ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Master-Thesis

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)
Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Schwarmroboter (vorläufiger Arbeitstitel)

vorgelegt von
Sven Manier

betreut und begutachtet von
Prof. Dr. Markus Esch
Prof. Dr. Martina Lehser

Saarbrücken, 10. 05. 2018

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“ zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 10. 05. 2018

Sven Manier

Zusammenfassung

In dieser Thesis wird geprüft ob, und wie, ein Schwarm aus autonomen Robotern in der Lage ist Transportaufträge zu erledigen. Dabei handelt es sich um Aufträge, die mehrere Robotern vorraussetzen und nicht von einzelnen erledigt werden können. Außerdem unterliegen die Roboter dem Schwarmverhalten das von Graig Raynolds beschrieben und skiziiert wurde, das heißt sie orientieren sich am Gruppenverhalten von echten Tieren wie man es in der Natur bei großen Gruppen beobachtet hat. Die Kommunikation innerhalb des Schwarms beläuft sich demnach auf ein Minimum und soll eher fehlende Sensorik und Gestik/Mimik ausgleichen als dass sie für Absprachen genutzt wird.

Um der Frage auf den Grund zu gehen, wurde in iterativen Schritten zunächst ein Verhalten erdacht, prototypisch umgesetzt und anschließend evaluiert, um die nächste Runde der Konzeption zu verbessern. Die Analyse hat letztlich gezeigt, dass Schwarmverhalten während eines Transports unerwünscht ist und Roboter die einer klaren Steuerung folgen deutlich besser sind. Allerdings zeigte sich Schwarmverhalten außerhalb des Transports, also während es Standby-Betriebs oder auf dem Weg dorthin, als durchaus sinnvoll, da sich Roboter ohne große Ressourcen dynamischer und natürlicher in ihre Umgebung einbinden.

Diese Thesis hat somit gezeigt, dass Schwarmverhalten nicht für Transportaufträge selbst geeignet ist, aber durchaus viel Potential zeigt, wenn es generell darum geht, dass sich Roboter einer Umgebung anpassen müssen, ohne zu Störobjekten zu werden.

Inhaltsverzeichnis

1 Einleitung

Diese Thesis wird im Rahmen eines Informatik-Studiengangs mit dem Abschluss eines Master of Science geschrieben. Worum es genau geht und welche Bedingungen ich voraussetze, wird im folgenden Kapitel beschrieben.

1.1 Motivation

Schaut man in die Natur sieht man sehr häufig, dass sich Tiere in riesigen Gruppen organisieren, die sich meist ohne zentralen Koordinator bewegen. Bei Fischschwärmen handelt jedes Individuum vollkommen autonom und nur anhand dessen was es in seiner (unmittelbaren) Umgebung erkennen kann. Dennoch können komplexe Handlungen wie die Flucht vor einem Räuber oder die gemeinsame Wanderung zu einem neuen Futterort koordiniert werden. In manchen Herden dagegen besteht ein 'Schwarm' aus Untergruppen, welche zwar einen Anführer haben, der die Richtung für seine Gruppe angibt, diese Führer sich jedoch am Rest der Herde orientieren und somit ein Schwarmverhalten entsteht, bei dem die Akteure aus koordinierten Einzelgruppen und nicht aus einzelnen Tieren bestehen. Ebenso wird es bei Vögeln vorgefunden, welche durch ihre Koordination in einer energiesparenden Formation fliegen können. Solch ein Schwarmverhalten hat den großen Vorteil, dass jeder Akteur nur für seine eigene Bewegung verantwortlich ist. Der Denkaufwand verteilt sich auf alle Mitglieder und die, meist langsame, Kommunikation untereinander wird auf ein Minimum reduziert.

Man fing früh an dieses Verhalten zu studieren und es auf Roboter anwenden zu wollen. Schwärme von Robotern können in vielerlei Hinsicht eingesetzt werden, darunter Sensornetzwerke, die dynamisch einen bestimmten Raum überwachen oder Arbeitsroboter, die ohne weiteres Zutun bestimmte Tätigkeiten wie Transport übernehmen. Zentrale Systeme mit Koordinator können dafür allerdings schnell unzulänglich werden. Ein großer Nachteil ist schon dadurch gegeben, dass es diesen einen Koordinator geben muss. Fällt er wegen einem technischen Defekt aus, steht der Schwarm still. Baut man einen Backup-Koordinator hat man einen vielfach erhöhten Aufwand beim Design der Schwarm-Architektur, einen größeren Aufwand in der Kommunikation, da der Backup mit einbezogen werden muss, und letztlich auch einen höheren Preis, weil der rechenstarke Computer gleich mehrmals gebaut werden muss. Auch die Reichweite des zentralen Koordinators kann dem Schwarm schnell zum Verhängnis werden. Hat ein Schwarm Beispielsweise die Aufgabe ein Gebiet zu erkunden um später eine Karte aus den Sensorwerten zu erstellen, muss der Koordinator entweder stets in Reichweite aller Roboter sein um mit ihnen kommunizieren zu können oder er muss die Kommunikation durch teures Flooding im Kommunikationsnetz aufrecht erhalten. Bei der Koordination von Drohnen kann allein die Erreichbarkeit zu einem sehr realen Problem werden, da der Koordinator aufgrund seiner Rechenleistung ein vielfaches der Arbeiter wiegen wird. Dies führt zu beträchtlich steigenden Kosten für die Motoren und den Akku des Koordinators, beeinflusst aber auch sein Verhalten, da er deutlich träger ist als die anderen. Wer das Bild eines Fischschwarms vor Augen hat, wird vermutlich auch direkt an die Anzahl der Mitglieder des Schwarms denken müssen. Zentrale Systeme sind nur schwer skalierbar und müssen von Anfang an für die

jeweilige Größe konzipiert werden.

Schwarmverhalten zeichnet sich dadurch aus, dass die einzelnen Mitglieder des Schwarms ihre Bewegung selbst koordinieren. Dabei wird sich entweder nur nach der Bewegung der anderen gerichtet oder es werden sehr lokal Informationen ausgetauscht. Die Rechenleistung wird auf alle Mitglieder gleichmäßig aufgeteilt und jeder ist nur für sich selbst verantwortlich. Durch die lokale Kommunikation stellt die Größe oder Ausbreitung des Schwarms kein Hindernis dar, da ein Roboter nur Verbindung zu seinen nächsten Nachbarn braucht und auch auf teures Flooding verzichtet werden kann. Die Größenordnung des Schwarms spielt dabei auch eine untergeordnete Rolle, da ein Roboter aufgrund des lokalen Informationsaustauschs ohnehin nur wenige Kommunikationspartner hat. Ebenfalls ist solch ein Schwarm ausfallsicherer da eine ausgefallene Einheit (egal welche) letztlich nur sich selbst stoppt.

1.2 Aufgabenstellung

In dieser Thesis befasse ich mich mit der Thematik einen Roboterschwarm dafür nutzen zu können Transportaufträge zu bearbeiten. Ein Auftrag wird an beliebig viele Roboter eines Schwarms verteilt, woraufhin diese selbstständig eine Untergruppe bilden, die für die Ausführung des Auftrages zuständig ist. Da die zu transportierenden Objekte beliebiger Größe und Form sein können, gilt es unter Umständen eine Formation zu bilden und diese bis zum Ende des Auftrags einzuhalten. Dabei gilt es Kollisionen mit anderen Objekten zu vermeiden und selbstständig navigieren zu können. Eine eventuell benötigte Karte ist zunächst voreingestellt, wird im späteren Verlauf aber, von bereits im System der Roboter vorhandenen Funktionen, selbstständig erstellt werden.

Hauptgegenstand dieser Thesis ist die Erarbeitung des Konzepts und die Umsetzung der Koordination bzw. des Verhaltens der Roboter als Algorithmus, sodass dieser produktiv eingesetzt werden kann. Eine Implementierung findet prototypisch auf den Robotern selbst oder einer Simulation dieser statt und dient vor allem dazu festzustellen, ob die theoretischen Überlegungen auch in der Praxis angewendet werden können.

1.3 Aufbau der Thesis

Zunächst werde ich im Kapitel Grundlagen und Hintergründe auf notwendige Grundlagen zum Thema Schwarmverhalten eingehen. Darunter verschiedene Definitionen, Abgrenzungen und Einstiege in für Schwarmverhalten wichtige Themen. Danach folgt eine Analyse der Aufgabenstellung im Detail. Es wird darauf eingegangen, welche Probleme sich bei der Aufgabenstellung auftun und die es zu lösen gilt und welche Fähigkeiten bei den Robotern genau gefordert werden. Im darauf folgenden Kapitel wird bereits vorhandene Forschung zum Thema Schwarmverhalten untersucht. Diese Forschung wird sich nicht ausschließlich um den Bereich der Roboter-Technik drehen, zumal es für das behandelte Thema noch keine ausreichende Literatur gibt. Die Forschung wird darauf geprüft, welche Themenbereiche meiner Thesis bereits abgedeckt werden und welche noch offen sind. Als nächstes folgen die Kapitel der Konzeption, Implementierung und Evaluation. In diesen wird das Konzept für die Roboter zunächst theoretisch, dann praktisch entwickelt werden wird um es anschließend Simulationen zu unterziehen, die das Konzept testen und ein Gefühl dafür geben, wie sich der Schwarm unter bestimmten Vorraussetzungen verhält. Erkenntnisse aus der prototypischen Entwicklung und der Evaluation werden wieder zurück in das Konzept fließen und dieses verbessern. Am Ende der Thesis findet eine Evaluation der erreichten Ziele statt

und, sollten einige nicht erreicht worden sein, eine Erläuterung woran es gelegen hat. Außerdem wird es einen Überblick darüber geben, was mit mehr Zeit und Ressourcen noch hätte erreicht werden können, bzw. wo man Ansetzen könnte um die Entwicklung weiterzuführen.

2 Grundlagen

In diesem Kapitel werden einige grundlegende Begriffe erklärt und voneinander abgegrenzt, welche für das Verständnis dieser Thesis wichtig sind und sonst für Verwirrung sorgen könnten.

2.1 Begrifflichkeiten und Definitionen

Im Verlauf der Thesis werden einige Begriffe immer wieder verwendet werden, die eine feste Definition innerhalb dieser Thesis haben und nicht verwechselt oder missverstanden werden sollten. Solche Begriffe werden im folgenden Abschnitt definiert.

Lokale Nachbarschaft Die lokale Nachbarschaft einer Einheit ist definiert durch einen Kreis mit fixem Radius um die Einheit herum. Andere Einheiten, die sich, bezogen auf ihre Position, innerhalb dieses Kreises befinden, sind Teil der lokalen Nachbarschaft dieser Einheit, Einheiten außerhalb nicht.

Erster Anführer In späteren Kapiteln wird der Anführer als spezielle Rolle eines Roboters eingeführt. Der erste Anführer ist sozusagen der Anführer unter den Anführern und tut Dinge, die nur von einem Roboter im ganzen Schwarm ausgeführt werden müssen.

Tick Ein Tick ist ein Takt im System der Roboter. Die Roboter berechnen ihre nächste Aktion, führen diese aus und warten anschließend auf die Reaktionen der anderen Roboter. Dieser sich ständig wiederholende Zyklus wird Tick genannt.(siehe Abschnitt 6.2)

2.2 Selbstorganisation

Als Selbstorganisation bezeichnet man Prozesse, bei denen aus einer ungeordneten Menge, mit Hilfe von lokaler Kommunikation, ein global geordnetes System entsteht. Es entsteht oft durch zufälliges Verhalten, welches positives Feedback bekommt. In der Robotik versteht man unter Selbstorganisation Gruppen von Systemen, die eigenständig, ohne zentralen Anführer arbeiten können, um gemeinsam ein höheres Ziel zu erreichen. Dies kann ein physikalischer Prozess sein, wie der Transport eines Objektes, es kann aber auch eine 'Denkaufgabe' sein, wie das Berechnen eines Wertes oder die Erstellung einer Karte mit Hilfe von Sensoren die die Umgebung scannen und die Daten der verstreuten Roboter zu einer gemeinsamen Karte bündeln.

Zentrale Systeme

Normale Systeme sind heterogen aufgebaut. Zentralisierte Systeme mit Robotern bestehen grundsätzlich aus einer zentralen Einheit, welche als das Rechenzentrum dient und mehreren Robotern, die die Arbeiter darstellen. In der Praxis kommen schließlich

noch viele weitere Systeme hinzu, die unter Anderem der Ausfallsicherheit und Informationsaufzeichnung dienen. Die zentrale Einheit bekommt von den Arbeitern Informationen wie Sensorwerte zugesendet, die zentrale Einheit berechnet daraufhin das weitere vorgehen und die Aktionen für die einzelnen Arbeiter und sendet sie diesen schließlich zu. Solche zentralisierten Systeme sind wenig skalierbar und sehr kompliziert in der Implementierung, da die additionalen Systeme eingebunden werden müssen. Auch die Komplexität der Nachrichten innerhalb des Netzwerks nimmt zu, da letztlich nicht nur von/zu der zentralen Einheit kommuniziert werden muss, sondern auch die Kommunikation zu Backup, Datenbanken und anderen Hilfssystemen integriert und ausgeführt werden muss.

Selbstorganisierte Systeme

Systeme mit Selbstorganisation sind, im Gegensatz zu den zentralen Systemen, homogen aufgebaut. Jede Einheit ist für sich selbst verantwortlich und muss, gegeben der Abwesenheit der zentralen Steuereinheit, gezwungenermaßen selbst entscheiden, was sie zu tun hat. Durch den homogenen Aufbau und die fehlenden Hilfssysteme gibt es keinen Grund für unterschiedliche Programme/Algorithmen, welche aufeinander abgestimmt werden müssen, sondern man braucht nur ein einziges Programm, welches auf jedem Roboter gleichermaßen eingespielt wird und sich nur durch den darunterliegenden Hardware-Layer und einer einzigartigen Identifizierung von den anderen Unterscheidet. Entscheidungen werden, entweder für sich selbst oder in Gruppen, mithilfe verteilter Algorithmen getroffen. Innerhalb von Abstimmungen oder kleinerer Tätigkeiten kann es zur Bildung von Hierarchien und der Erstellung von Anführern kommen, diese Konstrukte sind aber meist wieder verworfen, sobald die entsprechende Abstimmung oder auszuführende Tätigkeit erledigt wurde. Dadurch dass jede Einheit für sich selbst rechnet und keine permanente Kommunikation zu einer zentralen Stelle notwendig ist, lassen sich homogene Gruppen besser skalieren, wenn auch die Algorithmen für die Kommunikation insgesamt aufwendiger sind.

2.3 Schwarmverhalten

Bei Schwarmverhalten geht es darum, dass sich eine Gruppe von (meist homogenen) Einheiten ohne zentrale Kontrolle gemeinsam organisiert und eine geordnete Bewegung entsteht.

Die 4 Regeln Schwarmverhalten lässt sich grob auf 4 Regeln zurückführen:

1. Zusammenhang: Versuche deinen Nachbarn nahe zu sein
2. Ausrichtung: Passe deine Bewegungsrichtung deinen Nachbarn an
3. Abschottung: Vermeide Kollisionen mit deinen Nachbarn
4. Flucht: Fliehe vor Dingen, die eine potentielle Gefahr darstellen[6]

Die Grundprinzipien von Schwarmverhalten lassen sich grundsätzlich ohne jegliche Kommunikation umsetzen. Sie lassen sich durch reine Beobachtung der Nachbarschaft und entsprechender Reaktion auf das Verhalten der Nachbarn durchsetzen. Verfügen die Einheiten nicht über die notwendige Sensorik um die Nachbarschaft beobachten zu können, lässt sich dies aber durch entsprechende Kommunikation der eignen Werte ausgleichen.

Kommunikation innerhalb eines Schwarm findet, wenn überhaupt, nur mit der Nachbarschaft statt. Experimente mit Drohnen innerhalb eines Vogelschwarms zeigten, dass die Reichweite der Kommunikation recht gering ist und der Informationsfluss mit der Entfernung überproportional abnimmt. Das bedeutet, Informationen werden nie vollständig weitergeben, was dazu führt, dass der Fluss letztlich zum erliegen kommt und eine Information somit nur lokal verfügbar ist.[7]

Wer sich die 4 Regeln anschaut wird auch bemerken, dass es keine Regel gibt, die einen Roboter normal dazu veranlassen würde still zu stehen. Ein Stillstand im Schwarm ist daher immer auf besondere Bedingungen (zum Beispiel fehlender Platz für Bewegung) oder Fehler im System zurückzuführen. Ein Schwarm der wartet, ist gut vergleichbar mit dem Rauschen bei älteren Fernsehern (auch 'Schnee' genannt), bei dem überall zwar Bewegung zu erkennen ist, aber keine die in eine bestimmte Richtung führt. Der Schwarm steht also als Gesamteinheit still, die einzelnen Einheiten bewegen sich aber weiterhin kontinuierlich.[27][26][24]

Vorteile von Schwarmverhalten Ein großer Vorteil von Robotern, die im Schwarm arbeiten, ist das simple Programm. Die 4 Regeln eines Schwarms sind relativ leicht zu implementieren (unter der Annahme dass der Hardware-Layer bereits fertig ist) und auch die Parameter, die letztlich das dynamische Verhalten des Schwarms beeinflussen, sind mittels Simulationen gut zu verstehen und auf die eigenen Wünsche anzupassen. Generell braucht es für die Abarbeitung der 4 Regeln nicht viele Hardware-Ressourcen. Ganz allgemein ließe sich ein Roboter, der nur diese 4 Regeln beachtet, bei heutiger Technik mittels Microcontrollern umsetzen.

Die einzelnen Roboter eines Schwarms lassen sich also ohne größeren Aufwand zusammenstellen und die Software schnell auf andere Systeme anpassen. Dadurch können, je nach Aufgabengebiet, direkt mehrere Roboter in Serie gefertigt werden, wodurch ein großer, dynamischer Schwarm entsteht.

Nachteile von Schwarmverhalten Einer der großen Nachteile ist Thema dieser Thesis: wie steuert man einen Schwarm. Die Logik der Roboter eines Schwarms ist sehr simpel. Wer sich die 4 Regeln des Schwarmverhaltens anschaut, bemerkt, dass ein Schwarm im Grunde genommen nicht gesteuert werden kann, da keine der Regeln ein eingreifen von außen erlaubt. Möchte man es dennoch schaffen einen Schwarm zu steuern, ist die einzige Möglichkeit dazu ihn möglichst so zu beeinflussen, dass der Schwarm nicht mitbekommt, dass er gesteuert wird. Dadurch ist es nicht sehr einfach einen Schwarm dorthin zu bekommen, wo man ihn haben will und die Steuerung ist eher schwammig und ungenau und wenn man Fehler bei der Steuerung macht, wird man Mühe und Not haben ihn wieder auf Kurs zu bringen.

Der Große Vorteil eines Schwarms, sein simpler Aufbau, ist also auch direkt seine größte Schwäche. Auf der einen Seite hat man das simple Verhalten, das durch sehr günstige Hardware und ohne großen Aufwand in der Erstellung der Software realisiert werden kann. Auf der anderen Seite braucht es einen sehr speziellen Umgang mit solch simplen Systemen und die Steuerung des Kollektivs braucht ein ganzes Stück an Arbeit. Man verlagert die Schwierigkeit von der Steuerung einzelner Roboter also auf die Schwierigkeit einen ganzen Schwarm lenken zu müssen.

2.3.1 Bewegung innerhalb des Schwarms

Schwarmverhalten zeichnet sich dadurch aus, dass sich der Schwarm eher passiv bewegt. Beim Vicsek Model[32] zum Beispiel entsteht eine koordinierte Bewegung durch Wiederholung drei simpler Schritte:

1. Berechne die durchschnittliche Ausrichtung innerhalb deiner Nachbarschaft
2. Passe deine Ausrichtung der berechneten Ausrichtung an (+ Zufallsfaktor bestimmter Größe)
3. Bewege dich um x Einheiten nach vorne

Diese 3 Schritte werden immer wieder von den einzelnen Einheiten im Schwarm wiederholt. Nach einiger Zeit werden sich, je nach Wahl der Parameter, einige Schwärme bilden, die sich zusammen bewegen, ohne ihre Bewegungen miteinander absprechen zu müssen.[31]

2.3.2 Abgrenzung: Abgesprochene Bewegung

In der kollektiven Bewegung mit zentraler Steuereinheit berechnet diese die Positionen der einzelnen Arbeiter und teilt ihnen mit, wie sie sich in der nächsten Iteration auszurichten haben. Dadurch ist es leicht möglich eine Gruppe von Einheiten in eine gewollte Richtung zu lenken und die einzelnen Einheiten, sowie die gesamte Gruppe dadurch zu steuern. Solch eine 'abgesprochene Bewegung' lässt sich auch dezentral realisieren. Die einzelnen Mitglieder des Schwarms können ihre Daten den anderen mitteilen und in gemeinsamen Absprachen abstimmen, wohin sich der Schwarm bewegen soll und sich dementsprechend ausrichten. Diese Absprachen erfordern viel Kommunikation und richten sich entgegen typischem Schwarmverhalten.

Im typischen Schwarmverhalten ist die Bewegung, vor allem die Ausrichtung der Bewegung, etwas, dass sich, wie in Vicseks Modell, passiv durch Beobachtung der Nachbarschaft automatisch synchronisiert. Es gibt keinerlei Absprachen innerhalb des Schwarms wohin sich einzelne Einheiten bewegen sollen oder wohin es mit dem Schwarm im gesamten gehen soll. Entsprechend ist es schwer einen Schwarm in eine bestimmte Richtung zu steuern, da man den einzelnen Einheiten nicht einfach mitteilen kann wohin sie sich ausrichten sollen.

2.3.3 Steuern eines Schwarms

Möchte man einen Schwarm dazu bringen sich in eine bestimmte Richtung zu bewegen, darf man es ihm nicht kommunizieren, da es das Paradigma des Schwarmverhaltens brechen würde. Um dieses Ziel zu erreichen, muss man ihn passiv beeinflussen und ihn dazu bringen seine Ausrichtung selbständig in die gewollte Richtung zu ändern.

Anführer innerhalb eines Schwarms

Eine populäre Möglichkeit einen Schwarm zu lenken, ohne direkt mit ihm zu kommunizieren, ist der Einsatz von Anführern ('Leader'). Diese Anführer wissen wo der Schwarm sich hinbewegen soll oder haben einen Auftrag und bewegen sich entsprechend diesem. Im Laufe der Iterationen richten sich die Einheiten immer am Durchschnitt der Nachbarschaft aus, wobei die Anführer eben dies nicht tun, sondern sich entsprechend der Aufträge ausrichten. Dadurch bildet ihre Ausrichtung eine Art Konstante innerhalb des Schwarms die sich nicht relativ zu den anderen verändert. Der

Schwarm nimmt dadurch allmählich die Ausrichtung dieser Konstanten an und er bewegt sich in die Richtung die von dem Anführer (auch mehrere sind möglich) vorgegeben wird. Mit Hilfe der Technik dieser Anführer, lässt sich der Schwarm letztlich steuern ohne dass das Paradigma des Schwarmverhaltens gebrochen werden muss. Ein Vorteil dieser Technik ist zum Beispiel, dass die Berechnung eines normalen Schwarmverhaltens sehr günstig ablaufen kann, wohingegen ein Anführer über weit mehr Wissen verfügen muss. Ein normaler Schwarmroboter braucht die Positionen der anderen Roboter und errechnet über simple Funktionen seinen nächsten Zug. In der heutigen Zeit sind solche Berechnungen selbst mit billigsten Microcontrollern zu bewältigen. Ein Anführer jedoch kann darauf angewiesen sein seine Umgebung zu kennen, sich seine Karte womöglich sogar selbst zusammenbauen zu müssen, und in dieser Karte mit Wegfingungen navigieren zu müssen. Aus diesen Gründen braucht ein Anführer weit größere Ressourcen und ist entsprechend teurer und braucht mehr Energie.[30][5]

Stigmergie

Eine besondere Form des passiven Nachrichtenaustauschs innerhalb eines Schwarm ist Stigmergie. Stigmergie beruht auf passiven Nachrichten die von Einheiten in der Umgebung platziert und von anderen Einheiten wahrgenommen werden. Ein Beispiel von Stigmergie findet man im Tierreich bei Termiten. Diese rollen ihre Schlamm-Kugeln zusammen, platzieren eine Pheromon-Spur oben drauf und lassen sie anschließend zunächst zufällig irgendwo in der Umgebung liegen. Termiten mögen die Pheromone von anderen Termiten sehr und sind gewillt ihre Kugeln auf denen von anderen zu platzieren, wenn diese sich nicht zu weit entfernt befinden. Je mehr Kugeln sich auf einem Haufen befinden, desto attraktiver ist dieser Haufen dafür die nächste Kugel darauf zu platzieren. Auf diese Weise bilden sich die bekannten Termiten-Hügel die wie Spitzen aus dem Boden ragen.[2]

Diese Methode lässt sich auch auf Roboter übertragen. So gab es bereits erfolgreiche Experimente in denen ein Schwarm von Robotern durch Stigmergie dazu gebracht wurde einen Haufen von Frisbees zu sortieren. Dieses Verhalten ist von Ameisen als "brood sorting"bekannt und verzichtet bei der Arbeit vollkommen auf direkte Kommunikation. Die einzelnen Einheiten reagieren dabei nur auf das Umfeld, welches von anderen Einheiten stetig verändert wird und bilden so nach und nach die Frisbee-Haufen mit der entsprechenden Farbe, ohne sich untereinander absprechen zu müssen.[8][1][9]

2.4 ROS

Robot Operating System (ROS) ist ein modulares Framework für die Erstellung von Software für Roboter. Roboter-Software ist schwer zu designen, da man sehr viele Abstraktions-Level hat und viele Sensoren managen muss, die mitunter von eigenen Microcontrollern kontrolliert werden. An diesem Punkt setzt ROS and und bietet ein abstrakted Framework um diese Arbeit in kleinere, einfachere Schritte aufzuteilen und die Arbeitsumegbung von ROS zu abstrahieren. ROS ist dabei kein eigenes Betriebssystem, sondern eine Art System im System. Es kommt mit eigenen Kommandos wie `roscd` oder `rosls`, die nur ROS-Pakete beachten.

2.4.1 Allgemeines

ROS wurde erstmals 2007 veröffentlicht und ist seit dem in der 12. Version erschienen. Das Framework steht unter einer BSD-Lizenz und ist damit frei für jede Art von Projekt. Mittlerweile gibt es über 3000 Software-Pakete die man in der eigenen Software nutzen kann. Darunter Pakete mit denen sich Karten aus einzelnen Bildern zusammenstellen lassen[4].

Die Software die aus dem ROS-Framework entsteht ist kein Monolith, sondern lässt sich am ehesten mit Microservices vergleichen. Viele kleine Programme, die im ganzen zu einem höheren Ziel vereint werden. Diese kleinen Programme (Nodes), lassen sich in den Sprachen C++, Python und Lisp schreiben, wobei jede Node aus einer anderen Sprache hervorgehen kann. Grundsätzlich ist ROS für Ubuntu gebaut und getestet. Andere Betriebssysteme, wie Windows oder Fedora, lassen sich mit Aufwand auch bauen.

Catkin

Als Build-System nutzt ROS ein Programm namens catkin [15]. Catkin ist eine Eigententwicklung von den ROS-Machern und kombiniert CMake-Makros mit Python-Skripten um flexibler zu sein. Catkin ist entstanden, weil es mehrere, unabhängige Build-Targets besser verwalten kann als herkömmlich Build-Systeme. Gerade ROS mit seiner Microservice-Architektur baut im Normalbetrieb sehr viele kleine Programme, die alle unterschiedliche Abhängigkeiten haben können. Da andere Build-Systeme dafür zu langsam und unperformant waren, wurde catkin ins Leben gerufen.

Wichtige Bestandteile

ROS ist ein sehr modulares Framework, dass aus vielen kleinen Komponenten besteht, die alle zusammen ein eigenständiges, durchaus komplexes System aufstellen. Diese werden im folgenden kurz vorgestellt.

Nodes

Die Nodes von ROS sind die keinen Programme, die später Teil der Microservice-Umgebung werden[18]. Sie können einzeln oder durch Roslaunch im Paket gestartet werden. Sie können auch asynchron dazu- und abgeschaltet werden um im laufenden Betrieb mehr dazu zu schalten oder kleine Updates zu machen. Nodes kommunizieren mit Messages, wobei der ROS-Master in diesem Sub-System als Nameserver dient.

Nodes können auch Services bieten. Das bedeutet, dass eine Service-Message empfangen werden kann. Diese Messages werden nicht durch Topics verbreitet, sondern gehen direkt zu einer Node. Sendet eine Node eine Service-Message an eine andere, wird die Node warten bis die Antwort empfangen wurde. Es bietet sich also an diese direkt zu verarbeiten, um den Absender nicht unnötig warten zu lassen.

Messages

Um einzelne Nodes miteinander kommunizieren zu lassen, bedient sich ROS den Messages[4]. Diese werden in ROS ein einem speziellen Format definiert.

Message für einen Service der 2 Zahlen addiert und das Ergebnis zurück gibt

```
int64 A
int64 B
---
int64 Sum
```

Die Definition geschieht in `.msg` oder `.srv` - Dateien. In diesen werden die Parameter definiert, die in der Message enthalten sein sollen. Ist die Message für einen Service gedacht, braucht es noch einen Trennstrich `---`. Unter diesem Trennstrich können dann die Parameter definiert werden, die man von dem Service zurück erwartet, erwartet man keine, bleibt der untere Bereich einfach frei.

Der große Vorteil dieser Struktur ist, dass ROS die Messages, die nur einmal definiert werden müssen, automatisch in die entsprechenden Programmiersprachen übersetzt. So kann eine Message zwischen Python, Lisp und Cpp-Programmen ausgetauscht werden, ohne dass sie für die jeweiligen Programmiersprachen neu implementiert werden muss. Im Falle von Cpp werden die Messages als Header-Only definiert und landen alle (per Default) im gleichen Ordner. Dieser muss also nur in das Projekt eingebunden werden und die Nachrichten stehen damit jeder Sprache ständig zur Verfügung.

Das Message-System arbeitet mit TCP/UDP, benutzt also den Netzwerk-Layer zur Verbreitung der Nachrichten. Dadurch kann ROS nicht nur auf einem einzelnen Computer gestartet werden, sondern ein ROS-System kann problemlos über mehrere Computer und sogar Netzwerke hinweg aufgespannt werden. Baut man also einen Roboter der mit mehreren Computern arbeitet, zum Beispiel einer für je Arme und Beine, müssen diese nicht aufwendig verknüpft werden, es reicht ein Anschluss an einem gemeinsamen Router.

Topics

ROS arbeitet mit seinen Nachrichten, ähnlich einem MQTT-Server auf sogenannten Topics[22]. Will eine Node Informationen verbreiten, zum Beispiel die Position eines Roboters, so erstellt sie ein Topic und andere Nodes können sich dieses abonnieren. Um dies zu tun, fragt sie den ROS-Master wer ein bestimmtes Topic hält. Dieser vermittelt dann die Adresse und die Nodes können miteinander kommunizieren. Verbreitet eine Nodes dann neue Nachrichten über sein Topic, werden diese nicht über einen zentralen Knoten wie den Master kommuniziert, sonder direkt zwischen den Nodes. Der Master ist dadurch entlastet und das System ist ein Stück weit weniger anfällig für Fehler. Außerdem kann der ROS-Master zwischendurch abstürzen, ohne dass die Kommunikation zwischen den Nodes, die ihre Adressen breits kennen, zum erliegen kommen muss.

