UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Sony Legged League: Entwicklung von verteilten Algorithmen zur effizienten Kontrolle von autonomen Fußballrobotern

PG 426

Teilnehmer: Sebastian Deutsch, Thomas Dickhöfer, Wenchuan Ding, Kai Engel, Piotr Kudlacik, André Osterhues, Jan Prünte, Andreas Reiß, Sebastian Schmidt, Christian Thiel, Michael Wachter

Betreuer: Ingo Dahm, Christoph Richter, Jens Ziegler

September 2003

ENDBERICHT

Lehrstuhl für Systemanalyse
Fachbereich Informatik der
Universität Dortmund

Computer Engineering Institute
Fachbereich Elektrotechnik der
Universität Dortmund


Microsoft
Hellhounds

Inhaltsverzeichnis

Kapitel 1

Einleitung

> Die Drei Gesetze der Robotik:
>
> 1. Ein Roboter darf keinen Menschen verletzen oder es durch Untätigkeit zulassen, dass ein Mensch zu Schaden kommt.
> 2. Ein Roboter muss die Befehle, die ihm durch Menschen gegeben werden, ausführen außer wenn solche Befehle im Widerspruch zum Ersten Gesetz stehen.
> 3. Ein Roboter muss seine eigene Existenz schützen so lang dies nicht in Widerspruch zum Ersten oder Zweiten Gesetz steht.
>
> – Aus „Ich, Roboter“ von Isaac Asimov.

Die Projektgruppe 426 (nachfolgend als „Projektgruppe“ bezeichnet) befasste sich mit Fußball spielenden Robotern. Eingesetzt wurden Roboter des Typs „AIBO“ (siehe 2.1 auf der nächsten Seite) des japanischen Herstellers Sony.

Um ein sinnvolles Fußballspiel (siehe Spielregeln 2.2 auf Seite 8) zu ermöglichen, ist es notwendig, dass die Schwerpunkte Künstliche Intelligenz, Laufbewegung, Bildverarbeitung und Kommunikation geschickt kombiniert werden. Daher sind Fußball spielende Roboter gerade für die Forschung interessant.

Die Projektgruppe ist Teil des GermanTeams, welches sich aus Studenten, wissenschaftlichen Mitarbeitern und Professoren der HU Berlin, der TU Bremen, der TU Darmstadt und der Uni Dortmund zusammensetzt. Der bisher vom GermanTeam gemeinsam entwickelte Code „GT2002“ sollte als „GT2003“ (siehe Kapitel 2.4 auf Seite 10) weitergeführt und verbessert werden. Durch ein ausgefeiltes Software-Konzept ist es möglich, Programmteile sowohl gemeinsam als auch in Konkurrenz zu den anderen Unis zu entwickeln (siehe Kapitel 2.4.2 auf Seite 13).

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit der Projektgruppe – angefangen vom Roboter bis hin zur verwendeten Software – erläutert.

2.1 Der Sony AIBO

Die von der Projektgruppe genutzten Roboter heißen „AIBO“ (Artificially Intelligent roBOt bzw. japanisch „aibou“ = „Freund“) und haben die offizielle Bezeichnung Sony ERS-2100 und ERS-220. In ihrem äußeren Erscheinungsbild sind sie Hunden nachempfunden (siehe Abbildung 2.1 auf der nächsten Seite).

AIBOs haben 18 Gelenke:

- drei pro Bein
- drei am Hals (zur Bewegung des Kopfes)
- zwei für den Schwanz
- eins für das Maul

Durch diese Vielzahl an Freiheitsgraden werden komplexe Bewegungsabläufe möglich. Jedes Gelenk enthält einen Servomotor, dessen aktuelle Position abgefragt werden kann.

Im beweglichen Kopf befinden sich diverse Sensoren:

- eine CMOS-Kamera, die YUV-Bilder in einer Auflösung von 176x144 Bildpunkten liefert
- ein Infrarotsensor, der für die Abstandsmessung benutzt werden kann
- zwei Mikrophone, jeweils eins an jeder Wange (für Stereophonie)

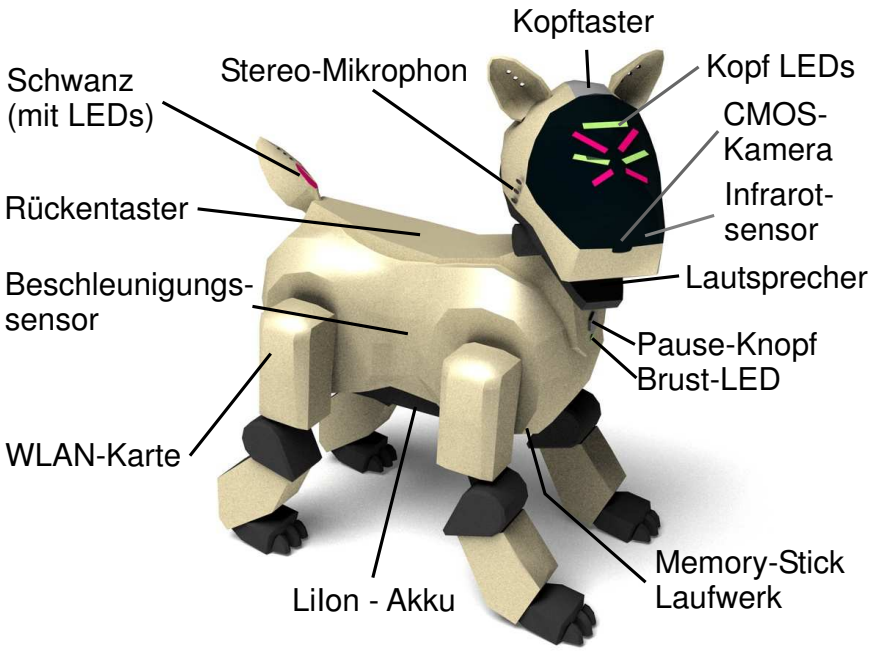

Abbildung 2.1: AIBO

Ein 3D-Beschleunigungssensor ist im Rumpf des AIBOs untergebracht. Mit seiner Hilfe kann u.a. überprüft werden, ob der Roboter umgefallen ist.

Zwei Taster (einer auf dem Kopf, der andere auf dem Rücken) können dazu benutzt werden, um bestimmte Aktionen zu schalten (z.B. das Zurücklaufen auf die Ausgangsposition beim Anstoß).

Im Inneren des Roboters befindet sich eine Batterie, ein Lese-/Schreibegerät für Memorysticks, ein PCMCIA-Slot für Erweiterungen und die Hauptplatine, auf der CPU, RAM, Flash-ROM und diverse Controller untergebracht sind.

Als Prozessor kommt eine 64-Bit MIPS R4000 RISC-CPU von Quantum Effect Devices (QED) mit 200 MHz zum Einsatz. Neuere AIBOs (ERS-220, die so genannten „Supercores“) haben eine mit 400 MHz getaktete MIPS-CPU. Die Kapazität des SDRAM-Speichers beträgt 16 MB (32 MB bei Supercores).

Die grundlegende Software (BIOS) ist in einem 4 MB großen Flash-ROM untergebracht. Sie umfasst den Betriebssystemkern sowie ein Programm mit einem relativ einfaches Verhalten für den AIBO. Dieses wird nur dann gestartet, wenn kein Memorystick im Laufwerk steckt. Weitere Software kann über das eingebautes Memorystick-Laufwerk eingespielt und verwendet werden. Es können jedoch nur spezielle AIBO-Memorysticks, die so genannten „Pink Sticks“, mit einer Kapazität von 8, 16 oder 32 MB benutzt werden.

Als Ausgabeeinheiten dienen zahlreiche verschiedenfarbige LEDs an Kopf und Schwanz sowie ein Lautsprecher, der sich im Mund des Roboters befindet.

Die Kommunikation mit Computern und anderen Robotern ist über eine Funknetzwerk-

schnittstelle (Wireless LAN bzw. WLAN) möglich. Die von der Projektgruppe genutzten Roboter verfügen alle über eine solche WLAN-Erweiterung.

Eine weitere zusätzliche Hardware stellt die sog. Debugbox dar. Diese kann auf dem Rücken des AIBOs angebracht werden und verfügt über einen Videoausgang und eine serielle Schnittstelle. Dadurch ist es möglich, sich Debug-Meldungen und die Kameradaten anzeigen zu lassen.

2.2 Spielregeln

Gespielt wird mit einem orangen Ball auf einem 4,60m x 3,10m großen Feld (siehe Abbildung 2.2). Die Tore sind 60cm breit und farbig gekennzeichnet (Gelb und Hellblau).

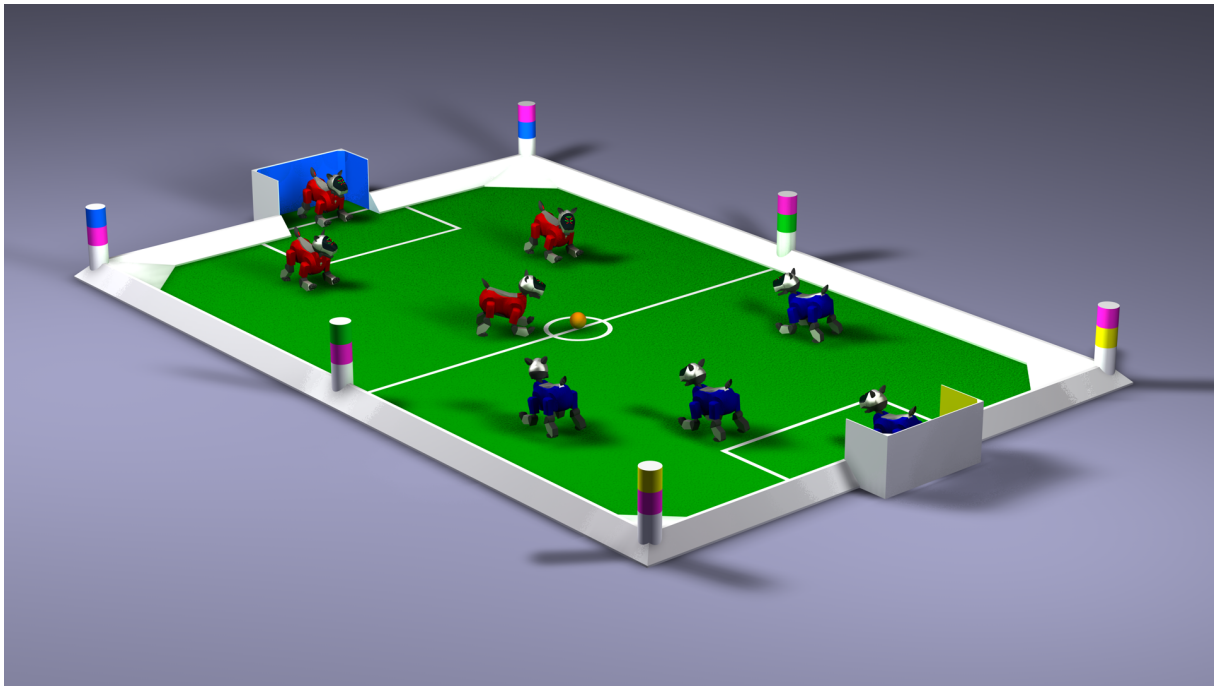

Abbildung 2.2: Das Spielfeld. Die Roboter orientieren sich anhand der Tore und der farbigen Landmarken

Jeweils vier Roboter bilden eine Mannschaft, die entweder an roten oder blauen Trikots zu erkennen ist. Jeder Roboter hat dabei eine festgelegte Rolle (Torwart, Verteidiger oder Angreifer).

Am Rand des Feldes befinden sich Landmarken, die den Robotern zur Orientierung dienen. Jede Landmarke hat einen eindeutigen Farbcode, der sich aus den Farben Weiss, Pink und einer weiteren zusammensetzt. Dabei stehen Gelb und Hellblau für die Seiten (entsprechend der Torfarbe) und Grün für die Mitte.

Ein Spiel geht über zwei Halbzeiten à 10 Minuten. In der Halbzeitpause werden die Trikots und die Seiten gewechselt.

Weitere Details stehen im offiziellen Regelwerk des Technischen Komitees des Robocup (siehe [1]).

2.3 API und Betriebssystem des AIBOs

Um den AIBO programmieren zu können, bietet Sony eine Entwicklungsumgebung an. Diese besteht aus dem Betriebssystem Aperios und einer darüber liegenden Middleware API-Bibliothek „Open-R“.

2.3.1 Aperios

Aperios ist ein Betriebssystem, das aus dem Betriebssystem Apertos hervorgegangen ist. Aperios wird auf vielen Consumer-Geräten von Sony eingesetzt. Wesentliche Merkmale sind Echtzeitfähigkeit und Objekt-Orientierung.

In Aperios ist jeder Prozess ein Objekt. Aperios ermöglicht Kommunikation zwischen zwei Prozessen per Messages. Dies sind Nachrichten, die von einem Sender-Objekt an ein Empfänger-Objekt verschickt werden. Messages bestehen aus einer MessageInfo-Struktur, die Informationen über Art und Größe der Message enthält, und den entsprechenden Daten (z.B. Kameradaten).

Wesentlich ist die Aufteilung der Objekte in Sender und Observer. Jedes Objekt muss die folgenden Funktionen (auch „Entry-Points“ genannt) bereitstellen:

- Konstruktion
  - DoInit(): Initialisierung eines Objektes
  - DoStart(): Start des Sendens/Empfangens von Messages
- Destruktion
  - DoStop(): Stop des Sendens/Empfangens von Messages
  - DoDestroy(): Entfernen des Objektes
- Subject-spezifisch (Sender)
  - ControlHandler(): Verbindung herstellen
  - ReadyHandler(): Empfangen von Messages
- Observer-spezifisch (Empfänger)
  - ConnectHandler(): Verbindung herstellen
  - NotifyHandler(): Empfangen von Messages

2.3.2 Open-R

Da Aperios nicht nur für Roboter entwickelt wurde, gibt es noch eine darüberliegende Schnittstelle, die Funktionen für die allgemeine Programmierung von Robotern, die unter Aperios laufen, zur Verfügung stellt. Diese Open-R genannte Middleware-API ermöglicht den Zugriff auf alle Sensoren (Kamera, Taster, usw.) und Aktoren (Gelenke, LEDs, usw.) des Roboters.

Open-R ist eine abstrakte API für verschiedenste Roboter. Man ist theoretisch dazu in der Lage, das Verhalten eines Four-Legged Robot auf einem Roboter laufen zu lassen, der sich auf Rädern fortbewegt.

Die elementaren Bestandteile des Roboters wie Gelenke, die Kamera, die LEDs usw. werden als Primitives bezeichnet.

Um z.B. einen Sensor abzufragen, muss zunächst das entsprechende Primitive geöffnet werden. In der Initialisierung des Prozesses wird mit OpenPrimitive der Zugriff auf einen Sensor aktiviert. Danach besteht die Möglichkeit, per ControlPromitive Einstellungen vorzunehmen (z.B. Weissabgleich bei der Kamera). Die Daten werden dann per Message geschickt und können per GetInfo und GetData empfangen und ausgewertet werden (für Details siehe [2]).

Ein weiteres wichtiges Feature ist, dass man die Open-R Umgebung auch auf dem PC laufen lassen kann, was für das Testen einer Roboter-Simulation sehr von Vorteil ist. Auch der Router, der für die WLAN-Kommunikation verantwortlich ist, macht von dieser Funktion Gebrauch.

2.4 GT2003

GT2003 ist der Name des gesamten Projektes, an dem von der Projektgruppe mitgearbeitet wurde. Da an diesem Projekt insgesamt vier Universitäten teilnehmen, gibt es einen zentralen CVS-Server (Concurrent Versions System, ein System zur Versionskontrolle, das u.a. das gleichzeitige Arbeiten mehrerer Personen an einem Quelltext ermöglicht) in Berlin, auf dem alle Quelldateien abgelegt sind.

Die Struktur des Gesamtprojektes ist so angelegt, dass es für die einzelnen Universitäten möglich ist, eigene Ideen für Teilbereiche des Projektes zu entwickeln, diese separat zu speichern und aufgrund der modularen Struktur von GT2003 gegeneinander zu testen.

Ein weiteres Ziel der Modularisierung von GT2003 ist es, eine Umgebung zu schaffen, in der es möglich ist, den Code sowohl auf dem Roboter, als auch unter Windows in einem Simulator zu testen.

Unter Windows sind dafür zwei Programme, RobotControl und SimGT2003, vorhanden. SimGT2003 bietet die Möglichkeit die Roboter auf einem PC zu simulieren und RobotControl beinhaltet darüberhinaus die Möglichkeit den Roboter zu steuern und sich vom Roboter Debuginformationen schicken zu lassen.

2.4.1 Process-Framework

Auf die von Aperios und Open-R bereitgestelle API ist im GT2003-Projekt eine weitere Schicht aufgesetzt, die es ermöglicht, den Code ohne Änderungen, also nur durch Kompilieren für die entsprechende Plattform, zu nutzen. Der Code ist also für den Simulator in RobotControl und SimGT2003, und für den Roboter selbst gleich [3].

2.4.1.1 Prozesse in GT2003

Prozesse sind in GT2003 als Klassen realisiert, die von der Klasse Process erben. Die Klasse Process enthält dabei den betriebssystemspezifischen Teil. Unter Windows werden die Prozesse als Threads von RobotControl realisiert. Auf dem Roboter sind es Aperios-Prozesse. Die Prozessklasse selber muss die Methode main überschreiben. In dieser kann nun programmiert werden. Am Ende muss noch das Makro MAKE_PROCESS aufgerufen werden, das dafür sorgt, dass die Klasse richtig in das Gesamtsystem eingebaut wird.

```
class Process1 : public Process
{
    public :
        virtual int main()
        {
            cout << "Hello World";
            return (1000);
        }
}
MAKE_PROCESS(Process1);
```

Dieses Beispiel gibt „Hello World“ aus. Mit return(1000) wird festgelegt, dass der Prozess in 1000 ms aufgerufen wird. Bei einem Wert von 0 wird der Prozess gleich noch einmal gestartet. Bei negativen Werten versucht das System den Prozess regelmäßig alle z.B. 1000 Millisekunden zu starten (egal wie hoch die Laufzeit ist). Sollte der Prozess bis dahin schon 1000 Millisekunden gelaufen sein, so wird er gleich noch einmal gestartet.

Da normalerweise mehrere Prozesse gleichzeitig verwendet werden, lassen sie sich zu einem „Process-Layout“ zusammenfassen.

Das aktuell im GT2003-Projekt verwendete Process-Layout besteht aus drei Prozessen, die Cognition, Motion und Debug genannt werden. Dabei ist der Cognition-Process für die Bildverarbeitung (Kapitel 3 auf Seite 22), die Weltmodellierung (Kapitel 5 auf Seite 49 und 6 auf Seite 57) und die Verhaltenssteuerung (Kapitel 7 auf Seite 69) zuständig. Der Motion-Prozess berechnet die Bewegungen des Roboters und der Debug Prozess dient zur Kommunikation mit RobotControl.

Es ist möglich verschiedene Process-Layouts zu definieren um die Module (siehe 2.4.2 auf Seite 13) anders auf die Prozesse zu verteilen. Das zu verwendete Prozess-Layout muss vor dem Compilieren ausgewählt werden.

2.4.1.2 Inter-Process-Kommunikation im Process-Framework

Damit die Prozesse untereinander Daten austauschen können, wird eine Inter-Process-Kommunikation verwendet. Unter Aperios wird dabei die integrierte Inter-Process-Kom-

munikation verwendet, unter Windows wird das Verhalten nachgebildet.

Wenn man nun Daten von einem Prozess zum anderen schicken möchte, so muss dafür zuerst eine Datenpaket-Klasse erstellt werden. Diese Klasse besteht aus einigen Attributen und Streaming-Operatoren, die dafür sorgen, dass diese Attribute verschickt werden.

```
class IntPackage
{
    public:
       int number;
       IntPackage() { number = 0; }
};

Out& operator<<(Out& stream,const IntPackage& p)
  { return stream << p.number; }

In& operator>>(In& stream, IntPackage& p)
  { return stream >> p.number; }
```

Außerdem benötigt man in dem Prozess, der die Daten verschickt, einen Sender für das Datenpaket und im empfangenden Prozess einen Receiver.

Ein Sender beinhaltet eine Instanz der Datenpaket-Klasse und sorgt dafür, dass diese an den Receiver übertragen wird, nachdem die Methode send() aufgerufen wurde. Der Receiver wiederum sorgt nun dafür, dass diese Daten in seiner Instanz der Datenpaket-Klasse ankommen.

Das Empfangen der Daten im empfangenden Prozess läuft dabei gleich vorm Ausführen der main-Methode ab.

```
#include "Tools/Process.h"

class Process1 : public Process {
    private:
          SENDER(IntPackage);
    public:
           Example1() : INIT_SENDER(IntPackage,false) {}
           virtual int main()
           {
              ++theIntPackageSender.number;
              theIntPackageSender.send();
              return 100;
           }
}; MAKE_PROCESS(Process1);
```

Ein Sender für eine Datenpaket-Klasse wird mit dem SENDER Makro angelegt. Auf das Datenpaket kann dann mit theDatenPaketNameSender zugegriffen werden. Durch den Aufruf von theDatenPaketNameSender.send() werden die Daten nun am Ende verschickt.

Der empfangende Prozess sieht dann so aus:

```
#include "Tools/Process.h"
class Process2 : public Process
{
       private:
             RECEIVER(IntPackage);
       public:
             Process2() : INIT_RECEIVER(IntPackage,true) {}
             virtual int main()
             {
                  cout << "Zahl" << theIntPackageReceiver.number);
                  return 0;
             }
};
MAKE_PROCESS(Example2);
```

Hier wird ähnlich wie im sendenden Prozess mit dem RECEIVER Makro ein Receiver für ein Datenpaket angelegt. Auf das Datenpaket kann dann über theDatenPaketNameReceiver zugegriffen werden. Sollte der sendende Prozess keine neuen Daten geschickt haben so liefert der Zugriff auf den Receiver immer die selben Daten zurück. Man kann aber mit theDatenPaketNameReceiver.receivedNew() abfragen, ob neue Daten vorhanden sind.

2.4.2 Modulkonzept

Jede Funktionseinheit im GT2003-Projekt ist als so genanntes Modul realisiert. Für jedes dieser Module können mehrere Möglichkeiten – so genannte Solutions – implementiert werden. Für jedes Modul ist eine Schnittstelle – Interface – zum Rest des Projektes definiert. Daher kann unter allen Solutions, die für ein Modul implementiert sind, eine beliebige ausgewählt werden, die verwendet werden soll. Es ist sogar möglich, zur Laufzeit zwischen verschiedenen Solutions zu wechseln [3].

Welche Solution beim Starten des Roboters ausgewählt wird, entscheidet sich durch die Einträge in einer Datei namens *modules.cfg*.

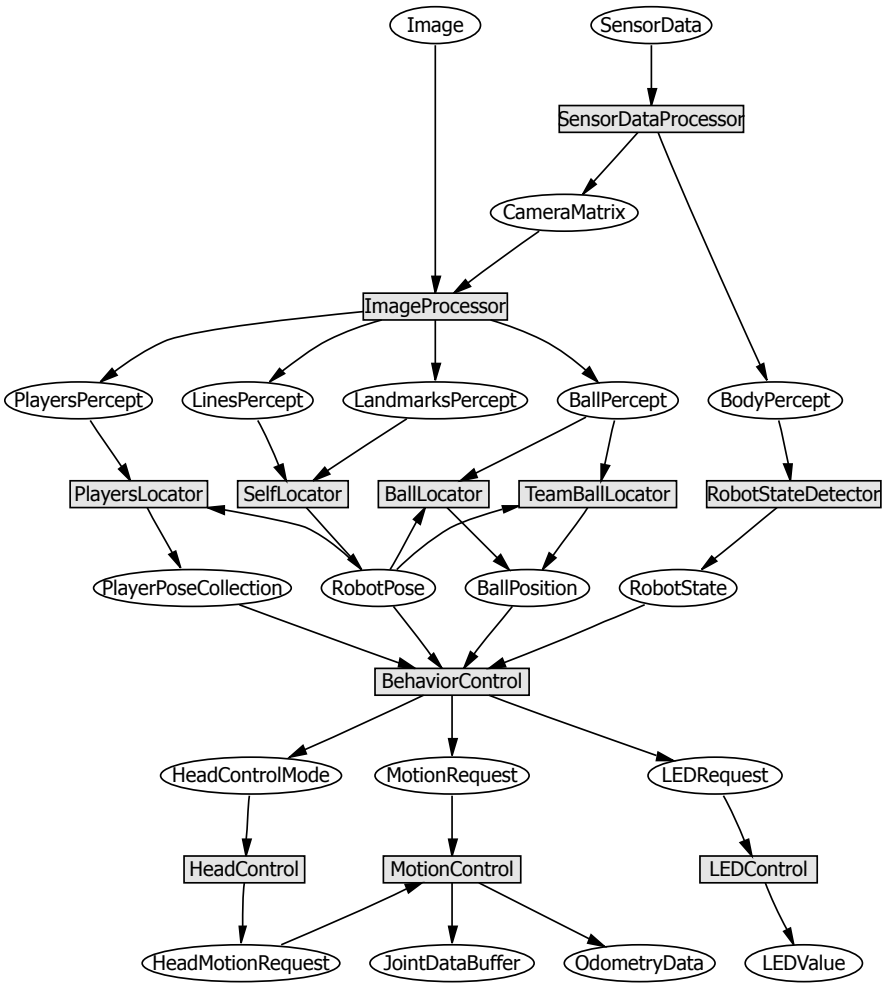

Abbildung 2.3: Module und deren Datenabhängigkeiten in GT2003

In den Interfaces werden Referenzen auf die Attribute und Variablen des Prozesses gesetzt, die von den Solutions des jeweiligen Modules benötigt werden. Bei den meisten Modulen ist dies der Prozess Cognition. So können die Solutions direkt auf den Daten des Prozesses arbeiten.

In Abbildung 2.3 auf der vorherigen Seite sind die wichtigsten Module von GT2003 als Rechtecke zu sehen. Zwischen den einzelnen Modulen bestehen Datenabhängigkeiten. Diese sind in der Abbildung durch Verbindungen zwischen den Modulen angedeutet. Diese Datenabhängigkeiten bedeuten, dass ein Modul Daten (in der Abbildung durch Ellipsen dargestellt) benötigt, die ein vorhergehendes Modul erst berechnet. So benötigt der BallLocator beispielsweise vom ImageProcessor aus dem Kamerabild ermittelte Daten (hier: ein BallPercept), um aus diesen die Position des Balles zu ermitteln.

In jedem Durchlauf durch die Prozesse werden nacheinander unter Beachtung der Datenabhängigkeiten die execute-Methoden der einzelnen Module aufgerufen, wodurch die ausgewählten Solutions ihre jeweilige Aufgabe erfüllen.

2.4.2.1 Die Module im Einzelnen und deren Bedeutung

Wie schon in Abbildung 2.3 auf der vorherigen Seite zu sehen, gibt es eine ganze Anzahl von Modulen. Hier sollen alle Module einmal aufgeführt werden und in ihrer Bedeutung kurz erläutert werden:

ImageProcessor Ermittelt aus dem Bild so genannte Percepts – Positionen von Objekten in relativen Koordinaten.

SensorDataProcessor Ermittelt Daten von den verschiedenen Sensoren des AIBOs. Unter anderem auch die CameraMatrix, die die Lage der Kamera relativ zum Körper des Roboters angibt.

RobotStateDetector Ermittelt Daten über den Zustand des Roboters, bspw. Schwanzposition oder Zustand der Schalter (gedrückt/nicht gedrückt).

SpecialVison Normalerweise nicht verwendete Bildverarbeitung, die spezielle Aufgaben übernimmt (z.B. das Einlesen von Barcodes).

CollisionDetector Überprüft, ob eine Kollision stattgefunden hat.

BallLocator Berechnet aus dem BallPercept eine Ballposition in Feldkoordinaten.

TeamBallLocator Berechnet eine zweite Ballposition aus der im BallLocator berechneten Ballposition und den Ballpositionen, die von den Mitspielern empfangen werden.

PlayersLocator Berechnet aus den PlayerPercepts Spielerpositionen auf dem Feld.

SelfLocator Berechnet die eigene Position auf dem Spielfeld.

ObstaclesLocator Berechnet Positionen von Hindernissen.

BehaviorControl Bestimmt aus den bekannten Daten das auszuführende Verhalten des Roboters.

SensorBehaviorControl Eine normalerweise nicht verwendete Verhaltenssteuerung, die direkt auf den Sensordaten des AIBOs arbeitet.

PerceptBehaviorControl Eine normalerweise nicht verwendete Verhaltenssteuerung, die direkt auf den Percepts anstelle der von den verschiedenen Locatorn gelieferten Positionen arbeitet.

HeadControl Steuert die Bewegungen des Kopfes.

LEDControl Steuert die LEDs des AIBOs an.

MotionControl Steuert die Motoren des AIBOs an, um Bewegungen zu erzeugen.

WalkingEngine Wird als Submodul der MotionControl verwendet und berechnet Gelenkwinkel für die Motoren, die für bestimmte Laufbewegungen benötigt werden.

SpecialActions Wird als Submodul der MotionControl verwendet und berechnet Gelenkwinkel für die Motoren, wenn spezielle Bewegungen – sogenannte SpecialActions – durchgeführt werden.

SoundControl Erzeugt Soundausgabe mit Hilfe der Lautsprecher des AIBOs.

2.4.3 Router

Der Router ist ein Programm, das auf einem PC läuft und die Kommunikation zwischen den Robotern verwaltet. Dieses Programm baut zu jedem Roboter mehrere TCP-Verbindungen auf, über die dann Daten mit anderen Robotern ausgetauscht werden können. Er bietet außerdem die Möglichkeit, eine Verbindung zu RobotControl aufzubauen und erledigt die entsprechenden Protokollanpassungen zwischen RobotControl und den Robotern (siehe Kapitel 4.1.2 auf Seite 39).

Der Router läuft in einer Open-R-Umgebung (siehe Kapitel 2.3.2 auf Seite 10) und schickt die Daten über den TCP-Gateway-Prozess.

Nachdem alle Roboter gebootet sind, werden mit `start.bash [-gm] xxx.xxx.xxx aaa [bbb [ccc [ddd [eee [fff [ggg [hhh]]]]]]]` die Konfigurationsdateien für die Open-R-Umgebung generiert und der Router gestartet. Dieses Script befindet sich im Verzeichnis GT2003/Bin.

„xxx.xxx.xxx“ bezeichnet dabei die Teile der IP-Adresse, der für alle Roboter gleich ist; „aaa“, „bbb“, . . . , „hhh“ den für bis zu 8 Roboter unterschiedlichen Teil. Wird noch der Parameter „-gm“ angegeben, so startet das Skript auch noch den Game Controller, der auch häufig Game Manager genannt wird (Kapitel 2.4.4 auf der nächsten Seite)

`Stop.bash` beendet alle Prozesse, die mit `Start.bash` gestartet worden sind.

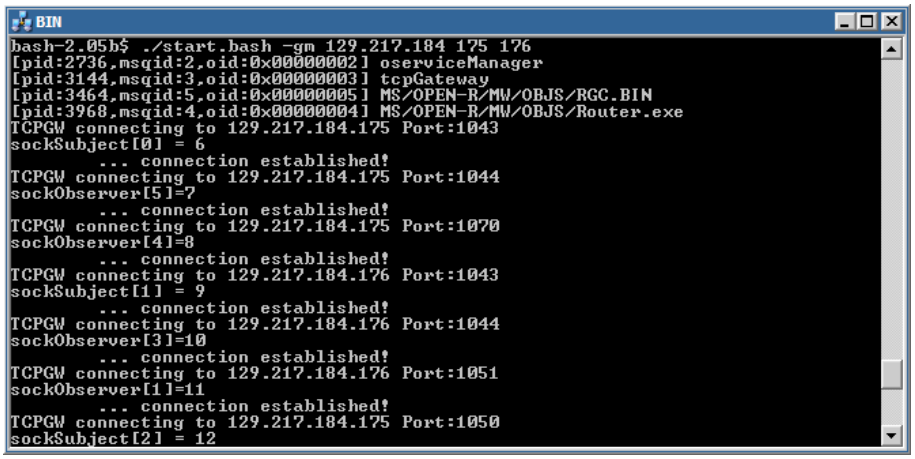

Abbildung 2.4: Beispiel: `start.bash -gm 129.217.184.175 176` startet den Router und den Game-Controller für 2 Roboter, die die Ip-Adressen 129.217.184.175 und 129.217.184.176 haben.

2.4.4 Game Controller

Der Game Controller, auch Game Manager genannt, wird mit dem Skript `start.bash` (siehe Kapitel 2.4.3 auf der vorherigen Seite) gleichzeitig mit dem Router gestartet. Er dient dem Schiedsrichter dazu, bei einem Spiel seine Entscheidungen an die Roboter weiterzugeben und so das Spiel zu leiten.

Während der offiziellen Spiele auf den verschiedenen Wettkämpfen wird auch dieser Game Controller verwendet. Dort läuft er allerdings unter Linux und wird auf eine andere Art und Weise gestartet.

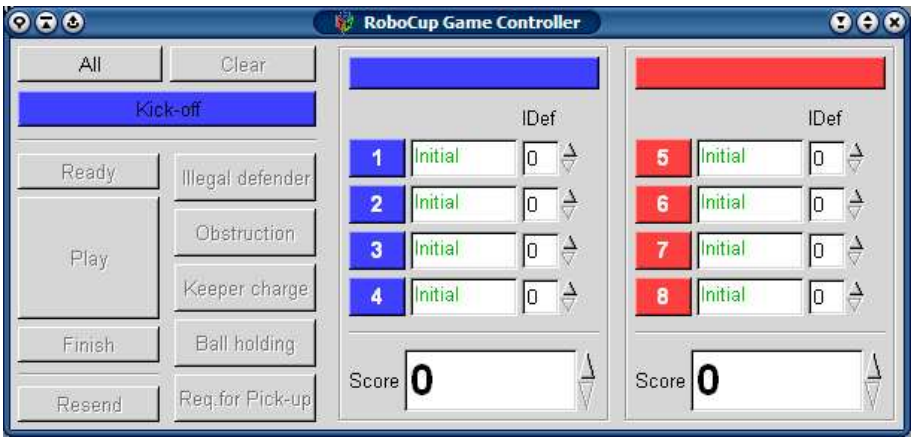

Abbildung 2.5: Der Game Controller dient dem Schiedsrichter dazu, den Robotern seine Entscheidungen mitzuteilen

Nach dem Start des Game Manager (siehe Abbildung 2.5) kann man über verschiedene Schaltflächen in das laufende Verhalten der AIBOs eingreifen.

Im rechten und mittlerem Bereich des Game Managers ist eine Übersicht über die maximal acht AIBOs, die an einem Spiel teilnehmen dürfen. Über die Ziffern-Schaltflächen oder die

Tasten 1 bis 8 kann ein AIBO angewählt werden. Durch Drücken der „All“-Schaltfläche werden alle Roboter ausgewählt und mit „Clear“ wird die Auswahl rückgängig gemacht.

Die Reihenfolge der AIBOs ist die Reihenfolge, in der die IP-Adressen beim Aufruf der `start.bash` definiert wurden. Sollte die Zuordnung der Teamfarben nicht stimmen, so kann man diese durch Anklicken des Teamfarben-Balkens über der Roboterliste ändern.

Das zweite Feld zu jedem AIBO gibt dessen aktuellen Status an und das letzte Feld zeigt an, wie oft er gefoult hat.

Mit Kick-off wird die Mannschaft ausgewählt, die Anstoß hat. Ready schickt die Roboter an ihre Startposition, Play startet das Spiel und Finish sagt den Robotern, dass sie fertig sind.

Hat ein Roboter ein Foul begangen, so kann man ihm dies mitteilen, indem man ihn zuerst auswählt und dann auf den Button für das entsprechende Foul (Illegal defender, Obstruction, Keeper charge und Ball holding) klickt. Nach Ablauf der für das Foul vorgesehenen Strafzeit wird die Ziffern-Schaltfläche des entsprechenden Roboters grün und er kann nach Auswahl mit dem „Playing“-Button wieder ins Spiel geschickt werden.

Hat ein Roboter ein Problem, so kann ein Verantwortlicher des entsprechenden Teams diesen z.B. zum Neustart aus dem Spiel nehmen. Dieser „Request-for-Pickup“ wird vom Game Controller intern wie ein Foul behandelt, da der AIBO frühestens nach 30 Sekunden ins Spiel zurückgestellt werden darf.

Erzielt eine Mannschaft ein Tor so kann man im Feld „Score“ die Punktzahl erhöhen. Ein anschließendes „Kick-Off“, „Ready“ und „Playing“ sorgt dann dafür, dass es weiter geht.

2.4.5 RobotControl

RobotControl dient in erster Linie zur Debug-Kommunikation mit den Robotern, also eine Visualisierung von Variablenbelgungen der Programmteile, die auf dem Roboter laufen. Man kann über den Router zu den einzelnen Hunden, auch mehreren gleichzeitig, eine Verbindung aufbauen. Außerdem ist noch ein Simulator integriert, mit dem man einzelne Module und vor allem das Verhalten testen kann. Allerdings ist keine Physik-Engine integriert, so dass keine Schüsse oder Kollisionen von Robotern simuliert werden können. Ferner gibt es noch einen primitiven GameManager, mit der Möglichkeit die Roboter zur Startaufstellung zu schicken sowie das Spiel zu starten und zu beenden.

2.4.5.1 Verbindung zum Roboter

Nachdem eine Verbindung zwischen Router und Roboter zustande gekommen ist, wobei der Router auf dem gleichen Rechner wie RobotControl gestartet sein sollte, kann man eine Verbindung zum Roboter mittels der IP-Adresse einrichten. Falls der Router nicht auf dem gleichen Rechner wie RobotControl gestartet wurde, muss bei der Konfiguration der Verbindung die IP-Adresse des Rechners, auf dem der Router läuft, angegeben werden.

2.4.5.2 Möglichkeiten in RobotControl

Nachdem eine Verbindung zustande gekommen ist, kann man sich verschiedene Daten vom Roboter in wählbaren Intervallen schicken lassen.

- Informationen vom Roboter:
  Die wichtigsten Möglichkeiten sind folgende:
  - sendImage: Hier kann man sich das aktuelle Bild der Kamera zusenden lassen. Dies ist nötig, um eine Farbtabelle zu erstellen, mit der die segmentierten Bilder erstellt werden können.
  - sendWorldState: Der Roboter schickt Informationen über alle Positionen die er kennt, also die eigene Position, die der eigenen und gegnerischen Mitspieler sowie die des Balls. So kann man die Qualität der Selbstlokalisierung und der Balllokalisierung abschätzen.
  - sendXABSL2DebugMessage: Dies beinhaltet den aktuellen Zustand des XABSL-Automaten sowie dessen Eingangs- und Ausgangssymbole.
  - send_hexAreasPotentialfieldDrawing: Der Roboter schickt Höheninformationen des Potenzialfelds.
  - send_strategicalDatabseMove: Diese Information dient zur Anzeige, welche Strategie zur aktuellen Spielsituation passt und welche Aktionen deshalb ausgeführt werden sollten

  Die Abbildung 2.6 auf der nächsten Seite zeigt das Spielfeld, wenn man sich Informationen per sendWorldState und send_hexAreasPotentialfieldDrwaing senden lässt. Die Rechtecke stellen die Roboter dar, die Kreise das Potenzialfeld, wobei rot auf ein hohes Potential hindeutet. Der Punkt in der Mitte entspricht der Ballposition.

- Anzeige- und Einstellungsmöglichkeiten:
  Auf dem dargestellten Spielfeld lassen sich die Informationen, welche man sich wie oben beschrieben senden lässt, anzeigen. Man muss jedoch zuvor mit einem rechten Mausklick auf das Spielfeld die entsprechenden Optionen aktivieren. Dies betrifft insbesondere WorldState, Strategical Database Move und hexAreaPotentialfield.

  Außer dem Spielfeld kann man noch im „VIEW“-Menü (siehe 2.7 auf Seite 20) Anzeige- und Einstellungsfenster aufrufen. Es gibt folgende Einstellungsfenster:
  - Setting: Hier lassen sich für alle Module Solutions auswählen beziehungsweise ausstellen.
  - ColorTable64Tool: Aus den empfangenen Bildern lässt sich eine Farbtabelle zur Bildsegmentierung erstellen. Mit diesem Tool kann sie erstellt, abgespeichert und direkt zum Roboter gesendet werden.
  - TSLColorTableTool: Dies arbeitet ähnlich wie das ColorTable64Tool, wurde aber auf den neu entwickelten Farbraum angepasst.

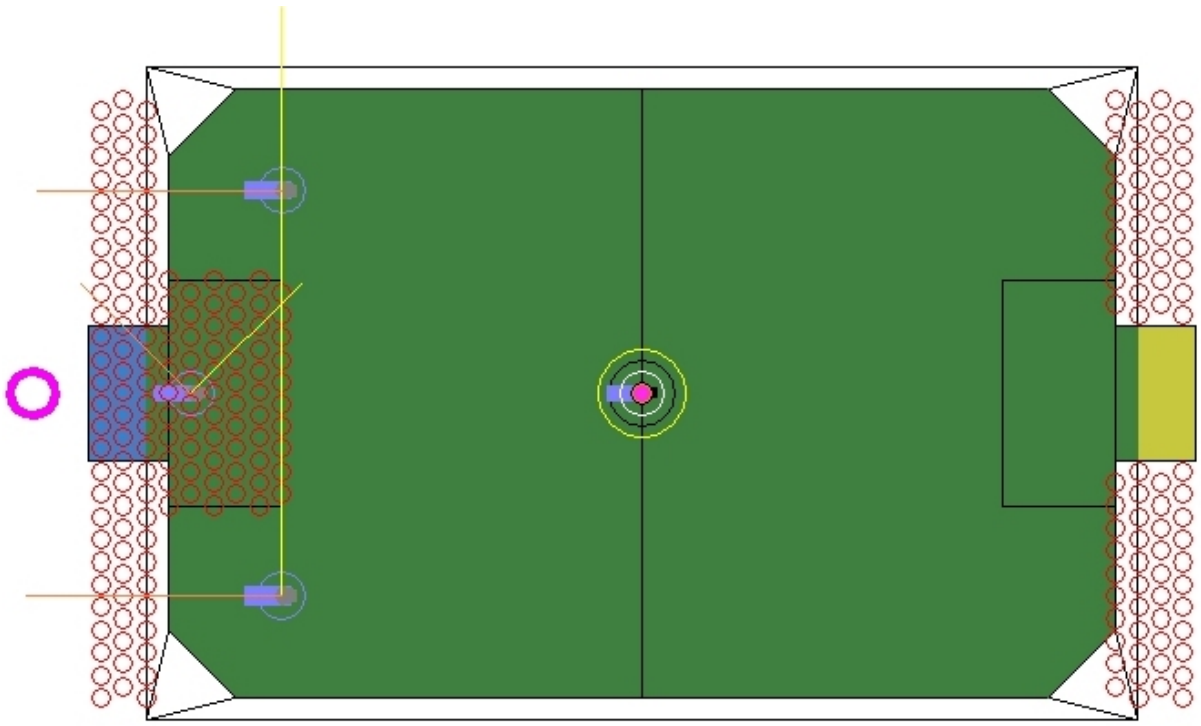

Abbildung 2.6: Spielfeld mit den eingeschalteten Debugdrawings WorldState und hexAreaPotentialfieldDrawing

- Xabsl2BehaviorTester: Mit diesem Tool lassen sich die Informationen über den XABSL-Automaten anzeigen, welche man sich per sendXABSL2DebugMessage schicken lassen kann.
- JoystickMotionTester: Wenn man zuvor das Verhalten ausschaltet, kann man den Roboter über einen angeschlossenen Joystick steuern.

Folgende Anzeigefenster spielen eine besonders wichtige Rolle:

- ImageViewer: Der ImageViewer beinhaltet mehrere Unterfenster, die nach Bedarf konfiguriert werden können. Man kann sich zum Beispiel das aktuelle Bild, das segmentierte Bild, den Horizont und noch weitere für die Bildverarbeitung wichtigen Informationen anzeigen lassen.
- MessageViewer: Hier können Ausgaben des Codes über ein OUTPUT Macro als reiner Text angezeigt werden. Dies ist zum Debugging nützlich.
- Time Diagram Dialog: Dies öffnet ein Fenster mit Balkendiagrammen, welche die Programmlaufzeit der einzelnen Module darstellt. Welche Laufzeiten angezeigt werden sollen, lässt sich ebenfalls konfigurieren. Ineffiziente Module lassen sich so leicht erkennen.

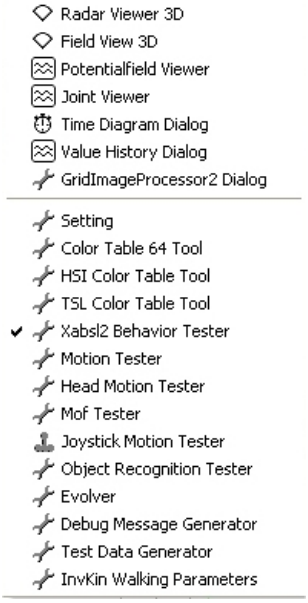

Abbildung 2.7: Das „VIEW“-Menü von RobotControl

2.4.6 SimGT2003

Mit dem Simulator kann ein komplettes Spiel oder nur ein einzelner Roboter simuliert werden. Die simulierten Roboter erhalten simuliert Cognition-Daten. Allerdings existiert kein physikalisches Modell des Hundes oder des Balles. So können keine Objekte zusammenstoßen. Die Roboter laufen also berührungsfrei durch einander hindurch. Sie sind auch nicht in der Lage, den Ball zu bewegen. Dies muss manuell nachgeführt werden, was bedeutet, dass der Ball mit dem Mauszeiger angeklickt und an die Zielposition gezogen wird. Es wird ein Spielschema als .scn-Datei ausgewählt. Hier ist vermerkt, welcher Roboter unter welchen Umständen simuliert wird. Es können einzelne Spielschritte simuliert werden. Es kann auch die Simulation gestartet werden. Bis zum Beenden laufen dann beliebig viele Schritte hintereinander ab. Während die Simulation läuft, können einzelne Objekte, wie der Ball und jeder Roboter, mit der Maus bewegt werden. An den Vorderbeinen wird der Roboter gedreht, an den Hinterbeinen verschoben.

Der Simulator arbeitet kommandozeilengesteuert. Einzelne Befehle sind in der Kommandozeilenbox, einem Textboxfenster, einzugeben. Sämtliche Textausgaben, wie auch Textdebugausgaben werden hier ausgegeben. Hierbei ist zu beachten, dass bei Eingaben aus der Kommandozeile jede einzelne Zeile noch mit der Eingabetaste bestätigt werden muss. Die Textbox hat nur eine begrenzte Größe. Bei vollständiger Füllung muss, um weitere Ausgaben angezeigt zu bekommen, Text markiert und gelöscht werden. Es können auch Stapelverarbeitungen abgearbeitet werden. Diese müssen als .con-Dateien vorliegen. Die wichtigsten Befehle sind:

- ? : Es werden alle möglichen Befehle aufgelistet.
- dk ? | <key> off | on | <number> | every <number> [ms]: Es wird ein Debug-Schlüssel gesetzt.

- sr ? | <module> (? | <solution>) : Es wird ein Solutionrequest gesetzt. Dies ist die Auswahl eines Moduls.
- msg off | on : Text-Nachrichten werden aus- oder eingeschaltet.

Debugdrawings können jeweils in einem eigenen Fenster dargestellt werden. Die Auswahl findet über den Ausgabebaum statt.

Kapitel 3

Bildverarbeitung

Die Bildverarbeitung ist ein zentrales Modul von GT2003. Sie ist dafür verantwortlich, dass der AIBO „verstehen kann, was er sieht“. Als Auge dient das Objektiv einer Videokamera, die in der Schnauze des AIBO eingebaut ist. Die Kamera liefert ein 176x144 Pixel großes Bild im YUV-Format (siehe Abschnitt 3.2.1 und Abbildung 3.1) mit einer Farbtiefe von 24 Bit. Das Bild wird dem Process Framework von Open-R zur Verfügung gestellt.

Die Anforderungen an die Bildverarbeitung sind sehr hoch, da sie trotz geringer Prozessorleistung in Echtzeit die Anzahl, Position, Art, Entfernung und Richtung (relativ zur Eigenrichtung) der auf dem Spielfeld erkannten Objekte bestimmen muss. Zwar sind die Umgebungs- und Spielbedingungen für einen AIBO nicht sehr kompliziert, da alle Objekte eindeutige Signalfarben haben – Orange für den Ball, Gelb und Blau für die Tore usw. – aber die schwache Auflösung der Kamera und die relativ geringe Prozessorleistung lassen den Programmierer rasch an die Leistungsgrenzen des Systems stoßen.

3.1 Verfahren

In der Bildverarbeitung kamen drei Verfahren mit unterschiedlichen Aufgaben zur Anwendung:

- Segmentierung des Bildes: Klasse ColorTable64 bzw. ColorTableTSL
- Objekterkennung: Modul GridImageProcessor
- Bereitstellen der Weltkoordinaten: Klasse PerceptCollection

Diese Verfahren werden im folgenden genauer beschrieben. Dabei gliedert sich dieses Kapitel systematisch in die Kategorien Ist-Stand/Definieren von Zielen/Entwicklung und Implementierung/Evaluation der Ergebnisse.

3.2 Segmentieren

Das Segmentieren dient dazu, die YUV-Eingangsdaten zunächst nach Farben zu klassifizieren und anschließend in gleichmäßige Bereiche aufzuteilen. Die so entstandene Segmentierung bildet dann die Grundlage für eine weitere Verarbeitung, z.B. im GridImageProcessor. Zu Beginn der Projektgruppe existierte für die Farbklassifizierung nur ColorTable64 (siehe Abschnitt 3.2.2) mit dem dazu gehörigen ColorTable64Tool (siehe Abschnitt 3.2.3). Da diese mit einigen Nachteilen, auf die in Abschnitt 3.2.4 näher eingegangen wird, verbunden waren, wurden ColorTableTSL (siehe Abschnitt 3.2.5) und das dazugehörige TSLColorTableTool (siehe Abschnitt 3.2.6) von der Projektgruppe entwickelt.

Für das Fußballspiel in der Sony Legged League gibt es folgende Farben, anhand derer relevante Objekte erkannt werden können (siehe Tabelle 3.1):

Farbe	Objekte
Orange	Ball
Grün	Spielfeld und Landmarken
Pink	Landmarken
Gelb, Hellblau	Tore und Landmarken
Rot, Blau	Trikots der Roboter
Weiß	Bande, Landmarken und Linien
Schwarz	Schwarz-weißer Ball (siehe Kapitel 8 auf Seite 103), Spieler
Undefiniert	andere, nicht relevante Objekte

Tabelle 3.1: Relevante Farben

3.2.1 Das YUV-Farbmodell

Die Kamera liefert Bilder im YUV-Format, welches im Video-Bereich ein Standard ist. Der Y-Kanal enthält Helligkeitsinformationen (auch Luminanz genannt), also das, was auf einem Schwarz-Weiß-Fernseher darstellt wird. U und V sind Chrominanz-Kanäle, wobei U die Blauanteile und V die Rotanteile enthält. Manchmal werden U und V auch Cb und Cr genannt. Jeder Kanal wird mit 8 Bits pro Pixel abgetastet, so dass eine Farbtiefe von insgesamt 24 Bits pro Pixel entsteht. Abbildung 3.1 auf der nächsten Seite zeigt die UV-Farbebene bei drei verschiedenen Helligkeiten.

3.2.2 ColorTable64

ColorTable64 arbeitet mit einer so genannten Look-Up Table. Die ColorTable64 ist ein dreidimensionales Array, welches einem YUV-Tripel eine Farbklasse zuweist. Da das Array aus Speichergründen nicht 2^{24} Bytes (24 Bit Farbtiefe) groß sein kann (es würde dann 16 MB verschlingen), wurde jeder Farbkanal auf 6 Bits verkürzt, so dass ein 2^{18} Bytes (18 Bit Farbtiefe) großes Array entsteht. Daher auch der Name ColorTable64, da jeder Kanal $2^6 = 64$ Werte annehmen kann.

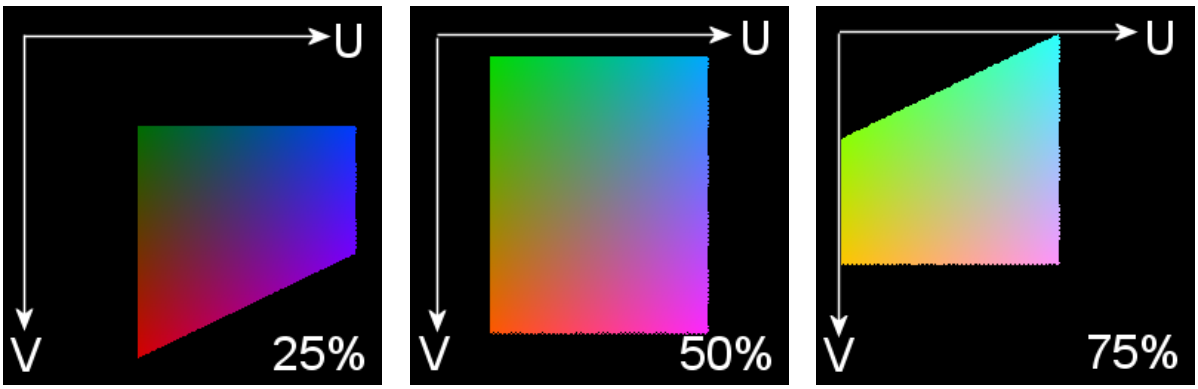

Abbildung 3.1: Die UV-Farbebene bei 25%, 50% und 75% Y-Anteil

3.2.3 ColorTable64Tool

Um eine Farbtabelle anzulegen bzw. zu editieren, gibt es innerhalb von RobotControl einen Dialog namens ColorTable64Tool (siehe Abbildung 3.2).

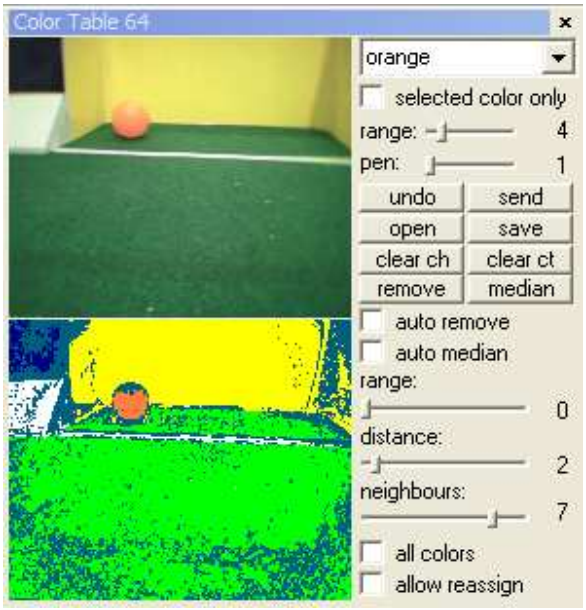

Abbildung 3.2: ColorTable64Tool

Es werden das aktuelle Kamerabild sowie das segmentierte Bild angezeigt. Es kann dann ein Bildpunkt ausgewählt und einer bestimmten Farbklasse (siehe Tabelle 3.1 auf der vorherigen Seite) zugeordnet werden. Die gewählte Farbklasse für diesen Bildpunkt wird in der Farbtabelle gespeichert. Es muss für jede relevante Farbe überprüft werden, ob sie der korrekten Farbklasse zugeordnet wird. Zur Beurteilung müssen in der Regel mehrere Bilder (beispielsweise aus zuvor aufgezeichneten Log-Dateien) herangezogen werden, da es auch innerhalb des Spielfeldes unterschiedliche Lichtverhältnisse gibt. Wurde auf diese Weise eine komplette Farbtabelle erstellt, so kann sie abgespeichert und/oder zum Roboter übertragen werden. Bereits früher erstellte Farbtabellen lassen sich laden und editieren.

3.2.4 Probleme von ColorTable64 und ColorTable64Tool

Beim Einsatz in der Praxis wurden von der Projektgruppe einige Probleme in Zusammenhang mit ColorTable64 bzw. ColorTable64Tool erkannt, die im folgenden näher beschrieben werden:

3.2.4.1 Hoher manueller Aufwand

Es ist sehr mühsam, mit dem ColorTable64Tool die Farbklassen einzustellen, da für jede Farbklasse mehrere Bilder betrachtet werden müssen. Oft ist es erforderlich, umfangreiche Log-Dateien zu erstellen, die alle relevanten Farben aus verschiedenen Blickwinkeln und unter verschiedenen Lichtverhältnissen enthalten. Diese Log-Dateien müssen dann von Hand Bild für Bild durchgearbeitet werden.

3.2.4.2 Zeitaufwand

Das Erstellen einer Farbtabelle erfordert einen hohen Zeitaufwand. In der Praxis hat sich gezeigt, dass die Erstellung einer guten Farbtabelle mit ColorTable64 durchaus länger als 30 Minuten dauern kann.

3.2.4.3 Geringe Robustheit bei wechselnden Lichtverhältnissen

Beim Erstellen der Farbtabellen mit ColorTable64Tool werden nur die Farben der Tabelle hinzugefügt, die auch im Bild vorkommen. Ähnliche Farben werden nicht hinzugefügt, auch wenn die Abweichung sehr gering ist. Wird mit ColorTable64Tool eine Farbtabelle erstellt und ändern sich kurz darauf die Lichtverhältnisse auf dem Spielfeld, so werden meist wesentliche Farben wie z.B. Orange nicht mehr richtig erkannt, da die Farbe nun in der Regel nicht in der Farbtabelle steht und als „undefiniert“ interpretiert wird. Abhilfe schafft hier nur das langwierige Editieren der Farbtabelle unter Verwendung von Log-Dateien mit unterschiedlichen Lichtverhältnissen.

3.2.4.4 Ungenauigkeit

Ein weiteres Problem von ColorTable64 ist die Ungenauigkeit, die bei der Konvertierung von 8 Bits pro Pixel auf 6 Bits pro Pixel entsteht. Dies ist ein generelles Problem des Lookup-Table-Ansatzes, da nicht genügend Speicher auf den AIBOs zur Verfügung steht.

3.2.5 ColorTableTSL

Da insbesondere die Schwankungen der Lichtverhältnisse und die damit verbundene Änderung der Farbtemperatur für ColorTable64 ein Problem darstellen, wurde überlegt, eine Transformation von YUV in einen anderen Farbraum durchzuführen, der robuster

gegenüber derartigen Veränderungen ist. Außerdem sollte dieser neue Farbraum das Klassifizieren von Farben mit Hilfe von Schwellenwerten erleichtern. Idealerweise sollten mit Hilfe der Schwellenwerte die Farben unabhängig von ihrer Helligkeit richtig klassifiziert werden.

3.2.5.1 Das TSL-Farbmodell

Als Grundlage dient der in [4, 5] beschriebene TSL-Farbraum, der die gewünschten Eigenschaften besitzt. TSL steht für Tint, Saturation, Luminance; also Farbton, Sättigung und Luminanz. Er ist verwandt mit HSL (Hue, Saturation, Luminance) und wurde ursprünglich für die Erkennung von Gesichtern entwickelt, wobei besonders die Unempfindlichkeit gegenüber dem Umgebungslicht im Vordergrund stand.

3.2.5.2 Das modifizierte TSL-Farbmodell

Da die ursprüngliche Formel für TSL bei der Sättigung keine Blauwerte berücksichtigt, diese aber für die Bildverarbeitung der Roboter relevant sind, musste die Formel für die Konvertierung von YUV nach TSL für S entsprechend angepasst werden (siehe Abbildung 3.3).

Um die TSL-Formel für die relevanten Farben zu optimieren, wurde ein (1 + 1) Evolutionärer Algorithmus [6, 7], implementiert, der ausgehend von Bildausschnitten mit den neun Farben (siehe Tabelle 3.1 auf Seite 23) optimale TSL-Formeln erzeugt [8]. Hierzu wurde die ursprüngliche Transformation faktorisiert.

Die Formel mit den evolvierten Parametern für die Transformation von YUV nach TSL befindet sich in Abbildung 3.3.

$$
\begin{aligned}
norm &= 4.3403 \cdot y + 2 \cdot u + v \\
r &= (-0.6697 \cdot u + 1.6959 \cdot v)/norm \\
g &= (-1.168 \cdot u - 1.3626 \cdot v)/norm \\
b &= (1.8613 \cdot u - 0.331 \cdot v)/norm \\
t &= \begin{cases} \arctan(r/g)/2\pi + \frac{1}{4}, & g > 0 \\ \arctan(r/g)/2\pi + \frac{3}{4}, & g < 0 \\ 0, & g = 0 \end{cases} \\
s &= \sqrt{1.8 \cdot (r^2 + g^2 + b^2)} \\
l &= y
\end{aligned}
$$

Abbildung 3.3: Formel für die Transformation von YUV nach TSL

Abbildung 3.4 auf der nächsten Seite zeigt die TS-Farbebene bei drei verschiedenen Helligkeiten.

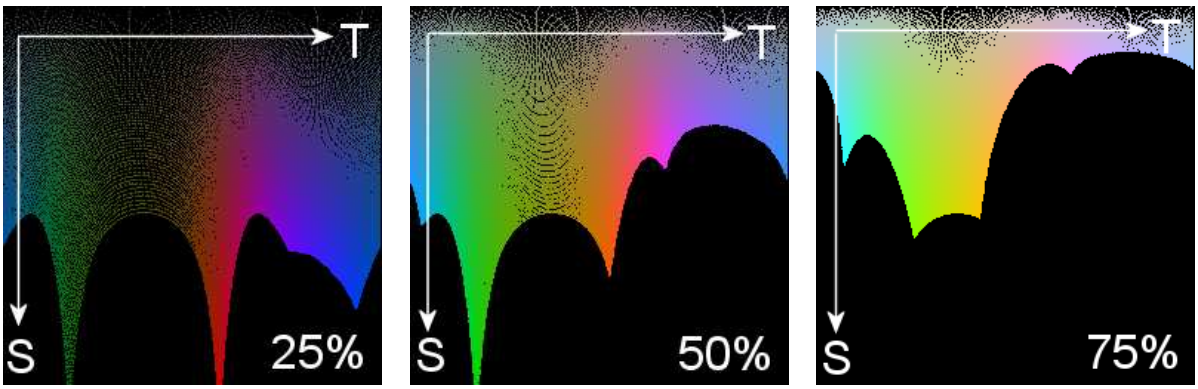

Abbildung 3.4: Die TS-Farbebene bei 25%, 50% und 75% L-Anteil

3.2.5.3 Klassifizierung anhand von Schwellenwerten

Ein Vorteil von TSL gegenüber YUV ist die Möglichkeit des Einsatzes von einfachen Schwellenwerten, um Farbklassen voneinander abzugrenzen.

Im YUV-Farbraum ist dies nicht ohne weiteres möglich, da die Farben sich in der UV-Ebene nicht durch Rechtecke voneinander separieren lassen (siehe die beispielhafte Abbildung 3.5 links und Abbildung 3.1 auf Seite 24).

Im TSL-Farbraum hingegen lassen sich die Farben leicht durch Rechtecke in der TS-Ebene abdecken, da T die Farbe und S die Farbsättigung angibt (siehe die exemplarische Abbildung 3.5 rechts und Abbildung 3.4).

Der Vorteil der Verwendung von Schwellenwerten ist die einfache Implementierbarkeit, die geringe Laufzeit (nur wenige **if-then**-Abfragen werden benötigt) und der minimale Speicherverbrauch (nur die Schwellenwerte selbst müssen gespeichert werden).

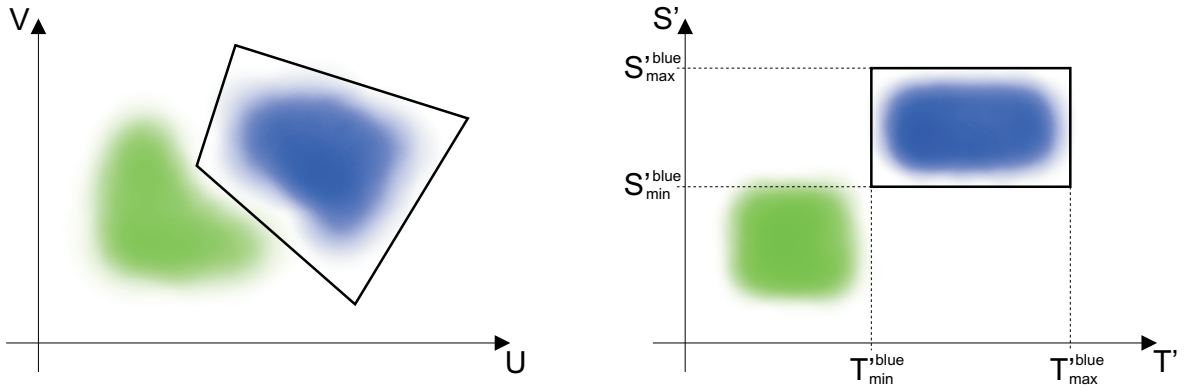

Abbildung 3.5: Ein zweidimensionaler Farbraum (links) wird in einen anderen Farbraum transformiert. Die relevanten Farben (hier exemplarisch blau und grün) werden durch Rechtecke abgedeckt bzw. voneinander abgegrenzt (rechts)

3.2.5.4 Implementierung von TSL in GT2003

Es wurden zwei verschiedene Varianten von ColorTableTSL implementiert. Die eine arbeitet mit einer zu ColorTable64 kompatiblen Look-Up Table und ist daher schnell, aber durch die Rundung von 8 auf 6 Bits pro Farbkanal etwas ungenau. Die andere berechnet die TSL-Werte ausgehend von oben stehender Formel für jedes Pixel einzeln und klassifiziert dann anhand der Schwellenwerte. Dieser Ansatz ist genauer, aber dafür auch langsamer. Im Kapitel 3.4 auf Seite 35 befindet sich ein Vergleich der Laufzeiten dieser beiden Varianten beim Einsatz von unterschiedlichen Bildverarbeitungsmodulen.

3.2.6 TSLColorTableTool

Das TSLColorTableTool dient – analog zum ColorTable64Tool bei Verwendung von Colortable64 – zur Einstellung der Farbtabelle (siehe Abbildung 3.6).

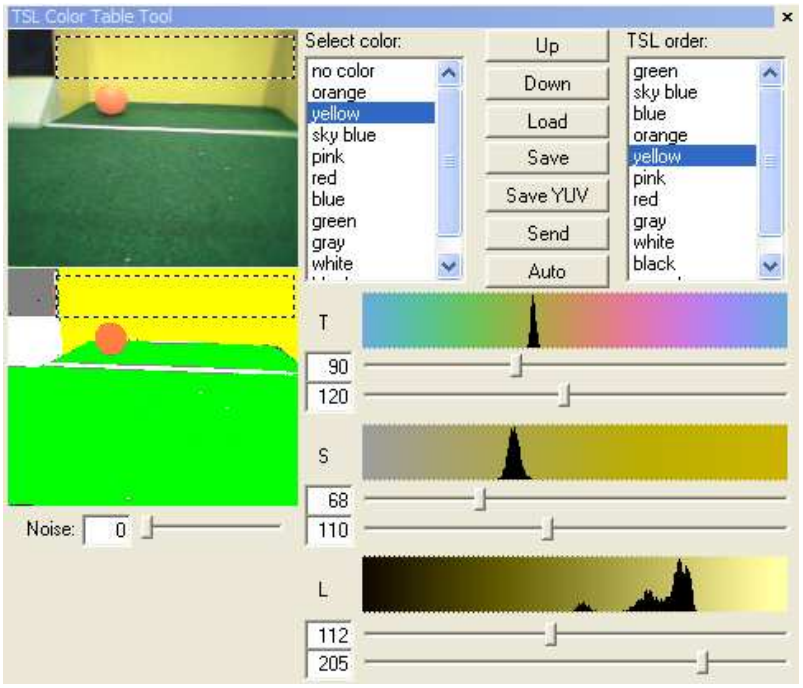

Abbildung 3.6: Das TSLColorTableTool

Ein wesentlicher Unterschied zum ColorTable64Tool ist dabei, dass die Farbklassen per Schwellenwert editiert werden können, d.h. es genügen sechs Parameter (jeweils Maximum und Minimum von T, S und L) zur Zuordnung in eine bestimmte Farbklasse.

Da sich die Farbklassen auch überlappen dürfen, ist die Reihenfolge der Klassifizierung von Bedeutung, die in einer Liste (oben rechts in Abbildung 3.6) editiert werden kann.

Darüber hinaus bietet das TSLColorTableTool einige weitere nützliche Funktionen, so kann z.B. eine bestimmte Region im Bild ausgewählt werden.

Von dem Bild bzw. der ausgewählten Bildregion werden für die aktuelle Farbe Histogramme erstellt, die den Anwender bei der Auswahl der Minima und Maxima unterstützen.

Alternativ kann die Wahl der Schwellenwerte für die aktuelle Farbe auch automatisch vorgenommen werden, dazu sollte sich das entsprechende Objekt in einer spezifizierten Bildregion befinden.

Außerdem ist der Hintergrund der Histogramme mit dem entsprechenden Farbton-, Sättigungs-, und Helligkeitsverlauf gefüllt, was ein intuitives Arbeiten ermöglicht.

Über einen Schieberegler lässt sich das Eingangsbild künstlich verrauschen. Als Rauschgenerator dienen dabei Zufallszahlen, die über eine Gauss-Normalverteilung normiert werden. Diese Funktion ist insbesondere dann sinnvoll, wenn man sich statt Kamera-Bildern die Bilder aus dem Simulator schicken lässt.

Die Farbtabellen-Dateien bestehen aus den sechs Schwellenwerten pro Farbe und der Reihenfolge der Farben. Infolgedessen sind sie sehr kompakt und lassen sich schnell auf die Roboter übertragen.

3.3 Verarbeiten

Die Objekterkennung wird von dem Softwaremodul GridImageProcessor geleistet. Der GridImageProcessor ist ein Hauptmodul der kognitiven Fähigkeiten des Roboters. Er verarbeitet das Kamerabild und generiert so genannte Percepts, die das Behavior verarbeiten kann, um die Art der Objekte auf dem Spielfeld, deren Position, sowie ggf. deren Entfernung und Richtung zum Ball abzufragen.

Ein intuitiver Ansatz wäre es das Bild Pixel für Pixel durchzugehen und zu analysieren. Dies bedeutet allerdings eine zu hohe Last für den Prozessor. Deshalb wird ein Raster bzw. ein Gitter über das Bild gelegt und somit nur ein Teil des Bildes farbklassifiziert. Werden relevante Bildpunkte gefunden, übernehmen so genannte Spezialisten eine nähere Analyse des Bildes. Der Vorteil ist, dass die Spezialisten nur einen kleinen Teil des Bildes exakt untersuchen müssen.

Die Aufgaben, die bereits erworbenen Fähigkeiten sowie die Ziele für diese Spezialisten werden nun im einzelnen behandelt.

3.3.1 BallSpecialist

Der BallSpecialist wird aufgerufen, wenn der GridImageProcessor mehrere orangefarbene Pixel findet. Von dieser Stelle aus geht er horizontal, vertikal und diagonal in alle acht Richtungen, bis er einen Farbsprung von der Farbe Orange bis zu einer anderen Farbe erkennt. Mittels der so gefunden Randpunkte berechnet er aus mindestens 3 Punkten den zum Ball gehörigen Kreis und generiert später das BallPercept.

3.3.1.1 Probleme des BallSpecialist

Eines der Hauptprobleme des BallSpecialist war, dass er oft Bälle an Stellen erkannte, an denen sich tatsächlich kein Ball befand. Aufgrund der schlechten Bildqualität der

Kamera kam es zum Beispiel regelmäßig vor, dass an den Grenzlinien zwischen einem rotem Roboter vor einem gelben Tor orangefarbene Pixel identifiziert wurden. Dieser Effekt verstärkte sich, wenn sich der beobachtende Roboter schnell bewegte und die Bilder verschwommen aufgenommen wurden. Aufgrund der „SensorFusion“ (siehe Kapitel 5 auf Seite 49) wurde eine falsche Ballposition im „worst case“ an alle Roboter übertragen.

3.3.1.2 Ziel und Lösung

Unter Rückgriff auf den vorhandenen Spezialisten für die Ballerkennung ist zwischen die Erkennung des Balls und die Generierung des BallPercepts das so genannte „Probing“ geschaltet worden. Der Probing-Algorithmus wird aufgerufen, nachdem der den Ball repräsentierende Kreis mit den Koordinaten des Mittelpunkts und dem Radius berechnet wurde, aber noch bevor das BallPercept versendet wird. Dabei wird geprüft, ob sich innerhalb des Kreises eine definierte Anzahl von Pixel mit der Farbe Orange (zur Abgrenzung von gelben Pixeln) befindet. Fällt dieser nachträgliche Test positiv aus, haben z.B. mehr als 50 Prozent der in einer zufälligen Streuung geprüften Pixel die Farbe Orange, wird das Percept gesendet, ansonsten wird es verworfen.

3.3.1.3 Fazit

Dieser Algorithmus hat sich in der Simulation und in der Praxis als sehr effektiv erwiesen. Man konnte während der Spiele deutlich erkennen, dass die Roboter kaum noch falsche Bälle im gelben Tor gesehen haben. In der Abbildung 3.7 kann man einmal ein oranges BallPercept sehen, welches in die PerceptCollection übernommen wird und ein pinkes BallPercept, welches verworfen wird. Die gelben Pixel stellen die „Probes“ dar.

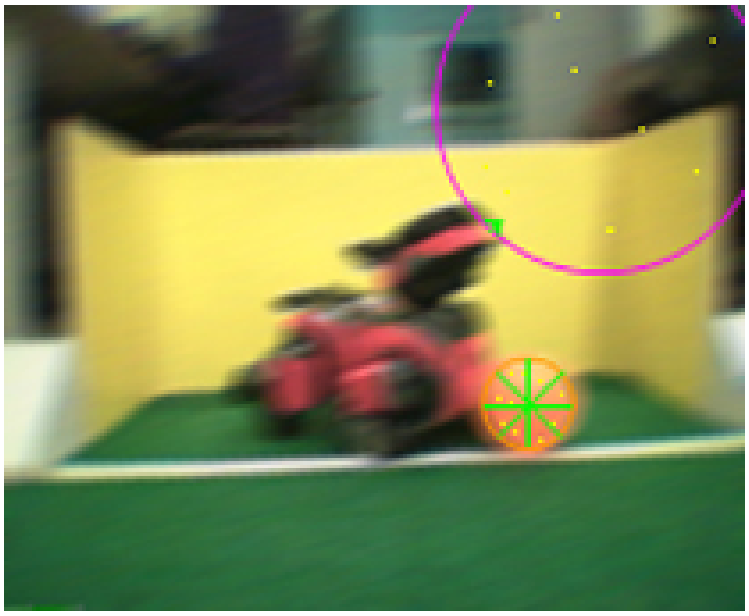

Abbildung 3.7: Screenshot aus einer Log-Datei

3.3.2 PlayerSpecialist

Der PlayerSpecialist wird aufgerufen, wenn – je nach Teamfarbe – rote bzw. blaue Pixel gefunden werden. Dieser Spezialist war zu Beginn der Projektgruppe nicht vollständig und fehlerfrei implementiert, weshalb er im GridImageProcessor auskommentiert war. Er sollte eigentlich ein PlayerPercept generieren, welches die Richtung und die Entfernung des gegnerischen Spielers relativ zur Eigenposition kennzeichnet.

3.3.2.1 Probleme des PlayerSpecialist

Der PlayerSpecialist hatte leider noch nie richtig funktioniert. Er musste deshalb während der Projektgruppe 2003 innerhalb des GridImageProcessors auskommentiert werden. Somit gab es keine PlayerPercepts, was dazu führte, dass die Roboter sich oft untereinander oder mit den Gegnern auf einem Fleck „knubbelten“ und sich gegenseitig behindern.

3.3.2.2 Ziel und Lösung

Da zu Beginn der Projektgruppe kein funktionierender PlayerSpecialist vorlag, mussten neue Lösungsansätze entwickelt werden. Dabei wurden zwei Verfahren ausprobiert:

- Berechnung einer Gegnerposition mittels einer Hough-Transformation
- Berechnung einer Gegnerposition mittels einer Multiplen Bounding Box

3.3.2.3 Berechnung einer Gegnerposition mittels einer Hough-Transformation

Die Hough-Transformation [9] ist ein globaler Ansatz, um strukturelle Beziehungen zwischen Pixeln zu finden.

Angenommen, in einem Bild seien n Punkte gegeben und die Teilmengen dieser Punkte, die auf einer Geraden liegen, seien gesucht. Dazu bietet die Hough-Transformation eine Lösung an. Für die Ermittlung solcher Beziehungen wird das Bild in die a-b-Ebene (Parameterraum) transformiert. Eine Gerade im x-y-Raum entspricht dabei einem Punkt im a-b-Raum und ein Punkt im x-y-Raum einer Gerade im a-b-Raum. Betrachtet von einem Punkt (x_i, y_i) in der x-y-Ebene in der Normalform $y_i = ax_i + b$, seien unendlich viele Linien durch den Punkt (x_i, y_i) in der gleichen Form, aber mit unterschiedlichen Werten von a und b. Jedoch ergibt sich eine einzelne Linie im Parameterraum für den Punkt (x_i, y_i) durch $b = -x_i a + y_i$. Ein zweiter Punkt (x_j, y_j) hat ebenfalls eine Linie im Parameterraum, die sich im Punkt (a', b') schneidet. Also haben alle Punkte einer geraden Linie im Parameterraum Linien, die durch den Schnittpunkt (a', b') verlaufen (siehe Abbildung 3.8 auf der nächsten Seite).

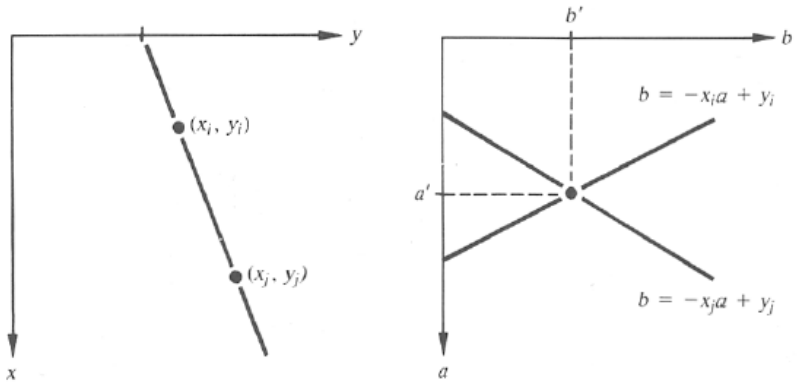

Abbildung 3.8: Parameterraum

Um die Punkte auf den Geraden zu finden, wurde zuerst ein sogenannter Akkumulator angewandt, der aus der Unterteilung des Parameterraumes in Akkumulatorzellen $A(i, j)$ entsteht (siehe Abbildung 3.9).

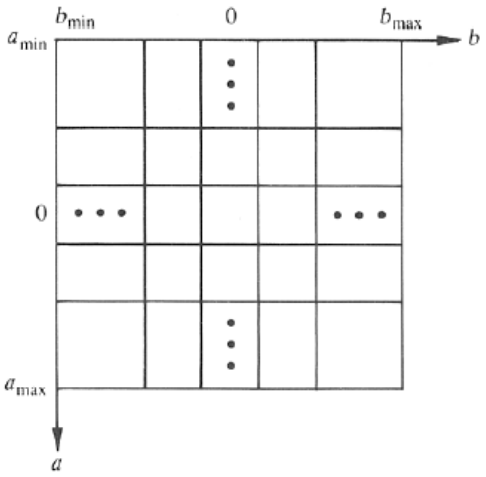

Abbildung 3.9: Akkumulatorzelle

Die Parameter (a_{max}, a_{min}) und (b_{max}, b_{min}) geben den erwarteten Bereich für die Neigung und die Schnittpunkte an. Die Zellen sind am Anfang auf Null initialisiert. Wenn eine Wahl getroffen wurde, wird die Gerade $b = -x_k a + y_k$ mit den vertikalen Zellrändern (i) geschnitten. Die Schnittpunkte werden zu dem nächsten horizontalen Zellrand (j) aufgerundet und $A(i, j)$ um 1 inkrementiert. Am Ende von diesem Verfahren entspricht $A(i, j)$, der Anzahl der Punkte, die näherungsweise auf der Geraden $y = a_i x + b_j$ liegen. Die Genauigkeit ist abhängig von der Unterteilung des Parameterraumes. Bei einer $k \times k$-Zellaufteilung wird $O(nk)$ Zeit gebraucht. Nahezu senkrechte Geraden führen also zu großen a- und b-Werten. Um dieses Problem zu lösen, wurde eine Geradendarstellung in der ρ-θ-Ebene eingeführt: $x \cos\theta + y \sin\theta = \rho$, (ρ, θ) bilden den Hough-Raum.

Es war geplant mit Hilfe der Hough-Transformation die Kanten der Gegner zu erkennen. Zuerst wird der Akkumulator erzeugt und die $A(i, j)$ berechnet. Nachdem die maximalen

Werte herausgefunden sind, werden die Bildpunkte ihrer Nachbarschaften entsprechend verbunden.

3.3.2.4 Berechnung einer Gegnerposition mittels einer Multiplen Bounding Box

Der Multiple Bounding Box Algorithmus sammelt während der Durchlaufphase des GridImageProcessors alle Informationen über rote bzw. blaue Pixel. Findet er die Farbe Rot, beginnt er eine Bounding Box aufzuspannen. Findet er in weiteren fünf Gridlines keine rote Farbe mehr, wird die Bounding Box abgeschlossen und in eine Liste eingefügt. Danach wird aus der Höhe der Bounding Box und der Anzahl gezählten roten Pixel die Entfernung zum Gegner berechnet.

Um eine vernünftige Relation zwischen der Größe der Bounding Box und der tatsächlichen Entfernung des Gegners zu finden, haben wir Logfiles nach folgendem Schema erstellt:

Der blaue Roboter, der den roten Gegner erkennen soll, wurde im blauen Tor aufgestellt, der Gegner im Abstand von genau 20 Zentimetern gegenüber. Danach wurde ein Screenshot der Kamera gemacht, aus dem die Höhe der Bounding Box und die gezählten roten Pixel in eine Excel-Tabelle übertragen wurden. Die weiteren Messwerte ergaben sich durch Weiterrücken des roten Roboters um jeweils 10 cm. Auf die Daten der Excel-Tabelle wurde dann das externe Programm „Discipulus“ angewendet mit dem Ziel, Logdaten für eine Entfernungsfunktion, die aus Boxhöhe und Pixelzahl die Entfernung bestimmt, durch Evolution zu gewinnen.

3.3.2.5 Fazit

Berechnung einer Gegnerposition mittels einer Hough-Transformation: Da um das oben genannte Problem (Nahezu senkrechte Geraden führen also zu großen a- und b-Werten) zu vermeiden eine $\rho \times \theta$-Zellaufteilung des Hough-Raums vorgenommen wird und außerdem noch die Bildpixel gelesen werden müssen, benötigt der Algorithmus nicht $O(nk)$, sondern $O(nkk)$ (drei For-Schleife ineinander). Also wird im worst case $O(n^3)$ Zeit gebraucht. Wir haben Verfahren in einer separaten Windows Anwendung implementiert und versucht die Kamerabilder des Roboters in den Hough-Raum zu transformieren. Dabei haben wir festgestellt, das die Transformation auf einem PC mit 1,8 Mhrz. rund 7 Sekunden benötigt. Umgerechnet auf die Prozessorgeschwindigkeit des AIBOS wären das ca. 63 Sekunden pro Frame. Da dies für eine robuste Gegnererkennung definitiv zu langsam ist haben wir uns für einen anderen Algorithmus entschieden.

Berechnung einer Gegnerposition mittels einer Multiplen Bounding Box: Leider hat sich dieser Algorithmus auch nicht als besonders effizient erwiesen. Eine Fehlerquelle bestand z.B. darin, dass nur teilweise sichtbare rote Roboter aufgrund der geringen Pixelzahl als zu weit entfernt eingestuft wurden. Solche Sprünge in den Percepts führten dazu, dass das Behavior empfindlich gestört wurde und so beispielsweise versucht wurde, irgendwelche Gegner zu umgehen, die gar nicht existierten (Abbildung 3.10 auf der nächsten Seite und Abbildung 3.11 auf der nächsten Seite).

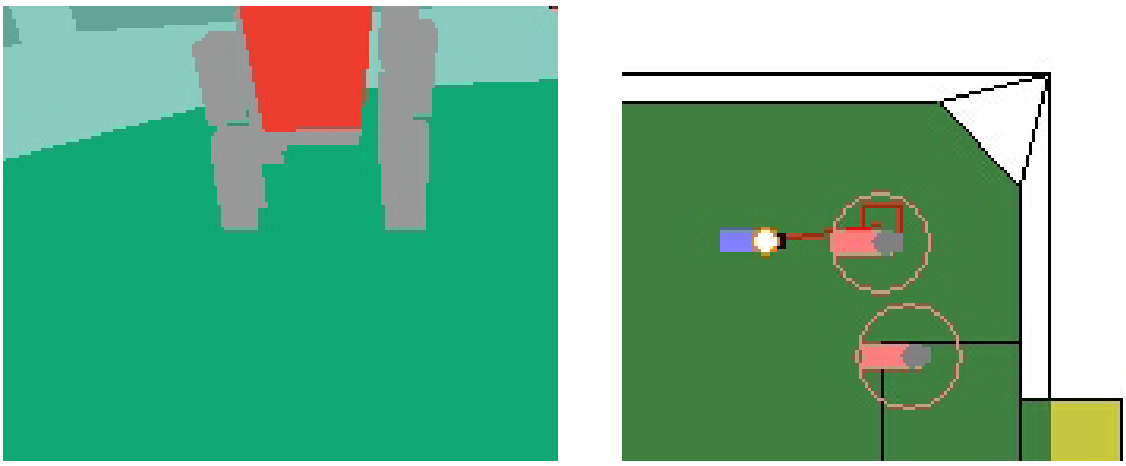

Abbildung 3.10: Links ist das Simulatorbild zu sehen, welches der Roboter verarbeitet. Rechts ist das dazugehörige Weltmodell abgebildet, in der die echten Positionen der Gegner blaßrot eingezeichnet wurden. Die berechnete Gegnerposition ist die rote Linie mit Rechteck. Bei diesem Bild funktioniert der Erkennungsalgorithmus einigermaßen gut.

Abbildung 3.11: Links wieder das Simulatorbild und rechts das Weltmodell. Hier kann man sehen, dass der Gegner viel zu weit hinten erkannt wird, obwohl er unmittelbar vor dem Spieler steht.

3.3.3 FlagSpecialist

Der FlagSpecialist wird aufgerufen, wenn der GridImageProcessor mehrere pinkfarbene Pixel findet. Von dieser Stelle zieht er Linien, die orthogonal zum Horizont liegen und bestimmt so ein Rechteck. Aus der Lage und Größe dieses Rechtecks wird dann ein LandmarkPercept berechnet.

Der FlagSpecialist wurde im Rahmen der Projektgruppe nicht verändert.

3.3.4 GoalSpecialist

Der GoalSpecialist wird aufgerufen, wenn – je nach Teamfarbe – gelbe bzw. hellblaue Pixel gefunden werden. Auch hier wird ein Rechteck um die farbige Fläche gespannt, um in einem LandmarkPercept ein Goal zu berechnen.

Der GoalSpecialist wurde im Rahmen der Projektgruppe nicht verändert.

3.3.5 GridImageProcessor3

Der GridImageProcessor3 ist eine Variation des GridImageProcessor. Er enthält keine neuartigen Ideen, wurde jedoch vom Code her optimiert. Zum Beispiel wurden einige Variablen in Registervariablen umgewandelt oder Berechnungen, die sich oft wiederholen, wurden in Look-Up-Tabellen gespeichert.

3.4 Benchmarks

Um entscheiden zu können, wie sich die Änderungen in der Bildverarbeitung auf Qualität und Laufzeit des Bildverarbeitungsmodules auswirken, wurden einige Vergleichsmessungen durchgeführt. Untersucht wurden der BlobImageProcessor, der GridImageProcessor, der GT2003ImageProcessor und der GridImageProcessor3 sowohl mit C64-Farbraum als auch mit TSL.

Ein durchgeführter Benchmark sollte die Durchlaufzeiten durch das Bildverarbeitungsmodul ermitteln. Dazu wurde ein AIBO auf das Spielfeld gesetzt und in RobotControl die Laufzeiten mit dem „Time Diagram Dialog“ angezeigt. Da die Laufzeiten auch von der Bildqualität abhängen können, wurden die Messungen mit verschiedenen Bildern durchgeführt (darunter auch Bilder mit ausgeschalteter Beleuchtung, um schwer zu klassifizierende Bilder zu erhalten). Die Ergebnisse sind in Tabelle 3.2 auf der nächsten Seite dargestellt. Es kam heraus, dass TSL ohne Look-Up Table die Laufzeiten des Bildverarbeitungsmodules unabhängig von der verwendeten Solution stark erhöhten. Beim BlobImageProcessor wirkte sich dies so stark aus, dass eine genaue Messung nicht mehr möglich war. Es war festzustellen, dass der BlobImageProcessor und der GridImageProcessor bei Verwendung von TSL mit Look-Up Table die besten Geschwindigkeiten erreicht wurden. Der im German Team eingesetzte GT2003ImageProcessor wurde mit

den schlechtesten Geschwindigkeiten gemessen, wenn man einmal von der exakten TSL-Variante absieht.

Solution	Farbraum	Laufzeit [in ms]
BlobImageProcessor	C64	12–30
BlobImageProcessor	TSL (exakt)	$\gg$120
BlobImageProcessor	TSL (Look-Up-Table)	8–10
GridImageProcessor	C64	13–20
GridImageProcessor	TSL (exakt)	25
GridImageProcessor	TSL (Look-Up-Table)	5–7
GridImageProcessor3	C64	10–20
GridImageProcessor3	TSL (exakt)	30–37
GridImageProcessor3	TSL (Look-Up-Table)	10–18
GT2003ImageProcessor	C64	17–22
GT2003ImageProcessor	TSL (exakt)	100–110
GT2003ImageProcessor	TSL (Look-Up-Table)	14-20

Tabelle 3.2: Durchlaufzeiten des Bildverarbeitungsmoduls bei verschiedenen Solutions

Um die von der Kamera gelieferte Wiederholrate von 25 Hz zu erreichen, dürfen alle Module zusammen die Laufzeit von $\frac{1000}{25} = 40$ ms nicht überschreiten. Die Bildverarbeitung macht – neben der Selbstlokalisierung – dabei einen großen Teil der Laufzeit aus, so dass hier eine Laufzeit ≤ 20 ms angestrebt werden sollte.

Ein zweiter durchgeführter Benchmark sollte die Qualität der Ergebnisse der Bildverarbeitung beurteilen. Dieser Benchmark wurde ohne die exakte Variante des TSL-Farbraums durchgeführt, da sich dieser auf Grund der beim ersten Benchmark beobachteten Laufzeiterhöhungen als nicht gut brauchbar erwies. Zur Durchführung des Benchmarks wurde im Code des Cognition-Prozesses eine Änderung implementiert, die Zeit, Koordinaten der Selbstlokalisierung und Validität der Selbstlokalisierung ausgibt, sobald in der Selbstlokalisierung eine Validität von 90% erreicht wird. Vor jeder neuen Messung wurden die alten Daten der Selbstlokalisierung gelöscht, damit bei jeder Messung der gleiche Ausgangszustand vorliegt. Für die Selbstlokalisierung wurde dabei der MonteCarloSelfLocator verwendet. Die Ergebnisse dieses Benchmarks finden sich in Tabelle 3.3 auf der nächsten Seite. Es wurde festgestellt, dass auch hier der BlobImageProcessor in Verbindung mit der TSL-Variante mit Look-Up Table die besten Ergebnisse erzielen konnte: im Durchschnitt wurde 1 Sekunde benötigt, um eine Validität von ca. 90% zu erreichen. Der im German Team hauptsächlich verwendete GT2003ImageProcessor benötigt – unabhängig vom verwendeten Farbraum – für diese Aufgabe im Schnitt 3 bis 4 Sekunden. Der GridImageProcessor lag zwischen diesen beiden Werten und der GridImageProcessor3 benötigte am Längsten.

Anhand dieser Benchmarks lässt sich erkennen, dass der BlobImageProcessor durch Verwendung von TSL einen Geschwindigkeitsvorteil gegenüber dem GT2003ImageProcessor hat. Allerdings ermittelt der GT2003ImageProcessor weitere Daten, z.B. die für den LinesSelfLocator benötigten LinesPercepts. Diese werden vom BlobImageProcessor nicht ermittelt, so dass dies wiederum einen Nachteil des BlobImageProcessors darstellt.

Solution	Farbraum	gemessene Zeit [in ms]
BlobImageProcessor	C64	2000–3000
BlobImageProcessor	TSL (Look-Up-Table)	1000
GridImageProcessor	C64	1500
GridImageProcessor	TSL (Look-Up-Table)	2000–3000
GridImageProcessor3	C64	3500
GridImageProcessor3	TSL (Look-Up-Table)	2500–7000
GT2003ImageProcessor	C64	4000
GT2003ImageProcessor	TSL (Look-Up-Table)	3000–5000

Tabelle 3.3: Zeit bis zum Erreichen einer Selbstlokalisierungsvalidität von 90% bei verschiedenen Solutions für das Bildverarbeitungsmodul

3.5 Erzielung einer höheren Kamera-Auflösung

Ein wesentliches Problem für die Bildverarbeitung ist die geringe Kamera-Auflösung von nur 176x144 Pixeln (für Y, U und V).

Da die meisten der im Videobereich verwendeten Kameras eine Auflösung von 352x288 (für Y) und 176x144 (für U und V) haben, liegt die Vermutung nahe, dass auch die im AIBO verwendete Kamera ursprünglich 352x288 Pixel im Y-Kanal liefert.

Unterstützt wird diese Vermutung von einigen Defines in den Open-R-Includes, die darauf hindeuten, dass man außer auf die Y-, U- und V-Kanäle noch auf drei weitere Kanäle zugreifen kann (die Kanäle haben alle die Größe 176x144 Pixel):

- ofbkImageBAND_Y
- ofbkImageBAND_U
- ofbkImageBAND_V
- ofbkImageBAND_Y_LH
- ofbkImageBAND_Y_HL
- ofbkImageBAND_Y_HH

Die Bezeichnungen LH, HL und HH lassen den Schluss zu, dass es sich um Wavelet-basierte (siehe Kapitel 8.2.1 auf Seite 104) Kanäle handelt. Der Y-Kanal ist dann als LL zu sehen. Stünden all diese Daten zur Verfügung, so könnte man leicht daraus ein 352x288 Pixel großes Y-Bild berechnen.

In der derzeitigen Version von Open-R ist dieser Zugriff allerdings blockiert. Ein aufwändiges Disassemblieren des Open-R ergab, dass die Daten der drei weiteren Kanäle nicht geschickt werden, da vermutlich einige Datenstrukturen nicht entsprechend gesetzt sind. Versuche, diese von Hand zu ändern, scheiterten jedoch an der mangelnden Verfügbarkeit von Beschreibungen der Hardware-Module (CCD-Kamera und angeschlossener Mikro-Controller).

Ein sich während der RoboCup-WM in Padua ergebendes Gespräch mit den Entwicklern von Open-R ergab, dass hierzu in den Treibern der Klasse OVirtualRobot einige Register anders initialisiert werden müssten. Auf Anfrage per e-Mail bestätigte der Entwickler, dies in der kommenden Version von Open-R ändern zu wollen.

Kapitel 4

Team-Kommunikation

Die Roboter haben eine eingebaute WLAN-Karte und können so untereinander Nachrichten austauschen. Dadurch kann das Spiel selbstverständlich gegenüber einem Spiel ohne Kommunikation verbessert werden, wenn z.B. Ballpositionen übermittelt werden oder das Verhalten zwischen den Robotern abgesprochen wird.

Es wird beschrieben, wie die Übermittlung der Daten funktioniert und wie sie erweitert wurde.

4.1 Stand zum Beginn der Projektgruppe

Die Roboter kommunizieren untereinander über das TCP-IP Protokoll.

Open-R bietet dafür verschiedene Möglichkeiten an:

4.1.1 Direkte IP-Kommunikation

In Open-R gibt es eine Bibliothek zum Programmieren von Netzwerkapplikationen, die „ANT-Library“. Mit dieser Bibliothek kann man TCP-Verbindungen aufbauen, auf TCP-Verbindungen warten und UDP- und RAW-Pakete senden und empfangen. Außerdem bietet sie Unterstützung für verschiedene Application-Level Protokolle wie mail und http.

4.1.2 TCP-Gateway

Die in den Regeln vorgeschriebene Art der Teamkommunikation funktioniert allerdings etwas anders: Auf den Robotern läuft sie über einen speziellen Prozess des Betriebsystems, den TCP-Gateway, ab. Dieser bekommt über die in Aperios integrierte Inter-Prozess-Kommunikation Daten übermittelt. Anhand von Konfigurationsdateien entscheidet dieser Prozess, wohin welche Daten müssen und versendet sie über die direkte IP-Kommunikation (der Sourcecode ist nicht veröffentlicht).

Für den Verbindungsaufbau zwischen den Robotern muss nun auf einem PC ebenfalls ein TCP-Gateway gestartet werden. Dieser baut nun für jeden Datentyp eine Verbindung zu jedem Roboter auf.

Möchte man nun von einem Prozess A von Roboter A Daten zu einem Prozess B auf Roboter B senden, so läuft das folgendermaßen ab (siehe dazu auch Abbildung 4.1):

1. Prozess A sendet über die Inter-Prozess-Kommunikation die Daten an den TCP-Gateway auf Roboter A
2. Der TCP-Gateway sendet die Daten über ein TCP-basierendes, nicht veröffentlichtes Protokoll an den TCP-Gateway auf dem PC.
3. Der TCP-Gateway sendet die Daten über jenes Protokoll weiter an den TCP-Gateway auf Roboter B
4. Der TCP-Gateway auf Roboter B sendet die Daten über die Inter-Prozess-Kommunikation zu Prozess B.

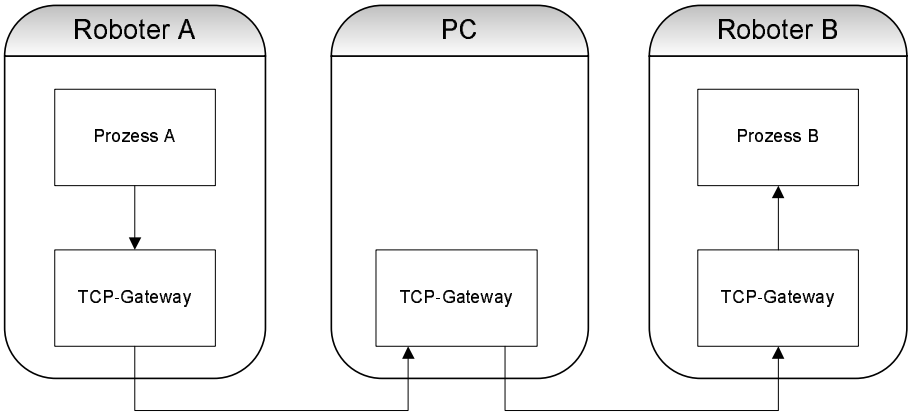

Abbildung 4.1: TCP-Gateway basierende Kommunikation zwischen zwei Robotern.

Dieser gesamte Ablauf dauert sehr lange. So brauchen Daten, um von Prozess A zu Prozess B und wieder zurück zu kommen (Ping-Zeit), zwischen 600ms und 800ms.

4.1.3 Kommuniaktion in GT2003

Im German Team wird wegen der in der Liga festgelegten Regel eine TCP-Gateway basierende Kommunikation verwendet, die über die Datenpaket-Klassen (siehe 2.4.1.2 auf Seite 11) TeamMessage1, TeamMessage2 und TeamMessage3 realisiert wird. Diese Klassen erben alle von TeamMessage und existieren nur, da man bei Verwendung des TCP-Gateways wegen der Inter-Prozess-Kommunikation für jeden Kommunikationskanal einen Sender/Receiver einer eigenen Klasse benötigt.

Es werden nun für TeamMessage1, TeamMessage2 und TeamMessage3 im German-Team Process-Framework jeweils ein Sender und ein Receiver definiert. Über die connect.cfg-Datei und die robotgw-Datei auf den Robotern und den entsprechenden Dateien

des TCP-Gateway auf dem PC wird das System nun so konfiguriert, dass die Daten mit einem der 3 Sender jeweils an einen der 3 anderen Roboter verschickt werden (Abbildung 4.2).

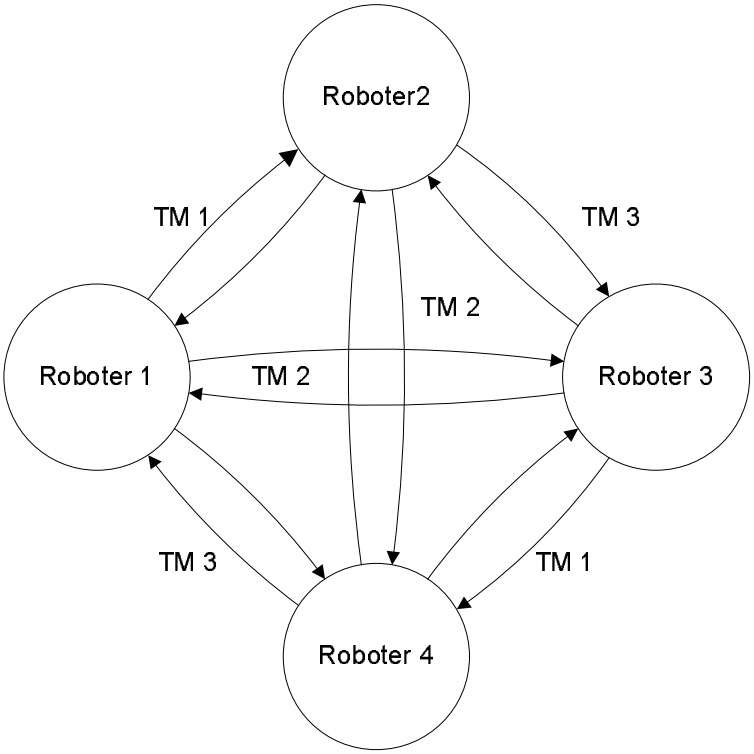

Abbildung 4.2: Kommunikation zwischen den Robotern mit den Klassen TeamMessage1 (TM 1) , TeamMessage2 (TM 2) und TeamMessage3 (TM 3)

Um all dies zu verwalten, gibt es die Klasse TeamMessageCollection. Sie verwaltet eine Liste von 3 TeamMessages, die am Anfang des Cognition-Prozesses mit den Receiver-Objekten initialisiert wird. Über diese Klasse kann man nun auf die von den 3 anderen Robotern versandten Daten zugreifen.

Die TeamMessage Klasse selbst bestand aus den Public-Attributen ballPosition, playerPoseCollection usw., die zur Aufnahme der entsprechenden Daten dienen und 2 Zeitstempeln, dem TimeStamp und dem LastReceivedTimeStamp. Außerdem wurden Streaming-Operatoren definiert, die dafür sorgten, dass die Daten über das Sender-Receiver Konzept an den TCP-Gateway verschickt wurden. Diese Streaming-Operatoren verschickten immer alle Daten.

4.2 Verbesserungsmöglichkeiten / Erweiterungen

Die im German-Team verwendete Teamkommunikation wurde nun folgendermaßen verbessert:

- Das System, über 3 TCP-Gateways Daten zu übermitteln, ist sehr langsam, falls zu viele Daten übermittelt werden. Diese Datenmenge sollte reduziert werden.

- Es gab zu Beginn der Projektgruppe keine Möglichkeit, kommunizierte Zeiten zu vergleichen. Dafür wurde eine eine „Teamzeit“ eingeführt. Damit verfügen die Roboter über eine gemeinsame Zeitbasis, die z.B. von der Verhaltenssteuerung zum Synchronisieren von Aktionen verwendet werden kann.
- Zur Synchronisierung des Verhaltens wurde das Aushandeln eines Masters benötigt. Es soll nur einen Roboter im Team geben, der Master ist, und es soll möglich sein, dass ein anderer Roboter Master wird, wenn der Master ausfallen sollte.

Um die Auswirkungen dieser Änderungen auf den restlichen Code des GermanTeams möglichst gering zu halten, wurden die Klassen TeamMessage und TeamMessageCollection erweitert.

4.3 Umbau der Klassen TeamMessage und TeamMessageCollection

Im Streaming-Operator der TeamMessage wurden bisher immer alle Informationen übertragen, die als Attribute in ihr vorhanden waren. Zur effizienteren Bandbreitennutzung sollten nur die Daten übertragen werden, die wirklich übertragen werden müssen.

Die Aufgabe, zu entscheiden, ob und welche Daten gesendet werden müssen, muss dabei in einer Klasse stattfinden, die alle 3 TeamMessages kennt. Die Klasse TeamMessageCollection wurde um diese Funktionalität erweitert.

Dazu wurden in die Klasse TeamMessageCollection send-Methoden eingeführt. Die send-Methode ist überladen, um die verschiedenen Datentypen senden zu können. Zusätzlich kann man ein Ziel (Goalie, Defender, Striker1, Striker2) angeben und bestimmen, ob die Nachricht sofort oder erst nach Ablauf des Delay (default 500ms) gesendet werden soll.

Die Klasse TeamMessage wurde für diese Funktionalität ebenfalls umgebaut. Zu jedem Datentyp wurde eine bool - Variable hinzugefügt, die von der entsprechenden send-Methode der TeamMessageCollection gesetzt wird, und besagt, dass der entsprechenden Datentyp gesendet werden soll.

Der Streaming-Operator der TeamMessage sendet nun zuerst ein Bitfeld, in dem jedes Bit angibt, ob ein entsprechender Datentyp gesendet wird. Danach kommen die zu sendenden Daten in einer festgelegten Reihenfolge.

4.3.1 Verwendung der neuen TeamMessage / TeamMessageCollection

Damit die neue TeamMessageCollection funktionieren kann, muss sie zusätzlich zu den TeamMessageReceivern auch die TeamMessageSender kennen. Zum Initialisieren müssen daher die Methoden setInTeamMessages und setOutTeamMessages im Konstruktor von Cognition aufgerufen werden.

```
teamMessageCollection.setInTeamMessages(theTeamMessage1Receiver,
    theTeamMessage2Receiver,
    theTeamMessage3Receiver);

teamMessageCollection.setOutTeamMessages(theTeamMessage1Sender,
    theTeamMessage2Sender,
    theTeamMessage3Sender);
```

Am Beginn des Cognition-Prozesses (main methode) wird nun mit

```
teamMessageCollection.processMessages();
```

die Verarbeitung der TeamMessages in der TeamMessageCollection aufgerufen.

Am Ende werden dann die ausgehenden Teammessages noch einmal verarbeitet, bevor sie gesendet werden.

```
if ((teamMessageCollection.processOutMessages()))
{
    theTeamMessage1Sender.send();
    theTeamMessage2Sender.send();
    theTeamMessage3Sender.send();
}
```

Zwischen dem Aufruf von processMessages und processOutMessages können nun überall, also auch in den Modulen, die die TeamMessageCollection übergeben bekommen, die send-Methoden verwendet werden.

Zur Zeit wird die Möglichkeit der Bandbreitenschonenden Übertragung im German Team-Code noch nicht verwendet. Es werden durch die Aufrufe

```
teamMessageCollection.send(thePackageCognitionMotionSender.robotPose);
teamMessageCollection.send(thePackageCognitionMotionSender.ballPosition.seen);
teamMessageCollection.send(outgoingBehaviorTeamMessage);
teamMessageCollection.send(playersPercept);
```

stets alle Daten versendet.

4.4 Zeitsynchronisation

Eine der wichtigsten Erweiterungen der Team-Kommunikation ist die Möglichkeit der Zeitsynchronisation. Mit ihr ist es möglich, die Zeitdifferenz (Offset) der Systemzeiten zu den anderen Robotern zu ermitteln und Zeitstempel von anderen Robotern in eigene umzurechnen.

4.4.1 Algorithmus zur Zeitsynchronisation

Zur Berechnung der Zeitdifferenz wurde die TeamMessage um einen weiteren Timestamp erweitert werden, der benötigt wird, um den Offset berechnen zu können:

(im Folgenden soll der Offset zwischen Roboter A und Roboter B berechnet werden [10])

- T_1 = LastReceivedTimeStamp = Zeitpunkt der letzten von A an B gesendeten Nachricht.

- T_2 = IncomingTimeStamp = Ankunftszeitpunkt dieser Nachricht bei B
- T_3 = Zeitpunkt des Versendens der Antwort von B nach A
- T_4 = Systemzeit von A beim Empfang dieser Nachricht
- T_4^* = Systemzeit von B beim Empfang dieser Nachricht.

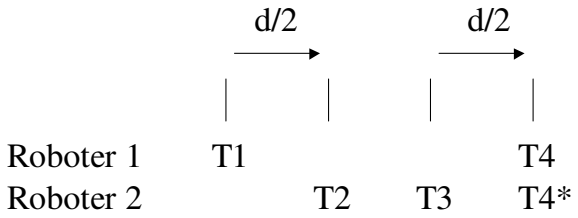

Abbildung 4.3: Zeitpunkte bei der Berechnung des Offsets

Aus diesen vier Zeiten kann man die Differenz der Systemzeiten (Offset) beider Roboter mit der Formel 4.1 berechnen:

$$\begin{aligned} \text{Offset} &= T_4^* - T_4 \\ &= T_3 + \frac{d}{2} - T_4 \\ &= T_3 + \frac{T_4 - T_1 - T_3 + T_2}{2} - T_4 \\ &= \frac{1}{2} * (T_2 - T_1 + T_3 - T_4) \end{aligned} \tag{4.1}$$

Da diese Offsets schwanken, wird der Mittelwert der letzten 100 Offsets benutzt. Ist ein Roboter nicht verbunden, so steigt der Offset zu diesem natürlich kontinuierlich.

4.4.2 Testergebnisse

Zur Überprüfung der Zeitsynchronisation wurde diese sowohl im Simulator als auch auf den Robotern getestet.

4.4.2.1 im Simulator

Die Methode GetSystemTime, die die aktuelle Systemzeit der Roboter zurück gibt, liefert im Simulator für alle Roboter die gleiche Systemzeit zurück. Daher sollten die Offsets zwischen allen Robotern nahezu 0 sein. Im Test lagen die Offsets zwischen -3 und +3 ms.

Zur besseren Visualisierung wurde nun ein Test-Verhalten für die Roboter geschrieben, das die Zeitsynchronisation benutzt. Dabei sollten die Roboter eine Bewegung, den Kopfstand, möglichst synchron starten.

Dieser Test wird mit einem Druck auf den Rückentaster eines der Roboter gestartet. Dieser sendet dann einen Befehl an die anderen Roboter, so dass sie exakt 2 Sekunden später den Kopfstand starten. Ein Problem dabei war, dass es nicht möglich ist, dem Roboter in SIMGT2003 auf den Rücken zu drücken (in RobotControl ist dafür ein Button vorhanden). Daher wurde noch eine weitere Möglichkeit zum Auslösen des Befehls eingebaut: Wenn ein Roboter über den GameManager eine „gelbe Karte“ bekommt, so sendet dieser den Befehl auch.

Der Test dieses Verhaltens im Simulator verlief erwartungsgemäß.

4.4.2.2 auf Robotern

Nach einigen Problemen mit dem Netzwerk konnte der Kopfstand-Test auch auf realen Robotern durchgeführt werden.

Dabei machten alle Roboter ihren Kopfstand nach zwei Sekunden ohne einen sichtbaren Synchronisationsfehler.

Wenn man allerdings die Roboter längere Zeit laufen gelassen hat, kam es zu Problemen, deren Grund leider nicht genau nachvollzogen werden konnte. Die Berechnung der Offsets war nicht mehr richtig, was wir daran erkennen konnten, dass bei den Kopfständen eine deutliche Verzögerung von mehr als einer Sekunde eintrat.

Diese Probleme konnten dadurch behoben werden, dass nicht alle 250ms eine Nachricht gesendet wurde sondern nur noch, wie auch schon in der alten Version der Teamkommunikation, alle 500 ms. Daraufhin wurde noch die Möglichkeit eingebaut, die Verzögerung zur Laufzeit zu ändern.

4.5 Master-Wahl

Zum Synchronisieren der Verhaltenssteuerungen der verschiedenen AIBOs war es notwendig, ein Verfahren zu finden, welches unter den eingeschalteten und ins WLAN eingebundenen Robotern einen auswählt, der dann als Master seine Daten den anderen zur Synchronisierung zur Verfügung stellt. Hierbei mussten verschiedene Dinge gewährleistet werden:

1. Es darf nur einen Master geben. Sollte festgestellt werden, dass zwei AIBOs sich für den Master halten, muss dieser Umstand beseitigt werden.
2. Es muss einen Master geben. Sollte festgestellt werden, dass kein Master aktiv ist, muss einer der anderen Roboter sich zum Master machen.
3. Jeder Roboter muss wissen, wer der Master ist.
4. Da die Roboter nicht in fester Reihenfolge booten und somit nicht gleichzeitig ins WLAN eingebunden werden und außerdem Spiele mit weniger Spielern möglich sein sollen, darf die Rolle des Masters nicht an einen physikalischen Roboter gebunden sein.

Um dies alles zu gewährleisten, wurde ein Verfahren implementiert, das darauf basiert, dass Nachrichten zwischen den AIBOs ausgetauscht werden und damit festgestellt werden kann, ob es einen Master gibt und wer dieses ist.

4.5.1 Algorithmus zur Master-Wahl

Der Algorithmus benötigt zwei Nachrichten, die zwischen den Robotern ausgetauscht werden können. Eine Nachricht (genannt I-Am-Master) teilt den anderen mit, dass der Roboter, der diese schickt, Master ist; die andere Nachricht (Master-search) teilt den anderen mit, dass der sendende Roboter herauszufinden versucht, wer der Master ist. Die Nachrichten sind als enum im Code realisiert, der zusätzlich zu den Nachrichten auch noch den Eintrag „none“ enthält, für den Fall, dass keine Nachricht zu senden ist.

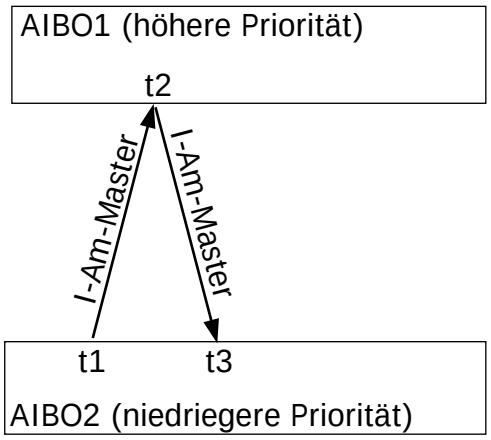

Abbildung 4.4: Konflikt zwischen zwei AIBOs, die sich beide für den Master halten

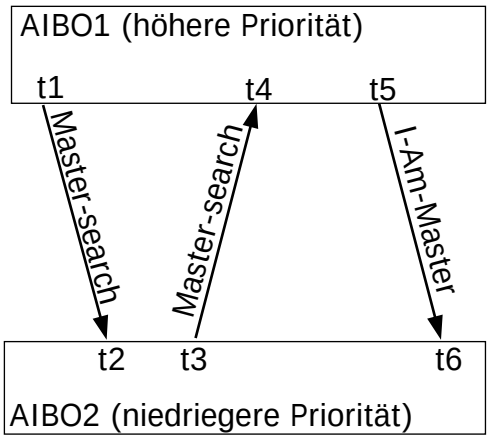

Abbildung 4.5: Nachrichtenaustausch zwischen zwei Robotern, die gerade ins WLAN kommen

Der Algorithmus unterscheidet zwischen zwei Fällen:

1. Der Roboter ist Master.

   In diesem Fall werden die eingehenden TeamMessages daraufhin untersucht, ob sie eine Master-search-Nachricht enthalten. Sollte dies der Fall sein, wird als Antwort eine I-Am-Master-Nachricht an den Absender zurückgeschickt.

   Sollte eine eingehende TeamMessage eine I-Am-Master-Nachricht enthalten, halten sich zwei Roboter für den Master. Um diesen Fall aufzulösen, muss einer der beiden Roboter die Masterrolle fallen lassen. Dazu werden die PlayerRoles vom Sender und Empfänger miteinander verglichen. Diese sind als enum implementiert, so dass eine einfache Möglichkeit einer Prioritätswahl darin besteht, die PlayerRole einfach als Zahl aufzufassen und denjenigen Roboter mit der größeren PlayerRole die Masterrolle aufgeben zu lassen.

Sollte der Roboter, der gerade die I-Am-Master-Nachricht empfangen hat, derjenige sein, der die schlechtere Priorität hat, gibt er die Masterrolle auf und speichert die Nummer des anderen Roboters als Nummer des Master ab.

Sollte der andere Fall eintreten, so bleibt der Roboter selbst Master. Damit sichergestellt ist, dass der andere Roboter mitbekommt, dass ein zweiter (besser priorisierter) Roboter Master ist, wird an diesen erneut eine I-Am-Master-Nachricht geschickt (siehe dazu auch Abbildung 4.4 auf der vorherigen Seite).

Um zu vermeiden, dass die anderen Roboter eine neue Masterwahl wegen eines Timeouts starten (mehr dazu auch unter Punkt 2), wird in regelmäßigen Abständen eine I-Am-Master-Nachricht an alle abgeschickt.

2. Der Roboter ist nicht Master.

   In diesem Fall werden die eingehenden TeamMessages untersucht, ob sie Masterwahl-Nachrichten enthalten. Bei einer eingehenden I-Am-Master-Nachricht wird die Nummer des Absenders als Master abgespeichert. Bei einer eingehenden Master-search-Nachricht eines AIBOs mit besserer Priorität (zur Prioritätsberechnung mehr unter Punkt 1 auf der vorherigen Seite) wird ein eventuell gestarteter Versuch, Master zu werden, abgebrochen.

   Für den Fall, dass keine Nummer eines Masters bekannt ist, wird der Versuch gestartet, Master zu werden. Dazu wird eine Master-search-Nachricht rausgeschickt und der Sendezeitpunkt gespeichert. Wenn seit dem Senden dieser Nachricht eine gewisse Zeitspanne vergangen ist, ohne dass sich ein Master mit einer I-Am-Master-Nachricht gemeldet hat und ohne dass die Suche (wie oben beschrieben) abgebrochen wurde, erklärt sich der AIBO zum Master und sendet eine I-Am-Master-Nachricht an alle.

   Dies ist in Abbildung 4.5 auf der vorherigen Seite dargestellt. Zum Zeitpunkt t1 beginnt Aibo1 mit der Suche nach dem Master. Kurz darauf (zum Zeitpunkt t2) beginnt auch Aibo2, bevor er dann (t3) die Nachricht von Aibo1 empfängt. Da Aibo1 besser priorisiert ist, beendet Aibo2 zu diesem Zeitpunkt den Versuch, Master zu werden. Zum Zeitpunkt t4 erhält dann Aibo1 die Nachricht von Aibo2, erkennt aber, dass er eine höhere Priorität hat, und unternimmt daher nichts weiter. Zum Zeitpunkt t5 ist dann die Wartezeit abgelaufen und Aibo1 erklärt sich zum Master. Dies teilt er allen anderen Robotern mit. Aibo2 empfängt zum Zeitpunkt t6 dann die I-Am-Master-Nachricht und weiß von da an, wer Master geworden ist.

   Wenn dagegen die Nummer des Masters bekannt ist, wird nur überprüft, wie lange es her ist, dass die letzte Nachricht vom Master empfangen wurde. Wird dabei ein TimeOut überschritten, wird davon ausgegangen, dass der Master nicht mehr verfügbar ist. Dies kann z.B. der Fall sein, wenn es Probleme mit der Kommunikation gibt oder der Master abgeschaltet wurde. In diesem Fall wird wieder der Versuch gestartet, Master zu werden.

4.5.2 Tests des Algorithmus

Um die Funktionsfähigkeit des beschriebenen Algorithmus zu testen, wurden Tests im Simulator und auf den Robotern gemacht. Dazu wurde zuerst einmal getestet, ob überhaupt ein Master gewählt wird. Außerdem wurden Sonderfälle getestet, wie z.B. der vorüber gehende Ausfall der Kommunikation.

4.5.2.1 Tests im Simulator

Bei ersten Tests im Simulator wurde ein gravierendes Problem bemerkt: Da eine eingegangene TeamMessage erst durch eine später gesendete überschrieben wird, wurde mehrfach auf die gleiche Nachricht reagiert, so dass der Master beispielsweise auch nach abgeschlossener Masterwahl andauernd I-Am-Master-Nachrichten schickte, ohne dass dies notwendig gewesen wäre. Dieses Problem wurde behoben, indem die Zeitstempel, die sich in den TeamMessages befinden, abgespeichert werden und der Empfang einer neuen Nachricht durch Vergleich mit dem gespeicherten Zeitstempel verifiziert wird.

Die weiteren Tests verliefen erwartungsgemäß. Während des vorübergehenden Ausfalls der Kommunikation wurden sämtliche Roboter nach kurzer Zeit zu Mastern, da ja keine Nachrichten vom ursprünglichen Master empfangen werden konnten. Sobald die Kommunikation wiederhergestellt war, wurde dieses Problem bemerkt und nur ein Roboter blieb Master, alle anderen gaben ihren Masterstatus auf.

4.5.2.2 Tests auf dem Roboter

Um Tests auf dem Roboter durchführen zu können, musste zuerst eine Möglichkeit gefunden werden, von außen feststellen zu können, wer sich gerade für einen Master hält. Dazu wurde eine kleine Modifikation vorgenommen, so dass die LEDs am Kopf der AIBOs für diese Debugausgabe genutzt werden konnten. Während der Tests auf dem Roboter wurden keine Probleme festgestellt.

4.6 Fazit

Diese Verbesserungen in der TeamMessage und TeamMessageCollection konnten in anderen Modulen gut verwendet werden. Die Zeitsynchronisation wurde z.B. bei der Ballfusion zur Berechnung des „Alters“ von Percepts (siehe Kapitel 5.4 auf Seite 53) verwendet. Die Verhaltenssteuerung benutzt die Masterwahl zur Synchronisation.

Kapitel 5

Sensorfusion zur Ballmodellierung

Die Ballmodellierung ist eines der wichtigsten, wenn nicht sogar das wichtigste Gebiet der Weltmodellierung, da man das Fußballspielen da der Roboter immer wissen muss, wo der Ball ist.

Dieses Kapitel beschreibt wie die Ballmodellierung durch eine probabilistische Modellierung und durch Sensorfusion verbessert wurde. Dazu werden nach einer Einführung die mathematischen Grundlagen kurz erläutert und danach der Algorithmus erklärt.

5.1 Stand zu Beginn der Projektgruppe

Die Ballmodellierung als Teil der Weltmodellierung fand man als eigenes Modul, dem BallLocator, im GT2003-Projekt. Er bekam von der Bildverarbeitung BallPercepts geliefert und gab eine Ballposition relativ zum Feld zurück (siehe Kapitel 2.4.2 auf Seite 13).

In diesem Modul wurde auch recht lange Zeit gearbeitet und ein großer Teil der im folgenden beschriebenen Algorithmen implementiert.

Nach den German Open wurde dieses Modul in 2 Module aufgeteilt. Eines dieser Module, der BallLocator, ist dafür zuständig, die BallPercepts auf Feldkoordinaten umzurechnen. Dafür wollten die Berliner eigene Algorithmen, die die Ballposition glätten, ausprobieren. Das 2. Modul, der TeamBallLocator, bekommt nun die vom BallLocator berechneten Ballpositionen aller Roboter geliefert, um eine CommunicatedBallPosition zu liefern.

Zu Beginn der Projektgruppe wurde im GT2003-Code die Ballposition nur dadurch bestimmt, dass die BallPercepts nur unter Zuhilfenahme der Position des Roboters in Ballpositionen auf dem Feld umgerechnet wurden. Außerdem wurden noch die letzten drei Ballpositionen gespeichert, um die Geschwindigkeit des Balls ermitteln zu können.

Auf von den anderen Robotern kommunizierte Ballpositionen wurde nur zurückgegriffen, wenn der Ball 7 Sekunden lang nicht gesehen wurde.

5.2 Verwendung von Sensorfusion

Zur Verbesserung der Qualität der Ballposition war es nahe liegend, so genannte Sensorfusion zu betreiben, d.h. auf über WLAN übermittelte Sensordaten von anderen Robotern zurückzugreifen, um die Position des Balles mit höherer Zuverlässigkeit bestimmen zu können.

Die Sensordaten, die zur Positionsbestimmung des Balles zur Verfügung stehen, sind die Bilddaten. Da diese aber relativ groß sind, wurden nicht diese, sondern die BallPercepts (Entfernung und Richtung des Balls) und die Position von jedem Roboter an alle anderen übermittelt.

Aus all diesen Daten soll der Roboter nun eine Ballposition ermitteln.

5.3 Ballmodellierung mit Normalverteilungen

Da es nicht möglich ist bei mehr als zwei Robotern durch Schnittpunksberechnungn der Sichtlinien eine Position zu berechnen, musste eine Möglichkeit der Ballmodellierung gefunden werden, die das Verarbeiten von mehreren BallPercepts von verschiedenen Robotern zulässt.

5.3.1 Beobachtung der Ballverteilungen

Um die beobachteten Ballpositionen und deren Verteilung einschätzen zu können, wurde in den GT2003-Code ein Debug-Drawing eingebaut, mit dem gesehen werden konnte, wo die erkannten Ballpositionen auf dem Feld liegen.

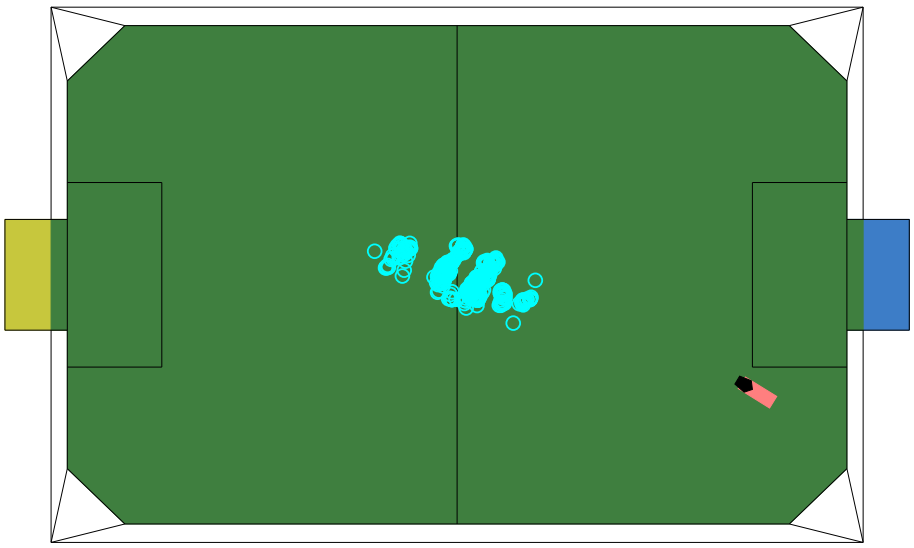

Abbildung 5.1: Verteilung von 250 beobachteten Ballpositionen eines auf dem Mittelpunkt liegenden Balls (nicht simuliert)

An dem in Abbildung 5.1 auf der vorherigen Seite gezeigten Beispiel ist deutlich zu erkennen, dass die Ballpositionen normalverteilt sein könnten. Daher wurde beschlossen, die Ballpositionen mittels zweidimensionalen Normalverteilungen zu modellieren.

Interessant an dieser Verteilung ist auch noch die Bildung von „Wellen“. Diese Häufungen von Ballpositionen rechtwinklig zur Blickrichtung des Roboters auf den Ball, entstehen durch die geringe Auflösung der Kamera. Die Bildverarbeitung kann den Rand des Balls nur mit einer Genauigkeit von einem Pixel erkennen. Da die Entfernung des Balls aus seiner Größe bestimmt wird und der Roboter sehr flach auf das Feld schaut, ergibt sich bei größeren Entfernungen eine deutliche Quantisierung des Abstandes.

5.3.2 Zweidimensionale Normalverteilung

Eine 2-dimensionale Normalverteilung wird durch Gleichung 5.1 beschrieben.

$$p(X) = \frac{1}{2\pi\sqrt{|C|}} e^{\left(\frac{1}{2}(X-\overline{X})^T C^{-1}(X-\overline{X})\right)} \tag{5.1}$$

Die Form und Lage wird durch die Kovarianz-Matrix C und den Vector $\overline{X}$ bestimmt. Dabei ist $\overline{X}$ der Erwartungswert der Normalverteilung. Die Kovarianz-Matrix ergibt sich aus den Standardabweichungen in die min und may Richtung, gedreht um den Winkel θ.

$$\overline{X} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{5.2}$$

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{5.3}$$

$$C = R(\theta) \begin{bmatrix} \sigma_{maj}^2 & 0 \\ 0 & \sigma_{min}^2 \end{bmatrix} R(-\theta) \tag{5.4}$$

All diese Werte müssen nun sinnvoll gefüllt werden. $\overline{X}$ ist dabei die aus dem Percept und der Roboterposition ermittelte Ballposition auf dem Feld. σ_{maj} ist die Standardabweichung in Blickrichtung des Roboters, σ_{min} die Standardabweichung rechtwinklig zur Blickrichtung des Hundes und θ der Blickwinkel [11] (siehe Abbildung 5.2 auf der nächsten Seite).

Wurde nun ein Ball von zwei Robotern beobachtet, so ergeben sich beispielsweise die in Abbildung 5.3 auf der nächsten Seite dargestellten Verteilungen.

5.3.3 „Mergen“ der Normalverteilungen

Die oben aufgestellten Verteilungen können nun zu einer gemeinsamen Verteilung zusammengefasst werden. Dazu wurde das „Merge“-Verfahren [11] angewandt.

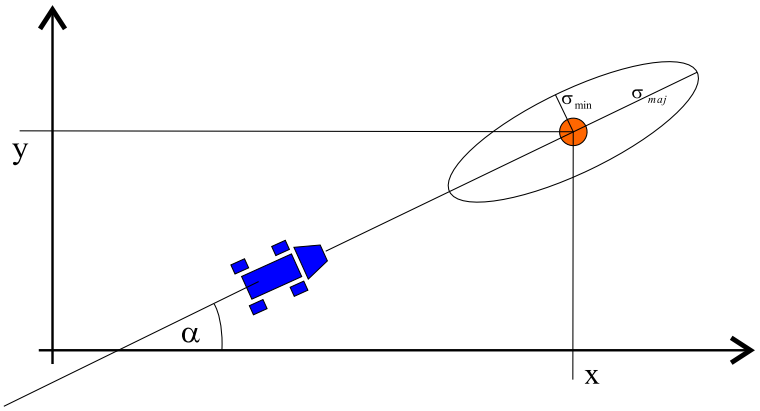

Abbildung 5.2: Bedeutung der Normalverteilungs-Parameter bei der Ballerkennung. Der Roboter sieht den Ball im Winkel α. Um den Ball herum wird die Normalverteilung mit den Varianzen σ_{min} und σ_{may} aufgestellt.

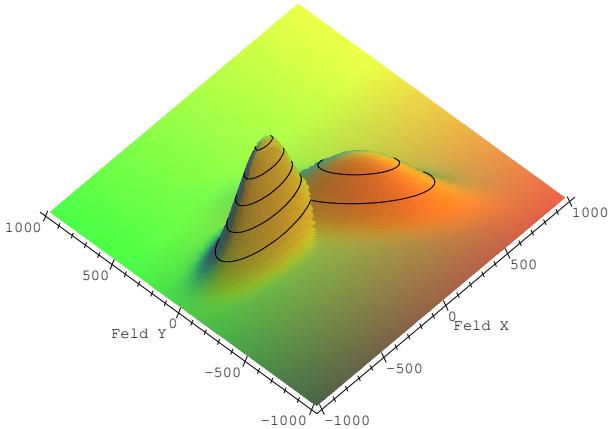

Abbildung 5.3: Verteilungen von zwei Beobachtungen des selben Balles. Der Ball liegt im Ursprung. Er wird von zwei Robotern aus unterschiedlichen Richtungen und Entfernungen beobachtet was zu zwei Normalverteilungen führt.

$$C_3 = C_1 - C_1 \left(C_1 + C_2\right)^{-1} C_1 \tag{5.5}$$

$$\overline{X_3} = \overline{X_1} + C_1(C_1 + C_2)^{-1}(\overline{X_2} - \overline{X_1}) \tag{5.6}$$

$$\theta = \frac{1}{2} \arctan\left(\frac{2C_{3_{1,2}}}{C_{3_{1,1}} - C_{3_{2,2}}}\right) \tag{5.7}$$

Mit diesen Formeln wird eine neue Kovarianz-Matrix (Formel 5.5) und eine neue Position (Formel 5.6) berechnet. Diese hat gewöhnlich kleinere Standardabweichungen als die beiden einzelnen Verteilungen. Abbildung 5.4 zeigt die Verteilungen aus Abbildung 5.3 auf der vorherigen Seite zusammen mit dem Merge-Ergebnis.

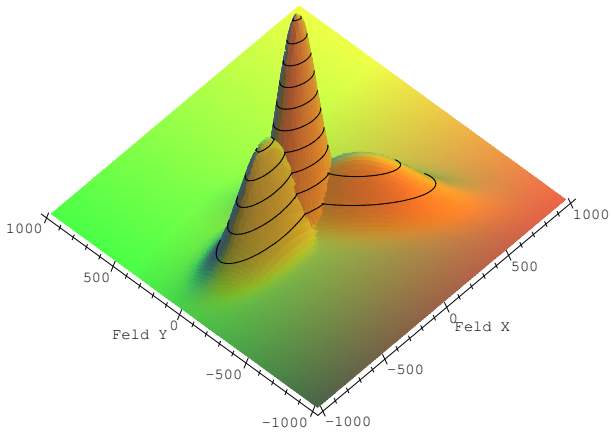

Abbildung 5.4: Verteilungen aus 5.3 und „gemergte“ Verteilung. Diese Verteilung liegt näher an der richtigen Ballposition im Ursprung und hat eine geringere Standardabweichung

Dieses „Merge“-Verfahren eignet sich für die Ballmodellierung besonders gut, da es assoziativ ist und es somit möglich ist, die Normalverteilungen der von verschiedenen Robotern beobachteten Ballpositionen nacheinander zu „mergen“.

5.4 Berechnung einer verbesserten Ballposition

Mithilfe der oben beschriebenen Normalverteilungen und des „Mergens“ wurde nun eine neue, exaktere Ballposition ermittelt.

Dazu wurde folgender Algorithmus entwickelt:

1. Für jedes, d.h. für das eigene und die übermittelten, BallPercept wird festgestellt, ob es nicht älter als 2 Sekunden ist.

2. Alle BallPercepts, die die Ballpostion relativ zum Roboter angeben und die Bedingung in 1 erfüllen, werden in Ballpositionen auf dem Feld umgerechnet.

   In Abbildung 5.5 auf der nächsten Seite sind die Percepts als gepunktete Linie vom Roboter zur Ballposition dargestellt.

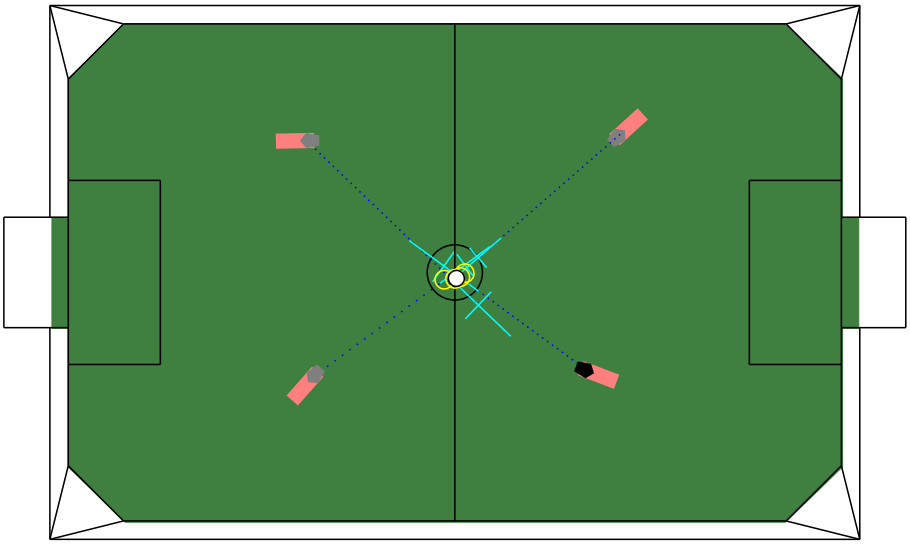

Abbildung 5.5: Debugdrawing eines Beispiels zur Sensorfusion beim Ball. vier Roboter beobachten einen Ball auf dem Mittelpunkt des Feldes. Sie sehen den Ball in unterschiedlichen Richtungen und Entfernungen (gepunktete Linie) woraus sich vier Verteilungen (Kreuze) und Nach dem Mergen eine neue Ballposition (weißer Kreis) ergeben.

3. Für diese Positionen wird nun eine Normalverteilung aufgestellt. Für σ_{min} und σ_{maj} wurde dazu eine Tabelle angelegt, in welche die für verschiedene Entfernungen experimentell ermittelten Werte eingetragen wurden. (Tabelle 5.1) Die aus dieser Tabelle für eine gewisse Entfernung abgelesenen σ_{min} und σ_{maj} werden dann mit dem Kehrwert der Validität der Roboterpositionen multipliziert, um den Einfluss einer schlechten Lokalisierung zu berücksichtigen. Der Drehwinkel für die Rotationsmatrix R ist der Winkel zwischen Roboter und der X-Achse des Feldes.

   Im Beispiel 5.5 sind diese Verteilungen als Kreuze in σ_{min} und σ_{maj} Richtung dargestellt.

Entfernungsbereich cm	σ_{min} cm	σ_{maj} cm
0 - 20	1,8	8,2
20 - 40	3,5	7,0
40 - 60	5,6	8,4
60 - 80	5,9	10,5
80 - 100	7,0	15,4
100 - 120	7,7	18,2
120 - 200	8,8	21,0
200 - 220	10,5	31,5
220 - 240	14,0	35,0
240 - 260	14,0	38,5
$>$ 260	14,0	42,0

Tabelle 5.1: Experimentell ermittelte σ_{min} und σ_{maj} für verschiedene Entfernungen.

4. Es wird eine History für die letzten fünf berechneten resultierenden Verteilungen verwaltet. Die σ_{min} und σ_{maj} Werte werden dabei mit 1,1 multipliziert, da mit

der Zeit die Wahrscheinlichkeit geringer wird, dass der Ball noch immer an diesem Punkt ist. Die Multiplikation mit 1,1 führt dazu, dass σ_{min} und σ_{maj} nach ca. 7 Durchläufen verdoppelt hat ($(1,1)^7 = 1,95$).

Die History-Positionen sind in 5.5 auf der vorherigen Seite als nicht gefüllte Kreise dargestellt.

5. Alle Gaussverteilungen der Positionen und der History werden gemergt.

6. Der Erwartungswert der resultierenden Verteilung wird als neue Ballposition verwendet und der History hinzugefügt. Außerdem bekommt die Verhaltenssteuerung noch die Möglichkeit, auf die Covarianz-Matrix zuzugreifen, damit diese für das Potenzialfeld nicht neu aufgestellt werden muss.

   Dieses Endergebnis ist der gefüllte, schwarz umrandete Kreis in 5.5 auf der vorherigen Seite.

Nach dem Umbau der Module (siehe Kapitel 5.1 auf Seite 49) wurde die Umrechnung aus Punkt 2 auf Seite 53 in dem BallLocator-Modul erledigt und die daraus resultierenden Ballpositionen über WLAN übermittelt. Die restlichen Schritte des Algorithmus wurden dann in dem neu geschaffenen GaussBellTeamBallLocator durchgeführt und als communicatedBallPosition zurückgegeben.

5.5 Fazit

Durch die Verwendung der Sensorfusion konnte die Qualität der Ballposition doch deutlich verbessert werden.

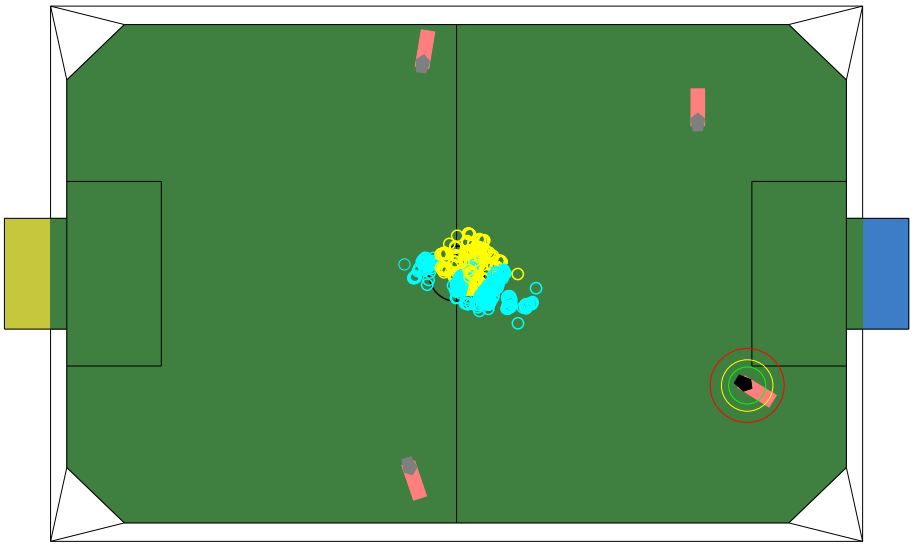

Abbildung 5.6: Ballverteilungen ohne (blau/dunkel) und mit (gelb/hell) Sensorfusion. Der Ball liegt in der Mitte des Feldes.

Um das zu zeigen, wurde wiederum ein Debug-Drawing eingebaut, welches die ermittelten Ballpositionen anzeigt (Abbildung 5.6). Die vom BallLocator-Modul übergebenen

Ballpositionen werden hier als blaue (dunklere) Kreise dargestellt. Die gelben (hellere) Kreise geben die durch die Sensorfusion ermittelten Ballpositionen an. Man sieht, dass die Abweichung der Sensorfusion-Ballpositionen wesentlich geringer ist als die ohne.

Allerdings ist es notwendig, das Verhalten auf den BallLocator anzupassen. Es muss z.B. zwischen gesehener und kommunizierter Position unterschieden werden. Außerdem muss das Verhalten z.B. durch häufiges Suchen nach Landmarken dafür sorgen, dass die Validität der Roboter-Positionen nicht zu gering wird, da sonst keine sinnvolle Sensorfusion möglich ist.

Der Algorithmus ist außerdem sehr schnell. Es wird für einen Durchlauf maximal 8 mal (3 mal für die Daten der anderen Roboter und 5 mal für die History) der Merge-Vorgang aufgerufen. Der einzelne Merge-Vorgang beinhaltet dabei nur Additionen und Multiplikationen, aber keine Schleifen. Die mit RobotControl gemessene Zeit für einen Durchlauf liegt unter einer Millisekunde.

Kapitel 6

Sensorfusion zur Spielermodellierung

Bei der sensorfusionierten Spielererkennung gibt es zwei große Teile: Einmal den Bereich der Bildverarbeitung, in dem aus dem Kamerabild des Roboters gewonnene Eindrücke dahingehend verarbeitet werden, Gegner oder Freunde, also Spieler generell zu erkennen. Diese Daten werden dann als so genannte PlayerPercepts gesammelt. Dabei sind PlayerPercepts eine spezielle Datenstruktur, die alle nötigen Informationen (wie Position, Entfernung, Wahrscheinlichkeiten dieser Werte, etc.) zu einem erkannten Spieler beinhaltet. Alle PlayerPercepts, auch die von anderen Robotern, werden dann zum zweiten Bereich, der Spielermodellierung, geschickt. Dort werden diese umgewandelt und umgerechnet, den einzelnen Spielern zugeordnet und fusioniert. In diesem Kapitel geht es um die Spielermodellierung. Zuerst wird auf die bisherige Spielermodellierung eingegangen. Danach folgen Erklärungen zu den neu entwickelten Algorithmen, die anschließend miteinander verglichen werden. Eine Beschreibung zur Bildverarbeitung findet sich im Kapitel 3.3.2 auf Seite 31.

6.1 Stand zum Beginn der Projektgruppe

Die bisherige Spielererkennung war sehr ungenau und lieferte zudem stark springende Ergebnisse. Als „Percept-Verarbeiter“ existierte der GT2001PlayersLocator. Dieser ist eine Solution des Moduls PlayersLocator (siehe auch Kapitel 2.4.2 auf Seite 13), die allerdings zu schlicht gehalten war. Grundsätzlich ist seine Eingabe auf die eigene Wahrnehmung beschränkt und bekommt als zusätzliche Information nur die per Teamkommunikation übermittelten Positionen seiner Mitspieler.

Der GT2001PlayersLocator verwendet eine Karte mit einem Gitter, auf der Punkte eingetragen und je nach Alter und/oder Validität ein- oder aussortiert werden. Beim Hinzufügen eines Punktes werden die Verteilungswerte so aktualisiert, dass der Punkt selbst einen um 2 erhöhten und seine waagerechten und vertikalen Nachbarn auf der Karte einen um 1 erhöhten Wert zugeordnet bekommen. Dadurch bilden sich in dem Array der Verteilungswerte so genannte „Hügel“, die somit ein Potenzialfeld ausprägen (siehe Abbildung 6.1 auf der nächsten Seite). Ihre Spitzen werden durch Berechnung des Maximums über die Verteilungswerte ermittelt, in eine PlayerPose umgewandelt und

der PlayerPoseCollection hinzugefügt. Zu den beiden Datenstrukturen aber später Genaueres.

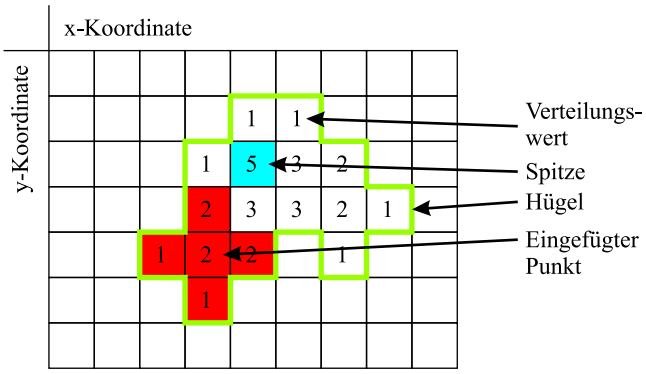

Abbildung 6.1: Karte der Verteilungswerte beim GT2001PlayersLocator

Das Funktionsprinzip des GT2001PlayersLocator ähnelt also dem Konzept des „Map Clustering“. Es berücksichtigte aber nicht die damit auftretenden Problematiken sensorfusionierter Daten wie z.B. eine Beachtung der Percept-Herkunft (mehr dazu im Kapitel 6.3.2 auf Seite 61). Aufgrund der ungenauen, sprunghaften Eingabewerte und ohne Mittel zur Kompensation dieser Sprünge scheiterte das damalige Konzept der Spielererkennung. Es wurde nur die Freunderkennung weiter verwendet, da ihre Daten auf der sehr gut funktionierenden Selbsterkennung der einzelnen Roboter basieren. Aus diesem Grund wurde nach einem neuen oder verbesserten Konzept der Gegnererkennung gesucht.

6.2 Ziel der Sensorfusion zur Spielermodellierung

Es sollten Teile der Spielererkennung, also sowohl die Bildverarbeitung (siehe Kapitel 3.3.2 auf Seite 31) als auch die Spielermodellierung, verbessert, bzw. neu aufgebaut werden. Das Ergebnis sollte insoweit zufrieden stellend sein, als dass man durch die genauere Positionsbestimmung neue Möglichkeiten bei dem Verhalten bekommt, z.B. bessere Interaktion unter den Mitspielern.

6.3 Umsetzung der Sensorfusion zur Spielermodellierung

Alle neuen Konzepte wurden auf der Basis des GT2001PlayersLocators verwirklicht. In der neuen Klasse wurden alle alten Bereiche erweitert und verbessert.

Die verwendeten Datenstrukturen wurden wieder verwertet, verändert und neue wurden ergänzt: Grundsätzlich werden die Daten als PlayersPercepts und Robotposes (Daten des eigenen Roboters) eingegeben und in Instanzen der Klasse PlayerCollection eingefügt. Mit dieser werden dann alle Arbeitsschritte durchgeführt und abgespeichert. Zum Schluss werden die Instanzen der PlayerCollection in die Instanzen PlayerPose

umgewandelt und der PlayerPoseCollection hinzugefügt. Letztere wird dann von den Klassen der Verhaltenssteuerung weiterverarbeitet.

Während vorher von den eingegebenen PlayersPercepts nur die umgerechneten (x, y)-Koordinate verarbeitet wurden, kamen nun noch die Wahrscheinlichkeit für diese Werte und die sich ergebenden Standardabweichungen σ_{Maj} und σ_{Min} zur Erstellung einer ellipsenförmigen Gaussglocke (siehe auch bei der Ballsensorfusion in Kapitel 5.3.2 auf Seite 51) um diese Koordinaten hinzu. Damit konnten die Daten besser in die Struktur um die Gaussglocken-Datenstruktur eingebettet werden. Des weiteren gibt es ein Array source, das die Herkunft des „PlayerPercepts“ allgemein abspeichert.

Neue Konstanten machten die Klasse kompatibler gegenüber späteren Änderungen. Zur Unterdrückung von Sprüngen eingegebener Percept-Daten wurde zudem eine History-Datenstruktur eingeführt, die die Ergebnisse des letzten Durchlaufs durch den PlayersLocator speicherte und in der Neuverarbeitung wieder hinzumischte. Außerdem gab es für die einzelnen Verteilungskonzepte entsprechende Ergänzungen.

Die Methode zum Empfangen eigener und externer Daten wurde umgestaltet und erweitert. Während man sich bisher auf eine beschränkte Eingabe verließ, wurde jetzt ein Verwaltungsbereich eingebaut, der zu den eigenen und den über WLAN erhaltenen Daten auch History-Daten verarbeitete. Außerdem bestand nun die Möglichkeit, mehr als nur 4 Mitspieler- und 4 Gegner-Percepts zu behandeln. Empfangen werden als Gegner-Percepts alle von den Mitspielern übertragenen, die eigenen und die History-Daten. Als Mitspieler-Percepts werden die von den Mitspielern übertragenen Selbstlokalisierungsdaten, die eigenen und die Historydaten eingesetzt.

Bei dem Verarbeiten dieser Erweiterungen wurde auch der Bereich zur Lokalisierung, die Methode locatePlayers, stark ergänzt und an die neuen Bedürfnisse angepasst. Dabei wird das Mischen der neu gewonnenen Informationen nach der Vorbereitung durch die Zuordnungsalgorithmen von der neuen Gaussglocken-Datenstruktur übernommen. Dazu wurden drei verschiedene Verfahren implementiert und in ihrer Leistungsfähigkeit verglichen (siehe Kapitel 6.3.5 auf Seite 66):

1. Map Clustering (Kapitel 6.3.2 auf Seite 61)
2. Ellipsenschnitt (Kapitel 6.3.3 auf Seite 62)
3. Paarweise Zuordnung(Kapitel 6.3.4 auf Seite 64)

Nach einer automatischen Fehlerkorrektur, bzw. Anpassung einiger Werte wie Koordinaten außerhalb des Spielfeldes, exorbitante Ellipsenradien, etc. werden die verarbeiteten, neuen Daten der PlayerCollection sowohl in die History für den nächsten Durchlauf, als auch als PlayerPoses umgewandelt in die PlayerPoseCollection zur Verarbeitung an andere Klassen weitergegeben.

Um herausfiltern zu können, wenn kommunizierte Percepts den „lokalen“ Roboter enthalten, wird bei der Lokalisierung von Mitspielern die eigene selbstlokalisierte Position mit in die Algorithmen aufgenommen. Nach Durchlaufen des jeweiligen Algorithmus werden alle Ergebnisse verworfen, mit denen diese Position zusammengefasst wurde. Dies ist notwendig, damit ein Roboter sich nicht selbst in die PlayerPoseCollection einträgt.

6.3.1 Problem bei der Sensorfusion

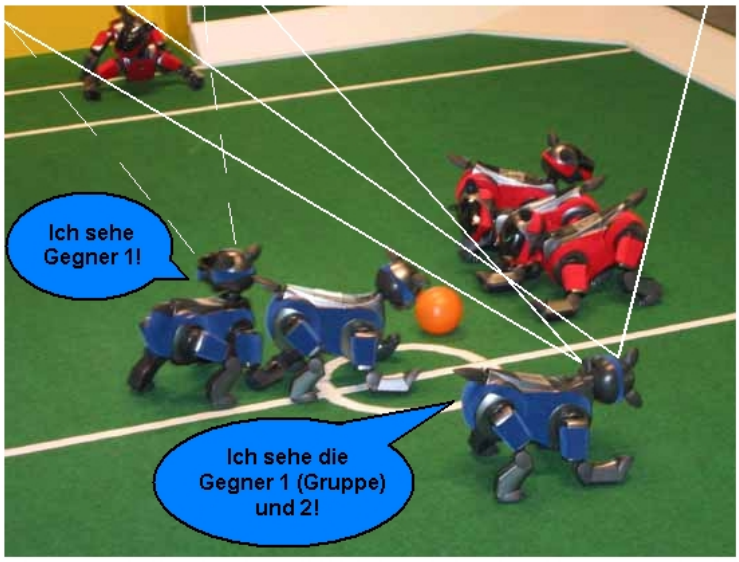

Abbildung 6.2: Zuordnungsproblem im Spiel

Grundsätzlich hat man bei der Nutzung von Sensorfusion zur Spielermodellierung das Problem, dass die Roboter zwar Gegner und Mitspieler erkennen können, die daraus gewonnenen PlayerPercepts aber mit einer eigenen, lokalen Indizierung versehen. Also sieht beispielsweise Roboter 1 Gegner 1, 2 und 3 ebenso wie Roboter 2. Nur ist Gegner 1 vom Roboter 1 beim Roboter 2 Gegner 2 (anderes Beispiel siehe auch Abbildung 6.3).

Roboter / Percept	Aibo 1	Aibo 2	Aibo 3	Aibo 4
Gegner 1	1	3	0	2
Gegner 2	0	2	0	4
Gegner 3	2	1	0	1
Gegner 4	0	0	1	3
Gegner 5 ?	0	0	2	0

Abbildung 6.3: Verschiedene Nummerierungen der erkannten Gegner bei verschiedenen Mitspielern

Hierbei sind diverse Permutationen möglich. Es geht also darum ein vernünftiges Verfahren zu finden, das nicht naiv alle Permutationen durchgeht und per Durchschnittsberechnung die Positionen einem Spieler zuordnet. Am Rande sei aber noch bemerkt, dass ein Roboter wesentlich mehr als 4 Spieler pro Farbe erkennen kann – die Bilderverarbeitung generiert bis zu 17 PlayerPercepts. Bei einer großen Anzahl zu verarbeitender Percepts wird die Permutationszahl in die Höhe getrieben und die Zuordnung zu kleiner gleich 4 Spielern erschwert.

6.3.2 Map Clustering

Das Map Clustering orientiert sich an den klassischen Clustering-Methoden wie beispielsweise dem „K-Mean Clustering“ (Zugehörigkeitsfindung zu K Zentroiden, bzw. Cluster mit Mittelpunkt)[12], dem „Hierarchical Clustering“ (Punktmengen bezüglich Grenzwert vereinigen.)[13] oder der „Diversity Selection“ (Entferne vielverbundene Objekte, der Rest sind dann Zentroide)[14]. Es wurden einige Ideen übernommen und zu folgendem Algorithmus zusammengelegt:

Beim Clustering werden alle Daten der PlayerCollection, bzw. hier die Koordinaten informal auf einer Karte eingetragen und bezüglich der Einzelabstände möglichst günstig zu Clustern zusammengefasst. Dabei wird Depth-First-Search zur Clusterfindung verwendet. Ein beispielhafter Ablauf des Algorithmus ist in Abbildung 6.4 zu sehen.

Am Anfang stehen alle Daten, d.h. die Positionen in Form von Instanzen der PlayerCollection, in der Array-Datenstruktur PerceptInput. Diese wird informal als eine Karte mit Punkten verstanden. Dabei ist die Karte das Spielfeld mit dem entsprechenden Koordinatensystem und die Punkte sind die umgerechneten Koordinaten einer Instanz der PlayerCollection. Eine Nummerierung der Punkte ist durch die Arrayindizes von PerceptInput bereits gegeben.

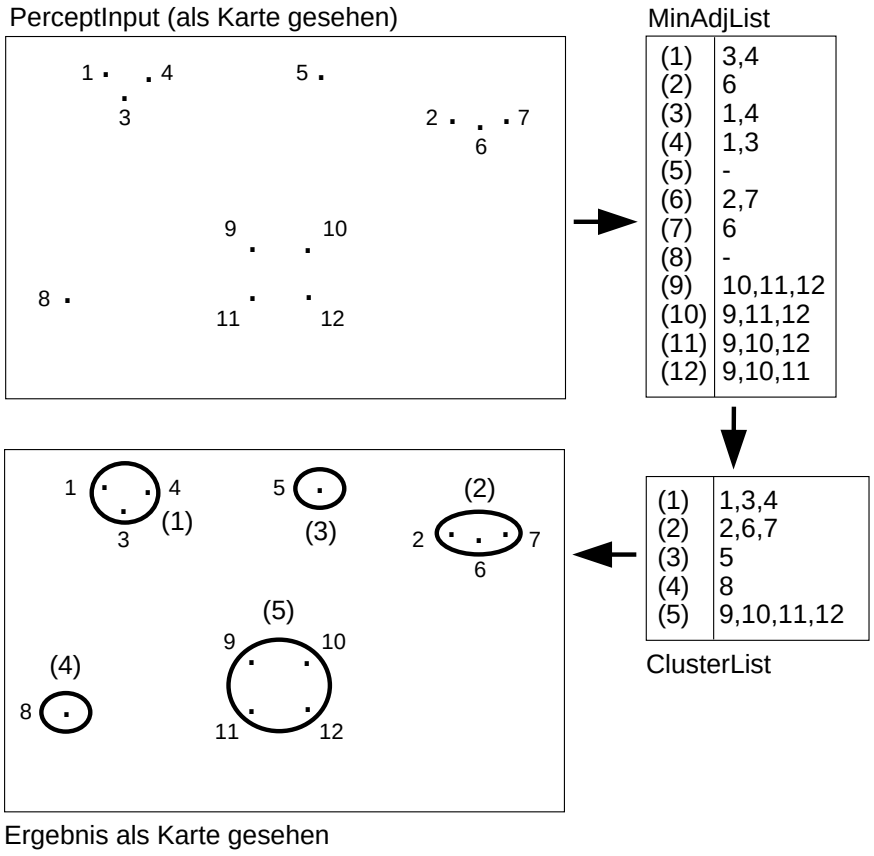

Abbildung 6.4: Ablauf des Map Clusterings

Im ersten Verarbeitungsschritt werden nun nacheinander der Abstand jedes Punktes zu

jedem anderen berechnet und mit einem Wert maxDistance verglichen. Es wird Punkt 1 mit 2, 3, 4, . . . , dann Punkt 2 mit 3, 4, 5, . . . usw. verglichen. Dadurch entsteht ein zur Eingabelänge (Anzahl der Positionen) quadratischer Aufwand. maxDistance entspricht dabei einem Vielfachen (in der Regel dem Zweifachen) der Länge eines AIBOs (120 mm). Aus Geschwindigkeitsgründen wird die Wurzelberechnung bei der Distanzformel weggelassen, so dass obiger Wert noch quadriert wird. Sollte nun der Abstand kleiner als maxDistance sein, so werden die Arrayindizes beider Punkte in die so genannte minimale Adjazenzliste MinAdjList eingetragen. Auch in der MinAdjList entsprechen die Arrayindizes einer Percept-Nummerierung. Wenn beispielsweise die Punkte x und y nahe beieinander liegen, so wird in die Adjazenzliste von Punkt x der Index y und bei Punkt y der Index x angefügt. Damit wird zuerst einmal eine Nachbarschaftszuordnung aufgestellt.

Der zweite Schritt durchläuft dann die MinAdjList per Tiefensuche und trägt dabei alle zu einem Nachbarschaftbereich (Cluster) gehörenden Punkte in der ClusterList ein. Diese ist ebenfalls eine Adjazenzliste, bei der allerdings die Arrayindizes einer Cluster-Nummerierung entsprechen. Es werden also bei Index A alle zum Cluster A gehörenden Punkte eingetragen. Auch wenn zu einem Cluster B nur ein Punkt gehört, so wird C zunächst als ein Cluster gewertet. Damit die Adjazenzlisten nicht mehrfach durchlaufen werden, wird immer ein Visited-Flag auf true gesetzt. Schließlich wird nach nochmals maximal quadratischen Aufwand eine Auflistung aller Cluster ermittelt.

Im letzten Schritt wird nun der Mittelpunkt der Clustereinträge berechnet und als Koordinaten eines erkannten Spielers in die tmpPlayerCollection abgespeichert. Diese wird dann in die History und über PlayerPoses in die PlayerPoseCollection kopiert. Dabei werden auch die Anzahl der Sources pro Cluster berücksichtigt, so dass bei mehr als 4 Clustern keine Phantome mitgeführt werden. Phantome sind in der Regel (aber nicht immer!) Cluster mit nur 1 Punkt. Um Phantome sicher zu entdecken, wird auch die Source-Herkunft geprüft. Wenn es sich bei dem einem Punkt nicht um einen Historyeintrag und nicht um selbst lokalisierte Daten handelt, liegt höchstwahrscheinlich ein Phantom vor. Auch wenn dem nicht so ist, wie zum Beispiel wenn bei insgesamt 4 Spielern zwei rote AIBOs zwei verschiedene, blaue Gegner sehen, so ist es günstiger auch diese korrekten Cluster als Phantome abzutun. Da die Spielererkennung schließlich nur zur Ergänzung und Verbesserung der eigentlichen Wahrnehmungsverarbeitung da ist, spielt es keine große Rolle, wenn lieber weniger, aber dafür richtigere Daten ausgeben werden.

6.3.3 Ellipsenschnitt

Dieses Verfahren legt um alle aus Percepts generierten Positionen Ellipsen, deren Radien aus den Standardabweichungen von Gaussverteilungen (siehe auch Kapitel 5.3.2 auf Seite 51) berechnet werden. Ebenso werden die History-Daten und die Daten der Selbstlokalisierung als Ellipsen interpretiert. Um eine Zuordnung von Ellipsen zu Spielern zu bekommen, prüft der Algorithmus, ob jeweils zwei Ellipsen eine gemeinsame Fläche haben, sich also schneiden oder ineinander liegen. Sollte dies der Fall sein, wird angenommen, dass es sich um den selben Spieler handelt und die beiden zugehörigen Gaussverteilungen werden dann genau wie bei der Ballmodellierung (siehe Kapitel 5.3.3 auf Seite 51) gemergt. Aus der dabei entstehenden Gaussverteilung wird eine neue Ellipse generiert, die dann später eventuell mit anderen Ellipsen zusammengefasst wird.

Zwei sich schneidende oder ineinander liegende Ellipsen werden nur zusammengefasst, wenn sie aus unterschiedlichen Quellen stammen. Mit Quellen ist dabei die Herkunft der Daten gemeint. Es gibt 6 verschiedene Quellen: History-Daten, Selbstlokalisierungsdaten, von der Bildverarbeitung generierte PlayerPercepts und dreimal die von jeweils einem der drei Mitspieler gesendeten PlayerPercepts. Eine nach dem Mergevorgang generierte Ellipse bekommt beide Ursprungs-Quellen als neue Quelle vermerkt, so dass Ellipsen nur zusammengefasst werden, falls die Menge der Quellen beider Ellipsen disjunkt ist.

Die Überprüfung, ob zwei Ellipsen eine gemeinsame Fläche besitzen, unterscheidet zwischen folgenden Fällen:

1. Haben die Ellipsenmittelpunkte einen Abstand, der größer als die Summe der beiden größeren Radien ist, gibt es keine gemeinsame Fläche.

2. Haben die Ellipsenmittelpunkte einen Abstand, der kleiner als die Summe der beiden kleineren Radien ist, gibt es eine gemeinsame Fläche.

3. Liegt der Abstand der Ellipsenmittelpunkte zwischen den beiden genannten Summen, ist eine Aussage über eine gemeinsame Fläche nicht ohne weitere Betrachtung möglich. Leider lässt sich eine exakte Betrachtung nicht effizient durchführen, so dass hier nur eine Näherung verwendet wird. Die Ellipsen werden in diesem Fall als Rechtecke betrachtet, deren Seitenlängen jeweils den doppelten Radien entsprechen (siehe Abbildung 6.5). Dadurch ergibt sich ein größerer Bereich, der als zur Ellipse gehörig angesehen wird. Hierdurch können zwei Ellipsen als sich schneidend erkannt werden, auch wenn sie eigentlich keine Schnittfläche besitzen.

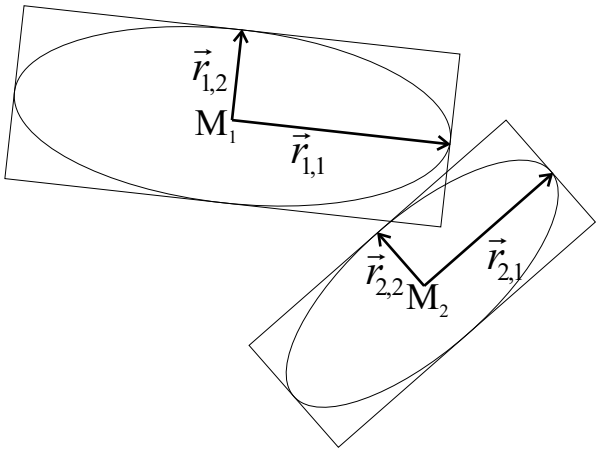

Abbildung 6.5: Interpretation von Ellipsen als Rechtecke

Wenn die zwei Rechtecke sich schneiden, so gibt es einen gemeinsamen Randpunkt. Liegen die Rechtecke ineinander, so liegt auch der Mittelpunkt des einen Rechtecks innerhalb des anderen. Um zu überprüfen, ob es eine gemeinsame Fläche gibt, genügt es daher, Gleichung 6.1 darauf zu untersuchen, ob die variablen Vorfaktoren $v_{i,j}$ bestimmte Bedingungen erfüllen.

$$\vec{m}_1 + v_{1,1} * \vec{r}_{1,1} + v_{1,2} * \vec{r}_{1,2} = \vec{m}_2 + v_{2,1} * \vec{r}_{2,1} + v_{2,2} * \vec{r}_{2,2} \tag{6.1}$$

Dabei bezeichnen $\vec{m}_i$ den Mittelpunkt der Ellipse bzw. des Rechtecks i und $\vec{r}_{i,1}$ und $\vec{r}_{i,2}$ die Radien der Ellipse bzw. des Rechtecks i. Die zu überprüfenden Bedingungen sind folgende:

(a) $\forall i, j \in \{1, 2\} : -1 \leq v_{i,j} \leq 1$ ist auf jeden Fall erfüllt, falls die Rechtecke gemeinsame Punkte haben.

(b) Wenn $\exists i \in \{1, 2\} : v_{i,1} = v_{i,2} = 0$ erfüllt ist, liegt ein Mittelpunkt innerhalb des anderen Rechtecks.

(c) Wenn $\exists i, j \in \{1, 2\} : |v_{1,i}| = |v_{2,j}| = 1$ erfüllt ist, haben die Rechtecke einen gemeinsamen Randpunkt.

Da die Vektoren in Gleichung 6.1 auf der vorherigen Seite zweidimensional sind, kann man die Gleichung auch koordinatenweise als Gleichungssystem betrachten. Da dass Gleichungssystem dann 4 Veränderliche aber nur 2 Gleichungen besitzt, ist es nicht exakt lösbar. Dies ist hier aber auch gar nicht nötig, da nur interessant ist, ob eine Lösung existiert. Um dies zu überprüfen, helfen die oben genannten Bedingungen weiter. Aus den Bedingungen 3b und 3c lassen sich 18 mögliche Kombinationen für eine Vorbelegung von 2 der 4 Vorfaktoren ermitteln. Damit lassen sich dann 18 Gleichungssysteme mit 2 Veränderlichen und 2 Gleichungen erstellen. Es kann dann untersucht werden, ob eine Lösung von mindestens einem dieser Gleichungssysteme Bedingung 3a erfüllt. Ist dies der Fall, so wird angenommen, dass die Ellipsen sich schneiden.

Sollte beim Versuch, diese Gleichungssysteme zu lösen, festgestellt werden, dass unendlich viele Lösungen existieren, muss weiter überprüft werden, ob eine der existierenden Lösungen Bedingung 3a erfüllt. Dazu wird je einer der beiden Variablen mit -1 und 1 vorbelegt, um dann zu sehen, ob die andere Variable Bedingung 3a erfüllt.

Diese Überprüfungen werden für alle Paare von Ellipsen durchgeführt. Wird dabei festgestellt, dass sich zwei Ellipsen schneiden, werden diese zusammengefasst. Findet sich zu einer Ellipse keine andere Ellipse, mit der sie eine gemeinsame Fläche hat, so wird sie ohne Änderung übernommen. Für die so ermittelten Ellipsen werden die Überprüfungen wiederholt, bis sich kein Ellipsenpaar mit einer gemeinsamen Fläche mehr findet.

Das dabei ermittelte Ergebnis wird dann nicht nur in PlayerPoses umgewandelt und in die PlayerPoseCollection gespeichert, sondern auch in der History gespeichert, die dann beim nächsten Durchlauf mit vergrößerten Radien – das entspricht einer größeren Standardabweichung bei den zugehörigen Gaussverteilungen – wieder mit in die Berechnung eingehen. Dies soll die Positionen glätten und Sprünge in den Percepts filtern.

6.3.4 Paarweise Zuordnung

Die Paarweise Zuordnung löst ein Optimierungsproblem, bei dessen Rechnung lokale Optima zur Gesamtlösung kombiniert werden. Dabei wird ein lokales Optimum immer zwischen zwei Percepts liefernden Robotern berechnet. Eine Berechnung des globalen Opti-

mums, bei dem die Percepts aller Roboter in einem Schritt verglichen werden, wäre zu ineffizient.

Gegeben:

Percept \ Roboter	Aibo 1	Aibo 2
Gegner 1	(10,20)	-
Gegner 2	-	(180,-650)
Gegner 3	-	-
Gegner 4	(-45,1670)	(11,21)

Gesucht:

Percept \ Roboter	Aibo 1	Aibo 2
Gegner 1	(10,20)	(11,21)
Gegner 2	-	(180,-650)
Gegner 3	(-45,1670)	-
Gegner 4	-	-

Abbildung 6.6: Beispiel einer paarweisen Zuordnung

Wie schon erwähnt können maximal 17 Spieler pro AIBO erkannt werden. Die maximale Anzahl von Spielern pro Mannschaft ist aber wesentlich kleiner, nämlich 4. Für die verwendeten Datenstrukturen ist die Arraygröße durch das Maximum aus den Anzahlen der erkannten Spieler der AIBOs und der Spieler pro Mannschaft gegeben.

Als Datenstrukturen gibt es zwei Schichten. In der ersten Schicht stehen die beiden Arrays A[] und B[], die von der PlayerCollection nur die Koordinaten zum späteren Distanzvergleich beinhalten. Als Indexarray dienen ihnen searchedPermPairA[], searchedPermPairB[], permOfA[] und permOfB[]. Die ersten beiden werden als Speicher für Ein- und Ausgabe benutzt und beinhalten Daten in der weiter unten erklärten Permutationsschreibweise. Das selbe speichern auch die Arrays permOfA[] und permOfB[]. Sie werden im Algorithmus als Zwischenspeicher benutzt, d.h. kurzzeitige Änderungen und Arbeitsschritte werden in ihnen ausgeführt.

Die Permutationsschreibweise beschreibt am Besten ein Beispiel: AIBO A hat 3, AIBO B 2 Percepts. Die errechnete Arraygröße ist 5. Anfangs soll dann in searchedPermPairA[] und permOfA[] die Permutation (0,0,1,2,3) und in searchedPermPairB[] und codepermOfB[] (0,0,0,1,2) stehen. Dabei entspricht jede Position in der Permutationsschreibweise einem Arrayplatz; vorne Index 0, hinten 4. Die einzelnen Ziffern in der Permutation stellen jeweils den Index für eine Position dar, wobei eine 0 für eine nicht existierende Position steht.

Der Algorithmus besteht nun aus einer Schleife, in der nun alle Permutationen von AIBO B durchgegangen und mit A verglichen werden. Das heißt, (0,0,1,2,3) wird mit (2,1,0,0,0), (2,0,1,0,0), (2,0,0,1,0), (2,0,0,0,1), (1,2,0,0,0) usw. nacheinander kombiniert. Der Vergleich findet dabei immer zwischen zwei Positionen gleichen Index statt: Bei (0,0,1,2,3) und (2,1,0,0,0) werden (0,2), (0,1), (1,0), (2,0) und (3,0) gegenübergestellt. Da diese beiden Ziffern dem Index eines Arrayplatzes in den Arrays Arrays A[] und B[] entsprechen, können nun hier die Koordinaten abgerufen und ihre Distanz berechnet werden. Wird mit einer 0 verglichen ist die Distanz mit der Konstanten distanceSinglePercept 60000 und bei zwei 0en mit distanceNoPercepts 120000. Dabei kommen die hohen Zahlen von der beschleunigten

Abstandsberechnung ohne Wurzelziehen.

Die berechneten Distanzen, in unserem Beispiel 5, werden nun zum so genannten Vergleichswert der Permutationen zusammenaddiert (siehe Formel 6.2).

$$\begin{aligned} \text{Vergleichswert} = {} & \text{dist}(P_{1,1}, P_{2,1}) + \text{dist}(P_{1,2}, P_{2,2}) + \text{dist}(P_{1,3}, P_{2,3}) + \text{dist}(P_{1,4}, P_{2,4}) \\ & \text{mit } P_{i,j} = \text{Gegner } j \text{ von Aibo } i \\ & \text{und } \text{dist}(a, b) = (a_x - b_x)^2 + (a_y + b_y)^2 \end{aligned} \tag{6.2}$$

Der Vergleichswert stellt eine die Optimalität der zwei gerade betrachteten Permutationen beurteilende Zahl dar. Je kleiner, umso besser, es gilt also, mit jedem Durchlauf den kleinsten Vergleichswert zu finden. Wenn nacheinander alle Permutationen durchgegangen werden, wird immer der aktuelle Vergleichswert totalDistance mit den gerade vorhandenen Optimum minimalDistance verglichen. minimalDistance hat am Anfang einen Wert, der größer ist, als alle durch die Formel errechenbaren. Sollte totalDistance kleiner sein, so wird die aktuelle Permutationskonstellation in searchedPermPairA[] und searchedPermPairB[] abgespeichert.

Schlussendlich werden mit den ermittelten Indexarrayes die entsprechenden Daten aus der PlayerCollection extrahiert und in die tmpPlayerCollection kopiert. Von dort aus wandern sie dann in die History und über PlayerPoses in die PlayerPoseCollection.

6.3.5 Vergleich der Zuordnungsverfahren

Da die Effizienz der verschiedenen Algorithmen nicht exakt messbar war, wurden die Verfahren empirisch miteinander verglichen.

Die Paarweise Zuordnung war sehr schnell, da sie im worst case nur in der Eingabelänge linear viele Vergleichsoperationen auszuführen hatte (bei je 4 Gegnerpercepts pro AIBO insgesamt $24 + 24 + 24 = 72$ Vergleiche). Allerdings ließ ihre Genauigkeit zu wünschen übrig, da jeweils ein nur lokales Optimum berechnet wurde, das unter Umständen auch falsch sein konnte. Eine Berechnung des globalen Optimums hätte bei je 4 Gegnerpercepts pro AIBO insgesamt $24*24*24 = 13824$ Vergleiche, also polynomiellen Aufwand erfordert.

Das Map Clustering war nicht so schnell wie die Paarweise Zuordnung, da es im worst case in der Eingabelänge quadratische Laufzeit hatte. Aber da hier nicht die Herkunft der PlayerPercepts berücksichtigt wurde, konnte es passieren, dass alle PlayerPercepts eines AIBOs als ein Spieler bewertet wurden. Interessant war die Tatsache, dass auch längere Punktketten zu einem Hindernis zusammengefasst wurden, was eigentlich eher als Vorteil, denn als Nachteil angesehen wurde.

Der Ellipsenschnitt war – bedingt durch die vielen Durchläufe durch die Spielerpositionen – der langsamste der implementierten Verfahren. Er hatte die folgende Art von Problem: Wenn zwei nebeneinander stehende AIBOs gemeinsam einen Gegner sahen, die Ellipsen sich jedoch nicht schnitten oder nicht als schneidend erkannt wurden, wurden aus einem, zwei Gegner gemacht. Dadurch können tatsächlich vorhandene Gegner aus dem Ergebnis verdrängt werden. Als Vorteil aber war es anzusehen, dass die verwendete Struktur nahe an der mehrfach verwendeten Gaussglocken-Datenstruktur lag und damit berück-

sichtigt wurde, dass die Richtung zu Objekten im Bild besser erkannt werden kann als die Entfernung (siehe auch Kapitel 5.3.1 auf Seite 50).

Im Gesamtvergleich und in Tests schnitt das Map Clustering als bester Algorithmus ab und wurde bevorzugt verwendet.

6.3.6 Test und Debugging

Unter den Debugausgaben gibt es verschiedene Varianten, die mit zu setzenden Konstanten einzeln ein- und ausgeschaltet werden können.

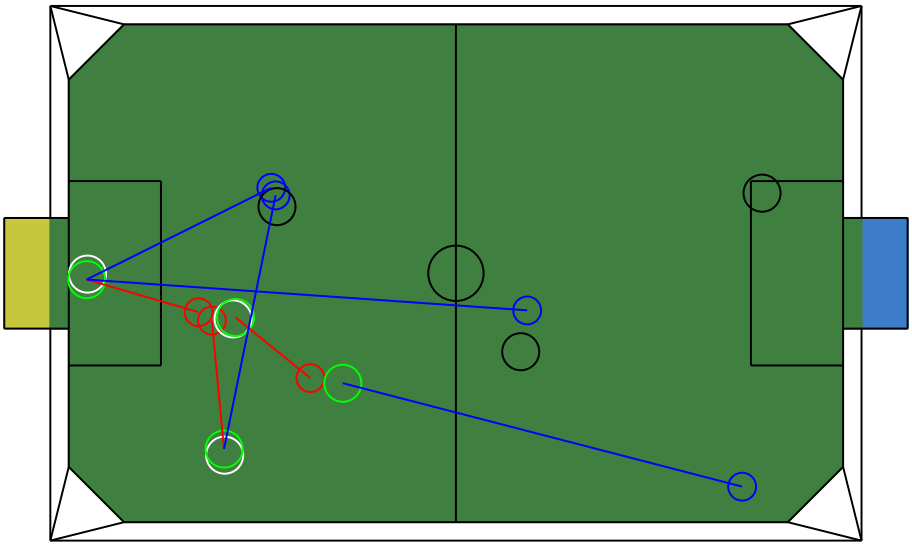

Abbildung 6.7: Debug-Drawings zur Sensorfusion der Spielermodellierung

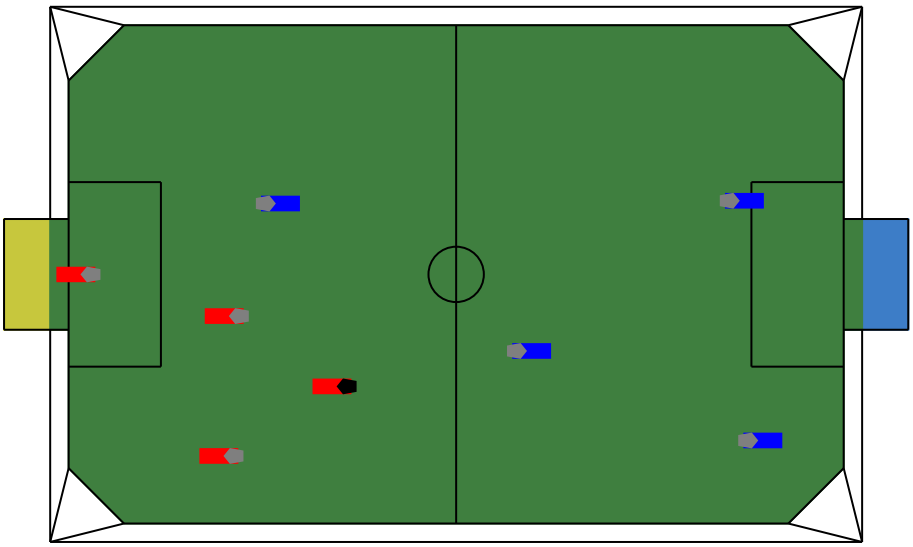

Abbildung 6.8: Worldstate zum Zeitpunkt des Debug-Drawings

Im Simulator kann bei View die Ansicht PlayersLocator angewählt werden. Wenn auch die Konstante DEBUGDRAWINGS auskommentiert wurde, so sieht man beim gestarteten

Spielverlauf auf dem Spielfeld zusätzliche Kreise und Linien. Abbildung 6.7 auf der vorherigen Seite zeigt eine Debugausgabe, die in der in Abbildung 6.8 auf der vorherigen Seite dargestellten Situation ausgegeben wurde. Das Zentrum eines Kreises dokumentiert die Position (Koordinaten) eines erkannten Spielers, seine Farbe symbolisiert die Art der Ursprungsdaten. Dabei werden gesehene Spieler in ihrer Trikotfarbe und mit einer Linie gezeigt, die von der Position des Roboters, der sie erkannt hat, ausgeht. Die selbstlokalisierten Positionen der Mitspieler werden in Grün (in der Abbildung Hellgrau) dargestellt. Die Historydaten werden in Schwarz (für blaue Spieler) und Weiß (für rote Spieler) ausgegeben. Es sei aber zu bedenken, dass die History die berechneten Daten des vorherigen Verarbeitungsschrittes anzeigt, alles andere sind direkt ausgegebene Eingabedaten.

Eine andere Variante ist die textuelle Debugausgabe in den MessageViewer (siehe Kapitel 2.4.5.2 auf Seite 18). Die hier ausgegebenen Daten sind von sehr kurzlebiger Natur und dienen eher oberflächlicher Beobachtung. Diese Ausgaben können mit dem Einkommentieren von OUTPUT_MESSAGES aktiviert werden.

Wirklich informativ dagegen sind die Ausgaben in eine extra angelegte Textdatei namens *SensorFusionPlayersLocator_Debugmessages_x.txt*. Dabei steht das x für eines der verwendeten Zuordnungsverfahren CM, PA, EC. Hier wird der gesamte Entscheidungsverlauf und die Inhalte der signifikanten Datenstrukturen dokumentiert. Ist man Fehlern auf der Spur so wird man hier die meisten und besten Informationen erhalten. Eine Aktivierung dieser Ausgabe wird mit dem kombinierten Einschalten der Konstanten USEDEBUGFILE und FILEOUTPUT_x erreicht. Auch hier ist x in der jeweiligen CPP-Datei nachzulesen, da die Bezeichnungen variieren.

6.4 Fazit

Das Problem der Verteilung ist also im Endeffekt nur eine wohl überlegte (und gut geprüfte) Auswahl eines geeigneten Verfahrens. Allerdings bleibt das Problem der Gegnererkennung aufgrund der Schwächen der optischen Sensoren und der Schwierigkeiten in der Bildverarbeitung bestehen. Solange dafür keine besseren Lösungen gefunden werden, bleibt das Konzept der Gegnererkennung mitsamt seiner strategischen Möglichkeiten leider in der Schublade.

Kapitel 7

Verhaltenssteuerung

Das folgende Kapitel ist in vier Teile unterteilt. Im ersten wird der bisherige Stand des GermanTeam-Codes erläutert, und ein Überblick über die zu treffenden Entscheidungen bei der Entwicklung der Verhaltenssteuerung vermittelt. Im zweiten Teil wird das Ergebnis des Diskussionsprozesses beschrieben und das Konzept dargestellt. Der dritte Teil, die Realisierung, beschreibt die tatsächliche Umsetzung. Hierbei treten unweigerlich Abweichungen vom Konzept auf. So wurden zum Beispiel einige Klassen, die in der Grundidee noch deutsche Namen hatten, bei der Realisierung ins Englische umbenannt, um Konsistenz mit der Namensgebung und der Dokumentation des GermanTeams zu erreichen. Das Fazit schließlich betrachtet sowohl die bei der Entwicklung aufgetretenen Probleme als auch konzeptionelle Fehler, die bei der Verhaltensentwicklung gemacht wurden.

Im Folgenden wird unser Verhaltenskonzept und die Unterschiede zur bisherigen Lösung beschrieben. Um die Darstellung verständlicher zu machen, wird zunächst die bisherige Verhaltenssteuerung beschrieben.

7.1 Stand zu Beginn der Projektgruppe

Damit jede am GermanTeam beteiligte Universität ein eigenes Verhalten entwickeln kann, besteht die Möglichkeit verschiedene Lösungen für bestehende Probleme parallel als Module in das Programm zu integrieren (siehe Kapitel 2.4.2 auf Seite 13). Im letzten Jahr hat sich hierbei eine Beschreibungssprache Namens XABSL etabliert, auf deren Basis das Verhalten der Roboter entwickelt wurde.

Bei Verwendung von XABSL wird das Roboterverhalten durch Automaten modelliert (siehe [3]). Diese Vorgehensweise entstammt aus der Softwaretechnologie, wo Automaten zum Beschreiben komplexer Systeme eingesetzt werden. XABSL wurde im Rahmen der GermanOpen 2002 von Martin Lötzsch (HU Berlin) entworfen.

7.1.1 Die XABSL/XABSL2.0 Verhaltenssteuerung

XABSL ist eine auf XML beruhende Beschreibungssprache für hierarchische Automaten. Mit Hilfe von XSLT Transformations-Schemata[15] wird aus den XABSL-Quellendateien sowohl eine DOTML-Datei (s.u.) als auch eine Intermediate-Code-Datei (s.u.) erzeugt. XABSL wurde während der Arbeitszeit der Projektgruppe zu XABSL2 weiterentwickelt, welches mit seinem Vorgänger bis auf ein paar neue Features und einige andere Bezeichnungen weitgehend identisch ist. DOTML ist eine ebenfalls von Martin Lötzsch entwickelte Sprache, die wiederum durch ein XSLT-Schema in eine Textdatei transformiert werden kann, die als Eingabe für das Programm „dot“ [16] verwendet wird. Dieses Programm erzeugt mit Hilfe dieser Dateien Bilder, welche die Automaten graphisch darstellen. Damit lässt sich dann automatisch eine Dokumentation erstellen, die sich in einem Web-Browser darstellen lässt. Beim Intermediate Code handelt es sich um eine Textdatei, in der zeilenweise jeweils eine Zahl oder Klartext enthalten ist. Sie beschreiben den Automaten und ermöglichen der auf den Robotern laufenden XABSL-Engine, die Automatenzustände zu erzeugen und der Beschreibung gemäß miteinander zu verbinden.

Neben den bisher erwähnten Sprachen wird darüberhinaus noch die XInclude Sprache verwendet, um die Dateien, mit denen die Automaten beschrieben werden, klein und übersichtlich zu halten. Zur Veranschaulichung zeigt Abbildung 7.1 den gesamten Transformationsprozess als Pipes&Filters-Diagramm.

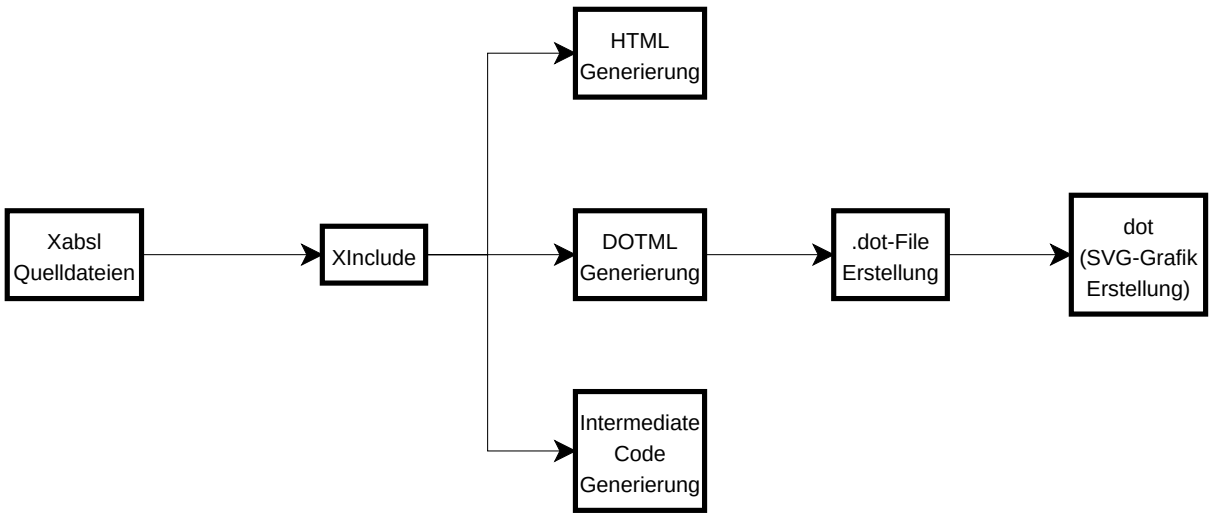

Abbildung 7.1: Hier wird ein Überblick über den XABSL-Transformationsprozess vermittelt. Die verschiedenen Quelldateien werden zunächst mittels XInclude zu einer Einzigen zusammengesetzt, aus der dann HTML-Dateien, Intermediate Code und über Zwischenschritte mit DOTML/dot Grafiken für die HTML-Dateien im SVG-Format erstellt werden.

Da die Modellierung von komplexen Automaten sehr unübersichtlich werden kann, bietet XABSL die Möglichkeit, mehrere Automaten hierarchisch miteinander zu Verknüpfen. Eine solche Hierarchieebene wird in XABSL2 „Option“ genannt. Dabei wird abhängig vom Zustand, in dem sich der Automat einer Option befindet, eine Folgeoption oder ein Folgebasicbehavior (also eines der möglichen Grundverhalten der Roboter) gewählt.

Letztere bilden die unterste Ebene der Optionshierarchie und bestimmen das Verhalten des Roboters.

Im folgenden Abschnitt soll nun erläutert werden, wie XABSL in der Praxis aussieht, wie es benutzt wird, und wie die Anbindung von XABSL an den C++-Code funktioniert.

7.1.2 XABSL in der Praxis

Im groben ist XABSL in vier Bereiche unterteilt: Input-Symbols, Output-Symbols, Basic-Behaviors und Options. Alle Bereiche bis auf die Options dienen als Schnittstelle zwischen C++ Code und XABSL und müssen deshalb sowohl in C++ als auch in XABSL eingebunden werden. Siehe dazu folgendes Beispiel:

Anhand des Input-Symbols potentialfield.priority wird hier das Anlegen eines Symbols durchgespielt.

- Zuerst wird ein Symbol in einer XABSL-Symboldatei angelegt.

```
<decimal-input-symbol
name="searchforball.priority"
description="Describes the priority"
measure="percent"/>
```

- Anschließend muss eine Methode in C++ Code geschrieben werden.

```
double getPotentialfieldPriority();
```

- Und zu aller letzt wird in der Methode registerSymbols der gleichen Klasse wie oben das XABSL-Symbol mit der C++ Methode verbunden und in XABSL registriert.

```
engine.registerDecimalInputSymbol("potentialfield.priority",this,
(double (Xabsl2FunctionProvider::*)())
&DroneSymbols::getPotentialfieldPriority);
```

Es ist nicht zwingend notwendig, dass Symbols, Options und BasicBehaviors alle in einer Datei untergebracht sein müssen. Es ist sogar für die Übersicht von Vorteil mehrere Dateien zu benutzen.

Input-Symbols stellen die Eingabe-Schnittstelle für XABSL dar. Zum Beispiel lassen sich dadurch die Positionen auf dem Spielfeld oder auch Systemeinstellungen des Betriebssystems abfragen. XABSL selbst kann Werte von Input-Symbols nur auslesen und nicht verändern. Für die Ausgabe von Werten sind im XABSL Output-Symbols vorhanden.

Die Output-Symbols dienen als Ausgabe von Anweisungen aus XABSL heraus. Mit diesen kann man alle Funktionen des AIBO ansteuern. Im wesentlichen dienen die Output-Symbols zur Ansteuerung der Kopf- und Schwanzbewegungen. Die Bewegung des AIBO wird durch BasicBehaviors angesteuert.

BasicBehaviors sind C++ Methoden, die aus XABSL aufgerufen werden und Parameter übergeben bekommen. Da die Berechnung der Anforderungen an die Motion-Control zu

komplex für XABSL sind, dienen die BasicBehavior als Zwischensicht zwischen XABSL und der Motion-Control. Sie berechnen aus den übergebenen Parametern Instruktionen für die Motion-Control. Als Beispiel sei die Basic Behavior „go-to-point“ angeführt. Sie bekommt als Parameter eine Feldposition übergeben und berechnet abhängig von der eigenen Position die Parameter für die Walking-Engine.

Options sind Automatendefinitionen, für die normalerweise eine eigene .xml-Datei angelegt wird. Ein Verhalten besteht im Allgemeinen aus mehreren solcher Options. Da eine Option eine Automat ist, besteht sie aus mehreren Zuständen, die in XABSL „States“ genannt werden. Wenn eine Option aktiviert wird (s.u.), so wird als erster Zustand der Startzustand verwendet, der explizit angegeben werden muss.

Jeder dieser Zustände besteht aus den folgenden Teilen:

- dem Aufruf einer BasicBehavior oder Option. Solange der XABSL Automat in diesem State verbleibt wird diese BasicBehavior oder Option ausgeführt.
- dem Setzen verschiedener Output-Symbols.
- dem Decision-Tree. Durch die Benutzung von booleschen Ausdrücken in Verbindung mit Input-Symbols und Konstanten lassen sich Entscheidungen treffen, die zu Zustandswechseln führen, und so den AIBO steuern.

7.1.3 Entwicklung der Konzepte

Da es sich bei der hier vorgestellten Verhaltenssteurung größtenteils um eine Neuentwicklung handelt, stellte der Entwicklungsprozess, beginnend mit einem gemeinsamen Brainstorming bis hin zu einer Verfeinerung der Konzepte einen wesentlichen Teil des Arbeitsprozesses dar. Diese Arbeit wurde als Gruppendiskussion der Behavior-Teilgruppe durchgeführt. Dieser Abschnitt soll den Entwicklungsprozess und die angedachten Teillösungen erläutern, die erdachten Konzepte erklären und die Entscheidungen transparent machen.

Zu Beginn der Projektgruppenarbeit wurde die bisherige Spielweise des German Teams, welche auch in bei der Weltmeisterschaft 2002 in Fukuoka benutzt wurde, analysiert. Beim Betrachten ergaben sich schnell folgende Beobachtungen:

- Die Roboter bewegen sich im Vergleich zu denen von anderen, guten Teams wie z.B. CMU, langsam.
- Objekte werden oft falsch erkannt.
- Die Roboter nehmen bei ihren Bewegungen keine Rücksicht auf Hindernisse, wie andere Roboter, und verhaken sich dadurch oft.
- Das Spielgeschehen wirkt sehr statisch, jeder Roboter behält seine einmal zugewiesene Rolle stets bei, ohne dabei die aktuelle Spielsituation zu berücksichtigen.
- Die Roboter handeln nicht koordiniert.

Da die ersten beiden Punkte sich durch Entwicklungen am Verhalten kaum verändern lassen, und Punkt 1 bereits von Mitarbeitern des Lehrstuhls Datenverarbeitungssysteme der Fakultät für Elektrotechnik der Universität Dortmund und eines Mitarbeiters des Lehrstuhls für Systemanalyse des Fachbereichs Informatik der Universität Dortmund bearbeitet sowie Punkt 2 von den anderen beiden Teilgruppen der Projektgruppe in Angriff genommen wurden, konzentrierte sich die Diskussion über das Verhalten auf die letzten drei Punkte.

Um das statische Spielgeschehen dynamisch zu gestalten, die Spielintelligenz der Roboter zu erhöhen und die Koordination zu verbessern, wurde beschlossen, eine an den echten Fußball angelehnte Trainer-Spieler-Architektur zu entwerfen. Hierbei sollte den Spielern ein Trainer zur Seite gestellt werden, der ihnen Instruktionen gibt, wie sie sich zu verhalten haben. Da es den Regeln nach nicht erlaubt ist, den Robotern Informationen von außen zukommen zu lassen, wurde beschlossen, den Trainer als verteiltes Programm auf den Spielern laufen zu lassen.

Es gibt verschiedene Möglichkeiten zur Realisierung des Trainers. Eine erste Vorstellung war, den Trainer durch eine XABSL-Maschine zu simulieren, also auf jedem Roboter zwei Automaten parallel Laufen zu lassen. Da sich Automaten allerdings nur schwer zur Modellierung des Trainers einsetzen ließen, wurde dieser Vorschlag schnell zu Gunsten einer Implementierung des Trainers in C++ verworfen. Eine weitere Grundüberlegung befasst sich mit der Art des Trainers, wobei die Möglichkeit bestand, einen starken oder einen schwachen Trainer zu verwenden. Ein schwacher Trainer ist hierbei wie ein Trainer beim echten Fußball zu verstehen, bei dem die Spieler weitgehend autonom sind und nur einen sehr groben Einfluss auf das Spielgeschehen hat. Beim Modell des starken Trainers hingegen haben die Spieler eine eher geringe Entscheidungskompetenz. Fast alle Entscheidungen werden vom Trainer getroffen. Da die Roboter im Gegensatz zu einem echten Trainer eine wesentlich effizientere Kommunikation haben, fiel die Entscheidung zugunsten eines starken Trainers. Das Konzept erschien vielversprechender, da sich dadurch ein größerer Handlungspielraum ergab.

Bei der weiteren Analyse wurde erörtert, wie ein auf den Robotern verteilt laufender Trainer realisiert werden kann. Ausgehend von der Annahme, dass durch die Sensorfusion auf allen Hunden ein einheitliches und präziseres Weltbild entstehen würde, wurde beschlossen, keine echte verteilte Berechnung zu realisieren, sondern die Trainerberechnung auf jedem Roboter einzeln ablaufen zu lassen, und dann die Ergebnisse abzugleichen.

Damit ergab sich ein einfaches Modell: Auf jedem Hund läuft ein Trainer. Da die Trainer alle über die gleichen Daten verfügen, kommt jeder zum gleichen Ergebnis: eine Spielstrategie, die dann mit den anderen Trainern abgeglichen wird. Der Spieler weiß davon nichts, bekommt aber von seinem Trainer die Strategie und führt sie aus, sofern er kein wirklich dringliches Eigeninteresse hat. Die Dringlichkeit eines Spielerinteresses kann auch als „Aufmerksamkeit“ des Spielers dem Trainer gegenüber interpretiert werden.

Das nächste erörterte Konzept waren die so genannten „Wohlfühlzonen“. Hierbei sollte die starre Rollenverteilung der Spieler aufgehoben werden, und durch Bereiche ersetzt werden, in denen sich der Spieler am liebsten aufhält. So würde sich ein Verteidiger zwar mit vorne aufhalten können, aber doch lieber im hinteren Bereich des Spielfelds verweilen, wenn ihn nicht ein besonderer Grund dort weglockt. In diesem Konzept wurden die Spieler

und der Ball als Krümmungen der Wohlfühlzonen verstanden. So bildet der Ball eine Mulde, wohingegen ein anderer Roboter einen Berg darstellt. Wenn nun noch der Raum, in dem sich der Spieler normalerweise aufhält, als Vertiefung dargestellt wird, und man dem Spieler den Auftrag gibt, sich möglichst zu dem Punkt zu bewegen, an dem die Vertiefung am größten ist, ergibt sich ein teilweise dynamisches Potenzialfeld mit einer Art „Raumkrümmung“.

Der Vorschlag der Wohlfühlzonen/des Potenzialfeldes wurde zunächst sehr kontrovers diskutiert. Insbesondere die Frage, ob sich das Konzept mit der Trainer-Spieler-Architektur vereinbaren ließ, war umstritten. Dieses Problem wurde gelöst durch den Entwurf eines volldynamischen Potenzialfeldes. Im volldynamischen Potenzialfeld gibt es keine Wohlfühlzonen mehr, die für den Spieler je nach Rolle immer gleich bleiben. Vielmehr übernimmt nun der Trainer die Erstellung des Feldes, indem er die Spielstrategie in Raumkrümmungen realisiert. Das Potenzialfeld ist nun eine Aufbereitung der Trainerdaten für den Spieler. In das Potenzialfeld fließen im Unterschied zum reinen Trainer-Spieler-Konzept auch die Positionen der Spieler (Mitspieler/Gegner) ein. Damit können Zusammenstöße mit anderen Robotern langfristig vermieden werden. Außerdem ließ sich durch das Potenzialfeld eine Aufmerksamkeitssteuerung realisieren. Der Spieler durfte nun genau dann seine eigenen Interessen verfolgen, wenn das Potenzialfeld nur schwach um ihn herum gekrümmt war. Eine effektive Nutzung des Potenzialfeldes setzt allerdings voraus, dass Gegner zuverlässig erkannt werden. Es ist allerdings möglich, Gegner auch dann zu umlaufen, wenn sich der Spieler nicht ganz sicher ist, ob dort wirklich ein Roboter steht, indem man bei geringen Gegnervaliditäten größere Bereiche des Potenzialfeldes verändert.

Nun wurde auch spezifiziert, woher der Trainer seine Strategien nehmen sollte. Es wurde beschlossen, die Spielzüge in einer Datenbank auf den Robotern abzulegen. Eine weitere Idee war, diese Datenbank lernfähig zu halten, sodass nach einem Torerfolg gewisse Spielzüge gleich noch einmal gemacht werden, um Schwächen in der Programmierung der Gegner gezielt auszunutzen.

In der weiteren Diskussion ergab sich, dass es noch globale (also Trainer-)Interessen gibt, die durch das Konzept bisher nicht abgedeckt waren. So kann es passieren, dass auf allen Robotern die Ballvalidität oder die Validität von Gegnern stark absinkt. In diesem Fall muss der Trainer reagieren, indem er den Spielern befiehlt, gezielt nach gewissen Objekten auf dem Spielfeld zu suchen. Da so etwas nicht mit statischen Spielzügen realisiert werden kann, wurde eine weitere Datenaufbereitungsinstanz in den Trainer eingearbeitet: Der AdvisoryGenerator.

Der Trainer besteht nun aus zwei Ebenen: Dem Teil, der die Strategieberechnungen vornimmt und mit den anderen Trainern abgleicht, und dem Teil, der die Datenaufbereitung vornimmt, also das Potenzialfeld berechnet und sonstige Empfehlungen (Advises) ausspricht.

Jetzt stellte sich die Frage, was denn eigentlich eine Strategie sei. Hierfür standen mehrere Ideen zur Verfügung. Die erste Idee war, dass eine Strategie jedem Spieler sagt, er solle zu einem bestimmten Punkt auf dem Spielfeld gehen. Eine weitere Idee bestand darin, die Spielern je nach Spielsituation in bestimmte Zonen zu dirigieren. Weiterhin bestand die Möglichkeit, Strategien als Spielaufstellungen zu begreifen (1-1-2-System, 1-2-1-System etc). Aufgrund der größeren Flexibilität wurde schließlich beschlossen, dass eine Strategie

sich aus Zonen zusammensetzt, für die Zielkoordinaten festgelegt werden. Die Roboter bewegen sich zu den Koordinaten, die durch die Zone festgelegt sind, in der sie sich befinden. Um eine bequeme Strategieeingabe zu ermöglichen, erschien ein graphischer Strategieeditor als ein sinnvolles Werkzeug.

7.1.4 Zusammenfassung der Ergebnisse

Nachdem im vorigen Abschnitt die Diskussion und Beschlüsse dargestellt wurden, soll hier nun eine Zusammenfassung des Konzepts beschrieben werden.

Das bisher rein lokale Verhalten jedes Roboters soll um eine globale Stufe erweitert werden. Diese globale Stufe fungiert als Trainer, der das Spielgeschehen im ganzen berücksichtigt und allen Spielern Handlungsempfehlungen gibt.

Da die Mannschaft ausschließlich aus maximal vier autonomen Robotern mit gegenseitiger WLAN-Kommunikation besteht, kann es kein wirkliches globales Objekt geben. Der Trainer muss auf den maximal vier Robotern verteilt ablaufen.

Da die Mannschaft ausschließlich aus maximal vier autonomen Robotern mit gegenseitiger WLAN-Kommunikation besteht, ist ein globales Objekt nur durch eine entsprechende Architektur möglich, etwa dass der Trainer auf den maximal vier Robotern verteilt abläuft.

Dies wurde so umgesetzt, dass auf jedem Roboter eine vollständige Trainerinstanz läuft. Diese Redundanz ist die beste Absicherung gegen Wegfall eines oder mehrerer Roboter (z.B. durch Ausfall des WLAN) aus dem Netzwerk. Eigentlich müssten alle Trainerinstanzen synchron handeln, da sie idealerweise über die gleichen Eingangsdaten verfügen. Da dies aber in der Realität, z.B. durch begrenzte Systemgenauigkeit in der Bildverarbeitung, nicht der Fall ist, müssen sich die Trainerinstanzen in regelmäßigen Zeitabständen synchronisieren. Aufgrund der begrenzten WLAN-Resourcen ist auch die Häufigkeit der Synchronisation begrenzt. Damit nun die Trainerinstanzen während eines ausreichend großen Anteils der Spielzeit genügend synchron sind, werden die Traineranweisungen entsprechend grob gestaltet. Deshalb sollten bei ähnlichen Spielsituationen die Traineranweisungen ebenso ähnlich sein.

Die Traineranweisungen werden einer Strategiedatenbank entnommen, die identisch auf jedem Roboter existiert und vor dem Spiel mit einem eigens dafür entworfenen Tool erstellt wurde. Jede Spielsituation kann durch den Trainer aus einer Menge, die natürlich auch leer sein kann, von Traineranweisungen abgebildet werden.

Hier wird dann die nach einer bestimmten Bewertung günstigste Anweisung gewählt. Eine Traineranweisung bietet für eine Teilmenge aller Spieler folgende mögliche Empfehlungen:

- sich zu einer bestimmten Position zu bewegen
- sich zum Ball zu bewegen und ihn zu einer bestimmten Position zu schießen
- sich zum Ball zu bewegen und ihn zu einer bestimmten Position zu führen

Diese Handlungsempfehlungen liest der Spieler nicht direkt. Da sich Potenzialfelder als

sinnvoll für die Steuerung von Robotern erwiesen haben (siehe [3]) werden Bewegungsempfehlungen für jeden Spieler in einem Potenzialfeld eingearbeitet. Dieses bildet jede Spielfeldposition auf ein Potential ab. Ein hohes Potential ist vermeidenswert, ein niedriges erstrebenswert. Große Potentialunterschiede verbessern bzw. verschlechtern das Potential auf kurzem Weg stark. Solch steile Bereiche drücken eine hohe Dringlichkeit aus sich zu verbessern. Empfehlungen für Schüsse, Pässe und andere Aktionen werden zusammengefasst. Diese Trainerempfehlungen kann man als globale Interessen bezeichnen. Jeder Spieler kann aber noch lokale Interessen haben. Diese sollen gelten, wenn keine dringenden globalen Interessen vorliegen oder lokale Entscheidungen sehr dringend sind. Lokale Interessen können z.B. das Verhalten in Risikosituationen wie einem Torabschluss oder das Verhalten zur Sichtverbesserung sein. Der Spieler kann nun die globalen gegen die lokalen Interessen abwägen und seine Handlungen durchführen.

Schlussendlich wurde für das Verhaltenskonzept der treffende Name CollectiveBehavior gewählt.

7.2 Die Module der CollectiveBehavior

Die Arbeit der CollectiveBehavior arbeitet in mehreren Phasen und wurde dementsprechend in Module aufgeteilt. Jedes Modul setzt dabei eine Phase um.

Phase 1 ist der **Trainerkern**. Er entnimmt entsprechend des Weltbildes eine Strategie der Strategie-Datenbank und schreibt diese aufs Blackboard. Außerdem führt er einen Datenabgleich mit den anderen Trainern durch.

Phase 2 ist der Generator des **Potenzialfeldes**. Er erstellt aufgrund des Weltbildes und der Strategie für jeden Spieler ein Potenzialfeld und schreibt dieses aufs Blackboard.

Phase 3 ist der Generator der **Handlungsempfehlung**. Er entscheidet aufgrund des Weltbildes und der Strategie, welche globalen Interessen vorliegen und schreibt diese aufs Blackboard.

Phase 4 ist der **Spieler** selbst. Er berechnet aus der aktuellen Spielposition und seinem Potenzialfeld die globale Bewegungsempfehlung. Diese hat eine gewisse Dringlichkeit. Aufgrund dieser und der Dringlichkeit des Advises (zur Schreibweise: siehe Kapitel 7.3.5.1 auf Seite 93) wird entschieden, ob die globalen Interessen die lokalen überwiegen. Jetzt wird das dominante Verhalten ausgeführt.

Um einen Informationsaustausch zwischen diesen Phasen zu gewährleisten gibt es eine Blackboard-Architektur. Eine Blackboard-Architektur beschreibt ein Datenhaltungssystem, auf das alle es benutzenden Instanzen sowohl lesend als auch schreibend zugreifen können. Der Vorteil einer solchen Architektur ist ihre Einfachheit.

Jede Phase liest die für sie bereitgestellte Informationen sowie Informationen von einem Blackboard, verarbeitet diese und schreibt ihr Ergebnis ausschließlich wieder in das Blackboard.

Nach dem ursprünglichen Konzept der CollectiveBehavior läuft auf jedem Roboter ein vollwertiger Trainer, der die Information für jeden Spieler aufbereitet. Aus Gründen der Leistungssteigerung werden Bearbeitungen wie z.B. die Generation der Potenzialfelder nur für den Spieler erstellt, der mit der Trainerinstanz real auf dem Roboter läuft. So werden nur die Informationen bearbeitet, die später auch benutzt werden.

Im weiteren wird jeweils das Konzept der einzelnen Module beschreiben.

7.2.1 Trainer-Kern

Der Trainer-Kern integriert das Modul Visionary (siehe 7.2.1.2) und die Strategie-Datenbank (siehe 7.2.1.1). Darüberhinaus ist er auch dafür verantwortlich, dass die Strategie-Berechnungen der Roboter miteinander abgeglichen werden. Damit übernimmt der Trainerkern in diesem Konzept die Rolle der Entscheidungsfindung. Die darauf folgenden Stufen dienen zur Aufbereitung der Daten und zum Fällen der Spieler-Entscheidung.

7.2.1.1 Strategie-Datenbank

Die Strategiedatenbank ist dafür gedacht, eine „Ideallösung“ für ein bestehendes Problem zu liefern, ähnlich einer Eröffnungsdatenbank beim Schach. Der Strategy-Chooser bekommt als Eingangsdaten die Position sämtlicher bekannter Roboter sowie die des Balles übergeben, befragt mit diesen Daten die Strategiedatenbank und liefert dem Trainer eine Bewegungs- und Aktionsempfehlung für alle Roboter der eigenen Mannschaft. Auf diese Art lässt sich ein teambasiertes Verhalten relativ einfach planen und realisieren.

Desweiteren führt der StrategyChooser noch eine interne Statistik, welche Strategien besonders oft zu einem Erfolg, also dem Schießen eines Tores in der Offensive und dem Verhindern eines Tores in der Defensive, geführt haben und kann auf Grund dieser Informationen sich an das Verhalten und die Strategie des Gegners anpassen.

7.2.1.2 Visionary

Mit dem Modul Visionary wurde in diesem Konzept das Ziel verfolgt ein Orakel zu schaffen. Dieses "Orakel" sollte den StrategyChooser in seiner Arbeit, die richte Strategie auszuwählen, unterstützen.

Aus den übergebenen Positionen und Bewegungsrichtungen der Spieler errechnet die Visionary die Position dieser Spieler für einen Zeitpunkt in der Zukunft. Dieser Zeitpunkt kann beliebig gewählt werden. Es sollte aber darauf geachtet werden, dass je weiter dieser Zeitpunkt in der Zukunft liegt die Wahrscheinlichkeit einer richtigen Vorhersage durch die Visionary abnimmt. Für die Berechnung der zukünftigen Position des Balls wird dieses Modul ebenfalls angewandt.

7.2.2 Potenzialfeld

Das Potenzialfeld dient zur Einbindung des durch Sensorfusion entstehendes Weltbildes in die CollectiveBehavior. Die Daten des StrategyChoosers werden aufbereitet und mit den Daten des Weltbildes zu einem Potenzialfeld verschmolzen. Das Potenzialfeld ist eine Abbildung der Menge der Spielfeldpositionen auf ein Potential (siehe Abbildung 7.2). Hier sieht man, wie sämtliche Feldpositionen auf ein Potential und damit auf eine Höhe abgebildet werden. Einflussfaktoren sind die zu vermeidenden Spieler 2-4 und die Anweisung für Spieler 1, sich nach X zu bewegen.

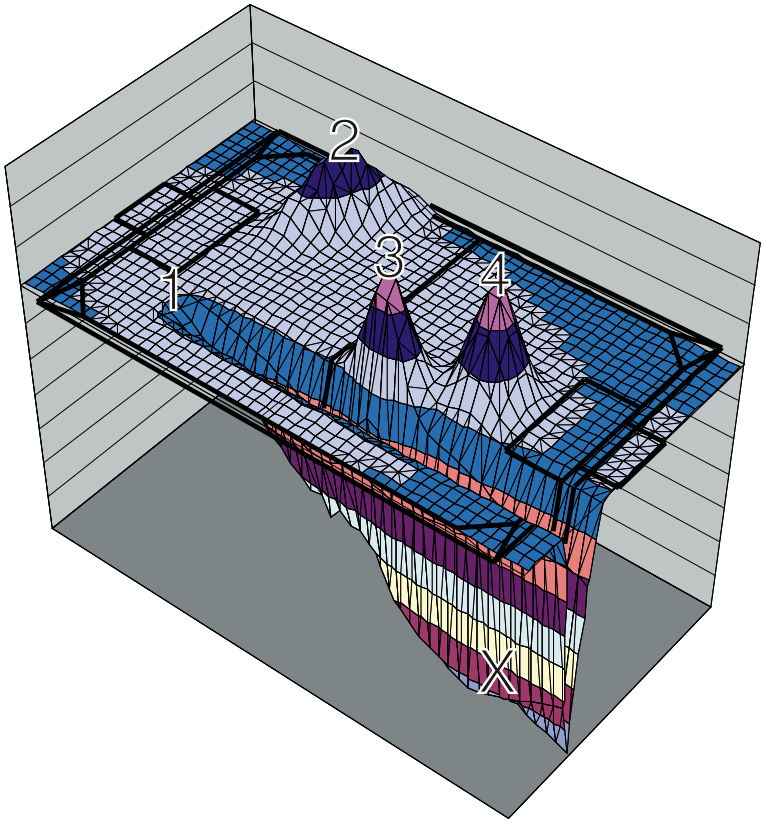

Abbildung 7.2: Das Potenzialfeld als Abbildung: Jede Feldposition wird auf ein Potential abgebildet. Es wird eine klassische Spielsituation dargestellt. Position 1 ist der aktuelle Spieler. Dieser soll sich nach Position X bewegen, wobei die Positionen 2-4 Hindernisse, das heißt andere Spieler, repräsentieren.

Hier soll ein hohes Potential eine schlechte Spielerposition bedeuten, ein tiefes Potential eine gute Spielerposition. Jeder Spieler besitzt ein individuelles Potenzialfeld. Während des Spiels soll sich nun der Spieler von seiner aktuellen Spielerposition auf dem Potenzialfeld „abwärts“ bewegen; er begibt sich in die Richtung, die in einer gewissen Umgebung den stärksten Potentialabfall bietet. Jeder Mitspieler, jeder Gegenspieler, jede Strategische Weisung übt unabhängig voneinander Einfluss auf das Potenzialfeld aus. Diese Einflussfaktoren werden im weiteren Aspekte genannt.

Die Spielerpositionen sind als Gauss-Verteilungen gegeben (siehe Kapitel 5.3.2 auf Seite 51) und werden proportional in Höheninformationen überführt, sie bilden Berge als Gauss-Glocken (siehe Abbildung 7.3 auf der nächsten Seite).

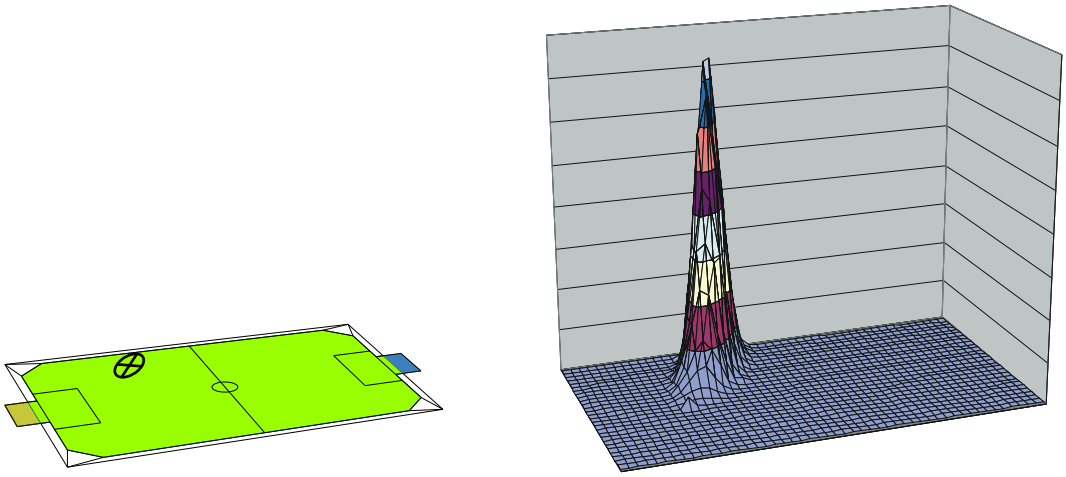

Abbildung 7.3: Eine Gauss-Verteilung (dargestellt durch das μ (Position) und σ (Größe) ausdrückende Parameterkreuz (siehe Kapitel 5.3.2 auf Seite 51)) auf dem linken Bild soll in Potential überführt werden. Auf dem rechten Bild sieht man die typische Ausprägung als Gauss-Glocken-Form.

Die strategischen Weisungen werden in abfallende Strukturen überführt, z.B. die Weisung vom aktuellen Punkt einen Pfad abzulaufen wird zu einem abfallenden Graben über den Pfad zum Zielpunkt. Der Abfall ist linear, der Querschnitt des Grabens entspricht einer umgedrehten Gauss-Kurve (siehe Abbildung 7.4).

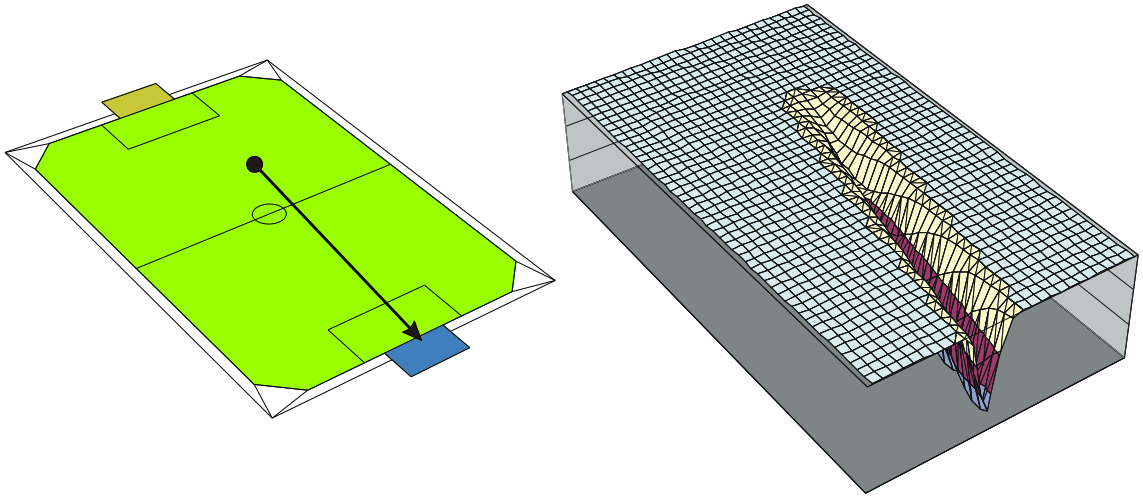

Abbildung 7.4: Die strategische Weisung auf dem linken Bild, sich vom Punkt zur Pfeilspitze zu bewegen, soll in Potential überführt werden. Auf dem rechten Bild sieht man das Ergebnis, einen zum Ziel abfallenden Graben, der im Querschnitt eine Ausprägung als Gausskurve besitzt.

Die Summe der Höhen jedes Aspektes auf einem Punkt ist die Höhe dort auf dem Potenzialfeld. Dazu wird für jede Position ein Potential gespeichert. Hierfür ist das Feld hexagonal aufgeteilt (siehe Abbildung 7.5 auf der nächsten Seite).

Für jedes dieser Sechsecke wird ein Wert gespeichert, d.h. für alle Positionen innerhalb eines Sechsecks gilt vereinfachend ein Potential.

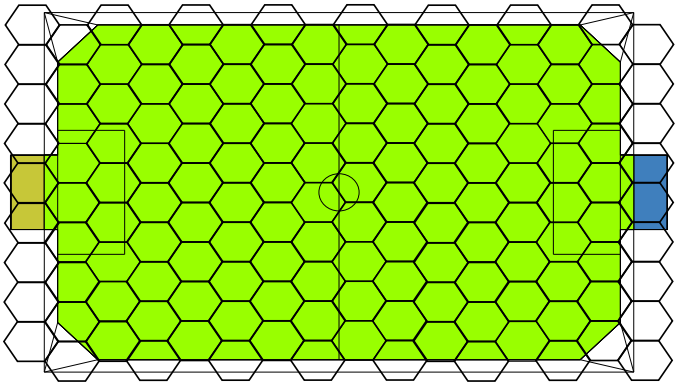

Abbildung 7.5: Aufteilung des Feldes in Sechsecke

Die Hexfelder enthalten eine $\mathbb{Z} \times \mathbb{N}$ Nummerierung (siehe Abbildung 7.6).

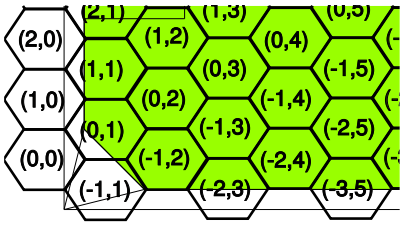

Abbildung 7.6: Nummerierung der Hexfelder

Der Potenzialfeldgenerator erzeugt jeweils aus den Spielerpositionen und den strategischen Weisungen Potenzialfeldaspekte, die dann dem Potenzialfeld zum Verarbeiten bzw. Aktualisieren übergeben werden. Da zu jeder Zeit eine Potenzialfeld-Annäherung als Quantelung auf Hexfelder gegeben ist, werden vom Potenzialfeldgenerator nur die Aspekte übergeben, die eine hinreichende Veränderung mit sich bringen, z.B. die Spielerposition ist über einen gewissen Wert hinaus gewandert, eine andere Strategieanweisung ist gegeben oder bei der Weisung zum Ball zu laufen hat sich der Ball hinreichend bewegt.

Es werden stellvertretend für die Potentiale aller Spielfeldpositionen die Potentiale aller Hexfelder gespeichert. Zudem werden alle Aspekte in einer Liste gespeichert. Einzelne Aspekte können hinzugefügt und gelöscht werden; außerdem kann das ganze Potenzialfeld zurückgesetzt werden. Werden einzelne Aspekte hinzugefügt bzw. gelöscht, so geschieht die Manipulation der Menge der Hexfelder über einen Manipulator. Um eine Handlungsempfehlung abzugeben, wird um die hiermit gegebene Position (i.A. die aktuelle Spielerposition) das Potenzialfeld nach einer geeigneten Richtung durchsucht: Im Doppelring an Hexfeldern um das der Position zugeordneten Hexfeldes wird nach dem Feld geprüft, welches ein größtes Gefälle zur aktuellen Position bietet. Die Richtung zeigt von der gegebenen Position auf die Mitte des gefundenen Hexfeldes. Ist kein Feld besser, ist die Richtung der Nullvektor (siehe Abbildung 7.7 auf der nächsten Seite).

Der Manipulator soll einen Potenzialfeldaspekt in die Menge der Hexfelder einarbeiten bzw. wieder ausarbeiten. Dafür muss er mit dem Aspekt initialisiert werden und benötigt die Abgrenzung dessen. Dies ist eine Liste von Punkten, die die Eckpunkte eines konvexen Vielecks bilden.

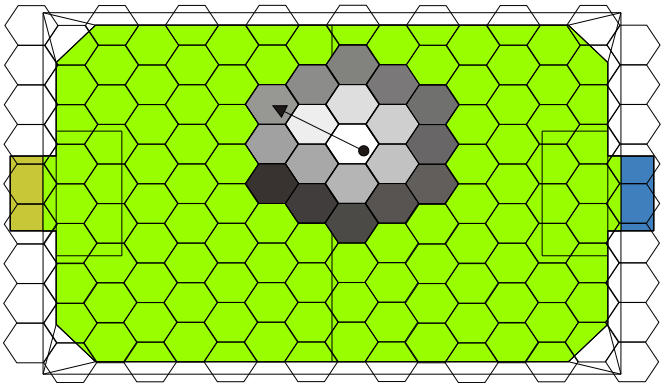

Abbildung 7.7: Suche nach geeigneter Richtung (Reihenfolge: von hell nach dunkel) - Der Pfeil drückt die gefundene Richtung aus.

Alle Positionen auf der HexArea, die von dem Aspekt hinreichend beeinflusst werden (d.h. auf ganze Zahlen gerundet ist ihr Einfluss ungleich 0) sollen innerhalb des Vielecks sein. Bei Aspekten, die Spieler und den Ball darstellen, also durch Gauss-Glocken gegeben sind, sind dies hier Rechtecke (siehe Abbildung 7.8).

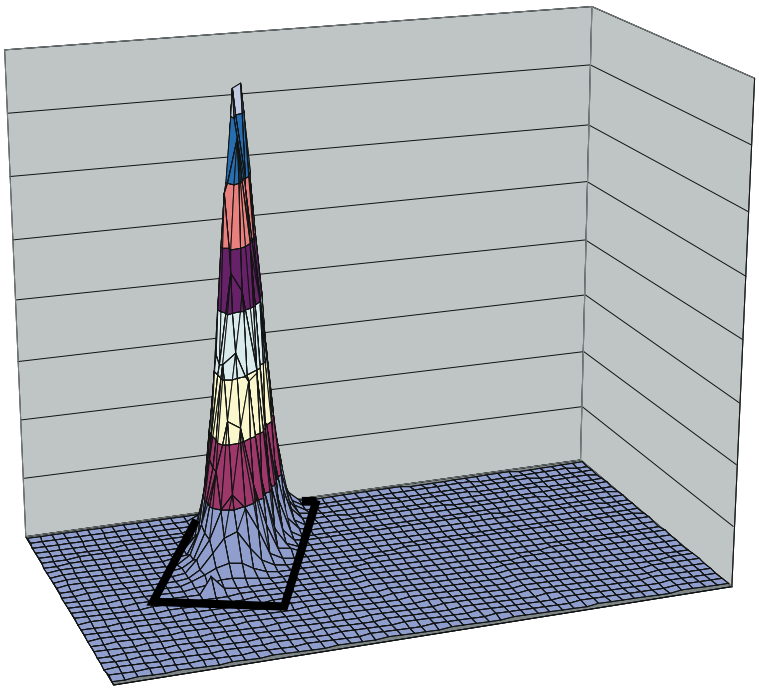

Abbildung 7.8: Begrenzung der Gauss-Glocke durch ein Vieleck

Bei Aspekten, die Bewegungsanweisungen darstellen, also durch Gräben gegeben sind, sind dies hier ebenso Rechtecke.(siehe Abbildung 7.9 auf der nächsten Seite)

Ist der HexAreaManipulator initialisiert, soll er alle Punkte berechnen und in eine Liste schreiben, dessen zugehörige Hexfelder durch das konvexe Begrenzungsvieleck mindestens teilweise überdeckt werden. Für den Algorithmus sind die Punkte des Vielecks gegeben im Uhrzeigersinn. Der Algorithmus geht folgendermaßen vor: Die HexArea wird zeilenweise durchlaufen. Es gibt immer eine linke und eine rechte Begrenzung, die jeweils auf

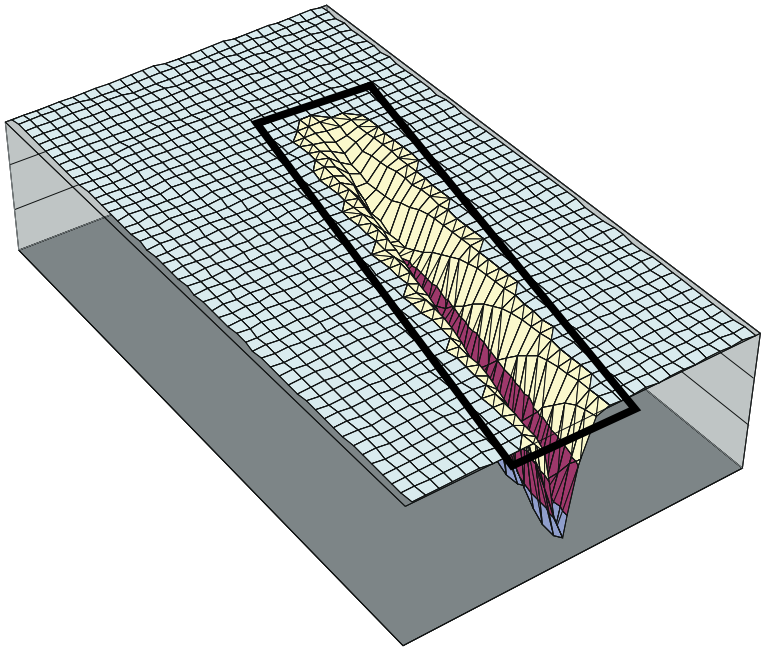

Abbildung 7.9: Begrenzung der Bewegungsanweisung durch ein Vieleck

der Vieleckkante liegen. Die Hexfelder, die zwischen der linken und rechten Begrenzung liegen, werden der Liste hinzugefügt. Dann wird die Begrenzung für die nächste Zeile anhand der Steigung der Vieleckkanten hochgerechnet. Da die Punkte im Uhrzeigersinn gegeben sind, werden neue Kanten in den zugehörigen Zeilen erkannt. Der Algorithmus startet mit dem Punkt, bzw. den Punkten, die in der obersten Zeile liegen. Dies lässt sich mit linearem Zeitaufwand ermitteln. Er endet, wenn alle Punkte abgearbeitet sind (siehe Abbildung 7.10).

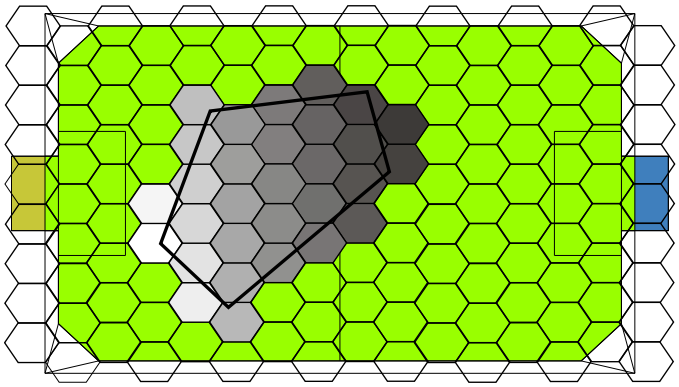

Abbildung 7.10: Die Hexfelder, die vom Vieleck berührt werden, werden in der hier dargestellten Reihenfolge von hell nach dunkel abgearbeitet.

Insgesamt wird jeder Punkt genau einmal besucht. Der Algorithmus hat bezüglich der Anzahl der Hexfelder linearen Zeitbedarf. Das durchlaufende Vieleck muss konvex sein, da sich sonst ein Nachfolgepunkt in einer schon abgearbeiteten Zeile befinden kann.

7.2.3 Handlungsempfehlung

Die Aufgabe des Generators der Handlungsempfehlung ist, die lokalen Interessen des Roboters, also zum Beispiel um die Lokalisation des Balls oder die Selbstlokalisation, einzuleiten. Diese Interessen müssen auch gegen die globalen Interessen abgewogen werden, so dass mit höherer Wahrscheinlichkeit nach dem Ball gesehen wird, wenn das globale Interesse, also z.B. die Dringlichkeit des Potenzialfeldes, gering ist. Außerdem werden die Ergebnisse der Strategiedatenbank vorverarbeitet, so dass sie für den XABSL-Automaten und dem Potenzialfeldgenerator bereitliegen.

7.2.4 Spieler

XABSL hat sich als ein leistungsstarkes und mächtiges Kozept zur Steuerung eines AIBO erwiesen. In Ansätzen gibt es auch schon Versuche alle AIBOs mit XABSL simultan zu steuern und zwar mit Hilfe der Kommunikation über das WLAN.

Allerdings war die synchrone Steuerung von mehreren Robotern war zum Zeitpunkt der Projektarbeit mit XABSL nicht vernünftig realisierbar. Es fehlten Input/Output-Symbols für die Synchronisation der einzelnen AIBOs und es erschien zu schwierig, dieses mit Hilfe von XABSL zu realisieren. Die größten Befürchtungen waren, dass der XABSL-Code zu stark aufgebläht würde und mit dieser Aufgabe überfordert sein könnte.

Es wurde beschlossen, XABSL weiterhin für die Steuerung der einzelnen Roboter - also zur Realisierung des Spielerteils - zu benutzen, da dieses Konzept sich bereits bewährt hatte, und mit dem Trainer-Spieler-Konzept die vernünftige Steuerung und Synchronisation aller AIBOs zu gewährleisten war.

7.3 Realisierung

Hier soll nun beschrieben werden, wie die Module der CollectiveBehavior umgesetzt und welche Klassen dafür entworfen wurden. Es wurde eine Klassenhierarchie konzipiert.

Die Klasse Blackboard kapselt alle Informationen des Blackboards. Diese sind die Handlungsempfehlungen (Advise, die strategischen Empfehlungen (StrategicalDatabaseMoveAdvise), die Potenzialfelder (Potentialfield) und die gefilterten Informationen des Weltbildes (PlayerPoseCollection und BallPosition). Um den Zugriff auf die Blackboard-Daten zu vereinfachen, wurde eine weitere Klasse Namens Blackboarduser eingeführt. Sämtliche davon abgeleiteten Klassen dürfen auf das Blackboard zugreifen und mit seinen Daten arbeiten. Außerdem unterstützt Blackboarduser das Phasenkonzept mit der Methode execute. So kann jede Instanz von Blackboarduser ausgeführt werden. Die konkreten Phasen werden nun durch Ableitungen von Blackboarduser realisiert. Der TrainerCore führt den Datenabgleich mit den anderen Robotern durch und startet den StrategyChooser. Der AdvisoryGenerator erstellt die Empfehlung und der PotentialfieldGenerator das Potenzialfeld.

Dies wird in Abbildung 7.11 dargestellt.

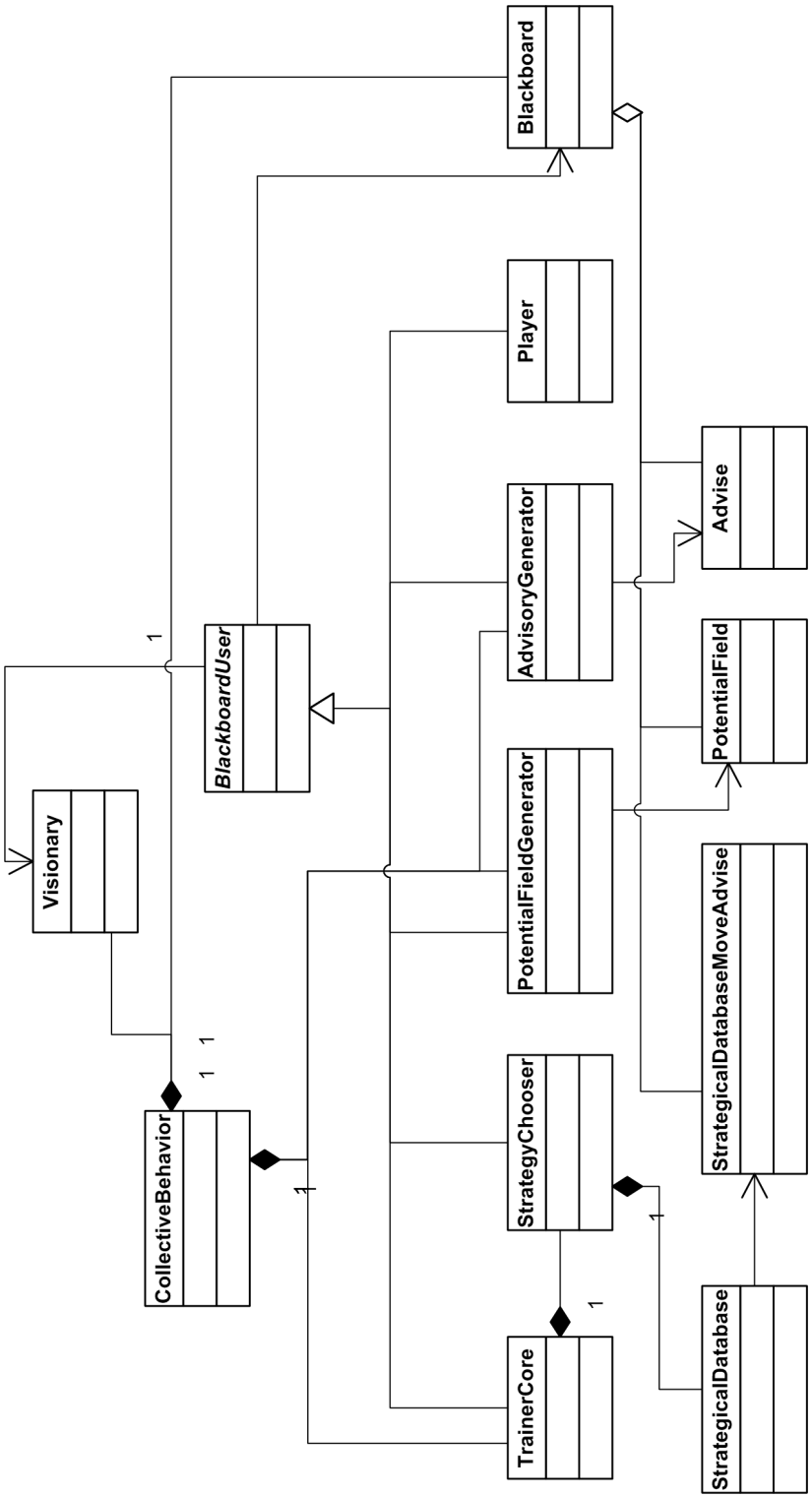

Abbildung 7.11: Klassendiagramm der CollectiveBehavior

7.3.1 Strategie-Chooser

Der Strategie-Chooser ist zuständig für eine Klassifizierung der Spielsituation, der Suche in der Strategiedatenbank nach dieser Spielsituation und der Ausgabe einer Handlungsempfelung.

7.3.1.1 Klassifizierung der Spielsituation

Das Matching in der Strategiedatenbank läuft über ein bis zu 18-dimensionales Matching. Jede in der Datenbank enthaltene Teilstrategie muss einen Ball (2 Dimensionen) enthalten und kann bis zu 4 eigene (4x2 Dimensionen) und vier gegnerische Roboter (4x2 Dimensionen) matchen. Hierzu werden so genannte Locations verwendet. Eine Location ist ein n-Eck mit drei bis acht Ecken auf dem zweidimensional betrachteten Spielfeld. Für eine Teilstrategie kann man als notwendige Bedingung nun festlegen, dass der Ball, ein Gegner oder ein Mitspieler sich in einem bestimmten Bereich befinden muss, damit eine bestimmte Teilstrategie durchgeführt werden kann. Jede Teilstrategie ist außer den Matchingbereichen auch noch mit einer Priorität ausgestattet, die festlegt, welche Teilstrategie genommen wird, wenn mehrere Teilstrategien für die konkrete Spielsituation passen. Des weiteren ist es bei mancher Teilstrategie möglich, diese invertiert an der Längsachse des Feldes auszuführen. Dies wird in einem zusätzlichen Flag signalisiert. Jedenfalls wird versucht, die aktuelle Spielsituation in einem Eintrag der Strategiedatenbank wiederzufinden, um entsprechende Instruktionen aus der Datenbank holen zu können. Wie dies geschieht, wird im folgenden Kapitel erklärt.

7.3.1.2 Konzept des Matchings

Um das oben erwähnte 18-Dimensionale Matching durchführen zu können, ist es nötig, den Strategie-Chooser in zwei Teilbereiche aufzuteilen. Zum einen der Preprocessing-Teil, der beim Bootvorgang des Hundes einmalig ausgeführt wird, und zum anderen das Echtzeitmatching, welches während des Cognition-Durchlaufs genutzt wird.

Beim Preprocessing wird zuerst das Spielfeld in 5x5cm große „Sektoren“ unterteilt. Dann wird für jede Location geschaut, welche Sektoren sie umfasst. Bei jedem dieser Sektoren wird nun in einem Bitvektor das entsprechende Bit gesetzt. Auf diese Art und Weise wird eine relativ einfache Look-Up Karte erstellt, bei der man mit einem Zugriff feststellen kann, welche Locations in einem bestimmten Sektor vorhanden sind. (siehe Abbildung 7.12 auf der nächsten Seite)

Diese Karte wird die Basis für das Echtzeitmatching darstellen. Desweiteren werden auch alle Teilstrategien durchlaufen und eine Zuordnungstabelle von Locations nach Teilstrategien, bei denen der Ball sich in der Location befindet, aufgebaut. Diese Tabelle ist in der Form $1 \rightarrow n$, bei der eine Location auf mehrere Teilstrategien abgebildet werden kann.

Beim Echtzeitmatching wird in der ersten Phase mit Hilfe der Lookup-Karte geschaut, in welchen Locations sich der Ball momentan befindet. Mit Hilfe der Zuordnungstabelle wird nun eine Liste von Teilstrategien erzeugt, bei denen die erwartete Ballposition mit der aktuellen Ballposition übereinstimmt.

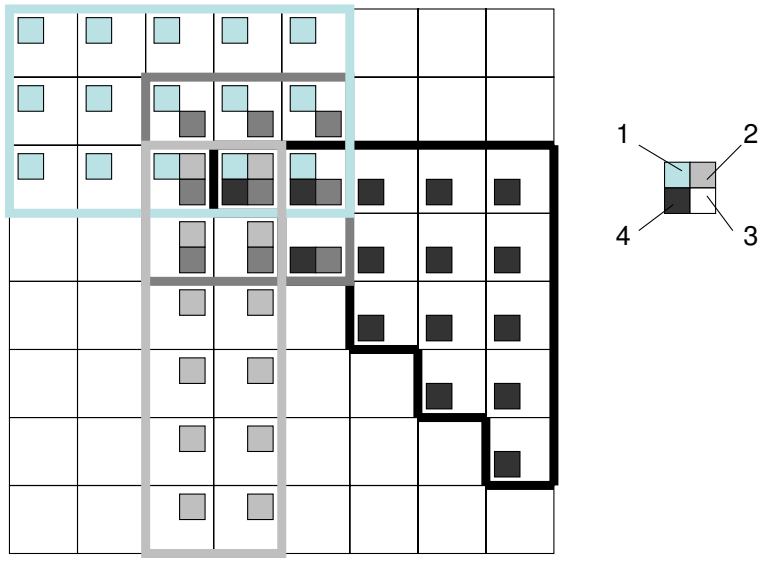

Abbildung 7.12: Karte mit den Bitvektoren. Jedes Quadrat entspricht einem Sektor auf dem Spielfeld. Die Zahlen geben die entsprechenden Bits an, welche in den Sektoren gesetzt sind.

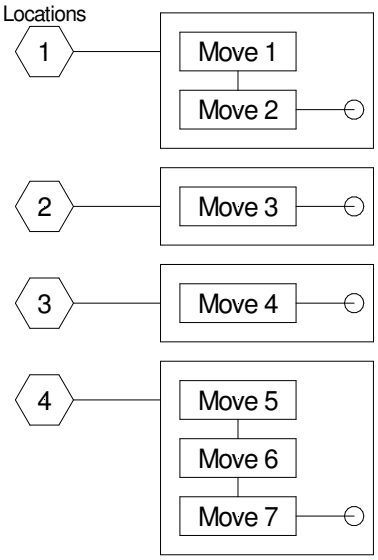

Abbildung 7.13: Karte mit den Move-Lookup

In der nächsten Phase wird nun diese Liste durchlaufen und es werden alle jene Strategien aus der Liste entfernt, bei denen ein erwarteter Mitspieler nicht in der entsprechenden Location ist. Hierbei kann es vorkommen, dass ein Spieler für zwei Positionen benutzt wird, jedoch wird dies in Phase 4 abgefangen.

In der dritten Phase wird die Liste ein weiteres mal durchlaufen und es werden alle Teilstrategien gesucht, wo entweder ein benötigter Gegner nicht in der Location ist oder ein als gegnerfrei benötigtes Gebiet doch von Gegnern besetzt ist. Diese Teilstrategien werden ebenfalls aus der Liste entfernt.

In der vierten und letzten Phase wird nun aus der Menge der verbleibenden Strategien diejenige ausgewählt, welche die höchste Priorität hat. Bei dieser Teilstrategie wird jetzt versucht über einen Backtracking-Algorithmus eine Zuordnung zwischen den Robotern auf dem Feld und denjenigen der Strategie zu finden. Ein effizienterer Algorithmus würde an dieser Stelle keine merkliche Beschleunigung bewirken, da höchstens 4*3*2*1=24 Zuordnungsmöglichkeiten ausprobiert werden müssen. Bei der Zuordnung fließen unter anderem auch die Länge des längsten Laufweges in die Bewertungsfunktion ein, was kurze Laufwege für die Roboter garantiert (siehe Abschnitt 7.3.1.3 und 7.3.1.4 auf der nächsten Seite).

Ist durch den Algorithmus keine mögliche Zuordnung gefunden, dann wird die Teilstrategie aus der Liste entfernt und es wird die nächst vielversprechende gewählt.

Wenn das Matching erfolgreich war, wird die Teilstrategie übernommen und als Ergebnis des Strategy-Choosers in das Blackboard geschrieben. Des weiteren werden noch die entsprechenden Empfehlungen für die Roboter betreffend Bewegung und Aktionen in das Blackboard geschrieben, andernfalls wird keine Empfehlung gegeben und das nachgeschaltete XABSL-Verhalten nimmt das Fallback-Verhalten.

Die folgenden beiden Abschnitte erläutern nun eine textuelle Beschreibung der von uns zum Zuordnen der Spieler verwendeten Algorithmen. Zunächst wird der Zuweisungs-Algorithmus dargestellt, darauf folgt der im Zuordnungs-Algorithmus verwendete Bewertungs-Algorithmus.

7.3.1.3 Der Assignment-Algorithmus

Der Assignment-Algoritmus erhält als Eingabe eine Menge s von Strategien und berechnet einen Ausgabevektor $a = (a_0, a_1, a_2, a_3)$, in dem a_0 die Strategie, a_1 ein Assignment, a_2 den Gesamtlaufweg und a_3 ein Flag (eine Ja/Nein-Aussage) enthält, die besagt, ob ein Assignment (eine Zuordnung der Spieler zu den verschiedenen Zonen) überhaupt möglich war.

1. Setze a_0 auf einen Wert, der eine ungültige Strategie darstellen soll. Setze a_1 auf einen Wert, der ein ungültiges Assignment bezeichnet. Setze a_2 auf $+\infty$. Setze a_3 auf „falsch“.

2. Für jedes s_0 aus s mache folgendes:

   (a) Sei $b = (b_0, b_1, b_2, b_3)$ ein Vektor aus der gleichen Menge wie a. Bewerte die Strategie s_0 mittels des Bewertungs-Algorithmus aus Abschnitt 7.3.1.4 und

schreibe das Ergebnis in b.

(b) Wenn b_3 wahr ist:
Wenn a_3 falsch ist, oder die Priorität der Strategie b_0 größer als die Priorität der Strategie a_0 ist, oder wenn a_0 und b_0 die gleiche Priorität haben und der Laufweg b_2 kleiner ist als der Laufweg a_2, setze $a = b$.

3. Gib a zurück.

7.3.1.4 Der Bewertungs-Algorithmus

Der Bewertungs-Algorithmus erhält als Eingabe eine Strategie s. Er überprüft, ob die Strategie möglich ist, berechnet die Summe der Laufwege der Roboter, und ermittelt mit deren Hilfe eine gültige Zuordnung von Robotern (Assignment) zu den verschiedenen Zonen der Strategie. Der Algorithmus gibt einen Vektor $a = (a_0, a_1, a_2, a_3)$ zurück, in dem a_0 eine Referenz auf die Strategie s enthält, wenn eine Zuordnung möglich war, a_1 verweist auf eine Permutation von Robotern, a_2 bemisst die Summe der Länge der Laufwege der Roboter unter dieser Zuordnung, und a_3 ist ein Flag, das angibt, ob ein Assignment überhaupt möglich war.

Der Algorithmus greift auf die Hilfsmenge π_4 zu, der Menge der Permutationen von vier Robotern. Im folgenden Text wird von so genannten „gültigen“ Permutationen von Robotern gesprochen. In diesem Zusammenhang bedeutet gültig, dass wenn eine Strategie n Roboter benötigt, die Vertauschung der Roboter (also die Anwendung der Permutation auf die Roboter) so ausfällt, dass von den Robotern r_1 bis r_n die Positionsdaten und ihre jeweilige Rolle bekannt sind. Dies impliziert insbesondere, dass sich in den ersten n Robotern nur solche befinden, zu denen WLAN-Verbindungen bestehen.

1. Setze a_0 auf einen Wert, der eine ungültige Strategie darstellen soll. Setze a_1 auf einen Wert, der ein ungültiges Assignment bezeichnet. Setze a_2 auf $+\infty$. Setze a_3 auf „falsch“.

2. Für jedes p aus π_4:

   (a) Vertausche die Roboter $r = (r_1, r_2, r_3, r_4)$, sodass jetzt $r = (r_{p(1)}, r_{p(2)}, r_{p(3)}, r_{p(4)})$ ist.

   (b) Wenn die Vertauschung gültig ist, und r_1 bis r_n jeweils in Zone 1 bis Zone n der Strategie stehen:

      i. Setze l auf die Summe der Laufwege der Roboter r_1 bis r_n zu den Zielpunkten der Zonen 1 bis n.

      ii. Wenn a_3 falsch ist, oder $l < a_2$:

         A. Setze a_0 auf s.

         B. Setze a_1 auf p.

         C. Setze a_2 auf l.

         D. Setze a_3 auf wahr.

3. Gib a zurück.

7.3.2 Strategie-Editor

Bei dem Strategie-Editor handelt es sich um ein externes Programm, mit dem es möglich ist, Teilstrategien und Locations auf einfachem Wege zu erstellen und zu bearbeiten. Es wurde - analog zu XABSL - ein XML-basiertes Dateiformat gewählt, um eine möglichst portable Datenablage zu gewährleisten.

Das Dateiformat wurde in der XML-Schema Sprache definiert und hat etwa folgenden Aufbau:

```
<Locations>

 <LocationArea>
    <! Hier wird eine Location (wie weiter oben erwaehnt) genauer spezifiziert>
 </LocationArea>
    <! Hier koennen weitere LocationAreas folgen>

 <Move>

  <Ball>
    <! Hier wird angegeben in welcher Location der Ball liegen muss>
  </Ball>

  <FriendlyPlayer>
    <! Hier wird eine Location festgelegt, in der ein eigener Spieler stehen>
    <! muss, damit diese Zone gematcht wird>
  </FriendlyPlayer>
    <! Bei Bedarf folgen hier weitere FriendlyPlayers>

  <EnemyPlayer>
    <! Hierfuer gilt das gleiche wie fuer den FriendlyPlayer, es ist aber ein>
    <! Gegner gemeint>
  </EnemyPlayer>

 </Move>

</Locations>
```

Wie sich aus dem Listing leicht entnehmen lässt, handelt es sich beim Element „<Locations>“ um das Root-Element der XML-Datei. Es enthält zunächst eine Liste von „<LocationArea>’s“. LocationAreas beschreiben die Zonen, auf denen die Strategien definiert werden. LocationAreas können bis zu acht Eckpunkte haben, in der Praxis werden jedoch selten mehr als vier benutzt. Die Eckpunkte beschreiben ein Polygon, dass vom StrategyChooser gematcht werden kann.

Nach den LocationAreas folgen die „<Move>’s“. Moves beschreiben die eigentlichen Spielzüge. Ein Move enthält genau ein „<Ball>“-Element, mindestens ein bis maximal vier „<FriendlyPlayer>-“ und bis zu vier „<EnemyPlayer>“ -Elemente.

Das Ball-Element referenziert diejenige Zone, in der der Ball liegen muss, damit die Strategie gematcht werden kann. Das Element ist zwingend erforderlich, da ansonsten der Matching-Algorithmus kein Ergebnis liefern könnte. Ebenso ist es nicht zulässig, ein zweites Ball-Element für den Move anzugeben, da der Ball selbstverständlich nicht an zwei Positionen gleichzeitig sein kann.

Das FriendlyPlayer-Element enthält einen Verweis auf eine Location, an denen sich ein eigener Spieler befinden muss. Außerdem können mit ihr Koordinaten angegeben werden, zu denen sich ein Spieler, der sich in der jeweiligen Zone befindet, bewegen soll. Um Sonderfunktionen zu erlauben, gibt es noch die Möglichkeit, die Flags „dribble“ und

„ballPlayerKick“ zu setzen. Bei dribble bewegt sich der jeweilige Spieler mit dem Ball zu einem Zielpunkt, bei ballPlayerKick wird der Ball in Richtung eines Zielpunkts gespielt.

Das EnemyPlayer-Element nimmt eine Sonderfunktion ein. Neben einer Zone enthält es ein „Inverted“-Flag, das seine Bedeutung angibt. Ist das Flag nicht gesetzt, bedeutet das, dass sich ein gegnerischer Spieler in der EnemyPlayer-Zone befinden muss. Ist es gesetzt, darf sich in der Location kein Spieler befinden. Damit lassen sich Spielzüge formulieren, die vorraussetzen, dass gewisse Spielfeldbereiche frei sind, zum Beispiel um den berühmten „Pass in den freien Raum“ zu realisieren.

Analog zum XABSL muss auch unser Dateiformat vor dem Gebrauch auf dem Roboter transformiert werden. Daher wurde auch hier Gebrauch von einem XSLT-Übersetzungs-Schema gemacht, welches die XML-Datei in eine reine ASCII-Textdatei überführt. Da die gesamte Übersetzung von XABSL mit Tools des LibXML-Teams gemacht wird, welches eigentlich eine Unix-Bibliothek entwickelt, wurde auch hier (wie bei XABSL) ein Unix-Makefile geschrieben, das die Übersetzung erleichtert. Das Makefile beherrscht die drei Standard-Build-Regeln (make clean, make [all] und make install), wobei ein „make install“ die erzeugte ASCII-Datei nach /cygdrive/t/GT2003/Config, also ins Konfigurationsverzeichnis des Projektes, installiert. Eine Strategie-Datei im Config-Verzeichnis muss den Namen „strategy.stp“ tragen, damit sie als solche erkannt wird und wird standardmäßig nicht mit auf den Stick kopiert, wenn „copyfiles.bash“ aufgerufen wird.

7.3.3 Visionary

Die Visionary hat die Aufgabe, mit Hilfe der PlayerPoseCollection, der BallPosition und der RobotPose die „Zukunft vorherzusagen“. Für einen bestimmten Zeitpunkt in der Zukunft berechnet die Visionary die neue Position der Roboter und des Balls auf dem Spielfeld. Dafür wurden mehrere Methoden entwickelt, welche die momentan gesehene bzw. übermittelt bekommene Position aus den drei oben erwähnten Klassen auslesen und aus diesen durch einfache Weg-Zeit-Berechnung geschätzte Positionen für die Zukunft berechnen.

Als Hilfe wurde die Struktur VisionaryPoseCollection implementiert. Sie fungiert in der Visionary als Container für neu berechnete Positionen der AIBOs. Bei ihrer Entwicklung war es angedacht, vier Gegner und sieben eigene Spieler an dieser Stelle unterzubringen. Es werden sieben eigene Spieler benötigt, da es sein kann, das jeder Spieler seine Rolle in die PlayerPoseCollection eingetragen hat. Dies muss aber nicht so sein. Man braucht also vier Speicherplätze für die bekannten Spieler (Goalie, Defender, Striker1 und Striker2) und drei für Spieler ohne angegebene Rolle. Die eigene Rolle befindet sich in der RobotPose, weshalb sie also bekannt ist. Für jeden dieser elf Spieler wird in der VisionaryPoseCollection festgehalten, ob der Eintrag gültig ist oder nicht. Für die gegnerischen Spieler reichen vier Speicherplätze, da es nicht möglich ist die Rolle der gegnerischen Spieler zu erkennen und diese somit immer unbekannt ist. Die Ballposition wird separat in der BallPositionXY als zweidimensionales Feld gespeichert.

Die Visionary besteht des weiteren aus mehreren Methoden, die für die Berechnung der „Zukunft“ ausschlaggebend sind. Als Interface dienen die Methoden getNewRobotPosition(), getNewPlayerPosePosition() und getNewBallPosition() wobei die ersten beiden die

zukünftige Position der Spieler ausgeben und getNewBallPosition() die Position des Balls. Diese Methoden rufen zur Berechnung der Positionen entsprechend die Methoden calculateNewRobotPosition(), calculateNewPlayerPosePosition() und calculateNewBallPosition() auf.

Beispielhaft für diese Methoden wird an dieser Stelle die Funktionsweise der Methode calculateNewRobotPosition() erklärt. CalculateNewRobotPosition() extrahiert aus der RobotPose die Position und Ausrichtung des AIBO, auf dem das Programm läuft und berechnet unter der Annahme, das der AIBO in Blickrichtung mit konstanter Geschwindigkeit weiter läuft, die Position, die der AIBO nach verstrichener Zeit Time einnimmt. Die berechneten Daten werden in der VisionaryPoseCollection gespeichert.

Die beiden anderen Methoden funktionieren Analog. Wobei die Methode calculateNewPlayerPosePosition() die Daten aus der PlayerPoseCollection und die Methode calculateNewBallPosition() die Daten aus der BallPosition extrahiert. Die zuletzt erwähnte Methode calculateNewBallPosition() speichert die errechneten Daten in der BallPositionXY.

Gesetzt wird die Zeit Time für die in die Zukunft geschaut werden soll mittels der Methode setTime(int newTime). Diese Zeit wird in Millisekunden gespeichert und wird immer als relativ zum gegenwärtigem Zeitpunkt angesehen.

7.3.4 PotentialfieldGenerator

Es wurde ein Klassenmodell der Potenzialfeldklassen erzeugt. Dies ist in Abbildung 7.14 auf der nächsten Seite zu sehen.

Die Klasse PotentialfieldGenerator übernimmt die Aufgabe des oben genannten Generators. Es ist eine Ableitung des BlackboardUsers. So hat sie Zugriff auf die Strategieninstanzen und die Potenzialfeldinstanz des Blackboards. Außerdem wird der Potentialfieldgenerator mit einem Referenzsatz des Weltbildes instanziert und hat so lesenden Zugriff auf Ball und Spielerpositionen.

Für die Datenhaltung und Funktionalität des Potenzialfeldes steht die Klasse Potentialfield. Eine Instanz speichert eine feste Instanz der Klasse HexArea und eine Liste von Potentialfieldaspekts, bei der mit Polymorphismus gearbeitet wird.

Eine Repräsentation der Menge der Hexfelder ist die Klasse HexArea. Sie kapselt ein Array von Potentialwerten und Methoden für den Zugriff auf Spielfeldpositionsebene und Hexfeld-Koordinaten.

Die Klasse PotentialfieldAspect kann zu jeder Spielfeldposition eine Potentialänderung angegeben. Da die hier benutzten Aspekte eine gewisse Symmetrie auf ein auf das Spielfeld aufgelegtes Koordinatenkreuz aufweisen, ist die gemeinsame Oberklasse die von PotentialfieldAspect abgeleitete Klasse PotentialfieldAspectCrossSpanned. Sie kann die Koordinaten auf dieses aufgelegte Koordinatenkreuz berechnen und für x- und y-Wert unabhängige Bewertungsfunktionen aufrufen und die Ergebnisse zu einer Potentialänderung verschmelzen.

Die Klasse PotentialfieldPeak ist eine Ableitung von PotentialfieldAspectCross-

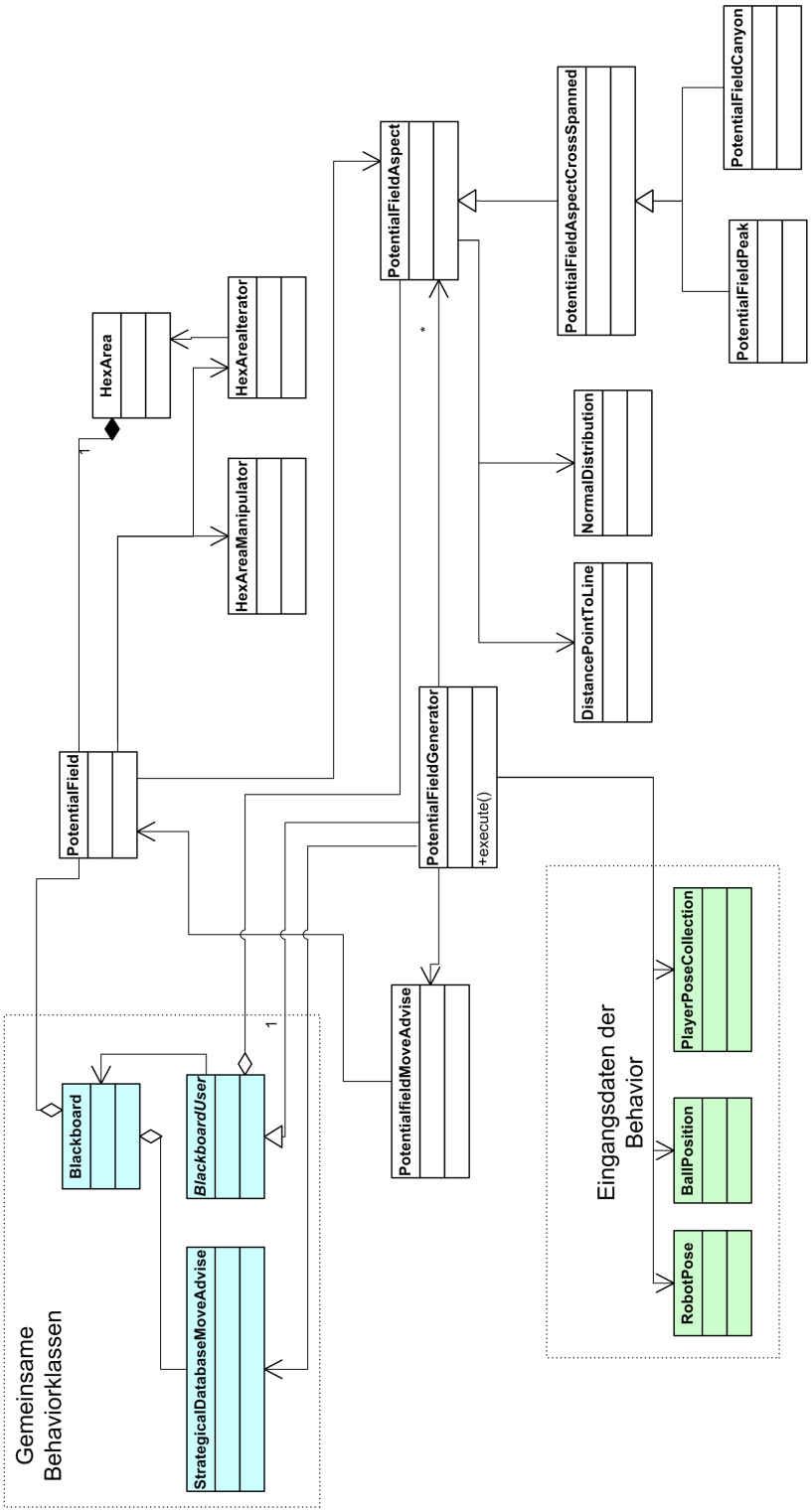

Abbildung 7.14: UML-Klassendiagramm der Potenzialfeldklassen

Spanned und steht für zu vermeidende Hindernisse, also Mit- und Gegenspieler oder zu errechnende Positionen wie der Ball. Beide Bewertungsfunktionen sind durch Gauss-Verteilungen ausgedrückt (siehe Abbildung 7.15 auf der nächsten Seite links).

Die Klasse PotentialfieldCanyon ist eine Ableitung von PotentialfieldAspectCrossSpanned und steht für Strategieempfehlungen sich von einem Punkt zu einem zweiten zu bewegen. Die eine Bewertungsfunktion ist eine Gauss-Verteilung und drückt den Querschnitt des Canyons aus. Die zweite Bewertungsfunktion ist linear und drückt einen linearen Abfall von Punkt A zu Punkt B aus (siehe Abbildung 7.15 auf der nächsten Seite rechts).

Die Handlungsempfehlung des Potenzialfeldes wird von der Klasse PotentialfieldMoveAdvise dargestellt. Eine Datenaktualisierung erfolgt bei jedem Behaviordurchlauf.

Die Klasse HexAreaManipulator bietet die Funktionalität um die HexArea zu verändern. Sie wird beim Hinzufügen und Löschen von Aspekten im Potenzialfeld benutzt. Eine Instanz wird mit dem Potenzialfeldverweis erzeugt. Immer wenn eine Aspektänderung vorgenommen werden soll, wird die Instanz mit dem Aspekt initialisiert, die Hexfelder, die beeinflusst werden, berechnet und dann auf die Felder die Beeinflussung durchgeführt.

Die Klasse HexAreaIterator dient dem oben beschriebenen Durchlauf durch den Hexfeld-Doppelring und kann immer das nächste zu besuchende Feld zurückgeben.

7.3.5 AdvisoryGenerator

Der AdvisoryGenerator wurde eingefügt, um die notwendigen Aufgaben des Roboters selbst und die Aufgaben, die der Trainer-Kern für den Roboter vorsieht, miteinander zu vereinbaren. Die notwendigen Aufgaben des Roboters ist die Selbstlokalisation und die Balllokalisation. Weniger wichtig und selten erforderlich ist die Aufstellung auf dem Spielfeld, Reaktionen auf einen Torschuss, wie irgendwelche Jubelaktionen, und Aktionen am Spielende, wie unser selbsttätiges Zusammenpacken zum leichteren Abtransport. Der Trainer-Kern ist verantwortlich für Aktionen wie die Bewegung zu einer bestimmten Position mit anschließender Rotation, um einen Pass anzunehmen oder in diese Position einen Kick auszuführen. Diese Aufgaben werden durch die gewählte Strategie bestimmt.

7.3.5.1 Vereinbarung der Aufgaben

Die Vereinbarung aller Aufgaben geschieht, indem für jede Aufgabe ein „strict“-Wert bestimmt wird. Dieser Wert sagt aus, wie dringlich es für den Roboter ist, die Aufgabe zu erfüllen und befindet sich zwischen null und eins. Diese Werte werden miteinander verglichen. Die Aufgabe mit dem höchsten Wert wird als Advise an den XABSL-Automaten weitergegeben. Irrtümlicherweise wurde es Advise anstatt „Advice“. Dieser Rechtschreibfehler wurde im Quelltext nicht behoben.

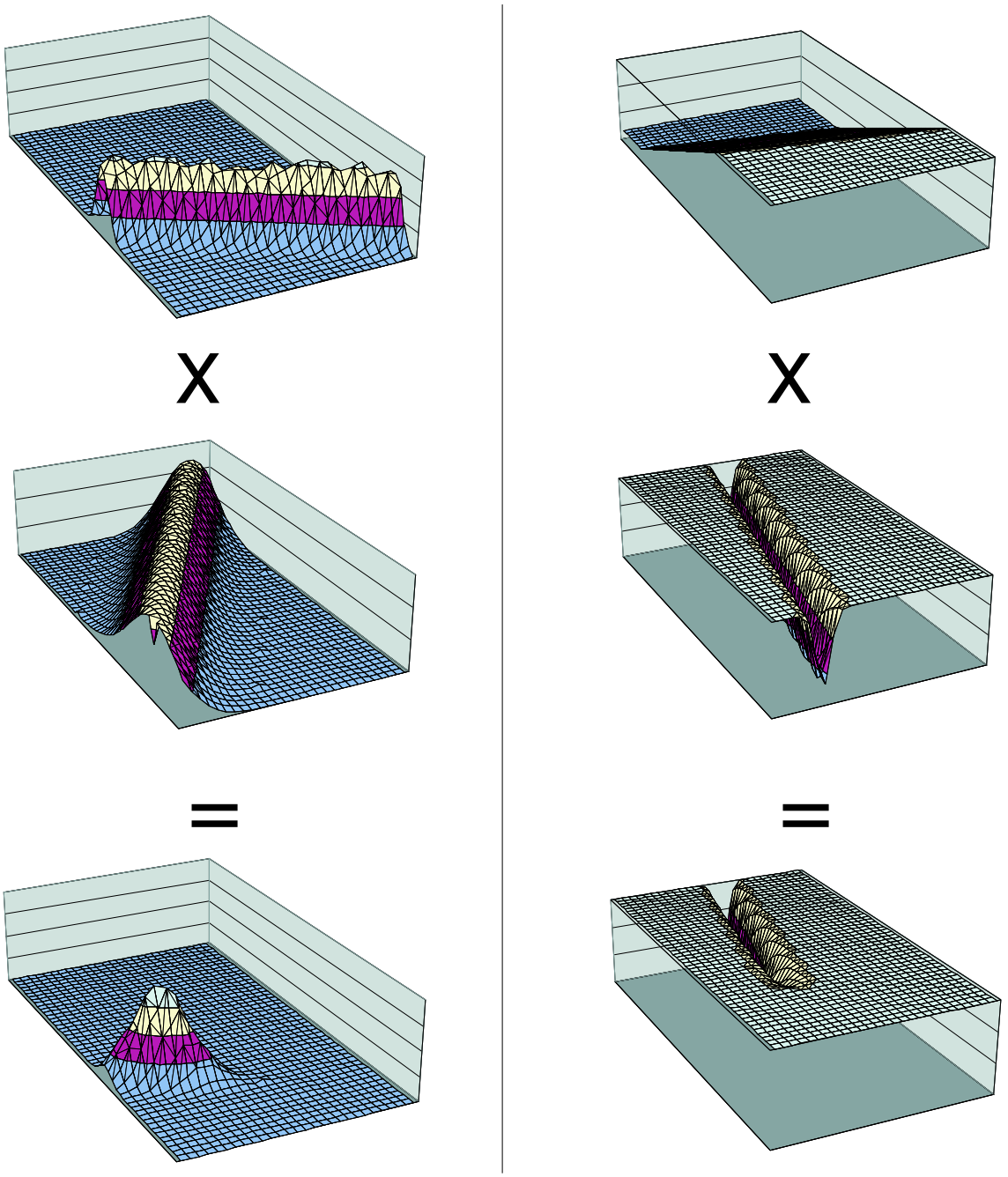

Abbildung 7.15: Verrechnung der Bewertungsfunktionen der orthogonalen Ausprägungsrichtungen - **links**: eines PotentialfieldPeaks - **rechts**: eines PotentialfieldCanyons

7.3.5.2 Aufbau eines Advises

Ein Advise besteht nicht nur aus der zu erfüllenden Aufgabe an sich, sondern besteht aus einer komplexen Struktur. Die Struktur ist wie folgt aufgebaut:

- Order
- strict
- avoidBall
- oldScore
- SActionID
- kickDirX
- kickDirY
- playerOrientation

Jedes Element der Struktur hat eine gewisse Aufgabe, die im folgenden erläutert wird.

Order repräsentiert die Aufgabe, die der Algorithmus wie oben erwähnt gefunden hat. Mögliche Werte sind:

free: Der AdvisoryGenerator hat keine sinnvolle Aufgabe gefunden und es obliegt dem XABSL-Automaten, welche Aktion er ausführen will.

val_own_pos: Der Roboter soll sich auf dem Spielfeld besser lokalisieren.

val_ball_pos: Die Position des Balls ist schlecht oder gar nicht bekannt und soll gefunden werden.

doSpecialAction: Eine „SpecialAction“ soll ausgeführt werden (siehe 7.3.5.2 auf der nächsten Seite).

we_scored: Die eigene Mannschaft hat ein Tor erzielt, was zu einer „Jubel“-Aktion der Roboter führen soll.

we_rule: Die eigene Mannschaft hat ein Tor erzielt und führt mit mindestens 3 Toren. Dies soll ebenfalls zu einer - eventuell. anderen - „Jubel“-Aktion führen.

penalized: Der Roboter wurde vom Schiedsrichter aus dem Spiel entfernt und soll nun bis auf weiteres nichts machen.

ballKick: Der Roboter soll den Ball an eine gegebene Position schießen.

dribble: Der Roboter soll sich mit dem Ball fortbewegen.

strict ist gleich dem vom Algorithmus gefunden Wert für die Aufgabe. Da also „Order“ und „strict“ zusammengehören, wurde es als „Adv“ (Advise) zusammengefasst.

avoidBall ist ein boolean-Wert und gibt an, ob der Roboter zum Ball gehen darf. So wird verhindert, dass mehrere Roboter auf den Ball zugehen und sich gegenseitig behindern. Dies ist wichtig, damit ein Roboter, der den „free“-Advise bekommen hat, nicht zum Ball geht, obwohl ein anderer einen „kick“-Advise hat.

oldScore ist ein Wert, in dem sich der Algorithmus den eigenen Punktestand merkt. Dieser Wert wird benötigt, um zu entscheiden, ob gerade ein Tor von der eigenen Mannschaft geschossen wurde. So kann entschieden werden, ob der Advise we_scored bzw. we_rule ausgegeben werden muss.

SActionID wird im Zusammenhang mit dem doSpezialAction-Advise verwendet und gibt an, welche SpecialAction ausgeführt werden soll. Der Wert ergibt sich aus der gewählten Strategie.

kickDirX und kickDirY enthalten den Punkt, zu dem der Roboter den Ball im Fall eines „kick“-Advises schießen soll und entsprechen ebenfalls den Werten in der Strategie.

playerOrientation gibt die eigene Drehrichtung relativ zum Spielfeld an und ist ebenfalls von der Strategie abhängig. Dies ist aus 2 Gründen sinnvoll:

1. Er schaut in die Ballrichtung, um die Sensorfusion zu unterstützen
2. Er positioniert sich so, das er einen Pass gut annehmen kann

Für den Fall, das der Winkel Null ist, dreht sich der Roboter immer in Richtung Ball. Dies wurde später hinzugefügt, um die Sensorfusion zu unterstützen und ein manuelles Setzen des Winkels nicht notwendig ist. Diese Eigenschaft ist bei der Erstellung der Strategie zu berücksichtigen, indem man, für den Fall dass der Roboter wirklich geradeaus stehen soll, den Winkel auf knapp ungleich Null setzt. Eine Abweichung von einem Grad sollte keinen sichtbaren Einfluss haben.

7.3.5.3 Der Algorithmus zur Advise-Erzeugung

Der Algorithmus arbeitet in mehreren Schritten:

1. Vorbereitungen: Im Konstruktor werden zunächst vier Advises für jeden Roboter im Blackboard angelegt. Geschrieben wird jedoch nur ein Advise für den eigenen Roboter. Durch diese, auf den ersten Blick unnötige Anlegen von Advises ist es möglich, alles Advises von nur einem Roboter anlegen zu lassen und diese an die anderen zu versenden. Dies macht zur Zeit noch keinen Sinn, da die Latenzzeiten bei der Kommunikation zwischen den Roboter zu hoch ist. Die Werte der Advises werden mit sinnvollen Startwerten belegt. Der Konstruktor wird nur einmal beim Einschalten des Roboters aufgerufen.

2. Auslesen der benötigten Daten: Zunächst wird ein Zeiger auf den Advise des eigenen Roboters vom Blackboard ausgelesen und der „Adv“-Teil zurückgesetzt. Dann werden noch die Informationen des Potenzialfeldes und die aktuelle Strategie für den lokalen Roboter aus dem Blackboard ausgelesen.

3. Ballvalidität: Am Anfang der Projektgruppe existierte ein brauchbarer Wert für die Ballvalidität. Dieser Wert wurde jedoch auf konstant Null gesetzt, weswegen hier eine eigene Berechnung stattfinden muss. Sie errechnet sich aus der Zeit, seitdem der Ball vom eigenen Team gesehen wurde. Sie ist Null, wenn sie seit einer konfigurierbaren Zeit nicht mehr wahrgenommen wurde. Im Moment ist der Wert auf vier Sekunden gesetzt.

4. „strict"-Wert und „Order" Belegung: Im folgenden werden „strict"-Werte berechnet. Die Werte werden allerdings nicht direkt gesetzt, sondern mit dem aktuellen „strict"-Wert des Advises verglichen und nur für den Fall, dass der neue Wert größer ist, in den Advise geschrieben und die „Order" des zugehörigen Wertes gesetzt. Dies gilt nicht für Werte, die entweder Null oder Eins sind. Sie überschreiben den Advise sofort, da sie unterbrechend auf den Spielfluss wirken.

   - Penalized: Zunächst wird der „strict"-Wert für den Penalized-Advise berechnet. Entweder der Roboter wurde vom Schiedsrichter mit einer Zeitstrafe belegt oder nicht, also ist der Wert 1 oder 0.
   - Torerzielung: Hier wird der aktuelle Punktestand mit dem Punktestand aus dem zuvor ausgelesen eigenen Advise verglichen. Ist dieser Wert höher, wird die entsprechende Order ausgegeben und der „strict"-Wert auf 1 gesetzt.
   - Eigene Position: Nun wird geprüft, ob die eigene Position gut genug bestimmt ist. Die Validität wird durch das Modul für die Self-location bestimmt. Da jede Solution diesen Wert anders bestimmt und nicht linear zwischen Null und Eins verläuft, muss dieser Wert angepasst werden. Die Anpassung muss also für jede Solution einzeln erfolgen und kann nur durch ausprobieren bestimmt werden. Der „strict"-Wert ergibt sich aus der angepassten Validität minus 1.
   - Ballposition: Hier kann der „strict"-Wert einfach auf die zuvor berechnete Ballvalidität minus 1 gesetzt werden.

5. SpecialAction ausführen: Falls in der Strategie für den Roboter eine SpecialAction gesetzt ist und der „strict"-Wert unter 0.7 ist, wird die entsprechende „Order" gesetzt und die ID der Aktion von der Strategie in das Advise übertragen. Die Abfrage auf 0.7 ist notwendig, da es keinen Sinn macht eine SpecialAction auszuführen, wenn der Roboter nicht weiß, wo er oder der Ball genau ist. Die zuvor geprüften „Order" haben also eine gewisse Priorität. Dies gilt auch für den Folgenden Punkt.

6. BallKick: Hier wird zunächst auf die Distanz zwischen Roboter und Ball geprüft und ob die Strategie einen Kick vorsieht. Die Distanz ist konfigurierbar und liegt zur Zeit bei 400 mm. Die Distanzüberprüfung ist sinnvoll, da der XABSL-Automat nicht mehr das Potenzialfeld berücksichtigt, wenn der kickBall-Advise gesetzt wird und so andere Roboter nicht mehr umgehen kann. Es obliegt dem Potenzialfeld, den Roboter nah genug an den Ball zu führen. Der „strict"-Wert wird auf einen Wert höher als die Potenzialfeldpriorität gesetzt (siehe XABSL-Automat).

7. Finales setzen: Zum Schluss muss noch der „strict"-Wert, für den Fall das dieser zwischen 0.7 und 1 liegt, um die Potenzialfeldpriorität erhöht werden (siehe XABSL-Automat) und der entstandene Advise in das Blackboard kopiert werden.

7.3.6 XABSL

Eine XABSL-basierte Verhaltenssteuerung basiert - wie bereits vorher erwähnt - auf mehreren hierarchisch organisierten Automaten. Schaltet ein Automat in einer Ebene in einen anderen Zustand, wird gewöhnlich auf der darunter liegenden Ebene ein anderer Automat ausgewählt.

Da zu Beginn der Projektgruppenarbeit auf dem Workshop in Darmstadt entschieden wurde, dass die im GermanTeam beteiligten Universitäten bis auf Bremen XABSL zur Verhaltenssteuerung verwenden, wurde ebenfalls beschlossen, XABSL in unser Verhalten zu integrieren. XABSL übernimmt dabei die Steuerung des Spieler-Teils unseres Konzepts.

Zur Implementierung der Spieler wurden verschiedene XABSL2.0-Eingabesymbole geschrieben und eine komplexe Optionshierarchie angelegt. Dabei wurde teilweise auf Ergebnisse der außerhalb der Projektgruppe von Arthur Cesarz und Matthias Hebbel geleisteten Arbeit zurückgegriffen. Abbildung 7.16 auf der nächsten Seite zeigt eine Übersicht über unsere XABSL-Optionen.

7.3.6.1 Symbole

Wie in vorherigen Kapiteln bereits erwähnt ist das Trainer-Spieler Konzept neu. Es existierte keine Schnittstelle zwischen dem Spieler, simuliert durch einen XABSL-Automaten, und dem Trainer. Zu diesem Zweck wurden im Laufe der Projektgruppe-Zeit neue XABSL-Symbole implementiert. Symbole in XABSL sind die einfachste Möglichkeit diese Lücke zu schließen. Die Struktur des XABSL-Automaten wird nicht verändert. Dieser bekommt nur eine größere Funktionsfähigkeit.

In diesem Abschnitt wird nun kurz auf die globalen Aufgaben dieser Symbole eingegangen. Die komplette Auflistung, und Beschreibung dieser Symbole und deren Aufgaben befindet sich im Anhang.

Alle neu hinzugekommenen Symbole haben eins gemeinsam: Ihre Aufgabe ist es die Anweisungen des Trainers für den XABSL-Automaten verständlich zu machen. Dazu wurden zwei Klassen von Symbolen entwickelt.

Der erste Teil dieser Symbole ist für die eigentliche Aufgabe gedacht die Daten des Trainers aufzuarbeiten. Dafür werden die Daten, die der AdvisioryGenerator, das Potentialfield und für Testzwecke auch der StrategieChooser ausgeben, verarbeitet. Zu den Aufgaben dieser Klasse gehören z.B. der Bewegungsrichtung, die das Potentialfield vorschlägt zu folgen, oder auch das Ausführen der Anweisungen des AdvisioryGenerators. Symbole, die auf Daten vom StrategieChooser zugreifen, wiedersprechen zwar der Trainer-Architektur, erwiesen sich aber zum Testen der einzelnen Trainer und Spieler-Komponenten als hilfreich und werden deshalb im Anhang mit angegeben.

Zu der zweiten Klasse gehören Symbole für die Entscheidungsfindung innerhalb des Spielers. Symbole dieser Klasse geben die Prioritäten der einzelnen Anweisungen und der eigenen Interessen aus und können für eine Situation Anweisungen ausgeben, die der Spieler tun soll.So kann der Spieler mit diesen Symbolen entscheiden, ob er auf das Potentialfield hört, lieber zum Ball rennt, oder doch auf den AdvisioryGenerator hört und stehen bleibt.

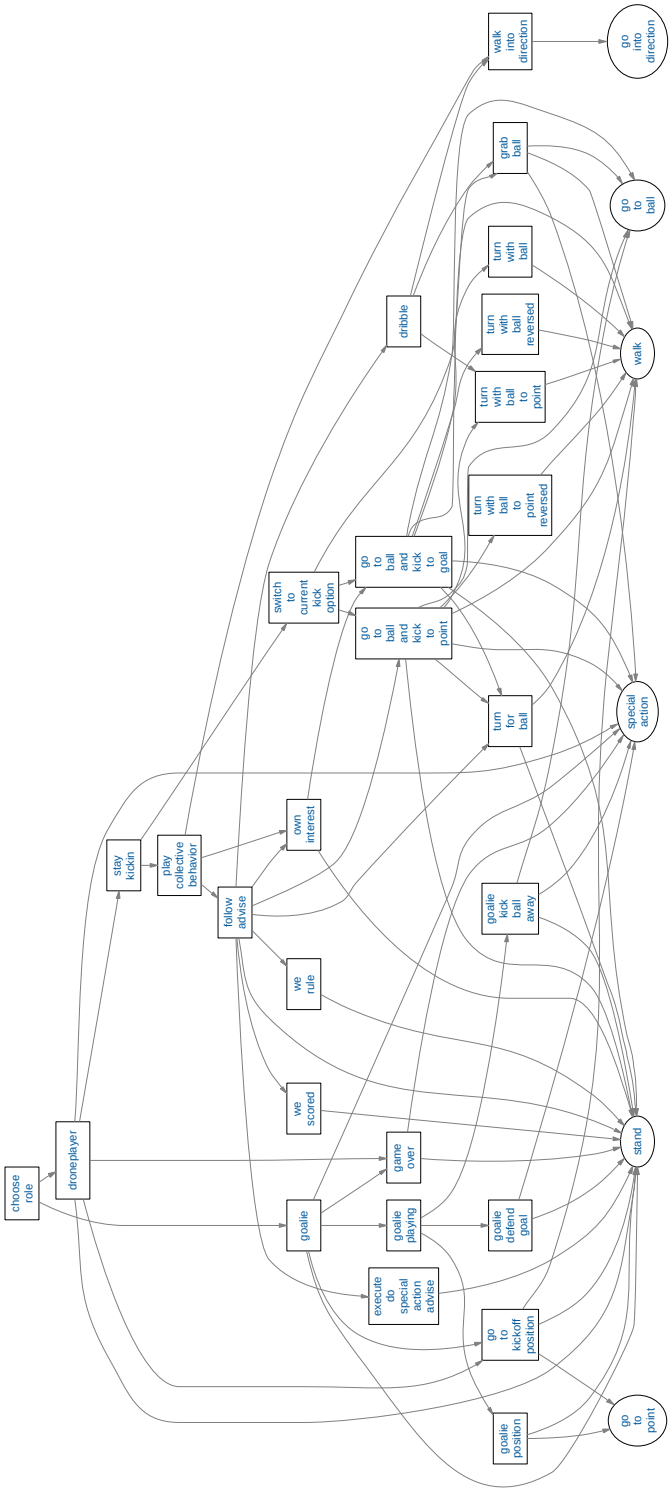

Abbildung 7.16: Ein Überblick über die XABSL Optionen

Auf Optionen, die diese hier vorgestellten Symbole benutzen, wird in dem folgendem Abschnitt eingegangen.

7.3.6.2 Optionen

Ein XABSL-basiertes Verhalten lässt sich vollständig durch die automatisch erzeugte Optionsgrafik (Abbildung 7.16 auf der vorherigen Seite) und eine Erläuterung der Optionen verstehen. Hier folgt nun eine kurze Beschreibung der von uns implementierten Optionen:

- **choose-role.** In dieser Option wird festgestellt, welche Rolle der Spieler auf dem Spielfeld übernimmt. Diese sind zwar für unser Verhalten größtenteils irrelevant, jedoch besteht eine Ausnahme: Der Torwart wird in unserem Konzept sich selbst überlassen. Das hat den Grund, dass der Torwart nach den Regeln mehr Rechte hat als ein normaler Spieler, und um diese zu nutzen, muss er sich in seiner Torwart-Zone aufhalten. Da unser Verhalten jedoch dazu führen könnte, dass der Torwart sich wie jeder andere Spieler verhält (insbesondere den Torraum verlässt), wurde beschlossen, den von Matthias Hebbel entwickelten Torwart in unser Verhalten zu integrieren und den Trainer für den Goalie komplett zu ignorieren (Das heißt übrigens nicht, dass der Torwart in unserem Konzept keine sinnvolle Rolle hat, denn er steht besonders günstig, um Ball-Perzepte für die Sensorfusion zu liefern, die wiederum das Verhalten beeinflusst).

- **droneplayer.** Die Option droneplayer ist ein Clone der alten GermanTeam-Option striker1. Diese managt wie ihre Geschwister (goalie,defender,striker2) den Spielzustand, startet zum Beispiel den Hauptteil des Verhaltens, wenn das Spiel gestartet wird.

- **stay-kickin.** Diese Option wurde eingefügt, weil es in der play-collective-behavior-Option, die früher an dieser Stelle der Hierarchie stand, manchmal bei leichten Schwankungen der Perzepte zu starken Änderungen in der Entscheidungsfindung kommen konnte. Das hatte zur Folge, dass Kicks und andere spezielle Aktionen mittendrin auf einmal abgebrochen wurden, was zur Folge hatte, dass das Spiel extrem unstetig wirkte. Nach dem Einfügen dieser Option merken sich die Roboter, welche Aktion sie gerade ausführen, und springen aus dieser Option wieder direkt dort hinein. Wenn vorher keine Spezialaktion durchgeführt wurde, wird wie früher auch mit play-collective-behavior fortgefahren.

- **play-collective-behavior.** Dies ist die zentrale Entscheidungsoption des Spieler-Verhaltens. Hier werden die Prioritäten des Potenzialfeldes und des AdvisoryGenerators gegeneinander abgewogen, und je nach dem welches höher ist, eine Entscheidung getroffen. Falls beide Prioritäten sehr niedrig sind, kann der Spieler auch eine eigene Entscheidung treffen.

- **walk-into-direction** ist eine Hilfsoption, die bewirkt, dass der Roboter in die Richtung geht, in die das Potenzialfeld ihn lenkt. Da es verschiedene Stellen gibt, an denen der Roboter das Kommando dazu bekommt, braucht man bei Änderungen am Laufverhalten nur diese Option ändern, nicht alle Anderen.

- **switch-to-current-kick-option.** Auch dies ist eine Hilfsoption. Sie wird von stay-kickin benutzt, um die gemerkte Kick-Option anzuspringen.

- **follow-advise** enthält einen auf den ersten Blick sehr kompliziert aussehenden Automaten. Tatsächlich ist ihre Funktion aber relativ simpel: Es gibt für jeden möglichen Advise genau einen Zustand, von dem aus in jeden anderen Zustand verzweigt werden kann. Abhängig vom Zustand des Automaten wird dann in die dem Advise-Typ entsprechende Folgeoption gewechselt.

- **go-to-ball-and-kick-to-point** ist ein Clone von Arthur Cesarzs go-to-ball-and-kick-to-goal, die so verändert wurde, dass ihr ein Koordinatenpaar übergeben werden kann, das einen Zielpunkt bestimmt, in dessen Richtung der Ball gespielt wird.

- **execute-do-special-action-advise.** Diese Option wurde für spezielle Tricks entworfen, die man den Robotern beibringen kann (z.B. trauern, wenn ein Gegentor kassiert wurde). Im Moment ist sie jedoch bloß ein Platzhalter, der noch mit Leben gefüllt werden muss, sobald der AdvisoryGenerator do-special-action-Advises erzeugt.

- **we-scored** wird bei einem „we-scored"-Advise aufgerufen, also wenn wir ein Tor geschossen haben. Im Moment ist auch sie Platzhalter für ein Demo-Verhalten.

- **we-rule.** Analog zu den zwei vorherigen Optionen macht auch diese Option im Moment nichts, und ist für zukünftige Spezialaktionen gut, wenn wir mehr als zwei Tore in Führung liegen.

- **own-interest.** Wenn Potenzialfeld und AdvisoryGenerator keine vernünftigen Vorschläge machen, oder es dem Verhalten explizit befohlen wird, tritt diese Option in Kraft. Sie bewirkt, dass der Roboter, wenn er dem Ball am nächsten ist, zum Ball geht, und ihn Richtung Tor schießt.

- **dribble** bewirkt, dass der Roboter zum Ball geht, ihn nimmt, sich in Richtung eines angegebenen Zielpunkts dreht und mit dem Ball zum Zielpunkt läuft.

- **grab-ball** ist eine Hilfsoption für dribble. Der Roboter läuft mit Hilfe des Potenzialfeldes zum Ball und hält ihn fest, um sich zu drehen.

- **turn-with-ball-to-point und turn-with-ball-to-point-reversed** sind Hilfsoptionen, die bewirken, dass der Roboter sich mit dem Ball in Richtung eines Punktes dreht (rechts oder links herum).

7.4 Fazit

Eines der größten Probleme während der Entwicklung der Behavior war, dass sich in den davor liegenden Ebenen ständig Änderungen ergeben haben. Diese wurden kaum diskutiert und meistens einfach ohne mögliche Konsequenzen zu überprüfen in den Code eingebaut. Das plötzliche Verschwinden der Ballvalidität ist nur ein Beispiel dafür.

Ein weiterer Schwachpunkt ist, dass die in die Behavior kommenden Daten kaum auf Stimmigkeit geprüft sind. Dies führte aufgrund der Komplexität unseres Verhaltens zu vielen kaum vorhersehbaren Abstürzen, die zur Folge hatten, dass schließlich die Visionary, eine Klasse, die ursprünglich zur internen Benutzung des TrainerCores vorgesehen war, zu einer generellen Filterklasse umgebaut wurde, die nun auch springende und unmögliche Wahrnehmungen ausfiltert. So wurden viele „Division durch Null“- und „Not-A-Number“-Fehler ausgefiltert. Ein weiteres Beispiel für merkwürdige Eingangsdaten waren in Bogenmaß angegebene Winkel, die so groß waren, dass sie nahe den Maximalwerten eines doubles lagen. Solche Werte haben aufgrund der beschränkten Mantissengenauigkeit von doubles dann natürlich keinerlei Aussagekraft mehr, den wirklichen Winkel muss man dann mehr oder weniger kreativ raten.

Leider konnte auch einer der großen Vorteile des Konzepts, das Vermeiden anderer Roboter, nicht ausgeschöpft werden, da keine zuverlässige Gegnererkennung erreicht werden konnte.

Viel schwerwiegender ist jedoch noch ein konzeptioneller Fehler im Design des Verhaltens: der positive Effekt, der durch die Sensorfusion zu erwarten war, wurde schlicht überschätzt. Dies liegt daran, dass die Sensorfusion nur im begrenzten Maße stabilisierend auf die Perzepte wirken kann, da sie selbst von der Selbstlokalisation der Roboter abhängig ist.

Das Verhalten funktioniert im Simulator sehr gut, auf dem Feld wirkt es jedoch sehr ruckartig und konzeptionslos. Der Grund dafür ist, dass Strategien nicht konsequent verfolgt werden können, wenn bei jedem Durchlauf durch das Strategie-Matching eine andere Strategie gefunden wird. Zudem kann es Probleme beim Assignment geben, wenn mal ein Roboter, mal ein Anderer näher zum Ball oder einer Zone steht.

Dennoch besteht vielleicht eine kleine Chance, dass das Verhalten einmal produktive Ergebnisse hervorbringen wird: Wenn mit der nächsten - bereits angekündigten - Open-R-Version endlich die Möglichkeit besteht, Bilder mit höheren Auflösungen von den Robotern zu bekommen, und vielleicht noch die Selbstlokalisierungsalgorithmen verbessert werden, ist vermutlich eine Steigerung der Effizienz des Verhaltens zu erreichen. Wenn die Ergebnisse der University of New South Wales, die angeblich schon Bilder höherer Auflösung benutzt, bei der Localization-Challenge auf der WM in Padua betrachtet werden, lässt sich feststellen, dass die Selbstlokalisierung des GermanTeams noch stark verbessert werden kann. Damit sollten dann stabilere Sensor-Fusion-Perzepte machbar sein, auf die das Verhaltenskonzept so dringend angewiesen ist.

Kapitel 8

Ball-Challenge

Bei der Weltmeisterschaft im Roboterfußball (RoboCup) finden neben den eigentlichen Fußballspielen auch noch so genannte Challenges statt. Dabei handelt es sich um Wettbewerbe, an denen die einzelnen Teams teilnehmen. Die dabei gewonnenen Erkenntnisse können auch in Zukunft bei den Spielen verwendet werden. Auf diese Weise nähert sich der Roboterfußball immer näher dem Ziel an, unter realen Bedingungen Fußball spielen zu können.

Dieses Kapitel beschäftigt sich mit der Ball-Challenge. Zuerst wird die Problemstellung erläutert, anschließend werden verschiedene Lösungsmöglichkeiten (insbesondere mit Blick auf die Ball-Erkennung und Verhalten) vorgestellt und auf deren Vor- bzw. Nachteile eingegangen.

8.1 Einleitung

Das Ziel der Ball-Challenge ist es, einen schwarz-weißen Ball zu erkennen und in das gelbe Tor zu schießen. Dabei befinden sich lediglich der Roboter, der die Challenge bestreitet, und ein schwarz-weißer Ball auf dem Spielfeld. Sowohl die Lage des Balls als auch die des Roboters wird einmal zufällig bestimmt und ist anschließend für alle Teams gleich. Es gibt eine Zeitbeschränkung von drei Minuten, um diese Aufgabe zu erfüllen. Die Teams werden nach folgenden Gesichtspunkten beurteilt:

1. Es wurde ein Tor erzielt:
   - Ausschlaggebend ist die Zeit, die benötigt wurde, um den Ball ins Tor zu befördern.
2. Es wurde kein Tor erzielt:
   i) Der Roboter hat den Ball innerhalb der drei Minuten berührt:
      - Ausschlaggebend ist der Zeitpunkt des ersten Kontakts.

   ii) Der Roboter hat den Ball nicht innerhalb der drei Minuten berührt:

– Ausschlaggebend ist die Distanz des Roboters zum Ball bei Ablauf der drei Minuten.

Im normalen Spiel wird der Ball durch Farbsegmentierung erkannt. Er ist das einzige Objekt auf dem Spielfeld, das orange ist. Schwieriger wird es bei einem schwarz-weißen Fußball, da die Banden weiß sind und auch ein Schattenwurf als Schwarz erkannt werden kann. Darüber hinaus kann die Bande auch schwarze Striche haben, die leicht enstehen, wenn ein Roboter vor die Bande läuft. Insofern hat sich eine Erkennung, die ausschließlich auf den Farbwerten des segmentierten Bildes arbeitet, als nicht (gut) brauchbar erwiesen.

8.2 Strategien zur Erkennung eines schwarz-weißen Balls

Zur Erkennung des schwarz-weißen Balls wurden parallel vier verschiedene Ansätze entwickelt und implementiert:

- Ballerkennung mit Hilfe von Wavelets
- Ballerkennung mit Hilfe von Convolution-Filtern
- Ballerkennung mit Hilfe einer Hough-Transformation für Kreise im Kantenbild.
- Ballerkennung mit Hilfe eines modifizierten BallSpecialists

Auf diese vier Ansätze wird in den folgenden Unterkapiteln näher eingegangen.

8.2.1 Ballerkennung mit Hilfe von Wavelets

In diesem Ansatz wurde versucht, zunächst die Übergänge zwischen schwarz und weiß zu erkennen. Dazu wurden die Bilddaten durch eine zweidimensionale Wavelet-Transformation in Frequenzdaten umgewandelt. Niedrige Bildfrequenzen entsprechen einem gleichmäßigen Verlauf und hohe Bildfrequenzen entsprechen starken Kontrasten bzw. dem Übergang zwischen verschiedenen Farben. Da für die Erkennung von schwarz-weißen Bällen nur die Luminanz relevant ist, wurde die Wavelet-Transformation auch nur auf den Luminanz-Daten (also dem Y-Kanal) durchgeführt.

8.2.1.1 Das Wavelet-Verfahren

Eine eindimensionale Wavelet-Transformation besteht generell aus einem Tiefpass- und einem Hochpass-Filter. Jeder Filter hat eine gewisse Anzahl von Koeffizienten $l_0, ..., l_{L-1}$ (Tiefpass) und $h_0, ..., h_{H-1}$ (Hochpass). Hier die Formel der eindimensionalen Wavelet-Transformation:

$$\omega_L(x) = \sum_{i=0}^{L-1} f(2x+i) \cdot l_i \tag{8.1}$$

$$\omega_H(x) = \sum_{i=0}^{H-1} f(2x+i) \cdot h_i \tag{8.2}$$

Dabei ist $\omega_L(x)$ der Tiefpass-(Formel 8.1) und $\omega_H(x)$ der Hochpass-Filter (Formel 8.2).

Das einfachste Wavelet ist das Haar-Wavelet, das für beide Filter nur je zwei Koeffizienten hat:

$$l = (\frac{1}{2}, \frac{1}{2}) \tag{8.3}$$

$$h = (\frac{1}{2}, -\frac{1}{2}) \tag{8.4}$$

Dabei entspricht der Tiefpass (Formel 8.3) dem arithmetischen Mittel und der Hochpass (Formel 8.4) der Differenz. Das Haar-Wavelet hat eine schnelle Laufzeit, kann mit Integer-Arithmetik implementiert werden und liefert trotz seiner Schlichtheit zufrieden stellende Resultate.

Die zweidimensionale Wavelet-Transformation besteht aus zwei Schritten; im ersten wird die eindimensionale Transformation auf den horizontalen Bilddaten durchgeführt, im zweiten vertikal auf den transformierten Werten. Die so entstehenden vier Bilder werden als LL, LH, HL und HH bezeichnet, wobei das L für Tiefpass und das H für Hochpass steht.

Da das Haar-Wavelet nur wenige Koeffizienten hat, kann eine zweidimensionale Transformation auch in einem Schritt durchgeführt werden:

$$\begin{aligned}
\omega_{LL}(x,y) &= \frac{1}{4} \cdot \Big(f(2x,2y) + f(2x+1,2y) + f(2x,2y+1) + f(2x+1,2y+1) \Big) \\
\omega_{LH}(x,y) &= \frac{1}{4} \cdot \Big(f(2x,2y) + f(2x+1,2y) - f(2x,2y+1) - f(2x+1,2y+1) \Big) \\
\omega_{HL}(x,y) &= \frac{1}{4} \cdot \Big(f(2x,2y) - f(2x+1,2y) + f(2x,2y+1) - f(2x+1,2y+1) \Big) \\
\omega_{HH}(x,y) &= \frac{1}{4} \cdot \Big(f(2x,2y) - f(2x+1,2y) - f(2x,2y+1) + f(2x+1,2y+1) \Big)
\end{aligned}$$

Werden diese Formeln für ein YUV-Bild angewandt, so entsteht das Wavelet-Bild in Abbildung 8.1 auf der nächsten Seite, das sich aus den vier Teilbildern LL, LH, HL und HH zusammensetzt.

Das LL-Bild kann als Ausgangsbild für eine weitere Wavelet-Iteration (siehe Abbildung 8.1 auf der nächsten Seite) dienen. Es ist sinnvoll, zwei Iterationen durchzuführen, da so auch Kanten entdeckt werden können, die mehr als zwei Pixel breit bzw. hoch sind:

LL ist das arithmetische Mittel von vier Pixeln und wenn zwei benachbarte LL-Werte stark voneinander abweichen, so deutet dies auf eine Kante hin. Es könnten noch mehr als zwei Iterationen durchgeführt werden, jedoch haben Tests gezeigt, dass dies kaum noch Vorteile, allerdings eine größere Laufzeit mit sich bringt.

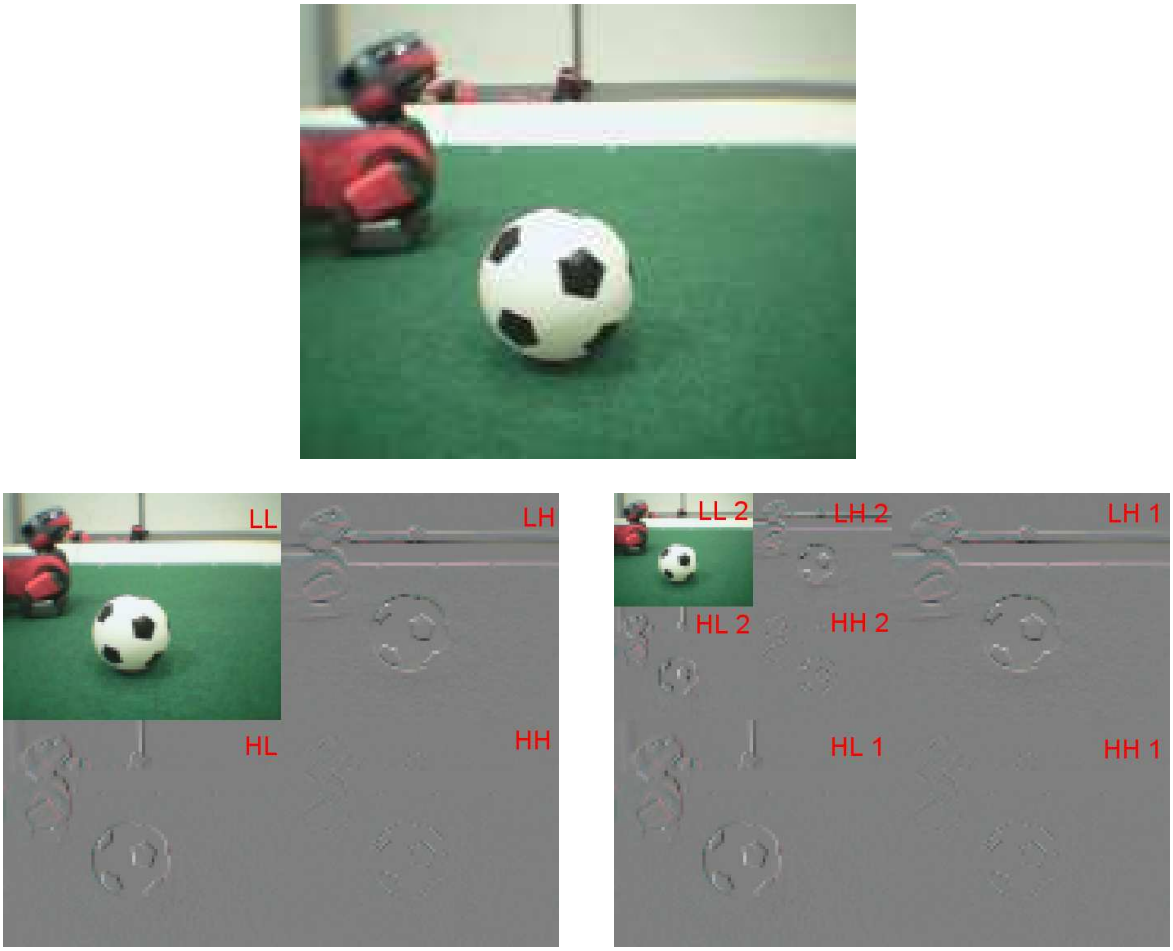

Abbildung 8.1: Wavelet-Bild. Oben das Ausgangsbild, links nach der ersten und rechts nach der zweiten Iteration

Auffällig ist, dass im LH-Bild horizontale, im HL-Bild vertikale und im HH-Bild diagonale Kanten stark von 0 abweichende Wavelet-Werte erzeugen.

Da für die Erkennung des schwarz-weißen Balls vor allem diagonale Kanten relevant sind (bedingt durch die fünfeckigen schwarzen Bereiche auf dem Ball), wurden nur die HH-Bilder weiter untersucht.

Treten dort Werte auf, die mehr als ein gewisser Schwellenwert t von 0 entfernt sind (also $|\omega_{HH}| > t$ ist), kann dort eine Kante vermutet werden. Experimente ergaben für $t = 8$ eine recht gute Kantenerkennung.

Entdeckte Kanten in den beiden HH-Bildern werden in Abbildung 8.2 auf der nächsten Seite als gelbe, orange oder rote (je nach Intensität) Pixel dargestellt. Zusätzlich werden sie – zur Kontrolle mit den Bilddaten – auch in den beiden LL-Bildern als gelbe, orange

oder rote (für HH-Werte aus der ersten Iteration) bzw. als blaue und grüne (für HH-Werte aus der zweiten Iteration) Pixel angezeigt.

Die auf diese Weise ermittelten Punkte werden in einem zweidimensionalen Array, der so genannten Landschaft, eingetragen. Dabei wird im Radius von vier Pixeln ein Berg erzeugt. Auf diese Weise entstehen in der Landschaft verschieden grosse Gebirge (siehe Abbildung 8.2).

Im letzten Schritt wird das größte zusammenhängenden Gebirge gesucht und der Mittelpunkt sowie der Radius bestimmt. Diese stimmen dann mit Mittelpunkt und Radius des vermuteten Balls überein (Abbildung 8.2).

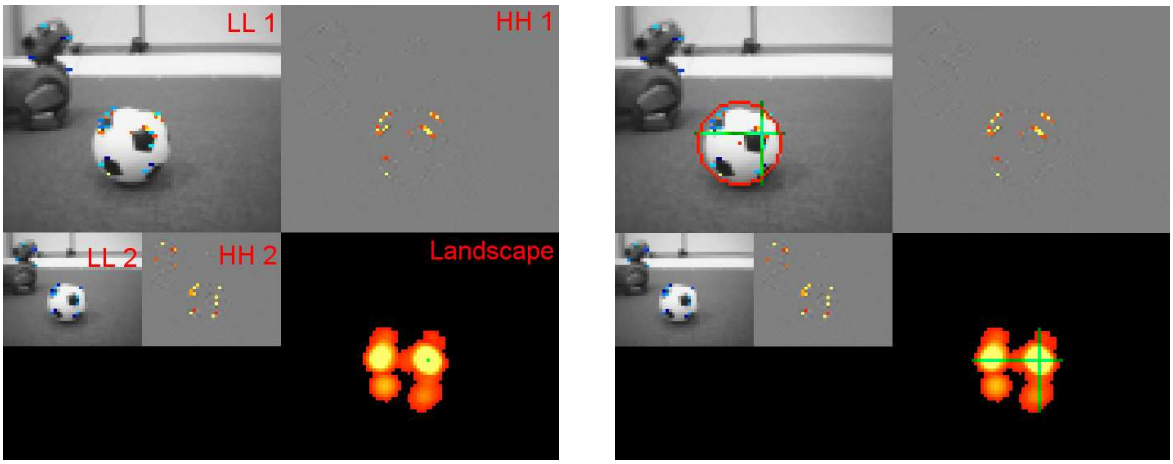

Abbildung 8.2: Landschaft. Links mit Gebirgen und rechts mit Mittelpunkt und Radius

8.2.1.2 Vorteile des Wavelet-Ansatzes

Die Vorteile des Wavelet-Ansatzes bestehen in der völligen Unabhängigkeit von einer Farbtabelle und der schnellen Laufzeit. Schwarz-weiße Bälle, die nicht allzu nah vor dem Roboter liegen, werden nahezu immer erkannt. Auch weit entfernte Bälle werden meist richtig lokalisiert.

8.2.1.3 Nachteile des Wavelet-Ansatzes

Problematisch bei diesem Ansatz ist, dass teilweise auch die Feldlinien auf dem Rasen diagonale Kanten bilden, die dann als Bälle fehlinterpretiert werden. Abhilfe könnte in diesem Fall ein dem Algorithmus nachgeschalteter Farbtabellen-Test schaffen.

Das größte Problem stellen jedoch sehr nahe, nicht komplett sichtbare Bälle dar. Hier werden zwar die einzelnen Übergänge von schwarz nach weiß richtig erkannt, allerdings entstehen in der Landschaft keine zusammenhängenden Gebirge. Eine mögliche Lösung dieser Problematik wäre die Ersetzung des simplen Landschaft-Ansatzes durch eine Hough-Transformation.

8.2.2 Ballerkennung mit Hilfe von Convolution-Filtern

Da die einfachen Algorithmen nicht flexibel genug waren, um das Problem der Ballerkennung zu lösen, wurden komplexere Algorithmen implementiert. Eine mögliche Pipeline für die Ballerkennung sieht dann wie folgt aus:

- Transformieren oder Filtern des Pixelbildes, um die Ballkante besser zu erkennen
- Selektieren von Punkten auf der Ballkante
- Berechnen der Kreisdaten aus drei Kantenpunkten.

Das Prinzip eines Kantenfilters [17] [18] beruht auf dem Vergleichen benachbarter Pixel. Sind sich die Farbwerte sehr ähnlich, so wird im Zielbild z.B. ein schwarzer Pixel gesetzt. Sind sich der Pixel nicht ähnlich, wird ein weißer Pixel gesetzt. Das Vergleichen von zwei Pixeln geschieht durch eine simple Subtraktion der Farbwerte der Pixel. Ist der Absolutbetrag der Differenz sehr nahe bei Null, so befindet sich an dieser Stelle wahrscheinlich keine Kante. Ist der Absolutbetrag der Differenz groß, handelt es sich bei diesem Pixel möglicherweise um einen Kantenpixel. In der Praxis werden nicht je zwei Pixel, sondern je zwei Blöcke von 3x3 Pixel verglichen.

Diese Operation wurde als Abbildung des Originalbildes (Abbildung 8.3 auf Seite 110) auf das Zielbild aufgefasst, die sich durch eine Matrix beschreiben lässt. Bei der Auswahl der Matrixparameter wurde versucht, die Ballkante mit möglichst geringem Aufwand gut erkennbar zu machen. Da z. B. der Y-Kanal das Bild als Grauwertbild repräsentiert, reichte es völlig aus, den Kantenfilter auf diesem Kanal durchzuführen und die Kantenpixel mittels eines simplen Threshhold-Algorithmus zu selektieren. Zusätzlich wurden Nebenbedingungen definiert, um einmal gefundene Ballkanten zu verifizieren. Damit ein Pixel als Ballkante bestätigt wird, müssen genügend Pixel links und rechts von ihm im Originalbild (Abbildung 8.3 auf Seite 110) die Farben Grün und Weiß/Schwarz enthalten.

Es wurden folgende Filtermatrizen untersucht:

Sobel Horizontal (Abbildung 8.3 auf Seite 110):

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} \\ s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \end{pmatrix} := \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Sobel Vertikal (Abbildung 8.3 auf Seite 110):

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} \\ s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \end{pmatrix} := \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

Prewitt Horizontal (Abbildung 8.3 auf Seite 110):

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} \\ s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \end{pmatrix} := \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

Prewitt Vertikal (Abbildung 8.3 auf der nächsten Seite):

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} \\ s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \end{pmatrix} := \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

Median Diagonal (Abbildung 8.3 auf der nächsten Seite):

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} \\ s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \end{pmatrix} := \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

8.2.3 Fazit

Der Algorithmus, mit dem wir den Ball erkennen (siehe Abschnitt 8.2.4) benötigt einen Filter, der möglichst viele Punkte auf der Ballkante erkennt und möglichst wenige Punkte, die nicht auf der Ballkante liegen. Wir haben uns dann für den Sobel-Vertikal Filter entschieden, da er unseren Anforderungen am besten entsprach.

8.2.4 Ballerkennung mit Hilfe einer Hough-Transformation

Wie oben erwähnt kann der Ball nicht ausschließlich über seine Farbe erkannt werden. Hilfreich ist aber die Erkenntnis, dass der Ball das einzige runde Objekt auf dem Spielfeld ist. Doch im Gegensatz zum menschlichen Gehirn kann ein Computer runde Formen nicht ohne weiteres erkennen. Abhilfe schafft allerdings die Hough-Transformation für Kreise [19]. Um die Hough-Transformation anwenden zu können, werden zunächst möglichst viele Randpunkte des Balles gesucht. Diese werden über einen Kantenfilter gefunden. Allerdings werden auch viele andere Kanten des Bildes gefunden. Diese müssen mittels Plausibilitätstests wieder verworfen werden. Somit gliedert sich die Erkennung eines schwarz-weißen Balles in folgende drei Teilaufgaben:

1. Finden von Kanten bzw. Kantenpunkten mit Hilfe eines Kantenfilters
2. Selektion aller gefundenen Kantenpunkte, die höchstwahrscheinlich auf dem Rand des Balles liegen
3. Anwendung der Hough-Transformation auf diese selektierten Punkte

Zum Auffinden der Kanten wurde auf das Bild im Y-Kanal ein Kantenfilter angewendet. Dabei wurde ein vertikaler Sobel-Filter verwendet, da dieser für das vorliegende Problem

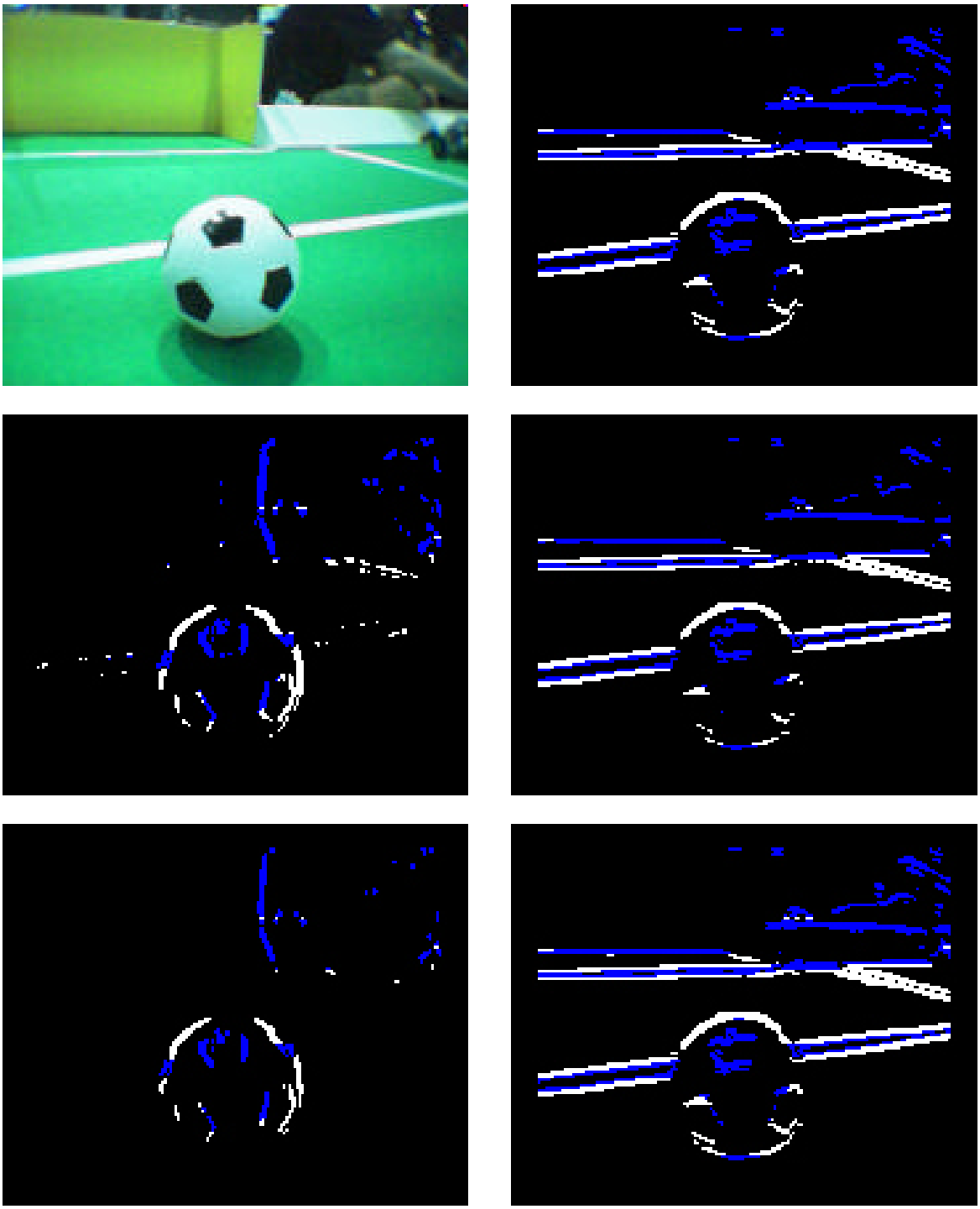

Abbildung 8.3: Convolution-Filter. Ausgangsbild, in dem die Ballkantenpixel erkannt werden sollen, Sobel Horizontal, Sobel Vertikal, Prewitt Horizontal, Prewitt Vertikal und Median Diagonal (von links nach rechts und von oben nach unten)

bessere Ergebnisse lieferte als andere getestete Filter. Seine Filter-Matrix sieht wie folgt aus:

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} \\ s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \end{pmatrix} := \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

Dieser Filter hat die Eigenschaft, dass er alle vertikalen Kanten detektiert. Da der Ball rund ist und somit sowohl horizontale als auch vertikale Kantensprünge aufweist, ist es für das Ergebnis der Hough-Transformation egal, ob ein waagerechter oder ein senkrechter Filter anwendet wird. Die vertikale Filterung ist allerdings besser geeignet als die horizontale, da in fast allen Bildern die Bande, die in Blickrichtung des Roboters liegt, mit dem Rasen eine lange horizontale Kante bildet. Da die Banden aber nur wenige vertikale Kantensprünge hervorrufen, wurde schließlich der oben genannte Sobel-Filter verwendet. Zur eigentlichen Filterung wird der Teil des Bildes, der unterhalb des Horizonts liegt, komplett durchlaufen. Dies bewirkt eine deutliche Laufzeitreduzierung gegenüber einem Durchlauf durch das komplette Bild und ist zulässig, da der Ball stets unterhalb des Horizonts liegt. Für jeden durchlaufenen Bildpunkt $P_{x,y}$ wird ein Wert pY wie folgt ausgerechnet:

$$pY = \sum_{i=0}^{2} \sum_{j=0}^{2} (s_{i,j} \cdot P_{x+i-1,y+j-1})$$

pY ist somit ein Maß für die Stärke des Helligkeitssprungs im Punkt $P_{x,y}$. Wenn pY einen bestimmten Schwellwert überschreitet, liegt eine Kante vor.

Aus der gefundenen Menge von Bildpunkten, die auf einer Kante liegen, werden diejenigen herausgesucht, die sehr wahrscheinlich auf dem Rand des Balles liegen. Für diese müssen folgende 3 Bedingungen gelten:

a) Es müssen genügend grüne Pixel in der Nähe sein. Schließlich liegt der Ball ja auf dem grünen Teppich.

b) Es müssen genügend weiße Pixel in der Nähe liegen.

c) Es müssen genügend schwarze Pixel in der Nähe liegen. Die schwarze Farbe ist ein wesentlicher Bestandteil des Balles. Da auch durch Schattenwurf einzelne Bildpunkte als schwarz segmentiert werden, müssen mindestens 2 Pixel in der Nähe schwarz sein.

Um die Pixel in der Nachbarschaft zu prüfen, wird sowohl um 10 Pixel nach links und rechts gescannt als auch um 5 Pixel nach unten (beim Suchen nach grünen Pixeln) bzw. oben (beim Suchen nach schwarzen Pixeln). Alle gefunden Kantenpunkte, die nicht den genannten Bedingungen entsprechen, werden verworfen.

Im folgenden Schritt wird versucht, aus den nun verbliebenen Kantenpunkten Informationen über die Lage des Balles im Kamerabild zu bekommen. Wenn kein Ball im Sichtfeld des Roboters liegt, soll auch diese Tatsache zuverlässig erkannt werden. Aus der Lage des Balles im Bild und der Kopfposition des Roboters kann man die Position des Balles auf

dem Spielfeld ermitteln. Zur Erkennung eines Balles im Bild wurde eine auf Kreisen arbeitende Hough-Transformation verwendet. Diese Transformation hat einige Vorteile [20] gegenüber anderen Verfahren:

1. Sie ist unanfällig gegenüber stark verrauschten Daten. Dies bedeutet, dass Messtoleranzen kaum Auswirkungen haben. Dieser Punkt kommt der noch relativ geringen Auflösung der verwendeten Kameras zu Gute.

2. Verfälschte Daten haben keinen Einfluss auf das Ergebnis des Algorithmus. Dies ist ein großer Vorteil, denn der Algorithmus scheitert erst dann, wenn mehr falsche als richtige Daten vorhanden sind, da bei ihm keinerlei Mittelwertbildung durchgeführt wird. Das Auftreten eines solchen Falls ist allerdings überaus unwahrscheinlich.

Bei der Hough-Transformation wird neben dem originalen Bild (auch Bildbereich genannt) noch ein weiteres, das so genannte transformierte Bild (oder Parameterbereich), verwendet. Es gibt eine spezielle Variante dieser Transformation, mit der aus einer Menge von Punkten die Teilmenge aller derjenigen Punkte bestimmt werden kann, die auf einem Kreis liegen. Dazu wird ausgenutzt, dass ein Kreis im Bildbereich mit einem Punkt im Parameterbereich korrespondiert (und umgekehrt). Dahinter steckt folgende Idee: Alle Punkte K_i auf dem Rand eines Kreises K haben von dessen Mittelpunkt M denselben Abstand. Wenn man nun also um einen Randpunkt einen Kreis mit dem Radius des gegebenen Kreises zeichnet, so verläuft dieser auf jeden Fall durch den Mittelpunkt M von K. Das gilt für alle Randpunkte. Konstruiert man nun für mehrere gegebene Randpunkte einen solchen Umkreis, so schneiden sich alle diese konstruierten Kreise in M. Darüber hinaus gibt es noch eine große Anzahl weiterer Schnittpunkte. Allerdings schneiden sich in allen anderen Schnittpunkten weniger Kreise, als in M. In Abbildung 8.4 auf der nächsten Seite ist dieser Zusammenhang gut zu erkennen. Um einige Randpunkte des gegebenen roten Kreises wurden Kreise mit demselben Radius konstruiert. Diese schneiden sich alle in M.

Kennt man nun den Radius r des Kreises, der im Bild erkannt werden soll, so kann man auf einer gegebenen Menge von potentiellen Ball-Randpunkten die oben beschriebene Transformation ausführen. Zur Konstruktion der Kreise wurde der Bresenham-Algorithmus[21] verwendet. Dabei handelt es sich um ein Verrasterungs-Verfahren, das einen Achtel-Kreis liefert. Durch mehrmalige Spiegelung erhält man schließlich einen vollständigen Kreis. Der Vorteil dieses Verfahrens ist, dass es mit einfachen Vergleichsoperationen auskommt und ausschließlich auf ganzen Zahlen (Integer-Werten) arbeitet. Somit ist es ein äußerst schnelles Verfahren zur Kreisberechnung.

Es wird also nun um jeden der gegebenen Punkte ein Kreis mit dem Radius r konstruiert und anschließend derjenige Schnittpunkt heraus gesucht, in dem sich die meisten Kreise schneiden. Es ist sehr wahrscheinlich, dass dieser Punkt der Mittelpunkt des zu findenden Kreises ist. Allerdings muss dies erst durch Plausibilitätstests verifiziert werden. Ergeben diese Tests, dass es sich um den wahren Mittelpunkt handeln könnte, so kann man den zu findenden Kreis rekonstruieren, da ja der Radius bekannt ist. Problematisch ist, dass bei einem beliebigen Bild der Radius weder bekannt ist, noch bestimmt werden kann. Aus diesem Grund wurde ein Array mit elf verschiedenen Radien (zwischen 3 und 100 Pixeln) angelegt, und für jeden dieser Radien geprüft, ob es sich dabei ungefähr um

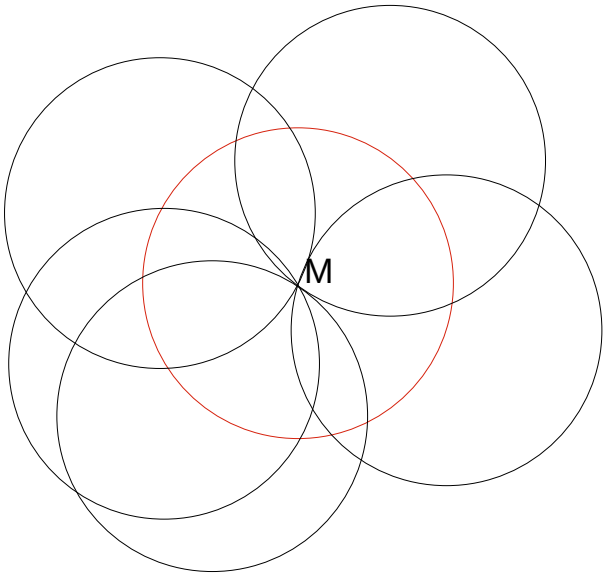

Abbildung 8.4: Werden um einige Randpunkte eines Kreises (rot) mit dem Radius r einige Kreise, die ebenfalls Radius r haben, konstruiert, so schneiden diese sich alle in dem Mittelpunkt M des ursprünglichen Kreises.

den Ballradius handeln könnte. Aus Laufzeitgründen können nicht alle möglichen Radien untersucht werden. Diese Laufzeitbeschleunigung führt allerdings auch wieder zu einer größeren Ungenauigkeit im Ergebnis. Werden nicht ausreichend viele Radien verwendet, wird der Ball zu ungenau erkannt. Es hat sich herausgestellt, dass bei Verwendung von elf Radien eine gute Erkennung in vertretbarer Zeit (ca. 100 ms pro Bild) erfolgt.

Da nur noch einige wenige Radien benutzt werden und die gefundenen Kantenpunkte nicht unbedingt exakt auf dem Rand des Balles liegen, wurde der Parameterbereich in Quadrate mit einer Seitenlänge von 5 Pixeln gequantelt. Diese Quadrate werden Akkumulatorzellen [20] genannt. Man geht davon aus, dass ein Schnittpunkt vorliegt, wenn mehrere Kreise durch dieselbe Akkumulatorzelle verlaufen. Abbildung 8.5 auf der nächsten Seite veranschaulicht diese Situation. Es gibt keinen gemeinsamen Schnittpunkt mehr. Dafür laufen alle Kreise durch die fünfte Akkumulatorzelle in der vierten Zeile.

Für jeden einzelnen Radius r_i wird also eine Transformation des Kantenbildes in den Parameterbereich durchgeführt und die Akkumulatorzelle bestimmt, durch die die meisten Kreise verlaufen. Anschließend wird um den Mittelpunkt dieser Zelle ein Kreis mit dem aktuellen Radius r_i konstruiert und die Farbe von 80 zufällig aus diesem Kreis ausgewählten Pixeln im segmentierten Bild ermittelt. Sind mehr als 10 Prozent der untersuchten Pixel schwarz und haben gleichzeitig weniger als 50 Prozent eine andere Farbe als schwarz oder weiß, so ist die Wahrscheinlichkeit sehr groß, dass der konstruierte Kreis den Ball im originalen Bild überdeckt. In diesem Fall handelt es sich um eine Lösung, die im folgenden zulässig genannt wird. Es kann vorkommen, dass der Kreis kleiner ist, als der Ball im Bild. Er liegt dann meistens vollständig innerhalb des Balles im Bild. Deswegen wird als Ergebnis der Ballerkennung der größte aller zulässigen Kreise genommen.

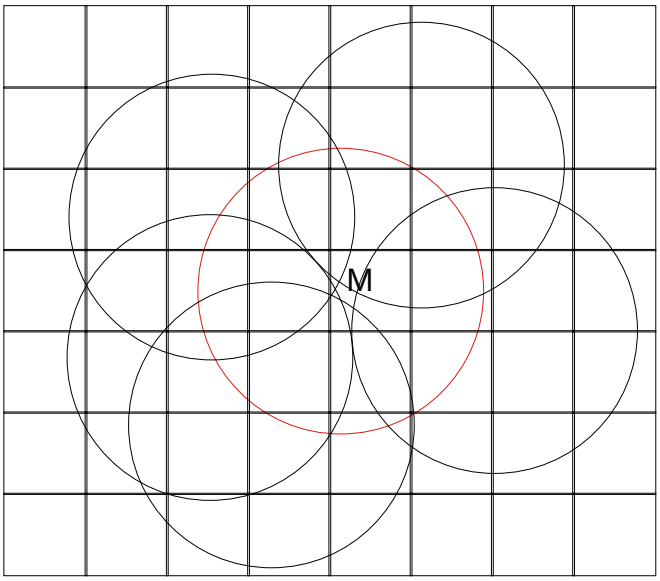

Abbildung 8.5: Werden um einige Punkte in der Nähe des Randes eines gegebenen Kreises (rot) mit dem Radius r einige Kreise gezogen, die ungefähr den Radius r haben, so schneiden diese sich nicht im gleichen Punkt. Somit verlaufen sie auch nicht alle durch den Mittelpunkt M des gegebenen Kreises. Sie verlaufen aber alle durch dieselbe Akkumulatorzelle.

Andererseits kann es auch passieren, dass der Ball zu groß erkannt wird, wenn er vor einer weißen Bande liegt. Dieser Fall lässt sich allerdings nicht einfach eliminieren, da die Bande den gleichen Weißton hat wie der Ball und man nicht sagen kann, dass keine großen weißen Regionen innerhalb des Balles vorkommen dürfen, da ein Ball, der sehr nah vor dem Roboter liegt, großflächige weiße Stellen haben kann. Das kann vor allem dann auftreten, wenn nur ein Teil des Balles sichtbar ist.

Um Laufzeit einzusparen, wird schon beim Konstruieren der Kreise im Parameterraum für den aktuellen Radius und jede durchlaufene Zelle automatisch ein Zähler erhöht, der angibt, wie viele Kreise bisher für diesem Radius durch die Zelle laufen. Außerdem wird das bisherige Maximum aller dieser Werte für einen festen Radius sowie die Koordinaten der Zelle, in der dieses Maximum liegt, gespeichert. Gibt es einen Maximalwert, der in mehreren Zellen vorkommt, so wird ausschließlich die zuletzt gefundene Zelle genommen. Bei allen durchgeführten Testszenarien ergab sich dadurch keine Verschlechterung des Ergebnisses. Auf diese Weise wird noch mehr Laufzeit eingespart.

In manchen Situationen befindet sich der Ball nur zum Teil im sichtbaren Bereich des Bildes. Ist das der Fall, so kann es vorkommen, dass auch der Mittelpunkt des Balles nicht sichtbar ist. Trotzdem muss er korrekt bestimmt werden, um ein geeignetes Percept berechnen zu können. Deswegen muss sowohl der Bildbereich als auch der Parameterbereich an den Rändern erweitert werden. So kann man gewährleisten, dass auch in diesem Sonderfall der Ball korrekt erkannt wird.

Wenn der Ball ganz nah vor dem Roboter liegt, sind zumindest die unteren Zeilen des Bildes fast ausschließlich schwarz oder weiß. Wenn am oberen Bildrand relativ viele grüne

Pixel sind, so liegt der Ball direkt vor dem Roboter, da oberhalb einer Bande kein Grün mehr vorkommt. Allerdings kann es auch sein, dass der Ball das ganze Bild ausfüllt. In diesem Fall gibt es im kompletten Bild kaum eine andere Farbe als schwarz oder weiß. Allerdings kann man diesen Fall nicht von der Situation unterscheiden, in der ein Roboter direkt auf die Bande schaut und nur diese sieht. Deswegen muss durch andere Methoden festgestellt werden, ob es sich um den Ball handelt oder nicht. Eine mögliche Lösung ist, den Roboter versuchen zu lassen, den Ball mit den Vorderbeinen zu greifen. Handelt es sich wirklich um den Ball, so bilden die Vorderbeine einen anderen Gelenkwinkel, als wenn er ins Leere greift.

8.2.4.1 Vorteile der Hough-Transformation bei der Ballerkennung

Die Hough-Transformation hat sich als gutes und robustes Verfahren zur Erkennung eines schwarz-weißen Balles erwiesen. Sie zeichnet sich durch eine Reihe von Vorteilen aus, die im Folgenden aufgezählt werden:

1. Nahe Bälle werden sehr gut erkannt

   Liegt ein Ball nicht allzu weit vom Roboter entfernt, so funktioniert die Erkennung sehr gut. An Testbeispielen konnte man ermitteln, dass eine zuverlässige Erkennung bis zu einer Distanz, die der halben Spielfeldlänge entspricht, möglich ist. Ein Beispiel dafür ist Abbildung 8.6.

Abbildung 8.6: Dieses Bild wurde mit der Kamera eines der Roboter aufgenommen. Es zeigt eine für die Ball-Challenge typische Szene. Der in der Mitte des Bildes befindliche schwarz-weiße Ball wurde mit der Hough-Transformation sehr gut erkannt. Das Ergebnis der Ballerkennung ist als roter Kreis visualisiert.

2. Erkennung von weit entfernten Bällen

   Es ist äußerst schwer, einen schwarz-weißen Ball zu erkennen, wenn er weiter als zwei Meter entfernt ist. Mit dem Verfahren der Hough-Transformation ist allerdings eine gelegentliche Erkennung möglich, wie Bild 8.7 zeigt.

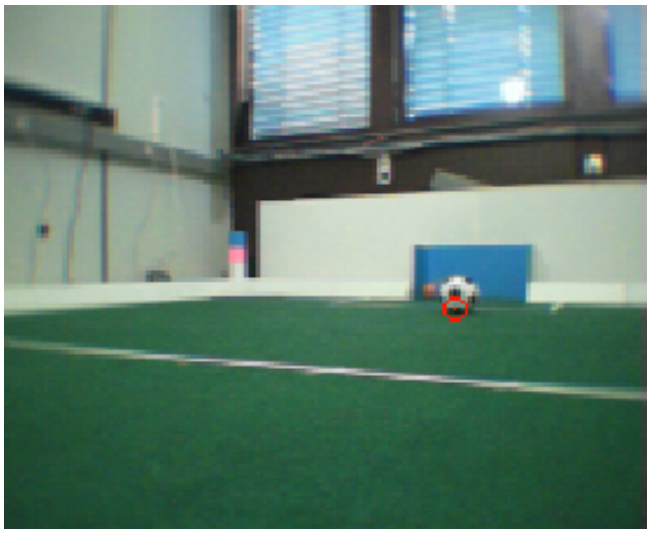

Abbildung 8.7: Vor dem blauen Tor befindet sich der zu findende schwarz-weiße Ball. Seine Entfernung zum Roboter, der dieses Bild aufgenommen hat, beträgt ungefähr 75% der Spielfeldlänge. Trotzdem wird der Ball noch erkannt (roter Kreis).

3. Erkennung von Bällen, die nur partiell sichtbar sind

   Es ist wichtig, auch solche Bälle gut zu erkennen, von denen nur ein Teil sichtbar ist. Eine solche Situation kann dann vorkommen, wenn ein Teil des Balles außerhalb des Bildes liegt. Es ist wesentlich schwieriger, einen solchen Ball relativ genau zu erkennen. Durch die Erweiterung des Bild- und Parameterbereichs um einen Rand ist die Hough-Transformation dazu in der Lage. Da aber dabei wichtige Informationen fehlen, die aus Bildern zu entnehmen sind, in denen der Ball vollständig sichtbar ist, ist das Ergebnis der Ballerkennung in solchen Extremfällen oft ein wenig schlechter. Dennoch kann daraus ein einigermaßen gutes BallPercept generiert werden. Abbildung 8.8 auf der nächsten Seite verdeutlicht diese Situation. Bei diesem Bild ist vom Ball nur ein kleiner Ausschnitt am rechten Bildrand zu sehen. Trotzdem wird seine Größe und die ungefähre Lage sehr gut erkannt.

4. Richtung wird gut erkannt

   Die Richtung, in der der Ball liegt, wird in 80% der Fälle nahezu optimal bestimmt. Dieser Wert wurde anhand eines Tests auf zahlreichen unterschiedlichen Bildern von Spielsituationen ermittelt. Das ist selbst dann der Fall, wenn der Ball nicht ganz korrekt erkannt wird. Diese Eigenschaft hat zwei positive Auswirkungen. Zum einen läuft der Roboter auch in solchen Situationen auf den Ball zu und somit in die richtige Richtung. Außerdem ändert er seine Laufrichtung nicht, wenn er denn Ball bereits vorher gut erkannt hat und in einzelnen folgenden Bildern der Ball falsch

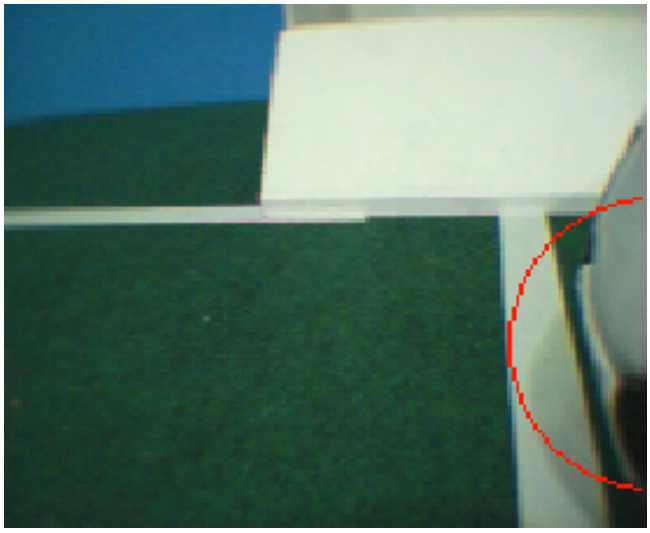

Abbildung 8.8: Am rechten Bildrand ist ein kleiner Bereich eines sehr nahe am Roboter liegenden schwarz-weißen Balles zu sehen. Auch solche Bälle, von denen nur ein Teil zu sehen ist, werden erkannt.

erkannt wird, sofern der Mittelpunkt des erkannten Balles innerhalb des Balles im Bild liegt.

5. Keine Phantom-Bälle

   Die Hough-Transformation ist überaus robust, d.h. es ist sehr unwahrscheinlich, dass ein Ball an einer falschen Stelle im Bild erkannt wird. Im aktuellen Projekt wurde diese Wahrscheinlichkeit mit Hilfe von Plausibilitätstests noch weiter gesenkt. Außerdem konnte dadurch fast vollkommen ausgeschlossen werden, dass ein Ball erkannt wird, obwohl keiner im Bild zu sehen ist. Das ist ein riesiger Vorteil, da auf diese Weise nur äußerst selten ein total fehlerhaftes BallPercept generiert wird. Dieser Sachverhalt wurde anhand zahlreicher aufgenommener Logfiles verifiziert. Dabei besteht ein Logfile aus einer Sequenz von Bildern (bis zu mehreren hundert Bildern pro File), die mit der Kamera des Roboters aufgenommen worden sind. Auf diesen Bildern konnte der Algorithmus angewendet und mittels so genannter Debug-Drawings das Ergebnis angezeigt werden.

   Durch die Plausibilitätstests haben auch einige potentielle Störfaktoren keinen Einfluss auf das Ergebnis. So wird zum Beispiel ein orangener Ball nicht als Ball erkannt, obwohl auch er rund ist. Das Bild 8.6 auf Seite 115 dient auch als Beispiel in diesem Fall, da der am rechten Bildrand liegende orangene Ball nicht als Ball erkannt wird und auch keine Auswirkung auf das Ergebnis hat.

8.2.4.2 Optimierungspotential

Obwohl die Hough-Transformation bei der Erkennung eines schwarz-weißen Balles sehr gute Ergebnisse liefert, gibt es dennoch Optimierungspotential. Folgende Punkte fielen auf:

1. Verwendung von festen Radien

   Um Laufzeit einzusparen, musste (wie oben beschrieben) die Anzahl der zu testenden Radien beschränkt werden. Dadurch ist es im allgemeinen nur möglich, den Radius des Balles möglichst gut zu approximieren. Eine hundertprozentige Übereinstimmung ist nur in den seltensten Fällen zu erzielen. Abbildung 8.9 veranschaulicht diese Situation. Der erkannte Radius ist zwar ungefähr so groß wie der tatsächliche, aber dennoch ein wenig zu klein.

Abbildung 8.9: Der Radius des in der Mitte des Bildes befindlichen schwarz-weißen Balles wird zu klein erkannt.

2. Quantisierung des Parameterbereichs

   Durch die Einteilung des Parameterbereiches in Akkumulatorzellen kann es vorkommen, dass der Ball nicht ganz korrekt erkannt wird, da als Lösung für den Mittelpunkt des Balles der Mittelpunkt einer Akkumulatorzelle genommen wird. Somit kann die Position des erkannten Balles leicht von der des zu erkennenden Balles abweichen (vergl. Abbildung 8.10 auf der nächsten Seite).

3. Ball liegt direkt vor einer Bande

   Wenn ein Ball sich direkt vor einer weißen Bande befindet, ist keine eindeutige Abgrenzung zwischen Ball und Bande möglich. Dadurch kann es vorkommen, dass der Ball zu groß erkannt wird. Diese Situation ist im Bild 8.11 auf der nächsten Seite gut zu erkennen.

   Ein ähnliches Problem kann sich dann ergeben, wenn der Ball direkt an einer Spielfeldlinie liegt.

Abbildung 8.10: Durch die Quantisierung des Parameterbereichs ist die erkannte Form (roter Kreis) gelegentlich ein wenig verschoben. Hier wird der Radius des Balles zwar gut erkannt, aber die Lage des berechneten Mittelpunktes ist gegenüber dem Mittelpunkt des Balles um einige Pixel nach links-unten verschoben.

Abbildung 8.11: Der Ball liegt direkt vor der weißen Bande. Im Hintergrund ist eine Landmarke zu erkennen. Da keine Abgrenzung zwischen den Weißtönen der Bande und des Balles möglich ist, wird der Ball hier zu groß erkannt.

8.2.5 Ballerkennung mit Hilfe eines modifizierten BallSpecialists

Im ersten Anlauf war der BallSpecialist so zu modifizieren, dass er aufgerufen wird, wenn der GridImageProcessor schwarze Pixel findet. Die Definition für einen Farbsprung wurde so abgeändert, dass er auf Übergange der Farbe Weiß zu Grün bzw. Schwarz zu Grün reagiert.

Im Simulator erwies sich dieser Algorithmus als durchaus praktikabel – in der Praxis stellte sich jedoch heraus, dass Teile der weißen Banden häufig als Ball erkannt wurden und die Farbsprungsuche vor allen Dingen dann versagte, wenn der Ball in der Nähe einer Bande lag.

Da der SoccerBallSpecialist kurz vor dem RoboCup 2003 immer noch nicht korrekt funktionierte, wurde ein Backup Algorithmus implementiert, der in zwei unterschiedlichen Fällen eine schnelle Ballerkennung garantieren soll:

- Der Ball liegt sehr nah vor dem Roboter.

  Der Ball wird mit Hilfe von Farbhistogrammen gefunden, die sich aus den fünf oberen und den fünf unteren Linien des Bildes ergeben. Sind im Histogramm der oberen fünf Linien genügend Pixel grün und in den unteren fünf Linien genügend Pixel weiß, wird die Ballposition direkt unter dem Kopf des Roboters angenommen.

- Der Ball liegt weiter entfernt vom Roboter.

  Dazu werden alle Pixel unterhalb des Horizonts untersucht. Wird ein schwarzer Pixel gefunden wird, beginnt der Algorithmus, eine Bounding Box aufzuspannen. Aus der Position und der Größe der Bounding Box wird dann der Ballkreis berechnet.

8.2.6 Fazit

Die Tests für diesen Algorithmus ergaben, dass die Richtung des Balls eigentlich ganz gut erkannt wurde. Allerdings verschätzte sich der Algorithmus oft bei der Größe des Balls. Auch wurden teilweise nicht vorhandene Bälle (z. B. durch Schatten auf der Bande) erkannt. Dagegen überzeugte dieser Algorithmus häufig im ersten Ballkontakt, wobei die Position jedoch nicht genau genug war, um ein korrektes Dribbeln oder Schießen zu realisieren.

8.3 Implementierung

Mit allen beschriebenen Verfahren zur Ball-Erkennung konnte die Position und Größe des Balls gut bestimmt werden. Allerdings werden Bälle, die relativ weit vom Roboter entfernt liegen, nur äußerst selten erkannt. Da das Verfahren, das die Hough-Transformation verwendet, Bälle bis zu einer Entfernung einer halben Spielfeldlänge noch gut erkennt

und außerdem nur mit verschwindend geringer Wahrscheinlichkeit ein BallPercept an einer Stelle liefert, wo kein Ball liegt (z.B. in einer Bande), wurde dieses Verfahren zur Erkennung des Balls verwendet.

8.4 Verhalten

Für das Verhalten bei der Ball Challenge ergaben sich einige Probleme, die beim normalen Spiel nicht vorkommen. Zum einen wurde der Ball nur über die halbe Spielfeldlänge zuverlässig erkannt. Das zweite große Problem betraf das Ball-Handling. Der schwarzweiße Ball hatte wesentlich mehr Oberflächenreibung als der orangene Ball, was zur Folge hatte, dass der Ball nicht mit den normalen Kicks gespielt werden konnte und neue Kicks erarbeitet werden mussten.

Das grobe Verhalten für die Challenge sah wie folgt aus:

1. Ballsuche

   Als Ballsuche wurde ein Suchlauf programmiert, der eine Menge von festgelegten Punkten auf dem Spielfeld angelaufen ist. Dies hat sichergestellt, dass jeder Punkt des Spielfeldes während eines Suchlaufs aus einer Entfernung von weniger als 80 cm betrachtet wurde. Ein Suchlauf über das gesamte Feld dauerte etwa 60 Sekunden.

2. In Position bewegen

   Nachdem der Ball lokalisiert wurde – mit einer guten Richtungsangabe und einer schlechten Entfernungsangabe – hat sich der Roboter in Position bewegt. Seine Vorgabe dabei war, sich auf einer Gerade a, die die Mitte des Tores und den Ball schneidet, sich 30 cm hinter dem Ball zu positionieren. Dieser Zustand wurde abgebrochen, sobald der Winkel zwischen einer Gerade

   $$\bar{a} := \overline{Tor, Ball}$$

   und einer zweiten Gerade

   $$\bar{b} := \overline{Ball, Hund}$$

   kleiner als 5° wurde.

   $$\text{Deviation } \delta = |\angle(\bar{a}, \bar{b})|$$

3. Den Ball ins Tor schieben

   Für die Lösung des Ball-Handling-Problems haben sich drei Methoden angeboten. Zum einen ein konventionelles Vorgehen mit Kicken in Torrichtung. Dieses Vorgehen hat sich aber bald als nicht praktikabel erwiesen, da keine sinnvollen Kicks (außer nach hinten) gefunden werden konnten. Die zweite Methode war ein „Tragen in das Tor“. Der Roboter sollte bis zum Ball laufen, ihn dann greifen und ins Tor tragen. Diese Möglichkeit wurde aber in Padua verworfen, da die Oberflächenbeschaffenheit

des Spielfeldbodens einerseits ein Vorwärtskrabbeln mit den Hinterbeinen stark behinderten und zum anderen der Ball zu oft verloren ging und aus den Vorderbeinen herausrutschte.

Als dritte und letztendlich erfolgreiche Methode stellte sich das Bulldozer-Prinzip heraus. Sobald die Bedingung

$$\delta \leq 5^\circ$$

erfüllt war, begann der Hund, sich mit ausgestreckten Vorderbeinen in Richtung Tor zu bewegen. Dieses Verhalten wurde so lange beibehalten, bis entweder der Hysteresebereich

$$\delta \leq 15^\circ$$

verlassen wurde und der erkannte Ball mehr als 20 cm entfernt war (oder gar nicht mehr gesehen wurde). In diesem Fall wurde wieder mit Punkt 1 bzw. Punkt 2 weitergemacht.

8.5 Ein HeadControl für die Ball-Challenge

Da zu Beginn die Bilderkennung für die Ball-Challenge eine hohe Laufzeit hatte, kam es dazu, dass nicht alle Frames, die die Kamera lieferte, verarbeitet werden konnten. Daher kam es zu dem Problem, dass es vorkam, dass der Ball übersehen wurde.

Daher wurde versucht, einen langsameren HeadControlMode einzubauen, um dieses Problem zu lösen. Die Voraussetzungen für den HeadControlMode waren folgende:

- Der Kopf sollte sich langsam bewegen.

  Damit sollte erreicht werden, dass während mehrerer aufeinander folgender Frames der Ball gesehen wird.

- Waagerechte Kopfhaltung

  Der momentane HeadControlMode „SearchForBall“ verdreht den Kopf, um so einen größeren Blickbereich zu erreichen. Für die Kantenfilter ist es aber besser, wenn der Kopf parallel zum Grund bewegt wird.

- Spezielles Bewegungsmuster des Kopfes

  Es soll langsam nach dem Ball gesucht werden und schnell nach Landmarken. Der Geschwindigkeitsunterschied ist darauf zurückzuführen, dass die Landmarken leichter zu erkennen sind.

Nach diesen Überlegungen wurde der GT2003HeadControl geklont und für die oben erwähnten Zwecke umgeschrieben. Damit sich der Kopf langsam bewegt mussten alle Aufrufe, welche die Bewegung des Kopfes steuern, verlangsamt werden. Die Dauer einer Kopfbewegung wird in der neuen neuen HeadControl-Solution genauso wie im GT2003HeadControl beim Aufruf von headPathPlanner.init() angegeben. Dafür ist der dritte Parameter von Nöten.

Hierbei wird eine Zeitspanne in ms angegeben, in der der Kopf die angegebenen Punkte passieren soll. Als Lösung für das erste Problem wurde ein globaler Parameter deklariert und in allen Aufrufen von headPathPlanner.init() die Zeitspanne durch das Produkt mit diesem Parameter ersetzt. Dadurch wurde es möglich durch Anpassen dieses Parameters Einfluss auf die Kopfgeschwindigkeit zu nehmen. Als optimal hat sich ein Wert zwischen 3 und 5 erwiesen.

Um die beiden weiteren Anforderungen zu lösen wurden zwei neue HeadControlModi entwickelt:

- SpecialSearchForBall

  sucht in einer rechteckigen Bewegung nach dem Ball und nach den Landmarken. Dabei wird der Kopf von der linken unteren Ecke über die rechte untere, die rechte obere und die linke obere Ecke bewegt. Die komplette Bewegung erfolgt langsam. (siehe Abbildung 8.12)

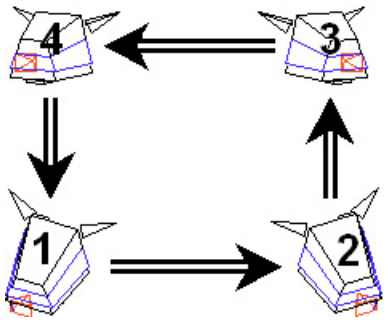

Abbildung 8.12: HeadControl-Bewegung bei SpecialSearchForBall

- SearchForLandmarksWithBall

  sorgt dafür, dass ein Roboter der im Ballbesitz ist, sich mit Hilfe der Landmarken orientieren kann, ohne unbemerkt den Ball zu verlieren. Mit seinem Maul ist er in der Lage, einen möglichen Ballverlust zu erkennen. Der AIBO schaut von links nach rechts und zurück nach Landmarken. Nach jedem Durchgang lässt der AIBO den Kopf leicht nach unten fallen um zu überprüfen, ob er den Ball noch hat. Der AIBO braucht den Ball nicht im Bild zu suchen, da er davon ausgeht, das er ihn noch festhält. Deshalb kann die Kopfbewegung mit normaler Geschwindigkeit erfolgen. (siehe Abbildung 8.13)

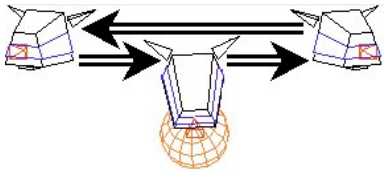

Abbildung 8.13: HeadControl-Bewegung bei SearchForLandmarksWithBall

In den Tests in Padua haben sich diese HeadControlModi als brauchbar erwiesen. In der eigentlichen Ball-Challenge wurden sie aber nicht benutzt, da die Bildverarbeitung bis

dahin stark verbessert wurde und somit kein Bedarf mehr für spezielle HeadControlModi mehr bestand. Die Ball-Challenge wurde mit dem GT2003HeadControl und dem HeadControlMode SearchAuto gewonnen.

8.6 Fazit

Der in diesem Kapitel beschriebene Ansatz hat sich als hervorragend erwiesen. Das German Team ist als Sieger aus Ball-Challenge hervorgegangen und hat dadurch auch die Gesamtwertung der Challenges gewonnen. Nicht nur die Erkennung des Balles funktionierte äußerst gut, auch das Verhalten war sehr robust und gut auf die Erkennung abgestimmt. Da der Roboter den Ball nicht über die komplette Spielfeldlänge erkennen kann, wurde ein Laufmuster entwickelt, das diese Tatsache berücksichtigt.

Kapitel 9

Analyse von Bewegungsmustern zur Ballfortbewegung

Dieses Kapitel beschreibt die Untersuchung einer Auswahl von Kicks, stellt den Testbogen und die Analysekriterien vor und gibt einen Ausblick über die Ergebnisse.

9.1 Einleitung

Im Laufe der Jahre hat sich eine relativ große Auswahl an Bewegungsmustern zum Schießen des Balls angehäuft. Ein „Muster“ ist dabei der Inhalt einer so genannten „MOF-Datei“, die eine Folge von Motorpositionen angibt, die dann nacheinander ausgeführt eine Bewegung ergeben. Es sollten nun diese Muster analytisch untersucht und nach Brauchbarkeit beurteilt werden. Das Ergebnis sollte dann per Auswahlfunktion in das Spielverhalten mit einfließen und es optimieren.

9.2 Beschreibung des Testablaufes

Zur Evaluierung Bewegungsmuster zum Schießen wurde ein Testbogen (siehe Kapitel 9.3 auf Seite 127) erstellt. Damit dieser Testbogen ausgefüllt werden konnte, wurde auch ein Testverhalten des AIBOs benötigt. Dazu wurde das Schussverhalten „Go-To-Ball-And-Kick.xml“ als Basis verwandt. Die dortige Auswahlfunktion wurde herausgenommen und durch ein simples Verhalten ersetzt, bei dem der Roboter sich nach dem Ball umschaut, zu ihm hingeht, ihn mit dem gewünschten Bewegungsmuster schießt und danach stehen bleibt. Nachdem der Roboter wieder auf Startposition gesetzt wurde, konnte das Schussverhalten mit einem Druck auf dem Rückenknopf erneut gestartet werden. Die abgeänderte XML-Datei wurde durch weitere Anpassungen, bzw. „Registrierungen“ in der „options.xml“, der „agents.xml“ und der „BehaviorControlSelector.cpp“ in soweit erweitert, dass ein Memorystick angefertigt werden konnte. Mit diesem konnte der AIBO obiges Verhalten ausführen. Die Startposition des Roboters war in der Mitte der Torgrundlinie und die des Balles in der Mitte der dem gelben Tor abgewandten Strafraumlinie (siehe

Abbildung 9.1).

Der umgebende Bereich des Spielfeldbereiches des gelben Tores wurde mit Tesafilmmarkierungen in einem 10°-Abstand markiert (siehe Abbildung 9.1). Mit dieser Grundstellung wurden alle Bewegungsmuster zum Schießen des Balls dann y-mal wiederholt (y in der Regel 20 - 30). Bei manchen Mustern, wie bspw. dem „Bicycle.Kick“, musste der Roboter gespiegelt zum Ball positioniert werden, weil der Ball in Heckrichtung bewegt wurde. Als Farbtabelle wurde eine primitive C64-Colortable genommen, die nur den grünen Rasen und den orangen Ball enthielt. Damit war auch ein Testen ohne viel Licht möglich.

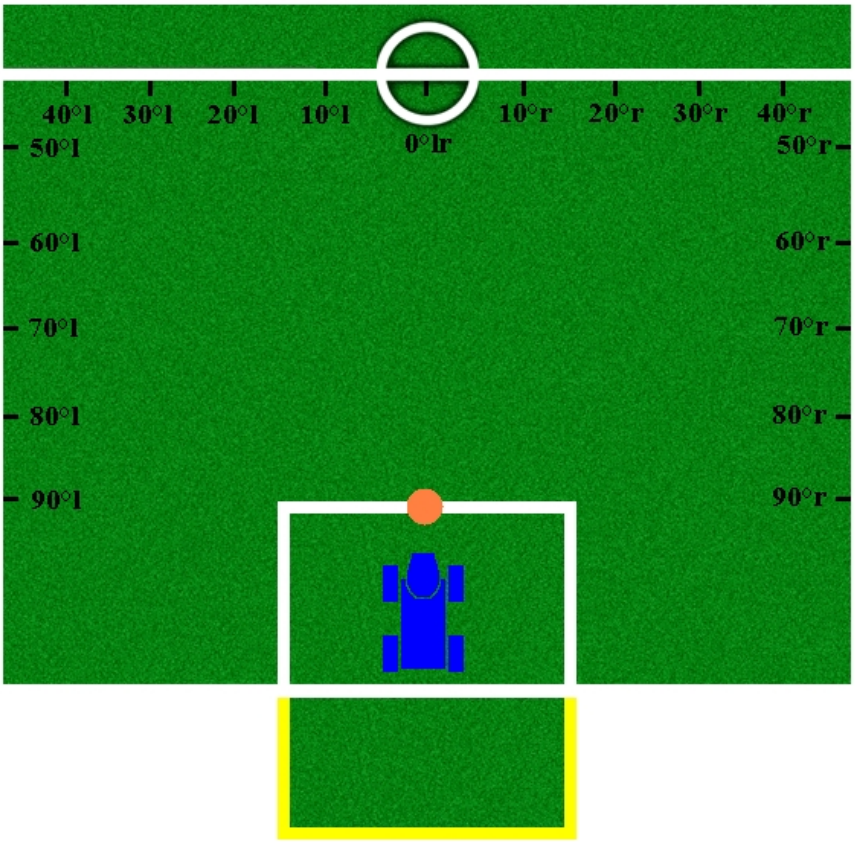

Abbildung 9.1: Halbes Spielfeld mit Tesafilmmarkierungen

Aufgrund der tabellarischen Aufbereitung der Beobachtungen konnte eine Auswahlfunktion aufgestellt werden. Insgesamt wurden auf diese Weise 31 Bewegungsmuster untersucht (siehe Tabelle 9.1 auf der nächsten Seite).

Nr.	Val.	Kickname	Nr.	Val.	Kickname
1	1	chest-kick-1	17	20	kick-with-left-to-right
2	2	chest-kick-2	18	24	heel-kick-right
3	3	left-kick	19	25	heel-kick-left
4	4	right-kick	20	33	kick-from-hold
5	5	left-head-kick	21	34	left-kick-from-hold
6	6	right-head-kick	22	35	right-kick-from-hold
7	7	leg-kick	23	36	chest-kick-3
8	8	mantis-kick	24	55	unsw-chest-kick
9	9	arm-kick-left	25	56	bicycle-kick
10	10	arm-kick-right	26	57	bicycle-kick-left
11	14	unsw-bash	27	58	bicycle-kick-right
12	15	head-kick	28	60	unsw-bash-from-hold
13	16	left-paw-kick-forward	29	62	left-head-kick2
14	17	right-paw-kick-forward	30	63	right-head-kick2
15	18	unsw-kick-forward-optimized	31	73	bicycle-kick-from-hold
16	19	kick-with-right-to-left			

Tabelle 9.1: Getestete Bewegungsmuster

9.3 Bewertungsbogen

Kick: „Name“
Option: test-„Name“.xml
Optionvalue: Index des Kicks „Name“ in der options.xml
Vorbedingungen: Beispielsweise, ob der Roboter umgedreht werden musste.
Benötigte Zeit: Ungefähre Zeit (geschätzt) in ganzzahligen Sekunden, kann auch in RobotControl gemessen werden.

Beschreibung:
Kurze Darstellung des Bewegungsmusters.

Testergebnis (nach y Schüssen)

(Richtung vorne=0°,hinten=180°, links=90°l, rechts=90°r):
Notierung der genommenen Winkelbereiche.

Fall i:
Richtung relativ zum Roboter: Generelle Schussrichtung in Grad und mit Seitenkürzel (a°l).
Streuwinkel: Ungefährer Winkelbereich, in dem der Ball geschossen wurde (ca. b° l/r). Dabei war die generelle, verlängerte Richtung der Ballbewegung ausschlaggebend. Der oft krumme Verlauf am Ende der Bewegung aufgrund der Unausgewogenheit des Balles wurde nach Augenmaß geglättet.

Erreichte Distanz: Ungefähre Schussweite in cm.
Geschwindigkeit: (langsam bis mittel, mittel bis schnell, langsam, mittel, schnell)
Wahrscheinlichkeit: Wahrscheinlichkeitsverteilungswert für die Auswahlfunktion (xi/y).

Bemerkung: Kurze Beschreibung, was falsch oder genau richtig lief.

Kommentar:
Eigene Meinung, was am Schuss gut oder schlecht ist.

Verbesserungspotential:
Eigene Meinung zu Verbesserungsmöglichkeiten.

9.4 Beispiel

Als ausführliches Beispiel folgt nun der ausgefüllte Testbogen zum Bewegungsmuster „Leg-Kick“.

Kick: Leg-Kick
Option: test-legKick.xml
Optionvalue: 7
Vorbedingungen: keine
Benötigte Zeit: ca. 1 s

Beschreibung:
Beim Leg-Kick begibt der Hund sich durch Spreizen der Hinterbeine und Nach-Vorne-Schieben der Vorderläufe in eine Erwartungsposition mit Schräglage nach vorne. Dann werden beide Vorderläufe nach oben genommen und schnell etwas nach innen geschoben nach unten geschlagen.

Testergebnis (nach 30 Schüssen)

(Richtung vorne=0°,hinten=180°, links=90°l, rechts=90°r):

Fall 1:
Richtung relativ zum Roboter: 0°
Streuwinkel: ca. 10° l/r
Erreichte Distanz: 100 - 200 cm
Geschwindigkeit: mittel bis schnell
Wahrscheinlichkeit: 5/30
Bemerkung: Optimaler Kick.

Fall 2:
Richtung relativ zum Roboter: 0°
Streuwinkel: ca. 10° l/r
Erreichte Distanz: 30 - 100 cm
Geschwindigkeit: langsam bis mittel
Wahrscheinlichkeit: 6/30
Bemerkung: Schwach erwischter, optimaler Kick.
Fall 3:
Richtung relativ zum Roboter: 0°
Streuwinkel: ca. 10° l/r
Erreichte Distanz: 0 - 20 cm
Geschwindigkeit: langsam
Wahrscheinlichkeit: 5/30
Bemerkung: Kaum erwischter, optimaler Kick.
Fall 4:
Richtung relativ zum Roboter: 50° l/r
Streuwinkel: ca. 10° l/r
Erreichte Distanz: 30 - 100 cm
Geschwindigkeit: langsam bis mittel
Wahrscheinlichkeit: 8/30
Bemerkung: Einbeinig erwischter, optimaler Kick.
Fall 5:
Richtung relativ zum Roboter: -
Streuwinkel: -
Erreichte Distanz: -
Geschwindigkeit: -
Wahrscheinlichkeit: 6/30
Bemerkung: Ball wurde durch falsches Herangehen weggekickt.

Kommentar:
Der Leg-Kick ist leider nur relativ sicher.

Verbesserungspotential:
Das Klammern kann hier Wunder bewirken!

9.5 Zusammenfassung der Untersuchungsergebnisse

In der folgende Tabelle (siehe Tabelle 9.3 auf der nächsten Seite) befindet sich nun eine Übersicht über das gesamte Testergebnis. Die tabellarische Form (Trefferwinkel: im Uhrzeigersinn) wurde dabei weiterbenutzt für die oben erwähnten Auswahlfunktion. Von den vormals 31 Mustern gelangten dann nur 16 in die Tabelle, da diverse Bewegungsmuster nicht ausführbar, doppelt oder als unbrauchbar deklariert worden waren.

Kickname	*Dauer*	*Entfernung*	*Nichts passiert*	*Trefferwinkel*		*Wahrsch.*	*Distanz*
				von	bis		
Mantis-Kick	3s	70	0,2	340	20	0,64	> 100
				30	90	0,08	$0 - 100$
				270	330	0,08	$0 - 100$
Left-Kick	1,5s	70	0,3667	290	310	0,4	$70 - 200$
				330	340	0,2	$0 - 50$
				50	50	0,1	$0 - 100$
Right-Kick	1,5s	70	0,5	50	70	0,3	$70 - 200$
				70	90	0,1333	$0 - 100$
				10	30	0,0667	$0 - 50$
Kick-with-Left- -to-Right	1s	90	0,35	50	70	0,5	$0 - 40$
				80	100	0,1	$10 - 30$
				60	80	0,05	$80 - 110$
Kick-with-Left- -to-Right(110)	1s	110	0,2	50	70	0,4	$10 - 50$
				80	100	0,15	60
				60	60	0,25	$0 - 10$
Kick-with-Right- -to-Left	1s	90	0,32	290	310	0,44	$80 - 120$
				260	280	0,04	$30 - 50$
				290	310	0,16	$10 - 40$
				0	0	0,04	$0 - 10$
Heel-Kick-Left	1s	90	0,3667	200	240	0,3667	$80 - 150$
				330	30	0,1667	$0 - 30$
				230	230	0,0333	20
				260	280	0,0667	$50 - 100$
Heel-Kick-Right	1s	90	0,3	120	160	0,4	$80 - 150$
				330	30	0,2667	$0 - 30$
				140	140	0,0333	50
Leg-Kick	1s	90	0,2	350	10	0,1667	$100 - 200$
				350	10	0,2	$30 - 100$
				350	10	0,5	$0 - 20$
				40	60	0,1333	$30 - 100$
				300	320	0,1333	$30 - 100$
Left-Head-Kick2	2s	80	0,05	270	290	0,85	$30 - 70$
				310	310	0,05	20
				0	0	0,05	30
Right-Head-Kick2	2s	80	0	70	90	1	$30 - 70$
arm-kick-left	1,5s	110	0,1034	300	300	0,3793	$40 - 100$
				310	310	0,2069	$30 - 60$
				290	290	0,2759	$40 - 80$
				270	270	0,0345	$40 - 60$
arm-kick-right	1,5s	110	0,3939	60	60	0,3333	$40 - 100$
				50	50	0,0909	$30 - 40$
				70	70	0,1212	$80 - 100$
				90	90	0,0606	$40 - 60$
bicycle-kick	1,5s	90	0,2	180	180	0,68	$80 - 180$
				150	150	0,12	$80 - 100$
bicycle-kick-left	1,5s	90	0,08	175	185	0,4	$150 - 180$
				150	150	0,08	$150 - 180$
				200	210	0,2	$150 - 180$
				190	190	0,08	$150 - 180$
				170	170	0,16	$150 - 180$
unswBashOptimized	-	90	0,4	350	10	0,48	300
				340	20	0,12	30

Tabelle 9.3: Tabellarische Zusammenfassung der Testergebnisse

9.6 Ergebnis

Auch wenn schlussendlich die entwickelte Auswahlfunktion nicht verwendet wurde, so fanden die Ergebnisse doch in indirekter Form in das Spielverhalten. Aufgrund der Beurteilung der Bewegungsmuster wurde eine feste Auswahl an Kicks für entsprechende Spielsituationen getroffen, wie bspw. das Bandenverhalten. Dadurch war das German Team auf der Weltmeisterschaft in Padua, Italien 2003 eine der wenigen Mannschaften, mit einer Auswahl von mehr als 4 Kicks.

Des weiteren führten die Untersuchungen zu einer Gesamtbeurteilung, die das Fehlen oder die Überflüssigkeit einiger Kicks aufdeckte. So wurde bspw. die „Dive-Kick“-Reihe aufgestellt. Aufgrund spezieller Umstände (weicher, klebriger Ball) wurde auch für die Ballchallenge ein Greif- und Laufmuster als Kicks, bzw. gesonderte Schussmuster entwickelt.

Außerdem wurde ein Videoarchiv von Bewegungsmustern angelegt. Mit diesen Filmchen wurde nun eine Kickvorführung durch den AIBO selbst unnötig, wodurch man sich die Erstellung eines Memorysticks ersparte und die Gelenke der Roboterhundes schonte.

Kapitel 10

Teilnahme an Veranstaltungen

10.1 German Open

Die German Open ist die internationale deutsche Meisterschaft im Roboterfußball. In diesem Jahr fand sie vom 10. bis 13. April im Heinz-Nixdorf-Museums-Forum in Paderborn statt. Vertreten waren zahlreiche Mannschaften aus sechs verschiedenen Ligen:

- Middle-Size League
- Small-Size League
- Sony Legged League
- Rescue Simulation League
- Soccer Simulation League
- Junior League

In der Sony Legged League traten neben den Microsoft Hellhounds noch sieben weitere Teams an.

Bei dieser Meisterschaft wurden erstmals die Sensordaten der einzelnen Roboter fusioniert (vergleiche Kapitel 5 auf Seite 49 und Kapitel 6 auf Seite 57). Die Auswirkungen konnte man sehr gut erkennen: Alle Roboter wendeten sich zum Ball (selbst wenn sie selbst ihn nicht sehen konnten). Als problematisch stellte sich heraus, dass das verwendete Verhalten nicht hundertprozentig auf die Sensorfusion abgestimmt war. So liefen alle Roboter immer auf den Ball zu, wodurch sie sich häufig verhakten.

Ein weiteres Problem stellte das Licht dar. Das Spielfeld befand sich in der Nähe einer großen, nicht abgedunkelten Fensterscheibe. Da das einfallende Licht nicht konstant war, stimmten die vorher erstellten Farbtabellen häufig nicht mehr exakt. Auch das Licht der Scheinwerfer war nicht optimal; es war sehr rötlich, wodurch die Roboter oft orangene Stellen im gelben Tor gesehen und diese als Ball erkannt haben. Durch die Fusionierung

der Sensordaten ergaben sich daraus gelegentlich Situationen, in denen sich alle Roboter vom Ball abwendeten und in Richtung des gelben Tores ausrichteten.

Das Problem, dass orangene Flecken im gelben Tor erkannt werden, trat erstmals bei dieser Veranstaltung auf. Damit eine solche Fehlerkennung keine Auswirkung auf das Ergebnis der anderen Roboter hat, wurde dieses Problem während der German Open dadurch umgangen, dass die so genannte KnownPosition nur aus solchen Percepts gebildet wird, die nicht innerhalb des gelben Strafraumes bzw. Tores liegen.

Außerdem wurde festgestellt, dass jeder einzelne Roboter den Ball zu ungenau erkennt. Um die Position des Balles zu ermitteln, muss der Roboter zunächst seine eigene Position kennen sowie den Abstand und den Winkel zum Ball. Sowohl die Erkennung der eigenen Position, als auch die des Abstandes zum Ball variierte sehr stark. Dadurch „sprang“ das BallPercept sehr stark. Dieses Problem wurde dadurch gelöst, dass bei der Ballerkennung eine Plausibilitätsabfrage eingebaut wurde. Von 100 zufällig ausgewählten Pixeln aus dem Inneren des vermuteten Balles müssen mindestens 40% orange sein. Durch diesen Test wurden die Percepts genauer.

Von den insgesamt 6 Spielen der Microsoft Hellhounds wurde bei vier Siegen nur eins verloren. So erreichte das Team einen hervorragenden dritten Platz. In der folgenden Tabelle sind die Ergebnisse dargestellt.

Spiel	Mannschaft 1			Mannschaft 2
Vorrunde	Microsoft Hellhounds	0	0	Darmstadt Dribbling Dackels
Vorrunde	Microsoft Hellhounds	2	1	Bremen Byters
Vorrunde	Microsoft Hellhounds	2	0	Team Griffith
Viertelfinale	Microsoft Hellhounds	2	1	SPQR-Legged
Halbfinale	Microsoft Hellhounds	0	3	Aibo Team Humboldt
Spiel um Platz 3	Microsoft Hellhounds	2	0	Dynamo-Pavlov Uppsala

10.2 RoboCup 2003 Padua

Jedes Jahr wird eine Roboter-Fußball-Weltmeisterschaft veranstaltet, die RoboCup genannt wird. In diesem Jahr fand sie vom 02. bis 11. Juli in Padua, Italien statt. Am RoboCup 2003 haben zahlreiche Mannschaften teilgenommen, die in sieben verschiedenen Ligen gespielt haben. Zusätzlich zu den auch bei den German Open vertretenen Ligen (siehe Kapitel 10.1 auf der vorherigen Seite) traten hier auch Mannschaften in der Humanoid League an.

In der Sony Legged League waren insgesamt 24 Mannschaften vertreten. Das Dortmunder Team hat mit drei weiteren deutschen Universitäten (Humboldt Universität zu Berlin, Universität Bremen und Technische Universität Darmstadt) zusammen das German Team gebildet. Nach einer souverän überstandenen Vorrunde ist das German Team allerdings unglücklich im Achtelfinale ausgeschieden. Nach der regulären Spielzeit war der Spielstand unentschieden (2:2). Das anschließende Penalty-Schießen wurde dann aber unglücklich verloren. Die einzelnen Ergebnisse sind der folgenden Tabelle zu entnehmen:

Abbildung 10.1: Die Microsoft Hellhounds auf der German Open.

Spiel	Mannschaft 1			Mannschaft 2
Vorrunde	German Team	-	-	Essex (zurückgezogen)
Vorrunde	German Team	9	0	UT Austin
Vorrunde	German Team	2	2	UTS Unleashed
Vorrunde	German Team	4	0	Asura
Vorrunde	German Team	3	1	UPennalizers
Achtelfinale	German Team	17 (2)	18 (2)	CMPack

Neben den eigentlichen Spielen mussten in der Sony Legged League auch noch so genannte Challenges bestritten werden, die bereits einige Monate vor der Veranstaltung bekannt gegeben worden waren. In diesem Jahr gab es folgende Challenges:

1. Ball-Challenge:

   Ziel ist es, einen schwarz-weißen Ball auf einem leeren Spielfeld zu erkennen und innerhalb von drei Minuten in das gelbe Tor zu befördern (vergl. Kapitel 8 auf Seite 103).

2. Localization-Challenge:

   Bei diesem Wettbewerb muss sich ein Roboter auf dem Spielfeld ohne Landmarken lokalisieren können. Er hat die Aufgabe, fünf bestimmte Stellen anzulaufen und dort für eine kurze Zeit stehen zu bleiben. Die Bewertung verläuft reziprok zur Entfernung des Roboters von den einzelnen vorgegebenen Anlaufpunkten.

3. Obstacles-Avoidance-Challenge:

   Bei dieser Challenge soll ein Roboter von einem Tor ins gegenüber liegende laufen und dabei Hindernissen ausweichen. Als Hindernisse dienten sieben SONY-Laufroboter.

Bei der Ball-Challenge konnte die Lösung des German Teams überzeugen. Der Ball wurde sehr schnell erkannt und in Richtung des gelben Tores geschossen. Dabei ist der Roboter stets dem Ball gefolgt und hat ihn immer näher an das Tor heran gebracht. Kurz vor dem gelben Tor hat der Roboter den Ball aber aus dem Blickfeld verloren, wodurch kein Tor erzielt worden ist. Trotzdem stand das German Team als Gewinner fest, weil dessen Roboter den Ball schneller berührt hat als jeder andere und auch kein anderes Team ein Tor erzielen konnte.

Auch in der Obstacles-Avoidance-Challenge konnte ein Sieg errungen werden. Dadurch reichte dem German Team der dritte Platz in der Localization-Challenge, um als Challenge-Sieger ausgezeichnet zu werden.

10.3 Messen und Veranstaltungen

Einige Mitglieder der Projektgruppe sowie die studentischen Hilfskräfte vertraten die Microsoft Hellhounds auf verschiedenen Messen und Veranstaltungen.

10.3.1 CeBit Hannover

Auf der diesjährigen CeBit in Hannover stellten die Hellhounds auf dem Microsoft-Stand im Rahmen der „Microsoft Academic Alliance“ ihre Arbeit vor.

Dazu wurde von Matthias Hebbel und Arthur Cesarz ein Glas-Schaukasten entworfen und gebaut, in dem ein Roboter eigentlich Fußball spielen sollte. Da die Vorgaben bezüglich der Größe allerdings nicht zum Spiel mit dem Ball ausreichten, wurde der AIBO dazu programmiert, einige Aktionen wie Sitz, Winken usw. ohne weiteres Zutun auszuführen.

Einige Mitglieder der Projektgruppe halfen dann, diese Kiste inklusive der Hunde auf das Messegelände zu bringen und dort aufzubauen. Dazu musste natürlich auch eine Farbtabelle erstellt werden.

Abbildung 10.2: Schaukasten der Hellhounds auf dem Microsoft-Stand

Wegen der allgemeinen Hektik beim Aufbau auf dem Microsoft-Stand wurde die Glaskiste allerdings nicht an der richtigen Stelle aufgebaut. Da sie dann von Microsoft-Mitarbeiten an einer anderen Stelle neu aufgebaut wurde, musste am ersten Messetag noch einmal nach Hannover gefahren werden, um die Farbtabelle an die neuen Lichtverhältnisse anzupassen. Der endgültige Standort des Schaukastens ist in Abbildung 10.2 zu sehen.

Problematisch bei diesem Auftritt waren vor allem die Lichtverhältnisse am zweiten Aufbauort sowie die enorme Anzahl an WLAN-Netzen, die dazu führte, dass es am zweiten Tag nicht vor 18.00 Uhr möglich war, eine Verbindung zum Roboter zur Erstellung der Farbtabelle aufzubauen.

Am dritten Tag wurde von Ingo Dahm auf dem Microsoft-Stand noch ein Vortrag gehalten, in dem er die Arbeit der Hellhounds einem breiten Publikum vorstellte.

Unsere Präsens auf dem Microsoft-Stand wurde von der Menge begeistert aufgenommen. Der AIBO als Publikumsmagnet machte die Leute auf die Microsoft-Academic-Alliance aufmerksam, so dass auch Microsoft von dieser Kooperation profitierte.

10.3.2 Control Sinsheim

Auf der Control in Sinsheim, einer internationalen Fachmesse für Qualitätssicherung, stellten die Microsoft Hellhounds ihre Arbeit am „Vision"-Stand der Fraunhofer-Gesellschaft vor. Auf diesem Stand wurden verschiedene Entwicklungen rund um die Bildverarbeitung gezeigt, wobei auch Teams aus verschiedenen RoboCup Ligen ihre Entwicklungen präsentierten.

Dazu wurde an allen 4 Messetagen wiederum die Glaskiste, die für die CeBit gebaut worden war, eingesetzt. Auf dieser Messe war allerdings immer jemand dabei, der interessierten Leuten Fragen beantworten und gegebenenfalls neue Farbtabellen erstellen konnte.

Am ersten Tag wurden des weiteren in 2 Stunden Abständen immer wieder Spiele gezeigt. Der Andrang des Publikums war immer sehr groß, da dies für das recht fachkundige Publikum eine angenehme Abwechslung zum normalen Messegeschehen war.

Außerdem waren Mitarbeiter der Middle-Size Teams des Fraunhofer AIS und der Universität Stuttgart da, die an den anderen Tagen ebenfalls Spiele vorführten.

Probleme gab es hier weniger, da für die Spielfelder eine adäquate Beleuchtung vorhanden war. Ebenso gab es keine Probleme mit dem WLAN, da nur zwei Funknetze – das der Midsize-League und das der Hellhounds – vorhanden waren.

10.3.3 Campusfest Universität Dortmund

Auch auf dem Campusfest der Universität Dortmund im Juli, kurz vor der Weltmeisterschaft, wurde die Arbeit der Hellhounds ausgestellt. Dabei wurde, da nicht genügend Platz für das komplette Feld zur Verfügung stand, wiederum der Schaukasten eingesetzt. Zusätzlich wurde ein Beamer verwendet, um ein Video mit Spielszenen der German Open zu zeigen (siehe Abbildung 10.3 auf der nächsten Seite).

Die Positionierung gleich am Eingang des Physikgebäudes war eigentlich nicht schlecht, denn jeder, der dort vorbeikam, kam auch gleich am Schaukasten vorbei.

Hierbei kam es wiederum zu etwas Problemen mit dem Licht. Der 500W Scheinwerfer der Feldbeleuchtung war einfach zu hell, so dass die Bilder fast komplett weiß waren. Durch eine indirekte Beleuchtung über die Decke wurde dieses Problem dann gelöst. Dies führte aber dazu, dass das Beamer-Bild schlechter zu erkennen war.

Abbildung 10.3: Fallrückzieher bei den German Open gegen die Darmstadt Dribbling Dackles

Anhang

Anhang A

XABSL-Symbole

Im Laufe der Projektgruppe-Zeit wurden neue Symbole implementiert, die in diesem Anhang aufgelistet werden. Da einige der Symbole nur zur Fehlersuche und zum Testen benutzt wurden, werden nicht alle hier vorgestellten Symbole auch im endgültigem Verhalten benutzt.

- **shouldi.potentialfield.priority**
  Mit diesem Symbol kann die Priorität des Potenzialfeldes abgefragt werden..
- **shouldi.kickball.priority**
  Mit diesem Symbol kann die Priorität des Advises kickball abgefragt werden. Als Ausgabe wird ein Wert zwischen 0 und 100 ausgegeben.
- **shouldi.stand.priority**
  Mit diesem Symbol kann die Priorität des Advises stand abgefragt werden. Als Ausgabe wird ein Wert zwischen 0 und 100 ausgegeben.
- **shouldi.searchforlandmarks.priority**
  Mit diesem Symbol kann die Priorität des Advises searchforlandmarks abgefragt werden. Als Ausgabe wird ein Wert zwischen 0 und 100 ausgegeben.
- **shouldi.searchforball.priority**
  Mit diesem Symbol kann die Priorität des Advises searchforball abgefragt werden. Als Ausgabe wird ein Wert zwischen 0 und 100 ausgegeben.
- **Ishoulddothis**
  Das Symbol gibt an, welche der folgenden Anweisungen die höchste Priorität hat. Diese Werte sind möglich
  - dothis.followpotentialfield
  - dothis.kickball
  - dothis.stand
  - dothis.searchforlandmarks
  - dothis.searchforball

Dieses Symbol wurde zu dem Zweck implementiert die vier vorherigen Symbole zusammenzufassen und so kleinere XABSL Optionen zu ermöglichen.

- **Advisesactionid**
  Als Ausgabe dieses Symbols steht die Special Action ID, die nach Meinung des Trainers der XABSL-Automat durchführen soll.

- **Advisestrictness**
  Dieses Symbol wurde eingeführt um dem XABSL-Automat mitzuteilen, wie strikt der Spieler auf einen Advise hören soll.

- **Advisekickdirx**
  Dieses ist die X-Richtung in die der Spieler den Ball spielen soll, wenn er ein entsprechendes Advise bekommt.

- **Advisekickdiry**
  Dieses Symbol entspricht dem vorherigen, gibt aber die Y-Richtung an.

- **Adviseplayerorientation**
  Der Spieler soll sich in diese Richtung drehen, wenn er auf den Trainer hört.

- **Advisenumber**
  In diesem Symbol findet sich der Advise, von dem der Trainer meint, dass er benutzt werden soll.

  Die folgenden Werte sind möglich:

  - advise.free
  - advise.valOwnPos
  - advise.valOwnBall
  - advise.doSpecialAction
  - advise.weScored
  - advise.weRule
  - advise.penalized
  - advise.ballKick
  - advise.dribble

- **advistoavoidball**
  Dieses Symbol sagt dem Spieler, ob er, wenn er nah genug ist, zum Ball laufen (advistoavoidball.yes), oder an ihm vorbeigehen (advistoavoidball.no) soll.

- **potentialfield.moverelativex**
  Der Spieler soll sich relativ zu seiner Position in die X-Komponente der Richtung bewegen.

- **potentialfield.moverelativey**
  Der Spieler soll sich relativ zu seiner Position in die Y-Komponente der Richtung bewegen.

- **relative-angle-to-player**
  Dieses Symbol gibt einen relativen Winkel aus. Dieser Winkel wird aus der Position des Spielers und den eingegebenen Koordinaten relative-angle-to-player.x und relative-angle-to-player.y berechnet.

- **home-position**
  Gedacht ist dieses Symbol dafür, jedem Spieler seine Startposition auf dem Spielfeld anzugeben. Mit home-position.number muss dazu die Spielernummer angegeben werden und mit home-position.x-or-y, ob das Symbol die X- oder Y-Komponente ausgeben soll.

- **get-kickoff-angle**
  Die Funktion hinter diesem Symbol berechnet wo keine Gegner stehen und der Ball hingeschossen werden soll. Da die Gegnererkennung nicht hinreichend genug funktioniert konnte diese Funktion nicht getestet werden und wird deshalb auch nicht benutzt.

- **minpotentialfieldpriority**
  Dieses Symbol bestimmt die Minimale Priorität, die das Potenzialfeld ausgeben muss, damit sich der Spieler noch danach richtet. Wenn die Priorität unter diese Schwelle sinkt, und die minadviseimportance ebenso unterschritten wird, fällt der Spieler seine eigene Entscheidung.

- **minadviseimportance**
  Dieses Symbol entspricht der minpotentialfieldpriority beim Potenzialfeld. Es gibt deswegen kein einheitliches Symbol, weil beide Werte in verschiedenen Wertebereichen liegen können.

- **self.get-angle-to-point**
  Dieses Symbol gibt den berechneten Winkel zu einem Punkt auf dem Spielfeld aus. Mit self.get-angle-to-point.x und self.get-angle-to-point.y muss der Punkt angegeben werden.

- **strategy.orientation**
  Dieses Symbol gibt es im C++-Quellcode zweimal. Eine Ausführung bewirkt, dass die Roboter sich zum Strategiezielpunkt ausrichten, die andere momentan aktive führt dazu, dass sich die Roboter in Richtung Ball drehen

- **fieldplayer.smallest.distance-to-ball**
  Es war angedacht, mit diesem Symbol die kürzeste Strecke zwischen einem eigenem Spieler und dem Ball zu suchen und auszugeben.

- **fieldplayer.2ndsmallest.distance-to-ball**
  Die Berechnung dieses Symbols ist identisch wie des vorherigen. Nur wird im Gegensatz zu diesem die zweitkürzeste Strecke angegeben.

- **calculate-best-kick-angle**
  Als Ausgabe liefert dieses Symbol einen Winkel zu einem Mitspieler. Dieser steht von allen Spielern am günstigsten und sollte deshalb angespielt werden.

- **relative-angle-to-point**
  bewirkt das selbe wie das Symbol self.get-angle-to-point. Mit den Parametern point.x und point.y wird ein Punkt übergeben. Anschließend wird als Ausgabe ein relativer Winkel von dem Roboter zu dem Punkt berechnet.

- **single-strategy.move**
  Für diesen Spieler existiert eine Strategie, die ihn zum Bewegen veranlasst.

- **single-strategy.ball-kick**
  Eine Strategie, die besagt, dass der Spieler einen Kick ausführen soll existiert.

- **single-strategy.dribble**
  Wenn der Spieler einen Ball dribbeln soll, ist dieses Symbol mit true belegt.

- **single-strategy.ball-to-x und single-strategy.ball-to-y**
  Wenn eine Strategie besagt, dass ein Ball geschossen werden soll, so wird mit diesen Symbolen der x-Wert des Punktes angezeigt, in den der Spieler den Ball schießen soll.

- **single-strategy.destinationarea.x und single-strategy.destinationarea.y**
  Für die Strategien dribble und move wird mit Hilfe der beiden Symbole das Zielfeld angegeben in das sich der Spieler bewegen soll.

- **single-strategy.destinationarea.rotation**
  Dieses Symbol gibt bei der gewählten Strategie dribble oder move aus, wie der Spieler am Ende der Strategie ausgerichtet sein soll.

Die folgenden Symbole wurden von Mitarbeitern des Lehrstuhls während der Projektgruppe-Zeit entwickelt und werden hier deshalb erwähnt, da sie im Projekt eingesetzt wurden.

- goalie.should-defend-left
- goalie.should-defend-right
- goalie.should-defend-in-front
- block.position.x
- block.position.y
- active-kick-option
- executed-option

Literaturverzeichnis

[1] RoboCup Technical Committee, "Sony Four Legged Robot Football League Rule Book," 2003, http://www.openr.org/robocup/rule/RoboCup03_rules.pdf.

[2] F. Serra and J.-C. Baillie, "Aibo programming using Open-R SDK - Tutorial," 2003, http://www.ensta.fr/~baillie/tutorial_OPENR_ENSTA-1.0.pdf.

[3] H.D. Burkhard, U. Düffert, J. Hoffmann, M. Jüngel, M. Lötzsch, R. Brunn, M. Kallnik, N. Kuntze, M. Kunz, S. Petters, M. Risler, O. von Stryk, N. Koschmieder, T. Laue, T. Röfer, K. Spiess, A. Cesarz, I. Dahm, M. Hebbel, W. Nowak, and J. Ziegler, "German Team 2002 - Team report," 2002, http://www.tzi.de/kogrob/papers/GermanTeam2002.pdf (vom 01.08.2003).

[4] J. C. Terrillon and S. Akamatsu, "Comparative Performance of Different Chrominance Spaces for Color Segmentation and Detection of Human Faces in Complex Scene Images," http://citeseer.nj.nec.com/terrillon00comparative.html (vom 25.9.2003).

[5] J. C. Terrillon, H. Fukamachi, S. Akamatsu, and M. N. Shirazi, "Comparative Performance of Different Skin Chrominance Models and Chrominance Spaces for the Automatic Detection of Human Faces in Color Images," in *Fourth IEEE International Conference on Automatic Face and Gesture Recognition*, 2000, p. 54ff.

[6] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann, dpunkt.verlag, 1998.

[7] I. Rechenberg, *Evolutionsstrategie '94*, Number ISBN 3-772-81642-8. Frommann–Holzboog Verlag, 1st edition, 1994.

[8] I. Dahm, S. Deutsch, M. Hebbel, and A. Osterhues, "Robust Color Classification for Robot Soccer," in *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences), Lecture Notes in Artificial Intelligence*. 2004, Springer.

[9] Rafael C. Gonzalez and Richard E. Woods, *Digital image processing*, Number ISBN 0-201-50803-6. Addison-Wesley Publishing Company, 1992.

[10] D.L. Mills, "Request for Comments: 958, Network Time Protocol (NTP)," 9 1985, http://www.faqs.org/rfcs/rfc958.html (vom 22.07.2003).

[11] A. W. Stroupe, C. Martin, and T. Balch, "Distributed Sensor Fusion for Object Position Estimation by Multi-Robot Systems," 2001.

[12] Brian T. Luke, "K-Means Clustering," 2002, http://fconyx.ncifcrf.gov/~lukeb/kmeans.html (vom 26.09.2003).

[13] Brian T. Luke, "Hierarchical Clustering," 2002, http://fconyx.ncifcrf.gov/~lukeb/hiercl.html (vom 26.09.2003).

[14] Brian T. Luke, "Diversity Selection," 2002, http://fconyx.ncifcrf.gov/~lukeb/divsel.html (vom 26.09.2003).

[15] The W3-Consortium, "The Extensible Stylesheet Language Family (XSL)," http://www.w3c.org/Style/XSL/.

[16] AT&T Labs Research, "Graphviz - open source graph drawing software," http://www.research.att.com/sw/tools/graphviz/.

[17] C. Graus, "Image Processing - Convolution Filters," 2002.

[18] C. Graus, "Image Processing - Edge Detection Filters," http://www.thecodeproject.com/cs/media/edge_detection.asp (vom 25.9.2003).

[19] T. Walter, "Digitale Bildverarbeitung - Lagebestimmung einfacher geometrischer Objekte in zweidimensionalen Bildern," 2001, http://wwwipr.ira.uka.de/~megi/SEMINAR/WS_01_02/Bildverarbeitung.pdf (vom 28.08.03).

[20] C. Connot, "Ein Algorithmus zur Positionsbestimmung mobiler autonomer Roboter," 2001, http://ais.gmd.de/~paul/thesis/Connot.pdf (vom 28.08.03).

[21] O. Müller and R. Kunze, "Computergrafik," 2002, http://www-lehre.informatik.uni-osnabrueck.de/~cg/2002/Pdf/skript03.pdf (vom 28.08.03).