



# ENTWICKLUNG EINER GESTENERKENNUNG MITHILFE VON STEREOKAMERAS MIT ROS

BACHELORARBEIT  
ZUR ERLANGUNG DES AKADEMISCHEN GRADES  
BACHELOR OF ENGINEERING (B. ENG.)

Oliver Bosin

Betreuer:  
Prof. Dr. Ferdinand Englberger

Tag der Abgabe: 27.06.2019

Universität der Bundeswehr München  
Fakultät für Elektrotechnik und Technische Informatik  
Institut 4

Neubiberg, Juni 2019



---

## **Erklärung**

**gemäß Beschluss des Prüfungsausschusses für die Fachhochschulstudiengänge der UniB-wM vom 25.03.2010**

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.

Neubiberg, den 26. Juni 2019

---

Oliver Bosin

## **Erklärung**

**gemäß Beschluss des Prüfungsausschusses für die Fachhochschulstudiengänge der UniB-wM vom 25.03.2010**

Der Speicherung meiner Masterarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Neubiberg, den 26. Juni 2019

---

Oliver Bosin



# **Abstrakt**

Gegenstand dieser Bachelorarbeit ist die Entwicklung eines Gestenerkennungssystems, mit Hilfe von der Stereokamera "Kinect" von "MICROSOFT", mit ROS. Es wird zuerst auf alle Komponenten, die den Systemaufbau ausmachen, eingegangen. Das Konzept hinter dem System ist ebenfalls Teil der Ausführungen. Zu den wichtigsten Komponenten, dies sind die Stereokamera, ROS, "MoveIt!" und "OpenNI", werden danach die Grundlagen erklärt. Auf die Beschreibung des Systems und seiner Komponenten folgen die Ausführungen zur Implementation und Inbetriebnahme des Systems. Nachfolgend wird die Implementation und Funktionsweise von zwei Paketen, die im Zuge dieser Bachelorarbeit entwickelt wurden, erklärt. Diese Pakete zeigen mögliche Anwendungen, für das Gestenerkennungssystem auf. Die Ergebnisse, Probleme und Erkenntnisse dieser Bachelorarbeit, werden am Schluss diskutiert. Ein Ausblick wird ebenfalls gegeben.

Für Studierende und Interessierte, welche den Einstieg in ROS und die natürliche Interaktion suchen, bietet diese Arbeit eine erste Orientierung.

# Inhaltsverzeichnis

<b>Tabellenverzeichnis</b>	<b>V</b>
<b>Abbildungsverzeichnis</b>	<b>VII</b>
<b>Listings</b>	<b>IX</b>
<b>Abkürzungsverzeichnis</b>	<b>XI</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Aufgabenstellung . . . . .	1
1.3. Gliederung . . . . .	1
<b>2. Systemaufbau</b>	<b>3</b>
2.1. Software . . . . .	3
2.1.1. Betriebssysteme . . . . .	3
2.1.2. Integrierte Entwicklungsumgebungen . . . . .	3
2.1.3. Robot Operating System . . . . .	4
2.1.4. OpenNI . . . . .	4
2.1.5. SensorKinect . . . . .	5
2.1.6. Ros-Kinetic-OpenNI . . . . .	5
2.1.7. NITE . . . . .	5
2.1.8. OpenNI Tracker . . . . .	5
2.1.9. MoveIt! . . . . .	5
2.1.10. Turtle Simulator . . . . .	5
2.1.11. Softwarekonzept . . . . .	6
2.2. Hardware . . . . .	8
2.2.1. Rechner . . . . .	8
2.2.2. Kinect . . . . .	8
2.2.3. Roboterarm "rob_arm_small" . . . . .	8
2.2.4. Mikrocontrollerboard . . . . .	9
2.3. Systemaufbau . . . . .	10
<b>3. Grundlagen</b>	<b>11</b>
3.1. Stereokamera . . . . .	11
3.1.1. Typen . . . . .	11

3.2. OpenNI . . . . .	13
3.2.1. Was ist OpenNI . . . . .	13
3.2.2. Module . . . . .	13
3.2.3. Kompatibilität und Verfügbarkeit . . . . .	14
3.3. ROS . . . . .	14
3.3.1. Was ist ROS . . . . .	14
3.3.2. Konzept und Komponenten . . . . .	15
3.3.3. Wichtige ROS-Befehle . . . . .	17
3.4. MoveIt! . . . . .	17
3.4.1. Architektur . . . . .	18
3.4.2. Das move_group_interface . . . . .	19
<b>4. Implementation des Gestenerkennungssystems mit ROS</b>	<b>23</b>
4.1. Vorbereitung . . . . .	23
4.1.1. Catkin Workspace erstellen . . . . .	23
4.1.2. Abhängigkeiten . . . . .	24
4.2. Installation der Komponenten . . . . .	24
4.2.1. OpenNI installieren . . . . .	24
4.2.2. Ros-Kinetic-OpenNI installieren . . . . .	25
4.2.3. NITE installieren . . . . .	25
4.2.4. OpenNI Tracker installieren . . . . .	26
4.3. Start und Test des Gestenerkennungssystems . . . . .	26
4.3.1. Starten der Komponenten . . . . .	26
4.3.2. Mit Rviz visualisieren und überprüfen . . . . .	27
<b>5. Implementation von Geste steuerungen basierend auf dem Gestenerkennungssystem</b>	<b>31</b>
5.1. Vorbereitung . . . . .	31
5.1.1. Installation MoveIt! . . . . .	31
5.1.2. Einrichtung der Pakete rob_arm_small und rob_arm_small_hw_interface . . . . .	31
5.1.3. Erstellung eines ROS Paketes . . . . .	32
5.2. Paket turtlesim_gesture_control . . . . .	33
5.2.1. Der Knoten gesture_control . . . . .	34
5.2.2. Starten der Geste steuerung . . . . .	35
5.2.3. Anleitung zur Geste steuerung . . . . .	36
5.3. Paket roboticarm_gesture_control . . . . .	38
5.3.1. Der Knoten roboticarm_gesture_control . . . . .	38
5.3.2. Starten der Geste steuerung . . . . .	42
5.3.3. Anleitung zur Geste steuerung . . . . .	42
<b>6. Diskussion</b>	<b>43</b>
6.1. Ausblick . . . . .	44
<b>A. Quellcode</b>	<b>45</b>

<b>B. Arbeitsaufwand</b>	<b>53</b>
<b>Literaturverzeichnis</b>	<b>55</b>
<b>Index</b>	<b>57</b>



# **Tabellenverzeichnis**

2.1. Winkelwerte der Gelenke[18] . . . . .	9
2.2. Griffweite des Greifers[18] . . . . .	9
3.1. Wichtige Befehle in ROS[3, 13] . . . . .	17



# Abbildungsverzeichnis

2.1.	Softwarearchitektur . . . . .	6
2.2.	Abstract Layered View - OpenNI . . . . .	7
2.3.	Softwarekonzept . . . . .	7
2.4.	Kinect . . . . .	8
2.5.	Freiheitsgrade des Roboterarms[18] . . . . .	9
2.6.	Aufbau Gesamtsystem . . . . .	10
3.1.	Kommunikationsaufbau von Knoten . . . . .	16
3.2.	Architektur MoveIt! . . . . .	18
4.1.	Ordnerstruktur Catkin-Workspace . . . . .	24
4.2.	Psi Pose . . . . .	27
4.3.	Rviz - Fixed Frame Auswahl . . . . .	28
4.4.	Rviz - DepthCloud hinzufügen . . . . .	28
4.5.	Rviz - Color Image Topic Auswahl . . . . .	29
4.6.	Rviz - TF hinzufügen . . . . .	29
4.7.	Rviz - Displays Endergebnis . . . . .	30
4.8.	Rviz - Visualisierung von DepthCloud und Skelett . . . . .	30
5.1.	ROS Paket Ordnerstruktur . . . . .	32
5.2.	turtlesim_gesture_control Sequenz-Diagramm . . . . .	33
5.3.	Turtle Simulator Oberfläche . . . . .	37
5.4.	Gestensteuerung Anleitung . . . . .	37
5.5.	roboticarm_gesture_control - Sequenzdiagramm . . . . .	39
B.1.	Arbeitsaufwand . . . . .	53



# Listings

3.1. Planen auf ein Pose Goal . . . . .	20
3.2. Planen auf ein Joint Goal . . . . .	20
3.3. Planen auf ein Position Target . . . . .	21
3.4. Plan ausführen . . . . .	21
5.1. Erzeugen TransformListener . . . . .	34
5.2. Hauptschleife vom Knoten gesture_control . . . . .	34
5.3. Launch-Datei - start_demo.launch . . . . .	36
5.4. Hauptschleife vom Knoten roboticarm_gesture_control . . . . .	40
A.1. move_group_interface.cpp . . . . .	45
A.2. gesture_control.cpp . . . . .	50



# Abkürzungsverzeichnis

**API** Application Programming Interface

**RGB** Rot Grün Blau

**NI** Natural Interaction

**XML** Extensible Markup Language

**URDF** Unified Robot Description Format

**ROS** Robot Operating System

**FPGA** Field Programmable Gate Array

**PWM** Pulsweitenmodulation

# **1. Einleitung**

## **1.1. Motivation**

Nach dem Erfolg des Touchscreen ist sicher, dass die intuitive Bedienung von Technologien, einen großen Einfluss auf den Erfolg dieser hat. Selbst in Küchenmaschinen, wie dem "Thermomix" von "Vorwerk", werden heutzutage Touchscreens verbaut. Da Roboter, für Industrie und auch für andere Nutzergruppen, immer wichtiger werden, stellt sich nun die Frage, wie eine intuitive Steuerung hier aussehen könnte. Bei den humanoiden Robotern oder Roboterarmen bietet sich hier die Gestensteuerung oder auch die Sprachsteuerung an. Mit einer Gestensteuerung könnte ein neuer Nutzer einen Roboterarm, nach nur einer kurzen Einweisung, mit Sicherheit ohne größere Probleme steuern. Wenn eine intuitive Steuerung auch Roboter mehr Menschen näher bringen könnte, dann würde dies die Forschung und Entwicklung, in dem Bereich der Robotik, beschleunigen. Im Falle vom Robot Operating System, würde, durch das größere Interesse an der Robotik, die ROS-Community weiter anwachsen und die Weiterentwicklung von ROS vorantreiben. Innerhalb des Instituts 4 für Datentechnik und Schaltungstechnik, hat sich ROS bereits, als Basis für Projekte in der Robotik, bewährt. Unter Anderem wurden autonom fahrende Fahrzeuge, mit ROS als Basis, entwickelt. Die im Institut vorhandenen Fahrzeuge werden entweder durch Programmcode oder durch Fernsteuerungen gesteuert. Um diese Möglichkeiten der Steuerung, um die Gestensteuerung, zu erweitern, wird im Zuge dieser Bachelorarbeit eine Gestenerkennung, mit Hilfe von Stereokameras, mit ROS entwickelt und implementiert.

## **1.2. Aufgabenstellung**

Das Ziel dieser Bachelorarbeit ist es, mit Hilfe einer Stereokamera, mit ROS eine Gestenerkennung zu entwickeln und zu implementieren. Als sekundär Ziel wird die Entwicklung eines ROS-Paketes, welches ein Beispiel für die Anwendung des Gestenerkennungssystems aufzeigt, angestrebt. Der Hauptzweck dieser Arbeit, neben der Entwicklung der Gestenerkennung, ist es, eine erste Grundlage, für Entwicklungen und nachfolgende Arbeiten, im Bereich der natürlichen Interaktion mit Robotersystemen zu legen. Wie sich die Gestenerkennung in die Umgebung von ROS einfügt und wie sie in dieser Umgebung verwendet werden kann, sind Kernfragen dieser Grundlage.

## **1.3. Gliederung**

Die Arbeit besteht aus folgenden Kapiteln:

## *1. Einleitung*

---

- **Kapitel 2 - Systemaufbau:**

Hier werden die Systemkomponenten aufgeführt und es werden jeweils kurze Erklärungen, zu jeder Komponente, gegeben. Ebenfalls wird hier Software und Hardware die verwendet wurde, aber nicht direkt zum Gestenerkennungssystem gehört, aufgeführt und kurz erklärt. Am Ende des Kapitels wird der gesamte Systemaufbau dargestellt und erklärt.

- **Kapitel 3 - Grundlagen:**

In diesem Kapitel werden grundlegende Funktionsweisen und Konzepte, der verwendeten Hardware und Software, erläutert. Es wird hier auf die Stereokamera, das Framework "OpenNI", ROS und das Framework "MoveIt!" eingegangen.

- **Kapitel 4 - Implementation des Gestenerkennungssystems mit ROS:**

Hier wird auf die Implementation des Gestenerkennungssystems eingegangen. Es wird ebenfalls auf die Inbetriebnahme und den Test des Systems eingegangen.

- **Kapitel 5 - Implementation von Gesteinsteuerungen basierend auf dem Gestenerkennungssystem:**

In diesem Kapitel werden die entwickelten ROS-Pakete vorgestellt, welche Gesteinsteuerungen, die das Gestenerkennungssystem verwenden, implementiert haben. Es wird auf mögliche Voraussetzungen, die Inbetriebnahme der Pakete und die Verwendung der Pakete eingegangen.

- **Kapitel 6 - Diskussion:**

Im letzten Kapitel werden die Ergebnisse erläutert und bewertet. Es wird zusätzlich auf aufgetretene Probleme eingegangen. Zudem werden Empfehlungen, für weitere Arbeiten und Verbesserungen am System, gegeben. Am Ende wird noch ein allgemeiner Ausblick gegeben.

# **2. Systemaufbau**

Der Aufbau des gesamten Systems lässt sich, in den Hardwareanteil und den Softwareanteil, aufgliedern. Zuerst wird auf die verschiedenen Pakete und Treiber, die den Softwareanteil ausmachen, eingegangen und deren Zusammenspiel im Gesamtsoftwarekonzept erläutert. Im zweiten Teil des Kapitels werden die Hardwarekomponenten des Systems vorgestellt. Hier wird als erstes die Aufgabe jeder relevanten Komponente beschrieben, um diese dann verständlich in einen Kontext bringen zu können. Am Ende dieses Kapitels werden der Softwareanteil und der Hardwareanteil, im Gesamtkonzept, zu einer verständlichen Darstellung des Systems zusammengefügt.

## **2.1. Software**

In diesem Abschnitt wird die, in dem Entwicklungsprozess, verwendete und eingebundene Software vorgestellt. Dies soll einen ersten Überblick über den Entwicklungsprozess geben und das Softwarekonzept des Systems verständlich darstellen. Die Installation und die genauere Verwendung, der hier genannten Software, ist integraler Teil des vierten Kapitels und wird somit hier nicht behandelt.

### **2.1.1. Betriebssysteme**

Auf dem zur Entwicklung genutzten Rechner ist "WINDOWS 10" als Betriebssystem installiert. Da ROS ein Framework für Linuxdistributionen ist, ist es notwendig ein Linux-Betriebssystem, in einer virtuellen Maschine, zu betreiben. Hierfür wird eine virtuelle Maschine von "VM-Ware" verwendet. In der virtuellen Maschine wird ein "Ubuntu" in der Version 16.04 LTS, welches im Institut 4 laufend gepflegt und erweitert wird, betrieben. Unter dieser Installation von "Ubuntu", des Institut 4, ist das ROS-Framework, in der Version ""Kinetic Kame", bereits installiert. Bei der virtuellen Maschine von "VM-Ware" ist darauf zu achten, dass der USB-Kompatibilitätsmodus auf "USB 3.0" gesetzt ist, da sonst Probleme mit USB-Geräten auftreten können.

### **2.1.2. Integrierte Entwicklungsumgebungen**

Für die Entwicklung von Software reicht in der Regel ein Texteditor und ein Compiler aus. Um diese Entwicklung komfortabler und übersichtlicher zu gestalten, werden integrierte Entwicklungsumgebungen verwendet. Hier sind normalerweise Texteditor, Compiler, Linker und weitere Programme in einer Anwendung integriert, um ein ständiges wechseln zwischen den Programmen zu vermeiden.

## **2. Systemaufbau**

---

### **Eclipse**

Eclipse Um Programme in der der Programmiersprache C/C++ zu entwickeln, wurde die integrierte Entwicklungsumgebung "Eclipse"[2] verwendet. Die verwendete Installation von "Eclipse" wurde innerhalb des Institut 4, mit dem "Ubuntu" Betriebssystem zusammen, gepflegt. Weiterhin waren keine Anpassungen, der Konfiguration von "Eclipse", nötig um Programme für das ROS-Framework darin zu entwickeln.

### **Keil**

Keil Die integrierte Entwicklungsumgebung " $\mu$ -vision" von "KEIL" wurde verwendet, um gegebenenfalls Änderungen an der Software, die auf dem Mikrocontrollerboard läuft, vorzunehmen.[5] Auf das genannte Mikrocontrollerboard wird im Hardwareteil des Kapitels eingegangen.

#### **2.1.3. Robot Operating System**

Das "Robot Operating System" ist ein quelloffenes und flexibles Framework, das darauf abzielt die Entwicklung von Software, für Robotersysteme, zu vereinfachen. Dazu stellt dieses Framework eine Sammlung von Tools, Bibliotheken und anderer Software zur Verfügung.[13, 7] Unter ROS werden zur Entwicklung von Software die Programmiersprachen C/C++ und Python verwendet. Um die Konsistenz zu anderer Software die im Institut 4 verwendet wird zu wahren, wurde, für die im Zuge dieser Bachelorarbeit entwickelte Software, die Programmiersprache C/C++ gewählt. Auf den Aufbau und die Funktionen von ROS wird im Grundlagenkapitel eingegangen.

### **Rviz**

Rviz Die Anwendung "Rviz" ist ein Tool zur 3D-Visualisierung in ROS. Diese wird unter Anderem für die Simulation von Robotern genutzt. Weiterhin wird mit "Rviz" visualisiert was die Sensoren "sehen" und die Aktoren "tun".[14]

#### **2.1.4. OpenNI**

Um auf 3D-Sensoren wie Stereokameras zuzugreifen, wurde unter "Ubuntu" das quelloffene Framework "OpenNI" verwendet. Es bietet API's um für 3D-Sensoren, RGB-Kameras, IR-Kameras und Audioeingabegeräte Anwendungen zu entwickeln.[10] Im Zuge dieser Bachelorarbeit wurde die Möglichkeit von "OpenNI" genutzt, unkompliziert Treiber, für die genannten Gerätetypen, einzubinden. Weiterhin ist dieses Framework notwendig, um Software die auf diesem basiert ausführen zu können. Dieses Framework ist als erster Baustein in dem Gesamtsoftwarekonzept des Systems zu sehen.

## 2.1.5. SensorKinect

Das Paket "SensorKinect" ist ein Modul, für das Framework "OpenNI". Dieses Modul ermöglicht den Zugriff, über die API's von "OpenNI", auf die Stereokamera "Kinect" von "MICROSOFT". [16] Somit dient "SensorKinect" als Hardwaretreiber, für die Stereokamera "Kinect". Als Modul fügt sich "SensorKinect" in den Baustein "OpenNI", des Gesamtsoftwarekonzeptes, ein. Wird ein zum Treiber von "PrimeSense" kompatibler Sensor genutzt, ist die Installation von "SensorKinect" nicht erforderlich.[10]

## 2.1.6. Ros-Kinetic-OpenNI

Um die Funktionen des Framework "OpenNI" auch in Verbindung mit ROS verwenden zu können, werden zusätzlich mehrere ROS-Pakete, die im Weiteren unter dem Namen "Ros-Kinetic-OpenNI" gefasst werden sollen, benötigt.

## 2.1.7. NITE

Das Paket "NITE" ist eine Middleware, die sich in die Infrastruktur des Framework "OpenNI" mit eingliedert. Diese Middleware erweitert die Funktionen von "OpenNI", unter Anderem um das Tracking von Handbewegungen und Körperbewegungen im ganzen.[12, 10]

## 2.1.8. OpenNI Tracker

Das ROS-Paket "OpenNI Tracker" greift auf die durch "NITE" implementierten Funktionen zu, um die Positionen von Körpergelenken zu bestimmen und diese in ROS zur Verfügung zu stellen.[11]

## 2.1.9. MoveIt!

Mit "MoveIt" wurde ein weiteres Framework in ROS genutzt. Das Framework "MoveIt" ist die "State-Of-The-Art-Software" für Navigationsplanung und Manipulationsplanung in der Robotik.[9, 7] Dieses Framework war notwendig, da, basierend auf der zu implementierenden Gestenerkennung, eine Gestensteuerung, für einen Roboterarm entwickelt wurde.

## 2.1.10. Turtle Simulator

Der "Turtle Simulator" ist eine Anwendung, in der Schildkröten, auf einem 2D-Spielfeld, gesteuert werden können. In der ROS-Community wird der "Turtle Simulator" für Tutorials genutzt.[17]

## 2. Systemaufbau

---

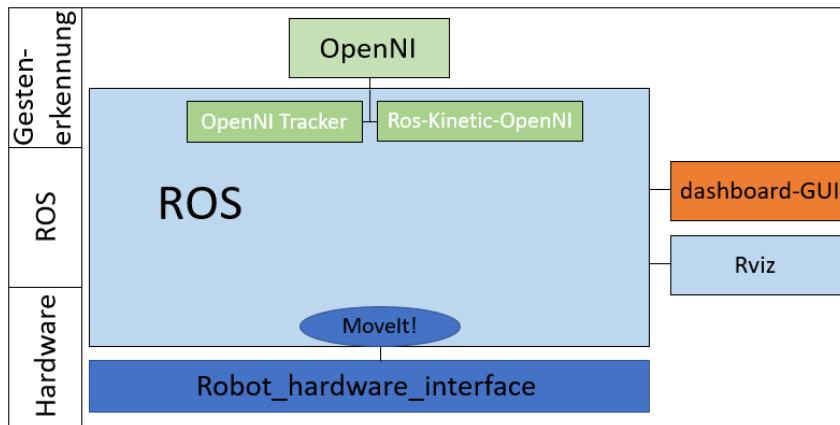


Abbildung 2.1.: Softwarearchitektur

### 2.1.11. Softwarekonzept

In diesem Abschnitt wird das Softwarekonzept des Gesamtsystems dargestellt und erklärt. In der Abbildung 2.1 ist die Softwarearchitektur des Gestenerkennungssystems, mit Anteilen der Steuerung eines Roboterarms, zu sehen. Die Softwarearchitektur lässt sich in drei Schichten aufteilen. Die ROS-Schicht in der Mitte bildet die Basis und stellt die Umgebung zur Verfügung in der die Anwendungen laufen werden, welche die Gestenerkennung nutzen. Zu der Gestenerkennungsschicht gehören die Bausteine "OpenNI", "OpenNI Tracker" und "Ros-Kinetic-OpenNI". In der Abbildung 2.2, einer "Abstract Layered View", ist der schematische Aufbau des Bausteins "OpenNI" zu sehen. Wie zu sehen ist, fügen sich die Module "SensorKinect" und "NITE" in diesen ein und werden im Weiteren nicht mehr eigenständig behandelt. Die Hardwareschicht setzt sich aus den Bausteinen "MoveIt!" und "Robot\_hw\_interface" zusammen. Unter dem Baustein "Robot\_hw\_interface" sind alle Bestandteile, die zur Kommunikation von "MoveIt!" bis zum Roboterarm "rob\_arm\_small" und zur Steuerung des Roboterarms dienen, zusammengefasst. Auf die Anteile der Hardwareschicht und den Roboterarm "rob\_arm\_small" wird im Hardwareteil dieses Kapitels sowie im Grundlagenkapitel näher eingegangen. Mit dem Baustein "Rviz" können 3D-Punktfolgen, basierend auf den von "Ros-Kinetic-OpenNI" bereitgestellten Daten, dargestellt werden. Parallel zur Punktfolge kann das von "OpenNI Tracker" erkannte Skelett, über TF's, visualisiert werden. Der Baustein "dashboard-GUI" bietet die Möglichkeit unkompliziert GUI's zu erstellen. Dies ist für das einfache und übersichtliche Debugging von Software und Hardware interessant. Die Abbildung 2.3 zeigt das Softwarekonzept des Gesamtsystems. Den äußeren Rahmen bildet hier das Hostbetriebssystem "Windows 10". Auf diesem Hostbetriebssystem läuft im "VmWare Player", einer virtuellen Maschine, ein "Ubuntu" der Version 16.04 LTS. Unter diesem Betriebssystem fügt sich dann die Softwarearchitektur des Gestenerkennungssystems ein.

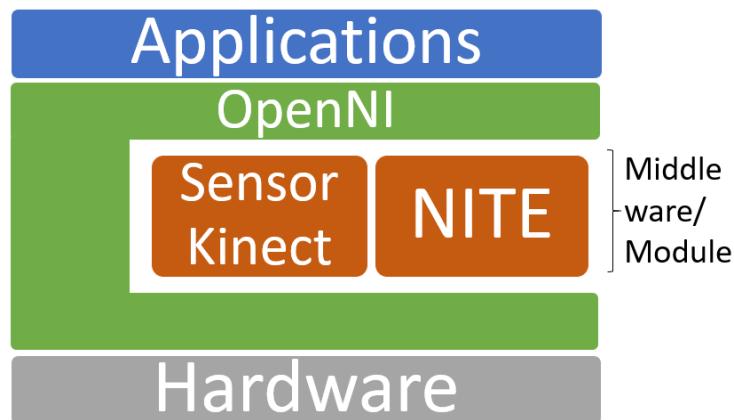


Abbildung 2.2.: Abstract Layered View - OpenNI

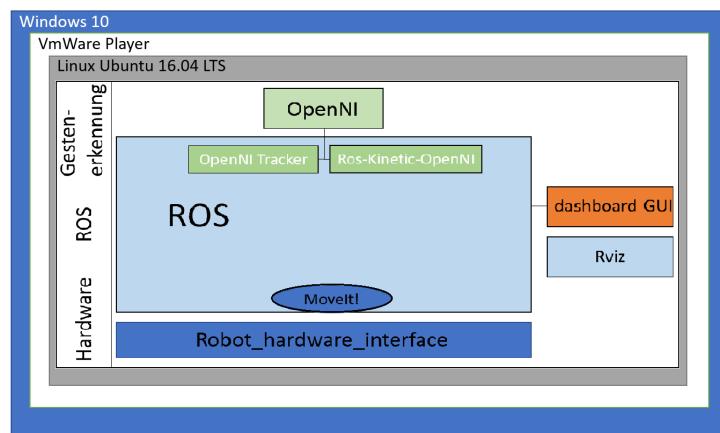


Abbildung 2.3.: Softwarekonzept

## 2. Systemaufbau

---



Abbildung 2.4.: Kinect

## 2.2. Hardware

In diesem Abschnitt wird die Hardware des Gesamtsystems kurz vorgestellt. Auf die Funktionsweise und die Grundlagen zu elementarer Hardware wird erst im Grundlagenkapitel näher eingegangen.

### 2.2.1. Rechner

Der verwendete Rechner, auf dem das Hostbetriebssystem lief, hat folgende Systemspezifikation:

**Prozessor:** AMD Ryzen Threadripper 1900X

**Arbeitspeicher:** 64 GB

**Grafikkarte:** AMD Radeon RX580

Bei dem verwendeten Rechner ist darauf zu achten, dass der verbaute Prozessor die Befehlsätze SSE3, SSE4 und SSSE3 unterstützt. Bei nicht unterstützen der genannten Befehlssätze, können Fehler, zur Laufzeit der verwendeten Software, auftreten. Weiterhin ist darauf zu achten, dass mindestens ein vollwertiger USB 3.X Port belegbar ist.

### 2.2.2. Kinect

Für das Gestenerkennungssystem wurde die "Kinect for Windows"<sup>[6]</sup> Modell 1517 von "MICROSOFT", zu sehen in Abbildung 2.4, als Stereokamera genutzt. Es ist hier darauf zu achten, dass die "Kinect" nur an einem vollwertigen USB 3.X Port betrieben wird. Wenn der genutzte USB Port nicht die volle elektrische Leistung bringt, kann es zu Problemen während dem Betrieb kommen.

### 2.2.3. Roboterarm "rob\_arm\_small"

Basierend auf dem implementierten Gestenerkennungssystem wurde eine Gestensteuerung, für den Roboterarm "rob\_arm\_small", entwickelt. Dieser Roboterarm wurde von der Firma "Crust-Crawler" unter der Bezeichnung "SG6-UT" vertrieben. Wie der Abbildung 2.5 zu entnehmen ist,

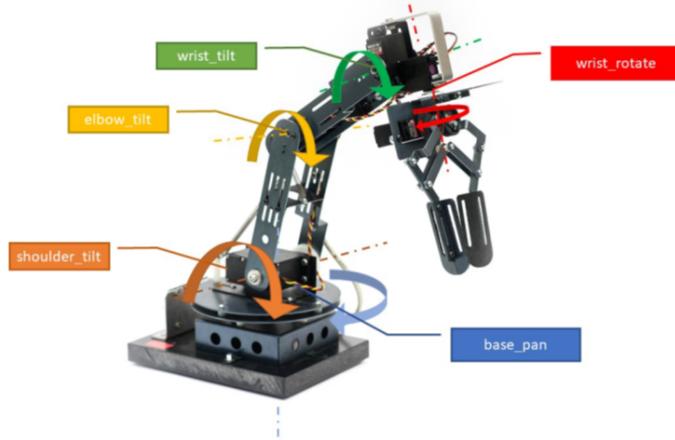


Abbildung 2.5.: Freiheitsgrade des Roboterarms[18]

Joint	Min. Winkel			Max. Winkel		
	[rad]	[°]	[μs]	[rad]	[°]	[μs]
base_pan	-0.785	-45	1950	+0.785	+45	1050
shoulder_tilt	-0.261	-15	1854	+1.57	+90	1067
elbow_tilt	0.0	0	1906	+1.57	+90	1027
wrist_tilt	0.0	0	1934	+1.57	+90	1042
wrist_rotate	-0.785	-45	1959	+0.785	+45	1032

Tabelle 2.1.: Winkelwerte der Gelenke[18]

ist der "rob\_arm\_small" ein Roboterarm mit sechs Freiheitsgraden.[18] In der Tabelle 2.1 sind die Winkelwerte, mit zugehörigen Pulsweiten, festgehalten. Die Griffweite des Grippers, mit zugehöriger Pulsweite, ist in der Tabelle 2.2 festgehalten. Diese Werte wurden bereits in einer vorangegangenen Projektarbeit[19] ,durch Christian Waldner, ermittelt.

## 2.2.4. Mikrocontrollerboard

Das Mikrocontrollerboard, ein "STM32F746ZG" von "STMicroelectronics", wird zur Erzeugung der PWM-Signale, für die Servomotoren des Roboterarmes "rob\_arm\_small", verwendet. Die auf dem Mikrocontrollerboard laufende Software wurde durch das Institut 4 zur Verfügung gestellt. Die Verbindung zum Rechner, auf dem das Gestenerkennungssystem implementiert ist,

Joint	Min. Winkel			Max. Winkel		
	[rad]	[μs]	Öffnung [cm]	[rad]	[μs]	Öffnung [cm]
z_gripper	0.0	1555	6,4	1.0	1156	3,3

Tabelle 2.2.: Griffweite des Greifers[18]

## 2. Systemaufbau

---

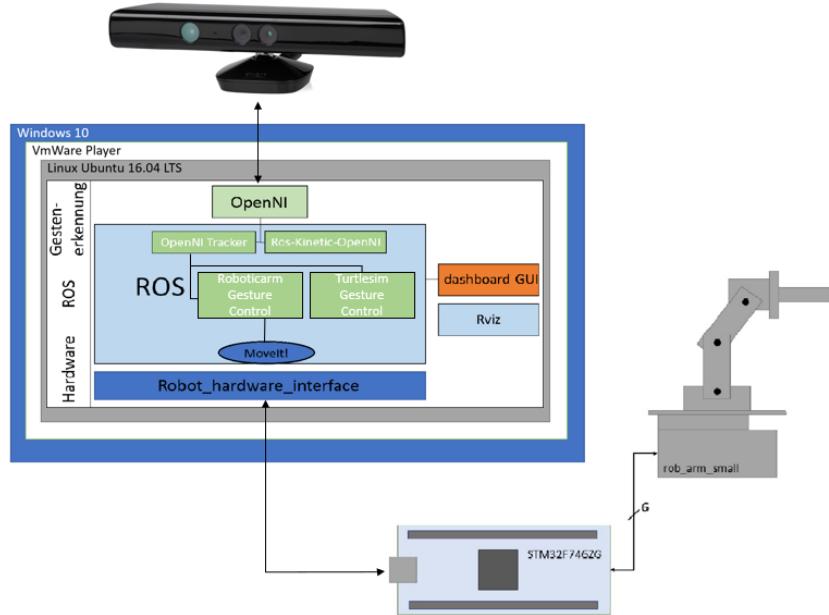


Abbildung 2.6.: Aufbau Gesamtsystem

wird über Ethernet hergestellt.[18]

## 2.3. Systemaufbau

Im Systemaufbau wird das Softwarekonzept und die verwendete Hardware zusammengeführt. In der Abbildung 2.6 ist der Aufbau des Gesamtsystems dargestellt. In diesem Aufbau ist, mit dem Mikrocontrollerboard und dem Roboterarm "rob\_arm\_small", der Hardwareanteil der Gestensteuerung für den Roboterarm enthalten. Weiterhin wurden die Pakete "Roboticarm Gesture Control" und "Turtlesim Gesture Control" in den Aufbau eingefügt. Als integraler Teil dieser Bachelorarbeit wird auf die Entwicklung, Implementation und die Funktionsweise der beiden Pakete im vierten Kapitel eingegangen. Der Baustein "OpenNI" hat, wie in Abbildung 2.6 zu sehen, Zugriff auf die Hardware der "Kinect". Über die API's von "OpenNI" greifen die Pakete, "OpenNI Tracker" und "Ros-Kinetic-OpenNI", auf die verarbeiteten Bilddaten der "Kinect" zu. Das Paket "OpenNI Tracker" nutzt zusätzlich das durch NITE implementierte "Skelett-Tracking". Das Framework "MoveIt!" stellt, über das "Robot\_hardware\_interface", eine Verbindung zum Mikrocontrollerboard her. Den Anwendungen in der ROS-Schicht bietet "MoveIt!" Funktionen zur Generierung von Steueranweisungen für Roboter an. In diesem Systemaufbau werden die Steueranweisungen an das Mikrocontrollerboard gesendet und von diesem in PWM-Signale, für die Servomotoren des "rob\_arm\_small", umgewandelt.

# 3. Grundlagen

In diesem Kapitel wird auf die Grundlagen zu verwendeter Hardware und Software eingegangen. Dies soll Hintergrundinformationen, zur gesamten Arbeit und speziell für die Ausführungen im vierten Kapitel, bereitstellen. Hierbei wird sich nur auf die, für das Gesamtsystem, wesentlichen Komponenten beschränkt.

## 3.1. Stereokamera

Eine Stereokamera ist ein Kamerasystem, dass zur Gewinnung von Tiefenbildern genutzt wird. Diese Tiefenbilder enthalten, im Gegensatz zu normalen Farbbildern, den Abstand einzelner Punkte zum Sensor der Kamera. Diese Kamerasysteme haben immer zwei optische Sensoren zur Bildzeugung. Es gibt unter anderem Systeme mit zwei RGB-Sensoren sowie Systeme mit einem RGB-Sensor und einem Infrarotsensor.

### 3.1.1. Typen

Unter den Stereokameras gibt es verschiedene Typen, die sich in die zwei folgenden Gruppen einteilen lassen: Diese Gruppen sind, Kameras mit passiven Verfahren und Kameras mit aktiven Verfahren. Für diese Arbeit wurden die Beschreibungen auf die gängigsten Typen, von Stereokameras, begrenzt.

### Embedded Stereo

Die "Embedded Stereo" Kameras nutzen das passive Stereoverfahren, um Tiefenbilder zu erzeugen. Diese Kamerasysteme haben meist zwei RGB-Sensoren um Bilder aufzunehmen, sowie einen eingebetteten Mikroprozessor oder FPGA, für die Berechnung von Tiefenbildern, integriert. Das passive Stereoverfahren ist an das menschliche Sehen angelehnt, bei dem aus zwei 2D-Bildern ein 3D-Bild gemacht wird. Das passive Verfahren, zur Erzeugung der Tiefenbilder, lässt sich in mehrere Schritte aufteilen. Zu erst wird mit beiden RGB-Sensoren gleichzeitig ein Bild aufgenommen. Anschließend wird an beiden Bildern eine Merkmalsextraktion durchgeführt. Diese Bildmerkmale, auch als "Keypoints" bezeichnet, lassen Unterschiede zu Ihrer Umgebung erkennen. Die gefundenen Bildmerkmale, beider Bilder, werden bei der Korrespondenzsuche mit einander verglichen. Nachdem die Korrespondenzsuche abgeschlossen ist, ist es noch notwendig falsche Korrespondenzen aus den Ergebnissen herauszufiltern. Mit den endgültigen korrespondierenden Bildmerkmalen kann nun, mittels Triangulation, die Entfernung zu diesem Merkmal im Bild errechnet werden.[15, 8]

### 3. Grundlagen

---

Um den Prozessor des nutzenden Hauptsystems zu entlasten, werden die Berechnungen, für das beschriebene Verfahren, auf dem eingebetteten Mikroprozessor oder FPGA durchgeführt. Da der hohe Rechenaufwand auf eine erhöhte Leistungsaufnahme schließen lässt, werden die "Embedded Stereo" Kameras nicht für mobile Systeme empfohlen.

## Time-Of-Flight

Eine "Time-Of-Flight" Kamera gehört zu den Kamerasystemen die aktive Verfahren, zur Generierung von Tiefenbildern, nutzen. Das von "Time-Of-Flight" Kameras genutzte Verfahren ist das Laufzeitverfahren. Bei dem Laufzeitverfahren wird ein Signal von der Kamera ausgesendet und die Zeit ermittelt wie lange das Reflektierte Signal gebraucht hat, um wieder an der Kamera aufzutreffen. Aus der Laufzeit  $t$  und Geschwindigkeit  $v$  des Signals kann, über die Formel  $S = vt$ , direkt auf die Strecke  $S$  zum reflektierenden Objekt geschlossen werden. Als typische Emitter-Sensor Systeme, in dem Laufzeitverfahren, zählen Radar-, Ultraschall- und Infrarotsysteme.[15]

Die "Time-Of-Flight" Kameras nutzen in der Regel Infrarotsysteme. Die gängigen "Time-Of-Flight" Kameras haben meist einen RGB-Sensor, einen oder mehrere Infrarotemitter und einen Infrarotsensor verbaut. Aufgrund der hohen Ausbreitungsgeschwindigkeit von Licht, werden hohe Anforderungen an die Elektronik zur Zeitmessung, gestellt. Unter anderem legt die kleinste messbare Zeitdifferenz  $\Delta t$  die maximale Tiefenauflösung  $\Delta R$ , mit der Formel  $\Delta R = \frac{v\Delta t}{2}$ , fest.[4] Durch diese hohen Anforderungen an die Elektronik, sind hochauflösende "Time-Of-Flight" Kameras kostenintensiv. Da hier Licht, im Infrarotbereich, detektiert wird, sind "Time-Of-Flight" Kameras nicht für Bereiche mit direkter Sonneneinstrahlung geeignet.[1] In Bereichen mit direkter Sonneneinstrahlung empfehlen sich stattdessen Kameras die auf passive Verfahren zur Tiefengewinnung zurückgreifen.

## Infrarotmuster

Wie die "Time-Of-Flight" Kameras nutzen auch "Infrarotmuster" Kameras ein aktives Verfahren, zur Gewinnung von Tiefenbildern. Das Triangulationsverfahren, mit dem Ansatz des codierten Lichts, wird von den "Infrarotmuster" Kameras verwendet. Bei diesem Ansatz wird ein Infrarotmuster, mit bekanntem Code, in den Raum projiziert und mit einer Infrarotkamera aufgenommen. Somit sind, über den bekannten Code, alle vorher definierten Messpunkte im Bild eindeutig identifizierbar. Aufgrund der Kenntnis über die Position von Infrarotemitter und Infrarotkamera, kann für jeden Messpunkt der Abstand zur Kamera, über die Triangulation, errechnet werden.[4, 8] Wie die "Time-Of-Flight" Kameras sind auch die "Infrarotmuster" Kameras, aufgrund der Verwendung von Infrarotlicht, nicht für Bereiche mit direkter Sonneneinstrahlung geeignet. Die für die Umsetzung des Gestenerkennungssystems genutzte Stereokamera, die "Kinect", ist eine "Infrarotmuster" Kamera. Somit ist das entwickelte Gestenerkennungssystem nur für Innenbereiche geeignet.

## 3.2. OpenNI

Das Framework "OpenNI" soll in diesem Abschnitt, in seiner Funktion und seinem Aufbau, beschrieben werden.

### 3.2.1. Was ist OpenNI

"OpenNI" ist ein Framework, dass verschiedene Programmiersprachen erlaubt und für verschiedene Plattformen verfügbar ist. Weiterhin definiert es API's zur Entwicklung von Anwendungen, die natürliche Interaktion, wie Sprache und Gesten, verwenden. Die API's von "OpenNI" bestehen aus mehreren Schnittstellen, zur Entwicklung von NI-Anwendungen. Der Hauptzweck von "OpenNI" ist es, eine Standard-API zur Verfügung zu stellen, die eine Kommunikation mit folgenden Komponenten ermöglicht:

- **optischen und akustischen Sensoren**
- **Middleware, die Funktionen zur Verarbeitung von optischen und akustischen Sensordaten implementiert**

"OpenNI" liefert hierzu Schnittstellen, welche von den Sensorgeräten implementiert werden müssen sowie Schnittstellen die von der Middleware implementiert werden müssen. Durch diesen Ansatz werden Abhängigkeiten, zwischen Sensorgeräten und Middleware, vermieden. Dies ermöglicht es Anwendungen, ohne den Aufwand der Portierung, auf der Basis von verschiedenen Middleware's zu arbeiten. Des Weiteren bietet "OpenNI" die Möglichkeit, Anwendungen zu entwickeln die Sensorrohdaten verwenden, unabhängig vom Sensor der diese liefert.[10]

### 3.2.2. Module

Als Module werden Gerätetreiber und Middleware bezeichnet, die im "OpenNI" Framework registriert wurden. Die folgenden Module werden von "OpenNI" unterstützt:[10]

#### Sensormodule

- **3D Sensor**
- **RGB-Kamera**
- **Infrarotkamera**
- **Mikrofon**

### *3. Grundlagen*

---

#### **Middleware-Module**

- **Middleware zur Ganzkörperanalyse**
- **Middleware zur Analyse der Handposition**
- **Middleware zur Gestenerkennung**
- **Middleware zur Analyse von Szenen in Bildern**

#### **3.2.3. Kompatibilität und Verfügbarkeit**

Die Entwickler von "OpenNI" garantieren volle Rückwärtskompatibilität[10]. Dies bedeutet das Anwendungen, die auf Basis irgendeiner Version von "OpenNI" entwickelt wurden, mit neueren Versionen von "OpenNI" arbeiten können.

"OpenNI" ist für folgende Betriebssysteme verfügbar[10]:

- **Windows XP und aktueller**
- **Linux Ubuntu 10.10 und aktueller**

### **3.3. ROS**

Dieser Abschnitt soll eine Einführung in das Konzept, die Komponenten und das Vokabular von ROS geben. Diese Informationen sind, zum Verständnis des vierten Kapitels, elementar. Die folgenden Punkte werden in dieser Einführung behandelt:

- **Was ist ROS**
- **Konzept und Komponenten**
- **Wichtige ROS-Befehle**

#### **3.3.1. Was ist ROS**

Da ROS eine Vielzahl von Aufgaben eines Betriebssystems übernimmt, aber als Umgebung ein Linux Betriebssystem benötigt, wird es oft als "Meta-Betriebssystem" bezeichnet. Die elementare Aufgabe von ROS ist es, eine Kommunikation zwischen Nutzer, Betriebssystem und externer Hardware bereitzustellen. Zur externen Hardware, mit der kommuniziert werden kann, zählen beispielsweise Sensoren, Kameras und Roboter. In dem ROS sich wie ein Betriebssystem verhält, bringt es den Vorteil der Hardwareabstraktion mit. Somit kann ein Nutzer einen Roboter steuern, ohne große Kenntnis über dessen Hardware- und Softwaredetails zu haben. Als quelloffenes Framework, ist ROS für jeden frei verfügbar. Dieser Ansatz wird auch bei den meisten ROS-Paketen verfolgt und diese unter der BSD-3-Lizenz veröffentlicht. Aufgrund dieser Voraussetzungen, werden neue Entwicklungen und neues Wissen, in der ROS-Community, schnell

verbreitet. Aber auch für Unternehmen ist ROS interessant, da auch kommerzielle Produkte mit ROS entwickelt werden können. Hierfür ist die einzige Bedingung, dass der Copyright-Vermerk der ursprünglichen Software zitiert wird. [3, 13]

### 3.3.2. Konzept und Komponenten

Das Konzept von ROS teilt sich in drei Konzeptschichten auf[13]:

- **ROS Dateisystem**
- **ROS Computation Graph**
- **ROS Community**

#### ROS Dateisystem

Das ROS-Dateisystem setzt sich aus folgenden Komponenten zusammen:

- **Packages:** In Paketen wird die Software in ROS organisiert.
- **Manifests:** Im Manifest sind Metadaten über das Paket enthalten. Diese Manifests sind in XML geschrieben.
- **Stacks:** Pakete, die gemeinsam eine Funktion bereitstellen, werden in Stacks zusammengefasst.
- **Stack Manifest:** Im Stack Manifest sind Metadaten über das Stack enthalten.[3, 13]

#### ROS Computation Graph

Das Netzwerk von Prozessen, die eine gemeinsame Aufgabe erfüllen, in ROS bildet den "Computation Graph". Die folgenden Komponenten versorgen den Graph mit Daten:

- **Nodes:** Als Knoten werden Prozesse in ROS bezeichnet, welche eine Aktion ausführen. Diese Knoten haben die Möglichkeit sich bei dem Master zu registrieren und untereinander zu kommunizieren. Die Verbindungsinformationen, zur Kommunikation untereinander, erhalten die Knoten vom Master. Hierfür melden die Knoten dem Master welche Topics abonniert werden und auf welchen Topics Daten veröffentlicht werden. Knoten können auf verschiedene Arten gestartet werden. Zum einen über ein Terminalfenster durch ausgeführte Befehle, zum anderen als Teil eines Programms, geschrieben in Python oder C++. [3, 13]
- **Master:** Durch den Master wird die Kommunikation zwischen Knoten aufgebaut. Der Master bietet dafür Namensdienste und Dienste zum registrieren an. In Abbildung 3.1 ist ein Kommunikationsaufbau beispielhaft dargestellt. Im ersten Schritt meldet sich der Knoten "Camera" beim Master an und informiert diesen darüber, dass, unter dem

### 3. Grundlagen

---

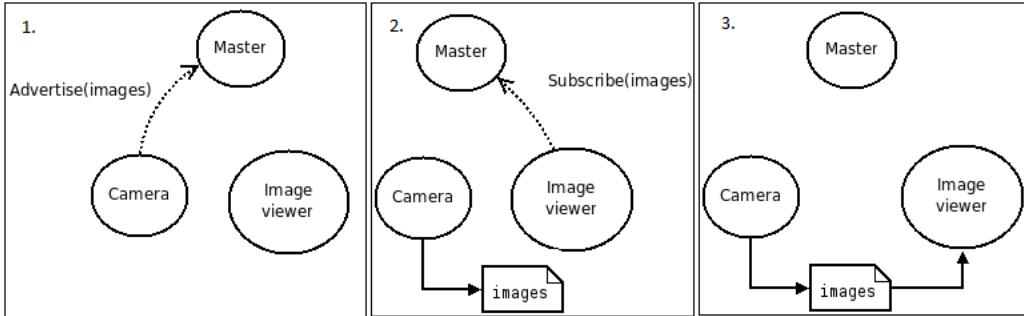


Abbildung 3.1.: Kommunikationsaufbau von Knoten

Topic "images", Bilder veröffentlicht werden. Im zweiten Schritt meldet sich der Knoten "Imageviewer" beim Master an und abonniert über diesen das Topic "images". Im dritten Schritt beginnt der Knoten "Imageviewer" die Bilder direkt über das Topic "images" zu empfangen.[3, 13]

- **Parameter Server:** Der "Parameter Server" stellt eine geteilte Bibliothek von Parametern dar, welche innerhalb des Masters ausgeführt wird. Knoten können zur Laufzeit Parameter in dieser Bibliothek speichern und abrufen.[3, 13]
- **Messages:** Die Kommunikation der Knoten untereinander findet durch Nachrichten statt. Diese Nachrichten sind Datenstrukturen, welche die auszutauschenden Informationen in typisierten Feldern beinhalten. Von diesen Nachrichten gibt es verschiedene Typen, welche unterschiedliche Aufgaben haben. Es können auch neue Nachrichtentypen definiert werden.[3, 13]
- **Topics:** Die Verteilung der Nachrichten, zwischen den Knoten, wird über einen "Publisher-Subscriber" Mechanismus durchgeführt. Um diesen Mechanismus umzusetzen werden Themen verwendet. Die Knoten können Nachrichten zu einem Thema veröffentlichen, welches über einem Namen identifiziert wird. Ein anderer Knoten, der an den Nachrichten interessiert ist, kann dieses Thema abonnieren, um die veröffentlichten Nachrichten zu erhalten. Knoten können unter mehreren Themen veröffentlichen und sie können mehrere Themen abonnieren.[3, 13]
- **Services:** Um, neben dem "Publisher-Subscriber" Mechanismus, Knoten die Möglichkeit zu geben Anfragen von anderen Knoten zu erhalten und zu beantworten werden Dienste verwendet.[3, 13]

## ROS Community

- **Distributions:** Neue Versionen von ROS werden über Distributionen verteilt. Diese Distributionen setzen sich aus verschiedenen "Stacks" zusammen, die jeweils auch eine Versionsnummern tragen. Durch die Verteilung über Distributionen, wird die Installation von ROS erleichtert.[13]

Befehl	Aktion	Befehlsbeispiele und Beispiele für Unterbefehle
roscore	Durch diesen Befehl wird der ROS Master gestartet	\$ roscore
rosrun	Führt ein Programm aus und erstellt einen Knoten	\$ rosrun [Paketname] [Ausführbare Datei]
rosnode	Zeigt Informationen zu Knoten und listet aktive Knoten auf	\$ rosnode info [Name des Knoten] \$ rosnode <Unterbefehl> <b>Unterbefehle:</b> list
rostopic	Zeigt Informationen über Topics an	\$ rostopic <Unterbefehl> <Name vom Topic> <b>Unterbefehle:</b> echo, type, info
rosmsg	Zeigt Informationen über Message-Typen an	\$ rosmsg <Unterbefehl> [Paketname] / [Message Typ] <b>Unterbefehle:</b> show, type, list
rosservice	Zeigt Informationen zu Services an	\$ rosservice <Unterbefehl> [Servicename] <b>Unterbefehle:</b> args, call, find, info, list, type
rosparam	Liefert und setzt Werte von Parametern	\$ rosparam <Unterbefehl> [Parameter] <b>Unterbefehle:</b> get, set, list, delete

Tabelle 3.1.: Wichtige Befehle in ROS[3, 13]

- **Repositories:** In der ROS Community werden Pakete und Quellcode über Repositories, wie "github"<sup>1</sup>, verteilt.[13]
- **The ROS Wiki:** Im ROS Wiki<sup>2</sup> sind Dokumentation und Tutorials rund um ROS veröffentlicht. Nach einer Anmeldung können hier eigene Dokumentationen, Tutorials, Korrekturen und andere Informationen veröffentlicht werden.[13]
- **ROS Answers:** "ROS Answers"<sup>3</sup> ist ein weiteres Portal für Informationen über ROS. Hier können Fragen gestellt und Antworten geteilt werden.

Durch diese organisierte Community werden Lösungen für Probleme schneller gefunden und diese Lösungen an zentraler Stelle bereitgestellt.

### 3.3.3. Wichtige ROS-Befehle

In der Tabelle 3.1 ist eine Auswahl von wichtigen ROS-Befehlen aufgeführt:

## 3.4. MoveIt!

"MoveIt!" ist ein Framework das Fähigkeiten für Manipulation, Bewegungsplanung, Steuerung und mobile Manipulation bietet. Zusätzlich bietet "MoveIt" verschiedene Tools, welche bei der Entwicklung von Anwendungen unterstützen. Weiterhin steht auch hinter "MoveIt" eine Community, die gemeinsam "MoveIt!" erweitert und pflegt. Ein weiterer Vorteil ist, dass für viele gängige Robotermodelle bereits Dokumentationen, in Verbindung mit "MoveIt!", vorhanden

<sup>1</sup><https://github.com/>

<sup>2</sup><http://wiki.ros.org/de>

<sup>3</sup><https://answers.ros.org/>

### 3. Grundlagen

---

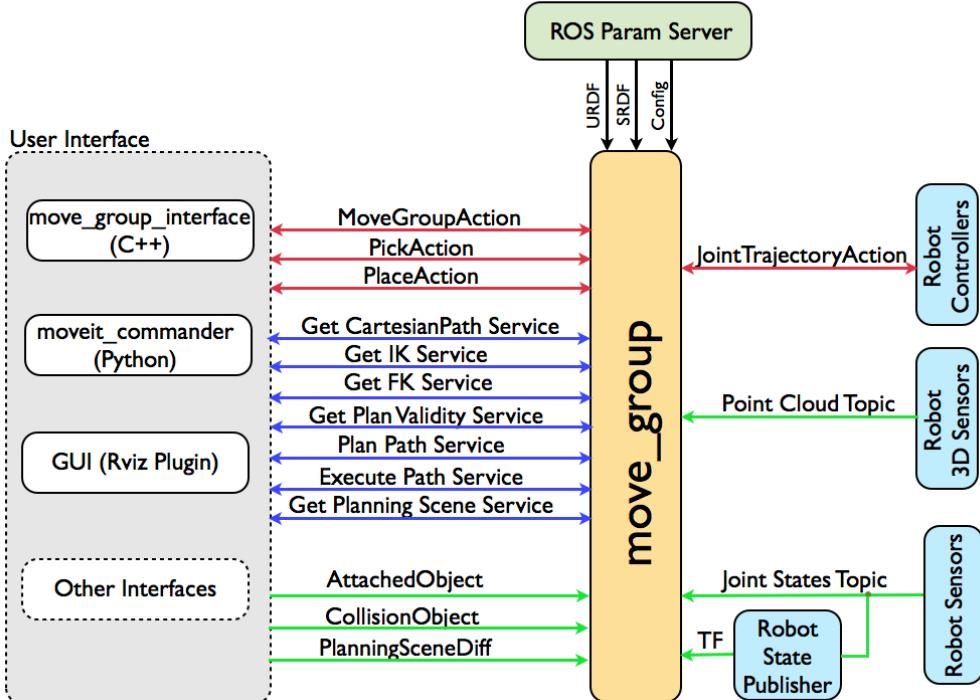


Abbildung 3.2.: Architektur MoveIt! <sup>4</sup>

sind.[7] Dies erleichtert den Einstieg in die Entwicklung von Anwendungen mit "MoveIt!". Um die Erstellung eines Pakets mit "MoveIt!" weiter zu erleichtern, wird ein Setup-Assistent bereitgestellt. Dieser Setup-Assistent bietet die Möglichkeit neue Roboter zu importieren und für diese alle benötigten Komponenten eines "MoveIt"-Paketes zu generieren[7]. Erläutert werden in diesem Abschnitt nur Informationen zu "MoveIt!", welche essentiell für diese Arbeit sind. Für weitergehende Information wird auf die folgenden Quellen verwiesen:

- **Die MoveIt! Webseite**<sup>5</sup>
- **Die MoveIt! Tutorials**<sup>6</sup>
- **Den MoveIt! Quellcode**<sup>7</sup>

#### 3.4.1. Architektur

In der Abbildung 3.2 ist die Architektur von "MoveIt!" dargestellt.

---

<sup>4</sup>(besucht am 03.06.2019): <https://moveit.ros.org/documentation/concepts/>

<sup>5</sup><https://moveit.ros.org/>

<sup>6</sup>[https://ros-planning.github.io/moveit\\_tutorials/](https://ros-planning.github.io/moveit_tutorials/)

<sup>7</sup><https://github.com/ros-planning>

## Kinematik

Für die direkte Kinematik bietet "MoveIt!" eine native Implementation, andererseits bietet es für die inverse Kinematik eine auf Plugins basierte Architektur. Dies bedeutet das die Berechnung der inversen Kinematik jederzeit, durch Austausch des Plugin, an die eigenen Bedürfnisse angepasst werden kann.[7]

## Bewegungsplanung

Die Bewegungsplanung wird durch eine Plugin-Schnittstelle implementiert. Dies ermöglicht es "MoveIt!" mit verschiedenen Bewegungsplanern, aus einer Vielzahl von Bibliotheken, zu kommunizieren. Dieser Ansatz betont noch einmal die weitgehende Erweiterbarkeit und Anpassungsfähigkeit von "MoveIt!".[7]

## Planning Scene

Die "Planning Scene" repräsentiert die Welt um den Roboter herum und beinhaltet auch den aktuellen Zustand vom Roboter selbst. Die "Planning Scene"-Schnittstelle bietet den primären Zugriff für Nutzer, um den Zustand der Welt, in der der Roboter operiert, zu verändern.[7]

## Trajectory Processing

Im Gegensatz zu reinen Bewegungsplanern bietet "MoveIt!" zusätzlich ein "Trajectory Processing". Dies bedeutet das neben dem Weg auch die Beschleunigung, an bestimmten Punkten, berechnet wird. Somit kann der Weg zeitparametrisiert werden. Notwendige Limits für die Beschleunigung der einzelnen Gelenke werden aus der Datei `joint_limits.yaml` ausgelesen.[7]

### 3.4.2. Das move\_group\_interface

Das "move\_group\_interface" ist eine Benutzerschnittstelle, welche API's für den "move\_group"-Knoten bereitstellt. Die Schnittstelle abstrahiert die ROS-API auf "MoveIt!" und vereinfacht deren Nutzung dadurch. [7]

## Pose Goal planen

Bei einem "Pose Goal" wird die Koordinate im Raum und die einzunehmende Orientierung des Endeffektors angegeben. Der folgende C++ Code zeigt wie auf ein "Pose Goal" geplant werden kann:

### 3. Grundlagen

---

```
1 moveit::planning_interface::MoveGroup group("rob_arm_group");  
2  
3 geometry_msgs::Pose pose;  
4 pose.orientation.w = 1.0;  
5 pose.position.x = 0.22;  
6 pose.position.y = -0.5;  
7 pose.position.z = 0.0;  
8 group.setPoseTarget(pose);  
9  
10 moveit::planning_interface::MoveGroup::Plan my_plan;  
11 bool success = group.plan(my_plan);
```

Listing 3.1: Planen auf ein Pose Goal

In Zeile eins wird die "MoveGroup", mit der gearbeitet werden soll, definiert. Als nächstes werden der Message "pose" die benötigten Werte zugewiesen. Die Message "pose" wird der Funktion "setPoseTarget()" übergeben und mit der Funktion "plan()" ein Plan berechnet.

### Joint Goal planen

Bei einem "Joint Goal" werden für jedes Gelenk die Winkelwerte angegeben, welche in der Endposition eingenommen werden sollen. Wie auf ein "Joint Goal" geplant wird zeigt der folgende C++ Code:

```
1 std::vector<double> group_joint_values;  
2 group.getCurrentState() -> copyJointGroupPositions(group.  
3     getCurrentState()  
4 -> getRobotModel() -> getJointModelGroup(group.getName()),  
5     group_joint_values);  
6 group_joint_values[0] = -0.700;  
7 group.setJointValueTarget(group_joint_values);  
8 moveit::planning_interface::MoveGroup::Plan my_plan;  
success = group.plan(my_plan);
```

Listing 3.2: Planen auf ein Joint Goal

In diesem Beispiel werden in Zeile zwei bis Zeile vier die aktuellen Winkelwerte der Gelenke kopiert. Darauf folgend wird ein Winkelwert verändert und mit der Funktion "setJointValueTarget()" als neues Ziel gesetzt.

### Position Target planen

Wenn die Orientierung des Endeffektors nicht von Bedeutung ist, kann auf ein "Position Target" geplant werden. Bei dem "Position Target" wird nur die Koordinate im Raum angegeben, zu der sich der Endeffektor bewegen soll. Im folgenden C++ Code wird beispielhaft auf ein "Position Target" geplant:

```

1  tf2 :: Vector3 distance;
2  distance[0] = conversion_factor * (transformStamped_left_hand_1 .
3      transform . translation .x - 0.15);
4  distance[1] = conversion_factor * (transformStamped_left_hand_1 .
5      transform . translation .z*(-1.0));
6  distance[2] = conversion_factor * (transformStamped_left_hand_1 .
7      transform . translation .y - 0.15);
8
9  group . setPositionTarget(distance[0], distance[1], distance[2]);
moveit :: planning_interface :: MoveGroup :: Plan my_plan;
success = group . plan(my_plan);

```

Listing 3.3: Planen auf ein Position Target

In dem Beispiel werden der Funktion "setPositionTarget()" die drei Komponenten, der Koordinate im Raum, als Gleitkommazahlen übergeben und diese Koordinate als neues Ziel gesetzt.

## Plan ausführen

Um einen Plan auszuführen, werden die Funktionen "move()" und "execute()" bereitgestellt. Um einen vorher durch die Funktion "plan()" berechneten Plan auszuführen wird die Funktion "execute()" verwendet. Die Funktion "move()" berechnet erst einen Plan und führt diesen direkt aus.

```

1  moveit :: planning_interface :: MoveGroup :: Plan my_plan;
2
3
4  // entweder
5  success = group . plan(my_plan);
6  group . execute(my_plan);
7
8  // oder
9  group . move(my_plan);

```

Listing 3.4: Plan ausführen



# 4. Implementation des Gestenerkennungssystems mit ROS

In diesem Kapitel wird die gesamte Implementation, des Gestenerkennungssystems, beschrieben. Das Kapitel ist so gegliedert, dass die Abfolge der Schritte im Text auch die einzuhaltende Implementationsreihenfolge darstellt. Bei nicht einhalten der Implementationsreihenfolge sind Konflikte zwischen Paketen nicht auszuschließen.

## 4.1. Vorbereitung

Die vorbereitenden Maßnahmen werden in diesem Abschnitt beschrieben.

### 4.1.1. Catkin Workspace erstellen

Um veränderte und eigene Pakete einfach zu compilieren und zu bauen, ist es notwendig einen Catkin-Workspace zu erstellen. Die folgenden Befehle sind in einem Terminal einzugeben, um einen Catkin-Workspace zu erstellen und initial zu bauen:

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

Nachdem der Catkin-Workspace erstellt und gebaut wurde, sollte die Ordnerstruktur wie in Abbildung 4.1 dargestellt aussehen. Im Ordner "src" wurde eine Datei mit dem Namen "CMake-Lists.txt" generiert. Im Ordner "devel" wurden verschiedene Setup-Dateien generiert. Um den Pfad des Catkin-Workspace in die Umgebungsvariablen aufzunehmen, muss der folgende Befehl ausgeführt werden:

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

Weitergehende Informationen zu "Catkin" sind im ROS Wiki<sup>8</sup> zu finden.

---

<sup>8</sup><http://wiki.ros.org/catkin>

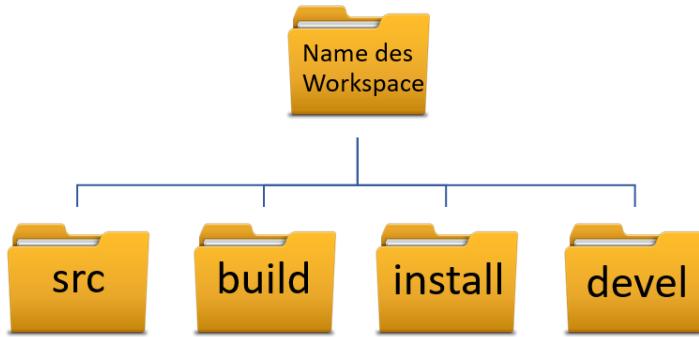


Abbildung 4.1.: Ordnerstruktur Catkin-Workspace

### 4.1.2. Abhangigkeiten

Die Komponenten des Gestenerkennungssystems haben Abhangigkeiten zu anderen Paketen. Diese Abhangigkeiten mussen vor der Implementation, des Gestenerkennungssystems, erfullt sein. Um diese Abhangigkeiten zu erfullen, werden mit den folgenden Befehlen die benotigten Pakete installiert:

```
$ sudo apt-get install git build-essential python libusb-1.0-0-dev freeglut3-dev openjdk-8-jdk  
$ apt-get install doxygen graphviz mono-complete
```

## 4.2. Installation der Komponenten

In diesem Abschnitt werden die Komponenten des Gestenerkennungssystems installiert. Um die Installation von "OpenNI" und der dazu gehorigen Module von anderen Paketen getrennt zu halten, wird ein neuer Ordner erstellt. Der folgende Befehl erstellt den neuen Ordner:

```
$ mkdir -p ~/kinect
```

### 4.2.1. OpenNI installieren

Um "OpenNI" zu installieren wird erst ein Repository, in den Ordner "kinect", geklont und "OpenNI" aus dieser Kopie heraus installiert. Die folgenden Befehle fuhren dies aus:

```
$ cd ~/kinect
$ git clone https://github.com/OpenNI/OpenNI.git
$ cd OpenNI
$ git checkout Unstable-1.5.4.0
$ cd Platform/Linux/CreateRedist
$ chmod +x RedistMaker
$ ./RedistMaker
$ cd ../Redist/OpenNI-Bin-Dev-Linux-x64-v1.5.4.0
$ sudo ./install.sh
```

Das Modul "SensorKinect" für "OpenNI" wird durch analoges Vorgehen installiert. Die folgenden Befehle sind auszuführen:

```
$ cd ~/kinect
$ git clone https://github.com/avin2/SensorKinect
$ cd SensorKinect
$ cd Platform/Linux/CreateRedist
$ chmod +x RedistMaker
$ ./RedistMaker
$ cd ../Redist/Sensor-Bin-Linux-x64-v5.1.2.1
$ chmod +x install.sh
$ sudo ./install.sh
```

## 4.2.2. Ros-Kinetic-OpenNI installieren

In diesem Schritt wird die Sammlung von Paketen, die für den Zugriff auf die Daten der "Kinect" unter ROS notwendig sind, installiert. Um die Installation durchzuführen ist der folgende Befehl auszuführen:

```
$ sudo apt-get install ros-kinetic-openni*
```

## 4.2.3. NITE installieren

Die Middleware "NITE" wird mit folgenden Befehlen installiert und bei "OpenNI" als Modul registriert:

#### *4. Implementation des Gestenerkennungssystems mit ROS*

---

```
$ cd ~/kinect  
$ git clone https://github.com/arnaud-ramey/NITE-Bin-Dev-Linux-v1.5.2.23  
$ cd NITE-Bin-Dev-Linux-v1.5.2.23/x64  
$ chmod +x install.sh  
$ sudo ./install.sh
```

#### **4.2.4. OpenNI Tracker installieren**

Das Paket "OpenNI Tracker" wird von einem Repository in den Catkin-Workspace kopiert. Anschließend wird der Catkin-Workspace erneut gebaut und das Paket "OpenNI Tracker" installiert. Diese Schritte werden mit folgenden Befehlen ausgeführt:

```
$ cd ~/catkin_ws/src  
$ git clone https://github.com/ros-drivers/openni_tracker.git  
$ cd ~/catkin_ws  
$ catkin_make  
$ catkin_make install
```

### **4.3. Start und Test des Gestenerkennungssystems**

In diesem Abschnitt wird beschrieben wie die Komponenten des Gestenerkennungssystems gestartet werden und das Gestenerkennungssystem getestet wird. Bevor die Komponenten gestartet werden können, muss die "Kinect", über einen freien USB 3.X Port, mit dem Rechner verbunden werden. Zusätzlich muss sichergestellt werden, dass die virtuelle Maschine Zugriff auf die "Kinect" hat. Dies ist der Fall, wenn die virtuelle Maschine die "Kinect" als angeschlossene Hardware erkennt.

#### **4.3.1. Starten der Komponenten**

In einem Terminal wird der folgende Befehl ausgeführt, um die Knoten zu starten, welche die Daten der "Kinect" in ROS veröffentlichen:

```
$ roslaunch openni_launch openni.launch camera:=openni
```

Der Knoten "OpenNI Tracker", der die Daten des erkannten Skeletts in ROS veröffentlicht, wird in einem zweiten Terminal mit folgenden Befehl gestartet:

```
$ roslaunch openni_tracker openni_tracker
```



Abbildung 4.2.: Psi Pose

Wenn eine Person, nach Start des Knotens "OpenNI Tracker", vor die "kinect" tritt, dann gibt der Knoten den Text "New User 1" im Terminal aus. Die Person muss nun die "Psi-Pose", wie in Abbildung 4.2 dargestellt, einnehmen, um die Kalibrierung des Skeletts zu starten. Die optimale Entfernung für die Erkennung des Skeletts sind 2,5 Meter. Das die Pose korrekt eingenommen wurde, wird mit dem Text "Pose Psi detected for user 1" quittiert. Der Knoten startet nun die Kalibrierung und gibt dies mit dem Text "Calibration started for user 1" an. Wenn die Kalibrierung abgeschlossen wurde, gibt der Knoten den Text "Calibration complete, start tracking user 1" aus und beginnt die Positionsdaten des Skeletts zu veröffentlichen.

### 4.3.2. Mit Rviz visualisieren und überprüfen

Um die Funktion des Gestenerkennungssystems zu überprüfen, wird hier mit dem Tool "Rviz" gearbeitet. Im folgenden Abschnitt wird beschrieben wie eine "Point Cloud" und das, von "OpenNI Tracker" erkannte, Skelett mit TF's in "Rviz" visualisiert wird.

"Rviz" wird mit folgenden Befehl gestartet:

```
$ rosrun rviz rviz
```

In Abbildung 4.3 ist der Bereich Displays, des Fensters von "Rviz", abgebildet. Im Bereich "Displays" wird als "Fixed Frame" die Auswahlmöglichkeit "openni\_link" eingestellt. Im Fenster, unter dem Bereich "Displays", wird der Button "Add" angeklickt. In Abbildung 4.4 ist das sich öffnende Fenster abgebildet. In diesem Fenster wird der Reiter "By topic" ausgewählt. Unter diesem Reiter wird in der Liste, oben im Fenster, unter "openni/depth/image" die Zeile "Depthcloud" angeklickt und anschließend mit einem Klick auf "OK" bestätigt. Im Bereich "Displays" wurde eine Zeile mit dem Text "DepthCloud" hinzugefügt. Durch einen Klick auf das Dreieck, links neben dem Text, wird die Zeile, wie in Abbildung 4.5 abgebildet, erweitert. Hier wird als "Color Image Topic" die Auswahlmöglichkeit "openni/rgb/image\_color" ausgewählt, um der "DepthCloud" Farbinformationen hinzuzufügen. Nachdem die "DepthCloud" eingerich-

#### 4. Implementation des Gestenerkennungssystems mit ROS

---

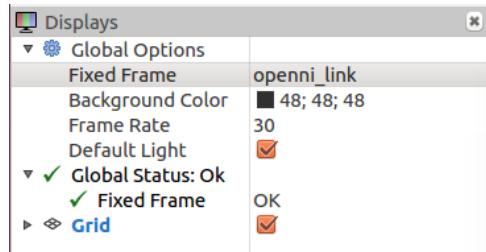


Abbildung 4.3.: Rviz - Fixed Frame Auswahl

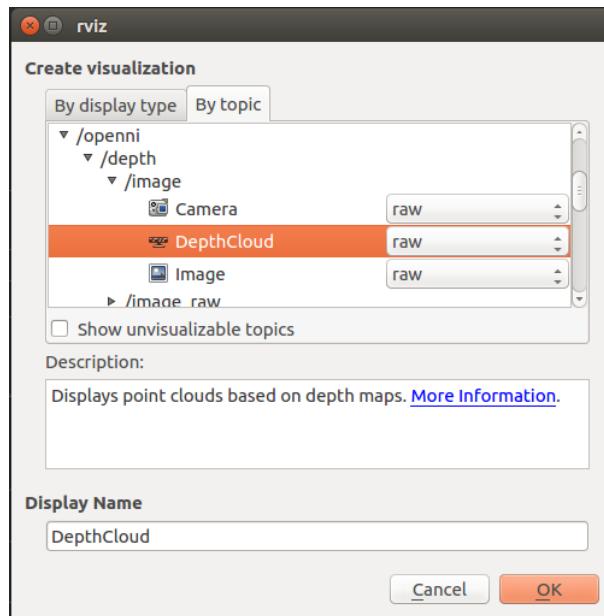


Abbildung 4.4.: Rviz - DepthCloud hinzufügen

tet wurde, wird nun erneut auf den Button "Add" geklickt. Wie in Abbildung 4.6 abgebildet wird, unter dem Reiter "By display type", die Zeile "TF" markiert und mit einem Klick auf "OK" bestätigt. Im Bereich "Displays" wurde, wie in Abbildung 4.7 zu sehen, eine Zeile mit dem Text "TF" hinzugefügt. Wenn, wie in Abbildung 4.8 abgebildet, die "DepthCloud" richtig dargestellt wird und das erkannte Skelett korrekt positioniert ist, ist die korrekte Funktion des Gestenerkennungssystems gegeben.

#### 4.3. Start und Test des Gestenerkennungssystems

---

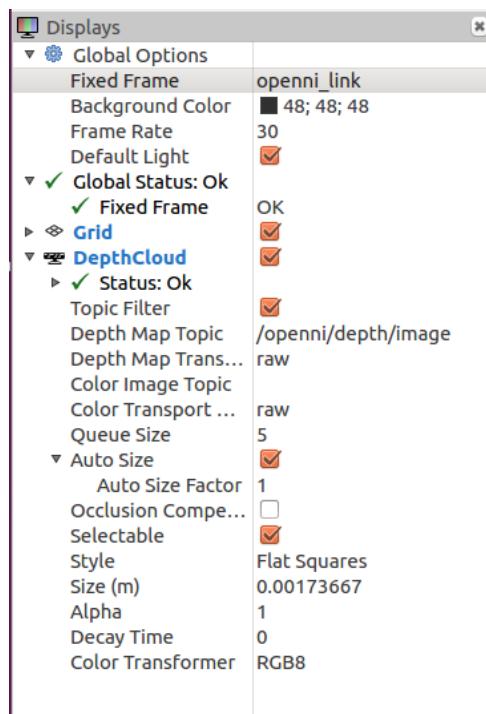


Abbildung 4.5.: Rviz - Color Image Topic Auswahl

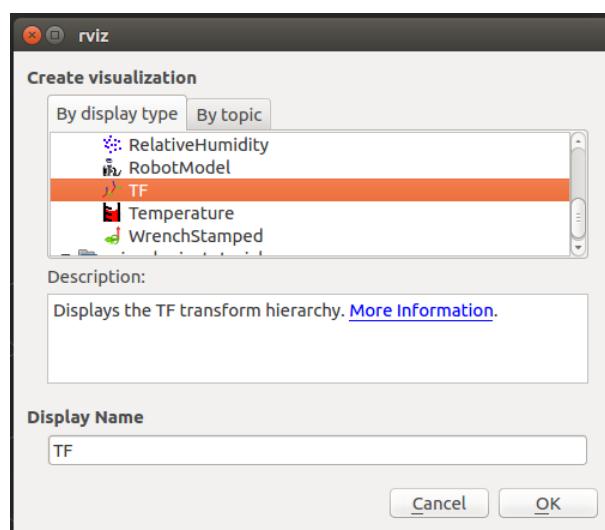


Abbildung 4.6.: Rviz - TF hinzufügen

#### 4. Implementation des Gestenerkennungssystems mit ROS

---

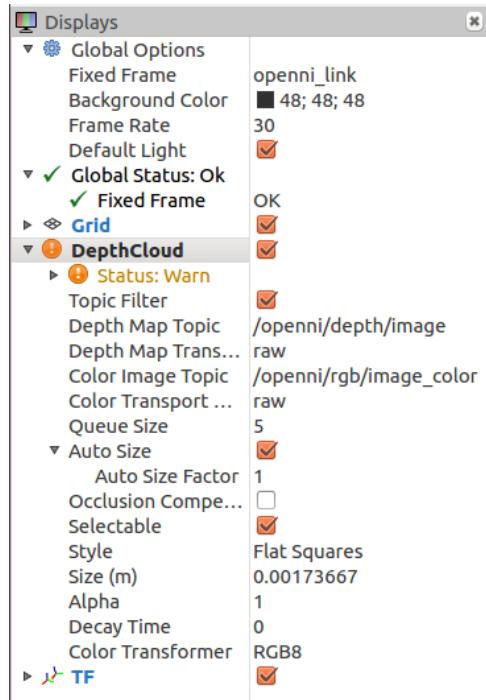


Abbildung 4.7.: Rviz - Displays Endergebnis

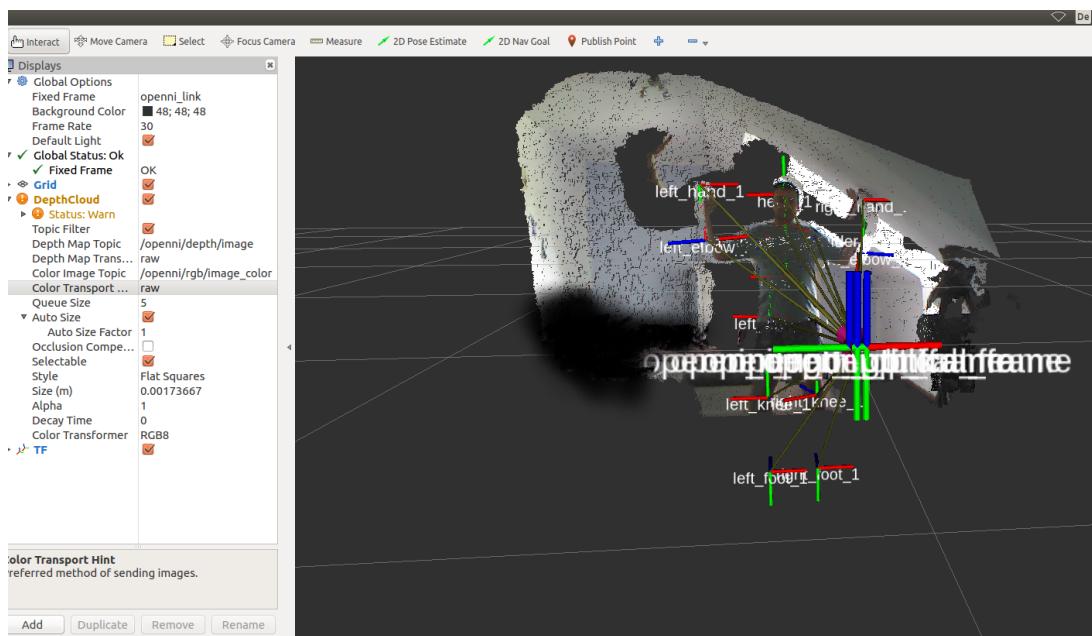


Abbildung 4.8.: Rviz - Visualisierung von DepthCloud und Skelett

# **5. Implementation von Gestensteuerungen basierend auf dem Gestenerkennungssystem**

In diesen Kapitel wird auf die Entwicklung der Pakete "turtlesim\_gesture\_control" und "roboticarm\_gesture\_control" eingegangen. Innerhalb dieser zwei Pakete wurden Gestensteuerungen implementiert. Für die Implementation der Gestensteuerungen wurde das in dieser Arbeit entwickelte Gestenerkennungssystem verwendet.

## **5.1. Vorbereitung**

In diesem Abschnitt wird auf die Vorbereitungen eingegangen, welche für die Entwicklung der zwei Pakete nötig waren.

### **5.1.1. Installation MoveIt!**

Für die Entwicklung des Paketes "roboticarm\_gesture\_control" wurde das Framework "MoveIt!" verwendet. Um "MoveIt!" zu installieren, muss der folgende Befehl in einem Terminal ausgeführt werden:

```
$ sudo apt-get install ros-kinetic-moveit
```

### **5.1.2. Einrichtung der Pakete rob\_arm\_small und rob\_arm\_small\_hw\_interface**

Zur Ansteuerung des Roboterarms "rob\_arm\_small" wurden die Pakete "rob\_arm\_small" und "rob\_arm\_small\_hw\_interface" verwendet. Diese Pakete wurden in einer vorangegangenen Bachelorarbeit[18], von Christian Waldner, entwickelt.

Um diese Pakete getrennt vom Gestensteuerungssystem zu halten, wurde ein weiterer Catkin-Workspace mit dem Namen "ROS\_ws" erstellt. Folgende Befehle führen dies aus:

```
$ mkdir -p ~/ROS_ws/src  
$ cd ~/ROS_ws/  
$ catkin_make
```

## *5. Implementation von Gestensteuerungen basierend auf dem Gestenerkennungssystem*

---

In diesen Catkin-Workspace wurden die beiden Pakete kopiert. Der Catkin-Workspace wird mit folgenden Befehlen erneut gebaut und in den Umgebungsvariablen aufgenommen:

```
$ catkin_make  
$ source devel/setup.bash
```

### **5.1.3. Erstellung eines ROS Paketes**

Um ein neues Paket in ROS zu entwickeln, muss dieses erst einmal angelegt werden. In diesem Abschnitt wird an dem Paket "turtlesim\_gesture\_control" beispielhaft erklärt, wie ein neues Paket in einem Catkin-Workspace angelegt wird. Zum erstellen des Paketes müssen folgende Befehle ausgeführt werden:

```
$ cd ~/catkin_ws/src  
$ catkin_create_pkg turtlesim_gesture_control std_msgs roscpp
```

Nach der Erstellung eines neuen Paketes, muss der Catkin-Workspace erneut gebaut werden. Die folgenden Befehle bauen den Catkin-Workspace neu:

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

Die in Abbildung 5.1 dargestellte Ordnerstruktur wurde angelegt. Zusätzlich wurden die beiden Dateien "CMakeLists.txt" und "package.xml" im Ordner "turtlesim\_gesture\_control" erstellt. Die Dateien "package.xml" und "CMakeLists.txt" werden hier nicht näher betrachtet, da die Änderungen an diesen nicht essentiell für diese Arbeit sind. Informationen zum Umgang mit den beiden Dateien sind im ROS Wiki<sup>89</sup> zu finden.

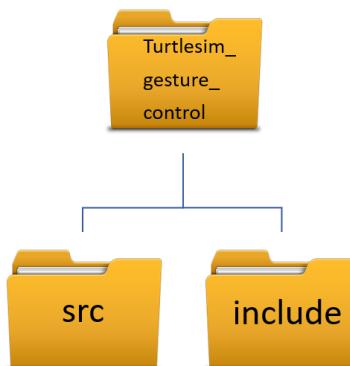


Abbildung 5.1.: ROS Paket Ordnerstruktur

---

<sup>8</sup><http://wiki.ros.org/ROS/Tutorials/CreatingPackage>

<sup>9</sup><http://wiki.ros.org/catkin/CMakeLists.txt>

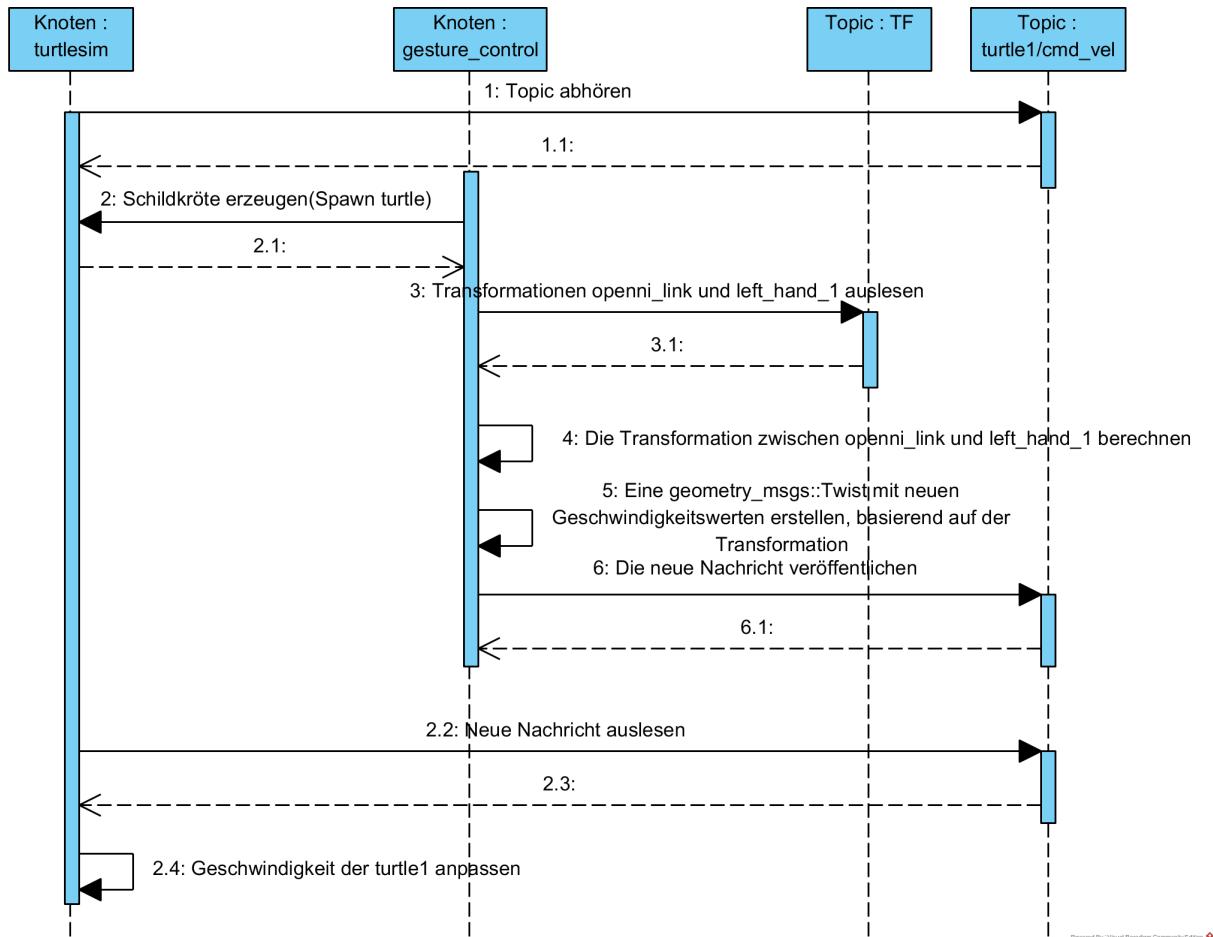


Abbildung 5.2.: turtlesim\_gesture\_control Sequenz-Diagramm

## 5.2. Paket turtlesim\_gesture\_control

In diesem Abschnitt wird auf das Paket "turtlesim\_gesture\_control" eingegangen. Die Funktion des Paketes wird hier beschrieben und die Verwendung des Gestenerkennungssystems dabei näher betrachtet. Es wird nicht auf die Grundlagen der Programmierung in ROS eingegangen. Grundlegende Informationen zur Programmierung in ROS sind im ROS Wiki<sup>10</sup> zu finden. Durch das Paket "turtlesim\_gesture\_control" wurde eine Gesteuerung für den "Turtle Simulator" in ROS implementiert. Hierfür wurde, in der Programmiersprache C++, ein Knoten Programmiert, der die Gesteuerung für den "Turtle Simulator" bereitstellt.

In der Abbildung 5.2 ist, mit einem Sequenz-Diagramm, eine grobe Übersicht über den Funktionsablauf der Gesteuerung gegeben. Im weiteren wird auf die essentiellen Passagen im Quellcode, des Knotens "gesture\_control", näher eingegangen.

<sup>10</sup><http://wiki.ros.org/ROS/Tutorials>

### 5.2.1. Der Knoten `gesture_control`

Vor Beginn der Hauptschleife des Knotens, wird eine neue Schildkröte im "Turtle Simulator" erzeugt. Zusätzlich wird ein "Publisher", welcher unter dem Topic "turtle1/cmd\_vel" veröffentlicht, erzeugt. Um das Topic "TF" abzuhören wird ein "TransformListener" erzeugt. Der Code im Listing A.2 führt dies aus.

```
1 tf2_ros::Buffer tfBuffer;
2 tf2_ros::TransformListener tfListener(tfBuffer);
```

Listing 5.1: Erzeugen TransformListener

Im Listing 5.2 ist die Hauptschleife des Knotens aufgeführt. Im "Try-Catch-Block", in den Zeilen 3-14, wird versucht eine Transformation, zwischen den "Frames", "openni\_link" und "left\_hand\_1", zu berechnen. Wenn dies gelingt, wird das Ergebnis in einer Nachricht, vom Typ "TransformStamped", gespeichert. Diese Transformation enthält die Position und Orientierung des "Frames", "left\_hand\_1", relativ zu der Position des "Frames", "openni\_link". Der "Frame", "openni\_link", stellt die Position und Orientierung der "Kinect" dar. Der "Frame", "left\_hand\_1", stellt die Position und Orientierung der rechten Hand, von der Person die durch "OpenNI Tracker" erkannt wurde, dar. In den Zeilen 16-50 wird, aufgrund der relativen Position des "Frames", "left\_hand\_1", eine Nachricht vom Typ "geometry\_msgs::Twist" angelegt und diese mit Werten, für die Linear- und die Rotationsgeschwindigkeit, gefüllt. Beispielsweise wird in Zeile 19-21 der X-Anteil, der Lineargeschwindigkeit, auf 1.0 gesetzt, falls der Y-Anteil der Translation, die Teil der Transformation ist, größer als 0.2 ist. Dies bedeutet praktisch, dass wenn die Person, die von der "Kinect" erkannt wurde, ihre rechte Hand um 0.2 Meter rechts von der Kameraachse hat, dann wird der Wert in der Nachricht wie beschrieben gesetzt. In Zeile 52 wird die Nachricht, unter dem Topic "turtle1/cmd\_vel", veröffentlicht.

```
1 * and buffers them for up to 10 seconds*
2 tf2_ros::Buffer tfBuffer;
3 tf2_ros::TransformListener tfListener(tfBuffer);
4
5 ros::Rate rate(10.0);
6 while (node.ok()){
7     geometry_msgs::TransformStamped transformStamped;
8     try{
9         /* here the transform between openni_link and left_hand_1 is calculated
10          *openni_link -> kinect , left_hand_1 -> right hand of person*/
11         transformStamped = tfBuffer.lookupTransform("openni_link", "left_hand_1",
12                                         ros::Time(0));
13     }
14     catch (tf2::TransformException &ex) {
15         ROS_WARN("%s", ex.what());
16
17         ros::Duration(1.0).sleep();
18         continue;
19     }
20 }
```

```

19 }
20
21     geometry_msgs::Twist vel_msg;
22     /*if right hand is right of the kinect from the perspective of the person
23      the
24      *turtle will move forward*/
25     if(transformStamped.transform.translation.y > 0.2)
26     {
27         vel_msg.linear.x = 1.0;
28     }
29     else{
30         /*if right hand is left of the kinect from the perspective of the
31          person the
32          *turtle will move backward*/
33         if(transformStamped.transform.translation.y < -0.2){
34             vel_msg.linear.x = - 1.0;
35         }
36         else{
37             /*if right hand is in front of the kinect from the perspective of the
38              person the
39              *turtle will stop*/
40             vel_msg.linear.x = 0.0;
41         }
42     }
43     /*if right hand is above the kinect from the perspective of the person
44      the
45      *turtle will turn left*/
46     if(transformStamped.transform.translation.z > 0.3){
47         vel_msg.angular.z = 1.0;
48     }
49     else{
50         /*if right hand is below the kinect from the perspective of the person
51          the
52          *turtle will turn right*/
53         if(transformStamped.transform.translation.z < -0.3){
54             vel_msg.angular.z = - 1.0;
55         }
56         else{
57             /*if right hand is in front of the kinect from the perspective of the
58              person the
59              *turtle will not turn*/
60             vel_msg.angular.z = 0.0;
61         }
62     }

```

Listing 5.2: Hauptschleife vom Knoten gesture\_control

## 5.2.2. Starten der Gestensteuerung

Zum Start der Gestensteuerung, muss das Gestenerkennungssystem zuvor gestartet worden sein. Um die benötigten Knoten zu starten, wurde eine "Launch-Datei" erstellt. Die Knoten

## 5. Implementation von Gestensteuerungen basierend auf dem Gestenerkennungssystem

---

können auch manuell gestartet werden, doch mit einer "Launch-Datei" muss nur ein Befehl ausgeführt werden, um alle benötigten Knoten zu starten. Diese Datei ist in dem Paket "turtlesim\_gesture\_control" enthalten. Im Listing 5.3 ist die "Launch-Datei" aufgeführt. Mit dem folgenden Befehl wird die "Launch-Datei" ausgeführt:

```
$ rosrun turtlesim_gesture_control start_demo.launch
```

```
1 <launch>
2     <!-- Turtlesim Node-->
3
4     <node pkg="turtlesim" type="turtlesim_node" name="sim"/>
5     <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>
6
7     <!-- Axes -->
8     <param name="scale_linear" value="2" type="double"/>
9     <param name="scale_angular" value="2" type="double"/>
10
11    <!-- gesture_control Node -->
12    <node pkg="turtlesim_gesture_control" type="gesture_control"
13        name="gest_cntrl" />
14
15
16 </launch>
```

Listing 5.3: Launch-Datei - start\_demo.launch

In der vierten Zeile wird der "Turtle Simulator"-Knoten gestartet, in welchem die Schildkröte, auf der grafischen Oberfläche, gesteuert wird. In Zeile fünf wird ein Knoten gestartet, welcher die optionale Steuerung der Schildkröte, mit den Pfeiltasten der Tastatur, ermöglicht. Der Knoten "gesture\_control" wird in Zeile 12 gestartet. Die grafische Oberfläche des "Turtle Simulators" ist, mit der vom Knoten "gesture\_control" erzeugten Schildkröte darauf, in der Abbildung 5.3 dargestellt. Nachdem der Start aller Knoten erfolgreich war, kann die Schildkröte durch Gesten gesteuert werden. Hierbei ist zu beachten, dass nur der "User 1", der von "OpenNI Tracker" erkannt wurde, die Schildkröte steuern kann. Falls kein "User 1" erkannt wurde oder Probleme auftreten, kann der Knoten "openni\_tracker" neugestartet werden.

### 5.2.3. Anleitung zur Gestensteuerung

Um die Schildkröte im "Turtle Simulator" zu steuern, muss die steuernde Person ihre rechte Hand, relativ zur Kameraachse der "Kinect", im Raum bewegen. Wenn die Schildkröte sich vorwärts bewegen soll, dann muss die Person ihre rechte Hand mehr als 0.2 Meter rechts von der Kameraachse positionieren. Für eine Rückwärtsbewegung muss die Hand mehr als 0.2 Meter links sein. Damit die Schildkröte stoppt, muss die Hand näher als 0.2 Meter, in der Horizontalen, an der Kameraachse sein. Um die Schildkröte gegen den Uhrzeigersinn drehen zu lassen, muss die Hand mehr als 0.3 Meter über der Kameraachse sein. Für die Rotation im Uhrzeigersinn, muss die Hand mehr als 0.3 Meter unter der Kameraachse sein. Um die Rotation zu stoppen,

## 5.2. Paket turtlesim\_gesture\_control

---

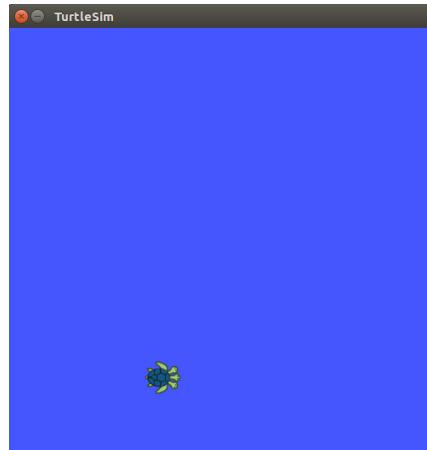


Abbildung 5.3.: Turtle Simulator Oberfläche

muss die Hand näher als 0.3 Meter, in der Vertikalen, an der Kameraachse sein. In der Abbildung 5.4 ist die Steuerung zusätzlich noch einmal grafisch dargestellt.

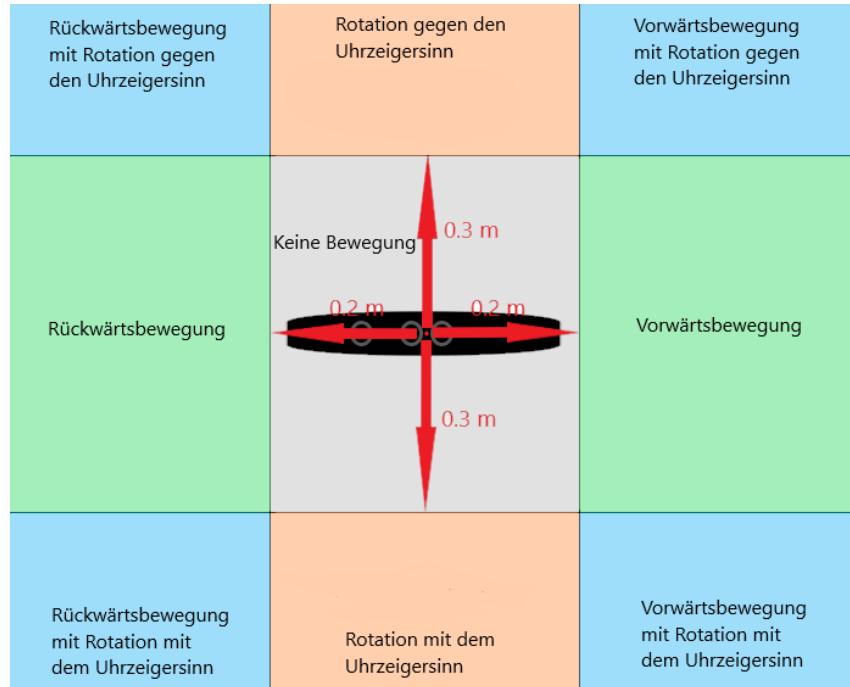


Abbildung 5.4.: Gestensteuerung Anleitung

## 5.3. Paket roboticarm\_gesture\_control

Mit "roboticarm\_gesture\_control" wurde eine zweites Paket, basierend auf dem Gestenerkennungssystem, entwickelt. Dieses Paket implementiert eine Gestensteuerung, für den Roboterarm "rob\_arm\_small".

### 5.3.1. Der Knoten roboticarm\_gesture\_control

Der Knoten "roboticarm\_gesture\_control" wird in diesem Abschnitt beschrieben. Es wird auf die Funktion und den Aufbau des Funktionsablaufes eingegangen. Die Hauptschleife des Knotens, die den zentralen Algorithmus enthält, wird direkt an Passagen im Quellcode erklärt. Der Funktionsablauf, außerhalb der Hauptschleife, der nicht die zentrale Funktion darstellt, wird im Text, ohne Quellcodeverweise, erklärt.

In der Abbildung 5.5 ist, mit einem Sequenz-Diagramm, eine grobe Übersicht über den Funktionsablauf gegeben.

Zu Beginn erzeugt der Knoten Objekte und Schnittstellenobjekte, für den Zugriff auf Funktionen, vom "MoveIt!" Framework. Über die Schnittstellenobjekte werden die Toleranzen, für Ziele des Endeffektors, gesetzt. Die maximale Zeit, die zur Planung eines Ziels zur Verfügung steht, wird ebenfalls über das Schnittstellenobjekt festgesetzt. Nachdem die Objekte, die "MoveIt!" betreffen, erstellt wurden, wird ein "Transformlistener", der das Topic "TF" abhört, erstellt. Als nächstes werden die Transformationen zwischen linker Schulter und linkem Ellbogen sowie linkem Ellbogen und linker Hand berechnet. Aus den beiden Transformationen wird die Armlänge des Nutzers ermittelt. Aus der ermittelten Armlänge und der Länge des Roboterarmes wird ein Umrechnungsfaktor, für die Zielkoordinaten des Endeffektors, berechnet.

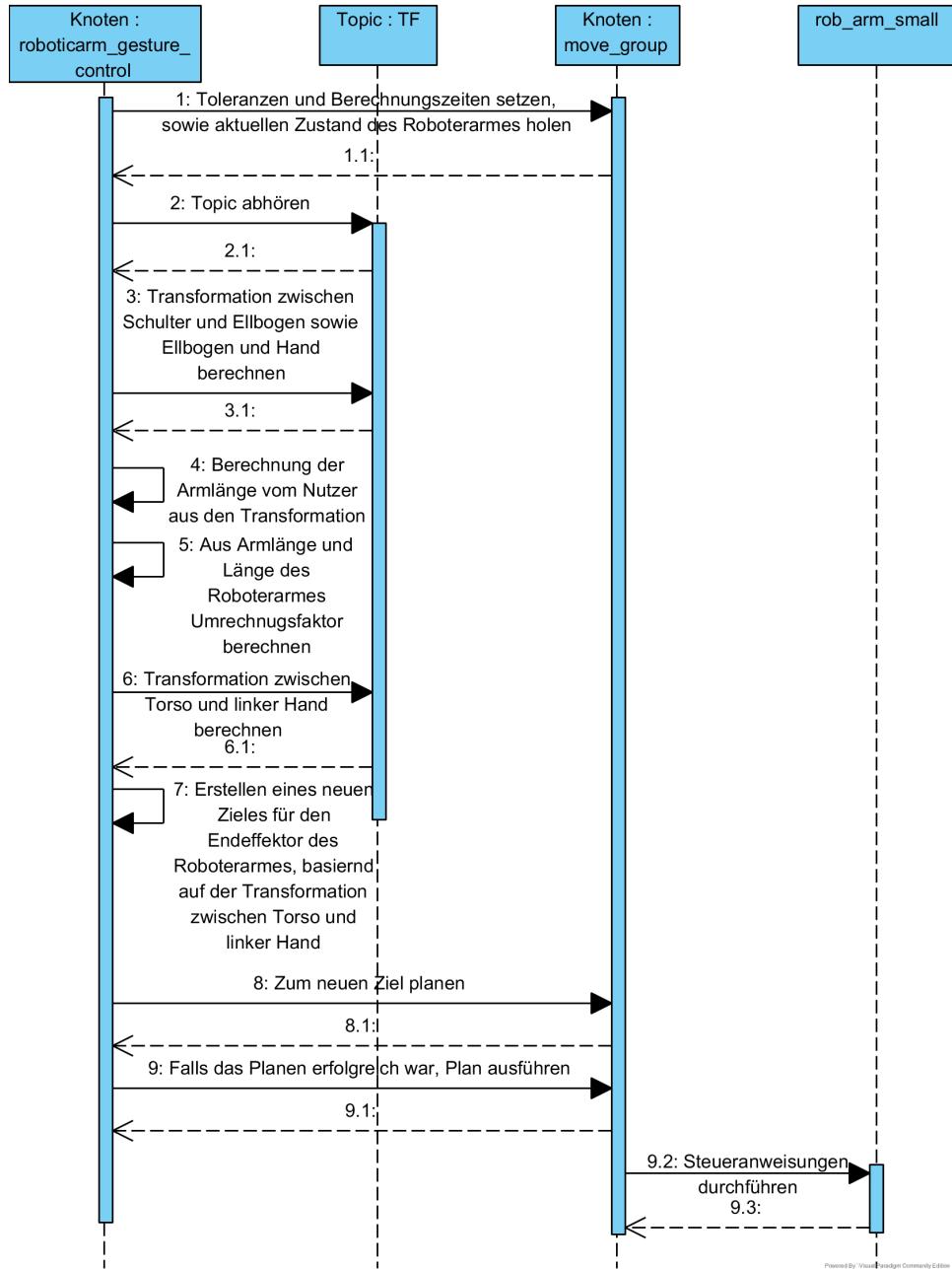


Abbildung 5.5.: `roboticarm_gesture_control` - Sequenzdiagramm

## 5. Implementation von Gestensteuerungen basierend auf dem Gestenerkennungssystem

---

Nach diesen vorbereitenden Maßnahmen, beginnt die Hauptschleife des Knotens. Die Hauptschleife ist im Listing 5.4 aufgeführt. Im "Try-Catch-Block", in den Zeilen 5-21, wird versucht eine Transformation zwischen den "Frames", "torso\_1" und "left\_hand\_1", zu berechnen. Statt des "Frames" der linken Schulter als Referenz, wurde der "Frame" des Torsos als Referenz gewählt. Der Grund hier ist, dass die Orientierung des "Frames" der Schulter sich verändert, wenn der Oberarm bewegt wird. Somit ist die Schulter, da sie nicht statisch in ihrer Orientierung ist, als Referenz nicht geeignet. Die berechnete Transformation wird in einer Nachricht, vom Typ "TransformStamped", gespeichert. In dieser Transformation ist die Position der linken Hand, relativ zu dem Torso, enthalten. In den Zeilen 24-26 wird die Zielkoordinate, für den Endeffektor, berechnet und in einem Vektor gespeichert. Bei der Berechnung wurde der Vektor, der vom Torso zur linken Hand zeigt, so verschoben, sodass dieser nun die Position der linken Hand, relativ zur linken Schulter, darstellt. Zusätzlich wurde, mit einem Umrechnungsfaktor, der Vektor skaliert. Dieser Umrechnungsfaktor bezieht die Armlänge des Nutzers mit ein. Somit entspricht die Länge des Vektors, bei ausgestreckten Arm des Nutzers, der maximalen Reichweite des Roboterarmes. Dies stellt sicher, dass sich die Zielkoordinate, für den Endeffektor, nie außerhalb der Reichweite des Roboterarmes befindet. Wenn die Hand des Nutzers mit dem Torso, über die Schulter, einen rechten Winkel bildet, dann soll die Zielkoordinate mittig im Operationsbereich, des Roboterarmes, liegen. Hierfür wurde, in Zeile 35, der Vektor noch passend rotiert. In Zeile 29 wird die berechnete Zielkoordinate als neues Ziel, für den Endeffektor, gesetzt. In Zeile 33 wird ein Plan, zum erreichen der Zielkoordinate, erstellt. In den Zeilen 38-40 wird der errechnete Plan abgeändert, um den "Gripper" eine horizontale Position einnehmen zu lassen, da der "Gripper" sonst irgendeine Orientierung einnehmen würde. Dies soll das Aufnehmen von Gegenständen, mit dem "Gripper", erleichtern. Nach erneuter Berechnung des Plans, mit neuen Parametern, wird dieser in Zeile 45 ausgeführt.

```
1 ros :: Rate rate(60.0);  
2  
3 /* main loop */  
4  
5 while (node_handle.ok()) {  
6  
7     try{  
8  
9         transformStamped_left_hand_1 = tfBuffer.lookupTransform("torso_1", "  
10            left_hand_1",  
11            ros::Time(0));  
12  
13         /* optional Transform calculation to implement grab functionality of  
14            the gripper  
15             transformStamped_right_hand_1 = tfBuffer.lookupTransform("torso_1", "right_hand_1",  
16             ros::Time(0));  
17         */  
18     }  
19     catch (tf2::TransformException &ex) {
```

```

19         ROS_WARN( "%s" , ex.what() );
20         continue;
21     }
22
23     /* calculate new position target for endeffector */
24     distance[0] = conversion_factor * (transformStamped_left_hand_1.
25         transform.translation.x - 0.15);
26     distance[1] = conversion_factor * (transformStamped_left_hand_1.
27         transform.translation.z*(-1.0));
28     distance[2] = conversion_factor * (transformStamped_left_hand_1.
29         transform.translation.y - 0.15);
30
31     distance = distance.rotate(rotate, 0.800);
32     move_group_arm.setPositionTarget(distance[0], distance[1], distance[2])
33         ;
34
35     ROS_INFO_NAMED("planning", "x : %lf y: %lf z: %lf", distance[0],
36         distance[1], distance[2]);
37     ROS_INFO_NAMED("distance", "distance: %lf", distance.length());
38     bool succes = (move_group_arm.plan(my_plan) == moveit::
39         planning_interface::MoveItErrorCode::SUCCESS);
40
41     if(succes){
42
43         succes = false;
44         joint_group_positions = my_plan.trajectory_.joint_trajectory.points.
45             back().positions;
46         joint_group_positions[4] = -0.0;
47         move_group_arm.setJointValueTarget(joint_group_positions);
48         succes = (move_group_arm.plan(my_plan) == moveit::planning_interface
49             ::MoveItErrorCode::SUCCESS);
50
51     }
52
53     /* optional gripper control */
54     if(true){
55
56         // joint_group_positions[0] = -0.300;
57
58         // move_group_gripper.setJointValueTarget(joint_group_positions);
59
60         // move_group_gripper.move();
61     }
62     rate.sleep();

```

## 5. Implementation von Gestensteuerungen basierend auf dem Gestenerkennungssystem

---

```
63     }
64     return 0;
65
66 }
```

Listing 5.4: Hauptschleife vom Knoten roboticarm\_gesture\_control

### 5.3.2. Starten der Gestensteuerung

Zum Start der Gestensteuerung, muss das Gestenerkennungssystem zuvor gestartet worden sein. Zuerst wird der "move\_group" Knoten mit folgenden Befehl gestartet:

```
$ roslaunch rob_arm_small move_group.launch
```

Da der "move\_group" Knoten nach den Actionservern sucht, müssen diese zeitnah, mit folgenden Befehlen, gestartet werden:

```
$ rosrun rob_arm_small_hw_interface grip_command_server
$ rosrun rob_arm_small_hw_interface action_server
```

Für die Kommunikation mit dem Roboterarm muss der "hw\_interface\_node" Knoten gestartet werden. Folgender Befehl führt dies aus:

```
$ rosrun rob_arm_small_hw_interface hw_interface_node
```

Mit dem folgenden Befehl wird der Knoten "roboticarm\_gesture\_control" gestartet:

```
$ roslaunch roboticarm_gesture_control roboticarm_gesture_control
.launch
```

Nachdem der Start aller Knoten erfolgreich war, kann der Roboterarm durch Gesten gesteuert werden. Hierbei ist zu beachten, dass nur der "User 1", der von "OpenNI Tracker" erkannt wurde, den Roboterarm steuern kann. Falls kein "User 1" erkannt wurde oder Probleme auftreten, kann der Knoten "openni\_tracker" neugestartet werden.

### 5.3.3. Anleitung zur Gestensteuerung

Um den Roboterarm zu steuern, muss die steuernde Person ihre rechte Hand, relativ zur rechten Schulter, im Raum bewegen. Falls ein Plan für eine Zielkoordinate gefunden wird, wird diese auch vom Endeffektor eingenommen. Diese eingenommene Position des Endeffektors, relativ zur Basis des Roboterarmes, entspricht immer der Position der rechten Hand, relativ zur rechten Schulter.

# 6. Diskussion

Im Zuge dieser Bachelorarbeit wurde ein Gestenerkennungssystem, mithilfe einer Stereokamera, in ROS entwickelt und implementiert. Dieses Ziel wurde, nach Schwierigkeiten mit veralteten Rechner und nicht aktueller ROS-Distribution, erfolgreich erreicht. Darüber hinaus wurde, mit zwei entwickelten ROS-Paketen, gezeigt, wie dieses Gestenerkennungssystem in ROS eingesetzt werden kann. Damit ist das Ziel, eine Grundlage für weitere Arbeiten und neue Interaktionsmöglichkeiten mit Robotersystemen zu schaffen, ebenfalls als erfüllt zu betrachten. In diesem Kapitel werden die erzielten Ergebnisse, die aufgetretenen Probleme und die gewonnenen Erkenntnisse zusammenfassend erläutert.

Das entwickelte und implementierte Gestenerkennungssystem ist funktionsfähig und bildet eine gute Grundlage, für zu entwickelnde Applikationen. Das System bindet die verwendete Stereokamera ein, greift auf die durch diese gelieferten Daten zu und verarbeitet die erhaltenen Daten. Die verarbeiteten Daten werden in ROS zur Verfügung gestellt und mit Hilfe dieser ein "Skelett-Tracking" ermöglicht. Das "Skelett-Tracking" wurde in den beiden entwickelten Paketen verwendet. Das System verfügt, über das verwendete "Skelett-Tracking" hinaus, über weitere Funktionen, wie zum Beispiel die Erkennung von bestimmten Bewegungsabläufen.

Zu Beginn der Entwicklung traten mehrere Probleme auf. Das erste Problem, welches viel Zeit kostete, wurde durch die virtuelle Maschine ausgelöst. Das Linux-Betriebssystem, welches in der virtuellen Maschine lief, konnte nur fehlerhafte Daten von der Stereokamera empfangen. Dies Problem wurde durch eine Anpassung der USB-Kompatibilitätseinstellungen, von USB 2.0 auf USB 3.0, gelöst. Doch auch nach dieser Anpassung, wurde die Verbindung zu Stereokamera in Einzelfällen noch unterbrochen. Um diese Art von Problemen zu vermeiden, sollte, bei weiteren Arbeiten oder Weiterentwicklungen am System, wenn möglich auf eine native Linux-Installation zurückgegriffen werden. Eine native Linux-Installation ist ebenfalls für die gesamte Systemperformance positiv. Als zweites, aber kleineres, Problem zeigte sich, dass Alter der verwendeten ROS-Distribution. Durch das, im Mai 2016, zurückliegende Veröffentlichungsdatum von "ROS Kinetic Kame" und durch die Tatsache, dass es bereits seit Mai 2018 eine neue ROS-Distribution gibt, wurden verschiedene Komponenten des Gestenerkennungssystems, für "ROS Kinetic Kame", nicht mehr weiterentwickelt. Aufgrund dieser Tatsache, musste auf nicht aktuelle Versionen der Komponenten zurückgegriffen werden. Dies führte zu einem Mehraufwand in der Entwicklung, aber nicht zu einer feststellbaren Einschränkung des Systems. Bei der entwickelten Gesteuerung, für den Roboterarm, traten Probleme bei der Bewegungsplanung auf. Es wird hier nur unzuverlässig ein Bewegungspfad gefunden. Das Problem wurde auf die Berechnung der inversen Kinematik und den verwendeten Pfadplaner, innerhalb von "MoveIt!", eingegrenzt. Der Versuch eine Lösung, innerhalb der verfügbaren Zeit, zu finden, war leider nicht erfolgreich. Mit der hier durchgeföhrten Eingrenzung des Problems, wird die Suche nach einer Lösung jedoch kein großes Problem darstellen.

## *6. Diskussion*

---

Als erste Erkenntnis steht am Ende dieser Arbeit, dass die natürliche Interaktion, des Menschen mit technischen Systemen, viel Potential ins sich trägt. Durch weitere Arbeiten in diesem Bereich, werden ganz sicher neue Ansätze entstehen und somit die Interaktion mit Robotern, neu definiert werden. Als zweite Erkenntnis steht, dass ROS, durch sein modulares Konzept und die daraus resultierende einfache Integration neuer Hardware und Software, eine passende Umgebung für weitere Entwicklungen in diesem Bereich darstellt.

### **6.1. Ausblick**

Diese Bachelorarbeit bietet eine Grundlage für weitere Entwicklungen, in dem Bereich der natürlichen Interaktion mit Robotersystemen. Die, zu Beginn der Arbeit, gesteckten Ziele wurden erreicht, jedoch sind weitere Arbeiten an dem System notwendig. Als erstes ist hier die Migration des Systems auf die aktuellste ROS-Distribution zu nennen. Im Zuge der Migration werden Aktualisierungen, an den Systemkomponenten, durchgeführt werden müssen. Gegebenenfalls wird es auch notwendig sein, dass Komponenten des Systems ausgetauscht werden müssen. Neben den Aktualisierungen der Software, bietet sich der Austausch der "Kinect", durch eine aktuellere und leistungsstärkere Stereokamera, an. Durch einen Austausch lassen sich, durch die höhere Genauigkeit aktueller Stereokameras, neue Funktionen implementieren. Durch neue Funktionen lassen sich wiederum neue Anwendungen, für das Gestenerkennungssystem, finden. Aus der Entwicklung der Gesteuerung, für den Roboterarm, ging hervor, dass die Bewegungsplanung noch nicht zufriedenstellend funktioniert. Hier empfiehlt es sich die Pakete "rob\_arm\_small" und "rob\_arm\_small\_hw\_interface" zu überarbeiten. Hier sollte das Hauptaugenmerk auf die Kinematik, die verwendeten Bewegungsplaner und auf die verwendete URDF-Datei gelegt werden.

Das Konzept, dass hinter dem Gestenerkennungssystem steht, ermöglicht es auch andere Sensoren, neben Stereokameras, zu integrieren. Somit können, in Weiterentwicklungen des Systems, verschiedene Sensoren miteinander kombiniert werden. Die Implementation einer Spracherkennung wäre beispielsweise eine Möglichkeit. Neben den genannten Weiterentwicklungen, gibt es die Möglichkeit, dass System in andere Projekte zu integrieren. Zum Beispiel eine Personenfolgefunktion, für mobile Robotersysteme, könnte damit integriert werden. Abgesehen von so spezifischen Verwendungen, wie einer Personenfolgefunktion, kann auch nur die Umgebungswahrnehmung, von bestehenden Robotersystemen, erweitert werden.

# A. Quellcode

```
1  /**
2   * @file      move_group_interface.cpp
3   * @author    Oliver Bosin
4   * @version   V1.0.0
5   * @date     13.06.2019
6   * @copyright 2011 – 2019 UniBw M – ETTI – Institute 4
7   * @brief    Node to control the roboticarm rob_arm_small
8   * @details  This Node listen to the tf of skeleton broadcasted by
9   *           openni_tracker and calculate a goal for the roboticarm
10  *           which is then executed by the framework MoveIt!
11  *
12  */
13  * 1. Listeners
14  *     - tfListener
15  *
16  * *****
17  * @par History:
18  *
19  * @details V1.0.0 13.06.2019 Oliver Bosin
20  *         - Initial Release
21  * *****
22  * @todo   Optimize tolerances , try asyncExecute() funktion
23  * *****
24  * @bug   none
25  * *****
26  */
27
28 #include <ros/ros.h>
29 #include <moveit/move_group_interface/move_group_interface.h>
30 #include <moveit/planning_scene_interface/planning_scene_interface.h>
31 #include <tf2/LinearMath/Vector3.h>
32 #include <moveit_msgs/DisplayRobotState.h>
33 #include <moveit_msgs/DisplayTrajectory.h>
34 #include <moveit_msgs/AttachedCollisionObject.h>
35 #include <moveit_msgs/CollisionObject.h>
36 #include <moveit_visual_tools/moveit_visual_tools.h>
37
38 /**
39  * @brief Main function for move_group_interface
40  * @details In this function tf-listener listen to tf broadcasted
41  *          by openni_tracker and calculate a goal for the
42  *          roboticarm which is then executed by MoveIt!
43  * @param argc: Non-negative value representing the number of
```

## A. Quellcode

---

```
44 *           arguments passed to the program from the
45 *           environment in which the program is run.
46 * @param    [in] argv: Pointer to an array of pointers to null-terminated
47 *           multibyte strings that represent the arguments
48 *           passed to the program from the execution
49 *           environment
50 * @retval   If the return statement is used, the return value is used as
51 *           the argument to the implicit call to exit().
52 *           This value can be:
53 *               @arg EXIT_SUCCESS [indicate successful termination]
54 *               @arg EXIT_FAILURE [indicate unsuccessful termination]
55 */
56 int main(int argc, char** argv){
57
58     ros::init(argc, argv, "move_group_interface");
59
60     ros::NodeHandle node_handle;
61
62     ros::AsyncSpinner spinner(1);
63
64     spinner.start();
65
66     /* MoveIt! declarations and initializations */
67     static const std::string PLANNING_GROUP_ARM = "roboter_arm";
68     static const std::string PLANNING_GROUP_GRIPPER = "gripper";
69
70     moveit::planning_interface::MoveGroupInterface move_group_arm(
71         PLANNING_GROUP_ARM);
72     moveit::planning_interface::MoveGroupInterface move_group_gripper(
73         PLANNING_GROUP_GRIPPER);
74
75     // Raw pointers are frequently used to refer to the planning group for
76     // improved performance.
77
78     const robot_state::JointModelGroup* joint_model_group_arm =
79
80         move_group_arm.getCurrentState() -> getJointModelGroup(
81             PLANNING_GROUP_ARM);
82
83     const robot_state::JointModelGroup* joint_model_group_gripper =
84
85         move_group_arm.getCurrentState() -> getJointModelGroup(
86             PLANNING_GROUP_GRIPPER);
87
87     move_group_arm.clearPoseTargets();
88     move_group_arm.setStartStateToCurrentState();
89     move_group_arm.setPoseReferenceFrame("base_link");
90
91     move_group_arm.setGoalPositionTolerance(0.02);
92     move_group_arm.setGoalJointTolerance(0.07);
93     move_group_arm.setPlanningTime(0.08);
```

```

91 move_group_gripper.clearPoseTargets();
92 move_group_gripper.setStartStateToCurrentState();
93 move_group_gripper.setPoseReferenceFrame("base_link");
94
95 move_group_gripper.setGoalTolerance(0.025);
96 move_group_gripper.setPlanningTime(0.08);
97
98 /* Here a TransformListener object is created.
99    * Once the listener is created it starts receiving tf2
100   transformations over the wire
101   * and buffers them for up to 10 seconds*/
102
103 tf2_ros::Buffer tfBuffer;
104 tf2_ros::TransformListener tfListener(tfBuffer);
105
106 /* Declarations of messages to store Transforms
107
108 geometry_msgs::TransformStamped transformStamped_left_hand_1;
109 geometry_msgs::TransformStamped transformStamped_right_hand_1;
110 geometry_msgs::TransformStamped transformStamped_shoulder_to_elbow;
111 geometry_msgs::TransformStamped transformStamped_elbow_to_hand;
112
113 /*loop to get transform from shoulder to elbow and elbow to hand for
114 arm length*/
115
116 while (node_handle.ok()){
117
118     try{
119
120         transformStamped_shoulder_to_elbow = tfBuffer.lookupTransform("left_shoulder_1", "left_elbow_1",
121                                         ros::Time(0));
122
123         transformStamped_elbow_to_hand = tfBuffer.lookupTransform("left_elbow_1", "left_hand_1",
124                                         ros::Time(0));
125     }
126
127     catch (tf2::TransformException &ex) {
128         ROS_WARN("%s", ex.what());
129
130         continue;
131     }
132     break;
133 }
134
135 /* convert from geometry_msgs::Vector3 to tf2::Vector3 so the use of
136 the length() function is possible*/
137
138 tf2::Vector3 shoulder_to_elbow_vector;
139 tf2::Vector3 elbow_to_hand_vector;
140 tf2::Vector3 rotate = {0.0,0.0,1.0};

```

## A. Quellcode

---

```
138 tf2::Vector3 distance;
139
140 shoulder_to_elbow_vector[0] = transformStamped_shoulder_to_elbow.
141     transform.translation.x;
142 shoulder_to_elbow_vector[1] = transformStamped_shoulder_to_elbow.
143     transform.translation.y;
144 shoulder_to_elbow_vector[2] = transformStamped_shoulder_to_elbow.
145     transform.translation.z;
146
147 /* calculate armlength of user */
148
149 double armlength = shoulder_to_elbow_vector.length() +
150     elbow_to_hand_vector.length();
151
152 /* calculate conversion factor for position target */
153
154 double conversion_factor = 0.38 / armlength;
155
156
157 moveit::planning_interface::MoveGroupInterface::Plan my_plan;
158 moveit::core::RobotStatePtr current_state;
159 std::vector<double> joint_group_positions;
160
161 ros::Rate rate(60.0);
162
163 /* main loop */
164
165 while (node_handle.ok()){
166     try{
167
168         transformStamped_left_hand_1 = tfBuffer.lookupTransform("torso_1", "left_hand_1",
169                                         ros::Time(0));
170
171         /* optional Transform calculation to implement grab functionality of
172          the gripper
173
174             transformStamped_right_hand_1 = tfBuffer.lookupTransform("torso_1", "right_hand_1",
175                                         ros::Time(0));
176
177         */
178     }
179     catch (tf2::TransformException &ex) {
```

```

180         ROS_WARN( "%s" , ex.what() );
181     continue;
182 }
183
184 /* calculate new position target for endeffector */
185 distance[0] = conversion_factor * (transformStamped_left_hand_1.
186     transform.translation.x - 0.15);
187 distance[1] = conversion_factor * (transformStamped_left_hand_1.
188     transform.translation.z*(-1.0));
189 distance[2] = conversion_factor * (transformStamped_left_hand_1.
190     transform.translation.y - 0.15);
191
192 distance = distance.rotate(rotate, 0.800);
193 move_group_arm.setPositionTarget(distance[0], distance[1], distance[2]);
194 ;
195
196 ROS_INFO_NAMED("planning","x : %lf y: %lf z: %lf", distance[0],
197                 distance[1], distance[2]);
198 ROS_INFO_NAMED("distance","distance: %lf", distance.length());
199 bool succes = (move_group_arm.plan(my_plan) == moveit::
200     planning_interface::MoveItErrorCode::SUCCESS);
201
202 if(succes){
203     succes = false;
204     joint_group_positions = my_plan.trajectory_.joint_trajectory.points.
205         back().positions;
206     joint_group_positions[4] = -0.0;
207     move_group_arm.setJointValueTarget(joint_group_positions);
208     succes = (move_group_arm.plan(my_plan) == moveit::planning_interface
209         ::MoveItErrorCode::SUCCESS);
210
211     if(succes){
212         move_group_arm.execute(my_plan);
213     }
214 }
215
216 /* optional gripper control */
217
218 if(true){
219     // joint_group_positions[0] = -0.300;
220     // move_group_gripper.setJointValueTarget(joint_group_positions);
221     // move_group_gripper.move();
222 }
223 rate.sleep();

```

## A. Quellcode

---

```
224     return 0;  
225  
226 }
```

Listing A.1: move\_group\_interface.cpp

```
1  /**  
2  * *****  
3  * @file      gesture_control.cpp  
4  * @author    Oliver Bosin  
5  * @version   V1.0.0  
6  * @date      22.02.2019  
7  * @copyright 2011 – 2019 UniBw M – ETTI – Institute 4  
8  * @brief     Node to control the turtle in turtlesim  
9  * @details   This Node listen to the tf of skeleton broadcasted by  
10 *             openni_tracker  
11 *             and then publish messages to change velocity and angle of the  
12 *             turtle  
13 *             Therefore following publishers are created.  
14 *  
15 *             1. PUBLISHERS  
16 *                 -# turtle_vel (publish on "/UDPcommand" topic)  
17 *  
18 * *****  
19 * @par History:  
20 *  
21 *     @details V1.0.0 22.02.2019 Oliver Bosin  
22 *             – Initial Release  
23 * *****  
24 * @todo  
25 * *****  
26 * @bug    none  
27 * *****  
28 */  
29 #include <ros/ros.h>  
30 #include <tf2_ros/transform_listener.h>  
31 #include <geometry_msgs/TransformStamped.h>  
32 #include <geometry_msgs/Twist.h>  
33 #include <turtlesim/Spawn.h>  
34  
35  
36 /**  
37 * @brief Main function for gesture_control  
38 * @details In this function tf-listener listen to tf broadcasted by  
39 *          openni_tracker and calculates the values to be send by  
40 *          the publisher turtle_vel to control the turtle in  
41 *          turtlesim  
42 * @param  [in] argc: Non-negative value representing the number of  
43 *                  arguments passed to the program from the  
44 *                  environment in which the program is run.  
45 * @param  [in] argv: Pointer to an array of pointers to null-terminated  
46 *                  multibyte strings that represent the arguments
```

---

```

47 *                         passed to the program from the execution
48 *                         environment
49 * @retval If the return statement is used, the return value is used as
50 *         the argument to the implicit call to exit().
51 *         This value can be:
52 *             @arg EXIT_SUCCESS [indicate successful termination]
53 *             @arg EXIT_FAILURE [indicate unsuccessful termination]
54 */
55 int main(int argc, char** argv){
56     ros::init(argc, argv, "gesture_control");
57
58     ros::NodeHandle node;
59
60     ros::service::waitForService("spawn");
61     ros::ServiceClient spawner =
62         node.serviceClient<turtleSim::Spawn>("spawn");
63     turtleSim::Spawn turtle;
64     turtle.request.x = 4;
65     turtle.request.y = 2;
66     turtle.request.theta = 0;
67     turtle.request.name = "turtle2";
68     spawner.call(turtle);
69
70     ros::Publisher turtle_vel =
71         node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);
72
73     /* Here a TransformListener object is created.
74      * Once the listener is created it starts receiving tf2 transformations
75      * over the wire
76      * and buffers them for up to 10 seconds*/
77     tf2_ros::Buffer tfBuffer;
78     tf2_ros::TransformListener tfListener(tfBuffer);
79
80     ros::Rate rate(10.0);
81     while (node.ok()){
82         geometry_msgs::TransformStamped transformStamped;
83         try{
84             /*here the transform between openni_link and left_hand_1 is calculated
85              *openni_link -> kinect , left_hand_1 -> right hand of person*/
86             transformStamped = tfBuffer.lookupTransform("openni_link", "left_hand_1",
87                                         ros::Time(0));
88         }
89         catch (tf2::TransformException &ex) {
90             ROS_WARN("%s", ex.what());
91
92             ros::Duration(1.0).sleep();
93             continue;
94         }
95
96         geometry_msgs::Twist vel_msg;
97         /*if right hand is right of the kinect from the perspective of the person

```

## A. Quellcode

---

```
97     the
98     *turtle will move forward*/
99     if(transformStamped.transform.translation.y > 0.2)
100    {
101        vel_msg.linear.x = 1.0;
102    }
103    /*if right hand is left of the kinect from the perspective of the
104     person the
105     *turtle will move backward*/
106    if(transformStamped.transform.translation.y < -0.2){
107        vel_msg.linear.x = - 1.0;
108    }
109    /*if right hand is in front of the kinect from the perspective of the
110     person the
111     *turtle will stop*/
112    vel_msg.linear.x = 0.0;
113    }
114    /*if right hand is above the kinect from the perspective of the person
115     the
116     *turtle will turn left*/
117    if(transformStamped.transform.translation.z > 0.3){
118        vel_msg.angular.z = 1.0;
119    }
120    /*if right hand is below the kinect from the perspective of the person
121     the
122     *turtle will turn right*/
123    if(transformStamped.transform.translation.z < -0.3){
124        vel_msg.angular.z = - 1.0;
125    }
126    /*if right hand is in front of the kinect from the perspective of the
127     person the
128     *turtle will not turn*/
129    vel_msg.angular.z = 0.0;
130    }
131    turtle_vel.publish(vel_msg);
132    rate.sleep();
133    }
134    return 0;
135    };
136}
```

Listing A.2: gesture\_control.cpp

## B. Arbeitsaufwand

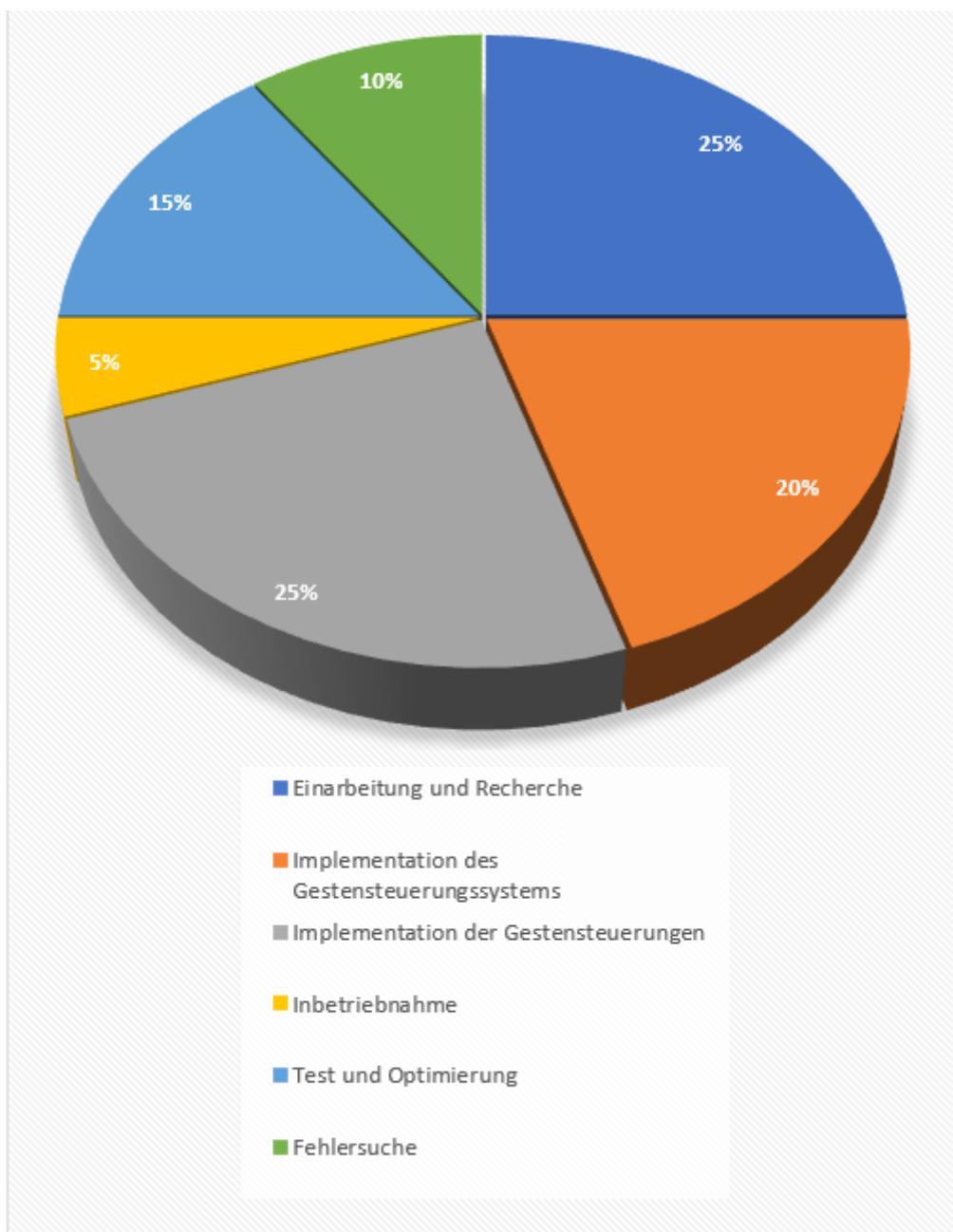


Abbildung B.1.: Arbeitsaufwand



# Literaturverzeichnis

- [1] Christian Blank. „Generierung von Tiefenbildern mittels Stereoskopie“. Bachelorarbeit. Hochschule für Angewandte Wissenschaften Hamburg, 2013.
- [2] *Eclipse*. 15. Mai 2019. URL: <https://www.eclipse.org/>.
- [3] Carol Fairchild und Dr. Thomas Harman. *ROS Robotics By Example*. Packt Publishing, 2016.
- [4] Xiaoyi Jiang und Horst Bunke. *Dreidimensionales Computersehen*. Springer, 1997.
- [5] *KEIL*. 15. Mai 2019. URL: <http://www.keil.com/>.
- [6] *Kinect*. 20. Mai 2019. URL: <https://developer.microsoft.com/de-de/windows/kinect>.
- [7] Anis Koubaa. *Robot Operating System(ROS)*. Springer International Publishing Switzerland, 2016.
- [8] Daniel Modrow. „Echtzeitfähige aktive Stereoskopie für technische und biometrische Anwendungen“. Dissertation. Technische Universität München, 2008.
- [9] *MoveIt!* 17. Mai 2019. URL: <https://moveit.ros.org/>.
- [10] *OpenNI*. 15. Mai 2019. URL: [https://github.com/OpenNI/OpenNI/blob/master/Documentation/OpenNI\\_UserGuide.pdf](https://github.com/OpenNI/OpenNI/blob/master/Documentation/OpenNI_UserGuide.pdf).
- [11] *OpenNI Tracker*. 16. Mai 2019. URL: [http://wiki.ros.org/openni\\_tracker](http://wiki.ros.org/openni_tracker).
- [12] *PrimeSense NITE Algorithms 1.5*. PrimeSense. 16. Mai 2019. URL: [http://cvrlcode.ics.forth.gr/web\\_share/OpenNI/NITE\\_SDK/NITE\\_1.x/NITE-Algorithms.pdf](http://cvrlcode.ics.forth.gr/web_share/OpenNI/NITE_SDK/NITE_1.x/NITE-Algorithms.pdf).
- [13] *Robot Operating System*. 15. Mai 2019. URL: <https://www.ros.org/about-ros/>.
- [14] *Rviz*. 19. Mai 2019. URL: <http://wiki.ros.org/rviz>.
- [15] Christoph Schmiedecke. „Tiefenbilderzeugung mit Hilfe von skalierungsinvarianten Merkmalen für ein Stereokameramodul“. Bachelorarbeit. Hochschule für Angewandte Wissenschaften Hamburg, 2009.
- [16] *SensorKinect*. 15. Mai 2019. URL: <https://github.com/avin2/SensorKinect>.
- [17] *Turtle Simulator*. 17. Mai 2019. URL: <http://wiki.ros.org/turtlesim>.
- [18] Christian Waldner. „Entwicklung der Steuerung eines Roboterarms mit ROS“. Bachelorarbeit. WE4 ETTI Universität der Bundeswehr München, 2018.

## *Literaturverzeichnis*

---

- [19] Christian Waldner. „Konzept für die Steuerung eines Roboterarms mit ROS“. Projektarbeit.  
WE4 ETTI Universität der Bundeswehr München, 2018.

# Index

- \* , 4
- Überblick, 5
- Abschlussarbeit, 5
- Abstrakt, 5
- Anhang, 45
- Aufwand, 49
- Ausblick, 43
- Betriebssysteme, 3
- Diskussion, 43
- Einleitung, 1
- Embedded Stereo, 11
- GANTT-Diagramm, 49
- Grundlagen, 11
- Hardware, 8
- Infrarotmuster, 12
- Integrierte Entwicklungsumgebungen, 3
- Kinect, 8
- Mikrocontrollerboard, 9
- Module, 13
- MoveIt, 5, 17
- NITE, 5
- OpenNI, 4, 13
- OpenNI Tracker, 5
- Platine, 46
- Platinen-Nummer, 46
- Projektplan, 49
- Rechner, 8
- Robot Operating System, 4
- Roboterarm "rob\_arm\_small", 8
- ROS, 14
- Ros-Kinetic-OpenNI, 5
- Schaltpläne, 46
- Schaltplan, 45
- SensorKinect, 5
- Software, 3
- Softwarekonzept, 6
- Stereokamera, 11
- Systemaufbau, 3, 10
- Time-Of-Flight, 12
- Turtle Simulator, 5
- Version, 46
- Wichtige ROS-Befehle, 17
- Zeitaufwand, 49