Abschlussbericht
Projektseminar Echtzeitsysteme

Team Apollo 13
Projektseminar eingereicht von
Li Zhao, Sebastian Ehmes, Huynh-Tan Truong, Nicolas Acero
am 7. April 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachgebiet Echtzeitsysteme

Elektrotechnik und
Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Gutachter: Géza Kulcsár
Betreuer: Géza Kulcsár

Erklärung zum Projektseminar

Hiermit versicheren wir, das vorliegende Projektseminar selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die wir aus fremden Quellen direkt oder indirekt übernommen haben, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Wir erklären uns damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 7. April 2016

(Li Zhao, Sebastian Ehmes, Huynh-Tan Truong, Nicolas Acero)

Inhaltsverzeichnis

1 Einführung

1.1 Aufgabenstellung

Im Projektseminar Echtzeitsysteme WS15/16 der TU-Darmstadt gestaltet sich die Aufgabenstellung sehr offen. Das heißt es gilt im wesentlichen ein Standardprogramm zu absolvieren, was folgende Punkte umfasst:

- Autonomes Fahren mittels Sensorik
- Erkennen von sogenannten “Aruco Markern“ mit Hilfe einer RGB-Kamera
- Eine Anwendung für diese Marker finden (z.B. durch markierte Tore fahren)

Außerdem wurde das Fahrzeug auf eine neue Plattform gestellt und als Softwarebasis soll ab jetzt das **R**obot **O**perating **S**ystem (ROS) genutzt werden. Darüber hinaus gibt es noch einige optionale Aufgabestellungen, wie etwa die Inbetriebnahme von Intertialsensorik, einer Art Fernsteuerung oder Implementierung einer ROS basierten Simulation.

1.2 Interpretation der Aufgabenstellung

Da der Begriff “autonomes Fahren“ Spielraum zur Interpretation lässt, hat sich das Team in Absprache mit dem Betreuer, neben den Pflichtaufgaben eigene Ziele gesetzt. Diese sollen im Nachfolgenden näher beleuchtet werden.

1.2.1 Motivation

Das autonome Fahren wurde in den vergangenen Projektseminaren immer unter vorher klar definierten Umweltbedingungen bearbeitet und gelöst. Das heißt, dass Lösungen immer für einzelne streng definierte Szenarien gefunden wurden und es lediglich, wenn überhaupt, primitive Kriteria für Übergänge dazwischen gab. Diese Herangehensweise ist gerade mit Hinblick auf eine mögliche zukünftige Teilnahme an einem Wettbewerb zu beschränkt und kaum ausbaufähig.

1.2.2 Neuinterpretation des autonomen Fahrens

Das Auto soll sich auf unbekanntem Terrain ohne zu strikte Rahmenbedingungen und ohne Kollision mit Objekten aus der Umwelt fortbewegen können. Zusätzlich soll es möglich sein zur Laufzeit vorgegebene Ziele im Raum eigenständig zu erreichen.

1.3 Herausforderungen

Um sich im Raum frei bewegen zu können muss das Fahrzeug zunächst wissen wo es sich befindet, das setzt zwei Dinge voraus. Erstens, muss das Auto in der Lage sein sich anhand von Stellgrößen bzw. Sensoren selbst zu lokalisieren. Zweitens muss es eine Karte der Umgebung haben um Wege zu Zielen planen zu können.

1.4 Neue finale Aufgabenstellung

Aus den vorherigen Erkenntnissen ergeben sich folgende neue Aufgaben die es zu lösen gilt.

Einarbeiten auf ROS und die neue Plattform:
Da sich sowohl die Hardware als auch die Software der Plattform seit dem letzten Projektseminar geändert haben, gilt es sich zuerst darin einzuarbeiten. Dafür bietet es sich an eine Version des autonomen Fahrens von vorherigen Generationen auf dem neuen System zu reimplementieren, sprich einen einfachen Abstandsregler zu einer Wand zu programmieren.

Fernsteuerung:
Um das Auto nicht über die Linux Konsole steuern zu müssen, soll eine Android App erstellt werden, die eine Art RC-Modus bietet.

Erkennen von Aruco-Markern:
Hier gilt es geeignete ROS Pakete zu finden, die dem Benutzer Position und Orientierung dieser Marker liefern, so dass man diese Information im weiteren Verlauf zur Anwendung bringen kann.

Aufrüstung der Sensorik:
Da sich Selbslokalisierung mit der Vorhandenen Sensorik nicht so realisieren lässt wie vorgestellt, wird das Auto um eine Kinect, einen Drehgeber und ein 6-Achsen Gyroskop erweitert.

Selbslokalisierung:
Anhand seiner Sensorik und Stellgrößen soll das Fahrzeug wissen wo es sich befindet.

Wegplanung:
Mit Hilfe der bereitgestellten Selbstlokalisierung muss nun ein geeignetes Wegeplanungsverfahren entwickelt werden, oder die Schnittstellen für ein fertiges Paket bereitgestellt werden.

Beispiel für die Anwendung der Wegplanung:
Das Durchfahren von Aruco markierten Toren soll als Anwendung einer funktionierenden Wegplanung verwendet werden. Hierbei soll kein Tor berührt werden und die Position sei unbekannt.

2 Hardware und Software des Autos

Seit dem letzten Projektseminar haben sich die Hardware und die Software stark geändert, daher ist es sinnvoll alle Komponenten die zur Verfügung stehen aufzuzählen.

2.1 Vorhandene Hard – und Software

Um den Abstand zu möglichen Hindernissen messen zu können, kommen wie in den Jahren zuvor Ultraschall Sensoren des Typs „SRF08“ [1] zum Einsatz. Wobei je ein Sensormodul an der Vorderseite, sowie an der linken und rechten Seite des Fahrzeuges angebracht ist.
Ein Hall-Sensor misst, mit Hilfe von am rechten Hinterrad angebrachten Magneten, die Umdrehungen des Rades. Indem die Impulse, die von den Magneten am Sensor hervorgerufen werden, gezählt werden, kann man daraus die gefahrene Strecke errechnen.
Unter der vorderen Stoßstange des Autos sind Lichtsensoren angebracht, die dazu verwendet werden können um Linien am Boden zu erkennen. Die Daten der Sensoren werden von einem 16Bit Microcontroller der Serie „MB96300“ von Fujitsu Microelectronics gesammelt, ausgewertet und dann an den darüber eingebauten PC via UART gesendet. Dieses Board interpretiert auch die Steuerbefehle an die Aktuatoren, die vom PC kommen und steuert dann die Servomotoren für den Lenkeinschlag und den DC-Motor für den Antrieb.
Der PC besteht im wesentlichen aus einem Mainboard, einem Intel I3 Quadcore CPU, 8GB DDR3 RAM, einer 60 GB SSD Festplatte (über 1xPCI-e angeschlossen) und einem WLAN-Modul. Das Mainboard ist über USB mit dem Fujitsu-Board verbunden und beide Komponenten beziehen ihre Energie aus einem 3200mAh starken Li-Fe Akku. Zuletzt ist noch eine herkömmliche RGB USB Kamera von Logitech an den PC angeschlossen. Diese bietet eine maximale Auflösung von 1280×720 Pixeln.
Als Echtzeitbetriebssystem kommt auf dem Fujitsu Board „FreeRTOS“ [2] zum Einsatz. Der PC nutzt das Robot Operating System – ROS [3] auf einem Linux Betriebssystem um nebenläufige Prozesse, in ROS „Nodes“ genannt, möglichst optimal versorgen zu können.
Diese Nodes im weiteren auch als Knoten bezeichnet, können miteinander über sogenannte „Topics“ kommunizieren, indem sie entweder abonnieren (subscriben) um Nachrichten von anderen Knoten zu erhalten, oder auf ihnen veröffentlichen (publishen) um Nachrichten zu senden. Es spricht auch nichts dagegen das ein Knoten den selben Topic gleichzeitig abonniert und darauf veröffentlicht, da mehrere Knoten gleichzeitig zugriff auf einen Topic haben. Wenn im folgenden davon die Rede ist, dass Nachrichten oder Daten zu bestimmten Knoten gesendet werden, so ist diese Abonnenten-/Veröffentlicher-Mechanik gemeint.

2.2 Ergänzungen im Verlauf des Projektseminars

Es hat sich im Laufe des Projektes herausgestellt, dass die Sensoren für die gesetzten Ziele nicht ausreichend waren. Daher wurde das Fahrzeug durch die im Folgenden erläuterten Sensoren erweitert.

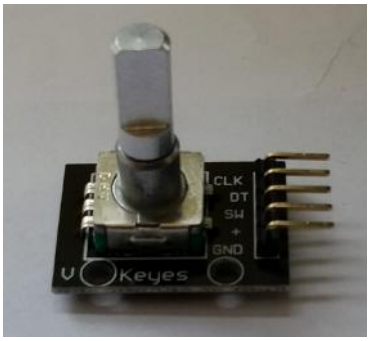

Abb. 2.1: *Drehgeber KY-040*

Um die gefahrene Strecke noch hochauflösender messen zu können, ist ein Drehgeber des Typs „KY-040“ [4] verbaut, der in der aktuellen Konfiguration 60 Impulse pro voller Umdrehung liefert. Im Vergleich dazu misst der Aufbau mit dem Hall-Sensor nur 8 Impulse pro voller Umdrehung. Für die Bestimmung der Orientierung in der Ebene wird ein 6-Achsen Gyroskop/Accelerometer vom Typ „GY-521“ [5] verwendet, das zu der Drehgeschwindigkeit um die 3 Raumachsen auch die Beschleunigung entlang dieser Achsen messen kann.

Die beiden Sensoren werden von einem MEGA2560 R3 Mikrocontroller-Board der Firma Funduino (Arduino Fremdhersteller) ausgewertet und die Daten über USB mit UART an den PC gesendet. Zusätzlich ist noch eine Microsoft Kinect [7] an die Stelle der RGB USB Kamera getreten, um Tiefeninformation des Raumes zu erhalten. Diese ist mit einer Halterung aus dem 3D Drucker an die Befestigung der alten Kamera angebracht. Die Kinect ist ebenfalls über USB angeschlossen und wird von einem zweiten Li-Fe Akku versorgt.

