



Entwicklung einer KI-basierten Robotersteuerung mit Industrial ROS

Automatische Erkennung und Lokalisierung von
Objekten durch ein 3D-Kamerasystem

Bachelorarbeit von
David Gries

27. Februar 2022

Entwicklung einer KI-basierten Robotersteuerung mit Industrial ROS
Automatische Erkennung und Lokalisierung von Objekten durch ein
3D-Kamerasystem

Autor:
David Gries
2217395

Prüfer:
Prof. Dr. Hartmut Bruhm



TECHNISCHE HOCHSCHULE ASCHAFFENBURG
FAKULTÄT INGENIEURSWISSENSCHAFTEN
WÜRZBURGER STRASSE 45
D-63743 ASCHAFFENBURG

Erklärung zur Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen Hilfsmittel als die angegebenen verwendet habe. Die Stellen, die anderen Werken (gilt ebenso für Werke aus elektronischen Datenbanken oder aus dem Internet) wörtlich oder sinngemäß entnommen sind, habe ich unter Angabe der Quelle und Einhaltung der Regeln wissenschaftlichen Zitierens kenntlich gemacht. Diese Versicherung umfasst auch in der Arbeit verwendete bildliche Darstellungen, Tabellen, Kartenskizzen und gelieferte Zeichnungen.

Mir ist bewusst, dass Täuschungen nach der für mich gültigen Studien- und Prüfungsordnung / nach § 6 RaPO / § 48 BayVwVfG geahndet werden.

Die Zustimmung zur elektronischen Plagiatsprüfung wird erteilt.

Ort, Datum

Unterschrift des Verfassers / der Verfasserin

Veröffentlichung der Arbeit in der Bibliothek der Technischen Hochschule Aschaffenburg

Der Veröffentlichung der Master-/Bachelorarbeit in der Bibliothek der Technischen Hochschule Aschaffenburg wird zugestimmt.

Ort, Datum

Unterschrift des Verfassers / der Verfasserin

Entwicklung einer KI-basierten Robotersteuerung mit Industrial ROS

Automatische Erkennung und Lokalisierung von Objekten durch
ein 3D-Kamerasystem

David Gries

Abstract

Die vorliegende Arbeit befasst sich mit der Entwicklung einer KI-basierten Robotersteuerung auf Basis von ROS, die mithilfe eines 3D-Kamerasystems die Position und Art von Objekten erkennt. Diese nutzt das Intel RealSense 435 3D-Kamerasystem und den Objekterkennungsalgorithmus YOLOv3. Die Umsetzung des Systems erfolgt am 6-Achs-Knickarmroboter RV6L der Firma Reis Robotics. Es wird ein ROS-Paket implementiert, das die zweidimensionale Objektposition mit dem 3D-Bild der RealSense abgleicht, um zusätzlich zum Objektmittelpunkt dessen Abstand zur Kamera zu ermitteln. Der YOLOv3-Algorithmus weist eine geringe Fehlerrate bei der Klassifizierung auf und erkennt die Positionen der Objekte zuverlässig, kann jedoch nicht deren Orientierung feststellen. Im Vergleich zu traditioneller Computer Vision bietet der verwendete Deep Learning Algorithmus vor allem in Bezug auf den Einlernprozess und der möglichen Weiterentwicklung Vorteile.

This thesis describes the development of an AI-based robot controller using ROS. It uses a 3D camera system to recognize the position and class of unordered objects. To accomplish this, an Intel RealSense 435 3D camera system and the object recognition algorithm YOLOv3 are used. The system is implemented on the six-axis articulated arm robot RV6L from Reis Robotics. A ROS package that matches the two-dimensional object position with the three-dimensional image of the RealSense is implemented to determine the distance between the camera and the object center. The YOLOv3 algorithm has a low error rate in classification and reliably detects the positions of the objects, but cannot determine their orientation. Compared to traditional computer vision, the Deep Learning algorithm offers advantages especially in terms of the learning process and possible further development.

Inhaltsverzeichnis

Abkürzungsverzeichnis	III
1 Einleitung	1
2 Technische Grundlagen	3
2.1 Robot Operating System (ROS)	3
2.1.1 Aufbau	4
2.1.2 ROS Versionen	8
2.2 Arten von 3D-Kameras	9
2.2.1 Stereokamera	9
2.2.2 RGB-D Kamera	10
2.3 Objekterkennungsalgorithmen	11
2.3.1 Traditionelle Computer Vision	11
2.3.2 Deep Learning	12
3 Software und Hardware	15
3.1 Konfiguration des Kamerasystems	15
3.1.1 Technische Daten	15
3.1.2 Kalibrierung	19
3.1.3 Befestigung	21
3.2 Implementierung der Objekterkennung	21
3.2.1 Voraussetzungen für das Objekterkennungsprogramm	22
3.2.2 Training des YOLO-Algorithmus	23
3.2.3 rv6l_3d-Paket	25
3.2.4 Bounding Box Subscriber	27
3.2.5 Algorithmus zur Positionserkennung	27
4 Gesamtsystem	32
4.1 Programmstruktur	32
4.2 Programmablauf	35
4.3 Bewegungsablauf	35
5 Evaluierung und Weiterentwicklung	38
5.1 Auswertung und Optimierung	38
5.2 Ansätze zur Weiterentwicklung des rv6l_3d-Pakets	39
6 Fazit	42

Abbildungsverzeichnis **IV**

Literatur **V**

Anhang **XII**

Abkürzungsverzeichnis

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
CV	Computer Vision
DCNN	Deep Convolutional Neural Network
DL	Deep Learning
HBBs	Horizontal Bounding Boxes
LTS	Long Term Support
OBBs	Orientation Bounding Boxes
ROS	Robot Operating System
RNN	Recurrent Neural Network
SDK	Software Development Kit
SIFT	Scale-invariant Feature Transform
SURF	Speeded up Robust Features
SVM	Support Vector Machine
ToF	Time of Flight
YOLO	You Only Look Once

1 Einleitung

Roboter spielen aufgrund der schnellen Anpassungsmöglichkeit bei Produktionsprozessen, die heute einen hohen Stellenwert hat, eine immer wichtigere Rolle in der Industrie. Mit steigender Rechenleistung und dem schnellen Fortschritt im Bereich der Bilderkennung, vor allem unter Anwendung von *Convolutional Neural Networks (CNNs)*, findet die Erkennung von Objekten mit Kamerasystemen in immer mehr Bereichen der Robotik Anwendung. So können beispielsweise schon heute auf *Deep Learning (DL)* basierende Techniken, die noch vor Kurzem nur selten für einen praktischen Einsatz geeignet waren, Anwendung in der Industrie finden. Zukünftig kann *DL* für die Vereinfachung der Bedienbarkeit von Robotern und eine weiter reichende Automatisierung von Prozessen genutzt werden [1]. Dabei erfolgt die Programmierung von Robotersteuerungen meist mithilfe herstellerspezifischer Programme, die an bestimmte Hardware gebunden sind. Diese Herstellerbindung bringt Nachteile bei der Interaktionsfähigkeit mehrerer Roboter mit sich. Auch die Integration von Komponenten wie Kameras und weiteren Sensoren, die nicht vom jeweiligen Hersteller der Robotersteuerung unterstützt werden, erschwert die Automatisierung von Prozessen und erhöht die Umsetzungskosten. Diese Problematik versucht das Open-Source-Projekt Robot Operating System (ROS) zu lösen. Das seit 2007 kontinuierlich weiterentwickelte System verfolgt das Ziel, die Steuerung verschiedener Roboter zu standardisieren und eine gemeinsame Schnittstelle zu schaffen [2, Kapitel 1]. Der öffentlich zugängliche Quellcode bietet die Möglichkeit, eigene Anpassungen vorzunehmen und zur Entwicklung beizutragen. Neben der Interoperabilität der Programmteile stellen aufgrund des Open-Source-Codes auch die umfangreiche Verfügbarkeit anwendungsspezifischer Pakete und Anwendungen, die mit der vom Hersteller bereitgestellten Schnittstelle nicht realisierbar sind, große Vorteile des ROS dar. Durch die Möglichkeit, eigene mit ROS kompatible Programme zu veröffentlichen, erfolgt eine schnelle und kontinuierliche Weiterentwicklung des Systems.

Da ROS Industrial nicht nur in der klassischen Robotik eingesetzt wird, sondern auch in anderen Bereichen wie beispielsweise der Forschung an autonomen Fahrzeugen, ergibt sich ein breites Spektrum an Softwarepaketen mit unterschiedlichen Anwendungszwecken. Dazu zählt unter anderem das Programm *darknet_ros*, das eine Schnittstelle zwischen *You Only Look Once (YOLO)*, einem Algorithmus zur Objekterkennung, und ROS Industrial zur Verfügung stellt. Die Erkennung, Unterscheidung und Ortung von Objekten ist, wenn überhaupt eine

1 Einleitung

Schnittstelle vorhanden ist, bei Nutzung von Systemen der Hersteller oft hardwaregebunden und nur schwierig umsetzbar.

Das Ziel dieser Arbeit ist die Programmierung einer Robotersteuerung unter Anwendung des ROS Industrial und eines 3D-Kamerasystems. Diese soll aus einer ungeordneten Zusammenstellung von Bauteilen festgelegte Teile erkennen und gemäß einem variablen Bauplan anordnen. Die Bauteile sollen hierbei, wenn bestimmte Bedingungen wie eine passende Größe und Oberflächenbeschaffenheit eingehalten werden, nach einem Einlernprozess frei gewählt werden können. Das Programm soll außerdem mit wenigen Anpassungen unabhängig von der Kameraposition am Endeffektor und der Roboterkinematik funktionieren. Es kann als Grundlage für viele Anwendungen, die zum Beispiel eine einfach anpassbare Sortierung von Objekten oder die Unterscheidung und Lokalisierung ungeordneter Teile erfordern, genutzt werden.

Wie kann ein solches System zur Erkennung von Art und Position ungeordneter Bauteile unter Verwendung des Robot Operating System umgesetzt werden?

Zur Beantwortung dieser Frage befasst sich die vorliegende Arbeit primär mit der Objekterkennung. Die Implementierung eines Bauplans und der Ansteuerung wird in einer separaten Arbeit behandelt [3, Dennis Steinbeck: Entwicklung einer KI-basierten Robotersteuerung mit Industrial ROS - Generierung von Handhabungssequenzen für mechanische Konstruktionen nach gegebenen Bauplänen]. Für die Umsetzung wird aufgrund der Verfügbarkeit relevanter Pakete und der Stabilität und Zuverlässigkeit die Version *ROS-I Noetic* genutzt. Die Objekterkennung erfolgt mit einem Intel RealSense 435 3D-Kamerasystem, das am Endeffektor des 6-achsigen Knickarmroboters RV6L von Reis Robotics befestigt ist. Der Hauptteil der Arbeit ist in drei Teile gegliedert. Der erste Teil befasst sich mit den technischen Grundlagen, die zum Verständnis des Gesamtsystems notwendig sind oder das gewählte Vorgehen begründen. Anschließend wird auf die verwendete Software und Hardware zur Objekterkennung und Steuerung des Roboters eingegangen. Der dritte Teil bietet eine Übersicht über die Zusammenhänge des Gesamtsystems und den Ablauf von Erkennung bis zur Handhabung der Objekte.

2 Technische Grundlagen

Da die Erkennung, Lokalisierung und Handhabung der Objekte eine Interaktionsfähigkeit zwischen dem verwendeten Kamerasystem, einem Objekterkennungsalgorithmus und der Robotersteuerung voraussetzen, werden in diesem Kapitel notwendige Grundlagen und für die Umsetzung angewandte Techniken erläutert.

2.1 Robot Operating System (ROS)

Die Kommunikation zwischen in der Robotik eingesetzter Hardware bereitete in der Vergangenheit aufgrund nicht vorhandener Standards bei der Datenübermittlung oft Probleme, die nur mit hohem Entwicklungsaufwand gelöst werden konnten [2, Kapitel 1]. Daher ist ein einheitliches System, das verschiedene Programme und Hardware verbindet, notwendig. Diese Problematik kann durch die Verwendung des ROS gelöst werden. Dieses bildet mit einer Vielzahl verfügbarer Pakete eine Grundlage für komplexe Steuerungen. Es kann auf einem einzelnen Computer oder einem Netzwerk aus Computern genutzt werden und vereinfacht die Interoperabilität verschiedener Steuerungshardware. ROS dient außerdem als Middleware, die die Kommunikation zwischen Teilen von Software ermöglicht. Die Pakete sind dabei entweder direkt auf ROS ausgelegt, oder bilden eine Schnittstelle zu ROS für Hardwarekomponenten oder inkompatible Software. Die Website ros.org beschreibt das System als ein Paket aus Software und Softwarebibliotheken, mit dem Roboterprogramme entwickelt werden können [4].

ROS wurde 2007 unter dem Namen *Switchyard* von Morgan Quigley gestartet [5, Kapitel 1]. Das öffentlich verfügbare Repository wurde im Januar 2012 publiziert. Es existieren heute mehrere Konsortien für verschiedene Regionen, darunter beispielsweise das vom Fraunhofer-Institut für Produktionstechnik und Automatisierung geleitete *ROS-I Consortium Europe*. Diese stellen technische Unterstützung und Training für ROS bereit und legen Entwicklungsziele fest [6]. Ein großer Vorteil des ROS verglichen mit anderen Systemen neben der Hardwareunabhängigkeit ist, dass es ein Open-Source-Projekt ist. Somit kann es gut auf alle Anforderungen angepasst werden und es kann eine Vielzahl an Pro-

2 Technische Grundlagen

grammen, die mit ROS kompatibel sind, genutzt werden. Dabei ist laut ros.org das primäre Ziel des Projekts nicht die Bereitstellung besonders vieler Features, sondern die Unterstützung der Wiederverwendung von Quellcode in der Forschung und Entwicklung [7, Absatz 2].

Nachfolgend werden der Aufbau und die Versionen des ROS sowie Möglichkeiten zur Integration einer Objekterkennung beschrieben. Der Fokus liegt auf der im Projekt verwendeten Version ROS-I.

2.1.1 Aufbau

Anders als der Name ROS impliziert, handelt es sich technisch gesehen nicht um ein klassisches Betriebssystem, sondern Softwarepakete, die auf bestehenden Betriebssystemen installiert werden können. Das Programm unterstützt offiziell die Linux-Distributionen *Ubuntu* und *Debian*, während für *Windows* und *Arch Linux* experimentelle Versionen zur Verfügung stehen [8]. Die empfohlene Installationsmethode besteht im Hinzufügen eines Repositorys, wodurch ROS über den Paketmanager der verwendeten Linux-Distribution installiert werden kann. Gründe für die Bezeichnung als Operating System sind die Bereitstellung von Services wie Hardware Abstraktion, Paketmanagement sowie eine Kommunikation zwischen Prozessen, wobei es sich um typische Eigenschaften von Betriebssystemen handelt [7, Absatz 1].

ROS ist grundsätzlich in mehrere Teile gegliedert. Die offizielle Dokumentation unterscheidet hier zwischen dem „Filesystem Level“, „Computation Graph Level“ und „Community Level“ [9]. Für eine bessere Übersicht wird hier zwischen **Paketmanagement, Struktur und Entwicklung und Dokumentation** unterschieden und verstärkt auf die für diese Arbeit relevanten Teile eingegangen.

2.1.1.1 Paketmanagement

Das Paketmanagement des ROS kann in folgende Punkte gegliedert werden:

Packages

Packages bilden die Grundlage für die Organisation von Softwarekomponenten. Die Softwarepakete enthalten alle notwendigen Teile eines einzelnen Programms. Eine spezielle Art von Packages sind Metapackages, bei denen es sich um eine Zusammenstellung mehrerer Einzelpakete handelt [9, Absatz 1].

2 Technische Grundlagen

Repositories

Repositories sind Paketquellen, die ein oder mehrere Softwarepakete zum Download bereitstellen. Neben dem primären ROS-Repository, das essenzielle Pakete enthält, können Institutionen oder einzelne Entwickler eigene *Repositories* zur Verfügung stellen [9, Absatz 1/3].

Dependencies

Dependencies sind *Packages*, die von anderen Softwarepaketen zur korrekten Funktionsweise benötigt werden [9, Absatz 1].

Workspaces

Workspaces sind Ordner im Nutzerverzeichnis des Systems, die bei der Organisation lokal installierter Pakete helfen. Diese fungieren als Overlay über den vom Paketmanager systemweit installierten Softwarepaketen [10, Absatz 4]. Die Priorität liegt hier, wenn Pakete gleichen Namens lokal und systemweit vorhanden sind, bei den lokal installierten Paketen.

Build System

Für die Installation eigens erstellter Pakete, Software außerhalb der vom Paketmanager nutzbaren *Repositories*, oder lokale Änderungen von Paketen wird für ROS-I standardmäßig das Tool *catkin* verwendet, wobei es sich um das offizielle *Build System* von ROS-I und einen Nachfolger von *rosbuild* handelt. Im Vergleich zu Tools wie *Autotools* oder dem von *catkin* unter anderem genutzten *CMake*, zielt *catkin* auf Vereinfachung des Build-Prozesses, Standardisierung bei Verwendung verschiedener Programmiersprachen und das Auflösen von Abhängigkeiten, wodurch der Build-Prozess bei komplexen Zusammenstellungen von Paketen auch von Nutzern mit wenig Programmiererfahrung durchführbar ist [11, Absatz 1.2]. Pakete für *catkin* werden in *Workspaces* organisiert.

Das Paketmanagement ähnelt somit dem vieler Linux Distributionen. Pakete werden vorzugsweise direkt über den *Package Manager* des verwendeten Betriebssystems installiert, was auch zur automatischen Installation notwendiger *Dependencies* führt. Bei manuell kompliierten Paketen stellt ROS zudem das Programm *rosdep* bereit, mit dem *Dependencies* eines bestimmten Pakets oder mehrere Pakete unter einem gemeinsamen Pfad automatisch installiert werden können [12, Absatz 2].

2.1.1.2 Struktur

ROS läuft nicht als ein Prozess, sondern in Form einzelner, verteilter Programme. Neben einer erhöhten Ausfallsicherheit des Gesamtsystems hat diese

2 Technische Grundlagen

Struktur weitere Vorteile, wie eine unkomplizierte Verteilung von Prozessen über mehrere Systeme in einem Netzwerk oder die vereinfachte Kommunikation zwischen unabhängig voneinander entwickelten ROS-Packages. Auch die Möglichkeit, Programme in verschiedenen Programmiersprachen problemlos zu kombinieren ist ein Vorteil dieses modularen Aufbaus. So können viele einzelne, öffentlich verfügbare oder selbst programmierte Pakete zu komplexen Systemen kombiniert werden.

Node

Nodes sind logisch zusammenhängende Teile eines Programms. So können Softwarepakete in mehrere Prozesse aufgeteilt werden, die untereinander über den *Master Node* kommunizieren [9, Absatz 2]. Die Kommunikation erfolgt dabei mithilfe von *Messages*.

Message

Als Message wird eine Datenstruktur bezeichnet, die Felder verschiedener, einfacher Datentypen beinhaltet. Dazu gehören beispielsweise *integer*, *boolean* oder *float* [9, Absatz 2]. Die Kommunikation mittels Messages kann auf folgende Arten erfolgen:

Topic: Topics agieren als *Publisher/Subscriber*-System. Nachrichten werden von einem *Node* an ein *Topic* gesendet. Die Identifikation erfolgt hierbei über den Namen des *Topics*. Nodes haben die Möglichkeit, ein *Topic* zu registrieren, dessen Nachrichten benötigt werden. Die Anzahl an *Publishern* und *Subscribers*, die den gleichen *Topic* verwenden, ist hierbei nicht limitiert. Auch die Anzahl an *Topics*, die von einem *Node* verwendet werden können, ist unbegrenzt [9, Absatz 2].

Ein Vorteil dieser Methode ist die Unabhängigkeit zwischen *Publisher* und *Subscriber*. Diese kann aber auch dazu führen, dass ein fehlerhafter *Node* nicht erkannt wird.

Service: Im Vergleich zur einseitigen Kommunikation der *Topics* handelt es sich hier um ein *Request/Reply*-System, wodurch Antworten auf empfangene Nachrichten möglich sind. Ein Service besteht aus jeweils einer Message für Anfrage und Antwort [9, Absatz 2].

Services eignen sich für die aktive Interaktion zwischen zwei *Nodes*.

Master

Als Master wird der Prozess für die Namensregistrierung und den *Parameter Server* bezeichnet. Dieser ist für die Kommunikation zwischen mehreren *Nodes* zwingend notwendig. Mit dem *Parameter Server* stellt er außerdem globale Konfigurationsdateien bereit [9, Absatz 2].