Topic sind, genau wie in MQTT, gestaffelt, ein Topic kann zum Beispiel `/swarm/robot/position` genannt werden. Im Gegensatz zu MQTT, wo man aber nun `/swarm/*` abonnieren könnte um alle Nachrichten aus diesem Bereich zu bekommen, kann man in ROS derzeit keine Wildcards verwenden. Es müssen alle Nachrichten, die man abonnieren möchte, einzeln abonniert werden. Dies geht durch die Beschaffenheit der Namen als Strings aber immerhin mit Schleifen.

Mit dem Tool `rqt_graph` [19] lassen sich die verknüpften Topics als Graph darstellen um während des Betriebs eine Übersicht darüber zu bekommen, welche Topics veröffentlicht wurden und welche Node diese jeweils abonniert haben.

ROS-Master

Damit die einzelnen Nodes wissen, wo sie ihre Nachrichten hinschicken müssen, gibt es den ROS-Master[16]. Diese spezielle Node muss in jedem ROS-System einmal gestartet werden. Sie dient sozusagen als DNS-Server in der Programmumgebung von ROS. Möchte eine Node eine Nachricht an eine andere senden, hat sie nur den Namen der Node. Sie fragt dann den ROS-Master, der ihr die entsprechende Adresse zurück gibt. Aber nicht nur Namen werden vermittelt, sondern auch Topics, wodurch diese dezentral laufen. Abbildung 2.1 zeigt den ROS-Master bei seiner Tätigkeit.

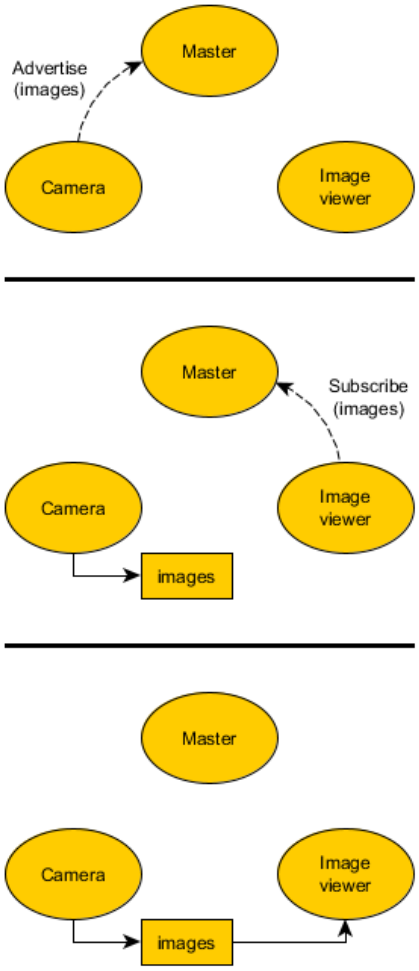

Abbildung 2.1: Der ROS-Master als Nameserver

Parameter-Server

Als letztes Modul wird der Parameter-Server vorgestellt[20]. Dieser dient sozusagen als Repository in dem Konfigurationen abgespeichert werden können. Möchte man bestimmte Werte, wie zum Beispiel Preise bei einer Ware, nicht in seinen Programmen hartkodieren um sie flexibel zu halten, kann man sich dem Parameter-Server bedienen. Die Struktur der Parameter ist wie die der Topics `/camera/left/lens`. Es lassen sich aber ganze Level abfragen, die dann als JSON-ähnliches Format versendet werden.

Der Parameter-Server ist ein Bestandteil des ROS-Masters. Das Abfragen der Parameter setzt also voraus, dass eine Verbindung zu diesem besteht. Entschließt man sich also dazu den Parameter-Server zu nutzen, rückt der ROS-Master wieder ein Stück weiter in den Vordergrund und seine Rolle als kritisches System wird umso stärker. Außerdem erfordert das Abfragen eines Parameters, dass man die TCP/UDP-Verbindung nutzt. Einen Parameter abzufragen ist somit vergleichsweise sehr langsam. Sinnvoller wäre es einen Cache in die eigenen Nodes einzubauen und diesen, entweder über Timer oder Broadcasts über einen Topic manuell, aktualisieren zu lassen.

Roslaunch

Roslaunch ist ein Programm dass es erlaubt mehrere Nodes miteinander zu verketten und als ein Gesamtpaket zu starten[25]. Die Pakete werden in einem XML-Format erstellt und bieten besondere Funktionen um die Nodes miteinander zu bündeln und interagieren zu lassen.

Es ist so möglich Nodes mit Parametern[1] zu starten und sie zum Beispiel als `required` zu kennzeichen. Dies bedeutet, dass alle anderen Nodes die mit dieser im Paket gestartet

[1] `http://wiki.ros.org/roslaunch/XML/`

wurden, sich automatisch abschalten, wenn die betreffende Node sich (freiwillig oder durch Fehler) beendet. Auch ein umleiten von In- und Output eines Services ist möglich. So lässt sich zum Beispiel ein Logger recht simpel in einer eigenen Node erstellen, der eine Kopie der privaten Inputs und Outputs eines Services kommt um sie zu Loggen. Auch für die Fehlerprüfung ließe sich diese Technik nutzen. Man könnte eine alte Version einer Node mitlaufen lassen und auf diese Weise prüfen, dass die neue Version sich genau so verhält wie die alte (aber performanter ist).

Turtlesim

Die Turtlesim[23] (Abbildung 2.2) ist ein ROS-Package das eine Simulationsumgebung bereitstellt. Sie stellt eine grafische Oberfläche zur Verfügung und eine steuerbare Einheit namens 'Turtle'. Diese Turtle kann über die Turtlesim mit einfachen Befehlen in Form von Messages gesteuert werden.

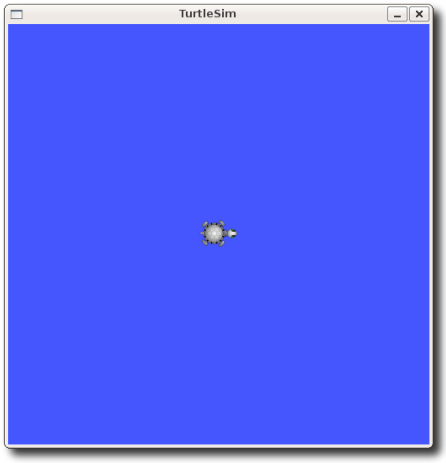

Abbildung 2.2: Die Turtlesim

- turtleX/cmd_vel

  (geometry_msgs/Twist)

- turtleX/teleport_absolute

  (turtlesim/TeleportAbsolute)

- turtleX/teleport_relative

  (turtlesim/TeleportRelative)

Die Turtlesim kennt noch viele Nachrichten mehr und bietet sowohl Schnittstellen für Messages sowie Services. Sie selbst bietet auch selbst Messages an, darunter Feedback darüber, wo sich eine Turtle befindet.

Eigenschaften

Die Turtlesim bietet ein Fenster mit einer 'Spielfläche' auf der mehrere der Turtles erstellt werden können. Die erste Turtle wird beim öffnen der Turtlesim automatisch erstellt und trägt den Namen 'Turtle1'. Weitere Turtles können mit einem eigenen (einzigartigen) Namen erstellt werden oder per Default, wobei die Turtlesim dann den Namen erstellt.

Mit `geometry_msgs/Twist.msg` ist es Möglich eine Turtle in der Turtlesim eine bestimmte Strecke fahren zu lassen. Der Fahrbefehl nimmt eine Geschwindigkeit und einen Winkel an, wodurch auch Kurven gefahren werden können. Eine Turtle fährt immer genau 1 Sekunde lang. Die Geschwindigkeit bestimmt demnach auch direkt die gefahrene Strecke der Turtle. Nachdem ein Fahrbefehl ausgeführt wurde, gibt die Turtlesim über `turtlesim/Pose.msg` die neue Position der Turtle zurück.

Die Größe der Turtlesim beträgt 11.088889*11.088889 Einheiten, wobei dieser Wert experimentell ermittelt wurde, indem eine Turtle gegen die Wand gefahren und der Wert des Feedbacks aufgezeichnet wurde, und hat den Start seines Koordinaten-Systems in der linken, unteren Ecke.

Einschränkungen

Der eigentliche Sinn der Turtlesim ist es ROS-Anfängern eine Möglichkeit zu geben erste Erfahrungen mit ROS zu sammeln. Eine grafische Oberfläche mit einer einfach zu steuernden Turtle ist dabei auch der richtige Weg. Allerdings ist die Turtlesim nicht für mehr gedacht und kann aus diesem Grund auch nicht mehr. Es ist nicht möglich Dinge einzeichnen zu lassen, beispielsweise Rechtecke. Es ist auch nicht möglich die Größe zu ändern oder die Skalierung der Fläche anzupassen, sodass die Turtles kleiner und langsamer sind.

Nutzung in der Thesis

Trotz dessen, dass die Turtlesim sehr eingeschränkt ist, hat sie dennoch noch den Vorteil eine einfache, grafische Oberfläche zu bieten. Zu sehen wie die Turtles über die Simulation fahren und sich verhalten ist sehr hilfreich das generelle Verhalten zu beobachten und Fehler in der Implementierung zu erkennen. Die gesammelten Daten sind zwar wichtig als Diskussionsgrundlage, das dynamische Verhalten beobachten zu können bietet aber viel Kontext zu den Daten. Eine eigene grafische Oberfläche zu schreiben hätte weiteren Arbeitsaufwand bedeutet der nicht in die Konzeption der eigentlichen Frage der Thesis hätte einfließen können.

Die Turtlesim wurde allerdings dahingehend angepasst, dass die Bilder der großen Turtles durch kleinere, minimale Pfeile ersetzt wurden. Dadurch wurde mehr übersichtlichkeit geschaffen und die Turtles können direkt beisammen sein, ohne dass sich ihre Darstellung gegenseitig zu sehr behindert.

3 Analyse

Im folgenden Kapitel wird es eine Analyse der Aufgabenstellung geben, darunter die Analyse der Probleme die mit der Aufgabenstellung einher gehen und die Anforderungen des späteren Systems.

3.1 Problembeschreibung

Laut Aufgabenbeschreibung (Abschnitt 1.2) ist das Ziel dieser Thesis, mit Hilfe von Einheiten, die im Schwarm agieren, Transportaufträge zu verrichten. Als Einschränkung war gegeben, dass die Einheiten nicht erst den Auftrag verteilen müssen, sondern die entsprechenden Einheiten ihn bereits haben und die Verteilung des Auftrags, bzw. die Auswahl der geeignetsten Einheiten somit bereits abgeschlossen ist.

Die nachfolgenden Abschnitte bilden den Verlauf des typischen Transports ab und zeigen die verschiedenen Herausforderungen die es dabei zu bewältigen gibt und die Probleme die dabei auftraten können.

Allgemeine Probleme

In diesem Abschnitt sind die Probleme aufgelistet, die generell in jeder Phase des Transports auftreten können.

Erlaubtes Gelände

Generell ist immer darauf zu achten, dass es Gebiete gibt in denen der Schwarm sich bewegen darf und solche, zu denen er keinen Zutritt hat. Dabei geht es nicht unbedingt darum gefährliche Orte, und damit Beschädigung, zu vermeiden, sondern auch darum dass er nicht überall erwünscht ist. Lässt man den Schwarm Transportaufträge innerhalb eines offenen Geländes erfüllen, besteht die Möglichkeit dass er es auf der Suche nach dem kürzesten Weg kurzerhand verlässt und damit auch Eigentumsgrenzen. Andererseits kann es aber auch Orte geben die beispielsweise besonders sauber bleiben sollten und in denen die Einheiten mit ihren (eventuell) dreckigen Rädern nicht erwünscht ist. Idealerweise lassen sich solche Bereiche dynamisch in die Gebietskarte einzeichnen oder vor Ort markieren, sodass auch auf Gebiete geachtet werden kann die erst kürzlich zu einer Gefahrenzone wurden, wie beispielsweise eine ausgelaufene Flüssigkeit.

Gruppe finden

Die neue Gruppe die für den Auftrag zuständig ist muss sich selbst finden. Die Vergabe des Auftrags schließt nicht mit ein den Einheiten Erklärungen mitzugeben wie sie zu arbeiten haben, sondern lediglich die Daten für den Auftrag, darunter wie die Ware aussieht, wo sich der Aufnahmeort befindet und wo der Abgabeort. Nachdem eine Einheit also festgestellt hat dass er einen neuen Auftrag besitzt, müssen die anderen Gruppenmitglieder innerhalb des Netzwerks gefunden werden um über die Dauer des

Auftrags hnweg Informationen untereinander auszutauschen. Eine schnelle Möglichkeit sich zu finden könnten IDs darstellen mit man in Zusammenhang mit einer Auftrags-ID stellt. Auf diese Weise könnten sich Roboter eindeutig identifizieren und einem Auftrag zuordenen, sodass auch andere Roboter erkennen, ob zusammengearbeitet werden muss oder nicht.

Standby im Schwarm

Ohne Auftrag gehen die Einheiten ihrem typischen Schwarmverhalten nach. Praktisch heißt das, dass sie sich bewegen und dabei die 4 Regeln von Schwarmverhalten (Abschnitt 2.3) beachten. Die ständige Bewegung mag aus energietechnischer Sicht nicht die optimale Lösung darstellen, da ein Schwarm selbst aber nie still steht, bildet sie aus Sicht eines Schwarms letztlich die Standard-Lösung für 'warten'. Ist der Schwarm im Standby, hat also keinen Auftrag, muss der Faktor des Rauschens so gewählt werden, dass die einzelnen Einheiten des Schwarm zwar noch in Bewegung bleiben, der Schwarm als großes ganzes aber keine Bewegung zu erkennen gibt und augenscheinlich auf der Stelle steht.

Dichte des Schwarms

Da sich der Schwarm auf einem Gelände, wahrscheinlich sogar innerhalb eines Gebäudes oder einer Fabrikhalle, befinden wird, muss auch darauf geachtet werden, dass der Schwarm während des Wartens kein Hindernis für andere Beschäftigte oder Fahrzeuge darstellt. Er sollte im Hintergrund verschwinden und kein aktiver Teil der Umgebung werden auf den im besonderen Maße geachtet werden muss. Der Schwarm sollte sich daher nirgendwo (partiell) örtlich zusammenballen, sondern, wenn man ihn sich als einen Festkörper vorstellt, seine Dichte so gering wie möglich werden lassen, indem er sich auf dem (ihm erlaubten) Gelände so gut es geht ausbreitet. Ein möglicher Ansatzpunkt hierfür wäre es, auf der Regel SZusammenhang: Versuche deinen Nachbarn nahe zu seinänzusetzen. Mit entsprechenden Parametern könnte es möglich sein, dass sich Roboter nicht versuchen nahe zu sein, sondern sich möglichst voneinander zu entfernen. Ein anderer Ansatz für dieses Problem könnte ein Algorithmus sein, der die Größe der Roboter dynamisch verändert. Wie Luftballons in einem Sack, die gleichzeitig mit Luft gefüllt werden, würde jeder Roboter somit den Platz finden, der am weitesten weg ist, von der Mitte der anderen Roboter.

Bildung der Untergruppe

Es muss beachtet werden, dass die neue Untergruppe nach wie vor Teil des großen Schwarms ist und alle Regeln des großen Schwarms erbt, aber auch neue dazu bekommen kann. Auch wenn die Roboter einen speziellen Auftrag haben, sollten sie davon so wenig wie nur möglich mitbekommen. Neue Regeln sollten so wenig wie möglich eingesetzt werden und die Änderung für neue Regeln in den System der Roboter sollten so gering wie möglich ausfallen. Die Roboter sollten von den 4 Hauptregeln (siehe Abschnitt 2.3) so wenig wie möglich abweichen um das Verhalten so natürlich wie nur möglich zu belassen.

Darüber hinaus muss während der gesamten Zeit beachtet werden, dass die Roboter für den neuen Auftrag nur eine Untergruppe bilden und nach wie vor auf die anderen Roboter beachtet werden müssen.

Zusammenfinden am Aufnahmeort

Nachdem die Einheiten ihren Auftrag erhalten haben, gilt es sich am Aufnahmeort zusammenzufinden. Die Einheiten waren gerade noch in Standby ein Teil des Schwarms und müssen ab sofort eine Untergruppe innerhalb des Schwarms bilden, die für den Auftrag zuständig ist.

Als Untergruppe müssen sie sich dann am Aufnahmeort zusammenfinden. Das Problem in dieser Situation ist, dass die Untergruppe nach wie vor als Schwarm agieren muss. In Abschnitt 2.3.2 wurde bereits erläutert, dass Schwarmverhalten nicht mit 'Abgesprochener Bewegung' verwechselt werden darf und Einheiten innerhalb eines Schwarms sich nicht aktiv darüber absprechen dürfen, welche Einheit sich wohin bewegen soll. Das bedeutet, dass die Untergruppe sich ausschließlich durch passive Beeinflussung und die Informationen des gegebenen Auftrags gemeinsam am Aufnahmeort zusammenfinden muss. Natürlich darf dabei aber nicht davon ausgegangen werden, dass die Gruppe die den Auftrag erhält auch eine erkennbare Gruppe war und die Einheiten daraus örtlich nahe zusammen waren, sondern es muss davon ausgegangen werden, dass die Einheiten zufällig ausgewählt werden können.

Einnehmen einer Formation

Damit die Ware sicher gelagert ist, muss die Untergruppe eine Formation einnehmen, die für den Transport der Ware geeignet ist. Beispiele für solche Formationen wären z.B. ein Viereck, wenn eine große viereckige Kiste transportiert werden muss oder eine Linie, sollte ein Stahlträger transportiert werden müssen. Wichtig für die Formation ist jedoch nicht nur die Form der Ware, sondern auch die Anzahl der Einheiten die für den Transport zugeteilt wurden. So macht es bei einer Kiste für vier Einheiten am meisten Sinn, wenn sich die Einheiten an den vier Ecken/Kanten verteilen. Eine fünfte Einheiten würde zunächst in der Mitte am meisten Sinn machen, wohingegen es bei sechs Einheiten wieder mehr Sinn machen würde, wenn die Einheit aus der Mitte zu einer Ecke/Kante wandert die noch nicht belegt ist und der sechste dann gegenüber.

Das Einnehmen einer Formation ist für einen Schwarm eine besondere Herausforderung, weil sich Schwarmverhalten signifikant von abgesprochener Bewegung (siehe Abschnitt 2.3.2) unterscheidet. Ein schneller erster Gedanke wie Einheiten eine Formation einnehmen, wäre die Kontur der Ware zu zeichnen und die Einheiten an den signifikanten Stellen, möglichst symmetrisch, zu verteilen. Für solch eine Koordination braucht es aber letztlich auch einen Koordinator. Einen Koordinator zu wählen, wäre für die Untergruppe grundsätzlich kein Problem, ein einfacher Echo-Algorithmus wäre vollkommen ausreichend, da dies jedoch über die Kommunikation innerhalb eines Schwarms weit hinaus geht, ist diese Form der Verständigung nicht erlaubt. Es muss für dieses Problem also ein Algorithmus gefunden werden, mit dessen Hilfe sich ein Schwarm in eine bestimmte Formation bringen lässt, ohne dass es einen Koordinator gibt der den einzelnen Einheiten mitteilt wo sie sich hinbewegen müssen. Grundlage für diese Formation muss ein simpler Austausch von Informationen sein, darunter vor allem die Größe der Gruppe.

Stillstand am Aufnahmeort

Am Aufnahmeort angekommen, müssen die Einheiten still stehen bleiben. Der Grund dafür ist, dass es eine gewisse Zeit dauert bis ein Mitarbeiter oder eine andere Maschine die zu transportierende Ware auf die Einheiten verladen hat. In dieser Zeit darf die Untergruppe nicht in den typischen Standby des großen Schwarms verfallen, denn dabei

würden sich die Einheiten leicht bewegen und das verladen schwierig machen. Statt dessen müssen sie eher in eine Art Starre verfallen, bis ein bestimmtes Ereignis eintritt, welches ihnen signalisiert dass die Ware vollständig verladen wurde und sie ihren Weg antreten können. Ein solcher Stillstand könnte erreicht werden, indem die Motoren der Roboter schlicht über Software außer Betrieb genommen werden würden. Aber auch Schwarmtechnische Lösungen wie das Arbeiten mit Gefahrenzonen, die gerade groß genug sind um den Roboter darin stehen zu lassen, wären denkbar.

Gemeinsamer Transport zum Abgabeort

Nachdem die Ware erfolgreich auf die Einheiten verladen wurde gilt es nun diese an ihren Abgabeort zu transportieren und den Transportauftrag dadurch zu beenden.

Halten der Formation

Wichtig beim Transport der Ware ist, dass die Formation möglichst genau beibehalten werden muss. Ein abdriften der Formation könnte schnell dazu führen, dass die zu transportierende Ware herunterfällt und beschädigt wird oder dabei andere Gegenstände, insbesondere die Einheiten selbst, beschädigt werden. Ebenso ist eine Beschädigung der Einheiten oder der Ware möglich, wenn die Ware rutschfest auf den Einheiten liegt oder gar festgeschraubt wurde. In diesem Fall könnte das verlassen der Formation dazu führen, dass die Einheiten versuchen in eine Richtung zu fahren, aber von der Befestigung daran gehindert werden. Die Ware oder die Einheiten könnten durch die Zugkräfte beschädigt werden, insbesondere könnten die Einheiten umfallen oder die Motoren die für den Antrieb zuständig sind überhitzen und die Einheit ausfallen lassen.

Wieder ist darauf zu achten, dass die Formation ohne Koordinator eingehalten werden muss, allein durch Regeln die der Untergruppe als Schwarm allgemein inne herrschen. Die gleichen Algorithmen die dazu führten, dass die Untergruppe am Aufnahmeort ihre Formation einnahm, müssen nun dazu verwendet werden um die Formation während des Transports zu halten. Dabei ist ein ständiger Check erforderlich, ob sich die Roboter noch im Rahmen der Toleranz befinden. Bestenfalls wird dies direkt bei der Berechnung der Bewegung beachtet, sodass es von vornerein nicht möglich ist, dass ein Roboter 'ausversehen' die Formation verlässt.

Maße des Transportobjektes

Auch die Ausmaße des Transportobjektes müssen beim Transport beachtet werden. Es muss nicht immer der Fall sein, dass die Roboter das Transportobjekt vollständig umspannen. Im einfachsten Fall hat man nur einen Roboter, der in der Mitte sitzt, der ein Objekt transportieren muss, dass im Durchmesser ein Vielfaches seiner Größe hat. Der Roboter hat auf dem Weg zum Abgabeort darauf zu achten, dass das Objekt nicht mit seiner Umgebung kollidiert und damit von seinem Kopf fällt oder beschädigt wird. Dies ist besonders dann wichtig, wenn der Roboter in der Nähe anderer Menschen arbeitet, die dadurch von dem Transportobjekt verletzt werden könnten. Doch ist nicht nur der Abstand selbst ein wichtiger Faktor, sondern auch die Breite von Gängen. Es kann bei breiten Werkstücken dazu kommen, dass der Roboter eine Route meiden muss, für die er selbst groß genug wäre, in der das Transportobjekt aber stecken bleiben würde.

Bei vielen Robotern wäre es eine Möglichkeit die Roboter so zu setzen, dass die das Transportobjekt möglichst umspannen, ein Zusammenstoß der Umgebung mit dem Transportobjekt also gleichbedeutend wäre mit einem Zusammenstoß mit einem Roboter.

Andere Lösungen schließen mit ein, die Roboter über die Ausmaße des Transportobjektes in Kenntniss zu setzen, was aber widerum neue Fähigkeiten vorausstetzen würde und das Wissen darüber, dass sie sich gerade in einem Transport befinden.

Standby am Abgabeort

Am Abgabeort angekommen muss die Untergruppe wieder zum Stillstand kommen und solange in einer Art Starre verharren bis die Ware vollständig umgeladen wurde. Die Starre muss erneut anhalten bis ein bestimmtes Event eintritt.

Anschließend muss die Untergruppe wieder in den Schwarm eingegliedert werden. Das heißt, die neuen Regeln die für die Erfüllung des Auftrags notwendig waren fallen nun weg und es werden wieder die Regeln des großen Schwarms übernommen, wie zum Beispiel das Verteilen um anderen Arbeitern nicht im Weg zu stehen.

3.2 Anforderungen an das System

Nachdem im vorigen Abschnitt Herausforderungen anhand des generellen Verhaltens der Einheiten dargestellt wurden, wird nun auf die Anforderungen des Systems selbst eingegangen.

Unabhängigkeit

Die einzelnen Einheiten des Schwarms oder der Untergruppen die gerade einen Auftrag erledigen müssen vollständig unabhängig agieren und dürfen nicht aktiv von einer anderen Einheit geleitet werden. Dies betrifft auch eventuelle Anführer die innerhalb der Untergruppen gewählt werden. Diese mögen zwar eine Sonderrolle einnehmen, dürfen die anderen Einheiten der Untergruppe aber nur passiv beeinflussen und ihnen nicht explizit mitteilen, wo sie sich hinzubewegen haben.

Ausfallsicherheit

Einer der besonderen Vorteile des Schwarms ist es, dass die Einheiten generell unabhängig agieren und nicht auf die Kommunikation mit anderen Einheiten angewiesen sind. Daraus resultiert dass ausgefallene Einheiten kein generelles Problem für den Schwarm sind. Betrachtet man einen Schwarm im allgemeinen, so ist eine ausgefallene Einheit vergleichbar mit einem Stein der herumliegt; er kommuniziert nicht und er bewegt sich auch nicht mehr. Er spielt also im Schwarm keine große Rolle mehr und dieser kann ungehindert weiter agieren, nur eben mit einer Einheit weniger.

Im besonderen Fall des Auftrags sind die Einheiten allerdings im gewissen Maße abhängig voneinander. Gerade wenn es um die Bildung der Formation geht, spielt die Größe der Untergruppe eine wichtige Rolle, da sie mitunter bestimmt wo sich die einzelnen Einheiten aufzuhalten haben. Sollen 2 Einheiten eine Stange bewegen und eine fällt unterwegs aus, fällt die Ware aufgrund des fehlenden Gleichgewichts zu Boden. Es muss daher darauf geachtet werden, dass die Untergruppe in ständigem Kontakt untereinander steht um sicherzugehen dass alle Einheiten nach wie vor aktiv und einsatzbereit sind. Fällt plötzlich eine Einheit aus, muss darauf reagiert werden indem die Formation entsprechend angepasst wird oder der Auftrag abgebrochen und eine Fehlernachricht an ein zuständiges System gesendet wird.

Auch im Sinne von möglichen Anführern ist die Ausfallsicherheit ein Problem für sich. Wird ein Auftrag von 5 Robotern + 1 Anführer ausgeführt und dieser fällt aus, wird der

Schwarm nicht mehr gelenkt und wandert Ziellos umher, treibt möglicherweise sogar auseinander und bringt so das Transportobjekt zum fallen. Fällt ein Anführer aus muss dieser entweder von einer leitenden Stelle, zum Beispiel der Auftragsvergabe, umgehend ersetzt werden oder ein bereits beteiligter Roboter wird zum Anführer 'befördert' und lenkt den Schwarm von nun an. Auch ein Abbruch des Auftrags wäre möglich, indem die Roboter auf direktem Wege wieder zum Aufnahmeort zurück kehren (dies könnte beispielsweise mit einer Neuvergabe gelöst werden).

Örtlichkeit

Ein Schwarm zeichnet sich nicht nur dadurch aus, dass die Einheiten unabhängig voneinander sind, sondern auch dadurch, dass Nachrichten zur Kommunikation nicht den gesamten Schwarm belasten. Sendet eine Einheit eine Nachricht an seine Nachbarschaft (siehe Abschnitt 2.1) aus, so darf diese nicht den gesamten Schwarm durchqueren, sondern muss mit der Zeit 'kleiner' werden und letztlich verschwinden. Dies hat nicht nur den Grund näher am Vorbild der Natur zu sein sondern ist auch deshalb notwendig, da die Nachrichten von mehreren hundert Einheiten sonst das Netzwerk lahmlegen würden und die Einheiten nur noch damit beschäftigt wären die Nachrichten zu verarbeiten und weiterzuleiten. Die Ressourcen für die eigentliche Aufgaben würden fehlen und der Schwarm würde still stehen.

Effizienz

Die Algorithmen die auf den Systemen der Roboter laufen müssen möglichst simple und effizient sein. Diese Anforderungen kommen einerseits daher, dass die Roboter sich möglichst natürlich verhalten sollen und Tiere sich bei genauerer Betrachtung meist sehr effizient verhalten. Außerdem berechnen Tiere nicht wie sie ihre Gruppen anordnen. Das Verhalten geschieht meist aus Instinkt heraus und folgt sehr einfachen Regeln. Sollen Roboter die Schwärme der Tiere imitieren, müssen sie auch das Verhalten einzelner Mitglieder imitieren.

Zum anderen sind Roboter aufgrund ihrer Hardware und ihrer Anzahl sehr teuer. Wird ein Roboter-Schwarm innerhalb eines Betriebes genutzt, kann deren Anzahl von wenigen bis zu hunderten reichen. Das Projekt um die Kilobots herum[13] hat gezeigt, dass sich Roboter in großer Menge produzieren lassen, aber dass sie dafür auch entsprechend simple in der Hardware beschaffen sein müssen. Eine simple Hardware verlangt nach einer ebenso simplen Software um diese nicht zu überfordern.

4 Stand der Technik

In diesem Kapitel werden kurz einige verwandte Arbeiten vorgestellt, die eine Grundlage für diese Thesis bildeten.

4.1 Grundmodell

Craig Reynolds hat als einer der ersten versucht einen Schwarm mit einfachen Mitteln darzustellen. Er modellierte einen Schwarm mit Vögel im 3dim-Raum. Dazu stellte er 3 Regeln auf mit denen ein Schwarm sein Verhalten berechnet:

- Separation: steer to avoid crowding local flockmates alignment diagram
- Alignment: steer towards the average heading of local flockmates cohesion diagram
- Cohesion: steer to move toward the average position of local flockmates

Auf dieser Grundlage wurde in den nachfolgenden Jahren viel Forschung betrieben und das Verhalten der heutigen Roboter-Schwärme ausgearbeitet die sich an der Natur orienten.[3]

4.2 Erweiterung des Modells

Vicsek hat in seinen weiteren Forschungen das Modell seiner Vogel-Schwärme erweitert. In einer seiner Veröffentlichungen erweitert er das Modell der schwarmgetriebenen Partikel dahingehend, dass nicht nur die Ausrichung der anderen Roboter entscheidend ist, sondern auch deren Geschwindigkeit. Dabei konnten neue Muster in den Schwärmen erkannt werden, die in Abhängigkeit der Variable s standen. s ist dabei der Wert, der angibt, wie sehr sich Einheiten an der Geschwindigkeit anderer Einheiten orientieren im Intervall von [0,1]. Ab einem Wert von 0.59 stieg die Bindung des Schwarms rapide an, ähnlich wie man es bisher bei einem kritischen Wert des freien Willens kannte.[29]

4.3 Informierte Anführer

Da ein Schwarm nicht direkt gelenkt werden kann, werden mittel gesucht ihn mit seinen eigenen Mitteln zu lenken. Ein Versuch dabei ist das Vorgehen mit 'Informierten Anführern'. In dem Paper wurden einige Roboter über das Ziel informiert, zu dem der Schwarm gelenkt werden soll. Diese informierten Anführer lenkten daraufhin den Schwarm dadurch, dass sie eine konstante (und einigermaßen gleiche) Ausrichtung hatten, an der sich die Roboter während ihres normalen Schwarmverhaltens orientierten. [5]

4.4 Strukturen

Die folgenden Arbeiten befassen sich insbesondere damit, dass Roboter vollkommen autonom Strukturen erstellen, die vorher definiert und dem Schwarm in irgendeiner Weise mitgeteilt wurden.