Abb. 2.2: *Gyro./Acc. GY-521*

2.3 Übersicht der verbauten Komponenten

1. Ultraschall Sensoren
2. Hall-Sensor
3. Helligkeitssensoren
4. Fujitsu Board
5. Mainboard
6. Drehgeber
7. Gyrosensor
8. Arduino
9. Kinect

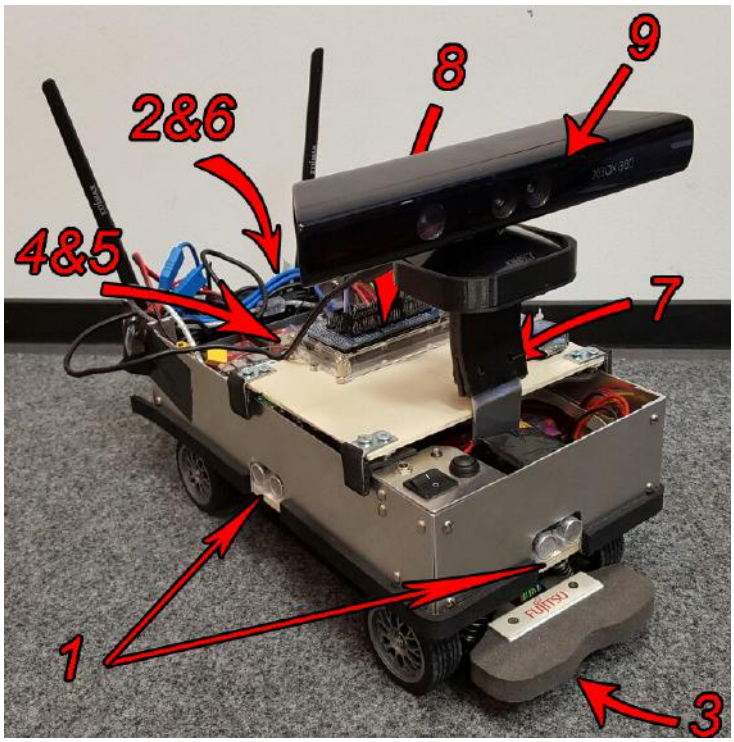

Abb. 2.3: *Aufgerüstetes Projektseminarauto*

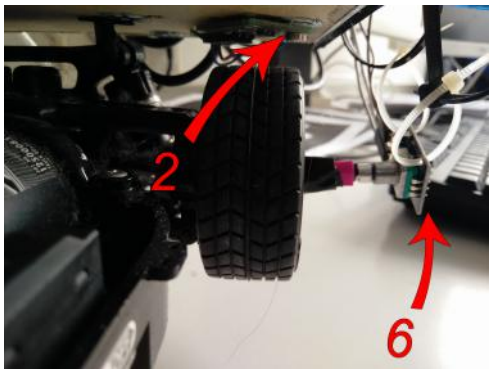

Abb. 2.4: *Drehgeber(6) und Hall-Sensor(2)*

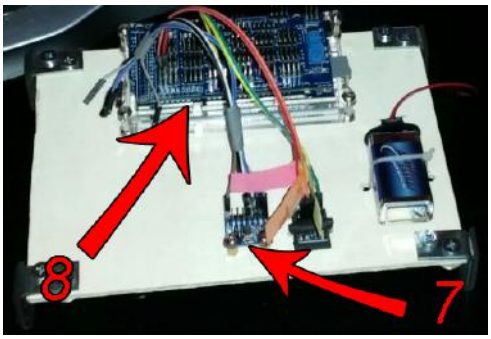

Abb. 2.5: *Gyrosensor(7) und Arduino(8) mit Sensor Shield*

Anmerkungen:

- Punkt 4. und 5. befinden sich unter der Holzplatte
- Punkt 7. ist hinter der Kinect-Halterung
- Punkt 2. und 6. befinden sich am hinteren linken Rad

3 Mission I

Die Mission 1 orientiert sich an einem Lösungsansatz aus den vorherigen Projektseminaren. Das autonome Fahren wurde darin über einen PID-Regler realisiert. Zusätzlich gehört zur Mission 1 eine Fernsteuerung und das Durchfahren der Aruco-Tore mit Hilfe des ROS-Packages ar-sys. Der Grundgedanke dieses Aufgabenteils ist es, sich durch einfache schnell implementierbare Funktionen, mit ROS und dem Verhalten des Fahrzeuges vertraut zu machen. Außerdem soll während der Bearbeitung klar werden wie es im Weiteren vorangehen soll.

3.1 Implementieren einer Abstandsregelung

Die Grundidee zur Realisierung eines einfachen autonomen Fahrens ist, dass sich das Auto an einer Wand orientiert und mit einem bestimmten Abstand zu ihr fortbewegt. Gelöst wurde dieses Problem durch einen PID-Regler der die Ist-Größe, in diesem Fall der Abstand zur Wand, so korrigieren soll, dass sich das Auto auf der gewünschten Fahrbahn bewegt. Aus dem Namen erkennbar besteht der Regler aus drei Gliedern: Einem Proportionalglied P, das die Abweichung vom Ist-Wert mit dem Soll-Wert vergleicht, einem Integralglied I, der sich durch die Summe der Abweichung aus Soll- und Ist-Wert auf die Stellgröße auswirkt, einem Differentialglied D, der auf die Änderungsgeschwindigkeit reagiert. Die allgemeine Formel eines PID-Reglers sieht wie folgt aus:

$$y(t) = p * e(t) + i \int_{t_0}^{t_i} e(t)\mathrm{d}x + d * (\frac{\partial e(t)}{\partial t})$$

Wobei y(t) die Stellgröße, e(t) die Differenz zwischen Soll- und Ist-Wert darstellt und p, i, d, die einzelnen Glieder gewichten. Ziel ist es die p,i,d – Werte so zu ermitteln, dass sich das Fahrzeug schnell mit dem Soll-Abstand zur Wand auf einer möglichst geraden Fahrbahn fortbewegt. Durch einige Versuche in der realen Welt wurden die p, i, d -Werte empirisch ermittelt. Dabei ist aufgefallen, dass die P -, D-Glieder das größte Gewicht haben, das I-Glied hat so gut wie keine Auswirkung auf das makroskopische Verhalten des Fahrzeugs und entfällt somit komplett.

3.2 Fernsteuerung TouchOSC

Damit das Fahrzeug nicht immer umständlich über die Linux Konsole gesteuert werden muss und um zur Laufzeit schnell reagieren zu können, falls ein Schaden durch Softwarefehlfunktion droht, bietet es sich an eine Fernsteuerung für ein Smartphone zu entwickeln. In diesem Fall wurde die Fernsteuerung mit Hilfe des ROS-Packages „rosOSC“ und der Handyapp "TouchOSC“ der Firma Hexler entwickelt, die einige praktische Funktionalitäten mit sich bringt.

Unter anderem kann man mit TouchOSC schnell eine Android App zur Fernsteuerung entwickeln ohne sich vorher eingehend mit der Entwicklung von Android Apps beschäftigt zu haben. So lässt sich das Interface der Fernsteuerung graphisch per Drag-and-Drop zusammen bauen und Daten können einfach über das WLAN versand und empfangen werden. Da die Datentypen der App nicht mit ROS kompatibel sind und über das Netzwerkprotokoll Open Sound Control (OSC) verschickt werden, ist eine Schnittstelle einzusetzen, die die Nachrichten in ROS-Topics übersetzt und veröffentlicht. Diese Rolle übernimmt das Paket rosOSC. Somit ist es kein Aufwand die „Topics“ auf denen die Nachrichten „gepublished“ werden zu „subscriben“ und so für die weitere Verwendung im Code zur Verfügung zu stellen. Oben ist der erste Tab der App zu sehen, auf dem der typische RC-Modus umgesetzt ist. Links stellt man die Lenkung ein, rechts die Geschwindigkeit und Richtung und in der Mitte gibt es den Notfallknopf um das Auto sofort anzuhalten.

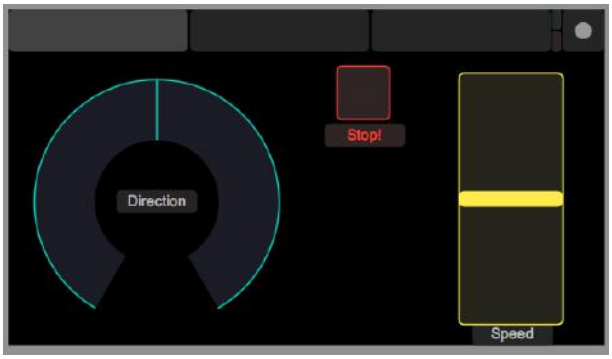

Abb. 3.1: *Tab 1*

Abb. 3.2: *Tab 2*

Links sieht man den zweiten Tab, in dem verschiedene autonome Fahrmodi ausgewählt werden können. „Follow Aruco markers“ aktiviert die Wegplanung und das Fahren durch Tore. „Autonomous mode“ lässt die Steuerung des Autos durch das ROS Navigation Stack zu.

Abb. 3.3: *Tab 3*

Tab 3 der App hat keine Funktion, sondern stellt nur Kurz das Team vor.

Die späteren Funktionen des Autos sind darauf ausgelegt mit der Fernbedienung genutzt zu werden. Möchte man das ohne tun, so bleibt nur die Möglichkeit die Nachrichten der Fernbedienung manuell über die Linux-Konsole zu publishen.

3.3 Aruco-Marker erkennen mit ar_sys

Das ROS-Package "ar_sys" ermöglicht das Erkennen von einzelnen oder mehreren Aruco-Markern gleichzeitig. Die wohl wichtigste Funktion von "ar_sys" ist die Möglichkeit, nach sorgfältiger Kalibrierung der Kamera, sowohl Orientierung als auch Anordnung der Marker im dreidimensionalen Raum zu erkennen.

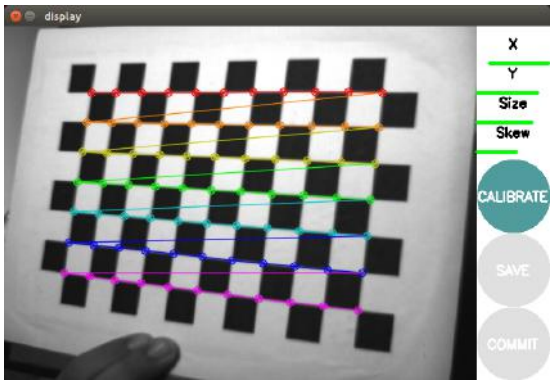

Abb. 3.4: *Kalibrierung der Kamera [19]*

Die Kalibrierung erfolgt mit Hilfe des Paktes "camera_calibration" und eines Schachbrettmusters, bei dem die Maße der schwarzen und weißen Quadrate exakt bekannt sind. Damit kann die optische Verzerrung, die durch die Linse entsteht, herausgerechnet werden. Im Bild rechts kann man gut erkennen wie Marker mit einzigartiger ID erkannt werden und die räumliche Orientierung dargestellt wird. Die Z-Achse ist im Paket immer als die Flächennormale für die Arcuo-Marker definiert. Hat der Algorithmus einen gültigen Marker erkannt, so wird ein neues Datenpaket veröffentlicht, welches die ID des Markers, Raumkoordinaten, Ausdehnung und Orientierung enthält.

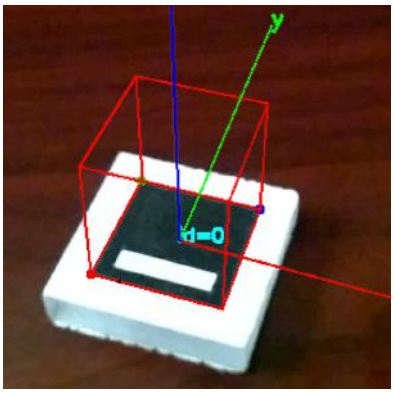

Abb. 3.5: *Erkannter Aruco Marker [20]*

4 Mission II

Durch den Ansatz der Regelung aus Mission 1 fährt das Auto nur unter sehr speziellen Rahmenbedingungen autonom. Zusätzlich gibt es nur begrenzt Ausbaumöglichkeiten, da neue Aufgaben wohldefinierte Bedingungen benötigen. Mission 2 verfolgt daher einen Ansatz aus der Robotik: Wegplanung mit Hilfe von Odometrie, der Vorwärtskinematik und entsprechender Sensorik. Dadurch entfallen besondere Rahmenbedingungen und das Auto fährt wirklich autonom.

