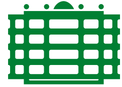

TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Elektrotechnik und Informationstechnik

Professur für Prozessautomatisierung

Modul: Projektpraktikum Autonome Systeme

TUCar
Technische Dokumentation

1 Allgemein

Das TUCar (siehe Abbildung 1.1) ist im Rahmen einer gemeinsamen Bachelorarbeit (Leopold Mauersberger und Daniel Käppler, 2018) entstanden und ist konzipiert für die Nutzung in der Lehre. Ergänzend zu unserem Praktikumsroboter TUCBot (Lange u. a., 2018), liegt der Fokus beim TUCar verstärkt auf höheren Navigationsalgorithmen. Entsprechend ist das Fahrzeug u.a. mit einer Kamera und einem Embedded PC System ausgestattet.

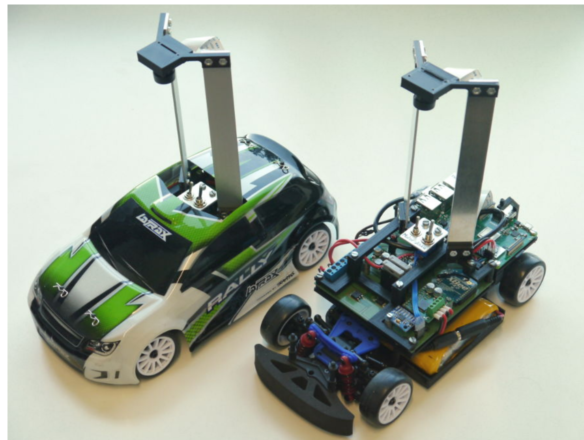

Abbildung 1.1: Ansicht des TUCar mit und ohne Hülle.

Technische Daten	
Abmessungen	26 × 13 × 24 cm (LxBxH)
Radstand	ca. 167 mm
Chassis	Modellauto 1:18
Embedded PC	Raspberry Pi 3B+
Mikrocontroller Board	Arduino Micro
IMU	6 DoF (Beschleunigungs- und Drehratensensor)
Kamera	200° Öffunungswinkel
Inkrementalgeber	Auf Motorachse zur Geschwindigkeitsmessung

1.1 Koordinatensysteme

Nachfolgend werden die Koordinatensystemdefinitionen für das TUCar eingeführt:

- Die **Welt** $\mathcal{W}$ befindet sich in einer Ecke des Spielfelds und bildet das Referenzkoordinatensystem.
- Das Vehiclekoordinatensystem oder **Body** $\mathcal{B}$ liegt in der Mitte der Hinterachse des TUCar.
- Das **Kamera**-Koordinatensystem $\mathcal{C}$ liegt im optischen Zentrum der Kamera.
- Die Koordinatensysteme für die **Tags** $\mathcal{T}_i$ befinden sich jeweils in der Mitte eines Tags (siehe 2.1 TagMap).

