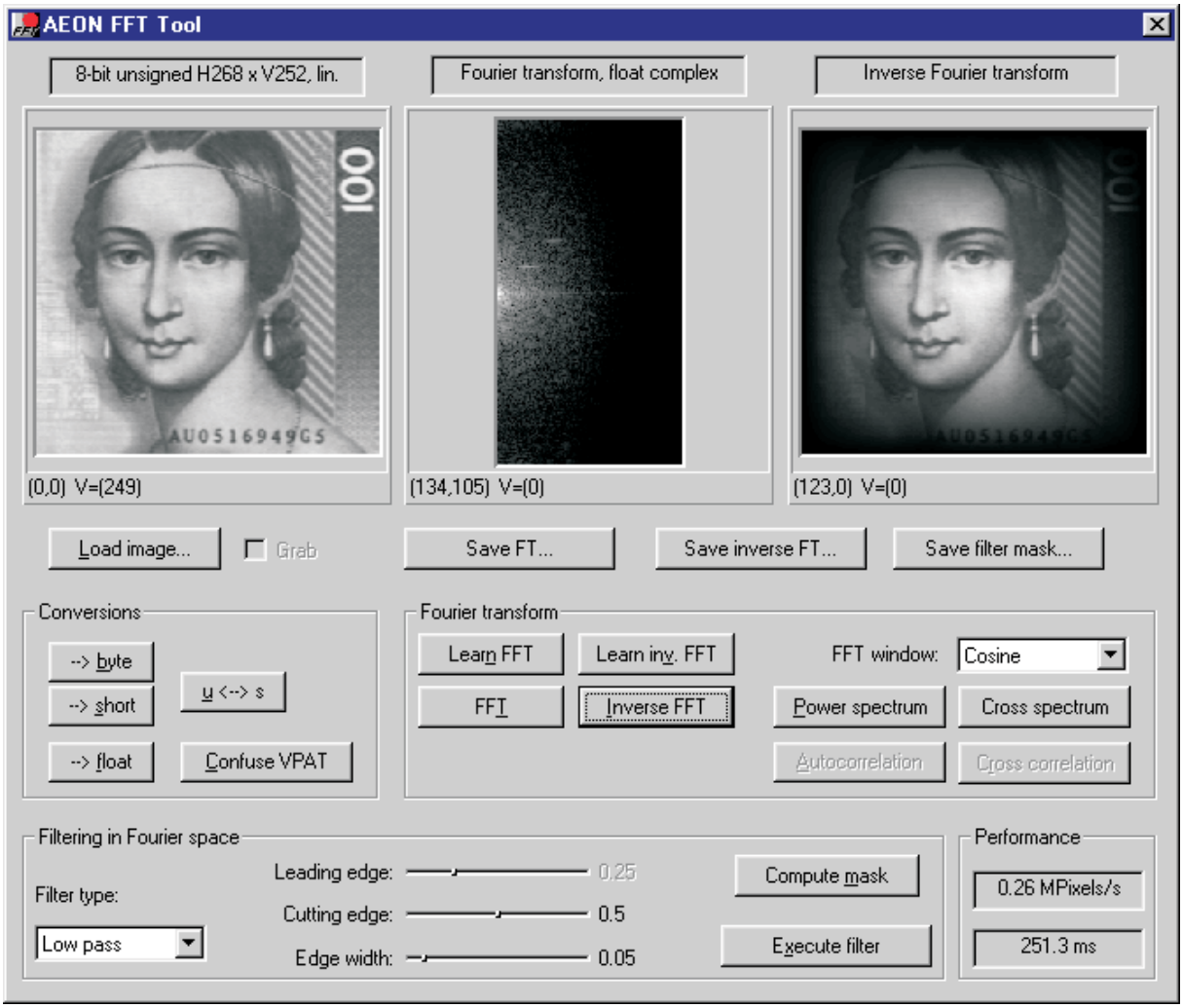


***Abbildung 6.12:*** *Evaluierungsprogramm für fas FFT-Modul zu Common Vision Blox*

1. Für die von den Bibliotheken angebotenen Funktionen müssen alle Dateien aus der heurisko-Codebasis herausgesucht werden, in denen benötigte Atom- und Vektorfunktionen implementiert sind. Da der Code sorgfältig nach Datentypen und nach funktionalen Gesichtspunkten sortiert ist, ist das eine leichte Arbeit.

2. Für die Bibliotheken kann nicht die übliche Initialisierung über den heurisko-Kern verwendet werden. Deshalb muss für jedes Werkzeug eine neue Datei zur inneren Initialisierung geschrieben werden. Im Wesentlichen handelt es sich dabei um den Aufbau der Vektorfunktionslisten, aus denen von den Atomfunktionen mit Hilfe einer dem jeweiligen Datentyp entsprechenden Kennzahl die passenden Vektorfunktionen gefunden werden. Dies ist keine prinzipielle Schwierigkeit, nur ein wenig Fleißarbeit.

3. Im dritten Arbeitsschritt muss die Schnittstelle zwischen CVB und den Atomfunktionen erstellt werden. Sie besteht aus den exportierten Bibliotheksfunktionen, welche als Parameter CVB-Bilder annehmen, diese in heurisko-Objekte umsetzen und dann die Atomfunktionen aufrufen. Das Umsetzen in heurisko-Objekte geschieht wie oben beschrieben.