Kilobots

In einem Paper von Harvard wird demonstriert, wie Roboter (’Kilobots’) eine vordefinierte Fläche eigenständig ausfüllen. Dafür wurde ein "Bild"genommen, das den Robotern dient um die Fläche zu erkennen. Die Roboter kommunizieren nur wenig miteinander. Sie sind in der Lage ein Koordinatensystem aufzuspannen und ihre Position innerhalb dessen zu erkennen. Außerdem können sie sich selbst einen ’Gradienten’, eine Art numerische Einstufung zuteilen und diese mit anderen mitteilen. Als letzte Fähigkeit können sie einer Kante folgen. Sie sind damit in der Lage den Rand eines bereits bestehenden Roboter-Haufens entlang zu gehen ohne mit ihnen zusammenzustoßen.

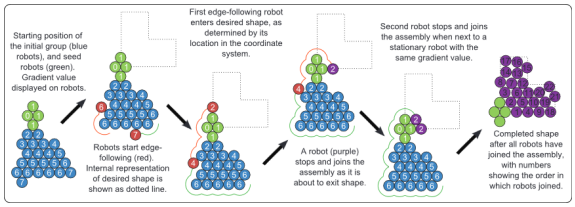

Abbildung 4.1: Struktur-Erstellung mit Kilobots

Gestartet wird der Algorithmus mit einem SSeed", einem Roboter der der die Koordinate (0,0) bildet. Die anderen Roboter kommen nach und nach dazu und hangeln sich die bereits stehenden Roboter entlang. Sein Gradient wird dabei ständig angepasst. Dieser wird auf den kleinsten Gradienten in seiner Umgebung + 1 gesetzt, während der Seed immer auf 0 bleibt. Verlässt ein Roboter das Bild oder stößt auf einen anderen Roboter mit dem selben Gradienten, bleibt dieser stehen.[13] Der Algorithmus ist in Abbildung 4.1 zu sehen.

Strukturen mit Bausteinen

Mit einem anderen Ansatz ging man bei dem folgenden Paper an das Problem der Erstellung von Strukturen. Statt die Struktur mit den Robotern selbst aufzubauen, wurde sie stattdessen mit Bausteinen aufgebaut, die von Robotern zu ihrem ort getragen wurde. Die Roboter selbst wurden dabei genau wie die Steine zufällig auf dem Gebiet ausgeworfen. Anschließend wurde ein Beacon erstellt, der die Position markiert, an der die entsprechende Struktur aufgebaut werden soll.

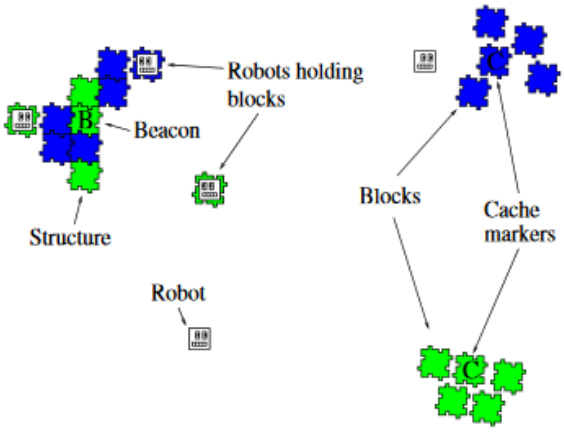

Abbildung 4.2: Struktur-Erstellung mit Bausteinen

Die Roboter selbst können sich im 2dim-Feld in jede beliebige Richtung bewegen. Die Kommunikation der Roboter beschränkt sich darauf Kollisionen zu vermeiden. Das besondere an diesem Paper ist, dass die Roboter selbst nicht die Struktur kennen die gebaut werden soll, sondern die Bausteine. Diese werden

von den Robotern getragen und teilen dann den Robotern mit, wenn sie sich an einem geeigneten Platz befinden um abgesetzt zu werden.[33] Der Aufbau ist in Abbildung 4.3 zu sehen.

4.5 Transport

Am *Intelligent Robotics Research Centre, Monash University* beschäftigte man sich damit, wie Bienen andere tote Bienen aus einem Nest befördern. Bienen sondern bei ihrem Tod ein Pheromon ab, dass andere Bienen dazu veranlasst, den Leichnam aus dem Bau zu stoßen.

Im Experiment wurden Roboter vorgestellt die mit einem Gas-Sensor ausgestattet sind und ein kleiner, spezieller Roboter, der in der Lage ist eben solches Gas auszustoßen. Die Roboter suchen permanent nach der Quelle des Gases und fahren dabei in Kreisen, um die Quelle zu lokalisieren. Ist die Quelle gefunden und angestoßen, positionieren sie sich so, dass die Quelle zwischen ihnen und einer Lichtquelle steht. Anschließend wird die Quelle geschoben und eine erneute Suche nach der Quelle gestartet, da diese unterwegs abhanden gekommen sein kann (die Roboter verfügen über keinerlei Sensorik außer der Gasmenge und der Tatsache, dass sie irgendwo gegen gestoßen sind.

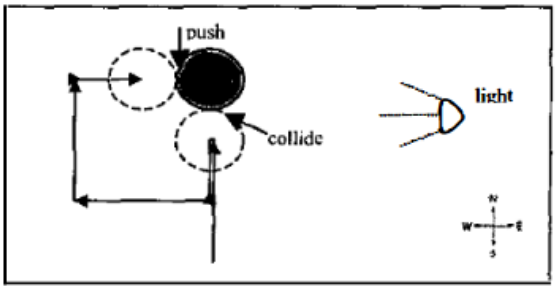

Abbildung 4.3: Struktur-Erstellung mit Bausteinen

Die Roboter waren in den Experimenten, trotz dem simplen Aufbaus durchweg erfolgreich damit, die Quelle aus ihrem 'Nest' zu befördern. [14]

5 Konzeption

In diesem Kapitel werde ich die Konzeption hinter den Schwarmrobotern genauer beschreiben. Dazu gehört die Struktur des Systems generell, aber auch wichtige Algorithmen die die Roboter selbst, aber auch deren Interaktion mit anderen beschreiben.

5.1 Generelles zur Konzeption

Die Konzeption dieser Thesis beruht darauf, dass ROS (Robot Operating System) genutzt wird. ROS ist ein Framework, dass die Erstellung von Robotern erleichtern soll und arbeitet mit Nodes, ähnlich Microservices. Diese Nodes können als Publisher fungieren, die Nachrichten an ein Topic senden und welche man dann abonnieren kann. Sie können aber auch als Services dienen, denen man Daten sendet und von denen man anschließend eine Antwort erwarten kann.

Die Konzeption ist in 3 Phasen unterteilt, die das Konzept zum letztlichen Ziel führen werden. Der Sinn hinter diesen 3 Phasen, ist eine Iteration. Die anderen Kapitel (Implementierung und Evaluation) werden parallel geführt werden und nach jeder Phase der Konzeption wird das System zuerst prototypisch gebaut und anschließend evaluiert. Dadurch kann die Konzeption in seinen weiteren Phasen auf die Ergebnisse der vorangegangenen eingehen und auf den neuen Erkenntnissen aufbauen. Außerdem wird dadurch sichergestellt, dass das Verhalten der Roboter nicht zu sehr auf die Aufgabe des Transports zugeschnitten wird. Das ist deswegen wichtig, weil das zugrundeliegende System ein natürliches Schwarmverhalten, ähnlich dem von Tieren, sein soll und keine Gruppe von autonomen, aber strategisch denkenden Robotern.

5.2 Nachbau des Schwarms nach Craig Reynolds

Da der Transport des Schwarms auf den 3 (bzw. 4) Grundregeln des Schwarmverhaltens besteht, welche von Craig Reynolds erstmals modelliert wurden[3], galt es diese erst nachzustellen und zu analysieren. Raynolds selbst stellte nur die ersten drei Regeln auf. Eine vierte wurde hinzugefügt, da diese im realen Einsatz Sinn macht und auch im natürlichen Verhalten von Tieren zu sehen ist. Der Schwarm wurde zunächst nachgebaut, ohne auf die spätere Verwendung für Transporte zu achten, da diese (möglichst) ohne Kompromisse auf dem Schwarm-Prinzip aufbauen sollten.

Ziel

Das Ziel der ersten Phase ist es, grundsätzliches Schwarmverhalten nachzubauen und dieses zu messen. Es werden individuelle Roboter erstellt, die sich nur dadurch bewegen, indem sie sich an anderen Einheiten orientieren und dabei noch ihren freien Willen in die Entscheidung der Bewegungsrichtung einfließen lassen. Anschließend wird gemessen, auf welche Weise die verschiedenen Parameter das Verhalten der Roboter beeinflussen.

Messages

Die Roboter brauchen einige Informationen über die anderen Roboter, darunter vor allem die Positionen und Ausrichtungen derer, die innerhalb einer bestimmten Reichweite liegen. Diese Informationen könnten über Sensoren kommen. Da diese Sensoren aber ungenau und kostspielig sind, wird davon ausgegangen, dass sie in der Praxis nicht zur Verfügung stehen. Die Roboter müssen ihre eigene Position stets kennen und werden daher diese Informationen direkt mit den anderen Robotern teilen.

Da der ROS-Master als Nameserver innerhalb des Systems dient, ist zu diesem eine ständige Verbindung notwendig. Dieser kann aber auf einem normalen Roboter gestartet werden. Es ist also keine weitere Einheit notwendig, wodurch das Gesamt-System homogener bleibt, wenn auch immer eine direkte Verbindung zum Master bestehen bleiben muss. Es gibt Bemühungen ROS für mehrere, parallel laufende, Master-Nodes zu öffnen, bisher sind die Packages aber noch in der Entwicklung und es muss auf einem zentralen Master aufgebaut werden[1]. Jedem Roboter wird zudem eine eindeutige ID zugewiesen werden, die es erlaubt ihn von den anderen unterscheiden zu können.

Für eine einfache Schwarm-Simulation ist nur eine Art von Nachricht notwendig. Diese enthält die Position, Ausrichtung und die ID der Roboter und wird nach jedem ausgeführten Bewegungs-Befehl gesendet. Um das System der Roboter einfacher zu halten, abonnieren die Roboter auch die Bewegungs-Daten von sich selbst. Dadurch ist es nicht notwendig diese gesondert zu behalten, sondern die Daten werden in das eigene System eingepflegt, wie die von allen anderen Einheiten auch. Möchte ein Roboter dann auf die eigenen Daten zugreifen, kennt er seine eigene ID und weiß daher welche Daten seine eigenen sind.

Neue Nachricht

Der bisher einzig genutzte Nachrichten-Typ beinhaltet somit die Positionsdaten der einzelnen Roboter, zusammen mit der ID, wem die Daten gehören. Für die Deklaration der Nachrichten-Typen werden die Datentypen genutzt die ROS beinhaltet[17]. Dies erleichtert die Implementierung und die Namen der Datentypen sind außerdem selbsterklärend.

Nachrichten-Typ: Robot_Position.msg

```
uint8 robot_id  // Zur Identifizierung des Absenders
float32 pos_x   // Position des Roboters entlang der X-Achse
float32 pos_y   // Position des Roboters entlang der Y-Achse
float32 angle   // Ausrichtung des Roboters im Winkel von [0, 360] Grad
```

Diese Art von Nachricht wird auch den Großteil der gesendeten Nachrichten ausmachen. Das Verhalten der Roboter wird überwiegend von den Positionen anderer gesteuert, deshalb sollte dieser Nachrichten-Typ so oft wie möglich gesendet werden. Da dieser Nachricht-Typ aber auch von allen Robotern empfangen wird, beträgt die Anzahl der Nachrichten die bei der Aktualisierung des gesamten Schwarms durch das Netzwerk gehen $O(n^2)$. Damit das Netzwerk nicht überfordert wird, muss deshalb ein Mittelwert gefunden werden. Zu viele Nachrichten verstopfen das Netzwerk, bei zu wenigen ist die Aktualisierung zu langsam und Roboter stoßen eventuell zusammen, weil sie die Position der anderen Roboter woanders vermuten.

[1] http://wiki.ros.org/multimaster_fkie

Einhalten der 4 Grundregeln

Die Roboter werden als autonome Einheiten konzipiert, die sich ausschließlich anhand von 3 fixen Parametern bewegen und dabei versuchen die 4 Grundregeln des Schwarmverhaltens einzuhalten. Nachfolgend werden diese vorgestellt.

Ausrichtung: Passe deine Bewegungsrichtung deinen Nachbarn an

Da ein Roboter die Position und Ausrichtung jedes anderen Roboters kennt, ist es ohne weiteres möglich die durchschnittliche Ausrichtung der Nachbarschaft zu ermitteln. Dafür wird die Liste der Roboter genommen und die Roboter, die Nahe genug ist um relevant zu sein, in eine neue Liste kopiert. Von diesen Robotern wird dann der Durchschnitt des Winkels berechnet. Der einfache Durchschnitt, über die Summe mehrerer Winkel, würde jedoch zu einem mathematischen Schnitt führen, statt zu einem, der in der Praxis relevant ist. Daher müssen die Winkel für die Berechnung zunächst in Vektoren umgerechnet werden. Diese Vektoren werden dann addiert und der resultierende Gesamtvektor wird letztlich wieder in einen Winkel zurück gerechnet. Bei zwei gegenüberliegenden Winkeln müsste ein Roboter, der sich danach ausrichten muss, theoretisch still stehen, da die Winkel sich gegenseitig aufheben. In diesem Fall entscheidet schlicht die Ungenauigkeit der Gleitkomma-Zahlen darüber, ob er den Winkel in die eine oder in die andere Richtung nimmt.

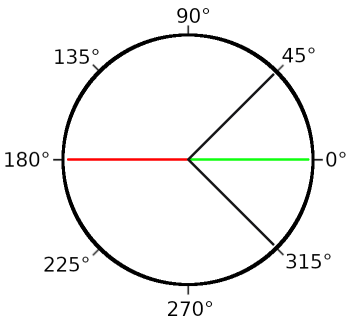

Abbildung 5.1: Winkel-Übersicht

Beispiel: Fehlerhafte Winkel-Berechnung Abbildung 5.1 zeigt ein Beispiel für einen Fehler, beim Berechnen des Durchschnitts zweier Winkel. Zwei Roboter mit den Winkeln 45° und 315° würden zu einem durchschnittlichen Winkel von 180° führen. Der richtige Winkel in Anbetracht der Ausrichtung wäre jedoch 0°.

Zusammenhang: Versuche deinen Nachbarn nahe zu sein

Dieser Abschnitt wird realisiert, indem zuerst die Liste der Roboter, die nahe genug sind um zur Nachbarschaft zu gehören, wieder hergenommen wird. Anschließend wird ein simpler Durschnitt über alle Positionen (x/y - Koordinaten), der vorhandenen Roboter berechnet. Die resultierenden Koordinaten werden von den eigenen Koordinaten abgezogen, um die relative Position des Schwarm-Mittelpunkts zu bekommen. Indem man nun die relativen Koordinaten als Vektor behandelt und dadurch zu einem Winkel umrechnet, erhält man letztlich den relativen Winkel, den der Roboter anfahren müsste um zum Mittelpunkt seines Schwarms zu gelangen. Dieser Winkel wird mit normaler Geschwindigkeit angefahren.

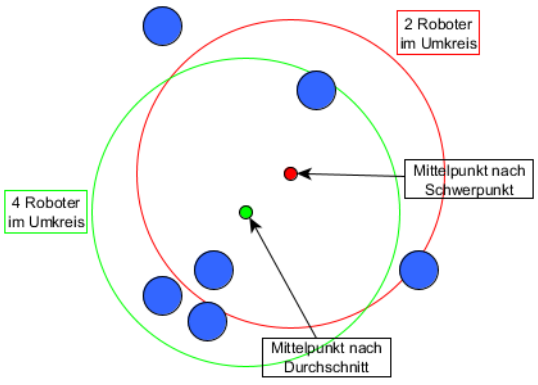

Abbildung 5.2: Berechnung des Mittelpunkts

Design-Entscheidung: Mittelpunkt des Schwarms Um den Mittelpunkt des Schwarms zu berechnen, können verschiedene Algorithmen verwendet werden. Die einfachste Wahl, wäre der Mittelpunkt über die Koordinaten, den die (bis zu 4) äußersten Roboter im Koordinaten-System aufspannen. Dieser Mittelpunkt wäre sehr schnell zu berechnen, bezieht aber nicht mit ein, wie die Roboter innerhalb dieser Fläche verteilt sind. Eine andere Wahl wäre der geometrische Schwerpunkt der konvexen Hülle, den die Roboter in ihrer Formation aufspannen. Dieser Mittelpunkt wäre etwas genauer in Hinblick auf die konkrete Form. Aber auch hier würde nicht die tatsächliche Verteilung beachtet werden, wenn auch genauer als in der vorigen Methode.

Eine sehr einfache Methode um die einzelnen Roboter innerhalb der Formation besser gewichten zu können, ist ein normaler Durchschnitt über die Summe der Koordinaten der einzelnen Roboter. Diese Methode verbraucht wenig Ressourcen, fast keine Kenntniss bei der Implementierung und bietet theoretisch auch die Möglichkeit einzelne Roboter stärker zu gewichten als andere, falls dies später sinnvoll wäre. Da diese Methode sowohl die schnellste, die genauste, als auch die flexibelste ist, wurde sich für diese entschieden.

Abbildung 5.2 zeigt eine Darstellung, wie der Mittepunkt in einem Schwarm mit mehreren Roboter angelegt ist. Die Roboter zieht es dadurch eher zu Gruppen anderer Roboter als zu Einzelgängern. Würde es den Roboter zur Mitte der konvexen Hülle ziehen, wäre die Gefahr groß, dass er alle Roboter verlieren würde, da sie zu allen Seiten gleich weit entfernt sind. Durch die einfache Methode des Mittelwerts bleibt der Roboter stets näher an Gruppen und lässt notfalls einzelne Roboter ziehen um die Gruppe zu behalten.

Abschottung: Vermeide Kollisionen mit deinen Nachbarn

Bevor ein Roboter seinen Bewegungs-Befehl ausführt, wird mit den gewünschten Parametern eine interne Simulation durchgeführt, um die Roboter vor Kollisionen zu bewahren. Außerdem müssen sie eine Größe und eine Fläche (in einer 2dim Simulation) erhalten. Während der Simulation wird getestet, ob der gewünschte Winkel im Zusammenhang mit der Geschwindigkeit zu einer Kollision mit anderen Robotern führen würde. Eine Bewegung wird als gültig befunden, wenn der Roboter an seinem Zielpunkt mindestens seine Größe als Abstand zu allen anderen Robotern hat. Praktisch wurde die Größe um 10% erweitert, um eine gewisse Fehlertoleranz zu erlauben. Auch die Frequenz, der in Abschnitt 5.2 erwähnten Positions-Nachrichten, spielt hier wieder eine Rolle. 2 Roboter können dann aufeinander zufahren, wenn der Abstand beider Roboter nicht innerhalb der nächsten Aktualisierung der Positionsdaten überwunden werden kann.

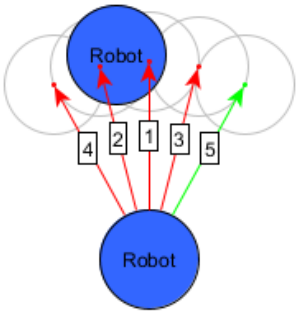

Abbildung 5.3: Ausweichen gegenüber Hindernissen

Ob eine Kollision am Ziel (oder unterwegs) stattfinden wird, wird mittels interner Simulationen überprüft, die die letzten Positionsdaten der Roboter nutzt. Findet eine Kollision statt, prüft der Roboter Ausweichmöglichkeiten. Der Algorithmus wird dabei angewendet, egal ob das Hinderniss ein anderer Roboter ist oder eine Gefahrenzone. Ist der Weg geradeaus nicht möglich, wird abwechselnd versucht nach links und rechts auszuweichen, wie in Abbildung 5.3 gezeigt wird. Dabei wird der Winkel, der zum Ausweichen genutzt wird, stufenweise erhöht, bis man letztlich 180° zu beiden Seiten erreicht hat, was einem Schritt rückwärts entspricht. Führt auch dieser letzte Winkel nicht zu einem erfolgreichen Ergebnis, wird der Algorithmus erneut mit einer leicht

geringeren Geschwindigkeit (was ebenfalls einer leicht geringeren Bewegungsdistanz entspricht) gestartet. Erst wenn eine Simulation eine erfolgreiche Bewegung generieren konnte, wird diese auch ausgeführt.

Flucht: Fliehe vor Dingen, die eine potentielle Gefahr darstellen

Gefahren wie Hindernisse, Löcher im Boden oder Zonen mit sich bewegenden Maschinen können die Roboter beschädigen, weshalb es gilt diesen auszuweichen. Bei jeder Bewegung muss geprüft werden, ob man mit der angestrebten Bewegung eine Gefahrenzone betreten würde. Wenn ja, wird versucht mit dem Algorithmus des vorigen Abschnitts auszuweichen.

Um Gefahrenzonen abbilden zu können, werden Objekte erzeugt die sich aus verschiedenen Geometrien, mitsamt Positionen, zusammensetzen. Betritt ein Roboter eine der im Objekt inneliegenden Geometrien, ist das Objekt als ganzes betreten worden und der Roboter befindet sich in einer Gefahrenzone.

Design-Entscheidung: Abbildung von Objekten Um ein Objekt zu definieren ließen sich auch andere Methoden nutzen. Punkte die die Zone umspannen und eine konvexe Hülle bilden oder der Reihenfolge nach mit Geraden verbunden werden, wären ebenfalls eine Option. Allerdings wären diese viel komplexer zu berechnen und es würde mehr Rechenleistung verbrauchen. Auch Funktionen, aus denen sich die einzelnen Punkte heraus berechnen lassen, wären denkbar um Objekte darzustellen. Um diese aber zu verteilen, müsste sie aber in einer Syntax hinterlegt werden die sich serialisieren lässt und anschließend interpretiert werden. Außerden wären die Funktionen kompliziert zu erstellen, wohingegen ein paar Geometrien schnell abgemessen sind.

Die Berechnung, ob sich ein geometrischen Objekt innerhalb eines anderen geometrischen Objektes befindet, ist allerdings ein häufiges und gut gelöstes Problem von Videospielen[11], weswegen diese Lösung als die beste angesehen wurde.

Parameter der Bewegung

Konfiguriert wird das Schwarmverhalten der Roboter allgemein durch 4 Parameter, die in den nachfolgenden Abschnitten näher erläutert werden.

Geschwindigkeit Die Geschwindigkeit gibt an, wie schnell ein Roboter sich bewegt. Ein kleinerer Wert sorgt für langsamere, aber auch 'vorsichtigere' Roboter. Optimal wäre ein kleiner Wert in Geschwindigkeit, aber eine hohe Frequenz an Bewegungs-Befehlen.

Lokale Reichweite Da ein Roboter nur seine unmittelbare Umgebung imitiert, braucht es dafür einen Grenzwert. In dieser Thesis wurde sich für eine Begrenzung in der Reichweite (im Gegensatz zu einer Begrenzung in der Anzahl der ausgewählten Roboter) entschieden. Für die Orientierung wurde

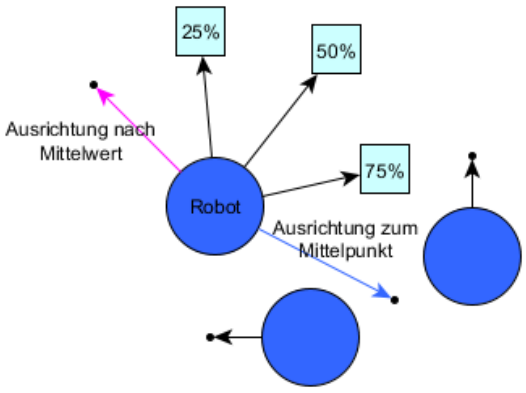

Abbildung 5.4: Berechnung des Drangs zur Gruppierung

mehr Wert auf die örtliche Reichweite gelegt, da die unmittelbare Umgebung wichtiger ist, als eine feste Anzahl an Robotern an denen sich orientiert wird. Einheiten die weiter entfernt sind als dieser Wert (in der entsprechenden Maßeinheit), werden für Berechnungen ignoriert. Ein Wert kleinerer als die eigene physikalische Größe, führt zwangsläufig dazu, dass die Roboter nur sich selbst beachten.

Freier Wille Einheiten im Schwarm orientieren sich nicht nur an ihren Nachbarn, sondern besitzen in gewisser Weise auch einen eigenen Willen. Nachdem die Berechnung der Orientierung am Schwarm abgeschlossen ist, wird der eigene Wille einberechnet. Bei einem freien Willen von x°, wird dieser zufällig zwischen [-x/2, x/2] ausgesucht und dem bisherigen Winkel aufgerechnet wird. Der freie Wille ist der Parameter der mich am meisten das generelle Verhalten des Schwarms beeinflusst. Er entscheidet darüber, wie viel Zufall in der Bewegung des Schwarms enthalten ist und wie viel sie von ihrem 'erzwungenen' Verhalten abweichen dürfen.

Nähe zum Schwarm Dieser Wert ist verbunden mit der Regel 'Zusammenhang: Versuche deinen Nachbarn Nahe zu sein'. Ein Roboter hat den Drang seinem Schwarm nahe zu bleiben und versucht sich, bis zu einem gewissen Grad, in dessen Richtung zu bewegen. Da der Drang zum Schwarm nahe zu bleiben als Teil des freien Willens betrachtet wird, wird dieser Wert prozentual angegeben und nach dem freien Willen auf den endgültigen Winkel aufgerechnet. Wie genau der Drang zur Gruppierung in Prozente eingeteilt ist, wird in Abbildung 5.4 skizziert dargestellt. Der pinke Pfeil stellt die Richtung dar, in der sich der Roboter bei keinem Gruppendrang bewegen würde. Der blaue Pfeil ist die Richtung zum Mittelpunkt des Schwarms, diesen Weg würde er bei 100% Gruppendrang fahren. Die anderen Pfeile wären die Richtungen bei den jeweiligen Prozentwerten.

Beispiel: Berechnung einer Bewegung

In Abbildung 5.5 ist die Berechnung einer Bewegung skizziert. Die Pfeile geben dabei nur Richtungen an und sind nicht als Vektoren zu verstehen, die eine Aussage über die bewegte Entfernung treffen.

Der pinke Pfeil ist das Resultat der Berechnung des Mittelwerts der Winkel, den die anderen Roboter beim letzten Update ihrer Position haben.

Der blaue Pfeil ist der Mittelpunkt des Schwarms (in diesem Fall sind beide Roboter ein Teil der Nachbarschaft), auf den der Roboter sich aufgrund seiner Gruppenzugehörigkeit zubewegen möchte.

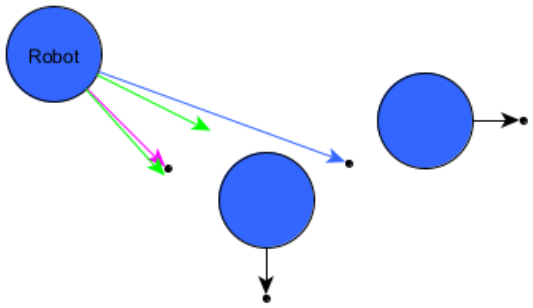

Abbildung 5.5: Berechnung einer Bewegung

Die beiden grünen Pfeile sind letztlich die obere und untere Grenze des Spielraums, den der Roboter durch seinen eigenen freien Willen bekommt. Dieser wurde durch die Ausrichtung zum Schwarm-Mittelpunkt zusätzlich leicht verschoben. Aus diesen beiden Grenzen wird letztlich ein Winkel zufällig bestimmt und anschließend auf Gültigkeit (fehlende Kollisionen) geprüft, um dann schlussendlich ausgeführt zu werden.

5.3 Anführer

Um später Transportaufträge erledigen zu können, muss der Schwarm in irgendeiner Form gesteuert werden können. Bei einer direkten Steuerung aller Einheiten des Schwarms, wäre das Schwarmverhalten allerdings außer Kraft gesetzt und man hätte schlicht eine Menge gelenkter Roboter. Deswegen wurden Mittel gesucht den Schwarm indirekt zu lenken und fand diese Methode in einem Paper in dem beschrieben wurde, wie ein Schwarm über 'eingeweihte Anführer' gesteuert werden kann.[12]

Funktionsweise

Der Anführer bietet eine passive Möglichkeit einen Schwarm zu lenken, ohne dass die normalen Einheiten angepasst werden müssen oder in ihrem Verhalten speziell abweichen müssen. Die normalen Einheiten müssen dadurch nicht wissen, dass sie ein bestimmtes Ziel haben, es muss ihnen nicht einmal bewusst sein, dass sie passiv gesteuert werden. Dadurch, dass der Anführer weiß in welche Richtung der Schwarm am besten gelenkt werden sollte, aber auch aufgrund des Gruppendrangs durch die Nachbarn, pendeln sich die anderen Einheiten des Schwarms langsam auf das Ziel ein. Ein Anführer ist somit eine Art Leuchtturm, der permanent eine Richtung angibt, ohne sich von der Umwelt beeinflussen zu lassen und der den anderen Robotern eine Orientierung gibt, ohne dass diese davon etwas mitbekommen.

Als Anführer versucht die entsprechende Einheit außerdem den Kontakt zu seinem Schwarm nicht zu verlieren. Damit ist nicht nur der Drang gemeint die Gruppenmitte aufzusuchen, der vom Anführer sowieso ignoriert wird. Es ist eine bewusstere Art auf den eigenen Schwarm acht zu geben und sich so zu bewegen, dass er weder die Gruppe verliert, noch diese ihre Ausrichtung zum Ziel zu sehr verändert. Dies ist wichtig, damit immer möglichst viele Robter des Schwarms von der Ausrichtung des Anführers beeinflusst werden.

Ziel

Ziel dieser Phase ist es, den Schwarm von einem Ort zu einem anderen zu lenken, ohne ihn wissen zu lassen, dass er gelenkt wird und möglichst ohne Einheiten auf dem Weg dorthin zu verlieren. Dies soll mit Hilfe von (möglichst wenigen) Anführer geschehen, die über das Ziel Bescheid wissen. Auf diese Weise muss in das ursprüngliche Verhalten der Roboter nicht eingegriffen werden und die Verhaltensweise der Roboter, die nicht zum Anführer erwählt wurden, bleibt exakt so bestehen wie es in der vorherigen Phase konzipiert wurde.

