Lehrstuhl für Raumfahrttechnik

Technische Universität München

Prof. Prof. h.c. Dr. Dr. h.c.
Ulrich Walter

Semesterarbeit

Integration einer RGB-D-Kamera mit Visual-SLAM-Algorithmen in eine Simulationsumgebung zur Verbesserung der Nahbereichsnavigation im Orbit

LRT-SA 2015/10

Autor:

Tim Wiese

Betreuer: Dipl.-Ing. Martin Dziura
Lehrstuhl für Raumfahrttechnik
Technische Universität München

Zusammenfassung

Durch die immer weiter zunehmende Problematik des Weltraummülls sind in den letzten Jahren verstärkt wissenschaftliche Untersuchungen im Bereich des On-Orbit Servicings unternommen worden. Dabei besteht ein Teil der Mission in der kontrollierten Annäherung an ein Zielobjekt. Zur Unterstützung von Regelungsalgorithmen oder auch menschlichen Teleoperatoren ist es dabei hilfreich, sowohl die Lage also auch die Form des Objekts ermitteln zu können und digital zu erfassen bzw. darzustellen.

Als Sensorsystem kommen dabei als Alternative zu kostspieligen und komplizierten LIDAR-Systemen sogenannte RGB-D-Sensoren in Betracht. Sie bieten die Möglichkeit, aus einem zeitgleich aufgenommenen Farb- und Tiefenbild auf kurze Distanzen Lageschätzungs- und 3D-Rekonstruktionsaufgaben durchzuführen. Die entsprechenden Algorithmen sind mittlerweile soweit fortgeschritten, dass sie in Echtzeit auf einer handelsüblichen Desktop-CPU ausgeführt werden können.

In dieser Arbeit wurden für den RACOON-Hardware-in-the-Loop-Simulator aktuelle Algorithmen ausgewählt, die mit einer auf einem Verfolgersatelliten installierten Kinect-v2-Kamera die Lage eines unbekannten, unkooperativen Zielsatelliten schätzen und eine 3D-Darstellung erzeugen sollen. Im Gegensatz zu klassischen, merkmalspunktbasierten Ansätzen kommen dabei dichte Lageschätzungs- und Rekonstruktionsalgorithmen zum Einsatz.

Zu diesem Zweck wurde ein Softwareframework entwickelt, in das sich diese und andere Algorithmen und Sensortypen integrieren und auf einfache Weise miteinander vergleichen lassen. Über eine flexible Pipelinestruktur und Wrapperklassen mit klar definierten Schnittstellen für Algorithmen und Datenquellen wurde eine einfache Erweiterbarkeit und vielfältige Einsetzbarkeit sichergestellt.

Mit dieser Arbeit soll ein erster Schritt hin zu einer umfassenden Lösung zur Lageschätzung und 3D-Rekonstruktion im Rahmen der teleoperierten Nahbereichsnavigation im Orbit unternommen werden. Die Ergebnisse erster qualitativer Rekonstruktionstests zeigen, dass die ausgewählten Algorithmen grundsätzlich in der Simulatorumgebung einsetzbar sind, allerdings erweiterte Funktionalitäten zur Driftreduktion nötig sind, um in einem Teleoperationsszenario einsetzbar zu sein.

Abstract

Due to the growing problem of space debris, research in the area of on-orbit servicing has intensified over the last years. Part of such missions is a controlled approach of a target object. To support control algorithms or human teleoperators in such a task, it can be helpful to detect the pose as well as the geometry of the object.

As an alternative to expensive and complicated LIDAR systems, RGB-D sensors could be an option. They allow to perform pose estimation and 3D reconstruction tasks by recording a color and a depth frame simultaneously. Applicable algorithms have advanced such that they can be run on a consumer desktop CPU in real-time.

In this thesis, current state-of-the-art algorithms have been selected for the RACOON hardware-in-the-loop-simulator to make use of a Kinect v2 camera that has been installed on a chaser satellite to estimate the pose of an unknown, uncooperative target satellite and create a 3D reconstruction. In comparison with traditional, feature-based methods, dense pose estimation and 3D reconstruction algorithms are used.

For this purpose, a software framework was developed, with which these and other algorithms and sensor types can be easily compared. By employing a flexible pipeline structer as well as wrapper classes with clearly defined interfaces for algorithms and data sources, easy extensibility and versatility are ensured.

With this thesis, a first step should be taken towards a comprehensive solution for pose estimation and 3D reconstruction in the context of teleoperated close-range navigation in orbit. Results of preliminary qualitative reconstruction tests show that the chosen algorithms are generally usable in the given simulation environment, even though extended functionalities for dift reduction are necessary to make them applicable for a teleoperation scenario.

Inhaltsverzeichnis

1 MOTIVATION 1

2 STAND DER TECHNIK 3

2.1 Lageschätzung und Objektrekonstruktion im Orbit 3

2.2 RGB-D-Sensoren 4

2.3 Algorithmen für RGB-D-Kameras 6

3 AUFGABENSTELLUNG 10

3.1 Szenariobeschreibung 10

3.2 Ziele der Arbeit 11

3.3 Vorgehensweise 11

3.4 Abgrenzung 12

4 ALGORITHMENAUSWAHL 13

4.1 Anforderungen 13

4.2 Auswahl 14
4.2.1 Lageschätzung 14
4.2.2 3D-Rekonstruktion 15

4.3 DIFODO Tracker 16
4.3.1 Basisinformationen 16
4.3.2 Mathematische Grundlagen 16

4.4 dvo_slam Tracker 19
4.4.1 Basisinformationen 19
4.4.2 Mathematische Grundlagen 19

4.5 fastfusion Mapper 21
4.5.1 Basisinformationen 21
4.5.2 Mathematische Grundlagen 21

5 SOFTWAREFRAMEWORK 25

5.1 Anforderungen 25

5.2 Grundidee 26

5.3 Verwendete Bibliotheken 27
5.3.1 PCL 27
5.3.2 OpenCV 27
5.3.3 Eigen 27
5.3.4 Qt 27

5.4 Verwendete Programmiersprache 28

5.5 Softwarearchitektur 28
5.5.1 Grundaufbau 28
5.5.2 DriverManager 30
5.5.3 SLAMPipeline 35
5.5.4 ViewManager 39
5.5.5 DataManager 41

5.6 Datentypen & Schnittstellen zur Erweiterung 42
5.6.1 FramePackage 43
5.6.2 Wrapper-Klassen 44
5.6.3 Externe Sensordatensätze 46

6 ERGEBNISSE 48

6.1 Algorithmenauswahl 48

6.2 Softwarefunktionen 49

6.3 Testdatensätze 50

6.4 Rekonstruktionsqualität 51

6.5 Parametereinfluss 53

6.6 Ausführungsgeschwindigkeit 54

7 DISKUSSION 55

7.1 Softwareframework 55

7.2 Rekonstruktionsgüte 55

8 ZUSAMMENFASSUNG 56

9 AUSBLICK 57

A ANHANG 58

A.1 Literaturverzeichnis 58

Abbildungsverzeichnis

Abb. 1–1: Konzeptzeichnung des RACOON-Lab, entnommen aus (Harder 2017) 1
Abb. 2–1: ATV-5 Georges Lemaître mit eingezeichnetem LIDAR-Sensor, entnommen aus (Kolb et al. 2015) 3
Abb. 2–2: ISS abgetastet mit LIDAR-Sensor aus 30 Metern Entfernung, entnommen aus (Kolb et al. 2015) 3
Abb. 2–3: Kinect for Xbox One (Kinect v2) 5
Abb. 2–4: Bewegungstracking auf Marsuntergrund mit Merkmalspunkten, entnommen aus (Maimone et al. 2007) 6
Abb. 2–5: KinectFusion-Pipeline, entnommen aus (Izadi et al. 2011) 8
Abb. 2–6: TSDF-Visualisierung, entnommen aus (Korcz 2011) 8
Abb. 3–1: Aufbau des RACOON-Simulators, entnommen aus (Dziura et al. 2017) 10
Abb. 4–1: Geometrische Zusammenhänge, entnommen aus (Jaimez und González-Jiménez 2015) 17
Abb. 4–2: Bewegungsschätzung durch Minimierung der photometrischen und geometrischen Abweichung, entnommen aus (Kerl et al. 2013a) 21
Abb. 4–3: Visuelle Odometrie, optimierte Trajektorie und Schleifenerkennung, entnommen aus (Kerl et al. 2013a) 21
Abb. 4–4: Visualisierung des fastfusion-Algorithmus über die Zeit, entnommen aus (Steinbrücker et al. 2014). Oben nach unten: Ergebnisse nach 50, 170, 300 Frames. Links nach rechts: Geometrie und Textur, nur Geometrie, texturiertes Drahtgitter, Darstellung des Voxelrasters. 23
Abb. 5–1: Klassendiagramm der Kernkomponenten 28
Abb. 5–2: Verarbeitungspipeline der RVDSLAM-Software 29
Abb. 5–3: `DriverManager`-Klassendiagramm 30
Abb. 5–4: Gesamtklassendiagramm 31
Abb. 5–5: Auswirkungen des `BoxFilters` auf Farb- und Tiefenbild, entnommen aus (Dziura et al. 2017) 34
Abb. 5–6: `SLAMPipeline`-Klassendiagramm 35
Abb. 5–7: `ViewManager`-Klassendiagramm 39
Abb. 5–8: Benutzeroberfläche 40
Abb. 5–9: `FramePackage`-Klassendiagramm 43
Abb. 5–10: Klassendiagramm der Wrapperklassen 44
Abb. 5–11: Zustandsübergangsdiagramm des Zustandsautomaten der Module 45
Abb. 6–1: Beispielbilder aus den aufgezeichneten Datensätzen 50
Abb. 6–2: Rekonstruktionsergebnisse – Beleuchtung von oben nach unten: Laborbeleuchtung, Sonne hinten, Sonne links – Drehwinkel des Modells von links nach rechts: 45°, 90°, 180°, 270° 52

Abb. 6–3: Durch den Tracking-Algorithmus berechnete Relativtrajektorie zwischen Chaser und Target 52
Abb. 6–4: Einfluss der Anzahl der Ebenen der Gaußfilterpyramide des DIFODO-Algorithmus auf die Rekonstruktionsqualität 53
Abb. 6–5: Einfluss der Voxelgröße des fastfusion-Algorithmus auf die Rekonstruktionsqualität 53

Tabellenverzeichnis

Tab. 2–1: Technische Daten der Kinect v2, entnommen aus (Fankhauser et al. 2015) 5
Tab. 4–1: Anforderungen an die Algorithmenauswahl 13
Tab. 5–1: Anforderungen an das Softwareframework 25
Tab. 5–2: `BoxFilter`-Parameter 33
Tab. 5–3: DIFODO-Parameter 37
Tab. 5–4: fastfusion-Parameter 38
Tab. 5–5: Zu implementierende Methoden der `FramePackage`-Klasse 43
Tab. 5–6: Zu implementierende Methoden der `FramePackage::Filter`-Klasse 44
Tab. 5–7: Zu implementierende Methoden der `DriverWrapper`-Klasse 45
Tab. 5–8: Zu implementierende Methoden der `TrackerWrapper`-Klasse 46
Tab. 5–9: Zu implementierende Methoden der `MapperWrapper`-Klasse 46
Tab. 6–1: Übersicht über die Erfüllung der Algorithmenanforderungen 48
Tab. 6–2: Übersicht über die Erfüllung der Softwareanforderungen 49
Tab. 6–1: Eigenschaften des Referenzszenarios 50
Tab. 6–2: Kinect-v2-Eigenschaften 51
Tab. 6–3: Verwendete DIFODO-Parameter 51
Tab. 6–4: Verwendete fastfusion-Parameter 51
Tab. 6–5: Verwendete `BoxFilter`-Parameter 51
Tab. 6–6: fastfusion-Verarbeitungsgeschwindigkeit (Intel Core i7-5820K @ 3,30 GHz) 54

Abkürzungen

3D	dreidimensional
ATV	Automated Transfer Vehicle
API	Application Programming Interface
BRIEF	Binary Robust Independent Elementary Features
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CL	Computing Language
CV	Computer Vision
DARPA	Defense Advanced Research Projects Agency
DEOS	Deutsche Orbitale Servicing Mission
DIFODO	Differential Odometry
DVO	Dense Visual Odometry
EKF	Extended Kalman Filter
ESA	European Space Agency
ETS	Engineering Test Satellite
FAST	Features from Accelerated Segment Test
GNC	Guidance, Navigation, Control
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
ICP	Iterative Closest Point
ISS	International Space Station
LIDAR	Light Detection and Ranging
MOC	Meta Object Compiler
MRPT	Mobile Robot Programming Toolkit
NASA	National Aeronautics and Space Administration
NASDA	National Space Development Agency
NI	Natural Interaction
OOS	On-Orbit Servicing
ORB	Oriented FAST and Rotated BRIEF
PCL	Point Cloud Library
PNG	Portable Network Graphics

RACOON	Robotic Actuation and On-Orbit Navigation
RANSAC	Random Sample Consensus
RGB-D	Red Green Blue Depth
ROS	Robot Operating System
RVD	Rendezvous and Docking
SDK	Software Development Kit
SLAM	Simultaneous Localization and Mapping
SIFT	Scale-Invariant Feature Transform
SIMD	Single Instruction Multiple Data
SURF	Speeded Up Robust Features
ToF	Time of Flight
TSDF	Truncated Signed Distance Function

1 Motivation

Am Lehrstuhl für Raumfahrttechnik existiert mit dem RACOON-Lab (Robotic Actuation and On-Orbit Navigation Laboratory) ein umfangreich ausgestatteter Hardware-in-the-Loop-Simulator für Rendezvous-Szenarien im Erdorbit (Fleischner et al. 2012). Abb. 1–1 zeigt eine Konzeptzeichnung. Hier können verschiedene Konstellationen zwischen einem Zielsatelliten und einem von einem Teleoperator ferngesteuerten Verfolgersatelliten abgebildet und orbitmechanisch korrekt modelliert abgefahren werden. Der Verfolger ist dabei mit verschiedenen Sensorsystemen ausgestattet, deren Daten auf einem angeschlossenen Computer weiterverarbeitet und dem Operator optisch aufbereitet zur Verfügung gestellt werden können.

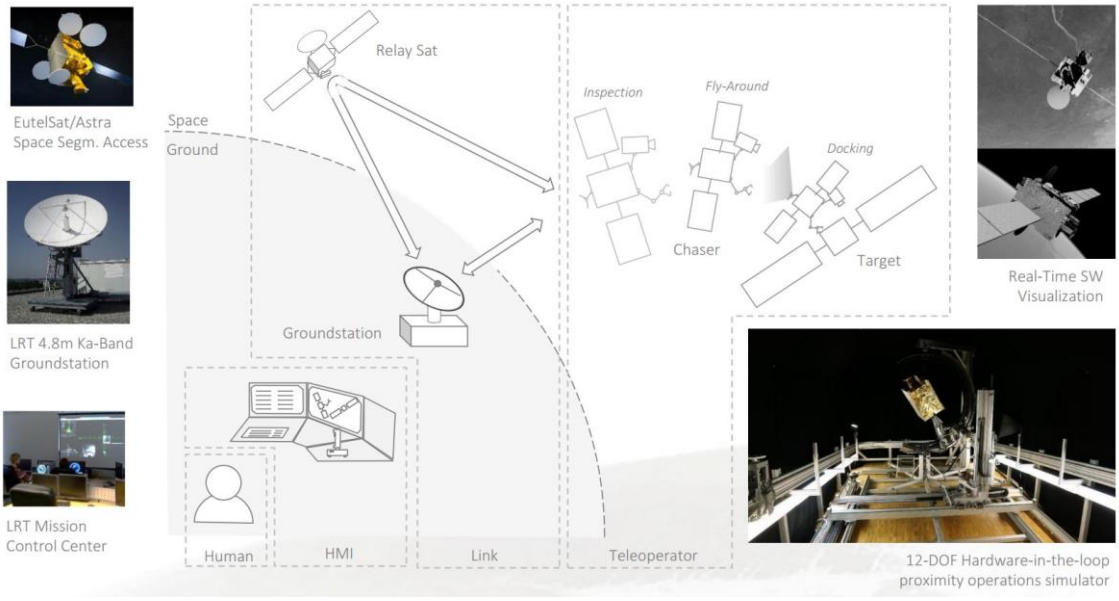


Abb. 1–1: Konzeptzeichnung des RACOON-Lab, entnommen aus (Harder und Dziura 2017)

Im Rahmen von On-Orbit-Servicing-Aufgaben (kurz OOS) besteht ein Teil der Mission in der kontrollierten Annäherung an ein Objekt (bspw. einen Satelliten oder ein Stück Weltraummüll). Zur Unterstützung von Regelungsalgorithmen und menschlichen Teleoperatoren ist es dabei hilfreich, sowohl die Lage (Tracking) als auch die Form (Mapping) des Objekts ermitteln zu können und digital zu erfassen bzw. darzustellen. Derzeit kommen in diesem auch GNC (Guidance, Navigation, Control) genannten Gebiet insbesondere sogenannte LIDAR-Sensoren (Light Detection and Ranging) zum Einsatz (siehe Abschnitt 2.1).

Für erdgebundene Tracking- und Mapping-Szenarien sind in den letzten Jahren kamerabasierte Systeme in den Fokus gerückt. Systeme wie die Microsoft Kinect sind günstig, kompakt und energieeffizient (Zhang 2012). Die zugehörigen Algorithmen wurden in den letzten Jahren bezüglich Genauigkeit, Robustheit und Laufzeit stark weiterentwickelt. Der aktuelle Stand der Technik wird in **Kapitel 2** näher beleuchtet.

Einer der im RACOON-Simulator angeschlossenen Sensoren ist eine Kinect-v2-Kamera, die neben einem Farbbild auch ein Tiefenbild liefert. Mithilfe entsprechender Algorithmen lassen sich daraus dreidimensionale Farbdarstellungen der Umgebung in Echtzeit generieren, bei gleichzeitiger Robustheit in Bezug auf Verdeckungen und

ungünstige Lichtverhältnisse. Es erscheint vielversprechend, den Einsatz dieser Techniken im Kontext von Weltraumanwendungen zu untersuchen. In dieser Arbeit soll dazu der Grundstein gelegt werden. Details zur Aufgabenstellung finden sich in **Kapitel 3**.

Grundsätzlich sind die entsprechenden 3D-Rekonstruktionsalgorithmen für kamerabasierte Systeme auf den Einsatz in Erdanwendungen wie der dreidimensionalen Kartierung von Räumen ausgerichtet, bei denen sich die Kamera durch eine statische Umgebung bewegt. Bei der Verwendung in einem OOS-Szenario bewegen sich Verfolger- und Zielobjekt relativ zueinander vor einem (weit entfernten) Hintergrund bei dynamischen Lichtverhältnissen (Sonneneinstrahlung, Erdalbedo). In dieser Arbeit sollen deshalb Algorithmen ausgewählt werden, die sich aufgrund ihrer Eigenschaften bzgl. Robustheit und Laufzeit für den Einsatz in solch einem Szenario eignen könnten, und mit im RACOON-Simulator aufgenommenen Datensätzen getestet und bewertet werden. Die Auswahl ist Gegenstand von **Kapitel 4**.

Zur Unterstützung dieser Aufgabe soll im Rahmen dieser Arbeit außerdem ein Softwareframework entwickelt werden, das die verschiedenen Algorithmen integriert und sich flexibel um weitere Algorithmen und Sensortypen erweitern lässt. Die Software soll eine Visualisierungskomponente beinhalten, die sowohl Lage als auch Form des Zielobjekts darstellt. So lassen sich in zukünftigen Untersuchungen unterschiedliche Ansätze auf einfache Weise miteinander vergleichen. Im Gegensatz zu bereits existierenden, umfangreicheren Frameworks wie ROS[1] (Robot Operating System) bietet solch eine spezialisierte Softwarelösung sowohl für Nutzer als auch für Entwickler eine kürzere Einarbeitungszeit und geringere Komplexität bei Verwendung und Erweiterung. **Kapitel 5** beschäftigt sich mit der Architektur, der Entwicklung und den Funktionen des Softwareframeworks.

Um die Eignung der ausgewählten Algorithmen zu verifizieren und ihre Eigenschaften im gegebenen Szenario zu analysieren, wurden im RACOON-Simulator einige Testdatensätze mithilfe des entwickelten Frameworks aufgezeichnet und die 3D-Rekonstruktion durchgeführt. Die Ergebnisse werden in **Kapitel 6** aufgezeigt und in **Kapitel 7** diskutiert.

Abschließend fasst **Kapitel 8** die Arbeit zusammen und **Kapitel 9** gibt einen Ausblick auf mögliche nächste Schritte, hin zu einer umfassenden Lösung zur Lageschätzung und 3D-Rekonstruktion im Rahmen der teleoperierten Nahbereichsnavigation im Orbit.

[1] http://www.ros.org/

2 Stand der Technik

2.1 Lageschätzung und Objektrekonstruktion im Orbit

OOS-Szenarien sind in den letzten Jahren insbesondere aufgrund der immer weiter zunehmenden Problematik des Weltraummülls in den Fokus gerückt (Pelton 2013). Ziel ist dabei, entweder Objekte im Orbit wie bspw. Satelliten zu warten, um die Nutzungsdauer zu verlängern, oder ausgediente Objekte kontrolliert aus ihrer Umlaufbahn zu entfernen. Dabei besteht ein Teil der Mission in der kontrollierten Annäherung an ein Zielobjekt, welches durch Sensoren erfasst wird.

Erste Forschungsmissionen im Bereich des OOS wurden 1997 von der japanischen Raumfahrtagentur NASDA (National Space Development Agency) mit dem Satellit ETS-VII (Engineering Test Satellite VII) und 2007 von der amerikanischen Militärforschungsbehörde DARPA (Defense Advanced Research Projects Agency) mit der Mission Orbital Express durchgeführt . Die erste geplante deutsche Mission ist DEOS (Deutsche Orbitale Servicing Mission, Reintsema et al. 2010), welche 2018 starten soll. Auf europäischer Ebene wird derzeit die Mission e.Deorbit der ESA (European Space Agency) mit einem geplanten Start im Jahr 2021 vorangetrieben, bei der ein Rendezvous mit einem ausrangierten, unkooperativen Satelliten durchgeführt werden soll.

Ein aktuelles System zum Rendezvous, welches im Orbit getestet wurde, kam in der letzten ATV-Mission (Automated Transfer Vehicle) im Jahr 2014 zum Einsatz (Cavrois et al. 2015). Zur autonomen Annäherung wurden ab 3,5 km Entfernung LIDAR-Sensoren eingesetzt, um die internationale Raumstation ISS (International Space Station) zu erfassen und die relative Lage zwischen Raumfahrzeug und Raumstation zu ermitteln. Abb. 2–1 und Abb. 2–2 zeigen das System im Einsatz.