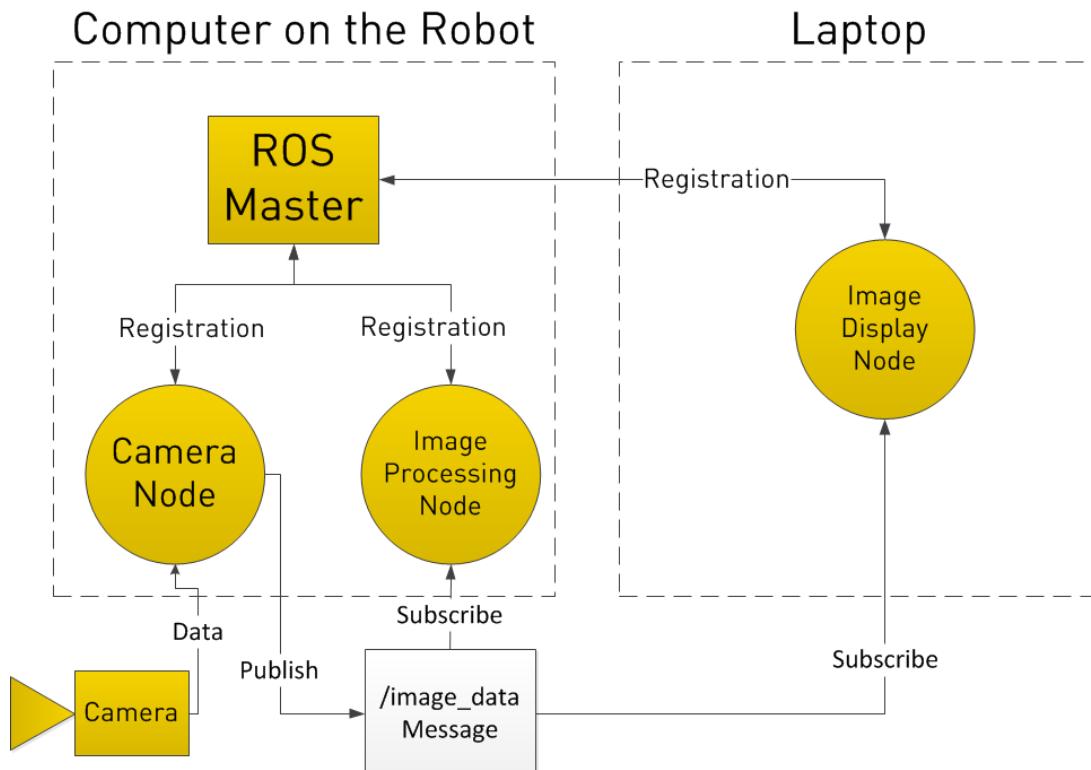


Abbildung 2.1: ROS Struktur [13]

Abbildung 2.1 zeigt eine mögliche ROS Netzwerkstruktur mit Integration einer Kamera und mehreren Computern.

2.1.1.3 Entwicklung und Dokumentation

Da ROS von der Community entwickelt wird, sind Ressourcen zum Austausch von Code und Wissen notwendig. Folgendes ist für die Entwicklung besonders relevant:

Distributionen

ROS ist ähnlich wie Linux-basierte Betriebssysteme in *Distributionen* aufgeteilt. Diese Gliederung sorgt dafür, dass Pakete pro *Distribution* eine bestimmte Softwareversion nutzen [9, Absatz 3]. Dadurch kann eine hohe Kompatibilität zwischen mehreren Paketen gewährleistet werden. Details zu verfügbaren ROS-Versionen werden in **Abschnitt 2.1.2** erläutert.

ROS Wiki

Das *ROS Community Wiki* ist die Hauptinformationsquelle für Dokumentationen, Tutorials und Korrekturvorschläge [9, Absatz 3]. Einträge können

2 Technische Grundlagen

von Jedem nach Erstellen eines Accounts hinzugefügt oder bearbeitet werden. Das *Community Wiki* sollte von Programmierern und Paketerstellern zur Dokumentation genutzt werden.

Support

Für Neuigkeiten über ROS kann das ROS-Forum genutzt werden [14, Absatz 2]. Bei Fehlern, Vorschlägen oder Feature Requests stehen entsprechende Ressourcen zur Eigenrecherche oder Kommunikation zur Verfügung. Aktuelle Links und Leitlinien sind unter wiki.ros.org/Support verfügbar [14, Absatz 3-6].

2.1.2 ROS Versionen

Wie im letzten Abschnitt erwähnt, gibt es mehrere *ROS-Distributionen*. Dabei handelt es sich um Zusammenstellungen von *Packages* miteinander kompatibler Versionen, weshalb sich die Weiterentwicklung der Hauptpakete einer *Distribution* großteils auf Fehlerbehebungen und das Schließen von Sicherheitslücken beschränkt [15, Absatz 2]. Dieses Konzept vereinfacht durch die konstante Code-Basis einer *Distribution* die Entwicklung und Wartung von Softwarepaketen.

Das *Community Wiki* führt Linux als Beispiel für das Konzept der *Distributionen* auf [15, Absatz 2]. Im ROS-Kontext sind diese jedoch, besonders bei Betrachtung neuer Versionen, eher mit *Releases* einer einzelnen Linux-Distribution vergleichbar. Frühere ROS-Versionen wurden je nach verfügbaren Ressourcen und Nachfrage unterschiedlich lange unterstützt, während aktuelle Releases einen *Long Term Support (LTS)* mit Unterstützungsduer von fünf Jahren anstreben [15, Absatz 5]. Beim Vergleich zwischen den Daten der Veröffentlichung und Unterstützungsduer von ROS mit den Daten von *Ubuntu LTS* [16] fallen Parallelen auf. Das Timing der Veröffentlichung neuer ROS-Versionen orientiert sich an dem der *Ubuntu LTS*-Versionen.

Da die Aktualisierung der alten Code-Basis von ROS-I nicht ohne große Änderungen der Funktionsweise realisierbar ist, wurde 2018 die erste stabile Version von ROS-II veröffentlicht. ROS-II folgt einem ähnlichen Release-Modell wie die erste Version, wobei die Unterstützungsduer der jeweiligen Versionen mit ein bis drei Jahren deutlich kürzer ausfällt. Neben den einzelnen Releases gibt es die „*Rolling Distribution*“, die als Version für Entwickler dient und kontinuierlich - auch mit Änderungen, die zu Fehlern im Code führen können - aktualisiert wird [17].

2.2 Arten von 3D-Kameras

¹Die heutige Arbeit wird in vielen Bereichen durch mobile Roboter sowie statio-näre Roboterarme ersetzt. Hierbei spielen vor allem die Genauigkeit sowie die Effizienz eine große Rolle. Die Produktion von Bauteilen, Sortierung, Kartierung und vielen weiteren Arbeitsschritten wurde hauptsächlich durch Kamerasyste-me ermöglicht. Diese sind die Augen des Roboters, mit denen der Prozess auto-matisiert werden kann. Solche Systeme ermöglichen ein autonomes Arbeiten des Roboters und erfordern dadurch zum Beispiel bei einer Qualitätsprüfung, einer Bauteilidentifizierung oder beim Scannen von Produkt-Codes keine Fach-kräfte. Aufgaben, die eine sich ändernde Arbeitsumgebung haben, erfordern eine präzise Umgebungswahrnehmung. Mithilfe einer Kamera kann das Robo-tersystem so auf veränderte Situationen reagieren. Dadurch steigt zum einen die Flexibilität in Bezug auf die Einsatzbereiche, zum anderen kann durch die Umgebungswahrnehmung so die Wahrscheinlichkeit einer Kollision stark re-duziert werden. Beim Einsatz von Kamerasystemen kann man zwischen zwei be-kannten Verfahren unterscheiden. Diese werden im folgenden in Bezug auf die Funktion und Einsetzbarkeit verglichen.

2.2.1 Stereokamera

¹Die Stereoskopie beschreibt ein Verfahren, welches mithilfe von zweidimen-sionalen Bildern einen räumlichen Eindruck der Tiefe vermittelt. Eine Stereo-kamera ist mit zwei nebeneinander angeordneten Kameralinsen ausgestattet. Diese haben einen konstanten Versatz zueinander. Die Verschlüsse der Linsen sind miteinander gekoppelt und müssen nicht individuell eingestellt werden.

Die Funktionsweise der Stereoskopie basiert auf dem binokularen Sehen. Der Ursprung liegt nach Sir David Brewster beim Menschen, indem er das vor ihm liegende Bild durch zwei Augen erkennt und durch die Kombination beider Per-spektiven die Tiefen der 2D-Bilder wahrnimmt [18]. So nimmt die Kamera zwei Bilder aus den jeweiligen Perspektiven der Linsen auf und berechnet mithilfe von Prozessoren und dem mathematischen Verfahren der Triangulierung die Tiefe des zusammengeführten Bildes [19]. Dabei muss man jedoch bedenken, dass dieses Verfahren eine hohe Rechenleistung erfordert.

¹verfasst von Steinbeck [3]

2.2.2 RGB-D Kamera

²Der Begriff RGB-D beschreibt die Kombination aus einer Farb- und Distanzerkennung. Dabei werden zusätzlich zum einfachen 2D-Farbbild über entsprechende Infrarot-Sensoren die Tiefen gemessen, sodass eine 3D-Aufnahme berechnet werden kann. Dafür wird in der Regel das Time of Flight (ToF) Prinzip genutzt [20]. Dieses sogenannte Laufzeitverfahren misst die Distanzen für jeden Bildpunkt der Szene, indem ein Infrarot-Sensor eine *Point Cloud* auf die Oberfläche projiziert. Die Reflexionen werden von einem CMOS-Bildsensor erkannt. Mit Algorithmen wird die Laufzeit jedes Bildpunktes analysiert und zu Informationen zur Tiefe verarbeitet. Die RGB-Kamera charakterisiert das Bild zusätzlich, um weitere Merkmale festzustellen [20].

Dadurch, dass die 3D-Aufnahme zusätzlich mit Farbinformationen versehen ist, lassen sich gesamte Szenen und Objekte wesentlich genauer unterscheiden. Bei einer quasi identischen Form können zwei Objekte mit der Farbinformation trotzdem unterschieden werden. Außerdem sind die Anforderungen an Licht- und Umgebungsverhältnisse im Vergleich zur Stereokamera geringer, da die Tiefe mit Infrarot Licht gemessen und nicht berechnet wird. Dennoch sollte genügend Licht für eine saubere Erkennung vorhanden sein. Zudem darf das Sichtfeld der Kamera nicht eingeschränkt werden, damit die Objekte komplett sichtbar sind.

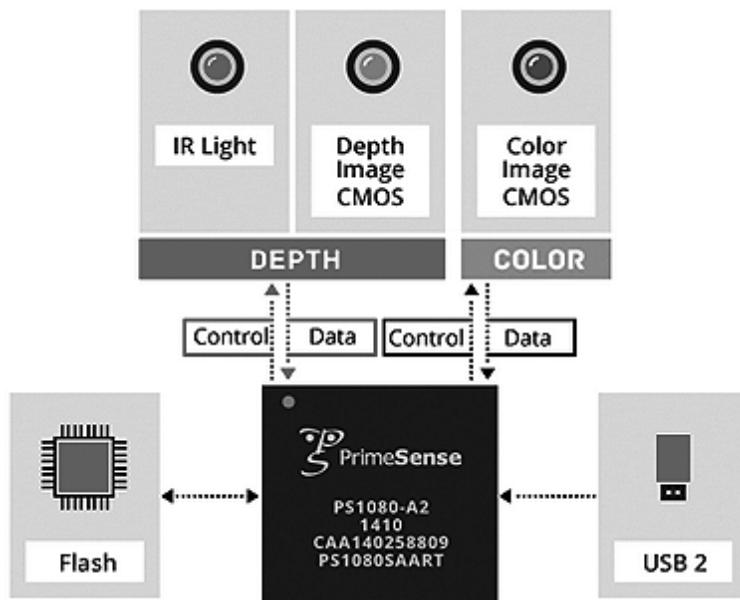


Abbildung 2.2: Aufbau RGB-D Kamera [21]

²verfasst von Steinbeck [3]

2.3 Objekterkennungsalgorithmen

Für die Erkennung und Identifikation von Objekten wird ein geeigneter Bilderkennungsalgorithmus benötigt. Hauptunterschiede bezüglich der Anwendung sind Genauigkeit, Geschwindigkeit und Hardwareanforderungen im Hinblick auf Leistung.

Der Algorithmus muss unter den Umgebungsbedingungen des Bauraums zuverlässig funktionieren und ausreichend schnell sein, um keine großen Verzögerungen durch die Erkennung zu verursachen. Die Schnittstelle zu ROS muss außerdem die Möglichkeit bieten, Positionsdaten der Objekte an das System weiterzugeben, um die Lokalisierung dieser zu ermöglichen. Die Weitergabe der Tiefenkoordinate durch das Objekterkennungsprogramm ist hierbei optional, da diese auch durch Abgleich der 2D-Koordinaten mit dem Tiefensensor berechnet werden kann. Da in der Arbeit mehrere Objekte verwendet werden, ist zudem die Unterstützung zur Erkennung verschiedener Bauteile im gleichen Bild notwendig.

Die Ansätze von Objekterkennungsalgorithmen können in traditionelle und moderne Ansätze gegliedert werden. Letztere basieren auf *DL* und sind grundsätzlich für ein breiteres Spektrum an Objekten geeignet [22, Absatz 2]. Folgend werden die Ansätze verglichen und Algorithmen aufgeführt, die gut in ROS integrierbar sind und mit RGB-Kamerasystemen funktionieren.

2.3.1 Traditionelle Computer Vision

Scale-invariant Feature Transform (SIFT) von Lowe [23] und *Speeded up Robust Features (SURF)* von Bay et al. [24] sind Algorithmen, die auf *Keypoints* in Bildern basieren, welche vom jeweiligen Algorithmus generiert und zur Erkennung von Objekten bei einer bestimmten Übereinstimmungsrate genutzt werden können. *Keypoints* sind um Punkte im Bild, die nahezu invariant gegenüber der Bildparameter wie Vergrößerung, Verzerrung und Rotation sind und somit gut als Orientierungspunkte genutzt werden können [23, Kapitel 13].

Genannte Techniken können für traditionelle Ansätze genutzt werden. Diese funktionieren grundsätzlich folgendermaßen: Zuerst wird die Position der Objekte im Bild durch Scannen des kompletten Bildes näherungsweise bestimmt. Anschließend können Techniken wie beispielsweise *SIFT* genutzt werden, um relevante Merkmale der Objekte zu extrahieren. Zuletzt werden Algorithmen wie *Support Vector Machine (SVM)* [25] zur Identifikation der Objektkategorie genutzt [26, Kapitel 1]. **Abbildung 2.3** zeigt den beschriebenen Ablauf.

2 Technische Grundlagen



Abbildung 2.3: Traditioneller Ansatz [26, Abbildung 3]

Traditionelle *Computer Vision (CV)* kann durch die geringere Komplexität effizienter als *DL* sein. Auch bei Abweichung der zu erkennenden Objekte vom Datensatz, der zum Training des Algorithmus genutzt wird, sind Algorithmen der traditionellen *CV* manchmal besser geeignet. Im Gegensatz zu *DL* ist traditionelle *CV* mit hohem Forschungsaufwand beim Einlernprozess verbunden. [27, Absatz 2.3].

Eine Möglichkeit zur Integration traditioneller Bilderkennung in ROS bietet beispielsweise das Paket *find_object_2d*, das neben Objektklassifikation und Positionserkennung auch die dreidimensionale Orientierung von Objekten im Raum erkennen kann [28].

Aufgrund des Forschungsaufwandes zum Einlernen der Objekte und der hohen Abhängigkeit von der Art und Form zu erkennender Teile eignet sich diese Art der Bilderkennung nur mäßig für den vorliegenden Fall.

2.3.2 Deep Learning

Verglichen mit traditioneller Bilderkennung werden beim *DL* keine mathematischen Modelle durch manuelles Sammeln und Auswerten von Prozessdaten (Deskriptive Analytik) oder Erstellen eines prädiktiven Modells (Prädiktive Analytik) erstellt [29, Kapitel 1]. Stattdessen werden *Neural Networks* verwendet. Diese sind mit dem menschlichen Gehirn vergleichbar, da sie aus vielen interagierenden Berechnungszellen bestehen, die zur Entscheidungsfindung miteinander kommunizieren [30, Kapitel 3 Abschnitt B]. Dabei fällt der Schritt der Feature-Extraktion weg. Man unterscheidet zwischen *Artificial Neural Networks (ANNs)*, *CNNs* und *Recurrent Neural Networks (RNNs)*. Im Rahmen dieser Arbeit wird verstärkt auf die aktuell weit verbreiteten *CNNs* eingegangen, da diese für viele Anwendungsfälle den besten Kompromiss zwischen Genauigkeit und Geschwindigkeit darstellen.

State of The Art Methoden zur Objekterkennung basieren auf *Deep Convolutional Neural Networks (DCNNs)*. Dabei kann laut Yurtsever et al. zwischen „Single Stage Detection Frameworks“, die die Position und Klassifikation simultan durchführen und „Region Proposal Detection Frameworks“ unterscheiden werden, bei denen die Kategorisierung in einem zusätzlichen Schritt erfolgt [31, Kapitel V, Absatz A].

2 Technische Grundlagen

Aktuell sind DCNNs - besonders Single-Stage-Algorithmen - aufgrund der hohen Erkennungsrate in kurzer Zeit besonders relevant. Aufgrund der weiten Verbreitung und somit aktiven Weiterentwicklung, dem vergleichsweise einfachen Prozess zum Einlernen neuer Objekte und einem für das Projekt geeigneten Kompromiss zwischen Geschwindigkeit und Genauigkeit wird hier der Fokus auf den YOLO-Algorithmus gelegt.

Beim ursprünglich von Redmon et al. entwickelten YOLO-Algorithmus [32] handelt es sich um einen *Single-Stage Detector*. Anders als bei *Two-Stage* Algorithmen werden nicht erst Bildabschnitte, die Objekte enthalten könnten, vorgeschlagen. Stattdessen findet die Objekterkennung, -klassifikation und Wahrscheinlichkeitsvorhersage in einem Schritt statt [33]. Der Algorithmus teilt Bilder in Regionen auf und erkennt und lokalisiert Objekte in allen Teilen gleichzeitig. Dabei werden sogenannte „Bounding Boxes“, die sich mit den Eckpunkten der Objekte decken, generiert [34, Absatz 2.1.3].

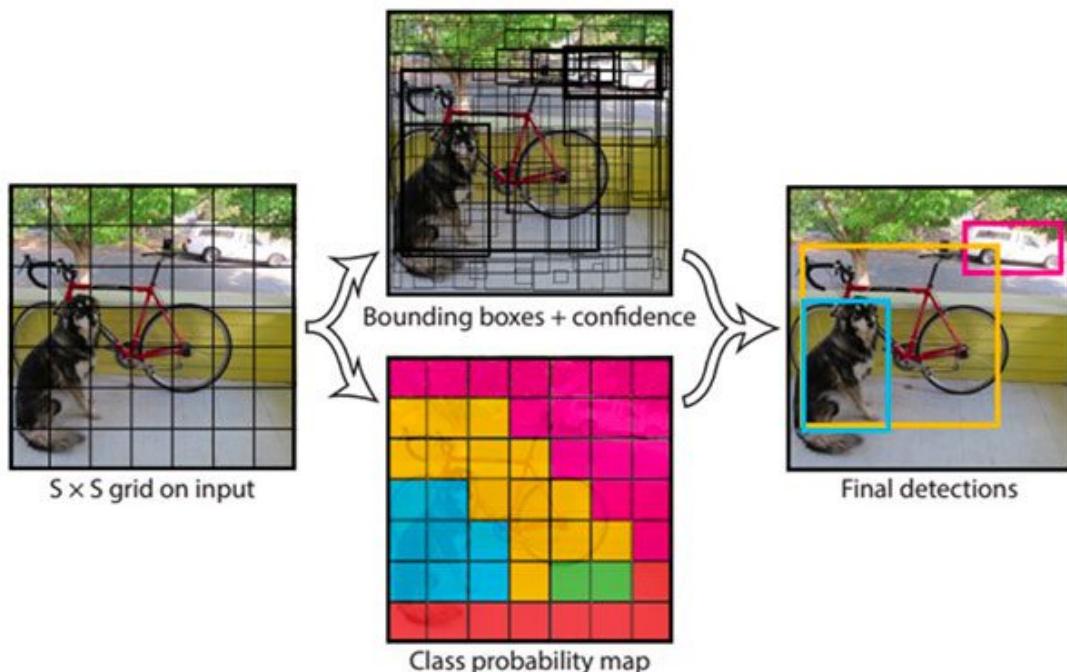


Abbildung 2.4: Vereinfachte Struktur von YOLO [32]

Abbildung 2.4 zeigt die Zerlegung zur Generierung von *Bounding Boxes*. In allen Teilen des Rasters wird eine Objekterkennung und -lokalisierung durchgeführt, auf die die Generierung von *Bounding Boxes* mit Koordinaten relativ zum jeweiligen Teil folgt. Duplikate, die durch über mehrere Teile reichende Objekte entstehen, werden in einem mehrstufigen Prozess mithilfe von *Non-Maximum Suppression* zusammengefasst. [33, Abschnitt 4]

2 Technische Grundlagen

Ein *DCNN* wird hier genutzt, um Bildmerkmale zu extrahieren, was zu einer Verringerung der Auflösung und somit geringeren Ressourcenanforderungen führt [31, Kapitel V, Absatz A]. Eine Implementierung des Algorithmus wurde von Redmon selbst unter dem Namen „Darknet“ veröffentlicht [35]. Nach der Ankündigung von Redmon, den Algorithmus aufgrund ethischer Bedenken nicht weiterzuentwickeln [36], wurde 2020 eine weitere auf *Darknet* basierende Version von Bochkovskiy et al. veröffentlicht [37].

Im Rahmen dieser Arbeit wird die Version *YOLOv3 Tiny* genutzt. Dabei handelt es sich um eine Version, die speziell auf schwache Hardware ausgelegt ist und somit im Anwendungsfall mit einer hohen Bildwiederholrate betrieben werden kann. Diese ist zur Ermittlung des Objektmittelpunktes bei der Annäherung an die Bauteile vorteilhaft. Die Integration des *Darknet* Frameworks in ROS erfolgt mit dem Paket *darknet_ros* [38].

3 Software und Hardware

Folgendes Kapitel umfasst grundlegende Hardwareanforderungen, die verwendete Software und vorbereitende Arbeiten zur Verwendung des Kamerasytems und zum Einlernen benutzerdefinierter Objekte.

3.1 Konfiguration des Kamerasytems

Für die Bearbeitung wurde ursprünglich eine Microsoft Kinect V2 Kamera verwendet. Aufgrund von Hardwarekompatibilitäts- und Treiberproblemen zwischen Kinect und aktueller Hardware und dem schlechten ROS-Treibersupport der Kinect für den Anwendungsfall, wurde im späteren Verlauf eine Intel RealSense 435 verwendet. Diese bietet zudem die Möglichkeit einer Befestigung an der Roboterhand. Weitere Tests wurden somit ausschließlich mit der RealSense durchgeführt.