Einbindung des Anführers in ROS

Der Anführer wurde umgesetzt, indem einer normalen Einheit ein Ziel gegeben wird und diese damit beginnt seinen Schwarm an einen bestimmten Ort zu lenken. Dazu werden die Roboter ein neues Topic abonnieren. Wenn ein Roboter diese Nachricht erhält, prüft er ob die darin enthaltene ID seiner eigenen entspricht und wenn ja, wird er den Auftrag annehmen, in den 'Anführer-Modus' wechseln und versuchen seinen Schwarm zu dem definierten Ziel zu führen. Roboter auf die ID nicht zutrifft werden die Nachricht komplett ignorieren und mit ihrem bisherigen Verhalten fortfahren.

Neue Nachricht

Um den Transport einzuleiten wird es einen neuen Nachrichten-Typ geben. Die Nachricht kann an den Schwarm rausgesendet werden und durch die eindeutige ID weiß der entsprechende Roboter, dass er als Anführer ausgewählt wurde.

Nachrichten-Typ: New_Mission.msg

```
uint8 leader_id // Die ID des ausgewaehlten Anfuehrers
float32 pos_x   // Position des Ziels entlang der X-Achse
float32 pos_y   // Position des Ziels entlang der Y-Achse
```

Generelles Verhalten

Er steuert direkt auf das Ziel zu und wartet gegebenenfalls, wenn er sich zu weit vom eigenen Schwarm entfernt. Ab einer Entfernung von 75% , der lokalen Reichweite zum Mittelpunkt seiner Herde, bleibt er stehen, bis er wieder bei 50% Entfernung angekommen ist. Anschließend setzt er seinen Weg mormal fort. Ausweichmanöver würden dafür sorgen, dass der Anführer nicht mehr auf das Ziel zeigt und damit auch die anderen Roboter dazu bringen der neuen Ausrichtung zu folgen. Aus diesem Grund wird anderen Robotern nicht ausgewichen, wie es für Schwarmroboter üblich ist. Stattdessen wird der Anführer seine Geschwindigkeit verlangsamen, wenn er mit der normalen Geschwindigkeit eine Kollision verursachen würde. Kann er sich, trotz geringerer Geschwindigkeit nicht bewegen, bleibt er stehen und wartet eine kleine Zeitspanne ab.

Einfangen eines verlorenen Schwarms

Sollte der Schwarm den Grenzbereich der lokalen Reichweite ganz verlassen, wäre es außerdem möglich, den Anführer seinen Schwarm wieder einfangen zu lassen. Dies ist in Abbildung 5.7 dargestellt. Im ersten Abschnitt ist der Schwarm verloren gegangen, obwohl der Anführer gewartet hat. Anschließend ist er mit erhöhter Geschwindigkeit hinter den Schwarm gefahren, sodass der Schwarm sich nun genau zwischen Ziel und Anführer befindet. Nun nimmt er wieder sein normales Verhalten an und versucht den Schwarm in die Richtung des Ziel zu lenken. Beim umdrehen, um den Schwarm wieder einzufangen, dreht sich der Anführer allerdings in die entgegengesetzte Richtung. Die anderen Roboter wissen nichts von einem Manöver und orientieren sich normal am Anführer. Das kann dazu führen, dass der Schwarm zunächst noch weiter abgelenkt wird. Aus diesem Grund sollte das Einfangen so schnell wie möglich beendet werden. Die Geschwindigkeit des Anführers sollte also möglichst hoch sein.

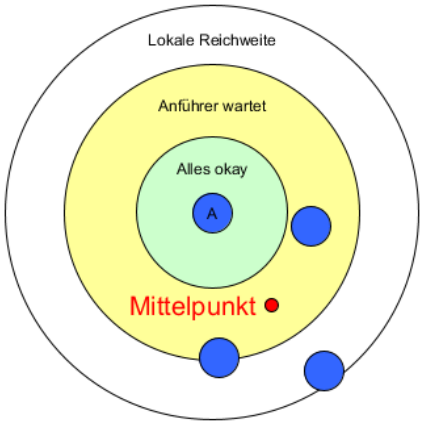

Abbildung 5.6: Entfernungen des Anführers

5.4 Transport von Waren mit Hilfe eines Schwarms

Das letztliche Ziel dieser Arbeit ist es zu prüfen, ob es möglich, und sinnvoll, ist Gegenstände mit Hilfe eines autonomen Schwarms zu transportieren. Dazu musste der Schwarm nun dazu gebracht werden Waren zu bewegen, ohne die einzelnen Roboter zu sehr zu beeinflussen. Da generell die Roboter nicht zentral gesteuert werden sollen und auch die Logik möglichst simple bleiben soll, sollte das Programm der einzelnen Einheiten möglichst wenig verändert werden und der 'natürliche' Trieb der Einheiten genutzt werden.

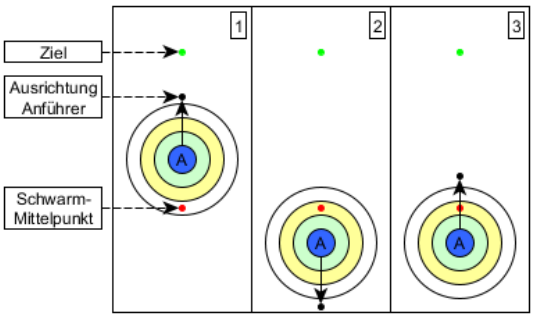

Abbildung 5.7: Anführer fängt seinen Schwarm wieder ein

Erteilen von Aufträgen

Eine der ersten Dinge im Ablauf eines Transports ist die Erteilung des Auftrags. Um die Roboter einem bestimmten Auftrag zuzuteilen und den jeweiligen Anführern mitzuteilen, dass sie diese Rollen übernehmen müssen, braucht es neue Informationen innerhalb des ROS-Systems. Da es den Nachrichten-Typ `New_Mission` bereits gibt, muss dieser nur erweitert werden, sodass er nun die folgenden Felder hat:

Veränderte Nachricht

Nachrichten-Typ: New_Mission.msg

```
uint8 robot_index_from    // Beginn der ID-Range
uint8 robot_index_to    // Ende der ID-Range

uint8 leader_number     // Anzahl der Leader die gebraucht werden
uint8 mission_id      // Die ID der Mission

float32 object_position_x // Die Start-Position des Objektes, X-Achse
float32 object_position_y // Die Start-Position des Objektes, Y-Achse

float32 object_size_x   // Die Laenge des Objektes, in X-Richtung
float32 object_size_y   // Die Laenge des Objektes, in Y-Richtung

float32 target_x      // Die Ziel-Position des Objektes, X-Achse
float32 target_y      // Die Ziel-Position des Objektes, Y-Achse
```

Der Nachrichten-Typ definiert nun eine Spanne von Robotern die den Auftrag ausführen sollen. Diese werden mit ihren IDs angesprochen. Das Intervall der IDs ist, gemäß Programmierstandards, als halboffenes Intervall definiert: [robot_index_from, robot_index_to[.

leader_number gibt die Anzahl der Leader an die verwendet werden sollen und mission_id gibt dem derzeitigen Auftrag eine fixe ID um diesen und zugehörige Dinge genau identifizieren und verbinden zu können. Da die Anführer den Schwarm

kontrollieren müssen, werden spezielle Nachrichten notwendig sein. Um zu verhinden, dass diese von allen Anführer gesendet werden, braucht es einen 'Haupt-Anführer'. Da der einfachste Weg ist diesen zu bestimmen, den mit der kleinsten ID zu nehmen, wird er im Folgenden entsprechend der erste Anführer genannt werden.

Mit object_position_x/-_y ist die Startposition des zu transportierenden Objektes angegeben. Zusammen mit object_size_x/-_y, welche die Ausmaße des Objektes angeben. Die Position des Objekts ist als Mitte des Objekts definiert. target_x/-_y gibt die letztliche Position an die das Objekt einnehmen soll.

Der Auftrag wird von außen an das Topic 'flock/mission/new' gesendet. Der Ersteller des Auftrags muss keine ROS-Node sein, auch wenn dies verschiedene Vorteile hätte. Dadurch das ROS-Topics auch über das Terminal engesprochen werden können, ist das Senden der Nachrichten auch über jedes andere Programm möglich. Das Topic für die neuen Missionen wird von jedem aktiven Roboter abonniert. Entsprechend nimmt jeder Roboter Notiz von diesem Auftrag, auch wenn er nicht direkt mit seiner ID angesprochen wird.

Von Gefahrenzonen zu Sicherheitszonen

Um die betreffenden Roboter in die Zone zu senden, auf der ihnen später das Transportobjekt aufgesetzt wird, wird ein Mechanismus verwendet der bereits vorher Einzug in das ROS-System hielt: Gefahrenzonen. Diese werden leicht angepasst, um sie invertieren zu können und somit aus einer Gefahrenzone eine Sicherheitszone zu machen. Ist ein Bereich als Sicherheitszone definiert, ist jeder andere Bereich automatisch eine Gefahrenzone. Dieser Mechanismus birgt nur eine kleine Änderung im System, schafft es aber Roboter in einen bestimmten Bereich zu locken ohne sie aktiv steuern zu müssen, indem einfach ihr 'natürlicher' Trieb verwendet wird, von Gefahren zu flüchten.

Wird ein Auftrag erteilt, so nehmen die Roboter die dem Auftrag zugeteilt sind, eine neue Sicherheitszone auf, die den Ausmaßen und der Position des Transportobjekts am Aufnahmeort entspricht. Da jeder andere Ort außerhalb der Sicherheitszone analog eine Gefahrenzone wird, werden die Roboter nun versuchen sich in diese Sicherheitszone zu bewegen. Roboter die nicht dem Transport zugeteilt wurden, werden diese neue Zone als Gefahrenzone auffassen um die Transport-Roboter nicht bei ihrer Arbeit zu stören und versuchen sich diesem Gebiet fernzuhalten.

Ein Objekt kann grundsätzlich aus verschiedenen Geometrien zusammengesetzt werden. Dadurch ist es möglich nicht nur Objekte zu transportieren die eine einfache Form wie Rechtecke oder Kreise haben, sondern verschiedene Rechtecke können dann zu einem 'L' oder 'U' zusammengesetzt werden.

Füllen eines Raums

Wird eine Sicherheitszone definiert, versuchen die Roboter, die sich in der Gefahrenzone befinden, auf möglichst schnellstem Wege die Sicherheitszone zu betreten. Dazu wird der nächste, verfügbare Ort der Sicherheitszone ins Ziel genommen und (ohne mit anderen Robotern zu kollidieren) der direkte Weg darauf zu genommen. Da die Roboter sich in Gefahr denken, wird während dieser Zeit der freie Wille außer Kraft gesetzt, da das das eigene überleben den höchsten Willen eines Roboters darstellt. Die Roboter werden sich nun darin aufhalten und durch ihre zufälligen Fahrtrouten automatisch Platz für nachkommende Roboter schaffen.

Nachrichten

Damit das Transportobjekt nun sicher auf den Robotern aufgesetzt werden kann, müssen diese nun still stehen. Ein Stillstand von Einheiten eines Schwarms ist laut den Regeln des Schwarmverhaltens nicht möglich, da es keine Regel gibt, die diesen Zustand hervorrufen könnte. Daher wird ein Trick angewendet, der weniger in das Verhalten der Roboter eingreift, als viel mehr direkt in die unteren Layer des Roboter-Systems. Bemerkt der erste Anführer, dass alle Roboter innerhalb des Transportobjektes sind, befiehlt er ihnen mit dem folgenden neuen Nachrichten-Typ still zu stehen.

Nachrichten-Typ: Robot_Freeze.msg

```
uint8 robot_index_from    // Beginn der ID-Range
uint8 robot_index_to    // Ende der ID-Range
```

Jeder Roboter der sich in dieser Range befindet, (egal ob er dem Auftrag zugeteilt ist oder nicht) wird daraufhin still stehen. Von hier an braucht es ein Event, dass den Robotern zeigt dass sie mit dem Transportobjekt Richtung Ziel losfahren können. Möglich wäre eine Zeitsteuerung für streng automatisierte Prozesse. Aber auch interne Signale über ROS sind möglich. Durch die ID die jeder Auftrag hat, wäre eine einfache Nachricht '[TransportID#][Losfahren]' bereits zielführend. Ein zweiter, neuer Nachrichten-Typ wird daraufhin die Roboter wieder aus ihrem Schlaf erwecken und sie in ihr normales Verhalten zurück werfen.

Nachrichten-Typ: Robot_Continue.msg

```
uint8 robot_index_from    // Beginn der ID-Range
uint8 robot_index_to    // Ende der ID-Range
```

Der Transport

Um den Transport selbst zu realisieren, bedient man sich nun der Hilfe der Anführer. Diese richten sich dynamisch nach dem Winkel aus, den das Transportobjekt zum Ziel hat, wie Abbildung 5.8 zeigt. Danach fangen sie an sich langsam in die Richtung zu bewegen in die sie sich ausgerichtet haben. Die anderen Roboter werden sich aufgrund des Gruppenverhaltens ebenfalls mehr oder weniger, gehemmt durch den eigenen freien Willen und ihrem Drang sich zur Mitte ihres Schwarms zuzubewegen, nach ihren Anführern ausrichten und das Objekt so langsam in Richtung des Ziel zu bewegen.

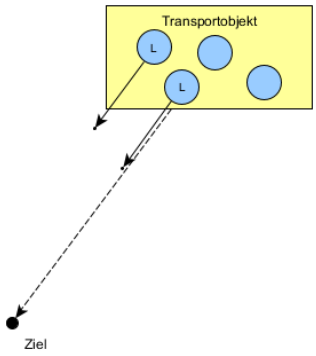

Abbildung 5.8: Der Transport des Objekts

Einhalten der Richtung hat Priorität

Kann ein Anführer keine normale Bewegung ausführen, weil er sonst mit einem anderen Roboter kollidieren würde, darf er nicht versuchen auszuweichen, da dies sonst die Ausrichtung der passiven Roboter negativ beeinflussen könnte. Stattdessen drosselt er zunächst sein Bewegungstempo oder bleibt, falls notwendig, ganz stehen. Die richtige Richtung beizuhalten ist wichtig, da sich passive Roboter nicht unmittelbar nach ihren Anführern ausrichten. Gerade wenn es zahlenmäßig wenige Anführer im Vergleich

zu passiven Robotern sind, kann es einige Zeit dauern, bis die Einheiten so weit beeinflusst wurden, dass sie in die gewünschte Richtung zeigen. Dreht sich ein Anführer in die verkehrte Richtung, vielleicht sogar in die entgegengesetzte, kann dies zu einer Kettenreaktion führen die alle Roboter betrifft und zusätzlich das Transportobjekt stark vom Weg abbringen. Dieser Effekt konnte bereits in früheren Simulationen erkannt werden (siehe: Abschnitt 7.3) und wird deshalb in dieser Phase der Konzeption vermieden.

Mitziehen des Transportobjektes

Die Roboter haben keine Sensorik, um den Zustand des physikalischen Transportobjekt zu erkennen. In der Software ist es daher als Sicherheitszone markiert. Bisher konnten Objekte in der Software allerdings nicht verschoben werden. Daher bekommen Objekte nun eine ID zugewiesen, die es ihnen erlaubt eindeutig erkannt zu werden. Außerdem müssen die Objekte nun verschoben werden können und es braucht jemanden der die Änderung des physikalischen Standorts erkennt und ihn der Software verbreitet.

Grundsätzlich gibt es für dieses Vorgehen verschiedene Vorgehensweisen. Eine davon wäre ein Indoor Positioning System (IPS), das heißt ein Sensor auf dem Transportobjekt, mit einer ROS-Node die diesen Sensor verfolgt und als Broadcast im ROS-System verbreitet. Leider haben bisherige Erfahrungen gezeigt, dass solche IPSs nicht sehr zuverlässig sind. Nicht nur ist die Qualität der Ortung sehr ungenau, sie springt auch zwischen den verschiedenen Messintervallen hin und her. Mehr dazu im Einschub Abschnitt 5.4.

Um die Position des Transportobjektes im ROS-System bekannt zu machen, und aktuell zu halten, wird deshalb darauf gesetzt, dass das Objekt auf den Robotern bleibt und diese ihre Position selbst erkennen können. Zu beginn des Transports wird die relative Position des Objektes zum Mittelpunkt des Schwarms gespeichert. Bewegt sich nun der Mittelpunkt des Schwarms in eine Richtung, wird die Position des Objektes immer relativ von diesem angesehen. Eine Verdeutlichung ist in Abbildung 5.9 zu sehen. Dabei werden allerdings ein paar Eingestädnisse gemacht.

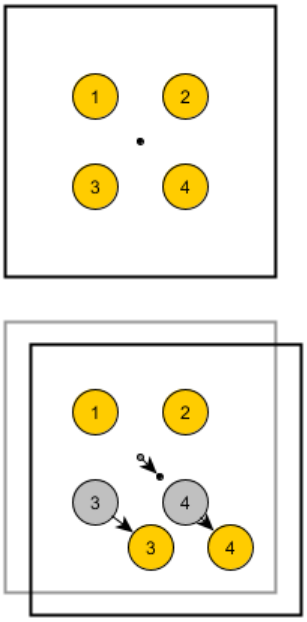

Abbildung 5.9: Tracking des Transportobjektes

Zugeständnisse der Simulation Zum einen wird angenommen, dass die Robter alle die selbe Reibung am Objekt haben. Würde der Transport von nur 2 Roboter erledigt werden und diese bewegten sich in entgegensetzter Richtung von einander weg, müsste das Objekt somit stillstehen. In der Realität wäre die Reibung wahrscheinlich nicht gleich, sodass das Objekt in einer Richtung abdriften würde. Es gilt aber in der Praxis festzustellen, wie viel Ungenauigkeit dadurch wirklich entstehen würde. Eine andere Annahme wäre, dass die Reibung vom Roboter zum Objekt so hoch wäre, dass die Räder der Roboter eher stehen bleiben, statt dass der Roboter sich bewegt. Dadurch würden zwei Roboter, die in entgegengesetzter Richtung fahren, stehen bleiben. Die Roboter müssten natürlich auch erkennen, dass die Räder stehen bleiben. Da dies bei Servo-Motoren allerdings kein Problem darstellt, gilt dieser Punkt als unproblematisch.

Um die Rollenverteilung weiter einzuhalten, sollte einer der Anführer die Position des Objektes innerhalb des Schwarms aktualisieren. Dazu braucht es eine weitere, simple Nachricht.

Nachrichten-Typ: Update_Object.msg

```
uint8 mission_id      // Die ID der Mission

float32 object_position_x // Die Position des Transport-Objektes, X-Achse
float32 object_position_y // Die Position des Transport-Objektes, Y-Achse
```

Da jeder Mission nur ein Objekt zugeordnet ist und dieses, passenderweise, die ID der Mission eingetragen bekommen kann, ist es leicht möglich die Position des Objektes über die ID der Mission zu aktualisieren. Wann immer ein Roboter diesen Nachrichten-Typ empfängt, schaut er in seiner Liste nach einem Objekte mit dieser ID und setzt dessen Position auf die der Nachricht. Bei mehreren Anführern ist es grundsätzlich egal, wer die Aktualisierung sendet, es sollte aber generell bei einem Anführer bleiben, da es nicht notwendig sein sollte die Position in hoher Frequenz zu aktualisieren. Der Einfachheit halber sendet der Anführer mit der geringsten ID die Aktualisierungen. Da jeder die selbe Nachricht bekommt, wissen die anderen Roboter wer alles Anführer ist und ob sie selbst die niedrigste ID besitzen.

Einschub: Unzulänglichkeiten von IPSs Indoor-Positioning-Systems dienen zur Positionsbestimmung innerhalb von geschlossenen Räumen/Hallen und sind damit das kleine Equivalent zu GPS. In vorangegangenen Tests wurde ein System eines namenhaften Hersteller getestet und für zu schlecht empfunden. Die Nachteile waren im groben:

- Ungenauigkeiten von durchschnittlich 81cm
- Der Sensor muss stets Sichtkontakt zu 3 Beacons haben
- Der Bereich der eingegrenzt werden konnte, betrug < 100m²

In Abbildung 5.10 wurden Daten eines IPS in ein Bild eingetragen. Jede Farbe steht für einen anderen Testdurchlauf. Die Kreuze markieren den Punkt, an dem der Sensor, der getrackt werden sollte, tatsächlich stand. Die Punkte in selbiger Farbe sind die Standorte, an denen er laut System gemessen wurde. Abweichungen von über 1m waren eher die Regel als eine Ausnahme. Auch plötzliche Sprünge zwischen 2 Messungen (zeitlicher Abstand 100-300ms) von über 50cm waren keine Seltenheit.

Die Evaluation zeigt klar, dass solche Systeme zum aktuellen Stand noch nicht genug ausgereift, und deshalb nicht für die Ortung eines Transportobjektes geeignet, sind.

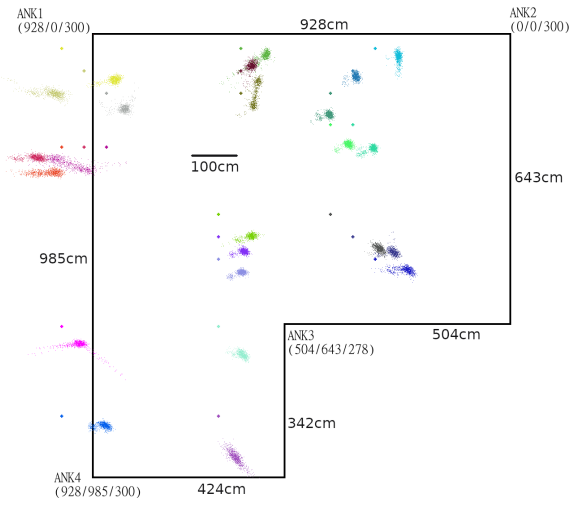

Abbildung 5.10: Evaluation eines IPS

Abschluss des Transports

Ist das Transportobjekt nahe genug am Ziel, ist der Transportauftrag abgeschlossen. Da die Anführer um den Mittelpunkt herum verteilt sind, kann es notwendig sein, dass einige davon selbst über das Ziel hinausschießen müssen, um den Mittelpunkt des Objektes zum Ziel zu bringen. Hier stellt sich wieder die wichtig heraus, dass sie Anführer nicht ihren Winkel zum Ziel einnehmen müssen, sondern den, den der Mittelpunkt zum Ziel hat. Hätten sie den Winkel genommen, den sie selbst zum Ziel haben, würden sie, sobald sie das Ziel erreichen, um dieses herum pendeln und den Schwarm durcheinander bringen. Dadurch, dass sie den Mittelpunkt aus Ausgangswinkel nehmen, fahren sie weiter ihrer Linie nach und das Objekt schafft es letztlich zum Ziel.

Am Ziel angekommen, sendet der erste Anführer sein letztes Update der Position und direkt danach erneut eine Nachricht vom Typ `Robot_Freeze`, woraufhin die Roboter erneut stehen bleiben. An dieser Stelle kann, genau wie beim Aufnehmen des Transports, eine Nachricht von außen das Signal geben, dass das Objekt abgenommen wurde. Es kann, in streng gesteuerten Prozessen, aber auch ein Timer sein. Sobald das Signal kommt, wird der erste Anführer den Stillstand wieder mit einem `Robot_Continue` aufheben und der Schwarm kehrt in seinen normalen Modus zurück. Zusätzlich muss aber noch das Transportobjekt als Hinderniss/Sicherheitszone aus dem Speicher der Roboter gelöscht werden. Dies geschieht mit dem Befehl:

Nachrichten-Typ: Delete_Object.msg

```
uint8 mission_id      // Die ID der Mission
```

Durch diesen Befehl, der auch für die Roboter wichtig ist, die nicht am Transport beteiligt waren, wird das Transportobjekt als Hinderniss gelöscht und der Auftrag ist endgültig vorbei.

5.5 Weiterführende Konzeption

Die Implementierung der Konzeption ist prototypisch und dient vor allem dazu Statistiken erheben zu können. Durch diese Statistiken kann das Konzept in den nachfolgenden Phasen nachgebessert werden und es kann ein Gefühl dafür entwickelt werden, wie die einzelnen Parameter einen Schwarm beeinflussen, um ihn später den gegebenen Umständen besser anpassen zu können.

An dieser Stelle endet die prototypische Implementierung jedoch und die Konzeption wird nicht mehr nachgeprüft. Es werden Ideen vorgestellt und ausgearbeitet, wie man den Schwarm besser und dynamischer gestalten kann, die Kapitel der Implementierung und Evaluation sind dagegen beendet.

5.5.1 Dynamischer Schwarm

Der Schwarm ist bisher statisch. Man stellt eine bestimmte Anzahl an Robotern ein, aus denen der Schwarm bestehen soll und diese müssen alle gestartet werden. Fehlen Roboter oder werden zu viele gestartet, führte das bisher zu Fehlern. Neue Roboter hinzufügen oder bestehende wieder entfernen ist nicht möglich. Dies muss für einen dynamischen Schwarm geändert werden, weswegen zwei neue Nachrichten eingeführt werden die genau das erlauben.

Wird ein neuer Roboter erschaffen (oder aktiviert), stellt sich dieser mit seiner neuen ID im Schwarm vor. Die anderen Roboter im Schwarm werden diesen in ihre Liste und damit in ihre Berechnungen aufnehmen.

Nachrichten-Typ: New_Robot.msg

```
uint8 robot_id      // Die ID des neuen Roboters
uint8 sub_id      // Die Sub-ID des Roboters
```

Nachrichten-Typ: Delete_Robot.msg

```
uint8 robot_id      // Die ID des zu entfernenden Roboters
uint8 sub_id      // Die Sub-ID des Roboters
```

Einzigartigkeit gefährdet

Mit dem dynamischen hinzufügen von Robotern kommt es jedoch zu einem weiteren Problem. Der Einzigartigkeit von IDs. Ist dies bei physischen Robotern noch gut zu kontrollieren, indem jeder Hardware eine bestimmte ID zugewiesen wird und diese in einem zentralen System kontrolliert werden. Kommt es zu Problemen, wenn Roboter virtuelle Kopien erzeugen wollen und diese neue IDs brauchen. Sind beispielsweise die IDs im Bereich von [1,10] vergeben und 2 Roboter wollen neue Kopien erzeugen, kann es zu Kollisionen kommen weil sie einer Kopie die gleiche ID zuweisen.

Natürlich könnten diese Kollisionen aufgelöst werden, beispielsweise indem man pro neuer ID die man erschafft einen Echo-Algorithmus startet und der der die Wahl gewinnt die ID bekommt. Erschafft ein Roboter aber 20 Kopien von sich kann dies sehr viele Nachrichten zur Folge haben die das Kommunikationsnetz belasten. Aus diesem Grund ist es besser Kollisionen in den ID von Anfang an zu vermeiden. Als Lösung wird die ID global im gesamten System um ein weiteres Byte erweitert, der sub_id.

Konzept: Sub-ID Eine sub_id ist ein Zusatz zur normalen ID. Die robot_id wird die Haupt-ID eines Roboters und physische Roboter werden immer unterschiedliche robot_id's haben mit der sub_id '0'. Möchte ein Roboter nun virtuelle Kopien von sich erstellen, erstellt er diese mit seiner robot_id und der nächsten freien sub_id. Auf diese Weise kann es nicht zu kollisionen kommen, wenn mehrere Roboter gleichzeit virtuelle Kopien erstellen. Es liegt damit außerdem in der Verantwortung der Roboter selbst ihre sub_id's zu verwalten.

5.5.2 Virtuelle Anführer

Bisher waren die Anführer richtige Roboter, mit dem Unterschied, dass sie über das Ziel Bescheid wussten. Der nächste Schritt wäre es, die Anführer zu entkoppeln und virtuell zu machen[28]. Der große Vorteil eines virtuellen Anführers ist es, dass er zum einen den Schwarm nicht blockiert mit seiner Anwesenheit. Zum anderen kann eine neue Technik genutzt werden die 'Stacking' genannt wird.

Stacking Als Stacking bezeichnet man es, mehrere Roboter übereinander zu stapeln, um ihren Einfluss zu erhöhen. Auf diese Weise nehmen n Roboter die Position und den Platz eines einzelnen Roboters ein. Dadurch kann man im Prinzip einen Roboter haben, der aber den Einfluss von n Robotern hat. Für diese Methode kann selbstverständlich nur

maximal ein pysischer Roboter eingesetzt werden. Die anderen die auf diesem gestackt werden, müssen virtuell sein, da deren Körper sonst zu Problemen führen würde. Natürlich kann aber auch komplett auf einen pysischen Roboter verzichtet werden und es werden ausschließlich virtuelle Roboter genutzt.

Mit Hilfe des Stacking kann also nun ein pysischer Anführer den Einfluss von n Anführern besitzen und seinen Schwarm effektiver steuern als es allein möglich wäre. Virtuelle Anführer können von einer neuen Art von Node gesteuert werden, die diese erstellt und deren Position im Schwarm pflegt oder, wenn ein pyhsischer Anführer involviert ist, können seine pyhsischen Kopien direkt von diesem gesteuert werden, indem er seine eigene Position mit mehreren IDs sendet. Dies setzt voraus, dass der Schwarm vorher dynamisch gemacht wurde. Dadurch kann ein Anführer nun mehrere Kopien von sich erzeugen und diese Anzahl von den beim Transport beteiligten Roboter abhängig machen.

Bewegung als virtueller Anführer

Zwar muss ein virtuell erstellter Anführer nicht physisch aktiv sein, dennoch muss er für die anderen Roboter so aussehen als wäre er physisch aktiv und sich entsprechend bewegen. Er wird mit seinem Schwarm mitziehen müssen und darauf achten, mit niemandem zu kollidieren, da die physischen Roboter eben dies auch tun werden.

Um einen virtuellen Anführer zu erstellen, muss im Grunde genommen nur der untere Layer ausgetauscht werden, der für die Bewegung des Motors zuständig ist und der nach einer Bewegung die Positionsdaten an die höheren Layer zurück gibt. Dieser Layer wird keinen Motor ansteuern, da die entsprechende ROS-Node sich nicht auf einem Roboter befinden wird (oder zumindest keinem physischen Körper zugeteilt wurde). Stattdesswen wird der Layer die Bewegung entgegen nehmen und die Positionsdaten zurückgeben, die bei einer perfekt ausgeführten Bewegung (ohne Rutschen oder Blockieren) zustande gekommen wären.

5.5.3 Globale Statisten

Ein anderer Ansatz um einen Schwarm zu lenken, im Gegensatz zur Nutzung eines Anführers, ist die Einführung eines still stehenden Roboters, der Statist genannt wird.

Unterschiede zum Anführer

Ein Anführer lenkt seinen Schwarm dadurch, dass er sich innerhalb des Schwarms befindet und den Schwarm auf Grundlage der Regel 'Ausrichtung: Passe deine Bewegungsrichtung deinen Nachbarn an' (Abschnitt 2.3) manipuliert. An dieser Regel anzusetzen ist ein valider Gedanke, es gibt aber auch andere Regeln die sich manipulieren lassen. Der Statist setzt an der Regel 'Zusammenhang: Versuche einen Nachbarn nahe zu sein' an und versucht einen Schwarm dadurch zu steuern, dass die Roboter sich an der Mitte ihres Schwarms ausrichten.

Grundlage des Statisten

Grundlage des Statisten ist es, dass er immer in die Berechnung der Schwarmmitte einbezogen wird, auch dann, wenn er sich außerhalb aller lokalen Reichweiten befindet. Er (oder vielleicht auch mehrere) werden somit einfach an den Ort setzen, an dem man den Schwarm braucht. Dadurch, dass dieser weit weg vom Schwarm positioniert wird, verzerrt er die Mitte des Schwarms entsprechend stark. Die Roboter werden nun versuchen wieder in Richtung der Schwarmmitte zu fahren und somit indirekt in die Richtung des Statisten.

In Abbildung 5.11 ist die Funktionsweise des Statisten dargestellt. Da der Statist nur dazu dient den Schwarmmittelpunkt zu verschieben, muss dieser kein physischer Roboter sein, sondern kann virtuell dargestellt werden. Dieser Statist kann auch das Stacking verwenden und somit seinen Effekt als Mittelpunkt-Verschieber vergrößern.

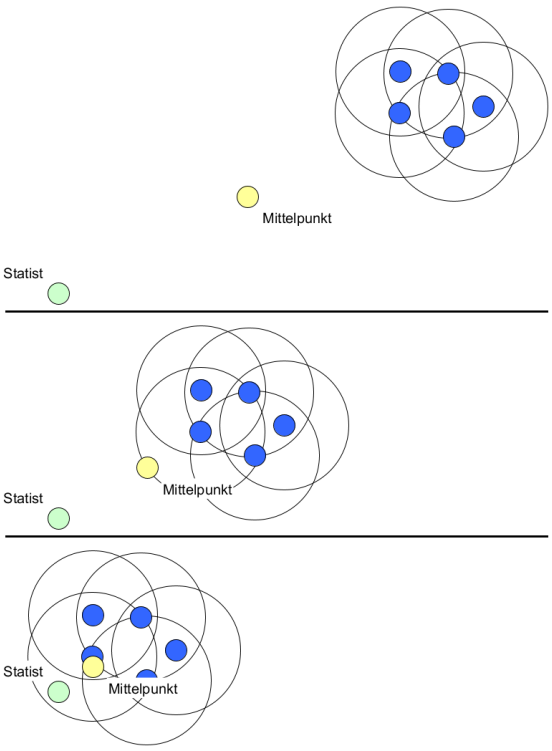

Abbildung 5.11: Funktionsweise des Statisten

Der Statist als Node

Der Statist ist leicht umzusetzen als ROS-Node-Implementierung. Er muss nur irgendwo erstellt werden, bleibt dort stehen, sendet seine Positionsdaten einmalig an die anderen Roboter und wird letztlich wieder gelöscht werden, wenn er nicht mehr gebraucht wird. Da er sich nicht an anderen Robotern orientiert muss er selbst keine Topics abonnieren. Er selbst verbreitet auch nur seine Positionsdaten als einziges Topic.

Beispiel-Implementierung Aufgrund seiner Einfachheit kann der Statist leicht von anderen Robotern erstellt werden. Als Beispiel einer Implementierung würde sich ein einfaches Objekte mit Konstruktor und Destruktor realisieren lassen.

```
1  class Statist {
2    public:
3    Statist(int id, Position pos) {
4      // send New_Robot.msg
5      // send Position.msg
6    }
7
8    ~Statist() {
9      // send Delete\_Robot.msg
10   }
11 };
```

Am Anfang der Funktion erstellt, die für die Abarbeitung eines Auftrags zuständig ist, wäre dieses Objekt quasi ein Selbstläufer der keinerlei Verwaltung brauchem würde.

Sonderregel für Statist

Der Statist ist aber aus zweierlei Sicht eine Sonderregel im Schwarm. Zum einen muss er von den anderen Robotern auch dann beachtet werden, wenn er sich nicht innerhalb deren lokalen Reichweite befindet. Zum anderen arbeitet der Roboter nur mit seiner Position, nicht mit seiner Ausrichtung (da sich diese immer wieder anpassen müsste und es den Statisten wesentlich komplizierter gestalten würde). Aus diesem Grund muss der Statist als Roboter-Typ im Schwarm bekannt sein und seine Positionsdaten gesondert behandelt werden. Die Nachricht um einen neuen Roboter zu erstellen, würde also um einen Punkt erweitert werden, der eine Typzuordnung erlaubt.

Nachrichten-Typ: New_Robot.msg

```
uint8 robot_id      // Die ID des neuen Roboters
uint8 sub_id      // Die Sub-ID des Roboters
uint8 type        // Der Typ des Roboters
```

Damit ist der Schwarm physisch gesehen immer noch homogen, virtuell gesehen aber nicht mehr, da es nun einen Roboter-Typ gibt der nur virtuell auftreten kann.

Vorteile gegenüber Anführern Der große Vorteil gegenüber der Steuerung des Schwarms mit Hilfe von Anführern ist, dass der Statist den Schwarm nicht aktiv lenken muss. Während ein Anführer den Schwarm mit seiner Ausrichtung aktiv beeinflussen muss, lenkt der Statist eher passiv, indem er den Schwarm zu sich kommen lässt. Dies braucht wesentlich weniger Ressourcen und ein Roboter braucht keine zweite Verhaltensweise.

Nachteile gegenüber Anführern Auf der anderen Seite gibt es bei dieser Methode aber auch den Nachteile, dass sie einen zweiten Roboter-Typ einführt und damit erstmals die Grenze des homogenen Schwarms übertritt. Der neue Roboter-Typ braucht gesondertes Verhaltes ihm gegenüber und er öffnet die Tür für weitere Roboter-Typen die den Schwarm nach und nach immer heterogener werden lassen.

Letztlich wird das Verhalten mit einem Statisten aber ein anderes sein. Ist ein Anführer theoretisch noch in der Lage seinen Schwarm zu lenken, und damit auch um Hindernisse herum, kann ein Statist dies allgemein nicht. Der Schwärm würde in Richtung des Statisten wandern und Hindernissen auf dem Weg gegebenenfalls mit dem normalen Algorithmus ausweichen. Dabei könnte es vorkommen, dass sie in eine Senke getrieben werden und von selbst nicht mehr herauskommen (Veranschaulichung siehe Abbildung 5.12).

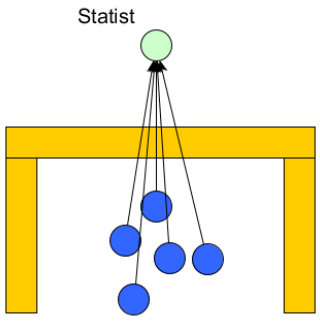

Abbildung 5.12: Roboter stecken in einer Senke fest

Andererseits könnte man mit mehreren Statisten, einem nach dem anderen, einen Weg abstecken dem die Roboter dann folgen. Es ließen sich also Etappen berechnen die die Roboter eine nach der anderen bewältigen müssen, wobei dies von einer zentralen Einheit gesteuert werden müsste. Die Auftragsvergabe oder eine andere Art von Anführer innerhalb des Schwarms wären dafür zuständig.

5.5.4 Statisten als Auftragsvergabe

Der Statist lässt sich nicht nur benutzen um einen Schwarm zu lenken. Mit seiner Besonderheit andere Roboter aus jeder Entfernung zu sich ziehen zu können, wäre es sehr interessant ihn zu benutzen um Roboter ihren Aufträgen zuzuteilen. Dies würde erfordern ihn von einer Auftragsvergabe-Node erstellen zu lassen, da diese Node noch einige weitere Aufgaben zu erledigen hätte.

Eine neue Herangehensweise an Aufträge

Den Statisten zu benutzen um Aufträge zu 'verteilen' würde im Grunde genommen bedeuten, Aufträge nicht mehr zu verteilen. Stattdessen wird ein Statist dorthin gesetzt wo sich das Transportobjekt an seinem Aufnahmeort befindet.

Der gesamte Standby-Schwarm wird nun passiv in die Richtung des Statisten, und damit auch in Richtung des Transportobjektes, gelenkt. Übertritt nun ein Roboter die Grenze des Transportobjektes und befindet sich in seinem innere, wird ihm von der Node, die auch den Statisten hält, der Auftrag zugewiesen. Andere Roboter die folgen werden ebenfalls den Auftrag zugeteilt und die Roboter werden einen gesonderten Schwarm verschoben, genau wie es auch bei der vorigen Auftragsvergabe geschehen würde. Der Statist bleibt so lange aktiv bis sich so viele Roboter gefunden haben, wie es für den Auftrag erfoderlich ist. Ist die erforderliche Zahl erreicht, wird die Vergabe-Node die Roboter mit einer `Freeze.msg` bewegungsunfähig machen und den Statisten mit der entsprechenden Nachricht löschen. Es befindet sich nun die geforderte Zahl an Robotern unterhalb des Transportobjektes und sie stehen still. Somit wurde wieder die gleiche Ausgangslage geschaffen, wie sie bei der vorigen Auftragsvergabe vorzufinden war, bevor der eigentliche Transport startete. Der Statist wurde wie ein Magnet dazu benutzt die Roboter einzusammeln die den Transport vollziehen sollen. Nun hier an kann der Transport normal ablaufen. Entweder wird ein Anführer bestimmt der das Ziel kennt, und der eventuell virtuelle Kopien von sich erschafft, oder der Transport-Schwarm wird mit Hilfe eines zweiten Statisten an sein Ziel geführt.

Entgegen der normalen Auftragsvergabe, würde man die Roboter nicht von Anfang an zuweisen, sondern man würde den Statisten als eine Art Magnet nutzen um die Roboter aufzusammeln, die am schnellsten da sind. Man würde dadurch die Roboter bekommen die am nächsten dran sind oder schlicht schneller sind, als die anderen.

Vorteile zur direkten Zuweisung Die Vorteile zur direkten Zuweisung liegen darin, dass die Roboter nicht von einem übergeordneten System ausgewählt werden müssen. Es braucht keine übergordnete Instanz die Roboter aussucht und dabei womöglich noch weitere Kriterien in Betracht ziehen muss wie Entfernung und eventuelle Unterschiede in der Geschwindigkeit der einzelnen Roboter. Die Lösung über einen Statisten ist nicht so streng wie über eine Sicherheitszone und passt sich gut in den Schwarm ein, es ist eine schöne, dynamische Lösung.

Nachteile zu direkten Zuweisung Zusätzlich zu den Nachteilen die ein Statist selbst schon mitbringt, lässt sich die Geschwindigkeit anmerken. Ohne Tests lässt sich dies zwar nicht mit absoluter Sicherheit sagen, vermutlich wird die Variante mit einem Statisten aber langsamer sein als die Version mit der manuellen Zuordnung und einer Sicherheitszone. Der Grund hierfür ist schlicht, dass die Roboter die in die Sicherheitszone flüchten dies auf direktem Wege machen, wohingegen sich die Roboter nur grob an der Mitte des Schwarms ausrichten und der Statist dadurch nicht ansatzweise die selbe Wirkung hat wie eine Gefahrenzone.

5.5.5 Ausschwärmen

Abschnitt 7.2 hat bereits gezeigt, dass es mit Hilfe von negativem Gruppendrang möglich ist einen Schwarm ausschwärmen zu lassen und sich somit zu verteilen. Dies ist nützlich, damit die Roboter während des Standbys nicht im Weg anderer Roboter oder Menschen stehen. Ein anderer Algorithmus wäre ebenfalls denkbar, der die Größe der Roboter langsam wachsen lässt.

Algorithmus

Ähnlich wie bei einem Sack voll mit Luftballons, die alle gleichzeitig aufgepumpt werden würde sich die Größe der Roboter vergrößern. Dies geschieht so lange bis der 'aufgepumpte' Körper an mindestens 3 Punkten den eines anderen Roboters oder eine andere Grenze (beispielsweise ein Gefahrengebiet) berührt. Solange es nur weniger als drei Berührungspunkte gibt, wird der Körper wachsen und der Roboter durch seinen Fluchtreflex dafür sorgen wollen, dass er nicht weiter mit anderen Objekten kollidiert. Solange es nur weniger als drei Berührungspunkt gibt, wird es immer eine Seite geben, zu der er ausweichen kann. Trifft er auf den dritten Punkt ist er eingesperrt und seine Position befindet sich so weit weg von anderen Robotern wie nur möglich.

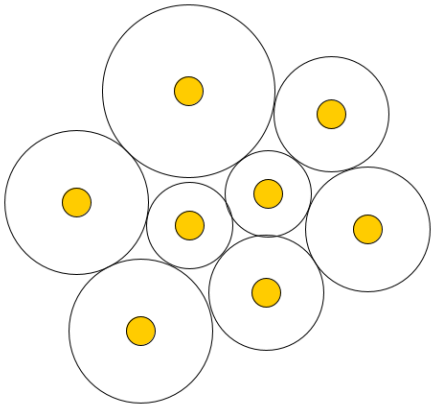

Abbildung 5.13: Skizzierung des Luftballon-Algorithmus

Umsetzung

Für die Umsetzung müsste das generelle Verhalten der Roboter nicht angepasst werden. Das verteilen des Roboters geschieht durch den Fluchtreflex der bereits implementiert ist. Es müsste nur, ähnlich beim Stillstand über das `Freeze` -Kommando, eine kleine Routine im 'Unterbewusstsein' geschaffen werden, die die interne Größe immer weiter verändert bis der Roboter keinen Weg mehr sieht aus seiner Misere herauszukommen.

Nutzung für den Transport

Auch im Transport wäre dieser Algorithmus nützlich. Finden sich die Roboter unter dem Transportobjekt ein, stehen sie still, sobald sie alle da sind. Der hier vorgestellte Algorithmus würde stattfinden, nachdem alle Roboter unter dem Transportobjekt sind. Sie würden ihre Größe anwachsen lassen und sich so möglichst gleichmäßig unterhalb des zu transportierenden Objektes verteilen. Die Grenzen der Sicherheitszone bilden dabei meist den dritten Berührungspunkt. Stehen alls Roboter still, braucht es theoretisch gesehen sogar nicht mal ein `Freeze` -Kommando, da sie sich aufgrund des beendeten Algorithmus nicht mehr weiter bewegen können dürften. Sobald das Transportobjekt auf den Köpfen der Roboter abgesetzt ist, kann die Größe wieder auf den Normalzustand zurück schrumpfen und die Roboter beginnen damit den Transport umzusetzen.

6 Implementierung

Dieses Kapitel wird sich der Implementierung der Thesis widmen. Es wird gezeigt, wie die Konzeption als Software umgesetzt wurde und einige wichtige Code-Stücke vorgestellt. Die Implementierung wurde auf dem ROS-Framework aufgebaut.

6.1 Generelles zur Implementierung

Die Implementierung baut auf Ubuntu LTS 16.04 mit der ROS-Version 'Lunar' auf, da diese beiden Komponenten das derzeit stabilste Duo bilden. Aufgrund mangelnder Kapazitäten, befindet sich das Ubuntu-System in einer Virtuellen Maschine, was, bis auf einen höheren Ressourcen-Verbrauch, allerdings keinerlei erkennbare Nachteile mit sich zog.

Generell ist das ROS-Framework in C++ und Python verfügbar, wobei diese Sprachen gemischt werden könnten, wenn sie auf verschiedenen Nodes genutzt werden. Als Programmiersprache wurde letztlich C++ verwendet, da es deutlich besser mit den verfügbaren Ressourcen umgeht. In den späteren Auswertungen hat sich diese Wahl positiv bestätigt, da 25 gleichzeitige Roboter eine Menge Ressourcen verschlungen haben und den verfügbaren Computer (auch wegen der VM) bereits an die Grenzen seiner Leistung brachte.

Da ein experimentelles Vorgehen direkt an realen Robotern zu Aufwändig wäre, insbesondere was die Zeitkosten für die Durchführung und Auswertung einer Simulation angeht, beruht die Implementierung auf der Node Turtlesim[23] von ROS (siehe **??**). Die Turtlesim wurde dahingehend angepasst, dass die Bilder der Turtles durch wesentlich kleinere Bilder schwarzer Pfeile ersetzt wurden die visuellen Aufschluss auf Position und Aurichtung erlauben. Dadurch wurde mehr Platz und Übersichtlichkeit gescahffen. Das Verhalten der Turtles selbst in der Simulation ist unverändert.

Grundsätzlich ist die Turtlesim nicht dafür gemacht worden um aufwändigere Simulationen zu erproben. Der Zweck ist es einen einfachen Einstieg in das ROS-Framework zu geben und Neulinge durch die Tutorials zu begleiten. Sie bietet aber eine grafische Oberfläche, die bei der Implementierung sehr nützlich ist, insbesondere um Fehler besser entdecken zu können. Bei der Implementierung wurde viel Wert auf eine modulare Bauweise gesetzt. Die Bewegungsbefehle die aktuell an die Turtlesim gehen sind abstrahiert und lassen sich später schnell auf das System des jeweiligen Roboters umschreiben. Andere Nachrichten gehen nicht über die Turtlesim, wodurch die Bewegung das einzige Sub-System ist, dass es nachher anzupassen gilt.

6.2 Nachbau des Schwarms nach Craig Reynolds

Sendet man einen Bewegungs-Befehl an die Turtlesim, wird diese den Befehl für 1 Sekunde ausführen und nach Abschluss die Position des Roboters automatisch an die Subscriber verteilen. Die Subscriber erhalten in ROS ihre Informationen immer über Callback-Funktionen. Da diese Funktionen statisch sein müssen, ist es nicht einfach möglich die Daten in einem Objekt zu speichern, sondern sie müssen in globalen

Variablen untergebracht werden. Die Informationen über die Position der Roboter werden in einem Array gespeichert. Da die Roboter IDs haben, ist es leicht diese bei 0 beginnen zu lassen und sie dann als Index zur Adressierung innerhalb des Arrays zu nutzen.

Da Roboter eine Position und eine Fläche haben die deren Körper darstellt, macht es am meisten Sinn ihnen einen Kreis als geometrische Form zu vererben, sodass letztlich die Klassen-Struktur aus Abbildung 6.1 entsteht (Funktionen wurden ausgelassen). Der Nutzen der Klasse `Angle` ist der, dass die Klasse automatisch darauf achtet im Bereich [0, 360] zu bleiben: `Angle(350) + Angle(20) == Angle(10)`.

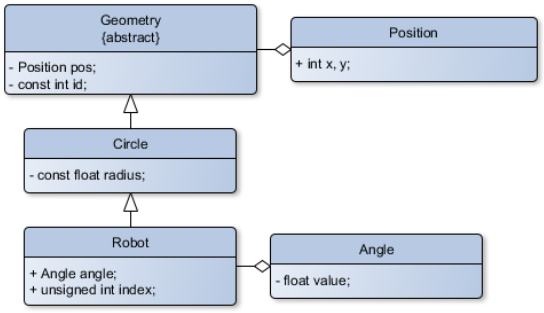

Abbildung 6.1: Die Klassenhierarchie eines Roboter-Objektes

Messages

Die Steuerung der Roboter innerhalb der Turtlesim geschieht nicht direkt durch das eigene Programm. Stattdessen müssen Nachrichten an die Turtlesim gesendet werden die dann letztlich die Roboter bewegt. Insgesamt werden dafür 5 Nachrichten verwendet. Die Nachrichten-Typen werden der Übersichtlichkeit halber in einer verkürzten Schreibweise vorgestellt, wie sie in ROS nicht erlaubt, von Programmiersprachen wie C++ oder Java, aber bekannt ist. Einige Nachrichten arbeiten mit dem Topic 'turtleX', was nichts anderes bedeutet als eine ID für die Roboter der Turtlesim (turtle1, turtle2, ...).

turtlesim/Pose.msg

```
// Empfangen von dem Topic: turtleX/pose
// Gibt Feedback zur Position und Ausrichtung des Roboters
// Wird in dieser Arbeit genutzt, um die Position der Roboter zu speichern
float32 x, y, theta, linear_velocity, angular_velocity
```

geometry_msgs/Twist.msg

```
// Gesendet an das Topic: turtleX/cmd_vel
// Fahrbefehl fuer den Roboter, wird 1 Sekunde lang ausgefuehrt
// Wird in dieser Arbeit genutzt, um die Roboer geradeaus zu fahren zu lassen
Vector3  linear, angular
```

turtlesim/TeleportRelative Service

```
// Gesendet an den Service: turtleX/teleport_relative
// Teleportiert den Roboter zu einem relativen Ort
// Wird in dieser Arbeit genutzt, um die Roboter vor dem Fahrbefehl zu drehen
float32 linear, angular
---
```

turtlesim/Spawn Service

```
// Gesendet an den Service: spawn
// Erschafft einen neuen Roboter in der Turtlesim
float32 x, y, theta
string name
---
string name
```

turtlesim/SetPen Service

```
// Gesendet an den Service: turtleX/set_pen
// Schaltet den Pen aus. Ein 'Stift' der den gefahrenen Weg markiert
uint8 r, g, b, width, off
---
```

Unterscheiden der einzelnen Pose-Callbacks

Die Turtlesim verbreitet die Position der Roboter nach jedem Bewegungsbefehl. Leider ist in der Nachricht Pose.msg keinerlei Information darüber enthalten, von wem die Nachricht kommt.

Callback-Funktion für Pose.msg

```
1 void flockCallback(const turtlesim::PoseConstPtr &msg);
```

Um die Nachrichten nun den einzelnen Robotern zuordnen zu können, gäbe es nun die Möglichkeit jedem Callback eine eigene Callback-Funktion zuzuordnen. Diese Methode wäre allerdings Handarbeit und nicht dynamisch. Braucht man mehr (oder weniger) Roboter, müsste man die Funktionen eigenhändig umändern. Einen kleinen Trick um dies nicht tun zu müssen, bietet die Boost-Bibliothek mit der ROS arbeitet. Mit Hilfe dieser kann man einer Methode einen weiteren, festen Parameter zuordnen.

Callback-Funktion für Pose.msg mit Boost

```
1  void flockCallback(const turtlesim::PoseConstPtr &msg, const unsigned int index);
2
3  std::vector<ros::Subscriber> flockSubscriber;
4  void subscribeToFlock() {
5      ros::NodeHandle node;
6
7      for (int i = 0; i < FLOCK_SIZE; i++) {
8          boost::function<void(const turtlesim::PoseConstPtr &)> callback =
               bind(flockCallback, _1, i);
9          flockSubscriber.push_back(node.subscribe("turtle" + std::to_string(i) +
               "/pose", 100, callback));
10     }
11 }
```

Dadurch lassen sich alle Callbacks auf die selbe statische Methode verlinken, bei Aufruf wird jedoch immer ein zusätzlicher, fester Parameter mit eingebunden. Die Callbacks können nun in einer Schleife abgearbeitet werden und die Dauer der Schleife als konstanter Parameter codiert werden. Eine ähnliche (wenn nicht sogar die selbe) Klasse hat es mit dem Standard C++11 auch in den STL-Header <functional>

geschafft[35]. Da ROS aber intern mit der Version von Boost arbeitet[34][21], ist man dazu gezwungen deren Version zu nutzen und nicht die der STL.

System-Takt: Ticks

Der Ablauf eines Programms geschieht in Ticks (der Takt des Systems, periodische Zeitabstände), in denen zuerst eine Aktion gestartet und danach eine kurze Zeit gewartet wird um Nachrichten anderer Roboter zu empfangen und zu verarbeiten. Es ist damit implizit nur eine Bewegungs-Aktion pro Tick möglich. Diese kann allerdings Drehung und Fahren kombinieren. Die Bewegung eines Roboter wird von ROS genau 1 Sekunde lang ausgeführt[1]. Die Steuerung der bewegten Entfernung pro Tick lässt sich daher nur über die Geschwindigkeit der Bewegung steuern. In der Praxis hat sich aufgrund von weiteren Verzögerungen in der Turtlesim eine Tick-Länge von 1.1 Sekunden bewährt um sicherzustellen, dass die `Pose.msg` der Turtlesim auch rechtzeitig ankommt. ROS bietet dafür eine bequeme Funktion mit seiner Klasse `ros::Rate` und deren Funktion `sleep()`.

Ticks abschließen mit ROS Funktionen

```
1 ros::Rate sleeper(0.9);
2
3 while (true) {
4   // do something
5   sleeper.sleep();
6   ros::spinOnce();
7 }
```

Der Konstruktor nimmt einen float entgegen der die Frequenz angibt. Die wirkliche Länge der Ticks beträgt also 1.111 Sekunden. Der Funktionsaufruf `sleeper.sleep()` blockiert solange, bis der letzte Aufruf (innerhalb des gleichen Objektes) 1.111 Sekunden her ist. Braucht das Programm zwischen den Aufrufen länger als die Periode lang ist, kehrt die Funktion entsprechend sofort zurück. Während der Wartezeit gehen (hoffentlich) alle neuen Positionsdaten der anderen Roboter ein. Nach der Wartezeit werden diese dann abgearbeitet und das Programm arbeitet an seinem nächsten Zyklus.

Entschließt sich ein Roboter dazu sich nicht zu bewegen, muss er einen 'leeren' Bewegungsbefehl senden, der dazu führt, dass sich der Roboter nicht bewegt und somit einen Tick abschließen, statt einfach zu warten und nichts zu senden. Der Grund hierführ liegt darin, dass die Systeme Single-Thread Architekturen sind und das Abfragen der Topics explizit ausgelöst werden muss. Dies geschieht über die Funktion `ros::spinOnce()` von ROS. Diese Funktion bewirkt, dass ROS alle Callback-Funktionen der entsprechenden eingegangenen Events aufgerufen werden. Es werden also die bisher eingegangenen `Pose.msg` s durch ihre Callbacks verarbeitet. Außerdem werden dadurch die eigenen Daten auf den Robotern aufgefrischt und eventuelle Informationen gesendet, die nichts mit der Bewegung zu tun haben. Das Abschließen eines Ticks synchronisiert somit generell die Daten des eigenen Systems mit dem der anderen.

Die Ticks sind trotz allem keine globalen Ticks, die jeden Roboter gleichzeitig betreffen. Jeder Roboter arbeitet auf seiner 'Zeitlinie' und agiert unabhängig der Ticks der anderen Einheiten.

[1] `http://wiki.ros.org/turtlesim#Subscribed_Topics`

Einhalten der 4 Grundregeln

Die Roboter wurden als autonome Einheiten konzipiert die sich ausschließlich anhand von 3 fixen Parametern bewegen und dabei versuchen die 4 Grundlegen des Schwarmverhalten einzuhalten. Grundsätzlich werden von einem Roboter nur andere Einheiten beachtet, die in einem gewissen Radius um ihn herum positioniert sind. Daher wird Am Anfang eine Liste derjenigen erstellt. Dies geschieht durch simples iterieren über die Liste aller Roboter und einer Berechnung über Pythagoras.

Ausrichtung: Passe deine Bewegungsrichtung deinen Nachbarn an

Die Turtlesim selbst arbeitet bei Winkeln mit dem Bereich von [-3.14, 3.14], wohingegen die Steuerung der Roboter mit dem Bereich [0°, 360°] arbeitet, da dieser besser bearbeitet (und von Menschen gelesen) werden kann. Außerdem ist dadurch generell eine Abstraktions-Ebene entstanden, die es später erlaubt mit jedem bliebigen anderen Winkel-System zu arbeiten. Eintreffende Positionsdaten, von anderen Robotern, müssen also erst in die interne Darstellung umgerechnet werden. Ebenso wird der Winkel vor dem Senden eines Bewegungs-Befehls automatisch so umgerechnet, dass die Turtlesim ihn akzeptiert. Wird der Turtlesim ein zu großer Winkel gesendet, z.B.: 5, so wird der überschüssige Winkel einfach abgezogen und es wäre eine Drehung um 1.86. In Abbildung 6.2 ist eine Übersicht über die Winkel zu sehen wie sie im System und von der Turtlesim genutzt werden.

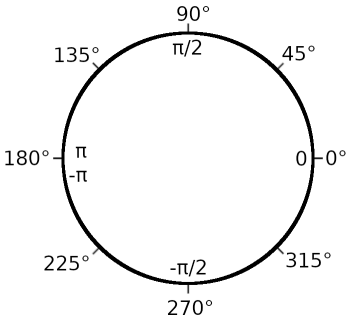

Abbildung 6.2: Winkel-Übersicht

Berechnung der mittleren Ausrichtung eines Schwarms

```
Angle meanAngle(const std::vector<Angle> &angles) {
    double y_part = 0, x_part = 0;

    for (const Angle &angle : angles) {
        x_part += std::cos(angle.get() * M_PI / 180.0f);
        y_part += std::sin(angle.get() * M_PI / 180.0f);
    }

    return Angle(x_part / angles.size(), y_part / angles.size());
}

Angle::Angle(float vec_x, float vec_y) {
    this->value = std::atan2(vec_y, vec_x) * 180.0f / M_PI;
    normalize(); // value = value % 360.0f;
}
```

Zusammenhang: Versuche deinen Nachbarn nahe zu sein

Für die Berechnung des Mittelpunkts der Nachbarschaft, wurde schlicht die Liste der Roboter, die nahe genug sind, genommen und ein mathematischer Durchschnitt gebildet. Die Koordinaten wurden danach von den eigenen abgezogen und die resultierenden X/Y-Werte als Vektoren genutzt um den Winkel zu berechnen, den der Mittelpunkt zum Roboter hat. Anschließend wurde die Differenz dieses Winkels zu dem eigenen Winkel gebildet, den man vorher fahren wollte. Von diesem entstandenen Winkel wird nun ein

prozentualer Wert genommen (je nach Einstellung des Parameters) und vom ursprünglichen Winkel abegzogen.

Berechnung des Fahrtwinkels

```
Angle RobotPilot::nextStep() {
  const std::vector<Robot> &flock = findFlock(self());
  Angle turnAngle = averageDegree(flock);
  addFreeWill(turnAngle);

  const Angle &angleToFlockCentre = self().angleTo(flockCentre(flock));
  turnAngle += (angleToFlockCentre - turnAngle) * FLOCK_TURN_TO_FLOCK_PERCENT;

  return turnAngle;
}
```

Abschottung: Vermeide Kollisionen mit deinen Nachbarn

Um Kollisionen mit anderen Robotern zu vermeiden, werden zur Kollisionserkennung interne Simulationen verwendet. Im Grunde bedeutet dies, dass berechnet wird wo der Roboter mit dem aktuell gewünschten Winkel und normaler Geschwindigkeit hinfahren wird. Ist dieser Ort weit genug von anderen Robotern entfernt, wird er akzeptiert. Ist er es nicht, wird der gewünschte Winkel in Schritten von 1° erst nach links und dann nach rechts verändert und erneut geschaut ob es am Zielort zu einer Kollision kommen wird.

Simulation auf Kollisionen am Zielpunkt

```
bool RobotPilot::predictRobotCollision(const Position &destination) const {
       for (const Robot &robot : globalSwarm) {
           if ((robot != self()) and robot.inBoundary(destination)) {
               return true;
           }
       }

       return false;
}
```

Unzureichende Erkennung auf dem Weg Um die Berechnung einfacher zu gestalten und Ressourcen zu schonen wurde nicht der Weg selbst überprüft. Es ist also möglich, dass ein Roboter weder auf seinem Ursprungs- oder Zielort mit anderen Robotern kollidiert, wohl aber auf dem Weg dorthin. Ebenfalls ist es in der Turtlesim nicht möglich, eine angefangene Bewegung wieder abzubrechen. Kommt es also während der Ausführung zu einer Kollision, z.B. weil sich zwei Roboter aufeinander zubewegen, kann diese nicht mehr verhindert werden. Ein Ausgleich in der Simulation ist durch eine Abfrage gegeben, ob sich zwei Roboter 'ineinander' befinden. Die entsprechenden Roboter werden dann auf direkter Luftlinie voneinander weggeschoben, um wieder einen plausiblen physikalischen Zustand einzunehmen.

