Multimodale Interaktion mit einem mobilen Roboter

Teilnehmer

Meike Brink
Bernd Focke
Markus Gärtner
Patrick Knabben
Dinu Niessner
Viktor Peters
Martin Saerbeck
Thorsten Spexard
Andreas Vangerow
Torsten Vollmann

Betreuer

Dr. Ing. G. Fink
Dipl. Ing. J. Fritsch
Dipl. Inform. S. Lang
Dipl. Inform. M. Kleinehagenbrock

17. Juli 2003

Inhaltsverzeichnis

1 Einleitung

Das Projektseminar multimodale Interaktion mit einem mobilen Roboter im Wintersemester 2002/2003 beschäftigte sich mit den verschiedenen Erkennungssystemen eines mobilen Roboters und deren Weiterverarbeitung. Das Ziel des Projektes war es, Grundlagen einer Mensch-Maschine-Interaktion mit einem Roboter zu verwirklichen. Dabei war die Mensch-Mensch-Interaktion unser Vorbild. Wenn ein Mensch einen Raum betritt, um mit anderen Menschen zu kommunizieren, muss er zuerst feststellen, ob sich überhaupt ein Gesprächspartner im Raum befindet. Falls mehrere Menschen im Raum sind, muss er feststellen, welche der Personen vielleicht spricht. Er möchte sie nicht unterbrechen oder vielleicht spricht sie auch gerade zu ihm. Diese Lokalisation muss der Roboter auch leisten. Sie findet auf mehreren Ebenen statt. Der Scanner des Roboters befindet sich auf Beinhöhe und sucht im Raum nach Beinpaaren. Die zwei Mikrofone dienen als Gehör. Mit ihnen kann festgestellt werden, aus welchen Richtungen Geräuschquellen kommen. Mit Hilfe der Kamera auf dem Roboter können Gesichter erkannt und in einem weiteren Schritt den Gesichtern Namen zugeordnet werden.

Genauso wie der Mensch die verschiedenen Eindrücke durch Ohren und Augen verarbeitet und dann danach agiert, so muss das auch der Roboter können. Der Roboter muss die verschiedenen Sensordaten (Mikrofone, Scanner, Kamera) auswerten. Aus allen Daten bestimmt er, ob eine Person gefunden hat und diese auch als Kommunikationspartner wählt. Es wurde eine Aufmerksamkeitssteuerung entwickelt, die bei mehreren gefundenen Personen entscheidet, wer der Kommunikationspartner ist. Er bewegt die Kamera oder sich selbst zum Kommunikationspartner und signalisiert seine Aufmerksamkeit. Bei einer Mensch-Mensch-Interaktion schauen sich die Kommunikationspartner auch an. Aus diesem Grund gab es drei Projektgruppen. Die Sprachlokalisation, die Gräuschequellen im Raum bestimmt, die Gesichtserkennung, die mit Hilfe der Kamera Gesichter im Raum erkennt und das *Anchoring*, welches die Sensordaten auswertet, den jeweiligen Kommunikationspartner auswählt und die Aufmerksamkeit auf ihn richtet.

Die neue Gesichtserkennung baut auf einem maschinellen Lernansatz für visuelle Objekterkennung auf, der in der Lage ist, Bilder in kürzester Zeit zu verarbeiten und gleichzeitig eine hohe Erkennungsrate besitzt. Das System baut auf drei Grundideen auf. Zum einen werden wir eine Bildrepräsentation benutzen, - Integral Image genannt - die es uns erlaubt, die Features, die wir zur Gesichtserkennung benutzen, sehr schnell berechnen zu können. Als zweites benutzen wir einen Lernalgorithmus auf dem AdaBoost Ansatz basierend, der diejenigen Features mit dem größten Aussagegehalt für unser Klassifikationsproblem aus einem großen Satz an Features herausfiltert und sehr effiziente Klassifikatoren ausgibt. Die dritte Grundidee verschachtelt mehrere dieser Klassifikatoren zu einem größeren Klassifikator in einer kaskadierten Struktur, welche sehr früh Bildhintergrundstrukturen verwirft und sich in späteren mehr und mehr auf Gesichter konzentriert. Unsere Gesichtserkennung erreicht durch die genannten Konzepte Echtzeitperformance und kann

mehrere Bilder innerhalb einer Sekunde verarbeiten.

Die Sprachlokalisation erkennt Geräuschquellen im Raum und kann ihre Richtungen angeben. Dies geschieht mit zwei Mikrofonen, die wie zwei Ohren am Roboter montiert sind. Ein Signalauschnitt vom linken Mikrofon wird mit dem Signal aus dem rechten Mikrofon verglichen, da das Geräusch zu unterschiedlichen Zeiten an den Mikrofonen ankommt. Dieser zeitliche Versatz wird berechnet und aus ihm der Winkel zur Quelle ermittelt. Um nicht zu viele Störgeräusche zu haben, werden bestimmte Frequenzen herausgefiltert, die nicht mit der menschlichen Stimmlage übereinstimmen. So ist es möglich mehrere Geräuschquellen zu lokalisieren, die potentielle Kommunikationspartner sein könnten.

Der Bereich Anchoring wurde überarbeitet. Die Aufmerksamkeitssteuerung des Roboters sollte robuster werden, damit die „Kommunikation“ zwischen Maschine und Mensch verbessert wird. Dazu müssen die Sensordaten (Gesicht, Pullover, Sprache) so verarbeitet werden, dass der Roboter die „*Person of Interest*“ als Gesprächspartner erkennt und sein Verhalten ändert.

2 Gesichtserkennung

2.1 Einleitung

Das Gesicht eines Menschen ist kennzeichnend für seine Kommunikation. Daher gibt es viele Ansätze für die Motivation, die hinter der Gesichtserkennung steckt. Zum einen wird die Gesichtserkennung zur Personen-Identifikation benötigt. Der Roboter kann somit die jeweilige Person direkt mit Namen ansprechen. Zudem kann diese Person vergangenen Aufzeichnungen zugeordnet werden, wenn zum Beispiel bei einer Interaktion die Person für kurze Zeit vollständig aus dem Sensorbereich des Roboters gerät. Auch ein Erkennen von Emotionszuständen, sowie das Erkennen eines Gesichtes trotz unterschiedlicher Blickrichtungen ist von großem Interesse. Die Gesichtserkennung leistet in diesem Projekt einen wesentlichen Beitrag zu dem Anchoring-Verfahren, bei dem die Detektion des Gesichtes und die Erfassung der Beine zusammenwirken.

Aber erst durch die eindeutige Detektion kann überhaupt eine Interaktion stattfinden. Die Schwierigkeiten liegen hier bei den variierenden Merkmalen eines Gesichtes und den äußeren Umständen. Eine gute Gesichtsdetektion sollte das Gesicht trotz einer Brille oder einem Bart erkennen. Weitere Eigenschaften wie Haaransatz zum Beispiel sollten keinen negativen Einfluss auf die Detektion haben. Zu den äußeren Umständen gehören die Beleuchtungsverhältnisse sowie die jeweilige Lokalisation des Gesichtes im Bild. Je weiter die Person von der jeweiligen Kamera entfernt ist, ist das Gesicht größer oder kleiner.

Die bisherige Gesichtserkennung des Roboters basierte auf einer Hautfarben- Segmentierung. Jedes einzelne Pixel wird mit der vorgegebenen Hautfarbe abgeglichen. Wird ein solcher Bereich erkannt, werden diese Pixel zu Regionen zusammengefasst. Nun wird die gefundene Region auf Gesicht oder Nicht-Gesichter verifiziert. Dabei wird das gesamte Bild in immer größer werdenden Stufen durchsucht, um zu gewährleisten, dass auch Gesichter in unterschiedlicher Größe gefunden werden. Diese Klassifizierung erfolgt mit der *Eigenface* Methode, die auf dem Verfahren der Karhunen-Loewe-Transformation, auch *Principal-Component-Analysis* (PCA) genannt, basiert [16]. Ziel des Verfahrens ist es, charakteristische Informationen aus den Gesichtern der Trainingsmenge zu extrahieren und ein neues Gesicht als gewichtete Summe dieser Informationen zu speichern. Mit diesem *Eigenface* wird an verschiedenen Positionen und in unterschiedlichen Skalierungen ein Rekonstruktionsfehler ermittelt. Unterschreitet der Fehler einen festgelegten Schwellwert, befindet sich an dieser Position mit hoher Wahrscheinlichkeit ein Gesicht. Durch einen Gradientenabstieg über den Rekonstruktionsfehler kann das Verfahren beschleunigt werden.

Der Erfolg dieser Vorgehensweise hängt stark von der Vorsegmentierung ab, die jedoch bei verschiedenen Beleuchtungen und unterschiedlichen Hautfarben variiert. Wird ein Gesicht erkannt, wird die Farbe in dem Bereich des lokalen Minimums als Hautfarbe gespeichert. So ist es zum Beispiel möglich, dass Gesicht und hautfarbene

Möbel zu einer Region zusammengefasst werden; dann ist durch die größere Region und den verschobenen Regionen-Schwerpunkt der Ansatzpunkt der Suche (des Scannens) falsch und das Gesicht wird i.d.R. nicht gefunden. Ein weiterer Nachteil ist der hohe Zeitaufwand, der benötigt wird, das gesamte Bild auf die Hautfarbenregionen zu durchsuchen.

Wir haben uns deshalb für ein merkmalbasiertes Verfahren entschieden, das nicht von Farbinformationen abhängt und effizient zu implementieren ist.

2.2 Neuer Ansatz

In diesem Abschnitt möchten wir die zugrunde liegenden Techniken unserer Gesichtserkennung vorstellen, um in den folgenden Abschnitten detaillierter darauf einzugehen.

Unser Ansatz stützt sich hauptsächlich auf die beiden Paper von Paul Viola & Michael Jones [19, 18] und auf einen Teil aus den Papern von Zhang et al. [20, 21], die auf den ersten beiden aufbauen.

Der neue Ansatz klassifiziert Bildausschnitte. Dabei finden alle möglichen Bildausschnitte in allen Größen und an allen Positionen Beachtung. Dadurch ergibt sich eine Vielzahl an zu untersuchenden Bildausschnitten. Damit das Verfahren in Echtzeit alle Gesichter innerhalb eines Bildes finden kann, muss es einen einzelnen Bildausschnitt schnell verarbeiten können.

Auf unterster Ebene des Klassifikators setzen deshalb Merkmale (*engl. Features*, Kap. 2.3, S. 8) an, die mit konstantem Aufwand – nach einer einfachen Vorverarbeitung des Bildes – berechnet werden können. Durch jedes Merkmal wird jedes 20×20 Pixel Bild auf eine diskrete Zahl reduziert. Für die Vorverarbeitung ist eine einfache Iteration über die Pixel des Bildes notwendig. Dabei wird das sogenannte *Integral-Image* (Kap. 2.3.2, S. 10) berechnet. Die verwendeten Merkmale lassen sich pro Bildausschnitt mit vier Zugriffen auf dieses *Integral-Image* berechnen. Der Aufwand ist damit konstant. Für ein 20×20 Pixel Fenster existiert eine Vielzahl derartiger Merkmale.

Für jedes dieser Merkmale wird zunächst ein *Weak*-Klassifikator (Kap. 2.4.1, S. 12) trainiert, um für jedes Merkmal einen einfachen Schwellwertklassifikator zu erhalten. Anschließend sucht der Lernalgorithmus *AdaBoost* (Kap. 2.4.2, S. 14) eine vorher bestimmte Anzahl (Kap. 2.5.2, S. 18) an *Weak*-Klassifikatoren aus dieser Menge heraus und fügt diese zu einem *Strong*-Klassifikator (Kap. 2.4.1, S. 14) zusammen. *AdaBoost* wählt dabei genau die *Weak*-Klassifikatoren aus, die für die Menge der Trainingsbilder am besten die Gesichter von den Nicht-Gesichtern trennen kann.

Durch serielles Anwenden von immer komplexer werdenden starken Klassifikatoren auf Gesichtsbereiche schließen wir möglichst früh einen überwiegenden Teil der Nicht-Gesichter von weiteren, komplexeren Klassifikationsstufen aus. Die zugrunde liegende Struktur hierfür ist eine *Kaskade* (Kap. 2.5, S. 17) aus starken Klassifikatoren. Unser System ist in der Lage, mit der ersten Stufe der Kaskade bereits 80% aller Nicht-Gesichter als solche zu klassifizieren, wobei im ersten Klassifikator der

Kaskade lediglich zwei Merkmale der Klassifikation zugrunde liegen. Damit sinkt die anfangs sehr hohe Anzahl an zu klassifizierenden Bildausschnitten sehr schnell, während sich spätere Klassifikationsstufen mit weitaus mehr Merkmalen auf weniger, aber dafür schwerer zu klassifizierende Bildausschnitte konzentrieren.

Diese Strategie findet ebenfalls Anwendung in der *Detektorenpyramide* (Kap. 2.6, S. 21), die zur Klassifikation von Gesichtern mit verschiedenen Drehungen in der horizontalen Ebene eingesetzt wird. Es handelt sich hierbei um eine Strategie, die sowohl den Aspekt „vom Groben zum Feinen“ als auch den Aspekt „vom Einfachen zum Komplexen“ beachtet. Der Grad der Kopfdrehung kann bis auf eine Genauigkeit von ca. 10° bestimmt werden. Auf oberster Stufe der Pyramide werden hierbei Gesichter mit beliebiger Drehung erkannt. In der zweiten Stufe der Pyramide kommen drei Detektoren zum Einsatz, um zwischen drei groben Blickrichtungen zu unterscheiden, bevor in der dritten und letzten Stufe der Pyramide Blickrichtungen von jeweils ca. 10° Differenz in jedem Detektor Beachtung finden.

2.3 Merkmalsmenge

Für die Klassifikation wird ein Raum von Merkmalen (*engl. Features*) in einem 20×20 Bild bestimmt. Diese Merkmale sind Abbildungen des 20×20 Bildes auf einen Zahlenwert. Bei der Klassifikation wird die Verteilung der durch die Merkmale bestimmten Zahlenwerte für Gesichter und Nicht-Gesichter betrachtet. Die im folgenden beschriebenen Merkmale ermöglichen es aufgrund ihrer Definition, dass für die Anwendung der gesamten Merkmalsmenge auf ein Bild nur ein einziges Mal direkt auf die Pixelwerte des Bildes zugegriffen wird.

2.3.1 Merkmale

Die Abbildung eines Bildes auf einen Zahlenwert durch ein Merkmal muss für eine hohe Verarbeitungsgeschwindigkeit der Klassifikation möglichst schnell ablaufen. Für die von uns benutzten Merkmale wird im nächsten Abschnitt ein Algorithmus beschrieben, um diese Abbildung bei der Anwendung von vielen Merkmalen auf dasselbe Bild mit nur einem einzigen Auslesen der einzelnen Pixel des Bildes zu bestimmen.

Wir haben uns für die Merkmale aus Zhang et al.[20] entschieden. Diese Merkmale setzen sich aus gleichgroßen, rechteckigen Bereichen des Bildes zusammen. Es gibt Zwei-Block-Merkmale, bestehend aus zwei rechteckigen Bereichen und Drei-Block-Merkmale, bestehend aus drei Bereichen. Die Zuordnung eines einzelnen Zahlenwertes zu einem Merkmal soll durch den Helligkeitsunterschied zwischen den Blöcken eines Merkmals bestimmt werden.

Der Zahlenwert eines Merkmals berechnet sich durch die Summe der mit einem Faktor multiplizierten Summen der Pixelwerte der einzelnen Blöcke (Abb. 1). Die Faktoren für die Blöcke sind bei einem Zwei-Block-Merkmal fest vorgegeben mit $+2$ für den ersten Block und -2 für den zweiten Block. Bei einem Drei-Block- Merkmal

sind die Faktoren der Blöcke +1, +1 und −2.

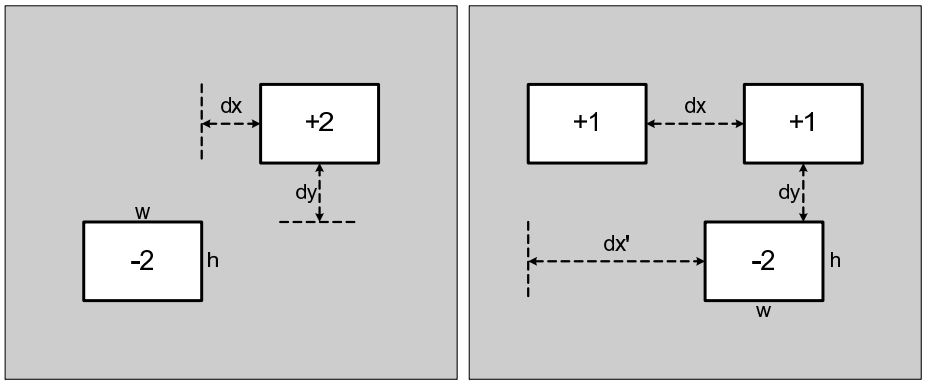

Abbildung 1: Konstruktion von Zwei- und Drei-Block-Merkmalen in einem 20×20 Bild: Die Pixelsummen der einzelne Blöcke werden jeweils mit einem Faktor multipliziert und anschliessend zusammenaddiert um den Zahlenwert des Merkmals zu bestimmen

Allein für Zwei-Block-Merkmale gibt es über 800.000 mögliche Kombinationen der zwei Blöcke in einem 20×20 Bild, wenn man keine Beschränkung für die Permutation der Parameter für die Bestimmung der Koordinaten vorgibt. Damit der Suchraum für den *AdaBoost*-Algorithmus und damit auch die Laufzeit des Trainings kürzer wird, muss die Anzahl der Merkmale begrenzt werden. Wir haben dazu versucht sinnvolle Regeln für die Generierung der Koordinaten der Zwei- und Drei-Block- Merkmale festzulegen (Abb. 1), damit etwa 130.000 Zwei-Block-Merkmale und 180.000 Drei-Block-Merkmale übrigbleiben, womit die Berechnung von Detektoren aber immer noch mehrere Wochen dauert:

- Breite und Höhe der Blöcke (w und h) haben nur die Werte $\{2, 3, 4, 5, 6\}$
- Maximaler Abstand der Blöcke (dx und dy) ist bei einem Zwei-Block-Merkmal drei Pixel und bei einem Drei-Block-Merkmal fünf Pixel
- Begrenzung des Verhältnisses von Höhe zu Breite der Blöcke: $1 : 4 < dx : dy < 4 : 1$
- Zwei der drei Blöcke eines Drei-Block-Merkmals sind auf derselben Höhe angeordnet

Bei der Auswahl der Werte haben wir hauptsächlich auf die Laufzeit des Trainings geachtet, die linear abhängig zu der Anzahl der Merkmale ist, und haben nicht untersucht, ob die Klassifikation ist, wenn die maximal mögliche Anzahl an Merkmalen verwendet wird.

2.3.2 Integral-Image

Die für die Auswertung der Merkmale benötigten Summen der Pixelwerte in einem Block des Bildes lassen sich über ein sogenanntes *Integral-Image* (auch *Summed-Area-Table* genannt) effizient und für beliebig große Blöcke mit konstantem Aufwand berechnen.

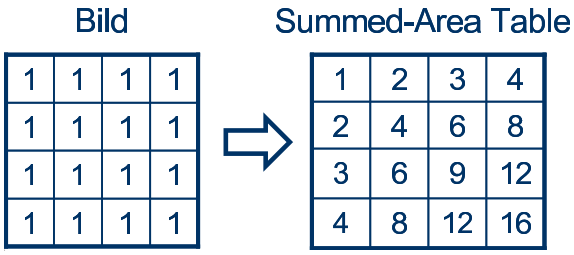

Abbildung 2: Beispiel eines Integral-Image: die Werte im Integral-Image sind die Summen der Pixel des Bildes in einem rechteckigen Bereich ausgehend vom ersten Pixel des Bildes links-oben bis zu dem Pixel mit denselben Koordinaten wie das betreffende Integral-Image-Pixel.

Das *Integral-Image* hat die Dimensionen des zu verarbeitenden Bildes. Für ein Bild B mit den Pixelwerten b_{ij} und einem *Integral-Image* M mit den Pixeln m_{ij} berechnen sich die Pixel des *Integral-Image* wie folgt (Abbildung 2):

$$m_{ij} = \sum_{x \leq i} \sum_{y \leq j} b_{xy}$$

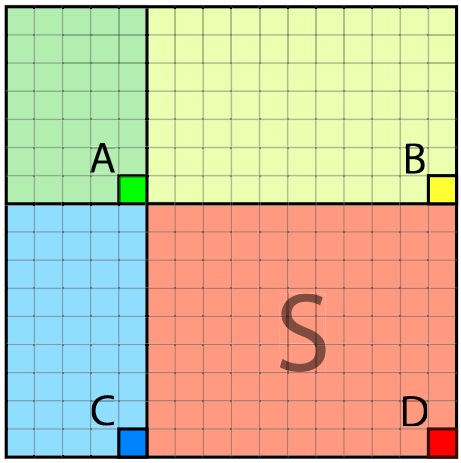

Abbildung 3: Berechnung der Pixelsumme eines rechteckigen Bereiches S durch die Integral-Image-Werte A, B, C, D: $S = D - B - C + A$

Ein beliebiger rechteckiger Bereich S aus Abbildung 3 lässt sich mit den *Integral-Image*-Werten an den Positionen A bis D berechnen.

$$S = D - B - C + A$$

Mit vier Speicherzugriffen lassen sich also die Pixelsummen für beliebig große Blöcke berechnen. Während des Trainings hat die Anwendung eines Merkmals auf ein zu untersuchendes 20×20 Bild also konstante Laufzeit.

2.3.3 Skalierung der Merkmale

Beim Training eines Detektors werden Merkmale immer auf 20×20 Bildern angewendet. Mit dem fertig trainierten Detektor sollen aber Gesichter in einem beliebig dimensionierten Bild gefunden werden. Die Gesichter sollen dabei auch andere Größen haben können als nur 20×20 Pixel. Dazu muss bei der Detektion ein beliebig großes, quadratisches Fenster im Bild untersucht werden können. Auf dieses quadratische Fenster werden die Merkmale genauso angewendet wie beim Training in einem 20×20 Bild, wobei man die Koordinaten der Blöcke des Merkmals aber für die neue Position und Größe umrechnen muß, was nachfolgend beschrieben wird.

Das *Integral-Image* wird für das Bild nur einmal berechnet und kann dann für alle zu untersuchenden Fenster benutzt werden, um die Merkmale anzuwenden. Dazu wird eine Transformation für die Koordinaten der Blöcke der Merkmale aus ihrem 20×20-Pixel Raum in das zu untersuchende quadratische Fenster bestimmt. Die Koordinaten der Merkmalsblöcke werden also transponiert und skaliert. Das Anwenden der skalierten Merkmale in dem Fenster des Bildes hat dieselbe konstante Laufzeit wie die ursprünglichen Merkmale für 20×20 Bilder.

Es tritt jedoch das Problem auf, dass die Eckkoordinaten der Merkmalsblöcke durch die angewendete Transformation keine diskreten Werte mehr sind und nicht auf dem diskreten Raster der Pixel des *Integral-Image* und des Bildes liegen. Wir haben festgestellt, dass man mit den Merkmalen keine Klassifikation mehr durchführen kann, falls man die Koordinaten nach dem Skalieren auf ganzzahlige Werte rundet.

Unsere erste Lösung für das Problem bestand darin, für reelle Koordinaten die vier Pixelwerte im *Integral-Image*-Raster zu betrachten, zwischen denen die reellen Koordinatenwerte sind. Aus diesen vier *Integral-Image*-Pixelwerten wird dann ein Wert interpoliert unter Berücksichtigung der durch die skalierten Koordinaten angegebenen Position.

Dadurch sind bei der Detektion zum Anwenden eines Merkmals, im Vergleich zu der Trainingsphase, viermal so viele Speicherzugriffe auf das *Integral-Image* nötig. Für die Berechnung der Pixelsumme eines Blockes benötigt man somit 16 Speicherzugriffe statt nur vier.

Aus Laufzeitgründen wird in der momentanen Realisierung eine andere Lösung für dieses Problem benutzt, so dass doch nur vier Speicherzugriffe pro Pixel-Block benutzt werden. Dazu werden die Koordinaten eines Blockes nach der Transformation zum Auslesen der Werte aus dem *Integral-Image* auf ganzzahlige Werte gerundet. Wir haben festgestellt, dass der Detektor noch hinreichend gut arbeitet, wenn man danach die mit dem *Integral-Image* bestimmte Pixelsumme mit dem Verhältnis der

Größe des Pixel-Blockes mit reellen Koordinaten zu der Größe des Pixel-Blockes mit ganzzahligen Koordinaten, multipliziert.

Beispielsweise hat ein 2×3 Block bei einer Skalierung von $1,5$ eine Größe von $3 \times 4,5 = 13,5$ Pixeln und in gerundeten Koordinaten eine Größe von $3 \times 5 = 15$ Pixeln. Die aus dem *Integral-Image* bestimmte Pixelsumme wird dann mit einem Korrekturwert $13,5 : 15 = 0,9$ multipliziert.

Durch diese zweite Lösung ist die Detektion schlechter, aber immer noch hinreichend gut und annähernd viermal schneller.

2.4 Der Lernalgorithmus

Für ein zu klassifizierendes Bildfenster[1] können wir mehr als 300.000 Merkmale heranziehen. Obwohl ein Merkmal für einen Bildausschnitt schnell berechnet werden kann, ist es nicht praktikabel für alle Bildausschnitte alle Merkmale auszuwerten und trotzdem in der Anwendung akzeptable Geschwindigkeiten zu erhalten. Deshalb müssen nur einige wenige Merkmale, die die Gesichtzüge am besten beschreiben können, ausgewählt werden.

Zum Selektieren der relevanten Merkmale benutzen wir eine etwas modifizierte Form von *AdaBoost*. Boosting ist eine der bedeutendsten Ideen der letzten zehn Jahre auf dem Gebiet der Lernalgorithmen. Es ermöglicht die Ergebnisse vieler äußerst schwacher (*engl.* weak) Klassifikatoren zu einem kombinierten, „starken" (*engl.* strong) Klassifikator zusammenzufassen und somit das Klassifikationsergebnis deutlich zu steigern (*engl.* boosted) [14, 3, 10].

Im Folgenden benutzen wir in Anlehnung an die englischen Bezeichnungen die Begriffe *Weak*- bzw. *Strong*-Klassifikator für einen schwachen bzw. starken Klassifikator.

2.4.1 Klassifikatoren

Weak-Klassifikator Ein *Weak*-Klassifikator ist sehr simpel aufgebaut und muss nur in der Lage sein, ein besseres Ergebnis zu liefern als eine zufällige Schätzung.

In unserer Implementierung wurde für jedes der oben vorgestellten Merkmale jeweils ein einfacher Schwellwertklassifikator verwendet. Es gibt also ca. 300.000 *Weak*-Klassifikatoren, aus denen später die besten ausgewählt werden müssen.

Wie bereits erwähnt, transformiert ein Merkmal einen kompletten Bildausschnitt in eine reelle Zahl. Wenn C_0 die Menge der Trainingsbeispiele für Hintergrundbilder und C_1 die der Gesichter ist, dann bildet ein Merkmal $f_j(x)$ für die beiden Mengen zwei Verteilungen wie in Abbildung 4(a) angedeutet. Der dazugehörige Klassifikator h_j trifft dabei mit Hilfe eines Schwellwerts θ_j eine Entscheidung, ob ein transformier-

[1] Alleine in der ersten Skalierungsstufe (Fenstergröße 20×20 Pixel) müssen bei einem 320×240 Bild bereits $(\frac{1}{2}(320-20))(\frac{1}{2}(240-20)) = 16500$ Bildfenster untersucht werden, wenn es in beide Richtungen jeweils um zwei Pixel verschoben wird.

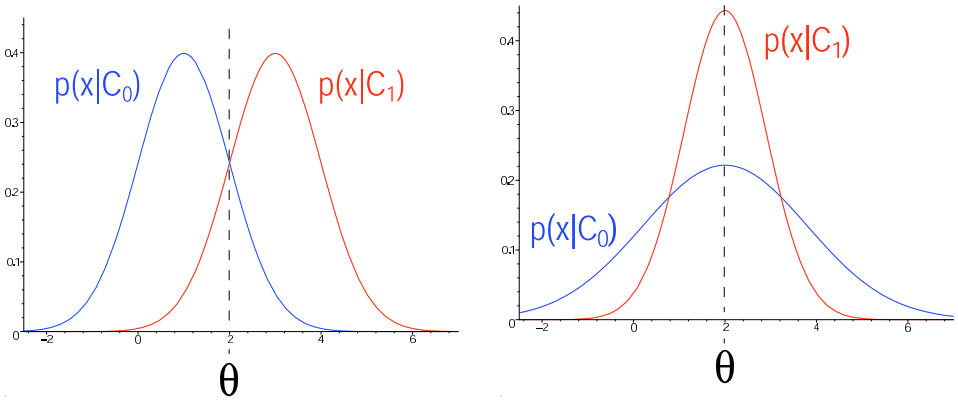

Abbildung 4: (a) Zwischen den beiden Klassen für Gesichter und Nicht-Gesichter muss für einen Weak-*Klassifikator ein geeigneter Schwellwert ermittelt werden. (b) Bei komplexeren Verteilungen ist es nicht immer möglich einen guten Wert zu finden.*

ter Bildausschnitt ein Gesicht bzw. Nicht-Gesicht ist. Mathematisch ausgedrückt:

$$h_j(x) = \begin{cases} 1 & \text{falls } p_j f_j(x) < p_j \theta_j \\ 0 & \text{sonst} \end{cases}$$

wobei x der zu untersuchende Bildausschnitt ist und p_j die Richtung des Vergleichs angibt. Bei $p_j = 1$ liegen die meisten positiven Beispiele unterhalb und bei $p_j = -1$ oberhalb des Schwellwerts.

Ein solcher Klassifikator kann auch als ein äußerst einfacher Entscheidungsbaum mit einem Wurzel-Knoten und zwei Terminal-Knoten gesehen werden (*engl.* decision stump). Das Ziel beim Training dieses Klassifikators ist es, einen möglichst guten Schwellwert zu finden, der die positiven und negativen Beispiele am besten voneinander trennt. Jedoch ist es nicht immer möglich, einen optimalen Wert zu finden wie man in Abbildung 4(b) sehen kann.

In ersten Versuchen benutzten wir ein sehr einfaches und schnelles Verfahren, um den Schwellwert θ_j zu bestimmen. Dazu wurde θ_j wie folgt berechnet:

$$\theta_j = \frac{1}{2}\left(\frac{1}{|C_0|}\sum_{x \in C_0} f_j(x) + \frac{1}{|C_1|}\sum_{x \in C_1} f_j(x)\right)$$

Der Schwellwert wird also zwischen den beiden Mittelwerten der beiden Klassen positioniert und p_j ist demnach einfach das Vorzeichen der Differenz zwischen den Mittelwerten der negativen und positiven Klasse.

Später haben wir jedoch den optimalen Schwellwert gesucht, was ca. zweimal langsamer ist, jedoch bessere Klassifikationsergebnisse liefert und somit auch zur schnelleren Laufzeit während der Klassifikationsphase führt. Um den optimalen

Schwellwert zu finden, müssen alle Merkmalswerte aus C_0 und C_1 erstmal sortiert werden:

$$f_j(x_1) <= f_j(x_2) <= \ldots <= f_j(x_n), \qquad n = |C_0| + |C_1|$$

Außerdem wird zu jedem Merkmalswert vermerkt, zu welcher Klasse er gehört. Anschließend werden die sortierten Werte sequentiell vom Anfang bis zum Ende einmal durchlaufen, um zu analysieren, an welcher Stelle i die beiden Klassen am besten separiert werden. Dabei wird geprüft, wie viele Falsch-Klassifikationen sich an der festgelegten Stelle ergeben würden. Der Schwellwert bildet sich dann aus

$$\theta_j = \frac{1}{2}\left(f_j(x_i) + f_j(x_{i+1})\right)$$

Strong-Klassifikator Wie bereits erwähnt, können mehrere *Weak*-Klassifikatoren zu einem einzigen, leistungsfähigerem Klassifikator, dem so genannten *Strong*-Klassifikator, zusammengeschaltet werden. Dabei bedient man sich eines gewichteten Mehrheitsentscheids über die Ausgaben der einzelnen schwachen Klassifikatoren:

$$H(x) = \begin{cases} 1 & \text{falls } \sum_{t=1}^{T} \alpha_t h_t(x) > \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{sonst} \end{cases}$$

wobei α_t der Gewichtungsfaktor für den *Weak*-Klassifikator h_t ist und die Summe $\gamma = \frac{1}{2} \sum_{t=1}^{T} \alpha_t$ der Schwellwert für den Mehrheitsentscheid ist.

Welche *Weak*-Klassifikatoren und die dazu gehörenden Gewichtungsfaktoren zu wählen sind, wird durch *AdaBoost* bestimmt.

2.4.2 AdaBoost

Zu den bekanntesten Boosting-Algorithmen gehört *AdaBoost* [15], was von „adaptive boosting“ abgeleitet ist. In einem iterativen Prozess werden *Weak*-Klassifikatoren zu einem komplexeren Klassifikator zusammengefaßt, bis ein gewünschter niedriger Gesamttrainingsfehler erreicht wird. Jedem Beispiel wird ein Gewicht zugeordnet, das bestimmt, wie wichtig es für die nachfolgenden Trainingschritte ist. Wenn ein Trainingsbeispiel korrekt klassifiziert wurde, wird sein Gewicht verringert, bei einer falschen Klassifikation hingegen erhöht. Somit konzentriert sich *AdaBoost* auf die aussagefähigsten oder schwer erlernbaren Beispiele.

Genauer gesagt, werden zuerst alle Gewichte für eine Trainingmenge so gesetzt, dass sie eine gleichförmige Verteilung bilden. Bei jeder Iteration, wird ein *Weak*-Klassifikator ausgewählt, der die Trainingsmenge in Bezug zu ihrer Verteilung (den Gewichten) am besten kategorisiert. Als nächstes werden alle Gewichte erhöht bzw. erniedrigt, die von diesem *Weak*-Klassifikator falsch bzw. richtig klassifiziert wurden. Es entsteht eine neue Verteilung über die Trainingsmenge, die im nächsten Schritt des Prozesses zur Bestimmung des neuen *Weak*-Klassifikators herangezogen wird.

In Pseudocode lässt sich *AdaBoost* für unseren Fall wie folgt darstellen:

- Gegebene Trainingsbeispiele $(x_1, y_1), \ldots, (x_n, y_n)$, wobei x_i ein 20×20 Pixel großes Bild ist und $y_i = 0$ bei Nicht-Gesichtern bzw. $y_i = 1$ bei Gesichtern.
- Initialisiere Gewichte:
$$w_{1,i} = \begin{cases} \frac{1}{2m} & \text{für } y_i = 0 \\ \frac{1}{2l} & \text{für } y_i = 1 \end{cases}$$
mit m und l Anzahl der negativen bzw. positiven Beispiele.
- Iteriere über $t = 1, \ldots, T$:
  1. Normalisiere Gewichte:
  $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
  (die Gewichte $w_{t,i}$ bilden somit eine Verteilung: $\sum_{i=1}^{n} w_{t,i} = 1$)
  2. Trainiere bei Beachtung der Verteilung w_t für jedes Merkmal einen *Weak*-Klassifikator[2] h_j und bestimme den Klassifikationsfehler
  $$\epsilon_j = \sum_i w_{t,i} |h_j(x_i) - y_i|$$
  Es werden also alle Gewichte aufaddiert, deren Beispiele falsch klassifiziert wurden.
  3. Wähle den *Weak*-Klassifikator mit dem kleinsten Fehler ϵ_t und füge ihn zusammen mit seinem Gewicht α_t zu der Menge der ausgewählten *Weak*-Klassifikatoren hinzu. Wobei
  $$\alpha_t = \log \frac{1}{\beta_t}$$
  mit $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
  4. Aktualisiere Gewichte:
  $$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$
  wobei $e_i = 0$ bzw. 1, falls x_i korrekt bzw. falsch klassifiziert wurde.
- Der fertige *Strong*-Klassifikator bildet sich dann aus allen vorher gewählten *Weak*-Klassifikatoren:
$$H(x) = \begin{cases} 1 & \text{falls } \sum_{t=1}^{T} \alpha_t h_t(x) > \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{sonst} \end{cases}$$

[2]Um es nochmal herauszustellen: zu jedem *Weak*-Klassifikator gehört ein Merkmal. Bei 300.000 Merkmalen müssen auch für genau so viele *Weak*-Klassifikatoren die dazugehörenden Schwellwerte bestimmt werden. Das ist der rechenintensivste Teil, während eines Iterationsschritts.

Zu beachten ist hier, dass bei der Berechnung der *Weak*-Klassifikatoren und des Klassifikationsfehlers die gewichteten Trainingsbeispiele einfließen. Das bedeutet z.B., dass die Ergebnisse für das Training eines *Weak*-Klassifikators unterschiedlich ausfallen können, wenn die Trainingmenge unterschiedlich gewichtet wird. Das führt dazu, dass während der *AdaBoost*-Iterationsschritte ein *Weak*-Klassifikator bzw. das dazu gehörende Merkmal mehrmals aber mit verschiedenen Parametern gewählt wird.

In der Praxis kann kein Merkmal alleine besonders gute Ergebnisse liefern, so dass sich der Fehler $\epsilon_j = \sum_i w_{t,i}|h_j(x_i) - y_i|$ zu Anfang zwischen etwa 0,1 und 0,3 bewegt. In späteren Phasen wird es immer schwieriger Unterschiede zwischen den positiven und negativen Beispielen zu finden; somit steigt der Fehler auf 0,4 bis 0,5 an.

Es hat sich sogar gezeigt, dass bei einer so großen Merkmalsmenge nicht immer alle Merkmale betrachtet werden müssen. Sucht man zum Beispiel bei jedem Schritt zufällig nur einen Teil der Elemente aus dieser Menge aus und trainiert damit, verschlechtert sich der Fehler nur marginal. Diese leichte Modifikation von *AdaBoost* nennt sich *LazyBoost* [2]. Zurückzuführen ist dieses auf die Tatsache, dass einige Merkmale für bestimmte Situationen äquivalente Ergebnisse liefern. Wir haben in späten Phasen des Trainings die tatsächlich verwendete Merkmalsmenge auf 25% ihrer ursprünglichen Größe reduziert und erhielten einen erheblichen Zeitgewinn, ohne einen sonderlichen Einbruch in der Klassifikationsleistung zu bemerken.

In Abb. 5 sieht man ein Gesicht aus der Trainingmenge und wie sich die ersten beiden von *AdaBoost* gewählten Merkmale dazu verhalten. Mit diesen beiden Merkmalen ist es bereits möglich 80% der Hintergrund-Bildausschnitte auzusortieren und dabei immer noch 99% der Trainings-Gesichter zu erkennen. Obwohl hier nur Zwei-Block-Merkmale zu sehen sind, wurde bei unserer Trainingsmenge das Drei-Block-Merkmal in 33% der Fälle von *AdaBoost* gewählt.

Abbildung 5: Ein Trainingsbeispiel und die beiden ersten von AdaBoost *ausgewählten Zwei-Block-Merkmale. Es ist zu erkennen, dass die Regionen zwischen Nase und Augen für die Detektion besonders interessant sind.*

2.5 Klassifikatoren Kaskade

Aus Performanzgründen haben Viola & Jones [18] und Zhang [20] einen kaskadierten Klassifikator eingeführt. Die Klassifikatoren Kaskade besteht aus mehreren *Strong*-Klassifikatoren, die hintereinander auf einen Bildausschnitt angewendet werden. Jeder *Strong*-Klassifikator besteht dabei aus einer vorher festgelegten Anzahl an *Weak*-Klassifikatoren. Viola & Jones haben ebenfalls eine Vorgehensweise beschrieben, um die einzelnen Größen der *Strong*-Klassifikatoren dynamisch zu bestimmen. Aus Zeitgründen haben wir im Training jedoch die von Viola & Jones angegebenen Größen für die einzelnen *Strong*-Klassifikatoren benutzt.

2.5.1 Funktionsweise der Kaskade

Zweck der Kaskade soll es im späteren Klassifikationsprozess sein, dass möglichst viele Nicht-Gesichter keine weitere Beachtung in späteren Stufen der Kaskade finden. Prinzipiell arbeitet unser Detektor wie ein Filter, der alle Bildausschnitte, die keine oder nur geringe Gemeinsamkeiten mit einem Gesicht haben, möglichst früh von der Verarbeitung durch spätere Klassifikationsstufen ausschließt.

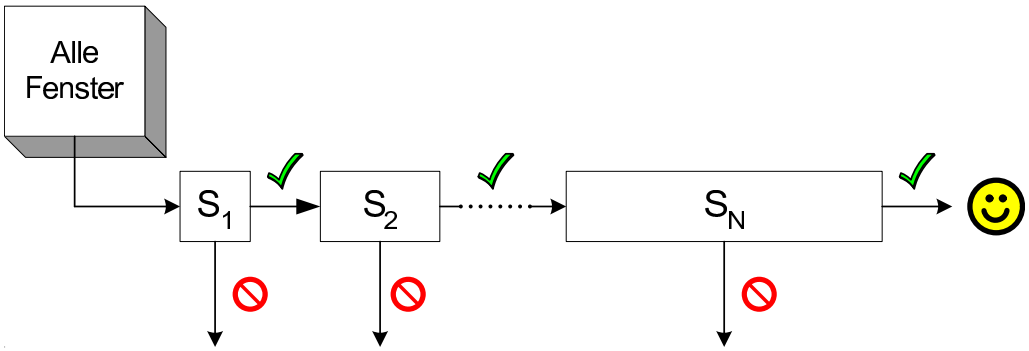

Abbildung 6: Funktionsweise der Klassifikatorenkaskade – Ein Subfenster wird zunächst vom ersten Strong*-Klassifikator der Kaskade klassifiziert. Lautet das Ergebnis Gesicht, wird das Subfenster an den folgenden* Strong*-Klassifikator zur weiteren Verarbeitung weitergeleitet. Im Falle einer Klassifikation als Nicht-Gesicht liefert die Kaskade als Gesamtergebnis für den Bildausschnitt Nicht-Gesicht.*

Daraus resultiert, dass, wenn ein *Strong*-Klassifikator in der Kaskade einen Bildausschnitt als Nicht-Gesicht klassifiziert, das Gesamtergebnis der Klassifikation für dieses Bild Nicht-Gesicht lautet (Abb. 6). Erkennt jeder *Strong*-Klassifikator der Kaskade das Bild als Gesicht, so ist das Ergebnis der Klassifikation des Bildauschnittes das Label Gesicht. Durch diese Struktur der Kaskade erhält man einen Gesamtklassifikator, der in Echtzeit laufen kann, sofern die frühen *Strong*-Klassifikatoren genügend Nicht-Gesichter von der weiteren Verarbeitung ausschließen.

Für das Training wird deshalb ein Qualitätsmerkmal für den trainierten Klassifikator benötigt. Wesentliche Erkenntnisse über die Güte des Klassifikators erhält man hierbei, wenn man den Anteil falsch klassifizierter Nicht-Gesichter auf einem Testset betrachtet. Die sogenannte *False-Positive-Rate* oder Mißklassifikationsrate gibt das Verhältnis zwischen als Gesicht erkannten Nicht-Gesichtern zu allen Nicht-Gesichtern an. Das Training kann damit als erfolgreich betrachtet werden, wenn die Erkennungsrate, die das Verhältnis von als Gesicht erkannten Gesichtern zu der Gesamtzahl der Gesichter angibt, möglichst groß und die Mißklassifikationsrate möglichst klein ist.

Die endgültige Qualität des Klassifikators lässt sich vorher approximieren. Legt man die Erkennungsrate und Mißklassifikationsrate jedes *Strong*-Klassifikators der Kaskade zugrunde, so lassen sich Erkennungs- und Mißklassifikationrate der gesamten Kaskade durch das Produkt der einzelnen *Strong*-Klassifikatoren approximieren [19]:

$$dr = \prod_i dr_i \quad \text{Erkennungsrate}$$

$$fpr = \prod_i fpr_i \quad \text{Mißklassifikationrate}$$

Der Index i steht dabei für die Erkennungs- bzw. Mißklassifikationsrate des iten *Strong*-Klassifikators der Kaskade. Somit lassen sich vor dem Training beim Bestimmen der Trainingsparameter gute Abschätzungen über die Qualität und die Performanz des zu trainierenden Klassifikators machen.

2.5.2 Training der Kaskade

Um eine Kaskade zu trainieren, gibt es zwei Ansätze. Zum einen kann die Kaskade mit fest vorgegeben Größen für die einzelnen *Strong*-Klassifikatoren berechnet werden. Um die Größen vorgeben zu können, benötigt man in diesem Fall entweder Erfahrung mit sinnvollen Werten für die Anzahl der *Weak*-Klassifikatoren in jedem *Strong*-Klassifikator oder man versucht, die Erfahrungswerte von erfolgreich trainierten Systemen zu benutzen und diese damit nachzubilden. Zum anderen haben Viola & Jones einen Algorithmus zur dynamischen Bestimmung der optimalen Klassifikatorgrößen vorgestellt. Auch für diesen Ansatz werden Erfahrungswerte benötigt, um die erforderlichen Trainingsparameter geschickt zu wählen. Beide unterscheiden sich nur leicht von einander. Im Folgenden gehen wir auf beide Vorgehensweisen ein.

Fest vorgegebene Klassifikatorengrößen Für das Training einer Kaskade mit fest vorgegebenen Klassifikatorengrößen werden zwei Parameter für jeden *Strong*-Klassifikator der Kaskade benötigt. Zum einen muss festgelegt werden, wie viele *Weak*-Klassifikatoren hinzuzufügen sind, zum anderen muss das Trainingsprogramm die zu erreichende Erkennungsrate dr_{min} für den jeweiligen *Strong*-Klassifikator sicherstellen. *AdaBoost* wählt für jeden *Strong*-Klassifikator die vorher bestimmte

Anzahl an *Weak*-Klassifikatoren aus. Der von *AdaBoost* berechnete Schwellwert $\gamma = \frac{1}{2}\sum_{t=1}^{T}\alpha_t$ liefert allerdings nicht immer die geforderte Erkennungsrate dr_{min}. Um aber sicher zu stellen, dass der *Strong*-Klassifikator die benötigte Erkennungsrate dr_{min} erfüllt, passen wir den von *AdaBoost* gelieferten Schwellwert des *Strong*-Klassifikators an. Bei der Schwellwertanpassung des *Strong*-Klassifikators sind die Erkennungsrate dr und die Mißklassifikationsrate fpr allerdings korreliert, so dass eine Anpassung des Schwellwertes, um die Erkennungsrate zu erhöhen, eine Erhöhung der Mißklassifikationsrate mit sich zieht. Somit muss der optimale Schwellwert gefunden werden, so dass der *Strong*-Klassifikator nach der Anpassung möglichst dr_{min} erfüllt und gleichzeitig die Mißklassifikationsleistung des Klassifikators gering bleibt. Hierbei haben wir es offensichtlich mit einem Optimierungsproblem zu tun, weshalb wir uns für ein 1d-Minimierungsverfahren entschieden haben, um den optimalen Schwellwert zu bestimmen.

Deshalb führen wir zunächst eine Fehlerfunktion ein, die es im folgenden Schritt zu optimieren gilt. In der Fehlerfunktion muss sowohl die aktuelle Erkennungsrate dr als auch die Mißklassifikationsrate fpr berücksichtigt werden. Des weiteren muss die geforderte Erkennungsrate dr_{min} Einfluss nehmen. Als letzter Parameter für die Fehlerfunktion muss der Schwellwert des Klassifikators γ einfließen.

$$f = \begin{cases} -4000(dr - dr_{min}) + 10fpr + 1 - \frac{1}{\gamma} & \text{wenn } dr - dr_{min} < 0 \\ 1000(dr - dr_{min}) + 10fpr + 1 - \frac{1}{\gamma} & \text{sonst} \end{cases}$$

erfüllt gerade die oben gestellten Bedingungen an eine Fehlerfunktion. Diese Fehlerfunktion basiert auf Erfahrungswerten während unserer Trainingstests und hat sich zur Schwellwertanpassung als sinnvoll erwiesen. Bei einer zu niedrigen Erkennungsrate besitzt die Funktion einen höheren Faktor als bei einer zu hohen Erkennungsrate. Damit erreichen wir, dass im Minimierungsprozess möglichst schnell die geforderte Erkennungsrate dr_{min} erfüllt wird. Anschließend wird der Schwellwert gesucht, für den die Mißklassifikationsrate fpr minimal ist.

Zur Suche des optimalen Schwellwertes benutzen wir ein Minimierungsverfahren, das auf die Verwendung von Ableitungen der Fehlerfunktion verzichtet. Als Ausgangspunkt betrachten wir zunächst den Schwellwert γ, den *AdaBoost* liefert. Ist die Erkennungsrate des *Strong*-Klassifikators zu gering, muss der optimale Schwellwert zwischen dem von *AdaBoost* gelieferten Schwellwert γ und dem Schwellwert 0 liegen, denn der Klassifikator erkennt bei einem Schwellwert von 0 jeden Bildausschnitt als Gesicht. Aus diesem Intervall wird nun ein weiterer Schwellwert hinzugenommen und anschließend mit den beiden Ausgangswerten und der zugehörigen Klassifikationsleistung des *Strong*-Klassifikators verglichen. Liegt die Erkennungsrate des Klassifikators mit dem neuen Schwellwert über der geforderten Erkennungsrate, so wird der Schwellwert 0 durch den neuen Schwellwert ersetzt. Im umgekehrten Fall wird der ursprünglich von *AdaBoost* gelieferte Schwellwert γ mit dem neuen Wert ersetzt. Anschließend setzt man das Verfahren auf dem neuen Intervall an. Verwendet man zur Teilung der Intervalle das Verhältnis des goldenen Schnittes, so kann man zei-

gen, dass das Verfahren in endlich vielen Schritten konvergiert. Das Verhältnis des goldenen Schnittes beträgt dabei $\frac{\sqrt{5}-1}{2} \approx 0.61803$.

Diese Schwellwertanpassung muss nach jedem trainierten *Strong*-Klassifikator der Kaskade vorgenommen werden, allerdings ist das Verfahren durch das Konvergenzverhalten des Algorithmus schnell und besitzt damit nur einen vernachlässigbaren Einfluss auf die Dauer des Trainings.

Da der endgültige Klassifikator prinzipiell als Filter eingesetzt wird, muss nach jedem hinzugefügten *Strong*-Klassifikator die Trainingsmenge für *AdaBoost* angepaßt werden. Dabei müssen aus der Trainingsmenge diejenigen Nicht-Gesichter gefiltert werden, die weiterhin als Gesicht erkannt werden. *AdaBoost* konzentriert sich dann beim Hinzufügen des nächsten *Strong*-Klassifikators auf gerade die Trainingsbeispiele, die bisher falsch klassifiziert wurden, so dass mit dem nächsten *Strong*-Klassifikator die Kaskade einen Großteil dieser Bilder herausfiltert.

Dynamische Klassifikatorengrößen Viola & Jones beschreiben in [19] einen Algorithmus, um die einzelnen Größen der *Strong*-Klassifikatoren in Abhängigkeit von den gewünschten Werten für die Erkennungsrate und Mißklassifikationsrate pro *Strong*-Klassifikator und der Gesamtmißklassifikationsrate zu bestimmen. Damit ergeben sich für das Training von dynamischen Klassifikatorgrößen folgende, notwendige Parameter:

- minimale Erkennungsrate pro *Strong*-Klassifikator dr^j_{min}
- maximale Mißklassifikationsrate pro *Strong*-Klassifikator fpr^j_{max}
- maximale Mißklassifikationsrate insgesamt fpr^{total}_{max}

Der Algorithmus in [19] sieht vor, dass so lange jeweils ein *Weak*-Klassifikator hinzugefügt und der Schwellwert des entstehenden *Strong*-Klassifikators auf die minimale Erkennungsrate dr^j_{min} angepaßt wird, bis der *Strong*-Klassifikator die maximale Mißklassifikationsrate fpr^j_{max} erreicht. Das Trainingsprogramm sollte dann solange derartige *Strong*-Klassifikatoren hinzufügen, bis der gesamte Detektor die maximale Mißklassifikationsrate insgesamt fpr^{total}_{max} erreicht.

Uns erschien beim Lesen von [19] merkwürdig, dass die Größen der *Strong*-Klassifikatoren in der Beschreibung sehr glatten Werten entsprechen. Des weiteren hatten wir in unseren Tests mit dynamischen Klassifikatorengrößen Probleme, da die zu erreichende Mißklassifikationsrate in den meisten Fällen zu niedrig gewählt war, so dass das Trainingsprogramm immer mehr *Weak*-Klassifikatoren zum aktuellen *Strong*-Klassifikator hinzugefügt hat. Aus diesem Grund haben wir uns für die Variante des Trainings mit fest vorgegebenen Klassifikatorengrößen entschieden und uns dabei an die von Viola & Jones angegebenen Klassifikatorengrößen gehalten.

2.6 Detektorenpyramide

Um Gesichter aus mehreren Drehungen zu erkennen, haben Zhang et al. [20] eine ähnliche Vorgehensweise wie bei den kaskadierten Detektoren von Viola & Jones [19] eingeführt. Die Pyramide erfüllt dabei zwei Zwecke: Zum einen ist es möglich, verschiedene Bereiche der Gesichtsdrehung zu erkennen, zum anderen verfolgt die Pyramide eine „vom Groben zum Feinen“ und „vom Einfachen zum Komplexen“ Strategie, welche die Echtzeitperformanz des Gesamtklassifikators erhalten soll.

Die Vorgehensweise ist einfach. Auf der obersten Ebene der Detektorenpyramide werden Gesichter im Bereich von -90° bis 90° Drehung von einem Detektor[3] erkannt (Abb. 7). Dabei entspricht eine 0° Drehung einem Frontalgesicht, eine -90° Drehung entspricht einem Gesicht in der Profilansicht nach rechts. Auf der zweiten Ebene haben Zhang et al. die Erkennung in drei Bereiche unterteilt: $[-90°, -30°]$, $[-30°, 30°]$ und $[30°, 90°]$. Den letzten Detektor brauchten Zhang et al. in dieser Ebene der Pyramide nicht trainieren, sondern haben lediglich die Merkmale des ersten Detektors gespiegelt. Auf der dritten Ebene sprechen Zhang et al. in [21] von neun Unterteilungen von Gesichtsdrehungen, in [20] von sieben Unterteilungen. Die Strategie ist hierbei die gleiche wie in der vorhergehenden Ebene der Pyramide. Es muss lediglich ein Teil der Detektoren trainiert werden, Detektoren für die umgekehrten Drehungen erhält man durch Spiegelung der Merkmale. Somit ergibt sich für die Pyramide der 1-3-9 Struktur ein Trainingsaufwand von acht Detektoren für das Gesamtsystem, für die Pyramide mit der 1-3-7 Struktur werden entsprechend sieben Detektoren notwendig. Obwohl sich dieses Konzept weiter verfeinern ließe, haben Zhang et al. in [20, 21] darauf verzichtet, da das System in dieser relativ simplen Struktur beste Ergebnisse lieferte.

Aus Zeitgründen konnten wir diese Ergebnisse leider nicht rekonstruieren, dieses wird ein Bestandteil weiterführender Arbeit an der Gesichtserkennung sein.

2.7 Realisierung

Abschließend geben wir in diesem Abschnitt Hinweise zu unserer Implementation, die wichtig für eine Reproduktion unseres Ergebnisses sein könnten.

Trainingsdaten Trainiert haben wir mit 5.000 Frontal-Gesichtern und 7.000 Nicht-Gesichtern. Die Menge der Gesichter besteht zu 70% aus sieben Personen, die mit der Kamera des Roboters (Sony EVI D-31) unter verschiedenen Beleuchtungssituationen aufgenommen wurden. Die restlichen 30% wurden aus Bildern aus dem Internet ausgeschnitten und um den Bildmittelpunkt im Uhrzeigersinn und entgegengesetzt leicht gedreht. Dieses hat zur Folge, dass auch etwas zur Seite geneigte Gesichter vom Detektor erkannt werden.

[3] Ein Detektor ist in diesem Fall eine Klassifikatorenkaskade mit einem oder mehreren *Strong*-Klassifikatoren

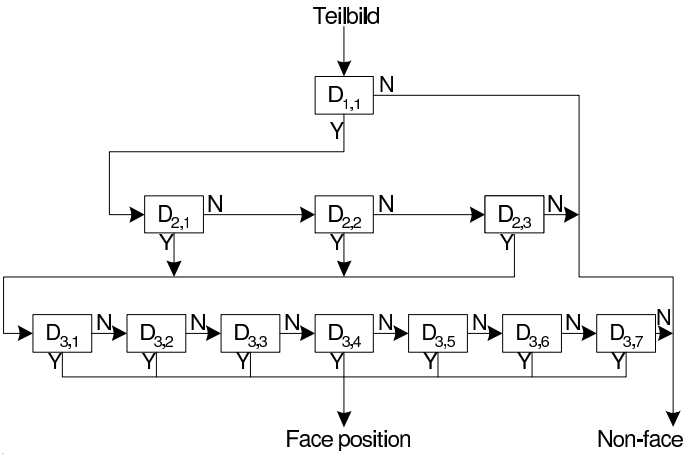

Abbildung 7: Funktionsweise der Detektorenpyramide. Ähnlich der Vorgehensweise bei dem kaskadierten Klassifikator, werden Nicht-Gesichter zunächst an weitere Detektoren der aktuellen Stufe weitergereicht. Erkennen diese ebenfalls kein Gesicht, wird der Bildausschnitt als Nicht-Gesicht klassifiziert.

Die Vorlagen für die Nicht-Gesichter sind verschiedene Fotos, aus denen zufällig die 7.000 beim Training benutzten 20×20 Pixel Bereiche ausgewählt werden. Die Nicht-Gesichter werden nur vor der Berechnung des ersten *Strong*-Klassifikators auf

Abbildung 8: Einige Beispiele für Gesichter (20×20 Pixel), die für das Training benutzt wurden.

diese Weise erzeugt.

Bei allen folgenden *Strong*-Klassifikatoren ist bereits ein Detektor vorhanden, der nur um einen neuen *Strong*-Klassifikator erweitert wird. Durch den vorhandenen Detektor kann man nun Nicht-Gesichter sinnvoller auswählen. Dazu werden alle möglichen 20×20 Pixel Bereiche in allen Vorlagen mit dem Detektor klassifiziert und alle korrekt als negativ bewerteten Bereiche nicht weiter betrachtet. Die positiv und damit falsch klassifizierten Bereiche werden danach sortiert, wie „gut“[4] sie erkannt wurden. Beim Training neuer *Strong*-Klassifikatoren erzielt man die besten Ergebnisse, wenn man aus dieser sortierten Liste gleichmäßig verteilt die Nicht-Gesichter für das Training entnimmt.

Durch diese Suche nach Nicht-Gesichtern belegt unser Trainingsprogramm zeitweise mehrere hundert Megabyte RAM. Der Speicherplatz für die Trainingsdaten liegt ebenfalls in dieser Größenordnung.

Gruppierung erkannter Bildbereiche Bei der Anwendung des Detektors auf ein Bild mit Gesichtern, werden für jedes Gesicht im allgemeinen mehrere leicht verschobene und unterschiedlich skalierte Bereiche positiv klassifiziert. Die Position und Größe, die wir für ein Gesicht bestimmen, wird bei unserem Programm aus den Koordinaten der sich überlappenden Bereiche gemittelt. Die Anzahl der sich überlappenden Bereiche, aus denen ein Gesicht bestimmt wird, wird gezählt. Diese Zahl benutzen wir um fälschlich positiv erkannte Nicht-Gesichter zu erkennen.

Bei 320×240 Bildern muss bei uns eine Gesichtsposition aus mindestens zwei positiv klassifizierten Bereichen bestimmt sein. Durch diese Bedingung wurden Mißklassifikationen bei uns zuverlässig aussortiert, da positiv bewertete Nicht-Gesichter bei unseren Versuchen mit unserem Detektor im Allgemeinen als einzelne positiv klassifizierte Bildbereiche auftraten.

Kontrasterhöhung Durch die spezielle Wahl unserer Merkmale ist es auch möglich, eine schnelle Helligkeitskorrektur vorzunehmen, um von unterschiedlichen Beleuchtungssituationen unabhängiger zu werden [8]. Für einen Bildausschnitt mit den Pixelwerten b_{ij}, lässt sich eine Kontrastanpassung wie folgt durchführen:

$$\bar{b}_{ij} = \frac{b_{ij} - \mu}{\sigma}$$

wobei μ der Mittelwert und σ die Standardabweichung der Helligkeitswerte in dem Bildausschnitt ist.

Der Mittelwert μ ist hier jedoch noch nicht von Belangen, da sich dieser Teil der Formel bei der Berechnung der Merkmale auflöst. Als Beispiel betrachten wir dafür das Zwei-Block-Merkmal[5], das aus zwei rechteckigen Bereichen, die wir hier B_1 und

[4] Abstand des Bereiches zum Schwellwert des Detektors

[5] Zur Erinnerung: das Zwei-Block-Merkmal besteht aus zwei rechteckigen Regionen, die mit $+2$ und -2 gewichtet werden.

B_2 nennen, besteht. Damit sieht die Rechnung dafür wie folgt aus:

$$2 \sum_{b_{ij} \in B_1} \frac{b_{ij} - \mu}{\sigma} - 2 \sum_{b_{ij} \in B_2} \frac{b_{ij} - \mu}{\sigma}$$

Da die Anzahl der Pixel in B_1 und B_2 gleich ist, löst sich μ auf:

$$\frac{2}{\sigma} \left(\sum_{b_{ij} \in B_1} b_{ij} - \sum_{b_{ij} \in B_2} b_{ij} \right)$$

Die Berechnung für das Drei-Block-Merkmal sieht analog aus. Anstatt jeden Helligkeitswert zu normieren, reicht es also, den berechneten Merkmalswert durch die Standardabweichung des Bildausschnitts zu dividieren.

Für die Berechnung dieser Standardabweichung benötigt man jedoch die Pixelwerte zum Quadrat:

$$\sigma^2 = \frac{1}{N} \sum_{i,j} b_{ij}^2 - \mu^2$$

wobei N die Anzahl der aufsummierten Pixel im Bildausschnitt ist. Um die Quadratsumme einfach berechnen zu können, wird parallel zum normalen *Integral-Image* ein zweites aufgebaut, in dem die Pixelwerte zum Quadrat aufsummiert werden. Damit läßt sich σ effizient mit konstantem Aufwand für beliebig große Bildausschnitte berechnen.

Man wird von der Beleuchtung unabhängiger, da durch diese Rechnung der Kontrast eines Bildbereiches erhöht wird. Diese Kontrasterhöhung geschieht wegen der Standardabweichung abhängig davon, wie stark die Helligkeitsunterschiede im betreffenden Bildbereich sind. Die Kontrasterhöhung ist umso stärker, je geringer die Helligkeitsverteilung im Bildausschnitt ist.

Wir haben außerdem festgelegt, dass Bilder mit einer Standardabweichung[6] unter 13, 8 von der Detektion von vornherein ausgeschlossen werden, da diese praktisch einfarbig sind. Allerdings hatten wir trotzdem Mißklassifikationen in diesen Bildausschnitten, da nach der Kontrasterhöhung Muster sichtbar werden können.

Das Verfahren zur Kontrasterhöhung wenden wir während des Trainings auf die Trainingsdaten an und später beim Anwenden des Detektors auf die Bildfenster, die klassifiziert werden sollen.

Testergebnisse Getestet haben wir unseren Detektor mit der CMU Testmenge [22], die aus 130 Graustufenbildern mit 468 menschlichen Frontal-Gesichtern besteht. Leider erhalten wir darauf bei vier falsch als Gesicht erkannten Regionen nur eine moderate Erkennungsrate von 52%. Zuruckzuführen ist dies auf die Tatsache, dass einige Bilder in dieser Testmenge recht verrauscht oder unscharf sind. Die Trainingsmenge bestand jedoch zum großen Teil aus scharfen Aufnahmen mit der robotereigenen Kamera.

[6] bei einem Wertebereich von 0 bis 255 für die Pixelwerte

2.8 Fazit

In der praktischen Anwendung zeigt sich der Erfolg unseres Ansatzes. Boosting ist ein wichtiger Lernalgorithmus, der leicht zu implementieren ist und wenig Vorwissen braucht, um gute Resultate zu erzielen. Außerdem ist es durch den Einsatz der verwendeten Merkmale in Kombination mit dem Integral Image und der Kaskade möglich, den gelernten Klassifikator in verschiedenen Skalierungen schnell auf größere Bildbereiche anzuwenden.

Anhand der unterschiedlichen Trainingsbilder konnte auch gewährleistet werden, dass auch Brillen- und Bartträger erkannt werden. Um möglichst gute Ergebnisse zu erhalten, ist es deshalb für Boosting notwendig, eine große Trainingsmenge mit möglichst vielen Variationen zu haben. Doch auch eine Trainingsmenge von 10.000 Bildern kann nicht alle Eigenarten von Gesichtern und noch viel weniger von allen Nicht-Gesichter erfassen. So erschweren zum Beispiel verschiedene Beleuchtungssituationen die Umstände weiter. Eine Vergrößerung der Trainingsmenge ist heutzutage aber klar nicht praktikabel.

Trotz verschiedener Schwierigkeiten ist es uns jedoch gelungen, einen robusten und schnellen Gesichtsdetektor zu realisieren, der in verschiedenen Situationen gute Ergebnisse liefert.

(Markus Gärtner, Dinu Niessner, Viktor Peters, Andreas Vangerow)

3 Sprachlokalisation

Die Grundlagen des Projektes wurden in der Übung zur Vorlesung „Anwendungsorientierte Sprachverarbeitung“ im Sommersemester 2002 gelegt. Dort haben wir die theoretischen Grundlagen und die entsprechenden Algorithmen erarbeitet. Die Aufgabe bestand in der Lokalisation von Geräuschquellen mit zwei Mikrofonen - der übliche Ansatz mit Mikrofon-Arrays, bestehend aus äquidistanten Mikrofonpaaren, war nicht möglich, denn der Aufbau musste auf einem mobilen Roboter installiert werden.

In der Übung entstand das Programm *sploc*. Dieses Programm errechnet aus den Mikrofondaten eine Laufzeitverschiebung und den korrespondierenden Winkel. Es wurden drei verschiedene Algorithmen zur Berechnung der Laufzeitdifferenz implementiert. Am Ende der Übung konnten wir erste vorzeigbare Ergebnisse liefern.

Für das Projektseminar wurde das Programm erweitert. Auf der einen Seite wurden Schnittstellen geschaffen, die eine Kommunikation mit der Aufmerksamkeitssteuerung ermöglichen. Auf der anderen Seite sollten nun auch mehrere Geräuschquellen durch das Programm erkannt werden. Hierzu wurde eine sogenannte Peak-Detektion und parabolische Interpolation entwickelt, mit der verschiedene Quellen identifiziert werden können. Zur Optimierung der errechneten Ergebnisse werden Daten aus vorherigen Rechenschritten in das aktuelle Ergebnis miteingerechnet. Diese Neuerungen führen zu einer Verbesserung der Anwendung.

3.1 Projekt-Modell

Im Folgengenden wird Architektur der Lokalisation vorgestellt (siehe Abb.9). Sie gibt den Verlauf der Information und die jeweiligen Rechenschritte an, beginnend bei den Mikrofonen bis zum endgültig errechneten Winkel ϕ.

Die Aufnahme des Signals geschieht mit zwei Mikrofonen (Kap.3.2.1). Sie zeichnen die Umgebungsgeräusche auf. Die Geräusche aus dem Raum kommen leicht verzögert an einem Mikrofon an, da die Strecke zu beiden Mikrofonen unterschiedlich ist (gilt nur, falls die Geräuschquelle nicht *genau* in der Mitte zwischen den beiden Mikrofonen befindet). Das heißt, dass ein Signal zeitlich versetzt an den Mikrofonen ankommt.

Auf dieser Grundlage haben wir unser Programmstruktur entwickelt. Es wird ein Signalausschnitt von einem Mikrofon mit dem Signal aus dem anderen Mikrofon verglichen. Wir suchen ähnliche Signalstücke, deren zeitlicher Versatz den Winkel zu den Mikrofonen widergibt. Das kontinuierliche Signal wird diskretisiert. Aus dem linken (= linkes Mikrofon) und rechten (= rechtes Mikrofon) Kanal wird ein Stück (oder auch Stichprobe *engl.* Sample) herausgenommen. Im nächsten Schritt, der Berechnung der Laufzeitdifferenz (Kap. 3.2.2), wird auf dem Sample des linken Kanals ein zentrierter Ausschnitt (*engl.* Frame) extrahiert und mit dem rechten Sample verglichen. Aus diesem Grund schiebt man den Frame schrittweise über den rechten Sample und errechnet für jede Verschiebung einen Energiewert. Dadurch entsteht

eine Reihe von Energiewerten. In der Winkelberechnung (Kap. 3.2.2) werden die lokalen Maxima der Energiereihe gesucht (Peak-Erkennung). Sie stellen jeweils eine potentielle Geräuschquelle dar. Um die Fehler bei der Diskretisierung bei der Aufnahme des Signals auszugleichen, wird eine parabolische Interpolation durchgeführt. Aus der Position der nun gefunden Maxima können die Einfallswinkel der Geräuschquellen ϕ berechnet werden.

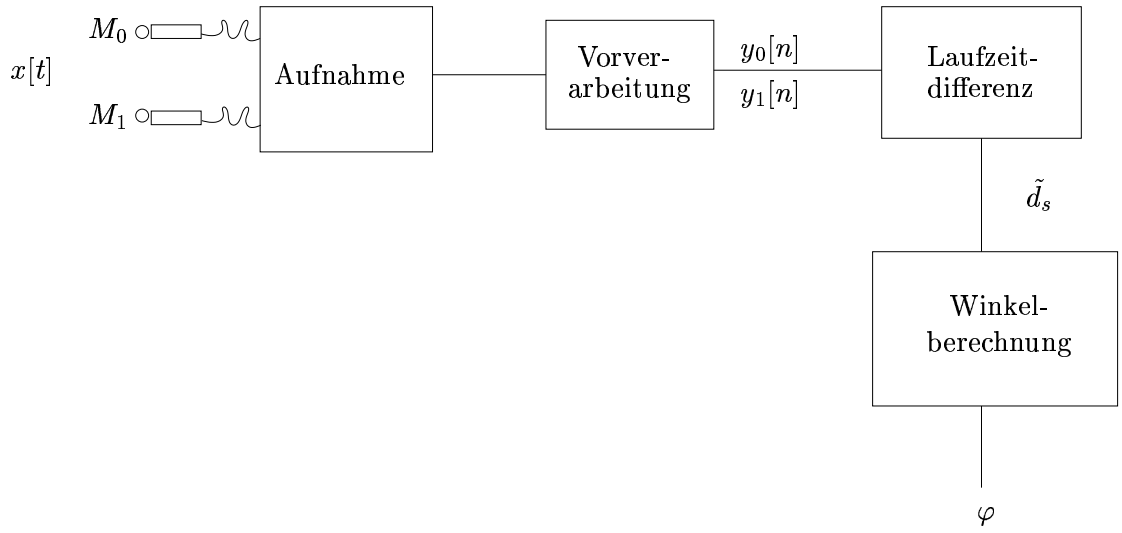

Abbildung 9: Modellaufbau

3.2 Geometrische Grundlagen

Nachdem wir nun die Grundlagen der Sprecherlokalisation erörtert haben, folgt in diesem Abschnitt ein detaillierter Überblick über die von uns eingesetzten Methoden und Prozeduren der Signalaufnahme.

3.2.1 Aufnahme des Signals

Wir verwenden für die Signalaufnahme das *raw*-Format, welches die beiden Kanäle alternierend kodiert; d.h. ein Byte linker Kanal, ein Byte rechter Kanal. Aus dem kontinuierlichen Signal der Mikrofone wird ein diskretes Signal $x[t] \rightarrow y_{0/1}[n]$ (Diskretisierung). Im ersten Verarbeitungsschritt wird daher aus dem *raw*-Format ein rechter und ein linker Kanal extrahiert. Aus einem der beiden Samples benötigen wir nun einen Ausschnitt (*Frame*). Die Länge des Frames setzt sich hierbei zusammen aus der gewünschten Framebreite und dem benötigten Überlapp $\tilde{d}_{res}$ (Kap. 3.2.2) um auch Geräusche zu erkennen, die sich 90° links oder rechts von den Mikrofonen befinden.

3.2.2 Berechnung der Laufzeitdifferenz

Zur Berechnung der Laufzeitdifferenz sind Überlegungen zum Signalweg unabdingbar, um zu verstehen, warum eine Laufzeitdifferenz entsteht und wie man sie berechnet. Die maximale Laufzeitdifferenz gibt dann Hinweise zur Auflösung und Messungenauigkeit.

Signalweg Zu Beginn ein kleiner Einblick in den Aufbau: Es wird häufig mit Mikrofonarrays gearbeitet. Mikrofone werden dabei paarweise in äquidistanten Abständen an einer Ebene befestigt. Dies ist hier nicht möglich, da zum einen die Rechenleistung zur Auswertung der großen Datenmengen nicht reicht und zudem die Möglichkeiten zum Anbringen der Mikrofone begrenzt sind. Auf dem mobilen Rechner laufen noch einige andere Prozesse, wie z.B. Gesichtserkennung, *Anchoring*, die auch ihre Rechenkapazität benötigen. Aus diesem Grund und im Unterschied zu den meisten anderen Ansätzen, soll hier die Berechnung mit nur zwei Mikrofonen erfolgen (siehe Abb. 10).

Der Aufbau ähnelt dem menschlichen Gehör. Analog zu den zwei Ohren gibt es zwei Mikrofone, die das Signal einer Geräuschquelle (z.B. eines Sprechers) empfangen. Da wir später an die Abmessungen des mobilen Roboters gebunden sind, rechnen wir in den nachfolgenden Beispielen mit einem realistischen Mikrofonabstand von 25 cm.

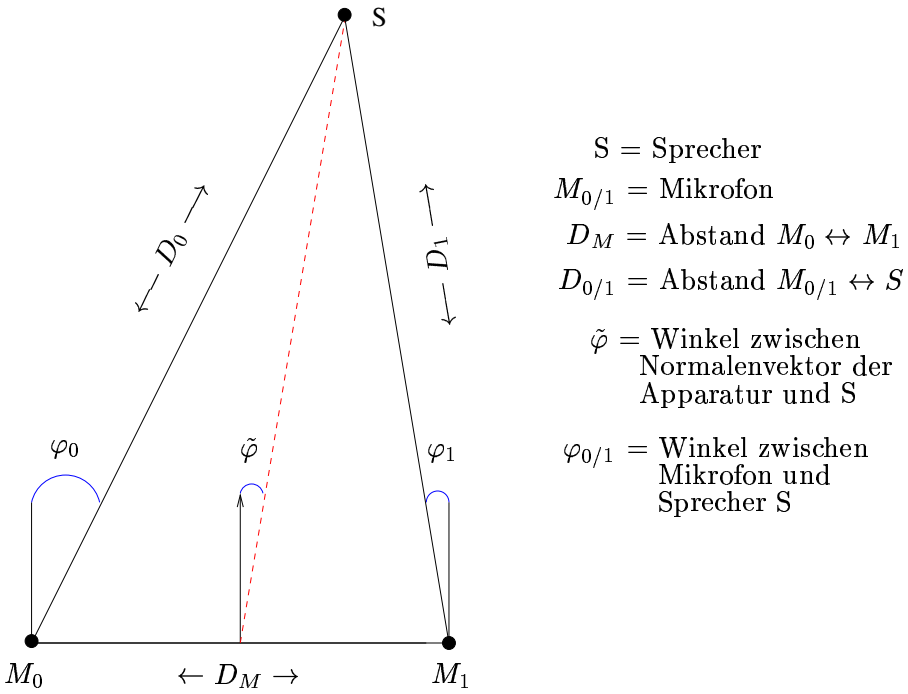

Abbildung 10: Prinzipskizze Signalweg: Unterschiede im Signalweg zu den einzelnen Mikrofonen

Laufzeitunterschied Um die Berechnungen weiter zu vereinfachen, wird folgende Annahme gemacht: Man geht davon aus, dass der Abstand zwischen den Mikrofonen sehr klein gegenüber dem Abstand zum Sprecher ist (siehe Abb. 11). Daher kann man näherungsweise davon ausgehen, dass das Signal an beiden Mikrofonen parallel einfällt. Dadurch ist auch der Einfallswinkel an beiden Mikrofonen gleich (siehe Gl. 1). Das Signal hat zu Mikrofon D_0 einen um $\tilde{D}$ verlängerten Weg (siehe Gl. 2). Dies bedeutet ein um $\tilde{d}$ verzögertes Eintreffen durch die längere Distanz zum Mikrofon. Damit und mit Hilfe der Schallgeschwindigkeit (a = 340 m/s) lässt sich der zeitliche Versatz $\tilde{d}$ berechnen (siehe Gl. 3). Mit dem Satz des Pythagoras ist es nun möglich, den Einfallswinkels $\tilde{\varphi}$ zu ermitteln (siehe Gl. 4); d.h. die Hauptaufgabe liegt in der Berechnung von $\tilde{d}$.

$$D_0, D_1 \gg D_M \quad \Rightarrow \quad \varphi_0 \simeq \varphi_1 \simeq \tilde{\varphi} \tag{1}$$

$$\tilde{D} \quad = \quad D_0 - D_1 \tag{2}$$

$$\tilde{d} \quad = \quad \frac{\tilde{D}}{a} \tag{3}$$

$$\tilde{\varphi} \quad = \quad \arcsin \frac{\tilde{D}}{D_M} \tag{4}$$

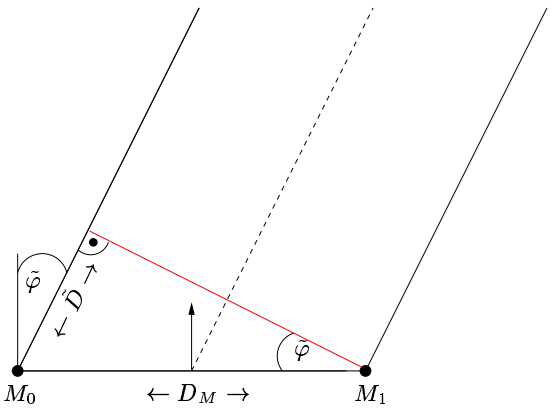

Abbildung 11: Prinzipskizze Laufzeitunterschied: Verlängerter Signalweg zu Mikrofon M_0 bei parallel einfallendem Signal

Maximale Laufzeitdifferenz Für jeden Mikrofonabstand gibt es eine maximale Laufzeitdifferenz; d.h. der maximale zeitlich Versatz eines Geräuschs an den beiden Mikrofonen (siehe Gl. 5). Dies ist der Fall, falls das Geräusch sich neben dem Mikrofon befindet (siehe Abb. 12) Bei dem jetzigen Mikrofonabstand von 25 cm muss das Signal in jede Richtung einen Überlapp von $\approx 0,8$ ms haben, damit auch eine

Quelle gefunden werden kann, die neben den Mikrofonen steht.

$$\tilde{d}_{max} = \frac{\tilde{D}_M}{a} = \frac{0.25}{340} = \frac{1}{1360} \approx 0,735ms \tag{5}$$

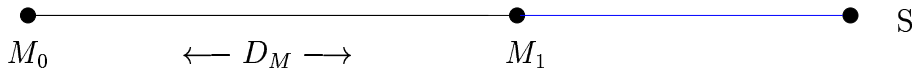

Abbildung 12: Extremfall: Sprecher steht neben den Mikrofonen; d.h. $\tilde{\varphi} = 90^o$; die Geräuschquelle S befindet sich neben einem Mikrofon $M_{0/1}$.

Auflösung: Gemäß dem Fall, dass sich das Geräusch im 90° Winkel zu den Mikrofonen befindet (siehe Abb. 12), kommt es zu einer zeitlichen Verzögerung an den Mikrofonen von 0, 8 ms. Das ist die Zeit, die das Geräusch braucht, um am zweiten, weiter entfernten, Mikrofon anzukommen. Aus dieser Zeit und der Sampling-Rate von 48 Samples pro Sekunde kann man berechnen, dass der dadurch entstandene maximale Versatz des Musters $\tilde{d}_{res}$ 36 Samples beträgt (siehe Gl. 6). Da man den Ort der Quelle vorher nicht kennt, muss immer mit einen maximalen Versatz gerechnet werden; d.h. 36 Samples müssen sich links und rechts vom aktuellen Sample befinden.

$$\begin{aligned} \tilde{d}_{res} &= \tilde{d}_{max} \cdot Samples/ms \\ \tilde{d}_{res} &\approx 35,294\text{Samples} \end{aligned} \tag{6}$$

Messgenauigkeit Die Genauigkeit der Messung hängt vom Winkel zwischen Person und Aufbau ab. Je größer dieser ist, desto ungenauer ist das Ergebnis. In Tabelle 1 ist der errechnete Winkel(°) für alle möglichen Laufzeitdifferenzen (S) sowie der Unterschied zwischen zwei benachbarten Ergebnissen (Werte ohne Einheit) aufgeführt.

Die Genauigkeit nimmt nach aussen ab. Man kann diese Ungenauigkeit grob in drei Stufen einteilen. In Abbildung 13 entspricht der Bereich von 0° bis 30° einer Messungenauigkeit von weniger als 2^o, der Bereich von 30° bis 60° einer Abweichung zwischen 2^o und 3^o und der Bereich ab 60° einer Abweichung von mehr als 3^o.

3.2.3 Bearbeitung der Samples

Grob betrachtet erfolgt die Berechnung des Laufzeitunterschieds immer nach dem selben Prinzip: Der Theorie folgend muss ein Signal-Muster, welches im linken Kanal vorkommt, auch ähnlich im rechten Kanal vorhanden sein, jedoch um ein paar Samples verschoben (siehe Abb. 14).

Daher nimmt man sich einfach aus dem linken Kanal ein zentriertes Fenster definierter Breite und vergleicht es mit einem ebenso breiten Ausschnitt aus dem

0 S	1 S	2 S	3 S	4 S	5 S	6 S	7 S	8 S	9 S
0^o	$1,6^o$	$3,2^o$	$4,9^o$	$6,5^o$	$8,1^o$	$9,8^o$	$11,4^o$	$13,1^o$	$14,7^o$
	1,6	1,6	1,7	1,6	1,6	1,6	1,7	1,6	1,7

10 S	11 S	12 S	13 S	14 S	15 S	16 S	17 S	18 S
$16,5^o$	$18,1^o$	$19,9^o$	$21,6^o$	$23,4^o$	$25,1^o$	$26,9^o$	$28,8^o$	$30,7^o$
	1,6	1,7	1,8	1,7	1,8	1,9	1,9	

19 S	20 S	21 S	22 S	23 S	24 S	25 S	26 S	27 S
$32,8^o$	$34,5^o$	$36,5^o$	$38,5^o$	$40,7^o$	$42,9^o$	$45,1^o$	$47,5^o$	$49,9^o$
	1,7	2,0	2,0	2,2	2,2	2,2	2,4	2,4

28 S	29 S	30 S	31 S	32 S	33 S	34 S	35 S
$52,5^o$	$55,3^o$	$58,2^o$	$61,4^o$	$65,5^o$	$69,2^o$	$74,4^o$	$82,6^o$
	2,8	2,9	3,2	4,1	3,7	5,2	8,2

Tabelle 1: Messergebnisse und ihre Differenzen; obere *Zeile: S gibt Laufzeitunterschied in Samples wieder; Winkel darunter ergibt den errechneten Winkel; untere (farbige)Zahl ist die Differenz zwischen den Winkeln ⇒ Messungenauigkeit*

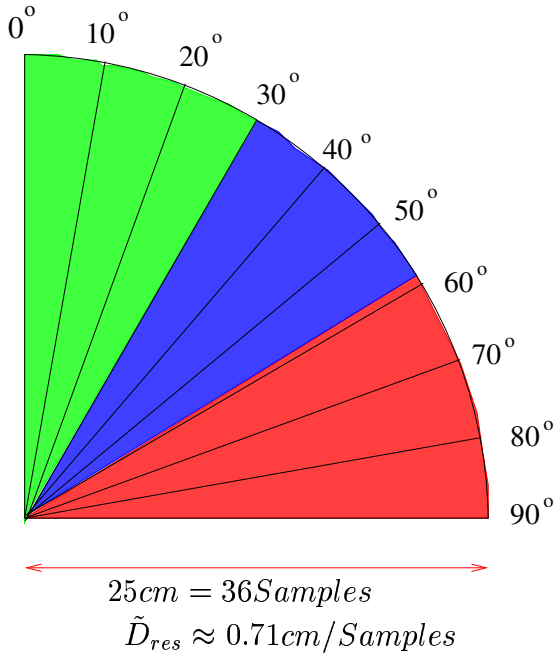

Abbildung 13: Messgenauigkeit: Grobe Einteilung in Bereiche mit Abweichung unter 2^o, zwischen 2^o und 3^o und über 3^o

anderen Kanal, welcher aber um einen paar Samples verschoben wurde (siehe Abb. 15). Das Ergebnis dieses Vergleichs ist ein Energiewert der beiden Ausschnitte.

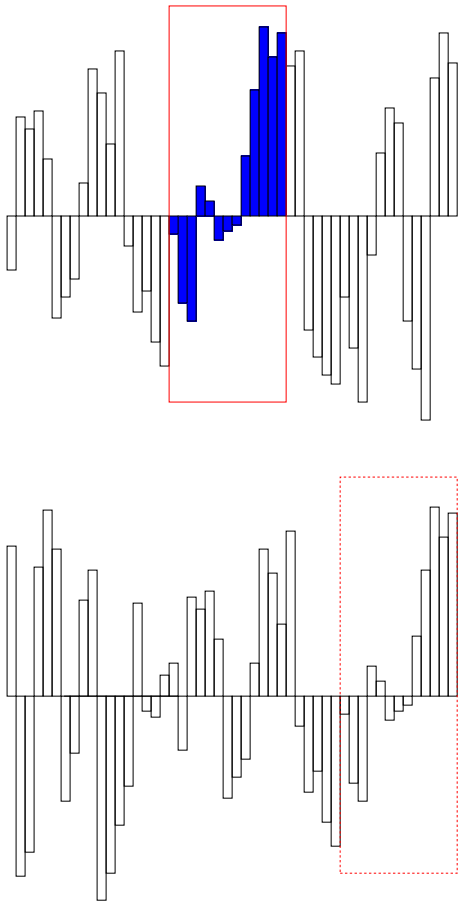

Abbildung 14: Verschobenes Signal-Muster. Oben: *linker Kanal;* Unten: *rechter Kanal; Die Werte des zu vergleichenden Samples sind markiert.*

Dies geschieht mit allen möglichen Verschiebungen von $-\tilde{d}_{res}$ bis $+\tilde{d}_{res}$. Pro Verschiebungsschritt wird mit der *Cross Power Spectrum Phase* (Kap. 3.3) ein Energiewert berechnet. Diese Werte sind die Grundlage für weitere Berechnungen, um die Laufzeitdifferenz zu ermitteln.

An den Mikrofonen können mehrere Geräuschquellen gleichzeitig auftreten und sich überlagern. Aus diesem Grund suchen wir im rechten Kanal ähnliche Muster, um jede einzelne Quelle zu lokalisieren. Um nun mehrere Geräuschquellen zu finden, müssen die lokalen Maxima der Energiewerte mit Hilfe einer Peak-Erkennung gefunden und Störgeräusche unterdrückt werden. Im Folgenden stellen wir unser Verfahren vor. Durch eine Gewichtung der Energiewerte können wir kurze Störgeräusche verkleinern. Um die Peak-Erkennung effizient durchzuführen, grenzen wir die Maxima mit einem Schwellwert (Varianz) ein. Die Ungenauigkeiten durch die Diskretisierung bei der Aufnahme werden durch eine parabolische Interpolation geglättet.

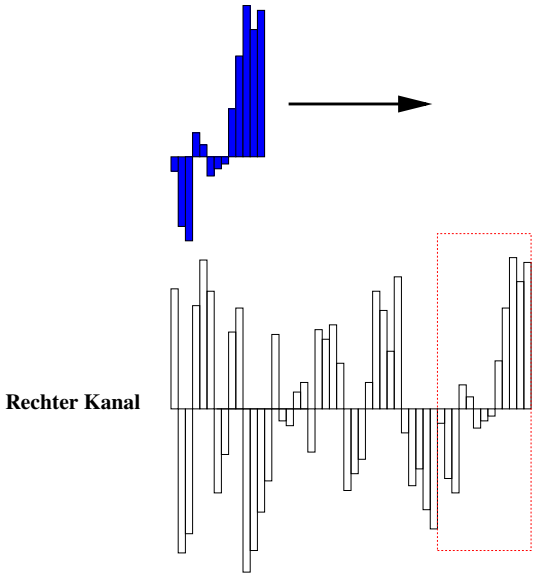

Abbildung 15: Vergleich des Referenzmusters mit dem Ausschnitt

Energiegewichtung Durch die Energiegewichtung erfolgt eine Verstärkung der gefundenen Maxima und zufällig hohe Werte werden gedrückt. Durch kurze und plötzliche Geräusche kann das Ergebnis verfälscht werden. Dadurch treten bei der Energieberechnung einzelne (Höchst-)Werte in den Vordergrund, obwohl keine Geräuschquelle vorhanden ist. Aus diesem Grund werden eine gewisse Anzahl von Vorgängerwerten (H) gewichtet in den aktuellen Wert miteingerechnet (siehe Gl. 7). Sollte also ein zufälliger hoher Wert auftreten und z.B. die vier vorangegangenen Werte sehr klein sein, so wird auch der aktuell hohe Werte durch die gewichteten Vorgänger verkleinert. So können kurze und intensive Störgeräusche unterdrückt werden.

$$E(t) = \frac{\sum_{h=0}^{H} \left(\left(\frac{2}{3} \right)^2 \cdot E(t-h) \right)}{\sum_{h=0}^{H} \left(\frac{2}{3} \right)^2} \tag{7}$$

Peak-Erkennung Das System ist insoweit weiterentwickelt worden, dass mehrere Quellen erkannt werden können. Aus diesem Grund gibt es entsprechend viele Maxima. Jeder der sogenannten Peaks stellt eine potentiell gefundene Geräuschquelle dar. Um die Anzahl einzugrenzen, haben wir einen Schwellwert eingefügt, wobei N die Anzahl der Energiewerte wiedergibt (siehe Gl. 8).

$$Var = \sqrt{\frac{\sum_{h=0}^{H} E(t)}{N}} \tag{8}$$

Dieser Schwellwert ist in Abbildung 16 durch die gestrichelte Linie (a) dargestellt ist. Falls mehrere Werte nebeneinander über dem Schwellwert liegen, wird nur der Maximalwert gesucht (Max). Die Maxima, die über dem Schwellwert liegen, werden weiter bei der Berechnung berücksichtigt. Jeder Maximalwert ist ein gefundenes Objekt. Von ihm aus wird nun das lokale Minimum rechts und links bestimmt. Die jeweiligen Minima links und rechts (Min_L, Min_R) werden für die parabolische Interpolation gebraucht.

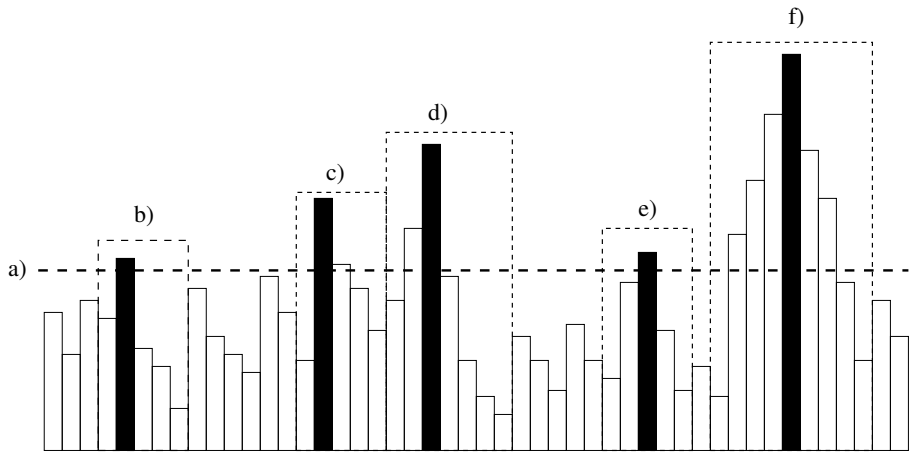

Abbildung 16: Die errechneten Energiewerte E(t). Die schwarzen Säulen sind die gefundenen Maxima. Die Bereiche um die Maxima (b - f) geben die gefundenen Minima links und rechts wieder. Der Schwellwert (a) ist ebenfalls aufgetragen.

Parabolische Interpolation Bei der Vorverarbeitung und weiteren Rechenschritten kann es zu Ungenauigkeiten durch die Diskretisierung kommen. Aus diesem Grund sind die Maxima leicht verschoben. Um dieses auszugleichen führen wir eine parabolische Interpolation durch, bei der eine Parabel durch die Minima und das Maximum gelegt wird. Das Maximum der Parabel unterscheidet sich von dem gefundenen Maximawert (schwarze Säule, s. Abb. 16). Je nach Lage der Minima ist der Maximalwert der Parabel links oder rechts vom gefunden Wert (siehe Gl. 9). Diese Verschiebung des Maximums lässt aus der Parabelgleichung ableiten.

$$Verschiebung = \frac{Min_L - Min_R}{2 \cdot (Min_L - (2 \cdot Max) + Min_R)} \tag{9}$$

Zuordnung und Bereinigung durch lineare Vorhersage Ein großes Problem bei der Berechnung sind Stör- bzw. kurze Hintergrundgeräusche, die nicht durch Energiegewichtung herausgefiltert wurden. Sie führen zu plötzlichen und einmaligen Maxima. Im Energiespektrum sind dies plötzlich auftretende Spitzen(-werte). Um eine aufwendige Vorverarbeitung (durch verschiedene Filter) zu vermeiden, werden diese Ausreisser erst nach der Berechnung eliminiert. Alle errechneten Winkel werden in einem Array gespeichert. Um herauszufinden, ob es sich um ein einmaliges Störgeräusch handelt, arbeiten wir nur mit Werten, die zwei Vorgänger haben; d.h. wir durchsuchen das Array der letzten beiden Rechenschritte. In ihnen sind die letzten Winkelwerte gespeichert. Taucht der Winkel nicht in beiden Buffern auf, so handelt es sich um ein Störgeräusch.

Bei allen gefundenen Geräuschquellen zum Zeitpunkt t werden Vorgänger gesucht, die sich in einem Toleranzbereich von 3 Grad bewegen. Bei stationären Quellen ist das kein Problem. Es wird schwieriger, wenn die Quelle sich vor dem Roboter bewegt und das mit mehr als 3 Grad pro Rechenschritt. Der Algorithmus würde den Winkel verwerfen. Aus diesem Grund haben wir die Berechnung erweitert (siehe Abb. 17): Im ersten Zeitschritt t_0 ist nichts gefunden worden. Im nächsten Zeitschritt t_1 wurde der Winkel α gefunden. Er ist unser erster gefundener Winkel und hat daher keinen Vorgängerwinkel V_x. Als seinen Nachfolgerwinkel N_α nehmen wir wiederrum α, da wir keinen weiteren Informationen haben.

Im dritten Zeitschritt t_2 finden wir den Winkel β. Wir suchen nun nach einem eventuellen Vorgänger. Dieser wird ermittelt, in dem man überprüft, ob β im Bereich von $\pm 3°$ des Nachfolgewertes N_α von α liegt. In der Abbildung 17 ist dies durch $\alpha + \triangle$ angegeben, wobei $\triangle$ den Bereich von $\pm 3°$ abdeckt. Diese Vorgänger sind deshalb wichtig, da wir einen gefundenen Winkel erst dann einer Geräuschquelle zuschreiben, falls der Winkel zwei Vorgängerwerte hat. Andernfalls ist es ein kurzes Störgeräusch und wird nicht berücksichtigt.

Nun schätzen wir den wahrscheinlichen Nachfolgewinkel von β. Bei diesem Wert N_β gehen wir davon aus, dass sich die Geräuschquelle mit einer gewissen Geschwindigkeit vor dem Roboter bewegt. Aus diesem Grund addieren wir zum aktuellen Winkel β die Differenz zum Vorgängerwinkel α hinzu. Man kann diesen Term als Steigung interpretieren, die uns zeigt, wohin sich die Geräuschquelle bewegt. Dieser Nachfolgewinkel N_β ist im nächsten Zeitschritt wieder die Grundlage, um zu bestimmen, ob der dort Gefundene und dieser zusammenhängen; d.h. vielleicht einer Geräuschquelle zugeordnet werden können.

Im Zeitschritt t_4 werden zwei Winkel gefunden. Der linke Winkel θ im Flussdiagramm ist neu dazugekommen. In diesem Beispiel gäbe es keinen Vorgängerwinkel für θ. Er liegt nicht im Bereich ($\pm 3°$) von N_γ. Es wäre neuer Winkel, der erst dann angezeigt wird, falls in den nächsten zwei Schritten ein ähnliche Winkel ermittelt werden. Eine gefundene Geräuschquelle stellt der Winkel γ dar. Er hat zwei Vorgängerwinkel α und β, die sich in seiner engeren Umgebung befinden. Erst jetzt sprechen wir von einer erkannten Geräuschquelle mit zugehörigen Winkeln.

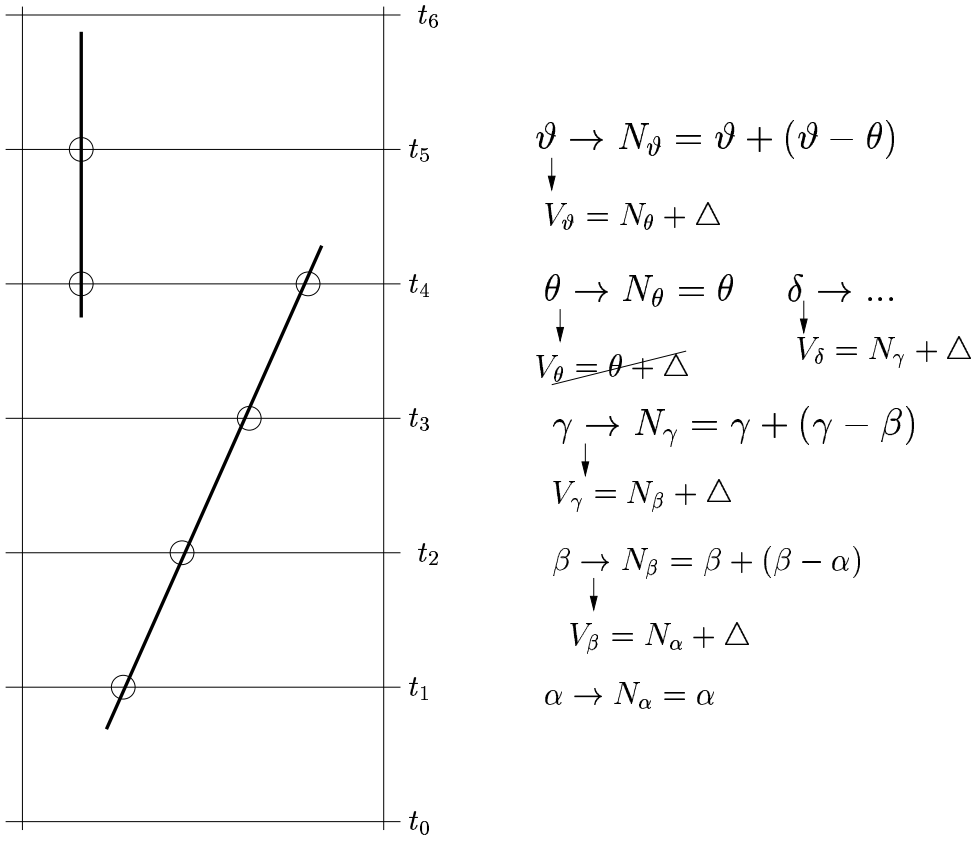

*Abbildung 17: Suchverfahren;*links*: Wasserfalldiagramm mit zwei bewegenden Geräuschquellen. Die Striche im Diagramm geben die Richtung der bewegenden Quellen an. Die Kreise geben den gefundenen Winkel wieder.* rechts*: Rechenschritte*

3.3 Verarbeitung im Detail

Es gibt verschiedene Möglichkeiten, die Energie zu berechnen. In Frage kommen unter anderem das Quadratische Mittel und die Normalized Cross-Correlation. Wir haben uns jedoch für einen weiteren Ansatz, der Cross Power Spectrum Phase, entschieden, da dieses Verfahren laut einer Untersuchung [13] effektiver ist.

Cross Power Spectrum Phase (CSP)

Bei der Cross Power Spectrum Phase Berechnung wird eine (diskrete Fast-) Fouriertransformation (FFT) auf den beiden zu vergleichenden Frames durchgeführt; d.h. das feste linke Kanalmuster $\hat{Y}_0[t]$ und der konjugiert komplexe Anteil des fouriertransformierten Frames $\hat{Y}_1^*[t+\delta]$ aus dem rechten Sample werden multipliziert (siehe Gl. 10). Das Frame wird über das ganze Sample verschoben. Zusätzlich wird der errechnete Wert normiert. Im nächsten Schritt wird die Kreuz-Korrelation errechnet ([13], [5]) . Das Ergebnis wird wiederum in den Signalraum zurücktrans-

formiert (FFT^{-1}). Man muß dann nur noch den Phasenanteil vergleichen, da die aufgenommenen Signale lediglich aus reellen Anteilen bestehen und uns nach der Rücktransformation lediglich der Phasenversatz interessiert, nicht aber die Amplitude.

$$\arg \max_{\substack{\delta \\ -\tilde{d}_{max} < \delta < \tilde{d}_{max}}} FFT^{-1} \left[\frac{\sum_{t=1}^{N} (\hat{Y}_0[t] \cdot \hat{Y}_1^*[t+\delta])}{|\hat{Y}_0[n]| \cdot |\hat{Y}_1[n]|} \right] \tag{10}$$

$$\hat{Y} = FFT(\hat{y})$$

Diese Verfahren scheinen im ersten Moment aufwendig zu sein, doch lassen sich durch folgende Überlegungen zeitintensive Schritte sparen:

1. Der linke Kanal als Muster bleibt fest.

   Da wir ja einen Kanal-Frame invariabel lassen, müssen wir auch nur einmal die Fast Fourier Transformation (FFT) berechnen. Dies erspart ca. 25% der Rechenzeit

2. Inverse Fast Fourier Transformation (FFT^{-1}) ist nicht nötig

   Nach der Rücktransformation ist nur der Phasenanteil für den Vergleich wichtig. Daher ist es nicht erforderlich, das Ergebnis wieder in den Signalraum zu transformieren, sondern es genügt lediglich die Summe über die einzelnen Werte zu bilden, da dies genau dem Phasenanteil entspricht. Dies erspart noch einmal 25% der Rechenzeit

3. Niedrige und sehr hohe Frequenzen werden einfach gefiltert, da man dafür nur die unteren Werte bei der die Fouriertransformation löschen muss. Mit diesem Verfahren können Diese Störgeräusche wie z.B. Brummen eines Rechners oder das Quietschen einer Türe herausgefiltert werden.

4. Symmetrieeigenschaften

   Die Fouriertransformierte eines (reellen) Audio-Signals hat folgende Eigenschaften:

   - Der Imaginär-Anteil ist punktsymmetrisch

     Dies bedeutet für uns, daß er sich in der Summe (Gl. 10) aufhebt und daher nicht berechnet werden muß!

- Der Real-Anteil ist achsensymmetrisch.

  Es genügt also, nur die Hälfte der Werte zu berechnen und das Ergebnis einfach zu verdoppeln. Dies verringert die Rechenzeit noch einmal um 15 - 20%.

Dies verringert die Rechenzeit noch einmal um 15 - 20%. Alles in allem sollte sich somit mehr als 70% der Rechenarbeit einsparen lassen, was den anfangs komplizierten Algorithmus als beste Wahl für unsere Lokalisierungsaufgabe qualifiziert.

3.4 Fazit

Die Sprachlokalisation hat sich in wichtigen Punkten verbessert. Es stellte sich im Laufe des Projektes heraus, dass die Energieberechnung auf einem Sample durch die Cross Power Spectrum Phase (CSP) ein sinnvoller Algorithmus ist, um einen schnellen und guten Vergleich der Kanäle zu sichern. Die Möglichkeit, mehrere Personen zu lokalisieren, ist wohl die wichtigste Neuerung. Sie ermöglicht dem Gesamtsystem des Roboters, mehrere potentielle Kommunikationspartner zu finden. Um eine geringe Fehlerquote (Störgeräusche) zu erhalten, ist die lineare Vorhersage und Zuordnung ein großer Fortschritt. In Detailverbesserungen (z.B. parabolische Interpolation) ist es uns gelungen, die Meßgenauigkeit zu erhöhen. Mit den beiden Mikrofonen ist auch eine Spracherkennung möglich. Dies wäre ein weiterer Schritt zu einer Mensch-Maschine-Interaktion. Der Roboter könnte konkrete Aufträge vom Menschen erhalten, die während einer Kommunikation mit dem Roboter geäussert worden sind.
(Bernd Focke, Torsten Vollmann)

4 Multi-Modal-Anchoring bei mobilen Robotern

Die Gruppe AmSeL (AufmerksamkeitsSteuerungsLeute) befasste sich mit der Integration der in Kapitel 2 und 3 vorgestellten Module zur Gesichts- und Stimmdetektion. Außerdem wurde die Integration eines noch nicht implementierten Moduls zur Pullovererkennung vorbereitet.

Zudem wurde eine Aufmerksamkeitssteuerung implementiert. Dies bedeutet, dass der Roboter erkennen sollte, ob jemand in seiner Umgebung mit ihm kommunizieren möchte. Um anzuzeigen, dass der Kommunikationswunsch der Person vor dem Roboter detektiert wurde, dreht der Roboter die auf ihm befindliche Kamera in Richtung der Person. Die Robustheit der Sensordatenauswertung ist eine wichtige Voraussetzung für eine Interaktion zwischen Mensch und Roboter (*engl.* Human-Robot-Interaction, kurz HRI). Je zuverlässiger die Auswertung der Sensordaten ist, desto besser lassen sich mobile Roboter in nicht speziell für sie gestaltete Umgebungen z.B. einem Büro einsetzen. Außerdem sollen die Sensordaten des Interagierenden ohne spezielle Ausrüstung gewonnen werden, z.B. sollte ihre Auswertung aussagekräftig genug sein, damit eine Person als Kommunikationspartner für den Roboter erkannt wird.

Im Folgenden wollen wir den Lösungsansatz vorstellen, der für dieses Problem im Projektseminar erweitert wurde. Zuerst betrachten wir hierfür schon bestehende Robotersysteme. Dann erklären wir den theoretischen Lösungsansatz, anschließend die Ausgangsrealisierung vor Beginn des Projektseminars und erläutern schließlich, was erweitert wurde.

4.1 SIG und Robita

SIG (Symbiotic Intelligence Group) und Robita sind zwei Entwicklungen auf dem Gebiet der HRI [9, 11]. Bei dem Entwurf der beiden Roboter wurde zunächst untersucht, wie eine normale Kommunikation zwischen Menschen stattfindet. Typisch ist z.B., dass Kommunikationspartner von Angesicht zu Angesicht miteinander interagieren. D.h. wenn mehr als ein Kommunikationspartner vorhanden ist, wird dem Sprecher üblicherweise der Kopf zugewandt. Auch die Äußerung von Sprache kann eine Einleitung einer Kommunikation sein. Aus diesem Grund verfügen beide Roboter über visuelle und auditive Sensoren (Kamera und Mikrofone). Bei wechselnder Umgebung muss das System zwischen menschlicher Sprache und anderen nichtmenschlichen Schallquellen unterscheiden können. Auch die wechselnden Lichtverhältnisse müssen berücksichtigt werden. Sie können die Gesichtserkennung durch Schwankungen in der Beleuchtung und Schattenwurf stark beeinflussen.

SIG ist ein Torsohumanoid, der an der Universität in Kyoto entwickelt wurde. Er wird als Rezeptionsroboter eingesetzt. Damit befindet er sich in einer festen Umgebung. Der Raum ist gleichmäßig ausgeleuchtet und enthält keine zusätzlichen Schallquellen. SIG soll in diesem Raum Menschen erkennen, sie mit Namen ansprechen und sie auffordern einen Nebenraum zu betreten. Am Anfang des Szenarios hat

SIG nicht immer die Position, dass er den Kommunikationspartner sofort erkennt. Wenn der Eintretende zu sprechen beginnt, wendet SIG ihm sein „Gesicht“ bzw. die dahinter versteckten Sensoren zu, so dass er Angesicht zu Angesicht mit dem Eintretenden spricht.

Robita wurde als Moderator für die Unterhaltung über Fachwissen an der Waseda Universität in Tokyo konzipiert. Es ist ihm möglich, gezielt an ihn gestellte Anfragen zu erkennen oder Falschaussagen eines Kommunikationspartners zu korrigieren. Dies wurde an einer Konversation über Baseball erprobt. Um an der Kommunikation teilzuhaben, verfügt Robita über die „Kenntnis“ einiger Schlüsselwörter, die in Zusammenhang mit Baseball stehen. Die Informationen über Baseballspiele, wie Spielergebnis, Werfer usw., werden aus dem Internet bezogen. Im Gegensatz zu SIG wird er mit mehr als einem Kommunikationspartner konfrontiert. Dafür muss er unterscheiden können, wann er angesprochen wird. Die Entscheidung wird nach folgenden Regeln getroffen:

1. Wenn der augenblickliche Sprecher den nächsten Sprecher aussucht, so hat der Ausgesuchte das Recht zu sprechen.

2. Falls der augenblickliche Sprecher keinen Nachfolger aussucht, so kann jeder außer dem augenblicklichen Sprecher das Wort erhalten.

3. Wenn der augenblickliche Sprecher nicht aktiv jemanden auswählt und niemand anderes von sich aus anfängt zu sprechen, so kann der augenblickliche Sprecher fortfahren.

Das Konzept, dass in unserem Fall verwandt haben, nennt sich *Multi-Modal-Anchoring* [4, 6] (siehe Kap. 4.3). Unser Roboter soll mit Hilfe dieser Methodik Personen zunächst einmal finden. Ziel ist es anschließend, dass der Roboter erkennen kann, ob jemand mit ihm kommunizieren möchte. Dieses wird genauer in Abschnitt 4.5 beschrieben.

4.2 Anchoring

Für eine erfolgreiche Mensch-Maschine-Interaktion besteht die erste Aufgabe des Roboters darin, einen Menschen in seiner Umgebung als solchen zu erkennen. Ein weitergehender Schritt ist, ihn als potentiellen Kommunikationspartner in der Umgebung auszumachen und mit Personenobjekten, die einen Menschen auf der Programmebene abbilden, zu assoziieren. Hierzu bedienen wir uns der Methode des *Multi-Modal-Anchoring* [4, 6] (siehe Kap. 4.3). Im Folgenden gehen wir zuächst einmal auf das klassische *Anchoring* ein. Eine Definition des klassischen *Anchoring* stammt von Coradeschi und Saffiotti [1]. Sie beschreiben die grundlegende Idee des *Anchoring* wie folgt:

> Anchoring is the problem of creating and maintaining in time the correspondence between symbols and sensor data that refer to the same physical object.

Anchoring setzt ein Symbolsystem, in dem die interne Repräsentation eines Objekts gespeichert ist, und ein perzeptuelles System, das die Sensordaten enthält, voraus. Um sicherzustellen, dass das Symbol das gleiche Objekt bezeichnet wie durch den Sensor wahrgenommen, gibt es sog. *Anchor*. Ein *Anchor* besteht zu jedem Zeitpunkt aus drei Komponenten: Einem Symbol, einem Perzept und einer Signatur. Coradeschi und Saffiotti haben diese Komponenten wie folgt definiert [1]:

- Ein Symbolsystem Σ besteht aus einer Menge von individuellen Symbolen $\chi = \{x_1, x_2, ... \}$, die Variablen und Konstanten darstellen sowie einer Menge von Prädikatsymbolen $\rho = \{p_1, p_2, ... \}$.
- Ein perzeptuelles System Ξ besteht aus einer Menge von Perzepten $\Pi = \{\pi_1, \pi_2, ... \}$ und einer Menge von Attributen $\Phi = \{\phi_1, \phi_2, ... \}$. Ein Perzept ist eine strukturierte Sammlung von Messwerten, die von einem Objekt stammen. Ein Attribut ϕ_i stellt eine messbare Eigenschaft des Perzeptes dar mit Werten aus den Domänen D_i, wobei $\mathrm{D} = \bigcup D_i$ sei.
- Eine Signatur $\gamma : \Phi \rightarrow \mathrm{D}$ ist eine Funktion von Attributen zu den Attributwerten. $\Gamma = (\Phi \rightarrow \mathrm{D})$ bezeichnet die Menge aller Signaturen γ. Die Signatur ordnet zu jedem Zeitpunkt t den Attributen entsprechende Werte zu, die aus Sensordaten gewonnen wurden.
- Die *Predicate-Grounding-Relation* $\mathrm{g} \subseteq \rho \times \Phi \times \mathrm{D}$ beschreibt die Verbindung von Prädikaten des Symbolsysems mit Werten messbarer Attribute des perzeptuellen Systems. Existiert eine Übereinstimmung zwischen Prädikaten und den Werten der messbaren Attributen, so wird der Status des zugehörigen *Anchors* als *grounded* bezeichnet, andernfalls als *ungrounded*. *Grounded* bedeutet, es kann ein Perzept durch einen *Anchor* mit einem Symbol verbunden werden, und es wird eine aktualisierte Signatur gebildet , *ungrounded* bedeutet, es gibt keine Eingangsdaten, die zugeordnet werden können und die bestehende Signatur wird beibehalten.

Hieraus ergibt sich folgende formale Definition für einen *Anchor*: Ein *Anchor* α ist eine partielle Funktion zum Zeitpunkt $\mathrm{t} \rightarrow \chi \times \Pi \times \Gamma$.

Weniger formell bedeutet dies: Ein *Anchor* besteht zu jedem Zeitpunkt aus einem Symbol, einem Perzept und einer Signatur. Das Symbol ist zur internen Benennung des Objektes vorgesehen, während ein Perzept durch das perzeptuelle System aus den eingehenden Sensordaten errechnet wird, dass das Objekt beobachtet. Die Signatur dient der Vorhersage von im nächsten Zeitschritt beobachtbaren Eigenschaften des Objektes, wie z.B. der Größe von hautfarbenen Bereichen (siehe Abb. 18). Es erfolgt also durch die Signatur eine Zuordnung von konkreten Werten des Sensors zu den Eigenschaften. Hier wäre z.B. noch die Abbildung der RGB-Werte auf das Attribut Farbe zu nennen.

Anchoring nach Coradeshi und Saffiotti [1] findet nur zwischen einem Sensor und einer internen Repräsentation statt. Daraus ergibt sich ein Problem, wenn ein

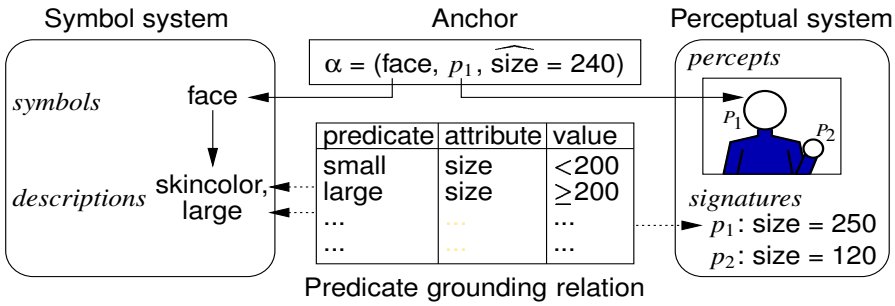

Abbildung 18: Anchoring *am Beispiel für Gesichter : Der* Anchor α *enthält zu jedem Zeitpunkt ein Symbol, ein Perzept und eine Signatur. Ein* Anchor *stellt eine Verbindung her zwischen einem Sensor bzw. der durch ihn eingehenden Daten (Perzept) und einer internen Repräsentation im Symbolsystem.*

Sensor nicht ausreicht, bestimmte Objekte zu verfolgen, z.B. komplexe Objekte, wie einen Menschen. In unserem Fall jedoch wurden zur Kommunikationspartnererkennung zwei Sensoren eingesetzt: Ein Laserscanner zum Finden von Beinpaaren und eine Kamera zur Aufnahme von Bildern, in denen eine Gesichtserkennung basierend auf der Detektion hautfarbener Bereiche durchgeführt wurde. Um die Daten mehrerer Sensoren zu kombinieren, wurde das *Anchoring* auf zwei Ebenen erweitert. Auf der unteren Ebene findet, wie im klassischen *Anchoring*, eine Verbindung zwischen einem Perzept und einem Symbol statt [6]. Dies bedeutet für unser Szenario, dass jedem reellen Beinpaar ein Symbol BEIN zugeordnet wird. Ebenso wird einem Gesichtsperzept ein Symbol FACE zugeordnet. Auf der neuen übergeordneten Ebene fasst man die einzelnen *Anchor* der unteren Ebene (*Component-Anchor*), die jeweils zu einer Person gehören, zu einem übergeordneten *Composite-Anchor* zusammen. Dieser wird in unserem Fall auch als *Person-Anchor* bezeichnet, da wir ihn für das *Anchoring* von Personen verwenden (siehe Abb. 19). Er verwaltet die einzelnen *Component-Anchor*. Die *Component-Anchor* verbinden also die Perzepte für die einzelnen Bereiche eines Menschen, die detektiert werden, mit den dazugehörigen Symbolen. In unserem Fall sind diese Bereiche das Gesicht und die Beine. Der *Composite-Anchor* ist eine Zusammenstellung aus diesen Teilen. Hierbei ist zu beachten, dass die Daten der *Component-Anchor* asynchron an den übergeordneten *Composite-Anchor* geliefert werden, weil die einzelnen Sensoren unterschiedliche Aufnahmeraten haben. Zu dem dauert die Berechnung auf den Eingangsdaten unterschiedlich lange. Die Zusammenstellung findet im *Anchor-Fusion-Module* statt. Auch die *Predicate-Grounding-Relation* (siehe Abb. 18) existiert für jeden *Component-Anchor* und für den übergeordneten *Composite-Anchor*. Sie stellt die Verbindung zwischen dem perzeptuellen und dem Symbolsystem her.

Das Auffinden und die Verfolgung einer der Person über die Zeit geschieht mit Hilfe von drei Zuständen der einzelnen *Component-Anchor*: *Find, Track* und *Reac-*

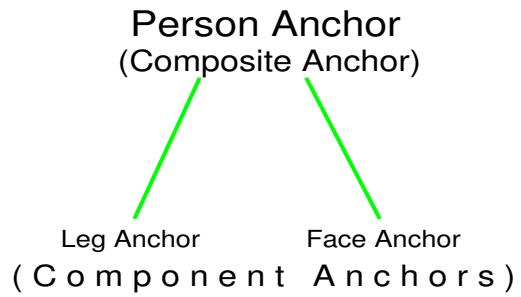

Abbildung 19: Ausgangssituation: Unter dem Composite-Anchor *werden die verschiedenen* Component-Anchor *zusammengefasst. Es gibt jeweils einen der* Component-Anchor *für Beine und Gesichter.*

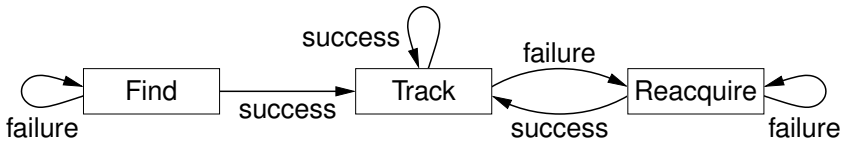

Abbildung 20: Ein Anchor *kann sich in den folgenden Modi befinden:* Find, Track *und* Reacquire. *Dies kann in einem endlichen Automaten dargestellt werden. Die Zustände deuten Unterschiede im Suchverhalten über die Zeit an.*

quire. Zunächst einmal befindet sich ein *Anchor* im *Find*-Modus. Hier werden Perzepte gesucht. Sobald es passende Eingangsdaten gibt, wird das Perzept mit dem zugehörigen Symbol verbunden. D.h. es wird ein *Anchor* etabliert und als *grounded* definiert. Anschließend geht der *Anchor* vom *Find*- in den *Track*-Modus. In diesem Zustand werden die *Anchor*-Daten mit jedem Zeitschritt erneuert. Außerdem wird die vorhergesagte Signatur mit den eingehenden Daten verglichen, um sicherzustellen, dass die Perzepte den bereits bestehenden Symbolen richtig zugeordnet werden. Der *Reacquire*-Modus wird angenommen, wenn keine passenden Eingangsdaten mehr gefunden werden. Gleichzeitig nimmt der *Anchor* den Status *ungrounded* an. Über einen gewissen Zeitraum wird versucht, anhand der neuen Signatur an der abgeschätzten Stelle wieder passende Eingangsdaten, die zu einer Person gehören zu finden. Wenn diese Daten gefunden werden geht der *Anchor* wieder in den *Track*-Modus. Werden nach einem bestimmten Zeitraum keine passenden Daten gefunden, erhält der alte Anchor den Status *lost* und wird gelöscht. Anschließend wird von neuem eine Person im *Find*-Modus gesucht (siehe Abb. 20).

4.3 Multi-Modal-Anchoring

Wie schon angedeutet, gab es anfänglich eine Realisierung des *Anchorings* basierend auf zwei *Component-Anchor*, *Face* und *Leg*.

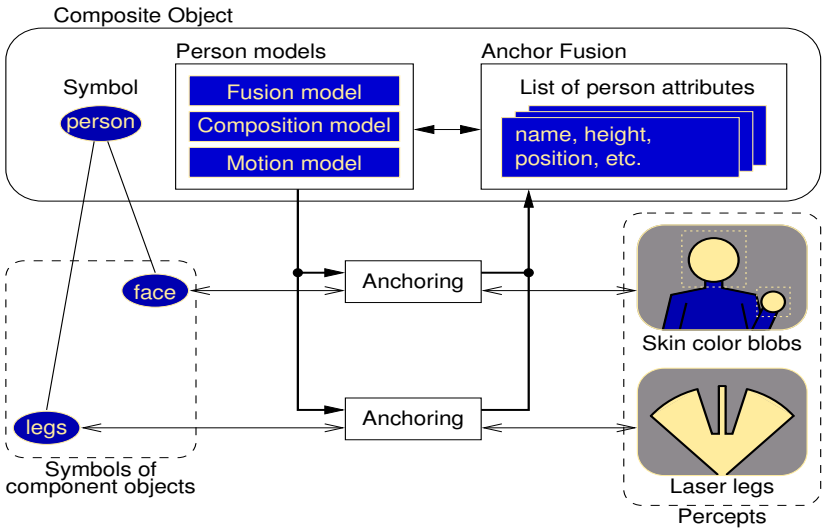

Abbildung 21: Anchoring *im Ausgangsprogramm: Die Abbildung zeigt die Verbindungen zwischen den an der Perzeptverarbeitung beteiligten Module. Jedes Symbol* Face *oder* Leg *wird über einen* Anchor *mit einem zugehörigen Perzept verbunden. Hierzu müssen die sog.* Person-Models *erfüllt sein. Es besteht aus drei Teilmodellen: einem* Fusion-Model, *einem* Composition-Model *und einem* Motion-Model.

Die Perzepte sind über die zugehörigen *Anchor* mit dem jeweiligen Symbolen verbunden. Die Einzelsymbole werden im *Composite-Object* unter dem Symbol Person zusammengefasst. Außerdem beinhaltet das *Composite-Object* über das *Anchoring* die zugehörigen Attribute (siehe Abb. 21). Mit den Messwerten und Auswertungen aus den Sensorendaten wird in den einzelnen *Person-Models* gearbeitet. Dort existieren drei Teilmodelle: das *Fusion-Model*, das *Composition-Model* und das *Motion-Model*. Mit Hilfe des *Fusion-Model* werden die Attributdaten des *Composite-Object* mit den Daten der *Component-Anchor* aktualisiert. Die Daten aus den *Component-Anchor* werden asynchron geliefert. Damit sie aber in der chronologisch korrekten Reihenfolge abgearbeitet werden, haben die Daten einen Zeitstempel anhand dessen sie in eine Liste eingeordnet werden. Anhand des *Motion-Model* wird eine Vorhersage der möglichen Positionen des *Composite-Object* (Personen vor dem Roboter) für den nächsten Zeitschritt erstellt. Hierbei wird berücksichtigt, wie weit sich eine detektierte Person in einem Zeitschritt bewegt haben könnte. Dies ist besonders wichtig, da die Sensoren und das Steuerprogramm eine begrenzte zeitliche Auflösung aufweisen. Geringe Veränderungen der Position zwischen den einzelnen Zeitschritten sollen der Bewegung einer Person

zugeschrieben werden. Hierbei spielt die schon genannte Liste eine wichtige Rolle, denn ohne sie wäre es nicht möglich Bewegung zu detektieren. Somit würde ein Mensch der sich vor dem Roboter bewegt immer als neue Person erkannt. Dies ist natürlich unerwünscht. Das *Composition-Model* legt fest, welcher *Leg-Anchor* und *Face-Anchor* zu einem *Composite-Anchor* gehören können. Pro Person existiert maximal ein *Face-* und *Leg-Anchor*. Mit Hilfe des *Composition-Model* wird berechnet, welche Abweichungen der gemessenen Perzeptposition von der vermuteten Position der Person noch akzeptiert werden. Es werden die Perzepte verworfen, rein logisch nicht möglich sind (siehe Abb. 22). Es wird hierbei berücksichtigt, dass sich die Perzepte Gesicht und Beine bei einer Person nicht beliebig weit voneinander entfernt sein können. Es wird eine horizontale Verschiebung von 30 cm akzeptiert. Bei einer Seitenansicht können Abweichungen durch Beugen der Person vor dem Roboter entstehen. Auch dies ist bei den Schwellwerten eingeflossen.

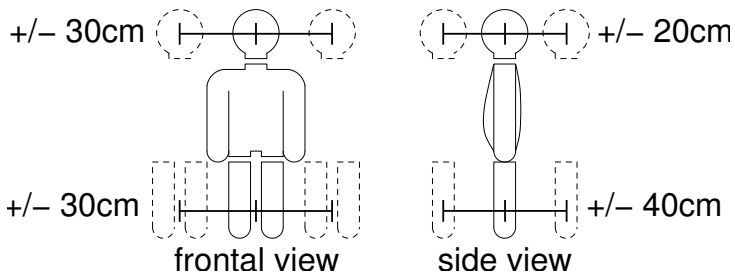

Abbildung 22: Gesichts- und Beinperzepte können nicht beliebig weit voneinander entfernt sein. Deshalb wurde ein Schwellwert von 30 cm angenommen für eine Frontalansicht. Bei einer Seitenansicht können Abweichungen durch Beugen der Person vor dem Roboter entstehen. Auch dies wurde in Schwellwerten berücksichtigt.

Wir erläutern den Sachverhalt aus Abbildung 20 nun noch einmal an einem Beispiel (siehe Abb. 23). Dieses Beispiel zeigt nun das *Multi-Modal-Anchoring*, also das *Anchoring* mit mehreren Perzepten und Symbolen. Zunächst wird das *Person-Anchoring* gestartet. Es befinden sich alle *Component-Anchor* im *Find*-Modus. Zum Zeitpunkt t_1 wird ein Beinpaar gefunden und der *Leg-Anchor* eingerichtet. Anschließend wird die Kamera auf die vermutete Gesichtsposition der Person ausgerichtet.

In t_2 wird ein passendes Gesichtsperzept gefunden. Damit sind beide *Anchor* vom *Find-* in den *Track*-Modus übergegangen. Nach einem weiteren Zeitschritt zum Zeitpunkt t_3 gehen erneut passende Daten für die Beine ein. Somit werden die Daten zuerst für die Beine und dann für die Person aktualisiert. Bei t_4 werden keine passenden Daten für Beinperzepte gefunden. Dadurch geht der *Leg-Anchor* in den *Reacquire*-Zustand über. Dergleichen geschieht in t_5 für das Gesicht. Für den *Leg-Anchor* werden erneut in t_6 passenden Daten gefunden. Somit geht der *Leg-Anchor* wieder in den *Track*-Modus über (siehe Abb. 23). Der übergeordnete

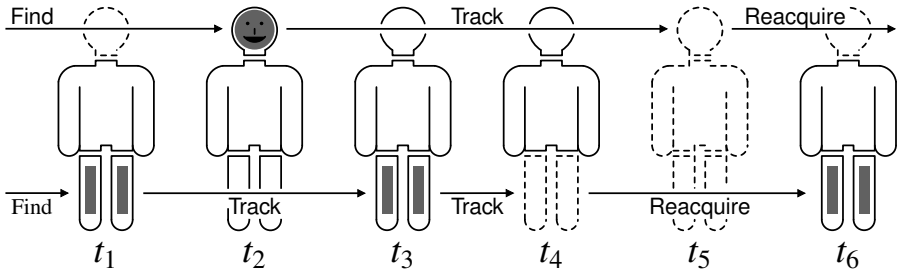

Abbildung 23: Find, Track, Reacquire *Ein Beispieldurchlauf: Die Abbildung zeigt eine Person zu unterschiedlichen aufeinander folgenden Zeitschritten. Nach dem Start des* Person-Anchorings *befinden sich alle* Component-Anchor *im* Find-*Modus. Danach werden die einzelnen* Anchor *gemäß der Abbildung 20 abgearbeitet.*

Composite-Anchor bleibt solange *grounded*, wie mindestens einer seiner *Component-Anchor* im Status *grounded* ist.

4.3.1 Multi-Person-Anchoring

Bis jetzt ist nur der Fall betrachtet worden, dass eine Person durch den Roboter erkannt wird. Um in einer Umgebung mit mehreren Kommunikationspartnern zu interagieren, verwenden wir das sog. *Multipersonanchoring.* Wenn mehrere Gesichter und Beine gefunden werden, findet eine Zuordnung statt. Durch das *Composition-Model* werden Perzepte aussortiert, deren räumlicher Abstand zu groß ist. Befinden sich nun mehrere Personen im Sichtbereich des Roboters so ergibt sich die Schwierigkeit der Zuordnung von Perzepten zu den jeweiligen Personen (siehe Abb. 24). Es ist festgelegt, dass jede Person nur ein Gesichts- und ein Beinperzept zugeordnet bekommt. Vereinfacht ausgedrückt existiert für jedes Perzept, welches einer Person zugeordnet werden kann, eine Bewertung, die ein Maß für das Passen des Gesichts- und Beinperzeptes ist. Über diese Bewertungen wird summiert. Die Bewertung mit der höchsten Summe wird als sehr wahrscheinlich angesehen. Nach ihr erfolgen dann die Zuordnungen der jeweiligen Perzepte zu den Symbolen der Personen.

(Meike Brink)

4.4 Erweiterungen des Anchoring

Unsere Erweiterung besteht in der Ergänzung um die *Anchor* für ein perzeptuelles System, das Stimmen detektiert (Mikrofon mit Stimmerkennungssoftware) sowie für eine Pullovererkennung basierend auf Kamerabildern und Gesichtsdaten und der Anpassung des *Face-Anchor* für das in Kapitel 2 vorgestellte Modul (siehe Abb. 25). Die neu hinzugefügten *Anchor* sollen hier vorgestellt werden.

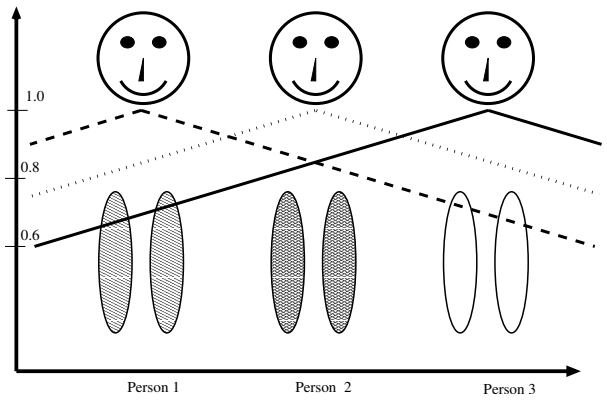

Abbildung 24: Beim Multipersonanchoring *werden die Gesichter- und Beinpaare einander bzw. den Personen zugeordnet. Mit zunehmendem Abstand nimmt die Wahrscheinlichkeit ab, dass sie zusammengehören.*

4.4.1 Voice-Anchoring

Das *Voice-Anchoring* wurde weitestgehend analog zu dem bereits implementierten *Leg-* und *Face-Anchoring* gestaltet, daher sei es an dieser Stelle nur kurz und der Vollständigkeit halber erwähnt. Als perzeptlieferndes System dient das in Kapitel 3 beschriebene Programm *sploc*. Es liefert ein Array von Winkeln, an denen vermeintlich menschliche Audiodaten gefunden wurden. Da kurze Pausen Bestandteile verbaler Kommunikation sind, ergab sich zu Beginn das Problem einen Sprecher kontinuierlich zu verfolgen. Häufig ging während einer Sprachpause der *Voice-Anchor* verloren. Daher haben wir die Zeitspanne, bevor der *Voice-Anchor* mangels Daten verloren geht, von zwei auf fünf Sekunden erhöht. Dieser Wert hat sich in Probeläufen als vorteilhaft erwiesen.

4.4.2 Pullover-Anchoring

An dieser Stelle möchten wir besonders auf das *Pullover-Anchoring* eingehen, da sich das entsprechende Modul in seiner Arbeitsweise von den Modulen *Leg*, *Face* und *Voice* unterscheidet. Es sei hier vorangestellt, dass zur Zeit nur die Vorbereitung für ein solches perzeptuelles System innerhalb des *Anchoring*-Programmes existiert, jedoch noch kein funktionsfähiges Programm, welches Perzeptdaten liefern könnte.

Unsere Motivation, ein perzeptuelles System zur Verarbeitung von Pulloverdaten zu entwickeln lag darin begründet, dass der Roboter auch in der Lage sein soll, einem erfolgreich identifiziertem Kommunikationspartner zu folgen, wenn er sich umdreht. Da bei der Rückansicht eines Menschen keine Gesichtsperzepte zur Verfügung stehen und eine Person im Allgemeinen beim Voranschreiten nicht verbal mit dem Roboter interagieren wird, sollen die Pulloverdaten neben den Beinperzepten zur dauerhaften Erfassung dienen.

Als vorrangiges Problem stellte sich uns vor allem die Frage, wie der Pullover

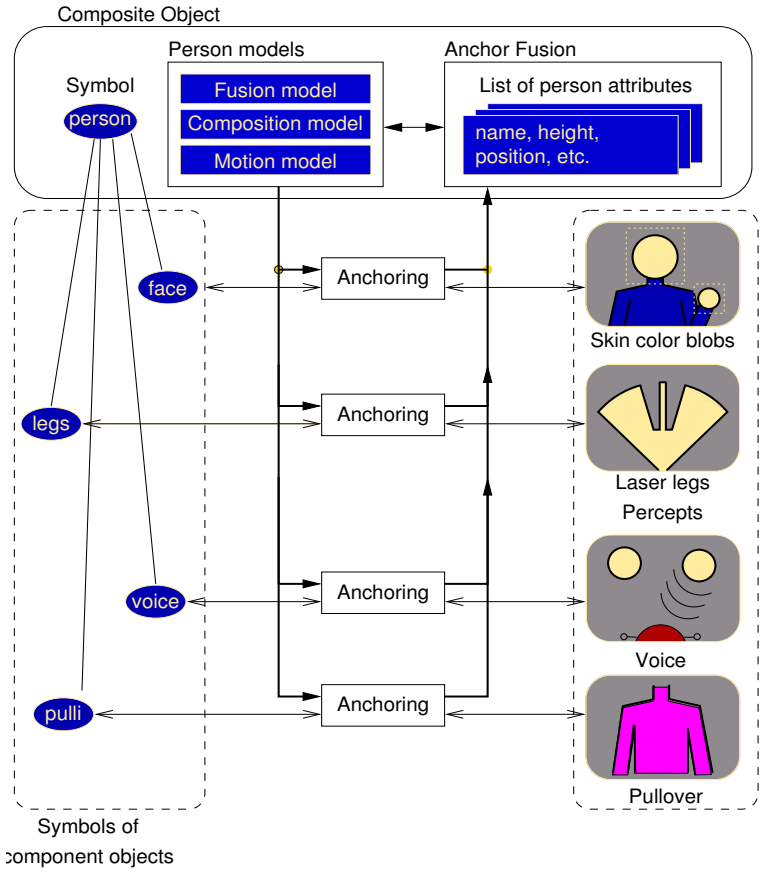

Abbildung 25: Das ursprüngliche Anchoring *aus Abb. 21 wurde um die beiden* Anchor *für Pullover und Stimme erweitert. Die* Person-Models *wurden jeweils daran angepasst. Zusätzlich zu den Gesichts- und den Beinperzepten werden jetzt auch auch Pullover- und Stimmperzepte über einen* Anchor *mit dem passenden Symbol verknüpft.*

einer Person zuverlässig gefunden werden könnte. Im Gegensatz zu Gesichts- oder Audiodaten ist die Menge an Gemeinsamkeiten zur korrekten Klassifikation bei unterschiedlichen Pullovern, oder genereller Oberbekleidungsstücken, zu gering. So können die Größe und Form abhängig von der Statur der Person stark variieren, und es sind prinzipiell beliebige Farben und Texturen vorstellbar.

Wir haben uns daher dafür entschieden, auf bereits existierende Anchordaten einer Person zurückzugreifen, um ein Suchgebiet für den zu einer Person gehörenden Pullover zu definieren (siehe Abb. 26). In diesem Punkt unterscheidet sich das Pulloversystem erheblich von allen anderen. Die Systeme zur Ermittlung von Bein-, Gesichts-, und Stimmperzepten liefern ihre Daten unabhängig voneinander. Das Programm zur Bestimmung von Pulloverpositionen benötigt jedoch zur Initialisierung eine bereits erfolgreich gefundene Gesichtsregion. Wurde die Initialisierung durchgeführt, so soll das Pulloverprogramm auch ohne weitere Gesichtspositionen

Pulloverdaten liefern, sofern sich diese in Farbe und Textur nicht zu sehr von den gemessenen Daten der Initialisierung unterscheiden.

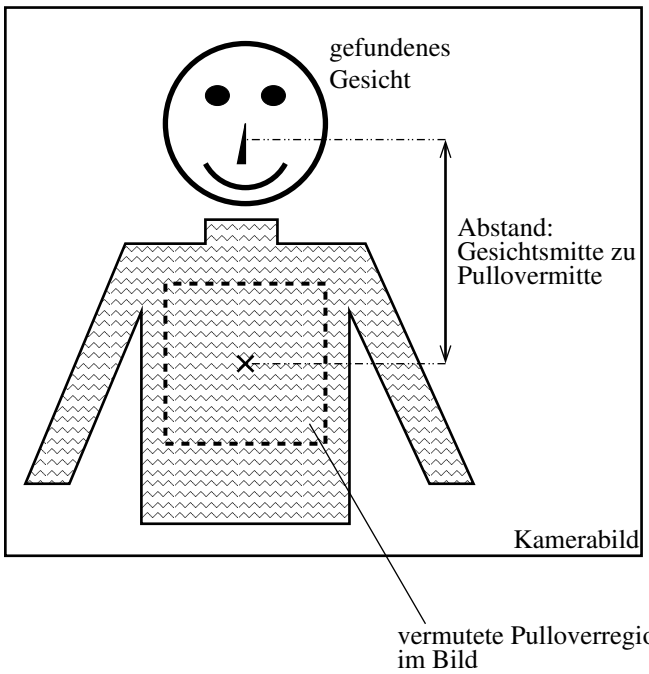

Abbildung 26: Die vermutete Pulloverregion lässt sich in Größe und Position aus der Gesichtsgröße und der gefundenen Gesichtsposition der Person bestimmen, da sich der Pullover immer eine konstanten Abstand unterhalb des Gesichtes eines Menschen befindet, und ein festes Verhältnis zwischen der Durchschnittsgröße eines Gesichtes und eines Pullovers existiert.

Durch die gefundene Gesichtsregion kann man den Abstand der Person ermitteln, indem man die Gesichtsgröße auf dem aktuellen Kamerabild mit einer vorgegeben Gesichtsgröße bei bekanntem Abstand vergleicht. Des Weiteren kann man aus den Pixelkoordinaten der Gesichtsmitte im Bild und der *Pan-Tilt*-Position der Kamera den horizontalen und vertikalen Winkel des Gesichtes im Raum feststellen. Wir kehren dieses Verfahren um: Aus den gegebenen Winkel- und Abstandswerten eines Punktes im Raum und unter Berücksichtigung der Kameraposition erhält man dessen Pixelkoordinaten im Bild. Geht man davon aus, dass sich bei einem Menschen der Mittelpunkt eines Pullovers immer eine konstante Anzahl von Zentimetern unterhalb der Gesichtsmitte befindet, so kann man diese Position ebenfalls in Pixelkoordinaten transformieren. Die Höhe und Breite der Pulloverregion bestimmt sich aus der Gesichtsgröße im Bild, da wir das Größenverhältnis Gesicht zu Oberkörper (und somit zum Pullover) als konstant annehmen. Da bislang keine Implementierung dieses Moduls existiert, konnten hierzu geeignete Werte noch nicht ermittelt werden. Weitere Annahmen und Voraussetzungen für einen erfolgreichen Einsatz

einer Pullovererkennung sind:

- Wir betrachten nur Pullover, deren Vorder- und Rückseite gleich sind.
- Die Pullover besitzen eine möglichst homogene Textur.

Diese Voraussetzungen werden notwendig, da die Erfassung von Pulloverdaten bei der Initialisierung mit einer dem Roboter zugewandten Person erfolgt. Der entsprechende Pullover soll jedoch auch erkannt werden, nachdem sich die Person umgedreht hat. Damit auch Informationen über die Textur des Pullovers verwendet werden können, wäre eine Histogrammanalyse der Farbwerte des potentiellen Pullovers ein vielversprechender Ansatz. Somit wäre es z.B. möglich, zwischen einem rot-weiß gemustertem Pullover und einem rose-farbenem Pullover zu Unterscheiden. Basierend auf den Histogrammdaten kann dann ein RGB-Wert generiert werden. Hierzu wird aus dem Histogramm jedes Farbkanals – also rot, grün und blau – jeweils ein Durchschnittswert berechnet, und dieser an das *Anchoring* -Programm übergeben. Die Histogrammdaten verbleiben in dem Programm, welches für die Erkennung zuständig ist. Nachdem in der Initialisierung die Histogrammdaten berechnet wurden, könnte bei einem neuen Bild nach einem Bereich mit möglichst ähnlichen Histogrammwerten gesucht werden. Hierzu könnte man mit einem Bildausschnitt in der oberen linken Ecke des neuen Bildes beginnen, welches dieselbe Größe wie die Pulloverregion der Initialisierungsphase besitzt. Über diesen Ausschnitt wird eine Histogrammanalyse angefertigt. Das Bild wird dann zeilen- und spaltenweise in entsprechend große Ausschnitte zerlegt und alle so gewonnenen Histogrammdaten mit denen der Initialisierung verglichen. Der ähnlichste Bereich könnte dann als neue Position des Pullovers angenommen werden, sofern die Abweichung unterhalb eines noch experimentell zu ermittelnden Schwellwertes liegt.

Um die Einbindung des *Pullover-Anchorings* in das Gesamtsystem zu verdeutlichen, möchten wir hier den Programmablauf (siehe Abb. 27) anhand des folgenden Szenarios schildern, bei dem es Aufgabe des Roboters ist, einer gefundenen *Person-of-Interest* möglichst lange zu folgen: Eine Person begibt sich vor den Roboter und versucht dessen Interesse zu erregen. Dabei muss die Person ihr Gesicht und ihren Oberkörper der Kamera zuwenden, um eine erfolgreiche Initialisierung des Pullovermodules zu ermöglichen. Danach wendet sich die Person von dem Roboter ab und bewegt sich von ihm fort, während der Roboter der Person folgt. Sobald die Person vor dem Roboter als *Person-of-Interest* erkannt wurde, wozu auch die erfolgreiche Gesichtserkennung notwendig ist, muss der Pulloverdetektor mit der Gesichtsposition im aktuellen Bild initialisiert werden. Über die Region in der wir das Bild des Pullovers erwarten, wird eine Histogrammanalyse durchgeführt. Jedes neue Bild wird wie oben beschrieben gerastert, indem jeweils eine rechteckige Region entsprechend der Bildausschnittsgröße der Initialisierungsphase selektiert und deren Histogrammergebnis mit demjenigen aus der Initialisierung verglichen wird. Sofern die Abweichung des Histogrammes innerhalb einer vorgegebenen Fehlertoleranz liegt, soll das Ergebnis mit der geringsten Abweichung als neue Pulloverposition

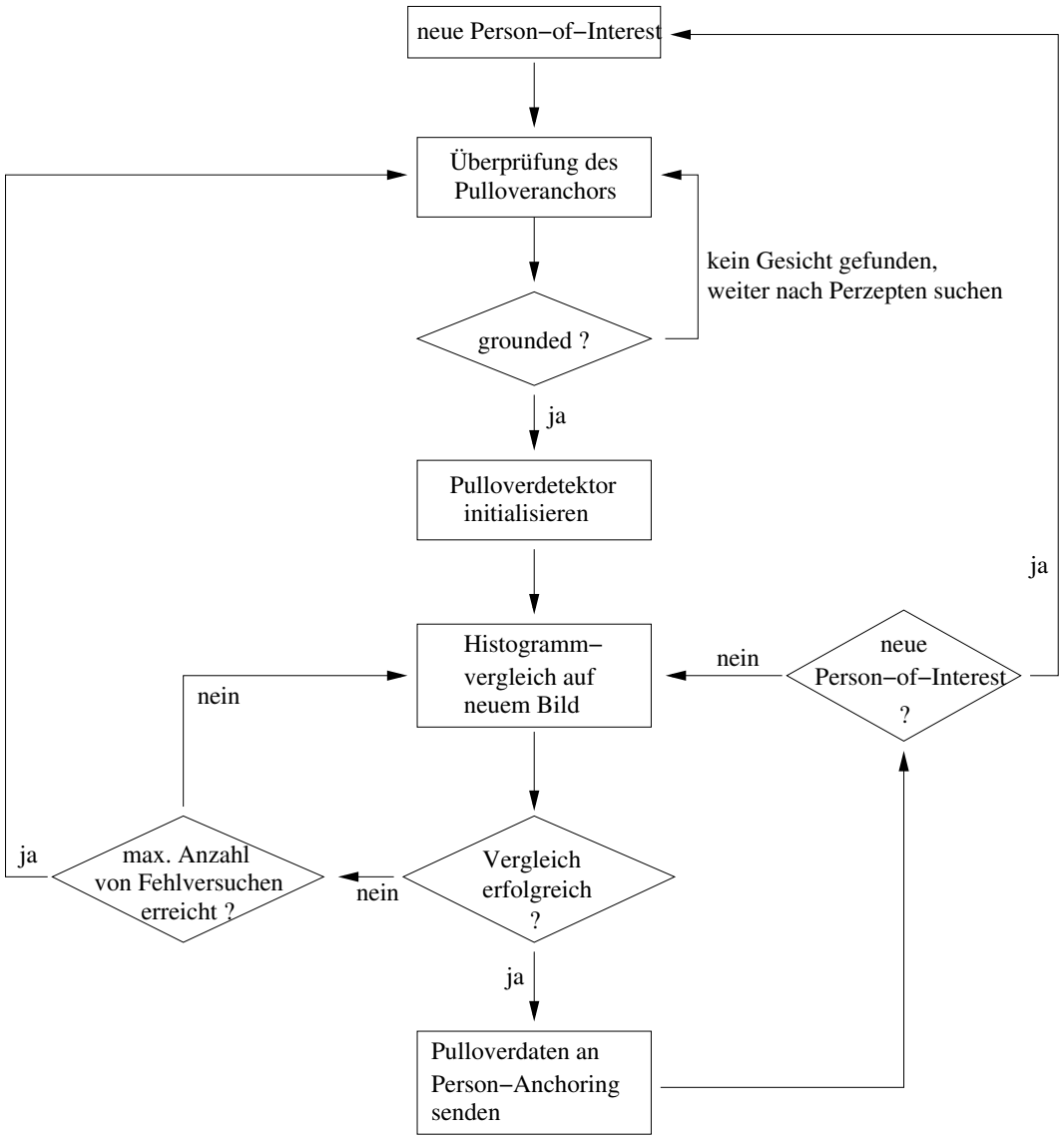

Abbildung 27: Beispiel für den Programmfluß unter Verwendung eines Pullover-Anchors*: Nachdem eine* Person-of-Interest *mit Gesichtspositionen gefunden wurde, wird der Pulloverdetektor mittels eines Histogrammes auf die Region unterhalb des gefundenen Gesichtes initialisiert und liefert solange Pulloverpositionen wie eine Region mit genügend ähnlichem Histogramm auf neuen Bilddaten existiert, oder die* Person-of-Interest *wechselt.*

angenommen und deren Pixelkoordinaten an das *Anchoring*-Programm übergeben werden. Anderenfalls soll eine Rückweisung stattfinden. Damit wird vermieden, dass bei einer kurzzeitig verlorenen *Person-of-Interest* nicht einfach der nächstbeste Histogrammwert als Basis für den *Pullover-Anchor* dient und sich dieser somit im-

mer im Status *grounded* befindet, auch wenn sich die ehemalige *Person-of-Interest* gar nicht mehr in Reichweite befindet. Sollte eine noch zu ermittelnde Anzahl von Rückweisungen hintereinander auftreten, so wechselt der *Pullover-Anchor* in den Zustand *lost*. Alternativ zu einer Begrenzung von Rückweisungen wäre auch eine zeitliche Einschränkung denkbar, so dass der *Pullover-Anchor* in den Zustand *lost* wechselt, wenn eine gegebene Zeit lang kein passendes Histogramm gefunden wurde. Ist der *Pullover-Anchor* in den Zustand *lost* übergegangen und existiert die *Person-of-Interest* noch, so wird eine erneute Initialisierung des Pulloverdetektors ausgelöst, wenn oder sobald zugehörige Gesichtspositionen gefunden sind. Wendet sich wie in unserem Beispiel die *Person-of-Interest* ab, welcher der Roboter folgen soll, so geht zwar die Gesichtsposition verloren, solange aber die Abweichung in den Histogrammwerten den Schwellwert nicht übersteigt, werden weiter Pulloverperzepte geliefert. Wechselt die *Person-of-Interest* basierend auf der Verhaltenssteuerung, so beginnt der Prozess von neuem.

Wie bereits erwähnt, ist dieses Modul bislang nur auf der perzeptverarbeitenden Ebene des Systems implementiert, der nächste Schritt wird es sein, ein entsprechendes perzeptgenerierendes System zu konstruieren. Des Weiteren sei hier darauf hingewiesen, dass die Pullovererkennung nach dem obigen Beispiel nur für die *Person-of-Interest* durchgeführt wird. Eine Erweiterung der Pullovererkennung für eine potentiell beliebige Anzahl von Menschen wurde bereits vorbereitet. So können die auf den Histogrammwerten basierenden RGB-Werte als Attribute des Pullover-Symbols einer Person gespeichert werden. Sollte nun eine verfolgte Person aus dem Kamerabild geraten, z. B. weil durch Erfassung eines neuen akustischen Signals ein Kameraschwenk auf die Schallquelle ausgelöst wurde, so können nach dem Zurückschwenken der Kamera eingehende RGB-Werte mehrerer Personen mit dem alten RGB-Wert verglichen werden. Dies ist nur solange möglich, wie die Ausgangsperson zum Beispiel durch den Leg-Anchor im Status *grounded* gehalten wird, jedoch kann der *Pullover-Anchor* zwischenzeitlich verloren gehen. Bei Personen mit ähnlichen oder gleichen Pullover kann es Verwechslungsprobleme geben, vor allem da durch die Reduzierung von Histogrammen auf RGB-Werte Informationen über die Farbverteilung verloren gehen. Für jede neugefundene Person muss bei erfolgreicher Gesichtsdetektion ebenfalls ein Histogramm im Pulloverdetektor initialisiert werden. Bei der anschließenden Rasterung würde dann auf alle Initial-Histogramme geprüft und die gefundenen Werte müßten mit den einzelnen Personen in Übereinstimmung gebracht werden. Ein weiterer Verbesserungsvorschlag ist, nach jedem erfolgreichen Histogrammvergleich das Referenzhistogramm anzupassen, um variierende Lichtverhältnisse besser zu kompensieren. Außerdem sollte die Pulloverregion des Bildes in ihrer Größe der Entfernung der verfolgten Person angepasst werden, was bedingt, dass Histogrammdaten nicht in absoluten Pixelwerten, sondern relativ zur Ausschnittsgröße berechnet werden.

(Thorsten Spexard)

4.5 Implementierung der Aufmerksamkeitssteuerung

Neben der Einfügung neuer perzeptueller Systeme war es unsere Aufgabe, eine Aufmerksamkeitssteuerung zu implementieren, die die Reaktionen des Roboters, basierend auf allen vorhandenen Sensordaten, einer dem Menschen ähnlichen Aufmerksamkeitserregung anzupassen („Mensch-Mensch-Interaktion“).

Zu Beginn unseres Projekts bestand die Sensorik des Roboters aus einer Kamera und einem laserbasierten Entfernungsmesser. Die Kamera ist eine steuerbare Kamera, eine sogenannte *Pan-Tilt*-Kamera, die in ca. 140 cm Höhe auf dem Roboter angebracht ist. Die Zentral-Position der Kamera entspricht einem Winkel von 0 Grad. In der horizontalen Ebene steht der Kamera ein Bereich von -100 Grad bis +100 Grad zur Verfügung. Der maximale vertikale Bereich reicht von -25 Grad bis +25 Grad. Der laserbasierte Entfernungsmesser, ein sogenannter *Laser-Scanner*, ist 30 cm über dem Boden an dem Roboter installiert. Dieser misst in der horizontalen Ebene Entfernungen, die in einem Bereich von 180 Grad in Halbgradschritten aufgenommen werden. Somit ergeben sich 361 Messwerte pro Messung. Für unser Projekt wurde der Roboter zusätzlich zu den vorhandenen Sensoren mit zwei Mikrophonen ausgestattet. Diese haben einen Abstand von ca. 25 cm zu einander und wurden parallel zur Front des Roboters angebracht. Durch die Erweiterung ist es nun möglich, die Richtung von Audiosignalen im Raum zu lokalisieren (siehe Kap. 3).

Das Verhalten des Roboters bestand zu Beginn unserer Arbeit lediglich aus dem Folgen einer gefundenen Person. Initialisiert wurde eine solche Interaktion, indem sich eine Person frontal, in einem Abstand von ca. 80 cm, vor den Roboter stellte. Sobald die Person vom Roboter gefunden wurde, befand sich diese Person im Zustand *Anchored* (siehe Kap. 4.2). Der Roboter folgte dann dieser sogenannten *Person-of-Interest*. Der Roboter konnte also nur eine Person erkennen und dieser (unter festgelegten Bedingungen) folgen. Diese Art der Verhaltenssteuerung war noch keine vollwertige Aufmerksamkeitssteuerung. Damit der Roboter zuverlässig einer Person folgen konnte, musste sich die Person rückwärts durch den Raum bewegen und dabei immer in die Kamera blicken. Doch aufgrund variierender Beleuchtungsverhältnisse, (problematisch für die Verarbeitung der Gesichts-Erkennung, die deshalb in unserem Projekt verbessert wurde (siehe Kap. 2)) sowie Tisch- und Stuhlbeine (problematisch für die Verarbeitung der Bein-Erkennung) konnte auch auf diese Weise nicht immer die zu verfolgende Person ermittelt werden. In diesem Fall wurde das Verfolgen beendet. Außerdem war es mit der bisherigen Verhaltenssteuerung nicht möglich mehrere Personen im Sensor-Bereich des Roboters zu lokalisieren. Dies wurde erst durch die Implementierung vom *Multi-Person-Anchoring* (siehe Kap. 4.3.1) möglich.

Unser Ziel war es, das Verhalten des Roboters auf die „Suche nach einem Kommunikationspartner“ entsprechend [7] auszurichten. Kommunikationspartner wird die Person, die vor dem Roboter *steht*, in seine Richtung *schaut* und *spricht*. Ist eine Person erstmal Kommunikationspartner, soll diese es bleiben, auch wenn diese aufhört zu sprechen oder nicht mehr in die Kamera schaut. Erst wenn diese Person sich vom Roboter entfernt oder eine andere Person im Sensor-Bereich zum

Kommunikationspartner wird, weil diese Person alle Auswahl-Kriterien für das implementierte Verhalten des Roboters (*steht*, *spricht* und *schaut*) erfüllt, verliert diese Person den Status *Kommunikationspartner*. Von vornherein wird keine Person, die nur am Roboter vorbei geht, Kommunikationspartner. Tabelle 2 stellt die Auswahl-Kriterien für das implementierte Verhalten des Roboters auf der Suche nach einem neuem Kommunikationspartner dar.

Implementiert haben wir eine Funktion, die berechnet, ob eine Person *steht* oder geht. Dazu benutzen wir die Personen-Daten, die wir vom *Anchoring* bekommen. Über die letzten 5 Positionen der Person, gemessen in Grad, wird eine Standard-Abweichung S ermittelt.

$$S = \sum_{i=0}^{5} |p_i - P| \tag{11}$$

P ist dabei die mittlere Position während der 5 Positionen. Überschreitet die Standard-Abweichung einen Wert von 6 Grad, wird die Person als nicht *stehend* erkannt. Die 6 Grad haben wir durch Evaluationen ermittelt. Kleinere Werte führten dazu, dass jede Person als gehend (*steht nicht*) erkannt wurde und größere Werte ergaben, dass jede Person als stehend *steht* erkannt wurde. Ob eine Person *spricht* stellen wir fest, indem wir überprüfen, ob der *Voice-Anchor* dieser Person *grounded* ist (siehe Kap. 4.4.1). Genauso stellen wir fest, ob eine Person in die Kamera *schaut* (*Face-Anchor* ist *grounded*).

steht	**spricht**	**schaut**	**ist KP**
Nein	Ja	Ja	Nein
Nein	Ja	Nein	Nein
Nein	Nein	Ja	Nein
Nein	Nein	Nein	Nein
Ja	Ja	Ja	Ja
Ja	Ja	Nein	Nein
Ja	Nein	Ja	Unverändert
Ja	Nein	Nein	Unverändert

Tabelle 2: Auswahl-Kriterien für die Suche nach einem Kommunikationspartner(KP): Unverändert: *Roboter übernimmt Status der Person im vorherigen Zeitschritt.*

Allgemein kann davon ausgegangen werden, dass in einer natürlichen Umgebung nicht immer nur eine Person die Auswahl-Kriterien für das implementierte Verhalten des Roboters erfüllt. Zum Beispiel könnten zwei Personen, A und B, vor dem Roboter *stehen* und ihn *anschaut* (siehe Abb. 28). Person A unterhält[7] sich mit dem Roboter und wird von der Kamera zentriert. Person B *spricht nicht* und ist außerhalb des

[7]Im Moment bedeutet dies, dass die Person in Richtung des Roboters *spricht*.

Kamera-Bildes. Anhand der Auswahl-Kriterien für das implementierte Verhalten des Roboters ist Person A Kommunikationspartner, da die Person *steht*, *spricht* und in die Kamera *schaut*. Stellen wir uns weiter vor, Person A hört auf zu sprechen (*spricht nicht*) und Person B fängt an zu *sprechen*. Dadurch wird Person B zum potentiellen KP und die Kamera schwenkt zu dieser *Person-of-Interest*[8]. Da diese Person in die Kamera *schaut*, vor dem Roboter *steht* und zu ihm *spricht*, wird jetzt diese Person B zum neuen Kommunikationspartner, woraufhin sich der Rumpf des Roboters auch auf diese Person ausrichtet.

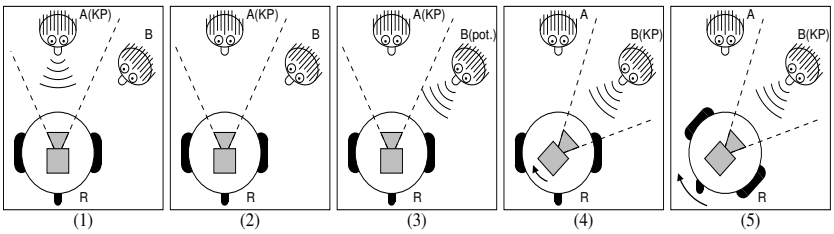

Abbildung 28: Beispiel für die Verhaltenssteuerung: (1) Zwei Personen stehen beim Roboter R. Person A ist Kommunikationspartner (KP). (2) A bricht die Unterhaltung ab (spricht nicht), bleibt aber KP, da B als KP nicht in Frage kommt. (3) B spricht zum Roboter, ist somit potentieller KP. (4) Die Kamera schwenkt in Richtung von B. B ist neuer KP, da dieser zum Roboter spricht. (5) Auch der Rumpf schwenkt zu B.

Für den Fall, indem sich mehrere Personen im Sensor-Bereich aufhalten, aber keine Person alle Auswahl-Kriterien für die Suche nach einem Kommunikationspartner erfüllt, lassen wir den Roboter nach der „interessantesten" Person suchen, also nach der *Person-of-Interest*. In unserer Version der Aufmerksamkeitssteuerung ist die Person *Person-of-Interest*, die die meisten Kriterien erfüllt. Gibt es zwei oder mehr Personen, die gleich interessant sind, wird zufällig die erste Person in der Verarbeitungsreihenfolge zur *Person-of-Interest*. Mit dieser Erweiterung haben wir erreicht, dass einzelne Sensorausfälle nicht zu einem unvorhersehbaren Verhalten des Roboters führen.

Durch die Implementierung der Aufmerksamkeitssteuerung ist das Verhalten des Roboters natürlicher geworden. Dies wurde durch die Umsetzung der Auswahl-Kriterien und dadurch, dass die Kamera auf einen Kommunikationspartner ausgerichtet wird, erreicht. Der Kombination aller Sensordaten, die im Bereich der Gesichts-Erkennung wesentlich verbessert wurden, ist es zu verdanken, dass der Roboter nun wesentlich stabiler arbeitet. Hierzu haben auch die Daten der neu installierten Mikrophone beigetragen. Da die Pullover-Erkennung nur vorbereitet wurde, aber im Rahmen des Projekts nicht realisiert wurde, konnte sie nicht in die

[8]Wichtig: In unserer Version ist die *Person-of-Interest* die Person, die die meisten Auswahl-Kriterien erfüllt.

Aufmerksamkeitssteuerung einbezogen werden. Auch eine ausführliche Evaluation war aus Zeitmangel nicht mehr möglich.

(Patrick Knabben)

4.6 Interne Kommunikation

Unser Ziel, mit einem mobilen Roboter in Echtzeit zu interagieren, stellt auch einige Anforderungen an die Hardware. Für die einzelnen Module muß genügend Rechenzeit zur Verfügung gestellt werden. Auf dem Roboter sind zu diesem Zweck zwei PCs installiert, die sich die Rechenlast teilen und nicht zu hohe Ansprüche an Stromversorgung und Platz stellen. Einer der beiden PCs verfügt über eine Soundkarte und ist direkt an die Antriebsmotoren des Roboters und den Laserscanner angeschlossen. Der andere besitzt eine Frame-Grabber-Karte, an die eine Kamera angeschlossen werden kann. Verbunden sind beide Rechner über ein 100 MBit Ethernet.

Das *Anchoring*-System besteht aus mehreren Modulen (siehe Kap. 4). Das *Person-Anchor*-Modul (im folgendem *panchor* genannt) vereinigt die Daten aus dem perzeptuellen System und ist verantwortlich für die Aufmerksamkeitssteuerung. Das perzeptuelle System untergliedert sich in Module für die Bein-, Gesicht-, Sprach- und Pullover-Erkennung. Eine weiteres Modul kümmert sich um die Visualisierung und zeigt während der Laufzeit den Status der einzelnen *Anchor* an. Aufgrund der Hardware-Konfiguration haben wir die Module so verteilt, dass der Rechner mit der Sound-Karte die Sprachlokalisation, Beinpaarerkennung und die Bewegungssteuerung übernimmt. Der andere kümmert sich um das *panchor*-Modul, die Gesichtserkennung und die Visualisierung. Da die einzelnen Module Daten austauschen müssen, benötigten wir ein Werkzeug das den Verwaltungsaufwand übernimmt und die Fähigkeit zur netzwerkweiten Kommunikation besitzt. *DACS* erfüllt genau diese Spezifikationen.

4.6.1 DACS

DACS (Distributed Application's Communication System) wurde im Rahmen des Teilprojektes D3 des Sonderforschungsbereiches 360 „Situierte Künstliche Kommunikatoren“ entwickelt. Es realisiert die Kommunikation zwischen verschiedenen und voneinander unabhängigen Programmen. Diese können parallel auf einem Rechner, oder auf verschiedenen Rechnern im Netzwerk gestartet werden. Dabei stehen neben dem Basissystem zur Kommunikation eine Reihe von Werkzeugen zur Verfügung, die das Entwickeln und Testen verteilter Systeme erleichtern. Hauptsächlich haben wir uns dabei auf DDT (Dacs Debugging Tool) beschränkt, welches es uns ermöglichte die Datenströme der Module einzusehen und zu Testzwecken Einfluss zu nehmen.

Aus *DACS* benötigten wird hauptsächlich die Funktionalitäten der *Streams* und *Function-Calls*.

Ein *Stream* realisiert eine ungerichtete, asynchrone Kommunikation. Ein Programm, das in Datenstrukturen gekapselte Daten (zum Beispiel Ergebnisse von

Berechnungen) zur Verfügung stellen möchte, registriert sich einen *DACS-Stream* und legt diese nach jedem Zyklus auf dem *Stream* ab. Ein oder mehrere Empfänger können zu beliebiger Zeit und netzwerkweit den *Stream* abonnieren und die aktuellen Daten empfangen. Dabei kann man noch entscheiden ob nur der aktuellste Datensatz benötigt wird, oder ob alle empfangen werden sollen. Ist beispielsweise der Empfänger mit aufwendigen Berechnungen beschäftigt hat das keinen Einfluß auf den Sender. Sobald der Empfänger mit seinen Berechnungen durch ist kann er entscheiden ob die Ergebnisse die in der Zwischenzeit angefallen waren, noch von Bedeutung für ihn sind oder ob sie verworfen werden sollen. *DACS* übernimmt hier den Verwaltungsaufwand des Zwischenspeicherns und des Verteilens der Daten an die Empfänger.

Mit einem *Function-Call* kann ein Programm netzwerkweit spezielle Funktionalitäten zur Verfügung stellen. Die Funktion kann dann von anderen Programmen über *DACS* aufgerufen werden. Berechnet wird sie aber auf dem Rechner, der sie anbietet. Als Eingabe- und Rückgabeparameter können die selben Datenstrukturen wie bei den *Streams* verwendet werden.

4.6.2 Einsatz von DACS im Anchoring

Die einzelnen Module des *Anchoring*-Systems sind als eigenständige Programme entworfen. Sie können einzeln zum Betrieb hinzugeschaltet oder abgeschaltet werden. Die Module des perzeptuellen Systems berechnen anhand ihrer Sensordaten mögliche Perzepte und legen diese auf einem *DACS-Stream* ab. Da die Module unterschiedlich komplex sind stehen die Ergebnisse zu unterschiedlichen Zeiten mit unterschiedlicher Frequenz zur Verfügung. Das *panchor*-Modul abonniert sich die *Streams* der einzelnen Module und liest die berechneten Ergebnisse aus. Für diesen Schritt greifen wir auf die netzwerkweiten Fähigkeiten von *DACS* zurück (siehe Abb. 29). Intern leitet das *panchor*-Modul die Perzepte an die einzelnen *Anchoring*-Ebenen weiter, wo sie fusioniert werden (siehe Kap. 4.3).

Bei Aufnahme der Sensordaten wird ein Zeitstempel generiert der den Ergebnissen der einzelnen Module hinzugefügt wird. Dieser wird bei der Fusionierung zur zeitlichen Einordnung der Daten benötigt. So ist das *panchor*-Modul in der Lage die asynchron vorliegenden Ergebnisse richtig einzuordnen, und ist nicht mehr abhängig von den Berechnungszeiten. Auf der anderen Seite müssen die Module des perzeptuellen Systems nicht auf das *panchor*-Modul warten, sondern können sich nach einem Zyklus sofort um die nächsten Sensordaten kümmern. *DACS* sorgt dafür, dass das *panchor*-Modul die Ergebnisse erhält.

Für die Pullovererkennung ist es erforderlich, dass das *panchor*-Modul Einfluss auf das Pullover-Modul nehmen kann. Die Pullovererkennung kann erst initialisiert werden, wenn ein Gesicht gefunden wurde. Mit dem Gesicht lässt sich auf die wahrscheinliche Position des Pullovers schließen und ein Referenzhistogramm kann aufgenommen werden. Der Schwerpunkt des angenommenen Pullovers wird als Parameter in einem *Function-Call* an das Gesten-Programm übermittelt, der die Initialisierung

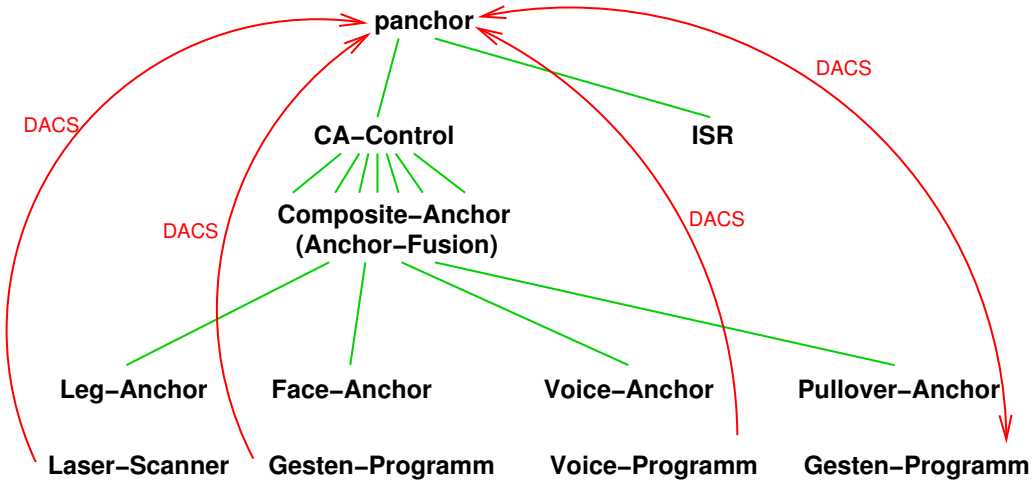

Abbildung 29: Kommunikation innerhalb des Anchoring*-Systems. Panchor sammelt die Ergebnisse der einzelnen Module über* DACS *ein und reicht sie intern an die verschieden* Anchoring*-Ebenen weiter. Die Aufmerksamkeitssteuerung zeigt Reaktionen durch Bewegen des Roboters über die Robotersteuerungs-Software ISR*

des Erkenners anstößt (siehe Kap. 4.4.2). Der Parameter ist wieder eine Datenstruktur, in dem die Informationen gekapselt sind. Damit *DACS* versteht damit umzugehen müssen für die Datenstrukturen Repräsentationen in einem einheitlichen Format vorliegen.

4.6.3 Format der Datenstrukturen für DACS

Das *DACS*-System verlangt für die über das Netzwerk verschickten Datenstrukturen eine an das vom ONC-RPC (Open Network Computing Remote Procedure Call) her bekannte Format XDR (External Data Representation) angelehnte Repräsentation von Daten NDR (Network Data Representation). Für jedes einzelne Modul aus dem perzeptuellen System existiert eine eigene Datenstruktur, mit der Ergebnisse der Berechnungen verschickt werden.

Die *Face*-, *Voice*- und *Leg*-Module verarbeiten nach ihrem Start kontinuierlich Sensor-Daten und legen ihre Ergebnisse auf ihrem *Stream* ab. Diese Ergebnisse sind einfache C-Structs für die entsprechende NDR-Repräsentation angelegt wurden.

Das *Pullvoer*-Modul muss erst noch initialisiert werden. Dafür wird eine Kommunikation von *panchor*- zum *Pullover*-Modul benötigt. Erst mit gefundenem Gesicht kann die Pullovererkennung initialisiert werden. Aus Gesichtsposition kann auf Pulloverposition geschlossen werden (siehe Kap. 4.4.2). Dieses Wissen wird als Parameter in einem *Function-Call* übermittelt. Die Pullovererkennung nimmt ein Referenzhistogramm auf und versucht auf folgenden Bildern den Pullover wieder zu finden. Das Ergebnis wird wie bei den anderen Modulen auf einen Stream gelegt. Für all diese

Datenstrukturen die übermittelt werden, musste eine *NDR* Repräsentation angelegt werden.

4.6.4 NDR Datenstrukturen

Die verschiedenen Module des perzeptuellen Systems gewinnen aus den Sensordaten unterschiedliche Informationen über den Menschen, der mit dem Roboter interagieren möchte. So unterscheiden sich auch die jeweiligen Datenstrukturen (siehe Kap. 4.2). In diesem Abschnitt wird auf den Inhalt der Datenstrukturen eingegangen, die über *DACS* verschickt werden. Die versendeten Daten haben schon einen symbolhaften Charakter, der die anfallende Datenmenge minimiert. In jeder Datenstruktur ist ein Zeitstempel enthalten, der bei Aufnahme der Sensordaten generiert wurde. Dieser ist wichtig für die zeitliche Fusionierung der Ergebnisse. Die Unterschiede werden im folgenden behandelt.

Voice Die Voice-Struktur enthält ein Feld mit bis zu 10 *float* Werten, die jeweils den Winkel einer Geräuschquelle enthalten. Mehr mögliche Sprecher zu verwalten bringt keinen Gewinn für das System, da nur die konstant lautesten Quellen übernommen werden (siehe Kap. 3.2.3). So läßt sich insgesamt der Rechen- und Verwaltungsaufwand minimieren.

Face Die Face-Struktur besteht aus einem dynamisches Feld, welches die X,Y Position der linken oberen Ecke eines Quadrats um das Gesicht, Größe des Quadrates in Pixeln und eine ID der Person enthält. Mit Größe kann man auf die Entfernung einer Person schließen. Die ID ordnet der Person einen Namen zu.

Pullover Die Pullover-Struktur enthält die X,Y Position des Schwerpunktes des erkannten Pullovers und einen Referenzfarbwert des Pullovers in RGB. Mit dem Referenzfarbwert kann man auf anderer Ebene im Anchoring-System die Zuordnung von Pullovern zu Namen vornehmen. Da die Pullovererkennung noch nicht vollständig implementiert wurde, konnte dies auch noch nicht eingebunden werden (siehe Kap. 4.4.2).
Die Pullovererkennung kann außerdem noch über einen *Function-Call* über *DACS* angesprochen werden (siehe Kap. 4.6.2). Der Eingabeparameter enthält die X,Y Position des Schwerpunktes des Pullovers, auf die im *panchor*-Modul geschlossen werden kann. Außerdem enthält es den aktuellen Winkel der Kamera und einen Integer, mit dem das Rechenverhalten der Pullovererkennung gesteuert werden kann. Solange das Gesicht einer Person gefunden wird, ist es nicht notwendig auf die Position des Pullovers zurückzugreifen. Also kann die Rechenzeit von den anderen Modulen genutzt werden.

Leg Die Leg-Struktur wurde von uns nicht angepasst. Diese lag schon funktionsbereit vor. Sie enthält eine Liste von Winkeln, an denen Beinpaare gefun-

den wurden. Außerdem wird für jedes Beinpaar die Entfernung zum Roboter mitübermittelt.

4.7 Parameter

Das Verhalten des Gesamtsystems kann über eine Vielzahl von Parametern beeinflußt werden. Jedes einzelne Modul bringt seine eigenen Parameter mit, die extern in einer gemeinsamen Datei zusammengefaßt werden. Die hier hinterlegten Werte können verändert werden, ohne dass dabei eine neue Übersetzung der Quelltexte nötig wird. Hier sind unter anderem hardwarespezifische Daten hinterlegt, auf die die einzelnen Module zurückgreifen müssen. Die wichtigsten sind hier die Kamerahöhe, Abstand der Stereobasis und Mikrofonhöhe. Aber auch vor allem softwarespezifische Einstellungen werden hier vorgenommen, wie zum Beispiel die maximale Anzahl der *Composite-Anchor*. Die Werte einiger Parameter ergaben sich schon in der Entwicklungsphase. Auf diese soll hier eingegangen werden. Andere wurden erstmal auf Standardwerte gesetzt, die zu einem späteren Zeitpunkt noch angepasst werden müssen.

Der mobile Roboter reagiert auf gefundene Personen mit Ausrichten der Kamera und Drehen des Rumpfes (siehe Kap. 4.5). Die Bereiche, ab wann bewegt werden soll, wenn eine getrackte Person sich zu weit entfernt, werden hier eingestellt. Für X,Y-Richtung der Kamera ergaben sich 7 und 2 Grad. Das bedeutet, sobald sich eine Person weiter als 7 Grad nach links oder rechts bewegt, wird die Kamera nachgeführt. Stellt man diese Werte zu hoch ein kann es vorkommen, dass der Roboter die Person verliert, ohne sich hinterher bewegt zu haben. Wählt man die Winkel aber zu klein, befindet sich die Kamera ständig in Bewegung und die Gesichtserkennung versagt.

Wenn einem *Anchor* zu einem bestimmtem Zeitpunkt keine aktuellen Ergebnisse des perzeptuellen Systems zugeordnet werden können, wird dieser nicht gleich verworfen, sondern eine bestimmte Zeit noch gehalten (siehe Kap. 4.2). Diese Zeiten sind für jedes Modul unterschiedlich und könne ebenfalls als Parameter eingestellt werden. Diese Werte sind von großer Bedeutung wenn man in Echtzeit mit dem Roboter interagieren möchte. Als Standardeinstellung haben sich 2 Sekunden als nützlich erwiesen. Soll der Roboter aber in einem belebten Raum einer Person folgen, ist es nützlich den Voice-Anchor etwas länger zu halten. Bei einem Wert von 4 bis 5 Sekunden lässt sich das System nicht so leicht ablenken und konzentriert sich recht robust auf die getrackte Person. Soll er aber der Konversation zweier Menschen folgen, sollte der Wert wieder gesenkt werden, da er sonst nur eine Person zentriert. *(Martin Saerbeck)*

A Einfügen eines neuen perzeptuellen Systems

In diesem Anhang soll kurz beschrieben werden, welche Ergänzungen und Modifikationen an dem Programm durchzuführen sind, um ein neues perzeptuelles System hinzuzufügen.

A.1 Zu erstellende Klassen und zugehörige Dateien

- ***Perzeptname*Anchor** (*Perzeptname*Anchor.h / -.cc)
- ***Perzeptname*PercAttr** (PercAttr.h / -.cc)
- ***Perzeptname*PercAttrList** (PercAttr.h / -.cc)
- ***Perzeptname*Signature** (*Perzeptname*Anchor.h / -.cc)

*Perzeptname*Anchor

Diese Klasse kann analog zu den bereits bestehenden Perzept-Klassen erzeugt werden und wird in dem gleichnamigen Header- bzw. cc-File gespeichert. Zu beachten ist, dass sie die *Anchor*-Klasse erweitert und somit die in der *Anchor*-Klasse als virtual deklarierten Funktionen implementiert werden müssen.

Hierzu gehören auch die beiden folgenden Funktionen, welche einen leeren Funktionsrumpf besitzen, da sie in unserer Variante des *Anchoring* keine Verwendung finden. Sie basieren auf der Idee zusammengehörige Perzepte und Symbole nur dann zu verbinden, wenn noch weitere Zusatzbedingungen, z.B. eine gewisse Haarfarbe, oder Personengröße vorliegt. Sollte das Verhalten des Roboters in Zukunft derart abgeändert werden, dass er auf Anweisung nur bestimmten Personen (z.B. mit blondeen Haaren) folgen soll, müssen diese Funktionen aktualisiert werden.

- void vComponentMatch(PercAttrList& PAttrList)
- void vCompositeMatch(PercAttrList& PAttrList)

*Perzeptname*PercAttr

Diese Klasse wird in die bereits bestehenden Dateien **PercAttr.h** bzw. **PercAttr.cc** eingefügt und erweitert die *PercAttr*-Klasse.

Sie kann ebenfalls analog zu den bereits bestehenden Unterklassen von PercAttr erstellt werden.

*Perzeptname*PercAttrList

Diese Klasse wird in die bereits bestehenden Dateien **PercAttr.h** bzw. **PercAttr.cc** eingefügt und erweitert die *PercAttrList*-Klasse.

Aufgabe dieser Klasse ist es, eventuell mehrere dem Konstruktor übergebene Datensätze des neuen perzeptuellen Systems in einzelne Perzept-Attribute aufzuteilen, so z.B. sechs während eines Laserscans aufgenommene Bein-Perzepte auf sechs Objekte der *LegPercAttr*-Klasse zu separieren. Dementsprechend sollte der Konstruktor so konzipiert sein, dass er die zur Aufteilung notwendigen Daten aufnehmen kann. Die weitere Implementierung kann analog zu den bestehenden Unterklassen von *PercAttrList* geschehen.

***Perzeptname*Signature**

Diese Klasse wird in die neu erstellten Dateien ***Perzeptname*Anchor.h** bzw. ***Perzeptname*Anchor.cc** hinzugefügt.

Sie verfügt nur über den Standardkonstruktor bzw. -destruktor, sowie über ein öffentliches Klassenattribut vom Typ **ProbPos** pro Datum, welches der entsprechende *Anchor* zur Verfügung stellt. Zum Beispiel existieren zwei **ProbPos** Elemente, wenn der *Anchor* des entsprechenden Perzeptes über Winkel- und Abstandsdaten verfügt.

A.2 Zu aktualisierende Klassen und Dateien

- **CompositeAnchor** (CompositeAnchor.h / -.cc)
- **GTKpanchor** (GTKpanchor.h / -.cc)
- **PercAttr** (PercAttr.h / -.cc)
- **Person** (Person.h / -.cc)
- **Settings** (Settings.h / -.cc)

CompositeAnchor

1. Pointer auf neuen *PerceptAnchor* in Header hinzufügen
2. Neue *PerceptAnchor*-Klasse mittels **# include** in Header einbinden
3. Im Konstruktor Pointer auf neuen *PerceptAnchor* initialisieren
4. Im Destruktor Pointer auf neuen *PerceptAnchor* freigeben
5. In Methode **vResetAnchor()** über Pointer auf neuen *PerceptAnchor* **vResetAnchor()** dieses *Sub-Anchors* auslösen
6. Methode **pAnchor(Symbol s) const** um case *NeuerSymbolName* ergänzen
7. Neue Methode **bGetPerson*Perzeptname*Perceptdatum()** hinzufügen
8. Methode **vPrintInfo** um Ausgabe neuer Perzeptdaten erweitern

GTKpanchor

1. Funktion **void vGet*Perzeptname*Data()** hinzufügen
2. Funktion **void vStore*Perzeptname*Data(*Perzeptname*data_t *in)** hinzufügen
3. Funktion **void vSave*Perzeptname*Data(*Perzeptname*data_t *data)** hinzufügen
4. Funktion ***Perzeptname*data_t * ptLoadFile*Perzeptname*Data (char *load_*Perzeptname*_name)** hinzufügen
5. Funktion ***Perzeptname*data_t * ptLoadFile*Perzeptname*Data (char *filename, int load_data_nr)** hinzufügen
6. Funktion ***Perzeptname*data_t * ptReadDacs*Perzeptname*Data (char **Perzeptname*_dacsname)** hinzufügen
7. Variable **char sz*Perzeptname*_name[100]=““** hinzufügen
8. Variable **int iLoad_*Perzeptname*= 0** hinzufügen
9. Variable **int iLoad_*Perzeptname*_nr = 0** hinzufügen
10. Variable **int iSave_*Perzeptname* = 0** hinzufügen
11. Variable **int iSave_*Perzeptname*_nr = 0** hinzufügen
12. Variable **int i*Perzeptname*_running = 1** hinzufügen
13. Variable ***Perzeptname*_t *pt*Perzeptname*daten** = NULL hinzufügen
14. Variable **struct timeval t_*Perzeptname*** hinzufügen
15. Evtl. neue Flags für Programmstart aus Kommandozeile in **main()** hinzufügen
16. Methode **vSendToRobot(Symbol symb)** um case *NeuerSymbolName* erweitern
17. Abschnitt *„Welche Component-Anchor werden benutzt?“* in .cc-File aktualisieren

PercAttr

1. Im Header-File die Symbole um *Perzeptname* erweitern
   Achtung! *LAST_SYMBOL* muss als letztes Element stehen bleiben

Person

allgemeine Veränderungen:

1. Funktion **void vFuse*Perzeptname*Attrib(HistList::iterator it, bool bFirstFuse=true)** hinzufügen
2. Klasse **HistData** erweitern, sofern *Perzeptname* neue Perzeptgattung (wie Distanz, Höhe, etc.) beinhaltet
3. Funktion **bGetPers*PerzeptnamePerzeptgattung* (float &f*PerzeptnamePerzeptgattung*) const** hinzufügen, wenn neue Perzeptgattung eingeführt wird
4. Funktion **bGetPers*PerzeptnamePerzeptgattung* (ProbPos &pp*PerzeptnamePerzeptgattung*) const** hinzufügen, wenn neue Perzeptgattung eingeführt wird
5. Funktion **bGetLastPers*PerzeptnamePerzeptgattung*(int &iNum, float f*PerzeptnamePerzeptgattung*[]) const** hinzufügen, wenn neue Perzeptgattung eingeführt wird
6. Funktion **bGetLastPers*PerzeptnamePerzeptgattung*(int &iNum, ProbPos pp*PerzeptnamePerzeptgattung*) const** hinzufügen, wenn neue Perzeptgattung eingeführt wird

spezielle Veränderungen im Header-File:

7. Wenn neue Perzeptgattung eingeführt wird, Variablen vom Typ **static PPMoveDisp** und **static PPFuseGauss** hinzufügen
8. Wenn neue Perzeptgattung eingeführt wird, Perzeptcounter **int m_i*PerzeptnamePerzeptgattung*Cnt** einfügen

spezielle Veränderungen im cc-File:

9. *#include* "*Perzeptname*Anchor.h"
10. Funktion **vAddAttribs(Signature const& tSignat)** mehrmals um case *Perzeptname* erweitern
11. In Funktion **vShortenList()** Variable **m_i*PerzeptgattungCnt*** für case *Perzeptname* dekrementieren
12. In Funktion **vPrintList(int iStartPos)** für case *Perzeptname*, *strcpy(szSymb,"Perzeptname")* hinzufügen und AI_DEBUG am Funktionsende aktualisieren

13. Wenn neue Perzeptgattung eingeführt wird, für diese *Movement-* und *Fusion-Model* einrichten
14. Evtl. neu eingerichtete Counter für neue Perzeptgattungen in Konstruktor auf 0 setzen
15. Wenn neue Perzeptgattung eingeführt wird, Funktion **bIsEmpty()** aktualisieren
16. Wenn neue Perzeptgattung eingeführt wird, in Funktion **vReset()** Variable **m_i*Perzeptgattung*Cnt** auf 0 setzen

Settings

Es müssen folgende Variablen hinzugefügt und abhängig von dem neuen perzeptuellen System sinnvoll definiert werden:

1. float f*Perzeptname*Var*Perzeptgattung*
2. float fPersMax*Perzeptname**Perzeptgattung*
3. float fPersMax*Perzeptname**Perzeptgattung*Trck
4. float fPersMax*Perzeptname**Perzeptgattung*Init
5. float fPersDef*Perzeptname**Perzeptgattung*
6. float f*Perzeptname*Rate*Perzeptgattung*Init
7. int i*Perzeptname*Move*Perzeptgattung*Init
8. float fRate*Perzeptname*Max
9. long l*Perzeptname*MaxTimeBetwUpdate
10. int i*Perzeptname*AErrLev = 3
11. int iPersMove*Perzeptname**Perzeptgattung*Init

Achtung:
Beim Aktualisieren der Methode **vInitSettingsParam()** muss die Variable **iNumParams** auf die aktuelle Parameteranzahl angepasst werden.
(Thorsten Spexard)

B Klassendokumentation

B.1 CompositeAnchor Klassenreferenz

`#include <CompositeAnchor.h>`

Öffentliche Attribute

- **LegAnchor * leganchor**

  Pointer auf zugehörigen LegAnchor.

- **FaceAnchor * faceanchor**

  Pointer auf zugehörigen FaceAnchor.

- **VoiceAnchor * voiceanchor**

  Pointer auf zugehörigen **VoiceAnchor** (S. 82).

- **PulloverAnchor * pulloveranchor**

  Pointer auf zugehörigen **PulloverAnchor** (S. 74).

- **bool bIsNewPersonOfInterest**

  Flag, ob **Person** (S. 73) *of Interest gewechselt hat.*

Ausführliche Beschreibung

Die *CompositeAnchor*-Klasse stellt Methoden zur Verfügung, um Informationen zu einer Person zu erhalten. Die Personendaten, wie z.B. Abstand und Winkel zum Roboter basieren auf den bereits ausgewerteten Daten aller zu dieser Person zugeordneten Perzepte. Des Weiteren existieren in dieser Klasse Methoden zur Abfrage des *Anchoring*-Status für eine Person. Mittels der oben aufgeführten Pointer ist es möglich auch auf die zu einem *Composite-Anchor* zugehörigen *Component-Anchor* und deren Daten zuzugreifen.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **CompositeAnchor.h**
- **CompositeAnchor.cc**

B.2 EviCamera Klassenreferenz

`#include <EviCamera.h>`

Ausführliche Beschreibung

Diese Klasse beinhaltet öffentliche Funktionen zur Initialisierung und Steuerung der Kamera, zur Abfrage aktueller Kameradaten, wie z.B. die Position, und Methoden zur Umrechnung zwischen Roboterkoordinaten eines Objektes und dessen Koordinaten in Pixeln auf dem aufgenommenen Bild.

Dokumentation der Elementfunktionen

int EviCamera::RadToxPixel (float *fAngHori*, int *imgWidth*)
Funktion berechnet aus horizontalem Kamerawinkel und Bildbreite die x-Pixel-Koordinate eines Objektes im Bild.
Liegt das Objekt außerhalb des Bildbereiches, wird -1 zurückgegeben

int EviCamera::RadToyPixel (float *fAngVert*, int *imgHeight*)
Funktion berechnet aus vertikalem Kamerawinkel und Bildhöhe die y-Pixel-Koordinate eines Objektes im Bild.
Liegt das Objekt außerhalb des Bildbereiches, wird -1 zurückgegeben

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **EviCamera.h**
- EviCamera.cc

B.3 Face2PercAttr Klassenreferenz

`#include <PercAttr.h>`

Öffentliche Attribute

- **ProbPos ppAng**
  Winkel.
- **ProbPos ppDst**
  Abstand.
- **ProbPos ppHgt**
  Höhe.
- **int iID**
  Identifikations-Nummer.

Ausführliche Beschreibung

Die Klasse erweitert die *PercAttr-Klasse* und stellt die oben genannten öffentlichen Attribute zur Speicherung von Perzeptdaten zur Verfügung, die von dem *iceWing*-PlugIn *Face2* geliefert werden.

Beschreibung der Konstruktoren und Destruktoren

Face2PercAttr::Face2PercAttr (FaceHyp_t * *phypFace*, EviCamera * *pEviCam*, unsigned int *sec*, unsigned int *usec*)

Die Parameter sind:
Pointer auf *Face2*-Hypothesen, Anzahl der Arrayelemente,
Pointer auf Kamerakontrolle zur Winkelbestimmung,
Timestamp in Sek. seit 1970, restliche Mikrosekunden

Dokumentation der Elementfunktionen

void Face2PercAttr::vPrintInfo () `[virtual]`

Perzeptinformationen ausgeben.
Die Funktion verdeckt die Implementation aus *PercAttr*.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **PercAttr.h**
- PercAttr.cc

B.4 Face2PercAttrList Klassenreferenz

`#include <PercAttr.h>`

Ausführliche Beschreibung

Diese Klasse erweitert die *PercAttrList*-Klasse und stellt eine Liste aus den über den Konstruktor erhaltenen Daten zusammen, deren Elemente Instanzen der *Face2-PercAttr*-Klasse sind.

Beschreibung der Konstruktoren und Destruktoren

Face2PercAttrList::Face2PercAttrList (FaceHyp_t ** *rh*, int *iNumRh*, EviCamera * *pEviCam*, unsigned int *sec*, unsigned int *usec*)

Die Parameter sind:
Pointer auf Array von *Face2*-Hypothesen-Pointer, Anzahl der Arrayelemente,
Pointer auf Kamerakontrolle zur Winkelbestimmung, *Timestamp* in Sek.

seit 1970,
restliche Mikrosekunden

Dokumentation der Elementfunktionen

void Face2PercAttrList::vPrintInfo () `[virtual]`

Perzeptinformationen ausgeben, die Funktion verdeckt die Implementation aus *PercAttr*.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **PercAttr.h**
- PercAttr.cc

B.5 GTKpanchor Klassenreferenz

`#include <GTKPanchor.h>`

Private Attribute

- **int iLoad_face2**

  Flag [0/1], ob Face2daten aus Datei gelesen werden

- **int iLoad_face2_nr**

  Flag [0/1], ob einzelne Face2daten aus Datei gelesen werden

- **int iLoad_pullover**

  Flag [0/1], ob Pulloverdaten aus Datei gelesen werden

- **int iLoad_pullover_nr**

  Flag [0/1], ob einzelne Pulloverdaten aus Datei gelesen werden

- **int iLoad_voice**

  Flag [0/1], ob Voicedaten aus Datei gelesen werden

- **int iLoad_voice_nr**

  Flag [0/1], ob einzelne Voicedaten aus Datei gelesen werden

- **int iFace2_running**

Flag [0/1], ob Face2-Perzepte verwendet werden

- **int iPullover_running**
  Flag [0/1], ob Pullover-Perzepte verwendet werden
- **int iSave_face2**
  Flag [0/1], ob Face2daten in Datei gespeichert werden
- **int iSave_face2_nr**
  Flag [0/1], ob einzelne Face2daten in Datei gespeichert werden
- **int iSave_pullover**
  Flag [0/1], ob Pulloverdaten in Datei gespeichert werden
- **int iSave_pullover_nr**
  Flag [0/1], ob einzelne Pulloverdaten in Datei gespeichert werden
- **int iSave_voice**
  Flag [0/1], ob Voicedaten in Datei gespeichert werden
- **int iSave_voice_nr**
  Flag [0/1], ob einzelne Voicedaten in Datei gespeichert werden
- **int iVoice_running**
  Flag [0/1], ob Voice-Perzepte verwendet werden
- **FacePercept_t *ptFacePercept**
  Pointer auf Face2-Daten
- **Pullover_t *ptPulloverdaten**
  Pointer auf Pulloverdaten
- **Voice_t *ptVoicedaten**
  Pointer auf Voicedaten
- **char[] szFace2_name**
  Name des Face2-DACS-Steams
- **char[] szPullover_name**
  Name des Pullover-DACS-Steams
- **char[] szVoice_name**
  Name des Voice-DACS-Steams

Ausführliche Beschreibung

Die *GTKpanchor*-Klasse beinhaltet die *main()*funktion des Anchoring-Programmes. Hier sind die Abfrage von Kommandozeilenparametern bei Programmstart, die Entgegennahme von Daten mittels DACS aus externen Modulen wie z.B. dem Gesten-Programm, die Weitergabe an die entsprechenden datenverarbeitenden Routinen und die Kommunikation mit der ISR-Software implementiert. Außerdem wird in dieser Klasse die grafische Oberfläche initialisiert.

Dokumentation der Elementfunktionen

FacePercept_t* GTKpanchor::ptReadDacsFace2Data (char **face2_dacsname*)

Daten des Face2 Klassifikators von DACS lesen. Der Parameter beinhaltet den Namen des zu lesenden DACS-Streams.

Pullover_t* GTKpanchor::ptReadDacsPulloverData (char **pullover_dacsname*)

Pulloverdaten von DACS lesen. Der Parameter beinhaltet den Namen des zu lesenden DACS-Streams.

Voice_t* GTKpanchor::ptReadDacsVoiceData (char **voice_dacsname*)

Voicedaten von DACS lesen. Der Parameter beinhaltet den Namen des zu lesenden DACS-Streams.

void GTKpanchor::vGetFace2Data ()

Funktion ruft alle Unterfunktionen auf, um Daten des *Face2*-Klassifikators aus Dateien oder von DACS zu verarbeiten.

void GTKpanchor::vGetPulloverData ()

Funktion ruft alle Unterfunktionen auf, um Pulloverdaten aus Dateien oder von DACS zu verarbeiten.

void GTKpanchor::vGetVoiceData ()

Funktion ruft alle Unterfunktionen auf, um Voicedaten aus Dateien oder von DACS zu verarbeiten.

FacePercept_t* GTKpanchor::ptLoadFileFace2Data (char **load_face2_name*)

Alle Daten des *Face2*-Klassifikators aus Datei laden. Der Parameter beinhaltet den Namen der zu lesenden Datei.

FacePercept_t* GTKpanchor::ptLoadFileFace2Data (char *load_face2_name, int load_data_nr)

Nur das Datum mit der Nummer *load_data_nr* des *Face2*-Klassifikators aus Datei laden. Der Parameter *load_face2_name* beinhaltet den Namen der zu lesenden Datei.

Pullover_t* GTKpanchor::ptLoadFilePulloverData (char *load_pullover_name)

Alle Daten des *Pullover*-Klassifikators aus Datei laden. Der Parameter beinhaltet den Namen der zu lesenden Datei.

Pullover_t* GTKpanchor::ptLoadFilePulloverData (char *load_pullover_name, int load_data_nr)

Nur das Datum mit der Nummer *load_data_nr* des *Pullover*-Klassifikators aus Datei laden. Der Parameter *load_pullover_name* beinhaltet den Namen der zu lesenden Datei.

Voice_t* GTKpanchor::ptLoadFileVoiceData (char *load_voice_name)

Alle Daten des *Voice*-Klassifikators aus Datei laden. Der Parameter beinhaltet den Namen der zu lesenden Datei.

Voice_t* GTKpanchor::ptLoadFileVoiceData (char *load_voice_name, int load_data_nr)

Nur das Datum mit der Nummer *load_data_nr* des *Voice* Klassifikators aus Datei laden. Der Parameter *load_voice_name* beinhaltet den Namen der zu lesenden Datei.

FILE* GTKpanchor::ptOpenFace2Data (void)

Datei mit Daten des *Face2*-Klassifikators öffnen

void GTKpanchor::vSaveFace2Data (FILE *fp, FacePercept_t *data)

Daten des *Face2*-Klassifikators in Datei schreiben

void GTKpanchor::vSavePulloverData (FILE *fp, Pullover_t *data)

Pulloverdaten in Datei schreiben

void GTKpanchor::vSaveVoiceData (FILE *fp, Voice_t *data)

Voicedaten in Datei schreiben

void GTKpanchor::vStoreFace2Data (FacePercept_t *in)

Daten des Face2Klassifikators zur Weiterverarbeitung in Perzeptliste schreiben

void GTKpanchor::vStorePulloverData (Pullover_t *in)
Pulloverdaten zur Weiterverarbeitung in Perzeptliste schreiben

void GTKpanchor::StoreVoiceData (Voice_t *in)
Voicedaten zur Weiterverarbeitung in Perzeptliste schreiben

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **GTKpanchor.h**
- GTKpanchor.cc

B.6 Person Klassenreferenz

`#include <Person.h>`

Statische öffentliche Attribute

- **PPMoveDisp MovePulloverAngVert = PPMoveDisp***Settings* :: *iPersMovePulloverAngVertInit*

  Bewegungsmodell für Pullover (vertikal).

- **PPFuseGauss FusePulloverAngVert = PPFuseGauss()**

  Fusion-Modell für Pullover (vertikal).

Private Attribute

- int **m_iPulloverAngVertCnt**

  Anzahl der fusionierten Pulloverwinkel.

Ausführliche Beschreibung

Diese Klasse stellt das sog. Personenmodell zur Verfügung. Jeder von dem System *geanchorten* Person wird eine Instanz dieser Klasse zugeordnet. Mittels der Methoden dieser Klasse läßt sich das *Movementmodell* (hat sich eine Person bewegt, oder handelt es sich um eine neue Person) und das *Fusionmodell* auf eingehende Perzeptdaten anwenden. Nach dem *Fusionmodell* werden die Daten perzeptweisein zeitlich geordnete Listen sortiert und duch Verrechnung mehrerer Daten einer Perzeptliste zu unterschiedlichen Zeiten der aktuelle Wert bestimmt. Hierdurch wird die Auswirkung von Ausreißern verrringert. Außerdem stellt die Klasse Funktionen zum Abrufen der aktuellen Personendaten wie Abstand, Winkel und Höhe zur Verfügung.

Dokumentation der Elementfunktionen

bool Person::bGetPersPulloverAngVert (float & *fPulloverAngVert*) const

Pulloverwinkel (vertikal) einer Person als Referenz holen, die Funktion gibt zurück, ob dieser Wert bereits existiert

void Person::vFusePulloverAttrib (HistList::iterator *itNew*, bool *bFirstFuse* = true) `[private]`

Pullover-Attribute zeitlich fusionieren

void Person::vFuseVoiceAttrib (HistList::iterator *itNew*, bool *bFirstFuse* = true) `[private]`

Voice-Attribute zeitlich fusionieren

void Person::vGetLastPersPulloverAngVert (int & *iNum*, ProbPos *ppPulloverAngVert*[ ])

Funktion liefert die letzten <iNum> angenommenen vertikalen Pulloverwinkel der Person

void Person::vGetLastPersPulloverAngVert (int & *iNum*, float *fPulloverAngVert*[ ])

Funktion liefert die letzten <iNum> echten vertikalen Pulloverwinkel der Person

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **Person.h**
- Person.cc

B.7 PulloverAnchor Klassenreferenz

`#include <PulloverAnchor.h>`

Private Attribute

- **ProbPos ppPredAngHori**

  Vorhergesagter Horizontalwinkel.

- **ProbPos ppPredAngVert**

Vorhergesagter Vertikalwinkel.

- **bool bPredAngHori**

  existiert vorhergesagter Horizontalwinkel.

- **bool bPredAngVert**

  existiert vorhergesagter Vertikalwinkel.

- **PPMoveDisp MoveAngHori**

  Bewegungs-Modell für horizontalen Winkel.

- **PPMoveDisp MoveAngVert**

  Bewegungs-Modell für vertikalen Winkel.

- **PPFuseCp FuseAngHori**

  Fusions-Modell für horizontalen Winkel.

- **PPFuseCp FuseAngVert**

  Fusions-Modell für vertikalen Winkel.

Ausführliche Beschreibung

Die *PulloverAnchor*-Klasse erweitert die *Anchor*-Klasse und beinhaltet Funktionen zur Berechnung, in wie weit ein neues Perzept zu den vorrausberechneten Daten einer Personenkomponente passt. Es werden eingehende Pullover-Positionsdaten mit dem erwarteten Wert für eine Person verglichen und bewertet, je nachdem ob ein eingehendes Datum eine neue Pulloverposition desselben Symbols sein kann, oder außer Reichweite liegt und einem anderen oder neuen Pulloversymbol zugeordnet werden muss.

Beschreibung der Konstruktoren und Destruktoren

PulloverAnchor::PulloverAnchor (CompositeAnchor * *ca*)
Konstruktor, zur Zuornung wird ein Zeiger auf den Übergeordneten *Composite-Anchor* (S. 66) übergeben.

PulloverAnchor::~PulloverAnchor ()
Destruktor

Dokumentation der Elementfunktionen

bool PulloverAnchor::bGetPulloverAngHori (float & *fAngHori*)
Zugriff auf horizontalen Winkel.
Winkeldaten fuer diesen Pullover-Anchor als Referenzen übergeben.

bool PulloverAnchor::bGetPulloverAngVert (float & *fAngVert*)
Zugriff auf vertikalen Winkel.
Winkeldaten fuer diesen Pullover-Anchor als Referenzen übergeben.

bool PulloverAnchor::bGetPulloverCol (int & *iRed*, int & *iGreen*, int & *iBlue*)
Zugriff auf Pulloverfarben (RGB).
Farbwerte fuer diesen *Pullover-Anchor* als Referenzen übergeben.

void PulloverAnchor::vCompModelMatch (PercAttrList & *PAttrList*) `[private]`
Nach Composition Model zu Personen zusammenfügen.
Die Bewertung ungeeigneter Perzepte wird auf 0.0 gesetzt.

void PulloverAnchor::vComponentMatch (PercAttrList & *PAttrList*) `[private]`
Platzhalter, in Anchor.h virtual deklariert (siehe auch Anhang A).

void PulloverAnchor::vCompositeMatch (PercAttrList & *PAttrList*) `[private]`
Platzhalter, in Anchor.h virtual deklariert (siehe auch Anhang A).

void PulloverAnchor::vOneStepPredict () `[private]`
Vorhersage über Pulloverdaten für den nächsten Zeitschritt mittels *Component-Anchor*, wenn im Status *Track*.

void PulloverAnchor::vPredict () `[private]`
Vorhersage über Pulloverdaten für den nächsten Zeitschritt mittels *Composite-Anchor* und Personenmodell, wenn im status *Reacquire*.

void PulloverAnchor::vPrintInfo ()
Status textuell ausgeben.

void PulloverAnchor::vRate (PercAttrList & *PAttrList*) `[private]`
Bewerten der Perzepte.
Geeignete Perzepte werden bewertet (mit Werten aus [0,1]). Perzepte, die schon für ungeeignet erklärt wurden, haben eine Bewertung von 0.0 und muessen daher nicht mehr bewertet werden. Alle anderen Perzepte haben initial eine Bewertung von 1.0 (optimistische Initialisierung).

void PulloverAnchor::vResetAnchor ()
Anchor zurücksetzen

void PulloverAnchor::vUpdate (PercAttr & *PAttr*) `[private]`
Aktualisierung des *Anchors* mit ausgewähltem Perzept

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **PulloverAnchor.h**
- PulloverAnchor.cc

B.8 PulloverPercAttr Klassenreferenz

`#include <PercAttr.h>`

Öffentliche Attribute

- **ProbPos ppAngHori**

  Horizontalwinkel in rad.

- **ProbPos ppAngVert**

  Vertikalwinkel in rad.

- **int iID**

  ID fuer jeden neuen Pullover, anhand derer er identifiziert und vom Pulloverprogramm angesprochen werden kann.

- **int iRed**

  Farbwerte 0 bis 255, -1 = nicht gesetzt.

- **int iGreen**

  Farbwerte 0 bis 255, -1 = nicht gesetzt.

- **int iBlue**

  Farbwerte 0 bis 255, -1 = nicht gesetzt.

Ausführliche Beschreibung

Die Klasse erweitert die *PercAttr-Klasse* und stellt die oben genannten öffentlichen Attribute zur Speicherung von Perzeptdaten zur Verfügung, die von dem *iceWing-PlugIn* Pullover geliefert werden.

Beschreibung der Konstruktoren und Destruktoren

PulloverPercAttr::PulloverPercAttr (int *id*, int *x*, int *y*, int *iR*, int *iG*, int *iB*, long *lTimeStamp*, EviCamera * *pEviCam*)

Die Parameter sind:
ID zur Identifikation mehrerer Pullover (zur Zeit 0, da nur ein Pullover),
x-Koordinaten der Pullovermitte, y-Koordinate der Pullovermitte,
die drei Farbwerte des Pullovers (RGB, Werte: 0-255, -1 = nicht gesetzt),
Timestamp in mSek. seit 1970, Pointer auf Kamerakontrolle zur Winkelbestimmung

Dokumentation der Elementfunktionen

void PulloverPercAttr::vPrintInfo () `[virtual]`

Perzeptinformationen ausgeben.
Die Funktion verdeckt die Implementation aus *PercAttr*.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **PercAttr.h**
- PercAttr.cc

B.9 PulloverPercAttrList Klassenreferenz

`#include <PercAttr.h>`

Ausführliche Beschreibung

Diese Klasse erweitert die *PercAttrList*-Klasse und stellt eine Liste aus den über den Konstruktor erhaltenen Daten zusammen, deren Elemente Instanzen der *PulloverPercAttr*-Klasse sind.

Beschreibung der Konstruktoren und Destruktoren

PulloverPercAttrList::PulloverPercAttrList (int * *ids*, int * *pXCoord*, int * *pYCoord*, int * *pRedColors*, int * *pGreenColors*, int * *pBlueColors*, long *lSec*, long *lUsec*, int *iNumPullover*, EviCamera * *pEviCam*)

Die Parameter sind:
IDs der unterschiedlichen Pullover (zur Zeit 0, da nur ein Pullover),
Array von x-Koordinaten, Array von y-Koordinaten für mehrere Pullover,

drei Arrays für Farbwerte der Pullover (RGB, Werte: 0-255, -1 = nicht gesetzt),
Timestamp in Sek. seit 1970, restliche Mikrosekunden,
Anzahl der Arrayelemente, Pointer auf Kamerakontrolle zur Winkelbestimmung

Dokumentation der Elementfunktionen

void PulloverPercAttrList::vPrintInfo () `[virtual]`

Perzeptinformationen ausgeben
Erneute Implementation von **PercAttrList**.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **PercAttr.h**
- PercAttr.cc

B.10 PulloverSignature Klassenreferenz

`#include <PulloverAnchor.h>`

Öffentliche Attribute

- **ProbPos ppPulloverAngHori**

  horizontaler Winkel in Roboterkoordinaten [rad].

- **ProbPos ppPulloverAngVert**

  vertikaler Winkel in Roboterkoordinaten [rad].

- **int iRed**

  Pulloverfarbwert für Rot (0-255) zwischenspeichern.

- **int iGreen**

  Pulloverfarbwert für Grün (0-255) zwischenspeichern.

- **int iBlue**

  Pulloverfarbwert für Blau (0-255) zwischenspeichern.

Ausführliche Beschreibung

Die Klasse beinhaltet neben dem Kon- und dem Destruktur nur die oben genannten öffentlichen Attribute. Eine Instanz dieser Klasse wird beim *update()* an den *Composite-Anchor* (S. 66) übergeben.

Beschreibung der Konstruktoren und Destruktoren

PulloverSignature::PulloverSignature () `[inline]` Konstruktor

PulloverSignature::~PulloverSignature () `[inline]` Destruktor

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- **PulloverAnchor.h**
- PulloverAnchor.cc

B.11 Settings Klassenreferenz

`#include <Settings.h>`

Statische öffentliche Attribute

- **float fVoiceVarAng** = 0.0027

  mögliche Varianz der Voice-Daten.

- **long lVoiceMaxTimeBetwUpdate** = 2000

  max. Zeitspanne, ab der Voice-Anchor (S. 82) *ohne Perzeptdaten verloren geht.*

- **long lPulloverMaxTimeBetwUpdate** = 2000

  max. Zeitspanne, ab der Pullover-Anchor (S. 74) *ohne Perzeptdaten verloren geht.*

- **float fVoiceRateAngInit** = 1.

  Skalierung des Abstands für Voice-Percept.

- **int iVoiceMoveAngInit** = 250

  Zeit [ms] nach der sich Varianz für Voice-Perzept verdoppelt.

- **int iPulloverMoveAngHoriInit** = 250

  Zeit [ms] nach der sich Varianz für Pullover-Perzept verdoppelt (horizontal).

- **int iPulloverMoveAngVertInit** = 250

  Zeit [ms] nach der sich Varianz für Pullover-Perzept verdoppelt (vertikal).

- **float fPulloverRateAngHoriInit** = 1.

Skalierung des Abstands für Pullover-Perzept in der Horizontalen.

- **float fPulloverRateAngVertInit** = 1.

  Skalierung des Abstands für Pullover-Perzept in der Vertikalen.

- **int iPersMovePulloverAngVertInit** = 250

  Zeit [ms] nach der sich Varianz für Pulloverdaten in Person (S. 73) *verdoppelt.*

- **float fPersMaxVoiceAng** = 0.149

  maximaler Winkelabstand Voice-Perzept - Person (S. 73).

- **float fPersMaxPulloverAngVertTrck** = 0.2

  max. Höhenabweichung des Pullover-Perzepts, wenn Daten vorhanden.

- **float fPersMaxPulloverAngVertInit** = 0.2

  max. Höhenabweichung des Pullover-Perzepts, wenn keine Daten vorhanden.

- **float fPersMaxPulloverAngHori** = 0.149

  maximaler Winkelabstand Pullover-Perzept - Person (S. 73).

- **float fPersDefPulloverAngVert** = 0.0

  vordefinierte Pulloverhöhe, wenn keine Daten vorhanden.

- **int iPoAErrLev** = 3

  initialer Error-Level des Pullover-Anchors.

- **int iVAErrLev** = 3

  initialer Error-Level des Voice-Anchors.

- **float fRateVoiceMax** = 0.29

  max. Winkelabweichung des Voice-Perzepts zur Vorhersage, ab der mit 0 bewertet wird.

- **float fRatePulloverAngHoriMax** = 0.29

  max. Winkelabweichung (seitlich) des Pullover-Perzepts zur Vorhersage, ab der mit 0 bewertet wird.

- **float fRatePulloverAngVertMax** = 0.2

  max. Winkelabweichung (Höhe) des Pullover-Perzepts zur Vorhersage, ab der mit 0 bewertet wird.

Ausführliche Beschreibung

Die Klasse Settings beinhaltet Standardparameter zur Ausführung des Programmes, sowie zwei Methoden zum Laden und Speichern eigener Programmeinstellungen als Textdatei. Bei der ersten Programmausführung wird automatisch eine entsprechende Konfigurationsdatei mit der Bezeichnung *s*ettings.txt angelegt und bei einem erneuten Programmstart automatisch eingelesen.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **Settings.h**
- Settings.cc

B.12 VoiceAnchor Klassenreferenz

`#include <VoiceAnchor.h>`

Private Attribute

- **ProbPos ppPredAng**

  Vorhergesagte Position.

- **bool bPredAng**

  existiert vorhergesagte Position.

- **PPRateBhatta RateAng**

  Bewertungs-Modell.

- **PPMoveDisp MoveAng**

  Bewegungs-Modell.

- **PPFuseCp FuseAng**

  Fusions-Modell.

Ausführliche Beschreibung

Die *VoiceAnchor*-Klasse erweitert die *Anchor*-Klasse und beinhaltet Funktionen zur Berechnung, in wie weit ein neues Perzept zu den vorrausberechneten Daten einer Personenkomponente passt. Es werden eingehende Voice-Positionsdaten mit dem erwarteten Wert für eine Person verglichen und bewertet, je nachdem ob ein eingehendes Datum eine neue *Voice*-Position desselben Symbols sein kann, oder außer Reichweite liegt und einem anderen oder neuen *Voice*-Symbol zugeordnet werden muss.

Beschreibung der Konstruktoren und Destruktoren

VoiceAnchor::VoiceAnchor (*CompositeAnchor * ca*)
Konstruktor, zur Zuornung wird ein Zeiger auf den übergeordneten *Composite-Anchor* (S. 66) übergeben

VoiceAnchor::~VoiceAnchor ()
Destruktor

Dokumentation der Elementfunktionen

bool VoiceAnchor::bGetVoiceAng (float & *fAng*)
Zugriff auf Positionswerte.
Winkeldaten für diesen *Voice-Anchor* als Referenzen übergeben

void VoiceAnchor::vCompModelMatch (PercAttrList & *PAttrList*) `[private]`
Nach Composition-Model zu Personen zusammenfügen.
Die Bewertung ungeeigneter Perzepte wird auf 0.0 gesetzt.

void VoiceAnchor::vComponentMatch (PercAttrList & *PAttrList*) `[private]`
Platzhalter, in Anchor.h virtual deklariert (siehe auch Anhang A).

void VoiceAnchor::vCompositeMatch (PercAttrList & *PAttrList*) `[private]`
Platzhalter, in Anchor.h virtual deklariert (siehe auch Anhang A).

void VoiceAnchor::vOneStepPredict () `[private]`
Vorhersage über Voicedaten für den nächsten Zeitschritt mittels *Component-Anchor*, wenn im Status *Track*

void VoiceAnchor::vPredict () `[private]`
Vorhersage über Voicedaten für den nächsten Zeitschritt mittels *Composite-Anchor* und Personenmodell, wenn im Status *Reacquire*

void VoiceAnchor::vPrintInfo ()
Status textuell ausgeben.

void VoiceAnchor::vRate (PercAttrList & *PAttrList*) `[private]`
Bewerten der Perzepte.
Voice-Perzepte werden bewertet (mit Werten aus [0,1]). Perzepte, die schon für ungeeignet erklärt wurden, haben eine Bewertung von 0.0 und müssen daher nicht mehr bewertet werden. Alle anderen Perzepte haben initial eine Bewertung von 1.0 (optimistische Initialisierung).

void VoiceAnchor::vResetAnchor ()
Anchor zurücksetzen

void VoiceAnchor::vUpdate (PercAttr & *PAttr*) `[private]`
Aktualisierung des Anchors mit ausgewähltem Perzept

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **VoiceAnchor.h**
- VoiceAnchor.cc

B.13 VoicePercAttr Klassenreferenz

`#include <PercAttr.h>`

Öffentliche Attribute

- **ProbPos ppAng**

  Winkel.

Ausführliche Beschreibung

Die Klasse erweitert die *PercAttr*-Klasse und stellt die oben genannten öffentlichen Attribute zur Speicherung von Perzeptdaten zur Verfügung, die von dem Programm *sploc* geliefert werden.

Beschreibung der Konstruktoren und Destruktoren

VoicePercAttr::VoicePercAttr (float *fVoiceAng*, long *lTime*)
Die Parameter sind:
Winkel (in Rad) einer auftretenden Stimme, Timestamp in mSek. seit 1970

Dokumentation der Elementfunktionen

void VoicePercAttr::vPrintInfo () `[virtual]`
Perzeptinformationen ausgeben
Die Funktion verdeckt die Implementation aus *PercAttr*.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **PercAttr.h**
- PercAttr.cc

B.14 VoicePercAttrList Klassenreferenz

`#include <PercAttr.h>`

Ausführliche Beschreibung

Diese Klasse erweitert die *PercAttrList*-Klasse und stellt eine Liste aus den über den Konstruktor erhaltenen Daten zusammen, deren Elemente Instanzen der *VoicePercAttr*-Klasse sind.

Beschreibung der Konstruktoren und Destruktoren

VoicePercAttrList::VoicePercAttrList (float * *afVoiceData*, int *iNumOfVoices*, long *lSec*, long *lUsec*)
Die Parameter sind:
Pointer auf Array von *Voice*-Daten, Anzahl der Arrayelemente,
Timestamp in Sek. seit 1970, restliche Mikrosekunden

Dokumentation der Elementfunktionen

void VoicePercAttrList::vPrintInfo () `[virtual]`
Perzeptinformationen ausgeben
Die Funktion verdeckt die Implementation aus *PercAttrList*.

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- **PercAttr.h**
- PercAttr.cc

B.15 VoiceSignature Klassenreferenz

`#include <VoiceAnchor.h>`

Öffentliche Attribute

- **ProbPos ppVoiceAng**

  Spezielle Anchordaten.

Ausführliche Beschreibung

Die Klasse beinhaltet neben dem Kon- und dem Destruktur nur das oben genannte öffentliche Attribut. Eine Instanz dieser Klasse wird beim update() an den *Composite-Anchor* (S. 66) uebergeben.

Beschreibung der Konstruktoren und Destruktoren

VoiceSignature::VoiceSignature () `[inline]` Konstruktor

VoiceSignature::~VoiceSignature () `[inline]` Destruktor

Dateiinformationen

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Datei:

- **VoiceAnchor.h**
- VoiceAnchor.cc

(Thorsten Spexard)

Literatur

[1] Silvia Coradeschi, Alessandro Saffiotti: *Perceptual Anchoring of Symbols for Action*, Proc. of the 17th Intl. Joint Conf. On Artificial Intelligence (IJCAI-01), Seattle, WA, 2001, pp.407-412

[2] G. Escudero, L. Marquez, and G. Rigau: *Boosting Applied to Word Sense Disambiguation*, To appear in Proceedings of the 12th European Conference on Machine Learning, Barcelona, Spain, 2000

[3] Y. Freund, R. E. Schapire: *A short introduction to boosting*, Journal of Japanese Society for Artificial Intelligence, 14(5):771–780, September 1999

[4] J. Fritsch, M. Kleinehagenbrock, S. Lang, T. Ploetz, G. A. Fink, G. Sagerer: *Multi-Modal Anchoring for Human-Robot-Interaktion. Robotics and Autonomous Systems, Special issue on Anchoring Symbols to Sensor Data in Single and Multiple Robot Systems*, 2003, pp. 133-147

[5] D. Giuliani, M. Omologo, P. Svaizer: *Talker localization and speech recognition using a micrphone array and cross-powerspectrum phase analysis*, ICSLP 1994, pp. 1243 - 1246

[6] M. Kleinehagenbrock, S. Lang, J. Fritsch, F. Lömker, G. A. Fink, G. Sagerer: *Person Tracking with a Mobile Robot based on Multi-Modal Anchoring*, EEE Int. Workshop on Robot and Human Interaktive Communication (ROMAN) 2002

[7] S. Lang, M. Kleinehagenbrock, J. Fritsch, G. A. Fink, G. Sagerer: *Detection of Communication Partners from amobile Robot*, 4th Workshop on Dynamic Perception, Bochum (Germany) 2002, pp. 183-188

[8] Rainer Lienhart, Jochen Maydt: *An Extended Set of Haar-like Features for Rapid Object Detection*, Intel Labs, Intel Corporation, Santa Clara, CA 95052, USA

[9] Yosuke Matsusaka, Shinya Fujie, Tetsunori Kobayashi: *Modeling of Conversational Strategy for the Robot Participating in the Group Conversation*, Proc. European Conf. on Speech Communication and Technology 2001, Aalborg (Denmark), pp. 2173-2176

[10] R. Meir, G. Rätsch: *An introduction to boosting and leveraging*, Advanced Lectures on Machine Learning, LNCS, pages 119–184, Springer, 2003

[11] Hiroshi G. Okuno, Kazuhiro Nakadai und Hiroaki Kitano: *Social Interaction of Humanoid Robot Based on Audio-Visual Tracking*, Computer Science 2002, p. 725

[12] M. Omologo, P. Svaizer: *Acoustic Event Loclalization using a Crosspower-Spectrum Phase based Technique*, Pro. ICASSP , Adelaide 1994, pp. II273 - II 276

[13] M. Omologo, P. Svaizer: *Talker localization and enhancement in a noisy enviroment using a micrphone array based acquisition system*, IRST-Istituto per la Ricerca Scientifica e Tecnologica, Povo di Trento (Italy), pp. 605 - 608

[14] R. E. Schapire.: *A brief introduction to boosting*, In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999

[15] Robert E. Schapire: *The Boosting Approach to Machine Learning An Overview*, MSRI Workshop on Nonlinear Estimation and Classification, 2002

[16] Matthew Turk, Alex Pentland: *Eigenfaces for Recognition*, Journal of Cognitive Neuroscience Volumne 3, Number 1, 1991 Massachuettes Institute of Technology

[17] Paul Viola, Micheal Jones: *Fast and Robust Classification using Asymmetric AdaBoost and a Detector Cascade*, Neural Information Processing Systems 14, December 2001

[18] Paul Viola, Micheal Jones: *Rapid Object Detection using Boosted Cascade of Simple Features*, Conference on Computer Vision and Pattern Recognition, 2001

[19] Paul Viola, Micheal Jones: *Robust Real-Time Object Detection*, Second International Workshop on Statistical and Computational Theories of Vision - Modeling, Learning, Computing, and Sampling, Vancouver, Canada, July 13, 2001

[20] Stan Zhang, Li Long Zhu, ZhenQiu Zhang, HongJiang Zhang: *Learning to Detect Multi-View Faces in Real-Time*, Mircosoft Research Asia, Bejing Sigma Center, Bejing 100080, China

[21] ZhenQui Zhang, Long Zhu, Stan Z. Li, HongJiang Zhang: *Real-Time Multi-View Face Detection*, Mircosoft Research Asia, Bejing Sigma Center, Bejing 100080, Chi

[22] CMU frontal faces test set: http://www-2.cs.cmu.edu/∼har/faces.htmlna