TUHH
Technische Universität Hamburg-Harburg

Vision Systems

Prof. Dr.-Ing. Rolf-Rainer Grigat

Softwarekonzepte für wiederverwendbare Bildverarbeitungsmethoden am Beispiel der Automatisierung eines Befüllroboters

Diplomarbeit

Marco Budde

Dezember 2000

Erklärung

Hiermit erkläre ich, daß die vorliegende Arbeit von mir selbständig und nur unter Verwendung der aufgeführten Hilfsmittel erstellt wurde.

Harburg, den 11. Dezember 2000

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Ziel dieser Diplomarbeit, die in Koorporation mit der Firma Feige entstanden ist, war die Verbesserung eines bestehenden Roboters zur automatischen Befüllung von Fässern.

Durch den Befüllroboter werden mit Hilfe eines Tragkettenförderers Paletten transportiert, auf denen sich in beliebiger Anordung Fässer bzw. Kanister befinden, die befüllt werden sollen. Eine solche Palette ist in Abbildung 1.1 zu sehen.

Während früher solche Befüllungen manuell gesteuert wurden, ist dieses im Zeitalter der Rationalisierung undenkbar. Der bestehende Befüllroboter verfügt deshalb bereits heute über eine automatische Erkennung der Spundlöcher der zu befüllenden Behälter, so daß die Maschine eine Palette mit Behälter völlig selbständig befüllen kann. Allerdings stammt dieses Erkennungssystem noch aus einer Zeit, wo die Rechenleistung und der zur Verfügung stehende Speicher sehr begrenzt waren, so daß ein relativ primitives Erkennungskonzept eingesetzt wird.

Dieses führt dazu, daß die Erkennung nicht immer genau funktioniert und z.B. Bereiche zwischen den Behältern als Spundlöcher erkannt werden. Dieses ist insbesondere deshalb sehr problematisch, da mit diesen Anlagen teilweise brennbare, toxische und umweltschädliche Stoffe wie z.B. Benzin abgefüllt werden. Wenn diese Stoffe nicht in die Fässer sondern daneben abgefüllt werden, kann das im harmlosesten Fall zu viel Arbeit und im schlimmsten Fall zu Gefahren für Mensch und Umwelt führen. Hierbei ist vor allem zu bedenken, daß diese Maschinen oft im EX-Bereich eingesetzt werden.

Es entstand deshalb von Seiten der Firma Feige der Wunsch, ein völlig neues Bilderkennungssystem zu erforschen und zu implementieren, das mit Hilfe der heute zur Verfügung stehenden Leistung moderner Rechner eine wesentlich sicherere Erkennungsgenauigkeit erreichen sollte. Als Erkennungsalgorithmen sollten die bereits von Klaus Wiehler in seiner Diplomarbeit für eine ähnliche Problemstellung aus der Medizin eingesetzten Algorithmen Verwendung finden [Wie96].

Ein Problem für die bestehende und die zu entwickelnde Bildverarbeitung stellt die große Anzahl von unterschiedlichen Objekten dar, die erkannt werden müssen. Es handelt sich zwar immer um Fässer oder Kanister, allerdings existieren diese mit vielen verschiedenen Abmessungen und Farben. Auch die verwendeten Materialien reichen von z.B. Plastik bis zu Metall, wobei all diese Materialien verschiedene Reflexionsgrade haben. Es gibt außerdem verschiedene Spundlochverschlüsse. Teilweise stehen die Behälter mit oder ohne Verschluß auf der Palette.

Durch die unterschiedlichen Größen der Fässer stehen auch nicht immer die gleiche Anzahl von Fässer auf einer Palette. Es konnte bei der Entwicklung allerdings davon ausgegangen werden, daß sich auf einer Palette immer nur Fässer von einem Typ befinden. Dieses führt dazu, daß sich die maximale Anzahl von Fässern eines bestimmten Typs, die sich auf einer Palette befinden können, berechnen läßt, so daß eine obere Schranke existiert. Allerdings muß die Maschine auch automatisch damit zurechtkom-

Abbildung 1.1: Palette mit Fässern

men, daß sich weniger Fässer auf der Palette befinden. Dieses kann z.B. dann passieren, wenn nur eine bestimmte Menge eines Stoffes verfüllt werden kann bzw. soll.

Zur Zeit ist die Anpassung der Anlage an neue Faßtypen sehr aufwendig, da hierfür Einstellungen an der Maschine vor Ort von einem Servicetechniker durchgeführt werden müssen. Dieses kann aus zwei Gründen zu einem Problem werden. Zum einen werden die Anlagen weltweit eingesetzt, so daß viel Zeit durch die Anreise des Servicetechnikers verloren geht. Dieses ist natürlich für die Kunden der Firma Feige auch kein ganz günstiges Unterfangen. Das zweite Problem ist, daß einige Kunden fast täglich andere Faßtypen befüllen möchten. Hier ist es fast unmöglich, daß die Einstellungen jedesmal vor Ort vorgenommen werden müssen.

Aus diesem Grund bestand ein weiterer Teil der Diplomarbeit darin, eine Fernwartung der Anlage zu ermöglichen. Durch die Fernwartung sollte es dann möglich sein, die Bilderkennung einer irgendwo auf der Erde installierten Anlage vom Firmensitz der Firma Feige aus zu konfigurieren. Hierbei sollte nicht auf eine proprietäre Lösung zurückgegriffen werden. Vielmehr war das Ziel, die Fernwartung über eine beliebige IP-Verbindung (Internet, Telefonleitung usw.) zu ermöglichen.

1.2 Struktur der Anlage

In Abbildung 1.2 ist die Systemübersicht der ganzen Anlage zu sehen, wie sie am Anfang beschlossen wurde. Im Verlauf der Entwicklung wurden einige Teile geändert, dazu später noch mehr.

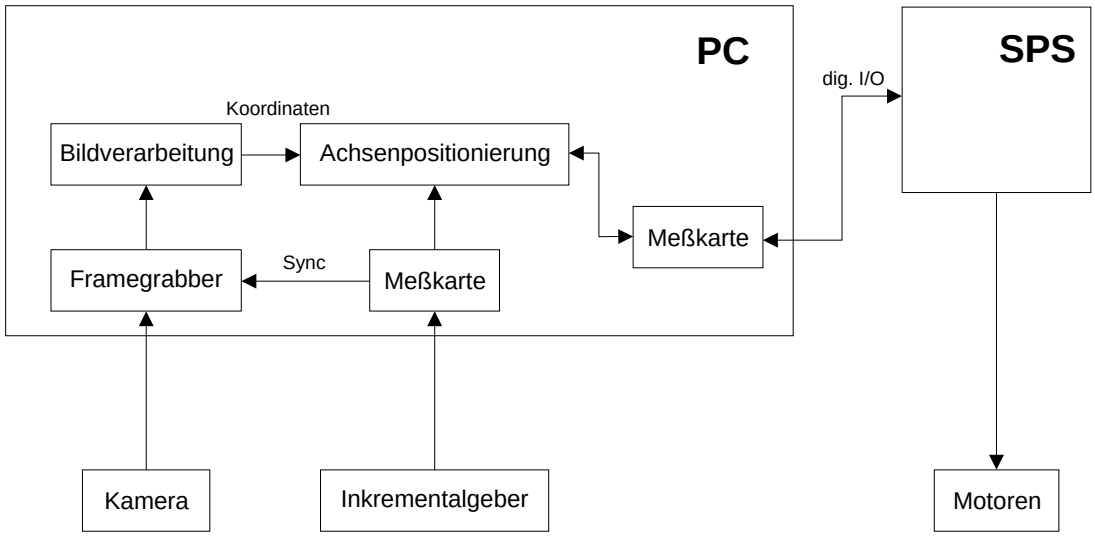


Abbildung 1.2: Ursprüngliche Systemübersicht

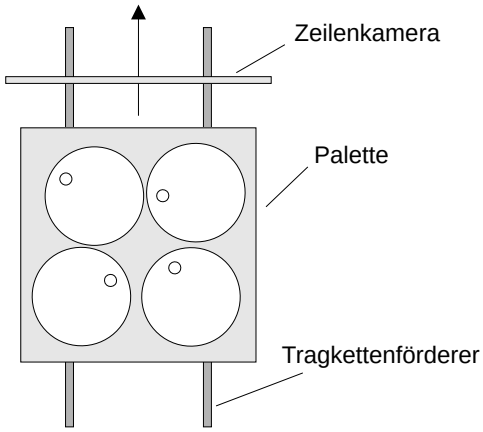


Abbildung 1.3: Bildaufnahme in der Anlage

Der PC soll gemäß der Systemübersicht zwei Aufgaben wahrnehmen: die Bildverarbeitung und die Achsenpositionierung. Die Bildverarbeitung liest das Bild einer Palette mit Hilfe einer Kamera und eines Framegrabbers ein, berechnet die Koordinaten der Spundlöcher und liefert diese an die Achsenpositionierung. Die Achsenpositionierung, die von der Firma Feige erstellt werden sollte, steuert dann die Palette und den Befüllrüssel an die von der Bildverarbeitung errechneten Koordinaten. Die Kommunikation zwischen PC und SPS soll dabei über mehrere digitale I/O-Leitungen ablaufen. Über zwei Inkrementalgeber in der Anlage kann die Achsenpositionierung die momentane Position der Palette und des Befüllrüssels ermitteln.

In Abbildung 1.3 ist der Aufbau der Bilderfassung der Anlage zu erkennen. Die Palette mit den Fässern bewegt sich auf einem Tragkettenförderer unter einer Zeilenkamera hindurch. Eine solche Zeilenkamera verfügt gegenüber einer Flächenkamera nur über eine einzige Bildzeile. Da sich die Palette unter der Kamera hindurch bewegt, kann durch regelmäßiges Auslesen der einen Zeile der Kamera ein zweidimensionales Bild der Palette erzeugt werden. Eine solche Zeilenkamera hat gegenüber einer normalen Flächenkamera den Vorteil, daß man zu einem viel niedrigeren Preis ein viel höhere Auflösung erzielen kann.

Wichtig ist aber natürlich, die Zeile immer zur richtigen Zeit auszulesen, um kein verzerrtes Bild zu erhalten. Aus diesem Grund wird der Inkrementalgeber, der die Bewegung des Tragkettenförderers mißt, als Trigger für das Auslesen der Kamera verwendet.

Die Bildverarbeitung sollte auf einem handelsüblichen PC erfolgenden. Als Betriebssystem wurde Linux gewählt, da dieses im Gegensatz zu dem weit verbreiteten Betriebssystem Windows viele Eigenschaften aufweist, die für dieses Projekt sehr wichtig waren: Stabilität, gute Netzwerkunterstützung, ausgereiftes Paketkonzept, Multiuser-fähig und gutes Preis-/Leistungsverhältnis.

Bereits bei der bestehenden Anlage verfügte die SPS über eine Fernwartungsschnittstelle per Modem. Da man ungerne zwei Fernwartungsverbindungen haben wollte, sollte nach Möglichkeiten gesucht werden, wie sich die Fernwartungsschnittstelle der SPS in die des PCs integrieren läßt.

Kapitel 2

Linux-Treiber

2.1 Motivation

Laut Lastenheft der Firma Feige sollte der PC nicht nur für die Bildverarbeitung und die Fernwartung zuständig sein, sondern auch die Achsenpositionierung übernehmen.

Hierfür sollten die beiden in der Anlage eingebauten Inkrementalgeber, die die Bewegung des Tragkettenförderers und des Befüllrüssels registrieren, mit dem PC verbunden werden.

Die Inkremtalgeber werden über jeweils zwei differentielle Kanäle (A/B) mit dem Auswertungssystem verbunden. Wird ein Inkrementalgeber gedreht, so erzeugt er regelmäßig Impulse auf den beiden Kanälen, siehe Abbildung 2.1. Durch die spezielle Codierung der Signale ist es möglich, zu erkennen, in welche Richtung der Inkrementalgeber gedreht wird. Dieses ist wichtig, da die Palette mit dem Tragkettenförderer bei der eigentlichen Befüllung in beide Richtung bewegt werden muß. Die meisten Systeme zur Auswertung der vom Inkrementalgebern gelieferten Signale besitzen deshalb einen Zähler, dessen Inhalt je nach Drehrichtung verkleinert oder vergrößert wird.

Neben dem Anschluß der Inkrementalgeber mußte eine Möglichkeit gefunden werden, der SPS der Anlage mitzuteilen, die einzelnen Motoren an- bzw. auszuschalten. Zusammen mit den Inkrementalgebern hätte der PC dann die Möglichkeit gehabt, den Befüllrüssel in beiden Achsen genau zu positionieren. Das Lastenheft sah vor, diese Kommunikation über digitale I/O-Leitungen zwischen SPS und PC zu realisieren. Für jeden Befehl sollte es eine eigene I/O-Leitung geben.

Es mußte deshalb eine Meßkarte für den PC gefunden werden, die sowohl Inkrementalgeber unterstützt wie auch digitale Ein- und Ausgänge besitzt. Da der PC unter Linux laufen sollte, wurde außerdem eine Karte benötigt, für die ein Linux-Treiber existiert.

Die Suche nach einer solchen Karte gestaltete sich als sehr schwierig. Es wurde keine einzige Karte gefunden, die die benötigten Funktionen besaß und für die ein Linux-Treiber verfügbar war. Es wurde deshalb beschlossen, die Suche auf Karten ohne Treiber für Linux auszuweiten und einen solchen Treiber dann selbst zu entwickeln.

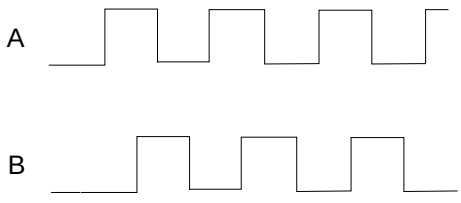


Abbildung 2.1: Signale eines Inkrementalgebers

Selbst nach dieser Erweiterung der Suche konnten nur drei passende Karten gefunden werden, da es nur sehr wenige Karte für den Anschluß von Inkrementalgebern an den PC gibt:

- Kolter 3x24BIT U/D-LCA
- ADDI-DATA APCI-1710
- Sensoray Model 626

Für die erste Karte existierten Beispielsourcen, wie man diese unter Linux aus dem User Mode ansteuern kann. Allerdings unterstützte diese Karte lediglich Inkrementalgeber, so daß eine weitere Meßkarte für die digitalen Ein- und Ausgänge benötigt worden wäre. Die Entwicklung eines Treibers für diese Karte wäre sehr einfach gewesen, da die Karte nur über sehr wenige Funktionen verfügt. Allerdings hatte diese Karte einen entscheinden Nachteil: sie verwendet den ISA-Bus. Da immer weniger PC-Systeme heute überhaupt noch über einen ISA-Bus verfügen, wäre dieses langfristig für die Firma Feige zu einem Problem geworden, so daß diese Karte aus den engeren Wahl entfernt werden mußte.

Die zweite Karte war sehr viel leistungsfähiger und flexibler. Sie besaß vier »Prozessoren«, in die verschiedene Funktionen per Software geladen werden konnten. So konnte ein »Prozessor« für digitales I/O, als Watchdog, als Auswertung für Inkrementalgeber usw. reserviert werden. Man konnte die Karte also mehr oder weniger vollständig an die eigenen Bedürfnisse anpassen. Probleme hätte allerdings die Entwicklung eines Treibers gemacht, da der Hersteller nicht bereit war, die notwendigen Informationen zur Programmierung dieser komplexen Karte herauszurücken.

Da auch die zweiten Karte nicht verwendet werden konnte, fiel die Wahl auf die dritte Karte. Diese von der Firma Sensoray hergestellte PCI-Karte verfügt über Funktionen wie: Anschluß von Inkrementalgebern, eine Watchdog, digitale Ein- und Ausgänge und A/D-D/A-Wandler. Die Entscheidung fiel vor allem deshalb auf die Karte, da der Hersteller das Handbuch mit den Informationen zur Programmierung dieser Karte im Internet bereitstellte [Sen99]. Mit Hilfe dieses Handbuches sollte die Entwicklung, wie jedenfalls am Anfang vermutet wurde, eines Treibers für Linux kein großes Problem darstellen. Später zeigte sich allerdings, daß diese Einschätzung nicht ganz richtig war.

2.2 Grundlagen

In diesem Abschnitt werden einige wichtige Grundlagen zu Linux vermittelt, die für das weitere Verständnis notwendig sind. Weitere Information zu diesen Themen sind in [BBD+99, SG97] zu finden.

2.2.1 Module

Der Linux Kernel kennt zwei verschiedene Arten von Treibern.

In der Anfangszeit von Linux mußten alle Treiber direkt im Kernel Source enthalten sein und mit dem eigentlichen Kernel zusammen kompiliert werden. Sie waren nach der Übersetzung fest mit dem Rest des Kernels verbunden. Dieses Konzept hat diverse Nachteile. So läßt sich ein Treiber nur durch Änderungen am Source des Linux Kernels in diesen einfügen. Dieses ist kein Problem, wenn der Treiber direkt mit dem Kernel Source ausgeliefert wird; aber dieses ist in der Praxis z.B. bei neuen Treibern nicht der Fall. Ein solcher Treiber muß dann als Patch ausgeliefert werden, der in den Kernel Source eingespielt werden und schließlich mit dem Kernel zusammen neu übersetzt werden muß. Dieses Vorgehen ist nicht nur umständlich, sondern es kommt zu Problemen, wenn man mehrere solcher Patches gleichzeitig einspielen möchte. Außerdem lassen sich Treiber dann nur im Source weitergeben, was für kommerzielle Anbieter ein Problem darstellen kann.

Um diese Probleme zu vermeiden, wurde das Konzept der Module eingeführt. Ein Modul enthält jeweils einen Treiber. Diese Treiber werden nicht mehr fest in den Kernel einkompiliert. Vielmehr ist jedes

Modul in einer eigenen externen Datei abgelegt. Wird ein Treiber benötigt, kann er zur Laufzeit beliebig oft in den Kernel geladen und aus ihm entfernt werden. Dieses sehr moderne Konzept bietet sonst kein anderes weit verbreitetes Betriebssystem. Zwar legen auch andere Betriebssysteme ihre Treiber in externen Dateien ab, diese können aber nur beim Booten geladen und während der Laufzeit garnicht entladen werden. Es gibt bei Linux sogar Ansätze, Module bei Bedarf automatisch zu laden und am Ende der Benutzung auch automatisch wieder zu entfernen.

Die Möglichkeit, den Treiber während der Laufzeit laden zu können, bietet diverse Vorteile. Für den Anwender bestehen die Vorteile vor allem darin, daß er bei Problemen mit einem Treiber diesen schnell entladen und vielleicht mit anderen Parametern neu laden kann. Für den Entwickler von Treibern bieten die Module den Vorteil, daß der Rechner nicht jedesmal neu gebootet werden muß, wenn ein modifizierter Treiber getestet werden soll. Vielmehr reicht es, den alten Treiber zu entladen und den neuen zu laden.

2.2.2 Devices

Es gibt verschiedene Schnittstellen, über die ein User Mode Programm mit einem Treiber Daten austauschen kann. Die meisten Treiber bieten ein oder mehrere sogenannte Devices an [BBD+99]. Hierbei handelt es sich um virtuelle Dateien im Verzeichnis »/dev«, auf die mit den ganz normalen Schreib- und Leseoperationen für Dateien wie z.B. »read()« und »write()« zugegriffen werden kann.

Schreibt man z.B. Daten auf das Device »/dev/ttyS0«, so speichert der Kernel diese nicht in dieser Datei, sondern leitet sie an den Treiber für die erste serielle Schnittstelle des PCs weiter. Die Zuordnung, an welchen Treiber die Daten vom Kernel geschickt werden, geschieht durch die sogenannten Major- und Minornummern, die jedes Device besitzt.

Über die Majornummer wird der Treiber spezifiziert. Nun kann es ja vorkommen, daß ein Treiber mehrere Devices anbieten möchte. Diese werden dann über die Minornummer spezifiziert.

Problematisch ist bei beiden Nummern, daß sie nur einmal vorkommen dürfen. Es darf also nicht zwei Devices mit der gleichen Majornummer geben. Dieses Problem läßt sich bei den Standardtreibern noch leicht durch Koordination lösen, bei der großen Anzahl von externen Modulen wird dieses aber immer schwieriger; das Problem wird dadurch verschärft, daß es nur 256 Majornummern gibt.

Das Konzept der Abbildung als virtuelle Dateien hat in vielen Bereichen sehr große Vorteile. Die Anwender und die Programmierer können auf völlig unterschiedliche Geräte immer mit den gleichen Programmen und Befehlen zugreifen. So ist es z.B. für das Backup-Programm »tar« egal, ob sich hinter dem Device, auf das die Daten gesichert werden sollen, ein Streamer, ein Diskettenlaufwerk, eine Wechselplatte oder eine Festplatte befindet. Die Schnittstelle zwischen Programm und Treiber ist immer die gleiche.

Ein anderes gutes Beispiel, um die Vorteile dieses Konzeptes erkennen zu können, ist der Treiber für ISDN-Karten. Dort existiert z.B. ein Device, das die AT-Befehlsschnittstelle eines normalen Modems emuliert. Folglich können die meisten Programme, die für analoge Modems entwickelt worden sind, ohne Veränderungen auch mit ISDN-Karten verwendet werden.

2.2.3 ioctl

Eine weitere Möglichkeit, über die ein User Mode Programm mit dem Kernel Daten austauschen kann, stellt der Befehl »ioctl()« dar.

Im Gegensatz zu den Devices muß der Anwender jetzt nicht mit irgendwelchen Funktionen auf virtuelle Dateien zugreifen, sondern er übergibt einfach eine Befehlsnummer und die Daten an die Funktion »ioctl()« und teilt ihr mit, für welchen Treiber diese gedacht sind.

Ob man jetzt Devices oder ioctls zur Kommunikation mit dem Kernel benutzt, ist letztendlich eine reine Design-Entscheidung des jeweiligen Entwicklers. Üblicherweise benutzt man Devices für den

Austausch von Daten und ioctls für die Konfiguration des Treibers.

Der Treiber für die serielle Schnittstelle überträgt z.B. die Daten über ein Device, während z.B. die Geschwindigkeit der Schnittstelle per ioctl eingestellt wird.

2.2.4 Echtzeit

Standardmäßig ist Linux wie eigentlich alle andere Multitasking Betriebssysteme kein Echtzeit-Betriebssystem [Gal95]. Linux garantiert also nicht, daß ein Programm nach einer bestimmten Zeit Rechenzeit erhält. Auch die Reihenfolge, in der die Programme an die Reihe kommen, ist nicht garantiert. Dieses ist ein großer Unterschied im Vergleich zu Singletasking Betriebssystemen wie z.B. MS-DOS, wo man jederzeit die komplette Kontrolle hat, wann was passiert.

Für die meisten Anwendungen ist der Verlust dieser Echtzeit-Garantien kein Problem. Im schlimmsten Fall »fühlt« sich die Anwendung etwas langsamer an. Bei einer Anwendung wie der Achsenpositionierung, die für dieses Projekt zu entwickeln war, ist der Verlust der Garantien allerdings ein sehr großes Problem.

Dieses läßt sich leicht verdeutlichen. Man stelle sich vor, der Tragkettenförderer soll die Palette an eine bestimmte Position fahren. Hierzu muß er über die SPS den Motor einschalten. Um den richtigen Zeitpunkt abzupassen, an dem der Motor wieder abgeschaltet werden muß, würde die Achsenpositionierung den Inkrementalgeber überwachen. Da Linux aber keine Echtzeit garantiert, kann es passieren, daß das Programm den Motor startet und dann erst nach 30 Sekunden wieder Rechenzeit erhält. Inzwischen ist aber die Palette vom Ende des Tragkettenförderers gefallen. Man sieht also sehr schnell, daß für solche Anwendungen Echtzeit-Garantien unbedingt erforderlich sind.

Da viele Anwender schon ähnliche Probleme hatten, haben einige Entwickler Lösungen entwickelt, mit denen man Linux zu einem Echtzeit-Betriebssystem machen kann:

- RT-Linux
- RTAI
- KURT

Alle drei Lösungen liegen in Form von Patches für die normalen Linux Kernels vor und sind wie Linux selbst frei verfügbar. Allerdings wurde inzwischen der Ansatz von RT-Linux von dessen Entwickler zum Patent angemeldet. Am weitesten verbreitet ist zur Zeit RT-Linux.

Die Idee von RT-Linux ist, die Echtzeitfähigkeit nicht direkt in den Linux Kernel einzufügen, da dieses schwierig wäre und einen nicht unbeträchtlichen Overhead für normale Anwendungen produzieren würde. Statt dessen ist RT-Linux ein kleines Echtzeit-Betriebssystem. Ein Task dieses Betriebssystems ist der normale Linux Kernel [Yod97].

Andere Echtzeitprozesse können zur Laufzeit als Modul geladen werden. Der Prozeß des Linux Kernels hat dabei die niedrigste Priorität. Für Hardware, auf die innerhalb eines Echtzeitprozesses zugegriffen werden soll, wird ein spezieller RT-Linux-Treiber benötigt. Da die Achsenpositionierung als Echtzeitprozeß implementiert werden sollte, mußte der Treiber für die Meßkarte am Ende für RT-Linux vorliegen. Da das Schreiben von Treibern für RT-Linux nicht sonderlich gut dokumentiert ist, wurde beschlossen, zuerst einen Treiber für den Linux Kernel selbst zu schreiben, um die Karte genauer kennenzulernen. Dieser sollte dann später portiert werden.

2.3 Implementation des Treibers

2.3.1 Philips SAA7146

Am Anfang der Entwicklung des Treibers für die Meßkarte lag nur das Programmierhandbuch nicht aber die Karte selbst vor. Es wurde beschlossen, trotzdem bereits mit der Implementation des Treibers anzufangen und diesen dann erst später mit der Hardware zu testen.

Das vorliegende Handbuch erweckte den Eindruck, daß sich die Karte über einige in diesem Handbuch beschriebene I/O-Ports direkt steuern lassen würde. So ist z.B. im Handbuch zu lesen, daß sich die ersten 16 digitalen I/O-Leitungen über den I/O-Port mit dem Offset 0x48 ansteuern lassen. In der Regel beziehen sich solche Offsets auf den Basis-I/O-Port, den eine PCI-Karte über ihre Konfigurationsregister an das BIOS und das Betriebssystem meldet. Entsprechend wurde der Treiber so ausgelegt, daß die einzelnen Funktionen auf die I/O-Ports zugreifen, die sich aus Basis-I/O-Port plus Offset berechnen.

Als die Karte dann eingetroffen war und mit dem bis dahin implementierten Treiber getestet wurde, funktionierte keine der implementierten Funktionen. Es ließ sich z.B. keiner der digitalen Ausgänge ein- oder ausschalten.

Nachdem kein Fehler im Source Code gefunden werden konnte, war es sehr wahrscheinlich, daß die Programmierinformationen entweder falsch oder lückenhaft waren. Es entstand deshalb die Idee, daß sich die Offset-Adressen eventuell doch nicht auf den Basis-I/O-Port der Karte beziehen würden. Deshalb wurde im nächsten Schritt, die Karte genauer untersucht.

Jeder IC, der direkt mit dem PCI-Bus verbunden ist, besitzt eine Device ID und eine Vendor ID. Es wäre also eigentlich logisch gewesen, daß die Karte die Vendor ID von Sensoray und die Device ID für das Model 626 gehabt hätte. Laut Handbuch identifiziert sich die Meßkarte mit:

Device ID	0x7146
Vendor ID	0x1131

Linux enthält in dem Kernel Header File »include/linux/pci.h« eine Liste der bekannten Vendor und Device IDs. Eine Suche nach den beiden Werten ergab dann folgendes:

PCI_DEVICE_ID_PHILIPS_SAA7146	0x7146
PCI_VENDOR_ID_PHILIPS	0x1131

Die Karte kommuniziert also über den IC SAA7146 von Philips mit dem PCI-Bus. Das von der Internetseite von Philips besorgte Datenblatt [Phi98] dieses ICs weist diesen als »Multimedia bridge, high performance Scaler and PCI circuit« aus. Laut Datenblatt unterstützt der IC verschiedene Operationen für die Video- und Audioverarbeitung und bietet zusätzlich einige Interfaces:

- D1-Videoschnittstelle
- DMSD2-Videoschnittstelle
- Data Expansion Bus Interface (DEBI)
- digitale Audio I/O-Schnittstellen
- General Purpose I/O-Ports (GPI/O)
- I^2O Bus Master

Da die Meßkarte weder Video- noch Audiofunktion besitzt, wurde daraus geschlossen, daß der SAA7146 wohl auf dieser Karte die Funktion hat, den zweiten großen IC der Meßkarte mit dem PCI-Bus zu verbinden. Dieser zweite IC enthält wohl die eigentliche Funktionalität der Meßkarte, siehe Abbildung 2.2.

Auf der Internetseite von Sensoray fand sich dann noch der Source eines Treibers für das Echtzeit-Betriebssystem QNX. Die Funktionen der untersten Schicht dieses Treibers hatten alle die Zeichenkette

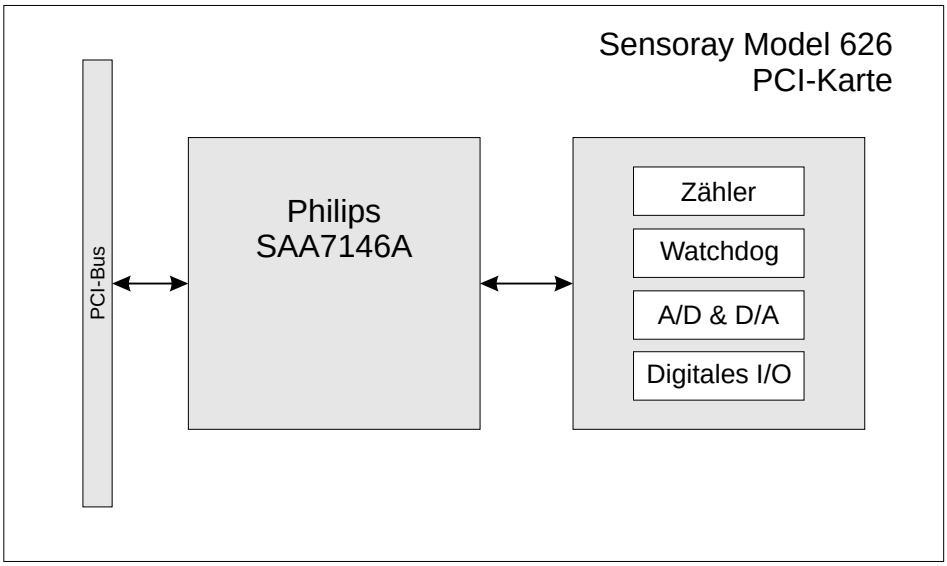


Abbildung 2.2: Die zwei ICs der Meßkarte Sensoray Model 626

»DEBI« im Funktionsnamen, so daß man daraus schließen konnte, daß der zweite IC über den DEBI-Bus mit dem SAA7146 verbunden ist.

Eine weitere Analyse des QNX-Treibers führte zu der Erkenntnis, daß sich die im Handbuch angegebenen Offsets nicht auf einen I/O-Port des PCs beziehen, sondern eine Adresse auf dem DEBI-Bus spezifizieren.

Es mußten also Funktionen entwickelt werden, die den SAA7146 passend initialisieren und die Möglichkeit bieten, bestimmte I/O-Ports des zweiten ICs über den DEBI-Bus zu lesen und zu schreiben. Da das Handbuch der Meßkarte keinerlei Informationen darüber hergab, wie genau der zweite IC mit dem SAA7146 verbunden ist, konnte die notwendige Konfiguration des SAA7146 nicht anhand des Datenblattes ermittelt werden. Als einzige Informationsquelle konnte der QNX-Treiber dienen.

Aus diesem Grund wurden die passenden Lowlevel-Funktionen des QNX-Treibers einfach nach Linux portiert. Es entstanden folgende Funktionen:

```
static void saa7146_irq_enable (void);
static void saa7146_irq_disable (void);
static void DEBI_init (void);
static void DEBI_close (void);
static unsigned short DEBI_read (unsigned short addr);
static void DEBI_write (unsigned short addr, unsigned short wdata);
```

Mit den ersten beiden Funktionen wird dem SAA7146 erlaubt bzw. verboten, dem PC über den PCI-Bus Interrupts zu schicken. Der zweite IC kann dann z.B. einen Interrupt auslösen, wenn sich der Wert eines digitalen Einganges der Meßkarte geändert hat. Leider ist es während der Entwicklung des Treibers nie gelunden, den Interrupt-Betrieb stabil zum Laufen zu bringen; dazu später noch mehr. Vermutlich wird der SAA7146 durch die beiden ersten Funktionen für den IRQ-Betrieb noch nicht korrekt initialisiert.

Die beiden nächsten Funktionen konfigurieren den SAA7146 und seine Bussysteme passend für die Meßkarte. Am Ende der beiden Funktionen wird mit einer der beiden ersten Funktionen der IRQ-Betrieb ein- bzw. ausgeschaltet, so daß die Karte nur dann Interrupts an den PC schicken kann, wenn der Treiber für die Karte geladen ist, so daß keine Rechenzeit mit der Behandlung von unnötigen Interrupts verschwendet wird.

Zum Lesen bzw. Schreiben eines 16 Bit-Wertes auf dem DEBI-Bus können die letzten beiden Funktionen benutzt werden. Beiden wird eine der im Handbuch der Meßkarte spezifizierten Offset-Adressen übergeben.

Nachdem der am Anfang entwickelte Treiber so umgeschrieben worden war, daß er nicht mehr direkt auf irgendwelche I/O-Ports des PCs schrieb, sondern die entsprechenden Funktionen für den DEBI-Bus benutzte, wurde ein weiterer Test des Treibers durchgeführt. Auch dieser verlief nicht erfolgreich. Nach dem Studium der Literatur zur Kernelprogrammierung kam dann die Vermutung auf, daß der SAA7146 nicht über I/O-Ports mit dem PC kommuniziert, sondern ein Fenster in den Speicher einblendet: Memory Mapped I/O. Dieses wurde ausprobiert, indem die DEBI-Funktionen entsprechend umgeschrieben wurden. Ein weiterer Test zeigt, daß diese Vermutung richtig war: mit dem Treiber konnten die digitalen Ausgänge der Meßkarte ein- und ausgeschaltet werden.

2.3.2 Watchdog

Aufbauend auf den implementierten DEBI-Funktionen wurden dann Funktionen zur Ansteuerung der Funktionalität des zweiten ICs implementiert. Der zweite IC unterstützt neben digitalen Ein- und Ausgängen, Zählern und A/D-D/A-Wandlern auch eine Watchdog.

Eine Watchdog ist insbesondere für Embedded Systems sehr interessant. Man stelle sich vor, ein Meßrechner befindet sich irgendwo am Nordpol und kann nur über eine Funkverbindung abgefragt werden. Sollte der Rechner aus irgendwelchen Gründen hängenbleiben, müßte ein Servicetechniker den beschwerlichen Weg zum Nordpol auf sich nehmen, um den Rechner neu zu booten. Ähnliche Probleme gibt es in vielen Bereichen. Eine Watchdog verfügt über einen Timer. Wird dieser nicht nach einer bestimmten Zeit immer wieder zurückgesetzt, bootet die Hardware einfach den Rechner neu.

Die Karte von Sensoray verfügt über eine Watchdog, wobei der Timer auf $\frac{1}{8}$, $\frac{1}{2}$, 1 oder 10 Sekunden eingestellt werden kann. Der Treiber enthält drei Lowlevel-Funktionen für die Ansteuerung der Watchdog:

```
static void s626_wtd_enable (void);
static void s626_wtd_clear (void);
static void s626_wtd_disable (void);
```

Mit »s626_wtd_enable()« wird die Watchdog eingeschaltet und der Timer auf 10 Sekunden konfiguriert. Im Gegensatz zu anderen Registern kann das der Watchdog nicht direkt beschrieben werden, sondern muß vorher über ein weiteres Register freigeschaltet werden. Dieses Konzept dient der Erhöhung der Sicherheit, da so vermieden wird, daß ein defektes Programm aus Versehen die Watchdog abschaltet.

Damit der Rechner von der PCI-Karte nicht neu gebootet wird, muß mindestens alle 10 Sekunden die Funktion »s626_wtd_clear()« aufgerufen werden, die den Timer der Watchdog zurücksetzt. Mit »s626_wtd_disable()« kann die Watchdog schließlich wieder abgeschaltet werden. Dieses ist z.B. dann notwendig, wenn der Treiber der Meßkarte entladen wird. Würde man die Watchdog nicht vorher abschalten, würde die Karte den Rechner trotz des nicht geladenen Treibers spätestens nach 10 Sekunden neu booten.

2.3.3 Digitale Ein- und Ausgänge

Die Meßkarte von Sensoray verfügt über 48 digitale Ein- und Ausgänge. Jeder dieser 48 Kanäle kann entweder als Eingang oder als Ausgang benutzt werden. Die ersten 40 Eingänge können so konfiguriert werden, daß sie eine fallende oder steigende Flanke erkennen. Bei Bedarf kann eine erkannte Flanke per Interrupt an den PC gemeldet werden. Aufgrund der Flankenerkennung und Interrupt-Unterstützung gibt es relativ viele Lowlevel-Funktionen:

```
static void s626_dio_set_irq (unsigned short a,
```

```
                                unsigned short b,
                                unsigned short c);
    static void s626_dio_set_edge_type (unsigned short a,
                                        unsigned short b,
                                        unsigned short c);
    static void s626_dio_set_edge_capture (unsigned short a,
                                           unsigned short b,
                                           unsigned short c);
    static void s626_dio_set_16_source (unsigned char value);
    static void s626_dio_read_edges (unsigned short *a,
                                     unsigned short *b,
                                     unsigned short *c);
    static void s626_dio_read_irq (unsigned short *a,
                                   unsigned short *b,
                                   unsigned short *c);
    static void s626_dio_reset_irq (unsigned short a,
                                    unsigned short b,
                                    unsigned short c);
    static void s626_dio_out (int nr, int status);
    static void s626_dio_in (unsigned short *a,
                             unsigned short *b,
                             unsigned short *c);
```

Mit »s626_dio_set_irq()« wird festgelegt, ob eine auf einem bestimmten Kanal erkannte Flanke einen Interrupt erzeugen soll. Wie bei allen anderen Funktionen auch kann die Entscheidung, ob ein Interrupt erzeugt werden soll oder nicht, für jeden Kanal getrennt getroffen werden.