Im der Praxis sollte eine Kollision auf dem Weg kein Problem darstellen, da mit parallelen Threads nebenläufiges Verhalten eingesetzt werden kann. Außerdem ließe sich, im Falle einer Single-Thread-Architektur, die Frequenz der Ticks wesentlich schneller eingestellen, sodass sich die Roboter effektiv nur wenige Millimeter pro Tick bewegen, dafür aber mit über 100 Ticks pro Sekunde.

Flucht: Fliehe vor Dingen, die eine potentielle Gefahr darstellen

Kollisionen mit Gefahrenzonen werden erkannt, indem über ein globales Array von `Object`-Klassen iteriert wird und die einzelnen Objekte nach Kollisionen auf den genannten Koordinaten gefragt werden. Es muss nur die `Object`-Klasse nach einer Kollision befragt werden, diese leitet die Anfrage an alle Geometrien weiter. Wird eine Kollision bei einem Objekt erkannt, wird sie genau wie bei der Kollision mit Robotern durch iterativ steigendes Ausweichen zu beiden Seiten verhindert. Um Gefahrenzonen darzustellen, wird auf Abstraktion und Vererbung gesetzt. Es gibt, wie in Abbildung 6.3 zu sehen ist, eine Hierarchie von Klassen aus denen sich die Zonen zusammensetzen lassen. Die `Geometry`-Klassen die sich in der `Object`-Klasse befinden, werden alle von dieser gesteuert. Bewegt sich das Objekt, bewegen sich alle darin enthaltenen Geometrien. Das Hinderniss bleibt dadurch immer in einem plausiblen Zustand.

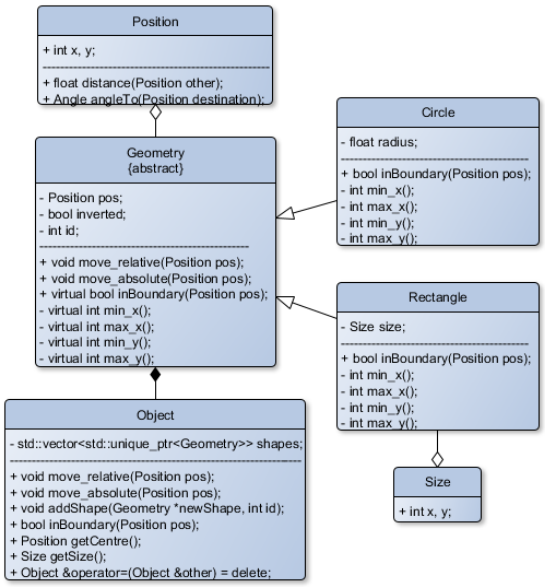

Abbildung 6.3: Die Klassenhierarchie eines Hindernisses

6.3 Anführer

Da der Anführer seinen Schwarm lenkt, verfällt die Einheit, die zum Anführer wurde, in ein spezielles Verhalten. Dieses Verhalten ist ein neuer Zweig im Programm der Roboter, welches von dem Verhalten des normalen Schwarmroboter gänzlich abweicht. Auf das Ziel zuzusteuern und den Schwarm dabei nicht zu verlieren, ihn sogar wieder einzufangen, wenn er abhanden kommt, kann mit den bisherigen Schwarmregeln nicht umgesetzt werden. Aus diesem Grund ist die Implementierung weniger ein umändern der bisherigen Implementierung, sondern es wird eher ein neues Verhalten hinzugefügt und eine Abzweigung im Code um dieses auszuführen.

Generell ist der Anführer frei von den Verhaltensweisen des sonstigen Schwarms. Weder verfügt er über einen freien Willen, noch versucht er seinem Schwarm nahe zu sein, in dem Sinne in dem es bisher Implementiert wurde. Auch die Orientierung an der Ausrichtung anderer Roboter entfällt vollständig, da er sich auf sein Ziel ausrichten muss.

Generelles Verhalten

Abbildung 6.4 zeigt das Verhalten wie es umgesetzt wurde. Der Anführer ist ein Zweig im normalen Verhalten der Roboter. Erhält ein Roboter eine Nachricht vom Typ `New_Mission` und seine ID entpricht der ID in der Nachricht, richtet er sich nach diesem Ziel aus und fährt ihm entgegen. Behindert ein anderer Roboter, dass der Anführer geradeaus weiter kann, verringert er seine Geschwindigkeit oder bleibt letztlich einen Tick lang stehen. Ist er nahe genug am Ziel angekommen, fällt er in sein altes Verhalten zurück. Abbildung 6.4 zeigt das implementierte Verhalten des Anführers.

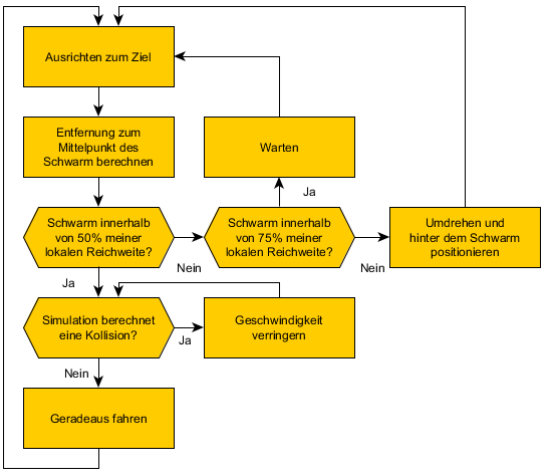

Abbildung 6.4: Ablaufdiagramm Schwarm

Das Einfangen des Schwarms, wenn dieser sich zu weit entfernt, ist widerum eine Subroutine im Verhalten des Anführers. Ist diese Routine gestartet, bleibt er in diesem Zustand, bis sie vollständig ausgeführt ist und der Schwarm wieder eingefangen wurde.

Einbindung des Anführers in ROS

Die Einbindung der Anführer in ROS geschah mit einem zusätzlichen Subscriber der das Topic `flock/mission/new` abonniert hat. Die Funktion des entsprechenden Callbacks nimmt die `New_Mission` an und speichert sie, wenn die ID in der Mission der ID des Roboters entspricht, in einer globalen FIFO-Struktur. Ein Roboter überprüft diese Struktur in seinem normalen Ablauf immer wieder und arbeitet die Mission ab, wenn er eine findet.

Da die Implementierung als Single-Thread-Architektur implementiert ist, braucht es keine entsprechenden Vorsichtsmaßnahmen, wie sie bei Multi-Thread üblich sind. Die Struktur der Nachricht konnte für die Implementierung so beibehalten werden, wie sie in der Konzeption vorgegeben wurde.

Nachrichten-Typ: New_Mission

```
uint8 leader_id // Die ID des ausgewaehlten Anfuehrers
float32 pos_x   // Position des Ziels entlang der X-Achse
float32 pos_y   // Position des Ziels entlang der Y-Achse
```

6.4 Transport von Waren mit Hilfe eines Schwarms

Da der Schwarm und die Anführer implementiert sind, galt es nun diese zu verbinden und damit zu erlauben, den Schwarm zu benutzen um Transportaufträge zu erledigen. Dazu mussten aber noch einmal das Verhalten der Anführer und die Implementierung der Roboter allgemein angepasst werden.

Erteilen von Aufträgen

Um die Aufträge erteilen zu können, musste der Nachrichten-Typ angepasst werden. Hier konnte wieder die direkte Vorlage aus der Konzeption übernommen werden.

Nachrichten-Typ: New_Mission

```
uint8 robot_index_from
uint8 robot_index_to

uint8 leader_number
uint8 mission_id

float32 object_position_x
float32 object_position_y

float32 object_size_x
float32 object_size_y

float32 target_x
float32 target_y
```

Erhält ein Roboter diese Nachricht, wird zunächst überprüft, ob die eigene ID in im vorgegebenen Intervall enthalten ist. Anschließend wird das Transportobjekt erstellt. Ist man dem Auftrag nicht zugeteilt, wird das Transportobjekt als normales Objekt mit entsprechender ID erstellt und stellt somit eine Gefahrenzone dar, von der sich der Roboter fern halten wird. Wurde man dem Auftrag hingegen zugeteilt, wird das Objekt invertiert und damit als Sicherheitszone erstellt.

Teilen des eigenen Schwarms nach dem Erhalt eines Auftrags

```
1  void missionNewCallback(const swarmRobot::MissionNewConstPtr &mission,
2    const unsigned int index) {
3      const std::pair<unsigned int, unsigned int> range(mission->robot_index_from,
           mission->robot_index_to);
4
5      if (alg::inRange<unsigned int>(index, range.first, range.second)) {
6          missions.push_back(mission);
7
8          mySwarmIndices.clear();
9          for (unsigned int i = range.first; i < range.second; i++) {
10             mySwarmIndices.push_back(i);
11             robotsNotInPosition.push_back(i);
12         }
13     } else {
14         auto newEnd = std::remove_if(mySwarmIndices.begin(),
               mySwarmIndices.end(), [range](unsigned int i) {
15             return alg::inRange<unsigned int>(i, range.first, range.second);
16         });
17         mySwarmIndices.erase(newEnd, mySwarmIndices.end());
18     }
19 }
```

Anführer wird jeder Roboter, dessen ID sich im Bereich [robot_index_from, robot_index_from + leader_number[befindet. Die Anführer müssen sich nicht erkenntlich machen, es reicht aus wenn sie es selbst wissen, entsprechend muss hier auch keine Nachricht zur Bekanntmachung versendet werden. Zwar könnten die

Nicht-Anführer eine Notiz abspeichern wer ein Anführer geworden ist, da dies im Algorithmus aber keine Rolle spielt, und sogar ein wenig entgegen dem Schwarmverhalten geht, ist es schlicht nicht notwendig.

In der Implementierung wird der Auftrag von einer neuen ROS-Node gesendet. Diese wird gestartet, stellt die entsprechenden Verbindungen zu ROS her, sendet die Nachricht und beendet sich anschließend wieder.

Füllen eines Raums

Das generelle Füllen des Raumes, ist bereits durch die Flucht- und Ausweich-Algorithmen des Schwarms implementiert. Neu hinzu kamen hier nur die beiden Nachrichten `Robot_Freeze` und `Robot_Continue`, welche direkt aus der Konzeption übernommen und Implementiert werden konnten.

Speziell `Robot_Freeze` hat dabei eine besondere Bedeutung. Dieser Nachrichten-Typ schaltet einen Boolean, der dafür sorgt, dass Fahrbefehle nicht mehr gesendet werden und stattdessen der Roboter einen Tick lang still steht. Damit greift dieser Befehl direkt in die unteren Layer der Architektur ein, in dem der Roboter per Software nicht mehr in der Lage ist sich zu bewegen. Abgesehen von Transportaufträgen kann er auch bei anderen Dingen nützlich sein, beispielsweise bei der Wartung oder generell um Roboter aus dem Verkehr zu nehmen. In der Turtlesim-Implementierung wird schlicht der Fahrbefehl nicht mehr an die Turtlesim gesendet, in der realen Implementierung wird entsprechend der Motor nicht mehr angesprochen. Der Roboter selbst wird davon nichts wissen und nach wie vor seine Routen berechnen und versuchen Teil des Schwarms zu sein. Er wird sich dabei nur nicht von der Stelle bewegen. `Robot_Continue` schaltet diesen Boolean wieder zurück, sodass Fahrbefehle wieder durch kommen (bzw. der Motor wieder angesprochen wird) und der Roboter sich wieder normal bewegt.

Der Rest des Transports ist im Grunde schon implementiert. Die Anführer werden sich Ausrichten, entsprechend dem Winkel, den das Transportobjekt zum Ziel hat und losfahren. Versperrt ihnen ein anderer Roboter den Weg, werden sie langsamer oder bleiben gar stehen, ändern aber keinesfalls die Richtung.

Am Ziel angekommen, wird der erste Anführer wieder den Schwarm einfrieren und nach einem gegebenen Event losfahren lassen. Das Transportobjekt wird durch den ersten Anführer aus den Speichern der anderen Roboter gelöscht werden und der Transportauftrag wird abgeschlossen sein. Da die Liste der Aufträge eine simple Queue ist, können sich darin mittlerweile neue Aufträge befinden. Ist dies der Fall, werden diese abgearbeitet. Wenn nicht, werden sich die Roboter verteilen oder anderweitig versuchen nicht im Weg zu stehen.

6.5 Änderungen für die Implementierung auf einem Roboter

Da bisher die Implementierung auf der Turtlesim basierte, ist folgend ein Abschnitt darüber, was zu verändern wäre, wenn das Konzept auf einen Roboter übertragen wird.

Bewegung

Die Bewegung eines Roboters wurde bisher erreicht, indem eine Nachricht vom Typ `Twist.msg` an die Turtlesim gesendet wurde. Dieser Aufbau ist für einen späteren Einsatz unvorteilhaft weil es eine neue Node voraussetzt die für die Bewegung zuständig ist. Stattdessen muss ein neuer Layer erschaffen werden der das

Roboter-System mit dem Hardware-System verbindet. Eine Bewegung wird dann dadurch ausgeführt, dass ein Motor angesprochen wird und dieser Feedback über die zurückgelegte Strecke gibt. Dies setzt natürlich vorraus, dass der Motor (oder das Sub-System das für den Motor zuständig ist) in der Lage ist dies zu erkennen. Die wichtigsten Faktoren, neben der trivialen Berechnung über Dauer der Bewegung und Umfang des Rades, sind dabei ob der Motor blockierte oder durchdrehte. Sollte der Motor (bzw. das entsprechende Subsystem) nicht in der Lage sein dies zu erkennen, muss es innerhalb des Systems durch andere Faktoren ausgeglichen werden, beispielsweise durch ein IPS. Das Feedback darüber in welcher Position sich der Roboter befindet ist für das Verhalten jedoch essentiell und eine Grundvoraussetzung and die Hardware eines späteren Roboters.

Position aktualisieren

Nicht nur die Bewegung selbst ging bisher über ROS, sondern auch die aktualisierung der Positionsdaten. Die kamen bisher von der Turtlesim über die Pose.msg . Die Roboter werden diese Nachrichten nun selbst verbreiten müssen, bestensfalls direkt im Anschluss an die Bewegung, nachdem das Feedback zur neuen Position kam.

7 Evaluation

In diesem Kapitel werden die Ergebnisse der einzelnen Phasen der Thesis beschrieben. Nach jeder Phase wurden Simulationen mit der Implementierung gestartet und verschiedene Informationen aufgezeichnet. Der Hauptgrund hierfür ist, dass durch die Ergebnisse zum einen ein allgemeines Gefühl für den Schwarm gegeben werden kann und wie er einzustellen ist. Zum anderen liefern die Beobachtungen wichtige Informationen für die späteren Phasen der Konzeption. In diesem Kaputel werden nicht alle Statistiken vorgestellt, sondern nur die, die als interessant empfunden wurden.

7.1 Generelles zur Evaluation

Da die Roboter des Schwarms einen gewissen freien Willen haben, der entscheidet wo sie hinfahren, und dieser schlicht Zufall ist, werden die Simulationen mit ihren Parametern immer mehrmals durchgeführt. Um Ausreißer in der Auswertung auszumerzen, wurde ebenfalls nicht mit einem normalen Durchschnitt gerechnet, sondern mit einem Durchschnitt über die mittleren Ergebnisse.

Da die Simulationen mit über 5 Parametern weit mehr als 100 verschiedene Einstellungen zulassen, konnte nicht jede Konfiguration getestet werden. Stattdessen wurden die Einstellungen für die Simulationen durch Überlegung und den Ergebnissen der vorangegangenen Simulationen ausgewählt um die Anzahl im Bereich des zeitlich machbaren zu halten.

7.2 Nachbau des Schwarms nach Craig Reynolds

Nachdem die Implementierung des Schwarms abgeschlossen ist, galt es diese zu testen und Statistiken zu erheben. Dazu wurde die Simulation mit verschiedenen Parametern gestartet und 25 Roboter zufällig in der Simulation verteilt. Die Simulationen wurden für 1 Stunde laufen gelassen und in 10s-Abständen gemessen, wie viele Schwärme sich gebildet haben. Ein Schwarm war hierbei eine zusammenhängende Kette von Robotern, die nicht weiter als ihre lokale Reichweite voneinander entfernt sind. Das bedeutet, dass alle Roboter innerhalb eines Schwarm die anderen beeinflussen. Teilweise mag dies direkt geschehen sein, wenn sie innerhalb der lokalen Reichweite waren. Aber auch indirekte Beeinflussung ist es möglich, indem Roboter beeinflusst wurden, durch Roboter die von anderen beeinflusst wurden. Abbildung 7.1 zeigt ein Beispiel mit 3 Schwärmen. Die Roboter sind als blaue Kreise eingezeichnet, ihre jeweilige lokale Reichweite als schwarzer Kreis drum herum. Eine Zahl im inneren zeigt die Zugehörigkeit zum jeweiligen Schwarm.

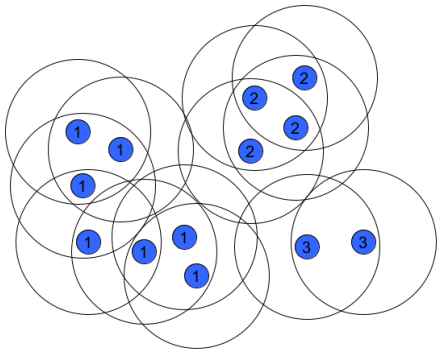

Abbildung 7.1: Einordnung der Roboter in Schwärme

Da das Verhalten der Roboter, aufgrund ihres freien Willens und der zufälligen Platzierung in der Simulation, starken Schwankungen ausgesetzt ist, wurden immer 10 Simulationen gestartet und aus den mittleren 6 Werten ein Durchschnitt gebildet. Dadurch wurden Ausreißer eliminiert und die Statistiken können sinnvolle Mittelwerte zeigen. Der Rand der Simulation wurde als Hinderniss gewertet, was bedeutet, dass die Roboter nicht stumpf weiter gefahren sind nachdem dieser erreicht wurde, sondern sie versucht haben diesem auszuweichen.

Statistik: Freier Wille

Tabelle 7.1: Eigenschaften des Schwarms

Größe des Schwarms:	25 Roboter
Lokale Reichweite:	5% der Größe der Simulation
Geschwindigkeit:	10% der lokalen Reichweite
Drang zur Gruppierung:	0%
Freier Wille:	Variabel

In Abbildung 7.2 zu sehen, ist eine Statistik die die Entwicklung von Schwärmen mit verändertem freien Willen zeigt. Über die X-Achse hinweg ist die Zeit der Simulation von 0-60 Minuten, die Y-Achse zeigt die Anzahl der vorhandenen Schwärme nach obiger Definition. Grundsätzlich wäre es innerhalb der Simulation aufgrund des verfügbaren Platzes möglich gewesen, dass jeder Roboter nur sich selbst als Schwarm hat. Neben den Graphen selbst, die die Entwicklung der Schwärme zeigen, ist ebenfalls ein linearer Trend in gleicher Farbe und gestrichelter Linie eingezeichnet worden.

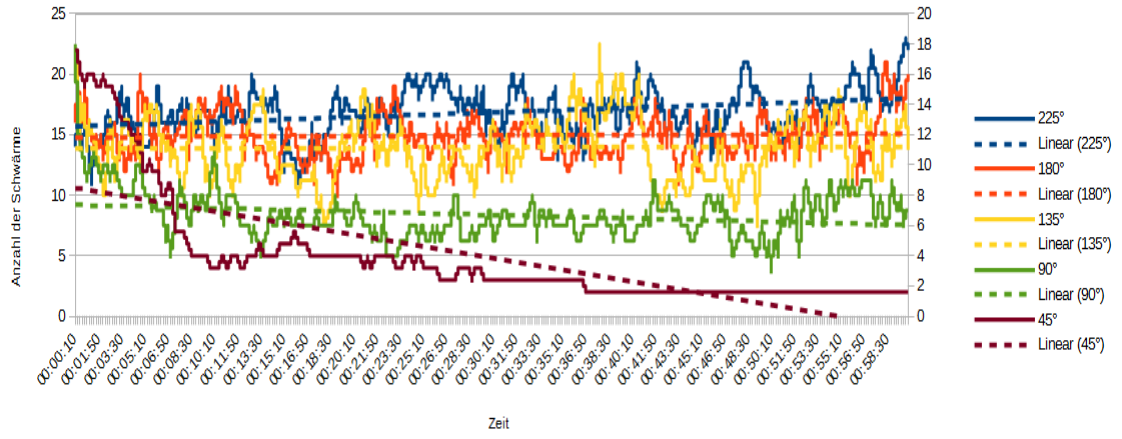

Abbildung 7.2: Entwicklung der Schwärme in Abhängigkeit ihres freien Willens

Zu sehen ist, dass der Schwarm bei einem freien Willen von 90° und weniger dazu neigt sich zu gruppieren. Der Drang sich nach seinen Nachbarn auszurichten und in die selbe Richtung wie diese zu fahren, ist deutlich größer als der freie Wille der sie auseinander bringen würde. Auf lange Sicht wäre die Zahl der Schwärme Richtung 1 gesunken, da sich die Roboter immer weiter zusammengeballt haben. Dies ist vor allem bei 45° freiem Willen zu sehen, da die Schwärme, im Gegensatz zu 90°, kaum noch aufbrechen und relativ stabil bleiben. Während einigen der einstündigen Stimulationen trat dieser Fall sogar ein. Will man seine Roboter zusammenballen, ist es also sinnvoll den freien Willen geringer zu halten.

Bei 180° und 135° freiem Willen ist die Grenze, an der die Schwärme von Anfang bis Ende nahezu konstant bleiben. Sie brechen genauso schnell auseinander wie sie zusammen kommen, wodurch sich am Gesamtzustand der Schwärme wenig verändert. Dieser Breich ist damit gut geeignet um den Schwarm neutral zu lassen und weder zusammenballen zu lassen, noch zu verstreuen.

Ab einem freien Willen von 225° (112.5° in beide Richtungen) zeigt sich, dass die Roboter eher dazu neigen sich zu verteilen statt sich zu sammeln. Der Einfluss der Regel, sich wie seine Nachbarn auszurichten, macht weniger als 50% bei der Berechnung des endgültigen Winkels aus und hat gegenüber dem freien Willen somit das Nachsehen.

Warum haben sich trotzdem Schwärme gebildet? Einerseits resultiert es daraus, dass aus der Menge [-112.5°, 112.5°] der letztliche Wert zufällig ermittelt wurde, der Durchschnitt also um 0° herum liegt, mit einer durchschnittlichen Abweichung von 56.25° zu beiden Seiten. Anderseits spielt aber auch der mangelnde Platz zum verteilen eine Rolle. Roboter die zufällig in die lokale Reichweite anderer Roboter gefahren sind, werden im nächsten Tick als Schwarm erkannt, unabhängig vom Grund warum sie zusammen waren.

Statistik: Gruppendrang

Tabelle 7.2: Eigenschaften des Schwarms

Größe des Schwarms:	25 Roboter
Lokale Reichweite:	5% der Größe der Simulation
Geschwindigkeit:	10% der lokalen Reichweite
Drang zur Gruppierung:	Variabel
Freier Wille:	90° bzw 225°

In Abbildung 7.3 und Abbildung 7.4 zu sehen, sind Statistiken die die Entwicklung von Schwärmen mit verändertem Drang zur Gruppierung zeigen. Über die X-Achse hinweg ist die Zeit der Simulation von 0-60 Minuten, die Y-Achse zeigt die Anzahl der vorhandenen Schwärme. In der ersten Statistik behielten sie dabei einen freien Willen von 90°, bei der zweiten wurde ein freier Wille von 225° eingestellt.

Freier Wille: 90°

Bei einem freien Willen von 90° ist zu sehen, dass die Anzahl der Schwärme bei einem Drang zur Gruppierung von 0% konstant bleibt. Diese Entwicklung war bereits in Abschnitt 7.2 zu sehen. Die Start-Anzahl war dabei in allen Simulationen annähern gleich. Sobald der Drang zur Gruppierung über 0% steigt, zeigt sich, dass die Roboter sich abhängig vom eingestellten Wert unterschiedlich schnell zu Schwärmen zusammenziehen und diese auch nicht wieder verlassen. Ein höherer Wert sorgte dabei analog dafür, dass sich die Schwärme schneller sammelten. Dennoch arbeitet der Wert des Gruppendrangs prozentual. Die Berechnung findet auf der Spanne von [Schwarm-Mittelpunkt, vorige Ausrichtung] statt. Entsprechend hat der freie Wille noch immer einen großen Einfluss auf die Schwärme, wenn sie auch durch den Gruppendrang jetzt stabiler sind.

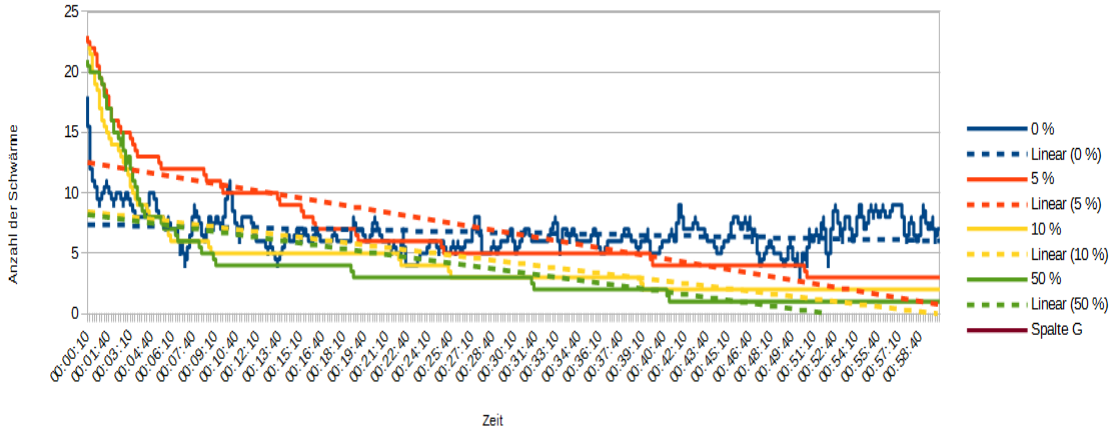

Abbildung 7.3: Entwicklung der Schwärme in Abhängigkeit ihres Dranges zur Gruppierung mit 90° eigenem Willen

Freier Wille: 225°

Bei einem freien Willen von 225° zeigt sich dagegen deutlich mehr Varianz. Die Entwicklung um 0% herum ist dabei wieder die gleiche wie in Abschnitt 7.2 bereits zu sehen war. Ein Wert von 5% neigt die Trendlinie zwar leicht nach unten, die Daten zeigen aber nach wie vor hohe Fluktuationen und Schwärme die sich zunächst gebildet haben, brechen aufgrund des großen freien Willens wieder auseinander und die Roboter trennen sich. Bei 10% wird der Trend zur Gruppierung zwar umso deutlicher, die Fluktuationen sind aber nach wie vor gut zu sehen und die Schwärme neigen noch immer dazu sich gelegentlich zu trennen. Erst ab einem Wert von 50% bleiben die Schwärme immer stabil und langsam aber sicher fügen sich alle Roboter zu einem einzelnen Schwarm zusammen.

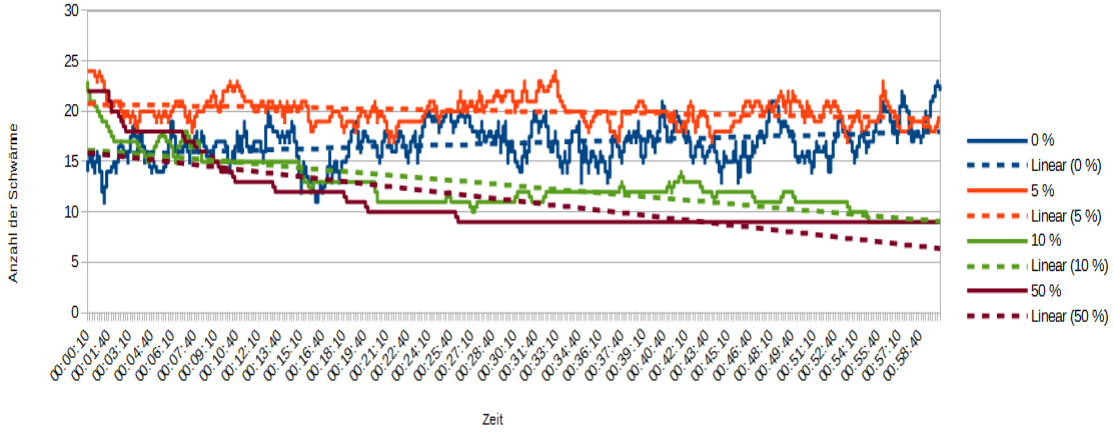

Abbildung 7.4: Entwicklung der Schwärme in Abhängigkeit ihres Dranges zur Gruppierung mit 225° eigenem Willen

Allgemein lässt sich also sagen, dass der freie Wille einen großen Einfluss darauf hat, wie schnell sich die Schwärme zusammenfinden, wohingegen der Gruppendrang seinen Teil dazu beiträgt, die Schwärme stabil zu halten. Dies kommt eben auch daher, dass der Gruppendrang auf dem freien Willen aufsetzt. Ein höherer freier Wille macht auch den Gruppendrang 'stärker'.

Statistik: Negativer Gruppendrang

Zum Schluss wurde noch eine Statistik mit negativen Gruppendrang erstellt. Hintergrund dieses Versuchs war es, die Roboter verteilen zu lassen. Damit sie sich während des Leerlaufs nicht versammeln und so in ihrer Masse zu einem größeren Hinderniss werden, wurde versucht die Roboter dazu zu bringen sich zu meiden und damit auszulösen, dass sie sich verteilen. Der Wert des freien Willens von 45° wurde absichtlich so gering gewählt, dass dieser normalerweise dafür sorgt, dass die Schwärme sich schnell bündeln. Außerdem baut der Gruppendrang auf dem freien Willen auf, ein kleinerer freier Wille sollte den Effekt des Gruppendrangs dementsprechend auch gering halten.

Tabelle 7.3: Eigenschaften des Schwarms

Größe des Schwarms:	25 Roboter
Lokale Reichweite:	5% der Größe der Simulation
Geschwindigkeit:	10% der lokalen Reichweite
Drang zur Gruppierung:	Variabel
Freier Wille:	45°

Über die X-Achse von Abbildung 7.5 hinweg ist die Zeit der Simulation von 0-60 Minuten, die Y-Achse zeigt die Anzahl der vorhandenen Schwärme. Wie zu sehen ist, versuchen sich die Roboter bei einem freien Willen von 45° und einem Gruppendrang von 0% zu Schwärmen zusammenzuschließen. Dieses Verhalten ist nachvollziehbar, da ein derart geringer freier Wille dazu führt, dass die Bewegung größenteils davon abhängig ist die anderen zu imitieren. Roboter im Radius ihrer lokalen Reichweite fahren also meist in die selbe Richtung und gruppieren sich so auf natürliche Weise. Wenn sie gegen die Grenzen der Simulation stoßen und versuchen dieser auszuweichen, kommen sie dabei noch näher zusammen.

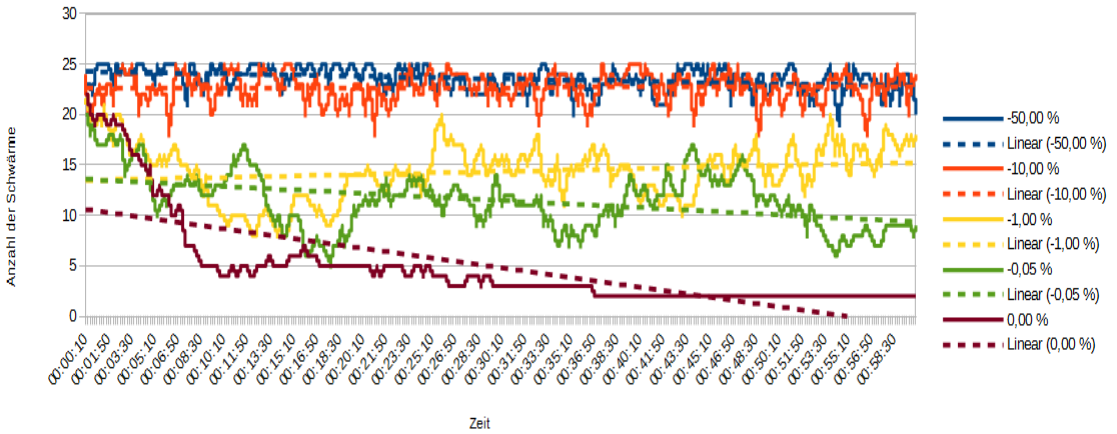

Abbildung 7.5: Entwicklung der Schwärme mit negativem Gruppendrang

Ein negativer Gruppendrang von bereits 0.05% reicht jedoch aus um die Trendlinie merklich nach oben zu verlagern und die Gruppierung in Schwärmen zumindest langsamer vollziehen zu lassen. Die Fluktuationen nehmen deutlich zu und die Schwärme scheinen nun deutlich instabiler zu sein. Zwar finden sich immer wieder Roboter in Schwärmen ein, sie brechen aber auch oft auseinander und die Einheiten gehen wieder getrennte Wege.

Auch hier lässt wieder deutlich sehen, dass der freie Wille der ausschlaggebende Wert ist, wenn es darauf ankommt die Roboter zusammenzuführen, wohingegen der der Gruppendrang dafür sorgt, dass die Schwärme in sich stabil bleiben und die Roboter nicht versuchen abzudriften.

Schon ab 1% negativen Gruppendrangs zeigt die Trendlinie, dass die Gruppen eher dazu neigen sich aufzulösen statt sich neu zu bilden. Der Winkel zur Mitte des eigenen Schwarms kann maximal 180° (in beide Richtungen) groß sein, das heißt der negative freie Wille kann einen Einfluss von maximal 1.8° einnehmen. Trotzdem reicht bereits dieser Einfluss aus um die Roboter dazu zu bewegen sich voneinander zu entfernen.

Ab einem Wert von 10% zeigt sich letztlich eine totale Streuung und die Trendlinie bleibt eine Konstante. Die Roboter schaffen es teilweise keinen einzigen Schwarm zu bilden (bzw. nur Schwärme in denen sie selbst als einziger Roboter vertreten sind).

Überraschenderweise zeigt ab einem Wert von 50% die Trendlinie wieder nach unten und die Roboter scheinen sich erneut zu gruppieren. Der Grund hierführ ist, dass die Roboter beim Ausweichen ihrer Kollegen überschießen und sich so weit in die andere Richtung drehen, dass sie letztlich vom Winkel wieder näher dran sind als vorher. Legt man Wert darauf, dass sich die Roboter verteilen, gilt es also einen passenden Prozentwert einzustellen und diesen nicht zu übertreiben, da man sonst das Gegenteil dessen erreicht, was man als Ziel hatte.

7.3 Anführer

Nachdem der Anführer implementiert wurde, wurde das konzipierte Verhalten getestet und verschiedene Statistiken erhoben, inwiefern ein Anführer in der Lage ist einen Schwarm an sein Ziel zu führen, ohne dass diese aktiv gesteuert werden müssen. Dafür wurde der gesamte Schwarm an einer Ecke der Simulation platziert. Die Einheiten waren dabei immer nahe genug beieinander, um als einzelner Schwarm angesehen zu werden, aber ansonsten zufällig verteilt. Einem der Roboter wurde daraufhin zufällig eine `New_Mission` zugeteilt und damit zum Anführer gewählt. Dieser hat daraufhin versucht seinen Schwarm zum Ziel zu führen.

Auch in dieser Analyse spielt der Zufall eine beachtliche Rolle, da der freie Wille der Einheiten darauf ausgelegt ist. Es wurden daher immer 15 Simulationen durchgeführt und aus den mittleren 9 Ergebnissen ein Mittelwert berechnet, um Aureißer möglichst zu ignorieren.

Nachbesserung der Implementierung

Recht schnell hat sich gezeigt, dass das Einfangen des Schwarms, wie es in Abschnitt 5.3 vorgestellt wurde, mehr negative als positive Auswirkungen hat. Versuchte der Roboter seinen Schwarm einzufangen, änderte er, dadurch dass er sich um 180° umdrehen musste, die gesamte Ausrichtung des Schwarms. Dies führte in den Simulationen meist dazu, dass der Schwarm vollkommen abgedriftet ist und der Anführer Mühe und Not hatte, ihn wieder auf das Ziel auszurichten. Meist verlor er dabei den Schwarm wieder, noch bevor dieser wieder ausgerichtet war, womit das Manöver wieder von vorne los ging. Durch dieses Fehlverhalten mussten die meisten Simulationen

Abbildung 7.6: Entfernungen des Anführers

abgebrochen werden und die Ergebnisse waren letztlich unbrauchbar, da zu viel von Hand gefiltert werden musste.

Aus diesem Grund wurde der Teil des Algorithmus' wieder entfernt. Stattdessen wartet der Anführer nun, wenn der Schwarm 50% der lokalen Reichweite entfernt ist und gibt auf, wenn der Schwarm-Mittelpunkt die eigene lokale Reichweite ganz verlassen hat. Aufgeben bedeutet in diesem Kontext, dass der Anführer ohne seinen Schwarm zum Ziel gefahren ist. Dies führte allgemein zu besseren Ergebnissen und der Schwarm konnte zumindest in Teilen zum Ziel gebracht werden.

Abhängigkeit: Freier Wille

Tabelle 7.4: Eigenschaften des Schwarms

Größe des Schwarms:	25 Roboter
Antahl Anführer:	1
Lokale Reichweite:	5% der Größe der Simulation
Geschwindigkeit:	10% der lokalen Reichweite
Drang zur Gruppierung:	5%
Freier Wille:	Variabel

In dieser Simulation wurde geprüft, wie sich verschiedene Werte für den freien Willen darauf auswirken, dass der Schwarm zum Ziel geführt werden kann. Über die X-Achse von Abbildung 7.7 hinweg ist die vergangene Wegstrecke in Prozent angegeben, die Y-Achse zeigt die Anzahl der Roboter die im Schwarm des Anführers vorhanden sind. In der Statistik zu sehen, ist dass der Schwarm anfangs immer zusammen blieb. Da die Roboter als gemeinsamer Schwarm zusammen gestartet sind, mussten sie sich nicht erst finden. Außerdem musste der Anführer erst einmal eine gewisse Zeit lang fahren, bevor der anfängliche Schwarm nicht mehr als sein Schwarm registriert wird.

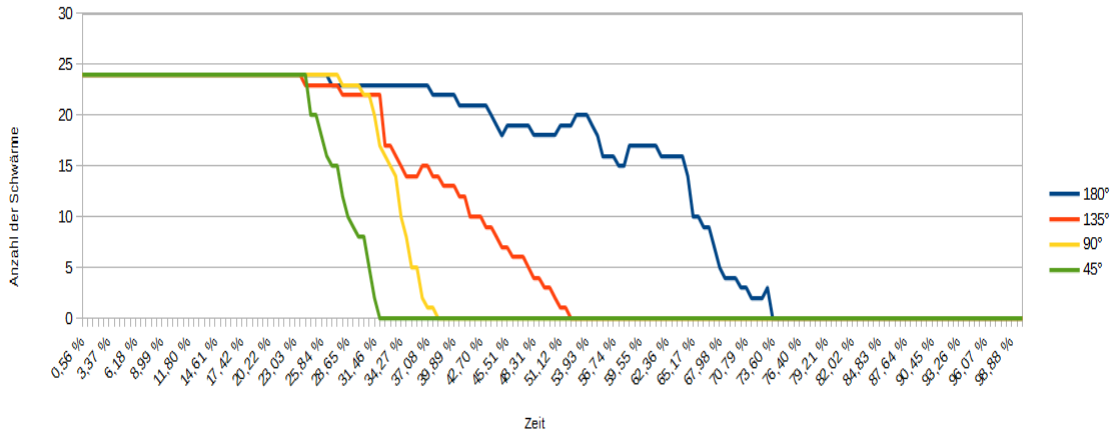

Abbildung 7.7: Erfolg eines Anführer ins Abhängigkeit des freien Willens

Wenn der Schwarm in eine andere Richtung lenkte als der Anführer, blieb der Anführer ab 50% seiner lokalen Reichweite stehen und wartete. Zog der Schwarm weiterhin in eine verkehrte Richtung, löste er sich irgendwann vom Anführer und dieser zog alleine weiter Richtung Ziel. Durch den relativ hohen Wert von 5% im Gruppendrang blieb der Schwarm immer gut zusammen. Bei geringeren werten des freien Willens zog er öfter

im gesamten davon, während die Roboter bei mehr freiem Willen immer mehr Stück für Stück weg zogen.

Je niedriger der Wert im freien Willen war, desto eher löste sich der Schwarm scheinbar von seinem Anführer. Dies ist darauf zurückzufiühren, dass bei einer Gruppengröße von 25 Robotern der Einfluss eines Anführers nur 4.2% ausmacht und der Schwarm mehr vom Zufall als vom Anführer gelenkt wird. Dass die Roboter mit dem größeren freien Willen letztlich scheinbar länger beim Anführer blieben, ist dem geschuldet, dass die Roboter weniger eng als Schwarm zusammen bleiben und durch die Verteilung der Roboter mehr in der Nähe des Anführers blieben. Die Roboter mit dem geringen freien Willen hingegen blieben mehr zusammen und sind somit als ganzes abhanden gekommen.

Abhängigkeit: Gruppengröße und -drang

Tabelle 7.5: Eigenschaften des Schwarms

Größe des Schwarms:	Variabel
Anzahl Anführer:	1
Lokale Reichweite:	5% der Größe der Simulation
Geschwindigkeit:	10% der lokalen Reichweite
Drang zur Gruppierung:	Variabel
Freier Wille:	90°

In den folgenden Statistiken wurde ein Anführer mit steigender Gruppenzahl und unterschiedlichem Gruppendrang gemessen. Über die X-Achse von Abbildung 7.7 hinweg ist der eingestellte Gruppendrang angegeben. Das erste Diagramm zeigt dabei, für wie viel Prozent der Wegstrecke der Anführer erfolgreich war. Erfolgreich war er, wenn der gesamte Schwarm bei ihm blieb. Ging auch nur ein Roboter abhanden, war der Anführer ab dort nicht mehr erfolgreich. Das zweite Diagramm zeigt die Größe des Schwarms um den Anführer herum beim erreichen des Ziels. Wenn der Anführer nicht alle Roboter beisammen hielt, ist es dennoch interessant zu wissen, wie viele Roboter es im Durchschnitt mit ihm zum Ziel geschafft haben. Das dritte Diagramm zeigt die Anzahl der Wartezeiten die der Anführer hatte. Eine höhere Anzahl an Wartezeiten ist direkt equivalent zu einer längeren Zeit die die Simulation gebraucht hat.

Dabei werden immer 2 Werte gezeigt. 'Mittelwert', der einen einfachen Mittelwert über alle 15 Messwerte anzeigt und 'Median', bei dem der Mittelwert auf die mittleren 11 Messwerte begrenzt wurde um Ausreißer auszusortieren. Da der Anführer selbst immer am Ziel ankommt, wurde er bei den Messwerten abgezogen. Die Anzahl der dargestellten Roboter entspricht also immer denjenigen, die passiv zum Ziel geführt wurden.

5 passive Roboter

In Abbildung 7.8 zu sehen ist zunächst einmal, dass der Anführer in einer Gruppe mit nur 5 Robotern einen sehr starken Einfluss auf die Gruppe hat. Bereits ohne Gruppendrang gibt es eine Erfolgsquote von über 20%. Ab 5% Gruppendrang gibt es schon annähernd 80% Erfolgsquote und ab 10% bekommt der Anführer seinen Schwarm immer vollständig ans Ziel gebracht. Median und Mittelwert sind meist relativ nahe zusammen, was bedeutet dass es sehr wenige Ausreißer gab, wenn überhaupt.

Die Größe des Schwarms zeigt ein sehr ähnliches Bild. Kommen bei 0% Gruppendrang im Schnitt nur sehr wenige Roboter an, steigert es sich sprunghaft auf 80% Einheiten und

ab einem Gruppendrang von 10% kommen immer alle Roboter an. Trotz einem Einfluss von 16.6% kann sich der Anführer also nicht durchsetzen, wenn es keinen Gruppendrang gibt.

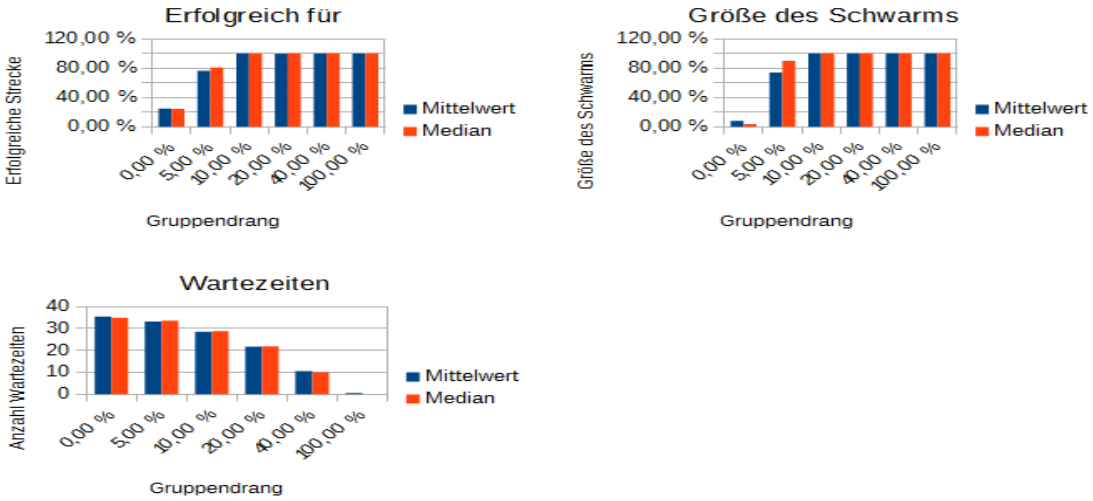

Abbildung 7.8: Erfolg eines Anführer ins Abhängigkeit der Gruppengröße und des Gruppendrangs mit 5 passiven Robotern

Die Wartezeiten sind nicht ganz so konstant, zeigen aber, dass es noch deutliche Unterschiede zwischen den Prozentwerten im Bereich von 10-100% gibt, auch wenn die beiden vorigen Diagramme dort keine mehr gezeigt haben. Mit einem höheren Gruppendrang sinkt die Anzahl der Wartezeiten immer mehr und bei 100% gibt es sogar annähernd gar keine mehr.

Bei 100% Gruppendrang zeigt sich generell überall das beste Ergebnis, was nicht weiter verwunderlich ist. Bei 100% richten sich die Roboter alle ausschließlich nach ihrer Mitte aus. Diese Konfiguration gab es nur in dieser ersten Statistik. Sie dürfte für die anderen nicht anders ausfallen und zeigen eher, dass das Konzept in dieser Hinsicht generell funktioniert. Eine Abweichung von einem perfekten Ergebnis (abgesehen bei den Wartezeiten) hätte höchstwahrscheinlich auf einen Fehler in der Implementierung hingewiesen.

10 passive Roboter

Die Diagramme in Abbildung 7.9 zeigen, dass der Einfluss des Anführers mit 5 Robotern mehr schon deutlich nachlässt. Konnten vorher schon ab 10% Gruppendrang beste Ergebnisse erziehlt werden, zeigt der Durchschnitt nun deutlich niedrigere Werte. Außerdem liegen der Mittelwert und der Median nicht mehr so eng beieinander. Dies bedeutet vor allem, dass die Fehlerquote zunahm und die 11 mittleren Ergebnisse nun ebenfalls höhere Schwankungen aufweisen. Nur bei 40% sieht man noch einen vollen Erfolg über alle Messwerte hinweg.

Die Größe des Schwarms ist von ähnlichen Rückschlägen betroffen. Bei 0% Gruppendrang schafft es nur jeder zehnte Roboter ins Ziel, fast der gleiche Wert wie bei 5 passiven Robotern. Bei 5% Gruppendrang ist die Erfolgsquote jedoch von 90% auf 50% gesunken (bezogen auf den Median). Bei 10% Gruppendrang schafft es der Median noch auf einen vollen Erfolg, der Mittelwert hingegen, der in Abbildung 7.8 noch bei 100% stand, ist nun auf 80% gesunken.

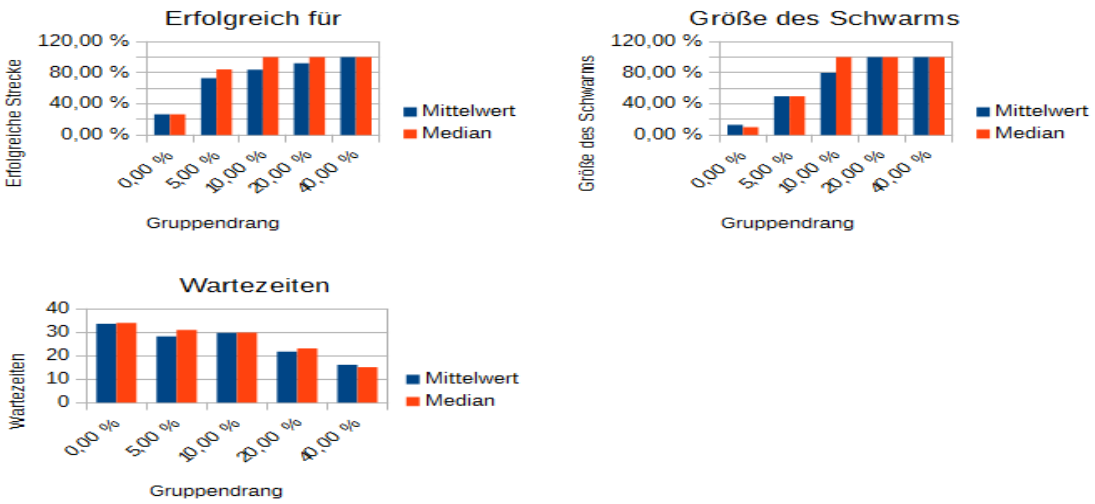

Abbildung 7.9: Erfolg eines Anführer ins Abhängigkeit der Gruppengröße und des Gruppendrangs mit 10 passiven Robotern

Die Wartezeiten hingegen zeigen nicht unbedingt den gleichen Trend, wenn auch hier Mittelwert und Median weiter auseinander sind als beim Vorgänger Abbildung 7.8. Sie sind zwar allgemein gestiegen, was zu erwarten war, wenn der Platz aufgrund weniger Roboter nun knapper wird. Der Trend ist aber nicht mehr so steil wie zuvor. Steigen beide Diagramme bei ca. 35 Wartezeiten ein, ist der Trend nun flacher und bei 40% Gruppendrang nun eine deutlich höhere Wartezeit zu sehen (+50%).

15 passive Roboter

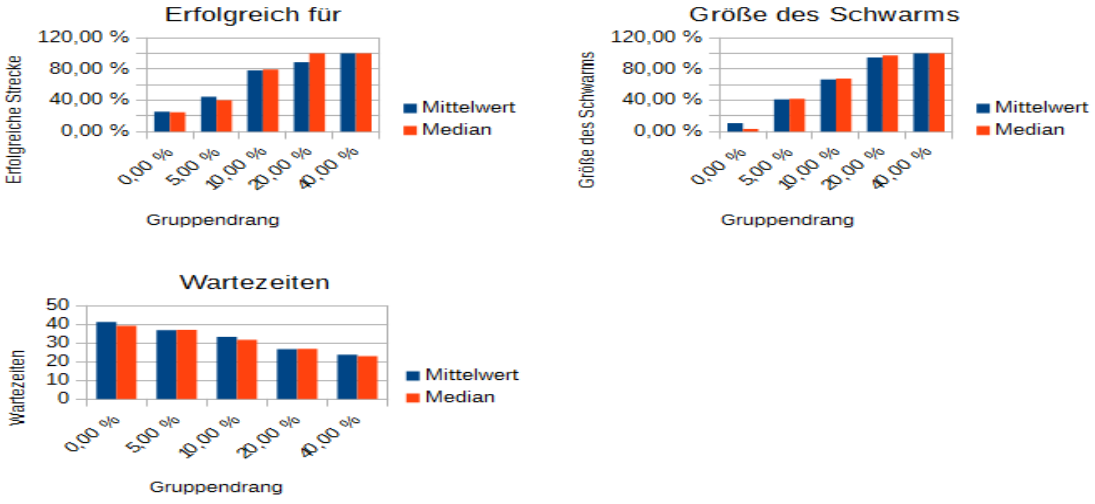

Abbildung 7.10: Erfolg eines Anführer ins Abhängigkeit der Gruppengröße und des Gruppendrangs mit 15 passiven Robotern

In Abbildung 7.10 nimmt der allgemeine Trend weiter seinen Lauf. Die Erfolgsquote für die Werte unter 40% nehmen immer weiter ab, wenn auch die 40% selbst noch einen vollen Erfolg vorweisen kann.

Es kommen auch weniger Roboter insgesamt im Ziel an, wenn die Gesamtzahl der

ankommenden Roboter auch bei 20% Gruppendrang noch recht hoch ist. Bei 40% Gruppendrang ist nach wie vor ein voller Ausschlag zu sehen.

Auch die Wartezeiten zeigen den selben Trend wie zuvor. Sie starten recht hoch, der Trend bleibt aber flacher als bei den Messungen mit 5 passiven Robotern weniger.

20 passive Roboter

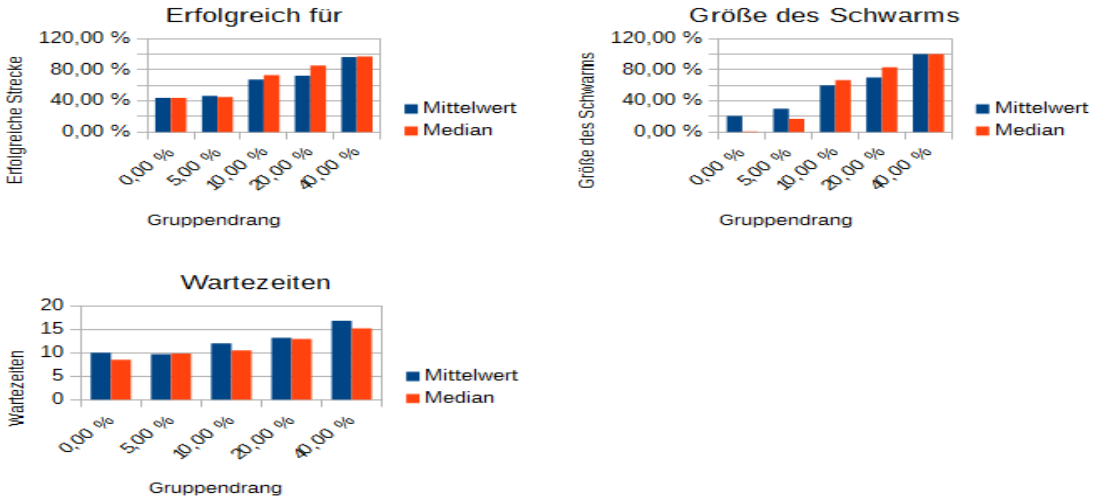

Abbildung 7.11: Erfolg eines Anführer ins Abhängigkeit der Gruppengröße und des Gruppendrangs mit 20 passiven Robotern

Ab einer passiven Anzahl von 20 Robotern zeigt sich in Abbildung 7.11 erstmals der Trend, dass die Wartezeiten zunehmen, wenn der Gruppendrang größer wird. Ein genauerer Blick auf das Diagram zeigt jedoch auch, dass die Wartezeiten allgemein sehr viel niedriger sind als in den Messungen mit weniger Robotern.

Dies ist vor allem darauf zurückzuführen, dass der Anführer ab einer Zahl von 20 passiven Robotern einen Schwarm viel schneller vollständig verliert und es keine Wartezeiten mehr gibt, weil er schneller dazu übergeht das Ziel alleine aufzusuchen. Auch die anderen beiden Diagramme zeigen nun bei 40% Gruppendrang keinen vollen Erfolg mehr.

25 passive Roboter

In den letzten Diagrammen (Abbildung 7.12) ist letztlich zu sehen dass sich der allgemeine Trend weiter fort führt. Die Wartezeiten sinken noch deutlicher, weil der Anführer seinen Schwarm immer früher verliert. Die erfolgreiche Strecke nimmt genau wie die Anzahl der Roboter die im Ziel ankommen immer mehr ab. Nur bei 40% Gruppendrang bleiben die Werte noch immer recht hoch.

Der Trend mit mehr Robotern zeigt letztlich, dass ein Anführer nicht in der Lage ist einen Schwarm im freien Feld vernünftig zu lenken. Selbst wenn man bedenkt, dass die Strecke recht kurz und gerade ist. Die immer höhere Zahl passiver Roboter lässt den Einfluss des Anführers mit der Zeit zu sehr schrumpfen, bis er am Ende fast nicht mehr wahrgenommen wird. Nur bei 40% Gruppendrang funktioniert das System noch, wahrscheinlich dadurch dass der Roboter trotz geringem Einfluss noch den Mittelpunkt des Schwarms verschiebt.

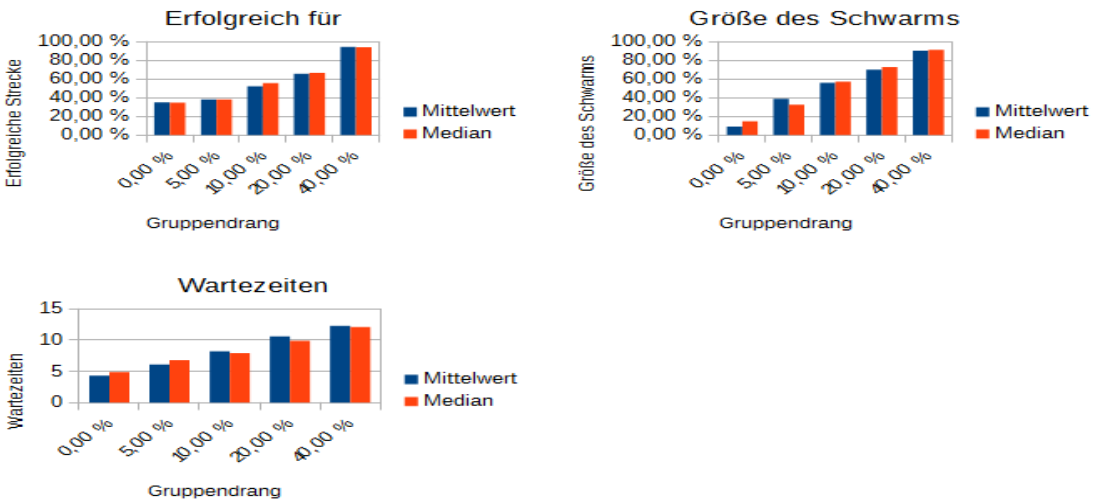

Abbildung 7.12: Erfolg eines Anführer ins Abhängigkeit der Gruppengröße und des Gruppendrangs mit 25 passiven Robotern

7.4 Transport von Waren mit Hilfe eines Schwarms

Nun da alles zusammengesetzt ist, gilt es letzte Messungen zu machen und zu schauen, wie sich die Roboter unter verschiedenen Bedingungen beim Transport machen.

Bei den folgenden Statistiken wurden jeweils 15 Simulationen durchgeführt und die mittleren 11 zu einem Durchschnitt verrechnet um Ausreißer zu eliminieren. Die Roboter wurden zufällig auf der gesamten Simulation erstellt und nach wenigen Sekunden der Auftrag gesendet. Sobald sie alle versammelt waren, wurde die Aufzeichnung gestartet und der Schwarm machte sich auf den Weg zum Ziel. Dabei wurden 4 verschiedene Eigenschaften beobachtet:

Anzahl der Ticks Dies ist die Anzahl der Ticks die während des Transports verstrichen sind. Da ein Tick genau 1.1s dauert, ist es damit auch ein direktes Equivalent für die Zeit die der Transport gedauert hat.

Roboter-Schlupf Von einem Tick zum anderen wurde gemessen, wie viel sich die einzelnen Roboter in Relation zur Mitte des Objektes bewegt haben. Dadurch wurde der Schlupf berechnet, den es während des Transports gab. Zu bedenken ist bei diesem Wert allerdings, dass die Simulation über keine Berechnung von Reibung verfügt, die Roboter also eine unendliche Kraft besitzen. Die Werte stellen demnach eine obere Grenze dar, für den Schlupf der in der Realität entstehen könnte.

Gefahrene Strecke Verdeutlicht den Weg, der von einem Roboter durchschnittlich zurückgelegt wurde. Die direkte Strecke vom Start zum Ende hat eine Länge von 5.656 (ROS-Einheiten). Da das Transportobjekt nur innerhalb der Reichweite von 0.05 am Ziel sein musste, ergibt sich somit eine Strecke von 5.606 Einheiten die gefahren werden musste.

Wartezeiten Anführer Letztlich wurde noch gemessen, wie oft es vor kam, dass ein Anführer sich nicht bewegen konnte, weil ein anderer Roboter ihm den Weg versperrte.

Abhängigkeit: Größe des Flocks und Anzahl Anführer

In den ersten Simulationen wurden verschiedene Kombinationen von passiven Robotern und Anführern getestet. Abbildung 7.13 zeigt die Entwicklung über verschiedene Schwärme hinweg. '5R, 1L' bedeutet dabei, 5 Roboter + 1 Anführer (Leader).

Unschwer zu sehen ist, dass der Transport mit 5 Robotern und 1 Anführer am längsten dauert, mit 66 Ticks. Überraschend jedoch ist, dass 10 Roboter und 2 Anführer schneller ist und 15 Roboter mit 3 Anführern wieder etwas schneller. Die Geschwindigkeit ist also nicht unbedingt nur davon abhängig, wie viel Prozent der Roboter Anführer sind. Generell ist der Transport schneller, je mehr Roboter man hat.

