Fakultät Informatik/Mathematik

Projektstudium SS/WS 18

Verbesserter Aufbau eines **A**utonomen **L**aser **F**ahrzeugs (ALF)

---

eingereicht von:	Bierschneider Christian,	3118760
	Beck Dennis,	1234567
	Grauvogl Stefan,	1234567
	Studiengang: Informatik	
	Schwerpunkt: Technische Informatik	
betreut durch:	Prof. Dr. Alexander Metzner	
	OTH Regensburg	

Regensburg, 26. November 2018

Abstract

Here is space for a fancy short summary.

Inhaltsverzeichnis

Kapitel 1

1 Einführung

Kapitel 2

2 Projektanforderungen

Kapitel 3

3 Neuaufbau des Vorgängerprojekts

3.1 Probleme an der Hardware

3.2 Probleme an der Software

3.3 Aktueller Aufbau des Autonomen Laser Fahrzeugs (Alf)

Kapitel 4

4 Verwendetes Betriebssystem

In diesem Kapitel wird die Installation des Betriebssystem Ubuntu Mate, sowie die Einrichtung des Frameworks ROS näher erläutert. Außerdem werden alle nötigen Konfigurationen gezeigt, um die Software kompilieren und ausführen zu können.

4.1 Installation von Ubuntu Mate auf Raspberry Pi 3b+

Aufgrund einer sehr großen Ubuntu Community und die gute Anbindung an das Framework ROS, wurde sich für das Betriebssystem Ubuntu Mate entschieden. Da keine durchgängige Echtzeitanbindung gefordert ist, werden hier der SLAM sowie die Wegefindung berechnet und ausgeführt. Die Sensoren der Motorsteuerung agieren dagegen auf einem STM Board um schneller auf Änderungen reagieren zu können. Da zum aktuellen Zeitpunkt (25.10.2018) kein angepasstes Betriebssystem für den Raspberry Pi 3b+ zur Verfügung steht, mussten kleinere Konfigurationen stattfinden um das vorhandene Raspberry Pi 3 Image auf einem Rasperry Pi 3b+ zum laufen zu bekommen. Problem ohne diese Einstellungen: Der Raspberry Pi 3b+ startet nicht und es wird nur ein Regenbogen Bildschirm angezeigt!

Nachfolgend werden alle Konfigurationen genauer erläutert.

1. Download eines Images für Raspberry Pi 2/3 auf folgender Seite:

2. Flashen des Images auf einer SD-Karte mit Win32DiskImages oder (linux):

3. Diese SD-Karte in einen **Raspberry Pi 2** oder **Raspberry Pi 3** einstecken und starten.

4. Danach ein Terminal öffnen und folgenden Befehl für ein Kernel Update eingeben:

```
1 $ sudo CURL_CA_BUNDLE=/etc/ssl/certs/ca-certificates.crt rpi-update
```

oder alternativ:

```
$ sudo BRANCH=stable rpi-update
```

5. Raspberry Pi 2/3 herunterfahren und diese SD-Karte in den Raspberry Pi 3b+ einstecken. Der Pi sollte dann wie gewünscht booten.

6. Zu diesem Zeitpunkt ist aber noch keine Wlan-Verbindung verfügbar. Installiere das neueste Raspbian Image von hier auf eine weitere SD-Karte.

7. Starte einen Raspberry Pi mit dem Raspbian Betriebssystem und kopiere folgenden Ordner auf einen USB-Stick.

```
$ sudo cp -r /lib/firmware/brcm /path_to_usb
```

8. Starte den Raspberry Pi 3b+ mit der SD-Karte auf der Ubuntu Mate installiert ist.

9. Ersetze den aktuellen /lib/firmware/brcm Ordner durch den am USB-Stick

```
$ sudo cp -r /path_to_usb/lib/firmware/brcm /lib/firmware/brcm
```