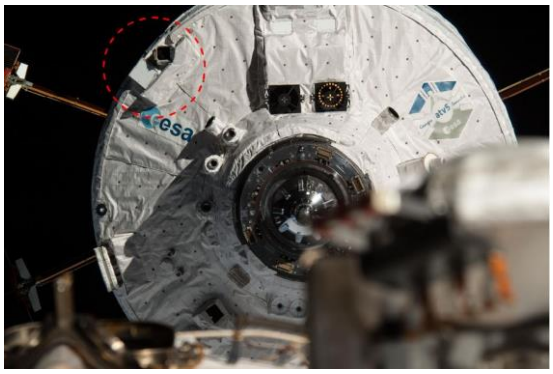


Abb. 2–1: ATV-5 Georges Lemaître mit eingezeichnetem LIDAR-Sensor, entnommen aus (Kolb et al. 2015)

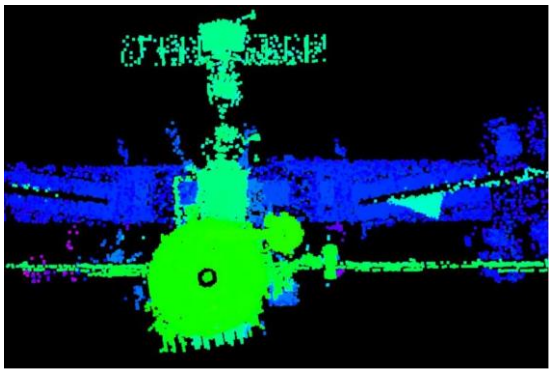


Abb. 2–2: ISS abgetastet mit LIDAR-Sensor aus 30 Metern Entfernung, entnommen aus (Kolb et al. 2015)

LIDAR-Sensoren (Light Detection and Ranging) basieren auf einem dem Radar ähnlichen Prinzip zur optischen Abstands- und Geschwindigkeitsmessung (Fisher und Konolige 2008). Bei den meisten Systemen kommt ein Laserstrahl zum Einsatz, der, über einen Spiegel abgelenkt, in jedem Zeitschritt die gesamte Szene abtastet. Über die Laufzeit des zurückgestreuten Lichts kann die Entfernung ermittelt werden (Time-of-Flight-Messprinzip, kurz ToF).

Als Alternative zu LIDAR-Sensoren bietet sich die Verwendung von kamerabasierten System an, die Vorteile in Bezug auf Kosten, Gewicht und Leistungsaufnahme besitzen. Dazu wurden in den letzten Jahren mehrere Studien veröffentlich, die sich mit dem Einsatz von Einzel- oder Stereokamerasystemen zur Lageschätzung und Objektrekonstruktion im Orbit beschäftigen (Schnitzer et al. 2012; Oumer und Panin 2012; Petit et al. 2011). Die dabei verwendeten Algorithmen, wie bspw. EKF SLAM (Extended Kalman Filter Simultaneous Localization and Mapping, Solà 2014) und FastSLAM (Montemerlo et al. 2002), basieren allesamt auf dem Verfolgen einzelner Merkmalspunkte (engl.: feature points) und den daraus resultierenden dünnbesetzten Punktwolken (engl.: sparse point clouds).

2.2 RGB-D-Sensoren

Ein neuer Sensortyp, der insbesondere im Bereich der Robotik in den Fokus gerückt ist, sind sogenannte RGB-D-Kameras (Red Green Blue Depth), welche gleichzeitig ein Farb- und ein Tiefenbild der Umgebung liefern. Das Tiefenbild wird erzeugt, indem Licht einer bestimmten Wellenlänge in die Umgebung ausgesendet und von einem Sensor wieder erfasst und ausgewertet wird (Hirschmüller et al. 2015). Da Farb- und Tiefensensor bei dieser Art Kamera nicht exakt an der gleichen Position liegen, muss eines der Bilder vor Verwendung so umgerechnet werden, dass die Werte an gleichen Positionen im Bild miteinander korrelieren. Dies geschieht durch die Reprojektion der Pixelwerte von der einen zur anderen Sensorposition mittels der bekannten Transformation zwischen den beiden Sensorursprungskoordinaten.

RGB-D-Kameras sind in gewisser Weise ein Hybrid aus einer monokularen Kamera und einem scannerlosen LIDAR. Gegenüber Stereokameras bieten RGB-D-Sensoren verlässlichere und genauere Tiefenmessungen in Szenen mit geringer visueller Textur (Hirschmüller et al. 2015). Durch den dedizierten aktiven Tiefensensor wird das Korrespondenzproblem (Enqvist 2009) umgangen und die Sensoren arbeiten auch bei Dunkelheit. Im Vergleich mit scannerbasierten LIDAR-Sensoren arbeitet die Tiefenerfassung jedoch nur auf Distanzen von wenigen Metern, mit geringerer Genauigkeit, stärker rauschbehafteten Messwerten und einem eingeschränkten Sichtfeld (Henry et al. 2012), weshalb RGB-D-Kameras nur im Bereich der Nahbereichsnavigation als Alternative in Betracht kommen. Je nach Funktionsweise kann Umgebungslicht außerdem störend sein (Hirschmüller et al. 2015). Die Sensoren sind jedoch sehr günstig und kompakt, haben eine geringe Leistungsaufnahme und bieten durch die Farbbildaufnahme weitergehende Möglichkeiten zur Bildverarbeitung.

Die bekannteste kommerziell verfügbare RGB-D-Kamera ist die Kinect von Microsoft. Sie wurde als Eingabegerät zur Bewegungssteuerung für die Spielekonsole Xbox entwickelt, verbreitete ich jedoch aufgrund ihres günstigen Preises und ihrer globalen Verfügbarkeit schnell in Forschungslaboren (Zhang 2012). 2010 wurde die erste Version veröffentlicht. Die Tiefenbilder der Kinect v1 werden über eine Technik erzeugt, die „strukturiertes Licht“ (engl.: structured light) genannt wird. Dabei wird ein Infrarotmuster auf die Umgebung projiziert, aus dessen Verformung in der Sensorabbildung auf die Tiefe geschlossen werden kann. In der aktuellen Version der Kamera von 2013, der Kinect v2, wird dagegen das ToF-Prinzip eingesetzt (Lachat et al. 2015). Dabei wird die Entfernung über die Verzögerung des Wiedereintreffens von ausgesendeten, von der Umgebung zurückgeworfenen Infrarotlichtpulsen ermittelt.

Die dadurch erzeugten Tiefenwerte sind genauer und feiner aufgelöst, da für jeden Pixel direkt ein Tiefenwert gemessen werden kann.

Abb. 2–3 zeigt die aktuelle Kinect-Version. Die technischen Daten sind in Tab. 2–1 aufgelistet.

Abb. 2–3: Kinect for Xbox One (Kinect v2)

Tab. 2–1: Technische Daten der Kinect v2, entnommen aus (Fankhauser et al. 2015)

Eigenschaft	Wert
Farbbildauflösung	bis zu 1920 x 1080 px
Horizontales Farbbildsichtfeld	84,1 Grad
Vertikales Farbbildsichtfeld	53,8 Grad
Farbtiefe	24 bit
Tiefenbildauflösung	512 x 424 px
Horizontales Tiefensichtfeld	70,6 Grad
Vertikales Tiefensichtfeld	60 Grad
Tiefenwertgenauigkeit	13 bit
Datenrate	15 oder 30 Hz, je nach Lichtbedingungen
Minimale Tiefe	0,5 Meter
Maximale Tiefe	4,5 Meter
Energieverbrauch	ca. 15 Watt

Für die Entwicklung mit der Kinect hat Microsoft für Windows ein zugehöriges SDK (Software Development Kit) veröffentlicht[2], mit dem auf die Funktionen der Kamera zugegriffen werden kann und die Daten ausgelesen werden können. Für Linux gibt es keine offiziellen Treiberpakete. Hervorgerufen durch großes Interesse im Forschungsbereich gibt es jedoch inoffizielle Treiber wie libfreenect[3] (für die Kinect v1) und libfreenect2[4] (für die Kinect v2), die den Einsatz unter Linux ermöglichen.

[2] http://www.microsoft.com/en-us/download/details.aspx?id=44561
[3] https://github.com/OpenKinect/libfreenect
[4] https://github.com/OpenKinect/libfreenect2

Darüber hinaus existiert mit OpenNI[5] (Open Natural Interaction) bzw. OpenNI 2[6] ein Framework für verschiedene RGB-D-Kameras, das neben einem einheitlichen Zugriff auf die Sensordaten eine API (Application Programming Interface) zur weitergehenden Dateninterpretation wie Handgestenerkennung und Personentracking bietet. Das Framework ist sowohl unter Linux als auch unter Windows einsetzbar. Es unterstützt beispielsweise die ASUS Xtion Pro Live RGB-D-Kamera, die auch häufig zu Forschungszwecken eingesetzt wird.

2.3 Algorithmen für RGB-D-Kameras

Algorithmen zur Lageschätzung von RGB-D-Kameras werden unter dem Begriff „visuelle Odometrie“ (Nister et al. 2004) zusammengefasst. Dabei wird aus aufeinanderfolgenden Farb- und Tiefenbildern die Bewegung der Kamera zwischen den beiden Aufnahmepositionen geschätzt. Die meisten bisher entwickelten, „klassischen“ Methoden in diesem Bereich arbeiten dabei mit einzelnen Merkmalspunkten, die aus den Bildern extrahiert werden, und zwischen denen Korrespondenzen ermittelt werden (Huang et al. 2011; Endres et al. 2012; Henry et al. 2012). Merkmalspunkte sind charakteristische Punkte, die robust gegenüber Ausrichtungs-, Skalen- und Farbänderungen sind. Bekannte Merkmalspunktarten sind beispielsweise SIFT (Scale Invariant Feature Transform, Lowe 1999), SURF (Speeded-up Robust Features, Bay et al. 2008) und ORB (Oriented FAST and Rotated BRIEF, Rublee et al. 2011). Die Korrespondenzen zwischen Merkmalspunkten werden meist über Varianten des ICP-Algorithmus (Iterative Closest Point, Besl und McKay 1992) ermittelt, welcher versucht, zwei dünnbesetzte Punktwolken möglichst gut übereinanderzulegen. Die nötige Transformation zwischen den Punktwolken beschreibt dann die zugehörige Kamerabewegung. Ausreißern kann durch den Einsatz von RANSAC (Random Sample Consensus) begegnet werden (Fischler und Bolles 1981).

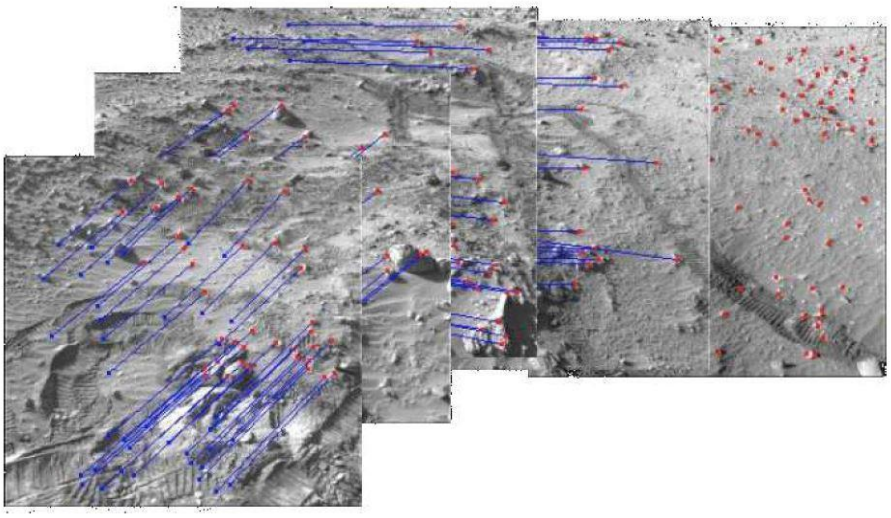


Abb. 2–4: Bewegungstracking auf Marsuntergrund mit Merkmalspunkten, entnommen aus (Maimone et al. 2007)

[5] https://github.com/OpenNI/OpenNI
[6] https://structure.io/openni

Ursprünglich wurden Algorithmen im Bereich visuelle Odometrie für Mono- und Stereo-Kamerasysteme entwickelt. Ein System basierend auf der Zuordnung von Merkmalspunkten ist beispielsweise auf den beiden Marsrovern Spirit[7] und Opportunity der NASA (National Aeronautics and Space Administration) im Einsatz, um den Bewegungsfortschritt zu ermitteln (Maimone et al. 2007). Abb. 2–4 zeigt dabei entstandene Aufnahmen mit eingezeichneten Punktkorrespondenzen.

Aktuellere Methoden verwenden die gesamte Bildinformation zur Bewegungsschätzung. Hierbei werden keinerlei Informationen verworfen, was diese Ansätze genauer und robuster macht. Auch hier kann erneut ICP zum Einsatz kommen, um die (nun dichten) 3D-Punktwolken zur Überlappung zu bringen. Hierfür werden die Informationen aus dem Tiefenbild direkt genutzt. Ein vielversprechender aktueller Ansatz, der auch nur die Informationen des Tiefenbildes verwendet, basiert auf der sogenannten Entfernungsflussbedingung (engl.: range flow constraint), bei der die zeitliche Änderung der Tiefenwerte mit den Translations- und Rotationsgeschwindigkeiten der Kamera in Zusammenhang gebracht wird (Jaimez und González-Jiménez 2015).

Alternativ kann auch die photometrische Abweichung zwischen zwei Bildern minimiert werden, d.h. die Abweichung zwischen den Pixelintensitäten. Ein aktueller Ansatz dazu stammt von (Steinbrücker et al. 2011) und eine darauf aufbauende Weiterentwicklung von (Kerl et al. 2013b), in der die Fehlerfunktion probabilistisch formuliert wird und dadurch ein Modell für das Sensorrauschen und für die erwartete Bewegung integriert werden kann, was das Tracking genauer und robuster macht.

Grundsätzlich haben rein photometrische Ansätze allerdings den Nachteil, dass sie beleuchtungsabhängig sind und die Trackingqualität bei schlechten Lichtverhältnissen abnimmt. Um die Vorteile von beiden Methoden zu nutzen, bietet es sich deshalb an, die photometrische Abweichung zusammen mit der geometrischen Abweichung zu betrachten. Aktuelle Veröffentlichung dazu sind bspw. (Tykkälä et al. 2011; Meilland und Comport 2013).

Eine allgemeine Limitierung von grundlegenden Algorithmen für visuelle Odometrie ist, dass sie lediglich das vorhergehende und das aktuell aufgenommene Bild verwenden, um die Kamerabewegung zu schätzen. Fehler werden so über die Zeit akkumuliert, der Kamerapositionsverlauf „driftet“ von der tatsächlichen Trajektorie ab. Um diesem Problem zu begegnen, können die Algorithmen um sogenannte „SLAM“-Funktionalitäten erweitert werden. SLAM steht für Simultaneous Localization and Mapping und beschreibt die gleichzeitige Ermittlung einer konsistenten Karte der Umgebung und die Lokalisation innerhalb dieser Karte. Anstelle einem Bild-zu-Bild-Tracking verwendet der Algorithmus dann das interne Modell der Umgebung, um die Bewegung der Kamera von der letzten Position zu der des aktuell aufgenommenen Bildes zu ermitteln.

[7] R.I.P.

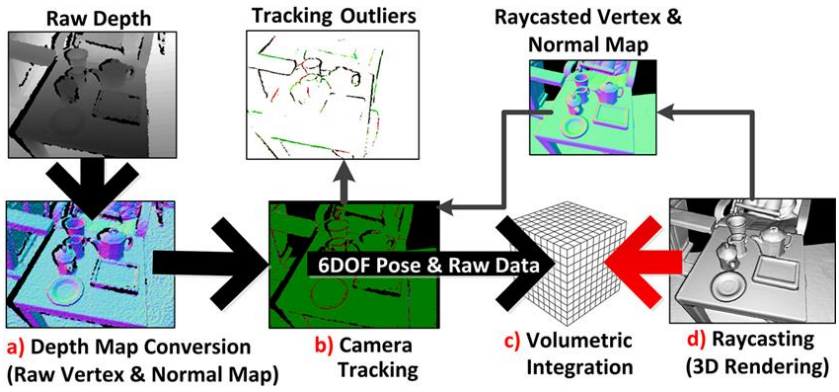


Abb. 2–5: KinectFusion-Pipeline, entnommen aus (Izadi et al. 2011)

Ein bekannter Algorithmus, der auf dem Prinzip des Bild-zu-Modell-Trackings beruht, ist KinectFusion, vorgestellt von (Newcombe et al. 2011; Izadi et al. 2011). Abb. 2–5 zeigt die Verarbeitungspipeline. Der Algorithmus erstellt im Laufe der Zeit eine interne Repräsentation der Umgebung auf Basis einer abgeschnittenen, vorzeichenbehafteten Abstandsfunktion (engl.: truncated signed distance function, kurz TSDF), die in einem dreidimensionalen Voxelraster gespeichert wird. Der Nullübergang der TSDF zeigt dabei die Position der Oberfläche an (siehe Abb. 2–6 für eine Visualisierung). Aus neu aufgenommenen Bildern wird eine dreidimensionale Punktwolke erzeugt und mit einer aus der internen TSDF-Repräsentation generierten Punktwolke korreliert.

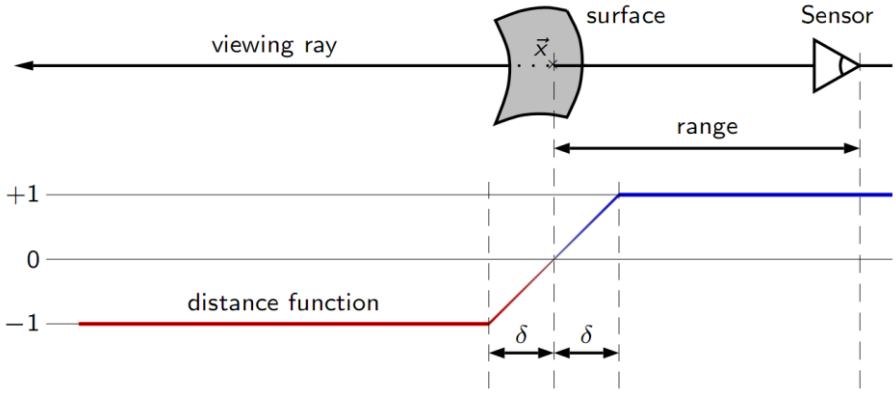


Abb. 2–6: TSDF-Visualisierung, entnommen aus (Korcz 2011)

Ein großer Vorteil von KinectFusion und Algorithmen mit einem ähnlichen Ansatz ist die automatische Generierung einer 3D-Repräsentation der Umgebung zusammen mit der Lageschätzungsaufgabe. Aus der TSDF-Voxel-Repräsentation kann ohne großen Aufwand ein dreidimensionales Mesh generiert werden. Nachteilig ist der hohe Speicherverbrauch und der verhältnismäßig große Rechenaufwand. Durch aktuelle Entwicklungen im Bereich der Allzweck-Berechnung auf Grafikprozessoreinheiten (engl.: General Purpose Computation on Graphics Processing Unit, kurz GPGPU) wie CUDA[8] (Compute Unified Device Architecture) und OpenCL[9] (Open Computing Language) ist jedoch auch hier eine Echtzeitverarbeitung möglich.

[8] http://www.nvidia.de/object/cuda-parallel-computing-de.html
[9] https://www.khronos.org/opencl/

Der von KinectFusion gewählte Ansatz eliminiert den Drift allerdings nicht vollständig. Dafür sind weitergehende Funktionalitäten notwendig, die eine globale Optimierung der Umgebungskarte vornehmen und beispielsweise durch Schleifenerkennung (engl.: loop closure) vorhergehende Lageschätzungen in einer Graphrepräsentation korrigieren. Algorithmen aus dieser Kategorie, die jedoch keine Darstellung der Umgebung erzeugen, nennen sich Pose-SLAM-Algorithmen. Ein aktueller Ansatz für dichte Odometrie (engl.: dense odometry), der sowohl die photometrische als auch die geometrische Abweichung betrachtet und auf dem Pose-SLAM-Konzept basierende Funktionen wie Schleifenerkennung integriert, wurde veröffentlicht von (Kerl et al. 2013a). Er ist auf einer handelsüblichen Desktop-CPU (Central Processing Unit) in Echtzeit lauffähig.

Wird ein eben solcher Algorithmus zur Lageschätzung eingesetzt, der auf dem Pose-SLAM-Konzept beruht, wird zusätzlich ein Algorithmus zur 3D-Rekonstruktion benötigt, um eine digitale Darstellung der Umgebung zu generieren. Erste Ansätze beruhten auch dabei auf dem Konzept von Merkmalspunkten (Henry et al. 2012), wodurch die generierten Darstellungen nur aus dünn besetzten Punktwolken bestanden. Durch Weiterverarbeitung dieser Punktwolken kann versucht werden, aus diesen diesen eine geschlossene 3D-Darstellung zu erzeugen; die Ergebnisse sind jedoch Ansätzen, die die gesamte Bildinformation verwenden, visuell unterlegen.

Der im KinectFusion-Algorithmus gewählte Ansatz einer TSDF-Voxelgitter-basierten Umgebungsrepräsentation bietet sich auch als Basis für einen eigenständigen Algorithmus zur dichten 3D-Konstruktion an. Dem Problem des hohen Speicherverbrauchs und der verhältnismäßig langsamen Verarbeitungs-geschwindigkeit kann durch den intelligenten Einsatz erweiterter Datenstrukturen wie Octrees begegnet werden (Zeng et al. 2013). Ein aktueller Ansatz, der auf diesem Prinzip beruht, ist in Echtzeit auf einer handelsüblichen Desktop-CPU lauffähig (Steinbrücker et al. 2014). Zur Erzeugung der 3D-Darstellung aus der TSDF-Voxel-Repräsentation wird hier ein Marching-Cubes-Meshing-Algorithmus eingesetzt.

3 Aufgabenstellung

3.1 Szenariobeschreibung

Das allgemeine Szenario für den Einsatz der Algorithmen ist die Annäherung eines Verfolger-Satelliten (im Folgenden als Chaser bezeichnet) an ein unkooperatives Zielobjekt im Orbit (ein ausgedienter Satellit oder ein Weltraummüllteil, im Folgenden als Target bezeichnet) im Rahmen einer Servicing- oder Deorbiting-Mission. Der Chaser befindet sich dabei in solch einer Entfernung und Lage relativ zum Target, dass sich dieses vollständig und großflächig im Sichtfeld einer RGB-D-Kamera befindet. Chaser und Target bewegen sich relativ zueinander und relativ zum Hintergrund. Das Target dreht sich gleichmäßig um eine Achse. Es wird angenommen, dass die Form des Targets unbekannt ist.

Ein solches Szenario wurde im RACOON-Simulator zur Validierung der Algorithmen nachgestellt. Abb. 3–1 zeigt den Aufbau des Simulators während einer Testkampagne. Die geometrischen Abmaße des Simulators sind um den Faktor vier zum realen Szenario herunterskaliert. Als Target kommt ein Satelliten-Mockup mit Abmaßen von ca. 60 cm x 70 cm x 100 cm zum Einsatz, dessen äußere Form und Oberflächenbeschaffenheit einem generischen GEO-Satelliten entspricht. Das Target besitzt fünf Rotationsfreiheitsgrade, dreht im gegeben Referenzszenario jedoch lediglich mit 2 °/s um seine z-Achse. In zwei Metern Entfernung dazu befindet sich der Chaser, auf dem die RGB-D-Kamera und der Sensor-PC montiert sind. Der Chaser besitzt drei Translations- und drei Rotationsfreiheitsgrade, bleibt im gegebenen Szenario jedoch fest in Position, so dass die RGB-D-Kamera mittig auf das Target ausgerichtet ist. Die Szene wird von einer künstlichen Sonne und einem Erdalbedo-Scheinwerfer beleuchtet, um realistische Lichtverhältnisse nachzustellen.