3.1.1 Technische Daten

Folgender Abschnitt behandelt die technischen Daten der Microsoft Kinect V2 und der Intel RealSense 435.

Microsoft Kinect Bei der Microsoft Kinect handelt es sich um ein ursprünglich für die Spielkonsole Xbox entwickeltes Kamerasystem, das mit einer RGB- und Infrarotkamera zur Tiefenmessung ausgestattet ist. Die RGB-Kamera löst mit $1920 \cdot 1080$ Pixeln auf, die Infrarotkamera mit $512 \cdot 424$. Beide Kameras besitzen eine maximale Bildwiederholrate von 30 Bildern pro Sekunde [39].

Die Kinect wird aufgrund der vergleichsweise geringen Kosten und der Verfügbarkeit vieler Open-Source-Programme oft in der Forschung der Robotik eingesetzt. Da jedoch kaum Kompatibilität mit aktuellen Systemen vorhanden ist und kein direkter Nachfolger der Kinect V2 existiert, werden jedoch zunehmend andere Kamerasysteme wie die Intel RealSense eingesetzt.

3 Software und Hardware

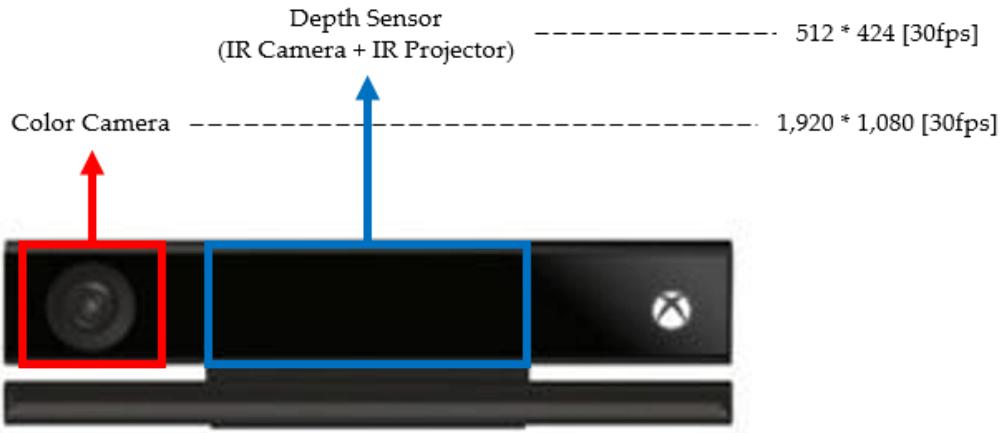


Abbildung 3.1: Aufbau der Microsoft Kinect v2 [39, Abbildung 5]

Die minimale Distanz der Kinect V2 beträgt aufgrund der Sättigung des Sensors durch das reflektierte Infrarotlicht ungefähr $0,5m$ [40, Kapitel 2.1]. Die Abweichung nimmt mit steigender Entfernung zwischen Kamera und Bauteil leicht zu. Während ein hoher Offset von $-0,02m$ vorhanden ist, beträgt die Standardabweichung auch bei Entfernungen über $1m$ unter $0,002m$, was für den Anwendungsfall ausreichend genau ist (s. **Abbildung 3.2**).

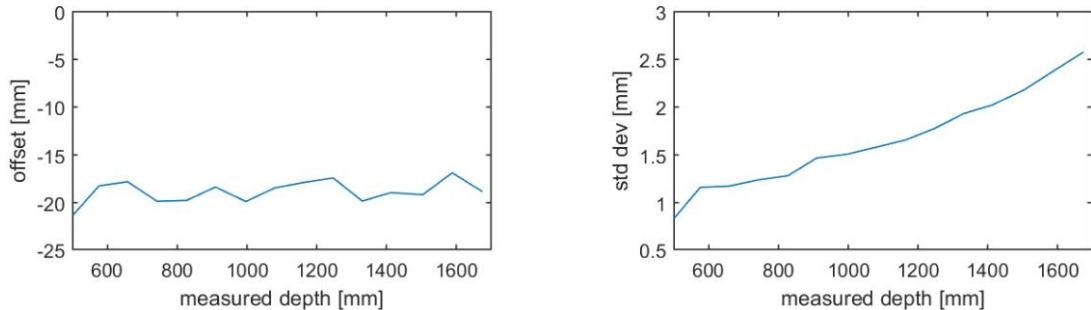


Abbildung 3.2: Genauigkeit der Microsoft Kinect v2 [41, Abbildung 5]

Für die Verwendung der Kinect unter Linux-Systemen wird aufgrund der fehlenden offiziellen Herstellerunterstützung der inoffizielle *libfreenect2* Treiber verwendet. Mit diesem können Informationen der RGB- und Infrarotkamera unter Linux ausgelesen werden [42]. Als Schnittstelle zwischen Kinect und ROS Noetic wird das Paket *iai_kinect2* [43] genutzt. Es stellt neben Funktionen zur Interaktion mit der Kinect über *Topics* Werkzeuge zur Kalibrierung und Anzeige der Kamerabilder zur Verfügung.

3 Software und Hardware

Intel Realsense Die Intel Realsense 435 hat im Vergleich zur Kinect V2 eine höhere Tiefenauflösung. Im Gegensatz zur Kinect wird hier nicht die ToF-Technik verwendet, sondern die Tiefe unter Verwendung einer Stereokamera berechnet (s. **Abbildung 3.3**).

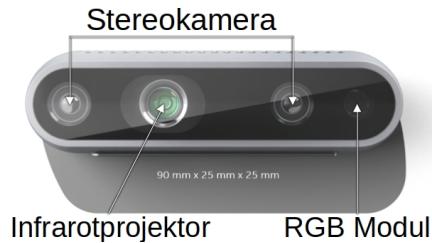


Abbildung 3.3: Aufbau der Intel RealSense 435, Abbildung nach [44]

Um die Distanzdaten zu ermitteln, ist außerdem ein Infrarotlicht zur Beleuchtung des Aufnahmefeldes integriert, das die Genauigkeit mithilfe der Projektion von Beleuchtungsmustern erhöht (s. **Abbildung 3.4**). Durch die geringe Baugröße von $90\text{mm} \cdot 25\text{mm} \cdot 25\text{mm}$ und die Kombination Strom- und Datenfluss in einem Kabel kann die RealSense mit geringem Aufwand am Handstück des Roboterarms befestigt werden. Die für Stereokameras typische ungenaue Tiefenmessung im Vergleich zu ToF-Systemen wird durch den Infrarotprojektor kompensiert, weshalb die Kamera für diese Anwendung geeignet ist.

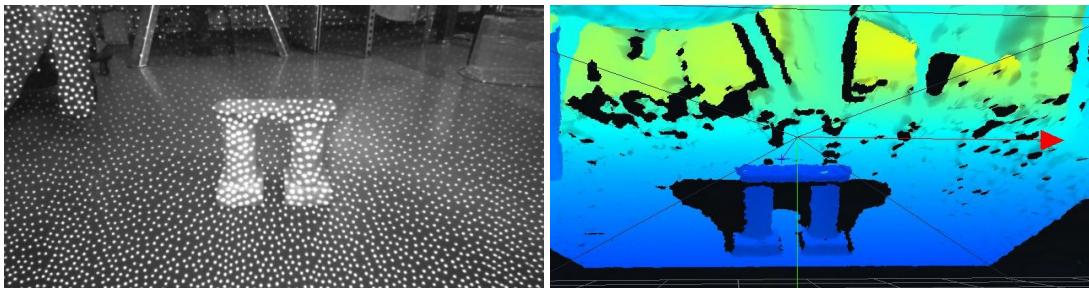


Abbildung 3.4: Infrarotproj. und 3D-Bild der Intel RealSense 435

Zur Ermittlung der Genauigkeit wurden Präzisionstests für die Distanz zwischen Kamera und Bauteil durchgeführt. Die Messungen wurden in einem Distanzbereich zwischen $0,3\text{m}$ und $1,0\text{m}$ getätigt, da die minimal mögliche Distanz der RealSense zur Tiefenmessung $0,28\text{m}$ beträgt. Diese Distanz ist auf die Funktionsweise der Stereokamera-Technik zurückzuführen. **Abbildung 3.5** zeigt die prozentuale mittlere Abweichung des gemessenen Abstandes zwischen Kamera- und Bauteiloberfläche in Bezug auf den realen Abstand. Dabei überschreitet die Abweichung in keinem Fall -2% . Bei einem Offset der Messwerte von $+0,001\text{m}$ beträgt die maximale Abweichung $\pm 1\%$. Bei geringen Entfernung ist sie durchschnittlich kleiner. Die im Versuch gemessene Abweichung mit einem Wert von maximal 2% stimmt mit der von Intel angegebenen Genauigkeit überein [44].

3 Software und Hardware

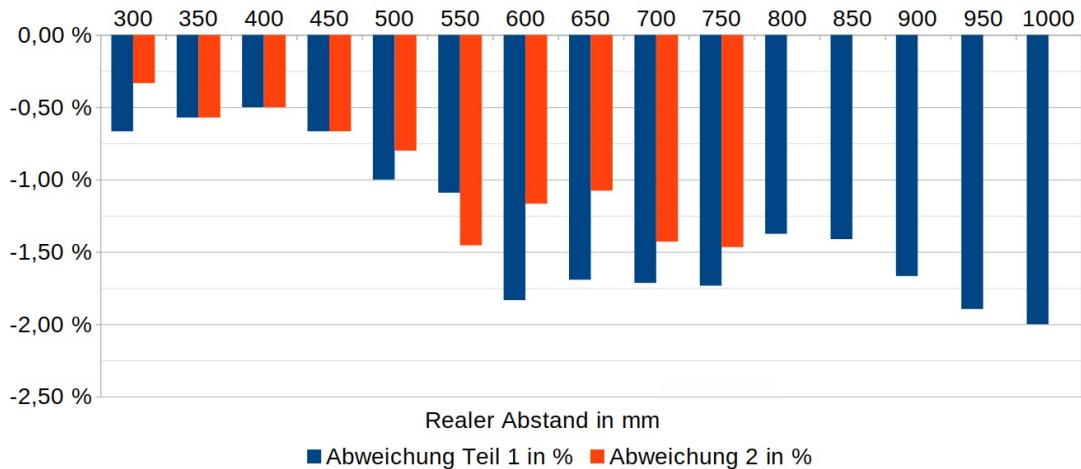


Abbildung 3.5: Genauigkeit der Intel RealSense 435

Die Genauigkeit auf planarer Ebene variiert mit Änderung der Entfernung, verschiedenen Bildauflösungen, Version des verwendeten Algorithmus und dem für den Einlernprozess verwendeten Datensatz. Aufgrund der dadurch entstehenden hohen Variabilität sind Tests dieser im Rahmen der Arbeit nicht zielführend. Bei allen durchgeführten Tests war die Erkennung jedoch für den Anwendungsfall mit einer Abweichung von wenigen Millimetern ausreichend genau.

Intel stellt für die Verwendung der RealSense unter Linux das *LibRealSense open source Software Development Kit (SDK)* [45] bereit. Mit diesem können, ähnlich wie beim Treiber der Kinect, Daten der RGB-Kamera ausgelesen werden. Die 3D-Daten werden in diesem Fall über eine Kombination der Bilddaten beider Kameras berechnet und nicht wie bei ToF direkt gemessen. Der Treiber stellt *Point Clouds* - wobei es sich um dreidimensionale, aus einzelnen Punkten zusammengesetzte Bilder handelt - mit optionaler Überlagerung der Farbinformationen zur Verfügung.

Die Integration in das ROS erfolgt mit dem offiziellen *realsense-ros Package* [46]. Dieses stellt die notwendigen *Topics* für die Verwendung des Bilderkennungsalgorithmus und der Abstandsmessung bereit. Dazu zählt neben dem RGB-Bild und den Hardwareinformationen die Integration der *Point Cloud*, mit der der Abstand zwischen Objekt und Kamerasystem bestimmt werden kann (s. **Abbildung 3.6**). Zum Starten des in *realsense-ros* enthaltenen *realsense2_camera*-Pakets wird eine Konfiguration verwendet, die zum Erreichen einer höheren Kompatibilität mit anderen ROS-Paketen die *Point Cloud* nach dem *OpenNI*-Standard bereitstellt (s. **Anhang B**). Dabei handelt es sich um einen weit verbreiteten, offiziell eingestellten aber weiterentwickelten, Standard zur Verbesserung der Interoperabilität zwischen Software und Hardware [47].

3 Software und Hardware



Abbildung 3.6: RealSense Point Cloud in Rviz

3.1.2 Kalibrierung

Für die Positionserkennung muss eine Transformation zwischen Roboter- und Kamerakoordinatensystem durchgeführt werden. Diese kann mithilfe einer Hand-Augen-Kalibrierung berechnet werden. Außerdem ist ein Abgleich der Koordinaten zwischen RGB- und Tiefensensor möglich, da durch Ungenauigkeiten bei der Kameraausrichtung und Bildverzerrung Koordinaten in x - y -Ebene abweichen können.

3.1.2.1 RGB-D Kalibrierung

Zum Abgleich zwischen RGB- und Tiefenkamera kann eine RGB-D Kalibrierung durchgeführt werden.

Microsoft Kinect Für die Durchführung mit einer Kinect V2 kann beispielsweise das im Paket *iai_kinect2* enthaltene Programm zur Kalibrierung genutzt werden [43]. Diese gleicht Abweichungen durch Verzerrung, Rotation und Verschiebung zwischen den Kamerasystemen aus [48, Beispieldaten].

Die Kalibrierung erfolgt in diesem Fall mithilfe eines Schachbrett-Musters. Dieses kann bei bekannten Kameraparametern der 2D-RGB-Kamera zur dreidimensionalen Positionsbestimmung eines Objekts im Raum genutzt werden [49]. Somit ist eine Berechnung der Abweichung zwischen den von den 2D- und 3D-Kameras erkannten Parametern möglich.

3 Software und Hardware

Um die Kalibrierung durchzuführen, wird die Kinect nacheinander in mehreren Positionen relativ zum Kalibrierungsmuster aufgestellt. Nach jeder Bewegung des Musters wird ein Bild mit beiden Kameras der Kinect gemacht, was zum Abgleich genutzt werden kann. Es wird in der Dokumentation von *iai_kinect2* empfohlen, den Abstand und die Neigung des Kalibrierungsmusters zu verändern, um ein gutes Ergebnis zu erzielen [43].

Intel RealSense Die Intel RealSense ist im Gegensatz zur Kinect bei Verwendung des *iai_kinect2*-Treivers ab Werk kalibriert. Eine klassische manuelle Kalibrierung ohne spezielle Hardware führt laut Intel mit hoher Wahrscheinlichkeit zu einer Verschlechterung der Kalibrierung. Die aktuelle Firmware der RealSense bietet jedoch Optimierungsmöglichkeiten für diverse Parameter wie Ebenheit der *Point Cloud* oder Distanzgenauigkeit. Diese sind hauptsächlich auf Abnutzungerscheinungen nach längerer Nutzung ausgelegt [50]. Da es sich bei der verwendeten RealSense um ein neues Gerät mit ausreichender Genauigkeit handelt (s. **Abbildung 3.5**), wird auf eine Kalibrierung verzichtet.

3.1.2.2 Hand-Auge-Kalibrierung

Als Hand-Augen-Kalibrierung bezeichnet man die Ermittlung von homogenen Transformationsmatrizen zwischen dem Endeffektor des Roboters und dem Kamerasytem sowie die Transformation zwischen Welt- und Basiskoordinatensystem [51, Kapitel 1].

Man unterscheidet hier zwischen zwei Fällen: Die Kamera kann entweder stationär nahe dem Roboter oder wie im vorliegenden Fall am Endeffektor befestigt werden. Letztere Möglichkeit hat den Vorteil, dass das Kamerasytem durch Anfahren des Bauteilmittelpunktes und mit einer geringen Entfernung zwischen Kamera und Objektoberfläche eine präzisere Steuerung erzielen kann.

Die Kalibrierung in ROS kann mit dem *moveit_calibration*-Paket [52] durchgeführt werden. Dieses unterstützt beide erwähnten Konfigurationsmöglichkeiten. Die vom Programm durchgeführten Messungen erfolgen anhand eines visuellen Kalibermusters mit distinktiven Formen, mit dem bei bekannter Größe die Lage des Kamerakoordinatensystems bestimmt werden kann. Dazu sollten zum Erfassen mindestens fünf Posen angefahren werden [53].

Für den Anwendungsfall dieser Arbeit ist die manuelle Koordinatentransformation ausreichend, da die Lage des Kamerakoordinatensystems im Ursprung der

3 Software und Hardware

RealSense nur in Relation zum Koordinatensystem, das vom Bewegungsplanungsprogramm von Steinbeck [3] verwendet wird, berechnet werden muss. Die Koordinatensysteme haben eine rotationsfreie Verschiebung. Die durchgeführte Transformation ist im **Abschnitt 3.2.5** näher beschrieben.

3.1.3 Befestigung

Die RealSense kann aufgrund der geringen Baugröße und niedriger Erschütterungsanfälligkeit in der Nähe des Endeffektors am Roboter befestigt werden. Datenübertragung und die Stromversorgung erfolgen über ein einziges USB-Kabel. Aufgrund des größeren Sichtfeldes und besserer Abdeckung des Arbeitsraumes wird die Kamera mit einer Drehung von 90° gegenüber dem Endeffektor montiert (s. **Abbildung 3.7**, oben).

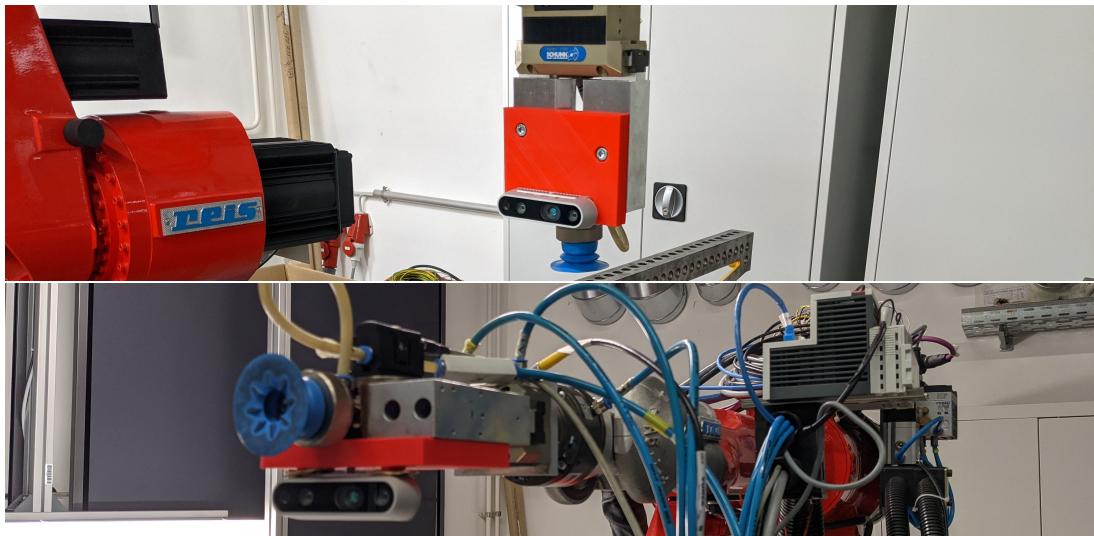


Abbildung 3.7: Befestigung der Intel RealSense am Endeffektor

3.2 Implementierung der Objekterkennung

YOLO eignet sich aufgrund schneller und ressourcenschonender Erkennung und Lokalisierung von Objekten ohne große Einschränkungen der Genauigkeit gut für den Anwendungsfall dieser Arbeit. Pro Zelle des Bildrasters können in YOLOv1 bis zu drei, in YOLOv3 bis zu neun Objekte gleichzeitig erkannt und lokalisiert werden [33].

3.2.1 Voraussetzungen für das Objekterkennungsprogramm

Wegen der weitreichenden Kompatibilität und der langen Support-Dauer wird die Version ROS-I verwendet. Verglichen mit dem verhältnismäßig neuen ROS-II ist ein breiteres Spektrum an kompatibler Hardware und ausgereiften Paketen verfügbar. Bei Noetic handelt es sich um die aktuelle *LTS* Version des ROS-I-Systems, die bis Mai 2025 unterstützt wird [15, Abschnitt 3].

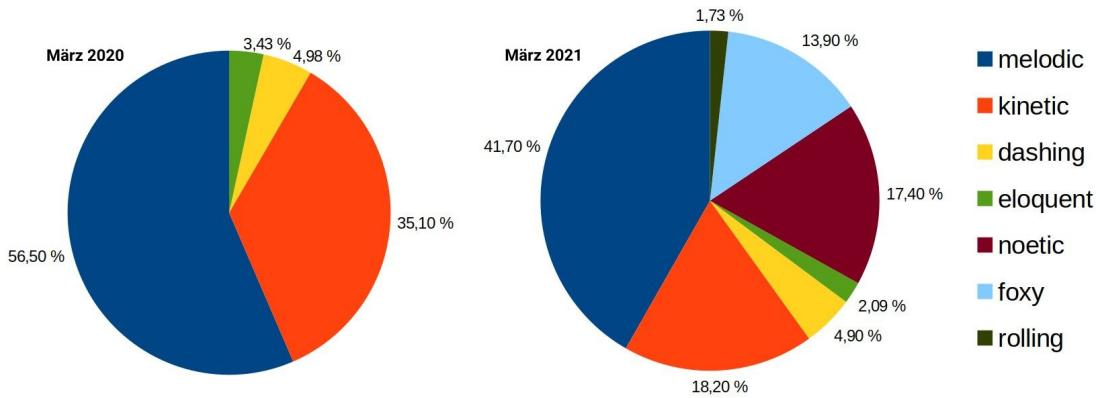


Abbildung 3.8: Anzahl der ROS-Downloads im März 2020 und 2021, Grafik nach [54]

Für die Ausführung von ROS-I Noetic wird ein Computer benötigt, der die Hardwareanforderungen von *Ubuntu 20.04 LTS* erfüllt. Bei den Funktionstests wurde ein Computer mit x86-64-Architektur verwendet, bei entsprechender Treiberunterstützung des Kamerasytems können aber auch ARM-basierte Prozessoren genutzt werden. Für den Einlernprozess der Objekte (s. **Abschnitt 3.2.2**) wird aufgrund der Notwendigkeit von *Nvidia Compute Unified Device Architecture (CUDA)* eine Grafikkarte des Herstellers *Nvidia* benötigt. Zur Objekterkennung ist die Herstellerbindung nicht zwingend, aber für die Verwendung der Hardwarebeschleunigung notwendig. Die Erkennung über den Hauptprozessor des Computers ist somit grundsätzlich möglich, wird jedoch aufgrund der deutlich langsameren Verarbeitung der Objekte nicht empfohlen [38].