10. Führe einen Neustart durch und eine Wlan-Verbindung sollte verfügbar sein.

11. Aktiviere ssh durch folgenden Befehl

```
$ sudo systemctl enable ssh
```

4.2 Installation des Frameworks Robot Operating System (ROS)

Nachfolgend wird Installation des Frameworks ROS durchgeführt. Dazu auf dem Betriebssystem Ubuntu Mate ein Terminal öffnen und folgende Befehle eingeben:

1. Einrichten der sources.list

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
    lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

2. Einrichten der keys

```
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

3. Update der Packages

```
$ sudo apt-get update
```

4. Install ros-kinetic-desktop-full

```
$ sudo apt-get install ros-kinetic-desktop-full
```

5. Initialisierung und Update der Rosdep

```
$ sudo rosdep init
```

```
$ rosdep update
```

6. Einrichten der ROS Umgebungsvariablen

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

Neues Terminal öffnen oder nachfolgenden Befehl eingeben:

```
$ source ~/.bashrc
```

7. Erstellen eines catkin workspaces

```
$ mkdir -p ~/catkin_ws/src
```

```
$ cd ~/catkin_ws/
```

```
$ catkin_make
```

```
$ source ~/catkin_ws/devel/setup.bash
```

Durch nachfolgenden Befehl hat man von überall im Linux System Zugriff auf die im catkin workspace gebildeten Packages:

4.3 Schreibzugriff auf den Hokuyo Port für den aktuellen Benutzer

Um nicht Root Rechte besitzen zu müssen um auf den Hokuyo Port zugreifen zu können, wurde der akuelle Benutzer in die Dialout Gruppe mit folgendem Befehl hinzugefügt:

```
$ sudo adduser "user_name" dialout
```

4.4 Roscore Master und Launch file als Systemd Service

Da sich die SLAM Map sofort nach dem Start aufbauen soll, wurde sich für systemd services entschieden. Diese werden beim hochfahren des Raspberry Pi 3b+ ausgeführt, außerdem kann der jeweilige Service auch nachträglich per Kommando gestartet und gestoppt werden. Da der roscore Master benötigt wird, um innerhalb des ROS Frameworks zu kommunizieren, wurde dieser als einzelner Service entworfen. Hierfür muss unter /etc/systemd/system eine Datei roscore.service mit folgendem Inhalt erstellt werden.

```
[Unit]
Description=starts roscore master as a systemd service

[Service]
Type=simple
ExecStart=/bin/bash -c "source /opt/ros/kinetic/setup.bash; /usr/bin/
    python /opt/ros/kinetic/bin/roscore"

[Install]
WantedBy=multi-user.target
```

Um das ROS launch File hokuyo_hector_slam.launch als Service auszuführen wurde eine Datei hector.service in /etc/systemd/service mit folgendem Inhalt erstellt.

```
[Unit]
Description=starts hokuyo_hector_slam launch file as a systemd service

[Service]
Type=simple
ExecStart=/bin/bash -c "source /opt/ros/kinetic/setup.bash; /usr/bin/
    python /opt/ros/kinetic/bin/roscore"
ExecStop=
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Kapitel 5

5 Hardware

5.1 Lidar

Als Laserscanner wird ein Hokuyo URG-04LX verwendet.

Dieser hat eine Auflösung von 360° / 1024 pro Step. Insgesamt kann ein Winkel von 240° (= 768 Datenpunkte) je Scan aufgezeichnet werden.