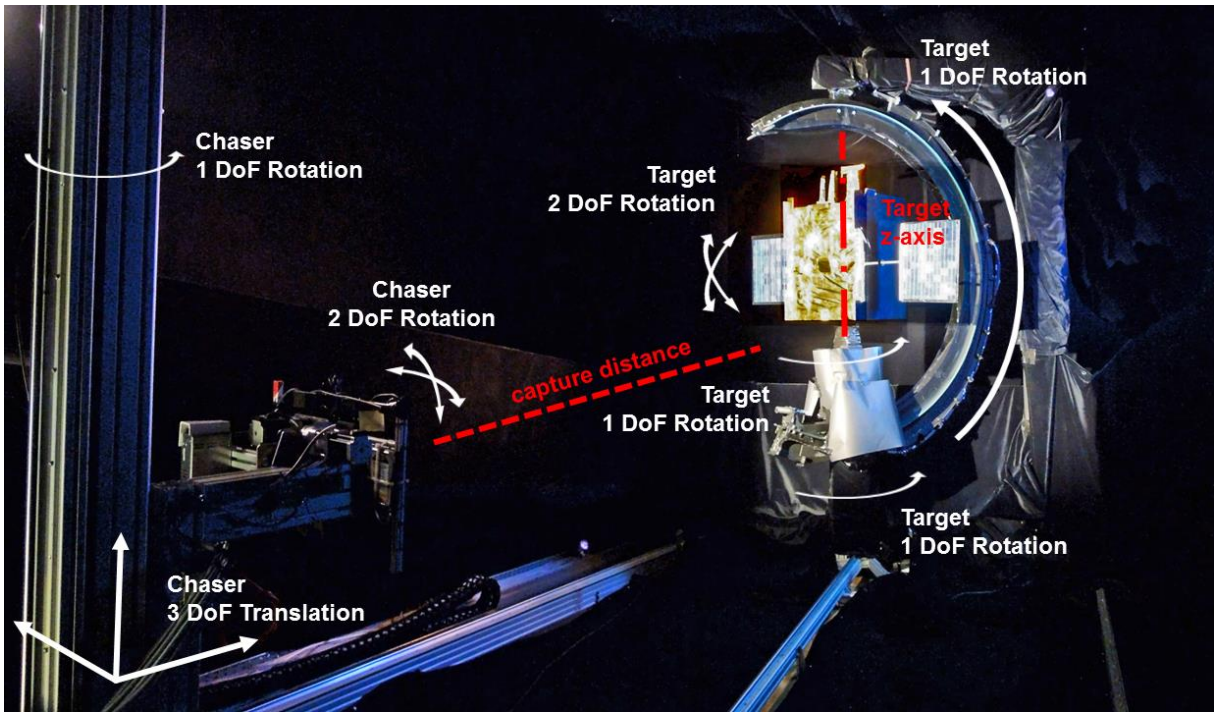


Abb. 3–1: Aufbau des RACOON-Simulators, entnommen aus (Dziura et al. 2017)

3.2 Ziele der Arbeit

Das übergeordnete Ziel der Arbeit ist die Untersuchung der Eignung von RGB-D-Kameras für Teleoperationsaufgaben im Rahmen der Nahbereichsnavigation beim OOS. Die Arbeit soll hierzu keine abschließenden Ergebnisse liefern, sondern eine Basis für weitergehende Untersuchungen.

Dazu sollen zwei Bereiche bearbeitet werden:

Zum einen sollen Algorithmen aus dem Bereich der Bewegungsschätzung und 3D-Rekonstruktion von unbekannten Objekten für RGB-D-Kameras ausgewählt werden, die sich für den Einsatz auf der gegebenen Infrastruktur eignen und die ausgehend von den zugehörigen wissenschaftlichen Veröffentlichungen zu den besten Implementierungen bzgl. Laufzeit und Robustheit gehören.

Die Algorithmen sollen zum Einsatz kommen, um die Relativposition zwischen Chaser und Target zu ermitteln und eine 3D-Darstellung des Targets zu erzeugen, die die wahre Form bestmöglich abbildet und mit der das Objekt virtuell vollständig betrachtet werden kann. Sie sollen für die Ausführung auf einer CPU ausgelegt sein, d.h. ohne Beschleunigung mittels der GPU (Graphics Processing Unit) auskommen, da in Weltraumanwendungen derzeit keine dedizierte Grafikhardware zum Einsatz kommt.

Die Lageschätzung und 3D-Rekonstruktion wird im vorgegebenen Szenario durch die sich verändernden Lichtverhältnisse durch (künstliche) Sonne und Erdalbedo erschwert. Die allgemeine Rekonstruktionsgüte und die Robustheit gegenüber den Lichtverhältnissen soll mithilfe von im RACOON-Simulator aufgenommenen Testdatensätzen evaluiert werden.

Zum anderen soll ein Softwareframework zur Evaluierung von Lageschätzungs- und 3D-Rekonstruktionsalgorithmen entstehen, in das die ausgewählten Algorithmen integriert werden sollen. Es soll dazu dienen, den Vergleich verschiedener Algorithmen und Sensortypen zu erleichtern. Dafür muss es flexibel um weitere Funktionen erweiterbar sein. Es soll sowohl die Liverekonstruktion als auch die Verarbeitung voraufgezeichneter Datensätze unterstützen und das Ergebnis, die berechnete virtuelle Darstellung des Objekts, visuell darstellen.

Langfristiges, über diese Arbeit hinausgehendes Ziel ist es, den Kern der Software in die am Lehrstuhl entwickelte Teleoperationsumgebung zu integrieren.

3.3 Vorgehensweise

Das Forschungsgebiet der visuellen Odometrie ist breit gefächert. Es gibt Algorithmen für verschiedenste Einsatzbereiche und Systeme. Um einen Überblick über den aktuellen Stand der Forschung zu gewinnen und mögliche Kandidaten für den Einsatz im gegebenen Szenario auszuwählen, war als Vorarbeit eine Literaturrecherche und eine Einarbeitung in die grundlegende Funktionsweise der Algorithmen notwendig. Durch das gewonnene Wissen konnte eine Vorauswahl basierend auf den gegebenen Anforderungen getroffen werden. Um die grundsätzliche Einsatzfähigkeit zu verifizieren und den Einfluss von Parametern zu verstehen, wurden die Algorithmen vorab eigenständig auf der Laborhardware getestet. Anschließend wurden die entsprechenden Einsprungpunkte im Quellcode erfasst, um eine Integration in das zu entwickelnde Softwareframework zu ermöglichen.

Aus den gegebenen Anforderungen an das Framework wurde eine grobe Programmstruktur abgeleitet und notwendige Funktionalitäten festgelegt. Gleichzeitig wurden zu verwendende Bibliotheken ausgewählt um die Entwicklung zu vereinfachen und Schnittstellen und Datentypen zu standardisieren. Das Benutzeroberflächenkonzept entstand dynamisch während der Entwicklungsphase.

Eine erste Version der Software mit grundlegender Funktionalität wurde entwickelt und im Laboreinsatz getestet, woraus sich wünschenswerte Zusatzfunktionalitäten und strukturelle Überarbeitungsnotwendigkeiten ergaben. Nach einer weiteren Entwicklungsphase wurden mithilfe der fertigen Software Labordatensätze zum vorgegebenen Referenzszenario aufgenommen und mit den ausgewählten und in das Framework integrierten Algorithmen verarbeitet. Abschließend wurden die Ergebnisse ausgewertet und die eingangs formulierten Anforderungen verifiziert.

3.4 Abgrenzung

Forschung im Bereich der visuellen Odometrie ist auf terrestrische Anwendungen fokussiert. Die meisten Algorithmen sind für den Einsatz in Innenraumanwendungen (Endres et al. 2012; Henry et al. 2012; Bylow et al. 2013) und auf bodengebundenen Systemen wie mobilen Robotern (Hirschmüller et al. 2015; Endres et al. 2014) gedacht, die sich in ihrer Umgebung zurechtfinden sollen. Es existieren auch einige Ansätze für fliegende Systeme in Bodennähe wie bspw. Quadrokopter (Huang et al. 2011; Scherer und Zell 2013; Cavrois et al. 2015). Der Einsatz in einer Weltraumumgebung mit entsprechend speziellen Anforderungen und Umgebungsbedingungen ist ein noch junges Feld. Zwar gibt es bereits Studien zum Einsatz von Stereokamerasystemen (Dionnet und Marchand 2007; Schnitzer et al. 2012), die Verwendung von RGB-D-Sensoren in diesem Zusammenhang ist jedoch neu.

In dieser Arbeit wird kein umfassender Vergleich der Performance einer Vielzahl von Algorithmen durchgeführt. Es wird lediglich eine qualitative, keine quantitative Analyse der Rekonstruktionsqualität durchgeführt. Die Algorithmen wurden basierend auf den Angaben in den zugehörigen Veröffentlichungen ausgewählt. Verwendet wurden die von den Autoren veröffentlichten Implementierungen der zugehörigen Veröffentlichungen. Sie wurden nicht auf Basis von Veröffentlichungen selbst nachimplementiert.

Um verschiedene Algorithmen und Sensortypen vergleichen zu können, ist es sinnvoll, diese in ein Softwareframework einzubetten, um Abläufe und Schnittstellen zu standardisieren, sowie die Datenaufzeichnung und -ausgabe zu vereinheitlichen. Existierende allgemeine und umfangreiche Frameworks wie ROS benötigen eine lange Einarbeitungszeit zur Verwendung und Erweiterung. Das in dieser Arbeit entwickelte Softwareframework ist speziell auf die vorgegebene Anwendung zugeschnitten. Es ist ohne langwierige Konfigurationsphase einsetzbar und kann leicht um zusätzliche Komponenten erweitert werden. Die Implementierung ist nicht auf absolute Effizienz und Ausführungsgeschwindigkeit ausgelegt. Das Hauptaugenmerk liegt auf der Flexibilität für künftige Weiterentwicklungen. Die entstandene 3D-Darstellung wird nicht weiterverarbeitet. Unter der gegebenen Annahme eines unbekannten Objekts ist kein Fitting von 3D-CAD-Modellen o.ä. möglich.

Die verwendete Hardware ist in keiner Weise weltraumqualifiziert. Die Software ist auf den Einsatz auf der Simulatorhardware ausgerichtet.

4 Algorithmenauswahl

Die Algorithmen, die für diese Arbeit ausgewählt werden sollen, müssen zwei Aufgaben erfüllen: Zum einen die Schätzung der relativen Lage zwischen verfolgtem und verfolgendem Objekt (Tracking) und zum anderen die 3D-Rekonstruktion des verfolgten Objekts (Mapping). Als Eingabe stehen dabei lediglich die Daten einer RGB-D-Kamera zur Verfügung – pro Zeitschritt ein Farb- und ein Tiefenbild der Umgebung.

Zuerst werden in diesem Kapitel die spezifischen Anforderungen an die Algorithmen und der Auswahlprozess beschrieben. Anschließend wird auf die ausgewählten Algorithmen und deren theoretische Funktionsweise näher eingegangen.

4.1 Anforderungen

Die spezifischen Anforderungen, die sich aus den in Abschnitt 3.2 definierten Zielen und der Entwicklungs- und Simulationsumgebung ableiten lassen, sind im Folgenden aufgelistet:

Tab. 4–1: Anforderungen an die Algorithmenauswahl

ID	Anforderung
1-01	Die Algorithmen sollen für die Ausführung auf einer x86-CPU ausgelegt sein.
1-02	Die Algorithmen dürfen nicht an spezielle Sensorhardware gebunden sein, d.h. sie dürfen nicht nur mit einem speziellen Kameratyp funktionieren.
1-03	Die Algorithmen müssen öffentlich verfügbar als Programmbibliothek oder als Quellcode vorliegen, der über entsprechende Schnittstellen aufgerufen werden kann, und sie müssen ggf. aus Programmpaketen mit erweiterten Funktionalitäten heraustrennbar sein.
1-04	Die Algorithmen müssen für den Einsatz unter Windows portierbar sein, da die Entwicklungsarbeit im Projekt RACOON unter Windows erfolgt.
1-05	Die Algorithmen im Bereich der Lageschätzung müssen als Eingabe ein Tiefenbild, ein Tiefen- und ein Farbbild oder eine Punktwolke (generiert aus Tiefen- und Farbbild) akzeptieren.
1-06	Die Algorithmen im Bereich der Lageschätzung müssen als Ausgabe eine Schätzung der aktuellen Kameralage relativ zur Initialposition generieren.
1-07	Die Algorithmen im Bereich der 3D-Rekonstruktion müssen als Eingabe ein Tiefen- und ein Farbbild oder eine Punktwolke (generiert aus Tiefen- und Farbbild) und eine Schätzung der aktuellen Kameralage relativ zur Initialposition akzeptieren.
1-08	Die Algorithmen im Bereich der 3D-Rekonstruktion müssen als Ausgabe eine dichte Punktwolke oder ein 3D-Mesh des verfolgten Objekts generieren.

1-09	Die Algorithmen sollen auf einer aktuellen Desktop-CPU eine Echtzeitverarbeitung der Eingabedaten ermöglichen.
1-10	Die Algorithmen sollen eine Objektrekonstruktion bei einer vorgegebenen Zielobjektdrehrate von 2 °/s in einem Abstand von 2 m ermöglichen.
1-11	Die Algorithmen sollen robust gegenüber (simulierten) Orbitlicht-verhältnissen sein.

4.2 Auswahl

4.2.1 Lageschätzung

Im Bereich der Objekt-Lageschätzung (engl.: pose estimation) mit RGB-D-Kameras gibt es generell zwei Ansätze:

Der allgemeinere Ansatz ist der dedizierter Objekt-Tracking-Algorithmen (Choi und Christensen 2013; Klodt et al. 2013; Yuheng Ren et al. 2013; Krull et al. 2014), die vor einem beliebigen Hintergrund eine oder mehrere Instanzen eines Objektes beliebiger Größe erkennen und verfolgen – häufig, aber nicht immer, basierend auf einem vorher angelegten Modell des Objekts. Von Nachteil ist dabei die geringe Laufzeitperformance dieser Algorithmen, da der gesamte Bildausschnitt nach dem Objekt abgesucht und die Bewegung des Objekts von der des Hintergrunds getrennt werden muss.

Der andere Ansatz ist die Verwendung von Algorithmen aus dem Bereich der Bewegungsschätzung, die für die Ermittlung der Lage einer mobilen Kamera in einer statischen Umgebung ausgelegt sind (visuelle Odometrie, Scaramuzza und Fraundorfer 2011). Dabei wird das gesamte Sichtfeld für die Ermittlung der Kamerabewegung verwendet. Ist das zu verfolgende Objekt groß im Verhältnis zum Hintergrund, kann mit diesen Algorithmen die relative Bewegung zwischen Kamera und Objekt ermittelt werden, ohne das Objekt selbst getrennt vom Hintergrund erfassen zu müssen. Das Objekt selbst kann damit als die statische Umgebung angesehen werden. Der Berechnungsaufwand ist bei diesem Ansatz deutlich geringer, die Einsatzmöglichkeit aber auf Szenarios mit den genannten Eigenschaften beschränkt.

Das vorgegebene Szenario dieser Arbeit entspricht den Anforderungen für den zweiten Ansatz. Chaser und Target bewegen sich relativ zueinander vor einem weit entfernten Hintergrund, der nicht mehr von der Tiefenkamera erfasst wird. Im Vordergrund befindet sich ein einziges Objekt unbekannter Form. Damit bieten sich Algorithmen des zweiten Ansatzes an.

Im praktischen Fall der Analyse im RACOON-Simulator bewegen sich die Objekte allerdings relativ zum nahen Laborhintergrund. Daher muss in der späteren Umsetzung eine geeignete Lösung zur Hintergrundunterdrückung gefunden werden (siehe Abschnitt 5.5.2.3).

Im Bereich der Algorithmen für visuelle Odometrie gibt es wiederum zwei verschiedene grundlegende Ansätze – Sparse-Tracking- und Dense-Tracking-Algorithmen. Sparse-Tracking-Algorithmen (bspw. Klein und Murray 2007) basieren auf der Extraktion von

Merkmalspunkten aus dem Bild, die anschließend für das Tracking zwischen aufeinanderfolgenden Bildern verwendet werden. Dense-Tracking-Algorithmen verwenden die gesamte Bildinformation. Letztere liefern generell die robusteren Ergebnisse mit weniger Drift, müssen jedoch eine erheblich größere Anzahl an Bildpunkten verarbeiten. Die meisten dieser Algorithmen bedienen sich Techniken des GPU-Computings, um echtzeitnahe Verarbeitungsgeschwindigkeiten zu erreichen (Newcombe et al. 2011; Whelan et al. 2013). Neueste Entwicklungen sind jedoch bereits in der Lage, die Berechnungen in Echtzeit auf einer Desktop-CPU auszuführen (Jaimez und González-Jiménez 2015; Kerl et al. 2013a). Deshalb werden für diese Arbeit nur Dense-Tracking-Algorithmen in Betracht gezogen.

Im Bereich der Dense-Tracking-Algorithmen wurde eine Literaturrecherche durchgeführt, um die aktuell in Bezug auf Laufzeit und Robustheit besten Algorithmen aus dem Forschungsbereich zu ermitteln. Besonders einschränkend war dabei Anforderung 1-03 (siehe Tab. 4–1)Tab. 4–1: Anforderungen an die Algorithmenauswahl zur öffentlichen Verfügbarkeit einer Implementierung. Basierend auf dieser und den weiteren in Tab. 4–1 beschriebenen Anforderungen wurden mit DIFODO (Differential Odometry, Jaimez und González-Jiménez 2015) und dvo_slam (Dense Visual Odometry and SLAM, Kerl et al. 2013a) zwei Algorithmen ausgewählt, die in den Abschnitten 4.3 und 4.4 näher beschrieben werden, und die für die Integration in das Softwareframework vorgesehen wurden (siehe Kapitel 5, Abschnitt 5.5.3.1 und 5.5.3.2).

4.2.2 3D-Rekonstruktion

Im Bereich der dichten 3D-Rekonstruktion muss der ausgewählte Algorithmus in der Lage sein, aus den gegebenen Tiefen- und Farbbildern und den zugehörigen Lageschätzungsdaten ein eine dreidimensionale Darstellung des Objekts zu erzeugen. Ein simpler Ansatz dazu wäre, die für jeden Datensatz aus Tiefen- und Farbbildern generierten Punktwolken in eine Gesamtdarstellung zusammenzufassen und anschließend in ein Mesh zu konvertieren. Dabei würden jedoch die Anzahl der Punkte und somit auch der Speicherverbrauch kontinuierlich mit der Zeit steigen und über alle Grenzen wachsen.

Bessere Ansätze basieren auf der Verwendung von Oberflächenflicken (engl.: surfels) oder volumetrischen Darstellungen, bei denen die Umgebung in ein Voxelraster unterteilt wird (siehe rechte Spalte von Abb. 4–4). Die Voxel können dabei entweder speichern, ob sie Teil des belegten oder freien Raums sind, oder als abgeschnittene, vorzeichenbehaftete Abstandsfunktion (TSDF) die Entfernung bis zur Objektoberfläche angeben (Osher und Fedkiw 2003).

Eine Literaturrecherche hat mit fastfusion (Steinbrücker et al. 2014) einen Algorithmus dieses Typs hervorgebracht, für den eine Implementierung öffentlich zur Verfügung steht, der die Möglichkeit bietet, ein 3D-Mesh aus der internen Darstellung zu extrahieren, und der die sonstigen Anforderungen aus Abschnitt 4.1 erfüllt.

Der Algorithmus wird in Abschnitt 4.5 näher beschrieben und wurde in das Softwareframework integriert (siehe Kapitel 5 und Abschnitt 5.5.3.3).

4.3 DIFODO Tracker

4.3.1 Basisinformationen

DIFODO ist ein Algorithmus für visuelle Odometrie, der von (Jaimez und González-Jiménez 2015) veröffentlicht wurde. Er verwendet lediglich die Tiefenbilder einer RGB-D-Kamera, um deren Lage zu schätzen. Die zugehörige Veröffentlichung verspricht eine Geschwindigkeit von bis zu 60 Hz auf einem einzelnen Kern einer aktuellen Desktop-CPU. Der Algorithmus verfügt über keine erweiterten Funktionalitäten, um den Drift der Lageschätzung über die Zeit zu minimieren, wie beispielsweise Graphoptimierung oder Schleifenerkennung.

DIFODO wurde als Teil des Mobile Robot Programming Toolkit (MRPT)[10] veröffentlicht – einer Sammlung von Programmbibliotheken aus den Bereichen SLAM, maschinellen Sehens und Bewegungsplanung. MRPT kann sowohl unter Linux als auch unter Windows kompiliert werden und DIFODO kann als eigenständige Bibliothek verwendet werden. Dazu muss lediglich die zugehörige Basisklasse `mrpt::vision::CDifodo` als Elternklasse für die eigene Implementierung verwendet werden und deren abstrakte Methode `void loadFrame()` implementiert werden. Für Details zur Integration von DIFODO siehe Kapitel 5, Abschnitt 5.5.3.1.

4.3.2 Mathematische Grundlagen

Die im Weiteren ausgeführten Herleitungen sind aus (Jaimez und González-Jiménez 2015) entnommen.

Der Algorithmus basiert im Kern auf der sogenannten Entfernungsflussbedingung (engl.: range flow constraint, (Spies und Barron 2002). Die Bedingung ist das Tiefenbild-Äquivalent zur Helligkeitsänderungsbedingung des optischen Flusses für Farbbilder (Horn und Schunck 1980).

$z: (\Omega \subset \mathbb{N}^2) \rightarrow \mathbb{R}$ beschreibe ein durch eine 3D-Tiefenkamera aufgenommenes Tiefenbild, also eine Abbildung zwischen Pixelkoordinaten $(u, v) \in \Omega$ und einem Tiefenwert z. Dann lautet die Entfernungsflussbedingung durch Entwicklung der Tailorreihe von z nach u und v und Ableitung nach der Zeit t:

$$\dot{z} = \frac{\partial z}{\partial t} + \frac{\partial z}{\partial u}\dot{u} + \frac{\partial z}{\partial v}\dot{v} + O(\Delta t, \dot{u}, \dot{v}) \qquad \textbf{(4–1)}$$

und damit

$$\dot{z} \simeq z_t + z_u\dot{u} + z_v\dot{v} \qquad \textbf{(4–2)}$$

bei Vernachlässigung der Terme höherer Ordnung. Diese Vernachlässigung ist zulässig unter der Annahme, dass die Bewegung zwischen aufeinanderfolgenden Bildern klein ist und damit als linear angenommen werden kann und dass die betrachteten Punkte zu lokalen planaren Flicken gehören.

Aus dieser Gleichung kann nun die Geschwindigkeit der Kamera berechnet werden. Dazu müssen die Ableitungen $\dot{z}$, $\dot{u}$ und $\dot{v}$ durch Terme für die Translations- und Rotationsgeschwindigkeit der Kamera ausgedrückt werden. Abb. 4–1 zeigt die Zusammenhänge zwischen den verschiedenen Koordinaten. Die partiellen

[10] http://www.mrpt.org/

Ableitungen von z nach der Zeit und den beiden Pixelkoordinaten können direkt aus den Tiefenbildern als diskrete Näherung berechnet werden.