Zur Nutzung des Programms und der Visualisierung werden zusätzlich zum ROS-Hauptpaket das 3D-Visualisierungspaket *RViz* [55] und das Bewegungsplanungs-Framework *Moveit* [56] benötigt. Nähere Informationen zu *Moveit* sind in der Arbeit von Steinbeck aufgeführt [3, Abschnitt 3.2].

Für die Aufnahme der Objekte müssen deren Art und Position bekannt sein. Durch die Kombination der zwei- und dreidimensionalen Bilddaten kann die Erkennung verschiedener Objekte mithilfe des RGB-Bildes erfolgen. Nach Berechnung des Mittelpunktes der *Bounding Boxes* kann der Abstand zwischen Kamerasytem und der Objektoberfläche mithilfe des 3D-Bildes bestimmt werden. Somit sind alle Koordinaten der vom Greifer anzufahrenden Position bekannt.

3.2.2 Training des YOLO-Algorithmus

Zur Erkennung der in der Arbeit verwendeten Objekte müssen diese durch einen Einlernprozess als sogenannte *Weights* gespeichert werden. Von jedem Objekt werden dazu zuerst mehrere Bilder mit verschiedenen Orientierungen, Lichtverhältnissen und in unterschiedlichen Umgebungen gemacht. Die Genauigkeit steigt üblicherweise mit der Anzahl der Bilder und den Iterationen im Einlernprozess. Diese Bilder müssen anschließend mit *Bounding Boxes* und Objektnamen versehen werden. Die *Bounding Boxes* werden hier mit dem Programm *LabelImg* [57] erstellt, das das *YOLO* Format nativ unterstützt. Sie werden so genau wie möglich auf das Objekt angepasst, sodass die äußereren Punkte die Kanten des Rechtecks berühren. So ist eine möglichst genaue Positionserkennung im späteren Verlauf möglich. Der Datensatz ist unabhängig von der verwendeten *YOLO*-Version anwendbar. **Abbildung 3.9** zeigt beispielhaft den Aufbau einer Textdatei zur Beschreibung des Labels. Für das Training des neuronalen Netzes wird für jedes Bild eine solche Datei benötigt. Diese setzt sich aus einer Nummer für die Klasse des Objekts, dem Mittelpunkt der *Bounding Box* und der Breite und Höhe dieser zusammen. Die Nummer der Objektklasse richtet sich hier nach der Reihenfolge in einer weiteren Textdatei, die die Klassennamen enthält. Pro Bild können auch mehrere, durch Zeilenumbrüche getrennte, *Bounding Boxes* definiert werden.

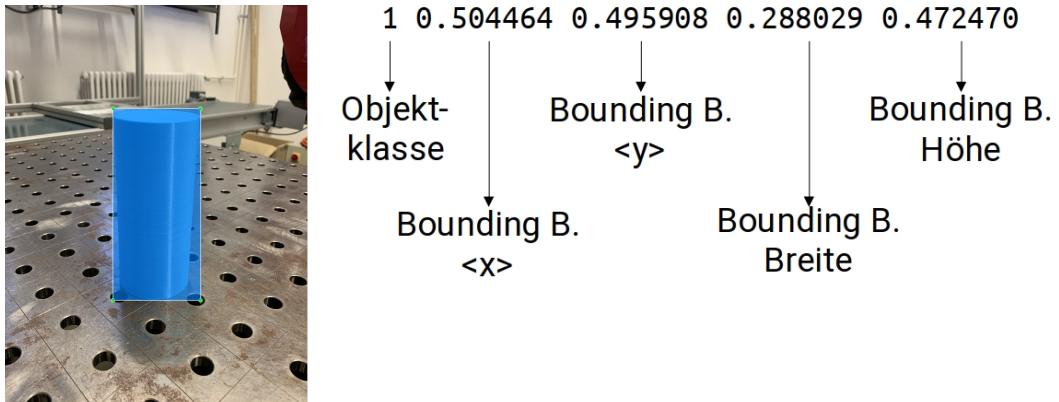


Abbildung 3.9: YOLO Label

Das *Darknet* Framework beinhaltet Tools zum Training mit dem durch beispielsweise *LabelImg* erstellten Datensatz. Dieser besteht aus Bildern, in denen ein oder mehrere zu erkennende Objekte enthalten sind und den erwähnten Textdateien mit Informationen über die *Bounding Box*-Positionen. Das Training muss mit der gleichen Version des Algorithmus erfolgen, die später zur Erkennung genutzt wird.

3 Software und Hardware

Für das Training des verwendeten *YOLOv3 Tiny*-Algorithmus wurde die in **Anhang D** aufgeführte Konfiguration mit einem Datensatz aus ungefähr 300 Bildern pro Objekt genutzt. Die generierten *Weights* können an die entsprechende Stelle des *rv6l_3d*-Programms kopiert werden, um eine Objekterkennung zu ermöglichen. Der Einlernprozess wurde nach 10000 Iterationen gestoppt, da die Genauigkeit bei den verwendeten Objekten mit mehr Wiederholungen nur wenig erhöht wird (s. **Abbildung 3.10**).

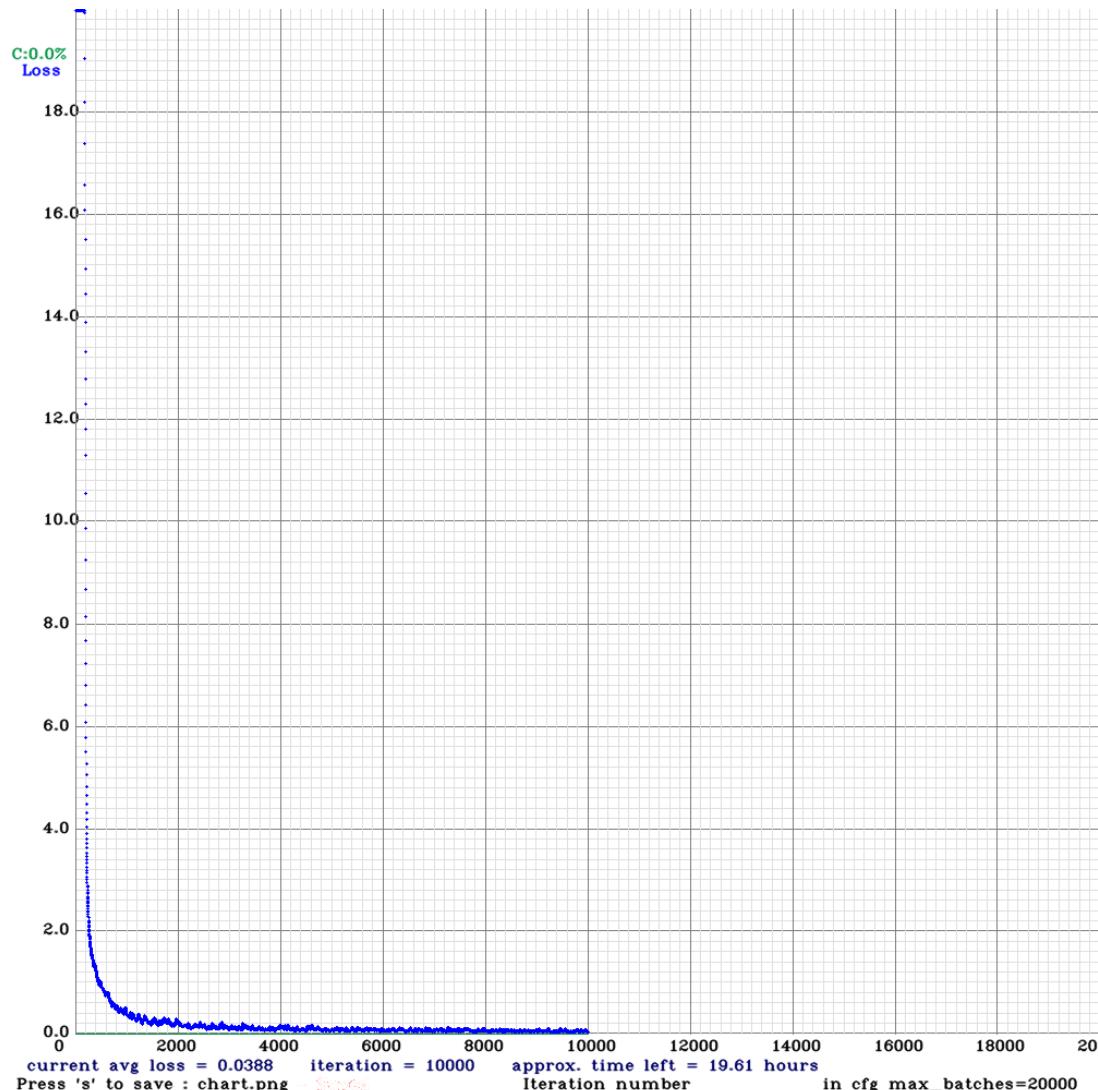


Abbildung 3.10: Darknet Iterationen [generiert mit Darknet]

3.2.3 rv6l_3d-Paket

Die Grundlage für die Positionserkennung mit *Darknet* in ROS bildet das *rv6l_3d*-Paket (s. **Anhang A**). Dabei handelt es sich um ein für diese Anwendung erstelltes Paket für ROS, das auf dem 2016 von IntelligentRoboticsLabs veröffentlichten *darknet_ros_3d* [58] basiert. Dieses baut wiederum auf *darknet_ros* [38] auf. Eine Übersicht der Zusammenhänge zwischen den Paketen ist in **Abbildung 4.1** zu finden. Modifikationen umfassen die Entfernung nicht benötigter Funktionen, eine Mittelpunktberechnung der *Bounding Boxes* und Anpassungen auf das Programm zur Handhabung der Bauteile. Die Kernfunktionalität des *rv6l_3d*-Pakets besteht in der Bereitstellung dreidimensionaler Objektmittelpunktkoordinaten unter Verwendung einer 3D-Kamera. Das Projekt nutzt die von *darknet_ros* zur Verfügung gestellten Daten zur Erkennung der planaren Objektposition und gleicht diese mit dem von einer Tiefenkamera gemessenen oder Stereokamera berechneten dreidimensionalen Bild ab, um den Abstand zwischen Kamera und Bauteiloberfläche zu ermitteln.

Die wichtigsten Modifikationen des Pakets umfassen das Entfernen der Marker, die zur Visualisierung der *Bounding Boxes* in *Rviz* dienen, die Berechnung der Objektmittelpunkte sowie Anpassungen des *Topics* mit den Objektpositionen. Somit werden statt vollständigen *Bounding Boxes* die Koordinaten der *Bounding-Box*-Mitten auf der Objektoberfläche in Bezug zum Kamerakoordinatensystem durch den *Publisher* zur Verfügung gestellt. Außerdem wird die Objekterkennung visualisiert (s. **Abbildung 3.11**).

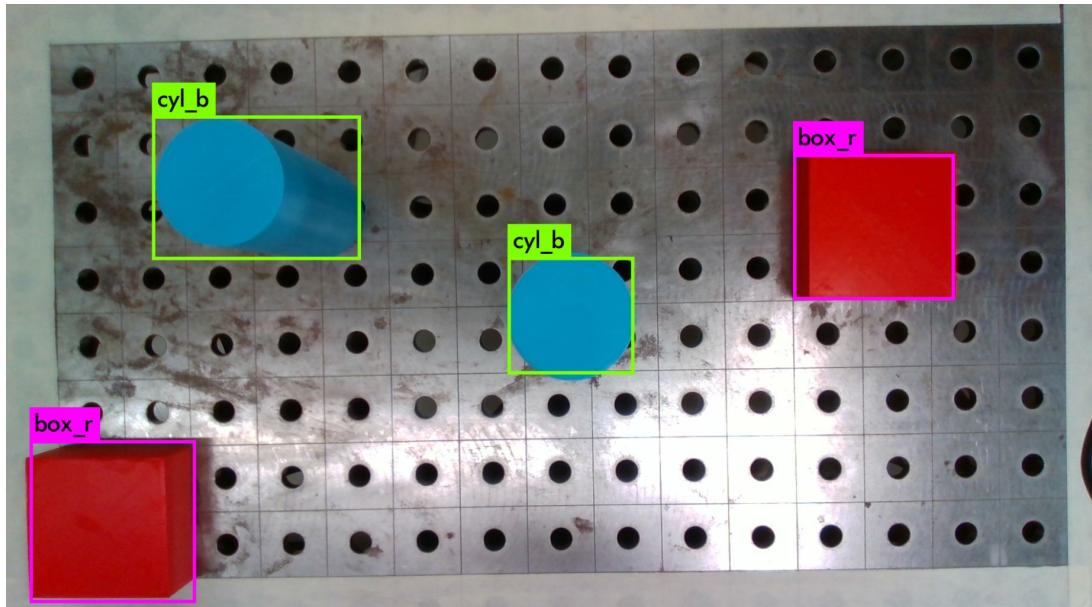


Abbildung 3.11: *rv6l_3d* Bounding Boxes

3 Software und Hardware

Die Rückgabe des *Bounding Box Publishers* ist im Paket folgendermaßen umgesetzt (s. **Anhang A**) [C++]:

```
1 [ . . . ]
2     midx = (maxx + minx) / 2; // Objektmittelpunkt (x)
3     midy = (maxy + miny) / 2; // Objektmittelpunkt (y)
4     midz = (maxz + minz) / 2; // Objektmittelpunkt (z)
5 }
6
7 rv6l_3d_msgs::RV6L_position bbx_msg;
8 bbx_msg.Class = bbx.Class;           // Klasse
9 bbx_msg.probability = bbx.probability; // Wahrscheinlichkeit
10
11 bbx_msg.w = midy; // "horizontale" Position im Bild
12 bbx_msg.h = midz; // "vertikale" Position im Bild
13 bbx_msg.d = minx; // Abstand (min) durch Point Cloud
14
15 boxes->bounding_boxes.push_back(bbx_msg); // Rueckgabe an
     BB Message
16 [ . . . ]
```

Die modifizierte Message des *rv6l_3d*-Pakets setzt sich somit aus diesen Werten zusammen:

```
1 string Class          # Objektklasse
2 float32 probability   # Confidence
3 float32 w              # horizontale Koordinate im Bild (width)
4 float32 h              # vertikale Koordinate im Bild (height)
5 float32 d              # Abstand zum Objektmittelpunkt (depth)
```

Es werden Objektklasse, Erkennungssicherheit und die Mittelpunktkoordinaten ausgegeben. Der Datentyp *float32* wird von *C++* und *Python* als *float* interpretiert, weshalb er sich gut für die Übermittlung eignet. Die Koordinaten der Objekte werden relativ zum Kameramittelpunkt unter Berücksichtigung des Abstands zum Objekt in Metern ausgegeben. Dabei ist *w* in Richtung Bildoberseite und *h* in Richtung der rechten Bildkante positiv. Der Abstand *d* beginnt mit 0 an der Kameraoberfläche und kann nur positive Werte annehmen. Bei der Abstandsberechnung wird der Abstand zwischen der *h-w*-Ebene der Kamera (s. **Abbildung 3.12**) und der *x-y*-Ebene des Basiskoordinatensystems genutzt, sodass dieser unabhängig vom horizontalen Versatz zwischen Objekt und Kamerasytem immer die gleiche Distanz zurückgibt.

Das Programm gibt die ermittelten Positionen in Form eines *Topics* an den *Subscriber* des Programms für die Bewegungssteuerung des Roboters weiter:

3 Software und Hardware

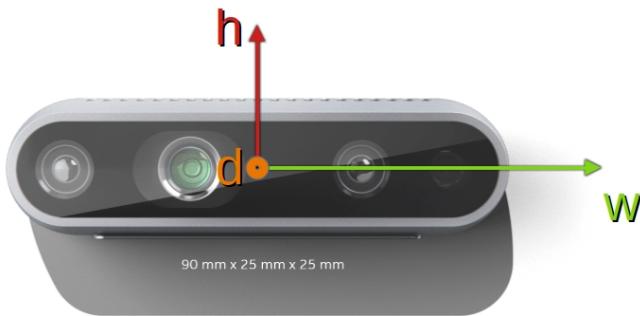


Abbildung 3.12: Koordinatensystem der Intel RealSense, Abbildung nach [44]

3.2.4 Bounding Box Subscriber

Der *Subscriber* nimmt die vom *rv6l_3d* ermittelten Rückgabewerte auf und speichert diese in einer Liste (s. **Anhang C**) [Python]:

```
1 [ . . . ]
2     def callback_pos(self, data):
3         for box in data.bounding_boxes:
4             self.object_pos_cam = [box.Class, box.h,
5                                     box.w,
5                                     box.d]
5 [ . . . ]
```

Diese Liste kann dann vom Bewegungssteuerungsprogramm (basierend auf Steinbeck [3, Abschnitt 3.5.2]) zum Anfahren der Zielpositionen genutzt werden. Dieser Algorithmus wurde zur Nutzung mit der Objekterkennung folgendermaßen angepasst:

3.2.5 Algorithmus zur Positionserkennung

Für die Positionserkennung müssen die Koordinaten des Objekts in Relation zum Basiskoordinatensystem bekannt sein:

$${}^G\underline{p}(G \rightarrow Cam) = {}^G\underline{p}(G \rightarrow Tool) + {}^G\underline{p}(Tool \rightarrow Cam)$$

Die aktuelle Transformation vom in **Abbildung 3.13** eingezeichneten Koordinatensystem G in das Kamerakoordinatensystem $Tool$ ${}^G\underline{p}(G \rightarrow Tool)$ kann jederzeit ausgelesen werden. Somit muss für die Bauteilpositionsbestimmung die Transformation ${}^G\underline{p}(Tool \rightarrow Cam)$ durchgeführt werden.

3 Software und Hardware

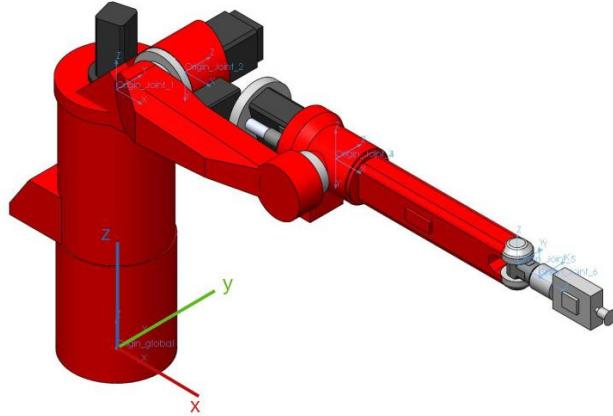


Abbildung 3.13: Koordinatensystem des RV6L, Abbildung nach [59, Abbildung 12]

Die Tabelle in **Abbildung 3.14** zeigt eine Tabelle zur Umrechnung der Orientierungen aller Koordinatensysteme. Diese gilt ausschließlich für die im Test verwendete Orientierung mit einer Drehung von 90° gegenüber dem Endeffektor und einer Ausrichtung des Endeffektors parallel zur x -Achse und zur x - y -Ebene (s. **Abbildung 3.7**). Dabei ändert sich die Orientierung im Schritt der Bilderkennung des verwendeten Algorithmus nicht, weshalb nur die Translation berechnet werden muss. Diese wird hier als Kameraoffset bezeichnet:

$$\begin{aligned}x_{pCam} &= x_{pTool} + o_x \\x_{pObj} &= x_{pCam} + h_{pObj} \\x_{pObj} &= x_{pTool} + o_x + h_{pObj}\end{aligned}$$

$$\begin{aligned}y_{pCam} &= y_{pTool} + o_y \\y_{pObj} &= y_{pCam} + w_{pObj} \\y_{pObj} &= y_{pTool} + o_y + w_{pObj}\end{aligned}$$

$$\begin{aligned}z_{pCam} &= z_{pTool} - o_z \\z_{pObj} &= z_{pCam} - d_{pObj} \\z_{pObj} &= z_{pTool} - o_z - d_{pObj}\end{aligned}$$

Alle Variablen stehen in Bezug zum Koordinatensystem G . $[...]_{pCam}$ beschreibt die Position der Kamera, $[...]_{pTool}$ die des Tool-Koordinatensystems und $[...]_{pObj}$ des Objekts. $o[...]$ ist der Offset zwischen Kamera und Tool.