Damit überhaupt Flanken erkannt werden, muß die Erkennung mit »s626_dio_set_edge_capture()« für die jeweiligen Kanäle eingeschaltet werden. Die Meßkarte kann sowohl positive wie auch negative Flanken erkennen. Welcher Typ erkannt werden soll, wird mit »s626_dio_set_edge_type()« festgelegt. Um zu überprüfen, auf welchen Kanälen Flanken erkannt worden sind, kann die Funktion »s626_dio_read_edges()« Verwendung finden. Soll überprüft werden, ob eine Flanke einen Interrupt erzeugt hat, kann »s626_dio_read_irq()« benutzt werden. Diese Funktion ist insbesondere im Zusammenhang mit Interrupt Sharing, welches von jedem Treiber für PCI-Karten beherrscht werden sollte, wichtig. Beim Interrupt Sharing muß sich beim Auftreten eines Interrupts feststellen lassen, welche PCI-Karte ihn erzeugt hat und welcher Treiber somit zuständig ist. Nachdem der Treiber die erkannten Flanken ausgewertet hat, kann mit »s626_dio_reset_irq()« die Flankenerkennung zurückgesetzt werden.

Die ersten sechs digitalen Eingängen können so konfiguriert werden, daß ihr Inhalt anzeigt, ob bei einem der sechs auf der Meßkarte enthaltenen Zähler ein Überlauf aufgetreten ist; zu den Zählern siehe Abschnitt 2.3.4. Jeder Eingang signalisiert die Überläufe für einen bestimmten Zähler. Mit »s626_dio_set_16_source()« läßt sich konfigurieren, welche Funktion der jeweilige Eingang wahrnehmen soll.

Das Auslesen der digitalen Eingänge erfolgt mit »s626_dio_in()«. Der Wert der digitalen Ausgänge kann mit »s626_dio_out()« gesetzt werden. Wichtig ist, daß Kanäle, die als Eingänge benutzt werden sollen, mit »s626_dio_out()« auf den Wert »0« gesetzt werden, da die ansonsten eventuell entstehenden hohen Ströme die Meßkarte zerstören können. Ein Wert von »0« entspricht einer Spannung von 5 V am Ausgang des jeweiligen Kanals. Eine Spannung von 0 V liegt am Ausgang an, wenn der Wert »1« gesetzt worden ist.

2.3.4 Zähler

Auf der Meßkarte sind sechs Zähler enthalten, die für verschiedene Aufgaben konfiguriert werden können. Die Zähler können z.B. über die Systemuhr oder aber auch über Inkrementalgeber gesteuert werden.

```
static void s626_counter_setup (int nr, unsigned short a,
                                unsigned short b);
static void s626_counter_preload (int nr, unsigned int value);
static unsigned int s626_counter_read (int nr);
static void s626_counter_read_irq (unsigned char *overflow,
                                   unsigned char *index);
static void s626_counter_reset_irq (unsigned char overflow,
                                    unsigned char index)
```

Mit »s626_counter_setup()« können jeweils zwei Zähler konfiguriert werden. Da sich zwei Zähler jeweils zwei Konfigurationsregister teilen, ist die Konfiguration recht unübersichtlich [Sen99]. Die obige Funktion wertet die Konfiguration selbst nicht aus, sondern schreibt einfach die beiden übergebenen Werte in die beiden Register.

Manchmal ist es sinnvoll, daß der Zähler nicht beim Wert »0« anfängt, zu zählen. Mit »s626_counter_preload()« kann für jeden Zähler ein Startwert gesetzt werden.

Der aktuelle Wert eines Zählers kann mit »s626_counter_read()« ausgelesen werden. Die Zähler können bei einem Überlauf bzw. bei einem anliegenden Index-Signal einen Interrupt erzeugen. Ob und wann ein Zähler einen Interrupt erzeugen soll, wird mit der schon erwähnten Konfigurationsfunktion »s626_counter_setup()« festgelegt. Ob und welcher Zähler einen Interrupt ausgelöst hat, kann mit »s626_counter_read_irq()« überprüft werden. Nach dem Aufruf dieser Funktion sollte der Interrupt-Status mit »s626_counter_reset_irq()« zurückgesetzt werden.

Wie bereits am Anfang dieses Abschnittes erwähnt wurde, können die Zähler so konfiguriert werden, das mit ihnen bis zu sechs extern angeschlossene Inkrementalgeber ausgewertet werden können. Die für den Anschluß der Inkrementalgeber notwendige Konfiguration erfolgt nicht im Treiber, da dieser allgemein gehalten werden sollte. Auch ist es wenig sinnvoll, den Treiber mit Funktionen zu überladen, die man genauso gut im User Mode implementieren könnte. Deswegen erfolgt die passende Konfiguration der Zähler durch das User Mode Programm, das diesen Treiber benutzt.

Mit jedem Impuls, der von dem jeweiligen Inkrementalgeber kommt, wird der Wert des Zähler in Abhängigkeit von der Drehrichtung entweder um Eins erhöht oder verringert.

2.3.5 Sensoray Model 626 Device

Wie bereits im Abschnitt 2.2.2 erläutert wurde, erfolgt die Kommunikation zwischen Treiber und den User Mode Anwendungen oftmals über sogenannte Devices. Der Treiber für die Meßkarte von Sensoray benutzt ein Device mit dem Namen »/dev/s626« und der Majornummer 241, um mit den Anwendungen zu kommunizieren.

Ein Treiber kann für jedes Device einige bestimmte Funktionen definieren und dem Kernel Zeiger auf diese Funktionen übergeben: »seek()«, »read()«, »write()«, »readdir()«, »select()«, »ioctl()«, »mmap()«, »open()«, »flush()« und »close()«. Öffnet z.B. eine Anwendung ein Device mit »open()« und schreibt dann Zeichen mit »write()« auf dieses Devices, so ruft der Kernel die »open()«- und »write()«-Funktionen des passenden Treibers für dieses Device auf. Es ist wichtig zu beachten, daß diese Funktionen zwar von User Mode Anwendung aufgerufen werden, aber im Kernel Mode laufen.

Das Device »/dev/s626« implementiert vier der obigen Funktionen, nämlich »open()«, »close()«, »read()« und »write()«:

```
static int device_open (struct inode *inode, struct file *file);
static int device_release (struct inode *inode, struct file *file);
static ssize_t device_read (struct file *file, char *buffer,
                            size_t length, loff_t *offset);
static ssize_t device_write (struct file *file, const char *buf,
```

```
                                size_t count, loff_t *ppos);
```

Sobald eine Anwendung mit »open()« das Device der Meßkarte öffnet, wird vom Kernel die Funktion »device_read()« aufgerufen. Diese sorgt dafür, daß das Device von keiner weiteren Anwendung geöffnet werden kann und das Modul der Meßkarte nicht aus dem Kernel entfernt werden kann. Als nächstes wird mittels der Funktion »s626_reset_card()« die Meßkarte in einen definierten Anfangszustand versetzt. Ruft eine Anwendung die Funktion »close()« auf, um das Device zu schließen, führt der Kernel die Funktion »device_release()« aus. Diese hebt die Blockierung des Devices für andere Anwendungen und die Blockierung des Entladens des Moduls wieder auf. Die Blockierung des Entladens ist wichtig, damit das Modul nicht entladen werden kann, während es von einer Anwendung benötigt wird.

Die Meßkarte besitzt sowohl Eingänge wie auch Ausgänge. Der Treiber benötigt deshalb eine Möglichkeit, Daten vom Treiber an eine Anwendung wie auch von einer Anwendung zum Treiber zu übermitteln. Dieses geschieht durch Lese- und Schreibzugriffe auf das Device des Treibers mit den User Mode Funktionen »read()« und »write()«. Werden diese benutzt, werden die Funktionen »device_read()« bzw. »device_write()« des Treibers aufgerufen.

Die Schreibfunktion muß dabei zwei Aufgaben erfüllen. Zum einen soll sich mit ihr der Pegel jedes digitalen Ausganges ändern lassen. Zum anderen soll über diese Funktion die Karte konfiguriert werden können. Da über ein Device einfach nur Bytes übertragen werden, mußte ein »Protokoll« definiert werden, welches festlegt, wie welche Aktion von einer Anwendung ausgelöst werden kann:

Digitaler Ausgang	D=<kanal>,<status>
Poll oder IRQ Modus	M=<modus>
Konfiguration	I=
	<struct s626_config_in>

Soll z.B. der fünfte digitale Ausgang eingeschaltet werden, muß die Anwendung auf das Device den Befehl »D=4,1« schreiben. Die Konfiguration kann von der Anwendung in einer Struktur vom Typ »s626_config_in« abgelegt und mit dem Befehl »I=« an den Treiber übermittelt werden:

```
struct s626_config_in
{
  unsigned short  cnt_setup[6];   /* counter: config registers */
  unsigned int    cnt_preload[6]; /* counter: preload values */

  unsigned short  dio_irq[3];           /* dio: enable IRQs? */
  unsigned short  dio_edge_type[3];     /* dio: type of the edges */
  unsigned short  dio_edge_capture[3];  /* dio: capture edges? */
  unsigned char   dio_16_source;        /* dio: channel 1-6? */
};
```

Mit dem dritten Befehl »M=« kann dem Treiber schließlich mitgeteilt werden, ob »read()«-Operationen im Poll oder im Interrupt Modus ausgeführt werden sollen. Dazu später noch mehr.

Ein Problem bei der Implementation der »device_write()«-Funktion stellte die Tatsache dar, daß dieser Funktion vom Kernel eine beliebige Anzahl von Bytes übergeben werden kann. Dieses führt dazu, daß nicht garantiert ist, daß der Funktion bei einem Aufruf ein kompletter Befehl übergeben wird. Es kann vielmehr sein, daß ein Befehl sich über mehrere Aufrufe erstreckt oder daß in einem Aufruf gleich mehrere Befehle enthalten sind. Als Lösung werden die eingehenden Bytes zuerst in einem Buffer gesammelt. Sobald die Funktion in dem Buffer einen kompletten Befehl erkannt hat, führt sie diesen aus und löscht den Buffer, so daß dieser Sammel- und Auswerteprozeß von neuem beginnen kann.

Bei dem Austausch von Daten zwischen dem Treiber und der Anwendung muß bedacht werden, daß sich die Daten entweder im Kernel oder im User Mode befinden. Sie müssen deshalb mit »get_user()« bzw. »put_user()« vom User Mode in den Kernel Mode bzw. umgekehrt konvertiert werden.

Die »device_read()«-Funktion liefert einfach eine Struktur an die Anwendung zurück. Diese Struktur enthält den Status aller digitalen Eingänge und die Werte aller Zähler. Außerdem ist in ihr vermerkt, welcher Eingang bzw. Zähler einen Interrupt ausgelöst hat:

```
struct s626_data_out
{
  unsigned short  din_a, din_b, din_c;  /* dio */
  unsigned int    counter[6];           /* counter */

  unsigned short  irq_din_a, irq_din_b,
                  irq_din_c;            /* dio: IRQs */
  unsigned char   irq_counter_overflow; /* counter: overflow IRQ? */
  unsigned char   irq_counter_index;    /* counter: index IRQ? */
};
```

Bei dieser Funktion ist genau wie bei der Funktion »device_write()« nicht garantiert, daß der Kernel immer die komplette Struktur bei der Funktion abholt. Deswegen wird auch hier ein Buffer benutzt. In diesem wird die Struktur abgespeichert. Sobald eine komplette Struktur abgeholt worden ist, wird eine neue im Buffer mit den aktuellen Werten abgelegt. Die aktuellen Werte werden mit den in den Abschnitten 2.3.3 und 2.3.4 beschriebenen Lowlevel-Funktionen ermittelt.

Eine Besonderheit dieser Funktion ist noch die Unterstützung von zwei Modi: dem Pollmodus und dem IRQ-Modus. Im Pollmodus liefert die »read()«-Funktion einfach die aktuellen Werte der Meßkarte als Struktur an die Anwendung zurück. Im IRQ-Modus wartet der Treiber hingegen, bis die Meßkarte einen Interrupt auslöst. Erst danach werden die aktuellen Werte der Meßkarte an die Anwendung übermittelt.

Der IRQ-Modus ist immer dann sinnvoll, wenn die Anwendung nur dann an Werten interessiert ist, wenn sich diese geändert haben. Im Pollmodus müßte die Anwendung in einem solchen Fall immer die Werte auslesen und selbst überprüfen, ob diese sich geändert haben. Das ist nicht nur umständlich, sondern verbraucht auch für ein Multitasking System wertvolle Rechenzeit.

Im IRQ-Modus versetzt der Treiber die Anwendung durch den Aufruf der Funktion »interruptible_sleep_on()« solange in einen Schlafzustand, bis die Karte einen Interrupt erzeugt hat [BBD+99]. In diesem Zustand verbraucht die Anwendung keinerlei Rechenzeit, so daß diese von anderen Anwendungen benutzt werden kann. Erreichen tut der Kernel dieses dadurch, daß er die betroffene Anwendung aus der Warteschlange des Schedulers entfernt und sie in eine spezielle Warteschlange für Programme ablegt, die auf ein Device warten [SG97].

Sobald ein Interrupt auftritt, wird die im Treiber implementierte Callback Funktion »device_do_irq()« vom Kernel aufgerufen. Hierbei handelt es sich um einen sogenannten IRQ-Handler. Die Funktion überprüft mit den Lowlevel-Funktionen, die in den Abschnitten 2.3.3 und 2.3.4 beschrieben wurden, ob die Meßkarte einen Interrupt ausgelöst hat. Ist dieses der Fall, wird die schlafende Anwendung mit dem Befehl »wake_up()« wieder aufgeweckt. Eine Überprüfung, ob der Interrupt von der Meßkarte ausgelöst wurde, ist deshalb notwendig, da sich eventuell mehrere PCI-Karten einen Interrupt teilen.

2.3.6 Watchdog Device

Der Linux Kernel verfügt über eine Standardschnittstelle für Watchdogs [Cox00]. Die Schnittstelle verwendet das Device »/dev/watchdog« mit der Majornummer 10 und der Minornummer 130.

Sobald eine Anwendung dieses Device öffnet, wird die Watchdog aktiviert und bleibt aktiv, bis das Device wieder geschlossen wird. Damit die Watchdog den Rechner nicht neu bootet, muß die Anwendung, die das Device geöffnet hat, regelmäßig mindestens ein Zeichen auf dieses schreiben. Hierdurch wird der Timer der Watchdog zurückgesetzt. Nach welcher Zeit spätestens auf das Device geschrieben werden muß, um einen Reset des Rechners zu verhinden, ist von der Watchdog-Hardware abhängig.

Die Idee hinter diesem Konzept ist, daß das Programm, das für das Schreiben auf das Watchdog Device zuständig ist, wahrscheinlich auch hängen bleibt, wenn der Rechner hängt. Bei Problemen wird also das Device nicht regelmäßig beschrieben und die Karte bootet den Rechner neu. Ein solches Programm zur Ansteuerung des Watchdog Devices sieht z.B. so aus:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main (void)
{
  int fd = open ("/dev/watchdog", O_WRONLY);
  if (fd == -1)
  {
    perror("watchdog");
    return 1;
  }
  while (1)
  {
    write (fd, "\0", 1);
    sleep (5);
  }
}
```

Für die Meßkarte, die ja eine Watchdog enthält, wurde deshalb ein Watchdog Device implementiert, das aus den drei Funktionen für »open()«, »close()« und »write()« besteht:

```
static int s626_wtd_open (struct inode *inode, struct file *file);
static ssize_t s626_wtd_write (struct file *file, const char *buf,
                               size_t count, loff_t *ppos);
static int s626_wtd_release (struct inode *inode, struct file *file);
```

Die »open()«-Funktion hat die Aufgabe, das Öffnen des Watchdog Devices durch mehrere Programme und das Entladen des Moduls zu verhindern. Außerdem schaltet sie mit der Lowlevel-Funktion »s626_wtd_enable()« die Watchdog der Meßkarte ein; siehe Abschnitt 2.3.2. Durch die Funktion »close()« werden die Einstellungen, die »open()« gesetzt hat, wieder rückgängig gemacht.

Fordert eine Anwendung den Treiber durch das Schreiben eines Zeichens auf, die Watchdog zurückzusetzen, wird die »write()«-Funktion des Watchdog Devices aufgerufen. Die Funktion überprüft einfach, ob mindestens ein Zeichen durch die Anwendung geschrieben wurde. Ist dieses der Fall, wird der Timer durch die Lowlevel-Funktion »s626_wtd_clear()« zurückgesetzt.

2.3.7 Modul Management

Der Treiber wurde, wie bereits weiter am Anfang erwähnt wurde, als Modul implementiert. Jedes Linux Modul muß zwei Standardfunktionen besitzen, die vom Linux Kernel als Einsprungstelle benutzt werden:

```
int init_module();
void cleanup_module();
```

Die erste Funktion wird vom Kernel ausgeführt, sobald das Modul mit dem Programm »insmod« zur

Laufzeit geladen wird. Wird das Modul zur Laufzeit mit »rmmod« wieder entladen, so wird die zweite Funktion ausgeführt.

Beim Treiber für die Meßkarte hat die Initialisierungsfunktion mehrere Aufgaben. Als erstes versucht sie, eine PCI-Karte zu finden, die die PCI Device ID und Vendor ID der Meßkarte hat. Wird eine solche Karte gefunden, wird deren Interrupt und die Basisadresse des Fensters, das diese in den Speicherbereich des PCs einblendet, ermittelt und in Variablen abgelegt.

Die vom Kernel ermittelte Basisadresse ist eine physikalische Adresse. Da der PC eine MMU (Memory Management Unit) besitzt und nicht mit physikalischen Adressen arbeitet, muß diese vor Benutzung noch mit der Funktion »ioremap()« in eine virtuelle Adresse umgewandelt werden [Mac99]. Auf diese virtuelle Adresse kann allerdings aus dem Kernel auch nicht direkt zugegriffen werden, sondern hierfür müssen die »read*«- und »write*«-Makros benutzt werden.

Nachdem Interrupt und Basisadresse ermittelt sind, wird der DEBI-Bus der Meßkarte mit der Lowlevel-Funktion »DEBI_init()« initialisiert; siehe Abschnitt 2.3.1. Als nächster Schritt werden die beiden Devices des Treibers registriert. Das Watchdog Device nimmt hierbei eine Sonderstellung ein. Im Gegensatz zu dem anderen Device kann es nicht mit der Standardfunktion »module_register_chrdev()« registriert werden, da der Kernel selbst bereits Treiber für einige Watchdogs enthält. Es gibt deshalb eine zusätzliche Schicht, bei der ein Device für eine Watchdog registriert werden muß. Diese Registrierung erfolgt mit der Funktion »misc_register()«. Für die Watchdog wird außerdem eine Callback Funktion registriert, die im Fall eines vom Administrator veranlaßtem Herunterfahren des Rechners aufgerufen wird. In dieser Funktion wird die Watchdog der Karte abgeschaltet, um so ein Ablaufen des Timers der Watchdog während des Herunterfahrens zu verhindern.

Als letztes wird noch der Interrupt-Handler registriert, der von der »device_read()«-Funktion benutzt wird; siehe Abschnitt 2.3.5. Bei der Registrierung wird dem Kernel auch gleich mitgeteilt, daß der Treiber das Sharen von Interrupts unterstützt.

Die »cleanup_module()« Funktion teilt dem Kernel mit, das die in »init_module()« registrieren Callback Funktionen und Devices entfernt werden sollen.

2.4 Implementation der Bibliothek

Da eine manuelle Ansteuerung der beiden Devices aus Programmen heraus recht umständlich wäre, wurde eine Laufzeitbibliothek entwickelt, die alle Möglichkeiten des Treibers in C-Funktionen kapselt. Dieses erleichtert nicht nur die Nutzung, sondern sorgt auch für eine Typsicherheit.

Alle Funktionen dieser Bibliothek erwarten ein Handle von folgendem Typ:

```
struct s626_hd
{
  struct s626_config_in config;     /* stores the configuration
                                       of the card */
  struct s626_data_out  data;       /* data read from the card */
  int                   device_hd;  /* device: file handle */
};
```

In diesem ist zum einen das Handle der Device Datei abgelegt und zum anderen sind hier die Konfiguration und die Daten, die von der Karte gelesen wurden, zu finden. Erzeugt und zerstört wird ein solches Handle mit den beiden folgenden Funktionen:

```
int s626_device_open (char *name, struct s626_hd *hd);
void s626_device_close (struct s626_hd *hd);
```

Auf diesen Handle setzen vier Grundfunktionen auf:

```
int s626_device_write_config (struct s626_hd *hd);
int s626_device_write_string (struct s626_hd *hd, char *str);
int s626_device_read_data (struct s626_hd *hd);
int s626_device_read_modus (struct s626_hd *hd, enum read_mode mode);
```

Mit der ersten Funktion wird die Konfiguration der Karte, die im Handle abgelegt ist, an den Treiber übermittelt. Dieser konfiguriert die Karte dann entsprechend. Die zweite Funktion erlaubt es, einen beliebigen Befehl an den Treiber der Karte zu schicken. Die im Handle enthaltene Variable »data« wird mit der dritten Funktionen gefüllt, die hierfür eine »read()«-Operation auf dem Device der Meßkarte ausführt. Ob hierbei gepollt werden soll oder ob der IRQ-Modus Verwendung finden soll, wird mit der letzten Funktion festgelegt. Auf diesen Funktionen bauen weitere spezielle Funktionen für die digitalen Ein- und Ausgänge und für die Zähler auf. Folgende Funktionen sind für die Zähler implementiert worden:

```
void s626_cnt_setup (struct s626_hd *hd, int cnt_nr,
                     enum s626_cnt_idx_source idx_source,
                     enum s626_cnt_source source,
                     enum s626_cnt_idx_egde idx_egde,
                     enum s626_cnt_preload_trigger preload_trigger,
                     enum s626_cnt_multiplier multiplier,
                     enum s626_cnt_irq_source irq_source,
                     enum s626_cnt_source_egde source_egde,
                     enum s626_cnt_enable enable,
                     enum s626_cnt_clear clear);
void s626_cnt_preload (struct s626_hd *hd, int cnt_nr,
                       unsigned int preload);
unsigned int s626_cnt_read (struct s626_hd *hd, int cnt_nr);
int s626_cnt_overflow (struct s626_hd *hd, int cnt_nr);
int s626_cnt_index (struct s626_hd *hd, int cnt_nr);
```

Die Aufgaben der einzelnen Funktionen sind leicht aus deren Namen zu erkennen. Die erste Funktion erlaubt die Konfiguration eines jeden Zählers. Mit »s626_cnt_preload()« kann jedem Zähler ein Wert zugewiesen werden, mit dem er anfangen soll, zu zählen. Soll dann irgendwann der Stand eines Zählers ermittelt werden, kann dafür »s626_cnt_read()« benutzt werden. Ob ein Zähler aufgrund eines Überlaufes oder eines Index-Signals einen Interrupt ausgelöst hat, läßt sich mit den letzten beiden Funktionen ermitteln.

Alle obigen Funktionen arbeiten wie auch die meisten Funktionen für die digitalen Ein- und Ausgänge nur mit den in dem Handle abgelegten Werten. Diese Werte müssen dann mit den Funktionen »s626_device_write_config()« bzw. »s626_device_read_data()« an den Treiber übermittelt bzw. von diesem gelesen werden. Dieses Konzept ist sinnvoll, damit nicht für jede kleinste Anfrage immer wieder mit dem Treiber kommuniziert werden muß. Außerdem ist es so möglich, beim Lesen einen bestimmten Status festzuhalten und diesen auszuwerten. Ansonsten wäre es z.B. möglich, daß während der Auswertung eines Zählers sich ein anderer bereits geändert hat.

Die Funktionen für die digitalen Ein- und Ausgänge sehen entsprechend aus:

```
void s626_dio_setup (struct s626_hd *hd, int nr, enum s626_bool irq,
                     enum s626_dio_edge_type edge_type,
                     enum s626_bool edge_capture);
void s626_dio_s16_setup (struct s626_hd *hd, int nr,
                         enum s626_dio_s16_type type);
```

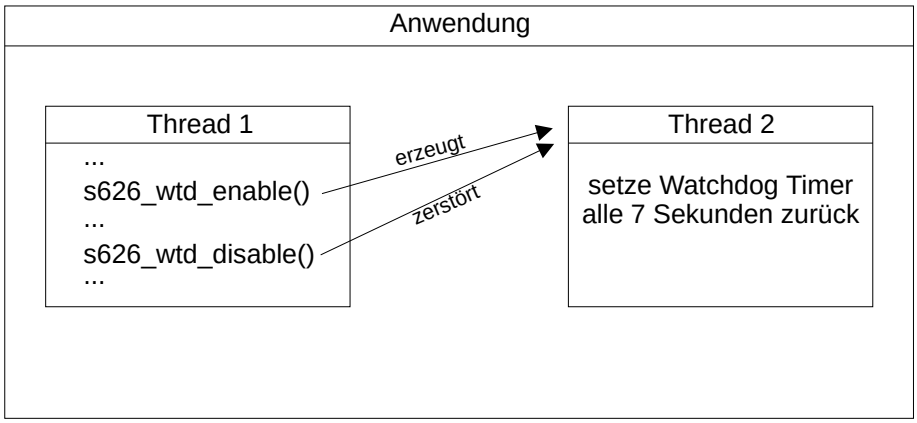


Abbildung 2.3: Watchdog Funktionen der Bibliothek

```
void s626_dio_out (struct s626_hd *hd, int nr,
                   enum s626_dio_onoff onoff);
enum s626_dio_onoff s626_dio_in (struct s626_hd *hd, int nr);
int s626_dio_read_irq (struct s626_hd *hd, int nr);
```

Auch hier gibt es zwei Funktionen zur Initialisierung, mit denen z.B. festgelegt werden kann, ob Flanken erkannt werden sollen. Es gibt dann zwei weitere Funktionen, mit denen digitale Eingänge gelesen und digitale Ausgänge gesetzt werden können. Die letzte Funktion erlaubt schließlich die Überprüfung, ob ein Kanal einen Interrupt ausgelöst hat.

Neben den Funktionen zur Ansteuerung des Devices, über das die Meßdaten und die Konfiguration übertragen werden, unterstützt die entwickelte Bibliothek durch zwei weitere Funktionen außerdem das Watchdog Device von Linux:

```
void s626_wtd_enable (int *hd);
void s626_wtd_disable (int hd);
```

In Abbildung 2.3 ist die Funktionsweise dieser beiden Funktionen zu erkennen. Wird in einer Anwendung die Funktion »s626_wtd_enable()« aufgerufen, erzeugt die Bibliothek einen zweiten Thread. Dieser läuft dann parallel zu dem eigentlichen Hauptprogramm der Anwendung. Der zweite Thread schreibt alle sieben Sekunden ein Zeichen auf das Watchdog Device und verhindert so, daß die Meßkarte den Rechner neu bootet. Mit »s626_wtd_disable()« kann der zweite Thread beendet und damit die Watchdog abgeschaltet werden.

Dieses Konzept erlaubt es einer Anwendung sehr leicht, die Watchdog zu benutzen. Durch den zweiten Thread, der den Timer automatisch zurücksetzt, muß sich die Anwendung selbst garnicht darum kümmern, wann der Timer als nächstes zurückgesetzt werden müßte. Eine solche Überprüfung würde bei großen Anwendungen zu viel unnötigem Code führen und diesen damit unübersichtlich machen.

2.5 Ergebnisse

2.5.1 Probleme

Bei der Entwicklung des Treibers gab es drei Hauptprobleme. Das erste Problem war die Dokumentation der Meßkarte selbst. Diese beschrieb leider, wie bereits in Abschnitt 2.3.1 erwähnt wurde, die

Karte nur sehr lückenhaft. Einige Informationen konnten zwar aus dem Source Code des vorliegenden Treibers für das Betriebssystem QNX extrahiert werden, dieses war jedoch ein sehr mühsames und fehleranfälliges Verfahren.

Eine weitere Quelle für Probleme war die Dokumentation des Linux Kernels. Der Kernel selbst wird mit eigentlich keinerlei Dokumentation ausgeliefert, die diesen Namen verdient hätte. In solchen Fällen wird oft empfohlen, auf den Source Code selbst zurückzugreifen. Dieses gestaltete sich jedoch sehr schwierig, da viele Bereiche des Kernels im Source Code fast keine Kommentare enthalten. Da der Kernel mit inzwischen fast 90 MByte Source Code sehr komplex ist, fällt es ohne Dokumentation und Kommentare sehr schwer, die Zusammenhänge nur aus dem Source Code selbst zu erkennen. Die verwendete Literatur wie [BBD+99] gibt zwar einen Überblick über einige Bereiche des Kernels, beschreibt allerdings seltsamerweise trotz des Titels »Linux Kernelprogrammierung« nicht die Entwicklung von Treibern, wobei ja gerade Treiber den größten Teil des Kernels ausmachen.

Schwierig ist auch das Debuggen von Treibern. Da diese ja im Kernel Modus laufen, gibt es keine Möglichkeit, normale Debugger wie z.B. »gdb« zu verwenden, die im User Mode laufen. Die einzige Möglichkeit, dem Programmierer zu helfen, besteht in der Verwendung der Funktion »printk()«. Mit ihr können beliebige Texte und Variablen in den Logdateien von Linux abgelegt werden. Der Programmierer muß also manuell an interessanten Stellen im Source Code diese Funktion einfügen und z.B. Werte von interessanten Variablen in die Logdatei ausgeben.

Diese Methode hilft zwar teilweise, bringt aber z.B. recht wenig, wenn der Treiber einfach den kompletten Rechner zum Absturz bringt. Denn im Gegensatz zum Debugger weiß man nach dem Absturz nicht, wo dieser genau aufgetreten ist. Um dieses herauszufinden, müßte man im Prinzip jede Funktion mit »printk()«-Zeilen versehen, was wiederum dazu führen kann, daß die Logdatei innerhalb kürzester Zeit die 100 MByte Grenze überschreitet.

Der entwickelte Treiber für die Meßkarte lief am Ende der Diplomarbeit mehr oder weniger stabil. Sowohl die Zähler wie auch die digitalen Ein- und Ausgänge funktionieren im Pollmodus problemlos. Der IRQ-Modus war allerdings trotz intensiver Fehlersuche nicht zum Laufen zu bringen. Sobald der IRQ-Handler registriert wurde, stürzte der komplette Rechner ab. Ob dieses an der Initialisierung des SAA7146 ICs auf der Meßkarte oder an einer anderen Stelle des Treibers lag, ließ sich aufgrund fehlender und lückenhafter Literatur nicht ermitteln.

2.5.2 Verzicht auf die Meßkarte

Da die Firma Feige und ich aufgrund der Probleme bei der Treiberentwicklung und vor allem aufgrund der Echtzeit-Problematik nicht überzeugt waren, daß sich die Achsenpositionierung auf dem PC überhaupt problemlos realisieren lassen würde, wie dieses gemäß Abbild 1.2 ursprünglich geplant war, wurde beschlossen, die Achsenpositionierung in die SPS auszulagern. Der PC sollte dann einfach, wie in Abbildung 2.4 zu sehen ist, die Koordinaten der Spundlöcher an die SPS übermitteln, die dann völlig selbständig den Befüllvorgang steuern kann.

Aufgrund der sehr lückenhaften Dokumentation zum Thema RT-Linux ist dieses sicherlich eine sinnvolle Entscheidung gewesen.

In der geänderten Systemübersicht ist überhaupt keine Meßkarte mehr vorgesehen. Die Kommunikation zwischen PC und SPS erfolgt nicht mehr über digitale Ein- und Ausgänge sondern über eine genormte RS-232-Schnittstelle. Auf dem PC läuft ein Kommunikationsdaemon, der von der Firma Feige entwickelt wurde. Dieser kommuniziert mit der SPS über die serielle Schnittstelle und ruft die Routinen der Bilderfassung und Bildverarbeitung auf, die in einer Laufzeitbibliothek vorliegen; dazu später in den Kapiteln 3 und 4 noch mehr.

Eine Leitung des Inkremtalgebers, der am Tragkettenförderer montiert ist, ist jetzt mit dem Eingang für externe Synchronisation des Framegrabbers verbunden. Bei jedem Impuls des Inkremtalgebers wird eine Zeile aufgenommen. Diese Methode hat den Vorteil, daß die Synchronisation jetzt vollständig in Hardware verläuft und somit vom PC unabhäng ist. Es soll jedoch nicht verschwiegen werden, daß die Lösung zwei Nachteile hat. Zum einen wird jetzt auch eine Zeile vom Framegrabber aufgenommen,

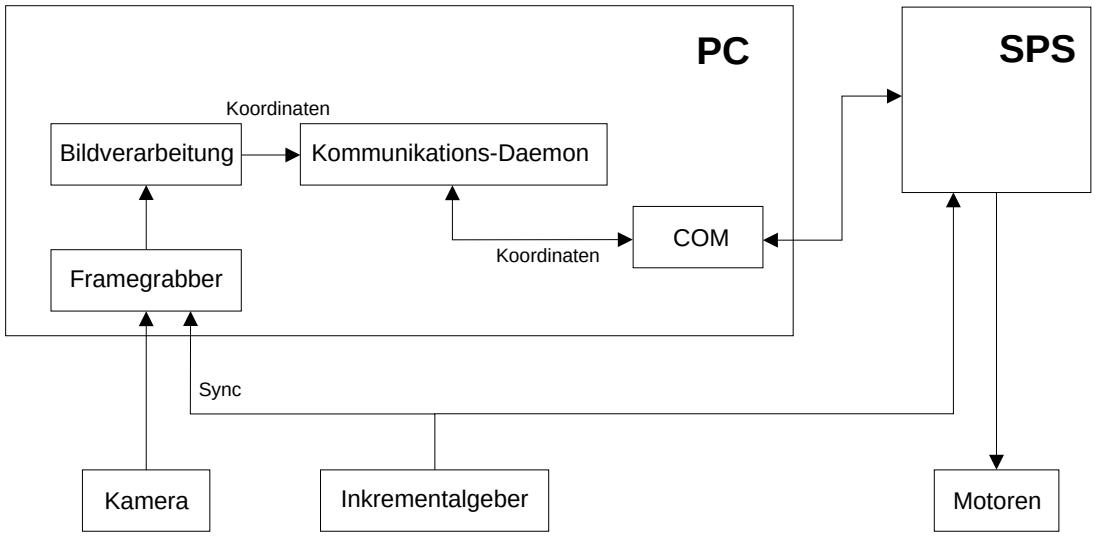


Abbildung 2.4: Geänderte Systemübersicht

wenn der Tragkettenförderer rückwarts läuft, da nur das A-Signal nicht aber das B-Signal ausgewertet wird und somit die Drehrichtung unbekannt ist. Ein weiteres Problem entsteht dann, wenn der Tragkettenförderer zu schnell läuft, so daß die Kamera nicht mitkommt und man so ein verzerrtes Bild erhält.

Kapitel 3

Bildverarbeitungsbibliothek

3.1 Motivation

Ursprünglich war geplant, für die Erkennung der Spundlöcher die in der Diplomarbeit von Klaus Wiehler [Wie96] entwickelten C++-Sourcen zu verwenden, da dort bereits ein ähnliches Problem gelöst worden war. Die Wiederverwendung der bereits implementierten Algorithmen erwies sich jedoch aus mehreren Gründen als schwierig.

So stammte ein Teil der implementierten Algorithmen aus einer älteren Bildverarbeitungsbibliothek, die zuerst von Fortran nach C und dann von C nach C++ konvertiert worden war. Ein solches Vorgehen führt selten zu guten Implementierungen und ist oft sehr anfällig für das Hinzufügen von Fehlern. Verschlimmert wurde die Situation dadurch, daß der Code und vor allem die Parameter der Funktionen so gut wie garnicht dokumentiert waren. Außerdem nutzte die Implementation aufgrund ihres Alters nicht die aktuellen C++-Schnittstellen und -Bibliotheksfunktionen wie z.B. die »Standard Template Library« (STL), was das Zusammenspiel mit Funktionen anderer Entwickler doch erheblich erschwert hätte.

Um Zeit zu sparen und am Ende ein stabiles System zu erhalten, entstand deshalb zuerst die Idee, die für das Projekt notwendigen Algorithmen in C++ einfach neu zu implementieren. Dieses erschien sinnvoll, da man für das Verstehen und Debuggen von fremden, alten Implementationen oftmals deutlich mehr Zeit benötigt wie für eine neue Implementierung.

Ähnlich war bisher eigentlich auch bei den meisten anderen Projekten des Arbeitsbereichs verfahren worden. Dieses führte dazu, daß jedes Mitglied des Arbeitsbereichs, ob nun wissenschaftlicher Mitarbeit oder Student, immer wieder die gleichen Algorithmen neu implementieren mußte, wobei unnötig viel Zeit verloren ging. Aus diesem Grund bestand bei Prof. Grigat der Wunsch, die Wiederverwendbarkeit der Implementationen der Algorithmen zu erhöhen.

Oftmals ist ja zu hören, daß allein durch die Verwendung einer objektorientierten Sprache wie C++ die Wiederverwendbarkeit erhöht werden könne. Das ist jedoch nicht richtig. Unter bestimmten Umständen erschweren solche Sprachen sogar die Wiederverwendbarkeit, wie man am nachfolgenden Beispiel sehen kann:

```
void filter1 (MyImageType *img);
...
void filter2 (char *img);
...
void filter3 (Image *img);
```

Alle drei Funktionen stammen von unterschiedlichen Autoren und sollen jetzt z.B. nacheinander auf ein

und dasselbe Bild angewendet werden. Dieses erweist sich jedoch als schwieriger als zunächst angenommen, da jeder Autor ein anderes Konzept hat, wie er die Bilddaten ablegt. Man müßte die Bilddaten also, falls dieses überhaupt möglich ist, zwischen den Aufrufen der Filter passend konvertieren. Dieses kostet natürlich auch viel Rechenleistung.

Um eine Wiederverwendung zur ermöglichen, ist es also notwendig, sich auf gemeinsame Datenstrukturen für den Austausch der Daten zwischen den einzelnen Funktionen und Klassen zu einigen. Zuerst wurde überlegt, hierfür existierende Bibliotheken wie »Target Junior« oder »Vision SDK« zu verwenden; diese Idee wurde jedoch verworfen.

»Target Junior« schien wenig geeignet, da der Source sehr chaotisch strukturiert und teilweise schon ziemlich alt ist. Viele Funktionen sind auch nicht richtig dokumentiert. Auch benutzt die Bibliothek an vielen Stellen eigene Typen statt der Typen aus der modernen STL, was das Zusammenfügen mit anderen Sourcen schwierig macht. Schließlich ließ sich die Bibliothek auch unter keinem Betriebssystem fehlerfrei übersetzen.

Die »Vision SDK« Bibliothek war ebenfalls nicht geeignet, da sie nur unter MS-Windows läuft und mit einer ungeeigneten Lizenz vertrieben wird. Diese Bibliothek verfügt außerdem über fast keine Bildverarbeitungsalgorithmen.

Deshalb wurde nach einiger Diskussion im Arbeitsbereich beschlossen, eine eigene Bibliothek für die Bildverarbeitung in C++ zu entwickeln. Die Aufgabe des Designs und der Implementierung dieser Bibliothek wurden hauptsächlich von dem Studenten Eric Sommerlade und mir übernommen.

3.2 Designziele

Bei dem Design der neuen Bildverarbeitungsbibliothek wurden einige Ziele verfolgt, die im weiteren kurz erläutert werden sollen.

Das oberste Ziel war die schon vorher angesprochene Wiederverwendbarkeit innerhalb des Arbeitsbereichs. Es sollte nicht mehr notwendig sein, für jedes Projekt immer wieder die gleichen Algorithmen neu zu implementieren. Um dieses zu ermöglichen, war es also als erstes notwendig, Klassen zu definieren, die Bilder im Speicher repräsentieren. Außerdem bedeutet dieses, daß Algorithmen nicht mehr speziell für die Lösung des eigenen Problems sondern möglichst allgemein implementiert werden mußten.

Ebenfalls in Richtung Wiederverwendbarkeit zielt die Bevorzugung der STL gegenüber eigenen Klassen, wann immer dieses möglich ist. Es macht z.B. keinen Sinn, eine eigene Klasse für eine Liste zu implementieren, wenn man statt dessen auch die bei jedem ANSI konformen C++ Compiler vorhandenen Klassen wie z.B. »vector<>« verwenden kann. Die STL-Klassen bieten vor allem den Vorteil, daß die meisten Programmierer ihr Interface bereits kennen. Außerdem ist davon auszugehen, daß diese Implementationen bereits auf Fehler untersucht worden sind.

Da sich im Fachbereich verschiedene Betriebssysteme und Architekturen im Einsatz befinden, muß eine Bibliothek, die von möglichst vielen Mitglieder benutzt werden soll, natürlich portabel gestaltet werden. Es ist außerdem sowieso immer eine gute Idee, ein Programm so portabel wie möglich zu gestalten und portable und proprietäre APIs zu trennen. Bis auf einige wenige I/O-Routinen läßt sich sowas auch bei einer Bildverarbeitungsbibliothek problemlos realisieren.

Als problematisch hat sich bei vielen Projekten von Universitäten das Thema Lizenzen herausgestellt. Da jedem Studenten die Rechte an den von ihm entwickelten Source gehören, müßte er vor der Verwendung seiner Sourcen in anderen Projekten um Erlaubnis gefragt werden, was sich oftmals als schwierig gestaltet und damit die Wiederverwendbarkeit erschwert. Aus diesem Grunde wurde von den beiden Hauptentwicklern der Bibliothek vorgeschlagen, diese unter die GNU »Library General Public License« zu stellen, so daß die Bibliothek nicht nur von Mitgliedern des Fachbereiches sondern von jedem interessierten Entwickler genutzt werden kann, solange gewisse Rechte gewahrt bleiben. Diese Lizenz ermöglicht auch, gewisse Teile eines Projektes gemäß NDA geheim zu halten.

3.3 C++ Theorie

Da die Bibliothek intensiv spezielle Konzepte von ANSI C++ verwendet, mit denen viele ANSI C Benutzern noch nicht vertraut sein dürften, werden diese im nachfolgenden kurz vorgestellt. Viele Konzepte sind zwar in [Str98, Str00, Mey95, Mey97, Lak96, Jos99] beschrieben, allerdings sind diese Beschreibungen gerade für Neulinge sehr schwer verständlich.

3.3.1 const

Das Schlüsselwort »const« gibt es zwar schon in ANSI C, allerdings hat es in C++ einige weitere Aufgaben. Grundsätzlich kann man Variablen und Funktionen von Klassen als konstant definieren.

Die folgende Zeile definiert z.B. einen variablen Zeiger auf konstante Daten:

```
const char *pnt;
...
pnt = pnt2;    // ok
pnt[0] = 20;   // Fehler
```

Man kann also den Zeiger jederzeit an eine andere Stelle zeigen lassen, aber man kann die Daten, auf die dieser zeigt, nicht über den Zeiger ändern.

Neu ist die Verwendung im Zusammenhang mit Klassen. Wenn man ein konstantes Objekt einer Klasse hat, kann man nur die Funktionen des Objektes ausführen, die als konstant definiert sind.