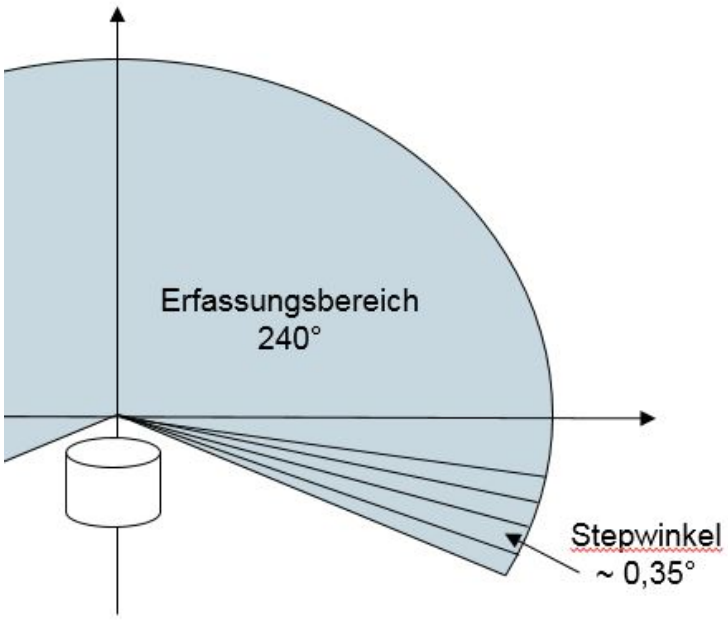

Abbildung 5.1: Lidar Uebersicht

Die Daten werden für jeden erfassten Punkt jeweils im Abstand von 0,35° als Entfernung geliefert. Die Punkte sind somit in Polarkoordinaten Darstellung vorhanden.

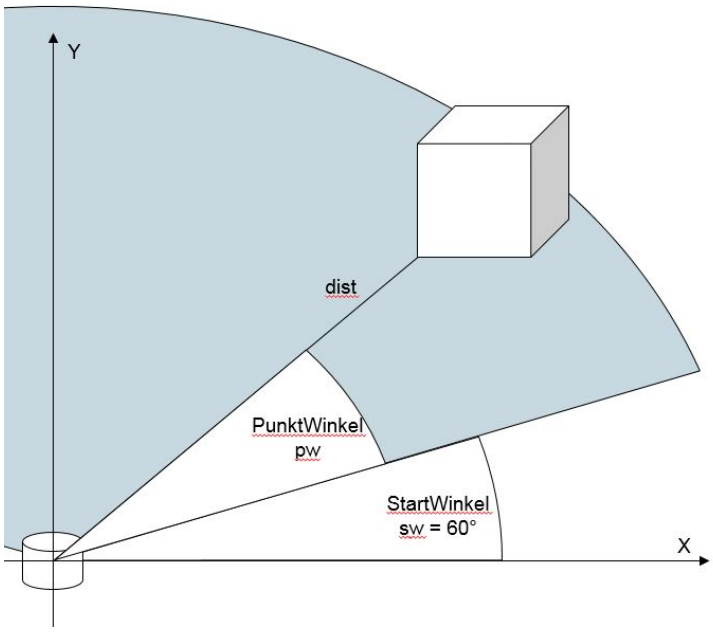

Abbildung 5.2: Lidar Uebersicht

5.2 Interface Board

Das ALF verfügt über folgende zusätzliche Komponenten:

- 1 Fahrmotor
- 1 Servo für die Lenkung
- 3 Ultraschallsensoren (verbunden über I^2C)
- 1 kombinierter Beschleunigungssensor und Gyroskop (IMU MPU6050, verbunden über I^2C)

Um diese Komponenten in einer deterministischen Zeit kontrollieren und auslesen zu können, wurde neben dem Raspberry Pi 3B+ Board ein zweites Hardwaremodul verwendet. Es handelt sich hierbei um ein *STM32 F334R8 Nucleo Board* der Firma STMicroelectronics [1]. Es handlet sich hierbei um einen ARM Prozessor auf einem Arduino kompatiblen Evaluationsboard. Der ARM Cortex M4 kann für beliebige Mess- und Regelungsaufgaben programmiert werden. Da das verwendete Raspberry Pi Betriebssystem kein Echtzeitbetriebssystem ist, können die notwendigen Echtzeitaufgaben damit auf den ARM Controller ausgelagert werden.

[1] www.st.com

Kapitel 6

6 Software

Nachfolgend wird das Design dieses Softwareprojekts dargestellt, sowie die leichte Erweiterung durch nachträgliche Module. Anschließend wird noch auf die bereits umgesetzten Module eingegangen und ihre Funktionsweise erklärt.

6.1 Design des Projektes mit CMake

6.2 Verbindung zum Lidar

Der Lidar Sensor ist direkt via USB mit dem Raspberry Pi verbunden. Die Daten werden vom Lidar in Polarkoordinaten Darstellung geliefert. Um eine Karte aufbauen zu können, wurde ein Modul erstellt, das die Daten in kartesische Koordinaten transformiert. Somit ist es möglich nach jeder Messung des Lidars eine neue Karte mit der äktuellen Sichtdes Sensors zu erstellen. Die Daten werden in einem Integer-Array gespeichert und können von einem SLAM Algorithmus verarbeitet werden .

Das Modul verwendet zur Verbindung mit dem Lidar die unter der GNU GPL v3 stehende API ÜRG04LX-. Mit ihr ist es möglich einen kompletten Scan des Lidars aufzuzeichnen.

```
int data[MEASUREMENT_POINTS];
int measuredPoints;
URG04LX laser;

laser = URG04LX('dev/tty/ACM0')

measuredPoints = laser.getScan(data);
```

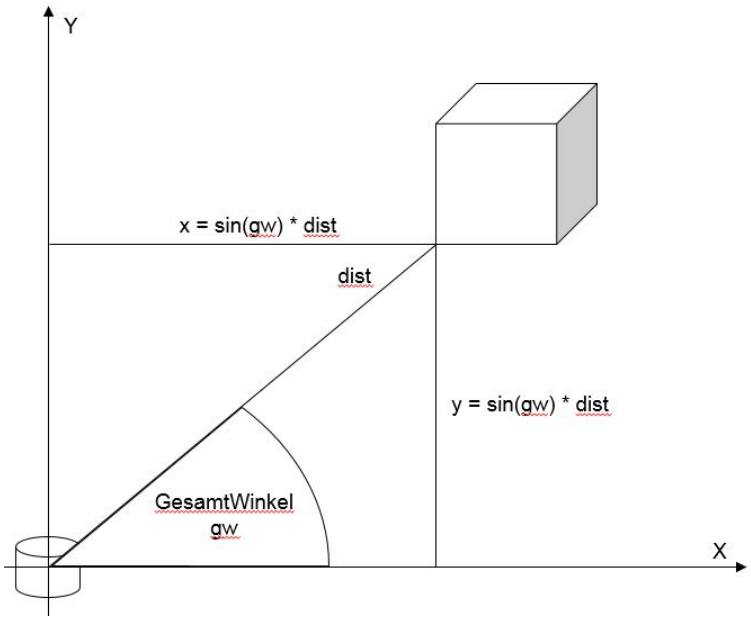

Abbildung 6.1: Koordinaten errechnen

Die Datenpunkte werden in polarkoordinaten Darstellung geliefert und müssen zur Weiterverarbeitung in kartesische Koordinaten umgerechnet werden.

Die Umrechnung muss für alle aufgezeichneten Datenpunkte des Scans vorgenommen werden und liefert dann die Sicht des Lidars in einem Koordinatensystem:

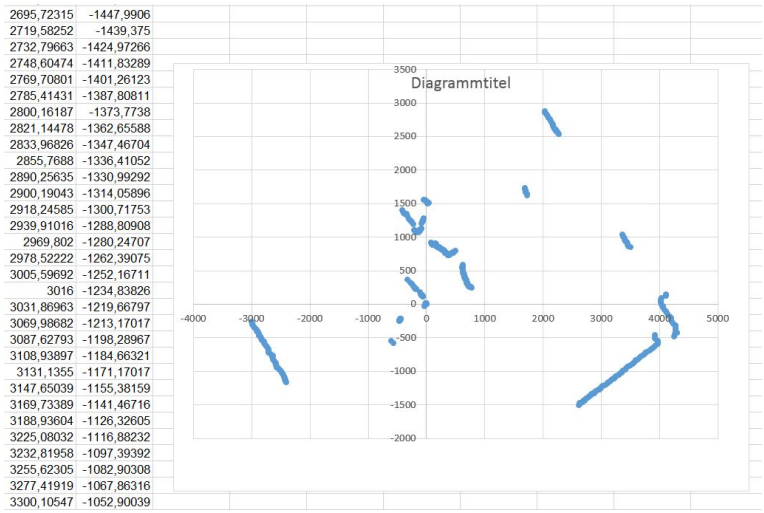

Abbildung 6.2: Koordinaten Veranschaulichung

Das Modul wird momentan nicht verwendet, da der SLAM Algorithmus mit Hilfe von ROS realisiert wurde und die entsprechende Library einen eigenen Connector zum Lidar bereitstellt. Um folgenden Gruppen jedoch die Arbeit zu erleichtern, wurde das Modul trotzdem vorbereitet.

6.3 SLAM

6.4 Wegefindung

Vorüberlegung: Das definierte Ziel ALF autonom einen Raum erkunden zu lassen, beinhaltet neben dem Erstellen einer Karte auch eine Wegberechnung zu unbekannten Flächen im Raum, die noch nicht vom Lidar erfasst wurden. Somit muss basierend auf der vom SLAM erstellten Karte ein Pfad zu den unbekannten Flächen gefunden werden. Dabei muss berücksichtigt werden, dass ALF durch seine Lenkung einen eingeschränkten Aktionsradius hat und es nicht möglich ist aus einer Geradeaus-Fahrt sofort nach Rechts oder links abzubiegen. somit ist es nicht möglich direkt an einer Wand entlang üms Eckßu fahren. Der Lenkwinkel muss in die Routenplanung mit einbezogen werden.

Eingabe: Karte als Matrix Egoposition auf Karte

1) Übergabe Daten von SLAM

Die vom SLAM erhaltenen Daten entsprechen der einer PGM-Datei. In einem 2-dimensionalen Integer-Array wird die erstellte Karte als Grauwerte übergeben. Mögliche Werte sind erkanntes Objekt (Schwarz), Unbekannte Fläche (Grau), Freifläche(Weiß).

Beispiel:

```
[ 127 127 127 127 127 127 127 ... ]
[ 127 127 127 127 127 127 127 ... ]
[ 127   0   0   0   0 127 127 ... ]
[ 127   0 255 255 255 127 127 ... ]
[ 127   0 255 255 255 127 127 ... ]
[ 127   0 255 255 255 127 127 ... ]
[ 127   0 255 255 255 127 127 ... ]

0 = erkanntes Objekt
127 = unbekannte Flaeche
255 = Freiflaeche
```

2) Whiten

Die vom SLAM erhaltene Karte kann unter Umständen nicht nur Schwarze, Weiße und fest definierten graue Punkte enthalten. Je nach verwendetem SLAM werden für Messpunkte, die nicht sicher als Objekt oder Freifläche definiert werden können, als Zwischengrauwert angegeben. Dies führt jedoch bei der weiteren Berechnung des

Pfades zu Problemen. Daher wurde eine Grenze definiert, unter der alle Punkte als Objekt und über der alle als Freifläche angesehen werden. Somit kann im weiteren Programm von sauberen Werten (Objekt, Freifläche, Unbekannt) ausgegangen werden.