3 Software und Hardware

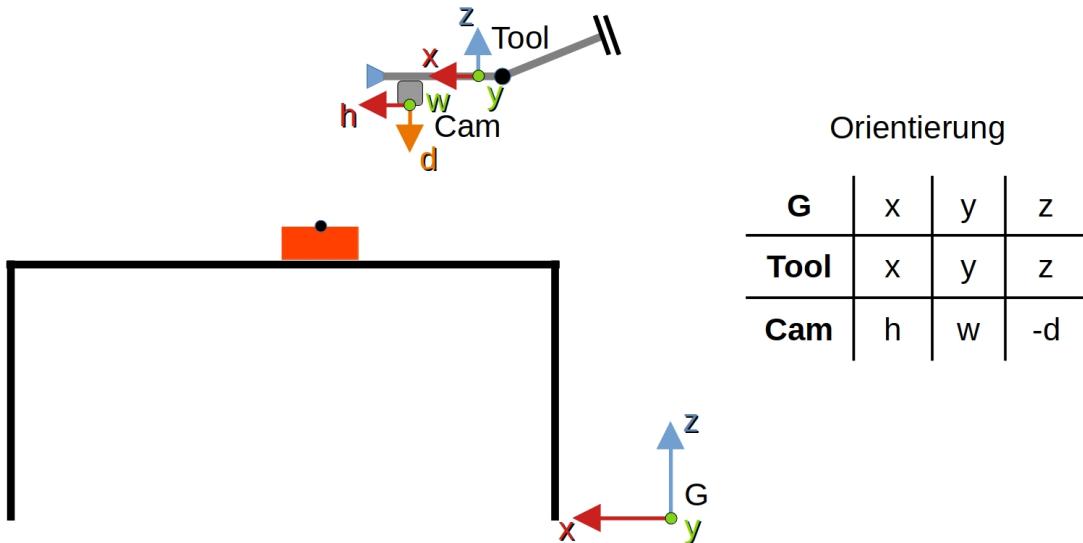


Abbildung 3.14: Objektkoordinaten

Die Positionserkennung ist folgendermaßen implementiert [Python]:³

```

1 STATIC_X = 0.237 # Offset in X-Richtung (Greifposition)
2 STATIC_Y = 0      # Offset in Y-Richtung (Greifposition)
3 STATIC_Z = 0.237 # Offset in Z-Richtung (Greifposition) =
                  STATIC_X durch Orientierungswechsel
4
5 SCAN_NEAR = 0.30 # Hoehe fuer nahen Scan
6
7 [ . . . ]
8
9     py.go_to_scan_pose() # Scanposition anfahren
10
11    ichbinder = uwe() # Klasse aus subscriber_cam
                      initialisieren, Objektpos. auslesen
12    rospy.wait_for_message("/darknet_ros_3d/bounding_boxes/",
                           RV6L_positions, timeout=None) # Warten auf Nachricht
                           mit Objektposition relativ zur Kamera
13    cam_pos = ichbinder.object_pos_cam # Aktuelle Position in
                                         cam_pos speichern
14
15    py.go_to_scan((cam_pos[1] + REF_X), (cam_pos[2] + REF_Y),(
                    REF_Z - cam_pos[3] + SCAN_NEAR)) # Anfahren der
                    Scanposition ~ ueber dem Bauteil
16
17    cam_near = ichbinder.object_pos_cam # Ermitteln des
                                         genauen Objektmittelpunktes
18

```

³Basiert auf [3], (s. **Anhang C**). Unwichtige Funktionsaufrufe und -definitionen wurden zur Verbesserung der Übersichtlichkeit entfernt

3 Software und Hardware

```
19     py.go_to_pose_goal((cam_pos[1] + REF_X + cam_near[1] +
20                         STATIC_X), (cam_pos[2] + REF_Y + cam_near[2] + STATIC_Y
21                         ), (REF_Z - cam_pos[3] + STATIC_Z - (cam_near[3] -
22                           SCAN_NEAR) + 0.25)) # Greifposition anfahren
23
24     py.go_to_pose_goal((cam_pos[1] + REF_X + cam_near[1] +
25                         STATIC_X), (cam_pos[2] + REF_Y + cam_near[2] + STATIC_Y
26                         ), (REF_Z - cam_pos[3] + STATIC_Z - (cam_near[3] -
27                           SCAN_NEAR))) # Endeffektor absenken
```

Der Roboter fährt zuerst in eine Position, die den für die Erkennung nutzbaren Gesamtbereich im Kamerabild sichtbar macht (Zeile 9). Anschließend wird ein Scan durchgeführt und die Objektposition in der Variable `cam_pos` gespeichert (Zeile 13). Diese Position ist im Kamerakoordinatensystem (s. **Abbildung 3.12**) angegeben. Aufgrund der Perspektive und der Funktionsweise der *Bounding Boxes* weicht der Mittelpunkt jedoch vom realen Bauteilmittelpunkt ab. Diese Abweichung ist in **Abbildung 3.11** erkennbar.

Wegen der Abweichung wird diese ungefähre Mittelpunktposition mit dem Mittelpunkt der Kamera in der x - y -Ebene $30cm$ über dem Bauteil angefahren. Dadurch ist nur noch die Objektoberseite sichtbar, sodass der genaue Objektmittelpunkt bestimmt werden kann (Zeile 15). Zum Ermitteln der Zielkoordinaten werden die Startkoordinaten und die von der Kamera ermittelten Objektkoordinaten gemäß der in **Abbildung 3.14** dargestellten Umrechnungstabelle addiert. Außerdem werden $30cm$ (Variable `SCAN_NEAR`) zum Anfahren der korrekten Höhe zur z -Koordinate addiert. Der Kameraversatz ist in diesem Fall irrelevant, da er in Start- und Zielposition den gleichen Wert hat und somit eliminiert wird. So mit bewegt sich die am Endeffektor befestigte Kamera beim Aufruf der Funktion `go_to_scan` zum ungefähren Objektmittelpunkt. Nach Anfahren dieser Koordinaten wird der Mittelpunkt des Objekts erneut ermittelt und in der Variable `cam_near` gespeichert (Zeile 17). Dazu wird das Kamerakoordinatensystem verwendet, wodurch nur die Abweichung zwischen Kamera- und Bauteilmittelpunkt gespeichert wird.

Zum Greifen des Bauteils muss der Versatz und die Rotation zwischen Kamera und Endeffektor beachtet werden. Die Variablen `STATIC_<...>` legen diesen Offset zwischen Kamera und *Tool*-Koordinatensystem fest. Sie unterscheiden sich je nach Roboterkinetik und Kameraposition. Im Test wurde die Intel RealSense mit einem Offset von $23,7cm$ in x -Richtung und $-5,7cm$ zur Vorderkante der Kamera in z -Richtung des *Tool*-Koordinatensystems montiert. Somit wird mit `STATIC_X` die x -Koordinate der anzufahrenden Position um $23,7cm$ erhöht, damit das *Tool*-Koordinatensystem in der Greifposition sich mit der x -Koordinate der Kamera in der Scanposition deckt. Im vorliegenden Fall ist aufgrund des Orientierungswechsels zwischen Scan- und Greifposition der Versatz zwischen Kamera und *Tool* von $23,7cm$ in x -Richtung äquivalent zum Versatz der Scan-

3 Software und Hardware

position in z -Richtung (s. **Abbildung 3.15**). Der Kameraoffset kann beispielsweise durch Messungen am realen System ermittelt werden. Im angefügten Quellcode wurde der Wert `STATIC_Z` durch experimentelle Annäherung leicht angepasst.

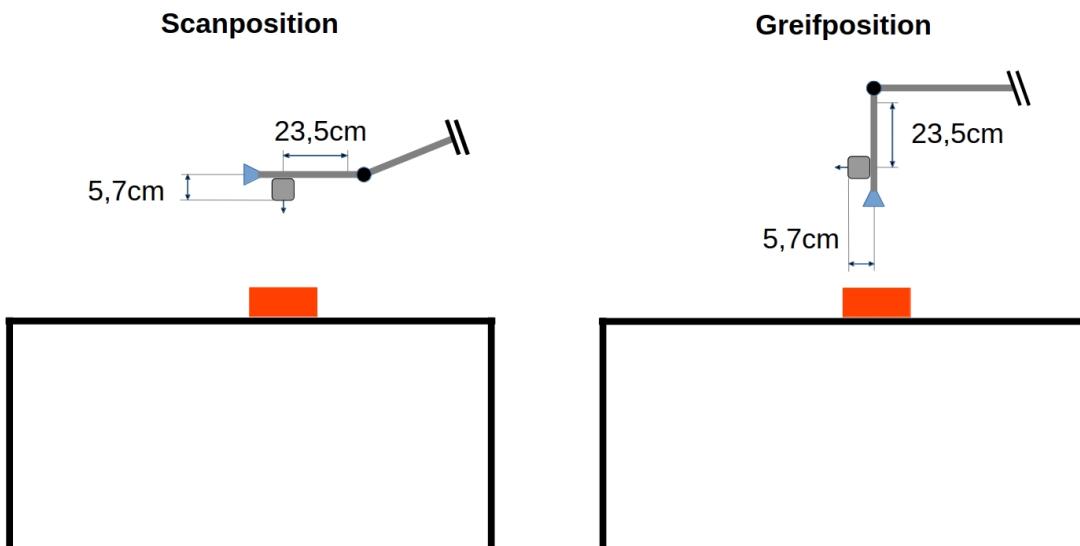


Abbildung 3.15: Roboterpositionen

Im nächsten Schritt wird die Funktion `go_to_pose_goal` aufgerufen (Zeile 19). Diese fährt die Zielposition im Gegensatz zur `go_to_scan`-Funktion nicht in Scan-, sondern in Greifposition an. Somit erfolgt hier der erwähnte Orientierungswechsel. Als Zielkoordinaten werden hier sowohl die in `cam_near` gespeicherten genauen Offset-Werte für den Bauteilmittelpunkt, die anzufahrende Höhe von 25cm vor Absenken des Endeffektors, als auch die Werte der `STATIC_[...]`-Variablen zu den anfangs ermittelten Koordinaten addiert, wodurch die genaue Zielposition angefahren wird. Im letzten Schritt wird der Greifer um die erwähnten 25cm abgesenkt, um das Bauteil zu greifen.

4 Gesamtsystem

Das Gesamtsystem besteht softwareseitig aus einer Kombination öffentlich verfügbarer und eigens erstellter ROS-Pakete. Das Programm ist so gestaltet, dass die Roboter- und Bilderkennungshardware, wenn passende Konfigurationsdateien für Kamerasystem und Roboterspezifikation vorhanden sind, variabel sind. Folgendes Kapitel bietet einen Überblick über die Aufgaben aller verwendeten Pakete und die Struktur des gesamten Systems mit Fokus auf der Objekterkennung sowie den zeitlichen Ablauf eines einzelnen Bewegungsschrittes.

Hinweis: Das in diesem Kapitel beschriebene Gesamtsystem ist noch nicht vollständig implementiert. Die Ansteuerung des Greifers ist aufgrund von Kommunikationsproblemen mit der verwendeten Reis Robotersteuerung nicht funktionsfähig und muss somit manuell bedient werden. Die Schnittstelle zwischen der Objekterkennung und Handhabung funktioniert aktuell ausschließlich, wenn nur ein einzelner beliebiger Block vom Kamerasystem erkannt wird. Zu Demonstrations- und Testzwecken werden daher zwei Algorithmen zur Verfügung gestellt: für die intelligenten Handhabungssequenzen der Teile wird ein fester Bauplan für Start- und Zielposition vorgegeben¹. Zur Nutzung der Bilderkennung in Verbindung mit der Bewegung der Objekte wird der in **Abschnitt 3.2.5** beschriebene Algorithmus genutzt.

4.1 Programmstruktur

Die Zusammenhänge aller Programme sind in **Abbildung 4.1** aufgeführt. Hier wird verstärkt auf die Objekterkennung (rot) eingegangen. Die Roboterbewegung wird in einer Arbeit von Steinbeck [3, Abschnitt 4.1] behandelt.

Die Eingangsdaten des Systems bilden die Daten der beiden RGB-Kameras und der *3D-Point Cloud*. Diese nimmt mithilfe des RealSense-Treibers für ROS ein Bild der Bauteile im Arbeitsbereich auf. Da es sich bei der RealSense um eine Stereokamera handelt, wird aus den Bildern beider Kameras mit Unterstützung der Punkte des Infrarotprojektors eine *Point Cloud* berechnet, die anschließend

¹ausführlich behandelt in [3, Steinbeck, Abschnitt 3.4]

4 Gesamtsystem

vom *rv6l_3d*-Paket verwendet wird. Ein RGB-Bild der Kamera wird außerdem an das *Darknet*-Framework und von dort an *darknet_ros* weitergegeben.

Die Grundlage der Erkennung bildet der *YOLO*-Algorithmus. Dieser wird vom *Darknet*-Framework, was auch zum Training des Algorithmus verwendet wird, zur Erkennung der Objekte genutzt. Es gibt die Objektklasse und jeweils die Koordinaten des oberen linken und unteren rechten Punktes der *Bounding Boxes* zurück. Diese Koordinaten sind in Pixeln mit dem Mittelpunkt als Ursprung des 2D-Koordinatensystems angegeben. Für die Interaktion mit ROS ist das Paket *darknet_ros* zuständig. Dieses nutzt einen *Subscriber* zur Verwendung von Kameradaten und -aufnahmen und gibt relevante Daten an *Darknet* weiter. Daraufhin extrahiert es relevante Informationen, die von *Darknet* zurückgegeben werden. Unter Nutzung dieser Daten erfolgt eine Bereitstellung eigener *Topics*, beispielsweise mit den Koordinaten der *Bounding Boxes*, einer Visualisierung dieser und den Objektklassen.

Da die Position in Pixeln mit dem Abstand der Objekte variiert, werden für die Handhabung der Bauteile aufgrund der variablen Höhe und der wechselnden Kameraposition *Bounding Box*-Koordinaten in einer abstandsunabhängigen Einheit benötigt. Dafür wird von dem in **Abschnitt 3.2.3** erwähnten *rv6l_3d*-Paket unter Nutzung von Abstands- und Hardwareinformationen der 3D-Kamera und den *Bounding Box*-Koordinaten eine Position in Metern relativ zum Kameramittelpunkt berechnet. Durch die Anpassung des Pakets auf den vorliegenden Anwendungsfall erfolgt keine Rückgabe der gesamten *Bounding Box*, sondern der dreidimensionalen Position des Mittelpunktes der Objektoberfläche relativ zum Ursprung des Kamerakoordinatensystems. Dazu werden die Informationen aus dem *darknet_ros*-Paket mit den *Point Cloud*-Daten des verwendeten Kamerasystems abgeglichen.

4 Gesamtsystem

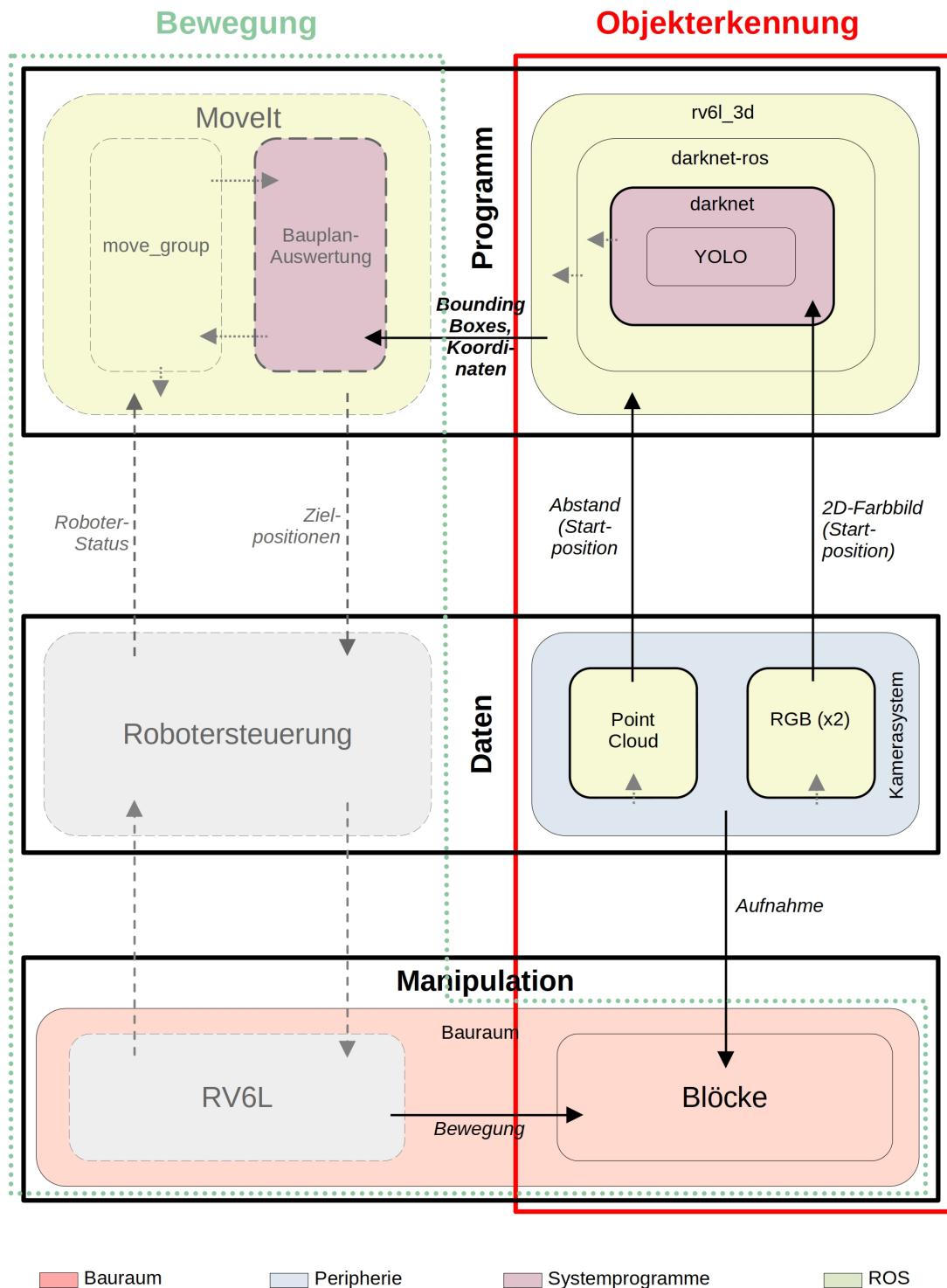


Abbildung 4.1: Struktur des Gesamtsystems

4.2 Programmablauf

Dieser Abschnitt beschreibt die Abläufe beim Start des Gesamtprogramms.

1: Hardware-Interface

Zuerst werden die Programme für die Kommunikation zwischen ROS und der Robotersteuerung und für die Interaktion mit dem realen Roboter gestartet.

2: Bilderkennung

Der für die Objektklassifikation und Positionserkennung zuständige *Node* wird gestartet

3: Bewegungsalgorithmus

Zuletzt wird der Bewegungsalgorithmus gestartet. Dieser nutzt die Daten der Bilderkennung zur Berechnung der Objektpositionen und zum anschließenden Anordnen der Objekte nach dem vorgegebenen Bauplan.

4.3 Bewegungsablauf

Abbildung 4.2 zeigt den Ablauf der gesamten Roboterbewegung nach Programmstart. Dieser kann in folgende Schritte unterteilt werden:

1: Startposition

²Die Startposition kann beliebig gewählt werden, solange es sich nicht um eine singuläre Pose handelt.

2: Position für die Bilderkennung

Diese Position mit einer Höhe von $0,60m$ über der Tischemebene stellt einen Kompromiss zwischen Zuverlässigkeit der Bilderkennung und Abdeckung des Arbeitsraumes, in dem sich die zufällig angeordneten Objekte befinden, dar. Diese Position kann je nach Anwendung individuell gewählt werden und hängt vom verwendeten Algorithmus, dem Datensatz zum Einlernen des Algorithmus, der Auflösung der Kamera und der Anzahl der Objekte ab.

3: 2D-Lokalisierung

Dieser Schritt ist notwendig, damit das System die Anzahl und Klasse der

²ausführlich behandelt in [3, Steinbeck, Abschnitt 4.2]

4 Gesamtsystem

verwendeten Teile sowie eine ungefähre Position aller Objekte erfassen kann. Der in **Abschnitt 3.1.1** erwähnte Distanzbereich ist hier irrelevant, da nur die Position in horizontaler Ebene benötigt wird.

4: Teilmittelpunktposition

³Hier wird mithilfe des in Schritt 3 erkannten Mittelpunktes des ersten Objekts eine genauere Mittelpunktposition ausfindig gemacht.

5: 3D-Lokalisierung

Für die Lokalisierung wird das in **Abschnitt 3.2.3** beschriebene Paket verwendet. Sobald sich die Kamera näherungsweise über dem Objektmittelpunkt befindet, wird der Abstand zwischen Kamera und dem durch eine *Point Cloud* erkannten höchsten Punkt des Objekts ermittelt. Dieser Punkt wird in Form einer Liste aus vier Variablen mit Relativposition zur Kamera und Objektklasse zurückgegeben.

6: Zielposition

³Die im Bauplan festgelegte Zielposition des Teils wird angefahren.

7: Endposition

Nach Ablegen aller Teile kann der Roboter in eine beliebige, nicht singuläre Pose fahren.

Die Schritte **4** bis **6** werden für jedes Teil wiederholt.

³ausführlich behandelt in [3, Steinbeck, Abschnitt 4.2]

4 Gesamtsystem

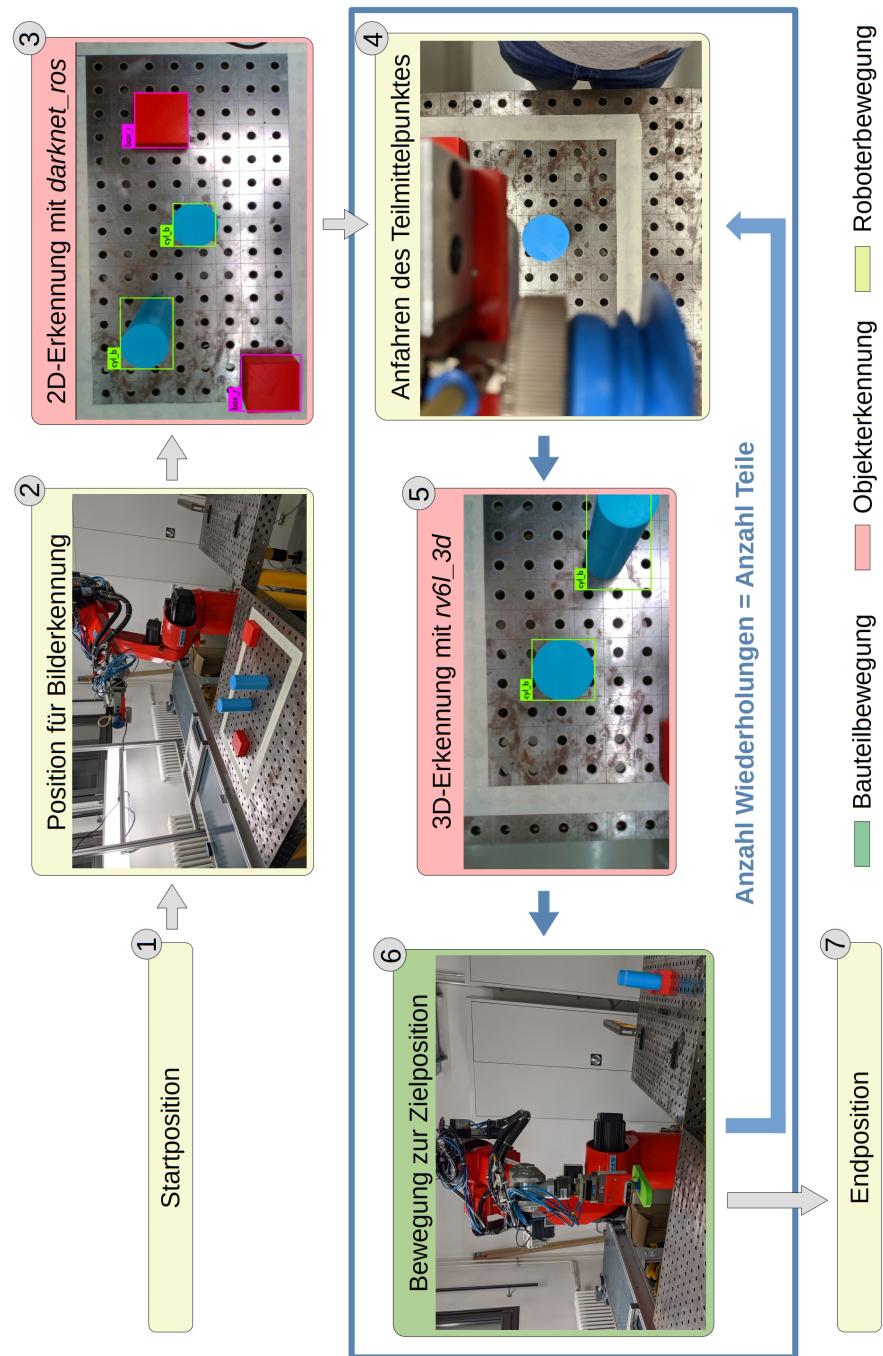


Abbildung 4.2: Ablauf des Gesamtprogramms

5 Evaluierung und Weiterentwicklung

Folgendes Kapitel beinhaltet eine Evaluierung der verwendeten Methoden, Probleme und mögliche Optimierungsansätze in Bezug zur Objekterkennung. Verbesserungen der Handhabung sind in der Arbeit von Steinbeck aufgeführt [3, Abschnitt 5].

5.1 Auswertung und Optimierung

Die Verwendung von *DL* mit dem *YOLOv3 Tiny*-Algorithmus ist aufgrund der für ein *CNN* sehr schnellen Klassifizierung und Positionserkennung auch mit schwacher Hardware und der ausreichenden Genauigkeit gut für die Anwendung geeignet. Auch die sehr geringe Fehlerrate trotz des kleinen Datensatzes zum Einlernen mit *Darknet* spricht für *DL* und gegen die mit höherem Aufwand verbundenen Verwendung traditioneller *CV*. *YOLO* kann außerdem mit dem Paket *darknet_ros*, für das schon Versionen für die Verwendung von ROS-II verfügbar sind, gut in ROS integriert werden. Aufgrund dieser Verfügbarkeit und der aktiven Entwicklung ist eine zukünftige Weiterentwicklung des *rv6l_3d*-Pakets gut umsetzbar. Wegen der Verfügbarkeit von *Packages* wie *darknet_ros_3d* ist die Verwendung der ROS-Version *Noetic* aktuell für die Anwendung geeignet. Mit der Nutzung der Intel RealSense als Alternative zur Microsoft Kinect kann aufgrund der Möglichkeit einer Befestigung am Endeffektor bei guter Ausrichtung dieser eine hohe Genauigkeit durch Ausgleichen der Positionsabweichung des verwendeten Roboters erzielt werden. Die Objekte können wegen der guten Kalibrierung ab Werk und des integrierten Infrarotprojektors sehr genau lokalisiert werden. Auch die Treiberunterstützung bietet mit Bereitstellung der Daten in *OpenNI* kompatilem Format Vorteile gegenüber anderen 3D-Kamerasystemen. Durch die zweistufige Mittelpunkterkennung kann der Zielpunkt sehr genau angefahren werden. Die Abweichung in *z*-Richtung ist hier minimal. Wegen Toleranzen bei der Befestigung der Kamera am Endeffektor können beispielsweise aufgrund einer Neigung Abweichungen in der *x-y*-Ebene entstehen. In den durchgeführten Tests waren diese allerdings gering und führten somit nicht zu Problemen. Die Kommunikation zwischen allen verwendeten *Packages* funktioniert auch bei Nutzung verschiedener Programmiersprachen problemlos.

5 Evaluierung und Weiterentwicklung

Auch wenn die Erkennungsrate durch den YOLO-Algorithmus bei den verwendeten Objekten sehr hoch ist, können vor allem bei ähnlichen Teilen Fehler bei der Klassifizierung auftreten. Mit der Verwendung eines größeren Datensatzes, einer Erhöhung von Auflösung oder Iterationen, oder Optimierungen des bestehenden Datensatzes kann die Erkennungsrate und Positionsgenauigkeit gesteigert werden. Die um 90° gegenüber dem Endeffektor rotierte Montage ist aufgrund der Nähe zu einer Singularität des RV6L durch einen gestreckten Arm, bei dem die Achsen 4 und 6 deckungsgleich sind, in der maximalen Höhe limitiert. Eine Montage mit Blickrichtung parallel zum Kontaktpunkt des verwendeten Endeffektors würde diese Problematik lösen, jedoch den für die Kamera erreichbaren Bereich in x -Richtung des Basiskoordinatensystems verkleinern. Zur Optimierung der Genauigkeit in der x - y -Ebene kann eine Hand-Augen-Kalibrierung durchgeführt werden. Die Genauigkeit in z -Richtung wird auch hier nur wenig durch den Kamerawinkel beeinflusst.

5.2 Ansätze zur Weiterentwicklung des rv6l_3d-Pakets

Während die Grundfunktionalität zur Objekterkennung und Positionserkennung gegeben ist, ist eine Weiterentwicklung des *rv6l_3d* Pakets notwendig. Aktuell funktioniert die Schnittstelle zwischen den Objekterkennungs- und Handhabungspaketen nur teilweise, da der *Bounding Box Subscriber* (s. **Anhang C**) trotz Erkennung aller Bauteile nur die Position des zuletzt erkannten Blocks an das Programm zur Handhabung weitergibt. Somit wird eine Logik zur gleichzeitigen Weitergabe aller ungefähren Positionen der ersten Scanposition benötigt. Außerdem ist für die zweite Position eine Logik notwendig, die bei mehreren Teilen im Erkennungsbereich das mittlere Objekt identifiziert und nur dessen Position an den *Subscriber* weitergibt. Somit ist die Erkennung und Aufnahme des korrekten Objekts gewährleistet. Zur Optimierung des Systems können außerdem folgende Ansätze genutzt werden: Aktuell ist der Erkennungsbereich durch den maximalen Aufnahmebereich des Kamerasytems limitiert, da der erste Scavorgang des kompletten Bereichs nur mit fester Kameraposition durchgeführt wird. Dieses Problem kann durch die Implementierung einer Bewegung parallel zur x - y -Ebene des Basiskoordinatensystems während des Scans und anschließender Umrechnung der Objektpositionen in vom Handhabungsprogramm nutzbare Werte gelöst werden, wobei jedoch die durch die Perspektive abweichende Mittelpunktposition der Objekte beachtet werden muss. Auch eine Erkennung aus unterschiedlichen Kameraorientierungen oder ein von der Objektgröße abhängiger Abstand zwischen Kamera und Bauteil ist denkbar. Bei entsprechender Leistung des zur Bilderkennung verwendeten Computers ist ein Anfahren des Objekts mit Abgleich zwischen Kamera- und

5 Evaluierung und Weiterentwicklung

Objektmittelpunkt in Echtzeit zur Steigerung der Genauigkeit in x - y -Ebene möglich. Auch wenn der Mittelpunkt bei den verwendeten Objekten eine sinnvolle Greifposition darstellt, ist dieser nicht in allen Anwendungsfällen optimal. Eine Alternative ist etwa die Erkennung des höchsten Punktes eines Objekts oder einer geraden Fläche. Zur Verringerung der Fehleranfälligkeit kann zudem die Ablagehöhe für das Bauteil in der Zielposition gemessen werden, um Kollisionen mit Objekten im Ablagebereich zu vermeiden. Ein weiterer wichtiger Aspekt ist die Implementierung von Funktionen zur Ausnahmebehandlung. Wird beispielsweise ein Block nicht erkannt, kommt es beim Bewegungsprogramm in der aktuellen Form zu einem Fehler. Ein möglicher Lösungsansatz ist hier die Nutzung der *3D-Point Cloud*, um durch Erhöhungen im Bauraum Objekte zu erkennen und einen erneuten Scavorgang zu starten. Auch die Erkennung von mehr Teilen, als im Bauplan genutzt werden, kann zu unvorhersehbaren Ereignissen beim Ausführen des Paktes führen.

Ein Nachteil des *YOLO*-Algorithmus ist die fehlende Unterstützung für *Orientation Bounding Boxes* (*OBBs*). Stattdessen werden *Horizontal Bounding Boxes* (*HBBs*) verwendet, die immer aus horizontalen und vertikalen Kanten ohne eine Möglichkeit zur Orientierungserkennung bestehen. Somit muss für eine Erkennung der Bauteilarbeitung ein anderer Algorithmus genutzt werden. Dazu kann beispielsweise der Ansatz von Yin et al. zur Kalkulation von *OBBs* basierend auf *YOLO* genutzt werden (s. **Abbildung 5.1**) [60].

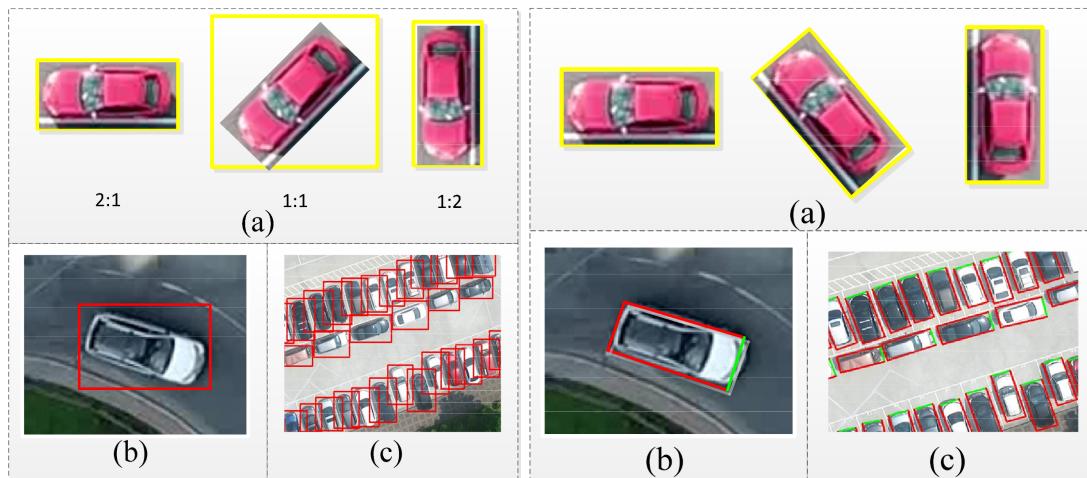


Abbildung 5.1: Vergleich zwischen HBBs (links) und OBBs (rechts) [60]

Eine Implementierung von *YOLOv5* mit *OBBs*, die ein Paket für die Nutzung mit ROS beinhaltet, wurde im Dezember 2021 unter dem Namen "RoSA: Cube Detector" veröffentlicht [61].

Eine weitere Optimierung besteht in der Kollisionsvermeidung mit erkannten Objekten. Der genutzte Bahnplanungsalgorithmus umfährt bekannte Objekte im Bauraum. Jedoch werden die erkannten Bauteile diesem nicht zur Verfü-

5 Evaluierung und Weiterentwicklung

gung gestellt. Die Bereitstellung der Kollisionsobjekte kann entweder über Einfügen der bekannten Objekte an den durch die Objekterkennung ermittelten Mittelpunktpositionen in die Simulation oder der Nutzung der von der Kamera generierten *Point Cloud* umgesetzt werden.

6 Fazit

Die Nutzung von 3D-Kamerasystemen und CNNs in Verbindung mit Robotern schafft eine gute Grundlage für die Handhabung verschiedener Teile. An den Ergebnissen werden der Fortschritt und die Möglichkeiten aktueller neuronaler Netze wie YOLO, die auch mit vergleichsweise schwacher Hardware geboten werden, deutlich. Durch die Nutzung des ROS ist eine standardisierte Schnittstelle für die Kommunikation verschiedener Pakete verfügbar. Hier erschließen sich die Vorteile der Open-Source-Software besonders. Die Verwendung bestehender Pakete als Grundlage für die Implementierung der Objekterkennung unter Nutzung des Meta-Betriebssystems schafft eine gute Basis für die Umsetzung der Bilderkennung.

Das Ziel der Erkennung von Klasse und Position einer ungeordneten Zusammenstellung von Objekten wurde erreicht. Das auf *darknet_ros_3d* basierende *rv6l_3d* kann mithilfe des ROS und eines 3D-Kamerasystems zuverlässig einzelne Bauteile im Aufnahmebereich erkennen und Positionsdaten und Art an den *Subscriber* des Handhabungsprogramms weitergeben. Die Intel RealSense 435 eignet sich durch die kleine Baugröße, genaue Abstandsberechnung und für den Anwendungsfall ausreichende Auflösung der Kameras gut für die durch *rv6l_3d* umgesetzte Positionserkennung. Ein Nachteil der genutzten Konfiguration ist die Nähe zu einer singulären Stellung in der Scanposition. Eine Befestigung des Kamerasystems parallel zum Endeffektor kann diesen lösen, was allerdings eine Verringerung des Erkennungsbereichs zur Folge hat. Die Nutzung von *DL* zur Erkennung der Objekte bietet im Vergleich zu traditioneller *CV* viele Vorteile, wie die hohe Erkennungsrate oder den vergleichsweise wenig komplexen Prozess zur Erstellung eigener Datensätze. So eignet sich YOLO in den Grundzügen wegen der für ein CNN guten Performance und kontinuierlich zuverlässigen Erkennung der verwendeten Objekte sehr gut für den Anwendungsfall.

Auch wenn eine Grundfunktionalität gegeben ist, muss das *rv6l_3d*-Paket im Hinblick auf die in **Abschnitt 5** beschriebenen Probleme optimiert werden. Neben notwendigen Verbesserungen wie der Weitergabe mehrerer Bauteile durch die Schnittstelle zwischen Objekterkennungs- und Handhabungsprogramm und der Ausnahmebehandlung können die erwähnten Vorschläge wie die Positionserkennung zu einer Erweiterung des Anwendungsbereichs genutzt werden.

6 Fazit

Die Ergebnisse zeigen die durch ROS gebotenen Möglichkeiten zur Kombination verschiedener Hardware und Software. Dies wird besonders in der Übersicht des Gesamtsystems (s. **Abbildung 4.1**) deutlich. Durch diese Eigenschaft bietet das System eine Basis, die für zukünftige Weiterentwicklungen genutzt werden kann.

Abbildungsverzeichnis

2.1	ROS Struktur [13]	7
2.2	Aufbau RGB-D Kamera [21]	10
2.3	Traditioneller Ansatz [26, Abbildung 3]	12
2.4	Vereinfachte Struktur von YOLO [32]	13
3.1	Aufbau der Microsoft Kinect v2 [39, Abbildung 5]	16
3.2	Genauigkeit der Microsoft Kinect v2 [41, Abbildung 5]	16
3.3	Aufbau der Intel RealSense 435, Abbildung nach [44]	17
3.4	Infrarotproj. und 3D-Bild der Intel RealSense 435	17
3.5	Genauigkeit der Intel RealSense 435	18
3.6	RealSense Point Cloud in Rviz	19
3.7	Befestigung der Intel RealSense am Endeffektor	21
3.8	Anzahl der ROS-Downloads im März 2020 und 2021, Grafik nach [54]	22
3.9	YOLO Label	23
3.10	Darknet Iterationen [generiert mit Darknet]	24
3.11	rv6l_3d Bounding Boxes	25
3.12	Koordinatensystem der Intel RealSense, Abbildung nach [44]	27
3.13	Koordinatensystem des RV6L, Abbildung nach [59, Abbildung 12]	28
3.14	Objektkoordinaten	29
3.15	Roboterpositionen	31
4.1	Struktur des Gesamtsystems	34
4.2	Ablauf des Gesamtprogramms	37
5.1	Vergleich zwischen HBBs (links) und OBBs (rechts) [60]	40

Literatur

- [1] G. Diesing. (1. Sep. 2021). How AI and machine vision impact vision robotics, Quality Magazine, Adresse: <https://www.qualitymag.com/articles/96664-how-ai-and-machine-vision-impact-vision-robotics> (besucht am 27. 02. 2022).
- [2] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler und A. Ng, „ROS: An open-source robot operating system,” Nr. 3, S. 6, 12. Mai 2009.
- [3] D. Steinbeck, *Entwicklung einer KI-basierten Robotersteuerung mit Industrial ROS - Generierung von Handhabungssequenzen für mechanische Konstruktionen nach gegebenen Bauplänen*, 27. Feb. 2022.
- [4] ros.org. (2021). ROS/Home, Adresse: <https://www.ros.org/> (besucht am 11. 12. 2021).
- [5] L. Joseph und J. Cacace, *Mastering ROS for robotics programming: design, build, and simulate complex robots using the Robot Operating System*, Second edition. Birmingham Mumbai: Packt, 2018, 559 S., ISBN: 978-1-78847-895-3.
- [6] rosindustrial.org. (o. D.). Brief History, Adresse: <https://rosindustrial.org/briefhistory> (besucht am 09. 12. 2021).
- [7] A. Dattalo. (8. Aug. 2018). ROS/Introduction - ROS Wiki, Adresse: <https://wiki.ros.org/ROS/Introduction> (besucht am 11. 12. 2021).
- [8] wiki.ros.org. (31. Dez. 2020). noetic/Installation - ROS Wiki, Adresse: <https://wiki.ros.org/noetic/Installation> (besucht am 11. 12. 2021).
- [9] A. M. Romero. (21. Juni 2014). ROS/Concepts - ROS Wiki, Adresse: <https://wiki.ros.org/ROS/Concepts> (besucht am 12. 12. 2021).
- [10] D. Thomas. (7. Juli 2017). catkin/workspaces - ROS Wiki, Adresse: <http://wiki.ros.org/catkin/workspaces> (besucht am 14. 12. 2021).
- [11] F. Van Eeden. (26. März 2020). catkin/conceptual_overview - ROS Wiki, Adresse: http://wiki.ros.org/catkin/conceptual_overview (besucht am 14. 12. 2021).
- [12] S. Loretz. (20. Mai 2020). rosdep - ROS Wiki, Adresse: <https://wiki.ros.org/rosdep> (besucht am 14. 12. 2021).

