Projektgruppe 292: *Ahelp*

Optischer Sensor zur Erkennung von medizinischen Notfällen im Wohnbereich

– Endbericht –

WS96/97–SS97

Lehrstühle I & VII
Fachbereich Informatik
Universität Dortmund

Autoren:

Andre Gronemeier
Lothar Grünz
Peter Imhoff
Martin Kolter
Christian Kopka
Sven Müller
Holger Netthöfel
Martin Stein
Harald Tillmann

Betreuer der Projektgruppe:

Dipl.-Inform. Markus Kohler
Dipl.-Inform. Michael Wittner

Inhaltsverzeichnis

1 Motivation **1**

1.1 Zielsetzung 1

1.2 Lösungsmöglichkeiten 2

1.3 Umsetzbarkeit 4

2 Grundlagen **5**

2.1 Argus 6

2.1.1 Was ist ARGUS ? 6

2.1.2 Das System ARGUS 6

2.1.3 ARGUSaugen - Die Bewegungserkennung von ARGUS 7

2.1.4 Unterschiede zwischen ARGUS und ***Ahelp*** 8

2.2 S-Pfinder 8

2.3 Kalman-Filter 10

2.3.1 Grundlagen 10

2.3.2 Anwendung beim Tracking 10

2.3.3 Verbesserungsmöglichkeiten 11

2.3.4 Implementierung in ***Ahelp*** 12

2.4 Fuzzy-Logik 13

2.4.1 Klassische Mengenlehre und Aussagenlogik 13

2.4.2 Fuzzy-Logik und Fuzzy-Mengen 14

2.4.3 Linguistische Variablen und Fuzzy-Regeln 17

3 Szenario **22**

3.1 Notfallsituationen 23

3.2 Normalsituationen 24

4 Design **27**

4.1 Entwurf 27

4.1.1 Analyse 27

4.1.2 Modularisierung 28

4.1.3 Spezifikation der Schnittstellen 29

4.2 Erweiterbarkeit 31

4.3 Basisklassen 32

4.3.1 TOKEN 32

4.3.2 HISTORY 35

4.3.3 LHISTORY 35

4.3.4 EVENT 35

4.3.5 INT_FUZZY_SET 35

4.3.6 DOUBLE_FUZZY_SET 36

4.4 Softwaretools 37

4.4.1 Motivation 37

4.4.2 BONSAI 37

4.4.3 Allgemeines über BONSAI 38

4.5 Technik 39

4.5.1 Programmiersprache 39

4.5.2 Verwendete Hardware 39

4.5.3 Sonstige Technik 39

5 Implementierung **41**

5.1 Die Modellierung des Raums 41

5.1.1 ROOM 42

5.1.2 MODEL . . . 42
5.1.3 CAMERA . . . 43
5.1.4 OBJECT . . . 43
5.1.5 RECTANGLE . . . 44
5.1.6 TRIANGLE . . . 44
5.1.7 CIRCLE . . . 44
5.1.8 DOOR . . . 45
5.1.9 WINDOW . . . 45
5.1.10 SQUARE . . . 45
5.2 Die Bewegungserkennung von ***Ahelp*** . . . 46
5.2.1 Faster_CreateBinaryMatrix . . . 46
5.2.2 FindBody . . . 48
5.2.3 Create_HeadSearchArea . . . 48
5.2.4 FindHead . . . 49
5.2.5 CreateHeadAndBodyCenter . . . 49
5.2.6 Schätzung von Pixelpositionen (KALMAN2D) . . . 49
5.3 Die Analyse der menschlichen Bewegung durch das Objekt ***HUMAN*** . . . 50
5.3.1 Aufgabenstellung . . . 50
5.3.2 Designideen . . . 51
5.3.3 HUMAN_ACTION . . . 58
5.4 Generierung des Alarmstatus . . . 59

6 Ablauf **64**
6.1 Teilnehmer und Zeitraum . . . 64
6.2 Projektphasen . . . 64
6.3 Probleme . . . 65
6.4 Organisation . . . 66

7 Vergleich **67**

8 Ausblick **68**

8.1 Soziale Problematik der Kameraüberwachung 68

8.2 Sprachsteuerung 68

A Anhang und Grundlagen **70**

A.1 Schnittstellen 70

A.1.1 INT_FUZZY_SET 70

A.1.2 DOUBLE_FUZZY_SET 75

A.1.3 VECTOR2D 81

A.1.4 VECTOR3D 85

A.1.5 MATRIX2D 90

A.1.6 MATRIX3D 94

A.1.7 TIME 98

A.1.8 TIMER 103

A.1.9 CALC 105

A.1.10 PIC_MANAGER 109

A.1.11 METEOR 112

A.1.12 PERSON_CALC 114

A.1.13 KALMAN2D 115

A.1.14 ROOM 116

A.1.15 MODEL 119

A.1.16 CAMERA 127

A.1.17 OBJECT 130

A.1.18 RECTANGLE 132

A.1.19 TRIANGLE 133

A.1.20 CIRCLE 134

A.1.21 DOOR 135

A.1.22 WINDOW 137

A.1.23 SQUARE 139

A.1.24 SURVEILLANCE 142
A.1.25 ALARM 145
A.1.26 HUMAN 148
A.1.27 HUMAN_ACTION 155

B Theoretische Grundlagen **157**
B.1 Transformation von Bildkoordinaten in Raumkoordinaten . . . 157
B.1.1 Definition des räumlichen Koordinatensystems 157
B.1.2 Verwendete Transformationsmethode 158
B.2 Grundlagen der Bildverarbeitung 162
B.2.1 Farbmodelle 162
B.2.2 Methoden der Bildverbesserung 163
B.2.3 Bewegungserkennung 164

Index **167**

Literaturliste **175**

Kapitel 1

Motivation

1.1 Zielsetzung

Wenn schon die Renten nicht sicher sind, so sollte doch wenigstens das Leben der Senioren möglichst sicher sein. Ein Schritt in diese Richtung ist die Entwicklung eines Hausnotrufsystems, mit dessen Hilfe bei unerwarteten medizinischen Notfällen in der Wohnung (z.B. Herzinfarkt oder Schlaganfall) schnell und möglichst zuverlässig Hilfe herbeigeholt wird.

Die Wichtigkeit eines solchen Systems gewinnt in Anbetracht der starken Zunahme des Anteils von alten Menschen an der Bevölkerung der westlichen Industriestaaten zusätzlich an Bedeutung. Plätze in guten Altersheimen sind rar und oft unerschwinglich. Parallel dazu haben viele Senioren ohnehin den Wunsch, auch im Alter in der gewohnten Umgebung wohnen bleiben zu können. Auf der anderen Seite besteht die Angst und die Gefahr, nach einem medizinischen Notfall in den eigenen vier Wänden hilflos liegen zu bleiben. In einer ähnlichen Situation befinden sich natürlich auch jüngere medizinisch gefährdete Menschen, die im Zeichen der Single-Gesellschaft lieber alleine leben.

Es besteht also ein dringender Bedarf an einem Notrufsystem,

1. das dem zu überwachenden Menschen Sicherheit suggeriert und
2. das auch tatsächlich sicher ist.

Dazu ist es unter anderem notwendig,

- daß ein erforderlicher Notruf auch und gerade dann gesendet wird, wenn die Person hilflos ist,
- daß die Überwachung "rund um die Uhr" und in der gesamten Wohnung realisiert wird und
- daß die Systemfunktionalität auch von nicht technisch versierten Menschen nachvollzogen werden kann.

1.2 Lösungsmöglichkeiten

Eine mögliche Realisierung eines Notrufsystems läßt sich zunächst danach unterscheiden, ob die Auslösung des Alarms aktiv durch die zu überwachende Person oder passiv durch das System selbst erfolgt.

Eine Form des aktiven Notrufsystems sind die bereits existierenden Hausnotrufsysteme mit Taster und Funkgerät. Solche Systeme haben jedoch zwei massive Nachteile:

1. Der Taster muß zur Alarmauslösung immer mitgeführt werden. Neben der Lästigkeit dieses Zwangs besteht dann auch immer die Gefahr, daß der Taster vergessen wird oder aber (wie zum Beispiel in der Badewanne) prinzipiell nicht mitgeführt werden kann.
2. Die aktive Alarmauslösung ist gerade bei schweren Unfällen, die Bewußtlosigkeit oder geistige Umnachtung zur Folge haben, nicht mehr möglich.

Der erstgenannte Nachteil ließe sich durch ein System mit Mikrophon beheben. Der Notruf könnte dann etwa durch einen Hilferuf ausgelöst werden. Den zweitgenannten und entscheidenden Nachteil gäbe es jedoch auch noch bei solch einem System. Aktive Systeme können deshalb den Anforderungen eines qualitativ hochwertigen Hausnotrufsystems nicht gerecht werden.

Der Wunsch einserseits nach einem von Zeitpunkt, Ort und Art des Notfalls unabhängigen und andererseits nach einem bequemen System führt zur Idee

von passiven Systemen. Diese sollten medizinische Notfälle selbständig erkennen können und dann auch das Alarmsignal selbständig abschicken können. Desweiteren wird von solchen Systemen ein Maximum an Sicherheit der Notfallerkennung verlangt werden.

Eine naheliegende und in ***Ahelp*** realisierte Form eines passiven Systems besteht in der digitalen Auswertung der Videosignale von in der jeweiligen Wohnung aufgestellten Kameras. Die Auswertung folgt dabei folgendem Ablauf:

- Aus dem Kamerabild wird zunächst mittels Techniken der digitalen Bildverarbeitung der zu überwachende Mensch extrahiert. (Die dazu für ***Ahelp*** entwickelten Algorithmen sind im Kapitel "Bildverarbeitung" genau beschrieben. Ähnliche Funktionen erfüllt auch der am MIT entwickelte Ansatz des "Pfinder".).
- Aus diesen Daten wird die dreidimensionale Bewegung des Menschen berechnet.
- Dieser wiederum werden Aktionen (z.B. Gehen, Fallen, Bewegungslosigkeit) zugeordnet. Bei Aktionsfolgen, die in Kombination mit der Realzeit auf einen Notfall schließen lassen, wird ein Alarm ausgelöst.

Zur Steigerung der Sicherheit des Systems könnten ergänzend folgende Techniken angewendet werden:

- Der zu überwachende Raum wird rechnerintern dreidimensional modelliert. Die Interpretation der durch die Bildverarbeitung gefundenen Bewegungen erfolgt dann durch Kombination mit dem Raummodell. (Diese Technik wird auch in ***Ahelp*** angewendet.)
- Mit Hilfe der Techniken der künstlichen Intelligenz kann ein adaptives System entwickelt werden. Individuelle Verhaltensmuster werden dann zur Laufzeit vom System erlernt.

Auch nichtoptische passive Systeme (z.B. in den Fußboden eingelassene Belastungsensoren) sind prinzipiell denkbar.

Für die Praxis interessant sind natürlich auch Kombinationen der genannten Systeme.

1.3 Umsetzbarkeit

Leider gibt es ein Fülle von Schwierigkeiten, die einer praktischen Umsetzung eines Notfallerkennungssystems widersprechen.

Aktive Systeme kommen (zumindest in ihrer Ausschließlichkeit) bereits wegen der oben genannten massiven Lücken nicht in Frage.

Aber auch passive Systeme wie ***Ahelp*** könnten aus einer Vielzahl von Gründen (zumindest zum jetzigen Zeitpunkt) nicht umsetzbar sein:

- Das System ist zu teuer. (z.B. Kameras, Sensoren, Rechner)
- Das System benötigt zu viel Platz. (z.B. Kameras und ihre Stative)
- Die Auswertung ist für ein zwangsläufig zumindest annähernd echtzeittaugliches System zu langsam.
- Die Auswertung ist zu unsicher. Das heißt es werden zuviele Fehlalarme ausgelöst oder aber nicht alle tatsächlichen Unfälle erkannt.
- Die Überwachungsmethoden werden aus psychologischen oder ethischen Gründen nicht akzeptiert. Eventuell entsteht beim potentiellen Anwender zum Beispiel kein Vertrauen in das System. Schwerer wiegen dürfte allerdings die Angst vor Verletzung der Intimsphäre. Man denke nur an das Horrorszenario eines Überwachungsstaats a la George Orwell.

Bis auf den letzten Punkt handelt es sich bei diesen Problemen jedoch nur um technische Realisierungsschwierigkeiten und nicht um prinzipiell unvermeidbare Verhinderungsgründe. Die rasante Technologieentwicklung läßt im übrigen auf ein stetiges Zurückgehen des Gewichts dieser Schwierigkeiten hoffen. Bereits die in der Projektgruppe realisierte erste Version von ***Ahelp*** erreicht zum Beispiel mit vergleichsweise einfacher Hardware eine Echtzeittauglichkeit auf halbwegs sicherem Niveau.

Bezüglich der Akzeptanz eines solchen Notrufsystems bedarf es natürlich noch entsprechender empirischer Untersuchungen und Pilotversuche.

Kapitel 2

Grundlagen

Ahelp besteht aus zwei wesentlichen Komponenten. Zunächst versucht die Bildverarbeitung auf Basis der Kameraparameter eine möglichst genaue Einschätzung des Standorts der Zielperson zu liefern. Anschließend erfolgt in den nachgelagerten Modulen eine Auswertung dieser Daten.
Ahelp ist nicht das einzige System, das sich mit der Interpretation einer Person im Raum beschäftigt. So versuchte bereits die Projektgruppe ARGUS einzelne Körperteile zu extrahieren und zu bewerten. Desweiteren werden natürlich seit vielen Jahren in den verschiedensten Firmen und Institutionen Forschungen auf dem Gebiet der Erkennung bewegter Objekte betrieben. Ein Projekt, daß auch in der Öffentlichkeit Beachtung fand, ist das vom MIT entwickelte System Pfinder. Die Kapitel 2.1 sowie 2.2 stellen beide Systeme kurz vor.
Die Auswertung realer Daten über ein Bildverarbeitungssystem ist prinzipiell mit Fehlern und Unsicherheiten behaftet. Daher enthält ***Ahelp*** Methoden, um diese Fehler möglichst zu minimieren. Bereits innerhalb des Moduls Bildverarbeitung wird der Kalmanfilter zur Schätzung zukünftiger Positionen eingesetzt. Die Auswertung der Bilddaten erfolgt dann mit Hilfe von Methoden der Fuzzy-Logik. Diese Systeme, und deren Bedeutung für ***Ahelp*** , werden in den nachfolgenden Kapiteln 2.3 und 2.4 erläutert.

2.1 Argus

2.1.1 Was ist ARGUS ?

ARGUS[27796] ist ein Dialogsystem zur Steuerung von Heimgeräten im Haushalt. Ziel ist es, Handgesten einer frei im Raum befindlichen Person zu interpretieren und in Steuerungsbefehle für verschiedene Geräte (Fernseher, Lichtschalter) umzusetzen. Somit könnte das System als komfortabler Ersatz für eine Fernbedienung dienen oder auch körperbehinderten Menschen mit eingeschränktem Bewegungsspielraum das Leben erleichtern.
ARGUS wurde vom Wintersemester 1995/96 bis Sommersemester 1996 von der PG 277 entwickelt. Der entwickelte Prototyp erlaubt es, eine Person im Raum und deren Handgeste zu erkennen. Der Aktionsradius dieser Person ist durch die Verwendung statischer Kameras beschränkt, die Person kann sich also nicht frei im Raum bewegen. Die Richtung der Hand und damit das ausgewählte Gerät wird von ARGUS ermittelt. Für die Erkennung der Geste wird dann auf die Projektgruppe ZYKLOP [24795] verwiesen.

2.1.2 Das System ARGUS

Abbildung 2.1 zeigt den Ablauf des ARGUS-Systems.

Bei Systemstart ist der Trackingalgorithmus deaktiviert. Daher werden von beiden Kameras die aktuellen Bilddaten ausgelesen.
Zur Bewegungserkennung wird das Differenzbildverfahren verwendet. Bei erkannter Bewegung werden die Hände und der Kopf gesucht. Dieses Verfahren wird solange durchlaufen, bis die Suche erfolgreich war. Konnten die Körperteile extrahiert werden, wird der Trackingalgorithmus eingeschaltet. Beim nächsten Durchgang werden dann nur noch Bilddaten einer Kamera benötigt.
Die extrahierten Hände der Zielperson werden an ein Teilsystem der Projektgruppe ZYKLOP übertragen. Dieses liefert im Erfolgsfall für eine Hand die Art, sowie Richtung und Position der Geste. Die Kamera, in der diese Handgeste gefunden wurde wird dann im weiteren Verlauf für die Datengewinnung verwendet.
Die Richtung und Position der Geste wird von ARGUS verwendet, um die Position der Hand im Raum zu berechnen. Diese Position liefert das ausgewählte Gerät. Die Geste selbst wird dann zur Steuerung verwendet.

Der verwendete Trackingalgorithmus wird deaktiviert, wenn eine Handgeste im weiteren Verlauf wieder verloren wurde oder der Benutzer den Gestendialog abbricht.

2.1.3 ARGUSaugen - Die Bewegungserkennung von ARGUS

Die Bewegungserkennung von ARGUS umfaßt das Finden der Körperteile im zweidimensionalen Bildraum der angeschlossenen Kameras.
Ziel ist es, die Hände und den Kopf der Zielperson zu ermitteln. Menschliche Haut ist durch einen relativ hohen Rotanteil gekennzeichnet. Zur Datengewinnung wird daher lediglich der V-Kanal verwendet. Dieser repräsentiert die Rot-Grün-Achse des PAL-Standards und ist zur Filterung roter Bildbereiche besonders geeignet. Die Häufigkeitsverteilung der Rotanteile wird durch eine Histogrammberechnung auf den Differenzbildpunkten ermittelt. Bewegungspunkte mit einem hohen Rotanteil sind dan ein Zeichen für bewegte Körperteile.
Zur Ermittlung zusammenhängender Flächen kamen bei ARGUS zwei Verfahren zum Einsatz. Zunächst wurde der Motion-To-Shape-Sensor von Hans Röttger verwendet [Röt95]. Dieser erwies sich in der Praxis als zu langsam, um die angestrebte Bildrate von mindestens 10 Bildern pro Sekunde zu erreichen. Das daraufhin entwickelte sog. Run-Length-Code-Verfahren lieferte dann wesentlich bessere Ergebnisse.
Die Zuordnung der gefundenen Flächen erfolgt mit einer einfachen Heuristik. So konnte etwa davon ausgegangen werden, daß die Person steht, der Kopf also stets die oberste der gefundenen Flächen ist.
Der verwendete Trackingalgorithmus bedeutet eine wesentliche Steigerung der Geschwindigkeit. Bei aktiviertem Tracking wird die gefundene Hand verfolgt. Es ist dann nur noch notwendig, einen Bildausschnitt zu betrachten, was besonders der Differenzbildberechnung zugute kommt. Vorraussetzung hierfür ist die Möglichkeit, ausgehend von einer vorhandenen Position die zukünftige Position schätzen zu können. Um dieses Ziel zu erreichen wurde, ebenso wie bei ***Ahelp*** , der Kalmanfilter verwendet.

2.1.4 Unterschiede zwischen ARGUS und *Ahelp*

Die Bewegungserkennung von ARGUS und ***Ahelp*** gehen von ähnlichen Voraussetzungen aus. In beiden Fällen sollen signifikante Körperteile gefunden werden. In ARGUS sind dies Hände und Kopf der Person. In ***Ahelp*** dagegen sollen Kopf und Oberkörper gefunden werden. Trotz ähnlicher Startbedingungen sind die Unterschiede im Ablauf beider Systeme so gravierend, daß ein direkter Einsatz der Bewegungserkennung von ARGUS in ***Ahelp*** nicht möglich war.
Ziel von ***Ahelp*** ist es, neben dem Kopf der Zielperson auch den Oberkörper zu erkennen. An dieser Stelle ist das alleinige Betrachten roter Bewegungspunkte nicht mehr möglich. Es mußte daher neben dem V-Kanal auch der U-Kanal betrachtet werden. Desweiteren ist es für eine sichere Berechnung der dreidimensionalen Positionen nötig, von beiden Kameras Positionsangaben zu erhalten. Die zu betrachtende Bildmenge erhöht sich damit bei jedem Arbeitsdurchgang auf acht Bilder gegenüber zwei (max. vier) Bilder bei ARGUS. Es mußten daher neue Möglichkeiten entwickelt werden, um die Arbeitsgeschwindigkeit zu erhöhen.
Für die Zuordnung der bewegten Flächen zu den gesuchten Körperteilen, wurde bei ARGUS eine Heuristik verwendet, die von der Voraussetzung ausgeht, daß die Person steht und der Kamera zugewandt agiert. Diese Annahmen können bei einer sich frei im Raum bewegenden Person nicht mehr getroffen werden. Besonders in Notfallsituationen, etwa einem Sturz, wären Annahmen wie “Der Kopf ist stets obenünsinnig. ***Ahelp*** benötigte daher eigene Strategien, um Körperteile bewegten Flächen zuzuordnen.
Eher kosmetischer Natur sind die Unterschiede in der Entwicklungsumgebung beider Systeme. Für ARGUS wurde eine C-Umgebung unter DOS gewählt, während ***Ahelp*** unter UNIX in C++ entwickelt wurde.
Die Gesamtheit der Unterschiede führte daher zu der Entscheidung, selbst eine Bewegungserkennung zu entwickeln und nicht auf die vorhandene ARGUS-Umgebung aufzusetzen.

2.2 S-Pfinder

Der S-Pfinder (”stereo Person finder”) wurde am MIT entwickelt und ist ein System zur Verfolgung menschlicher Bewegung, das aus 2D-Objekten die

Bewegung und Form von 3D-Objekten schätzt. Es benutzt 2D Musterklassifizierungstechniken zur Verfolgung von "Blasen"-Merkmalen in Videobildern. Der S-Pfinder wurde auf einer SGI-Indy-Workstation mit zwei Videokameras implementiert und besitzt eine Reihe von Applikationen, die auf dem System aufbauen.

Es folgt eine kurze Beschreibung der Arbeitsweise des S-Pfinders:

1. Initialisierung

   (a) Das System lernt die Szenerie ohne eine Person. Dabei werden sowohl die Farbmodelle in YUV- als auch in der normierten U^*V^*-Darstellung gespeichert. Es gelten $U^* = U/Y$ und $V^* = V/Y$. Die normierte Darstellung dient zur Kompensation des Schattenwurfes. Tritt eine Person in den Raum, wird ein Personenmodell gebildet.

   (b) Finden der Person. Die Person wird anhand einer großen farblichen Veränderung gefunden. Im laufe der Zeit wird ein Multi-Blasen-Modell gebildet. Die Entwicklung des Modells wird durch die Verteilung der Farben der erkannten Kontur gebildet, wobei Blasen für jede unterschiedliche Farbregion hinzukommen. Normalerweise werden seperate Blasen für die Hände, den Kopf, die Füße, Oberkörper und die Beine benötigt.

2. Tracking

   Ist das Personen- und das Hintergrund-Modell entwickelt, kann man sich den nächsten Bildern zuwenden, sie interpretieren und die beiden Modelle aktuallisieren. Um dies zu erreichen, sind verschiedene Schritte notwendig:

   (a) Schätzung der neuen Position anhand des Flächenmodells mittels Kalman Filter.

   (b) Bemessung der Wahrscheinlichkeit für jeden Punkt, ob es zum Blasen- oder Raum-Modell gehöhrt..

   (c) Korrektur der Supportmap. Bei der Supportmap handelt es sich um eine Liste, die für jedes Pixel angibt, ob es zu einer der Blasen, oder zur Szenerie gehöhrt.

(d) Aktualisierung des Personenmodells und des Raummodells.

Der S-Pfinder arbeitet in Echtzeit mit ca. 10-30 Hz. Er war zum Zeitpunkt der Entwicklung unseres Systems nur auf der SGI lauffähig, so daß wir uns nicht näher mit ihm beschäftigten. Es ist jedoch eine Lizenz und der Quellcode am LS 7 vorhanden.

Quellen: [AP96] , [AWP96] , [AWDP96]

2.3 Kalman-Filter

2.3.1 Grundlagen

R.E. Kalman entwickelte 1960 den nach ihm benannten mathematischen Filter. Er dient zur Verbesserung der Vorhersage des zukünftigen Verhaltens allgemeiner dynamischer Systeme. Berücksichtigt wird dabei zum einen die Historie des Systems und zum zweiten der Grad des Rauschens der Dynamik und der Grad der Messungenauigkeit.

Anwendung findet der Kalman-Filter beispielsweise bei militärischen Navigationssystemen und in den Wirtschaftswissenschaften.

Die Bestandteile des Kalman-Filters sind zum einen der Statusvektor und zum anderen diverse Kovarianzmatrizen. Im Statusvektor werden die aktuellen Werte der relevanten Variablen des jeweiligen Systems oder Sachverhalts festgehalten. In den Matrizen stehen zum einen die Erwartungswerte von Rauschvorgängen und Messfehlern und zum anderen die Abhängigkeiten dieser Werte untereinander. Darüberhinaus werden in einer internen Matrix Werte der Historie festgehalten.
Die Untersuchung und Vorhersage des Systems erfolgt zeitdiskret durch Iterationsschritte. Bei jedem dieser Iterationsschritte werden der Statusvektor und teilweise die Matrizen neu berechnet.

2.3.2 Anwendung beim Tracking

In bewegtbildverarbeitenden Systemen kann der Kalman-Filter zur Vorhersage zukünftiger Positionen von zu verfolgenden (zu “trackenden”) Objekten

angewendet werden. In ***Ahelp*** handelt es sich bei diesen Objekten um Merkmalspunkte (Kopf und Schwerpunkt des Oberkörpers) von durch Kameras aufgenommenen Menschen.

Im Statusvektor müssen Informationen über die aktuelle Situation des jeweiligen Merkmalspunkts abgelegt werden. In der Bewegtbildverarbeitung sind dies Position und beliebig viele Ableitungen nach der Zeit. In der einfachsten Form reicht Position und Geschwindigkeit (erste Ableitung). Die Beschleunigung (zweite Ableitung) wird dann (obwohl dies von der Physik betrachtet inkorrekt ist) als weißes Rauschen modelliert.

Position und Geschwindigkeit sind im vorliegenden Fall natürlich zweidimensional. Die beiden Dimensionen werden unabhängig voneinander aber unter gleichen Voraussetzungen betrachtet. Die fehlenden Korrelationen führen in den Kovarianzmatrizen an den entsprechenden Stellen zu Nullen. Die Erwartungswerte des Rauschens und des Messfehlers erhalten in beiden Dimensionen die gleichen Werte.

2.3.3 Verbesserungsmöglichkeiten

Sollte sich der in ***Ahelp*** realisierte Kalman-Filter in der Anwendung als zu ungenau erweisen, bieten sich zwei Verbesserungsmöglichkeiten an.

1. Initialisierung
   Bevor die Kalman-Iteration durchgeführt werden kann, müssen die Matrizen initialisiert werden. Für ***Ahelp*** wurden zunächst ungetestet die Initialisierungswerte aus der PG "ARGUS" übernommen. Andere Initialisierungswerte könnten allerdings zu besseren Ergebnissen führen. Insbesondere die Werte für den Grad des Rauschens und des Messfehlers sind wichtig, da diese während der Kalman-Iteration nicht verändert werden.

2. Verbesserung der Modellierung der Beschleunigung
   Wie oben beschrieben kann die Beschleunigung in einem einfachen Bewegungsmodell als weißes Rauschen modelliert werden. In der Realität zeigt aber auch die zweite Ableitung nach der Zeit kontinuierliches Verhalten. Eine entsprechende Erweiterung von Statusvektor und Matrizen

käme der Realität näher und würde zu einer Verbesserung der Prädiktion führen. In diesem Fall würde erst die dritte Ableitung als weißes Rauschen modelliert werden.

2.3.4 Implementierung in *Ahelp*

In ***Ahelp*** hätte der Kalman-Filter an zwei Stellen eingesetzt werden können:

1. Unterstützung des Tracking im zweidimensionalen Bild durch Vorhersage der zukünftigen Position zu extrahierender Merkmalspunkte
2. Unterstützung der Menschmodellierung durch Vorhersage der zukünftigen Position des zu überwachenden Menschen

Auf Grund der möglichen Umrechnung zwischen Kamerapositionen und realen Positionen im Raum handelt es sich dabei allerdings um redundante Anwendungen. Sowohl funktionale als auch rechenzeitbezogene Gründe sprachen für eine Realisierung in der Bildverarbeitung.

In ***Ahelp*** wird für jede Kombination aus Kamera und Objektpunkt (Kopf und Oberkörper) eine Instanz der Klasse "KALMAN-2D" angelegt. Private Daten sind der Statusvektor und diverse Parameter. Der Iterationsfunktion wird die zuletzt bestimmte Position übergeben. Es folgt die der Iterationsvorschrift gemäße Aktualisierung der privaten Daten. Rückgabewert ist die prädizierte Position.

In ***Ahelp*** konnte die Iterationsvorschrift auf Grund des physikalischen Hintergrunds (z.B. fehlende Kovarianzen) erheblich vereinfacht werden. Es werden lediglich die Weg- und Geschwindigkeitsvarianzen und die Weg- und Geschwindigkeitskovarianz verwendet.

$$p = \begin{pmatrix} x_{Pixel} \\ y_{pixel} \end{pmatrix} \tag{2.1}$$

$$v = \begin{pmatrix} vx \\ vy \end{pmatrix} \tag{2.2}$$

$$p_{neu} = p_{alt} + (p_{mess} - p_{alt}) \frac{ss}{r + ss} + v_{alt} \Delta t \frac{sv}{r + ss} \tag{2.3}$$

$$v_{neu} = v_{alt} + (p_{mess} - p_{alt}) \frac{sv}{r + ss} \tag{2.4}$$

Varianzen- und Kovarianzenauffrischung

$$ss = r \frac{ss}{r + ss} + 2\Delta t r \frac{sv}{r + ss} + \Delta t^2 \left(vv - \frac{sv^2}{r + ss} \right) + \Delta t a \tag{2.5}$$

$$vv = \left(vv - \frac{sv^2}{r + ss} \right) + \Delta t^3 \frac{a}{3} \tag{2.6}$$

$$sv = \frac{rsv}{r + ss} + \Delta t \left(vv - \frac{sv^2}{r + ss} \right) + \Delta t \frac{a}{2} \tag{2.7}$$

Dies lief zwar auf Kosten der Nachvollziehbarkeit des Quellcodes, war jedoch im Hinblick auf die Echtzeittauglichkeit des Systems anzuraten.

2.4 Fuzzy-Logik

2.4.1 Klassische Mengenlehre und Aussagenlogik

Eine Menge ist eine Zusammenfassung von verschiedenen Elementen (aus einer gegebenen Grundgesamtheit) zu einem Ganzen. Für jedes Element x aus der Grundgesamtheit U und eine Menge A ist genau eine der folgenden Aussagen erfüllt:

- $x \in A : x$ ist in A enthalten.
- $x \notin A : x$ ist nicht in A enthalten.

Mit diesen beiden Aussagen und der üblichen Aussagenlogik lassen sich die bekannten Mengenoperationen Durchschnitt $\cap$, Vereinigung $\cup$ und Komplement ^{C} definieren:

- $(x \in A \cap B) \Leftrightarrow (x \in A \wedge x \in B)$
- $(x \in A \cup B) \Leftrightarrow (x \in A \vee x \in B)$
- $(x \in A^C) \Leftrightarrow (x \notin A)$

Dies verdeutlicht den engen Zusammenhang zwischen Mengenlehre und Aussagenlogik.

Fuzzy-Logik ist eine Verallgemeinerung der Aussagenlogik auf eine vielwertige Logik und eine Fuzzy-Menge ist der daraus resultierende Mengenbegriff.

2.4.2 Fuzzy-Logik und Fuzzy-Mengen

In einer vielwertigen Logik existieren nicht nur die beiden Wahrheitswerte "wahr" und "falsch", sondern beliebige Abstufungen von Wahrheit. Beispielsweise kann man Wahrheitswerte aus dem Intervall [0..1] wählen. 0 entspricht hierbei dem üblichen "falsch", 1 entspricht "wahr". Allerdings sind auch beliebige Zwischenwerte zulässig.

Betrachtet man in einer derartigen Logik die Aussage $x \in M$, so erkennt man, daß ein Element nun zu einem beliebigen Grad zu einer Menge gehören kann. Formal kann man eine derartige Menge folgendermaßen beschreiben:

Definition 2.4.1 (Fuzzy-Menge) *Sei X eine beliebige Menge. Eine Fuzzy-Menge auf der Grundmenge X ist eine Menge*

$$A = \{(x, \mu_A(x)) | x \in X\}$$

mit

$$\mu_A : X \longrightarrow [0, 1]$$

$\mu_A(x)$ heißt Zugehörigkeitsfunktion von A.

Die Zugehörigkeitsfunktion gibt für ein Element aus der Grundmenge an, zu welchem Grad es zu einer Menge gehört. Der klassische Mengenbegriff ist in dieser Definition als Spezialfall enthalten ($\mu(x) = 0$ oder $\mu(x) = 1$ für alle $x \in X$).

Die folgenden Definitionen sind Verallgemeinerungen üblicher Mengeneigenschaften auf Fuzzy-Mengen:

Zwei Fuzzy-Mengen A und B auf der Grundmenge X sind gleich, wenn für alle $x \in X$ $\mu_A(x) = \mu_B(x)$ gilt. Die *leere Fuzzy-Menge* auf X ist die Menge, bei der für alle Zugehörigkeitsgrade $\mu(x) = 0$ gilt, für die *Universalmenge* gilt entsprechend $\mu(x) = 1$ für alle $x \in X$. A ist *Teilmenge* von B ($A \subseteq B$), wenn für alle $x \in X$ die Bedingung $\mu_A(x) \leq \mu_B(x)$ erfüllt ist. Wenn sogar $\mu_A(x) < \mu_B(x)$ gilt, so ist A *echte Teilmenge* von B ($A \subset B$). Die *Stützmenge* (oder *Trägermenge,Support*) $S(A)$ einer Fuzzy-Menge A sind diejenigen $x \in X$, deren Zugehörigkeitsgrad größer als Null ist, also

$$x \in S(A) :\Leftrightarrow \mu_A(x) > 0$$

Die kleinste obere Grenze von $\mu_A(x)$ heißt *Höhe* $hgt(A)$ der Menge A, also ist

$$hgt(A) = sup_{x \in X}\{\mu_A(x)\}$$

Der *Betrag* einer endlichen Fuzzy-Menge A ist als Summe

$$|A| := \sum_{x \in X} \mu_A(x)$$

definiert. Der Betrag von gewöhnlichen Mengen ist in dieser Definition wieder als Spezialfall enthalten.

Da Fuzzy-Mengen ohne die zugehörigen Mengenoperatoren nicht besonders spannend oder sinnvoll sind, sollte man auch nach entsprechenden Verallgemeinerungen von Durchschnitt, Vereinigung und Komplement suchen. Analog zu den klassischen Mengenoperatoren, kann man die Mengenoperatoren aus verallgemeinerten logischen Operatoren ableiten. Eine Möglichkeit, diese Operatoren zu definieren ist, axiomatisch Forderungen aufzustellen und nach Realisierungen dieser Axiome zu suchen. Sinnvoll sind sicher folgende Forderungen:

- Für die Menge $0, 1$ sollten die verallgemeinerten logischen Verknüpfungen den üblichen Verknüpfungen $\wedge$, $\vee$ und $\neg$ entsprechen.
- $A \vee \neg A = 1$, d.h. die Oder-Verknüpfung aus A und dem Komplement von A ist immer völlig wahr.
- Die Verknüpfungen sollten assoziativ und kommutativ sein.

