KickOffTUG

Erweiterung des Weltmodells

Máté WOLFRAM
M#0312966 mate.wolfram@student.tugraz.at
Institut für Softwaretechnologie
Technische Universität Graz

Abstract

Dieses Dokument dient als Zusammenfassung meiner Arbeit am Weltmodell des RoboCup Simulation League Agenten des Teams KickOff-TUG. Diese Komponente ist die grundlegende Verbindung des Agenten zwischen relativen Serverinformationen und Abstraktions- und Planebene und sorgt für die korrekte Verarbeitung und Verwaltung eingelangter Informationen über die Umwelt. Ziel der Arbeit war unter anderem die Integration fortgeschrittener Methoden zur Filterung von Sensordaten, hauptsächlich um dem Agenten die akkurate Ausführung grundlegender Aktionen zu ermöglichen, sowie die Implementierung eines Vorhersage- und Erinnerungsmodells basierend auf bekannten Weltzuständen, z.B. um auf Planebene eine vollständigere Darstellung der Umwelt anbieten zu können.

Contents

1 Einleitung 4
1.1 Subprojekt Weltmodell 4
1.2 Aufbau der Arbeit 5

2 Ziele 6
2.1 Verbesserung des Fehlers bei der Selbstlokalisation 6
2.2 Einbindung eines Vorhersagemodells 6
2.3 Abgleich des Weltmodells mit anderen Agenten 6
2.4 Sonstige Erweiterungen 6

3 Restrukturierung des Weltmodells 7
3.1 WorldState 7
3.2 Update-Struktur 8

4 Vorhersage- und Erinnerungsmodell 9

5 KalmanFilter 10
5.1 KalmanFilter allgemein 10
5.2 KalmanFilter des KickOffTUG Agenten 10

6 Positionsbestimmung 11
6.1 Grundlegendes zum Problem der Positionsbestimmung 11
6.2 Positionsbestimmung anhand zweier Flags 11
6.3 Positionsbestimmung mittels Partikelfilter 13
6.3.1 Zum Aufbau 13
6.3.2 Resampling 14

7 Richtungsbestimmung 15

8 Wahrnehmung des Balls 17
8.1 Problemstellung 17
8.2 BallFilter 18

9 Resultate 19
9.1 Positionsbestimmung 19
9.2 Wahrnehmung des Balls 21

10 Interception 23
10.1 InterceptPlayer 23

11 Hilfsklassen 25
11.1 Diagnostics 25
11.2 CircularSector 25
11.2.1 RingSector 25
11.2.2 Area 25

12 TurnNeckModel 28
12.1 Auswahl der Blickrichtung 29
12.2 Overrides 29
12.3 Resultat 30

13 Conditions 31
13.1 Dribble 32
13.2 FastestTo 33
13.3 InFrontOf 33
13.4 Unguarded 33
13.5 PointTo 33
13.6 PassRequested 34
13.7 PassInterceptable 34
13.8 BallHeadingTowardsGoal 34
13.9 OnBarBallsWayPosition 35
13.10BallCatchable 35
13.11BallConfident 35
13.12InOffside 35
13.13FreePlayer 35
13.14OnLine 35
13.15ResponsibleToCover 36

14 Zusammenfassung und Ausblick 37
14.1 Projektaufwand 37
14.2 Ausblick 37
14.3 Zusammenfassung 37

1 Einleitung

KickOffTUG[1] wurde von drei Studenten - Monika Schubert, Stephan Gspandl und Michael Reip - im Rahmen einer Bakkalaureatsarbeit der Studienrichtung Softwareentwicklung und Wissensmanagement am Institut für Softwaretechnologie (IST) unter der Aufsicht von Prof. Franz Wotawa ins Leben gerufen. [SGR05]

Es handelt sich hierbei um ein Multi-Agent System, das mit dem SoccerServer der Robocup[2] Simulation League interagiert, um gegen andere Teams in einer virtuellen Umgebung antreten zu können. Ziel der Simulation League ist es, die Entwicklung der Agenten im Bereich der künstlichen Intelligenz voranzutreiben, indem die durch Hardwareanforderungen verursachten Hindernisse (Mangel an Zeit, Personal, finanziellen Mitteln) beseitigt werden.

Das Ergebnis ist die realistischste aller Ligen des Robocup, denn sie ist die einzige in der das FIFA Reglement Gültigkeit hat, sei es aufgrund der Tatsache, dass pro Team 11 Agenten gegeneinander antreten oder weil die Proportionen auf dem Spielfeld genau mit denen in einem realen Fußballstadion übereinstimmen.

1.1 Subprojekt Weltmodell

Im Sommer 2006 bin ich als Entwickler ins Projekt KickOffTUG aufgenommen worden, mit der Aufgabe, das Weltmodell unseres Agenten zu verbessern und zu erweitern. Dieses Subprojekt existiert im Rahmen einer Bakkalaureatsarbeit, die ich mit diesem Dokument beschreiben und in dieser Form abschließen möchte. Aufgrund der in den vergangenen Monaten gesammelten Erfahrung ist jedoch meine weitere Teilnahme am Gesamtprojekt sehr wahrscheinlich, in welchem Rahmen auch immer.

Eine genaue Beschreibung des Systems vor meiner Teilnahme am Projekt würde vermutlich den Rahmen sprengen, genauso auch Informationen über Robocup und die Simulation League. Ich möchte den Leser daher auf das KickOffTUG Team Description Paper[3] verweisen, das von den Gründungsmitgliedern von KickOffTUG verfasst wurde. Diese Lektüre ist für einen Neueinstieg in dieses Thema unbedingt zu empfehlen.

Ebenso essentiell ist auch das Verständnis des SoccerServers[4][CFH+02] , der die Basis der Simulation League bildet und von dessen Bedingungen unsere Agenten abhängig sind. Weltwahrnehmung, Kommunikation und Aktionen der Agenten laufen alle über eine Instanz dieses Servers.

[1] `http://kickofftug.tugraz.at`

[2] `http://www.robocup.org/`

[3] `http://kickofftug.tugraz.at/assets/TDP-KickOffTUG.pdf` bzw. **/related/TDP-KickOffTUG.pdf** auf der CD

[4] **/related/SoccerServerManual.pdf** auf der beiliegenden CD

1.2 Aufbau der Arbeit

Die Arbeit ist wie folgt gegliedert. Nach einer kurzen Präsentation der im Sommer 2006 festgelegten Ziele in Abschnitt 2 folgen genaue Beschreibungen der Module, die im Laufe der Arbeit verändert, erweitert oder hinzugefügt wurden. Das Dokument schließt mit Informationen zum benötigten Arbeitsaufwand und einem Ausblick auf zukünftige Änderungen.

2 Ziele

2.1 Verbesserung des Fehlers bei der Selbstlokalisation

Ziel war es, einen KalmanFilter zur Filterung der mit einem Rauschen behafteten **see**-Nachrichten des SoccerServers zu verwenden. Diese Erweiterung wurde bereits in der Anfangsphase des Projekts implementiert und filtert sowohl die anhand von Flags bestimmbare Position als auch die Ausrichtung des Agenten. Mit der Filterung der eigenen Position wird auch die Berechnung der absoluten Position anderer Objekte genauer. Der KalmanFilter wurde unlängst durch den noch effizienteren Partikelfilter ersetzt.

2.2 Einbindung eines Vorhersagemodells

Es war uns wichtig, über ein möglichst einfaches Interface zukünftige Weltmodelle, die anhand aktueller Objektinformationen hergeleitet werden, abfragen zu können. Wir haben somit die Möglichkeit, simulierte Spielzustände in verschiedenen Routinen abzufragen, und die Qualität der Simulation völlig unabhängig davon laufend zu verbessern. Das Vorhersagemodell kann gleichzeitig auch als Erinnerungsmodell genützt werden. Eine genauere Erklärung folgt im Abschnitt 4.

2.3 Abgleich des Weltmodells mit anderen Agenten

Dieser Teilbereich wurde in Zusammenarbeit mit der Bakkalaureatsarbeit Team Strategy von Christoph Zehentner (wird veröffentlicht) implementiert. Das Weltmodell bietet eine Schnittstelle für Informationen, die wir über say-Nachrichten senden und empfangen.

2.4 Sonstige Erweiterungen

Im Laufe der Weiterentwicklung des Systems stießen wir auf neue Hindernisse bzw. sahen neue Möglichkeiten, wie wir den Agenten verbessern konnten. Diese Erweiterungen konnten wir zu Beginn meiner Teilnahme nicht genau definieren oder sie waren noch nicht in Planung und scheinen daher auch nicht im Proposal zur Bakkalaureatsarbeit[5] auf. Zu diesen Erweiterungen gehören grob zusammengefasst die Restrukturierung des Weltmodells (Kapitel 3), eine neue Routine für Interception (Kapitel 10) und Conditions (Kapitel 13), die mit booleschen Werten auf bestimmte Abfragen zum Zustand der Umgebung antworten können. Sie werden in den gleichnamigen Kapiteln behandelt.

[5] **/thesis/Proposal.pdf**

3 Restrukturierung des Weltmodells

Um das Weltmodell übersichtlicher zu gestalten, haben wir beschlossen, die Anforderungen an diese Klasse neu zusammenzufassen und sie anhand dieser Informationen einer Restrukturierung zu unterziehen. Das erleichterte wiederum die Aufgabe, neue Funktionalität hinzuzufügen. Das Weltmodell hält nun eine Liste von Objekten der Klasse WorldState, die den absoluten Zustand der Umwelt in einem bestimmten Zyklus beschreiben. Es wird garantiert für jeden Zyklus ein WorldState erstellt und mit den verfügbaren Informationen aus dem letzten WorldState vereinigt. Es wird sichergestellt, dass

- von jedem Objekt in Erfahrung gebracht werden kann, wann es zuletzt gesehen wurde
- jedes Objekt "sauber" gespeichert wird, also dass der tatsächliche Zustand im besagten Zyklus zur Verfügung steht und nicht eine simulierte Vermutung der Position/Geschwindigkeit etc

3.1 WorldState

Ein Objekt der Klasse WorldState ist folgendermaßen aufgebaut:

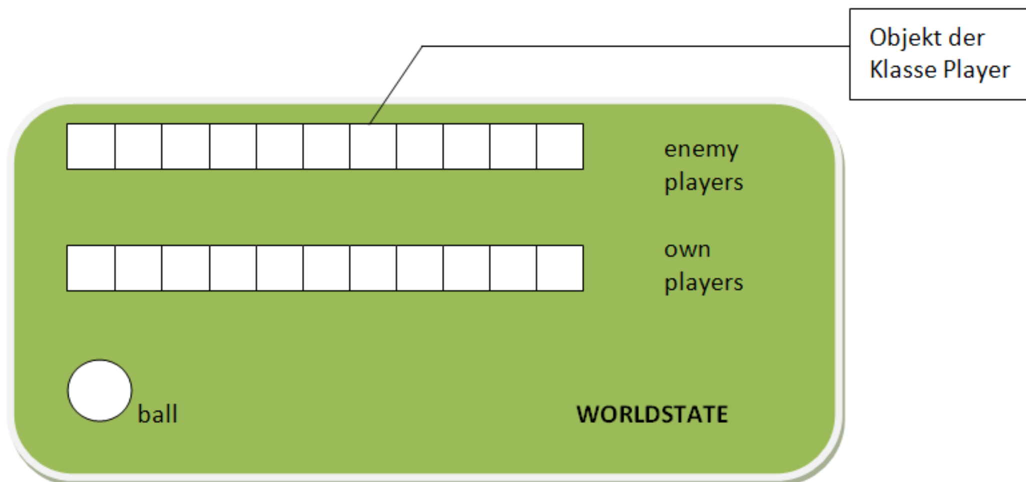

Figure 1: WorldState

Die Spieler beider Teams sind in separaten Arrays gespeichert. Die Größe dieser Arrays ist mit 11 festgelegt. Falls weniger Spieler am Platz sind, werden diese einfach nicht gesetzt. Die Spieler werden anhand ihrer Spielernummer in das passende Array eingeordnet. Um diese Zuordnung zwischen Spielernummer und Platz im Array so intuitiv wie möglich zu gestalten, fassen die Arrays in Wirklichkeit 12 Spieler, wobei der 0-te nie gesetzt wird, da die Spielernummern

mit 1 beginnen. Der Timestamp, über den wir herausfinden können, in welchem Zyklus eines der vorhandenen Objekte zuletzt gesehen wurde, ist in der Klasse MovingObject implementiert, von der sowohl Spieler als auch Ball abgeleitet sind.

3.2 Update-Struktur

Um ein Erinnerungsmodell zu ermöglichen, kann ein WorldState die nützlichen Informationen aus dem letzten WorldState übernehmen, ohne sich selbst zur Gänze damit zu überschreiben. Diese Update-Struktur zieht sich von oberster Ebene bis in die niedrigsten Elemente der Klassenhierarchie, somit müssen z.B. auch Spieler und Ball nicht komplett mit einer neuen Version überschrieben werden, sondern können Informationen ihrer Vorgänger übernehmen.

4 Vorhersage- und Erinnerungsmodell

Die neue Struktur des Weltmodells erlaubt uns die Implementierung eines bequem verwendbaren Vorhersage- und Erinnerungsmodells. Aus der Kombination von Timestamps und dazugehörigen sauberen Objektzuständen lässt sich jederzeit von einem beliebigen Zeitpunkt angefangen ein aktueller Weltzustand simulieren. Wir müssen dazu nur die Differenz zwischen aktuellem Timestamp und gewünschtem Zyklus berechnen und die resultierende Anzahl an Zyklen simulieren, was wiederum die Klasse Simulation übernimmt.

Erinnerung und Vorhersage hängen in diesem Kontext eng miteinander zusammen. Um zu wissen, was seit dem letzten Blickkontakt mit einem Objekt passiert ist, müssen wir dessen damalige Position und Timestamp kennen. Anschließend können wir auf den aktuellen Zyklus aufsimulieren und so erahnen, was hinter unserem Rücken passiert. Der nächste Blickkontakt korrigiert dann nur mehr die errechneten Daten. Dieses System erlaubt uns sogar, die Wahrscheinlichkeit der Korrektheit einer Simulation vorherzusagen, da wir natürlich wissen wie akkurat diese in einer bestimmten Situation ist und wie oft sie angewandt wurde.

Während das Erinnerungsmodell in fast jedem Zyklus zur Anwendung kommt, verwenden wir das Vorhersagemodell hauptsächlich beim Abfangen des Balls. Der Algorithmus dazu wird in Kapitel 10 - Interception behandelt.

5 KalmanFilter

Der KalmanFilter wurde als Generic implementiert, der mit Werten vom Typ Cartesian und Double umgehen kann. Die Verwendung anderer Typen wird explizit verhindert. Eine Erweiterung ist mit geringem Aufwand möglich, war aber bislang nicht erforderlich.

5.1 KalmanFilter allgemein

Die Idee eines KalmanFilters ist es, eine Abfolge von Messwerten, z.B. Positionsinformationen zu diskreten Zeitabständen im Laufe der Fortbewegung eines Roboters, zu glätten. Dazu Bedarf es zu jedem Zeitpunkt einer Messung und einer Abschätzung der Position anhand von Odometrie. Sämtliche Messungen, auch die Abschätzung der Position, müssen richtig gewichtet werden, um ein optimales Ergebnis zu erzielen. Dazu muss sowohl die Wahrscheinlichkeitsverteilung der Messung als auch die der Schätzung bekannt sein, und die Schätzung des neuen Zustands (der Zustandsübergang) muss natürlich auch möglich sein. Es ist also unerlässlich, die Charakteristiken der Umgebung zu kennen.

5.2 KalmanFilter des KickOffTUG Agenten

Für im Filter verwendete Formeln habe ich den Artikel **Stochastic models, estimation, and control**[6] von Peter S. Maybeck herangezogen [May99].

Der Filter nimmt jeweils ein Tupel (Messung, Varianz) entgegen und speichert diese Werte in zwei parallel laufenden Listen. Wird die Methode **calculateAndReturnBestResult()** aufgerufen, berechnet der Filter aus allen in der Liste befindlichen Werten mit dazugehörigen Gewichtungen ein Optimum, speichert Ergebnis und dazugehörige Varianz als einzigen Wert ab und returniert die Messung. Die Liste der Messungen und Varianzen wird also zu einem einzigen optimalen Wert verschmolzen. Es wird dabei über die gesamte Liste iteriert, das aktuelle Massenzentrum als Schätzung und der neue Wert als die nächste Messung interpretiert.

Sei die Schätzung der Position zum Zeitpunkt x $\hat{x}(t_x)$ und Varianz des Fehlers der entsprechenden Schätzung $\sigma_x^2(t_x)$. Die Messung unserer Position ist z_x zum Zeitpunkt t_x. $K(t_x)$ ist der Korrektor zum Zeitpunkt x.

Es gelten folgende zwei wichtigen Formeln:

$$\hat{x}(t_k) = \hat{x}(t_{k-1}) + K(t_k)[z_k - \hat{x}(t_{k-1})] \tag{1}$$

$$K(t_k) = \sigma_{z_{k-1}}^2 / (\sigma_{z_{k-1}}^2 + \sigma_{z_k}^2) \tag{2}$$

In der Praxis kann unser Filter so gleichzeitig mit einer Unzahl an Messungen gefüllt werden, die sich aber alle auf ein und dieselbe Lösung beziehen.