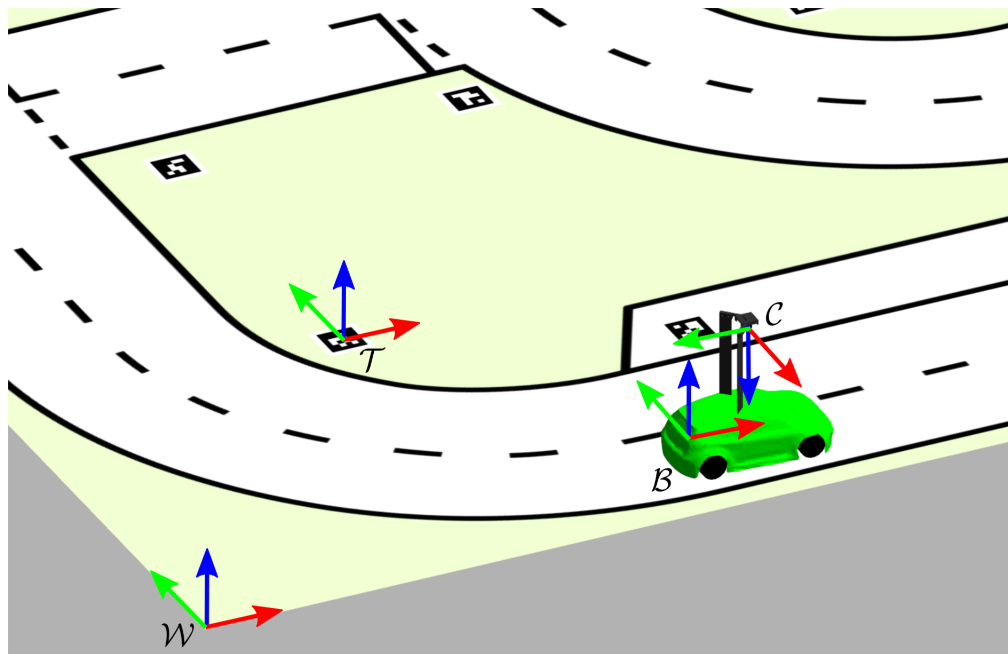

Abbildung 1.2: Darstellung der Koordinatensysteme Welt $\mathcal{W}$, Body $\mathcal{B}$, Camera $\mathcal{C}$ und Tag $\mathcal{T}$.

Die Transformation von Punkten aus dem Kamera-Koordinatensystem in das Body-Koordinatensystem ist über eine extrinsische Kalibrierung gegeben und erfolgt über die Transformation

$$\mathbf{T}^{\mathcal{BC}} = \begin{bmatrix} 0 & -1 & 0 & 0.085 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.23 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

Auch wenn die Transformation dreidimensional formuliert ist, wird die Z-Komponente meist vernachlässigt und davon ausgegangen, das sich alles in der x-y-Ebene befindet.

2 TUCarToolbox

Die TUCar-Toolbox wurde entwickelt, um grundlegende Funktionen für das TUCar in Matlab und Matlab Simulink zur Verfügung zu stellen. Es beinhaltet ebenso die Anbindung an ROS und damit die Möglichkeit, Steuerbefehle an das TUCar zu senden bzw. Sensordaten zu empfangen.

Nachfolgend werden einige Komponenten der Toolbox beschrieben.

2.1 TagMap

Die TagMap enthält die Posen aller Tags im Welt-Koordinatensystem (2D). Die Daten sind in der Datei `tagmap.json` gespeichert. Diese kann in Matlab mit der Funktion

```
loadTagMap(filename, useGtsam)
```

geladen werden und gibt ein Struct mit allen Tags zurück. Mit Setzen des Arguments `useGtsam =true`, wird die Tag-Pose zusätzlich als GtSAM-Pose Klasse (`gtsam.Pose2`) zurückgegeben.

2.2 StreetMap

Die Klasse StreetMap wurde zur einfacheren verwendung der Straßenkarte angelegt. Diese besteht aus der linken ($LeftLane$) und rechten (*RightLane*) Fahrbahnmarkierung und der Mittellinie (*MidLane*). Die Linien bestehen jeweils aus einem Array mit Punkten in der xy-Ebene.

Folgende Funktionen sehen zur Verfügung:

```
getParallelPath(points, scale)
plotMap(ax)
```

2.3 TUCarLocalisation

TUCarLocalisation ist eine MATLAB Klasse, die eine Lokalisierungslösung mit hilfe eines EKFs aus den positionen der Apriltags und der Odometrie berechnet.

2.3.1 Input

- **Odometrie**(geometry_msgs/TwistStamped): vom TUCar bereitgestellte Odometrie aus Encoder und IMU im Body-Koordinatensystem.
- **AprilTags Positionen**(apriltags2_ros/AprilTagDetectionArray): Positionen der April-Tags in Kamera-Koordinatensystem.

2.3.2 Output

- **Pose**(nav_msgs/Odometry): EKF Lösung für die Pose im Welt-Koordinatensystem (und Geschwindikeit und Drehrate unverändert aus Input im Body-Koordinatensystem).