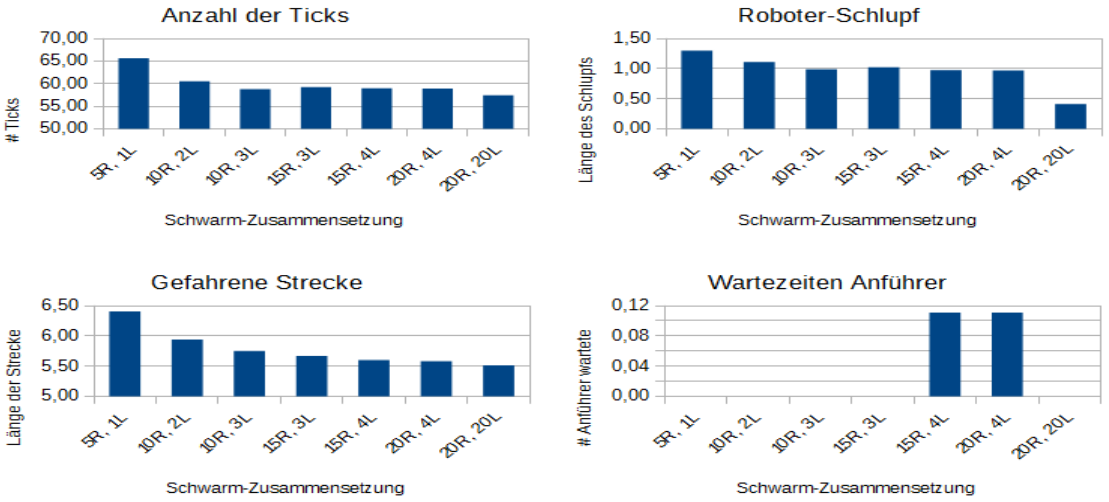

Abbildung 7.13: Abhängigkeit: Größes des Flocks und Anzahl der Anführer

Gestützt wird die Grafik von der Länge des Schlupfs, die ebenfalls abnimmt, wenn mehr Roboter den Auftrag erledigen, genau wie die Länge des Weges der pro Roboter gefahren wurde. Bei 20 Robotern und 4 Anführern, fuhren die Roboter (pro Roboter) 12.95% weniger Strecke, als bei 5 Robotern mit 1 Anführer. Im Schnitt hatten die Anführer keine Probleme sich zu bewegen. Nur in 2 Konfigurationen gab es überhaupt Ausschläge, diese bedeuten aber gerade mal einen Durchschnitt von 0.11 Stillständen.

Generell lässt sich also sagen, dass ein Auftrag tatsächlich schneller beendet werden kann, wenn er von mehr Robotern ausgeführt wird. Da er mit doppelt so vielen Robotern allerdings nicht doppelt so schnell ist, ist es letztlich wohl eine Abschätzung, ob die zusätzlichen Roboter besser einen eigenen Auftrag annehmen oder wirklich frei sind helfen können.

Der letzte Balken der Statistik zeigt den Transport mit allen Robotern als Anführer. Dies entspricht keinem Schwarm mehr, sondern streng gesteuerten Robotern und zeigt klar die Grenze, wie gut es generell werden kann. In jeder der 4 verschiedenen Eigenschaften stellen sie klar den besten Wert auf, wenn auch bei der Länge des Weges nur ganz knapp zum zweiten Platz.

Abhängigkeit: Freier Wille

In der zweiten Runde von Simulationen wurde der freie Wille verändert. Bei 0° haben sich die Roboter streng nach ihren Nachbarn ausgerichtet und der einzige Zufall waren ihre Ausrichtungen bei Beginn des Transports. Entsprechend stellt diese Konfiguration wieder die Grenze auf, wie gut der Schwarm werden kann.

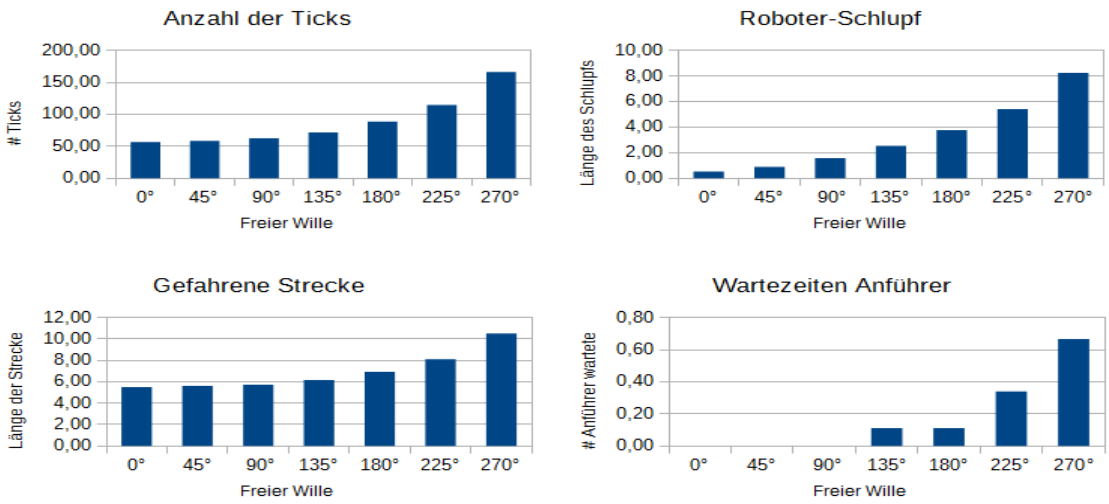

Abbildung 7.14: Abhängigkeit: Freier Wille

Generell ist in dieser Runde, wenig überraschend zu sehen, dass der Transport besser wird, je geringer der freie Wille ist. Der Slack steigt bei einem freien Willen von 270° auf über 1600% an. Ebenfalls haben die Anführer bis zu einem freien Willen von 90° keine Probleme sich zu bewegen. Erst ab da steigt der Balken deutlich an, ist aber immer noch recht gering, selbst bei 270°.

Abhängigkeit: Lokale Reichweite

Die letzte Runde der Simulationen zeigt den Transport mit verschiedenen Reichweiten. Das Objekt ist 2x2 groß. Bei einer Reichweite von 4 ist das Transportobjekt von einer Ecke zur anderen abgedeckt. Bei einer lokalen Reichweite von 0.3 kommen sie gerade so über die eigene Größe hinaus, die 0.1 beträgt.

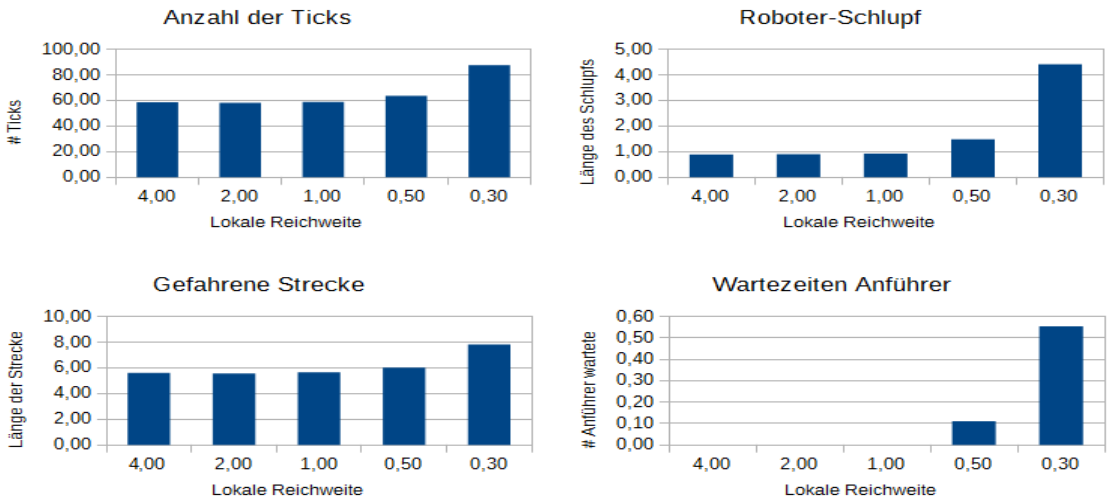

Abbildung 7.15: Abhängigkeit: Lokale Reichweite

Bis zu einer lokalen Reichweite von 1 scheint dies aber keinen Effekt zu haben. Erst ab 0.5 zeigt sich ein kleiner Unterschied zu seinen Vorgängern, obwohl hier nur noch ein viel kleinerer Teil der Roboter innerhalb des Transportobjektes mit einbezogen werden

konnte. Das bedeutet dass der Einfluss von Roboter über 1-2 Ecken hinweg nicht allzu viel abnimmt. Erst bei einer lokalen Reichweite von 0.3 wird der Fehlerbalken größer, ist aber immer noch recht gering, wenn man denkt, dass gerade einmal die eigene Größe Abstand zwischen den Robotern war.

8 Fazit und Ausblicke

Um die Thesis abzuschließen, werden in diesem Kapitel noch einmal die Ergebnisse der Thesis rekapituliert. Außerdem wird kurz beschrieben, welche Probleme es im Lauf der Thesis gab und wie diese gelöst wurden. Anschließend wird noch ein Ausblick gegeben, was noch hätte geschehen können, wenn die Zeit dieser Thesis kein Ende gesetzt hätte.

8.1 Fazit der Thesis

Die Frage die mit dieser Thesis beantwortet werden sollte war: Ist es möglich mit Hilfe eines Schwarms aus autonomen Robotern Transportaufträge zu erledigen? Die Antwort lautet: Ja, aber...

Die Ergebnisse die in Abschnitt 7.4 vorgestellt wurden, zeichnen das Bild, dass es geht, aber umständlich ist. Die 4 Regeln um einen Schwarm zu bilden, haben zwar den Vorteil, dass die einzelnen Roboter tatsächlich sehr performant arbeiten können, das Verhalten generell sehr übersichtlich ist und daher schnell und unkompliziert implementiert werden kann. Auf der anderen Seite ist die Steuerung des Scharms dafür umso komplizierter und allgemein sehr schwarmmig.

Freier Wille als größter Nachteil

Der freie Wille der Roboter sorgt innerhalb des Schwarms für ein möglichst natürliches Verhalten und dafür dass die Roboter als Schwarm dynamisch werden. Er sorgt auch dafür, dass der Schwarm verschiedene Verhaltensweisen zeigt und dass die Roboter in sich in Schwärmen zusammenfinden und gemeinsam herumfahren. Während dies aus Verhaltenstechnischer Sicher interessant ist, zeigt sich, dass beim Transport eines Objektes eben dieser freie Wille aber ein Nachteil ist.

Lenkung ist nicht präzise genug

Es ist sehr kompliziert einen Schwarm präzise zu lenken. In den Tests der Evaluation sollten die Roboter das Objekt auf einer freien Fläche in einer geraden Linie bewegen. dabei zeigte sich ganz eindeutig, dass der Transport besser verläuft, wenn die Roboter möglichst wenig freien Willen zur Verfügung hatten. Ebenfalls hatte eine höhere Anzahl von Anführern einen positiven Einfluss auf den Transport des Objektes. Ein Schwarm der nur aus Anführern bestand, und somit über keinerlei freien Willen verfügte, sondern nur noch aus gelenkten Robotern bestand, lieferte letztlich das beste Ergebnis.

Waren die Tests auf der geraden Strecke noch recht erfolgreich, dürften sich die Ergebnisse ändern, sobald es darum geht Kurven zu fahren. Es ist recht schwer einen Schwarm in eine Richtung zu beeinflussen. Auf der Geraden muss dies nur 1x geschehen. In Kurven muss dies mehrmals passieren. In früheren Tests wurde der Schwarm dazu veranlasst in Kurven zu fahren. Aufgrund geändertet Implementierung haben es diese Statistiken leider nicht in die Evaluation geschafft, dass sie nicht mehr vergleichbar waren. Dennoch zeigte sich in diesen Tests, dass bereits eine einfache Kurve den

Schwarm dazu bring sehr zu schlängeln. Eine 90°-Wende führte eher zu einem großen Bogen wodurch Schwarm der am Ende auch etwas über das Ziel hinausschoss und mit einer Spirale schließlich zum Ziel gelangte.

Schlupf oder blockierte Räder

Allerdings ist es nicht nur ein Problem den Schwarm als ganzes möglichst präzise in eine Richtung zu lenken, ein weiteres großes Problem ist der Schlupf der einzelnen Roboter. Die Roboter die für den Transport zuständig waren, blieben zwar unter dem Objekt, sie fuhren aber unterhalb des Objektes herum und verursachten dadurch Schlupf. Dadurch dass Reibung nicht implementiert wurde, betrug sie 0 und war bei allen Robotern gleich. In der Praxis hätte dieser Schlupf entweder dazu geführt, dass zwischen den Roboter und dem Transportobjekt Reibung entsteht oder sie hätte blockierte Räder zur Folge.

Reibung zwischen dem Transportobjekt und den Robotern hätte auf kurze Sicht eine mögliche Beschädigung des Transportobjektes zur Folge, auf lange Sicht eine beschädigung der Auflagefläche der Roboter. Außerdem darf nicht angenommen werden, dass der Schlupf in der Praxis bei allen Robotern gleich ist. Wurde in der Simulation angenommen, dass das Transportobjekt auf allen Robotern gleich rutscht und dadurch immer in der Mitte des Schwarms bleibt, dürfte sich diese Annahme in der Realität als falsch erweisen und sich das Transportobjekt bei zu viel herumgerutsche nicht mehr dort befinden, wo es von den Robotern angenommen wird. Die Folge wäre schlicht, dass das Transportobjekt von den Robotern rutscht oder an Kanten hängen bleibt, weil die Roboter es nicht präzise genug manövrieren können. Es müsste also eine andere Möglichkeit gefunden werden um die Position des Objektes zu identifizieren und bei den Robotern zu aktualisieren.

Ein permanentes blockieren oder durchdrehen der Räder der Roboter, würde diese auf lange Zeit ebenfalls beschädigen. Außerdem bestimmen die Roboter ihre Position vor allem dadurch, dass sie ihre Ausgangsposition wissen und diese mit der gefahrenen Strecke verrechnen. Es dürfte daher auf Lange sich schwer werden die Position der Roboter präzise zu halten, wenn die Räder durch permanentes blockieren oder durchdrehen nicht die Bewegung verursachen die von ihnen erwartet wird.

Außerhalb des Transports nützlich

Außerhalb des Transports war Schwarmverhalten dagegen durchaus sehr nützlich. Wie in Abschnitt 7.2 gezeigt werden konnte, lässt sich ein Schwarm mit den richtigen Parametern gut dazu bringen sich zu verteilen und anderen (Robotern oder Menschen) nicht im Weg zu stehen. Auch das Einfinden unterhalb des Transportobjektes lässt sich durch das Fluchtverhalten der Roboter sehr gut realisieren. Durch ihre Methode sich gegenseitig auszuweichen, wurde unterhalb des Objektes schnell Platz geschaffen und der Raum recht schnell und unkompliziert gefüllt. Ein Befehl der ihre Räder blockieren ließ, sorgte dann für den Stillstand den man braucht, um das Transportobjekt auf ihren Köpfen aufsetzen zu können.

Konklusion

Als abschließendes Fazit bin ich der Meinung, dass es sich durchaus lohnt Roboter mit Schwarmverhalten zu bauen und zu nutzen. Schwarmverhalten hat den großen Vorteil autonom denkender Roboter die sich dynamisch bewegen und dadurch Verhaltensmuster aufweisen, die mit einer zentralen Steuereinheit nur schwer

realisierbar wären. Im Kernpunkt dieser Thesis, dem Transport, ist Schwarmverhalten aber mehr Nachteil als Vorteil und sollte daher nicht dafür genutzt werden.

Beim Transport selbst ist es sinnvoller auf gelenkte Einheiten zu setzen oder Einheiten die sich im Team darüber absprechen, welcher Roboter sich wie bewegt. Es ist wichtig möglichst wenig Schlupf zu haben und die Roboter so fahren zu lassen, dass sie möglichst optimierte Fahrtwege haben.

Eine hybride Nutzung ist daher in meinen Augen die beste Option. Der Standby der Roboter sollte von Schwarmverhalten beherrscht werden. Auch das Einfinden unter dem Objekt geht schnell und effizient mit Schwarmverhalten. Sobald die Transportobjekt allerdings auf den Robotern abgesetzt wurde, gilt es auf gelenktes, berechnetes Verhalten zu setzen. Sobald das Objekt an seinem Ziel abgegeben wurde, ist wieder das Schwarmverhalten einzusetzen und die Roboter sich so auf natürliche Weise verteilen zu lassen.

8.2 Probleme mit ROS

Die Arbeit mit ROS brachte einige Probleme mit sich, die es zu bewältigen gab und die für Leser, die ROS nutzen wollen, wichtig zu wissen wären. Dieser Abschnitt beschreibt Probleme die zu Beginn der Thesis auftraten (März, 2018) und schildert die Probleme auf die ich gestoßen bin. Es ist möglich, dass einige dieser Probleme bereits mit erscheinen der Thesis behoben sind. Die folgenden Probleme sind kein expliziter Teil der Thesis, sondern lediglich als Erfahrungsbericht anzusehen, von dem andere profitieren können.

Bau von ROS

Möchte man mit ROS starten wird man gewillt sein die neueste Version von ROS zu nehmen. Diese ist derzeit `Melodic Morenia` und wird mit Ubuntu 18.04 empfohlen. Allerdings scheint diese Version von ROS noch nicht alle Packages auf der neuesten Version zu haben. Der Bau nach Anleitung geht, aufgrund fehlender Pakete schief.

Die Version davor ist 'Lunar Loggerhead', welche als 'primarily targeted at the Ubuntu 17.04 (Zesty)' beschrieben wird. Jedoch geht auch dieser Bau schief, wieder aufgrund fehlender Pakete. Nach einigem Probieren wurde mit `Ubuntu LTS 16.04 Xenial` eine Ubuntu-Version gefunden, bei der ROS ohne Probleme bauen konnte.

Bugs

Beim arbeiten mit ROS wurden einige Bugs erkannt, die nachfolgend etwas erläutert werden. Die Bugs wurden allerdings nicht behoben oder genauer untersucht. Da dies nicht Teil der Thesis ist, wurde nur versucht den schnellsten Workaround zu finden um mit der Arbeit fortfahren zu können.

Veraltete Position der Turtles

In der Praxis hat sich gezeigt, dass die Position der Turtles, die man von der Turtlesim am Ende eines Fahrbefehls zurück bekommt, nicht immer gestimmt hat. Teleportiert man sich mittels `teleport_absolute` an die Stelle, die man als Position zurück bekommt, wird die Turtle einige Einheiten nach hinten verschoben. Ohne dies genauer zu überprüfen, wirkt es, als würde der Fahrbefehl 1s ausgeführt werden, die Position wird allerdings bereits nach 900ms gespeichert und gesendet. Aus diesem Grund wird in der Thesis später kein

Gebrauch von `teleport_absolute` gemacht, sondern nur `teleport_relative` genutzt um sich zu drehen. Dadurch wird die Position im Raster nicht verändert.

Nicht angezeigte Turtles

Im Laufe der Thesis wurde sowohl mit meinem normalen PC, als auch mit einem Laptop an der Thesis gearbeitet. Beide mal wurde das selbe System genutzt und ROS auf die selbe Weise installiert. Trotzdem kam es bei der Arbeit zu einem Fehler, wenn der Laptop genutzt wurde. Beim Starten der Simulationen (Turtlesim mit vielen Turtles) kam es dazu, dass einige Turtles nicht in der Turtlesim angezeigt wurden. Sie wurden zwar erstellt, blitzten dann kurz als Symbol auf, aber verschwanden innerhalb eines bruchteils einer Sekunde wieder. Die Aufzeichnungen belegten aber, dass sie erstellt wurden, sich bewegten und mit den anderen Turtles interagiert haben. Die fehlende Darstellung wurde dabei nicht von einer Fehlermeldung oder ähnlichem begleitet. Dieser Fehler trat reproduzierbar auf, folgte aber keiner mit ersichtlichen Logik, bezogen darauf welche Turtles es traf. Es waren jedoch stets immer die selben Turtles, welche sich im Sourcecode nicht von den anderen unterschieden. **Ein Workaround konnte bis heute nicht gefunden werden.**

NaN als Position in der Turtlesim

Ebenfalls reproduzierbar und nicht nachvollziehbar war der Fehler, dass einige Roboter nach dem Start scheinbar 'NaN' als Position zurück gaben (die Position hat `float` als Datentyp). Dieser Fehler trat manchmal auf und blieb über die Dauer der Turtlesim bestehen. Ein Workaround hierfür war das Programm neu zu kompilieren.

Veraltete Sources

Ein anderer Bug betraf nicht ROS direkt, sondern sein Build-System catkin. Dieses kompiliert nur Source-Code neu, wenn er verändert wurde. Ein Feature welches (wahrscheinlich) dazu dienen soll die Kompilierzeit gering zu halten. Es stellte sich aber heraus, dass catkin dabei scheinbar nur `.c` und `.cpp` Dateien beachtet und Header-Dateien auslässt, wenn die Source-Dateien keine Veränderung aufweisen.

Der Code meiner Thesis wurde als Header-Only geschrieben. Das bedeutet, dass Klassen sich nicht in Header und Source aufteilen, sondern dass die gesamte Implementierung stets in den Header-Dateien vorzufinden ist. Source-Dateien sind nur die Dateien, welche die `main()` beinhalten. Wurden nur Änderungen an den Header-Dateien vorgenommen, übersprang catkin diese Dateien und die angepasste Simulation wurde mit veralteten Source-Dateien gebaut.

Ein einfacher Workaround dieses Problems wurde mit einem Skript geschaffen, welches rekursiv den gesamten Source-Ordner durchgeht und den Änderungs-Zeitstempel mittels den `touch` -Befehls[10] von Linux modifiziert. Dadurch wurden zwar immer alle Dateien neu kompiliert und der ganze Kompilier-Vorgang verlief wesentlich langsamer ab als er hätte sein müssen. Es war letztlich aber die schnellste Methode um den Fehler in catkin zu umgehen.

8.3 Ausblicke

Da die Thesis aufgrund der zeitlichen Begrenzung ein Ende finden musste, gab es viele Ideen die nicht umgesetzt werden konnten und die das System unvollständig

hinterlassen. Einige dieser Konzepte werden hier nun als Ausblick vorgestellt, die es in einer weiterführenden Arbeit anzugehen gilt.

Vergabe des Transportauftrags

Ein Punkt der bisher nicht in der Thesis Beachtung fand ist die Vergabe der Aufträge. Diese werden bisher im Auftrag festgelegt and bilden ein Spanne [a,b[. Die ersten n Roboter des Arrays bilden die Anführer.

Für eine Auftragsvergabe wäre es sinnvoller die Roboter nach ihrer örtlichen Nähe auszuwählen und anhand dessen, welche Aufträge noch anstehen (periodische Aufträge sind trivial vorherzusehen). Auch wenn die Roboter letztlich nicht mehr homogen aufgebaut sind, sondern verschiedene Einheiten verschiedene Werte aufweisen, wie zum Beispiel ihre Fläche oder Belastbarkeit. Eine Intelligenz die darüber entscheidet welche Roboter am besten für den Auftrag geeignet sind, wäre ein großer Gewinn für das System.

Wegfindung

Auch die Wegfindung wäre ein großes Kapitel für den Schwarm. In der Thesis wurden nur gerade Strecken beachtet die keinerlei Kurven besitzen. Müssen die Roboter über mehrere Eckpunkte hinweg, braucht es dafür eine neue Logik im System. Nicht nur wäre es ein Problem, weil es womöglich mehrere Eckunkte bräuchte, es wäre auch deswegen eine Herausforderung, weil ein Schwarm nicht eckig gelenkt werden kann. Ändert ein Schwarm seine Richtung, tut er dies sehr träge und macht dabei einen Bogen. Dieser müsste in der Routenplanung mit einberechnet, und vlt sogar genutzt, werden.

Lokalität von Informationen

Ein Aspekt der ebenfalls nicht in dieser Thesis beachtet wurde, ist der Effekt der lokalen Informationen. Es ist bekannt, dass in großen Schwärmen von Vögel Informationen nicht den ganzen Schwarm erreichen, sondern Informationen mit steigender Entfernung 'dünner' werden[7]. Die Informationen die in dieser Thesis genutzt wurden, wurden immer zu 100% an den gesamten Schwarm gesendet. Simulationen, welchen Einfluss eine abnehmende Informationsdichte hat wären für weitergehende Arbeiten interesssant.

Literatur

[1] R. Beckers, O. E. Holland und J. L. Deneubourg. „From local actions to global tasks: Stigmergy and collective robotics“. In: MIT Press, 1994, S. 181–189.

[2] R. Beckers, O.E. Holland und J.L. Deneubourg. „FROM LOCAL ACTIONS TO GLOBAL TASKS:STIGMERGY AND COLLECTIVE ROBOTICS“. In: (). URL: https://web.archive.org/web/20131104125931/http://www.eecs.harvard.edu/~rad/courses/cs266/papers/beckers-alife94.pdf.

[3] *Boids*. URL: http://www.red3d.com/cwr/boids/ (besucht am 25. 02. 2019).

[4] *Build a map with SLAM*. URL: http://wiki.ros.org/Build%20a%20map%20with%20SLAM (besucht am 25. 02. 2019).

[5] Hande Celikkanat, Ali Emre Turgut und Erol Sahin. „Guiding a Robot Flock via Informed Robots“. In: (). URL: http://sci-hub.tw/10.1007/978-3-642-00644-9_19.

[6] C. Delgado-Mata, J. I. Martinez, S. Bee, R. Ruiz-Rodarte und R. Aylett. „On the Use of Virtual Animals with Artificial Fear inVirtual Environments“. In: (2002). URL: https://sci-hub.tw/10.1007/s00354-007-0009-5.

[7] Esteban Fernández-Juricic und Victor Kowalski. „Where does a flock end from an information perspective? A comparative experiment with live and robotic birds“. In: *Behavioral Ecology* (). URL: https://academic.oup.com/beheco/article/22/6/1304/220324.

[8] Owen Holland und Chris Melhuish. „Stigmergy, Self-Organization, and Sorting in Collective Robotics“. In: (). URL: https://www.mitpressjournals.org/doi/10.1162/106454699568737.

[9] C. Ronald Kube und Hong Zhang. „Collective Robotics: From Social Insectsto Robots“. In: (). URL: http://sci-hub.tw/https://doi.org/10.1177/105971239300200204.

[10] *Linux touch*. URL: https://wiki.ubuntuusers.de/touch/ (besucht am 25. 02. 2019).

[11] MingC.Lin. „Efficient Collision Detection for Animation and Robotics“. Diss. UNIVERSITY of CALIFORNIA at BERKELEY, 1993.

[12] K. Ozogany und T. Vicsek. „Modeling the emergence of modular leadership hierarchyduring the collective motion of herds made of harems“. In: (). URL: https://arxiv.org/pdf/1403.0260.pdf.

[13] Jaime A. Pimentel, Maximino Aldana, Cristián Huepe und Hernán Larralde. „Programmable self-assembly in a thousand-robot swarm“. In: *Science* (). URL: http://science.sciencemag.org/content/345/6198/795/tab-article-info.

[14] Anies Hannawati Pumamadjaja und R. Andrew Russell. „Pheromone Communication: Implementation of Necrophoric Bee Behaviour in a Robot Swarm“. In: (). URL: http://sci-hub.tw/10.1109/RAMECH.2004.1437993.

[15] *ROS Catkin*. URL: wiki.ros.org/catkin (besucht am 25. 02. 2019).

[16] *ROS Master*. URL: http://wiki.ros.org/Master (besucht am 25. 02. 2019).

[17] *ROS Message*. URL: http://wiki.ros.org/msg (besucht am 25. 02. 2019).

[18] *ROS Nodes*. URL: http://wiki.ros.org/Nodes (besucht am 25. 02. 2019).

[19] *ROS RQT-Graph*. URL: http://wiki.ros.org/rqt_graph (besucht am 25. 02. 2019).

[20] *ROS Server*. URL: http://wiki.ros.org/Parameter%20Server (besucht am 25. 02. 2019).

[21] *ROS Sourcecode: NodeHandle*. URL: http://docs.ros.org/lunar/api/roscpp/html/classros_1_1NodeHandle.html (besucht am 25. 02. 2019).

[22] *ROS Topics*. URL: http://wiki.ros.org/Topics (besucht am 25. 02. 2019).

[23] *ROS Turtlesim*. URL: http://wiki.ros.org/turtlesim (besucht am 25. 02. 2019).

[24] Craig W. Reynolds. „Flocks, Herds, and Schools: A Distributed Behavioral Model“. In: (). URL: http://www.red3d.com/cwr/papers/1987/SIGGRAPH87.pdf.

[25] *Roslaunch*. URL: http://wiki.ros.org/roslaunch (besucht am 25. 02. 2019).

[26] Michael Rubenstein, Alejandro Cornejo und Radhika Nagpal. „Intrinsic and extrinsic noise effects on phase transitions of network modelswith applications to swarming systems“. In: *PHYSICAL REVIEW E77, 061138 2008* (). URL: https://sci-hub.tw/10.1103/PhysRevE.77.061138.

[27] Andrey V. Savkin. „Coordinated Collective Motion of Groups of AutonomousMobile Robots: Analysis of Vicsek's Model“. In: *Physical Review Letters* (). URL: https://sci-hub.tw/10.1109/TAC.2004.829621.

[28] Housheng Su, Xiaofan Wang, Senior Member und Zongli Lin. „Flocking of Multi-Agents With a Virtual Leader“. In: *IEEE TRANSACTIONS ON AUTOMATIC CONTROL, VOL. 54, NO. 2, FEBRUARY 2009* (). URL: http://sci-hub.tw/10.1109/TAC.2008.2010897.

[29] Peter Szabo, Mate Nagy und Tamas Vicsek. „Transitions in a self-propelled-particles model with coupling of accelerations“. In: *Physical Review* (). URL: https://journals.aps.org/pre/abstract/10.1103/PhysRevE.79.021908.

[30] Norbert Tarcai, CsabaVir, Daniel Abel, Mate Nagy, Peter L Varkonyi, Gabor Vasarhelyi und Tamas Vicsek. „Patterns, transitions and the role ofleaders in the collective dynamics of asimple robotic flock“. In: (). URL: https://hal.elte.hu/flocking/browser/trunk/public/references/vasarhelyi/Tarcai2011.pdf?format=raw.

[31] Tamas Vicsek und Anna Zafeiris. „Collective motion“. In: (). URL: https://arxiv.org/pdf/1010.5017.pdf.

[32] Tamas Vicsek, Andras Czirok, Eshel Ben-Jacob, Inon Cohen und Ofer Shoche. „Novel Type of Phase Transition in a System of Self-Driven Particles“. In: *Physical Review Letters* (). URL: https://sci-hub.tw/10.1103/PhysRevLett.75.1226.

[33] Justin Werfel, Yaneer Bar-Yam und Radhika Nagpal. „Building Patterned Structures with Robot Swarms“. In: (). URL: https://ssr.seas.harvard.edu/files/ssr/files/ijcai05-werfel.pdf.

[34] *boost::function*. URL: https://theboostcpplibraries.com/boost.function (besucht am 25. 02. 2019).

[35] *std::function*. URL: https://en.cppreference.com/w/cpp/utility/functional/function (besucht am 25. 02. 2019).

Abkürzungsverzeichnis

ROS	Robot Operating System
IPS	Indoor Positioning System

Anhang

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 1.0). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt