Thema:

Entwicklung einer RoboCup Soccer-Mannschaft

Ausarbeitung
im Rahmen des Seminars

Agenten in simulierten Umgebungen

am Lehrstuhl für Mathematik und Informatik

Themensteller
und Betreuer: Dr. Dietmar Lammers

vorgelegt von: Nils Miehe
Dirk Toetz
Axel Burkert

Inhaltsverzeichnis

1 Einleitung

1.1 SoccerServer

Der `SoccerServer` ist die Basis der Simulation von Fußballspielen zwischen Mannschaften autonomer Softwareagenten. Die Simulation ist als Client-Server-System angelegt.
Der Server stellt die Simulationsumgebung zur Verfügung und übernimmt dabei die Verwaltung des Weltmodells, bestehend aus einem Spielfeld und den Objekten, die sich auf dem Spielfeld befinden. Auf dem Spielfeld befinden sich der Ball, die Spieler und stationäre Objekte (Flaggen), zur Orientierung der Spieleragenten. Nur der Server verfügt über ein fehlerfreies und umfassendes, also objektives Weltbild. Er nimmt die Aktionen der Spieleragenten entgegen und errechnet die resultierenden Positionen, Richtungen und Geschwindigkeiten der Objekte des Weltmodells. Aus dieser Veränderung des Weltbildes resultiert die Ausführung der Aktionen der Agenten. Gleichzeitig überwacht der Server die Einhaltung der Spielregeln, berechnet die Objektwahrnehmung der Spieler. Entsprechend ihrer Position werden die Spieleragenten vom Server mit Informationen versorgt, die das subjektive Weltbild der Spieler ergeben. Die Spieler führen auf Grundlage dieser Informationen wiederum Berechnungen durch, die zu erneuten Befehlen führen, die sie an den Server übermitteln.
Zusätzlich zu diesen beiden Komponenten existieren noch diverse Monitore, welche auf Wunsch zugeschaltet werden können, um das Spielgeschehen zu visualisieren.

1.2 Aufgabenstellung, Motivation und Ziele

Unsere Aufgabenstellung im Rahmen des Seminars war es, eine lauffähige, möglichst intelligente Mannschaft für die beschriebene Umgebung zu implementieren. Die Wahl der Programmiersprache war dabei nicht vorgegeben, unsere einzige Einschränkung lag in den normalen Beschränkungen, denen die Mannschaften des Robocups unterliegen. Dabei ist vor allem zu nennen, dass die einzelnen Spieler als unabhängige Agenten realisiert werden müssen und demnach nur über den Server miteinander kommunizieren dürfen.
Nachdem wir uns schon zu Beginn des Seminars entschieden hatten, eine eigene Mannschaft programmieren zu wollen statt bereits existierende zu analysieren oder zu verbessern, wollten wir möglichst weitgehend auf bestehende Lösungen verzichten und unserer eigenen Kreativität freien Lauf lassen.
Da uns zu Beginn allerdings jegliche Kenntnisse über die Handhabung der `UDP`-Schnittstelle, die zur Client-Server-Kommunikation genutzt wird, fehlten, griffen wir hierzu auf die Struktur des `SimpleClients` zurück, welcher neben der Kommunikation nur rudimentärste Programmlogik enthielt. Aufbauend auf dieser Struktur wollten wir eine Mannschaft entwickeln, welche nicht nur lauffähig sein sollte, sondern durch ihre Handlungen Intelligenz zumindest erahnen lassen sollte.
Unser Ansporn war dabei neben der grundsätzlichen Affinität zum Thema Fußball vor allem auch Interesse am Thema künstliche Intelligenz und die Herausforderung eines größeren Programmierprojektes an sich.

2 Aufbau des Clients

2.1 Klassendiagramm

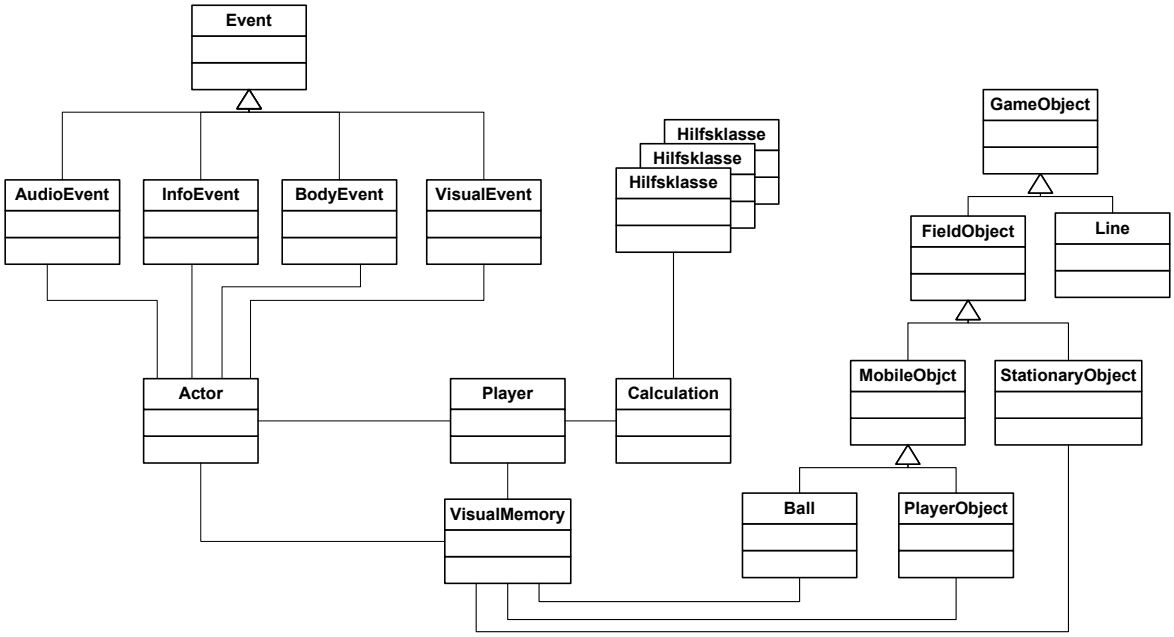

Abbildung 1: *Klassendiagramm unseres Clients*

2.2 Erläuterungen zu den einzelnen Klassen

2.2.1 Player

Die Klasse `Player` ist die Hauptklasse des Clients, sie enthält die `main()`-Methode. In `Player` ist die Logik für das Verhalten des Spielers hinterlegt. Für jeden Spieler wird beim Start seines Threads mit einem Initialisierungsparameter der Spielertyp festgelegt. Durch die Methode `run()`, welche die `start()`-Methode des Threads ersetzt, wird die Entgegennahme der einzelnen Nachrichten verwaltet und entschieden, auf Grund welcher Nachrichten der Spieler handelt. Die Handlung des Spielers findet durch die ebenfalls in `Player` implementierte Funktion `play()` statt. Hierin werden Hilfsfunktionen, meist aus der Klasse `Calculation`, aufgerufen, welche die Entscheidungen des Spielers treffen und die Befehle übermitteln. Zusätzlich enthält die Klasse Attribute wie `ballFocused`, welche für das Verhalten der Spieler von Bedeutung sind und später näher erläutert werden. Der strukturelle Aufbau der Klasse ist das Einzige, was der Implementation dieser Klasse beim `SimpleClient` entspricht

2.2.2 Actor und CommunicationChannel

Die Klasse `Actor` enthält in Verbindung mit der Klasse `CommunicationChannel` die Methoden für die Kommunikation mit dem Server, die es ermöglichen, in den anderen Klassen von den Kommunikationsprotokollen zu abstrahieren. Zum einen werden die Informationen, die der Server für den Spieler bereitstellt, aufbereitet, zum anderen werden hier Methoden zur Verfügung gestellt, die die Aktionen des Spielers protokollgerecht an den Server übermitteln. Diese beiden Klassen wurden unverändert vom `SimpleClient` übernommen.

2.2.3 Event und Unterklassen

Die vom Server empfangenen Nachrichten werden von `Actor` als so genannte `Events` gespeichert. Die Einordnung der Informationen in die verschiedenen `Event`-Klassen findet während der Erzeugung der entsprechenden `Event`-Klasse durch Parsen der Information im Constructor statt. Ein `BodyEvent` enthält Informationen zu Simulationsobjekten in der unmittelbareren Umgebung des Spielers. `VisualEvents` sind Informationen über Simulationsobjekte, die der Spieler „sehen“ kann, also in Abhängigkeit von Blickrichtung und –entfernung wahrnimmt. `AudioEvents` enthalten Zurufe von Mitspielern oder Kommandos des Schiedsrichters. Übermittelte Metadaten werden als `InfoEvents` klassifiziert. Zwar basiert auch diese Eventstruktur auf dem `SimpleClient`, der Großteil der (Parsing-)Logik wurde allerdings von uns hinzugefügt.

2.2.4 GameObject, Unterklassen von GameObjekt und Position

Die verschiedenen Objekte des Weltmodells, welche durch die `VisualEvents` wahrgenommen werden, wurden entsprechend der Objektklassenhierarchie des Servers auch im Client modelliert. Objekte der Klassen `Ball`, `PlayerObject`, `StationaryObject` und `Line` nehmen die wahrgenommenen Informationen über die entsprechenden Simulationsobjekte auf. Sie enthalten jeweils Methoden zur eigenen Identifizierung und zum Setzen und Auslesen der Parameter. Zusätzlich zu den Daten, die durch die `VisualEvents` übermittelt werden, speichern wir zu diesen Objekten auch noch die globale `Position` ab. `Position` ist eine simple Klasse welche neben zwei Koordinaten und deren Gettern und Settern nur noch die Methode `distanceTo()` enthält, welche die Distanz zu einer anderen `Position` berechnet.

2.2.5 VisualMemory

Das Gedächtnis eines Spielers befindet sich in einem Objekt der Klasse `VisualMemory`. Es enthält die vom `Actor` aufbereiteten Daten, die dem Spieler vom Server übermittelt werden. Abgesehen von den `get()`- und `set()`-Methoden zum Auslesen und Speichern der Daten enthält `VisualMemory` nur eine Methode zur Konvertierung von Arrays in Vektoren zwecks einfacherer Weiterverarbeitung der Daten.

2.2.6 VisualInfoUpdate

Um die Informationen in `VisualMemory` durchgehend aktuell zu halten, stellt die Klasse `VisualInfoUpdate` diverse Funktionen zur Verfügung, welche wahrgenommene Informationen weiterverarbeiten oder extrapolieren. Als wichtigste Methode sei hier `orientate()` genannt, welche die wahrscheinliche `Position` des Spielers auf dem Spielfeld bestimmt.

2.2.7 Calculation

Die Klasse `Calculation` enthält diverse Berechnungs-Methoden für die Entscheidung über die Ausführung von Aktionen und für die Erzeugung der Parameter der Aktionen. Dabei bedient sich `Calculation` der im `VisualMemory`-Objekt abgelegten Informationen und neben verschiedener `Hilfsklassen` auch der Klasse `SoccerParameter`.

2.2.8 SoccerParameter

In der Klasse `SoccerParameter` werden Parameter verwaltet, welche zum Justieren unserer Entscheidungsalgorithmen benötigt werden. Damit jederzeit ohne Zeigerübergabe auf

diese Informationen zugegriffen werden kann, haben wir diese Klasse als `final-Class` mit lauter Konstanten implementiert.

2.2.9 Polynom3 und Square- und LinearEquation

Diese drei Klassen werden zur Repräsentation von mathematischen Funktionen benötigt, auf welche in diversen Funktionen der Klasse `Calculation` zurückgegriffen wird. Neben den einzelnen Koeffizienten und deren Zugriffsfunktionen lässt sich durch die Methode `derivation()` von jeder Funktion auch eine Ableitung bilden, welche als Objekt der jeweils untergeordneten Klasse zurückgegeben wird. Außerdem lässt sich der Funktionswert für einen speziellen x-Wert berechnen und die Klasse `SquareEquation` bietet die Möglichkeit der Nullstellenberechnung nach der PQ-Formel an.

2.3 Zeitablauf der Spieleraktivitäten

Unser Hauptaugenmerk bei der Koordinierung der Spieleraktivitäten lag darauf, sie in jedem Zyklus handeln zu lassen, um keine Zeit mit nutzloser Inaktivität zu verschwenden und bei diesen Handlungen stets auf die aktuellsten Handlungen zuzugreifen, so dass sie möglichst angepasst an die aktuelle Situation auf dem Spielfeld reagieren können. Da dies teilweise auch Warten auf später ankommende Informationen beinhaltete, gehen wir nun zuerst auf die Informationen ein, die uns der Server zur Verfügung stellte und zu welchen Zeitpunkten sie bei den Spielern ankommen.

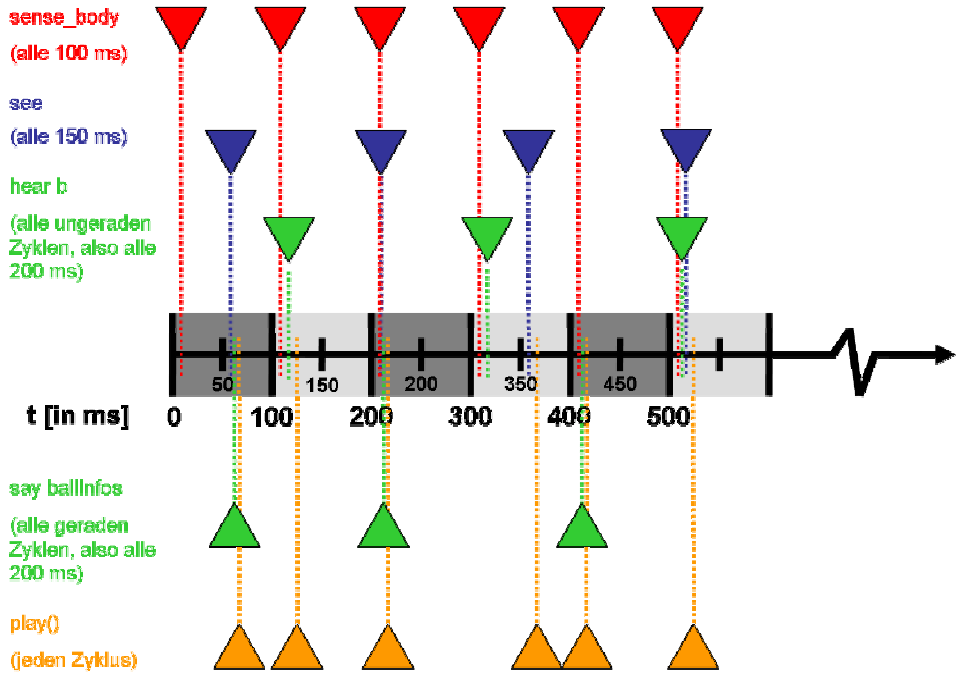

Abbildung 2: *Zeitablauf der Spieleraktivitäten*

Taktgeber des Zeitablaufs stellt die so genannte `BodySensor`-Nachricht dar, welche alle 100ms vom Server verschickt wird und Informationen über die Bewegung des Spielers, Ausdauer und ähnliches enthält. Fast zeitgleich mit dieser Nachricht erhalten die Spieler `say`-Nachrichten, falls im vorhergehenden Zyklus welche verschickt wurden. Für unsere Spieler waren dabei primär die Ballinformationsnachrichten entscheidend, welche alle zwei Zyklen von unseren Mittelfeldspielern versandt werden (dazu später mehr). Als letzte und wahrscheinlich wichtigste Information erhalten die Spieler alle 150ms Nachrichten über die Objekte, die sie aktuell wahrnehmen. Die Rate, mit der diese Informationen empfangen

werden, ließe sich zwar durch Veränderungen der Sichtqualität oder Winkels beeinflussen, wir haben aber auf diese Option verzichtet, da uns ein damit verbundener Informationsverzicht zu teuer erschien. Wichtig ist dabei zu erwähnen, dass diese Nachricht prinzipiell nach der `BodySensor`- und `Hear`-Nachricht empfangen wurde.
Die Handlungen, die unsere Spieler nun ausführen sollten, waren zuerst eine angemessene Verarbeitung der eingehenden Informationen und danach die dadurch beeinflusste Reaktion auf das Spielgeschehen. Da `See`-Nachrichten immer die letzte Nachricht innerhalb einer Runde darstellen, sollten unsere Spieler nach jeder erhaltenen `See`-Nachricht agieren. Da dadurch noch nicht alle Runden abgedeckt wurden, mussten sie außerdem noch nach `BodySensor`- oder `Hear`-Nachrichten handeln. Da wir die Zyklen zu denen wir die `Hear`-Nachrichten erhalten genau bestimmen konnten, handelten wir immer dann nach einer `Hear`-Nachricht, wenn wir wussten, dass auf diese keine See-Nachricht mehr folgen würde. Selbiges gilt für die `BodySensor`-Nachrichten, mit dem Unterschied, dass hier gehandelt wird, wenn weder `Hear`- noch `See`-Nachricht folgen. Im Rahmen der Handlungen der Spieler wird bei den äußeren Mittelfeldspielern zum Abschluss ihrer Aktionen zusätzlich noch die Nachricht mit den Ballinformationen verschickt. Auf diese Art und Weise wird sichergestellt, dass unsere Spieler immer auf Grund der aktuellsten Informationen handeln und auch die weitergegebenen Ballinformationen immer so aktuell wie möglich sind.

3 Taktik und Strategie unserer Soccer-Mannschaft

Bei „realen“ Fußballmannschaften wird die Taktik und die Strategie, neben den zur Verfügung stehenden eigenen Mitteln (in diesem Falle dem „Spielerkader“), meist durch den Gegner und dessen Mitteln bestimmt. Hierbei wird durch die Wahl der Aufstellung eine grundlegende Ausrichtung in Bezug auf Offensiv- und Defensivverhalten vorgegeben. Des Weiteren wird in der Realität manchen Spielern eine Sonderaufgabe mit auf dem Weg gegeben. Als Beispiel für eine Sonderaufgabe kann man die Manndeckung oder Spielmacherrolle nennen.
Bei der Entwicklung einer Soccer-Simulation-Mannschaft hingegen bestimmt in erster Linie nicht der nächste Gegner die Wahl der Aufstellung (wobei man hier anmerken kann, dass es durchaus Mannschaften gibt, die sich an der Aufstellung der Gegner orientieren bzw. deren Aufstellung sich mit dem Spielstand ändert), sondern die eigene Vorstellung, wie die Mannschaft auflaufen soll.

3.1 Spielerpositionen und Zuständigkeitsbereiche

Wir haben uns entschlossen, unseren Gegnern mit einer offensiveren Aufstellung entgegenzutreten, dem 3-4-3[1]-System. Demnach gibt es neben dem Torwart drei verschiedene Spielergruppen: die Abwehrspieler, das Mittelfeld und den Sturm. Die Abwehr hat in erster Linie die Aufgabe gegnerische Angriffe zu vereiteln, während das Mittelfeld bei gegnerischem Ballbesitz schon frühzeitig die Angriffsbemühungen des Gegners zu, unterbinden versucht. Bei eigenem Ballbesitz fällt den vier Mittelfeldspielern die Aufgabe zu als Bindeglied zwischen Abwehr und Sturm zu fungieren. Der Sturm hat die zum Gewinnen essentielle Aufgabe, Tore zu schießen.

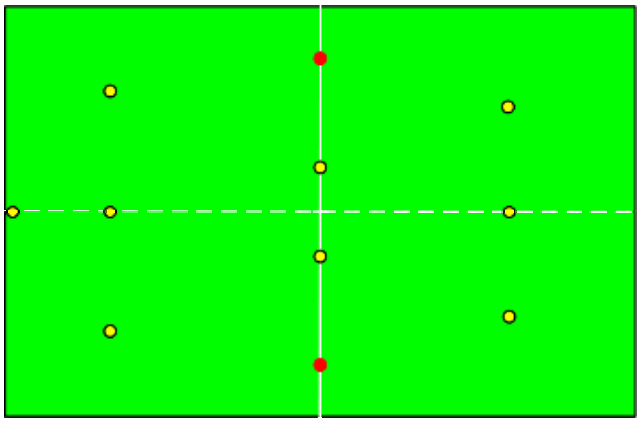

Abbildung 3: *Das von uns verwendete 3-4-3-System.*

Die beiden äußeren Mittelfeldspieler haben die Sonderaufgabe, ständig den Ball im Blick zu behalten (`ballFocused = true`), um somit Mitspielern, die den Ball im letzten Zyklus nicht gesehen haben, mit Hilfe eines Rufes mit relativ aktuellen Informationen zu versorgen. Der Ruf wird über das `say`-Kommando alle geraden Zyklen an den Server übermittelt und erreicht alle Spieler im Wahrnehmungsradius (Standardwert: 50m) zu Beginn des nächsten (ungeraden) Zyklus.

Auf Grund der aktuell gehaltenen Ballposition (sollten keine Informationen über den Ball bekannt sein und sollte auch keine Nachricht der äußeren Mittelfeldspieler eintreffen, kann

[1] 3 Abwehrspieler - 4 Mittelfeldspieler – 3 Stürmer

die aktuelle Ballposition mit Hilfe von alten Positionen, Geschwindigkeitsrichtung und Geschwindigkeit auch extrapoliert werden), verschieben sich die Spieler in den ihnen zugeordneten, strategischen Bereichen. Dazu wird die Ballposition auf dem Spielfeld verwendet um die sog. `BasePosition` des Spielers zu ermitteln. Dabei findet eine Verschiebung der Aufstellung relativ zur Ballposition auf dem Spielfeld statt. Wie oben beschrieben wirkt sich auch der Ballbesitz auf das Verhalten der Spieler aus. Als die am besten zu handhabende Definition haben wir uns entschieden, den Ballbesitz mit der Ballbewegung auf dem Spielfeld gleichzusetzen. Wenn sich der Ball auf das gegnerische Tor bewegt, sind wir im Ballbesitz, bewegt sich der Ball auf unser Tor zu, ist es der Gegner.

Der strategische Bereich eines Spielers ist als sein Hauptaufenthaltsort zu verstehen, deren Grenzen der Spieler nur in besonderen Spielsituationen übertritt, etwa wenn sich der Ball geringfügig außerhalb des Bereiches in erreichbarer Entfernung befindet. Angrenzende, strategische Bereiche überlappen sich um einige Meter, um eine bessere Abdeckung des Spielfeldes zu gewährleisten.

Neben dem strategischen Bereich ist jedem Spieler auch ein eindeutiger, d.h. ein nicht überlappender, über die Außenlinien hinausgehender Zuständigkeitsbereich zugeordnet, in dem er für die Ausführung der verschiedenen Standardsituationen, wie Einwurf, Frei- und Eckstoß, zuständig ist.

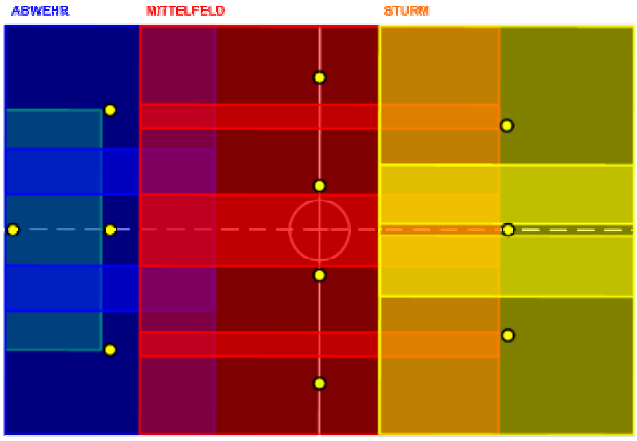

Abbildung 4: *`BasePositions` mit Ballposition (0, 0) und strategische Bereiche der Spieler*

3.2 Darstellung der Außenwelt

3.2.1 Globale Position

Wir haben uns zu Beginn erschlossen, die relativen Informationen über Richtungen und Positionen des Servers in globale Informationen zu überführen.

Für die globale Positionsbestimmung haben wir uns an das vom Server für die Flags verwendete Koordinatensystem angelehnt (Ursprung = Anstoßpunkt). Durch Informationen über Entfernung und Richtung der gesehenen Objekte, haben wir die Möglichkeit, für diese Positionen der Form (x, y) zu bestimmen.

3.2.2 Globale Richtung

Um relative Richtungen zu normieren, haben wir die rechte Torauslinie als 0°-Richtung festgelegt. Schaut ein Spieler parallel zur Außenlinie in Richtung der rechten Torauslinie hat er die globale Blickrichtung 0°, schaut er auf die linke Torauslinie -180° oder +180°.
Zusätzlich ist es nun auch möglich, für andere Objekte eine globale Richtung in Bezug auf unsere eigene Position zu bestimmen. In Abb. 5 werden die globalen Informationen illustriert.

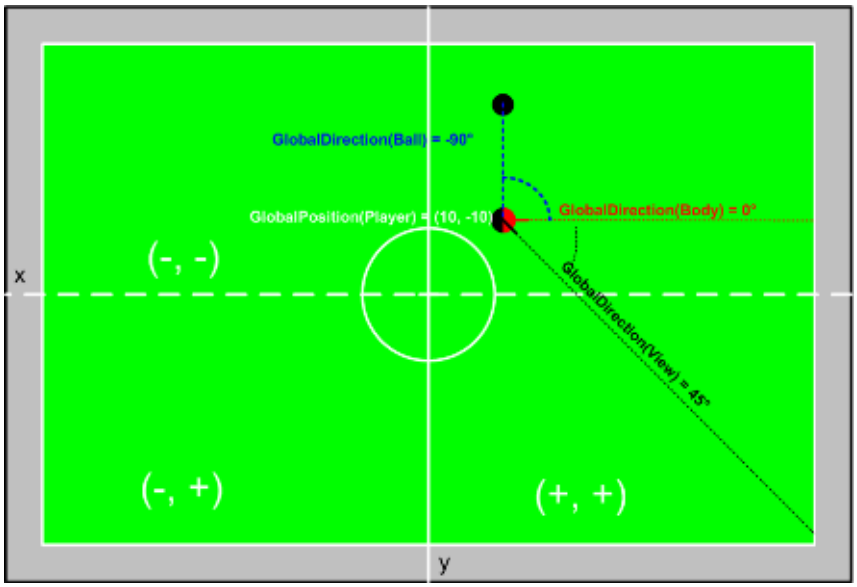

Abbildung 5: *Globale Position und Globale Richtung*

3.3 Offensiv- und Defensivverhalten

In Anlehnung an eine Veröffentlichung von Brünau und Wrede[2], in der ein Gravitationsmodell zur optimalen Positionierung vorgestellt wurde, haben wir zwei verschiedene Modelle entwickelt. Das positive und negative Gravitationsmodell.

Durch diese Modelle bestimmt der Spieler, abhängig von Ballbesitz, Ball- und Spielerpositionen dynamisch eine der Spielsituation angemessene Optimalposition. Einfluss auf die optimale Position haben alle nichtstatischen Objekte (Spieler und Ball) in der festzulegenden Umgebung eines Spielers, sowie die oben genannte `BasePosition`. Eine besondere Rolle kommt auch hier wieder dem Ball zu. Er ist in einer bestimmten Distanz (z. Z. 5 m) immer höher priorisiert als alle anderen Komponenten, d.h. befindet sich der Ball nah am Spieler, ist die - ggf. zukünftige - Ballposition die optimale Position des Spielers.
Anhand des oben definierten Ballbesitzes wird entschieden, welches der beiden Gravitationsmodelle angewandt wird (Hier wird die Wichtigkeit der äußeren Mittelfeldspieler und deren Aufgabe, die Mitspieler mit aktuellen Ballinformationen zu versorgen, nochmals deutlich.):
Das positive Gravitationsmodells ist für das Defensivverhalten der Abwehrspieler und der Mittelfeldspieler zuständig. Gegnerische Spieler ziehen in diesem Modell an, so dass eine Raumdeckung bzw. Manndeckung zustande kommt. Umgekehrt verhält sich das - im Ballbesitz zum Tragen kommende - negative Gravitationsmodell. Hier wirken auf das Mittelfeld und den Sturm Gegenspieler abstoßend, wodurch ein „Freilauf"-Effekt forciert wird.

Da es sich bei diesen Gravitationsmodellen um die zentralen Komponenten unserer Implementierung handelt, wird darauf in Kapitel 5.2 noch näher eingegangen.

[2] Bünau, P. v.; Wrede, C. v.: Cognitive Robotics Project, RoboCup Simulation League - Abschlussbericht.

3.4 Entscheidungsalgorithmus kick()

Befindet sich der Ball in der `kickable_margin` eines Spielers, werden die Gravitationsmodelle zur optimalen Positionsbestimmung ausgesetzt. Es wird die Methode `kick()` aufgerufen, welche einen Entscheidungsalgorithmus beinhaltet, durch den bestimmt wird, was der Spieler mit dem Ball macht. Als Alternativen stehen ihm hierbei der Torschuss, der Pass oder das Dribbling in genau dieser Reihenfolge zur Verfügung. Wir ziehen das Passspiel dem Dribbling vor, da durch einen Pass größere Distanzen überwunden werden können (der Ball ist schneller als die Spieler).

3.4.1 Torschuss

Steht ein Spieler nahe am Tor, d.h. ist seine Position innerhalb einer festgelegten Schussentfernung, schießt der Spieler auf die Winkelhalbierende des größten Winkels zwischen Torwart und Pfosten.

3.4.2 Pass

Ist die Entfernung zum Tor noch zu groß oder wird ein Pass durch den `play_mode` vorgegeben (Standardsituationen), werden umliegende Mitspieler, die näher zum Tor stehen als der Spieler, durch ein Ratingsystem bewertet. Hierbei spielen die Position des Mitspielers (wie nah steht der Spieler an der Außenlinie etc.), sowie der sein Abstand von Gegnern eine Rolle. Ist ein Passpartner gefunden worden, wird der Pass mit Hilfe einer `say`-Nachricht der Form `(say Sendezeit p aimPosX aimPosY Spielernummer)` angekündigt. Hierbei ist Sendezeit der Zyklus, in der der Pass angekündigt worden ist, und `aimPosX` bzw. `aimPosY` die Position, an die der Pass gespielt wird. Zurzeit wird `aimPosX` und `aimPosY` nur dazu verwendet, die serverseitigen Fehler zu umgehen. Denkbar wäre hier aber auch, den Pass in „den freien Raum“ zu spielen und dem Spieler somit implizit den Laufweg anzugeben. Die Spielernummer bestimmt den Passempfänger.
Die `say`-Nachricht wird allerdings nur in ungeraden Zyklen ausgeführt, damit keine Kollision mit den BallInfo-Nachrichten der äußeren Mittelfeldspieler auftritt. Sollte deshalb in diesem Zyklus kein Nachrichtenversand möglich sein, wird der Pass trotzdem ausgeführt, die Nachricht allerdings erst in der nächsten Runde versandt.

Erhält der Passempfänger die Nachricht in der nächsten Runde, hat der die Aufgabe, den Ball im Auge zu behalten (`ballFocused = true`) und ggf. an die angekündigte Passposition zu laufen. Hier wird die oben angekündigte Dynamik des Gravitationsmodells deutlich. Die Ballanziehung hat zur Folge, dass der Passempfänger dem Ball auf den letzten Metern entgegenkommt, wodurch ggf. Ballverluste minimiert werden können.

3.4.3 Dribbeln

Ist kein Torschuss ausgeführt und auch kein optimaler Passpartner gefunden worden, wird in Richtung gegnerisches Tor gedribbelt. Hierzu wird über ein Bewertungssystem eine optimale Richtung herausgefunden, in die ein leichter Schuss ausgeführt wird. Durch das Gravitations-modell wird erreicht, dass der Spieler im nächsten Zyklus wieder vom Ball angezogen wird. Holt er den Ball wieder ein, wird der Entscheidungsalgorithmus von `kick()` erneut angestoßen. Ein Dribbling über eine längere Distanz findet also nur implizit statt, nämlich genau dann, wenn die soeben beschriebene Situation in mehreren aufeinander folgenden Zyklen auftritt.

4 Detaillierte Darstellung ausgewählter Module

4.1 Orientierung: Berechnung der globalen Position

4.1.1 Vorstellung der Umsetzung

Wie bei vielen anderen war auch unser erster Ansatz zur Orientierung die Trigonometrie. Als sich dort jedoch erste Schwierigkeiten bei der Implementierung auftaten, beschlossen wir eine alternative Möglichkeit auszuprobieren, um uns von den anderen Gruppen abzuheben und weil uns die Idee einer ungewöhnlichen Lösung faszinierte.
Unser Ansatz basiert darauf, dass man, wenn man den Abstand eines Spielers von zwei Punkten auf dem Spielfeld kennt, seine Position auf zwei mögliche Punkte einschränken kann, nämlich die Schnittpunkte zweier Kreise um die bekannten Punkte mit dem Radius des jeweiligen Abstands.

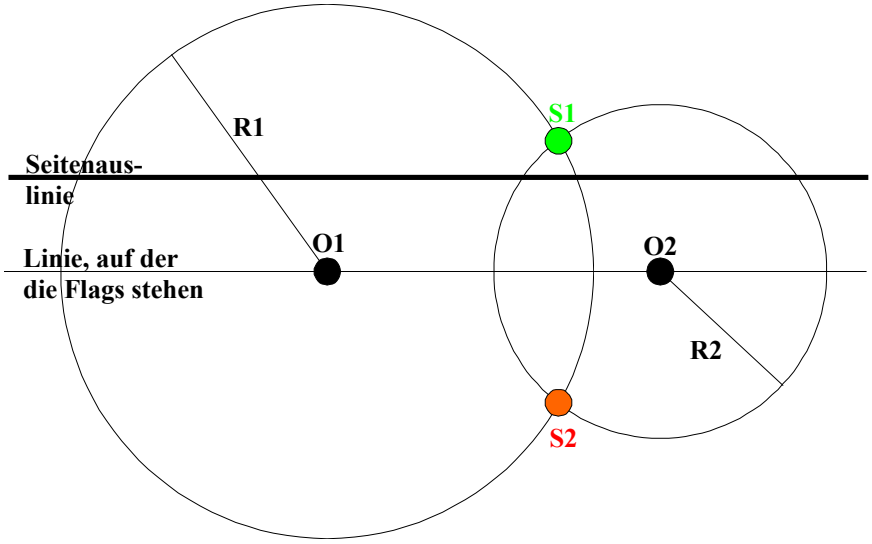

Abbildung 6:
O1: *Objekt 1*
R1: *Distanz zum ersten Objekt*
O2: *Objekt 2*
R2: *Distanz zum zweiten Objekt*
S1*: erster, möglicher Standpunkt*
S2*: zweiter, möglicher Standpunkt*

Die Berechnung dieser beiden Punkte gestaltete sich relativ einfach. Eine Kreisgleichung hat die Form:

$$\left(x - x_{Kreismittelpunkt}\right)^2 + \left(y - y_{Kreismittelpunkt}\right)^2 - r^2 = 0$$

Um nun die Schnittpunkte zweier Kreise zu berechnen, muss man ihre Kreisgleichungen gleichsetzen. Dabei kürzen sich x^2 und y^2 weg, so dass man eine lineare Gleichung mit x und y erhält. Dies ist die Gerade, auf der die beiden Schnittpunkte liegen. Um nun die Schnittpunkte zu erhalten, löst man die Gleichung wahlweise nach y oder x auf und setzt sie in eine der Kreisgleichungen ein. Dabei entsteht eine quadratische Gleichung, deren Nullstellen man berechnen muss. Dadurch erhält man entweder eine der beiden Koordinaten. Die jeweils andere Koordinate erhält man dann zum Beispiel durch Einsetzen des Wertes in die Geradengleichung.
Das einzige Problem bestand nun darin, festzustellen, welcher der beiden Schnittpunkte derjenige ist, auf dem man sich befindet. Dazu griffen wir auf die weiteren Informationen zurück, die wir vom Server erhielten, nämlich den Winkel, in dem unser Spieler die Objekte sah. Durch die Koordinaten eines der Objekte war es uns möglich, für jeden der beiden möglichen Standpunkte die globale Richtung zu ermitteln, in die unser Spieler schaut. Davon ausgehend konnten wir nun an Hand der Abstandsinformationen und des Winkels, in dem der Spieler das zweite Objekt wahrnahm, dessen Position auf dem Spielfeld berechnen. Dies taten wir für beide möglichen Schnittpunkte. Während nun die eine berechnete Position relativ

dicht an der tatsächlichen Position des Objektes lag, war die andere an einer völlig anderen Stelle des Spielfeldes. Wir konnten uns also einfach für den Schnittpunkt entscheiden, dessen Berechnung der Objektposition genauer war.

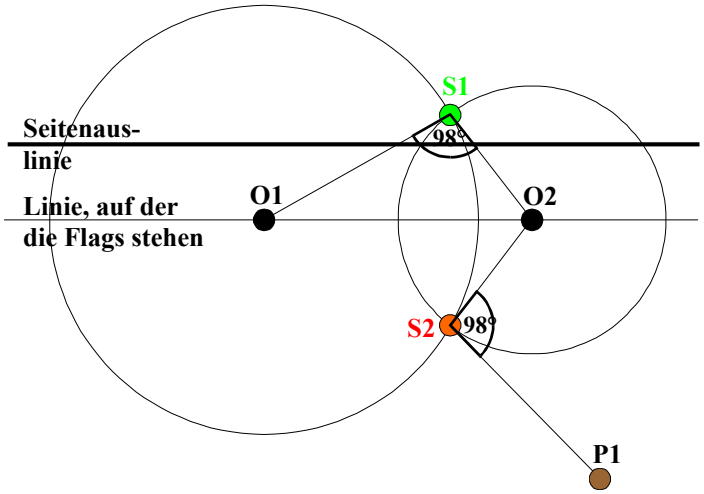

Abbildung 7:
O1: *Objekt 1 und gleichzeitig die von S1 aus berechnete Position von Objekt 1*
O2: *Objekt 2*
S1: *der tatsächliche Standpunkt des Spielers*
S2: *der ausgeschlossene Standpunkt des Spielers*
P1: *die Position, an der sich von Objekt 2 ausgehend Objekt 1 befinden müsste, wenn der Spieler an S2 stünde*

4.1.2 Kritik an der Positionsberechnung durch Kreisschnittpunkte

Leider verursachen die Ungenauigkeiten der übermittelten Informationen bei unseren Berechnungen größere Probleme als wir zuerst annahmen, welche wir zudem erst sehr spät bemerkten. Die Ungenauigkeiten der visuellen Informationen lagen neben gerundeten Gradzahlen vor allem in der Unzuverlässigkeit der Distanzinformationen. Diese sind so beschaffen, dass die übermittelte Distanz oft größer ist als die reale. Berechnet wird diese Ungenauigkeit durch das Aufrunden des Zehnerlogarithmus des Abstands und späteres, erneuten Potenzieren. Dadurch können sich speziell bei größeren Distanzen große Abweichungen ergeben, während es allerdings genauso sein, kann, dass der Wert exakt stimmt.
Dies führte bei unserer Berechnung genau dann zu Problemen, wenn die beiden betrachteten Objekte sehr dicht beieinander lagen. In diesem Fall konnte es vorkommen, dass entweder gar kein Schnittpunkt berechnet werden konnte oder die beiden Schnittpunkte bis zu zehn Meter vom tatsächlichen Standpunkt abwichen. Da unsere Spieler auf Grund der geringeren Ungenauigkeit immer die beiden nächsten Objekte zur Positionsberechnung benutzten, traten diese Probleme sogar relativ häufig auf, meist jedoch mit deutlich geringeren Schwankungen.
Für den Fall, dass keine Schnittpunkte berechnet werden konnten, hatten wir auch schon vorgesorgt, indem wir in dem Fall unsere Position auf Grund des Bewegungsvektors des Spielers, der uns vom Server mit einer gewissen Ungenauigkeit übermittelt wurde, berechneten. Die großen Abweichungen der Schnittpunkte vom realen Punkt fielen uns jedoch viel zu spät auf und konnten auch nicht durch diesen Work-around gelöst werden, da sich die Situation nicht von der normalen Schnittpunktberechnung unterschieden ließ.

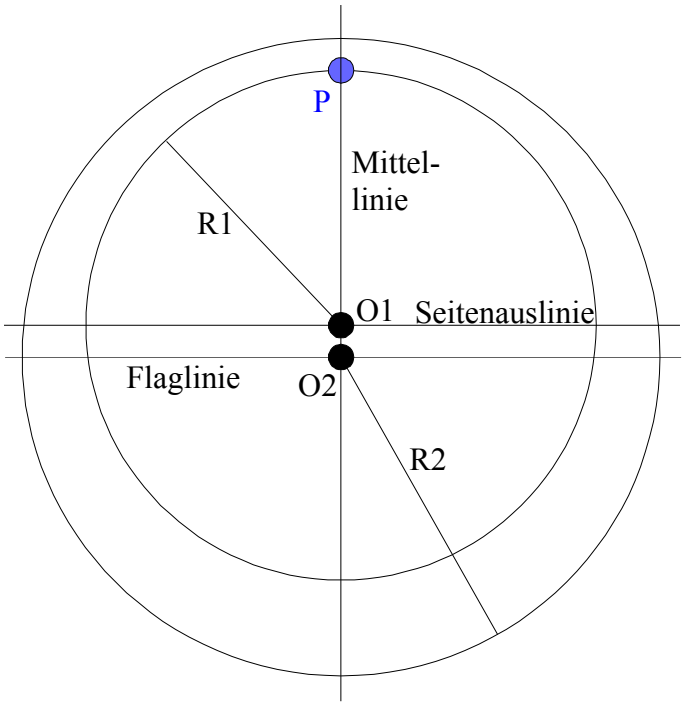

Abbildung 8:
O1: *Objekt 1*
R1: *wahrgenommener Abstand von Objekt 1(korrekte Distanz)*
O2: *Objekt 2*
R2: *wahrgenommener Abstand von Objekt 2 (überschätzte Distanz)*
P: *Position des Spielers (mangels Schnittpunkt nicht zu bestimmen)*

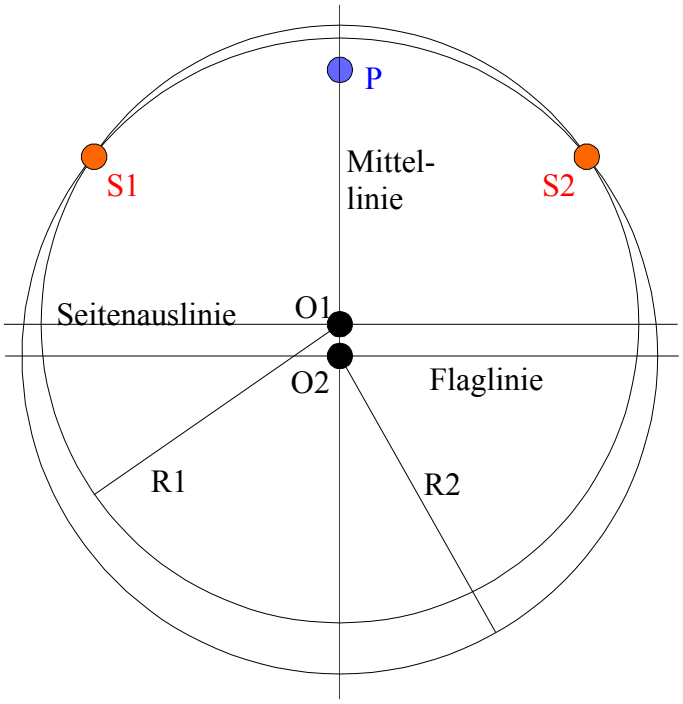

Abbildung 9:
O1: *Objekt 1*
R1: *wahrgenommener Abstand von Objekt 1(überschätzt)*
O2: *Objekt 2*
R2: *wahrgenommener Abstand von Objekt 2(überschätzt)*
S1: *möglicher Standpunkt 1 (große Abweichung vom tatsächlichen Standpunkt)*
S2: *möglicher Standpunkt 2 (große Abweichung vom tatsächlichen Standpunkt)*
P: *tatsächlicher Standpunkt des Spielers*

4.1.3 Verbesserungsmöglichkeiten der Positionsberechnung

Als wichtigster Punkt ist natürlich die Beseitigung der Fehler zu nennen, die im vorhergehenden Abschnitt erläutert wurden. Dies wäre wahrscheinlich schon dadurch machbar, dass man bei der Auswahl der beiden zu benutzenden Objekte prüft, ob die beiden nächsten Objekte zu nah beieinander liegen und in diesem Fall eines der Objekte durch das drittnächste zu ersetzen. Alternativ könnte man eventuell von Beginn an drei Objekte nutzen, paarweise Schnittpunkte berechnen und den Schnittpunkt nehmen, auf den die meisten Berechnungen verweisen.
Als genaueste und komplexeste Weiterentwicklung wäre es sogar denkbar, alle wahrgenommenen Objekte in die Berechnung einfließen zu lassen. Dabei macht man sich zu Nutze, dass man die Formel kennt, nach der die fehlerhafte Komponente in die Entfernung einberechnet wird. Durch das weiter oben beschriebene Abrunden des Logarithmus entsteht die Situation, dass eine Entfernung wie zum Beispiel 15 Meter (dieser Wert ist beliebig gewählt und entspricht wahrscheinlich nicht den realen Werten) für alle Punkte zurückgegeben wird, die zwischen 13,5 und 15 Metern Entfernung liegen. Da sich diese einzelnen Entfernungsbereiche berechnen lassen, kann man von jeder Entfernungsangabe bestimmen, in welchem Bereich die tatsächliche Entfernung liegen muss. Wenn man nun zwei Kreise mit den jeweiligen Bereichsgrenzen als Radius um jedes Objekt zieht, dann ergibt sich durch Überlagerung eine Fläche, in der der tatsächliche Standort des Spielers liegen muss. Dies ist in unseren Augen die genaueste Möglichkeit der Positionsbestimmung, ist allerdings auch mit entsprechend großem Aufwand verbunden.

4.2 Gravitationsmodelle

Für die Positionierung unserer Spieler auf dem Spielfeld wollten wir ein Modul schaffen, welches es ermöglicht, in jeder Spielsituation angemessen zu reagieren, ohne diese vorher erdacht zu haben und durch if-Abfragen zu prüfen, ob sie eingetreten ist. Die Spieler sollten also so durch ihre Umwelt beeinflusst werden, dass sie automatisch die für sie richtige Position einnehmen.
Wie schon erwähnt, sind wir durch einen Abschlussbericht eines anderen Teams auf die Idee eines Gravitationsmodells gebracht worden, bei dem die Spieler durch Gegner, Ball und auch Mitspieler je angezogen oder abgestoßen werden. Da wir nicht einfach blind Ideenklau betreiben wollten, haben wir selbst versucht, verschiedene Ansätze zu entwickeln.
Eine der ersten Ideen war es, die Anziehung oder das Abstoßen durch Kraftvektoren auf die anderen Objekte zu oder von ihnen weg zu realisieren. Hierbei zeigten sich jedoch bald konzeptionelle Schwächen: Für denn Fall, dass zwei Gegner dicht beieinander stehen und beide einen unserer Spieler anziehen, weil dieser in der Verteidigung ist und decken möchte, war es wahrscheinlich, dass bereits einer der Kraftvektoren ausreichen würde, um unseren Spieler komplett anzuziehen, zwei aufaddierte Kraftvektoren aber dazu führen würden, dass unser Spieler quasi über die Position der Gegner „hinausschießt“. Auch für den Fall des Abstoßens, der negativen Gravitation, gab es Schwierigkeiten. Sie sollte ja dazu dienen, dass sich Spieler freilaufen, um für das Empfangen von Pässen möglichst weit von Gegnern entfernt zu sein. Falls jedoch einer unserer Angreifer genau zwischen zwei Gegnern stünde, so würde er sich nicht von der Stelle bewegen, da sich die beiden Vektoren aufheben würden. Dabei wäre es für ihn sinnvoll, sich nicht direkt von den Gegnern zu entfernen, sondern im rechten Winkel zu deren Verbindungslinie.
Diese Probleme waren der Grund, warum wir uns letztlich für zwei verschiedene Modelle entschieden. Für die Abwehr und das defensive Mittelfeld wurde die positive Gravitation mit Hilfe der gewichteten Durchschnittsberechnung realisiert und für den Sturm und das offensive Mittelfeld die negative Gravitation mit einer Methode der maximierten quadrierten Abstände.

4.2.1 Die positive Gravitation durch gewichtete Durchschnitte

Dieses Verfahren sollte sicherstellen, dass unsere Verteidiger Gegner, die in ihre Nähe kommen, automatisch decken, dieses aber beenden, wenn sie sich dabei zu weit von ihrer taktischen Ausgangsposition entfernen. Um dies sicherzustellen, wurden nur Gegner berücksichtigt, die sich innerhalb eines gewissen Gravitationsradius um ihre taktische Ausgangsposition, die `BasePosition`, befinden. Da ausführliche Tests nicht möglich waren, haben wir uns zunächst für einen Radius von 5 Metern entschieden. Außerdem sollte sich unser Spieler, sobald mehr als ein Gegner im Radius ist, auf denjenigen konzentrieren, welcher sich näher an der `BasePosition` befindet, und die Position des anderen nur tendenziell berücksichtigen, solange er weiter von der `BasePosition` entfernt ist. Es war also eine Gewichtung der einzelnen Gegner notwendig.
Da diese Gewichtung von der Entfernung der `BasePosition` abhängen sollte, war es nahe liegend, die Differenz zwischen dem Gravitationsradius und der Entfernung der Gegner von der `BasePosition` zu nehmen. Da dies allein noch nicht zur beschriebenen Konzentration auf einen Gegner führte, quadrierten wir die Differenz und erhielten so die Gewichtung für die einzelnen Gegner. Nun multiplizierten wir die Koordinaten der Gegner mit ihren jeweiligen Gewichtungsfaktoren und summierten sie auf. Die Summe teilten wir wiederum durch die aufsummierten Faktoren und erhielten so gewichtete Durchschnittswerte als unsere optimalen Koordinaten.

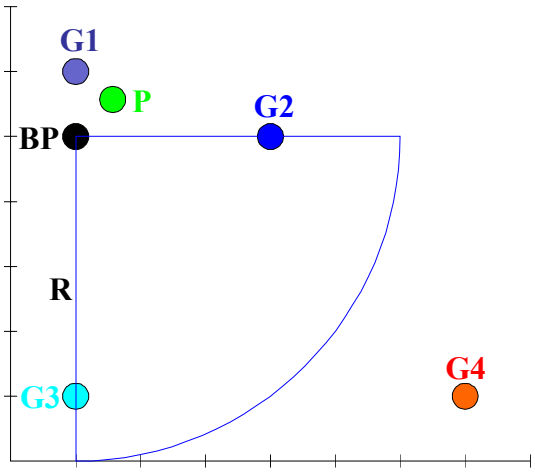

Abbildung 10:
BP: *BasePosition (10; -10)*
R: *Umkreis um die BP, in dem Gegner den Spieler beeinflussen*
G4: *Gegner 4 (16; -6 – beeinflusst nicht)*
G1: *Gegner 1 (10; -11)*
G2: *Gegner 2 (13; -10)*
G3: *Gegner 3 (10; -6)*
P: *optimale Position des Spielers (10,57; -10,57 – ergibt sich aus [(10; -11)*16 + (13; -10)*4 + (10; -6)*1] / (16 + 4 + 1)*

4.2.2 Die negative Gravitation durch quadrierte Abstände

Dieses Verfahren wurde von unseren Stürmern benutzt, um sich möglichst fern von allen Gegnern aufzuhalten und somit anspielbereit für Mitspieler zu sein. Im Gegensatz zur positiven Gravitation war es uns hier nicht möglich, mit gewichteten Durchschnittswerten zu rechnen, da für das Abstoßen ja negative Werte notwendig wären und diese einfach nur einem anderen Spielfeldquadranten entsprechen.
Da es ja das Ziel der Berechnung sein sollte, den Abstand von den Gegnern möglichst groß zu halten, formulierten wir diese Zielsetzung einfach in eine Funktion um und versuchten das Optimum dieser Funktion zu finden. Als Funktion boten sich natürlich die aufaddierten Abstände der Gegner an. Da dies jedoch bedeutet hätte, mit der Wurzel der Summe der quadrierten Koordinatendifferenzen zu rechnen und eine Optimumsberechnung dieser Funktion sehr aufwendig schien, haben wir uns dafür entschieden, mit quadrierten Abständen zu arbeiten.
Wie schon bei der positiven Gravitation sollten uns Gegner, die näher an der `BasePosition` stehen, stärker beeinflussen als Spieler, die weiter entfernt stehen. Ein Problem wurde jedoch direkt bei der Betrachtung der Funktion offensichtlich: Für jede der beiden Variablen entstand eine nach oben geöffnete Parabel, welche nur ein Minimum aufwies und von der wir bestenfalls ein Randmaximum berechnen könnten. Da dies jedoch gleichbedeutend damit wäre, dass sich unsere Spieler in den Spielfeldecken aufhalten würden, mussten wir uns eine Modifikation der Funktion ausdenken, welche es uns ermöglichen würde, ein echtes Optimum auszurechnen und somit unsere Spieler auf dem Spielfeld zu halten.

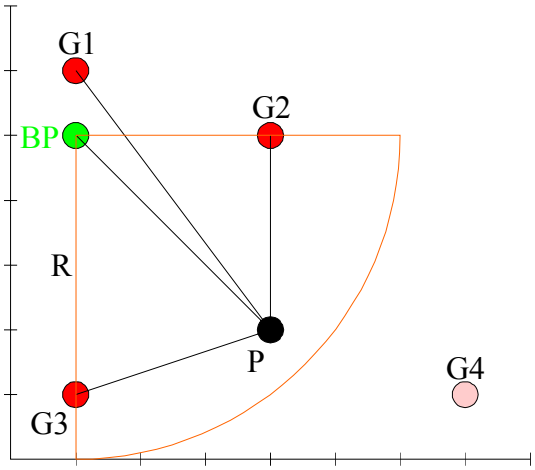

Abbildung 11:
BP: *BasePosition (10; -10)*
R: *Umkreis um die BP, in dem Gegner den Spieler beeinflussen*
G4: Gegner 4 (16; -6 – beeinflusst nicht)
G1: *Gegner 1 (10; -11)*
G2: *Gegner 2 (13; -10)*
G3: *Gegner 3 (10; -6)*
P: *beispielhaft betrachteter Punkt auf dem Spielfeld mit einer Bewertung von 74 (ergibt sich aus*
*4 * (3^2 + 4^2) {Gegner 1}*
*+ 2 * (0^2 + 3^2) {Gegner 2}*
*+ 1 * (3^2 + 1^2) {Gegner 3}*
*1 * (3^3 + 3^3) {BasePosition}*

Hierbei bot sich erneut die `BasePosition` an, von der wir uns nicht zu weit entfernen wollten. Um sicherzustellen, dass diese auf größere Distanzen betrachtet stärker beeinflussen würde als die Gegner, haben wir beschlossen, den Abstand von der `BasePosition` mit drei zu potenzieren, so dass die Gegner mit ihren Gewichtungsfaktoren bei geringerem Abstand unseres Spielers von der `BasePosition` stärker wirken, der negative Abstand von der `BasePosition` aber bei größeren Distanzen stärker beeinflusst. Dadurch taten sich jedoch direkt weitere Schwierigkeiten auf:
Bis jetzt war es uns möglich gewesen, die x- und y-Koordinaten unabhängig voneinander zu optimieren, da sie bei den quadrierten Abständen ohne gegenseitige Beeinflussung in die Formel einflossen. Bei einem mit drei potenzierten Abstand würde dies allerdings der Fall sein, da man die Wurzel der quadrierten Koordinatendifferenzen mit drei potenzieren würde. Um dieses Problem zu umgehen, beschlossen wir, die Koordinatendifferenzen getrennt

voneinander zu potenzieren, in der Hoffnung, dass dies das Ergebnis nicht zu stark verfälschen würde.
Damit blieb nur noch das Problem, dass die Differenzen der Koordinaten je nach Position des Spielers auch negativ sein konnten und durch das Potenzieren mit drei auch negativ blieben. Um das Rechnen mit dem Betrag und die damit verbundenen Probleme der Optimumsbestimmung zu umgehen, beschlossen wir zwei Funktionen aufzustellen. Eine mit negativem Vorzeichen vor dem potenzierten Abstand von der `BasePosition` und eine mit positivem Vorzeichen. Für diese wurden getrennt Optima berechnet, es wurden allerdings nur solche berücksichtigt, die auf der richtigen Seite der `BasePosition` lagen. Falls zum Beispiel die `BasePosition` an der Stelle (0; 0) war und die Koordinatendifferenz mit

$$(Spielerkoordinate - BasePosition - Koordinate)$$

berechnet wurde, so wurden von der Funktion mit negativem Vorzeichen vor dem potenzierten `BasePosition`-Abstand nur Optima berücksichtigt, deren x-Koordinate größer als 0 war.
Für den Fall, dass die Optimumsberechnung mehr als ein Ergebnis lieferte, wurde das Optimum mit dem größeren Funktionswert als beste Position genommen. Falls die Funktion kein Optimum aufwies, wurde einfach die `BasePosition` als optimale Position angesehen.

4.2.3 Kritik an den Gravitationsmodellen

Die Vorteile, wegen derer wir uns für die Gravitationsmodelle entschieden haben, wurden bereits in der Einleitung des Kapitels angesprochen. Nun möchten wir noch auf mögliche Schwächen eingehen.
Ein großes Problem besteht in den mangelnden Informationen über die Positionen der gegnerischen Spieler. Da unsere Spieler jeweils nur einen begrenzten Teil ihres Umkreises wahrnehmen können, sind sie sich zu keinem Zeitpunkt aller Gegner in ihrer Umgebung bewusst und entscheiden sich somit auf Grund von eventuell sehr unvollständigen Informationen. Ein weiteres Problem kann dadurch entstehen, dass ein Spieler in einem Tick Gegner in einer bestimmten Richtung wahrnimmt, und auf Grund seiner Wahrnehmung beschließt, dass seine optimale Position irgendwo in seinem Rücken liegt. Folglich dreht er sich, um im nächsten Tick zu der Position gelangen zu können. Durch die Drehung nimmt er allerdings völlig neue Gegner wahr, durch die er sich womöglich genötigt sieht, seine optimale Position erneut in seinen Rücken zu verlagern…
Wie real dieses Problem ist, konnten wir wegen dem fehlerhaften Zusammenarbeiten unserer Module leider nicht feststellen. Mögliche Gegenmaßnahmen könnten jedoch sein, die wahrgenommenen Spieler länger zu speichern und zu berücksichtigen und eventuell auch Informationen über wahrgenommene Spieler an Mannschaftskollegen weiter zu rufen.

4.3 Ratingsysteme

Angeregt durch die Gravitationsmodelle haben wir versucht, ähnliche Prinzipien auch in anderen Bereichen zum Tragen kommen zu lassen. Dabei eröffneten sich besonders im Bereich der Schussentscheidung vergleichbare Situationen.
Immer, wenn im Spiel einer unserer Spieler feststellte, dass sich der Ball in Schussreichweite befindet, sollte er entscheiden, ob es angebracht ist, den Ball zu stoppen, aufs Tor zu schießen, zu Passen oder zu Dribbeln. Während die Entscheidung des Torschusses durch die Distanz vom Tor und die Entscheidung zu stoppen durch die Ballgeschwindigkeit relativ einfach gemacht wurden, stellten die beiden anderen Bereichen größere Schwierigkeiten dar.
So musste zu Beispiel bei einem Pass entschieden werden, welcher Mitspieler die beste Position hat oder ob eventuell gar nicht gepasst werden soll, da alle Mitspieler zu schlecht positioniert sind. Sehr verwandt dazu war die Entscheidung, in welche Richtung das kurze Vorlegen des Balles geschehen sollte, durch welches wir Dribbeln simulieren wollten.

4.3.1 Anwendung eines Ratingsystems beim Passen zu Mitspielern

Bei der Bewertung der Positionierung der Mitspieler wollten wir mehrere Punkte einfließen lassen. Zum einen sollte er näher an der gegnerischen Torauslinie stehen als der passende Spieler, da wir direktes Spiel auf das gegnerische Tor für sinnvoller hielten als riskantes Querpassen in der eigenen Hälfte. Außerdem sollte der Mitspieler natürlich möglichst weit von gegnerischen Spielern entfernt sein, so dass er den Ball ungestört annehmen kann. Eng damit verwandt war die Entfernung des Mitspielers, welche weder zu groß noch zu klein sein sollte, da bei einem langen Pass die Abfangwahrscheinlichkeit steigt, während bei einem kurzen Pass das Risiko bestünde, dass der Passempfänger zu spät reagiert und deshalb den Pass nicht annehmen kann. Letztlich schienen uns Pässe zu Spielern, die dicht an der Seitenlinie stehen zu riskant, weil in diesem Fall ein nicht korrekt angenommener Ball gleichbedeutend mit Ballbesitz für den Gegner ist.
All diese Faktoren sollten bei der Bewertung der Mitspieler berücksichtigt werden und derjenige Spieler mit der besten Bewertung sollte schließlich als Passempfänger gewählt werden. Da wir Rückpässe von vornherein ausschließen wollten, wurden nur Mitspieler berücksichtigt, die sich mindestens auf gleicher Spielfeldhöhe befinden. Eine Ausnahme zu dieser Regel stellten Anstoß oder andere Standardsituationen dar, wo auch weiter zurückliegende Spieler berücksichtigt wurden. Jeder der in Frage kommenden Spieler wurde nun per `Enumeration` durchgegangen und für jeden der Faktoren mit einem Rating versehen. Da die einzelnen Werte und somit die Gewichtung der Faktoren noch nicht abschließend getestet wurden, werden wir sie hier nicht im Detail vorstellen. Allgemein haben wir den Abstand vom Optimalzustand faktorabhängig linear oder quadratisch einfließen lassen.

4.3.2 Anwendung eines Ratingsystems beim Dribbeln

Auch bei der Entscheidung, in welche Richtung ein Spieler sich den Ball leicht vorlegen sollte, um ihn dann erneut zu spielen, kamen ähnliche Faktoren zum Tragen, bloß dass nun die einzelnen Richtungen statt der einzelnen Mitspieler geratet werden mussten. Wie schon bei den Mitspielern entschieden wir uns auch hier, uns auf die Richtungen zu beschränken, die den Ball näher Richtung gegnerische Grundlinie bringen würden. Da wir Performanceprobleme nicht einschätzen konnten, beschlossen wir, vorerst nur jede zweite Gradzahl zu berücksichtigen, um die Schleife nicht zu lang werden zu lassen.
Als Faktoren kamen auch hier die Nähe zur Außenlinie als negativer Punkt und die Nähe zum gegnerischen Tor als positiver Punkt zum Tragen. Außerdem wurden für jeden gegnerischen Spieler Gradzahlen, die in seine Nähe wiesen umso schlechter bewertet, je näher der Spieler war und je näher die Richtungsgradzahl der seinen war. Die Stellung von Mitspielern wurde

bei der Bewertung nicht berücksichtigt, da wir davon ausgingen, dass der Spieler den Ball sowieso selber erneut spielen würde.

4.4 Berechnung der Schussparameter

Um ein möglichst realistisches Ballverhalten zu simulieren, wirkt sich beim `SoccerServer` die Bewegungsenergie, die der Ball vor einem Schuss hat, auf die nachfolgende Bewegung aus. Genau genommen wird der Bewegungsvektor der vorherigen Restbewegung mit dem Kraftvektor des Schusses addiert und eventuell auf die Länge normiert, die der maximal möglichen Ballgeschwindigkeit entspricht. Nachdem wir durch Vorberechnungen festgestellt haben, in welche Richtung sich der Ball nach unserem Schuss bewegen soll, müssen wir also berechnen, in welche Richtung wir ihn schießen müssen, damit sich die vorherige Bewegung nicht verfälschend auf unsere gewünschte Richtung auswirkt.
Um dies sicherzustellen, greifen wir auf die Vektorrechnung zurück. Damit der Ball nach unserem Schuss einen Bewegungsvektor entsprechend unseren Vorstellungen aufweist, müssen wir ihm zum einen die entsprechende Kraft in die entsprechende Richtung mitgeben und gleichzeitig die vorherige Ballbewegung durch einen Kraftvektor gleicher Länge aber entgegen gesetzter Richtung neutralisieren. Unser Schussvektor ergibt sich demnach aus der Addition des gewünschten Bewegungsvektors und des negativen, vorherigen Ballbewegungsvektors. Dabei kann es durchaus vorkommen, dass die vorherige Ballbewegung der unseren so weit entspricht, dass wir weniger Kraft aufwenden müssen, als wir ohne vorherige Ballbewegung müssten.

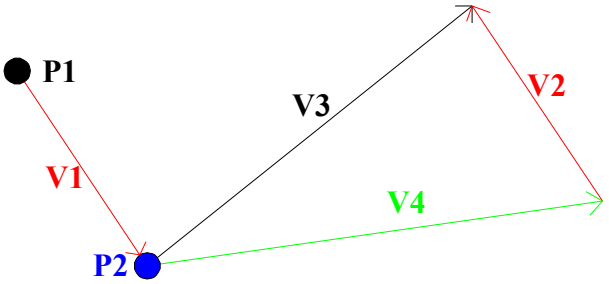

Abbildung 12:
P1: *Ball*
P2: *Spieler*
V1: *Bewegungsvektor des Balls*
V2: *negativer Bewegungsvektor des Balls*
V3: *erforderlicher Schussvektor*
V4: *gewünschter Ballvektor*

Nachdem wir nun den erforderlichen Schussvektor berechnet haben, müssen wir testen, ob unsere Schusskraft ausreicht, um diesen Vektor zu erzeugen. Abhängig von der Entfernung und der Position des Balles in Bezug auf die Körperausrichtung ist die maximal mögliche Schusskraft nämlich im schlimmsten Fall nur halb so groß wie im besten. Um also zu kontrollieren, ob wir den gewünschten Schuss ausführen können, betrachten wir die Länge des benötigten Schussvektors mit unserer maximalen Schusskraft. Ist die Länge des Vektors kürzer, so reicht die Kraft für den Schuss aus. Ist dies nicht der Fall, so muss die Schussrichtung erneut angepasst werden damit trotz der geringeren Schusskraft noch die gewünschte Richtung beibehalten werden kann.

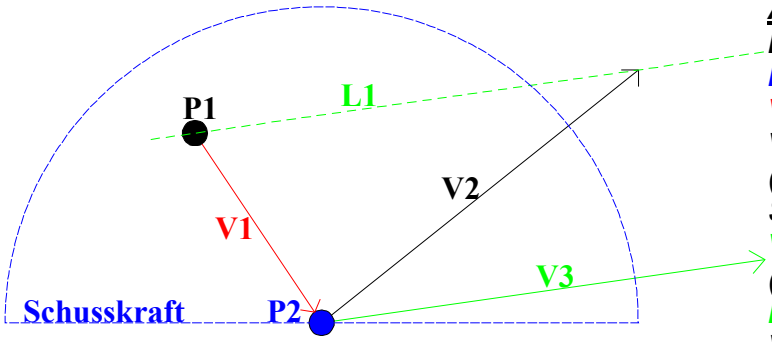

Abbildung 13:
P1: *Ball*
P2: *Spieler*
V1: *Bewegungsvektor des Balls*
V2: *erforderlicher Schussvektor (nicht abgedeckt durch die Schusskraft)*
V3: *gewünschter Ballvektor (nicht realisierbar)*
L1: *Parallele zum gewünschten Vektor durch den Punkt P*

Dazu legen wir an das Ende des umgekehrten Ballbewegungsvektors eine Gerade an, die parallel zu dem gewünschten Schussvektor verläuft. Da sich der Schussvektor ja aus der Additino der umgekehrten Schusskraft und des Wunschvektors zusammensetzt, muss der Endpunkt des Schussvektors auf dieser Geraden liegen. Um diesen Punkt zu ermitteln, greifen wir erneut auf die maximale Schusskraft zurück und ziehen einen Kreis mit dieser Kraft als Radius um den Standpunkt des Spielers. Der Schnittpunkt dieses Kreises mit der Parallelen muss der Zielpunkt des Schussvektors sein. Zwar ergeben sich in der Regel zwei Schnittpunkte, allerdings kann der richtige Schnittpunkt durch die Addition des erhaltenen Schussvektors und des Ballbewegungsvektor kontrolliert werden. Falls der Ergebnisvektor der Addition in die gleiche Richtung weist wie der Wunschvektor, ist dies der richtige Endpunkt des Schussvektors. Dieser Schussvektor kann jetzt benutzt werden, um den Ball mit etwas geringerer als der gewünschten Geschwindigkeit in die richtige Richtung zu befördern.

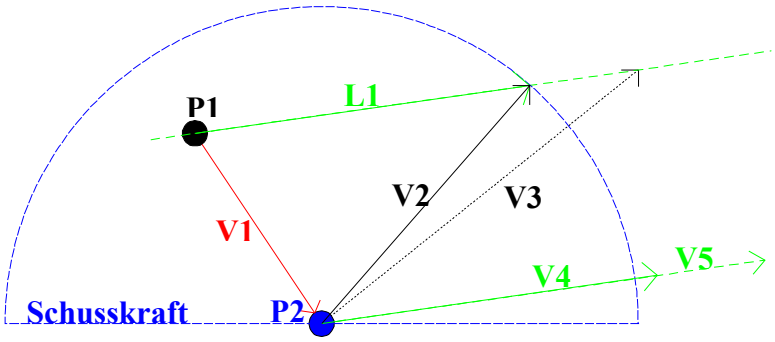

Abbildung 14:
P1: *Ball*
P2: *Spieler*
V1: *Bewegungsvektor des Balls*
V2: *berechneter Schussvektor*
V3: *erforderlicher Schussvektor (nicht abgedeckt durch die Schusskraft)*
V4: *gewünschter Ballvektor (realisierbar)*
V5: *gewünschter Ballvektor (nicht realisierbar)*
L1: *Parallele zum gewünschten Vektor durch den Punkt P*

5 Kritische Betrachtung

Trotz hoher Motivation und großem Arbeitsaufwand ist es uns leider nicht gelungen, eine spielfähige Mannschaft zu programmieren. Im Folgenden wollen wir versuchen, die Ursachen dafür aufzuzeigen und die Konsequenzen, die wir für uns daraus gezogen haben, darzulegen. Neben unseren hausgemachten Problemen gab es allerdings auch Schwierigkeiten, mit denen alle `Soccerer` zu kämpfen hatten, auf die wir zuerst eingehen wollen.

5.1 Kritik am SoccerServer

Unser Hauptproblem bestand in der schweren Zugänglichkeit der Informationen. Während wir uns größtenteils auf die Informationen des Manuals verlassen haben, mussten wir im Verlauf des Projekts feststellen, dass diese Informationen an einigen Stellen nicht vollständig oder sogar widersprüchlich, wenn nicht falsch waren. Leider haben wir trotz Opensource-Aufbaus des `SoccerServers` keine anderen Quellen auftun können, die uns bei Unklarheiten hätten weiterhelfen können.
Eine grundsätzliche Schwierigkeit beim `SoccerServer` besteht darin, dass es als Quasi-Echtzeitsystem nicht mit den herkömmlichen Möglichkeiten zu debuggen ist. Eine Idee, wie sich dieses Problem zumindest abschwächen ließe, wäre, ein schrittweises Zuschalten der einzelnen Komponenten zu ermöglichen. So könnte man zum Beispiel als Option einbauen, dass die Spieler ihre genauen Positionen übermittelt bekommen, so dass andere Module bereits unabhängig vom Orientierungsmodul getestet werden können. Ähnliches ließe sich auch für andere Komponenten wie die Restbewegung des Balls, die die Schussberechnung erschwert, vorstellen. Dadurch könnte man zum einen die Komplexität des `SoccerServers` schrittweise erhöhen und zum anderen das modulare Programmieren im Team erleichtern.

5.2 Selbstkritik

Die eben vorgestellten Schwächen des `SoccerServers` waren aber nicht die Ursache für das Nichterreichen unserer Ziele. Diese ist vielmehr in unserer Unerfahrenheit im projekt-orientierten Programmieren zu suchen.
Schon zu Beginn haben wir den Fehler gemacht, unsere Ziele zu hoch anzusetzen. Auch im Verlauf des Projektes ist es uns nicht gelungen diese zu relativieren, weil wir zu spät bemerkt haben, dass nicht all unsere Vorhaben in der uns zur Verfügung stehenden Zeit umzusetzen waren. Das hing sicher auch damit zusammen, dass wir nur ungern auf eins der von uns entwickelten Konzepte verzichten wollten.
Neben dieser grundsätzlichen Problematik machte sich außerdem unsere mangelnde Erfahrung in der sauberen Planung von größerer Softwarearchitektur bemerkbar. So hätten wir viel mehr Zeit auf die Planung der Schnittstellen unserer Module verwenden müssen, da wir zum Ende unseres Projektes durch die unsaubere Trennung der einzelnen Komponenten in der Fehlersuche stark behindert wurden. Dies wurde sicher auch dadurch verursacht, dass wir auf der Struktur des `SimpleClients` aufgebaut haben und diese nicht frühzeitig besser an unsere Bedürfnisse angepasst haben. Außerdem ließen sich viele Probleme, die im Programmierverlauf auftauchten, zum Zeitpunkt der Planung nicht erahnen, so dass diese Problematik wohl auch bei einer besseren Planung unsererseits nicht gänzlich beseitigt worden wäre.

5.3 Konsequenzen für zukünftige Projekte

Durch unsere Probleme bei der Fertigstellung des Projektes haben wir Einsichten über das Programmieren größerer Programme gewonnen, die uns hoffentlich bei zukünftigen Projekten helfen werden, Schwierigkeiten im Vorfeld zu erkennen und richtig einzuschätzen. So haben

wir feststellen müssen, dass eine gründliche Planung für ein Programmierprojekt essentiell ist und diese auf jeden Fall abgeschlossen sein muss, bevor mit der Erstellung von Code begonnen wird. Bei der Planung sollte vor allem auf eine Trennung der einzelnen Module durch feste Schnittstellen eingegangen werden, so dass die einzelnen Teilnehmer besser unabhängig voneinander arbeiten können und eine gegenseitige Beeinflussung der Module minimiert wird. Des Weiteren sollte durch festgelegte Schnittstellen das getrennte Testen der einzelnen Module besser möglich sein als es bei uns der Fall gewesen ist. Im Programmierverlauf erscheint es im Nachhinein sinnvoll, die Komplexität einzelner Module schrittweise zu erhöhen und dabei auf jeder Entwicklungsebene ein Debuggen durchzuführen.

Zum Abschluss wollten wir noch ein Feedback zu dem Seminar im Allgemeinen geben. Trotz des für uns teilweise frustrierenden Ausgangs hat uns das Seminar viel Spaß gemacht und tiefgehende Einblicke in die Abläufe des projektorientierten Programmierens gegeben. Für zukünftige Seminare dieser Art würden wir aber empfehlen, die Programmierer einer Soccermannschaft frühzeitig auf den Umfang der Aufgabe hinzuweisen, so dass sie nicht der Versuchung erliegen, ihre Ziele zu hoch anzusetzen. Alternativ könnte man eventuell auch mehr Teilnehmer pro Team zulassen, wobei in diesem Fall vielleicht eine Anleitung bei der Architekturplanung nötig wäre.

Literaturverzeichnis

Mao, C.; Foroughi, E. ; Heintz, H.; ZhanXiang, H.; Kapetanakis, S.; Kostiadis, K.; Kummeneje, J.; Noda, I.; Obst, O.; Riley, P.; Steffens, T.; Yi, W.; Xiang, Y.: Users Manual. RoboCup Soccer Server for Soccer Server Version 7.07. and later. http://sserver.sourceforge.net/docs/manual.pdf. Abrufdatum: 2004-10-01.

Bünau, P. v.; Wrede, C. v.: Cognitive Robotics Project, RoboCup Simulation League - Abschlussbericht. http://www.ottermoor.de/paul/CogRob_0304_Buenau_Wrede.ps. Abrufdatum: 2004-10-01.