2.3.3 Verwendung

```
TUCarLocalisation(Odometrietopic, Apriltagtopic, Outputtopic, Worldname,
    Plotlevel)
```

Parameter	Erklärung
Odometrietopic	ROS-Topicname für Odometrieinformationen
Apriltagtopic	ROS-Topicname für AprilTag-Detektionen
Outputtopic	ROS-Topicname für berechnete Pose-Schätzung
Worldname	Name der Welt (Default: street_original)
Plotlevel	0 keine Grafik 1 xy-Plot mit Pose 2 ausführlicher xy-Plot und Darstellung über die Zeit mit Kovarianz **Vorsicht:** führt ev. zu Problemen da sehr rechenintensiv!

2.4 Custom Message Types

Um die Simulink-Modelle oder andere Funktionalitäten, wie die Lokalisierungs-Klasse, nutzen zu können, müssen ROS-Messages aus selbst compilierten ROS-Paketen ggf. bekannt gemacht werden. Die *Robotics System Toolbox* von Matlab enthält bereits typische Nachrichten-Definitionen, welche zum Zeitpunkt der Matlab-Version vorlagen. Darauf aufbauend können weitere Message-Definitionen vorgenommen werden[1]. In unserem Fall betrifft das bspw. das Paket zur AprilTag-Erkennung sowie Nachrichtendefinitionen des TUCar.

Ab der Matlab Version R2020b hat sich das Vorgehen zum Einbinden der eigenen Nachrichten-Definitionen grundlegend geändert, aus diesem Grund wird nachfolgend zwischen *Vor R2020b* und *Ab R2020b* unterschieden.

2.4.1 Vor R2020b

Das Laden Nachrichten-Definitionen findet in der `startup.m` der TUCarToolbox statt. Bitte zunächst diese Datei öffnen und die Version für vor R2020b entsprechend einkommentieren. Anschließend sollte auf den entsprechenden Hinweis geachtet werden, dass die `*.jar`-Dateien mit den Klassendefinitionen in die entsprechende Datei `javaclasspath.txt` aufgenommen werden, wie von `startup.m` ausgegeben wurde.

Ob die AprilTag-Messages korrekt bekannt sind, kann bspw. mit folgendem Maltab-Befehl getestet werden:

```
1 rosmessage('apriltags2_ros/AprilTagDetectionArray')
```

Wenn keine Fehlermeldung erscheint, funktioniert alles.

2.4.2 Fehlerbehandlung

> **Achtung:** Sollten sich zwischen zwei ROS-Versionen Nachrichtendefinitionen verändert haben, welche von einem Custom-Message Typ verwendet werden, kann es zu Inkompatibilitäten kommen. Eventuell wird ein md5 Checksummen Fehler ausgegeben, oder der selbst definierte Message-Typ wird gar nicht geladen.

Funktioniert der obige befehl nicht, sollte zuerst geprüft werden, ob die Java-Klassendefinitionen korrekt geladen worden:

```
1 s = javaclasspath('-static');
2 s{find(contains(s,'custom_msgs'))}
```

[1] siehe Matlab Dokumentation unter: *Create Custom Messages from ROS Package*
`https://de.mathworks.com/help/ros/ug/create-custom-messages-from-ros-package.html`

Sollte hier ein leeres Ergebnis erscheinen, bzw. keine Klasse namens `apriltags2_ros-*` erscheinen, dann liegt der Fehler womöglich in der fehlenden Pfaddefinition in der Datei `javaclasspath.txt`.

Weiterhin muss der Matlab-Pfad für die `*.m`-Dateien korrekt bekannt sein.
Hier: `TUCarToolbox/ros_msg_definitions/msggen`.