[6] **/related/Maybeck99.pdf**

6 Positionsbestimmung

6.1 Grundlegendes zum Problem der Positionsbestimmung

Bevor die Informationen über Flags bei uns anlangen, werden die berechneten Werte vom Server gerundet. Die an uns gesendete Distanz zu einem Objekt wird folgendermaßen berechnet:

$$d' = Q(e^{Q(\log(d), quantize_step)}, 0.1) \tag{3}$$

$$Q(v, q) = ceil(\frac{v}{q}) * q \tag{4}$$

Der Parameter **quantize_step** wird vom Server vorgegeben, genauso wie **quantize_step_l**, der für Flags verwendet wird. Richtungsinformationen werden auf ganze Zahlen gerundet. Kehren wir diese Formeln um, erhalten wir für jede vernommene Distanz- oder Richtungsinformation den minimal und maximal möglichen Wert. Die folgenden Formeln stammen aus dem Team Description Paper des Simulation Teams UvA[7] [dBK02].

$$r_{min} = e^{invQMin(\ln(invQMin(r', 0.1)), quantize_step_l), 0.1)} \tag{5}$$

$$r_{max} = e^{invQMax(\ln(invQMax(r', 0.1)), quantize_step_l), 0.1)} \tag{6}$$

$$invQMin(q, Q) = (rint(\frac{q}{Q}) - 0.5) * Q \tag{7}$$

$$invQMax(q, Q) = (rint(\frac{q}{Q}) + 0.5) * Q \tag{8}$$

Diese Informationen können wir nun verwenden, um entweder Messungen richtig zu gewichten (Positionsbestimmung anhand jeweils zweier Flags und anschließende Filterung mittels KalmanFilter) oder mit einem Ausschlussverfahren den Fehler einzugrenzen (Partikelfilter).

6.2 Positionsbestimmung anhand zweier Flags

Um den KalmanFilter effektiv anwenden zu können, war es notwendig, einen neuen Algorithmus zur Berechnung der Position des Agenten einzuführen. Dieser Algorithmus ist an die Lösung des Teams Brainstormers[8] von der Universität Osnabrück[9] angelehnt. Sie findet sich jedoch auch im UvA Team Description Paper wieder.

Die Position des Agenten wird dabei ausschließlich aus der relativen visuellen Information über zwei fix festgelegte Flags und das Wissen über deren absolute Positionen hergeleitet.

[7] **/related/UVA-TDP.pdf**

[8] http://www.ni.uos.de/index.php?id=3

[9] http://www.uos.de/

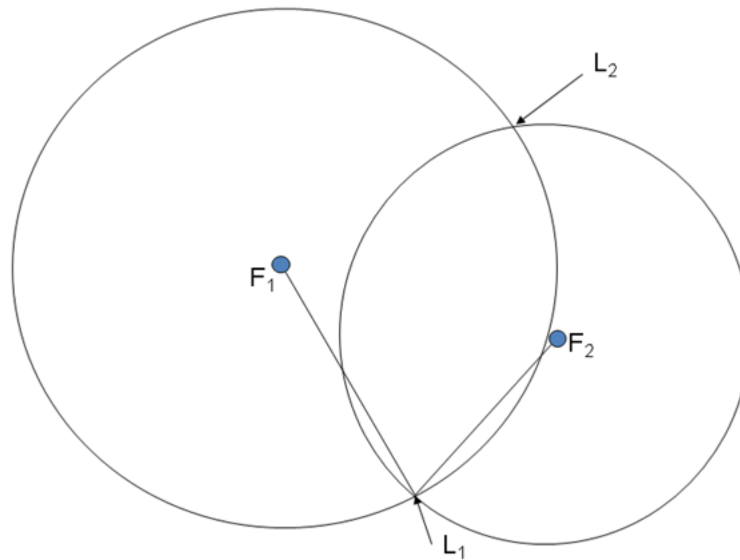

Figure 2: Positionsbestimmung

Die Flags F_1 und F_2 sind gegeben. Wir kennen den Winkel, in dem wir sie sehen, wie weit wir von ihnen entfernt sind und an welchen absoluten Positionen sie sich befinden. Folglich zeichnen wir zwei Kreise mit jeweils einer Flag als Mittelpunkt und der Distanz zu einer Flag als Radius. Wenn wir die beiden Kreise nun miteinander schneiden, erhalten wir zwei Schnittpunkte (oder Lösungen) L_1 und L_2. Einer dieser Schnittpunkte ist unsere Position, in obiger Abbildung offenbar der Punkt L_1. Da die Reihenfolge, in der der Algorithmus zur Schnittpunktberechnung die Ergebnisse liefert und der Winkel zwischen Flags und unserer Blickrichtung[10] in direktem Zusammenhang stehen, kann ohne zusätzlichen Aufwand eindeutig bestimmt werden, welcher der beiden Schnittpunkte eine Lösungsmöglichkeit für die Frage nach unserer Position repräsentiert und welche Nonsens sein muss.

Der Server liefert jedoch meistens so stark verfälschte Werte, dass ein Durchlauf dieses Algorithmus mit zwei Flags nicht ausreicht, um unsere Position mit ausreichender Sicherheit zu bestimmen. Zum Glück stehen uns in der Regel mehr als zwei Flags zur Verfügung. Aus allen möglichen Kombinationen von Flags werden mögliche Positionen berechnet und diese mittels KalmanFilter gefiltert. Zur Gewichtung der Ergebnisse wird die sich aus jeweils zwei Flags ergebende Fläche von möglichen Positionen verwendet.

[10] Wir haben stets eine linke und eine rechte Flag

6.3 Positionsbestimmung mittels Partikelfilter

Der Partikelfilter stellt die Position des Agenten durch Filterung möglicher Lösungen anhand verschiedener Flags fest. Dabei werden in einem mehrstufigen Ausschlußverfahren alle Werte, die nicht in den Bereich des Möglichen fallen, eliminiert. Übrig bleiben nur solche Partikel, die die Position des Agenten repräsentieren können. Der Mittelwert dieser Menge wird als Schätzung der Position herangezogen.

Einer der großen Vorteile dieser Methode ist, neben ihrer Effektivität, die später unter Beweis gestellt wird, ihre Robustheit. Sie benötigt lediglich eine Flag, um eine Position zu liefern. Bei Anwendung von Translation älterer Partikel kommt sie einige Zeit sogar ohne diese Information aus. Der Nachteil dieser Methode ist die höhere Laufzeit, die wir in Kauf nehmen müssen.

Die grundlegende Funktion des Filters ist eine Approximation der sich nach mehreren Flags ergebenden Fläche, auf der sich der Agent befinden muss. Die entsprechende Darstellung ist dem Dainamite[11] Team Description Paper 2006[12] [E$^+$06] entnommen (Figure 3).

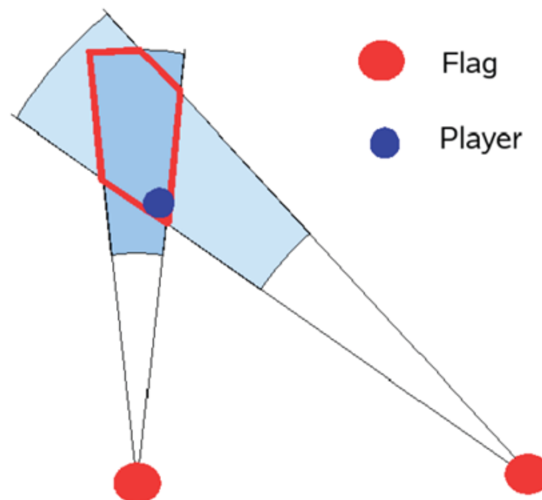

Figure 3: Partikelfilter

6.3.1 Zum Aufbau

Der Partikelfilter arbeitet mit Objekten der Klasse Area (S. 25) um die Bereiche, die anhand einzelner Flags festgestellt werden, zu verwalten. Pro Flag wird also

[11] http://www.dainamite.de/
[12] **/related/DainamiteFramework2006-11-23.pdf**

eine neue Area angelegt und geprüft, welche Partikel in diesen Bereich fallen und welche nicht. Letztere werden gelöscht.

6.3.2 Resampling

Um eine konstante Anzahl an Partikeln in jedem Schritt zu garantieren, kann man vor jeder Überprüfung mit einer neuen Area ein Resampling folgendermaßen durchführen:

Man berechne Erwartungswert und Standardabweichung aller Partikelpositionen, aus denen eine Wahrscheinlichkeitsfunktion resultiert. Man wähle nun, solange die maximale Anzahl an Partikeln nicht erreicht ist, einen zufälligen Partikel aus der Menge und addiere zu dessen x- und y-Koordinate einen zufälligen Wert entsprechend der eben errechneten Wahrscheinlichkeitsfunktion. Mit diesen Koordinaten erstelle man einen neuen Partikel.

7 Richtungsbestimmung

Ein neuer Algorithmus zur Filterung von Richtungsinformationen ist unter dem Namen DirFilter verfügbar. Dieser arbeitet, wie der Partikelfilter, mit einem Ausschlussverfahren, mit dem Unterschied, dass die resultierende Fläche von möglichen Lösungen nicht durch Partikel simuliert werden muss, sondern vielmehr exakt berechnet werden kann.

Eine Richtungsinformation beinhaltet immer maximal $\pm 0.5°$ Fehler. Es bietet sich also an, einen Kreissektor[13] mit einem Gesamtwinkel von 1° und der Richtungsinformation als zentrale Richtung anzulegen. Wir wissen nun, dass die tatsächliche Richtung innerhalb dieses Sektors liegen muss. Üblerlappen wir diesen nun mit einem zweiten Kreissektor (für eine weitere Richtungsinformation) erhalten wir den neuen Sektor möglicher Richtungen mit der gemeinsamen Fläche der beiden ursprünglichen Sektoren. Haben wir alle Informationen, die uns zur Verfügung stehen, konsumiert, liefern wir die zentrale Richtung des letzten Kreissektors als Ergebnis zurück.

