Infrastruktur zur Evaluierung konkurrierender Algorithmen zur Anwendung im RoboCup

Infrastructure for evaluation of competitive algorithms for application in RoboCup

Master-Thesis von Armin Berres

Oktober 2009

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Fachgebiet Simulation,
Systemoptimierung und Robotik

Infrastruktur zur Evaluierung konkurrierender Algorithmen zur Anwendung im RoboCup
Infrastructure for evaluating competitive algorithms for application in RoboCup

vorgelegte Master-Thesis von Armin Berres

Aufgabensteller:	Prof. Dr. Oskar von Stryk
Betreuer:	Dipl.-Inform. Dirk Thomas
	Dipl.-Inform. Sebastian Petters
Tag der Einreichung:	31. 10. 2009

Darmstadt, 31. Oktober 2009

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Armin Berres

Zusammenfassung

Das Fachgebiet Simulation, Systemoptimierung und Robotik (SIM) der Technischen Universität Darmstadt beschäftigt sich mit der Forschung an mobilen autonomen Robotersystemen. Eine besondere Bedeutung haben hierbei robuste und fehlerfreie Algorithmen.

Im Rahmen dieser Arbeit wurde eine Infrastruktur zur computergestützten Auswertung beliebiger Algorithmen implementiert. Besondere Rücksicht wurde hierbei auf die Evaluierung von Bildverarbeitungsalgorithmen für den Einsatz auf humanoiden Robotern im RoboCup gelegt. Zur Unterstützung der Evaluierung wurden die Grundlagen einer weitgehend kalibrierungsfreien Bildverarbeitung geschaffen.

Abstract

The Simulation, Systems Optimization and Robotics Group (SIM) of the Department of Computer Science of the Technische Universität Darmstadt concerns itself with the research of mobile autonomous robotic systems. In this area of research special emphasis is given to robust and accurate algorithms.

In the course of this thesis an infrastructure for evaluating arbitrary algorithms was implemented. Particular attention was placed on the evaluation of image processing algorithms for the application on humanoid robots in RoboCup. To support this the foundations of a largely calibration-free image processing algorithm have been created.

Inhaltsverzeichnis

Glossar

BSD-Lizenz *Berkeley Software Distribution*-Lizenz – an der University of California, Berkeley entworfene Lizenz für freie Software. Sie erlaubt es eine Software zu kopieren, zu verändern und zu verbreiten, solange der Copyright-Vermerk nicht entfernt wird.

Chrominanz Farbsignal eines Bildpunkts – zwei voneinander unabhängige Farbsignale können die vollständige Farbinformation eines Bildpunkts speichern.

CRT Cathode Ray Tube – Kathodenstrahlröhre, wird beispielsweise in Fernsehgeräten oder Computermonitoren zur Bilderzeugung eingesetzt

GUI Graphical User Interface – Grafische Benutzeroberfläche

Hough-Transformation Verfahren zur Erkennung von Geraden, Kreisen oder beliebigen anderen parametrisierbaren geometrischen Figuren basierend auf einem Gradientenbild nach Kantenerkennung

kinematische Kette System aus starren Körpern, die durch Gelenke verbunden sind

LCD Liquid Crystal Display – Flüssigkristallbildschirm

LED Light Emitting Diode – Leuchtdiode

Lookuptabelle Datenstruktur, die vorberechnete Daten eines Algorithmus' enthält

Luminanz Helligkeit eines Bildpunkts

PC/104 Standard zur Entwicklung von Hardware für eingebettete Systeme, welcher sowohl einen Formfaktor als auch ein Bussystem spezifiziert

RoboCup jährliche Weltmeisterschaft im autonomen Roboterfußball: `http://www.robocup.com`

SIFT Scale-invariant feature transform – Algorithmus zum Erkennen und Vergleichen lokaler Bildmerkmale

UML	Unified Modeling Language (Vereinheitlichte Modellierungssprache) – standardisierte Sprache mit grafischer Notation zur Modellierung von Software und anderen Systemen
WLAN	Wireless Local Area Network – drahtloses lokales Netzwerk, lokales Funknetz

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Ziel der Arbeit

Ziel der Arbeit ist es, eine Infrastruktur zum Vergleich und der Bewertung verschiedener im RoboCup zum Einsatz kommender Algorithmen zu schaffen. Besonders berücksichtigt wird hierbei der Einsatz zur Untersuchung verschiedener Bildverarbeitungsalgorithmen.

Zur Bewertung und Weiterentwicklung der aktuellen Bilderverarbeitung des RoboCup-Teams der TU Darmstadt, *Darmstadt Dribblers* [1, 2], werden die Grundlagen einer Referenzbildverarbeitung mit anderen und rechenzeitaufwendigeren Konzepten erstellt. Neue und bisherige Bildverarbeitung werden mithilfe der geschaffenen Infrastruktur miteinander verglichen.

1.2 Aufbau

Diese Arbeit erläutert Grundlagen, den aktuellen Stand der Forschung, getroffene Designentscheidungen und die genaue Umsetzung sowie erzielte Ergebnisse.

Zusätzlich dient sie als Dokumentation für die Nutzung der implementierten Infrastruktur zur Algorithmenevaluierung. Hierzu empfiehlt sich besonders die Lektüre der ersten Teile der Kapitel 5 und 6.

Die vorliegenden Arbeit gliedert sich wie folgt:

- **Kapitel 2** Überblick der Motivation dieser Arbeit
- **Kapitel 3** Aktueller Stand der Forschung und Grundlagen
- **Kapitel 4** Anforderungen an die zu schreibenden Softwarekomponenten
- **Kapitel 5** Vorstellung des Konzepts der Infrastruktur und der zu schreibenden Bildverarbeitung
- **Kapitel 6** Beschreibung der Umsetzung des im vorherigen Kapitel erarbeiteten Konzepts und detaillierte Darstellung der entwickelten Softwarekomponenten
- **Kapitel 7** Darstellung der in dieser Arbeit erzielten Ergebnisse

- **Kapitel 8** Zusammenfassung der Arbeit und Ausblick auf mögliche Weiterentwicklungen

1.3 Quellcode

Der Quellcode dieser Arbeit findet sich im aktuellen Repository der *Darmstadt Dribblers*. Er liegt nicht auf einer separaten CD bei, da der Code nicht aus einem einzeln nutzbaren Teil besteht, sondern sich über mehrere Stellen verteilt. Teile des Codes bestehen zusätzlich aus Änderungen von bestehendem Code. In der Arbeit wird darauf eingegangen, an welchen Stellen Code neu entwickelt und an welchen Stellen bestehender Code weiter entwickelt wurde.

Kapitel 2

Motivation

Im RoboCup kommen verschiedenste Algorithmen zum Einsatz. Probleme können mit unterschiedlichen konkurrierenden Algorithmen oder dem gleichen Algorithmus mit abweichenden Parametersätzen gelöst werden. Des Weiteren liefern viele Algorithmen keine Ergebnisse, die mathematisch verifizierbar sind. Ohne eine Möglichkeit, die Ergebnisse der verschiedenen Algorithmen vergleichen zu können, kann nicht objektiv bewertet werden, ob Änderungen oder andere Ansätze zu einer Verbesserung oder Verschlechterung führen.

Wünschenswert wäre es, die Möglichkeit zu haben, die Ergebnisse verschiedener Algorithmen zu speichern und später auswerten zu können. Der Nutzer sollte hierbei vom System unterstützt werden und sich beispielsweise nur die Datensätze anzeigen lassen können, bei denen Algorithmen zu signifikant verschiedenen Ergebnissen kommen.

Eine der essenziellen Komponenten im RoboCup ist die Bildverarbeitung. Ohne dass ein Roboter seine Umgebung korrekt erkennt, kann er nicht mit dieser interagieren. Da die Rechenleistung der Roboter aufgrund verschiedener Faktoren, wie begrenzter Akkukapazität und möglichst geringem Gewicht, begrenzt ist, müssen gerade die Algorithmen der Bildverarbeitung sehr effizient programmiert werden. Die Beschränkung der verfügbaren Rechenleistung führt dazu, dass viele in der Forschung bekannten Bildverarbeitungsalgorithmen nicht angewendet werden können. Mit mehr Rechenleistung können bessere Algorithmen genutzt werden, die wiederum zu besseren Ergebnissen führen.

Um einerseits für zukünftige Entwicklungen gerüstet zu sein und gleichzeitig die bestehende Bildverarbeitung bewerten und verbessern zu können, ist es wünschenswert, neben der Bildverarbeitung, die auf den Robotern läuft, eine Bildverarbeitung zu schreiben, die nicht durch die Rechenzeit begrenzt ist und nicht in Echtzeit auf den Robotern arbeiten können muss.

Die Motivation dieser Arbeit ist es, genau diese zwei Dinge zur Verfügung zu stellen.

- Eine Infrastruktur um konkurrierende Algorithmen und ihre Ergebnisse effizient vergleichen zu können
- und zusätzlich die Grundlagen einer alternativen Bildverarbeitung zu schaffen.

Kapitel 3

Grundlagen

3.1 Stand der Forschung Evaluierung

Die im RoboCup entwickelten und genutzten Anwendungen unterscheiden sich von denen in vielen anderen Anwendungsgebieten. Bei der Entwicklung eines Roboters kommen verschiedenste komplett unterschiedliche Algorithmen zum Einsatz. Um einen Roboter nicht nur als Blackbox zu betrachten und aus seinem Verhalten auf die Ergebnisse der Algorithmen schließen zu können, ist es erforderlich, die Ergebnisse und idealerweise auch internen Zustände der genutzten Algorithmen auswerten zu können. Hierfür können Standardverfahren nicht direkt verwendet werden, sondern es müssen je nach Algorithmus geeignete Tools entwickelt werden. Die Teams im RoboCup haben hierfür modulare Softwareplattformen entwickelt.

Im Folgenden werden die Entwicklungen zweier Teams vorgestellt.

3.1.1 *German Team*

Das *German Team*, ein Zusammenschluss der Universitäten Humboldt Universität zu Berlin, Universität Bremen, Technische Universität Dortmund und Technische Universität Darmstadt, nahm von 2001 bis 2008 an der RoboCup Weltmeisterschaft in der *Sony Four-Legged Robot League* teil. In dieser Liga spielten bis 2008 vierbeinige, hundeähnliche Roboter von Sony, die sogenannten AIBOs, im Team gegeneinander Fußball (Abbildung 3.19).

Das *German Team* hat über mehrere Jahre mit *RobotControl* (Abbildung 3.1) ein Debuggingtool entwickelt, um mit den AIBOs zu arbeiten [3]. Über WLAN kann sich mit bis zu acht Robotern verbunden werden, um deren internen Zustand anzuzeigen und grafisch darzustellen. Durch den modularen Aufbau ist es möglich, das Tool um beliebige Elemente zum getrennten Testen und Überwachen einzelner Module zu erweitern. Daten von den Robotern können aufgenommen und später erneut abgespielt und ausgewertet werden. Neben Debuggingmöglichkeiten erlaubt es *RobotControl* ebenso Farbkalibrierung oder Laufoptimierung durchzuführen.

Neben *RobotControl* steht mit *SimRobot* eine Anwendung zur Verfügung, um bis zu acht Roboter zu simulieren und Algorithmen offline, ohne tatsächliche Hardware zu nutzen, testen zu können.

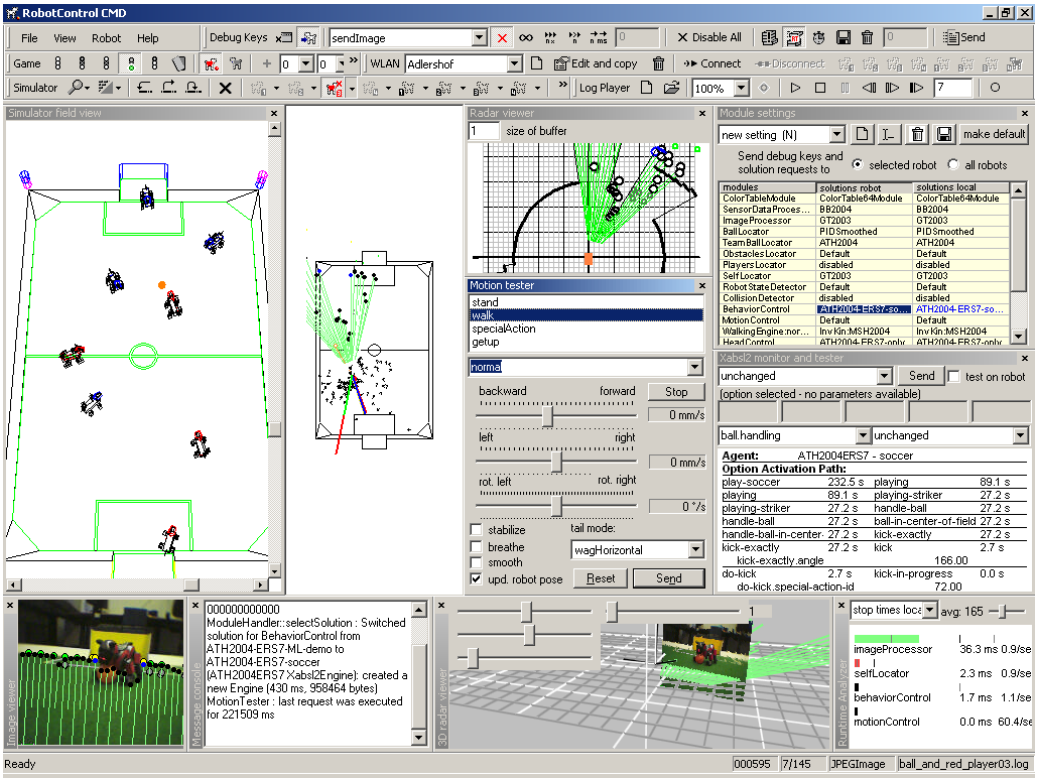


Abbildung 3.1: Screenshot des vom *German Team* genutzten Debuggingtools *RobotControl* (Quelle [3]).

3.1.2 *Darmstadt Dribblers*

Die *Darmstadt Dribblers* treten seit 2004 in der RoboCup Kid-Size-Humanoidliga an. Für eine Übersicht der derzeit verwendeten Hardware siehe 3.5.1.

Seit 2005 baut die zum Einsatz kommende Software auf dem Robotersteuerungsframework *RoboFrame* auf (siehe 3.4). Passend zu jeder geschriebenen Anwendung kann mit *RoboGui* eine grafische Benutzeroberfläche, ähnlich wie *RobotControl*, geschrieben werden (Abbildung 3.2). Alle verwendeten Module, wie Bildverarbeitung, Selbstlokalisierung, Verhalten oder Weltmodellierung, können über diese untersucht und kontrolliert werden. Es ist möglich, sich von Roboter die aufgenommenen Bilder und erkannte Objekte schicken und anzeigen zu lassen. Hierbei ist es auch möglich, die Schritte der Bildverarbeitungsalgorithmen durch das Einzeichnen weiterer Informationen ins Bild, sogenannte Drawings, zu verdeutlichen. Das aktuell ausgeführte Verhalten kann angezeigt und manipuliert werden. Der Roboter speichert intern eine Repräsentation der wahrgenommenen Umwelt, genannt Weltmodell. Die Weltmodelle eines oder mehrerer Roboter können grafisch dargestellt und bewertet werden. Des Weiteren können verschiedene Statusinformationen vom Roboter angefordert und Parametersätze auf dem Roboter geändert werden.

Auf dem Roboter erzeugte Daten können entweder an die Gui gesendet und dort gespeichert oder direkt vom Roboter auf eine CF-Karte gespeichert werden. Diese Daten können

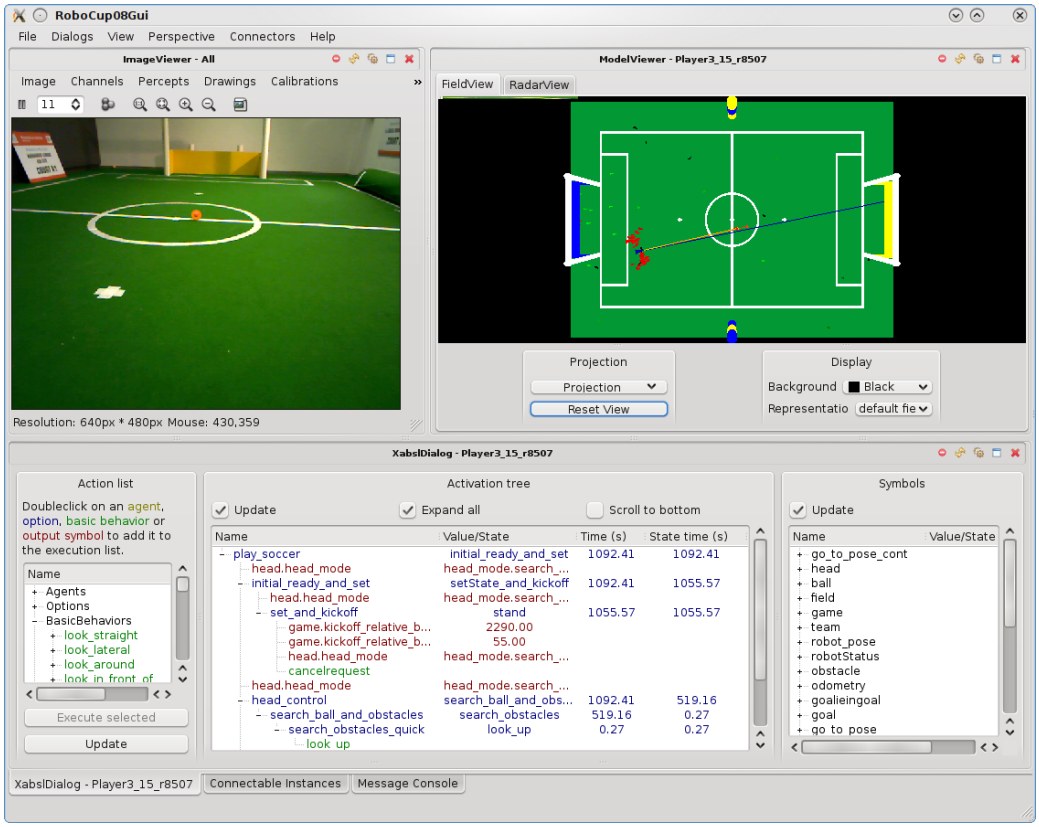


Abbildung 3.2: Screenshot der von den *Darmstadt Dribblers* genutzten GUI. Zu sehen sind die Module zum Anzeigen von Bildern, dem Debuggen und Steuern des Verhaltens und dem Anzeigen der Weltmodellierung.

offline erneut abgespielt und analysiert werden. Die Gui erlaubt zum Beispiel die gesammelten Daten mehrerer Roboter und eines Videostroms zu synchronisieren, um im Nachhinein den internen Status der Roboter auswerten und über das Kamerabild mit der Wirklichkeit vergleichen zu können [4].

Genau wie vom *German Team* wurde auch von den *Darmstadt Dribblers* ein Simulator, basierend auf dem entwickelten Framework Multi-Robot-Simulation-Framework (MuRoSimF) [5], entwickelt. Er wird eingesetzt, um offline Algorithmen testen zu können. Der Simulator ermöglicht es, sowohl die Firmware der Roboter als auch die Applikation, die auf den Robotern läuft, gegen simulierte Hardware und ein simuliertes Spielfeld zu testen.

3.1.3 Auswertung und Bewertung empirischer Daten

Sollen gesammelte Daten oder die Ergebnisse von Algorithmen ausgewertet und verglichen werden, empfiehlt sich nicht nur manuelle Auswertung, sondern auch die Unterstützung durch statistische Methoden [6].

Nach Cohen bieten sich bei der Untersuchung von Daten mit einer Variable insbesondere

folgende Kennziffern an: Anzahl der Messwerte, Durchschnitt der Messwerte, Median, Minimum, Maximum, Abstand zwischen Maximum und Minimum, interquartil Abstände oder Standardabweichung beziehungsweise Varianz. Soll die Abhängigkeit zweier linearer Variablen geprüft werden, bietet sich die Kovarianz an. Ist sie groß und positiv, liegt ein gleichsinnig linearer Zusammenhang vor. Größere Werte einer Variable führen also zu größeren Werten der anderen Variable und umgekehrt. Eine große negative Kovarianz spricht für einen gegensinnigen linearen Zusammenhang. Ist die Kovarianz nahe null, sind die beiden Zufallsvariablen nicht linear abhängig.

Hat man einen Datensatz aufgenommen, ist es oft nötig, die Werte zu glätten, um die Daten besser interpretieren zu können. Die Idee dahinter ist, dass die Nachbarn eines Wertes Informationen darüber enthalten, wie der wirkliche Wert an dieser Position aussehen müsste. Eine oft genutzte Technik ist, einen Wert durch den Median oder Durchschnitt aus sich selbst und dem linken und rechten Nachbarn zu ersetzen.

Hat man anhand eines Datensatzes eine Hypothese über seine Verteilung aufgestellt, muss diese Hypothese statistisch überprüft werden. Hierfür bietet die Statistik verschiedenste klassische Tests wie den Z- oder t-Test. Mit dem Aufkommen leistungsfähiger Computer wurden Methoden entwickelt, die darauf basieren, die Stichprobenentnahme zu simulieren. Beispiele sind Monte-Carlo-Methoden, Bootstrapping oder Randomisierungstests.

Liegt eine Menge von Messwerten vor, durch die eine Kurve gelegt werden soll, gilt es die Parameter so zu wählen, dass sich die Kurve den Messwerten bestmöglich anpasst. Je nachdem, ob die Daten linear oder nichtlinear verteilt sind, kommen hierfür Methoden wie die lineare oder nichtlineare Regression zum Einsatz.

3.2 Stand der Forschung Bildverarbeitung

Im RoboCup werden einige Standardverfahren zur Bildverarbeitung eingesetzt, die von verschiedenen Teams sehr ähnlich implementiert werden. Neben diesen Standardverfahren gibt es verschiedene Versuche, die Bilderkennung weiter zu optimieren und robuster zu machen.

Auch Forschungsergebnisse aus anderen Bereichen, zum Beispiel der Erkennung von Verkehrszeichen in bewegten Bildern, sind für diese Arbeit interessant. Diese werden weiter unten in 3.3.1.3 besprochen.

3.2.1 Klassischer Ansatz

Die meisten der im RoboCup eingesetzten Bildverarbeitungen basieren auf Variationen des im Folgenden beschrieben Ansatzes.

Farbtabellen

Bei der Objekterkennung im RoboCup spielen Farben eine große Rolle. Alle zu erkennenden Objekte sind eindeutig eingefärbt. Ein wichtiger Bestandteil der Bildverarbeitung ist es, die

absoluten Farbwerte in einem von der Kamera gelieferten Bild den verschiedenen möglichen Farbklassen, zum Beispiel Rot, Grün, Blau oder Gelb, zuzuordnen. Die Zuordnung zwischen einem Pixelwert und einer Farbklasse ist meist statisch und in Form einer Lookup-Tabelle implementiert. Für jeden möglichen Farbwert ist in dieser die zugehörige Farbklasse gespeichert. Bei der weiteren Bearbeitung des Bildes wird meist nur noch ein klassifiziertes Bild mit Farbklassen statt Pixelwerten betrachtet (Abbildung 3.3(a) und (b)). Dieses kann verglichen mit dem Originalbild sehr effizient bearbeitet werden.

Um effizient arbeiten zu können, wird die Anzahl der Einträge in den Farbtabellen beschränkt. Die *Darmstadt Dribblers* arbeiten im Moment mit einer Lookup-Tabelle (LUT) mit einem 16 Bit Index, also maximal $2^{16} = 65536$ Einträgen. Die von der Kamera gelieferten 24 Bit YUV-Pixel[1] können jedoch 2^{24} verschiedene Werte annehmen. Für die Nutzung der Lookup-Tabelle müssen die Pixel auf die 65536 Werte quantifiziert werden. Hierfür müssen von den 24 Bit eines Pixels mindestens 8 Bit verworfen werden. Im Moment nutzen die *Darmstadt Dribblers* folgendes Verfahren: Der Y-Kanal wird auf 6 Bit reduziert, dies entspricht einer Quantifizierung auf 32 Werte.[2] U- und V-Kanal werden auf 5 Bit reduziert und entsprechend auf 16 verschiedene Werte quantifiziert.

Erzeugt werden die Farbtabellen meistens vor dem Starten des Roboters, indem von der Kamera aufgenommene Bilder manuell mit entsprechenden Tools ausgewertet werden. Die statischen Farbtabellen sind die größte Stärke und gleichzeitig auch Schwäche des Ansatzes. Durch das manuelle Erzeugen der Tabellen ist, wenn exakt gearbeitet wird, die Abstimmung auf die jeweiligen Lichtverhältnisse und damit die Erkennungsrate sehr gut. Ändern sich die Lichtbedingungen jedoch nur minimal, kann es passieren, dass die Farbtabelle keine oder eine falsche Klassifizierung liefert. Die Ergebnisse der Bildverarbeitung sind damit unbrauchbar. Hierdurch ist es auch nicht möglich, die Roboter an einem Ort mit unbekannten Lichtbedingungen einzusetzen, ohne vorher eine zeitaufwendige Kalibrierung durchzuführen.

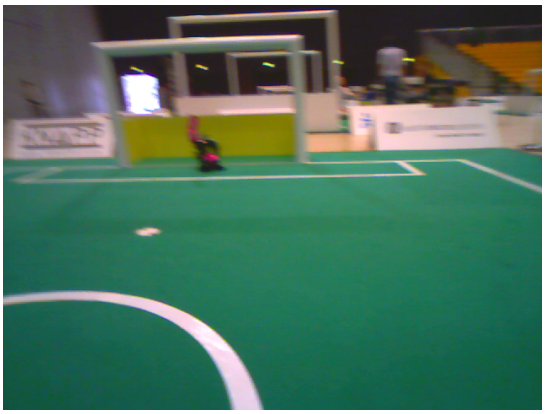
Ansätze, um diese Probleme zu umgehen, werden weiter unten vorgestellt.

Farbklassifikation und Vorverarbeitung

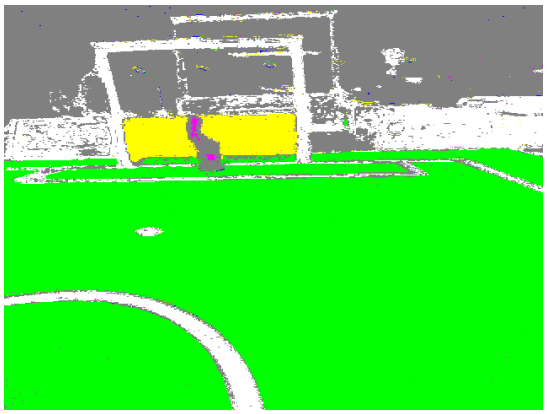
Mit der Farbtabelle werden Teile des Bildes klassifiziert und die Farbpixel Farbklassen zugeordnet. Oft wird hierfür ein Gitter über das Bild gelegt, dessen Linien, die sogenannten Scanlines, bearbeitet werden. Eingesetzt werden im einfachen Fall Gitter aus horizontalen und/oder vertikalen Linien mit fixem Abstand. Die kompliziertere Variante ist, das Gitternetz je nach Blickwinkel des Roboters anzupassen. Schaut der Roboter in seine Nähe, sind die Objekte im Bild größer und die Scanlines können weiter voneinander entfernt generiert werden (Abbildung 3.3(c)).

Der Sinn der Scanlines ist, möglichst wenige Teile des Bildes auswerten zu müssen. Gleichzeitig können aber noch alle Objekte erkannt werden, da das Raster so eng gewählt wird, dass kein Objekt kleiner als das minimale Raster ist.

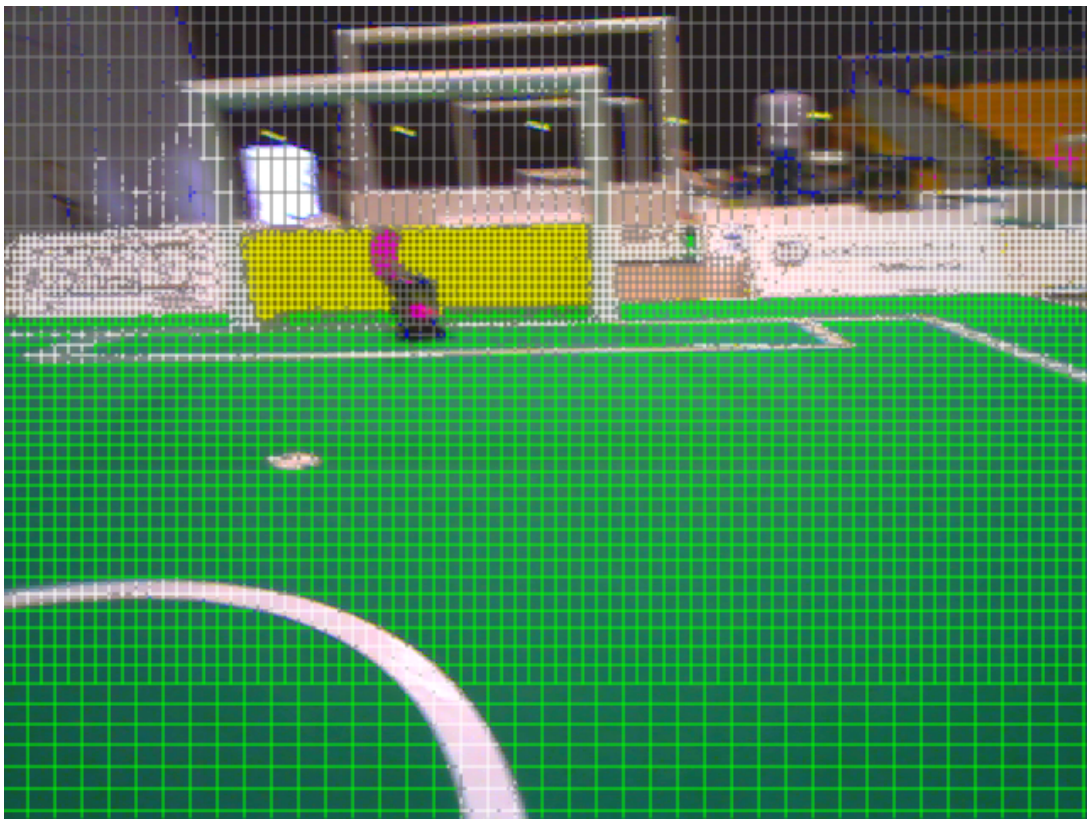
[1] Die Y-, U- und V-Kanäle bestehen jeweils aus 8 Bit und können Werte zwischen 0 und 255 annehmen.
[2] Umgesetzt wird dies durch Entfernen von niederwertigen Bits.



(a) Kamerabild



(b) Klassifiziertes Bild



(c) Kamerabild mit überlagerten Scanlines

Abbildung 3.3: Kamerabild, mit Farbtabelle klassifiziertes Bild und Kamerabild überlagert mit den tatsächlich klassifizierten Scanlines. Die Bilder stammen aus der Bildverarbeitung der *Darmstadt Dribblers*.

Die entstandenen farbklassifizierten Scanlines bestehen nur aus den im RoboCup vorkommenden Farbklassen. Da deren Anzahl sehr gering ist und große gleichfarbige Flächen auftreten, können die Scanlines sehr effektiv RLE-kodiert werden. In manchen Implementierungen werden benachbarte gleichfarbige Teile der Scanlines zu größeren Objekten, genannt Blobs, zusammengefasst.

Objekterkennung

Mithilfe der klassifizierten Scanlines/Blobs wird nach Objektmerkmalen gesucht, die mit den im RoboCup auftretenden Objekten übereinstimmen. Große gelbe oder blaue Objekte sind Tore, kleine runde orange Bälle, lange weiße mit grüner Umgebung Linien und so weiter. Wenn nötig kann hierbei zur genaueren Verifizierung auch auf die Originalfarbwerte oder Pixel außerhalb der Scanlines zugegriffen werden.

Beispiele für Umsetzungen, die auf diesem Ansatz basieren, sind [7–9]. Auch die aktuelle Bildverarbeitung der *Darmstadt Dribblers* baut auf diesem Konzept auf.

Andere Teams arbeiten nicht mit Scanlines, sondern führen direkt auf dem Bild eine Bloberkennung durch.

3.2.2 (Semi-)Automatische Farbkalibrierung

Um die Probleme der Farbtabellen zu lösen, gibt es verschiedene Forschungsarbeiten. Ein Ansatz ist, die Farbtabelle automatisch vom Roboter generieren zu lassen, um so die aufwendige manuelle Kalibrierung obsolet zu machen. Neben der statischen Erzeugung und Initialisierung von Farbtabellen gibt es verschiedene Ansätze, die Farbtabellen während der Laufzeit sich ändernden Lichtbedingungen anzupassen.

Statische Farbtabellen

An der University of Texas at Austin wurde ein System entwickelt, um mit einen Sony AIBO automatisch Farbtabellen zu erstellen [10]. Der Roboterhund wird hierfür auf eine definierte Position auf einem leeren Spielfeld gestellt. Durch die bekannten Positionen der farbigen Objekte und des Roboters relativ zu ihnen kann dieser Bewegungen ausführen, um nach und nach alle Objekte aufzunehmen und ihre Farbinformationen zu speichern.

Auch an der TU Darmstadt wurde ein System zur automatischen Farbklassifizierung entwickelt [11]. Hierbei werden vom Roboter aufgenommene Bilder auf einem externen Computer ausgewertet und aus diesen inkrementell eine Farbtabelle errechnet.

Die automatischen Ansätze haben ein Problem, wegen welchem sie in der Praxis, insbesondere in Wettbewerben, selten eingesetzt werden: Die generierten Farbtabellen sind nicht fehlerfrei und von Hand generierte Farbtabellen erreichen eine höhere Genauigkeit.

Zur automatischen Optimierung bestehender Farbtabellen schlagen Alvarez et al. vor, implizite Oberflächen zu nutzen [12]. Hierbei wird um die Punktwolke, die eine Farbklasse in einer LUT repräsentiert, eine geschlossene Oberfläche gelegt. Alle Punkte innerhalb der Oberfläche werden der Farbklasse zugeschlagen. Daraus resultieren eine bessere Abdeckung

des Farbraums und damit weniger Fehlererkennungen. Werden Farbklassen vergrößert, ohne dass sie sich überlappen, kann auch eine höhere Robustheit gegen leichte Beleuchtungsänderungen erreicht werden.

Stanton und Williams schlagen vor, sich überlappende Farbklassen zu erlauben und je nach Kontext zu entscheiden, welcher Farbklasse ein Pixel wirklich angehört [13]. Das System wird hierdurch robuster gegen Änderungen in der Beleuchtung.

Adaptive Farbtabellen

Um Farbtabellen verschiedenen Lichtbedingungen anpassen zu können, wurden diverse Ansätze vorgestellt. Vier Beispiele folgen: Die University of Texas at Austin schlägt vor, verschiedene Farbtabellen bei verschiedenen Beleuchtungen zu erstellen und dann zur Laufzeit, je nach erkannter Helligkeit, zwischen diesen Farbtabellen zu wechseln [14]. Kukichi et al. erzeugen zweidimensionale Farbtabellen, indem sie im YUV-Farbraum (siehe 3.3.1.2) für jeden Pixel die U- und V-Komponente durch die Helligkeit teilen [15]. Die Cluster der einzelnen Farben in der Farbtabelle werden je nach Beleuchtung um einen Offset verschoben. Jüngel teilt den YUV-Farbraum in Quadrate verschiedener Präzision ein, die adaptiv den Lichtbedingungen angepasst werden.

Allen Ansätzen gemein ist, dass sie gute theoretische Ansätze sind, in der Praxis jedoch noch erhebliche Schwächen haben. Bis heute ist es nicht möglich, in Echtzeit für bewegte Bilder Farbtabellen so zu aktualisieren, dass sie zuverlässig unter verschiedensten Lichtbedingungen funktionieren.

3.2.3 Weiterführende Ansätze

Das Hauptproblem des „klassischen Ansatzes“ ist, dass seine wichtigste Informationsquelle die Farbklassifikation des Bildes ist. Werden Pixel nicht oder falsch klassifiziert, kann dies direkt zu Fehlerkennungen führen. Die Lösung dieses Problems besteht darin, den Einfluss von Farbtabellen zu reduzieren. Es werden weitere Informationen aus dem Bild extrahiert und mit den Informationen des farbklassifizierten Bildes verknüpft.

Murch und Chalup von der University of Newcastle kombinieren Kantenerkennung und Bloberkennung über Farben [16]. Basierend auf einer Farbtabelle wird das Bild zuerst farbklassifiziert und zur Erzeugung farbiger Blobs genutzt. Ausgehend vom Mittelpunkt der erkannten Blobs werden Strahlen projiziert. Auf diesen Strahlen wird nach den nächsten Kanten gesucht, die das farbige Objekt begrenzen. Anhand der Kanten kann die Form des erkannten Objekts bewertet und eine Plausibilitätsprüfung durchgeführt werden. Wird ein Ball gesucht und entspricht die erkannte Form nicht in etwa einem Kreis, muss der Blob verworfen werden.

Lovell wählt einen ähnlichen Ansatz [16]. Neben der optimierten Erkennung ist sein Ziel jedoch, auch die Bilderkennung stabil gegen sich ändernde Lichtbedingungen zu machen. Er setzt hierfür eine Farbtabelle ein, die nur Werte enthält, die unter verschiedensten Lichtbedingungen richtig klassifiziert werden. Um dies zu erreichen, werden Trainingsbilder unter fünf verschiedenen Lichtbedingungen ausgewertet.

3.2.4 Bilderverarbeitung und Objekterkennung außerhalb des RoboCups

Im RoboCup werden aufgrund der beschränkten Rechenleistung der mobilen Systeme relativ simple Verfahren zur Objekterkennung eingesetzt. In vielen anderen Bereichen steht mehr Rechenleistung zur Verfügung, zusätzlich ist die Umgebung nicht wie im RoboCup durch Farbinformationen eindeutig definiert. Teile der folgenden Verfahren werden in RoboCup-Ligen mit größerer Rechenkapazität, wie der Small- oder Mid-Size-Liga, bereits eingesetzt. Ebenso werden sie in dieser Arbeit benutzt.

Zur Objekterkennung basierend auf Form und Kontur werden oft Kantenerkennung (siehe 3.3.7), die darauf basierende Hough-Transformation oder die Minimierung einer Energiefunktion, sogenannten Snakes, eingesetzt [17]. Zur koordinatentransformationsinvarianten und beleuchtungsinvarianten Extraktion lokaler Bildmerkmale wurde SIFT entwickelt [18]. Kombiniert man mehrere Merkmale, lassen sich Objekte eindeutig identifizieren. Über die Eigentransformation können komplette Objekte durch einen Vektor im sogenannten Eigenraum dargestellt werden [19]. Durch den Vergleich zweier Vektoren kann die Ähnlichkeit von Objekten mit einem Referenzobjekt festgestellt werden. Einen ähnlichen Ansatz verfolgen histogrammbasierte Verfahren. Objekte werden nicht durch ihre Form, sondern durch Eigenschaften wie ihre Farbinformationen oder Ableitungen, identifiziert (Beispiel: [20]). Diese werden in Histogramme quantifiziert und auf Übereinstimmung verglichen.

Alle genannten Verfahren können mit Lernverfahren gekoppelt werden, um aus Referenzbildern automatisch Featurevektoren zu extrahieren und zur Klassifikation unbekannter Bilder zu nutzen.

Die oben genannten Verfahren sind nur ein kleiner Ausschnitt, der in verschiedenen Bereichen eingesetzten Bildverarbeitungstechniken.

3.3 Grundlagen Bildverarbeitung

In diesem Abschnitt werden Grundlagen der digitalen Bildverarbeitung erläutert. Sie werden für die Umsetzung der neuen Bildverarbeitung benötigt.

3.3.1 Farbmodelle

Sowohl in der Bildverarbeitung als auch in vielen anderen Anwendungsgebieten[3] spielt es eine wichtige Rolle, Farben eindeutig erkennen und reproduzieren zu können. Durch Farbmodelle ist es möglich, Farben anhand verschiedener Attribute eindeutig zu identifizieren. Die Attribute werden im Normalfall in Form von drei- oder vierdimensionalen Zahlentupeln gespeichert. Durch ein Farbmodell wird ein Raum aufgespannt, der alle theoretisch möglichen Farben enthält, der sogenannte Farbkörper. Jeder Punkt im Farbkörper entspricht einem eindeutigen Tupel. Nicht alle Farben des Farbkörpers können auch tatsächlich dargestellt

[3] Beispiel: Konzernfarben wie das Telekom Magenta oder Drucktechnik im Allgemeinen

werden. Je nach Farbmodell und farbgebendem System kann nur ein Teil des Farbkörpers tatsächlich dargestellt werden. Dieser Unterraum ist der sogenannte Farbraum. Jedes Farbmodell hat unterschiedliche Eigenschaften, die es je nach Einsatzgebiet mehr oder weniger gut nutzbar machen.

Die heutigen Farbmodelle basieren auf den Gesetzen von Graßmann [21]. Diese besagen unter anderem, dass aus drei linear unabhängigen Farben alle anderen Farben gemischt werden können. Erreicht wird dies, indem die Intensität der drei Mischfarben entsprechend angepasst wird, bis die gewünschte Zielfarbe erreicht wird.

Im Folgenden werden drei verschiedene Farbmodelle vorgestellt, die für diese Arbeit relevant sind.

3.3.1.1 RGB

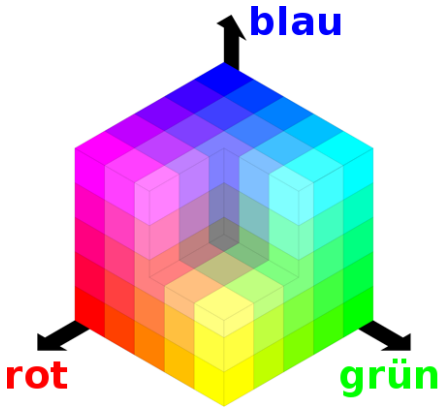


Abbildung 3.4: Darstellung des RGB-Farbraums als Würfel. Die Ecken entsprechen den additiven Primärfarben Rot, Grün und Blau und den subtraktiven Primärfarben Magenta, Cyan und Gelb. Die nicht sichtbare hintere Ecke entspricht Schwarz, die fehlende Ecke Weiß. Die Mitte des Würfels ist grau. Basierend auf `http://www.flickr.com/photos/7702002@N08/3103830956`.

Das RGB-Farbmodell gehört zu den linearen additiven Farbmodellen. Bei diesen werden drei Primärfarben gemischt, um die jeweilige Zielfarbe zu erreichen. Es ist nicht möglich alle Farben darzustellen, da für manche Farben negative Anteile einer Primärfarbe benötigt werden.

Beim RGB-Farbmodell werden die Primärfarben Rot, Grün und Blau genutzt (Abbildung 3.4). Es wird sehr oft zum Anzeigen und Austauschen von Bildern im Computerbereich eingesetzt. Der Grund ist vor allem, dass die ersten Farbmonitore zur Bilddarstellung Farbpixel aus leuchtenden roten, grünen und blauen Phosphorpunkten zusammensetzten. Das Mischen von Farben aus Rot, Grün und Blau wird auch heute in den meisten CRT-, LCD- oder LED-Bildschirmen, vom Fernseher über den Computer bis zum Handy, genutzt. Viele Aufnahmegeräte funktionieren analog und haben Sensoren für die Farben Rot, Grün und Blau.

Zur Darstellung eines Bildes im RGB-Farbraum müssen die einzelnen Pixel entsprechend umgesetzt werden, um die roten, grünen und blauen Bildgeber zu aktivieren. Durch unterschiedliche farbgebende Elemente sehen Bilder auf unterschiedlichen Ausgabegeräten verschieden aus. Durch den RGB-Farbraum selbst ist also nicht festgelegt, wie genau Farben dargestellt werden. Hierfür werden Farbprofile benötigt, welche die Darstellung für jedes Ausgabegerät anpassen, sodass überall das gleiche Ergebnis erzielt wird.

3.3.1.2 YUV

Der Ursprung des YUV-Farbmodells ist die Einführung des Farbfernsehens. Für die übertragenen s/w-Bilder reichte ein einziger Helligkeitskanal mit der Luminanz Y aus. Um alte Fernseher weiter betreiben zu können, wurden neben diesem Farbkanal zwei Farbkanäle U und V aufgeschaltet, um die Chrominanz zu übertragen (Abbildung 3.5).

Dieses Farbmodell wird heutzutage noch häufig angewendet. Durch die Trennung von Helligkeits- und Farbinformationen ist, basierend auf der menschlichen Wahrnehmung eine Reduktion der Datenrate möglich. Menschen nehmen Unterschiede in der Helligkeitskomponente stärker wahr als in der Farbkomponente. Die Farbinformationen können also gegenüber den Helligkeitsinformationen stärker reduziert werden, ohne dass sichtbare Artefakte im Bild auftreten. Ein Beispiel der sogenannten Farbunterabtastung ist YUV422 [22]. Hierbei wird für jedes Pixel ein Y-Wert übertragen, zwei Pixel teilen sich jedoch immer einen U- und einen V-Wert. Die Datenrate wird um ein Drittel reduziert. Im Rahmen dieser Arbeit ist dieses Farbmodell interessant, da die viele Webcams Bilder im YUV-Format liefern und die aktuell eingesetzte Bildverarbeitung der DD200x (siehe 3.5.1) direkt mit diesem Format arbeitet.

RGB-Farbwerte können folgendermaßen in YUV-Farbwerte umgerechnet werden:[4] Die Luminanz Y ergibt sich aufgrund der menschlichen Wahrnehmung aus unterschiedlich gewichteten Anteilen von Rot, Grün und Blau. Mit der Differenz zwischen Y und dem Blau- beziehungsweise Rotanteil und Korrekturfaktoren lassen sich U und V berechnen.

$$
\begin{aligned}
Y &= 0.299R + 0.587G + 0.144B \\
U &= 0.493(B - Y) \\
V &= 0.877(R - Y)
\end{aligned}
$$

Analog ergibt sich die Umwandlung von YUV nach RGB.

$$
\begin{aligned}
R &= Y + V/0.877 \\
G &= 1.7Y - 0.509R - 0.194B \\
B &= Y + U/0.493
\end{aligned}
$$

[4] Die folgenden Gleichungen gehen von zwischen null und eins normalisierten Eingabewerten aus.

3.3.1.3 HSV

Die beiden linearen Farbmodelle RGB und YUV basieren beide auf verschiedenen Eigenschaften genutzter Hardware. Das HSV-Modell dagegen beruht auf Eigenschaften der menschlichen Wahrnehmung. Farben werden durch ihren Farbton, ihre Reinheit oder Sättigung und durch ihre Helligkeit repräsentiert. Erste Anwendungen fand das HSV-Modell bei der Auswahl von Farben in Zeichenprogrammen. Steht nur das RGB-Modell zur Verfügung, muss der Nutzer so lange die Werte R, G und B ändern, bis die gewünschte Farbe gemischt ist. Dieses Schema ist nicht intuitiv und fehleranfällig. Statt dessen wird eine Methode angewendet, die Künstler beim Mischen von Farben auf einer Palette nutzen: Eine Grundfarbe wird zuerst mit Weiß gemischt, bis die gewünschte Farbschattierung erreicht ist. Danach wird je nach gewünschter Helligkeit schwarze Farbe zugemischt. Im HSV-Modell entspricht dies der Auswahl des Farbtons H, gefolgt von der Sättigung S und der Helligkeit V [23].

Der Farbton H ist als Winkel im Bereich $[0, 2\pi]$ definiert. Ein Winkel von 0 entspricht Rot, $2\pi/3$ Grün, $4\pi/3$ Blau und 2π wiederum Rot (Abbildung 3.7). Die Sättigung S entspricht der Reinheit der Farbe. Je höher die Reinheit ist, desto geringer ist der Weißanteil in der Farbe. Eine Sättigung von eins entspricht einer komplett reinen Farbe, ein Sättigung von null Weiß. Eine Farbe mit einer Sättigung von null entspricht bei maximaler Helligkeit V Weiß. Bei geringerer Helligkeit durchläuft die Farbe verschiedene Grautöne und ist bei minimaler Helligkeit schwarz (Abbildung 3.6).

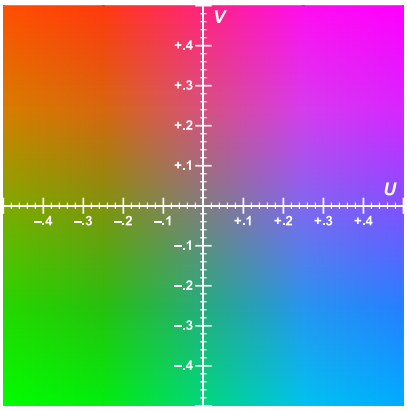


Abbildung 3.5: UV-Ebene des YUV-Farbmodells bei einem Luminanzwert Y von 0.5.[5]

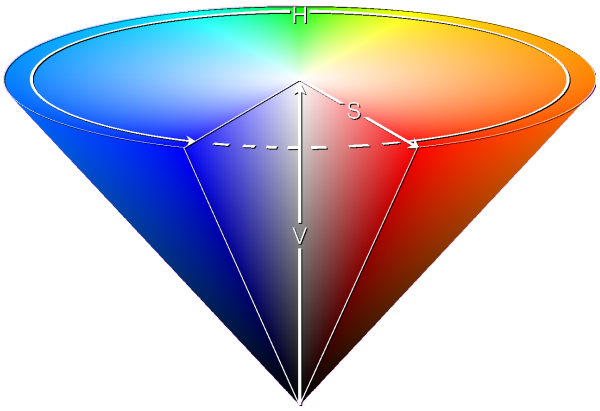


Abbildung 3.6: Darstellung des HSV-Farbraums als Kegel. Ein Farbwert besteht aus Farbton H, Sättigung S und Helligkeit V.[6]

Die Umwandlung zwischen dem RGB- und dem HSV-Farbraum ist komplexer und damit auch vergleichsweise teuer verglichen mit der Umwandlung zwischen RGB und YUV. Es gibt bei der Umwandlung einen Spezialfall zu beachten: Wenn $R = G = B$ gilt ist die Farbe komplett ungesättigt, der Farbton H in diesem Fall aussagelos und daher undefiniert. Die Algorithmen zur Umwandlung zwischen RGB und HSV finden sich im Anhang A.

[5] Quelle: `http://en.wikipedia.org/wiki/File:YUV_UV_plane.svg`
[6] Quelle: `http://en.wikipedia.org/wiki/File:HSV_triangle_and_cone.png`

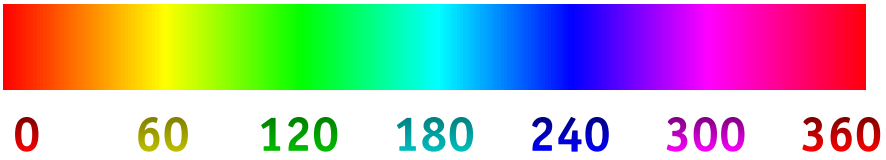


Abbildung 3.7: Farbtonskala der Farbwinkel H auf dem Farbkreis.
Quelle: `http://de.wikipedia.org/w/index.php?title=Datei:HueScale.svg`

Besonderheiten verglichen mit anderen Farbräumen

Das besondere am HSV-Farbmodell im Vergleich zu den beiden anderen Modellen ist, dass die Farbinformation in einer einzelnen Komponente, dem Farbton H, gespeichert ist. Das Problem ist, dass H nicht komplett unabhängig von der Beleuchtung ist. Ist V sehr klein, hat H fast keine Aussagekraft mehr. Auch wenn die Sättigung S unter einen gewissen Wert fällt, ist H nicht mehr sehr aussagekräftig.

Vitabile et al. haben in [24] ein System zum Erkennen von Verkehrszeichen entwickelt. Sie teilen je nach Sättigung und Beleuchtung den HSV-Farbraum in drei verschiedene Bereiche ein: den *achromatisch*, also nicht-farbigen, *Bereich*. Für diesen gilt $S \leq 0.25$ oder $V \leq 0.2$. Liegen Pixel in diesem Bereich werden sie bei der Erkennung nicht weiter betrachtet, da H keine Aussage hat. Im *unstabilen chromatischen Bereich*, $0.25 < S < 0.5$ und $0.2 < V < 0.9$, kann ein Drift von H beobachtet werden. Das System zur Verkehrszeichenerkennung rechnet mit einem vorher bestimmten Offset den Drift heraus. Im *chromatischen Bereich*, $S \geq 0.5$ und $0.2 < V < 0.9$, ist H stabil und kann direkt übernommen werden.

Sural et al. nutzen in [25] den HSV-Farbraum zur inhaltsbasierten Bildersuche. Hierbei werden Histogramme erstellt, zu denen alle Pixel beitragen. Wie in [24] gibt es auch hier das Problem zu lösen, dass je nach der Kombination von H, S und V die Werte unterschiedliche Aussagekraft haben. Menschen erkennen bei niedriger Sättigung kaum noch Farbunterschiede und nehmen fast nur noch den Helligkeitswert wahr. Wo genau dieser Schwellwert liegt, hängt wiederum von der Helligkeit ab. Sie nutzen die Formel $th_{sat}(V) = 1.0 - \frac{0.8V}{255}$ (wobei 255 hier der maximalen Helligkeit entspricht) um zu entscheiden, ob der H- oder V-Anteil eines Pixels dominant ist.

3.3.2 Faltung (Convolution) / Lineare Filterung

Bei der linearen Filterung eines Bildes wird jedes Pixel mit einer linearen Kombination aus sich und seinen Nachbarpixeln ersetzt. Erreicht wird dies durch eine diskrete 2D-Faltung eines Bildes I mit einer Maske M. Das gefilterte Bild I' ergibt sich, indem die Formel 3.1 auf jedes Pixel des Originalbildes angewendet wird.

$$I'[x, y] = I \otimes M = \sum_{i,j} I[x - i, y - j]g[i, j] \tag{3.1}$$

Da die Faltung linear ist, gelten folgende Eigenschaften:

$I'[x,y]$				$I[i,j]$				$M[i,j]$		
				3	5	2		$1/9$	$1/9$	$1/9$
	5		=	1	9	5	$\otimes$	$1/9$	$1/9$	$1/9$
				4	7	6		$1/9$	$1/9$	$1/9$

Abbildung 3.8: Faltung eines Pixels mit einer $3x3$-Maske

- Kommutativität: $f \otimes g = g \otimes f$
- Assoziativität: $(f \otimes g) \otimes h = f \otimes (g \otimes h)$
- Distributivität: $f \otimes (g + h) = f \otimes g + f \otimes h$

Mithilfe des Assoziativitätsgesetzes kann man sehr effizient mehrere Filter nacheinander auf ein Bild anwenden. Anstatt jeden einzelnen Filter auf das Gesamtbild anzuwenden, faltet man die einzelnen Filter und wendet den entstehenden Filter auf das Bild an. Da Filtermasken im Allgemeinen ein Vielfaches kleiner sind als die eigentlichen Bilder, wird so Rechenzeit eingespart.

Der umgekehrte Fall gilt jedoch auch: Lässt sich ein 2D-Filter in zwei 1D-Filter zerlegen, können diese nacheinander auf das Bild angewendet werden (Abbildung 3.9).

$1/3$	$1/3$	$1/3$	$\otimes$	$1/3$	=	$1/9$	$1/9$	$1/9$
				$1/3$		$1/9$	$1/9$	$1/9$
				$1/3$		$1/9$	$1/9$	$1/9$

Abbildung 3.9: Zerlegung einer $3x3$-Maske in zwei 1D-Filter

3.3.3 Glättung

Jedes Bild enthält Informationen, die für dessen Verarbeitung nicht benötigt werden oder nicht mit der Realität übereinstimmen. Vor der eigentlichen Erkennung wird versucht, möglichst viele dieser Informationen zu verwerfen. Als Beispiel kann man sich das Bild eines gelben Tores vorstellen. Im Idealfall wäre das Tor eine einheitliche gelbe Fläche, die von der Bildverarbeitung als solche erkannt werden sollte. In der Realität ist ein Tor jedoch keine einheitliche gelbe oder blaue Fläche, sondern setzt sich aus verschiedenen Flächen mit verschiedenen Störungen zusammen. Äußere Einflüsse wie Schatten, Glanzlichter oder Rauschen erschweren die Erkennung. Ein orangefarbener Ball ist auch nicht einheitlich, seine Struktur ist jedoch vollkommen irrelevant und kann verworfen werden. In den nächsten Abschnitten werden einige Techniken aufgezeigt, um unnötige oder falsche Informationen zu entfernen.

3.3.3.1 Rauschen

Jedes aufgenommene Kamerabild unterscheidet sich von der Realität, dem eigentlichen Bildsignal. Das Eingangssignal wird ungewollt verändert, die entstehende Differenz zwischen dem Nutzsignal und dem gemessenen Signal bezeichnet man als Rauschen. Eine häufig auftretende Art des Rauschens ist das sogenannte additive Gaußsche Rauschen. Die Amplituden der Störung, die auf das Nutzsignal aufaddiert werden, sind hierbei gaußverteilt.

Unter der Annahme, dass die Umgebung eines Pixels Informationen über die korrekte Intensität eines Pixels enthält, werden in den folgenden Abschnitten drei Filter vorgestellt, um Rauschen zu entfernen. Rauschen findet fast nur in den hohen Frequenzen statt, alle Filter sind also im Fourierraum Tiefpassfilter.

3.3.3.2 Durchschnittsfilter

Der Durchschnittsfilter ersetzt alle Pixel eines Bildes mit dem Durchschnitt des Originalpixels und der umliegenden Pixel. Für die Filtermaske eines Durchschnittsfilters der Größe $3x3$ siehe Abbildung 3.10. Andere Größen folgen analog. Der Filter entfernt uniformes und gaußsches Rauschen sehr gut, allerdings wird das Bild durch diesen Filter unscharf.

$1/9$	$1/9$	$1/9$
$1/9$	$1/9$	$1/9$
$1/9$	$1/9$	$1/9$

Abbildung 3.10: Durchschnittsfilter der Größe $3x3$

3.3.4 Medianfilter

Der Medianfilter arbeitet ähnlich wie der Durchschnittsfilter, ersetzt Pixel jedoch nicht mit dem Durchschnitt, sondern dem Median des Originalpixels und umliegender Pixel. Hierdurch ergeben sich zwei Vorteile gegenüber dem Durchschnittsfilter:

- Ein Ausreißer in der Nachbarschaft hat keinen signifikanten Einfluss auf das Ergebnis des Medians.
- Das Ergebnis des Medians ist immer ein Pixelwert, der in umliegenden Pixel bereits vorhanden war. Es werden keine neuen Pixelwerte erzeugt – aus diesem Grund erhält der Medianfilter scharfe Kanten besser.

Der Medianfilter verwendet keine Filtermaske und ist nicht linear. Ein großes Problem des Filters ist, dass er relativ teuer ist. Für die Berechnung des Medians eines Pixels müssen alle umliegenden Pixel sortiert werden. Die Sortieroperation ist selbst mit schnellen Algorithmen durchschnittlich mindestens $\mathrm{O}\big(n \log n\big)$, während die Faltung, also ein maskenbasierter Filter, $\mathrm{O}\big(n\big)$ garantiert.

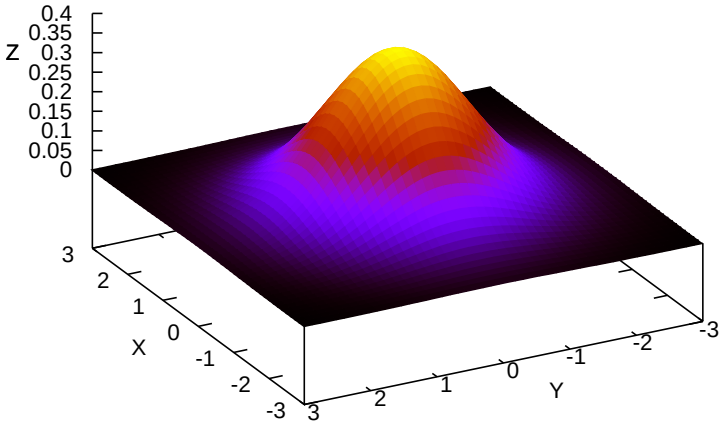


Abbildung 3.11: 2D-Gaussfunktion mit $\sigma = 1$

3.3.5 Gaußfilter

Der Durchschnittsfilter bewertet alle Pixel gleich. Der Gaußfilter dagegen geht davon aus, dass Pixel näher am Ursprungspixel einen stärkeren Einfluss haben, und gewichtet diese entsprechend stärker. Die 2D-Gaussverteilung (Abbildung 3.11) hat die Form

$$g(x,y) = \frac{1}{\sqrt{2\pi}} exp(-\frac{x^2+y^2}{2\sigma^2})$$

Da der Gaußfilter jedoch separabel ist, ist die 1D-Form

$$g(x) = \frac{1}{\sqrt{2\pi}} exp(-\frac{x^2}{2\sigma^2})$$

ausreichend zur Berechnung des Filters. Der Erwartungswert fällt in den Formeln weg, da er mit null angesetzt wird. Abhängig von σ ergibt sich eine Glättung über einen größeren oder kleineren Bereich. Mathematisch gesehen ist die Gaußfunktion zwar überall größer als null, aus praktischer Sicht sind die meisten Werte jedoch quasi null. Die diskretisierte Approximation der Gaußfunktion wird daher nur in einem Bereich bis etwa 3σ betrachtet.

Durch die stärkere Gewichtung naher Pixel ist der Gaußfilter besser als der Durchschnittsfilter geeignet, bestehende Strukturen zu erhalten und gleichzeitig Rauschen zu entfernen. Eine gewisse Unschärfe des Bildes lässt sich trotzdem nicht vollständig vermeiden.

3.3.6 Bresenham-Algorithmus

Der erste Rasterisierungsalgorithmus wurde in den 1960er Jahren von Jack Bresenham entwickelt [26]. Der Algorithmus von Bresenham wird dazu verwendet, um Linien zu rasterisieren, also auf ein Punktraster abzubilden (Abbildung 3.12). Das Ziel hierbei ist es, eine Linie nur mit ganzzahligen Werten zu zeichnen.

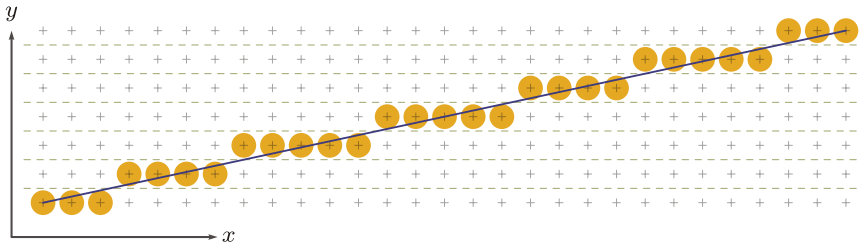


Abbildung 3.12: Rasterisierung einer Linie mit dem Bresenham-Algorithmus. Basierend auf `http://upload.wikimedia.org/wikipedia/commons/6/6a/Bresenham_run-based.svg`

3.3.7 Kantenerkennung

Kantenerkennung ist ein essenzieller Teil des Objekterkennungs- und Bildsegmentierungsprozesses, da Kanten im Idealfall am Rand von Objekten auftreten und diese vom Rest des Bildes trennen (Abbildung 3.15). Des Weiteren ist die zu betrachtende Datenmenge in einem (binären) Kantenbild, verglichen mit dem original Bild, erheblich reduziert. Kanten treten jedoch auch innerhalb von Objekten in einem Bild an allen Stellen auf, an denen der Kontrast zwischen Pixeln groß ist.

Idealerweise erfüllt eine Kantenerkennung folgende Bedingungen (siehe [27]):

- Gute Erkennung: Möglichst alle Kanten sollten erkannt werden. Gleichzeitig sollten Nicht-Kantenpunkte, zum Beispiel Rauschen, nicht fälschlicherweise als Kante markiert werden.
- Gute Lokalisierung: Eine erkannte Kante sollte möglichst nah am Mittelpunkt der tatsächlichen Kante liegen.
- Eine Kante sollte nur ein Mal erkannt werden.

Bei der Glättung von Bildern kamen Tiefpassfilter zum Einsatz, um hohe Frequenzen auszufiltern. Da wir uns jetzt jedoch genau für diese Frequenzen interessieren, entspricht die Kantenerkennung einem Hochpassfilter.

Ein großer Kontrast zwischen zwei Pixeln entspricht einem hohen Gradienten des Bildes, also einem großen Maximum der ersten Ableitung oder einem Nulldurchgang der zweiten Ableitung (Abbildung 3.13). Da ein Bild als diskrete Funktion gesehen werden kann, kann

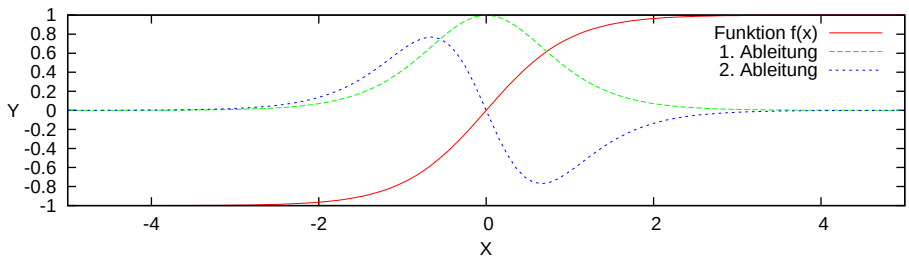


Abbildung 3.13: Kante in einem Bild und ihre ersten zwei Ableitungen

die Ableitung in x-Richtung durch

$$\frac{df(x))}{d(x))} = f(x+1) - f(x) \tag{3.2}$$

approximiert werden. Die y-Richtung folgt analog. Ist der Gradient in x- und y-Richtung bestimmt, ergibt sich der Betrag des Gradienten der Kante zu $|G| = \sqrt{|G_x|^2 + |G_y|^2}$. Oft wird jedoch die Approximation $|G| = |G_x|^2 + |G_y|$ verwendet. Die Orientierung der Kante ergibt sich zu $\theta = \arctan G_y/G_x$.

Der naive Filterkernel für Gleichung 3.2 ist $[-1, 1]$, dieser ist jedoch nicht symmetrisch und wird normalerweise nicht eingesetzt. Statt dessen gibt es Alternativen wie den Prewitt- oder Sobelfilter (Abbildung 3.14). Diese sind symmetrisch und betrachten nicht nur zwei benachbarte Pixel, sondern ein Pixel und die komplette 8-Nachbarschaft. Der Sobelfilter ist dem Prewittfilter sehr ähnlich, gewichtet die direkten Nachbarn jedoch stärker. Dies entspricht einer Glättung senkrecht zum bestimmten Gradienten.

-1	0	1
-1	0	1
-1	0	1

(a) Prewitt x-Ableitung

-1	-1	-1
0	0	0
1	1	1

(b) Prewitt y-Ableitung

-1	0	1
-2	0	2
-1	0	1

(c) Sobel x-Ableitung

-1	-2	-1
0	0	0
1	2	1

(d) Sobel y-Ableitung

Abbildung 3.14: Filtermasken des Prewitt- und Sobelkantenerkenners

Nach der Berechnung der Gradienten gilt es zu entscheiden, welche der Gradienten zu einer Kante gehören und welche nicht. Ein erster naiver Ansatz ist beispielsweise, alle Gradienten, die einen bestimmten Betrag überschreiten, als Kante zu definieren. Dieser Ansatz ist jedoch sehr anfällig für Rauschen, wenn der Threshold zu klein gewählt wird. Bei einem zu großen Threshold dagegen werden viele Kanten nicht erkannt. Sinnvoll ist es beispielweise, statt einzelner Pixel zusammenhängende Konturen zu erkennen. Die Pixel der Kontur werden nun als Kante erkannt, wenn einige der Pixel über einem oberen Schwellwert und die anderen Pixel mindestens über einem unteren Schwellwert liegen. Bei diesem Ansatz gibt es weniger Fehlerkennungen, da der Threshold ab dem eine Kante erkannt wird relativ hoch liegt. Gleichzeitig werden jedoch mehr Kanten erkannt, da nicht alle Pixel einer Kontur über dem hohen Schwellwert liegen müssen. John Canny hat diesen Ansatz mit dem sogenannten Canny-Algorithmus verfolgt und mathematisch verifiziert [27].

3.3.7.1 Canny-Algorithmus

Der Canny-Algorithmus läuft in mehreren Schritten ab.

Glättung Als Erstes wird ein Gaußfilter auf das Bild angewendet, um möglichst viel Rauschen zu entfernen.

(a) Original (b) Sobel (c) Canny

Abbildung 3.15: Standardtestbild für Bildverarbeitungsalgorithmen „Lena“ und die von Sobel-Kantenerkenner beziehungsweise Canny-Algorithmus erkannten Kanten in diesem Bild.
Quelle: `http://sipi.usc.edu/database/misc/4.2.04.tiff`

Bestimmung der Gradienten Im nächsten Schritt werden, beispielsweise mit dem Sobel-Operator, die Gradienten des Bildes berechnet. Die Winkel der Gradienten werden jetzt quantisiert (beispielsweise auf 0, 45, 90 und 135 Grad).

Bestimmung der Maxima Von den bestimmten Gradienten im Bild können nur die Gradienten einer Kante entsprechen, die ein Maximum der Pixel senkrecht zu ihrem Winkel darstellen. Ein Pixel mit dem Winkel null kann beispielsweise nur zu einer Kante gehören, wenn der Absolutwert der Gradienten der Pixel über und unter ihm kleiner ist, als der eigene. Für alle anderen Winkel funktioniert dies analog. Alle Pixel, die einem Maximum entsprechen, werden auf eins gesetzt, alle anderen auf null. Das so entstandene binarisierte Bild enthält alle Kandidaten für Kanten.

Hysterese Wie bereits erwähnt, arbeitet der Canny-Algorithmus mit zwei Thresholds. Im binarisierten Bild werden alle Pixel als zu einer Kante zugehörig betrachtet, wenn sie oberhalb des größeren Thresholds liegen. Startend von diesen Punkten werden so lange Pixel in beide Richtungen des Gradienten als Kante akzeptiert, bis ein Pixel unterhalb des kleineren Thresholds liegt. Eine zusammenhängende Kontur startet also immer mit einem Pixel, das oberhalb des größeren Thresholds liegt, und enthält nur Pixel, die mindestens über dem unteren Threshold liegen.

Parameter Erfolg und Misserfolg des Canny-Algorithmus, sowie von jedem anderen Kantenerkenner, hängen von der Wahl der Parameter ab. Ist das σ des Gaußfilters zu groß werden feine Kanten eliminiert, ist es zu klein wird zu viel Rauschen fälschlich als Kante erkannt. Genau das gleiche Problem gibt es bei der Wahl der Thresholds.

Es ist bis heute nicht möglich, automatisch die besten Parameter zu finden. Sie müssen von Hand sowohl für die verwendeten Eingangsdaten als auch für den Verwendungszweck optimiert werden.

3.4 Eingesetztes Robotersteuerungsframework

Am Fachgebiet SIM der TU Darmstadt wurden 2005 im Rahmen einer Diplomarbeit die Grundlagen für die Robotersteuerungssoftware *RoboFrame* entwickelt [28–30]. Dieses Framework ermöglicht es, plattformunabhängige Anwendungen für beliebige autonome Robotersysteme zu entwickeln. Bei der Portierung auf oder der Entwicklung eines neuen Robotertyps können oft große Teile der Anwendungen übernommen werden. Durch eine Abstraktionsschicht im Framework werden durch plattformspezifische Änderungen, wie einem Wechsel des Betriebssystems oder der Hardwarebasis, nur Änderungen im Framework nicht jedoch in den Anwendungen nötig.

Eine mit *RoboFrame* geschriebene Anwendung besteht aus mehreren gekapselten Modulen, beispielsweise Verhalten und Bewegungserzeugung, die untereinander Daten austauschen. Diese Module können nacheinander oder parallel in verschiedenen Threads laufen. *RoboFrame* selbst besteht aus zwei Teilen – *RoboApp* und *RoboGui*. Mit *RoboApp* werden die eigentlichen Applikationen geschrieben, die auf dem Roboter laufen. *RoboGui* erlaubt es, eine grafische Oberfläche zu schreiben, die über Netzwerk mit der Applikation auf dem Roboter kommuniziert. Die grafische Oberfläche kann zum Steuern, Debuggen und Testen der Applikation genutzt werden.

Für den Austausch von Daten zwischen verschiedenen Modulen und/oder Anwendungen sind in *RoboFrame* sogenannte *Connector*-Objekte zuständig. Diese können dynamisch zur Laufzeit einem Router hinzugefügt oder aus diesem entfernt werden, der alle Datenpakete zwischen den registrieren Konnektoren austauscht oder and den nächsten Router weiterreicht (siehe 6.1.8 in [28]). Die Datenpakete selbst werden in sogenannte `Streamable`-Objekte verpackt, um verschickt zu werden. Durch Serialisieren und Deserialisieren ist es hierdurch auch problemlos möglich, die Pakete über das Netzwerk zu verschicken. Alle Streamables können entweder direkt vom Roboter oder nach Anforderung von der grafischen Oberfläche in Logdateien gespeichert werden. Aus diesen können die gespeicherten Daten später offline abgespielt und ausgewertet werden.

Diese Arbeit baut auf *RoboFrame* und insbesondere den mit ihr für den RoboCup entwickelten Anwendungen auf.

3.5 Hardwareplattformen

Am Fachgebiet SIM werden verschiedene Hardwareplattformen eingesetzt und erforscht. In diesem Abschnitt werden drei der Hardwareplattformen beschrieben, deren Softwarearchitektur auf *RoboFrame* basiert. Auf den DD200x wurde diese Arbeit umgesetzt, auf den restlichen Plattformen können die Ergebnisse dieser Arbeit genutzt werden.

Abbildung 3.16: DD2008 Bruno
(Quelle: [2])

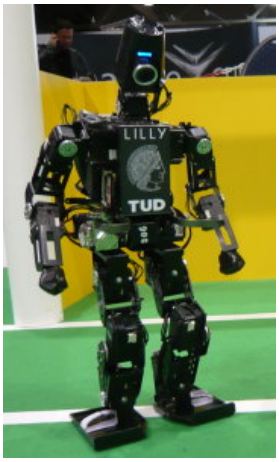


Abbildung 3.17: DD2007 Lilly
(Quelle: [2])

3.5.1 DD200x

Das Hajime Sakamoto Institute Ltd. [31] entwickelt seit mehreren Jahren in Kooperation mit den *Darmstadt Dribblers* die Roboter, welche die *Darmstadt Dribblers* in der RoboCup Kid-Size-Humanoidliga einsetzen. Die Roboter werden ständig weiter entwickelt, im Moment sind die Revisionen DD2007, DD2008 und DD2009 im Einsatz (Abbildungen 3.16 und 3.17).

Roboter des Typs DD2007 sind 55 cm, Roboter des Typs DD2008 oder DD2009 57,5 cm hoch. Ihr Gewicht beträgt 3,3 kg. Der Bewegungsapparat der Roboter wird von 21 Motoren angetrieben. Diese 21 Freiheitsgrade teilen sich wie folgt auf: sechs in jedem Bein, drei in jedem Arm, einer in der Hüfte und zwei im Hals. Die *Knochen* des Roboters bestehen aus eloxiertem Aluminium.

Um die eigene Lage im Raum und auftretende Beschleunigungen zu messen, sitzen in der Hüfte Gyroskope und Accelerometer. Diese messen die Drehbewegung beziehungsweise Beschleunigung in allen drei Raumachsen. Mit diesen Sensoren wird unter anderem die Laufbewegung stabilisiert, der Roboter erkennt, wenn er umgefallen ist oder zieht den Kopf nach hinten, wenn er fällt.

Die Umwelt können die Roboter über eine im Kopf angebrachte USB-Kamera wahrnehmen. Diese wird mit einer Auflösung von 640x480 Pixeln und einem Öffnungswinkel von etwa $65°$ betrieben. Über die Kamera nimmt der Roboter Tore, den Ball, Feldlinien, Landmarken sowie Hindernisse wie Gegner wahr. Mit einem USB-WLan-Stick kann ein Roboter mit anderen Robotern kommunizieren. Über diesen oder eine alternative Kabelverbindung kann der Nutzer sich direkt mit einem Roboter verbinden, von diesem Informationen wie Kamerabilder empfangen und ihm Befehle schicken.

Im Wettkampf agieren die Roboter vollkommen autonom, also ohne Steuerung von außen. Hierfür tragen sie ihr Gehirn und Nervensystem in Form eines PC/104-Boards und eines Mikrokontroller mit sich. Das PC/104-Board mit einem AMD Geode LX800 500 MHz als Prozessor nimmt die meisten Aufgaben der Softwaresteuerung wahr. Zeitkritische Teile

die Steuerung und Regelung betreffen, wie zum Beispiel die Erzeugung der Laufbewegung, Ansteuerung der Motoren oder das Auslesen von Sensoren, werden von einem 160 MHz schnellen Mikrokontroller übernommen. Die beiden Boards kommunizieren über einen RS232-Bus miteinander.

Die Stromversorgung der Roboter wird je nach Modell von zwei oder drei Lithium-Polymer-Akkus übernommen. Mit einer Ladung kann ein Roboter etwa 15 Minuten betrieben werden.

Abbildung 3.18: DRRT Monstertruck (Quelle: [32])

3.5.2 Monstertruck

Neben den *Darmstadt Dribblers* mit den DD200x nimmt seit 2009 ein weiteres Team der TU Darmstadt am RoboCup teil. In der Search-And-Rescue-Liga tritt das *Darmstadt Rescue Robot Team* [33] mit einem umgebautem Kyosho Twin Force RC, einem sogenannten Monstertruck, an (Abbildung 3.18) [32].

Der Monstertruck wird von einem Vierradantrieb bewegt. Dieser erlaubt auch eine Fortbewegung des Fahrzeugs, wenn nicht alle Räder den Boden berühren. Die 4-Radlenkung erlaubt es, alle vier Räder einzuschlagen und so das Fahrzeug sehr flexibel zu steuern.

Mit einem Laserscanner vom Typ Hokuyo URG04-LX kann der Monstertruck in horizontaler Ebene in einem Winkel von 240° Objekte in einem Abstand von bis zu vier Metern erfassen. Um die Fortbewegung des Fahrzeugs und seine Position im Raum besser bestimmen zu können, sind zusätzlich alle vier Räder mit optischen Encodern zum Auslesen der aktuellen Radposition versehen. Über ein inertiales Navigationssystem können zusätzlich mit Gyroskopen und Accelerometern Drehbewegungen und Beschleunigungen um Raum gemessen werden. Die Informationen aus diesen Sensoren werden zu einer Schätzung der aktuellen Position, Geschwindigkeit und Lage fusioniert. Zusätzlich wird eine Karte der Umgebung erstellt.

Lokalisierung und Mapping, Verhaltensentscheidungen oder Ansteuerung der Motoren werden, ähnlich wie bei den DD200x, von einer Kombination aus PC/104-Board und einem Interfaceboard zur Kommunikation mit Sensoren und Motoren übernommen.

In der Search-And-Rescue-Liga besteht ein wesentlicher Teil der zu lösenden Aufgaben darin, Opfer zu finden und zu identifizieren. Die Identifikation der Opfer findet über die Auswertungen von Kamerabildern einer Tageslicht- und einer Wärmebildkamera statt. Die Bildverarbeitung findet nicht auf dem PC/104-Board statt, sondern auf einem weiteren unabhängigen System, der sogenannten *Vision Box*. Mithilfe einer eingebauten nVidia Grafikkarte ist es möglich, die Bilder in Echtzeit auszuwerten. Gefundene Opfer werden an das PC/104-Board weiter gegeben.

Genau wie bei den DD200x wird die Stromversorgung von Lithium-Polymer-Akkus übernommen.

3.5.3 HR27

Bis 2008 waren die Sony AIBOs eine wichtige Plattform im RoboCup. Da Sony jedoch die Produktion des vierbeinigen Roboterhundes eingestellt hat, musste nach Alternativen gesucht werden. In Kooperation mit dem Hajime Sakamoto Institute Ltd. entwickelte das Fachgebiet SIM der TU Darmstadt basierend auf Anforderungen der RoboCup Federation [34] den Roboterhund HR27 (Abbildung 3.19) [35].

Der 2.8 kg schwere Roboter besitzt 15 Freiheitsgrade, drei im Hals sowie drei in jedem Bein. Die damit möglichen Bewegungen ähneln denen des AIBOs, es können jedoch auch Bewegungen ausgeführt werden, welche mit den AIBOs nicht möglich sind. Die maximale Laufgeschwindigkeit liegt bei etwa 40-50 cm/s.

Die Umwelt wird über eine Kamera im Kopf mit einer Auflösung von 640x480 Pixeln wahrgenommen. Zusätzlich liefern Infrarotsensoren in Brust und Kopf Position und Abstand von Objekten im Abstand von 10 cm bis 80 cm. Der Kopf und damit das Kamerabild kann

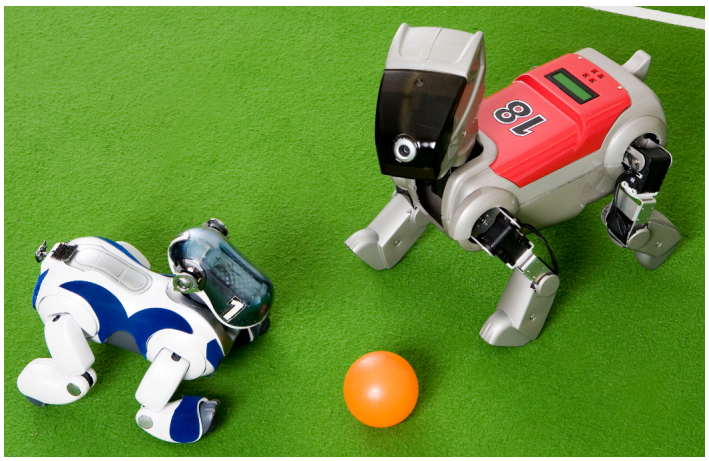


Abbildung 3.19: HR27 mit Sony AIBO (Quelle: Katrin Binner/TU Darmstadt)

durch ein Gyroskop im Kopf stabilisiert werden. Ein weiteres Gyroskop um die z-Achse sowie ein 3D-Accelerometer befinden sich im Körper.

Ähnlich wie bei den DD200x kommt ein PC/140-Board in Kombination mit einem Mikrocontrollerboard zur Datenverarbeitung und Steuerung des Roboters zum Einsatz. Als Betriebssystem kann neben Linux auch Windows CE verwendet werden.

Mit einem Satz Lithium-Polymer-Akkus wird eine Laufzeit von über 30 Minuten erreicht.

Kapitel 4

Anforderungen

In diesem Kapitel werden aufbauend auf den vorherigen Kapiteln die Anforderungen an den zu schreibenden Code formuliert.

4.1 Evaluator Dialog

Die zu schaffende Infrastruktur soll es ermöglichen, aufgenommene Daten verschiedener Algorithmen zu vergleichen. Die Infrastruktur selbst soll hierbei jedoch keine Annahme über die verwendeten Algorithmen treffen, sondern nur generische Schnittstellen zur Verfügung stellen. Durch diese Schnittstellen können Module auf die Rohdaten zugreifen und diese aufbereiten, indem Daten passend in Datensätze gruppiert werden. Beispielsweise könnte ein Modul alle Daten zusammenfassen, die zu einem bearbeiteten Bild gehören.

Die aufbereiteten Rohdaten sollen im nächsten Schritt durch sogenannte Evaluatoren ausgewertet werden können. Wie die Evaluatoren die Daten auswerten, ist nicht durch die Infrastruktur, sondern nur durch die Implementierung der Evaluatoren selbst, zu begrenzen. Den Evaluatoren sollen feste Schnittstellen zur Verfügung gestellt werden, um ihre Ergebnisse anzuzeigen. Evaluatoren sollen dabei Feedback geben können wie interessant ein Datensatz ist[1] sowie die Auswertung eines Datensatzes und die Auswertung aller Datensätze anzeigen können. Mögliche Arten, eine Auswertung anzuzeigen, wären beispielsweise in tabellarischer Form oder in Form eines Graphen. Die Infrastruktur hat hierfür Schnittstellen bereitzustellen, um weitere Möglichkeiten der Visualisierung einzubinden. Um die Auswertung der Daten zu erleichtern, ist eine Möglichkeit vorzusehen, interessante Datensätze anhand des von den Evaluatoren gegebenen Feedbacks auszufiltern und diese getrennt darzustellen.

Die gesamte Implementierung ist auf Basis von *RoboFrame* zu schreiben. Stellt *RoboFrame* gewünschte Funktionalität nicht zur Verfügung ist diese, soweit sie für andere Nutzer von *RoboFrame* interessant sein könnte, direkt in dieses zu integrieren. Die Infrastruktur ist in zwei Teile zu trennen. Die Benutzeroberfläche soll auf *RoboGui* basieren, während die geschriebenen Module komplett auf *RoboApp* basieren sollen. Hierdurch wird einerseits eine Trennung von Daten und Visualisierung erreicht. Es wird jedoch vor allem damit ermöglicht, die Evaluierungsalgorithmen einfacher komplett von der GUI zu trennen und diese in

[1] Beispiel: Je unterschiedlicher die Ergebnisse der Algorithmen sind, desto interessanter ist ein Datensatz.

einem anderen Prozess oder auf einem anderen Computer laufen zu lassen. Vorstellbar wäre beispielsweise, dass ein Roboter Daten sammelt, diese direkt auswertet und nur die Zusammenfassung an die GUI schickt. Die vorliegende Arbeit soll dies jedoch noch nicht leisten. GUI und Auswertung sind nicht zu trennen.

Die den Datenmodulen zur Verfügung gestellten Rohdaten müssen nicht vollständig sein. Es muss möglich sein, die vorhandenen Daten abzuspielen, diese durch Algorithmen bearbeiten zu lassen und die von ihnen neu generierten Daten wieder zu speichern. Welche Daten abgespielt und gespeichert werden, wird von den Datenmodulen selbst entschieden. Ein Beispiel hierfür ist die Bildverarbeitung. Nachdem vom Roboter Bilder und Transformationsmatrizen aufgenommen wurden, könnte man diese wiederum abspielen und mit verschiedenen Bildverarbeitungsalgorithmen bearbeiten. Dabei generierte Perzepte wiederum würde das Datenmodul speichern und den Evaluatoren zur Verfügung stellen. Da *RoboGui* es momentan nicht erlaubt, Bildverarbeitungen oder allgemeiner gesagt Prozesse und Connectoren aus der GUI zu starten, ist diese Funktionalität *RoboFrame* hinzuzufügen.

Die Funktionalität und Flexibilität der Infrastruktur ist anhand von verschiedenen zu implementierenden Datenmodulen und Evaluatoren zu zeigen.

4.2 Neue Bildverarbeitung

Die aktuelle Bildverarbeitung der Roboter arbeitet, um Rechenzeit zu sparen, nicht auf ganzen Bildern, sondern nur auf einem Bruchteil des Bildes, den sogenannten Scanlines. Des Weiteren werden die genutzten Farbinformationen der Bildverarbeitung aus von Hand erzeugten Lookuptabellen gewonnen. Ändern sich die Lichtbedingungen, müssen die Lookuptabellen angepasst werden. Die Ergebnisse der Bildverarbeitung können nur manuell überprüft werden, es gibt keine Referenzbildverarbeitung, um automatisch Fehler in der Bildverarbeitung zu entdecken.

Um oben genannte Punkte zu adressieren, soll eine Bildverarbeitung mit einem neuen Ansatz geschrieben werden. Diese wird nicht durch die verfügbare Rechenzeit begrenzt. Es ist nicht das Ziel, eine neue Bildverarbeitung zu entwickeln, die auf aktuellen humanoiden Robotersystemem lauffähig ist. Sie soll vielmehr später, wenn mehr Rechenzeit zur Verfügung steht, die derzeitige Bildverarbeitung ersetzen können. Bis dahin soll es möglich sein, die Bildverarbeitung zu nutzen, um die Ergebnisse der derzeitigen Bildverarbeitung zu bewerten. Die Bildverarbeitung läuft in diesem Fall nicht auf einem Roboter, sondern wird *offline* auf einem beliebigen Computer in nicht Realzeit ausgeführt.

Das Ziel ist nicht, einen vollständigen Ersatz der aktuellen Bildverarbeitung zu schaffen. Vielmehr sollen die Möglichkeiten untersucht werden, mit mehr Rechenleistung die nötigen Eingriffe des Nutzers bei veränderten Lichtbedingungen zu minimieren. Die neue Bildverarbeitung ist so einzubinden, dass sie anstelle der bestehenden Bildverarbeitung genutzt werden kann.

Kapitel 5

Konzept

In diesem Kapitel wird die konzeptuelle Umsetzung der in Kapitel 4 definierten Anforderungen beschrieben.

5.1 Evaluator Dialog

Der sogenannte *Evaluator Dialog* ist die grafische Oberfläche, die dem Benutzer zur Evaluation zur Verfügung gestellt wird. Der Dialog ist ein *RoboGui*-Dialog und kann somit direkt in alle mit *RoboGui* geschriebenen Nutzeroberflächen integriert werden. Mithilfe des Dialogs kann der Nutzer Logdateien laden, ein Storagemodul auswählen und passend zu diesem aus einer Menge von Evaluatoren wählen. Der Dialog legt die nötigen Module an und initialisiert diese. Danach kann er von diesen Informationen abfragen.

Abbildung 5.1 stellt die wichtigsten Komponenten und ihren Zusammenhang schematisch dar.

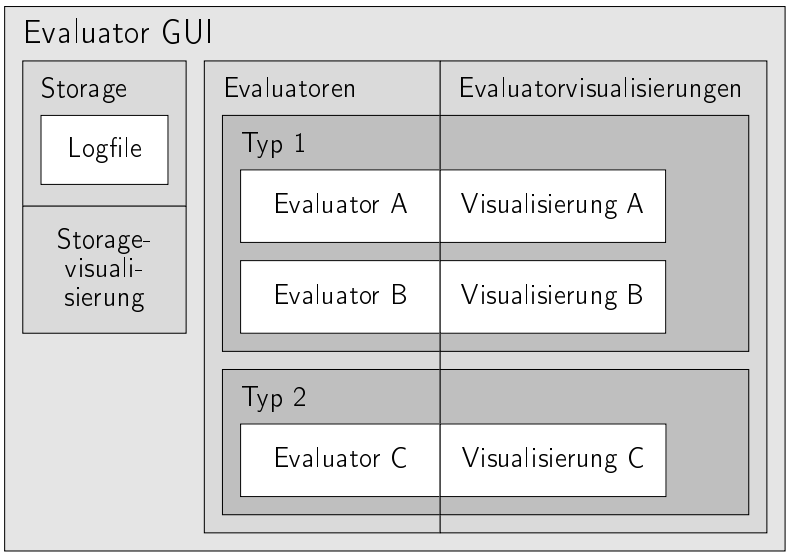


Abbildung 5.1: Aufbau der GUI und ihrer Komponenten mit einer aktiven Storage und drei Evaluatoren

Storage

Die auszuwertenden Daten werden von sogenannten Storagemodulen gekapselt. Diese benutzen die von *RoboFrame* zur Verfügung gestellte `Logfile`-Klasse und entsprechende Logdateien, um die rohen Daten der Algorithmen zu speichern. Die Aufgabe der Storagemodule ist es, aus den Logdateien die interessanten Informationen zu extrahieren und aufzubereiten. Wie genau ein konkretes Modul die Daten aufbereitet, ist nicht vorgegeben. Die einzige Annahme, die gemacht wird, ist, dass die Daten in Datenpakete gepackt werden, die einzeln bearbeitet werden können. Um eine Visualisierung in der GUI zu ermöglichen, kann der Dialog von einem Storagemodul die Anzahl der Datenpakete sowie eine textuelle Beschreibung aller Pakete abfragen.

Evaluatoren

Zu jedem Storagemodul können beliebig viele Evaluatoren geschrieben werden. Über ein modulspezifisches Interface kann ein Evaluator Datenpakete vom Storagemodul abholen und auswerten. In welcher Form die Evaluatoren Datenpakete verarbeiten, ist nicht festgelegt. Je nachdem wie ein Evaluator das Ergebnis seiner Auswertung in der GUI darstellen will, implementiert er ein spezielles Interface, mit dem der Dialog die zur Visualisierung benötigten Daten abfragt. Verschiedene Typen der Darstellung entsprechen verschiedenen Interfaces.

Evaluatorvisualisierung

Der Dialog weiß nichts über die Visualisierung verschiedener Evaluatoren. Zur Visualisierung der Ergebnisse erstellt er je nach Typ des Evaluators ein passendes Visualisierungsobjekt. Wenn die Auswertung eines Evaluators angezeigt werden soll, übernimmt dieses Objekt die grafische Darstellung.

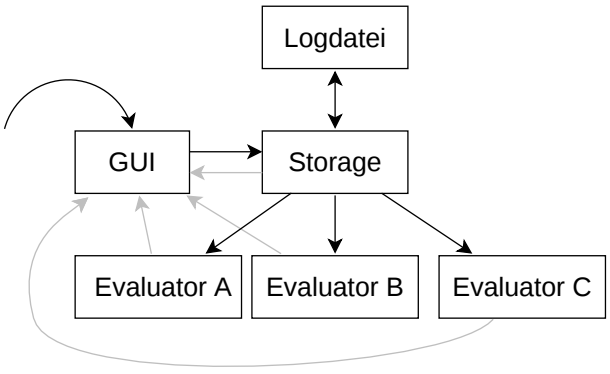


Abbildung 5.2: Datenfluss zwischen den einzelnen Komponenten. Schwarze Pfeile entsprechen dem Austausch von auswertbaren Daten, graue Pfeile Visualisierungsinformationen.

Abbildung 5.2 stellt den Fluss der Daten durch die verschiedenen Module der Infrastruktur dar. Im Normalfall wird ein Logfile von einem Storagemodul geladen und bearbeitet. Im nächsten Schritt teilt der Dialog jedem erzeugten Evaluator mit, welchen Datensatz er auswerten soll. Falls der Evaluator den Datensatz nicht bereits bearbeitet hat, ruft der Evaluator den

passenden Datensatz aus dem Storagemodul ab und verarbeitet diesen. Sobald ein Ergebnis vorliegt, signalisiert der Evaluator dies dem Dialog und die passende Visualisierung kann das Ergebnis anzeigen. Informationen fließen in diesem Szenario also aus der Logdatei über das Storagemodul in die Evaluatoren und von dort in Form einer grafischen Auswertung in den Dialog. Über den Dialog steuert der Nutzer hierbei, welche Informationen genau ausgewertet werden.

In einem Fall können auch Daten aus dem Dialog über das Storagemodul ins Logfile fließen: Der Dialog ist ein *RoboFrame*-Connector und kann Daten, die von anderen Connectoren erzeugt werden, empfangen. In einem speziellen Aufnahmemodus des Dialogs kann ein Storagemodul der GUI mitteilen, an welchen Daten es interessiert ist. Der Dialog fragt diese Daten beim Framework an und reicht sie bei Verfügbarkeit an das Storagemodul weiter. Sinn dieser Möglichkeit ist es, mithilfe der Daten im Logfile weitere Informationen zu erzeugen. Einem Logfile, das nur Bilder und Kameramatrizen enthält, könnten beispielsweise Perzepte hinzugefügt werden.

5.2 Bildverarbeitung

Die Kernkomponenten, auf denen die Bildverarbeitung aufbaut, sind Farbklassifikation im HSV-Bildraum und Kantenerkennung.

Wie in 3.3.1.3 beschrieben besteht der HSV-Farbraum aus einer Komponente H, die den Farbton angibt, einer Komponente S, die die Sättigung oder auch Reinheit der Farbe angibt, und der Komponente V, die der Helligkeit des Farbwertes entspricht. Über Farbwerte, die im RoboCup im Allgemeinen auftreten, kann man einige Annahmen treffen (siehe auch 3.3.1.3):

- Eine Farbe deckt einen bestimmten Bereich and Farbtönen ab. Die für den Menschen als blau erkennbaren H-Werte liegen zum Beispiel etwa zwischen $180°$ und $300°$.
- Die Sättigung von klar erkennbaren Farbtönen wird zwischen dem Maximum und einem Schwellwert liegen, der größer als Null ist. Sobald die Sättigung unter dem Schwellwert liegt, kann der Pixel nicht mehr sicher einer Farbklasse zugeordnet werden.
- Genau wie bei der Sättigung gibt es für die Helligkeit einen unteren Schwellwert. Unterhalb dieses Schwellwerts ist das Kamerabild für die Farbklassifikation nicht mehr nutzbar.
- Die in Objekten erkannten Farbtöne werden nicht immer mit der Farbe des Objekts übereinstimmen. Es ist zum Beispiel nicht garantiert, dass das Bild eines blauen Tores nur blaue Farbtöne enthält.
- Die Farbräume verschiedener Farben können sich überlappen. Gelb und Orange liegen zum Beispiel direkt nebeneinander. Sie sind nicht komplett zu trennen. In einem Bild können ein gelbes Tor teilweise orange und ein orangener Ball gelb erscheinen. Um den

Ball trotzdem zu erkennen, muss der Orange-Farbraum um Teile des Gelben erweitert werden.

5.2.1 Farbklassifikation

Unter obigen Annahmen können wir für jede im RoboCup auftretende Farbe minimale und maximale Werte für H, S und V festlegen. Diese schneiden ein Stück aus dem HSV-Kegel heraus und definieren, ob ein Farbwert zu einer speziellen Farbklasse gehört oder nicht. Durch dieses Mapping sollte ein Großteil der Pixel, die in der Realität einer Farbe entsprechen[1], der korrekten Farbklasse zugeordnet werden. Gleichzeitig werden den jeweiligen Farbklassen Pixel angehören, die in Wirklichkeit nicht zu dieser Farbklasse gehören. Werden Ober- und Untergrenzen der Farbklassen groß genug gewählt, ist die Klassifikation fast unabhängig vom verwendeten Material und gegebener Beleuchtung. Ist die Beleuchtung jedoch zu dunkel, versagt die Methode, da die Helligkeit zu gering ist, alle Farben nahezu schwarz aussehen und dadurch für den Computer sehr schwer zu unterscheiden sind.

Da die obige Farbklassifikation nicht alle Pixel korrekt zuordnen kann, reicht sie zur Bilderkennung nicht aus. Die Annahme, auf der die Bilderkennung aufbaut, ist, dass man auch ohne perfekte Farbklassifizierung durch Nutzen weiterer Informationen korrekte Ergebnisse erzielen kann. Um Objekte zu erkennen, die genau zu einer Farbklasse gehören, bietet es sich an, als zusätzliche Information die erkannten Kanten zu nutzen. Ein Farbwechsel in einem Bild, zum Beispiel der Übergang von einem blauen Tor zu grünem Rasen oder weißem Pfosten, sollte durch die Kantenerkennung als Kante erkannt werden. Im Idealfall sollten alle Seiten eines farbigen Objektes durch erkannte Kanten von der Umgebung abgegrenzt sein. Genauso wie die Farbklassifikation alleine nicht ausreicht, um solche Objekte zu erkennen, reicht auch die Kantenerkennung nicht alleine aus. Ein erkanntes Viereck in der Kantenerkennung muss noch nicht Teil einer Pole sein. Erst wenn das Viereck gelb oder blau gefüllt ist, kommt es als Kandidat infrage.

5.2.2 Bloberkennung

Mit Hilfe von Farbklassifikation und Kantenerkennung werden im ersten Schritt sogenannte Blobs erkannt. Algorithmus 5.1 stellt das Vorgehen schematisch dar. Für alle Farbklassen wird nach zusammenhängenden Flächen einer Farbe gesucht. Alle diese Flächen sind potenziell Teile eines erkannten Objektes. Da die Aussagekraft der reinen Farbklassifikation wie oben besprochen nicht groß genug ist, wird von den erkannten farbigen Blobs der Schwerpunkt ermittelt. Ausgehend von diesem Schwerpunkt werden die nächstliegenden erkannten Kanten im Bild gesucht. Hierfür wird, je nach gewünschter Anzahl an Strahlen x, alle $\frac{360}{x}^\circ$ vom Schwerpunkt aus ein Strahl ins Bild projiziert. Diese Strahlen werden verfolgt, bis eine Kante gefunden oder der Bildrand erreicht ist. Verbindet man die gefunden Kanten, ergibt sich ein neuer Blob.

[1] Tore: blau oder gelb, Landmarken: blau und gelb, Bälle: orange und so weiter.

Eingabe : Binarisierte farbklassifizierte Bilder für jede Farbklasse und ein binarisiertes Bild der Kanten
Ausgabe : Für jede Farbklasse eine Liste der erkannten Blobs

```
foreach Farbklasse i do
    Finde Blobs in farbklassifizierten Bildern(i);
    foreach FarbBlob j do
        Berechne Schwerpunkt(j);
        Suche Kanten ausgehend vom Schwerpunkt;
        Erzeuge Blob anhand erkannter Kanten;
        if farbklassifizierter Blob und Blob aus Kanten stimmen überein then
            speichere Blob;
        else
            verwerfe Blob;
        end
    end
    Kombiniere direkt nebeneinanderliegende/sich überlappende Blobs;
    Verwerfe nicht plausible Randpunkte;
    Verwerfe Blobs, die bestimmte Kriterien nicht erfüllen;
end
```

Algorithmus 5.1 : Bloberkennung anhand von farbklassifizierten Bildern und Kantenerkennung

Ein erster Hinweis, ob der Blob wirklich einem Teil eines gefundenen Objektes entspricht, ist, ob die über die Farbklassifikation und die Kantenerkennung erzeugten Blobs zusammenpassen. Im Allgemeinen müsste der Kantenerkennungsblob größer oder gleich groß sein wie der Farbblob. Die Größe darf sich jedoch nicht zu sehr unterscheiden, da es somit sehr wahrscheinlich ist, dass entweder die Kanten des Objekts nicht richtig erkannt wurden oder eine farbliche Fehlklassifikation vorliegt. Nachdem alle potenziellen Blobs für eine Farbklasse bearbeitet wurden, können sich überlappende oder direkt nebeneinanderliegende Blobs vereinigt werden, da sie dem gleichen Objekt entsprechen müssen. Es gibt im RoboCup keine Objekte gleicher Farbe, die direkt nebeneinanderliegen.

Die jetzt erkannten Blobs enthalten immer noch Fehlerkennungen. Einige der Fehlerkennungen können jedoch sofort erkannt und entfernt werden. Hierfür sind einige Überlegungen nötig: Alle Objekte, die im RoboCup auftreten und mit Hilfe von Blobs erkannt werden können, sind entweder rechteckig oder rund. Objekte, wie die Segmente einer Pole, sind nicht nur rechteckig, sondern auch quadratisch. Wenn man die Standardabweichung und Varianz von Blobs kreisförmiger Objekte betrachtet, fällt Folgendes auf: Standardabweichung und Varianz sind null. Für quadratische Objekte sind beide ungleich null, aber klein. Je mehr sich bei einem Rechteck die Seitenverhältnisse unterscheiden, desto größer werden die Werte. Blobs, die keine Ähnlichkeit zu Quadraten haben oder aus sehr vielen Ausreißerpunkten bestehen, haben eine große Standardabweichung und/oder Varianz. Je nachdem, welche Objekte erkannt werden sollen, können Objekte mit großer Varianz/Standardabweichung verworfen werden. Genauso können Objekte mit unpassenden Größen verworfen werden.

Soll ein Tor erkannt werden, können kleine Blobs verworfen werden, geht es um einen Ball große Blobs.

5.2.3 Erkenner

Nach Farbklassifikation und Bloberkennung können die Daten von den einzelnen Erkennern genutzt werden. Erkenner, auch Perzeptoren genannt, sind dafür zuständig, aus den Eingangsdaten Objekte und Informationen wie deren Position oder Größe zu extrahieren. Im Folgenden werden Konzepte für verschiedene Erkenner beschrieben. Im Rahmen dieser Arbeit wurde jedoch nur die Poleserkennung vollständig implementiert und getestet.

Polesperzeptor

Die im RoboCup genutzten runden Poles bestehen aus drei gleich großen, etwa rechteckigen Teilen. Sie sind entweder blau-gelb-blau oder gelb-blau-gelb eingefärbt. Aus Sicht der Bilderkennung besteht eine Pole aus drei etwa gleich großen, entsprechend eingefärbten, sich berührenden Blobs, deren Schwerpunkte übereinander liegen. Unter dem untersten Blob befindet sich je nach Blickwinkel entweder grüner Rasen oder eine weiße Linie.

Sind alle obigen Bedingungen erfüllt, wurde mit großer Wahrscheinlichkeit eine Pole gefunden. Die Chance, dass diese Kombination auftritt und eine Pole fälschlich erkannt wurde, ist in einer RoboCup-Umgebung sehr gering. Ergebnisse der Implementierung werden in Kapitel 7 betrachtet.

Torperzeptor

Zur Erkennung eines Tores kann man erkannte Blobs der Größe nach bearbeiten, da ein Tor im Bild im Allgemeinen dem größten gelben oder blauen Blob entspricht.

Um einen Blob zu verifizieren, bieten sich verschiedene Tests an. Prüfen der Abmessungen: Höhe und Breite des Tores sind bekannt, der Blob sollte diesen, abhängig vom Bildausschnitt, entsprechen. Steht der Roboter nicht direkt vor dem Tor und sieht daher nicht nur gelb oder blau, ist mindestens ein Torpfosten im Bild sichtbar. Ein Pfosten ist zu schmal und wird nicht als Blob erkannt, entspricht aber einem auf einer Gerade liegenden Übergang von der Torfarbe zu Weiß auf der linken und/oder rechten Seite des Blobs. An der Unterkante des Blobs muss sich ein auf einer Linie liegender Übergang von der Torfarbe zu Grün befinden. An der Oberkante des Tores findet kein definierter Farbübergang statt, allerdings müssen auch hier die erkannten Randpunkte etwa auf einer Linie liegen.

Nicht immer kann ein Tor mit einem einzigen Blob erkannt werden. Stehen Hindernisse im Blickfeld vor dem Tor, zum Beispiel der Torwart, zerfällt ein Tor in mehrere farbige Blobs. Die Erkennung muss diese entsprechend zusammenführen.

Linienperzeptor

Die Linienerkennung kann die Bloberkennung nicht nutzen, da die Linien relativ schmal, oft nur wenige Pixel breit, sind. Es bietet sich an, eine Kombination aus Kantenerkennung und

Farbklassifikation zu nutzen.

Im ersten Schritt wird eine Kantenerkennung auf dem Bild durchgeführt. Die erkannten Kanten werden dazu genutzt, alle im Bild zu erkennenden Linien, unabhängig davon, ob es Feldlinien sind oder nicht, zu extrahieren. Hierzu kann zum Beispiel die Hough-Transformation genutzt werden [36]. Nach der Linienerkennung müssen aus den erkannten Linien die Feldlinien erzeugt werden. Hierfür kann man sich einige Eigenschaften von Feldlinien zunutze machen: Vom Roboter aus gesehen nahe Feldlinien bestehen immer aus zwei im Bild erkennbaren Linien. Die eine Linie trennt einen weißen von einem grünen, die andere Linie einen grünen von einem weißen Bereich. Beide erkannten Linien sind in etwa parallel. Weiter entfernte Feldlinien sind nur noch als eine Kante im Bild erkennbar. Diese Kante ist von einem dünnen weißen Bereich umgeben, der eine grüne Fläche teilt.

Zwei zusammenpassende Linien[2] oder eine weiter entfernte Linie, die obige Kriterien erfüllt, entsprechen also einer Feldlinie.

Ballperzeptor

Als Ball kommen alle orangefarbenen Blobs infrage. Da die Ballgröße und die genaue Entfernung des Blobs vom Roboter bekannt sind, kann berechnet werden, wie groß ein Ball an der Position des Blobs im Bild sein muss. Unterscheidet sich die Größe des Blobs signifikant von der erwarteten Größe des Balls, kann der Blob verworfen werden. Stimmen beide Größen in etwa überein, muss überprüft werden, ob der Blob ein erkannter Ball sein kann. Hierfür muss der Blob von einer grün oder weiß klassifizierten Fläche umgeben sein. Zusätzlich müssen die Übergänge von Orange nach Grün/Weiß einen Kreis beziehungsweise in der Praxis aufgrund der Bewegung des Roboters eher ein Oval bilden.

[2] Da die Breite einer Feldlinie bekannt ist, kann der genaue Abstand von zwei Linien die eine valide Feldlinie bilden, bestimmt werden.

Kapitel 6

Realisierung

Aufbauend auf *RoboFrame* wurde die gesamte Arbeit in objektorientiertem C++ implementiert. Die Realisierung orientiert sich an den Anforderungen aus Kapitel 4 und dem erarbeiteten Konzept aus Kapitel 5.

In diesem Kapitel werden die Realisierung der Evaluationsinfrastruktur, darauf aufbauende Module und die neu geschriebene Bildverarbeitung erläutert. Die Klassenstrukturen werden anhand von UML-Diagrammen verdeutlicht. Auf die Darstellung von privaten Methoden und Variablen wird allerdings verzichtet, falls diese für die Erläuterungen nicht relevant sind. Ebenso wird aus Platzgründen auf konstante Referenzen und die doppelte Darstellung von konstanten und nicht konstanten Methoden verzichtet. Auch Namespaces werden nicht immer komplett ausgeschrieben.

Zur einfacheren und effektiveren Entwicklung mit C++ wurden im Rahmen dieser Arbeit die Boost C++ Bibliotheken [37] eingebunden. Sie können jetzt in jedem auf *RoboFrame* basierenden Projekt genutzt werden. Die Boost C++ Bibliotheken, im Weiteren nur noch Boost genannt, sind eine Sammlung von freien portablen Bibliotheken[1], mit dem Ziel den C++-Standard zu erweitern und die Produktivität zu steigern. Zehn Bibliotheken, die Teil von Boost sind, wurden in den C++-Library Technical Report 1 (TR1) [38] aufgenommen und sind damit ein Teil der Standard-Bibliothek des kommenden C++1x-Standards. Im Anhang B.1 werden einige der genutzten Funktionen vorgestellt.

Der gesamte Code wurde direkt im Robocode-Repository der *Darmstadt Dribblers* entwickelt und in *RoboFrame* sowie die genutzte GUI-Applikation eingebunden.

6.1 Evaluator Dialog

Wie in Kapitel 5 beschrieben ist der für den Nutzer sichtbare Teil der Infrastruktur ein Dialog, der von der *RoboFrame*-Klasse `robogui::Dialog` erbt. Durch das Erben von `robogui::Dialog` kann der Dialog wie ein normales `QWidget` benutzt werden. Gleichzeitig ist er jedoch vollständig in eine mit *RoboFrame* realisierte GUI eingebunden. Hierdurch ist es zum Beispiel möglich, Daten zu versenden und zu empfangen. Wie man in Abbildung 6.1 erkennt, hat die `EvaluatorDlg`-Klasse ein minimales Interface. Sie reimplementiert nur

[1] Die *Boost Software License* (`http://www.boost.org/LICENSE_1_0.txt`) erlaubt die kommerzielle und nicht-kommerzielle kostenlose Verbreitung, Nutzung und Modifikation.

Methoden der `roboapp::Dialog`-Klasse, die zur eigentlichen Implementierung genutzten Methoden sind nicht sichtbar, auch nicht als private Methoden. Möglich wird dies durch die Nutzung des sogenannten PIMPL-Idioms [39]. Es sorgt dafür, dass das Interface und die eigentliche Implementierung einer Klasse getrennt sind. Dies erhöht die Kapselung und Wiederverwertbarkeit einer Klasse. Alle im Rahmen dieser Arbeit geschriebenen Dialoge verwenden diese Technik. Eine genauere Erklärung des PIMPL-Idioms befindet sich im Anhang C.

Im nächsten Abschnitt werden erst die Basis der Evaluationsinfrastruktur Storages, Evaluatoren und ihre Interfaces beschrieben. Im Abschnitt danach wird auf den genauen Aufbau der des Dialogs, im Weiteren nur noch GUI genannt, eingegangen.

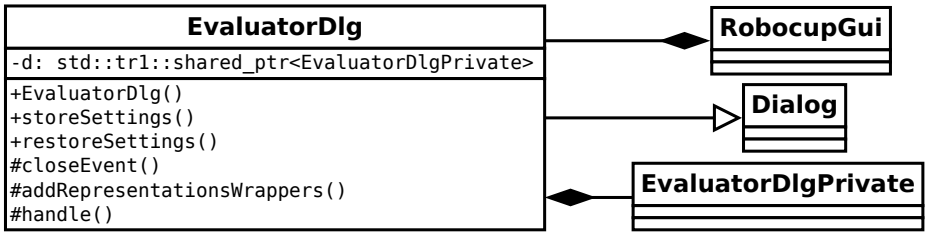


Abbildung 6.1: Integration des Evaluator Dialogs in *RoboGui*

6.1.1 Storages und Evaluatoren

Damit der Dialog beliebige Storages und Evaluatoren nutzen kann ohne ihre konkreten Implementierungen zu kennen, erben diese von abstrakten Klassen, die ein Interface zur Kommunikation bereitstellen.

6.1.1.1 AbstractStorage

`AbstractStorage` stellt ein Interface zum Abfragen und Hinzufügen von Informationen aus einem Storageobjekt zur Verfügung. Ein Großteil des Interfaces dient nur zur Kommunikation zwischen GUI und Storage. Konkrete Storages kommunizieren mit den passenden Evaluatoren über spezielle, nicht generische Interfaces. Beispiele für konkrete Storages und Evaluatoren werden in 7.1 beschrieben.

Der `AbstractStorage`-Konstruktor erwartet als Parameter einen Zeiger zu einem Objekt welches `roboapp::IConnectorPool` implementiert und einer Instanz der Klasse `roboapp::StreamedDataQueue`. Der `ConnectorPool` wird benötigt, um neue Connectoren erstellen zu können. Diese ordnen Daten aus verschiedenen Quellen beim Abspielen wieder verschiedenen Quellen zu. Eine Storage, die diese Funktion nutzt, ist weiter unten in 7.1.1 beschrieben. Die `StreamedDataQueue` wird benötigt, um Daten wieder verschicken zu können.

Die Storages gruppieren Daten wie in Kapitel 5 besprochen nach Datenpaketen. Mit den Methoden `columnHeader`, `rowCount` und `data` können allgemeine Informationen über die

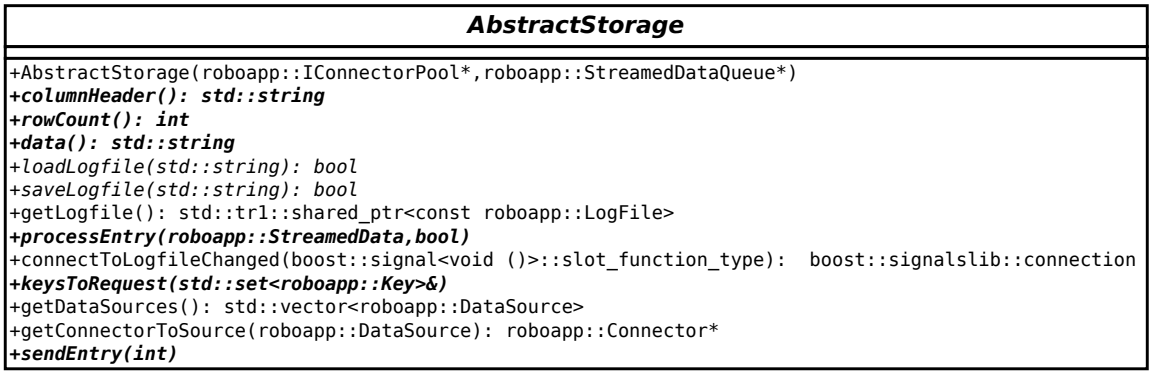


Abbildung 6.2: Schnittstellen der abstrakten Storageklasse

Datenpakete abgefragt werden. `columnHeader` liefert einen String zurück, der in der GUI über den einzelnen Einträgen der Datenpakete angezeigt wird. Storages, die Daten nach Zeitstempel gruppieren, könnten zum Beispiel *Timestamps* zurück liefern[2]. Mit `rowCount` kann man die Gesamtzahl an gruppierten Datenpaketen abfragen. Um jedes dieser Pakete visualisieren zu können, kann man zu jedem Datenpaket mit `data` einen String abfragen. Eine Storage, die nach Zeitstempeln gruppiert, könnte hier beispielsweise den Zeitstempel als String zurück liefern. Datenpakete werden von null bis `rowCount()-1` nummeriert.

Alle Storages basieren auf einem `roboapp::Logfile`. Eine bestehende Logdatei kann mit `loadLogfile` und einem Pfad als Parameter geladen werden. Die Defaultimplementierung ruft auf jedem Eintrag der Logdatei `processEntry` auf. Mit `saveLogfile` werden eventuelle Änderungen in der Logdatei gespeichert. Via `saveLogfile(std::string)` ist es möglich eine Kopie der benutzen Logdatei unter einem gegebenem Pfad zu speichern. Die Storage arbeitet danach auf der Kopie.

RoboFrame verschickt und speichert Daten serialisiert in Form von `roboapp::StreamedData`-Objekten. Alle ausgetauschten Daten zwischen Modulen im Framework und alle in Logdateien gespeicherten Daten sind `roboapp::StreamedData`-Objekte. Um ein `StreamedData`-Objekt zu verarbeiten, stellt `AbstractStorage` die Methode `processEntry` zur Verfügung. Als Parameter bekommt die Methode neben dem Datenpaket einen Boolean übergeben . Dieser gibt an, ob das Datenpaket bereits in der Logdatei gespeichert ist oder nicht. Dies ermöglicht für das Verarbeiten einer Logdatei und das Verarbeiten von neuen Daten, die von der GUI empfangen wurden, den gleichen Code zu nutzen. Der einzige Unterschied ist, dass die Storage die Möglichkeit hat, Daten, die nicht aus der Logdatei kommen, in dieser zu speichern. Wie genau die Verarbeitung eines `StreamedData`-Objekte aussieht, ist nicht festgelegt und Storage spezifisch. Ebenso ob nicht in der Logdatei gespeicherte Objekte in dieser gespeichert werden oder nicht.

Sobald eine Storage eine neue Logdatei geladen hat, sind alle Informationen, die vorher von der Storage abgefragt wurden, ungültig. Sowohl die GUI als auch die zugehörigen Evaluatoren müssen sich neu initialisieren und die Informationen erneut abfragen. Alle

[2] Zu Zeitstempeln siehe auch 6.1.3.1.

Objekte, die an der Information interessiert sind, ob eine neue Logdatei geladen wurde, können sich mit `connectToLogfileChanged` bei der Storage registrieren. Sie werden mithilfe eines Boost Signals benachrichtigt, sobald eine neue Logdatei geladen wurde. Genauere Informationen zu Boost Signals finden sich im Anhang B.1.3.

Eine Storage kann Interesse daran haben, neben den in der Logdatei vorhandenen Daten weitere Daten aufzunehmen. Da eine Storage selbst keine Daten beim Framework anfordern kann, kann die GUI über `keysToRequest` die Keys abfragen, an denen die Storage interessiert ist, und diese beim Framework anfordern. Sobald entsprechende Daten eintreffen, werden diese an `processEntry` zur Verarbeitung weiter gegeben.

Mithilfe des im Konstruktor übergebenen `IConnectorPools` und der `StreamedDataQueue` hat die Storage die Möglichkeit Datenpakete zu verschicken. Über die Methode `sendEntry` kann sie aufgefordert werden ein bestimmtes Datenpaket oder je nach Storage und Einstellung Teile davon, zu verschicken.

6.1.1.2 AbstractEvaluator

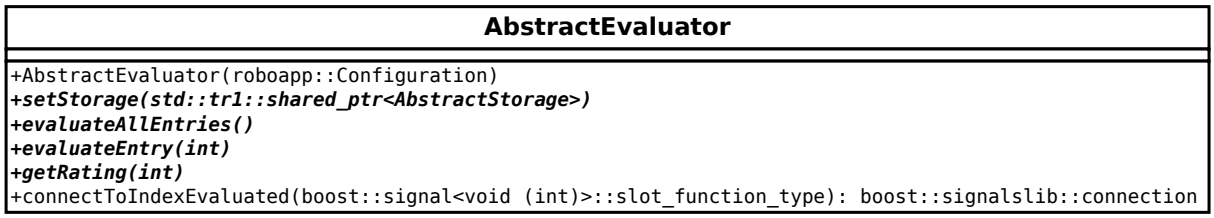


Abbildung 6.3: Schnittstellen der abstrakten Evaluatorklasse

Alle Evaluatoren erben von der Klasse `AbstractEvaluator`. Das zur Verfügung gestellte Interface ermöglicht es der GUI den Evaluator zu steuern und Informationen von diesem abzufragen.

Der Konstruktor jedes Evaluators bekommt ein `roboapp::Configuration`-Objekt übergeben. Über dieses können Evaluatoren von der GUI verschieden konfiguriert werden. Wie genau die Konfiguration der konkreten Evaluatoren funktioniert wird weiter unten besprochen.

Das Setzen der zu verwendenden Storage erfolgt über die Methode `setStorage`. Jeder Evaluator arbeitet mit genau einem konkreten Storage-Typ zusammen. Da der Storage-Typ nicht aus dem Interface hervorgeht, überprüft jeder Evaluator, ob ihm ein passender Typ übergeben wurde. Durch die Verwendung der in 6.1.1.3 beschriebenen Factorys wird sicher gestellt, dass von der GUI nur zu den Storages passende Evaluatoren erzeugt werden.

Mit `evaluateAllEntries` und `evaluateEntry` kann ein Evaluator dazu aufgefordert werden, alle oder ein spezielles Datenpaket der Storage auszuwerten. Mittels `getRating` kann abgefragt werden, ob ein Datenpaket bereits evaluiert wurde (Rückgabewert -1) oder alternativ, wie interessant ein Datenpaket ist. Hierfür geben Evaluatoren einen Wert zwischen null und eins zurück, der von der GUI für die Visualisierung genutzt wird. Hierzu später

mehr. `evaluateEntry` und `getRating` müssen threadsicher sein, da sie später aus der GUI von verschiedenen parallel laufenden Threads genutzt werden.

Die Evaluierung eines Eintrags kann beliebig lange dauern. Um ein Blockieren der GUI zu vermeiden, wird `evaluateAllEntries` in einem separaten Thread ausgeführt. Die GUI registriert sich mit `connectToIndexEvaluated` bei den Evaluatoren, um benachrichtigt zu werden, sobald ein Datenpaket fertig evaluiert ist. Diese triggern ein Signal mit dem Index des evaluierten Pakets, sobald die Auswertung abgeschlossen ist.

6.1.1.3 Storage und Evaluator Factory

Die GUI nutzt nur die oben beschrieben abstrakten Klassen und ist unabhängig von konkreten Implementierungen. Die konkreten Implementierungen werden der GUI zur Kompilierzeit über eine sogenannte Factory bekannt gemacht. Über die Factory kann die GUI abfragen, welche Storages und zugehörige Evaluatoren existieren und Instanzen von diesen anfordern.

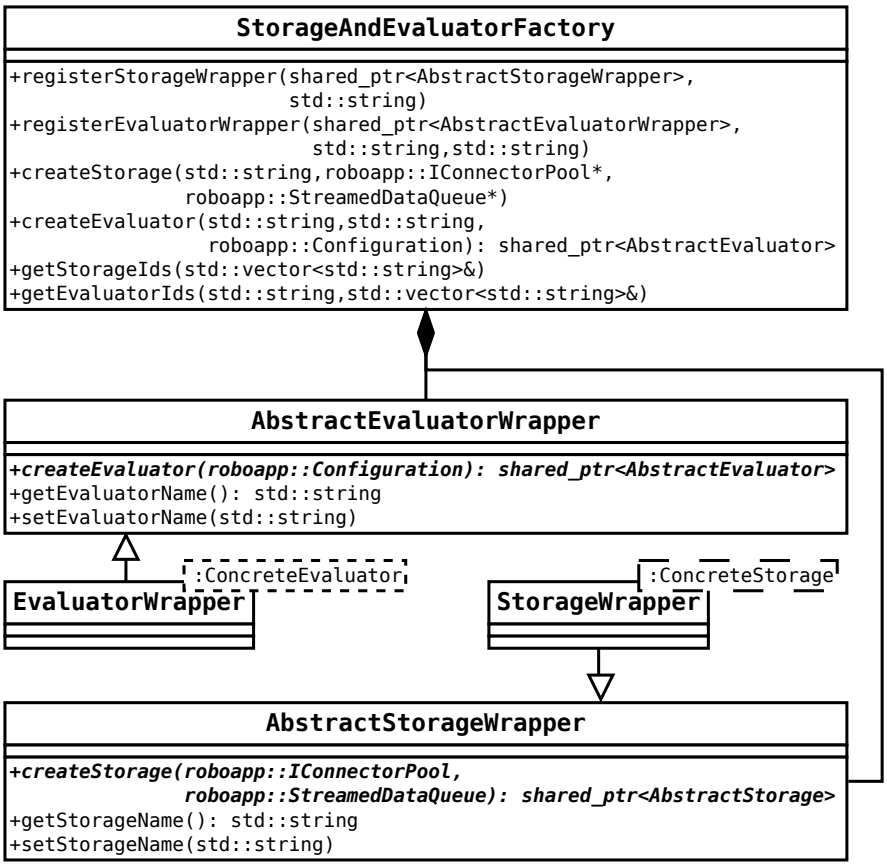


Abbildung 6.4: Factory zum Erzeugen von Storages und Evaluatoren

Bevor Storages erzeugt werden können, müssen diese über `registerStorageWrapper` bei der Factory registriert werden. Hierfür wird der GUI ein Objekt übergeben, welches das von `AbstractStorageWrapper` vorgegebene Interface implementiert. Das konkrete

Objekt ist eine Templateklasse, die mit Hilfe von `createStorage` Instanzen eines konkreten Storageobjektes erzeugen kann. Zusätzlich wird der Factory ein String übergeben, der die Storage eindeutig identifiziert.

Die Registrierung der Evaluatoren funktioniert analog, allerdings wird der GUI ein eindeutiger String für den Evaluator und der String der zugehörigen Storage mitgeteilt. Hierdurch weiß die GUI, welche Evaluatoren zu welcher Storage gehören und bietet nur valide Kombinationen an.

Sobald Storages und Evaluatoren registriert sind, kann die GUI die eindeutige Strings der registrierten Klassen mit `getStorageIds` beziehungsweise `getEvaluatorIds` abfragen. `getEvaluatorIds` bekommt als Parameter hierfür die Id einer Storage übergeben und liefert nur die zu dieser Storage passenden Evaluatoren zurück.

Registrierte Storages können mit der Methode `createStorage` erzeugt werden. Parameter sind die Id der Storage sowie der `ConnectorPool` und die `StreamedDataQueue` der GUI. Die letzten beiden Parameter werden direkt an den Konstruktor der Storage weiter gereicht (siehe 6.1.1.1). Evaluatoren können über `createEvaluator` instanziiert werden. Parameter hierbei sind die Ids des Evaluators und der zugehörigen Storage sowie ein `roboapp::Configuration`-Objekt, das an den Konstruktor des Evaluators weiter gegeben wird. Wie die Konfiguration erstellt wird, ist in 6.1.1.5 beschrieben.

6.1.1.4 Storageabhängige Erweiterung der GUI

Für verschiedene konkrete Storages kann es nötig sein, verschiedene Möglichkeiten der Interaktion über die GUI bereitzustellen. Da die GUI nichts über die konkreten Storages weiß, gibt es eine weitere Factory, die es erlaubt je nach Storage Buttons zu einer Toolbar hinzuzufügen. Die Implementierung der Factory ist analog zur in 6.1.1.3 vorgestellten Storage und Evaluator Factory, daher zeigt das UML-Diagramm in Bild 6.5 nur die Interfaces der GUI-Erweiterungsklasse und deren Wrapper. Um sicherzustellen, dass GUI-Erweiterung und Storage zusammenpassen, wird bei der Registrierung wieder die Id der passenden Storage übergeben. Wird eine neue Storage erzeugt, fragt die GUI die Factory, ob eine entsprechende GUI-Erweiterung vorhanden ist, und lässt diese falls vorhanden instanziieren.

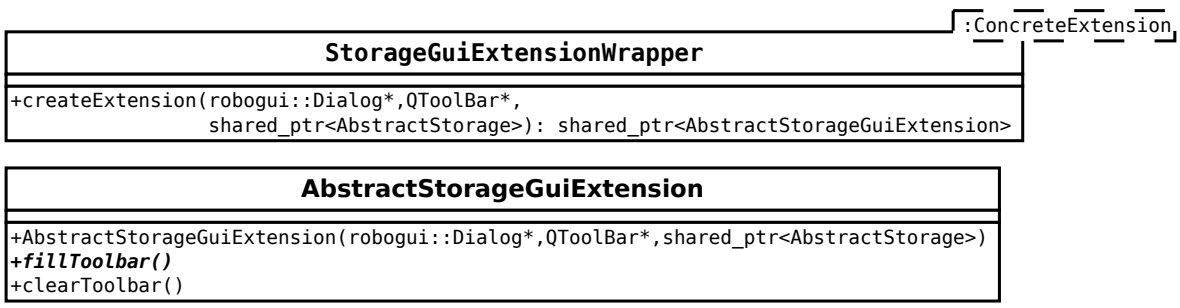


Abbildung 6.5: Storageabhängige Erweiterung der GUI

Die zu erzeugende Klasse bekommt als Parameter einen Zeiger auf die GUI, einen Zeiger auf die `QToolBar` die sie erweitern kann und einen Zeiger auf die konkrete Storage-Instanz

übergeben. Mit `fillToolbar` kann die GUI die GUI-Erweiterung dazu auffordern die Toolbar zu erweitern. Wie genau sie das tut, ist der Erweiterung überlassen. Ändert sich die Storage oder will die GUI aus einem anderen Grund die Toolbar wieder bereinigen kann sie die Klasse mit `clearToolbar` dazu auffordern, ihre Elemente wieder von der Toolbar zu entfernen.

Da die aktuelle Storage bekannt ist, kann die Erweiterung die Storage abhängig von den Benutzereingaben über die Toolbar manipulieren. Um sinnvoll auf der Storage arbeiten zu können, sollte die Klasse die übergebene `AbstractStorage` auf den Storage-Typ den sie unterstützt casten.

6.1.1.5 Evaluatorkonfiguration

Ein Evaluator kann verschiedene variable Parameter haben. Es bietet sich an zu ermöglichen, Parameter zu setzen, wenn der Evaluator erzeugt wird und nachdem der Evaluator erzeugt wurde. Auch hierfür steht eine Factory zur Verfügung. Der Aufbau ist wieder analog zu den beiden vorhergehenden Factorys. Diese erlaubt jedoch, zu jeder Kombination aus Evaluator und Storage, Dialoge zur Konfiguration zum Erstellungszeitpunkt und zur Konfiguration zur Laufzeit zu registrieren. Zu jedem Evaluator können also zwei passende Dialoge registriert werden.

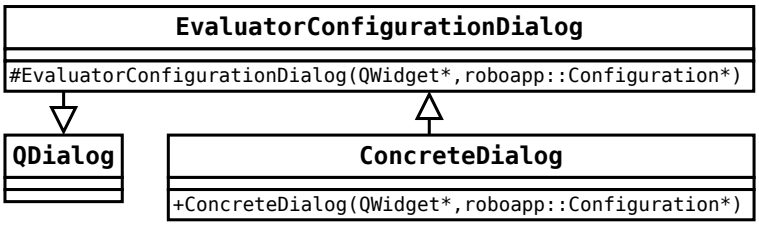


Abbildung 6.6: Basisklasse für Evaluatorkonfigurationsdialoge und konkrete Implementierungen

Um Parameter zwischen den Dialogen und Evaluatoren austauschen zu können wird die Klasse `roboapp::Configuration` genutzt. Diese erlaubt es Paare aus einem String und einem Objekt (String, Zahl, Boolean) zu gruppieren und auszutauschen.

Bevor ein neuer Evaluator erzeugt wird, fragt die GUI die Factory ab, ob ein Konfigurationsdialog zum Erstellungszeitpunkt registriert wurde. Wenn ja legt sie eine neue Instanz einer `roboapp::Configuration` an, übergibt diese an die Factory, die einen konkreten Dialog erzeugt, welcher die Konfiguration übergeben bekommt. Der Dialog wird von der GUI ausgeführt und füllt abhängig von den Eingaben des Nutzers das Konfigurationsobjekt. Wie in Bild 6.6 zu erkennen ist, erbt ein Konfigurationsdialog von `QDialog` und kann daher ein beliebiges Userinterface implementieren.

Alle Evaluatoren erlauben es, ihnen über den Konstruktor ein Konfigurationsobjekt zu übergeben (siehe 6.3). Evaluatoren, die eine Rekonfiguration zur Laufzeit erlauben wollen, müssen das `AbstractRuntimeConfigurableEvaluator`-Interface implementieren, indem sie von dieser Klasse erben. Wenn ein Evaluator von dieser Klasse erbt und ein

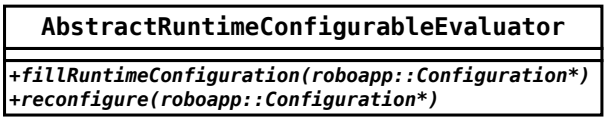


Abbildung 6.7: Interface für zur Laufzeit konfigurierbare Evaluatoren

passender Dialog registriert wurde, ermöglicht die GUI es, den Evaluator zur Laufzeit zu rekonfigurieren. Hierfür erstellt sie ein Konfigurationsobjekt und übergibt dieses über `fillRuntimeConfiguration` an den existierenden Evaluator. Dieser kann das Objekt nach Belieben befüllen, um Informationen an den registrierten Dialog weiterzugeben. Analog zum Konfigurationsdialog zum Erstellungszeitpunkt wird jetzt der Laufzeitdialog erstellt und kann auf dem übergebenen Konfigurationsobjekt arbeiten. Nachdem der Dialog die Konfiguration angepasst hat, wird der Evaluator über `reconfigure` dazu aufgefordert seinen internen Zustand zu aktualisieren.

6.1.2 Implementierung des Evaluatordialogs

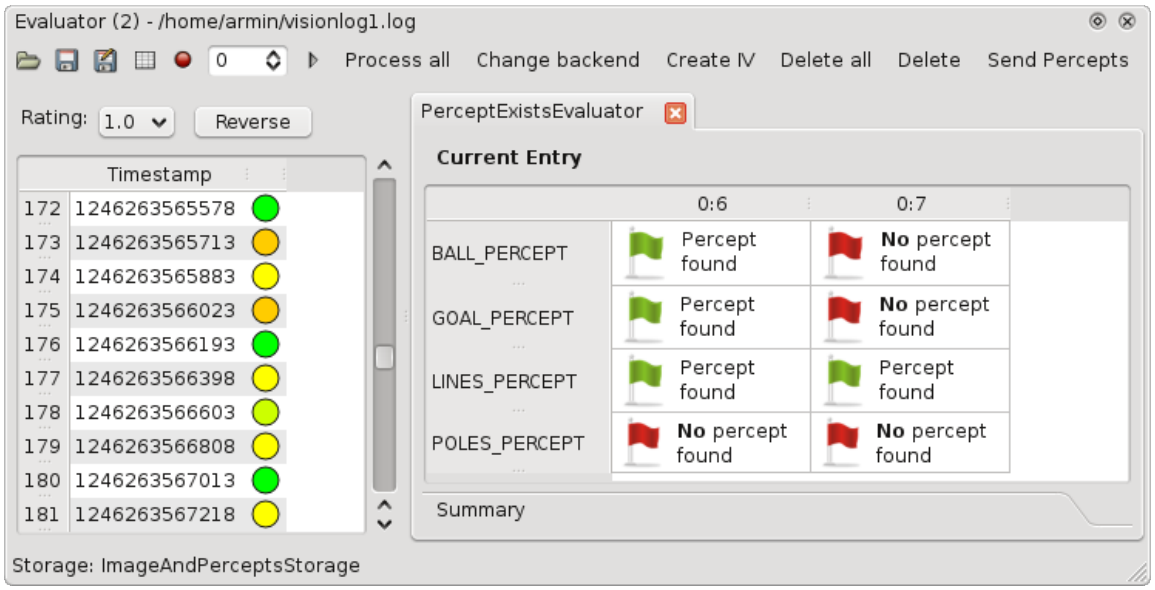


Abbildung 6.8: Screenshot des Evaluatordialogs mit einer Storage und einem Evaluator

Der Evaluator Dialog besteht aus Sicht des Nutzers aus drei wichtigen Komponenten: Einem Bereich in dem die Storage visualisiert wird und einzelne Datenpakete ausgewählt werden können, einem Bereich indem die Evaluatoren visualisiert werden und einer Toolbar. Im Folgenden werden diese Komponenten ihre Interaktion und die Einbindung der in 6.1.1 vorgestellten Klassen erläutert.

6.1.2.1 Model/View-Architektur

Um Daten, ihre Darstellung und die Interaktion mit ihnen zu entkoppeln, wird häufig das Design Pattern Model-View-Controller (MVC) genutzt. Seinen Ursprung findet das Pattern im GUI-Design der Sprache Smalltalk [40]. Durch die Entkopplung ist es möglich, verschiedene Daten mit dem gleichen Code darzustellen, gleiche Daten auf verschiedene Arten und Weisen darzustellen oder einzelne Komponenten wieder zu verwenden und einfach zu testen.

Qt vereinfacht dieses Pattern und legt View und Controller zusammen [41]. Es ergibt sich eine Model/View-Architektur, die genau wie MVC für eine saubere Trennung von Model, also Daten und View, der Darstellung sorgt. Qt erlaubt es, beliebige Daten mit den gleichen View-Klassen darzustellen. Um jedoch auch eine individuelle Darstellung und Interaktion mit den Daten zu erlauben, werden sogenannte *Delegates* eingeführt. Mit diesen ist es möglich, Daten beliebig zu zeichnen und zu editieren. Geht es jedoch nur darum Standarddatentypen wie Strings oder Zahlen anzuzeigen, wird der Standard-*Delegate* genutzt.

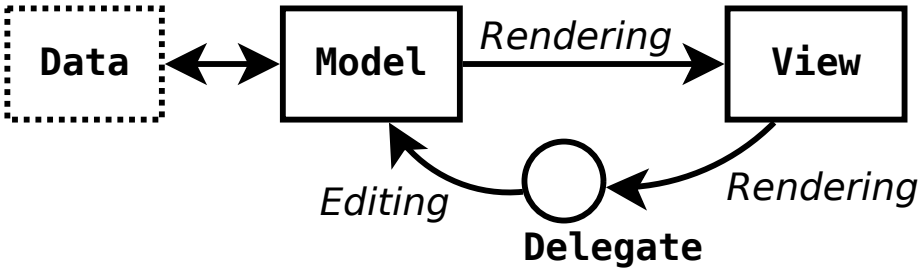


Abbildung 6.9: Model/View-Architektur in Qt. Grafik basierend auf `http://doc.trolltech.com/4.4/images/modelview-overview.png`

Die drei Komponenten Model, View und Delegate werden jeweils durch die abstrakten Klassen `QAbstractItemModel`, `QAbstractItemView` und `QAbstractItemDelegate` definiert. Um neue Models, Views oder Delegates zur Verfügung zu stellen, muss von diesen Klassen geerbt werden. Qt stellt jedoch schon einige Implementierungen zur Verfügung, auf denen aufgebaut werden kann. Beispielsweise `QListView` um Listen, `QTableView` um Tabellen oder `QTreeView` um Baumstrukturen darzustellen. Passend dazu gibt es auch die Modelle `QAbstractListModel` und `QAbstractTableModel` von denen geerbt werden sollte, wenn die Daten in Listen oder Tabellenform vorliegen.

Um die angezeigten Daten eines Modells zu filtern, können ein oder mehrere *Proxy Models* zwischen das eigentliche Model und den View geschaltet werden. Alle *Proxy Models* erben von `QAbstractItemView`. Qt stellt jedoch die Klasse `QSortFilterProxyModel` zur Verfügung, die um Code für eigene Filter- und Sortierroutinen erweitert werden kann.

6.1.2.2 Storage View und Modeling

Die Visualisierung der Storage zeigt eine Liste aller in der Storage gespeicherten Datenpakete an. Als textuelle Repräsentation wird der von der Storage mit `data` abgefragte String genutzt. Jeder aktive Evaluator kann Bewertungen für jedes Datenpaket abgeben. Diese

Bewertungen werden neben dem String eines Datenpakets als gefüllte Kreise dargestellt. Ein nicht evaluiertes Datenpaket wird als Kreis mit weißer Füllung dargestellt. Evaluierte Datenpakete erhalten von den Evaluatoren einen Wert zwischen null und eins. Null entspricht in der Darstellung rot, eins grün. Die Farbwerte werden je nach Bewertung zwischen diesen beiden Farben interpoliert.

Textuelle Repräsentation und Rating können in Form einer Tabelle dargestellt werden. Jedes Datenpaket erhält eine Zeile, textuelle Repräsentation und das Rating jedes Evaluators entsprechen einer Spalte. Wie in 6.1.2.1 beschrieben wird die Qt Klasse `QTableView` zur Visualisierung genutzt. Die Klasse kann unverändert genutzt werden. Es müssen nur ein passendes Modell zur Umwandlung von Storage und Rating der Evaluatoren sowie ein Delegate zum Anzeigen der Ratings geschrieben werden.

`SelectionTableModel` und Visualisierung

Das `SelectionTableModel` ist ein dünner Wrapper um die aktuelle Storage und die aktuellen Evaluatoren welcher die Daten in ein Format konvertiert, das `QTableView` versteht. Die folgenden Methoden der Basisklasse `QAbstractModel` werden hierfür überschrieben: `rowCount`, `columnCount`, `data` und `headerData`. Über `rowCount` fragt der View die Anzahl der Tabellenzeilen vom Model ab. Das Model reicht direkt den Wert, den die Storage mit ihrer `rowCount`-Methode liefert, weiter. Jedes Datenpaket der Storage bekommt also eine Zeile. Der `columnCount`, also die Anzahl der Spalten, entspricht wie oben besprochen der Anzahl an registrierten Evaluatoren plus eins. Über `headerData` fragt der View die gewünschte Spalten- und Zeilenbeschriftung ab. Für Zeilen wird die Zeilennummer, die auch dem Index des Datenpakets in der Storage entspricht, zurückgegeben. Die erste Spalte erhält als Beschriftung den von der Storage mittels `columnHeader` abgefragten String. Die Spalten, die das Rating der Evaluatoren enthalten, werden nicht beschriftet.

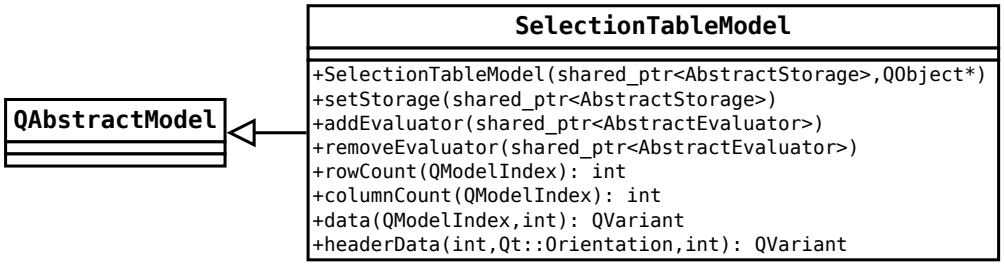


Abbildung 6.10: Wrapper um Storage und Evaluatoren

Am Wichtigsten ist die Methode `data`. Mit dieser fragt der View die anzuzeigenden Daten ab. Qt erwartet als Rückgabewert der Methode einen `QVariant`. Diese Klasse kapselt, ähnlich wie eine Union, Qt bekannte Datentypen. Unbekannte Datentypen können über `Q_DECLARE_METATYPE` bei Qt registriert werden. Fragt der View ein Element aus der ersten Spalte ab, entspricht dieses der textuellen Repräsentation, die die Storage mit `data` und der Zeilennummer als Parameter zurück liefert. Dieser `std::string` kann in einen `QString` umgewandelt werden und wird dann von Qt direkt verstanden. Für die

Evaluatorratings funktioniert dies nicht so einfach. Die von den Evaluatoren zurückgelieferten Werte sollen in Form von farbigen Kreisen dargestellt werden, Qt weiß jedoch nichts von der Umwandlung von Zahlen in Kreise. Um die Ratings zu visualisieren, wird eine Klasse Namens `EvaluatorRating` genutzt. Diese weiß, wie man einen gegebenen Wert visualisiert. Statt einem `std::string` gibt das Model also ein `EvaluatorRating` zurück. Damit ein `QVariant` ein `EvaluatorRating` akzeptiert muss dieses mit `Q_DECLARE_METATYPE(EvaluatorRating)` registriert werden. Der genutzte `QTableView` bekommt als Delegate einen `EvaluatorRatingDelegate` gesetzt. Dieser Delegate kennt `EvaluatorRating` und ruft für Objekte dieses Typs die passenden Methoden zum Zeichnen auf. Alle anderen Objekte gibt er direkt an Qt weiter, welches diese entsprechend zeichnet.

Neben den reimplementierten Methoden gibt es auch ein Interface, über das die GUI mit dem `SelectionTableModel` kommunizieren kann. Dieses ermöglicht mit `setStorage` die Storage auf der das Model arbeitet zu setzen sowie mit `addEvaluator` beziehungsweise `removeEvaluator` Evaluatoren hinzuzufügen oder zu entfernen. Das Model registriert sich bei der Storage, um über ein geändertes Logfile informiert zu werden. In diesem Fall wird ein Reset ausgeführt und alle Daten werden erneut aus der Storage gelesen. Bei den Storages registriert sie sich ebenfalls, um informiert zu werden, wenn ein Datenpaket evaluiert wurde, um die Visualisierung der Ratings zu aktualisieren.

In Bild 6.8 ist ein Dropdown-Menu zu sehen. Über dieses Menü kann man die angezeigten Datenpakete filtern. Ist der Button *Reverse* nicht gedrückt, werden alle Datenpakete angezeigt, die ein Rating haben, das kleiner oder gleich dem gewählten Wert im Dropdown-Menu ist. Bei gedrücktem Button alle, die größer als der gewählte Wert sind. Hierdurch lassen sich aus allen Datenpaketen die *interessanten*[3] ausfiltern. Umgesetzt wird dies, indem das `SelectionTableModel` mit dem Proxy-Model `SelectionTableSortFilterProxyModel` gefiltert wird. Dieses filtert alle Zeilen aus, in denen ein Evaluatorrating nicht den beschriebenen Bedingungen entspricht.

Evaluieren und Versenden von Datenpaketen

Neben dem Anzeigen der verschiedenen Datenpakete soll es natürlich auch möglich sein, diese einzeln zu evaluieren und wieder zu versenden. Passieren soll dies, sobald auf die Zeile eines Datenpakets im genutzten `QTableView` geklickt wird. Hierfür registriert sich die GUI beim View und wird jedes Mal benachrichtigt, sobald ein neues Datenpaket ausgewählt wird. Die Aufgabe der GUI ist es jetzt die Storage aufzufordern das Datenpaket zu versenden und die Evaluatoren dazu aufzufordern, das Datenpaket zu evaluieren.

Die Evaluation eines Datenpakets durch einen Evaluator kann eine unbestimmte Zeit dauern. Würde die GUI die Evaluatoren synchron, also blockend aus ihrem eigenen Thread aufrufen, wäre die GUI bis die Evaluation abgeschlossen ist blockiert und könnte keine Events mehr verarbeiten. Um dies zu vermeiden, wird für jedes zu evaluierende Datenpaket ein eigener Thread gestartet, der die GUI nicht blockiert. Implementiert wird dies mit der Klasse `DataUpdaterRunnable`, die auf einem übergebenen Evaluator `evaluateEntry`

[3] Welche Pakete interessant sind und welche nicht hängt vom Ratingalgorithmus der Evaluatoren ab.

ausführt. Dadurch, dass `DataUpdaterRunnable` von `QRunnable` erbt, kann die Klasse mit `QThreadPool::globalInstance()->start` in einem separaten Thread gestartet werden. Qt scheduled die zu startenden Prozesse automatisch und sorgt dafür, dass nicht mehr Threads als Prozessorkerne erzeugt werden. Für genauere Informationen siehe [42]. Aufgrund der Benutzung verschiedener parallel laufender Threads müssen alle Evaluatoren threadsicher sein.

Da das Evaluieren in verschiedenen Threads abläuft, weiß die GUI nicht, wann ein Evaluator mit Evaluieren fertig ist. Um diese Information zu erhalten, können sich Klassen wie bereits besprochen mit `connectToIndexEvaluated` bei den Evaluatoren registrieren.

Um alle Datenpakete auf einmal zu evaluieren, stellen die Evaluatoren `evaluateAll` zur Verfügung. Sollen alle Datenpakete mit allen Evaluatoren bearbeitet werden, wird diese Methode asynchron und gescheduled aufgerufen. Zusätzlich gibt es einen Dialog mit einem Fortschrittsbalken, um zu visualisieren, wie viele Evaluatoren bereits beendet sind.

Das Versenden von Datenpaketen über die `sendEntry`-Methode der Storages ist synchron und asynchron implementiert. Da das synchrone Versenden der Datenpakete bei den bisher implementierten Evaluatoren jedoch nie ein Problem darstellte, ist dies die Standardeinstellung, um keine unnötigen Threads erzeugen zu müssen.

6.1.2.3 Evaluator View

Neben der Visualisierung der Storage ist die grafische Darstellung der Evaluation einer der zentralen Punkte der GUI. Jeder Evalutor-Typ erfordert eine eigene Darstellung der Auswertung. Gleichzeitig soll die GUI jedoch möglichst generisch und unabhängig von konkreten Evaluatoren sein. Um dies zu erreichen, erzeugt die Evaluator GUI für jeden instanziierten Evaluator einen Tab ein einem `QTabWidget` und füllt dieses mit einem `QWidget`. Das `QWidget` wird nicht von der GUI-Klasse selbst erzeugt, sondern von einer extra Klasse, die dafür zuständig ist, zu einem gegebenen Evaluator die Klassen für seine grafische Darstellung zu erzeugen.

Aus Sicht der GUI besteht die Darstellung des Evaluationsergebnisses neben dem View, also dem `QWidget`, auch aus einem Model, mit dem der View arbeitet. Über dieses Model kann die GUI dem View mitteilen, die Auswertung welches Datenpakets aktuell angezeigt werden soll. Hierfür muss das von `EvaluatorModel` zur Verfügung gestellte Interface implementiert werden. Auch wenn aus Sicht der GUI View und Model getrennt sind, muss dies in der Implementierung der Visualisierungen nicht der Fall sein. Gekapselt werden die Implementierungen von der Klasse `GuiEvaluator` (Abbildung 6.11).

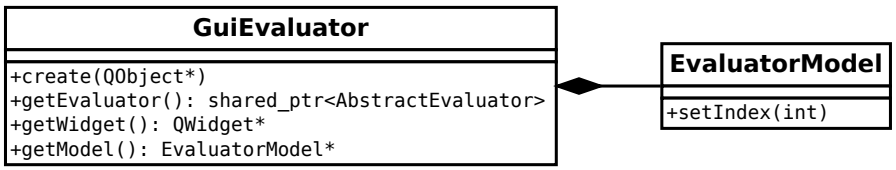


Abbildung 6.11: Kapselung von Evaluatoren und ihrer grafischen Darstellung

`GuiEvaluator` speichert eine Evaluatorinstanz für die GUI. Zusätzlich erzeugt es intern die passenden Models und Views und erlaubt der GUI den Zugriff auf diese. Beim Erstellen des Visualisierungswidgets unterscheidet `GuiEvaluator` zwischen zwei Evaluatortypen: Evaluatoren mit und Evaluatoren ohne Zusammenfassung. Für Evaluatoren ohne Zusammenfassung wird je nach Typ nur ein passendes Widget zurückgegeben. Bei Evaluatoren mit Zusammenfassung werden ein Widget für die aktuelle Auswertung und ein Widget für die Zusammenfassung erzeugt.

Wie in 6.1.1.5 besprochen kann ein konkreter Evaluator unterstützen zur Laufzeit konfiguriert zu werden, indem er das `AbstractRuntimeConfigurableEvaluator`-Interface implementiert. Um dies zu ermöglichen prüft die GUI bei einem Rechtsklick auf das Visualisierungs-Widget des Evaluators, ob es sich um einen rekonfigurierbaren Evaluator handelt und ob ein Dialog bei der Factory registriert wurde. Ist beides der Fall, wird ein Kontextmenü geöffnet, welches es erlaubt den Dialog zum Konfigurieren des Evaluators zu öffnen. Die beiden Widgets werden mit einem `QToolBox`-Widget gruppiert, welches zurückgegeben wird. Wird ein neuer Evaluatortyp eingeführt, muss `GuiEvaluator` entsprechend angepasst werden, um die passende Visualisierung zu erzeugen. Verschiedene Evaluatortypen und ihre Visualisierungen werden in 7.1 beschrieben.

Im Moment werden zwei verschiedene Evaluatortypen unterstützt: Evaluatoren die ihre Auswertung in tabellarischer Form und Evaluatoren die ihre Auswertung in Form eines Graphen darstellen lassen.

Visualisierung als Tabelle/Text

Evaluatoren, die ihre Auswertung in Form einer Tabelle (Abbildung 6.8) präsentieren, müssen von `AbstractTableEvaluator` erben und das gegebene Interface implementieren. Über dieses wird festgelegt, wie viele Zeilen und Spalten die Tabelle hat sowie deren Beschriftung. Die wichtigste Methode ist `data`. Über diese können die darzustellenden Daten für einzelne Zellen einer Tabelle abgefragt werden. Der Evaluator muss hierfür ein Objekt vom Typ `TableEvaluatorOutput` erzeugen und zurück liefern. `TableEvaluatorOutput`-Objekte enthalten ein mögliches Statusicon sowie den anzuzeigenden Text. Die Icons sind fest vorgegeben, die Evaluatoren können nur aus einer Liste bestehender Icons wählen. Diese Liste kann jedoch beliebig erweitert werden, falls bisher nicht vorhandene Icons benötigt werden. Der anzuzeigende Text kann mit HTML-Tags erweitert werden[4], um die Darstellung zu steuern. Überschriften, verschiedene Schriften oder Tabellen sind möglich. Hierdurch ist eine sehr komplexe Darstellung möglich. Wenn die Visualisierung mit der gegebenen Tabellenstruktur nicht gewünscht ist, kann es Sinn machen, nur eine Zeile und Spalte anzeigen zu lassen und die gewünschte Darstellung durch passenden HTML-Code zu erreichen.

Die Darstellung der Tabelle wird von einem `QTableView` in Kombination mit einem eigenen Delegate, `TableEvaluatorOutputDelegate`, übernommen (Abbildung 6.9). Der Delegate wird benötigt, um die Icons und den Rich Text in die einzelnen Zellen der Tabellen zu zeichnen. Die von `QAbstractTableModel` erbende Klasse `TableEvaluatorTableModel`

[4] Es werden hierbei nur die wichtigsten HTML-Elemente unterstützt. Für eine genaue Liste siehe [43].

setzt die von `AbstractTableEvaluator` gelieferten Informationen so um, dass sie von `QTableView` verarbeitet werden können (Abbildung 6.12).

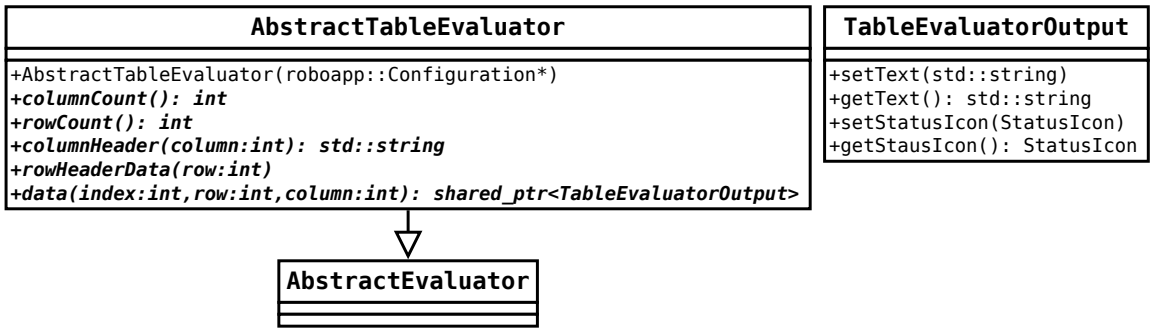


Abbildung 6.12: Evaluator mit Visualisierung in tabellarischer Form

Visualisierung als Graph

Neben der Darstellung in tabellarischer oder textueller Form bietet sich eine Darstellung in Form von Graphen an (Abbildung 6.13). Zum Zeichnen der Graphen wird die auf Qt aufsetzende Bibliothek *Qwt* genutzt [44]. In den von `AbstractGraphEvaluator` erbenden Evaluatoren selbst wird jedoch nicht direkt mit *Qwt* kommuniziert[5], sondern es werden Objekte vom Typ `GraphEvaluatorOutput` erzeugt, die in der GUI entsprechend in *Qwt*-Aufrufe übersetzt werden.

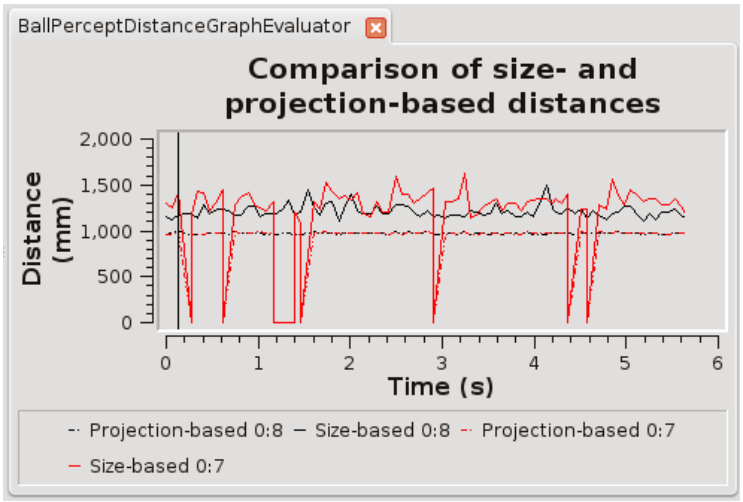


Abbildung 6.13: Evaluator der seine Ergebnisse, in diesem Fall ermittelte Ballabstände, als Graph darstellt

`GraphEvaluatorOutput` (Abbildung 6.12) können einzelne Kurven sowie spezielle Marker, entweder horizontale oder vertikale Linien beziehungsweise einzelne Punkte, hinzugefügt werden. Der Graph sowie die einzelnen Achsen können beschriftet werden. Kurven und

[5] Wie in 4.1 erwähnt, sind Evaluatoren und GUI unabhängig voneinander.

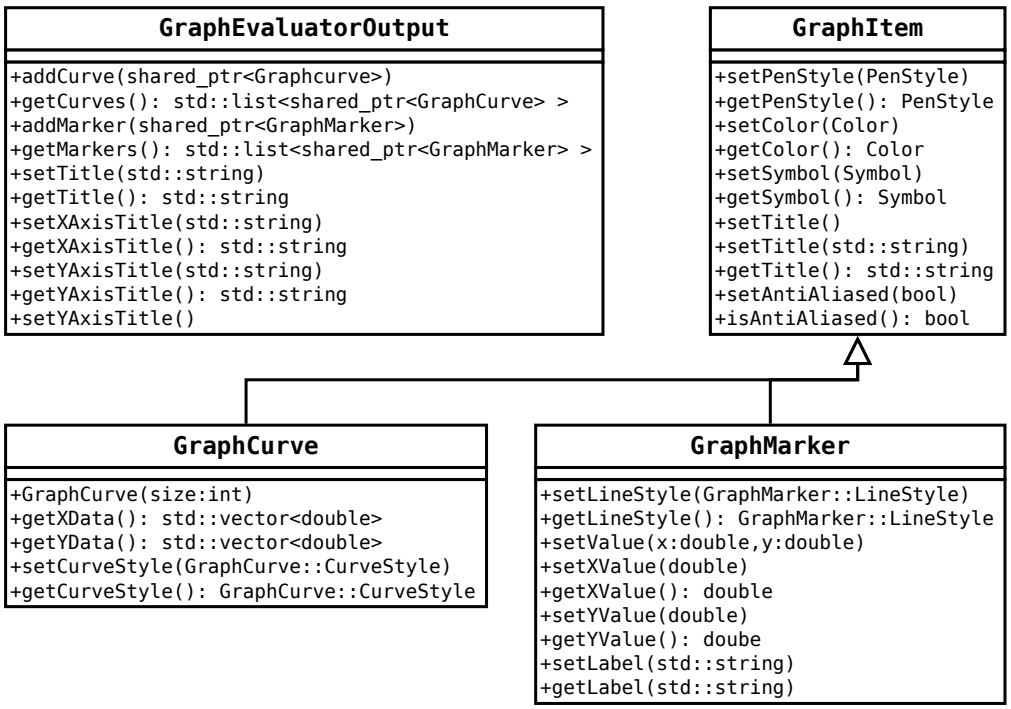


Abbildung 6.14: Evaluator mit Visualisierung als Graph

Marker erben beide von `GraphItem`. Über dieses kann festgelegt werden, wie die Elemente des Graphen gezeichnet werden sollen. Linienstil, Farbe, Titel, Kantenglättung und mehr können gesetzt werden.

`GraphCurve` speichert eine festgelegte Anzahl von Punkten, die zur Darstellung durch Linien verbunden werden. Der `GraphMarker` dagegen wird nur durch einen Punkt charakterisiert. Je nach gewünschtem Stil wird dieser unterschiedlich benutzt: Soll eine vertikale Linie gezeichnet werden, wird nur die x-Koordinate betrachtet, bei einer horizontalen Linie nur die y-Koordinate. Soll ein Punkt gezeichnet werden, werden x- und y-Koordinate benötigt.

Die eigentliche Visualisierung wird von der Klasse `GraphEvaluatorView` übernommen.

6.1.2.4 Toolbar

Neben dem Auswählen von Datenpaketen und dem Anzeigen der Evaluationsergebnisse muss es auch möglich sein, Dinge zu tun wie: Logdateien laden, speichern oder automatisch abspielen beziehungsweise die verwendete Storage und die verwendeten Evaluatoren auszuwählen. Solche Optionen sind über eine Toolbar am Kopf des Dialoges zu erreichen (Abbildung 6.15). Wie in 6.1.1.4 beschrieben, kann eine Storage zusätzlich zu den immer vorhandenen Optionen der Toolbar beliebige weitere Buttons hinzufügen. Im Weiteren werden die standardmäßig vorhandenen Optionen erklärt.

Abbildung 6.15: Standardtoolbar mit vier zusätzlichen Knöpfen einer Storage

Laden und speichern von Logdateien

Drei Buttons erlauben es, eine neue Logdatei zu laden, die aktuell geöffnete Logdatei auf der Festplatte zu speichern und die aktuell geöffnete Datei unter neuem Namen auf der Festplatte zu speichern. Genutzt wird hierfür das in 6.1.1.1 beschriebene Interface der aktuellen Storage. Wird eine neue Logdatei geöffnet oder die Datei unter neuem Namen gespeichert, arbeitet die Storage danach auf der neu angelegten Datei.

Angeforderte Keys bearbeiten

Alle von *RoboFrame* verschickten Daten werden durch einen eindeutigen *Key* identifiziert. Wie bereits beschrieben, kann eine Storage über die GUI Interesse an bestimmten Keys anmelden. Mit dieser Option wird ein Dialog geöffnet, der die angeforderten Keys anzeigt. Dieser Dialog erlaubt des Weiteren, die ausgewählten Keys zu entfernen oder neue hinzuzufügen. Im Normalfall wird diese Option nur bei der Entwicklung einer neuen Storage benötigt.

Abspielen eines Logfiles und Aufnehmen neuer Datenpakete

Oft nimmt man eine Logdatei auf und will diese im Nachhinein mit weiteren Informationen ergänzen. Ein Beispiel ist, von einem Roboter Bilder und Transformationsmatrizen aufnehmen zu lassen. Mit diesen beiden Datenpaketen kann eine Bildverarbeitung offline Perzepte[6] erkennen. Statt nur eine Bildverarbeitung über das Logfile laufen zu lassen, könnte man verschiedene Bildverarbeitungen die Bilder evaluieren lassen und alle Perzepte zusätzlich im Logfile speichern. Dieses vollständige Logfile könnte dann von einem Evaluator ausgewertet werden.

In Kombination mit dem Anfordern von Keys durch die Storage unterstützt die GUI genau dies. Die GUI arbeitet schematisch folgendermaßen: Die Storage wird aufgefordert, das aktuell in der GUI ausgewählte Datenpaket zu verschicken. Gestartete Prozesse arbeiten mit den Daten und verschicken ihre Ergebnisse wieder über *RoboFrame*. Die GUI empfängt die von ihr angeforderten Datenpakete und leitet sie an die Storage weiter. Sobald alle Prozesse das Datenpaket bearbeitet haben, wählt die GUI das nächste Datenpaket zum Verschicken aus, bis das letzte Datenpaket der Storage verschickt wurde.

Im obigen Ansatz gibt es zwei Probleme zu lösen: Die GUI muss wissen, wie viele Prozesse gerade laufen und die Daten bearbeiten und die GUI muss wissen, wann ein Prozess fertig mit dem Bearbeiten der Daten ist. Die Anzahl der Prozesse muss der Nutzer der GUI über ein Eingabefeld in der Toolbar mitteilen. Um zu signalisieren, dass ein Datenpaket fertig bearbeitet wurde, muss jeder Prozess ein Datenpaket mit einem vorher festgelegten Key verschicken. Hat die GUI so viele Datenpakete mit diesem Key empfangen, wie Prozesse gestartet wurden, wählt sie das nächste Paket aus.

[6] Bälle, Linien, Tore…

Nicht immer ist es wünschenswert, die angeforderten Datenpakete wirklich zu speichern. Manchmal ist es ausreichend, eine Logdatei abzuspielen und eingehende Datenpakete zu verwerfen. Die GUI stellt daher zwei Buttons zur Verfügung: einen *Play*-Button um das Abspielen des Logfiles zu starten und einen *Record*-Button um festzulegen, ob Datenpakete gespeichert werden oder nicht.

Änderung von Storage und Evaluatoren

Die verwendete Storage und die zugehörigen Evaluatoren sind nicht hardkodiert, sondern können zur Laufzeit geändert werden. Die GUI erzeugt hierfür einen Dialog, der mit allen Storages und Evaluatoren aus der oben beschriebenen Factory gefüllt wird. Zu einer Storage können immer nur die passenden Evaluatoren ausgewählt werden. Wurde ein zu einem Evaluator passender Konfigurationsdialog registriert wird dieser ausgeführt, sobald der Evaluator ausgewählt wird.

Evaluieren aller Datenpakete

Wenn alle Datenpakete der Storage auf einmal evaluiert werden sollen, ist es mühsam, jedes Paket einzeln anzuwählen. Durch einen einfachen Klick auf einen Button werden alle instanziierten Evaluatoren dazu aufgefordert, alle Datenpakete zu evaluieren.

6.1.3 Nötige Änderungen an *RoboFrame*

Um alle Features wie gewünscht umsetzen zu können, war es nötig *RoboFrame* teilweise zu erweitern oder sein Verhalten zu ändern. Die wichtigsten Änderungen werden im Folgenden erläutert.

6.1.3.1 Semantische Zeitstempel

Bei der Auswertung von Daten ist es nötig, zusammengehörige Daten zu gruppieren und einander zuordnen zu können. Aus einer gegebenen Menge von Bildern sowie einer Menge von Perzeptoren muss es beispielsweise möglich sein, die Perzeptoren den passenden Bildern zuzuordnen. In *RoboFrame* werden alle Daten, wie Bilder oder Perzepte, in sogenannte Streamables verpackt. Damit man Streamables einander zuordnen kann, müssen alle Streamables die zu einer Gruppe gehören mit einer eindeutigen Kennung versehen werden.

Bisher wurde in jedem Streamable zum Zeitpunkt seiner Erzeugung ein Zeitstempel mit der aktuellen Uhrzeit gespeichert. Hierdurch ist es zumindest möglich, Datenpakete nach Alter zu sortieren. Für zwei aufeinanderfolgende Bilder gehören alle Perzepte zum ersten Bild, deren Zeitstempel größer als der des ersten Bildes und kleiner als der des zweiten Bildes ist. Als Erstes wird ein Bild, danach die dazugehörigen Perzepte und danach das nächste Bild erzeugt. Spätestens wenn nicht jedes Bild, jedoch jedes Perzept, verfügbar ist, funktioniert dieses System nicht mehr.

Es gibt jedoch noch ein weiteres Problem: *RoboFrame* erlaubt es Streamables aufzunehmen, in Dateien zu speichern und zu einem späteren Zeitpunkt wieder abzuspielen, um weiter

mit den Streamables zu arbeiten. Beim Abspielen der Streamables wird der gespeicherte Zeitstempel jedoch jeweils wieder mit der aktuellen Uhrzeit ersetzt. Werden die Streamables also nicht in der Reihenfolge abgespielt, in der sie aufgenommen wurden, sind die Zeitstempel bei der weiteren Verarbeitung der Daten unbrauchbar.

Um das Problem zu lösen, wird neben dem Zeitstempel, an dem ein Streamable erzeugt wird, noch ein weiterer Zeitstempel eingeführt – der sogenannte *semantische Zeitstempel*. Wird dieser von der Anwendung nicht gesetzt, entsprechen sich der bisherige Zeitstempel und der semantische Zeitstempel. Der semantische Zeitstempel kann jedoch von Anwendungen genutzt werden, um Datenpakete zu gruppieren. Die Bildverarbeitung auf dem Roboter wurde zum Beispiel so angepasst, dass der semantische Zeitstempel aller Perzepte immer dem Zeitstempel des zugehörigen Bildes entspricht. Eine weitere Besonderheit des semantischen Zeitstempels ist, dass er nie vom Framework selbst geändert wird. Auch bei einem erneuten Abspielen der Daten bleiben die semantischen Zeitstempel also erhalten.

Evaluatoren sollten daher nie auf den Zeitstempeln, sondern immer auf den semantischen Zeitstempeln arbeiten.

6.1.3.2 Starten von Threads aus der GUI

In *RoboGui* war es bisher nicht vorgesehen, zur Laufzeit Threads mit zugehörigen Modulen zu starten und zu stoppen. Die GUI wurde nur zum Debuggen und zur Steuerung laufender Applikationen genutzt. Unter anderem um Algorithmen im Nachhinein auf aufgenommene Daten anwenden zu können, ist es interessant direkt aus der GUI Threads mit zugehörigen Modulen starten zu können und mit diesen Daten austauschen zu können.

Ein Prozess in *RoboFrame* ist ein eigenständig laufender Thread. Gleichzeitig erbt dieser aber von `Connector` und ist damit ein Konnektor, der beim Router registriert wird. Über Module können einem Thread dann weitere Konnektoren hinzugefügt werden. Um dies abzubilden, ist es über die GUI nicht nur möglich Threads zu starten und zu stoppen, sondern allgemein Konnektoren.

Das Anlegen der Konnektoren über die GUI läuft folgendermaßen ab: Der Nutzer wählt über das Menü einen Konnektor/eine zu startende Gruppe von Konnektoren. Wurde ein Konfigurationsdialog registriert, wird dieser angezeigt und füllt ein Konfigurationsobjekt. Das Konfigurationsobjekt wird an eine Klasse übergeben, welche Prozesse und andere Konnektoren erzeugt und konfiguriert. Eine Liste der erzeugten Konnektoren wird an die GUI zurückgegeben, welche die Konnektoren beim Router registriert. Für jeden erzeugten Konnektor wird in der GUI ein Menüeintrag registriert, um diesen zu stoppen und aus dem Router zu entfernen.

Wie bei der Registrierung von Evaluatoren und deren Konfigurationsdialogen ist für die Konnektoren eine Registrierung bei Factorys nötig. `ConnectorFactory` registriert Objekte vom Typ `IWrappedConnectors`, die dafür zuständig sind, die Konnektoren zu erzeugen. `ConnectorConfigurationDialogFactory` erledigt Registrierung und Erzeugung der Konfigurationsdialoge. Die von `QMenu` erbende Klasse `ConnectorsMenu` wird in das Menü von jeder auf *RoboFrame* basierenden GUI eingebunden und stellt für alle bei den

Factorys registrierten Objekte die passenden Menüeinträge zur Verfügung. Sie kapselt das Konfigurieren, Erzeugen, Registrieren und Entfernen der Konnektoren.

Der wichtigste Einsatzzweck dieser Erweiterung für diese Arbeit ist die Möglichkeit verschiedene Bildverarbeitungsprozesse aus der GUI starten zu können. Vom Team der *Darmstadt Dribblers* wurden jedoch basierend auf der Erweiterung Möglichkeiten geschaffen Prozesse aus der GUI zu starten, um über die serielle Kommunikation mit dem Roboter kommunizieren und diesen beispielsweise Kalibrieren zu können. Bisher musste hierfür neben der GUI immer noch eine Applikation gestartet werden. Eine weitere Möglichkeit ist es, die Weltmodellierung und Teamkommunikation in der GUI zu starten und so den Status verschiedener laufender Roboter zu visualisieren.

6.2 Bildverarbeitung

Um nicht alle gängigen Bildverarbeitungsalgorithmen neu implementieren zu müssen, setzt der Code auf die bestehende Bibliothek *OpenCV* [45,46] auf. Die freie, quelloffene, unter einer BSD-Lizenz stehende Bibliothek wurde speziell für den Einsatz in der Echtzeitbildverarbeitung entwickelt. Sie stellt mehr als 500 Funktionen aus allen Bereichen der Bildverarbeitung zur Verfügung.

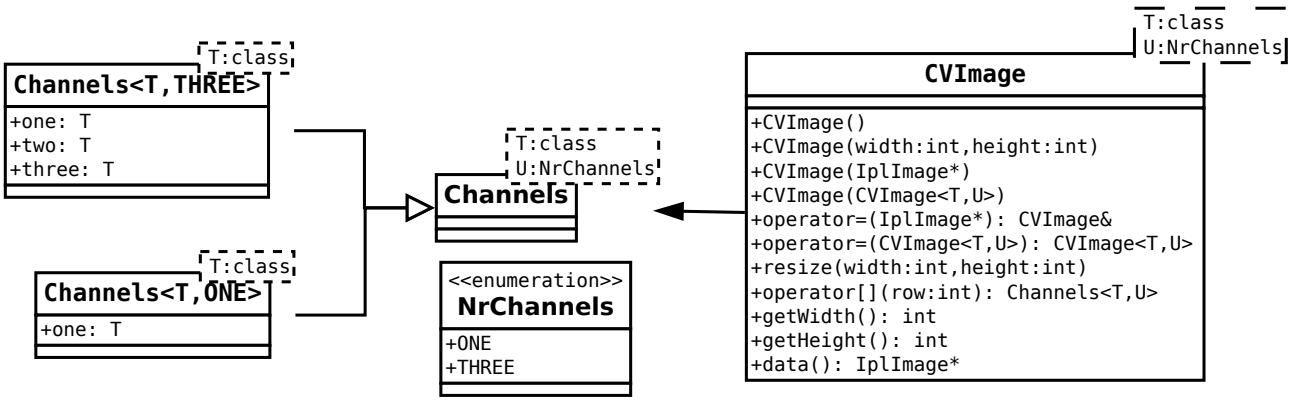


Abbildung 6.16: C++-Wrapper um `IplImages`

OpenCV 1 ist komplett in C geschrieben und damit zwar C++ kompatibel, allerdings gibt es keine nutzbaren C++-Bindungen. Die wichtigste von *OpenCV* zur Verfügung gestellte Struktur ist ein Bild, genannt `IplImage`. Das `IplImage` stellt Bilder mit verschiedenen Farbtiefen und einer Anzahl von Kanälen dar. Um diese Bilder komfortabel in C++ nutzen zu können, wurde eine Templateklasse Namens `CVImage` geschrieben (Abbildung 6.16). Diese kapselt die C-Funktionen zum Erstellen, Kopieren, Vergrößern und Verkleinern der `IplImages`. Mit Templateparametern werden der Datentyp, der ein einzelnes Pixel speichert, und damit die Farbtiefe sowie die Anzahl der Farbkanäle festlegt. In dieser Arbeit werden Bilder mit 8 Bit pro Kanal genutzt, die als `unsigned chars` gespeichert werden. Alternativ könnten je nach Einsatzgebiet auch `chars`, (`unsigned`) `shorts`/`ints`, `floats` oder `doubles`

genutzt werden. Die Farbkanäle pro Bild sind auf einen oder drei festgelegt. Kodiert wird dies durch einen Wert des Enums `NrChannels`. Um auf ein spezielles Pixel zugreifen zu können, wird der vom Bild allokierte Speicher auf ein Objekt vom Typ `Channels` gemappt. Dieses erlaubt es, je nach Anzahl an Farbkanälen auf ein oder auf drei Kanäle des Pixels zuzugreifen. Da die von *OpenCV* zur Verfügung gestellten Methoden nur direkt auf `IplImage`s arbeiten können, kann auf diese über `data` zugegriffen werden.

6.2.1 Vorverarbeitung eingehender Bilder

Alle Perzeptoren können auf unterschiedlichen Daten wie verschiedenen Farbräumen, Farbkanälen, Kanten- oder Bloberkennern arbeiten. Diese Daten werden aus dem aktuellen vom Roboter aufgenommen Bild generiert. Je nachdem welche Evaluatoren in einem Bearbeitungszyklus gestartet werden, müssen entsprechende Daten bereitgestellt werden. Damit nicht in jedem Zyklus alle möglichen Daten erzeugt werden müssen, können in einem Zyklus laufenden Perzeptoren die für sie interessanten Daten anfordern. Zuständig für Registrierung, Erzeugung und Verbarbeitung ist die Klasse `NativeCVFactory` (Abbildung 6.17).

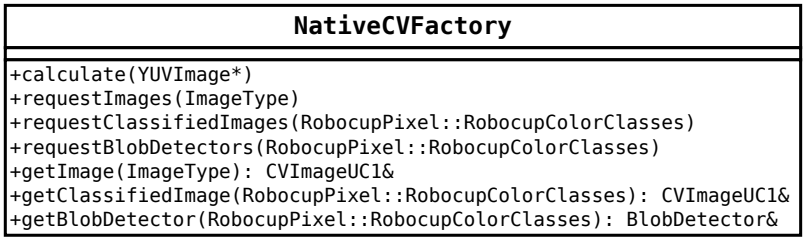


Abbildung 6.17: Factory zur Vorverarbeitung der Kamerabilder

Perzeptoren können bei ihr mit `requestImages` einzelne Kanäle des Bildes im RGB-, YUV- oder HSV-Farbraum sowie die Kanten im Y-, U- oder V-Kanal beziehungsweise einer Kombination dieser Kanten anfordern. Mit `requestClassifiedImages` können farbklassifizierte Bilder aller im RoboCup verwendeten Farben, mit `requestBlobDetectors` analog die zu einer Farbe passenden Blobs, angefordert werden.

Wurden alle Daten angefordert, bekommt die GUI über `calculate` ein Bild im YUV-Farbraum übergeben und berechnet aus diesem alle benötigten Daten.

Nach der Berechnung können analog zu den `request`-Methoden über `get`-Methoden die berechneten Daten abgefragt werden.

Farbklassifikation

Wie in 5.2 beschrieben basiert die Farbklassifikation auf festgelegten Grenzwerten im HSV-Farbraum. Aus einem Eingangsbild und den Grenzwerten wird ein binarisiertes Bild berechnet. In diesem werden Pixel die einer Farbe entsprechen auf 255 und Pixel die einer Farbe nicht entsprechen auf null gesetzt (siehe Algorithmus 6.1). Das entstehende Bild wird noch mit einem Medianfilter geglättet, um einzelne Fehlklassifizierungen zu entfernen.

Eingabe : Bild im HSV-Farbraum, obere und untere Grenzwerte für jeden Kanal und jede Farbe
Ausgabe : Binarisiertes Bild für jede angeforderte Farbe

```
foreach angeforderte Farbe do
    foreach Pixel do
        foreach Farbkanal do
            if Wert liegt in Grenzwerten then
                speichere 255;
            else
                speichere 0;
            end
        end
        setze Pixelwert im klassifizierten Bild auf Minimum der drei Kanäle;
    end
    entferne Ausreißer mit Medianfilter;
end
```

Algorithmus 6.1 : Farbklassifikation mit Grenzwerten im HSV-Farbraum

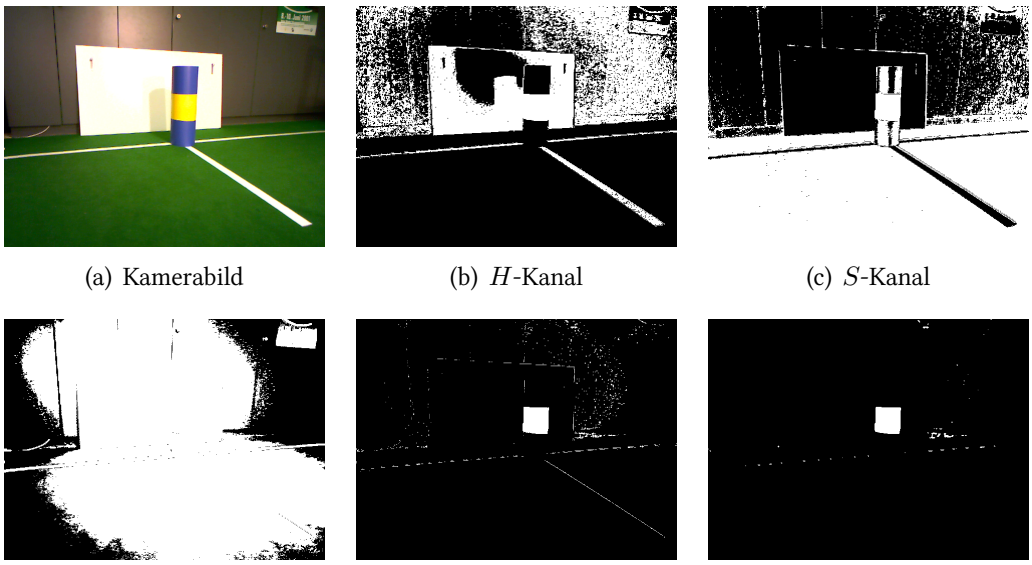
Der wichtigste Teil der Farbklassifikation ist, die korrekten Grenzwerte zu finden. Anhand von Bildern mit unterschiedlichen Farbtönen und unterschiedlichen Beleuchtungen müssen Grenzwerte gefunden werden, die einen möglichst großen Bereich abdecken, um vielseitig nutzbar zu sein. Gleichzeitig sollte aber ein möglichst kleiner Bereich abgedeckt werden, um Fehlerkennungen zu vermeiden. Die theoretischen Grundlagen der Grenzwerte wurden in 3.3.1.3 und 5.2 beschrieben. Im Moment gibt es keine Methode die Grenzwerte automatisch zu erstellen, sie müssen von Hand für jede Farbe und jeden der drei Kanäle ermittelt und optimiert werden. Um das Debuggen der aktuellen Einstellung zu vereinfachen, können die binarisierten Bilder mit der in 6.2.3.1 beschriebenen Technik visualisiert werden (Abbildung 6.18).

Wie in 3.3.1.3 beschrieben, besteht die H-Komponente eines Pixels im HSV-Farbraum aus einem Winkel zwischen 0 und 360 Grad. Dieser Wert eignet sich jedoch nicht zur Speicherung in einem `unsigned char`. *OpenCV* mappt diesen Winkel daher auf einen Wert zwischen 0 und 180. Die S- und die V-Komponente nutzen den kompletten Wertebereich aus und liegen zwischen 0 und 255. Die genutzten Werte je nach Farbe finden sich in Tabelle 6.1.

Kantenerkennung

Die in 3.3.7 beschriebene klassische Kantenerkennung kann nur auf Bildern mit einem Farbkanal arbeiten. Je nach Farbton und Beleuchtung können Kanten jedoch in unterschiedlichen Farbkanälen unterschiedlich gut erkannt werden. Daher empfiehlt es sich, Kanten in mehreren Farbkanälen zu erkennen und das Ergebnis in einem Bild zu vereinigen (Abbildungen 6.19(a)–6.19(d)).

Der HSV-Farbraum kommt für die Kantenerkennung nicht infrage, da die H-Komponente durch ihre zyklische Form bei Farbwerten um 0 Grad fast 360 Grad springen kann, obwohl sich an dieser Stelle keine Kante befindet. Der YUV-Raum bietet sich an, da diese Werte



(a) Kamerabild (b) H-Kanal (c) S-Kanal

(d) V-Kanal (e) Minimum von H-, S- und V-Kanal (f) Minimum nach Rauschunterdrückung

Abbildung 6.18: Original Kamerabild, Klassifizierung für Gelb im H-, S- und V-Kanal sowie sich das daraus ergebende klassifizierte Bild vor und nach dem Entrauschen

direkt von der Kamera geliefert werden und ohne Umwandlung nutzbar sind.

Genau wie bei der Farbklassifizierung gibt es auch bei der Kantenerkennung das Problem passende Parameter zu finden. Der hier verwendete Canny-Algorithmus erwartet zwei Parameter (siehe 3.3.7.1). Bei drei Farbkanälen ergibt dies sechs zu bestimmende Parameter. Für jeden Farbkanal müssen die Parameter so gewählt werden, dass möglichst viele Kanten erkannt werden. Gleichzeitig sollen aber möglichst wenige Kanten erkannt werden, die im Bild keiner Kante, aber zum Beispiel einem Farbübergang, entsprechen. Eine automatische Bestimmung dieser Parameter ist nicht möglich, sie müssen von Hand empirisch optimiert werden. Für die in dieser Arbeit genutzten Parameter siehe Tabelle 6.2.

Auch nach der Kombination der im Y-, U- und V-Kanal erkannten Kanten, wird es immer Kanten geben, die nicht erkannt wurden. Dies passiert zum Beispiel bei zu geringem Kontrast (blaue Pole vor dunkler Wand) oder bei sehr dünnen Kanten wie weit entfernten Linien. Mit niedrigeren Grenzwerten können diese Kanten meistens erkannt werden, allerdings ist dies keine Option, da die Fehlerkennungen zu stark zunehmen. Im nächsten Abschnitt werden Methoden beschrieben, um trotz Fehlerkennungen und nicht erkannter Kanten korrekte Ergebnisse zu erzielen.

Wie in Abbildung 6.19(d) werden oft auch Kanten mitten in Objekten erkannt, in diesem Fall beispielsweise durch Glanzlichter. Da innerhalb einer zusammenhängenden klassifizierten Fläche (Abbildung 6.19(d)) jedoch per Definition keine Kanten auftreten können, ist es möglich diese Fehlerkennungen zu korrigieren (Abbildung 6.19(e)).

	H [0..359]		S [0..255]		V [0..255]	
Farbe	unterer	oberer	unterer	oberer	unterer	oberer
blau	180	270	80	255	40	255
gelb	35	70	120	255	75	255
orange	25	60	100	255	75	255
grün	80	160	80	255	20	255
weiß	0	360	0	130	100	255

Tabelle 6.1: Grenzwerte der Farbklassifizierung

	Kanal		
Schwellwert	Y	U	V
oberer	900	370	210
unterer	725	270	90

Tabelle 6.2: Genutzte Grenzwerte des Canny-Algorithus' für Y-, U-, und V-Kanal

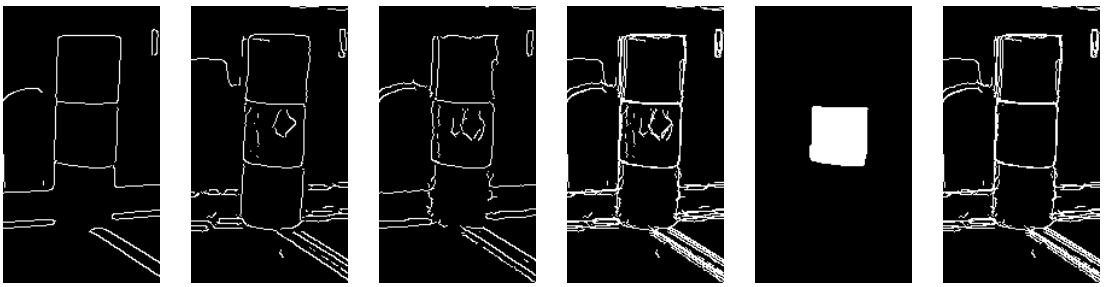
Bloberkennung

Mit den gewonnenen Informationen aus Farklassifizierung und Kantenerkennung können jetzt die farbigen Blobs aus dem Bild extrahiert werden. Wie im Algorithmus 5.1 beschrieben werden im ersten Schritt zusammenhängende Flächen in den klassifizierten Bildern gesucht und deren Schwerpunkte ermittelt. Ausgehend von diesen Schwerpunkten werden mithilfe der erkannten Kanten die finalen Blobs ermittelt.

Die Ermittlung der Schwerpunkte (Abbildung 6.20(a) und 6.20(b)) kann komplett mit *OpenCV*-Funktionen erledigt werden.[7] Blobs werden in *OpenCV* Konturen genannt. Die Konturenerkennung arbeitet nicht direkt auf einem Bild, sondern auf einem binarisierten Bild, das die Umrisse der Objekte enthält. Für das farbklassifizierte Bild erhalten wir dieses Bild durch Anwenden des Canny-Algorithmus'. Das Erkennen von Konturen wird mit *cvStartFindContours* initialisiert. Durch Setzen der Parameter `mode` auf `CV_RETR_EXTERNAL` und `method` auf `CV_CHAIN_APPROX_NONE` ermittelt *OpenCV* nur die äußere Kante einer Kontur und liefert diese in Form einer Liste von Punkten zurück. Die erkannten Konturen können nach und nach mit `cvFindNextContour` erzeugt werden. Sollen sie zu Debuggingzwecken visualisiert werden, steht `cvDrawContours` zur Verfügung. Konturen mit einer sehr kleinen Fläche werden direkt verworfen.

`cvMoments` berechnet die Momente bis zur dritten Ordnung einer Kontur. Mit den Momenten M_{00}, M_{01} und M_{10}, kann der Schwerpunkt in x- und y-Richtung berechnet werden. Siehe dazu die Gleichungen 6.1. $I(u, v)$ ist, da das Bild binarisiert ist, für alle Pixel innerhalb einer Kontur eins und kann daher entfallen.

[7] Für die Dokumentation der verwendeten Funktionen siehe [47].



(a) Y-Kanal (b) U-Kanal (c) V-Kanal (d) Y-, U-, V-Kanal kombiniert (e) gelbe Farbklassifikation (f) Korrigierte Kanten

Abbildung 6.19: Ausschnitt aus Abbildung 6.18(a). (a)–(c) sind das Ergebnis des Canny-Algorithmus' für den Y-, U- und V-Kanal des Kamerabildes. (d) ist die Kombination der Kanten aller drei Kanäle. Um fehlerhaft erkannte Kanten zu eliminieren, wird mit der erodierten gelben Farbklassifikation (e) die Zusammenfassung der Kanäle zu (f) korrigiert.

$$M_{pq} = \sum_{u,v \in K} I(u,v) u^p v^p \qquad \bar{x} = \frac{M_{10}}{M_{00}} \qquad \bar{y} = \frac{M_{10}}{M_{00}} \tag{6.1}$$

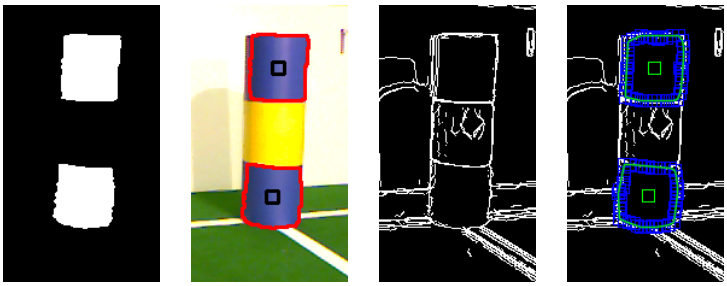
M_{00} entspricht der Anzahl der Pixel, M_{10} der Summe der x- und M_{01} der Summe der y-Werte der Koordinaten innerhalb der Kontur.

Der nächste Schritt nach der Bestimmung der Schwerpunkte im klassifizierten Bild ist das Bestimmen von Blobs mithilfe der erkannten Kanten (Abbildung 6.20(c) und 6.20(d)). 48 Strahlen werden hierfür von den Schwerpunkten ausgehend im Abstand von $7.5°$ erzeugt und verfolgt, bis sie entweder auf eine Kante oder den Bildrand treffen. Um die kontinuierlichen Strahlen auf Pixel abzubilden, wird der Bresenham-Algorithmus (siehe 3.3.6) genutzt. Da eine Kante oft nicht durchgehend ist, sondern kleine Löcher haben kann, werden bei der Suche nach einer Kante nicht nur der Pixel auf dem Strahl, sondern auch die Pixel darüber und daneben betrachtet. Neben den Kanten im Kamerabild wird gleichzeitig nach Kanten im abgeleiteten klassifizierten Bild gesucht. Diese können verwendet werden, um Fehlerkennungen und nicht erkannte Kanten zu korrigieren. Wurde im klassifizierten Bild und im Kamerabild keine Kante gefunden und der Bildrand erreicht, ist es sehr wahrscheinlich, dass ein Objekt nur teilweise sichtbar ist. Der Bildrand kann dem Blob hinzugefügt werden. Ist der Bildrand erreicht, wobei keine Kante im Kamerabild aber eine Kante im klassifizierten Bild gefunden wurde, ist es wahrscheinlich, dass die richtige Kante im Bild nicht erkannt wurde. Der Strahl wird verworfen.

Mit den validen gefundenen Kanten werden Objekte vom Typ `Blob` erzeugt. Diese Klasse erlaubt es, zu gegebenen Punkten unter anderem die Fläche der konvexen Hülle, Schwerpunkt oder Standardabweichung und Varianz der einzelnen Punkte vom Schwerpunkt zu bestimmen. Da die Farbklassifizierung sehr großzügig gewählt ist, sollten die Blobs die über die Kanten im Kamerabild erkannt wurden nicht signifikant größer sein als die Blobs, die über die Farbklassifikation erkannt wurden. Ist dies doch der Fall, ist es sehr wahrscheinlich, dass

Kanten im Bild nicht erkannt wurden dafür aber weiter entfernte Kanten, die nicht zum gleichen Objekt gehören.

Tritt dieser Fall auf, wird versucht, mit einem einfachen Algorithmus die fehlerhafte Kante zu korrigieren: Für alle Randpunkte des Blobs wird überprüft, ob sie signifikant vom durchschnittlichen Abstand vom Schwerpunkt abweichen und der farbklassifizierte Randpunkt für diesen Winkel einen besseren Kandidaten darstellt. Ist dies der Fall, wird der Randpunkt entsprechend ersetzt. Nach der Prüfung aller Randpunkte wird erneut überprüft, ob die Blobs aus Kamerabild und klassifiziertem Bild zusammenpassen.



(a) blaue Farbklassifikation (b) ermittelte Schwerpunkte (c) erkannte Kanten (d) erkannte blaue Blobs

Abbildung 6.20: Ausschnitt aus Abbildung 6.18(a). Erkennen blauer Blobs: Basierend auf der Farbklassifikation für Blau (a) werden blaue Flächen und deren Schwerpunkte gesucht (b). Mithilfe der ermittelten Kanten (c) werden die fertigen Blobs (d) erzeugt.

Die Erkennung der Blobs und die erzeugten `Blob`-Objekte werden von der Klasse `BlobDetector` übernommen (Abbildung 6.21). Mit `findBlobs` werden Blobs aus einem gegebenen binarisierten farbklassifizierten Bild und einem Bild der Kanten des Kamerabildes extrahiert und in einen Vektor gespeichert. Über `getFoundBlobs` kann auf diesem Vektor gearbeitet werden. `BlobDetector` erlaubt es zudem, auf den erkannten Blobs zu arbeiten, diese zu sortieren oder uninteressante Blobs zu verwerfen. Da je nach Anwendungsfall unterschiedliche Operationen durchgeführt werden müssen, wird beim Kopieren eines `BlobDetector`-Objekts eine Kopie aller gespeicherten Blobs erstellt, sodass die Arbeit auf einem `BlobDetector` die anderen nicht beeinflusst.

Mit `joinBlobs` können direkt nebeneinanderliegende Blobs zusammengefasst werden. Da ein `BlobDetector` nur auf einer Farbe arbeitet und im RoboCup keine direkt aneinandergrenzenden Objekte der gleichen Farbe vorkommen, sind solche Blob-Kombinationen im Normalfall eine Fehlerkennung und gehören zum gleichen Objekt. Für einen Erkenner kann es wichtig sein, dass Blobs in einer speziellen Reihenfolge, beispielsweise im Bild von oben nach unten oder von links nach rechts sortiert, bearbeitet werden. `BlobDetector` stellt die Methode `sortBlobs` zur Verfügung. Diese erhält als Parameter einen sogenannten *Comparator*. Dies ist ein Objekt, welches zwei `Blob`-Objekte übergeben bekommt und `true` zurückgibt, wenn die beiden Objekte in der gewünschten Reihenfolge sortiert sind. Der *Comparator* muss eine strenge schwache Ordnung erzeugen, das heißt, die Blobs haben

eine eindeutige Reihenfolge, es sind jedoch mehrere Objekte mit gleichen Werten erlaubt. Beispiele für *Comparatoren* sind Vergleiche der x-Werte des Schwerpunkts, der Fläche der Blobs oder des durchschnittlichen Abstands der Randpunkte vom Schwerpunkt mit dem kleiner Operator.

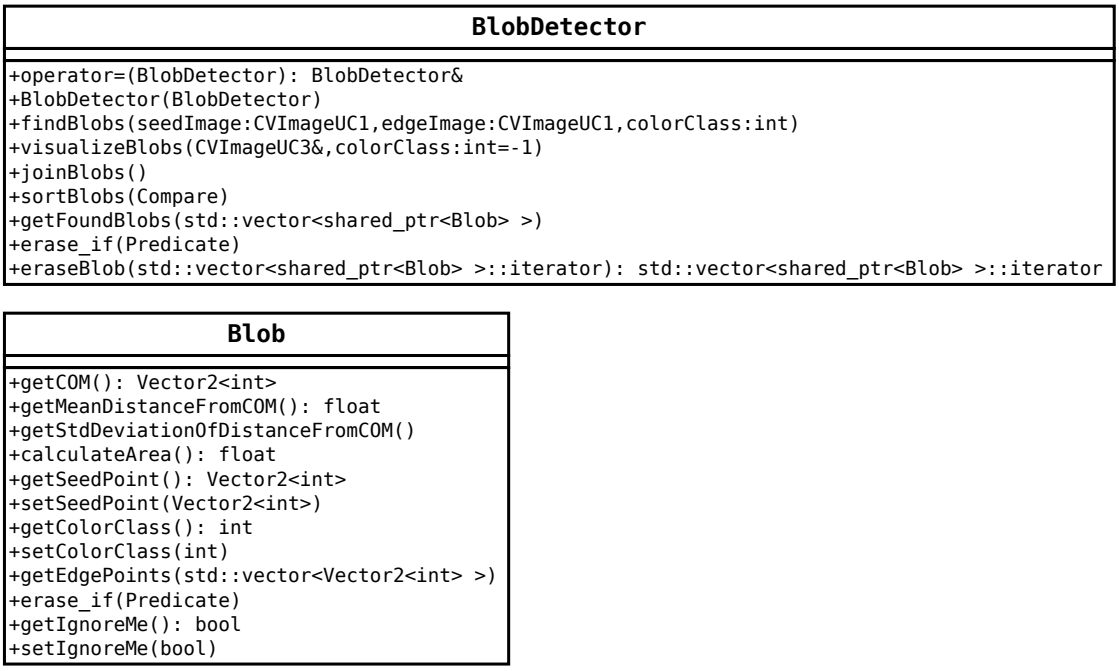


Abbildung 6.21: Interface für die Bloberkennung

Neben dem Sortieren der Blobs ist es interessant, Blobs anhand bestimmter Kriterien zu entfernen und danach nur die interessanten Blobs zu betrachten. Analog zu `sortBlobs` funktioniert die Methode `erase_if`. Hier wird jedoch statt einem *Comparator* eine *Predicate*-Klasse übergeben. Diese gibt für einen Blob *false* zurück, wenn der Blob entfernt werden soll. Möglich ist es zum Beispiel Blobs mit unpassender Größe oder zu großer Standardabweichung/Varianz der Randpunkte zu verwerfen. Auch die `Blob`-Klasse besitzt eine `erase_if`-Methode. Mit dieser können einzelne Randpunkte entfernt werden. Dies ermöglicht insbesondere einzelne Ausreißer, die sehr vom Durchschnitt abweichen, zu entfernen.

Zum Debuggen der erkannten Blobs können diese mit `visualizeBlobs` in ein RGB-Bild gezeichnet werden. Hierbei werden die Randpunkte der Blobs mit kleinen Quadraten visualisiert und durch Linien verbunden. Die verwendete Farbe ist von der Farbklasse der erkannten Blobs abhängig.

6.2.2 Erkenner

Zu jeder Bildverarbeitung gehört ein Satz von Erkennern, auch Perzeptoren genannt, die aus den vorverarbeiteten Bildern die eigentlichen Objekte extrahieren. Die Perzeptoren für ein Objekt unterscheiden sich dabei nicht vollständig, sondern haben eine gewisse

Schnittmenge. Die Perzeptoren der bestehenden Bildverarbeitung werden daher in zwei Teile zerlegt. Einen generischen, universellen Teil und den für eine konkrete Bildverarbeitung spezifischen Teil. Der universelle und wiederverwertbare Teil wird in Klassen Namens `Universal*Perceptor` extrahiert, wobei `*` für den jeweiligen Perzeptor steht. Die Perzeptoren der einzelnen Bildverarbeitungen erben von den `Universal*Perceptoren`.

Polesperzeptor

Der Polesperzeptor fordert von der `NativeCVFactory` Blobs für die Farbklassen Gelb und Blau sowie die klassifizierten Bilder für Weiß und Grün an.

Die erkannten gelben und blauen Blobs werden nach ihrer Position im Bild von oben nach unten sortiert. Wie in 5.2.3 beschrieben besteht eine Pole aus einer Kombination von Blobs der Farben blau-gelb-blau oder gelb-blau-gelb unter denen sich eine grüne oder weiße Fläche befindet. Der erste Schritt eine Kombination aus drei passenden Blobs zu finden ist es, zwei korrekt übereinander liegende Blobs zu finden. Hat man zwei Blobs gefunden, sucht man nach einem weiteren Pärchen und zwar nach einem Blob der zu dem unteren Blob des vorher gefundenen Pärchens passt. Durch weitere Tests kann nun festgestellt werden, ob die drei gefundenen Blobs zu einer Pole gehören oder nicht.

Matching Das Matching zweier Blobs (entweder gelb zu blau oder umgekehrt) funktioniert folgendermaßen: Zu einem Blob wird über die von oben nach unten sortierten Blobs der anderen Farbe iteriert, bis der erste Blob einen Schwerpunkt mit größerem y-Wert als der Schwerpunkts des gegebenen Blobs hat. Der Schwerpunkts des zweiten Blobs ist also größer. Der Schwerpunkt muss nicht nur unterhalb des Schwerpunkts des anderen Blobs liegen, gleichzeitig müssen die Blobs auch noch direkt übereinander liegen und sich berühren. Um dies zu garantieren, werden alle Blobs verworfen, bei denen die x-Werte der Schwerpunkte zu weit voneinander abweichen. Je größer die Blobs sind, desto größere Abstände werden erlaubt. Gleiches gilt für die y-Werte.

Sind zwei bis auf leichte Abweichungen übereinander liegende Blobs gefunden, muss überprüft werden, ob diese sich berühren und damit zum gleichen Objekt gehören können. Die Abstände der zehn unteren Randpunkte des oberen Blobs und zehn oberen Randpunkte des unteren Blobs werden hierbei verglichen. Mindestens ein Pärchen aus oberem und unterem Blob muss hierbei nur wenige Pixel entfernt sein. Bei gut segmentierten Objekten liegt mehr als ein Randpunkt direkt beieinander, unter schlechten Lichtbedingungen ist dies jedoch nicht garantiert.

Verifikation Wurden drei passend übereinander liegende Blobs gefunden wird überprüft, ob diese auf grünem oder weißem Grund stehen. Vom untersten Blob wird die Unterkante bestimmt. Von dieser aus werden, abhängig von der Orientierung der Pole[8], Pixel auf einer Linie, die ein Drittel der Höhe der Pole lang ist, überprüft. Sind mindestens ein Viertel der Pixel Grün oder Weiß werden die gefundenen Blobs als valide Pole anerkannt.

[8] Die Richtung ergibt sich aus der Differenz des Schwerpunkts des ersten und des dritten Blobs

Die bestehende Bildverarbeitungsinfrastruktur stellt bereits Methoden zur Verfügung, um aus Positionen in einem aufgenommenen Bild die relative Position des Punktes zum Roboter in der wirklichen Welt zu berechnen. Zusätzlich ist es möglich, aus Abständen im Bild, deren Abstand in der wirklichen Welt bekannt ist, korrektere Entfernungen zu errechnen. Da die Höhe einer Pole bekannt ist, kann anhand der erkannten Blobs genau die Entfernung der Pole bestimmt werden. Über die schon vorher bestimmte Unterkante der Pole kann schließlich deren genaue Position relativ zum Roboter errechnet werden.

6.2.3 Debugging

6.2.3.1 Visuelles Debugging

Beim Einsatz von Bildverarbeitungsalgorithmen ist es, entgegen vieler anderen Algorithmen, oft nicht praktikabel, mit einem Debugger nach Problemen zu suchen. Es bietet sich vielmehr an, die bearbeiteten Bilder direkt anzusehen und zu beurteilen. Um die Qualität einer Kantenerkennung zu beurteilen, ist es zum Beispiel am effektivsten das Originalbild und ein Bild der erkannten Kanten anzuzeigen und diese zu vergleichen. Es wird also eine Methode benötigt, die von *OpenCV* genutzten `IplImage`s zu visualisieren. *OpenCV* stellt mit *HighGui* (siehe [48]) eine Bibliothek mit entsprechender Funktionalität zur Verfügung. Da diese jedoch nicht auf Qt 4 als GUI-Bibliothek aufsetzt, gibt es bei einer Nutzung in Kombination mit *RoboGui* Probleme, wie sich nicht mehr aktualisierende Fenster und Abstürze.

Um diese Probleme zu umgehen, wurde die Klasse `VisionDebugHelper` geschaffen. Sie erlaubt es Dialoge mit einem bestimmten Namen zu erzeugen und in diesen Dialogen `IplImage`s anzuzeigen. Hierfür muss das `IplImage` in ein für Qt verständliches `QImage` umgewandelt werden. Herausforderung ist, dass Dialoge nur von dem Thread, in dem die GUI läuft, erzeugt und gezeichnet werden können. Die Bildverarbeitung läuft jedoch normalerweise in einem anderen Thread. Um dieses Problem zu lösen, wird vom aufrufenden Thread ein `QEvent` and den GUI-Thread geschickt. Dieser führt sobald er wieder läuft die gewünschte Aktion durch. Da verschiedene Threads gleichzeitig auf dem `VisionDebugHelper` arbeiten können, werden alle Methoden durch Mutexe geschützt.

6.2.3.2 Error Handler

Trifft *OpenCV* auf einen Fehler, zum Beispiel inkompatible Matrixgrößen, beendet dieses standardmäßig das laufende Programm ohne eine aussagekräftige Fehlermeldung. Selbst im Debugger ist es nicht möglich zu sehen, an welcher Stelle im Code der Fehler Auftritt. Um dieses Problem zu umgehen, wird mittels `cvRedirectError` ein sogenannter *Error Handler* installiert. Dies ist eine Funktion, die von *OpenCV* aufgerufen wird, sobald ein Fehler auftritt. Die installierte Funktion ruft den Standard Error Handler von *OpenCV* auf, allerdings bricht sie das Programm danach nicht ab, sonder triggert einen *Assert*. Startet man das Programm im Debugger, kann genau festgestellt werden, an welcher Stelle im Programmcode der Fehler aufgetreten ist.

Kapitel 7

Ergebnisse

Im letzten Kapitel wurde die Realisierung einer Infrastruktur zur Evaluierung verschiedener Algorithmen vorgestellt. Mit diesem Framework selbst können noch keine Ergebnisse erzielt werden. Um die Nutzbarkeit der Implementierung und die Effektivität der geschriebenen Bildverarbeitung zu demonstrieren, wurden aufbauend auf der im letzten Kapitel beschriebenen Implementierung konkrete Storages und Evaluatoren implementiert.

Im Folgenden werden erst die Storages und Evaluatoren vorgestellt, im darauf folgenden Abschnitt mit diesen durchgeführte Experimente und deren Ergebnisse.

7.1 Implementierte Storages und Evaluatoren

7.1.1 ImageAndPerceptsStorage

Wie bereits erläutert werden aus einem Bild von der Bildverarbeitung Perzepte extrahiert. Die `ImageAndPerceptsStorage` gruppiert Bilder mit zugehörigen Transformationsmatrizen sowie alle zu einem Bild passenden Perzepte getrennt nach Erzeuger. Evaluatoren können diese Storage nutzen, um auf den erzeugten Perzepten zu arbeiten und beliebige Statistiken zu erstellen.

Hat man mit dem Roboter nur Bilder und Transformationsmatrizen aufgenommen, kann die Storage genutzt werden, um offline die Bilder von verschiedenen Bildverarbeitungen bearbeiten zu lassen und die Perzepte zu speichern. Hierfür startet man als Erstes wie in 6.1.3.2 beschrieben alle gewünschten Bildverarbeitungen in der Gui. Die nächsten Schritte funktionieren analog zu 6.1.2.4. Als Erstes muss der *Record*-Knopf zum Speichern der eintreffenden Datenpakete aktiviert werden und es muss die Anzahl der gestarteten Prozesse eingestellt werden. Sollen nur einzelne Bilder bearbeitet werden, können diese nun manuell ausgewählt werden. Die Storage wird automatisch Bild und Transformationsmatrix verschicken sowie die eingehenden Perzepte speichern. Soll das komplette Logfile bearbeitet werden, geht man folgendermaßen vor: Man wählt den ersten Eintrag im Logfile und drückt den *Play*-Knopf. Es werden jetzt nacheinander alle Bilder von den laufenden Bildverarbeitungen bearbeitet und die generierten Prozesse gespeichert. Die gefüllte Logdatei kann jetzt gespeichert und mit beliebigen Evaluatoren ausgewertet werden.

Die `ImageAndPerceptsStorage` erweitert die Toolbar der Gui um weitere Funktionen.

Wurden Bilder und Perzepte verschiedener Bildverarbeitungen aufgenommen, ist es nicht nur interessant, die Perzepte automatisch auswerten zu lassen. Es ist auch interessant, die Bilder mit passenden Perzepten anzuzeigen und direkt zu vergleichen. Die Storage ermöglicht es, für jede Bildverarbeitung einen sogenannten *ImageViewer* zu starten. Wählt man in der Gui ein Bild aus, zeigt der jeweilige *ImageViewer* das aktuelle Bild und die erkannten Perzepte grafisch an.

Um alle Perzepte aus dem Logfile oder nur die Perzepte, die zu einem speziellen Bild gehören, zu löschen, stehen ebenfalls Knöpfe zur Verfügung.

7.1.1.1 PerceptExistsEvaluator

Aufbauend auf der `ImageAndPerceptsStorage` wurde der `PercetExistsEvaluator` geschrieben. Mit ihm ist es möglich festzustellen, ob in Bildern Perzepte erkannt wurden, beziehungsweise bei welchen Bildern verschiedene Bildverarbeitungen zu unterschiedlichen Ergebnissen kommen.

Wird der Evaluator erzeugt, wählt der Nutzer aus allen verfügbaren Perzepten die für ihn relevanten aus (Abbildung 7.1). Je nachdem ob im Logfile die Perzepte einer oder mehrerer Bildverarbeitungen gespeichert sind, verhält sich der Evaluator unterschiedlich.

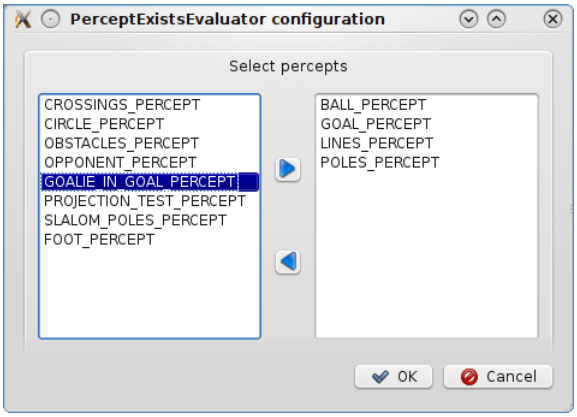


Abbildung 7.1: Auswählen von relevanten Perzepten

Eine Bildverarbeitung Im Falle einer Bildverarbeitung entscheidet der Evaluator nur zwischen *Perzept gefunden* und *Perzept nicht gefunden*. Für jedes vom Nutzer ausgewählte Perzept wird, je nachdem ob es gefunden wurde oder nicht, eine grüne oder rote Fahne gezeichnet. Das Rating eines Bildes ergibt sich aus dem Quotienten der erkannten Perzepte und der Anzahl an ausgewählten Perzepten.

$$Rating = \frac{\text{Anzahl gefundene Perzepte im Bild}}{\text{Maximale Anzahl Perzepte}}$$

Interessiert sich der Nutzer nur für ein Perzept, ergibt sich dadurch ein Rating von eins, wenn das Perzept im Bild gefunden wurde und ein Rating von null, wenn es nicht gefunden wurde.

In der Zusammenfassung wird für jedes Perzept angezeigt, wie oft es erkannt wurde.

Mehrere Bildverarbeitungen Werden mehrere Bildverarbeitungen, also Quellen von Perzepten, im Logfile gefunden, wird eine ähnliche grafische Darstellung gewählt. Jedes gewählte Perzept erhält eine Spalte, jede Datenquelle eine Zeile. Gefundene Perzepte mit einer grünen, nicht gefundene mit einer roten Fahne visualisiert.

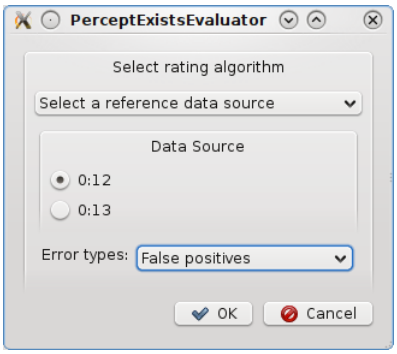


Abbildung 7.2: Auswahl des gewünschten Ratingalgorithmus'

Der genutzte Ratingalgorithmus kann zur Laufzeit geändert werden (Abbildung 7.2). Standardmäßig werden alle Datenquellen gleichbehandelt. Das Rating ergibt sich aus dem Quotienten der Perzepte, bei denen sich die Datenquellen unterscheiden und der Anzahl an ausgewählten Perzepten.

$$Rating = \frac{\text{Anzahl Perzepte mit unterschiedlichen Ergebnissen der Datenquellen}}{\text{Maximale Anzahl Perzepte}}$$

Neben der Gleichbehandlung aller Datenquellen ist es möglich, eine der Datenquellen als Referenzdatenquelle zu wählen. Die gewählte Datenquelle liefert für die weiteren Annahmen per Definition richtige Ergebnisse. Unterscheiden sich die Ergebnisse der Referenzdatenquelle und einer anderen Datenquelle ist dies entweder eine falsch positive (*false positive*) oder eine falsch negative (*false negative*) Erkennung.

falsch positive Erkennung Die Referenzquelle hat kein Perzept erkannt, die andere Datenquelle jedoch fälschlicherweise.

falsch negative Erkennung Die Referenzquelle hat ein Perzept erkannt, die andere Datenquelle jedoch fälschlicherweise nicht.

Je nach gewünschtem Algorithmus entspricht das Rating entweder der prozentualen Anzahl an falsch positiven, der prozentualen Anzahl an falsch negativen Erkennungen oder der

prozentualen Anzahl an Abweichungen von der Referenzbildverarbeitung. Sollen falsch positive Erkennungen betrachtet werden, ergibt sich

$$Rating = \frac{\text{Anzahl falsch positive Erkennungen}}{(\text{Anzahl Datenquellen} - 1) * \text{Maximale Anzahl Perzepte}}$$

Falsch negative Erkennungen folgen analog.

Wurden zum Beispiel die Perzepte von zwei Bildverarbeitungen aufgenommen, ist es je nach Auswahl des Ratingalgorithmus möglich

- alle Bilder zu finden in denen sich die Bildverarbeitungen unterscheiden,
- alle Bilder zu finden in denen die erste Bildverarbeitung ein Perzept erkennt, die Zweite jedoch nicht
- oder alle Bilder zu finden in denen die zweite Bildverarbeitung ein Perzept erkennt, die Erste jedoch nicht.

Ist bekannt, dass sich in jedem Bild ein bestimmtes Perzept befindet, ist es interessant, alle Bilder auszuwählen, in denen mindestens eine der Bildverarbeitungen kein Perzept erkannt hat. Hierbei handelt es sich garantiert um eine Fehlerkennung. Das Rating wird in diesem Fall aus dem Quotienten aller Perzepte, die mindestens eine Bildverarbeitung nicht erkannt hat, und der Gesamtzahl an ausgewählten Perzepten gebildet.

In der Zusammenfassung für jede Datenquelle ist aufgelistet, wie oft die gewählten Perzepte erkannt wurden. Zusätzlich wird für jede paarweise Kombination aus Quellen angegeben, wie oft eine Datenquelle ein Perzept erkannt hat, ohne dass die andere Datenquelle das Perzept erkannt hat. Dies entspricht der Anzahl der falsch positiven und falsch negativen Erkennungen, wenn man immer eine der Datenquellen als Referenz nimmt (Abbildung 7.3).

	0:7	0:8	0:7 > 0:8	0:8 > 0:7
POLES_PERCEPT	97	117	8	28
GOAL_PERCEPT	8	0	8	0

Abbildung 7.3: Zusammenfassung des `PercetExistsEvaluator` mit zwei Datenquellen und zwei Perzepten. Die erste Datenquelle erkannte 97, die Zweite 117 Poles. In 8 Fällen konnte die erste Datenquelle Poles erkennen, die Zweite jedoch nicht. Umgekehrt erkannte die zweite Datenquelle 28 Poles, welche die Erste nicht erkannte.

7.1.2 SimpleDataStorage

Für viele praktische Anwendungsfälle wird eine nicht so komplexe Storage wie `ImageAndPerceptsStorage` benötigt. Oft ist es vollkommen ausreichen, alle Datenpakete eines bestimmten Typs zu betrachten. In *RoboFrame* werden Datenpakete eines Typs durch sogenannte Keys identifiziert. `SimpleDataStorage` stellt zu einem gegebenem Logfile den Evaluatoren alle Datenpakete eines bestimmten Typs zur Verfügung. Datenpakete aus verschiedenen Quellen werden nach Zeitstempeln gruppiert. Mit `SimpleDataStorageTemplated`

kann zur Kompilierzeit ein `SimpleDataStorage` mit einem speziellen Key, der als Templateparameter übergeben wird, erstellt werden. Werden Storages und Evaluatoren korrekt bei den Factorys registriert (siehe 6.1.1.3) ist es nicht möglich, zu einer Storage Evaluatoren zu erzeugen, die einen anderen Key erwarten. Trotzdem sollten Evaluatoren zur Laufzeit prüfen, ob der Key der Storage und der erwartete Key zusammenpassen.

7.1.2.1 BallPerceptStatisticsTableEvaluator und BallPerceptDistanceGraphEvaluator

Einer der wichtigsten Erkenner in der Humanoidliga ist der Ballerkenner. Ohne erkannte Bälle kann weder ein Stürmer einen Ball ins Tor schießen noch kann der Torwart einen Ball, der auf sein Tor geschossen wird, halten. Im Idealfall könnte man den vom Roboter erkannten Ball mit der exakten Position des Balls in der wirklichen Welt vergleichen. Über die Differenz aus erkannter Position und tatsächlicher Position könnte man Aussagen über die Ballerkennung, zum Beispiel währen eines Spiels, treffen. Leider fehlt den *Darmstadt Dribblers* im Moment die Möglichkeit, über eine externe Quelle wie eine Deckenkamera, die wirkliche Position des Balles zu bestimmen.

Da um verifizierbare Aussagen über die Ballerkennung zu treffen die tatsächliche Position von Ball zu Roboter bekannt sein muss, kann die Ballerkennung nur sinnvoll bei stehendem Roboter und liegendem Ball bewertet werden. Um diesen Prozess zu unterstützen, wurden die Evaluatoren `BallPerceptStatisticsTableEvaluator` und `BallPerceptDistanceGraphEvaluator` geschrieben. Beide arbeiten nur auf einer Liste von Ballperzepten. Der Ballerkenner berechnet die Ballentfernung auf zwei Arten und speichert beide im Perzept. Es ist die Aufgabe der mit dem Ballperzept arbeitenden Ballmodellierung, aus diesen beiden Positionen die angenommene Ballposition zu bestimmen. Die erste Ballposition wird ermittelt, indem aus der Position des gefundenen Balls im Bild und der kinematischen Kette des Roboters die Entfernung und der Winkel des Balles über Projektion bestimmt werden. Steht der Roboter, ist die ermittelte Position sehr exakt. Bewegt sich der Roboter jedoch, ist die Projektion nicht mehr zuverlässig. Daher gibt es eine zweite Methode die Entfernung des Balles zu bestimmen. Da die Größe des Balles in der Wirklichkeit genau bekannt, kann aus der Größe des Balles im Bild seine Entfernung bestimmt werden. Dies funktioniert jedoch nur genau, wenn der komplette Ball gut erkannt werden konnte. Die Ergebnisse sind sehr von der Beleuchtung und damit der Farbtabelle abhängig.

BallPerceptStatisticsTableEvaluator

`BallPerceptStatisticsTableEvaluator` erstellt für jede Perzeptquelle für beide Ballpositionen folgende Statistiken:

- Anzahl bisher bearbeiteter Ballperzepte
- durchschnittliche Ballentfernung
- Varianz der Ballentfernung

- durchschnittlicher Winkel zwischen Ball und Roboter
- Varianz des Winkels zwischen Ball und Roboter

Die errechneten Werten können mit der tatsächlichen Ballposition verglichen werden, um Aussagen über die Ballerkennung zu treffen.

BallPerceptDistanceGraphEvaluator

`BallPerceptDistanceGraphEvaluator` plottet die in den Perzepten gespeicherten projektions- und die größenbasierten Ballentfernungen. Perzepte aus verschiedenen Datenquellen werden in verschiedenen Kurven dargestellt. Der in der Gui ausgewählte Zeitstempel wird durch eine vertikale Linie visualisiert.

Durch die grafische Darstellungen kann man sehr schnell Ausreißer in der Erkennung sowie Unterschiede zwischen den beiden ermittelten Ballentfernungen oder zwischen verschiedenen Bildverarbeitungen erkennen (Abbildung 6.13).

7.1.2.2 ValidityPoseTableEvaluator und ValidityPosePlotter

Neben dem Erkennen des Balls ist es sehr wichtig, dass der Roboter seine Position auf dem Spielfeld korrekt ermittelt. Diesen Prozess nennt man Selbstlokalisierung. Mit einer falsch erkannten Position läuft der Roboter nicht korrekt an den Ball und kann kein Tor erzielen. Wie auch bei der Ballerkennung wäre es wünschenswert, die vom Roboter ermittelte Position mit einer externen Quelle wie einer Deckenkamera vergleichen zu können. Auch wenn diese Referenzdaten nicht zur Verfügung stehen, kann man in den vom Roboter berechneten Positionen nach typischen Fehlern suchen.

Ein Fehler sind springende Roboterpositionen. Ein Roboter wird sich immer mit begrenzter Geschwindigkeit bewegen. Ist die Differenz zweier aufeinanderfolgender Positionen größer als die Strecke, die der Roboter sich in der Zeit zwischen den Datenpaketen bewegen konnte, ist mindestens eines dieser Datenpakete falsch. Neben zufälligen Sprüngen in der Selbstlokalisierung ist ein häufiges Problem, dass die vom Roboter ermittelte Position sich am Mittelpunkt spiegelt. Die vom Roboter geschätzte Position springt also zum Beispiel von rechts vor dem gelben Tor zu rechts vor dem blauen Tor.

Um diese beiden offensichtlichen und weitere Fehler zu erkennen und danach untersuchen zu können, stehen zwei Evaluatoren zur Verfügung.

ValidityPoseTableEvaluator

Der `ValidityPoseTableEvaluator` vergleicht jedes Paket der Selbstlokalisierung mit seinem zeitlichen Vorgänger. Es wird die aktuelle Position, die vergangene Zeit seit dem letzten Paket sowie die Differenz der Positionen und die sich daraus ergebende Geschwindigkeit angezeigt.

Wird eine zu große Geschwindigkeit oder eine gespiegelte Position festgestellt, wird das Rating von eins auf null gesetzt und eine entsprechende Nachricht angezeigt.

Die Zusammenfassung besteht aus der Anzahl an zu hohen Geschwindigkeiten und gespiegelten Positionen.

ValidityPosePlotter

Der `ValidityPosePlotter` zeichnet die vom Roboter geschätzten Positionen in einen Graphen ein. Die vom Roboter ermittelten Positionen können so einfach mit einer tatsächlich gelaufenen Trajektorie vergleichen werden.

7.2 Durchgeführte Experimente

Mithilfe der im letzten Abschnitt vorgestellten Storages und Evaluatoren wurden einige Experimente durchgeführt und ausgewertet.

7.2.1 Vergleich von YUV- und HSV-Farbtabellen

In dieser Arbeit wurde bisher nur auf externe Quellen verwiesen, um zu zeigen, dass der HSV-Farbraum stabiler gegen sich ändernde Lichtbedingungen ist als der YUV-Farbraum.

Wie in 3.2.1 beschrieben wird die Farbklassifizierung der aktuellen Bildverarbeitung auf Basis von Lookup-Tabellen, sogenannten Farbtabellen durchgeführt. Bisher werden die YUV-Pixel des Kamerabildes über YUV-Farbtabellen klassifiziert. Zusätzlich zu den YUV-Farbtabellen wurde die Möglichkeit geschaffen, Bilder im HSV-Farbraum klassifizieren zu können. Hierfür werden die Pixel vor dem klassifizieren wie in 3.3.1.3 beschrieben aus dem YUV- über den RGB- in den HSV-Farbraum umgerechnet. Wie in 3.2.1 beschrieben, dürfen hierfür die einzelnen Kanäle maximal 16 Bit belegen. Mit einem 6 Bit H- und S- sowie einem 3 Bit V-Kanal konnten gute Klassifizierungsergebnisse erzielt werden. Für den V-Kanal würde noch ein viertes Bit zur Verfügung stehen, allerdings ist dieses nicht nötig und würde die Farbtabellen nur weniger robust gegen Änderungen der Lichtbedingungen machen. Wie beschrieben nutzen die *Darmstadt Dribblers* in der Praxis für die YUV-Farbtabellen einen Y-Kanal mit 6 Bit. Da gerade der Y-Kanal jedoch sehr anfällig für sich ändernde Lichtbedingungen ist, wurde für die folgenden Tests, analog zum H-Kanal, nur mit 3 Bit gearbeitet, um die HSV- und YUV-Farbtabellen besser vergleichen zu können.

7.2.1.1 Experimentaufbau

Um die beiden Farbräume im praktischen Einsatz vergleichen zu können, wurde folgendes Experiment durchgeführt: Mit einem DD2009 wurden jeweils etwa 100 Bilder und Transformationsmatrizen von Bällen in einer Entfernung von 50 cm, 100 cm, 150 cm und 200 cm bei jeweils 1000 lx, 900 lx, 800 lx, 700 lx, 600 lx und 500 lx aufgenommen. Anhand des Logfiles mit Bildern von Bällen in 50 cm Entfernung bei 1000 lx wurden eine HSV- und eine YUV-Farbtabelle für Orange und Grün erzeugt. Die Farbtabellen wurden bewusst nur mit einer

begrenzten Anzahl Bilder erstellt, um garantieren zu können, dass beide Farbtabellen in etwa die gleichen Pixel klassifizieren und die Auswahl der Pixel das Ergebnis nicht verfälscht.

In der Gui wurde für die YUV- und HSV-Farbtabelle je ein Bildverarbeitungsprozess gestartet. Mithilfe der `ImageAndPerceptsStorage` in der EvaluatorGui wurden für alle Logdateien alle Ballperzepte bei unterschiedlichen Ballentfernungen und Beleuchtungsbedingungen von beiden Bildverarbeitungen gespeichert. Die erkannten Ballperzepte wurden mit dem `BallPerceptStatisticsTableEvaluator` ausgewertet.

7.2.1.2 Ergebnisse

Die aufgenommenen Messwerte finden sich in Tabelle 7.1. Für beide Farbräume gilt wie erwartet, dass eine geringere Beleuchtunge zu schlechteren Erkennungsraten führt. Je weiter Bälle vom Roboter entfernt sind, desto weniger Pixeln entsprechen sie im Bild. Dies erschwert die Erkennung und macht sie besonders empfindlich für geänderte Lichtbedingungen. Auch diese Tendenz lässt sich für beide Farbräume in den Messwerten erkennen. Neben der Erkennungsrate lassen vor allem der durchschnittliche Abstand und die Standardabweichung der größenbasierten Erkennung auf die Qualität der Farbtabellen schließen. Je weniger Pixel korrekt als Orange klassifiziert werden, desto kleiner wird der im Bild erkannte Ball. Er erscheint dadurch weiter entfernt, als er tatsächlich ist.

Vergleicht man die Werte für den HSV- und YUV-Farbraum, fällt auf, dass die Erkennungsraten mit der HSV-Klassifizierung immer mindestens so gut, meistens aber signifikant besser als die Erkennungsraten mit der YUV-Klassifizierung sind. Nicht nur bei sich ändernder Beleuchtung, sondern auch bei gleicher Lichtstärke jedoch einem geänderten Ballabstand gegenüber der initialen Farbtabelle, ist der HSV- dem YUV-Farbraum überlegen. Bereits bei einem Ballabstand von 50 cm und einer Erkennungsrate von 100% mit beiden Farbräumen kann man erkennen, dass mit YUV-Klassifikation die Bälle weiter entfernt erkannt werden, die Klassifikation also schlechter ist. Auch bei der Standardabweichung erreicht die HSV-Klassifizierung meist kleinere Werte.

Mit diesen Tests konnte gezeigt werden, dass bei gleicher Bildverarbeitung eine Farbklassifikation im HSV-Farbraum dem YUV-Farbraum bei sich ändernden Beleuchtungsbedingungen im Allgemeinen überlegen ist. Es wurde in dieser Arbeit nur die Ballerkennung getestet. Da diese den bisherigen Erfahrungen nach jedoch immer die kritischste Komponente war, ist zu erwarten, dass auch die anderen Erkenner von einer Klassifizierung im HSV-Farbraum profitieren würden. Die Klassifizierung im HSV-Farbraum hat jedoch einen Nachteil: Die Umrechnung vom YUV- in den HSV-Farbraum ist relativ teuer. Vor einem Einsatz des HSV-Farbraums muss die Auswirkung auf die Performance erst in der Praxis geprüft werden.

	Erkennungs-		Durchschnittlicher Abstand				Standardabweichung			
	rate (%)		Projiziert		Größenbasiert		Projiziert		Größenbasiert	
Lux	HSV	YUV	HSV	YUV	HSV	YUV	HSV	YUV	HSV	YUV
50 cm										
1000	100	100	470	472	538	575	3	3	27	28
900	100	100	470	472	523	566	3	3	18	23
800	100	100	472	473	530	574	3	3	18	46
700	100	100	474	473	553	567	3	3	35	49
600	100	75	479	477	665	668	20	23	85	90
500	99	48	477	480	699	765	4	7	87	111
100 cm										
1000	91	91	971	972	1128	1213	9	8	39	73
900	98	98	972	971	1167	1259	7	8	50	83
800	100	89	972	973	1211	1321	8	9	67	99
700	95	77	973	974	1286	1399	11	11	113	162
600	85	29	974	977	1337	1488	11	12	161	148
500	38	6	975	981	1350	1395	10	7	157	106
150 cm										
1000	90	35	1465	1466	1858	1941	21	17	112	149
900	69	12	1473	1461	1922	2012	20	19	150	169
800	46	12	1469	1461	1949	2152	19	22	129	191
700	41	5	1471	1488	2064	2331	13	28	157	18
600	26	2	1462	1470	2175	2306	24	3	131	16
500	2	0	1480	n/a	2285	n/a	0	n/a	0	n/a
200 cm										
1000	93	12	2147	2159	2422	2551	11	6	163	161
900	93	59	2144	2149	2520	2700	14	14	155	205
800	96	24	2155	2156	2647	2788	13	11	184	182
700	84	12	2152	2146	2431	2617	9	4	111	167
600	37	0	2150	n/a	2719	n/a	10	n/a	126	n/a
500	0	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Tabelle 7.1: Erkennungsrate, durchschnittlicher Abstand und dessen Standardabweichung bei Ballerkennungen unter unterschiedlichen Lichtbedingungen und bei unterschiedlichen Entfernungen. Ergebnisse bei Farbklassifikation im HSV- und YUV-Farbraum. Abstände werden in Millimetern angegeben.

7.2.2 Test neuer Poleserkennung / Vergleich mit bestehender Bildverarbeitung

Die neue Bildverarbeitung wurde basierend auf Kamerabildern aus dem Labor der *Darmstadt Dribblers* unter verschiedenen Lichtbedingungen und Kamerabildern vom RoboCup 2009 in Graz entwickelt und getestet.

7.2.2.1 Experimentaufbau

Um die Poleserkennung der neu geschriebenen und der bestehenden Bildverarbeitung unter definierten Bedingungen zu vergleichen, wurde folgendes Experiment, ähnlich dem Test der Ballerkennung, durchgeführt: Eine Pole[1] wurde auf ihrer Position am Rand des Spielfelds einmal vor einem grauen Schrank, einmal vor weißem Hintergrund platziert. Durch die Änderung des Hintergrunds wurde die Kantenerkennung mit zwei sehr unterschiedlichen Fällen konfrontiert. Mit dem Roboter wurden Bilder und Kameramatrizen aufgenommen. Die Beleuchtung wurde hierbei zwischen 1000 lx und 100 lx variiert. Um die verschiedenen Helligkeiten zu erhalten, wurde die Beleuchtung aus Halogenstrahlern und Deckenlampen gemischt. Die hierbei entstehende Schwierigkeit für die Bildverarbeitung ist, dass sich nicht nur die Helligkeit, sondern in gewissem Maße auch die Farbtemperatur des Lichts ändert.

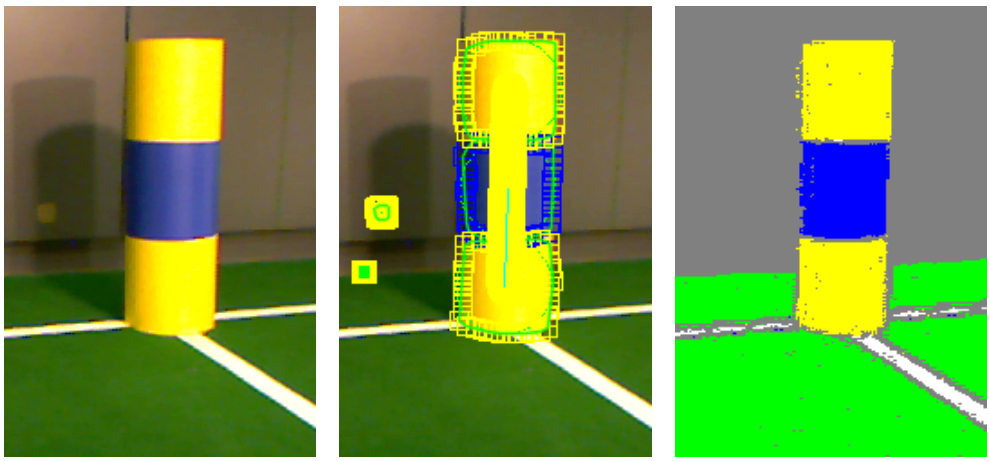
Die aufgenommenen Bilder wurden sowohl mit der neuen als auch mit der bestehenden Bildverarbeitung verarbeitet und die generierten Polesperzepte gespeichert. Die bestehende Bildverarbeitung arbeitete hierbei mit einer bei 1000 lx erstellten Farbtabelle in der Gelb, Blau, Grün und Weiß klassifiziert wurden.

7.2.2.2 Ergebnisse

Abbildungen 7.4 bis 7.12 sind repräsentative Beispiele für aufgenommene Bilder unter verschiedenen Beleuchtungsbedingungen. Jede Abbildung zeigt immer ein Kamerabild, das Kamerabild in dem die neue Bildverarbeitung erkannte Blobs und Poles eingezeichnet hat sowie das farbklassifizierten Bild, auf dem die bestehende Bildverarbeitung arbeitet. Für beide bearbeiteten Bilder ist angegeben, ob die Bildverarbeitung die Pole erkannt hat oder nicht.

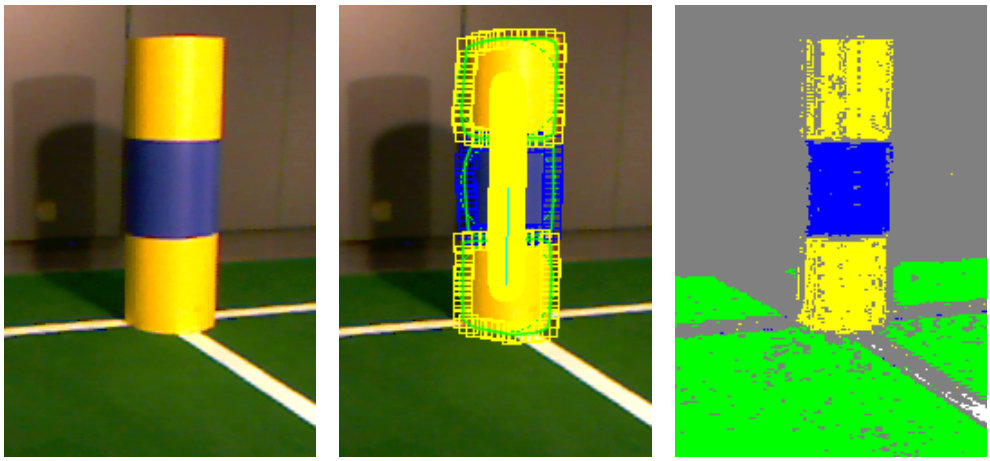
Die neu geschriebene Bildverarbeitung erkennt Poles zuverlässig zwischen 1000 lx und etwa 300 lx. Unterhalb von 300 lx werden teilweise Poles noch korrekt erkannt (Abbildungen 7.10 und 7.11), allerdings werden sie verworfen, da der Boden unter den Poles zu dunkel ist und nicht mehr als Grün klassifiziert wird. Es wäre möglich, den Bereich in dem Pixel als Grün klassifiziert werden weiter zu vergrößern, allerdings ist bei Pixeln mit einem niedrigen Helligkeitsanteil keine zuverlässige Klassifizierung mehr möglich (siehe 3.3.1.3). Wie in Abbildung 7.10 zu erkennen, ist neben der Klassifikation von Grün, auch die Klassifikation von Blau bei 200 lx bereits schwierig. Bei 100 lx sind die blauen Teile der Pole sehr nah bei

[1] Das Experiment wurde sowohl mit der blauen als auch mit der gelben Pole durchgeführt.



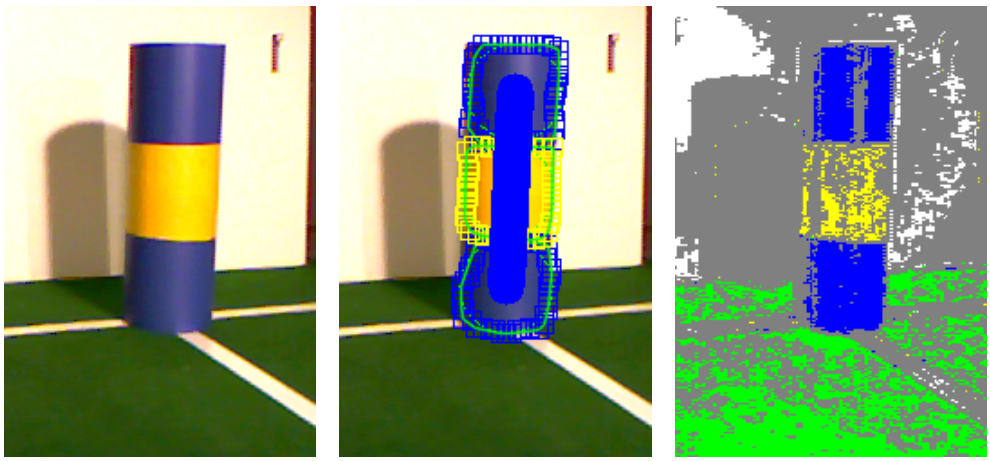
(a) (b) erkannt (c) erkannt

Abbildung 7.4: Aufnahme bei etwa 1000 lx



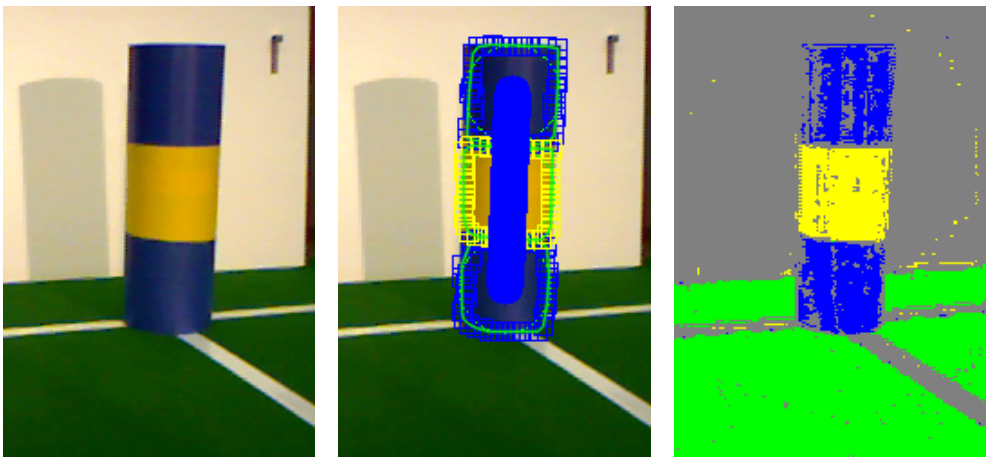
(a) (b) erkannt (c) erkannt

Abbildung 7.5: Aufnahme bei etwa 700 lx



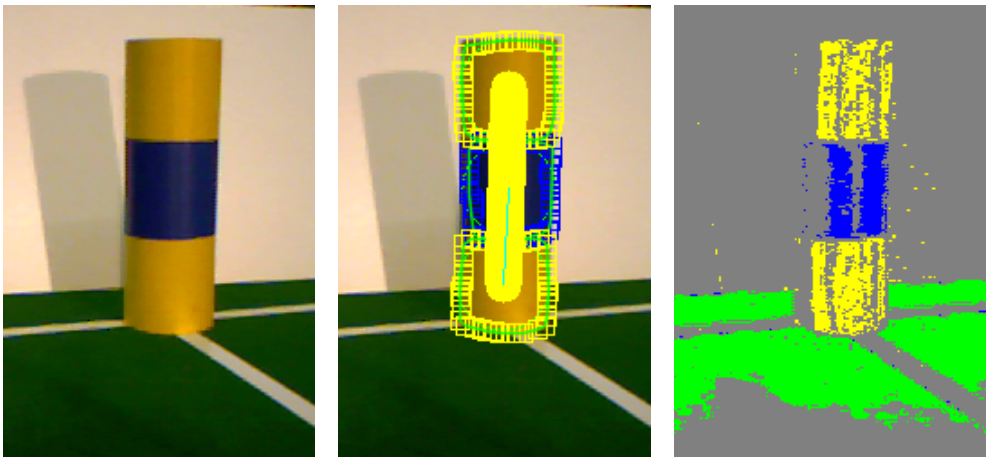
(a) (b) erkannt (c) nicht-erkannt

Abbildung 7.6: Aufnahme bei etwa 700 lx



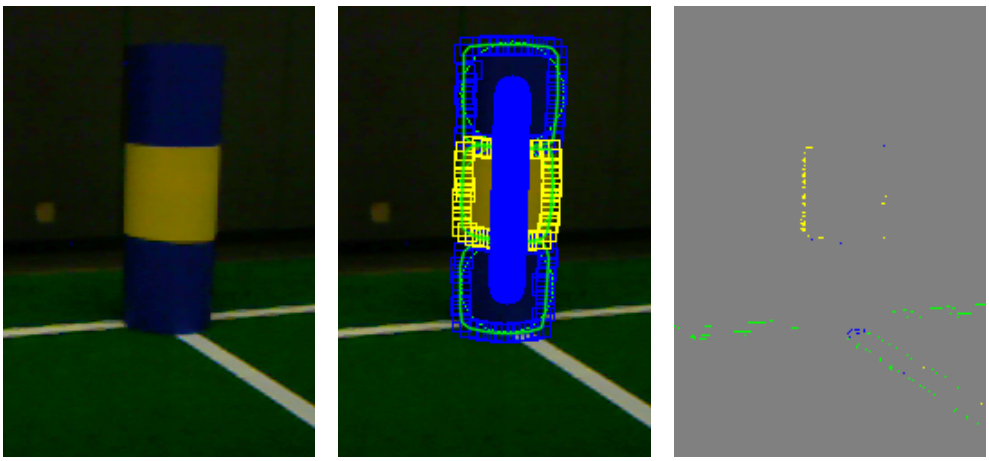
(a) (b) erkannt (c) erkannt

Abbildung 7.7: Aufnahme bei etwa 600 lx



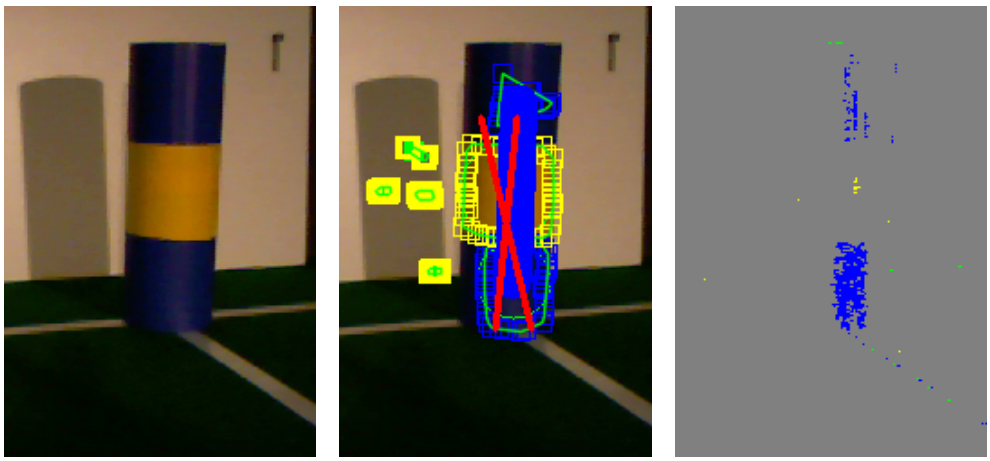
(a) (b) erkannt (c) nicht-erkannt

Abbildung 7.8: Aufnahme bei etwa 400 lx



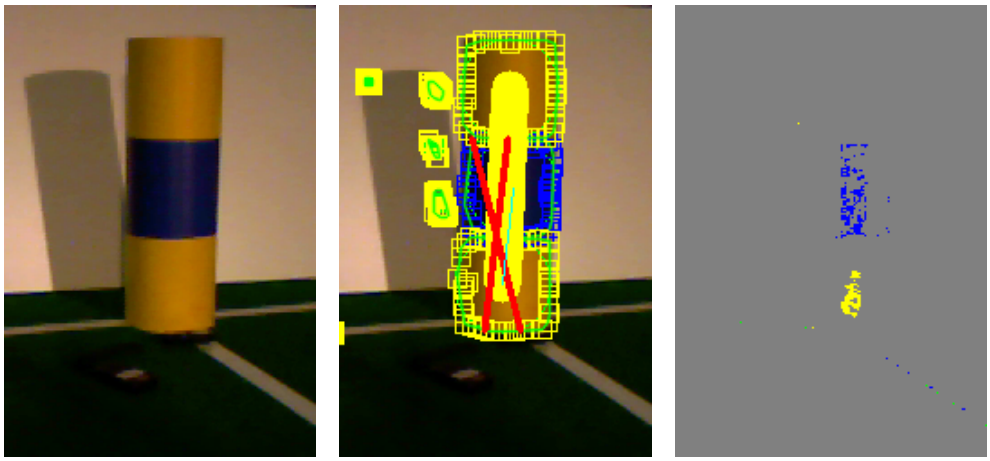
(a) (b) erkannt (c) nicht-erkannt

Abbildung 7.9: Aufnahme bei etwa 300 lx



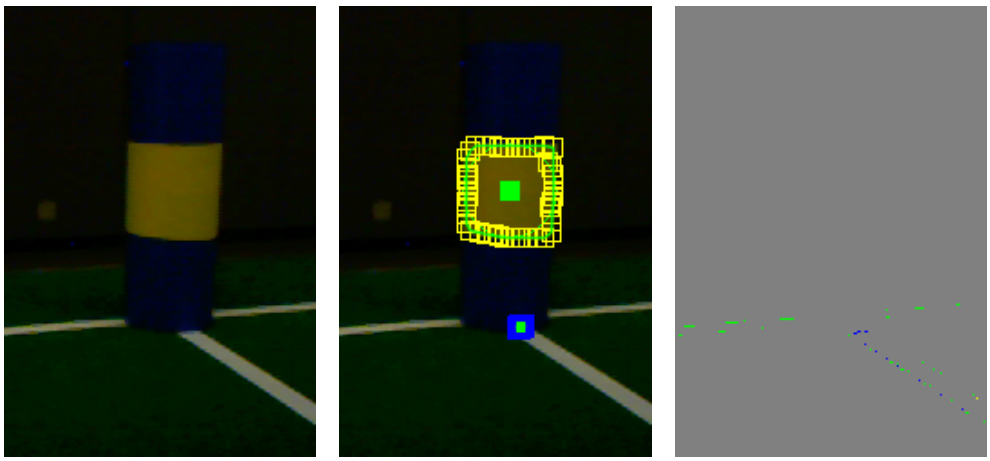
(a) (b) nicht-erkannt (c) nicht-erkannt

Abbildung 7.10: Aufnahme bei etwa 200 lx



(a) (b) nicht-erkannt (c) nicht-erkannt

Abbildung 7.11: Aufnahme bei etwa 200 Lux



(a) (b) nicht-erkannt (c) nicht-erkannt

Abbildung 7.12: Aufnahme bei etwa 100 lx

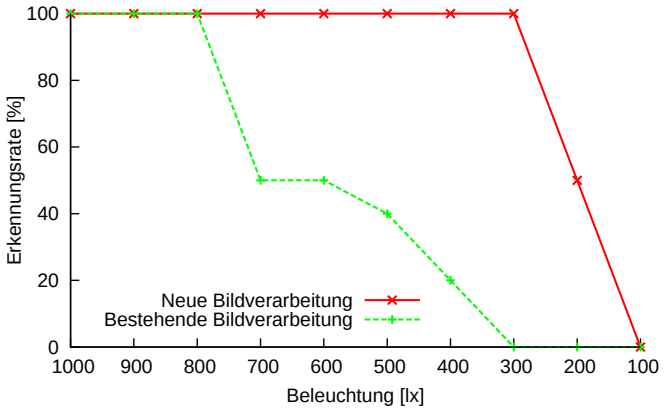


Abbildung 7.13: Erkennungsraten der der Polesperzeptoren beider Bilderkennungen im Vergleich

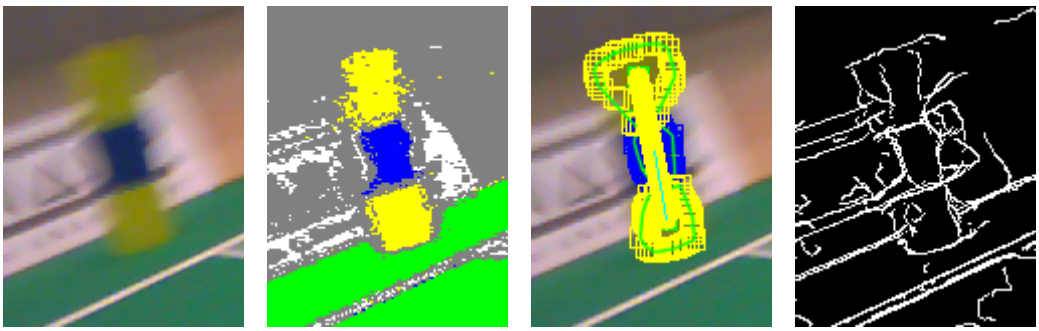
Schwarz und werden nicht mehr klassifiziert. Der gelbe Teil dagegen kann weiterhin korrekt erkannt werden.

Im Vergleich mit der neuen Bildverarbeitung ist die bestehende Bildverarbeitung durch Nutzung von Farbtabellen sehr empfindlich für geänderte Lichtbedingungen. Wie in Abbildungen 7.4 bis 7.12 zu sehen nimmt die Anzahl der klassifizierten Pixel bei niedrigerer Luxzahl ab. Vergleicht man Abbildung 7.5, 7.6 und 7.7 fällt auf, dass nicht nur die reine Luxzahl für die Klassifizierung verantwortlich ist. Bild 7.5 und 7.6 wurden beide bei etwa 700 lx aufgenommen, allerdings mit unterschiedlich gemischtem Licht. Dies führt dazu, dass in Abbildung 7.5 die Farbklassifikation ausreichend ist, um eine Pole zu erkennen, bei gleicher messbarer Helligkeit in Bild 7.6 jedoch keine Pole mehr erkannt werden kann. Das bei 600 lx aufgenommene Bild 7.7 dagegen kann noch so gut klassifiziert werden, dass eine Pole erkannt wird. Bei etwa 400 lx ist noch eine Klassifikation möglich anhand derer für einen Menschen problemlos eine Pole erkennbar ist. Da die aktuelle Bildverbeitung jedoch auf einem Gitter arbeitet (siehe 3.2.1) sind nicht genug Pixel klassifiziert, um eine Pole zuverlässig erkennen zu können. Unterhalb von 300 lx werden kaum noch Pixel klassifiziert und das Erkennen von Poles anhand der klassifizierten Bilder ist auch theoretisch nicht möglich.

Abbildung 7.13 fasst die Erkennungsraten der verschiedenen Tests zusammen. Die bestehende Bildverarbeitung bricht sehr schnell ein und erreicht bereits ab 700 lx nur noch eine Erkennungsrate um 50%. Wie beschrieben hängt die Erkennungsrate jedoch nicht nur von der Helligkeit, sondern auch von Farbtönen der Lichtquellen ab. Die bestehende Bildverarbeitung im Vergleich erreicht bis 300 lx eine Erkennungsrate von 100%, bricht darunter jedoch sehr schnell ein.

Wird die bisherige Bildverarbeitung mit einer der Beleuchtung angepassten Farbtabelle gestartet, kommen beide Bildverarbeitungen bei gleichbleibendem Licht zu sehr guten Ergebnissen von nahezu 100%. Ändern sich die Lichtbedingungen, ist die neue Bildverarbeitung klar im Vorteil. Zusätzlich liefert sie ohne aufwendige und langwierige Kalibrierung sofort korrekte Ergebnisse.

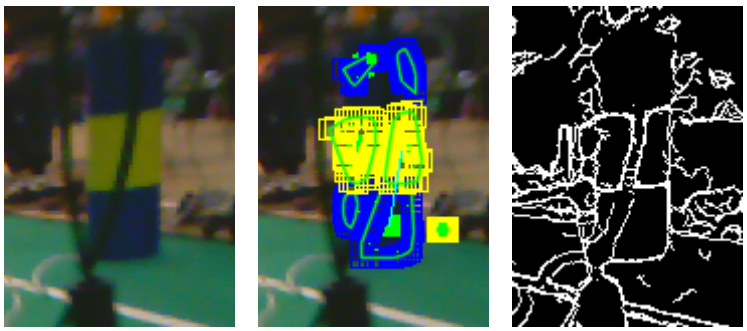
Beide Bildverarbeitungen sind jedoch nicht perfekt. Sie erkennen die meisten Poles, unter speziellen Bedingungen kann es jedoch passieren, dass Poles nicht erkannt werden.



(a) Kamerabild (b) Klassifiziertes Bild (c) Erkannte Pole (d) Erkannte Kanten

Abbildung 7.14: Kamerabild mit Pole welche von der neuen, jedoch nicht der bestehenden Bildverarbeitung erkannt wird.

Abbildung 7.14 zeigt einen Fall, in dem die bisherige Bildverarbeitung eine Pole nicht erkennen konnte. Zum Zeitpunkt der Aufnahme bewegt der Roboter den Kopf sehr schnell, wodurch das Bild unscharf wird. Zusätzlich erscheint die Pole nicht senkrecht im Bild, sondern leicht gedreht. Die bisherige Bildverarbeitung sucht auf senkrechten Linien nach gelb-blau-gelb-Übergängen. Im gedrehten und verzerrten Fall findet sie nicht genug dieser Übergänge und verwirft die Pole. Die neue Bildverarbeitung erkennt die Pole korrekt. Wie man sieht, ist jedoch die Kantenerkennung bei unscharfen Bildern schwierig und die entstehenden Blobs daher nicht exakt quadratisch.



(a) Kamerabild (b) Nichterkannte Pole (c) Erkannte Kanten

Abbildung 7.15: Kamerabild mit Pole, welche von der neuen Bildverarbeitung nicht erkannt wird

Bild 7.15 ist ein Fall, indem die neue Bildverarbeitung eine Pole verwirft. Das Problem ist, dass die Pole im Bild durch ein Kabel in zwei Teile getrennt wird. Hierdurch zerfallen die gelben und blauen Teile der Pole in mehrere farbige Blobs. Da jedoch nach Definition keine gleichfarbigen Blobs im RoboCup direkt nebeneinander auftreten, sollten die gleichfarbigen Blobs vereinigt werden. Auf diese Art und Weise könnte die Pole korrekt erkannt werden.

7.2.2.3 Neue Bilderkennung in der Praxis

Im praktischen Einsatz haben die neue und bestehende Poleserkennung mit geeigneter Farbtabelle beide eine fast 100-prozentige Erkennungsrate. Es wurde gezeigt, dass die neue Poleserkennung unter verschiedensten Lichtbedingungen ohne Kalibrieren direkt einsatzbereit ist. Die beiden Bildverarbeitungen können genutzt werden, um wechselseitig Fehler in der jeweils anderen zu finden. Zusätzlich kann die neue Erkennung dafür genutzt werden, Farbtabellen zu überprüfen.

Für den praktischen Einsatz sind die neue Bildverarbeitung beziehungsweise Teile von ihr aus zwei Gründen noch nicht geeignet. Zum einen erreicht die neue Bildverarbeitung bisher noch nicht den Leistungsumfang der alten Bildverarbeitung. Der einzige bisher vollständig implementierte Perzeptor ist der Poleserkenner. Ein weiteres und bei aktueller Hardware größeres Problem ist die erreichbare Performance.

Performance

Die neue Bildverarbeitung ist in ihrer bisherigen Implementierung etwa um den Faktor 40 langsamer als die bestehende Bildverarbeitung ist. Hierbei muss man bedenken, dass die bestehende Bildverarbeitung nur auf einem Bruchteil des Bildes arbeitet und in den letzten Jahren kontinuierlich optimiert wurde. Die bestehende Bildverarbeitung nutzt viel komplexere Algorithmen und ist bisher nur eine Proof-of-Concept-Implementierung. Es wurden noch keine Performanceoptimierungen, wie beispielsweise gezieltes Profiling, vorgenommen. Über die Hälfte der Rechenzeit verbringt die neue Bildverarbeitung damit, die Kamerabilder vom YUV- über den RGB- in den HSV-Farbraum umzurechnen und dort anhand der Schwellwerte binarisierte Bilder zu erstellen. Dieser Prozess ist derzeit einfach, aber sehr ineffizient implementiert und kann noch hochgradig optimiert werden. Auch die Kantenerkennung nimmt relativ viel Zeit in Anspruch. Diese kann reduziert werden, indem die Kanten nicht auf dem gesamten Bild, sondern nur in interessanten Regionen berechnet werden. Neben Optimierungen der Algorithmen selbst ist auch zu bedenken, dass die verfügbare Rechenleistung für mobile Systeme in den nächsten Jahren massiv steigen wird und dadurch komplexere Algorithmen ermöglicht werden.

Kapitel 8

Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Infrastruktur zur Evaluierung beliebiger Algorithmen geschrieben. Die Infrastruktur selbst ist dabei generisch und unabhängig von den auszuwertenden Algorithmen. Gespeicherte Daten können erneut abgespielt und von in der GUI laufenden Algorithmen komplettiert werden. Die Ergebnisse der Evaluierung können auf verschiedene Art und Weise, beispielsweise in tabellarischer oder grafischer Form, dargestellt werden.

Die Funktionalität der Infrastruktur wurde durch die Implementierung von verschiedenen Storagemodulen und Evaluatoren gezeigt. Diese wurde benutzt, um Experimente durchzuführen und auszuwerten. Es konnte gezeigt werden, dass der HSV- dem YUV-Farbraum bei sich ändernden Lichtbedingungen überlegen ist. Dem HSV-Farbraum konnte beim Einsatz von Farbtabellen eine bessere Farbklassifizierung und Ballerkennung nachgewiesen werden.

Basierend auf Farbklassifikation und Kantenerkennung wurden die Grundlagen einer neuen Bildverarbeitung geschaffen. Entgegen der bisherigen Bildverarbeitung ist keine manuelle Kalibrierung mehr nötig. Die neue Bildverarbeitung ist unter verschiedensten Lichtbedingungen direkt nutzbar. Sie nutzt hierfür aus, dass über den H-Kanal des HSV-Farbraums sehr zuverlässig die Farben im Bild klassifiziert werden können. Auf Basis des klassifizierten Bildes werden über die Kantenerkennung sogenannte Blobs erstellt, die einfarbigen Flächen im Bild entsprechen. Durch Zusammenfügen von Blobs und Plausibilitätsprüfungen werden schließlich komplette Objekte erkannt.

Neben der Umsetzung der nötigen Grundlagen wurde die Erkennung von Poles komplett implementiert und getestet. Im Vergleich mit der bestehenden Bildverarbeitung konnten keine signifikanten Unterschiede zwischen beiden Bildverarbeitungen festgestellt werden, wenn die bestehende Bildverarbeitung eine zur Beleuchtung passende Farbtabelle nutzt. Ändern sich die Lichtbedingungen, brechen die Erkennungsraten der bestehenden Bildverarbeitung massiv ein. Die neue Bildverarbeitung dagegen erreicht ohne Anpassung eine hohe Erkennungsrate bei unterschiedlichen Lichtquellen und Beleuchtungsstärken.

Es konnte hierdurch gezeigt werden, dass auch ohne statische Farbtabellen und damit aufwendiger manueller Kalibrierung im RoboCup eine zuverlässige, beleuchtungsunabhängige Bildverarbeitung möglich ist.

8.1 Ausblick

8.1.1 Evaluierung

Mit dieser Arbeit wurden hauptsächlich die Grundlagen geschaffen, um Evaluierungsalgorithmen zu schreiben. Einige Beispiele wurden implementiert, es sind aber noch verschiedenste andere Einsatzmöglichkeiten denkbar. Im nächsten Schritt könnte zum Beispiel der Perzepterkennungsevaluator so erweitert werden, dass nicht nur die Existenz von Perzepten überprüft wird, sondern auch die Perzepte selbst verglichen werden. Zwei Ballperzepte könnten nur dann als gleich gelten, wenn sie in etwa an derselben Position erkannt werden. Aus den Differenzen zwischen zwei Erkennern könnten Statistiken erstellt werden. Zur Bewertung verschiedener Modellierungen sollte eine Deckenkamera eingebunden werden, um die vom Roboter berechneten Werte verifizieren zu können.

Im Moment kann auf einem Logfile immer nur eine Storage arbeiten. Es wäre wünschenswert, die Implementierung so zu erweitern, dass verschiedene Storages gleichzeitig auf einem Logfile arbeiten können und so auch Evaluatoren verschiedener Storages das Logfile gleichzeitig auswerten können. Ähnliches gilt für die Evaluatoren. Im Moment kann ein Evaluator nur einmal instanziiert werden, auch wenn die jeweiligen Instanzen unterschiedlich konfiguriert werden könnten und so zu unterschiedlichen Ergebnissen kommen.

8.1.2 Bildverarbeitung

Die neu geschriebene Bildverarbeitung ist wie oben beschrieben momentan aus verschiedenen Gründen nicht dafür geeignet, in der derzeitigen Form auf den Robotern eingesetzt zu werden. Die Ergebnisse dieser Arbeit könnten jedoch nach und nach auch in die bestehende Bildverarbeitung eingebaut werden. Im ersten Schritt könnte überprüft werden, wie praktikabel es ist, die aktuelle Farbklassifizierung im YUV-Farbraum, durch eine Klassifizierung im HSV-Farbraum zu ersetzen. Ebenso könnten die aktuellen Algorithmen nach und nach so erweitert werden, dass sie die Kantenerkennung nutzen, um Schwächen der Farbklassifizierung auszugleichen.

Die neue Bildverarbeitung sollte optimiert und um die fehlenden Erkenner erweitert werden. Die Grundlagen von Tor- und Linienperzeptor sind bereits vorhanden. Die Neuentwicklung kann genutzt werden, um die aktuelle Bildverarbeitung zu bewerten. Sobald mittelfristig mehr Rechenzeit auf den Robotern zur Verfügung steht, ist es möglich, die aktuelle Bildverarbeitung zu ersetzen.

Anhang A

Umwandlung zwischen RGB- und HSV-Farbraum

Input : $R, G, B \in [0, 1]$
Output : $H \in [0°, 360°]; S, V \in [0, 1]$

$MIN = min(R, G, B)$;
$MAX = max(R, G, B)$;
$V = MAX$;
$S = (MAX! = 0)?(MAX - MIN)/MAX : 0$;
if $S = 0$ **then**
    $H = undefined$;
else
    $delta = MAX - MIN$;
    if $R == MAX$ **then**
        $H = (G - B)/delta$;
    else if $G == MAX$ **then**
        $H = 2. + (B - R)/delta$;
    else if $B == MAX$ **then**
        $H = 4. + (R - G)/delta$;
    if $H < 0.$ **then**
        $H+ = 360.$
    end
end

Algorithmus A.1 : Umwandlung von RGB nach HSV. Quelle: [49]

Input : $H \in [0°, 360°]$ *oder* $undefined$; $S, V \in [0, 1]$
Output : $R, G, B \in [0, 1]$

if $S = 0$ **then**
    if $H == undefined$ **then**
        R = G = B = V;
    else
        Fehler!
    end
else
    if $H == 360$ **then**
        $H = 0$;
    end
    $H/ = 60.$;
    $i = floor(H)$;
    $f = h - i$;
    $p = v(1. - s)$;
    $q = v(1 - (s \cdot f)$;
    $t = v(1.(s(1. - f)))$;
    switch i **do**
        case 0*:* R = v; G = t; B = p;
        case 1*:* R = q; G = v; B = p;
        case 2*:* R = p; G = v; B = t;
        case 3*:* R = p; G = q; B = v;
        case 4*:* R = t; G = p; B = v;
        case 5*:* R = v; G = p; B = q;
    end
end

Algorithmus A.2 : Umwandlung vom HSV nach RGB. Quelle: [49]

Anhang B

Verwendete Bibliotheken

B.1 Boost

Im Folgenden werden einige der verwendeten von Boost zur Verfügung gestellten Funktionen erklärt.

B.1.1 Foreach

BOOST_FOREACH erlaubt es sehr einfach über verschiedene Arten von Sequenzen zu iterieren. Unterstützt werden zum Beispiel STL-Container, Arrays oder null-terminierte Strings. Der große Vorteil beim Iterieren über Container ist, dass man die oft komplizierten Iteratoren nicht ausschreiben muss.

```
std::vector<std::vector<int> > matrix;
BOOST_FOREACH( const std::vector<int> & row, matrix ) {}

for (std::vector<std::vector<int> >::const_iterator it = matrix.begin();
  it != matrix.end(); ++it) {}
```

Listing B.1: Beispiel für die Nutzung von BOOST_FOREACH

B.1.2 Shared Pointers

Eines der großen Probleme von C++ sind Speicherlöcher, also angeforderter Speicher, der nie wieder freigegeben wird, obwohl er nicht mehr benutzt wird. Um dieses Problem zu umgehen, stellt Boost eine Template-Klasse namens `shared_ptr` zur Verfügung. Diese Klasse enthält einen Referenzzähler und sorgt dafür, dass ein gekapseltes Objekt zerstört wird, sobald es keinen `shared_ptr` mehr gibt, der ebenfalls auf dieses Objekt zeigt, der Referenzzähler also null ist. Ein `shared_ptr` bekommt hierfür ein mit `new` allokiertes Objekt übergeben und kann danach kopiert werden. Jede Kopie erhöht den Referenzzähler, jeder zerstörte `shared_ptr` erniedrigt den Referenzzähler wieder. Ein `shared_ptr` lässt sich, abgesehen von seinen zusätzlichen Funktionen, wie ein normaler Zeiger benutzen.

```
void main() {
  std::tr1::shared_ptr<std::list<int> > sharedList1(new std::list<int>);
  {
    std::tr1::shared_ptr<std::list<int> > sharedList2 = sharedList1;
    sharedList2->push_back(5);
  } // sharedList2 wird zerstört, das Listenobjekt jedoch nicht
  // sharedList1 zeigt auf eine Liste mit dem Element 5
  return; // sharedList1 wird freigegeben. Da kein shared_ptr die Liste
          // mehr referenziert wird auch diese freigegeben
}
```

Listing B.2: Beispiel für die Nutzung von `std::tr1::shared_ptr`

B.1.3 Signals

B.1.4 Unordered

Um Daten auf einen Schlüssel zu mappen stellt die Standardbibliothek `std::set`, `std::map`, `std::multiset` und `std::multimap` zur Verfügung. Diese Container basieren im Normalfall auf binären Bäumen und haben daher eine logarithmische Komplexität beim Zugriff auf Elemente. Ist die Reihenfolge der Objekte im Container irrelevant, bieten Boost und der TR1 auf Hashing basierende Funktionen mit normalerweise konstanter Zugriffszeit an: `std::tr1::unordered_set`, `std::tr1::unordered_map`, `std::tr1::unordered_multiset` und `std::tr1::unordered_multimap`. Die einzige Voraussetzung ist, dass für zu speichernde Objekte eine Hashfunktion und der Vergleichsoperator implementiert sind. Der Kleineroperator muss entgegen den bisher benutzen Containern nicht existieren.

B.1.5 Lexical Cast

```
std::string zahlenString = "5.67834";
try {
  // Umwandlung eines Strings in eine Zahl
  float zahl = boost::lexical\_cast<float>(zahlenString);
  // Umwandlung einer Zahl in einen String
  zahlenString = boost::lexical\_cast<std::string>(zahl);
} catch(boost::bad_lexical\_cast & e) {
  // wenn nicht sicher ist, ob die zu castenden Variablen wohlgeform sind,
  // muss die Exception boost::bad_lexical_cast gefangen werden
}
```

Listing B.3: Beispiel für die Nutzung von `boost::lexical_cast`

Oft ist es nötig, Strings in Zahlen oder Zahlen in Strings umzuwandeln. Sowohl C als auch C++ stellen hierfür verschiedene Möglichkeiten bereit, allerdings sind diese fehleranfällig oder nicht direkt und einfach nutzbar. Für die einfache Umwandlung von Strings in Zahlen und umgekehrt stellt Boost das Funktions-Template `lexical_cast` zur Verfügung.

Anhang C

PIMPL-Idiom

Beispiel einer dem PIMPL-Idiom folgenden Klasse.

```
#include <memory>
class FooPrivate;
class Foo {
public:
  /* ... */
private:
  friend class FooPrivate;
  const std::tr1::shared_ptr<FooPrivate> d;
};
```

Listing C.1: *Foo.h* mit dem Interface der Klasse `Foo`

```
#include "a-header.h"
#include "another-header.h"
class FooPrivate {
public:
  FooPrivate(Foo * const parent)
  int aMethod();
  Foo * const parent;
};

FooPrivate(Foo * const parent)
  : parent(parent)
{}

int FooPrivate::aMethod() {
  /* ... */
}

Foo::Foo() : d(new FooPrivate)
{}
```

Listing C.2: *Foo.cpp* mit der Implementierung der Klasse `Foo` und der privaten Klasse `FooPrivate`

Die Headerdatei enthält beim PIMPL-Idiom nur die Methoden, die für das Interface einer Klasse benötigt werden. Alle normalerweise privaten Methoden werden direkt in der Cpp-Datei in einer privaten Klasse implementiert. Ändert sich die Implementierung sind alle Klassen, die die Klasse nur über das Interface nutzen, nicht betroffen und müssen nicht neu kompiliert werden. Des Weiteren enthält die Header-Datei nur die für das Interface benötigten *Includes*, was die Abhängigkeiten der Klassen untereinander reduziert.

Über die (optionale) Membervariable `parent` kann die private Klasse auf die eigentliche Klasse zugreifen. Da sie ein *Friend* der Klasse ist, wird die Kapselung aufgehoben und sie kann zum Beispiel auf vererbte Methoden oder Variablen zugreifen.

Literaturverzeichnis

[1] M. Friedmann, K. Petersen, S. Petters, K. Radkhah, D. Thomas und O. von Stryk: *Darmstadt Dribblers: Team Description for Humanoid KidSize League of RoboCup 2008.* Technischer Bericht, Technische Universität Darmstadt, 2008.

[2] Fachgebiet SIM der Technischen Universität Darmstadt: *Darmstadt Dribblers.* Webseite: `http://www.dribblers.de`, Oktober 2009.

[3] Thomas Röfer, Ronnie Brunn, Ingo Dahm, Matthias Hebbel, Jan Homann, Matthias Jüngel, Tim Laue, Martin Lötzsch, Walter Nistico und Michael Spranger: *GermanTeam 2004 - The German National RoboCup Team.* In: *RoboCup 2004: Robot Soccer World Cup VIII, volume 3276 of Lecture Notes in Artificial Intelligence.* Springer, 2004.

[4] Simon Templer: *Fusion von externen Videos und intrinsischen Daten zur Offline-Analyse von Teams mobiler autonomer Roboter.* Diplomarbeit, TU Darmstadt, FB Informatik, 2009. Noch nicht veröffentlicht.

[5] Martin Friedmann, Karen Petersen und Oskar von Stryk: *Adequate motion simulation and collision detection for soccer playing humanoid robots.* Robot. Auton. Syst., 57(8):786–795, 2009.

[6] Paul R. Cohen: *Empirical Methods for Artificial Intelligence.* MIT Press, 1995.

[7] Peter Stone Mohan Sridharan: *Real-time vision on a mobile robot platform.* In: *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005.

[8] James Bruce, Tucker Balch und Manuela Veloso: *Fast and Inexpensive Color Image Segmentation for Interactive Robots.* In: *In Proceedings of IROS-2000*, Seiten 2061–2066, 2000.

[9] T. Röfer, J. Brose, E. Carls, J. Carstens, D. Göhring, M. Jüngel, T. Laue, T. Oberlies, S. Oesau, M. Risler, M. Spranger, C. Werner und J. Zimmer: *GermanTeam 2006: The German National RoboCup Team.* In: *RoboCup 2006: Robot Soccer World Cup X Lecture Notes in Artificial Intelligence.* Springer, 2007.

[10] Mohan Sridharan und Peter Stone: *Autonomous Planned Color Learning on a Legged Robot.* In: *RoboCup*, Seiten 270–278, 2006.

[11] Josef Baumgartner: *Automatische Farbklassifikation zur Anwendung im RoboCup.* Diplomarbeit, TU Darmstadt, FB Informatik, 2008.

[12] Raziel Alvarez, Erik Millán, Alejandro Aceves-López und Ricardo Swain-Oropeza: *Mobile Robots: Perception and Navigation*, Kapitel Accurate Color Classification and Segmentation for Mobile Robots. Pro Literatur Verlag, 2007.

[13] Christopher Stanton und Mary-Anne Williams: *A Novel and Practical Approach Towards Color Constancy for Mobile Robots Using Overlapping Color Space Signatures.* In: Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda und Yasutake Takahashi (Herausgeber): *RoboCup 2005: Robot Soccer World Cup IX*, Band 4020 der Reihe *Lecture Notes in Computer Science*, Seiten 444–451. Springer, 2005.

[14] Mohan Sridharan und Peter Stone: *Towards Illumination Invariance in the Legged League.* In: *RobuCup*, Seiten 196–208. Springer, 2004.

[15] Toshifumi Kikuchi, Kazunori Umeda, Ryuichi Ueda, Yoshiaki Jitsukawa, Hisashi Osumi und Tamio Arai: *Improvement of Color Recognition Using Colored Objects.* In: *RoboCup 2005: Robot Soccer World Cup IX*, Seiten 537–544, 2005.

[16] Craig L. Murch und Stephan K. Chalup: *Combining edge detection and colour segmentation in the Four-Legged League.* In: *Australasian Conference on Robotics and Automation 2004*, 2004.

[17] Michael Kass, Andrew Witkin und Demetri Terzopoulos: *Snakes: Active contour models.* International Journal of Of Computer Vision, 1(4):321–331, 1988.

[18] David Lowe: *Object Recognition from Local Scale-Invariant Features.* Seiten 1150–1157, 1999.

[19] Alireza Tavakoli Targhi, Eric Hayman, Jan-Olof Eklundh und Mehrdad Shahshahani: *The Eigen-Transform and Applications.* In: *Computer Vision – ACCV 2006, 7th Asian Conference on Computer Vision*, Seiten 70–79, 2006.

[20] Bernt Schiele und James L. Crowley: *Recognition without correspondence using multidimensional receptive field histograms.* International Journal of Computer Vision, 36:31–50, 2000.

[21] Stephen Turner: *Hermann Günther Graßmann (1809-1877): Visionary Mathematician, Scientist and Neohumanist Scholar*, Band 187 der Reihe *Boston Studies in the Philosophy of Science*, Kapitel The origins of colorimetry: What did Helmholtz and Maxwell learn from Grassmann?, Seiten 78–79. Springer, 1996.

[22] Dave Wilson: *FourCC (Four Character Code) YUV Formats.* Webseite: `http://www.fourcc.org/yuv.php`, Oktober 2009.

[23] Alvy Ray Smith: *Color gamut transform pairs.* In: *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, Seiten 12–19, New York, NY, USA, 1978. ACM.

[24] S. Vitabile, G. Pollaccia, G. Pilato und E. Sorbello: *Road signs recognition using a dynamic pixel aggregation technique in the HSV color space.* Seiten 572–577, 2001.

[25] S. Sural, Gang Qian und S. Pramanik: *Segmentation and histogram generation using the HSV color space for image retrieval.* Band 2, Seiten II–589–II–592 vol.2, 2002.

[26] Jack Bresenham: *Algorithm for Computer Control of a Digital Plotter.* IBM Systems Journal, 4(1):25–30, 1965.

[27] John Canny: *A Computational Approach to Edge Detection.* IEEE Trans. Pattern Anal. Mach. Intell., (6):679–698, November 1986.

[28] Sebastian Petters und Dirk Thomas: *RoboFrame – Softwareframework für mobile autonome Robotersysteme.* Diplomarbeit, TU Darmstadt, FB Informatik, 2005.

[29] S. Petters, D. Thomas und O. von Stryk: *RoboFrame – A Modular Software Framework for Lightweight Autonomous Robots.* In: *Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, San Diego, CA, USA, Oct. 29 2007.

[30] Sebastian Petters und Dirk Thomas: *RoboFrame.* Webseite: `http://www.roboframe.info`, Oktober 2009.

[31] Hajime Sakamoto: *Hajime Sakamoto Institute Ltd.* Webseite: `http://www.hajimerobot.co.jp`, Oktober 2009.

[32] Micha Andriluka, Martin Friedmann, Stefan Kohlbrecher, Johannes Meyer, Karen Petersen, Christian Reinl, Peter Schauß, Paul Schnitzspan, Armin Strobel, Dirk Thomas und Oskar von Stryk: *RoboCupRescue 2009 – Robot League Team: Darmstadt Rescue Robot Team (Germany).* Technischer Bericht, Technische Universität Darmstadt, 2009.

[33] *Darmstadt Rescue Robot Team.* Webseite: `http://www.gkmm.tu-darmstadt.de/rescue/`, Oktober 2009.

[34] RoboCup Federation: *RoboCup Federation Call for Tenders: A Standard Robot Platform for Robot Soccer*, 2006.

[35] M. Friedmann, S. Petters, M. Risler, H. Sakamoto, D. Thomas und O. von Stryk: *A new, open and modular platform for research in autonomous four-legged robots.* In: K. Berns und T. Luksch (Herausgeber): *Autonome Mobile Systeme 2007*, Informatik aktuell, Seiten 254 – 260, Kaiserslautern, 18 - 19 Oct. 2007. Springer Verlag.

[36] Richard O. Duda und Peter E. Hart: *Use of the Hough transformation to detect lines and curves in pictures.* Commun. ACM, 15(1):11–15, January 1972.

[37] *Boost C++ Libraries.* Webseite: `http://www.boost.org`, Oktober 2009.

[38] Matt Austern: *Draft Technical Report on C++ Library Extensions.* Technischer Bericht N1836 05-0096, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005.

[39] Robert C. Martin: *More C++ gems.* Cambridge University Press, Seiten 407–416, 2000.

[40] Glenn E. Krasner und Stephen T. Pope: *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80.* Journal of Object Oriented Programming, 1(3):26–49, 1988.

[41] Nokia Corporation: *Model/View Programming.* Webseite: `http://doc.trolltech.com/4.4/model-view-programming.html`, Oktober 2009.

[42] Nokia Corporation: *QThreadPool Class Reference.* Webseite: `http://doc.trolltech.com/4.4/qthreadpool.html`, Oktober 2009.

[43] Nokia Corporation: *Rich Text Processing: Supported HTML Subset.* Webseite: `http://doc.trolltech.com/4.5/richtext-html-subset.html`, Oktober 2009.

[44] Uwe Rathmann: *Qwt – Qt Widgets for Technical Applications.* Webseite: `http://qwt.sourceforge.net/`, Oktober 2009.

[45] Willow Garage: *OpenCV.* Webseite: `http://opencv.willowgarage.com/`, Oktober 2009.

[46] Gary Bradski und Adrian Kaehler: *Learning OpenCV: Computer Vision with the OpenCV Library.* O'Reilly, Cambridge, MA, 2008.

[47] Willow Garage: *OpenCV Reference – Image Processing.* Webseite: `http://opencv.willowgarage.com/documentation/image_processing.html`, Oktober 2009.

[48] Willow Garage: *OpenCV Reference – HighGui.* Webseite: `http://opencv.willowgarage.com/documentation/highgui.html`, Oktober 2009.

[49] *Computer Graphics: Principles and Practice in C.* Addison-Wesley Professional, 2. Auflage, 1990.