Ob ein bestimmter Message-Typ in Matlab zumindest registriert wurde, d.h. der Pfad zu den `*.m`-Dateien der Custom Message bekannt ist, kann wie folgt getestet werden:

```
s = rosmsg("list");
s{find(contains(s,'april'))}
```

Sollten alle Pfade korrekt definiert sein und das Erstellen von Message-Klassen in Matlab trotzdem nicht funktionieren, kann auch die Matlab-Version ein Grund sein. Wir sind dem nicht genau auf die Spur gegangen, jedoch sind die für Matlab generierten Message-Typen mindestens zwischen Matlab R2018 und R2019 inkompatibel. Wenn eine aktuelle Matlab-Version verwendet wird, sollte alles ohne Anpassungen funktionieren. Falls nicht, gibt es die Möglichkeit auf ältere Message-Definitionen zurückzugreifen. Im TUCarToolbox-Ordner gibt es wie bereits angesprochen, den Ordner `ros_msg_definitions`. Dieser ist ein symbolischer Link zu den eigentlichen Message-Definitionen im Ordner `custom_msgs`:

```
tucar-toolbox> ll ros_msg_*
  [...] ros_msg_definitions -> custom_msgs/R2019a/matlab_gen/
```

Für eine ältere Version kann dieser Link einfach korrigiert werden:

```
tucar-toolbox> rm ros_msg_definitions
tucar-toolbox> ln -s custom_msgs/R2018a/matlab_gen ros_msg_definitions
tucar-toolbox> ll ros_msg_*
  [...] ros_msg_definitions -> custom_msgs/R2018a/matlab_gen/
```

2.4.3 Ab R2020b, Erstellen der Custom Message Types in Matlab

Die neueren Matlab-Versionen besitzen nun intern die Möglichkeit, Nachrichten-Definitionen für die Nutzung in Matlab zu übersetzen. Hierzu steht die Funktion `rosgenmsg` bereit. Im unterschied zu früheren Matlab-Versionen lassen sich die bereits übersetzten Nachrichten-Definitionen nicht mehr so einfach zwischen verschiedenen Rechnern austauschen und die folgende Vorgehensweise muss bei jeder Änderung oder Neuinstallation bzw. sogar beim Verschieben des Quellcode-Ordners durchgeführt werden. Leider gibt es beim Übersetzen einige Fallstricke in Verbindung mit Ubuntu, die ggf. einen höheren Aufwand bedeuten. Aus diesem Grund folgen Hinweise und eine mögliche Vorgehensweise.

> **CMake:** Zum übersetzen der ROS-Nachrichten wird `CMake` verwendet. Da Matlab eigene Standard-Bibliotheken verwendet, besteht die Möglichkeit, dass diese nicht mit den Systemweiten Bibliotheken kompatibel sind und CMake nicht aufgerufen werden kann. Um dies zu testen, wird folgendes Matlab-Kommando ausgeführt:
>
> ```
> system('cmake --version')
> ```
>
> Werden hier fehlermeldungen ausgegeben, so muss Matlab mit `LD_PRELOAD` gestartet werden, damit die Systemweiten Bibliotheken anstatt der Matlab-Varianten genutzt werden. Matlab wird wie folgt gestartet:
>
> ```
> LD_PRELOAD="/usr/lib/x86_64-linux-gnu/libstdc++.so.6:/usr/lib/x86_64-linux-gnu/
>     libcurl.so.4" matlab -softwareopengl
> ```

> **CMake Version aktualisieren:** Ggf. kann es sein, dass die installierte CMake-Version zu alt ist. In diesem Fall kann wie folgt vorgegangen werden:
>
> 1. Ermitteln der URL zum Download des aktuellen Quellcodes.
> 2. herunterladen:
>   `wget https://github.com/Kitware/CMake/releases/download/v3.20.2/cmake-3.20.2.tar.gz`
> 3. entpacken:
>   `tar -xzvf cmake-3.20.2.tar.gz`
> 4. Konfigurieren mit einem lokalen Ziel:
>   `./bootstrap -prefix=/home/YOURHOME/.local`
> 5. Kompilieren und installieren:
>   `make -j8 install`
> 6. Ggf. muss noch folgendes in die `~/.bashrc` aufgenommen werden:
>   `export PATH="$(systemd-path user-binaries):$PATH"`