4. Der letzte Arbeitsschritt ist der aufwendigste. Hier soll gemäß der verbindlichen Vorgabe für alle CVB-Werkzeuge ein Evaluierungs- und Übungsprogramm erstellt werden. Für die beiden Module stellen diese Programme eine graphische Benutzeroberfläche zur Verfügung, mit der sämtliche Funktionen mit allen unterstützten

Datentypen ausprobiert werden können. Die beiden Programme zeigen Abbildung 6.11 und Abbildung 6.12.

Als Fazit kann man festhalten, dass die Aufgabe mit erfreulich wenig Aufwand gelöst werden kann. An dieser Stelle soll kurz auch noch ein anderes erfolgreiches Anwendungsbeispiel erwähnt werden. Die deutsche Firma ABW produziert Projektoren, die zur 3-D-Objektvermessung durch Anwendung des "codierten Lichtansatzes", einer Erweiterung des Lichtschnittverfahrens, eingesetzt werden [ABW]. Zur Auswertung liefert ABW die Software-Bibliothek *ABW-3D*. Durch ein Erweiterungsmodul ist diese Software in heurisko eingebunden, so dass es möglich ist, mit heurisko-Skripten die Projektoren und Kameras zu steuern, eine Auswertung in ABW-3D zu veranlassen, die Ergebnisse in heurisko hinein zu holen und dort weiter zu verarbeiten. Hier ist also die Benutzungsrichtung gegenüber den CVB-Werkzeugen umgekehrt. Allerdings kann ABW-3D heurisko zur Bildakquisition einsetzen. Für jede unterstützte ABW-3D-Funktion gibt es in heurisko einen Operator. Beim Aufruf müssen lediglich die heurisko-Objekte wie immer ausgewertet und dann in Parameter für die ABW-3D-Funktionen umgesetzt werden. Für die Übernahme der Ergebnisse aus ABW-3D werden die Daten kopiert, wobei alle Daten vom Typ `float` sind. Ebenso wird bei der Bildakquisition von heurisko zu ABW-3D kopiert. Auch hier gibt es nur einen Datentyp, nämlich `ubyte`. Auf Seiten von ABW-3D wurde die eigentliche C++-Bibliothek für die Integration mit einer C-Schnittstelle versehen. Damit wurde die Arbeit zwischen beiden Partnern etwa gleichmäßig aufgeteilt und in wenigen Tagen erledigt.

## 6.6 Anwendungsbeispiel: Qualitätskontrolle bei der Kameraherstellung

### 6.6.1 Aufgabenstellung

Ein Hersteller von hochwertigen CCD- und CMOS-Kameras für Wissenschaft und Industrie möchte in der Fertigung und Qualitätssicherung Bildverarbeitung einsetzen. Seine Ziele sind:

- *objektive Qualität:* Durch den Einsatz einer maschinellen Prüfung oder durch die maschinelle Unterstützung der Fertigung nimmt die Abhängigkeit der Qualität von der aktuellen Form der sonst die Arbeiten alleine ausführenden Menschen ab.

- *höhere Effizienz:* Viele Arbeiten können schneller und trotzdem genauer durchgeführt werden.

- *Ausweitung der Qualitätssicherung:* Manche Eigenschaften, die von Menschen nur sehr schwer oder gar nicht beurteilt oder gemessen werden können, lassen sich mit Bildverarbeitung erfassen.

- *Dokumentation der Kameraeigenschaften:* Während der gesamten Lebensdauer lassen sich die Eigenschaften einer Kamera mit Bildverarbeitung objektiv und gleich bleibend messen und dokumentieren.

In einer ersten Ausbaustufe werden folgende Messaufgaben realisiert:

1. *Zentrierung des* Sensors*:* Der Träger des lichtempfindlichen Sensors muss justiert werden. Die Abweichung von der Sollposition soll automatisch berechnet werden.

2. *objektives Schärfemaß:* Die objektive Berechnung eines Schärfemaßes ist sowohl für die Überprüfung der korrekten Tiefenlage eines Sensors als auch für die Einstellung von Optiken wichtig.

3. *Modulations-Transferfunktion (MTF):* Die MTF ist die Übertragungsfunktion einer Kamera, d. h. die Amplitude des Ausgangssignals (Grauwert) in Abhängigkeit von der Frequenz eines sinusförmigen Eingangssignals (Licht). Nach dem Abtasttheorem müsste die Kamera Signale bis zu einer Periode, die noch zwei Punkte des Sensors umfasst, übertragen können. In der Realität fällt die MTF jedoch schon weit eher ab.

4. *Reinheit der Eingangsgläser:* Die Eingangsdeckgläser zum Schutz der Sensoren vor Umwelteinflüssen können Schlieren, Kratzer oder Blasen und äußere Verschmutzungen aufweisen.

5. *Blooming:* Bei hoher Intensität eines Signals in einem Bildpunkt gelangen Ladungsträger auch zu den benachbarten Bildpunkten. Bei guter Unterdrückung erfolgt das Blooming erst bei sehr hoher Intensität und weniger ausgedehnt.

6. *H-synchrone Störungen:* Damit sind zeilensynchrone Störungen gemeint, die in jeder Bildzeile auftreten. Das kann beispielsweise der Einfluss eines Netzteils sein.

7. Rauschmessungen: Eine gute Kamera soll das Photonenrauschen unverfälscht wiedergeben. Das Photonenrauschen resultiert aus der Statistik der auf einen Sensor treffenden Photonen und wächst proportional zum Signal.

8. Linearitätsmessungen: Bei Sensoren mit linearer Kennlinie soll auch zwischen der Bestrahlung und dem Grauwert eine möglichst exakte lineare Beziehung bestehen.

Besondere Kennzeichen:

- Die Anwendung ist nicht zeitkritisch. Trotzdem ist teilweise interaktives Arbeiten wie bei der CCD-Kopf-Justierung erwünscht.

- Die Anwendung muss leicht an neue Kameras anpassbar sein.

- Der Kamerahersteller beauftragt den Hersteller von heurisko zur Entwicklung einer Startversion der Algorithmen. Den Aufbau des Qualitätssicherungssystems, die Integration der Algorithmen und deren Verfeinerung nimmt der Auftraggeber selbst vor, wobei er dazu eine Diplomarbeit an eine Fachhochschule vergibt [Eckl 2004].

### 6.6.2 Realisierung

Der Kamerahersteller hat in der Vergangenheit umfangreiche Software inklusive graphischer Benutzeroberflächen zum Testen während der Entwicklung und Erprobung im Werk und zur Inbetriebnahme und Konfiguration beim Kunden erstellt. Die Software ist an seine Bedürfnisse angepasst und ermöglicht leicht die Integration neuer Kameras. Weil dies ein elementares Element im Geschäftsablauf des Herstellers ist, entschließt er sich dazu, die Kameraunterstützung und Datenerfassung des Projekts keinerlei Bedingungen zu unterwerfen, welche die Realisierung als Erweiterungsmodule zu heurisko möglicherweise aufzwingen würde, und sich alle Freiheiten der eigenen Weiterentwicklung durch Verwendung der vorhandenen Software zu erhalten. Das Bildverarbeitungssystem soll eine eigene, angepasste Bedienoberfläche erhalten. Außerdem sind viele Verwaltungs- und Archivierungsaufgaben denkbar. Deshalb wird heurisko nur als reines Bild*verarbeitung*spaket gemäß Abbildung 6.13 verwen-

det. Bei Bedarf können allerdings zusätzlich die Visualisierungsfunktionalität heuriskos und die speziellen Operatoren mit Dialogen eingesetzt werden.

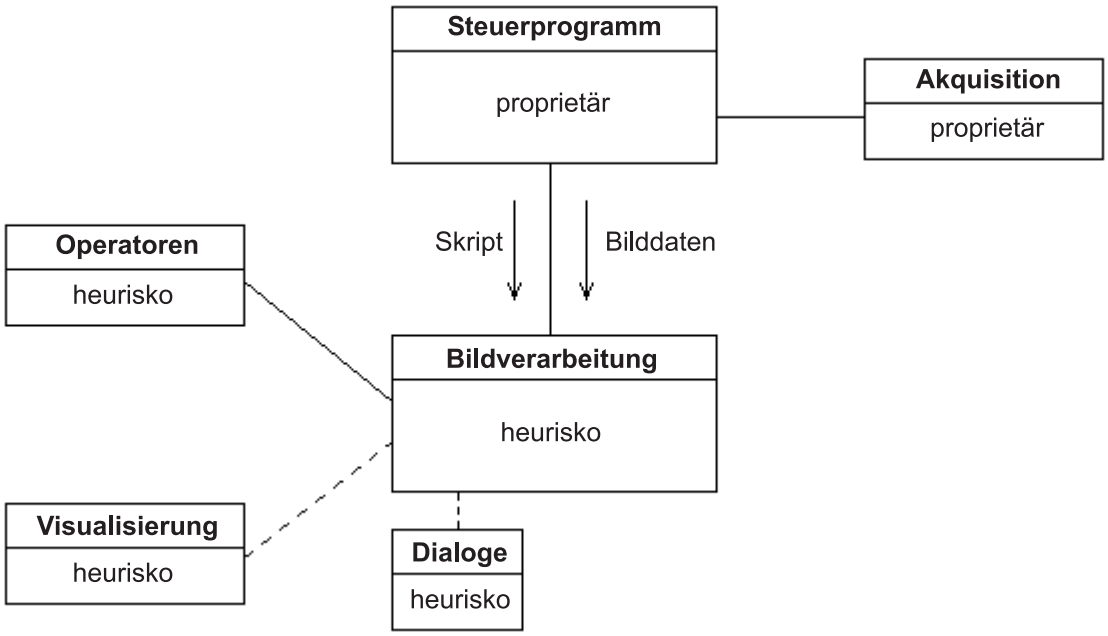


***Abbildung 6.13:*** *Architektur des Bildverarbeitungssystem eines Kameraherstellers*

Die gewählte Architektur hat zur Folge, dass die zu verarbeitenden Daten im Steuerprogramm erzeugt werden und zur Bildverarbeitung an heurisko übergeben werden müssen. Da Kopieren innerhalb des Speichers vergleichsweise schnell ist (siehe Tabelle 6-4), wird für den Datenaustausch folgender Weg gewählt: Das Steuerprogramm lässt heurisko ein passendes Objekt erzeugen und besorgt sich über die Funktion `hiObjAtomDataPtr()` des Interpreters den Zeiger auf die Zeilenzeigerliste und eventuell zur Kontrolle die Kenndaten des Objekts. Dann kopiert es die Daten zeilenweise aus dem Puffer des Steuerprogramms in das heurisko-Objekt. Das Programm dazu könnte folgende Zeilen enthalten:

```
// C-Code
hERR error;
void** ptr2;
hUINT_PTR size, nlist, nvec, par[10];
 …
// Objekt x in heurisko erzeugen:
error = hiDo("ushort x[480][640]", T_INSTRUCT_CL);
…
error = hiObj("x", &size); // x wird aktuelles Objekt
if (!hiObjAtomDataPtr(&ptr2, &nlist, &nvec, par)) {
    // (unsigned short*)ptr2[i] zeigt auf i-te Bildzeile
    …
}
```