4.1 Sensoren & Arduino

Die bereits vorhandene Austattung der Sensoren ist für die weiteren Ziele wie sie in 1.4 gesetzt wurden nicht ausreichend. Im Folgenden soll aufgezeigt werden wieso und wie man die Situation verbessern kann.

4.1.1 Probleme mit den alten Sensoren

Zum einen liefert der Hall-Sensor am Hinterrad, der zur Bestimmung der zurückgelegten Strecke gedacht ist, zu selten brauchbare Werte (<1Hz). Diese niedrige Rate entsteht, weil sehr viele offensichtlich falsche Werte gesendet werden, die dann durch eine Filterfunktion verworfen werden. Das wird wahrscheinlich durch eine Intersymbolinteferenz bei der UART Übertragung vom Microcontroller zum PC verursacht. Das dieses Verhalten durch einen Fehler in der Auslesefunktion im Code auf dem Fujitsu-Board ausgelöst wird, kann man so gut wie ausschließen, da es sich hierbei um den Code eines der früheren Teams handelt, das scheinbar keine Probleme mit der Streckenvermessung hatte. Der Verdacht liegt nahe, dass die Baudrate mit der die serielle Schnittstelle überträgt wohl nicht korrekt eingestellt ist. Desweiteren ist die Bestimmung der Orientierung anhand der Lenkeinstellungen sehr Fehleranfällig, da nicht gemessen werden kann was aktuell eingestellt ist, sondern man darauf vertrauen muss, dass der Stellwert dem gewollten Lenkwinkel entspricht. Genaueres zu diesem Problem unter Punkt 4.2. Zuletzt besteht das Problem, dass man seine Umgebung kennen muss, also eine Karte der Umwelt braucht, um in dieser Wege planen zu können. Dafür benötigt man entweder eine sehr gute Karte, sprich einen Grundriss des Gebäudes oder aber man benutzt einen Sensor mit dem man die Welt ausreichend genau vermessen kann.

4.1.2 Wegmessung mit einem Drehgeber

Der Hall-Sensor wird bei diesem Auto durch einen angeflanschten Drehgeber am Hinterrad ersetzt. Diese Sensor kann intern optisch eine vollständige Umdrehung in bis zu 60 diskrete Messpunkte auflösen, wohingegen der frühere Aufbau nur maximal 8 Punkte unterscheiden konnte. Damit reduziert sich die kleinste messbare Wegstrecke von $s_{min,vorher} = \frac{r_{Rad}2\pi}{n_{PulseproUmdrehung}} = \frac{0{,}032m\,2\pi}{8} \approx 2,5cm$ auf $s_{min,nachher} = \frac{0{,}032m2\pi}{60} \approx 3,4mm$. Das Problem der schlechten Übertragung wird durch einen Arduino Microcontroller umgangen, an dem der Drehgeber angeschlossen ist. Die Messwerte werden auf dem Arduino mit einer kostenlosen Bibliothek [8] für Drehgeber verarbeitet, die auch aufgrund der Eigenschaften des Drehgebermoduls Vorwärts - und Rückwärtsbewegungen unterscheiden kann. Zusätzlich wird sichergestellt, dass der mit dem Arduino kommunizierende ROS Node alle 10ms mit neuen Messwerten versorgt wird. Somit wird eine höhere Genauigkeit, Richtungsauflösung und eine zuverlässige Streckenvermessung erzielt.

4.1.3 Messung der Orientierung in der X-Y-Ebene

Um die Orientierung zu messen wurden zwei verschiedene Methoden ausprobiert. Zum einen ein Magnetometer, der mit Hilfe des Erdmagnetfeldes die absolute Orientierung im Raum misst und als Roll-, Pitch-, Yaw – Winkel ausgibt. Durch Recherche hat sich allerdings ergeben, dass diese Art der Winkelbestimmung nur für Anwendungen zu empfehlen ist die außerhalb eines Gebäudes stattfinden. Eisenhaltige Metalllegierungen in den Wänden, sowie stromdurchflossene Leiter und andere EM-Felder beeinflussen die Messung der Flussdichte des Erdmagnetfeldes negativ und können so zu Ungenauigkeiten führen. Alternativ kann man die Ausrichtung im Raum mit Hilfe eines 6-Achsigen Gyrosensors bestimmen, indem die Winkelgeschwindigkeit um die Z-Achse über die Zeit integriert wird. So erhält man in der Theorie $\Theta(t) = \Theta_0 + \int_t \omega_z(\tau)\,d\tau$ einen zeitlich abhängigen Winkel. Aufgrund der Nachteile des Magnetometers in den geplanten Anwendungsszenarien und der Tatsache, dass ein Gyro - / Accelerometermodul zur Verfügung steht, ist mit Methode zwei weitergearbeitet worden. Der Gyrosensor ist über dem Achsmittelpunkt, in zukünftigen Rechnungen auch der Koordinatenursprung des Fahrzeugs, angebracht und ebenfalls an den Arduino angeschlossen. Das Modul ist so orientiert, dass eine Richtungsänderung des Autos in der X-Y-Ebene eine Messung der Geschwindigkeitsänderung um die Z-Achse des Gyromoduls provoziert. Mit Hilfe einer Arduino OpenSource Bibliothek [9] für diesen Typ Sensor „MPU6050“, kann man die Register des Gyromoduls auslesen und von Quaternionen in Roll-, Pitch-, Yaw-Winkel umrechen und an den PC senden.

Data		Data		Data		Data		Data		Break
Distance s=[m]	*	Angular velocity $\omega = \left[\frac{rad}{s}\right]$	*	Angle Yaw=[deg]	*	Time stamp t=[s]	*	Check sum	*	\n

Die Tabelle zeigt das selbst entworfene Arduino Übertragungsprotokoll, mit dem die Kommunikation zwischen Sensoren und PC realisiert wird. Die Prüfsumme am Ende ergibt sich durch das Aufaddieren aller Daten eines Frames und soll verhindern, dass Übertragungsfehler die odometrischen Berechnungen entwerten. Mit Hilfe einer OpenSource Bibliothek [11] „cArduino.h“ kann ein Arduino Objekt erstellt werden, mit dem man die Daten der seriellen Schnittstelle auffangen und dann weiterverarbeiten kann.

4.1.4 Räumliche Wahrnehmung

Um Tiefeninformation der Umgebung zu erhalten nimmt man am besten einen Planarlaser, der punktuell die Distanz zwischen Hinternis und Sensor in der Ebene bestimmen kann. Diese Sensoren haben im Allgemeinen eine große Reichweite und gute Genauigkeit, aber sie sind sehr teuer und zu schwer für das Fahrzeug in der aktuellen Konfiguration. Deshalb und wegen der unmittelbaren Verfügbarkeit ist auf dem Roboter eine Microsoft Kinect Kamera verbaut. Diese liefert eine Punktwolke des Raums mit Tiefeninformation, aus der man später eine Karte der Umgebung erstellen und sich zusätzlich mit Hilfe optischer Odometrie im Raum wiederfinden kann. Mit Hilfe des ROS Paketes "freenect_stack"[10] können Topics mit Informationen und Daten der Kinect Kamera erstellt und veröffentlicht werden, insbesondere das Topic auf dem die errechneten Werte der Punktwolke veröffentlicht und dann später von anderen Paketen weiterverarbeitet wird.

4.2 Odometrie

Damit das Auto sich selbst im Raum einordnen kann, muss es in der Lage sein anhand von Sensoren und oder Steuergrößen die durchgeführten Bewegungen nachzuvollziehen. In der klassischen Auslegung berechnet die Odometrie nur mit Hilfe der Steuergrößen, also Lenkeinschlag und zurückgelegter Strecke, die Positions – und Orientierungsänderung des Fahrzeuges im Raum. Im Falle eines Autos handelt es sich bei dem Lenktyp um eine Ackermann-Lenkung, eine nicht-holonome (beschränkte Bewegung) Fahrzeugart. Das heißt, dass der Roboter sich nicht auf der Stelle drehen oder seitwärts bewegen kann, sondern nur auf kreisförmigen Bahnen um ein momentanes Rotationszentrum, oder auf Tangenten dieser Bahnen fährt.

4.2.1 Vorwärtskinematik Modell der Ackermann Lenkung

Im Folgenden wird mittels einer ausführlichen Herleitung gezeigt wie man die Position und Orientierung dieses Fahrzeugtyps in der X-Y-Ebene, aus dem Lenkeinschlag α und der zurückgelegten Strecke S, berechnen kann.

Dabei gilt:

α: Lenkeinschlag bezogen auf die Radnormale

ϕ: gefahrenes Kreissegment um ICC

ICC: Instantaneous center of curvature - momentanes Rotationszentrum

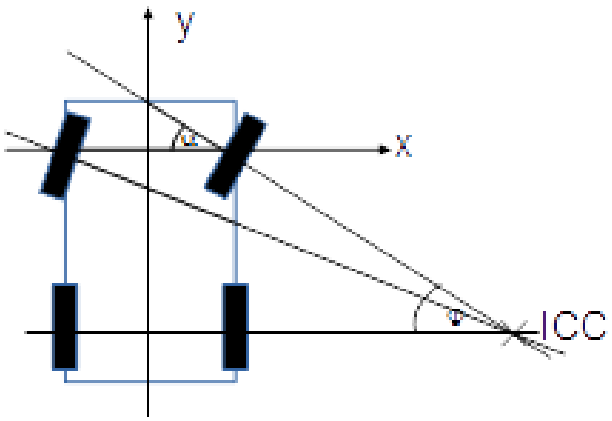

Abb. 4.1: *Skizze zur geometrischen Überlegung*

Geradengleichung: $f(x) = (x - x_0)m + t_0 \quad mit: \; t_0 = -k$

Wobei k der Betrag des Abstandes der beiden Achsen ist, setzt man den Koordinatenursprung in die Vorderachse ergibt sich als Verschiebung auf der Y-Achse -k.

Es folgt: $x = \frac{-k}{\tan(\alpha)}$ und $y = -k \quad \Rightarrow \quad I\vec{C}C = -\begin{bmatrix} \frac{k}{\tan(\alpha)} \\ k \end{bmatrix}$

Wobei x und y die Koordinaten des ICC aus sich des neuen Koordinatenursprungs sind.

Also gilt für den Radius der Kreisbahn: $R = |I\vec{C}C| = \sqrt{\left(\frac{k}{\tan(\alpha)}\right)^2 + k^2}$

Damit gilt für das Kreissegment: $\Theta = \frac{S}{R} = \frac{S}{\sqrt{\left(\frac{k}{\tan(\alpha)}\right)^2 + k^2}}$

Wobei S die bereits gefahrene Strecke auf der Kreisbahn ist. Es stellt sich nach Überlegung heraus, dass ϕ nicht nur das Keissegment um das ICC ist, sondern auch die Orientierung des Fahrzeuges im Bezug zum Ursprung der Ebene. Also berechnen sich die Koordinaten in der x-y-Ebene wie folgt:

$$(I): \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} R \cdot \cos(\Phi + \alpha) - \frac{k}{\tan(\alpha)} \\ R \cdot \sin(\Phi + \alpha) - k \end{bmatrix}, \quad (II): \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} R \cdot \cos(\pi - \Phi + \alpha) - \frac{k}{\tan(\alpha)} \\ R \cdot \sin(\pi - \Phi + \alpha) - k \end{bmatrix} \qquad (1)$$

Da das Fahrzeug je nach Vorzeichen des Lenkeinschlags den Kreisbogen in unterschiedlicher Richtung durchquert und sich auch der Startpunkt verlagert, muss man an dieser Stelle eine Fallunterscheidung machen. Ist der Lenkwinkel positiv, also lenkt das Auto nach Links, so wird der Kreis in mathematisch positivem Sinne durchfahren und die Gleichung (I) gilt.

Lenkt man nach Rechts, also ist der Lenkwinkel negativ, so wird der Kreis in mathematisch negativem Sinne durchquert mit dem Startpunkt π. Es gilt deswegen Gleichung (II).

Bei α=0 entsteht ein mathematischer Grenzfall, ein Kreis mit unendlich großem Radius. Was in der Praxis bedeutet, dass man geradeaus fahren möchte. Dieser Fall wird in der Anwendung einfach im Code abgefangen und produziert eine einfache Translation ohne rotatorischen Anteil.

Jetzt kann man die homogene Transformationsmatrix T der Vorwärtskinematik aufstellen mit der man Punkte vom lokalen in das globale Koordinatensystem transformieren kann und umgekehrt.

$$
\begin{aligned}
{}^{0}T_i &= trans(x,y,0)\cdot rot(z,\Theta) = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(\Theta) & -\sin(\Theta) & 0 & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
\Rightarrow \quad {}^{0}T_i &= \begin{pmatrix} \cos(\Theta) & -\sin(\Theta) & 0 & x \\ \sin(\Theta) & \cos(\Theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad mit \quad \Theta = \begin{pmatrix} -\Theta, & wenn & \alpha < 0 \\ \Theta, & wenn & \alpha > 0 \end{pmatrix}
\end{aligned}
\tag{2}
$$

Die vierte Spalte der i-ten Matrix entspricht dem letzten Translationsvektor des Fahrzeuges in der X-Y-Ebene. Spalte eins und zwei bilden die Einheitsvektoren des i-ten Koordinatensystems. Mit Hilfe des Arcus-Tangens mit Sektorinformation lässt sich die Orientierung des Fahrzeuges in der Ebene im Bezug zum 0-ten Koordinatensystem als Yaw-Winkel (Rotationswinkel um die Z-Achse) berechnen.

4.2.2 Anwendung der Odometrie

In jedem Durchlauf der while-Schleife in der main Funktion des car_arduino nodes bekommt eine Instanz des Objekts ForwardKinematics die Information zum eingestellten Lenkwinkel und der seit der letzten Iteration zurückgelegten Strecke. Darin wird nun die Transformationsmatrix vom letzten Zeitpunkt zum aktuellen Zeitpunkt berechnet, in dem man für diesen Schritt annimmt, dass der letzte Messwert aus dem Basiskoordinatensystem kam. Somit ergibt sich die Transformationsmatrix von ${}^{i-1}T_i$ (i-1 nach i). Im nächsten Schritt wird dann an die letzte gespeicherte Matrix ${}^{0}T_{i-1}$ (0 nach i-1) ${}^{i-1}T_i$ anmultipliziert und es ergibt sich ${}^{0}T_{i-1} * {}^{i-1}T_i = {}^{0}T_i$ (0 nach i). Als Rückgabe erhält man, wie im Absatz darüber beschrieben, die Position in X-Y-Koordinaten und die Orientierung gemäß der Yaw Konvention.

4.2.3 Konstruktionsbedingte Probleme und Lösungsansatz

Es hat sich allerdings herausgestellt, dass die Verwendung der eingestellten Lenkwinkel zur Berechnung der Position und Orientierung einen Fehler erzeugt, der mit der Strecke sehr schnell wächst und das Ergebnis der Berechnungen zunehmend unbrauchbarer macht.
Der Fehler entsteht, weil der Lenkwinkel nicht direkt eingestellt wird, sondern über eine Stellgröße $s \in [-50, 50] \subseteq \mathbb{Z}$ die ein Servo am Vorderrad in einen Winkel umsetzt. Das heißt man muss vorher alle Stellgrößen einstellen, den entsprechenden resultierenden Lenkwinkel messen und in einer Lookup-Tabelle festhalten. Allerdings stellt man schnell fest, dass die Winkel die den Stellgrößen zugeordnet sind, sich einerseits zur Laufzeit ändern und von Fahrzeug zu Fahrzeug unterschiedlich ausfallen können.
Die Lösung des Problems ist, die Verwendung des zusätzlich angebrachten Gyroskops, was den Orientierungswinkel in der Ebene durch Integration der Winkelgeschwindigkeit um die Z-Achse über die Zeit liefert. Diesen kann man einfach in die Sinus und Cosinus Argumente der Transformationsmatrizen einsetzen und die aufwändige Berechnung über die Lenkstellgrößen entfällt vollständig und damit auch der oben beschriebene Fehler.
Natürlich ist diese Methode auch nicht fehlerfrei, da die Integration der Winkelgeschwindigkeiten den Winkelfehler über die Zeit größer werden lässt, der Winkel fängt mit der Zeit langsam an zu "klettern". Mit jeder Richtungsänderung fängt diese Art Verhalten wieder von vorne an. Allerdings ist dieser Fehler über die durchschnittliche Betriebsdauer des Fahrzeugs gesehen und durch Korrektur von später erläuterten Systemen zwar nicht unbedingt vernachlässigbar aber tolerierbar klein.
Es ist dennoch sinnvoll die Methoden zur Berechnung der Position und Orientierung über die Lenkstellgrößen zu behalten, da später vielleicht die Möglichkeit besteht die eingestellten Lenkwinkel über eine Mechanik mit Hilfe eines Drehgebers direkt zu messen. So könnte man über beide Verfahren die Position berechnen und durch Fusion beider Daten eine noch viel genauere Positionsbestimmung erlangen, welche weder über die Strecke noch über die Zeit an Genauigkeit einbüßt.

4.3 Navigation und TF

Als nächstes muss die Positionsinfomation, die der Knoten car_arduino mit Hilfe der Sensoren am Arduino bereitstellt und die optische Information der Kinect in Einklang gebracht werden. Zusätzlich soll die bisherige Arbeit mit dem ROS Standard vereint werden, vorallem mit Hinblick auf die ROS ßtd_msgs“ Datentypen. Ziel ist es zuverlässige Positionsdaten zu liefern, ohne vom Anwender zu verlangen, den Ursprung dieser vollständig verstehen zu müssen.
Der ROS Navigation-Stack [13] liefert in Verbindung mit dem Paket „transform“- tf [14] gute Lösungen für die aufgezählten Forderungen. Vor allem bietet der Navigation-Stack auch die Möglichkeit einen beliebigen globalen Wegplanungsalgorithmus einzusetzen, in diesem Fall sbpl.

Das heißt, dass alle Odometriedaten mit tf an Navigation gesendet werden, dort werden sie von „amcl“ (siehe Kapitel 4.3.3) mit den Daten der optischen Odometrie, die aus der Kinect gewonnen werden, vereint. Anschließend werden die korrigierten Positionsinformationen des Autos von tf gebroadcastet. Zwischen diesen Schritten wird tf dazu verwendet, um entweder Transformationen wegen der räumlich unterschiedlichen Lage der Bauteile durchzuführen oder weil das Fahrzeug sich bewegt hat und man diese dann neuen Koordinaten im Entsprechenden Kontext der Weltkarte erhalten möchte.
In der Grafik unten kann man gut erkennen wie tf zwischen den einzelnen Kontexten vermittelt und dabei auch beobachtet wie oft Informationen gepublished werden.

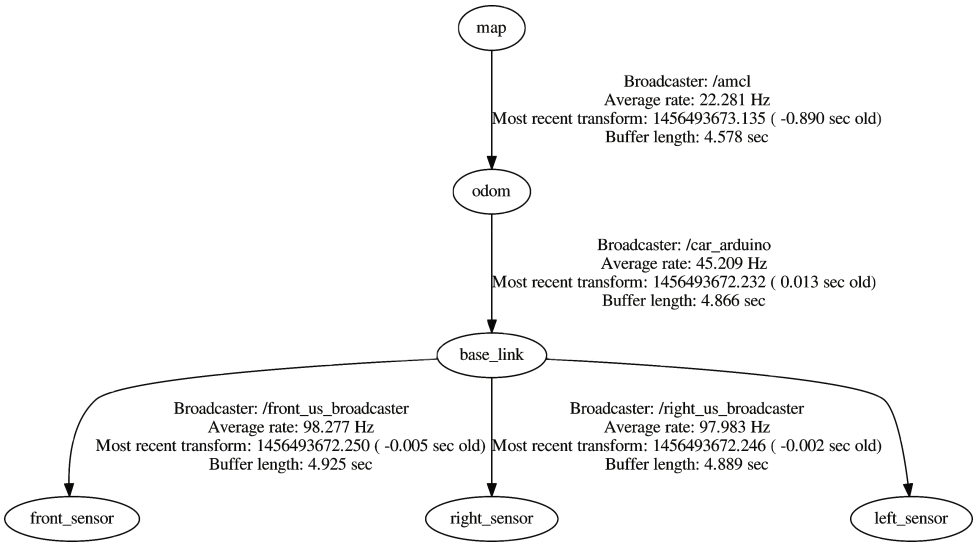

Abb. 4.2: *Teil des TF-Trees*

4.3.1 Rviz

Rviz ist ein ROS Standardprogramm um beispielsweise Positionsdaten zu visualisieren. Dank Navigation und tf werden standardisierte Nachrichten versendet und Rviz kann benutzt werden um das Auto und dessen gefahrenen Weg in der Welt darzustellen. Das ist sehr hilfreich um Fehler in den Transformationen zu finden, das Verhalten von neu implementierten Funktionen in Echtzeit zu beobachten und um schnell zu sehen ob Sensoren noch richtig funktionieren. Die meisten Bilder, die in dieser Ausarbeitung Visualisierungen zeigen, wurden mit Rviz erstellt. Außerdem ist es später für den sbpl-planner sehr nützlich um Navigationsziele zu setzen, zu denen dann ein Weg geplant werden soll.

4.3.2 Mapping

Mit den nun verfügbaren Transformationen von tf kann mit Hilfe des ROS Paketes „gmapping“ [15] eine Karte der Umgebung erstellt werden. Das Paket benötigt dafür Odometrieinformationen des Fahrzeuges und die Entfernungsdaten eines planaren Laserscans. Da in diesem Fall kein Planarlaser zur Verfügung steht, sondern eine Kinect, muss man die Punktwolke in einen Laserscan umwandeln.

Das ist im Prinzip einfach, die Punktwolke wird oberhalb und unterhalb einer bestimmten Höhe, bei diesem Auto etwa 35cm, mit einer gewünschten Toleranz abgeschnitten und in das gewünschte Datenformat eines Planarlasers umgewandelt. Hier wird diese Aufgabe vom Paket „depth_image_to_laserscan“ [16] erledigt, womit die zweite Anforderung von gmapping erfüllt ist. Das Mapping erfolgt dann mit Hilfe eines SLAM (simultaneous localization and mapping) Algorithmus, der gleichzeitig die Position und Orientierung während einer Erkundungsfahrt schätzt und die empfangenen Tiefeninformation auf einer Karte anordnet. Diese Karte kann dann mit Hilfe des „map_servers“ vom Navigation package dazu benutzt werden um einen Roboter in der Welt einzuordnen und darauf zu planen.