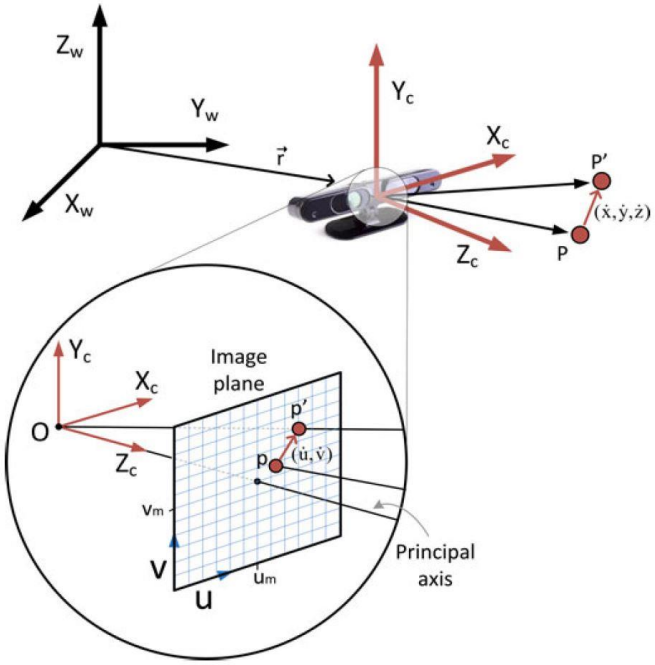


Abb. 4–1: Geometrische Zusammenhänge, entnommen aus (Jaimez und González-Jiménez 2015)

Über das Lochkameramodell (Forsyth und Ponce 2002) ergibt sich ein Zusammenhang zwischen den Pixelkoordinaten (u, v) und den Koordinaten eines zugehörigen Punktes $\boldsymbol{p} = (x, y, z)^T$ im Referenzsystem der Kamera:

$$u = f_x \frac{x}{z} + c_x \qquad \textbf{(4–3)}$$

$$v = f_y \frac{y}{z} + c_y \qquad \textbf{(4–4)}$$

(c_x, c_y) beschreibt hier die Koordinaten des Mittelpunkts des Bildes (Hauptpunkt), f_x und f_y sind die Brennweiten, jeweils in Pixeln.

Für die Ableitungen ergibt sich dadurch:

$$\dot{u} = f_x \left(\frac{\dot{x}z - \dot{z}x}{z^2}\right) \qquad \textbf{(4–5)}$$

und

$$\dot{v} = f_y \left(\frac{\dot{y}z - \dot{z}y}{z^2}\right) \qquad \textbf{(4–6)}$$

Die Punktgeschwindigkeiten $\dot{x}, \dot{y}, \dot{z}$ können nun durch die Translationsgeschwindigkeiten $\boldsymbol{\xi} = (\xi_x, \xi_y, \xi_z)^T$ und die Rotationsgeschwindigkeiten $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)^T$ der Kamera ausgedrückt werden.

Unter der Annahme einer statischen, nicht-deformierbaren Umgebung ist die Geschwindigkeit eines betrachteten Punktes im globalen Koordinatensystem gleich Null. Mithilfe der Eulerableitung ergibt sich damit:

$$\mathbf{0} = \boldsymbol{\xi} + \dot{\boldsymbol{p}} + \boldsymbol{\omega} \times \boldsymbol{p} \qquad \textbf{(4–7)}$$

und umgestellt und ausgeschrieben

$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{z}\end{pmatrix} = \begin{pmatrix}-\xi_x - z\omega_y + y\omega_z\\ -\xi_y + z\omega_x - x\omega_z\\ -\xi_z - y\omega_x + x\omega_y\end{pmatrix} \qquad \textbf{(4–8)}$$

Mit **(4–8)** eingesetzt in **(4–5)** und **(4–6)** und dieses wiederum eingesetzt in **(4–2)** ergibt sich damit:

$$-z_t = \left(1 + \frac{xf_x}{z^2}z_u + \frac{yf_y}{z^2}z_v\right)\left(\xi_z + y\omega_x - x\omega_y\right) + \frac{f_x}{z}z_u\left(-\xi_x + y\omega_z - z\omega_y\right) + \frac{f_y}{z}z_v\left(-\xi_y - x\omega_z + z\omega_x\right) \qquad \textbf{(4–9)}$$

Gleichung **(4–9)** ist linear in Bezug auf die sechs Kamerageschwindigkeitsterme. Damit kann ein lineares Gleichungssystem aufgestellt und nach der Kamerageschwindigkeit aufgelöst werden, wenn mindestens sechs linear unabhängige Gleichungen für sechs Punkte im Bild vorhanden sind. In der Praxis werden jedoch deutlich mehr Punkte (N) benötigt, um die Lösung robust gegenüber Rauschen und Näherungsfehlern zu machen. Das führt zu einem überbestimmten System, dessen Lösung über die Methode der gewichteten kleinsten Quadrate (engl.: weighted least-squares) mithilfe der Moore-Penrose-Pseudoinverse (Moore 1920; Penrose 1955) gefunden werden kann:

$$WA\begin{pmatrix}\boldsymbol{\xi}\\ \boldsymbol{\omega}\end{pmatrix} = -WB \;\Rightarrow\; \begin{pmatrix}\boldsymbol{\xi}\\ \boldsymbol{\omega}\end{pmatrix} \approx -(A^TWA)^{-1}A^TWB \qquad \textbf{(4–10)}$$

$A \in \mathbb{R}^{N\times 6}$ beinhaltet hier die Koeffizienten in **(4–9)** der sechs Terme aus $\boldsymbol{\xi}$ und $\boldsymbol{\omega}$. $B \in \mathbb{R}^{N\times 1}$ beinhaltet die zeitlichen Ableitungen der Tiefenwerte pro zu einem Punkt gehörendem Pixel. $W \in \mathbb{R}^{N\times 6}$ ist eine diagonale Gewichtungsmatrix, die Ungenauigkeiten durch Sensorrauschen und Linearisierungsfehler modelliert.

Um dem Problem zu begegnen, dass Bewegungen zwischen aufeinanderfolgenden Bildern möglichst klein sein müssen, um dem linearen Ansatz zu genügen, verwendet der Algorithmus eine Pyramidenberechnung mit einer Gaußfilterung der Eingabebilder auf mehreren Ebenen. Auf höheren Ebenen, d.h. einem stärker gefilterten Bild niedrigerer Auflösung, dürfen die Bewegungen bei gleichbleibender Näherungsgüte größer ausfallen. Die Kamerageschwindigkeit wird auf jeder Ebene, von grob nach fein, geschätzt und dazu verwendet, um das Bild auf der darunterliegen Ebene im 3D-Raum mit den aktuell geschätzten Werten zu transformieren und zurück zu projizieren. Dadurch entstehen Zwischenbilder, die deutlich geringere Bewegungen als die Ursprungsbilder aufweisen.

Aus den Lösungen für die Translations- und Rotationsgeschwindigkeitswerte der Kamera in jedem Zeitschritt ergibt sich abschließend durch numerische Integration und Transformation in das globale Koordinatensystem die Gesamtschätzung der Lage der Kamera.

Für tiefergehende mathematische Feinheiten, Implementierungsdetails und Benchmarkergebnisse des Algorithmus sei an dieser Stelle wiederum auf die Veröffentlichung (Jaimez und González-Jiménez 2015) verwiesen.

4.4 dvo_slam Tracker

4.4.1 Basisinformationen

dvo_slam (Kerl et al. 2013a) ist ein Algorithmus zur Lageschätzung für RGB-D-Kameras, der neben den Tiefenwerten auch die Farbinformationen verwendet, was Vorteile bei Szenen mit wenig vorhandener geometrischer Struktur bietet. Er berechnet die Lageschätzung nicht nur aus dem aktuellen und vorhergehenden Bild, sondern setzt erweiterte Strategien wie Schlüsselbildtracking (engl.: key frame tracking), Lagegraphoptimierung (engl.: pose graph optimization) und Schleifenerkennung (engl.: loop closure) ein, um Driftfehler zu eliminieren. Der Algorithmus geht damit über die reine visuelle Odometrie hinaus und fällt in den Bereich der SLAM-Algorithmen.

Die reine Schlüsselbildtrackingkomponente des Algorithmus läuft auf einer handelsüblichen Desktop-CPU mit 30 Hz auf einem Kern (Kerl et al. 2013b). Die weitergehenden SLAM-Funktionalitäten zur Driftreduzierung benötigen deutlich mehr Rechenzeit. Nach Ende der Verarbeitung aller Datensätze wird ein aufwändiger abschließender Lagegraphoptimierungsschritt durchgeführt, der im Echtzeiteinsatz nicht verfügbar ist.

Die Entwicklung von dvo_slam basiert auf mehreren aufeinander aufbauenden Veröffentlichungen, die bis ins Jahr 2011 zurückreichen (Steinbrücker et al. 2011; Kerl et al. 2013b). Der Algorithmus ist als Softwarepaket für das Robot Operating System (ROS) auf GitHub zum Download verfügbar[11]. Durch die starke Einbindung in die Struktur von ROS ist eine unabhängige Verwendung jedoch nicht ohne Weiteres möglich.

4.4.2 Mathematische Grundlagen

Die im Weiteren ausgeführten Herleitungen sind aus (Kerl et al. 2013a) entnommen.

Zur Bestimmung der Kamerabewegung zwischen zwei Bildern verwendet der Algorithmus einen Ansatz zur Minimierung der photogrammetrischen und der geometrischen Abweichung zwischen diesen (siehe Abb. 4–2).

Dabei kommt erneut das Lochkameramodell (Hartley und Zisserman 2004) zum Einsatz, welches einen Punkt $\boldsymbol{p} = (x, y, z)^T$ in Kamerakoordinaten auf einen Punkt $\boldsymbol{q} = (u, v)^T$ in Pixelkoordinaten abbildet mit der Projektionsfunktion

$$\pi(\boldsymbol{p}) = \boldsymbol{q} = \left(f_x \frac{x}{z} + c_x, f_y \frac{y}{z} + c_y\right)^T \qquad \textbf{(4–11)}$$

und der Umkehrfunktion

$$\pi^{-1}(\boldsymbol{q}, Z(\boldsymbol{q})) = \boldsymbol{p} = \left(\frac{u - c_x}{f_x} z, \frac{v - c_y}{f_y} z, z\right)^T \qquad \textbf{(4–12)}$$

wobei (c_x, c_y) die Koordinaten des Mittelpunkts des Bildes (Hauptpunkt) und f_x und f_y die Brennweiten sind, jeweils in Pixeln. $Z(\boldsymbol{q})$ ist der zum Pixel $\boldsymbol{q}$ gehörige Tiefenwert.

Insgesamt wird eine Transformationsmatrix $\boldsymbol{T}$ ermittelt, die das eine auf das andere Bild möglichst gut abbildet.

[11] https://github.com/tum-vision/dvo_slam/

Zu $\boldsymbol{T}$ kann eine Transformationsfunktion τ definiert werden, die den Pixel $\boldsymbol{q}_1$ aus dem ersten Bild auf den Pixel $\boldsymbol{q}_2$ im zweiten Bild abbildet (Erweiterungen von Vektoren auf homogene Koordinaten sind impliziert):

$$\boldsymbol{q}_2 = \tau(\boldsymbol{q}_1, \boldsymbol{T}) = \pi\big(\boldsymbol{T}\pi^{-1}(\boldsymbol{q}_1, Z_1(\boldsymbol{q}_1))\big) \qquad \textbf{(4–13)}$$

Die photogrammetrische Abweichung ist dann für den Intensitätswert $I(\boldsymbol{q})$ eines Pixels definiert als

$$r_I = I_2\big(\tau(\boldsymbol{q}_1, \boldsymbol{T})\big) - I_1(\boldsymbol{q}_1) \qquad \textbf{(4–14)}$$

und die geometrische Abweichung als

$$r_Z = Z_2\big(\tau(\boldsymbol{q}_1, \boldsymbol{T})\big) - \big[\boldsymbol{T}\pi^{-1}\big(\boldsymbol{q}_1, Z_1(\boldsymbol{q}_1)\big)\big]_z \qquad \textbf{(4–15)}$$

wobei $[\cdot]_z$ die dritte Komponente des Punktes zurückgibt. Der zweite Term beschreibt also die Tiefe des zum Pixel des ersten Bildes gehörigen Punktes, transformiert in die neuen Kamerakoordinaten.

$\boldsymbol{T}$ ist eine homogene Transformationsmatrix im Format

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \qquad \textbf{(4–16)}$$

mit zwölf Parametern. Die Kamerabewegung hat jedoch nur sechs Freiheitsgrade. Die Transformationsmatrix $\boldsymbol{T} \in \mathrm{SE}(3)$ kann mittels der Lie-Algebra aus der minimalen Repräsentation $\boldsymbol{\xi} \in \mathrm{se}(3)$ mit sechs Koordinaten über die Matrix-Exponentialfunktion $\boldsymbol{T} = \exp(\hat{\boldsymbol{\xi}})$ erzeugt werden.

Ziel ist nun, die Parameter $\boldsymbol{\xi}$ zu ermitteln, die sowohl die photogrammetrische Abweichung r_I als auch die geometrische Abweichung r_Z möglichst minimieren. Hierfür verwendet der Algorithmus eine probabilistische Formulierung, die es erlaubt, die Verteilung der Abweichungen zu modellieren und ein Bewegungsmodell über eine A-priori-Verteilung zu integrieren. Ohne näher ins Detail zu gehen, ergibt sich daraus eine nichtlineare Fehlerfunktion für $\boldsymbol{\xi}$, die über eine iterative Linearisierung gelöst werden kann (siehe Kerl et al. 2013a).

Um den Drift der Lösung durch Rauschen und Ungenauigkeiten in der Modellierung zu reduzieren, führt der Algorithmus die Bestimmung der Transformationsparameter $\boldsymbol{\xi}$ nicht jeweils zwischen dem aktuellen und dem vorhergehenden Bild aus, sondern zwischen dem aktuellen Bild und einem Schlüsselbild. Das Schlüsselbild wird immer dann durch das aktuelle Bild ersetzt, wenn die Unsicherheit von $\boldsymbol{\xi}$, die sich durch die Kovarianzmatrix Σ_ξ ergibt, einen gewissen Schwellenwert überschreitet.

Außerdem werden die Schlüsselbilder verwendet, um eine Schleifenerkennung durchzuführen. Die Schlüsselbilder werden mit ihren Transformationen in einem Graphen gespeichert, der vorherige mit nachfolgenden Schlüsselbildern verknüpft. Die Kantengewichte sind die Kovarianzmatrizen $\boldsymbol{\Sigma}_{\boldsymbol{\xi}}$ der Unsicherheit zwischen den beiden zugehörigen Transformationen. Kommt die Kamera im Laufe ihrer Bewegung erneut in die Nähe einer Schlüsselbildposition und wird dies durch die Schleifenerkennung festgestellt, wird eine neue Kantenverbindung zwischen den beiden beteiligten Schlüsselbildern in den Graphen eingetragen. Anschließend kann ein Optimierungsalgorithmus für ein nichtlineares Kleinste-Quadrate-Problem die

Kovarianzmatrizen neu berechnen und die Unsicherheiten minimieren. Abb. 4–3 zeigt die Auswirkungen der Optimierungsschritte auf die geschätzte Trajektorie.

Weitere Details zur Theorie und Implementierung finden sich in der zugehörigen Veröffentlichung (Kerl et al. 2013a).

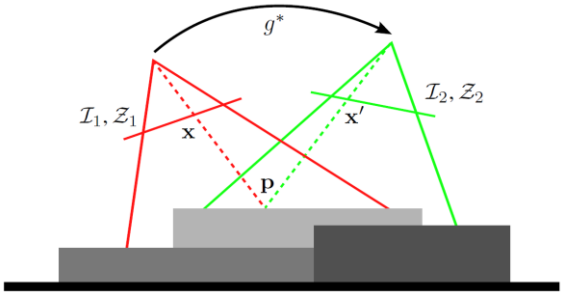


Abb. 4–2: Bewegungsschätzung durch Minimierung der photometrischen und geometrischen Abweichung, entnommen aus (Kerl et al. 2013a)

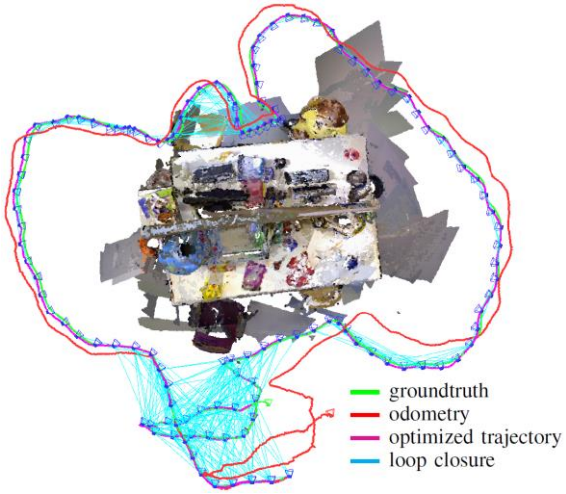


Abb. 4–3: Visuelle Odometrie, optimierte Trajektorie und Schleifenerkennung, entnommen aus (Kerl et al. 2013a)

4.5 fastfusion Mapper

4.5.1 Basisinformationen

fastfusion ist ein Algorithmus zur volumetrischen Umgebungsrekonstruktion für RGB-D-Kameras. Er wurde von (Steinbrücker et al. 2014) veröffentlicht und verspricht Echtzeitperformance auf einer regulären Desktop-CPU. Der Algorithmus generiert eine texturierte Meshdarstellung der Umgebung aus einer abgeschnittenen, vorzeichenbehaftete Abstandsfunktion, die kontinuierlich durch neue RGB-D-Bildpaare aktualisiert wird.

Eine offizielle Implementierung von fastfusion ist auf GitHub zum Download verfügbar[12]. Dort ist der eigentliche Algorithmus in ein Softwarepaket mit Ein- und Ausgabefunktionen und einem Nutzerinterface eingebettet. Er ist jedoch unabhängig davon nutzbar, indem ein Objekt der Klasse `FusionMipMapCPU()` direkt erzeugt wird und deren Methoden `addMap()`, `updateMeshes()` und `getMeshInterleavedMarchingCubes()` aufgerufen werden. Für Details zur Integration von fastfusion siehe Kapitel 5, Abschnitt 5.5.3.3.

4.5.2 Mathematische Grundlagen

Die im Weiteren ausgeführten Herleitungen sind aus (Steinbrücker et al. 2014) entnommen.

[12] https://github.com/tum-vision/fastfusion/

Der Algorithmus basiert auf der internen Repräsentation der Umgebung als vorzeichenbehaftete Abstandsfunktion. Dabei befindet sich die Oberfläche an der Position, an der die Funktion einen Nullübergang aufweist. Zur diskretisierten Speicherung dieser Funktion wird die Umgebung in ein Raster aus Voxeln unterteilt, von denen jeder den Funktionswert an der Position, einen Farbwert und einen Gewichtswert speichert.

Die Aktualisierung der Voxelwerte bei Eingang eines neuen Tiefenbildes geschieht dabei wie folgt:

Sei $\boldsymbol{p}_w \in \mathbb{R}^3$ die Position eines Voxels in Weltkoordinaten. Bei bekannter Translation $T_t \in \mathbb{R}^3$ und Rotation $R_t \in SO(3)$ der Kamera kann die Position in Kamerakoordinaten $\boldsymbol{p}_c = (x, y, z)^T$ wie folgt berechnet werden:

$$\boldsymbol{p}_c = R_t^T(\boldsymbol{p}_w - T_t) \qquad \textbf{(4–17)}$$

Durch Projektion nach dem Lochkameramodell ergeben sich daraus die Pixelkoordinaten $(u, v)^T \in \Omega \subset \mathbb{N}^2$ des Voxels im Kamerabild:

$$\pi(\boldsymbol{p}_c) = \begin{pmatrix} f_x \frac{x}{z} + c_x \\ f_y \frac{y}{z} + c_y \end{pmatrix} \qquad \textbf{(4–18)}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \lfloor \pi(\boldsymbol{p}_c) \rfloor \qquad \textbf{(4–19)}$$

(c_x, c_y) beschreibt hier die Koordinaten des Mittelpunkts des Bildes (Hauptpunkt), f_x und f_y sind die Brennweiten, jeweils in Pixeln.

Im Fall von $z \leq 0$ oder $(u, v)^T \notin \Omega$ liegt die Voxelposition nicht im Blickfeld der Kamera und der Voxel muss nicht aktualisiert werden.

Sei Z_t der (gültige) Tiefenwert an Pixelposition $(u, v)^T$. Dann ergibt sich der Wert der Abstandsfunktion Δ_D für den betrachteten Voxel zu

$$\Delta_D = |\boldsymbol{p}_c| \left(1 - \frac{Z_t}{z}\right) \qquad \textbf{(4–20)}$$

Da zur Erkennung des Nullübergangs der Abstandsfunktion und damit der Position der Oberfläche nur einige Werte um den Nullübergang herum benötigt werden, wird der Wert der Abstandsfunktion zusätzlich beschränkt:

$$\Delta_D^\Phi = \max(\min(\Delta_D, \Phi), -\Phi) \qquad \textbf{(4–21)}$$

Je kleiner der Beschränkungswert Φ, umso feinere Strukturen lassen sich abbilden, aber desto größer ist die Wahrscheinlichkeit, dass Oberflächen durch Rauschen oder Lageschätzungsfehler mehrfach auftauchen.

Im Voxel gespeichert wird nicht direkt der aktuell berechnete Wert der Abstandsfunktion, sondern ein gewichteter Durchschnittswert $D(\boldsymbol{p}_c, t)$ aus dem aktuell berechneten und dem bisher gespeicherten Wert. Der Farbwert des Voxels $C(\boldsymbol{p}_c, t)$ wird auf analoge Weise mit dem aktuellen Farbwert I_t^c aktualisiert. Die Gewichtung $w(\Delta_D)$ der aktuellen Werte nimmt für Voxel hinter der Oberfläche ab, um dahinterliegende weitere Oberflächen nicht zu beeinflussen:

$$w(\Delta_D) = \begin{cases} 1 & \text{if } \Delta_D < 0 \\ \frac{\Phi - \Delta_D}{\Phi} & \text{if } 0 \leq \Delta_D \leq \Phi \\ 0 & \text{if } \Delta_D > \Phi \end{cases} \quad \textbf{(4–22)}$$

$$W(\boldsymbol{p}_c, t) = w(\Delta_D) + W(\boldsymbol{p}_c, t-1) \quad \textbf{(4–23)}$$

$$D(\boldsymbol{p}_c, t) = \frac{D(\boldsymbol{p}_c, t-1)W(\boldsymbol{p}_c, t-1) + \Delta_D^{\Phi} w(\Delta_D)}{w(\Delta_D) + W(\boldsymbol{p}_c, t-1)} \quad \textbf{(4–24)}$$

$$C(\boldsymbol{p}_c, t) = \frac{C(\boldsymbol{p}_c, t-1)W(\boldsymbol{p}_c, t-1) + I_t^c w(\Delta_D)}{w(\Delta_D) + W(\boldsymbol{p}_c, t-1)} \quad \textbf{(4–25)}$$

Um nicht mit jedem neuen Bild alle Voxel aktualisieren zu müssen und den Speicherverbrauch bei großen Umgebungen trotz akzeptabler Voxel-Auflösungen im Rahmen zu halten, speichert fastfusion die Voxel in einer Octree-Struktur. Dadurch können für verschiedene Entfernungen verschiedene Voxel-Auflösungen verwendet werden. 8^3 Voxel werden außerdem jeweils zu einem Würfel zusammengefasst, der je nach dem sich aus der Entfernung ergebenden Skalierungsfaktor $s_l \in \mathbb{N}$ auf der entsprechenden Ebene des Octrees gespeichert wird. Zusätzlich wird zu jedem Würfel die Position seines Ursprungs im Octree $\boldsymbol{o}_l \in \mathbb{N}^3$ gespeichert. Der Ursprungs des Octrees selbst ist $\boldsymbol{o}_t \in \mathbb{R}^3$ mit einem Skalierungsfaktor $s_t \in \mathbb{R}_+$.