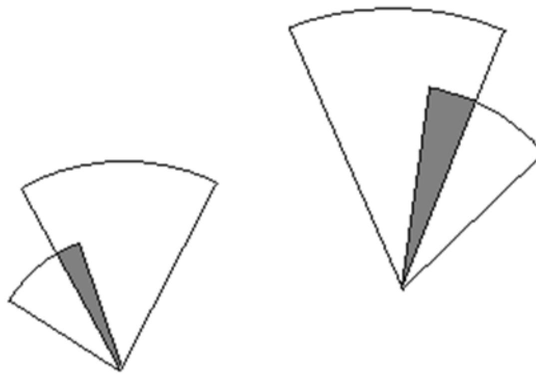

Figure 4: DirFilter

Die Radien der Kreissektoren sind in Figure 4 nur zu Demonstrationszwecken unterschiedlich gewählt. In Wirklichkeit ist dieses Detail irrelevant. Die Methode zur Berechnung der Überlappung zweier Sektoren wird in ähnlicher Form auch im TurnNeckModel[14] angewandt. Es muss hierbei lediglich erkannt werden, wie die beiden Sektoren relativ zueinander ausgerichtet sind und der Winkel zwischen den entsprechenden Kanten berechnet werden.

So einleuchtend die allgemeinen Vorteile dieses Filters auch sind, kommt er zur Zeit nicht zum Einsatz, da nur eine Richtungsbestimmung aufgrund von Vektoren zwischen der Positionsabschätzung und den absoluten Positionen von Flags genug mögliche Ergebnisse liefern würde. Diese Messung ist jedoch durch die Abweichung der eigenen Position bereits zu stark fehlerbehaftet, weshalb wir

[13] CircularSector, S.25
[14] TurnNeckModel, S.28

unsere Richtung weiterhin mit Lines feststellen. Dieser Algorithmus wiederum liefert leider i.d.R. nur eine Lösung (da wir nur eine Line sehen), wodurch eine Filterung wenig Sinn macht.

8 Wahrnehmung des Balls

8.1 Problemstellung

Das wohl schwerwiegendste Problem bei der Verwertung der eingelangten Informationen über den Ball ist die lückenhafte Dokumentation des SoccerServers. Die genauen Charakteristiken des Servers müssen erforscht werden, bevor man beginnen kann, nach Lösungen zu suchen, und das ist selbstverständlich eine zeitraubende Aufgabe. Dieser Abschnitt besteht aus Informationen, die wir aufgrund zahlreicher Tests gesammelt haben. Für die meisten ließen sich bereits weitgehend sichere Lösungen finden, die hier präsentiert werden sollen.

Eines der unangenehmen Probleme ist das Fehlen von Geschwindigkeitsinformationen, trotz Sichtkontakt zum Ball. Grund: Im Server ist ein Bereich um den Agenten definiert, in dem er unabhängig vom Blickwinkel sämtliche Objekte erkennen kann. Was leider nicht im Manual des SoccerServers dokumentiert ist, ist die Tatsache, dass die Geschwindigkeitsinformation des Balls in diesem Bereich nur dann übertragen wird, wenn wir tatsächlich in dessen Richtung blicken.

Die Geschwindigkeitsinformation des Balls ist für uns unerläßlich, da wir oft schon zu Beginn eines Zyklus Entscheidungen treffen und Kommandos schreiben wollen und wir dazu sowohl neue Position als auch Geschwindigkeit des Balls aufgrund alter Werte aufsimulieren müssen. Der Grund dafür ist die oft späte Ankunft von **see**-Nachrichten, die unser Zeitfenster für notwendige Berechnungen zu stark einengen würde. Natürlich lässt sich dieses Problem einigermaßen umgehen, indem wir die bisherige Geschwindigkeit des Balls weitersimulieren. Folgende Schwierigkeiten können daraus jedoch resultieren:

- Wenn seit mehreren Zyklen die Geschwindigkeitsinformation fehlt, wird die Information verständlicherweise ungenau.
- Wenn der Ball unerwartet bewegt wird (von einem anderen Spieler), können wir dessen Bewegung nicht mehr simulieren, da wir keine Informationen über Kick-Befehle des Gegners erhalten.
- Im Falle einer Kollision mussen die kollidierenden Objekte so weit zurückgesetzt werden, bis sich ihre Flächen nicht mehr überlappen. Zudem werden ihre Geschwindigkeitsvektoren jeweils mit -0.1 multipliziert. Die Simulation einer Kollision ist trivial, ganz im Gegensatz zu deren Erkennung. Wenn keine Information über die Geschwindigkeiten der Objekte vorliegt, müssen wir die Positionsinformationen zur Kollisionserkennung verwenden, die jedoch ungenau sind. Momentan verwenden wir die Geschwindigkeit des eigenen Agenten als Anhaltspunkt, um Kollisionen zwischen dem Agenten und dem Ball festzustellen. Diese Information erhalten wir mit jeder **sense**-Message.

8.2 BallFilter

Um möglichst zu jedem Zeitpunkt die genaue Position und Geschwindigkeit des Balls vorhersagen zu können, haben wir die Klasse **BallFilter** eingebaut, die auf einem **KalmanFilter** basiert. Er übernimmt die relativen Informationen, die mit einer **see**-Nachricht einlangen bzw. Benachrichtigungen über Kicks und Kollisionen infolge einer erhaltenen **sense**-Nachricht. Es wird momentan lediglich die Geschwindigkeit gefiltert. Der Zustandsübergang findet bei Ankunft einer **sense**-Nachricht statt und wird folgendermaßen (siehe auch [dBK02]) durchgeführt:

$$\vec{v}_{t+1} = \vec{v}_t * ball_decay \tag{9}$$

$$var_{t+1} = var_t * ball_decay^2 + (\frac{2 * ball_rand * ||\vec{v}_t||}{12})^2 \tag{10}$$

Der Ballfilter arbeitet mit absoluten Geschwindigkeitswerten, was momentan ein Problem darstellt, da in der Wahrnehmung unserer eigenen Geschwindigkeit von Zeit zu Zeit unerklärliche Fehler auftauchen, die wir bisher noch mit keiner bestimmten Aktion und keinem Weltzustand in Zusammenhang bringen konnten. Die Suche nach der Dokumentation dieser Eigenschaft lieferte wie erwartet keine Resultate. Solange dieser Fehler nicht behoben ist, kann der ansonsten funktionstüchtige Ballfilter nicht zum Einsatz kommen, da ein grober Fehler, der nicht als solcher erkannt wird und daher auch nicht entsprechend gewichtet ist, die Messungen über viele Zyklen stark verfälscht. Der Filter wird aktiviert, sobald entweder dieser Fehler behoben ist oder wir die Simulationsmethoden für relative Ballinformationen inkludiert haben, da wir dann den Filter mit diesen relativen Werten füllen könnten. Damit würden wir den eben genannten Fehler einfach umgehen.

9 Resultate

9.1 Positionsbestimmung

Der KalmanFilter liefert bei normalem Blickwinkel hervorragende Ergebnisse[15] (Figure 5).

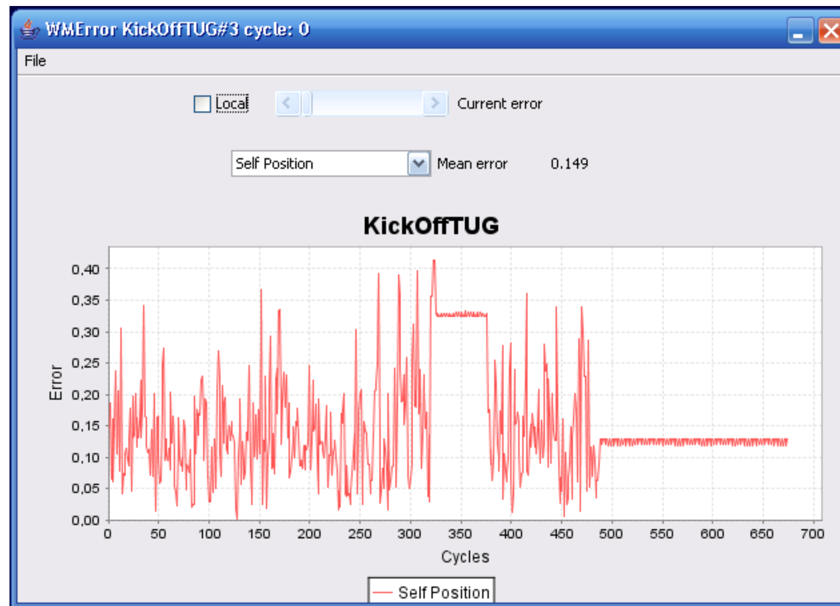

Figure 5: Positionsbestimmung (KalmanFilter / 90° Winkel)

Bei unserem neuen ViewModel ergibt sich leider das unerwartete Problem, dass wir nur sehr selten optimale Messungen erhalten. Unsere Spieler haben hauptsächlich einen Blickwinkel von 45°, den sie von Zeit zu Zeit auf 90° ändern, da wir ohnehin nicht mehr als eine See-Message in einem Zyklus benötigen. Die Verarbeitung geht von einem Optimalwinkel von 90° zwischen zwei aktuell erblickten Flags aus. Eine gute Annäherung an diesen Winkel kann somit nur selten erreicht werden, woran die Positionsbestimmung offensichtlich leidet. Daraus folgt also ein gröberer Fehler (Figure 6)

Genau hier hilft uns der Partikelfilter weiter. Dieser benötigt weniger Messungen, um ein optimales Ergebnis zu liefern. Die Grafik (Figure 7) zeigt den Fehler über 200 Zyklen, wenn ausschließlich ein Partikelfilter zur Bestimmung der Position herangezogen wird.