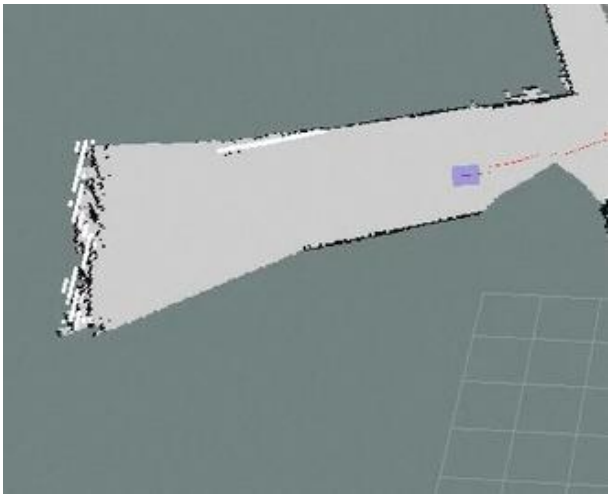

Abb. 4.3: *Der Laserscan(weiß markierte Pixel) beim Mappen*

Abb. 4.4: *Rechts die mit „gmapping“ erstellte Karte nach einer Erkundungsfahrt durch das HBI 3. Stock. Links der Grundriss, wobei die mit Grün hinterlegten Flächen in etwa den erkundeten Bereich aufzeigen sollen.*

4.3.3 amcl

Ein weiterer Vorteil bei der Verwendung von Navigation ist das beigelegte „amcl“ (adaptive Monte Carlo localization) Paket [17]. Dieser Algorithmus nimmt die vom „map_server“, der Teil vom Navigation-Stack ist, bereitgestellte Karte der Umgebung und einen Laserscan um die Qualität der mit tf transformierten odometrischen Daten zu bewerten und zu verbessern.

Amcl überprüft wie wahrscheinlich es ist, dass die angebliche Position des Roboters korrekt ist, indem die Daten des Laserscanners mit der Karte abgeglichen werden und formuliert daraus eine neue Hypothese über den Aufenthaltsort des Fahrzeuges.

Zunächst wird ein grober Hinweis auf den Aufenthaltsort benötigt, diesen kann man in Rviz mit einem „2D Pose estimate“ geben. Dann generiert amcl auf Basis der Initialwerte mehrere Vermutungen über den Aufenthalt des Autos. Mit zunehmender Anzahl an eingehenden Laserscannerdaten und Odometriedaten wird die Schätzung immer besser und unbrauchbare Hypothesen werden verworfen.

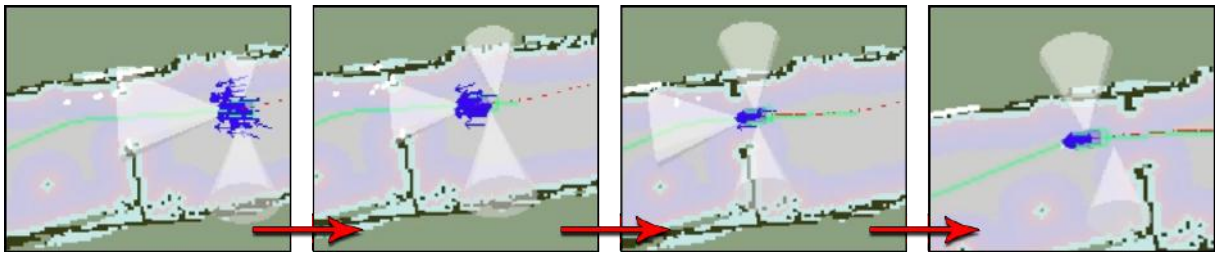

Abb. 4.5: *Zeigt wie „amcl“ über die Zeit Hypothesen verwirft und die Position immer besser schätzt. (Verlauf von Links nach Rechts)*

Dieser Algorithmus findet nicht nur Anwendung bei der Selbslokalisierung in der späteren Wegplanung, sondern wird auch während dem SLAM-Verfahren angewendet und korrigiert zur Laufzeit nicht plausible Aufenthaltsorte des Fahrzeugs.

4.4 Wegplanung mit sbpl

Die Wegplanung ist ein komplexes Problem, woran aktuell geforscht wird. Im Folgenden wird erklärt welche Herausforderungen sich dem Team gestellt haben, was sbpl ist, warum es verwendet wird und wie sbpl mit dem Auto verwendet wird.

4.4.1 Herausforderung der Wegplanung

Erst als man sich klar gemacht hat was bei der Wegplanung alles in Betracht gezogen werden muss, hat sich herausgestellt, dass die Lösung zu dem Problem eine größere Herausforderung darstellt als vermutet.

Die grundsätzliche Überlegung ist einfach. Man hat eine Karte der Umgebung auf dem sich das Auto bewegen soll, diese Karte wird diskretisiert und in Form eines Graphen dargestellt. Dabei sind Knoten Positionen an denen sich das Auto befinden kann. Die Diskretisierung der Karte soll so funktionieren, dass man ein Gitter darauf legt und Kreuzungen Knoten sind. Hier treten schon die ersten Probleme auf und es folgen viele weitere: Welche Knoten müssen verbunden verbunden werden, sodass das Auto nicht durch Objekte fährt, wie sind dann die Kanten zu Gewichten, sodass der schnellste und beste Weg gefunden wird, wie ist der Graph mit der Odometrie vereinbar, wie kann sich das Auto auf dem Graphen wieder finden usw.

Nach langer Recherche kam die Erkenntnis, dass die Problematik ein aktuelles Forschungsthema ist und ansatzweise im ROS-Package „sbpl“ gelöst wurde. Das eigenständige lösen eines aktuellen Forschungsthema hätte den Rahmen des Projektseminars gesprengt, außerdem wäre nicht klar gewesen ob überhaupt eine Lösung in der Zeit hätte entwickelt werden können. Aus diesem Grund erscheint es angemessen auf einen bereits vorhandenen Lösungsansatz, in Form eines Open Source Paketes, zu zuarbeiten.

4.4.2 sbpl

Das ROS-Package „sbpl“ wurde von der Arbeitsgruppe **s**earch-**b**ased-**p**lanning-**l**aboratory der Universität Carnegie Mellon in Pittsburgh entwickelt.

Mit „Search-based planning“ sind Wegplanungs-Algorithmen gemeint, die die Wege planen in dem die Umgebung diskret dargestellt wird, dass heißt in unserem Falle als einen Graphen. Um nun von einem Punkt A nach B zu gelangen, kann man einen Graph-Algorithmus anwenden. „Search-based planning“ stellt somit folgende Herausforderungen: Wie stelle ich meine Umgebung als Graph dar und wie findet man dann den besten Weg von Punkt A nach B. Die Arbeitsgruppe aus Pittsburgh hat sich diesen Problemen gestellt und präsentiert mit dem Package einen funktionierenden Planer.

Das sbpl-Package erlaubt es einen Punkt auf einer Weltkarte anzugeben, es wird dann ein Weg geplant und das Auto fährt dieses Ziel an, ohne mit der Umgebung zu kollidieren. Grob kann man die Funktion der sbpl-planners in drei Schritte zusammen fassen:

1.Die Umgebung wird als Graph dargestellt, dabei sind Knoten diskrete Punkte an der der Roboter sich befinden kann und Kanten zwischen den Knoten entsprechen gefahrenen Strecken mit bestimmtem Kostenfaktor.

2.Die Gültigkeit der Knoten wird dahingehend überprüft,ob es Kollisionen gibt wenn der Roboter sich auf besagtem Knoten befindet oder ob die Kosten der eingehenden Kante im vergleich zu einer Anderen zu hoch sind.

3. Anwendung von bekannten Graph Algorithmen um kürzeste Pfade anhand von Kostenfunktionen zu planen. Prinzipiell sind alle Graph Algorithmen möglich, der sbpl-planner verwendet jedoch einen ARA* - Algorithmus, welches im Prinzip ein modifizierter Dijkstra-Algorithmus ist.

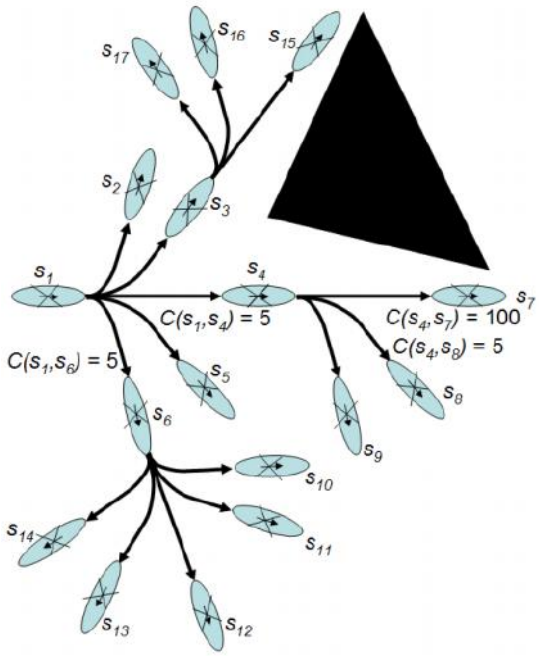

Abb. 4.6: *sbpl-Graph aus Motion Primitives*

4.4.3 Motion Primitives

Die Kanten des sbpl Graphen bestehen aus sogenannten Motion Primitives. Motion Primitives sind verschiedene Grundmanöver mit Informationen wo sich das Auto anschließend befindet, sie werden einmal vor der Laufzeit berechnet und machen damit den Aufbau eines Graphen weniger Rechenaufwändig. Damit die im Paket enthaltene Matlab-Funktion ein Motion Primitive berechnen kann, muss man als Parameter die Änderung der X- und Y-Koordinate des Fahrzeuges nach einem Manöver angeben.

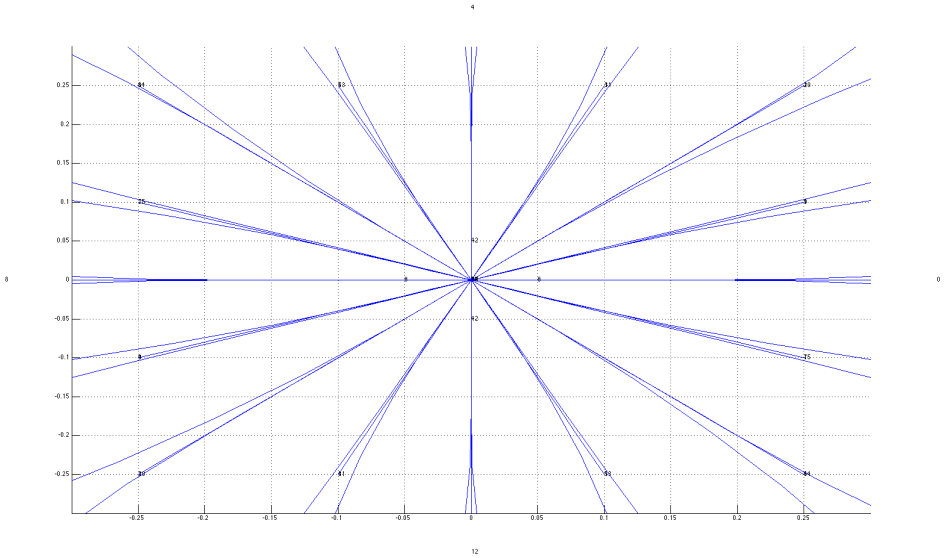

Abb. 4.7: *Motion Primitves die mit einem Matlab-Hilfsprogramm erstelltet worden sind*

Wobei dieses Manöver immer ein ganzzahliges Vielfaches einer vorher definierten Winkeländerung bewirken soll. Es ist nicht notwendig Grundmanöver für jeden der diskreten Winkel zu bestimmen, die Funktion benötigt nur Definitionen von 0 bis 45 Grad, der Rest wird wegen der Symmetrieeigenschaften interpoliert. Diese Positionsänderungen kann man aus der Vorwärtskinematik berechnen, indem man diese nach den Stellgrößen Strecke und Lenkwinkel auflöst, also die Inverskinematik bestimmt.

$$
\begin{aligned}
&(III):\ y = R\cdot\sin(\Theta+\alpha)-k \quad\Leftrightarrow\quad (y+k)^2 = R^2\cdot\sin^2(\Theta+\alpha) \quad sowie\\
&(IV):\ x = R\cdot\cos(\Theta+\alpha)-\frac{k}{\tan(\alpha)} \quad\Leftrightarrow\quad \left(x+\frac{k}{\tan(\alpha)}\right)^2 = R^2\cdot\cos^2(\Theta+\alpha)\\
&mit\quad (III)\&\,(IV) \quad\Rightarrow\quad \ldots \quad\Rightarrow\quad \alpha = \tan^{-1}\left(\frac{2xk}{-x^2-y^2 2yk}\right),\quad (V)
\end{aligned}
\tag{3}
$$

Hat man α aus (V), so kann man den Radius um das ICC berechnen, mit der Gleichung oben. $\Rightarrow$ (VI)
Jetzt kann man Theta mit Hilfe der Gleichungen (I) , (II) und dem Ergebnis aus (V) und (VI) bestimmen:

$$\text{für (I) gilt:} \quad \Theta = \sin^{-1}\left(\frac{y+k}{R}\right) + \alpha\,, \quad \text{für (II) gilt:} \quad \Theta = \pi + \alpha - \sin^{-1}\left(\frac{y+k}{R}\right), \quad (VII) \tag{4}$$

Zuletzt kann man die zu fahrende Strecke S mit (V), (VI) und (VII) berechnen mit:
$S = \theta \cdot R$, wobei zu beachten ist, dass S vorzeichenbehaftet sein kann und die Fahrtrichtung angibt.
Man benötigt hauptsächlich Gleichung (III) und (IV) um ΔX und ΔY zu bestimmen, für α kann man zum Beispiel den maximalen Lenkeinschlag verwenden und für Θ die geforderte diskrete Winkeländerung. Allerdings muss man dabei beachten, dass die Anfangs -und Endpunkte der Primitives immer auf Gitterknoten liegen, sonst kann der sbpl-planner aus den diskreten Manövern später keine kontinuierlichen Wege erzeugen. Daher hat es sich als Hilfreich erwiesen, die MotionPrimitives von einem C++ Programm, dem „motion_primitives_designer“[12], überprüfen zu lassen um sicher zu stellen, dass die Zwangsbedingung der kontinuierlichen Pfade erfüllt bleibt. Da die berechneten Größen fast immer die Zwangsbedingung verletzt haben, sind die MotionPrimitives für dieses Fahrzeug größtenteils durch Ausprobieren im Editor des Designers entstanden, wobei man sich möglichst an den vorher berechneten Werten orientiert hat.

4.4.4 Anwendung auf dem Auto

Wenn das Planungsverfahren sbpl arbeitet und ein „navigation goal“ vorgegeben ist, so veröffentlicht das Paket, sobald ein Weg berechnet ist, Stellgrößen auf seinem Topic „cmd_vel“.

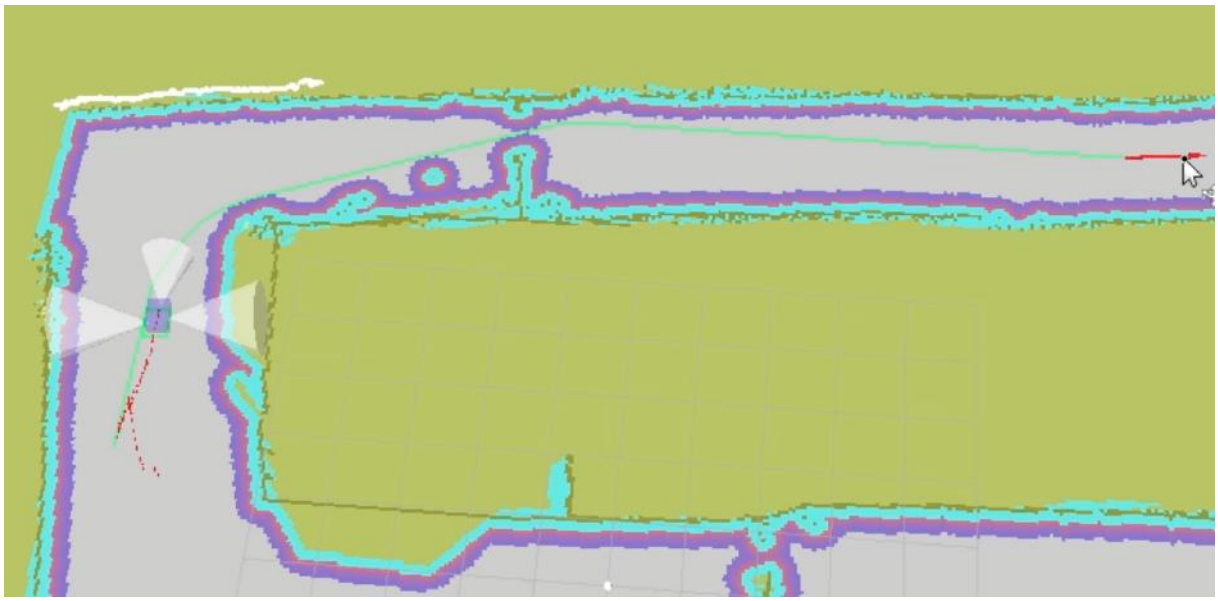

Abb. 4.8: *eine erfolgreiche Wegplanung(grüne Linie) zu einem „navigation goal“(roter Pfeil)*

Diese Stellgrößen umfassen zum Einen den einzustellenden Lenkwinkel und zum Anderen die geforderte Bahngeschwindigkeit. Im Node „car_arduino“ wird dann dieser Topic abboniert und die gesendeten Werte so umgewandelt, dass der Roboter damit seine Aktoren ansteuern kann. Die Lenkwinkel werden mit Hilfe einer „lookup table“, ein Array in dem jeder Stellgröße ein Lenkeinschlag zugeordnet ist, umgerechnet. Diese Tabelle wurde erstellt indem zu jedem zehnten Steuerwert der Lenkwinkel gemessen wurde, die Werte dazwischen wurden dann mit Matlab interpoliert.

```
double angleArray[101]={ 25.0,  24.6,  24.2,  23.8,  23.4,  23.0,  22.6,  22.2,  21.8,  21.4,
                         21.0,  20.6,  20.2,  19.8,  19.4,  19.0,  18.6,  18.2,  17.8,  17.4,
                         17.0,  16.5,  16.0,  15.5,  15.0,  14.5,  14.0,  13.5,  13.0,  12.5,
                         12.0,  11.7,  11.4,  11.1,  10.8,  10.5,  10.2,   9.9,   9.6,   9.3,
                          9.0,   8.7,   8.4,   8.1,   7.8,   7.5,   7.2,   6.9,   6.6,   6.3,
                          6.0,   5.2,   4.4,   3.6,   2.8,   2.0,   1.2,   0.4,  -0.4,  -1.2,
                         -2.0,  -2.3,  -2.6,  -2.9,  -3.2,  -3.5,  -3.8,  -4.1,  -4.4,  -4.7,
                         -5.0,  -5.4,  -5.8,  -6.2,  -6.6,  -7.0,  -7.4,  -7.8,  -8.2,  -8.6,
                         -9.0,  -9.4,  -9.8, -10.2, -10.6, -11.0, -11.4, -11.8, -12.2, -12.6,
                        -13.0, -13.5, -14.0, -14.5, -15.0, -15.5, -16.0, -16.5, -17.0, -17.5, -18.0};
```

Abb. 4.9: *Zeigt wie Lenkwinkel den Steuergrößen in der Klasse „car_arduino“ zugeordnet sind*

In Zukunft wäre es sinnvoll die Tabelle während des Betriebs zu verbessern, da man über die Vorwärtskinematik weiß welcher Lenkwinkel für ein gefahrenes Manöver eingestellt war. Vorraussetzung dafür ist aber eine absolut zuverlässige Odometrie, die über die Zeit nicht an Qualität einbüßt, sonst schlägt sich das auf die Tabelle nieder und der Roboter wird immer unpräziser. Bei der Umwandlung der Geschwindigkeitsvorgaben, wird derzeit nur betrachtet ob nach Vorne, Rückwärts oder gar nicht gefahren werden soll. Soll der Roboter nach vorne Fahren so geschieht das mit der Geschwindigkeitsstufe 2, was etwa 22cm/s entspricht. Muss das Auto rückwärts fahren, so wird Stufe -3 gewählt, was ca. -22cm/s entspricht. Soll das Fahrzeug still stehen, so wird natürlich Stufe 0 gewählt. Diese Einschränkungen sind nicht willkürlich entstanden, sondern mussten so gewählt werden, da der Drehgeber, der die zurückgelegte Distanz misst, durch Verschleiß immer ungenauer geworden ist. Eine für die Odometrie ausreichende Genauigkeit ist leider über Stufe 2 bzw. unter Stufe -3 nicht mehr gewährleistet. Wäre dies nicht der Fall, so spricht nichts dagegen die Geschwindigkeit des Autos dynamisch zu regeln, da man die maximale Geschwindigkeit die der lokale Planer vorgibt in einer Konfigurationsdatei festlegen kann. Nach der Umrechnung werden die Daten dann an den Knoten „apollo_13“ gesendet, worin entschieden wird, je nach auf der Fernsteuerung eingestellten Modus, ob die Steuerwerte an „car_handler“, der mit der mit der Motorsteuerung kommuniziert weitergegeben werden.

4.5 Aufbau der angelegten Pakete und Knoten

4.5.1 Das Paket „car_handler“

Dieses Paket beinhaltet alle Knoten die mit speziellen Eigenschaften des Fahrzeugs umgehen müssen, wie dem Fahrverhalten oder der Kommunikation mit dem Fujitsu Mikrocontroller oder dem Arduino. Der Knoten „car_handler“ ist so gut wie unverändert von der Vorlage übernommen worden. Dieser ist ausschließlich für die Kommunikation mit dem Fujitsu-Board zuständig und enthält keine Logik. Es werden nur die Kontrollsignale von „apollo_13“ empfangen und so an die Lenk- und Motorsteuerung weitergegeben. Zusätzlich werden die Sensordaten der Ultraschallsensoren auf einem Topic veröffentlicht. In „car_arduino“ findet die Kommunikation mit dem Arduino statt und um unnötigen Datenverkehr zu vermeiden wird hier auch die Odometrie aus den Sensordaten berechnet und gepublished. Ebenfalls werden hier die Steuersignale vom „local planner“ für das Auto interpretiert, wie in 4.4.4. beschrieben und dann für „apollo_13“ veröffentlicht.

4.5.2 Das Paket „apollo_13“

Das nach der Gruppe benannte Paket verbindet Knoten aus „car_handler“, die sich um den Hardwareaspekt kümmern, mit der Fernbedienung und Zusatzunktionen. Im Knoten „apollo_13“ befindet sich eine Kontrollstruktur die anhand von Steuereingaben der Fernsteuerung entscheidet welcher Fahrmodus aktiv ist. Das kann entweder die manuelle Steuerung sein, das Anfahren eines „navigation goals“ oder das Durchfahren von Arcuo-Toren. Dann werden die Steuerdaten des jeweils aktiven Modus an „car_handler“ geschickt. Außerdem werden hier die Ultraschallsensordaten abonniert, in „sensor_msgs::Range“ Datentypen umgewandelt und für die Navigation veröffentlicht. Dort können die Daten dann benutzt werden um zur Laufzeit Hindernissen auszuweichen, die nicht auf der Karte verzeichnet sind. Der Knoten „car_aruco“ ist eine Zusatzfunktion, bzw. ein Fahrmodus, der wie in 5. beschrieben dazu dient durch markierte Tore zu fahren. Hier werden alle Grundlagen benutzt die in den anderen Knoten realisiert wurden. Möchte man zusätzliche Funktionen einbauen so geschieht das am besten im Paket „apollo_13“ und wird dann in der Kontrollstruktur des Knotens mit dem selben Namen verankert.

Da die Selbstlokalisierung, das Kartografieren, die Wegplanung und das Befahren dieser geplanten Pfade in Mission 2 erflogreich abgeschlossen wurde, soll nun eine Anwendung für die neu erlangten Fähigkeiten des Roboters gefunden werden.

Hierfür bietet sich der letzte Punkt der Standardaufgabenstellung an, dem bisher noch kaum Aufmerksamkeit geschenkt wurde, das Durchfahren von Toren die mit Aruco-Markern beklebt sind.
Bei dieser Aufgabe stellen sich mehrere Fragen:

1. Wie sehen solche Tore aus ?
2. Je nach Art des Tores, wo soll das Auto durchfahren, in der Mitte am Rand, oder im Zweifelsfall außen rum ?
3. Welche Orientierung soll das Fahrzeug am Ende eines Manövers haben ?

Punkt 1 lässt sich einfach festlegen, da keine Angaben gemacht wurden wie diese Tore auszusehen haben. In diesem Fall wird ein Tor durch zwei Marker, je einer links und einer rechts, begrenzt. Die Aruco-Marker werden zu diesem Zweck ausgedruckt und auf Pappkartons geklebt, so dass sie aus Sicht der Ultraschallsensoren und der Kinect als Hindernis wahrgenommen werden können.

Die Antwort zu Punkt 2 ergibt sich damit zum Teil von selbst, da die „Eckpfosten“ der Tore nun eine physische Ausdehnung besitzen und man beim Durchfahren die Begrenzungen nicht berühren möchte, muss das Auto durch die Mitte der Begrenzungen fahren. Um die Tore soll der Roboter nur fahren wenn es nicht möglich ist ohne Kollisionen das nächste Ziel zu erreichen.

Der letzte Punkt wird maßgeblich davon beeinflusst, ob mehrere Tore kurz hintereinander durchfahren werden sollen oder nicht. Da diese Möglichkeit im Rahmen der Möglichkeiten des Roboters liegen soll, ist es am besten wenn das Auto von dem Mittelpunkt eines Tores zum nächsten Zeigt. Das Problem dabei ist, dass man nicht pauschal davon ausgehen kann ob immer auf ein Tor ein weiteres Folgt.

Daher ist es momentan so gelöst, dass das Fahrzeug mit der Orientierung, mit der es die Tore entdeckt hat, nach der Durchfahrt auch wieder im Raum stehen soll. Dadurch wird im überwiegenden Teil der Fälle sichergestellt, dass der Roboter senkrecht auf der gedachten Linie zwischen den Begrenzungspfosten steht und so in alle Richtungen gleich gut weiterfahren kann. Das Funktioniert oft gut, weil auch die Marker am besten mit senkrechter Sicht auf ihre Flächen detektiert werden.
Weil die Wegplanung und das Erkennen von Markern bereits implementiert ist, gestaltet sich der Entwurf dieser Funktionalität als nicht all zu schwer, der Knoten „car_aruco“ tut im wesentlichen folgendes:
Während der Laufzeit packt der Knoten neue noch nicht erkannte Marker in eine Liste und sendet den erkannten Marker an Rviz um ihn darstellen zu lassen.

Alle Marker in dieser Liste sind vom Typ „visualization_msgs::Marker“, einzigartig und können nur einmal erkannt werden. Die Informationen mit denen die Marker angelegt werden, kommen von „ar_sys“ und werden durch abonnieren des Topics „ar_multi_boards/transform“ abgefragt.
Wenn der Anwender auf der Fernsteuerung den Modus „follow Aruco Markers“ aktiviert, so wird zuerst die Liste mit den erkannten Markern der Distanz nach aufsteigend zum Auto sortiert. Danach werden die zwei dem Auto am nächsten der Liste entnommen und es wird geprüft ob das Auto sich vor, oder Hinter dem möglichen Tor befindet und ob das Fahrzeug überhaupt in der Lage ist dieses zu durchqueren.

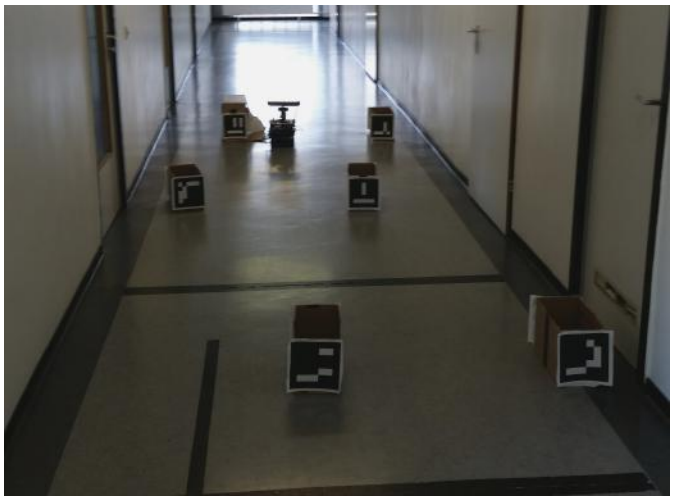

Abb. 5.1: *Der Roboter im Gang mit Markern*

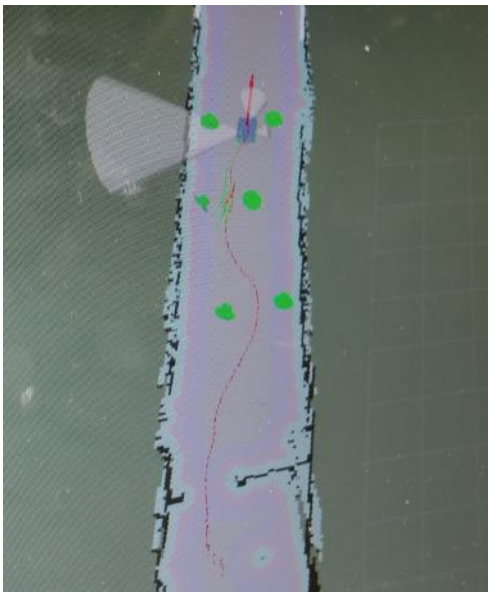

Abb. 5.2: *Der gefahrenen Weg und die in Grün markierten Marker die während der Fahrt erkannt wurden*

Sind die Bedingungen erfüllt, so wird als Zielpunkt die Mitte des durch die Marker definierten Tores angenommen und die Orientierung des Autos als Zielorientierung gesetzt. Wenn die Bedingungen nicht erfüllt sind, so wird der zweit näheste Knoten ans Ende der Liste zurückgelegt und der Nächste Knoten wird entnommen. Wurde das vorgegebene Ziel noch nicht erreicht, ist die Liste leer oder ist die maximal zulässige Zahl an Iterationsdurchläufen überschritten, so terminiert der Algorithmus bis zum nächsten Durchlauf der While-Schleife und es werden keine neuen Ziele mehr geplant.

6 Fazit

Im Folgenden wird reflektiert inwiefern die Ziele des Projektseminars mit der Projektarbeit erreicht wurden. Durch die gewonnen Erkenntnisse werden mögliche Verbesserungen und Ausblicke diskutiert.

6.1 Fazit: Ziele und Ergebnisse

In Kapitel 1 sind unsere selbstdefinierten Ziele auf Grundlage der gegebenen Aufgabenstellung beschrieben. Diese wurden vorsichtig realistisch definiert und auch im Zeitrahmen erfüllt
In Mission 1 wurde ein von vorangegangenen Projektseminaren altbekannter Lösungsansatz verfolgt, welcher sich gut mit der gegebenen Sensorik vereinen lässt. Da ein einfacher PID-Regler implementiert wurde, der für unsere Zwecke nur mäßig gut sein sollte, war es ausreichend an jeder Seite des Fahrzeuges einen Ultraschallsensor zur Verfügung gehabt zu haben. Zusätzlich musste die Kamera nicht mehr können als Aruco-Marker erkennen und das bei geringer Geschwindigkeit auf kurze Distanz, zu diesem Zwecke ist die gegebene Kamera ausreichend. Das autonome Fahren in der Form, sowie das durchfahren der Aruco-Tore wurden dadurch in der frühen Phase des Projektseminars abgeschlossen.
Durch den großen Zeitpuffer konnten wir viel Zeit in Mission 2 investieren. Da Mission 2 mit der standardmäßig gegebenen Sensorik sehr begrenzt bis garnicht zu erfüllen ist, war zunächst die Integration einer neuen Sensorik notwendig. Die Sensoren sind aus unserem eigenen Bestand und es wurde nur der Drehgeber während der Bearbeitungszeit gekauft.
Da wir zu Beginn nicht wussten ob wir mit dem neuen Konzept und der neuen Sensorik Erfolg haben werden, wirkt der Aufbau der Sensoren sehr improvisiert, da wir auch sehr auf Rückbaubarkeit geachtet haben. Dank dem Gyroskop, der Kinect-Kamera, dem Drehgeber und der vielen sehr guten ROS Pakete konnte Mission 2 erfolgreich abgeschlossen werden.
Beim Implementieren wurde darauf geachtet, dass Mission 2 möglichst in den navigation_stack vom Navigation Paket zu integrieren und std_msgs von ROS zu verwenden um eine gewisse Modularität aufzubauen.
Das heißt, dass die Teams in den nächsten Projektseminaren einfach Positionsdaten vom Navigation Paket abfragen können ohne zu wissen wie die Odometrie genau funktioniert. Außerdem können sie dem sbpl-planner Ziele aus den unterschiedlichsten Gründen schicken ohne sich in die kinematischen Eigenschaften des Fahrzeugs eingearbeitet zu haben. Desweiteren wurden mehrere Launch-Files erstellt die dem User möglichst einfach das Fernsteuern, autonome Fahren mit Zielsetzung, Durchfahren der Aruco-Tore oder das Mappen erlauben, ohne jedes Paket mit den entsprechenden Parametern einzeln ausführen zu müssen. Dazu wurde auch eine „ReadMe“ erstellt, die das Initialisieren der verschiedenen Modi genauer erklärt.

6.2 Fazit: Allgemein Projekseminar ES

Vorgestelltes Ziel des Projektseminars Echtzeitsysteme ist neben der fachliche Weiterbildung auch der Ausbau der Softskills. Wir, das Team Apollo 13, sind der Meinung dass das Projektseminar die Ziele im vollen Umfang erfüllt. Durch die offene Aufgabenstellung mussten sich die Teammitglieder zusammen mit dem Betreuer klare Ziele definieren, außerdem ist die Kommunikation im weiteren Verlauf sehr wichtig gewesen. Regelmäßige Treffen mit Geza Kulcsar verhinderten im Vorfeld Unklarheiten bei der Bearbeitung der Aufgaben. Die neue Plattform ermöglichte uns mit aktuellen Werkzeugen zu arbeiten zB. ROS – Robot Operating System welches auch in der aktuellen Forschung benutzt wird. Trotz neuer Plattform war es nur begrenzt möglich das autonome Fahren in unserer Vorstellung umzusetzen, was uns dazu veranlasst hat das Fahrzeug mit weiteren Sensoren auszustatten. Zusammenfassend lässt sich sagen, das Projektseminar bietet einem die Möglichkeit theoretisch gelerntes aus dem Studium in der Praxis anzuwenden, mit dem positiven Nebeneffekt zu lernen wie man sich in einer Gruppe organisiert.

6.3 Ausblicke und Verbesserungen

6.3.1 Verbesserungsvorschläge für die Konstruktion des Autos

Nachdem alle neuen Aufbauten auf dem Fahrzeug angebracht waren ist aufgefallen, dass das Fahrwerk durch das höhere Gewicht sehr stark belastet ist. Das äußert sich zum Einen in einem sehr kurzen Federweg, wobei das linke Hinterrad sogar schon fast am Hall-Sensor schleift. Zum Anderen, in bedenklichen Geräuschen beim Fahren und vor allem beim Lenken. Hin und wieder hatten wir während der Probefahrt Sorge wie lange die Lenkmechanik und der Antriebsstrang diese zusätzliche Belastung noch mitmachen. Die Gewichtszunahme des Autos führt direkt zum ersten Verbesserungsvorschlag, den Servos, die die Lenkung antreiben. Diese haben im Moment auf PVC-Boden einige Schwierigkeiten den gewollten Lenkeinschlag im Stand herzustellen, auf Teppichboden ist das gar nicht mehr möglich. Es wäre also zu erwägen, ob das Plastikfahrwerk eines ferngesteuerten Autos in Zukunft nicht durch ein stabileres ausgetauscht werden sollte und ob man das Gewicht der Aufbauten irgendwie reduzieren kann. In jedem Fall wäre ein stärkerer Antrieb der Lenkung, da stimmen wir auch mit der Meinung der anderen Gruppen überein, eine sinnvolle Verbesserung.
Als wir eine Möglichkeit gesucht haben den Arduino und den Gyro unterzubringen, hat sich die Befestigung auf dem Rahmen hinter der Kinect als am leichtesten zu realisieren herausgestellt. Allerdings ist uns im späteren Verlauf, vor allem als wir an die Kapazitätsgrenze des Systems gegangen sind, aufgefallen, dass der CPU recht hohe Temperaturen entwickelt, da durch das Holzbrett über dem passiven Kühler der Wärmeaustausch verschlechtert wird. Man könnte natürlich einfach einen Lüfter auf die Heatsink bauen, wie in jedem anderen PC üblich. Alternativ wäre es eine gute Idee einen anderen Ort zur Unterbringung des Arduinos zu suchen, das würde Gewicht von der Lenkachse nehmen und einen zusätzlichen Verbraucher in Form eines Lüfters vermeiden.

6.3.2 Verbesserungsvorschläge für die Sensorik des Autos

An der Ausstattung der Sensorik wie sie an unserem Auto existiert besteht zunächst nicht viel Handlungsbedarf. Vielmehr müsste die Anbringung einiger Sensoren verbessert werden.

In erster Linie muss der Drehgeber, der die zurückgelegte Strecke misst, irgendwie ins Gehäuse verlegt werden. Denn bei Fahrten mussten wir sehr darauf achten, dass wir den Drehgeber nicht abreißen oder sonst an irgendeinem Hindernis beschädigen. Bei der autonomen Fahrt war das weniger ein Problem, wir haben die „collision box“ entsprechend groß eingestellt, aber beim Transport oder ferngesteuerten Fahren hat man nicht immer alle Seiten des Autos im Blick und da kann der Sensor leicht beschädigt werden. Am schönsten wäre es, wenn ein Drehgeber oder eine Inkrementalscheibe über eine Mechanik, vielleicht mit einer Übersetzung, an der Heckachse angebracht wäre. So könnte man entweder auf eine höhere Genauigkeit abzielen oder mehr Geschwindigkeit zulassen.

Ein weiterer Punkt ist die Befestigung der Kinect, denn dort haben wir hauptsächlich darauf geachtet, dass man diese auch wieder leicht abnehmen und das Auto in den Ursprungszustand zurückversetzen kann. So hat sich ein ziemlich langer Hebelarm ergeben auf dem die Kinect sitzt und deshalb bei Beschleunigungen das Schwingen anfängt. Wir haben nicht genau nachgeprüft wie sehr sich das auf die Genauigkeit der optischen Odometrie ausgewirkt hat, aber in Rviz konnte man beobachten wie der Laserscan anfängt zu schwingen. Wir haben das temporär mit einem Klotz, der den Abstand zwischen Aufhängung und Vorderseite konstant hält, gelöst. Eleganter wäre es eine kleine Manschette zu drucken oder aus Alu zu sägen, die diese Verbindung permanent herstellt, so muss man die Aufhängung nicht komplett neu gestalten.

Die Stromversorgung der Kinect ist noch ein Punkt bei dem es Verbesserungsbedarf gibt, denn im Moment wird die Kinect über einen Spannungsbegrenzer von einem zweiten Akku über der Heckachse versorgt. Das bringt zwei Probleme mit sich, zum Einen haben wir keine Schaltung eingebaut die sicher stellt, dass der Akku keine Unterspannung erzeugt, was irgendwann den Akku beschädigen kann. Zum Anderen bringt ein zweiter Akku sehr viel Gewicht mit sich, was die oben genannten konstruktorischen Probleme erzeugt.
Von Vorteil wäre es, wenn wir die Kinect über das bereits vorhandene Bordnetz versorgen könnten, dann müsste man nicht auf den Ladestand von zwei Akkus acht geben.
Als weitere Verbesserung könnte man die Kinect durch eine ASUS Xtion Pro Live ersetzen, da diese zum Einen leichter ist als die Kinect und zum Anderen vollständig über USB mit Strom versorgt wird. Man würde damit nicht nur das Gewichtsproblem erheblich mindern, sondern auch keine neue Konstruktion der Stromversorgung benötigen.

Sollte die Lenkmechanik überarbeitet werden, wäre es unserer Meinung nach eine gute Idee, wenn man dabei an die Möglichkeit denken könnte einen Drehgeber anzubringen. So könnte man nach einstellen einer Lenkstellung überprüfen ob auch wirklich der gewünschte Lenkwinkel erreicht ist und eventuell nachregeln. Zusätzlich würde das auch die Möglichkeit eröffnen, eine klassische Odometrieberechnung anhand von Stellgrößen durchzuführen und diese dann mit der aktuell benutzen Variante zu fusionieren, siehe Kapitel 4.2.3.

6.3.3 Ausblick auf zukünftige Anwendungen

Im Verlauf des Projektseminars hatte unsere Gruppe Ideen was man noch mit der neuen Sensorik und Funktionalitäten des Autos machen könnte. Diese wurden allerdings aus Zeitmangel verworfen oder hinten angestellt.

Eine dieser Ideen war die Implementierung eines Erkundungsmodus, also das Durchfahren einer unbekannten Umgebung mit dem Ziel diese zu kartographieren. Das sollte möglichst autonom passieren und nach erreichen von ein oder mehreren Kriteria von alleine beendet werden.

Eine weitere Idee war es, die Inertialsensorik noch besser auszunutzen. Wir haben uns überlegt, dass es mit Hilfe der Beschleunigungsmessung entlang der drei Raumachsen möglich sein muss, über doppelte Integration die Position im Raum zu bestimmen. Teilweise haben wir das nebenbei auch schon ausprobiert aber nie so weit entwickelt, dass es einsatzfähig oder präzise genug gewesen wäre. Der Vorteil dieser Positionsbestimmung wäre, dass das Auto unabhängig vom Kraftschluss der Räder seine Position im Raum bestimmen kann. Das Auto könnte also feststellen ob die Räder Schlupf entwickeln oder ob es getragen wird. Eine interessante Anwendung davon wäre es, die Inertialsensorik und die Kinect in einem Modul abnehmbar zu vereinigen. So könnte man zu Fuß ein Gebäude erkunden und kartographieren, das Modul dann auf den Roboter setzen, der dann eine vollständige Karte hat und direkt losfahren kann.

A Quellenverzeichnis

Quellenverzeichnis: (vom 04.03.2016)

[1]http://www.robot-electronics.co.uk/htm/srf08tech.html
[2]http://www.freertos.org
[3]http://www.ros.org
[4]http://henrysbench.capnfatz.com/henrys-bench/keyes-ky-040-arduino-rotary-encoder-user-m
[5]http://www.dipmicro.com/?datasheet=PS-MPU-6000A-00v3.4.pdf
[6]http://www.robotshop.com/media/files/PDF/ArduinoMega2560Datasheet.pdf
[7]https://dev.windows.com/en-us/kinect/hardware
[8] http://www.pjrc.com/teensy/td_libs_Encoder.html
[9]https://github.com/jrowberg/i2cdevlib/blob/master/Arduino/MPU6050/
[10]https://github.com/ros-drivers/freenect_stack
[11]http://playground.arduino.cc/Interfacing/LinuxTTY
[12]https://github.com/aurone/mprims
[13]http://wiki.ros.org/navigation
[14]http://wiki.ros.org/tf
[15]http://wiki.ros.org/gmapping
[16]http://wiki.ros.org/depthimage_to_laserscan
[17]http://wiki.ros.org/amcl

Bildquellen: (vom 08.03.2016)

[18]http://www.avc-shop.de/WebRoot/Store15/Shops/64272905/5360/3541/99B4/AF18/6AAD/C0A8/2AB9/F431/gyachseng-1398813913-20828.jpg
[19] https://pixhawk.org/_media/projects/screenshot_from_2014-11-06_14_18_54.png
[20] https://paradigmshiftright.files.wordpress.com/2014/12/wp_ss_20140522_0001.png
[21]http://sbpl.net/node/48