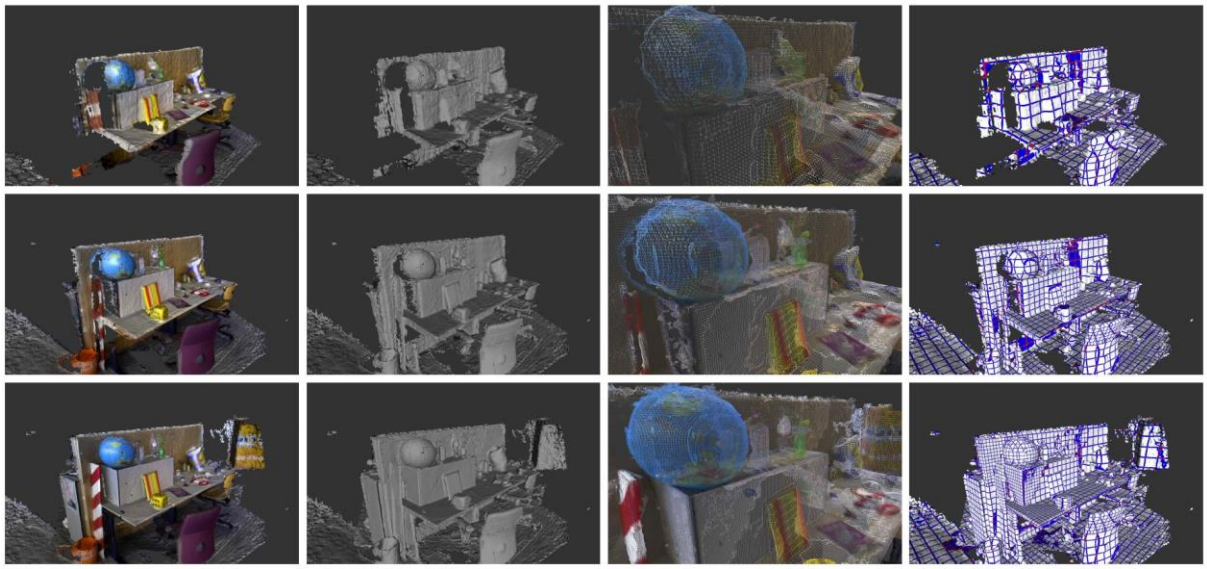


Abb. 4–4: Visualisierung des fastfusion-Algorithmus über die Zeit, entnommen aus (Steinbrücker et al. 2014). Oben nach unten: Ergebnisse nach 50, 170, 300 Frames. Links nach rechts: Geometrie und Textur, nur Geometrie, texturiertes Drahtgitter, Darstellung des Voxelrasters.

Aus der 3D-Weltkoordinate eines Pixels und dem gewünschten Skalenfaktor kann nun der zugehörige Würfel ermittelt werden. Nur noch dieser Würfel und die sich an derselben Stelle befindlichen Würfel mit gröberem Skalierungsfaktor müssen aktualisiert werden. Für die 8^3 Voxel innerhalb eines Würfels ergibt die benötigte Position in Weltkoordinaten $\boldsymbol{p}_w$ aus den lokalen Würfelkoordinaten $\boldsymbol{p}_l \in \{0, \dots, 7\}^3$ wie folgt:

$$\boldsymbol{p}_w = (\boldsymbol{o}_t + (\boldsymbol{o}_l + \boldsymbol{p}_l s_l) s_t) \quad \textbf{(4–26)}$$

Abb. 4–4 zeigt beispielhaft einen Rekonstruktionsverlauf über die Zeit. Klar zu erkennen sind in der rechten Spalte die unterschiedlichen Voxelgrößen je nach Entfernung der Kamera. Die Rekonstruktion wird feiner, wenn die Oberfläche von der Kamera aus näherer Entfernung aufgenommen wird.

Der Algorithmus implementiert die beschriebenen Rechenvorschriften in einer für SIMD-Instruktionen (Single Instruction, Multiple Data) optimierten Weise. Für Details hierzu sei auf die Veröffentlichung (Steinbrücker et al. 2014) verwiesen.

Zur Ausgabe und Anzeige wird die interne Repräsentation der Umgebung in ein 3D-Mesh konvertiert, welches deutlich weniger Speicherplatz benötigt. Dazu kommt der Marching-Cubes-Algorithmus zum Einsatz, auf den hier nicht näher eingegangen wird. Details zur Umsetzung des Marching-Cubes-Algorithmus für die Würfeldatenstruktur in fastfusion finden sich wiederum in der zugehörigen Veröffentlichung.

5 Softwareframework

5.1 Anforderungen

Die spezifischen Anforderungen, die sich aus den in Abschnitt 3.2 definierten Zielen ableiten lassen, sind im Folgenden aufgelistet:

Tab. 5–1: Anforderungen an das Softwareframework

ID	Anforderung
2-01	Die Software soll im Allgemeinen die Aufnahme von Datensätzen mittels Sensoren der visuellen Odometrie unterstützen.
2-02	Im Speziellen soll die RGB-D-Kamera Kinect v2 in die Software eingebunden werden.
2-03	Aufgenommene Datensätze sollen abgespeichert werden können.
2-04	Gespeicherte Datensätze sollen zur späteren Verarbeitung erneut eingelesen werden können.
2-05	Für die Datenaufzeichnung soll ein Filter entwickelt werden, der die Separierung des Zielobjekts vom Laborhintergrund ermöglicht.
2-06	Die Software soll ein Tracking-Modul enthalten, welches basierend auf den gegebenen Sensordaten die Relativlage zwischen Sensor und Zielobjekt schätzt.
2-07	Im Speziellen soll der Algorithmus DIFODO in die Software eingebunden werden.
2-08	Die aktuell geschätzte Relativlage soll in der Benutzeroberfläche visualisiert werden.
2-09	Die berechnete Trajektorie soll zur externen Weiterverarbeitung abgespeichert werden können.
2-10	Die Software soll ein Mapping-Modul enthalten, welches basierend auf den gegebenen Eingabedaten und der Ausgabe des Tracking-Moduls eine 3D-Darstellung des aufgenommenen Objektes erzeugt.
2-11	Im Speziellen soll der Algorithmus fastfusion in die Software eingebunden werden.
2-12	Die 3D-Darstellung des Objekts soll in der Benutzeroberfläche visualisiert werden.
2-13	Das erzeugte 3D-Mesh soll zur externen Weiterverarbeitung abgespeichert werden können.

2-14	Die einzelnen Softwaremodule sollen unabhängig voneinander in getrennten Threads arbeiten, um von Mehrkernprozessoren zu profitieren.
2-15	Die Verarbeitungspipeline soll zwei Modi besitzen: die vollständige Verarbeitung aller Eingabedaten und die Echtzeitverarbeitung der jeweils aktuellsten Daten.
2-16	Die Software soll flexibel erweiterbar sein und über wohldefinierte Schnittstellen weitere Datenquellen (Sensoren) und Algorithmen integrieren können.
2-17	Der Visualisierungsteil soll entkoppelt von der restlichen Programmlogik implementiert werden.
2-18	Die Software soll als Windows-Anwendung entwickelt werden.
2-19	Die Software soll soweit möglich Gebrauch von ausgereiften Programmbibliotheken zur Verarbeitung der Bilddaten machen.

5.2 Grundidee

Mit der implementierten Software sollte ein Framework geschaffen werden, mit dem Algorithmen zur Lageschätzung und 3D-Rekonstruktion zusammen mit Sensoren für visuelle Odometrie getestet und verglichen werden können. Statt für jede Kombination aus Datenquellen und Algorithmen eine eigene Toolchain aufzubauen, kann so eine einzige Anwendung mit einheitlichem Nutzerinterface zur Durchführung von Messkampagnen und zur Visualisierung der Ergebnisse verwendet werden.

Die Software wurde unter dem Namen RVDSLAM (Rendezvous and Docking Simultaneous Localization and Mapping) entwickelt.

Bei der Entwicklung standen eine einfache Bedienbarkeit und insbesondere eine leichte Erweiterbarkeit der Software um weitere Algorithmen und Datenquellen (Sensoren) im Vordergrund. Letzteres wurde durch eine objektorientierte Struktur mit wenigen, klar definierten Schnittstellen unter Verwendung standardisierter Datenstrukturen erreicht. Eingabedaten können sowohl von echten Sensoren als auch von voraufgezeichneten Datensätzen stammen. Die Daten werden in einer Pipeline verarbeitet, in der die Algorithmen über Wrapperklassen integriert sind. Die einzelnen Module der Pipeline laufen unabhängig voneinander in separaten Threads. Die Rekonstruktionsergebnisse werden unmittelbar in einem Visualisierungsfenster dargestellt.

5.3 Verwendete Bibliotheken

Die hier aufgelisteten Bibliotheken wurden direkt im Softwareprojekt verwendet. Bibliotheken, von denen die genannten wiederum selbst abhängig sind, werden nur erwähnt und nicht näher beschrieben.

5.3.1 PCL

Die Point Cloud Library (PCL)[13] ist eine freie Bibliothek zur Anzeige und Verarbeitung von Punktwolken. Sie umfasst viele auf dem aktuellen Stand der Forschung basierende Algorithmen im Bereich der Filterung, Merkmalsextraktion, Oberflächenrekonstruktion, Registrierung, Modellanpassung und Segmentierung und bietet damit eine gemeinsame Infrastruktur für Softwareanwendungen im Bereich der Punktwolkenverarbeitung. Die PCL ist in C++ geschrieben und plattformunabhängig.

Die ersten Algorithmen für die PCL wurden von Dr. Radu Bogdan Rusu an der Technischen Universität München geschrieben. Die erste Version der PCL erschien 2010. Sie wurde ursprünglich von Willow Garage entwickelt. Mittlerweile wird die Entwicklung von der Open Perception Foundation geleitet. Beiträge kommen von Universitäten und anderen Organisationen weltweit. In diesem Softwareframework wird die Version 1.7.2 der PCL eingesetzt.

PCL selbst hängt wiederum ab von den Bibliotheken Boost, Eigen, FLANN und VTK.

5.3.2 OpenCV

OpenCV[14] ist eine freie Bibliothek zur Anzeige und Verarbeitung von 2D-Bilddaten. Sie umfasst viele auf dem aktuellen Stand der Forschung basierende Algorithmen aus dem Bereich der Bildverarbeitung und des maschinellen Lernens und bietet damit eine gemeinsame Infrastruktur für Softwareanwendungen im Bereich des maschinellen Sehens. OpenCV ist in C und C++ geschrieben und plattformunabhängig.

Die erste Version von OpenCV erschien im Jahr 2000 und wurde von einem Intel-Entwickler geschrieben. Die Entwicklung wurde bei Willow Garage weitergeführt. Beiträge kommen von Universitäten und anderen Organisationen weltweit. In diesem Softwareframework wird die Version 3.1 von OpenCV eingesetzt.

5.3.3 Eigen

Eigen[15] ist eine freie Bibliothek für lineare Algebra. Sie umfasst Algorithmen für Matrizen, Vektoren, numerische Lösungsansätze und verwandte Bereiche. Eigen ist hochgradig laufzeitoptimiert, in C++ geschrieben und plattformunabhängig. In diesem Softwareframework wird die Version 3.2.7 von Eigen eingesetzt.

5.3.4 Qt

Qt[16] (gesprochen „cute“) ist eine Bibliothek zur plattformübergreifenden Programmierung grafischer Benutzeroberflächen. Sie bietet aber auch Möglichkeiten

[13] http://www.pointclouds.org/
[14] http://www.opencv.org/
[15] http://eigen.tuxfamily.org/
[16] http://www.qt.io/

zur einfachen plattformübergreifenden Implementierung weiterer Konzepte wie Threads oder Slots und Signals. Qt ist sowohl mit einer kommerziellen als auch mit einer Open-Source-Lizenz verfügbar. Die Open-Source-Lizenz beinhaltet die volle Funktionalität der Bibliothek, stellt jedoch wie alle Open-Source-Lizenzen bestimmte Bedingungen zur Weitergabe der entwickelten Software. Qt ist in C++ geschrieben, verwendet jedoch einen Präprozessor, den Meta Object Compiler (MOC), um die Programmiersprache um einige Fähigkeiten zu erweitern.

Die erste Version von Qt erschien 1995. In diesem Softwareframework wird die Version 5.2.1 von Qt eingesetzt.

5.4 Verwendete Programmiersprache

Das Softwareframework wurde in C++ entwickelt. Da sowohl die wichtigsten Bibliotheken im Bereich des maschinellen Sehens (OpenCV, PCL) und generell auch die im Forschungsbereich entwickelten Algorithmen im Bereich der visuellen Odometrie in C++ vorliegen, bot sich die Verwendung dieser Programmiersprache an. Zusätzlich wurden Funktionalitäten des Qt-Frameworks zur ereignisgesteuerten und nebenläufigen Programmierung verwendet, die die Programmiersprache erweitern.

5.5 Softwarearchitektur

5.5.1 Grundaufbau

Der Aufbau der Software besteht auf der obersten Ebene aus vier getrennten Komponenten, auf die im Verlauf des Kapitels noch im Detail eingegangen wird. Abb. 5–1 zeigt die Komponenten in einem Klassendiagramm. Abb. 5–2 stellt die grundlegende Funktionsweise der Software dar, die im Folgenden beschrieben wird.

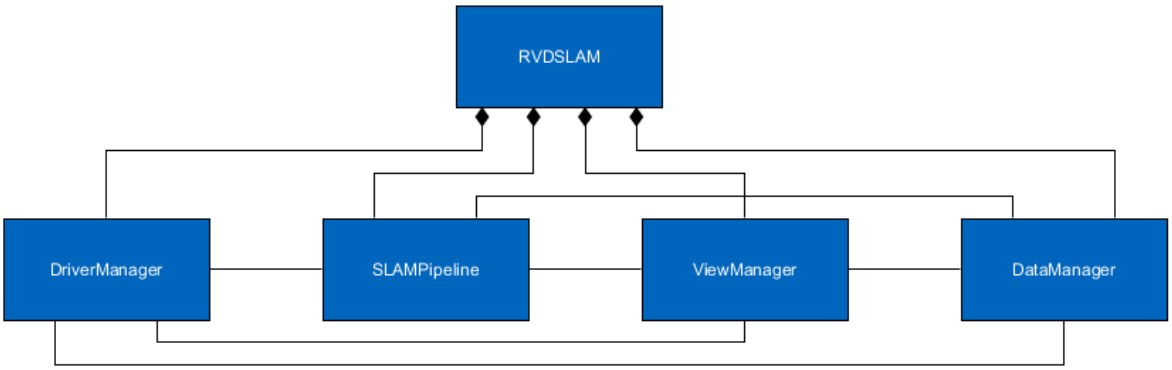


Abb. 5–1: Klassendiagramm der Kernkomponenten

Die `DriverManager`-Klasse (siehe Abschnitt 5.5.2) ist für das Einlesen der Sensordaten zuständig. Die Einbindung erfolgt über den Aufruf der entsprechende API-Aufrufe der Sensoren. Aus den zurückgelieferten Daten wird ein `FramePackage`-Objekt gebildet (siehe Abschnitt 5.6.1) und mit einem optionalen Filter versehen. Anschließend wird das Datenpaket zur 3D-Rekonstruktion an die `SLAMPipeline` weitergereicht, wo es in die Warteschlange des Tracking-Moduls eingereiht wird. Gleichzeitig wird ein Verweis auf das Datenpaket im `DataManager` abgelegt, um bei Bedarf den gesamten Datensatz auf der Festplatte zur späteren Verwendung speichern zu können. Der `DataManager` ist

grundsätzlich für die Verwaltung und Speicherung der eingelesenen bzw. generierten Daten zuständig (siehe Abschnitt 5.5.5).

In der `SLAMPipeline` (siehe Abschnitt 5.5.3) entnimmt das Tracking-Modul die Datenpakete aus der Warteschlange, um die aktuelle geschätzte Relativlage zwischen Sensor und Zielobjekt zu berechnen. Die Relativlagedaten werden dem Datenpaket hinzugefügt, welches anschließend in die Warteschlange des Mapping-Moduls eingereiht wird. Die gesamte Trajektorie kann außerdem über den `DataManager` abgespeichert werden. Zusätzlich wird der `ViewManager` über die neuen Lagedaten informiert, der sich um die Visualisierung auf der Benutzeroberfläche kümmert.

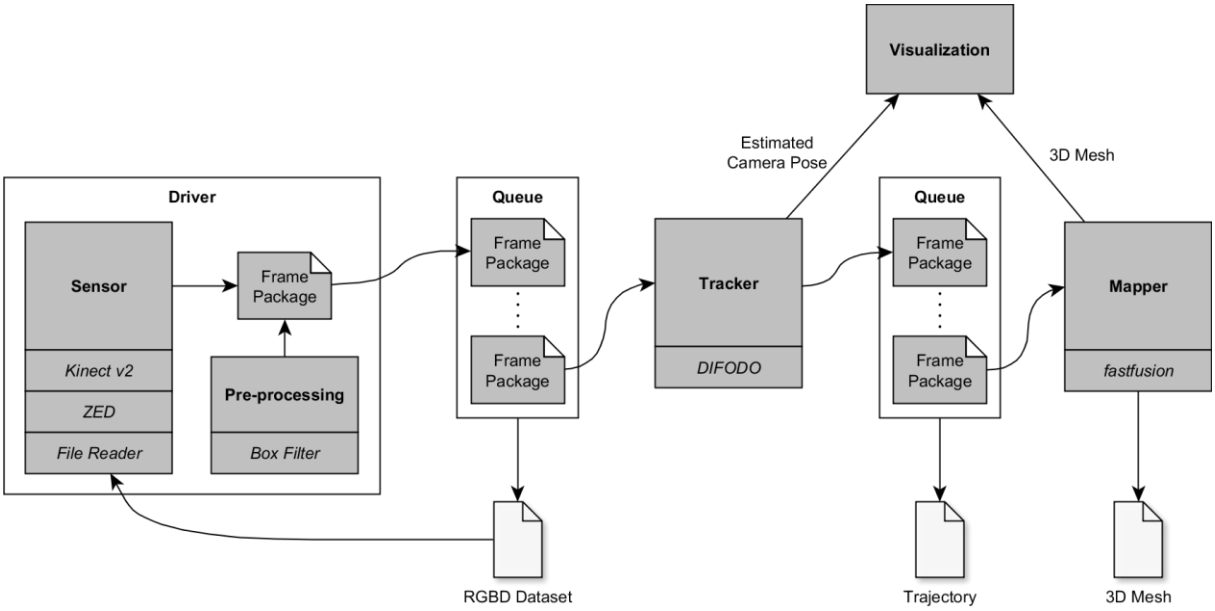


Abb. 5–2: Verarbeitungspipeline der RVDSLAM-Software

Den Abschluss der Verarbeitungspipeline bildet das Mapping-Modul. Es entnimmt die Datenpakete aus seiner Warteschlange und integriert sie in die interne 3D-Repräsentation der vom Sensor erfassten Umgebung. Daraus kann abschließend ein 3D-Mesh oder eine Punktwolke erzeugt werden, die an den `ViewManager` zur Visualisierung übergeben wird. Zusätzlich kann das 3D-Mesh über den `DataManager` zur weiteren Verarbeitung in externen Programmen auf der Festplatte gespeichert werden.

Der `ViewManager` (siehe Abschnitt 5.5.4) ist im Allgemeinen für die Interaktion des Nutzers mit der Software zuständig. Über die Benutzeroberfläche können die einzelnen Module gesteuert, Einstellungen vorgenommen und Speichervorgänge angestoßen werden. Der `ViewManager` verwaltet die Fenster der Benutzeroberfläche und dient als Vermittler zwischen Front- und Backend. Er ist vom Rest der Software so entkoppelt, dass er später leicht herausgelöst werden kann und der Berechnungsteil der Software in die Teleoperator-Umgebung des RACOON-Labs integriert werden kann.

Die Einbindung zusätzlicher Datenquellen (Sensoren) und Algorithmen als Module in das Softwareframework geschieht über sogenannte Wrapper-Klassen (siehe Abschnitt 5.6.2). Diese Wrapper-Klassen stellen ein einheitliches Interface zur Verfügung, so dass die einzelnen Komponenten zur Laufzeit gegeneinander ausgetauscht werden

können. Zur Erweiterung muss lediglich das Interface der entsprechenden Wrapper-Basisklasse implementiert werden.

Die Wrapper-Klassen weisen alle eine gemeinsame Basisklasse auf. Über diese wird das Modul in das Framework integriert und gesteuert. Jedes Modul läuft in einem separaten Thread, um die Eingabedaten unabhängig bearbeiten zu können und eine bestmögliche Performance zu erzielen.

Ein vollständiges Klassendiagramm aller Komponenten findet sich in Abb. 5–4.

Zur Verarbeitung der Daten bietet die Software zwei verschiedene Modi: Einen Echtzeit- und einen Gesamtdatenmodus. Im Echtzeitmodus verarbeiten die Module der `SLAMPipeline` immer das zuletzt eingegangene Datenpaket. Das ermöglicht die Live-Darstellung der Ansicht des Objekts, kann sich jedoch Nachteilig auf die Rekonstruktionsqualität auswirken, wenn die Verarbeitungsgeschwindigkeit zu gering ist und zu viel Datenpakete verworfen werden. Im Gesamtdatenmodus werden die sämtliche aufgenommenen Datenpakete nacheinander verarbeitet. Das ist optimal für die Rekonstruktionsqualität, bei zu langsamer Verarbeitungsgeschwindigkeit ist jedoch keine Echtzeitdarstellung mehr möglich.

5.5.2 DriverManager

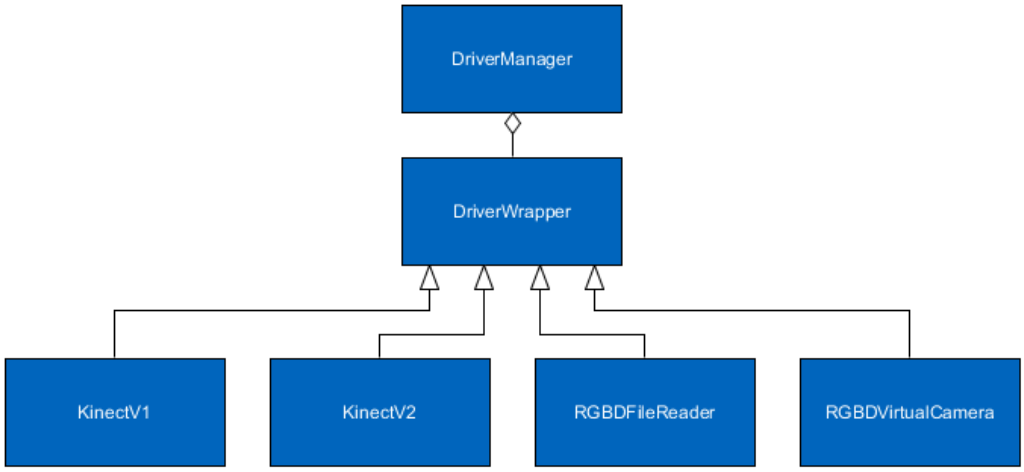


Abb. 5–3: `DriverManager`-Klassendiagramm

Der Dateneingabeteil umfasst die `DriverManager`-Klasse und die `DriverWrapper`-Klasse mit den spezifischen Implementierungen für verschiedene Sensoren in den entsprechenden abgeleiteten Klassen (siehe Klassendiagramm in Abb. 5–3). Die `DriverManager`-Klasse ist für den Wechsel zwischen den einzelnen `DriverWrapper`-Objekten („Treiber-Modulen“) zuständig und reicht Anfragen zum Zustandswechsel (Start/Pause/Resume/Reset) an sie weiter. Das Treiber-Modul liest die Sensordaten in einem separaten Thread ein und übergibt sie in Form eines `FramePackage`-Objekts (siehe Abschnitt 5.6.1) an den `DriverManager`. Dieser fügt, falls spezifiziert, ein `Filter`-Objekt hinzu, speichert einen Verweis auf das Datenpaket im `DataManager`, benachrichtigt den `ViewManager` zur Darstellung des aufgenommenen Farb-/Tiefenbildpaars und übergibt das Paket an die `SLAMPipeline`, wo es in die Warteschlange des Tracking-Moduls eingereiht wird.