Literatur

- [13] Clearpath Robotics. (2015). Intro to ROS – ROS Tutorials 0.5.2 documentation, Adresse:
<https://www.clearpathrobotics.com/assets/guides/kinetic/ros/Intro%20to%20the%20Robot%20operating%20System.html> (besucht am 19.12.2021).
- [14] G. Staples. (6. Nov. 2020). Support - ROS Wiki, Adresse:
<https://wiki.ros.org/Support> (besucht am 13.12.2021).
- [15] Y. Miura. (15. Mai 2021). Distributions - ROS Wiki, Adresse:
<https://wiki.ros.org/Distributions> (besucht am 13.12.2021).
- [16] B. Murray. (21. Okt. 2021). Releases - Ubuntu Wiki, Adresse:
<https://wiki.ubuntu.com/Releases> (besucht am 13.12.2021).
- [17] Open Robotics. (2021). Distributions – ROS 2 Documentation: Foxy documentation, Adresse:
<https://docs.ros.org/en/foxy/Releases.html> (besucht am 13.12.2021).
- [18] Sir David Brewster. (1856). The stereoscope : Its history, theory, and construction, with its application to the fine and useful arts and to education / by sir david brewster., Wellcome Collection, Adresse:
<https://wellcomecollection.org/works/fehj4h6n> (besucht am 10.02.2022).
- [19] F. Waack, Stereofotografie: *Einführung in der Fototechnik und praktische Ratschläge für der Aufnahme*. Selbsverl., 1979. Adresse:
<https://books.google.de/books?id=TK5fPgAACAAJ>.
- [20] Prof. Dr.-Ing. Günter Pomaska. (2013). Tiefenkameras, Adresse:
<https://www.scanner.imagefact.de/de/depthcam.html> (besucht am 23.12.2021).
- [21] P. D.-I. G. Pomaska. (2013). primeSensor.png (PNG Image, 380 × 312 pixels), Adresse:
<https://www.scanner.imagefact.de/img/primeSensor.png> (besucht am 23.12.2021).
- [22] S. Mohan. (30. Juni 2020). 6 Different Types of Object Detection Algorithms in Nutshell, Adresse:
<https://machinelearningknowledge.ai/different-types-of-object-detection-algorithms/> (besucht am 19.12.2021).
- [23] D. G. Lowe, „Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision*, Jg. 60, Nr. 2, S. 91–110, 1. Nov. 2004, Number: 2, ISSN: 1573-1405. doi:
10.1023/B:VISI.0000029664.99615.94. Adresse:
[https://doi.org/10.1023/B:VISI.0000029664.99615.94.](https://doi.org/10.1023/B:VISI.0000029664.99615.94)

Literatur

- [24] H. Bay, T. Tuytelaars und L. Van Gool, „SURF: Speeded Up Robust Features,” in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof und A. Pinz, Hrsg., Veranstaltungsort: Berlin, Heidelberg, Springer Berlin Heidelberg, 2006, S. 404–417, ISBN: 978-3-540-33833-8.
- [25] C. Cortes und V. Vapnik, „Support-vector networks,” *Machine Learning*, Jg. 20, Nr. 3, S. 273–297, Sep. 1995, ISSN: 0885-6125, 1573-0565. doi: 10.1007/BF00994018. Adresse: <http://link.springer.com/10.1007/BF00994018> (besucht am 19.12.2021).
- [26] X. Youzi, Z. Tian, J. Yu, Y. Zhang, S. Liu, S. Du und X. Lan, „A review of object detection based on deep learning,” *Multimedia Tools and Applications*, Jg. 79, 1. Sep. 2020. doi: 10.1007/s11042-020-08976-6.
- [27] N. O’Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan und J. Walsh, „Deep learning vs. traditional computer vision,” in *Advances in Computer Vision*, K. Arai und S. Kapoor, Hrsg., Bd. 943, Titel der Serie: Advances in Intelligent Systems and Computing, Cham: Springer International Publishing, 2020, S. 128–144, ISBN: 978-3-030-17794-2 978-3-030-17795-9. doi: 10.1007/978-3-030-17795-9_10. Adresse: http://link.springer.com/10.1007/978-3-030-17795-9_10 (besucht am 04.02.2022).
- [28] M. Labb  . (2011). Find Object, Adresse: <http://introlab.github.io/find-object> (besucht am 15.12.2021).
- [29] G. Bonacorso, *Machine Learning Algorithms*, 2. Aufl., Ser. Machine Learning Algorithms. Packt Publishing, 2018, ISBN: 978-1-78934-799-9. Adresse: <https://www.packtpub.com/product/machine-learning-algorithms-second-edition/9781789347999> (besucht am 04.02.2022).
- [30] N. O’ Mahony, T. Murphy, K. Panduru, D. Riordan und J. Walsh, „Adaptive process control and sensor fusion for process analytical technology,” in *2016 27th Irish Signals and Systems Conference (ISSC)*, Juni 2016, S. 1–6. doi: 10.1109/ISSC.2016.7528449.
- [31] E. Yurtsever, J. Lambert, A. Carballo und K. Takeda, „A Survey of Autonomous Driving: Common Practices and Emerging Technologies,” *IEEE Access*, Jg. 8, S. 58 443–58 469, 22. M  rz 2020, ISSN: 2169-3536. doi: 10.1109/ACCESS.2020.2983149. arXiv: 1906.05113. Adresse: <http://arxiv.org/abs/1906.05113> (besucht am 15.12.2021).
- [32] J. Redmon, S. Divvala, R. Girshick und A. Farhadi, „You Only Look Once: Unified, Real-Time Object Detection,” *arXiv:1506.02640 [cs]*, 9. Mai 2016. arXiv: 1506.02640. Adresse: <http://arxiv.org/abs/1506.02640> (besucht am 20.12.2021).

Literatur

- [33] H. Bandyopadhyay. (9. Sep. 2021). YOLO: Real-Time Object Detection Explained, Adresse:
<https://www.v7labs.com/blog/yolo-object-detection> (besucht am 11.01.2022).
- [34] Z. Zou, Z. Shi, Y. Guo und J. Ye, „Object Detection in 20 Years: A Survey,” *arXiv:1905.05055 [cs]*, 15. Mai 2019. arXiv: 1905.05055. Adresse:
<http://arxiv.org/abs/1905.05055> (besucht am 15.12.2021).
- [35] J. Redmon, *Darknet*, 20. Dez. 2021. Adresse:
<https://github.com/pjreddie/darknet> (besucht am 20.12.2021).
- [36] Joseph Redmon. (20. Feb. 2020). I stopped doing CV research, Adresse:
<https://twitter.com/pjreddie/status/1230524770350817280> (besucht am 20.12.2021).
- [37] A. Bochkovskiy, C.-Y. Wang und H.-Y. M. Liao, „YOLOv4: Optimal Speed and Accuracy of Object Detection,” *arXiv:2004.10934 [cs, eess]*, 22. Apr. 2020. arXiv: 2004.10934. Adresse: <http://arxiv.org/abs/2004.10934> (besucht am 20.12.2021).
- [38] M. Bjelonic, *YOLO ROS: Real-Time Object Detection for ROS*, 2016. Adresse: https://github.com/leggedrobotics/darknet_ros.
- [39] N.-J. Sung, Ma, C.-S. Choi und G.-R. Hong, „Real-Time Augmented Reality Physics Simulator for Education,” *Applied Sciences*, Jg. 9, S. 4019, 25. Sep. 2019. DOI: 10.3390/app9194019. Adresse:
https://www.researchgate.net/figure/Microsoft-Kinect-V2s-Hardware-Structure-Information-Resolution-Frame-per-Second-FPS_fig5_336061295.
- [40] P. J. Noonan, J. Howard, W. A. Hallett und R. N. Gunn, „Repurposing the microsoft kinect for windows v2 for external head motion tracking for brain PET,” *Physics in Medicine and Biology*, Jg. 60, Nr. 22, S. 8753–8766, 21. Nov. 2015, ISSN: 0031-9155, 1361-6560. DOI: 10.1088/0031-9155/60/22/8753. Adresse: <https://iopscience.iop.org/article/10.1088/0031-9155/60/22/8753> (besucht am 15.02.2022).
- [41] O. Wasenmüller und D. Stricker, „Comparison of kinect v1 and v2 depth images in terms of accuracy and precision,” in *Computer Vision – ACCV 2016 Workshops*, C.-S. Chen, J. Lu und K.-K. Ma, Hrsg., Bd. 10117, Titel der Serie: *Lecture Notes in Computer Science*, Cham: Springer International Publishing, 2017, S. 34–45, ISBN: 978-3-319-54426-7 978-3-319-54427-4. DOI: 10.1007/978-3-319-54427-4_3. Adresse:
http://link.springer.com/10.1007/978-3-319-54427-4_3 (besucht am 15.02.2022).

Literatur

- [42] L. Xiang, F. Echtler, C. Kerl, T. Wiedemeyer, Lars, hanyazou, R. Gordon, F. Facioni, laborer2008, R. Wareham, M. Goldhoorn, alberth, gaborpapp, S. Fuchs, jmtatsch, J. Blake, Federico, H. Jungkurth, Y. Mingze, vinouz, D. Coleman, B. Burns, R. Rawat, S. Mokhov, P. Reynolds, P. E. Viau, M. Fraissinet-Tachet, Ludique, J. Billingham und Alistair, *libfreenect2: Release 0.2*, 28. Apr. 2016. doi: 10.5281/zenodo.50641. Adresse: <https://zenodo.org/record/50641> (besucht am 08.02.2022).
- [43] T. Wiedemeyer, *IAI Kinect2*, 16. Dez. 2021. Adresse: https://github.com/code-iai/iai_kinect2/blob/0e2c5f63134a076606bb79963406e1d47f2da651/kinect2_calibration/README.md (besucht am 20.12.2021).
- [44] Intel Corporation. (2022). Depth camera d435, Intel® RealSense™ Depth and Tracking Cameras, Adresse: <https://www.intelrealsense.com/depth-camera-d435/> (besucht am 14.02.2022).
- [45] Intel Corporation, *Intel/RealSense/librealsense*, Erstveröffentlichung: 17.11.2015, 8. Feb. 2022. Adresse: <https://github.com/IntelRealSense/librealsense> (besucht am 08.02.2022).
- [46] Intel Corporation, *ROS Wrapper for Intel® RealSense™ Devices*, Erstveröffentlichung: 23.02.2016, 7. Feb. 2022. Adresse: <https://github.com/IntelRealSense/realsense-ros> (besucht am 08.02.2022).
- [47] Occipital. (2022). OpenNI programmers guide, OpenNI 2 SDK Binaries & Docs, Adresse: https://s3.amazonaws.com/com.occipital.openni/OpenNI_Programmers_Guide.pdf (besucht am 08.02.2022).
- [48] T. Wiedemeyer, *iai_kinect2/kinect2_bridge/data*, 18. Mai 2015. Adresse: https://github.com/code-iai/iai_kinect2 (besucht am 20.12.2021).
- [49] opencv.org. (31. Dez. 2019). Camera calibration and 3d reconstruction – OpenCV 2.4.13.7 documentation, Adresse: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html (besucht am 27.12.2021).
- [50] A. Grunnet-Jepsen, J. Sweetser, T. Khuong, D. Tong und O. Mulla. (2021). Intel® RealSense™ self-calibration for d400 series depth cameras, Intel® RealSense™ Developer Documentation. Revision 2.1, Adresse: <https://dev.intelrealsense.com/docs/self-calibration-for-depth-cameras> (besucht am 15.02.2022).

Literatur

- [51] A. Tabb und K. M. A. Yousef, „Solving the robot-world hand-eye(s) calibration problem with iterative methods,” *Machine Vision and Applications*, Jg. 28, Nr. 5, S. 569–590, Aug. 2017, ISSN: 0932-8092, 1432-1769. DOI: 10.1007/s00138-017-0841-7. arXiv: 1907.12425. Adresse: <http://arxiv.org/abs/1907.12425> (besucht am 15. 02. 2022).
- [52] C. Rauch, R. Haschke, T. Weaver, RoboticsYY, D. Coleman, V. Magnago, J. Stechschulte, S. Brahmbhatt, Jakubach, M. Ferguson und I. Brijacak, *MoveIt Calibration*, Erstveröffentlichung: 02.06.2020, 11. Feb. 2022. Adresse: https://github.com/ros-planning/moveit_calibration (besucht am 15. 02. 2022).
- [53] J. Stechschulte. (2022). Hand-Eye Calibration – moveit_tutorials Noetic documentation, Adresse: https://ros-planning.github.io/moveit_tutorials/doc/hand_eye_calibration/hand_eye_calibration_tutorial.html (besucht am 15. 02. 2022).
- [54] K. Scott. (14. Mai 2021). ROS News for the Week of 5/10/2021 - General, Adresse: <https://discourse.ros.org/t/ros-news-for-the-week-of-5-10-2021/20399> (besucht am 17. 01. 2022).
- [55] Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, Jakub Steiner, David Gossow, Chad Rockey, Kei Okada, Julius Kammerl, Acorn Pooley, Rein Appeldoorn und Robert Haschke, *ros-visualization/rviz*, Erstveröffentlichung: 18.09.2012, 14. Feb. 2022. Adresse: <https://github.com/ros-visualization/rviz> (besucht am 15. 02. 2022).
- [56] Ioan A. Sucan und Sachin Chitta. (o. D.). MoveIt motion planning framework, MoveIt, Adresse: <https://moveit.ros.org/> (besucht am 15. 02. 2022).
- [57] Tzutalin, *LabelImg*, 2015. Adresse: <https://github.com/tzutalin/labelImg> (besucht am 04. 01. 2022).
- [58] IntelligentRoboticsLabs, *darknet_ros_3d*, 16. Dez. 2021. Adresse: https://github.com/IntelligentRoboticsLabs/gb_visual_detection_3d (besucht am 20. 12. 2021).
- [59] M. Leber, *Steuerung eines Roboterarms mit Industrial ROS und KI-basierte Handhabung von ungeordnet zugeführten Teilen*, 30. Sep. 2021. (besucht am 01. 02. 2022).

Literatur

- [60] J. Yin, H. Pan, H. Su, Z. Liu und Z. Peng, „A Fast Orientation Invariant Detector Based on the One-stage Method,” *MATEC Web of Conferences*, Jg. 232, S. 7, 2018. DOI: 10.1051/matecconf/201823204036. Adresse: https://www.researchgate.net/figure/Detection-results-of-OIYOL0-using-OBB-and-YOLO-using-HBB-on-UAV-DAHUA-dataset-The-first_fig2_329038189.
- [61] T. Hempel, *RoSA: Cube Detector*, Version 1.0.0, 17. Dez. 2021. DOI: 10.5281/zenodo.5788773. Adresse: <https://orcid.org/0000-0002-3621-7194>.
- [62] The AI Guy, *YoloGenerateTrainingFile*, Erstveröffentlichung: 08.01.2020, 17. Dez. 2021. Adresse: https://github.com/theAIGuysCode/YoloGenerateTrainingFile/blob/2bcb9b1ba7e75061ce1c267a48860c67cb229b62/generate_train.py (besucht am 14. 02. 2022).

Anhang

Anhang

A	RV6L 3D	XV
A.1	Konfiguration	XV
A.2	Bounding Box Funktion	XVI
A.3	Bounding Box Message	XVII
A.4	Launchfiles	XVIII
A.5	ROS Package	XX
B	Realsense Launchfile	XXIII
C	Handhabung	XXVIII
C.1	Bounding Box Subscriber	XXVIII
C.2	Anfahren des Objekts	XXVIII
D	Darknet	XXXIII
D.1	Darknet Konfiguration	XXXIII
D.2	Objekt Konfiguration	XXXIII
D.3	Generator für train.txt (Python)	XXXIII

Auf dem beigelegten Datenträger (Verzeichnis David Gries/Dokumente und Medien) befindet sich eine HTML-Datei mit Anleitungen zur Installation der ROS-Pakete. Außerdem ist der vollständige Quellcode aller relevanten Pakete und eine Demonstration des Projekts in Form eines Videos enthalten. Entfernte Zeilen zur Verbesserung der Übersicht sind mit [. . .] gekennzeichnet. Die Verzeichnisstruktur von Quellcode und Konfigurationsdateien richtet sich dabei nach der Struktur des folgenden schriftlichen Anhangs. Dieser beinhaltet alle Code-Abschnitte, die zum besseren Verständnis der Arbeit beitragen.

Informationen zu Lizenzen der in dieser Arbeit verwendeten Projekte sind auf dem Datenträger zu finden. Diese befinden je nach Anforderungen der jeweiligen Lizenz im Ordner des Softwarepaketes oder am Anfang des Quellcodes. Kennzeichnungspflichtige Änderungen wurden an den entsprechenden Stellen ergänzt.

Anhang

```
David Gries
  └── Anhang A - RV6L 3D
    ├── A_1 - Konfiguration
    │   ├── RV6L.yaml
    │   ├── ros.yaml
    │   └── darknet_3d.yaml
    ├── A_2 - Bounding Box Funktion
    │   ├── Darknet3DListener.cpp
    │   ├── Darknet3D.cpp
    │   └── darknet3d_node.cpp
    ├── A_3 - Bounding Box Message
    │   ├── RV6L_positions.msg
    │   └── RV6L_position.msg
    ├── A_4 - Launchfiles
    │   ├── first.launch
    │   ├── RV6L.launch
    │   └── second.launch
    └── A_5 - ROS Package
        └── package.xml
  └── Anhang B - RealSense Launchfile
    └── RV6L.launch
  └── Anhang C - Handhabung
    ├── C_1 - Bounding Box Subscriber
    │   └── subscriber_cam.py
    └── C_2 - Anfahren des Objekts
        └── pick_and_place_detection.py
  └── Anhang D - Darknet
    ├── D_1 - Darknet Konfiguration
    │   └── RV6L.cfg
    ├── D_2 - Objekt Konfiguration
    │   └── obj.data
    └── D_3 - Generator fuer train
        └── generate_train.py
  └── Dokumente und Medien
    ├── Bachelorarbeit David Gries.pdf
    ├── Bilder und Quelldateien/...
    ├── Installationsanleitung.html
    └── Objekterkennung Demo.mp4
```

A RV6L 3D

A.1 Konfiguration

A.1.1 Hauptkonfiguration

[darknet_3d.yaml]

```
1 darknet_ros_topic: /darknet_ros/bounding_boxes
2 output_bbx3d_topic: /darknet_ros_3d/bounding_boxes
3 point_cloud_topic: /camera/depth_registered/points
4 working_frame: camera_link
5 minimum_detection_therehold: 0.3
6 minimum_probability: 0.3
7 interested_classes: ["box_r", "box_g", "cyl_b"]
```

A.1.2 Darknet ROS Konfiguration

[ros.yaml]

```
1 subscribers:
2
3   camera_reading:
4     topic: /camera/color/image_raw
5     queue_size: 1
6 [ . . . ]
```

A.1.3 RV6L Konfiguration

[rv6l.yaml]

```
1 yolo_model:
2
3   config_file:
4     name: RV6L.cfg
5   weight_file:
6     name: RV6L.weights
7   threshold:
```

Anhang

```
8     value: 0.3
9     detection_classes:
10    names:
11      - box_r
12      - box_g
13      - cyl_b
```

A.2 Bounding Box Funktion

[darknet3d_node.cpp, Darknet3D.cpp, Darknet3DListener.cpp]

```
1 [ . . . ]
2
3 void
4 Darknet3D::calculate_boxes(const sensor_msgs::PointCloud2&
5     cloud_pc2,
6     const pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr
7     cloud_pcl,
8     rv6l_3d_msgs::RV6L_positions* boxes) // Funktion zur
9     Berechnung der Positionsdaten
10
11    for (auto bbx : original_bboxes_)
12    {
13        if ((bbx.probability < minimum_probability_) ||
14            (std::find(interested_classes_.begin(),
15                      interested_classes_.end(), bbx.Class) ==
16                      interested_classes_.end()))
17        {
18            continue;
19        }
20
21        int center_x, center_y;
22        center_x = (bbx.xmax + bbx.xmin) / 2;
23        center_y = (bbx.ymax + bbx.ymin) / 2;
24
25        int pcl_index = (center_y * cloud_pc2.width) + center_x;
26        pcl::PointXYZRGB center_point = cloud_pcl->at(pcl_index);
27
28        if (std::isnan(center_point.x))
29            continue;
30
31        float maxx, minx, maxy, miny, maxz, minz, midx, midy, midz
32            , w, h, d;
```

Anhang

```
32     maxx = maxy = maxz = -std::numeric_limits<float>::max();
33     minx = miny = minz = std::numeric_limits<float>::max();
34
35     for (int i = bbx.xmin; i < bbx.xmax; i++)
36         for (int j = bbx.ymin; j < bbx.ymax; j++)
37     {
38         pcl_index = (j* cloud_pc2.width) + i;
39         pcl::PointXYZRGB point = cloud_pcl->at(pcl_index);
40
41         if (std::isnan(point.x))
42             continue;
43
44         if (fabs(point.x - center_point.x) >
45             mininum_detection_thereshold_)
46             continue;
47
48         maxx = std::max(point.x, maxx);
49         maxy = std::max(point.y, maxy);
50         maxz = std::max(point.z, maxz);
51         minx = std::min(point.x, minx);
52         miny = std::min(point.y, miny);
53         minz = std::min(point.z, minz);
54
55         midx = (maxx + minx) / 2; // Objektmittelpunkt (x)
56         midy = (maxy + miny) / 2; // Objektmittelpunkt (y)
57         midz = (maxz + minz) / 2; // Objektmittelpunkt (z)
58     }
59
60     rv6l_3d_msgs::RV6L_position bbx_msg;
61     bbx_msg.Class = bbx.Class;
62     bbx_msg.probability = bbx.probability;
63
64     bbx_msg.w = midy; // "horizontale" Position
65     bbx_msg.h = midz; // "vertikale" Position
66     bbx_msg.d = minx; // Abstand (min)
67
68     boxes->bounding_boxes.push_back(bbx_msg);
69 }
70
71 [ . . . ]
```

A.3 Bounding Box Message

[RV6L_position.msg, RV6L_positions.msg]

A.3.1 Struktur

Anhang

```
1 string Class
2 float32 probability
3 float32 w
4 float32 h
5 float32 d
```

A.3.2 Ausgabe (Topic)

```
1 ---
2   header:
3     seq: 5
4     stamp:
5       secs: 1644414390
6       nsecs: 296165943
7     frame_id: "camera_link"
8   bounding_boxes:
9   -
10    Class: "box_r"           // Objektklasse
11    probability: 0.9995404481887817 // Wahrscheinlichkeit
12    w: 0.3090042471885681          // hor. Position in m
13    h: 0.18566249310970306          // ver. Position in m
14    d: 0.5624291300773621          // Abstand in m
```

A.4 Launchfiles

A.4.1 RV6L 3D (Darknet)

[RV6L.launch]

```
1 <launch>
2
3   <!-- Config camera image topic -->
4   <arg name="camera_rgb_topic" default="/camera/color/
      image_raw" />
5
6   <!-- Console launch prefix -->
7   <arg name="launch_prefix" default="" />
8
9   <!-- Config and weights folder. -->
10  <arg name="yolo_weights_path"           default="$(find
      darknet_ros)/yolo_network_config/weights"/>
11  <arg name="yolo_config_path"           default="$(find
      darknet_ros)/yolo_network_config/cfg"/>
12
```