[15] Zum Vergleich: Meßergebnis im Sommer 2006 lieferte einen durchschnittlichen Fehler von 0.6

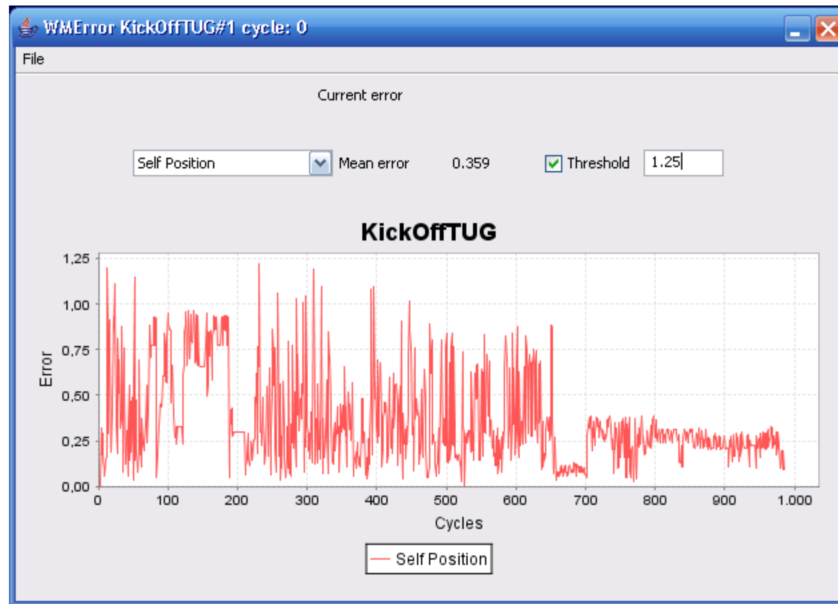

Figure 6: Positionsbestimmung (KalmanFilter / überwiegend 45° Winkel)

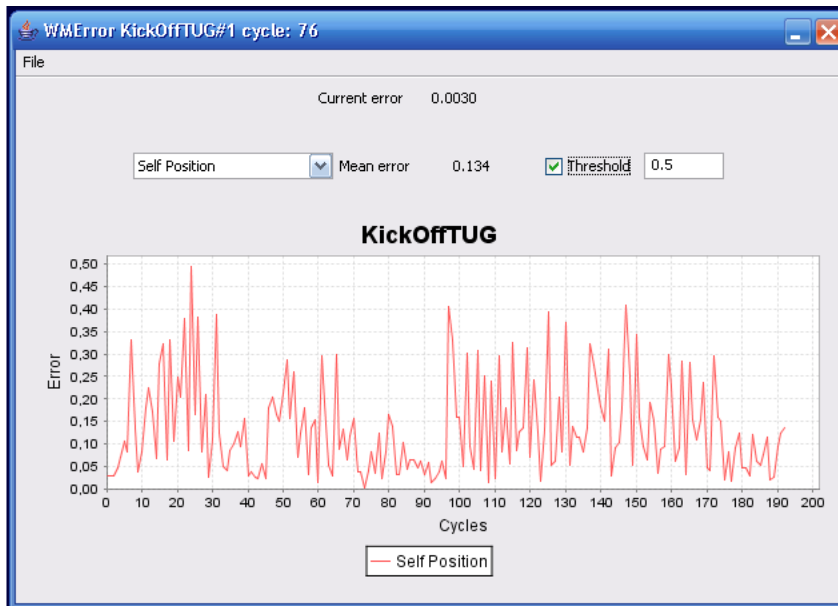

Figure 7: Positionsbestimmung (Partikelfilter)

9.2 Wahrnehmung des Balls

Die Problematik der Ballwahrnehmung wurde bereits in Kapitel 8 diskutiert. Figure 8 und Figure 9 stellen die Resultate der Verbesserungen im Weltmodell dar.

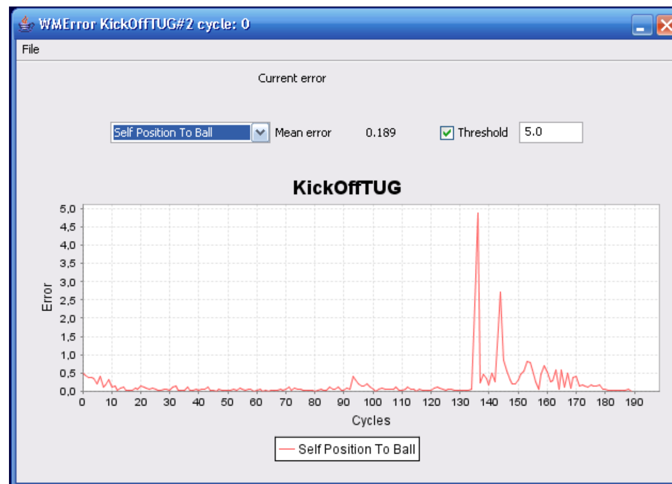

Figure 8: Ballposition

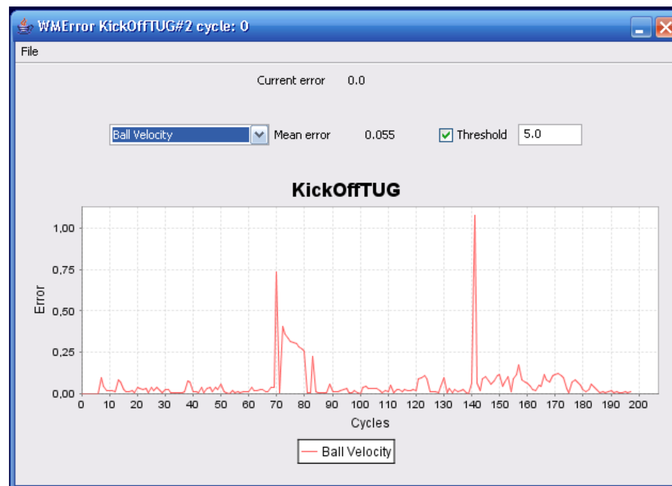

Figure 9: Ballgeschwindigkeit

10 Interception

Interception, also das Abfangen von beweglichen Objekten ist eines der komplizierten Teilprobleme der Robotik, das auch in unserer Liga zu finden ist. Es wird z.B. in [OM02][16] beschrieben. Eines der Hauptprobleme hierbei ist die Abschätzung der Zeit, die ein Objekt benötigt, um einen bestimmten Punkt zu erreichen. Hat man diese Hürde einmal überwunden, beginnt die Suche nach einem effizienten Algorithmus. Zum Glück handeln die Agenten hier in diskreten Zeiteinheiten, was diesen zweiten Teil erheblich erleichtert.

Zur Berechnung der Wegzeit verwenden wir momentan für den Ball die Vorgaben des Servers und für Spieler einen Durchschnittswert, abhängig von Distanz und Ausrichtung, der bislang zufriedenstellende Ergebnisse geliefert hat. Eine Simulation wäre performancemäßig aufwändiger und ausserdem inakkurat, da wir nicht wissen können, über wieviel Stamina andere Spieler noch verfügen und was sie in zukünftigen Zyklen unternehmen werden.

Der Algorithmus zur Berechnung eines Rendezvouspunktes für ausschließlich diskrete Zeitpunkte funktioniert wie folgt. Sei das abzufangende Objekt **t** und der abfangende Spieler **p**.

1. Berechne die Position von t im nächsten Zyklus. Sei diese Position x_{k+1} und die Zeit, die das Objekt t zu dieser Position benötigt jeweils die Anzahl der bisher simulierten Zyklen.

2. Messe die benötigte Zeit von der Position von p zu x_{k+1}.

3. Wenn die Wegzeit in Zyklen (aus 2.) geringer war als die Anzahl der bisher (in 1.) insgesamt simulierten Zyklen, dann liefere die aktuelle Position von t als Ergebnis, sonst fahre bei (1.) fort.

