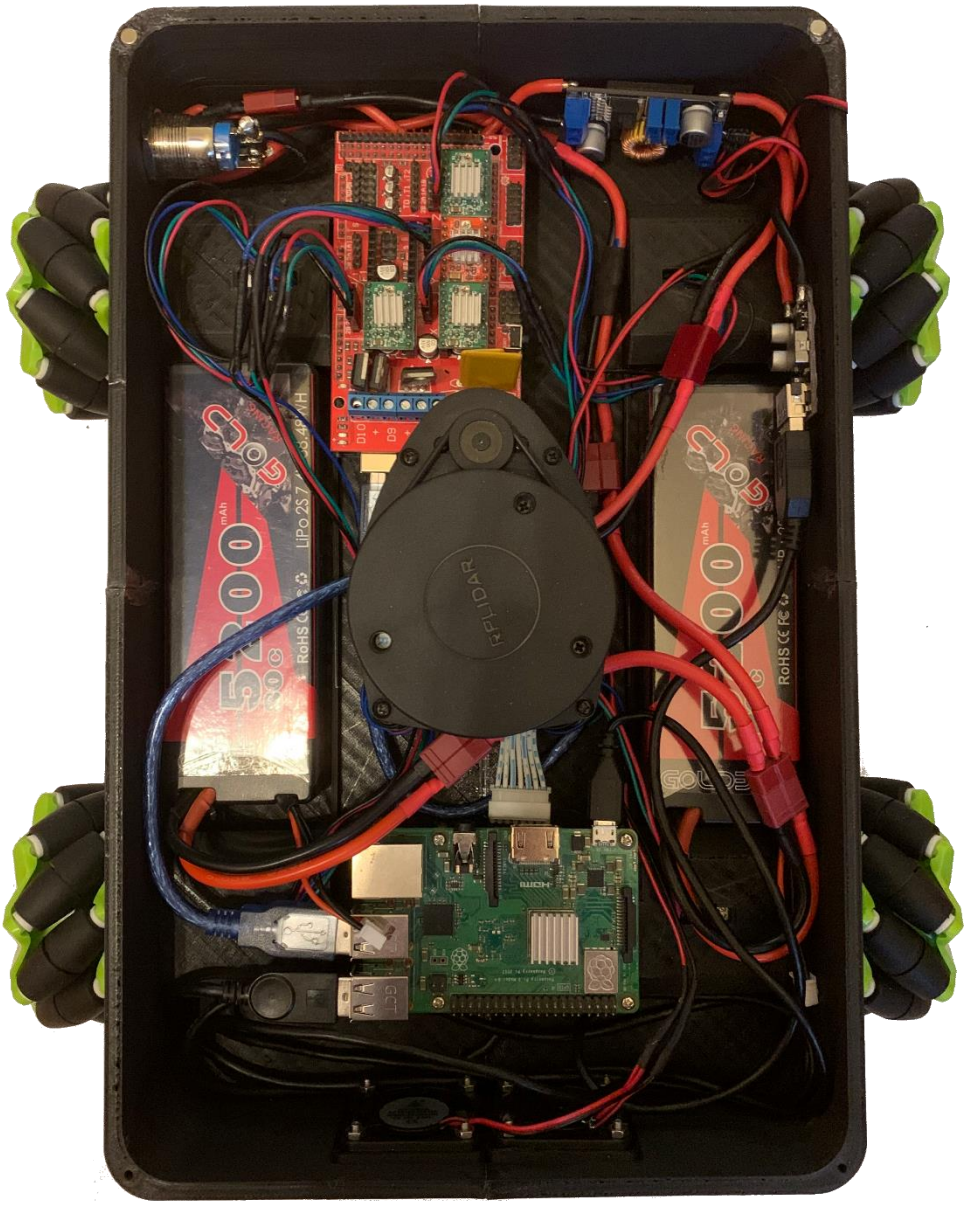
Auf LIDAR basierender selbstfahrender Roboter



BGT 191

Niklas Otten;
Maarten Behn;
Tim Jaeschke;
Yesenia Möhring

11.06.2021

Inhaltsverzeichnis

1 Einleitung

In unserem P5-Projekt haben wir uns das Ziel gesetzt, einen Roboter zu entwickeln. Diesen sollte man über eine App fernsteuern können. Mit Hilfe eines LIDAR Sensors sollte der Roboter seine Umgebung aufnehmen und mit den Daten des LIDAR Sensors seine eigene Position bestimmen.

Die Hauptbestandteile des Projektes haben wir wie folgt aufgeteilt:

- Den Bau und die Testung des Roboters hat Tim übernommen.
- Das Aufsetzen der Netzwerkverbindung war die Aufgabe von Niklas und Yesenia.
- Bei der Entwicklung und Testung der App haben Niklas und Maarten zusammengearbeitet.
- Die graphische Darstellung der App war die Aufgabe von Yesenia.
- Den SLAM Algorithmus V1 und V2 hat Maarten entwickelt. Bei V3 haben sich Tim und Maarten zusammengeschlossen.
- Niklas hat einen Pathfinding Algorithmus für spätere Versionen entwickelt.

2 Roboter (Tim)

2.1 LIDAR Sensor

LIDAR steht für „Light detection and ranging“ und ist ein Lasermessgerät, welches in mehrere Richtungen Entfernungen messen kann. Es gibt verschiede Arten, im neuen IPhone zum Beispiel ist ein LIDAR Sensor, welcher nach vorne viele Messpunkte misst. In unserem Fall haben wir einen LIDAR Sensor, der 360 Grad im 2D Bereich misst, mit einem Messbereich von 0.15m – 12m und sechs Umdrehungen in der Sekunde. Innerhalb einer Sekunde kann man mit unserer Einstellung bis zu 2000 Punkte messen. Die Daten werden als Polar-Koordinaten wiedergegeben.

Abbildung 1 Ein LIDAR Sensor.

2.2 Motorsteuerung

Bei der Motorsteuerung mussten wir uns als Erstes überlegen, was für Motoren wir verwenden wollen. Zuerst haben wir uns für normale DC-Motoren entschieden. Dies hat leider nicht funktioniert, da DC-Motoren nicht immer gleich schnell drehen und dadurch der Roboter nicht geradeaus fahren konnte. Deswegen haben wir uns nach anderen Motoren erkundigt und haben uns schlussendlich für Schrittmotoren entschieden. Diese haben einen sehr wichtigen Vorteil gegenüber DC-Motoren, denn man kann sie auf einen Grad genau kontrollieren, so dass alle gleich schnell fahren können.

Als Nächstes haben wir uns darüber Gedanken gemacht, wie man die Motoren am besten ansteuern kann. Dies wurde durch einen Arduino und ein Erweiterungsboard (ramps 1.4) realisiert. Auf dem Board sind vier Schrittmotortreiber (A4988), die den jeweiligen Motor kontrollieren. Da die Schrittmotoren und das Erweiterungsboard mit einer Spannung von 12V arbeiten, aber die beiden Akkus jeweils nur 7.4V ausgeben, sind die Akkus in Reihe geschaltet und geben eine Spannung von 14.8V aus. So mussten wir einen Spannungswandler zwischen die beiden Akkus und das Erweiterungsboard schalten, welcher die Spannung auf genau 12V herunterregelt.

Auf dem Arduino läuft ein Programm, welches ein serielles Signal empfängt und in die jeweilige Motorbewegung umwandelt. Über die serielle Verbindung werden 15 Byte gesendet:

- 2 Byte: Start(0x3C) und End(0x3E) Byte
- 8 Byte: 2 Byte pro Motor für Stepps (0 - 65,535)
- 4 Byte: Geschwindigkeit pro Motor (0 - 255)
- 1 Byte: Richtung (ersten 4 Bits)

Die Schrittmotortreiber funktionieren so, dass sie immer, wenn sie einen Impuls bekommen, den Motor einen Schritt machen lassen. In der ersten Version des Programmes wurden die Motoren im „main loop“ angesteuert. Das hat nicht richtig funktioniert, da das Programm zu langsam war oder von der seriellen Übertragung unterbrochen wurde. Das Problem wurde dadurch gelöst, dass ein Timer Interrupt genutzt wurde. Dieser wird alle 250ms ausgeführt. Er errechnet, welche Motoren einen Schritt machen müssen und steuert diese dann an.

Das serielle Signal wird von einem Raspberry Pi übertragen. Auf dem Raspberry Pi läuft ein TCP Server, der über TCP Signale von der App empfängt, interpretiert und an den Arduino weiterleitet.

2.3 3D-Modell und Drucken

Das 3D-Modell wurde mit dem Programm Fusion 360 erstellt. Zuerst haben wir einen Prototyp erstellt, um alles testen zu können. Danach wurde das finale 3D-Modell erstellt. Das ganze Modell wurde mit einem 3D-Drucker ausgedruckt. Da der Roboter größer ist als der 3D Drucker, musste das Gehäuse in acht Teile aufgeteilt werden.

Da die Komponenten warm werden, haben wir im 3D-Modell Lüfter eingeplant. Schlussendlich haben wir uns für vier Lüfter entschieden. Damit ein Luftstrom entsteht, haben wir uns für zwei Lüfter, die warme Luft aus dem Gehäuse rausblasen und zwei Lüfter, die kühle Außenluft einsaugen, entschieden.

Für ein paar Teile wie den LIDAR [1] als 3D-Modell (um die Halterung zu designen) oder die Lüfterhalter [2] habe ich vorgefertigte 3D-Modelle von der Seite „thingiverse“ genutzt.

Ebenso musste ein Adapter konstruiert werden, der die Motorenwellen mit den Rädern verbindet. Da dieses Teil sehr klein ist und die Motorenwellen nahezu eine runde Oberfläche haben, verschleißen die Teile sehr schnell und es kann passieren, dass die Motoren sich ohne die Räder drehen.

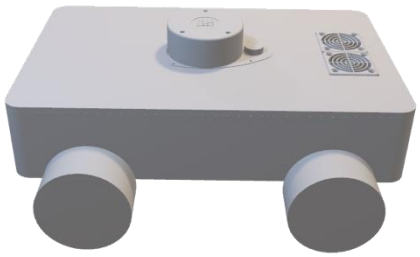


Abbildung 2 3D Modell des Roboters.

3 App

3.1 Network (Niklas)

3.1.1 TCP

Um unsere App mit dem Roboter zu verbinden, brauchten wir ein einfaches Protokoll, um Daten zu übertragen. Da wir W-Lan als Übertragungsweg nutzen, war das erste Protokoll, welches uns eingefallen ist, HTTP. Dieses wird auch für Internetseiten verwendet und ist weit verbreitet. Nach dem wir einen einfachen HTTP-Server in Python aufgesetzt hatten, wollten wir einen Client für diesen in Unity schreiben. Allerdings haben wir keine Dokumentation zu HTTP in Unity gefunden, daher wäre es in Unity wesentlich aufwändiger gewesen, einen HTTP Client zu schreiben. So haben wir uns für ein einfacheres Protokoll entschieden, TCP, welches das HTTP zugrundeliegende Protokoll ist.

3.1.2 Datenformat

TCP gibt uns die Möglichkeit, Daten zu übertragen. Um das Datenformat leicht verständlich zu machen und die Fehlersuche zu erleichtern, haben wir das Datenformat menschenlesbar gemacht.

Eine Beispielnachricht wäre

roboter summlidardata 20

Eine Nachricht gliedert sich in drei Worte Ort, Befehl und Argument, welche mit Leerzeichen getrennt sind.

Das erste Wort der Nachricht ist der Ort, an welchen der Befehl und das Argument geleitet werden. Die Idee hierbei ist, dass dieses Wort dem TCP Server/Client sagt zu welchem weiteren Programm er die empfangenen Daten senden soll. Zum Beispiel war ursprünglich die Idee, dass es ein Python-Programm gibt, welches Daten empfängt, um den Roboter zu bewegen und ein C++ Programm, welches die LIDAR Daten des Sensors auswertet.

Das zweite Wort ist der Befehl, der dem Programm mitteilt, was nun zu tun ist. In diesem Fall Daten messen und zurücksenden.

Das dritte Wort ist das Argument, welches extra Informationen zu dem Befehl gibt, in diesem Fall, dass der Sensor 20 Umdrehungen macht.

Um die Daten im Argument besser zu organisieren, haben wir uns ein System ausgedacht. Es können einzelne Informationen durch Kommata getrennt werden. Dies ist nötig, da zum Beispiel die LIDAR-Daten aus vielen Messpunkten bestehen. Allerdings besteht jeder einzelne Messpunkt aus zwei Zahlen. Daher werden diese mit dem nächsten Trennzeichen (Semikolon) getrennt. Wir haben uns gegen Klammern als Trennzeichen entschieden, da diese wesentlich schwerer auszuwerten sind. Die meisten Programmiersprachen bringen bereits die Möglichkeit mit, Zeichenketten an bestimmten Zeichen aufzuteilen. Dies macht es sehr einfach, ein Argument rekursiv zu verarbeiten: Man teilt die erste Ebene nach den Leerzeichen auf, ersetzt Kommata durch Leerzeichen und die anderen Trennzeichen mit dem Trennzeichen der Ebene darüber und gibt diese Liste an Argumenten einzeln an die entsprechende Funktion.

Ein Beispiel ist die Multi-Funktion, welche mehrere Bewegungsbefehle an den Roboter sendet.

multi move,1;1;1,rotate,90;1

Bei der Übertragung unserer Lidar-Daten sind wir dann auf eine interessante Begrenzung von TCP gestoßen. Wenn eine Nachricht zu lang wird, kommt es zu Fehlern bei der Übertragung. Daher mussten wir unsere Befehle zur Übertragung von LIDAR-Daten so umschreiben, dass mehrmals kleinere Abschnitte der Gesamtdaten gesendet werden und zum Schluss ein Befehl, um das Ende der Übertragung zu kennzeichnen.

3.1.3 Python zu C++

Ursprünglich hatten wir den gesamten Code, welcher auf dem Roboter laufen soll, in Python geschrieben. Es war jedoch nicht möglich, den Code für den LIDAR-Sensor in Python zu schreiben, da Python zu langsam für diesen Sensor ist. Daher musste der Code zum Auslesen des Sensors in C++ geschrieben werden. Da der TCP-Server und die Motorsteuerung schon in Python implementiert waren, wollten wir ein Programm schreiben, welches den LIDAR ausliest und diese Daten zurück an Python überträgt. Da wir keine funktionierende Schnittstelle hierfür gefunden haben, mussten wir den gesamten Code in C++ umschreiben. Zum Glück war das Python-Programm nur 122 Zeilen lang und so recht schnell umzuschreiben.

3.2 Design (Yesenia)

Meine Aufgabe in dem Projekt war es, sich ein Design für die App zu überlegen und umzusetzen.

3.2.1 Konzept

Unsere Überlegung war es, die App im Stil von Ironman bzw. dem Design der Overlays im Film zu gestalten. Im Internet findet man viele Bilder die diese Overlays isoliert also auf schwatzen Hintergrund zeigen.

Im Allgemeinen haben wir uns dann auf ein futuristisches Design geeinigt, wie es gerade in Science-Fiction-Videospielen und Filmen genutzt wird.

Ein Problem beim Designen war, das die Buttons abhängig vom Gerät, auf dem die App gerade läuft, an verschiedenen Stellen sind.

3.2.2 Shader

Einen Großteil der Grafik haben wir mit Shadern erstellt. Shader sind Programme, die im Allgemeinen auf der GPU laufen. Programme, die man auf einer GPU laufen lassen kann, sind sehr limitiert, sodass man eigentlich nur Rechnungen ausführen kann, um Graphiken zu verändern.

Es gibt verschiedene Programmiersprachen für Shader, die wir jedoch nicht genutzt haben, da die in Unity vorhandenen Shadergraphen deutlich übersichtlicher sind. Hierbei kann man mit bestimmten vorgefertigten Funktionen eigene Bilder verändern und animieren.

Abbildung 3 Startscreen unserer App.

Abbildung 4 Wenn die Verbindung zum Roboter aufgebaut worden ist.

4 SLAM

Einer der technischen Hauptbestandteile unseres Projektes ist es, mit den Daten des LIDAR Sensors die Umgebung zu erfassen und den Roboter in ihr zu lokalisieren. Dieser Prozess nennt sich SLAM (Simultaneous Localization and Mapping). Wir haben einige Versuche benötigt, um einen einfachen und zuverlässigen Algorithmus zu entwickeln, der diese Aufgabe löst. Von Anfang an haben wir uns dafür entschieden, keinen vorgefertigten SLAM-Algorithmen zu nutzen. Aktive Forschung in SLAM-Algorithmen gibt es schon seit den 90er Jahren. Deshalb haben wir uns entschieden, dass wir nicht auf all dieser Forschung aufbauen wollen, sondern unseren eigenen Weg finden wollen.

Wir haben uns so entschieden, da es nur fertige Algorithmen oder wissenschaftliche Abhandlungen zu dem Thema gibt. Da es darum geht, etwas Eigenes zu schaffen, mussten wir in Kauf nehmen, dass unser Projekt wesentlich schwieriger wird. Wissenschaftliche Abhandlungen hingegen konnten wir nur teilweise nutzen, da die verwendete Fachsprache und Mathematik für uns schwer bis unverständlich ist.

4.1 Version 1 (Maarten)

Unsere erste Idee war, die Lidar-Daten über TCP an die App zu senden und dort in die Position des Roboters umzurechnen. Um unabhängig arbeiten zu können, hat Tim einige Messungen mit dem LIDAR Sensor vorbereitet. Mit diesen konnten wir unabhängig von ihm den SLAM-Algorithmus erarbeiten.

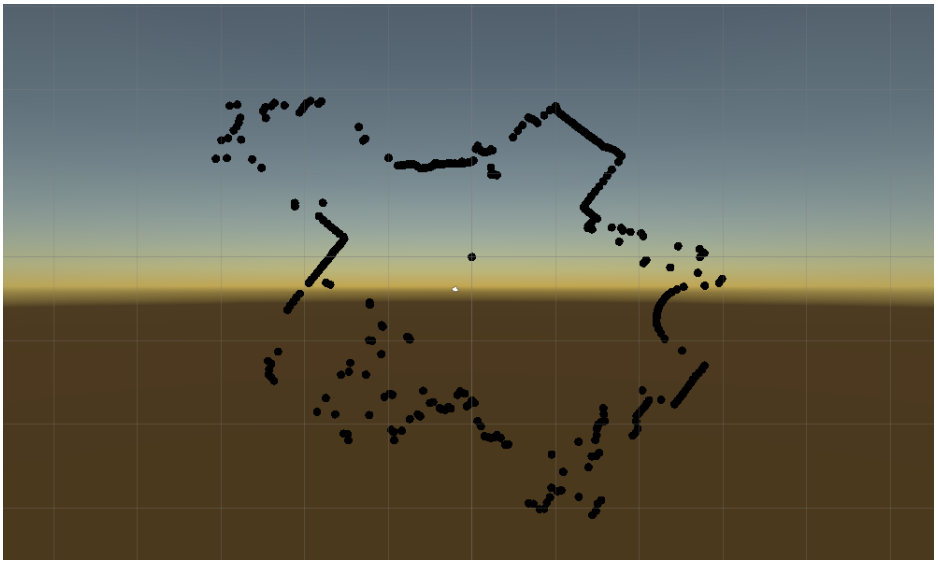


Abbildung 5 Rohdaten vom LIDAR Sensor.

In unserem ersten Ansatz wurde die Verschiebung errechnet, indem wir die Veränderung (Delta) der Mittelpunkte der Punktwolken nutzen.

$$\vec{d} = \frac{\sum_{i=0}^{n} \Delta \underset{p}{\rightarrow}_i}{n}$$

Hier steht d für das errechnete Delta, n für die Menge an Messpunkten in den Punktwolken und p von i für die Position des Messpunktes mit dem Index.

Dies führte aber schon bei Verschiebungen über 10 cm zu großen Fehlern. Der LIDAR-Sensor nahm zu viele Punkte auf, die schon bei kleinen Bewegungen nicht mehr gemessen werden können. Dieser Effekt sorgte dafür, dass sich der Mittelpunkt der Messdaten über die Zeit änderte und sie so verfälschte.

Um dieses Problem zu lösen, haben wir die Anzahl der Messpunkte reduziert, indem nur gerade Flächen betrachtet wurden. Diese gehören meistens zu festen großen Objekten, welche auf mehreren Messungen aufgenommen werden.

Der Algorithmus zum Finden dieser Flächen arbeitet wie folgt:

Er geht durch die Liste aller Messpunkte durch. Für jeden Punkt erzeugt er eine Gerade durch diesen und den nächsten in der Liste. Dann geht er nach

vorne und nach hinten von dem Punkt durch die Liste, bis der Abstand der Punkte von der gezogenen Geraden größer als ein maximaler Wert ist.

Alle Punkte, die er pro Punkt auf diese Weise findet, werden als eine Linie behandelt. Nun wählt er von allen Linien immer die längste aus und löscht alle Punkte der Linie aus allen anderen Linien. Das wiederholt er, bis die größte Linie zu wenig Punkte hat oder zu kurz ist. Durch diesen Algorithmus haben wir nun alle geraden Flächen, die der LIDAR aufgenommen hat, in einer Liste zusammengefasst. Doch auch hier macht unsere Delta-Rechnung große Fehler, da teilweise verschiedene Messdaten verschiedene Flächen des Raumes enthalten.

Zu diesem Zeitpunkt kamen wir erstmal nicht weiter, da teilweise neue Flächen erkannt werden, die nicht zuzuordnen sind. Wir hatten nun die Idee, statt den Flächen die Ecken im Raum zu vergleichen. Mit Ecken haben wir mehr Daten zum Mappen, da nun wir auch die Richtung und den Winkel der Ecke nutzen können. Diese Informationen helfen, zwei gleiche Ecken in den Messdaten zu finden.

Also haben wir einen Algorithmus geschrieben, der alle Flächen vergleicht. Wenn ein Endpunkt nah beim Roboter ist und der Winkel zwischen den beiden Geraden, größer als 20° und kleiner als 160° ist, haben wir diese als Ecke zusammengefasst.

Mit der Liste aller Ecken haben wir nun die Verschiebung und die Drehung ausgerechnet, indem wir alle möglichen Überlagerungsmöglichkeiten ausprobiert haben. Für jede Möglichkeit haben wir einen Wert ermittelt, der den Fehler angibt und sich aus dem Abstand der anderen Ecken ergibt. Schließlich ist nun die Überlagerungsmöglichkeit, die den kleinsten Fehler hat, die hoffentlich richtige. Diese haben wir für die neue Position und Drehung des Roboters genutzt.

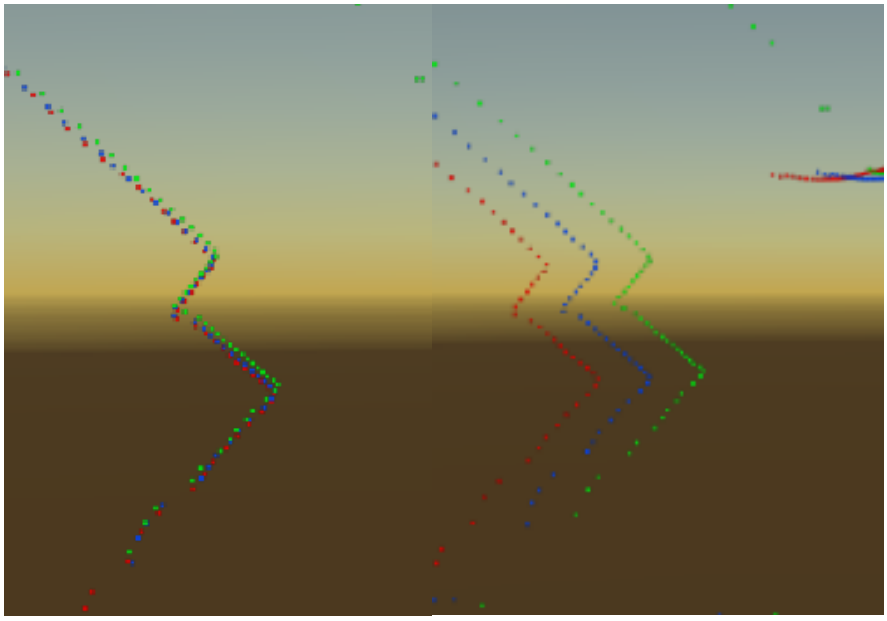


Abbildung 6 Messdaten vom LIDAR Sensor. Abbildung 7 Messdaten wurden übereinander gelegt.

4.1.1 Multithreading

In unserem Ansatz haben wir oft mit Brute-Force-Methoden gearbeitet, welches teilweise zu Performance-Problemen führt. Gerade das Errechnen aller Überlagerungsmöglichkeiten braucht seine Zeit. Daher haben wir für diese Rechnungen ein Multithreadsystem genutzt. In Unity ist der Hauptthread für Graphik und alle Scripte verantwortlich, daher verlagerten wir die ganze Berechnung der neuen Position in einen Hintergrund-Thread, welcher über mehrere Frames arbeiten kann. Hierfür haben wir ein Task-System gebaut. Dieses nutzt enum-Flags, um anzugeben, in welchem Schritt der Berechnung das Programm ist und ob wir die Daten in den Hauptthread einbinden können.

Außerdem haben wir ausprobiert, für jede Überlagerungsmöglichkeit einen neuen Thread zu starten. Da das Starten der Threads länger dauert als die eigentliche Rechnung, haben wir uns für einen langsamen Hauptthread entschieden. Der zweite Grund für diese Entscheidung war die Überlegung, dass unser Programm später auf Handys oder IPads laufen soll. Die haben nicht genug Kerne, so dass sich tausende parallele Aufgaben nicht lohnen würden.

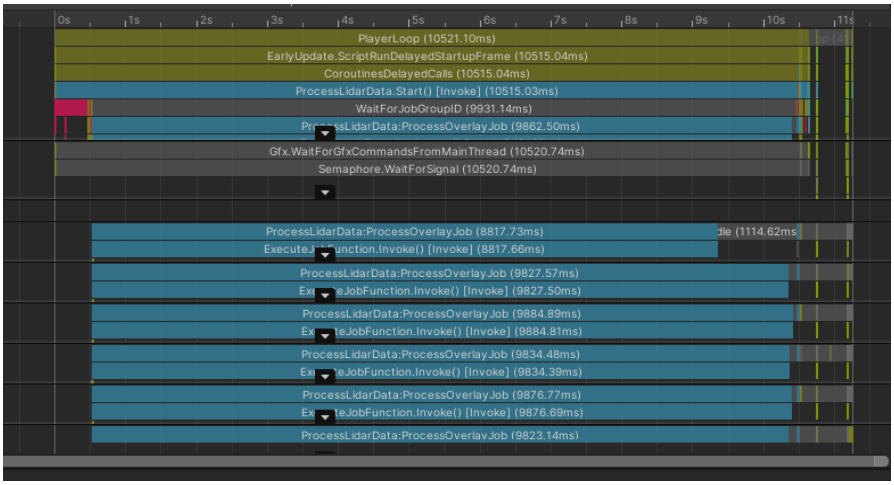


Abbildung 8 Multithreaded errechung

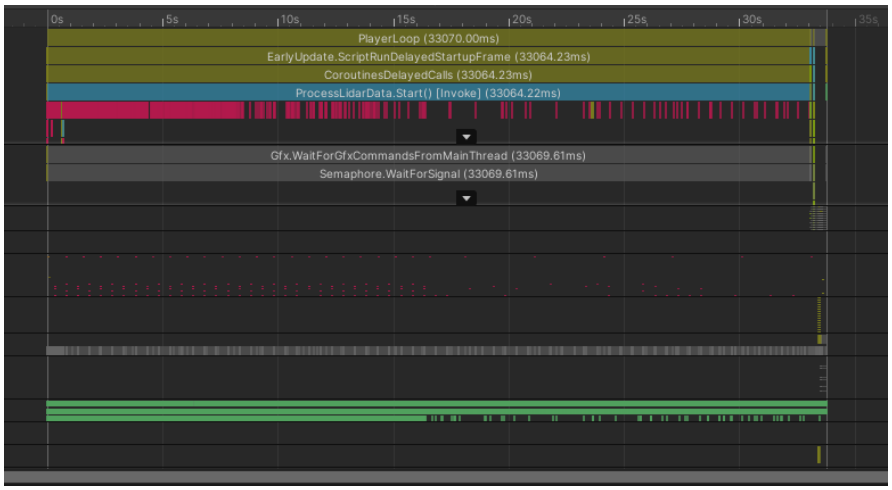


Abbildung 9 Ein Hintergrund Thread

Mit Threads wird nur dreimal so schnell gerechnet wie mit nur einem Thread. Dabei wird mein AMD Ryzen 5 6000 statt mit 5% nun zu 100% ausgelastet. Auf einem Ipad wäre der Multithread-Ansatz langsamer.

Was bisher beschrieben wurde, ist die erste Version von unserem Algorithmus. Sie funktioniert zuverlässig, wenn bestimmte Anforderung gegeben sind:

1. Wenn zu jedem Zeitpunkt drei oder mehr Ecken des Raumes von dem LIDAR aufgenommen werden.
2. Wenn Teile des Raums nicht durch bewegende Objekte wie Menschen verdeckt werden.
3. Wenn der Raum nicht spiegelbar ist.

4.2 Version 2 (Maarten)

Zu diesem Zeitpunkt in unserem Projekt wurde uns klar, dass wir auf schon vorhandene Forschung zurückgreifen müssen, um die zuvor genannten Probleme zu lösen. So kamen wir zu dem zuvor erwähnten Problem, dass die meisten Informationen in Form von wissenschaftlichen Abhandlungen vorhanden sind. Diese verstehen wir teilweise nicht. Trotzdem haben wir eine Abhandlung gefunden, welches einen relativ simplen, aber funktionsfähigen Algorithmus vorstellt.

Wir werden hier nicht den gesamten Inhalt der Abhandlung wieder geben, sondern wichtig ist in diesem Punkt nur, welche Ideen und Ansätze wir von diesem in Version 2 unseres Algorithmus übernommen haben. In der Abhandlung wurden alle Messpunkte in einer Liste in zweidimensionalen Texturen gespeichert. Die Texturen unterscheiden sich in ihrer Auflösung. Die Auflösungen gehen von der maximalen Genauigkeit (in unserem Fall 1 cm = 1 Pixel) zu der maximalen Verschiebung des LIDARs pro update (in unserem Fall 64 cm = 1 Pixel). Jede Textur hat die halbe Auflösung der vorherigen. Also gab es in diesem Beispiel Texturen mit 1, 2, 4, 8, 16, 32, und 64 cm pro Pixel.

Mit zwei Funktionen aus der Abhandlung kann man einen Vector durch differenziale Ableitung errechnen, der in die Richtung der nächsten Wand zeigt. Die einzige Vorgabe, ist, dass der Messpunkt nur ein Pixel von der Wand entfernt ist. Deshalb werden die verschiedenen Auflösungen benötigt. Wir fangen mit der gröbsten an und verschieben, bis wir die optimalste Position gefunden haben. Dann gehen wir zur nächst g enaueren Textur und wiederholen die Prozedur. Durch jede Textur wird die Position genauer, bis sie schließlich bis auf 1 cm der Wirklichkeit entspricht.

Der Algorithmus hat die Verschiebung sehr genau errechnet, aber wir hatten keine Ahnung, wie wir die Drehung errechnen sollten. In dem Ansatz des Papers wurde die Drehung als dritte Dimension behandelt und genauso optimiert, wie die beiden Karten-Dimensionen. Dies hat bei uns aber nicht funktioniert, da wir einen einfachen Ansatz zu Optimierung genutzt haben. So waren wir in einer Sackgasse. Deswegen haben wir nie eine

funktionierende Version mit diesem Ansatz gebaut, dennoch sind die Ansätze in dieser Abhandlung sehr vielversprechend und es ist unser Plan, sie in späteren Versionen soweit wir können zu nutzen und auf ihre Anwendbarkeit zu prüfen.

4.3 Version 3 (Tim)

In der dritten Version des LIDAR Mappings haben wir versucht, einen sehr simplen Weg zu gehen. Bei der zweiten Version hatten wir das Problem, dass wir die Drehung nicht korrekt errechnen konnten, deswegen haben wir als erstes daran gearbeitet, die Drehung zu errechnen und danach die Verschiebung.

Um dies zu erreichen, wird für jeden Winkel und die dazugehörigen Längen eine Abweichung errechnet, bis alle Winkel um 359 Grad verschoben wurden. Diese Daten werden jedes Mal abgespeichert. Danach wird die kleinste Abweichung rausgesucht und um den dazugehörigen Winkel hat sich der Roboter gedreht. Das funktioniert gut, solange man den LIDAR nicht mehrere Zentimeter verschiebt. Die Genauigkeit bei einer Drehung auf der Stelle liegt bei ±1 Winkel Grad.

Danach wird die Verschiebung ausgerechnet. Als erstes werden die Polar-Koordinaten in xy-Koordinaten umgewandelt. Nach dem die Karte richtig gedreht ist, müssten rein theoretisch die Deltas zwischen den Punkten mit dem gleichen Winkel die Verschiebung sein. Das kommt nicht immer hin, da es Punkte gibt, die nicht gefunden wurden oder hin und her springen. Im ersten Versuch wurde jeder Punkt mit jedem verglichen und dann das Delta in eine Liste gespeichert. Wenn genau das gleiche Delta wieder gefunden wird, wird eine Zählervariable hochgezählt. Die Zahl mit dem höchsten Wert ist die Verschiebung. Das hat leider nicht funktioniert, da auch Punkte verglichen wurden, die nichts miteinander zu tun haben oder ziemlich weit voneinander entfernt sind. Um dieses Problem zu lösen, teilen wir das Koordinatensystem in vier Bereiche auf. In diesen vier Bereichen wird dann wieder von jedem Punkt zu jedem anderen Punkt das Delta ausgerechnet und weggespeichert. Danach werden von allen vier Bereichen die Listen

zusammengesetzt und das Delta mit dem höchsten Wert ist die Verschiebung. Das funktioniert gut mit einer Genauigkeit von ±1 cm.

Umso weiter man den LIDAR vom Ursprung entfernt, um so ungenauer werden die Werte. Deswegen werden alle 3cm neue Referenzdaten genommen. Mit dieser Methode kann gut eine Karte erstellt werden. Die größten zwei Probleme sind zu einem der LIDAR selbst, da er nicht schnell genug Daten liefern kann, und zum anderen die Abweichungen, die entstehen, wenn der Abstand zur Startposition größer wird. Das resultiert darin, dass die gescannten Objekte oder Wände immer dicker werden.

5 Simulation (Yesenia)

Nach einer Weile hatten wir das Problem, dass der Roboter bei Tim steht und wir Änderungen nicht einfach testen konnten. Deshalb haben wir uns dazu entschieden, eine Simulation zu bauen, um weitere Änderungen testen zu können.

Ich hatte als einzige aus der Gruppe noch nicht mit Unity gearbeitet, und da man für die Simulation einigen Code neu schreiben musste, aber auch auf existierenden Code aus unserem Projekt zurückgreifen konnte, war es eine gute Aufgabe um sich mit C# und Unity vertraut zu machen.

In der Simulation kann man das 3D-Modell von Tim, der über Ray Casting auch einen LIDAR-Sensor simuliert, mit dem gleichen Protokoll wie den Roboter bewegen. Um das tun zu können, muss man die Nachrichten, welche die App sendet, auseinanderschneiden und Interpretieren. Dafür habe ich einen String-Interpreter geschrieben, der die Nachricht an den abgesprochenen Trennzeichen aufteilt.

Da ich noch keinen TCP-Server hatte, um mich tatsächlich mit der App zu verbinden, habe ich alles mit einem Textfeld in Unity ausprobiert. Nachdem die Bewegung und Drehung funktionierten, habe ich mich dazu entschieden, an dem Code zum Simulieren des LIDAR Sensors zu arbeiten.

Zum Simulieren des LIDAR Sensors habe wir wie oben erwähnt Ray Casting benutzt. Damit hatte ich vorher auch noch nie gearbeitet, es stellte sich aber als erstaunlich einfach heraus.

Das Schwierigste an der Simulation war es, den TCP-Server zum Laufen zu bekommen. Ich habe mich dazu entschieden, einen eigenen zu schreiben, obwohl wir schon einen funktionierenden hatten, da ich so viel wie möglich selbst machen wollte.

Als sich die App mit dem Server verbinden konnte, wurden die Nachrichten, die gesendet wurden, jedoch erst empfangen als man sie wieder vom Server getrennt hat. Dieses Problem habe ich dann doch noch mit Teilen des existierenden Server-Codes gelöst.

6 AI (Maarten)

Eine weitere Idee für unseren Roboter war die Entwicklung eines Algorithmus für autonomes Fahren. Man sollte Ziele auf der App markieren können und der Roboter würde zu diesen Positionen fahren. Problematisch war daran, dass unser SLAM Algorithmus nie genau genug ist, um die benötigte Position des Roboters bereitzustellen. Trotzdem haben wir eine funktionsfähige Version entwickelt, welche wir mit der Simulation erfolgreich testen konnten.

6.1 A* (Niklas)

Ich habe mich für A* entschieden, da es ein weit verbreiteter Pathfinding-Algorithmus ist und es gute Erklärungen gibt, wie dieser funktioniert. Um A* nutzen zu können und weil es allgemein das Nutzen unserer Daten einfacher macht, haben wir unsere Messdaten rasterisiert, das heißt wir speichern mit einer Auflösung von einem Zentimeter, ob ein Punkt eine Wand ist oder nicht. Man gibt diesem Algorithmus nun einen Start- und Endpunkt zusammen mit einer Liste der Wände und er findet einen Weg. Ein Problem, welches ich zu Beginn hatte, war das A* seine Daten in einer Liste gespeichert hat, was sehr ineffizient ist. Daher habe ich einen Heap implementiert, dies reduzierte die Laufzeit drastisch.

6.2 Was noch zu tun ist (Niklas)

Ein Ziel, welches für uns noch offengeblieben ist, wäre einen Algorithmus zu schreiben, welcher autonom einen Raum erkundet und vollständig kartiert. Uns ist allerdings noch keine zufriedenstellende Möglichkeit eingefallen, Orte zu erkennen welche nicht ausreichend erkundet wurden.

7 Fazit

7.1 Maarten

Im Allgemeinen bin ich mit der Zusammenarbeit unseres Teams sehr zufrieden. Alle Mitglieder haben sich interessiert in die Entscheidungsprozesse eingebracht. Alle haben die ihnen zugeteilten Arbeitspakete in den ausgemachten Zeitrahmen erarbeitet.

Als größte Herausforderung stellte sich die Aufteilung der Aufgaben auf mehrere Personen heraus. Die meisten Bestandteile der App hatten viele Abhängigkeiten untereinander, was zu vielen Problemen geführt hat, wenn nicht eine einzelne Person an ihnen gearbeitet hat. Wir haben die schlimmsten Abhängigkeitsprobleme gelöst, indem wir uns auf ein Allgemeines Abstraktionssystem mit loser Kopplung geeinigt haben. So konnte man z.B. den TCP-Client testen, ohne den Rest der App starten zu müssen. Dies hat modulares Arbeiten ermöglicht und das Unit-Testen von einzelnen Bestandteilen der App ermöglicht. Trotzdem mussten wir oft auf eine Person warten, um weiter arbeiten zu können.

Das Klima war größtenteils positiv. Den einzigen Kritikpunkt, den ich an mich selbst habe, ist, dass ich teilweise zu große Aufgabenteile des Projektes an mich gerissen habe. Dies hatte zur Folge, dass sich teilweise Gruppenmitglieder nicht richtig integriert gefühlt haben.

Für das nächste Projekt habe ich gelernt, noch klarer mit den Gruppenmitgliedern die Aufgaben aufzuteilen und vorher zu klären, wo die verschiedenen Leute motiviert sind und wo sie eher weniger Lust draufhaben. Außerdem würde ich mich mit der Gruppe für Testläufe persönlich treffen. Dies war durch Corona nicht möglich, aber hätte uns einiges an Zeit erspart.

7.2 Niklas

Meine Aufgabe in diesem Projekt waren das Networking, die AI und ein wenig UI zu implementieren. Zu allen diesen Themen gab es viele Beispiele im Internet, die den Prozess relativ einfach für mich gemacht haben. So habe ich leicht noch ein paar Dinge über Unity lernen können, und habe auch eine neue Datenstruktur kennengelernt, den Heap. Der größte Unterschied zu vorherigen Projekten war, dass dieses tatsächlich von anderen Menschen benutzt werden soll. Was bedeutet das wir nicht mit einem „Proof of Concept“ aufgehört haben. Das größte Problem war das wir alle an stark verknüpften Teilen des Projektes gearbeitet haben, und nur Tim tatsächlich dinge mit dem Roboter ausprobieren konnte. Wenn ich ein ähnliches Projekt noch einmal mache, würde ich die Simulation früher schreiben.

7.3 Tim

Insgesamt hat mir das Projekt sehr viel Spaß gemacht und auch die Teamarbeit hat gut funktioniert. Da ich sehr viel am Roboter gebaut habe und mich am meisten mit der Hardware beschäftigt habe, könnte ich sehr viel aus diesem Bereich mitnehmen wie zum Beispiel welche Motoren für welche Aufgabe gut geeignet sind oder wie Stepper-Motoren funktionieren. Des Weiteren habe ich mich mit dem Raspberry Pi auseinandergesetzt, wofür ich ein C++ Programm geschrieben habe. Obwohl ich mich schon mit C++ auskannte, habe ich sehr viel Neues über diese Programmiersprache gelernt. Besonders habe ich viel über TCP Server und serielle Schnittstellen erfahren.

Dass ich sofort angefangen habe zu arbeiten und nicht so wie bei anderen Projekten erst kurz vor Schluss, liegt daran, dass dieses Projekt mir sehr viel Spaß gemacht hat und sich auch mit meinen privaten Interessen überschneidet.

Was wir noch besser hätten machen könnte wäre die Aufgaben besser aufzuteilen und sich zu treffen, um Testläufe durchzuführen, dies ging wegen Corona leider nicht.

7.4 Yesenia

Eigentlich fand ich das Projekt und die Projekt-Idee sehr gut. Ich habe auch viel neues gelernt. Allerdings fand ich es gerade mit Corona schwierig in einer Gruppe zu arbeiten. Gerade die Kommunikation über Distanz fällt schwer, deshalb wurde die anfängliche Aufteilung von einigen nicht wirklich eingehalten. Dies führte zu einigen Problemen.

Im nächsten Projekt würde ich früher und besser kommunizieren, woran ich eigentlich arbeite.

8 Glossar

Begriff:	Erklärung:
TCP	Transmission Control Protocol: Protokoll zum Austausch von Daten zwischen Computern.
HTTP	Hypertext Transfer Protocol: Hauptsächlich eingesetzt, um Webseiten aus dem World Wide Web in einen Webbrowser zu laden.
Heap	Eine Datenstruktur, um daten mit einer festen Reinfolge schnell hinzufügen und entfernen zu können.
Thread	Ein Ausführungsstrang oder eine Ausführungsreihenfolge in der Abarbeitung eines Programms.
A*	Ist ein Algorithmus, um einen kurzen Weg zwischen zwei Punkten zu finden.

9 Abbildungen

10 Literaturverzeichnis

Webnachweis
Unity Forums: https://forum.unity.com/threads/how-do-i-detect-when-a-button-is-being-pressed-held-on-eventtype.352368/ 28.01.2021
danielbierwirth (2018): TCP Client-Server Connection Example: https://gist.github.com/danielbier-wirth/0636650b005834204cb19ef5ae6ccedb 28.01.2021
Waldo: Pinch to Zoom and Panning on Mobile in Unity: https://www.youtube.com/watch?v=0G4vcH9N0gc 09.02.2021
Waldo: Pan & Zoom: How to Pan and Zoom on Touch Devices in Unity: https://pressstart.vip/tutorials/2018/07/12/44/pan--zoom.html 09.02.2021
Sebastian Lague: A* Pathfinding Tutorial (Unity): https://www.youtube.com/watch?v=-L-WgKMFuhE&list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW 10.02.2021
Sebastian Lague: Pathfinding: Episode 04 - heap: https://github.com/SebLague/Path-finding/tree/master/Episode%2004%20-%20heap 10.02.2021
Waldo: Mobile Joystick in Unity: How to create your own touch screen virtual joystick: https://pressstart.vip/tutorials/2018/06/22/39/mobile-joystick-in-unity.html 12.02.2021
[1] NetBUG: RPLidar by RoboPeak: https://www.thingiverse.com/thing:1486705 28.03.2021
[2] Pakkko: 40mm Fan Cover Honeycomb: https://www.thingiverse.com/thing:4827203 05.04.2021
Linux Serial Ports Using C/C++: https://blog.mbedded.ninja/programming/operating-systems/linux/linux-serial-ports-using-c-cpp/ 20.05.2021
Multiple Client Handling TCP Server by rajabishek: https://gist.github.com/ra-jabishek/7102fe2b7b60b4833d72 05.04.2021
TCP Client-Server Tutorial: https://www.youtube.com/watch?v=4uHTSknGJaY 05.04.2021
Unity Documentation Physics.Raycast: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html: 28.03.2021
Google Bildersuche: Inspiration Design: 20.04.2021
Abbildung1: https://imgaz2.staticbg.com/thumb/large/oaupload/banggood/images/FF/96/b5489451-dfc8-404f-8041-cf88be3d21ab.jpg.webp 01.06.2021

11 Externe Mittel

- Unity 2020.3f1
- Shader Graph (Unity Erweiterung)
- Cinemachine (Unity Erweiterung)
- TextMeshPro (Unity Erweiterung)
- Rider IDE
- VS Code
- Visual Studio
- MySQL Workbench
- GitHub
- Photoshop
- Vectornator
- Gimp 2020

12 Anhang

Siehe nachfolgende Seiten.

12.1 Eigenständigkeit Erklärung

Hiermit versichern wir, dass alle abgegebenen oder präsentierten Inhalt von uns selbstständig erarbeitet wurden. Ausgenommen von diesem sind alle angegebenen Literaturnachweise und externe Mittel.

Name:	Datum:	Unterschrift:
Niklas Otten		
Maarten Behn		
Tim Jaeschke		
Yesenia Möhring		

12.2 Projektanmeldung

12.3 Protokoll zur Konsultationssitzung 1

12.4 Protokoll zur Konsultationssitzung 2

12.5 PAP

12.6 CD

Beispiel code und anhang dokumente

2011_SSRR_KohlbrecherMeyerStrykKlingauf_Flexible_SLAM_System.pdf