Eine mögliche Realisierung dieser Axiome ist folgende Auswahl von Verknüpfungen:

- $A \wedge B :\Leftrightarrow min(A, B)$
- $A \vee B :\Leftrightarrow max(A, B)$
- $\neg A :\Leftrightarrow 1 - A$

Man kann leicht Nachrechnen, daß für diese Auswahl sogar das De Morgan Gesetz gilt. Dies untermauert, daß *min* und *max* eine sinnvolle Verallgemeinerung der logischen Verknüpfungen "und" und "oder" sind. (Es gibt viele weitere sinnvolle Verallgemeinerungen der logischen Verknüpfungen. Da sich *min* und *max* besonders leicht und schnell berechnen lassen, werden diese Verknüpfungen aber besonders häufig benutzt.)

Die zugehörigen Mengenoperatoren sind damit folgendermaßen definiert:

Definition 2.4.2 *Der Durchschnitt von zwei Fuzzy-Mengen A und B ist folgendermaßen definiert:*

$$A \cap B =: C \text{ mit } \mu_C(x) := min\{\mu_A(x), \mu_B(x)\}, x \in X$$

Für die Vereinigung von Fuzzy Mengen gilt:

$$A \cup B =: C \text{ mit } \mu_C(x) := max\{\mu_A(x), \mu_B(x)\}, x \in X$$

und das Komplement einer Fuzzy-Menge ist:

$$\overline{A} =: C \text{ mit } \mu_C(x) := 1 - \mu_A(x), x \in X$$

Man kann leicht nachrechnen, daß diese Operatoren für den Spezialfall scharfer Mengen den üblichen Mengenoperatoren entsprechen.

Analog hierzu lassen sich natürlich auch andere logische Operatoren zu Mengenoperatoren fortsetzen. Die Auswahl entsprechender Operatoren ist hierbei durch die Anwendung bestimmt und läßt sich oft nur experimentell ermitteln. Beispielsweise hat der *min*-Operator folgende Schwäche: Er ist oft zu pessimistisch. Das Ergebnis wird allein durch den minimalen Wahrheitswert bestimmt. Auch durch weitere Verknüpfung mit höheren Wahrheitswerten läßt sich das Ergebnis nicht mehr verändern. Umgekehrt ist der *max*-Operator zu optimistisch. Das Ergebnis wird allein durch das Maximum festgelegt.

Im Gegensatz dazu wird bei *kompensatorischen Operatoren* das Verknüpfungsergebnis durch alle beteiligten Wahrheitswerte bestimmt. Ein Beispiel für einen derartigen Operator ist der Durchschnitt von Wahrheitswerten:

$$avg(A, B) := \frac{A + B}{2}$$

oder allgemeiner der gewichtete Durchschnitt beliebig vieler Wahrheitswerte $A_1 \ldots A_n$:

$$avg(A_i, i = 1 \ldots n) := \frac{\sum_{i=1}^{n} a_i A_i}{\sum_{i=1}^{n} a_i} \ , \ a_i \in \Re$$

Natürlich läßt sich genau wie bei *min* und *max* hieraus wieder ein Mengenoperator für Fuzzy-Mengen ableiten.

2.4.3 Linguistische Variablen und Fuzzy-Regeln

Will man beschreiben, wie man aus Beobachtungen Schlußfolgerungen ziehen kann, so tut man dies meistens umgangssprachlich. Beispiel:

> "Ein Sturz ist eine schnelle Abwärtsbewegung."

Die Daten, aus denen ein Computer "Schlußfolgerungen" ziehen soll, sind in unserem Fall numerische Werte wie Raumpositionen und Geschwindigkeitsvektoren. Die Lücke zwischen diesen beiden Extremen läßt sich durch Fuzzy-Logik schließen:

Begriffe wie "schnell" und "abwärts" haben einen direkten Bezug zu den numerischen Meßwerten. "Schnell" bezieht sich auf einen bestimmten Wertebereich des Betrags eines Geschwindigkeitsvektors, "abwärts" beschreibt Vektoren, deren z-Komponente (Vertikal-Geschwindigkeit) negativ ist. Die Zuordnung zwischen Begriff und numerischen Werten ist dabei nicht scharf mit festen Schwellwerten, sondern unscharf und graduell. Diese Beziehung sollte sich also am besten durch Fuzzy-Mengen beschreiben lassen. Dabei ist die Grundmenge der numerische Wertebereich der Meßwerte. Die Zugehörigkeitsfunktion gibt an, wie stark ein konkreter Meßwert einem gegebenen Begriff entspricht. Dies führt zu folgender Definition:

Definition 2.4.3 (Linguistische Variable) *Eine Linguistische Variable ist ein Tupel* (x, T, U, M)*. Die einzelnen Komponenten haben folgende Bedeutungen:*

x ist der Name der linguistischen Variable.

T ist die Menge aller linguistischen Werte, die x annehmen kann.

U ist die Grundmenge der zu x gehörigen Fuzzy-Mengen.

M ist eine semantische Regel, die jedem Wert aus T eine Fuzzy-Menge auf U zuordnet.

Linguistische Variablen sind Variablen, deren Wertebereich Wörter sind. Jedem linguistischen Wert ist eine Fuzzy-Menge auf einer festen Grundmenge zugeordnet. Die Zugehörigkeitsfunktion der Fuzzy-Menge gibt für ein festes $u \in U$ an, zu welchem Grad die Aussage $x = t$ für ein $t \in T$ erfüllt ist.

Damit könnte man nun bei gegebenen Meßwerten und passender Auswahl der Fuzzy-Mengen den Aussage "Die Geschwindigkeit ist schnell" oder "Die Bewegungsrichtung ist abwärts" Wahrheitswerte zuordnen. Diese Aussagen können wiederum mittels logischer Operatoren zu komplexen Aussagen und Schlußregeln verknüpft werden.

Beispiel 2.4.1 Gegeben ist folgende Schlußregel:

$$\begin{aligned} & \texttt{(Geschwindigkeit = schnell)} \\ \wedge\ & \texttt{(Bewegungsrichtung = abwärts)} \\ \rightarrow\ & \texttt{(Bewegung = Fallen)} \end{aligned}$$

Eine schnelle Abwärtsbewegung soll als Sturz interpretiert werden. Die Eingabe ist ein Geschwindigkeitsvektor. Das Ergebnis soll angeben, zu welchem Grad die Schlußfolgerung `(Bewegung = Fallen)` erfüllt ist. Die linguistische Variable `Geschwindigkeit` beschreibt den Betrag des Geschwindigkeitsvektors. Dem linguistischen Wert `schnell` ist die in Abbildung 2.2 (a) dargestellte Fuzzy-Menge zugeordnet. Die linguistische Variable `Bewegungsrichtung` ist mit der z-Komponente des Geschwindigkeitsvektors verknüpft. Dem Wert `abwärts` ist die Fuzzy-Menge aus Abbildung 2.2(b) zugeordnet. Die "und"-Verknüpfung aus der Voraussetzung der Regel soll als *min*-Verknüpfung realisiert werden.

Bei gegebenem Geschwindigkeitsvektor erhalten die beiden Aussagen in der Voraussetzung einen Wahrheitswert. Durch die "und" Verknüpfung erhält die gesamte Voraussetzung einen Wahrheitswert. Dieser Wert gibt an, zu

welchem Grad die Voraussetzung der Regel erfüllt ist und heißt daher *Aktivierungsgrad* der Regel. Der Schlußfolgerung der Regel sollte sinnvollerweise der gleichen Wahrheitswert zugewiesen werden (siehe Abb. 2.2).

Sind mehrere Regeln gegeben, so müssen die einzelnen Schlußfolgerungen sinnvoll verknüpft werden. In unserem Beispiel könnte man die Bewegung auswählen, die den größten Wahrheitswert hat.

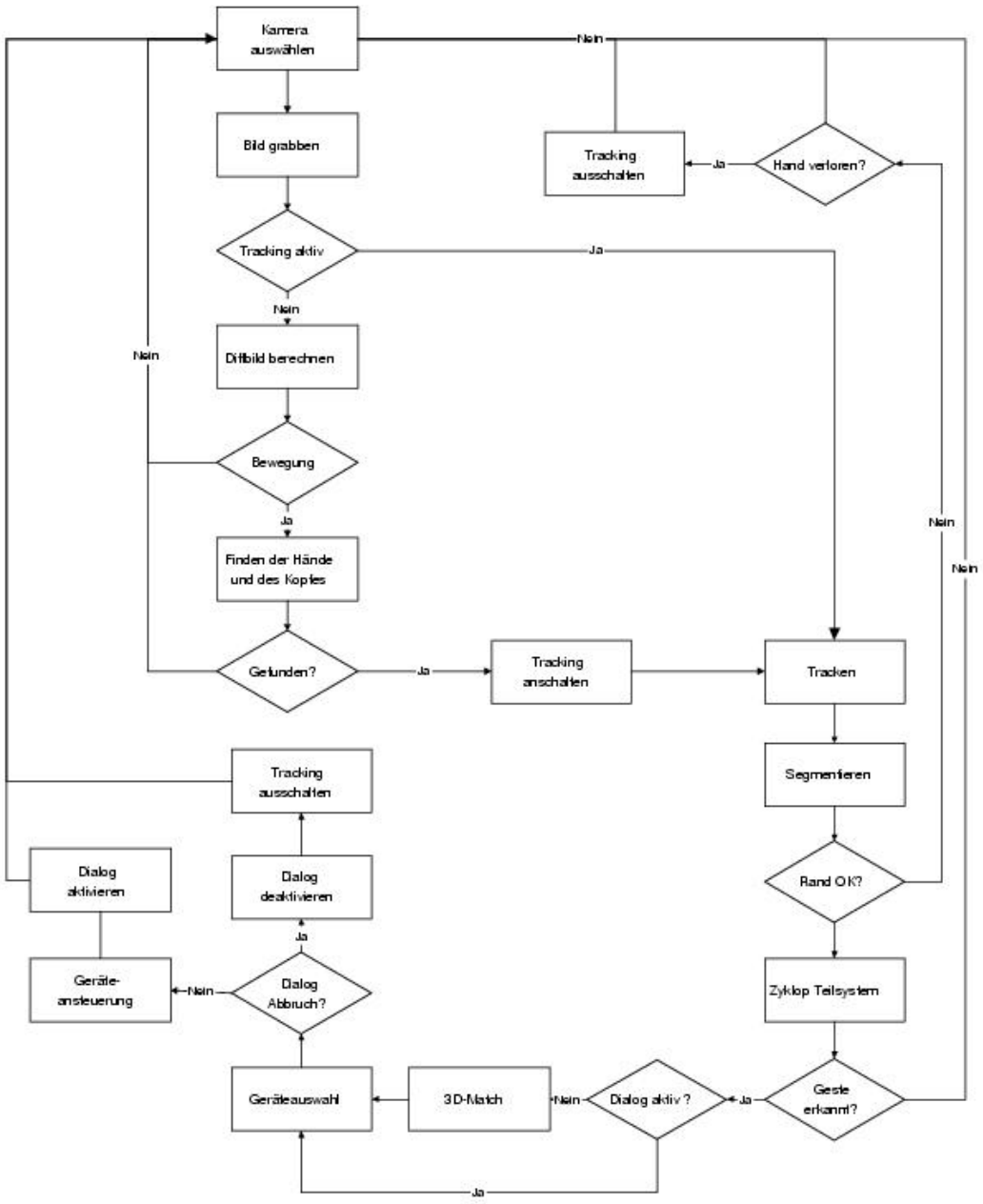

Abbildung 2.1: ARGUS Flußdiagramm

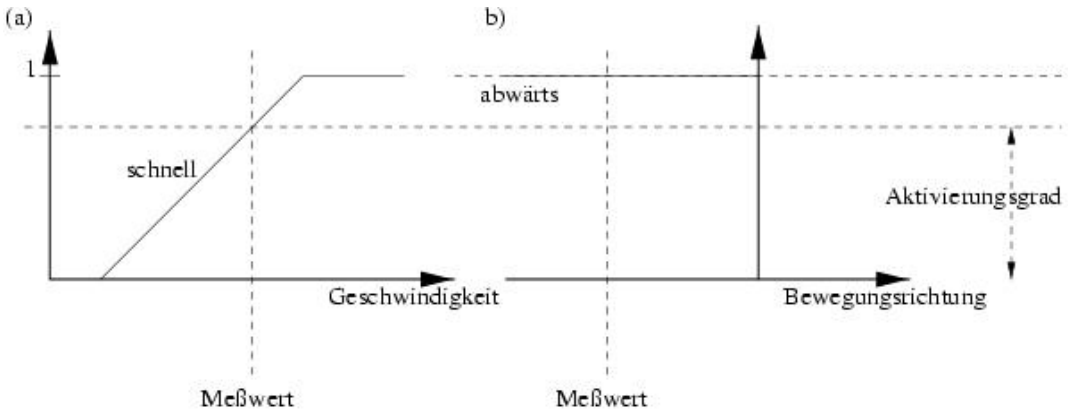

Abbildung 2.2: Schlußfolgerung mittels Fuzzy-Regel

Kapitel 3

Szenario

Im folgenden Kapitel werden wir uns mit den möglichen Notfällen und sonstigen Situationen, die eintreten und vom System erkannt werden können, befassen.

Bei der Überwachung gehen wir von einem normalen Wohnraum aus. Die wesentlichen Einrichtungsgegenstände, wie z.B. Betten, Schränke, Tische u.ä., sollten ihre Positionen, während das Programm aktiv ist, nicht verändern.

Nach der Modellierung über das Userinterface wird der Raum als konstant angenommen. Eine Veränderung hätte eine Neumodellierung zur Folge.

An den zu überwachenden Menschen werden keine weiteren besondern Anforderungen gestellt. Die Erkennung ist weitestgehend unabhängig von Aussehen, Statur und Bekleidung der beobachteten Person. Allerdings sollten keine die Umrisse wichtiger Körperteile entstellenden Kleidungsstücke, wie z.B. Helme, Skimasken und Vergleichbares, getragen werden.

Die Erkennung ist auf maximal eine Person ausgelegt, weil bei zwei sich im Raum befindenden Personen eine Beobachtung nicht mehr notwendig ist, da die zweite Person die Überwachungsfunktion übernimmt.

Die Bewegungungen dieser Person kann man grob in zwei Klassen unterscheiden: In Notfallsituationen, bei denen ein Alarm ausgelöst wird und Normalsituationen, die keine Änderung des Alarmstatus hervorrufen. Es wird versucht, die jeweils erkannten Bewegungen den zwei Klassen zuzuordenen und dementsprechend einen anderen Alarmstatus einzustellen. Im Folgenden werden die einzelnen Bewegungsmuster,die auftreten können, näher analy-

siert und in ihre Bewegungen in die drei Raumrichtungen zerlegt. Letztendlich wird ein Ergebnis, Notfallsituation oder Normalsituation, ausgegeben.

3.1 Notfallsituationen

Nachfolgend werden wir alle vom System erkennbaren Notfälle anführen und beschreiben.

Zu diesen gehören:

1. Sturz

   Ein Sturz ist gekennzeichnet durch eine schnelle, rapide Abwärtsbewegung des Körpersschwerpunktes und des Kopfes. Die hierbei auftretenden abwärtsgerichteten Geschwindigkeiten liegen dabei deutlich höher als bei kontrollierten, willentlich durchgeführten Abwärtsbewegungen wie z.B. beim Hinsetzen, Hinknien oder Bücken. Vom System wird der Sturz anhand der Bewegungsrichtung (größter Teil der Bewegung verläuft in Richtung z-Achse nach unten) und der Bewegungsgeschwindigkeit (siehe obige Ausführungen) erkannt.

2. Inaktivität

   Sollte sich eine Person unnatürlich lange nicht mehr bewegen, wird dies als Notfallsituation klassifiziert. Die Zeitspanne, nach der ein Alarm ausgelöst wird, richtet sich nach der Position der Person im Raum und der vorangegangenen erkannten Tätigkeit. Die Zeitschranke bis zur Alarmauslösung ist umso größer, je wahrscheinlicher eine Inaktivität an der zuvor bestimmten Position ist. In diese Wahrscheinlichkeitsberechnung fliessen die sich an dieser Stelle befindenden Einrichtungsgegenstände ein.

   Dies bedeutet, daß bei einer Person, die im Bett liegt, das Timeout bis zur Alarmauslösung länger angesetzt wird als bei einer Person, die inaktiv am Boden liegt, da hier die Wahrscheinlichkeit für einen Notfall höher zu bewerten ist.

3.2 Normalsituationen

Die Normalsituationen umfassen alle vom System erkannten Bewegungen, die nicht als Notfallsituation klassifiziert werden und somit keinen Alarm auslösen.

Diese umfassen:

1. Liegen (am Boden)

   Das Liegen einer Person am Boden wird durch die Höhe der Körper- und Kopfschwerpunkte, deren Ausrichtung zueinander und die Bewegungsrichtung und -geschwindigkeit charakterisiert.

   Charakteristische Werte sind:

   - Sowohl Körper- als auch Kopfschwerpunkt befinden sich bzgl. der z-Achse annähernd auf Bodenniveau.
   - Die Achse durch die beiden Schwerpunkte verläuft parellel zum Boden.
   - Die Geschwindigkeit ist Null, eine Bewegungsrichtung ist nicht vorhanden.

2. Liegen (erhöht)

   Für die Aktion erhöhtes Liegen gilt selbiges wie für das Liegen auf dem Boden, nur sind die Werte auf der z-Achse entsprechend dem sich an der Position, an der die Person liegt, befindenden Einrichtungsgegenstandes nach oben verschoben.

3. Sitzen (am Boden)

   Für das Sitzen auf dem Boden ergeben sich folgende charakteristische Werte:

   - Der Körperschwerpunkt befindet sich leicht erhöht, die Kopfposition ist vertikal über dem Körperschwerpunkt.
   - Die Bewegungsgeschwindigkeit ist gleich Null. Der Betrag des Vektors für die Bewegungsrichtung ist ebenfalls Null.

4. Sitzen (erhöht)

   Für das erhöhte Sitzen gilt die selbe Beziehung wie zwischen Liegen auf dem Boden und erhöhtem Liegen.

5. Rollen

   Diese Aktion soll die Bewegung eines Rollstuhlfahrers klassifizieren. In diesem Fall gehen wir von einer Person aus, die erhöht sitzt und deren Körper- und Kopfschwerpunkte sich mit gleicher, konstanter Geschwindigkeit in die selbe Richtung bewegen.

6. Stehen

   Beim Stehen bildet die Achse durch die Kopfposition und den Körperschwerpunkt einen rechten Winkel mit dem Boden. Die Höhe und relative Lage der Schwerpunkte zueinander sollte dabei den durch die anatomischen Gegebenheiten, wie z.B. Größe der jeweils zu überwachenden Person, entsprechend plausible Werte liefern. Da es sich wie beim Sitzen und Liegen um eine Ruheposition der Person handelt, sind auch hier Bewegungsgeschindigkeit gleich Null und keine Bewegungsrichtung beobachtbar.

7. Gehen

   Für das Gehen gelten die gleichen Rahmenbedingungen für Kopfposition und Körperschwerpunkt wie beim Stehen, allerdings sind in diesem Fall die Bewegungsgeschwindigkeit ungleich Null und die eine Bewegungungsrichtung definiert. Beide Schwerpunkte bewegen sich mit gleicher, konstanter Geschwindigkeit in die selbe Richtung. Evtl. leichte Auf- und Abwärtskomponenten, die sich im Bewegungsablauf abwechseln, werden in dem Bewegungsvektor geglättet. Die dominierende Komponente der Bewegung ist horizontal, parallel zum Boden gerichtet.

8. Aufwärtsbewegung

   Im Gegensatz zum Gehen ist hier Hauptkomponente des Bewegungsvektors vertikal nach oben gerichtet. Kopf und Körperschwerpunkt bewegen sich mit annähernd der gleichen Geschwindigkeit in die selbe Richtung. Eine Aufwärtsbewegung ist immer Anlaß für eine Entwarnung sofern ein (Vor-)Alarm vorangegangen ist.

9. Abwärtsbewegung

   Bei der Abwärtsbewegung ist die Hauptbewegungskomponente nach unten gerichtet. Je nach Bewegungsgeschwindigkeit kann eine rasche Abwärtsbewegung in einen Sturz übergehen.

Kapitel 4

Design

4.1 Entwurf

Der Entwurf im Projekt ***Ahelp*** gliederte sich in folgende Phasen:

Analyse In der Analysephase wurde die Gesamtspezifikation in Teilaufgaben gegliedert. Diese Teilaufgaben sollten so konkret sein, daß sie sich als grobe Spezifikation für eine Implementierung eignen.

Modularisierung Die Teilaufgaben wurden zu einzelnen Modulen gruppiert.

Spezifikation der Schnittstellen Die Modularisierung wurde in die konkrete Implementierungssprache (C++) umgesetzt und die Modulschnittstellen und zugehörige Datenstrukturen wurden definiert.

Die einzelnen Phasen und ihre Ergebnisse werden in den folgenden Abschnitten nähere beschrieben.

4.1.1 Analyse

In der Analysephase wurden folgende Teilaufgaben erkannt:

Bildverarbeitung Aus den Kamerabildern muß die Position der Person im Raum ermittelt werden. Da eine genaue Modellierung technisch kaum

realisierbar ist, beschränken wir uns auf Körpermerkmale, die einerseits leicht zu lokalisieren sind, die aber andererseits genug Information beinhalten, um Notfallsituationen zu erkennen. Als sinnvoll haben sich Kopf und Körperschwerpunkt erwiesen. Es werden sowohl Position als auch die Geschwindigkeit dieser Merkmale ermittelt.

Bewegungsanalyse Die in der Bildverarbeitung extrahierten Körpermerkmale werden als Bewegungen interpretiert. Die Zuordnung zu den Bewegungen wird als Fuzzy-Menge realisiert, da die Erkennung mit Unsicherheit behaftet ist.

Bewertung Die erkannten Bewegungen werden im Kontext der Raumposition bewertet:
Beispielsweise ist Liegen auf dem Bett als Normalsituation anzusehen, während längeres Liegen auf dem Boden als Notfall erkannt werden sollte.

Modellierung des Raums Die Bewertung der Bewegungen erfordert Zusatzwissen über den Raum. Dieses Wissen muß in ***Ahelp*** entsprechend modelliert werden. Auch die Bildverarbeitung kann durch ein Raummodell unterstützt werden: die Menschliche Bewegung muß von anderen Bewegungen im Raum (im Folgenden "Rauschen" genannt) unterschieden werden. Die Kenntnis von Bewegungsquellen (z.B. Fernseher, Fenster, Spiegel) kann diese Unterscheidung erleichtern.

Steuerung Die Steuerung beinhaltet die Benutzerschnittstelle, koordiniert das Zusammenspiel der anderen Module und verarbeitet in einem Notfall den Alarm.

4.1.2 Modularisierung

Die in der Analyse erkannten Teilaufgaben lassen sich direkt in Module umsetzen. Die Beziehungen zwischen den einzelnen Modulen sind in Abbildung 4.1 dargestellt. Insbesondere fällt auf, daß eine fast "Pipeline"-artige Verarbeitung vorliegt.

Da in der Implementierungssprache C++ Module durch Klassen realisiert werden, kann man von jedem Modul mehrere Instanzen erzeugen. Sollen mehrere Räume überwacht werden, so ist es sinnvoll, wenn jeder Raum je eine

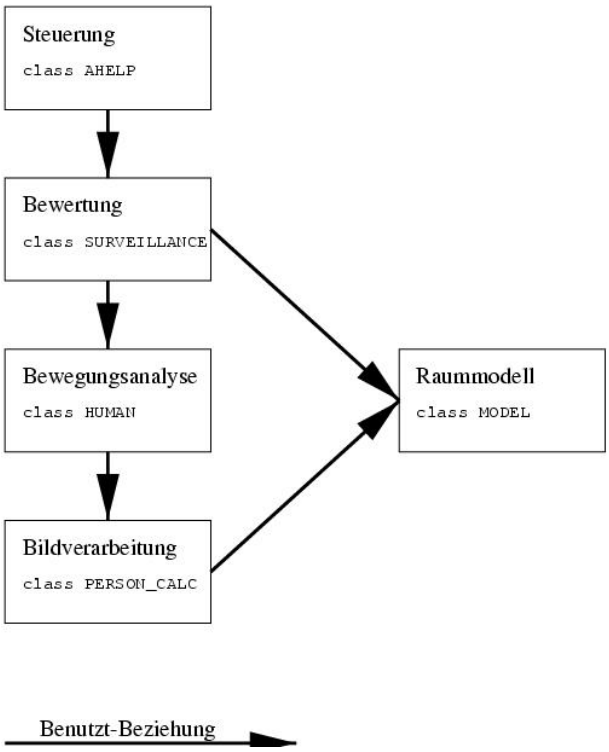

Abbildung 4.1: ***Ahelp*** Design

eigene Instanz der Module "Bildverarbeitung", "Bewegungsanalyse", "Bewertung" und "Raummodell" erhält. Ein Raum wird dann durch ein Objekt der Klasse ***ROOM*** repräsentiert, das Zeiger auf die einzelnen Instanzen der zugehörigen Module enthält (siehe Abbildung 4.2). Die einzelnen Räume werden im Steuerungsmodul verwaltet.

4.1.3 Spezifikation der Schnittstellen

Das Bildverarbeitungsmodul wird durch die Klasse ***PERSON_CALC*** implementiert. Die Schnittstelle der Klasse ist im Anhang A.1.12 dokumentiert. Die wichtigste Funktion ist `PERSON calculate_person(void)`. Diese Funktion liefert ein ***Person***-Objekt, das den momentanen Zustand der Person (bestehend aus Kopfposition, Körperschwerpunkt und zugehörigen Geschwindigkeiten) beschreibt (siehe **??**. Näheres zum Bildverarbeitungsmodul

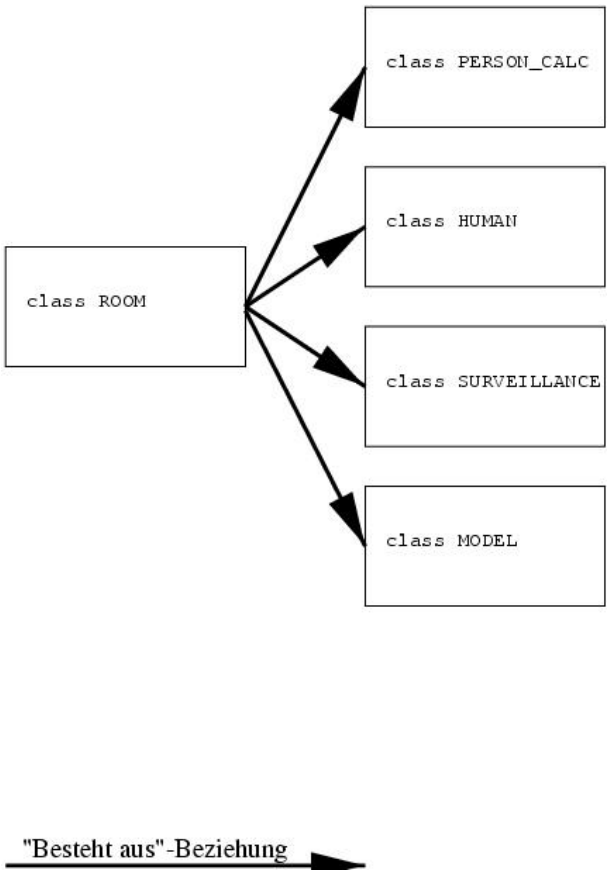

Abbildung 4.2: Die ***ROOM***-Klasse

findet man in Abschnitt 5.2.

Die Bewegungsanalyse ist in der Klasse ***HUMAN*** gekapselt. Die Schnittstelle der Klasse ist in A.1.26 beschrieben. Die Methode `HUMAN_ACTION get_human_action()` ruft die Bildverarbeitung auf, analysiert die Bewegungen und liefert als Ergebnis ein Objekt der Klasse ***HUMAN_ACTION***. In dieser Klasse sind die Fuzzy-Menge der erkannten Bewegungen, die Position im Grundriß und die Bewegungsrichtung gekapselt (siehe A.1.27). Eine nähre Beschreibung findet sich in Abschnitt 5.3.

Die Bewertung der erkannten Bewegungen findet in der Klasse ***SURVEILLANCE*** statt. Eine Beschreibung der Schnittstelle ist in A.1.24 angegeben.

Die wichtigste Funktion dieser Klasse ist `ALARM request_alarm_status()`. Diese Funktion ruft die Bewegungsanalyse auf, bewertet die erkannten Bewegungen und liefert als Ergebnis ein ***ALARM***-Objekt (siehe A.1.25). Dieses Objekt beschreibt den aktuellen Alarmzustand und eventuell zugehörige Parameter (z.B. den Raumnamen, falls die Person einen anderen Raum betritt). Details zur Bewertung findet man in Abschnitt 5.4.

Die Steuerung und Benutzeroberfläche wird in der Klasse ***AHELP*** zur Verfügung gestellt. Wird die Überwachung gestartet, so wird zyklisch die Funktion `ALARM request_alarm_status()` aufgerufen. Tritt ein Alarm auf (oder allgemein eine Situation, die eine Reaktion erfordert), so werden die entsprechenden Methoden in ***AHELP*** aufgerufen. Momentan wird nur die Überwachung eines Raums unterstützt. Wie in den vorigen Abschnitten angedeutet, läßt sich ***Ahelp*** aber leicht auf mehrere Räume erweitern.

Das Raummodell wird in der Klasse ***MODEL*** verwaltet. Die Schnittstelle dieser Klasse wird in A.1.15 beschrieben. Eine genaue Beschreibung des Raummodells findet man in 5.1.

4.2 Erweiterbarkeit

Ahelp ist zur Überwachung einer einzelnen Person in einem Raum kozipiert. Es ist jedoch wünschenswert die zu Übwachende Fläche auf mehrere Räume bzw. die ganze Wohnung oder das ganze Haus auszudehnen.
Dieser Gedanke wurde bei der Planung von ***Ahelp*** in soweit aufgenommen, in dem für den momentan verwendeten Raum ein eigenes Objekt erstellt wird. So ist es nicht besonders Schwierig für verschiedene Räume eigene Objekte anzulegen und beim Verlassen eines Raumes die Überwachung in einen anderen Raum zu verlegen.
Ursprünglich wurde das System unter Verwendung von beweglichen Kameras geplant, die die zu überwachende Person selbständig verfolgen, um so die Überwachung des gesamten Raumes zu gewährleisten. Dies stellte sich jedoch als ein nicht gerade triviales Ziel heraus, so daß wir ersteinmal starre Kameras verwendeten. Später, wenn noch Zeit vorhanden wäre (was natürlich nicht der Fall war), wollten wir zumindest die Kameras schrittweise ansteuern. Das heißt, bewegt sich die Person aus dem Bild, sollten die Kameras in einem festen Winkel der Person nachgeführt werden.

4.3 Basisklassen

Im Folgenden werden die Klassen bzw. Typen beschrieben, die sich nicht direkt einem Modul zuordnen lassen, da sie in mehreren Modulen bzw. Klassen Verwendung finden.

4.3.1 TOKEN

Bei vielen Klassen, die wir entworfen haben, stellte sich die Aufgabe, den Inhalt der Objekte in einer Datei speichern und ihn auf dem Bildschirm ausgeben zu können. In C++ bietet es sich daher an, die iostream-Operatoren zu überladen. Diese ermöglichen die Ein- und Ausgabe, sowohl auf dem Bildschirm als auch in einer Datei. Um diese Ein- und Ausgaben einheitlich zu gestalten, wurde der Aufzählungstyp ***TOKEN*** eingeführt. Die iostream-Operatoren sind so überladen, daß bei Ausgabe eines ***TOKEN***s eine vorher festgelegte, dem ***TOKEN*** zugeordnete Zeichenkette ausgegeben wird. Beim Lesen der vorher definierten Zeichenfolge (nach dem Löschen evtl. vorhandener "Whitespaces" und/oder mittels '#' eingeleiteter Kommentarzeile(n)) wird bei korrekter Zeichenfolge der ***TOKEN*** entsprechend gesetzt. War die Zeichenkette kein gültiger ***TOKEN*** wird "FAIL" als Wert zurückgeliefert und der ***istream*** geht in den "Bad"-Zustand über, so daß keine weiteren Eingaben von diesem Stream erfolgen können. Man kann leicht neue ***TOKEN***s einführen. Dazu muß man lediglich in "tokens.H" den Aufzählungstyp ***TOKEN*** vor dem "token_end"-***TOKEN*** und hinter dem ersten ***TOKEN***, der den Wert '0' haben muß, um einen neuen ***TOKEN*** erweitern. Schließlich muß die Variable *tokens* in "tokens.C" um ein Tupel {<***TOKEN***>, <Zeichenkette>} erweitert werden. Bei der Zeichenkette muß beachtet werden, daß keine "Unterbrechung" enthalten ist. Dies kann man an der Implementierung von `is_break` erkennen und falls notwendig ändern. Will man einzelne Zeichen als ***TOKEN*** benutzen, müssen diese eine "Unterbrechung" sein. Nun folgt eine Tabelle der bisher definierten ***TOKEN***s:

TOKENs		
KLASSE	***TOKEN***	'Zeichenkette'
MODEL		
	tok_MODEL	'MODEL'
	tok_name	'name'
	tok_x_l	'x_length'
	tok_y_l	'y_length'
	tok_cam_c	'Number_of_Cameras'
	tok_obj_c	'Number_of_Objects'
	tok_door_c	'Number_of_Doors'
	tok_win_c	'Number_of_Windows'
	tok_fileend	'END_OF_FILE'
SQUARE		
	tok_SQ	'SQUARE'
	tok_actions	'actions'
	tok_height	'height'
	tok_camcount	'camcount'
	tok_visible	'visible'
	tok_near_door	'near_door'
	tok_hidden_exit	'hidden_exit'

TOKENs		
KLASSE	***TOKEN***	'Zeichenkette'
CAMERA		
	tok_CAM	'CAMERA'
	tok_a_w	'angle_width'
	tok_a_h	'angle_height'
	tok_r_x	'resolution_x'
	tok_r_y	'resolution_y'
	tok_di	'direction'
	tok_pos	'position'
	tok_config	'config_data'
OBJECT		
	tok_OBJ	'OBJECT'
	tok_rectangle	'Rectangle'
	tok_circle	'Circle'
	tok_triangle	'Triangle'
TRIANGLE		
	tok_left_bottom	'left_bottom'
	tok_left_top	'left_top'
	tok_right_bottom	'right_bottom'
	tok_right_top	'right_top'
DOOR		
	tok_DOOR	'DOOR'
	tok_trgt_room	'target_room'
WINDOW		
	tok_WIN	'WINDOW'
	tok_edge	'edge'
	tok_move	'move'
Globale ***TOKEN***s		
	tok_l_paren	'('
	tok_r_paren	')'
	tok_l_brace	'{'
	tok_r_brace	'}'
	tok_semicolon	';'
	tok_colon	':'
	tok_equal	'='
	tok_comma	','
	tok_plus	'+'
	tok_minus	'-'
	FAIL	undefiniert

4.3.2 HISTORY

Es zeigte sich früh, daß an mehreren Stellen im Problem in irgendeiner Form Listen mit vergangenen Ereignissen geführt werden mußten. Um nicht an jeder dieser Stellen das Rad neu zu erfinden, wurden die Template-Klassen ***HISTORY*** und ***LHISTORY*** implementiert. History ermöglicht das einfache Führen virtuell unbeschränkter Listen vergangener Ereignisse (im weiteren Histories genannt). Die in der History gespeicherten Ereignisse müssen sich vom Typ ***EVENT*** ableiten, um ein zeitabhängiges Einfügen und Löschen der Ereignisse aus der History zu ermöglichen.

4.3.3 LHISTORY

LHISTORY stellt im Prinzip die gleiche Funktionalität wie ***HISTORY*** zur Verfügung, mit dem Unterschied, daß hier die Länge der History bei der Initialisierung vorgegeben werden kann. Wird diese Länge überschritten, so wird jeweils der älteste Event gelöscht. Auch ***LHISTORY*** verwaltet Objekte des Typs ***EVENT***.

4.3.4 EVENT

Die Klasse ***EVENT*** ist eine generische Klasse, die von ***HISTORY*** und ***LHISTORY*** verwendet wird. Sie kapselt lediglich eine Start- und eine Endzeit eines Events in sich. Eventuelle Beschreibungen des Events, also tatsächliche nutzbare Informationen bleiben den erbenden Klassen vorbehalten.

Für jede dieser Erben von ***EVENT*** muß dann eine entsprechende History Klasse in der Form `HISTORY<EVENT> eventhist;` abgeleitet werden.

4.3.5 INT_FUZZY_SET

Die Klasse ***INT_FUZZY_SET*** verwaltet Fuzzy-Mengen auf den positiven ganzen Zahlen. Im wesentlichen werden die Folgenden Operationen unterstützt:

- Setzen und Lesen von Zugehörigkeitswerten

- Durchschnitt (min) ($*$)
- Vereinigung ($+$)
- Komplement ($-$)

Die Zuordnung zwischen Zahlen und ihren Zugehörigkeitswerten wird in einem Array verwaltet. Die Größe des Arrays passt sich automatisch an den größten verwendeten Index an. Um unnötiges Kopieren des Arrays zu verhindern, kann man im Konstruktor eine Startgröße angeben. Zusätzlich wird ein Zugehörigkeitswert für alle nicht explizit definierten Elemente der Grundmenge definiert, um den Komplement-Operator zu unterstützen. Eine genauere Beschreibung aller Funktionen findet man in A.1.1.

4.3.6 DOUBLE_FUZZY_SET

Fuzzy-Mengen auf den reellen Zahlen kann man i.A. nicht durch abspeichern einzelner Zugehörigkeitswerte verwalten. Würde man sich nur auf endliche Teilmengen von $\Re$ beschränken, so wäre es beispielsweise nicht möglich, stückweise stetige Zugehörigkeitsfunktionen zu verwaltet.

Eine bessere Möglichkeit ist, die Zugehörigkeitsfunktion abzuspeichern. Hierbei tritt das Problem auf, daß die Zugehörigkeitsfunktionen von Durchschnitt und Komplement oft schwierig zu berechnen sind (falls sie sich überhaupt in geschlossener Form darstellen lassen ...).

Ein Kompromiß ist die Beschränkung auf eine Klasse von Funktionen, die eine Teilmenge der Operationen effizient unterstützt. Die Klasse ***DOUBLE_FUZZY_SET*** verwaltet Fuzzy-Mengen auf den reellen, Zahlen, deren Zugehörigkeitsfunktionen trapezförmig sind. Da sich eine trapezförmige Funktion durch vier Punkte auf der x-Achse und ihr Maximum beschreiben läßt, kann man sie Problemlos abspeichern. Auch das berechnen der Zugehörigkeitswerte besteht im wesentlichen aus wenigen Vergleichen und der Anwendung des Strahlensatzes und läßt sich damit effizient ausführen.

Man kann sich allerdings leicht an Beispielen klarmachen, daß Vereinigung und Komplement von trapezförmigen Zugehörigkeitsfunktionen i.A. *nicht* trapezförmig sind. Daher werden diese Operationen von der Klasse ***DOUBLE_FUZZY_SET*** nicht unterstützt. Als Durchschnitt von zwei trapezförmigen Zugehörigkeitsfunktionen ergibt sich jedoch meistens eine trapezförmige

Funktion. In den wenigen Ausnahmen läßt sich das Ergebnis sehr gut durch eine Trapezfunktion approximieren.

Eine genauere Beschreibung aller von ***DOUBLE_FUZZY_SET*** angebotenen Funktionen findet man in A.1.2.

4.4 Softwaretools

4.4.1 Motivation

Zum Entruf größerer Software-Projekte ist es meist sinnvoll, sich auf die Hilfe von Software-Entwicklungstools zu stützen. Deshalb beschlossen auch wir, eines dieser Entwurfstools für die Definition unsere Klassenhierachie und der Klassenschnittstellen einzusetzen. Nach Absprache mit dem LS10 und in Hinblick auf unseren geplanten Objekt-orientierten-Entwurf entschieden wir uns, das bereits im Software-Praktikum eingesetzte BONSAI zu benutzen.

4.4.2 BONSAI

Bei BONSAI handelt es sich um ein straight-forward-tool. Es bietet graphische und textuelle Editoren, die den Software-Entwurf mit der Objekt-orientierten Methode BON(Business Object Notation) nach Kim Walden und Jean-Marc Nerson unterstützen. Mit BONSAI kann man ein Dokument anlegen, das die Objekte und Relationen graphisch und textuell beschreibt, die zum Software-Entwurf gehören.

Desweiteren enthält BONSAI ein Systax- und Zyklencheck, um die Abhängigkeiten zwischen den einzelnen Objekten zu prüfen. Außerdem sollte eine Codegenerierung, die C++ Coderahmen sowie die dazu gehörigen Headerfiles erzeugt, angeschlossen werden.

Es besitzt mehrere Chart-Editoren, die zur Beschreibung der Klassenschnittstellen dienen. Da BONSAI bisher nur im Software-Praktikum im Zusammenhang mit der Programmiersprache BETA eingesetzt wurde, mußten die Chart-Editoren erst an C++ angepaßt werden. Viele der von C++ untersützten Programmierkonzepte, wie z.B. Inline-Routinen, public, protected und

private Variablendeklarationen, ließen sich nicht direkt codieren. Eine Nachbearbeitung der Coderahemen wäre in diesem Fall nötig gewesen, weil die Anpassung der Charteditoren in manchen Fällen nicht möglich war.

Aus diesem Grund verzichteten wir dann auf die Verwendung dieses Tools, weil wir in unserer Designphase bereits so weit fortgeschritten waren, daß ein Warten auf die Umstellung der Editoren und Fertigstellung der Coderahmengenerierung einen größeren Zeitverlust mit sich gebracht hätte.

An dieser Stelle danken wir dem LS10, insbesondere Dr. S. Dißmann und seiner Hiwi-Crew, für die Mühen.

4.4.3 Allgemeines über BONSAI

Bei BONSAI handelt es sich um ein CASE-Tools, das eine Instanz des Systems KOGGE ist. Dieses wurde an der Universität Koblenz im Rahmen eines Forschungsprojektes des Institut für Softwaretechnik unter der Leitung von Prof. Jürgen Ebert und Prof. Manfred Rosendahl entwickelt. Ziel dieses Projektes ist die Generierung eines benutzerspezifischen CASE-Tools für den Einsatz im Software-Entwurf.

In diesem Zusammenhang danken wir der Universität Koblenz für Ihre Zusammenarbeit.

4.5 Technik

4.5.1 Programmiersprache

In der ersten Phase des Designs stellte sich für uns die Frage, welche Programmiersprache zu verwenden sei. Innerhalb kurzer Zeit einigten wir uns auf C++, obwohl nur wenige von uns diese Sprache bereits kannten. Ausschlaggebend war für uns, daß diese Sprache es uns ermöglichte, zum einen Codesegmente aus Argus beziehungsweise Zyklop zu übernehmen, andererseits aber auch unserer Auffassung des Problems (die im wesentlichen objektorientiert ist) näher kam als die von unseren sogenannten Vorgängern verwendete Sprache C.

4.5.2 Verwendete Hardware

Als Hardware für die eigentliche Anwendung stand uns ein PC auf Basis eines Intel Pentium 133 mit 80MB RAM und insgesamt ca. 4GB PLattenspeicher zu Verfügung (leider nicht zur alleinigen Verfügung). Ausgestattet mit einer Matrox Millennium mit 4MB WRAM und zwei Matrox Meteor Framegrabber-Karten ist dieser PC für Bildverarbeitungszwecke fast wie geschaffen. Lediglich ein schnellerer Prozessor wäre manchmal (für uns meist) wünschenswert.

4.5.3 Sonstige Technik

Um gemeinsam ein größeres Projekt bearbeiten zu können muß man in geeigneter Weise die Zugriffe auf die Sourcen kontrollieren, da es sonst schnell dazu kommt, daß mehrere Personen gleichzeitig am gleichen Source-Segment arbeiten, ohne voneinander zu wissen. Schlimmer noch: Die Änderungen überschneiden sich unter Umständen zeitlich, was dazu führen kann, daß Änderungen verloren gehen.

Um diesen Problemen zu begegnen, entschlossen wir uns, eine Versionsverwaltung mit Zugangssteuerung zu benutzen, in unserem Falle entschlossen wir uns RCS einzusetzen. Da RCS pur jedoch eher schlecht benutzbar ist, wurden einige Shell- und Perlskripte geschrieben, um die Handhabung zu erleichtern. Diese sind auf unsere konkreten Bedürfnisse und insbesondere

auf die von uns verwendete Verzeichnisstruktur zugeschnitten. Im einzelnen wurden folgende Skripte geschrieben und verwendet:

- `rcsinit`: Dient zum erstmaligen Anlegen des RCS-Files. Es unterschiedet verschiedene Dateitypen, und kann, wenn die zu initialisierende Datei noch nicht existiert, ein passendes Template kopieren.
- `rcsfiles`: Zeigt die Dateinamen der im RCS-System bekannten Files an
- `rcsall`: Führt einen Checkout aller im System bekannten Dateien aus, ohne die existierenden Lockings zu ändern.
- `rcslocks`: Zeigt an, wer welches File gelockt hat (also bearbeiten darf).
- `checkout`: Checkt ein File aus dem System aus, und lockt es per Default.
- `checkin`: Checkt ein File wieder in das System ein, und hebt die Blockierung wieder auf.

Die Skripten akzeptieren teilweise über den Dateinamen hinaus die Optionen der RCS-Befehle, insbesondere die Optionen -u (no lock, beim checkout) und -l (lock, beim checkin).

Kapitel 5

Implementierung

5.1 Die Modellierung des Raums

Die Raumdaten, das heißt die Daten, die den Raum beschreiben, wie z.B. die Länge und die Breite des Raums, sind in der Klasse ***MODEL*** gekapselt. Zu den Daten, die den Raum beschreiben, zählen auch die Kameradaten, die in der Klasse ***CAMERA*** enthalten sind. Objekte wie Tische oder Schränke werden durch ***RECTANGLE***s, ***TRIANGLE***s oder ***CIRCLE***s beschrieben, die von der Klasse ***OBJECT*** als Basisklasse die Fähigkeit erben, sich mit ihrer zugehörigen Höhe in den Grundriß des Raums einzutragen. Die Einteilung des Grundrisses erfolgt in ***SQUARE***s. Ausgänge und Fenster werden gesondert behandelt. Sie werden durch die Klassen ***DOOR*** und ***WINDOW*** beschrieben (siehe Abbildung 5.1). Die Modellierung von kleineren beweglichen Gegenständen wie etwa Stühlen machte keinen Sinn, da wir mit der zur Zeit vorhandenen Rechenleistung nicht in der Lage sind, die Differenzbildmethode auf das gesamte Bild häufig genug anwenden zu können, um die Position eines beweglichen Gegenstandes zu ermitteln. (Differenzbildmethode benötigte bei Anwendung auf die zwei Kamerabilder ca. 1s auf P133) Wir mußten uns also auf das Verfolgen von menschlicher Bewegung, d.h. von Kopfposition und Körperschwerpunkt beschränken; daher die grobe Modellierung des Raums.

5.1.1 ROOM

Die Klasse ***ROOM*** faßt das Modell des Raums, welches durch die Klasse ***MODEL*** repräsentiert wird, die zugehörige Bildverarbeitung, repräsentiert durch ***PERSON_CALC***, und die auswertenden Klassen (***HUMAN*** und ***SURVEILLANCE***) zu einer Klasse zusammen. Sie bietet den verschiedenen Modulen die Möglichkeit Daten auszutauschen. Dies geschieht indem jeder Instanz ein Zeiger auf den ***ROOM***, zu dem sie gehört, übergeben wird. Die Klasse ***ROOM*** stellt selbst Funktionen zur Verfügung, die jeweils einen Zeiger auf die Instanz der jeweils anderen Klasse zurückliefert. So kann z.B. die Klasse ***PERSON_CALC*** auf die in der Klasse ***MODEL*** gespeicherten Kameradaten zugreifen.

5.1.2 MODEL

Die Klasse ***MODEL*** dient als Schnittstelle zu den übrigen Modulen. Es werden Funktionen zur Verfügung gestellt, die der Bildverarbeitung darüber Auskunft geben könnten, zu welchem Grad in einem bestimmten Bildausschnitt Störungen zu erwarten sind, die z.B. durch Fenster, Fernseher oder Spiegel entstehen. Außerdem besteht die Möglichkeit, die Sicherheiten für bestimmte Bewegungen in bestimmten ***SQUARE***s zu lesen, bzw. zu schreiben. Diese Möglichkeiten werden noch nicht benutzt, da sich unser Hauptaugenmerk auf das korrekte Erkennen der verschiedenen Bewegungen richtete. Diese Funktionen sollten einerseits dazu dienen, Störungen im Bild nicht als Bewegungen zu interpretieren, andererseits um bei einem längeren Einsatz entscheiden zu können ob die gerade erkannte Bewegung am aktuellen Ort eine Gefahr darstellt. Um Tests transparenter zu gestalten, wurde bis jetzt auf diese Funktionalität der Klasse ***MODEL*** verzichtet. Benutzt werden die Funktionen, die den Grad der durch im Modell enthaltene Objekte verursachten Verdeckung angeben, wie auch diejenigen, die die Wahrscheinlichkeit angeben, mit der eine Person den Raum durch eine Tür verläßt (`get_covering` und `get_exit`). Die Funktion `get_covering` liefert als Ergebnis 0, wenn eine Person vollständig sichtbar ist, d.h. wenn das ***SQUARE*** zu der angegebenen Grundrißposition von beiden Kameras in einer Höhe von 0.5m oder weniger einzusehen ist. Ist ein ***SQUARE*** nicht unterhalb von 1.5m einzusehen, liefert die Funktion `get_covering` den Wert 1, bei Werten dazwischen jeweils $(Wert - 0.5)$. Die eben genannten Werte werden zu Beginn in der Funktion

`init` berechnet und in den jeweiligen ***SQUARE***s abgespeichert. In dieser Initialisierungsfunktion werden neben *visible[1...MAXCAMCOUNT]* ebenfalls die booleschen Werte *near_door* und *hidden_exit* berechnet. Die Funktionen zum Laden und Speichern (`load`, `save`) werden von der Klasse ***ROOM*** aus aufgerufen. Desweiteren werden Funktionen zur Verfügung gestellt, die aus Grundrißkoordinaten die zugehörigen ***SQUARE***-Koordinaten berechnen (`get_square_coordinates`), mit denen man dann auf das entsprechende ***SQUARE*** zugreifen kann (`get_square`). Die Kameradaten werden mit der Funktion `get_camera`, die eine Referenz auf das im ***MODEL*** befindliche ***CAMERA***-Objekt liefert, wodurch ein Lesen bzw. Schreiben der Kameradaten möglich ist.

5.1.3 CAMERA

Das ***CAMERA***-Objekt dient lediglich dazu, die Kameradaten einer Kamera zu kapseln. Zu den Kameradaten gehören die horizontale Winkelweite (*angle_width*), die vertikale Winkelweite (*angle_height*), die Auflösung in x- und y-Richtung (*resolution_x*,*resolution_y*), die Blickrichtung (*direction*), so wie die Position im Raum (*position*), als auch die Konfigurationsdaten der beweglichen Kameras (*config_data*). Auf die oben genannten Variablen kann durch die Elementfunktionen, wie `write_res_x`, `read_res_x` usw., schreibend, bzw. lesend zugegriffen werden. Das Schreiben, insbesondere der *config_data*, hat aber keine direkte Auswirkung auf die Hardware. Hier werden nur die reinen Daten verändert. Falls man die Blickrichtung (*direction*) setzt, muß man im entsprechenden ***MODEL*** die Funktion `init` aufrufen, damit die von der Blickrichtung abhängigen Daten, wie Sichtbarkeit, neu berechnet werden.

5.1.4 OBJECT

Die Klasse ***OBJECT*** dient den Klassen ***RECTANGLE***, ***TRIANGLE*** und ***CIRCLE*** als Basisklasse. Diese Klassen erben von ***OBJECT*** die Fähigkeit, sich in den Grundriß einzutragen. Neben der Höhe wird dort auch ein ***INT_FUZZY_SET*** abgespeichert, welches die Grade der Plausibilität von bestimmten Aktionen an dem Ort des ***OBJECT***s repräsentiert. Wenn ein Element der diskreten Fuzzy-Menge beispielsweise die Plausibilität für “Liegen” beinhaltet, wird dieser Wert in einem ***RECTANGLE***, das ein

Bett repräsentiert, näher an 1 liegen als bei einem ***RECTANGLE***, das einen Tisch repräsentiert.

5.1.5 RECTANGLE

Die Klasse ***RECTANGLE*** erbt von der Klasse ***OBJECT*** die Fähigkeit, sich in den Grundriß eintragen zu können. Zusätzlich beinhaltet es die 2D-Vektoren *ll* und *ur*, die die untere linke Ecke bzw. die obere rechte Ecke beschreiben. Die Höhe des Rechtecks ist, da es von ***OBJECT*** erbt, natürlich konstant.

5.1.6 TRIANGLE

Die Klasse ***TRIANGLE*** erbt von der Klasse ***OBJECT*** die Fähigkeit, sich in den Grundriß eintragen zu können. Die Höhe des Dreiecks ist, da es von ***OBJECT*** erbt, konstant. Es handelt sich hier um ein rechtwinkliges Dreieck, dessen Katheten aus Rechteckseiten besteht. Die Variable *form* legt fest, welcher Eckpunkt des Rechtecks Kathetenschnittpunkt wird. Das Rechteck, das das Dreieck beinhaltet, wird durch die Variablen *ll* und *ur*(siehe ***RECTANGLE***) definiert. In der Funktion `registrate`, die die Eigenschaften des Dreiecks in den Grundriß einträgt, wird in der Variablen *step* der Kehrwert der Hypotenusensteigung abgelegt. Dann werden in Abhängigkeit von der Form des Dreiecks die Startwerte für die linke und rechte Grenze des Dreiecks gesetzt. Mittels *step* werden die Änderungen dieser Grenzen pro Zeile festgelegt. Die eigentliche Registrierung erfolgt dann zeilenweise abhängig von diesen 4 Werten.

5.1.7 CIRCLE

Der Kreis (bzw. die Ellipse) wird durch *ll*(lower_left) und *ur* (upper_right) einer Bounding-Box definiert. Die Registrierung erfolgt zeilenweise, wobei in jeder Zeile die Ränder durch Einsetzen der Zeile in eine Ellipsengleichung bestimmt werden. Alle anderen Funktionen sind analog zu ***RECTANGLE***.

5.1.8 DOOR

Die Klasse ***DOOR*** wird von der Klasse ***MODEL*** benutzt, um die im Raum vorhandenen Türen abzuspeichern. Hierzu werden die Grundrißdaten der Ecken der Tür abgespeichert. Außerdem wird der Name des Raums abgespeichert, der durch die Tür erreicht werden kann.

5.1.9 WINDOW

Die Klasse ***WINDOW*** wird von der Klasse ***MODEL*** benutzt, um die im Raum vorhandenen Fenster abzuspeichern. Hierzu werden die Raumkoordinaten der Fensterecken abgespeichert. Die Klasse ***WINDOW*** kann auch dazu benutzt werden, um allgemein störende Flächen im Raum zu modellieren. Zusätzlich zu den vier Ecken wird nämlich der Grad der Bewegung abgespeichert, der aussagen soll, wie viel Bewegung von diesem Fenster zu erwarten sein soll (0 =keine Bewegung, ..., 1 =viel Bewegung). Dieser Wert kann mit `write_movement` gesetzt werden. Die Funktion `get_single_movement` gibt, abhängig vom übergebenen ***CAMERA***-Objekt und dem Bildausschnitt, den Grad der zu erwartenden störenden Bewegung im ausgewählten Bildausschnitt an. Dieses Feature ist jedoch noch nicht benutzt worden, da wir auch ohne störende Flächen genügend Schwierigkeiten bei der Bildverarbeitung hatten.

5.1.10 SQUARE

Die Klasse ***MODEL*** legt ein zweidimensionales Array von ***SQUARE***s an. Jedes ***SQUARE*** repräsentiert eine Grundrißflächeneinheit. Zur Zeit beträgt die Fläche pro ***SQUARE*** $\frac{1}{4}m^2$. In einem ***SQUARE*** wird ein ***INT_FUZZY_SET*** gespeichert, welches die Plausibilitäten für bestimmte Bewegungen enthalten soll. Falls ein ***OBJECT*** auf diesem ***SQUARE*** steht und sich dort mittels der Funktion `registrate` eingetragen hat, werden dessen Höhe und die Aktionsplausibilitäten dort gespeichert.

5.2 Die Bewegungserkennung von *Ahelp*

Aufgabe der Bewegungserkennung ist es, eine Zielperson im Raum zu finden. Als Grundlage wurde hierfür das Differenzbildverfahren eingesetzt. Daher wird zunächst das Binärbild (bzw. in diesem Fall die Binärmatrix) erzeugt. Es sollen sowohl der Oberkörper, als auch der Kopf der Person gefunden werden. Da der Oberkörper größer als der Kopf ist (oder zumindest sein sollte), wirddieser als erstes gesucht. Auf dieser Basis wird dann an den äußeren Seiten des Oberkörpers versucht den Kopf der Person zu identifizieren.

5.2.1 Faster_CreateBinaryMatrix

Ziel der Funktion ist es, ausgehend von den Bildern der Kamera, eine Binärmatrix zu erzeugen.
Als Eingabe werden alle zum aktuellen Zeitpunkt relevanten Bilder einer Kamera zur Verfügung gestellt. Es handelt sich dabei um die folgenden Daten:

- Altes U-Kanal Bild
- Neues U-Kanal Bild
- Altes V-Kanal Bild
- Neues V-Kanal Bild

Aus Geschwindigkeitsgründen werden in der Funktion `Faster_CreateBinaryMatrix` die beiden Einzelfunktionen `CreateBinaryImage` sowie `CreateBinaryMatrix` zusammengefaßt.
Das zu berechnende Binärbild wird für die Berechnung in eine quadratische Matrix aufgeteilt. Die Anzahl der Matrixelemente wird durch die beiden Variablen *x_squares* sowie *y_squares* vorgegeben. Bei einer Bildgröße von 384 mal 288 Bildpunkten hat sich 48 als guter Wert für die Praxis erwiesen. Das Bild wird dann in 2304 Quadrate aufgeteilt. Jedes Quadrat hat eine Größe von acht mal sechs Pixeln und besteht aus 48 Bildpunkten.
Jeder Bildpunkt wird auf Bewegung untersucht. Entsprechend dem zugrundeliegenden Differenzbildverfahren wird dazu die Differenz zwischen dem alten und neuen Bild an der aktuellen Position berechnet. Übersteigt die Differenz

den Schwellenwert *motion_difference*, wird der Punkt gesetzt. Der Schwellenwert wird nicht berechnet sondern heuristisch ermittelt. Die Standardeinstellung ist fünf. Die Berechnung wird sowohl für den U-Kanal, als auch für den V-Kanal durchgeführt. Es existieren also u.U. zwei Bewegungspunkte an derselben Position. Um diese zu trennen, werden die Kanäle durch die Farbwerte *BLAU* (für den U-Kanal) und *ROT* (für den V-Kanal) abstrahiert. Für jedes Quadrat werden die Bewegungspunkte beider Kanäle gezählt. Die Anzahl wird in den Variablen *u_counter* und *v_counter* gesichert. Übersteigt das Maximum der beiden Variablen die vorher definierte Schwelle *square_percent*, so wird das gesamte Quadrat in der entsprechenden Farbe (blau oder rot) gesetzt. Ansonsten erhält das Quadrat die neutrale Farbe *WEIß* . Der Standardwert für *square_percent* beträgt 51, es müssen also 51 Prozent Bewegungspunkte müssen gefunden werden bevor das Quadrat gesetzt wird.
Als Ergebnis wird dann an die weiterführenden Funktionen die Bildmatrix übergeben. Alle Matrixelemente enthalten dann einen der drei folgenden Werte:

- **Weiß:** Keine Bewegung
- **Blau:** Bewegung im U-Kanal Differenzbild
- **Rot:** Bewegung im V-Kanal Differenzbild

Das Aufstellen der Binärmatrix kostet kaum Rechenzeit. Anstelle die Bewegungspunkte in das Differenzbild einzutragen, werden diese dem entsprechenden Quadrat zugeschlagen. Jeder Bildpunkt wird dabei nur einmal betrachtet. Für die Funktionen zum Finden der Körperteile müssen dann nicht mehr alle Bildpunkte, sondern nur noch die Quadrate der Matrix betrachtet werden. Die Geschwindigkeit steigt somit erheblich. Ein Nachteil dieser vorgehensweise ist natürlich die Vergröberung der Darstellung. Dieser Nachteil fällt jedoch bei den Anforderungen, die ***Ahelp*** an die Bewegungserkennung stellt, nicht sonderlich ins Gewicht.
Um die Geschwindigkeit zu steigern, kann noch die Anzahl der zu betrachtenden Bildpunkte reduziert werden. Über den Parameter *exclude_lines* werden Bildzeilen von der Betrachtung ausgeschlossen. Bei der Einstellung *exclude_lines = 2* wird etwa nur jede zweite Bildzeile betrachtet. Eine weitere Reduktion erfolgt durch den Einsatz des Kalmanfilters . Dieser bietet die Möglichkeit die Person zu tracken , also die Position im aktuellen Bild vorherzusagen. Es wird dann nur noch ein entsprechender Bildausschnitt betrach-

tet. Die Größe des Bildauschnitts wird durch die Variablen *img_searcharea* vorgegeben. Ein Teil des Bildes wird damit von der Betrachtung ausgeschlossen. Alle ausgeschlossenen Quadrate erhalten den Farbwert *GRÜN* . Dieser Farbwert dient lediglich der Visualisierung und entspricht bei der weiteren Bearbeitung der neutralen Farbe Weiß. Bei einem zuverlässigen Einsatz des Kalmanfilters sollte *exclude_lines* auf den Standardwert Eins zurückgesetzt werden (alle Zeilen werden betrachtet), da eine weitere Reduktion der Bildpunkte dann nicht mehr ratsam ist.

5.2.2 FindBody

Die Funktion `FindBody` versucht in der berechneten Binärmatrix den Oberkörper zu finden. Ausgangsbasis sind hierfür alle blauen Matrixelemente, also nur die Bewegungspunkte des U-Kanals. Ein blaues Quadrat wird gesetzt, wenn seinem Umfeld, gegeben durch den Parameter *body_radius*, mindestens *body_square_percent* rote oder blaue Quadrate gefunden wurden. Die gesamte Fläche des Oberkörpers wird dann über alle gesetzten Quadrate gebildet. Die Idee hinter dieser Vorgehensweise ist die folgende: Der Oberkörper sollte nicht aus Punkten des V-Kanals gebildet werden, da dieser ja gerade ein Indiz für menschliche Haut, in diesem Fall also für den Kopf ist. In der Praxis reicht es jedoch nicht aus, nur den U-Kanal zu betrachten. Hierfür wäre es nötig, eine Konvention bezüglich der Kleidung der Zielperson zu treffen. Da dies nicht erwünscht war, kann es sein, daß entweder die Anzahl der blauen Bildpunkte zu gering ist, um den Oberkörper zu bilden, oder daß eine strikte Trennung zwischen dem Oberkörper (blau) und dem Kopf (rot) eben nicht möglich ist. Daher werden für die Entscheidung, ob ein Quadrat für den Oberkörper gesetzt wird, alle Bildpunkte verwendet.

5.2.3 Create_HeadSearchArea

Wurde ein Oberkörper gefunden, wird an den äußersten Enden des Kopfes ein Suchraum (*HeadSearchArea*) gebildet. Grundlage für dieses Vorgehen ist die Tatsache, daß der Kopf der Person in direkter Nähe des Oberkörpers sein muß (zumindest sollte das so sein).

5.2.4 FindHead

Die Funktion `FindHead` sucht in dem gebildeten Suchraum nach roten Quadraten. Wegen des offensichtlichen Zusammenhangs zwischen roten Hautpartien und dem V-Kanal des YUV-Farbmodells können alle Bewegungsquadrate des U-Kanals vernachlässigt werden. Die größte zusammenhängende Fläche wird dann als Kopf der Person angenomnmen.
Gestartet wird die Funktion nur, wenn zuvor ein Oberkörper gefunden wurde. Wegen der relativen Größe des Oberkörpers ist es unwahrscheinlich, daß in einem Bild nur der Kopf sichtbar ist. Ein gefundener Kopf ohne zuvor gefundenen Oberkörper macht daher keinen Sinn.

5.2.5 CreateHeadAndBodyCenter

Bisher liegen Kopf und Oberkörper lediglich als gesamte Fläche vor. Die Funktion `CreateHeadAndBodyCenter` ermittelt nun den, zur weiteren Bearbeitung nötigen, Schwerpunkt. Dieser wird schlicht über die Mittelsenkrechten der gefundenen Flächen approximiert. Abbildung 5.2 zeigt den Ablauf der vorgestellten Funktionen, ausgehend von dem (nur theortisch vorhandenen) Binäerbild, über die Bildung der Binärmatrix, bis hin zur gefundenen Fläche für Kopf und Oberkörper.

5.2.6 Schätzung von Pixelpositionen (KALMAN2D)

Die Klasse ***KALMAN2D*** imlementiert mit der Funktion `recursion` den sogenannten Kalmanalgorithmus zum Schätzen einer neuen Pixelposition (in den Grabberbildern) aus einer gegeben Historie von Vorgängerpunkten. Dazu enthält die Klasse die Varianzen `ss,vv` und die Kovarianz `sv`, welche die Historie der Pixelkoordinatentrajectorie implizit enthalten. Jeder Aufruf der Funktion `recursion` bewirkt ein Updaten der Varianzen und Kovarianz, so daß gewährleistet ist, daß alle Pixelkoordinaten der Trajectorie bei der Schätzung einer neuen Position berücksichtigt werden können. Eine weitere Funktion `init` ermöglicht ein wiederholtes Initialisieren der varianten Kalmanfilterparameter. Dies ist notwendig, wenn der Filter ohne realen Messwert (Pixelkoordinate) eine Schätzung berechnet und die Transformation der Pixelkoordinaten in eine Raumkoordinate dabei einen ***VECTOR3D*** ergibt,

welche außerhalb des im Raummodel definierten Raumes liegt. Das Initialisieren bewirkt dabei indirekt in der Klasse ***CALC***, daß die Bewegungserkennung erneut nicht nur in einem eingeschränkten Teil des Grabberbildes nach der Person sucht, sondern den gesamten Pixelbereich des Kamerabildes als Suchbereich verwendet um neue Startpixelkoordinaten der Schwerpunkte zwecks Initialsierung des ***KALMAN2D*** zu finden.

5.3 Die Analyse der menschlichen Bewegung durch das Objekt *HUMAN*

5.3.1 Aufgabenstellung

Bevor nachfolgend auf die Details der Implementierung eingegangen wird, wollen wir zunächst einmal die eigentliche Aufgabenstellung des Objektes ***HUMAN*** betrachten.

In der Objekthierachie ist ***HUMAN*** zwischen der Informationsgewinnung (d.h. der Extraktion von Informationen aus den Rohbilddaten durch das Objekt ***IMAGE***) und der Alarmgenerierung (d.h. dem Erkennen potentiell gefährlicher Situationen durch das Objekt ***SURVEILLANCE***) angesiedelt. Welche Aufgabe hat das Objekt ***HUMAN*** hierbei zu erfüllen?

Ziel von ***HUMAN*** ist es, die durch ***IMAGE*** gewonnenen Daten (Schwerpunkte, Geschwindigkeit u.ä.) so zu verarbeiten, daß an ***SURVEILLANCE*** eine Liste von Bewegungszuständen (wie z.B. Sitzen, Gehen u.ä.) übergeben werden kann. Diese Liste beinhaltet zu jedem Zustand eine Sicherheit (prozentual), die angibt, mit welcher Sicherheit der Zustand erkannt wurde. Es wurde gefordert, die folgenden 10 Zustände zu erkennen:

Liegen am Boden Liegen (erhöht) Sitzen am Boden Sitzen (erhöht) Rollen (im Rollstuhl) Stehen Gehen Abwärtsbewegung Aufwärtsbewegung Sturz

Zusätzlich soll eine weitere Sicherheit berechnet werden, die angibt, wie zuverlässig die Datengrundlage für die ermittelten Bewegungszustände ist, d.h. wie sicher die von ***IMAGE*** gelieferten Daten sind.

5.3.2 Designideen

Nachdem wir nun das zu erfüllende Aufgabenspektrum von ***HUMAN*** kennengelernt haben, beschäftigen wir uns jetzt mit der Realisierung und Implementierung des Objektes. Dazu werfen wir zunächst einmal einen Blick auf das allgemeine Design des Objektes.

Laut Aufgabenspezifikation erhält ***HUMAN*** die zur Analyse benötigten Daten von ***IMAGE***. Diese werden in einem Objekt vom Typ ***PERSON*** (siehe dort) an ***HUMAN*** übergeben. Die in ***PERSON*** gekapselten Daten sind :

- Zuverlässigkeit der Datengewinnung
- Körperschwerpunktkoordinaten (3D)
- Kopfschwerpunktkoordinaten (3D)
- Bewegungsrichtung (3D-Vektor)
- Bewegungsgeschwindigkeit (3D-Vektor)

Mit Hilfe dieser Daten erfolgt die Analyse der Bewegung. Um die Sicherheiten für jede einzelne Bewegung berechnen zu können, werden die Daten mit Hilfe von Fuzzy-Sets vom Objekttyp ***DOUBLE_FUZZY_SET*** ausgewertet. Jede einzelne Bewegung hat spezielle, auf den Charakter der Bewegung abgestimmte Fuzzy-Sets, die zu dem aktuellen Input die entsprechenden Sicherheiten liefern. Diese berechneten Sicherheiten werden untereinander in Relation gesetzt und liefern so die endgültige Sicherheit für eine bestimmte Bewegung. So werden zum Beispiel zur Ermittlung der Sicherheit für den Bewegungstyp 'Sturz' Fuzzy-Sets für die Bewegungsrichtung und für die Geschwindigkeit mit den entsprechenden Inputvektoren verwendet, die die Wahrscheinlichkeit berechnen, daß die Inputdaten mit einem Sturz korrespondieren. Anschließend erfolgt eine Gewichtung der Sicherheiten im Verhältnis 2 : 1 um die Gesamtsicherheit zu ermitteln. Die Konfiguration der Fuzzy-Sets (d.h. Festlegung des Maximums und Steilheit der Flanken) erfolgt anhand empirisch ermittelter Referenzdaten.

Die ermittelten Daten stellen natürlich nur eine 'Momentaufnahme' der tatsächlichen Bewegung dar. Wie kann nun gewährleistet werden, daß es sich bei den

berechneten Sicherheiten auch um die tatsächliche Bewegung des Menschen handelt, und nicht um eine kurzfristige Abweichung? Eine solche Abweichung könnte durch falsche oder ungenaue Inputdaten erzeugt werden. Um solche Anomalien zu glätten bzw. zu eleminieren, wird eine Speicherung der letzten ermittelten Sicherheiten im Objekttyp ***LHISTORY*** durchgeführt. Bevor die Ergebnisse an ***SURVEILLANCE*** weitergegeben werden, wird der Mittelwert der modifizierten, gespeicherten Sicherheiten und der aktuellen Sicherheiten gebildet. Der Modifikator sorgt dafür, daß länger zurückliegende Ergebnisse schwächer eingehen als aktuellere. Die Anzahl der zur Mittelwertbildung einzubeziehenden Ergebnisse ist abhängig von der Bildanalysegeschwindigkeit des Gesamtsystems. Es sollten maximal Ergebnisse der letzten 2 Sekunden eingehen, d.h. ca. 8-12 berechnete Sicherheiten für jede Bewegung.

Nun, da Sicherheiten für jede Bewegung berechnet wurden, müssen die Ergebnisse an ***SURVEILLANCE*** weitergereicht werden. Hierzu wird der Objekttyp ***HUMAN_ACTION*** verwendet, der die Daten kapselt. Zusätzlich zu den Sicherheiten für die Bewegungen werden außerdem die bereits von ***IMAGE*** berechneten Werte für die Position des Menschen im Raum sowie seine Bewegungsrichtung weitergereicht.

An dieser Stelle hat das Objekt ***HUMAN*** seine in der Aufgabenstellung definierte Funktionalität erreicht und seine Arbeit ist beendet.

Da wir jetzt die grundlegenden Designideen des Objektes ***HUMAN*** kennen, können wir näher auf die Implementierung, d.h. auf die verwendeten Datenstrukturen und Methoden eingehen.

Bei der Instanzierung des Objektes ***HUMAN*** durch das Objekt ***ROOM*** wird zunächst einmal der Konstruktor aufgerufen. Der Konstruktor erhält einen Zeiger auf das Objekt ***ROOM***, der es später ermöglicht, auf ***ROOM*** zuzugreifen. Dies ist wichtig, da ***ROOM*** alle Objekte instanziert und ohne diese Zeiger Zugriffe auf andere Objekte nicht möglich wären. Der Konstruktor selbst initialisiert anschließend alle Fuzzy-Sets mit den benötigten Parametern (für Details siehe ***DOUBLE_FUZZY_SET***). Diese Parameter legen für jeden Fuzzy-Set den maximalen Sicherheitswert und die Steilheit der Flanken fest. Außerdem wird mit Hilfe der ***ROOM*** Methode `get_image` ein Zeiger auf das Objekt ***CALC_PERSON*** von ***IMAGE*** angelegt, das die zur Analyse benötigten Daten kapselt, um später auf diese Daten zugreifen zu können.

Kommen wir nun zur Beschreibung des Ablaufes der einzelnen Methoden, wenn das Objekt ***SURVEILLANCE*** auf die Schnittstellenmethode `get_human_action` zugreift.

Als erstes wird in der Methode `get_human_action` die Methode `get_actual_motion_data` aufgerufen. In dieser Methode wird mit Hilfe des im Konstruktors angelegten Zeigers auf das Objekt ***PERSON_CALC*** die Methode `calculate_person` aufgerufen. Diese liefert die aktuellen Bewegungsdaten, d.h. Schwerpunkte sowie Bewegungsrichtung und -geschwindigkeit, in Form eines Objektes vom Typ ***PERSON*** (siehe dort). `get_actual_motion_data` gibt dieses Objekt ***PERSON*** zurück an die aufrufende Methode.

Da wir die aktuellen Daten von ***IMAGE*** bekommen haben, beginnt die Verarbeitung. Die Methode `analyse_human_action` wird aufgerufen.

Hier werden zunächst einmal die von ***IMAGE*** erhaltenen Daten in eine für die Bewegungsanalyse besser geeignete Form transferiert. Zu diesem Zweck bedient sich analyse_human_action der Methode `check_direction`, `check_speed`, `check_vertical alignment`, `check mass_centre_hight` und `set_no_recognition_value`. Die Methode `check_direction` erhält als Input den aktuellen Bewegungsrichtungsvektor. Dieser Vektor wird zerlegt in den x-y-Achsen und z-Achsen Anteil. Der x- und y-Achsen Anteil wird zum xy-Anteil zusammenaddiert. Die Methode gibt dann ein Tupel (xy,z) mit den entsprechenden Anteilen zurück. Die Methode `check_speed` erhält den Körpergeschwindigkeitsvektor und berechnet mit Hilfe der Objekt ***VECTOR3D*** Funktion length (siehe dort) die Länge des Vektors. Das Ergebnis wird als aktuelle Geschwindigkeit als DOUBLE zurückgegeben. `check_vertical_alignment` bekommt als Übergabeparameter den Kopf- und Körperschwerpunkt. Aus diesen Eingaben berechnet die Funktion die relative Position des Kopfes zum Körper. Gemessen wird die relative Lage durch den Höhenunterschied zwischen Kopf- und Körperschwerpukt bezogen auf die z-Achse. Dies ist sehr aussagekräftig, da sich beim Liegen die beiden Schwerpunkte annähernd auf einer Höhe befinden, während beim Stehen sich ein deutlich meßbarer Höhenunterschied ergibt. Nach der Subtraktion der beiden z-Achsen-Werte wird der Höhenunterschied als Ergebnis zurückgegeben. Die Methode `check_mass_centre_height` ermittelt aus dem 3D-Übergabevektor für den Körperschwerpunkt dessen Höhe und gibt diesen Wert zurück. Die Methode `set_no_recognition_value` ist zuständig für die Ermittlung der Sicherheit der Genauigkeit der von ***IMAGE*** gelieferten Daten. Sie erhält als Übergabeparameter den von ***IMAGE*** berechneten Zuverlässigkeitswert der Daten.

Dieser Zuverlässigkeitswert besteht aus einer zweistelligen Zahl, wobei jede Stelle einen Wertebereich von 0 bis 2 umfaßt. Die Bedeutung der Zahl ist wie folgt:

1. Stelle :

- 0 = Keine Erkennung der Kopfposition möglich
- 1 = Kopfposition geschätzt
- 2 = Kopfposition (zuverlässig) erkannt

2. Stelle :

- 0 = Körperschwerpunkt nicht gefunden
- 1 = Körperschwerpunkt geschätzt
- 2 = Körperschwerpunkt (zuverlässig) erkannt

Daraus berechnet die Methode die Sicherheitswerte. Dabei werden folgende Zuordnungen Zuverlässigkeitswert - Sicherheit getroffen:

Zahl :

- 00 = 0% Genauigkeit der Erkennung
- 01 und 10 = 25%
- 02 und 20 = 40%
- 11 = 50%
- 12 und 21 = 80%
- 22 = 100%

Die so ermittelten Sicherheiten werden anschließend an die aufrufende Methode zurückgegeben.

Nach dieser Vorverarbeitung der erhaltenen Informationen beginnt nun die eigentliche Analyse. Zu diesem Zweck wird für jede der zehn zu erkennenden

Bewegungsarten eine Analysemethode aufgerufen, die die Wahrscheinlichkeit für diese Bewegungsart berechnet. Da die Analysemethoden immer gleich aufgebaut sind, wird im folgenden nur die erste näher beschrieben.

`set_lying_ground_value` ist die erste Analysemethode, die abgearbeitet wird. Hier soll die Sicherheit dafür berechnet werden, daß die aktuellen Bilddaten mit der Aktion - Liegen am Boden - übereinstimmen. Die Methode erhält als Input die Körpergeschwindigkeit (speed), den Höhenunterschied zwischen Kopf- und Körperschwerpunkt (alignment) und die Höhe des Körperschwerpunktes über dem Boden. Da es hier um die Aktion Liegen am Boden geht, sollte man sich fragen, wodurch genau diese Aktion gekennzeichnet ist. Zunächst einmal gibt es (fast) keine Bewegungsgeschwindigkeit des Körpers, der (fast) stationär am Boden liegt. Desweiteren ist für das Liegen charakteristisch, daß sich Kopf- und Körperschwerpunkt annähernd auf der gleichen Höhe (in Bezug auf die z-Achse) befinden. Da wir das Liegen AM BODEN untersuchen, wird sich der Körperschwerpunkt nahe des Bodens befinden (wieder in Bezug auf die z-Achse). Durch diese Charakterisierungen erklärt sich die Wahl der Inputinformation. Diese Werte werden nun durch 3 Fuzzy-Sets analysiert, d.h. jede Information wird durch ein eigenes Fuzzy-Set ausgewertet. Die Geschwindigkeit wird durch *speed_stationary*, die Kopf-Körper-Lage durch *alignment_lying* und die Körperschwerpunkthöhe durch *hight_ground* ausgewertet. Die Sicherheitswerte, die die Fuzzy-Sets liefern, ist dabei i.A. um so größer, je näher die Inputwerte an den für charakteristisch gehaltenen Werten (für die Geschwindigkeit z.B. Null) liegen. Da jedoch die Wertigkeiten der einzelnen Sicherheiten zur Ermittlung der Gesamtsicherheit eines Bewegungstypes nicht gleich groß sind, erfolgt eine Gewichtung der Sicherheiten. Im Falle der Aktion Liegen am Boden ist die Gewichtung 1 (Geschwindigkeit) : 2 (Kopf-Körper-Lage) : 2 (Höhe). Nach der Addition der gewichteten Sicherheiten wird der Mittelwert gebildet, der die Sicherheit für die Aktion repräsentiert. Der durch das oben beschriebene Vorgehen gebildete Wert wird anschließend in ein diskretes Fuzzy-Set (***INT_FUZZY_SET*** *action_list*, Wert 1) geschrieben, in dem alle ermittelten Werte für die einzelnen Bewegungstypen sowie für die Genauigkeit der Bilddaten gespeichert werden.

Diese grundlegende Vorgehensweise wiederholt sich bei der Analyse jedes einzelnen Bewegungstyps. Im folgenden wird deshalb die Betrachtung auf die von der Methode benutzten Inputdaten, die für den Bewegungstyp charakteristisch sind, die verwendeten Fuzzy-Sets und die Gewichtung der Sicher-

heiten beschränkt.

Die Methode `set_lying_value` erhält als Übergabeparameter die Körpergeschwindigkeit (speed), den Höhenunterschied zwischen Kopf und Schwerpunkt (alignment) sowie die Höhe des Schwerpunktes (hight). Diese werden durch die Fuzzy-Sets *speed_stationary*, *alignment_lying* und *hight_lying* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 2 : 2. Das Ergebnis wird in *action_list* als Wert 2 gespeichert.

Die Methode `set_sitting_ground_value` erhält als Übergabeparameter die Körpergeschwindigkeit (speed), den Höhenunterschied zwischen Kopf und Schwerpunkt (alignment) sowie die Höhe des Schwerpunktes (hight). Diese werden durch die Fuzzy-Sets *speed_stationary*, *alignment_sitting_standing* und *hight_ground* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 3 : 2. Das Ergebnis wird in *action_list* als Wert 3 gespeichert.

Die Methode `set_sitting_value` erhält als Übergabeparameter die Körpergeschwindigkeit (speed), den Höhenunterschied zwischen Kopf und Schwerpunkt (alignment) sowie die Höhe des Schwerpunktes (hight). Diese werden durch die Fuzzy-Sets *speed_stationary*, *alignment_sitting_standing* und *hight_sitting* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 3 : 2. Das Ergebnis wird in *action_list* als Wert 4 gespeichert.

Die Methode `set_rolling_value` erhält als Übergabeparameter die Bewegungsrichtung (direction), die Körpergeschwindigkeit (speed), den Höhenunterschied zwischen Kopf und Schwerpunkt (alignment) sowie die Höhe des Schwerpunktes (hight). Diese werden durch die Fuzzy-Sets *direction_walk*, *speed_moving*, *alignment_sitting_standing* und *hight_sitting* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 3 : 2 : 2. Das Ergebnis wird in *action_list* als Wert 5 gespeichert.

Die Methode `set_standing_value` erhält als Übergabeparameter die Körpergeschwindigkeit (speed), den Höhenunterschied zwischen Kopf und Schwerpunkt (alignment) sowie die Höhe des Schwerpunktes (hight). Diese werden durch die Fuzzy-Sets *speed_stationary*, *alignment_sitting_standing* und *hight_standing* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 3 : 2. Das Ergebnis wird in *action_list* als Wert 6 gespeichert.

Die Methode `set_walking_value` erhält als Übergabeparameter die Bewegungsrichtung (direction), die Körpergeschwindigkeit (speed), den Höhenunterschied zwischen Kopf und Schwerpunkt (alignment) sowie die Höhe des Schwerpunktes (hight). Diese werden durch die Fuzzy-Sets *direction_walk*,

speed_moving, *alignment_sitting_standing* und *hight_standing* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 3 : 2 : 2. Das Ergebnis wird in *action_list* als Wert 7 gespeichert.

Die Methode `set_downwards_value` erhält als Übergabeparameter die Bewegungsrichtung (direction) und die Körpergeschwindigkeit (speed). Diese werden durch die Fuzzy-Sets *direction_down* und *speed_downwards* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 2. Das Ergebnis wird in *action_list* als Wert 8 gespeichert.

Die Methode `set_upwards_value` erhält als Übergabeparameter die Bewegungsrichtung (direction) und die Körpergeschwindigkeit (speed). Diese werden durch die Fuzzy-Sets *direction_up* und *speed_upwards* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 2 : 1. Das Ergebnis wird in *action_list* als Wert 9 gespeichert.

Die Methode `set_falling_value` erhält als Übergabeparameter die Bewegungsrichtung (direction) und die Körpergeschwindigkeit (speed). Diese werden durch die Fuzzy-Sets *direction_down* und *speed_fall* ausgewertet. Die Gewichtung der ermittelten Sicherheiten ist 1 : 2. Das Ergebnis wird in *action_list* als Wert 10 gespeichert.

Nach dieser (zugegebenermaßen etwas trockenen) Beschreibung der Auswertung der einzelnen Bewegungszustände wollen wir nun wieder die Methode `analyse_human_action` betrachten, die diese Analysemethoden aufgerufen hat. Da nun alle Werte in *action_list* stehen, gibt die Methode dieses (und auch die Kontrolle) an `get_human_action` zurück.

Bis jetzt wurden jedoch erst die Daten eines einzelnen Bildes analysiert. Um jedoch den Auswerteprozeß robuster gegen (kurzfristig) fehlerhafte Inputdaten von ***IMAGE*** zu machen, sollten auch die Daten der letzten Bilder mit eingehen. Im Zuge dieses Vorgehens werden zunächst einmal die aktuellen Daten in einer Puffer-Datenstruktur, aufgebaut nach dem FIFO Prinzip, gespeichert. Diese Datenstruktur ist die sogenannte ***LHISTORY*** (siehe dort). Das Aktualisieren der History erledigt die Methode `update_action_history`, die als nächstes aufgerufen wird. Sie erhält als Input den ***INT_FUZZY_SET*** *action_list*, der anschließend in der History gespeichert wird. Da in der History Ereignisse unterschiedlichen Formats gespeichert werden können, wird zunächst einmal ein eigener ***EVENT***-Typ, der für unsere Zwecke geeignet ist, angelegt. Hierbei handelt es sich um ***MOTIONEVENT*** (in der Instanz *actual_motions*), der einen ***INT_FUZZY_SET*** und die zusätzlichen

Verwaltungsdaten, die von ***LHISTORY*** benötigt werden, aufnehmen kann. *actual_motions* wird dann mit Hilfe der HISTORY-Methode `put_event` gespeichert. Da diese Methode bei jedem Aufruf von `get_human_actions` ausgeführt wird, befinden sich immer die Daten der letzten 10 (dies ist die - variable - Größe des Ringpuffers) Bilder in der History.

Nach der Speicherung der aktuellen Daten erfolgt in `get_human_action` der Aufruf von `create_final_results`, der Methode, die die endgültigen Werte berechnet.

Die Methode `create_final_results` liest in einer Schleife die Daten von 10 Motionevents aus der History aus, da diese in die Berechnung der endgültigen Sicherheiten einfließen sollen. Da jedoch ältere Datensätze natürlich nicht ein so hohes Gewicht in der Berechnung bekommen sollten wie neuere, werden die Daten mit einem Vergangenheitsmodifikator (modifier) versehen. Der aktuelle Datensatz fließt dabei mit 100% in die Berechnung ein, der vorhergehende mit 85% und jeder weitere mit je 5% weniger. Am Ende wird der Mittelwert der modifizierten Werte gebildet und so die endgültige Sicherheit für jede Bewegung berechnet. Als Letztes werden die Daten in einem Objekt vom Typ ***HUMAN_ACTION***, *final_results*, gekapselt, da dies das Übergabeobjekt an ***SURVEILLANCE*** darstellt. Hierzu wird sowohl der ***INT_FUZZY_SET*** *help_action_list*, der die Sicherheiten für die Bewegungen enthält, als auch die Bewegungsrichtung und die Position des Menschen als 2D-Koordinaten (x- und y-Koordinate) in *final_results* gespeichert und dieses anschließend an `get_human_actions` zurückgegeben.

Wie wir gesehen haben, ist somit die Aufgabe von `get_human_action` erfüllt. Die vom Bedarfsträger ***SURVEILLANCE*** angeforderten Daten stehen bereit und werden jetzt nur noch von der Methode übergeben.

5.3.3 HUMAN_ACTION

Das von ***HUMAN*** zur Datenübergabe an ***SURVEILLANCE*** benutzte Objekt, das die Daten kapselt, ist ***HUMAN_ACTION***. Dieses Objekt beinhaltet Speichervariablen für einen ***INT_FUZZY_SET***, der die ermittelten Sicherheiten aufnimmt, sowie frr die Position des Menschen im Raum (***VECTOR2D***) und seine Bewegungsrichtung (***VECTOR2D***).

HUMAN_ACTION stellt Methoden bereit, mit der die o.g. Daten im Objekt gespeichert und wieder ausgelesen werden können. Die Speicherme-

thoden erhalten jeweils die zu speichernden Daten der o.g. Typen, während die Lesemethoden genau diese Datentypen zurückliefern.

5.4 Generierung des Alarmstatus

Das Modul ***SURVEILLANCE*** analysiert die im Modul ***HUMAN*** erkannten Bewegungen mit Blick auf deren Gefahrenpotential. Es verwendet dabei ***TIMER***, die je nach Gefährlichkeit der erkannten Bewegung unterschiedlich gesetzt werden. Außerdem wird berücksichtigt, wenn der Benutzer den Raum verläßt, oder in Bereiche eintritt, in denen er vom System nicht mehr erkannt werden kann. Die genauen Beziehungen der Timer zu den Bewegungen, sowie die Anpassungsmatrizen zum Übergang von einer Aktion in eine Zweite sollen ermittelt werden, wenn das Grundgerüst steht. Vorerst bleibt uns keine andere Möglichkeit, als zu erraten, welche Timerwerte sinnvoll sein könnten. Der Ablauf der Analyse ist im Diagramm auf Seite 63 zu sehen.

So stand es schon in unserem Zwischenbericht zu lesen. Das Problem ist, daß zwar mittlerweile das Grundgerüst steht, auch scheint mittlerweile die Erfassung der Bewegungsdaten so zu funktionieren, wie wir es wollen, aber ein Problem existiert noch immer: Wir habe noch nicht genügend Testdaten. Außerdem können wir noch keine so hohe Bildrate verarbeiten, wie wir es uns für vernünftige Daten wünschen. Der Effekt dieser Probleme ist, daß im ***SURVEILLANCE***-Modul nach wie vor nur geschätzte Werte verwendet werden für die Timeouts und Alarmzeiten.

Das Modul erfragt sich vom Modul ***HUMAN*** vor der Analyse die aktuelle Position und Aktion des Menschen. Fragen der Sichtbarkeit und eventueller Ausgänge werden mit Hilfe des Moduls ***MODEL*** beantwortet. Ebenso stellt ***MODEL*** Zulässigkeitswerte für bestimmte Bewegungen bzw. Zustände (liegen zum Beispiel). Diese konnten aber bis zum Schreiben dieses Textes leider noch nicht ausgewertet werden, da die Positionsbestimmung im Raum irreale Werte lieferte. Aus dem gleichen Grund konnte auch noch nicht festgestellt werden, wie gut die Erkennung von Aktionen in der Nähe von Ausgängen (genauer: das Verlassen oder Betreten des Raumes) funktioniert.

Am Ende der Analyse steht immer ein bestimmter Alarmstatus, der an das kontrollierende Modul zurückgeliefert wird. Dieser kann einen der folgenden Werte annehmen:

- Alarm
- Voralarm
- Ausgang aus dem Überwachungsbereich
- Ausgang in einen anderen Raum (der Name des anderen Raumes wird mitgeliefert)
- kein Alarm

Wie bereits erwähnt, wird bei einem Wechsel des erkannten Zustandes (oder der erkannten Bewegung) der Timeout angepasst. Diese Anpassung geschieht, entgegen der ursprünglichen Planung nicht mittels eines Arrays, sondern in einem sogenannten Spaghetti-Code (sorry), in einer ziemlich langen `switch`-Anweisung. Während der Implementierung stellte sich nämlich heraus, daß das ursprünglich geplante Array bei weitem nicht so übersichtlich und leicht zu handhaben war, wie angenommen.

In der `Switch`-Anweisung wird zunächst nach der neuen Aktion unterschieden, dann nach der Alten. Diese Struktur erlaubt zumindest subjektiv eine relativ gute Wartbarkeit der verwendeten Timeouts, vor allem war diese Struktur aber auch kürzer als die andere Variante, erst die alte Aktion abzufragen.

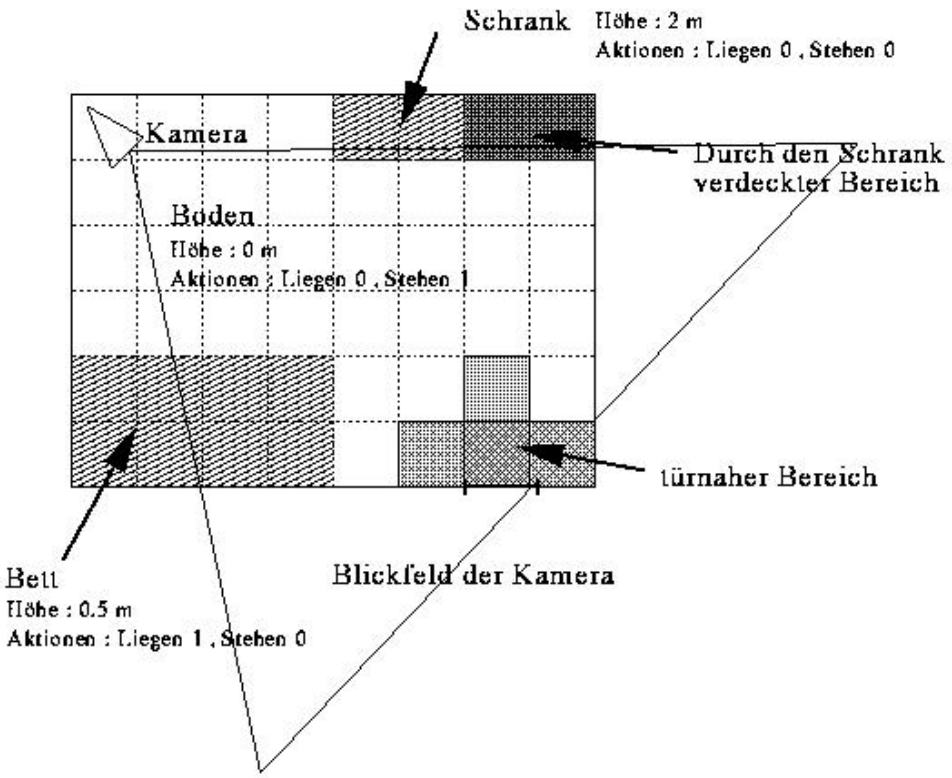

Abbildung 5.1: Grundrißplan

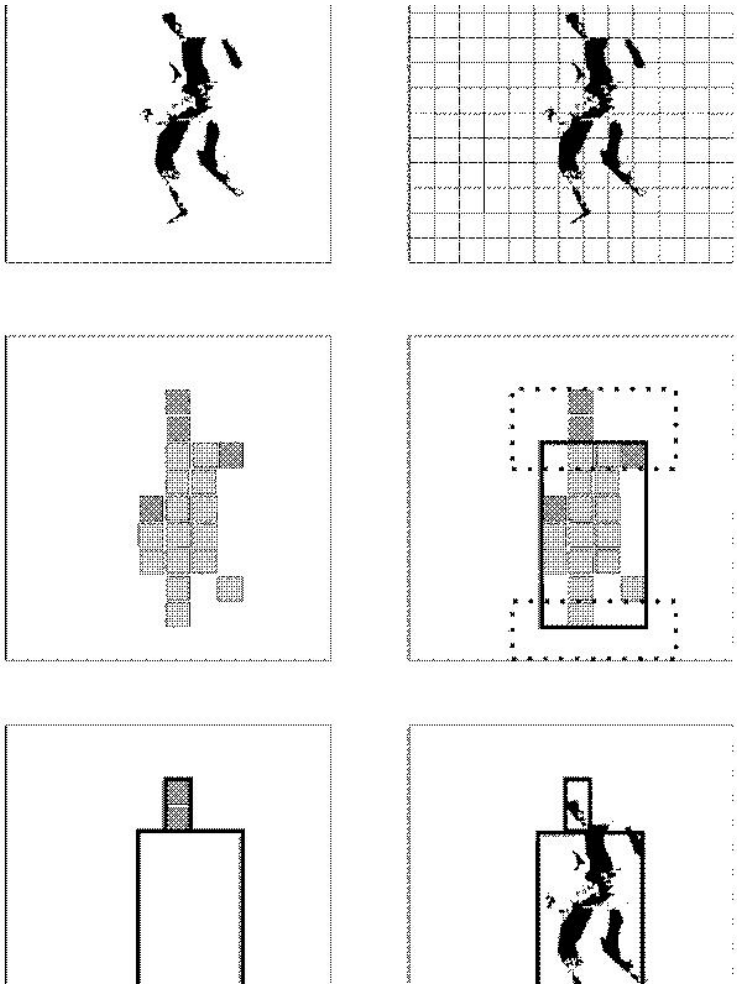

Abbildung 5.2: Beispiel zur Bewegungserkennung

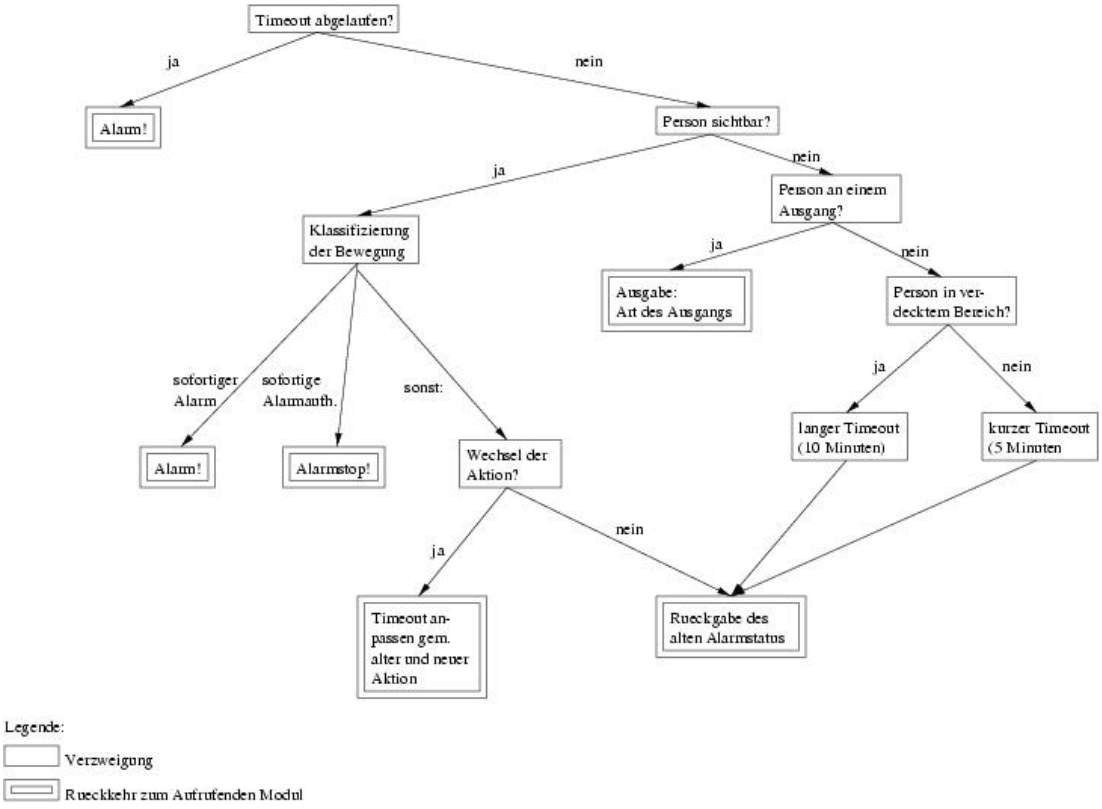

Abbildung 5.3: Der Ablauf der Auswertungen im Surveillance-Modul

Kapitel 6

Ablauf

6.1 Teilnehmer und Zeitraum

An der Entwicklung von ***Ahelp*** haben sieben Informatikstudenten und zwei Studenten der Ingenieurinformatik mitgewirkt. Damit hatte die PG nur neun statt der eigentlich vorgesehenen zwölf Teilnehmer.

Das Projekt umfaßte einen Zeitraum von einem Jahr.

6.2 Projektphasen

Die einzelnen Phasen richteten sich grob nach dem aus der Softwaretechnologie bekannten Projektablauf:

- Ansammlung von Know-how durch eine vorbereitende Seminarphase (damit sich die PG-Teilnehmer besser kennenlernen konnten, wurden die Seminarvorträge während einer dreitägigen Studienreise im Sauerland gehalten)
- Analyse der zu betrachtenden Szenarien und Bestimmung der notwendigen und wünschenswerten Funktionalitäten
- Softwaredesign (Klassenhierarchie und Schnittstellen)

- Implementierung und Test der einzelnen Module
- Integration
- Verfassung des Endberichts

Den zeitlich größten Anteil hatten dabei die Implementierung der Bildverarbeitung und die Integration.

Parallel dazu traten noch einige zusätzliche Aufgaben auf:

- Einrichten der Entwicklungsumgebung (Software, Hardware und Kameras)
- Implementierung der Basisklassen
- Vermessung des Testraums und Aufnahme von Testvideos
- Verfassung des Zwischenberichts

Treffen aller PG-Teilnehmer zwecks Klärung offener Fragen und Vorstellung von Zwischenergebnissen fanden anfangs zweimal und später einmal pro Woche statt.

6.3 Probleme

Auch die PG ***Ahelp*** hatte mit diversen teils selbst verschuldeten und teils unverschuldeten Widrigkeiten zu kämpfen:

- Die PG-Teilnehmer hatten wenig oder gar keine Erfahrung in der Programmierung in C++ und konnten diese erst parallel zur PG sammeln.
- Gelegentlich kam es zu räumlichen und hardwarebezogenen Überschneidungen mit einer weiteren PG, was zu unerwünschten Wartezeiten führte.
- Die benutzten Grabber-Karten verursachten unter Linux erhebliche Lauffähigkeitsprobleme.

- Mangelnde Kommunikation zwischen den PG-Teilnehmern hatte gelegentlich Informationsdefizite und damit verbunden Entwicklungsfehler und Frust zur Folge.
- Nicht ausgiebig genug getestete Module führten während der Integration manchmal zu langwierigen Fehlersuchen.
- Die eigentlich gewünschten Softwareentwicklungswerkzeuge haben wir nicht rechtzeitig erhalten. Folglich musste das Design "per Hand" erfolgen.
- Der vom MIT bestellte "Pfinder" lief nicht auf der vorhandenen Hardware.

6.4 Organisation

Die Organisationsstruktur unserer PG zeichnete sich im Vergleich zu anderen Projekten durch zwei Besonderheiten aus:

1. Auf eine Arbeitsteilung nach funktionalen Gesichtspunkten wurde verzichtet. Das heißt, dass alle PG-Teilnehmer an allen Projektphasen mitgewirkt haben.
   Dadurch wurde die Arbeit in der PG abwechslungsreicher und jeder konnte mehr Erfahrung in Projektorganisation sammeln. Andererseits war diese Form der Teamarbeit mit Sicherheit nicht zeitoptimal.
2. Es gab keinen Projektleiter.
   Dadurch steigerte sich einerseits die Freiheit in der PG andererseits aber auch die Verantwortung jedes PG-Teilnehmers.

Dass die PG trotz fehlenden Drucks (etwa durch einen Vorgesetzten oder durch Benotung) überwiegend reibungslos und erfolgreich verlief, ist in Anbetracht der in der Industrie immer noch vorherrschenden hierarchischen Strukturen als bedeutsam anzusehen.

Eine inhaltliche Arbeitsteilung in drei Gruppen a drei Personen ergab sich durch die oben beschriebene Aufteilung in die Bereiche Bildverarbeitung, Raummodellierung und Menschmodellierung. Zusätzlich anfallende Aufgaben wurden spontan übernommen.

Kapitel 7

Vergleich

Ein Teilziel des zu realisierenden PG-Systems war die Personennachführung der Kameras. Diese Forderung wurde im Prototypen nicht umgesetzt.
Desweiteren sollte eigentlich die Anzahl der im Raum befindlichen Personen beim Start des Systems bestimmt werden , damit danach entschieden werden kann ob das System in Aktion gesetzt werden soll oder ob mögliche weitere Personen im Raum die Überwachung übernehmen sollen. Dieses gesteckte Ziel wurde ebenfalls nicht realisiert. Das System geht initialmäßig davon aus, daß sich lediglich eine Person und zwar die gefährdete Person im Raum befindet. Eine Verwendung des Systems unter einer anderen Bedingung würde zu unbrauchbaren Ergebnissen führen.

Kapitel 8

Ausblick

8.1 Soziale Problematik der Kameraüberwachung

Personen , welche mit ***Ahelp*** überwacht werden sollte man vor der Inbetriebnahme des Systems die Angst vor der ununterbrochenen Überwachung nehmen in dem man ihnen die grundlegende Funktion des Systems erklärt damit ihnen die Kameraüberwachung kein Unwohlsein bringt. Denn gerade ältere Menschen für welche das System im Grunde konzipiert ist haben doch oft eine etwas übersteigerte Technikangst mit entsprechenden Vorurteilen. Vor allen Dingen sollte man den Personen das Gefühl vermitteln, daß ihre Anonymität und ihre Privatsphäre nicht durch die Überwachung in Mitleidenschaft gezogen wird. D.h. die Personen sollten verinnerlichen können, daß das System im Sinne des Datenschutzes nur ihrem eigenen Wohlergehen dient und nicht noch einen anderen Zweck erfüllt.

8.2 Sprachsteuerung

Es wäre durchaus denkbar das System ***Ahelp*** in der Hinsicht zu erweitern, daß es in die Lage versetzt wird auch mögliche akustische Alarmsignale nebenläufig zu der optischen Notfallanalyse zu detektieren. Eine solche Erweiterung des Systems auf zwei Sinne wäre mit Hilfe einer Spracherkennung

oder Lauterkennung unter Verwendung entsprechender Hard- und Software (Mikrofon/Soundkarte und Erkennungssoftware) realisierbar.

Anhang A

Anhang und Grundlagen

A.1 Schnittstellen

A.1.1 INT_FUZZY_SET

Die Klasse ***INT_FUZZY_SET*** implementiert normierte Fuzzy-Mengen auf positiven, ganzen Zahlen. Es werden die Operatoren Durchschnitt, Vereinigung und Komplement unterstützt. Der Durchschnitt wird mittels Minimum-Bildung berechnet, die Vereinigung mittels Maximum-Bildung.

A.1.1.1 Beschreibung

INT_FUZZY_SETS sind als offene Arrays implementiert. Außerdem wird für die nicht explizit im Array definierten Zugehörigkeitswerte ein Zugehörigkeitswert verwaltet (Wichtig bei Komplement-Bildung und clip). Wird ein Zugehörigkeitswert geschrieben, der außerhalb des Arrays liegt, so wird die Größe automatisch angepaßt. Dieser Luxus hat natürlich auch seinen Preis: will man nur Zugehörigkeitswerte für 0 und 10000 abspeichern, so bleiben (wenigstens) 9999 Arrayplätze ungenutzt. Daher sollte man möglichst immer zusammenhängende "Indexbereiche" am Anfang der Menge verwenden. Kennt man den größten vorkommenden Wert schon beim Anlegen des ***INT_FUZZY_SET***, so kann man im Konstruktor-Aufruf die Startgröße explizit angeben (siehe Headerdatei in Abschnitt. So spart man sowohl Spei-

cherplatz als auch Rechenzeit. Die momentane Größe des Arrays kann durch die Funktion `read_size` abgefragt werden.

A.1.1.2 Interface

Methode:	`INT_FUZZY_SET::INT_FUZZY_SET`

Header: `INT_FUZZY_SET(int s = 4)`

s: (i) Startgröße.

Beschreibung: Konstruktor für ***INT_FUZZY_SET***. Setzt den Zugehörigkeitswert aller Elemente der Trägermenge (hier int) auf 0. Da die Größe der verwendeten Datenstruktur dynamisch angepaßt wird, kann die richtige Auswahl der Startgröße eventuell Laufzeit einsparen.

Methode:	`INT_FUZZY_SET::get_mu`

Header: `double get_mu(int i) const`

Rückgabe: Zugehörigkeitswert von i ($\in [0..1]$) .

i: (i) Element der Trägermenge (int).

Beschreibung: Liest Zugehörigkeitswert von i aus.

Methode:	`INT_FUZZY_SET::put_mu`

Header: `int put_mu(int i, double x)`

Rückgabe: 1, falls Aufruf erfolgreich, 0 falls ein Fehler aufgetreten ist.

i: (i) Element der Trägermenge (int). *Achtung: i muß positiv sein!*

x: (i) Zugehörigkeitswert aus [0..1].

Beschreibung: Setzt den Zugehörigkeitswert von i.

Methode:	`INT_FUZZY_SET::hgt`

Header: `double hgt(void) const`

Rückgabe: Höhe des Fuzzy-Sets.

Beschreibung: Liefert die Höhe des Fuzzy-Sets, d.h. das Maximum aller Zugehörigkeitswerte über die gesamte Trägermenge.

Methode:	`INT_FUZZY_SET::read_size`

Header: `int read_size(void) const`

Rückgabe: Größe des Fuzzy-Sets.

Beschreibung: Liefert die Größe des Intervalls, das wirklich gespeichert wird. D.h. für `i` $\geq$ `read_size(i)` gilt `get_mu(i) = 0`.

Methode:	`INT_FUZZY_SET::empty`

Header: `void empty(void)`

Beschreibung: Setzt für alle Elemente der Trägermenge den Zugehörigkeitsgrad auf 0.0.

Methode:	`INT_FUZZY_SET::operator=`

Header: `INT_FUZZY_SET& operator=(const INT_FUZZY_SET&)`

Beschreibung: Zuweisung. Die Menge wird kopiert.

Methode:	`INT_FUZZY_SET::operator*`

Header:
```
friend
INT_FUZZY_SET operator*(const INT_FUZZY_SET& A,
                        const INT_FUZZY_SET& B)
```

Rückgabe: Durchschnitt von A und B

Beschreibung: Berechnet den Durchschnitt von A und B. Für `C = A*B` gilt: `C.get_mu(i) = min(A.get_mu(i),B.get_mu(i))`

Methode:	`INT_FUZZY_SET::operator+`

Header:
```
friend
INT_FUZZY_SET operator+(const INT_FUZZY_SET& A,
                        const INT_FUZZY_SET& B)
```

Rückgabe: Vereinigung von A und B

Beschreibung: Berechnet die Vereinigung von A und B. Für `C = A+B` gilt: `C. get_mu(i) = max(A. get_mu(i),B. get_mu(i))`

Methode:	`INT_FUZZY_SET::union_with_singleton`

Header: `int union_with_singleton(int i, double x)`

Rückgabe: 1, falls Aufruf erfolgreich, 0 falls ein Fehler aufgetreten ist.

i: (i) Element der Trägermenge (int $>= 0$).

x: (i) Zugehörigkeitswert aus [0..1].

Beschreibung: Berechnet die Vereinigung mit der einelementigen Fuzzy-Menge $\{(i, x)\}$. *Achtung: i muß positiv sein!*

Methode:	`INT_FUZZY_SET::operator-`

Header:
```
friend
INT_FUZZY_SET operator-(const INT_FUZZY_SET& A)
```

Rückgabe: Komplement von A.

Beschreibung: Berechnet das Komplement von A.

Methode:	`INT_FUZZY_SET::co`

Header: `void co(void)`

Beschreibung: Berechnet das Komplement einer Fuzzy-Menge.
`A.co();` entspricht `A = -A;`, ist aber schneller.

Methode:	`INT_FUZZY_SET::operator%`

Header:
```
friend
double operator%(const INT_FUZZY_SET& A,
                 const INT_FUZZY_SET& B)
```

Rückgabe: Höhe von $A*B$.

Beschreibung: Entspricht `hgt(A*B)` , ist aber schneller

Methode:	`INT_FUZZY_SET::operator*=`

Header: `INT_FUZZY_SET& operator*=(const INT_FUZZY_SET& A)`

Beschreibung: Durchschnittsbildung mit Zuweisung.
`A *= B` entspricht `A = A*B`, ist aber schneller.

Methode:	`INT_FUZZY_SET::operator+=`

Header: `INT_FUZZY_SET& operator+=(const INT_FUZZY_SET& A)`

Beschreibung: Vereinigung mit Zuweisung.
`A += B` entspricht `A = A+B`, ist aber schneller.

Methode:	`INT_FUZZY_SET::clip`

Header: `int clip(double x)`

Rückgabe: 1, falls Aufruf erfolgreich, 0 falls ein Fehler aufgetreten ist.

x: (i) Höhe, auf der die Fuzzy-Menge abgeschnitten wird ($\in$ [0..1]).

Beschreibung: Schneidet Fuzzy-Menge auf der Höhe x ab. D.h. für `B = clip(A)` gilt `B.get_mu(i) = min(x,A.get_mu(i))`.

Methode:	`INT_FUZZY_SET::operator<<`

Header:
```
friend
ostream& operator<<(ostream&,
                    const INT_FUZZY_SET&)
```

Beschreibung: Gibt eine Fuzzy-Menge in einen `ostream` aus.

Methode:	`INT_FUZZY_SET::operator>>`

Header:
```
friend
istream& operator>>(istream&,
                    INT_FUZZY_SET&)
```

Beschreibung: Liest Fuzzy-Menge aus einem `istream`.
Syntax-Beispiel:
`{ (1,0.3) , (3,0.7) , ... , (7,0.1) ; 0.1 }`
Der Wert nach dem Semikolon gibt den Zugehörigkeitswert für alle Elemente der Trägermenge an, die nicht explizit definiert wurden.

A.1.2 DOUBLE_FUZZY_SET

Die Klasse ***DOUBLE_FUZZY_SET*** dient zur Darstellung von Fuzzy-Mengen auf den reellen Zahlen mit trapezförmigen Zugehörigkeitsfunktionen. Da i.A. das Ergebnis der Vereinigung von zwei trapezförmigen Fuzzy-Mengen nicht trapezförmig ist, wird die Vereinigung nicht unterstützt. Ähnliches gilt für die Komplement-Operation.

A.1.2.1 Beschreibung

Ein ***DOUBLE_FUZZY_SET*** wird durch die Parameter seiner Zugehörigkeitsfunktion bestimmt. Eine Trapez-Funktion wird durch fünf Parameter lt, lr, rr, rt, h definiert, die folgende Bedingungen erfüllen:

$$lt \leq lr \leq rr \leq rt \; ; \; h \in [0, 1]$$

Die Bedeutung der Parameter läßt sich am einfachsten durch eine Skizze veranschaulichen (siehe Abbildung A.1). D.h. man kann nicht den Zugehörig-

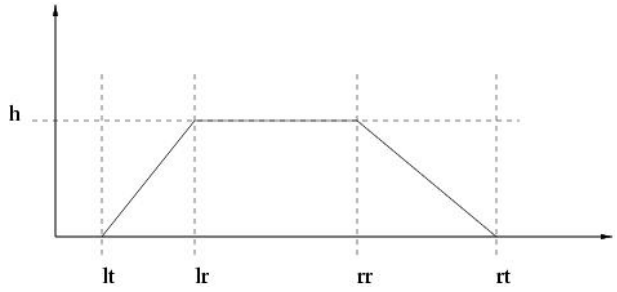

Abbildung A.1: Parameter eines ***DOUBLE_FUZZY_SET***

keitswert einzelner reeller Zahlen verändern, sondern kann nur die Parameter der Zugehörigkeitsfunktion setzen.

Auch der Durchschnitt von zwei trapezförmigen Zugehörigkeitsfunktionen liefert nicht immer eine Trapez-Funktion. Wie in diesem Fall das Ergebnis approximiert wird läßt sich wieder am besten durch eine Skizze darstellen (siehe Abbildung A.2).

A.1.2.2 Interface

Methode:	`DOUBLE_FUZZY_SET::DOUBLE_FUZZY_SET`

Header: `DOUBLE_FUZZY_SET(void)`

Beschreibung: Erzeugt leere Fuzzy-Menge mit `lt=lr=rr=rt=h=0`

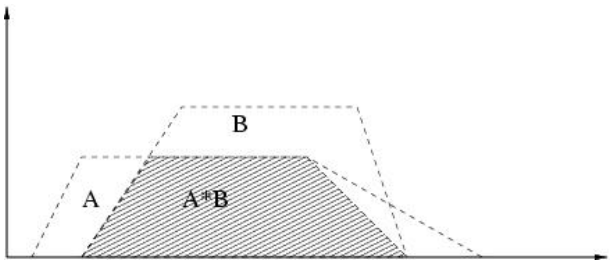

Abbildung A.2: Durchschnitt von zwei ***DOUBLE_FUZZY_SETS***

Methode:	`DOUBLE_FUZZY_SET::get_mu`

Header: `double get_mu(double x) const`

Rückgabe: Zugehörigkeitswert von x ($\in [0..1]$) .

x: (i) Element der Trägermenge (double).

Beschreibung: Liest Zugehörigkeitswert von x aus.

Methode:	`DOUBLE_FUZZY_SET::set_parameter`

Header:
```
int set_parameter(double lt,
                  double lr,
                  double rr,
                  double rt,
                  double h)
```

Rückgabe: 1, falls Aufruf erfolgreich, 0 falls ein Fehler aufgetreten ist.

Beschreibung: Setzt die Parameter der Zugehörigkeitsfunktion. Vorbedingung:
`lt` $\leq$ `lr` $\leq$ `rr` $\leq$ `rt` , `h` $\in [0, 1]$.

Methode:	`DOUBLE_FUZZY_SET::hgt`

Header: `double hgt(void) const`

Rückgabe: Höhe des Fuzzy-Sets.

Beschreibung: Liefert die Höhe des Fuzzy-Sets, d.h. das Maximum aller Zugehörigkeitswerte über die gesamte Trägermenge.

Methode:	`DOUBLE_FUZZY_SET::read_lt`

Header: `double read_lt(void) const`

Rückgabe: Parameter `lt`.

Beschreibung: Liest Parameter `lt`

Methode:	`DOUBLE_FUZZY_SET::read_lr`

Header: `double read_lr(void) const`

Rückgabe: Parameter `lr`.

Beschreibung: Liest Parameter `lr`

Methode:	`DOUBLE_FUZZY_SET::read_rr`

Header: `double read_rr(void) const`

Rückgabe: Parameter `rr`.

Beschreibung: Liest Parameter `rr`

Methode:	`DOUBLE_FUZZY_SET::read_rt`

Header: `double read_rt(void) const`

Rückgabe: Parameter `rt`.

Beschreibung: Liest Parameter `rt`

Methode:	`DOUBLE_FUZZY_SET::read_h`

Header: `double read_h(void) const`

Rückgabe: Parameter `h`.

Beschreibung: Liest Parameter `h`

Methode:	`DOUBLE_FUZZY_SET::operator=`

Header:
```
DOUBLE_FUZZY_SET&
operator=(const DOUBLE_FUZZY_SET&)
```

Beschreibung: Zuweisung. Die Menge wird kopiert.

Methode:	`DOUBLE_FUZZY_SET::operator*`

Header:
```
friend
DOUBLE_FUZZY_SET
operator*(const DOUBLE_FUZZY_SET& A,
          const DOUBLE_FUZZY_SET& B)
```

Rückgabe: Durchschnitt von A und B

Beschreibung: Berechnet den Durchschnitt von A und B (siehe A.1.2.1).

Methode:	`DOUBLE_FUZZY_SET::operator*=`

Header:
```
DOUBLE_FUZZY_SET&
operator*=(const DOUBLE_FUZZY_SET& A)
```

Beschreibung: Durchschnittsbildung mit Zuweisung.
`A *= B` entspricht `A = A*B`, ist aber schneller.

Methode:	`DOUBLE_FUZZY_SET::operator%`

Header: `friend`
`double operator%(const DOUBLE_FUZZY_SET& A,`
`const DOUBLE_FUZZY_SET& B)`

Rückgabe: Höhe von $A*B$.

Beschreibung: Entspricht `hgt(A*B)` , ist aber schneller

Methode:	`DOUBLE_FUZZY_SET::clip`

Header: `int clip(double x)`

Rückgabe: 1, falls Aufruf erfolgreich, 0 falls ein Fehler aufgetreten ist.

x: (i) Höhe, auf der die Fuzzy-Menge abgeschnitten wird ($\in$ [0..1]).

Beschreibung: Schneidet Fuzzy-Menge auf der Höhe x ab. D.h. für `B = clip(A)` gilt `B.get_mu(i) = min(x,A.get_mu(i))`.

Methode:	`DOUBLE_FUZZY_SET::operator<<`

Header: `friend`
`ostream& operator<<(ostream&,`
`const DOUBLE_FUZZY_SET&)`

Beschreibung: Gibt eine Fuzzy-Menge in einen `ostream` aus.

Methode:	`DOUBLE_FUZZY_SET::operator>>`

Header: `friend`
`istream& operator>>(istream&,`
`DOUBLE_FUZZY_SET&)`

Beschreibung: Liest Fuzzy-Menge aus einem `istream`.
Syntax-Beispiel:
`(lt , lr , rr , rt , h)`

A.1.3 VECTOR2D

Die Klasse ***VECTOR2D*** implementiert Standardfunktionen und überladene Operatoren zur Berechnung von 2D-Vektoren. Die Methoden und Operatoren der Klasse erlauben Zuweisung, Addition, Subtraktion, Multiplikation zweier ***VECTOR2D*** sowie einer ***MATRIX2D*** und eines ***VECTOR2D***, Normalisierung, Skalierung und Längenbetragsbildung.

A.1.3.1 Beschreibung

Die Komponenten eines ***VECTOR2D*** sind innerhalb der Klasse gekapselt und können über definierte Komponentenauslese- und schreibmethoden abgefragt und verändert werden. Der Standardkonstruktor ohne Parameter initialisiert einen ***VECTOR2D*** mit dem Nullvektor. Die Parameterversion des Konstruktors definiert einen beliebig festlegbaren ***VECTOR2D***.

A.1.3.2 Interface

Methode:	`VECTOR2D::VECTOR2D`

Header: `VECTOR2D(void)`

Beschreibung: Konstruktor für ***VECTOR2D***. Setzt die x- und y-Komponente auf Null.

Methode:	`VECTOR2D::VECTOR2D`

Header: `VECTOR2D(const double &xval,const double &yval)`

xval: (i) X-Komponente des ***VECTOR2D***.

yval: (i) Y-Komponente des ***VECTOR2D***.

Beschreibung: Konstruktor für ***VECTOR2D***. Initialisert die Komponenten entsprechend den Inputparametern.

Methode:	`VECTOR2D::VECTOR2D`

Header: `inline VECTOR2D(const VECTOR2D&)`

Beschreibung: Kopiert einen ***VECTOR2D*** in einen anderen ***VECTOR2D***.

Methode:	`VECTOR2D::read_x`

Header: `double read_x(void) const`

Rückgabe: X-Komponente des ***VECTOR2D***.

Beschreibung: Liefert aus dem ***VECTOR2D*** die X-Komponente.

Methode:	`VECTOR2D::read_y`

Header: `double read_y(void) const`

Rückgabe: Y-Komponente des ***VECTOR2D***.

Beschreibung: Liefert aus dem ***VECTOR2D*** die Y-Komponente.

Methode:	`VECTOR2D::write_x`

Header: `void write_x(double)`

..: (i) X-Komponente des ***VECTOR2D***.

Beschreibung: Verändert die X-Komponente des ***VECTOR2D***

Methode:	`VECTOR2D::write_y`

Header: `void write_y(double)`

..: (i) Y-Komponente des ***VECTOR2D***.

Beschreibung: Verändert die Y-Komponente des ***VECTOR2D***

Methode:	`VECTOR2D::operator=`

Header: `VECTOR2D& operator=(const VECTOR2D& v)`

Rückgabe: Der ***VECTOR2D*** v.

Beschreibung: Setzt einen ***VECTOR2D*** gleich einem anderen ***VECTOR2D***.

Methode:	`VECTOR2D::operator+`

Header:
```
friend
VECTOR2D operator+(const VECTOR2D&,
                   const VECTOR2D&)
```

Rückgabe: Die Summe der zwei ***VECTOR2D***.

Beschreibung: Addiert zwei ***VECTOR2D***.

Methode:	`VECTOR2D::operator−`

Header:
```
friend
VECTOR2D operator−(const VECTOR2D&,
                   const VECTOR2D&)
```

Rückgabe: Die Differenz der zwei ***VECTOR2D***.

Beschreibung: Subtrahiert zwei ***VECTOR2D***.

Methode:	`VECTOR2D::operator−`

Header:
```
friend
VECTOR2D operator−(const VECTOR2D&)
```

Rückgabe: Der negative ***VECTOR2D***.

Beschreibung: Negiert einen ***VECTOR2D***.

Methode:	`VECTOR2D::operator*`

Header: `friend`
`double operator*(const VECTOR2D&,`
`const VECTOR2D&)`

Rückgabe: Das Skalarprodukt der zwei ***VECTOR2D***.

Beschreibung: Berechnet das Skalarprodukt zweier ***VECTOR2D***.

Methode:	`VECTOR2D::operator*`

Header: `friend`
`VECTOR2D operator*(const MATRIX2D&,`
`const VECTOR2D&)`

Rückgabe: Produkt aus einer ***MATRIX2D*** und einem ***VECTOR2D***.

Beschreibung: Multipliziert eine ***MATRIX2D*** mit einem ***VECTOR2D***.

Methode:	`VECTOR2D::scale`

Header: `void scale(const double &f)`

f: (i) skalarer Wert.

Beschreibung: Skaliert eine ***VECTOR2D*** mit einem Skalar.

Methode:	`VECTOR2D::normalize`

Header: `void normalize(void)`

Beschreibung: Skaliert eine ***VECTOR2D*** mit dem Reziprokwert des entsprechenden Längenbetrages.

Methode:	`VECTOR2D::length`

Header: `double length(void) const`

Rückgabe: Der Längenbetrag einen ***VECTOR2D***.

Beschreibung: Berechnet den Längenbetrag eines ***VECTOR2D***.

Methode:	`VECTOR2D::operator>>`

Header:
```
friend
istream& operator>>(istream& is,
                    VECTOR2D& v)
```

Beschreibung: Schreibt einen ***VECTOR2D*** in einen `istream`.

Methode:	`VECTOR2D::operator<<`

Header:
```
friend
ostream& operator<<(ostream& is,
                    VECTOR2D& v)
```

Beschreibung: Liest einen ***VECTOR2D*** aus einen `ostream`.

A.1.4 VECTOR3D

Die Klasse ***VECTOR3D*** implementiert Standardfunktionen und überladene Operatoren zur Berechnung von 3D-Vektoren. Die Methoden und Operatoren der Klasse erlauben Zuweisung, Addition, Subtraktion, Multiplikation zweier ***VECTOR3D*** in Form des Skalarprodukts und des Kreuzprodukts sowie Multiplikation einer ***MATRIX3D*** und eines ***VECTOR3D***, Normalisierung, Skalierung und Längenbetragsbildung.

A.1.4.1 Beschreibung

Die Komponenten eines ***VECTOR3D*** sind innerhalb der Klasse gekapselt und können über definierte Komponentenauslese- und schreibmethoden abgefragt und verändert werden. Der Standardkonstruktor ohne Parameter initialisiert einen ***VECTOR3D*** mit dem Nullvektor. Die Parameterversion des Konstruktors definiert einen beliebig festlegbaren ***VECTOR3D***.

A.1.4.2 Interface

Methode:	`VECTOR3D::VECTOR3D`

Header: `VECTOR3D(void)`

Beschreibung: Konstruktor für ***VECTOR3D***. Setzt die x- und y- und z-Komponente auf Null.

Methode:	`VECTOR3D::VECTOR3D`

Header: `VECTOR3D(const double &xval,const double &yval,const double &zval)`

xval: (i) X-Komponente des ***VECTOR3D***.

yval: (i) Y-Komponente des ***VECTOR3D***.

zval: (i) Z-Komponente des ***VECTOR3D***.

Beschreibung: Konstruktor für ***VECTOR3D***. Initialisert die Komponenten entsprechend den Inputparametern.

Methode:	`VECTOR3D::VECTOR3D`

Header: `VECTOR3D(const VECTOR3D&)`

Beschreibung: Kopiert einen ***VECTOR3D*** in einen anderen ***VECTOR3D***.

Methode:	`VECTOR3D::read_x`

Header: `double read_x(void) const`

Rückgabe: X-Komponente des ***VECTOR3D***.

Beschreibung: Liefert aus dem ***VECTOR3D*** die X-Komponente.

Methode:	`VECTOR3D::read_y`

Header: `double read_y(void) const`

Rückgabe: Y-Komponente des ***VECTOR3D***.

Beschreibung: Liefert aus dem ***VECTOR3D*** die Y-Komponente.

Methode:	`VECTOR3D::read_z`

Header: `double read_z(void) const`

Rückgabe: Z-Komponente des ***VECTOR3D***.

Beschreibung: Liefert aus dem ***VECTOR3D*** die Z-Komponente.

Methode:	`VECTOR3D::write_x`

Header: `void write_x(double)`

..: (i) X-Komponente des ***VECTOR3D***.

Beschreibung: Verändert die X-Komponente des ***VECTOR3D***

Methode:	`VECTOR3D::write_y`

Header: `void write_y(double)`

..: (i) Y-Komponente des ***VECTOR3D***.

Beschreibung: Verändert die Y-Komponente des ***VECTOR3D***

Methode:	`VECTOR3D::write_z`

Header: `void write_y(double)`

..: (i) Z-Komponente des ***VECTOR3D***.

Beschreibung: Verändert die Z-Komponente des ***VECTOR3D***

Methode:	`VECTOR3D::operator=`

Header: `VECTOR3D& operator=(const VECTOR3D& v)`

Rückgabe: Der ***VECTOR3D*** v.

Beschreibung: Setzt einen ***VECTOR3D*** gleich einem anderen ***VECTOR3D***.

Methode:	`VECTOR3D::operator+`

Header:
```
friend
VECTOR3D operator+(const VECTOR3D&,
                   const VECTOR3D&)
```

Rückgabe: Die Summe der zwei ***VECTOR3D***.

Beschreibung: Addiert zwei ***VECTOR3D***.

Methode:	`VECTOR3D::operator−`

Header:
```
friend
VECTOR3D operator−(const VECTOR3D&,
                   const VECTOR3D&)
```

Rückgabe: Die Differenz der zwei ***VECTOR3D***.

Beschreibung: Subtrahiert zwei ***VECTOR3D***.

Methode:	`VECTOR3D::operator−`

Header:
```
friend
VECTOR3D operator−(const VECTOR3D&)
```

Rückgabe: Der negative ***VECTOR3D***.

Beschreibung: Negiert einen ***VECTOR3D***.

Methode:	`VECTOR3D::operator*`

Header:
```
friend
double operator*(const VECTOR3D&,
                 const VECTOR3D&)
```

Rückgabe: Das Skalarprodukt der zwei ***VECTOR3D***.

Beschreibung: Berechnet das Skalarprodukt zweier ***VECTOR3D***.

Methode:	`VECTOR3D::operator^`

Header:
```
friend
VECTOR3D operator^(const VECTOR3D&,
                   const VECTOR3D&)
```

Rückgabe: Der auf beiden ***VECTOR3D*** orthogonal stehende ***VECTOR3D***.

Beschreibung: Berechnet das Kreuzprodukt zweier ***VECTOR3D***.

Methode:	`VECTOR3D::operator*`

Header:
```
friend
VECTOR3D operator*(const MATRIX3D&,
                   const VECTOR3D&)
```

Rückgabe: Produkt aus einer ***MATRIX3D*** und einem ***VECTOR3D***.

Beschreibung: Multipliziert eine ***MATRIX3D*** mit einem ***VECTOR3D***.

Methode:	`VECTOR3D::scale`

Header: `void scale(const double &f)`

f: (i) skalarer Wert.

Beschreibung: Skaliert eine ***VECTOR3D*** mit einem Skalar.

Methode:	`VECTOR3D::normalize`

Header: `void normalize(void)`

Beschreibung: Skaliert eine ***VECTOR3D*** mit dem Reziprokwert des entsprechenden Längenbetrages.

Methode:	`VECTOR3D::length`

Header: `double length(void) const`

Rückgabe: Der Längenbetrag einen ***VECTOR3D***.

Beschreibung: Berechnet den Längenbetrag eines ***VECTOR3D***.

Methode:	`VECTOR3D::operator>>`

Header:
```
friend
istream& operator>>(istream& is,
                    VECTOR3D& v)
```

Beschreibung: Schreibt einen ***VECTOR3D*** in einen `istream`.

Methode:	`VECTOR3D::operator<<`

Header:
```
friend
ostream& operator<<(ostream& is,
                    VECTOR3D& v)
```

Beschreibung: Liest einen ***VECTOR3D*** aus einen `ostream`.

A.1.5 MATRIX2D

Die Klasse ***MATRIX2D*** implementiert Standardfunktionen und überladene Operatoren zur Berechnung von 2D-Matrizen. Die Methoden und Operatoren der Klasse erlauben Zuweisung, Addition, Subtraktion, Multiplikation zweier ***MATRIX2D*** sowie Skalierung, Invertierung , Festlegung einer Rotationsmatrix und Berechnung der Determinante.

A.1.5.1 Beschreibung

Die Komponenten einer ***MATRIX2D*** sind innerhalb der Klasse gekapselt und können über definierte Komponentenauslese- und schreibmethoden abgefragt und verändert werden. Der Standardkonstruktor ohne Parameter initialisiert eine ***MATRIX2D*** als Einheitsmatrix. Die Parameterversion des Konstruktors definiert eine beliebig festlegbare ***MATRIX2D***.

A.1.5.2 Beschreibung

Methode:	`MATRIX2D::MATRIX2D`

Header: `MATRIX2D(void)`

Beschreibung: Konstruktor für ***MATRIX2D***. Setzt die diagonalen Komponenten auf 1 und alle anderen Komponenten auf Null.

Methode:	`MATRIX2D::MATRIX2D`

Header:
```
inline
MATRIX2D(const double &c11val,const double &c12val,
         const double &c21val,const double &c22val)
```

c11val,...,c22val: (i) Koeffienten der ***MATRIX2D***.

Beschreibung: Kontruktor für ***MATRIX2D***. Initialisiert die Komponenten entsprechend den Inputparametern.

Methode:	`MATRIX2D::MATRIX2D`

Header: `inline MATRIX2D(const MATRIX2D& m)`

Beschreibung: Kopiert eine ***MATRIX2D*** in eine andere ***MATRIX2D***.

Methode:	`MATRIX2D::scale`

Header: `inline void scale(const double &f)`

f: (i) Skalarer Wert.

Beschreibung: Multipliziert die Koeffizienten einer ***MATRIX2D*** mit dem Inputskalar.

Methode:	`MATRIX2D::invert`

Header: `int invert(void)`

Rückgabe: Wahrheitswert , welcher für die Invertierbarkeit der ***MATRIX2D*** steht.

Beschreibung: Invertiert die ***MATRIX2D***.

Methode:	`MATRIX2D::det`

Header: `double det(void)`

Rückgabe: Der zur ***MATRIX2D*** gehörige Determinantenwert.

Beschreibung: Berechnet die Determinante der ***MATRIX2D***.

Methode:	`MATRIX2D::Define_rotation`

Header: `void Define_rotation(const double &angle)`

angle: (i) Rotationswinkel.

Beschreibung: Berechnet die Rotations- ***MATRIX2D*** für eine Rotation eines ***VECTOR2D*** um den Winkel `angle` im Gegenuhrzeigersinn um den Ursprung des Koordinatensystems.

Methode:	`MATRIX2D::read_c`

Header: `double read_c(int i,int j) const`

Rückgabe: Koeffizient der ***MATRIX2D*** mit Index `i` und `j`.

i: (i) Horizontaler Index (1,2).

j: (i) Vertikaler Index (1,2).

Beschreibung: Liefert den Koeffizient der ***MATRIX2D*** mit dem Index `i` und `j`.

Methode:	`MATRIX2D::write_c`

Header: `void write_c(int i,int j,double cval)`

i: (i) Horizontaler Index (1,2).

j: (i) Vertikaler Index (1,2).

cval: (i) Koeffizientenwert.

Beschreibung: Schreibt in die ***MATRIX2D*** an die Stelle mit dem Index `i` und `j` den Koeffizientenwert `cval`.

Methode:	`MATRIX2D::operator+`

Header:
```
friend
MATRIX2D operator+(const MATRIX2D&,
                   const MATRIX2D&)
```

Rückgabe: Die Summe der zwei ***MATRIX2D***.

Beschreibung: Addiert zwei ***MATRIX2D***.

Methode:	`MATRIX2D::operator-`

Header:
```
friend
MATRIX2D operator-(const MATRIX2D&,
                   const MATRIX2D&)
```

Rückgabe: Die Differenz der zwei ***MATRIX2D***.

Beschreibung: Subtrahiert zwei ***MATRIX2D***.

Methode:	`MATRIX2D::operator−`

Header:
```
friend
MATRIX2D operator−(const MATRIX2D&)
```

Rückgabe: Der negative ***MATRIX2D***.

Beschreibung: Negiert eine ***MATRIX2D***.

Methode:	`MATRIX2D::operator∗`

Header:
```
friend
double operator∗(const MATRIX2D&,
                 const MATRIX2D&)
```

Rückgabe: Das Produkt der zwei ***MATRIX2D***.

Beschreibung: Berechnet das Produkt zweier ***MATRIX2D***.

A.1.6 MATRIX3D

Die Klasse ***MATRIX3D*** implementiert Standardfunktionen und überladene Operatoren zur Berechnung von 2D-Matrizen. Die Methoden und Operatoren der Klasse erlauben Zuweisung, Addition, Subtraktion, Multiplikation zweier ***MATRIX3D*** , Skalierung, Invertierung und Berechnung der Determinante.

A.1.6.1 Beschreibung

Die Komponenten einer ***MATRIX3D*** sind innerhalb der Klasse gekapselt und können über definierte Komponentenauslese- und schreibmethoden abgefragt und verändert werden. Der Standardkonstruktor ohne Parameter initialisiert eine ***MATRIX3D*** als Einheitsmatrix. Die Parameterversion des Konstruktors definiert eine beliebig festlegbare ***MATRIX3D***.

A.1.6.2 Beschreibung

Methode:	`MATRIX3D::MATRIX3D`

Header: `MATRIX3D(void)`

Beschreibung: Konstruktor für ***MATRIX3D***. Setzt die diagonalen Komponenten auf 1 und alle anderen Komponenten auf Null.

Methode:	`MATRIX3D::MATRIX3D`

Header:
```
inline
MATRIX3D(double c11val,double c12val,double c13val,
         double c21val,double c22val,double c23val,
         double c31val,double c32val,double c33val)
```

c11val,...,c33val: (i) Koeffienten der ***MATRIX3D***.

Beschreibung: Kontruktor für ***MATRIX3D***. Initialisiert die Komponenten entsprechend den Inputparametern.

Methode:	`MATRIX3D::MATRIX3D`

Header: `MATRIX3D(int xyz,double angle)`

xyz: (i) Indexwert für die Achse für welche eine Rotationmatrix definiert werden soll..

angle: (i) Rotationswinkel.

Beschreibung: Berechnet eine Rotations-***MATRIX3D*** für eine Rotation eines ***VECTOR3D*** um eine durch den Parameterwert xyz definierte Achse im Gegenuhrzeigersinn und legt die entsprechenden Koeffizienten im zu instanziierenden Objekt der Klasse ***MATRIX3D*** ab.

Methode:	`MATRIX3D::MATRIX3D`

Header: `inline MATRIX3D(const MATRIX3D& m)`

Beschreibung: Kopiert eine ***MATRIX3D*** in eine andere ***MATRIX3D***.

Methode:	`MATRIX3D::scale`

Header: `inline void scale(const double &f)`

f: (i) Skalarer Wert.

Beschreibung: Multipliziert die Koeffizienten einer ***MATRIX3D*** mit dem Inputskalar.

Methode:	`MATRIX3D::invert`

Header: `int invert(void)`

Rückgabe: Wahrheitswert , welcher für die Invertierbarkeit der ***MATRIX3D*** steht.

Beschreibung: Invertiert die ***MATRIX3D***.

Methode:	`MATRIX3D::det`

Header: `double det(void)`

Rückgabe: Der zur ***MATRIX3D*** gehörige Determinantenwert.

Beschreibung: Berechnet die Determinante der ***MATRIX3D***.

Methode:	`MATRIX3D::read_c`

Header: `double read_c(int i,int j) const`

Rückgabe: Koeffizient der ***MATRIX3D*** mit Index `i` und `j`.

i: (i) Horizontaler Index (1,2,3).

j: (i) Vertikaler Index (1,2,3).

Beschreibung: Liefert den Koeffizient der ***MATRIX3D*** mit dem Index `i` und `j`.

Methode:	`MATRIX3D::write_c`

Header: `void write_c(int i,int j,double cval)`

i: (i) Horizontaler Index (1,2,3).

j: (i) Vertikaler Index (1,2,3).

cval: (i) Koeffizientenwert.

Beschreibung: Schreibt in die ***MATRIX3D*** an die Stelle mit dem Index `i` und `j` den Koeffizientenwert `cval`.

Methode:	`MATRIX3D::operator+`

Header:
```
friend
MATRIX3D operator+(const MATRIX3D&,
                   const MATRIX3D&)
```

Rückgabe: Die Summe der zwei ***MATRIX3D***.

Beschreibung: Addiert zwei ***MATRIX3D***.

Methode:	`MATRIX3D::operator−`

Header:
```
friend
MATRIX3D operator−(const MATRIX3D&,
                   const MATRIX3D&)
```

Rückgabe: Die Differenz der zwei ***MATRIX3D***.

Beschreibung: Subtrahiert zwei ***MATRIX3D***.

Methode:	`MATRIX3D::operator−`

Header: `friend`
`MATRIX3D operator−(const MATRIX3D&)`

Rückgabe: Der negative ***MATRIX3D***.

Beschreibung: Negiert eine ***MATRIX3D***.

Methode:	`MATRIX3D::operator∗`

Header:
```
friend
double operator∗(const MATRIX3D&,
                 const MATRIX3D&)
```

Rückgabe: Das Produkt der zwei ***MATRIX3D***.

Beschreibung: Berechnet das Produkt zweier ***MATRIX3D***.

A.1.7 TIME

Diese Klasse ***TIME*** stellt die Grundfunktionen zur Zeitmessung zur Verfügung. Sie wird von der Klasse ***TIMER*** benutzt.

A.1.7.1 Interface

Methode:	`TIME::TIME`

Header: `TIME()`

Beschreibung: Konstruktor der Klasse

Methode:	`TIME::TIME`

Header: `TIME(long int sec_ip)`

sec_ip: (i) aktuelle Zeitangabe, Sekunden seit dem 1.1.1970

Beschreibung: Konstruktor, der mit einer Initialzeit aufgerufen wird

Methode:	`TIME:: TIME`

Header: `TIME()`

Beschreibung: Destruktor der Klasse

Methode:	`TIME::TIME`

Header: `TIME(const TIME &old_TIME)`

old_TIME: (i) Zeiger auf ein Zeitobjekt

Beschreibung: Copy-Konstruktor

Methode:	`TIME::put_time`

Header: `void put_time(int hundredth_ip, int second_ip, int minute_ip, int hour_ip, int day_ip, int month_ip, int year_ip)`

hundreth_ip, second_ip, minute_ip, hour_ip, day_ip, month_ip, year_ip: (i) zu setzende Zeitangabe

Beschreibung: setzt das Zeitobjekt auf das angegebene Datum und Uhrzeit

Methode:	`TIME::actual_time`

Header: `void actual_time()`

Beschreibung: setzt sie Zeit auf die aktuelle Systemzeit

Methode:	`TIME::read_hundredth`

Header: `int read_hundredth() const`

Rückgabe: aktuelle Zeit (Hundestel)
Beschreibung: gibt den Hunderstel-Anteil der aktuellen Zeit aus

Methode:	`TIME::read_seconds`

Header: `int read_seconds() const`
Rückgabe: aktuelle Zeit (Sekunden)
Beschreibung: gibt den Sekunden-Anteil der akuellen Zeit aus

Methode:	`TIME::read_minutes`

Header: `int read_minutes() const`
Rückgabe: aktuelle Zeit (Minuten)
Beschreibung: gibt den Minuten-Anteil der aktuellen Zeit aus

Methode:	`TIME::read_hours`

Header: `int read_hours() const`
Rückgabe: aktuelle Zeit (Stunden)
Beschreibung: gibt den Stunden-Anteil der aktuellen Zeit aus

Methode:	`TIME::read_day`

Header: `int read_day() const`
Rückgabe: aktuelles Datum (Tag)
Beschreibung: gibt den Tag des aktuellen Datums aus

Methode:	`TIME::read_month`

Header: `int read_month() const`

Rückgabe: aktuelles Datum (Monat)

Beschreibung: gibt den Monat des aktuellen Datums aus

Methode:	`TIME::read_year`

Header: `int read_year() const`

Rückgabe: aktuelles Datum (Jahr)

Beschreibung: gibt das Jahr des aktuellen Datums aus

Methode:	`TIME::is_negative`

Header: `bool is_negative() const`

Rückgabe: TRUE oder FALSE

Beschreibung: liefert TRUE, wenn die Zeit negativ ist, ansonsten FALSE

Methode:	`TIME::TIME2sec`

Header: `long int TIME2sec() const`

Rückgabe: Zeit in Sekunden

Beschreibung: wandelt die aktuell gespeicherte Zeit in Sekunden seit dem 1.1.1970 um

Methode:	`TIME::operator*`

Header: `friend TIME operator* (double sklar, TIME time_ip)`

double skalar, TIME time_ip: (i) ein skalarer Faktor und ein Zeit-Objekt

Rückgabe: ein Zeit-Objekt, Sklares Produkt aus skalar und time_ip

Beschreibung: führt eine skalare Multiplikation mit einem Zeit-Objekt durch

Methode:	`TIME::operator+`

Header: `friend TIME operator+ (TIME time1_ip, TIME time2_ip)`

time1_ip, time2_ip: (i) zwei zu addierende Zeiten

Rückgabe: ein Zeit-Objekt, Summe aus time1_ip und time2_ip

Beschreibung: addiert zwei Zeiten

Methode:	`TIME::operator-`

Header: `friend operator- (TIME time1_ip, TIME time2_ip)`

time1_ip, time2_ip: (i) zwei Zeiten, dei subtrahiert werden sollen

Rückgabe: Differenz aus time1_ip und time2_ip

Beschreibung: subtrahiert zwei Zeiten von einander

Methode:	`TIME::operator<`

Header: `friend bool operator< (TIME time1_ip, TIME time2_ip)`

time1_ip, time2_ip: (i) zwei zu vergleichende Zeiten

Rückgabe: TRUE, wenn time1_ip ¡ time2_ip, ansonsten FALSE

Beschreibung: überprüft, ob time1_ip echt kleiner als time2_ip ist

Methode:	`TIME::operator>`

Header: `friend bool operator> (TIME time1_ip, TIME time2_ip)`

time1_ip, time2_ip: (i) zwei zu vergleichende Zeiten

Rückgabe: TRUE, wenn time1_ip ¿ time2_ip, ansonsten FALSE

Beschreibung: überprüft, ob time1_ip echt größer als time2_ip ist

Methode:	`TIME::operator==`

Header: `friend bool operator== (TIME time1_ip, TIME time2_ip)`

time1_ip, time2_ip: (i) zwei zu vergleichende Zeiten

Rückgabe: TRUE, wenn time1_ip = time2_ip, ansonsten FALSE

Beschreibung: Gleichheitstest zweier Zeiten

Methode:	`TIME::operator>>`

Header: `friend istream& operator>> (istream& is, TIME& time)`

is, time: (o,i) schreiben eines Zeit-Objektes

Rückgabe: ein Stream, in dem die Zeit steht

Beschreibung: speichert das aktuelle Zeit-Objekt als Stream

Methode:	`TIME::operator<<`

Header: `friend ostream& operator& (ostream& os, const TIME& time)`

os, time: (o,i) liest ein Zeit-Objekt aus einem Stream

Rückgabe: ein Zeit-Objekt

Beschreibung: lesen eines Zeit-Objektes aus einem Stream

A.1.8 TIMER

Die Klasse ***TIMER*** beinhaltet die Zeitmeß- und Stopfunktionen. Sie benutzt die von der Klasse ***TIME*** zur Verfügung gestellte Funktionalität.

A.1.8.1 Interface

Methode:	`TIMER::TIMER`

Header: `TIMER (const TIME alarm)`

alarm: (i) Alamzeit, mit der der Timer initialisiert wird

Beschreibung: Konstruktor der Klasse

Methode:	`TIMER::TIMER`

Header: `TIMER()`

Beschreibung: Konstruktor

Methode:	`TIMER:: TIMER`

Header: `TIMER()`

Beschreibung: Destruktor

Methode:	`TIMER::set_timer`

Header: `void set_timer(const TIME alarm)`

alarm: (i) Alarmzeit, bis der Timer ausgelöst wird

Beschreibung: setzt den Timer auf die angegebene Zeit

Methode:	`TIMER::still_running`

Header: `TIME still_running()`

Rückgabe: positive Werte: Zeit bis zum Ablauf, negative Werte: überschrittener Timer

Beschreibung: gibt die noch verbleibende Zeit bis zum Auslösen eines Alarms an, bzw. die Zeit, die der Timer bereits abgelaufen ist

A.1.9 CALC

A.1.12 Die Klasse ***CALC*** enthält alle Funktion zur Berechnung aller Koordinaten einer im Raum befindlichen Person.

A.1.9.1 Beschreibung

Zunächst wird in einer zweidimensionalen Berechnung versucht, den Kopf und Oberkörper der Person zu finden. Die Raumkoordinaten werden im Anschluß daran ermittelt.

A.1.9.2 Interface

Methode:	`CALC::CALC`

Header: `CALC`

Beschreibung: Konstruktor für ***CALC***. Holt Informationen ueber die aufgestellten Kameras ein und berechnet fuer die Berechnung der Raumkoordinaten benottigten Konstanten.

Methode:	`CALC::set_camera_data`

Header: `void set_camera_data(void)`

Beschreibung: Ermittelt die Kameradaten.

Methode:	`CALC::calculate_centre_head`

Header: `int calculate_centre_head (int status)`

status: (i) Status der 2D-Berechnung

Rückgabe: Status der 3D-Berechnung.

Beschreibung: Berechnet den Schwerpunkt des Kopfes (getrennt für links und rechts).

Methode:	`CALC::calculate_centre_body`

Header: `int calculate_centre_body (int status)`

status: (i) Status der 2DBerechnung

Rückgabe: Status der 3D-Berechnung.

Beschreibung: Berechnet den Schwerpunkt des Oberkörpers (getrennt für links und rechts).

Methode:	`CALC::estimate_centre_head`

Header: `void calculate_centre_head (int status)`

status: (i) Status der 2Dberechnung

Beschreibung: Berechnet den Schwerpunkt des Kopfes.

Methode:	`CALC::estimate_centre_head`

Header: `void calculate_centre_head (int status)`

status: (i) Status der 2Dberechnung

Beschreibung: Berechnet den Schwerpunkt des Kopfes.

Methode:	`CALC::find_person`

Header: `void find_person (void)`

Beschreibung: Startprozedur zum finden der Person im Raum.

Methode:	`CALC::faster_createbinarymatrix`

Header: `void faster_createbinarymatrix(BIN_MATR_T bin_matrix, unsigned char *uold_img, unsigned char *unew_img, unsigned char *vold_img, unsigned char *vnew_img, const int image_search_xup ,const int image_search_xdn, const int image_search_yup ,const int image_search_ydn)`

Rückgabe: Erzeugte Binärmatrix.

uold_img: (i) Altes U-Kanal Bild

unew_img: (i) Neues U-Kanal Bild

vold_img: (i) Altes V-Kanal Bild

vnew_img: (i) Neues V-Kanal Bild

image_search_xup: (i) Suchraum

image_search_xdn: (i) Suchraum

image_search_yup: (i) Suchraum

image_search_ydn: (i) Suchraum

Beschreibung: Berechnet aus den vorliegenden Binärbildern das Binärbild und parallel dazu die Binärmatrix. Die Größe der Matrix wird durch die Parameter *x_squares* und *y_squares* vorgegeben. Um nicht das ganze Bild betrachten zu müssen kann der Suchraum eingeschränkt werden (Parameter *image_search_xup* bis *image_search_ydn*).

Methode:	`CALC::findbody`

Header: `int findbody(BIN_MATR_T bin_matrix)`

Rückgabe: 1 bei Erfolg, 0 bei Fehler.

bin_matrix: (i) Berechnete Binärmatrix.

Beschreibung: Versucht innerhalb der Binärmatrix einen Oberkörper zu finden.

Methode:	`CALC::finder_createheadsearcharea`

Header: `void findbody(void)`

Beschreibung: Berechnet ausgehend von einem gefundenen Oberkörper einen Suchraum. Hier wird später der Oberkorrper gesucht.

Methode:	`CALC::searchheadarea`

Header: `int searchheadarea(BIN_MATR_T matrix,BOX_T element,int color, int xc,int yc,int found_in_area)`

Rückgabe: 1 bei Erfolg, 0 bei Fehler.

matrix: (i) Berechnete Binärmatrix.

element: (i) Kopfumrandung

color: (i) Suchfarbe

xc: (i) Zählvariable für den aktuellen Suchraum

yc: (i) Zählvariable für den aktuellen Suchraum

found_in_area: (a) Suchraum, in dem der Kopf gefunden wurde (oben oder unten).

Beschreibung: Versucht in dem gegebenen Suchraum einen Kopf zu finden.

Methode:	`CALC::findhead`

Header: `int findhead(BIN_MATR_T bin_matrix)`

Rückgabe: 1 bei Erfolg, 0 bei Fehler.

bin_matrix: (i) Berechnete Binärmatrix.

Beschreibung: Versucht der Binärmatrix den Kopf zu finden. Benötigt dazu die Funktion `searchheadarea`.

Methode:	`CALC::createHeadandBodyCenter`

Header: `void createHeadandBodyCenter(void)`

Beschreibung: Berechnet die 3D-Koordinaten frr den Schwerpunkt von Kopf- und Oberkörper.

Methode:	`CALC::calculate_room_coordinates`

Header: `void calculate_room_coordinates(VECTOR2D s1,VECTOR2D s2,VECTOR3D part)`

s1: (i) 2D-Koordinate der linken Kamera

s2: (i) 2D-Koordinate der rechten Kamera

Beschreibung: Berechnet aus den Schwerpunkts-Pixelpositionen der beiden Kamerabilder die Raumkoordinate des Schwerpunkts.

A.1.10 PIC_MANAGER

Die Klasse ***PIC_MANAGER*** stellt entweder die gegrabbten Bilder oder die Bilder eines vorher aufgenommen Filmes zur weiteren Verarbeitung zur Verfügung.

A.1.10.1 Interface

Methode:	`PIC_MANAGER::PIC_MANAGER`

Header: `PIC_MANAGER(ROOM* room)`

room: (i) Raummodell.

Beschreibung: Konstruktor für ***PIC_MANAGER***. Reseviert Speicher für die beiden Kamera Bilder.

Methode:	`PIC_MANAGER::determine_source`

Header: `int determine_source(int source = 1 , const char *dir = list_path_c)`

Rückgabe: Fehlerstatus.

source: (i) bestimmt die Bildquelle.

dir: (i) gibt den Pfad der gespeicherten Bilder an.

Beschreibung: Wählt je nach *source* entweder die Kamera oder einen Film als Quelle aus.

Methode:	`PIC_MANAGER::switch_cameras`

Header: `switch_cameras()`

Beschreibung: Vertauscht bei einem Film als Datenquelle das linke Bild mit dem rechten Bild.

Methode:	`PIC_MANAGER::read_display`

Header: `int read_display()`

Rückgabe: gibt an ob ein Bild angezeigt werden soll, oder nicht.

Beschreibung: gibt an ob ein Bild angezeigt werden soll, oder nicht.

Methode:	`PIC_MANAGER::write_display`

Header: `write_display (int i)`

i: (i) setzt show_pics.

Beschreibung: Es wird die Variable *show_pics* gesetzt. Diese zeigt an ob ein Bild angezeigt werden soll, oder nicht:
0 bedeutet nichts anzeigen
1 bedeutet nur U-Kanalanzeigen
2 bedeutet alles anzeigen.

Methode:	`PIC_MANAGER::put_pictures`

Header: `put_pictures(unsigned char *XBild_left , unsigned char *XBild_right , int channel)`

XBild_left: (i)

XBild_right: (i)

channel: (i)

Beschreibung: legt zwei Kamerabilder des angegebenen Kanals im Speicher ab.

Methode:	`PIC_MANAGER::get_ram_pic`

Header: `unsigned char *get_ram_pic(int side,int channel)`

Rückgabe: Adresse des Bildes.

side: (i)

channel: (i)

Beschreibung: liefert die Adresse des mit *side* und *channel* ausgewählten Bildes zurück.

Methode:	`PIC_MANAGER::get_camera_pic`

Header: `int get_camera_pic()`

Rückgabe: Fehlerstatus.

Beschreibung: liest das linke und das rechte Bild entweder von der Grabberkarte, oder von der Festplatte.

Methode:	`PIC_MANAGER::get_binary_pic`

Header: `unsigned char *get_binary_pic(int side,int channel)`

side: (i)

channel: (i)

Rückgabe: Adresse des Binärbildes.

Beschreibung: liefert die Adresse des mit *side* und *channel* ausgewählten Binärbildes zurück.

A.1.11 METEOR

Die Klasse ***METEOR*** stellt die Methoden zur Handhabung der Meteor-Grabberkarten und der Sony Kameras zur Verfügung. So ist es möglich den Grabvorgang zu starten und die Kameras an die Position zu bewegen , die im Kameramodell gespeichert ist.

A.1.11.1 Interface

Methode:	`METEOR::METEOR`

Header: `METEOR(ROOM* room)`

room: (i) Raummodell.

Beschreibung: Konstruktor für ***METEOR***. Liest aus dem verwendeten Kameramodell die Auflösung, stellt die zu verwendende Kanäle ein und initialisiert die Kameras, sowie die Grabberkarten.

Methode:	`METEOR::operator =`

Header: `METEOR& operator=(METEOR &met)`

Beschreibung: Die Kameradaten werden kopiert.

Methode:	`METEOR::meteorYUVgrab_setzekamera`

Header: `int meteorYUVgrab_setzekamera (const int &kamera)`

kamera: (i) zu setzende Kamera.

Rückgabe: Fehlerstatus.

Beschreibung: Wählt eine Kamera aus.

Methode:	`METEOR::meteorYUVgrab_GrabbeBilder`

Header: `meteorYUVgrab_GrabbeBilder (unsigned char *UBild , unsigned char *VBild)`

UBild: (i) Adresse des U-Kanals.

VBild: (i) Adresse des V-Kanals.

Beschreibung: Liest von der gerade gesetzten Kamera den U- und V-Kanal ein.

Methode:	`METEOR::meteorYUVgrab_initialized`

Header: `int meteorYUVgrab_initialized(void)`

Rückgabe: zeigt an , ob die Grabberkarten initialisiert wurden.

Beschreibung: zeigt an , ob die Grabberkarten initialisiert wurden.

Methode:	`METEOR::set_camera_position`

Header: `int set_camera_position()`

Rückgabe: Fehlerstatus.

Beschreibung: Liest die im Kameramodell gespeicherten Werte und sendet sie an die beiden Kameras.

Methode:	`METEOR::InitCom`

Header: `int InitCom(void)`

Rückgabe: Fehlerstatus.

Beschreibung: Initialisiert die serielle Schnittstelle.

Methode:	`METEOR::DeinitCom`

Header: `DeinitCom(void)`

Beschreibung: Deinitialisiert die serielle Schnittstelle.

A.1.12 PERSON_CALC

Die Klasse ***PERSON_CALC*** enthält die Hauptfunktion zur Berechnung der Personendaten.

A.1.12.1 Beschreibung

PERSON_CALC besteht im wesentlichen aus der Funktion `calculate_person`. Diese steuert alle Aufgaben der Bildverarbeitung für einen Funktionsdurchlauf.

A.1.12.2 Interface

Methode:	`PERSON_CALC::PERSON_CALC`

Header: `PERSON_CALC`

Beschreibung: Konstruktor für ***PERSON_CALC***. Initialisiert die Variablen und öffnet bei Bedarf Fenster zur Visualisierung des Kamerabildes.

Methode:	`PERSON_CALC::calculate_PERSON_CALC`

Header: `void calculate_person (void)`

Beschreibung: Liefert die gesamten Daten der Person und stellt sie den uebrigen Clustern zur Verfuegung

Methode:	`PERSON_CALC::Print_left_HeadAndBody`

Header: `int Print_left_HeadAndBody(VISUAL obj)`

Rückgabe: 1 falls erfolgreich, 0 wenn ein Fehler aufgetreten ist

obj: (i) Zeiger auf das Kameraobjekt.

Beschreibung: Zeigt zur Visualisierung die Rahmen um Kopf und Oberkörper im linken Kamerabild.

Methode:	`PERSON_CALC::Print_right_HeadAndBody`

Header: `int Print_right_HeadAndBody(VISUAL obj)`

Rückgabe: 1 falls erfolgreich, 0 wenn ein Fehler aufgetreten ist

obj: (i) Zeiger auf das Kameraobjekt.

Beschreibung: Zeigt zur Visualisierung die Rahmen um Kopf und Oberkörper im rechten Kamerabild.

Methode:	`PERSON_CALC::printbinarymatrix`

Header: `int printbinarymatrix(VISUAL obj,BIN_MATR_T matrix)`

Rückgabe: 1 falls erfolgreich, 0 wenn ein Fehler aufgetreten ist

obj: (i) Zeiger auf das Kameraobjekt.

matrix: (i) Erzeugte Binärmatrix

Beschreibung: Zeigt zur Visualisierung die Binärmatrix.

A.1.13 KALMAN2D

Die Klasse ***KALMAN2D*** implementiert eine Funktion zum Schätzen von Pixelpositionen und eine Funktion zum wiederholten Initialisieren der Filterparameter des Kalmanfilterobjekts. Die Schätzfunktion ist für eine Schätzung mit und ohne Messwert geeignet. (Im Falle , daß kein realer Messwert vorliegt muß der alte Schätzwert übergeben werden.)

A.1.13.1 Beschreibung

Die geschätzte Position wird nach jedem Schätzvorgang übergeben und gleichzeitig gekapselt. Der Standardkonstruktor initialisiert die Varianzen und Kovarianz sowie zwei weitere Kalmanparameter mit Standardausgangswerten.

A.1.13.2 Interface

Methode:	`KALMAN2D::KALMAN2D`

Header: `KALMAN2D(void)`

Beschreibung: Initialsiert die Kalmanfilterparameter (ss,sv,vv,r,a) mit Stanardwerten.

Methode:	`KALMAN2D::recursion`

Header:
```
VECTOR2D
recursion(const VECTOR2D& z,
          const double &dt)
```

Rückgabe: Neuer Schätzwert.

z: (i) Messwert oder alter Schätzwert.

dt: (i) Zeitintervall zwischen zwei Bildern.

Beschreibung: Berechnet eine neue Schätzposition aus dem übergebenen Messwert oder alten Schätzwert und frischt die Varianzen und die Kovarianz mit neuen Werten auf.

Methode:	`KALMAN2D::Init`

Header: `void Init(const VECTOR2D &s)`

s: (i) Neue initiale Messposition.

Beschreibung: Initialsiert die Filterposition mit dem ***VECTOR2D*** `s` sowie die Varianzen und Kovarianzen (ss,sv,vv) und zwei weitere Filterparameter (r,a) mit Standardinitialwerten.

A.1.14 ROOM

Die Klasse ***ROOM*** faßt das Modell des Raums, welches durch die Klasse ***MODEL*** repräsentiert wird, die zugehörige Bildverarbeitung, repräsentiert durch ***PERSON_CALC***, und die Auswertenden Klassen (***HUMAN*** und ***SURVEILLANCE***) zu einer Klasse zusammen. Sie bietet den verschiedenen Modulen die Möglichkeit Daten auszutauschen.

A.1.14.1 Interface

Methode:	`ROOM::ROOM`

Header: `ROOM()`

Beschreibung: Konstruktor der Klasse ***ROOM***. Die Variablen *name*, *model*, *image*, *human* und *surveillance* werden initialisiert.

Methode:	`ROOM::ROOM`

Header: `ROOM(String n)`

n: (i) Name des Raums.

Beschreibung: Konstruktor der Klasse ***ROOM***. Die Variablen *name*=n, *model*, *image*, *human* und *surveillance* werden initialisiert.

Methode:	`ROOM::write_name`

Header: `void write_name (const String& n)`

n: (i) Name des Raums.

Beschreibung: Der Name des Raums und des zum Raum gehörenden ***MODEL***s werden gesetzt.

Methode:	`ROOM::read_name`

Header: `String read_name () const`

Rückgabe: Name des Raums.

Beschreibung: Der Name des Raums wird gelesen.

Methode:	`ROOM::get_model`

Header: `MODEL* get_model () const`

Rückgabe: Zeiger auf das zum Raum gehörende ***MODEL***.

Beschreibung: Liefert Zeiger auf das zum Raum gehörende ***MODEL***.

Methode:	`ROOM::get_human`

Header: `HUMAN* get_human () const`

Rückgabe: Zeiger auf das zum Raum gehörende Objekt ***HUMAN***.

Beschreibung: Liefert Zeiger auf das zum Raum gehörende Objekt ***HUMAN***.

Methode:	`ROOM::get_image`

Header: `PERSON_CALC* get_image () const`

Rückgabe: Zeiger auf das zum Raum gehörende Objekt ***PERSON_CALC***.

Beschreibung: Liefert Zeiger auf das zum Raum gehörende Objekt ***PERSON_CALC***.

Methode:	`ROOM::get_surveillance`

Header: `SURVEILLANCE* get_surveillance () const`

Rückgabe: Zeiger auf das zum Raum gehörende Objekt ***SURVEILLANCE***.

Beschreibung: Liefert Zeiger auf das zum Raum gehörende Objekt ***SURVEILLANCE***.

Methode:	`ROOM::load`

Header: `bool ROOM::load()`

Rückgabe: Bei erfolgreichem Laden 'true', 'false' sonst.

Beschreibung: Ruft die 'load'-Funktionen der zum Raum gehörenden Klassen auf.

Methode:	`ROOM::save`

Header: `bool ROOM::save()`

Rückgabe: Bei erfolgreichem Speichern 'true', 'false' sonst.

Beschreibung: Ruft die 'save'-Funktionen der zum Raum gehörenden Klassen auf.

A.1.15 MODEL

A.1.15.1 Interface

Methode:	`MODEL::MODEL`

Header: `MODEL(ROOM* room)`

room: (i) Zeiger auf das zugehörige ***ROOM***-Objekt

Beschreibung: Erzeugt leeres ***MODEL***-Objekt

Methode:	`MODEL::get_name`

Header: `String get_name()`

Rückgabe: Name des Raums

Beschreibung: Liest den Namen des Raums aus.

Methode:	`MODEL::set_name`

Header: `void set_name(const String& n)`

n: (i) Neuer Name des Raums.

Beschreibung: Setzt den Namen des Raums

Methode:	`MODEL::get_camera_count`

Header: `int get_camera_count() const`

Rückgabe: Anzahl der Kameras

Beschreibung: Liest Anzahl der Kameras im Raummodell aus.

Methode:	`MODEL::get_camera`

Header: `CAMERA& get_camera(int n)`

n: (i) Nummer der Kamera

Rückgabe: Referenz auf n-tes ***CAMERA***-Objekt

Beschreibung: Liest ***CAMERA***-Objekt aus dem Modell aus.
Achtung: Die erste Kamera hat die Nummer 0, die letzte Kamera hat Nummer `get_camera_count` -1. *Unzulässige Kamera-Nummern führen zu undefinierten Ergebnissen.*

Methode:	`MODEL::swap_cams`

Header: `void swap_cams(void)`

Beschreibung: Vertauscht im Modell Kamera 0 und Kamera 1.

Methode:	`MODEL::get_movement`

Header:
```
double get_movement(const VECTOR2D& left_top,
                    const VECTOR2D& right_bottom,
                    const CAMERA& cam)
```

left_top: (i) Linke, obere Ecke eines Bildausschnitts.

right_bottom: (i) Rechte, untere Ecke des Bildausschnitts.

cam: (i) Kamera-Nummer.

Rückgabe: Flächenanteil potentieller Störungen im Bildausschnitt (Fenster, Fernseher etc.). 0 bedeutet keine Störungen, 1 bedeutet viele Störungen.

Beschreibung: Ermittelt für einen Bildausschnitt im Kamerabild den Flächenanteil von möglichen Störungen (Fernseher, Fenster, Spiegel etc.)

Methode:	`MODEL::get_actions`

Header: `INT_FUZZY_SET get_actions(const VECTOR2D& pos)`

pos: (i) Position im Grundriß

Rückgabe: Fuzzy-Menge der erlaubten Aktionen an dieser Raumposition

Beschreibung: Liest die Fuzzy-Menge der erlaubten Aktionen für eine gegebene Raumposition.

Methode:	`MODEL::put_actions`

Header:
```
void put_actions(const VECTOR2D& pos,
                 const INT_FUZZY_SET& actions)
```

pos: (i) Position im Grundriß

actions: (i) Fuzzy-Menge der erlaubten Aktionen

Beschreibung: Setzt die Fuzzy-Menge der erlaubten Aktionen für eine gegebene Raumposition.

Methode:	`MODEL::get_height`

Header: `double get_height (const VECTOR2D& pos)`

pos: (i) Position im Grundriß

Rückgabe: Höhe an Position *pos*.

Beschreibung: Liest die Höhe eines Gegenstandes an Position *pos* (Stichwort: $2\frac{1}{2}$-D Modell).

Methode:	`MODEL::put_height`

Header: `void put_height(const VECTOR2D& pos, double h)`

pos: (i) Position im Grundriß

h: (i) Höhe

Beschreibung: Setzt die Höhe an Position *pos*.

Methode:	`MODEL::get_covering`

Header: `double get_covering(const VECTOR2D& pos)`

pos: (i) Position im Grundriß

Rückgabe: Sicherheit (aus [0..1]) für eine Verdeckung der Person

Beschreibung: Liest die Sicherheit, mit der die Person an einer gegebenen Position im Grundriß verdeckt ist (d.h. nicht sichtbar für die Kameras).

Methode:	`MODEL::get_covering`

Header: `double get_covering(const VECTOR2D&, int camnum)`

pos: (i) Position im Grundriß

camnum: (i) Kamera-Nummer

Rückgabe: Sicherheit (aus [0..1]) für eine Verdeckung der Person

Beschreibung: Liest die Sicherheit, mit der die Person an einer gegebenen Position für Kamera Nummer *camnum* verdeckt ist.

Methode:	`MODEL::get_exit`

Header:
```
double get_exit(const VECTOR2D& pos,
                const VECTOR2D& direction,
                String& exit_to)
```

pos: (i) Position im Grundriß

direction: (i) Richtungsvektor

exit_to: (o) Raumname

Rückgabe: Sicherheit, mit der von der angegebenen Position aus in der angegebenen Richtung ein Ausgang liegt ($\in [0..1]$).

Beschreibung: Überprüft, ob der Raum von Position *pos* aus in Richtung *direction* verlassen werden kann. Falls der Möglichkeitswert größer als 0 ist, wird in *exit_to* der nächstliegende Raum in dieser Richtung ausgegeben. Ansonsten ist der Wert von *exit_to* undefiniert.

Methode:	`MODEL::get_exit`

Header:
```
double get_exit(const VECTOR2D& pos,
                String& exit_to)
```

pos: (i) Position im Grundriß

exit_to: (o) Raumname

Rückgabe: Gibt an, wie nahe die gegebene Position an einem Ausgang liegt (Sicherheit aus [0..1]).

Beschreibung: Überprüft, ob der Raum von Position *pos* aus verlassen werden kann. Falls der Möglichkeitswert größer als 0 ist, wird in *exit_to* der nächstliegende Raum ausgegeben. Ansonsten ist der Wert von *exit_to* undefiniert.

Methode:	`MODEL::load`

Header:
```
int load()
```

Rückgabe: 0 falls Fehler, 1 falls O.K.

Beschreibung: Model laden. Der Filename ist *name* + ".model".

Methode:	`MODEL::load`

Header: `int load (const char *file)`

file: (i) Filename

Rückgabe: 0 falls Fehler, 1 falls O.K.

Beschreibung: Model-File "file" laden.

Methode:	`MODEL::save`

Header: `int save(const char *file)`

file: (i) Filename

Rückgabe: 0 falls Fehler, 1 falls O.K.

Beschreibung: Modell unter dem Namen "file" abspeichern.

Methode:	`MODEL::save`

Header: `int save()`

Rückgabe: 0 falls Fehler, 1 falls O.K.

Beschreibung: Modell unter dem mit `set_name` gesetzten Namen abspeichern.

Methode:	`MODEL::read_name`

Header: `const String& read_name() const`

Rückgabe: Raumname

Beschreibung: Liest Raumnamen.

Methode:	`MODEL::get_square`

Header: `SQUARE* get_square(int x, int y) const`

x: (i) x-Koordinate im ***SQUARE***-Array

y: (i) x-Koordinate im ***SQUARE***-Array

Rückgabe: Zeiger auf ***SQUARE*** an position (x, y).
`NULL` falls die Koordinaten nicht zulässig waren.

Beschreibung: Liest das ***SQUARE***-Objekt an den gegebenen Koordinaten.

Methode:	`MODEL::get_square_coordinates`

Header: `VECTOR2D`
`get_square_coordinates(const VECTOR2D& pos)`

pos: (i) Koordinaten im Grundriß

Rückgabe: ***SQUARE***-Koordinaten im Raum-Modell

Beschreibung: Umrechnung von Raumkoordinaten in ***SQUARE***-Koordinaten.

Methode:	`MODEL::get_square_coordinate`

Header: `int`
`get_square_coordinate(double coordinate) const`

coordinate: (i) Koordinate im Grundriß

Rückgabe ***SQUARE***-Koordinate

Beschreibung: Umrechnung einer Raumkoordinate in eine ***SQUARE***-Koordinate.

Methode:	`MODEL::read_x_length`

Header: `double read_x_length() const`

Rückgabe Raumgröße in x-Richtung

Beschreibung: Liest die Raumgröße in x-Richtung.

Methode:	`MODEL::read_y_length`

Header: `double read_y_length() const`

Rückgabe Raumgröße in y-Richtung

Beschreibung: Liest die Raumgröße in y-Richtung.

Methode:	`MODEL::read_square_length`

Header: `double read_square_length() const`

Rückgabe Seitenlänge eines ***SQUARE***s

Beschreibung: Liest die Seitenlänge eines ***SQUARE***s.

Methode:	`MODEL::get_floor_pos`

Header: `double get_floor_pos(int x)`

x: (i) ***SQUARE***-Koordinate

Rückgabe Koordinate im Grundriß

Beschreibung: Umrechnung einer ***SQUARE***-Koordinate in eine Grundrißkoordinate.

Methode:	`MODEL::get_floor_pos`

Header: `VECTOR2D get_floor_pos(int x, int y)`

x: (i) ***SQUARE*** x-Koordinate

y: (i) ***SQUARE*** y-Koordinate

Rückgabe Koordinaten im Grundriß

Beschreibung: Umrechnung von classSQUARE-Koordinaten in Grundrißkoordinaten.

Methode:	`MODEL::init`

Header: `void init ()`

Beschreibung: Initialisierungsfunktion. Wird nach dem Laden eines Modells aufgerufen.

Methode:	`MODEL::operator<<`

Header: `friend`
`ostream& operator<<(ostream&, const MODEL&)`

Beschreibung: Gibt ein ***MODEL***-Objekt in einen ***ostream*** aus.

A.1.16 CAMERA

Die Klasse ***CAMERA*** kapselt die Kameradaten und stellt Zugriffsfunktionen zur Verfügung. Das Schreiben der Kameradaten hat aber keine direkte Auswirkung auf die Hardware. Hier werden nur die reinen Daten verändert. Falls man die Blickrichtung (*direction*) setzt, muß man im entsprechenden ***MODEL*** die Funktion `init` aufrufen, damit die von der Blickrichtung abhängigen Daten, wie Sichtbarkeit, neu berechnet werden.

A.1.16.1 Interface

Methode:	`CAMERA::CAMERA`

Header: `CAMERA()`

Beschreibung: Konstruktor für ***CAMERA***. Initialisiert die privaten Variablen
position= $(0, 0, 1)$,
direction= $(1, 1, -0.5)$,
angle_width= *angle_height*= M_PI_2,
resolution_x= 384 und
resolution_y= 288.

Methode:	`CAMERA::CAMERA`

Header: `CAMERA (const CAMERA& b)`

Rückgabe: Kopie von b.

b: (i) CAMERA-Objekt.

Beschreibung: Wie man sieht ist es der "copy-constructor".

Methode:	`CAMERA::operator=`

Header: `CAMERA& operator=(const CAMERA& b)`

Beschreibung: Zuweisung. Das CAMERA-Objekt wird kopiert.

Methode:	`CAMERA::CAMERA`

Header:
```
CAMERA CAMERA (const VECTOR3D& pos,
               const VECTOR3D& di,
               double a_w,
               double a_h,
               int x_res,
               int y_res)
```

Rückgabe: Mit den übergebenen Werten initialisiertes CAMERA-Objekt.

pos: (i) Position der Kamera im Raum.

di: (i) Blickrichtung der Kamera.

a_w: (i) 2∗ kleinster Winkel (im Bogenmaß) zwischen dem Vektor zum rechten Bildrand und dem Richtungsvektor.

a_h: (i) 2∗ kleinster Winkel (im Bogenmaß) zwischen dem Vektor zum unteren Bildrand und dem Richtungsvektor.

x_res: (i) Auflösung der Kamera in x-Richtung.

y_res: (i) Auflösung der Kamera in y-Richtung.

Beschreibung: Initialisiert ein mit den übergebenen Werten initialisiertes CAMERA-Objekt.

Methode:	`CAMERA::write_res_x`

Header: `void write_res_x (int x_res)`

x_res: (i) Auflösung der Kamera in x-Richtung.

Beschreibung: Schreibt die Auflösung der Kamera in x-Richtung.

Methode:	`CAMERA::write_res_y`

Header: `void write_res_y (int y_res)`

y_res: (i) Auflösung der Kamera in y-Richtung.

Beschreibung: Schreibt die Auflösung der Kamera in y-Richtung.

Methode:	`CAMERA::read_res_x`

Header: `int read_res_x (void)`

Rückgabe: Auflösung der Kamera in x-Richtung.

Beschreibung: Liest die Auflösung der Kamera in x-Richtung.

Methode:	`CAMERA::read_res_y`

Header: `int read_res_y (void)`

Rückgabe: Auflösung der Kamera in y-Richtung.

Beschreibung: Liest die Auflösung der Kamera in y-Richtung.

Methode:	`CAMERA::operator<<`

Header:
```
friend
ostream& operator<<(ostream&,
                    const CAMERA&)
```

Beschreibung: Gibt ein CAMERA-Objekt in einen `ostream` aus.

Methode:	`CAMERA::operator>>`

Header:
```
friend
istream& operator>>(istream&,
                    CAMERA&)
```

Beschreibung: Liest CAMERA-Objekt aus einem `istream`.
Syntax-Beispiel:
$angle\_width = 0.352961$
$angle\_height = 0.272386$
$resolution\_x = 384$
$resolution\_y = 288$
$direction = (-0.79, 7.2, -1.3)$
$position = (2.49, 0.3, 2.3)$
$config\_data$ =194 -223 1 0 0 1 14

A.1.17 OBJECT

Die Klasse ***OBJECT*** dient den Klassen ***RECTANGLE***, ***TRIANGLE*** und ***CIRCLE*** als Basisklasse.

A.1.17.1 Interface

Methode:	`OBJECT::OBJECT`

Header: `OBJECT(void)`

Beschreibung: Konstruktor der Basisklasse ***OBJECT***. Initialisiert die Höhe h auf 0m.

Methode:	`OBJECT::registrate`

Header: `virtual void registrate(MODEL&) = 0`

Beschreibung: Trägt ein Objekt in den Grundriß ein. (Muß in der Abgeleiteten Klasse implementiert sein.)

Methode:	`OBJECT::read_height`

Header: `double read_height(void) const`

Rückgabe: Liefert die Höhe des Objekts.

Methode:	`OBJECT::write_height`

Header: `int write_height(double height)`

height: (i) Höhe des Objekts.

Beschreibung: Setzt die Höhe des Objekts.

Methode:	`OBJECT::put_actions`

Header: `void put_actions(const INT_FUZZY_SET& a)`

a: (i) Plausibilitäten für bestimmte Aktionen innerhalb der Grundrißfläche, in die sich das Objekt einträgt.

Beschreibung: Schreibt Plausibilitäten für bestimmte Aktionen innerhalb der Grundrißfläche, in die sich das Objekt einträgt.

Methode:	`OBJECT::put_actions`

Header: `void put_action(int action, double possibility)`

action: (i) Aktionsnummer

possibility: (i) Plausibilität.

Rückgabe: '0' im Fehlerfall, '1' sonst.

Beschreibung: Setzt die Plausibilität einer, durch die Aktionsnummer bestimmten, Aktion.

Methode:	`OBJECT::get_actions`

Header: `INT_FUZZY_SET get_actions(void) const`

Rückgabe: Ein ***INT_FUZZY_SET***, das die Plausibilitäten für bestimmte Aktionen innerhalb des Objekts repräsentiert.

Beschreibung: Liefert ein ***INT_FUZZY_SET***, das die Plausibilitäten für bestimmte Aktionen innerhalb der Grundrißfläche repräsentiert, die durch das Objekt beschrieben wird.

Methode:	`OBJECT::print_to_stream`

Header:
```
virtual
ostream& print_to_stream(ostream&) const = 0
```

Beschreibung: Schreibt das Objekt in einen *ostream*. Diese Funktion wird vom überladenen Ausgabeoperator aufgerufen.

A.1.18 RECTANGLE

Die Klasse ***RECTANGLE*** erbt von der Klasse ***OBJECT*** die Fähigkeit, sich in den Grundriß eintragen zu können.

A.1.18.1 Interface

Methode:	`RECTANGLE::RECTANGLE`

Header: `RECTANGLE(void)`

Beschreibung: Konstruktor erzeugt leeres Rechteck.

Methode:	`RECTANGLE::read_lower_left`

Header: `VECTOR2D read_lower_left(void) const`

Rückgabe: Untere linke Ecke des Rechtecks.

Beschreibung: Liefert untere linke Ecke des Rechtecks.

Methode:	`RECTANGLE::read_upper_right`

Header: `VECTOR2D read_upper_right(void) const`

Rückgabe: Obere rechte Ecke des Rechtecks.

Beschreibung: Liefert als Ergebnis die obere rechte Ecke des Rechtecks (Wer hätte das gedacht?)

Methode:	`RECTANGLE::write_position`

Header:
```
int write_position(const VECTOR2D& lower_left,
                   const VECTOR2D& upper_right)
```

lower_left: (i) unten links

upper_right: (i) oben rechts

Rückgabe: '0' im Fehlerfall, '1' sonst.

Beschreibung: Schreibt die Eckdaten des Rechtecks.
Voraussetzung: lower_left.x ,-.y<= upper_right.x,-.y.

Zum Interface der virtuellen Funktionen siehe bei Klasse ***OBJECT***.

A.1.19 TRIANGLE

Die Klasse ***TRIANGLE*** erbt von der Klasse ***OBJECT*** die Fähigkeit, sich in den Grundriß eintragen zu können. Die Höhe des Dreiecks ist, da es von ***OBJECT*** erbt, natürlich konstant. Die Dreiecke werden definiert durch lower_left und upper_right ihrer Bounding-Box und durch die enum-Variable *form*, die die Ausrichtung des Dreiecks angibt (Position des Kathetenschnittpunkts innerhalb der Bounding-Box).

A.1.19.1 Interface

Die Klasse ***TRIANGLE*** hat die gleichen Funktionen wie die Klasse ***RECTANGLE***. Lediglich die Funktion `write_position` hat sich leicht geändert. Hinzugekommen ist nur die Funktion `read_form`.

Methode:	`TRIANGLE::read_form`

Header: `triangle_form read_form(void) const`

Rückgabe: Die Form des Dreiecks.

Beschreibung: Liefert Form des Dreiecks.

Methode:	`TRIANGLE::write_position`

Header:
```
int write_position(const VECTOR2D& lower_left,
                   const VECTOR2D& upper_right,
                   const triangle_form& form)
```

lower_left: (i) Untere linke Ecke der Bounding-Box.

upper_right: (i) Obere rechte Ecke der Bounding-Box.

form: (i) Die 'Form' des Dreiecks.

Beschreibung: Schreibt die Daten, die das Dreieck beschreiben.

Mehr hat sich im Vergleich zu ***RECTANGLE*** nicht geändert.

A.1.20 CIRCLE

Der Kreis (bzw. die Ellipse) wird durch ll(lower_left) und ur (upper_right) einer Bounding-Box definiert. Die Registrierung erfolgt zeilenweise, wobei in jeder Zeile die Ränder durch Einsetzen der Zeile in eine Ellipsengleichung bestimmt werden. Alle anderen Funktionen sind analog zu ***RECTANGLE***.

A.1.21 DOOR

Die Klasse ***DOOR*** wird von der Klasse ***MODEL*** benutzt, um die im Raum vorhandenen Türen abzuspeichern. Hierzu werden die Grundrißdaten der Ecken der Tür abgespeichert. Außerdem wird der Name des Raums abgespeichert, der durch die Tür erreicht werden kann.

Methode:	`DOOR::DOOR`

Header: `DOOR()`

Beschreibung: Konstruktor der Klasse ***DOOR***.
Initialisiert *to_room*='target-room'.

Methode:	`DOOR::DOOR`

Header: `DOOR(const String& target_room)`

target_room: (i) Name des Raums, in den man bei Benutzung der Tür gelangt.

Beschreibung: Konstruktor der Klasse ***DOOR***.
Initialisiert *to_room=target-room*

Methode:	`DOOR::read_edge_1`

Header: `VECTOR2D read_edge_1() const`

Rückgabe: Erster Eckpunkt in Grundrißkoordinaten.

Beschreibung: Liest ersten Eckpunkt in Grundrißkoordinaten.

Methode:	`DOOR::read_edge_2`

Header: `VECTOR2D read_edge_2() const`

Rückgabe: Zweiter Eckpunkt in Grundrißkoordinaten.

Beschreibung: Liest zweiten Eckpunkt in Grundrißkoordinaten.

Methode:	`DOOR::write_edge_1`

Header: `void write_edge1(const VECTOR2D& v)`

v: (i) Erster Eckpunkt in Grundrißkoordinaten.

Beschreibung: Schreibt ersten Eckpunkt in Grundrißkoordinaten.

Methode:	`DOOR::write_edge_2`

Header: `void write_edge2(const VECTOR2D& v)`

v: (i) Zweiter Eckpunkt in Grundrißkoordinaten.

Beschreibung: Schreibt zweiten Eckpunkt in Grundrißkoordinaten.

Methode:	`DOOR::read_target_room`

Header: `String read_target_room(void) const`

Rückgabe: Name des Zielraums.

Beschreibung: Liefert den Namen des Zielraums.

Methode:	`DOOR::write_target_room`

Header: `void write_target_room(const String& room)`

room: (i) Raumname

Beschreibung: Setzt den Namen des Raums, in den man durch Benutzung der Tür gelangt.

Methode:	`DOOR::near_exit`

Header: `double near_exit(const VECTOR2D& u) const`

u: (i) Position im Grundriß.

Rückgabe: Abstandsgrad zur Tür $[0, \ldots, 1]$.

Beschreibung: Liefert Abstandsgrad zur Tür.
1='sehr nah an der Tür'
0='nicht im geringsten nah der Tür'

Methode:	`DOOR::to_exit`

Header: `double to_exit(const VECTOR2D& u, VECTOR2D v) const`

u: (i) Position im Grundriß.

v: (i) Richtungsvektor.

Rückgabe: Der Grad des Schnittes des Richtungsvektors mit der Tür.

Beschreibung: Je näher der Schnitt der Geraden, die einerseits durch u mit Richtung v und andererseits durch den Eckpunkt der Tür *e1* mit Richtung d=(e1-e2) verlaufen, wobei d die Verbindung zwischen den Eckpunkten *e1* und *e2* der Tür repräsentiert, am Mittelpunkt (m=*e2*+$\frac{1}{2}d$) liegt, desto näher ist der Grad bei 1.

A.1.22 WINDOW

Die Klasse ***WINDOW*** wird von der Klasse ***MODEL*** benutzt, um die im Raum vorhandenen Fenster abzuspeichern. Hierzu werden die Raumkoordinaten der Fensterecken abgespeichert. Die Klasse ***WINDOW*** kann auch dazu benutzt werden, um allgemein störende Flächen im Raum zu modellieren. Zusätzlich zu den vier Ecken wird nämlich der Grad der Bewegung abgespeichert, der aussagen soll, wie viel Bewegung von diesem Fenster zu erwarten sein soll (0 =keine Bewegung, $\ldots$, 1 =viel Bewegung). Dieser Wert kann mit

`write_movement` gesetzt werden. Die Funktion `get_single_movement` gibt, abhängig vom übergebenen ***CAMERA***-Objekt und dem Bildausschnitt, den Grad der zu erwartenden störenden Bewegung im ausgewählten Bildausschnitt an.

A.1.22.1 Interface

Methode:	`WINDOW::WINDOW`

Header:
```
WINDOW(const VECTOR3D& p0,
       const VECTOR3D& p1,
       const VECTOR3D& p2,
       const VECTOR3D& p3,
       const double m)
```

$p0, \ldots, p3$**:** (i) Die vier Eckpunkte des ***WINDOW***-Objekts

m: (i) Grad der zu erwartenden Störung.

Beschreibung: Konstruktor der Klasse ***WINDOW***.

Methode:	`WINDOW::write_edge`

Header:
```
void write_edge(const int number,
                const VECTOR3D& edge)
```

number: (i) Eckennummer.

edge: (i) Raumkoordinaten des Eckpunktes.

Beschreibung: Schreibt die Ecke mit der entsprechenden Eckennummer $[0, \ldots, 3]$.

Methode:	`WINDOW::write_movement`

Header:
```
void write_movement(const double m)
```

m: (i) Störgrad.

Beschreibung: Setzt den Grad der zu erwartenden Störung auf m.

Methode:	`WINDOW::get_single_movement`

Header:
```
double
get_single_movement ( const VECTOR2D& left_top,
                      const VECTOR2D& right_bottom,
                      const CAMERA& cam)
```

left_top: (i) Linke obere Ecke des Bildausschnittes.

right_bottom: (i) Rechte untere Ecke des Bildausschnittes.

cam: (i) Kameraobjekt mit Kameradaten.

Rückgabe: Zu erwartender Störungsanteil relativ zur Fläche des mittels *left_top* und *right_bottom* beschriebenen Bildausschnittes.

A.1.23 SQUARE

Die Klasse ***MODEL*** legt ein zweidimensionales Array von ***SQUARE***s an. Jedes ***SQUARE*** repräsentiert eine Grundrißflächeneinheit. Zur Zeit beträgt die Fläche pro ***SQUARE*** $\frac{1}{4}m^2$. In einem ***SQUARE*** wird ein ***INT_FUZZY_SET*** gespeichert, welches die Plausibilitäten für bestimmte Bewegungen enthalten soll. Falls ein ***OBJECT*** auf diesem ***SQUARE*** steht und sich dort mittels der Funktion `registrate` eingetragen hat, werden dessen Höhe und die Aktionsplausibilitäten dort gespeichert.

A.1.23.1 Interface

Methode:	`SQUARE::SQUARE`

Header: `SQUARE()`

Beschreibung: Konstruktor initialisiert *height*=*near_door*=0. Ein ***SQUARE*** ist sichtbar, kein versteckter Ausgang und auch nicht in der Nähe einer Tür wenn es erzeugt wird.

Methode:	`SQUARE::SQUARE`

Header: `SQUARE ( const INT_FUZZY_SET& as, double h, double vis[MAXCAMCOUNT], double n_d, bool h_e)`

as: (i) Plausibilitäten für Aktionen.

h: (i) Die Höhe des Objekts, das auf dem ***SQUARE*** steht.

vis: (i) Die Sichtbarkeiten abhängig von der Kameranummer.

n_d: (i) Grad der Nähe zu einer Tür.

h_e: (i) Boolescher Wert: 'true' falls das ***SQUARE*** einen verdeckten Ausgang repräsentiert, 'false' sonst.

Beschreibung: Konstruktor der Klasse ***SQUARE***.

Methode:	`SQUARE::write_actions`

Header: `void write_actions (const INT_FUZZY_SET& acts)`

acts: (i) Diskrete Fuzzy-Menge, die die Plausibilitäten für bestimmte Aktionen innerhalb dieses ***SQUARE***s enthält.

Beschreibung: Kann zur Änderung der Plausibilitäten für bestimmte Aktionen genutzt werden.

Methode:	`SQUARE::read_actions`

Header: `const INT_FUZZY_SET& read_actions () const`

Rückgabe: Diskrete Fuzzy-Menge, die die Plausibilitäten für bestimmte Aktionen innerhalb dieses ***SQUARE***s enthält.

Beschreibung: Liest Plausibilitäten für bestimmte Aktionen innerhalb dieses ***SQUARE***s.

Methode:	`SQUARE::write_height`

Header: `void write_height (double h)`

h: (i) Die Höhe des Objekts, das auf dem ***SQUARE*** steht.

Beschreibung: Schreibt die Höhe des Objekts, das auf dem ***SQUARE*** steht.

Methode:	`SQUARE::read_height`

Header: `double read_height () const`

Rückgabe: Die Höhe des Objekts, das auf dem ***SQUARE*** steht.

Beschreibung: Liest die Höhe des Objekts, das auf dem ***SQUARE*** steht.

Methode:	`SQUARE::write_visibility`

Header: `void write_visibility (double v, int camnum)`

v: (i) Grad der Sichtbarkeit.

camnum: (i) Kameranummer.

Beschreibung: Schreibt den Grad der Sichtbarkeit für das ***SQUARE*** in Abhängigkeit von der Kameranummer.

Methode:	`SQUARE::read_visibility`

Header: `double read_visibility (int camnum) const`

camnum: (i) Kameranummer.

Rückgabe: Liefert den Grad der Sichtbarkeit für das ***SQUARE*** in Abhängigkeit von der Kameranummer.

Methode:	`SQUARE::write_near_door`

Header: `void write_near_door (double door)`

door: (i) Der Grad der Nähe zu einer Tür.

Beschreibung: Schreibt den Grad der Nähe zu einer Tür.

Methode:	`SQUARE::read_near_door`

Header: `double read_near_door () const`

Rückgabe: Der Grad der Nähe zu einer Tür.

Beschreibung: Liefert den Grad der Nähe zu einer Tür.

Methode:	`SQUARE::write_hidden_exit`

Header: `void write_hidden_exit (bool h_e)`

h_e: (i) Boolescher Wert 'true', falls das ***SQUARE*** ein 'versteckter Ausgang' ist.

Beschreibung: Wird während der Initialisierung eines ***MODEL***s aufgerufen. Parameter *h_e*='true', falls es eine versteckte Verbindung vom ***SQUARE*** zu einem Ausgang gibt.

Methode:	`SQUARE::read_hidden_exit`

Header: `bool read_hidden_exit () const`

Rückgabe: Boolescher Wert 'true', falls das ***SQUARE*** ein 'versteckter Ausgang' ist.

Beschreibung: Liefert 'true', falls das ***SQUARE*** ein 'versteckter Ausgang' ist, 'false' sonst.

A.1.24 SURVEILLANCE

Die Klasse ***SURVEILLANCE*** fragt beim Modul ***HUMAN*** die erkannte Bewegung des Typs ***HUMAN_ACTION*** ab, und "berechnet" daraus, sowie aus dem vorherigen Zustand einen neuen Alarmzustand und liefert diesen in Form eines ***ALARM*** (siehe A.1.25) an das Toplevel-Modul.

A.1.24.1 Interface

Methode:	`SURVEILLANCE::SURVEILLANCE`

Header: `SURVEILLANCE(void *r);`

Beschreibung: Konstruktor der Klasse, erhält einen Zeiger auf das ***ROOM***-Objekt übergeben. Die lokalen Variablen werden initialisiert.

Methode:	`SURVEILLANCE::load`

Header: `bool load()`

Rückgabe: TRUE, wenn laden erfolgreich, sonst FALSE

Beschreibung: Liefert immer TRUE (***SURVEILLANCE*** hat nichts zu laden)

Methode:	`SURVEILLANCE::save`

Header: `bool save()`

Rückgabe: TRUE, wenn sichern erfolgreich, sonst FALSE

Beschreibung: Liefert immer TRUE (s.o.)

Methode:	`SURVEILLANCE::alarm_accept`

Header: `void alarm_accept()`

Beschreibung: Setzt den letzten Alarm zurück, ohne ihn aus der lokalen History zu löschen.

Methode:	`SURVEILLANCE::alarm_cancel`

Header: `void alarm_cancel()`

Beschreibung: Setzt den letzten Alarm zurück und löscht ihn aus der lokalen History.

Methode:	`SURVEILLANCE::alarm_discard`

Header: `bool alarm_discard(TIME alarm_time)`

alarm_time: (i) Zeitpunkt, zu dem der zu löschende Alarm stattfand.

Rückgabe: TRUE, wenn erfolgreich, sonst FALSE

Beschreibung: Löscht den zum genannten Zeitpunkt in der History stehenden ALARM, sofern dieser existiert.

Methode:	`SURVEILLANCE::request_alarm_status()`

Header: `ALARM request_alarm_status()`

Rückgabe: Aktueller Alarmstatus

Beschreibung: Eigentliche Funktionalität des Moduls: Hier wird vom Modul ***HUMAN*** die aktuell erkannte Bewegung erfragt und diese in Bezug auf Ort, evtl. laufende Timeouts und die zuletzt erkannte Bewegung hin bewertet und daraus der neue Alarmstatus generiert.

Methode:	`SURVEILLANCE:: SURVEILLANCE`

Header: `SURVEILLANCE()`

Beschreibung: Destruktor

A.1.25 ALARM

Die Klase ***ALARM*** stellt einen Übergabetyp zwischen ***SURVEILLANCE*** und der Benutzerschnittstelle dar. Im Prinzip handelt es sich dabei um einen Record mit den Componenten *alarm_type*, *position*, *action* und *exitname*.

Der Alarmtyp kann dabei folgende Werte annehmen:

1. none: kein Alarm
2. Exit: Ausgang zur Umwelt
3. ExitNext: Ausgang zum nächsten überwachten Raum
4. Precall: Voralarm, noch kein konkreter Alarm, aber irgendwas stimmt nicht
5. Alarm: sofortiger Alarm
6. Lost: Die Person kann nicht mehr erkannt werden, aber es scheint auch (noch) keinen Grund für einen Alarm zu geben.
7. Hidden: Die Person kann nicht mehr erkannt werden, sie hat sich hinter einen großen (und dem System bekannten) Gegenstand bewegt.

Die Position wird in 2D-Koordinaten auf den Raumboden projeziert weitergereicht, die erkannte Aktion wird im Klartext durchgereicht, und der Name des Ausgangs (soweit vorhanden) wird ebenfalls im Klartext weitergereicht.

A.1.25.1 Interface

Methode:	`ALARM::ALARM`

Header: `ALARM()`

Beschreibung: Default Konstruktor. Kein Alarm an Position (0,0), keine Bewegung, kein Ausgang

Methode:	`ALARM::ALARM`

Header: `ALARM(const VERCTOR2d& pos)`

pos: (i) Position der Person

Beschreibung: Konstruktor, Kein Alarm an Position *pos*, keine Bewegung, kein Ausgang

Methode:	`ALARM::ALARM`

Header: `ALARM(const ALARM_E& alt)`

alt: (i) Alarmtyp

Beschreibung: Konstruktor, Alarm des Typs *alt* an Position (0,0),...

Methode:	`ALARM::ALARM`

Header: `ALARM(onst ALARM_E& alt, const VECTOR2D& pos)`

alt: (i) Alarmtyp

pos: (i) Position

Beschreibung: Konstruktor, Alarm des Typs *alt* an der Position *pos*,...

Methode:	`ALARM:: ALARM`

Header: `ALARM()`

Beschreibung: Destruktor

Methode:	`ALARM::write_alarm_type`

Header: `void write_alarm_type(const ALARM_E& alarmtype`

alarmtype: (i) Alarmtyp
Beschreibung: Setzt den Alarmtyp auf *alarmtype*

Methode:	`ALARM::read_alarm_type`

Header: `ALARM_E read_alarm_type() const`
Rückgabe: Typ des Alarms
Beschreibung: Liest den Alarmtyp

Methode:	`ALARM::write_pos`

Header: `void write_pos(const VECTOR2d& position_2d`
position_2d: (i) Position der Person
Beschreibung: Setzt die Position der Person aus *position_2d*

Methode:	`ALARM::read_pos`

Header: `VECTOR2D read_pos() const`
Rückgabe: Position der Person
Beschreibung: Liest die Position der Person

Methode:	`ALARM::write_exitname`

Header: `void write_exitname(const String& exit)`
exit: (i) Name des Ausgangs
Beschreibung: Setzt den Namen des Ausgangs bzw. des Nachbarraumes

Methode:	`ALARM::read_exitname`

Header: `String read_exitname() const`

Rückgabe: Name des Ausgangs

Beschreibung: Liest den Namen des Ausgangs bzw. des Nachbarraumes

Methode:	`ALARM::write_action`

Header: `void write_action(const String& ac`

ac: (i) Textbezeichnung der erkannten Aktion

Beschreibung: Setzt die erkannte Aktion auf *ac*

Methode:	`ALARM::read_action`

Header: `String read_action() const`

Rückgabe: Name der Aktion

Beschreibung: Liest den Namen der erkannten Aktion

A.1.26 HUMAN

A.1.26

Die Klasse ***HUMAN*** berechnet aus den vom Modul ***IMAGE*** erhaltenen Daten Sicherheiten für verschiedene Bewegungen und liefert diese zur (eventuellen) Generierung eines Alarms an die Klasse ***SURVEILLANCE***

A.1.26.1 Interface

Methode:	`HUMAN::HUMAN`

Header: `HUMAN(void *r);`

Beschreibung: Konstruktor der Klasse, erhält einen Zeiger auf das ***ROOM***-Objekt.

Methode:	`HUMAN::load`

Header: `bool load()`

Rückgabe: TRUE oder FALSE

Beschreibung: Liefert, da ***HUMAN*** nichts laden muß, immer TRUE

Methode:	`HUMAN::save`

Header: `bool save()`

Rückgabe: TRUE oder FALSE

Beschreibung: Genau wie load immer TRUE

Methode:	`HUMAN::get_human_action`

Header: `HUMAN_ACTION get_human_action()`

Rückgabe: Die analysierten Werte, gekapselt in ***HUMAN_ACTION***

Beschreibung: Die Schnittstelle zu ***SURVEILLANCE***. Durch den Aufruf wird die Analyse gestartet.

Methode:	`HUMAN::get_actual_motion_data`

Header: `PERSON get_actual_motion_data ()`

Rückgabe: Die aktuellen Bilddaten

Beschreibung: Diese Methode liest die aktuellen Daten von ***IMAGE***.

Methode:	`HUMAN::update_action_history`

Header: `void update_action_history (INT_FUZZY_SET new_actions)`

new_actions: (i) zu speichernde Bewegungssicherheiten

Beschreibung: Aktualisiert die History

Methode:	`HUMAN::create_final_results`

Header: `HUMAN_ACTION create_final_results ()`

Rückgabe: Die fertigen analysierten Daten

Beschreibung: Diese Methode erzeugt die endgültigen Sicherheiten durch Verarbeitung der in der History gespeicherten Daten.

Methode:	`HUMAN::set_no_recognition_value`

Header: `void set_no_recognition_value(int reliabillity ()`

reliabillity: (i) Die Genauigkeit der Bilddaten

Beschreibung: Berechnet eine Sicherheit fü die Zuverlässigkeit der Bilddaten.

Methode:	`HUMAN::set_lying_ground_value`

Header: `void set_lying_ground_value (double speed, double alignment, double hight)`

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

alignment: (i) Der Höhenunterschied Kopf - Körperschwerpunkt

hight: (i) Die Höhe des Körperschwerpunktes

Beschreibung: Berechnet die Sicherheit für die Aktion Liegen am Boden

Methode:	`HUMAN::set_lying__value`

Header: `void set_lying_value (double speed, double alignment, double hight)`

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

alignment: (i) Der Höhenunterschied Kopf - Körperschwerpunkt

hight: (i) Die Höhe des Körperschwerpunktes

Beschreibung: Berechnet die Sicherheit für die Aktion Liegen (erhöht)

Methode:	`HUMAN::set_sitting_ground_value`

Header: `void set_sitting_ground_value (double speed, double alignment, double hight)`

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

alignment: (i) Der Höhenunterschied Kopf - Körperschwerpunkt

hight: (i) Die Höhe des Körperschwerpunktes

Beschreibung: Berechnet die Sicherheit für die Aktion Sitzen am Boden

Methode:	`HUMAN::set_sitting_value`

Header: `void set_sitting_value (double speed, double alignment, double hight)`

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

alignment: (i) Der Höhenunterschied Kopf - Körperschwerpunkt

hight: (i) Die Höhe des Körperschwerpunktes

Beschreibung: Berechnet die Sicherheit für die Aktion Sitzen (erhöht)

Methode:	`HUMAN::set_rolling_value`

Header: `void set_rolling_value (double xy_direction, double speed, double alignment, double hight)`

xy_direction: (D)ie Bewegungsrichtung auf der xy-Achse

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

alignment: (i) Der Höhenunterschied Kopf - Körperschwerpunkt

hight: (i) Die Höhe des Körperschwerpunktes

Beschreibung: Berechnet die Sicherheit für die Aktion Rollen (im Rollstuhl)

Methode:	`HUMAN::set_standing_value`

Header: `void set_standing_value (double speed, double alignment, double hight)`

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

alignment: (i) Der Höhenunterschied Kopf - Körperschwerpunkt

hight: (i) Die Höhe des Körperschwerpunktes

Beschreibung: Berechnet die Sicherheit für die Aktion Stehen

Methode:	`HUMAN::set_walking_value`

Header: `void setwalking_value (double xy_direction, double speed, double alignment, double hight)`

xy_direction: (D)ie Bewegungsrichtung auf der xy-Achse

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

alignment: (i) Der Höhenunterschied Kopf - Körperschwerpunkt

hight: (i) Die Höhe des Körperschwerpunktes

Beschreibung: Berechnet die Sicherheit für die Aktion Gehen)

Methode:	`HUMAN::set_downwards_value`

Header: `void set_downwards_value (double z_direction, double speed`

z_direction: (i) Die Bewegungsrichtung auf der xy-Achse

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

Beschreibung: Berechnet die Sicherheit für die Aktion (langsame) Abwärtsbewegung

Methode:	`HUMAN::set_upwards_value`

Header: `void set_upwards_value (double z_direction, double speed`

z_direction: (i) Die Bewegungsrichtung auf der xy-Achse

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

Beschreibung: Berechnet die Sicherheit für die Aktion (langsame) Aufwärtsbewegung

Methode:	`HUMAN::set_falling_value`

Header: `void set_falling_value (double z_direction, double speed`

z_direction: (i) Die Bewegungsrichtung auf der xy-Achse

speed: (i) Die Geschwindigkeit der menschlichen Bewegung

Beschreibung: Berechnet die Sicherheit für die Aktion Sturz (!)

Methode:	`HUMAN::analyse_human_action`

Header: `INT_FUZZY_SET analyse_human_action ()`

Rückgabe: Die Sicherheiten für alle Bewegungen (frr das aktuelle Bild)

Beschreibung: Hauptanalysemethode des Objekts. Berechnet die Sicherheiten für die einzelnen Bewegungen.

Methode:	`HUMAN::check_vertical_alignment`

Header: `double check_vertical_alignment (VECTOR3D head_position, VECTOR3D body_position)`

head_position: (i) Die Lage des Kopfschwerpunktes

body_position: (i) Die Lage des Körperschwerpunktes

Rückgabe: Der Höhenunterschied als DOUBLE

Beschreibung: Berechnet den Höhenunterschied zwischen Kopf- und Körperschwerpunkt

Methode:	`HUMAN::check_speed`

Header: `double check_speed (VECTOR3D body_speed)`

body_speed: (i) Die Geschwindigkeit des Menschen

Rückgabe: Die Geschwindigkeit als DOUBLE

Beschreibung: Berechnet aus dem vektoriellen Geschwindigkeitswert einen absoluten Geschwindigkeitswert

Methode:	`HUMAN::check_mass_centre_higt`

Header: `double check_mass_centre_hight (VECTOR3D body_position)`

body_position: (i) Die Lage des Körperschwerpunktes

Rückgabe: Die Höhe als DOUBLE

Beschreibung: Berechnet die Höhe des Körperschwerpunkt

Methode:	`HUMAN::check_direction`

Header: `POSITIONS check_direction (VECTOR3D body_direction)`

body_direction: (i) Die Richtung der Bewegung

Rückgabe: Die Richtung aufgeteilt in xy- und z-Achsen Anteil

Beschreibung: Berechnet den Bewegungsanteil in Richtung der xy- und z-Achse

A.1.27 HUMAN_ACTION

Dieses Objekt kapselt die Daten, die von ***HUMAN*** an ***SURVEILLANCE*** übergeben werden.

A.1.27.1 Interface

Methode:	`HUMAN::get_actions`

Header: `INT_FUZZY_SET get_actions () const`
Rückgabe: Die Sicherheiten für die einzelnen Aktionen
Beschreibung: Extrahiert die Sicherheiten für die Bewegungen

Methode:	`HUMAN::put_actions`

Header: `void put_actions (INT_FUZZY_SET actions)`
actions: (i) Die zu speichernden Sicherheiten
Beschreibung: Speichert die übergebenden Sicherheiten

Methode:	`HUMAN::get_position`

Header: `VECTOR2D get_position () const`
Rückgabe: Die Position des Menschen
Beschreibung: Extrahiert die Position des Menschen

Methode:	`HUMAN::put_position`

Header: `void put_position (VECTOR2D position)`
position: (i) Die zu speichernde Position
Beschreibung: Speichert die übergebende Position des Menschen

Methode:	`HUMAN::get_direction`

Header: `VECTOR2D get_direction () const`

Rückgabe: Die Bewegungsrichtung des Menschen

Beschreibung: Extrahiert die Bewegungsrichtung des Menschen

Methode:	`HUMAN::put_direction`

Header: `void put_direction (VECTOR2D direction)`

direction: (i) Die zu speichernde Bewegungsrichtung

Beschreibung: Speichert die übergebenden Bewegungsrichtung des Menschen

Anhang B

Theoretische Grundlagen

B.1 Transformation von Bildkoordinaten in Raumkoordinaten

Bei der Ermittlung der Schwerpunktspositionen des Kopfs und Oberkörpers ergeben sich als primärer Output Pixelpositionen im rechten und linken Kamerabild , welche zwecks der Analyse der realen Koordinaten (3D-Koordinaten) in entsprechende Raumkoordinaten umgerechnet werden müssen. Notwendige Einflussgrössen zur Transformation der 2D- in 3D-Koordinaten sind die Parameter der zwei verwendeten Kameras :

- Raumpositionen der Objektive
- Blickausrichtungen (Als Richtungsvektoren)
- Horizontale und vertikale Erfassungswinkel

B.1.1 Definition des räumlichen Koordinatensystems

Die X- und Y-Koordinate definieren zusammen eine Position in der zweidimensinalen Draufsicht (siehe Abbildung B.1). Und Zwar verläuft die X-Achse längs der Verbindungslinie der Kameraorte. Die Y-Koordinate liegt in der Draufsicht senkrecht zur X-Koordinate und definiert damit von den Kameras ausgehend die räumliche Tiefe. Die Höhe im Raum ist durch die Z-Koordinate festgelegt (siehe Abbildung B.2).

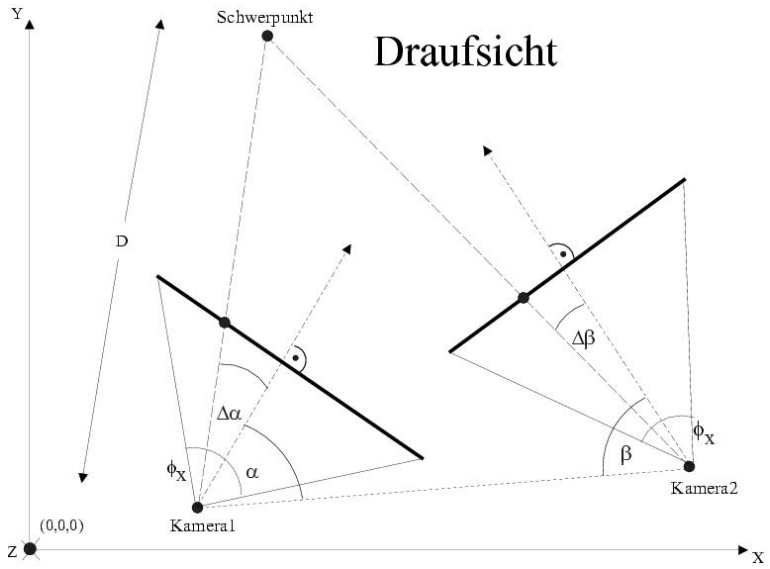

Abbildung B.1: Darstellung der Draufsicht

B.1.2 Verwendete Transformationsmethode

Die Ausrichtungsvektoren der Kameraobjektive und die Ortspositionen der Kameras im definierten XY-Draufsichtskoordinatensystem (siehe Abbildung B.1) sind die Grundlage für die Bildung zweier Geradengleichungen (B.8 / B.9) . Dazu werden die beiden Ausrichtungsvektoren gemäß der Lage der horizontalen Pixelposition in dem jeweilig betrachteten Grabberbild derart rotiert (B.6 / B.7) , daß sie in die Richtung des Schwerpunkts zeigen (Rotation um $\Delta\ \alpha$ für das linke Kamerabild und Rotation um $\Delta\ \beta$ für das rechte Kamerabild) . Zwecks Errechnung von $\Delta\ \alpha$ (B.4) und $\Delta\ \beta$ (B.5) werden vorerst zwei sogenannte *Pixeldistanzen* (B.1 / B.2) bestimmt(siehe auch Abbildung B.3). Außerdem wird zur späteren Berechnung des vertikalen Sichtausrichtungswinkels (B.13) eine Pixeldistanz B.3 für die linke Kamera errechnet. Mittels der Pixeldistanz B.1 bzw. B.2 und des horizontalen Abstandes des Schwerpunkts von der Grabberbildmitte werden schließlich die gesuchten Winkel $\Delta\ \alpha$ und $\Delta\ \beta$ berechnet.
Die zu den beiden Kameras gehörigen Geradengleichungen haben als Aufhängepunkte die Orte der Kameraobjektive und als Richtungsvektoren die jeweils durch Rotation der Ausrichtungsvektoren entstandenen Richtungsvektoren.