Anhang

```
13 <!-- ROS and network parameter files -->
14 <arg name="ros_param_file" default="$(find
15   darknet_ros)/config/ros.yaml"/>
16 <arg name="network_param_file" default="$(find
17   darknet_ros)/config/RV6L.yaml"/>
18 <!-- Load parameters -->
19 <rosparam command="load" ns="darknet_ros" file="$(arg
20   network_param_file)"/>
21 <rosparam command="load" file="$(find darknet_ros)/config/
22   ros.yaml"/>
23 <param name="darknet_ros/subscribers/camera_reading/topic"
24   type="string" value="$(arg camera_rgb_topic)" />
25 <!-- Start darknet and ros wrapper -->
26 <node pkg="darknet_ros" type="darknet_ros" name="darknet_ros"
27   " output="screen" launch-prefix="$(arg launch_prefix)">
28   <param name="weights_path" value="$(arg
29     yolo_weights_path)" />
30   <param name="config_path" value="$(arg
31     yolo_config_path)" />
32 </node>
33 <!-- Start darknet ros 3d -->
34 <node pkg="darknet_ros_3d" type="darknet3d_node" name="
35   darknet_3d" output="screen">
36   <rosparam command="load" file="$(find darknet_ros_3d)/
37     config/darknet_3d.yaml" />
38 </node>
39 </launch>
```

A.4.2 rv6l_object_detection

[first.launch]

```
1  <launch>
2
3  <!-- Start RealSense driver -->
4  <include file="$(find realsense2_camera)/launch/RV6L.launch"
5    />
6  <!-- Start rv6l_3d -->
7  <include file="$(find darknet_ros_3d)/launch/RV6L.launch" />
8
9  <!-- Start hardware interface -->
10 <include file="$(find rsv_cartesian_interface)/test/
11   test_hardware_interface.launch" />
```

Anhang

```
11
12 </launch>

[second.launch]

1  <launch>
2
3  <!-- Start execution -->
4  <include file="$(find rv6l_cell_config)/launch/
      rv6l_moveit_planning_execution_tut.launch" />
5
6  <!-- Start movement -->
7
8 </launch>
```

A.5 ROS Package

[package.xml]

```
1 <?xml version="1.0"?>
2 <package format="2">
3   <name>rv6l_main</name>
4   <version>1.0.0</version>
5   <description>Object detection and Point Cloud mapping using
        ROS Noetic and Intel RealSense 435</description>
6
7   <maintainer email="dgries@mailbox.org">David Gries</
        maintainer>
8
9   <url type="website">https://rzlapgle01.intern.th-ab.de/
        s170417/rv6l-kinect</url>
10
11  <buildtool_depend>catkin</buildtool_depend>
12
13  <build_depend>realsense2_camera</build_depend>
14  <build_depend>darknet_ros_3d</build_depend>
15  <build_depend>rv6l_3d_msgs</build_depend>
16  <build_depend>opencv_apps</build_depend>
17  <build_depend>pcl_ros</build_depend>
18  <build_depend>rv6l_pick_and_place</build_depend>
19  <build_depend>forward_command_controller</build_depend>
20  <build_depend>joint_state_controller</build_depend>
21  <build_depend>joint_trajectory_controller</build_depend>
22  <build_depend>position_controllers</build_depend>
23  <build_depend>rqt_joint_trajectory_controller</build_depend>
24  <build_depend>rsv_cartesian_interface</build_depend>
```

Anhang

```
25 <build_depend>rsv_joint_interface</build_depend>
26 <build_depend>rv6l</build_depend>
27 <build_depend>rv6l_cell</build_depend>
28 <build_depend>rv6l_cell_config</build_depend>
29 <build_depend>rv6l_config</build_depend>
30 <build_depend>nlopt</build_depend>
31
32 <build_export_depend>realsense2_camera</build_export_depend>
33 <build_export_depend>darknet_ros_3d</build_export_depend>
34 <build_export_depend>rv6l_3d_msgs</build_export_depend>
35 <build_export_depend>opencv_apps</build_export_depend>
36 <build_export_depend>pcl_ros</build_export_depend>
37 <build_export_depend>rv6l_pick_and_place</
    build_export_depend>
38 <build_export_depend>forward_command_controller</
    build_export_depend>
39 <build_export_depend>joint_state_controller</
    build_export_depend>
40 <build_export_depend>joint_trajectory_controller</
    build_export_depend>
41 <build_export_depend>position_controllers</
    build_export_depend>
42 <build_export_depend>rqt_joint_trajectory_controller</
    build_export_depend>
43 <build_export_depend>rsv_cartesian_interface</
    build_export_depend>
44 <build_export_depend>rsv_joint_interface</
    build_export_depend>
45 <build_export_depend>rv6l</build_export_depend>
46 <build_export_depend>rv6l_cell</build_export_depend>
47 <build_export_depend>rv6l_cell_config</build_export_depend>
48 <build_export_depend>rv6l_config</build_export_depend>
49 <build_export_depend>nlopt</build_export_depend>
50
51 <exec_depend>realsense2_camera</exec_depend>
52 <exec_depend>darknet_ros_3d</exec_depend>
53 <exec_depend>rv6l_3d_msgs</exec_depend>
54 <exec_depend>opencv_apps</exec_depend>
55 <exec_depend>pcl_ros</exec_depend>
56 <exec_depend>rv6l_pick_and_place</exec_depend>
57 <exec_depend>forward_command_controller</exec_depend>
58 <exec_depend>joint_state_controller</exec_depend>
59 <exec_depend>joint_trajectory_controller</exec_depend>
60 <exec_depend>position_controllers</exec_depend>
61 <exec_depend>rqt_joint_trajectory_controller</exec_depend>
62 <exec_depend>rsv_cartesian_interface</exec_depend>
63 <exec_depend>rsv_joint_interface</exec_depend>
64 <exec_depend>rv6l</exec_depend>
65 <exec_depend>rv6l_cell</exec_depend>
66 <exec_depend>rv6l_cell_config</exec_depend>
67 <exec_depend>rv6l_config</exec_depend>
68 <exec_depend>nlopt</exec_depend>
69
```

Anhang

```
70    <export>
71    </export>
72 </package>
```

B Realsense Launchfile

[RV6L.launch]

```
1 <!--
2 A launch file, derived from rgbd_launch and customized for
      Realsense ROS driver,
3 to publish XYZRGB point cloud like an OpenNI camera.
4
5 NOTICE: To use this launch file you must first install ros
      package rgbd_launch.
6
7 To launch Realsense with software registration (ROS Image
      Pipeline and rgbd_launch):
8     $ roslaunch realsense2_camera rs_rgbd.launch
9 Processing enabled by ROS driver:
10    # depth rectification
11 Processing enabled by this node:
12    # rgb rectification
13    # depth registration
14    # pointcloud_xyzrgb generation
15
16 To launch Realsense with hardware registration (ROS Realsense
      depth alignment):
17     $ roslaunch realsense2_camera rs_rgbd.launch align_depth:=
          true
18 Processing enabled by ROS driver:
19    # depth rectification
20    # depth registration
21 Processing enabled by this node:
22    # rgb rectification
23    # pointcloud_xyzrgb generation
24 -->
25
26 <launch>
27   <arg name="camera"                      default="camera"/>
28   <arg name="tf_prefix"                   default="$(arg camera)"/>
29   <arg name="external_manager"           default="false"/>
30   <arg name="manager"                    default="
          realsense2_camera_manager"/>
31
32   <!-- Camera device specific arguments -->
33
34   [ . . . ]
35
36   <arg name="depth_width"                default="-1"/>
37   <arg name="depth_height"               default="-1"/>
38   <arg name="enable_depth"              default="true"/>
39
40   [ . . . ]
41
```

Anhang

```
42 <arg name="color_width"           default="-1"/>
43 <arg name="color_height"          default="-1"/>
44 <arg name="enable_color"         default="true"/>
45
46 [ . . . ]
47
48 <arg name="depth_fps"           default="-1"/>
49 <arg name="infra_fps"           default="-1"/>
50 <arg name="color_fps"           default="-1"/>
51
52 [ . . . ]
53
54 <arg name="enable_pointcloud"    default="false"/>
55 <arg name="enable_sync"          default="true"/>
56 <arg name="align_depth"         default="true"/>
57 <arg name="filters"             default="pointcloud"/>
58
59 <arg name="publish_tf"          default="true"/>
60 <arg name="tf_publish_rate"     default="0"/> <!-- 0 --
      static transform -->
61
62 <!-- rgbd_launch specific arguments -->
63
64 <!-- Arguments for remapping all device namespaces -->
65 <arg name="rgb"                 default="color"
   />
66 [ . . . ]
67 <arg name="depth"               default="depth"
   />
68 <arg name="depth_registered_pub" default=""
   depth_registered" />
69 <arg name="depth_registered"    default=""
   depth_registered" unless="$(arg align_depth)" />
70 <arg name="depth_registered"    default=""
   aligned_depth_to_color" if="$(arg align_depth)" />
71 <arg name="depth_registered_filtered" default="$(arg
   depth_registered)" />
72 <arg name="projector"          default=""
   projector" />
73
74 <!-- Processing Modules -->
75 <arg name="rgb_processing"      default="true"/>
76 [ . . . ]
77 <arg name="depth_processing"    default="false"/
   >
78 <arg name="depth_registered_processing" default="true"/>
79 [ . . . ]
80
81 <group ns="$(arg camera)">
82
83 <!-- Launch the camera device nodelet-->
84 <include file="$(find realsense2_camera)/launch/includes/
   nodelet.launch.xml">
```

Anhang

```
85      [ . . . ]
86
87
88      <arg name="enable_pointcloud"           value="$(arg
89          enable_pointcloud)"/>
90      <arg name="enable_sync"                 value="$(arg
91          enable_sync)"/>
92      <arg name="align_depth"                value="$(arg
93          align_depth)"/>
94
95      [ . . . ]
96
97
98      <arg name="depth_width"               value="$(arg
99          depth_width)"/>
100     <arg name="depth_height"              value="$(arg
101    depth_height)"/>
102     <arg name="enable_depth"              value="$(arg
103    enable_depth)"/>
104
105     <arg name="color_width"               value="$(arg
106    color_width)"/>
107     <arg name="color_height"              value="$(arg
108    color_height)"/>
109     <arg name="enable_color"              value="$(arg
110    enable_color)"/>
111
112     [ . . . ]
113
114     <arg name="depth_fps"                value="$(arg
115        depth_fps)"/>
116     <arg name="color_fps"                 value="$(arg
117        color_fps)"/>
118     <arg name="filters"                  value="$(arg
119        filters)"/>
120
121     <arg name="publish_tf"               value="$(arg
122        publish_tf)"/>
123     <arg name="tf_publish_rate"          value="$(arg
124        tf_publish_rate)"/>
125
126     </include>
127
128     <!-- RGB processing -->
129     <include if="$(arg rgb_processing)"
130             file="$(find rgbd_launch)/launch/includes/rgb.
131             launch.xml">
132
133         <arg name="manager"                 value="$(arg
134            manager)"/>
135         <arg name="respawn"                value="$(arg
136            respawn)"/>
137         <arg name="rgb"                   value="$(arg
138            rgb)"/>
139         <arg name="debayer_processing"     value="$(arg
140            debayer_processing)"/>
```

Anhang

```
119      </include>
120
121  <group if="$(eval depth_registered_processing and
122        sw_registered_processing)">
123    <node pkg="nodelet" type="nodelet" name="register_depth"
124      args="load depth_image_proc/register $(arg manager
125            ) $(arg bond)" respawn="$(arg respawn)">
126      <remap from="rgb/camera_info"                      to="$(arg
127          rgb)/camera_info" />
128      <remap from="depth/camera_info"                   to="$(arg
129          depth)/camera_info" />
130      <remap from="depth/image_rect"                   to="$(arg
131          depth)/image_rect_raw" />
132      <remap from="depth_registered/image_rect" to="$(arg
133          depth_registered)/sw_registered/image_rect_raw" />
134    </node>
135
136    <!-- Publish registered XYZRGB point cloud with software
137        registered input -->
138    <node pkg="nodelet" type="nodelet" name="
139      points_xyzrgb_sw_registered"
140      args="load depth_image_proc/point_cloud_xyzrgb $(
141          arg manager) $(arg bond)" respawn="$(arg
142            respawn)">
143      <remap from="rgb/image_rect_color"                 to="$(arg
144          rgb)/image_rect_color" />
145      <remap from="rgb/camera_info"                     to="$(arg
146          rgb)/camera_info" />
147      <remap from="depth_registered/image_rect" to="$(arg
148          depth_registered_filtered)/sw_registered/
149              image_rect_raw" />
150      <remap from="depth_registered/points"           to="$(arg
151          depth_registered)/points" />
152    </node>
153  </group>
154
155  <group if="$(eval depth_registered_processing and
156        hw_registered_processing)">
157    <!-- Publish registered XYZRGB point cloud with hardware
158        registered input (ROS Realsense depth alignment) -->
159    <node pkg="nodelet" type="nodelet" name="
160      points_xyzrgb_hw_registered"
161      args="load depth_image_proc/point_cloud_xyzrgb $(
162          arg manager) $(arg bond)" respawn="$(arg
163            respawn)">
164      <remap from="rgb/image_rect_color"                 to="$(arg
165          rgb)/image_rect_color" />
166      <remap from="rgb/camera_info"                     to="$(arg
167          rgb)/camera_info" />
168      <remap from="depth_registered/image_rect" to="$(arg
169          depth_registered)/image_raw" />
170      <remap from="depth_registered/points"             to="$(arg
171          depth_registered_pub)/points" />
```

Anhang

```
148      </node>
149      </group>
150    </group>
151 </launch>
```

C Handhabung

C.1 Bounding Box Subscriber

[subscriber_cam.py]

```
1 #!/usr/bin/env python3
2
3 import rospy
4 from rv6l_3d_msgs.msg import RV6L_positions
5
6 class uwe:
7     def __init__(self):
8         rospy.init_node('cam_subscriber_Node', anonymous=None)
9         rospy.Subscriber("/darknet_ros_3d/bounding_boxes",
10                         RV6L_positions, self.callback_pos)
11
12     def callback_pos(self, data):
13         for box in data.bounding_boxes:
14             self.object_pos_cam = [box.Class, box.h, box.w,
15                                   box.d]
16
17 if __name__ == '__main__':
18     pos_cam = uwe()
19     rospy.wait_for_message("/darknet_ros_3d/bounding_boxes/",
20                           RV6L_positions, timeout=None)
21     print("[Objektklasse, horizontal, vertikal, Tiefe]: ",
22           pos_cam.object_pos_cam)
```

C.2 Anfahren des Objekts

[pick_and_place_detection.py]

```
1 #!/usr/bin/env python3
2
3 from bauplan_auswertung import get_position
4 from bauplan_auswertung import hoehenauswertung
5 from subscriber_cam import uwe
6 from rv6l_3d_msgs.msg import RV6L_positions
7
8 [ . . . ]
9
10 TCP    = 0.29      # Offset des TCP
11 REF_X = 0.89      # Scanposition X-Koordinate
```

Anhang

```
12 REF_Y = 0           # Scanposition Y-Koordinate
13 REF_Z = 1.56        # Scanposition Z-Koordinate
14
15 STATIC_X = 0.237   # Offset in X-Richtung
16 STATIC_Y = 0         # Offset in Y-Richtung
17 STATIC_Z = 0.235    # Offset in Z-Richtung
18
19 SCAN_NEAR = 0.30   # Hoehe fuer nahen Scan
20
21 [ . . . ]
22
23 class MoveGroupPythonInterface(object):
24     """MoveGroupPythonInterface"""
25
26     def __init__(self):
27
28 [ . . . ]
29
30     def go_to_pose_goal(self, posx, posy, posz):
31
32         move_group = self.move_group
33
34         ## Planning to a Pose Goal
35         ## ~~~~~
36         ## We can plan a motion for this group to a desired
37         ## pose for the
38         ## end-effector:
39         pose_goal = geometry_msgs.msg.Pose()
40         pose_goal.orientation.w = 0
41         pose_goal.orientation.x = 1
42         pose_goal.orientation.y = 0
43         pose_goal.orientation.z = 0
44         pose_goal.position.x = posx
45         pose_goal.position.y = posy
46         pose_goal.position.z = posz
47
48         move_group.set_pose_target(pose_goal)
49
50         ## Now, we call the planner to compute the plan and
51         ## execute it.
52         plan = move_group.go(wait=True)
53         # Calling 'stop()' ensures that there is no residual
54         ## movement
55         move_group.stop()
56         # It is always good to clear your targets after
57         ## planning with poses.
58         move_group.clear_pose_targets()
59
60         current_pose = self.move_group.get_current_pose().pose
61         return all_close(pose_goal, current_pose, 0.01)
62
63     def go_to_scan(self, posx, posy, posz):
```

Anhang

```
61     move_group = self.move_group
62
63     ## Planning to the specific scan of an object (only
64     ## with camera system)
65     ## ~~~~~
66     ## We can plan a motion for this group to a desired
67     ## pose for the
68     ## end-effector:
69     scan_goal = geometry_msgs.msg.Pose()
70     scan_goal.orientation.x = 0
71     scan_goal.orientation.y = 0.7071067811865464
72     scan_goal.orientation.z = 0
73     scan_goal.orientation.w = 0.7071067811865464
74     scan_goal.position.x = posx
75     scan_goal.position.y = posy
76     scan_goal.position.z = posz
77
78     ## Now, we call the planner to compute the plan and
79     ## execute it.
80     plan = move_group.go(wait=True)
81     # Calling 'stop()' ensures that there is no residual
82     ## movement
83     move_group.stop()
84     # It is always good to clear your targets after
85     ## planning with poses.
86     move_group.clear_pose_targets()
87
88
89 [ . . . ]
90
91 ######
92 # Codeabschnitt zur manuellen Objekterkennungserprobung #
93 ######
94
95 input("<ENTER> fuer Scanposition")
96 py.go_to_scan_pose() # Scanposition anfahren
97
98 # Klasse aus subscriber_cam initialisieren, Objektpos.
99 # auslesen
100 input("<ENTER> zum Ermitteln der ungefaehren
101 # Objektposition")
102 ichbinder = uwe()
103 rospy.wait_for_message("/darknet_ros_3d/bounding_boxes/",
104 RV6L_positions, timeout=None)
105 cam_pos = ichbinder.object_pos_cam # Aktuelle position in
106 # cam_pos speichern
107 print("Scanposition: ", (cam_pos[1] + REF_X), (cam_pos[2]
108 + REF_Y),(REF_Z - cam_pos[3] + SCAN_NEAR))
```

Anhang

```
104
105     input("<ENTER> Anfahren der Scanposition fuer den genauen
106         Scan")
107     py.go_to_scan((cam_pos[1] + REF_X), (cam_pos[2] + REF_Y),(
108         REF_Z - cam_pos[3] + SCAN_NEAR)) # Anfahren der
109         Scanposition ~ ueber dem Bauteil
110
111     print("genauer Mittelpunkt: ", (cam_pos[1] + REF_X +
112         cam_near[1]), (cam_pos[2] + REF_Y + cam_near[2]),(REF_Z
113         - cam_pos[3] + SCAN_NEAR - (cam_near[3] - SCAN_NEAR)))
114
115     input("<ENTER> Greifposition anfahren")
116     py.go_to_pose_goal((cam_pos[1] + REF_X + cam_near[1] +
117         STATIC_X), (cam_pos[2] + REF_Y + cam_near[2] + STATIC_Y
118         ), (REF_Z - cam_pos[3] + STATIC_Z - (cam_near[3] -
119             SCAN_NEAR)))
120
121
122     input("<ENTER> Endeffektor absenken")
123     py.go_to_pose_goal((cam_pos[1] + REF_X + cam_near[1] +
124         STATIC_X), (cam_pos[2] + REF_Y + cam_near[2] + STATIC_Y
125         ), (REF_Z - cam_pos[3] + STATIC_Z - (cam_near[3] -
126             SCAN_NEAR) - 0.25))
127
128     input("<ENTER> Zielposition anfahren")
129     py.go_to_pose_goal(float(pos1_gesucht[0]), float(
130         pos1_gesucht[1]), (float(pos1_gesucht[2]) + height2 +
131             TCP))
132
133     input("<ENTER> Objekt ablegen")
134     py.go_to_pose_goal(float(pos1_gesucht[0]), float(
135         pos1_gesucht[1]), (float(pos1_gesucht[2]) + height2 +
136             TCP - 0.25))
137
138     input("<ENTER> <SAUGER AUS>")
139     # Sauger aus
140
141     input("<ENTER> Endeffektor anheben")
142     py.go_to_pose_goal((cam_pos[1] + REF_X + cam_near[1] +
143         STATIC_X), (cam_pos[2] + REF_Y + cam_near[2] + STATIC_Y
144         ), (REF_Z - cam_pos[3] + STATIC_Z - (cam_near[3] -
145             SCAN_NEAR)))
```

Anhang

```
136
137      ##### Python pick and place Demo complete#####
138
139      print("===== Python pick and place Demo complete!")
140  except rospy.ROSInterruptException:
141      return
142  except KeyboardInterrupt:
143      return
144
145 if __name__ == "__main__":
146     main()
```

D Darknet

D.1 Darknet Konfiguration

[RV6L.cfg]

```
1 [net] # Alle
2 batch = 64
3
4 subdivisions = 16 (32 if issues occur)
5
6 max_batches = 10000 (higher number leads to greater accuracy
    but longer compute time)
7 max_batches = (2000 * number_of_classes, minimum 4000)
8 steps = (0.8 * max_batches), (0.9 * max_batches)
9
10
11 [convolutional] # Alle ueber [yolo]
12
13 filters = (number_of_classes + 5) * 3
14
15
16 [yolo] # Alle
17
18 classes = number_of_classes
```

D.2 Objekt Konfiguration

[obj.data]

```
1 classes = <Anzahl Klassen>
2 train = data/train.txt
3 valid = data/test.txt
4 names = data/obj.names
```

D.3 Generator für train.txt (Python)

[generate_train.py]

Anhang

```
1 #!/usr/bin/env python
2 import os
3
4 image_files = []
5 os.chdir(os.path.join("data", "obj"))
6 for filename in os.listdir(os.getcwd()):
7     if filename.endswith(".jpg"):
8         image_files.append("data/obj/" + filename)
9 os.chdir("..")
10 with open("train.txt", "w") as outfile:
11     for image in image_files:
12         outfile.write(image)
13         outfile.write("\n")
14     outfile.close()
15 os.chdir("..")
```

[62]