***Tabelle 6-4:*** *Vergleich der benötigten Zeiten für Kopieren und Glätten von Objekten mit 16-Bit-Zahlen*

| Operation | Bildgröße | Ausführungszeit [ms] |
|---|---|---|
| Kopieren | 480×640 | 0,7 |
| Glättung mit 3×3-Maske | 480×640 | 1,5 |
| Kopieren | 1024×1392 | 2,3 |
| Glättung mit 3×3-Maske | 1024×1392 | 7,5 |

Die Entwicklung der Algorithmen geschieht ohne das Steuerprogramm innerhalb von heurisko mit Hilfe von gespeicherten Beispielbildern und mit einer Beispielkamera des Auftraggebers. Für jede Messaufgabe wird ein Workspace entwickelt, der mit allen Kameras verwendet werden kann. Dazu helfen drei Konzepte.

Erstens wird mit dem Präprozessorkommando *select* durch eine 1-aus-2-Auswahl in einem Dialog festgelegt, ob ein Betrieb mit oder ohne Kamera stattfinden soll, und eine entsprechende Konstante definiert. Das entspricht einem `#define` in C. Allerdings geschieht die Definition mit `select` interaktiv zur Laufzeit während der Initialisierung. Die statische Variante mit `define` existiert ebenso. Mit darauf folgenden `ifdef`-Abfragen wird zu dem zur Auswahl passenden Code verzweigt.

```
select Betriebsweise des Programms mit oder ohne
       Kameras|MIT|OHNE;
…
ifdef MIT; // mit Kamera
…
elseifdef; // ohne Kamera
…
endifdef;
```

Zweitens verwendet jeder Workspace variable Objektgrößen. Auch der Datentyp könnte, falls nötig, variabel gehalten werden. Zurzeit wird allerdings der Typ `ushort` statisch verwendet, da alle Kameras mehr als 8 Bit und höchstens 16 Bit je Bildpunkt liefern. Mit diesem Konzept unterstützt jedes Skript alle Kameras.

```
ushort x[*256][*256]; // Bild, * bedeutet, dass variabel
ushort n[2];          // Bildgröße
ushort nx = n[0]; ushort ny = n[1];
string file;          // TIFF-Datei, wenn ohne Kamera
string camtype;       // Kameratyp
struct buffer {
    string name,
    long dim,
    long size[3],
    long offs[3],
    string format
};
// Bildspeicher, kurze Sequenz mit zwei Bildern
// -1 = voreingestellte Größe von Kamera übernehmen
buffer = "buf".{3}.{-1,-1, 2}.{-1,-1, 0}." %3i";
…
ifdef MIT;
    device cam, type camtype, buffers buffer;
    n = GetCoordinates(cam.buf);   // Objektgröße ermitteln
    global renew ushort x[ny][nx]; // x anpassen
elseifdef;
    x = Read(file); // x erhält automatisch richtige Größe
    n = GetCoordinates(x); // Objektgröße ermitteln
endifdef;
resize();                  // andere globale Objekte anpassen
```

Drittens erhalten die Skripte weitere Flexibilität durch Konfigurationsdialoge und Konfigurationsdateien. Einfache Standarddialoge wie 1-aus-*n*- und *m*-aus-*n*-Auswahlen lassen sich in heurisko mit wenigen Zeilen programmieren und zur Konfiguration eines Algorithmus nutzen. Speichert man die eingestellten Verfahrensparameter in eine Datei, kann man sich bei

späteren Anwendungen die erneute Konfigurationsprozedur sparen und stattdessen die Konfigurationsdatei einladen. Wie das gehen kann, zeigt das folgende Beispiel aus dem Projekt. Zunächst einmal wird eine globale Struktur zur Speicherung der Parameter erzeugt:

```
// globale Konfigurationsstruktur
struct config {
    long camera,     // Kameratyp
    long zoom,       // Darstellungsmaßstab
    long nb          // Anzahl Bilder
};
```

Möglicherweise muss man noch weitere Strukturen hinzunehmen, denn eine Komponente eines Verbundobjekts kann zurzeit nicht wieder ein Verbundobjekt sein, obwohl das von der internen Molekülverwaltung in heurisko her möglich wäre. Der Algorithmus für die objektive Schärfe benötigt z. B. AOIs:

```
# Region of interests (AOIs)
struct aoi/maxaoi/ {
    ushort offset[2],
    size[2]
};
```