> **Python:** Sollte beim ausführen von `rosgenmsg` die Fehlermeldung kommen, dass Python 2.7 benötigt wird, dann muss folgendes in Matlab ausgeführt werden (vorausgesetzt Python 2.7 ist systemweit installiert):
>
> ```
> pyenv('Version','/usr/bin/python2.7')
> ```

Das TUCar-Toolbox Repo besitzt im Ordner `custom_msgs/definitions` die notwendigen Nachrichten-Definitionen aus den betreffenden ROS-Paketen für die wir die Nachrichten-Definitionen in Matlab nutzen möchten. Wir erstellen diese für die beiden Pakete `apriltag2_ros` sowie `tucar_msgs`, indem der folgende Befehl in Matlab ausgeführt wird (Pfade anpassen!):

```
rosgenmsg('~/workspace/TUCar-Toolbox/custom_msgs/definitions')
```

Folgende Ausschrift sollte erscheinen:

```
To use the custom messages, follow these steps:

1. Add the custom message folder to the MATLAB path by executing:

addpath('~/workspace/TUCar-Toolbox/custom_msgs/definitions/
    matlab_msg_gen_ros1/glnxa64/install/m')
savepath

2. Refresh all message class definitions, which requires clearing the
    workspace, by executing:

clear classes
rehash toolboxcache

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.
```

Entsprechend werden die Pfadangaben für `addpath(.)` in der `startup.m` angepasst. Achtung, nachdem die Message-Definitionen erstellt wurden dürfen Sie nicht mehr verschoben werden.

3 Simulation

Die Simulation des TUCars erfolgt in Gazebo[1], einer Quelloffenen Robotersimulation. In Bezug auf das TUCar wurde eine entsprechende *Welt* für die Simulationsumgebung erstellt, welche die Straßenkarte sowie die AR-Tags enthält. Des weiteren wurde ein Modell des echten Roboters implementiert, um wahlweise zwischen *realem* oder *simulierten* Roboter umzuschalten. D.h. für beide Welten sind die gleichen Sensoren bzw. Aktoren mit deren gleichnamigen ROS-Topics implementiert.

Auch wenn damit eine theoretische Übertragung von Navigationsalgorithmen zwischen Realität und Simulation einfach möglich ist, sollte folgendes bedacht werden: die Herausforderung bei jeder Simulation besteht in der Abbildung sämtlicher Dynamiken und Störeinflüsse der realen Welt. Je nachdem, welcher Aufwand in die Entwicklung eines Simulationsmodells investiert wird, verringern sich die Abweichungen zwischen beiden Welten.

Das von uns genutzte Simulationsmodell basiert auf dem `seat_car_simulator`[2] und wurde entsprechend angepasst.

3.1 Installation

> **Achtung:** Für die Verwendung der Simulation sind neben der eigentlichen Gazebo-Simulation noch folgende, möglicherweise nicht standardmäßig installierten, ROS-Pakete notwendig:
>
> - `gazebo-ros-control`
>   siehe `http://wiki.ros.org/gazebo_ros_control`
> - `joint-state-controller`
>   siehe `http://wiki.ros.org/joint_state_controller`

Für die Lehrveranstaltung *Projektpraktikum Autonome Systeme* sind die ROS-Paketquellen im GitLab der TU Chemnitz zu finden.[3]. Um diese zu installieren, sollte der dort beschriebenen Anleitung gefolgt werden.

3.2 Paketstruktur

Der Code ist in folgende Pakete gegliedert:

tucar_gazebo_bringup Enthält alle Launchfiles und Parameter (*launch* & *config*) die zum Starten der Simulation benötigt werden. Außerdem sind darin die Modelle für die Welt (*world* & *urdf* & *materials*) enthalten.

In *materials* ist die Textur der Plane & Tags inklusive deren Definition als „material“ vorhanden. Der Ordner *worlds* enthält die für Gazebo relevanten Informationen zum Laden einer Umgebung/Welt. Das soeben erwähnte „material“ wird hier verwendet, indem es in das Modell der Plane (Textur + 3D Modell optisch + 3D Modell Kollision) eingebunden wird. Im „world“-Ordner/Datei werden auch Einstellungen für die Physik-Engine vorgenommen.

[1] Achtung: Es wird mindestens Version 7.16 benötigt! In Ubuntu 16.04 ist bspw. nur Version 7.0 in den offiziellen Paketquellen. Unter: `http://gazebosim.org/tutorials?tut=install_ubuntu` wird eine alternative Installation mit Paketquellen von `osrfoundation.org` beschrieben. Wenn der Anleitung folgend das Paket *gazebo7* installiert wird, sollte Gazebo ausreichend aktuell sein.

[2] siehe `https://gitlab.iri.upc.edu/mobile_robotics/adc/seat_car_simulator/tree/master`

[3] siehe `https://gitlab.hrz.tu-chemnitz.de/proaut/pas/tucar-ros`

tucar_description Enthält die Beschreibung des TUCars (Kann auch unabhängig von der Simulation verwendet werden.). In *meshes* sind die CAD-Datein (`*.stl`) des TUCars gespeichert. Die Karosserie wurde mit „Photogrammetry“ bzw. „Structure from Motion“ gescannt. Ein nützlicher Artikel findet sich bei Prusa[4]. Als Tools wurden COLMAP und Meshlab verwendet.

tucar_gazebo_controller Enthält die Implementierung der Ackermann-Lenkung bzw. der Regelung des Autos – d.h. hier werden die Fahrbefehle in Form von Lenkwinkel und Geschwindigkeit auf die simulierten Gelenke übertragen.

3.3 Verwendung

Zum Starten von Gazebo mit dem im Labor ausgelegten Straßenszenario und dem TUCar muss lediglich das entsprechende Launchfile gestartet werden:

```
> roslaunch tucar_gazebo_bringup tucar_street_scenario.launch
```

Folgende Parameter können dabei angegeben werden:

- **robot** (string, default: sim01)
  Name des Roboters
- **world** (string, default: street_original)
  Name der Welt/Straßenkarte
- **topcam** (bool, defalut: false)
  zusätzliche Kamera zur Draufsicht auf das Straßenszenario
- **gui** (bool, default: true)
  Gazebo-Gui (Deaktivierung zur Ressourcenschonung ev. sinnvoll)
- **start_pc_bringup** (bool, default: true)
  startet pc_bringup.launch mit Bildverarbeitung und Apriltag-Erkennung (Deaktivierung zur Ressourcenschonung bei verwendung der Gound Truth Pose ev. sinnvoll)

Um bspw. notwendige Rechenleistung einzusparen, kann Gazebo mit aktivierter Draufsicht und ohne GUI gestartet werden:

```
> roslaunch tucar_gazebo_bringup tucar_street_scenario.launch topcam:=true gui:=false
```

HINWEIS: Wird Gazebo ohne GUI über x2Go gestartet, muss ggf. vor Aufruf des obigen Befehls in der selben Konsole noch folgendes ausgeführt werden[5]:

```
> Xvfb :1 -screen 0 1600x1200x16 &
> export DISPLAY=:1.0
```

Um ein zweites TUCar (mit anderem Namespace) zu platzieren kann

```
> roslaunch tucar_gazebo_bringup spawn_tucar.launch x:=2.0 ns:=/sim_NAME
```

verwendet werden.
Zum zurücksetzen des TUCar auf die Startposition kann folgender Service verwendet werden:

```
> rosservice call /gazebo/reset_world "{}"
```

[4]siehe *Photogrammetry – 3D scanning with just your phone/camera* von 2018, unter `https://blog.prusaprinters.org/photogrammetry-3d-scanning-just-phone-camera/`
[5]Siehe: `https://answers.gazebosim.org//question/14625/running-a-camera-sensor-headless/`

reset_simulation funktioniert nicht, da dies zu Problemen mit den weiterverarbeitenden Nodes (aufgrund des Rücksetzens der Simulationszeit) führt. Gegebenenfalls sollten vorher noch die Ansteuerwerte auf 0 gesetzt werden:

```
> rostopic pub /sim01/drive_vel_angle tucar_msgs/TUCarDriveVelAngle
        "header:
        seq: 0
        stamp:
        ecs: 0
        nsecs: 0
        frame_id: ''
        vel: 0.0
        angle: 0" -1
```

Es ist auch möglich weitere Kameras zu initialisieren:

```
> roslaunch tucar_gazebo_bringup spawn_topcam.launch ns:=cam2
```

Dabei sollen die Parameter x/y/z, roll/pitch/yaw sowie rate/height/width/fov sinnvoll gesetzt werden. Die Kamera schaut in Richtung x-Achse.

Mit Ausführen von `spawn_box.launch` kann eine skalierbare (rote) Box an beliebigen Punkten des Testszenarios platziert werden. Die Parameter $< x/y/z > _size$, $< x/y/z > _pos$ und *mass* bestimmen Größe, Position und Gewicht der Box.

```
> roslaunch tucar_gazebo_bringup spawn_box.launch x_size:=1 y_size:=1 z_size:=1 x_pos:=2
    y_pos:=2 z_pos:=1 mass:=100
```

3.4 Aufbau und Funktion

Nachfolgend wird auf die Implementierungsdetails des Fahrzeug-Modells eingegangen.

3.4.1 Kamera

Die Simulation der Kamera des TUCars erfolgt in Gazebo als sogenannter Wideangle-Camera-Sensor[6], siehe Abbildung 3.1. Im Gegensatz zum Nomalen Kamera-Sensor kann hier auf komplexere Abbildungsfunktionen als das Lochkameramodell zurückgegriffen werden.

Informationen zu Sensoren in Gazebo finden sich z.B. in der SDF-Format-Beschreibung[7]. Die Abbildungsfunktion des RB-CAMERA-WW2-Objektivs kann gut als linear geteilt (äquidistant) angenommen werden, d.h.

$$r = c * \theta \tag{3.1}$$

wobei r dem Abstand vom Mittelpunkt des Bildsensors, c der Brennweite und θ dem Winkel des einfallenden Strahls zur optischen Achse des Objektives entspricht.

3.4.2 IMU

Die IMU wird mittels des `GazeboRosImuSensor`-Plugin[8] simuliert, welches in `sensors.gazebo` konfiguriert wird. Über die Parameter des Gazebo-Sensors und/oder des Gazebo-Plugins kann ein Gauß'sches Rauschen simuliert werden. Die Parameter hierfür können den Launchfiles `tucar_street_scenario.launch` oder `spawn_tucar.launch` übergeben werden.

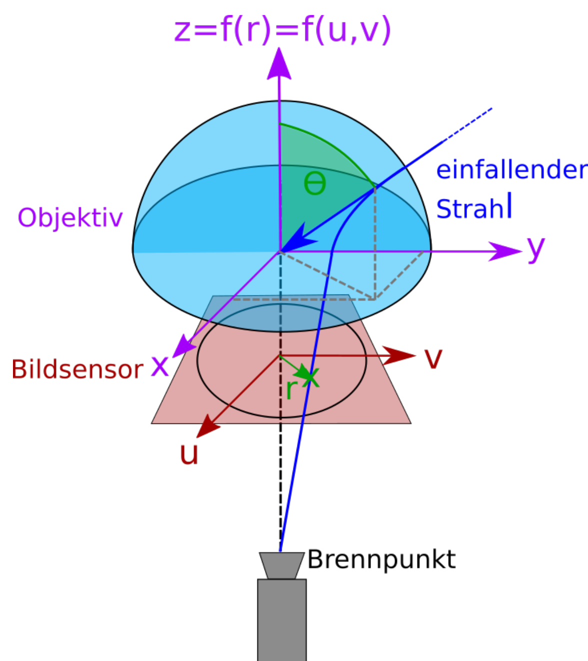

Abbildung 3.1: Kameramodell der Omnikamera in Gazebo

3.4.3 Odometrie

Die Odometrie des simulierten TUCars wird im Gegensatz zum realen Fahrzeug aktuell nur an den Hinterrädern abgegriffen. Es wird hierfür über die Drehgeschwindigkeiten / absolvierten Umdrehungen der Hinterräder gemittelt, da das reale TUCar nur einen Encoder am Motor besitzt.