¡ Bild vor/nach whiten ¿

3) Gradientenfüllung

Zur Realisierung der in der Einleitung genannten Lenkwinkel-Problematik, wurde die Karte mit selbst definierten Grauwerten eingefärbt. Je weiter das Fahrzeug von einem Gegenstand bzw. einer Wand entfernt ist, desto unkritischer wird die Navigation mit dem Lenkwinkel. Freiflächen der Karte, die nahe an einer Wand liegen, sollen nach Möglichkeit gemieden werden. Punkte, die in der Mitte eines Raumes ohne Gegenstände liegen, werden als positiv für die Routenplanung angesehen. Somit sollte der Pfad immer zuerst in die Mitte eines Raumes führen und sich erst am Zielpunkt wieder einer Wand nähern. Umgesetzt wurde dies mit einer Grau-Gradientenfüllung der Karte, die später bei der Pfadberechnung als Gewichtung dienen. Die Freiflächenpunkte nahe einer Wand wurden mit einem hohen Gewicht (repräsentiert durch "dunkelgrau") belegt und verringern ihr Gewicht, je weiter sie von einer Wand entfernt liegen.

¡Bild graymapped¿

4) in Graphen wandeln

Um auf der bestehenden Karte einen Pfad berechnen zu können, muss die Matrix in einen Graphen überführt werden. Die einfachste (wenn auch nicht die performanteste) Methode war es jeden Freiflächen-Messpunkt des Lidars als eigenen Knoten anzusehen, der eine Verbindung zu den jeweilig benachbarten Messpunkten/Knoten hat. Als Kantengewicht wurde der entsprechende Grauwert des Nachbarknoten gewählt. Somit werden Pfade auf Freiflächen belohnt (Kantengewicht = 0) und Annäherungen an Gegenstände bestraft (Kantengewicht = steigender Grauwert). Für unbekannte Flächen sowie erkannte Objekte wurden keine Knoten in den Graphen eingefügt und diese auch nicht als Nachbarn angesehen.

Dass aus jedem Pixel ein eigener Knoten wird, hat zur Folge, dass es extrem viele mögliche Pfade zu berechnen gibt. Hier existiert noch ein mögliches Verbesserungspotential für weitere Gruppenarbeiten. Da es sich jedoch um einen ersten autonomen Prototypen handelt, reicht die Umsetzung auf diesem Wege aus.

5) mögliche Ziele definieren und finden

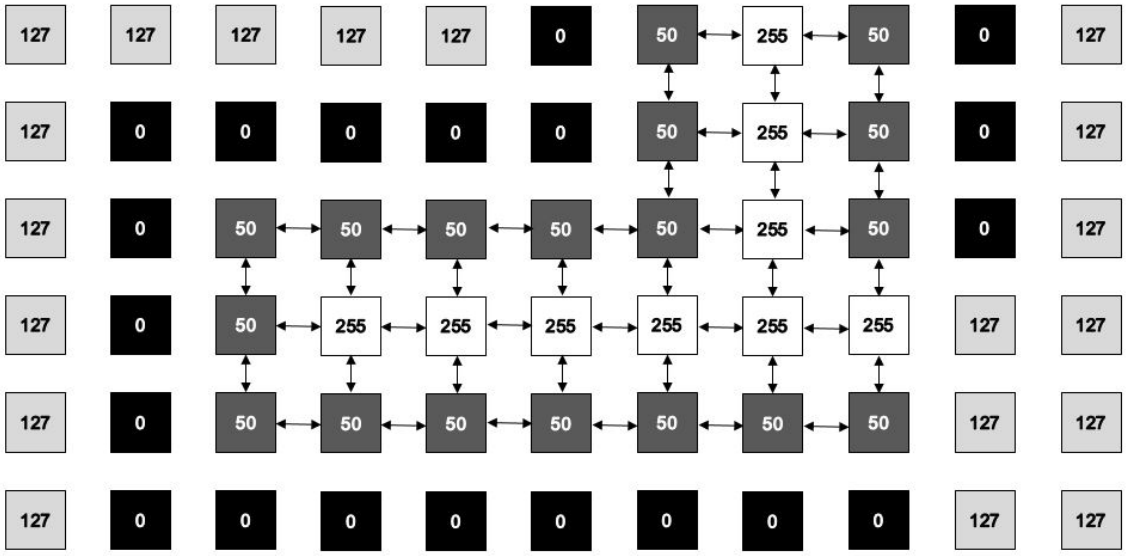

Abbildung 6.3: aus Map erstellter Graph

Als Voraussetzung wird immer angenommen, dass die aktuelle Position des Fahrzeugs auf einer erkannten Freifläche liegt. Das übergeordnete Ziel einen Raum vollständig autonom zu erkunden, lässt sich nur erreichen, indem das Fahrzeug nicht zufällig durch den Raum fährt, sondern gezielt unbekannte Flächen ansteuert. Mögliche Ziele sind somit alle Übergänge von Freifläche zu unbekannter Fläche.