Das Setzen der Parameter sei beispielhaft für die AOIs gezeigt. Es können bis zu einer Maximalzahl beliebig viele AOIs gewählt werden. Mit jedem Aufruf des selbst definierten Operators `aoiAdd()` wird eine weitere AOI zur Liste der AOIs hinzugefügt. Ist das Limit schon erreicht, wird mit dem Operator `Pause()` ein Dialog mit einem entsprechenden Hinweis geöffnet und nach Bestätigung die AOI-Auswahl verlassen. Ansonsten muss der Benutzer in dem Fenster mit dem Kamerabild die neue AOI mit einem Rechteck markieren. Ist er mit der Wahl nicht zufrieden, wird das Rechteck wieder gelöscht. Eine akzeptierte AOI wird mit ihrem Listenindex beschriftet.

```
# AOI hinzufügen
operator aoiAdd();
    if (caoi == maxaoi);
        Pause.error("Maximale Anzahl AOIs schon erreicht");
        return;
    endif;
    long ok;
    aoi/caoi/ = ivSelect.rect(d1);
    ivDraw.rect(d1, aoi/caoi/[:2]);
    ok = Pause.question("Auswahl OK?");
    if (!ok);
        ivSetPen.del(d1, 1, {255,0,0});
        ivDraw.rect(d1, aoi/caoi/[:2]); // AOI löschen
        ivSetPen.copy(d1, 1, {255,0,0});
        return;
    endif;
    text.pos = Add(aoi/caoi/.offset[:2], {0,20});
    caoi = Inc();
    text.s = caoi;
    ivDraw.text(d1, text);
endoperator;
```

Die Parameter werden in einer Konfigurationsdatei im Rohformat gespeichert. Wie das vor sich geht, kann man auch am Lesevorgang sehen. Dazu öffnet der selbstdefinierte Operator `config()` mit `FileOpenRead()` die Datei. Mit `FileRead()` wird Byte für Byte aus

der Datei gelesen, bis das Zielobjekt - im Beispiel ist das `config` – gefüllt ist. Mit nachfolgenden Leseoperationen kann weiter aus der Datei gelesen werden, so auch die AOIs. `FileClose()` schließt die Konfigurationsdatei wieder. Der Operator `config()` bietet dem Benutzer nicht nur das Lesen aus einer per Dateidialog selektierten Datei, sondern auch die Wahl vordefinierter Konfigurationen. In diesen Fällen enthält `configname` nicht den Namen einer Datei, sondern die Bezeichnung der Konfiguration.

```
// Kameraauswahl
operator config();
    cselect = -1;
    while(cselect < 0);
        cselect, cseldef = Input.select(cheader, cdescr);
        if (cselect == 2);
            short handle;
            configname = RequestFiles("Konfigurationsdatei
                         laden"."*.cfg");
            if (configname);
                handle = FileOpenRead.raw(configname);
                config = FileRead(handle);
                aoi = FileRead(handle);
                FileClose(handle);
            else;
                cselect = -1; // nochmals abfragen
            endif;
        else;
            configname = cdescr[cselect];
        endif;
    endwhile;
endoperator;
```

Die Anwendung des Kameraherstellers benutzt nicht direkt die entwickelten heurisko-Skripte, die man den Interpreter mit dem Kommando `WsRun` laden und ausführen lassen könnte. Vielmehr wird der heurisko-Code in den Quellcode des Steuerprogramms kopiert und zur Ausführung über die Interpreterschnittstelle an heurisko weitergereicht. Man gibt hier dem Argument, das vollständige Programm einschließlich der heurisko-Skripte in einer gemeinsamen Quelle zu halten, den Vorzug und nimmt in Kauf, dass beim Übertragen der Skripte Fehler entstehen könnten.

# 7 Resümee und Ausblick

Zu Beginn des Kapitels 4 wurden die Konzepte für ein effizientes Bildverarbeitungssystem diskutiert und dann bei der Entwicklung des Paketes heurisko umgesetzt. Nachdem in Kapitel 6 mit einer breiten Auswahl von Beispielaufgaben gezeigt wurde, wie sie mit heurisko gelöst wurden, sollen hier die wichtigsten Ergebnisse zusammengefasst werden und ein kleiner Ausblick erfolgen.

**Bibliotheksfunktionen und Skriptsprache:** Für heurisko wurde die Vielfalt der Bildverarbeitungsfunktionen systematisch analysiert und in Klassen eingeteilt. Außerdem wurde ein hierarchischer interner Aufbau der Bibliothek mit Molekül-, Atom- und Vektorfunktionen realisiert. Dies ermöglicht zum einen die Einführung von generischen Funktionen auf der Molekül- und Atomebene als auch eine erfreuliche Wiederverwendbarkeit von Code auf allen Ebenen. Dies hat weiter zur Folge, dass sich die Optimierung einzelner Vektorfunktionen vielfach auf mehrere Bildverarbeitungsfunktionen auswirkt.

Für heurisko wurde eine Skriptsprache entworfen, die eine schnelle Entwicklung und Änderung von Algorithmen ermöglicht. Es wurde ein binärer Zwischencode eingeführt, der den Geschwindigkeitsnachteil einer interpretierten gegenüber einer kompilierten Sprache praktisch ausräumt. Man muss allerdings im Falle eines benutzerdefinierten Steuerprogramms darauf achten, den Interpreter nicht mit einer Folge von Aufrufen zu belasten, sondern alle Abläufe möglichst weitgehend in ein heurisko-Skript zu verlagern. Die Skriptsprache wurde mit allen wichtigen Elementen einer modernen Programmiersprache ausgestattet, die man zur Programmflusssteuerung benötigt. Darüber hinaus enthält heurisko auch speziell auf die Bildverarbeitung zugeschnittene Möglichkeiten wie den einfachen und schnellen Zugriff auf Datenausschnitte über AOIs. Lange Zeit war es in einem heurisko-Skript nicht möglich, Operatoren zu verschachteln oder die Infix-Schreibweise mit Operatorsymbolen zu verwenden. Dies wurde von den Anwendern immer wieder als störender Mangel empfunden und deshalb jetzt unter weitgehender Vermeidung von Nachteilen durch temporäre Objekte ermöglicht. Gleichzeitig wird die Verschachtelung zu einer Laufzeitverbesserung durch Vektorisierung ausgenutzt.

**Einheitliche Datenstruktur für multidimensionale Bildverarbeitung:** Entscheidenden Anteil am effizienten Aufbau der Bibliothek hat die universale Datenstruktur für die zu verarbeitenden Objekte. Funktionen, die nicht wissen müssen, welche Objekte sie als Parameter erhalten und an andere Funktionen weitergeben, gehen einfach mit Zeigern auf Moleküle oder Atome oder mit Indices der Objektfelder um, ohne sich um weitere Details zu kümmern.

Die heurisko-Moleküle setzen sich aus beliebig vielen Submolekülen zusammen, welche die Knoten eines Molekülbaumes bilden. Die Blätter dieses Baumes, auch als Atome bezeichnet, enthalten die Daten. Das vorausschauende Molekülkonzept mit den von Grund auf beliebigdimensionalen Atomen bietet eine Datenstruktur, die auf absehbare Zeit den Anforderungen in der Bildverarbeitung gewachsen sein dürfte.

Alle Operatoren sind bezüglich ihrer Verwendung in der Skriptsprache unabhängig von der Objektstruktur. Allerdings verlangt die Implementierung, dass die Strukturen der an einem Operatoraufruf beteiligten Objekte zueinander passen. Ist das einmal nicht der Fall, erhält der Anwender eine entsprechende Fehlermeldung. Durch das Konzept des Expandierens, Minimierens und Reduzierens sind die Operatoren allerdings sehr flexibel einsetzbar.

**Datentypen:** Im Gegensatz zu numerischen Bibliotheken werden in der Bildverarbeitung eine Fülle von Datentypen benötigt. Deshalb unterstützt heurisko neben den einfach- und doppeltgenauen Gleitkommazahlen und ihren komplexen Varianten alle möglichen ganzen Zahlen von einem Bit bis zur Registerlänge eines Prozessors. Außerdem hält heurisko für mehrkanalige Bildobjekte, bei denen für einen Datenpunkt die Werte der einzelnen Kanäle hintereinander im Speicher stehen, eigene, so genannte gepackte Datentypen bereit. Dies erleichtert dem Anwender den Umgang mit solchen Daten.

Der datentypunabhängige Teil der Algorithmik steckt in den Molekül- und Atomfunktionen und der datentypabhängige Teil in den Vektorfunktionen. Damit kann ein Großteil der Algorithmik für alle Datentypen gemeinsam verwendet werden. Gleichzeitig ist es leichter, den abhängigen Codeteil mit im Idealfall deutlich geringerem Umfang gezielt zu optimieren, z. B. durch Benutzung der Multimedia-Instruktionssätze.

Alle Operatoren sind bezüglich ihrer Verwendung in der Skriptsprache datentypunabhängig. Allerdings ist es unnötig, jeden Operator auch tatsächlich für alle denkbaren Datentypen zu implementieren.

**Datenein- und -ausgabe:** Die Bildakquisition und mehr und mehr auch die Erfassung analoger und digitaler Messdaten sind zentrale Elemente vieler Bildanalysesysteme. Hinzu kommt aber auch die Steuerung und Regelung von Prozessen. Wegen der Vielfältigkeit der Systeme bietet heurisko mit dem Modul `hk_ac` eine offene Schnittstelle, die sich schon bei der Einbindung unterschiedlichster Geräte, von Kameras über I/O-Karten und Spektrometer bis zu Kommunikationsgeräten, bewährt hat. Für den Umgang mit Dateien gibt es ebenso eine spezielle Schnittstelle, welche die Unterstützung eines bestimmten Datenformates durch ein Erweiterungsmodul erleichtert. Die Schnittstelle ist in das Kernmodul integriert und bisher noch nicht offen gelegt worden, was allerdings keine prinzipiellen Gründe hat.

**Benutzerschnittstelle:** Die Benutzerschnittstelle von heurisko ist in einem eigenen Modul untergebracht, das als Steuerprogramm bezeichnet wurde. Das Steuerprogramm kann vom Anwender durch ein beliebiges Programm ersetzt werden. Dies erlaubt sowohl die Einbindung von heurisko in andere Programme als auch die Ausstattung der Anwendung mit einer Benutzerschnittstelle nach den Anforderungen, Vorlieben und Erfahrungen des Benutzers. Die gesamte Visualisierung und zu Operatoren gehörige Dialoge sind nicht im Steuerprogramm integriert, so dass sie dem Anwender bei Bedarf auch dann zur Verfügung stehen, wenn er sein eigenes Steuerprogramm einsetzt.

Für die Visualisierung und Inspektion von Objekten bietet heurisko mit dem Modul `hk_iv` eine weitere offene Schnittstelle. Auch diese Schnittstelle hat sich bereits bei der Einbindung anderer Softwarepakete bewährt.

Ein gewisser Nachteil der Benutzeroberfläche von heurisko ist, dass bislang kein sehr großer Aufwand für sie betrieben werden konnte und deshalb von anderen, bekannten Programmen gewohnte Eigenschaften fehlen oder weniger komfortabel sind. Besonders vermisst wird zurzeit noch ein komfortabler Debugger. Dieser ist allerdings in Arbeit.

**Modularität und Portabilität:** Heurisko ist hauptsächlich in ANSI/ISO C und in manchen Modulen in ANSI/ISO C++ geschrieben. Eine Kernversion mit der vollständigen Bildverarbeitungsfunktionalität, ermöglicht durch die Modularisierung, kommt völlig ohne Elemente einer graphischen Benutzeroberfläche aus. Damit kann man heurisko als ein höchst portables Programmsystem betrachten, was durch weitgehend problemlose Portierungen bereits mehrfach unter Beweis gestellt wurde. Das Konzept der Erweiterungsmodule ermöglicht es Anwendern und Drittanbietern, eigene Erweiterungen für heurisko zu erstellen und anzubieten.

Damit ist heurisko insgesamt ein zukunftsfähiges und effizientes System zur Entwicklung von Bildverarbeitungsanwendungen. Das hat auch der jahrelange Einsatz dieses Paketes als Hauptwerkzeug in der Bildverarbeitungsgruppe im IWR der Universität Heidelberg gezeigt. Es folgt eine kurze Zusammenstellung der Hauptentwicklungsziele für die Zukunft.

- **kurzfristige Ziele:** Es soll ein ähnlich komfortabler Debugger angeboten werden, wie viele ihn z. B. von Microsofts Visual Studio her gewohnt sind.

- **mittelfristige und langfristige Ziele:** Es sollen intelligente selbstlernende Operatoren entwickelt werden, die immer größere Teilaufgaben unter Vorgabe klarer Kriterien selbständig optimal lösen. Beispiele sind eine optimale Kantendetektion oder Orientierungsanalyse und die Auswahl optimaler Merkmale zur Klassifizierung. Low-Level-Bildverarbeitungsoperatoren sind für eine flexibel einsetzbare Bildverarbeitungsbibliothek unverzichtbar. Aber viele Probleme in der Bildverarbeitung wiederholen sich, so dass es sich lohnt, für diese High-Level-Bildverarbeitungsoperatoren zu entwickeln, die es dem Anwender erheblich einfacher machen, solche Standardfälle schnell zu lösen.

Das bedeutet, dass das Grundgerüst von heurisko steht und dass ein Ausbau im Sinne einer Vereinfachung der Anwendung von heurisko das weitere Hauptentwicklungsziel ist.

# 8 Literatur und andere Quellen


[ABW] Internetadresse: http://www.abw-3d.de

[Aho et. al. 1974] Alfred V. Aho, John E. Hopcroft und Jeffrey D. Ullmann: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[Bässmann und Besslich 1991] Alfred V. Aho, John E. Hopcroft und Jeffrey D. Ullmann: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[Besslich und Lu 1990] Philipp W. Besslich und Tian Lu: *Diskrete Orthogonaltransformationen: Algorithmen und Flussgraphen für die Signalverarbeitung*. Springer-Verlag, Berlin, 1990.

[Blahut 1985] Richard E. Blahut: Fast Algorithms for Digital Image Processing. Addison-Wesley, Reading, 1985.

[Bracewell 1986] Ronald N. Bracewell: *The Fourier Transform and Its Applications*. McGraw-Hill, New York, 2. Auflage, 1986.

[Cormen et. al. 2001] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms*. 2. Auflage, MIT Press, Cambridge, 2001.

[CVB] Internetadresse: http://www.commonvisionblox.de

[Eckl 2004] Andreas Eckl: *Entwicklung von Messprozeduren und –aufbauten zur Bestimmung der Güteparameter von digitalen Kamerasystemen*. Diplomarbeit, Fachhochschule Regensburg, 2004.

[Eisele 2002] Heiko Eisele: *Automated defect detection and evaluation in X-Ray CT images*. Dissertation, Universität Heidelberg, 2002.

[FFTW] Internetadresse: http://www.fftw.org

[FOLDOC] *Free on-line dictionary of computing*. Imperial College London, Faculty of Engineering, Department of Computing.

[Frigo und Johnson 1998] Matteo Frigo und Steven G. Johnson: *FFTW: An Adaptive Software Architecture for the FFT*. Im Tagungsband der ICASSP 1998 (Band 3, S. 1381-1384).

[Gleisinger 2000] Reinhold Gleisinger: *Entwurf und Implementation eines objektorientierten Inspektors für multidimensionale wissenschaftliche Daten*. Diplomarbeit, FernUniversität Hagen, 2000.

[HALCON] Internetadresse: http://www.halcon.de

[heurisko] Internetadresse: http://www.heurisko.de

[Hennessy und Patterson 1996] John L. Hennessey und David A. Patterson: *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann, San Francisco, 2. Auflage, 1996.

[Herrmann 1996] Helmut Herrmann: *Eine vergleichende Studie zu der HISC-Rechnerarchitektur des Imagine-Rechners*. Diplomarbeit, FernUniversität Hagen, 1996.

[Holub 1990] Allen I. Holub: *Compiler Design in C*. Prentice Hall, Englewood Cliffs, 1990.

[Intel] Intel: *Hyper-Threading Technology*. Ein im Internet veröffentlichter Artikel der Firma Intel, Santa Clara, http://www.intel.com/technology/hyperthread.

[IUE] Internetadresse: http://www.isbe.man.ac.uk/research/iue/flier.html

[Jähne et. al. 1995] Bernd Jähne, Robert Massen, Bertram Nickolay und Harald Scharfenberg: *Technische Bildverarbeitung – Maschinelles Sehen*. Springer, Berlin, 1995.

[Jähne und Herrmann 1999] Bernd Jähne und Helmut Herrmann: *Multimedia Architectures*. In: Bernd Jähne, Horst Haußecker und Peter Geißler (Hrsg.): *Handbook of Computer Vision and Applications*, Bd. 3. Academic Press, San Diego, 1999.

[Jähne 2002] Bernd Jähne: *Digitale Bildverarbeitung*. Springer, Berlin, 5. Auflage, 2002.

[Jähne 2004] Bernd Jähne: *Practical Handbook on Image Processing for Scientific and Technical Applications*. CRC Press, Boca Raton, 2. Auflage, 2004.

[Jain et. al. 1995] Ramesh Jain, Rangachar Kasturi, Brian G. Schunck: *Machine Vision*. McGraw-Hill, New York, 1995.

[Khoros] Internetadresse: http://www.khoral.com

[Klette und Zamperoni 1992] Reinhard Klette and Piero Zamperoni: *Handbuch der Operatoren für die Bildverarbeitung*. Vieweg, Braunschweig, 1992.

[Köthe 1999] Ullrich Köthe: *Reusable Software in Computer Vision*. In: Bernd Jähne, Horst Haußecker und Peter Geißler (Hrsg.): *Handbook of Computer Vision and Applications*, Bd. 3. Academic Press, San Diego, 1999.

[Köthe 2000] Ulrich Köthe: *Genetische Programmierung für die Bildverarbeitung*. Dissertation Universität Hamburg, 2000.

[LabView] Internetadresse: http://www.ni.com/labview

[LAPACK] Internetadresse: http://www.netlib.org/lapack

[LINPACK] Internetadresse: http://www.netlib.org/linpack

[Malamas et. al. 2003] Elias N. Malamas, Euripides G. M. Petrakis, Michalis Zervakis, Laurent Petit, Jean-Didier Legat: *A survey on industrial vision systems, applications and tools*. In: Image and Vision Computing 21 (2003), S. 171-188, Elsevier.

[MathWorks 2002] The MathWorks, Inc.: *Accelerating MATLAB: The MATLAB JIT-Accelerator*. Technology Backgrounder, 2002.

[MATLAB] Internetadresse: http://www.mathworks.com

[Müller-Schloer 1991] C. Müller-Schloer, E. Schmitter (Herausgeber): *RISC-Workstation-Architekturen*. Springer-Verlag, Berlin, 1991.

[Musser und Stepanov 1989] D. Musser und A. Stepanov: *Generic Programming*. In: 1st International Joint Conference of ISSAC-88 and AAECC-6. Lecture Notes in Computer Science, Bd. 358. Springer, Berlin, 1989.

[Musser und Stepanov 1994] D. Musser und A. Stepanov: *Algorithm-Oriented Generic Libraries*. In: Software – Practice and Experience, 24(7), S. 623 – 642, 1994.

[Nikolaidis und Pitas 2001] Nikos Nikolaidis and Ioannis Pitas: *3-D Image Processing Algorithms*. Wiley, New York, 2001.

[OpenCV] Internetadresse: http://www.intel.com/research/mrl/research/opencv

[Ousterhout 1998] John K. Ousterhout: *Scripting: Higher Level Programming for the 21st Century*. In: IEEE Computer magazine, März 1998.

[Parker 1997] James R. Parker: *Algorithms for Image Processing and Computer Vision*. Wiley, New York, 1997.

[Pitas 1993] Ioannis Pitas: *Digital Image Processing Algorithms*. Prentice Hall, New York, 1993.

[Press et. al. 1992] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, 1992.

[QT] Internetadresse: http://www.trolltech.com

[Ritter und Wilson 2001] Gerhard X. Ritter und Joseph N. Wilson: *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, Boca Raton, 2. Auflage, 2001.

[Sid-Ahmed 1995] Maher A. Sid-Ahmed: Image Processing: *Theory, Algorithms, and Architectures*. McGraw-Hill, New York, 1995.

[Stelz 2003] Rupert Stelz: *Standards für digitale Kameras; IEEE 1394, Camera Link und Alternativen: Praktische Erfahrungen und Zukunftsperspektiven*. In: 23. Heidelberger Bildverarbeitungsforum, Stuttgart, 2003.

[TargetJr] Internetadressen: http://www.esat.kuleuven.ac.be/~targetjr/ und http://www.targetjr.org

[Tkl/Tk] Internetadressen: http://www.tcl.tk/doc/compiler.html

[Umbaugh 1998] Scott E. Umbaugh: *Computer Vision and Image Processing: a practical approach using CVIPtools*. Prentice Hall, Upper Saddle River, 1998.

[VGL] Internetadresse: http://www.volumegraphics.com

[VIGRA] Internetadresse: http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra

[Voss und Süße 1991] Klaus Voss und Herbert Süße: *Praktische Bildverarbeitung*. Hanser-Verlag, München, 1991.

[VTK] Internetadresse: http://public.kitware.com/VTK

[VXL] Internetadresse: http://vxl.sourceforge.net

[VXL Book] Internetadresse: http://paine.wiau.man.ac.uk/pub/doc_vxl/books/core/book.html

[Wagner 2000] Thomas Wagner: *Automatische Konfiguration von Bildverarbeitungssystemen* (Habilitationsschrift Universität Erlangen-Nürnberg). Shaker Verlag, Aachen, 2000.