Rauschen Zur Simulation einer fehlerbehafteten Odometrie stehen zur Bearbeitung der Ground-Truth-Werte von Odometrie und Geschwindigkeit jeweils 4 Parameter zur Verfügung (siehe Tabelle 3.1)

	gefahrene Strecke	Geschwindigkeit
absolute Std.	std_odom_pose_abs	std_odom_vel_abs
rel. Std.	std_odom_pose_rel	std_odom_vel_rel
absoluter Offset	offset_odom_pose_abs	offset_odom_vel_abs
rel. Off.	offset_odom_pose_rel	offset_odom_vel_rel

Tabelle 3.1: Parameter für das Rauschen der Odometrie

3.4.4 Ground-Truth-Sensor (Pose + Geschwindigkeiten)

Die Ground-Truth Pose und Geschwindigkeiten des Body-Koordinatensystems $\mathcal{B}$ können mittels des P3D-Plugin abgegriffen werden, welches in `gt.gazebo` konfiguriert wird. Mittels des Parameters `std_vel_gt` (Standardabweichung) kann ein Gauß'sches Rauschen auf die Tangential- und Winkelgeschwindigkeiten simuliert werden. Dieser Parameter kann den Launchfiles `tucar_street_scenario.launch` oder spawn_tucar.launch übergeben werden.

[6] siehe `https://gist.github.com/klokik/08f5429aea86eba921d6`
[7] siehe `http://sdformat.org/spec?elem=sensor`
[8] siehe `http://gazebosim.org/tutorials?tut=ros_gzplugins`

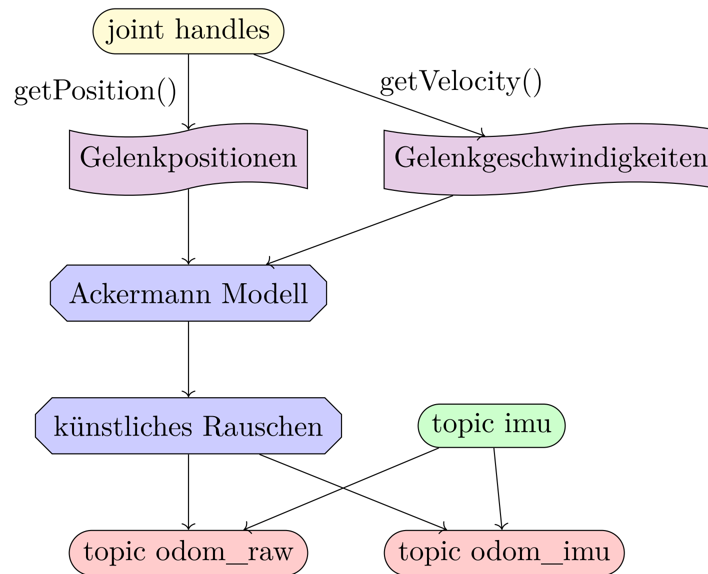

Abbildung 3.2: Flussdiagram zur Simulation der Odometriedaten des TUCar

Die Pose und Geschwindigkeiten werden als Typ nav_msgs/Odometry im Topic state_gt veröffentlicht.

Literatur

Lange, S., P. Weissig, A. Uhlig und P. Protzel (2018). „TUC-Bot: A Microcontroller Based Robot for Education“. In: *Robotics in Education*. Hrsg. von W. Lepuschitz, M. Merdan, G. Koppensteiner, R. Balogh und D. Obdržálek. Advances in Intelligent Systems and Computing. Springer International Publishing, S. 201–213. ISBN: 978-3-319-62875-2.

Leopold Mauersberger und Daniel Käppler (2018). „Kamerabasierte Navigation eines Modellfahrzeugs in einem Straßenverkehrsszenario“. Bachelorarbeit.