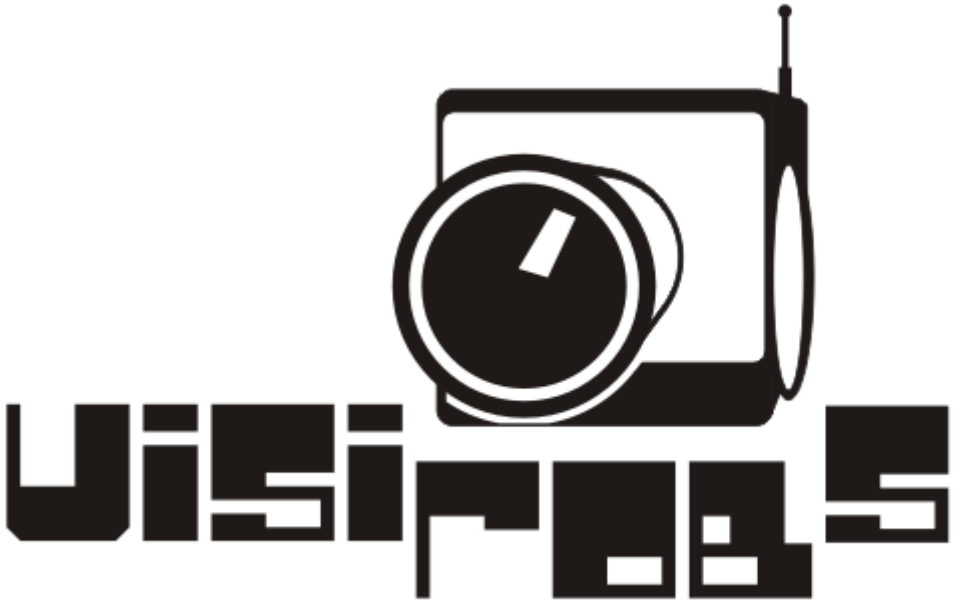


Endbericht der Projektgruppe 472

PG 472

„VisiRobs“

Mittwoch, 1. März 2006

Inhaltsverzeichnis

I. Einleitung

I.1 Die Projektgruppe 472

Gegenstand dieses Endberichts ist die Arbeit der Projektgruppe (PG) 472 während der Semester SS05 und WS05/06. Die Teilnehmer der PG befassten sich mit dem Thema “Roboterfußball“ am Lehrstuhl 1 der Universität Dortmund.

Die Aufgabe der PG 472 war es, die von der Vorgänger-PG entworfenen Roboter mit lokaler Kamera ins vorhandene System zu integrieren und dabei eine stark ressourcenbeschränkte Bildverarbeitung zu implementieren. Ebenfalls sollte untersucht werden, wie sich Kompass und Beschleunigungssensor der Roboter im Spiel nutzen lassen. Zu den Aufgaben gehörte aber auch die Pflege und Erweiterung des bestehenden Systems sowie die Unterstützung der Lehrstuhl-Mitarbeiter auf diversen Veranstaltungen.

Die PG 472 nahm ihre Arbeit im Februar 2005 auf und startete mit Seminarvorträgen, die alle Teilnehmer vorbereiten mussten. Dieser sollte die Teilnehmer auf die Aufgaben vorbereiten und wurde im Rahmen einer Seminarfahrt vorgetragen. Zusätzlich diente die Seminarfahrt dazu, dass sich die Teilnehmer gegenseitig und die Betreuer näher kennen lernen konnten. Die Ergebnisse der Seminarfahrt sind im Seminarband der PG 472 dokumentiert.

Aufgrund vieler Probleme im Hardwaredesign und in der Fertigung der neuen Roboter wurde ein weiterer Schwerpunkt auf die Verbesserung des alten Systems gelegt. So wurde eine dringend nötige Kameraentzerrung implementiert, sowie Strategie und Anfahrt auf die Bedürfnisse der neuen Roboter angepasst.

I.1.1 Teilnehmer

Die PG 472 bestand aus den folgenden Teilnehmern:

- Tom Bomhof
- Altay Cebe
- Christoph Ewerlin
- Tobias Jäger
- Christian Kintzel
- Sarah König
- Christian Kolek
- Daniel Mikus
- Andreas Nettsträter

- Rainer Oye
- Kristijan Pulina
- Christian Tesch

I.1.2 Betreuung

Die Betreuer der PG 472 waren:

- Prof. Dr. Bernd Reusch
- Dr. Lars Hildebrand
- Norman Weiss

II. Zwischenbericht

PG 472

II.1 Einführungsphase

Die PG begann mit einer Einführungsphase, in der sich zunächst alle Mitglieder mit Hilfe der Hiwis einen ersten und groben Überblick über das bestehende System und den Stand der Entwicklung bezüglich, des neuen Roboters verschaffen sollten. Da mittlerweile sieben Projektgruppen ihre Arbeit in das System investiert haben, kam eine größere Einarbeitungsphase auf die PG zu.
Daher wurde es als notwendig angesehen, zunächst Kleingruppen zu bilden, die sich mit der Einarbeitung in die verschiedenen Aspekte des Systems befassen. Diese Gruppen sollten sich nach wenigen Wochen wieder auflösen und neue Kleingruppen bilden.

II.2 Erste Kleingruppenphase

So wurden in der Sitzung am 18.04.2005 zunächst die im Folgenden aufgeführten Kleingruppen gebildet. Diese Einteilung war nicht für die gesamte Dauer der Projektgruppe gedacht, sondern sollte bei Bedarf angepasst werden. Es war für einige Gruppen ohnehin klar, dass sie nur zu Beginn der Projektgruppe sinnvoll sind und sie ihre Arbeit nach einigen Wochen beendet haben.

II.2.1 Gruppen und Ziele

Die Gruppen werden im Folgenden aufgelistet, ebenso die Mitglieder und die Ziele, die sie sich gesetzt haben:

Gruppe „Bestandsaufnahme“
Teilnehmer:
Sarah Koenig
Andreas Nettsträter
Rainer Oye
Ziele:

- Einarbeitung in die vorhandene Soft- und Hardware
- Ausarbeitung einer Dokumentation des bestehenden Systems
- Checkliste erstellen
- automatische Kurvengeschwindigkeitskalibrierung

Gruppe „GUI“

Teilnehmer:

Christian Kolek

Christian Tesch

Daniel Mikus

Rainer Oye

Altay Cebe

Ziele:

- 3D-Ansicht der GUI so ändern, dass sich die Zielpositionen der Roboter durch "klicken und ziehen" verschieben lassen
- Abhandlung über die interne Funktionsweise der GUI

Gruppe „lokale Kamera“

Teilnehmer:

Tobias Jäger

Christian Kolek

Tom Bomhof

Christoph Ewerlin

Ziele:

- Framework zum Testen der BV-Algorithmen schreiben
- Kameramodul am lokalen PC anschließen
- Einarbeitung in Fuzzy – Farbmodell
- Kamerakalibrierung

Gruppe „streamorientierte BV“

Teilnehmer:

Kristijan Pulina

Ziele:

- Literaturrecherche
- Zusammen mit der Gruppe „lokale Kamera“ erste Algorithmen testen

Gruppe „Deckenkamera / AIBO“

Teilnehmer:

Altay Cebe

Christian Kintzel

Ziele:

- Einarbeitung in die Funktionsweise der Software für die Deckenkamera
- Software der AIBOs ansehen und prüfen, ob sich der Ansatz für die Entzerrung des Kamerabildes auf unsere Bildverarbeitung übertragen lässt.
- Eventuell einen Vortrag eines Teilnehmers der AIBO - PG organisieren

Es wurde auch an größere und weiterführende Ziele gedacht, die zu diesem Zeitpunkt jedoch nicht abzuschätzen waren und daher zurückgestellt wurden. Dabei ging es auch um die Mechanik und die Konstruktion der neuen Roboter. Ein anderer dieser Punkte war, mit Hilfe der neuen Sensoren und Kameras ein Weltbild auf dem Roboter zu implementieren. Dazu sollten die geplanten Komponenten getestet werden, um festzustellen, wie brauchbar sie dafür sind. Zu diesem Zeitpunkt stand allerdings noch keine Hardware zur Verfügung und es war auch noch nicht klar, wie der Roboter in letzter Instanz aussehen und konstruiert sein sollte. Daher wurde auch dieses Ziel wegen zu vieler unbekannter Parameter zunächst zurückgestellt. Des Weiteren wurde über Benchmarks nachgedacht, die die Funktionsfähigkeit und Leistungsfähigkeit der Produkte der PG unter Beweis stellen.
Am Ende dieser ersten Kleingruppen-Phase fand eine Präsentation der gewonnen Erkenntnisse jeder einzelnen Gruppe statt.

II.2.2 Ergebnisse nach der ersten Kleingruppen-Phase

II.2.2.1 Bestandsaufnahme

Um den Einstieg in das bestehende System zu erleichtern, wurde eine Gruppe gebildet, die sich in den Aufbau und Betrieb der Roboter einarbeiten sollte. Ziel dieser Gruppe war es, eine Anleitung für die Nutzung der Hard- und Software sowie eine Checkliste für den Aufbau zu verfassen.
Nach einer Einweisung durch die Hiwis wurde klar, dass die Anleitung bebildert sein sollte, um das Finden und Zusammenfügen der einzelnen Komponenten des Systems zu erleichtern. Außerdem mussten die speziellen Begriffe wie Blobs und Patches erläutert werden. Des Weiteren wurden in der Anleitung die Vorbereitung für den Spielbetrieb und alle wichtigen Systemkomponenten des laufenden Systems beschrieben, sowie die Funktionsweise erklärt.
Die Vorbereitung für den Spielbetrieb beinhaltet gegebenenfalls den Aufbau des Spielfeldes und das Montieren und Anschließen der Kameras und Funksender. Außerdem muss für ausreichende Beleuchtung gesorgt werden. Anschließend müssen die Roboter mit Akkus und den entsprechenden Funkmodulen ausgestattet werden. Die Anleitung enthält auch ein ausführliches Kapitel, das beschreibt wie die Anwendung auf dem Host-System zu starten, einzurichten und das Spiel zu steuern ist. Hierbei werden verschiedene Einstellungen vorgenommen, damit die Patches der Roboter von der

Bildverarbeitung gut erkannt werden. Außerdem können in der Software diverse Einstellungen vorgenommen werden, die die Strategie betreffen.
Ein Ergebnis der Einarbeitung war, dass selbst bei genauem Befolgen aller Anleitungen viel Übung dafür nötig ist, um die Roboter sauber laufen zu lassen.
Neben der ausführlichen, etwa 30-seitigen Anleitung[1] wurde auch eine kurze Checkliste[2] verfasst, die beim Aufbau des Systems bei Wettkämpfen oder Veranstaltungen helfen soll.
Im nächsten Schritt befasste sich die Gruppe mit den Robotern selbst und ihrem Code und erstellte eine Anleitung[3] zum Flashen und Laden von Software auf die alten Roboter. Es wurden kleine einfache Programmbeispiele[4] aus der vorhandenen Software erstellt. Diese Beispiele dienten dazu, den Umgang mit den Robotern, den Geräten zum Laden von Software auf die Roboter und der Entwicklungsumgebung zu lernen. Diese Dokumente sollen späteren Gruppen eine Hilfestellung zur Einarbeitung in das System bieten.

II.2.2.2 GUI

In dieser Gruppe haben wir uns zum Ziel gesetzt, uns in die GUI der BV einzuarbeiten. Als Endziel dieser Einarbeitung sollten wir die Möglichkeit einbauen, die Zielposition der Roboter über Drag&Drop zu verändern. Dies sollte über den OpenGL Tab der Bildverarbeitung geschehen. Nach einiger Einarbeitung haben wir schließlich das Ziel erreicht. Nun ist es in der 3D-Ansicht möglich die Roboter durch Anklicken und Festhalten der linken Maustaste in ihrer Position auf der Spielfläche neu zu positionieren. Nach einem Wunsch der Hiwis haben wir dann auch die Möglichkeit eingebaut, die Software-ID's der Roboter über Drag&Drop zu tauschen. Im Gegensatz zu anderen Teams werden bei den Dortmund Droids die Roboter nicht über individuelle Rückenpatches identifiziert, sondern jedem Roboter wird am Anfang eine Software-ID zugewiesen. Ein Tracking- Algorithmus sorgt dafür, dass diese ID's beibehalten werden. Zur Zeit kann es jedoch passieren, dass die Software-ID's von zwei Robotern, die sich nah nebeneinander bewegen, aus Versehen vertauscht werden. Daher ist es notwendig, dass schnelle manuelle Vertauschen der ID's zu ermöglichen. Diese Feature wurde so realisiert, dass man zusätzlich beim Anklicken die STRG-Taste halten muss.

[1] https://www.robosoccer.de/intern/files/Anleitungen/Bestandsaufnahme_PG472/Anleitung_050505.pdf

[2] https://www.robosoccer.de/intern/files/Anleitungen/Bestandsaufnahme_PG472/Checkliste.zip

[3] https://www.robosoccer.de/intern/files/Anleitungen/Bestandsaufnahme_PG472/flash_jtag_alte_roboter.pdf

[4] https://www.robosoccer.de/intern/files/Code_Beispiele/PG472/led.zip

II.2.2.3 Deckenkamera

Im Gegensatz zu den anderen Gruppen, hat die Gruppe Deckenkamera/ Entzerrung/ Bildverarbeitung sich während das ganzen Semesters mit einem großen Ziel beschäftigt. Außerdem blieb die Gruppe in ihrer Besetzung über die ganze Zeit weitestgehend gleich. Daher ist der Bericht dieser Gruppe nicht in drei Phasen unterteilt, sondern findet sich komplett unter II.4.2.1.

II.2.2.4 Lokale Kamera

Die Bildverarbeitung spielt für die Ballerkennung eine zentrale Rolle. Sie lässt sich aber aufgrund mangelnder Ausgabemöglichkeiten schlecht direkt auf dem Roboter implementieren und testen, daher wurde die Gruppe „Lokale Kamera“ mit der Entwicklung einer Testumgebung für den PC beauftragt.
Diese Software sollte die Eigenheiten der später auf dem fertigen Roboter erwarteten Hardware nachbilden und es der Projektgruppe ermöglichen, auf einfache Weise verschiedene Ansätze der Bildverarbeitung vorab zu testen. Um dieses Ziel zu verwirklichen wurde ein modularer Aufbau konzipiert, bei dem Algorithmen in der Form von Filtern auf vorhandenen Bilddaten aufsetzen und eine wie auch immer geartete Ausgabe produzieren.
Der Aufbau der Testumgebung wurde so gewählt, dass auf der untersten Ebene die Bildquelle steht. Als Bilddaten soll entweder ein gespeichertes Bild beziehungsweise eine gespeicherte Bildfolge oder direkt ein Datenstrom aus einer Kameraplatine verwendet werden. Eine entsprechende Platine wurde in der Projektgruppe 449 entwickelt. Die Nutzung eines Datenstroms aus der Kameraplatine war jedoch aufgrund der beschränkten Bandbreite der verwendeten seriellen Schnittstelle nicht effektiv möglich, weswegen erst eine Zwischenspeicherung notwendig wurde. Eine weitere Eigenschaft der auf der Platine realisierten Schaltung ist, dass die Daten, welche im Format YUV444 vorliegen, nicht zeilen- sondern spaltenweise ausgegeben werden. Hierdurch soll das Finden des Balles beschleunigt werden, da die entsprechenden Pixel so schneller gefunden werden.
Die Daten aus der jeweiligen Bildquelle werden durch einen virtuellen Framebuffer geleitet, welcher die Funktionalität des auf dem Roboter eingesetzten Averlogic Framebuffers dupliziert. Insbesondere ist hierbei zu nennen, dass die Struktur dieses speziellen Pufferspeichers nur ein sequentielles Auslesen der Daten vorsieht. Daher müssen jegliche Random-Access Zugriffe, wie sie unter Umständen später bei der Verwendung eines auf dem Roboter befindlichen SRAMs möglich wären, erst durch einen auf den Puffer aufsetzenden Filter implementiert werden.
Diesen beiden Stufen – Quelle und virtueller Framebuffer – folgen in der Testumgebung die einzelnen Filter. Sie realisieren beispielsweise eine Umwandlung der Daten in eine RGB-Repräsentation oder auch eine Konturerkennung und können ihre Ergebnisse in ein selbst zu gestaltendes QT-Widget ausgeben. Die Koordinierung der Filter wird von einer Steuerklasse, dem Filtercontroller, vorgenommen. Somit wird ermöglicht, mehrere unterschiedliche Filter parallel auf denselben

Bilddaten laufen zu lassen um Vergleiche oder auch, falls erwünscht, Verkettungen durchzuführen. Hierzu müssen Filter im Kontroller registriert werden, der dann deren Bearbeitungs-Routine aufruft, sobald genügend verarbeitbare Daten im Framebuffer zur Verfügung stehen.
Die Steuerung der Testumgebung wird in einer GUI vorgenommen. Sie übernimmt die Aufgabe, installierte Filter, die der Benutzer anwählt, im Filterkontroller zu registrieren (1) und mit einer Datenquelle wie dem virtuellen Framebuffer oder einer abgeleiteten Klasse zu verbinden (2). Diese kann noch, falls nötig, ein Konfigurations-Widget bereitstellen (3). Ebenso ist sie für die Aktivierung und Deaktivierung der angezeigten Filter zuständig (4) und bietet den jeweils aktiven Filtern eine freie Fläche für die Darstellung des Ausgabe-Widgets (5).
Die Software ist unter dem Namen „bvalgotester“ in das CVS System der PG eingestellt und kann durch beliebige, von den Klassen „filter“ und „filtergui“ abgeleiteten Filtern um neue Funktionalität erweitert werden.

II.2.2.5 Streamorientierte BV

Ziel der Gruppe streamorientierte Bildverarbeitung war es, nach Bildverarbeitungs-Algorithmen zu suchen, die ausschließlich auf Video-/Bilddaten-Datenenströmen in Echtzeit arbeiten und diese „live“ analysieren und schnell Ergebnisse liefern. Dadurch sollte unsere Bildverarbeitung der lokalen Kamera so beschleunigt werden, dass die Ballfindung und Bandenerkennung in Echtzeit möglich wird.
Zunächst war die Gruppe mit der Literaturrecherche beauftragt. Diese gestaltete sich sehr schwierig: Fast alle Bücher über Bildverarbeitung enthielten Algorithmen, die auf gespeicherten Bildern (z.B. im Framebuffer) arbeiteten. Die Bücher über Streaming handelten hauptsächlich vom Erstellen von Video-Streams, aber nicht von deren Analyse. Theorie-Bücher über Datenenstromalgorithmen waren ebenfalls nicht verwendbar. Es gab keine brauchbare Literatur zu diesem Spezialthema, und die Suche blieb vorerst ergebnislos. Zusammen mit der Gruppe „lokale Kamera“ sollten Algorithmen getestet werden. Dazu wurde im BV-Algotester-Programm ein Framebuffer implementiert, der mehrere Bilder der lokalen Kamera live speichert und immer wieder überschreibt. Die aktuelle Schreibposition wird jeweils den Filtern übergeben, die meist auf dem letzten komplettierten Bild arbeiteten. Dadurch geht zwar Zeit verloren, aber es ist die sicherste Methode um nicht die Filter warten zu lassen, die meist schneller sind als die Bilderfassung. Nach einiger Zeit wurde das Streaming-Problem als schwierig erkannt. Aus verschiedenen Fachgesprächen ergab sich weiterhin, dass die entsprechenden Algorithmen am ehesten im Bereich der Spracherkennung zu finden sind. Das erschien der Gruppe lokale Kamera als Sackgasse, weil der Aufwand für die Einarbeitung in Spracherkennung wohl viel größer gewesen wäre als der Nutzen für unsere lokale BV. Wegen der Ergebnislosigkeit wurde ab Mai der Schwerpunkt auf die Erarbeitung des oben beschriebenen Filterframeworks gelegt, das auf eine

live-Stream-Analyse im wesentlichen verzichtet. Schließlich wurde die Gruppe Streaming-BV aufgelöst und mit der Gruppe lokale Kamera verschmolzen. Als Ergebnisse bleiben festzuhalten:

- Das System arbeitet stabiler auf gespeicherten Bildern als auf dem Stream.
- Keine brauchbare Literatur zu dem Thema „streamorientierte Bildverarbeitung“ war auffindbar.
- BV-Algorithmen arbeiten besser auf Bildern aus dem Framebuffer.

II.3 Zweite Kleingruppenphase

Nach Abschluss der ersten einführenden Arbeiten und damit der Beendigung der Arbeit einiger Gruppen wurden am 12.05.2005 neue Ziele gesetzt, das Erreichen von möglichen Zielen besprochen und neue Gruppen gebildet.

II.3.1 Gruppen und Ziele

Es wurden folgende Gruppen bestimmt:

- Deckenkamera Entzerrung
- Framework für lokale Kamera / Bildverarbeitung, Weltmodell, GUI
- Hardware der neuen Roboter und Sensorik

Gruppe „Deckenkamera Entzerrung“

Teilnehmer:

Altay Cebe

Christian Kintzel

Ziele:

Aufgabe war, den Algorithmus in das Robotsoccer Host-System einzubinden, was sich aufgrund der gegebenen Programmierung als recht komplizierte Aufgabe herausstellte.

Gruppe: „Framework“

Teilnehmer:

Christian Tesch

Andreas Nettsträter

Daniel Mikus

Tom Bomhof

Christian Kolek

Kristjian Pulina

Ziele:

Es sollte ein Framework erstellt werden, welches das Programmieren und Entwickeln auf und für den DSP erheblich erleichtern soll. Ein weiteres Ziel ist die Entwicklung einer lokalen Testumgebung, die es ermöglicht Filter auf den Desktop-PC's zu entwickeln und zu testen, bevor diese tatsächlich auf den DSP gespielt werden. Einige Aufgaben zu diesem Ziel waren:

- Die Erstellung eines Weltmodells für die Roboter.
- Eine neue Art der Farbverwaltung, zum Beispiel über YUV, da dieses Farbformat von der Kamera zur Verfügung gestellt wird. Auch die Effizienz von FuzzyLogic zur Farberkennung soll hier diskutiert und untersucht werden.
- Das Erstellen und Sammeln von Bildern, die mit der lokalen Kamera aufgenommen wurden, damit diese für eine spätere Auswertung zur Verfügung stehen.

Gruppe: „Hardware / Sensorik"

Teilnehmer:

Tobias Jäger

Sarah Koenig

Rainer Oye

Christoph Ewerlin

Ziele:

Ziel sollte sein, eine funktionierende Software für verschiedene Benchmarks zu erstellen. Dadurch kann die Funktionsweise der Komponenten implementiert und dokumentiert werden und eventuelle Problematiken, die sich durch die Benutzung ergeben, können erkannt und festgehalten werden.

- Erstellen einer Bibliothek für den Kompass -Sensor
- Ansteuerung der Sensoren (Kompass-Sensor) über den I²C-Treiber
- Hardwaredesign und Konstruktion des neuen Roboters
- Erstellen einer SPI-Library für die Ansteuerung des Beschleunigungssensors

II.3.2 Ergebnisse der zweiten Kleingruppenphase

II.3.2.1 Deckenkamera und Entzerrung

Der zusammengefasste Bericht dieser Gruppe findet sich unter Punkt II.4.2.1

II.3.2.2 Gruppe Framework

Die Gruppe Framework arbeitete unterteilt in drei Untergruppen: Weltmodell, GUI und Lokale Kamera

Ziel der Gruppe **Framework-Weltmodell** war es ein geometrisches Weltmodell des Spielfeldes und aller spielrelevanter Objekte zu erstellen. Dieses sollte es dem Roboter ermöglichen sich auf dem Spielfeld zu orientieren und stellt somit einen ersten Schritt zur Autonomie dar. In der KW 21 haben wir das geometrische Weltmodell durch eine Datenstruktur in Standard-C realisiert. Dabei kam die folgende Struktur zum Einsatz:

```
typedef struct
{
     s_spielfeld* Spielfeld;
     s_bande** Banden;
     s_linie** Linien;
     s_punkt** Punkte;
     s_bereich** Bereiche;
     s_marker** Marker;
     s_eigeneroboter** EigeneRoboter;
     s_gegnerroboter** GegnerRoboter;
     s_kreis** Kreise;
     s_ball* Ball;

     int AnzahlBanden;
     int AnzahlLinien;
     int AnzahlPunkte;
     int AnzahlBereiche;
     int AnzahlMarker;
     int AnzahlEigeneRoboter;
     int AnzahlGegnerRoboter;
     int AnzahlKreise;
} s_weltmodell;
```

Spielfeld, Banden, Linien, Kreis, Ball, EigeneRoboter, GegnerRoboter repräsentieren die realen Objekte, die auf dem Spielfeld zu finden sind. In den Objekten sind die genauen Positionen, die Größe und die Farbe gespeichert. Mit den Bereichen und Punkten können wichtige Fixpunkte definiert werden, wie zum Beispiel der Freistoßpunkt, der Torraum, die eigene Hälfte und so weiter. Die Marker wurden zu Testzwecken implementiert, sodass mit diesen die Marker für die Selbstlokalisation dargestellt werden können.

In der folgenden KW22 hat die Gruppe das Modell mit einer einfachen OpenGL-Implemetierung getestet und geprüft. (Implementierung: GUI[5]) Ein Screenshot der Ansicht liegt diesem Bericht bei.

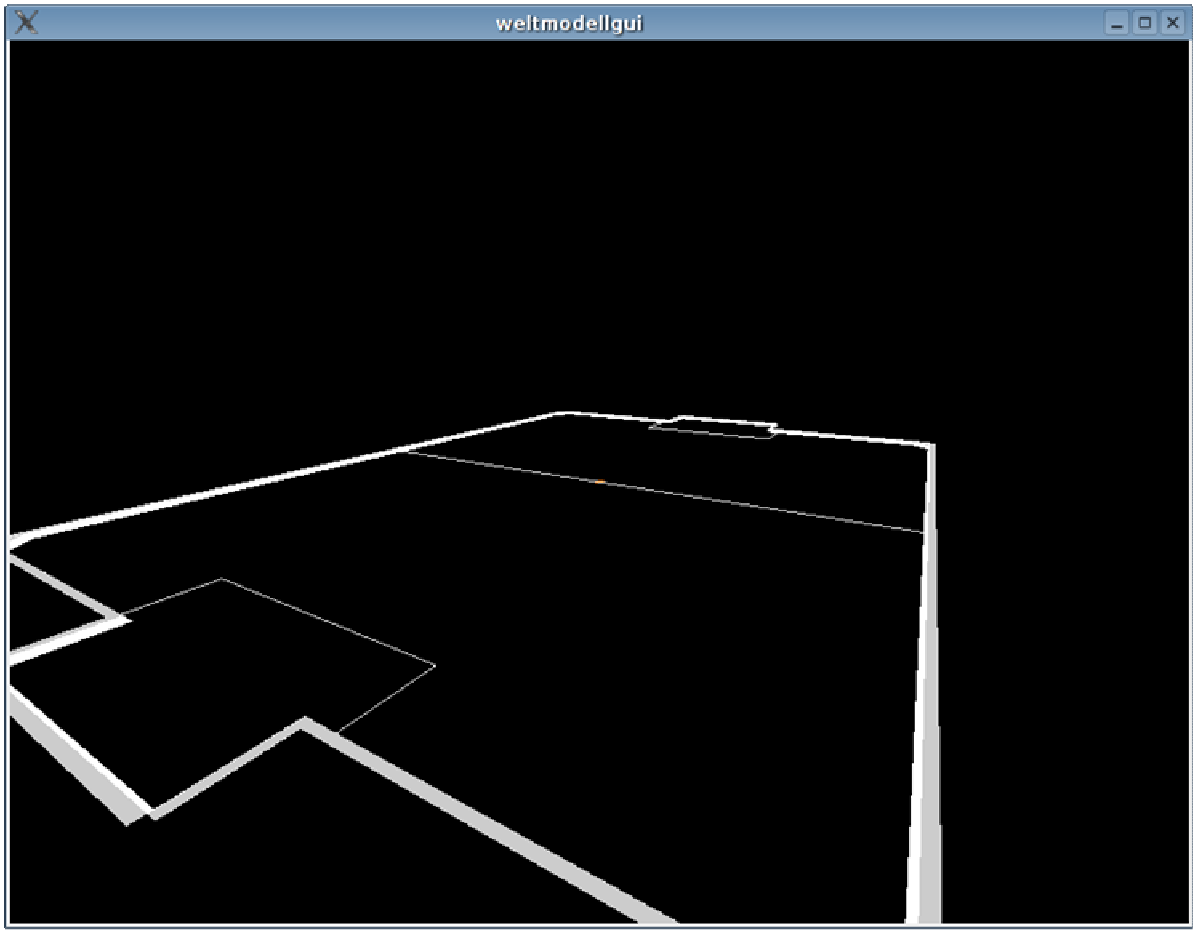


Abbildung II.3-1: OpenGL-Ansicht des Weltmodells

In der KW24 haben wir uns mit den theoretischen Grundlagen der Robotersicht und generell mit der 3D Berechnung beschäftigt. Ab der KW25 hat sich das Ziel der PG geändert und der Ansatz der lokalen Kamera wurde in den Hintergrund gestellt.

Für die Erstellung eines Frameworks zur Erprobung einer lokalen Kamera auf den Robotern wird eine graphische Benutzeroberfläche benötigt, die zur Eingabe spezifischer Umgebungsvariablen und zur Darstellung von Messwerten und Ergebnissen der lokalen Kamera dient. Diese GUI zu erstellen, war die Aufgabe der Untergruppe **Framework-GUI**.

[5] https://www.robosoccer.de/intern/files/Code_Beispiele/PG472/weltmodellgui.zip

Abbildung II.3-2: GUI lokale Kamera

Die hierbei entstandene GUI lässt sich folgendermaßen unterteilen:

lk-framework

- fps: Gibt die Bilder pro Sekunde der lokalen Kamera an.
- Kompass: Zeigt die übermittelten Werte des Kompasssensors an.
- vlinks: Radgeschwindigkeit links
- vrechts: Radgeschwindigkeit rechts
- $m/s^{2:}$ Aktuelle Beschleunigung

Variablen

- Spielfeldgröße

Zur Orientierung des Roboters auf dem Spielfeld ist die genaue Größe des Spielfeldes wichtig.

- Bandenhöhe

Zur Entfernungsberechnung wird die Bandenhöhe benötigt.

- Ballfarbe

Zur genauen Unterscheidung der Spielfeldobjekte muss die Ballfarbe eingestellt werden.

- Bandenfarbe

Zur Erkennung der Banden ist die genaue Angabe der Bandenfarbe wichtig.

- Torfarbe

Hier kann eine eventuelle Torfarbe eingestellt werden.

Filter

Dieser Teil wird aus dem vorhandenen Programm zur Kamerabildauswertung übernommen.

Strategie

- Ball anfahren

In dieser Funktion soll sich der Roboter so lange drehen, bis er den Ball mittig im Sichtfeld hat und auf den Ball zufahren. Eventuell soll die Richtung korrigiert werden, falls der Ball nicht mehr mittig zu sehen ist.

- Vor Bande stoppen

Diese Funktion soll testen, wie gut der Roboter während der Fahrt die Entfernung zur Bande erkennen kann und bei zu nahem Bandenabstand die Geschwindigkeit reduzieren, um eine Kollision mit der Bande zu vermeiden.

OpenGL

- Errechnete Robotersicht

Hier wird die Auswertung der Lokalisierung des Roboters dargestellt. Dabei wird das Spielfeld aus Sicht des Roboters gezeigt.
- Ballgröße
Gibt die errechnete Ballgröße an.
- Entfernung zum Ball
Gibt die errechnete Ballentfernung an.
- Mittlere Bandenhöhe
Gibt die Höhe der Bande an, die in der Mitte des Bildes zu sehen ist.
- Entfernung zur mittleren Bande
Dies ist die resultierende Entfernung zur Bande.
- Winkel zum Ball
Gibt den errechneten Winkel zum Ball an, falls dieser im Blickfeld der Kamera liegt.

Die Untergruppe **Framework-Lokale Kamera** hat sich unter anderem mit der Selbstlokalisation der Roboter beschäftigt. Anhand von Testbildern, die mit einer Digital-Foto-Kamera gemacht wurden, konnte gezeigt werden, dass der Roboter sich bis auf mindestens 20cm Genauigkeit lokalisieren könnte. Es wurden 10 Patches an den Rändern des Spielfeldes aufgestellt und an einer festen Position mit jeweils 45° Drehung aufgenommen. So entstanden 8 Bilder auf denen je nach Blickwinkel unterschiedlich viele Patches sichtbar sind. Durch Ausrechnen der Höhe der Patches wurde jeweils die Entfernung zum Patch ermittelt. Mit einigen Entfernungen ist es ähnlich wie bei GPS möglich die Position zu ermitteln von der aufgenommen wurde. Diese Berechnung wurde beispielhaft durchgeführt, um herauszufinden, wie genau eine Lokalisierung möglich wäre. Dabei kam eine Genauigkeit von ca. 7-20cm heraus.
Bei der Durchführung dieser Machbarkeitprüfung stellte sich heraus, dass die Spiegelung der Patches auf dem Spielfeld ein großes Problem darstellen könnte. Da die Roboter nur beschränkte Ressourcen haben, ist ein Algorithmus, der mit der Spiegelung fertig wird eine Herausforderung. Ein weiteres Problem könnte die Bildqualität werden. Im Gegensatz zu einer Digital-Foto-Kamera sind die Bilder der tatsächlichen Roboterkamera schlechter. Ein anderes Problem könnte die Ausrichtung der Roboterkamera sein. Einerseits ist es wichtig, dass der Roboter den Ball noch sieht, wenn er direkt vor ihm liegt. Anderseits sollte der Roboter noch die jeweils hinterste Bande sehen. Aufgrund des geringen FOV (Field of View oder Sichtfeld) der Roboterkamera, könnte das ein Problem werden.

II.3.2.3 Hardware und Sensorik

Ziel dieser Gruppe war es, den Beschleunigungssensor und den Kompasssensor auf einem neuen Roboter zu testen. Dazu sollte zunächst eine SPI-Bibliothek geschrieben werden und an den Beschleunigungssensor angepasst werden. Eine weitere Aufgabe war es, die vorhandene I²C-Bibliothek an den Kompasssensor anzupassen und ein entsprechendes Modul zur Ansteuerung des Sensors zu schreiben. Außerdem sollten Vorschläge für das Design der neuen Roboter gemacht werden. Hierzu wurde zusammen mit den HiWis ein Brainstorming durchgeführt, in dem Ideen wie die asymmetrische Form, die Bereifung und das Gewicht der neuen Roboter zur Sprache kamen. Es sollte ein weiterer Prototyp mit diversen Änderungen gefertigt werden. Nach Erarbeitung und Einreichen der Änderungen beim IML kam heraus, dass der neue Prototyp zum Testen erst im August zur Verfügung steht. Für einige der Gruppen, ins besondere die Gruppe Hardware bedeutete dies eine Verzögerung des Arbeitsflusses, daher wurde später versucht die Ziele entsprechend anzupassen. Daraus ergab sich das Problem, dass die Bibliotheken für die Sensoren zwar geschrieben, aber zunächst nicht getestet werden konnten.
Mangels Hardware hat die Gruppe versucht, auf dem Testboard EZDSP TMS320F2812 von Spectrum digital die I²C-Bibliothek zu testen. Hierzu wurde ein EEPROM über I²C an den DSP angeschlossen. Die Arbeit mit dem Testboard ergab, dass die vorhandene I²C-Bibliothek nicht sauber funktionierte. Daher wurde für die nächste Phase als Ziel gesetzt, einen neue I²C-Bibliothek zu schreiben. Die gesammelten Erfahrungen mit Code Composer Studio und dem Debuggen über das JTAG-Interface in einer Anleitung[6] zusammengefasst.

II.3.3 Tätigkeiten und Ereignisse außerhalb der Kleingruppen

II.3.3.1 Neuer PG- Raum

Nebenher wurde ein weiterer Raum zum Arbeiten hergerichtet. Der Raum wurde komplett umgeräumt, vorhandene Gerätschaften und Material gesammelt, Rechner aufgebaut, installiert und eingerichtet so dass dort die neuen Roboter programmiert und getestet werden können. Außerdem wurde ein Gruppentisch mit Verkabelung für Notebook-Arbeitsplätze vorbereitet.

II.3.3.2 Einrichten eines weiteren Arbeitsplatzes zur Programmierung der Roboter

Um effizienter arbeiten zu können hat die Gruppe Hardware ein älteres JTAG-Modul wieder in Betrieb genommen. Es lässt sich an den parallelen Port eines Rechners anschließen und kann auch an

[6] https://www.robosoccer.de/intern/files/Anleitungen/Bestandsaufnahme_PG472/Gesammelte_Erkenntnisse.pdf

einem modernen Notebook betrieben werden. Die Gruppe hat ein Softwarepaket[7] zusammengestellt, das auch Treiber für modernere Windows-Systeme beinhaltet. Die Software und ihre Installation wurde auf einem Arbeitsrechner in dem neuen Pool und auf verschiedenen Notebooks getestet.
Die Roboter können damit zwar nicht geflasht werden, aber die DSP-Software kann mit Hilfe dieses JTAG-Moduls getestet werden.

II.3.3.3 Betreuung der Roboter und des Publikums auf dem Campus-Fest

Ein in der Ausschreibung angegebenes PG- Ziel wurde noch einmal als wichtig herausgestellt: die Gruppe soll in der Lage sein das vorhandene System aufzubauen und in Betrieb zu nehmen. Daher hat die Gruppe am 18.6. das System auf dem Campusfest mit der Unterstützung zweier Hiwis betreut. Die Präsentation der Roboter auf dem Campusfest hat sich als Erfolg herausgestellt, der Stand war zu jeder Zeit gut besucht und die betreuenden Hiwis und PG- Mitglieder waren ständig mit Arbeiten oder Gesprächen mit Interessenten ausgelastet.

II.3.3.4 Exkurs zur EM 2005 nach Holland

Die Gruppe unternahm eine Exkursion nach Holland zur EM 2005, um den aktuellen Stand der Entwicklung des eigenen und der anderen Teams im Turnier zu beobachten. Nach dem mehr als enttäuschenden Abschneiden der Dortmund Droids in allen Klassen wurde beschlossen dass die Gruppe sich gemäß definiertem Semesterziel mehr mit dem bestehenden System befassen sollte um einige Verbesserungen herbeizuführen. Daher wurde danach eine Sitzung mit den Hiwis abgehalten um die bestehenden Probleme zu sammeln und zu besprechen. Die Gruppe kam zu dem Schluss, dass an vielen Punkten auch noch im vorhandenen System gearbeitet werden muss, um eine Konkurrenzfähigkeit mit anderen Teams zu erzielen und zu gewährleisten. Konkurrenzfähigkeit ist ein wichtiger Bestandteil der Öffentlichkeitsarbeit, da der Roboterfußball wenig populär ist. Der Erfolg des Teams spielt auch für die Finanzierung des Projektes eine wichtige Rolle.
Dabei sollte der Forschungsgedanke und das Ziel der Entwicklung autonomer Roboter jedoch nicht komplett verworfen werden. Dies bringt beim aktuellen Stand der Entwicklungen keinen Vorteil gegenüber anderen Teams, kann aber einen großen Vorsprung in der Zukunft bedeuten und sollte auf jeden Fall weiterverfolgt werden. Ein Großteil der vorhandenen Arbeitsleistung sollte jedoch von hier an in die Optimierung und Verbesserung des bestehenden Systems gelenkt werden. Dies ist auch insofern sinnvoll, da sich die Fertigung und Auslieferung der neuen Roboter verzögert hatte und zu wenig Hardware für alle Mitglieder vorhanden war.

[7] https://www.robosoccer.de/intern/files/CodeComposer/JTAG_parallelport

II.4 Dritte Kleingruppenphase

II.4.1 Gruppen und Ziele

Die EM läutete die dritte Phase ein. Am 20.06.2005 wurde daher folgende neue Gruppeneinteilung festgelegt, in die die Hiwis fest eingebunden wurden:

Gruppe „Bildverarbeitung“

Teilnehmer

HiWi: Thomas Pfeifer

PG: Christian Kintzel, Christian Kolek, Altay Cebe

Gruppe „Strategie“

Teilnehmer:

HiWi: Marco Wickrath

PG: Daniel Mikus, Kristjan Pulina, Christian Tesch

Gruppe „Anfahrt/Funk/Regelung“

Teilnehmer:

HiWi: Marco Wickrath, Christoph Michalski, Sven Bursch, Volkmar Frinken

PG: Tobias Jäger

Gruppe “Hardware”

Teilnehmer:

HiWi: Simon Schulz

PG: Tom Bomhof, Sarah Koenig, Christoph Ewerlin, Andreas Nettsträter, Rainer Oye

Gruppe „Investition“

Teilnehmer:

HiWi: Sven Bursch, Simon Schulz

PG: Rainer Oye, Tobias Jäger

Gruppe „Spielregeln“

Teilnehmer:

HiWi: Christoph Michalski, Marco Wickrath

PG: Andi Nettsträter, Sarah Koenig, Rainer Oye

Die ersten 4 Gruppen sollten auf Semesterziele hinarbeiten. Die Gruppen **Investition** und **Spielregeln** befassten sich aus gegebenem Anlass lediglich eine kurze Zeit mit den Aufgaben.

II.4.2 Ergebnisse der dritten Kleingruppenphase

II.4.2.1 Bildverarbeitung

Wie bereits weiter oben erläutert wurde, ist der Bericht der Gruppen **Deckenkamera/ Entzerrung/ Bildverarbeitung** unter diesem Punkt zusammengefasst.

Die Entwicklung der Entzerrerklasse geschah grob in zwei Schritten:

- Entwicklung einer ersten Entzerrerklasse als "Proof of Concept".
- Weiterentwicklung der Entzerrerklasse, die als Ersatz für die bisherige Entzerrung in Robotsoccer eingesetzt werden kann.

In beiden Phasen war es notwendig eine GUI für die neue Entzerrung zu schreiben, um zu sehen, ob die erzielten Resultate tatsächlich den erwarteten entsprachen.
Um die Entzerrerklasse auf ihre tatsächliche Praxistauglichkeit zu testen, wurde zunächst ein Testprogramm namens "Entzerrer Testbox" geschrieben. Dieses Programm visualisiert die Linsenentzerrung sowie die Abbildung (Mapping) des entzerrten Spielfeldbildes auf ein Rechteck. Dabei werden die Arbeitsschritte bereits in etwa so durchlaufen, wie es auch nachher im eigentlichen (Robotsoccer) System sein wird. Das Programm wurde in C++ entwickelt und nutzt die QT Bibliothek.
Der Programmablauf ist wie folgt:
Im ersten Schritt wird über den "Load" Button ein Kamerabild geladen, welches anschließend in einer zweigeteilten Ansicht jeweils oben und unten dargestellt wird. Im zweiten Schritt trägt der Benutzer die in Matlab ermittelten intrinsischen Kameraparameter ein. Anschließend kann über den "Update" Button die Linsenentzerrung auf das geladene Bild angewendet werden. Nach kurzer Rechenzeit wird das Resultat in der unteren Ansicht dargestellt.

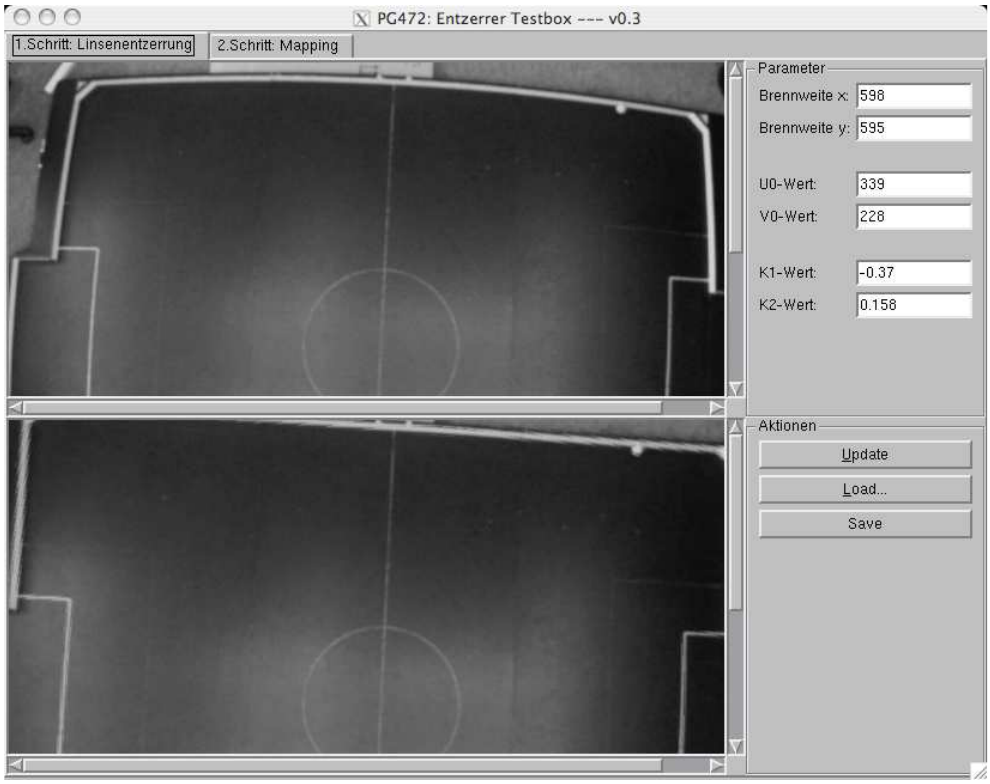


Abbildung II.4-2: Entzerrung

Der Aufbau des zweiten Tabs ähnelt stark dem Ersten. Das von der Linsenverzerrung befreite Kamerabild wird diesmal auf der oberen Ansicht gezeigt. Durch Klicken und Ziehen können nun auf diesem Bild vier Eckpunkte gesetzt werden. Diese dienen dazu den Bereich des Bildes zu definieren, der auf ein Rechteck (genauer gesagt auf ein 640 * 480 Bild) abgebildet werden soll. Die gesetzten Koordinaten erscheinen rechts im Panel "Mapping Eckpunkte" und können dort auch feiner justiert werden. Wurde das Rechteck gesetzt, kann das Mapping über den "Update" Button angewendet werden. Das Resultat der Transformation erscheint in der unteren Ansicht. Sollte der Anwender wiederum mit dem Ergebnis nicht zufrieden sein, so kann er den Vorgang wiederholen.

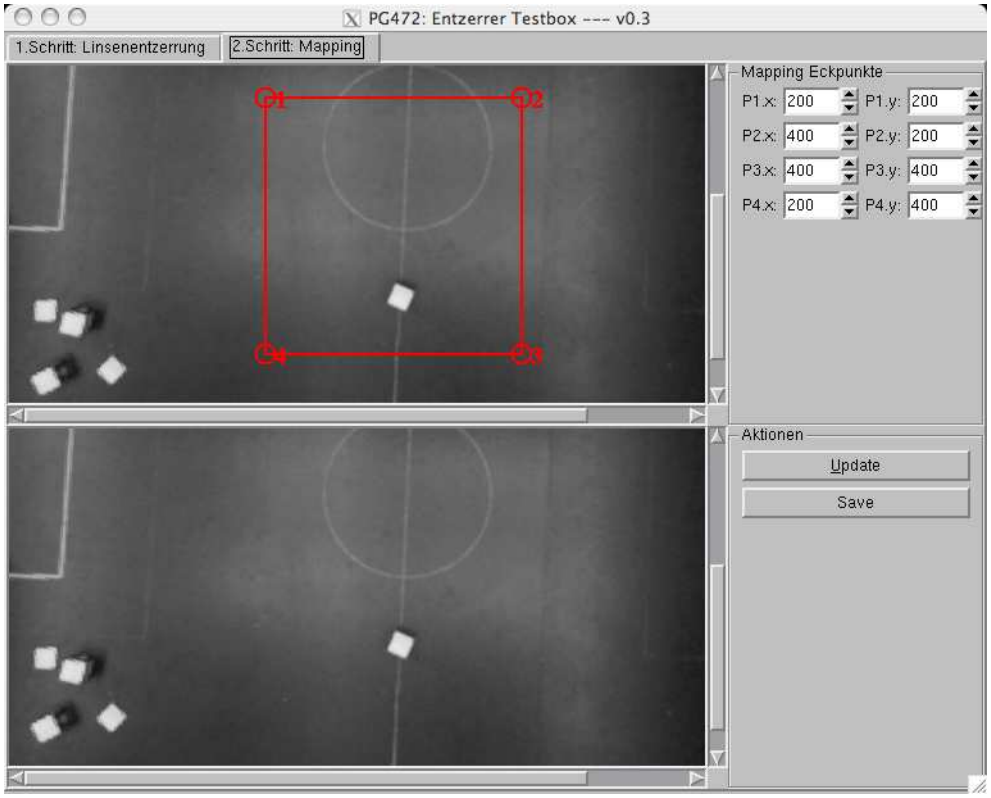


Abbildung II.4-4: Entzerrung

Mit dem Testprogramm konnte bereits eindrucksvoll demonstriert werden, dass das verwendete Verfahren zur Kameraentzerrung tatsächlich funktioniert. Der nächste Schritt war nun, das Verfahren in das bestehende robotsoccer System einzubauen und zu sehen, wie gut die Entzerrung innerhalb der tatsächlichen Software funktioniert.

Da die Implementierung einer neuen Entzerrung einen signifikanten Schritt darstellt, wurde hierfür ein neuer Branch "BV6" im CVS eröffnet. Es wurde eine - dem Testprogramm gegenüber erweiterte - Version der Entzerrerklasse verwendet. Für dessen GUI wurde mittels QT-Designer ein neues Tab im BV6Tab eingefügt. Die Interaktion zwischen GUI und der Entzerrerklasse findet zum größten Teil in der Klasse BV6Measurements statt. Der Aufbau und Ablauf der Entzerrereinstellung ist dem des Testprogramms ähnlich. Statt allerdings ein Tab zu benutzen, um zwischen der Einstellung der Linsenentzerrung und des Mappings hin und her zu schalten, wurden alle benötigen Eingabefelder rechts vom Bild platziert.

Abbildung II.4-6: BV-Einstellungen

Über eine Gruppe von Radiobuttons kann dann zwischen den benötigen Einstellungen hin und her geschaltet werden. Die Buttons rechts neben den Radiobuttons erlauben dann, bestimmte Aktionen (Aufnehmen eines Lifebildes, Anwendung der Entzerrung, Mapping) auszuführen. Da für die Parallaxenberechnung nachher die Kameraposition in Weltkoordinaten benötigt wird, kann diese hier ebenfalls über drei Spinboxen (X-Position, Y-Position, Höhe über dem Spielfeld) angegeben werden. Somit kann die neue Entzerrung komplett innerhalb des Robotsoccer System eingestellt werden.

Bis jetzt wurde nur das Spiel mit einer Kamera berücksichtigt. Wenn sich herausstellt, dass die neue Entzerrung zusammen mit dem Rest des Systems funktioniert, muss die GUI für einen Zweikamera Ansatz erweitert werden. Dies könnte z.B. realisiert werden, indem die intrinsischen Kameraparameter und die Mapping Punkte getrennt für das rechte und das linke Bild eingestellt werden.

II.4.2.2 Strategie

Da während der EM in Twente Anfang Juni 2005 festgestellt wurde, dass das schlechte Abschneiden unseres Teams nicht hauptsächlich an der alten Roboterhardware liegt, sondern sicherlich auch an Teilen unserer Strategie, hat die Projektgruppe daraufhin beschlossen, eine Strategie-Gruppe

einzuführen. Diese Strategie-Gruppe soll sich hauptsächlich mit der Verbesserung des vorhandenen Strategiesystems beschäftigen. Dazu sollen unter anderem Fehler entdeckt und korrigiert, aber auch komplett neue Spielzüge, Rollen und Handlungen implementiert werden.

In den ersten Wochen beschäftigen wir uns ausschließlich mit der Sammlung von Ideen für neue und vor allem bessere Spielzüge, Rollen und Handlungen. Außerdem sollte die Einarbeitung in die aktuelle Strategie erleichtert werden, indem der vorhandene Quellcode der Handlungsklassen geeignet kommentiert und dokumentiert wird. Um den Quellcode allerdings komplett zu verstehen, wurden einzelne Handlungen am echten Spielfeld getestet. Um mit unserem robosoccer-Hauptprogramm auch wirklich nur eine einzelne Handlung auf dem Spielfeld zu testen, müsste eigentlich das Bewertungssystem zur Auswahl geeigneter Handlungen komplett ausgeschaltet werden. Da dies aber nicht trivial und schnell lösbar ist, gibt es die Variante, in der „TestSpielzug.cpp“ die Bewertung von 0.0f auf 1.0f hinaufzusetzen. Dies bewirkt, dass die in „TestSpielzug.cpp“ zugewiesenen Rollen zu 100 Prozent vergeben werden. In unserem vorhandenen System wird so beispielsweise die „TestRolle“ an alle Roboter vergeben. Da eine Rolle aus einer oder mehreren Handlungen besteht, kann man nun daher gehen und „TestRolle.cpp“ editieren. Dort lässt sich die zu testende Handlung eintragen, die dann aufgrund der Bewertungsänderung in „TestSpielzug.cpp“ konstant für jeden Roboter ausgewählt wird.

Schon während des Testens einfacher Handlungen wie „BallBlocken“ oder „BallFuehren“ fiel uns auf, dass dort teilweise Zielpositionen vergeben wurden, die weit außerhalb des Spielfeldes lagen. Das Resultat konnte man schon während der EM beobachten: Einige Roboter fuhren mit höchster Geschwindigkeit gegen die Bande und ließen nicht davon ab, vor der Bande stehend „weiterzufahren“. Dieser Fehler wurde inzwischen weitestgehend behoben; die Methode „Zielposition();“ ist veraltet und sollte generell nicht mehr aufgerufen werden. Die richtigen Zielpositionen werden schon an anderer Stelle im Quellcode korrekt gegeben und der Aufruf von „Zielposition()“ ist daher eigentlich nur destruktiv.

Da die Dokumentation der vorhandenen Handlungen weitestgehend abgeschlossen ist, und es im Spiel an Angriffshandlungen mangelt, werden wir uns im Folgenden mit der Implementierung eigener, neuer und taktisch kluger Angriffssituationen beschäftigen. Eine Möglichkeit einer neuen Angriffssituation ist z.B. das Führen des Balls in Richtung rechten Torpfosten, worauf ein plötzliches Abbremsen des ballführenden Roboters und gleichzeitiges Anfahren eines vorher korrekt vor dem Strafraum positionierten und in Richtung linker Torecke ausgerichteten Roboters erstens Verwirrung beim Gegner stiftet und zweitens eine gute Torsituation provoziert. Der Gegner positioniert seine Abwehr vor dem rechten Torbereich, obwohl im Endeffekt der Ball im letzten Moment in die linke Seite transportiert wird.

Die Implementierung und das Testen neuer Handlungen und Rollen wird während der vorlesungsfreien Zeit von der Strategiegruppe fortgeführt.

II.4.2.3 Anfahrt/Funk/Regelung

Untergruppe **Anfahrt:** Die Wegberechnung des Robosoccer-Systems auf dem Host-System erfolgt durch Konstruktion einer S-Kurve vom Start- zum Zielpunkt mithilfe von zwei Kreisen und einer verbindenden Tangente. Allerdings handelt es sich bei dem aktuell verwendeten Algorithmus um ein mit der Zeit gewachsenes Konstrukt, weswegen eine Adaptierung für die neue Robotergeneration kompliziert wäre. Aus diesem Grund wird der Code zuallererst komplett neu strukturiert, wo möglich vereinfacht sowie nicht mehr gebrauchte Funktionalität entfernt. Gleichzeitig müssen bisher fest einprogrammierte Konstanten, welche sich auf die Geschwindigkeit der eingesetzten Kameras beziehen, aufgefunden werden. Sobald diese Aufgabe abgeschlossen und der Algorithmus auf Lauffähigkeit überprüft ist, wird er bezüglich der Eigenheiten der neuen Roboter ergänzt. Außerdem werden neue Funk-Befehle wie z.B. „Drehe um X Grad und fahre los mit Geschwindigkeit V“ und „Fahre mit Geschwindigkeit V und beschleunige dazu höchstens so stark“ hinzugefügt.

Untergruppe **Funk:** Die zwei wichtigsten den Funk betreffenden Aufgaben sind die endgültige Implementierung des bidirektionalen Funks sowie die Entwicklung einer Lösung, die das Aufkommen von Funklöchern minimiert. Die Technische Seite des bidirektionalen Funks wurde bereits auf den existierenden Funkmodulen implementiert, die zugehörigen Befehle werden noch entwickelt. Eine Möglichkeit zur Vermeidung von Funklöchern ist die Verwendung von zwei abwechselnd sendenden Funkmodulen und Antennen. Alternativ könnte auch die Bewegung des Moduls oberhalb des Spielfelds das Auftreten von Funklöchern vermeiden.

Untergruppe **Regelung:** Um die Bestimmung möglichst optimaler PID-Werte für die neuen Roboter zu erleichtern wird eine Testumgebung implementiert. Dadurch kann ein Intervall von Werten automatisch durchgetestet und das Ergebnis dieser Tests graphisch darstellt werden. Die Auswahl des Optimums wird von Hand vorgenommen. Mit diesem Tool kann es unter Umständen auch möglich sein, vor Turnieren die PID Werte auf das verwendete Spielfeld anzupassen. Sobald diese Aufgabe abgeschlossen und der Prototyp

II.4.2.4 Hardware

Den Zeitplan und die Ziele der Gruppe Hardware zeigt die folgende Grafik:

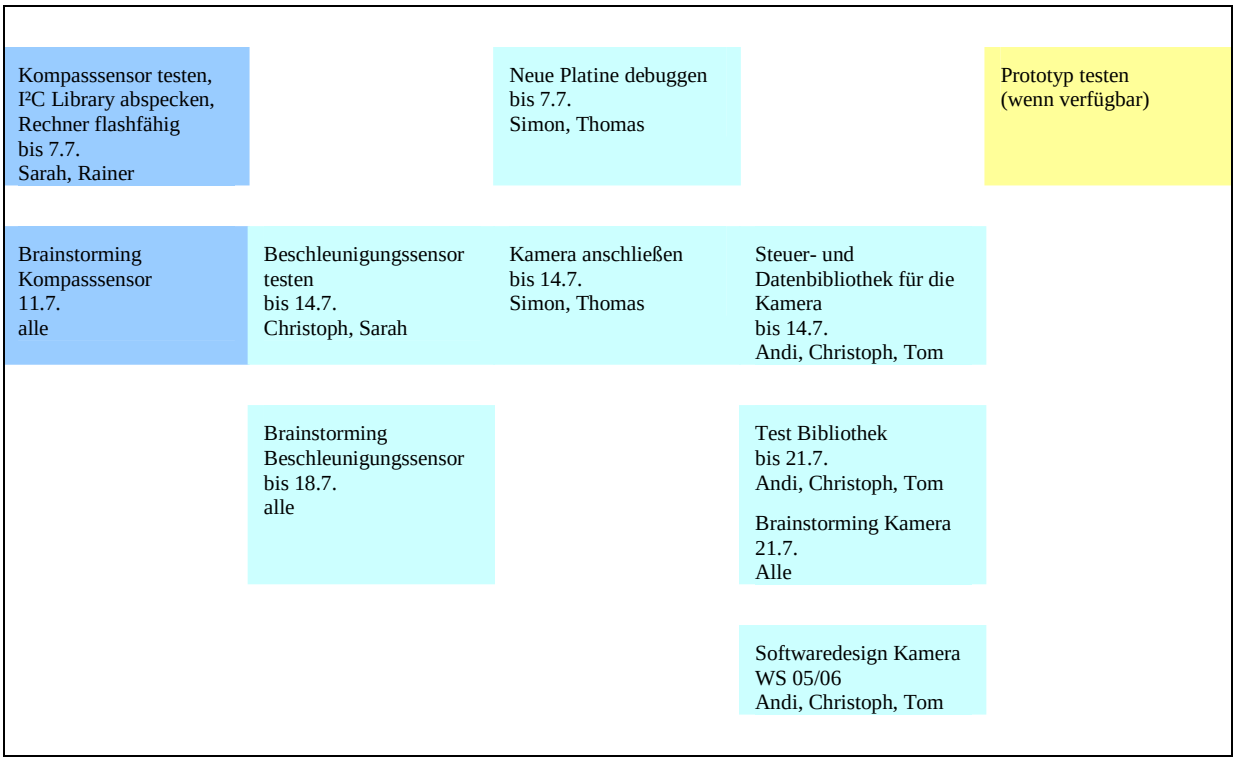


Abbildung II.4-7: Zeitplan Hardware

Test des Kompasssensors

Als die Gruppe Hardware Möglichkeiten zum Testen hatte, wurde festgestellt, dass das vorhandene I²C Modul kaum den geforderten Ansprüchen entsprechen konnte. Es war offensichtlich nicht als universelles Modul geeignet, wie es für eine derartige Anwendung notwendig gewesen wäre. Es war nicht ausreichend flexibel gestaltet um verschiedene Anbauten an den Roboter zu benutzen, das Konzept des I²C-Busses war nicht in Form einer allgemein zu verwendenden Funktionalität umgesetzt. Es war an einigen zentralen Stellen fehlerhaft. Es gab kein erkennbares Konzept für die Zustands-Steuerung des Busses, oder sie war nicht sauber implementiert. So musste die Gruppe zum Kompass-Sensor auch die I²C-Schnittstelle neu implementieren. Es wurde ein eindeutiges Konzept für den Treiber entwickelt und implementiert. Das Konzept ist in den Quellen kommentiert. Es entstand zunächst ein kleines, einfach zu durchschauendes Modul, das einer Standard-I²C-Anwendung genügt. Das Kompass-Modul wurde entsprechend dazu angepasst. Allzu viele Anpassungen waren aufgrund des bereits entwickelten Konzeptes nicht mehr notwendig. Die Module konnten mit einem V1-Board der neuen Roboter erfolgreich getestet werden.
Noch nicht fertig gestellt, aufgrund der Verzögerungen, sind ein optimiertes Timing für die Roboter-Boards und eine funktionierende Implementierung für das Board V2.

In einem Brainstorming sollen auf der Basis der gewonnenen Erkenntnisse Konzepte entwickelt werden, inwiefern die Verwendung des Kompass-Sensors im Roboterfußball –Spiels sinnvoll ist.

Lokale Kamera

Um die Kamera benutzen zu können, müssen mehrere Komponenten zusammenspielen. Zum einen wird natürlich die Kamera benötigt, um Bilder aufzunehmen. Der Takt für die Kamera muss gesondert angepasst werden, um die Kamera nicht zu zerstören. Konfigurationseinstellungen wie das Bildformat und der Bildausschnitt lassen sich über eine serielle Verbindung vornehmen, die aufgenommenen Bilder gehen dann im Normalbetrieb über eine parallele Verbindung an einen Averlogic – Framebuffer. Da dieser nur sequentiell gelesen werden kann, werden die Bilder aus dem Framebuffer in einen Bereich des RAMs kopiert, um einen wahlfreien Zugriff zu ermöglichen. Das interne RAM des TMS2812 reicht dafür nicht aus, die Daten werden daher in ein externes RAM abgelegt, das an das externe Interface des TMS2812 angeschlossen ist. Die implementierten Bibliotheken implementieren alle Funktionen, die für ein reibungsloses Zusammenspiel der angesprochenen Komponenten notwendig sind. Zu Testzwecken bietet die Steuer- und Datenbibliothek die Möglichkeit, aufgenommene Bilder an einen angeschlossenen PC weiterzuleiten. Um die dort empfangenen Daten darstellen zu können, wurde von der Gruppe ein kleines Programm entwickelt.
Zu Beginn des nächsten Semesters steht die interne Bearbeitung der Kameradaten an erster Stelle.

II.4.2.5 Investition

Die Gruppe **Investition** hat drei verschiedene Angebote für die Neuanschaffung von Host-Rechnern erarbeitet und die Vorschläge eingereicht. Die Gründe hierfür sind:

- die vorhandenen Rechner arbeiten nicht mehr zuverlässig (fallen aus, überhitzen, evtl. Schäden durch Transporte oder Verschleiß durch Dauerbetrieb)
- in einem Spiel 11 gegen 11 arbeiten die Hostrechner an ihrer oberen Leistungsgrenze. Für Spiele gegen Teams mit neuen, schnellen Robotern und den dadurch bedingten schnellen Ballbewegungen können die Rechner mit der aktuellen Software teilweise den Ball nicht mehr erkennen.

II.4.2.6 Spielregeln

Ziel der Gruppe war es, Vorschläge für Anpassungen des bestehenden Regelwerks der FIRA an die neuen Gegebenheiten des Spieles zu machen. Dabei wurde besonders auf die hohe Geschwindigkeit

der Roboter eingegangen, da hierdurch oft ein genaues und faires Spiel nicht möglich ist. Häufig werden gegnerische Roboter nicht umfahren, sondern einfach aus dem Weg geschoben oder gerammt. Da der Roboterfußball als ein Ziel die Entwicklung von Robotern hat, kann dies nicht Sinn des Spiels sein.

Folgende Vorschläge wurden von der Gruppe gemacht:

Die Goalkeeper-Charging Regel soll erweitert werden, so dass jede aktive Berührung des Torwarts (auch ohne Ball) zum Charging führt.

Wenn sich ein defekter Roboter auf dem Spielfeld befindet, sollte unterschieden werden, ob die Situation aus eigener Schuld oder durch ein Foul entstanden ist. Im ersten Fall sollte in der nächsten neutralen Situation ein Freeball gegeben werden, im zweiten ein Freekick.

Bei falsch ausgeführten Standardsituationen (insbesondere Anstoß) soll die Situation einmal wiederholt werden. Wenn dann wieder nicht richtig ausgeführt wird, soll ein Freekick gepfiffen werden. Diese Situation wird üblicherweise bereits so behandelt, und sollte daher auch in den Regeln aufgenommen werden.

Seitenwechsel zur Halbzeit muss Pflicht sein.

Automatische Anfahrt von Standardpositionen muss Pflicht sein.

Das System muss den Schiedsrichterbutton unterstützen.

Fouls sollten stärker geahndet werden. Wenn ein aktiver, ballführender Roboter einen passiven, stationären Gegner rammt (es muss nicht unbedingt der Torwart sein) sollte ein Foul gepfiffen und mit einem Freekick bestraft werden.

Es wird ein Passivitäts-Penalty als neue Strafe vorgeschlagen:

Die zu bestrafende Mannschaft darf sich nach dem Anpfiff 5 sec. lang nicht bewegen.

5vs5 soll zukünftig auf den größeren 7vs7 Spielfeldern gespielt werden. Dadurch wird einerseits ermöglicht, die Geschwindigkeiten der neuen Roboter zu nutzen, andererseits bietet das größere Spielfeld auch mehr Platz zum Ausweichen. Zusammen mit der Einführung strengerer Foulregeln wird dadurch ein schnelles Spiel, aber auch das Erkennen und Umfahren von anderen Roboter gefördert.

Timeout Regel soll vereinheitlicht werden. Zur Zeit gibt es zum Beispiel für 5vs5 andere Timeout Regeln als für 7vs7.

III.Gruppe Anfahrt

III.1 Aufgaben

Die Gruppe „Anfahrt“ wurde mit der Aufgabe betraut, den bestehenden Anfahrtsalgorithmus für die Pfadberechnung und Robotersteuerung auf Seiten des Host-Systems zu überarbeiten.
Insbesondere sollten folgende Teilaufgaben durchgeführt werden:

1. Code aufräumen und gegebenenfalls neu strukturieren:
   Der bisher verwendete Algorithmus ist ein über einen längeren Zeitraum gewachsenes Konstrukt, was Inkonsistenzen und redundante Funktionalität zur Folge hat. Eine Adaption an die neuen Roboter ohne vorherige Überarbeitung des Codes wurde daher als unpraktikabel eingestuft.
2. Fest-kodierte Werte für externe Komponenten suchen und entfernen oder markieren:
   Einige der Berechnungen im Anfahrtsalgorithmus benötigen Parameter externer Komponenten des Gesamtsystems, wie zum Beispiel die Anzahl der von der in der Sekunde verarbeiteten Bilder (fps) oder den Abstand der zwei Räder eines Roboters. Diese wurden in der Regel lokal fest einprogrammiert, wodurch bei einer Änderung der Parameter eine zeitaufwändige Durchsuchung des Codes notwendig war. Unnötige Vorkommen solcher Werte sollten entfernt, nötige zwecks schnellem Auffinden entsprechend markiert werden.
3. Die Behandlungen von Sonderfällen prüfen:
   Verschiedene während eines Spiels vorkommende Situationen wie plötzliches Bremsen oder das Wegfallen eines Zielpunktes wurden in separaten Methoden behandelt, die nachträglich in den normalen Ablauf des Anfahrtsalgorithmus eingeschoben wurden. Diese Methoden sollten überprüft und, so möglich, besser in den eigentlichen Algorithmus eingebettet werden.
4. „Unschönes“ Fahrverhalten verhindern:
   Der verwendete Anfahrtsalgorithmus beruht auf der Idee, Pfade in Form von S-Kurven ausgehend von den Start- und Zieltangenten zu planen. Dies führt dazu, dass Roboter unter bestimmten Umständen für Zuschauer unschöne oder ohne das nötige Hintergrundwissen unverständliche Pfade vom Ball weg fuhren. Abgesehen von diesem ästhetisch unschönen Aspekt ist so ein Verhalten während eines Spiels auch offensichtlich unpraktisch und gegebenenfalls sogar gefährlich. Aus diesem Grund sollten derartige Problemfälle gesucht und ihnen vorgebeugt werden.
5. Anpassungen an die neuen Roboter vornehmen:
   Um eine zufriedenstellende Pfadberechnung zu gewährleisten, wurden die internen Werte des Anfahrtsalgorithmus an die Eigenschaften und das Fahrverhalten der zum jeweiligen Zeitpunkt eingesetzten Roboter angepasst. Infolgedessen erforderte die Umstellung auf neue

Roboter eine erneute Optimierung der im Code eingestellten Parameter. Im aktuellen Fall wurde durch das unsymetrische Design der neuen Roboter sogar eine grundlegende Erweiterung der Funktionalität des Anfahrtsalgorithmus nötig, um der Strategie-Komponente des System die Möglichkeit zu geben, die Unterscheidung in Ballführungs- und Schuss-Seite zu nutzen.

6. Einsatzmöglichkeiten des neuen Funkprotokolls prüfen:
   Das im Rahmen von Projektgruppe 449 von Thomas Pfeiffer entwickelte neue Funkprotokoll für den Einsatz auf den neuen Robotern in Zusammenspiel mit der Option bi-direktionalen Funkverkehrs bietet die Möglichkeit für erweiterte Befehle. In Zusammenarbeit mit den für die Einstellung der Regler zuständigen HiWis sollten daher der Nutzen möglicher Erweiterungen geprüft werden und, so möglich, die Voraussetzung für den Einsatz dieser erweiterten Befehle geschaffen werden.

III.2 Grundlagen Anfahrtsalgorithmus

Der auf dem Host-System eingesetzte Anfahrtsalgorithmus hat die Aufgabe, für einen gegebenen Roboter einen möglichst kurzen beziehungsweise schnell zurücklegbaren Pfad zu einem von der Strategie bestimmten Zielpunkt auf dem Spielfeld zu errechnen. Weitere von der Strategie übergebene Parameter wie die erforderliche Endgeschwindigkeit am Zielpunkt, den Winkel, in dem der Zielpunkt erreicht wird oder – im Falle von unsymetrischem Design – die Ausrichtung des Roboters, müssen in diese Berechnung mit einbezogen werden.

Unter Berücksichtigung der aktuellen Bewegung des Roboters wird ein Pfad konstruiert, der aus zwei Kreisen und einer sie verbindenden Tangente besteht (siehe **Fehler! Verweisquelle konnte nicht gefunden werden.**). Der Radius der Kreise wird durch die Geschwindigkeit (aktuelle respektive Zielgeschwindigkeit) sowie die minimal und maximal möglichen Radien bestimmt. Unter der Annahme, dass der Zielpunkt möglichst schnell erreicht werden soll, wird für den Großteil der Tangente die maximal erlaubte Geschwindigkeit angesetzt. Demzufolge muss ein erstes Teilsegment für die Beschleunigung von Kreis- auf maximale Geschwindigkeit und analog ein Segment zum Abbremsen vorgesehen werden.

Optionale Komponenten des Pfades sind eine Startgerade, falls der Roboter aktuell mit einer für eine Kreisfahrt zu hohen Geschwindigkeit fährt, sowie eine Zielgerade zum Erreichen der vorgegebenen Zielgeschwindigkeit nach Verlassen des Zielkreises.

Eine weitere Option ist, den Startkreis durch einen Radius von 0 effektiv zu umgehen. Dies geschieht insbesondere dann, wenn der Roboter zum Zeitpunkt der Pfadplanung steht, also erst eine längere Beschleunigung notwendig wäre. In diesem Fall wird eine Drehung auf der Stelle und anschließend direkt die Tangentenfahrt geplant.

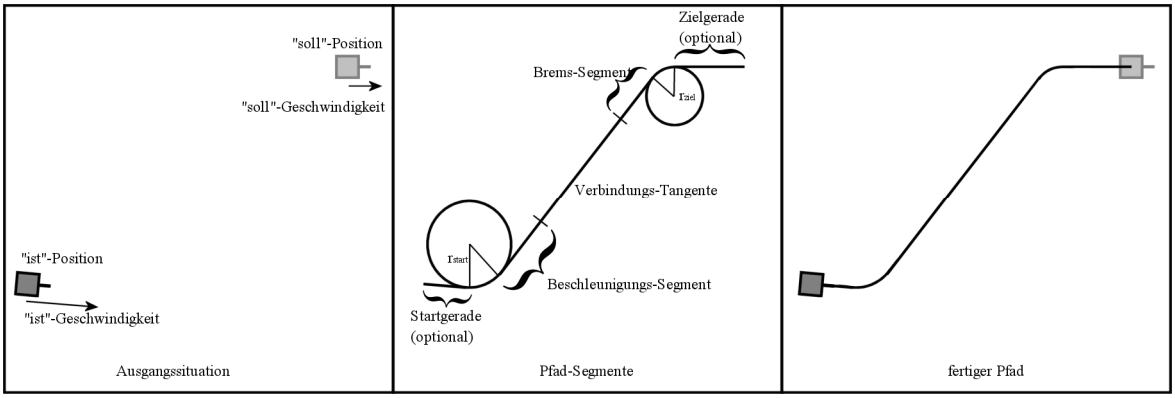


Abbildung III.2-1: Pfadplanung

Nach diesem Schema werden mehrere leicht unterschiedliche Pfade konstruiert, wobei jedes Mal geprüft wird, ob der Pfad an irgendeiner Stelle eine oder mehrere Banden berührt oder gar kreuzt. Aus den gültigen Pfaden wird anschließend derjenige ausgewählt, für den die berechnete Fahrtdauer minimal ist.

Dieser Pfad wird nun solange beibehalten, wie sich die von der Strategie übergebenen Zielparameter nicht so stark ändern, dass die berechnete Zielposition ungültig würde.

Anschließend wird in jedem Frame geprüft, ob sich der Roboter an der vorberechneten Stelle auf dem Pfad befindet und es werden, ausgehend von der aktuellen Position, Ausrichtung und Geschwindigkeit des Roboters, die auszuführenden Steuerbefehle ausgewählt und an das Roboter-Objekt übertragen.

III.2.1 Probleme der ursprünglichen Implementierung

Während die eingangs beschriebene S-Konstruktion es theoretisch ermöglicht, jede Zielposition zu erreichen, offenbaren sich im praktischen Einsatz eine Anzahl von Schwachstellen.

So ist zum Beispiel nicht vorgesehen, einen Pfad als Gerade zwischen Start- und Zielpunkt zu planen oder, falls eine Zielgeschwindigkeit vorgegeben ist, der Roboter aber steht, auf einer Kreisbahn zu beschleunigen. Stattdessen wird ihm letzteren Fall eine vom Ziel weg führende S-Kurve mit Startgerade konstruiert, was, wie bereits erwähnt, sowohl ästhetisch unschön als auch spieltechnisch gefährlich ist (siehe Abbildung III.2-2).

Ein weiteres Problem ist, dass die von der Strategie übergebene „soll“-Ausrichtung lediglich festlegt, von wo die Zielposition angefahren wird. Daher lässt sich dieser Wert nicht nutzen, um eine Anfahrtsrichtung für unsymetrische Roboter vorzugeben. Dies wiederum hat zur Folge, dass sämtlichen Entscheidungen während der Pfadplanung potentiell eine „falsche“ Zielausrichtung zugrunde liegt.

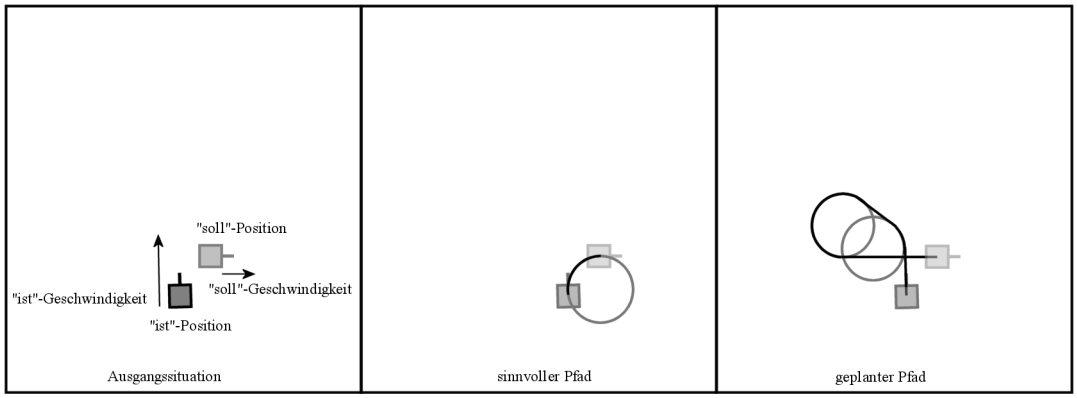


Abbildung III.2-2: Probleme bei der Pfadplanung

Eine mit der eigentlichen Konstruktion des Pfades indirekt verwandte Problematik ergibt sich aus der Tatsache, dass es sich bei den Robotern um physikalische Objekte handelt und nicht, wie für die durchgeführten Berechnungen vorausgesetzt, um Idealmodelle. So ist es offensichtlich, dass die berechneten Werte von den physikalischen Resultaten mehr oder minder stark abweichen. Verstärkt wird dieser Effekt durch die – bezogen auf die Genauigkeit der Berechnungen relativ unpräzise – Informationsgewinnung mithilfe der Deckenkamera und die ebenfalls hieraus resultierende zeitliche Verzögerung, mit der diese Informationen dem Anfahrtsalgorithmus zur Verfügung stehen.

Geringfügige Abweichungen des vorhergesagten vom tatsächlichen (beziehungsweise von der Bildverarbeitung als tatsächlich zurückgegebenen) Verhalten können gegebenenfalls durch Anpassung der in den folgenden Frames an den Roboter gesendeten Steuerkommandos kompensiert werden. Bei signifikanter Differenz jedoch wird erkannt, dass der berechnete Pfad nicht mehr eingehalten werden kann und der Anfahrtsalgorithmus startet infolgedessen eine komplett neue Pfadplanung. Die hieraus resultierenden Verzögerungen haben verständlicherweise negative Auswirkungen auf den Spielbetrieb, insbesondere weil zwar wiederum der schnellste Pfad gefahren, aber die vorab von der Strategie geplante Ankunftszeit nicht eingehalten wird.

Kurzum: der Anfahrtsalgorithmus liefert bei der Pfadberechnung einen Zeitpunkt, zu dem die vorgegebene Zielposition erreicht sein wird – vorausgesetzt, der Roboter verhält sich ideal. Aktuelle Informationen bezüglich der Einhaltung dieser Vorhersage werden nicht bereitgestellt.

III.2.2 Behandlung dieser Probleme in der neuen Implementierung

Im Zuge der Bereinigung und Umstrukturierung des Anfahrtsalgorithmus wurde versucht, Lösungen für die genannten Probleme zu finden und diese in den Algorithmus einzubauen.

Eine vergleichsweise problemlose Erweiterung erlaubte, die Unsymetrie der neuen Roboter zu berücksichtigen. Hierbei wurde ein neuer Parameter eingeführt, mithilfe dessen die Strategie festlegen kann, ob der Roboter seine Zielposition mit der Ballführungs-Seite (vorne) oder der Schuss-Seite (hinten) erreichen soll, oder ob es irrelevant ist.

Dieser Wert beeinflusst die Pfadplanung wie folgt:

- Falls der Roboter steht und der Parameter auf „vorne“ oder „hinten“ gesetzt ist, wird die Drehrichtung für die Drehung auf der Stelle entsprechend der gewünschten Ausrichtung gewählt.
- Steht der Roboter und der Parameter hat den Wert „irrelevant“, wird der kleinere Drehwinkel genommen.
- Fährt der Roboter und der Wert des Parameters gibt an, dass die Fahrtrichtung falsch ist, wird der Roboter zum Stehen gebracht und durch eine anschließende Neuberechnung des Pfades auf der Stelle gedreht.

Das Problem der „falschen“ S-Kurven bei unglücklicher Konstellation von Start- und Zielposition hingegen erforderte eine bei weitem aufwändigere Behandlung. Da das Problem insbesondere dann auftritt, wenn die beiden Positionen nahe beieinander liegen, wurde speziell für diesen Fall ein separater Anfahrtsalgorithmus entworfen, der die normale Pfadplanung und somit die damit einhergehenden Restriktionen umgeht.

Bezüglich der Korrektur von Abweichungen wurde ein Kompromiss eingegangen. Sollte der Roboter bereits auf dem Startkreis ausreichend weit vom Pfad abkommen, dann würde eine Pfadkorrektur effektiv einer Neuberechnung gleich kommen, weswegen von einer erweiterten Behandlung dieses Falles Abstand genommen wurde. Stattdessen wurde die Korrektur auf der Tangente erweitert, da dieser Abschnitt für das genaue Erreichen der Zielposition essentiell ist. Wird dabei festgestellt, dass die durchführbare Korrektur nicht ausreichend wäre, wird der Rest des Pfades so angepasst, dass die Zielposition weiterhin erreichbar bleibt.

Da aber auch eine derart erweiterte Funktionalität keine Aussage über den Zeitpunkt des Erreichens der Zielposition zuließ – was von der Strategie auch bis dato nicht vorgesehen war – wurde ein Mechanismus entworfen, um anhand von Positionsvergleichen eben solche Informationen zur Verfügung zu stellen. Des weiteren wurde die Möglichkeit geschaffen, von Seiten der Strategie Zeitvorgaben zu machen.

III.3 Neue Features

Im Folgenden soll darauf eingegangen werden, wie die drei vorgestellten neuen Funktionalitäten – teilweise Neuberechnung des Pfades, Zeitvorgaben sowie der spezielle Anfahrtsalgorithmus für nahe Zielpositionen – realisiert sind und auf welchen Ideen sie basieren.

III.3.1 Neuberechnung von Pfad-Teilsegmenten

Wie im vorherigen Kapitel bereits erläutert, wurde der Anfahrtsalgorithmus dahingehend erweitert, dass bei nicht kompensierbaren Abweichungen von der Tangente zwischen Start- und Zielkreis der restliche Pfad neu berechnet wird, um die Zielposition weiterhin zu erreichen.

Dies ist in Abbildung III.3-1 skizziert und im Detail wie folgt realisiert:

- Ausgehend von der aktuellen Position des Roboters sowie der Richtung der Tangente (es wird angenommen, dass der Roboter zumindest die Ausrichtung, wenn auch nicht die Tangente an sich, erreichen kann) wird ein Punkt berechnet, welcher als Eintrittspunkt für den neu zu konstruierenden Zielkreis dient.
- Es wird die Mittelsenkrechte der zwischen diesem Punkt und dem ursprünglichen Kreis-Austrittspunkt berechnet und mit der Senkrechten auf der Zielgeraden zum Schnitt gebracht.
- Dies ergibt den Mittelpunkt des neuen Kreises und über den Kreis-Austrittspunkt auch dessen Radius.
- Die Befehle, die an den Roboter übertragen werden, bewegen ihn möglichst genau auf den neuen Kreis-Eintrittspunkt und lassen ihn anschließend den vergrößerten oder verkleinerten Kreis mit einer entsprechend angepassten Geschwindigkeit fahren.

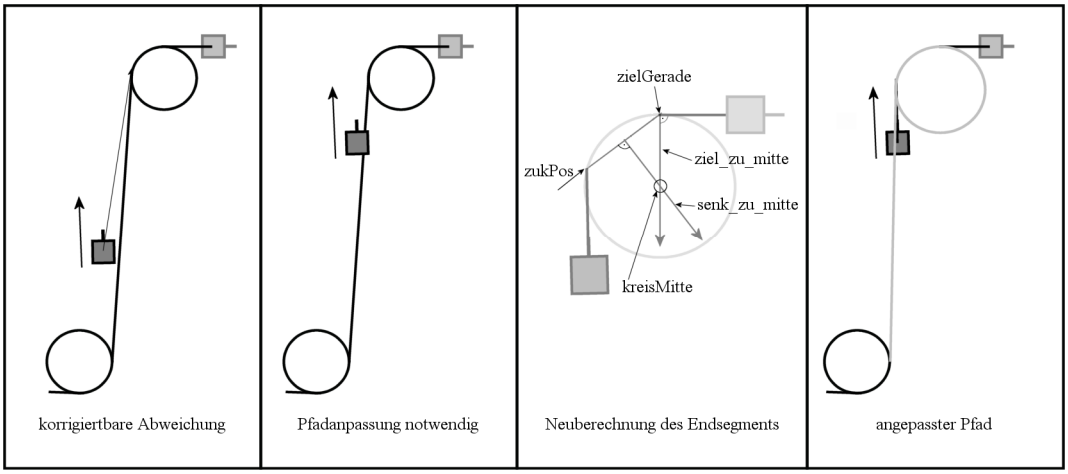


Abbildung III.3-1: Anpassung des Zielkreises

Im praktischen Einsatz hat sich gezeigt, dass diese Erweiterung eine durchaus geringere Abweichung der letztendlich erreichten Position von der vorgegebenen Zielposition bewirkt.
Es folgt die Implementierung der oben beschriebenen Berechnung, welche ursprünglich aus der Behandlung naher Zielpositionen hervorgegangen ist.

```
// relevante Berechnungen der Funktion zielRadius_anpassen()

// Mittelpunkt und Radius für den berechneten Kreis
 float kreisMitteX, kreisMitteY, kreisRadius;
// Stützvektor für die Mittelsenkrechte zwischen Ein- und Austrittspunkt
 float mittelSenkrechteX, mittelSenkrechteY;
// Richtungsvektor der Mittelsenkrechten
 float senk_zu_mitte_X, senk_zu_mitte_Y;
// auf Austritts-Tangente senkrecht stehender Richtungsvektor
 float ziel_zu_mitte_X, ziel_zu_mitte_Y;
// erlaubte Abweichung des Winkels = 5°
```

```
 float epsilon = M_PI / 18;

// voraussichtlicher Eintrittspunkt, abhängig von aktueller Position des
// Roboters und Richtung der ursprünglichen Tangente
// Wichtig: Y-Achse ist invertiert
 float entfernung = distance(istX, istY, zielKxStart, zielKyStart);
 float zukPosX = istX + cos(tangentenWinkel) * entfernung;
 float zukPosY =     istY - sin(tangentenWinkel) * entfernung;

// Bestimmung des Stützvektors der Mittelsenkrechten
if( zukPosX <= zielGeradeX ) {
 mittelSenkrechteX = zukPosX + 0.5 * ( zielGeradeX - zukPosX );
} else {
 mittelSenkrechteX = zielGeradeX + 0.5 * ( zukPosX - zielGeradeX );
};
if( zukPosY <= zielGeradeY ) {
 mittelSenkrechteY = zukPosY + 0.5 * ( zielGeradeY - zukPosY );
} else {
 mittelSenkrechteY = zielGeradeY + 0.5 * ( zukPosY - zielGeradeY );
};

// ist die Mittelsenkrechte parallel zur Zielausrichtung (+- Abweichung),
// dann ist der Kreismittelpunkt gefunden...
if(    ( fabs( sollA – ( atan2_pos(      mittelSenkrechteY – zielGeradeY,
                                 zielGeradeX - mittelSenkrechteX )
      ) ) > ( M_PI_2 - epsilon ) ) &&
 ( fabs( sollA - ( atan2_pos(     mittelSenkrechteY – zielGeradeY,
                                 zielGeradeX - mittelSenkrechteX )
      ) ) < ( M_PI_2 + epsilon ) )
) {
 kreisMitteX = mittelSenkrechteX;
 kreisMitteY = mittelSenkrechteY;
 kreisRadius =
      distance( zukPosX, zukPosY, zielGeradeX, zielGeradeY ) / 2;
 } else {
// ...sonst muss er errechnet werden

// Richtungsvektoren fuer Punkt-Richtungsform
 ziel_zu_mitte_Y = - sin( sollA + M_PI_2 );
 ziel_zu_mitte_X = cos( sollA + M_PI_2 );
 senk_zu_mitte_Y = - sin( M_PI_2 + atan2_pos(
      mittelSenkrechteY – zielGeradeY,
      zielGeradeX - mittelSenkrechteX ) );
 senk_zu_mitte_X = cos( M_PI_2 + atan2_pos(
      mittelSenkrechteY – zielGeradeY,
```

```
      zielGeradeX - mittelSenkrechteX ) );

// Division durch Null vermeiden
 if( ziel_zu_mitte_X == 0.0f )
      ziel_zu_mitte_X = 1/1000;
 if( ziel_zu_mitte_Y == 0.0f )
      ziel_zu_mitte_Y = 1/1000;
 if( senk_zu_mitte_X == 0.0f )
      senk_zu_mitte_X = 1/1000;
 if( senk_zu_mitte_Y == 0.0f )
      senk_zu_mitte_Y = 1/1000;

// 2 Punkt-Richtungs-Formen umgestellt und ineinander eingesetzt
// Ergebnis: Radius
 kreisRadius =
 ( senk_zu_mitte_X /
      ( ( ziel_zu_mitte_Y * senk_zu_mitte_X ) -
      ( ziel_zu_mitte_X * senk_zu_mitte_Y ) )
 ) *
 ( mittelSenkrechteY - zielGeradeY +
      ( ( senk_zu_mitte_Y *zielGeradeX ) / senk_zu_mitte_X ) -
      ( ( senk_zu_mitte_Y * mittelSenkrechteX ) / senk_zu_mitte_X )
 );

// aus Radius und dem zur Zielgeraden senkrechtem Vektor
// Kreismittelpunkt berechnen
 kreisMitteX = zielGeradeX + ziel_zu_mitte_X * kreisRadius;
 kreisMitteY = zielGeradeY + ziel_zu_mitte_Y * kreisRadius;
```

Es wurde entschieden, diese Berechnung pro Pfad nur einmal durchzuführen, da es sich gezeigt hat, dass mehrere Korrekturen der zu fahrenden Radien negative Auswirkungen auf das Fahrverhalten des Roboters und damit auf das Erreichen der Zielposition haben.

III.3.2 Unterstützen von Zeitvorgaben

Es ist für die Strategie unerlässlich, zeitliche Zusammenhänge in die Entscheidungsfindung mit einzubeziehen. Daher berechnet der Anfahrtsalgorithmus für jeden Pfad den voraussichtlichen Zeitaufwand, welcher abgefragt werden kann. Dies ist aber ein rein theoretischer Wert, der auf durchgängig idealem Fahrverhalten und auf Erfahrungswerten für Latenzen basiert. Abweichungen vom Pfad, unvorhergesehene Verzögerungen oder auch nur geringfügige zusätzliche Latenzen führen dementsprechend dazu, dass diese Vorgabe nicht eingehalten wird.

Während dies an und für sich schon problematisch ist, kommt erschwerend hinzu, dass die Strategie weiterhin von dem ursprünglich übermittelten Zeitpunkt des Eintreffens ausgeht. Besonders auffällig

ist diese Unzulänglichkeit, falls ein rollender Ball getroffen oder beispielsweise ein Torschuss blockiert werden soll.

Um dieser Problematik entgegen zu wirken, wurde der im Anfahrtsalgorithmus bereits intern zur Pfadverfolgung verwendete Mechanismus dahingehend adaptiert, dass er die jeweils noch zu fahrende Zeit berechnet. Somit ist es, nach einem Vergleich der aktuellen Zeit mit der Zeit, zu dem der Pfad ursprünglich berechnet wurde, möglich, Diskrepanzen zwischen der tatsächlich zu fahrenden und der theoretisch übrigen Fahrtzeit zu bestimmen.

Ferner wurde ein neuer Parameter eingeführt, mit dem die Strategie festlegen kann, ob der Anfahrtsalgorithmus – gegebenenfalls auf Kosten der Genauigkeit – versuchen soll, die ursprünglich prognostizierte Fahrtzeit einzuhalten. Letzteres wird durch kontinuierliche Anpassung der Geschwindigkeiten nach oben oder unten verwirklicht. Möglich ist dies, da die im Programm eingestellte Höchstgeschwindigkeit in der Regel unterhalb der durch den Roboter gegebenen maximal kontrolliert fahrbaren Geschwindigkeit liegt.

Der Ablauf im Detail:

- Die Strategie übergibt die Zieldaten für eine Anfahrt und spezifiziert, ob bei der Anfahrt auf Einhaltung der berechneten Zeit geachtet werden soll.
- Der Anfahrtsalgorithmus berechnet den Pfad und speichert die aktuelle Systemzeit.
- In jedem Frame wird anhand der aktuellen Position des Roboters berechnet, wieviel Zeit theoretisch verstrichen sein sollte. Dies wird dazu benutzt, um festzustellen, wann welche Befehle gesendet werden müssen und somit bekannten Latenzen auszugleichen.
- Soll auf die Fahrtzeit eingehalten werden, wird dieser errechnete Wert mit der tatsächlich verstrichenen Zeit verglichen. Für den Fall, dass die Abweichung zwischen diesen Werten klein genug ist, wird nichts unternommen.
- Andernfalls wird bestimmt, ob die Differenz zwischen den beiden Zeiten innerhalb einer festen Toleranz liegt und so entschieden, ob der Zeitverlust aufgeholt werden kann oder nicht, und dies an die Strategie übermittelt. So kann diese für den nächsten Frame gegebenenfalls eine neue Zielposition bestimmen.
- Ist der Zeitverlust aufzuholen oder der Roboter schon zu weit auf dem Pfad, wird die in diesem Frame an den Roboter übermittelte Geschwindigkeit um den Faktor der Abweichung erhöht oder erniedrigt. Beschränkt wird dieser Faktor entsprechend der aktuellen Position des Roboters auf dem Pfad, da die maximal kontrolliert fahrbare Geschwindigkeit auf der Verbindungsgeraden zwischen Start- und Zielkreis höher liegt als in einer Kurve.

Es folgt die Implementierung der Faktorberechnung und Entscheidungsfindung.

```
// Auslesen der tatsächlich verstrichenen Zeit
 double abgZeit = get_abgelaufene_zeit();
```

```
// falls die Zeit irrelevant oder die Zielgeschwindigkeit Null ist, mit
// der ursprünglich geplanten Geschwindigkeit fahren
 if( ( sollV == 0 ) || ( !beachteZeit ) ) {
      speed_adjust = 1.0f;
 } else {
// auf dem Startkreis keine Anpassung vornehmen, damit eine kontrollierte
// Beschleunigungsphase möglich ist
      if( phase <= 3 ) {
       speed_adjust = 1.0f;
// auf dem Beschleunigungs-Segment der Verbindungsgeraden kann eine
// höhere Geschwindigkeit gefahren werden.
       if( akt_phase == 3 && phase == 2 ) {
             speed_adjust = 1.1f;
       };
      } else {
// 120 % der geplanten Geschwindigkeit sind immer möglich
       float max_faktor = 1.2f;
// generell mit minimal 20% der geplanten Geschwindigkeit fahren,
// da sonst nicht wieder ausreichend schnell beschleunigt werden kann
       float min_faktor = 0.2f;

// auf dem Hauptsegment der Verbindungsgeraden kann mit bis zu 140%
// gefahren werden
       if( phase <= 4 ) {
             max_faktor = 1.4f;
       };

// Zeitdifferenzen von einem Frame sind innerhalb der Toleranz, daher
// in diesem Fall keine Änderung vornehmen
       if( ( abgZeit >= ( gefahrene_zeit - 0.033f ) ) &&
             ( abgZeit <= ( gefahrene_zeit + 0.033f ) ) ) {
             si->RobSchafftAnfahrt[RobLogID - 1] = true;
             speed_adjust = 1.0f;
       } else {
// aus der Position auf dem Pfad berechnete Restzeit
             float rz_theoretisch = getRestZeit();
// aus Vergleich von verstrichener Zeit und prognostizierter Zeit
// gewonnene Restzeit
             float rz_real = gesamtzeit - abgZeit;

// ist der Roboter schneller als geplant (Position auf Pfad weiter vorne
// als tatsächlich verstrichene Zeit vorgibt) ist der Faktor < 1, sonst
// > 1
             speed_adjust =  rz_theoretisch / rz_real;
// sollte der Roboter schon angekommen sein und muss aber noch mehr als 1
```

```
// Frame fahren, kann die Zeit nicht mehr aufgeholt werden...
              if( rz_real < 0.0f && rz_theoretisch > 0.33f ) {
                    si->RobSchafftAnfahrt[RobLogID - 1] = false;
                    speed_adjust = 1.0f;
// ...ansonsten wird versucht, die Verspätung durch maximale
// Geschwindigkeitserhöhung auszugleichen
              } else if( rz_real < 0.0f ) {
                    speed_adjust = max_faktor;
              };
// sofern keine negativen Restzeiten (=Verspätungen) berechnet sind und
// der maximale Faktor nicht überschritten wird, mit diesem fahren
              if( speed_adjust > 0.0f &&
                    speed_adjust <= max_faktor )
              {
                    si->RobSchafftAnfahrt[RobLogID - 1] = true;
// dabei untere Schranke einhalten
                    if( speed_adjust < min_faktor ) {
                          speed_adjust = min_faktor;
                    };
// wenn das nicht der Fall ist, die beiden Zeitwerte vergleichen und bei
// einer Differenz von mehr als 100ms (3 Frames) „nicht schaffbar“ melden
              } else {
                    if( fabs( gefahrene_zeit - abgZeit ) > 0.1 ) {
                          si->RobSchafftAnfahrt[RobLogID - 1] =
                                false;
                          speed_adjust = 1.0f;
                    } else {
                          si->RobSchafftAnfahrt[RobLogID - 1] =
                                true;
                          speed_adjust = max_faktor;
                    };
              };
       };
      };
 };
```

Eine Reihe von Tests hat gezeigt, dass dieser Mechanismus die Diskrepanz zwischen der prognostizierten und gefahrenen Zeit tatsächlich reduziert, allerdings ist es ebenfalls vorgekommen, dass eine Anfahrt abgebrochen wurde, obwohl die Einschätzung der Beobachter war, dass die angefahrene Position rechtzeitig erreicht worden wäre. Dies ist auf die harten Schranken zurückzuführen, anhand derer der Anfahrtsalgorithmus die Entscheidung über die rechtzeitige

Erreichbarkeit trifft. Insbesondere bei großer Nähe zum Ziel steigt der errechnete Faktor schon bei geringer Zeitdifferenz extrem an und kann so zum Abbruch führen.
Eine dynamische Auswertung der Situation, entweder durch den Anfahrtsalgorithmus oder die Strategie, könnte hier Abhilfe schaffen.

III.3.3 Sonderfallbehandlung „Nahe Zielposition“

Wie bereits in Abbildung III.2-2 illustriert, kann eine geringe Entfernung zwischen Start- und Zielposition zu ineffektiven Pfaden führen. Daher wurde ein spezialisierter Anfahrtsalgorithmus (im Folgenden: „NaheZielpos“) entworfen und in den Haupt-Algorithmus eingebettet. Dessen Funktionsweise soll hier nur kurz erläutert und im nächsten Kapitel im Detail betrachtet werden.
Basierend auf der aktuellen Geschwindigkeit des Roboters sowie der Entfernung von der durch die Strategie übermittelte Zielposition wird entschieden, ob es sich um eine nahe Zielposition im Sinne von NaheZielpos handelt. Ist dies der Fall und befindet sich der Roboter noch nicht auf einem Pfad, wird versucht, die Situation einem der durch NaheZielpos abgedeckten Unterfall zuzuordnen und, falls dies gelingt, dessen Behandlung in den nachfolgenden Frames direkt von NaheZielpos durchgeführt.
Abbruchkriterien sind Sprünge in der Ziel- oder Roboterposition oder ein zu großes Anwachsen der Entfernung zum Ziel im Laufe der Behandlung.

III.4 Details „Nahe Zielposition“

III.4.1 Grundlage

NaheZielpos ist ein separater, spezialisierter Anfahrtsalgorithmus, der immer dann ausgeführt wird, wenn ein neuer Pfad zu einer Position innerhalb eines wenige Roboter-Durchmesser umspannenden Radius berechnet werden soll. Er greift nicht in bestehende Anfahrten ein, auch wenn in deren Ablauf eine entsprechende Situation auftreten sollte. Würde dies der Fall sein, wäre das Fahrverhalten des Roboters nicht mehr vorhersagbar.
Andererseits gilt die gleiche Restriktion für den Haupt-Algorithmus. Auch wenn die Situation „nahe Zielposition“ während einer Behandlung durch NaheZielpos nicht mehr vorhanden ist, wird die Behandlung weiter durchgeführt, bis NaheZielpos die Anfahrt als geglückt oder erfolglos beendet, oder durch die Strategie eine neue Anfahrt gestartet wird.
Wie auch der Haupt-Anfahrtsalgorithmus verfügt auch NaheZielpos über einen internen Zustand, der für eine Konsistenz über mehrere Frames hinweg sorgt, und der die Behandlung einer spezifischen Situation auch nach deren Änderung weiterführt.
Beim Entwurf von NaheZielpos wurde anfangs davon ausgegangen, dass eine genaue Anfahrt wichtiger ist als eine möglichst schnelle. Später wurde die Möglichkeit zur Opferung dieser Genauigkeit zugunsten des Einsatzes für zeitkritische Anfahrten hinzugefügt.

III.4.2 Einbettung in den Anfahrtsalgorithmus

NaheZielpos ist wie folgt in die Hauptroutine des Anfahrtsalgorithmus integriert:

- In jedem Frame wird die Entfernung des Roboters zum Zielpunkt errechnet und, falls die aktuelle Geschwindigkeit kleiner oder gleich der von NaheZielpos gefahrenen ist, die Situation als nahe Zielposition deklariert.
- Sofern eine neue Pfadberechnung anliegt oder bereits in den vorigen Frames eine Behandlung einer nahen Zielposition gestartet wurde, wird NaheZielpos aufgerufen und anhand des Rückgabewertes gespeichert, ob die Behandlung übernommen werden konnte.
- Ist dies der Fall, wird der komplette Rest der Hauptroutine übergangen und der Aufruf des Anfahrtsalgorithmus für diesen Frame beendet.

Die beschriebene Abkapselung von der normalen Anfahrtsroutine ist deshalb erforderlich, da NaheZielpos unabhängig vom Phasen-System des Algorithmus agiert (jede Phase ist für einen Teilabschnitt des S-Kurven-Pfades verantwortlich) und dieses daher unweigerlich die Entscheidung treffen würde, einen neuen Pfad zu berechnen um das „Fehlverhalten“ des Roboters zu unterbinden.
Es hat sich allerdings als sinnvoll erwiesen, auch außerhalb von NaheZielpos dessen Abbruchbedingungen zu überprüfen, um zu verhindern, dass der Roboter fälschlicherweise in der Sonderbehandlung hängen bleibt. Insbesondere wurde dies für ein großes Anwachsen der Entfernung zwischen Roboter und Ziel implementiert, da dieses Kriterium leicht zu überprüfen ist.

III.4.3 Eine erste Implementierung

Die ursprüngliche Idee war es, die Sonderfallbehandlung „nahe Zielposition“ so einfach wie möglich zu halten. Es wurde daher nur der Fall „Roboter "hinter" Ziel“ bedacht, da davon ausgegangen wurde, dass der Roboter sonst versehentlich „durch“ einen ruhenden Ball gelenkt werden könnte.
Folgende Unterfälle (Abbildung III.4-1) wurden implementiert:

1. Ausrichtung des Roboters ≈ Zielausrichtung und die Gerade durch den Zielpunkt und die Gerade durch die Mitte des Roboters haben einen Abstand ≤ dem Radius des Roboters. Sind diese zwei Bedingungen erfüllt, so kann der Roboter direkt geradeaus fahren, und wird den Zielpunkt zumindest mit einem Teil seiner zum Ziel hin gerichteten Seite treffen.
2. Ausrichtung des Roboters ≠ Zielausrichtung, aber eine Gerade durch den Roboter, die parallel zur Zielausrichtung ist, hat einen Abstand zur Mitte des Roboters ≤ dem Radius des Roboters. In diesem Fall reicht es aus, den Roboter auf der Stelle zu drehen, bis seine Ausrichtung der Zielausrichtung entspricht, und anschließend mithilfe von Unterfall I geradeaus zu fahren.
3. Der Abstand der in Unterfall II konstruierten Geraden zum Roboter ist zu groß, um das Ziel noch mit der Roboterkante treffen zu können. Um diese Situation zu lösen, muss der Roboter so gedreht werden, dass er anschließend einen Kreis fahren kann, der den Zielpunkt mit einer Tangente entsprechend der Zielausrichtung erreicht. Hierzu wird die bereits unter

„Neuberechnung von Pfad-Teilsegmenten“ beschriebene Routine zur Berechnung eines verbindenden Kreises durch zwei Punkte verwendet. Ist die nötige tangentiale Ausrichtung erreicht, werden, ebenfalls durch diesen Unterfall, Befehle zur Fahrt eines Kreises mit dem berechneten Radius übermittelt.

Sämtliche denkbaren Situationen, die nicht von den beschriebenen Fällen abgedeckt werden, führen zu einer Behandlung durch den normalen Anfahrtsalgorithmus.

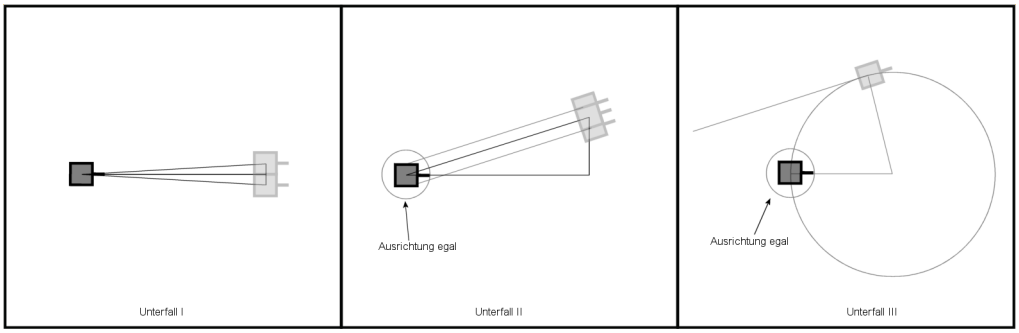


Abbildung III.4-1: NaheZielpos Unterfälle

Da diese Fälle ein Grundgerüst für die spätere Erweiterung von NaheZielpos bieten, sollen Teile ihrer ursprüngliche Implementierung hier aufgeführt werden.

Allerdings werden aus Gründen der Übersicht werden bei Ausrichtungs- und Positionsabfragen Toleranzen hier nicht aufgeführt. Ferner muss angemerkt werden, dass das der Strategie – und damit dem Anfahrtsalgorithmus – zugrundeliegende Koordinatensystem entgegen der Intuition eine nach unten gerichtete Y-Achse aufweist, was besonders im Falle von trigonometrischen Berechnungen unbedingt zu beachten ist.

- Überprüfung, ob Roboter "hinter" dem Ziel ist:

  Diese Entscheidung wurde in die Betrachtung von Quadranten um den Roboter herum aufgeteilt (siehe Abbildung III.4-2).

```
if( // links unterhalb Ziel
 ( ( sollA >= 0 ) && ( sollA < M_PI_2 ) &&
              ( istX < sollX ) && ( istY >= sollY ) ) ||
       // rechts unterhalb Ziel
       ( ( sollA >= M_PI_2 ) && ( sollA < M_PI ) &&
             ( istX >= sollX ) && ( istY > sollY ) ) ||
       // Robo rechts oberhalb Ziel
       ( ( sollA >= M_PI ) && ( sollA < 3 * M_PI_2 ) &&
             ( istX > sollX ) && ( istY <= sollY ) ) ||
       // Robo links oberhalb Ziel
       ( ( sollA >= 3 * M_PI_2 ) && ( sollA < 2 * M_PI ) &&
             ( istX <= sollX ) && ( istY < sollY ) )
       ) {
```

```
// Behandlung der genannten Unterfälle
};
```

- Die Prüfung, ob der Roboter schon auf dem Zielvektor steht, wird mithilfe des Winkels zwischen Roboter und Zielposition durchgeführt (siehe Abbildung III.4-3).

```
// Unterfall I: Ausrichtung Robo == Ausrichtung Ziel und Gerade durch
// Ziel und gerade durch RoboMitte haben höchstens RoboterRadius Abstand
if( ( sollA == istA ) &&
        ( fabs( atan2_pos( istY - sollY, sollX - istX ) - sollA ) <
              asin( k->RoboterRadius ) )
) {
        // Roboter kann einfach geradeaus fahren
        rob->setSollGeschwindigkeit( sollV );
        rob->setDelta( 0 );
        rob->setModus( FreundRoboter::MODUS_DELTADREH );
        return true;
       };
```

Für Unterfall II gilt Analoges.

- Die Berechnung des Kreises durch die aktuelle Position des Roboters und den Zielpunkt entspricht der bereits in Kapitel III.3.1 gezeigten Implementierung. Die Bestimmung Ausrichtung auf die Start-Tangente wird wie folgt durchgeführt.

```
// Tangentenrichtung des Roboters berechnen
       roboAusrichtung = atan2_pos( kreisMitteY - istY, istX -
       kreisMitteX ) +
              M_PI_2;
       if( roboAusrichtung > 2 * M_PI ) {
        roboAusrichtung -= 2 * M_PI;
       };

// Fahrtrichtung abhängig vom Quadranten in dem der Roboter ist
       // 1. Quadrant
       if( ( sollA >= M_PI ) && ( sollA < 3 * M_PI_2 ) ) {

//im Uhrzeigersinn fahren falls Kreismittelpunkt links von
// Mittelsenkrechter
        if( kreisMitteX <= mittelSenkrechteX ) {

       // nicht nach oben drehen
              if( ( roboAusrichtung >= 0 ) && ( roboAusrichtung < M_PI )
```

```
      ) {
                  roboAusrichtung += M_PI;
            };
            uhrzeigersinn = true;
       } else {
      // sonst gegen den Uhrzeigersinn
      // nicht nach rechts drehen
            if( (roboAusrichtung >= 0 ) &&
       ( roboAusrichtung < M_PI_2 )
      ) {
                  roboAusrichtung += M_PI;
            };
            if( ( roboAusrichtung >= 3 * M_PI_2 ) &&
       ( roboAusrichtung <= 2 * M_PI )
      ) {
                  roboAusrichtung -= M_PI;
            };
       }
// analog für 2. 3. und 4. Quadrant
      } else if( ... ) { ... }
```

- Es fehlt noch das eigentliche Ausrichten auf die Tangente und die Kreisfahrt.

```
// Roboter um Winkeldifferenz zur Tangente drehen
      if( istA > roboAusrichtung ) {

// um kleineren Winkel drehen
       if( ( istA - roboAusrichtung ) >= M_PI ) {
            rob->setSollGeschwindigkeit( 0 );
            rob->setDelta( -1 * ( istA - roboAusrichtung ) );
            rob->setModus( FreundRoboter::MODUS_DELTADREH );
       } else {
            rob->setSollGeschwindigkeit( 0 );
            rob->setDelta( ( 2 * M_PI ) - ( istA - roboAusrichtung ) )
      );
            rob->setModus( FreundRoboter::MODUS_DELTADREH );
       }
      } else {

// um kleineren Winkel drehen
       if( ( roboAusrichtung - istA ) >= M_PI ) {
            rob->setSollGeschwindigkeit( 0 );
            rob->setDelta( -1 * ( roboAusrichtung - istA ) );
            rob->setModus( FreundRoboter::MODUS_DELTADREH );
```

```
        } else {
                rob->setSollGeschwindigkeit( 0 );
                rob->setDelta( ( 2 * M_PI ) - ( roboAusrichtung - istA ) );
                rob->setModus( FreundRoboter::MODUS_DELTADREH );
        }
      };

      // Kreis fahren
      R2RadV( kreisRadius, 0, uhrzeigersinn, rad_links, rad_rechts );
      rob->setModus( FreundRoboter::MODUS_RADGESCHWINDIGKEITEN );
      rob->setRadgeschwindigkeiten( rad_l ,rad_r );
      return true;
```

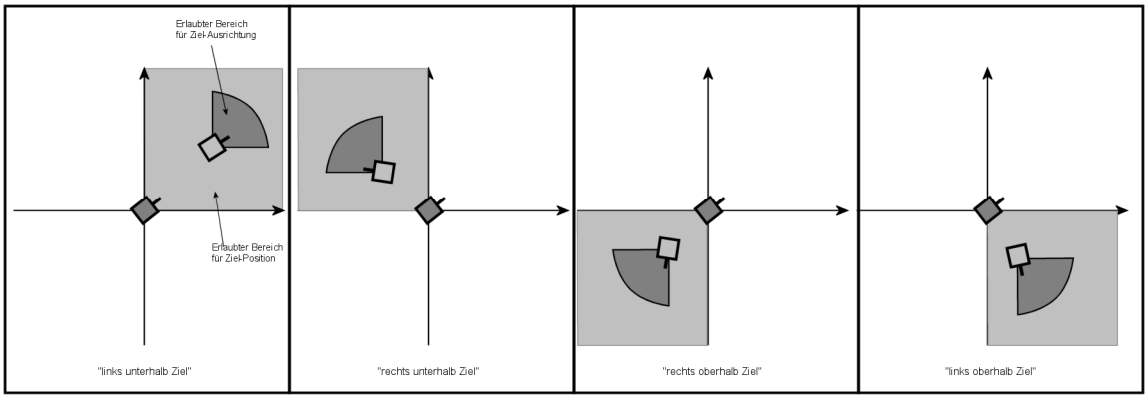


Abbildung III.4-2: Von "NaheZielpos" ursprünglich behandelte Situationen

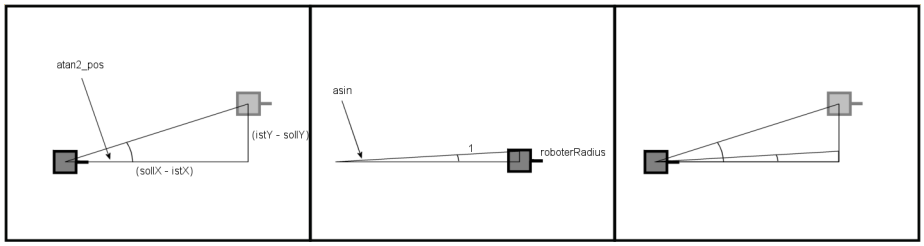


Abbildung III.4-3: Trefferberechnung

III.4.4 Erweiterte Funktionalität

Obwohl die beschriebene erste Implementierung von NaheZielpos bereits für einfache Anfahrten ausreichte, wurden viele Einsatzmöglichkeiten durch die diversen Beschränkungen und Vereinfachungen verhindert.

Aus diesem Grund wurde die Quadranten-basierte Betrachtung von „hinter Ziel“ in ein Ausschlussverfahren umgewandelt – lediglich in Fällen, bei denen die Roboterposition bereits im von der Zielausrichtung abgedeckten Bereich liegen, wird die Behandlung von NaheZielpos zurück an den

Haupt-Algorithmus gegeben. Dies wurde entschieden, damit der Roboter bei der Anfahrt nicht „durch“ den Ball gelenkt wird, sollte dieser vor der Zielposition liegen. Für alle anderen Situationen wurden entsprechende Behandlungsroutinen implementiert.

- Roboter steht bereits mehr oder weniger auf dem Ziel, und nur die Ausrichtung ist falsch: Es wird auf der Stelle gedreht.
- Roboter steht ungefähr parallel zur Zielausrichtung oder die vorgegebene Zielgeschwindigkeit ist Null:

  Es wird davon ausgegangen, dass kein Ball getroffen werden soll (sonst gäbe es eine Zielgeschwindigkeit), daher ist die Richtung, aus der der Roboter das Ziel trifft, egal. Er wird auf der Stelle gedreht, bis seine Ausrichtung auf die Zielposition zeigt. Dann wird geradeaus gefahren, und anschließend auf die Zielausrichtung gedreht.
- Für annähernd auf der Ausrichtung des Roboters liegende Ziele tritt Unterfall I der ursprünglichen Implementierung in Kraft. Während der Anfahrt wird versucht, die Fahrtrichtung so zu korrigieren, auszurichten, dass das Ziel genau mit der Mitte des Roboters getroffen wird.
- Unterfall II besteht weiterhin. Er wurde um eine Überwachung der Drehung erweitert, um die Ausrichtung des Roboters vor der Übergabe an Unterfall I gegebenenfalls noch einmal zu korrigieren. Diese Überwachung ist solange erforderlich, bis ein Funkbefehl zur Verfügung gestellt wird, bei dem ein Drehwinkel und eine direkt im Anschluss an die Drehung zu fahrende Geschwindigkeit übermittelt werden kann.
- Bei von Unterfall III gesteuerten Kreisfahrten konnte es nach der Relaxierung der Behandlungsbedingungen von NaheZielpos vorkommen, dass der Roboter einen fast vollständigen Kreis fuhr, um die Zielposition mit der korrekten Ausrichtung zu erreichen. Aus diesem Grund wurde eine Beschränkung des auf dem Kreisbogen zu fahrenden Winkels eingeführt. Wird diese überschritten, so wird stattdessen eine geradlinige Anfahrt auf eine Position hinter dem Ziel durchgeführt wird, um so die Zielposition nach einer Drehung anschließend korrekt zu erreichen.

Um die eingangs bereits erwähnte Unterstützung zeitkritischer Anfahrten zu erreichen, wurde ein zusätzlicher Parameter eingeführt, mithilfe dessen die Strategie NaheZielpos anweisen kann, keine „kleinen“ Drehungen des Roboters auszuführen. Diese sind insofern kritisch, da sie einerseits lange in der Durchführung brauchen und andererseits ihre Erkennung durch die Deckenkamera vergleichsweise ungenau ist. Ihre Verhinderung führt daher zu einer sehr viel schnelleren Reaktion vonseiten des Roboters. Eine weitere Verbesserung ist zu erwarten, falls der oben erwähnte „drehe und fahre“ Befehl Realität wird.

Ferner wird bei Setzen dieses Parameters die Ausrichtungskorrektur während des geradeaus Fahrens auf vergleichsweise große Winkel beschränkt, um bei Änderungen der Zielposition nicht darauf

warten zu müssen, dass der Roboter stehen bleibt und seine Ausrichtung somit bekannt ist, sondern direkt die neue Behandlung starten zu können.
Diese Erweiterung wurde insbesondere in Hinblick auf den Torwart vorgenommen, dessen Zielpositionen häufig nahe bei ihm liegen, und bei dem ein genaues Anfahren der Position weniger wichtig ist als deren zügiges Erreichen.

III.4.5 Bestehende Probleme

In der Praxis hat sich gezeigt, dass die abgedeckten Situationen mit nahen Zielpositionen für allgemeine Anfahrten ausreichend sind. Problematisch sind allerdings spezielle Rollen wie der Torwart oder auch Verteidiger, da für diese gesetzte Zielpositionen häufig springen.
Hierbei ergibt sich das Problem, dass die daraus resultierenden unterschiedlichen Situations-Klassifikationen zu teils widersprüchlichen Befehlssequenzen führen. Dies führt im Falle des Torwarts zu gelegentlichen Kollisionen mit den Begrenzungen des Tores; Kollisionen aber werden von der bisherigen Implementierung nicht berücksichtigt, da Positionen nahe den Banden im allgemeinen Fall eine Sonderbehandlung durch die Strategie erhalten und dem Anfahrtsalgorithmus als solches nicht übergeben werden.
Ferner können wiederholt springende Zielpositionen ebenfalls dazu führen, dass der Roboter durch widersprüchliche Korrekturbefehle bei einer an sich geraden Anfahrt vom Pfad abkommt, und so sein Ziel verfehlt. Um diesem Verhalten entgegen zu wirken, müsste der Roboter vor Beginn einer neuen Anfahrt gestoppt werden, was aber im Spielbetrieb aufgrund des damit verbundenen Zeitverlustes in der Regel nicht akzeptabel ist.

III.5 Ausblick/Probleme

Das wohl größte bestehende Problem des Anfahrtsalgorithmus ist die Tatsache, dass ein gutes Verhalten der Roboter eine Anpassung an deren Eigenschaften und Fahrverhalten erfordert. Hierzu stand nur ein Prototyp der neuen Robotergeneration zur Verfügung, dessen Verhalten nicht repräsentativ für das Verhalten der später gelieferten Roboter ist. Insofern besteht insbesondere in Hinblick auf das Beschleunigungs- und Bremsverhalten noch Justier-Bedarf.
Ferner bleibt es ein Risiko, dass der Anfahrtsalgorithmus als Ziel einen Punkt bekommt, auf den der Roboter fahren soll. Ist dieser Punkt ein Schnittpunkt mit dem Ballvektor und wird die Ankunft des Roboters aus irgendeinem Grund verzögert, so wird der Ball verfehlt.
Dem wurde insofern entgegen gewirkt, dass die Strategie nun zuerst einen Punkt auf dem Ballvektor ≠ der voraussichtlichen Ballposition anfahren und den Roboter anschließend entlang des Vektors auf den Ball treffen lassen kann. Eine direkte Unterstützung dieser Methodik durch den Anfahrtsalgorithmus wäre denkbar und sicherlich eine sinnvolle Erweiterung. Ebenso könnte dies auf allgemeine Wegpunkte ausgeweitet werden.

IV. Gruppe Bildverarbeitung

Altay Cebe,
Christian Kinzel,
Christian Kolek

IV.1 Entzerrer

Bilder, die durch ein Kamerasystem aufgenommen werden, unterliegen optischen Verzerrungen. Eine Positionsbestimmung von Objekten auf Basis der Kamerabilder ist daher nur möglich, wenn diese Bildfehler in der Software berücksichtigt und kompensiert werden. Für diese Aufgaben ist das Modul Entzerrer zuständig.

Die zu Beginn der Projektgruppe vorgefundene Bildentzerrung arbeitete zu ungenau und war zu unflexibel. Wesentliche Kritikpunkte dabei waren:

- Die Kamera musste exakt über dem Mittelpunkt des Spielfeldes angebracht werden mit der optischen Achse genau senkrecht zur Spielfeldebene.
- Roboter, welche auf dem Spielfeld eine gerade Linie abfuhren, fuhren in der Bildverarbeitung parabelförmige Kreisbögen.

Beide Fehler lassen sich auf die folgenden beiden Effekte zurückführen, die auftreten, wenn ein Kamerabild aufgenommen wird:

- Fischaugeneffekt
  Die Optik einer Kamera ist nicht perfekt. Lichtstrahlen, werden je nach Einfallwinkel auf die Linse anders gebrochen, und treffen so nicht an der exakt richtigen Stelle auf den CCD-Chip der Kamera. Entlang der optischen Achse arbeitet die Linse ideal, je größer die Winkelabweichung wird, desto größer wird der Verschiebeeffekt. Das Ergebnis ist, dass Geraden von der Linse parabelförmig auf den CCD-Chip geworfen werden.

- Parallaxen-Effekt
  Wenn mit einer Kamera eine Ebene aufgenommen wird, kann jedem Pixel ein eindeutiges Flächenstück auf der Ebene zugeordnet werden. Wenn jedoch auf der Ebene Objekte stehen, die aus dieser hinausragen, wird diese Zuordnung schwieriger. Die Höhe der Objekte sorgt dafür, dass sie optisch im Bild nach außen wandern. Wird die Zuordnung Pixel – Fläche auf

der Spielfeldebene verwendet, werden die Objekte als weiter außen auf der Ebene erkannt als sie tatsächlich stehen.

IV.1.1 Funktionsweise des Entzerrers

Der Entzerrer entfernt in einem Mehrschrittansatz die oben erwähnten optischen Verzerrungen und ermöglicht eine exakte Positionsbestimmung der auf dem Spielfeld stehenden Objekte.

Der zeitliche Ablauf ist dabei wie folgt:

- Entfernung des Fischaugeneffektes
- Umrechnung der Pixelkoordinaten der Objekte in Angaben in mm auf dem Spielfeld
- Korrektur des Parallaxeneffekts

IV.1.1.1 Fischaugenkorrektur

Die verwendete Fischaugenkorrektur basiert auf dem Ansatz von Zhang. [ZHANG]
In Zhangs Ansatz wird für die Kamera während einer einmalig durchzuführenden Kalibration ein Ausgleichspolynom errechnet, welches den von der Linse verursachten Fischaugeneffekt modelliert.

Dieses Polynom basiert auf folgenden Parametern:
Brennweiten fx und fy der Linse in horizontaler sowie vertikaler Richtung

- Durchstoßpunkt der optischen Achse durch die Bildebene der Kamera *nx* und *ny*
- Den Vorfaktoren des Ausgleichspolynoms *k1* und *k2*

Die exakten Werte dieser Parameter werden während der Kalibration ermittelt und können während der Entzerrung als bekannt vorausgesetzt werden. Der Ablauf einer Kalibration wird im Abschnitt Kamerakalibration weiter unten erläutert.

Der Ansatz von Zhang hat einen Nachteil: Er kann nur aus einem unverzerrten Bild ein verzerrtes Bild errechnen. Das heißt, wenn man in das Ausgleichspolynom eine Pixelkoordinate (u, v) einsetzt, wird ein Korrekturterm errechnet, welcher aufaddiert auf (u, v) die Koordinaten des Pixels in einem verzerrten Bild liefert. Zhang's Ansatz arbeitet also genau verkehrt herum.

```
Algorithmus: AddRadialLensDistortion
void addRadialLensDistortion(point2d &ideal, point2d &distorted)
{
 double xDiff = (ideal.x - nx);
```

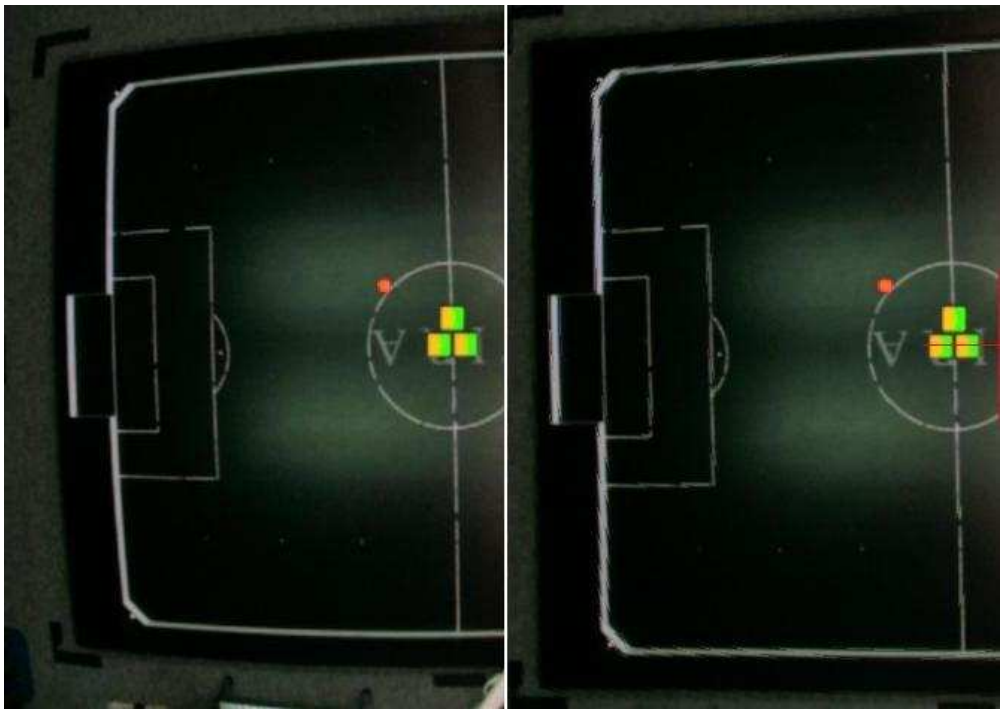
```
 double yDiff = (ideal.y - ny);

   double rSquared = (xDiff * xDiff)/(ep.fx * ep.fx) + (yDiff *
     yDiff)/(ep.fy * ep.fy);

 distorted.x = ideal.x + xDiff * (k1 * rSquared + k2 * rSquared *
     rSquared);
    distorted.y = ideal.y + yDiff * (k1 * rSquared + k2 * rSquared *
     rSquared);}
```

Eine Invertierung des Algorithmus ist nur mittels eines numerischen Iterationsverfahrens möglich.

Dabei wird in jeder Iteration die Koordinate des Pixels im unverzerrten Bild weiter angenähert. Sobald der Fehler klein genug ist, wird die Iteration abgebrochen und der letzte Schätzwert zurückgegeben.

```
Algorithmus: RemoveRadialLensDistortion
void removeRadialLensDistortion(point2d &distorted, point2d &ideal)
{
    //L muss in ]-1, 0[ liegen, gibt an, wo der nächste Suchpunkt
    //liegt. MAXITERATIONS gibt an, nach wie vielen Iterationen
    //abgebrochen wird.
    double L             = -0.5;
    int    MAXITERATIONS = 128;

    point2d old;
    old.x = distorted.x;
    old.y = distorted.y;

    point2d res;
    res.x = 0;
    res.y = 0;

    point2d err;
    err.x = 0;
    err.y = 0;

    for (int i=0; i < MAXITERATIONS; i++)
    {
      addRadialLensDistortion(old,res);
      err.x = res.x - distorted.x;
      err.y = res.y - distorted.y;

      if (err.x*err.x +err.y*err.y <= 0.01)
      {
```

```
            ideal.x = old.x;
            ideal.y = old.y;
            return;
        }
        old.x = old.x + L * err.x;
        old.y = old.y + L * err.y;
    }
    ideal.x = -1;
    ideal.y = -1;
}
```



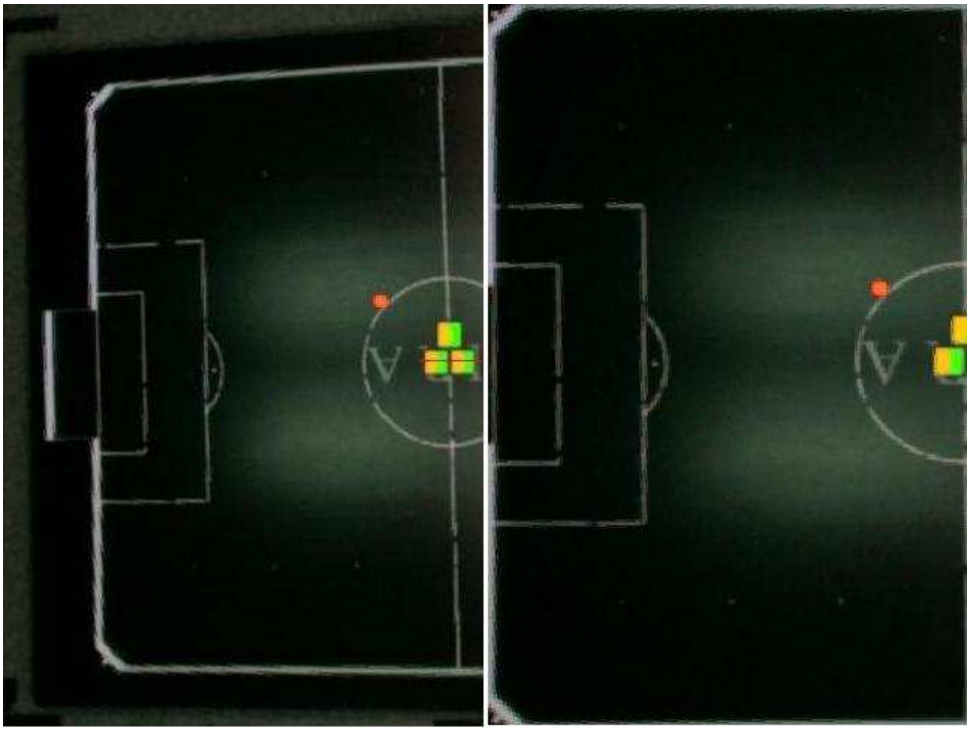
Vor und nach Fischaugenkorrektur

IV.1.1.2 Mapping von Pixelkoordinaten in Weltkoordinaten

In diesem Schritt wird jedem Pixel (u, v) eine Position (x, y) auf dem Spielfeld zugeordnet. Um dies zu erreichen, muss die perspektivische Projektion der Kamera umgekehrt werden.

Da ein Kamerabild zweidimensional ist, die aufgenommenen Objekte jedoch dreidimensional sind, ist im Allgemeinen eine eindeutige Zuordnung nicht möglich. Da jedoch alle für uns relevanten Punkte auf einer Ebene, dem Spielfeld, liegen, können wir dieses Problem trotzdem exakt lösen.

Es wird in homogenen Koordinaten gerechnet, das heißt, die 2D-Koordinaten (u, v) sowie (x, y) werden mit einer zusätzlichen z-Koordinate erweitert.



Kamerabild vor und nach Mapping

Die perspektivische Projektion der Kamera lässt sich nun ausdrücken als:

(u', v', w') = M * (x', y', z'); (u, v) = (u' / w', v' /w')

M ist dabei eine 3 x 3 Matrix, welche die perspektivische Transformation der Kamera ausdrückt.

Wenn die Matrix M bekannt ist, kann durch eine Invertierung dieser die perspektivische Projektion rückgängig gemacht werden und jedem Pixel eindeutig eine Koordinate auf dem Spielfeld zugewiesen werden.

(x', y', z') = M-1 * (u', v', w'); (x, y) = (x' / z', y' /z')

Es gilt also, die Matrix M zu bestimmen.
Dazu werden vier Korrespondenzen benötigt. Eine Korrespondenz ist ein Punkt, dessen Position sowohl in Bildschirmkoordinaten (u, v) als auch in Spielfeldkoordinaten (x, y) bekannt ist. Es bieten sich im Robosoccer-Projekt die vier Eckpunkte des Spielfelds an.

Aus diesen vier Korrespondenzen lässt sich die Matrix M bis auf einen Skalierungsfaktor eindeutig bestimmen. Dieser Skalierungsfaktor ist jedoch unerheblich, er gibt nur an, wie weit die Kamera von

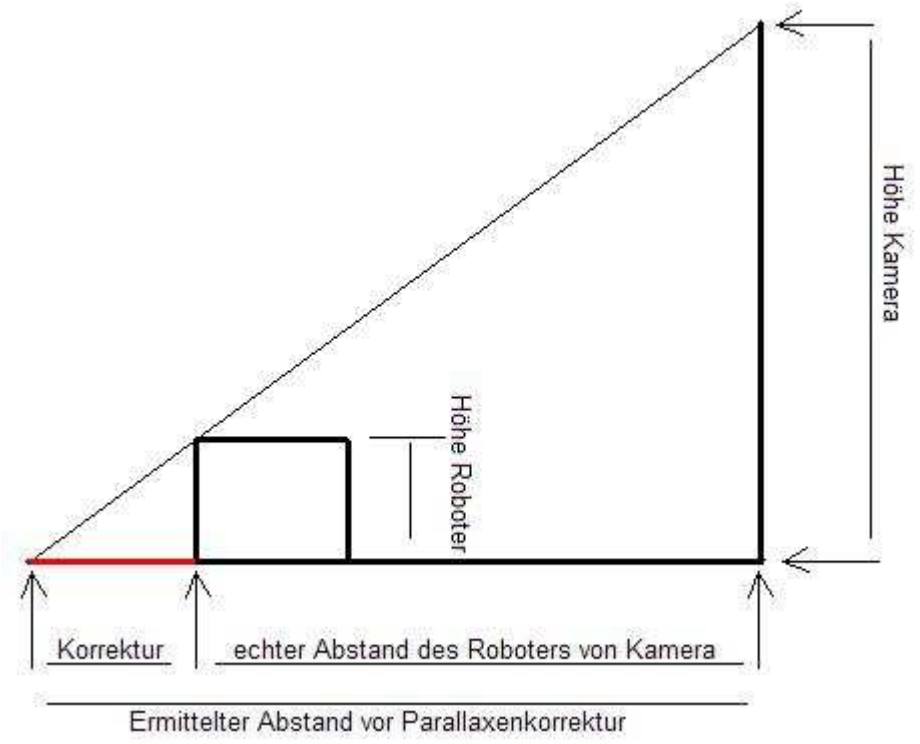
der Spielfeldebene entfernt ist. Im zweiten Schritt der Rechnung, der perspektivischen Division, wird er herausgerechnet.

Mathematische Details zur Berechnung der Projektionsmatrix finden sich im Paper [TEXFUN, Abschnitt 5] und werden hier nicht besprochen.

Der Mapping Algorithmus ergibt sich also als:

```
Algorithmus: UndistortedToWorld
void mapPoint(point2d &p, mat3x3d MappingMatrix, point2d &result)
{
     point2dh temp1, temp2;

     temp1.x = p.x;
     temp1.y = p.y;
     temp1.w = 1;

     temp2 = temp1 * MappingMatrix;
     result.x = temp2.x/temp2.w;
     result.y = temp2.y/temp2.w;
}
```

IV.1.1.3 Parallaxenkorrektur



Vorgehensweise der Parallaxenkorrektur

Der Parallaxeneffekt kann, wie aus dem Bild ersichtlich wird, mit Hilfe der Strahlensätze entfernt werden. Um den Korrekturterm für den Parallaxeneffekt zu berechnen, werden folgende Informationen benötigt:

- Position der Kamera in mm (x, y, z)
- Position des Objekts in Spielfeldkoordinaten (u, v)
- Höhe des Objekts in mm (h)

Die Parallaxenkorrektur kann erst durchgeführt werden, wenn die gefundenen Objekte bereits klassifiziert sind, da erst dann die Höhe des Objekts bekannt ist.

Der Korrekturterm ergibt sich unter Anwendung der Strahlensätze als:

Korrektur = Höhe des Roboters / Höhe der Kamera * (Objektposition – Kameraposition)

Daraus lässt sich folgender Algorithmus für die Parallaxenkorrektur ableiten:

```
Algorithmus: ParallaxCorrect
void parallaxCorrect(double &xDistPose, double &yDistPose, double
       &xUndistPose, double &yUndistPose, double &height)
{
     double factor = height/camPoseZ;

     xUndistPose = xDistPose - factor * (xDistPose - camPoseZ);
     yUndistPose = yDistPose - factor * (yDistPose - camPoseY);

}
```

IV.1.1.4 Optimierung

Die Schritte Fischaugenkorrektur und Mapping sind für jeden Pixel bei nicht bewegter Kamera und nicht bewegtem Spielfeld in jedem Kamerabild identisch und werden daher bei der Initialisierung einmal vorberechnet und in einer Look-Up-Table gespeichert.

IV.1.1.5 Berechnung der Fläche eines Pixels

Durch den Fischaugeneffekt und einer eventuell nicht senkrecht zur Spielfeldebene aufgehängten Kamera überdecken die einzelnen Pixel im Kamerabild nicht exakt die gleiche Fläche auf dem Spielfeld. Bei der Blobgütebewertung wird jedoch als Kriterium die Fläche des Blobs genutzt, und muss daher relativ exakt berechnet werden.

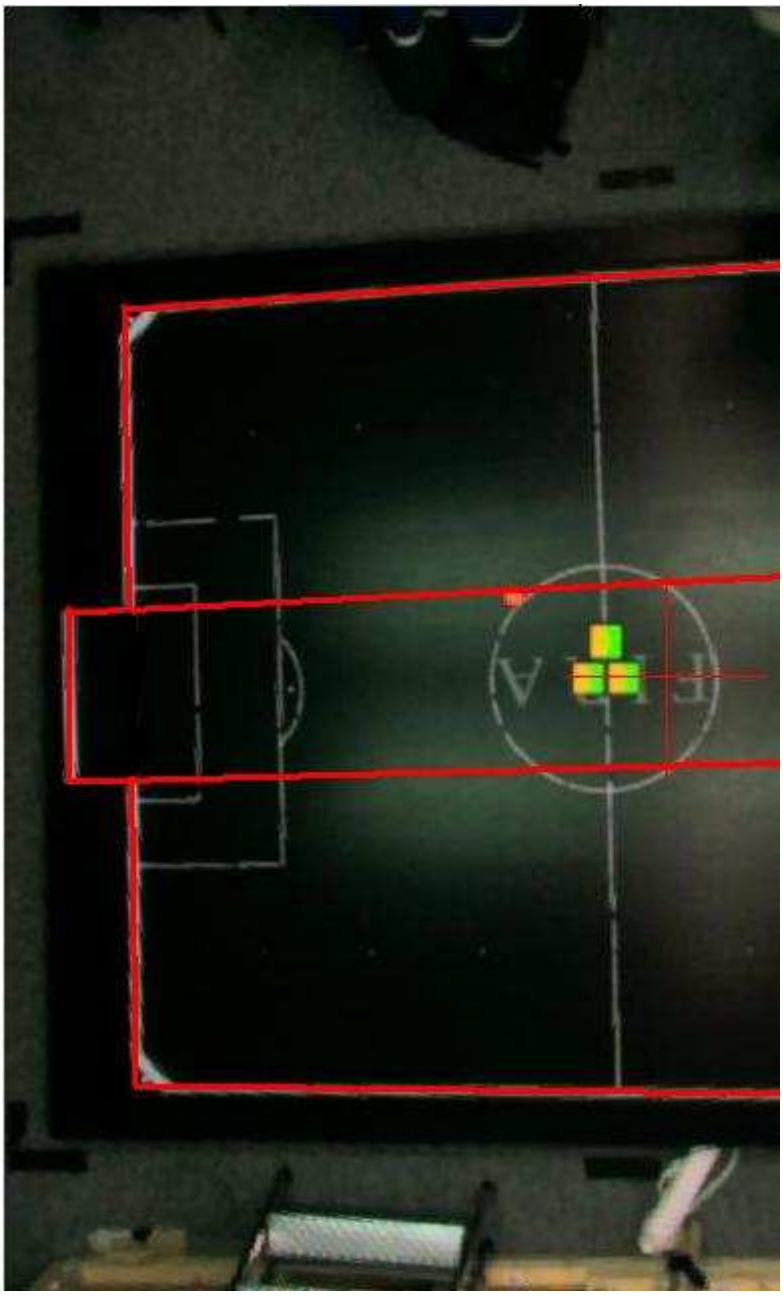
Der Entzerrer berechnet für jeden Pixel seine Größe in mm^2, indem er die folgende Formel anwendet:

```
Algorithmus: calcMMPerPixel
int calcMMPerPixel(int x, int y)
{
     point2d current, undist;
     point2d topLeft, bottomRight;

     double distX, distY;

     current.x = x-0.5;
     current.y = y-0.5;

     removeRadialLensDistortion(current, undist);
     mapPoint(undist, undistortedToWorld, topLeft);

     current.x = x + 0.5;
```

```
    current.y = y + 0.5;

    removeRadialLensDistortion(current, undist);
    mapPoint(undist, undistortedToWorld, bottomRight);

    distX = fabs(topLeft.x - bottomRight.x);
    distY = fabs(topLeft.y - bottomRight.y);

    return (int)(distX * distY);
}
```

IV.1.1.6 Bestimmung, ob ein Objekt noch auf dem Spielfeld ist

Der Entzerrer überprüft auch, ob ein Objekt, welches gefunden wurde, überhaupt auf dem Spielfeld steht und nicht hinter den Spielfeldbanden. Dazu wird das Spielfeld in drei Rechtecke aufgeteilt:



Falls sich eine Objektposition außerhalb dieser Rechtecke befindet, wird es als irrelevant markiert. Diese Funktionalität wird durch den BlobFinder aufgerufen.

IV.1.2 Kalibration einer Kamera in MatLab

Für die Kalibration wurde die kostenlos für MatLab erhältliche Camera Calibration Toolbox [http://www.vision.caltech.edu/bouguetj/calib_doc/] verwendet.

Jede Kombination aus Kamera und Objektiv muss genau einmal kalibriert werden.

Für die Kalibration wird ein großes Schachbrettmuster benötigt, welches auf einer möglichst exakt ebenen Fläche aufgeklebt ist.

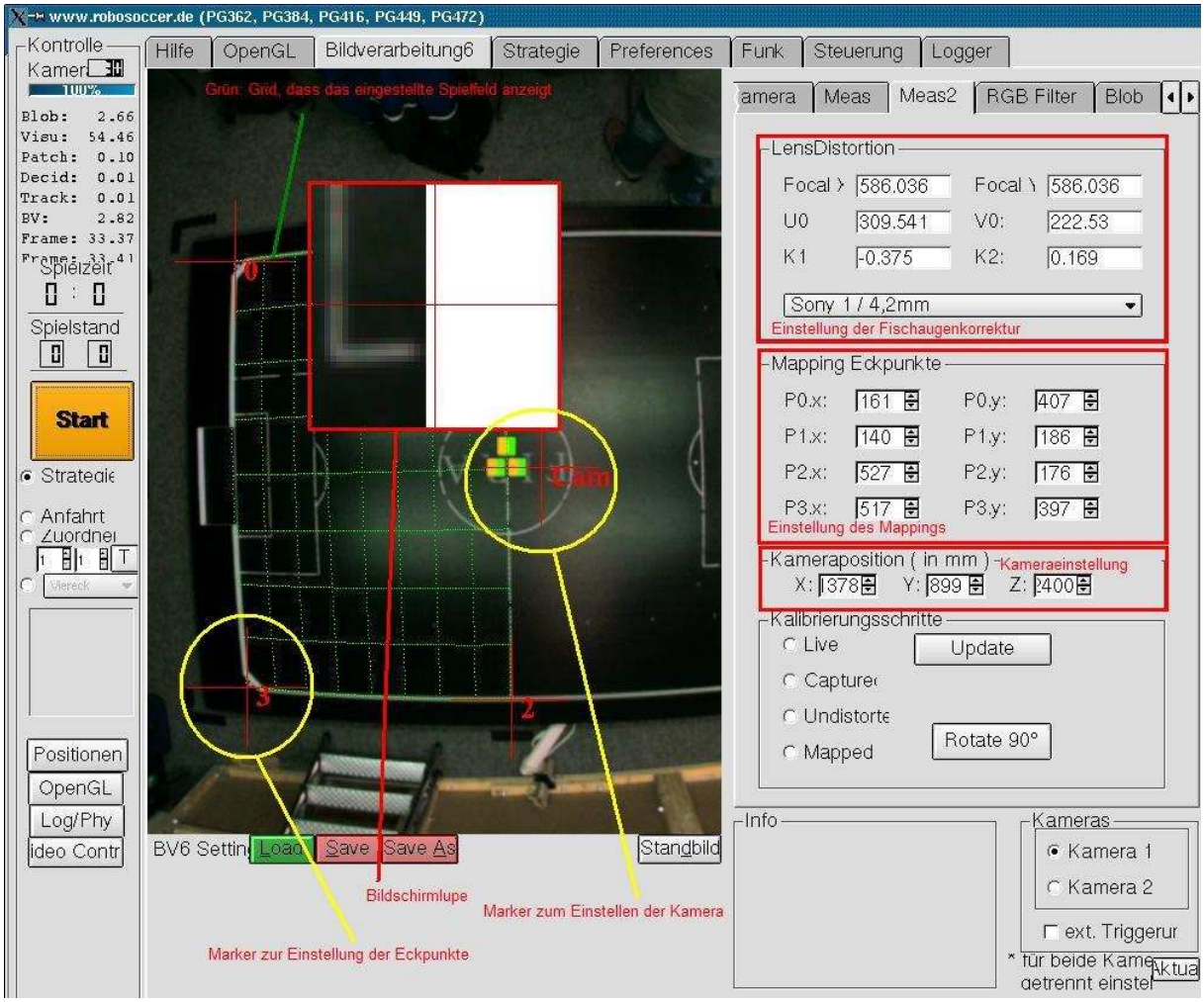
Eine Kalibration läuft prinzipiell wie folgt ab:

Mit der zu kalibrierenden Kamera werden 15 – 20 Fotos aus unterschiedlichen Blickwinkeln von dem Schachbrett geschossen.
In MatLab wird die Camera Calibration Toolbox gestartet
Die Bilder werden eingelesen
Auf jedem Kamerabild wird ein rechteckiger Bereich des Schachbrett markiert.
Die Kalibration wird durchgeführt.
Im Ausgabefenster von MatLab werden die gesuchten Werte ausgegeben.

Eine detaillierte Anleitung, wie die Kalibration durchzuführen ist, ist unter [http://www.vision.caltech.edu/bouguetj/calib_doc/] zu finden.

IV.1.3 Einstellung des Entzerrers in der GUI

Der Entzerrer kann in der GUI des Robosoccer-Programms auf dem Tab MEAS2 für jede Kamera einzeln eingestellt werden.



IV.1.3.1 Einstellung des Spielfelds

IV.1.3.1.1 Einstellung im 1-Kamera-Ansatz

Wird nur eine Kamera verwendet, sollte das gesamte Spielfeld im Visualizer zu sehen sein, die Tore sind oben und unten im Bild.

Die Einstellung erfolgt nun in folgenden Schritten:

- Auswahl der Kamera aus dem Drop-Down-Feld im Bereich Fischaugenkorrektur. Die sechs Werte für die Fischaugenkorrektur werden automatisch gesetzt.

- Die vier durchnummerierten Marker unter Zuhilfenahme der Bildschirmlupe auf die jeweils nächste Spielfeldecke ziehen.

- Update klicken. Das grüne Gitternetz sollte nun genau über die Bandenränder laufen.

- Wenn die Linien in den Ecken am Bandenrand korrekt sind jedoch in der Mitte des Spielfelds Abweichungen zu erkennen sind, wurde die falsche Kamera im Drop-Down-Feld Kamera gewählt.
- Wenn die Linien in den Ecken nicht über die Bandenränder laufen, müssen die Eckpunkte korrigiert werden.

IV.1.3.1.2 Einstellung im 2-Kamera-Ansatz

Das Standardsetup im 2-Kamera-Ansatz ist folgendes:

- Kamera 1: Linkes Tor, links im Bild
- Kamera 2: Rechtes Tor, rechts im Bild

Falls das Setup so ist, wird das Spielfeld wie im 1-Kamera-Ansatz eingestellt, mit einem Unterschied: Bevor Update geklickt wird, muss genau einmal die Schaltfläche „90 Grad drehen“ geklickt werden bei jeder Kamera.

Durch eine falsche Initialisierung kann es jedoch auch zu folgender Situation kommen:

- Kamera 1: Rechtes Tor, rechts im Bild
- Kamera 2: Linkes Tor, links im Bild

Die Einstellung ist jetzt ebenfalls genauso wie im 1-Kamera-Ansatz, jedoch muss vor klicken von „Update“ die Schaltfläche „90-Grad-Drehen“ genau dreimal geklickt werden.

Jedes Mal, wenn „90-Grad-drehen“ geklickt wird, ändern sich die Nummern an den Kreuzen.

IV.1.3.2 Einstellung der Kamera

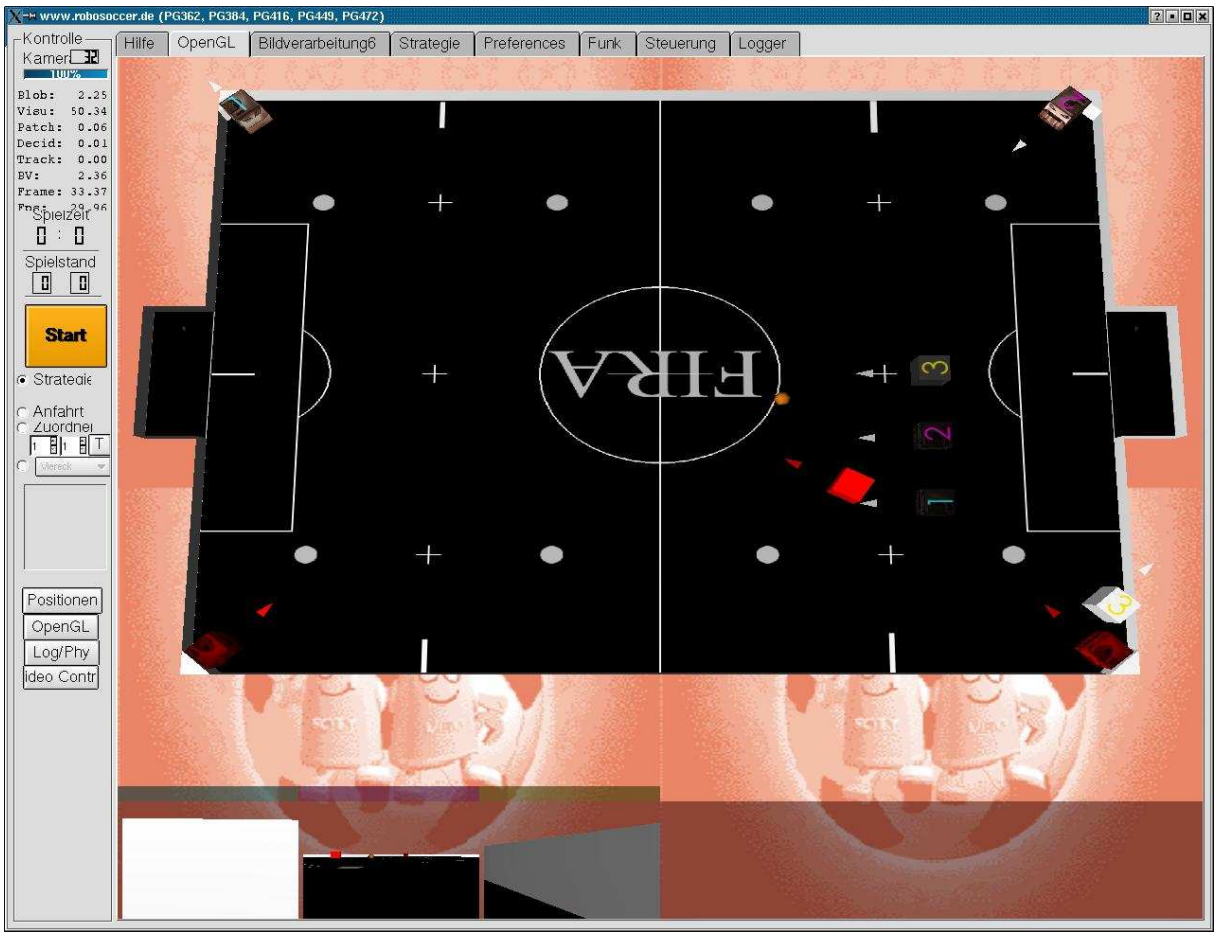
Um die Kamera korrekt einzustellen, muss ein Lot von der Kamera auf die Spielfeldebene gefällt werden. Dies geschieht am besten, wenn die Kamerahöhe gemessen wird. Der Lotpunkt auf dem Spielfeld kann dann kurzzeitig markiert werden, um die x- und y-Koordinate korrekt einzustellen.

In der GUI wird die Kamera nun wie folgt eingestellt:

- Kamerakreuz auf den Lotpunkt ziehen.
- Die gemessene Kamerahöhe in das Feld „Kamera Z“ eintragen

Hinweis: Das Kamera-Kreuz lässt sich erst verschieben, wenn es Rot eingefärbt ist. Dies ist erst der Fall, wenn das Spielfeld bereits eingestellt worden ist. Sollte trotz eingestelltem Spielfeld das Kamera-Kreuz nicht rot werden, ist die Kameraposition, welche in den Edit-Boxen eingetragen ist, außerhalb des sichtbaren Bildes. Dann muss die Kameraposition zunächst über die Edit-Boxen wieder in das sichtbare Bild verschoben werden. Dies kann nur im 2-Kamera-Ansatz passieren.

IV.1.4 Performanz und Probleme



Vollständig eingestelltes Spielfeld

Wenn der Entzerrer präzise eingestellt wird, ist in der OpenGL-Ansicht des Robosoccer-Programms keine Positionsabweichung zu erkennen. Alle Fehler, die auftauchen lassen sich auf Einstellprobleme zurückführen.

- Die Fischaugenkorrektur arbeitet bis auf ein Pixel genau und kann kaum noch verbessert werden.

- Das Mapping auf Spielfeldkoordinaten ist so genau, wie es die Kameraauflösung zulässt und hängt lediglich von der Genauigkeit der Auswahl der vier Spielfeldeckpunkte ab.

- Die Parallaxenkorrektur ist der kritische Punkt des Systems, die Kameraposition kann nur schwer hundertprozentig genau eingestellt werden. Durch geringe Abweichung der Kameraposition kann es schon zu Abweichungen von einem Zentimeter und mehr kommen. Besonders im 2-Kamera-Ansatz kann dies zu enormen Problemen führen, wenn das Objekt aus dem einen Kamerabild verschwindet und im anderen auftaucht.

Im Abschnitt „Zuordnung von Blobs zu Robotern“ wird auf diese Problematik noch einmal eingegangen.

Wünschenswerteste Lösung wäre eine Funktionalität, die in der Camera-Calibration-Toolbox für MatLab bereits vorhanden ist. Dort kann für jedes Kalibrationsbild die Position der Kamera automatisch bestimmt werden. Diese Funktionalität konnte jedoch aus den Quelltexten der Toolbox nicht separiert werden und ist daher im System noch nicht vorhanden.

IV.2 HLS-Pixelfilter

IV.2.1 Erweiterung um farbspezifische LS-Werte

IV.2.1.1 Motivation

In dem alten HLS-Pixelfilter wurde für jede Farbklasse ein H-Intervall eingestellt. Für die LS-Werte dagegen konnte nur jeweils ein Intervall ausgewählt werden, der dann für alle Farbklassen galt. Dies hatte zu Folge, dass das Intervall oft für die einzelne Farbklasse zu groß ausfiel. Für das Filterresultat bedeutet das eine eingeschränkte Filterung. Dies kann unter umständen zu falsch erkannten Pixel führen. Um die Flexibilität des HLS-Pixelfilters zu steigern haben wir uns entschieden diesen zu erweitern. Nach der Erweiterung ist es nun möglich für jede Farbklasse einen eigenen L und S-Intervall einzustellen.

IV.2.1.2 Implementierung

Bei der Implementierung haben wir uns an dem Quellcode für die individuellen H-Werte orientiert. Ähnlich wie für die H-Werte haben wir den Quellcode um separate Werte für die L uns S-Intervalle erweitert. Anstatt dass übergeordnet mit dem global gesetzten L und S-Intervall verglichen wird, findet nun ein individueller vergleich statt. Diese Vorgehensweise kostet etwas mehr Laufzeit. In der ursprünglichen Implementierung konnte vorab auf das LS-Intervall überprüft werden und so Pixel die nicht in das Intervall gehörten vorab herausfiltern. Nun muss zuerst auf das H-Intervall geprüft werden und danach erst auf die LS-Werte. Im späteren Verlauf der PG wurde der HLS-Filter jedoch zwecks Performance auf LookUp-Tables umprogrammiert so, dass dieser, ohnehin kleiner Performance einbußen, nur bei der Neugenerierung ins Gewicht viel.

IV.2.1.3 GUI-Erweiterung

Um die LS-Intervalle setzen zu können wurde die GUI um entsprechende Funktionalität erweitert. Da die GUI für die H-Intervalle der einzelnen Farbklassen bereits vorhanden war konnten wir uns auch hier auf deren Implementierung stützen. Anstatt das diese GUI nur die H-Intervalle setzen konnte, haben wir Schalter eingebaut mit denen nun ausgewählt werden konnte welche der nun drei Intervall-Klassen gesetzt werden wollte. Die Eingabemaske der GUI wird jeweils nach umschalten der HLS-Schalter mit den jeweiligen Daten aktualisiert. Hiernach können die Werte angepasst werden. Es ist auch immer noch möglich mit den globalen LS-Intervallen erstmal für alle Farbklassen die LS-Bereiche zu wählen. Danach kann mittels der individuellen LS-Intervalle ein Finetuning erfolgen.

IV.2.2 Optimierung durch LookUp-Table

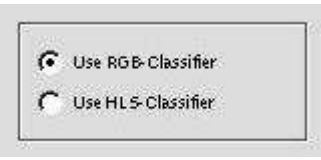
Es hat sich bereits in den Anfängen der PG herausgestellt, dass die robotsoccer Implementierung eine menge Laufzeit benötigte. Deswegen haben wir stets Optimierungen durchgeführt. Nicht zu letzt haben wir auch den HLS-Filter einer Optimierung unterzogen. Denn in der ursprünglichen Implementierung erfolgte die Einordnung in Farbklassen auf aufwendigem Wege:

- YUV-Pixel aus Bild holen
- YUV-Pixel in HLS-Pixel umwandeln
- HLS-Pixel mittels HLS-Filter in Farbklasse einordnen

Diese Vorgehensweise machte einen großen Teil der verbrauchten Rechenleistung aus. Um für die zukünftige Entwicklung des Projektes eine Reserve an Leistung zu schaffen und nicht zuletzt Raum für eine bessere Kamera mit einer höheren Auflösung und Bildrate zu machen, haben wir uns für eine LookUp-Table Variante des HLS-Filters entschieden. Dies kostet zwar nun zusätzlichen Speicher aber davon ist noch ausreichend vorhanden.
Die LookUp-Table deckt nun alle drei genannten Schritte mittels eines Zugriffes auf die Tabelle ab. Dazu muss vorher für jede YUV-Kombination die korrespondierende Farbklasse berechnet und in die LookUp-Table eingetragen werden. Nachdem also die HLS-Intervalle für die Farbklassen geändert wurden muss ein „rebuild“-Knopf betätigt werden, woraufhin eine neue Tabelle berechnet wird. Dies kostet einmalig Laufzeit. Im laufenden Spiel dagegen ist die Klassifizierung eines Pixels in eine Farbklasse nur noch ein einfacher Speicherzugriff.
Bei der Einstellung der HLS-Intervalle wird stets das Filterresultat vom Anwender überprüft, ist das Resultat nicht zufrieden stellend muss nachgestellt werden. Da das betätigen des „rebuild“Knopfes jeweils im Bereich von einigen Sekunden eine Neuberechnung der Tabelle zu folge hat, würde das störend auf die Einstellungsdauer wirken.
Um nicht bei jeder Änderung der HLS-Intervalle den „rebuild“-Knopf betätigen zu müssen, wird hier der alte drei stufen Filter benützt. Erst nachdem der Anwender mit der Einstellung fertig ist muss der Knopf für das Neuerstellen der Tabelle betätigt werden.

IV.2.3 Umschalten zwischen HLS und RGB-Pixelfilter zur Laufzeit

Bei der Einführung des neuen RGB-Pixelfilters stellte sich die Frage was mit dem bereits vorhandenen HLS-Filter passieren sollte. Da der HLS-Filter sich in Wettbewerben bereits bewert hatte, haben wir uns entschlossen ihn im Projekt zu behalten. Dazu musste eine Möglichkeit geschaffen werden zwischen den Pixelfiltern wählen zu können. Da beide Pixelfilter nun auf einer LookUp-Table basieren, reichte es diese LookUp-Table entsprechend zu wechseln. Für den Aufrufer des Filter-Moduls würde sich dagegen nichts ändern. Auf dieser Grundlage wurde das Projekt erweitert und nun ist ein Umschalten zur Laufzeit möglich. Dazu muss in der GUI einfach der bevorzugte Filter gewählt werden.



IV.3 Zuordnung von Blobs zu Robotern

Die Daten der Bildverarbeitung werden verwendet, um die Positionen der Spielfeldobjekte zu bestimmen. Jeder Roboter ist mit einer Farbmarkierung versehen, die neben seiner Position auch die Bestimmung seiner Orientierung ermöglicht.
Die aktuell genutzten Farbmarkierungen sind nicht eindeutig, sie bestehen aus zwei Farbrechtecken, das erste Rechteck ist die Teamfarbe, das andere eine gewählte Zusatzfarbe.

Die richtige Zuordnung von Blobs zu Robotern ist kritisch im Robosoccer-System, da ansonsten der falsche Roboter die falschen Funkbefehle bekommt.

Durch den Umbau und Austausch einiger Module der Bildverarbeitung kam es zu Problemen in der Blobzuordnung. Diese Probleme lassen sich in zwei Klassen einteilen:

- Falsche Zuordnung von Blobs zu Robotern innerhalb des Suchbereichs einer Kamera
- Falsche Zuordnung von Blobs zu Robotern innerhalb des Überlappungsbereichs der beiden Kameras

Das erste Problem hat mehrere Ursachen:

- Schnelle Roboter verwischen auf dem Kamera-Bild. Sie werden evtl. gar nicht erkannt oder die Farbstreifen vermischen sich mit den Farbmarkierungen anderer Roboter.
- Die Blobs werden falsch zu Patches zusammengesetzt, d.h. es wird einem Team-Blob der falsche Sekundär-Blob zugeordnet.

Das zweite Problem lässt sich hauptsächlich auf die Parallaxenkorrektur und ihre Einstellung zurückführen.

Die Fehler bei der Angabe der Kameraposition addieren sich und es kann passieren, dass die Vereinigung der Patchlisten von beiden Kameras nicht mehr korrekt arbeitet, d.h. ein Roboter taucht in der Ergebnisliste zweimal auf.

Lösungsansätze:

- Der Suchbereich für den Sekundärblob wurde stark reduziert. Ein Radius von 5,5 cm lieferte bei stehenden Robotern Ergebnisse ohne Vertauschungen. Allerdings kann es nun passieren, dass schnelle Roboter, die im Kamerabild verwischen nicht mehr erkannt werden können.

- Der Toleranzbereich für die Vereinigung von Patches im 2-Kamera-Ansatz wurde von 1cm auf 5cm erhöht. Es gibt seitdem keine doppelten Roboter mehr.

- Verwendung von eindeutigen Robotermarkierungen. Siehe Abschnitt ICRO-Patches

IV.4 ICRO-Patches

IV.4.1 Einleitung

Die bisher verwendeten bekannten DROIDS Roboterpatches bestehen aus zwei Farbblobs. Einer in Teamfarbe und einer in Zusatzfarbe. Mittels dieser zwei Blobs kann die Position und Ausrichtung des Patches und somit des Roboters auf dem Spielfeld genau genug erkannt werden.
Als die momentan aktuellen Patches der Dortmund DROIDS eingeführt wurden, waren sie bereits eine Verbesserung gegenüber der vorherigen Generation von Patches. Jene verwendete einen Blob in Teamfarbe und als Zusatzblobs Blobs in verschiedenen Farben. Dadurch war es möglich nicht nur Ausrichtung und Position aus einem Patch zu bestimmen, sondern auch direkt dessen Identität (ID). Unglücklicherweise zeigte sich, dass die verwendete (analoge) Kamera nicht in der Lage war ausreichend viele Farben zu differenzieren, so dass es oft zu Verwechselungen und falsch erkannten Patch IDs kam. Der neu eingeführte Patch verhinderte dieses, indem nur zwei Farben verwendet wurden, allerdings konnte nun nicht mehr die ID in die Patches mit hinein kodiert werden. Leider erwies sich dieses als eine Ursache von Problemen (siehe Kapitel 3), welche zwar mittels einigen Aufwands gelösten werden konnten, aber eben nicht vollends befriedigend.
Im Herbst 2005 hielt Herr DongHun Lee vom koreanischen Robot Soccer Team ICRO im Rahmen eines Deutschland Besuchs einen Vortrag, in welchem er der PG 472 das ICRO Team und deren Roboter genauer vorstellte. Unter anderem ging er auch auf neuartige Patches ein, welche von seinem Team entwickelt wurden und diverse Vorzüge gegenüber den bisher üblichen Patches haben. Da wir es zu dieser Zeit ohnehin für Notwendigkeit erachteten die bisherigen DROIDS Patches neu zu entwickeln, entschlossen wir uns zum Versuch unser "robosoccer " System so zu modifizieren, dass es auch mit den von Herrn Lee vorgestellten Patches arbeiten kann. Wir wollten sehen, ob und wie gut die, von uns im folgenden als ICRO Patches bezeichneten Patches, in unser bestehendes System integriert werden können, und inwieweit sie tatsächlich eine Verbesserung gegenüber den bisherigen

DROIDS Patches darstellen. Dabei wurden wir von Herrn Lee großzügig unterstützt, indem er uns einen, damals noch nicht veröffentlichten, Artikel über die ICRO Patches zu kommen ließ.

IV.4.2 Aufbau der neuen Patches

IV.4.2.1 Allgemeiner Aufbau der original ICRO Patches

Die ICRO Patches wie sie im Paper vom Lee beschrieben werden, bestehen aus folgendem einfachem Aufbau: Auf einer schwarzen quadratischen Grundfläche sind vier kreisförmige Blobs wie unten stehend angeordnet.

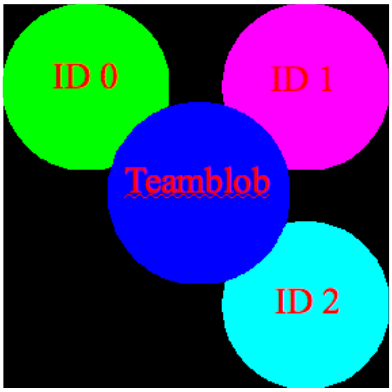


Abbildung IV.4-1Ein ICRO Patch.

Der Blob in der Mitte ist der Teamblob und ist deshalb stets in Teamfarbe (Blau oder Gelb) gehalten. Um ihn herum sind in den Ecken des Quadrats drei etwas kleinere Zusatzblobs angeordnet, welche im Folgenden mit ID0, ID1 und ID2 bezeichnet werden. Jeder dieser drei Zusatzblobs kann eine von drei Zusatzfarben annehmen. Auf diese Weise sind also insgesamt $3^3 = 27$ verschiedene Farbkombinationen möglich, wobei jede dieser 27 Farbkombinationen für eine Patch ID steht.

IV.4.2.2 Eigene Implementation der ICRO Patches

Für das DROIDS Robosoccer System wird eine leicht modifizierte Version der ICRO Patches verwendet. Bei unserer eigenen Implementation der ICRO Patches beträgt die Seitenlänge des Patches 7.5 cm und die Kreise haben jeweils einen Durchmesser von 3,25 cm. Im Paper von Lee werden leider keiner genauen Maßangaben gemacht, sie sollten aber in etwas unseren entsprechen. Im Unterschied zu den Original ICRO Patches ist der Teamblob in der Mitte allerdings nicht mehr kreisförmig sondern quadratisch. Dies entspricht unserer Auffassung nach, eher der in den offiziellen FIRA Regeln geforderten Maßgabe, dass der Teamblob eine mindestens 3,5 cm x 3,5 cm große

zusammenhängende Fläche sein muss. Das Team ICRO interpretiert diese Regel dagegen so, dass der Teamblob eine zusammenhängende Fläche von mindestens 12,25 cm^2 (3,5 cm * 3.5 cm) ist , d.h sie muss demnach nicht unbedingt quadratisch sein. Tatsächlich ist ein runder Teamblob günstiger für die Bildverarbeitung, da bei geraden Kanten tendenziell eher Bildfehler entstehen können.

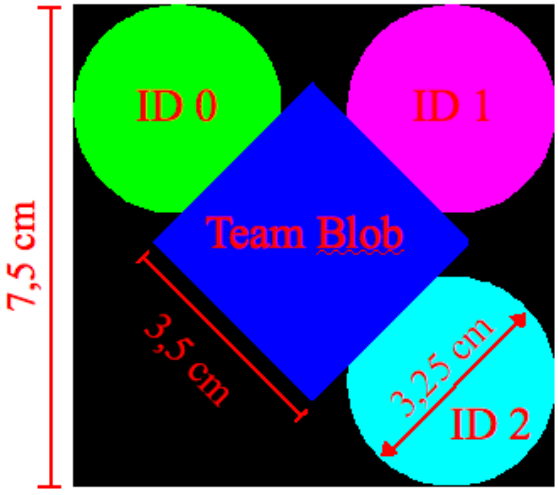


Abbildung IV.4-2Abmessungen des modifizierten ICRO Patches

Um Schwierigkeiten mit Verwischungen und Verschmelzungen zu minimieren, werden bei unserer Implementierung der ICRO Patches zudem keine Farbkombinationen verwendet bei denen zwei Zusatzblobs gleicher Farbe nebeneinander liegen bzw. alle Zusatzblobs die gleiche Farbe haben. Dieses reduziert die Anzahl der möglichen Patch Identitäten auf 12, was aber bei maximal 11 eigenen Robotern auf dem Feld kein Problem darstellt.

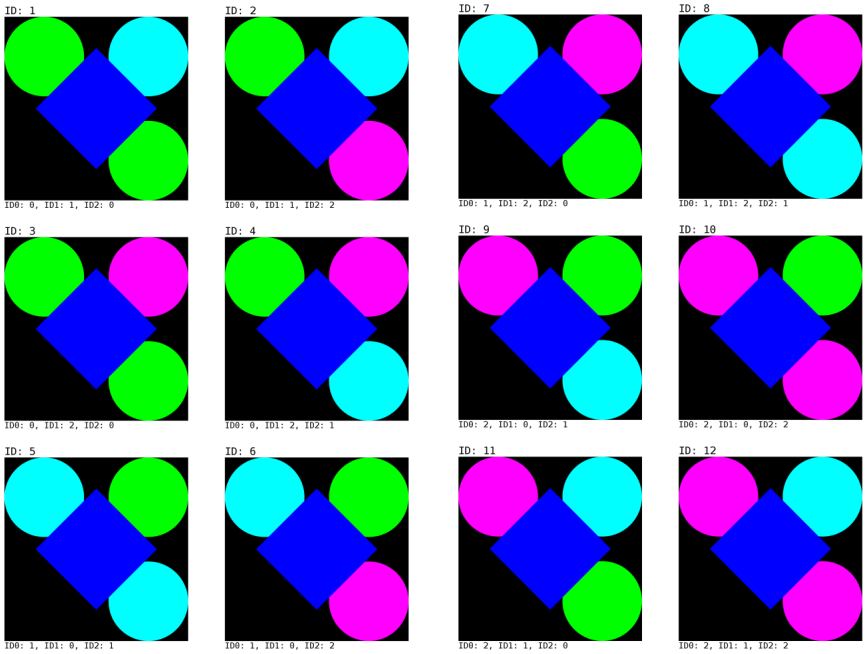


Abbildung IV.4-3: Patch mit IDs von 1 bis 12

IV.4.2.3 Die ID Erkennung

Um aus dem Bild eines Patches dessen ID zu berechnen muss nicht nur bekannt sein, welche Zusatzfarben generell auf dem Patch vorkommen können, sondern auch welcher der Zusatzblobs (ID0,ID1,ID2), welche Farbe hat. Erst dann kann die Patch ID zuverlässig bestimmt werden. Zu diesem Zweck wurden zwei Verfahren entwickelt, welche im folgenden beschrieben werden. Sie unterscheiden sich geringfügig darin auf welche Weise für einen gefundenen Teamblob dessen ICRO Zusatzblobs ermittelt werden. Für ein besseres Verständnis sei darauf hingewiesen, dass im Patchfinder zunächst alle vom Blobfinder auf dem Spielfeld gefundenen Blobs zu je einer der folgenden Listen zugewiesen werden:

- Liste der eigenen Teamblobs
- Liste der gegnerischen Teamblobs
- Liste der Ballblobs
- List der restlichen Blobs (Restblobs)

IV.4.2.4 Methode 1: Erkennung mittels Geometrie

Für jeden Blob aus der Liste der eigenen Teamblobs werden folgende Schritte ausgeführt:

Schritt 1: Durchlauf aller Restblobs

Es wird die Liste aller Restblobs durchlaufen. Dabei werden diejenigen Restblobs gesucht, deren Mittelpunkt sich innerhalb des doppelten Patchradius befinden, wobei mit Patchradius der Radius des Kreises gemeint ist, welcher alle vier Ecken des Patches durchläuft. In der unten stehenden fiktiven Spielsituation entspricht dieser Radius der gestrichelten roten Linie. Die so gefundenen Blobs werden in eine Liste gepackt, welche im nächsten Schritt weiter bearbeitet wird.

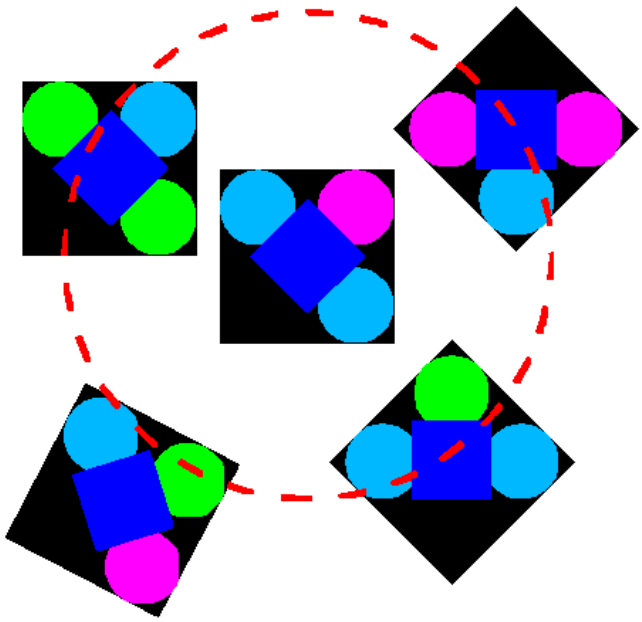


Abbildung IV.4-4: Blobsuche innerhalb des doppelten Patchradius

Schritt 2: Bestimmung der nächst liegenden Restblobs

Die Blobs in der in Schritt 1 generierten Liste, werden nun aufsteigend zur Entfernung zum aktuell betrachteten Teamblob sortiert. Dabei wird als Abstandsmaß nicht die Entfernung von Teamblob Mittelpunkt zum Restblob Mittelpunkt verwendet. Vielmehr wird für jeden Blob in der Liste der Abstand des nächsten *Pixels* zum Teamblob Mittelpunkt berechnet und als Abstandsmaß verwendet. Dieses Vorgehen garantiert, dass das Ergebnis nicht durch Blobverschmelzungen verfälscht wird, denn der Mittelpunkt eines verschmolzenen oder verwischten Blobs entspricht nicht mehr dem Tatsächlichen. Wurde dieses für alle relevanten Blobs gemacht, befinden sich die zum aktuellen Teamblob nächst gelegenen drei Blobs am Anfang der Liste. Diese sind also somit die potentiellen Zusatzblobs ID0, ID1 und ID2.

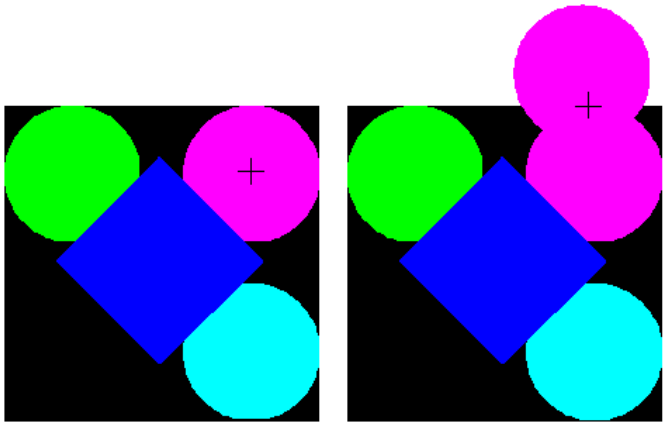


Abbildung IV.4-5: Mittelpunktverschiebung durch Verschmelzung

Schritt 3: Bestimmung von ID1

Nach Abarbeitung der Schritte 1 und 2, ist zu diesem Zeitpunkt für den momentan betrachteten Teamblob bekannt, welche drei Zusatzblobs er hat. Allerdings ist unklar, welcher Blob an welcher Position innerhalb des Patches steht. Das heißt also, es kann noch nicht gesagt werden, welcher der gefundenen Zusatzblobs den Blob ID0, ID1 oder ID2 darstellt. Der Schlüssel zur Lösung dieses Problems liegt nun darin, zuerst zu bestimmen, welcher Blob den Zusatzblob ID1 darstellt. Um dieses herauszufinden kann eine geometrische Eigenschaft der ICRO Patches ausgenutzt werden. Hierfür wird zunächst folgende Funktion definiert:

Definition: Winkel(X,Y)
Die Funktion Winkel(X, Y) bezeichne den Winkel zwischen zwei übergebenen Blobs X und Y, bezogen auf den momentan aktuellen Teamblob Mittelpunkt. Der Winkel wird gemessen indem ein Vektor vom Blob X Mittelpunkt zum Teamblob Mittelpunkt und ein Vektor vom Blob Y Mittelpunkt zum Teamblob Mittelpunkt berechnet wird. Anschließend wird der Winkel zwischen diesen beiden Vektoren berechnet.

Mithilfe dieser Funktionen können nun die Variablen WinkelSummeBlob0, WinkelSummeBlob1 und WinkelSummeBlob2 wie folgt definiert werden:

WinkelsummeBlob0 = (Winkel(Blob0, Blob1) + Winkel(Blob0, Blob2))

WinkelsummeBlob1 = (Winkel(Blob1, Blob0) + Winkel(Blob1, Blob2))

WinkelsummeBlob2 = (Winkel(Blob2, Blob0) + Winkel(Blob2, Blob1))

Für jeden der drei zum Patch gehörig ermittelten Zusatzblobs wird also die Summe der Winkel zu den zwei übrigen Blobs berechnet. Aufgrund der Geometrie der ICRO Patches gilt, dass der Blob mit der kleinsten Winkelsumme der Blob ist, welcher ID1 entspricht. Dieser Sachverhalt wird anhand der unten stehenden Abbildung unmittelbar klar:

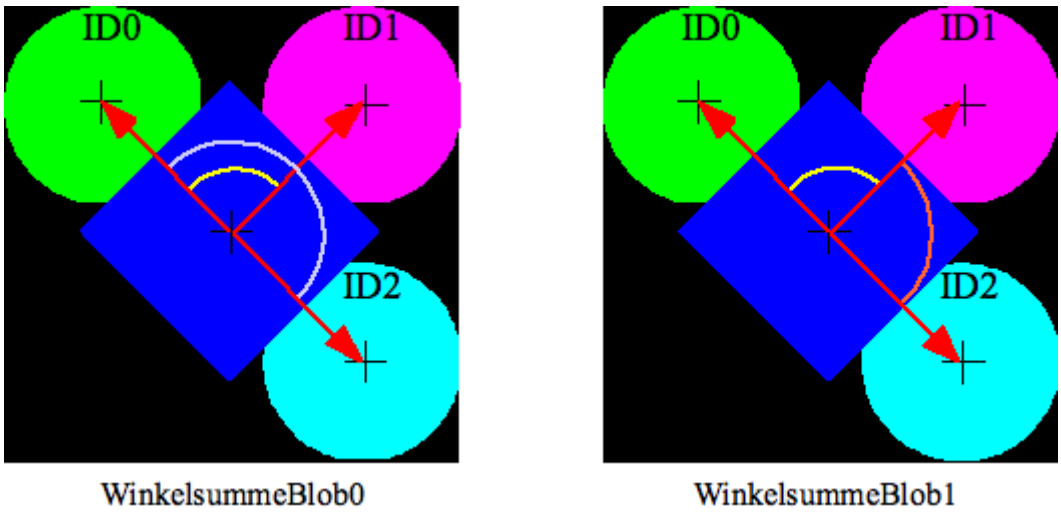


Abbildung IV.4-6: Winkelsummen

WinkelsummeBlob0 berechnet sich als die Summe von Winkel(ID0, ID1) und Winkel(ID0,ID2). Die beiden Winkel betragen (im Idealfall) 90° bzw. 180°, so dass eine Summe von 270° entsteht (WinkelsummeBlob2 analog). Die WinkelsummeBlob1 dagegen, setzt sich aus Winkel(ID0, ID1) und Winkel(ID1,ID2) zusammen, welche beide 90° betragen. Die Summe beträgt also nur 180°.

Schritt 4: Bestimmung von ID0 und ID2

Zu diesem Zeitpunkt ist bekannt, welcher der drei Zusatzblobs den ID1 Blob darstellt. Dank dieser Information kann nun relativ einfach ermittelt werden, welcher der übrigen zwei Blobs ID0 und welcher ID2 darstellt. Hierfür wird ein Richtungsvektor von Blob1 zu Blob2 berechnet (Reihenfolge darf auch umgekehrt sein). Dieser Vektor spannt eine Gerade im zweidimensionalen Raum auf bzgl. derer der Blob ID1 links oder rechts davon liegen kann. Dieses kann recht einfach mit dem Skalarprodukt berechnet werden. Ist dieses getan, wird wieder die Geometrie der ICRO Patches ausgenutzt, denn es gilt:

Befindet sich ID1 links vom Vektor, dann muss Blob1 dem Blob ID0 entsprechen.

Befindet sich ID1 rechts vom Vektor, dann muss Blob1 dem Blob ID2 entsprechen.

Somit ist von zwei der drei Blobs bekannt, welchem ICRO Patch Blob sie entsprechen, und die Identität des dritten verbliebenen Blobs ergibt sich somit automatisch.

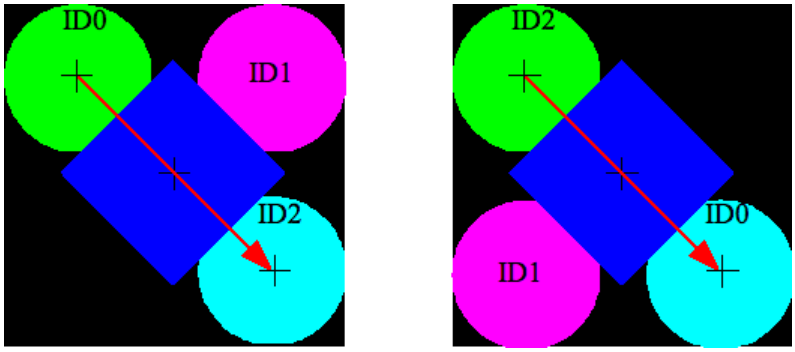


Abbildung IV.4-7: Bestimmung von ID0 und ID2 mittels Lagebestimmung von ID1 relativ zum Richtungsvektor

Schritt 5: Bestimmung der Patch ID

Ist das Verfahren an diesem Schritt angelangt, dann ist bekannt welcher der drei in Schritt 2 gefundenen Zusatzblobs welchem Patch Zusatzblob (ID0, ID1,ID2) entspricht. Somit ist also auch bekannt in welcher von den drei möglichen Farbklassen ID0, ID1 und ID2 jeweils liegen. Diese Information kann nun dazu genutzt werden, die Patch ID zu ermitteln. Dies geschieht mittels einem einfachen Array LookUp bei dem die ermittelten Farbklassen, der drei Patch Zusatzblobs, als Index benutzt werden (siehe auch Kapitel 5.4.2 "Implementierung des ICRO Patchfinders").

IV.4.2.5 Methode 2: Erkennung über Abstand

Diese Methode wurde zu Gunsten der oben beschriebenen aufgegeben. Sie unterscheidet sich allerdings im wesentlichen nur darin, dass in Schritt 3 der Zusatzblob ID1 nicht mithilfe von Winkeln sondern mittels Abständen bestimmt wird. Aufgrund der Patchgeometrie ist nämlich von allen Abständen zwischen den Blobs ID0, ID1 und ID2 der Abstand zwischen den Blobs ID0 und ID2 maximal. Zur Bestimmung von ID1 reicht es also aus, die Abstände zwischen allen drei gefundenen Zusatzblobs zu berechnen. Bei denjenigen zwei Blobs, für die der Abstand maximal ist, muss es sich um ID0 und ID2 handeln, woraus ableitbar ist, dass der verbliebene Zusatzblob ID1 darstellt.

IV.4.2.6 Gegenüberstellung Methode1 und Methode2

Der Grund warum Methode 1 gegenüber Methode 2 bevorzugt wurde, liegt darin begründet, dass die Ermittlung von ID1 über die Winkel zuverlässiger arbeitet als über Abstände. Verwischt ein Blob oder verschmilzt ein Blob mit einem anderen, so verschiebt sich dessen Mittelpunkt, weshalb der Abstand zwischen ID0 und ID2 nicht mehr der größte von allen Abständen sein muss. Demgegenüber ist die in

Methode 1 verwendete Bestimmung über Winkel robuster. Zwar verschieben sich auch hier bei Verwischung und Verschmelzung die Blob Mittelpunkte, allerdings bleibt die relative Lage der Blobs zueinander, also die Winkel, recht stabil. Bezogen auf den oben beschriebenen Schritt 3 bedeutet dieses also, dass WinkelsummeBlob1 immer die kleinste von allen Winkelsummen ist. Somit ist also die Methode 1 im Gegensatz zu Methode 2 relativ stabil gegenüber Verschmelzungen.

IV.4.2.7 Ausrichtungsbestimmung

Die Bestimmung der Ausrichtung eines ICRO Patches erfolgt ähnlich wie bei den alten DROIDS Patches. Es wird ein Vektor zwischen den Blobs ID0 und ID2 berechnet. Anschließend wird der Winkel zwischen diesem Vektor und einem Referenzvektor berechnet, woraus die Ausrichtung auf dem Spielfeld abgeleitet werden kann.

IV.4.3 Vorteile und Nachteile der neuen Patches

Im folgenden werden die Vor- und Nachteile der ICRO Patches und der bisherigen DROIDS Patches betrachtet.

IV.4.3.1 Vorteile der ICRO Patches:

- Robustheit gegenüber Vertauschungen
  Da alle Patches eine eindeutige ID haben, sind Vertauschungen kaum möglich. Sollte es dennoch zu einer Vertauschung kommen, findet das System mit hoher Wahrscheinlichkeit ohne Intervention von außen zu den richtigen IDs zurück, sobald die kritische Spielfeldsituation sich aufgelöst hat. Dies ist meistens schon wenige Frames später der Fall.

- Robustheit gegenüber Zusatzblob Verschmelzungen
  Die ICRO Patches sind so aufgebaut, dass alle Zusatzblobs in einem bestimmten Radius um den Patch Mittelpunkt herum liegen. Zusammen mit der Tatsache, dass der Mittelpunkt des Patches einfach dem Mittelpunkt des Teamblobs entspricht und somit trivialerweise gegeben ist, resultiert, dass die Zusatzblobs für den betrachteten Teamblob einfach ermittelt werden können: Vorausgesetzt der RGB Pixelfilter führt keine Fehlklassifikationen durch, handelt es sich immer um die drei nächsten zur Teamblob Mitte befindlichen Zusatzblobs.

- Robustheit gegenüber Teamblob Verschmelzungen
  Da die Teamblobs genau mittig auf dem Patch sitzen, ist es sehr unwahrscheinlich, dass Teamblobs mehrerer Patches miteinander verschmelzen. D.h es gibt immer so viele

Teamblobs auf dem Feld, wie es auch Roboter gibt. Und da der Teamblob Mittelpunkt (entspricht dem Patch Mittelpunkt) eine wichtige Grundlage für die zuverlässige Erkennung der Zusatzblobs ist, führt diese Robustheit zudem dazu, dass auch die Erkennung der korrekten Zusatzblobs besser wird.

- Kein Tracking notwendig
  Da bei den ICRO Patches die ID in den Patches kodiert ist, entfällt die Notwendigkeit für einen Tracker. Es müssen keine Roboterbewegungen mehr verfolgt werden, um zu wissen welcher Patch welchem Roboter entspricht. Für die Frage, warum dies ein großer Vorteil ist, sei auf den Abschnitt "Nachteile der DROIDS Patches" verwiesen.

- Vereinfachter Mixer
  Da jeder Patch eine eindeutige ID hat, kann das Mixen von zwei Kamerabildern sehr vereinfacht werden. Außerdem wird die Robustheit des Mixers bzgl. sehr stark gegeneinander verschobener Kameraaufnahmen erhöht, da nicht mehr die Position der Patches innerhalb des Überlappungsbereichs relevant ist, sondern nur deren ID. Siehe dazu auch im Kapitel 5 "Implementation" den Abschnitt 3 "Anpassungen des Mixers".

IV.4.3.2 Nachteile der ICRO Patches

- Viele Farben
  Bei den alten Patches müssen nur zwei Farben sicher unterschieden werden. Bei den ICRO Patches ist dagegen die Unterscheidung von insgesamt vier Farben notwendig. Die Leistungsfähigkeit der ICRO Patches hängt somit wesentlich auch von der Leistungsfähigkeit bzw. Einstellung des Pixelfilters ab, wie auch von der Qualität der Kamera.

- Trivialer Mittelpunkt
  Im Gegensatz zu den alten Patches, ist es für den Gegner bei Verwendung der ICRO Patches recht einfach die Position unserer Roboter zu ermitteln. Dieser entspricht trivialerweise dem Mittelpunkt des Teamblobs.

- Kleinere Blobflächen
  Die Flächen der Blobs sind kleiner und somit anfälliger für Bildstörungen. Je kleiner ein Blob ist, desto höher wird das Verhältnis von Pixeln, die auf dem Blob Umriss liegen zur Gesamtfläche. Da aber vor allem am Blob Umriss Rauschen auftritt, bedeutet dieses, dass das Rauschen mehr ins Gewicht fällt.

IV.4.3.3 Vorteile der DROIDS Patches

- Einfaches erstellen von Patches

  Da die Geometrie der DROIDS Patches recht einfach ist, können diese relativ leicht erstellt werden.

- Umtausch defekter Patches

  Sollte ein Patch im Laufe der Anwendung beschädigt werden, ist dessen Umtausch einfach, da alle Patches gleich aussehen.

- Wenige Farben

  Zur Erkennung der Patches sind nur die Farbklasse des Teamblobs und des Zusatzblobs notwendig.

- Nicht trivialer Mittelpunkt

  Der Mittelpunkt des Patches kann nur durch den Mittelpunkt des Team- und Zusatzblobs ermittelt werden. Der Gegner kann also nicht durch einfaches suchen des Teamblobs auf die Position des Roboters schließen.

IV.4.3.4 Nachteile der DROID Patches

- Tracking zwingend

  Das Fehlen von ID Informationen auf den Patches macht es notwendig, dass das System durch ständige Verfolgung der Roboterbewegungen festhalten muss, welcher Patch zu welchem Roboter gehört (Tracking). Dieses benötigt kaum Rechenzeit, denn es ist sehr einfach implementiert. Leider führt aber gerade diese einfache Art der Implementierung zu Fehlern. Prallen z.B zwei Roboter aufeinander auf, kann es beim Tracking zu Fehlzuordnungen kommen. Zudem ist die Anzahl der Bilder, die die Kamera liefert, begrenzt. Bei einer Framerate von 30 FPS ist der Tracking Algorithmus für ca. 33 ms Sekunden ohne Informationen. Haben in dieser Zeit zwei Roboter ihren Platz getauscht, kann es zu einer ID Vertauschung kommen. Das wesentliche Problem, das der Tracker in seiner jetzigen Implementierung hat, besteht darin, dass bei einer Vertauschung von zwei Roboter IDs dieses nicht reversibel ist. Um dieses Problem zu lösen, müsste der Tracker dahingehend geändert werden, dass er mit der Strategie kommuniziert: Die Strategie weiß, welche Roboter welche Befehle bekommen. Indem der Tracker nun den von der Strategie gegebenen Befehl mit der tatsächlichen Ausführung vergleicht, könnte er darauf schließen, ob die angenommene

Roboter ID der tatsächlichen entspricht oder fehlerhaft ist. Es ist unmittelbar klar, das solch ein Verfahren recht lange dauern würde und auch nicht trivial zu implementieren wäre.

- Mittelpunkt nicht trivial
  Dieser Aspekt wurde weiter oben bereits als Vorteil erwähnt, ist aber gleichzeitig auch ein Nachteil. Denn da bei den DROIDS Patches der Mittelpunkt der Patches erst berechnet werden muss, kann diese Informationen nicht dazu genutzt werden, direkt die Zusatzblobs zu finden. Im Vergleich zu den ICRO Patches entfällt also eine zuverlässige und einfach zu ermittelnde Berechnungsgrundlage.

IV.4.3.5 Resümee

Der wesentliche Vorteil der bisher eingesetzten DROIDS Patches ist der, dass diese Patches mit nur zwei Blobs (und damit nur zwei Farben) auskommen. Entsprechend können die Blobs eine relativ große Fläche haben. Somit ist es bei schlechten Lichtverhältnissen einfacher die korrekten Blobfarben zu erkennen, und durch die größere Fläche sinkt auch die Anfälligkeit gegenüber dem allgegenwärtigen Bildrauschen. Leider ist das recht fehleranfällige Tracking ein gewichtiges Argument gegen die DROIDS Patches.
Bei den ICRO Patches ist die Lage umgekehrt. Es müssen mehr Farben voneinander unterschieden werden und die Blobs sind auch kleiner als bei den DROIDS Patches. Dafür wird aber kein Tracking mehr benötigt, denn die IDs sind in die Patches mit hinein kodiert. Bedenkt man noch, dass - wie in der Einleitung erwähnt – das Unvermögen früherer Kameras mehrere Farben zu separieren eine Hauptmotivation für die DROIDS Patches war, dann kann durchaus argumentiert werden, dass diese Begründung mittlerweile obsolet ist. Die Farbauflösung der heute verwendeten Sony Kameras ist – bei geschickter Farbwahl - ausreichend (siehe Kapitel 6"Testergebnisse"). Somit spräche nicht viel gegen die Verwendung von ICRO Patches. Vor allem nicht, wenn man bedenkt, dass durch bessere Kameras mit höherem zeitlichem Auflösungsvermögen und besserer Farbseperation, die Leistungsfähigkeit leicht noch weiter gesteigert werden kann.

IV.4.4 Herstellung der neuen Patches

Wie im vorangegangen Kapitel bereits angedeutet wurde, ist die Farbwahl bei den ICRO Patches von wichtiger Bedeutung. Bei der Wahl der Farben wurde deshalb erheblicher Aufwand getrieben, weshalb in diesem Kapitel etwas genauer darauf eingegangen werden soll.
Probleme gab es insbesondere bei der Farbwahl für die drei zu definierenden Zusatzfarben. Die Problematik resultierte daraus, dass nicht mehr wie bei den DROIDS Patches nur zwei Farben einen Patch bilden, sondern es im ungünstigsten Fall nun vier unterschiedliche Farben auf einem Patch gibt.

Da die Blobs in unmittelbarer Nähe zueinander liegen, ist es wichtig Farben zu finden, welche gut separierbar sind, also welche im Farbraum möglichst weit auseinander liegen und tendenziell nicht zum verwischen und ineinander verschmelzen neigen.
Hinzu kommt, dass das der RGB Farbfilter sechs Farbklassen kennt, von denen drei bereits für die eigene und gegnerische Teamfarbe (Blau und Gelb) sowie den Ball (Rot) verwendet werden. Für die drei benötigten Farbklassen der ICRO Zusatzblobs bleiben somit noch Grün, Magenta und Cyan übrig. Sind die Farbklassen Magenta und Grün noch relativ einfach zu definieren und von den übrigen Klassen zu unterscheiden, stellte sich dieses bei Cyan als etwas schwieriger heraus: Erschienen auf einem Blatt Papier die gewählten Blau- und Cyanfarbtöne noch als relativ gut unterscheidbar, so war dieses im aufgenommen Kamerabild oft nicht der Fall. Allgemein ist es so, dass die Farben im aufgenommenen Kamerabild, je nach Beleuchtung und Blende, andere RGB Werte haben als angenommen. Farben, die auf dem ersten Blick relativ klar unterscheidbar wirken, neigen im aufgenommenen Kamerabild zuweilen stark zum verschmelzen. Wie oben erwähnt, galt dies vor allem für die Farben Blau und Cyan: Insbesondere an den Umrissen der Blobs, wo Cyan direkt an Schwarz angrenzt, kann dies dazu führen, dass fälschlicherweise Blau erkannt wird. Dies ist ein allgemeines Problem, das sich ergibt, wenn Farben sich im wesentlichen nur in der Helligkeit unterscheiden. Ähnliches kann auch bei der Verwendung von Magenta in Verbindung mit einer hautfarbenen Ersatzfarbe für Cyan (siehe Abschnitt "Problemfall: Cyan mit Blau") beobachtet werden. Dort ist es allerdings nicht so kritisch, da die beiden Farben nicht zum Teamblob gehören.

IV.4.4.1 Drucken von Patches

Eine einfache und gleichzeitig elegante Methode, die 11 benötigten Patches anzufertigen ist, diese in Form von zwei DIN A4 großen Bögen zu drucken. Wählt man das Papier stark genug (ab 200 g/cm^2) können die Patches dann sogar mittels Klettverschluss direkt auf die Roboter befestigt werden. Das Drucken von Patches eröffnet auch die Möglichkeit von "Einweg Patches“. Diese können bei starker Verschmutzung oder Beschädigung während eines Turniers einfach entsorgt und ohne großen Aufwand neu gedruckt werden. Zudem können in relativ kurzer Zeit, relativ viele Farben bzw. Farbkombinationen ausprobiert werden.

IV.4.4.2 Druckerevaluierung mittels Farbbögen

Leider hat das Drucken der Patches aber auch einen Nachteil: Abhängig davon welcher Druckertyp verwendet wird, ergeben sich bei gleichen Farben unterschiedliche Ausdrucke. Bei einem Test von drei Druckern am Lehrstuhl 1 schwankten die Farben der Ausdrucke teils erheblich. Möchte man also reproduzierbare Ergebnisse haben, sollte stets nur derselbe Drucker verwendet werden. Vor allem bei

Tintenstrahldruckern ist noch zu beachten, dass auch je nach verwendeter Papiersorte die Farben der Ausdrucke stark schwanken können. Darüber hinaus wurde festgestellt, dass die ausgedruckten Farben nicht nur vom Drucker sondern auch vom verwendeten Bildbearbeitungsprogramm abhängig sind (trotz immer gleicher RGB Werte), was sehr wahrscheinlich an der verwendeten Farbkalibrierung liegt. Zusammen mit der Tatsache, dass Farben auf dem Bildschirm nicht den ausgedruckten Farben entsprechen, führten diese Ergebnisse dazu, dass mittels relativ umfangreicher Tests erst festgestellt werden musste, welcher Drucker am besten für die Patches geeignet ist. Um dieses in möglichst kurzer Zeit zu bestimmen, wurde ein Konsolen Programm namens *color-table-gen* geschrieben, welches mit Hilfe der Cairo Bibliothek in der Lage ist, Farbbögen zu erzeugen. Zu diesem Zweck übernimmt es sechs Parameter:

```
color-table-gen subdiv_red subdiv_green subdiv_blue col_r_start col_r_end
                col_g_start col_g_end          col_b_start col_b_end
```

Mittels dieser Parameter wird das Aussehen der Farbbögen wie folgt spezifiziert:

- subdiv_red, subdiv_green, subdiv_blue
  Diese drei Parameter geben an, wie oft die entsprechende Grundfarbe (Rot, Grün, Blau) unterteilt werden soll. Zu beachten ist, dass das Produkt dieser drei Werte nicht größer als 48 sein darf.

- col_r_start col_r_end col_g_start col_g_end col_b_start col_b_end
  Mit diesen Parametern wird jeweils für Rot, Grün und Blau der Bereich spezifiziert, welcher - gemäß der vorher angegebenen drei Parameter – unterteilt werden soll. Dabei dürfen die Parameter nur Werte zwischen 0 und 255 annehmen.

Für jede der drei Grundfarben wird also ein Bereich spezifiziert. Jeder dieser Bereiche wird dann in *subdiv* viele Farbwerte unterteilt. Anschließend werden alle möglichen Kombinationen dieser Farbwerte gebildet. Dies sind, aufgrund der für die *subdiv* Parameter gegebenen Bedingung, maximal 48 Stück. Anschließend wird eine png Datei namens "color_table" erzeugt, welche DIN A4 Format hat und 48 farbige Quadrate enthält. Dabei stellt die Farbe jedes dieser Quadrate eine der möglichen 48 Farbkombinationen dar. Zusätzlich wird in jedem Quadrat vermerkt, welchen R,G und B Werten die Farbe entspricht.
Von jeder benötigten Farbklasse wurden nun mit Hilfe dieses Programms Farbbögen auf verschiedenen Druckern generiert. Die Farbbögen gestatten es auf einfache Art und Weise direkt zu

testen, wie Farben eines bestimmten Druckers im aufgenommenen Kamerabild wirken. Dazu werden diese Bögen unter die Kamera gelegt, und anschließend mittels dem RGB Filter direkt die RGB Werte der einzelnen Farbkästchen ausgelesen. So kann relativ leicht festgestellt werden, ob die gedruckte Farbe durch die Kameraaufnahme zu sehr in Richtung einer anderen tendiert. Stellt man z.B fest, dass eine bestimmte Farbe einen Blau Anteil von 255 hat, so sollte davon abgesehen werden, diese Farbe als Zusatzblobfarbe zu verwenden, wenn als Teamfarbe Blau verwendet wird.

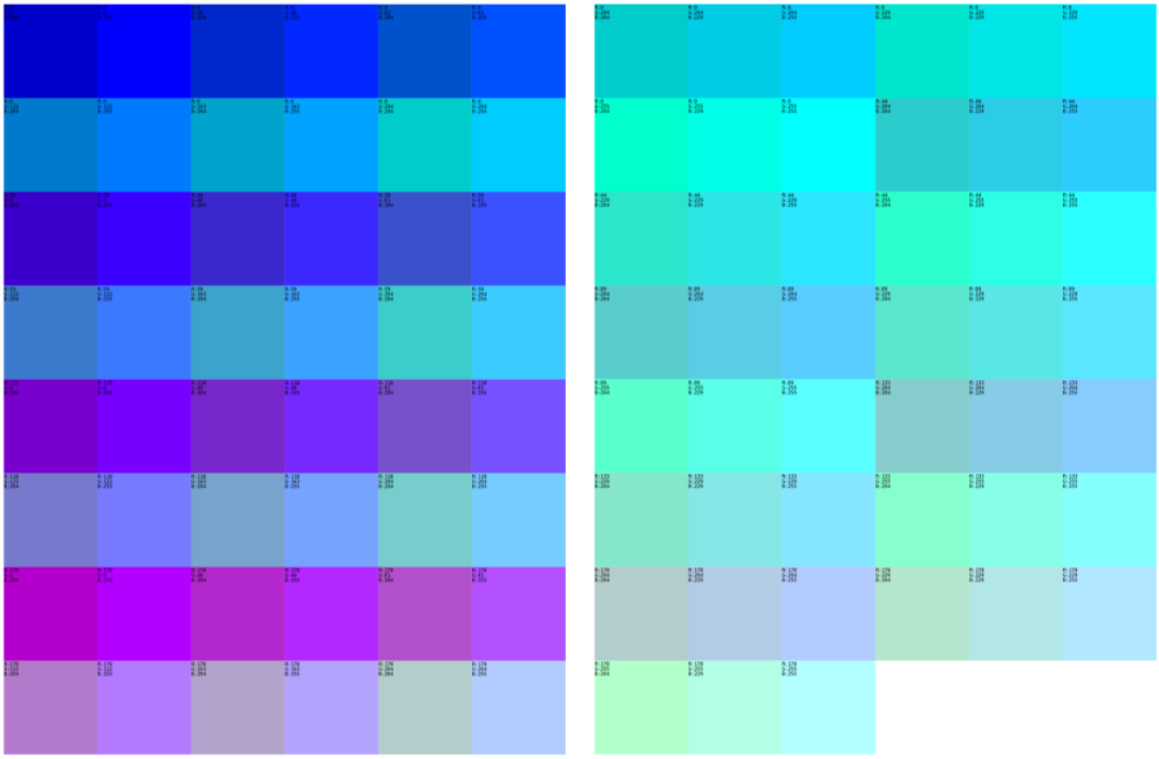


Abbildung IV.4-8: Farbbögen für Blau und Cyan

IV.4.4.2.1 Problemfall: Cyan mit Blau

Für die Erkennung eines ICRO Patches ist es wichtig, dass dessen Teamblob gut erkannt wird. Darum ist es besonders kritisch, wenn aufgrund von Störungen Phantom Blobs blauer Farbe entstehen, also Blobs die eigentlich keine sind. Insbesondere gab es diesbezüglich Probleme mit der Blobfarbe Cyan in Verbindung mit blauem Teamblob, da diese Farben im Farbraum recht dicht beieinander liegen und somit relativ leicht ineinander übergehen können. Darum kann es bei ungünstigen Lichtverhältnissen bzw. Kameraeinstellungen zu störenden Effekten kommen: Z.B kann ein zusätzlicher kleiner blauer Blob entstehen, welcher am Rand vom Cyanblob liegt. Oder es entsteht durch ungünstige

Verschmelzung von Blau mit anderen Farben ein Cyan Blob. Es wurde auch schon beobachtet, dass bei großer Verwischung ein Cyan farbiger Blob Blau erschien.
Diese erwähnten Probleme können aber relativ leicht umgegangen werden: Zum einen hilft es schon etwas mit den Shutter Werten der Kamera zu experimentieren, zum anderen könnte man die Verwendung von Cyan auf den Fall von gelben Teamblobs beschränken. Bei blauem Teamblob wird dann statt Cyan ein hautfarbener Ton verwendet, welcher sich relativ gut von Blau und den anderen Zusatzfarben separieren lässt. Die Einstellung in robotsoccer hierfür ist trivial: Im RGB-Filter wird die Farbklasse Cyan einfach mittels der hautfarbenen Blobs definiert. Zwar kann es dabei zu einem Magentarauschen am Umriss von diesen hautfarbenen Blobs kommen, dieses ist aber relativ unkritisch, da keiner der beiden Farben eine Teamfarbe ist.

IV.4.5 Das Patchgenerierungs Programm

Um die Patches nicht jedes mal neu mit einem Bildbearbeitungsprogramm zeichnen zu müssen, wenn sich eine der Zusatzfarben ändert, wurde ein kleines Programm namens *icro-patch-gen* entwickelt, welches in der Lage ist png Dateien zu generieren, die ICRO Patches enthalten. *icro-patch-gen* erwartet hierfür insgesamt neun Parameter:

```
icro-patch-gen colID0_r colID0_g colID0_b colID1_r colID1_g colID1_b
               colID2_r colID2_g colID2_b
```

Mittels dieser Parameter werden die drei möglichen Farben, die die Zusatzblobs haben können spezifiziert:

colID0_r, colID0_g, colID0_b
Der R, G und B Wert für die erste zu verwendende Zusatzblob Farbe

colID1_r, colID1_g, colID1_b
Der R, G und B Wert für die zweite zu verwendende Zusatzblob Farbe

colID2_r, colID2_g, colID2_b
Der R, G und B Wert für die dritte zu verwendende Zusatzblob Farbe

Dabei ist zu beachten, dass alle Parameter in einem Bereich von 0 bis 255 liegen müssen.

Wurden nun auf diese Weise die drei Zusatzblob Farben spezifiziert, erzeugt das Programm mit Hilfe der Cairo Bibliothek vier PNG Bild Dateien:

icro_patches_BLAU_page_0.png
icro_patches_BLAU_page_1.png
icro_patches_GELB_page_0.png
icro_patches_GELB_page_1.png

Diese enthalten alle elf Patches mit jeweils blauer und gelber Teamfarbe. Das Patchgenerierungsprogramm bietet somit also die Möglichkeit, auf sehr einfache Art und Weise Patches nach Bedarf zu modifizieren und zu generieren.

IV.4.6 Basteln von Patches

Bei diesem Ansatz wurde versucht bei der Produktion der neuen ICRO Patches möglichst viele Farben zu benutzen, die bereits in alten DROIDS Patches Verwendung fanden, und von denen bekannt war, dass sie gut funktionieren. Da diese alten Patches aus farbigen Papier- bzw. Pappstücken zusammengeklebt wurden, mussten somit die neuen Patches ebenfalls auf diese Weise gefertigt werden. Insgesamt wurden das Blau, das Gelb und das Grün der alten DROIDS Patches auch in den neuen weiter verwendet. Mit dem vorhandenen Cyan und Magenta waren wir hingegen nicht zufrieden. Nach der Erstellung und Evaluierung von entsprechenden Farbbögen entschieden wir uns schließlich, für einen Cyan Farbton vom Epson Drucker , sowie einen Magenta Ton vom HP Laserjet, mit denen jeweils eine DIN A4 Seite bedruckt wurde. Aus diesen Papier– bzw. Pappestücken wurden dann entsprechend große Kreise / Quadrate heraus geschnitten und diese dann auf ein schwarzes Quadrat aus Pappe mit 7,5 cm Seitenlänge geklebt. Da dies alles per Hand gemacht werden musste, wurde die Geometrie der Patches leider bei weitem nicht so genau, wie bei den gedruckten. Außerdem ist natürlich der Arbeitsaufwand wesentlich höher, als bei gedruckten Patches. Dafür wiederum können eben einige bereits bekannte und gute Farben weiterverwendet werden.

IV.4.7 Implementierung der ICRO Patches in Robosoccer

Die ICRO Patches wurden auf solche Weise in der BV6 implementiert, dass sowohl die alten als auch die neuen Patches genutzt werden können. Dabei kann die Umschaltung während des laufenden Betriebs erfolgen. Es sei allerdings angemerkt, dass die Farbklassen der drei ICRO Zusatzblob IDs momentan fest im Programm kodiert sind und nicht mittels GUI umgestellt werden können. Eine entsprechende Mutator Methode ist allerdings im Patchfinder bereits vorhanden.

IV.4.7.1 Aktivierung der neuen Patches

Die neuen ICRO Patches werden aktiviert, indem als Zusatzblob keine Farbe angegeben wird, sondern in der Combobox "Zusatzblob“ im Tab "HLS-Filter“ der Eintrag "ICRO“ ausgewählt wird.

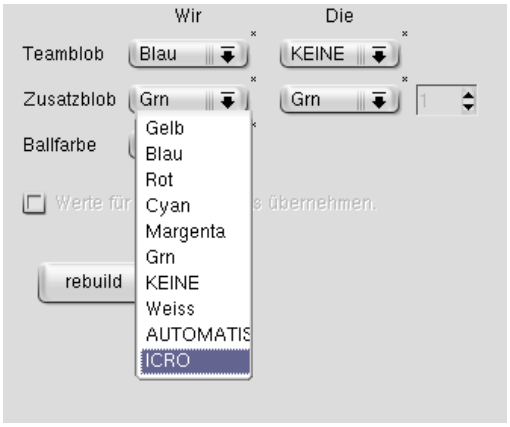


Abbildung IV.4-9: Aktivierung der ICRO Patches

Dadurch werden im Tracker, Mixer und Patchfinder boolesche Variablen gesetzt, die in die in entsprechenden Programmteile verzweigen. Außerdem wird in den farbigen Labels des RGB-Filters angezeigt, welche 3 Farbklassen (ID0, ID1, ID2) momentan vom ICRO Patchfinder benutzt werden.

Abbildung IV.4-10: Anzeige der gewählten ICRO Farbklassen im RGB-Filter

IV.4.7.2 Anpassungen im Tracker

In der momentanen Implementierung wird das Tracking bei Verwendung von ICRO Patches praktisch außer Kraft gesetzt. Es ist aber durchaus denkbar in künftigen Versionen das Tracking in modifizierter Form zwecks Fehlerkorrektur wieder einzuführen. Siehe Abschnitt 7.2.

IV.4.7.3 Anpassungen im Mixer

Aufgrund der Tatsache, dass die ICRO Patches alle eine eindeutige ID haben, kann der Mixer für den Zweikamera Modus sehr vereinfacht werden. Parallel zur bestehenden Methode "mixList" für die DROIDS Patches wurde hierfür eine Funktion "mixList_ICRO" implementiert, welches das Mixen bei der Verwendung der ICRO Patches übernimmt. Die grundlegende Vorgehensweise ist dabei folgende: Es liegen zwei Listen vor, welche alle eigenen Patches der linken bzw. der rechten Kamerahälfte enthalten. Diese beiden Listen werden nun zu einer verschmolzen: Für jeden Patch aus der rechten Kamerahälfte wird überprüft, ob ein Patch mit entsprechender ID auch in der linken Kamerahälfte vorkommt. Ist dies nicht der Fall, wird der entsprechende Patch in die zum linken Kamerabild gehörige Liste gepackt. Gibt es in der linken Kamerahälfte allerdings ebenfalls einen Patch mit dieser ID, dann wird der Patch aus dem rechten Bild ignoriert. Wurden allen Patches in der Liste für die rechte Kamerahälfte betrachtet, enthält die zum linken Kamerabild gehörige Liste alle eigenen Patches, die insgesamt auf dem Spielfeld vorkommen.

IV.4.7.4 Anpassungen im Patchfinder

Die umfangreichsten Änderungen, um Robosoccer ICRO Patch tauglich zu machen sind im Patchfinder zu finden. Im folgenden wird kurz auf die entsprechenden Modifikationen eingegangen. Für eine tiefer gehende Beschreibung des Programmablaufs sei auf den kommentierten Sourcecode in der Datei Patchfinder.cpp verwiesen.

IV.4.7.4.1 Verwendete Variablen

Um dem Patchfinder zu signalisieren, dass ICRO Patches verwendet werden, wurde eine boolesche Variable *friend2ColorICRO* eingeführt, welche mittels der Funktionen *useICRO* und *getICRO* gesetzt bzw. ausgelesen werden kann. Außerdem wurde ein Array *ColorOfICROID[3]* definiert, welches Elemente vom Typ *PixelFilter::ColorClass* enthält. In diesem Array werden die Farbklassen der drei verwendeten Zusatzblob Farben vermerkt. Momentan werden diese Werte fest im Konstruktor definiert:

ColorOfICROID[0] = PixelFilter::CC_GREEN;

ColorOfICROID[1] = PixelFilter::CC_CYAN;
ColorOfICROID[2] = PixelFilter::CC_MARGENTA;

Mittels der Methode *getICROIDColor* können die entsprechenden Werte ausgelesen werden. Außerdem existiert bereits eine Methode *setICROIDColor* mit der für jedes Arrayfeld die Werte einzeln gesetzt werden können.

IV.4.7.4.2 Implementierung des ICRO Patchfinders

In diesem Abschnitt wird auf die Methoden hingewiesen, welche vom ICRO Patchfinder verwendet werden. Dabei soll hier nicht zu sehr im Detail darauf eingegangen werden, wie die Methoden funktionieren. Vielmehr konzentrieren wir uns die Methoden in Bezug zu der in Kapitel 2 beschriebenen Vorgehensweise zu setzen.

FindRobPatchesICRO

Diese Methode wird aufgerufen, wenn ICRO Patches gefunden werden sollen (friend2ColorICRO == true). Als wichtigste Eingabeparameter erhält sie:

Teamlist	Eine Liste, welche alle Blobs in Teamfarbe enthält.
secondList	Eine Liste, welche alle restlichen gefunden Blobs enthält.
outputList	In diese Liste werden alle gefundenen Patches ausgegeben.

Die teamlist wird von Anfang bis Ende durchlaufen. Für jeden Blob in ihr werden mittels der Methode findAllCloseTo alle Blobs in der secondlist gefunden, welche sich innerhalb eines bestimmten Abstands zum momentan betrachteten Teamblob befinden. Dies entspricht der in Kapitel 2, Schritt 1 beschriebenen Vorgehensweise. Die so gefundenen Blobs werden anschließend sortiert und zwar aufsteigend nach Abstand zum Teamblob Mittelpunkt. Anschließend wird mittels getIDFromBlobList versucht die PatchID zu ermitteln.

getIDFromBlobList

Diese Methode wird innerhalb von findRobPatchesICRO aufgerufen. Die wichtigsten Übergabeparameter sind.

TeamBlob

Der momentan in findRobPatchesICRO betrachtete Teamblob

blobList
Alle Blobs, welche sich innerhalb eines bestimmten Radius um den TeamBlob finden.

Von der übergebenen Bloblist werden nur die ersten drei (also die zum Teamblob nächsten) Blobs betrachtet und in ein Array mit drei Feldern (first3Blobs) gepackt. Aus diesen drei Blobs wird nun versucht eine ID zu ermitteln. Dazu werden im weiteren Verlauf die Methoden CalcIndexOfID1 und SideOfPointToAVector aufgerufen. Wurden die Methoden abgearbeitet, ist bekannt welcher der Blobs aus dem Array first3Blobs, den Blob ID0, ID1 und ID2 darstellt, und somit ist auch bestimmbar, in welchen der drei in ColorOfICROID definierten Farbklassen, diese Blobs liegen (Farbklasse 0, 1, oder 2). Diese Information wird nun genutzt um mittels dem dreidimensionalem Array ColorIDToRobotID die PatchID zu bestimmten. ColorIDToRobotID enthält dabei für jede mögliche Farbkombination der drei ICRO Zusatzblobs den korrespondierenden ID Wert.

CalcIndexOfID1
Diese Methode wird innerhalb von getIDfromBloblist aufgerufen. Sie ermittelt, nach der in Kapitel 2, Schritt 3 beschriebenen Vorgehensweise, den Index des Blobs im Array first3Blobs, welcher den Zusatzblob ID1 darstellt.

SideOfPointToAVector
Diese Methode wird innerhalb von getIDfromBloblist aufgerufen. Nachdem mittels calcIndexOfID1 berechnet wurde, welcher der drei Blobs im Array first3Blobs den Zusatzblob ID1 darstellt, bestimmt sie, nach der in Kapitel 2, Schritt 4 beschriebenen Methode, welche ICRO Zusatzblobs (ID0 bzw. ID2) die noch verbliebenen zwei Blobs im Array darstellen.

IV.4.8 Die neuen Patches in der Praxis

Die ersten Tests des neuen ICRO Patches wurden zunächst an Standbildern durchgeführt. Die Patches wurden dabei stets gut erkannt. Dies galt sogar für Extremsituationen, in denen mehrere Patches unmittelbar nebeneinander lagen. Trotz der großen Nähe der Blobs zueinander, wurden ohne Fehler alle Zusatzblobs ihren jeweiligen Teamblobs zu gewiesen.

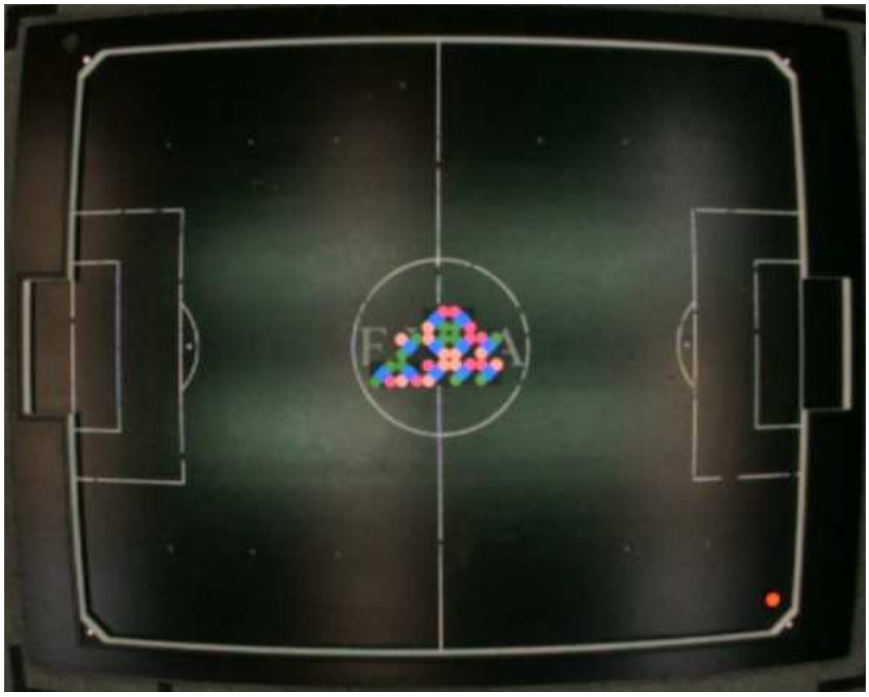


Abbildung IV.4-11Die Erkennung funktioniert auch in dieser Extremsituation.

Sogar in sehr unwahrscheinlichen Situationen, in denen ein Zusatzblob von einem anderem Patch überdeckt wird, funktionierte die ID Erkennung sehr gut. Dies liegt vor allem an der oben bereits erwähnten Vorgehensweise des Patchfinders, bei denen potentielle Zusatzblobs nicht anhand ihrer Mittelpunkte sondern anhand des Abstands des nächsten Pixels zum Teamblob ermittelt werden.
Im tatsächlichen Spielbetrieb zeigten sich die ICRO Patches ebenfalls der Aufgabe gewachsen. Auch bei relativ hohen Geschwindigkeiten und der damit einhergehenden mehr oder weniger starken Verwischung von Patches wurden die Identitäten zuverlässig erkannt.

Abbildung IV.4-12Verwischung bei hohem Shutterwert

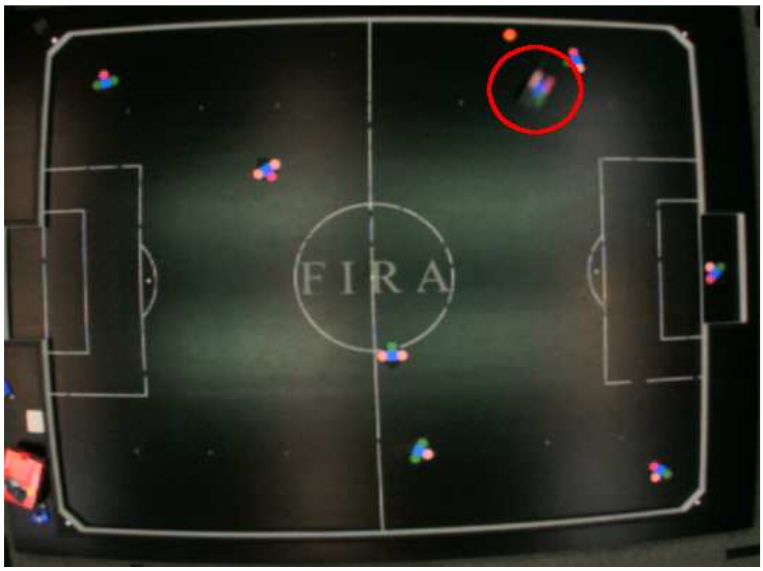


Abbildung IV.4-13Verwischung bei niedrigem Shutterwert

Als besonders positiv fiel auf, dass keine Vertauschungen auftraten. Kann bei den DROIDS Patches schon ein kurzes Stocken des Systems (und damit auch des Trackers) zu einer Fehlzuordnung oder Vertauschung führen, wurde dieses Phänomen – trotz diverser System bedingter Pausen – bei den ICRO Patches nicht beobachtet. Alles in allem verliefen die Praxis Tests sehr positiv.

IV.4.9 Weitere Verbesserungsmöglichkeiten

Wie die Tests bereits verdeutlichen, sind die ICRO Patches in ihrer jetzigen Form durchaus praxistauglich. Im Laufe der Entwicklung kamen allerdings einige Ideen für Verbesserungsmöglichkeiten auf, die allerdings aufgrund von Zeitmangel und terminlichem Druck nicht realisiert werden konnten. Darum wird im folgenden kurz auf Möglichkeiten hingewiesen, mit denen das bestehende System noch weiter verbessert werden kann.

IV.4.9.1 Automatische Farbwahl

Die in Kapitel 4 erwähnten Schwierigkeiten bei der Farbwahl, könnten durch ein automatisches Verfahren behoben werden. Hierfür könnte das System mittels Aufnahmen von einem oder mehreren Farbbögen, diejenigen Farben bestimmen, welche im RGB Farbraum den maximalen Abstand voneinander haben, und diese dann als Zusatzfarben wählen.

IV.4.9.2 Anpassung des Trackers

Momentan wird der Tracker bei Verwendung von ICRO Patches übersprungen. Dieses funktioniert, weil die ICRO Patches alle eine eindeutige ID haben, und es somit immer möglich ist zu bestimmen, wo welcher Roboter steht. Dennoch ist es denkbar, dass ein Tracking in gewissen Situationen auch bei den ICRO Patches Sinn macht. Z.B könnten im seltenen aber - aufgrund falsch klassifizierter Zusatzblob Farben - möglichen Fall, dass eine ID doppelt vergeben wurde, die korrekten ID Werte berechnet werden, indem die aktuellen Patchpositionen mit vorherigen Roboterpositionen verglichen werden. Auch in Phasen in denen die Patch ID Bestimmung aufgrund zu starker Verwischungen nicht mehr zuverlässig arbeitet, erscheint eine zusätzliche Entscheidungshilfe durch Tracking sinnvoll.

IV.4.9.3 Erhöhte Toleranz gegenüber Fehlern

Momentan reagiert das System nicht auf überflüssige Zusatzblobs. Denkbar wäre z.B ein Verfahren, dass versucht fälschlich erkannte Zusatzblobs (Phantomblobs), welche durch Verwischung oder sehr ungünstiges Licht in eine falsche Farbklasse eingeordnet wurden. Auch wäre eine Erweiterung sinnvoll, die Teamblobs, für welche keine ID ermittelt werden konnte, automatisch die ID zuweist die übrig ist.

IV.4.9.4 Farbkallibrierung beim Druck

Die bereits beschriebenen Schwierigkeiten beim Drucken der Patches könnten durch Verwendung von Farbkallibrierung behoben werden. Im Idealfall wären damit gleich farbige Patches auf allen Druckern realisierbar.

V. Gruppe Roboter-Hardware und Sensorik

Sarah Koenig,
Rainer Oye

V.1 Einleitung

Ursprüngliche Aufgabe der Gruppe Roboterhardware und Sensorik war es, Beschleunigungssensor und Kompass auf dem Roboter anzusprechen und damit Ausrichtungsdaten zur Verfügung zu stellen, die dem Roboter bei 2 Dingen helfen sollen:

- Ausgleich von Richtungsungenauigkeiten während einer Fahrt.
- Unterstützung der Berechnung eines Weltmodells, das zum Beispiel aus Kameradaten errechnet wird.

Die Gruppe konnte die zu Anfang gesteckten Ziele nicht vollständig erreichen, da die benötigten Roboter selbst zum Abschluss des Projektjahres nicht in der ursprünglich geplanten Form zur Verfügung standen. Die ständigen Änderungen an der wenigen verfügbaren Hardware machten die Entwicklung und besonders das Testen schwierig und vor allem sehr aufwendig.
Die Ziele sind aufgrund dieser Umstände mehrfach angeglichen worden.
Auch Ereignisse wie die Vorführung des Roboter-Teams an der Berufsschule Werne, ein Campustag an der Uni oder die WM in Singapur, für die kurzfristig sämtliche Roboter komplett montiert und verschiedene andere Dinge gebaut werden mussten, bedeuteten für die Hardwaregruppe Arbeit und Aufwand, der zeitlich bewältigt werden wollte. Dies alles diente allerdings auch zur Manifestierung des Umgangs mit dem System, besonders mit den Robotern.

V.2 Ziele und Motivation

Die endgültig gesteckten Ziele waren die folgenden:

1. Implementierung einer Schnittstelle für den I2C-Bus, die im üblichen Spielbetrieb des Roboters
verwendet werden kann.
2. Entwicklung einer Bibliothek zur Steuerung und Abfrage eines Kompass-Sensors, der an dem I2C-Bus angeschlossen ist.
3. Bereinigen und Kommentieren der Roboter-Programmquellen.

Die Projektgruppe zielt darauf, dass dieser Robotertyp eine gewisse Autonomie erhält, die immer weiter vervollständigt werden soll. In diesem Zusammenhang sollte auch der Kompass-Sensor getestet werden und festgestellt werden, inwieweit er den Roboter zum Treffen eigener Entscheidungen unterstützen kann.
Punkt 3 ergab sich aus der Tatsache, dass die Quellen im Laufe der Zeit so unübersichtlich geworden sind, so dass teilweise sogar unnötige oder redundante Berechnungen durchgeführt wurden. Das alles führte mittlerweile zu Performance-Problemen und auch zu Problemen bezüglich der Speichergrößen.

V.3 Abläufe auf dem Roboter

Um die Messung von Kompassdaten sinnvoll in den normalen Ablauf der Robotersteuerung zu integrieren, war eine Einarbeitung in die Funktionsweise der Steuerung notwendig. Die Erkenntnisse aus dieser Einarbeitung sind im folgenden Absatz zusammengefasst.

V.3.1 Scheduler und Timings

Zurzeit wird für das Scheduling eine einfache Endlosschleife verwendet, in der über einen Realtime-Interrupt das genaue Timing für die Aufrufe der Bewegungssteuerung des Roboters geregelt wird.
Beim Starten der Roboter wird der Code aus dem Flash-Speicher in den RAM geladen und ausgeführt. (Beim Testen und Debuggen über das JTAG-Interface aus dem RAM).
Nach der Initialisierung aller Module wird die scheduler-Schleife aufgerufen, eine while(1){…} Schleife, die also immer wieder durchlaufen wird, solange der Roboter eingeschaltet ist.
In der scheduler-Schleife erfolgen die Aufrufe für die komplette Steuerung des Roboters. Es wird zunächst der Funk abgefragt und geprüft, ob der Roboter eventuell einen neuen Befehl vom Hostsystem erhalten hat. Dann wird durch einen Aufruf der Methode doControl() der aktuell vorliegende Funkbefehl verarbeitet, und die Regelung der Motoren aufgerufen. Außerdem wird der Kompass abgefragt und die Leuchtdioden werden geschaltet.
Hier kann man keine 100% verlässlichen Annahmen machen, mit welcher Frequenz die scheduler-Schleife aufgerufen wird. Dies hängt auch von der eingestellten Taktfrequenz des DSP ab. Um die Aufrufe von doControl() und der Kompasskommunikation dennoch genau zu timen, wird ein **Realtime-Interrupt** benutzt: Ein CPU-Timer löst jede Millisekunde einen Interrupt aus, der eine Service-Routine startet. In der Interrupt-Service-Routine (ISR) wird die globale Variable rt_flag erhöht. Diese Variable wird bei jedem Durchlauf der Scheduler-Schleife abgefragt – nur wenn sie mindestens 1 ist, wird doControl() aufgerufen. Dadurch wird diese Methode jede Millisekunde aufgerufen. Auf diese Weise hat die Steuerung einen zeitlichen Anhaltspunkt, mit dem sie Geschwindigkeiten, zurückgelegte Entfernungen und ähnliches berechnen kann.

Um das rt_flag einzustellen, muss der CPU-Timer 0 entsprechend initialisiert werden. Dies geschieht in init_interrupthandler() (aus dem Modul intrupt.c) mit dem folgenden Aufruf der CPU-Timer-Konfiguration.

```
ConfigCpuTimer(&CpuTimer0,
       (int)(CLK_EXTERN_MHZ*(CLK_PLL_MULT/2.0)),TIMER_INTERVALL*1000)
```

Das erste Argument ist ein Funktionszeiger auf die Interrupt-Service-Routine (ISR). Das zweite Argument teilt der Routine die Taktfrequenz mit, auf die der DSP eingestellt ist. Über das dritte Argument, das Zeitintervall des CPU-Timers in Mikrosekunden, lässt sich die Frequenz des Realtime-Interrupts einstellen. Die Konstante TIMER_INTERVALL ist auf 1 eingestellt (siehe globals.h).

Um zu sichern, dass die Regelung nicht zu selten aufgerufen wird, ist dafür zu sorgen, dass die Ausführung aller Methoden in der Scheduler - Schleife gemeinsam nicht länger als eine Millisekunde dauert, denn dann würde rt_flag erneut erhöht, ohne dass es abgefragt wurde. Außerdem ist darauf zu achten, dass die ISR selbst nur sehr wenige Berechungen enthalten darf, da sonst ebenfalls das Timing gestört werden kann.

Die Gruppe entwickelte ein kleines Konzept für das Timing verschiedener Module deren Aufruf mit unterschiedlichen Frequenzen durch eine Modulo-Rechnung realisiert wird.

```
 Beispiel: if ((ctrl_counter % 1000) == 0)
{
        if (l==2)
               l=3;
        else
               l=2;

        set_led(l);
}
//      set_led(LED_OFF);
```

Es besteht die Möglichkeit, dass eine Modulo-Rechnung unnötig viel Rechenzeit verbraucht, vor allem wenn sie für schnellere Timer sehr oft durchgeführt wird. Andererseits wird die Modulo-Rechnung mit einer Zahl der Form 2^n auf dem von uns verwendeten DSP effizient durchgeführt, da es sich dabei um einen einfachen Vergleich handelt. Zurzeit wird das Timing jedoch mit verschiedenen globalen Variablen durchgeführt. Details dazu folgen im Kapitel über den I2C-Bus.

V.3.2 Alternativer Ansatz für das Timing auf dem DSP

Einen alternativen Ansatz hierzu könnte das Realzeit-Betreibssystem DSP/Bios darstellen, das von Texas Instrument für den zurzeit verwendeten TMS320F2812 zur Verfügung gestellt wird. Die Umstellung des Systems auf DSP/Bios wäre sehr aufwändig, bietet aber die Möglichkeit, den DSP optimal für ein Realzeitsystem zu nutzen.
Vorteile des Realzeit-Betriebssystems DSP/BIOS:

- Durch den Einsatz von Multithreading kann die Leistung des DSP voll ausgenutzt werden.
- Das Realzeit-Verhalten der Roboter kann mit Hilfe von Tools analysiert werden. Die Abhängigkeit verschiedener Komponenten und Berechungen kann mit Hilfe von verbundenen Threads sinnvoll und übersichtlich dargestellt werden.
- Der Code lässt sich leichter auf neue DSP's der TMS320x-Reihe übertragen, da durch den Einsatz von DSP/BIOS relativ stark von der Hardware abstrahiert wird.

Dieser Ansatz sollte eventuell in zukünftigen Projektgruppen weiterverfolgt werden. (TI SPRA 782).

V.4 Bereinigung und Dokumentation der Programmquellen

Aufgrund der Vorbereitungen für die Weltmeisterschaft in Singapur stand zu Beginn des Wintersemesters die Hardware zum Testen der Sensoren nicht zur Verfügung. Daher wurde als neues Ziel die Bereinigung, Optimierung und Kommentierung des Robotercodes beschlossen. Dies sollte zum einen das Einarbeiten nachfolgender Gruppen erleichtern, zum anderen sollte die Rechenzeit optimiert werden.

Außerdem sollte der Einsatz der IQ-Math-Library zur Verbesserung der Effektivität der Fließkommaberechungen abgewogen werden.

V.4.1 Konzept für die Bereinigung der Programmquellen

Anfangs wurde die Anpassung der Fließpunktberechnungen im Code an die IQ-Math-Library in Erwägung gezogen. Hierbei handelt es sich um eine Bi B bliothek von Texas Instruments, die Fixpunktformate statt der Fließkommazahlen des IEEE 754 Standards benutzt, mit welchen wesentlich effizienter gerechnet werden kann. (Quelle: IQmath.pdf) Nach genauerer Betrachtung des Codes wurde deutlich, dass die Übersicht und Effektivität zunächst durch Vereinfachen der Berechnungen verbessert werden sollte, bevor der Einsatz von IQ-Math sinnvoll ist. Viele der Fließpunktberechnungen im Robotercode sind historisch gewachsen und daher oft umständlich oder überflüssig. Andere werden aktuell noch nicht benötigt, sondern werden erst im Zusammenhang mit zukünftigen Änderungen benutzt und sollten daher im normalen Code nicht vorkommen. Da die

meisten Berechungen unzureichend kommentiert waren, erwies es sich als schwierig, zu unterscheiden, welche davon benötigt werden und welche nicht.

Die Bereinigung der Programmquellen wurde nach den folgenden Richtlinien vorgenommen:

- Wenn einer Berechung eine physikalische Formel zugrunde liegt, sollte diese im Kommentar angegeben werden. Insbesondere gilt das für den Fall, in dem mehrere Berechungen zusammengefasst oder experimentelle Konstanten benutzt werden.
- „Magische“ experimentelle Konstanten sind als solche zu kennzeichnen.
- Für jede Datei sollten alle printf über ein #define debug ausschaltbar sein. (Erklärung siehe „Optimieren von Codegröße“)
- Der Code für irgendwelche zukünftigen Konzepte sollte nicht in der normalen Codeversion stehen, bzw. gekennzeichnet sein und über defines komplett „ausschaltbar sein“.

V.4.2 Optimieren der Codegröße

Weitere Hinweise zum verkleinern der Codegröße finden sich unter V.8.2. An dieser Stelle soll zunächst allgemein auf das Konzept zur Optimierung des Codes eingegangen werden.
Da der Robotercode mit der Zeit immer wieder von verschiedenen Gruppen bearbeitet wurde, enthielt er viele unübersichtliche, historisch gewachsene Programmteile. Daher bestand eine der Hauptaufgaben zunächst aus der Einarbeitung in den Quellcode. Aufgrund der häufig unzureichenden Dokumentation konnte es nicht gelingen, alle Berechnungen zu erläutern. Viele der Formeln basieren auf Erfahrungswerten oder wurden so zusammengefasst, dass sie sich nicht mehr zweifelsfrei auf eine bekannte physikalische Formel zurückführen ließen.
Generell hat die Gruppe versucht, den Code von allen überflüssigen Berechnungen und Variablen zu befreien. Hierbei ging es zum einen darum alte, nicht mehr benötigte Teile zu entfernen. So enthielt zum Beispiel die Datenstruktur für die Rampenberechung eine Vielzahl von Werten für verschiedene Berechnungen. Da sich in der Praxis jedoch eine einfache Rampenberechnung als sinnvoller erwies, werden diese Werte nicht mehr benötigt, so dass die Datenstruktur verkleinert werden konnte. Zum anderen mussten auch die Berechnungen aus dem Code entfernt werden.
Einige Konzepte, wie zum Beispiel struct sensorfeedback wurden im Code belassen, obwohl sie in der Form zurzeit nicht benutzt werden. Dem lag der Gedanke zu Grunde, dass in Zukunft verschiedene Sensordaten auf dem DSP gesammelt und ausgewertet werden.

V.4.3 Übersicht über den Code

Im Folgenden werden die verbliebenen Module des Robotercodes aufgelistet und kurz erläutert. Für ein detaillierteres Verständnis wurden die Programmquellen - soweit möglich – umfassend kommentiert.

V.4.3.1 Funk:

Auf den V3-Robotern gibt es aktuell 2 Funkversionen: Funk4 und Funk5. In Zukunft wird es mit Funk6 eine weitere Funkversion geben. Die Funkversionen benutzen sowohl unterschiedliche Hardware als auch verschiedene Module des Robotercodes. Die benutzte Funkversion wird in der Datei globals.h mit einem #define eingestellt.
Für alle Versionen des Funks werden die Methoden von **uart.c** genutzt, um auf Hardware-Ebene Daten zu senden oder zu empfangen.
Funk4 benutzt dfrsm.c (deterministic finite receiver state machine).
dfrsm.c stellt die Software-Ebene für Funk 4 dar. Hier werden unter anderem die empfangen Daten decodiert. Je nachdem, was für ein Befehl empfangen wurde, wird die entsprechende Methode aus control.c aufgerufen. Zum Beispiel wird doDeltaDreh(winkel) aufgerufen, wenn ein Drehbefehl empfangen wurde. Ausgeführt wird die Drehung erst durch den Aufruf von doControl().
Für Funk5 wird eine andere Funkhardware genutzt, so dass die Funktionalität der statemachine direkt in der Hardware realisiert ist. Hierbei stellt **commander.c** die Software-Ebene für Funk5 dar, auf der wie oben beschrieben die Methoden aus control.c aufgerufen werden. **funkmodul.c** ist die Schnittstelle zwischen uart.c und commander.c.

V.4.3.2 Bewegungssteuerung

Beteiligte Module sind die folgenden:
Pid.c Enthält die PID-Regelung, wird in WheelspeedToPWM() aufgerufen.
Pwmmot.c regelt die Motorsteuerung auf Hardwareebene.
Rampe1.c Begrenzt die Beschleunigung, so dass der Roboter sauber anfährt
Speed1.c Rechnet Sollgeschwindigkeiten in PWM-Werte um.
Winkel2.c Berechnet die Winkelbeschleunigung
Quadimp.c In diesem Modul werden die Radencoder ausgelesen.

Control.c Enthält die Steuerung:
In der Regel erhält der Roboter einen Befehl über Funk, der die Motoransteuerung betrifft. Dabei kann es sich um einen der folgenden Befehle handeln:

- Stop
- Kurvenfahrt mit einer angegebenen Geschwindigkeit und einem angegebenen Winkel
- Setzen der Radgeschwindigkeiten auf die angegebenen Werte.

Dem Funkbefehl entsprechend wird eine der Methoden doStop, doDeltaDreh, doDeltaSpeed oder doWheelspeed aus control.c mit den entsprechenden Parametern aufgerufen.

In diesen Methoden wird der gesetzte Befehl in eine Variable geschrieben und die entsprechende Rampe – also die Beschleunigungskurve - ausgewählt. Außerdem werden einige Umrechnungen bezüglich der Winkel und Geschwindigkeiten vorgenommen. Die Abarbeitung des aktuellen Befehls erfolgt schrittweise in den regelmäßigen Aufrufen von doControl im Scheduler. Wenn der Roboter zum Beispiel einen Drehbefehl erhalten hat, wird bei jedem Aufruf von doControl berechnet, um wie viel sich der Roboter noch drehen muss, bis er die entsprechende Drehung ausgeführt hat. Dem entsprechend werden die PWM-Werte für die Motoren gesetzt. Falls vor Abschluss der Drehung ein anderer Befehl gesetzt wird, bricht der Roboter die Drehung ab und führt den anderen Befehl aus. Analog verhält sich der Roboter auch bei den anderen Befehlen. Auf die genaue Funktionsweise von doControl wird später genauer eingegangen, da es sich hierbei um das Herzstück der Steuerung handelt.

Die verschiedenen Methoden bewirken – wie gesagt nur im Zusammenspiel mit dem regelmäßigen Aufruf von doControl – folgendes:
(Im Code finden sich detaillierte Beschreibungen der Argumente.)
doStop: Die PWM-Werte beider Motoren werden auf 0 gesetzt, so dass der Roboter ausrollt. Dabei werden die Motoren ausgeschaltet, so dass keine kontrollierte Bewegung mehr stattfindet. Um den Roboter aktiv abzubremsen und ihn auf einer Position verharren zu lassen muss doWheelSpeed(0,0,0) aufgerufen werden. Das aktuelle Kommando ist COMMAND_STOP
doDeltaDreh (winkel): Der Roboter wird um den angegebenen Winkel gedreht. Hierbei sind sowohl Drehungen nach rechts (positive Winkel), als auch nach links (negative Winkel) möglich. Im Vergleich zu doDeltaSpeed () wird hier eine größere Rampe, also eine steilere Beschleunigungskurve gewählt. Das Kommando wird auf COMMAND_DELTADREH gesetzt.
doDeltaSpeed (direction, speed, delta): Hierbei werden die Werte so berechnet, dass der Roboter mit der angegebenen Geschwindigkeit fährt und sich dabei um den Winkel delta dreht. Das aktuelle Kommando wird auf COMMAND_DELTASPEED gesetzt.
doWheelSpeed (left, right, drehkick): Die Werte werden so berechnet, dass die Räder die jeweils angegebene Geschwindigkeit erreichen. Falls drehkick = 1, falls also ein Drehkick ausgeführt werden soll, wird hierzu eine steilere Rampe gewählt. Das gesetzte Kommando ist COMMAND_WHEELSPEEDS

Zu den Berechnungen der Geschwindigkeit ist folgendes zu beachten:
Wenn die Radgeschwindigkeiten über doWheelSpeed() gesetzt werden, wird je ein Wert für die Geschwindigkeit des rechten und des linken Rades übergeben. In der Methode werden diese Daten umgerechnet in wheelSpeedT und wheelSpeedR. Hierbei ist die T immer die Grundgeschwindigkeit des Roboters, also die Geschwindigkeit, mit der sich der Robotermittelpunkt bewegt. R ist die

Raddifferenzgeschwindigkeit, also der Wert um den sich die Geschwindigkeit eines Rades von der Geschwindigkeit des Mittelpunktes unterscheidet. Dieser Wert ist immer für beide Räder gleich, da T den Durchschnitt der beiden Radgeschwindigkeiten darstellt. Wenn der Roboter geradeaus fährt, ist R = 0.
Beim Aufruf von doDeltaSpeed werden nicht die einzelnen Radgeschwindigkeiten übergeben, sondern die Grundgeschwindigkeit T zusammen mit einem Winkel.
Zu der Berechnung der Winkel ist im Allgemeinen folgendes zu sagen: Die Winkel sind immer relativ zur „Blickrichtung“ des Roboters gesehen. Es sind Winkel aus dem Bereich [-2π, 2π [erlaubt, hierbei stehen negative Winkel für eine Drehung nach links. Bei den alten Robotern war nur eine Drehung bis 127 Grad in jede Richtung implementiert. Auf den neuen Robotern ist nun eine ganze Drehung in jede Richtung möglich.
An verschiedenen Stellen im Code bezeichnen negative Winkel und negative Kreisradien Drehungen beziehungsweise Krümmungen nach links, entgegen der mathematischen Konvention. Das ist zwar ungewöhnlich, da auf dem Hostrechner aber ebenfalls so gerechnet wird, ist es nicht sinnvoll, dies im DSP-Code zu ändern.

Grundsätzlich berechnet doControl() aus Winkeln und Geschwindigkeiten (Robotergeschwindigkeit) PWM-Werte für die Motoransteuerung.
Für die Funktionsweise der Steuerung mit Hilfe von doControl () muss zunächst noch die Methode **simulateMovement()** genauer erläutert werden. Diese Methode wurde durch Christian Kintzel optimiert, da in der alten Version einige nicht mehr benötigte Werte berechnet wurden, und daher ein kürzerer Rechenweg gewählt werden konnte. In der aktuellen Version wird nun im Wesentlichen nur noch der Kurvenradius des gefahrenen Kreises berechnet.
Zunächst wird aus den Rückgabewerten der Radencoder die zurückgelegte Strecke seit dem letzten Aufruf der Funktion berechnet. Hieraus wird wie folgt der Kurvenradius berechnet:

```
 /*
Kurze Herleitung der Formel für den Kurvenradius:

\omega: Winkelgeschwindigkeit
l = Strecke linkes Rad
r = Strecke rechtes Rad
v = Strecke auf der Kurvenbahn in mm
R = Kreisradius
W = Radabstand

\omega = v / R <=> R = v /  (\omega)
```

```
sin (\omega) = (l - r) / W

// \omega klein, dann sin(\omega) = \omega

=> \omega = (l - r) / W

v = (l + r) / 2

//Einsetzen, ausrechnen ergibt

R = W / 2.0 - r * W / (l -r)

Beachte: Rotationen sind im Uhrzeigersinn!!! (Nur damit keine Fragen
     kommen!)
 */

 //Abfrage verhindert eine Division durch Null bei gleichen Werten.
 if (distRight != distLeft)
 {
     kurvenRadius = WHEELDIST/2.0 - distRight * WHEELDIST / (distRight
     - distLeft);
 }
 else
 {
     /*
     Eine Gerade ist ein Kreis mit unendlichem Radius.
     Also Radius auf MAX_FLT setzen, wenn geradeaus gefahren wird.
*/

     kurvenRadius = 3.4028235e+38; // == MAX_FLT
 }
```

doControl()

Wie bereits erwähnt, wird diese Methode regelmäßig in der scheduler-Schleife aufgerufen. Bei jedem Aufruf der Methode doControl() werden zunächst die folgenden Werte zum aktuellen Zustand des Roboters ermittelt:

- PWM-Werte beider Motoren
- Wie weit haben sich die Räder seit dem letzten Aufruf von doControl() bewegt? Der Wert der zurückgelegten Strecke wird für die Berechnung des Kurvenradius in simulateMovement() benötigt.

- Wie weit haben sich die Räder seit dem letzten Realtime-Interrupt gedreht? Da der Zeitraum zwischen zwei Realtime-Interrupts immer gleich ist, kann aus diesem Wert die aktuelle Geschwindigkeit des Roboters berechnet werden.

Aus den ermittelten Werten wird in simulateMovement() die Bewegung berechnet, die der Roboter seit dem letzten Aufruf dieser Methode zurückgelegt hat. Insbesondere wird hier der Radius der Kurve berechnet, die der Roboter in diesem Zeitabschnitt gefahren ist.
Im Anschluss hieran werden die zu Setzenden PWM-Werte in mehreren Schritten berechnet, je nach Befehl fallen einige dieser Berechnungen weg. Für COMMAND_DELTASPEED, also eine Kurvenfahrt mit der Endgeschwindigkeit speed und eine Drehung um den Winkel delta, werden alle Schritte berechnet. Daher betrachten wir im Folgenden zunächst diesen Fall:

- Es wird berechnet, um wie viel sich der Roboter noch drehen muss, um die Drehung abzuschließen.
- Aus diesem Ergebnis wird berechnet, welche Geschwindigkeit die Räder hierfür erreichen müssen.
- In der Rampenfunktion werden die Radgeschwindigkeiten angepasst, so dass der Roboter möglichst gleichmäßig anfährt. (Die entsprechende Rampe wurde beim Setzen des Befehls ausgewählt, und wird in der Datenstruktur sensorfeedback übergeben)
- Unter Berücksichtigung der alten PWM-Werte und der PID-Regelung werden aus den Radgeschwindigkeiten aus 3. die neuen PWM-Werte berechnet.
- Die berechneten PWM-Werte werden gesetzt.

Je nachdem, um welchen Befehl es sich handelt, werden einige dieser Berechnungen ausgelassen. Das lässt sich gut in der folgenden switch-Anweisung nachvollziehen. (Man beachte, dass die einzelnen switch-Teile nicht durch break; getrennt werden, so dass alle Berechnungen nacheinander ausgeführt werden.)

```
switch (selectedCommand)
  {
    case COMMAND_DELTASPEED:

   /*si.DeltaWinkel ist der Winkel um den man sich seit dem letzten
      Aufruf gedreht hat. Die Drehung wird i.d.R. in mehreren Schritten
      durchgeführt und hier wird errechnet, um wieviel man sich noch
      drehen muss. sk..*/
     sollDeltaWinkel = normalisiere(sollDeltaWinkel - si.DeltaWinkel);

     /* Sollwinkel -> Radgeschwindigkeiten */
```

```
    Winkel2(&si,sollDeltaWinkel,sollSpeed,sollRichtung,&wheelSpeedT ,
     &wheelSpeedR);

  case COMMAND_WHEELSPEEDS:

  //sk.. in rampe1 werden die Radgeschwindigkeiten so korrigiert, dass
     der Roboter ohne zu ruckeln anfährt

    rampe1(&si,
          wheelSpeedT , wheelSpeedR,
          &wheelSpeedTAfterRampe , &wheelSpeedRAfterRampe
          );



  /* hier werden die korrigierten Radgeschwindigkeiten in PWM-Werte
     umgesetzt.
     In diesem Aufruf ist auch die PID - Regelung mit drin */

     WheelSpeedToPWM1(&si,
                   wheelSpeedTAfterRampe , wheelSpeedRAfterRampe,
                   &pwmLeft , &pwmRight);

    /* Flag zurücksetzen */
    si.LastCmdStop = 0;

  case COMMAND_STOP:
     //sk.. die oben berechneten, oder in doStop gesetzten PWM - Werte
     werden gesetzt
      setLeftPWM(pwmLeft,false);
      setRightPWM(pwmRight,false);
      newcmd = 0;

}
```

V.4.3.3 Timing

Beteiligte Module sind folgende:

Rtint.c

Intrupt.c

Scheduler.c

Zur Erläuterung des Timings siehe V.3.1

V.4.3.4 Weitere Module

Adc.c

Dieses Modul soll den A/D-Wandler initialisieren und die Betriebsspannung des DSP auslesen können. Auf dem aktuellen Board v3 ist ein Hardwarefehler. Daher ist der Code auskommentiert.

Globals.c

Hier werden Konstanten und einige globale Variablen gesetzt.

Hardware.c

In diesem Modul werden hardwarenahe Initialisierungen gesetzt.

Led1.c

Methoden zum Setzen der LED.

Main.c

Programmstart, Initialisierungen und Aufruf des Schedulers.

V.5 Der Kompass Sensor

Für den Einsatz des Roboters sollte ein Kompass Sensor des Typs HMC6352 der Firma Honeywell getestet werden.

V.5.1 Grundlagen

Der Sensor misst das Magnetfeld der Umgebung und gibt eine Ausrichtung zurück. Er unterstützt eine Kommunikation über einen I2C-Bus, über die er konfiguriert wird und über die die Daten abgefragt werden.

Der Sensor kann in einen so genannten "Continous"-Modus geschaltet werden, in welchem er Daten liefert wenn er einfach nur mit seiner Adresse angesprochen wird. Der Sensor liefert maximal 20 Messwerte pro Sekunde, arbeitet also maximal mit 20 Hz.

V.5.2 Umsetzung

Für den Sensor wurde eine relativ vollständige Bibliothek entwickelt, die den kompletten Funktionsumfang des Sensors abdeckt. Für den momentanen Betrieb wird diese Funktionalität nicht in vollem Umfang benötigt, aber für zukünftige Anwendungen ist das Modul auf diese Weise bereits vorbereitet und kann leicht ausgebaut werden.

Das Modul enthält Funktionen zur Initialisierung, über die der Operationsmodus und die Form der Datenausgabe konfiguriert werden kann. Bei Bedarf kann der Kompass in einen Sleep-Modus gesetzt werden, damit er weniger Strom verbraucht. Die Konfiguration kann im EPROM des Sensors gespeichert werden. Beim nächsten Einschalten des Sensors wird die gespeicherte Konfiguration in

die Arbeitsregister des Sensors geladen. Es können allerdings nicht alle möglichen Konfigurationen im EPROM abgelegt werden. Näheres dazu kann man dem Datenblatt (HMC6352.pdf) entnehmen.
Bei der Initialisierung wird der Kompasssensor in den Continous – Mode gesetzt, d.h. er antwortet auf ein Lesekommando, das lediglich eine Startsequenz und die Adresse des Kompasssensors schickt, mit dem aktuellen Messwert (2 Byte). Das ist bisher der einzige Modus, der für den vorgesehenen Betrieb benötigt wird.
Der Sensor muss initial einmal kalibriert werden. Dazu wird ein Kalibrierungsmodus eingeschaltet und der Sensor soll dann mehrere Male langsam und gleichmäßig um seine eigene Achse gedreht werden, mindestens zweimal und möglichst ohne den Sensor dabei zu verkippen. Das sollte mit einem gut konstruierten und gut eingestellten Roboter kein Problem darstellen. Leider sind die Roboter noch nicht optimal eingestellt, so dass die Kalibrierung eventuell kein optimales Ergebnis liefert.
Eine Routine für die Kalibrierung ist in die scheduler-Schleife eingebaut und auskommentiert, da sie im normalen Betrieb nicht laufen soll. Der Sensor sollte nicht jedes Mal beim Einschalten kalibriert werden, da die Lebensdauer des eingebauten EEPROM, in das bei der Kalibrierung geschrieben wird, durch eine gewisse Anzahl von Schreibvorgängen begrenzt ist.

V.6 Der I2C-Bus

V.6.1 Grundlagen

Der I2C-Bus ist eine bit-serielle Kommunikationsverbindung die auf der Verwendung von lediglich zwei Leitungen beruht, einer Datenleitung und einer Taktleitung. Dabei kann er laut Spezifikation mit einer Frequenz von bis zu 100kHz arbeiten. Er unterstützt mehrere Geräte, die über eindeutig definierte und eingestellte Adressen selektiert werden.
Hierbei wird die Kommunikation nicht dadurch gestört, wenn der Takt wesentlich langsamer als mit 100 kHz arbeitet oder sogar für eine gewisse Zeit angehalten wird. Der Takt ist kein regelmäßiger Takt im eigentlichen Sinne, er triggert lediglich die Aktionen auf der Datenleitung. Wenn keine Kommunikation stattfindet, braucht der Takt auch nicht gesetzt zu werden. Damit ist es hier sogar möglich, schrittweise zu debuggen.
An einem I²C-Bus können mehrere Teilnehmer angeschlossen sein, nach verfügbaren Adressen bis zu 127. Dabei gibt es einen Master, das ist meist der Prozessor, und einen oder mehrere Slaves. Die Taktleitung und die Datenleitung werden in der Regel durch den Master gesetzt. Wenn Daten von einem Slave, also zum Beispiel von unserem Kompasssensor, empfangen werden, gibt der Master die Datenleitung frei, so dass der Slave darauf zugreifen kann.

Der Master beginnt die Kommunikation durch Senden einer Startsequenz. Anschließend sendet er die Adresse des Slaves mit dem er kommunizieren möchte, zusammen mit der Information, ob es sich um ein Lese- oder ein Schreibkommando handelt.
Der Slave bestätigt den Empfang der Information durch Heruntersetzen der Datenleitung auf 0. Nach dem Empfangen der Daten beendet der Master die Verbindung durch das Senden einer Stopsequenz.
(Quelle: i2cspec.pdf)

V.6.2 Voraussetzungen

Ursprünglich sollte eine Implementierung der Vorgänger-PG zur Verfügung stehen, die entsprechend erweitert werden sollte. Die Gruppe stellte jedoch fest dass die vorhandene Implementierung kein durchgängiges Konzept enthielt und relativ undurchsichtig war. Zum Beispiel war nicht durchgängig und einheitlich festgelegt, in welchem Zustand sich die Leitungen befinden, wenn man in eine Funktion springt, damit waren die angebotenen Funktionen nicht universell einsetzbar. Die Gruppe entschied sich daher für eine komplette Neuimplementierung. Außerdem eignete sich die I²C-Implementierung der Vorgänger-PG nicht für den Normalbetrieb, da sie die Motorsteuerung stört.

V.6.3 Umsetzung

Es wurde ein komplett neues Konzept für die I²C-Kommunikation entworfen. Es entstand schrittweise, angepasst auf die Bedürfnisse und auch angepasst an die Umbauten der Roboter.

V.6.3.1 Grundsätzliches zum I²C-Konzept

Für den I²C-Bus wurde ein Konzept geschaffen, das eindeutige Zustände für die Implementierung festlegt. Da für eine Kommunikation verschiedene Funktionen durchlaufen werden, muss klar sein, welche Zustände vorzufinden sind wenn eine Funktion startet, oder in welchem Zustand eine Funktion den Bus zu hinterlassen hat, wenn sie sich beendet.

Leitung	SCL	SCL_DIR	SDA	SDA_DIR
Ruhezustand (also vor einem sendstart und nach einem sendstop)	High	Out	High	Out
während einer Kommunikation (also nach einem sendstart und vor einem sendstop oder zum Beispiel auch zwischen zwei Aufrufen der Funktion zum Senden von Bytes	Low	Out	(nicht definierbar)	Out

Tabelle V.6-1: Definition der Bus-Zustände des I²C für den Robotercode

Diese Definitionen erleichtern in jedem Fall die Implementierung und die Fehlersuche.

V.6.3.2 I²C-Statemachine

Bei der Testimplementierung wurden alle anderen Komponenten der Robotersteuerung zunächst außer Acht gelassen. Der Kompass ist mit dem DSP über einen I²C Bus verbunden, auf dem die maximale Taktfrequenz 100 kHz beträgt. Für die Abfrage eines Messwertes vom Sensor werden etwa 30 Bit gesendet bzw. empfangen. Der DSP ist in der Testimplementierung bei Ausnutzung der 100kHz also etwa 0,3 ms durch den Abruf der Daten blockiert. Da man mit einem Delay, der über eine einfache Zählschleife realisiert ist, jedoch nicht genau 100 kHz einstellen kann und den Wert des Delays zur Sicherstellung der Kommunikation erhöhen muss, kommt man sehr schnell in den Bereich mehrerer Millisekunden. Eine so lange Unterbrechung würde unter anderem die Funktionsweise der Motor-Steuerung stören.

Daher ist es im Normalbetrieb nicht möglich, den Kompass über die zunächst konzipierten Methoden anzusprechen. (Diese Methoden werden im Folgenden blockierende Methoden genannt). Außerdem soll keine wertvolle Rechenzeit des DSP mit Warten verbracht werden. Für die Initialisierung und Konfiguration des Kompass oder anderer Geräte ist die Verwendung der blockierenden Methoden dagegen unproblematisch.

Aus der Umsetzung der Timings auf dem Roboter durch eine Scheduler-Schleife ergab sich die Notwendigkeit einer State-Machine für den Normalbetrieb, die ohne Delays auskommt und die Motor-Steuerung nicht stört.

Da die Abfrage der Messdaten vom Kompass mit den blockierenden Funktionen zu lange dauert, kann dies nicht auf einmal geschehen, da man ansonsten einen oder mehrere Realtime-Interrupts für die Motorsteuerung verpassen würde. Im Gegensatz dazu müssen auch nicht jede Millisekunde Daten vom Kompass geholt werden, da dieser maximal 20 Messwerte pro Sekunde liefert.

Daher wurde eine Kommunikation entworfen die in Halbtakt-Schritten durchgeführt wird: bei jedem Aufruf der Steuerungsmethode i2c_doHalfCycle, führt der DSP einen halben Takt der Kommunikation aus. Dabei ist es natürlich notwendig, dass der Zustand und der Fortschritt beim Senden und Empfangen von Daten gespeichert werden, so dass die Kommunikation bei erneutem Aufruf von i2c_doHalfCycle an derselben Stelle fortgesetzt werden kann. (Diese Teile des Moduls werden im Folgenden nicht-blockierende Funktionen und Methoden genannt.)

Das I²C-Modul verfügt über einen Befehls-Puffer, der ein Kommando aufnehmen kann. Er besteht aus zwei integer-Arrays, wobei eines die Bitfolge der Daten aufnimmt, das andere die Bit-Sequenz, die auf die Takt-Leitung gesetzt wird. Der Puffer kann von einem Device-Modul wie zum Beispiel dem Kompass mit einer Bit-Sequenz gefüllt werden, die über den Bus geschickt werden soll. Ist der Puffer einmal gefüllt, kann das Kommando immer wieder neu gestartet werden, es muss nicht neu in den Puffer kopiert werden. Für einen gewünschten Befehl muss eine entsprechende Sequenz aus Halbtakten zusammengestellt werden. Für die Daten-Abfrage aus dem Continous-Modus des Kompasssensors sehen die Arrays folgendermaßen aus:

Takt	11	0101010101010101 01 0101010101010101 01 0101010101010101 01	011
Daten	10	0011000000001111 44 2222222222222222 00 2222222222222222 11	001

Start- und Stop-Sequenz und die Bestätigungen nach den einzelnen Bytes sind hier optisch ein wenig abgesetzt. Diese Zwischenräume haben keinerlei Bedeutung für einen zeitlichen Ablauf sondern dienen nur der Übersicht.

Erläuterung:
Die Sequenzen bestehen aus halben Takten, dies stellt sicher, dass zwischen dem Setzen der Datenleitung und dem Setzen der Taktleitung eine gewisse Zeit vergeht, so dass nicht durch eine langsam ansteigende Flanke Fehler entstehen, beim Lesen von Daten analog. Daher kommen die Datenbits immer doppelt vor. Ausnahme sind hier die Start- und die Stopsequenz.
Die Datenkodierung besteht nicht nur aus Bits, sondern enthält auch andere Werte, die zum Beispiel die Bedeutung "Daten lesen" haben. Es sind folgende Werte definiert:

0: Eine Null schicken.
1: Eine Eins schicken.
2: Datenbit lesen.
4: Bestätigung vom Slave holen.

Die Zustände der State-Machine beinhalten folgende Informationen:
Werden auf der Datenleitung zum betrachteten Zeitpunkt Daten oder Empfangsbestätigungen gesendet oder empfangen?
Welchen Wert hat der Master an der Taktleitung angelegt?
Falls Daten/Empfangsbestätigungen gesendet werden:
Welchen Wert hat der Master an die Datenleitung angelegt?
Wurden Daten empfangen?
Ist eine Sequenz abgearbeitet kann abgefragt werden, ob Daten verfügbar sind. Die Daten können dann abgeholt werden. Anschließend kann das Kommando neu gestartet oder ein anderes Kommando an den I²C geschickt werden. Während einer laufenden Kommunikation kann ein Kommando nicht neu gestartet und auch kein neues Kommando in den Puffer geschrieben werden. Diese Aktionen werden solange vom Modul blockiert, wie die aktuelle Sequenz noch bearbeitet wird, der Bus also noch belegt ist.
Um eine Vorstellung von der Dauer einer Befehlsbearbeitung zu bekommen erläutert der nächste Abschnitt wie das Timing des I²C in unserem Fall funktioniert.

V.6.3.3 Timing des I²C-Busses

Die scheduler-Schleife arbeitet im Millisekunden-Takt. Würde die Funktion i2c_doHalfCycle ebenfalls in diesem Takt arbeiten, würde die Befehls-Sequenz (siehe oben) mit ihren 59 Halbtakten in 59 Millisekunden abgearbeitet sein. Wenn der Kompass im 20 Hz Rhythmus arbeitet liegt alle 50 ms ein neuer Wert bereit. Es würden also nicht immer alle Werte verarbeitet, sondern ab und zu ein Wert ausgelassen.

Daher hat die Gruppe den Realtime-Interrupt um das 10-fache beschleunigt und eine zusätzliche globale Variable rt_flag10 in die Interrupt-Bearbeitung eingebaut, die damit im 10-kHz-Takt gesetzt wird Mit einem Interrupt - Takt von 10 kHz und der Realisierung über die Halbtakte erreicht der I²C - Bus hier eine Taktgeschwindigkeit von 5 kHz. Damit ist er noch lange nicht ausgelastet, für diese Anwendung ist das aber ausreichend, da so die Sequenz zur Abfrage der Kompasswerte in 5,9ms bearbeitet werden kann. Dieses Timing wurde zu Testzwecken gewählt, wobei zu beachten ist, dass die Frequenz eigentlich zu schnell ist, so dass der Kompass dann mehrfach dieselben Daten liefert. Die Abfrage kann bei Bedarf auch über einen Zähler in der scheduler-Schleife seltener gestartet werden.

Die ursprüngliche Variable rt_flag wird für die Motorsteuerung nach wie vor im 1-kHz-Takt gesetzt, also jede Millisekunde.

V.6.3.4 Konzept für die Verwaltung getrennter I²C-Busse

Auf dem neuen Board (V3) sind zwei I²C-Busse vorgesehen: An Bus 1 ist der Kompasssensor angeschlossen, Bus 0 wird für die Ansteuerung der Kamera und eines EPROMs verwendet. Außerdem bietet Bus 0 die Möglichkeit, eventuell weitere Devices an den Roboter anzuschließen. Da der zweite I²C-Bus zurzeit nur zur Initialisierung der Kamera genutzt wird, wurde die I²C-Statemachine in der aktuellen Codeversion zunächst nur für den Bus des Kompass realisiert. Die alten, blockierenden Funktionen können für beide Busse unabhängig voneinander genutzt werden, daher wurden diese Methoden um das Argument der Busnummer erweitert.

Die Idee dieser Änderung war, dass die State-Machine keine konkurrierenden Teilnehmer berücksichtigen muss. Damit sollte Verwaltungs-Overhead und damit wertvolle Rechenzeit in der Scheduler-Schleife gespart werden.

Es ergab sich allerdings ein Problem beim Design des V3-Boards. Der Kompass-Bus wurde an zwei Pins angeschlossen, deren Register auch für die Timer der Motorsteuerung benutzt werden. Die meisten general purpose-pins am TMS321f2812 sind mit einer doppelten Funktion versehen: Zum einen können sie als digitale I/O-Pins benutzt werden, so wie es für den Anschluss des Kompasssensors vorgesehen ist. Zum anderen haben die Pins jeweils eine feste Funktionalität.

Das Umschalten der Funktionalität der Pins ist über Multiplexer realisiert. Möchte man einen Pin also als digitalen I/O Pin nutzen, muss dieser dementsprechend initialisiert werden. Das entsprechende

Register ist geschützt, um unbeabsichtigtes Überschreiben der Werte zu verhindern. (Hierbei müssen X und NAME dem gewählten Pin entsprechend gesetzt werden)

```
EALLOW;//Schreibschutz der Register aufheben, die  EALLOW-protected sind
GpioMuxRegs.GPXMUX.bit.NAME = 0;
EDIS;// Register sind jetzt wieder schreibgeschützt
```

Über ein weiteres Register wird bestimmt, ob der digitale Pin als Eingang oder Ausgang arbeitet:

```
EALLOW;
GPXDIR.bit.NAME = 1 // 1 setzt den Pin auf Output, 0 auf Input
EDIS;
```

Über das folgende Register können Werte auf den Pin gesetzt werden, wenn er auf Output geschaltet ist. Falls der Pin als Input arbeitet, kann in diesem Register (nicht EALLOW-protected) ausgelesen werden, welcher Wert anliegt.

```
GpioDataRegs.GPXDAT.bit.NAME
```

Die zunächst für den Kompasssensor eingeplanten GPIO Pins D0 und D1 beeinflussen in ihrer zweiten Funktionalität das Verhalten der Event-Manager, die für die Motorsteuerung zuständig sind. Die Funktionalität der Pins lässt sich zwar umschalten, jedoch gibt es selbst dann noch Seiteneffekte auf die Eventmanager, wenn die Pins auf GPIO geschaltet werden.
Da diese Zusammenhänge in den Dokumentationen von TI nur unzureichend beschrieben sind, lassen sich die Pins nicht ohne größeren Aufwand wie geplant benutzen. Um die anstehenden Tests durchführen zu können wurde provisorisch an einen Roboter ein Kompass an den ursprünglich geplanten Bus angeschlossen. An diesem Bus befinden sind auch die übrigen Geräte. Dies sind die Kamera-Konfiguration und das EEPROM. Daher ergeben sich bei der aktuellen Realisierung der Bibliothek Einschränkungen für die Nutzung der Schnittstelle, die der Programmierer zu beachten hat:

- Der Programmierer muss darauf achten, dass der Bus innerhalb der Scheduler-Schleife nur von einem Gerät verwendet wird, da es sich auf dem I²C-Bus nur aus dem jeweiligen Kontext ergibt, von welchem Slave die empfangenen Daten stammen.
- Solange eine Kommunikation läuft, darf keine der blockierenden Funktionen gestartet werden. Diese Funktionen fragen nämlich nicht ab, ob der Bus belegt ist.

Zur Konfiguration und Initialisierung über die blockierenden Funktionen können jedoch alle Geräte ohne Einschränkungen angesteuert werden.

V.7 Ergebnisse der Tests des Kompasssensors

Die Gruppe konnte experimentell feststellen, dass das Magnetfeld der Umgebung nicht sehr gleichmäßig ist, sondern an verschiedenen Orten stark unterschiedlich sein kann. Diese Änderungen des Umgebungsmagnetfeldes hängen mit verschiedenen Faktoren zusammen, zum Beispiel mit der Beschaffenheit von Gebäuden oder mit Leitungen, durch die starke Ströme fließen. In Einzelfällen kann das Feld sogar um bis zu 180 Grad gedreht sein, so dass Daten eines Kompass, der Magnetfelder misst, völlig unbrauchbar sind. Für eine exakte Navigation des Roboters auf dem Spielfeld lässt sich der Sensor daher vermutlich nicht einsetzen. Sogar der weit weniger ehrgeizige Ansatz, bei dem die Kompassdaten zusammen mit den Bildern der lokalen Kamera nur zur Unterscheidung der eigenen von der gegnerischen Spielhälfte dienen, scheint sich nicht ohne weiteres umsetzen zu lassen.

V.7.1 Kompasstests

Es wurden verschiedene Tests mit dem Kompass durchgeführt. Die bedeutsamsten werden im Folgenden näher erläutert.

Test 1:
Es wurde eine Routine entwickelt, die den Roboter in eine Richtung mit einem Wert zwischen 0 und 4 Grad ausrichtet. Der Roboter versucht die Ausrichtung mit einer möglichst kleinen Drehung zu erreichen, das heißt er dreht entweder rechts oder links herum, je nachdem, welches die kürzere Distanz ist.
Mit dieser Routine wurden verschiedene Tests gemacht.
Der Test wurde an verschiedenen Orten im Gebäude an unterschiedlichen Positionen aus wiederum verschiedenen Ausgangspositionen durchgeführt.
Dabei wurde festgestellt:
Der Roboter findet eine Ausrichtung an einem Ort zuverlässig wieder, bis auf wenige Grad genau (eine genauere Ausrichtung ist bei aktueller Einstellung der PID-Regler für die Motoren nicht möglich, auch nicht bei langsamer Fahrt)
Er findet diese Ausrichtung an diesem Ort auch dann zuverlässig wieder, wenn er sich zwischendurch an einem anderen Ort oder sogar in einem anderen Teil des Gebäudes befunden hat
Der Roboter richtet sich an verschiedenen Orten unterschiedlich aus, in einem Extremfall sogar genau entgegengesetzt zur erwarteten Richtung. (im Flur vor dem kleinen Pool)

Der Kompass ist durch sich nähernde oder sich entfernende metallische Gegenstände beeinflussbar. Eine starke Beeinflussung ergibt jedoch nur bei magnetischen Gegenständen

Es muss angenommen werden, dass eine Räumlichkeit für ein Spiel oder Turnier ein stark variierendes Magnetfeld enthält.

Der Test wurde unter anderem in einer Räumlichkeit vorgenommen, die als sehr ungünstig anzusehen sein dürfte. Der Test zeigte, wie unterschiedlich das Magnetfeld an unterschiedlichen Positionen eines Raumes sein kann. Folgende Skizze gibt ein grobes Ergebnis wieder. Sie zeigt, dass es sich als überflüssig erwies, ein genaueres Ergebnis zu ermitteln. Die Skizze gibt grob einen Teil des Grundrisses des Raums im Treppenhaus des Lehrstuhlgebäudes Otto-Hahn-Straße 16 wieder. Der Bogen ist das metallene Treppengeländer. Das schwarze Rechteck stellt einen Kabelschacht dar, der senkrecht an der Wand entlang geht. Die blauen Pfeile zeigen in etwa die Ausrichtung, die der Roboter an diesen Positionen eingenommen hat. Er hat diese Ausrichtung jedes Mal an diesen Positionen eingenommen, auch wenn er zwischendurch in einem anderen Raum gewesen ist.

Die Unterschiede im Magnetfeld in diesem Raum wurden parallel mit einem Handkompass gemessen. Die anschließende Messung des Magnetfeldes mit dem Handkompass zeigte dabei die gleichen Abweichungen.

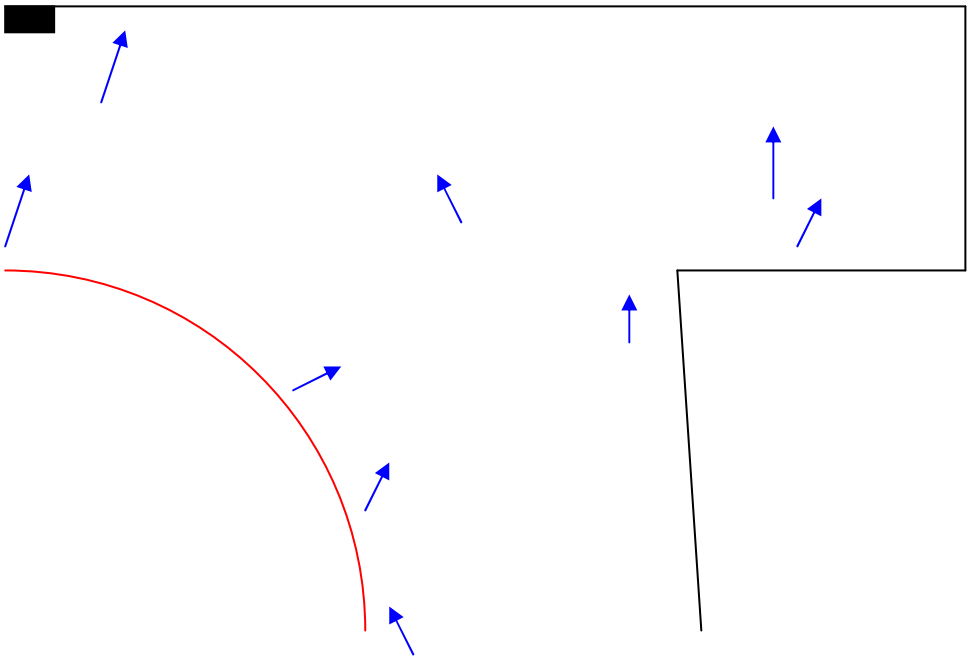


Abbildung V.7-1: Magnetfeldmessungen mit dem Roboter im Treppenhaus des Lehrstuhlgebäudes

Test 2:

Der Roboter wurde an der Stelle, an der der Kompass kalibriert wurde, von Hand in die vier Himmelsrichtungen ausgerichtet. Die Kompassdaten wurden per Debug-Ausgabe ermittelt. Das Ergebnis waren die Werte 0, 40, 180 und 220 Grad für die 4 Himmelsrichtungen. Herauskommen sollten optimalerweise Werte um die 0, 90, 180 und 270 Grad. Die gegensätzlichen Richtungen stimmen zwar, eine Drehung um 180 Grad würde im Beispiel dieser vier Ausrichtungen auch eine Kehrtwendung des Roboters ergeben, aber das Magnetfeld scheint an einer Stelle gestaucht zu sein, so dass z.B. Drehungen um 90 Grad nicht zum gewünschten Ergebnis führen würden.

Test 3:
Der Roboter soll auf einer kurzen Strecke (ca. 25 cm) hin- und herfahren. Er soll dabei ein Stück geradeaus fahren (die Geradeausfahrt wird dabei durch einen einfachen Aufruf der Methode doWheelSpeed realisiert, sie verwendet keine Daten des Kompass), anschließend wird die Ausrichtung des Roboters über den Kompass gemessen, daraus wird die Gegenrichtung berechnet und es wird versucht den Roboter über eine Rechtsdrehung in einem begrenzten Zeitfenster in die Gegenrichtung auszurichten, also eine Kehrtwendung durchzuführen. Danach beginnt die Routine von vorn.
Das Ergebnis ist dass der Roboter sich seitlich von seiner Ausgangsposition wegbewegt. Für den Test sind Toleranzen einkalkuliert, die zu diesen Ungenauigkeit führen können. Es zeigt sich jedoch dass dieser sichtbare Versatz der Position nur in einer der beiden Fahrtrichtungen stattfindet (wobei für beide Fahrtrichtungen der gleiche Code ausgeführt wird). Hier ist eine Ungenauigkeit mit bloßem Auge erkennbar, in der anderen Richtung nicht. In Einzelfällen weicht die Ausrichtung so stark ab, dass der Roboter eine komplett andere Fahrtrichtung einnimmt. Dies passiert allerdings nur sporadisch. Das Verhalten ist deutlich sichtbar aber nicht in jedem Fall nachvollziehbar.

V.7.2 Schlussfolgerungen

Drehungen um eine bestimmte Gradzahl oder gar Kurvenfahrten können mit dem Kompass-Sensor allein nicht exakt durchgeführt werden. Es könnte helfen, wenn der Roboter einen Bezug zwischen den Kompass-Daten und weiteren Informationen hat, zum Beispiel Positionen und Ausrichtungen an diesen Positionen. Denn der Sensor zeigt die Eigenschaft, dass er eine Ausrichtung an einem bestimmten Ort zuverlässig wieder findet, wenn er ein weiteres Mal an einer Position ankommt. Diese Eigenschaft könnte sich das System zu Nutze machen.

Die Störungen durch statische Magnetfelder ergaben sich bei ersten Tests als weitaus problematischer, als die Störfelder, die durch andere Roboter erzeugt werden. Daher wäre es denkbar, vor einem Spiel die statischen Abweichungen für das gesamte Spielfeld auszumessen, und diese Werte zur Korrektur der Sensordaten zu nutzen. Hierfür muss man allerdings wissen, wo der Roboter sich auf dem

Spielfeld befindet, um den richtigen Korrekturwert auswählen zu können. Daher ist dieser Ansatz nur sinnvoll, wenn der Roboter vom Hostsystem die aktuelle Position oder den entsprechenden Korrekturwert übermittelt bekommt.
Eine andere Möglichkeit, die Ausrichtung des Roboters auf dem Spielfeld zu bestimmen, würde ein Kreiselgyroskop bieten. Dies hätte den Vorteil, dass sich dieser Sensor nicht von Magnetischen Störfeldern beeinflussen lässt. Allerdings hat Thomas Klute bereits in seiner Diplomarbeit beschrieben, dass die Qualität der Daten beim Kreiselgyroskop von dessen Masse abhängt. Gyroskope mit einer geringen Masse haben eine starke Drift, außerdem summieren sich die Fehler mit zunehmender Laufzeit derart auf, dass die Ausrichtung letztlich gar nicht mehr stimmt. Daher ist es fraglich, ob sich ein geeignetes Kreiselgyroskop für die Verwendung mit autonomen Robotern finden lässt. Möglicherweise könnte ein kleines Kreiselgyroskop ausreichen, um über die Dauer eines Spieles die gegnerische und die eigenen Spielhälfte zu unterscheiden. Das Kreiselgyroskop könnte auch während des Spiels über Funk die nötigen Daten vom Hostsystem erhalten, um sich erneut exakt auszurichten.

V.8 Hindernisse bei der Umsetzung

Wie bereits erwähnt, gab es bei der Umsetzung der Ziele außergewöhnliche Hindernisse.
Das Hauptproblem bestand darin, dass über die gesamte Zeit der Projektgruppe keine Roboter-Hardware in der ursprünglich geplanten Form zur Verfügung stand. Das machte das ausführliche Testen praktisch unmöglich. Der Code und die Entwicklungsumgebung musste ständig an Umbauten und Veränderungen der Umgebung angepasst werden, was immer wieder sehr viel Zeit gekostet hat.

V.8.1 Roboter-Prototypen

Die ersten Tests des Kompass-Codes wurden auf einem V1-, später auf einem V2-Prototyp gefahren. Da die endgültigen Roboter kurz danach zur Verfügung stehen sollten, wurde der Prototyp nie fertig eingestellt, so dass er nur sehr unsauber fuhr. Da mit den Prototypen auch anderen Tests von anderen Gruppen durchgeführt wurden, war die Hardware oft nicht verfügbar und wurde häufig für die anderen Anwendungen umgebaut.
Daher konnte die Gruppe zwar zeigen, dass die entwickelte Kompass- und die I²C-Bibliothek funktionierten, aber präzise Messungen, die zeigen sollten wie genau man mit dem Kompass arbeiten kann, waren so nicht möglich. Als zusätzlich noch an mehreren Roboter-Boards Defekte auftraten, konnte die Testphase zunächst nicht beendet werden.

V.8.2 Das Speicherproblem

Als die Roboter zum Ende des Projektjahres gefertigt waren und nach der WM in Singapur dann auch der Gruppe mit Sensoren zur Verfügung standen, fehlten die SRAM Speicherbausteine. Für den Normalbetrieb ist das zwar unproblematisch, da der Code in diesem Fall im größeren Flash-Speicher gespeichert wird. Für das Testen und debuggen des Codes muss der Code allerdings in das RAM geladen werden. Durch das Fehlen des SRAM konnte das Robobrain-Programm nicht ohne weiteres in den Speicher geladen werden, um es zu testen und nach Fehlern zu durchsuchen. Debuggen des Codes im Flashspeicher ist nur mit sehr eingeschränkten Mitteln möglich, z.B. LED-Leuchten oder Räder drehen oder ähnliches. Den Debugger der Entwicklungsumgebung zu nutzen ist nicht möglich. Außerdem kann der Flash-Speicher der Roboter nur begrenzt oft neu beschrieben werden, so dass das Testen über Flashen die Lebensdauer der gerade neu gebauten Roboter dadurch verkürzen würde.

V.8.2.1 Verkleinerung des Codes

Eine Lösung dieses Problems bestand darin, das Programm so weit zu verkleinern, dass es in die internen Speicherbereiche des DSP geladen werden konnte. Diese Bereiche beschränken sich jedoch auf wenige kByte.

Leider liefert die Größe der ausführbaren Datei im Dateisystem des Entwicklungsrechners keinen Aufschluss über die Größe des Robotercodes. Stattdessen muss man hierfür die vom Compiler generierte Map-Datei betrachten. Sinnvollerweise wird diese Datei auch dann geschrieben, wenn der Compiler die ausführbare Datei aufgrund zu großer Programmblöcke nicht erzeugt.

Es stellte sich heraus, dass zum einen die printf-Ausgaben und zum anderen das relativ großzügig angelegte Array für die Logdaten viel Speicher benötigen. Verkleinert man das Array auf 3 Einträge und entfernt alle printf-Kommandos, so dass dieser Teil der Ein-/Ausgabe-Bibliothek nicht mit in das Programm gebunden werden muss, gelangt man in Bereiche von Codegrößen, die in den internen Speicher passen. Hierbei ist es zum einen wichtig, dass *alle* printf-Kommandos direkt oder indirekt über #define und #ifdef aus dem Code entfernt werden, selbst wenn sie in Methoden stehen, die nicht aufgerufen werden. Zum anderen werden in Zukunft vermutlich zusätzlich große Mengen Bilddaten verwaltet, die eventuell auch angepasst werden müssen.

Will oder muss man noch mehr Platz einsparen, kann nicht benötigter Code auskommentiert werden. Aufgrund der unterschiedlichen Hardwareversionen, für die der Code geschrieben ist, ist es allerdings sehr aufwändig, die Codegröße hierdurch noch weiter zu minimieren. Im Wesentlichen wurden bei der Bereinigung des Codes alle nicht benötigten Teile entfernt beziehungsweise über #define und #ifdef den entsprechenden Hardwareversionen angepasst.

Als weitere Maßnahme kann man die Optimierung des Codes durch den Compiler in Betracht ziehen. Wenn das .text-Segmentent, das den Programmcode enthält, in keinen der Speicherbereiche passt, hilft es eventuell, für den Compiler ein Optimierungs-Level von eins oder zwei anzugeben, und dadurch

den Code um ein paar hundert Byte zu verkleinern. Das genügte in unserem Fall bereits. Damit lässt sich der Code auch mit sämtlichen Modulen verwenden. Diese Maßnahme hat allerdings Einschränkungen beim Debuggen zur Folge, da der Debugger den optimierten Programmcode nicht in gewohnter Art und Weise durchläuft: Durch die Optimierung werden zum Teil Zeilen umgestellt oder ausgelassen. Das Wegoptimieren von Codezeilen durch den Compiler ist besonders dann ein Problem, wenn Testvariablen eingefügt werden, um ihren Inhalt beim Debuggen auszulesen. Werden diese Variablen nirgendwo sonst sinnvoll benutzt, entfernt der Compiler sie komplett, so dass das Setzen von Breakpoints in diesen Zeilen ohne Effekt bleibt. Vorsicht geboten ist mit inline-Funktionen, die zum Teil nicht mehr aufgerufen werden. Hier kann es sich allerdings nur um einen Compiler-Fehler handeln.

Um den Code zu debuggen gibt es verschiedene Möglichkeiten. Für den Fall dass eine kleine Codegröße benutzt werden muss (mangels RAM), scheiden printf-Ausgaben aus der Liste der Möglichkeiten aus. Die Library und die Datenstrukturen, die für printf-Kommandos in den Code eingebunden werden müssen, sind relativ groß. Sie kommen auch nicht in Frage, wenn es auf das Timing und die Motorsteuerung ankommt. Die Kommandos bereiten in diesem Fall zu große Performance-Probleme. Die unterschiedlichen Möglichkeiten bringen unterschiedliche Probleme mit sich, die in der folgenden Auflistung beschrieben sind:

- Die Nutzung von Breakpoints und der Step-Funktion des Debuggers in der Code-Composer-Umgebung ist sehr rechenintensiv, da sie einen erheblichen Datentransfer zwischen DSP und Entwicklungsumgebung erzeugt. Daher kann diese Methode zu Problemen mit den Realtime-Interrupts führen. Grundsätzlich kann und sollte sie aber verwendet werden.
- Die Verwendung von printf-Kommandos ist ebenfalls sehr rechenintensiv und führt dazu, dass Realtime-Interrupts übergangen werden. Dieses Verhalten kann für printf-Kommandos mit aufwendigen Formatierungen eindrucksvoll am Heartbeat der Leuchtdioden beobachtet werden. Falls keine Motoransteuerung für einen Test benötigt wird sind printf-Ausgaben jedoch eine sinnvolle Methode.
- Um zu untersuchen ob Taktsignale, Signalflanken oder kurze, sich wiederholende Datenfolgen an den Anschlüssen des Prozessors anliegen hilft ein Oszilloskop, das die Signale sichtbar machen kann. Voraussetzung ist hier, dass an die zu untersuchenden Anschlüsse Abgriffe angeschlossen oder angelötet werden können, was aufgrund der Baugrößen der Bausteine nicht immer möglich ist.
- Die Leuchtdioden des Roboters können ebenfalls eine große Hilfe darstellen. Im aktuellen Robotercode zeigt der Heartbeat an, dass der Roboter betriebsbereit ist und auf Funkbefehle wartet, das rote Blinken beim Einschalten des Roboters zeigt, dass die Initialisierung beginnt. Die Leuchtdioden verursachen keine Speicher- oder Performanceprobleme.

Hier noch mal eine Übersicht über die notwendigen Schritte zur Verkleinerung des Codes:

- entfernen aller printf-Kommandos,
- Minimierung der Logdaten und anderer großer Datenstrukturen,
- auskommentieren von Code,
- Einschalten eines Optimierungs-Level (-o1 oder –o2).

Als nächstes muss eine MemoryMap gebaut werden, die die Code-Blöcke passend in die internen Speicherbereiche des DSP aufteilt. Die Details dazu erläutert der nächste Abschnitt.

V.8.2.2 Bau einer Memory-Map

Die kleinen, internen Speicherbereiche des Prozessors teilen sich in unterschiedliche Blöcke auf. Welche davon verfügbar sind, kann dem Speicherblockbild des DSP entnommen werden.

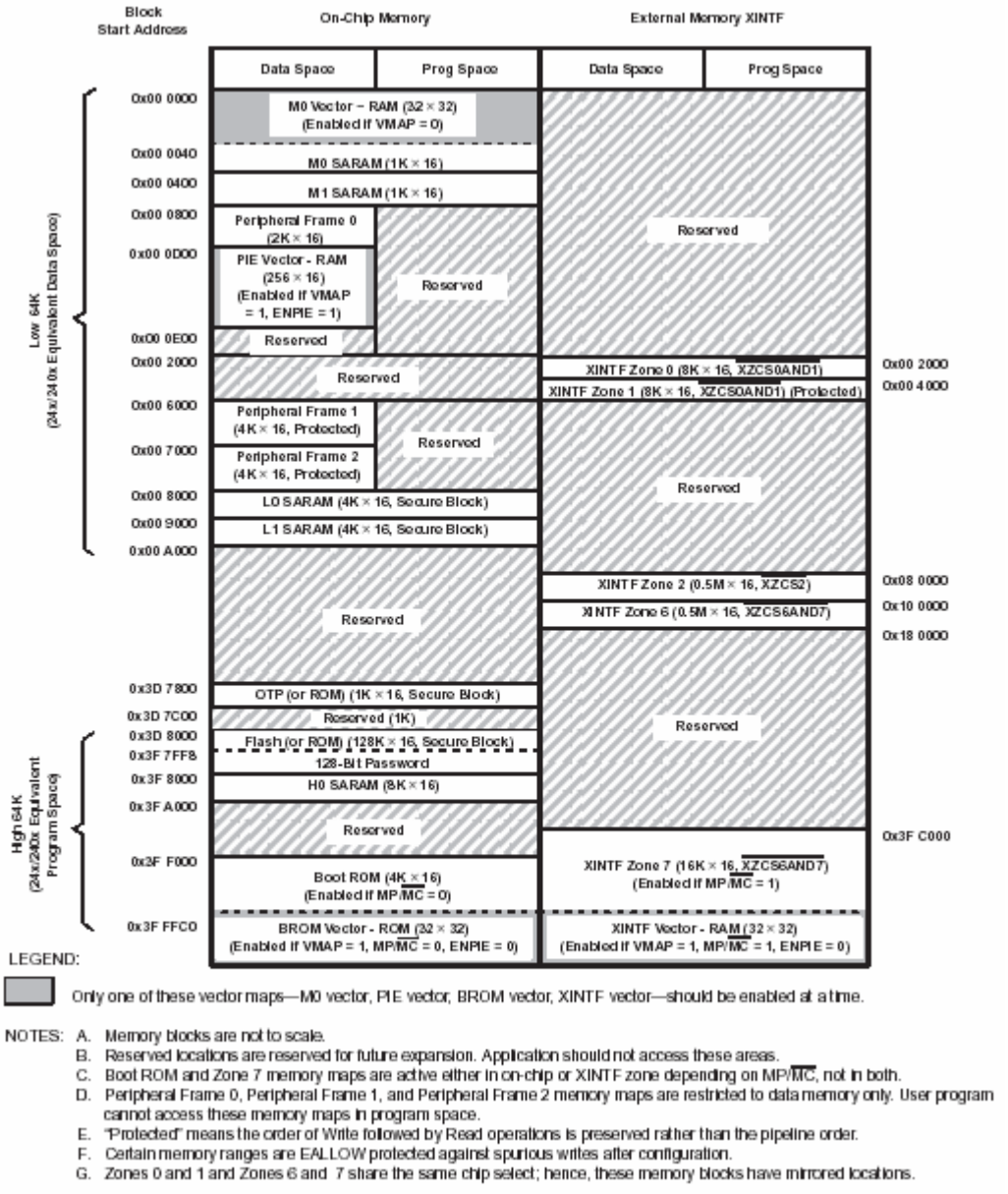


Abbildung V.8-1: Memory Map des TMS320F2812 (Quelle: TI, tms320f2812.pdf, Seite 29)

Die Bereiche, die in der ursprünglichen Map als freie und für Tests verwendbare Bereiche genutzt wurden, sind in der folgenden Tabelle aufgeführt. Das externe SRAM ist hier ebenfalls aufgeführt, diese großen Speicherbereiche standen für die Tests zunächst jedoch nicht zur Verfügung.

Bereich	Größe (Hex)	Bemerkung
M0RAM	0x3C0	Intern
M1RAM	0x400	Intern
L0L1RAM	0x2000	Intern
PROG	0x1F80	Intern
ZONE2	0x80000	externes SRAM (DROID3)
ZONE6	0x80000	externes SRAM (eZ-Board)

Tabelle V.8-1: verfügbare Speicherbereiche auf dem DSP und den Roboterboards

Nach Verkleinerung und Optimierung des kompletten Codes sind die in folgender Tabelle aufgeführten Code-Segmente von der Größe her das Hauptproblem. Die Werte sind ungefähre Werte für den zu diesem Zeitpunkt verwendeten Code. Je nach Optimierung oder Umfang der Code-Reduzierung fallen die Werte unterschiedlich aus.

Code-Segment	Größe (Hex)
.text (Code)	ca. 0x1D80 bis 0x1E80
.ebss (Daten)	ca. 0x6F0 bis 0x7A0
.stack (Stapel)	ca. 0x190
.econst (Konstanten)	ca. 0x17A

Tabelle V.8-2: Größen einiger Teile des compilierten Codes

Die ursprüngliche Map war wie folgt festgelegt (es sind nur die für die Anpassung relevanten Teile der Datei aufgeführt):

```
MEMORY {
/* interne Bereiche */
PAGE 0 : RESET(R)    : origin = 0x000000, length = 0x2
PAGE 1 : M0RAM(RW)   : origin = 0x000040, length = 0x3C0  /* 1K - 40 */
PAGE 1 : M1RAM(RW)   : origin = 0x000400, length = 0x400  /* 1K */
PAGE 1 : L0L1RAM(RW) : origin = 0x008000, length = 0x2000 /* 2x 4K */

PAGE 0 : BOOT(R)     : origin = 0x3f8000, length = 0x80
PAGE 0 : PROG(R)     : origin = 0x3f8080, length = 0x1f80
PAGE 0 : BOOTROM(RW) : origin = 0x3ff000, length = 0x000fc0
```

```
/* externes SRAM */
PAGE 1 : ZONE2       : origin = 0x080000, length = 0x80000
PAGE 1 : ZONE6       : origin = 0x100000, length = 0x80000
PAGE 1 : FLASHB      : origin = 0x3F4000, length = 0x002000
}

SECTIONS{
  .reset   : > RESET, PAGE = 0, TYPE = DSECT

  .codesection_internal_H0 : > PROG, PAGE = 0
  .codesection_external_zone6 : > ZONE6, PAGE = 1

  .pinit   : > PROG,  PAGE = 0
  .cinit   : > PROG,  PAGE = 0
  .text    : > ZONE6,  PAGE = 1


/* 16-Bit data sections */
  .const   : > M0RAM, PAGE = 1
  .stack   : > M1RAM, PAGE = 1
  .sysmem  : > L0L1RAM, PAGE = 1

/* Var */
  .bss     : > M1RAM, PAGE = 1
  .ebss    : > ZONE6, PAGE = 1

  .econst  : > L0L1RAM, PAGE = 1
  .esysmem : > L0L1RAM, PAGE = 1
  .cio     : > L0L1RAM, PAGE = 1
...
  IQmath   : load = PROG, PAGE = 0
...
```

Die Speicherbereiche M0RAM und M1RAM liegen direkt hintereinander, daher können sie zu einem größeren Bereich zusammengelegt werden. Eine Memory-Map könnte wie folgt aussehen:

```
MEMORY {
/* interne Bereiche */
PAGE 0 : RESET(R)    : origin = 0x000000, length = 0x2
PAGE 1 : M0M1RAM(RW) : origin = 0x000040, length = 0x7C0  /* 2K - 40 */
PAGE 1 : L0L1RAM(RW) : origin = 0x008000, length = 0x2000 /* 2x 4K */
```

```
PAGE 0 : BOOT(R)     : origin = 0x3f8000, length = 0x80
PAGE 0 : PROG(R)     : origin = 0x3f8080, length = 0x1f80
PAGE 0 : BOOTROM(RW) : origin = 0x3ff000, length = 0x000fc0

PAGE 1 : FLASHB      : origin = 0x3F4000, length = 0x002000
}

SECTIONS{
  .reset   : > RESET, PAGE = 0, TYPE = DSECT

  .codesection_internal_H0 : > PROG, PAGE = 0

  .pinit   : > L0L1RAM, PAGE = 1
  .cinit   : > L0L1RAM, PAGE = 1
  .text    : > PROG, PAGE = 0


/* 16-Bit data sections */
  .const   : > L0L1RAM, PAGE = 1
  .stack   : > L0L1RAM, PAGE = 1
  .sysmem  : > L0L1RAM, PAGE = 1

/* Var */
  .bss     : > L0L1RAM, PAGE = 1
  .ebss    : > M0M1RAM, PAGE = 1

  .econst  : > L0L1RAM, PAGE = 1
  .esysmem : > L0L1RAM, PAGE = 1
  .cio     : > L0L1RAM, PAGE = 1
...
  IQmath   : load = L0L1RAM, PAGE = 1
...
```

Auf diese Weise kann der Code mit allen Modulen in einer optimierten Version in die internen Speicherbereiche des DSP geladen und dort getestet werden.

VI. Gruppe Lokale Kamera

Tom Bomhof,
Christoph Ewerlin,
Andreas Nettsträter

VI.1 Aufgaben

Die Aufgabenstellung der Gruppe Lokale Kamera ist es, ein Kamera-Framework zu schaffen, welches einen einfachen Zugriff auf die Funktionen und Bilder der Kamera ermöglicht. Dabei soll einerseits ein modularer Aufbau das Modul unabhängig machen, sich jedoch andererseits in den vorhandenen Quellcode des Roboters einfügen. Die Aufgabe wurde von uns in die folgenden Unteraufgaben aufgeteilt:

- Initialisierung der Kamera
- Funktionstest der Kamera und des Framebuffers
- Tests der verschiedenen Aufnahmemodi
- Implementierung eines Frameworks zur Datenverarbeitung
- Erkennung des Balles

Um zu zeigen, dass die Aufgaben von unserer Gruppe erreicht wurden, haben wir folgende Ziele definiert:

- Der Roboter versucht automatisch den Ball zu finden, indem er sich solange um die eigene Achse dreht, bis ein Ball in seinem Sichtbereich auftaucht. Nun verfolgt der Roboter durch Drehung auf der Stelle den Ball, falls dieser droht, sich wieder aus dem Sichtbereich zu entfernen.
- Der Roboter erfüllt das oben genannte Ziel und wird zusätzlich hinter dem Ball herfahren, falls sich dieser von dem Roboter entfernt.

Für die hier definierten Ziele wird vorausgesetzt, dass der Roboter auf dem Spielfeld steht und für ausreichende Lichtverhältnisse gesorgt ist.

VI.2 Voraussetzungen

Im Rahmen der Arbeit dieser Teilgruppe wurde die von der Vorgänger-Projektgruppe erstellte Prototyp – Platine verwendet. Während der Arbeit ergaben sich noch einige kleinere Änderungen.
Für uns waren hauptsächlich die folgenden Bauteile interessant:

- Kamera: Agilent ADCM 1700
- DSP: Texas Instruments TMS320F2812

- Framebuffer: Averlogic AL422B 3Mbit FIFO-Memory

Die Kamera ist eine typische Handy-CMOS Kamera. Nähere Informationen hierzu finden sich im Abschnitt VI.3.1.1.
Der Framebuffer ist ein sequentieller Speicher. Über einen Port kann sequentiell geschrieben werden, über den anderen Port kann sequentiell ausgelesen werden. Dieses Bauteil dient hauptsächlich als Puffer zwischen Kamera und DSP.

Die Kamera wird per I²C-Bus vom DSP eingestellt. Die Bilddaten werden von ihr direkt an den Framebuffer geschickt. VSync und Kameratakt dienen hierbei zur Synchronisation.
HSync und VSync der Kamera werden ebenfalls an die externen Interrupts des DSP gesendet, der so auf das Ende einer Zeile bzw. das Ende eines Bildes reagieren und nach Bedarf die Bilddaten aus dem Framebuffer auslesen kann.

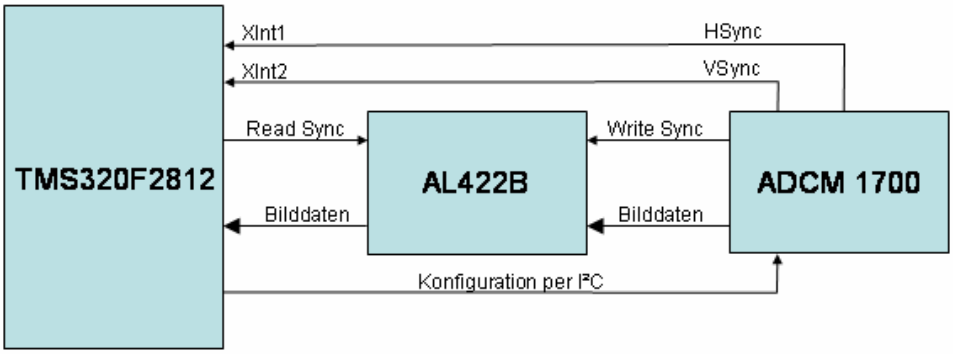


Abbildung VI.2-1: Verarbeitung von Bilddaten aus Hardwaresicht

VI.3 Implementierung auf dem DSP

Wir haben wie in den folgenden Kapiteln beschrieben den entsprechenden Code auf dem DSP implementiert. Dabei sind folgende Dateien entstanden. Für die Kamera (VI.3.1): *adcm1700.c*, *adcm1700.h*, *cam_config.h*, für den Framebuffer (VI.3.2.2): *framebuffer.c*, *framebuffer.h*, fürs Kamera-Framework (VI.3.2): *cam_framework.c*, *cam_framework.h*, *filter_transmit.c*, für die Ballfinder (VI.4): *check_ball.c*, *filter_findball.c*, *filter_findball_grid.c*, für die Ansteuerung (VI.3.5.2): *steuerung.c*, *steuerung.h* und für die Strategie (VI.3.5): *strategy.c*, *strategy.h*.
Den Code haben wir ins CVS im Repository *pg472_kameragruppe* eingecheckt.

VI.3.1 Einstellen der Kamera

VI.3.1.1 Kameratyp

Die im Roboter verbaute Kamera ist eine Agilent ADCM-1700. Das in der Kamera verwendete Objektiv ist ein Fixfokus-Objektiv. Als Folge davon sieht die Kamera erst ab einer Entfernung von 10cm scharf. Die maximale Auflösung der Kamera beträgt 352x288 Pixel, die maximale Framerate liegt bei 15 FPS. Die Kamera unterstützt eine Vielzahl von Ausgabeformaten, unter anderen YUV- und RGB-Formate (mehr dazu im Kapitel VI.3.1.3). Die ausgegebene Auflösung lässt sich sehr flexibel anpassen; so ist es zum einen möglich, eine beliebige Auflösung (natürlich innerhalb der maximal möglichen) einzustellen, zum anderen kann ausgewählt werden, an welcher Stelle des Sensors die gewünschte Auslösung abgegriffen wird. Alternativ ist es auch möglich, das Bild des gesamten Sensors zu benutzen und von der Kamera in die gewünschte Auflösung umrechnen zu lassen (mehr dazu im Kapitel VI.3.1.4). Das Blickfeld der Kamera beträgt in horizontaler Richtung 55°. Die Kamera kann einige Bildverbesserungverfahren direkt anwenden. So gibt es einen automatischen Weißabgleich und eine Rauschreduzierung. Zusätzlich ist es möglich statistische Daten wie ein Histogramm zu den aufgenommen Bilder abzufragen.

Die Kamera ist sowohl über eine Kontrollverbindung als auch eine Datenverbindung an den DSP angeschlossen. Die Kontrollverbindung wird seriell über einen I^2C-Bus realisiert während die Datenverbindung über eine Parallelverbindung läuft. Die Kontrollverbindung dient zur Konfiguration der Kamera, die Datenverbindung transportiert die Bilddaten.

Die Kamera bietet zusätzlich noch zwei Signale an, vsync und hsync. Das vsync-Signal wird nach jedem komplett ausgegebenen Frame ausgelöst, hsync nach jeder kompletten Zeile.

VI.3.1.2 Simple Control Registers / Advanced Control Registers

Die Kamera wird über so genannte Control Registers konfiguriert. Dabei wird zwischen zwei Typen von Registern unterschieden.

Die Simple Control Register dienen dazu, häufig benötigte Einstellungen bequem vornehmen zu können. Nach dem Beschreiben der Simple Control Register mit den gewünschten Werten setzt man die Kamera in einen Konfigurationsmodus. Diese liest nun den Inhalt der Simple Control Register, berechnet aus den dort gefundenen Werten entsprechende Werte für eines oder mehrere Advanced Control Register und speichert dort diese errechneten Werte ab.

Die eigentliche Konfiguration der Kamera liegt in den Advanced Control Registern. Diese können vom Benutzer auch direkt beschrieben werden, wenn Einstellungen vorgenommen werden sollen, für die keine Simple Control Register vorgesehen sind. Durch das direkte Beschreiben der Advanced Control Register hat der Benutzer die volle Kontrolle über alle Einstellungen der Kamera.

VI.3.1.3 Farbmodi

Die Kamera unterstützt verschiedene Farbmodi. Außer YUV- (bzw. YCbCr) und RGB-Modi werden auch RAW-Modi unterstützt, in denen die Sensordaten ohne Konvertierungen an den DSP weitergereicht werden. Die verschiedenen YUV- bzw. RGB-Modi unterscheiden sich durch eine unterschiedliche Anordnung der Daten sowie durch unterschiedliche Genauigkeiten der Farbdarstellung.
Wir haben uns im Wesentlichen auf ein Farbformat konzentriert, das CbYCrY 422B Format. Das Format basiert auf dem Helligkeitswert (Y), Rot- (Cr) und Blauanteil (Cb) der Farbe. Dabei werden die Cb- und Cr-Werte zweier Pixel zusammengefasst und nur als jeweils ein Durchschnittswert abgespeichert. Die Y-Werte werden für jeden Pixel einzeln gespeichert. Jeder Pixel kann so in 2 Bytes gespeichert werden. Eine genaue Anordnung der Daten ist in **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen.

Byte #	1	2	3	4	5	6	7	8	9	…
Daten	Cb_{12}	Y_1	Cr_{12}	Y_2	Cb_{34}	Y_3	Cr_{34}	Y_4	Cb_{56}	…

Tabelle VI.3-1: CbYCrY 422B Format

Dabei ist z.B. Cb12 der Durchschnittswert der Cb-Werte des ersten und zweiten Pixels.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.296875 & 0.585938 & 0.117188 \\ -0.171875 & -0.328125 & 0.500000 \\ 0.500000 & -0.421875 & -0.078125 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

Abbildung VI.3-1: Umrechnung von RGB-Werten in YCbCr-Werte

Das RGB332-Format reduziert die Bilddaten auf 1 Byte pro Pixel. In diesem Byte werden die Rot- und Grünwerte durch jeweils 3 Bit codiert, der Blauwert durch 2 Bit. Dabei werden nur die entsprechenden oberen Bits des ursprünglichen Werts gespeichert. Die genaue Anordnung ist in Tabelle VI.3-2 zu sehen.

Bit #	7	6	5	4	3	2	1	0
Daten	R_7	R_6	R_5	G_7	G_6	G_5	B_7	B_6

Tabelle VI.3-2: RGB332 Format

Dabei ist z.B. R_7 das siebte Bit des ursprünglichen Rotwerts.

Abbildung VI.3-2: CbYCrY422B (links), RGB332 (rechts)

VI.3.1.4 Sizer

Der Sizer kann dazu benutzt werden, die Größe des Ausgabebildes zu verändern. Als Eingabe bekommt der Sizer das vom Sensor aufgenommene Bild und liefert als Ausgabe ein Bild der gewünschten Größe. Um ein Bild einer gewünschten Größe zu erzeugen wäre es natürlich auch möglich, nur einen Teil des Sensorbildes zu benutzen. Im Gegensatz zu diesem Verfahren verringert sich bei Benutzung des Sizers der Blickwinkel der Kamera aber nicht.

VI.3.1.5 Initialisierung

Um die Kamera für unsere Zwecke zu initialisieren, haben wir Datenformat, FPS und Ausgabegröße des Bildes konfiguriert. Zusätzlich wurden noch technische Einstellungen vorgenommen. So beträgt die Frequenz mit der die Kamera betrieben wird 25 MHz und nicht 13 MHz, wie von der Kamera als Standardwert angenommen. Zusätzlich muss das vertikale Synchronisationssignal invertiert werden, um vom Framebuffer erkannt zu werden.

Da über Simple Control Register nur einige vordefinierte Ausgabegrößen eingestellt werden können, die unseren Zwecken aber nicht genügen (zur Ballerkennung brauchen wir ein breites aber nicht sehr hohes Bild), muss die Ausgabegröße über die Advanced Control Register eingestellt werden. Da dann zusätzlich zur gewünschten Größe auch noch interne Timing-Parameter der Kamera korrekt eingestellt werden müssen (was in der Dokumentation der Kamera aber nicht beschrieben wird), haben wir es leider nicht geschafft, die Initialisierung stabil zu implementieren. Oft hat die Kamera erst nach mehrmaligem Versuch die Initialisierung durchzuführen die Parameter akzeptiert und ein korrektes Bild geliefert. Da dies Verhalten für den Betrieb nicht akzeptabel ist, benutzen wir einen „Hack“, um eine zuverlässige Initialisierung durchführen zu können. Nachdem wir nach mehreren Versuchen die Kamera korrekt initialisiert hatten, haben wir uns den Inhalt aller relevanten Register ausgeben lassen und abgespeichert. Dieser gespeicherte Inhalt wird nun zur Initialisierung direkt in die entsprechenden Register geschrieben. Der Vorteil dieses Verfahrens liegt darin, dass die Initialisierung zuverlässig funktioniert. Ein großer Nachteil ist, dass eine einfache Änderung einzelner Parameter (z.B. andere

FPS) nicht möglich ist. Dazu müsste zuerst der entsprechende Registerinhalt für die gewünschte Parameterkombination bestimmt und abgespeichert werden.

VI.3.1.6 Timing-Probleme

Zusätzlich zum von uns letztendlich benutzen Datenformat CbYCrY422B haben wir andere Datenformat wie RGB888, RGB332 und YCbCr444 getestet. Leider kommt es bei allen Formaten außer CbYCrY422B zu Timing-Problemen zwischen Kamera und Framebuffer. Zum Teil werden die Bilder nur verschoben im Framebuffer abgelegt, zum Teil enthält der Framebuffer aber auch nur Rauschen. Woran diese Timing-Probleme genau liegen und ob sie behoben werden können, haben wir leider auch unter Mithilfe der Hiwis nicht feststellen können.

VI.3.2 Kamera - Framework

Das Kameraframework bildet die Schnittstelle zwischen Kameradaten und Strategie. Auf der einen Seite initialisiert es die Kamera und stellt deren Daten periodisch zur Weiterverarbeitung im Speicher bereit. Dabei wird immer dafür gesorgt, dass nur vollständige Bilder weiter verarbeitet werden und diese auch immer so aktuell wie möglich sind. Details zur Datensammlung werden im Abschnitt VI.3.2.2 beschrieben.

Auf der anderen Seite stellt es Filter zur Verfügung, die auf Basis der Bilddaten Informationen berechnen, die dann an die Strategie weiter geleitet werden. Hier wird gewährleistet, dass die Filter stets mit aktuellen Informationen arbeiten, ebenso unterstützt werden „langsame“ Filter, die für lange Zeit Bilddaten aus demselben Bild benötigen und „schnelle“ Filter, die Informationen sehr aktuell verarbeiten müssen. Details zum Filterkonzept werden im Abschnitt VI.3.2.3 beschrieben.

Im Abschnitt VI.3.2.1 wird eine generelle Übersicht über die Zusammenhänge im Kameraframework gegeben. In Abschnitt VI.3.2.3 wird dann noch einmal anhand eines beispielhaften Ablaufs verdeutlicht, wie das Kameraframework in der Praxis Bilddaten sammelt und Filtern Bilder zuweist.

VI.3.2.1 Generelle Übersicht

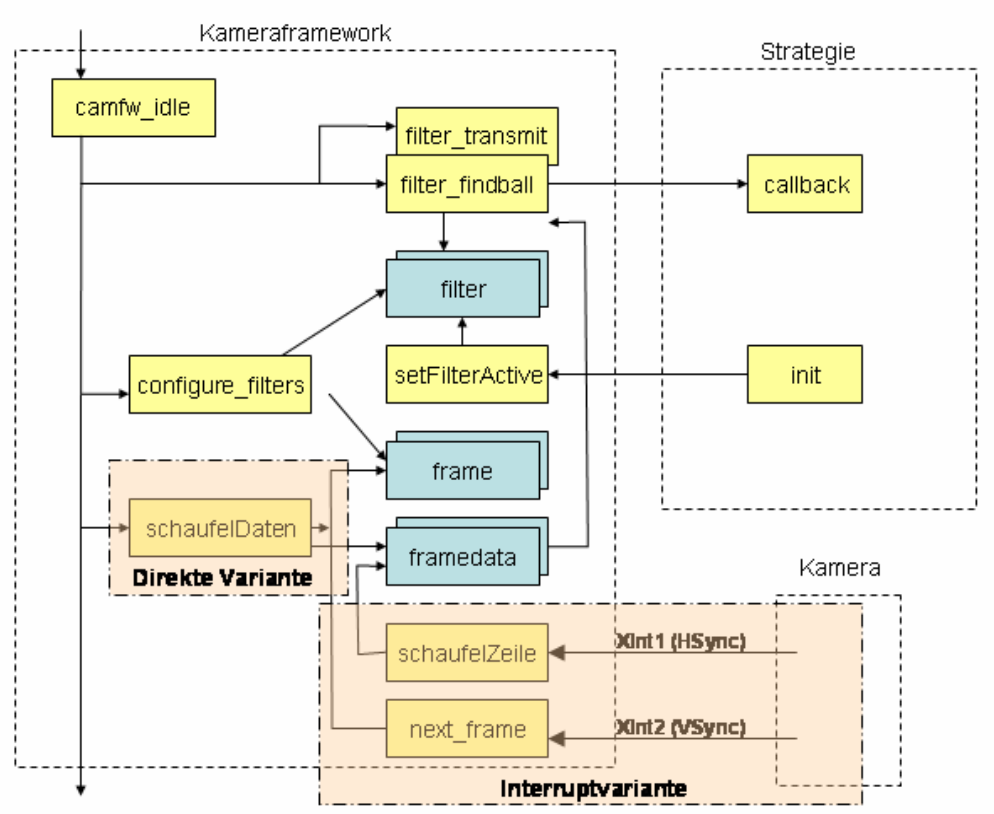


Abbildung VI.3-3: Abläufe im Kameraframework

VI.3.2.2 Datensammlung

Das Kameraframework ist für die Bereitstellung der Bilddaten zuständig. Wie bereits in Abschnitt VI.2 beschrieben, werden die Kameradaten zunächst in einem Framebuffer zwischengepuffert. Dieser ermöglicht jedoch nur sequenziellen Zugriff auf Bilddaten, was für viele Bildverarbeitungsaufgaben nicht ausreicht. Durch das Signal VSync der Kamera wird beim Schreiben jeweils wieder an den Anfang des Framebuffers gesprungen, so dass jeder Frame sofort wieder überschrieben wird. Das Kameraframework schafft hier Abhilfe.

In einem Puffer werden mehrere Frames gespeichert und mit Bilddaten aus dem Framebuffer gefüllt. Eine Frame kann von einem Filter gesichert werden (siehe Abschnitt VI.3.2.3) und wird dann solange der Filter daran arbeitet nicht mehr überschrieben. Durch das Ablegen der Bilddaten im externen Speicher ist beliebiger Zugriff möglich. Jeder neue Frame bekommt eine eindeutige, aufsteigende ID zugewiesen, um einen zeitlichen Zusammenhang herstellen zu können.
Die Anzahl der rotierten Frames lässt sich über die Konstante *FRAME_COUNT* in *cam_framework.h* einstellen.

Jeder Frame erhält eine eigene Struktur vom Typ *camfw_frame*, die eigentlichen Bilddaten werden im zusätzlichen globalen Array *framedata* gespeichert. Dieser Schritt ist sinnvoll, da Bilddaten nicht zwangsläufig im selben Speicherbereich wie die Kontrollstruktur liegen müssen.
Mit Bilddaten müssen auch nicht unbedingt Pixelwerte gemeint sein, es ist z.B. denkbar, dass von einer externen Quelle bereits vorsegmentierte Daten geliefert werden, die dann von den Filtern weiter verarbeitet werden. Die Konstante *PAGE_SIZE* in *cam_framework.h* gibt die Größe der Bilddaten eines Frames an, im Allgemeinen gilt für Pixeldaten *PAGE_SIZE = FRAME_SIZE*.

```
typedef struct
{
 long int id;
 char lock_count;
} camfw_frame;

extern camfw_frame frame[FRAME_COUNT];
extern uchar framedata[FRAME_COUNT][PAGE_SIZE];
```

Die Variable *id* gibt die laufende Nummer des in der Struktur gespeicherten Frames an. Die Variable *lock_count* zählt, wie viele Filter den Frame gesichert haben. Nur wenn *lock_count = 0* ist, wird ein Frame mit neuen Bilddaten beschrieben.

Die Auswahl, welcher Frame als nächster überschrieben wird, trifft die Funktion *next_frame* in *cam_framework.c*. Bei Fertigstellung eines Frames wird jeweils der älteste, nicht gesicherte Frame ausgewählt und die Schreibposition *writePointer* auf den Anfang von dessen Bilddaten gesetzt. Sollten alle Frames gesichert sein, wird ohne Rücksicht auf alle Filter der Frame 0 neu beschrieben. In so einem Fall ist allerdings so programmiert worden, dass die Resourcen des DSPs nicht ausreichen, um alle Aufgaben rechtzeitig zu erfüllen.

Für den Zugriff auf die Bilddaten im Framebuffer wurde eine kleine API implementiert, die in der Datei *framebuffer.h* zusammengefasst ist und von der Datensammlung benutzt wird.

```
unsigned char fb_read_byte();
void fb_reset();
void fb_init();
```

- *fb_read_byte*: Liest ein Byte aus dem Framebuffer an der aktuellen Position und inkrementiert die Leseposition des Framebuffers.
- *fb_reset*: Setzt die Leseposition des Framebuffers an den Anfang zurück.
- *fb_init*: Wird einmal in camfw_init aufgerufen und initialisiert den Framebuffer.

Die Daten können mit zwei unterschiedlichen Methoden gesammelt werden, einmal durch Interrupts gesteuert über die Signale *VSYNC* und *HSYNC* der Kamera, einmal durch unsynchronisiertes, direktes Lesen der Daten aus dem Framebuffer in der Funktion *camfw_idle*. Beide Methoden sind implementiert, im Code wird jedoch nur die direkte Variante benutzt, da bei der durch Interrupts gesteuerten Variante Probleme auftraten. Auf der von der Vorgänger-PG erstellten Platine waren die Signale VSYNC und HSYNC nicht mit den passenden Eingängen des DSPs verbunden. Nach Anlöten stellten wir fest, dass die Signalflanken nicht hoch genug waren, das Einlöten eines Pullup-Widerstandes brachte das Timing zwischen Kamera und Framebuffer durcheinander.

Nachteil der unsynchronisierten Methode ist, dass die tatsächliche Geschwindigkeit des Auslesens nicht mit der Schreibgeschwindigkeit der Kamera in den Framebuffer zusammen hängt. Angenommen die Kamera liefert 10fps, das Framework schafft es aber nur mit 7fps auszulesen. In diesem Fall „überholt“ die Kamera irgendwann das Framework beim Lesen aus dem Framebuffer und die Bilddaten kommen von zwei verschiedenen Bildern. Bei sehr schnellen Bewegungen könnten dadurch zum Beispiel zwei Bälle auf einem Bild auftauchen.

Abbildung VI.3-4: Beispiel für ein Synchronisationsproblem

Beim direkten Zugriff wird bei jedem Aufruf der Funktion *camfw_idle* einmal die Funktion *schaufel_daten* aufgerufen. Die Variable *write_pointer* ist global und zeigt auf die aktuelle Schreibposition im globalen Array *framedata*, in dem die Bilddaten gespeichert sind. Nach Lesen der letzten Zeile wird mit der oben beschriebenen Funktion *next_frame* ein neuer Frame ausgewählt.

```
void schaufel_daten()
{
 static int columns_read = 0;

 if(columns_read < FRAME_W)
 {
      for(i=0; i<COLUMN_SIZE; i++)
       *writePointer++ = fb_read_byte();
      columns_read++;
```

```
 }

 if(columns_read == FRAME_W)
 {
      next_frame();
      columns_read = 0;
 }
}
```

Beim Zugriff über Interrupts wird die Fähigkeit des TMS320F2812 externe Interruptquellen zu benutzen ausgenutzt. Die Signale *HSYNC* und *VSYNC* der Kamera wurden mit den Pins *XINT1* und *XINT2* des DSPs verbunden. In der Initialisierung des Kameraframeworks werden die externen Interrupts so konfiguriert, dass sie bei fallender Flanke (also wenn die Kamera eine neue Zeile / einen neuen Frame anfängt) ausgelöst werden.

```
EALLOW;

GpioMuxRegs.GPEMUX.bit.XINT1_XBIO_GPIOE0 = 1;
GpioMuxRegs.GPEMUX.bit.XINT2_ADCSOC_GPIOE1 = 1;

// Interruptroutinen setzen
PieVectTable.XINT1 = &camfw_hsync_isr;
PieVectTable.XINT2 = &camfw_vsync_isr;

// Interrupt1 einschalten, fallende Flanke
XIntruptRegs.XINT1CR.bit.ENABLE = 1;
XIntruptRegs.XINT1CR.bit.POLARITY = 0;

// Interrupt2 einschalten, fallende Flanke
XIntruptRegs.XINT2CR.bit.ENABLE = 1;
XIntruptRegs.XINT2CR.bit.POLARITY = 0;

// Interrupts aktivieren
PieCtrlRegs.PIEIER1.bit.INTx4 = 1;
PieCtrlRegs.PIEIER1.bit.INTx5 = 1;

EDIS;
```

Die Interruptroutinen sind vergleichsweise einfach. Die Funktion *camfw_hsync_isr* wird nach jeder fertig gestellten Zeile aufgerufen und kopiert ähnlich wie die Funktion *schaufel_daten* eine einzelne Zeile vom Framebuffer ins RAM, jedoch ohne sich die Zeilennummern zu merken. Die Funktion *camfw_vsync_isr* ruft nur die Funktion *next_frame* auf. Besonderer Vorteil dieser Methode ist, dass alle Frames synchronisiert gelesen werden und es keine Überschneidungen zwischen Bildern gibt.

VI.3.2.3 Filterkonzept

Die Verarbeitung von Bilddaten wird von so genannten Filtern übernommen. Der Name Filter ist im Sinne von „Informationen aus einem Bild filtern“ zu verstehen und ist nicht im Sinne von „Pipes and Filters“ gemeint. Nach Erklärung der allgemeinen Konzepte in diesem Abschnitt werden im Abschnitt VI.3.3 und im Abschnitt VI.3.4 zwei konkrete Anwendungsbeispiele erläutert.
Durch Angabe der Konstanten *FILTER_COUNT* in der Datei *cam_framework.h* kann die Anzahl der Filter festgelegt werden.
Jeder Filter wird durch eine Struktur *camfw_filter*, wie in *cam_framework.h* definiert, beschrieben, und in dem globalen Array *filter* gespeichert.

```
typedef struct
{
 unsigned char is_active;
 unsigned char is_working;
 char work_frame;
 void *callback;
 long int next_id;
} camfw_filter;

extern camfw_filter filter[FILTER_COUNT];
```

Im Folgenden eine kurze Beschreibung der Semantik der Variablen:

- is_active: Dieser Schalter legt fest, ob ein Filter aktiviert ist, d.h. ob er vom Kameraframework zur Datenverarbeitung (in der Funktion camfw_idle in cam_framework.c) aufgerufen wird oder nicht. Dies ist nützlich, wenn ein Filter in bestimmten Spielsituationen nicht benötigt wird. Selbst wenn der Filter nicht aktiv ist, werden ihm vom Kameraframework immer aktuelle Frames zugewiesen. Gültige Werte sind TRUE und FALSE, als Standard wird bei der Initialisierung FALSE gesetzt, d.h. keiner der Filter ist aktiviert. Ein Filter wird mit Hilfe der Funktion camfw_set_filter (in cam_framework.c) aktiviert oder deaktiviert. Alternativ kann hier eine Callback-Funktion angegeben werden, Details folgen weiter unten.
- is_working: Dieser Schalter gibt an, ob ein Filter gerade ein Bild bearbeitet, oder nicht. Das Kameraframework weist dem Filter keine neuen Frames zu, solange er noch arbeitet. Filter sollen so konzipiert sein, dass sie ihre Arbeit in kleinen Stücken (wie zum Beispiel einer Bildspalte pro Aufruf) verrichten, daher sind im Allgemeinen mehrere Aufrufe eines Filters erforderlich, bevor er seine Arbeit beendet. Gültige Werte sind TRUE und FALSE, als Standard wird bei der Initialisierung FALSE gesetzt.

- work_frame: Gibt den Index des Frames in der globalen Framestruktur des Kameraframeworks an, auf dem der Filter im Moment arbeitet. Diese Variable wird in der Funktion configure_filters aktualisiert und ist nötig, damit ein Filter weiß, auf welche Daten er zugreifen soll. Wenn ein Filter im Moment auf keinem Frame arbeiten kann (z.B. weil keine aktuellen Daten vorhanden sind oder der vom Filter gewünschte minimale Frame noch nicht erreicht ist) wird work_frame mit dem Wert -1 belegt. Gültige Werte sind 0..FRAME_COUNT und -1 falls kein Frame bearbeitet wird.
- callback: Diese Variable ist ein Zeiger auf eine Callback-Funktion, die vom Filter nach Verrichtung seiner Arbeit aufgerufen werden soll. Der Filter muss diesen Pointer selbst in den benötigten Zeigertyp casten, hier wird also implizit eine Vereinbarung zwischen Empfänger der Daten und Filter vorausgesetzt. Ein Beispiel findet sich hierzu im Abschnitt VI.3.4. Ist callback auf NULL gesetzt wird kein Callback aufgerufen. Standardinitialisierung ist NULL.
- next_id: Diese Variable gibt die ID des nächsten Frames an, ab dem der Filter wieder aufgerufen werden möchte. Einige Filter sollten so oft wie möglich arbeiten, für andere kann es reichen, nur etwa alle 100 Frames aktiv zu werden. Der Filter ist selbst dafür verantwortlich, diese Variable nach Beendigung seiner Tätigkeit neu zu setzen. Diese Variable wird von der Funktion configure_filters gelesen. Standardinitialisierung ist 1, d.h. der Filter wird nach Vervollständigung des ersten Frames aktiv.
- filter: Das globale Array, über das auf alle Filter zugegriffen werden kann. In der Datei cam_framework.h sind Konstanten (FILTER_TRANSMIT, FILTER_FINDBALL) für die Indizes der einzelnen Filter definiert, um den Zugriff zu erleichtern.

Das Kameraframework steuert die Zuweisung von Frames an die Filter durch die Funktion *configure_filters*, die in der Funktion *camfw_idle* aufgerufen wird.

```
für alle Filter f mit f.is_working == FALSE
    wenn aktuellste FrameID >= f.next_id
      f.work_frame auf aktuellsten Index setzen
    sonst
      f.work_frame = -1
```

Aufgerufen werden die Filter nach der Konfiguration, allerdings nur, wenn sie aktiv sind und ein aktueller Frame zum Arbeiten vorliegt.

```
if(filter[FILTER_FINDBALL].is_active &&
   filter[FILTER_FINDBALL].work_frame != -1)
 filter_findball_grid();
```

```
... (für weitere Filter)
```

Um einen reibungslosen Ablauf zu sichern, sollte jede Filterfunktion die folgende Struktur einhalten:

```
void filter_funktion()
{
 if(filter.is_working == FALSE)
 {
      // TODO: hier Initialisierung machen
      frame[ filter.work_frame ].lock_count++;
      filter.is_working = TRUE;
 }

 // TODO: hier Daten verarbeiten

 wenn fertig
 {
      frame[ filter.work_frame ].lock_count--;
      filter.is_working = FALSE;
      filter.next_id setzen
      ggf. Ergebnis über callptr zurückliefern
 }
}
```

Der erste Aufruf eines Filters auf einem Frame ist daran zu erkennen, dass die Variable *is_working* auf *FALSE* gesetzt ist. Hier können Initialisierungen vorgenommen werden, die einmal pro Frame nötig sind. Um einen reibungsfreien Ablauf mit dem Kameraframework zu gewährleisten, sollte der Filter hier den Frame auf dem er arbeitet sichern. Ansonsten ist nicht garantiert, dass die Bilddaten nicht vom Kameraframework überschrieben werden. Außerdem muss der Filter auch anzeigen, dass er jetzt arbeitet, denn sonst bekommt er vom Kameraframework unter Umständen während seiner Arbeit einen neuen Frame zugewiesen. Bei Filtern, die ihre Arbeit auf einem Frame in einem Aufruf verrichten, sind diese Maßnahmen nicht nötig.
Der Filter verrichtet seine eigentliche Arbeit im Hauptteil. Um den Ablauf nicht allzu sehr zu verzögern, ist es ratsam hier nur einen Teil des Bildes zu bearbeiten, normalerweise einige Spalten und die Verarbeitung dann bis zum nächsten Aufruf zu unterbrechen.
Hat ein Filter alle Arbeit auf dem Frame erledigt und benötigt ihn nicht mehr, muss er den Frame wieder freigeben, den er zuvor gesichert hat. Er sollte dem Kameraframework auch anzeigen, dass er nicht mehr arbeitet, damit er neue Daten zugewiesen kriegt. Der Filter kann durch Setzen von *next_id* angeben, welchen Frame er mindestens zum weiterarbeiten benötigt und so steuern, wie oft er arbeiten möchte. Wurde ein Filter mit einem Callback ausgestattet, muss er dieses nach Beendigung seiner

Arbeit ausführen. So ist sichergestellt, dass die berechneten Ergebnisse auch weiter verarbeitet werden können.

Bei der Arbeit mit Filtern sind zusätzlich zu den bereits genannten Punkten folgende Dinge zu beachten:

- Es ist in der aktuellen Implementierung nicht möglich, gezielt mehrere verschiedene Filter hintereinander auf einem Bild arbeiten zu lassen. Das Kameraframework weist jedem fertigen Filter immer das aktuellste verfügbare Bild zu. Realisieren lässt sich dieses Verhalten jedoch durch einen „Masterfilter“, der einen Filter im o.g. Sinne implementiert und in der Datenverarbeitung nacheinander einzelne Verarbeitungsroutinen startet.
- Filter arbeiten nicht exklusiv auf einem Bild, so dass Änderungen an den Bilddaten ungewünschte Nebeneffekte zur Folge haben können. Generell sollten Filter im laufenden Betrieb mit mehr als einem Filter nur lesend auf Bilddaten zugreifen.
- Es ist unbedingt darauf zu achten, dass ein Filter einen Frame wieder freigibt, nachdem er seine Arbeit beendet hat, da der Frame sonst vom Kameraframework sonst nicht mehr mit neuen Daten gefüllt wird.
- Ein Filter sollte seine Arbeit möglichst schnell tun und langwierige Aufgaben in mehrere Teile aufteilen.

VI.3.3 Transmit-Filter

Der Transmit-Filter wurde von uns geschrieben, um Bilder per serieller Kommunikation an einen Hostcomputer zu übertragen. Die Daten werden dort vom ImageViewer empfangen. Anhand dieses einfachen Beispiels kann man sich die Funktionsweise eines Filters noch einmal gut verdeutlichen.

VI.3.4 Ballfinder

Filter für Ballfinder wurden von uns in den Dateien *filter_findball_grid.c* und *filter_findball.c* implementiert. Nähere Informationen über die Funktionsweise finden sich in den Abschnitten VI.4.1 und VI.4.2.

VI.3.5 Strategiekomponente

Die lokale Strategie gehört eigentlich nicht zur lokalen Kamera, sondern ist ein eigenständiges Modul. Wir haben die Strategiekomponente aber dennoch erstellt, um bei unseren Tests eine saubere Trennung zwischen dem Kameraframework und der Robotersteuerung zu haben. Die Strategie ist so aufgebaut, dass sie von beliebigen Sensoren, die auf dem Roboter montiert sind, Daten anfordern und empfangen kann. In unserem Fall wird nur die Kamera als Sensor benutzt. Die Strategie trifft anhand der Daten Entscheidungen und sendet Steuerbefehle an die Motorsteuerung. Dabei ist die

Funktionalität sehr eingeschränkt und nur auf unsere Kamera zugeschnitten. Wie dies im Detail funktioniert, wird in den nächsten Kapiteln erklärt.

VI.3.5.1 Funktionsweise / Konzept

Die Strategie wird direkt über den Scheduler des Roboters periodisch aufgerufen. Das Verhalten des Strategiemoduls kann über drei verschiedene Parameter festgelegt werden. Diese sollen in Zukunft auch vom Hostsystem über Funk dem Roboter mitgeteilt werden, sodass die globale Strategie entscheiden kann, wann die lokale Strategie aktiv werden und eingreifen soll. Die drei Verhaltensweisen/Parameter sind folgende:

- *DoNothing* ist eine einfache Idle-Schleife in der keine weiteren Aktionen ausgeführt werden. Diese Funktion wird benutzt, wenn das Strategiemodul inaktiv ist, nichts zu tun hat oder auf Ergebnisse wartet.
- *Findball* sucht den Ball auf dem aktuellen Bild und richtet den Roboter mittig zum Ball aus. Wird in dem aktuellen Bild kein Ball gefunden, dreht sich der Roboter auf der Stelle um einige Grad nach rechts, bis ein Ball in sein Sichtfeld kommt.
- *ChaseBall* ist eine einfache Erweiterung von *Findball* und versucht, falls sich der Ball zu weit vom Roboter entfernt, diesem hinterherzufahren. Der Ball wird von dem Roboter verfolgt. Auch hier dreht sich der Roboter zuerst auf der Stelle, falls kein Ball zu sehen ist.

Folgend wird ein exemplarischer Durchlauf für *ChaseBall* beschrieben. Damit *ChaseBall* arbeiten und Entscheidungen treffen kann, benötigt die Strategie die Position und Größe des Balls. Diese fordert das Strategiemodul nun von dem Kamera-Framework an und weist dieses an, ein Bild auszuwerten und die Ballposition zurück zuliefern. In der Zeit, in der die Strategie auf ein Ergebnis wartet, kann diese keine weiteren Aktionen ausführen und sollte auch keine weiteren Aufgaben an andere Module geben. Die Strategie wartet mit *DoNothing* auf eine Antwort des Kameraframeworks. Das Kameraframework teilt der Strategie über eine Callback-Funktion die Antwort und die berechneten Daten mit. Die passende Callback-Funktion, welche das Kameraframework benutzen muss, wird bei dem ersten Aufruf übergeben. Hat die Strategie nun die Werte vom Kameraframework erhalten, in unserem Falle die Position und Größe des Balles, entscheidet die Strategie was zu tun ist. Ist kein Ball gefunden worden, wird die Strategie über die Motorsteuerung eine Drehung einleiten und den Roboter so lange auf der Stelle drehen, bis ein Ball im Sichtbereich des Roboter erkannt wurde. Liegt der Ball im rechten Bildbereich, dreht sich der Roboter noch weiter nach rechts, bis der Ball mittig vor ihm liegt. Ist der Ball im linken Sichtbereich, dreht er sich nach links. Wenn der Ball mittig im Sichtbereich liegt, entscheidet die Strategie, ob der Ball gegebenenfalls zu weit entfernt ist und fährt in Richtung des Balls. Die Strategie steuert den Roboter aber nicht direkt, sondern berechnet den Drehwinkel und die Strecke die nach vorne zu fahren ist und leitet diese Werte an die Motorsteuerung weiter. Der nächste Durchlauf durch die Strategie findet nun erst im nächsten periodischen Aufruf

statt. Die Strategie stellt eine neue Anfrage an das Kameraframework und wird mit den neuen Werten die bereits errechneten Werte, bzw. die bereits erfolgte Ansteuerung korrigieren.
Die Ansteuerung der Motoren wird im nächsten Kapitel erläutert.

VI.3.5.2 Ansteuerung

Wir benutzen die schon existieren Funktionen, die über einen PID-Regler die Motoren ansteuern. Um die Regelung benutzen zu können, muss sie erst über *initControl* initialisiert werden, danach muss in regelmäßigen Abständen *doControl* aufgerufen werden. Dies passiert in der idle-Schleife des Schedulers. Nun kann der Roboter über die Funktionen *doDeltaDreh*, *doWheelSpeed* und *doStop* bewegt werden. *doDeltaDreh* veranlasst dabei eine Drehung um einen anzugebenden Winkel, *doWheelSpeed* setzt die Drehgeschwindigkeit der Räder auf die anzugebenden Werte und *doStop* stoppt den Roboter.
Da wir mit einem v2-Board arbeiten (dem einzigen Board, an das eine Kamera angeschlossen werden kann), dieses aber auf ein v3-Chassis mit entsprechenden Motortreibern montiert haben, funktionieren die Reglerwerte, die für ein v3-Board mit v3-Chassis eingestellt wurden, nicht. Eine korrekte Einstellung dieser Werte nimmt viel Zeit in Anspruch und da außerdem die Ansteuerung des Roboters nur als Proof-Of-Concept des Kamera-Frameworks gedacht war, haben wir uns in Absprache mit den Betreuern dazu entschlossen, die Ansteuerung der Motoren nicht weiter zu verfolgen.
Zusätzlich zur Ansteuerung der Motoren über den vorhandenen Regler haben wir eine Ansteuerung entwickelt, die die PWM-Werte für die Motoren direkt setzt und den Roboter damit fährt. Dies funktioniert auch, nur leider ist es damit nicht möglich, den Roboter aktiv abzubremsen (da dies wieder eine Regelung erfordern würde), es ist nur möglich den Roboter ausrollen zu lassen. Wenn der Roboter sich auf Ballsuche nun dreht, einen Ball erkennt und versucht zu stoppen, so rollt er nur aus und verliert dabei den Ball meistens wieder aus dem Blickfeld.

VI.4 Ballfinder

Unter dem Begriff Ballfinder sammeln wir alle Funktionen und Überlegungen, die damit zu tun haben, den Ball auf dem Kamerabild zu identifizieren und zu extrahieren. In den folgenden Kapiteln geht es in Kapitel VI.4.1 um Klassifizier-Algorithmen und in Kapitel VI.4.2 um Sucher-Algorithmen.
Die von uns erstellten Algorithmen sind alle in dem Tool ImageViewer, welches in Kapitel VI.5.1. genauer beschrieben wird, entwickelt und lauffähig. Das Tool kann die Bilder mit lokalen Algorithmen und den Original-Algorithmen des Roboters darstellen. Außerdem sind Änderungen der verschiedenen Werte während der Laufzeit möglich.

Wir haben auf Bildern des Farbformates YcbCr gearbeitet, welches genauer in Kapitel VI.3.1.3 beschrieben wird. Eine Ausnahme stellen die Bilder im RGB332 Format dar, die für bestimmte Tests benutzt wurden.

VI.4.1 Klassifizierer

Klassifizierer teilen die Bildpixel in verschiedene Klassen ein. In unserem Fall sind dies die Klassen: „Ball“ und „kein Ball“. Die Entscheidung findet dabei auf Pixelebene statt, ohne sich um die Randbedingungen zu kümmern.
Die von uns implementierten Klassifizierer finden sich alle in der Datei check_ball.c. Über die folgenden Variablen, die sich in check_ball.c befinden, lassen sich die einzelnen Klassifizierer aktivieren. Es können auch mehrere gleichzeitig aktiv sein. Die zu überprüfenden Pixel müssen dann von allen aktiven Klassifizierern als Ballpixel gekennzeichnet werden. Die Parameter der einzelnen Klassifizierer werden in den Unterkapiteln erklärt. Es ist ebenfalls möglich die Parameter dynamisch zur Laufzeit mit dem Tool ImageViewer zu verändern.

```
// Parameter für den Ballcheck
// Welche Checkmethode
unsigned char checkLimit = 1;     // Limit-Klassifizierer
unsigned char checkLinear = 1;    // Linear-Klassifizierer
unsigned char checkDistance = 0;  // Euklidischer Klassifizierer
```

VI.4.1.1 Anforderungen

Die Anforderungen an einen Klassifizierer sind einfach zu formulieren. Der Klassifizierer soll schnell und sicher zwischen einem Ballpixel und einem Umgebungspixel unterscheiden können.
So einfach die Anforderungen zu formulieren sind, so schwieriger wird es, diese umzusetzen. Der Ball hat die Grundfarbe orange. Allerdings können durch verschiedene Lichtverhältnisse, falsche Kameraeinstellungen und Reflexionen die Farben von schwarz bis hin zu weiß gehen. Die Schwierigkeit liegt jetzt darin, die Pixel nicht zu optimistisch der Klasse Ball zu zuordnen, da dies dazu führt, dass viele Objekte die eigentlich zum Hintergrund gehören, als Ball erkannt werden und so das Ergebnis verfälschen. Allerdings ist ein zu pessimistisches Vorgehen ebenfalls nicht wünschenswert, da dann unter schlechten Lichtverhältnissen der Ball eventuell nicht gefunden wird, obwohl er auf dem Bild zu sehen ist.
Auf den folgenden Bildern sind einige Problemfälle gut zu erkennen.

Abbildung VI.4-1: Starke Spiegelung auf dem Spielfeld und der Bande

- **Spiegelungen:** Auf diesem Bild ist gut zu erkennen, dass die Reflexionen des Balls sehr stark und in den identischen Farben auf dem Spielfeld zu sehen ist. Dies macht eine vernünftige Ballerkennung extrem schwierig.

Abbildung VI.4-2: Falsche Farbwiedergabe und starke Spiegelung auf dem Feld

- **Farbverfälschungen:** Auf diesem Bild sieht man gut, dass der gleiche Ball sehr stark in der Farbgebung variieren kann. Je nach Lichtverhältnissen schwankt die Farbe zwischen orange, hellrot und dunkelrot. Außerdem kann die interne Bildkorrektur der Kamera, also der Weißabgleich und die Belichtungszeit, das Bild stark verändern. Auf diesem Bild ist ebenfalls gut ein Rauschen zu erkennen, welches durch die mangelhaften Lichtverhältnisse auftritt, in denen das Bild gemacht wurde.

Abbildung VI.4-3: Ball mit weißer Reflexion

- **Reflexionen:** Im oberen Bereich des Balles sind hier gut weiße Reflexionen zu erkennen. Die Bereiche mit den Reflexionen gehören offensichtlich mit zum Ball und sollten von da her der Klasse Ball zugeordnet werden. Das Problem in diesem Fall besteht darin, dass weiße Flächen natürlich nicht nur auf dem Ball zu finden sind, sondern auch die Banden und Spielfeldlinien weiß sind. Eine Zuordnung der weißen Pixel würde somit zwar den gesamten Ball richtig erkennen, aber ebenfalls zu erheblichen falschen Erkennungen führen.

In den folgenden Kapiteln werden wir unsere Ansätze etwas genauer erklären.

VI.4.1.2 Limits

Mit Limits bezeichnen wir unseren ersten Ansatz, den wir in Richtung Klassifizierer unternommen haben. Dabei entscheiden wir einfach über feste Schranken für die Werte Y, Cb und Cr, ob der betrachtete Pixel ein Ballpixel ist. Es wird also geschaut ob der Pixel innerhalb eines fest definierten Limits liegt. Dafür wird bei jedem Pixel geprüft, ob seine Farbwerte innerhalb der Grenzwerte von Ymin, Ymax, Cbmin, Cbmax, Crmin und Crmax liegen. Ist dies der Fall, wird der Pixel der Klasse Ball zugeordnet.
Das folgende Code-Beispiel realisiert solch eine Überprüfung:

```
if(    y >= ymin && y <= ymax &&
       cb >= cbmin && cb <= cbmax &&
       cr >= crmin && cr <= crmax )
 return TRUE;
else
 return FALSE;
```

Die Grenzwerte wurden von uns zu Beginn intuitiv festgelegt und schnell stellten sich einigermaßen brauchbare Werte heraus. Für die Unterscheidung der Klassen ist in diesem Ansatz nur der Cb-Wert von Bedeutung, da die anderen Werte in beliebigen Kombinationen von 0-250 springen. Unsere Versuche haben gezeigt, dass mit den folgenden Werten relativ gute Suchergebnisse erzielt werden:

```
Ymin = 0;      // Untere Schranke für Y
Ymax = 255;    // Obere Schranke für Y
Cbmin = 60;    // Untere Schranke für Cb
Cbmax = 110;   // Obere Schranke für Cb
Crmin = 80;    // Untere Schranke für Cr
Crmax = 255;   // Obere Schranke für Cr
```

Im weiteren Verlauf der PG, haben wir auch probiert die Werte dynamisch an die Umgebung anzupassen. Weitere Informationen dazu finden sich in dem Kapitel VI.4.1.6.

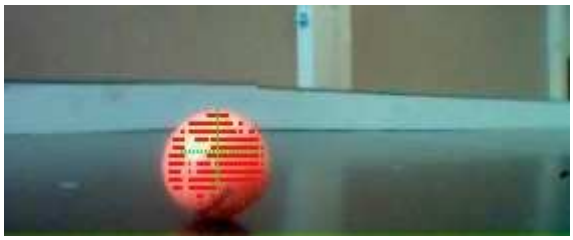


Abbildung VI.4-4: Ergebnisbild des Limit-Klassifizierers

Hier nun ein Bild auf dem das Ergebnis des Limit-Klassifizierers zu sehen. Die roten Pixel sind Pixel, die als Ball erkannt wurden. Die freien Zeilen zwischen den Pixeln und die grünen Linien kommen

durch den Sucher, die in Kapitel VI.4.2 behandelt werden. Auf dem Bild ist gut zu sehen, dass die Reflexionen und die Pixel nahe dem Ballrand nicht als Ball erkannt wurden.

VI.4.1.3 Linear

Der Ansatz des Linearen Klassifizieres basiert auf der Idee mit einer linearen Trennebene, den 3D-Farbraum der YcbCr-Werte in zwei Bereiche - Ball und kein Ball - zu teilen. Wir haben mit dem Tool ColorSpaceExplorer, weiteres in Kapitel VI.5, Bilder eingelesen und gesehen, dass sich die Ballpixel hauptsächlich im unteren Farbraum befinden. Nun mussten wir nur noch eine Ebene definieren über die der Linear-Klassifizierer entscheidet, ob ein Pixel unterhalb oder oberhalb dieser Trennebene liegt. Oberhalb bedeutet in unserem Fall eine Zuordnung des Pixels zur Klasse Ball.
Folgend ein Code-Beispiel wie diese Abfrage aussehen kann:

```
if(    (int)y*YCoeff +
       (int)cb*CbCoeff +
       (int)cr*CrCoeff +
       Offset > 0 )
  return TRUE;
else
  return FALSE;
```

Die folgenden Werte, die wir am Anfang als Standardbelegung benutzt haben, stammen aus Versuchen mit dem ColorSpaceExplorer:

```
YCoeff = 0;          // Koeffizient für Y-Ebene
CbCoeff = -14;       // Koeffizient für Cb-Ebene
CrCoeff = 11;        // Koeffizient für Cr-Ebene
Offset = 100;        // Verschiebungsoffset
```

Auch für den Linear-Klassifizierer haben wir nach Wegen gesucht, eine automatische Kalibrierung zu ermöglichen. Weitere Informationen dazu finden sich ebenfalls in dem Kapitel VI.4.1.6.

Abbildung VI.4-5: Ergebnis des Linear-Klassifizierers

Hier wieder das Ergebnisbild des Linear-Klassifizierers. Auch hier stammen die freien Zeilen zwischen den Pixeln und die grünen Linien vom Sucher. Auf dem Bild ist eindeutig zu sehen, dass

auch Pixel, die nicht zum Ball, sondern zu der Spiegelung gehören, erkannt wurden. Im Ganzen wurde der Ball aber wesentlich besser, vollständiger und runder erkannt als mit dem Limit-Klassifizierer.

VI.4.1.4 Euklidische Abstand

Unser letzter Klassifizierer für Bilder im YcbCr-Format ist der Euklidische Abstand. Mit dem Euklidischen Abstand wird der Abstand des aktuellen Pixels zu einem beliebig gewähltem Punkt, sinnvoll wäre hier ein Punkt, der mittig zu den Ballfarben liegt, berechnet. Auch hier betrachten wir den 3D-Farbraum und legen einen Farbmittelpunkt für den Ball fest. Um diesen Mittelpunkt erstellen wir nun eine Kugel und alle Pixel die innerhalb dieser Kugel, also in einem vorher definierten Abstand zum Farbmittelpunkt liegen, gehören zur Klasse Ball. Alle anderen Pixel zur Klasse kein Ball. Folgend die Formel zur Berechnung des euklidischen Abstands:

$$dist = \sqrt{(x-a)^2 + (y-b)^2 + (z-c)^2}$$

In unserem Programm benutzen wir die Formel allerdings in abgewandelter Form:

```
if(    (y - YCenter)*(y - YCenter) +
       (cb - CbCenter)*(cb - CbCenter) +
       (cr - CrCenter)*(cr – CrCenter)
       < Distance )
 return TRUE;
else
 return FALSE;
```

Wir berechnen in dem obigen Codebeispiel nicht die Wurzel der Summe, sondern arbeiten direkt mit dem quadrierten Ergebnis weiter. Die Wurzelberechnung würde auf dem DSP sehr viel Rechenzeit kosten.

Die folgenden Werte, die wir am Anfang als Standardbelegung benutzt haben, stammen ebenfalls aus Versuchen mit dem ColorSpaceExplorer:

```
YCenter = 175;       // Y-Wert des Mittelpunkts
CbCenter = 85;       // Cb-Wert des Mittelpunkts
CrCenter = 180;      // Cr-Wert des Mittelpunkts
Distance = 5000;     // Kreisradius
```

Abbildung VI.4-6: Ergebnis des eukl. Klassifizierers

Die Erkennung über diesen Ansatz funktioniert leider nicht gut. Die Pixel sind in der Realität nicht kreisförmig um den Mittelpunkt angeordnet. Eine Verbesserungsmöglichkeit dieses Ansatzes wäre es, nicht einen Kreis, sondern eine Ellipse zu benutzen. Über die Betrachtung mit dem ColorSpaceExplorer haben wir gesehen, dass die möglichen Ballpunkte eher schlauchförmig um den Mittelpunkt liegen. Mit der Ellipse ließen sich weitaus mehr Punkt korrekt zu ordnen, da diese in bestimmte Richtungen gestreckt werden kann. Da wir aber in dem Ansatz der linearen Trennebene mehr Potenzial vermuten, haben wir diesen Ansatz nicht weiter verfolgt.

VI.4.1.5 Table-Lookup bei RGB332

Diese Klassifizierer setzen eine Bild im RGB332-Format voraus. In diesem Format kommen nur 256 Farben vor und man kann daher im Voraus eine manuelle Auswahl möglicher Ballfarben treffen. Der Klassifizierer muss zur Laufzeit dann nur noch vergleichen, ob die Farbe des Pixels als mögliche Ballfarbe gespeichert wurde. Die Zuordnung der Ballfarben zu den Klassen sind in einem 0-1 Array gespeichert. Dabei stellt der Index die Farbe und der Wert die Zuordnung zur Klasse Ball dar. Ein *rgb332lookup[1]=TRUE* bedeutet also, dass die Farbe 1 zu der Klasse Ball gehört. Auf Grund der manuellen Vorauswahl benötigen wir für die Entscheidung nur 1 Zeile Quelltext:

```
return rgb332lookup[index];
```

VI.4.1.6 Kalibrierung

Die Idee zu einer automatischen Kalibrierung kam uns in der Zeit als wir für den Limit-Klassifizierer Werte gesucht haben. Wir wollten eine Möglichkeit finden, dass sich der Roboter automatisch an gegebene Lichtverhältnisse und verschiedene Ballfarben anpasst. Im folgendem werden unsere Versuche beschrieben, solch eine Kalibrierung umzusetzen. Dabei haben wir einmal eine Kalibrierung für den Limit Klassifizierer und eine für den Linear Klassifizierer versucht.

VI.4.1.6.1 Kalibrierung für den Limit Klassifizierer

Die Idee dieser Kalibrierung ist es, zwei bis auf den Ball identische Bilder zu nutzen, um eine durchschnittliche Ballfarbe zu berechnen. Es werden dafür zwei Bilder, das erste mit Ball, das zweite ohne Ball, gemacht. Auf diesen Bildern werden nun die Bereiche mit den größten Unterschieden zueinander bestimmt. Die Bereiche mit den größten Unterschieden sind mit hoher Wahrscheinlichkeit die Pixel auf denen in Bild 1 der Ball liegt und auf Bild 2 der Ball nicht zu sehen ist. Nun werden die eigentlichen Farbwerte von Bild 1 aufaddiert und anschließend Durchschnittswerte für Y, Cb und Cr bestimmt. Diese Mittelwerte, inklusive einer Toleranz, benutzen wir dann für den Limit-Klassifizierer. Leider waren diese Durchschnittswerte nicht wirklich für den Betrieb geeignet, da wie in Kapitel VI.4.1.2 bereits erwähnt, nur der Cb Wert wirklich eingeschränkt ist. Die anderen Werte laufen von 0 bis 250, sodass die Durchschnittswerte nicht zu benutzen waren.

VI.4.1.6.2 Kalibrierung für den Linear Klassifizierer

Die im Folgenden beschriebenen Ausführungen sind nur eine Möglichkeit, den linearen Klassifizierer zu kalibrieren. Die Resultate waren allerdings so gut, dass wir keine anderen Methoden in Betracht gezogen haben.

Unserer per Hand klassifizierter Testdatensatz mit sechs Bildern liefert uns eine Menge von Punkten in einem dreidimensionalen Raum (in diesem Falle [Y, Cb, Cr]), die der Klasse „Ball“ oder der Klasse „NoBall“ angehören. Gesucht ist eine lineare Trennebene, die diese beiden Klassen möglichst gut voneinander trennt.

Bei der Trennung ist zu berücksichtigen, dass die Klasse „Ball“ in den Testdaten viel seltener vorkommt. Ein Fehler auf dieser Seite ist also höher zu gewichten, da man sonst trivial schon oft eine Genauigkeit von über 90% bekommt, wenn man jeden Pixel als „NoBall“ klassifiziert.

Auch bei der geringen Menge von sechs Bildern ergibt sich ein Trainingsdatensatz von ca. 90.000 Tupeln. Probleme entstehen auch, da einzelne Farbwerte in einem Bild als Ballpixel, in einem anderen Bild als Hintergrundpixel (z.B. im Schatten) auftreten können.

Aus diesem Grund wurde bei den folgenden Überlegungen ein reduzierter Datensatz betrachtet. Ein „Ball“ – Tupel wurde nur benutzt, wenn es häufiger als Ballpixel, als als Hintergrundpixel erkannt wurde. Ein „NoBall“ – Tupel wurde nur benutzt, wenn es häufiger als Hintergrundpixel, als als 10 + Ballpixel erkannt wurde. Diese Grenzen wurden relativ willkürlich gesetzt, die genauen Auswirkungen bleiben zu untersuchen. Der Testdatensatz reduzierte sich durch diese Maßnahmen auf 11896 Testtupel, davon 8162 der Klasse „Ball“ und 3734 der Klasse „NoBall“.

Im Folgenden ist eine Visualisierung der Trainingsdaten durch den ColorSpaceExplorer zu sehen, dabei bedeutet ein rotes Pixel ein Tupel der Klasse „Ball“, ein blaues Pxiel ein Tupel der Klasse „NoBall“.

VI.4.1.7 Vergleich

Die folgend beschriebenen Vergleiche beruhen auf unserer subjektiven Einschätzung und den objektiven Ergebnisse des AlgoAnalyzers (siehe Kapitel VI.5). Wir haben die Tests mit einer Auswahl an Bildern durchgeführt um ein möglichst breites Spektrum an Varianten abzudecken. Die folgenden sechs Bilder haben als Basis für unsere Vergleiche gedient.

Abbildung VI.4-7: Testbild 1

Auf diesem Bild befindet sich ein Roboter im rechten Bereich und der Ball ist relativ klein und spiegelt sich relativ stark auf dem Spielfeld.

Abbildung VI.4-8: Testbild 2

Ein großer Ball mittig im Bild mit wenigen Reflexionen.

Abbildung VI.4-9: Testbild 3

Mittelgroßer Ball am rechten Bildrand. Auch hier eine Spiegelung auf dem Spielfeld.

Abbildung VI.4-10: Testbild 4

Dieses Bild entspricht dem Testbild 1 jedoch ist hier kein Roboter im Sichtbereich.

Abbildung VI.4-11: Testbild 5

Mittelgroßer Ball mittig im Sichtbereich jedoch mit starken Reflexionen, fast die gesamte obere Ballhälfte ist weiß.

Abbildung VI.4-12: Testbild 6

Mittelgroßer Ball mittig im Sichtbereich und ohne starke Reflexionen und Spiegelungen.

Zuerst haben wir eine subjektive Einschätzung der Ergebnisbilder vorgenommen, die nun im Folgenden aufgeführt wird. Die Bilder wurden über den ImageViewer mit den drei Algorithmen erstellt. Dabei haben wir die folgenden Parameter benutzt. Diese Werte entsprechen unseren intuitiv gewählten Werten.

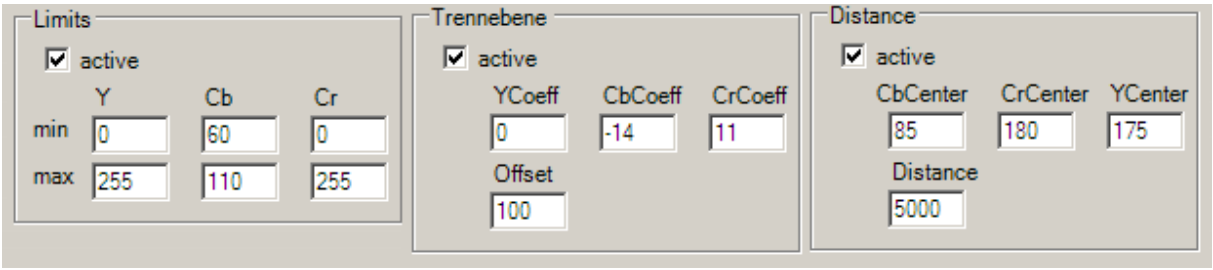


Abbildung VI.4-13: Parameter

Abbildung VI.4-14: Ergebnisbild 1 (Limit/Linear/Euklid)

Der Limit-Klassifizierer erkennt auf diesem Bild nur einen geringen Teil des Balls, der Linear-Klassifizierer erkennt große Teile des Balls und auch die Spiegelung, während der Euklidische-Klassifizierer zum großen Teil die Umgebung erkennt.

Abbildung VI.4-15: Ergebnisbild 2 (Limit/Linear/Euklid)

Abbildung VI.4-16: Ergebnisbild 6 (Limit/Linear/Euklid)

Auf den beiden Bildern erkennt man sehr gut die Stärken des Linear-Klassifizierers. Fast der gesamte Ball wurde erkannt und die Ballränder sind scharf angegrenzt. Der Limit-Klassifizierer hat auch hier Probleme mit den helleren und dunkleren Bereichen des Balls. Der Euklidische-Klassifizierer schneidet zumindestens auf dem Ergebnisbild2 ein wenig besser ab aber auch dieser kann den Ball nicht komplett erkennen.

Abbildung VI.4-17: Ergebnisbild 3 (Limit/Linear/Euklid)

Abbildung VI.4-18: Ergebnisbild 4 (Limit/Linear/Euklid)

Abbildung VI.4-19: Ergebnisbild 5 (Limit/Linear/Euklid)

Die drei Bilder sind vom Ergebnis her sehr ähnlich. Der Limit-Klassifizierer erkennt nie den kompletten Ball, findet allerdings auch nur sehr wenig Pixel außerhalb des Balls. Der Linear-Klassifizierer erkennt den Ball bis auf die weißen Reflexionen komplett. Die Spiegelungen auf dem Spielfeld erkennt dieser ebenfalls als Ball. Der Euklidische Klassifizierer erkennt nur teilweise den Ball und hat oftmals Pixel außerhalb des Balls oder Spiegelung.

An Hand dieser Bilder sind wir zu der subjektiven Einschätzung gekommen, dass für unsere Zwecke der Linear-Klassifizierer am besten geeignet ist. Dieser erkennt zwar zu großen Teilen die Spiegelung doch dies ist eher zu vernachlässigen, da für den Spielbetrieb hauptsächlich die X-Koordinate wichtig ist und diese durch die Spiegelung nicht verändert wird. Die Spiegelung ist ja immer vor dem Ball zu sehen. Und solange eine Ballerkennung nur über die Ballfarbe läuft wird die Spiegelung immer erkannt, da diese die original Ballfarbe aufweist.
Die Werte, mit denen die Vergleiche durchgeführt wurden, sind ebenfalls noch nicht optimal. Die automatische Kalibrierung verspricht in diesem Bereich noch weitere Optimierungen.
Als nächstes betrachten wir die Ergebnisse der Analyse des AlgoAnalyzers. Was der AlgoAnalyzer für diese Analyse benötigt und wie er die Werte bestimmt, wird in Kapitel VI.5 erklärt.

```
Result:

                                    output CheckPixel
                  ¦ +                               ¦ -
                --+---------------------------------+-------------------------
correct         + ¦ true pos  (BallCorrect)         ¦ false pos (BallIncorrect)
classification  - ¦ false neg (NoBallIncorrect)     ¦ true neg  (NoBallCorrect)

     +     -                    +     -   Limits
+ 5921  3341              + 63.93 36.07   0<=Y<=255 60<=Cb<=110 80<=Cr<=255
-  464 76921              -  0.60 99.40   Correct classified: 95.61

     +     -                    +     -   Linear
+ 8695   567              + 93.88  6.12   Y:0 Cb:-14 Cr:11 Off:100
- 5162 72223              -  6.67 93.33   Correct classified: 93.39

     +     -                    +     -   Distance
+ 6227  3035              + 67.23 32.77   Y:175 Cb:85 Cr:180 Dist:5000
- 3884 73501              -  5.02 94.98   Correct classified: 92.01
```

Abbildung VI.4-20: Ergebnisse des AlgoAnalyzers

Aus den obigen Ergebnissen geht hervor, dass der Limit-Klassifizierer nur 64% der Ballpixel auch als Ballpixel erkannt hat. Die übrigen Hintergrund Pixel wurden gut erkannt. Dieses Ergebnis unterstützt unsere Einschätzung, dass der Limit-Klassifizierer nur Teile des Balls und nie den ganzen erkennt und so gut wie nie Fehlpixel in der Umgebung falsch klassifiziert. Der Limit-Klassifizierer benötigt pro Pixel 6 Vergleiche um eine Klassifizierung vorzunehmen. Ist also sehr schnell, da keine zusätzlichen Berechnungen vorgenommen werden müssen.
Der Linear-Klassifizierer hat bei der Ballerkennung ein sehr gutes Ergebnis von 94% erreicht. Es werden also fast alle Ballpixel als Ball erkannt. Die fehlenden 6% stellen die weißen Reflexionen, die natürlich nicht als Ball erkannt werden können. Die falsch klassifizierten 7% bei den Umgebungspixeln kommen durch die Spiegelung. Diese wird ebenfalls als Ball erkannt, gehört aber streng genommen natürlich nicht zum Ball. Der Linear-Klassifizierer benötigt pro Pixel drei Multiplikationen, zwei Additionen und einen Vergleich. Allerdings lässt sich die Formel noch umstellen, sodass durch einen Koeffizienten geteilt wird und nun nur noch zwei Multiplikationen, zwei Additionen und einen Vergleich benötigt werden. Ein weiterer Sonderfall tritt auf, wenn in dem

Ergebis der Division ganze Zahlen auftreten. Nun können wir das ganze auf zwei Additionen, eine Multiplikation und einen Vergleich reduzieren.
Der Euklidische Klassifizierer erkennt ebenfalls nur einen Teil, nämlich 67%, des Balls, zusätzlich dazu findet dieser ungefähr 5% Umgebungspixel die er als Ball erkennt. Der euklidische Klassifizierer benötigt pro Pixel 3 Multiplikationen, 2 Additionen und 1 Vergleich.
Die hohe Rate der Umgebungserkennung im Vergleich zu den oftmals bescheidenen Werten der Ballerkennung ergibt sich dadurch,, dass es im Verhältnis zu den Ballpixeln sehr viel mehr Umgebungspixel gibt. In unseren Bildern gab es insgesamt 77385 Umgebungspixel aber nur 9262 Ballpixel.
Die Ergebnisse unterstützen unsere subjektive Einschätzung, da aus diesen ebenfalls hervorgeht, dass der Linear-Klassifizierer die besten Erfolge erzielt und eine hohe Rate der korrekten Ballerkennung aufweist.

VI.4.2 Sucher

Die Aufgabe des Suchers ist es, einen möglichst effizienten Weg über das Bild zu finden und zu entscheiden, ob und wo ein Ball gefunden wurde. Der Weg legt fest, welche und wie viele Pixel des Bildes betrachtet werden. Der Sucher benutzt zum betrachten der einzelnen Pixel die Klassifizierer und arbeitet mit deren Ergebnissen. Ein Sucher kann also nur dann effizient arbeiten, wenn auch der benutzte Klassifizierer effizient ist.

VI.4.2.1 Anforderungen

Ein effizienter Sucher sollte aus Geschwindigkeitsgründen so wenig Pixel wie möglich aber so viele Pixel wie nötig betrachten, sodass er eine sichere Aussage machen kann, ob und wo ein Ball gefunden wurde. Die Abwägung findet also zwischen Geschwindigkeit und Genauigkeit statt.

Abbildung VI.4-21: Kleiner Ball

- **Ball ist sehr klein:** Ein Problem, dass auftreten kann, sind weit entfernte Bälle. Diese sind auf dem Kamerabild nur wenige Pixel groß und können von dem Sucher unter Umständen übersehen/übersprungen werden.
- **Sucher zu langsam:** Falls der Sucher zu langsam arbeitet, hat sich die Spielsituation bereits entscheidend verändert, bis die Strategie Werte zurückgeliefert bekommt. Ein Ball wird somit an einer Stelle erkannt, an der er vor einigen Sekunden lag.

- **Störpixel:** Störpixel oder generelles Rauschen im Bild dürfen die Position oder Auswertung nicht beeinflussen.
- **Kein Ball:** Der Normalfall im Betrieb ist es, dass kein Ball im Blickfeld des Roboters zu sehen ist. Dies muss schnell erkannt werden und es sollte keine weitere Zeit auf die Suche nach dem Ball aufgewendet werden.

VI.4.2.2 Reguläres Gitter mit lokaler Suche

Unser erster Sucher läuft in einer vorher definierten Schrittweite durch die Spalten des Bildes. In jedem Schritt übergibt der Sucher den aktuellen Pixel an den Klassifizierer. Das Ergebnis des Klassifizierers bestimmt dann das weitere Vorgehen. Es gibt nun 2 Fälle:

1. **Pixel wurde nicht als Ballpixel klassifiziert**. In diesem Fall werden keine weiteren Aktionen unternommen. Die Suche wird fortgesetzt.
2. **Pixel wurde als Ballpixel klassifiziert**. Der Sucher stoppt seinen normalen Ablauf und startet eine lokale Suche. Es werden in der aktuellen Spalte alle Pixel, die über und unter dem aktuellen Pixel liegen überprüft. Solange nun Ballpixel gefunden werden geht der Sucher mit einer kleineren Schrittweite in beide Richtungen, also über und unter dem aktuellen Pixel, weiter. Findet der Sucher nun einen Pixel, der nicht als Ballpixel klassifiziert wurde, stoppt die lokale Suche in dieser Richtung an dieser Stelle. Die lokale Suche in die jeweils andere Richtung wird fortgesetzt bis dort ebenfalls ein Nicht-Ball-Pixel gefunden wurde oder die Spalte zu Ende ist. Wurde die lokale Suche in beiden Richtungen gestoppt, überprüft der Sucher die Länge diese Pixelstrecke und falls er vorher noch Strecke gefunden hat, die länger war, merkt er sich die Länge und Position. Die Suche geht nun am letzten gefundenen Ballpixel mit der normalen Sprungweite weiter.

Die Suche endet, wenn das gesamte Bild durchlaufen wurde. Falls der Sucher nun eine Pixelreihe gefunden hat, überprüft er die Länge mit einer minimalen Ballgröße, um Bildrauschen und ähnliches auszugrenzen. Ist die Reihe lang genug, um ein Ball zu sein, nimmt der Sucher an, dass die gefundene Reihe die Spalte der Ballmitte ist. Unter normalen Umständen wäre dies die längste Strecke in Ballfarbe. Nun berechnet der Sucher den Mittelpunkt und die Ballgröße. Der Ballmittelpunkt ist der Mittelpunkt der gefundenen Reihe und die Ballgröße entspricht der Länge dieser Reihe. Der Sucher liefert jetzt entweder als Ergebnis, dass kein Ball gefunden wurde oder das ein Ball mit dem berechneten Mittelpunkt und Größe gefunden wurde.

Die folgende Grafik stellt den Suchablauf schematisch dar. Die blauen Punkte sind die betrachteten Pixel in der normalen Suche, die grünen gefundene Ballpixel in der lokalen Suche. Die Leerspalten dienen nur der Übersichtlichkeit, im Suchalgorithmus wird natürlich jede Spalte besucht.

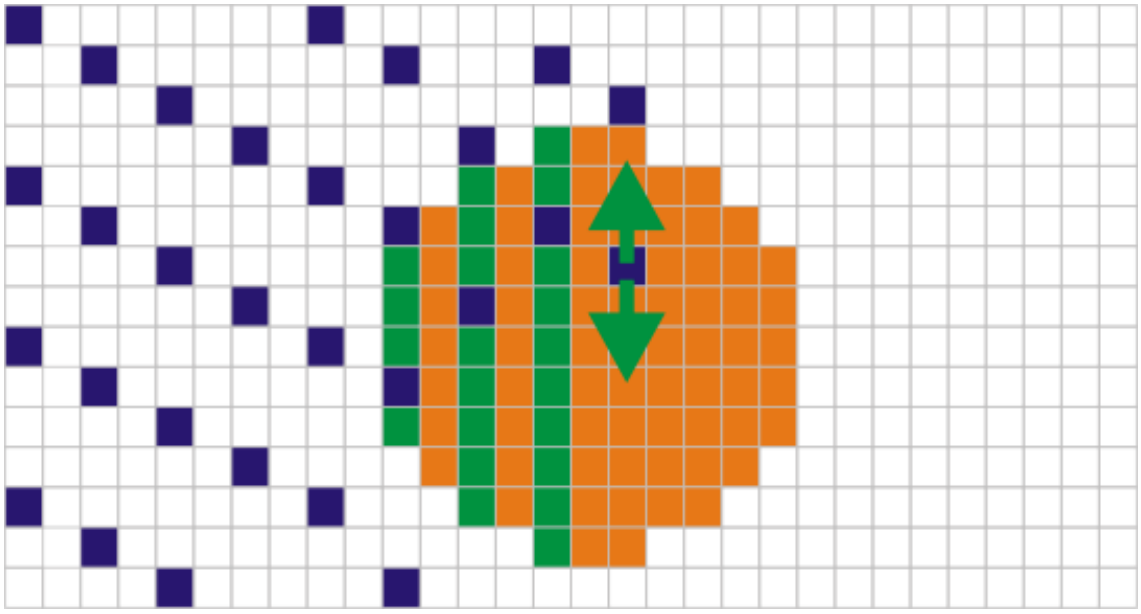


Abbildung VI.4-22: Schematischer Ablauf der lokalen Suche

Die Implementierung dieses Suchers findet sich in der Datei filter_findball.c. Über folgende Parameter kann der Sucher eingestellt werden.

```
#define DEBUG
#define TRANSMIT

// defines für den Sucher
#define PIX_SIZE           4
#define MIN_BALL_SIZE      5
#define DEEP_SEARCH_SIZE   2
```

Die Defines *DEBUG* und *TRANSMIT* werden nur für Testzwecke benötigt, im laufenden Betrieb können diese abgeschaltet werden.

- *DEBUG*, aktiviert Ausgaben die verschiedene Parameter zur Laufzeit anzeigen
- *TRANSMIT*, durch diese Option werden die aktuellen Bilder inklusive der Sucher-Visualisierung an das UART-Interface übermittelt. **Diese Option verbraucht sehr viel Rechenzeit und sollte nur zu Tests aktiviert werden.**
- *PIX_SIZE*, legt die Bytes pro Pixel für den aktuellen Farbmodus fest. Diese Option muss für YCbCr immer auf 4 stehen, da wir immer die 2 benachbarten Pixel, die sich die CbCr-Werte teilen, gleichzeitig betrachten.
- *MIN_BALL_SIZE*, die minimale Größe des Balles und gleichzeitig die Sprungweite für die normale Suche. Ein möglicher Ball muss also aus gleich vielen oder mehr Pixeln bestehen.
- *DEEP_SEARCH_SIZE*, die Schrittweite für die lokale Suche.

Der Vorteil dieses Algorithmus ist, dass durch die lokale Suche relativ genau die tatsächliche Größe abgeschätzt werden kann. Es werden bei einer Suche fast immer alle Ballpixel gefunden, sodass z.B. die Kontur des Balles gut überprüft werden kann.
Der größte Nachteil und auch der Grund, warum standardmäßig immer der Sucher mit der Schwerpunktberechnung benutzt wird, ist, dass die Laufzeit des Algorithmus nicht fest und in vielen Fällen zu groß ist. Auf einem Bild mit keinem oder nur einem kleinen Ball terminiert der Algorithmus relativ schnell, da nur wenige lokale Suchen gemacht werden müssen. Bei einem Bild mit einem großen Ball muss der Algorithmus sehr häufig in die lokale Suche springen und jedes Ballpixel überprüfen.

VI.4.2.3 Reguläres Gitter mit Schwerpunktberechnung

Dieser Ansatz wurde in Hinblick auf minimale Laufzeit entwickelt. Die Pixel werden hierbei ebenfalls mit einer definierten Sprungweite überprüft, jedoch findet keine lokale Suche statt. Die Idee ist, dass an einer Stelle mit vielen möglichen Ballpixeln wahrscheinlich auch der Ball zu finden ist. Daher merkt sich der Sucher nicht die Position eines jeden Pixels, sondern summiert die Positionen und berechnet den Schwerpunkt. Wurde das gesamte Bild überprüft, wird die Anzahl der gefundenen möglichen Ballpixel mit einem vorher definierten Minimalwert verglichen. Wird dieser Wert überschritten, meldet der Sucher einen gefundenen Ball und gibt den berechneten Schwerpunkt als Position und die größte Anzahl von gefundenen Pixeln in einer Spalte als Größe zurück. Falls der Minimalwert nicht überschritten wird, meldet der Sucher, dass kein Ball gefunden wurde.

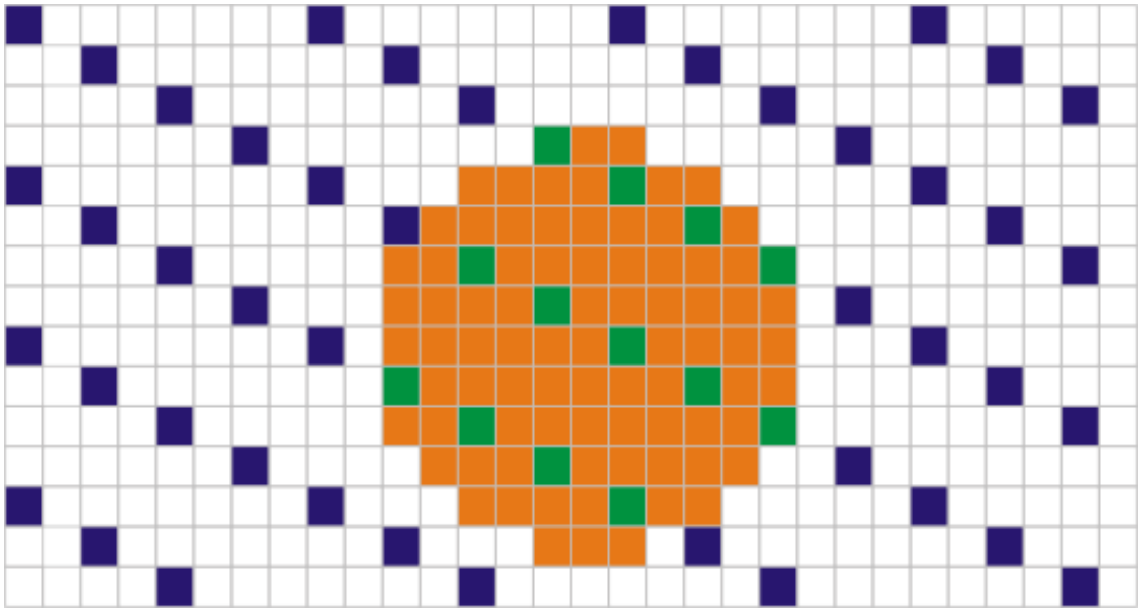


Abbildung VI.4-23: Schematischer Ablauf der Suche

Die Implementierung findet sich in der Datei filter_findball_grid.c und kann über die folgenden Parameter eingestellt werden.

```
#define MARKPOSITIVE
#define MARKCENTER
#define DEBUG
#define TRANSMIT

// Parameter, um den Ballfinder zu konfigurieren
#define MIN_HITS           20
#define STEP_SIZE          16
#define REDUCE_NOISE       0
```

Die Defines *MARKPOSITIVE*, *MARKCENTER*, *DEBUG* und *TRANSMIT* werden nur für Testzwecke benötigt, im laufenden Betrieb können diese abgeschaltet werden.

- *MARKPOSITIVE*, markiert gefundene Ballpixel. Wird nur übermittelt, falls *TRANSMIT* aktiv ist.
- *MARKCENTER*, markiert den berechneten Mittelpunkt des Balls. Wird nur übermittelt, falls *TRANSMIT* aktiv ist.
- *DEBUG*, aktiviert Ausgaben die verschiedene Parameter zur Laufzeit anzeigen
- *TRANSMIT*, durch diese Option werden die aktuellen Bilder inklusive der Sucher-Visualisierung an das UART-Interface übermittelt. **Diese Option verbraucht sehr viel Rechenzeit und sollte nur zu Tests aktiviert werden.**
- *MIN_HITS*, Minimalwert, der festlegt wie viele Pixel mindestens als mögliche Ballpixel erkannt werden müssen, damit der Algorithmus sinnvoll arbeitet.
- *STEP_SIZE*, Schrittweite mit der über das Bild gegangen wird. Wird in diesem Algorithmus anders berechnet als in dem vorherigen, da sich hier der Wert auf die einzelnen Farbwerte und nicht direkt auf einzelne Pixel bezieht. Der vergleichbare Wert für den vorherigen Algorithmus wäre in diesem Fall 4.
- *REDUCE_NOISE*, ermöglicht eine Überprüfung, ob umliegende Pixel ebenfalls Ballpixel sind. Durch diese Option kann Rauschen entfernt werden, da ein Pixel nur noch als positiv gekennzeichnet wird, wenn die umliegenden Pixel ebenfalls positiv sind. Mögliche Werte und deren Kombinationen sind 1 für links, 2 für rechts, 4 für oben und 8 für unten. Die Überprüfung ob alle angrenzenden Pixel positiv sind wäre 15.

Der Vorteil dieses Algorithmus ist seine Geschwindigkeit und feste Laufzeit. Egal wie viele positive Pixel gefunden werden, der Algorithmus terminiert immer nach einer festen Anzahl von Schritten.
Der Nachteil ist die Mittelpunktsbestimmung über die Berechnung des Schwerpunkts. Dieses kann dazu führen, dass der berechnete Mittelpunkt außerhalb des Balles liegt. Ausserdem ist diese Berechnung bei kleinen Bällen oftmals nicht sehr genau.

Jedoch hat diese im realen Umfeld kaum Auswirkungen, da es bei einem bewegten Ball eher auf die Geschwindigkeit als auf die Genauigkeit ankommt.
Ein weiteres Problem, welches dieser Sucher noch hat, ist, dass durch den normalen Ansatz keine Ballgröße bestimmt werden kann. Diese wird aber unter Umständen benötigt und wurde im nachhinein hinzugefügt. Die Größenberechnung funktioniert nun sehr ähnlich zu der Größenberechnung des ersten Suchers. Für jede Spalte, die gerade über das Gitter durchlaufen wird, merken wir uns die maximale Anzahl positiver Hits in der Variablen *maxHitsPerColumn*. Terminiert der Algorithmus, multiplizieren wir die maximale Anzahl von Hits in einer Spalte mit der STEP_SIZE und erhalten so einen Näherungswert für die Größe.

VI.5 Tools

VI.5.1 ImageViewer

VI.5.1.1 Anforderungen

Der ImageViewer stellt ein Tool dar, das genutzt werden soll, um Bilddaten, die direkt vom DSP über die serielle Schnittstelle gesendet werden, darzustellen. Zusätzlich sollten diese Bilddaten in Dateien abgespeichert und zu einem späteren Zeitpunkt wieder abgerufen werden können. Es sollten die verschiedenen Datenformate, die die Kamera unterstützt, darstellbar sein. Um die Entwicklung von Ballfindungsalgorithmen zu erleichtern, sollte es möglich sein, Algorithmen auf dem Hostrechner zu testen, indem sie auf die empfangenen Bilder direkt angewendet werden.

VI.5.1.2 Features

Der ImageViewer wurde in der .NET Programmiersprache C# geschrieben. Wie in den Anforderungen beschrieben wurde viel Wert auf eine leichte Erweiterbarkeit und eine gute Bedienbarkeit gelegt.
Die Beschreibung der Funktionen des ImageViewers erfolgt durch eine schrittweise Erläuterung seiner graphischen Oberfläche.

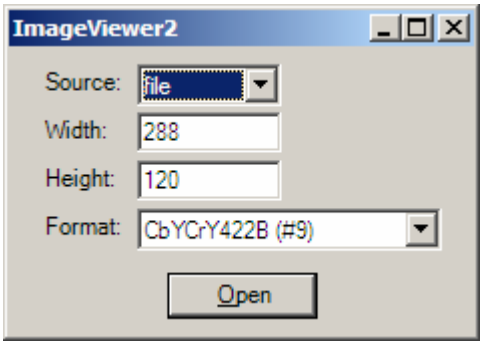


Abbildung VI.5-1: Hauptdialog

Der Hauptdialog erscheint, wenn das Programm gestartet wird. Hier können die Einstellungen für die Quelle der Daten, für die Dimension der erwarteten Bilddaten und für das erwartete Datenformat vorgenommen werden. Als mögliche Quellen können Dateien oder die serielle Schnittstelle ausgewählt werden. Wird die serielle Schnittstelle als Quelle ausgewählt, so werden die empfangenen Frames in Dateien mitgeschnitten, um sie später noch einmal betrachten zu können. Die Dimension der Bilddaten kann frei gewählt werden. Als Datenformate werden CbYCrY 422B, RGB888 und RGB332 unterstützt. Eine Beschreibung der Formate CbYCrY422B und RGB332 findet sich im Kapitel VI.3.1.3. Das RGB888-Format sieht für den Rot-, Grün- und Blauanteil der Farbe jeweils 8 Bit vor, benötigt also pro Pixel 3 Byte.
Nachdem der Benutzer im Hauptdialog die gewünschten Einstellungen vorgenommen hat, klickt er auf „Open“. Ist als Quelle „File“ gewählt, öffnet sich nun ein Dialog, in dem eine oder mehrere Dateien zur Betrachtung ausgewählt werden können. Für jede gewählte Datei öffnet sich danach ein Bildbetrachtungsfenster. Ist als Quelle „serial“ gewählt, so wird programmintern die serielle Schnittstelle geöffnet und ein Bildbetrachtungsfenster geöffnet.

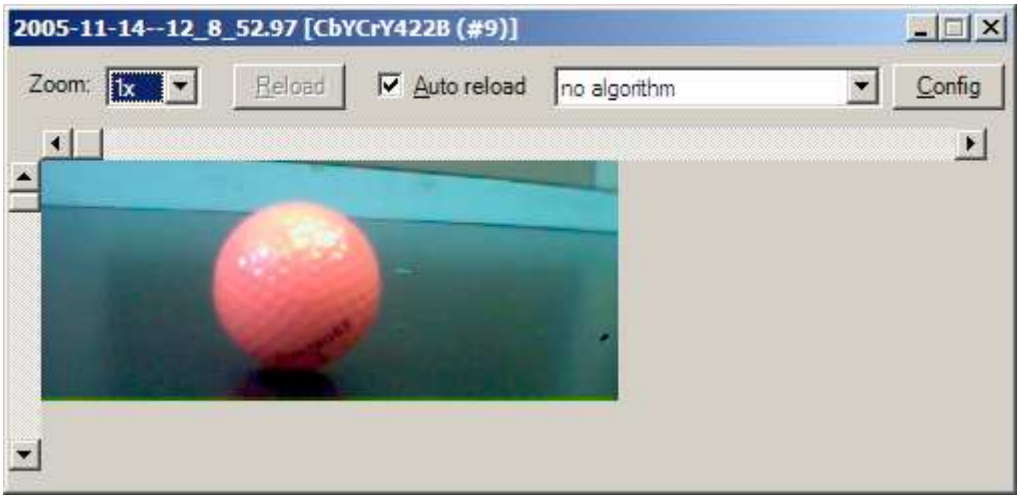


Abbildung VI.5-2: Bildbetrachtungsfenster

Das Bildbetrachtungsfenster dient zur Darstellung der über die serielle Schnittstelle empfangenen oder aus einer Datei gelesenen Bilddaten. In der Titelzeile werden der gewählte Dateiname (im Fall der seriellen Schnittstelle nur „serial“) und das gewählte Datenformat angezeigt. Über den Zoom-Faktor kann die Vergrößerung des Bildes gesteuert werden, um Details genauer betrachten zu können. Der Benutzer hat die Wahl zwischen einem automatischen („Auto reload“) oder einem manuellen Neuladen (Button „Reload“). Ist das automatische Neuladen aktiv, so wird das Bild sofort aktualisiert, wenn genügend Daten für ein neues Bild über die serielle Schnittstelle empfangen wurden. Ist als

Quelle eine Datei gewählt hat das automatische Neuladen keine Funktion. Ist das manuelle Neuladen aktiv, so wird das angezeigte Bild erst aktualisiert, wenn auf den „Reload“-Button geklickt wird. Dann wird das aktuellste, komplett empfangene Bild aus der seriellen Schnittstelle dargestellt bzw. es wird die gewählte Datei neu gelesen und dargestellt. Über das Algorithmen-Auswahlfeld kann gewählt werden, welcher Algorithmus auf das angezeigte Bild angewendet werden soll. Ein Algorithmus bekommt das von der Kamera aufgenommene Bild vorgelegt und kann es verändern, Informationen daraus extrahieren und anzeigen. Der Hauptzweck der Unterstützung von Algorithmen besteht darin, verschiedene Algorithmen zur Ballfindung zu testen. Über den „Config“-Button kann der Algorithmus konfiguriert werden, sofern er dies unterstützt.

Um verschiedene Ballfinder objektiv vergleichen zu können, ist es nötig eine manuelle Markierung des Balls in verschiedenen Testbildern vornehmen zu können. Ein Ballfinder enthält u.a. eine Komponente, die für einen Pixel bestimmen muss, ob er zum Ball gehört oder nicht (siehe Kapitel VI.4). Um einen Vergleich verschiedener Klassifizierer vornehmen zu können, muss also für jedes Pixel in den Testbildern manuell festgelegt werden, ob es zum Ball gehört oder nicht. Dies kann mit Hilfe des ImageViewers gemacht werden. Eine weitere Möglichkeit die manuell separierten Daten zu benutzen besteht darin, Verfahren des maschinellen Lernens wie z.B. Support Vector Machines zu benutzen, um optimale Einstellungen für verschiedene Klassifizierer zu bestimmen (siehe auch Kapitel VI.4.1.6). Neben dem Klassifikationsteil eines Ballfindungsalgorithmus sollte auch der gesamte Algorithmus bewertet werden können. Dazu kann die Ausgabe des Algorithmus (Position und Größe des Balls) benutzt werden und mit einer manuell festgelegten Position und Größe verglichen werden.

Sobald ein Bildbetrachtungsfenster geöffnet ist, kann mit einem Mausklick mit der linken Taste die manuelle Klassifikation gestartet werden. Der ImageViewer erwartet insgesamt drei Klicks, mit denen der Rand des Balls markiert wird. Aus diesen drei Punkten berechnet der ImageViewer dann einen Kreis, der den Ball genau umfasst. Der Mittelpunkt des markierten Kreises stellt also gleichzeitig auch den Mittelpunkt des Balls dar. Alles was innerhalb des Kreises liegt, gehört zum Ball, alles was außerhalb liegt nicht. Mit den drei Klicks hat der Benutzer alle nötigen Informationen gegeben, um die gewünschten Daten (Position und Größe des Balls, Klassifikation der Pixel in Ballpixel und Nicht-Ballpixel) durch den ImageViewer berechnen zu lassen. Dieses Verfahren ist in Abbildung VI.5-3 und Abbildung VI.5-4 dargestellt.

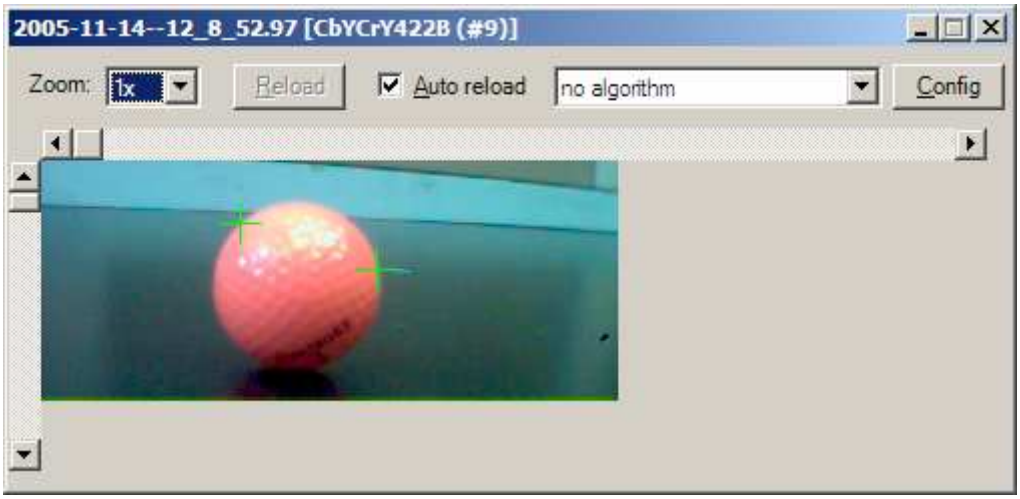


Abbildung VI.5-3: Beginn einer manuellen Separation

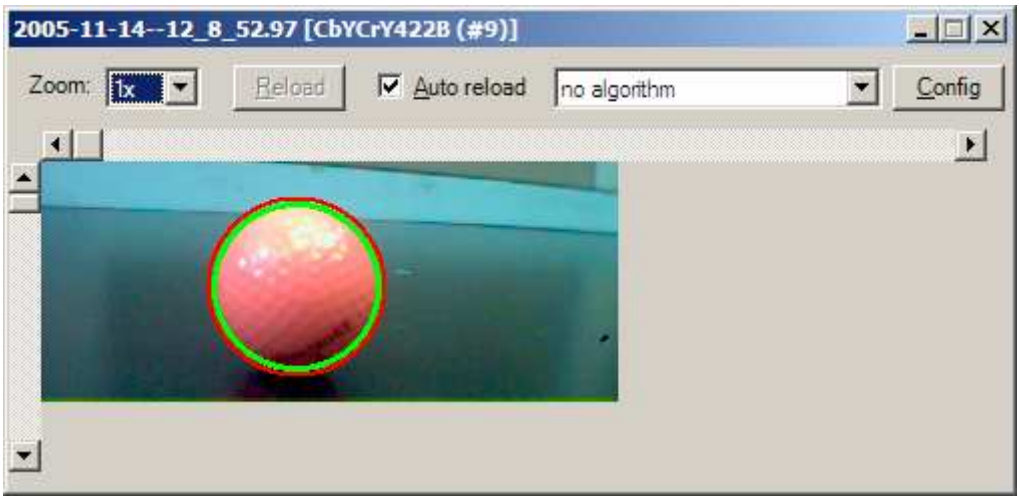


Abbildung VI.5-4: manuelle Separation

Um die Güte der Klassifikation abschätzen zu können, hat der Benutzer die Möglichkeit sich anzeigen zu lassen, welche Pixel zum Ball und welche nicht zum Ball gezählt werden. Dies wird über einen Algorithmus („show selected circle“) realisiert. Das Resultat ist in Abbildung VI.5-5 dargestellt.

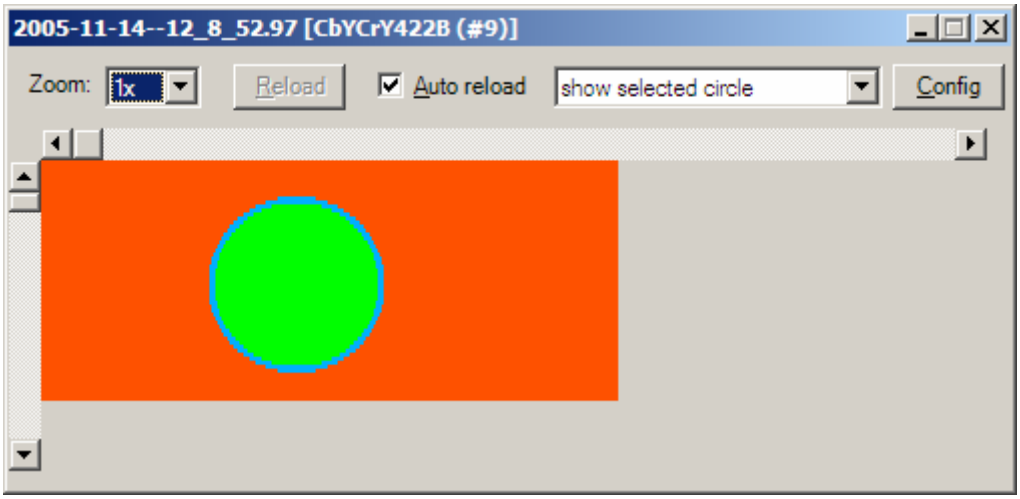


Abbildung VI.5-5: Darstellung der separarierten Daten

Um die Qualität der Klassifikation zu erhöhen, ist in den ImageViewer ein Toleranzwert eingebaut. Durch Unschärfe ist es mitunter sehr schwierig genau einzuschätzen, wo genau der Rand des Balls liegt. Um die Klassifikation zu vereinfachen, wird alles was im vom Benutzer markierten Kreis liegt als Ball erkannt und alles was außerhalb eines etwas größeren Kreises liegt als Hintergrund erkannt. Die Pixel, die außerhalb des kleineren aber noch innerhalb des größeren Kreises liegen, werden ignoriert. Dies ist in den oben stehenden Abbildungen an den grün- und rot-farbigen Kreisen sichtbar. In der Darstellung der separierten Daten sind die Ballpixel grün, die Nicht-Ballpixel rot und die ignorierten Pixel blau gefärbt.
Da es oft hilfreich ist, zu wissen welchen Farbwert ein bestimmter Pixel im Bild hat, ist eine Pipette eingebaut. Sobald sich der Mauszeiger über dem Bild befindet, wird in einem speziellen Fenster der Farbwert des Pixels unter dem Mauszeiger angezeigt. Das Format des Farbwertes ist vom benutzen Datenformat abhängig (z.B. YUV oder RGB). Die Pipette ist in Abbildung VI.5-6 dargestellt.

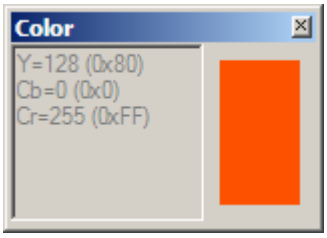


Abbildung VI.5-6: Pipette

Der ImageViewer unterstützt im aktuellen Zustand eine Reihe von Algorithmen (zusätzliche Algorithmen können leicht hinzugefügt werden, siehe Kapitel VI.5.1.4):

Grayscale:	Konvertiert das Bild in Graustufen. Dieser Algorithmus dient nur als einfaches Beispiel.
BallFinderAdv:	Ein Ballfindungs-Algorithmus. Siehe auch Kapitel 143. Dieser Algorithmus bietet auch die Möglichkeit, Konfigurationsdaten an den auf dem DSP laufenden Ballfindungsalgorithmus zu senden.
CalibrationTest:	Versucht eine automatische Kalibrierung der Farben bezüglich unterschiedlicher Lichtverhältnisse.
RGB332 Classifier:	Dient dazu im RGB332-Farbraum einzelne Farben manuell dem Ball zuzuordnen und das Ergebnis der damit vorgenommenen Klassifikation zu betrachten. Anschließend kann eine Liste der zum Ball gehörigen bzw. der nicht zum Ball gehörigen Farben ausgegeben werden.
Show selected circle:	Stellt wie oben beschrieben das Ergebnis der manuellen Separation dar (siehe Abbildung VI.5-5).
Separated data:	Wird im Zusammenhang mit Support Vector Machines benutzt. Die manuell separierten Daten können in einem Datenformat abgespeichert werden, dass von der SVM verstanden wird.

VI.5.1.3 Interne Struktur

In diesem Abschnitt wird knapp und vereinfacht auf die interne Struktur des ImageViewers eingegangen, da dies das Verständnis des nächsten Abschnitts über Erweiterungsmöglichkeiten erleichtert.
Der ImageViewer basiert auf einer Pipes&Filters-Architektur. Dabei werden die Daten wie in einer Pipeline durch verschiedene Verarbeitungsstufen geführt. Den Anfang macht dabei die Quelle (*Source*). Dies kann entweder eine Dateiquelle (*FileSource*) oder eine Quelle sein, die aus der seriellen Schnittstelle liest (*SerialSource*). Die hier gewonnen Daten kommen unverändert von der Kamera. Diese Daten werden nun an einen Algorithmus (*Algorithm*) weitergeschickt. Dieser Algorithmus kann z.B. versuchen, im Bild einen Ball zu erkennen. Ein Algorithmus kann das empfangene Bild ändern, wenn dies gewünscht ist. So kann ein Ballfindungs-Algorithmus z.B. markieren, wo ein Ball gefunden wurde. Die veränderten Daten sendet der Algorithmus dann weiter an einen Datenformat-Konvertierer (*DataFormat*). Die Aufgabe dieses Konvertierers ist es, aus dem Format der Kamera in das interne Format des ImageViewers zu konvertieren. Dieses interne Format ist ein RGB-Format, in dem alle Farbkomponenten 8 Bit Genauigkeit haben. Nach der erfolgten Konvertierung wird das Bild weiter an die GUI geschickt, die es nun auf dem Bildschirm darstellt.
Aus welchen einzelnen Komponenten sich die Pipeline zusammensetzt, wird vom Benutzer über die GUI bestimmt (indem er die gewünschte Quelle, das erwartete Datenformat und einen Algorithmus auswählt).

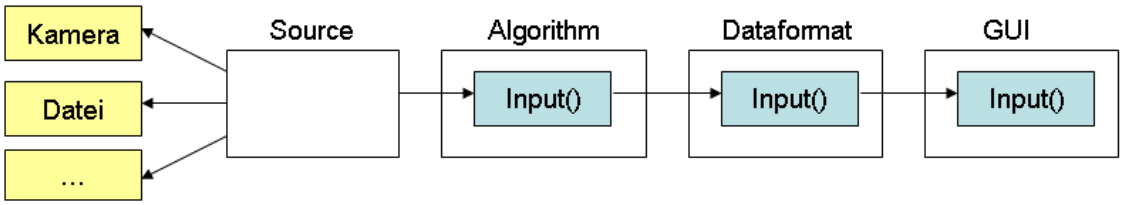


Abbildung VI.5-7: Pipeline des ImageViewers

VI.5.1.4 Erweiterungsmöglichkeiten

Durch die im vorherigen Kapitel beschriebene Pipes&Filters-Architektur, ist es sehr einfach einzelne Komponenten der Verarbeitungspipeline auszutauschen oder zu erweitern.
Um eine neue Datenquelle zu erstellen, muss von *DataSource* abgeleitet und folgende Methoden implementiert werden:

```
public abstract void Reload();
public abstract void Start();
public abstract void Stop();
```

Reload wird dabei aufgerufen, wenn neue Daten aktiv von der GUI angefordert werden. Die gewonnen Daten müssen dann weiter an die nächste Stufe der Pipeline geschickt werden. Dies geschieht über den Aufruf *Target.Input()*. *Start* und *Stop* werden aufgerufen, bevor das erste Mal Daten angefordert werden bzw. nachdem keine Daten mehr benötigt werden. Hier wird z.B. die serielle Schnittstelle geöffnet bzw. geschlossen. Unabhängig von *Reload* kann eine Quelle selbstständig durch einen Aufruf von *Target.Input()* damit beginnen Daten durch die Pipeline zu schicken. Die FileSource kann als einfaches Beispiel benutzt werden.
Um einen neuen Algorithmus zu erstellen, muss von *Algorithm* abgeleitet und folgende Methoden implementiert werden:

```
public abstract string Name { get; }
public abstract void Input(PipelineData Data);
public virtual void Configure();
```

Name wird dabei genutzt, um einen Namen für den Algorithmus in der GUI anzeigen zu können. *Input* wird von der unterliegenden Schicht (*Source*) aufgerufen und bekommt die gewonnen Daten übergeben. *Input* arbeitet dann auf den Daten und gibt sie an die höherliegende Schicht (*DataFormat*) über *Target.Input()* weiter. *Configure* wird von der GUI bei einem Klick auf den Config-Button

aufgerufen und sollte dazu verwendet werden einen Konfigurationsdialog anzuzeigen. Als einfaches Beispiel kann der GrayAlgorithm verwendet werden. RGB332Classifier ist ein relativ einfacher Algorithmus, der einen Konfigurationsdialog verwendet.
Um ein neues Datenformat zu erstellen, muss von *Dataformat* abgeleitet und folgende Methoden implementiert werden:

```
public abstract int Framesize { get; }
public abstract void Input(PipelineData Data);
```

Framesize berechnet dabei aus Höhe und Breite der Bilddaten die für einen Frame benötigten Bytes.
Input wird von der unterliegenden Schicht (*Algorithm*) aufgerufen und bekommt die vom Algorithmus bearbeiteten Daten übergeben. Die übergebenen Daten werden dann von Input ins von der GUI erwartete Datenformat (RGB) konvertiert und über *Target.Input()* an die GUI weitergereicht.
Um eine für den Benutzer ansprechende Darstellung in der GUI gewährleisten zu können, muss für jedes Datenformat noch eine DataformatDescription erstellt werden. Dazu muss von *DataformatDescription* abgeleitet und folgende Methoden implementiert werden:

```
public abstract string Name { get; }
public abstract Dataformat CreateDataformat();
```

Name gibt dabei den Namen des Datenformats zurück, wie er in der GUI dargestellt werden soll. *CreateDataformat* erzeugt eine neue Instanz des zugehörigen Dataformats.
Einfache Beispiele für *Dataformats* bzw. *DataformatDescriptions* sind DataformatRGB888 und DataformatRGB332.

VI.5.1.5 Probleme

Ein Problem bei der Benutzung des ImageViewers liegt in der langsamen Übertragung der Bilddaten über die serielle Schnittstelle. Die Übertragung eines Bildes dauert dabei einige Sekunden, eine Live-Übertragung ist daher nicht möglich. Dieses Problem lässt sich leider auch nicht beheben (außer durch eine andere Art der Übertragung, die über eine höhere Bandbreite verfügt).

VI.5.2 ColorSpaceExplorer

Der ColorSpaceExplorer visualisiert mit dem ImageViewer aufgenommenen Kamerabilder im dreidimensionalen (Y,Cb,Cr) - Farbraum. Für jedes im Bild vorkommende Pixel (Y,Cb,Cr) wird ein Punkt in der entsprechenden Farbe an die Stelle (Y,Cb,Cr) im Farbraum gemalt.
Über die Schaltfläche „Bilder hinzufügen“ lassen sich ein oder mehrere Bilder so laden. In Abbildung VI.5-8 wurden mehrere Bilder auf denen Bälle zu sehen waren eingelesen. Man sieht deutlich viele

rote Pixel, die wohl zum Ball gehören, daneben auch viele hellere und blaue Pixel, die dem Hintergrund zuzuordnen sind.

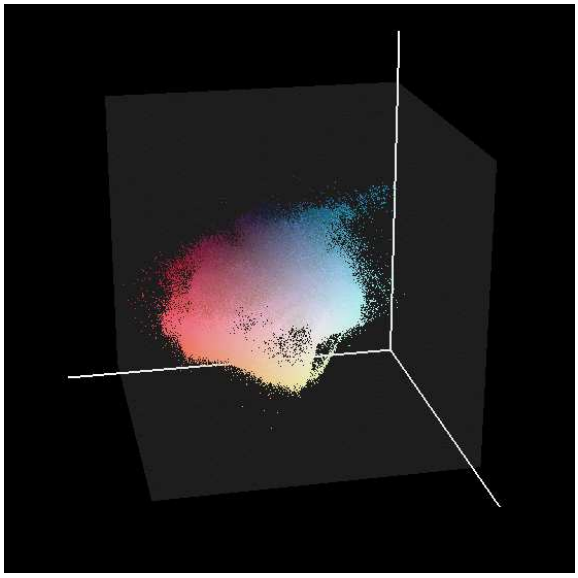


Abbildung VI.5-8: Visualisierung von mehreren Bildern mit Ball

Mit dem ColorSpaceExplorer ist es auch möglich, eine „Vorschau“ zu generieren, wie bei einer entsprechenden Einstellung die Funktion CheckPixel Ballpixel klassifizieren würde. Über zwei Reiter lassen sich die Parameter für lineare Trennebenen und Limit – Klassifikation einstellen.
In der Abbildung ist links eine Trennung des gesamten Farbraums durch die lineare Trennebene (-1,-1,-1,256), recht die Trennung des Farbraums in den Limits [50,150],[80,200],[0,200] zu sehen.

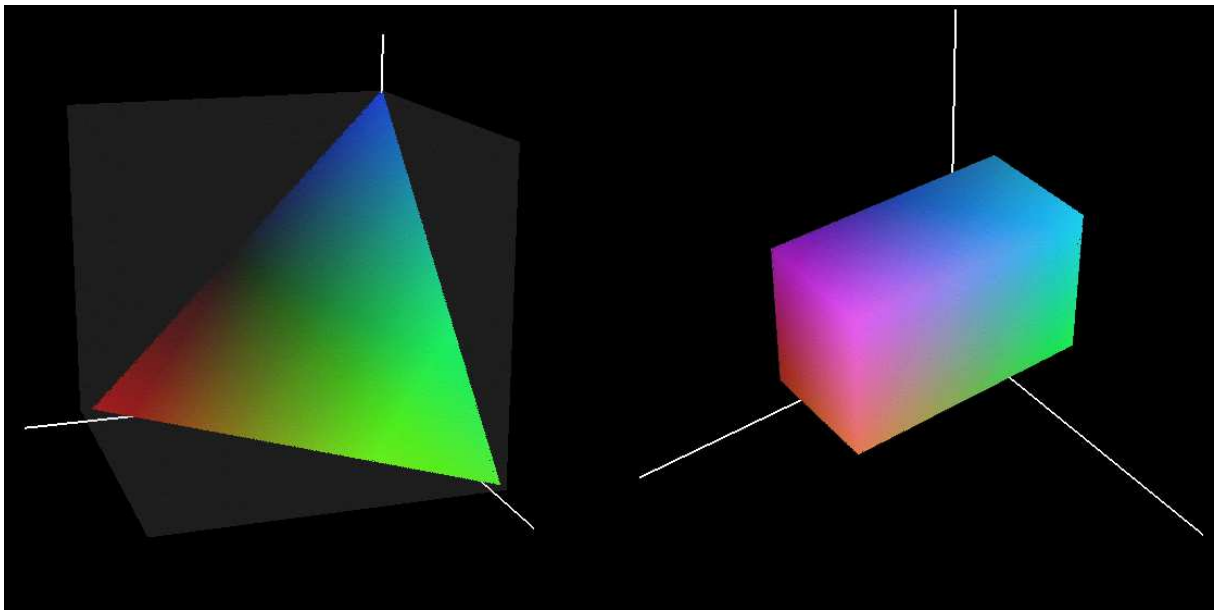


Abbildung VI.5-9: Lineare und Limit Klassifikation

Durch diese Möglichkeit kann man leicht anhand von realen Bilddaten an Parametern für eine gute Klassifikation experimentieren.

In der folgenden Abbildung sind vier unterschiedliche Trennebenen eingezeichnet. Links oben geschieht noch keine Trennung, die Pixel stammen aus verschiedenen Bildern mit Ball. Rechts oben wird der Farbraum durch die Trennebene (0,-11,14,0) getrennt. Hier werden noch viele Hintergrundpixel als Ballpixel klassifiziert.
Links unten wurde der Offset ein wenig verschoben, die Trennebene ist nun (0,-11,14,-400). Diese Trennung scheint schon gut zu funktionieren, einige dunklere Hintergrundpixel werden noch als Ballpixel erkannt. Rechts unten wurde der Offset noch etwas weiter verschoben, die Trennebene ist (0,-11,14,-600). Hier wird zwar kein Hintergrundpixel noch als Ballpixel klassifiziert, dafür werden aber viele der helleren Ballpixel nicht mehr als Ball erkannt.
Wie eine optimale Trennebene gefunden wurde, ist in Kapitel VI.4.1.6.2 beschrieben.

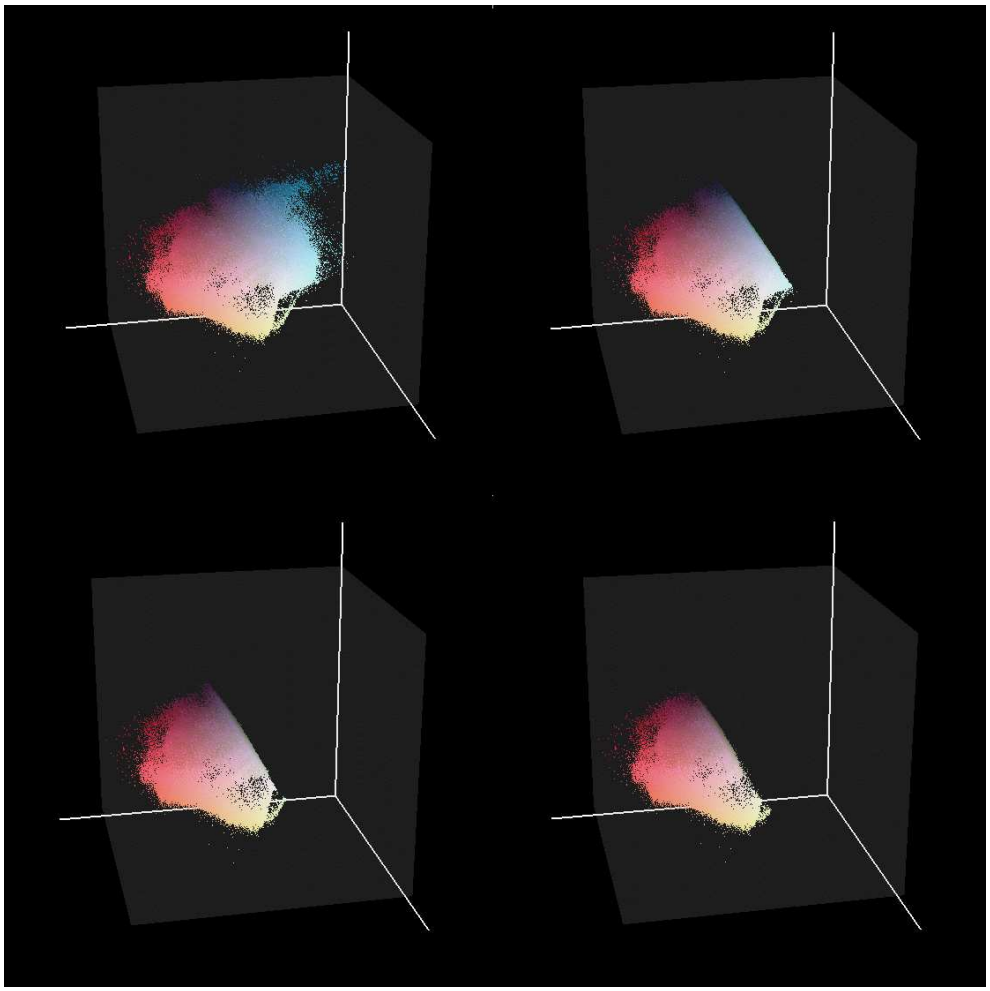


Abbildung VI.5-10: Bilddaten mit verschiedenen Trennebenen

Zusätzlich zu den Pixeldaten kann der ColorSpaceExplorer auch vom ImageViewer klassifizierte Daten (Endungen .ball und .noball) anzeigen. In der unteren Abbildung ist die Klassifizierung des Trainingsdatensets aus dem Vergleichs-Kapitel abgebildet. Rote Pixel beschreiben Ballpixel, blaue Pixel stehen für Hintergrundpixel.

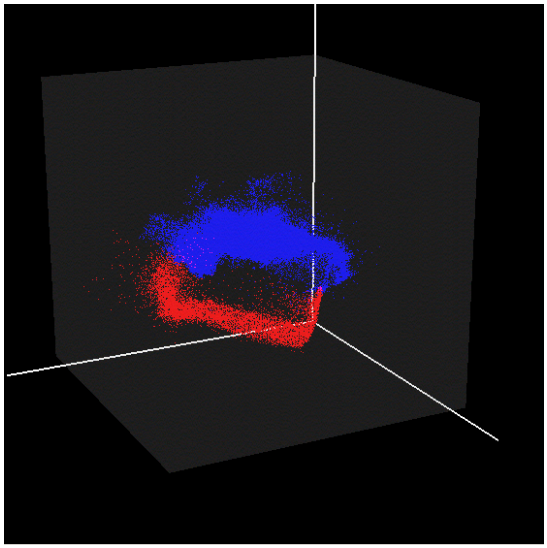


Abbildung VI.5-11: Klassifizierte Bildinformationen

VI.5.3 AlgoAnalyzer

VI.5.3.1 Anforderungen

Der AlgoAnalyzer soll dafür genutzt werden, die verschiedenen Ballfindungsalgorithmen zu vergleichen. Dabei wird vom AlgoAnalyzer nur der Klassifikationsteil der Ballfindungsalgorithmen betrachtet. Jeder Klassifikationsalgorithmus wird dabei auf eine Reihe von Testbildern angewendet. Diese Testbilder wurden vorher von Hand in Ball- und Nicht-Ball-Pixel separiert (mit Hilfe des im Kapitel VI.5.1 beschriebenen ImageViewers). Abweichungen des Klassifikationsalgorithmus von der korrekten Klassifizierung können so bestimmt werden und aus den daraus gewonnen Daten kann eine Fehlerquote errechnet werden.

VI.5.3.2 Features

Der AlgoAnalyzer wurde in der .NET Programmiersprache C# geschrieben.
Als Parameter bekommt der AlgoAnalyzer eine Konfigurationsdatei und eine beliebige Anzahl von Testbildern. Diese Testbilder können von zwei Typen sein. Der eine Typ enthält Daten über Ballpixel und der andere Typ Daten über Nichtballpixel. Unterschieden werden die Typen durch die Dateiendung, *.ball enthält Ballpixel und *.noball Nichtballpixel. Dabei wird als Datenformat nur das CbYCrY422B-Format unterstützt. Der AlgoAnalyzer wendet nun jeden ihm bekannten Algorithmus auf die Testdaten an und vergleicht die Ausgabe des Algorithmus mit dem korrekten Wert.
Die Konfigurationsdatei hat folgendes Format:

```
limits_active=1
limits_ymin=0
limits_ymax=255
limits_cbmin=63
```

```
limits_cbmax=103
limits_crmin=0
limits_crmax=255

linear_active=1
linear_ycoeff=0
linear_cbcoeff=-14
linear_crcoeff=11
linear_offset=100

dist_active=1
dist_ycenter=175
dist_cbcenter=85
dist_crcenter=180
dist_distance=5000
```

Dabei kann jeder Klassifizierer einzeln aktiviert und deaktiviert werden. Außerdem können die Parameter für die Klassifizierer gesetzt werden.

Die Ausgabe des AlgoAnalyzer umfasst die Anzahl der korrekt und inkorrekt klassifizierten Ball- bzw. Nichtball-Pixel, sowohl absolut als auch prozentual.

```
                                    output CheckPixel
                  ¦ +                               ¦ -
                --+-------------------------------+-------------------------
correct         + ¦ true pos  (BallCorrect)       ¦ false pos (BallIncorrect)
classification  - ¦ false neg (NoBallIncorrect)   ¦ true neg  (NoBallCorrect)

     +     -                    +     -   Limits
+ 3359  5114              + 39.64 60.36   0<=Y<=255 63<=Cb<=103 0<=Cr<=255
-   10 78056              -  0.01 99.99   Correct classified: 94.08

     +     -                    +     -   Linear
+ 8070   403              + 95.24  4.76   Y:0 Cb:-14 Cr:11 Off:100
- 5289 72777              -  6.78 93.22   Correct classified: 93.42

     +     -                    +     -   Distance
+ 5875  2598              + 69.34 30.66   Y:175 Cb:85 Cr:180 Dist:5000
- 3985 74081              -  5.10 94.90   Correct classified: 92.39
```

Abbildung VI.5-12: Ausgabe AlgoAnalyzer

VI.5.3.3 Erweiterungsmöglichkeiten

Um einen neuen Klassifizierungsalgorithmus hinzuzufügen, muss von der abstrakten Klasse *Classifier* abgeleitet und folgende Methoden implementiert werden.

```
protected abstract bool CheckPixel(int Y, int Cb, int Cr);
protected abstract string Name { get; }
protected abstract string Settings { get; }
```

CheckPixel stellt dabei den eigentlichen Klassifizierungsalgorithmus dar. Die Methode bekommt die Werte eines Pixels übergeben und entscheidet, ob dieser Pixel ein Ballpixel (Rückgabewert *true*) oder ein Nichtballpixel (Rückgabewert *false*) ist. *Name* liefert eine kurze Beschreibung des Algorithmus für die Ausgabe der Resultate. *Settings* liefert eine Zusammenfassung der Parameter für den Algorithmus, welche auch zur Ausgabe der Resulate benutzt wird.
Zusätzlich müssen die vom neuen Klassifizierungsalgorithmus benötigten Parameter in die Konfigurationsdatei eingetragen und von dort eingelesen werden.

VII. Strategie

Kristijan Pulina,
Daniel Mikus,
Christian Tesch

VII.1 Aufbau der Strategie

Die Strategie erhält aus der Verarbeitung des Deckenkamerabildes Informationen darüber, wo auf dem Spielfeld (x,y), mit welcher Ausrichtung und mit welcher Geschwindigkeit sich die Spielfeldobjekte wie Ball, Gegner- und Freundroboter bewegen. Dabei liegt das Koordinatensystem so auf dem Spielfeld, dass der Nullpunkt sich oben links auf Höhe der oberen langen Bande befindet (y=0) und der Wert x=0 links auf der Innenseite des eigenen Torraums erreicht wird. Der x-Wert steigt nach rechts hin bis zum gegnerischen Tor, und der y-Wert steigt nach unten hin an. Die Winkel werden im Bogenmaß angegeben, so dass der Winkel von 0 Grad eine Ausrichtung auf die gegnerische Torlinie bedeutet, $\pi/2$ eine Ausrichtung nach oben, π bedeutet nach links auf unser Tor und $3/2\pi$ nach unten. Die Strategie lässt sich in drei Ebenen unterteilen: Auf der Spielzugebene wird entschieden, welcher Spielzug gerade durchgeführt werden soll (Raumangriff / Raumdeckung). Abhängig von diesem Spielzug wird jedem Roboter auf der Rollenebene eine Rolle (Torwart / AngriffMitBall / ...) zugewiesen, und aufgrund eines situationsabhängigen Bewertungsfaktors wird dann für jeden Roboter entschieden, welche Handlung, die dieser Rolle zur Verfügung stehen, ausgewählt wird. Im folgenden Diagramm sind alle Spielzüge, Rollen und Handlungen aufgeführt, die im robosoccer Projekt im Juli 2005 eingebunden waren:

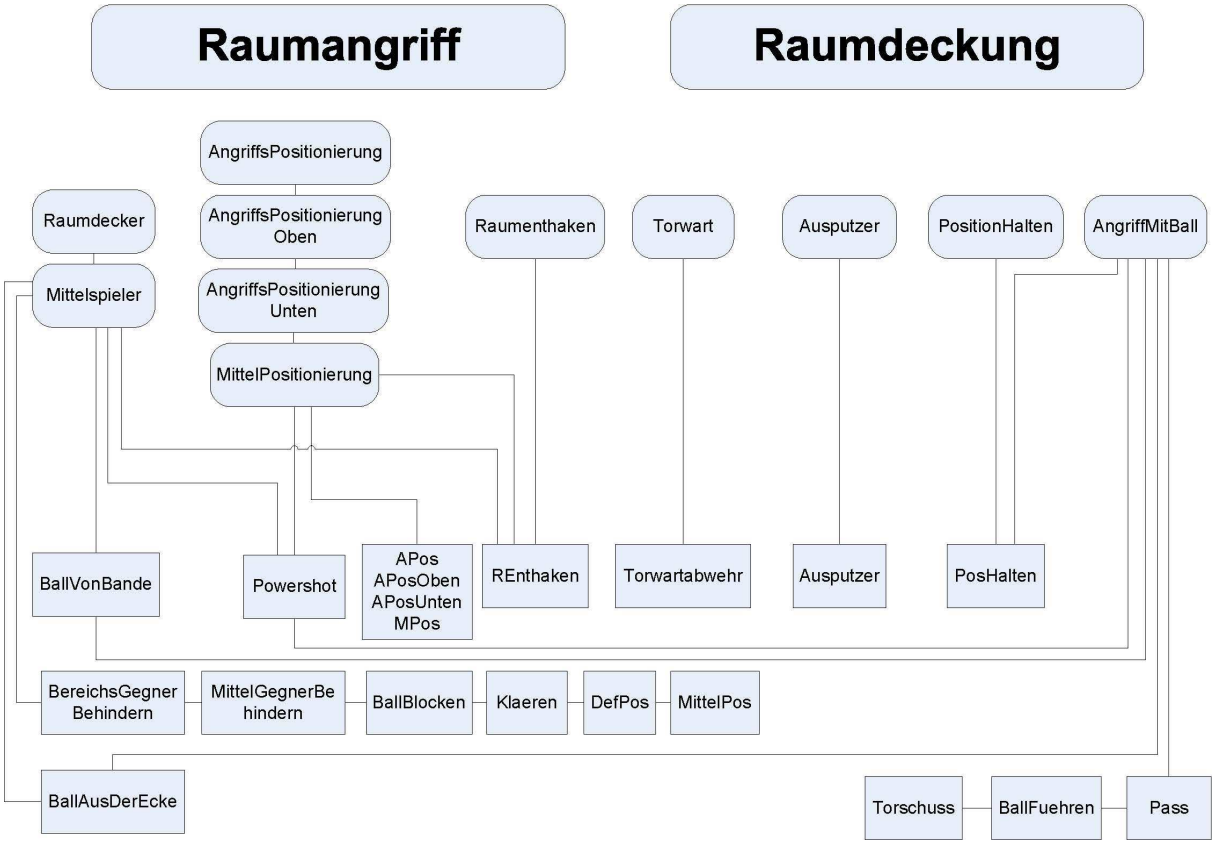


Abbildung VII.1-1: Spielzüge, Rollen und Handlungen

Auf der Handlungsebene (Torschuss / BallFuehren / BallVonBande / …) wird schließlich aus der Istsituation des Roboters (Position, Geschwindigkeit) und der Istsituation des Balles eine Sollsituation für den Roboter ermittelt (wohin der Roboter fahren soll und welche Geschwindigkeit und Ausrichtung er dort haben soll), die dann an den Anfahrtsalgorithmus weitergegeben wird, damit der Roboter sein Ziel anfährt und erreicht. An den Anfahrtsalgorithmus wird immer nur die Endgeschwindigkeit und Position übergeben. Leider kann man daher die Anfahrtsgeschwindigkeit (außer im Modus „manuell“) nicht festlegen.

Es soll nun am Beispiel der einfachen Handlung *StrafraumPos* gezeigt werden, wie eine Handlung aufgebaut ist:

```
void StrafraumPos::berechnenWennAktiv() {
```

Die Methode *berechnenWennAktiv()* wird immer wieder aufgerufen, solange einem Roboter diese Handlung zugewiesen wird. Sie bestimmt die Sollsituation *ss* aus der Istsituation *is*.

```
const IstSituation&  is = *ist;  SollSituation& ss = *soll;
const IstSpielfeldobjekt& ichAlt = is.wir[kennung];
SollSpielfeldobjekt& ichNeu = ss.wir[kennung];
```

Der Ausdruck *wir[kennung]* bezieht sich auf den aktiven Roboter, dessen Istposition nun in *ichAlt* und dessen Sollposition in *ichNeu* abgelegt werden.
Die für die Sollposition benötigten Variablen (*x,y* = Position, *v* = Geschwindigkeit, *w* = Winkel) sollten nun deklariert und initialisiert werden:

```
float sollxRob = (konstante.StrafraumRechts+konstante.TorraumRechts)/2;
float sollyRob = konstante.TorraumOben;
int sollvRob = 0;
float sollwRob = 0;
```

Hier soll sich der Roboter oben im gegnerischen Strafraum neben dem gegnerischen Tor positionieren. Als nächstes wird bestimmt, wo der Ball im nächsten Augenblick sein wird:

```
IstSpielfeldobjekt ballInZukunft = is.ball;
Vektor2D ball = ballInZukunft.pos;
```

Auf diesen zukünftigen virtuellen Ball beziehen sich alle Berechnungen, weil die von der Kamera aufgenommene Ballposition zu diesem Zeitpunkt schon „veraltet“ ist. Nun folgt der logische Teil: Ist der Ball in der unteren Spielfeldhälfte, soll der Roboter nach oben, sonst nach unten:

```
if (ball.y>(konstante.FeldMitteY)) sollyRob = konstante.TorraumOben;
if (ball.y<(konstante.FeldMitteY)) sollyRob = konstante.TorraumUnten;
```

Der Roboter soll nach unten schauen, wenn der Ball unten ist (analog oben):

```
if (ball.y>(konstante.FeldMitteY)) sollwRob = 3*M_PI/2; // Ball unten
if (ball.y<(konstante.FeldMitteY)) sollwRob = M_PI/2;  // Ball oben
```

Entsprechend wird der Sollwinkel gesetzt. Diese Rob-Variablen dienten allerdings nur zur Berechnung. Nun muss die Sollsituation tatsächlich gesetzt werden und der Anfahrtsalgorithmus aufgerufen werden. Dies geschieht über die Variable *ichNeu*:

```
ichNeu.pos.x = sollxRob;     ichNeu.pos.y = sollyRob;
ichNeu.ab = sollwRob;        ichNeu.v = sollvRob;
ichNeu.modus = konstante.Modus_Delta;
ichNeu.kollisionsvermeidung = false;
ichNeu.anfahrtsalgo = konstante.Default_Algo;
```

Die Konstanten zur Spielfeldgröße und den Raumlinienpositionen für das jeweilige Spielfeld sind in der Datei *Konstanten.cpp* gespeichert. Sie befindet sich im Ordner *Strategie*. Um diese Werte zu

benutzen, müssen vor das Programm noch folgende Inklusionen gesetzt werden, damit alles funktioniert:

```
#include "StrafraumPos.h3"
#include <exception>
#include <math.h>
#include "../../Konstanten.h2"
#include "../../RohrTypen/IstSpielfeldobjekt.h2"
#include "../../Bibliotheken/Zukunftsberechnung.h2"
#include "../../Bibliotheken/Strafraum.h2"
#include "../../Bibliotheken/BallZielKorrektur.h2"
```

Dies ist im wesentlichen der Aufbau einer Handlung. Bei größeren Handlungen kann der logische Teil sehr komplex bis unüberschaubar werden.

VII.2 Idee des Versenkers

VII.2.1 Historie

Aufgrund der Tatsache, dass ein gutes Fußballroboterspiel mit erfolgreichem Spielausgang auch bei sehr guter Hardware, guten Anfahrts- und Fahreigenschaften der Roboter und guter Bildverarbeitung des Hostsystems im Endeffekt nur durch ein strategisch gutes Verhalten der Roboter gegeben ist und schon viele Spiele unseres Teams unter anderem an diesem gescheitert sind, war es an der Zeit, die vorhandene Strategie zu überdenken und -bearbeiten.
Dabei fiel auf, dass es bisher nur zwei aktive Angriffshandlungen gibt, die für einen Torschuss vorgesehen sind: „Torschuss“ und „Powershot“. Dabei wurde sogar die Handlung „Powershot“ aufgrund mangelnder Effizienz in letzter Zeit aus dem System genommen. So bleibt im Prinzip nur eine einzige Handlung, von der ein möglicher Spielgewinn abhängig ist.
Da aber bei vielen Spielsituationen und bei guter gegnerischer Abwehr kein Torschuss mit gleichnamiger Handlung möglich ist, entschieden wir uns, die Erfolgschancen dadurch zu erhöhen, indem wir eine neuartige alternative Angriffshandlung implementieren, die sich jedoch grundlegend von dieser - meist für den Gegner leicht vorhersehbaren - Handlung abhebt.

VII.2.2 Das Prinzip Vorleger-Versenker

Die Idee besteht nun darin, nicht einen einzigen Roboter die komplette Arbeit von Ballanfahren, über Ballführen bis hin zum Torschuss übernehmen zu lassen, sondern mindestens zwei Roboter in diese Situation mit einzubinden.
So existiert ein Vorleger, der den Part des Ballanfahrens und Passens übernimmt, sowie ein Versenker, der zum richtigen Zeitpunkt den finalen Torschuss mit hoher Trefferquote ausübt und genau darauf

feinjustiert ist. Da der Versenker nur eine geradlinige Schussbahn aufs Tor abfährt, lassen sich alle bisherigen Probleme der kreisförmigen Ballanfahrt mit mangelnder Zukunftsberechnung effektiv umgehen.

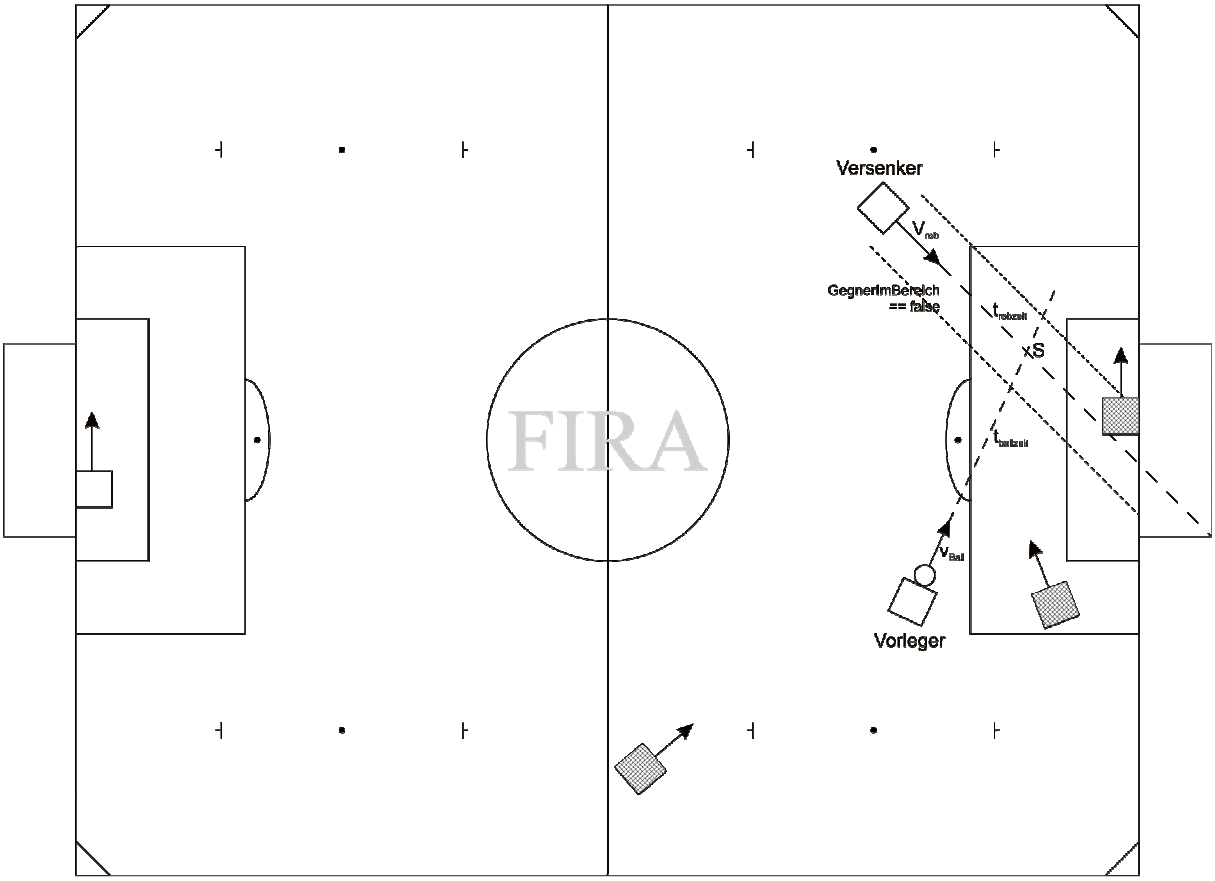


Abbildung VII.2-1: Das Prinzip Vorleger-Versenker

Eine mögliche konkrete Angriffssituation sieht dann so aus, dass der Vorleger den Ball an einer Position in der rechten Spielfeldhälfte abfängt, während der Versenker sich schon gegenüberliegend in der linken Spielfeldhälfte vor dem Strafraum mit Blickrichtung zur rechten Torecke positioniert. Der Vorleger führt nun den Ball über die Spielfeldmitte in Richtung linker Torecke, so dass der gegnerische Torwart sich vor genau dieser positionieren muss, um ein triviales Tor zu verhindern. Kurz nach Überquerung der Spielfeldmitte ändert der Vorleger seinen Richtungsvektor um 180 Grad, während für den Versenker 30 mal in der Sekunde der optimale Anfahrtszeitpunkt berechnet wird, um den Schnittpunkt S (Ball-Versenker) im genau richtigen Moment mit der richtigen Geschwindigkeit v_{Rob} zu erreichen. Ist der Anfahrtszeitpunkt mit Berücksichtigung der physikalischen Latenzzeiten erreicht, kickt der Versenker den Ball mit v_{max} in die rechte Torecke. Während der Anfahrt zum Ball wird kontinuierlich die Geschwindigkeit des Versenkers angepasst (vorzugsweise beschleunigt, auch über die Geschwindigkeit v_{max} hinaus), so dass der Roboter den Ball möglichst genau treffen und ablenken kann.

Diese Situation soll natürlich von jeder Seite oder auch mittig ausführbar sein. Da das gegnerische Team jeweils mit der anderen Torecke rechnet und dessen Abwehr entgegengesetzt gerichtet ist, verspricht das Prinzip Vorleger-Versenker hohe Trefferquoten.
Zu beachten wäre allerdings, dass ein gegnerischer Roboter den noch passiv wartenden Versenker bedrängen oder blockieren kann. Daher muss eine geeignete Gegnerberücksichtigung implementiert werden, um immer eine freie Schussbahn zu garantieren.

VII.2.3 Die drei Phasen des Versenkers

Damit der Versenker einen möglichst genauen Winkel in die Torecke anfährt und möglichst genau den Ball trifft, darf der Versenker nicht konstant in Bewegung sein. Er muss sich vielmehr zuerst ohne große Abweichung im richtigen Winkel positionieren, dort den richtigen Moment abwarten, um in seinem freien Schusskanal geradlinig auf die Geschwindigkeit v_{max} zu beschleunigen, den Ball zu treffen und abzubremsen.
Um dieses Problem anzugehen, wurde die Versenkerhandlung in drei Phasen unterteilt:

- Phase1:
  Der Versenker positioniert sich auf der dem Ball gegenüberliegenden Spielfeldhälfte. Sollte sich in einem bestimmten Bereich (dem Schusskanal) zwischen der Zielposition des Versenkers (VS) und des gegnerischen Tores außer dem gegnerischen Torwart noch ein weiterer feindlicher Roboter befinden, so wird die Zielposition gegebenenfalls verschoben, bis sich ein freier Schusskanal ergibt. Während dieser Phase kann der Vorleger (VL) damit beginnen, den Ball unter Kontrolle zu bringen.

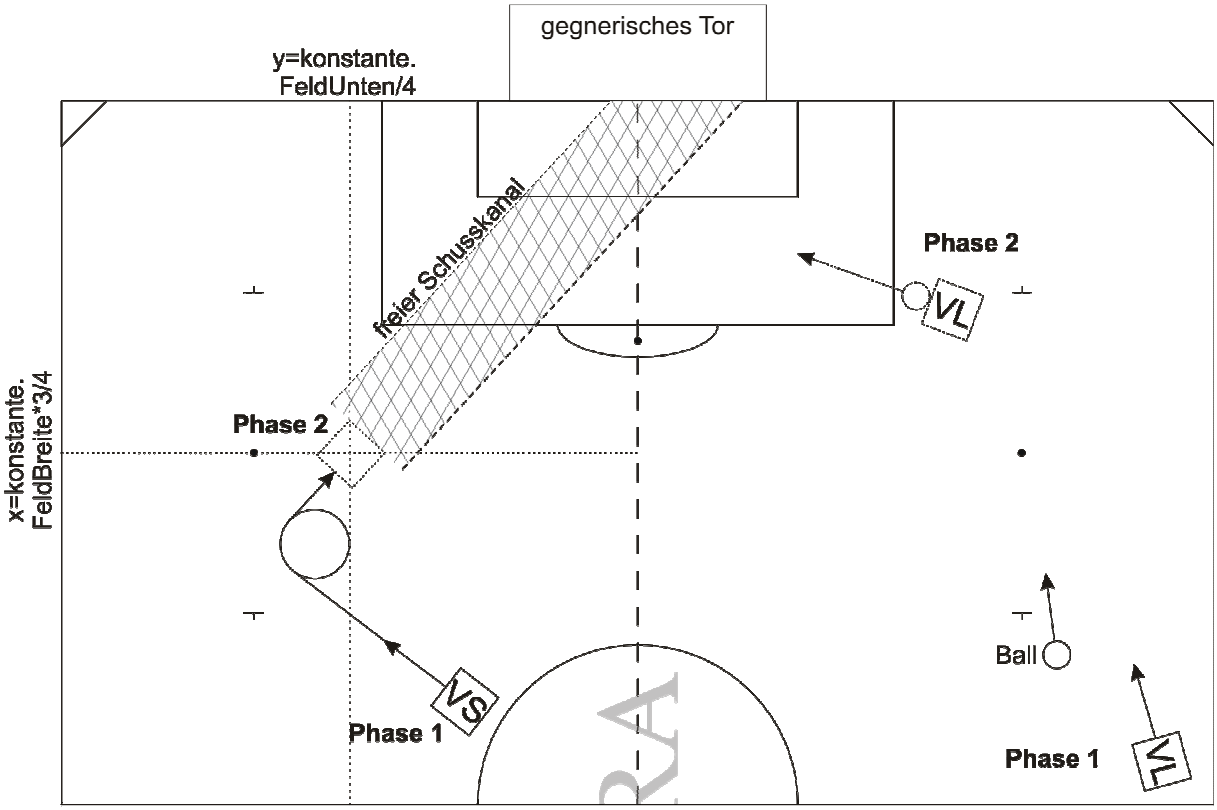


Abbildung VII.2-2: Phase 1 und 2 des Vorlegers und Versenkers

- Phase2:

  Der Versenker wartet nun auf seiner Position mit v=0 und berechnet anhand der momentanen Ballgeschwindigkeit und dessen Richtungsvektors den optimalen Anfahrtszeitpunkt, während der Vorleger die Vorlage gibt. Ist der richtige Anfahrtszeitpunkt erreicht, so wird Phase 3 eingeleitet. Sollte sich allerdings in dieser Wartephase ein Gegner in den Schusskanal bewegen und dort verharren, so wird erneut Phase 1 aufgerufen und die Angriffsposition geändert.

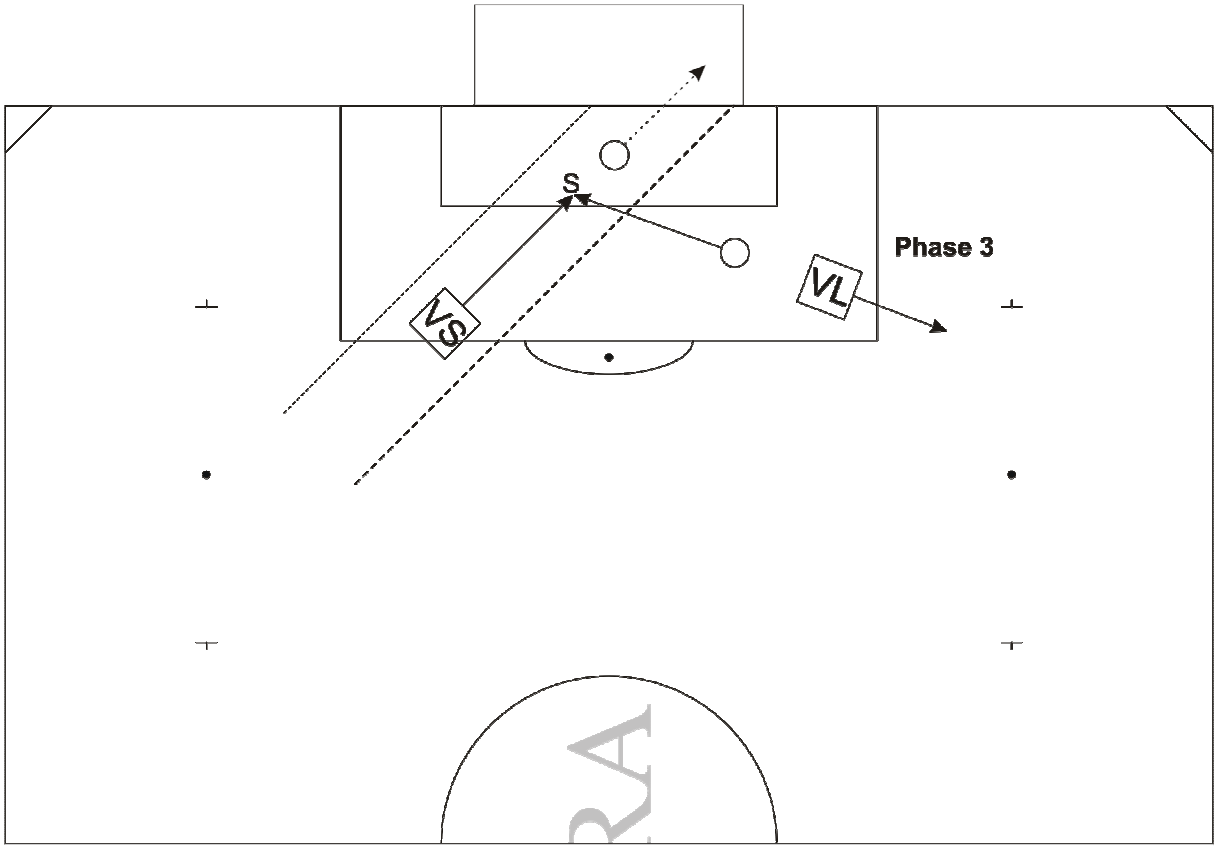


Abbildung VII.2-3: Phase 3 des Vorlegers und Versenkers

- Phase3:

  Die eigentliche Versenkerphase beginnt. Der Versenker beschleunigt nun auf die Geschwindigkeit vmax und korrigiert noch während der Fahrt ständig seine Geschwindigkeit, um den Ball im richtigen Moment zu treffen. Sollte der Versenker zu langsam sein, so wird gegebenenfalls auch über vmax hinaus beschleunigt. Hat der Versenker den Ball getroffen und fährt mit seiner hohen Geschwindigkeit in den Strafraum des Gegners ein, so tritt ein automatisches Bremssystem in Kraft, um ein mögliches Umstürzen des Roboters bei hoher Geschwindigkeit an einer Bande zu verhindern.

VII.2.4 Versenker auf Rollenbasis

Die neue Handlung „*Versenker*“ mit ihren drei Phasen wurde im weiteren Verlauf der PG durch drei separate Handlungen „*APos*“ (Phase 1: Angriffspositionierung), „*PosHalten*“ (Phase 2: Position halten) und „*VSVersenken*“ (Versenkerphase 3: Versenken) ersetzt, die von einer übergeordneten Rolle „*VersenkerAPos*“ ausgewählt werden. Außerdem wird die bereits implementierte Methode „*REnthaken*“ genutzt, falls sich der Versenker in der Positionierungsphase mit einem anderen Roboter verhakt.

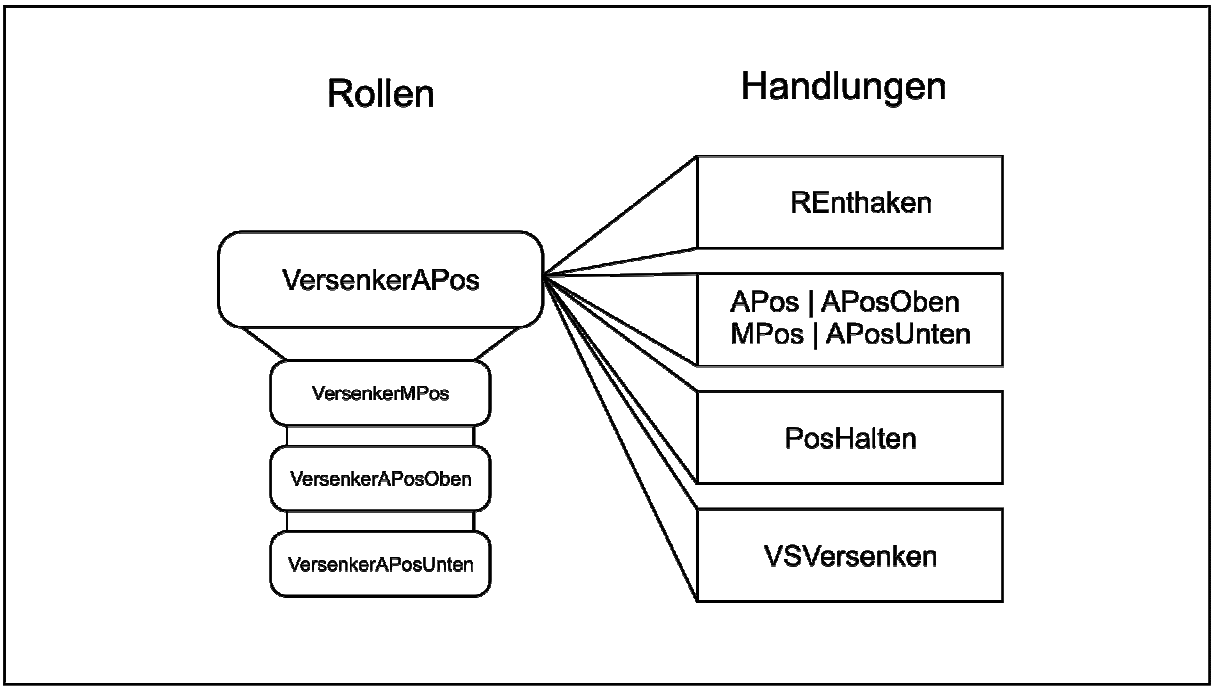


Abbildung VII.2-4: Versenker auf Rollenbasis

Durch drei weitere Rollen „*VersenkerMPos*“, „*VersenkerAPosOben*“ und „*VersenkerAPosUnten*“ wird somit ermöglicht, mehr als einen Versenker gleichzeitig einzusetzen, der zu unterschiedlichen Zeiten und Positionen agiert.

VII.3 Hinzugefügte Rollen

VII.3.1 VersenkerAPos

Die Rolle *VersenkerAPos* nutzt die vier Handlungen *REnthaken*, *APosOben*, *PosHalten* und *VSVersenken*. Sie wird dabei in drei Handlungsphasen unterteilt, die nacheinander abgearbeitet werden.

Die Rolle beginnt in Handlungsphase 1, welche noch recht trivial, überschaubar und hier auszugsweise dargestellt ist:

```
if (Handlungsphase==1) {
	if (VerhaktMitGegner(kennung)) {
		*gewaehlteHandlung = &typeid(Handlungen::REnthaken);
	} else {
		if (strategieinfo.APosPositioniert) {
			Handlungsphase=2;
		} else {
			*gewaehlteHandlung = &typeid(Handlungen::APos);
```

```
                }
        }
}
```

Es wird zuerst auf ein Verhakungsproblem überprüft und gegebenenfalls die Handlung *REnthaken* zugewiesen. Sollte kein solches Problem bestehen, oder bereits behoben sein, so wird mit

```
*gewaehlteHandlung = &typeid(Handlungen::APos);
```

die Positionierungshandlung *APos* eingeleitet. Diese Handlung setzt in der Strategieinfo die Variable *APosPositioniert* auf „true“, falls die Zielposition des Versenkers erreicht ist. Anhand dieser Variable kann nun entschieden werden, wann Handlungsphase 2 (Wartephase) eingeleitet wird.
Befindet sich die Rolle nach erfolgreicher Positionierung des Roboters nun in Handlungsphase 2, wird anhand

```
*gewaehlteHandlung = &typeid(Handlungen::PosHalten);
```

dem Roboter die Handlung *PosHalten* zugewiesen. In diesem Wartezustand finden nun bei jedem Aufruf der Rolle (in der Regel 30 mal in der Sekunde) diverse Rechnungen statt.
Eine der wichtigsten Berechnungen ist sicher die, den Schnittpunkt S des Ballrichtungsvektors mit dem Vektor des noch wartenden Versenkers zu bestimmen. Dazu wenden wir die Schulmathematik an und berechnen aus den beiden Vektoren zwei Geraden in der allgemeinen linearen Form t=mx+b, aus denen sich dann der Schnittpunkt ermitteln lässt:

```
point2d schnittpunkt;
       float m1 = (float)(ball.v.y / ball.v.x);
       float t1 = -m1 * ball.pos.x + ball.pos.y;
       float m2 = -sin(ichAlt.ab) / cos(ichAlt.ab); //Roboter
       float t2 = -m2 * ichAlt.pos.x + ichAlt.pos.y;
       schnittpunkt.x = (t1-t2)/(m2-m1);
       schnittpunkt.y = m1 * schnittpunkt.x + t1;
```

t1 entspricht dabei der Geraden des Ballvektors, *t2* der des Versenkers. *m1* und *m2* sind die dazugehörigen Steigungen. Die „Steigung“ des Balles (*m1*) wird anhand der Ballgeschwindigkeit in x- und y-Richtung (*ball.v.x* und *ball.v.y*) berechnet. Da die „Steigung“ des Roboters (*m2*) nicht anhand seines Geschwindigkeitsvektors berechnet werden kann (er ist schließlich in der Wartephase), muss hier auf den Ausrichtungswinkel zurückgegriffen werden. *ichAlt.ab* entspricht dabei dem absoluten Bogenmaß der alten Position (also der vor genau einem Frame). Wir müssen hier mit *–sin(ichAlt.ab)* rechnen, dass der Ursprung des Koordinatensystems in der linken oberen Ecke des Spielfeldes liegt und somit die y-Koordinate nach unten zeigt.

Aus den beiden berechneten Geraden ergibt sich nun der Schnittpunkt S mit den Werten *schnittpunkt.x* und *schnittpunkt.y*.
Viel wichtiger als die Schnittpunksberechnung ist allerdings die Ermittlung der Zeit, wie lange der Ball und wie lange der der Roboter (sollte er in diesem Moment losfahren) benötigt, um diesen Schnittpunkt zu erreichen. Die Zeit des Balles zum Schnittpunkt wird mit folgendem Code berechnet:

```
float ballzeit = sqrt(pow((schnittpunkt.x-
       ball.pos.x),2)+pow((schnittpunkt.y-ball.pos.y),2)) / ball.vabs;
```

Hierzu wird der Abstand des Balles zum Schnittpunkt durch die aktuelle Ballgeschwindigkeit dividiert. Die Zeit des Roboters berechnet sich ebenfalls aus seinem Abstand zu S und der gegebenen Angriffsgeschwindigkeit (hier v_{max}).

```
float AbstandRoboterSchnitt = sqrt(pow(schnittpunkt.x-
       ichAlt.pos.x,2)+pow(schnittpunkt.y-ichAlt.pos.y,2))-
       konstante.RoboterRadius;

float robzeit = (AbstandRoboterSchnitt / AngriffsGeschwindigkeit);
```

In der vereinfachten Fassung des Versenkers genügt es nun, beide Ankunftszeiten zu vergleichen. Sollte *robzeit* den gleichen Wert wie *ballzeit* besitzen, so wird Phase 3 eingeleitet, in der der Versenker mit v_{max} geradlinig in Richtung Schnittpunkt S fährt und genau dort den Ball (zumindest in der Theorie) trifft. Da in der Praxis allerdings der Roboter keine unendlich hohe Beschleunigung besitzt und der Ball auch nicht konstant seinen Pfad abrollt, sondern bedingt durch Reibungskräfte auch seine Geschwindigkeit verringert, bedarf es noch einigen Erweiterungen am Quellcode. Außerdem müssen noch zahlreiche weitere Situationen beachtet werden, so kann sich z.B. plötzlich ein Gegner im Schussbereich aufhalten, der Versenker kann bedrängt, verschoben, oder seine Ausrichtung durch äußere Einflüsse verändert werden, und natürlich kann auch der Ball abgelenkt werden, so dass sein Schnittpunkt mit dem des Roboters eventuell schon außerhalb des Spielfeldes liegt.
Betrachten wir als erstes eine Spielsituation ohne Gegner und Torwart, welche nur aus dem bereits positionierten Versenker und dem rollenden Ball besteht. Hier tritt dann nur das Problem der Latenz-, Beschleunigungszeiten und der Reibungskräfte auf.
Die Latenzzeiten vernachlässigen wir im ersten Schritt und setzen die float-Variable *latenzzeit* auf 0. Da die Zeiten ballzeit und robzeit im float-Zahlenbereich in der Praxis niemals den gleichen Wert erreichen werden, wird einfach abgefragt, ob die Differenz beider Zeiten unter einem bestimmten Wert liegt, der hier gleichzeitig als *Beschleunigungsausgleich* bezeichnet ist.

```
float Beschleunigungsausgleich=0.19;
```

```
/* kleineren Wert eintragen, falls Roboter zu früh den Schnittpunkt S
       erreicht. */

if (fabs(robzeit + latenzzeit - ballzeit) < Beschleunigungsausgleich) {
       if (BallRichtungRoboter) {
              Handlungsphase=3;
       }
}
```

Der *Beschleunigungsausgleich* muss bei neuerer Roboterhardware mit besserer Beschleunigung eventuell angepasst werden. *BallRichtungRoboter* gibt aus, ob der Ball sich in Richtung Roboter bewegt. Diese Abfrage ist sehr wichtig, sonst kommt es vor, dass der Versenker losfährt, während der Ball sich auf seiner Geraden auch nur minimal in die andere Richtung bewegt (das kann auch durch Ungenauigkeiten in der Kamera- und Bildverarbeitungseinstellung hervorgerufen werden). Die Berechnung von *BallRichtungRoboter* wird hier aus Platzgründen nicht weiter betrachtet.
Stellt man nun v_{max} auf einen kleinen Wert ein (unter 600 mm/s), und lässt den Ball langsam zwischen Tor und Versenker rollen, so trifft der Roboter den Ball schon recht genau und erzielt auch schon einige Treffer. Da die Rolle allerdings nur die Aufgabe hat, den Anfahrtszeitpunkt zu berechnen, liegt es mehr oder weniger auch an der Handlung *Versenker*, dass der Ball zur richtigen Zeit getroffen wird (Stichwort Geschwindigkeitskorrektur bei variabler Roboter- und Ballgeschwindigkeit).
Es gilt nun den Anfahrtszeitpunk weiter zu optimieren.
Bei höheren Spielgeschwindigkeiten ($v_{max} > 750$ mm/S) spielt die Latenzzeit eine große Rolle. So gehen bei der Berechnung des Bildes in der BV, bei der Strategieauswertung mit anschließendem Senden der Funkbefehle und Umsetzung in mechanische Kraft wertvolle Millisekunden verloren. Außerdem darf folgendes nicht vernachlässigt werden: Je höher die Endgeschwindigkeit v_{max} des Roboters ist, desto länger dauert es, bis sie erreicht wird. Daher soll die zu berücksichtigende Latenzzeit abhängig von der Endgeschwindigkeit v_{max} sein:

```
float latenzzeit = 0;
       if (konstante.VMax >= 750) {
              latenzzeit=(float)konstante.VMax/4800;
       } else latenzzeit=0;
```

Mit diesem Ansatz der Latenzzeitberücksichtigung lassen sich schon viel größere Erfolge erzielen. Bei geeigneter Geschwindigkeitskorrektur des Versenkers und dessen Implementierung (siehe Handlung *Versenker*) kann man schon fast von einer hundertprozentigen Trefferquote sprechen (vorausgesetzt es befinden sich natürlich keine Gegner auf dem Spielfeld und man führt den Ball mit einer Geschwindigkeit im Bereich des physikalisch Möglichen in Richtung Schnittpunkt, der nicht zu dicht am Roboter liegt, so dass dieser auch mit v_{max} erreicht werden kann).

Falls sich der Ball nun zu steil in Richtung gegnerisches Tor bewegt, kann es vorkommen, dass der Schnittpunkt S außerhalb des Spielfeldes liegt. In diesem Fall braucht die Versenkerphase nicht eingeleitet zu werden. Dieses trifft auch zu, falls der Schnittpunkt hinter dem Versenker liegen sollte. Daher wird die Berechnung der Ankunftszeiten und der eventuelle Aufruf von Handlungsphase 3 von einer weiteren Schnittpunktpositionsabfrage umrahmt:

```
if ((schnittpunkt.x - ichAlt.pos.x < konstante.TorLinieRechts -
       ichAlt.pos.x) && (schnittpunkt.x - ichAlt.pos.x > 0)) { /*CODE*/ }
```

Befindet sich nach erfolgreichem Versenken der Ball hinter der Torlinie, bekommt der Roboter erneut die Handlung *APos* zugewiesen. Dieses ist in einem Meisterschaftsspiel allerdings eher uninteressant und dient nur zu Testzwecken:

```
if (is.ball.pos.x > konstante.TorLinieRechts)||(GegnerImBereich) {
       *gewaehlteHandlung = &typeid(Handlungen::APos);
       Handlungsphase=1;
       strategieinfo.APosPositioniert=false;
}
```

Wie man schon erkennt, wird *APos* auch dann erneut zugewiesen, falls die Variable *GegnerImBereich* erfüllt ist. Um festzustellen, ob ein Gegner sich in diesem Bereich, dem Schusskanal, befindet, hatten wir anfangs eine relativ rechen- und zeitaufwändige Rechnung entworfen, in der wir jeden vorhandenen Roboter auf dem Spielfeld damit verglichen, ob er sich genau zwischen den beiden Aussenkanten des Schusskanals, sowie zwischen Tor und Versenker befindet.
Da diese Berechnung allerdings sehr uneffektiv war und man auch einen leichten Anstieg der Strategiezeit erkennen konnte (besonders bei vielen Robotern), haben wir uns entschlossen, nach anderen Möglichkeiten zu suchen. Unsere Berechnung soll deswegen hier auch nicht weiter detailliert beschrieben werden.
Die einfachste Möglichkeit bestand nun darin, auf die vorhandene Methode *EpsilonTest* der Bibliothek *BereichsTest* zurückzugreifen. Dieser Methode wird ein Bereich und eine Roboterkennung *i* übergeben, woraufhin sie zurückgibt, ob dieser Roboter mit der Kennung *i* in dem definierten Bereich zu finden ist.

```
EpsilonTest(XStart, YStart, XEnd, YEnd, StartEps, EndEps,
       is.die[i].pos.x, is.die[i].pos.y)
```

Dieser Test wird in einer for-Schleife für jeden der gegnerischen Roboter sowie für jeden der eigenen durchgeführt. Ausgeschlossen werden müssen der eigene Versenker sowie der gegnerische Torwart. Damit nicht sofort eine Positionsänderung des Versenkers erfolgt, wenn ein Roboter schnell durch den

Schusskanal fährt, wurde ein Buffer von 15 Frames eingebaut. Erst wenn 15 Bilder lang der Epsilontest „wahr“ ausgibt, wird *GegnerImBereich* auf „true“ gesetzt.
Der Buffer befindet sich in der Strategieinfo und wird dort entsprechend iteriert beziehungsweise wieder auf „0“ gesetzt.
Nach dieser ausführlichen Beschreibung von Handlungsphase 2 findet die Handlungsphase 3 nur kurze Erwähnung. Hier passiert auch nichts weiteres, als dass dem Roboter die Handlung Versenker zugewiesen wird und in der Strategieinfo der Versenker auf aktiv gesetzt wird:

```
if (Handlungsphase==3) {
	if(!strategieinfo.VersenkerAktiv){
		Handlungsphase=1;
	}
	if (is.ball.pos.x > konstante.TorLinieRechts) {
		*gewaehlteHandlung = &typeid(Handlungen::APos);
		strategieinfo.VersenkerAktiv = false;
		Handlungsphase=1;
	} else {
		strategieinfo.VersenkerAktiv = true;
		*gewaehlteHandlung = &typeid(Handlungen::VSVersenken);
	}
}
```

Ergänzend ist noch zu sagen, dass wir kurz vor der Fahrt zur Weltmeisterschaft in Singapur mit den neuen Robotern und einem anderem Anfahrtsalgorithmus vor dem Problem standen, dass sich die Roboter nur mit größerer Abweichung auf der Zielposition positioniert haben und praktisch *APosPositioniert* nur in seltenen Fällen auf *true* gesetzt wurde. Daher haben wir uns entschlossen, die Positionierungsabfrage in Phase 1 auszukommentieren und direkt in Handlungsphase 2 zu springen. Anstatt PosHalten wird dann kontinuierlich APos aufgerufen, was bei Erreichen der Position keinen großen Unterschied zu PosHalten macht. Daher wird *PosHalten* in der aktuellen Fassung des Versenkers nicht mehr benutzt.

VII.3.2 VersenkerAPosOben

Die Rolle *VersenkerAPosOben* nutzt die Handlungen *REnthaken*, *APosOben*, *PosHalten* und *VSVersenken*. Sie entspricht der Rolle *VersenkerAPos*, jedoch wird hier anstatt *APos* die Handlung *APosOben* benutzt, so dass durch diese Rolle festgelegt werden kann, dass der Versenker immer die Position der oberen Spielfeldhälfte einnimmt.
Genau wie in *VersenkerAPos* wird *PosHalten* in der aktuellen Fassung des Versenkers nicht mehr genutzt (siehe oben).

VII.3.3 VersenkerAPosUnten

Die Rolle *VersenkerAPosOben* nutzt die Handlungen *REnthaken*, *APosUnten*, *PosHalten* und *VSVersenken.* Auch diese Rolle entspricht der Rolle *VersenkerAPos*, allerdings greift der Versenker immer von der unteren Spielfeldhälfte an.
Auch hier wird *PosHalten* nicht mehr genutzt.

VII.3.4 VersenkerMPos

Zuletzt existiert noch die Rolle *VersenkerMPos*. Sie nutzt die Handlungen *REnthaken*, *MPos*, *PosHalten* und *VSVersenken.* Wie schon bei den anderen beiden Rollen entspricht sie der Rolle *VersenkerAPos*, allerdings greift der Versenker hier immer von der Mitte aus an.
PosHalten wird nicht mehr genutzt.

VII.4 Hinzugefügte Handlungen

Wie in der Übersicht der Strategie zu erkennen ist, wählt jede aktive Rolle eine aktive Handlung, in der die eigentlichen Befehle für den aktuellen Roboter vergeben werden. Zur Umsetzung der Idee des Versenkers wurde die vorhandene Strategie um die Handlung Versenker erweitert. Außerdem wurden die bereits vorhandenen Handlungen *APos*, *APosUnten*, *APosOben* und *MPos* dahin gehend geändert, dass sie als geeignete Startposition für den implementierten Versenker dienen.

VII.4.1 APos

Die bereits vorhandene Handlung *APos*, die vorher lediglich zur Positionierung eines Angreifers benutzt wurde, ist nun so verändert worden, dass sich der aktuelle Roboter so positioniert, dass er eine optimale Versenkerposition einnimmt. Dabei orientiert sich der Angreifer immer in die jeweils gegenüberliegende Längsspielfeldhälfte im Verhältnis zur Ballposition. Die genaue Positionierung erfolgt dann analog zu den Handlungen *APosUnten* bzw. *APosOben*.

VII.4.2 APosUnten

Die Handlung APosUnten wird zur Positionierung des Versenkers in der linken Spielfeldlängshälfte benutzt. Dabei wählt sie als Ausgangsposition die in der Konstanten-Bibliothek vorgegebene Position „*FeldUnten – APosY*“ als y-Koordinate und „*FeldUnten – APosX*“ als x-Koordinate.
Die Handlung APosUnten wird solange in Phase 1 wiederholt aufgerufen, bis eine gute Versenkerposition ohne Gegner im Schusskanal erreicht und der Versenker in Richtung Tor ausgerichtet ist.

VII.4.3 APosOben

APosOben unterscheidet sich von APosUnten nur in der Positionierung des Versenkers. Er nimmt als Ausgangsposition „*APosY*“. Mit diesen beiden Handlungen sollte ein Versenker vor dem gegnerischen Strafraum positioniert werden, wobei diejenige der beiden Handlungen ausgewählt werden sollte, die den Versenker in der jeweils entgegengesetzten Spielfeldlängshälfte zum Vorleger positioniert.

VII.4.4 MPos

Die Handlung MPos positioniert den Versenker mittig vor dem gegnerischen Strafraum und verhält sich analog zu den beiden Handlungen APosUnten bzw. APosOben.

VII.4.5 Versenker

Hier ist der Versenker bereits in seiner 3. Phase. Er hat den Befehl bekommen, auf den Schnittpunkt mit dem Ball zu fahren. Während der Fahrt überprüft er zusätzlich, ob der berechnete Schnittpunkt noch stimmt, da durch kleine Abweichungen in der Bildverarbeitung, durch Ungenauigkeiten des Spielfeldes oder weitere Einflussfaktoren die Ausrichtung und Position des Balles oder des eigenen Versenkers schwanken können. Hierbei wird die vorher erwähnte Schnittpunktberechnung in jedem Frame neu durchgeführt. Der Versenker reagiert auf diese Abweichungen, indem er seine Geschwindigkeit anpasst, um den Ball im richtigen Moment zu treffen:

```
if ((robzeit - ballzeit) > -0.05) {
       AngriffsGeschwindigkeit +=25;
} else {
       if (AngriffsGeschwindigkeit>200)
              AngriffsGeschwindigkeit -=15;
       }
}
```

Die genauere Beschreibung der Berechnung von *robzeit* und *ballzeit* befindet sich im Kapitel VII.3.1: VersenkerAPos.

Eine weitere Funktion, die die Trefferwahrscheinlichkeit erhöht, ist die Beschleunigung am Ball. Da der Versenker aufgrund Geschwindigkeitsanpassung mit einer niedrigen Geschwindigkeit auf den Ball treffen könnte, würde der Ball nicht ausreichend stark angestoßen werden. Sobald der Versenker den Schnittpunkt mit dem Ball erreicht hat und der Ball knapp vor dem Versenker rollt, wird die Geschwindigkeit um 75mm/s in jedem Frame erhöht.

```
// Falls Ball getroffen wurde, Roboter beschleunigen, damit Fuehrung
// verbessert wird...
       if ((fabs(ichAlt.ab - ball.ab))<0.225) {
```

```
            AngriffsGeschwindigkeit+=75;
      }
```

Um zu verhindern, dass der Versenker unnötiger Weise seine Fahrt fortsetzt, obwohl er den Ball nicht treffen kann, wurden einige Abbruchbedingungen eingebaut, die den Versenker stoppen, um u.a. Beschädigungen an den Robotern zu vermeiden:
Falls der Schnittpunkt von Ball und Versenker entweder hinter der gegnerischen Torlinie oder im Rücken des Versenkers liegt, wird die Radgeschwindigkeit auf 0 gesetzt und der Status des Versenkers auf inaktiv gesetzt, so dass die übergeordnete Rolle diesem Roboter eine neue Aufgabe zuweisen kann. Zusätzlich wird die Versenkerhandlung abgebrochen, wenn der Versenker in den gegnerischen Torraum hinein gefahren ist oder der Ball die Torlinie überschritten hat und somit ein Tor erzielt wurde:

```
if ((schnittpunkt.x > konstante.TorLinieRechts)
      || (schnittpunkt.x - ichAlt.pos.x < 0)
      || ((ichAlt.pos.x > konstante.TorraumRechts)
      || (ball.pos.x > konstante.TorLinieRechts)))
 {
      ichNeu.modus = (konstante.Modus_Drehkick);
      ichNeu.radgeschwL = 0;
      ichNeu.radgeschwR = 0;
      strategieinfo.VersenkerAktiv = false;
      strategieinfo.VersenkerObenAktiv = false;
      strategieinfo.VersenkerUntenAktiv = false;
 }
```

VIII. Fehlersuche

Kristijan Pulina

VIII.1 Fehlersuche

Es gibt kaum einen anderen Bereich als die Strategiekomponente eines komplexen Programms, bei dem die Welten von Theorie und Praxis so offensichtlich miteinander kollidieren. Meist funktioniert eine Roboterhandlung in der Praxis anders als von der Programmierung her zu erwarten wäre. Das kann an falschen Einstellungen in der Bildverarbeitung liegen oder an Hardwaremängeln. Aber meistens muss man in einem solchen Fall das Programm solange ändern, bis es in der Praxis richtig funktioniert. Dann kommt es aber automatisch wieder zu Problemen, wenn sich die Hardware ändert (z.B. durch neue Roboter), weil alle Programmteile neu angepasst werden müssen. Dieses Kapitel soll erleuchten, welche Probleme und Fehler am häufigsten auftreten, wo die Ursachen für seltsames Verhalten der Roboter liegen könnten und welche Folgen Programmiermängel haben können. Letztere können sogar zum Programmabsturz führen. Wenn das Betriebssystem, die angeschlossenen Geräte und das Programm sich nicht miteinander vertragen, hilft meistens nur noch ein Neustart.

VIII.1.1 Hardwaremängel

Ein Grundproblem ist, dass von der Softwareseite alle Roboter gleich behandelt werden, obwohl sie hardwaretechnisch zwar gleich gebaut sind, sich in der Praxis aber unterschiedlich verhalten. Abgesehen von baulichen Mängeln, kann das an der falschen Einstellung der Motorregler liegen. Aber auch so banale Ursachen wie Verschmutzung oder Abnutzung führen zu Unregelmäßigkeiten. Die Leistung eines Roboters hängt meistens vom Ladezustand des Akkus ab. Probleme gibt es, wenn dieser falsch sitzt, nur schwach oder zu stark geladen ist oder wenn aus anderen Gründen der Schwerpunkt des Roboters nicht mehr in der Mitte liegt. Oft kommt es auch zu Problemen durch das Schleifen des Roboters über den Boden, oder die Räder schleifen am Roboter, weil irgendwas dazwischen sitzt. Die Bereifung sollte daher regelmäßig kontrolliert werden. Außerdem sollte das Spielfeld sauber gehalten werden. Unabhängig von den einzelnen Robotern, bekommt die Strategie große Probleme, wenn kurzzeitig keine oder zufällige Werte von der Kamera übermittelt werden. Ursachen hierfür können sein, dass das Kamerakabel zu lang ist oder nicht richtig steckt. Aber manchmal liefert die Bildverarbeitung auch von sich aus falsche Werte, weil sie z.B. den Ball nicht findet. Ein weiteres Problem stellt der Funk dar, also die Datenübertragung zu den Robotern. Dazu muss man wissen, dass die Strategie ständig jedem Roboter Befehle sendet, wie er fahren soll. Bloß manchmal bekommt der Roboter keine oder falsche Befehle. Das liegt dann an Funklöchern oder Signalstörungen. Das Funkmodul sollte dann probeweise umpositioniert werden. Die richtige Einstellung der Deckenkamera und der Beleuchtung für ein gutes Bild sind sehr wichtig für die

Bildverarbeitung und damit auch für die Strategie. Problematisch sind auch Hindernisse wie Banden oder Gegnerroboter, wenn der Roboter versucht seine Sollposition anzufahren. Meist fährt er dann einfach weiter und blockiert. Für den Fall gibt es die Rolle *Enthaken*.

VIII.1.2 Softwaremängel

Ein Grundproblem in der Programmierung ist meistens die Grundeinstellung des Programmierers, der davon ausgeht, dass alles richtig funktioniert und keine falschen oder unsinnige Eingaben erfolgen. Das gilt auch für unser Programm, wobei es drei Arten von Eingaben gibt: Die automatischen von der Bildverarbeitung, die programminterne Variabelenbelegung und die Eingaben des Benutzers aus der GUI. Es sollte auf jeden Fall immer überprüft werden, ob diese Werte gesetzt sind und sich im richtigen Wertebereich befinden. Variablen sollten grundsätzlich sofort bei der Deklaration auch mit sinnvollen Werten belegt werden, damit später im Programm keine Überraschungen auftreten. Es kommt sehr häufig vor, dass mit uninitialisierten oder zufälligen Werten weitergerechnet wird, was dann natürlich zu unsinnigen Ergebnissen führt. Als Programmierer sollte man davon ausgehen, dass nicht alles immer so funktioniert wie es soll. Ein großes Problem stellen uninitialisierte Werte innerhalb von if oder case - Verschachtelungen dar. Diese können dazu führen, dass mit zufälligen Werten weitergerechnet wird und wenn dann auch noch Pointer benutzt werden, führt das schnell dazu, dass man auf unzulässige Speicherbereiche zugreift. Die Folge davon ist, dass das Betriebssystem interveniert und das Programm beendet wird (meist mit der Meldung Speicherzugriffsfehler). Innerhalb eines Spieles ist ein solcher Programmabsturz natürlich fatal. Daher wurde entschieden, das Speichertestprogramm Valgrind zu benutzen, um solche Fehler innerhalb unseres Programms ausfindig zu machen.

VIII.1.3 Speicherzugriffsfehler (Valgrind)

Valgrind ist ein Speichertestprogramm, unter dem man andere Programme laufen lassen kann, um Speicherzugriffsfehler zu finden. Es ergänzt das bestehende Programm um Analysecode und simuliert die Ausführung des Programms, die sich dadurch ca. um Faktor 30 verlangsamt. Das Haupttool ist *Memcheck*, welches die folgenden Probleme ausfindig machen kann: Benutzung von uninitialisiertem Speicher, das Lesen oder Schreiben auf befreite Speicherbereiche, das Überschreiten der Grenzen von einem allokiertem Speicherbereich, Schreib- oder Lesefehler beim Stackzugriff, Memorylecks durch verlorengegangene Pointer und das Überlappen von Quell- und Ziel-Pointern. Als Ergebnis liefert Valgrind Logfiles mit großen Fehlerlisten. Leider werden einige unerlaubte Speicherzugriffe, die Konflikte mit dem Betriebssystem verursachen, nicht erkannt und protokolliert, weil das Programm davor einfach extern vom Betriebssystem unterbrochen wird. Dies führt bei uns meistens zu der Meldung „Speicherzugriffsfehler“. Aber trotzdem liefern die Logfiles wertvolle Hinweise wie und wo Speicherplatz falsch benutzt wird.

Die in unserem robotsoccer-Programm auftretenden Fehler lassen sich abhängig von ihrer Ursache in 8 Klassen unterteilen:

QT-Fehler

Dies sind Fehler in der GUI-Programmierung mit *QT*, bei denen z.B. Textfeldgrößen undefiniert bleiben oder die Methode *qt_init* uninitialisierte Werte bekommt.

Kameraabfragefehler

Die häufigsten Fehler treten bei der Abfrage der Kameraparameter auf. Das Programm dc1394_control.c benutzt durch Kamerasignalstörungen uninitialisierte oder zufällige Werte. Probleme gibt es bei der Abfrage der Kameracontrolregister.

Libraryfehler

Die Librarys libraw1394,libX11,libqt und libICE machen Probleme beim Aufruf der Methoden raw1394_get_port_info,qt_init,QSessionmanager,_IceWrite,_XSend,Q_Label,qt_format_text,QTextLayout,QDialog,QTextEngine,libdc_startmain.

Input-Output-Fehler

Diese Fehler sind auf nicht behandelte Eingaben zurückzuführen, die behandelt werden müssten. Die Eingaben stammen meist aus der GUI vom Benutzer.

Terminatorfehler:

Die Methode *_dl_map_object* kann zum sofortigen Programmabsturz mit Speicherzugriffsfehler führen.

Schedulerfehler

Das Programm *vg_scheduler.c* verursacht Probleme bei der Threadverwaltung. Es kommt zu unerlaubten Lesezugriffen auf bereits beendeten Threads.

XIOfehler sind Lesefehler auf befreiten Speicherbereichen.

Strategie- und Weltmodellfehler

Es kommt zu Fehlern bei der Speicherplatzreservierung, die aus irgendwelchen Gründen nicht funktioniert. Vermutlich liegt es am Betriebssystem. Betroffen ist meist die Methode *Weltmodell:init()* die innerhalb der Mainmethode von robotsoccer aufgerufen wird. Die Methode *AusgabeAnschluss()* und die Handlung Klaeren verursachen unerlaubte Lesezugriffe innerhalb des *Weltmodell::startstrat()-Threads*.

Weil diese Fehler innerhalb von Schleifen immer wieder wiederholt auftreten, kommt es zu einem Aufblähen der Logfiles. Eine gekürzte Version mit den Fehlerklassen findet sich im Anhang. Abschließend lässt sich sagen, dass unser Programm voll von Speicherzugriffsfehlern ist. Es wurden zum Teil über 5000 Stück gezählt. Aber die Ursachen sind anhand der Logfiles leicht aufzuzählen: Es gibt Probleme in der Threadverwaltung, was zu unerlaubten Lesezugriffen führt. Dann gibt es Konflikte mit dem Betriebssystem, welche unser Programm abstürzen lassen. Es werden Werte von

Variablen einfach nicht gesetzt oder es wird darauf vertraut, dass die Kamera-Abfrage-Methoden sinnvolle Werte liefern, obwohl sie manchmal uninitialisiert sind. Die GUI-Programmierung weist Mängel auf und die Benutzereingaben aus der GUI werden manchmal wegen Timingproblemen erst gar nicht behandelt. Ein so komplexes Programm wie *robotsoccer* von Speicherzugriffsfehlern zu befreien, scheint aussichtslos. Aber zumindest könnte man darauf achten, die Werte von Variablen immer zu setzen, jede Variable auch zu benutzen und den Wertebereich und die Änderung der Werte daraufhin zu überprüfen, ob sie sinnvoll sind. Wenn dies nicht der Fall ist, dann ist es oft besser gar nichts zu tun, als mit uninitialisierten Werten weiterzurechnen und Programmfehler zu provozieren.

VIII.1.3.1 Anhang zu Valgrind

Als Anhang befinden sich hier die mit Valgrind untersuchten Speicherzugriffsfehlern innerhalb unseres *robotsoccer* Programms. Es wurden 8 Fehlerklassen ausfindig gemacht: QT-Fehler, Kameraabfragefehler, IOC-Warnings, Library-Fehler, Strategie-Fehler, Weltmodellfehler, Terminator- und xio-errors. Die QT-Fehler wurden in 3 weitere Klassen unterteilt. Es werden meistens uninitialisierte Werte übergeben, was zu Speicherzugriffsfehlern führt.

VIII.1.3.2 Fehlerklassen Kurzfassung

QT-size-Fehler:
QLabel::sizeForWidth(int) const (in /usr/lib/libqt-mt.so.3.3.4)
QLabel::minimumSizeHint()
qt_format_text (in /usr/lib/libqt-mt.so.3.3.4)
QTextLayout::beginLayout(QTextLayout::LayoutMode)
QTextEngine::shape(int) const (in /usr/lib/libqt-mt.so.3.3.4)
QTextLayout::currentItem()
QWidget::QWidget(QWidget*, char const*, unsigned)
QApplication::QApplication(int&, char**)
__libc_start_main

Kameraabfragefehler
Die Abfrage der Kamera-Parameter liefert keine oder uninitialisierte (zufällige) Werte....
Es folgen die Methodennamen mit Zeilennummern, wo die Fehler auftraten:
Format: Methode (Programmname : Zeilennummer)
GetCameraROMValue (dc1394_control.c:488)
dc1394_get_camera_info (dc1394_control.c:1184,1192,1202,1221,1229,1239,...)
dc1394_get_camera_nodes (dc1394_control.c:795)
dc1394_print_camera_info (dc1394_control.c:1092)

SetCameraControlRegister (dc1394_control.c:617)
dc1394_get_camera_feature_set (dc1394_control.c:1314)
GetCameraControlRegister (dc1394_control.c:539)
SetCameraControlRegister (dc1394_control.c:602)
dc1394_dma_setup_capture (dc1394_capture.c:540)
dc1394_stop_iso_transmission (dc1394_control.c:1909)
Das Programm dc1394_control.c verursachte hier Speicherzugriffsfehler.

QT-INIT-Fehler
XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal (in /usr/lib/libqt-mt.so.3.3.4)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QSessionManager::QSessionManager(QApplication*, QString&, Qstring&)

Library-Fehler
betroffene Methoden:
raw1394_get_port_info,qt_init,QSessionmanager,_IceWrite,_XSend,Q_Label,
qt_format_text,QTextLayout,QDialog,QTextEngine,libdc_startmain
betoffene Librarys: libraw1394,libX11,libqt und libICE

IOC-Warnings (Input-Output-Control, nicht behandelte Eingaben)
Warning: noted but unhandled ioctl 0x5415 with no size/direction hints
This could cause spurious value errors to appear.
See README_MISSING_SYSCALL_OR_IOCTL for guidance on writing a proper wrapper.

Qdialog,QWidget,QFontEngine,QApplication:exec
enthalten unadressierbare oder uninitialisierte Werte.

Terminator
_dl_map_object führt zum sofortigen Programmabsturz mit Speicherzugriffsfehler.

Strategie-Fehler (betroffen ist die Handlung Klaeren)
Strategie::StrategieSpielzuege::Handlungen::Klaeren_H::Klaeren::berechnenWennAktiv()
Strategie::StrategieSpielzuege::Handlungen::HandlungsFilter_H::HandlungsFilter::berechnen()
Strategie::Architektur::FilterTyp::ausfuehren() (in /robotsoccer/robotsoccer)
Weltmodell::startstrat() (in /robotsoccer/robotsoccer)

VIII.1.3.2.1 stratThreadStart(void*) (in /robotsoccer/robotsoccer)

thread_wrapper (vg_libpthread.c:867)

do__quit (vg_scheduler.c:1872)

Probleme mit Weltmodell::init() und Weltmodell::startstart()

XIO-Fehler: Lesen auf befreiten Speicherbereich

VIII.1.3.3 Fehlerklassen mit Valgrindlogs

VIII.1.3.3.1 Fehlerklasse 0: QT-Size-Fehler:

In der Methoden QLabel, qt-format_text, QTextlayout,... werden uninitialisierte Size Werte benutzt

==3325== Use of uninitialised value of size 4

==3325== at 0x1BF56EFE: QString::isRightToLeft() const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== by 0x1BC80C81: qt_format_text

==3325== by 0x1BC35514: QFontMetrics::boundingRect

==3325== by 0x1BD38D99: QLabel::sizeForWidth(int) const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== Conditional jump or move depends on uninitialised value(s)

==3325== at 0x1BC80DAC: qt_format_text (in /usr/lib/libqt-mt.so.3.3.4)

==3325== by 0x1BC35514: QFontMetrics::boundingRect const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== by 0x1BD38D99: QLabel::sizeForWidth(int) const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== by 0x1BD392FA: QLabel::minimumSizeHint() const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== Use of uninitialised value of size 4

==3325== by 0x1BD0A456: QTextEngine::itemize(int) (in /usr/lib/libqt-mt.so.3.3.4)

==3325== QTextLayout::beginLayout(QTextLayout::LayoutMode) (in /usr/lib/libqt-mt.so.3.3.4)

==3325== Use of uninitialised value of size 4

==3325== at 0x1BBF017F: QFontEngineXft::stringToCMap const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== by 0x1BD09890: QTextEngine::shape(int) const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== Use of uninitialised value of size 4

==3325== by 0x1BD09890: QTextEngine::shape(int) const (in /usr/lib/libqt-mt.so.3.3.4)

==3325== by 0x1BD077AD: QTextLayout::currentItem() (in /usr/lib/libqt-mt.so.3.3.4)

==30299== Syscall param write(buf) contains uninitialised or unaddressable byte(s)

==30299== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)

==30299== by 0x1CA94E0F: (within /usr/X11R6/lib/libX11.so.6.2)

==30299== by 0x1CA959FE: _X11TransWrite (in /usr/X11R6/lib/libX11.so.6.2)

==30299== by 0x1CA75261: (within /usr/X11R6/lib/libX11.so.6.2)

==30299== by 0x1CA7690C: _XReply (in /usr/X11R6/lib/libX11.so.6.2)
==30299== by 0x1CA639E0: XInternAtom (in /usr/X11R6/lib/libX11.so.6.2)
==30299== by 0x1CA8020A: XSetWMProperties (in /usr/X11R6/lib/libX11.so.6.2)
QWidget::create(unsigned long, bool, bool) (in /usr/lib/libqt-mt.so.3.3.4)
QWidget::QWidget(QWidget*, char const*, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
QDialog::QDialog(QWidget*, char const*, bool, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
at 0x1B905901: calloc (vg_replace_malloc.c:176)
by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
by 0x1BBAD449: qt_init_internal (in /usr/lib/libqt-mt.so.3.3.4)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
__libc_start_main (in /lib/libc-2.3.2.so)

VIII.1.3.3.2 Fehlerklasse I: Kameraabfragefehler

(Kamera liefert uninitialisierte, zufällige oder gar keine Werte mit denen dann weitergearbeitet wird):
Fehler treten im Programm dc1394_control.c in folgenden Methoden auf:
GetCameraROMValue (dc1394_control.c:488)
dc1394_get_camera_info (dc1394_control.c:1184,1192,1202,1221,1229,1239,...)
dc1394_get_camera_nodes (dc1394_control.c:795)
dc1394_print_camera_info (dc1394_control.c:1092)
SetCameraControlRegister (dc1394_control.c:617)
dc1394_get_camera_feature_set (dc1394_control.c:1314)
GetCameraControlRegister (dc1394_control.c:539)
SetCameraControlRegister (dc1394_control.c:602)
dc1394_dma_setup_capture (dc1394_capture.c:540)
dc1394_stop_iso_transmission (dc1394_control.c:1909)
Beispielausgaben:
==22208== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==22208== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==22208== by 0x1C12352E: raw1394_start_read (in /usr/lib/libraw1394.so.5.2.0)
==22208== by 0x1C123A86: raw1394_read (in /usr/lib/libraw1394.so.5.2.0)
==22208== by 0x1C12BAD7: GetCameraROMValue (dc1394_control.c:488)
==22208== Address 0x52BFE0DA is on thread 1's stack
==22208== Conditional jump or move depends on uninitialised value(s)

```
==22208== Use of uninitialised value of size 4
==22208==    at 0x1C2677DE: (within /lib/libc-2.3.2.so)
==22208==    by 0x1C26B010: _IO_vfprintf (in /lib/libc-2.3.2.so)
==22208==    by 0x1C270991: _IO_printf (in /lib/libc-2.3.2.so)
==22208==    by 0x1C12C6F9: dc1394_print_camera_info (dc1394_control.c:1092)
==4469== Conditional jump or move depends on uninitialised value(s)
==4469==    at 0x1C12CB39: dc1394_get_camera_info (dc1394_control.c:1184)
==1571== Conditional jump or move depends on uninitialised value(s)
==1571==    at 0x1C12BF95: GetConfigROMTaggedRegister (dc1394_control.c:732)
==1571==    by 0x1C12C879: dc1394_get_camera_info (dc1394_control.c:1143)
==1571==    by 0x1C12C11E: dc1394_get_camera_nodes (dc1394_control.c:795)
==1571== Conditional jump or move depends on uninitialised value(s)
==1571==    at 0x1C26777A: (within /lib/libc-2.3.2.so)
==1571==    by 0x1C26B010: _IO_vfprintf (in /lib/libc-2.3.2.so)
==1571==    by 0x1C270991: _IO_printf (in /lib/libc-2.3.2.so)
==1571==    by 0x1C12C6F9: dc1394_print_camera_info (dc1394_control.c:1092)
==1571== Conditional jump or move depends on uninitialised value(s)
==1571==    at 0x1C26A718: _IO_vfprintf (in /lib/libc-2.3.2.so)
==1571==    by 0x1C270991: _IO_printf (in /lib/libc-2.3.2.so)
==1571==    by 0x1C12C6F9: dc1394_print_camera_info (dc1394_control.c:1092)
==1571==    by 0x1C12C143: dc1394_get_camera_nodes (dc1394_control.c:798)
==1571== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==1571==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==1571==    by 0x1C1235CE: raw1394_start_write (in /usr/lib/libraw1394.so.5.2.0)
==1571==    by 0x1C123B36: raw1394_write (in /usr/lib/libraw1394.so.5.2.0)
==1571==    by 0x1C12BD01: SetCameraControlRegister (dc1394_control.c:617)
==1571==  Address 0x52BFE184 is on thread 1's stack
==1571== Conditional jump or move depends on uninitialised value(s)
==1571==    at 0x1C12BB71: GetCameraControlRegister (dc1394_control.c:539)
==1571==    by 0x1C12EB1C: dc1394_is_feature_present (dc1394_control.c:2345)
==1571==    by 0x1C12CDEC: dc1394_get_camera_feature (dc1394_control.c:1339)
==1571==    by 0x1C12CD91: dc1394_get_camera_feature_set (dc1394_control.c:1314)
==1571== Conditional jump or move depends on uninitialised value(s)
==1571==    at 0x1C12BC91: SetCameraControlRegister (dc1394_control.c:602)
==1571==    by 0x1C12DD0D: dc1394_set_iso_channel_and_speed (dc1394_control.c:1876)
==1571==    by 0x1C130004: _dc1394_basic_setup (dc1394_capture.c:166)
```

==1571== by 0x1C13085B: dc1394_dma_setup_capture (dc1394_capture.c:540)
==3325== Conditional jump or move depends on uninitialised value(s)
==3325== at 0x1C12BC91: SetCameraControlRegister (dc1394_control.c:602)
==3325== by 0x1C12DE7B: dc1394_stop_iso_transmission (dc1394_control.c:1909)

VIII.1.3.3.3 Fehlerklasse II: QT-Fehler vor allem in QT-INIT und QSessionManager

==22214== Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
==22214== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==22214== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
==22214== by 0x1BBAD449: qt_init_internal (in /usr/lib/libqt-mt.so.3.3.4)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==22214== Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==22214== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==22214== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
==22214== by 0x1BBAD449: qt_init_internal (in /usr/lib/libqt-mt.so.3.3.4)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==22214== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==22214== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==22214== by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==22214== by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==22214== by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==22214== Address 0x1D06EAD4 is 12 bytes inside a block of size 1024 alloc'd
==22214== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==22214== by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==22214== by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
QSessionManager::QSessionManager(QApplication*, QString&, QString&)
==22208== Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==22208== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==22208== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
==22208== by 0x1BBAD449: qt_init_internal (in /usr/lib/libqt-mt.so.3.3.4)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==22208== Address 0x1D061074 is 12 bytes inside a block of size 1024 alloc'd
==22208== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==22208== by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==22208== by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
QSessionManager::QSessionManager(QApplication*, QString&, QString&)

==22208== Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
==22208== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==22208== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==21309== Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
==21309== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==21309== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==21309== Syscall param writev(vector[...]) contains uninitialised or unaddressable byte(s)
==21309== at 0x1C2F195E: (within /lib/libc-2.3.2.so)
==21309== by 0x1CA94E8F: (within /usr/X11R6/lib/libX11.so.6.2)
==21309== by 0x1CA95A5E: _X11TransWritev (in /usr/X11R6/lib/libX11.so.6.2)
==21309== by 0x1CA76176: _XSend (in /usr/X11R6/lib/libX11.so.6.2)
==21309== Address 0x1D064044 is 12 bytes inside a block of size 1024 alloc'd
==21309== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==21309== by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==21309== by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
QSessionManager::QSessionManager(QApplication*, QString&, QString&)

VIII.1.3.3.4 Fehlerklasse III: Input-Output-Control-Warnings

Warning: noted but unhandled ioctl 0x5415 with no size/direction hints
This could cause spurious value errors to appear.
See README_MISSING_SYSCALL_OR_IOCTL for guidance on writing a proper wrapper.

VIII.1.3.3.5 Fehlerklasse IV: Libraryfehler in libraw1394,libX11,libqt und libICE

Methoden: raw1394_get_port_info,qt_init,QSessionmanager,_IceWrite,_XSend,Q_Label,
qt_format_text,QTextLayout,QDialog,QTextEngine,libdc_startmain...
==1571== Source and destination overlap in strcpy(0x52BFE384, 0x52BFE384)
==1571== at 0x1B904859: strcpy (mac_replace_strmem.c:102)
==1571== by 0x1C122D4E: raw1394_get_port_info (in /usr/lib/libraw1394.so.5.2.0)
==22208== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==22208== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==22208== by 0x1C122CC9: raw1394_get_port_info (in /usr/lib/libraw1394.so.5.2.0)
==22208== Conditional jump or move depends on uninitialised value(s)

```
==22208==    at 0x1C122D2B: raw1394_get_port_info (in /usr/lib/libraw1394.so.5.2.0)
==30333== Syscall param writev(vector[...]) contains uninitialised or unaddressable byte(s)
==30333==    at 0x1C2F195E: (within /lib/libc-2.3.2.so)
==30333==    by 0x1CA94E8F: (within /usr/X11R6/lib/libX11.so.6.2)
==30333==    by 0x1CA95A5E: _X11TransWritev (in /usr/X11R6/lib/libX11.so.6.2)
==30333==    by 0x1CA76176: _XSend (in /usr/X11R6/lib/libX11.so.6.2)
==30333==  Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==30333==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==30333==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
==30333==    by 0x1BBAE0D7: qt_init(int*, char**, QApplication::Type)
==30333== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==30333==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==30333==    by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==30333==    by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==30333==    by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==30333== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==30333==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==30333==    by 0x1C122CC9: raw1394_get_port_info (in /usr/lib/libraw1394.so.5.2.0)
==30299== Syscall param writev(vector[...]) contains uninitialised or unaddressable byte(s)
==30299==    at 0x1C2F195E: (within /lib/libc-2.3.2.so)
==30299==    by 0x1CA94E8F: (within /usr/X11R6/lib/libX11.so.6.2)
==30299==    by 0x1CA95A5E: _X11TransWritev (in /usr/X11R6/lib/libX11.so.6.2)
==30299==    by 0x1CA76176: _XSend (in /usr/X11R6/lib/libX11.so.6.2)
==30299==    by 0x1CBE3850: XRenderAddGlyphs (in /usr/lib/libXrender.so.1.2.2)
==30299==    by 0x1CC056FE: XftFontLoadGlyphs (in /usr/lib/libXft.so.2.1.1)
==30299==    by 0x1CC02A1D: XftGlyphExtents (in /usr/lib/libXft.so.2.1.1)
==30299==    by 0x1BBEFF60: QFontEngineXft::stringToCMap const (in /usr/lib/libqt-mt.so.3.3.4)
==30299==  Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==30299==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==30299==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==30299==    by 0x1BC19144: QApplication::construct(int&, char**, QApplication::Type)
==30299==    by 0x1BC18DF8: QApplication::QApplication(int&, char**)
==30299==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
```

```
==30304== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==30304==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==30304==    by 0x1CA94E0F: (within /usr/X11R6/lib/libX11.so.6.2)
==30304==    by 0x1CA959FE: _X11TransWrite (in /usr/X11R6/lib/libX11.so.6.2)
==30304==    by 0x1CA75261: (within /usr/X11R6/lib/libX11.so.6.2)
==30304==    by 0x1CA7690C: _XReply (in /usr/X11R6/lib/libX11.so.6.2)
==30304==    by 0x1CA639E0: XInternAtom (in /usr/X11R6/lib/libX11.so.6.2)
==30304==    by 0x1CA8020A: XSetWMProperties (in /usr/X11R6/lib/libX11.so.6.2)
QWidget::create(unsigned long, bool, bool) (in /usr/lib/libqt-mt.so.3.3.4)
QWidget::QWidget(QWidget*, char const*, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
QDialog::QDialog(QWidget*, char const*, bool, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
==30304==  Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==30304==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==30304==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==30304==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==30304== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==30304==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==30304==    by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==30304==    by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==30304==    by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==30304==    by 0x1CC8D041: IceFlush (in /usr/X11R6/lib/libICE.so.6.3)
==30304==    by 0x1CC7ED2E: SmcSetProperties (in /usr/X11R6/lib/libSM.so.6.0)
==30304==  Address 0x1D062124 is 12 bytes inside a block of size 1024 alloc'd
==30304==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==30304==    by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==30304==    by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
QSessionManager::QSessionManager(QApplication*, QString&, QString&)
QApplication::initialize(int, char**) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==30304==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==30313== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
```

```
==30313==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==30313==    by 0x1CA94E0F: (within /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CA959FE: _X11TransWrite (in /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CA75261: (within /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CA7690C: _XReply (in /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CA639E0: XInternAtom (in /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CA8020A: XSetWMProperties (in /usr/X11R6/lib/libX11.so.6.2)
QWidget::create(unsigned long, bool, bool) (in /usr/lib/libqt-mt.so.3.3.4)
QWidget::QWidget(QWidget*, char const*, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
QDialog::QDialog(QWidget*, char const*, bool, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
==30313==  Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
==30313==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==30313==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==30313==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==30313== Syscall param writev(vector[...]) contains uninitialised or unaddressable byte(s)
==30313==    at 0x1C2F195E: (within /lib/libc-2.3.2.so)
==30313==    by 0x1CA94E8F: (within /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CA95A5E: _X11TransWritev (in /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CA76176: _XSend (in /usr/X11R6/lib/libX11.so.6.2)
==30313==    by 0x1CBE3850: XRenderAddGlyphs (in /usr/lib/libXrender.so.1.2.2)
==30313==    by 0x1CC056FE: XftFontLoadGlyphs (in /usr/lib/libXft.so.2.1.1)
==30313==    by 0x1CC02A1D: XftGlyphExtents (in /usr/lib/libXft.so.2.1.1)
==30313==    by 0x1BBEFF60: QFontEngineXft::stringToCMap const (in /usr/lib/libqt-mt.so.3.3.4)
==30313==    by 0x1BD00A5D: (within /usr/lib/libqt-mt.so.3.3.4)
==30313==    by 0x1BD00B48: (within /usr/lib/libqt-mt.so.3.3.4)
==30313==  Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==30313==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==30313==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
```

```
==30313==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==30313== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==30313==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==30313==    by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==30313==    by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==30313==    by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==30313==    by 0x1CC8D041: IceFlush (in /usr/X11R6/lib/libICE.so.6.3)
==30313==    by 0x1CC7ED2E: SmcSetProperties (in /usr/X11R6/lib/libSM.so.6.0)
==30313==  Address 0x1D062124 is 12 bytes inside a block of size 1024 alloc'd
==30313==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==30313==    by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==30313==    by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
QSessionManager::QSessionManager(QApplication*, QString&, QString&)
==30313==    by 0x1BC19962: QApplication::initialize(int, char**) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==30313==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==31346== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==31346==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==31346==    by 0x1CA94E0F: (within /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CA959FE: _X11TransWrite (in /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CA75261: (within /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CA7690C: _XReply (in /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CA639E0: XInternAtom (in /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CA8020A: XSetWMProperties (in /usr/X11R6/lib/libX11.so.6.2)
QWidget::create(unsigned long, bool, bool) (in /usr/lib/libqt-mt.so.3.3.4)
 QWidget::QWidget(QWidget*, char const*, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
 QDialog::QDialog(QWidget*, char const*, bool, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
==31346==  Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
==31346==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==31346==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
 QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==31346==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
```

```
==31346== Syscall param writev(vector[...]) contains uninitialised or unaddressable byte(s)
==31346==    at 0x1C2F195E: (within /lib/libc-2.3.2.so)
==31346==    by 0x1CA94E8F: (within /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CA95A5E: _X11TransWritev (in /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CA76176: _XSend (in /usr/X11R6/lib/libX11.so.6.2)
==31346==    by 0x1CBE3850: XRenderAddGlyphs (in /usr/lib/libXrender.so.1.2.2)
==31346==    by 0x1CC056FE: XftFontLoadGlyphs (in /usr/lib/libXft.so.2.1.1)
==31346==    by 0x1CC02A1D: XftGlyphExtents (in /usr/lib/libXft.so.2.1.1)
QFontEngineXft::stringToCMap(QChar const*, int, unsigned short*, int*, int*, bool) const
==31346==    by 0x1BD00A5D: (within /usr/lib/libqt-mt.so.3.3.4)
==31346==    by 0x1BD00B48: (within /usr/lib/libqt-mt.so.3.3.4)
==31346==  Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==31346==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==31346==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==31346==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==31346== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==31346==    at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==31346==    by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==31346==    by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==31346==    by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==31346==    by 0x1CC8D041: IceFlush (in /usr/X11R6/lib/libICE.so.6.3)
==31346==    by 0x1CC7ED2E: SmcSetProperties (in /usr/X11R6/lib/libSM.so.6.0)
==31346==    by 0x1BBB5CD3: (within /usr/lib/libqt-mt.so.3.3.4)
==31346==  Address 0x1D062124 is 12 bytes inside a block of size 1024 alloc'd
==31346==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==31346==    by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==31346==    by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
QSessionManager::QSessionManager(QApplication*, QString&, QString&)
QApplication::initialize(int, char**) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==31346==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
```

==31943== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==31943== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==31943== by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==31943== by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==31943== by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==31943== Address 0x1D06EACC is 12 bytes inside a block of size 1024 alloc'd
==31943== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==31943== by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==31943== by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
==31943== by 0x1BBB73CA: QSessionManager::QSessionManager (in /usr/lib/libqt-mt.so.3.3.4)

VIII.1.3.3.6 Fehlerklasse V: Terminator

(sofortiger Programmabsturz mit Speicherzugriffsfehler bei Programmstart):
==31452== Process terminating with default action of signal 11 (SIGSEGV)
==31452== at 0x1B8E8CAE: (within /lib/ld-2.3.2.so)
==31452== by 0x1B8EA5A5: _dl_map_object (in /lib/ld-2.3.2.so)
==31452== by 0x1B8E6EC2: (within /lib/ld-2.3.2.so)
==31452== by 0x1B8F31F6: (within /lib/ld-2.3.2.so)

VIII.1.3.3.7 Fehlerklasse VI: Fehler in qt_init:

==31943== Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
==31943== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==31943== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==31943== Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==31943== at 0x1B905901: calloc (vg_replace_malloc.c:176)
==31943== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
==20396== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==20396== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==20396== by 0x1CA94E0F: (within /usr/X11R6/lib/libX11.so.6.2)
==20396== by 0x1CA959FE: _X11TransWrite (in /usr/X11R6/lib/libX11.so.6.2)
==20396== by 0x1CA75261: (within /usr/X11R6/lib/libX11.so.6.2)
==20396== by 0x1CA7690C: _XReply (in /usr/X11R6/lib/libX11.so.6.2)

```
==20396==    by 0x1CA639E0: XInternAtom (in /usr/X11R6/lib/libX11.so.6.2)
==20396==    by 0x1CA8020A: XSetWMProperties (in /usr/X11R6/lib/libX11.so.6.2)
QWidget::create(unsigned long, bool, bool) (in /usr/lib/libqt-mt.so.3.3.4)
QWidget::QWidget(QWidget*, char const*, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
 QDialog::QDialog(QWidget*, char const*, bool, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
==20396==  Address 0x1CDDF404 is 276 bytes inside a block of size 2048 alloc'd
==20396==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==20396==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==20396==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==20396== Syscall param writev(vector[...]) contains uninitialised or unaddressable byte(s)
==20396==    at 0x1C2F195E: (within /lib/libc-2.3.2.so)
==20396==    by 0x1CA94E8F: (within /usr/X11R6/lib/libX11.so.6.2)
==20396==    by 0x1CA95A5E: _X11TransWritev (in /usr/X11R6/lib/libX11.so.6.2)
==20396==    by 0x1CA76176: _XSend (in /usr/X11R6/lib/libX11.so.6.2)
==20396==    by 0x1CBE3850: XRenderAddGlyphs (in /usr/lib/libXrender.so.1.2.2)
==20396==    by 0x1CC056FE: XftFontLoadGlyphs (in /usr/lib/libXft.so.2.1.1)
==20396==    by 0x1CC02A1D: XftGlyphExtents (in /usr/lib/libXft.so.2.1.1)
 QFontEngineXft::stringToCMap() const (in /usr/lib/libqt-mt.so.3.3.4)
==20396==    by 0x1BD00A5D: (within /usr/lib/libqt-mt.so.3.3.4)
==20396==    by 0x1BD00B48: (within /usr/lib/libqt-mt.so.3.3.4)
==20396==  Address 0x1CDDF32B is 59 bytes inside a block of size 2048 alloc'd
==20396==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==20396==    by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==20396==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==31500== Conditional jump or move depends on uninitialised value(s)
==31500==    at 0x1CB22F2A: FcFreeTypeCharSetAndSpacing (in /usr/lib/libfontconfig.so.1.0.4)
==31500==    by 0x1CB217C1: FcFreeTypeQuery (in /usr/lib/libfontconfig.so.1.0.4)
==31500==    by 0x1CB201EE: FcFileScanConfig (in /usr/lib/libfontconfig.so.1.0.4)
```

==31500== by 0x1CB206BB: FcDirScanConfig (in /usr/lib/libfontconfig.so.1.0.4)
==31500== by 0x1CB1AF89: FcConfigBuildFonts (in /usr/lib/libfontconfig.so.1.0.4)
==31500== by 0x1CB238FB: FcInitLoadConfigAndFonts (in /usr/lib/libfontconfig.so.1.0.4)
==31500== by 0x1CB23954: FcInit (in /usr/lib/libfontconfig.so.1.0.4)
==31500== by 0x1CC064FA: XftInit (in /usr/lib/libXft.so.2.1.1)
qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)
qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)

VIII.1.3.3.8 Fehlerklasse XII: ICE-Errors: IceWrite,IceFlush,...

==22208== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==22208== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==22208== by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==22208== by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==22208== by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==21309== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==21309== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==21309== by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==21309== by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==21309== by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==20396== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==20396== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==20396== by 0x1CC9591F: (within /usr/X11R6/lib/libICE.so.6.3)
==20396== by 0x1CC966AE: _IceTransWrite (in /usr/X11R6/lib/libICE.so.6.3)
==20396== by 0x1CC8D42A: _IceWrite (in /usr/X11R6/lib/libICE.so.6.3)
==20396== by 0x1CC8D041: IceFlush (in /usr/X11R6/lib/libICE.so.6.3)
==20396== by 0x1CC7ED2E: SmcSetProperties (in /usr/X11R6/lib/libSM.so.6.0)

VIII.1.3.3.9 Fehlerklasse XIII: QDialog,QWidget,QFontEngine,QApplication:exec

enthalten unadressierbare oder uninitialisierte Werte:

==31467== Syscall param write(buf) contains uninitialised or unaddressable byte(s)
==31467== at 0x1C2EAE18: write (in /lib/libc-2.3.2.so)
==31467== by 0x1CA94E0F: (within /usr/X11R6/lib/libX11.so.6.2)
==31467== by 0x1CA959FE: _X11TransWrite (in /usr/X11R6/lib/libX11.so.6.2)
==31467== by 0x1CA75261: (within /usr/X11R6/lib/libX11.so.6.2)
==31467== by 0x1CA7690C: _XReply (in /usr/X11R6/lib/libX11.so.6.2)
==31467== by 0x1CA639E0: XInternAtom (in /usr/X11R6/lib/libX11.so.6.2)

```
==31467==    by 0x1CA8020A: XSetWMProperties (in /usr/X11R6/lib/libX11.so.6.2)
QWidget::create(unsigned long, bool, bool) (in /usr/lib/libqt-mt.so.3.3.4)
QWidget::QWidget(QWidget*, char const*, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
QDialog::QDialog(QWidget*, char const*, bool, unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
==31467== Syscall param writev(vector[...]) contains uninitialised or unaddressable byte(s)
==31467==    at 0x1C2F195E: (within /lib/libc-2.3.2.so)
==31467==    by 0x1CA94E8F: (within /usr/X11R6/lib/libX11.so.6.2)
==31467==    by 0x1CA95A5E: _X11TransWritev (in /usr/X11R6/lib/libX11.so.6.2)
==31467==    by 0x1CA76176: _XSend (in /usr/X11R6/lib/libX11.so.6.2)
==31467==    by 0x1CBE3850: XRenderAddGlyphs (in /usr/lib/libXrender.so.1.2.2)
==31467==    by 0x1CC056FE: XftFontLoadGlyphs (in /usr/lib/libXft.so.2.1.1)
==31467==    by 0x1CC02A1D: XftGlyphExtents (in /usr/lib/libXft.so.2.1.1)
QFontEngineXft::stringToCMap(QChar const*, int, unsigned short*, int*, int*, bool)
==31467==    by 0x1BD00B48: (within /usr/lib/libqt-mt.so.3.3.4)
 ==20396==  Address 0x1D062124 is 12 bytes inside a block of size 1024 alloc'd
==20396==    at 0x1B905901: calloc (vg_replace_malloc.c:176)
==20396==    by 0x1CC8A51F: IceOpenConnection (in /usr/X11R6/lib/libICE.so.6.3)
==20396==    by 0x1CC7E428: SmcOpenConnection (in /usr/X11R6/lib/libSM.so.6.0)
QSessionManager::QSessionManager(QApplication*, QString&, QString&)
QApplication::initialize(int, char**) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==20396==    by 0x1C236E35: __libc_start_main (in /lib/libc-2.3.2.so)
==31467== Conditional jump or move depends on uninitialised value(s)
==31467==    at 0x817EE0A: OpenglTab::mouseMoveEvent(QMouseEvent*)
 QWidget::event(QEvent*) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::internalNotify(QObject*, QEvent*) (in /usr/lib/libqt-mt.so.3.3.4)
 QApplication::notify(QObject*, QEvent*) (in /usr/lib/libqt-mt.so.3.3.4)
 QETWidget::translateMouseEvent(_XEvent const*) (in /usr/lib/libqt-mt.so.3.3.4)
 QApplication::x11ProcessEvent(_XEvent*) (in /usr/lib/libqt-mt.so.3.3.4)
==31467==    by 0x1BBC7253: QEventLoop::processEvents(unsigned) (in /usr/lib/libqt-mt.so.3.3.4)
==31467==    by 0x1BC301D7: QEventLoop::enterLoop() (in /usr/lib/libqt-mt.so.3.3.4)
==31467==    by 0x1BC30087: QEventLoop::exec() (in /usr/lib/libqt-mt.so.3.3.4)
==31467==    by 0x1BC1E070: QApplication::exec() (in /usr/lib/libqt-mt.so.3.3.4)
```

VIII.1.3.3.10 Fehlerklasse IX: Strategiefehler:

1) In der Handlung Klaeren_H, verursacht durch vg_scheduler.c und Weltmodell:startstart():

==31467== Invalid read of size 4

Strategie::StrategieSpielzuege::Handlungen::Klaeren_H::Klaeren::berechnenWennAktiv()

Strategie::StrategieSpielzuege::Handlungen::HandlungsFilter_H::HandlungsFilter::berechnen()

==31467== by 0x80EC398: Strategie::Architektur::FilterTyp::ausfuehren()

==31467== by 0x805C902: Weltmodell::startstrat()

==31467== by 0x805D321: stratThreadStart(void*)

==31467== by 0x1C0F5D52: thread_wrapper (vg_libpthread.c:867)

==31467== by 0xB000F5DF: do__quit (vg_scheduler.c:1872)

verursacht durch die Methode Weltmodell:init() innerhalb der main-Methode von robotsoccer:

==31467== Address 0x1D194148 is 0 bytes after a block of size 240 alloc'd

==31467== at 0x1B9052D9: operator new[](unsigned) (vg_replace_malloc.c:139)

Strategie::RohrTypen::IstSituation_H::IstSituation::IstSituation() Strategie::Architektur::Tafel<Strategie::RohrTypen::IstSituation_H::IstSituation>::Tafel() Strategie::Architektur::Tafel<Strategie::RohrTypen::IstSituation_H::IstSituation>::Schnittstelle::Schnittstelle()

Strategie::Architektur::AusgabeAnschluss<Strategie::Architektur::Tafel<Strategie::RohrTypen::IstSituation_H::IstSituation> >::AusgabeAnschluss() Strategie::Architektur::Filter<Strategie::FilterTypen::Eingang_H::Eingang, (Strategie::Architektur::Ausfuehrung)1>::Filter()

Strategie::Gesamtsystem_H::Gesamtsystem::Gesamtsystem()Strategie::Architektur::Filter<Strategie::Gesamtsystem_H::Gesamtsystem, (Strategie::Architektur::Ausfuehrung)1>::Filter()

==31467== by 0x8059275: Weltmodell::init() (in /robotsoccer/robotsoccer)

==31467== Address 0x1CDE0ABC is 212 bytes inside a block of size 2048 alloc'd

==31467== at 0x1B905901: calloc (vg_replace_malloc.c:176)

==31467== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)

qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)

qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)

QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)

QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)

==31467== by 0x8057B79: Weltmodell::init() (in //robotsoccer/robotsoccer)

==31467== by 0x805793D: main (in /robotsoccer/robotsoccer)

==31467== Address 0x1CDE0A23 is 59 bytes inside a block of size 2048 alloc'd

==31467== at 0x1B905901: calloc (vg_replace_malloc.c:176)

==31467== by 0x1CA6709C: XOpenDisplay (in /usr/X11R6/lib/libX11.so.6.2)

qt_init_internal(int*, char**, _XDisplay*, unsigned long, unsigned long)

qt_init(int*, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)

QApplication::construct(int&, char**, QApplication::Type) (in /usr/lib/libqt-mt.so.3.3.4)
QApplication::QApplication(int&, char**) (in /usr/lib/libqt-mt.so.3.3.4)
==31467== by 0x8057B79: Weltmodell::init() (in /robotsoccer/robotsoccer)
==31467== by 0x805793D: main (in /robotsoccer/robotsoccer)

VIII.1.3.3.11 Fehlerklasse X: XIO Error

==31467== Invalid read of size 4
==31467== by 0x1C32CEE4: __libc_freeres (in /lib/libc-2.3.2.so)
_vgw(float, long double,...)(...)(long double,...)(short) (vg_intercept.c:117)
==31467== by 0x1C24CB97: exit (in /lib/libc-2.3.2.so)
==31467== by 0x1BBA69D3: (within /usr/lib/libqt-mt.so.3.3.4)
==31467== Address 0x1DA56944 is 68 bytes inside a block of size 120 free'd
==31467== at 0x1B905460: free (vg_replace_malloc.c:153)
_vgw(float, long double,...)(...)(long double,...)(short) (vg_intercept.c:117)
==31467== by 0x1C24CB97: exit (in /lib/libc-2.3.2.so)
==31467== by 0x1BBA69D3: (within /usr/lib/libqt-mt.so.3.3.4)
==31467== by 0x1CA786CE: _XIOError (in /usr/X11R6/lib/libX11.so.6.2)
==31467== by 0x1CA75DE2: _XRead (in /usr/X11R6/lib/libX11.so.6.2)

IX. Fazit

IX.1 Fazit

Die Arbeit der Projektgruppe war insgesamt sehr erfolgreich. Mit der lokalen Bildverarbeitung konnte sich zwar aufgrund mangelnder Hardwarevoraussetzungen nur ein Teil der Projektgruppe beschäftigen, trotzdem wurden hier wichtige Grundlagen, wie zum Beispiel ein modulares Kameraframework, geschaffen. Die Studien zur Einsetzbarkeit des Kompasssensors und die Untersuchung verschiedener Ballfindungsansätze liefern ebenfalls wichtige Erkenntnisse für folgende Projektgruppen. Der Robotercode ist nun ebenfalls viel klarer strukturiert und leichter zu verstehen. Auch auf Seiten des Hostsystems wurde viel Arbeit getan. Die Geschwindigkeit und Qualität der Bildverarbeitung wurde drastisch verbessert, die Kameraentzerrung lieferte hier einen wichtigen Beitrag. Die Anpassung und Erweiterung des Anfahrtsalgorithmus sollte einen positiven Einfluss auf zukünftige Spiele haben. Hier wurde viel Arbeit bei der Anpassung auf die neuen Roboter geleistet. Die Strategiekomponente wurde so verbessert, dass sie sich besser für den Einsatz mit elf Robotern eignet. Einige alte Konzepte wurden verbessert, sowie neue Handlungen hinzugefügt.

IX.2 Danksagungen

Die Projektgruppe dankt ihren Betreuern Lars Hildebrand und Norman Weiss, sowie Prof. Reusch, für die Betreuung. Weiter geht unser Dank an die HiWis des LS1, die uns stets mit Rat und Tat zur Seite standen. Für die exzellente Verpflegung danken wir dem Divan-Grill und insbesondere unserem Kühlschrankwart Daniel Mikus.

X. Literaturverzeichnis

AverLogic. Datenblatt AL422B FIFO. http://www.averlogic.com/password/AL422B051104.pdf

Agilent. Datenblatt ADCM-1700 CIF CMOS Camera Module.

DongHun Lee et al. A Novel Color Patch System for Large League MIROSOT ,2005.

Kameragruppe PG472. CVS - Modul.
soccer.cs.uni-dortmund.de/development/cvsroot/pg472_kameragruppe

Paul S. Heckbert. Fundamentals of Texture Mapping and Image Warping.Master's Thesis. University of California, Berkley. Juni 1989.
http://www-2.cs.cmu.edu/~ph/texfund/texfund.pdf

Rüping, Stefan. mySVM - a support vector machine.
http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/index.html

Robobrain. Datenblätter gesammelt.
https://www.robosoccer.de/intern/files/Robobrain/datasheets

TexasInstruments. Datenblatt TMS320F2812 DSP.
http://focus.ti.com/lit/ds/symlink/tms320f2812.pdf

Z. Zhang. Flexible Camera Calibration By Viewing a Plane From Unknown Orientations. International Conference on Computer Vision (ICCV'99), Corfu, Greece, September 1999.
http://www.vision.caltech.edu/bouguetj/calib_doc/papers/zhan99.pdf

XI. Abbildungsverzeichnis

XII. Tabellenverzeichnis