```
class CFoo
{
  ...
  void foo1() const;
  void foo2();
  ...
};

void testfunc (const CFoo &test)
{
  test.foo1();  // ok
  test.foo2();  // Fehler
}
```

Da in der Funktion »testfunc« das Objekt »test« vom Typ »CFoo« konstant ist, kann die Funktion »CFoo::foo2()« nicht ausgeführt werden. Solche Probleme werden bereits während der Compilierung und nicht erst zur Laufzeit bemerkt.

Funktionen sollten nur dann als konstant definiert werden, wenn sie die Daten, die im Objekt abgelegt sind, nicht verändern. Auch sollten konstante Funktionen niemals Zeiger auf Daten, die innerhalb des Objektes gespeichert sind, zurückliefern, da diese ansonsten über diese Zeiger geändert werden können [Mey95, Lak96].

3.3.2 Vererbung

C++ unterstützt die OOP-Konzepte der einfachen und mehrfachen Vererbung. Die Vererbung bietet im Prinzip zwei Vorteile.

Dank der Vererbung können sich mehrere Klassen die Implementation einer Funktion teilen. So bieten z.B. alle Klassen der Bildverarbeitungsbibliothek, die ein Bild im Speicher definieren, die Funktion »getW()«, die die Breite des Bildes zurückliefert. Statt diese Funktion z.B. für jedes Farbmodell neu zu implementieren, erben alle Klassen diese Funktionen von einer Wurzelklasse:

```
class CImage
{
  ...
  int getW() const;
  ...
};

class CImageByte : public CImage
{
};

class CImageInt : public CImage
{
};
```

Die beiden abgeleiteten Klassen »CImageByte« und »CImageInt« kennen jetzt beide die Funktion »get W«, die in »CImage« implementiert ist.

Außerdem kann man Vererbung benutzen, um ein Interface festzulegen. So existiert z.B. ein Interface für Filter. Alle Filter haben damit die gleichen Grundfunktionen als Interface.

Interessant ist in diesem Zusammenhang auch das Schlüsselwort »virtual«. Die Implementation einer virtuellen Funktion einer Klasse kann in einer abgeleiteten Klasse verändert werden. Für Interfaces will man in der Regel in der Basisklasse überhaupt keine Implementierung der spezifizierten Funktionen haben. Deshalb werden solche Funktionen »pure virtual« definiert:

```
class CFilter
{
  ...
  virtual void filter (CImageIntGray &img) = 0;
  ...
};
```

Eine Klasse, die rein virtuelle Funktionen enthält, kann nicht als Objekt erzeugt und genutzt werden. Erst eine von dieser Klasse abgeleitete Klasse, die diese virtuellen Funktionen dann implementiert, kann als Objekt erzeugt werden.

3.3.3 Exceptions

Exceptions werden für die Fehlerbehandlung benutzt. In älteren Sprachen wie ANSI C melden Funktionen Fehler in der Regel über den Rückgabewert. Dieses hat jedoch den Nachteil, daß man diesen Rückgabewert nach jedem Funktionsaufruf überprüfen muß, was zu sehr viel Code führt. Aus diesem Grund vernachlässigen viele Programmierer gerne die Fehlerbehandlung; dieses gilt besonders für die Test- und Entwicklungsphase.

Exceptions gehen hier einen anderen Weg und sind eher mit Signalen zu vergleichen. Ein Vorteil von Exceptions ist, daß man sie abfangen muß. Tut man dieses nicht, beendet sich ein Programm im Fehlerfall automatisch. Der Programmierer wird also von Anfang an gezwungen, sorgfältig zu arbeiten.

Mit dem Befehl »throw()« wirft man eine Exception. Die betreffende Funktion wird dann verlassen und der Compiler sucht nach einem »try«-Block:

```
void f1 (void)
{
  ...
  throw CException ();
  ...
}

void f2 (void)
{
  ...
  f1();
  ...
}

int main (void)
{
  ...
  try
  {
    f2();
  }
  catch(...)
  {
    return 1;
  }
  ...
}
```

Tritt in diesem Beispiel irgendwo in der Funtionen »f1()« oder »f2()« ein Fehler auf, springt das Programm zurück in die »main()«-Funktion und führt die Anweisungen im entsprechenden »catch«-Block aus.

Dieses Konzept hat allerdings nicht nur Vorteile. Teilweise ist es sehr schwer, immer zu bedenken, welche Unterfunktionen eine Funktion aufruft und welche Exceptions folglich auftreten können. Es ist daher sehr wichtig, wenn man Exceptions benutzt, diese gut zu strukturieren und vor allem zu dokumentieren.

3.3.4 RTTI

Die Runtime Type Informations erlauben es, zur Laufzeit festzustellen, von welchem Typ ein Objekt im Speicher ist. Das erlaubt es C++ z.B., einen speziellen Cast-Operator mit dem Namen »dynamic_cast« anzubieten, der nicht einfach wie ein Cast in C den Typ eines Zeigers ändert, sondern vielmehr vorher überprüft, ob der Zeiger überhaupt von dem neuen Typ sein kann. Wenn das nicht der Fall ist, wird eine Exception ausgelöst.

Auch gibt es eine Funktion, mit der man einfach den Typ eines Objektes ermitteln kann. Dieses ist vor allem in Verbindung mit APIs von Vorteil. Man kann hier als Übergabeparameter eine Basisklasse verwenden und später dann noch herausfinden, von welchem von dieser Basisklasse abgeleiteten Klasse dieser Objekt stammt.

3.3.5 Namespaces

Bei ANSI C und vielen anderen Sprachen hat man oft Probleme, wenn man Programmteile von verschiedenen Autoren zusammenfügt, da dann eventuell manche globalen Variablen- und Funktionsnamen mehrfach für verschiedene Aufgaben verwendet werden.

Dieses Problem wurde bei ANSI C++ durch die Einführung der Namespaces sehr geschickt gelöst. Es kann jetzt ruhig mehrfach der gleiche Name verwendet werden, solange dieses in verschiedenen Namespaces passiert. Jeder Autor bzw. jedes Projekt sollte also einen eigenen Namespace verwenden:

```
...
namespace a { int test; }
namespace b { int test; }
...
a::test = 1;
b::test = 2;
...
```

Für die Bildverarbeitungsbibliothek wurde der Namespace »TUVision« gewählt.

3.3.6 Vorwärtsdeklarationen

In Header-Dateien, die das Interface einer Klasse beschreiben, verweist man häufig auf andere Klassen. Damit der Compiler mit einer Header-Datei etwas anfangen kann, muß er auch die Definition der anderen verwendeten Klassen kennen.

Es gibt zwei Methode, um dieses zu erreichen. Zum einen kann man einfach in der Header-Datei die Header-Dateien der Klasse, die man verwendet, einbinden:

```
...
#include "CImageByteGray.h"
...
void foo (const CImageByteGray &img)
...
```

Dieses ist jedoch bei großen einzubindenden Header-Dateien keine gute Lösung, da so das Kompilieren recht lange dauert. Eine bessere Lösung ist in vielen Fällen die sogenannte Vorwärtsdeklaration, die man verwenden kann, wenn man die betreffende Klasse für einen Zeiger oder eine Referenz benutzen möchte. In einigen Fällen, wo z.B. Informationen über die Größe eines Objektes dieser Klasse benötigt werden, funktioniert die Vorwärtsdeklaration hingegen nicht.

Eine Vorwärtsdeklaration sieht so aus:

```
...
class CImageByteGray;
...
void foo (const CImageByteGray &img)
...
```

Man teilt dem Compiler also mit, daß »CImageByteGray« eine Klasse ist. Wie diese genau aufgebaut ist, ist an dieser Stelle unwichtig.

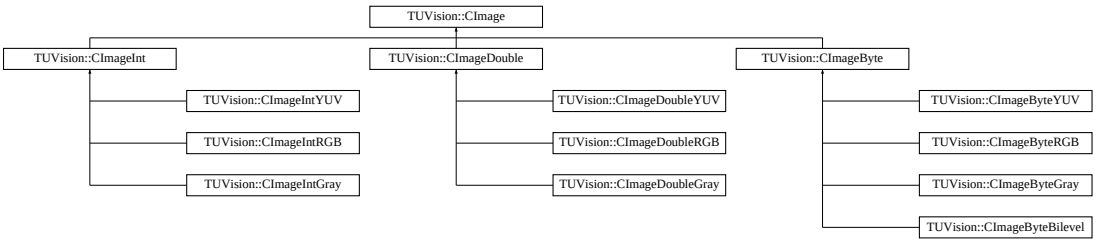


Abbildung 3.1: Vererbungsdiagramm von »CImage«

3.4 CImage

Als erster Schritt in Richtung Klassenbibliothek für die digitale Bildverarbeitung galt es, eine Klassenfamilie zu definieren, die ein Bild im Hauptspeicher des Computers repräsentiert.

Eine Untersuchung, was für Arten von Bildern es im Speicher gibt, ergab, daß man die Bilder einmal nach ihrem Zahlenformat und einmal nach ihrem Farbmodell unterteilen kann. Häufig verwendete Zahlenformate sind z.B. 8 Bit oder 16 Bit Ganzzahlen und Fließkommazahlen. Wichtige Farbmodelle sind z.B. Grau, RGB und YUV.

Bei dem Design galt es zu entscheiden, ob diese verschiedenen Bildtypen in Form von diversen Klassen oder in Form von Templates definiert werden sollten.

Bei einer direkten Implementation als Klassen hat man den entscheidenden Nachteil, daß man für jede Kombination aus Zahlenformat und Farbmodell eine eigene Klassen implementieren muß. Da die Klassen im Prinzip alle die gleiche Funktionalität besitzen, führt das zu einer sehr großen Menge an doppelten Code.

Die Template Lösung vermeidet dieses, da man hier nur für jedes Farbmodell ein eigenes Template definieren muß. Den Rest erzeugt der Compiler automatisch. Allerdings haben Templates zwei entscheidene Nachteile. Zum einen ist das Programmieren mit Templates recht schwierig, da die Fehlermeldungen der Compiler sehr unübersichtlich werden. Außerdem verliert man damit einen wichtigen Vorteil von C++: die Typsicherheit. Diese gewährleistet, daß man mit den Daten und Objekten nur das machen kann, was deren Programmierer bei ihrer Implementation vorgesehen hat. Z.B. macht es ja keinen Sinn, einen Sobel Filter auf ein binäres Bild anzuwenden. Dieses kann durch die Typsicherheit bereits zur Übersetzungszeit des Programmes verhindert werden.

Von daher wurde von Anfang an beschlossen, auf Templates in der Programmierschnittstelle zu verzichten. Folglich galt es eine Lösung zu finden, wie man die Bildtypen mit möglichst wenig Aufwand direkt als Klasse implementieren kann.

Als oberste Basisklasse dient die Klasse »CImage«. In ihr sind alle Funktionen enthalten, die von jedem Bildtyp benötigt werden. Allerdings sind nicht alle Funktionen, die hier definiert werden, auch hier implementiert. Statt dessen sind einige rein virtuell definiert und werden erst in den von »CImage« abgeleiteten Klassen implementiert.

```
class CImage
{
  public:
    virtual ~CImage();
    inline int getW() const;
    inline int getH() const;
    virtual CImageAttributes &getAttributes();
    virtual const CImageAttributes &getAttributes() const;
    virtual void setWindow (int x_0, int y_0, int width, int height);
```

```
    virtual void setWindow (const CRect &rect);
    virtual void resetWindow();
    virtual void clearImage() = 0;

  protected:
    CImage (int w, int h, int nchannels);
    CImage (const CImage &img);
    virtual void adjustRowPtr() = 0;

  protected:
    const int d_width;      // image width (in pixels)
    const int d_height;     // image height (in pixels)
    const int d_nchannels;  // number of color channels
    int d_win_x;      // window start point (x coordinate)
    int d_win_y;      // window start point (y coordinate)
    int d_win_width;  // width of the window
    int d_win_height; // height of the window
    CImageAttributes* d_img_attributes; // string and double
                                        // attributes
};
```

3.4.1 Konstruktoren

Die Klasse kennt zwei Konstruktoren. Mit dem ersten kann ein neues »CImage«-Objekt mit einer spezifizierten Breite, Höhe und Farbkanalanzahl erstellt werden.

Der zweite Konstruktor ist ein sogenannter Copy Konstruktor. Er übernimmt die oben genannten Parameter von einem bestehenden Objekt, das man ihm als Parameter übergibt.

Beide Konstruktoren setzen dann jeweils die Konstanten »d_width«, »d_height« und »d_nchannels« entsprechend. Es ist möglich, diese drei Werte konstant zu setzen, da sie nur einmal im Konstruktor gesetzt werden. Dieses ist ein spezielles Feature von C++.

3.4.2 Fenster

Ein interessantes Konzept, das in »CImage« implementiert ist, sind Fenster. Es ist möglich, bei jedem Bild ein Fenster zu setzen. Nach dem Setzen des Fensters verhält sich das Objekt so, als würde es nur den Teil des Bildes innerhalb des Fenster enthalten.

Dieses ermöglicht es dann später, Algorithmen nur auf einen bestimmten Teil eines Bildes anzuwenden. Dieses kann für sehr viele Berechnungen sehr hilfreich sein.

Ursprünglich sollten die »CImage«-Klasse statt dessen eine Funktion anbieten, die aus einem Bild-Objekt eines neues Bild-Objekt erzeugen kann, das nur einen Teil des ersten Bildes enthält. Diese Idee wurde jedoch verworfen, da man hierfür die Daten jedesmal von einem Objekt zu einem anderen Objekt hätte kopieren müssen, was viel Zeit kosten kann und insbesondere dann problematisch wird, wenn diese Operation einige tausendmal pro Bild aufgerufen werden soll.

Hier ist das Fenster-Konzept zusammen mit dem Copy Konstruktor wesentlich eleganter. Benötigt man z.B. gleichzeitig drei Ausschnitte aus dem Originalbild, so kopiert man das Bild-Objekt zweimal, so daß man das Originalbild insgesamt dreimal hat. Bei jedem dieser Objekte kann man jetzt das Fenster passend setzen. Natürlich würde man auch jetzt Speicher verschwenden, da man das Bild ja dreimal im Speicher vorhalten würde. Aus diesem Grund wurde eine Methode geschaffen, die Daten zwischen Objekten zu teilen, so daß man zwar drei Bild-Objekte hat, diese aber alle auf den gleichen Speicher

verweisen. Dazu später noch mehr.

Die Basisklasse »CImage« besitzt insgesamt drei Funktionen, die sich mit den Fenstern beschäftigen. Mit »setWindow()« kann ein Fenster gesetzt werden. Nach dem Aufruf dieser Funktion sind alle Pixel außerhalb des Fensters unsichtbar. Dieses ändert sich erst wieder nach Aufruf der Funktion »resetWindow()«, mit der ein Fenster gelöscht wird, so daß alle Pixel wieder sichtbar sind.

Beide Funktionen setzen die Variablen »d_win_x«, »d_win_y«, »d_win_weight« und »d_win_height« passend und rufen dann die Funktion »adjustRowPtr()« auf. Diese ist die dritte Funktion, die sich mit der Fensterverwaltung beschäftigt. Die Aufgabe dieser Funktion, die in »CImage« rein virtuell ist, ist es, einen Zeiger auf den ersten Pixel innerhalb des Fensters zu setzen.

3.4.3 Dimension

Die beiden Funktionen »getW()« und »getH()« liefern die Breite und Höhe des in dem Objekt enthaltenen Bildes zurück. Diese Werte werden den beiden Variablen »d_win_width« und »d_win_height« entnommen, die von den Fenster-Funktionen bzw. von den Konstruktoren gesetzt werden. Beide Variablen sind als »protected« definiert.

Das bedeutet, daß ein Nutzer dieser Klasse nicht direkt auf sie zugreifen kann, sondern dafür die beiden erwähnten Funktionen verwenden muß. Die Funktionen bieten gegenüber dem direkten Zugriff den Vorteil, daß man Schreiboperationen auf die Variablen verhindern kann. Jede Variable kann also nur über eine definierte Schnittstelle verändert werden.

Eine Definition als »protected« unterscheidet sich von einer »private«-Definition dadurch, daß Klassen, die von dieser Basisklasse abgeleitet werden - also ihre Funktionen erben - auf die Variablen zugreifen können. Dieses ist bei »private« nicht möglich.

3.4.4 Attribute

Sehr häufig enthalten Grafikdateien neben den Bilddaten noch weitere Informationen über das Bild. Z.B. ist in vielen Dateien die Auflösung des Bildes in DPI oder ein Copyright-Hinweis enthalten.

Viele auf dem Markt befindliche Bildverarbeitunssysteme verwerfen solche Informationen einfach. Liest man mit solchen Programmen ein Bild ein, bearbeitet es und speichert es wieder ab, so sind diese eventuell sehr wichtigen Informationen verloren.

Aus diesem Grund entstand bereits am Anfang des Projektes der Wunsch, diese Informationen in den Bild-Objekten mit abzuspeichern. Zuerst wurde überlegt, für bestimmte Werte einfach Variablen und entsprechende Auslesefunktionen für diese Variablen vorzusehen. Dieses wäre zwar leicht zu implementieren gewesen, hätte aber den Nachteil gehabt, daß die Lösung nicht sehr flexibel gewesen wäre. Daher wurde die Idee von Eric Sommerlade, für diese Zusatzinformationen eine neue Klasse zu entwickeln, die dann jeweils als Objekt in jedem Bild-Objekt enthalten sein sollte, implementiert. Hierzu verwendet die neue Klasse das »map<>« Template aus der C++ STL. Es ist also möglich, beliebige Werte unter einem frei wählbaren Namen in jedem Bild-Objekt abzulegen.

Interessant können die Attribute z.B. auch für Videos sein. Hier kann man in jedem Bild-Objekt ablegen, welches Bild eines Videos jeweils in dem Objekt enthalten ist.

Über die Funktion »getAttributes()« kann man eine Referenz auf das Attribut-Objekt eines Bild-Objektes erhalten. Die Funktion wurde während des Entwicklungsprozesses mehrfach umdefiniert. Ursprünglich sah die Funktion einfach so aus:

```
virtual CImageAttributes &getAttributes();
```

Diese Definition hat jedoch einen entscheidenen Nachteil, wie man im nachfolgenden Codesegment leicht erkennen kann:

```
void foo (const CImageByteGray &img)
{
  img.getAttributes();   // Fehler
}
```

Dieser Aufruf von »getAttributes« läßt sich nicht kompilieren. Da das Bild-Objekt als konstant definiert ist, können nur Member Funktionen aufgerufen werden, die auch als konstant definiert sind, siehe Abschnitt 3.3.1. Man könnte jetzt natürlich hingehen und das Objekt als nicht konstant definieren; das wäre jedoch eine sehr schlechte Lösung des Problems, da man Parameter, die man an eine Funktion übergibt, immer konstant definieren sollte, wenn sie innerhalb der Funktion nicht verändert werden. Dieses Prinzip muß man möglichst in der ganzen Bibliothek durchhalten, da sonst ihre Nutzung Probleme machen kann.

Man könnte das Problem natürlich auch lösen, indem man die Member Funktion konstant macht:

```
virtual CImageAttributes &getAttributes() const;
```

Jetzt könnte man die Funktion auch mit konstanten Objekten benutzen. Für den Compiler ist das Problem jetzt gelöst.

Allerdings wird jetzt das Prinzip des »const-correctness« [Mey95, Lak96] verletzt. Die Funktion »getAttributes« liefert nämlich jetzt eine Referenz auf ein nicht konstantes Attribut-Objekt zurück. Die Folge davon ist, daß man mit den Attribut Member Funktionen die Daten im Attribut-Objekt ändern kann. Dieses ist nicht sehr sinnvoll, da der Anwender das Bild-Objekt als konstant definiert hat und deshalb natürlich erwartet, daß der Inhalt des Objektes nicht geändert werden kann.

Eine Klasse ist dann »const-correct«, wenn es unmöglich ist, irgendwelche Daten innerhalb eines konstanten Objektes zu ändern. Hierzu zählen auch Objekte, die von diesem Objekt verwaltet werden.

Folglicherweise müßte man dann die Funktion so definieren:

```
virtual const CImageAttributes &getAttributes() const;
```

Jetzt erhält man eine Referenz, die auf ein konstantes Attribut-Objekt verweist. Es ist jetzt also nur noch möglich, Member Funktionen des Attribut-Objektes aufzurufen, die selbst als konstant definiert sind und deshalb den Inhalt des Attribut-Objektes nicht ändern können.

Jetzt ist die Funktion zwar korrekt definiert, aber man hat ein neues Problem: es ist jetzt überhaupt nicht mehr möglich, selbst wenn das Bild-Objekt nicht als konstant definiert ist, die Werte im Attribut-Objekt zu ändern. Folglich ist es jetzt völlig unmöglich, in dem Attribut-Objekt überhaupt Daten abzulegen.

Hier hilft schließlich ein Feature von C++, das Überladen genannt wird. Es ist möglich, die Funktion einer Klasse unter dem gleichen Namen mehrfach zu definieren, wenn sich die Parameter der Definitionen unterscheiden. Der Compiler wählt dann je nach übergebenen Parametern automatisch die richtige Version der Funktion aus. Neben den Parametern wählt der Compiler die Version auch danach aus, ob sie als konstant oder nicht konstant definiert ist. Ist das Objekt selbst konstant, wählt der Compiler immer die konstante Version der Member Funktion. Ist das Objekt hingegen nicht konstant, wird die nicht konstante Funktion verwendet.

Aus diesem Grund wurde die Funktion »getAttributes« zweimal definiert. Die nicht konstante Funktion liefert eine Referenz auf nicht konstante Daten zurück. Die konstante Funktion liefert dagegen eine Referenz auf konstante Daten.

3.4.5 Bild löschen

Normalerweise ist der Inhalt eines neu erzeugten Bild-Objektes undefiniert. Die Werte der Pixel sind völlig zufällig. Dieses ist, solange z.B. ein Algorithmus alle Pixel überschreibt, unproblematisch. Dieses ist jedoch nicht immer der Fall, so daß der Wunsch bestand, ein neues Bild irgendwie zu initialisieren.

Man könnte nun beim Erzeugen des Bild-Objektes einfach alle Pixel auf Schwarz setzen. Diese aufwendige Operation möchte man sich aber natürlich gerne sparen, wenn die nachfolgenden Berechnungen sowieso alle Pixel setzen und die Operation damit überflüssig ist.

Daher wurde entschieden, daß die Pixel standardmäßig nicht initialisiert werden. Mit der in »CImage« rein virtuellen Funktion »clearImage()« können aber »manuell« alle Pixel eines Bildes auf Schwarz gesetzt werden.

3.5 CImage<Zahlenformat>

Von der vorher beschriebenen Basisklasse »CImage« werden die Basisklassen für die verschiedenen Zahlenformate abgeleitet. Von diesen Klassen werden dann später die Klasse für die verschiedenen Farbmodelle abgeleitet.

Man hätte von der Basisklasse auch Basisklassen für die Farbmodelle und von diesen dann die Klassen für die Zahlenformate ableiten können. Diese Lösung wurde jedoch bewußt nicht gewählt, da die andere Lösung ein entscheidenen Vorteil hat: es können deutlich mehr Funktionen bereits in dieser Schicht implementiert werden und nicht erst in der dritten Schicht.

Insgesamt sind zur Zeit drei Zahlenformat-Basisklasse definiert: 8 Bit (Byte), 16 Bit mit Überlauf und negativem Bereich und Fließkommazahlen (Double). Das Erstellen einer Basisklasse für ein weiteres Format ist aber kein großer Aufwand; tatsächlich muß nur eine bestehende Basisklasse kopiert und etwas angepaßt werden.

```
class CImageByte : public CImage
{
  public:
    virtual inline void setPixel (int x, int y, unsigned char value,
                                  int channel);
    virtual inline unsigned char getPixel (int x, int y, int channel)
            const;
    unsigned char *getRow (int row);
    const unsigned char *getRow (int row) const;
    void clearImage();

  protected:
    CImageByte (int w, int h, int nchannels);
    CImageByte (const CImageByte &img, bool copy);
    void adjustRowPtr (void);
    void insertSubImage (const CImageByte &img, int x, int y);
    bool isEqual (const CImageByte &img);
    bool isUnequal (const CImageByte &img);

  private:
    CImageByte (const CImageByte &img);

  protected:
    CImageData<unsigned char>* d_img_data; // reference to data
    unsigned char *d_img_pnt; // pointer to the first available pixel
```

```
};
```

3.5.1 Konstruktoren

Diese Schicht definiert insgesamt zwei Konstruktoren.

Zum einen exisitiert ein Copy Konstruktor, der im Gegensatz zu einem normalen Copy Konstruktor allerdings noch einen zusätzlichen Parameter erwartet, der festlegt, ob das neue Objekte auf die gleichen Daten wie das zu kopierende Objekt verweisen soll oder ob auch die Daten selbst kopiert werden sollen. Dieses Feature ist inbesondere im Zusammenhang mit den Fenstern hilfreich. Es erlaubt, daß mehrere Bild-Objekte auf die gleichen Bilddaten im Speicher verweisen und trotzdem unterschiedliche Fenster gesetzt haben können. Standardmäßig werden die Daten kopiert.

Der Copy Konstruktor ruft zuerst den Copy Konstruktor der Basisklasse »CImage« auf. Um das Teilen eines Speicherbereiches zwischen mehrere Objekten zu ermöglichen, wurde die Template Klasse »CImageData« erstellt, die Reference Couting beherrscht. Diese Klasse wurde von Eric Sommerlade erstellt, so daß deren Funktionalität in dieser Diplomarbeit nicht genauer beleuchtet wird.

Des weiteren existiert ein Konstruktor, der ein neues Bild-Objekt durch Angabe von Breite, Höhe und Farbkanalanzahl erzeugt. Auch dieser Konstruktor ruft den passenden Konstruktor von »CImage« auf. Danach wird über »CImageData« ein passender Speicherbereich reserviert und das Fenster zurückgesetzt.

3.5.2 Fenster

In dieser Schicht ist die Funktion »adjustRowPtr()« implementiert. Sie berechnet einen Zeiger, der auf den ersten Pixel eines Bildes zeigt. Hierbei wird ein aktiviertes Fenster berücksichtigt. Die Berechnung folgt folgender Formel:

$$d\_img\_pnt = d\_img\_data + (d\_win\_y \cdot d\_width + d\_win\_x)$$

»d_img_data« ist ein Zeiger auf den ersten Pixel ohne Fenster. »d_win_x« und »d_win_y« beschreibt den Startpunkt des Fensters in den Koordinaten des Originalbildes. »d_width« ist die Breite des Originalbildes.

3.5.3 Zugriff auf die Pixel

Die Klassen enthalten zwei verschiedene Konzepte, um die einzelnen Pixel eines Bildes zu lesen und zu schreiben. Beide Konzepte haben Vor- und Nachteile, weshalb beide implementiert wurden, so daß der Anwender frei wählen kann, welches Konzept für ihn am besten geeignet ist.

Die erste Zugriffsmethode stellt die Funktion »getRow()« dar. Sie liefert einfach einen Zeiger auf eine bestimmte Zeile des Bildes im Speicher zurück. Hierbei wird ein gesetztes Fenster berücksichtigt. Bei dieser Funktion traten die gleichen Probleme wie in Abschnitt 3.4.4 auf, so daß sie überladen implementiert wurde, um die »const-correctness« sicherzustellen.

Bevor »getRow()« einen Zeiger zurückliefert, wird überprüft, ob die gewünschte Zeile überhaupt existiert. Ist das nicht der Fall, wird eine Exception ausgelöst, so daß die Anwendung nicht in fremden Speicher schreiben kann, was zu einem Absturz führen würde.

Der zurückgelieferte Zeiger berechnet sich als:

$$d\_img\_pnt + row \cdot (d\_width \cdot d\_nchannels)$$

Vorteilhaft ist diese Methode, wenn man auf mehr oder weniger alle Pixel einer Zeile bzw. eines Bildes hintereinander zugreifen möchte.

Bei einem zufälligen Zugriffsmuster auf die Pixel eines Bildes ist diese Methode nicht sehr gut geeignet. Für solche Fälle ist man mit den Funktionen »setPixel()« und »getPixel()« besser bedient. Den Wert, den man mit diesen Funktionen setzen bzw. lesen möchte, spezifiziert man durch drei Parameter: x, y und den Farbkanal. Daraus berechnen die Funktionen nach der folgenden Formel die richtige Position im Speicher:

$$position = (x + y \cdot d\_width) \cdot d\_nchannels + channel$$

Wie man leicht sehen kann, müssen diese Funktionen langsam sein, da für jeden Pixel Zugriff zwei Additionen und zwei Multiplikationen durchgeführt werden müssen. Noch schlechter wird die Leistung im Vergleich zu dem »getRow()«-Konzept, wenn man einen Wert ausliest, bearbeitet und dann wieder an der gleichen Stelle abspeichert. Dann wird die Positionsberechnung gleich zweimal durchgeführt.

Da die Funktion in der zweiten Schicht der Klassenstruktur angeordnet ist, benötigt man die Berücksichtigung von Farbkanälen. Dieses ist bei Graustufenbildern eigentlich überflüssig, so daß man bei diesen eigentlich mit einer Addition und einer Multiplikation auskommen würde. Hier werden die Kosten einer allgemeinen Implementierung bemerkbar. Lösen ließe sich das Problem aber relativ leicht, indem man die beiden Funktionen in den Klassen »CImage<Zahlenformat>Gray« überlädt und den Farbkanalparameter einfach ignoriert.

Beide Funktionen sind inline deklariert. Damit gibt man dem Compiler den Hinweis, daß er möglichst nicht die Funktion benutzen soll, sondern ihren Inhalt an die Stelle kopieren soll, wo der Benutzer die Funktion aufruft. Aus z.B.