Probleme: Es kann passieren, dass der Lidar Sensor durch Reflektionen spiegelnder Oberflächen fehlerhafte Werte liefert. Somit entsteht bei der Verarbeitung der Daten mit dem SLAM der Eindruck, dass eine Freifläche hinter einer Wand erkannt wurde. Da es auch dort zu Übergängen zwischen Freifläche und Unbekanntem Bereich kommen kann, werden diese Punkte auch als mögliche, zu erkundende Ziele erkannt. Da es jedoch keinen Weg zu diesen separierten Freiflächen gibt, ist es nicht möglich einen Pfad zu berechnen. Da sich dies als großes, nur sehr schwierig zu lösendes Problem herausstellte, wurde als Workaround die Pfadsuche so implementiert, dass alle Ziele durchgetestet werden und die Pfadberechnung nur abgeschlossen ist, wenn ein gültiger Pfad gefunden werden konnte.

¡Evtl Bild von allen unbekannten Übergängen¿

6) Dijkstra

Der erstellte Graph kann nun mit Hilfe eines Dijkstra-Algorithmus den kürzesten Weg von der Egoposition zum einem der möglichen, erkannten Ziele errechnen. Die in 3) eingeführte Gewichtung der Kanten führt nun dazu, dass ein Weg z.B. in der Mitte eines Ganges entlang errechnet wird. Bei 90° Winkeln wird eine leichte Biegung

errechnet. Die Kantengewichtung ist auf dem kürzeren Pfad zwar schlechter, jedoch ist der Weg kürzer. Somit wird auch die Lenkwinkelproblematik entschärft.

Als Dijkstra-Implementierung wurde die Veröffentlichung von Mahmut Bulut als Grundlage verwendet und für das Projekt angepasst.

// https://gist.github.com/vertexclique/7410577

7) ersteller Pfad

Als Ergebnis des Dijkstra-Algorithmus wird ein Pfad von der Egoposition über die Freiflächen bis hin zu einer zufälligen, unbekannten Fläche erzeugt. Die Navigation des Fahrzeuges übernimmt das Bewegungssteuerungsmodul. Sobald das Ziel erreicht wurde, kann ein neuer Pfad zu den noch verbleibenden, unbekannten Flächen erzeugt werden.

allgemeine Laufzeitoptimierung -¿ Blocks

Der verwendete SLAM liefert eine Auflösung von XXXX m / Pixel. Zur Berechnung eines Pfades ist diese Auflösung jedoch zu detailiert, sodass die gesamte Karte in Blocks mit Freiflächen eingeteilt werden kann. Um die Laufzeit des Dijkstra zu verringern, wurde nicht jeder Pixel als Knoten angesehen, sondern ein Block von z.B. 5x5 Pixeln als 1 Knoten. Dies verringert zwar die Genauigkeit des Pfades, durch die Lenkwinkelproblematik wird diese jedoch sowieso nicht benötigt.

Ausgabe: Pfad von Egoposition zu Freifläche

6.5 Bewegungssteuerung

Here i quote from a particular book: This quote is from one special book that i have to specify in the end. There are even two books, just so you see that [**?**, **?**]

Legislative texts, for example, need not to be quoted. Here is enough a reference in the footnote [1]

[1] This is the super footnote, here is something of BGB §§12 Abs. 3 Satz 4

Kapitel 7

7 Zusammenfassung und Ausblick

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang