Dokumentation

Praktikum MPGI3 RoboCup 3D im SoSe 2013

19. Juli 2013

Inhaltsverzeichnis

1 Einleitung

Im Praktikum RoboCup 3D ging es darum Roboter in der Simulation Fußball spielen zu lassen. Dazu wurde die 3D-Simulationsumgebung simspark genutz. Dieses verwendet aktuell das Roboter Modell des NAOs. Dabei ist jeder Roboter auf dem Spielfeld ein eigenständiger Agent, der seine Umgebung wahrnehmen kann, sich bewegen kann und Entscheidungen trifft. Damit ist es möglich elf gegen elf Roboter nach Regeln gegeneinander spielen zu lassen.
Damit die NAOs Fußball spielen, muss erst ein wenig Vorarbeit geleistet werden. Vom Server bekommt jeder Agent übermittelt, was er sieht und Informationen über sich, wie z.B. seine Gelenkstellungen und seine Lage im Raum. Diese müssen dann ausgewertet werden. Mit Hilfe eines Modells der Welt ist es möglich die Position des NAOs zu bestimmten und zu wissen wo der Ball ist. Auch muss eine Möglichkeit gefunden werden, einzelne Körperteile an bestimmte Positionen zu bringen, damit der Roboter z.B. aufstehen kann. Da das Laufen, als sehr schwierig betrachtet wurde, wurde am Anfang des Praktikums entschieden dieses herauszulassen. Wir beamen den Agenten über das Feld, was auch wieder neue Schwierigkeiten, wie die Kollisionsvermeidung mit sich bringt. Zusätzlich ist es auch möglich zu hören, was die Spieler in der näheren Umgebung sagen und selbst was zu sagen. Wie Nachrichten übermittelt werden, war ein weiteres Thema, womit wir uns als Gruppe beschäftigt haben.
Die eben beschrieben Themengebiete habe wir auf Kleingruppen aufgeteilt, die diese bearbeitet haben. Diese Aufteilung haben wir in der Dokumentation beibehalten.

2 Serververbindung

Ein paar Informationen zur Serververbindung bzw. Sockets in Pyhton:
Sockets in Python funktionieren ähnlich wie in Java. Per “import socket“ stehen sie uns zur Verfügung. s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) legt einen TCP Socket mit dem IPv4 Protokol an. Mittels s.connect(ip,port) verbindet man sich mit dem Server. Anschließend kann in einer while-Schleife auf Servernachrichten gewartet werden.

Agent und Server kommunizieren über das TCP Protokoll über den vordefinierten Standardport 3100. Bevor ein Agent am Spiel teilnehmen kann, muss er sich “registrieren“. Das geschieht über zwei Kommandos.
CreateEffector message: (scene <filename>). Dieser Befehl legt die .rsg Datei fest welche der Server für diesen Agent nutzt (enthält physische Repräsentation und effectors/perceptors).
InitEffector message: (init (unum <playernumber>)(teamname <yourteamname>)). Hier gibt der Agent an, zu welchem Team er gehört und welche Spielernummer er hat. Als Spielernummer ist auch 0 möglich - hier weist der Server dem Agent die nächste freie Nummer zu. Sofern wir keine festen Rollen verteilen, dürfte die Spielernummer aber irrelevant sein.
Sofern ich nichts übersehen habe, gibt es keine Rückmeldung vom Server - es gibt also keine Rückmeldung für gesendete Kommandos.

Die Frage wäre nun, wie Komplex das Serververbindungs-Modul sein darf/soll. Stupides Management der Verbindung (aufbauen, aufrechterhalten, senden, empfangen). Also reines Senden und Empfangen von Nachrichten ohne große Verifikationen (außer natürlich sicherzustellen, dass alles vollständig und fehlerfrei gesendet wird/ankommt). Oder darf es doch mehr wie z.B. das Registrieren mit dem Server übernehmen oder nach bestimmten Richtlinien Nachrichten validieren.

3 Datenstrukturen

3.1 To Do

- zu Linien:

- Was passiert, wenn ein Endpunkt nicht sichtbar ist? Bekommen wir einen Punkt am Rand des Sichtfeldes? Vektor...?
- Wir bekommen ***nicht*** die Info, ***welche Linie*** das ist, wenn wir eine sehen. Wenn wir Linien als statische Objekte für unsere Schablone nutzen wollen, müssen wir das selbst rausfinden. Dazu zwei Möglichkeiten:
  * Wir verlassen uns darauf, dass wir immer zusätzlich noch Flags oder Goalpoles sehen und es daraus ableiten können.
  * Bei genügender Confidency über die eigene Position, könnten wir auch diese Information dazu nutzen.

- Gewichtung

3.2 Weltenmodell

Umgesetzt in der world.py
Ziel ist es das Spielfeld intern darzustellen und mit möglichst vielen und genauen Daten zu füllen die dem Agenten später als Grundlage für seine Entscheidungen dienen. Um die Kommunikation zwischen den Naos zu erleichtern haben wir uns für ein statisches Weltbild entschieden. Konkret heißt das, dass wir den Koordinatenursprung im Spielmittelfeld angesiedelt haben und nicht an der Position des Agenten. Dadurch erhalten wir feste Positionen für unsere sogenannten static entities wie die Flagen, Linien und Torpfosten. (Fest meint hier während eines Spiels, die Spielfeldgröße, die sich im Laufe der Entwicklung öfters geändert hat, kann dem Konstruktor der world übergeben werden.)
Das Spielfeld sieht wir folgt aus:

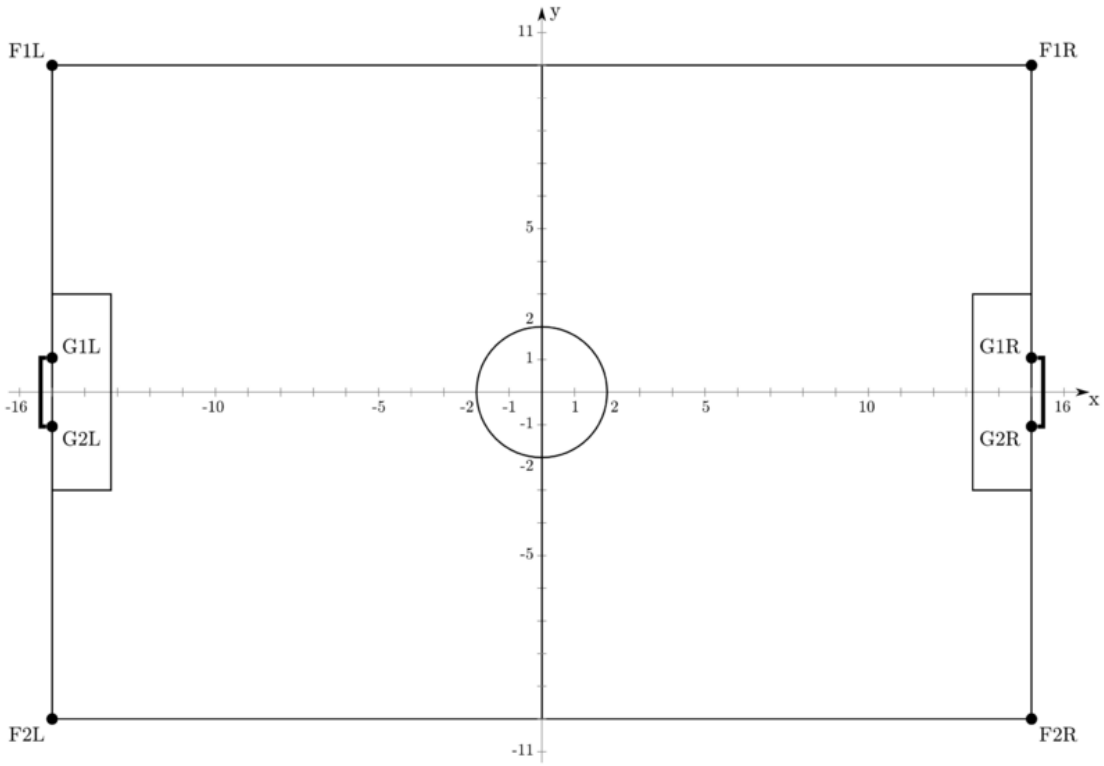

Abbildung 1: Spielfeldmaße

Wichtig dabei zu beachten ist das sowohl die 4 Eckfahnen als auch die 4 Torpfosten eindeutige Namen in der Simulation haben und auch vom Nao unterschieden werden können.

Dazu kommen dann noch bewegliche Objekte (die mobile entities) wie andere Mitspieler und der Ball, deren aktuelle Position permanent neu berechnet werden müssen. Dazu erhalten wir alle 3 Zyklen von der "virtuellen Kamera" des Naos was dieser gerade sieht. Die Daten werden von der *perception.py* verarbeitet und von ihr in die world bzw. die nao Klasse geschrieben.

3.3 Nao

Umgesetzt in der nao.py
Die nao.py hat das Ziel alle Agenten spezifischen Daten zu sammeln. Dazu gehören die eigenen Gelenkstellungen, die momentane Aufgabe, die Aktuelle Lage (also aufrecht, auf dem Rücken ect.)

3.4 Klassenmodell des Gegenstandsbereiches

(Stand 09.06)

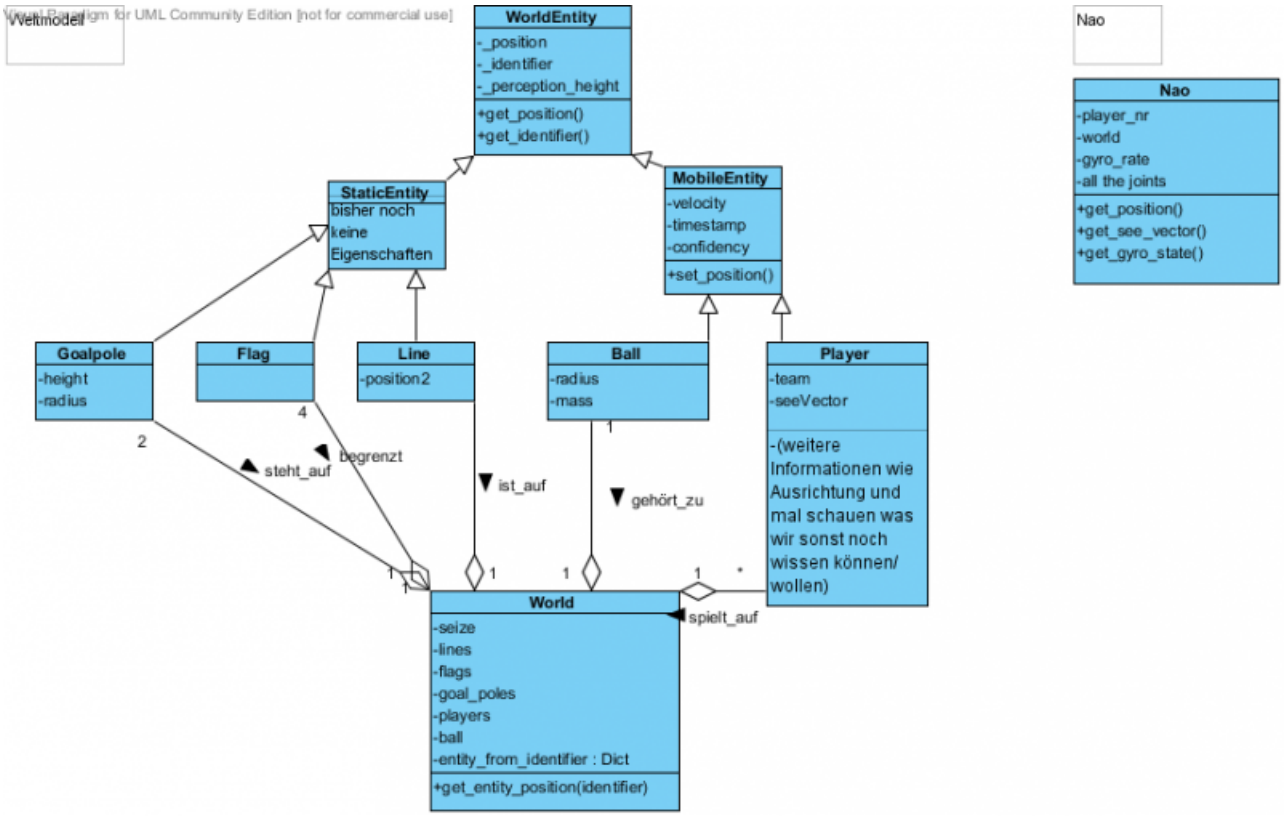

Abbildung 2: Klassenmodell des Gegenstandsbereichs

3.5 Perception

Umgesetzt in perception.py -Hier passiert die ganze Magie :D

3.5.1 Distanzberechnung: 3D-Kugelkoordinaten zu 2D kartesischen

- Wir bekommen vom Server aller Objekte die wir sehen können als Kugelkoordinaten relativ zum Roboterkopf (bzw. zur Kamera). Im allgemeinen interessiert uns allerdings mehr die Distanz zwischen uns und dem Objekt.

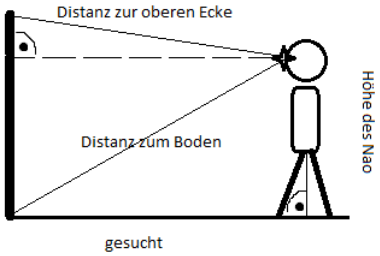

Abbildung 3: Veranschaulichung des Problems

- Glücklicherweise lässt sich das Problem einfach lösen: $\sqrt{(Distanz)^2 - (\texttt{Nao-Augenhoehe})^2}$
- Problem: Bei manchen Objekten liefert uns der Server die Distanz und Winkel zur oberen Ecke (z.B. Goal). Das verkompliziert das Problem aber nur minimal:
- $\sqrt{(Distanz)^2 - (\texttt{Objekthoehe} - \texttt{Nao-Augenhoehe})^2}$
- **WICHTIG! Deshalb sollte man die Positionsbestimmung abschalten wenn man merkt das man umgefallen ist!**

3.5.2 Eigene Position bestimmen

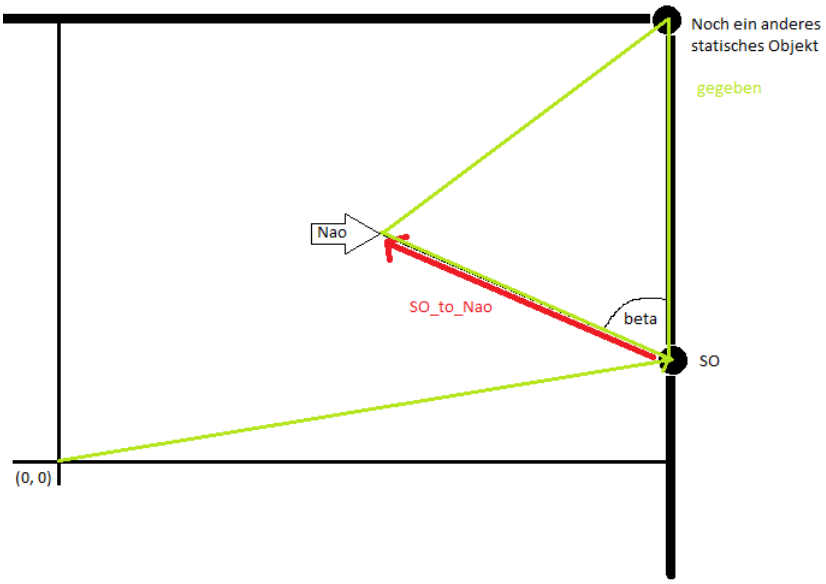

Abbildung 4: Skizze

- Was haben wir?
  - Wir wissen wo sich die Flaggen, Linien und Torpfosten befinden. Bzw. anders formuliert wir kennen die Vektoren zu ihrer Position.
  - Wir bekommen vom Server die Distanz zu diesen Objekten (sofern wir sie denn sehen).
  - Damit haben wir dann zusammen mit der Distanz zwischen den statischen Objekten (die ist immer gleich und uns bekannt) ein hübsches Dreieck.

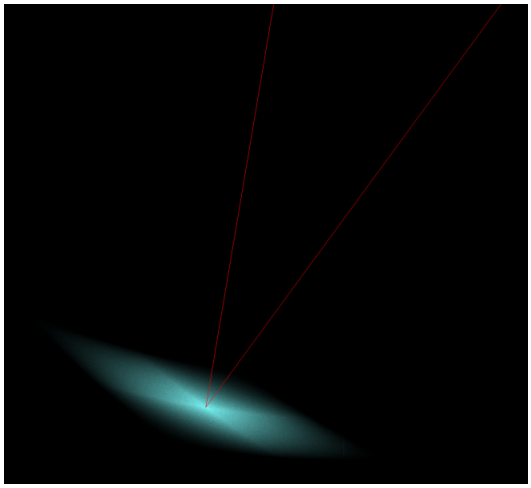

Abbildung 5: Wahrscheinlichkeitsverteilung unserer ermittelten Position

  - **Wir brauchen immer min. 2 statische Objekte im Sichtfeld!**

- Was wollen wir?
  - Den Vektor vom Koordinatenursprung zu unserer Eigenen Position.
- Wie bekommen wir den?
  - 1. Berechne beta an SO (z.B. mit dem Kosinussatz für beliebige Dreiecke). (Siehe Skizze!)
  - 2. Berechne ob SO das linke oder rechte Objekt ist.
    - Wir haben die Winkel zu den Objekten, von daher müssen wir testen welcher größer ist. (Im Sinne von mathematisch positiv bedeutet ein höherer Wert weiter links.)
  - 3. Berechne den Vektor von SO zu unserer eigenen Position. (In der Skizze SO_to_Nao)
    - Die Länge kennen wir ja schon, es fehlt nur noch die Richtung.
    - Nun drehen wir den den Vektor zwischen den statischen Objekten um unser beta *(-1) (im Uhrzeigersinn) wenn es das Linke ist und um beta (gegen den Uhrzeigersinn) wenn es das Rechte ist.
      - (Das ist total abgefahrene Mathemagie, weil man in das Koordinatensystem 2 Dreiecke malen kann die die entsprechende Seitenlängen haben und so auf 2 verschiedene Lösungen kommen könnte. Wir sparen uns also das Rätselraten das wir beim Berechnen der Schnittkreise haben.)
    - Wir passen seine Länge auf unsere gegebene Distanz an.
  - 4. Der Finale Schritt
    - Addiere den Vektor aus 3. zu dem Vektor vom Koordinatenursprung zu SO und glelange endlich ans Ziel :D

3.5.3 Mobile Entitäten lokalisieren

- Gesucht: absolute, kartesische Positionskoordinaten einer beweglichen Entität (`MobileEntity`, also `Ball` oder `Player`)
- Gegeben:
  - die Richtung, in die der NAO guckt als absoluten, kartesischen 3D-Vektor (`see_vector`)
  - eine Angabe in Polarkoordinaten, wo im Blickfeld des NAOs und wie weit entfernt sich die entsprechende Entität befindet (`pol`)

- Lösung:
  - Wir nehmen den `see_vector` und:
    - berechnen aus dessen 2D-Teil (ohne z) einen Winkel `rot2d`
  - Dann rotieren wir `see_vector`:
    - um die z-Achse um `-rot2d` auf die xz-Ebene (so, dass y = 0)
    - um die y-Achse um den vertikalen Winkel in `pol`
    - um die z-Achse um den horizontalen Winkel in `pol` + `rot2d`
  - Wir normieren den entstandenen Vektor.
  - Wir skalieren ihn um die Distanz in `pol` (Entfernung von Kamera zu Objekt).
  - Da isse, die mobile Entität.

3.5.4 Umgang mit Rauschen

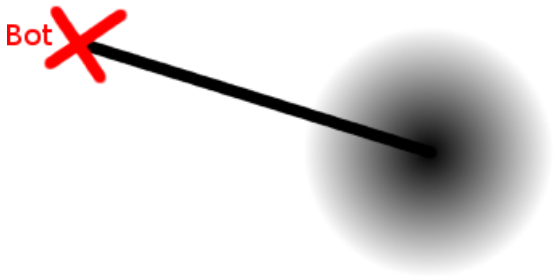

Abbildung 6: Vereinfachte Annahme: Rauschen kreisförmig verteilt

Abbildung 7: Tatsächliche Situation: Winkel- und Distanz-Rauschen

Um (später) die Genauigkeit unserer Positionsbestimmungen zu erhöhen, sollten wir beachten, welche Formen sich uns bei der Betrachtung der Wahrscheinlichkeitsverteilungen präsentieren.
Die einfachere Variante wäre, nach der Positionsbestimmung *ein* einfaches Skalar mit abzuspeichern, das die ungefähre Genauigkeit der Position angibt. Diese lässt sich als Radius einer Kugel / eines Kreises (wenn wir auf 3D verzichten) interpretieren. (obere Abbildung)
Tatsächlich sieht die Verteilung aber eher so aus wie in der unteren Abbildung.

Einfache Positionsbetimmung über Schnittpunkte von Kreisen - Wird im Moment nicht verwendet! Sollte iegentlich zur Positionsbestimmung genutzt werden, wurde aber durch einen besseren Ansatz verdrängt. Allerdings könnte es zur Berechnung der erwarteten Abweichung noch sehr nützlich sein, von daher erstmal behalten!

- 1. Wir definieren um 2 statische Objekte Kreise mit ihrer Position als Mittelpunkt und dem Abstand zum Agenten als Radius.
  - Z.B. $k = (x - x_M)^2 + (y - y_M)^2 - Distanz^2 = 0$
- 2. Wir berechnen die beiden Schnittpunkte der Kreise.

  - 1. $k_1 - k_2$
  - 2. nach y umstellen ¿ eine Gerade c: m * x + n (die nennt man übrigens Chordale)
  - 3. Setze c in die Kreisgleichung ein. ¿ 2 Ergebnisse
  - 4. Setze die aufgelöste Variable in c ein und erhalte die beiden zugehörigen Koordinatenpaare.

- Um zu bestimmen an welche Position wir uns nun tatsächlich befinden würde ich vorschlagen das wir zu uns zu jedem Zeitpunkt merken ob wir im Spielfeld sind oder nicht. Da all unsere statischen Objekte am Spielfeldrand liegen wäre eine Position immer im und die andere außerhalb des Spielfeldes.
  - Ist es möglich zu jedem Zeitpunkt zu wissen wo ob man im Spielfeld ist oder nicht?
- können wir uns vielleicht irgendwie berechnen in welche Richtung wir gerade schauen und dann überlegen welches der beiden Objekte links bzw. rechts ein müsste wenn wir im Spielfeld sind?
- Wir könnten auch unsere zu letzt bekannte Position mit einbeziehen.
  - kann man ohne es zu wissen plötzlich außerhalb des Spielfelds landen?
- Vorteile:
  - sehr schnell
  - kommt ohne den Winkel phi aus -¿ man spart sich einen weiteren verrauschten Parameter

4 Fähigkeiten

4.1 Allgemeines

Bevor angefangen wurde, Bewegungen für die viertuellen NAOs zu programmieren, wurde überlegt, welche Fähigkeiten in der 3d-Simulation benötigt werden könnten. Die Bewegungn wurden im vorhinein sortiert, danach ob sie für uns im ersten Moment umsetzbar klangen oder eher nicht. Auch wurde sich Gedanken darüber gemacht, was der Torwart für zusätzliche Bewegungen braucht. Im Folgen sind die wesentlichen Bewegungen aufgezählt und kurz beschrieben.

4.1.1 Relevante (allg.) Bewegungen

Bewegung	Kurzbeschreibung und Anwendung
Laufen	Die Laufbewegung wird nicht implementiert, da der NAO auf Grund von sonst aufwändigen Algorithmen über das Feld gebeamt wird.
Schuss	Soll per Tritt oder per "dagegen Beamenrealisiert werden, das wird u.U. testen. Der Schuss ist relevant für den Angriff und für Elfmeter.
Pass	Der Pass ist der gerichtete Schuss zu einem eigenen Mitspieler oder zu einer bestimmten Position. Er wird vermutlich auch (größtenteils) für Ëinwurf", Anstoß, Abstoß, Ecke und Freistoß verwendet.
Kopfbewegung	Der NAO muss sich umsehen, um seine Umgebung wahrnehmen zu können. Die Kopfbewegung ist also dringend nötig. Es wird wahrscheinlich ein Kopfbewegung zum rund um gucken und eine um an eine bestimmte Position zu sehen geben.
Armbewegung	Die Armbewegung ist u.U. relevant für die Kommunikation in gewissem Maße. Auch kann sie für den Torwart von nutzen sein.
"Dribblen"	Soll eine Kombination aus Schießen (Passen) und Laufen sein.
Aufstehen	Nach dem Hinfallen oder Umkippen muss der NAO wieder aufstehen können. Dieses soll nicht durch einfaches Beamen realisiert werden, falls das möglich sein sollte. (evtl. wird der NAO in der gleichen Position gebeamt, was bedeutet er liegt immer noch)
Drehen	Um sich umzusehen oder die Laufrichtung zu ändern sehr relevant. Soll auch durch Beamen passieren.

4.1.2 Torwartspezifische Bewegungen

Bewegung	Kurzbeschreibung und Anwendung
Halten	Der Keeper soll versuchen den Ball nicht ins Tor zu lassen. Dafür benötigt er gewisse Bewegungen, z.B. dem Ball entgegenlaufen, Arm ausstrecken, Springen.
Zum Ball springen	Bewegung des Keepers, die den Torwart möglichst viel verdecken lässt vom Tor, damit der Ball nicht hineingeschossen werden kann.
Aufstehen	Evtl. ist es nötig eine extra Aufstehbewegung zu kreieren, falls der Ball zu gefährlich liegt und durch das Aufstehen ins Tor befördert wird.

4.1.3 Irrelevante Bewegungen

Bewegung	Kurzbeschreibung und Anwendung
Fallrückzieher	Reaktionstechnisch (und möglicherweise auch durch die Technik des NAO's) vermutlich nicht realisierbar. Definitiv aber erstmal unnötig für uns.
Kopfball	Vermutlich aufgrund gewisser Reaktionszeiten nicht machbar.

4.1.4 Bewegen der Gelenke

Für die Kommunikation mit dem Server werden zwei verschiedene Bezeichnung genutzt. Die Perceptor names sind die Namen, die der Server sendet, damit der NAO weiß, wie seine Gelenke stehen. Die Effector names werden mit den zugehörigen Geschwindigkeitenan den Server geschickt. Die Geschwindigkeiten geben an wie schnell sich die Gelenke ab dem nächsten Zyklus bewegen sollen. Dabei ist zu beachten, dass die Gelenke eine maximale Auslenkung besitzen.
Die Gelenke werden prinzipiell durch den folgenden Befehl bewegt:
send(socket, message)
Die *message* sieht dabei wie folgt aus:
message = (EffectorName AngularSpeed)
Die Perceptor und Effector names sind in der folgenden Tabelle [5] aufgeführt.

No.	Desc.	Perceptor name	Effector name
1	Neck Yaw	hj1	he1
2	Neck Pitch	hj2	he2
3	Left Shoulder Ptch	laj1	lae1
4	Left Shoulder Yaw	laj2	lae2
5	Left Arm Roll	laj3	lae3
6	Left Arm Yaw	laj4	lae4
7	Left Hip YawPitch	llj1	lle1
8	Left Hip Roll	llj2	lle2
9	Left Hip Pitch	llj3	lle3
10	Left Knee Pitch	llj4	lle4
11	Left Foot Pitch	llj5	lle5
12	Left Foot Roll	llj6	lle6
13	Right Hip YawPitch	rlj1	rle1
14	Right Hip Roll	rlj2	rle2
15	Right Hip Pitch	rlj3	rle3
16	Right Knee Pitch	rlj4	rle4
17	Right Foot Pitch	rlj5	rle5
18	Right Foot Roll	rlj6	rle6
19	Right Shoulder Pitch	raj1	rae1
20	Right Shoulder Yaw	raj2	rae2
21	Right Arm Roll	raj3	rae3
22	Right Arm Yaw	raj4	rae4

4.2 Umsetzung

Nachdem die Bewegungen grob aufgelistet waren, wurde überlegt, welche Möglichkeit der Umsetzung es gibt. Für viele Bewegungen ist es möglich einen Keyframe zu erzeugen und diese in einer Keyframe-Engine auszuwerten, da diese nicht von Parametern oder nicht direkt von äußeren Einwirkungen abhängen. Für andere müssen andere Algorithmen geschrieben werden.

4.2.1 Keyframes

Ein Keyframe ist bei uns eine Abfolge von Momentaufnahmen von den Stellungen der Gelenke des NAOs. Der NAO hat 22 Gelenke, dessen Auslenkungswinkel jeweils angegeben wird. Diese einzelnen Positionen der Gelenke speichern wir in einem mehrdimensionalen Array ([n][23], wobei n die Anzahl der Frames für die jeweilige Bewegung ist). An den Stellen [i][0] für steht die Zeit, wie lange der einzelne Frame dauern soll. Dann Folgen die Gelenke, die so sortiert sind, wie die Reihenfolge im Array name", in dem die Namen der Gelenke stehen. Das war nötig, da wir verschiedene Reihenfolgen der Gelenkbenennenung hatten und so eine Zuordnung möglich ist.
Die Keyframes erschaffen in unserem Projekt definierte Bewegungen, die häufiger ausgeführt werden müssen. Damit wir nicht alle Keyframes selbst erzeugen müssen, wollten wir die Aufstehbewegungen (vom Bauch und vom Rücken) der Humboldt Universität[2] benutzen, die uns diese zur Verfügung gestellt hat. Das Aufstehen vom Rücken konnten wir auch mit einigen veränderungen benutzen, jedoch wollte unser NAO mit unserer Keyframe-Engine nicht aus seiner Bauchlage aufstehen, weshalb wir diesen neu entwickelten. Momentan stehen folgende Bewegungen durch Keyframes zur Verfügung:

- lookAround (umsehen)
- stand_up_from_back
- stand_up_from_front
- kick1 (ein erster sanfter Schuss)

4.2.2 Berechnungen (Keyframe-Engine)

Die Berechnunung findet in unserer Keyframe-Engine statt. Dort stehen die einzelnen Bewegungs-Methoden zur Verfügung.

- lookAround()
- stand_up_from_back()
- stand_up_from_front()
- kick1()

Alle diese Methoden holen sich als erstes das jeweilige Keyframe-Array und die Reihenfolge der Gelenke mit dem Array name. Da die Keyframe-Engine in jedem Zyklus aufgerufen wird, muss (der aktuelle Keyframe,) die aktuelle Zeile und die vergangene Zeit des Frames gespeichert werden. Zusätlich überprüfen wir auch, ob wir im letzten Frame der Bewegung angekommen sind. Sind wir im letzten Frame, werden alle Gelenkgeschwindigkeiten auf 0.0 gesetzt, damit sie sich nicht weiter bewegen. Die eigentliche Berechnung, der nächsten Geschwindigkeiten findet in der Methode get_new_joint_postion(keyframe, name). Dort werden die jeweiligen Gelenkepaare, bestehen aus aktueller Gelenkposition und der Position, wo wir in dem aktuellen Frame hinwollen berechnet. Dabei müssen auf mehrere Sachen geachtet werden:

1. Die aktuelle Gelenkposition ist die vom Zyklus davor, d.h. wir haben einen Versatz von 20ms.
2. Wir wollen nicht im nächsten Zyklus die gesamte Bewegung ausführen, sondern nur die, die in den nächsten 20ms passieren soll.

3. Das Gelenk darf nicht über seine maximale Auslenkung gesteuert werden.

Jede Gelenkgeschwindigkeit wird dann mit der folgenden Formel berechnet:

$$\texttt{Winkel} = \frac{20ms \cdot (\overbrace{keyframe\_joint}^{\texttt{Winkel aus Keyframe}} - (\overbrace{hinge\_joint}^{\texttt{akt. Gelenkstellung}} + \overbrace{last\_joint\_angle}^{\texttt{letzter berechneter Winkel}}))}{\underbrace{keyframe[0]}_{\texttt{Dauer akt. Frames}} - \underbrace{progressed\_time}_{\texttt{vergangene Zeit akt. Frames}}} \tag{1}$$

Dieser Winkel wird für den nächsten Zyklus gespeichert und in Rad/sec umgerechnet.

$$\texttt{Winkelgeschwindigkeit} = \frac{\overbrace{rad(Winkel)}^{\texttt{oben berechneter Winkel}}}{\underbrace{20 * 0,001}_{\texttt{20ms}}} \tag{2}$$

Danach wird die abgelaufe Zeit um 20ms erhöht und wenn die Zeile des Keyframes abgearbeitet ist wird die Zeile um 1 erhöht. Ist die Bewegung am Ende angekommen, wird die aktuelle Zeile aud 0 gesetzt.

4.2.3 Einzelne Bewegungen

Im Folgenden werden die einzelnen Bewegungen vorgestellt und dessen Umsetztung beschrieben.

Aufstehbewegungen Das Aufstehen ist besonders gut geeignet mit einem Keyframe umgesetzt zu werden, da hier keine Parameter nötig sind. Da es für den NAO einen Unterschied macht, ob er auf dem Rücken liegt oder auf dem Bauch mussten zwei verschiedene Keyframes entwickelt werden.

Aufstehen vom Rücken aus Wurde als erstes voll funktionstüchtig implementiert. Sobald der NAO auf dem Rücken liegt, steht er wieder auf und stellt sich nahezu vollständig gerade wieder hin. Dieser Keyframe wurde von der Humboldt Universität[2] benutzt und musste nur an einigen Stellen verändert werden.

Aufstehen vom Bauch aus Hier hatten wir einen Keyframe von der Humboldt Universität[2] zur Verfügung, der unseren NAO jedoch nicht aufstehen lassen hat. Deshalb wurde er entwickelt. ...

Kopfbewegungen Die Bewegung des Kopfes ist nicht nur wichtig, um sich umzusehen, damit der NAO z.B. seine Position besser bestimmen kann oder den Ball sehen kann, wenn er außerhalb des Sichtfeldes seines ist, sondern auch zur Fixierung eines Punktes.

Umsehen Dies ist eine feste Abfolge von Bewegungen. Es soll eine möglichst großer Bereich gesehen werden. Dafür wird ein Keyframe geschrieben, da hier keine Berechnungen gebraucht werden. Dazu wird der Kopf zuerst nach linkt, nach unten und dann nach rechts bewegt.

Auf eine Punkt gucken Diese Bewegung benötigt eine Berechnung, die ausgehend von der aktuellen Stellung der Kopfgelenke den Weg zum gewünschten Punkt berechnet. Es ist noch nicht geklärt, mit welchen Daten die Berechnung stattfinden sollen. Dazu ist eine Absprache vor allem mit der Taktikgruppe nötig.

Schuss Für den Schuss sind bis jetzt Keyframes geplant. Dabei soll unterschieden werden, ob der Schuss weit oder kurz sein soll (bis jetzt 2 Keyframes). Schwierigkeiten, die wir dabei bekommen könnten, sind dass der NAO ziemlich genau vor dem Ball stehen muss und dass er das Gleichgewicht halten muss.

4.3 Interfaces

4.3.1 Variablen

Working Wenn diese Variable auf True gesetzt ist, ist ein Keyframe noch nicht beendet. Daher darf kein Movement Befehl ausgeführt werden und nur Keyframes gestartet werden, die den Kopf bewegen

4.3.2 Funktionen

work Diese Funktion muss in jedem Cycle aufgerufen werden. Sie überprüft ob aktuell noch Keyframes auszuführen sind, ist dies der Fall werden die Geschwindigkeiten für diesen Cycle berechnet.

4.3.3 Keyframe Aufrufe

Diese Funktionen werden einmal aufgerufen und bestimmen damit den aktuell auszuführenden Keyframe. Die Bewegung des Naos erfolgt ausschließlich über **work**.

Name	Funktion
stand()	Grundhaltung des Nao, die nach dem Hinzufügen eines Nao aufgerufen werden sollte, um aus der vordefinierten Haltung in einen normalen Stand zu kommen.
stand_up_from_back()	Diese Bewegung lässt den Nao aufstehen, wenn er auf dem Rücken liegt (modifizierte Humboldt-Version).
stand_up_from_front()	Diese Bewegung lässt den Nao aufstehen, wenn er auf dem Bauch liegt (eigenständig geschriebener Keyframe der sich an der Aufstehbewegung der zweitplatzierten Mannschaft, der Osaka Open 2011 orientiert).
kick_left() / kick_right()	Dieser Tritt ist ein sehr kurzer, erster Versuch einen Tritt auszuführen. Er könnte in Zukunft zum Dribbling verwendet werden.
kick_strong_left() / kick_strong_right()	Ein erster starker Tritt wird insbesondere für Torschüsse benötigt. Er lässt den Nao regelmäßig umkippen, bei einem solchen Schuss lässt sich damit aber arbeiten.
kick_in_left() / kick_in_right()	Eine starke Schussvariante, die insbesondere für Anstoß, Abstoß, Freistoß, Eckstoß und Strafstoß gedacht ist.
head_lookAround()	Wird eine vollständige Bewegung zum umschauen benötigt, so wird **head_lookAround** verwendet.
head_move(angle)	Bewegt den Kopf horizontal um den angegebenen Winkel. Positive Werte bewegen den Kopf, vom Nao aus gesehen, nach links; negative nach rechts. Maximalwerte liegen bei 120 bzw -120 Grad. Die Bewegung wird vertikal betrachtet auf der 0 Grad Ebene ausgeführt.
head_stop()	Stoppt die Kopfbewegung an der aktuellen Position. Aufgrund des Delays stoppt der Kopf allerdings einen Schritt weiter als an der im aktuellen Cycle von den Sensoren angegebene Position.
head_reset()	Setzt den Kopf auf 0 Grad horizontal und vertikal zurück.
head_down()	Bringt den Nao dazu, nach unten zu schauen.
fall_on_front()	Diese Testfunktion lässt den Nao auf den Bauch fallen.
fall_on_back()	Diese Testfunktion lässt den Nao auf den Rücken fallen.
parry_left()	Der Torwart führt eine Parade nach links aus und steht nach kurzer Zeit wieder auf.
parry_right()	Der Torwart führt eine Parade nach rechts aus und steht nach kurzer Zeit wieder auf.
parry_straight()	Der Torwart versucht möglichst viel Weg zu versperren wenn der Ball nahezu gerade auf ihn zukommt.
parry_straight1()	Der Torwart geht in die Hocke um den Ball abzufangen wenn der Ball exakt auf ihn zukommt.

5 Laufen

Auf dieser Seite wird die Movementklasse beschrieben, die dafür verantwortlich ist, dass der Nao sich zu einem Ziel bewegen kann. Wir haben beschlossen, die Laufbewegungen zu abstrahieren und verwenden stattdessen von der Simulation zur Verfügung gestellte Beam-Befehle.

5.1 Aktueller Zustand

In der aktuellen Implementation wird der Beam-Befehl für die Monitorschnittstelle genutzt. Das heißt, es kann nach dem Anstoß immernoch genutzt werden. Außerdem steht uns die echte Position nicht zur Verfügung, die aber gebraucht wird, um sich realitätsnah zu bewegen. Zu diesem Zweck wird aktuell die Position in einer lokalen Variable gespeichert.

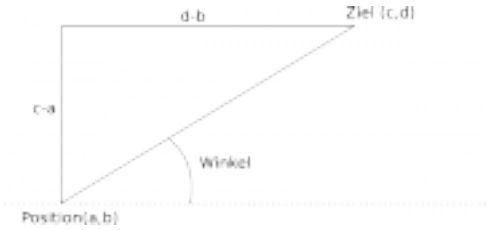

Abbildung 8: Winkelberechnung

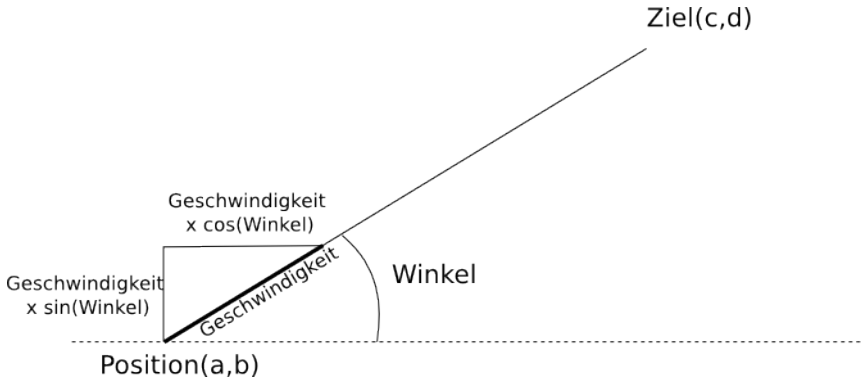

Abbildung 9: Berechnung des nächsten Schrittes

5.2 Nutzung

run(x,y) : Rufe diese Funktion der Movementklasse auf, um einen Schritt in die Richtung des angegeben Zieles zu machen. x und y sind die Zielkoordinaten.

5.3 Ansatz und Schnittstellen

Es gibt zwei verschiedene Arten von Beam-Befehlen:
Über die Agent-Schnittstelle gibt es einen Beam-Befehl, um die Naos vor dem Anstoß zu positionieren. Nach dem Anstoß wird diese Schnittstelle nicht mehr funktionieren, deswegen ist diese Schnittstelle nicht möglich.
Über das Monitor-Protokoll gibt es ebenfalls mehrere Beam-Befehle. Diese sind eigentlich für den Schiedsrichter oder ähnliches gedacht und können deshalb auch nach dem Anpfiff genutzt werden. Dieses soll am Ende genutzt werden.
Da alle Beam-Befehle mit absoluten Variablen arbeiten, benötigen wir unsere echte Position, um den nächsten Schritt (der natürlich mithilfe der Wahrnehmung berechnet werden soll) relativ zu unserer Position ausgeführt werden kann.

5.4 Berechnungen

Um uns dem Ziel zu nähern, berechnen wir den Winkel in dem das Ziel von der aktuellen Position aus liegt und bewegen uns einen Schritt in diese Richtung.
Wenn (a, b) die aktuelle Position ist und (c, d) die Zielkoordinaten, dann berechnen wir den Winkel mit folgender Formel:

$$angle = arctan\frac{c-a}{d-b} \tag{3}$$

Wir verwenden die Funktion atan2 von Python, die es ermöglicht zwei Argumente zu übergeben ohne die Division, so dass der eindeutige Winkel berechnet werden kann.

Danach wird Sinus und Kosinus des Winkels berechnet, mit der Geschwindigkeit multipliziert und auf die entsprechenden Positionskoordinaten aufsummiert, um eine einheitliche Schrittlänge zu gewährleisten.
Der Winkel muss vor dem Übergeben an die Simulation zu Grad konvertiert, negiert und um 90° $(n/2)$ erhöht werden damit er mit den, von simspark erwarteten Werten, übereinstimmt.
Danach wird die neue Position, sowie der berechnete Winkel in einem Beam-Befehl übergeben.

5.5 Protokoll

Der Beam-Befehl über die Agenten-Schnittstelle nimmt die folgende Form an:

```
(beam <x> <y> <rot>)
```

Wobei x und y die entsprechenden Koordinaten sind und <rot> der Winkel, in den der Nao schauen soll. Dieser Ansatz soll später durch das Monitorprotokoll ersetzt werden.
Im Monitorprotokoll gibt es ein Befehl, mit dem man einen einzelnen Nao bearbeiten kann:

```
(agent (unum <num>) (team <team>) (pos <x> <y> <z>)
                                  (move <x> <y> <z> <rot>)
                                  (battery <batterylevel>)
                                  (temperature <temperature>)
)
```

Da dieser Befehl normalerweise nicht vom Nao kommt sondern vom Schiedsrichter oder ähnliches, muss hier sowohl das Team als auch die Spielernummer mit angegeben werden. Da wir auch die Rotation mit angeben wollen werden wir auf den move-Abschnitt des Befehls zurückgreifen. pos battery und temperature können weggelassen werden, wenn sie nicht genutzt werden.
Alle Argumente müssen absolut abgegeben werden.

6 Kommunikation

6.1 Grundlegendes

Für die Kommunikation zwischen den NAOs stellt der Server 2 Schnittstellen zur Verfügung, den Say-Effector sowie den Hear-Perceptor. Diese erlauben es uns, einen NAO etwas sagen zu lassen und zu hören, was die anderen NAOs von sich geben. Die Kommunikation unterliegt allerdings einigen Einschränkungen:

- Eine Nachricht ist maximal 20 Zeichen lang.
- Die Nachricht darf nur einen eingeschränkten (92) ASCII-Zeichensatz verwenden. (zur Zeit benutzen wir 90, um den Umgang mit Python zu erleichtern)
- Nachrichten können 50m weit gehört werden.
- Die NAOs können maximal alle 0.04s eine Nachricht hören, in der Zwischenzeit (also <0.04s nachdem sie etwas gehört haben) sind sie taub.
- Sprechen zwei NAOs gleichzeitig, wird nur eine Nachricht gehört. Der Sprecher hört jedoch immer seine eigene Nachricht. Die Teams sprechen versetzt, können sich also nicht blockieren.

6.2 Konzept

Die Nachrichten werden zurzeit nach dem folgenden Schema codiert und übertragen: Zunächst wird eine Methode aufgerufen, welcher übergeben wird, was gesagt werden soll. Hierfür stehen sowohl Funktionen zur Verfügung, bei denen man über die Parameter genau definiert welche Nachricht gesendet werden soll;

```
%Beispielsweise sendet dieser Aufruf an NAO 5 die Nachricht, er soll zum Ball gehen
sayCommandTo(1, 5)
```

als auch vorgefertigte Funktionen, welche die Codelesbarkeit und den Umgang mit der Kommunikation erleichtern sollen.

```
%Derselbe Aufruf wie oben.
sayGoToBall(5)
%Fügt man weitere Parameter hinzu erweitert sich die Nachricht. Wir können z.B. auch Koordinaten
des Balls mitliefern.
sayGoToBall(5, 8.52, -3.11)
```

Die aufgerufene Methode codiert die Nachricht nun derart, dass sie für den Say-Effector brauchbar ist.
Eine codierte Nachricht ist folgendermaßen aufgebaut:
<commandcode><sender><target><parameter1>...<parameterN><checksum>

- commandcode - spezifiziert die Struktur und Aussage der Nachricht ("gehe zum Ball")
- target - enthält den Empfänger der Nachricht("5"). 0 steht hierbei für eine Nachricht an alle
- parameter1 bis N - enthalten mitgelieferte Daten wie Koordinaten oder dergleichen("8.52, -3.11")
- checksum - wird aus den übrigen Bestandteilen der Nachricht gebildet und validiert die Nachricht

Für das Beispiel von oben würden wir folgende Codierung erhalten: ${&{%]NCS!GI
Nun wird diese codierte Nachricht über den Say-Effector an den Server geschickt.
Ein NAO hat nun über die Hear-Methode die Möglichkeit, diese Nachricht zu empfangen. Diese wird dann zunächst vom Translator decodiert. Daraus wird nun ein HearObject erstellt, dessen Variablen mit den übergebenen Parametern übereinstimmen. Dieses HearObject wird nun zurückgegeben, und kann jederzeit per eval() ausgewertet werden.

6.3 Commandcode - Tabelle

Commandcode	Message Type	Mögliche Parameter
0	reserved, do not use	-
1	goToBall	target(0), x(0.0), y(0.0)
2	standUp	target(0)
3	iAmHere	sender, x, y
4	youAreHere	target, x, y
5	ballPosition	x, y
6..89	customMessage	-

In Klammern angegebene Werte sind Default-Werte, und müssen demnach nicht übergeben werden. Die bisherigen Commandcodes stellen lediglich Beispiele dar. Weitere Messages einzufügen stellt kein Problem dar. Erwünschte Commandcodes, samt Namen und Parametern, einfach hier hinschreiben, und wir fügen sie so schnell es geht ein.

7 Scenegraph

7.1 Einleitung

Über den TCP port 3200 kann eine Server/Monitor Verbindung aufgebaut werden. Verbindet sich dann ein Monitor mit dem Server werden Informationen in folgender Reihenfolge übermittelt:

1. ((<EnvironmentInformation>)(<SceneGraphHeader>)(<SceneGraph>))
2. ((<GameState>)(<SceneGraphHeader>)(<SceneGraph>))
3. ((<partial GameState>)(<SceneGraphHeader>)(<partial/full SceneGraph>))

Hinweis: Da für uns nur der Scene Graph Header und der Scene Graph an sich relevant sind, werden wir nicht näher auf 'EnviromentInformation' und 'GameState' eingehen.

7.2 Scene Graph Header

Der Scene Graph Header sieht folgt aus:
`(Name Version Subversion)`

- Name kann zwei werte annehmen:
  - RSG steht für Ruby Scene Graph und zeigt an, dass es ein vollständig neuer Scene Graph ist.
  - RDS steht für Ruby Diff Scene und zeigt an, dass es ein partieller Graph ist. Der partielle Scene Graph beinhaltet dann viele leere Knoten und die aktualisierten Knoten.
- Version: Versions Nummer des Scene Graph.
- Subversion: Die Subversion Nummer des Scene Graph.

7.3 Scene Graph

Der Scene Graph ist ein Baum, der aus verschiedenen Knoten besteht. Der root Knoten ist als <0;0;0> definiert und hat keine Rotation. Die Position und Rotation eines jeden Kindes kann durch die Multplikation des root Knotens bis zum Kind herausgefunden werden.
Es werden aber keine spezifischen Objekte gespeichert, sondern nur Texturen der Objekte.

7.3.1 Base Knoten

Jeder Knoten, der kein Transform Knoten, ein Static Mesh Knoten oder ein Light Knoten ist, ist ein Basis Knoten (inklusive des Root Knotens). Da er für uns irrelevant ist, gehen wir nicht näher auf den Knoten ein.

```
(nd BN <contents>)
```

7.3.2 Transform Knoten

Transform Knoten sind 4x4 Transformationsmatrizen vom Aufbau:

```
[nx ox ax Px]
[ny oy ay Px]
[nz oz az Pz]
[ 0  0  0  1]
```

wobei n, o, a für Normal, Orientierung und Annäherung stehen, die allesamt als Richtungsvektoren angegeben sind.
Die letzte Spalte ist ein Orstvektor, stellt also eine Position dar. Wenn wir die Transform Knoten vom Root bis zu einem Kind multipliziert haben, enthalten die ersten drei Einträge der letzten Spalte demnach die Position des Kindes, das in dem Blatt gespeichert ist. Dies kann zum Beispiel ein Teil eines Naos oder der Ball sein.
Beispielnachricht:

```
(nd TRF (SLT nx ny nz 0 ox oy oz 0 ax ay az 0 Px Py Pz 1 ))
```

7.3.3 Geometry Nodes

Die folgenden zwei Knoten definieren die Objekt Form. Sie spezifizieren ihre Maße und Texturen. Sie sind immer Blätter.

StaticMesh Defniert die Textur, die aus der .obj Datei geladen wird:

```
(nd StaticMesh (load <model>) (sSc <x> <y> <z>)
  (setVisible 1)
  (setTransparent)
  (resetMaterials <material-list>)
)
```

model: Ist der Dateipfad zur .obj Datei.
sSc: Definiert die Maße des Objektes.
setVisible (optional): Gibt an, ob das Objekt sichtbar sein soll oder nicht.
setTransparent (optional): Unterschied zu setVisible unklar.
resetMaterials: Definiert eine Liste von Materialen, die mit der .obj Datei verknüpft sind.

Beispielnachricht:

```
(nd StaticMesh (load models/rlowerarm.obj)
(sSc 0.05 0.05 0.05)(resetMaterials matLeft naowhite))
(nd StaticMesh (load models/naohead.obj)
(sSc 0.1 0.1 0.1)(resetMaterials matLeft naoblack naogrey naowhite))
```

SMN

```
(nd SMN (load <type> <params>) (sSc <x> <y> <z>)
  (setVisible 1)
  (setTransparent)
  (sMat <material-name>)
)
```

type:

- StdUnitkBox
- StdUnitCylinder wobei <params> zwei Angaben macht: Länge und Radius.
- StdUnitSphere
- StdCapsule with <params> (Nicht benutzt in rcssserver3d 0.6.3; Parameter unklar)

sSc: Definiert die Maße des Objektes.
setVisible (optional): Gibt an, ob das Objekt sichtbar sein soll oder nicht.
setTransparent (optional): Unterschied zu setVisible unklar.
sMat: Spezifiziert den Namen des Materials, das dem Objekt zugewiesen wird (z.B. matWhite, matYellow, etc.)

Beispielnachricht:

```
(nd SMN (load StdUnitBox) (sSc 1 31 1) (sMat matGrey))
(nd SMN (load StdUnitCylinder 0.015 0.08) (sSc 1 1 1) (sMat matDarkGrey))
```

7.3.4 Light Node

Ein Light Knoten zeigt an, wie Licht ein bestimmtes Objekt beeinflusst.

```
(nd Light (setDiffuse x y z w) (setAmbient x y z w) (setSpecular x y z w))
```

wobei < x; y; z > ein Vektor ist, der die Richtung des Lichtes definiert und w Skalenfaktor ist.
setDiffuse: Reflektion von unebenen Flächen.
setAmbient: Licht, das Objekte oder Szenerien umgibt.
setSpecular: Reflektion von glatten Oberflächen.

7.4 Beispielnachricht und Aufbau

```
[
  ['nd', 'TRF', ['SLT', 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, -10, 10, 10, 1],
     ['nd', 'Light', ['setDiffuse', 1, 1, 1, 1],
                     ['setAmbient', 0.8, 0.8, 0.8, 1],
                     ['setSpecular', 0.1, 0.1, 0.1, 1]]
  ],

  ['nd', 'TRF', ['SLT', -1, -8.74228, 'e-08', -3.82137, 'e-15', 0, 0,
                 -4.37114, 'e-08', 1, 0, -8.74228, 'e-08', 1, 4.37114,
                 'e-08', 0, 0, 0, 0, 1],
     ['nd', 'StaticMesh', ['setVisible', 1],
                          ['load', 'models/naosoccerfield.obj'],
                          ['sSc', 2.5, 1, 2.5],
                          ['resetMaterials', 'None_rcs-naofield.png']]
  ],
 ...
]
```

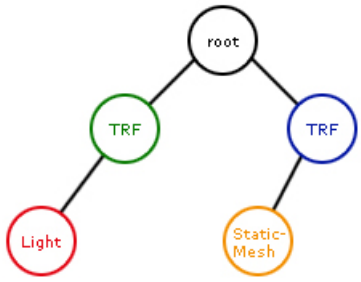

Abbildung 10: Graphische Veranschaulichung

Auszug aus einer Nachricht, die den kompletten Scene Graph enthält. Die Nachricht ist schon in der Form, in die sie der Parser bringt, also eine Liste und keine S-Expression mehr:

```
[
   ['nd', 'TRF', ['SLT', 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, -10, 10, 10, 1],
        ['nd', 'Light', ['setDiffuse', 1, 1, 1, 1], ['setAmbient', 0.8, 0.8, 0.8, 1], ['setSpecular'
   ],

   ['nd', 'TRF', ['SLT', -1, -8.74228, 'e-08', -3.82137, 'e-15', 0, 0, -4.37114, 'e-08', 1, 0, -8.74
       ['nd', 'StaticMesh', ['setVisible', 1], ['load', 'models/naosoccerfield.obj'], ['sSc', 2.5, 1
   ],
 ...
]
```

Abstrahieren wir die oben gezeigte Nachricht etwas um den Aufbau der Nachricht zu verstehen:

```
[
   -$\,$- 1. Kind des root (Kind 1) -$\,$-
   ['nd', 'TRF', ['SLT', MATRIX],
        -$\,$- 1. Kind von 'Kind 1' -$\,$-
        ['nd', 'Light', ['setDiffuse'], ['setAmbient'], ['setSpecular']]
   ],
   -$\,$- 2. Kind des root (Kind 2)-$\,$-
   ['nd', 'TRF', ['SLT', MATRIX],
       -$\,$- 1. Kind von 'Kind 2' -$\,$-
       ['nd', 'StaticMesh', ['setVisible'], ['load'], ['sSc'], ['resetMaterials']]
   ],
...
]
```

Der Vollständigkeit halber hier eine (partielle) Beispielnachricht eines aktualisierten Baumes:

```
[
    ['nd',
        ['nd']],
    ['nd',
        ['nd']],
    ['nd',
        ['nd', 'StaticMesh']
...
]
```

7.5 Implementierung

7.5.1 Datenstrukturen

Um alle Knoten zu speichern, haben wir für jeden Knoten eine Datenstruktur angelegt.

7.5.2 Wichtigsten Methoden

def run_cycle(self): Wird vom Agenten aufgerufen, um dann seine Position zu bestimmen. Hier wird der Header überprüft und es wird entweder ein komplett neuer Graph eingelesen oder ein vorhandener wird aktualisiert.
def create_trans_node(self,msg): Erstellt ein Transform Knoten und speichert die Matrix als numpy Matrix.

```
Scenegraph Matrix: [SLT, nx, ny, nz, 0, ox, oy, oz, 0, ax, ay, az, 0, Px, Py, Pz, 1]

numpy Matrix:
( ((nx, ox, az, Px]),
   (ny, oy, ay, Py),
   (nz, oz, az, Pz),
   (0 ,  0,  0,  1)) )
```

def create_light_node(self,msg): Erstellt einen Light Knoten. Speichere dabei den diffuse-, ambient- und specular-Inhalt jeweils als numpy Array.
def create_smn_node(self,msg): Erstellt einen SMN Knoten. Die sSc- und load-Inhalte werden als Listen gespeichert.
def create_static_mesh_node(self,msg): Erstellt einen SMN Knoten. Die sSc-, reset- und load-Inhalte werden als Listen gespeichert.
def update_scene(self, msg): Aktualisiert einen vorhandenen Scenegraph.
def get_position(self, team, naoID): Gibt die aufsummierte Matrix vom Root bis zum NAO zurück. Als Parameter werden das Team (left,right) und die Rückennummer des NAOs benötigt.
def get_position_xy(self, team, naoID): Gibt die x,y-Position des NAOs zurück. Als Parameter werden das Team (left,right) und die Rückennummer des NAOs benötigt.
def get_ball_position(self): Gibt die x,y-Position des Balles zurück.

8 Drawing

Diese Seite dokumentiert die Funktionen der beiden Klassen Drawing und DrawingAdvanced. Hintergrundinformationen zu den Zeichenfunktionen von roboviz finden sich auf der Roboviz-Seite[1] .**Hinweis: Das Zeichnen funktioniert nur in roboviz und nicht im Monitor!**

8.1 Drawing

Dies ist die Klasse die hauptsächlich zum Zeichnen benutzt werden sollte. Sie benutzt die Klasse Vector aus world.py für die Übergaben von Koordinaten und enthält in Zukunft auch einige komplexere Zeichenbefehle (zum Beispiel für das Zeichnen von Pfeilen). Es werden nur 2D-Koordinaten unterstützt. Wenn ihr Dinge in die Luft malen müsst, dann benötigt ihr die Klasse DrawingAdvanced.
Einige Dinge gelten bei den Übergaben an alle Funktionen:

- *name* sollte immer ein String sein
- *color* sollte ein array oder eine Liste mit Einträgen in der Reihenfolge R(ed),G(reen),B(lue) sein
- *thickness* kann je nach belieben float oder int sein
- x- und y-Koordinaten können float oder int sein
- beliebig viele gezeichnete Formen können den gleichen *name* haben

Die einzelnen Funktionen mit kurzer Erklärung:

8.1.1 showDrawingsNamed

Header:

```
showDrawingsNamed(self, name)
```

Roboviz benutzt einen doppelten Buffer beim zeichnen. D.h. Zeichnungen werden erst komplett im Hintergrund gemacht und dann angezeigt, um Flimmern zu vermeiden. Mit dieser Funktion gebt ihr Roboviz den Befehl alle gezeichneten Dinge anzuzeigen, die mit dem übergebenen String *name* **beginnen**.

```
showDrawingsNamed("circles")
```

Zeigt alle Objekte an, die "circles" heißen, aber auch solche mit Namen wie *"circles_are_supercool"* oder *"circles.team-left"*.
Achtung: Gleichzeitig werden alle Zeichnungen, die bereits sichtbar sind und einen passenden Namen haben ausgeblendet und **gelöscht**. showDrawingsNamed ist also eine der wichtigsten Funktionen. Wenn ihr die nicht am Ende aufruft seht ihr nämlich nix :-)

8.1.2 drawCircle

Header:

```
drawCircle(self, center, radius, thickness, color, name)
```

Zeichnet einen Kreis um die Stelle *center* (übergeben als Vector) mit Radius *radius* und Strichdicke *thickness*. *thickness* wird in Pixeln angegeben, *radius* in Metern.

8.1.3 drawLine

Header:

```
drawLine(self, startPoint, endPoint, thickness, color, name)
```

Eine schlichte Linie von startPoint nach endPoint.

8.1.4 drawPoint

Header:

```
drawPoint(self, position, size, color, name)
```

Ein Punkt. *size* ist eine Angabe in Pixeln, bei größeren Werten wird der Punkt zum Quadrat.

8.1.5 drawSphere

Header:

```
drawSphere(self, center, radius, color, name)
```

Zeichnet eine ziemlich eckige Kugel an der Stelle *center*. *radius* wird in Metern angegeben.

8.1.6 drawPolygon

Header:

```
drawPolygon(self, vertices, color, name)
```

vertices ist in dieser Funktion eine Liste oder ein Array von Vektoren (momentan zu finden in world.py). Diese Vektoren geben die Eckpunkte der zu zeichnenden Form an. Soweit ich feststellen konnte, kann man sich das Vorgehen von roboviz hier so vorstellen:

- Es wird eine Linie gezogen vom ersten zum zweiten Eckpunkt, vom zweiten zum dritten ... und von letzten zum ersten
- Die entstandene Form wird mit der übergebenen Farbe ausgefüllt

Für dreidimensionale Formen ist mir nicht ganz klar, wie das genaue Aussehen des Objekts entsteht, aber die Klasse Drawing unterstützt wie oben erwähnt sowieso nur zweidimensionale Koordinaten.

8.1.7 drawStandardAnnotation

Header:

```
drawStandardAnnotation(self, position, color, text, name)
```

Schreibt *text* an die Stelle *position*.

8.1.8 drawAgentAnnotation

Header:

```
drawAgentAnnotation(self, agentNum, teamNum, color, text)
```

Schreibt *text* über den Kopf des Agenten mit der Nummer *agentNum* aus dem Team *teamNum*. 0 steht dabei für das linke, 1 für das rechte Team. Diese Funktion hat keine Namensübergabe, weil der Text sofort erscheint und automatisch dem Agent folgt. Man entfernt ihn mit der Funktion removeAgentAnnotation

8.1.9 removeAgentAnnotation

Header:

```
removeAgentAnnotation(self, agentNum, teamNum)
```

Entfernt die Schrift über dem Kopf des Agenten mit der Numer *agentNum* aus dem Team *teamNum*. 0 steht dabei für das linke, 1 für das rechte Team.

8.1.10 drawGrid

Header:

```
drawGrid(self, color, name)
```

Zeichnet ein Gitter über das ganze Spielfeld, sodass man Positionen etwas besser vom Bildschirm ablesen kann.

8.1.11 drawArrow

Header:

```
drawArroW(self, startPoint, endPoint, thickness, color, name)
```

Zeichnet eine Linie mit Pfeilspitze bei *endPoint*.

8.2 DrawingAdvanced

DrawingAdvanced hält sich sehr viel näher an der Beschreibung der Commands auf der Roboviz-Seite und ist deshalb ein bisschen komplexer, kann dafür aber in 3D zeichnen.
Unterschiede zu *Drawing*

```
*Koordinaten werden einzeln übergeben, nicht als Vektor
*Farben werden einzeln übergeben, nicht als Array oder Liste
*Die Funktion showDrawingsNamed heißt hier swapBuffers
*DrawPolygon erhält hier eine Liste von einzelnen Koordinaten, wobei immer drei
 Koordinaten als x,y,z Gruppen aufgefasst werden
```

Es werden nur Funktionen aufgeführt, die in Drawing nicht vorkommen, oder die sich bei den Übergaben deutlich unterscheiden.

8.2.1 swapBuffers

Header:

```
swapBuffers(self, setName)
```

Funktioniert genauso, wie showDrawingsNamed.

8.2.2 drawAgentAnnotation

Header:

```
drawAgentAnnotation(self, agentTeam, red, green, blue, text)
```

Der wichtigste Unterschied zur gleichnamigen Funktion in der Klasse Drawing ist, dass hier die Nummer des Agenten und dessen Team in einem einzelnen int Übergeben werden. Die Formel zum berechnen des richtigen int-Werts lautet so (Pseudocode):

```
if(leftTeam):
  agentTeam = agentNum - 1
else:
  agentTeam = agentNum + 127
```

Dabei ist zu beachten, dass agentNum nicht größer sein darf als 127.

8.2.3 removeAgentAnnotation

Header:

```
removeAgentAnnotation(self, agentTeam)
```

Auch hier muss die Nummer des Agenten und sein Team in einem int untergebracht werden. Die Berechnung funktioniert genauso, wie bei drawAgentAnnotaion

8.3 Codebeispiel

```
import drawing
import world

def someFunction():
    #create an instance of Drawing. Parameters 0,0 will make it communicate with localhost on standa
    d = drawing.Drawing(0,0)

    #create some vectors
    v1 = world.Vector(1,2)
    v2 = world.Vector(3,4)

    #array to represent a color
    color = [255,255,0]

    #lets use those for drawing
    d.drawLine(v1,v2,3,color,"static.lines")

    #of course you don't have to init vectors and color beforehand
    d.drawCircle(world.Vector(4,5),2,3,[200,155,100],"static.circles")

    #now display everything
    d.showDrawingsNamed("static")
```

Das Ergebnis:

Taktik

Einleitung

Schwarmverhalten war uprspünglich das Grundkonzept der Taktikt des *DAI-Labor*-Teams. Da aber die Ansichten über die Taktik in der Gruppe auseinanderstrebten, wurde im Endeffekt das Modell auf ein hybrides Modell geändert, das sowohl Schwarmverhalten als auch starre Elemente benutzt.

Es gibt einen Tormann (Spieler mit der ID 1), Verteidiger und Stürmer/Mittelfeldspieler, was bei uns keine weitere Unterscheidung hat. Es gibt 5 Verteidiger und 5 offensive Spieler, also insgesamt 11 Spieler, wie in einem normalen Fußballteam

Genauere Erläuterung

Die Klasse `TacticsMain` enthält sämtliche Logik. Im Konstruktor werden nur alle Variablen initialisiert, die wir brauchen.

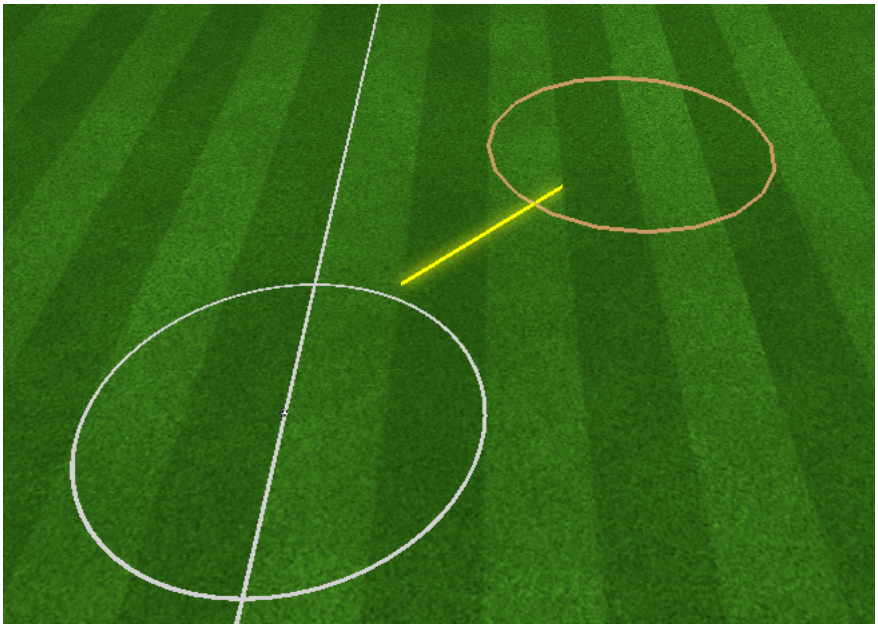

Abbildung 11: Ergebnis

Die eigentlich entscheidende Methode ist `run_tactics`, die extra in jedem Zyklus aufgerufen werden muss, damit die Taktik ihr Tupel mit der jeweiligen Entscheidung zurückgibt, das sogenannte „Entscheidungstupel “, das folgendermaßen aufgebaut ist: `(<run_tuple>, <stand_up_tuple>, <kick_tuple>, <say_tuple>, <head_tuple>)`.
Die anderen Funktionen sind Hilfsfunktionen, die von `run_tactics` aufgerufen werden.

Funktion `run_tactics`

Zuerst werden die Bestandteile des „Entscheidungstupels “mit Default-Werten initialisiert.

Dann wird überprüft, ob der Nao am Boden liegt, wenn ja, setzt er die Rückgabetupel so, dass er sich aufrichtet.

Als nächstes werden mit `clear_distances` alle gespeicherten Distanzen zu anderen Objekten gelöscht, dann wird die eigene Position bestimmt. Schlägt dies fehl, so drückt das „Entscheidungstupel “für diesen Zyklus aus, dass der Nao stillstehen soll und nur den Kopf bewegen soll.

Im nächsten Schritt wird das Ergebnis der Kommunikationsgruppe ausgewertet und die aktuellen Distanzen zu allen anderen Objekten werden mit `get_distances` ausgewertet. Außerdem evaluieren wir den Abstand zu den Randlinien, damit wir das Spielfeld nicht verlassen. Ferner wird noch der Standardwert für das Say-Tupel hier gesetzt.

Nun kommt der wichtigste Teil des Taktik:

- **Wenn der Nao den Ball hat:** Wenn wir aufgrund von `stop_run_to_shott` schießen sollen, dann laufen wir zum Ball.

  Sonst versuchen wir einen Schuss, wenn wir nahe genug am Ball sind.

- **Wenn der Nao ein offensiver Spieler ist, aber gerade den Ball nicht hat:** Wir versuchen nun nicht in verbündete Spieler zu laufen und auch um gegnerische Spieler versuchen wir herumzulaufen, da wir sonst durch das Teleportieren einen unfairen Vorteil hätten.

- **Wenn der Nao ein defensiver Spieler ist, aber gerade den Ball nicht hat:** Wir setzen nun nur `self.offence_member = False`.

Am Ende gibt `run_tactics` dann das „Entscheidungstupel “zurück.

Kurzerläuterung aller Taktik-Funktionen

TODO: Write arguments, return values and what the specific function does.

- `set_own_position`
- `get_distances`
- `calculate_goal_distances`
- `clear_distances`
- `calc_point_distance`
- `calc_line_distance`
- `i_own_ball`
- `offence_Player`
- `enemy_circumvention_behavior`
- `flocking_behavior`
- `calc_turn_angle`
- `search_ball`
- `ball_info`
- `check_lines`
- `run_to_xy`
- `set_own_position`
- `run_tactics`

9 Fazit

Ziel des Praktikums war es, ein Tor zu schießen und einen signifikante Unterschied auf ein Spiel gegen UT Austin Villa im Vergleich zu einem leeren Spielfeld zu haben.
Wir haben es geschafft uns über das Spielfeld zu bewegen, in dem wir uns beamen. Wir können uns auch lokalisieren indem wir feste Objekte benutzen. Diese Information und die daraus berechneten Positionen des Balles und der anderen Roboter werden in das Weltmodell des Agenten eingepflegt. Dabei wird die vergangene Zeit der Aktualisierung beachtet, damit nicht mit veralteten Informationen gearbeitet wird. Die Agenten kommunizieren untereinander, um z.B. die Position des Balles an die Mitspieler zu senden. Wenn die NAOs umfallen bemerken sie das und können von allein wieder aufstehen.
Wir haben ein hybrides Verhalten zwischen Schwarm- und Regelverhalten. Das Schwarm verhalten sorgt dafür, dass die einzelnen Roboter nicht alle auf eine Stelle laufen, sondern versuchen einen bestimmten Abstand zueinander zu halten. Wenn ein NAO am Ball ist, dann beginnt er mit diesem auf des Tor das Gegeners zu laufen. Zum Schluss ist es uns gelungen ein Tor zu schießen.
Während der Implementierung haben sich folgende Probleme ergeben: Beim Beamen wird der Oberkörper des NAOs immer in eine aufrechte Position gebracht, wodurch er beim Fortbewegen nicht umfallen kann. Kolliediert er nun mit einem Gegener, fällt dieser um, wir jedoch nicht.
Auch benötigt der Agent sehr viel Zeit, um zu entscheiden, was er als nächstes tun wird.
Bei der Gruppenarbeit ergab sich ein Problem mit der Organisation. Wir haben oft Aufgaben nicht klar definiert und diese dann nicht direkt jemandem zugewiesen. Auch haben wir uns nicht an die erlernten Konzepte der MPGI3 Vorlesung gehalten, diese unter anderem vielen Stellen nicht anzuwenden, da wir es mit keinem System zu tun hatten, das einem Benutzer gegenüber gestellt wird.
Abschließend bleibt zu sagen, dass alle etwas dazu gelernt haben und die Ziele fast erreicht wurden.

Literatur

[1] https://www.sites.google.com/site/umroboviz/drawing-api.

[2] http://www.naoteamhumboldt.de/de/.

[3] http://www.openbook.galileocomputing.de/python/python_kapitel_20_001.htm.

[4] http://www.simspark.sourceforge.net/wiki/index.php/effectors.

[5] http://www.simspark.sourceforge.net/wiki/index.php/models.

[6] http://www.simspark.sourceforge.net/wiki/index.php/network_protocol.