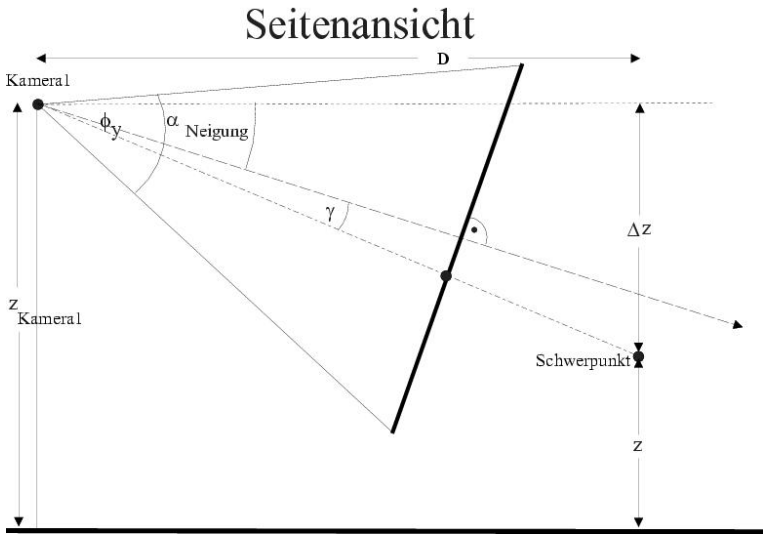

Abbildung B.2: Darstellung der Seitenansicht

Als nächster Schritt erfolgt die Bestimmung des Schnittpunkts dieser Geraden über die Berechnung der beiden Richtungsvektorskalierungsparameter (B.10) . Aus der Berechnung des Schnittpunkts ergibt sich die X- und Y-Koordinate des Schwerpunktes (B.11) . Zwecks Berechnung der Draufsichtsdistanz (B.12 , siehe auch Abbildung B.1) des Schwerpunkts vom linken Kameraobjektiv wird der Betrag des mit dem entsprechenden Geradenparameter skalierten Richtungsvektors der linken Kamera bestimmt.
Diese Distanz wird verwendet um die dritte Komponente der Raumkoordinate des Schwerpunkts zu berechnen (siehe Abbildung B.2) . Dazu wird vorerst der auf dem Ausrichtungsvektor der linken Kamera basierenden vertikale Neigungswinkel ($\alpha_{Neigung}$) des Objektivs und der Winkel zwischen der Sichtausrichtung und der Richtung des Schwerpunktes (B.13) mittels der vertikalen Pixelposition im Grabberbild und der Pixeldistanz (B.3) ermittelt. Die vorher berechnete Distanz wird nun über eine trigonometrische Funktion in Abhängigkeit der Summe des Neigungs- und Sichtwinkels (B.13) in den vertikalen Abstand des Objektivs vom Schwerpunkt umgerechnet. Letztendlich ergibt sich die eigentliche Z-Koordinate des Schwerpunkts (B.14) durch Differenzbildung der Objektivhöhe und des soeben berechneten vertikalen Abstands des Objektivs vom Schwerpunkt.

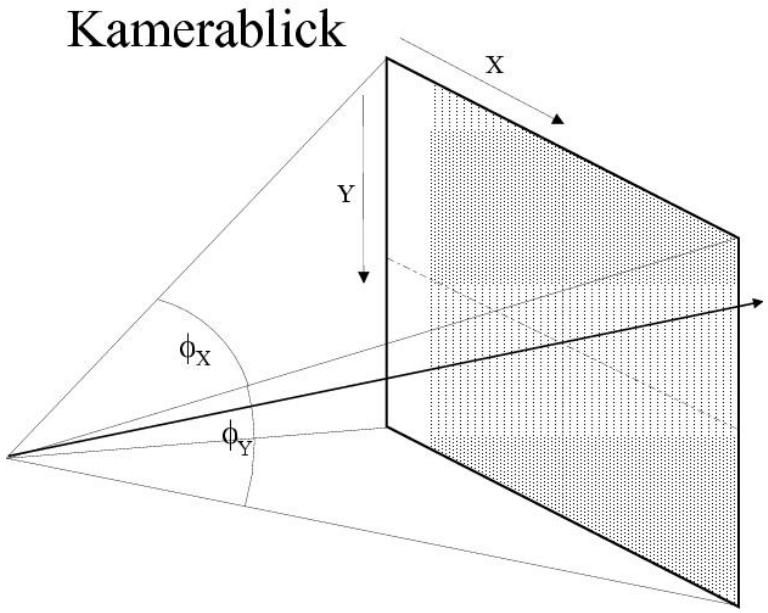

Abbildung B.3: Kamerablick

Berechnung der Schwerpunktsrichtungsvektoren

$$pixeldist_{x_{links}} = \frac{\frac{Bildbreite(Pixel)}{2}}{tan(\frac{\phi_{x_{links}}}{2})} \tag{B.1}$$

$$pixeldist_{x_{rechts}} = \frac{\frac{Bildbreite(Pixel)}{2}}{tan(\frac{\phi_{x_{rechts}}}{2})} \tag{B.2}$$

$$pixeldist_{y_{links}} = \frac{\frac{Bildhöhe(Pixel)}{2}}{tan(\frac{\phi_{y_{links}}}{2})} \tag{B.3}$$

$$\Delta\alpha = arctan\left(\frac{\frac{Bildbreite(Pixel)}{2} - x(Pixel)_{Schwerpunkt_{links}}}{pixeldist_{x_{links}}}\right) \tag{B.4}$$

$$\Delta\beta = arctan\left(\frac{\frac{Bildbreite(Pixel)}{2} - x(Pixel)_{Schwerpunkt_{rechts}}}{pixeldist_{x_{rechts}}}\right) \tag{B.5}$$

$$\begin{pmatrix} x_{r_{links}} \\ y_{r_{links}} \end{pmatrix} = \begin{pmatrix} cos(\Delta\alpha) & -sin(\Delta\alpha) \\ sin(\Delta\alpha) & cos(\Delta\alpha) \end{pmatrix} \begin{pmatrix} x_{Ausrichtung_{v_{links}}} \\ y_{Ausrichtung_{v_{links}}} \end{pmatrix} \tag{B.6}$$

$$\begin{pmatrix} x_{r_{rechts}} \\ y_{r_{rechts}} \end{pmatrix} = \begin{pmatrix} cos(\Delta\beta) & -sin(\Delta\beta) \\ sin(\Delta\beta) & cos(\Delta\beta) \end{pmatrix} \begin{pmatrix} x_{Ausrichtung_{v_{rechts}}} \\ y_{Ausrichtung_{v_{rechts}}} \end{pmatrix} \tag{B.7}$$

Definierte Geradengleichungen

$$G_{linkeKamera} : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_{Objektiv_{links}} \\ y_{Objektiv_{links}} \end{pmatrix} + \lambda \begin{pmatrix} x_{r_{links}} \\ y_{r_{links}} \end{pmatrix} \tag{B.8}$$

$$G_{rechteKamera} : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_{Objektiv_{rechts}} \\ y_{Objektiv_{rechts}} \end{pmatrix} + \mu \begin{pmatrix} x_{r_{rechts}} \\ y_{r_{rechts}} \end{pmatrix} \tag{B.9}$$

Berechnung des Geradenschnittes

$$\begin{pmatrix} \lambda \\ \mu \end{pmatrix} = \begin{pmatrix} x_{r_{links}} & -x_{r_{rechts}} \\ y_{r_{links}} & -y_{r_{rechts}} \end{pmatrix}^{-1} \begin{pmatrix} x_{Objektiv_{rechts}} - x_{Objektiv_{links}} \\ y_{Objektiv_{rechts}} - y_{Objektiv_{links}} \end{pmatrix} \tag{B.10}$$

$$\begin{pmatrix} x_{Schwerpunkt} \\ y_{Schwerpunkt} \end{pmatrix} = \begin{pmatrix} x_{Objektiv_{links}} \\ y_{Objektiv_{links}} \end{pmatrix} + \lambda_{Schnitt} \begin{pmatrix} x_{r_{links}} \\ y_{r_{links}} \end{pmatrix} \tag{B.11}$$

Berechnung der Schwerpunkthöhe

$$D_{Kameraobjektiv_{links} \rightarrow Schwerpunkt} = \left| \lambda \begin{pmatrix} x_{r_{links}} \\ y_{r_{links}} \end{pmatrix} \right| \tag{B.12}$$

$$\gamma = arctan \left(\frac{y_{Schwerpunkt_{links}} - \frac{Bildhöhe(Pixel)}{2}}{pixeldist_y} \right) \tag{B.13}$$

$$z_{Schwerpunkt} = z_{Objektiv_{links}} - D_{Kameraobjektiv_{links} \rightarrow Schwerpunkt} tan\left(\gamma + \alpha_{Neigung}\right) \tag{B.14}$$

B.2 Grundlagen der Bildverarbeitung

B.2.1 Farbmodelle

Farbmodelle [RCG93] werden dem Anwender häufig durch die verwendete Hardware vorgegeben. ***Ahelp*** verwendet das YUV-Farbmodell.

B.2.1.1 Das YUV-Farbmodell

Das YUV-Farbmodell wird in der Video- und Fernsehtechnik verwendet.Es spiegelt die in einem Videosignal kodierten Helligkeits- und Farbinformationen wieder. Entwickelt wurde das Modell aus der Notwendigkeit heraus, nach der Entwicklung des Farbfernsehns die reine Helligkeitsdarstellung um Möglichkeiten der Farbwiedergabe zu erweitern. Y entspricht der Helligkeit und UV der Farbinformation. Die Farbinformation ist vektoriell kodiert. U entspricht der Blau-Gelb-Achse und V der Rot-Grün-Achse. Beide Achsen sind um 90 Grad zueinander gedreht (siehe Abb. B.4.

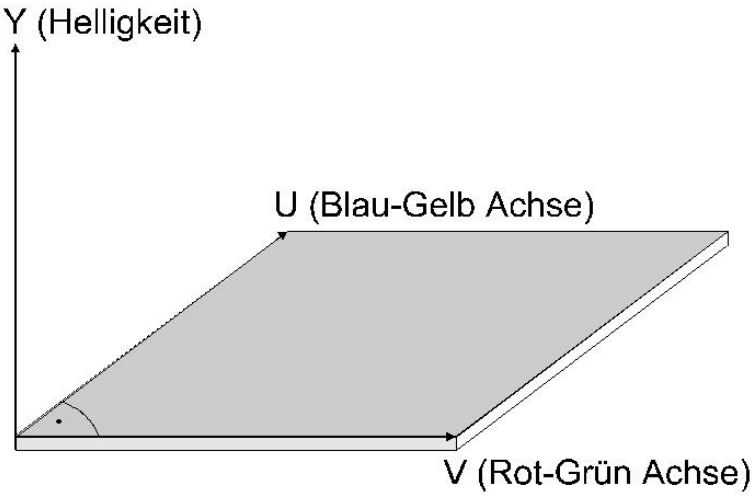

Abbildung B.4: Das YUV-Farbmodell

B.2.1.2 Andere Farbmodelle

Das wohl bekannteste Farbmodell ist das additive RGB-Modell. Hier werden die Bildpunkte durch Farbmischung aus den drei Grundfarben Rot, Grün und Blau zusammengesetzt. Dieses Modell ist jedoch nicht besonders intuitiv. So ist nicht auf Anhieb klar, aus welchen Anteilen der Grundfarben etwa ein Braunton zusammengesetzt ist. Auch eine Trennung der Helligkeitsinformation von der Farbinformation ist mit dem RGB-Modell nur schwer zu realisieren. Daher entstanden für unterschiedliche Anwendungsfälle neue Farbmodelle, wie etwa das CMYK- oder das HSI-Farbmodell. Hier steht H für Hue (Farbton), S für Saturation (Sättigung) und I für Intensity (Intensität). Der Farbton ist die dominante Lichtfarbe. Dieser wird als Winkel angegeben, der in der Praxis auf das diskrete Zahlenintervall abgebildet wird. Die Sättigung ist ein Maß für die Verdünnung mit Weiß, die Farbe wird heller. Die Intensität bestimmt die Helligkeit. Ein Verringern der Intensität entspricht dem Hinzumischen von Schwarz, die Farbe wird dunkler.

B.2.2 Methoden der Bildverbesserung

Bei der Digitalisierung von Bildern entstehen häufig Fehler, welche die weitere Bearbeitung erschweren können. Ziel einer Bildverbesserung ist es, diese Fehler soweit wie möglich zu reduzieren.Gleichzeitig können grundlegende Aspekte der Bildvorlage (Erhöhung der Helligkeit oder der Bildschärfe) an bestimmte Aufgaben angepaßt werden. Grundlage für Methoden der Bildverbesserung ist häufig eine symetrische Matrix, die über jeden Bildpunkt gelagert wird. Ausgehend von den Matrixelementen und der verknüpften Operation lassen sich dann verschiedene Filteroperationen definieren. Beispiele hierfür sind etwa der Sobel-Operator zur Kantenextraktion oder der Laplace-Operator zum entfernen gestörter Bildpunkte. Bezeichnet wird dieses Verfahren, als Manipulation im Ortsbereich, weil ein Bildpunkt auf Basis seiner direkten Nachbarn verändert wird.
Zur Geschwindigkeitsoptimierung kann der Versuch unternommen werden, zunächst den Bildausschnitt und die Matrix zu transformieren. Die eigentliche Manipulation enstpricht dann einer schlichten Multiplikation beider Komponenten. Das Resultat muß dann noch zurück transformiert werden. Eine der bekanntesten Möglichkeiten, diese Transformation durchzuführen ist die Fouriertransformation. Man bezeichnet diese Umwandlung als Trans-

formation in den Frequenzbereich des Bildpunktes und analog die Manipulation selbst als Manipulation im Ortsfrequenzbereich.
Ein wesentlicher Nachteil aller Bildverbesserungsverfahren ist der hohe Rechenzeitbedarf, der besonders in Echtzeitsystemen die bessere Bildqualität schnell relativiert. Aus diesem Grund kommen in ***Ahelp*** keine Bildverbesserungsverfahren zum Einsatz.

B.2.3 Bewegungserkennung

Ziel einer Bewegungserkennung ist es, allgemeine Bewegung in einem Bild festzustellen und diese Bewegung bestimmten Klassen zuzuordnen. ***Ahelp*** verwendet das Differenzbildverfahren zur Bewegungserkennung.

B.2.3.1 Differenzbildverfahren

Das Differenzbildverfahren ist ein relativ einfaches Verfahren zur Erkennung von Bewegung. Hierbei wird das alte Bild pixelweise mit dem neuen Bild verglichen. übersteigt die Differenz einen vorher festgestellte Schwellenwert, so wird Bewegung festgestellt. Dieses Verfahren hat die Einschränkung, daß sich nur das zu untersuchende Objekt auch wirklich bewegen darf. Der Raum selbst dagegen darf keinen wesentlichen Veränderungen unterliegen. Dadurch bedingt ist auch der Einsatz beweglicher Kameras mit diesem Verfahren nahezu ausgeschlossen. Denn dann müßte die erkannte Bewegung, die einen Wechsel einer Kameraposition entsteht, zunächst herausgefiltert werden, was wiederum Rechenzeit kostet.
Desweiteren sollte sich nur eine Person im Raum aufhalten. Sonst wäre es schwierig, erkannte Bewegung eindeutig einer Person zuzuordnen.
Die größte Einschränkung für dieses Verfahren ist jedoch dessen Komplexität. Jeder Bildpunkt muß mit seinem Vorgänger verglichen werden. Gerade bei großen Bildvorlagen ergiebt sich somit ein nicht zu unterschätzender Bedarf an Rechenzeit. Bei Echtzeitsystemen ist diese Rechenzeit pro Arbeitsgang jedoch sehr knapp bemessen. Glücklicherweise läßt sich dieses Problem durch den Einsatz geeigneter Maßnahmen reduzieren. Es bieten sich etwa eine Schätzung zukünftiger Bewegungen von Objekten an. Hierbei wird, ausgehend von einer vorhandenen Bewegung, die neue Position des betreffenden Objektes geschätzt. Der Bereich, in dem das Objekt gesucht werden muß,

läßt sich damit erheblich einschränken. ***Ahelp*** verwendet den Kalmanfilter zur Bewegungsschätzung.

B.2.3.2 Weitere Verfahren zur Bewegungserkennung

Auf dem Gebiet der Mustererkennung bewegter Objekte wird seit Jahrzehnten von vielen Firmen und Institutionen rege Forschung betrieben. Dadurch haben sich unterschiedliche Ansätze für verschiedene Anwendungsgebiete entwickelt.
So kann etwa versucht werden, für das zu untersuchende Objekt zunächst eine dreidimensionale Struktur zu finden. Ausgehend von der erkannten Struktur und der Kenntnis typischer Bewegungsabläufe, die für dieses Objekt zulässig sind, kann dann eine Bewegung klassifiziert werden.
Ein anderer Ansatz besteht darin, die Bewegung selbst zu analysieren. Dazu wird der Verlauf einzelner Bildpunkte oder bestimmter Bildregionen analysiert. Die erfalßte Bewegung innerhalb einer Bildsequenz wird mit vorher definierten Musterverläufen verglichen. Dem Modell mit der geringsten Abweichung wird die Bewegung dann zugeordnet.
Auf Basis der Bewegungsanalyse und einer zusätzlichen Betrachtung der Oberfläche wurde an der Unviversität von Rochester ein System zur Bewegungserkennung entwickelt. Zunächst werden in dem Bewegungsbild zusammenhängende Regionen gesucht. Diese Flächen werden vorher klassifizierten Strukturen zugeordnet. Ein davon unabhängiger Bewegungsfinder versucht dann bewegte Objekte zu ermitteln. Die Kombination beider Verfahren soll das System dann in die Lage versetzen, periodisch bewegte Objekte zu erkennen, und deren Bewegung zu klassifizieren (z.B. Laufen bei einer Person). Abbildung B.5 zeigt ein Beispiel zur praktischen Anwendung.

Für weitere Informationen siehe [NP92], [PN92], [NP94a] und [NP94b].

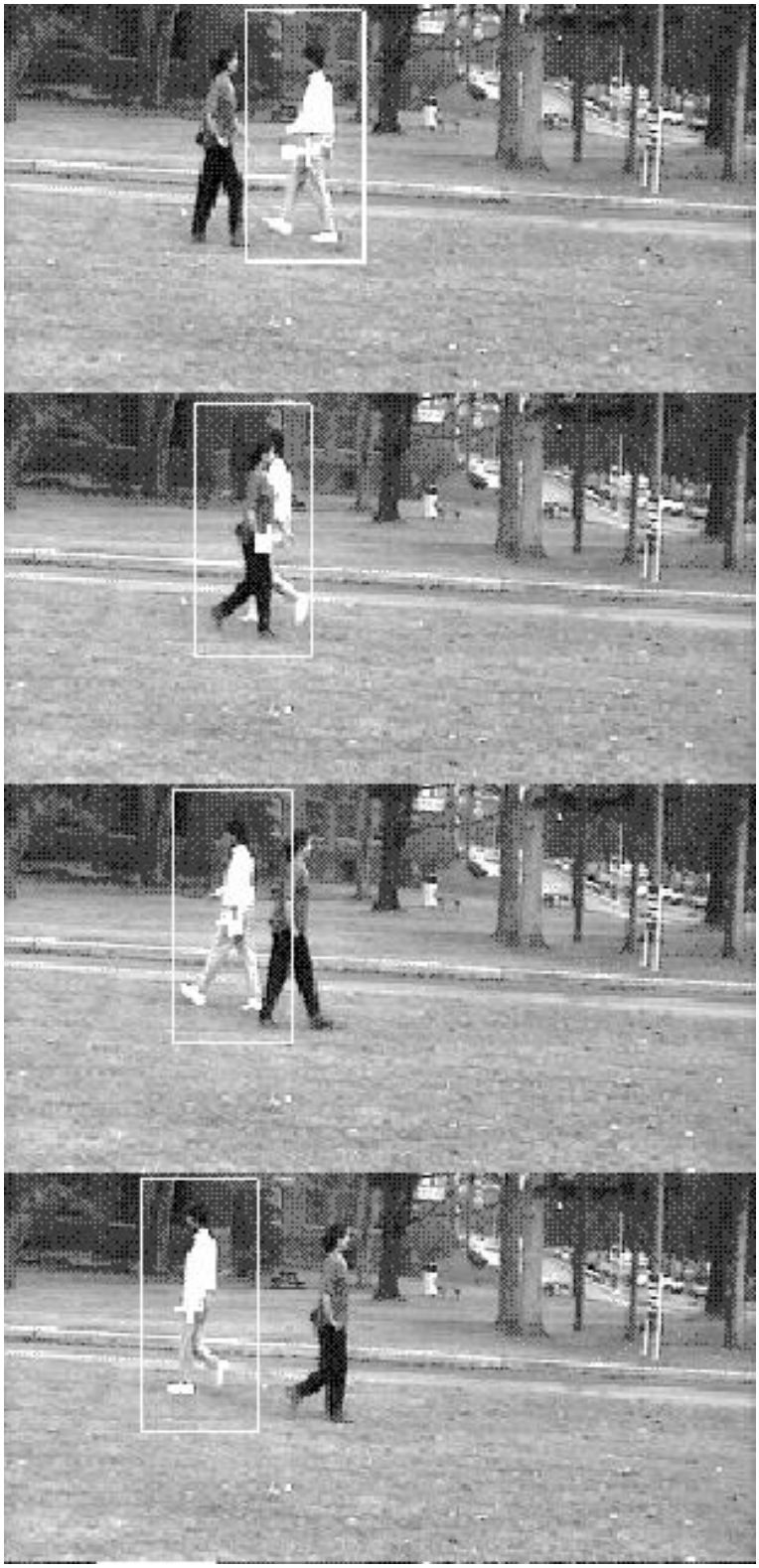

Abbildung B.5: Anwendungsbeispiel zur Bewegungserkennung

Index

ALARM 146
SURVEILLANCE 144
TIME 99
TIMER 104

actual_time 99
Aktion 59
ALARM
    ALARM 146
    ALARM 145, 146
    read_action 148
    read_alarm_type 147
    read_exitname 147
    read_pos 147
    write_action 148
    write_alarm_type 146
    write_exitname 147
    write_pos 147
ALARM 145, 146
alarm_accept 143
alarm_cancel 143
alarm_discard 144
Alarmzustände **59**
analyse_human_action 153
Argus 5, 6

Binärmatrix 46
Blau 47

CALC
    CALC 105
    calculate_centre_body . . 106
    calculate_centre_head . . 105
    calculate_room_coordinates 109
    createHeadandBodyCenter 108
    estimate_centre_head . . . 106
    faster_createbinarymatrix 106
    find_person 106
    findbody 107
    finder_createheadsearcharea 107
    findhead 108
    searchheadarea 108
    set_camera_data 105
CALC 105
calculate_centre_body 106
calculate_centre_head 105
calculate_PERSON_CALC 114
calculate_room_coordinates 109
CAMERA
    CAMERA 127, 128
    operator<< 129
    operator= 128
    operator>> 130
    read_res_x 129
    read_res_y 129
    write_res_x 129
    write_res_y 129
CAMERA 127, 128
check_direction 154
check_mass_centre_higt 154

check_speed 154
check_vertical_alignment . . 153
clip 74, 80
CMYK 163
co 73
create_final_results 150
Create_HeadSearchArea 48
createHeadandBodyCenter . . . 108
CreateHeadAndBodyCenter 49

Define_rotation 92
DeinitCom 113
det 92, 96
determine_source 109
Differenzbildberechnung 7
Differenzbildverfahren 6, 46
DOOR
  DOOR 135
  near_exit 137
  read_edge_1 135
  read_edge_2 135
  read_target_room 136
  to_exit 137
  write_edge_1 136
  write_edge_2 136
  write_target_room 136
DOOR 135
DOUBLE_FUZZY_SET
  clip 80
  DOUBLE_FUZZY_SET 76
  get_mu 77
  hgt 77
  operator<< 80
  operator= 79
  operator>> 80
  operator% 79
  operator* 79
  operator*= 79
  read_h 79
  read_lr 78
  read_lt 78
  read_rr 78
  read_rt 78
  set_parameter 77
DOUBLE_FUZZY_SET 76

empty 72
estimate_centre_head 106

Farbmodelle 162
faster_createbinarymatrix . 106
Faster_CreateBinaryMatrix 46
find_person 106
findbody 107
FindBody 48
finder_createheadsearcharea 107
findhead 108
FindHead 49
Frequenzbereich 164

get_actions 121, 132, 155
get_actual_motion_data 149
get_binary_pic 111
get_camera 120
get_camera_count 120
get_camera_pic 111
get_covering 122
get_direction 156
get_exit 123
get_floor_pos 126
get_height 121
get_human 118
get_human_action 149
get_image 118
get_model 117
get_movement 120
get_mu 71, 77

get_name 119
get_position 155
get_ram_pic 111
get_single_movement 139
get_square 125
get_square_coordinate 125
get_square_coordinates 125
get_surveillance 118
Grün 48

Handgeste 6
Hans Röttger 7
HeadSearchArea 48
hgt 72, 77
Histogramm 7
HSA 48
HSI 163
HUMAN
  analyse_human_action ... 153
  check_direction 154
  check_mass_centre_higt . 154
  check_speed 154
  check_vertical_alignment 153
  create_final_results ... 150
  get_actions 155
  get_actual_motion_data . 149
  get_direction 156
  get_human_action 149
  get_position 155
  HUMAN 148
  load 149
  put_actions 155
  put_direction 156
  put_position 155
  save 149
  set_downwards_value 152
  set_falling_value 153
  set_lying__value 150
  set_lying_ground_value . 150
  set_no_recognition_value 150
  set_rolling_value 151
  set_sitting_ground_value 151
  set_sitting_value 151
  set_standing_value 152
  set_upwards_value 153
  set_walking_value 152
  update_action_history .. 149
HUMAN 148

init 127
Init 116
InitCom 113
INT_FUZZY_SET
  clip 74
  co 73
  empty 72
  get_mu 71
  hgt 72
  INT_FUZZY_SET 71
  operator+ 73
  operator+= 74
  operator− 73
  operator<< 75
  operator= 72
  operator>> 75
  operator% 74
  operator* 72
  operator*= 74
  put_mu 71
  read_size 72
  union_with_singleton 73
INT_FUZZY_SET 71
invert 92, 96
is_negative 101

KALMAN2D

Init 116
KALMAN2D 116
recursion 116
KALMAN2D 116
Kalmanfilter 7, 47, 165
Körperteile 8

Laplace-Operator 163
length 84, 90
load 118, 123, 124, 143, 149

M2S-Sensor 7
MATRIX2D
Define_rotation 92
det 92
invert 92
MATRIX2D 91
operator∗ 94
operator+ 93
operator− 93, 94
read_c 92
scale 91
write_c 93
MATRIX2D 91
MATRIX3D
det 96
invert 96
MATRIX3D 95
operator∗ 98
operator+ 97
operator− 97
read_c 96
scale 96
write_c 97
MATRIX3D 95
METEOR
DeinitCom 113
InitCom 113
METEOR 112
meteorYUVgrab_GrabbeBilder 112
meteorYUVgrab_initialized 113
meteorYUVgrab_setzekamera 112
operator = ... 112
set_camera_position 113
METEOR 112
meteorYUVgrab_GrabbeBilder 112
meteorYUVgrab_initialized . 113
meteorYUVgrab_setzekamera . 112
MODEL
get_actions 121
get_camera 120
get_camera_count 120
get_covering 122
get_exit 123
get_floor_pos 126
get_height 121
get_movement 120
get_name 119
get_square 125
get_square_coordinate . . 125
get_square_coordinates . 125
init 127
load 123, 124
MODEL 119
operator<< 127
put_actions 121
put_height 122
read_name 124
read_square_length 126
read_x_length 125
read_y_length 126
save 124
set_name 119
swap_cams 120
MODEL 119

near_exit 137
normalize 84, 90

OBJECT
  get_actions 132
  OBJECT 130
  print_to_stream 132
  put_actions 131
  read_height 131
  registrate 130
  write_height 131
OBJECT 130
operator = ... 112
operator∗ 83, 84, 89, 94, 98
operator+ 73, 83, 88, 93, 97
operator+= 74
operator− .. 73, 83, 88, 93, 94, 97
operator<< .. 75, 80, 85, 90, 127, 129
operator= 72, 79, 82, 88, 128
operator>> ... 75, 80, 85, 90, 130
operator% 74, 79
operator∗ 72, 79
operator∗= 74, 79
operator* 101
operator+ 102
operator- 102
operator< 102
operator<< 103
operator== 103
operator> 102
operator>> 103
operator^ 89
Ortsbereich 163
Ortsfrequenzbereich 164

Pal-Stamdard 7
PERSON_CALC
  calculate_PERSON_CALC .. 114
  PERSON_CALC 114
  Print_left_HeadAndBody . 114
  Print_right_HeadAndBody 115
  printbinarymatrix 115
PERSON_CALC 114
PIC_MANAGER
  determine_source 109
  get_binary_pic 111
  get_camera_pic 111
  get_ram_pic 111
  PIC_MANAGER 109
  put_pictures 110
  read_display 110
  switch_cameras 110
  write_display 110
PIC_MANAGER 109
Print_left_HeadAndBody 114
Print_right_HeadAndBody 115
print_to_stream 132
printbinarymatrix 115
put_actions 121, 131, 155
put_direction 156
put_height 122
put_mu 71
put_pictures 110
put_position 155
put_time 99

read_action 148
read_actions 140
read_alarm_type 147
read_c 92, 96
read_day 100
read_display 110
read_edge_1 135
read_edge_2 135
read_exitname 147

read_form 134
read_h 79
read_height 131, 141
read_hidden_exit 142
read_hours 100
read_hundredth 99
read_lower_left 133
read_lr 78
read_lt 78
read_minutes 100
read_month 100
read_name 117, 124
read_near_door 142
read_pos 147
read_res_x 129
read_res_y 129
read_rr 78
read_rt 78
read_seconds 100
read_size 72
read_square_length 126
read_target_room 136
read_upper_right 133
read_visibility 141
read_x 82, 86
read_x_length 125
read_y 82, 86
read_y_length 126
read_year 101
read_z 87
RECTANGLE
    read_lower_left 133
    read_upper_right 133
    RECTANGLE 132
    write_position 133
RECTANGLE 132
recursion 116
registrate 130
request_alarm_status() 144
RGB 163
RLC-Verfahren 7
ROOM
    get_human 118
    get_image 118
    get_model 117
    get_surveillance 118
    load 118
    read_name 117
    ROOM 117
    save 119
    write_name 117
ROOM 117
Rot 47
Rotanteil 7

save 119, 124, 143, 149
scale 84, 89, 91, 96
searchheadarea 108
set_camera_data 105
set_camera_position 113
set_downwards_value 152
set_falling_value 153
set_lying__value 150
set_lying_ground_value 150
set_name 119
set_no_recognition_value ... 150
set_parameter 77
set_rolling_value 151
set_sitting_ground_value ... 151
set_sitting_value 151
set_standing_value 152
set_timer 104
set_upwards_value 153
set_walking_value 152
Sobel-Operator 163
SQUARE

read_actions 140
read_height 141
read_hidden_exit 142
read_near_door 142
read_visibility 141
SQUARE 139, 140
write_actions 140
write_height 140
write_hidden_exit 142
write_near_door 141
write_visibility 141
SQUARE 139, 140
still_running 104
SURVEILLANCE
SURVEILLANCE 144
alarm_accept 143
alarm_cancel 143
alarm_discard 144
load 143
request_alarm_status() . 144
save 143
SURVEILLANCE 143
SURVEILLANCE 143
swap_cams 120
switch_cameras 110

TIME
TIME 99
actual_time 99
is_negative 101
operator* 101
operator+ 102
operator- 102
operator< 102
operator<< 103
operator== 103
operator> 102
operator>> 103
put_time 99
read_day 100
read_hours 100
read_hundredth 99
read_minutes 100
read_month 100
read_seconds 100
read_year 101
TIME 98, 99
TIME2sec 101
TIME 98, 99
TIME2sec 101
TIMER
TIMER 104
set_timer 104
still_running 104
TIMER 104
TIMER 104
to_exit 137
Tracking 6, 47
Trackingalgorithmus 7
TRIANGLE
read_form 134
write_position 134

union_with_singleton 73
update_action_history 149

VECTOR2D
length 84
normalize 84
operator* 83, 84
operator+ 83
operator- 83
operator<< 85
operator= 82
operator>> 85
read_x 82

read_y 82
scale 84
VECTOR2D 81
write_x 82
write_y 82
VECTOR2D 81
VECTOR3D
length 90
normalize 90
operator∗ 89
operator+ 88
operator− 88
operator<< 90
operator= 88
operator>> 90
operator^ 89
read_x 86
read_y 86
read_z 87
scale 89
VECTOR3D 86
write_x 87
write_y 87
write_z 87
VECTOR3D 86

Weiß 47
WINDOW
get_single_movement 139
WINDOW 138
write_edge 138
write_movement 138
WINDOW 138
write_action 148
write_actions 140
write_alarm_type 146
write_c 93, 97
write_display 110
write_edge 138
write_edge_1 136
write_edge_2 136
write_exitname 147
write_height 131, 140
write_hidden_exit 142
write_movement 138
write_name 117
write_near_door 141
write_pos 147
write_position 133, 134
write_res_x 129
write_res_y 129
write_target_room 136
write_visibility 141
write_x 82, 87
write_y 82, 87
write_z 87

YUV-Farbmodell 162

Zyklop 6

Literaturverzeichnis

[24795] Projektgruppe 247. *Zyklop*. Universität Dotrmund, 1995.

[27796] Projektgruppe 277. *Argus*. Universität Dotrmund, 1996.

[AP96] Ali Azarbayejani and Alex Pentland. *Real-Time self-calibrating stereo person tracking using 3-D shape estimation from blob features*. Perceptual Computing Technical Report Nr.363. MIT Media Laboratory, 1996.

[AWDP96] Ali Azarbayejani, Christopher Wren, Trevor Darrell, and Alex Pentland. *Pfinder: Real-Time Tracking of the Human Body*. Perceptual Computing Technical Report Nr.353. MIT Media Laboratory, 1996.

[AWP96] Ali Azarbayejani, Christopher Wren, and Alex Pentland. *Real-Time 3-D Tracking of the Human Body*. Perceptual Computing Technical Report Nr.374. MIT Media Laboratory, 1996.

[NP92] Randal C. Nelson and Ramprasad Polana. *Qualitative Recognition of Motion Using Temporal Texture*. San Diego, CA, 555-559. Proc. DARPA Image, Understanding Workshop, 1992.

[NP94a] Randal C. Nelson and Ramprasad Polana. *Detecting Activities*. Journal of Visual Communication and Image Representation, 1994.

[NP94b] Randal C. Nelson and Ramprasad Polana. *Low Level Recognition of Human Motion (or How to Get Your Man without Finding his Body Parts)*. Austin, TX. IEEE Computer Society Workshop on Motion of Nonrigid and Articulate Objects, 1994.

[PN92] Ramprasad Polana and Randal C. Nelson. *Recognition of Motion from Temporal Texture.* Champaign, Illinois, 129-134. Proc. IEEE Conference on Computer Vision and Pattern Recognition, 1992.

[RCG93] Richard E. Woods Rafael C. Gonzales. *Digital Image Processing.* ISBN 0-201-50803-6. Addison-Wesley-Publishing Company, 1993.

[Röt95] Hans Röttger. *Echtzeit-Interaktion auf Basis visueller Bewegungserfassung.* Universität Dotrmund, 1995.