10.1 InterceptPlayer

Das Abfangen eines gegnerischen Spielers ist komplizierter als das Abfangen des Balls, da der Spieler Entscheidungen trifft, über die wir keine Informationen erhalten. Bei einem Spieler in Ballbesitz können wir davon ausgehen, dass dieser versuchen wird, sich unserem Tor zu nähern. Schon alleine mit dieser Information lässt sich eine sichere Routine definieren, mit der sich unser verteidigender Spieler dem Gegner annähert. Wir ziehen eine direkte Linie vom gegnerischen Spieler zu unserem Tor. Dabei wählen wir den Weg zur jeweils nächstgelegenen Goalflag.[17] Der Verteidiger versucht nun, sich auf dieser imaginären Linie zu positionieren und sich dabei konstant dem gegnerischen Spieler anzunähern. Um die Linie zu erreichen, wird die gleiche Methode angewandt, wie bei der normalen Interception. Die Bewegung des gegnerischen Spielers[18] wird Schritt

[16] **/related/Interception.pdf**

[17] Diese Methode ist für den Tormann derzeit nicht geeignet, da hierbei nicht unbedingt ein Punkt gewählt wird, von dem aus der größte Winkel zum Tor abgedeckt wird.

[18] Wir gehen momentan von einer konstanten Geschwindigkeit von 0.75m/cyc aus

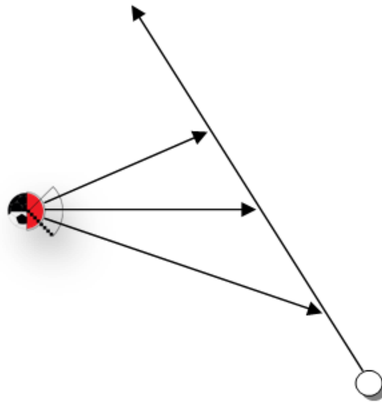

Figure 10: Interception

für Schritt aufsimuliert. Befinden wir uns bereits auf der besagten Linie, laufen wir einfach direkt auf den Gegenspieler zu.

Diese Methode garantiert uns, dass

- unser Spieler den schnellstmöglichen Weg sucht, um den Gegner zu blocken
- unser Spieler sich stets auf einer Position zwischen Gegner und Tor befindet und im Idealfall daher nie dem Gegner hinterherlaufen muss.

11 Hilfsklassen

11.1 Diagnostics

Die Klasse Diagnostics ist im Paket **worldmodel** untergebracht und bietet die Möglichkeit, schnell und unkompliziert Debugoutput in eine Datei zu schreiben. Sie enthält die statischen Methoden **write** und **writeln**, die beide einen String entgegennehmen. Sie lesen aus dem Stacktrace heraus, aus welcher Klasse sie aufgerufen wurden und schreiben den Output in eine entsprechend benannte Datei.

Beispiel: kickofftug.system.worldmodel.WorldModel_1.txt

Die Nummer nach der Klassenbezeichnung ist die Spielernummer, um bei mehreren aktiven Agenten den Output unterscheiden zu können. Vor den übergebenen String wird ein eindeutiger Index gestellt, der bei jedem Aufruf automatisch erhöht wird, um bei mehreren Dateien die Reihenfolge der Outputs eindeutig festlegen zu können.

11.2 CircularSector

Diese Klasse wird verwendet, um einen Kreissektor (Figure 11)zu beschreiben. Sie hält hierfür die Member **mid_direction_**, **angle_**, **origin_** und **radius_**. Ersteres steht für die zentrale Richtung des Sektors, **angle_** für den gesamten Winkel, **origin_** ist der Mittelpunkt und **radius_** der Radius des Kreissektors. Die Methode **contains(Cartesian coord)** beantwortet die Frage, ob sich ein Punkt innerhalb des Sektors befindet.

Der CircularSector hilft in zahlreichen Klassen und Methoden aus. Viele Problemstellungen in den Conditions können mit Hilfe von Kreissektoren gelöst werden, der Partikelfilter arbeitet mit der von RingSector (abgeleitet von CircularSector) abgeleiteten Klasse Area und das TurnNeckModel benötigt die Klasse Slice, die wiederum von CircularSector abgeleitet ist.

11.2.1 RingSector

Der Ringsektor ist dem Kreissektor sehr ähnlich. Der einzige zusätzliche Member ist **min_radius_**, der die minimale Entfernung von **origin_** angibt. Die contains-Abfrage musste demnach überschrieben werden.

11.2.2 Area

Area (Figure 12) ist eine Spezialisierung des Ringsektors für den Partikelfilter. Die Klasse bietet einen praktischen Konstruktor, der die Berechnung des Kreissektors erleichtert. Er benötigt lediglich eine Flag und die absolute Richtung in die unser Agent blickt.

Die absolute Richtung der Flag wird berechnet und um 180° rotiert, sodass die Flag der Ursprung des Ringsektors sein kann. Maximale und minimale

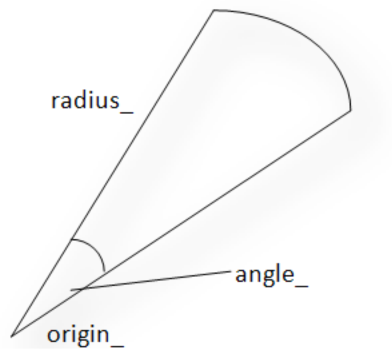

Figure 11: CircularSector

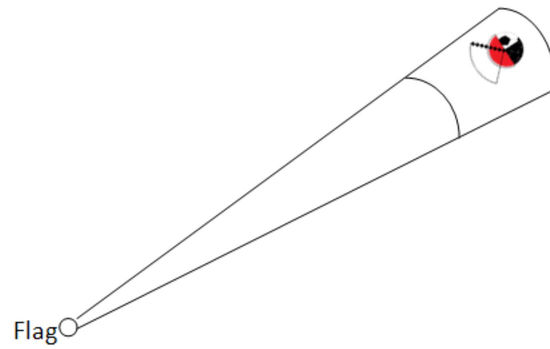

Figure 12: Area

Distanz werden automatisch berechnet[19]. Der Winkel muss nicht dynamisch berechnet werden, da der Fehler hier nicht von der Distanz zur Flag abhängig ist. Er bleibt immer 2°.

[19]Beschreibung im Abschnitt **Grundlegendes zum Problem der Positionsbestimmung**, S.11

12 TurnNeckModel

Besonders seitdem wir den Blickwinkel unseres Agenten auf 45°-90° umgestellt haben, tauchte des öfteren das Problem auf, dass wir ohne Veränderung der Blickrichtung relativ zum Körper zu wenig Informationen erhielten oder schlicht und einfach den Ball aus dem Blickfeld verloren haben, weil wir nicht gleichzeitig zum Intercept-Punkt dashen und ihn im Auge behalten konnten. Um solche Probleme in den Griff zu bekommen, wurde das TurnNeckModel (TNM) geschaffen, das für den jeweils nächsten Zyklus die optimale Blickrichtung errechnet.

Das folgende Konzept ist dem Dainamite Framework nachempfunden und kann im TDP des Teams [E+06] nachgelesen werden.

Herzstück des TNM sind die sogenannten Slices, in die das gesamte Umfeld des Agenten unterteilt wird. Slices sind lediglich Kreissektoren mit zusätzlicher Funktionalität, wie zum Beispiel der Verwaltung einer zuweisbaren Gewichtung. Zur Zeit arbeiten wir mit 6 Slices, diese Zahl läßt sich jedoch leicht verändern, da alle notwendigen Berechnungen darauf Rücksicht nehmen.

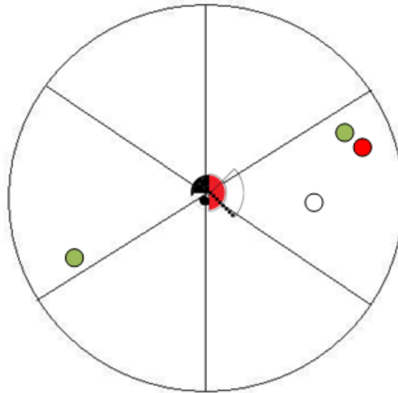

Figure 13: Slices

Zu Beginn eines TNM Durchlaufs werden die Mittelpunkte aller Slices auf die aktuelle Spielerposition gesetzt. Anschließend wird errechnet, wie groß der Blickwinkel im nächsten Zyklus sein wird. Somit erfahren wir auch schon, welche Bereiche wir mittels TNM erreichen können und welche nicht. Wir werden auf jeden Fall in der Lage sein, Objekte innerhalb eines Winkels von

$$\alpha_{vis} = nextangle + 180° \tag{11}$$

einzusehen. Die Ränder des einsehbaren Sektors sind somit auch gegeben:

$$edge_{ccw} = dir_{body} - \frac{\alpha_{vis}}{2} \tag{12}$$

$$edge_{cw} = dir_{body} + \frac{\alpha_{vis}}{2} \tag{13}$$

Nun folgt der aufwändigere Teil des TNM Durchlaufs. Wir prüfen für jedes dynamische Objekt am Spielfeld ab, ob es sich in einem dieser Slices befindet. Die Abfrage wird durch die **contains**-Methode des CircularSector ermöglicht. Trifft das zu, addieren wir einen voreingestellten Wert zum Gewicht des entsprechenden Slice. Maximal- und Minimalgewicht sind im Slice festgelegt, damit keine allzu großen Unterschiede entstehen können. Folgende Gewichtungen können konfiguriert werden:

Listing 1: TNM Konstanten

```
private final double BALL_WEIGHT = 4;
private final double ENEMY_PLAYER_WEIGHT = 2;
private final double OWN_PLAYER_WEIGHT = 1.5;

private final double NO_BALL_WEIGHT = 0.1;
private final double NO_ENEMY_PLAYER_WEIGHT = 0.1;
private final double NO_OWN_PLAYER_WEIGHT = 0.1;
```

Wir verteilen natürlich auch an Slices, die Objekte nicht enthalten eine geringe Gewichtung, damit diese auch von Zeit zu Zeit geprüft werden. Daher brauchen wir die letzten drei Felder.