```
inline int foo (int a)
{
  return a;
}

int main (void)
{
  int b;

  b = foo (10);
}
```

macht der Compiler intern dann sowas wie

```
int main (void)
{
  int b;

  b = 10;
}
```

Man spart damit also, wenn der Compiler den Ratschlag denn befolgt, den Overhead für einen Funktionsaufruf, der gerade bei einer Funktion, die jeweils nur einen Pixel ändert, erheblich sein kann. Außerdem kann der Compiler den Code auch sonst besser optimieren. Ein kleiner Nachteil kann sein, daß das ausführbare Programm größer wird, da ja jetzt an jede Aufrufstelle der Funktion deren Funktionalität kopiert werden muß.

3.5.4 Vergleichsoperationen

Mit den beiden Funktionen »isEqual()« und »isUnequal()« kann geprüft werden, ob das Bild in dem Objekt mit dem Bild in einem anderen Objekt, das als Parameter übergeben wird, übereinstimmt.

Hierbei wird zuerst getestet, ob die Breite und Höhe der beiden Bilder übereinstimmen. Wenn das nicht der Fall ist, kann man sich weitere Tests gleich sparen. Ansonsten werden die Werte aller Pixel miteinander verglichen.

Die beiden Funktionen sind wie einige andere auch nicht öffentlich sondern als »protected« definiert, so daß man sie von außen nicht benutzen kann. Benutzt werden die beiden Funktionen ausschließlich von den Operatoren »==« und »!=«, die die hiervon abgeleiteten Klassen anbieten. Die Operatoren in den abgeleiteten Klassen sind also nur Wrapper Funktionen. Man kann die Operatoren selbst nicht in der zweiten Schicht definieren, da diese ansonsten nicht vollständig typsicher wären.

Beide Funktionen erwarten ein Bild als Parameter vom Typ »CImage<Zahlenformat>«. Würde man die Operatoren direkt hier implementieren oder die beiden Funktionen öffentlich machen, wäre es möglich, z.B. ein Graustufenbild mit einem RGB-Bild zu vergleichen, was keinen Sinn machen würde.

3.5.5 Bild einfügen

Mit der Funktion »insertSubImage()« kann ein Bild an einer bestimmten Stelle in ein anderes Bild kopiert werden. Es ist mittels dieser Funktions z.B. möglich, mehrere Bilder zu einem zusammenzufassen.

Falls sich ein Bild aufgrund seiner Größe nicht in ein anderes Bild einfügen läßt bzw. falls der angegebene Punkt, wo das Bild einzufügen ist, außerhalb des Bildes liegt, wirft diese Funktion eine passende Exception vom Typ »CExceptionRange«.

Auch diese Funktion ist »protected« definiert, damit sie nicht direkt vom Anwender benutzt werden kann, der statt dessen die typsichere Funktion mit dem gleichen Namen aus der dritten Schicht verwenden kann, die lediglich eine Wrapper Funktion um diese Funktion der zweiten Schicht darstellt.

3.6 CImage<Zahlenformat><Farbmodell>

Von den Klassen »CImage<Zahlenformat>« werden die Klassen der dritten Schicht abgeleitet. Die Klassen dieser Schicht sind diejenigen, die direkt vom Benutzer genutzt werden können. Für jede Kombination aus Zahlenformat und Farbmodell existiert somit eine eigene Klasse. Für Graustufenbilder mit 8 Bit Auflösung sieht die Klasse der dritten Schicht so aus:

```
class CImageByteGray : public CImageByte
{
  public:
    CImageByteGray (int w, int h);
    CImageByteGray (int w, int h, const unsigned char *data);
    CImageByteGray (const CImageByteGray &img, bool copy=true);
    ~CImageByteGray();
    void insertSubImage (const CImageByteGray &img, int x, int y);
    bool operator== (const CImageByteGray &img) const;
    bool operator!= (const CImageByteGray &img) const;

  private:
    CImageByteGray& operator= (const CImageByteGray&);
};
```

Die dritte Schicht der Klassenstruktur enthält kaum Code, da die meisten Funktionen reine Wrapper Funktionen sind, die einfach Funktionen der ersten oder zweiten Schicht aufrufen. Die Hauptfunktion dieser Schicht ist es, das Interface der Bild-Objekte typsicher zu machen.

Außerdem wurde die Implementation so ausgelegt, daß sie schnell um neue Zahlenformate und Farbmodelle erweitert werden kann. Der Portierungsaufwand ist dabei sehr klein. Es ist möglich, in wenigen Minuten eine Klasse für ein neues Farbmodell zu erstellen.

3.6.1 Konstruktoren

Es existieren insgesamt drei Konstruktoren. Der einfachste Konstruktor erzeugt einfach ein neues Bild mit der angegebenen Breite und Höhe. Er bedient sich dazu des passenden Konstruktors der zweiten Schicht. Die Farbkanalanzahl muß der Anwender nicht angeben, da sich diese ja aus dem Typ der jeweiligen Klasse ergibt.

Ein weiterer Konstruktor erzeugt ein neues Bild, indem er ein bestehendes Bild aus einem Array kopiert. Dieses ist vor allem als Schnittstelle zu altem ANSI C Code sinnvoll, da man hier Bilder in der Regel einfach als Array speichert und an Funktionen übergibt. Der Konstruktor ruft den Konstruktor der zweiten Schicht auf und kopiert dann einfach die übergebenen Daten in das Bild-Objekt.

Schließlich gibt es noch einen Copy Konstruktor, bei dem der Benutzer wählen kann, ob die Bilddaten kopiert werden oder sich beide Objekte ein Bild im Speicher teilen sollen. Auch dieser Konstruktor ist einfach nur ein Wrapper um den entsprechenden Konstruktor der zweiten Schicht.

3.6.2 Operatoren

Es werden die beiden Operatoren »==« und »!=« angeboten, die die Funktionen »isEqual()« und »isUnequal()« aus der zweiten Schicht aufrufen. Dank dieser Operatoren kann der Benutzer dann zwei Bilder auf folgende Weise vergleichen:

```
...
CImageByte *img1, *img2;
...
if (*img1 == *img2)
  foo();
...
```

Außerdem ist noch der Zuweisungsoperator »=« als »private« definiert. Standardmäßig erzeugt der C++ Compiler für jede Klasse automatisch einen Zuweisungsoperator [Mey95]. Dieser kopiert bei einer Zuweisung einfach den Inhalt der Variablen. Es wird also einfach ein Zeiger jedoch nicht der Inhalt, worauf dieser zeigt, kopiert. Das bedeutet, daß nach dem Kopiervorgang die Zeiger beider Objekte auf den gleichen Bereich im Speicher zeigen. Dieses Verhalten ist selten gewollt.

Der Entwickler hat jetzt zwei Möglichkeiten: er schreibt den Zuweisungsoperator so um, daß er für das jeweilige Anwendungsgebiet korrekt funktioniert. Hier müßte er also z.B. den Speicher des Bildes kopieren. Eine zweite Möglichkeit ist, den Zuweisungsoperator zu deaktivieren. Dieses gelingt dadurch, daß man ihn als »private« definiert.

Bei der Bildverarbeitungsbibliothek wurde sich für die zweite Methode entschieden, da der Copy Konstruktor bereits die gleiche Aufgabe erfüllt. Außerdem macht der Copy Konstruktor deutlicher, daß es sich um eine »teure« Aktion handelt.

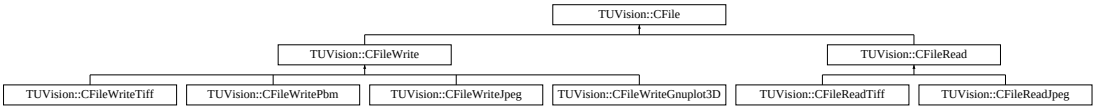


Abbildung 3.2: Vererbungsdiagramm von »CFile«

3.7 I/O-Klassen

Die Bildverarbeitungsbibliothek verfügt über einige I/O-Klassen, mit denen es möglich ist, Bilder in einer Datei auf der Festplatte zu speichern und wieder aus einer solchen zu lesen.

3.7.1 CFile

Alle I/O-Klassen verfügen über eine Anzahl von Standardfunktionen, die bei allen I/O-Klassen gleich sind und durch die abstrakten Basisklassen »CImage«, »CImageRead« und »CImageWrite« definiert werden, wobei die letzten beiden Klassen von der ersten Klasse abgeleitet sind; siehe Abbildung 3.2. Durch diese Standardfunktionen ist es für den Anwender sehr leicht, in seinem Programm von einem Dateiformat zu einem anderen zu wechseln. Er muß hierfür lediglich eine Zeile in seinem Programm ändern.

```
class CFile
{
  public:
    virtual ~CFile() {};
    virtual void open (const std::string &filename) = 0;
    virtual void close (void) = 0;

  protected:
    std::string  d_filename;   /* name of the file */
};
```

Die oberste Basisklasse »CImage« definiert die beiden Funktionen »open()« und »close()«, mit denen eine Grafikdatei geöffnet und geschlossen werden kann. Beide Funktionen sind rein virtuell und werden erst später implementiert. Da der Destruktor einer jeden I/O-Klasse beim Zerstören des I/O-Objektes automatisch die offene Datei schließt, braucht die »close()«-Funktion nur dann benutzt werden, wenn mit »open()« eine neue Datei geöffnet werden soll und deshalb vorher die alte geschlossen werden muß.

Da man manche I/O-Klassen eventuell nur für Lese- oder Schreiboperationen implementieren möchte bzw. kann und da es problematisch ist, wenn eine Datei gleichzeitig gelesen und geschrieben werden kann, existieren getrennte Basisklassen für Lese- und für Schreiboperationen, nämlich die bereits vorher erwähnten Klassen »CFileRead« und »CFileWrite«.

3.7.2 CFileRead

Die Basisklasse »CFileRead« dient als einheitliche Programmierschnittstelle für alle I/O-Klassen, die Leseoperationen anbieten.

```
class CFileRead : public CFile
{
  public:
```

```
    virtual CImage *read (int img_number = 0) = 0;
    virtual int getNumberImages (void) = 0;
};
```

Die Funktion »read()« liest eine Bild ein und gibt es als Bild-Objekt vom Typ »CImage« zurück. Der Funktion kann als Parameter übergeben werden, welches Bild einer Datei gelesen werden soll. Dieses ist sinnvoll, da einige Grafikformate wie z.B. TIFF das Speichern von mehreren Bildern in einer Datei erlauben. Außerdem ist es durch diesen Parameter möglich, diese Funktion auch für Videos zu benutzen.

Für die Leseoperation ist es natürlich wichtig, zu wissen, wieviele Bilder überhaupt in der Datei abgespeichert sind. Diese Zahl ermittelt die Funktion »getNumberImages()«. Falls es bei einem Format nicht möglich ist, diese Anzahl vorab zu ermitteln, da die Daten z.B. gestreamt werden, liefert die Funktion einen Fehler zurück.

Bei der Definition der »read()«-Funktion gab es ein Problem. Die verschiedenen Grafikformate unterstützen jeweils nur Bilder mit einigen bestimmten Zahlenformaten und Farbräumen. So unterstützt z.B. das PBM-Format beliebige Zahlenformate aber nicht den YUV-Farbraum. Das TIFF-Format unterstützt z.B. nur bestimmte Zahlenformate, dafür aber eine große Auswahl an Farbräumen. Da die »read()«-Funktion ein Bild-Objekt mit dem gleichen Zahlenformat und Farbraum, wie sie in der Bilddatei Verwendung finden, zurückliefern soll, ist es schwierig, eine allgemeine API zu definieren.

Für das Problem gibt es mindestens zwei Lösungsansätze. Zum einen könnte man in der Basisklasse die »read()«-Funktion für alle »CImage«-Klassen definieren:

```
class CFileRead : public CFile
{
  ...
  virtual read (CImageByteGray &img, int img_number = 0) = 0;
  virtual read (CImageByteRGB &img, int img_number = 0) = 0;
  ...
};
```

In den davon abgeleiteten Implementationen für die jeweiligen Dateiformate würden dann die Funktionen implementiert werden, wobei die Funktionen für die nicht unterstützten Formate keine Funktion hätten.

Diese Lösung hat jedoch zwei entscheidene Nachteile. Die Klassen wären nicht mehr typsicher. Man könnte die Funktionen für Bildtypen aufrufen, die von dem Dateiformat selbst garnicht unterstützt werden. Außerdem wäre es schwierig, verschiedene Bildformate mit einem Programm zu lesen, da bereits während der Übersetzung des Programmes festgelegt würde, daß die Grafikdatei z.B. als 8 Bit RGB-Bild im Speicher abgelegt werden soll. Handelt es sich bei der Datei auf der Festplatte aber z.B. um eine Datei mit YUV-Farbraum hat man ein Problem.

Beide Probleme kann man sehr schön durch die C++-Konzepte RTTI und Polymorphismus lösen [Str00]. Die »read()«-Funktion liefert jetzt nicht einen bestimmten Typ aus der dritten Schicht der Bildtypen zurück, sondern die oberste Basisklasse »CImage« aus der ersten Schicht; siehe Abbildung 3.1. Da alle Bildtypen von dieser Basisklasse abgeleitet sind, kann ein Zeiger vom Typ der Basisklasse dank Polymorphismus zur Repräsentation eines beliebigen Bild-Objektes dienen. Mit einem Zeiger vom Typ »CImage« kann man allerdings nicht viel machen, da man jetzt auch nur die Funktionen aufrufen kann, die in »CImage« definiert sind. Alle anderen Member Funktionen sind nicht aufrufbar.

Um mit dem Bild weiterarbeiten zu können, muß der Zeiger in einen Zeiger auf z.B. »CImageByteGray« umgewandelt werden. Dieses geschieht mit der C++-Funktion »dynamic_cast<>()«

```
...
CImage *CFileReadTiff::read (int img_number = 0)
```

```
{
  ...
  CImageByteGray *img;
  ...
  img = new CImageByteGray (100, 100);
  ...
  return img;
}
...
void foo (void)
{
  ...
  CFileReadTiff   src;
  CImageByteGray  *img;
  ...
  img = dynamic_cast<CImageByteGray*> (src.read());
  ...
}
...
```

Im Gegensatz zu einem ANSI C Cast wird hier der Typ des Zeiger nicht einfach blind während der Übersetzung des Programmes geändert, sondern es findet zur Laufzeit eine Überprüfung statt, auf was für einen Typ von Objekt der von »read()« gelieferte Zeiger wirklich verweist. Falls der Zeiger nicht auf ein Objekt des angegebenen Typs zeigt, liefert »dynamic_cast<>()« zur Laufzeit einen Fehler.

Im wissenschaftlichen Bereich ist das Zahlenformat und das Farbmodell der Eingangsdaten meistens bekannt. Es kann jedoch Fälle geben, wo dieses nicht der Fall ist und man zur Laufzeit feststellen möchte, welches Farbmodell z.B. das Bild in der Datei hat. Hier hilft die C++-Funktion »typeid()«, mit der der Typ eines Objektes ermittelt werden kann.

3.7.3 CFileWrite

Bei »CFileWrite« handelt es sich um die Basisklasse, von der alle I/O-Klassen abgeleitet werden, die ein Bild speichern möchten. Im Gegensatz zu »CFileRead« wird in dieser Basisklasse keine neue Funktion definiert. Sie dient ausschließlich dazu, alle Klassen für Schreiboperationen zu gruppieren.

```
class CFileWrite : public CFile
{
};
```

Die Dokumentation dieser Basisklasse beschreibt, welche Funktionen hiervon abgeleitete Klassen zu implementieren haben. Jede Klasse hat mindestens eine »write()«-Funktion anzubieten, mit der ein Bild gespeichert werden kann. Die Funktion ist hierbei typsicher zu implementieren und es sind nur Typen anzubieten, die das Dateiformat direkt unterstützt.

Weder das Zahlenformat noch das Farbmodell sollen in der »write()«-Funktion verändert werden. Es ist die Aufgabe des Benutzers, die Objekte vor dem Aufruf der »write()«-Funktion passend zu konvertieren. Die Klassenbibliothek bringt hierfür entsprechende Klassen und Funktionen mit. Eine Konvertierung innerhalb der »write()«-Funktion ist aus diversen Gründen keine gute Idee.

Zum einen würden dadurch die Konvertierungsroutinen immer und immer wieder neu implementiert werden müssen. Nämlich für jedes Dateiformat einmal. Zum anderen ist es keine gute Idee, ein Bild mit Double-Auflösung einfach als 8 Bit Ganzzahlenbild abzuspeichern, ohne den Benutzer hierüber zu informieren. Denn dieser wird dann eventuell viel Zeit in die Fehlersuche investieren, da er sich

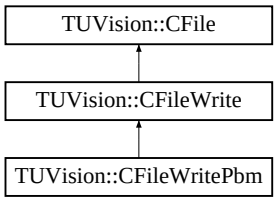


Abbildung 3.3: Vererbungsdiagramm von »CFileWritePbm«

über die schlechte Qualität seiner Berechnungen wundert. Gleiches gilt für die interne Wandlung des Farbmodelles.

3.7.4 PBM-, PGM- und PPM-Format

Die drei Formate PBM, PGM und PPM, die alle zu einer Formatfamilie gehören, sind unter Linux und Unix recht weit verbreitet, da das Format sehr einfach gehalten ist und sich deshalb schnell implementieren läßt.

Die einzelnen Pixel eines Bildes werden bei diesen Formaten einfach als Zahlen in einer ASCII-Datei durch Leerzeichen voneinander getrennt abgelegt [Pos91]. Dieses Konzept hat natürlich den Nachteil, daß sehr viel Platz verbraucht wird. Ein 8 Bit Pixel eines Graustufenbildes belegt z.B. 4 Bytes.

Das Format ist für Schwarz-/Weiß-, Graustufen- und RGB-Bilder definiert. Da der kleinste und größte mögliche Wert eines Pixels am Anfang der Datei spezifiziert wird, läßt sich dieses Format für Byte und für Integer Bilder verwenden.

Die Klasse »CFileWritePbm« implementiert dieses Format; siehe auch das Vererbungsdiagramm in Abbildung 3.3. Zur Zeit werden allerdings nur 8 Bit Bilder unterstützt, da keines der Mitglieder des Entwicklungsteams zur Zeit Bedarf an diesem Format hat. Im Gegensatz zu den anderen I/O-Klassen in der Bildverarbeitungsbibliothek benötigt diese Klasse keine weiteren externen Bibliotheken für den Zugriff auf die Grafikdateien, so daß sich diese Klasse gut für Tests auf Systemen eignet, wo diese sonst notwendigen Bibliotheken noch nicht verfügbar bzw. installiert sind.

```
class CFileWritePbm: public CFileWrite
{
  public:
    CFileWritePbm (void);
    ~CFileWritePbm();

    void open (const std::string &filename)
         throw (CException);
    void close (void)
         throw ();
    void write (const CImageByteGray &img)
         throw (CException);
    void write (const CImageByteRGB &img)
         throw (CException);
    void write (const CImageByteBilevel &img)
         throw (CException);

    private:
      std::ofstream  hd;       // file i/o class
};
```

Neu bei dieser Definition der Klasse ist die Verwendung von »throw()«. In Abschnitt 3.3.3 haben wir dieses Schlüsselwort bereits im Zusammenhang mit dem Werfen von Exceptions kennengelernt. Wie bei ANSI C und C++ üblich hat aber auch dieses Schlüsselwort mehrere Aufgaben. Im Zusammenhang mit der Definition von Funktion dient es der Spezifikation, welche Exceptions eine Funktion werfen kann [Str00].

Durch die Spezifikation werden zwei Dinge erreicht. Zum einen weiß der Benutzer jetzt, welche Exceptions auftreten können und welche er somit fangen muß. Dieses ist wichtig, da eine nicht gefangene Exception zu einem Programmabbruch führt. Außerdem sorgt der Compiler jetzt dafür, daß die Funktion auch nur diese Art von Exceptions werfen kann. Alle anderen Arten von Exceptions führen zu einem Programmabbruch.

Alle Klassen der dritten Schicht der »CFile«-Hierarchie benutzen diese Spezifikation der möglichen Exceptions. Da sich die Entwickler der Bibliothek nicht einigen konnten, ob diese Spezifikationen Sinn machen, wurde sie bei den anderen Teilen der Klassenbibliothek bisher nicht eingesetzt.

Auch die Meinungen der C++ Experten zu Exceptions an sich und zur Spezifikation der möglichen Exceptions einer Funktion gehen auseinander [Mey97]. Hauptproblem ist, daß Unterfunktionen einer Funktion selbst wieder Exceptions auslösen können und es teilweise sehr schwer zu durchblicken ist, welche Exceptions in der Summe auftreten können.

3.7.5 TIFF-Format

Neben dem im vorherigen Abschnitt erwähnten Format für die verlustfreie Speicherung von Bildern unterstützt die Bibliothek ein weiteres verlustfreies Bildformat: TIFF [Ald92]. Dieses Format bietet gegenüber dem PBM-Format diverse Vorteile:

- Es werden sehr viele verschiedene Farbräume unterstützt.
- Es können mehrere Bilder in einer Datei abgelegt werden.
- Das Format ist erweiterungsfähig ausgelegt, so daß sich neue Konzepte wie z.B. ICC-Profile problemlos haben einbinden lassen.
- Die Bilder belegen weniger Platz, da es sich um ein binäres Format handelt und da verschiedene Komprimierungen der Daten unterstützt werden.

Da dieses Format recht komplex ist, wurde für das Lesen und Schreiben von TIFF-Dateien auf eine externe Bibliothek zurückgegriffen: libtiff. Diese von Sam Leffler entwickelte Bibliothek, wird von eigentlich allen Programmen unter Linux und Unix verwendet, die TIFF-Dateien verarbeiten. Da die Bibliothek in portablen ANSI C implementiert worden ist, läßt sie sich wie die Bildverarbeitungsbibliothek unter eigentlich allen Betriebssystemen übersetzen. Durch die große Verbreitung dieser Bibliothek und durch die Verfügbarkeit des Source Codes ist die Bibliothek sehr gut getestet und läuft deshalb sehr stabil.

Die Bibliothek enthält aufgrund von Patentproblemen seit einiger Zeit nicht mehr die sehr leistungsfähige Lempel-Ziv & Welch (LZW) Komprimierung, die auch für das Projekt dieser Diplomarbeit sehr hilfreich gewesen wäre.

Die Definition der Klasse »CFileReadTiff« unterscheidet sich kaum von der Basisklasse, von der sie abgeleitet wurde:

```
class CFileReadTiff : public CFileRead
{
  public:
    CFileReadTiff (void);
    ~CFileReadTiff();
    void open (const std::string &filename)
```

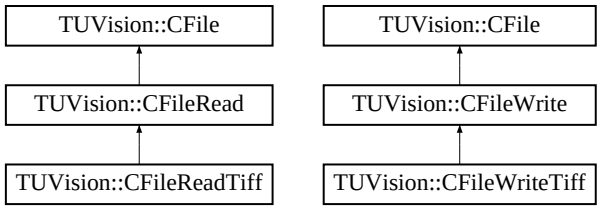


Abbildung 3.4: Vererbungsdiagramme von »CFileReadTiff« und »CFileWriteTiff«

```
        throw (CException);
    void close (void)
        throw();
    CImage *read (int img_number = 0)
        throw (std::exception, CException);
    int getNumberImages (void)
        throw (CException);

  private:
    void readAttributes (CImage &img);

  private:
    TIFF  *d_tif_hd;   // handle of libtiff
};
```

Neu hinzugekommen ist vor allem die private Funktion »readAttributes()«, die von der »read()«-Funktion benutzt wird, um die beiden Werte »xdpi« und »ydpi« aus der TIFF-Datei zu lesen und diese dann im Attribut-Objekt des Bild-Objektes abzuspeichern.

Außerdem wurden auch hier die Exceptions spezifiziert, die bei den jeweiligen Funktionen auftreten können.

```
class CFileWriteTiff: public CFileWrite
{
  public:
    CFileWriteTiff (void);
    ~CFileWriteTiff();
    void open (const std::string &filename)
        throw (CException);
    void close (void)
        throw();
    void write (const CImageByteBilevel &img)
        throw (CException, std::exception);
    void write (const CImageByteGray &img)
        throw (CException);
    void write (const CImageByteRGB &img)
        throw (CException);
    void setParameters (bool compression)
        throw();

  private:
    bool writeAttributes (const CImage &img);
```

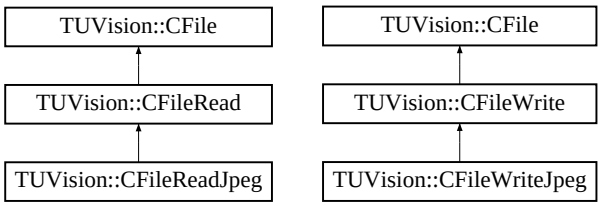


Abbildung 3.5: Vererbungsdiagramme von »CFileReadJpeg« und »CFileWriteJpeg«

```
  private:
    TIFF  *d_tif_hd;        // handle of libtiff
    bool  d_compression;    // Should we compress the image?
};
```

Die Klasse »CFileWriteTiff« unterstützt wie die Klasse »CFileReadTiff« zur Zeit drei Bildformate: s/w, Graustufen und RGB mit jeweils 8 Bit Auflösung. Die Funktion »write()« ist dreifach überladen.

Zusätzlich zu den Standardfunktionen existieren noch zwei weitere Funktionen. Mit »setParameters()« kann die Komprimierung ein- bzw. ausgeschaltet werden. Leider hat diese Funktion je nach Bildtyp keine Funktion mehr, da die LZW-Komprimierung nicht mehr von der libtiff unterstützt wird. Mit »writeAttributes()« speichern die »write()«-Funktionen einige Bildattribute in der TIFF-Datei.

Die libtiff Bibliothek entspricht in großen Teilen dem objektorientierten Konzept. Nur die Fehlerbehandlungsroutinen der Bibliothek tun dieses leider nicht. Die Funktionen der libtiff Bibliothek melden Fehler zum einen durch den Rückgabewert der Funktionen und zum anderen durch eine globale Fehlerbehandlungsfunktion. Da die Bildverarbeitungsbibliothek nur die Rückgabewerte benötigt, wird im Konstruktor der TIFF-Klassen diese globale Fehlerbehandlungsfunktion abgeschaltet. Da diese Funktion aber global ist, ist es durchaus möglich, daß eine weitere verwendete Bibliothek diese Fehlerbehandlungsfunktion wieder aktiviert. Hier wäre es deutlich besser gewesen, wenn die libtiff Bibliothek für jede geöffnete Datei eine eigene Fehlerbehandlungsfunktion anbieten würde.

3.7.6 JPEG-Format

Im Gegensatz zum TIFF-Format handelt es sich bei dem JPEG-Format um eine verlustbehaftete Komprimierung. Diese bietet vor allem den Vorteil, daß sich die Bilder wesentlich stärker komprimieren lassen als bei einer verlustfreien Komprimierung. Nur mit dem JPEG-Format ist es überhaupt möglich, größere Bilddateien über eine Verbindung mit einer geringen Bandbreite zu übertragen.

Die Schnittstelle der Klasse »CFileRead« sieht sehr ähnlich aus wie die Schnittstellen der anderen Klassen zum Lesen von Bildern:

```
class CFileReadJpeg : public CFileRead
{
  public:
    CFileReadJpeg (void);
    ~CFileReadJpeg();
    void open (const std::string &filename)
         throw (CException);
    void close (void)
         throw();
    CImage *read (int img_number = 0)
           throw (CException, std::exception);
    int getNumberImages (void)
```

```
        throw();

  private:
    std::FILE  *d_hd;          // file handle
};
```

Auch für das Schreiben und Lesen von JPEG-Dateien wird auf eine externe Bibliothek zurückgegriffen, die im Linux- und Unix-Bereich sehr weit verbreitet ist: libjpeg. Die Bibliothek unterstützt lediglich zwei Bildformate: 8 Bit Graustufen und 8 Bit RGB. Dieses spiegelt sich auch in der Schittstelle der Klasse »CFileWriteJpeg« wieder:

```
class CFileWriteJpeg: public CFileWrite
{
  public:
    CFileWriteJpeg (void);
    ~CFileWriteJpeg();
    void open (const std::string &filename) throw (CException);
    void close (void) throw();
    void write (const CImageByteGray &img) throw (CException);
    void write (const CImageByteRGB &img) throw (CException);
    void setQuality (int quality) throw();
    void setProgression (bool progression) throw();

  private:
    void write (const CImageByte &img, int color_space)
         throw (CException);

  private:
    std::FILE  *d_hd;          // file handle
    int        d_quality;      // quality/compression level
    bool       d_progression;  // Create progressive JPEG?
};
```

Die beiden als »public« definierten »write()«-Funktionen bieten selbst keinerlei Funktionalität, sondern rufen einfach die als »private« definierte »write()«-Funktion auf. Diese Funktion muß aus Gründen der Typsicherheit »private« definiert sein, da man sie ansonsten mit einem Bildformat aufrufen könnte, das von der Bibliothek nicht unterstützt wird.

Neben den üblich Funktionen enthält die Klasse zwei weitere. Mit »setQuality()« kann die Qualität des abzuspeichernden Bildes durch eine Zahl aus dem Wertebereich 0 bis 100 festgelegt werden, wobei größere Zahlen einer höheren Qualität und damit aber auch einer größeren Dateigröße entsprechen.

Mit der Funktion »setProgression()« kann festgelegt werden, ob das Bild im Progression oder im normalen Modus abgespeichert werden soll. Der Progression Modus ist vor allem dann interessant, wenn die Bilder über eine Verbindung mit einer niedrigen Bandbreite übertragen werden sollen und man das Bild schon während der Übertragung betrachten möchte. Am Anfang wird der Empfänger das Bild nur sehr grob und unscharf erkennen können und mit der Zeit wird die Qualität dann immer besser werden. Am Ende der Übertragung liegt dann ein ganz normales Bild vor.

Bei der Nutzung der Bibliothek libjpeg kam es zu diversen Problemen, die darauf schließen lassen, daß die Entwickler der Bibliothek nicht mit den Methoden moderner Softwareentwicklung vertraut sind.

Gleich am Anfang trat das Problem auf, daß der Linker meldete, daß die Funktionen der libjpeg nicht zu finden seien. Bei der Untersuchung der Header Files der libjpeg Bibliothek stellte sich dann heraus, daß ihr Entwickler vergessen hatte, für C++ das »Name Mangling« abzuschalten [Mey97]. Name Mangling

ist zwar ein reines C++ Problem und deshalb war es auch nicht zwingend notwendig, sich bei einer C-Bibliothek darüber Gedanken zu machen. Jedoch führt dieses dazu, daß die Bibliothek aus C++ heraus nicht direkt benutzt werden kann.

Die Art und Weise wie C und C++ die Namen einer Funktion in einer übersetzten Objektdatei für den Linker ablegen, unterscheiden sich, da C++ einige neue Möglichkeiten wie z.B. das Überladen von Funktionen für den Linker abbilden muß. Deswegen muß dem C++ Compiler bekannt gemacht werden, daß es sich um C Code handelt. Dieses hätte der Autor der Bibliothek durch eine kleine Änderung an seinen Header Files erreichen können:

```
#ifdef __cplusplus
extern "C" {
#endif

  ... die eigentlichen Funktionen ...

#ifdef __cplusplus
}
#endif
```

Um dieses Problem zu lösen, sieht das Einbinden der Header Files in »CFileReadJpeg« und »CFileWriteJpeg« etwas anders wie gewöhlich aus:

```
extern "C"
{
  #include <jpeglib.h>
}
```

Dieses war jedoch nicht das einzige Problem, das die Nutzung der Bibliothek in C++ erschwerte. Der Entwickler der libjpeg Bibliothek benutzt wie viele andere C-Programmierer nicht durchgängig das Schlüsselwort »const«, das auch schon in ANSI C existiert und in C++ nur erweitert wurde. Insbesondere erwarten einige Funktionen Zeiger auf variable Datenbereiche, obwohl die Funktionen diese Daten garnicht verändern. Hier hätten eigentlich Zeiger auf konstante Daten benutzt werden müssen. Dieses bereitet vor allem in der Funktion »CFileWriteJpeg::write()« Probleme:

```
void CFileWriteJpeg::write (const CImageByte &img,
                            int color_space)
     throw (CException)
{
  ...
  JSAMPROW                    row_pointer[1];
  int                         y;
  ...
  for (y = 0; y < img.getH(); y++)
  {
    row_pointer[0] = const_cast<unsigned char*> (img.getRow (y));
    jpeg_write_scanlines (&cinfo, row_pointer, 1);
  }
  ...
}
```

Das Bild, das abgespeichert werden soll, ist in der Bildverarbeitungsbibliothek korrekterweise als konstant definiert worden. Die Funktion »getRow()« liefert bei einem konstanten Bild einen Zeiger auf

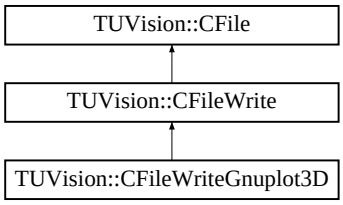


Abbildung 3.6: Vererbungsdiagramm von »CFileWriteGnuplot3D«

konstante Daten zurück; siehe Abschnitt 3.5. Jetzt hat man jedoch ein Problem, da »row_pointer« nicht einen Zeiger auf konstante Daten darstellt. Dieses Problem kann nur durch die Verwendung von »const_cast<>()« gelöst werden. Dieser Cast entfernt die Konstantheit. Es muß jedoch angemerkt werden, daß dieses eine sehr unsaubere Lösung ist.

3.7.7 Gnuplot-Format

Neben den Klassen für das PBM-Format ist dieses die einzige andere I/O-Klasse, die keine externe Bibliothek benötigt. Mit der Klasse »CFileWriteGnuplot3D« kann ein Bild in einer ASCII-Datei so abgelegt werden, daß die Datei mit dem Befehl »splot« des Programms »gnuplot« visualisiert werden kann. »splot« zeichnet hierbei eine 3D-Oberfläche der Bilddaten. Dieses kann interessant sein, um z.B. das Ergebnis von Algorithmen wie der Hough-Transformation zu visualisieren.

Da sich nur Bilder mit einem Farbkanal vernünftig darstellen lassen, unterstützt diese Klasse nur Graustufenbilder.

```
class CFileWriteGnuplot3D: public CFileWrite
{
  public:
    CFileWriteGnuplot3D (void);
    ~CFileWriteGnuplot3D();
    void open (const std::string &filename)
         throw (CException);
    void close (void)
         throw();
    void write (const CImageByteGray &img)
         throw (CException);

  private:
    std::ofstream  hd;         // file I/O class
};
```

3.8 Filter

Die Klassenbibliothek für die Bildverarbeitung enthält eine API für Filter und die Implementation einiger Algorithmen als Filter.

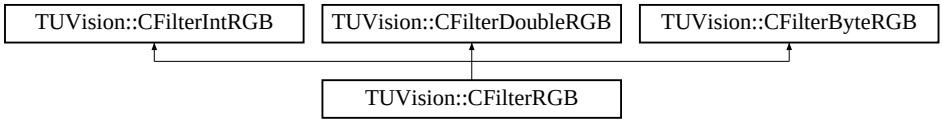


Abbildung 3.7: Vererbungsdiagramm von »CFilterRGB«

3.8.1 Basisklassen

Ein Filter wird von der Klassenbibliothek definiert als eine Funktion bzw. eine Klasse, der man ein Bild übergeben kann, die dann Modifikationen an dem Bild vornimmt und ein Bild mit dem gleichen Zahlenformat, Farbmodell und Abmessungen zurückgibt.

Die API eines Filters wird durch zwei Schichten von abstrakten Basisklassen definiert. In der ersten Schicht befindet sich für jede Kombination aus Zahlenformat und Farbmodell, die von der Bibliothek unterstützt wird, eine eigene Basisklasse der folgenden Form:

```
class CFilterByteRGB
{
  public:
    virtual ~CFilterByteRGB() {};
    virtual void filter (CImageByteRGB &img) =0;
};
```

Die Basisklassen der ersten Schicht bestehen also aus einem virtuellen Destruktor und der rein virtuellen Funktion »filter()«, der als Referenz das zu filternde Bild übergeben wird.

»filter()« ist dabei die Funktion, die die eigentliche Filterung ausführt und das Ergebnis wieder in dem übergebenen Bild-Objekt speichert. Nach der Filterung ist das Originalbild verloren. Wird es nach dem Aufruf des Filters noch benötigt, muß vorher eine Kopie angelegt werden.

Soll jetzt z.B. ein Filter für 8 Bit RGB-Bilder implementiert werden, leitet man einfach die zu implementierende Filterklasse von obiger Basisklasse der ersten Schicht ab. Oftmals soll eine Filterklasse nicht nur einen Bildtyp unterstützen; in diesem Fall wird einfach eine Mehrfachvererbung benutzt. Das bedeutet, die Klasse erbt alle Funktionen der Klassen, deren Erbe sie ist. Da die Filter Routine immer den Namen »filter()« hat, ist diese Funktion dann mehrfach überladen.

Der Benutzer ruft bei jedem Filter einfach die »filter()«-Funktion auf. Der Compiler wählt dann zur Übersetzungszeit die passende »filter()«-Funktion aus. Falls es keine »filter()«-Funktion für den Typ des übergebenen Bildes gibt, gibt der Compiler eine Fehlermeldung aus und meldet, welche Typen von dem jeweiligen Filter zur Zeit unterstützt werden.

In der Regel sollte eine Filterklasse immer alle Zahlenformate eines Farbraums unterstützten. Da die Bibliothek zur Zeit drei Zahlenformate unterstützt, nämlich Byte, Integer und Double, müssen die meisten Implementationen von Filtern von wenigstens drei Basisklassen der ersten Schicht abgeleitet werden. Um sich hier die unnötige Tipparbeit zu sparen und sicherzustellen, daß wirklich von allen Filtern alle Zahlenformate unterstützt werden, die zur Zeit in der Bibliothek implementiert sind, existiert eine zweite Schicht von Basisklassen. Diese faßt jeweils alle Filter eines Farbmodells zusammen. Die Basisklasse der zweiten Schicht für RGB-Bilder sieht z.B. so aus:

```
class CFilterRGB : public CFilterByteRGB,
                   public CFilterIntRGB,
                   public CFilterDoubleRGB
{
```

```
};
```

Die anderen Basisklassen der zweiten Schicht sehen entsprechend aus. In Abbildung 3.7 ist das Vererbungsdiagramm der Klasse »CFilterRGB« zu sehen.

Wenn es gilt, einen neuen Filter zu implementieren, wird man in der Regel zuerst mit einem konkreten Zahlenformat und Farbmodell beginnen und somit die eigene Klasse von einer Basisklasse der ersten Schicht ableiten. Sobald die Implementierung sich dann als Funktionsfähig erwiesen hat, sollte der Filter von einer Basisklasse der zweiten Schicht abgeleitet werden, wobei dann alle anderen Zahlenformate entsprechend zu implementieren sind. Einzige Ausnahme von dieser Regel sind Filter für s/w-Bilder, da es für diese keine Filter-API der zweiten Schicht gibt.

3.8.2 Sobel

Als erster Filter für die Klassenbibliothek wurde der Sobel-Filter implementiert, bei dem es sich um einen Kantenfilter handelt [Lea92, Pra91, GW92, Jäh97]. Ein Kantenbild wird z.B. als Grundlage für die Hough-Transformation benötigt.

Der Sobel-Filter besteht aus zwei 3x3 Matrizen:

$$M_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \;;\; M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Mit diesen beiden Matrizen wird jeder Pixel I_{alt} des Originalbildes gefaltet. Hieraus können dann die gefilterten Pixel I_{neu} berechnet werden:

$$I_{neu} = \sqrt{(M_x * I_{alt})^2 + (M_y * I_{alt})^2}$$

Die Berechnung hat jedoch den Nachteil, durch die Notwendigkeit, die Wurzel zu berechnen, relativ langsam zu sein. Die Implementation des Filters in der Bibliothek verwendet deswegen die folgende Näherung, die häufig in der Literatur vorgeschlagen wird:

$$I_{neu} = |M_x * I_{alt}| + |M_y * I_{alt}|$$

Bei der Implementation muß jetzt allerdings noch der beschränkte Wertebereich von Variablen bedacht werden. Wenn der größte zulässige Wert B_{max} ist, so kann das Ergebnis der Faltung im schlimmsten Fall

$$\begin{aligned} I_{neu} &= \left| \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * \begin{bmatrix} B_{max} & 0 & 0 \\ B_{max} & I_{alt} & 0 \\ B_{max} & 0 & 0 \end{bmatrix} \right| \\ &\quad + \left| \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & I_{alt} & 0 \\ B_{max} & B_{max} & B_{max} \end{bmatrix} \right| \\ &= 8\, B_{max} \end{aligned}$$

sein. Das bedeutet, daß wir das Ergebnis der Faltung immer durch acht teilen müssen.

Der Sobel-Filter wurde für Graustufen Bilder implementiert:

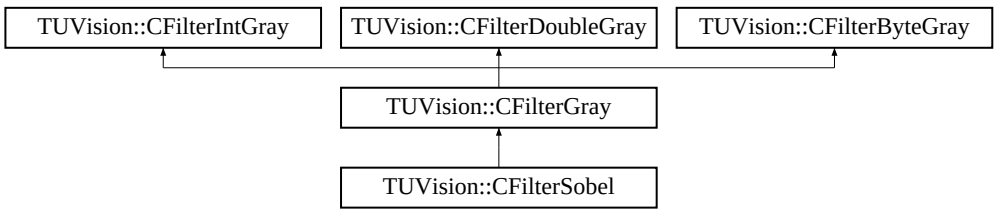


Abbildung 3.8: Vererbungsdiagramm von »CFilterSobel«

```
class CFilterSobel : public CFilterGray
{
  public:
    void filter (CImageByteGray &img);
    void filter (CImageIntGray &img);
    void filter (CImageDoubleGray &img);
};
```

Wie im Vererbungsdiagramm zu erkennen ist, das in Abbildung 3.8 dargestellt ist, wurde der Sobel-Filter von der Klasse »CFilterGray« abgeleitet.

Die Faltungsoperation wurde nicht mit Hilfe des ebenfalls implementierten allgemeinen Faltungsfilters, der im Abschnitt 3.8.3 beschrieben wird, implementiert. Dieser würde bei einer Faltung mit einer 3x3 Matrix pro Pixel neun Multiplikationen und acht Additionen durchführen. Bei einer direkten Implementation des Sobel-Filters kann die Form der beiden Matrizen ausgenutzt werden. Die Faltung mit der M_x Matrix läßt sich z.B. so formulieren:

$$\begin{aligned} M_x * I_{alt}(x,y) &= I_{alt}(x-1,y-1) - I_{alt}(x+1,y-1) \\ &\quad + I_{alt}(x-1,y+1) - I_{alt}(x+1,y+1) \\ &\quad + 2\ (I_{alt}(x-1,y) - I_{alt}(x+1,y)) \end{aligned}$$

Es werden dann nur fünf Additionen/Substraktionen und eine Multiplikation benötigt. Außerdem kann einiger zusätzlicher Code entfallen, der bei einer allgemeinen Formulierung benötigt würde.

3.8.3 Faltungsfilter

Für die Bildverarbeitungsbibliothek wurde ein allgemeiner Faltungsfilter [Pra91] implementiert. Dieser Filter berechnet den Wert jedes Pixels I_{neu} gemäß folgender Formel:

$$I_{neu} = I_{alt} * M$$

Hierbei ist M eine m x n Matrix:

$$M = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}$$

Problematisch bei allen Faltungsoperationen ist immer der Randbereich, da dort nicht alle Werte zur Verfügung stehen, die eigentlich für die Berechnung notwendig wären. Es gibt verschiedene Methoden,

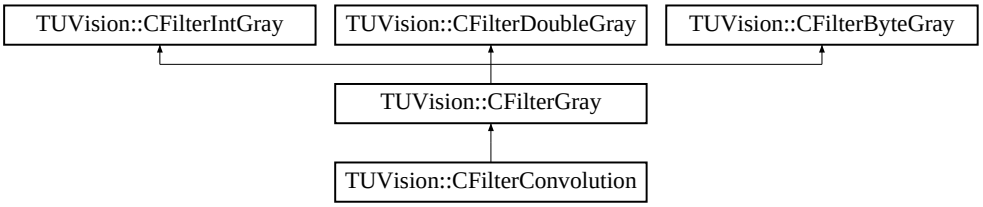


Abbildung 3.9: Vererbungsdiagramm von »CFilterConvolution«

diesem Problem zu begegnen. Die Implementation benutzt die Methode »centered, zero boundary«, bei der links und rechts ein $\frac{n}{2}$ und oben und unten ein $\frac{m}{2}$ Bereich im gefilterten Bild auf »0« gesetzt wird. Dieses ist genau der Bereich, für den nicht genug Werte vorliegen.

Der Faltungsfilter »CFilterConvolution« wurde für alle Graustufenbilder implementiert:

```
class CFilterConvolution : public CFilterGray
{
  public:
    CFilterConvolution();
    ~CFilterConvolution();
    void setParameters (int x, int y, const char *matrix_c,
                        char scaling_c);
    void setParameters (int x, int y, const float *matrix_f,
                        float scaling_f);
    void filter (CImageByteGray &img);
    void filter (CImageIntGray &img);
    void filter (CImageDoubleGray &img);

  private:
    char  *d_matrix_c;        // convolution matrix (char)
    float *d_matrix_f;        // convolution matrix (float)
    char  d_scaling_c;        // scaling factor (char)
    float d_scaling_f;        // scaling factor (float)
    bool  d_matrix_float;     // Use the float conv. matrix?
    int   d_width, d_height;  // dimensions of the conv. matrix
};
```

Mit der Funktion »setParameters()« wird als erstes immer die Matrix gesetzt, mit der das Bild gefiltert werden soll. Diese Funktion ist zweifach überladen: die Matrix kann einmal als 8 Bit-Zahlen und einmal als Fließkommazahlen übergeben werden. Diese Unterscheidung ist sinnvoll, da man mit Fließkommazahlen nicht so schnell rechnen kann, diese aber trotzdem teilweise benötigt. Das Zahlenformat der Matrix und das des Bildes sind unabhängig. Die übergebene Matrix wird in »d_matrix_c« bzw. »d_matrix_f« abgespeichert. Die boolsche Variable »d_matrix_float« legt fest, welche von beiden benutzt werden soll.

Außerdem kann der Funktion »setParameters()« ein Skalierungsfaktor übergeben werden. Neben dieser Skalierung führt der Filter auch ein Clipping durch.

Die eigentliche Filterung findet in der Funktion »filter()« statt.

Ein für beliebige Koeffizienten und Dimensionen implementierter Filter ist natürlich immer deutlich langsamer wie ein bestimmter Filter, der direkt implementiert worden ist. Ein deutlicher Geschwindigkeitsgewinn würde sich allerdings durch den Einsatz von Prozessoren erzielen lassen, die »Single Instruction Multiple Data« (SIMD) unterstützen. Während SIMD vor einiger Zeit nur bei digitalen Si-

gnalprozessoren zu finden war, werden immer mehr normale CPUs mit entsprechenden Befehlen ausgestattet. Die im PC verwendeten CPUs bieten SIMD-Befehle z.B. in Form der Technologien MMX, SSE und 3Dnow. Je nach Genauigkeit können z.B. mit MMX bis zu acht Ganzzahlen gleichzeitig multipliziert und danach addiert werden [Int96]. Trotz eines Verwaltungsoverheads kann durch MMX z.B. bei einer Multiplikation von acht Werten und einer drauffolgenden Summation ein Geschwindigkeitsgewinn von Faktor drei erzielt werden.

Die verwendete Hochsprache C++ unterstützt keine SIMD-Befehle, so daß SIMD-Operationen in Assembler implementiert werden müssen. Das führt allerdings dazu, daß die Programme dann nicht mehr portabel sind. Man müßte also neben einigen SIMD-Varianten des Filters auch eine direkt in C++ implementierte Variante für Systeme anbieten, deren SIMD-Befehle nicht unterstützt werden.

Die Unterstützung von SIMD wurde nicht implementiert, da dieser Filter für diese Diplomarbeit nicht benötigt wurde. Sollte der Filter jedoch später in anderen Projekten häufiger Verwendung finden, wäre eine SIMD-Unterstützung sicherlich sehr sinnvoll.

Eine weitere eventuell interessante Erweiterung dieser Klasse könnte darin bestehen, den Skalierungsfaktor durch Summation aller positiven Koeffizienten der Faltungsmatrix automatisch zu berechnen.

3.8.4 Histogramm

Sobald man Bilder im Rechner verarbeiten möchte, die mit einem Gerät eingelesen wurden, das ein CCD-Element benutzt, ist man vor das Problem gestellt, daß Weiß nicht mehr Weiß und Schwarz nicht mehr Schwarz ist. Außerdem ist meistens das ganze Bild im Vergleich zur Vorlage viel zu dunkel.

Jedes bessere Bildverarbeitungsprogramm enthält deshalb heute Routinen für die Schwarz- und Weißpunktkorrektur und eine Gamma-Korrektur. Die Bildverarbeitungsbibliothek enthält für diese Aufgabe den Filter »CFilterHistogram«, dessen Funktionsweise in der Abbildung 3.10 zu erkennen ist.

Die eigentliche Filterung besteht aus drei Schritten: dem Streching, der Gamma-Korrektur und der Compression.

Beim Streching werden im Histogramm des Bildes die beiden Helligkeiten definiert, die als Schwarz- und Weißpunkte verwendet werden sollen. In dem Bild wird z.B. »a« als Schwarzpunkt und »b« als Weißpunkt definiert. Der Filter zieht das Histogramm so auseinander, daß »a« den Helligkeitswert »0« und »b« den Helligkeitswert »B_{max}« zugewiesen bekommt.

Die Formel für das Streching läßt sich leicht herleiten. Zuerst wird das Histogramm um »a« nach rechts verschoben:

$$x_{st} = x_{orig} - a$$

Damit liegt der Schwarzpunkt wie gewünscht beim Wert »0«. Jetzt muß das Histogramm nur noch mit einem passenden Faktor multipliziert werden, so daß der Weißpunkt bei B_{max} liegt:

$$x_{st} = \frac{B_{max} \cdot (x_{orig} - a)}{b - a}$$

Hierbei ist B_{max} wieder der größte mögliche Wert des verwendeten Variablentyps.

Die Gamma-Korrektur ist definiert als:

$$x_\gamma = B_{max} \frac{x_{st}^{\frac{1}{\gamma}}}{B_{max}^{\frac{1}{\gamma}}}$$

Als dritte Operation führt der Filter eine Compression des Histogramms durch. Hiermit wird das Histogramm von x_γ, das sich von »0« bis »B_{max} erstreckt, auf den kleineren Wertebereich von »c« bis »d«

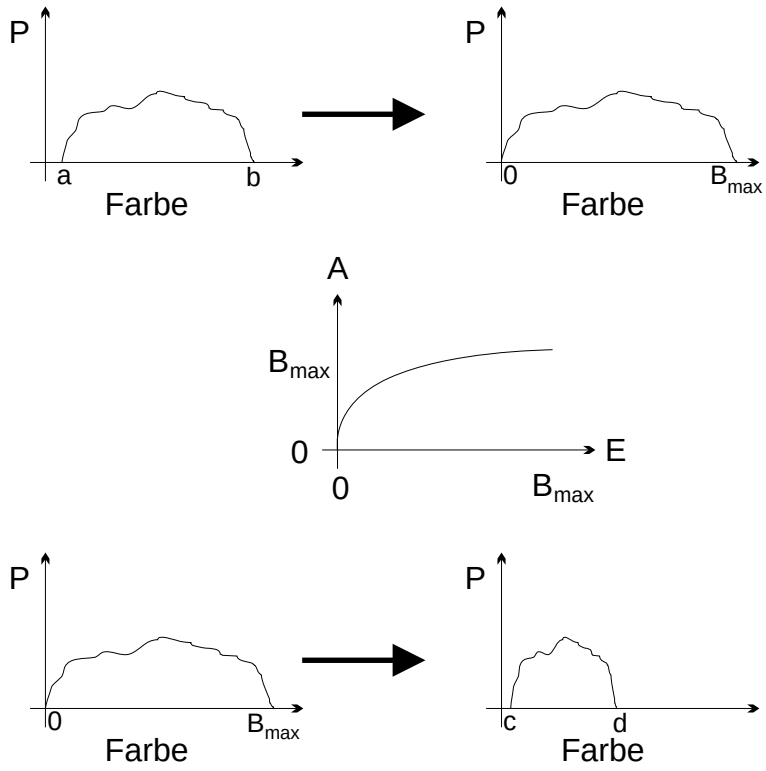


Abbildung 3.10: Funktionsweise des Histogramm Filters »CFilterHistogram«: (a) Streching, (b) Gamma-Korrektur, (c) Compression

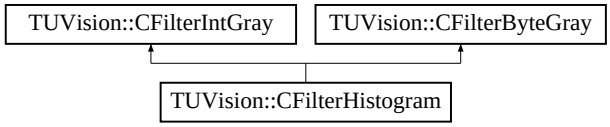


Abbildung 3.11: Vererbungsdiagramm von »CFilterHistogram«

abgebildet. Es kann hiermit also dafür gesorgt werden, daß das erzeugte Bild keine reinen Schwarz- und/oder Weißfarbtöne enthält.

Für die Compression muß zuerst einmal das Histogramm, das sich über den Wertebereich »0« bis »B_{max}« erstreckt, auf den Bereich »c« bis »d« verkleinert werden:

$$x_{cp} = \frac{x_\gamma \cdot (d - c)}{B_{max}}$$

Anschließend muß das Histogramm noch um »c« nach rechts verschoben werden, so daß die entgültige Formel für die Compression so aussieht:

$$x_{cp} = \frac{x_\gamma \cdot (d - c)}{B_{max}} + c$$

Wie im Vererbungsdiagramm der Klasse »CFilterHistogram«, das in Abbildung 3.11 dargestellt ist, zu erkennen ist, ist dieser Filter für Graustufenbilder mit den Zahlenformaten Byte und Integer implementiert worden:

```
class CFilterHistogram : public CFilterByteGray,
                         public CFilterIntGray
{
  public:
    CFilterHistogram();
    void setStreching (int black, int white);
    void setGammaCorrection (double gamma);
    void setCompression (int black, int white);
    void reset (void);
    void filter (CImageByteGray &img);
    void filter (CImageIntGray &img);

  private:
    int    d_strech_black;      // streching: black point
    int    d_strech_white;      // streching: white point
    bool   d_streching;         // Use streching?
    double d_gamma;             // gamma correction value
    bool   d_gamma_correction;  // Use gamma correction?
    int    d_comp_black;        // compression: min
    int    d_comp_white;        // compression: max
    bool   d_compression;       // Use compression?
};
```

Mit den drei »set«-Funktionen können die Parameter für das Streching, die Gamma-Korrektur und die Compression gesetzt werden. Die Funktionen legen dabei einfach die Werte im jeweiligen Objekt ab und setzen eine boolsche Variable, die angibt, daß für den jeweiligen Schritt Parameter vom Benutzer gesetzt worden sind. Die Filterroutinen führen nur die Schritte durch, deren boolsche Variablen gesetzt sind. Auf diese Weise ist es möglich, z.B. nur die Gamma-Korrektur des Filters zu nutzen. Mit der »reset()«-Funktion können alle Parameter in einem Objekt gelöscht werden.

Die beiden Filterfunktionen bauen beim Aufruf jeweils als erstes eine Lookup-Tabelle gemäß den oben hergeleiteten Formeln und den gesetzten Variablen des Objektes auf. Diese Lookup-Tabellen werden dann für die Filterung des übergebenen Bildes verwendet. Dank der Lookup-Tabelle ist die Filterung auch bei sehr großen Bildern noch ausreichend schnell.

3.8.5 Thinning

Bei der Erzeugung von Kantenbildern mit Filtern wie dem Sobel-Filter kommt es häufig vor, daß die Kanten mehrere Pixel breit sind, was teilweise unerwünscht sein kann. In solchen Fällen kann ein Thinning Filter [GW92] hilfreich sein, der mehrere Pixel breite Objekte in einem s/w-Bild in Objekte umrechnet, die nur noch einen Pixel breit sind. Ist z.B. eine Linien vorher fünf Pixel breit, so ist sie nach der Filterung nur noch einen Pixel breit.

Der Filter arbeitet in zwei Schritten, wobei der zu berechnende Punkt p_1 und die Nachbarpixel p_2 bis p_9 seien:

$$\begin{bmatrix} p_9 & p_2 & p_3 \\ p_8 & p_1 & p_4 \\ p_7 & p_6 & p_5 \end{bmatrix}$$

Für jeden weißen Punkt im Originalbild wird als erster Schritt folgende Berechnung durchgeführt:

$$2 \leq N(p_1) \leq 6$$

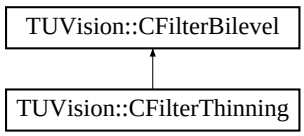


Abbildung 3.12: Vererbungsdiagramm von »CFilterThinning«

$$S(p_1) = 1$$
$$p_2 \cdot p_4 \cdot p_6 = 0$$
$$p_4 \cdot p_6 \cdot p_8 = 0$$

Hierbei ist $N(p_1)$ die Summe aller weißen Nachbarpixel, wobei davon ausgegangen wird, daß ein weißer Pixel durch einen Wert von eins definiert ist:

$$N(p_1) = \sum_{x=p_2}^{p_9} x$$

$S(p_1)$ ist die Anzahl von 0-1 Transitionen der Nachbarpixel, wenn diese von p_2 nach p_9 durchlaufen werden.

Weiße Punkte, die die Bedingungen des ersten Schrittes erfüllen, werden gelöscht. Der zweite Schritt sieht genauso wie der erste aus, allerdings lauten die dritte und vierte Bedingung jetzt:

$$p_2 \cdot p_4 \cdot p_8 = 0$$
$$p_2 \cdot p_6 \cdot p_8 = 0$$

Da der Thinning Filter nur für s/w-Bilder definiert ist, sieht die Schnittstelle der Filterklasse sehr kompakt aus:

```
class CFilterThinning : public CFilterBilevel
{
  public:
    void filter (CImageByteBilevel &img);
};
```

Das Vererbungsdiagramm der Klasse ist in Abbildung 3.12 zu sehen.

3.9 Operatoren

Operatoren liefern im Gegensatz zu Filtern nicht ein Bild mit dem gleichen Zahlenformat und Farbmodell als Ergebnis zurück. Vielmehr kann das Ergebnis eines Operators mehr oder weniger beliebig sein.

3.9.1 Converter

Die Aufgabe der Klasse »COperatorConverter« besteht darin, ein Bild von einem Zahlenformat in ein anderes zu konvertieren. Dieses kann sinnvoll sein, wenn z.B. eine I/O-Klasse das Zahlenformat des

Bildes, das abgespeichert werden soll, nicht direkt unterstützt. Auch in Verbindung mit der Berechnung eines Histogramms kann diese Klasse hilfreich sein, da sich Histogramme z.B. nicht für Bilder im Double-Format berechnen lassen.

Es wurde bewußt die Design-Entscheidung getroffen, daß diese Klasse nur das Zahlenformat nicht jedoch aber das Farbmodell eines Bildes ändert.

Da die Klasse wie alle anderen Klasse typsicher realisiert wurde, ist die API recht umfangreich:

```
class COperatorConverter
{
  public:
    void convert (const CImageByteGray &imgsrc,
                  CImageIntGray **imgdst);
    ...

  private:
    void convertByteToInt (const CImageByte &imgsrc,
                           CImageInt &imgdst,
                           int nchannels);
    void convertByteToDouble (const CImageByte &imgsrc,
                              CImageDouble &imgdst,
                              int nchannels);
    ...
};
```

Dieses ist nur ein kleiner Ausschnitt aus dem kompletten Interface. Die eigentliche Konvertierung findet in den privaten Funktionen. Es gibt für jede Kombination von zwei Zahlenformaten eine eigene Funktion. Da zur Zeit drei Zahlenformate unterstützt werden, existieren insgesamt sechs Konvertierungsfunktionen.

Jeder Funktion werden zwei Bild-Objekte übergeben, wobei das erste das Originalbild enthält. Im zweiten Objekt wird das konvertierte Bild abgelegt. Als dritter Parameter wird den Funktionen die Farbkanalanzahl übergeben. Diese wird benötigt, um zusammen mit den Informationen über die Dimensionen zu ermitteln, wieviele Werte konvertiert werden müssen. Die Konvertierungsfunktionen teilen bzw. multiplizieren einfach alle Werte eines Bildes passend.

Das typsichere öffentliche Interface der Klasse stellt die Funktion »convert()« zur Verfügung. Diese ist mehrfach überladen. Für jede Kombination der »CImage«-Klassen existiert eine eigene Funktion. Diese Funktionen sind hierbei mehr oder weniger reine Wrapper-Funktionen. Ihre einzige Aufgabe ist es, ein passendes Bild-Objekt für das konvertierte Objekt anzulegen und dieses zusammen mit der Farbkanalanzahl an die privaten Konvertierungsfunktionen zu übergeben.

Bei der Implementation dieser Klasse hat sich gezeigt, daß es entgegen der am Anfang getroffenen Design-Entscheidung doch sehr hilfreich wäre, wenn die Klasse »CImage« eine Funktion »getNumberChannels()« anbieten würde, mit der die Anzahl der Farbkanäle eines Bild ermittelt werden könnte. Dieses würde bei der Klasse »COperatorConverter« ermöglichen, auf die Wrapper-Funktion zu verzichten und die privaten Funktionen direkt als öffentliche Schnittstelle zu verwenden. Es ist von daher zu überlegen, ob man diese Funktion nicht zukünftig doch anbieten sollte, auch wenn die Gefahr besteht, daß unerfahrene Programmierer dadurch die Möglichkeit erhälten, Klassen zu implementieren, die nicht mehr typsicher sind.

3.9.2 Multiplexer

Viele Filter und Operatoren sind nur für Graustufenbilder definiert und implementiert. Möchte man diese auf farbige Bilder anwenden, hat man ein Problem. Eine Möglichkeit, dieses Problem zu lösen,

würde darin bestehen, die Filter und Operatoren entsprechend zu erweitern, so daß sie auch diese Farbmodelle unterstützen. Das bedeutet jedoch einen ziemlich Zeitaufwand und ist daher in der Regel nicht sinnvoll.

Die zweite Lösung besteht darin, daß Farbbild einfach in mehrere Graustufenbilder zu zerlegen. Auf diese können dann die vorhandenen Filter und Operatoren angewendet werden. Am Ende können die Graustufenbilder dann wieder zu einem Farbbild zusammengefügt werden.

In der Bildverarbeitungsbibliothek ist die zweite Methode in Form eines Multiplexers und Demultiplexers realisiert worden. Da auch die Klasse »COperatorMultiplex« eine sehr umfangreiche API besitzt, wird hier nur ein Ausschnitt wiedergegeben:

```
class COperatorMultiplex
{
  public:
    void demultiplex (const CImageByteRGB &src,
                      std::vector <CImageByteGray*> &dst) const;
    ...
    void multiplex (const std::vector <CImageByteGray*> &src,
                    CImageByteRGB **dst) const;
    ...

  protected:
    CImageByteGray *getChannel (const CImageByte &src,
                                int nchannels,
                                int channel) const;
    ...
    void setChannel (const CImageByteGray &src, CImageByte &dst,
                     int nchannels, int channel) const;
    ...
    void demultiplexGeneric (const CImageByte &src,
                             std::vector <CImageByteGray*> &dst,
                             int nchannels) const;
    ...
    void multiplexGeneric (const std::vector <CImageByteGray*> &src,
                           CImageByte &dst) const;
    ...
    void testDimensions (const std::vector <CImage*> &src,
                         int nchannels) const;
};
```

Alle Funktionen sind mehrfach überladen. Die hier gezeigte öffentliche Funktion »demultiplex()« nimmt ein 8 Bit RGB-Bild entgegen und erzeugt daraus drei 8 Bit Graustufenbilder, die als »vector<>« zurückgegeben werden. Hierbei ist »demultiplex()« nur eine Wrapper-Funktion, die zur Sicherstellung der Typsicherheit verwendet wird. Intern ruft »demultiplex« einfach »demultiplexGeneric()« auf. Diese Funktion erwartet als Parameter die Anzahl der Farbkanäle, die das Bild-Objekt enthält.

Die entgegengesetzte Richtung wird durch die Funktion »multiplex()« angeboten, der ein Vektor von Graustufenbildern übergeben wird. Auch diese Funktion ist nur ein Wrapper und zwar um die Funktion »multiplexGeneric()«. Das Bild-Objekt, in dem »multiplexGeneric()« das erzeugte Bild ablegt, kann nicht von »multiplexGeneric()« erzeugt werden, da dieser Funktion zwar bekannt ist, wieviele Farbkanäle das Zielbild enthalten wird, der genaue Typ des Farbmodells hier allerdings unbekannt ist. Aus diesem Grund ist die Wrapper-Funktion für die Erzeugung des Bild-Objektes zuständig, da hier das genaue Farbmodell und damit der Typ des zu erzeugenden Bild-Objektes bekannt ist.

Die beiden Funktion »demultiplexGeneric()« und »multiplexGeneric()« rufen in einer Schleife einfach

die Funktionen »getChannel()« bzw. »setChannel()« entsprechend der Farbkanalanzahl oft auf. Für jedes Zahlenformat existiert eine überladene Version dieser beiden Funktionen.

Mit »getChannel()« kann ein bestimmter Farbkanal eines Bildes in einem Graustufenbild abgelegt werden. Mit »setChannel()« kann ein solches Graustufenbild in einem bestimmten Farbkanal eines Farbbildes abgelegt werden.

Diese sehr komplexe Klasse könnte sicherlich durch die interne Verwendung von Templates noch deutlich vereinfacht werden. Auch für diese Klasse wäre eine Funktion hilfreich, mit der die Anzahl der Farbkanäle eines Bildes ermittelt werden könnte.

3.9.3 Threshold

Für einige Berechnungen wie z.B. der Hough-Transformation wird ein s/w-Bild benötigt, wobei das zu verwendende Bild nur als Graustufenbild vorliegt. Mit der Klasse »COperatorThreshold« kann ein Graustufenbild in ein s/w-Bild konvertiert werden:

```
class COperatorThreshold
{
  public:
    void threshold (const CImageByteGray &src,
                    CImageByteBilevel **tar,
                    unsigned char level);
    void threshold (const CImageIntGray &src,
                    CImageByteBilevel **tar,
                    int level);
    void threshold (const CImageDoubleGray &src,
                    CImageByteBilevel **tar,
                    double level);
};
```

Alle drei Versionen der Funktion »threshold()« erwarten als dritten Parameter den Schwellwert, der für die Entscheidung, ob es sich um einen schwarzen oder weißen Pixel handelt, herangezogen wird. Für die Berechnung gilt also folgende Formel:

$$I_{neu} = \begin{cases} 1 & : \quad I_{alt} > level \\ 0 & : \quad sonst \end{cases}$$

3.9.4 Histogramm

Zur Unterstützung des Histogramm Filters, der in Abschnitt 3.8.4 beschrieben wurde, ist ein Histogramm Operator implementiert worden. Dieser Operator hat zwei Aufgaben.

Zum einen kann mit ihm das Histogramm eines Bildes berechnet werden. Ein Histogramm gibt wieder, wie häufig ein bestimmter Helligkeitswert in einem Bild vorkommt. In Abbildung 4.2 sind einige Histogramme zu sehen. Ein Histogramm hilft bei der Beurteilung, ob z.B. ein Bild zu dunkel oder zu hell ist. In diesen Fällen ist die Häufigkeit von hellen bzw. dunklen Pixeln klein im Vergleich zu den dunklen bzw. hellen Pixeln.

Die zweite Aufgabe dieses Operators besteht darin, die Schwarz- und Weißpunkte eines Bildes zu ermitteln. Diese Werte können dann benutzt werden, um das Histogramm mit dem Histogramm Filter zu normalisieren.

```
class COperatorHistogram
```

```
{
  public:
    COperatorHistogram();
    ~COperatorHistogram();
    void setImage (const CImageByteGray &img);
    void setImage (const CImageIntGray &img);
    const int *getHistogramData (void) const;
    int getHistogramSize (void) const;
    int getBlackpoint (int threshold) const;
    int getWhitepoint (int threshold) const;

  private:
    int *d_histogram_data;  // histogram data
    int d_histogram_size;   // number of histogram data elements
};
```

Die Klasse »COperatorHistogram« unterstützt Byte und Integer Graustufenbilder. Das Zahlenformat Double wird nicht unterstützt, da sich ein Histogramm nur für Zahlenformate mit einer diskreten Werteverteilung berechnen läßt.

Die eigentliche Berechnung des Histogramms findet in einer der beiden Funktionen mit dem Namen »setImage()« statt. Das Histogramm wird hierbei in der Variablen »d_histogram_data« abgelegt. Da diese als privat definiert ist, kann der Benutzer nur über die Funktion »getHistogramData()« auf die Variable zugreifen. »getHistogramData()« liefert einen Zeiger auf konstante Daten zurück, so daß der Benutzer das Histogramm in dem Objekt nur lesen nicht aber verändern kann.

Mit den Funktionen »getBlackpoint()« und »getWhitepoint()« können die Schwarz- und Weißpunkte berechnet werden. Beiden Funktionen muß hierfür ein Grenzwert als Parameter übergeben werden. Die Funktion durchläuft dann bei der Schwarzpunktsuche das Histogramm von links nach rechts und summiert die Häufigkeiten der Helligkeitswerte solange auf, bis die Summe den übergebenen Grenzwert überschreitet. Dieser Helligkeitswert entspricht dann dem Schwarzpunkt. Die Weißpunktsuche läuft genauso ab, allerdings wird hier von rechts nach links gesucht.

Setzt man z.B. als Grenzwert jeweils den Wert »1«, so findet man genau den Wertebereich, den das Bild benutzt. Benutzt man die so gefundenen Werte für den Histogramm Filter, so nutzt das Bild nach der Filterung den gesamten Wertebereich. Dieses ist insbesondere bei Bildern sinnvoll, die von Scannern oder Kameras stammen, da bei diesen häufig nicht der gesamte Wertebereich genutzt wird, was dazu führt, daß es im Bild weder echtes Schwarz noch echtes Weiß gibt.

3.9.5 Draw

Mit den von »COperatorDraw« abgeleiteten Klassen kann in Bild-Objekten gezeichnet werden. Es stehen Funktionen zum Zeichnen von Linien, Kreisen und Ellipsen bereicht, wobei die letzten beiden jeweils gefüllt und nicht gefüllt gezeichnet werden können. Außerdem besteht die Möglichkeit, einen Text im Bild auszugeben und ein Fadenkreuz zu zeichnen.

```
class COperatorDraw
{
  public:
    virtual void drawLine (int x_0, int y_0, int x_d, int y_d);
    virtual void drawCircle (int x_0, int y_0, int r,
                             bool filled = false);
    virtual void drawEllipse (int x_0, int y_0, int a, int b,
                              bool filled = false);
    virtual void drawText (int x, int y, const string &text,
```

```
                              int scale = 1);
      virtual void drawCrosshair (int x, int y, int length,
                                  int scale, int distance);

    protected:
      virtual void setPixel (int x, int y) =0;
      virtual const CImage *getImage (void) const =0;
};
```

Die Funktion »drawLine()« zeichnet eine Linie. Zuerst wurde versucht, dieses einfach anhand der Gradengleichung

$$y = m \cdot x + b$$

zu realisieren. Bei bestimmten Steigungen m war die so gezeichnete Linie allerdings aufgrund der begrenzten Auflösung nicht zusammenhängend. Bei z.B. $m = 2$ wäre zwischen jedem Punkt der Linie eine Lücke von einem Pixel. Ein weiterer Nachteil dieses Verfahrens ist die Tatsache, daß die ganze Berechnung mit Fließkommazahlen stattfindet, was relativ langsam ist. Es wurden deshalb statt dessen der sogenannte Bresenham Algorithmus zum Zeichen von Linien verwendet [Fla, LH98].

Kreise lassen sich mit der Funktion »drawCircle()« und Ellipsen mit der Funktion »drawEllipse()« zeichnen. Auch hier hat man das Problem, daß der Linienzug nicht zusammenhängend ist, wenn man einfach die Formeln

$$x^2 + y^2 = r^2$$

bzw.

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1$$

implementiert. Um dieses Problem zu lösen, kann man jetzt wieder auf Algorithmen von Bresenham zurückgreifen [Milb, Mila]. Alternativ kann man aber z.B. für Ellipsen auch so vorgehen, daß man zuerst die obige Formel nach y auflöst:

$$y(x) = b\sqrt{1 - \left(\frac{x}{a}\right)^2}$$

Jetzt durchläuft man diese Berechnung für alle $0 \leq x \leq a$ und berechnet zusätzlich noch den Abstand Δy:

$$y(x+1) = b\sqrt{1 - \left(\frac{x+1}{a}\right)^2}$$

$$\Delta y = y(x+1) - y(x) - 1$$

Für jeden x-Wert zeichnet man jetzt eine senkrechte Linie, die bei (x, y) beginnt und bei $(x, y + \Delta y)$ endet.

Die Funktion »drawText()« gibt einen Text im Bild aus. Dazu wird ein Bitmap Font benutzt, der mittels eines kleinen selbstgeschriebenen Tools aus einer Fontdatei erzeugt wurde, die jeder Linux Distribution als Zeichensatz für die Console beiliegt. Jeder Buchstabe dieses Zeichensatzes ist 9x16 Pixel groß.

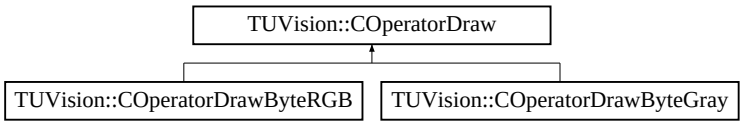


Abbildung 3.13: Vererbungsdiagramm von »COperatorDraw«

Dieses kann bei großen Bilder viel zu klein sein, so daß man die Buchstaben im Bild garnicht erkennen kann. Deswegen läßt sich der Text im Bild mit einem Skalierungsfaktor vergrößern. Hierbei findet keine Interpolation statt, sondern die Pixel werden einfach mehrfach ausgegeben. Dieses erzeugt zwar keine optisch zufriedenstellende Lösung, ist aber für technische Anwendungen völlig ausreichend.

Zukünftig könnte man diesen Font auch durch eine externe Bibliothek ersetzen, die einen Truetype Render enthält, so daß man ganz normale Vektorfonts verwenden kann, die sich auch frei skalieren lassen.

Alle Zeichenfunktionen benutzen die beiden als »protected« definierten Funktionen »setPixel()« und »getImage()«. Beide sind in »COperatorDraw« rein virtuell. Dieses ist notwendig, da ihre Implementation vom Typ des verwendeten Bildes abhängt.

Von »COperatorDraw« sind dann jeweils Zeichenklassen für die jeweiligen Bildtypen abgeleitet. Wie in Abbildung 3.13 zu sehen ist, sind bisher Zeichenklassen für Bilder vom Typ »CImageByteGray« und »CImageByteRGB« definiert worden. Die Definition der Klasse »COperatorDrawByteGray« sieht z.B. so aus:

```
class COperatorDrawByteGray : public COperatorDraw
{
  public:
    COperatorDrawByteGray();
    void setColor (unsigned char color);
    void setImage (CImageByteGray *img);

  private:
    void setPixel (int x, int y);
    const CImage *getImage (void) const;

  private:
    CImageByteGray *d_img;   // image to be used
    unsigned char  d_color;  // draw color to be used
};
```

Die Klasse hat zwei öffentliche Funktionen. Mit »setColor()« wird die Farbe gesetzt, mit der »setPixel()« einen Punkt zeichnet. Je nach Bildtyp, für den die jeweiligen Zeichenklasse gedacht ist, sehen die Parameter von »setColor()« anders aus. Für ein RGB-Bild müssen z.B. drei Werte übergeben werden.

Damit die eigentlichen Zeichenfunktionen in der Basisklasse »COperatorDraw« nicht für jeden Bildtyp implementiert werden müssen, exisitiert die virtuelle Funktion »setPixel()«. Diese erwartet lediglich als Parameter die beiden Koordinaten. Wie diese intern einen Pixel setzt, ist für die Basisklase uninteressant.

Mit »setImage()« wird das Bild festgelegt, in dem gezeichnet werden soll.

3.10 Geometry

Zum Bereich der »Geometry« werden in der Bildverarbeitungsbibliothek die Klassen gezählt, die numerisch ein geometrisches Objekt wie z.B. einen Punkt, eine Linie, einen Kreis usw. wiederspiegeln.

Im Rahmen der Diplomarbeit wurde die Klasse »CPoint« implementiert, die einfach einen Punkt wiederspiegelt. Eine solche Klasse für Punkte wird sehr häufig benötigt, um z.B. die Position von Objekten, die man in einem Bild gesucht hat, zurückzuliefern. In der Diplomarbeit wurde diese Klasse z.B. benutzt, um die Mittelpunkte der Spundlöcher und der Fässer an den Benutzer zu übergeben.

In ANSI C hätte man hierfür einfach eine Struktur wie die folgende definiert:

```
struct
{
  int x;
  int y;
} t_point;
```

In C++ macht man es sich etwas komplizierter und definiert gleich eine Klasse mit den entsprechenden »get«-, »set«- und Vergleichsfunktionen:

```
class CPoint
{
  public:
    CPoint();
    CPoint (int x, int y);
    CPoint (const CPoint &pnt);
    void setPoint (int x, int y);
    void setX (int x);
    void setY (int y);
    inline int getX (void) const;
    inline int getY (void) const;
    CPoint &operator= (const CPoint &rhs);
    bool operator== (const CPoint &pt) const;
    bool operator!= (const CPoint &pt) const;

  private:
    int d_x;  // x-coordinate
    int d_y;  // y-coordinate
};
```

Die Verwendung einer Klasse bietet viele Vorteile. So sind die eigentlichen Daten jetzt privat und können nur über entsprechende Funktionen gelesen und geschrieben werden. Da die Schreibfunktionen nicht als konstant definiert sind, lassen sich die Daten auch wirklich nur dann ändern, wenn das Objekt nicht konstant ist.

Sehr praktisch sind natürlich auch die Vergleichs- und Zuweisungoperationen. Man spart sich viel unnötige Tipparbeit und die Operationen sind alle typsicher.

Sehr interessant wird die Klasse in Verbindung mit der STL. So kann z.B. eine Funktion mit

```
...
void foo (vector <CPoint> &points)
{
 ... berechne einige Punkte ...
```

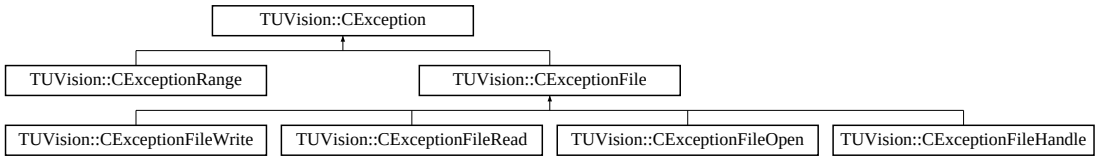


Abbildung 3.14: Vererbungsdiagramm von »CException«

```
}
...
```

eine beliebige Anzahl von Punkten an die aufrufende Funktion zurückgeben. Hierbei brauchen sich beide Funktionen nicht mit irgendwelchen Zeigern rumschlagen. Die notwendige Verwaltung der Punkte übernimmt das »vector<>«-Template.

3.11 Exceptions

Einige Klassen der Bildverarbeitungsbibliothek benutzen das in Abschnitt 3.3.3 beschriebene C++-Konzept der Exceptions, um Fehler an den Benutzer zu melden.

Alle von der Klassenbibliothek geworfene Exceptions basieren auf der Basisklasse »CException«:

```
class CException
{
  public:
    virtual ~CException() {};
    virtual const std::string &what (void) const;

  protected:
    std::string d_what;  // description of the problem
};
```

Von ihr werden alle Klassen abgeleitet, die als Exceptions geworfen werden sollen; siehe Abbildung 3.14. Sie besteht aus lediglich einer öffentlichen Funktion und einer als »protected« definierten Variablen. Die Funktion »what()« liefert einen String zurück, der die Ursache für das Werfen einer Exception beschreibt. Dieser String kann zur Information des Benutzers dienen:

```
try
{
  ...
  mache etwas, was eine Exception ausloesen koennte
  ...
}
catch (CException &ex)
{
  cerr << ex.what() << endl;
}
```

Wie man in diesem Beispiel sehen kann, wird mit »catch()« eine Ausnahme vom Typ »CException« gefangen. Die Referenz »ex« verweist auf das Objekt vom Typ »CException«, das von der Funktion, die

einen Fehler melden möchte, geworfen wurde. Die Funktion »what()« liefert jetzt die im Objekt »ex« gespeicherte Fehlerbeschreibung als String zurück, die dann mit der C++-Funktion »cerr« als Fehler auf dem Bildschirm ausgegeben werden kann.

Der String, der von »what()« zurückgeliefert wird, stammt aus der Variablen »d_what«. Nun fragt man sich natürlich, wie diese Variable überhaupt gesetzt wird. Dieses ist direkt von außen ja nicht möglich, da sie als »protected« definiert ist. Das Setzen dieser Variablen ist die Aufgabe der von »CException« abgeleiteten Fehlerklassen.

Die Fehlerklasse, die benutzt wird, um Fehler zu melden, die beim Lesen von Dateien aufgetreten sind, sieht z.B. so aus:

```
class CExceptionFile : public CException
{
};


class CExceptionFileWrite : public CExceptionFile
{
  public:
    CExceptionFileWrite (const std::string &filename,
                         const std::string &add_infos = "");
};
```

Sie bestehen nur aus einem Konstruktor. Dieser erzeugt aus einigen übergebenen Informationen wie dem Namen der Datei, bei der es Probleme gab, einen passenden »d_what« String. Die anderen Fehlerklassen sehen entsprechend aus.

Man könnte sich nun die Frage stellen, warum man überhaupt so viele verschiedene Klassen für Fehler erstellt. Dieses hat den Grund, daß man bei »catch()« den Typ der Fehlerklassen festlegen kann, die man fangen möchte. Es ist z.B. möglich, an einer Stelle nur auf »CExceptionFile«-Fehler zu reagieren, indem man nur diese mit »catch()« fängt. Es ist auch möglich, an einer Stelle verschiedene Fehler unterschiedlich zu behandeln:

```
try
{
  ...
  mache etwas, was eine Exception ausloesen koennte
  ...
}
catch (CExceptionFile &ex)
{
  cerr << ex.what() << endl;
}
catch (CException &ex)
{
  cout << ex.what() << endl;
}
```

In diesem Beispiel werden z.B. die »CExceptionFile«-Fehler mit »cerr« und alle restlichen Fehler mit »cout« ausgegeben.

3.12 Entwicklungstools

Für die Entwicklung der Bildverarbeitungsbibliothek wurden im Fachbereich erstmalig zwei neue Entwicklungstools eingesetzt: Doxygen und CVS.

Doxygen unterstützt den Entwickler bei der Dokumentierung seiner Sourcen. Der Entwickler versieht hierfür während der Entwicklung jede Funktion und jede Klasse mit einer genauen Beschreibung. Bei Funktionen wird außerdem mit einer speziellen Syntax die Aufgabe jedes Parameters dokumentiert. Aus diesen Kommentare erzeugt Doxygen dann automatisch eine Dokumentation in Form von HTML-Seiten bzw. einer druckbaren PostScript-Datei. Hierbei wertet Doxygen nicht nur die Kommentare sondern die Sourcen selbst aus und erstellt z.B. Vererbungsdiagramme der Klassen. Sämtliche in dieser Zusammenschrift enthaltenen Vererbungsdiagramme wurden von Doxygen automatisch erstellt.

CVS steht für »Concurrent Versions System« und wird für die Verwaltung des Source Codes eingesetzt. Statt die Sourcen einfach auf einer Festplatte abzulegen, wurden sie im CVS-System gespeichert. Jeder, der an der Entwicklung der Bibliothek beteiligt war, konnte jederzeit aus dem CVS-System den aktuellsten Source Code auschecken. Hat ein Entwickler etwas am Source Code geändert, so checkt er diese Änderungen in das CVS-System ein. Dieses speichert die Änderungen und merkt sich außerdem, wer sie wann eingecheckt hat. Außerdem ist es möglich, die Neuerungen mit einem Text zu kommentieren.

Dank des CVS-Systems sind also immer automatisch alle Interessierten auf dem gleichen Stand. Dieses ist ohne die Verwendung eines solchen Systems nur sehr schwer möglich. Treten bei der Nutzung des Source Codes plötzlich neue Probleme auf, können sich die Entwickler über das CVS-System informieren, was zuletzt von wem an dem Source Code verändert wurde.

Beide Programme haben sich während der Entwicklung als sehr hilfreich erwiesen. Mit Doxygen ist es mit relativ wenig Aufwand möglich, Sourcen so zu dokumentieren, daß sie für andere Entwickler nutzbar sind. Dieses ist insbesondere natürlich bei Bibliotheken wichtig, die wie die Bildverarbeitungsbibliothek später von anderen Entwicklern genutzt werden sollen. Gegen das Konzept von Doxygen, die Dokumentation und die Implementation eines Programmes in den selben Source Dateien zu speichern, spricht eigentlich nur, daß die Sourcen schnell unübersichtlich werden, wenn die Dokumentation sehr umfangreich ausfällt.

CVS war für dieses Projekt, an dem mehrere Entwickler mitgearbeitet haben, sehr wichtig. Es kam häufig vor, daß das eigene Programm nicht mehr lief, weil ein anderer Entwickler etwas an der Bildverarbeitungsbibliothek geändert hatte. Dank des CVS-Systems war es aber sehr leicht, festzustellen, was sich geändert hatte und was somit die Ursache für die Probleme war.

3.13 Debian

Ein Ziel der Diplomarbeit war es, die Wartung der Abfüllanlage über eine IP-Verbindung zu ermöglichen. Zur Wartung zählt neben der Änderung der Konfiguration der Anlage auch die Möglichkeit, Updates der verwendeten Programme einspielen zu können.

Die verwendete Debian GNU/Linux Distribution verfügt über ein Paketsystem mit dem Namen »dpkg«. Mit diesem Programm können Binärpakete installierst und rückstandslos deinstalliert werden. Ein ideales System also, um die Problemstellung zu lösen.

Die Pakete verwenden das Format »deb«. Ein solches Paket wird mit speziellen Tools erstellt, die der Distribution beiliegen. Es wurden deshalb Steuerdateien erstellt, mit denen sich die übersetzte Bildverarbeitungsbibliothek in ein solches Paket verpacken läßt.

Kapitel 4

Bildverarbeitung

4.1 Einleitung

Ziel der Bildverarbeitung für den Befüllroboter der Firma Feige war, wie bereits in Abschnitt 1.1 kurz angerissen wurde, die automatische Erkennung der Spundlöcher von Fässern, die auf einer Palette durch die Maschine laufen. Die erkannten Spundlöcher sollten dann vom PC an die SPS der Anlage übermittelt werden, die dann mittels dieser Koordinaten die Fässer völlig selbständig befüllen kann.

Die Idee, wie eine solche Erkennung durchzuführen sei, basiert auf der Diplomarbeit von Klaus Wiehler [Wie96]. In dieser hat er sich mit der Detektion der Iris eines Auges unter einem Operationsmikroskop beschäftigt. Wie bei der Iris müssen bei den Spundlöchern kreis- bzw. ellipsenförmige Objekte erkannt werden. Es bot sich also an, auf den Ergebnissen dieser Diplomarbeit aufzubauen.

Ursprünglich war geplant, einfach die für die obige Diplomarbeit implementierten Algorithmen wiederzuverwenden. Dieses scheiterte jedoch aus verschiedenen Gründen, so daß letztendlich beschlossen wurde, die Algorithmen aufbauend auf der entwickelten Bildverarbeitungsbibliothek neu zu implementieren; siehe Kapitel 3.

Die Bildverarbeitung zur Erkennung der Iris besteht im wesentlichen aus zwei Schritten. Als erstes wird mit einem Filter ein Kantenbild erstellt. Auf dieses wird dann eine Hough-Transformation angewendet.

4.2 Aufnahme von Beispielbilder

Relativ am Anfang der Entwicklung wurden mit der in der Anlage enthaltenen Zeilenkamera Musterbilder von Paletten mit Fässern aufgenommen, so daß die entwickelten Programme direkt mit diesen Musterbilder getestet werden konnten.

Um die Bilder überhaupt aufnehmen zu können, wurde ein Weg benötigt, die Kamera mit dem PC zu verbinden. Hierzu wurde ein PCI-Framegrabber vom Hersteller der Kamera benutzt, der zusammen mit dem passenden Linux-Treiber für dieses Projekt beschafft worden war.

Der Treiber für den Framegrabber besteht aus dem Treiber selbst und einer Laufzeitbibliothek, über die eine Anwendung mit dem Treiber kommunizieren kann. Da die API der Laufzeitbibliothek ANSI C verwendete und die Bildtypen der entwickelten Bildverarbeitungsbibliothek nicht kannte, wurde als Verbindungsglied die Wrapper Klasse »CFramegrabberSK« entwickelt. Diese kapselt die ANSI C Funktionen des Framegrabbers und liefert Bilder in Form von Objekten zurück, wie sie in der Bildverarbeitungsbibliothek definiert sind.

Bei der Implementation der Wrapper Klasse kam es zu mehreren Problemen. Die ursprünglich ausgelieferte Version der Treiber-Bibliothek ließ sich mit keinem C- oder C++-Programm linken, da die Ent-

wickler dieser Bibliothek eine ihrer Funktionen »time()« genannt hatten. Einen solchen Funktionsnamen definiert allerdings bereits die Standard Bibliothek von C und C++, so daß es zu einem Namenskonflikt kam. Dieses Problem wurde durch den Hersteller schnell behoben.

Wesentlich problematischer war allerdings, daß beim Aufnehmen eines Bildes mit dem Framegrabber teilweise garnichts passierte und teilweise der ganze Rechner abstürzte. Während der Hersteller die Abstürze bis zum Ende der Diplomarbeit beseitigen konnte, blieb das Problem, daß die Bildaufnahme manchmal garnicht funktionierte, bestehen. Verursacht wird dieses Problem vermutlich dadurch, daß die Befehle der Treiber-Bibliothek in einer ganz bestimmten Reihenfolge aufgerufen werden müssen. Diese Reihenfolge ist allerdings im Handbuch des Treibers nicht dokumentiert. Auch ansonsten war das Handbuch bei der Entwicklung keine große Hilfe.

Die Wrapper Klasse betreibt den Treiber des Framegrabbers im sogenannten »offline Flächenscan« Modus. In diesem Modus muß die Zeile der Kamera nicht in einer Schleife manuell immer wieder ausgelesen werden. Vielmehr wird dem Treiber mitgeteilt, wieviele Zeilen aufgenommen werden sollen. Der Treiber liefert dann nicht eine Zeile sondern ein komplettes zweidimensionales Bild zurück.

Der Zeitpunkt, zu dem jeweils eine Zeile aufgenommen werden soll, wird durch die Synchronisation festgelegt. Diese kann per Software oder per Hardware erfolgen. Wie bereits in Abschnitt 2.5.2 beschrieben wurde, wurde entschieden, einen der Inkrementalgeber als Quelle für die externe Synchronisation des Framegrabbers zu benutzen.

Mit Hilfe der entwickelten Bildverarbeitungsbibliothek, der Klasse »CFramegrabberSK« und einem Testprogramm wurden in der Anlage Bilder von verschiedenen Faßtypen aufgenommen. Pro Faßtyp wurden fünf Bilder mit verschiedenen Belichtungszeiten aufgenommen. Die Belichtungszeit der Zeilenkamera wird durch die Parameter Counter Stop und Pixelfrequenz des Framegrabbers gesteuert, wobei sich die Belichtungszeit so berechnet [Kir00]:

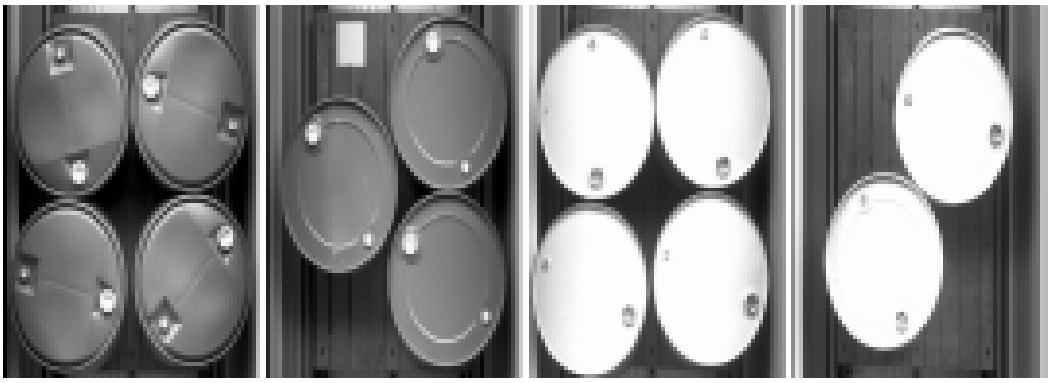
$$t_{Belichtung} = \frac{64 \cdot cs}{Pixelfrequenz}$$

Die Pixelfrequenz betrug immer 5 MHz. Für Counter Stop wurden die Werte 50, 100, 150, 200 und 250 verwendet. Einige Bilder von Fässer und Kanister, die mit einer Belichtungszeit von 1,92 ms aufgenommen wurden, sind in Abbildung 4.1 zu sehen. Alle Bilder wurden mit einer Gamma-Korrektur von 2,0 bearbeitet, da sie normalerweise wesentlich dunkler sind und man deshalb die Details im Ausdruck hätte schlecht erkennen können.

Bei der Aufnahme traten neben den Stabilitätsproblemen mit dem Treiber des Framegrabbers zwei weitere Probleme auf. So fiel nach einigen Aufnahmen auf, daß die Paletten im Bild nicht rechtwinklig abgebildet wurden, sondern zu einem Parallelogramm verzerrt waren. Als Ursache stellte sich die falsch montierte Zeilenkamera heraus. Diese war leicht verdreht eingebaut worden, so daß ihre CCD-Zeile nicht exakt rechtwinklig zur Bewegungsrichtung der Palette ausgerichtet war. Dieses führt dazu, daß einige Pixel der CCD-Zeile die Palette eher erfassen wie die restlichen Pixel.

Auch das zweite Problem wurde in Form von Verzerrungen deutlich. Lief der Tragkettenförderer der Anlage mit der schnellst möglichen Geschwindigkeit und wurde eine längere Belichtungszeit von z.B. 3,2 ms verwendet, wurde die Palette im Bild in Bewegungsrichtung gestaucht wiedergegeben. Verursacht wird dieses Problem, wie bereits erwähnt wurde, dadurch, daß der Framegrabber mit der Aufnahme von Zeilen nicht nachkommt. Es kommt von Inkremtalgeber also bereits das Signal, eine neue Zeile aufzunehmen, während die vorherige Zeile noch garnicht vollständig aufgenommen wurde. Es gehen also Zeilen verloren. Dieses merkt der Nutzer des Framegrabbers allerdings nicht, da die »Areascan« Routine des Framegrabbers immer ein Bild mit der gewünschten Zeilenanzahl zurückliefert. Im Fall von Zeilenverlusten ist die Palette im Bild allerdings gestaucht und die zusätzlichen Zeilen zeigen den leeren Tragkettenförderer.

Wie man sich leicht vorstellen kann, wird dieses Problem durch eine im Verhältnis zur Belichtungszeit zu hohe Geschwindigkeit des Tragkettenförderers verursacht. Die maximale Geschwindigkeit, bei der noch keine Zeilenverluste auftreten, läßt sich so berechnen:

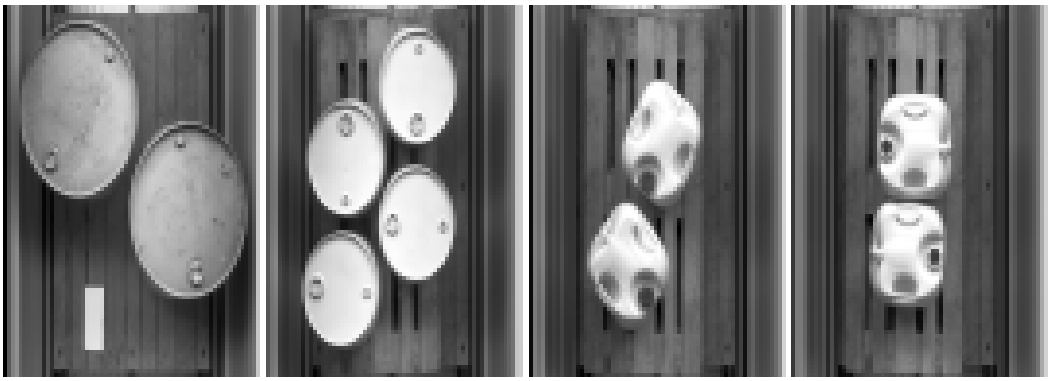


(a) Kunststoffässer, blau

(b) Metallfässer, dunkelblau lackiert

(c) Metallfässer, gelb/blau lackiert

(d) Metallfässer, grau lackiert

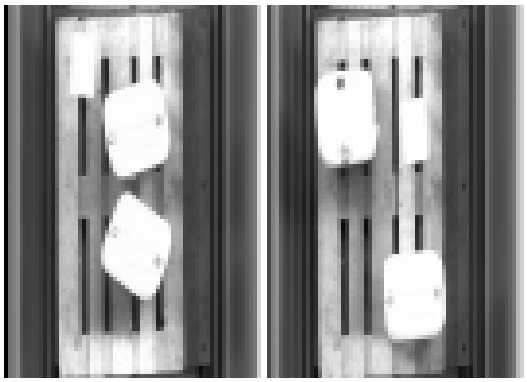


(e) Metallfässer, hellblau lackiert, verrostet

(f) Metallfässer (klein), orange lackiert

(g) Kunststoff-kanister, hellblau, mit Deckel

(h) Kunststoff-kanister, hellblau, ohne Deckel



(i) Kunststoffkanister, weiß, ohne Deckel

(j) Kunststoffkanister, weiß, ohne Deckel

Abbildung 4.1: Bilder von Fässern und Kanistern, die mit der Kamera des Befüllroboters aufgenommen wurden

$$v_{max} = \frac{s}{t} = \frac{0,43mm}{2 \cdot t_{Belichtung}}$$

Die 0,43 mm entsprechen der Strecke, die der Tragkettenförderer zurücklegt, bis der Inkrementalgeber einen Impuls erzeugt. Da bei der externen Triggerung nicht sofort eine Zeile aufgenommen werden kann, sondern gewartet werden muß, bis die Kamera zur Aufnahme bereit ist, wurde die Belichtungszeit nach oben mit dem Faktor zwei abgeschätzt. Die maximale Geschwindigkeit und die Zeit $t_{Palette}$, die benötigt wird, um eine 1200 mm x 1200 mm große Palette aufzunehmen, ist in der nachfolgenden Tabelle für verschiedene Belichtungszeiten abzulesen:

$t_{Belichtung}$	v_{max}	$t_{Palette}$
0,64 ms	0,34 m/s	3,57 s
1,28 ms	0,17 m/s	7,14 s
1,92 ms	0,11 m/s	10,72 s
2,56 ms	0,08 m/s	14,29 s
3,20 ms	0,07 m/s	17,86 s

Soll eine möglichst hohe Geschwindigkeit der Bilderfassung erreicht werden, wird also eine sehr gute Beleuchtung der Fässer benötigt, so daß mit kurzen Belichtungszeiten gearbeitet werden kann.

4.3 Auswertung der Beispielbilder

4.3.1 Histogramme

Bei der Auswertung der Bilder, die in der Anlage aufgenommen worden waren, fiel auf, daß die meisten Bilder so dunkel waren, daß man mit dem bloßen Auge kaum die Fässer erkennen konnte. Einige mit langen Belichtungszeiten aufgenommene Bilder waren zwar heller, zeigten aber kaum Kontraste.

Um zu untersuchen, welche Belichtungszeit am besten für einen bestimmten Faßtyp geeignet ist, wurden für einige Bilder, die mit verschiedenen Belichtungszeiten aufgenommen worden waren, Histogramme erstellt. In Abbildung 4.2 sind die Histogramme für die blauen Kunststoffässer, die bereits in Abbildung 4.1 (a) vorgestellt wurden, zu sehen.

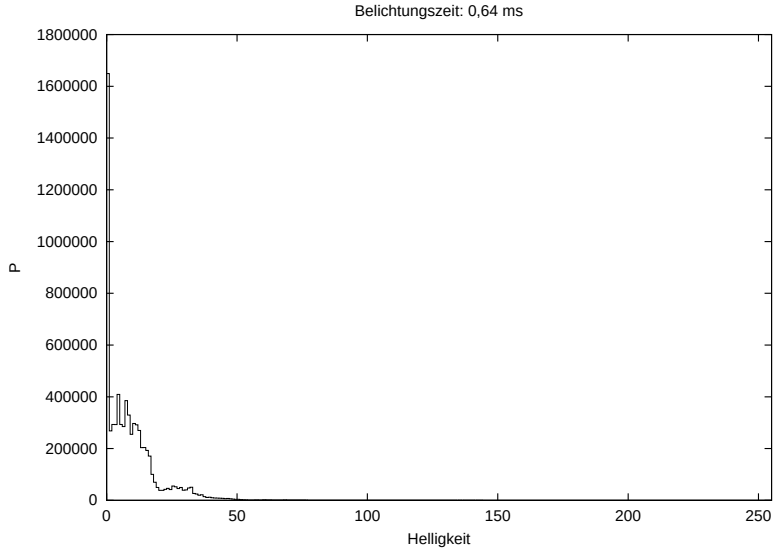
Es läßt sich in den Histogrammen erkennen, daß bei einer kurzen Belichtungszeit wie 0,64 ms nur sehr dunkle Helligkeitswerte im Histogramm vorhanden sind. Dieses ist die Ursache dafür, daß man mit dem Auge in diesem Bild fast nichts erkennen kann. Bei einer Erhöhung der Belichtungszeit verbessert sich die Verteilung, so daß wesentlich mehr Helligkeitswerte verwendet werden. Bei der Belichtungszeit von 3,20 ms läuft die Kamera anscheinend allerdings schon in einen Sättigungsbereich, wie an dem Peak beim hellsten Wert des Histogramms zu sehen ist.

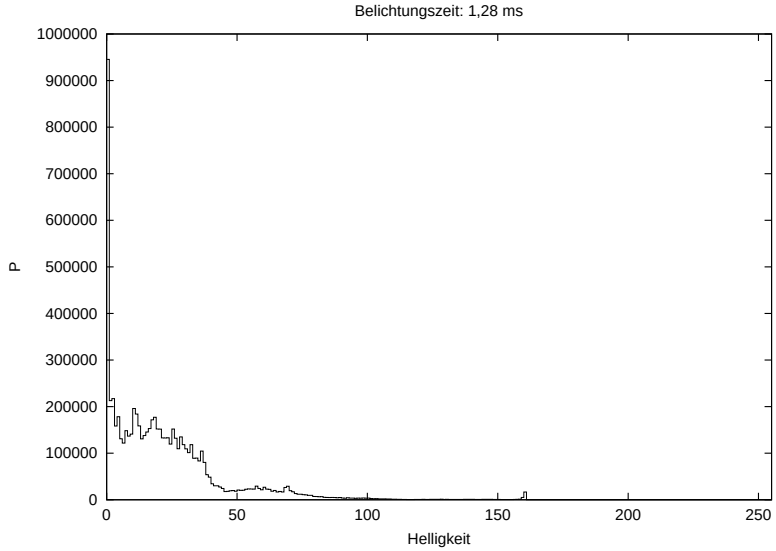
Wie man sehr deutlich erkennen kann, nutzt keines der fünf Bilder den gesamten Wertebereich aus. Alle Histogramme enden ungefähr bei einem Helligkeitswert von 160. In keinem Bild kommt also die Farbe Weiß vor.

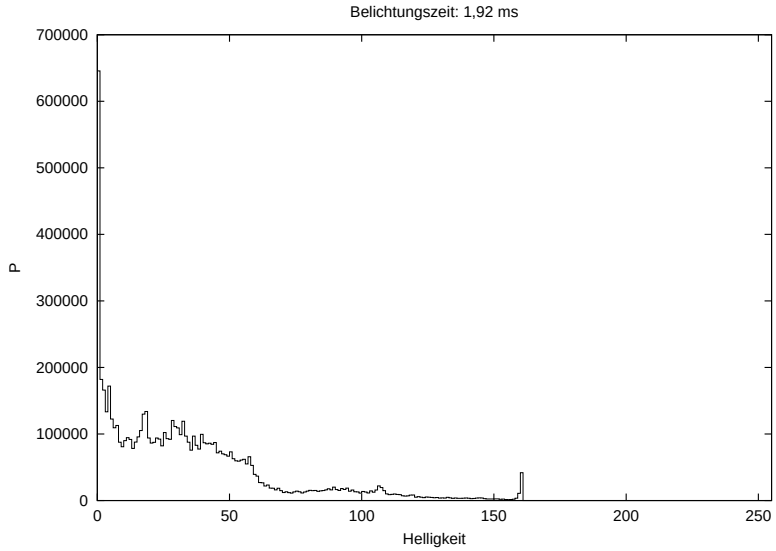
Dieses Problem läßt sich teilweise durch den Einsatz einer Weißpunkt- und Gamma-Korrektur beheben.

4.3.2 Helligkeitsabfall zum Rand

Ebenfalls auffällig bei den meisten Bilder war die Tatsache, daß die Bilder zum Rand hin dunkler wurden. Am Rand waren Fässer und Hintergrund deshalb kaum zu unterscheiden. Um dieses genauer zu untersuchen, wurde ein Bild der blauen Kunststoffässer, welches mit einer Belichtungszeit von 2,56 ms aufgenommen wurde, in vier gleichbreite Streifen aufgeteilt. Für jeweils einen Streifen vom Rand und einen von der Mitte wurden Histogramme berechnet; diese sind in Abbildung 4.3 zu finden.


Belichtungszeit: 0,64 ms
P
Helligkeit
0
200000
400000
600000
800000
1000000
1200000
1400000
1600000
1800000
0
50
100
150
200
250


Belichtungszeit: 1,28 ms
P
Helligkeit
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000
0
50
100
150
200
250


Belichtungszeit: 1,92 ms
P
Helligkeit
0
100000
200000
300000
400000
500000
600000
700000
0
50
100
150
200
250

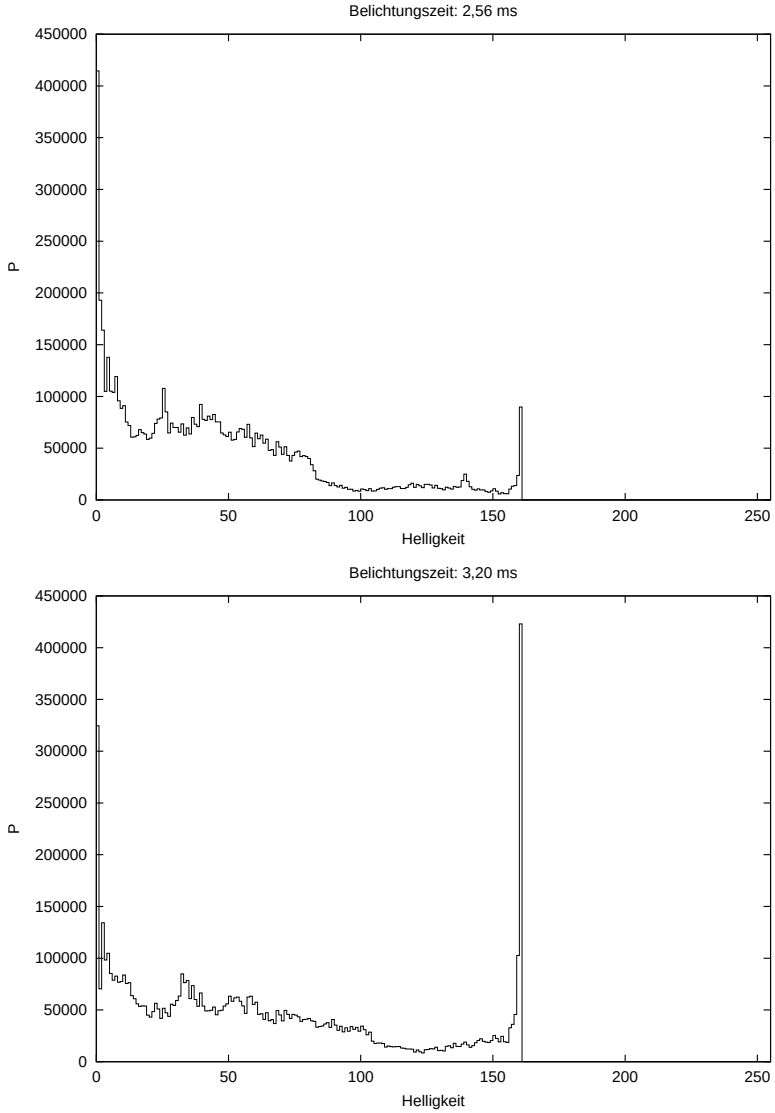


Abbildung 4.2: Histogramm der blauen Fässer aus Kunststoff für verschiedene Belichtungszeiten

Wie man relativ deutlich sehen kann, sind bei dem Histogramm des Randstreifens im Vergleich zu dem Histogramm der Bildmitte wesentlich mehr Pixel in dem Bereich der dunklen Helligkeitswerte zu finden. Bestätigt wird diese Beobachtung auch durch den Mittelwert, der für den Randstreifen 44,0 und für den mittleren Streifen 59,9 beträgt.

Ein solcher Helligkeitsabfall würde die Spundlocherkennung natürlich erschweren, so daß versucht wurde, dieses Problem zu lösen. Verursacht wird der Helligkeitsabfall durch die Länge der Lampe. Da in dem Gehäuse, das die Anlage umgibt, sehr wenig Platz ist, ist das Lampengehäuse mit den Hochfrequenz-Neonröhren kaum länger wie die Palettenbreite. Da sich das Licht der Lampe nicht nur senkrecht nach unter sondern natürlich auch zur Seite ausbreitet, wird der Rand nicht so stark ausgeleuchtet wie die Mitte der Palette.

Es gibt im Prinzip zwei Ansätze, um dieses Problem zu lösen. Im Idealfall muß die Lampe so lang sein, daß sie an beiden Enden der Palette deutlich übersteht. Dieses war in der bestehenden Anlage aus Platzgründen nicht möglich, so daß eine andere Problemlösung angewendet wurde. Vor die Lampe wurde eine Blende montiert, die in der Mitte der Lampe deutlich weniger Licht durchläßt wie am Rand.

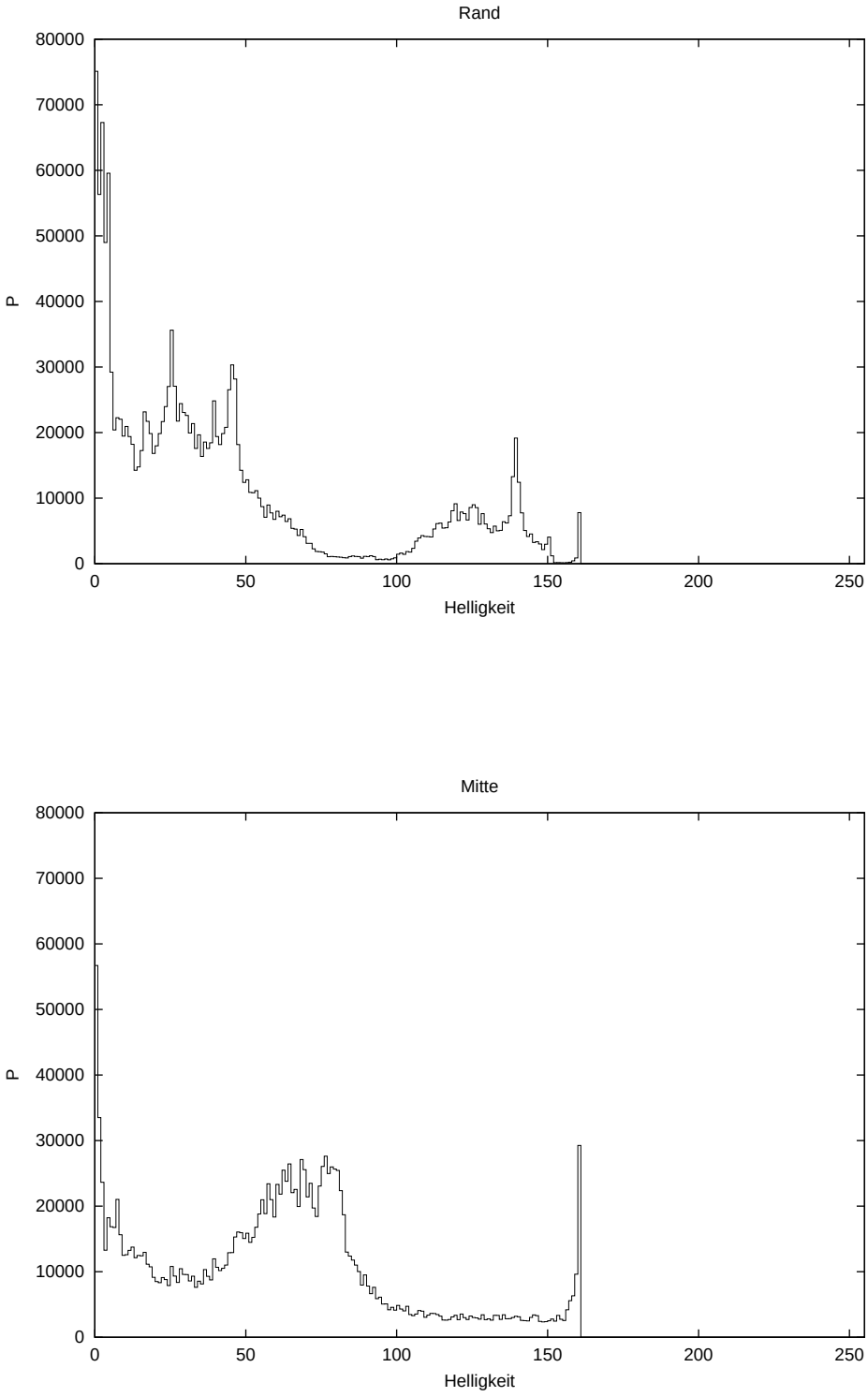


Abbildung 4.3: Helligkeitsabfall zum Rand

Abbildung 4.4: Spundloch mit Schatten

Hiermit wird zwar der Helligkeitsabfall verringt, allerdings reduziert man durch diese Blende die Helligkeit der Ausleuchtung insgesamt. Eine reduzierte Helligkeit bedeutet aber auch, daß länger belichtet werden muß und damit das Aufnehmen einer Palette länger dauert. Insgesamt also keine ideal Lösung.

4.3.3 Schatten

Ein weiteres Problem, das bei den aufgenommenen Bildern deutlich wurde, war die Position und der Winkel der Lampe. In der bestehenden Anlage ist die Lampe parallel zur CCD-Zeile eingebaut und strahlt unter einem Winkel von ungefährt 45 Grad auf die Faßdeckel. Dieser Winkel wurde gewählt, um Reflexionen auf den Metalloberflächen zu vermeiden, die z.B. bei einem beinahe senkrechten Auftreffen des Lichtes auf die Deckel entstehen würden. Das Problem der Reflexion hat man so zwar recht gut im Griff, allerdings entstehen durch den relativ flachen Einfallwinkel lange Schatten, wie man z.B. in Abbildung 4.4 erkennen kann.

Man könnte die Situation durch die Installation einer zweiten Lampe, die von der anderen Seite her beleuchtet, verbessern. Allerdings gäbe es auch dann wieder Probleme mit dem Platz in dem Gehäuse der Maschine.

Eine alternative Methode wäre die Installation einer großen weißen Fläche rund um die Kamera, die dann von unten her beleuchtet wird. Das Licht würde dann an der Fläche reflektiert werden und man hätte ein sehr diffuses Licht.

Aufpassen muß man bei allen Maßnahmen allerdings, daß genügend Kontrast für den Kantenfilter vorhanden ist. Besonders Fässer, bei denen Faßdeckel und Spundlochdeckel die gleiche Farbe habe, können hierbei problematisch sein. Sorgt man hier durch die Beleuchtung dafür, daß keine Schatten mehr vorhanden sind, kann es passieren, daß man Faßdeckel und Spundlochdeckel nicht mehr vernünftig trennen kann.

4.3.4 Blooming

Wurde eine lange Belichtungszeit gewählt, trat teilweise ein Effekt auf, der Blooming genannt. Abbildung 4.5 zeigt z.B. ein Spundloch der blauen Kunststoffässer. Aufgenommen wurde es mit einer Belichtungsdauer von 3,20 ms. Der rechte Rand weist einige Störungen auf, die durch Blooming entstanden sind.

Blooming entsteht dadurch, daß der Potentialtopf eines Pixels der CCD-Zeile überlauft [Mar00]. Befinden sich irgendwann zu viele Ladungen in dem Potentialtopf und treffen dann noch weitere ein, fließt die überschüssige Ladung in die Potentialtöpfe der benachbarten Pixel ab. Hierdurch wird natürlich der Inhalt der Nachbarpixel verfälscht und es treten Störungen im aufgenommenen Bild auf.

Vermeiden läßt sich dieser Effekt durch zwei Methoden. Die eine besteht darin, eine möglichst kurze Belichtungszeit zu wählen, so daß sich nicht zuviele Ladungen in den Potentialtöpfen ansammeln

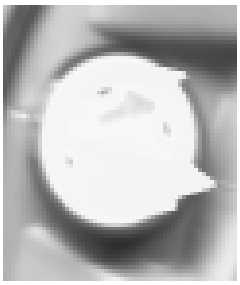


Abbildung 4.5: Spundloch mit Blooming

können. Diese Methode hat allerdings entscheidene Nachteile. Sie schränkt zum einen die möglichen Belichtungszeiten ein. Zum anderen kann es gerade bei Metallflächen sehr schnell zu Glanzlichtern kommen, die dann zum Blooming führen.

Die wesentlich bessere Methode zur Verhinderung von Blooming ist der Einsatz von Kameras, die eine CCD-Zeile benutzen, die über Hardware-Antiblooming verfügt. Bei einem solchen CCD-Chip hat jeder Potentialtopf einen Überlauf, der überflüssige Ladungen abführt, ohne das sie Störungen hervorrufen können. Die in der Anlage eingesetzte Kamera ist bei deren Hersteller für einen kleinen Aufpreis mit Hardware-Antiblooming verfügbar.

4.4 Implementation

Nachdem die Beispielbilder aufgenommen und untersucht worden waren, wurde als nächster Schritt die eigentliche Erkennung der Spundlöcher implementiert. Diese besteht aus zwei Schritten. Zuerst wird mit einem Filter ein Kantenbild des aufgenommenen Bildes erzeugt. Auf dieses wird dann die Hough-Transformation angewendet.

4.4.1 Hough-Transformation

Die Hough-Transformation ermöglicht es, in einem Kantenbild Kurven zu detektieren. Ursprünglich wurde dieses Verfahren von Paul Hough entwickelt, um Linien zu erkennen, die von geladenen Teilchen in einer Blasenkammer erzeugt wurden. Später wurde dieses Verfahren auch für andere Kurven wie z.B. Kreise und Ellipsen verallgemeinert.

Die diskrete Formulierung der allgemeinen Hough-Transformation lautet [Lea92, Wie96]:

$$H(\vec{p}) = \sum_x \sum_y I(x,y) \cdot \delta(C(x;y;\vec{p}))$$

Hierbei ist $I(x,y)$ ein Pixel im binären Kantenbild, in dem die Kurve erkannt werden soll. $C(x;y;\vec{p}) = 0$ beschreibt die zu suchende Kurve.

Die zu suchenden Spundlöcher sind in den aufgenommenen Bildern zu Ellipsen verzerrt. Dieses wird durch der Verhältnis von Inkrementalgeberimpulsen und Anzahl der Pixel der verwendeten Zeilenkamera verursacht. Die allgemeine Gleichung für Ellipsen lautet [BSMM93]:

$$\left(\frac{x-x_0}{a}\right)^2 + \left(\frac{y-y_0}{b}\right)^2 = 1$$

Daraus ergibt sich für die Parametrisierung der Hough-Transformation:

$$C(x;y;\vec{p}) = 1 - \left(\frac{x-x_0}{a}\right)^2 + \left(\frac{y-y_0}{b}\right)^2 = 0$$

Die beiden Parameter a und b definieren die Breite und Höhe der Ellipsen, die gesucht werden sollen. Sie können also für die gesamte Berechnung als konstant angesehen werden. Die Hough-Ebene hängt somit von zwei Parametern ab: x_0 und y_0.

Jeder Punkt in der Hough-Ebene definiert die Wahrscheinlichkeit dafür, daß der Punkt (x_0, y_0) im Kantenbild der Mittelpunkt einer Ellipse von dem gesuchten Typ ist. Die Hough-Transformation durchläuft alle Pixel der Hough-Ebene und überprüft, wieviele Pixel des Kantenbildes die Parametrisierungsfunktion $C(x;y;\vec{p})$ erfüllen.

Die Hough-Transformation wurde auf Basis der entwickelten Bildverarbeitungsbibliothek implementiert:

```
class CHoughCircleEllipse
{
  public:
    void setCircle (int r);
    void setEllipse (int a, int b);
    void search (const CImageByteBilevel &src, CImageIntGray **tar);

  private:
    vector<int> d_figure;  // the figure to be searched
};
```

Die beiden ersten Funktionen erlauben es, das Objekt zu setzen, das gesucht werden soll. Mit der dritten Funktion wird dann die eigentliche Hough-Transformation des übergebenen Kantenbildes berechnet, wobei die Hough-Ebene als Bild-Objekt zurückgeliefert wird.

Die Implementation der »search()«-Funktion führt die Transformation nach folgender Methode durch. Im Kantenbild wird jeder Pixel überprüft, ob er gesetzt ist. Wenn dieses der Fall ist, wird seine Position als Mittelpunkt in der Hough-Ebene benutzt. Um diesen Mittelpunkt wird in der Hough-Ebene das zu suchende Objekt gezeichnet, wobei die Werte aller betroffenen Pixel um Eins erhöht werden. Diese Konzept ist wesentlich effektiver wie eine direkte Umsetzung der Formel für die Hough-Transformation, da in diesem Fall nur dann eine Berechnung stattfinden muß, wenn der Pixel im Kantenbild gesetzt ist.

Um die Kreise bzw. Ellipsen nicht bei jeder Operation immer wieder neu berechnen zu müssen, wird der passende Kreis bzw. die passende Ellipse von den Funktionen »setCircle()« bzw. »setEllipse()« am Anfang einmal berechnet und in dem Vektor »d_figure« abgelegt.

4.4.2 Suche der Maxima in der Hough-Ebene

Die Hough-Transformation selbst liefert nicht die Position der zu suchenden Objekte. Vielmehr berechnet sie die Wahrscheinlichkeit eines jeden Punktes dafür, daß es sich um den Mittelpunkt des gesuchten Objektes handelt. Um Objekte mittels der Hough-Transformation zu finden und deren Koordinaten zu ermitteln, muß eine Maximasuche in der Hough-Ebene realisiert werden.

Das Flußdiagramm der Maximasuche, wie sie für diese Diplomarbeit implementiert wurde, ist in Abbildung 4.6 zu sehen. Der Algorithmus durchsucht einfach die Hough-Ebene nach dem größten Wert. Ist dieser Wert größer wie ein frei konfigurierbarer Grenzwert, werden die Koordinaten des Pixels mit dem größten Wert in einer Liste gespeichert.

In der Regel sucht man in der Hough-Ebene nicht nur ein Objekt von einem Typ sondern mehrere. Hat man bereits ein Objekt gefunden, kann es zu Problemen kommen, wenn man die Hough-Ebene jetzt

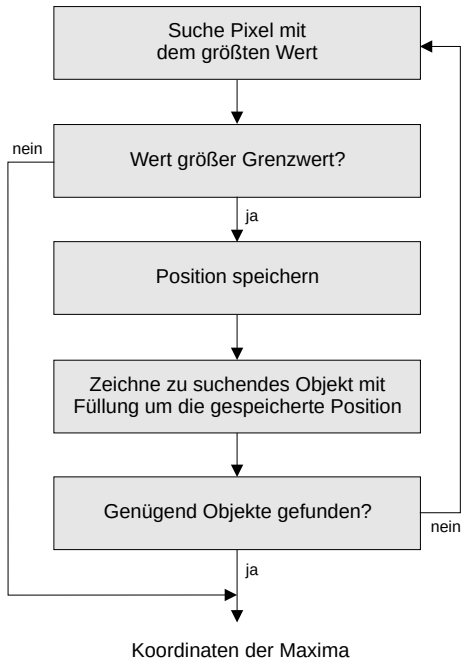


Abbildung 4.6: Flußdiagramm der Maximasuche in der Hough-Ebene

einfach nach dem zweit größten Wert durchsuchen würde, da es rund um ein Maximum meistens viele Nebenmaxima gibt; siehe Abbildung 4.7. Wenn die vom Benutzer vorgegebenen Abmessungen nicht exakt mit den Abmessungen der Objekte in den Bildern übereinstimmen, befinden sich die Maxima auf einem Kreis rund um den gesuchten Mittelpunkt des Objektes.

Es mußte also eine Methode gefunden werden, die mehrfache Erkennung ein und desselben Objektes zu verhindern. Der implementierte Algorithmus tut dieses, indem er, nachdem ein Objekt gefunden wurde, alle Werte in der Hough-Ebene, die sich unter dem gefundenen Objekt befinden, auf Null setzt. Dieses ist nur deshalb möglich, da keine sich überdeckenden Objekte in der Hough-Ebene gesucht werden.

Sobald genügend Objekte gefunden wurden, wird die Suche abgebrochen. Die Suche nach den Maxima wurde als Klasse »CFindCircleEllipse« implementiert. Das Interface der Klasse ist sehr einfach gehalten:

```
class CFindCircleEllipse
{
  public:
    void setFigureParameters (int a, int b, int max_figures,
                              int threshold = 0);
    void searchFigures (const CImageByteGray &img,
                        vector <CPoint> &results) const;

  private:
    void searchOneMaximum (const CImageByteGray &img,
                           CPoint &point) const;

  private:
    int d_a;           // "width" of the ellipse
    int d_b;           // "height" of the ellipse
    int d_max_figures; // maximum number of figures
```

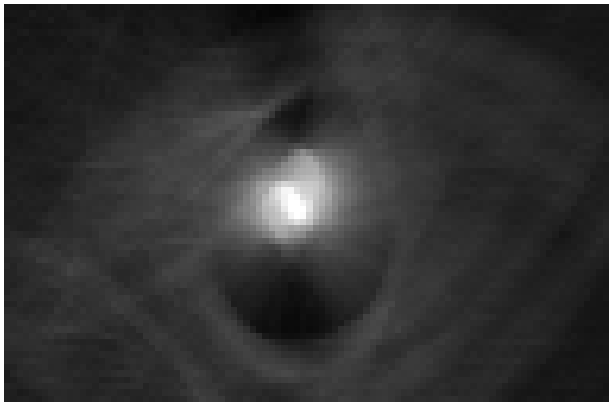


Abbildung 4.7: Nebenmaxima in der Hough-Ebene

```
    int d_threshold;
};
```

Mit der öffentlichen Funktion »setFigureParameters()« werden die Breite und die Höhe der zu suchenden Ellipse vorgegeben. Außerdem können mit dieser Funktion die maximale Anzahl der zu suchenden Ellipsen und der Grenzwert gesetzt werden. Die eigentliche Suche findet dann in der Funktion »searchFigures()« statt. Dieser wird die Hough-Ebene als Bild-Objekt übergeben. Die Koordinaten der gefundenen Ellipsen werden als Vektor von »CPoint«-Objekten zurückgegeben. Bei »vector<>« handelt es sich um einen Container der STL.

4.4.3 Konfigurationsdatenbank

Da nicht alle zu erkennenden Fässer und deren Spundlöcher die gleichen Dimensionen aufweisen, ist es nicht möglich, diese Dimensionen fest in die Bildverarbeitungsroutinen einzukompilieren. Vielmehr ist es notwendig, solche Einstellungen für verschiedene Typen von Fässern vorzuhalten und zur Laufzeit verändern zu können.

Es wurde deshalb eine kleine Datenbank für das Ablegen der Konfiguration entwickelt. Die Konfiguration wird in Dateien mit dem folgenden Format abgelegt:

```
...
<Schlüsselwort>: <Daten>
<Schlüsselwort>: <Daten>
...
```

Das Interface der implementierten Datenbankklasse sieht so aus:

```
class CDatabase
{
  public:
    void setDirectory (const string &dir);
    bool getEntry (const string &filename,
                   map <string, double> &result) const;
    bool setEntry (const string &filename,
                   const map <string, double> &result) const;
```

```
  private:
    string d_dir;  // name of the directory
};
```

Für jeden Faßtyp wird eine eigene Konfigurationsdatei in einem Verzeichnis abgelegt. Der Name dieses Verzeichnisses wird mit der Funktion »setDirectory()« gesetzt. Eine Konfigurationsdatei kann dann mit den Funktionen »getEntry()« bzw. »setEntry()« gelesen bzw. geschrieben werden.

Beim Lesen einer Konfigurationsdatei wird diese geparst und in einer Variablen vom Typ »map<>« abgelegt. Dieses Template ist Bestandteil der STL von ANSI C++ und definiert einen assoziativen Container [Str00]. In einem solchen Container werden Wertepaare, die jeweils aus einem Schlüssel und den eigentlichen Daten bestehen, abgespeichert. Der Parser legt alle Daten, die er in der Konfigurationsdatei finden kann, unter ihrem jeweiligen Schlüsselwort in der Map ab. Sei z.B. in der Konfigurationsdatei folgende Zeile zu finden:

```
...
threshold: 127
...
```

Dann könnte dieser Wert so auf dem Bildschirm ausgegeben werden:

```
...
map <string, double> data;
...
cout << "Threshold: " << map["threshold"] << endl;
...
```

Man könnte zwar wie früher bei ANSI C die Konfiguration einfach in eine Struktur einlesen; dieses hätte jedoch den entscheidenen Nachteil, daß der Parser nur Schlüsselwörter unterstützen könnte, für die in der Struktur eine Variable vorgesehen ist. Eine Map ist hingegen völlig dynamisch. Der Parser kann beliebige Daten unter beliebigen Schlüsselwörtern abgelegen, solange die Schlüsselwörter einmalig sind. Dieses ermöglich es, den Parser für Konfigurationsdateien mit beliebigen Schlüsselwörtern und Daten zu verwenden.

Der sehr allgemein gehaltene Parser hat sich im Laufe der Entwicklung der Bildverarbeitung als sehr hilfreich erwiesen. So konnte die obige Klasse z.B. auch für eine spezielle Konfigurationsdatei, die einige Einstellungen für den Framegrabber vornimmt, verwendet werden, obwohl diese Datei völlig andere Schlüsselwörter benutzt wie die Konfigurationsdateien der Fässer.

4.4.4 Interface der Bibliothek

Die Bildverarbeitung für den Befüllroboter wurde in Form einer Laufzeitbibliothek realisiert, die dann von dem Kommunikations-Daemon der Anlage, siehe Abbildung 2.4, benutzt wird. Die Schnittstelle zwischen der Bibliothek und dem Daemon wird durch die Klasse »CImageProcessingMain« definiert:

```
class CImageProcessingMain
{
  public:
    CImageProcessingMain (const std::string &dir_config,
                          const std::string &dir_images,
                          const std::string &url_help);
    ~CImageProcessingMain();
    void setType (const std::string &type);
```

```
        void getImageFramegrabber (void);
        void getImageFile (const std::string &filename);
        void searchCircles (std::vector <TUVision::CPoint> &points);
        ...
    };
```

Der Daemon erzeugt ein Objekt von dieser Klasse, wobei der Konstruktor dieser Klasse drei Parameter erwartet. Mit ihnen werden die Verzeichnisse gesetzt, in denen sich die Konfigurationsdatenbank und die Fernwartungsschnittstelle befinden. Außerdem wird die URL der Online Hilfe gesetzt. Dazu später noch mehr.

Wie bereits im vorherigen Abschnitt erläutert wurde, existiert eine Datenbank, in der die Konfigurationen für die verschiedenen Faßtypen abgelegt ist. Mit der Funktion »setType()« wählt der Daemon den Faßtyp aus, der verarbeitet werden soll. Die Funktion lädt die Konfiguration dieses Fasses mit der Klasse »CDatabase«.

Nachdem der Daemon den Typ der Fässer gewählt hat, muß er das Bild, der verarbeitet werden soll, »laden«. Für diesen Zweck existieren die beiden Funktionen »getImageFramegrabber()« und »getImageFile()«. Die erste Funktion benutzt den Framegrabber, um mit der Zeilenkamera ein Bild zu erfassen. Die Konfiguration des Framegrabbers ist in einer Konfigurationsdatei abgelegt, die ebenfalls mit der Klasse »CDatabase« geparst wird. Alternativ kann das Bild mit der zweiten Funktion auch einfach aus einer TIFF-Datei von der Festplatte gelesen werden. Dieses ist insbesondere für Tests sehr hilfreich, da die Aufnahme mit dem Framegrabber doch einige Zeit benötigt. Das Bild wird in dem Objekt der Klasse »CImageProcessingMain« gespeichert.

Die eigentliche Bildverarbeitung findet in der Funktion »searchCircles()« statt. Ein leicht vereinfachtes Flußdiagramm der Bildverarbeitung ist in Abbildung 4.8 zu sehen. Da die in der Anlage aufgenommenen Bilder, wie in Abschnitt 4.3.1 festgestellt wurde, viel zu dunkel sind und nicht den gesamten Wertebereich ausnutzen, besteht der erste Schritt der Bildverarbeitung aus einer Belichtungskorrektur. Diese wird mit den Klassen »CFilterHistogram« und »COperatorHistogram« der Bildverarbeitungsbibliothek durchgeführt. Nach der Belichtungskorrektur nutzt das Bild den gesamten Wertebereich und erscheint insgesamt heller.

An die Belichtungskorrektur schließt sich die Berechnung des Kantenbildes an. Hierfür wird der Sobel Filter der Bildverarbeitungsbibliothek verwendet. Es gibt zwar deutlich bessere Kantenfilter wie den Sobel Filter, diese konnten jedoch aus Zeitgründen nicht implementiert werden. Dieses hat sich jedoch als nicht problematisch herausgestellt, da man in den erzeugten Kantenbildern die Kanten der Fässer und Spundlöcher sehr gut erkennen kann.

Die Hough-Transformation erwartet ein binäres Kantenbild, welches mit der Klasse »COperatorThreshold« erzeugt wird, die ebenfalls Bestandteil der entwickelten Bildverarbeitungsbibliothek ist. Am Anfang der Entwicklung gab es bei diesem Schritt große Probleme bei der Wahl des passenden Grenzwertes für die Umwandlung in ein Binärbild. Wurde der Wert zu klein gewählt, waren im Bild zuviele Störungen vorhanden. War der Wert zu groß, verschwanden zwar die Störungen aber häufig auch die Kanten der Fässer und Spundlöcher. Erschwerend kam hinzu, daß der Grenzwert auch noch für jeden Faßtyp ein anderer war.

Gelöst wurde das Problem durch die Verwendung einer weiteren Belichtungskorrektur zwischen Sobel Filter und Grenzwertbildung. Diese sorgt dafür, daß das Kantenbild alle Werte von 0 bis 255 benutzt. Nimmt man jetzt einfach die Mitte zwischen diesen Werten als Grenzwert, erhält man sehr gute Ergebnisse. Durch diese Belichtungskorrektur wurde die Erkennungshäufigkeit deutlich verbessert.

Das binäre Kantenbild wird dann mit »CHoughCircleEllipse« in die Hough-Ebene transformiert. Die Hough-Transformation wird zweimal durchgeführt: für die Suche der Fässer und für die Suche der Spundlöcher. Die Klasse für die Hough-Transformation liefert die Hough-Ebene als Integer Graustufenbild zurück. Der große Integer Wertebereich ist für die weitere Verarbeitung eher hinterlich. Aus diesem Grund werden die beiden Hough-Ebenen mit der Klasse »COperatorConverter« in Byte Graustufenbilder konvertiert. Hierbei gab es anfangs jedoch Probleme, da die Größe der Werte in der Hough-

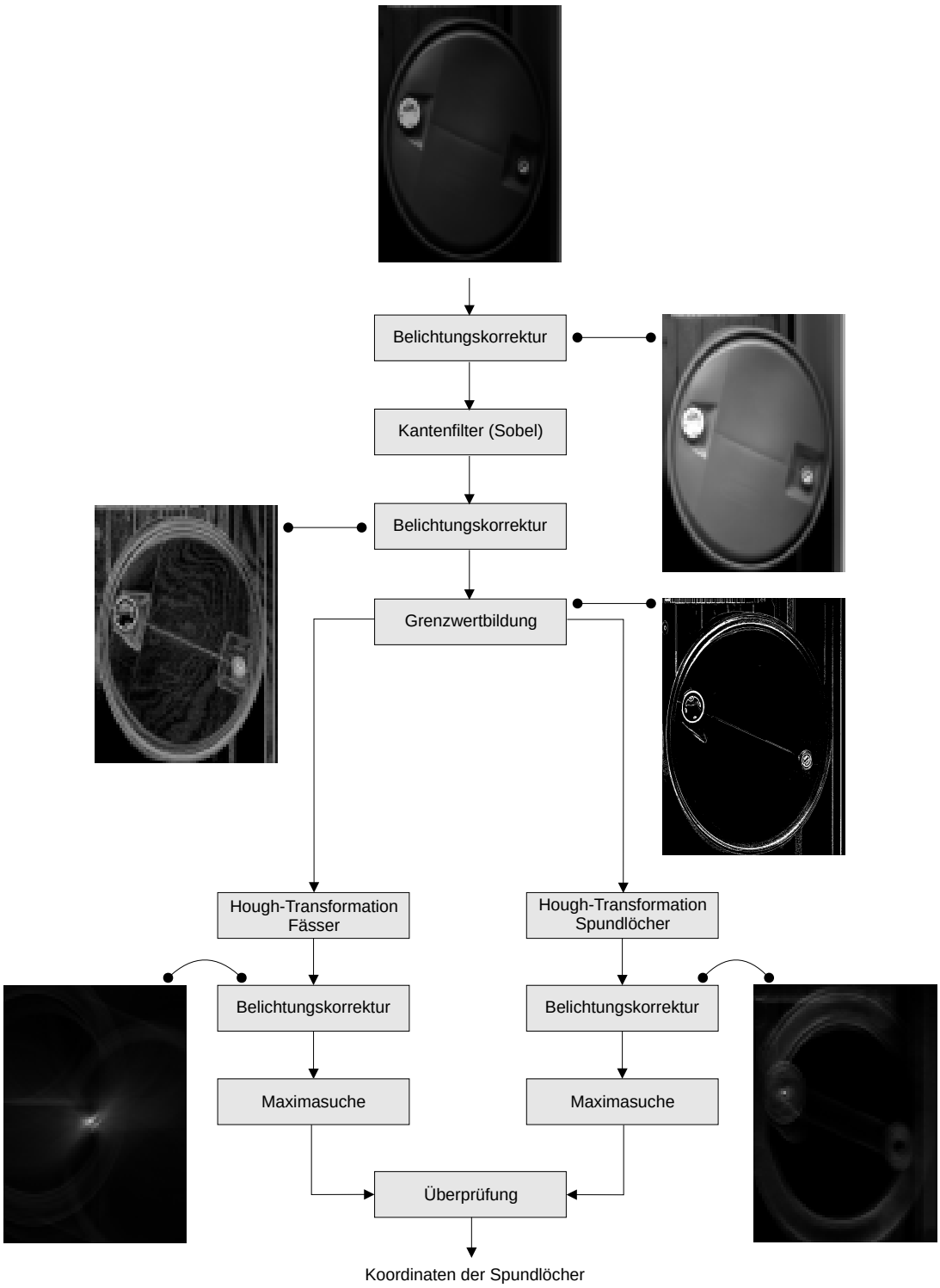


Abbildung 4.8: Flußdiagramm der Bildverarbeitung

Ebene von den Dimensionen der Objekte, die gesucht werden, abhängt. Nach der Konvertierung in das Byte Zahlenformat waren die Hough-Ebenen dann teilweise ganz schwarz. Auch hier war die Lösung die Anwendung einer Belichtungskorrektur. Diese bildet das Maximum der Hough-Ebene in die Farbe Weiß ab, so daß sich die Konvertierung dann problemlos durchführen läßt.

In den Hough-Ebenen werden dann mit der Klasse »CFindCircleEllipse« nach den Spundlöchern und den Fässern gesucht. Als Ergebnis erhält man nach diesem Schritt die Koordinaten der erkannten Spundlöcher und Fässer.

Um wirklich sicherzustellen, daß keine falschen Spundlöcher erkannt werden, werden alle erkannten Spundlöcher mit allen erkannten Fässern abgeglichen. Nur Spundlöcher, die sich in einem Ellipsenschlauch rund um einen erkannten Faßmittelpunkt befinden, werden an der Steuerungs-Daemon zurückgeliefert. Für diese Überprüfung werden zuerst der Abstand und der Winkel zwischen einem Faß und einem Spundlocher berechnet:

$$r = \sqrt{(x_b - x_h)^2 + (y_b - y_h)^2}$$

$$\varphi = \arctan \frac{y_b - y_h}{x_b - x_h}$$

Hierbei ist (x_b, y_b) die Position des Faßmittelpunktes und (x_h, y_h) der Mittelpunkt des Spundloches. Für den Abstand r soll gelten:

$$d_1 \leq r \leq d_2$$

Die beiden Grenzwerte lassen sich aus der Ellipsengleichung herleiten:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

Formt man diese in Polarkoordinaten um, erhält man:

$$d_1 = \frac{a_1 \cdot b_1}{\sqrt{b_1^2 \cdot \cos^2 \varphi + a_1^2 \cdot \sin^2 \varphi}}$$

$$d_2 = \frac{a_2 \cdot b_2}{\sqrt{b_2^2 \cdot \cos^2 \varphi + a_2^2 \cdot \sin^2 \varphi}}$$

Die Werte a_1, a_2, b_1 und b_2 sind entsprechend der gewünschten Ellipsenschläuche vorzugeben.

Der zu verwendene Ellipsenschlauch läßt sich über die Konfigurationsdatenbank für jeden Faßtyp getrennt konfigurieren. Er sollte möglichst schmall sein, so daß möglichst alle falsch erkannten Spundlöcher verworfen werden können. Allerdings besteht bei einem sehr schmallen Ellipsenschlau die Gefahr, daß richtig erkannte Spundlöcher verworfen werden, wenn die Faßmittelpunkte z.B. durch Schatten auf den Fässern nicht ganz exakt berechnet worden sind.

4.5 Ergebnisse

Um die Qualität der Spundlocherkennung zu testen, wurde die Klasse »CImageProcessingMain« mit den aufgenommenen Beispielbildern getestet, die in Abbildung 4.1 zu sehen sind.

Um die passenden Parameter für die Erkennung zu finden, wurde diese einmal mit irgendwelchen Grundwerten durchgeführt. Das dabei entstandene Kantenbild wurde mit dem Bildverarbeitungsprogramm »The GIMP« vermessen, um so die Abmessungen der Fässer und Spundlöcher zu ermitteln. Diese Werte wurden dann in der Konfigurationsdatenbank für den Faßtyp abgelegt. Danach wurde die Erkennung mit den neuen Parametern noch einmal durchgeführt.

Es wurden alle Spundlöcher und Fässer in allen getesteten Bildern erkannt. In nachfolgender Tabelle ist die Genauigkeit der Erkennung der Spundlöcher zu finden:

Faßtyp	max. Fehler
blaue Kunststoffässer	± 0 Pixel
dunkelblaue Metallfässer	± 2 Pixel
gelb blaue Metallfässer	± 2 Pixel
graue Metallfässer	± 0 Pixel
verrostete hellblaue Metallfässer	± 4 Pixel
orange Metallfässer	± 4 Pixel

Bei allen Angaben handelt es sich um Abschätzungen der maximalen Fehler. Insgesamt fiel es sehr schwer, überhaupt einen Meßfehler mit dem Auge zu erkennen, da die Spundlöcher in den aufgenommenen Bilder nicht scharf begrenzt sind. Vielmehr findet der Übergang zwischen Faßdeckel und Spundloch relativ fließend statt. Es darf außerdem nicht vergessen werden, daß die Meßergebnisse sehr stark von der Wahl der Parameter für den Erkennungsprozeß abhängen.

Die Zielvorgabe der Firma Feige war, die Spundlöcher möglichst mit einer Genauigkeit von ± 2 mm zu erkennen. Da eine Palette normalerweise 1200 mm x 1200mm groß ist, da die Kamera 2048 Pixel besitzt und da der Inkrementalgeber des Tragkettenförderers alle 0,43 mm einen Impuls erzeugt, war die geforderte Genauigkeit ± 4,6 bzw. ± 3,41 Pixel. Die Genauigkeit wurde mehr oder weniger immer erreicht, wenn die Parameter für den Erkennungsprozeß genau genug gewählt wurden. Die Ermittlung der richtigen Parameter wurde durch die von der Lampe erzeugten Schatten an den Rändern der Spundlöcher teilweise erschwert.

Die Ungenauigkeiten der Spundlocherkennung traten vor allem in der Transportrichtung auf. Verursacht werden sie durch die Schatten, die durch die Ausrichtung der Beleuchtung entstehen; siehe Abschnitt 4.3.3. Hier würde die Installation einer zweiten Lampe vermutlich zu einer verbesserten Erkennungsgenauigkeit führen.

In der Hough-Ebene sind die Maxima der Fässer sehr deutlich ausgeprägt, da die gesuchten Ellipsen sehr groß sind und so sehr viele Pixel zu einem Maximum beitragen. Bei den Spundlöchern sieht das anders aus. Hier sind die Unterschiede zwischen einem wirklichen und einem falschen Maximum recht klein. Dieses bereit bei der Erkennung Probleme, wenn sich auf der Palette weniger Fässer befinden, wie dieses laut des Konfigurationseintrages des Faßtyps der Fall sein müßte.

Befinden sich auf einer Palette normalerweise vier Fässer, so sucht der Algorithmus nach vier Spundlöchern und vier Fässern. Sind auf einer Palette jetzt aber nur zwei Fässer, würde man zwei falsche Spundlöcher und zwei falsche Fässer erkennen. Um dieses zu verhindern, besitzt der Algorithmus einen frei konfigurierbaren Grenzwert, den ein Maximum mindestens überschreiten muß, um als Maximum anerkannt zu werden. Dieses funktioniert bei der Faßerkennung sehr gut, bei der Erkennung der Spundlöcher aber eher schlecht.

Bei der Spundlochsuche werden oftmals Maxima zwischen zwei Fässern gefunden. Diese Maxima entstehen durch die Kanten zweier Fässer, die dicht nebeneinander stehen.

Der implementierte Abgleich von Spundlöchern und Faßmittelpunkten erkennt allerdings diese falschen Spundlöcher bei allen Beispielbildern und verwirft sie, so daß am Ende nur korrekte Spundlöcher an den Kommunikations-Daemon übermittelt werden.

4.6 Optimierung

Da die Befüllanlage die Schritte Bildaufnahme, Spundlocherkennung und Befüllung nicht parallel ausführen kann, war es wichtig, alle Schritte möglichst schnell auszuführen.

Die Geschwindigkeit der Bildaufnahme ist vor allem von der Beleuchtung und der Kamera abhängig, so daß eine Optimierung hier recht schwierig und teuer ist. Auch die Geschwindigkeit der Befüllung läßt sich kaum steigern, da ja bereits bestehende Anlagen mit dem System ausgerüstet werden sollen. Da diese beiden Schritte ausschieden, war insbesondere die Optimierung der Zeit, die für die Spundlocherkennung benötigt wird, wichtig.

Vor der Optimierung dauerte der gesamte Erkennungsprozeß etwas über eine Minute. Das gesamte Programm wurde deshalb mit einem Profiler untersucht. Diese ermittelt genau, wie oft eine Funktion aufgerufen wird und wieviel Prozent der gesamten Laufzeit in einer Funktion verbracht wird.

Wie es nicht anders zu erwarten war, benötigten die beiden Hough-Transformationen mit Abstand die meiste Rechenzeit. Dieses war an sich nicht erstaunlich. Interessant war allerdings, welche Teile der Implementation der Hough-Transformation am meisten Zeit benötigten.

Es fiel auf, daß das Programm über 10 Prozent seiner Rechenzeit in den Routinen »CImage::getW()« und »CImage::getH()« verbrachte, mit denen die Breite und die Höhe eines Bildes ermittelt werden kann. Verursacht wurde dieser genorme Zeitbedarf durch Schleifen wie die folgende:

```
...
for (y=0; y < src.getH(); y++)
...
```

Eigentlich sollte man vermuten, daß die »getH()«-Funktion hier nur einmal aufgerufen wird und das Resultat dann automatisch immer wieder für die Abbruchbedingung der Schleife verwendet wird. Dieses ist jedoch nicht der Fall. Vielmehr wird die Funktion bei jeder Überprüfung der Abbruchbedingung neu aufgerufen. Zählt die Schleife also bis 1000, so wird die Funktion auch tausendmal aufgerufen.

Um dieses Problem zu lösen, wurden die beiden Funktionen »inline« deklariert. Dadurch ist dem Compiler bekannt, was in den Funktionen genau passiert. Er muß diese dann nur einmal am Anfang der Schleife ausführen.

Eine zweite Schwäche, die der Profiler aufdeckte, war die Verwendung der Funktionen »setPixel()« und »getPixel()«. Um den Wert eines Pixels um Eins zu erhöhen, wurde dieser zuerst mit »getPixel()« gelesen, dann um Eins erhöht und schließlich mit »setPixel()« wieder im Bild gespeichert.

Beide Funktionen müssen die Position des Pixels im Speicher berechnen. Hierfür sind jeweils zwei Additionen und zwei Multiplikationen notwendig, siehe Abschnitt 3.5.3. Wie leicht einsichtig ist, berechnen beide Funktionen die gleiche Adresse. Hier wird also eine Berechnung zweimal durchgeführt. Außerdem sind die Funktionen so ausgelegt, daß sie nicht nur mit Graustufenbildern sondern auch mit Mehrkanalbildern funktionieren. Hierdurch werden allerdings doppelt so viele Operationen wie bei einer Funktion, die speziell für Graustufenbilder ausgelegt ist, benötigt.

Um die Geschwindigkeit der Hough-Transformation zu steigern, wurde deshalb von der Klasse »CImageIntGray« eine eigene lokale Klasse abgeleitet. Diese enthält zusätzlich die Funktion »increment()«. Diese kommt mit einer Multiplikation und einer Addition für die Berechnung der Position des Pixels aus. Mit diesem Trick benötigt man also pro Erhöhung um Eins nur noch 25 Prozent der ursprünglichen Operationen.

Diese beiden Optimierungen führten dazu, daß für den gesamten Erkennungsprozeß auf einem PentiumIII mit 700 MHz nur noch 25 Sekunden benötigt werden.

Es wurde versucht, die Geschwindigkeit durch eine Reduzierung der Auflösung noch weiter zu steigern. Bei diesem Ansatz wird die Auflösung vor der Hough-Transformation z.B. halbiert, so daß nur noch 25 Prozent der Rechenzeit benötigt wird. An den so gefundenen Stellen für potentielle Spundlö-

cher und Faßmittelpunkte führt man dann noch eine örtlich begrenzte Hough-Transformation mit voller Auflösung durch.

Dieser Ansatz wurde jedoch verworfen. Bei der Erkennung der Spundlöcher kam es zu noch mehr Fehlerkennungen jeweils zwischen den Fässern. Da bei den Spundlöchern sowieso sehr wenige Pixel zu den Maxima beitragen, verschlechtert hier die Reduzierung der Auflösung die Erkennung deutlich. Bei den Fässern hat man dieses Problem nicht. Allerdings nehmen die Fässer fast das gesamte Bild ein, so daß die örtlich begrenzten Hough-Transformationen doch wieder mehr oder weniger auf das gesamte Bild angewendet werden müssen. Hier erhöht sich also eher die benötigte Rechenleistung.

Kapitel 5

Fernwartung

Neben der besseren Erkennungssicherheit der Spundlöcher war die Möglichkeit der Fernwartung des Befüllroboters einer der Hauptgründe, ein neues System zu konzipieren. Statt für Änderungen und bei Problemen jedesmal einen Servicetechniker vor Ort schicken zu müssen, sollten sich diese Aufgaben zukünftig per Fernwartung erledigen lassen.

5.1 Vernetzung

Die Verbindung zwischen dem PC der Maschine und dem PC des Servicetechnikers sollte hierbei per analogem Modem, per ISDN-Karte, per Ethernet oder per Internet hergestellt werden können. Das Internet verwendet als Netzwerkprotokoll IP. Es lag also die Idee nahe, auch für die drei anderen Verbindungsarten das Netzwerkprotokoll IP zu verwenden, das sich in den letzten Jahren als der Standard für Netzwerkprotokolle herausgebildet hat.

Die Verwendung eines Netzwerkprotokolls hat gegenüber der Verwendung einer reinen Terminalverbindung den Vorteil, daß über eine physikalische Verbindung mehrere Sachen »parallel« übertragen werden können. Während man z.B. ein Bild von einer Palette überträgt, kann man gleichzeitig Einstellungen an der Anlage vornehmen oder diese steuern. Ermöglicht wird diese Parallelität durch die Verwendungen von Paketen. Alle Daten werden in IP-Pakete verpackt, die eine bestimmte maximale Größe haben, und von einer Quelladresse zu einer Zieladresse übertragen.

Die Übertragung von IP-Paketen über ein Modem oder eine ISDN-Karte erfolgt über das Protokoll PPP. Das in der Anlage verwendete Betriebssystem Linux verfügt nicht nur über einen PPP-Client sondern auch über einen PPP-Server: »pppd«. Dieser Daemon überwacht beliebige serielle Schnittstellen. Geht über ein Modem an einer seriellen Schnittstelle ein Anruf ein, baut der Daemon eine PPP-Verbindung zu dem anrufenden Client auf. Für ISDN-Karten bietet der Linux Kernel eine Modem Simulation. Jeder ISDN-Kanal besitzt unter Linux ein Device, das sich genau wie ein analoges Modem verhält, das an einer seriellen Schnittstelle des PCs angeschlossen ist. Neben dem PPP-Protokoll unterstützt Linux auch IP-Verbindungen über lokale Netzwerke wie das Ethernet.

Nachdem das Netzwerkprotokoll in Form von IP feststand, wurde dann überlegt, welche Funktionen die Fernwartungsschnittstelle der Anlage genau haben sollte. Hierbei ließen sich zwei Schwerpunkte festmachen. Zum einen benötigt der Servicetechniker Bildmaterial, anhand dessen er aus der Ferne feststellen kann, was genau schiefgelaufen ist bzw. wie die neuen Fässer, die verarbeitet werden sollen, aussehen. Zum zweiten wird eine Möglichkeit benötigt, die Anlage bei Fehlern oder im Fall von neuen Faßtypen umzukonfigurieren.

Es mußte also ein Programm entwickelt werden, was in der Anlage aufgenommene Bilder über die IP-Verbindung überträgt und auf dem PC des Servicetechnikers darstellt. Außerdem müßte dieses Programm Funktionen haben, um die Konfiguration über die IP-Verbindung zu verändern. Nach einigen

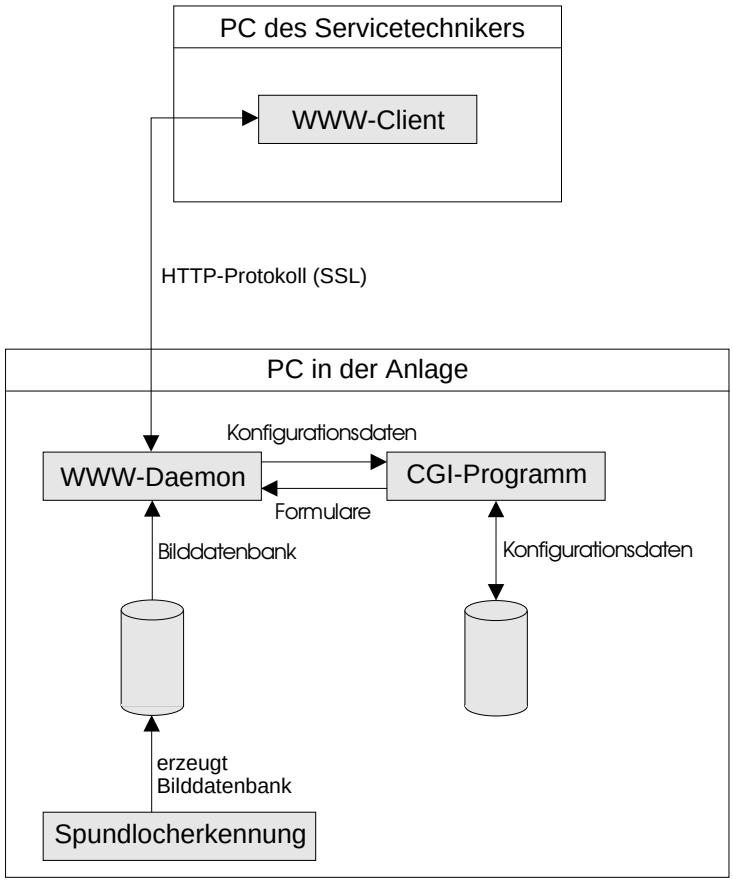


Abbildung 5.1: Übersicht über die Fernwartungsschnittstelle

Überlegungen wurde beschlossen, keines solches Programm zu entwickeln, sondern auf bestehende Lösungen zurückzugreifen.

Eine solche Lösung existiert in Form des HTTP-Protokolls bereits seit einigen Jahren und wird auf der ganzen Welt eingesetzt. Dieses Protokoll bildet die Grundlage des WWWs. Für eigentlich jedes verbreitete Betriebssystem gibt es fertige Clients für dieses Protokoll, so daß der Servicetechniker ein beliebiges Betriebssystem verwenden kann. Linux enthält außerdem einen freien WWW-Server: den Apache. In Abbildung 5.1 ist eine Übersicht über die implementierte Fernwartungsschnittstelle zu sehen.

Die Verwendung des Protokolls HTTP hat nicht nur den Vorteil, daß man dieses nicht selbst implementieren muß, sondern zusätzlich noch den Vorteil, daß es eine per SSL verschlüsselte Variante gibt, die HTTPS genannt wird. Durch die Verschlüsselung wird wirksam verhindert, daß die Daten auf dem Übertragungsweg von Dritten manipuliert oder auch nur gelesen werden können. Dieses ist insbesondere bei Übertragungen über das Internet sehr wichtig, da ansonsten immer die Gefahr besteht, daß ein Hacker die Anlage umkonfigurieren könnte. Neben der Verschlüsselung der Daten kann SSL auch zur Authentifizierung verwendet werden.

Die meisten WWW-Clients unterstützen diese Verschlüsselung. Auch vom Apache Daemon gibt es eine Version, die diese Verschlüsselung unterstützt.

Gegen den Einsatz der Verschlüsselung spricht eigentlich nur, daß einige Länder den Einsatz von solchen Verfahren per Gesetz verbieten. Aus einigen anderen Ländern wie den USA dürfen Geräte, die solche Verfahren benutzen, nicht exportiert werden, da sie den Kriegswaffenkontrollgesetzen unterliegen.

5.2 Bilddatenbank

Um dem Servicetechniker die Diagnose zu erleichtern, legt die Bildverarbeitungsklasse »CImageProcessingMain« die aufgenommenen und berechneten Bilder der letzten verarbeiteten Paletten auf der Festplatte ab. Für jede Palette werden die folgenden Bilder gespeichert:

- Originalbild
- Originalbild mit Belichtungskorrektur
- Kantenbild
- binäres Kantenbild
- Hough-Ebene der Spundlöcher
- Hough-Ebene der Fässer
- Originalbild mit erkannten Spundlöchern und Fässern

Diese Bilder können benutzt werden, um Fehler in der Konfiguration aufzudecken. Ist z.B. die Belichtungszeit völlig falsch gewählt worden, wird man dieses in den Originalbildern sehen. Die Kantenbilder können benutzt werden, um die zu suchenden Objekte mit einem Programm wie »The Gimp« zu vermessen und so die Parameter für die Erkennung zu ermitteln. In den Hough-Ebenen läßt sich gut erkennen, ob sich die Maxima an den zu erwartenden Stellen befinden.

Das letzte Bild liefert schließlich genaue Informationen darüber, was wo erkannt wurde, siehe Abbildung 5.2. Erkannte Spundlöcher bzw. Fässer werden mit kleinen bzw. großen grünen Fadenkreuzen gekennzeichnet. Der Ellipsenschlau, der festlegt, ob ein Spundloch gültig ist, wird gelb eingezeichnet. Als richtig erkannte Spundlöcher werden mit einem zusätzlichen kleinen blauen Kreis markiert. Damit auch dieses Bild zu Vermessung der Objekte herangezogen werden kann, haben die beiden Fadenkreuztypen Maßstäbe.

Alle Bilder mit Ausnahme des binären Kantenbildes werden in vier Versionen auf der Festplatte abgelegt. Zum einen wird jedes Bild im TIFF-Format auf der Festplatte abgelegt. Dieses Format ist verlustlos, benötigt allerdings pro Bild zwischen 7 und 21 MByte Speicherplatz. Es ist zwar kein Problem, solche Datenmengen auf einer Festplatte zu speichern, allerdings würde die Übertragung eines Bildes über eine ISDN-Verbindung, die eine Bandbreite von 64 kBit besitzt, bis zu 45 Minuten dauern. Dieses ist natürlich unakzeptabel.

Aus diesem Grund werden die Bilder außerdem in den Maßstäben 1:1, 1:2 und 1:4 im JPEG-Format gespeichert. Je nach Maßstab und Bilddaten belegt ein Bild dann nur noch Speicher im Bereich von ca. 100 kByte bis zu 1 MByte. Die reduzierte Auflösung reicht oftmals aus; insbesondere spricht natürlich die kurze Übertragungszeit für die Bilder mit einer geringeren Auflösung. So läßt sich ein 1:4 JPEG-Bild in ungefähr 13 Sekunden über eine ISDN-Verbindung übertragen.

Erreicht werden diese kleinen Dateigrößen durch die verlustbehaftete Komprimierung des JPEG-Formats. Durch diese Komprimierung leidet mehr oder weniger auch die Bildqualität, was für diesen Einsatz jedoch keine große Rolle spielt. Allerdings sind die so komprimierten Bilder für eine automatische Bildverarbeitung nicht mehr zu gebrauchen, da die entstehenden Fehler die Bildverarbeitung im Gegensatz zum menschlichen Auge stören. Soll ein in der Anlage aufgenommenes Bild also auf dem PC des Servicetechnikers mit der entwickelten Bildverarbeitung ausgewertet werden, muß unbedingt die TIFF-Version des Bildes hierfür verwendet werden.

Die Anzahl der Paletten, deren Bilder gespeichert werden, läßt sich bei der Übersetzung der Bibliothek frei konfigurieren und ist nur von dem verfügbaren Platz auf der Festplatte abhängig. Pro Palette werden ungefähr 50 MByte Speicherplatz benötigt.

Abbildung 5.2: Originalbild mit eingezeichneten Spundlöchern und Fässern

Neben den eigentlichen Bildern erzeugt die Klasse »CImageProcessingMain« auch gleich eine HTML-Oberfläche, wie sie in Abbildung 5.3 zu sehen ist. Diese Oberfläche enthält neben Links auf die eigentlichen Bilder auch gleich Informationen darüber, mit welchen Parametern die Erkennung abgelaufen ist. Es reicht hier nicht, einfach nur den Faßtyp anzuzeigen, da sich die Parameter des Faßtyps ja ändern können. Außerdem kann der Servicetechniker zu jedem Bild eine Online Hilfe abrufen, die erklärt, was in dem Bild zu sehen ist und durch welche Parameter es beeinflußt werden kann.

Die Oberfläche und die Bilder werden, wie bereits in Abbildung 5.1 zu erkennen war, über einen WWW-Server an den WWW-Client auf dem PC des Servicetechnikers übertragen. Wie man in der Abbildung sehen kann, sind Bildverarbeitung und Fernwartung unabhängig voneinander. Die Bildverarbeitung legt einfach die Bilder ab und erzeugt die HTML-Oberfläche. Der WWW-Daemon reagiert dann auf die Anfragen des Servicetechnikers und liefert die entsprechenden Bilder an diesen aus.

Damit die JPEG-Bilder auf dem Client möglichst schnell betrachtet werden können, werden sogenannte progressive JPEG-Dateien erzeugt. Hier zeigt der Client zuerst das Bild mit einer niedrigen Auflösung und einer schlechten Schärfe auf dem Bildschirm an. Im Laufe der Übertragung wird das Bild dann immer besser. Dieses ermöglicht es, teilweise die Übertragung schon frühzeitig abzubrechen, da man vielleicht bereits alles erkannt hat, was man erkennen wollte, oder weil man vielleicht das falsche Bild ausgewählt hat.

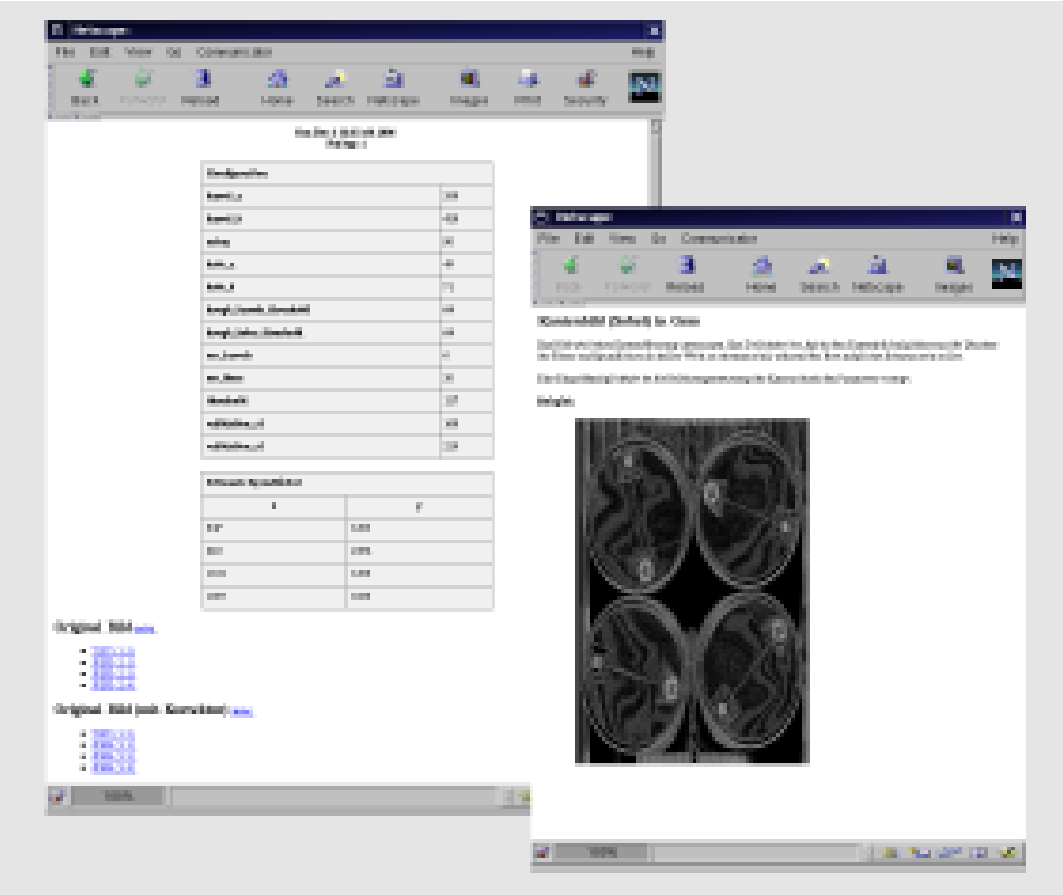


Abbildung 5.3: Fernwartungsschnittstelle: Bilddatenbank und Online Hilfe

5.3 CGI-Programm

Der zweite Teil der Fernwartungsschnittstelle besteht aus einem CGI-Programm. Dieses Programm wird bei Bedarf von dem WWW-Daemon gestartet. Benutzt wird dieses Programm, um dem Servicetechniker die Möglichkeit zu geben, die Datenbank mit den Informationen über die verschiedenen Fässer zu modifizieren.

Greift der Servicetechniker mit seinem WWW-Browser auf eine bestimmte URL des WWW-Daemons zu, startet der Daemon das CGI-Programm. Dieses erzeugt dann eine HTML-Seite, die vom Daemon an den Client übertragen wird, der diese dann auswertet und auf dem Bildschirm anzeigt, siehe Abbildung 5.1. Eine solche Seite kann z.B. ein Formular enthalten. Füllt der Servicetechniker dieses aus und schickt es an den Daemon zurück, so übermittelt dieser die Daten wiederum an das CGI-Programm.

Der WWW-Daemon und das CGI-Programm kommunizieren über die sogenannte CGI-Schnittstelle [CGI]. Ruft der Benutzer mit seinem WWW-Browser z.B. folgende URL auf:

```
http://localhost/cgi-bin/test?variable1=30&variable2=20
```

Dann startet der WWW-Daemon das CGI-Programm »test« und übergibt ihm die Daten, die in der URL nach dem Fragezeichen folgen, als Umgebungsvariable »QUERY_STRING«:

```
QUERY_STRING = "variable1=30&variable2=20"
```

Es ist dann die Aufgabe des CGI-Programms, diese Zeichenkette zu parsen. Obiges Beispiel enthält z.B. zwei Variablen, den man jeweils einen Wert zugewiesen hat. Jede Variable hat einen frei definierbaren Namen. Der Name und der Wert einer Variablen werden durch ein Gleichzeichen getrennt. Mehrere Variablen werden durch ein kaufmännisches Und-Zeichen getrennt. Zeichen, die nicht Bestandteil des ASCII-Zeichensatzes sind, werden im CGI-Interface als zweistellige Hexadezimalzahlen übertragen.

Die CGI-Fernwartungsschnittstelle besteht aus drei Klassen. Die Klasse »CCgi« enthält einen CGI-Parser; sie wertet die übergebene Umgebungsvariable aus und legt die einzelnen Variablen in einer STL Map ab. Diese Klasse wurde von mir bereits vor der Diplomarbeit entwickelt und für diese nur etwas erweitert.

Die Klasse »CMain« enthält das eigentliche Fernwartungsinterface. Von dieser Klasse wird der CGI-Parser »CCgi« und die schon für die Bildverarbeitung entwickelte Klasse »CDatabase« intern benutzt.

Wird das CGI-Programm aufgerufen, so ruft »CMain« zuerst den CGI-Parser auf und wertet die CGI-Variable mit dem Namen »form_type« aus. Diese legt fest, was das CGI-Programm machen soll. Ist die Variable leer, so befindet sich das Programm im Grundzustand. In diesem Modus gibt das Programm das Formular aus, das in Abbildung 5.4 zu sehen ist. In diesem Formular kann der Servicetechniker wählen, ob er einen bestehenden Eintrag für einen Faßtyp ändern oder löschen möchte. Außerdem ist es möglich, einen neuen Eintrag für einen Faßtyp anzulegen. Drückt der Benutzer einen der Knöpfe mit dem Namen »Absenden«, so werden die in den Feldern eingegebenen Werte als CGI-Variablen an das CGI-Programm übergeben. Nach dem Absenden hat »form_type« den Wert »barrel_selected« oder »barrel_delete«.

»barrel_selected« bedeutet, daß der Benutzer ein Faß konfigurieren möchte, während »barrel_delete« bedeutet, daß ein Faß aus der Datenbank gelöscht werden soll. Das Löschen wird sofort von »CMain« ausgeführt. Danach erscheint wieder die vorherige Auswahlmaske. Soll ein Faß konfiguriert werden, erscheint das Formular, das in Abbildung 5.5 zu sehen ist. Hier hat der Benutzer die Möglichkeit, alle Parameter, die für die Spundlocherkenung benötigt werden, einzustellen. Um den Servicetechnikern die Wahl der richtigen Werte zu erleichtern, enthält der Formular auch knappe Erklärungen, wofür der jeweilige Werte gedacht ist.

Wird dieses Formular mit »Absenden« an das CGI-Programm geschickt, hat die Variable »form_type« den Wert »barrel_configured«. Das Programm weiß dann, daß es die Werte in der Datenbank mit Hilfe der Klasse »CDatabase« ablegen muß.

5.4 Bandbreite

Das Hauptproblem bei der Fernwartungsschnittstelle stellt die in der Regel sehr kleine verfügbare Bandbreite dar. Über einen ISDN-Kanal können ca. 8 kByte/s übertragen werden, bei einer Modem-Verbindung ist es sogar nur die Hälfte. Aus diesem Grund ist eine leistungsfähige Komprimierung der Bilddaten notwendig. Im nachfolgenden werden zwei Komprimierungsverfahren genauer untersucht.

5.4.1 Progressive JPEG

Bei dem bereits in Abschnitt 5.2 vorgestellten Verfahren progressive JPEG ist vor allem interessant, wie schnell der Benutzer ein Bild mit einer akzeptablen Qualität auf dem Bildschirm sieht.

Es wurde deshalb ein von der Spundlocherkennung erzeugtes JPEG-Bild genauer untersucht. Verwendet wurde für diese Untersuchung das Testbild »blaue Kunststoffässer«. Das von der Spundlocherkennung erzeugte JPEG-Bild im Maßstab 1:1 belegt auf der Festplatte 733 kByte. Um den Qualitätsgewinn, der im Laufe der Übertragungszeit stattfindet, zu untersuchen, wurde das Bild dann mit dem Programm »split« in Dateien mit einer Größe von jeweils 160 kByte aufgeteilt. Um eine Datei einer solchen Größe über eine ISDN-Verbindung zu übertragen, werden ca. 20 Sekunden benötigt.

In Abbildung 5.6 ist das Ergebnis der Untersuchung in Form eines Ausschnittes aus dem Bild zu er-

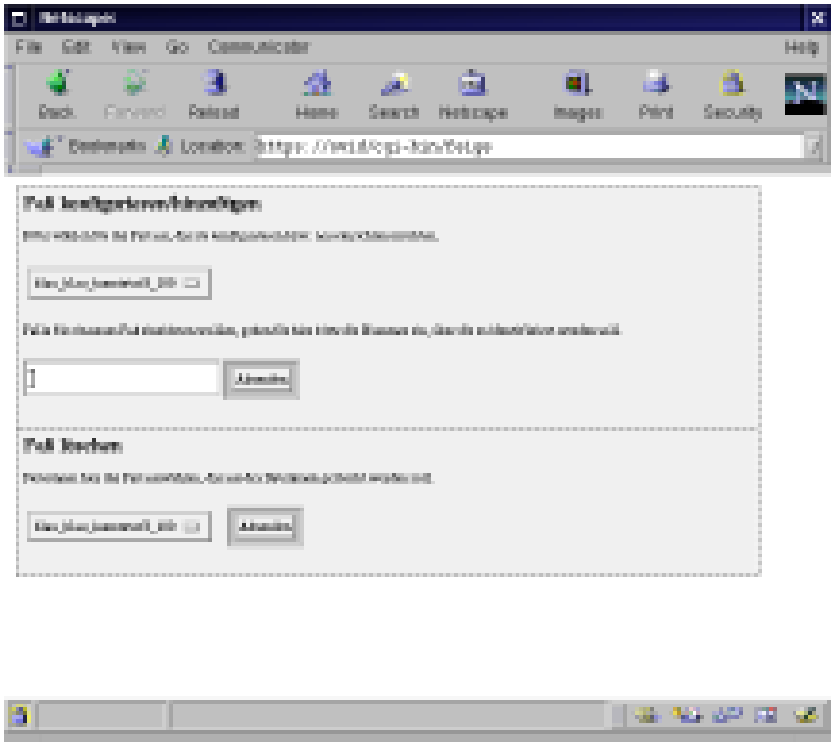


Abbildung 5.4: Fernwartungsschnittstelle: Auswahl des zu konfigurierenden Fasses

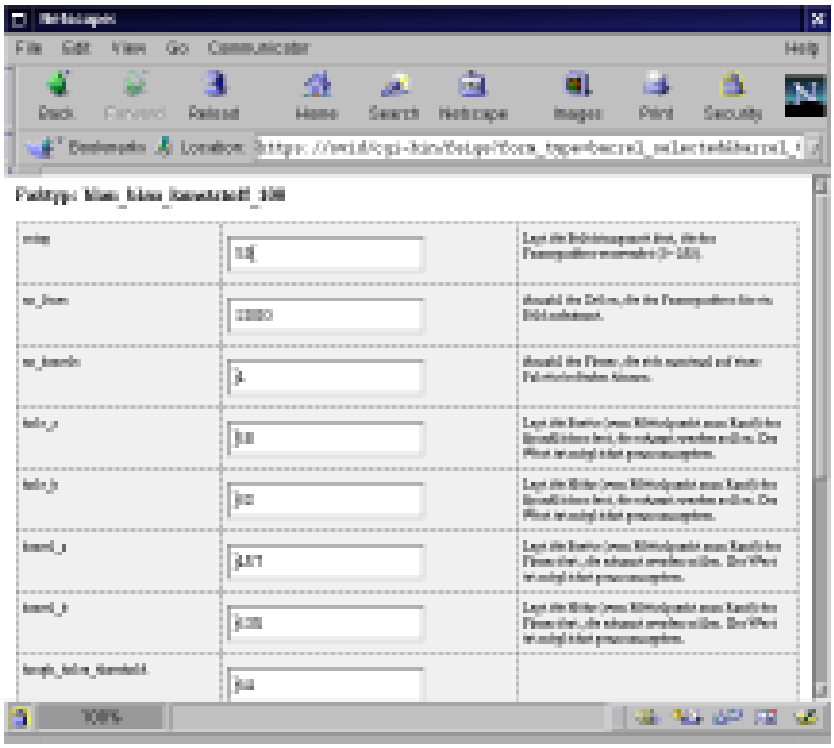
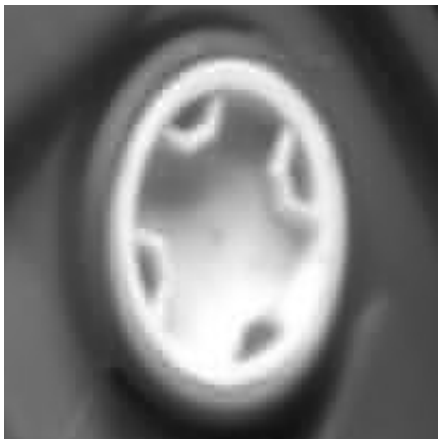
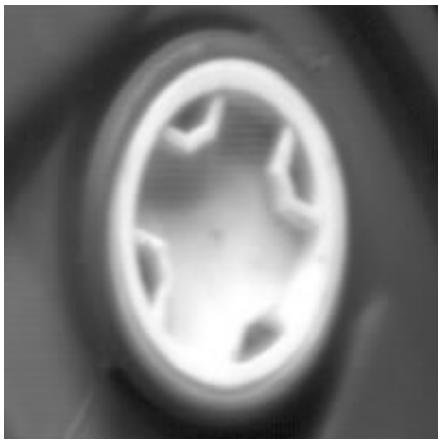


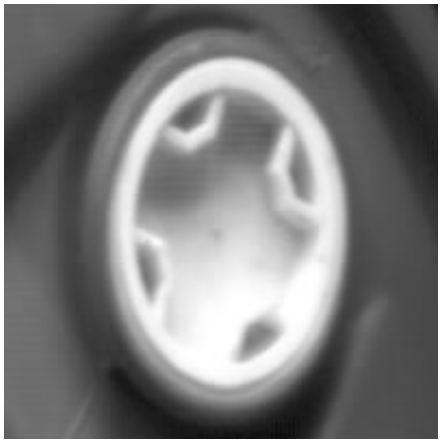
Abbildung 5.5: Fernwartungsschnittstelle: Konfiguration eines Fasses



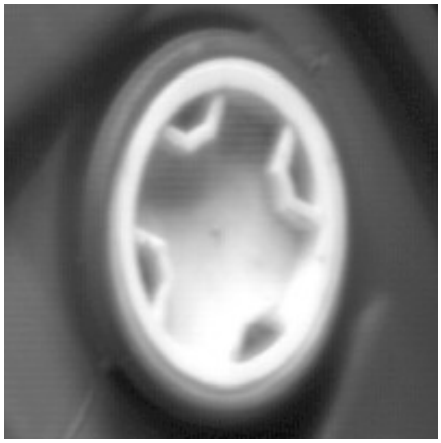
(a) nach 20 Sekunden (160 kByte)



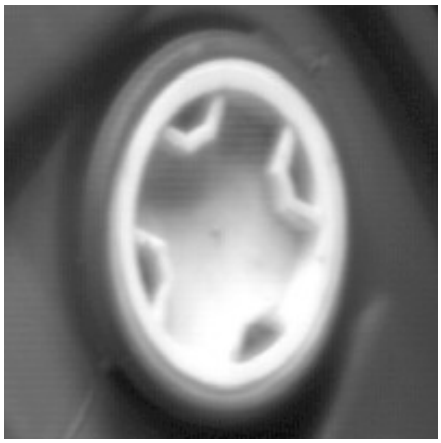
(b) nach 40 Sekunden (320 kByte)



(c) nach 60 Sekunden (480 kByte)



(d) nach 80 Sekunden (640 kByte)



(e) nach 91 Sekunden (733 kByte)

Abbildung 5.6: Test von progressive JPEG

kennen. Während nach 20 Sekunden noch deutliche Artefakte zu erkennen sind, ist das Bild, das der Benutzer nach 40 Sekunden sieht, selbst bei einer starken Vergrößererung kaum noch von dem vollständig übertragenen Bild nach 91 Sekunden zu unterscheiden. Das Bild nach 20 Sekunden enthält zwar Störungen, diese sind aber nicht so stark, daß man das Bild nicht mehr gebrauchen könnte.

Als Ergebnis der Untersuchung läßt sich feststellen, daß die progressive JPEG-Methode zu einer Ersparnis bei der Übertragungszeit um den Faktor zwei bis vier gegenüber dem normalen JPEG-Format führt.

5.4.2 TIFF mit LZW-Komprimierung

Die Fernwartungsschnittstelle legt die Bilder im TIFF-Format standardmäßig unkomprimiert ab. Das TIFF-Format unterstützt zwar die sehr leistungsfähige LZW-Komprimierung, allerdings enthalten die neusten Versionen der verwendeten »libtiff« keine Unterstützung mehr für diese Komprimierung, da diese in einigen Ländern patentiert ist.

Obwohl die LZW-Komprimierung zur Zeit nicht mehr zur Verfügung steht, ist natürlich trotzdem interessant, zu untersuchen, wieviel Bandbreite durch die Verwendung dieser Komprimierung gespart werden könnte. Es wurden deshalb das Testbild »blaue Kunststoffässer« einmal ohne und einmal mit LZW-Komprimierung verarbeitet. Das Ergebnis ist in nachfolgender Tabelle zu erkennen:

	ohne Komprimierung	LZW-Komprimierung
Originalbild	6,84 MByte	1,91 MByte
Originalbild mit Belichtungskorrektur	6,84 MByte	2,54 MByte
Kantenbild	6,84 MByte	2,69 MByte
binäres Kantenbild	0,85 MByte	0,22 MByte
Hough-Ebene Spundlöcher	6,84 MByte	1,66 MByte
Hough-Ebene Fässer	6,84 MByte	3,75 MByte
Ergebnisbild	20,50 MByte	4,39 MByte

Im Mittel werden die mit dem LZW-Algorithmus komprimierten Daten um mehr als Faktor drei kleiner. Besonders das farbige Ergebnisbild läßt sich sehr gut komprimieren, da dieses bis auf die eingezeichneten farbigen Elemente nur aus Graustufen besteht.

5.5 Integration der SPS-Fernwartung

Die SPS, die für die Steuerung der Anlage zuständig ist, verfügt ebenfalls über eine Fernwartungsschnittstelle. Diese erfolgt über eine serielle Verbindung mit der SPS. An die serielle Fernwartungsschnittstelle der SPS kann direkt oder über eine Modemverbindung ein PC angeschlossen werden. Der PC kommuniziert mit der SPS mittels eines proprietären Protokolls. Diese Tatsache erschwert die Integration der SPS-Fernwartung in die Fernwartungsschnittstelle der gesamten Anlage.

Die SPS-Fernwartung erlaubt mit einem Tool des Herstellers, den Inhalt von Variablen in der SPS zu lesen und zu verändern. Außerdem kann die SPS über diese Schnittstelle programmiert werden.

Die SPS-Fernwartung wird aber auch für die Fernwartung der Spundlocherkennung benötigt. Möchte der Servicetechniker z.B. einen neuen Faßtyp in der Konfigurationsdatenbank ablegen, so benötigt er hierfür einige Bilder aus der Anlage. Um diese Bilder aufzunehmen, muß aber z.B. der Tragkettenförderer gestartet werden. Dieses geht aufgrund des proprietären Protokolls der SPS nur über einen Mann vor Ort oder über die SPS-Fernwartungsschnittstelle. Es ist also festzustellen, daß die beiden Fernwartungsschnittstellen parallel benötigt werden.

Diese Parallelität ist jedoch ein Problem, da die Tools der SPS nur dann funktionieren, wenn die SPS und der PC direkt über eine serielle Verbindung verbunden sind. Das würde bei einer Fernwartung

bedeutet, daß man gleichzeitig zwei Modem-Verbindungen aufbauen müßte: eine zur SPS und eine zum PC in der Anlage. Dieses ist nicht nur sehr umständlich, sondern kann auch zu einem Problem werden, wenn die Anlage garnicht per Telefonleitung sondern nur per Internet zu erreichen ist.

Die Lösung für dieses Problem besteht darin, die serielle Verbindung zwischen der SPS und dem PC des Servicetechnikers durch die IP-Verbindung zwischen dem PC der Anlage und dem PC des Servicetechnikers zu tunneln. Hierfür wird die serielle Schnittstelle der SPS mit einer der seriellen Schnittstellen des PCs in der Anlage verbunden. Auf diesem PC wird ein Daemon installiert, der es einem Programm auf einem anderen Rechner über eine IP-Verbindung erlaubt, auf die serielle Schnittstelle zuzugreifen. Auf dem PC des Servicetechnikers wird schließlich ein Treiber installiert, der eine virtuelle serielle Schnittstelle implementiert.

Greift jetzt also das Fernwartungstool der SPS auf die virtuelle serielle Schnittstelle zu, so verpackt der Treiber die Daten und verschickt sie als IP-Pakete an den PC in der Anlage. Der Daemon der Anlage packt diese IP-Pakete wieder aus und schickt die Daten an die serielle Schnittstelle, die mit der SPS verbunden ist. Der umgekehrte Weg von SPS zum PC des Servicetechnikers funktioniert genauso. Durch diese Lösung bekommt das Fernwartungstool garnicht mit, daß zwischen dem Fernwartungs-PC und der SPS ein Netzwerk und ein weiterer Rechner hängt. Beide Fernwartungssysteme teilen sich so also eine physikalische Verbindung.

Kapitel 6

Abschlußbetrachtungen

Im Rahmen dieser Diplomarbeit wurden ein Linux-Treiber, eine portable Bildverarbeitungsbibliothek, eine Spundlocherkennung und eine Fernwartungsschnittstelle geplant, implementiert und untersucht.

Die Entwicklung des Linux-Treibers wurde eingestellt, da die dazugehörige Meßkarte nicht mehr benötigt wurde. Bis auf den Interrupt-Modus hatte der Treiber am Ende der Entwicklung einen stabilen Stand erreicht.

Der Aufwand für die Entwicklung der Bildverarbeitungsbibliothek hat sich bereits während dieser Diplomarbeit bezahlt gemacht. Durch das sorgfältige Design der Bibliothek war es möglich, in einer relativ kurzen Zeit eine sehr stabile und schnell laufende Spundlocherkennung auf Basis dieser Bibliothek zu implementieren.

Wenn zukünftig im Rahmen weiterer Diplomarbeiten und wissenschaftlicher Forschungsprojekte weitere Algorithmen und Filter in Form von Klassen für die Bibliothek implementiert werden, so wird die Bibliothek langfristig sicherlich den Einsatz von Testumgebungen wie Matlab überflüssig machen. Statt neue Konzepte zuerst mit Matlab zu testen, können diese dann mit dem gleichen Aufwand gleich in C++ realisiert werden. Dieses hat den Vorteil, daß C++ wesentlich schneller ist wie Matlab. Außerdem lassen sich die so entwickelten Programme gleich in der Praxis nutzen, ohne daß sie zuerst von Matlab nach C/C++ portiert werden müßten.

Die Spundlocherkennung lieferte am Ende der Diplomarbeit recht zufriedenstellende Ergebnisse. Auch der Projektpartner war mit der Qualität der Genauigkeit der Erkennung sehr zufrieden. Nichts desto trotz bietet gerade die Spundlocherkennung noch diverse Verbesserungsmöglichkeiten. So wurde z.B. der sehr einfache Sobel Kantenfilter verwendet. Hier wäre es sicherlich sehr interessant und sinnvoll, weitere bekannte Kantenfilter auf ihre Tauglichkeit für die Spundlocherkennung zu untersuchen.

Auch die Beleuchtung in der Anlage bietet noch einige Optimierungsmöglichkeiten. Es sollte z.B. untersucht werden, ob man mit zwei Lampen oder mit einer diffusen Beleuchtung nicht eine noch höhere Erkennungssicherheit erreichen könnte.

Das Konzept der Fernwartungsschnittstelle, ausgereifte und weit verbreitete Standardprotokolle und Standardprogramme zu verwenden, hat sich bei den Tests als richtig erwiesen. Dank der JPEG-Komprimierung der Bilder wird die Fernwartungsschnittstelle auch über Verbindungen mit niedrigen Bandbreiten in einer akzeptablen Geschwindigkeit arbeiten. Wünschenswert wäre zukünftig eine bessere Integration der SPS in die Fernwartungsschnittstelle. Insbesondere wäre es sicherlich sinnvoll, wenn der Anlage über die Fernwartungsschnittstelle des PCs bestimmte Befehle wie z.B. die Aufnahme einer Palette übermittelt werden könnten. Um die Wahl der richtigen Belichtungszeit für die Kamera zu erleichtern, wäre es eventuell sinnvoll, in die Fernwartungsschnittstelle ein Histogramm des aufgenommenen Originalbildes zu integrieren.

Insgesamt läßt sich feststellen, daß das Gesamtsystem einen Zustand erreicht hat, der eine Nutzung des Systems bei den Kunden der Firma Feige gestattet. Die für die Firma Feige verbleibenden Arbeiten sind

vor allem in dem Bereich der Systemadministration des PCs in der Anlage zu finden.

Literaturverzeichnis

[Ald92] Aldus Corporation. *TIFF Revision 6.0*, 1992.

[BBD+99] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Claus Schröter, und Dirk Verworrner. *Linux Kernelprogrammierung: Algorithmen und Strukturen des Version 2.2*. Addison-Wesley, fünfte Auflage, 1999.

[BSMM93] Ilja N. Bronstein, Konstantin A. Semendjajew, Gerhard Musiol, und Heiner Mühling. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 1993.

[CGI] *The Common Gateway Interface*.

[Cox00] Alan Cox. *Watchdog Timer Interfaces For The Linux Operating System*, 2000.

[Fla] Colin Flanagan. *Bresenham Line-Drawing Algorithm*.

[Gal95] Bill Gallmeister. *POSIX.4*. O'Reilly, 1995.

[GW92] Rafael C. Gonzalez und Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[Int96] Intel. *Intel MMX Technology - Overview*, 1996.

[Jäh97] Bernd Jähnke. *Digitale Bildverarbeitung*. Springer-Verlag, 1997.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library*. Addison-Wesley, 1999.

[Kir00] Schäfter + Kirchhoff. *Softwarepaket SK91PCI.LX*, 2000.

[Lak96] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[Lea92] V.F. Leavers. *Shape Detection in Computer Vision Using the Hough Transform*. Springer-Verlag, 1992.

[LH98] Steven R. Lerman und V. Judson Harward. *Introduction to Computation and Problem Solving Using Java - Lecture 26: Bresenham's Algorithm*, 1998.

[Mac99] Paul Mackerras. How to make sure your driver will work on the power macintosh. *Linux Magazine*, Juli 1999.

[Mar00] Jost J. Marchesi. Photographie digitalfoto kollegium: Lektion 20. *Photographie*, Mai 2000.

[Mey95] Scott Meyers. *Effektiv C++ programmieren*. Addison-Wesley, zweite Auflage, 1995.

[Mey97] Scott Meyers. *Mehr effektiv C++ programmieren*. Addison-Wesley, 1997.

[Mila] Todd Miller. *Circle Generating Algorithm*.

[Milb] Todd Miller. *Simple Polar-Coordinate Circle Generating Algorithm*.

[Phi98] Philips. *Data Sheet: SAA7146A - Multimedia bridge, high performance Scaler and PCI circuit (SPCI)*, 1998.

[Pos91] Jef Poskanzer. *pbm, pgm und ppm Manual Pages*, 1991.

[Pra91] William K. Pratt. *Digital image processing*. A Wiley-Interscience publication, zweite Auflage, 1991.

[Sen99] Sensoray. *Sensoray Model 626: Instruction Manual*, 1999.

[SG97] Abraham Silberschatz und Peter Galvin. *Operating system concepts*. Addison-Wesley, fünfte Auflage, 1997.

[Str98] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, dritte Auflage, 1998.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, vierte Auflage, 2000.

[Wie96] Klaus Wiehler. Bewegungsschätzung mittels bildverarbeitung zur nachführung von operationsmikroskopen in der augenheilkunde. Diplomarbeit, Technische Universität Hamburg-Harburg, 1996.

[Yod97] Victor Yodaiken. *The RT-Linux approach to hard real-time*, 1997.