12.1 Auswahl der Blickrichtung

Es wird in diskreten Schritten (Größe der Schritte ist einstellbar) der Gesamtwinkel, den TurnNeck erlaubt, durchlaufen und für jede Blickrichtung das Gesamtgewicht berechnet.

$$weight_{sum} = \sum_{i=0}^{n} weight(s_i) * overlap(s_v, s_i) \tag{14}$$

Die Funktion **overlap(slice,slice)** liefert die Überlappung zweier Slices als Double Wert im Wertebereich [0.0, 1.0]. Der Bezeichner s_v steht für den einsehbaren Sektor. Die Blickrichtung mit dem größten Gewicht wird als optimale Richtung geliefert.

Die Gewichte aller Slices, die mit dem zukünftigen einsehbaren Sektor überlappen, werden mit $1 - overlap(s_v, s_i)$ multipliziert.

12.2 Overrides

Das TNM muss manchmal ausser Kraft gesetzt werden:

- Wenn der Ball auf Planebene gesucht wird, setzen wir unseren zum Körper relativen Blickwinkel auf 0. Das hat sich bisher als sicherste Variante erwiesen.
- Wenn der Ball in den vergangenen zwei Zyklen nicht gesehen wurde, forcieren wir einen Blick direkt auf den Ball.

- Wenn der Ball sich in unserer Nähe, aber noch nicht innerhalb der visible_distance befindet, sehen wir ihn ebenfalls direkt an, da wir annehmen, dass der Agent versuchen wird, ihn abzufangen. In diesem Fall sollte er ihn so exakt wie möglich wahrnehmen.
- Während wir ein Objekt abzufangen versuchen, sei es der Ball oder ein Gegner, behalten wir es stets im Auge.

Weitere Overrides können, falls notwendig, nachträglich hinzugefügt werden. Die Gewichtungen aller Slices werden nach einem Override zurückgesetzt.

12.3 Resultat

Mit den aktuellen Einstellungen bietet das TurnNeckModel einen guten Überblick über das Feld, während wir den Ball immer im Auge behalten können.

13 Conditions

Die Conditions erlauben den Plänen den Zugang zum Weltmodell. Sie können als Abfragen definiert werden, die ausschließlich boolesche Werte returnieren und eine variable Anzahl an Parametern entgegennehmen können. Bei diesen Parametern handelt es sich zumeist um Abstrahierungen von Objekten am Spielfeld. Jede Abfrage, die in einen Plan eingetragen wird, hat den Aufruf einer oder mehrerer dieser Methoden zur Folge.

Die Conditions sind in vier Klassen gegliedert:

1. ConditionsBase
2. ConditionsMidLevel
3. ConditionsHighLevel
4. ConditionsAbstract

ConditionsMidLevel ist von ConditionsBase abgeleitet und so weiter bis hin zu ConditionsAbstract. Zusätzliche Hilfsklassen sind

- ConditionConst
- Conditions

Die Klasse ConditionConst hält essentielle Konstanten für den Bereich der Conditions und die Klasse Conditions ist die Schnittstelle zwischen dem abstrakten HighLevel und der Conditions-Hierarchie. Nur solche Methoden, die in Conditions vorkommen, können auch in den Plänen verwendet werden.

Die Conditions-Hierarchie enthält ab ConditionsHighLevel aufwärts keine Abfragen zu unserem aktuellen Weltmodell. Hier erhalten sämtliche Methoden ihre benötigten Objekte über Übergabeparameter, um somit Tests zu ermöglichen.

- **self** liefert **true**, wenn der übergebene Spieler der eigene (Self) ist.
- **goalie** liefert **true**, wenn der übergebene Spieler ein Tormann ist.
- Die Methoden
  - atSetupPosition
  - atPosition
  - atGuardPosition
  - atFirstGoalKickPosition
  - atSecondGoalKickPosition
  - closeToStrategicPosition

  prüfen jeweils, ob sich ein übergebenes Objekt an der entsprechenden Position $\pm$ vordefinierte Toleranz befindet.

- **closestTo** und **closestToBall** prüfen, ob sich ein Spieler näher an einem bestimmten Objekt (in letzterem Fall ist das der Ball) befindet als alle anderen Spieler des eigenen Teams.
- **hasBall** prüft, ob der Ball innerhalb der MaxKickDistance[20] des übergebenen Spielers liegt. Sonderformen dieser Condition sind **enemyTeamHasBall** und **ownTeamHasBall**.
- Conditions der Art **withinDistance** prüfen, ob die Distanz zwischen zwei gegebenen Punkten geringer ist als eine durch eine Konstante vorgegebene Distanz entsprechend der Abfrage/Situation. Solche sind z.B.
  - withinScoreGoalDistance
  - withinPassDistance
  - withinTackleDistance
  - withinDirectCoverDistance
  - withinMaxCoverDistance
- Die Methoden
  - beforeKickOff
  - kickOffOwn
  - kickOffEnemy
  - kickOff
  - kickInOwn
  - kickInEnemy
  - cornerKickOwn
  - cornerKickEnemy
  - goalKickOwn
  - goalKickEnemy
  - freeKickOwn
  - freeKickEnemy

  fragen die entsprechenden Standardsituationen ab.

13.1 Dribble

Um zu prüfen, ob der Agent in einer günstigen Lage ist, um den Ball zu führen, ruft man eine der Abfragen **freeDribbleSafeTarget**, **freeDribbleFastTarget** oder **freeDribbleSlowTarget** auf. Der Agent prüft in diesem Fall, ob er sich den Ball vorlegen könnte, ohne dass dieser von einem Gegner abgefangen wird. Dazu wird in einem kreisförmigen Sektor, dessen Zentrum die Position des Balls nach der Dribble-Aktion wäre, überprüft, ob ein gegnerischer Spieler vorhanden ist.

[20] Vom Server vorgegebener Parameter

13.2 FastestTo

FastestTo kann für jedes beliebige Objekt am Feld die Frage beantworten, welcher Spieler es am schnellsten erreichen kann. Dazu wurde eine eigene Klasse angelegt, die die nötigen Berechnungen in jedem Zyklus, in dem eine FastestTo Abfrage auftritt, für alle Spieler zu einem bestimmten Objekt ausführt. Für jedes mögliche Objekt wird eine Instanz der Klasse FastestTo in FastestToManagement (Teil von ConditionsBase) angelegt. Dieses Modul überprüft zugleich auch die Aktualität der Rechenergebnisse[21]. Über diese Klasse laufen nun sämtliche Abfragen:

- fastestTo
- fastestToBall
- enemyTeamFastestToBall
- ownTeamFastestToBall

13.3 InFrontOf

Diese Conditions prüfen, ob das erste übergebene Objekt näher zum gegnerischen Tor ist als das zweite. Wir fragen hierbei nur ab, ob die X-Koordinate des ersten Objekts größer ist als die des zweiten. Eine zusätzliche Abfrage, auf welcher Seite wir spielen, ist nicht notwendig, da immer ein Spiel auf der linken Seite vorgetäuscht wird.

13.4 Unguarded

Wir möchten irgendwie überprüfen können, ob ein Spieler gedeckt wird oder nicht. Das ist sowohl bei eigenem Ballbesitz wichtig als auch beim Deckungsspiel. Die Überprüfung besteht daraus, zu entscheiden ob der besagte Spieler zum gegnerischen oder zum eigenen Team gehört und dann zu prüfen ob sich einer der Spieler des jeweils anderen Teams innerhalb einer Distanz kleiner gleich GUARDING_RADIUS zu diesem Spieler befindet.

13.5 PointTo

Unsere Agenten erhalten vom Server die Möglichkeit, auf Objekte zu zeigen. Diese Condition prüft nun ab, ob so ein PointTo erfolgt ist. Dazu wird ein Spieler und ein Objekt übergeben, deren Status zur Zeit der letzten See-Message zur Berechnung herangezogen wird. Es wird ein Kreissektor (Mittelpunkt ist der zeigende Spieler) erstellt und geprüft ob sich das besagte Objekt in diesem Sektor befindet. Der Kreissektor ist notwendig, um Messfehler kompensieren zu können.

[21] Eine FastestTo Berechnung aus dem vorhergehenden Zyklus ist bereits ungültig.

Figure 14: PointTo

13.6 PassRequested

Es folgt auch gleich eine Anwendung von PointTo, und zwar PassRequested. Wir fordern einen Pass an, indem wir auf den Spieler mit dem Ball zeigen. Dieser erkennt das über diese Condition und kann anschließend entscheiden, ob er den Pass spielen möchte bzw. kann.

13.7 PassInterceptable

Nimmt ein Objekt als Ausgangspunkt und ein weiteres als Endpunkt für einen Pass. Als Ausgangsobjekt wird momentan der Ball angenommen.

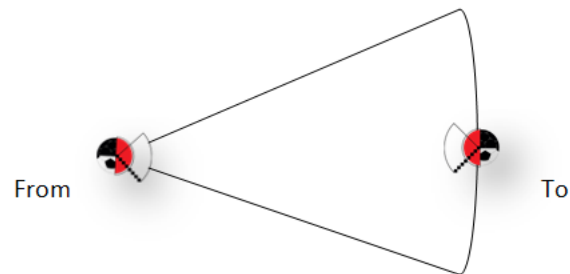

Figure 15: PassInterceptable

Es wird ein Kreissektor wie in der Abbildung konstruiert. Nun muss nur noch geprüft werden, ob sich ein gegnerischer Spieler im Sektor befindet. Eine Verfeinerung dieser Methode war bisher noch nicht notwendig. Bestehende Pläne wären z.B. den Kreissektor nicht gleich beim Ausgangspunkt beginnen zu lassen sondern stattdessen ein schmales Rechteck vorzuspannen, da die Gegner ohnehin nicht so schnell reagieren und den Ball abfangen können. Da unsere Agenten momentan "mutig" genug passen, ist es nicht notwendig den Sektor für einen sicheren Pass einzuengen.

13.8 BallHeadingTowardsGoal

Der Geschwindigkeitsvektor und die Position des Balls werden verwendet, um eine Linie zu zeichnen, die mit der Torlinie geschnitten wird. Existiert ein

Schnittpunkt, wird **true** geliefert, sonst **false**.

13.9 OnBarBallsWayPosition

Es wird die Linie berechnet, auf der sich der Tormann bewegt, wenn er im Tor steht. Diese ist momentan auf 1m vor dem Tor eingestellt. Der Geschwindigkeitsvektor des Balls wird wiederum verlängert und mit dieser Linie geschnitten. Der Schnittpunkt ist die gesuchte Position, auf der der Tormann den Ball abfangen kann. Wenn er sich innerhalb von POSITION_TOLERANCE Metern zu diesem Punkt befindet, wird **true** geliefert, sonst **false**.

13.10 BallCatchable

Prüft ob der Ball sich in der Catchable Area des Tormanns befindet.

13.11 BallConfident

Wir benötigen diese Condition, um auf Planebene mit dem Wissen über die Aktualität des Balls arbeiten zu können. BallConfident prüft, ob der Ball innerhalb der letzten BALL_CONFIDENCE_CYCLES Zyklen gesehen wurde und liefert in diesem Fall **true** zurück. Das ist besonders seit dem Einsatz des TurnNeckModels notwendig, da der Agent oft 1-2 Zyklen lang nicht den Ball sieht und in diesem Fall ohne diese Condition z.B. den Intercept Vorgang abbrechen könnte.

13.12 InOffside

Die Berechnung der Abseitsgrenze beginnt bei der Mittellinie und wird jeweils weitergesetzt, wenn ein Spieler aus dem gegnerischen Team sich näher zu seinem eigenen Tor befindet, als die bisherige Grenze. So wird über alle bekannten Spieler iteriert. Eine Berechnung der Abseitsgrenze ist für beide Seiten möglich, somit können wir sowohl in der Verteidigung als auch im Angriff auf diese Methode zurückgreifen.

13.13 FreePlayer

Diese Methode ist eine Kombination aus **unguarded** und **!passInterceptable**. Da diese beiden Abfragen oft zusammen auftauchen, haben wir sie miteinander verbunden, um einerseits die Pläne übersichtlicher zu machen und andererseits die Ausführung zu beschleunigen.

13.14 OnLine

Prüft ob sich ein Spieler in der Nähe der direkten Linie zwischen zwei anderen Spielern befindet. Wir verwenden diese Methode, um uns auf eine günstige Position zum Abfangen eines Passes zu begeben.

13.15 ResponsibleToCover

ResponsibleToCover hilft uns zu entscheiden, welcher Spieler welchen Gegenspieler decken soll. Diese Frage ist auch im realen Fußball schwer entscheidbar. Hier wird das Unterfangen dadurch erschwert, dass wir nicht davon ausgehen können, jederzeit zu allen Spielern Sichtkontakt zu haben, weder zu unseren, noch zu denen des Gegners.

Die Methode lässt sich folgendermaßen erklären: Wir fühlen uns verantwortlich, den Gegner zu decken, wenn keiner der eigenen Spieler näher zum Ziel ist[22] und wenn wir selbst nicht am nächsten zum Ball sind.

[22] Falls wir vermuten, dass dieser versuchen wird, den Ball abzufangen, ignorieren wir ihn

14 Zusammenfassung und Ausblick

14.1 Projektaufwand

Eine eigenständige Aufstellung der Arbeitszeit ist bis zum 16. Januar 2007 verfügbar[23]. Der Gesamtaufwand betrug zu diesem Zeitpunkt rund 340 Stunden. Seither wird die Arbeitszeit im internen Projektmanagementsystem über Tickets verwaltet. Der gesamte Aufwand beträgt inklusive der Tickets nach dem 16. Januar rund 600 Stunden.

14.2 Ausblick

Kurzfristig wären Projekte wie das Herausfiltern des Fehlers in der Bestimmung der eigenen Geschwindigkeit[24] und das Inkludieren der Geschwindigkeitsabschätzung in den Partikelfilter anzusetzen. Alternativ zu letzterem wäre Analyse und Umsetzung der in [KBH01][25] beschriebenen, auf einem KalmanFilter basierenden Methode zur Positions- und Richtungsbestimmung möglich, aufgrund des überzeugenden Verhältnisses zwischen Qualität und Laufzeit.

Langfristige Projekte am Gesamtsystem umfassen unter anderem Opponent Modeling, um erstens Positionen von Gegenspielern besser vorhersagen zu können und zweitens eine Möglichkeit zu schaffen, das Verhaltensmuster der gegnerischen Mannschaft zu analysieren.

14.3 Zusammenfassung

Im Laufe der Arbeit wurden sämtliche relevanten Änderungen am Weltmodell des Agenten und an Komponenten, die damit unmittelbar in Verbindung stehen, vorgestellt. Es wurde auf die Verbesserung der Positionsbestimmung und der Wahrnehmung des Balls eingegangen und mithilfe unseres Diagnosetools das Resultat bewiesen. Sämtliche Komponenten, die dazu beitragen, dass unser Abbild der Umwelt möglichst vollständig und aktuell ist, wie das restrukturierte Weltmodell mit Erinnerungsmodell und das TurnNeckModel, wurden genau beschrieben. Eine zur Zeit vollständige Auflistung der Conditions wurde in Abschnitt 13 geboten.

Insgesamt gewann das Weltmodell unseres Agenten durch dieses Projekt an Übersichtlichkeit und Genauigkeit. Ein Großteil der bestehenden Methoden wurde geprüft, die ungefilterte Positionsbestimmung durch wahlweise eine Berechnung mittels KalmanFilter oder Partikelfilter ersetzt und die Basis für die Filterung von Richtungsinformationen und Sensordaten zu beweglichen Objekten geschaffen.

[23] **/thesis/Aufwand.txt**
[24] Anschließender Einsatz des BallFilters
[25] **/related/2004_Kyrylov_OptimizingPrecision.pdf**

List of Figures

1 WorldState 7
2 Positionsbestimmung 12
3 Partikelfilter 13
4 DirFilter 15
5 Positionsbestimmung (KalmanFilter / 90° Winkel) 19
6 Positionsbestimmung (KalmanFilter / überwiegend 45° Winkel) 20
7 Positionsbestimmung (Partikelfilter) 20
8 Ballposition 21
9 Ballgeschwindigkeit 22
10 Interception 24
11 CircularSector 26
12 Area 26
13 Slices 28
14 PointTo 34
15 PassInterceptable 34

References

[CFH+02] Mao Chey, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, and Pat Riley. Robocup soccer server. *The RoboCup Federation*, 2002.

[dBK02] Remco de Boer and Jelle Kok. The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team. Master's thesis, University of Amsterdam, The Netherlands, February 2002.

[E+06] Holger Endert et al. *The Dainamite Agent Framework*. DAI Labor, TU Berlin, November 2006.

[KBH01] Vadim Kyrylov, David Brokenshire, and Eddie Hou. Optimizing precision of self-localization in the simulated robotics soccer. *Simon Fraser University - Surrey*, 2001.

[May99] Peter S. Maybeck. Stochastic models, estimation, and control, vol. 1. *ACADEMIC PRESS*, 1999.

[OM02] Oliver Obst and Jan Murray. Qualitative velocity and ball interception. *Fachberichte Informatik, Universität Koblenz - Landau*, 2002.

[SGR05] Monika Schubert, Stephan Gspandl, and Michael Reip. Kickofftug team description paper. *TUG*, 2005.