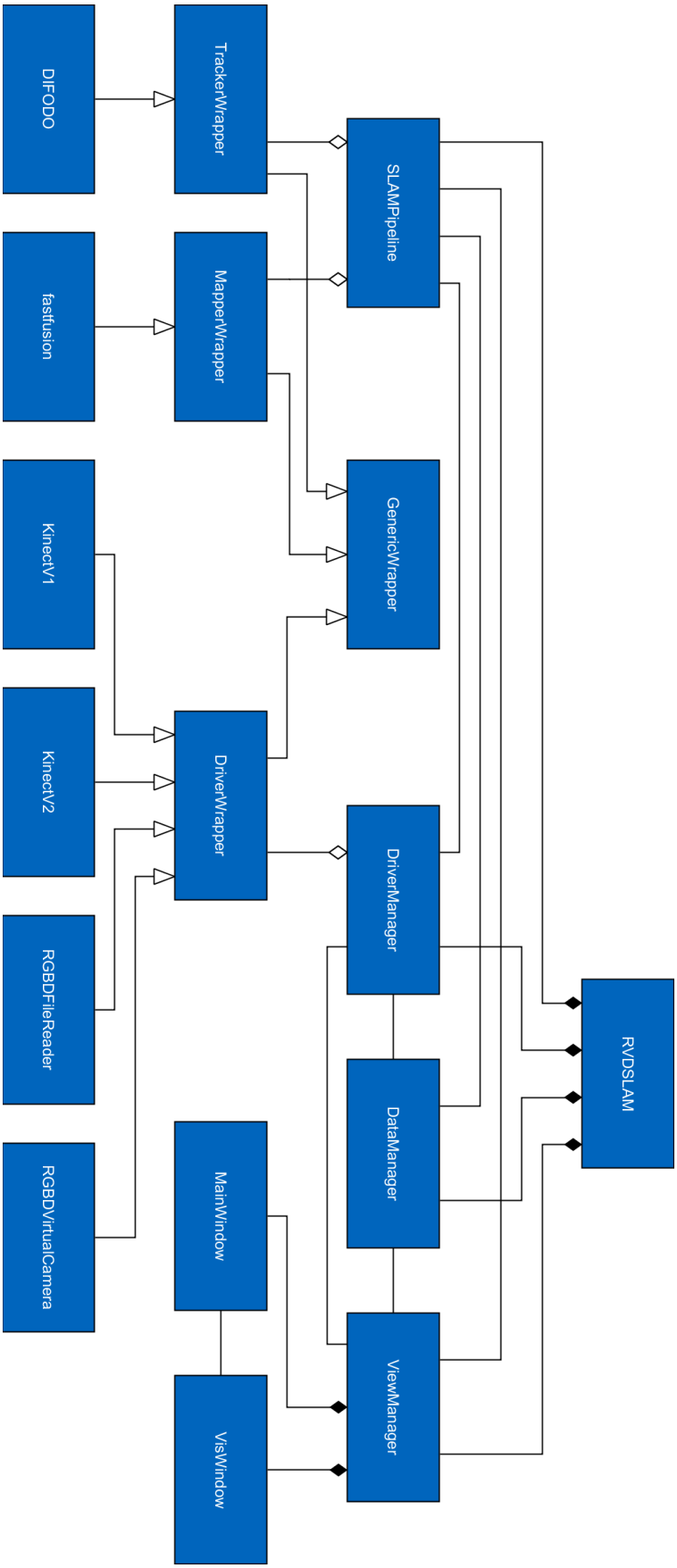


Abb. 5–4: Gesamtklassendiagramm

5.5.2.1 Verfügbare Sensoren

Die Sensoren sind je über eine spezifische Implementierung des `DriverWrapper`-Interfaces in die Software eingebunden (siehe auch Abschnitt 5.6.2.2). Die Hauptmethode, die dabei implementiert werden muss, ist `bool getFramePackage(std::shared_ptr<FramePackage>& frame_package)`, welche einen Verweis auf ein neu generiertes `FramePackage` zurückgibt. Innerhalb eines `DriverWrapper`-Objekts wird über eine sensorspezifische API auf die Sensoren zugegriffen, um die Daten auszulesen. Die folgenden Sensoren sind bisher in die Software integriert worden:

- Kinect v1 RGB-D-Kamera (im Rahmen dieser Arbeit, zu Testzwecken)
- Kinect v2 RGB-D-Kamera (im Rahmen dieser Arbeit)
- Bumblebee 2 Stereokamera (Gschwindt 2016)
- ZED Stereokamera (Dziura et al. 2017)

Die Kinect v2 arbeitet in der Software mit einer Auflösung von 512 x 424 px, was der Auflösung des Tiefenbilds entspricht. Um das zugehörige Farbbild (ursprünglich 1920 x 1080 px) zu ermitteln, werden die sich aus dem Tiefenbild ergebenden Punkte in das Koordinatensystem der Farbkamera transformiert und auf den Sensor projiziert, woraus sich die entsprechenden Farbwerte pro Pixel ergeben. Die intrinsischen Parameter (Hauptpunkt und Brennweite) werden aus dem Sensor ausgelesen und im `FramePackage` hinterlegt. Optional kann über die Benutzeroberfläche die Verwendung des Bilds der Infrarotkamera als Farbbild ausgewählt werden, bspw. zur Rekonstruktion bei Dunkelheit.

5.5.2.2 Virtuelle Sensoren

Zusätzlich zu den realen Sensoren bietet die Software die Möglichkeit, voraufgezeichnete Datensätze einzuspielen. Die zugehörigen `DriverWrapper`-Klassen (`DriverWrapperRGBDFileReader` und `DriverWrapperRGBDVirtualCamera`) agieren dabei sozusagen als „virtueller Sensor“, der die Datenpakete nach und nach von der Festplatte liest und mit dem originalen Zeitstempel versehen als `FramePackage` bereitstellt. Der Ordner des aufgezeichneten Datensatzes wird über die Benutzeroberfläche ausgewählt.

Zwei verschiedene Varianten dieser virtuellen Sensoren wurden in der Software implementiert:

- Ein `RGBDFileReader`-Modul, das den gesamten Datensatz linear so schnell wie möglich von der Festplatte einliest. Die Einleserate ist dabei lediglich von der Verarbeitungsgeschwindigkeit der Software beschränkt.
- Ein `RGBDVirtualCamera`-Modul, das den Datensatz von der Festplatte so einliest, als ob ein echter Sensor die Daten zur Verfügung stellen würde. Die virtuelle Kamera läuft dabei in einem separaten Thread, wo sie die Datenpakete mit der im Datensatz spezifizierten Bildwiederholrate einliest und verwirft, falls sie vom Treiber-Modul nicht rechtzeitig abgerufen werden. Die Einleserate wird hier von der Bildwiederholrate der virtuellen Kamera beschränkt.

5.5.2.3 Vorfilterung

Eine Vorfilterung der Eingabedaten kann aus verschiedenen Gründen notwendig sein. Beispielsweise zur Kontrastkorrektur, zur Korrektur einer Verzerrung oder zur Unterdrückung von Tiefenwerten unter/über einer bestimmten Schwelle. Für solche Fälle kann eine spezifische Filterklasse entworfen werden, die das Interface `FramePackage::Filter` implementiert.

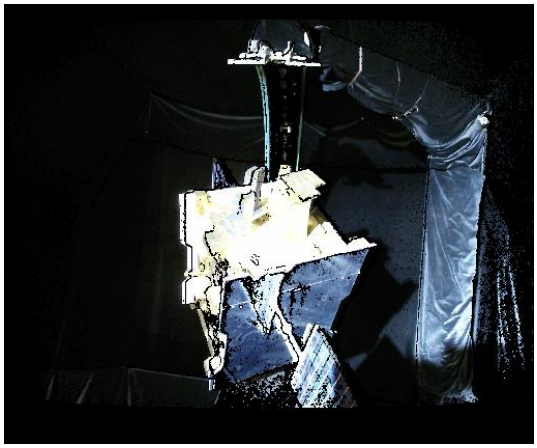
Für das für diese Arbeit vorgegebene Referenzszenario (siehe Abschnitt 3.1), welches in einer Simulator-Umgebung nachgebildet wird, wurde ein Filter zur Unterdrückung der Strukturen des Simulators im Hintergrund entwickelt. Das Satellitenmodell im Simulator befindet sich nicht vor einem weit entfernten (und sich damit außerhalb des Empfindlichkeitsbereichs der Tiefenkamera befindlichen) Hintergrund, sondern ist zum einen an einem Trägergestell verankert und zum anderen von der (mit schwarzen Tüchern abgedeckten) Laborwand umgeben.

Sowohl das Trägergestell als auch der Hintergrund sind statisch relativ zum sich drehenden Satelliten und können somit Probleme beim Lageschätzungsalgorithmus hervorrufen, der erwartet, dass sich Objekte im Bild nicht relativ zur Umgebung bewegen. Im normalen Einsatzbereich dieser Algorithmen (bewegte Kamera in statischer Umgebung) sind solche unerwünschten Bildkomponenten Vordergrundobjekte, im hier betrachteten Szenario Hintergrundobjekte.

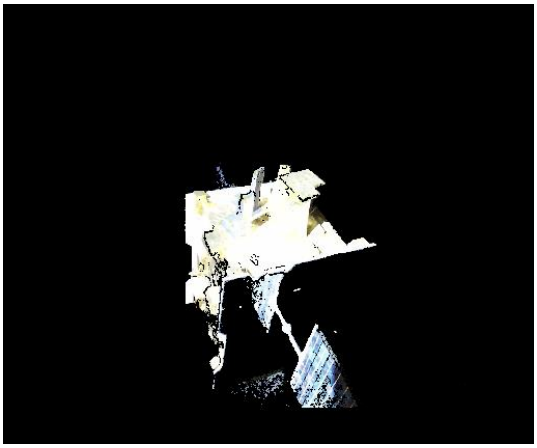
Zur Vorfilterung wurde deshalb ein `BoxFilter` implementiert, der Punkte außerhalb eines würfelförmigen Bereichs ignoriert. Der Bereich wird in horizontaler und vertikaler Richtung über minimale und maximale Pixelkoordinaten definiert. In Objektrichtung wird er über minimale und maximale Tiefenwerte festgelegt. Tab. 5–2 listet die Parameter auf. Da die eingesetzten Algorithmen grundsätzlich robust gegenüber kleineren Bildteilen sind, die sich relativ zum Hauptobjekt bewegen, ist bei der Wahl der Begrenzungswerte keine besondere Genauigkeit vonnöten. Abb. 5–5 zeigt die Auswirkungen des `BoxFilters` auf Farb- und Tiefenbild.

Tab. 5–2: `BoxFilter`-Parameter

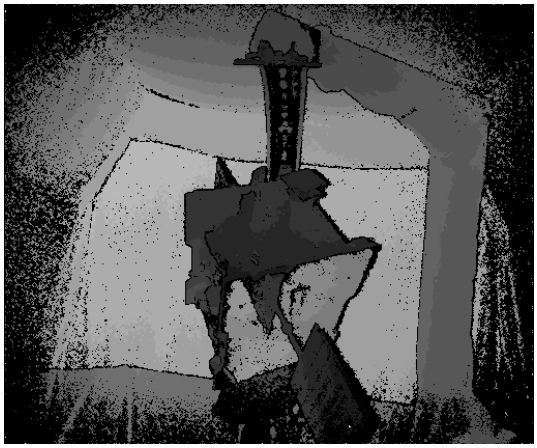
Parameter	Standardwert	Beschreibung
`u_min`	0 [px]	Minimale horizontale Pixelposition (von links gerechnet)
`u_max`	640 [px]	Maximale horizontale Pixelposition (von links gerechnet)
`v_min`	0 [px]	Minimale vertikale Pixelposition (von oben gerechnet)
`v_max`	480 [px]	Maximale vertikale Pixelposition (von oben gerechnet)
`z_min`	0 [m]	Minimaler Tiefenwert
`z_max`	10,0 [m]	Maximaler Tiefenwert



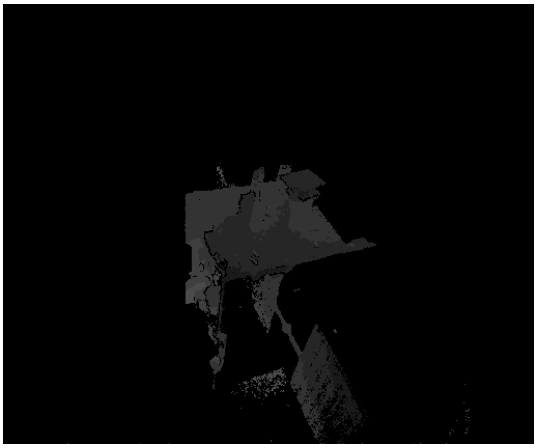
(a) Farbbild nach Filterung mit `BoxFilter`



(b) Farbbild nach Filterung mit `BoxFilter`



(c) Tiefenbild (kontrastversärkt) vor Filterung mit `BoxFilter`



(d) Tiefenbild nach Filterung mit `BoxFilter`

Abb. 5–5: Auswirkungen des `BoxFilters` auf Farb- und Tiefenbild, entnommen aus (Dziura et al. 2017)

5.5.3 SLAMPipeline

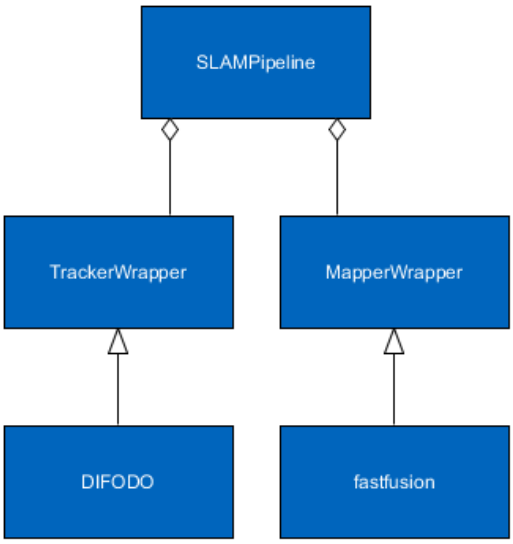


Abb. 5–6: `SLAMPipeline`-Klassendiagramm

Die `SLAMPipeline`-Klasse beinhaltet den eigentliche Kern der Software. Hier werden die eingelesenen Datenpakete verarbeitet und an die Algorithmen zur Lageschätzung (Tracker) und 3D-Rekonstruktion (Mapper) übergeben. Abb. 5–6 zeigt das zugehörige Klassendiagramm.

Die `SLAMPipeline` ist für die Verwaltung der Tracking- und Mapping-Module zuständig. Sie übergibt zu verarbeitende Datenpakete an die Warteschlangen der Module und kümmert sich um die Weiterverarbeitung der Ausgabedaten. Außerdem reicht sie Anfragen zu Status- oder Einstellungsänderungen an die Module weiter.

In der `SLAMPipeline` werden für jedes vom `DriverManager` übergebene `FramePackage` die folgenden Schritte ausgeführt:

1. Das `FramePackage` wird in die Warteschlange des Tracking-Moduls eingereiht.
2. Das Tracking-Modul, welches in einem separaten Thread läuft, entnimmt das `FramePackage` – je nach Einstellung des Verarbeitungsmodus das erste oder letzte der Warteschlange.
3. Das Tracking-Modul berechnet die geschätzte Relativlage zwischen Sensor und Zielobjekt und fügt sie dem `FramePackage` hinzu.
4. Das Tracking-Modul übergibt das `FramePackage` wieder an die `SLAMPipeline`.
5. Die `SLAMPipeline` informiert den `ViewManager` über das verarbeitete `FramePackage` zur Darstellung des Farb- und Tiefenbilds und der neuen Lagedaten.
6. Die `SLAMPipeline` informiert den `DataManager` über das verarbeitete `FramePackage` zur Speicherung der Trajektorie.
7. Die `SLAMPipeline` übergibt das verarbeitete `FramePackage` an die Warteschlange des Mapping-Moduls.
8. Das Mapping-Modul, welches in einem separaten Thread läuft, entnimmt das `FramePackage` – je nach Einstellung des Verarbeitungsmodus das erste oder letzte der Warteschlange.
9. Das Mapping-Modul integriert die Daten des `FramePackage` in seine interne 3D-Repräsentation.

10. Das Mapping-Modul meldet die fertige Berechnung an die `SLAMPipeline`.
11. Das Mapping-Modul berechnet aus der internen 3D-Repräsentation ein 3D-Mesh der vom Sensor erfassten Umgebung.
12. Das Mapping-Modul übergibt das fertige Mesh an die `SLAMPipeline`.
13. Die `SLAMPipeline` informiert den `ViewManager` über das verarbeitete `FramePackage` zur Darstellung des Farb- und Tiefenbilds.
14. Die `SLAMPipeline` übergibt das berechnete Mesh an den `DataManager` zur Speicherung.
15. Die `SLAMPipeline` übergibt das berechnete Mesh an den `ViewManager` zur Darstellung.

Die Algorithmen der `SLAMPipeline` werden über Wrapper-Klassen in die Software eingebunden, so dass sie über ein einheitliches Interface verfügen und leicht gegeneinander austauschbar sind.

Die Lageschätzungsalgorithmen werden über die `TrackerWrapper`-Klasse in die Software integriert (siehe auch Abschnitt 5.6.2.3). Das Interface verfügt über zwei Hauptmethoden, die von abgeleiteten Klassen implementiert werden müssen: `void computeNextPose(const FramePackage & frame_package)` dient zur Berechnung der nächsten Lageschätzung auf Basis des übergebenen `FramePackage`-Objekts. Anschließend kann über `Eigen::Affine3d getPose()` die geschätzte Relativlage in Form einer affinen 3D-Transformationsmatrix abgerufen werden.

Die 3D-Rekonstruktionsalgorithmen werden über die `MapperWrapper`-Klasse eingebunden (siehe auch Abschnitt 5.6.2.4). Über `void updateMap(const FramePackage & frame_package)` wird die interne Repräsentation der Umgebung aktualisiert. Anschließend kann über `std::shared_ptr<pcl::PolygonMesh> getMesh()` ein 3D-Mesh der Umgebung ausgelesen werden.

Im Folgenden wird darauf eingegangen, wie die in Kapitel 4 ausgewählten Algorithmen in ihre Wrapper-Klasse integriert wurden.

5.5.3.1 Einbindung von DIFODO

Details zur theoretischen Funktionsweise des Algorithmus können in Abschnitt 4.3.2 und in der Originalveröffentlichung (Jaimez und González-Jiménez 2015) nachgelesen werden.

Der DIFODO-Tracker wird in der Klasse `TrackerWrapperDIFODO` als Objekt der Klasse `DIFODO` eingebunden, die wiederum von der abstrakten Klasse `mrpt::vision::CDifodo` erbt und deren Methode `void loadFrame()` implementiert. In der Methode `loadFrame()` muss das aktuelle Tiefenbild in die Variable `depth_wf` vom Typ `Eigen::Matrix3f` geladen werden, allerdings horizontal und vertikal gespiegelt. Da die Methode keinen Übergabewert für das Tiefenbild vorsieht, wurde zusätzlich die Methode `void setInputDepthImage(cv::Mat image, unsigned int scale)` implementiert, die das übergebene Tiefenbild und den zugehörigen Skalierungsfaktor zwischen Tiefenbild-Pixelwerten und Entfernung in Metern in einer lokalen Variable speichert.

Der Ablauf der Berechnung der nächsten Lageschätzung in der Methode `computeNextPose()` der `TrackerWrapperDIFODO`-Klasse ist nun wie folgt:

- Falls sich die Brennweitenparameter der Kamera geändert haben, wird zuerst über die Methode `void setFOV(float new_fovh, float new_fovv)` das Sichtfeld neu gesetzt.
- Anschließend wird `setInputDepthImage()` mit dem aktuellen Tiefenbild aufgerufen.
- Das Tiefenbild wird über den Aufruf von `loadFrame()` nach `depth_wf` kopiert.
- Zusätzlich muss die Variable `fps` der Klasse `mrpt::vision::CDifodo` gesetzt werden. Die aktuelle Bildwiederholrate ergibt sich dabei einfach als

$$\frac{1}{t_{new}-t_{old}} \qquad \textbf{(5–1)}$$

  wobei t_{new} der aktuelle Zeitstempel in Sekunden ist und t_{old} der Zeitstempel in Sekunden des vorherigen Aufrufs.
- Anschließend wird mit dem Aufruf der Methode `odometryCalculation()` der Klasse `mrpt::vision::CDifodo` die nächste Berechnung angestoßen.

Über die Methode `getPose()` der `TrackerWrapperDIFODO`-Klasse kann nun die aktuelle Lageschätzung abgerufen werden. Dabei wird intern die Variable `cam_pose` der Klasse `mrpt::vision::CDifodo` vom Typ `mrpt::poses::CPose3D` ausgelesen und in den Typ `Eigen::Affine3d` konvertiert.

Die in Tab. 5–3 aufgelisteten Parameter des Algorithmus können über den Konstruktor der `DIFODO`-Klasse eingestellt werden.

Tab. 5–3: DIFODO-Parameter

Parameter	Standardwert	Beschreibung
`downsample`	1	Faktor, um den die Seitenlänge des Eingabebilds geteilt wird, um die Berechnung zu beschleunigen
`ctf_levels`	4	Anzahl der Ebenen der Gaußfilterpyramide (siehe Abschnitt 4.3.2)
`max_depth`	5,0 [m]	Maximale Distanz eines Eingabedatenpunktes

5.5.3.2 Einbindung von dvo_slam

Für Details zur theoretischen Funktionsweise des Algorithmus siehe Abschnitt 4.4.2 und die Originalveröffentlichung (Kerl et al. 2013a).

Der dvo_slam-Algorithmus konnte in der zur Verfügung stehenden Zeit nicht in das Framework eingebunden werden, da er stark in das Robot Operating System integriert ist und dessen spezifische Datentypen verwendet. Zur Verwendung wären aufwändige Quellcodeänderungen nötig gewesen, um den Algorithmus aus ROS herauszulösen.

An dieser Stelle sei auf die zugehörige Studienarbeit (Züfle 2017) verwiesen.

5.5.3.3 Einbindung von fastfusion

Details zur theoretischen Funktionsweise des Algorithmus können in Abschnitt 4.5.2 und in der Originalveröffentlichung (Steinbrücker et al. 2014) nachgelesen werden.

Der fastfusion-Mapper wird in der Klasse `MapperWrapperFastFusion` als Objekt vom Typ `FusionMipMapCPU` eingebunden. Das Objekt muss lediglich über seinen Konstruktor initialisiert werden.

Zur Aktualisierung der internen Repräsentation der Umgebung wird die Methode `int addMap(const cv::Mat &depth, CameraInfo caminfo, const cv::Mat &rgb, float scaling, float maxcamdistance)` des `FusionMipMapCPU`-Objekts aufgerufen.

Übergabeparameter sind

- `cv::Mat &depth` – das Tiefenbild in Form eines OpenCV-Matrix-Objekts
- `CameraInfo caminfo` – ein `CameraInfo`-Objekt (siehe unten)
- `cv::Mat &rgb` – das Farbbild in Form eines OpenCV-Matrix-Objekts
- `float scaling` – der Skalierungsfaktor zwischen Tiefenbild-Pixelwerten und Entfernung im Metern
- `float maxcamdistance` – der Maximalwert der zu beachtenden Entfernungswerte

Das `CameraInfo`-Objekt wird erzeugt aus

- der intrinsischen 3x3 Parametermatrix über die Objektmethode `bool setIntrinsic(const cv::Mat &intrinsic)`
- dem Rotationsanteil der affinen 3D-Pose als 3x3-OpenCV-Rotationsmatrix über die Objektmethode `bool setRotation(const cv::Mat &rotation)`
- dem Translationsanteilen der affinen 3D-Pose als 3x1-OpenCV-Matrix über die Objektmethode `void setTranslation(const cv::Mat &translation)`

Über die Methode `getMesh()` der `MapperWrapperFastFusion`-Klasse kann anschließend ein 3D-Mesh der Umgebungsrepräsentation erzeugt und abgerufen werden. Dabei geschieht intern Folgendes:

1. Die Methode `bool updateMeshes()` des `FusionMipMapCPU`-Objekts wird aufgerufen, um aus der internen TSDF-Repräsentation ein Mesh zu erzeugen.
2. Das erzeugte Mesh wird über die Methode `MeshInterleaved getMeshInterleavedMarchingCubes()` des `FusionMipMapCPU`-Objekts abgerufen.
3. Das `MeshInterleaved`-Objekt wird in den Typ `pcl::PolygonMesh` konvertiert.

In Tab. 5–4 sind die einstellbaren Parameter des Algorithmus aufgelistet.

Tab. 5–4: fastfusion-Parameter

Parameter	Standardwert	Beschreibung
voxel_scale	0,005	Skalierungsfaktor der Seitenlänge eines Voxels des Voxelgitters der internen TSDF-Umgebungsrepräsentation
max_depth	5,0 [m]	Maximale Distanz eines Eingabedatenpunktes

5.5.4 ViewManager

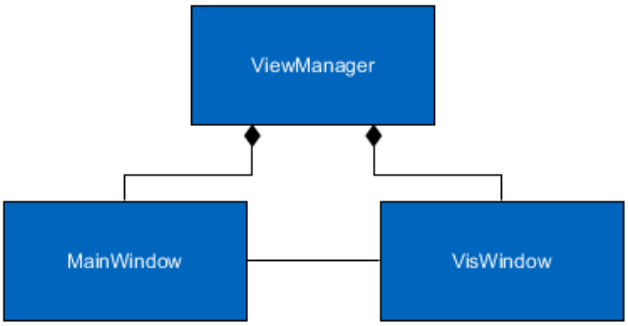


Abb. 5–7: `ViewManager`-Klassendiagramm

Der `ViewManager` verwaltet die Fenster der Benutzeroberfläche und dient als Vermittler zwischen Front- und Backend. Die Klassen `MainWindow` und `VisWindow` beinhalten Funktionen zur Manipulation und Aktualisierung der Fensterelemente sowie die Rückruffunktionen, die die Interaktionen des Benutzers verarbeiten. Abb. 5–7 zeigt das zugehörige Klassendiagramm.

Die Benutzeroberfläche wurde so gestaltet, dass sie eine einfache Steuerung der Software und einen direkten Überblick über den Status der Verarbeitungspipeline und der Ausgabevisualisierung ermöglicht. Alle Funktionalitäten sind auf einen Blick verfügbar, aufgeteilt auf zwei Fenster, eines zur Modulsteuerung (Hauptfenster) und eines zur Darstellung der Ausgabedaten (Visualisierungsfenster).

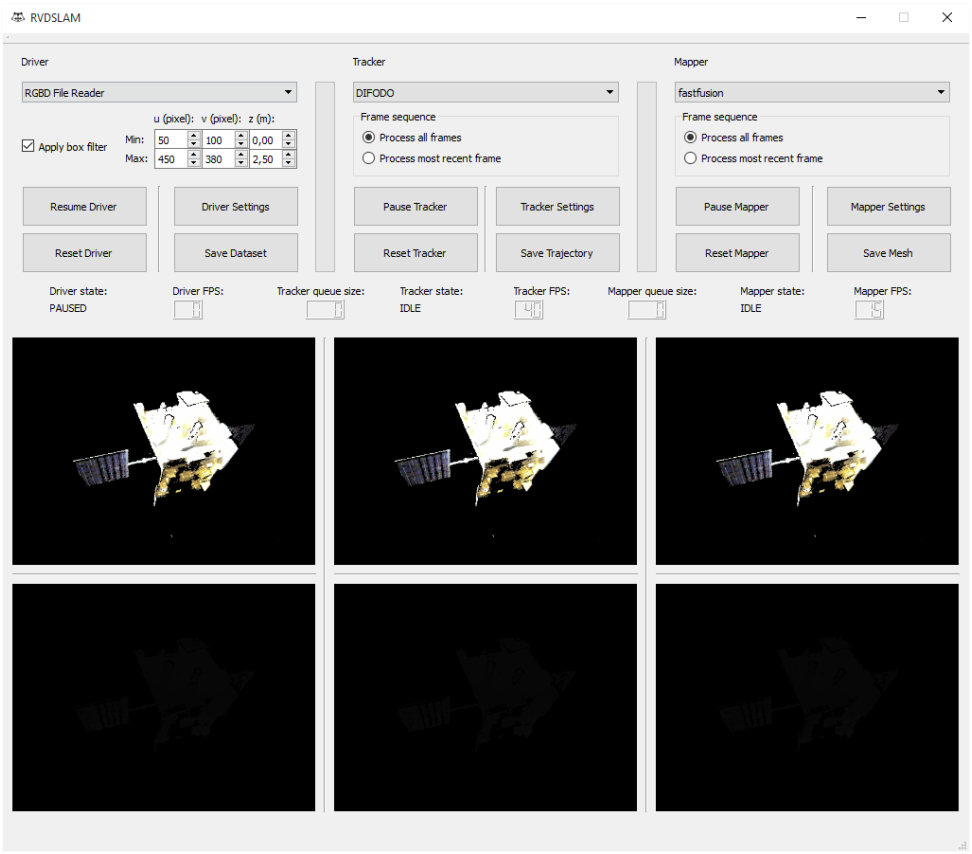
5.5.4.1 Hauptfenster

Über das Hauptfenster (siehe Abb. 5–8(a)) können die drei Module der Verarbeitungspipeline (Driver, Tracker, Mapper) gesteuert, eingestellt und ihr Status eingesehen werden. Die Anordnung ihrer Bedienbereiche von links nach rechts entspricht dem Durchlauf eines Datenpakets in der Pipeline. Da die Module auf einer gemeinsamen Kernarchitektur basieren (siehe Abschnitt 5.6.2.1), konnten die Bereiche ähnlich gestaltet werden.

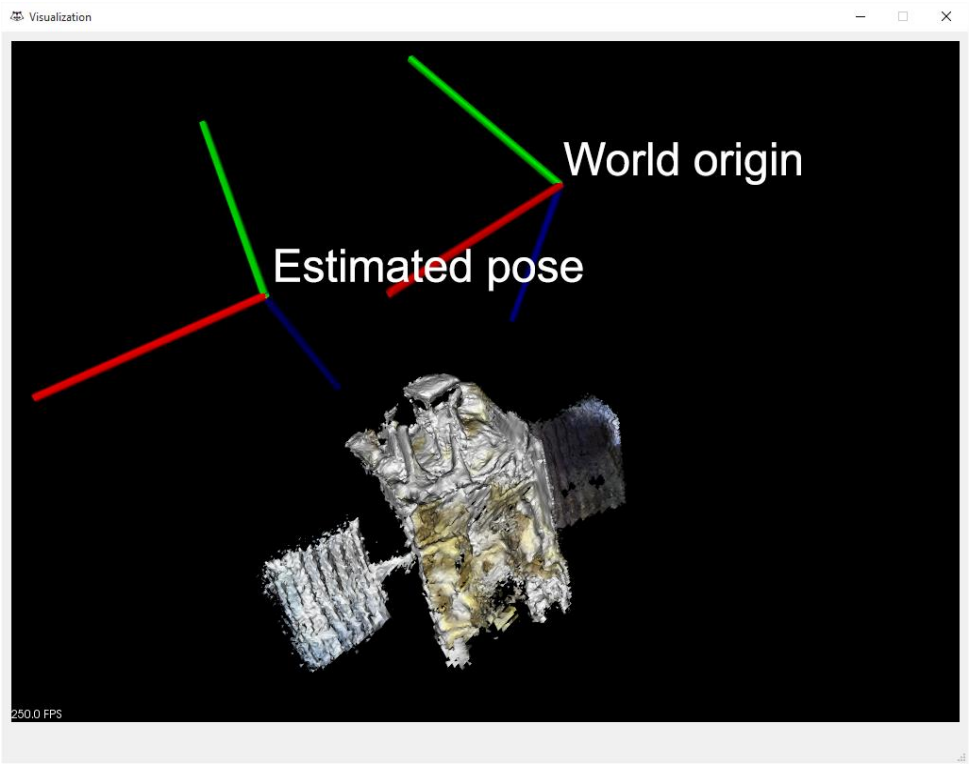
Die einzelnen Module können Über Drop-Down-Menüs zur Laufzeit gegen andere Implementierungen ausgewechselt werden. Die Bedienbereiche zeigen den aktuellen Zustand, in dem sich die Module befinden, und ihre aktuelle Verarbeitungsgeschwindigkeit. Über große Schaltflächen lassen sich der Zustand der Module steuern, ihre Einstellmöglichkeiten aufrufen und die Speicherung von Ausgabedaten anstoßen. Unter den Bedienbereichen werden die aktuell von den jeweiligen Modulen verarbeiteten Farb-/Tiefenbildpaare angezeigt. Zwischen den Modulen werden die Warteschlangen für die Datenpakete visualisiert.

5.5.4.2 Visualisierungsfenster

Im Visualisierungsfenster (siehe Abb. 5–8(b)) werden die Ausgabedaten der Tracking- und Mapping-Module „live“ während der Verarbeitung angezeigt. Es dient zum einen der Funktionskontrolle, zum anderen kann es für eine qualitative Bewertung der Rekonstruktionsqualität verwendet werden.



(a) Hauptfenster



(b) Visualisierungsfenster

Abb. 5–8: Benutzeroberfläche

Das Fenster besteht aus einem großen Renderingbereich, in dem die geschätzte Relativpose zwischen Sensor und Zielobjekt als virtuelles Koordinatensystem und die 3D-Rekonstruktion als Mesh dargestellt werden. Zusätzlich werden das Ursprungskoordinatensystem und (wenn im Datensatz vorhanden) die wahre Relativlage (engl.: ground truth pose) eingeblendet.

Der sichtbare Bereich kann per Mausklick verschoben und gedreht werden und über das Mausrad gezoomt werden.

5.5.5 DataManager

Die `DataManager`-Klasse beinhaltet Funktionen zur Speicherung von Datensätzen und Ausgabedaten. Sie wird von `DriverManager` (siehe Abschnitt 5.5.2) und `SLAMPipeline` (siehe Abschnitt 5.5.3) mit Daten versorgt.

5.5.5.1 Speicherung von Datensätzen

Über die Funktion `bool saveCapturedDataset(const std::string dirpath) const` können aufgenommene Datensätze zum späteren Wiedereinlesen über `RGBDFileReader` oder `RGBDVirtualCamera` oder zur externen Bearbeitung auf der Festplatte gespeichert werden. Als Parameter wird der Speicherpfad übergeben. Der Datensatz beinhaltet zusätzlich zu den Sensorbilddaten die intrinsischen Kameraparameter und den Tiefenbild-Skalierungsfaktor als Metadateninformationen.

Die Struktur eines auf der Festplatte abgelegten Datensatzes ist wie folgt:

- Ein Unterordner `depth` mit einer 16-bit-Grauwert-PNG-Datei pro Tiefenbild mit dem Zeitstempel als Dateinamen.
- Ein Unterordner `rgb` mit einer 8-bit-RGB-PNG-Datei pro Farbbild mit dem Zeitstempel als Dateinamen.
- Eine Datei `rgb.txt` mit einer Zeile pro Bild, jeweils mit dem Zeitstempel in Sekunden gefolgt vom relativen Pfad zum zugehörigen Farbbild. Beispiel:

```
# color images
# line format: timestamp filename
4597.977514 rgb/4597.977514.png
4598.044457 rgb/4598.044457.png
4598.111465 rgb/4598.111465.png
[...]
```

- Eine Datei `depth.txt` mit einer Zeile pro Bild, jeweils mit dem Zeitstempel in Sekunden gefolgt vom relativen Pfad zum zugehörigen Tiefenbild. Beispiel:

```
# depth images
# line format: timestamp filename
4597.977513 depth/4597.977513.png
4598.044457 depth/4598.044457.png
4598.111465 depth/4598.111465.png
[...]
```

- Eine Datei `intrinsics.txt` mit den intrinsischen Kameraparametern in Pixeln. Beispiel:

```
# rgb and depth image intrinsics
# fx fy cx cy
366.2 366.2 259.7 202.2
```

- Eine Datei `scale.txt` mit dem Skalierungsfaktor zwischen Tiefenbild-Pixelwerten und Entfernung im Metern. Beispiel:

```
# depth image scale factor
1000
```

- Die mit einem ‚#' beginnenden Zeilen sind Kommentarzeilen.

5.5.5.2 Speicherung der berechneten Relativtrajektorie

Über die Funktion `bool saveEstimatedTrajectory(const std::string filepath) const` kann die berechnete Relativtrajektorie zwischen Sensor und Zielobjekt exportiert werden. Dabei wird eine Textdatei erstellt mit einer Zeile pro Zeitschritt, jeweils mit dem Zeitstempel in Sekunden gefolgt von der berechneten Sensorlage als Quaternion.

Beispiel:

```
# estimated pose trajectory file
# line format: timestamp tx ty tz qx qy qz qw
4597.977514 0.000000 0.000000 0.000000 0.000000 -0.000000 1.000000 -0.000000
4598.044457 0.000189 0.000080 0.001020 -0.000108 0.000035 1.000000 -0.000007
4598.111465 0.000425 0.001803 0.000214 -0.000014 0.000190 1.000000 -0.000236
[...]
```

5.5.5.3 Speicherung des generierten 3D-Meshs

Über die Funktion `bool saveMesh(const std::string filepath) const` kann das berechnete 3D-Mesh zur weiteren Bearbeitung exportiert werden. Der Dateityp wird basierend auf der Dateiendung bestimmt. Mögliche Dateitypen sind PLY, STL und VTK.

5.6 Datentypen & Schnittstellen zur Erweiterung

Bei der Entwicklung des Softwareframeworks wurde insbesondere auf eine leichte Erweiterbarkeit um zusätzliche Sensoren und Algorithmen geachtet. Es gibt lediglich einen Datentyp, der alle Informationen der eingelesenen Sensordaten bündelt. Die einzelnen Module der Verarbeitungspipeline werden über Wrapper-Klassen in das System integriert, die über ein generisches Interface verfügen. Kann die Funktionalität der Sensoren bzw. Algorithmen über dieses Interface bereitgestellt werden, lassen sie sich als Modul in das System integrieren und zur Laufzeit wechseln.

5.6.1 FramePackage

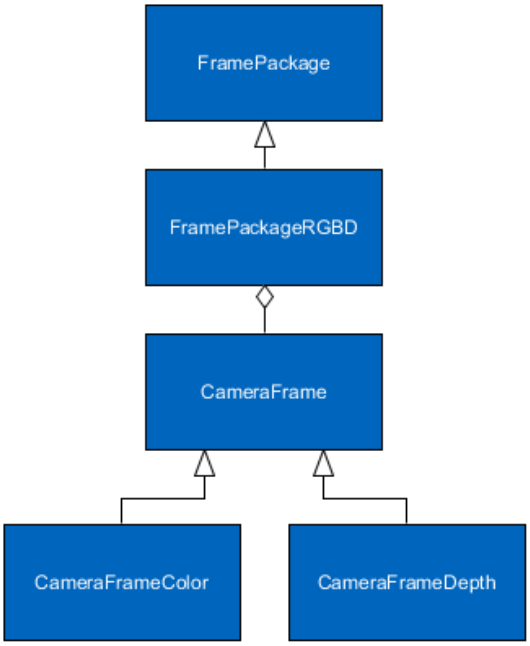


Abb. 5–9: `FramePackage`-Klassendiagramm

Die Datenpakete, die zwischen den Modulen der Verarbeitungspipeline ausgetauscht werden, basieren auf einem einheitlichen Datentyp, der `FramePackage`-Klasse. Diese `FramePackage`-Objekte bündeln die ausgelesenen Sensordaten mit zusätzlichen Metainformationen. Abb. 5–9 zeigt das zugehörige Klassendiagramm.

Tab. 5–5: Zu implementierende Methoden der `FramePackage`-Klasse

Methode	Rückgabewert	Format/Einheit
`cv::Mat getColorImageUnfiltered()`	Farbbild aus Sicht eines gemeinsamen Ursprungs	CV_8UC3, BGR
`cv::Mat getDepthImageUnfiltered()`	Tiefenbild aus Sicht eines gemeinsamen Ursprungs	CV_16U
`double getTimestampSecs()`	Zeitstempel der Aufnahme des Datenpakets	s
`unsigned int getDepthImageScale()`	Skalierungsfaktor der Tiefenbildwerte	1/m
`unsigned int getImageWidth()`	Gemeinsame Breite des Farb- und Tiefenbilds	px
`unsigned int getImageHeight()`	Gemeinsame Höhe des Farb- und Tiefenbilds	px
`Intrinsics getImageIntrinsics()`	Gemeinsame intrinsische Parameter des Farb- und Tiefenbilds	px

Jedes `FramePackage`-Objekt muss die gespeicherten Sensordaten als Kombination aus Farb- und Tiefenbild bereitstellen. Dazu müssen in jeder von `FramePackage` abgeleiteten

Klasse die in Tab. 5–5Tab. 5–5: Zu implementierende Methoden der `FramePackage`-Klasse aufgelisteten Interface-Methoden implementiert werden.

Für die implementierte Klasse `FramePackageRGBD` werden wir Daten der aufgenommenen Farb- und Tiefenbilder in `CameraFrame`-Objekten gespeichert. Jedes `FramePackageRGBD`-Objekt enthält ein `CameraFrameColor`-Objekt, in dem die Bild- und Metadaten des Farbbilds abgelegt werden, und ein `CameraFrameDepth`-Objekt, welches die Bild- und Metadaten des Tiefenbilds enthält. In der `FramePackageRGBD`-Klasse wird in den Interface-Methoden auf die Daten aus den `CameraFrame`-Objekten zugegriffen.

Soll ein zusätzlicher Nicht-RGB-D-Sensor integriert werden, kann eine neue Unterklasse von `FramePackage` erstellt werden, die die Sensordaten in sich aufnimmt. Solange die Klasse das `FramePackage`-Interface implementiert, ist sie mit dem Rest der Software kompatibel und die Daten können in der `SLAMPipeline` verarbeitet werden. Jeder Sensortyp, dessen Eingabedaten sich in ein Farb- und ein Tiefenbild umsetzen lassen, kann in das Softwareframework integriert werden.

Zusätzlich bietet die `FramePackage`-Klasse noch eine innere Klasse für `Filter`, die auf den `FramePackage`-Daten arbeiten. Von `FramePackage::Filter` abgeleitete Klassen müssen die in Tab. 5–6 aufgelisteten Interface-Methoden implementieren.

Tab. 5–6: Zu implementierende Methoden der `FramePackage::Filter`-Klasse

Methode	Rückgabewert	Format/Einheit
`cv::Mat getColorImageFiltered(const FramePackage & frame_package)`	Gefiltertes Farbbild des übergebenen `FramePackage`	CV_8UC3, BGR
`cv::Mat getDepthImageFiltered(const FramePackage & frame_package)`	Gefiltertes Tiefenbild des übergebenen `FramePackage`	CV_16U

5.6.2 Wrapper-Klassen

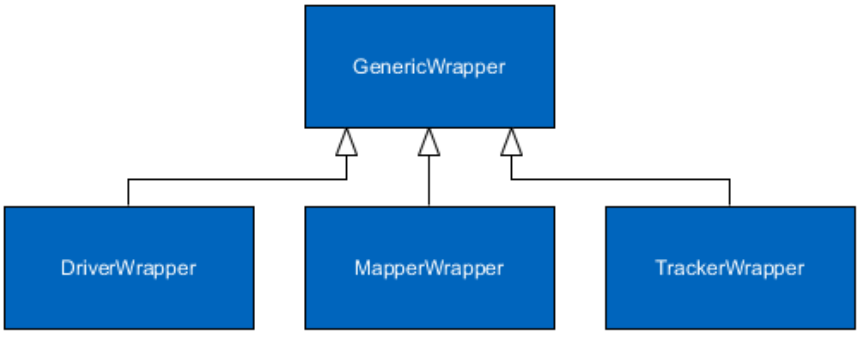


Abb. 5–10: Klassendiagramm der Wrapperklassen

Wrapper-Klassen existieren für das Driver-, Tracking- und Mapping-Modul. Das Interface dieser Klassen kommt mit einigen wenigen zu implementierenden Methoden aus. Die Module weisen außerdem eine einheitliche Kernfunktionalität auf, die in eine übergeordnete Wrapper-Klasse ausgelagert wurde. Abb. 5–10 zeigt das zugehörige Klassendiagramm.

5.6.2.1 GenericWrapper

Der `GenericWrapper` bündelt Kernfunktionalitäten der Wrapper-Klassen. Er verpackt die Funktionalität der Module in einen Zustandsautomaten, der eine zentrale, einheitliche Steuerung der Module und eine Verarbeitung der Eingabedaten in einem separaten Thread ermöglicht. Der Zustandsautomat ist in den Methoden `bool switchState(State next_state)` und `void runState()` implementiert. Abb. 5–11 zeigt das Zustandsübergangsdiagramm des Automaten. Der Zustandsübergang kann von der Benutzeroberfläche aus gesteuert werden, wo auch der aktuelle Zustand aller Module angezeigt wird. Zusätzlich beinhaltet der `GenericWrapper` die Grundstruktur für die Einstellungsmöglichkeiten der Module.

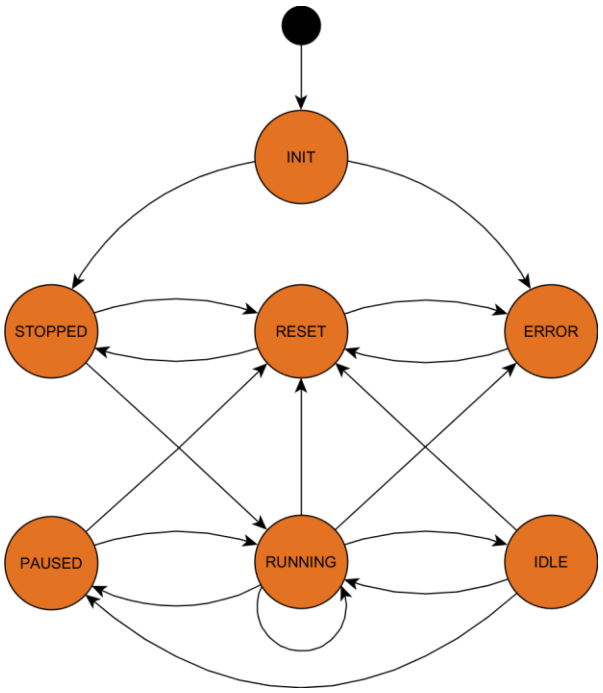


Abb. 5–11: Zustandsübergangsdiagramm des Zustandsautomaten der Module

5.6.2.2 DriverWrapper

Die `DriverWrapper`-Klasse bietet ein einheitliches Interface für Sensoren, die als Datenquellen in die Software integriert werden sollen. Dazu müssen die in Tab. 5–7 aufgelisteten Methoden implementiert werden. Die spezifische Implementierung kann frei gestaltet werden, so lange das Interface eingehalten wird. Innerhalb der Methoden werden im Allgemeinen die Funktionen der Sensor-SDK-APIs aufgerufen.

Tab. 5–7: Zu implementierende Methoden der `DriverWrapper`-Klasse

Methode	Beschreibung
`bool getFramePackage(std::shared_ptr<FramePackage>& frame_package)`	Gibt über den Referenzparameter einen Verweis auf ein neues `FramePackage`-Objekt zurück.
`bool doReset()`	Setzt den `DriverWrapper` zurück.

5.6.2.3 TrackerWrapper

Die `TrackerWrapper`-Klasse bietet ein einheitliches Interface für Lageschätzungsalgorithmen, die in die Pipeline zur Verarbeitung der Sensordaten integriert werden sollen. Dazu müssen die in Tab. 5–8 aufgelisteten Methoden implementiert werden. Die spezifische Implementierung kann frei gestaltet werden, so lange das Interface eingehalten wird. Innerhalb der Methoden werden die Funktionen der API der Algorithmen aufgerufen.

Tab. 5–8: Zu implementierende Methoden der `TrackerWrapper`-Klasse

Methode	Beschreibung
`void computeNextPose(const FramePackage & frame_package)`	Berechnet die nächste Schätzung der Relativlage auf Basis des übergebenen `FramePackage`-Objekts.
`Eigen::Affine3d getPose()`	Gibt die aktuelle Lageschätzung zurück.
`bool doReset()`	Setzt den `TrackerWrapper` zurück.

5.6.2.4 MapperWrapper

Die `MapperWrapper`-Klasse bietet ein einheitliches Interface für 3D-Rekonstruktionsalgorithmen, die in die Pipeline zur Verarbeitung der Sensordaten integriert werden sollen. Dazu müssen die in Tab. 5–9 aufgelisteten Methoden implementiert werden. Die spezifische Implementierung kann frei gestaltet werden, so lange das Interface eingehalten wird. Innerhalb der Methoden werden die Funktionen der API der Algorithmen aufgerufen.

Tab. 5–9: Zu implementierende Methoden der `MapperWrapper`-Klasse

Methode	Beschreibung
`void updateMap(const FramePackage & frame_package)`	Aktualisiert die interne Umgebungsrepräsentation auf Basis des übergebenen `FramePackage`-Objekts.
`std::shared_ptr<pcl::PolygonMesh> getMesh()`	Gibt eine Darstellung der internen Umgebungsrepräsentation als 3D-Mesh zurück.
`std::shared_ptr<pcl::PointCloud<pcl::PointXYZRGB>> getPointCloud()`	Gibt eine Darstellung der internen Umgebungsrepräsentation als Punktwolke zurück.
`bool doReset()`	Setzt den `TrackerWrapper` zurück.

5.6.3 Externe Sensordatensätze

Externe, nicht mit RVDSLAM aufgezeichnete Datensätze können ebenfalls in die Software geladen werden. Der Datensatz muss dazu in einer Form angelegt werden, der mit der „virtuellen“ Sensoren `RGBDFileReader` und `RGBDVirtualCamera` kompatibel ist (siehe Abschnitt 5.5.2.2).

Die Struktur eines solchen Datensatzes muss wie folgt aufgebaut sein:

- Eine Datei `rgb.txt` mit einer Zeile pro Bild, jeweils mit dem Zeitstempel in Sekunden gefolgt vom relativen Pfad zum zugehörigen Farbbild als 8-bit-RGB-PNG-Datei. Beispiel:

```
# color images
# line format: timestamp filename
1341841873.441849 rgb/1341841873.441849.png
1341841873.473830 rgb/1341841873.473830.png
1341841873.505817 rgb/1341841873.505817.png
[...]
```

- Eine Datei `depth.txt` mit einer Zeile pro Bild, jeweils mit dem Zeitstempel in Sekunden gefolgt vom relativen Pfad zum zugehörigen Tiefenbild als 16-bit-Grauwert-PNG-Datei. Beispiel:

```
# depth images
# line format: timestamp filename
1341841873.441855 depth/1341841873.441855.png
1341841873.473847 depth/1341841873.473847.png
1341841873.505863 depth/1341841873.505863.png
[...]
```

- Optional eine Datei `groundtruth.txt` mit einer Zeile pro Zeitschritt, jeweils mit dem Zeitstempel in Sekunden gefolgt von der wahren Sensorlage als Quaternion. Beispiel:

```
# ground truth trajectory
# line format: timestamp tx ty tz qx qy qz qw
1341841873.4151 -2.1837 0.6821 1.4854 -0.2651 0.7889 -0.5183 0.1970
1341841873.4251 -2.1802 0.6824 1.4860 -0.2631 0.7903 -0.5176 0.1958
1341841873.4351 -2.1767 0.6826 1.4864 -0.2609 0.7909 -0.5182 0.1948
[...]
```

- Die einzelnen Zeilen müssen zeitlich zusammengehören, d.h. Zeile 1 aus `rgb.txt` korrespondiert zu Zeile 1 aus `depth.txt` und Zeile 1 aus `groundtruth.txt`, etc.
- Die Dateien können Kommentarzeilen enthalten, gekennzeichnet durch ein ‚#' als erstes Zeichen.
- Für die Farb- und Tiefenbilder ist zu empfehlen, sie in je einem Unterordner abzulegen, bspw. `rgb` und `depth`.

6 Ergebnisse

6.1 Algorithmenauswahl

Die Algorithmen wurden entsprechend den gegebenen Anforderungen ausgewählt (siehe Kapitel 4). Dabei konnte für die Anforderungen **1-01** bis **1-08** im Vorhinein überprüft werden, ob die Algorithmen die gewünschten Eigenschaften besitzen. Als Algorithmus zur Lageschätzung wurde DIFODO ausgewählt, zur 3D-Rekonstruktion kommt fastfusion zum Einsatz. Beide sind für x86-CPUs ausgelegt (Anforderung **1-01**), nicht an spezielle Sensorhardware gebunden (Anforderung **1-02**), öffentlich als Programmbibliothek bzw. als Quellcode mit entsprechenden Schnittstellen verfügbar (Anforderung **1-03**) und für den Einsatz unter Windows geeignet (Anforderung **1-04**). DIFODO arbeitet auf Tiefenbildern (Anforderung **1-05**) und erzeugt eine Schätzung der Relativlage zwischen Sensor und Zielobjekt als Ausgabe (Anforderung **1-06**). Diese geschätzte Relativlage dient wiederum zusammen mit Farb- und Tiefenbild als Eingabe für den fastfusion-Algorithmus (Anforderung **1-07**), der als Ausgabe ein 3D-Mesh des Zielobjekts generiert (Anforderung **1-08**). Der ebenfalls als Kandidat zur Lageschätzung gehandelte Algorithmus dvo_slam konnte leider auf den ersten Blick Anforderung **1-03** nicht erfüllen, da er stark in ROS integriert und dessen spezifische Datentypen verwendet.

Aufgrund der zugehörigen Veröffentlichungen der ausgewählten Algorithmen konnte die vorläufige Annahme getroffen werden, dass diese die Anforderungen **1-09** bis **1-11** zur Performance, Qualität und Robustheit ebenfalls erfüllen. Zur abschließenden Verifizierung dieser Anforderungen wurden mithilfe der entwickelten Software einige Testdatensätze aufgezeichnet (siehe Abschnitt 6.3) und verarbeitet. Die Ergebnisse werden in Abschnitt 6.4 vorgestellt. Tab. 6–1 zeigt eine Übersicht über die Erfüllung der Anforderungen.

Tab. 6–1: Übersicht über die Erfüllung der Algorithmenanforderungen

ID	Resultat
1-01	erfüllt
1-02	erfüllt
1-03	teilweise erfüllt
1-04	erfüllt
1-05	erfüllt
1-06	erfüllt
1-07	erfüllt
1-08	erfüllt
1-09	teilweise erfüllt
1-10	teilweise erfüllt
1-11	erfüllt

6.2 Softwarefunktionen

Im Rahmen dieser Arbeit wurde das Softwareframework RVDSLAM zur Lageschätzung und 3D-Rekonstruktion in einem On-Orbit-Servicing-Szenario entwickelt (siehe Kapitel 5). Es beherrscht die Kette von der Dateneingabe mittels Sensoren der visuellen Odometrie (Anforderung **2-01**) über die Schätzung und Visualisierung der Relativlage zwischen Sensor und Zielobjekt (Anforderungen **2-06** und **2-08**) bis zur Berechnung und Visualisierung einer 3D-Darstellung (Anforderungen **2-10** und **2-12**). Sowohl die aufgenommenen Datensätze als auch die berechnete Trajektorie und die erzeugte 3D-Darstellung können abgespeichert werden, entweder zur späteren erneuten Verwendung (Anforderungen **2-03** und **2-04**) oder zur externen Weiterverarbeitung (Anforderungen **2-09** und **2-13**).

Als spezifische Sensoren wurden die Kinect v1 und die Kinect v2 sowie die Algorithmen DIFODO und fastfusion eingebunden (Anforderungen **2-02**, **2-07** und **2-11**). Ein Filter zur Hintergrundunterdrückung wurde integriert (Anforderung **2-05**). Die Verarbeitung der aufgenommenen Daten kann entweder im Echtzeitmodus erfolgen, bei dem immer nur der zuletzt aufgenommene Datensatz verarbeitet wird, oder im Gesamtdatenmodus, in dem alle Datensätze verwendet werden (Anforderung **2-15**).

Die flexible Erweiterbarkeit um weitere Datenquellen (Sensoren) und Algorithmen ist durch den Einsatz von Wrapperklassen mit einfachen Schnittstellen und einer Aufteilung der Funktionalitäten in abgegrenzte Module gegeben (Anforderung **2-16**). Diese Modularisierung ermöglicht eine Ausführung der einzelnen Komponenten in getrennten Threads (Anforderung **2-14**) und einen Wechsel zur Laufzeit gegen andere Komponenten mit den passenden Schnittstellen. Der Visualisierungsteil wurde entkoppelt von der restlichen Programmlogik implementiert, um den restlichen Teil der Software später in eine am Lehrstuhl für Raumfahrttechnik entwickelte Teleoperationsumgebung integrieren zu können (Anforderung **2-17**).

Die Software wurde als Windows-Anwendung entwickelt (Anforderung **2-18**). Dabei wurde auf etablierte Programmbibliotheken zurückgegriffen (Anforderung **2-19**), um eine schnellere und robustere Implementierung und eine vereinfachte Weiterentwicklung zu ermöglichen.

Tab. 6–2 zeigt eine Übersicht über die Erfüllung der Anforderungen.

Tab. 6–2: Übersicht über die Erfüllung der Softwareanforderungen

ID	Resultat	ID	Resultat
2-01	erfüllt	**2-11**	erfüllt
2-02	erfüllt	**2-12**	erfüllt
2-03	erfüllt	**2-13**	erfüllt
2-04	erfüllt	**2-14**	erfüllt
2-05	erfüllt	**2-15**	erfüllt
2-06	erfüllt	**2-16**	erfüllt
2-07	erfüllt	**2-17**	erfüllt
2-08	erfüllt	**2-18**	erfüllt

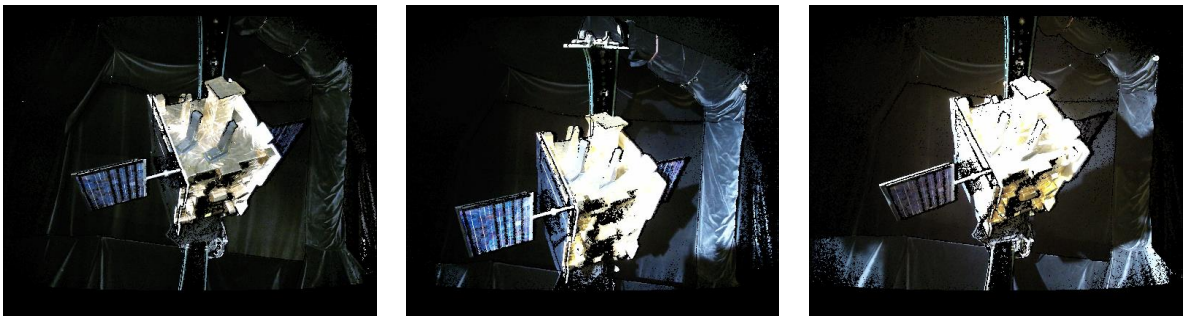
2-09	erfüllt	**2-19**	erfüllt
2-10	erfüllt		

6.3 Testdatensätze

Für eine qualitative Analyse der Rekonstruktionsqualität und -geschwindigkeit, die mit der implementierten Verarbeitungspipeline erzielt werden kann, wurden drei Testdatensätze zum gegebenen Referenzszenario (siehe Abschnitt 3.1) mit verschiedenen Beleuchtungsverhältnisse aufgenommen und ausgewertet. Eine Übersicht über das Szenario bietet Tab. 6–3. Abb. 6–1 zeigt Beispielbilder aus den Datensätzen. Ein Datensatz umfasst jeweils eine Umdrehung des Zielobjekts.

Tab. 6–3: Eigenschaften des Referenzszenarios

Eigenschaft	Wert
Größe des Zielobjekts	60 cm x 70 cm x 100 cm
Abstand zw. Sensor und Zielobjekt	2 m
Ausrichtung des Zielobjekts	z-Achse diagonal in Richtung des Sensors geneigt
Rotationsachse des Zielobjekts	z-Achse
Rotationsgeschwindigkeit des Zielobjekts	2 °/s
Beleuchtung	Laborbeleuchtung, künstliche Sonne hinter Sensor & links neben Sensor



(a) Laborbeleuchtung (b) künstliche Sonne hinter dem Sensor (c) künstliche Sonne seitlich links neben dem Sensor

Abb. 6–1: Beispielbilder aus den aufgezeichneten Datensätzen

Zur Aufzeichnung der Testdatensätze wurden die Kinect v2 als Sensor, der DIFODO-Algorithmus zur Lageschätzung und der fastfusion-Algorithmus zur 3D-Rekonstruktion verwendet. Die entsprechenden Eigenschaften und Parameter sind in Tab. 6–4, Tab. 6–5 und Tab. 6–6 aufgelistet.

Tab. 6–4: Kinect-v2-Eigenschaften

Eigenschaft	Wert
Auflösung	512 x 424 px
Bildwiederholrate	15 Hz
Brennweite horiz.	366,2 px
Brennweite vert.	366,2 px
Hauptpunkt horiz.	259,7 px
Hauptpunkt vert.	202,2 px

Tab. 6–5: Verwendete DIFODO-Parameter

Parameter	Wert
downsample	1
ctf_levels	4
max_depth	5,0 [m]

Tab. 6–6: Verwendete fastfusion-Parameter

Parameter	Wert
voxel_scale	0,005 [m]
max_depth	5,0 [m]

Zur Unterdrückung des Laborhintergrundes kam der implementierte `BoxFilter` zum Einsatz (siehe Abschnitt 5.5.2.3). Tab. 6–7 zeigt die verwendeten Parameter.

Tab. 6–7: Verwendete `BoxFilter`-Parameter

Parameter	Wert
u_min	40 [px]
u_max	450 [px]
v_min	100 [px]
v_max	424 [px]
z_min	0 [m]
z_max	2,3 [m]

6.4 Rekonstruktionsqualität

Abb. 6–2 zeigt die resultierende Darstellung des rekonstruierten Zielobjekts als 3D-Mesh bei verschiedenen Beleuchtungssituationen und Umdrehungswinkeln. Grundsätzlich ist die Satellitenform des rekonstruierten Objekts grob zu erkennen. Der Hauptkörper kann klar von den Solarzellenflügeln unterschieden werden. Die Abmaße und Proportionen lassen sich ansatzweise erkennen. Generell scheint das gegebene Szenario jedoch problematisch für die Algorithmen zu sein. Die Geometrie ist ungenau rekonstruiert und die Oberflächen sind zerklüftet, Formdetails sind verloren gegangen.

Über die Zeitachse von links nach rechts klar zu sehen ist das Problem des Drifts der Relativlageschätzung. Die Objektgeometrie wird ungenauer und Formdetails verwischen, wenn die tatsächliche Lage von der geschätzten immer weiter abweicht. Abb. 6–3 zeigt die berechnete Sensortrajektorie für das Szenario aus Abb. 6–1(b) über eine Drehung um 360°. Anfangs- und Endposition sollten an einem Ort liegen, weichen jedoch deutlich voneinander ab. Anforderung **1-10** ist somit nur teilweise erfüllt.

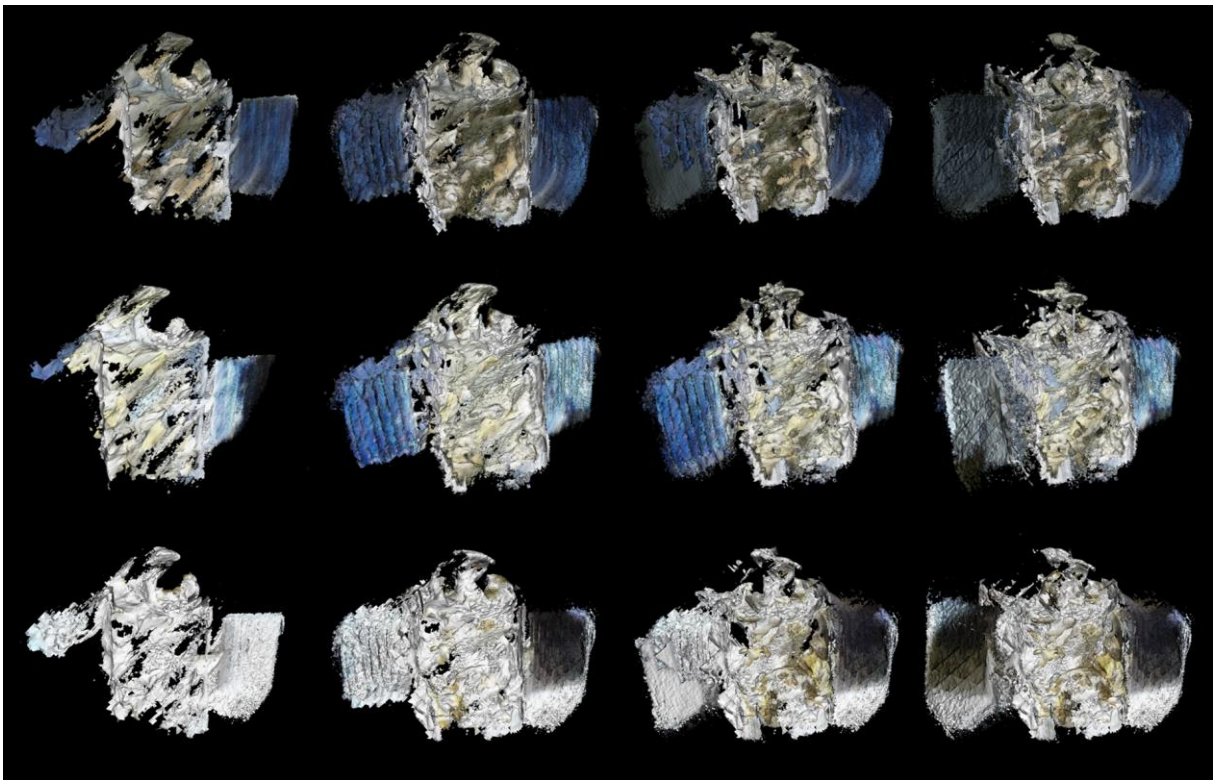


Abb. 6–2: Rekonstruktionsergebnisse – Beleuchtung von oben nach unten: Laborbeleuchtung, Sonne hinten, Sonne links – Drehwinkel des Modells von links nach rechts: 45°, 90°, 180°, 270°

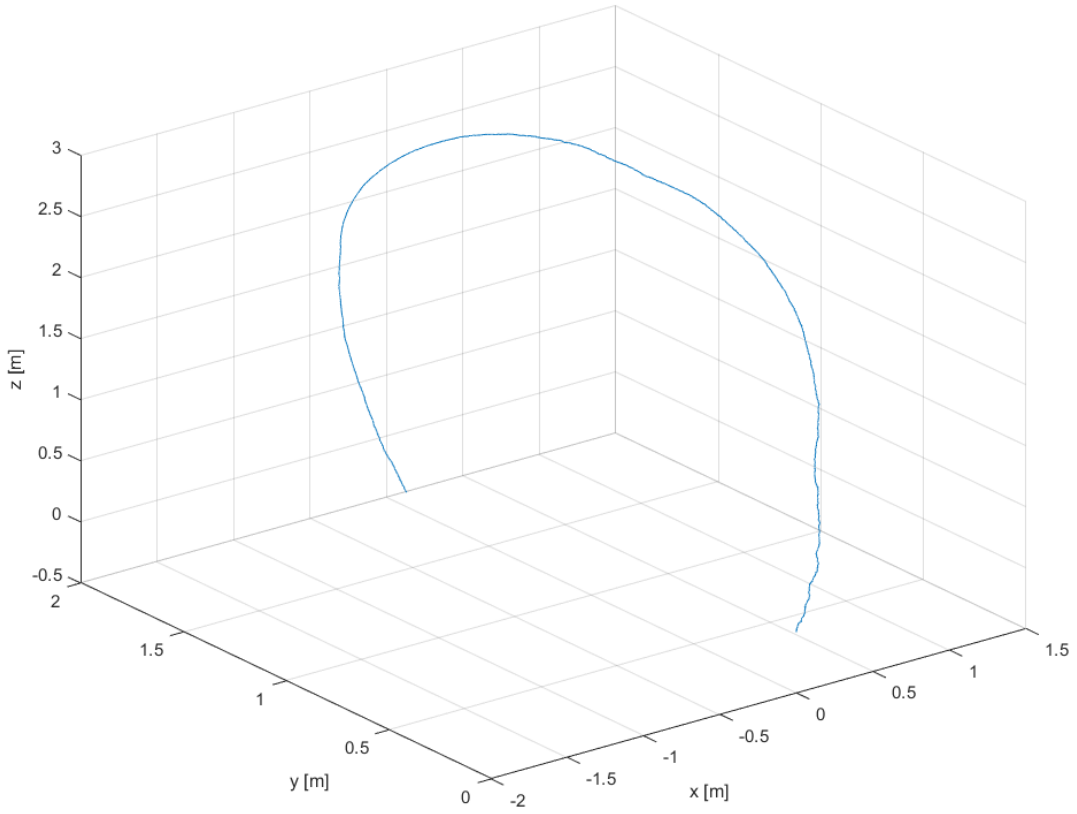
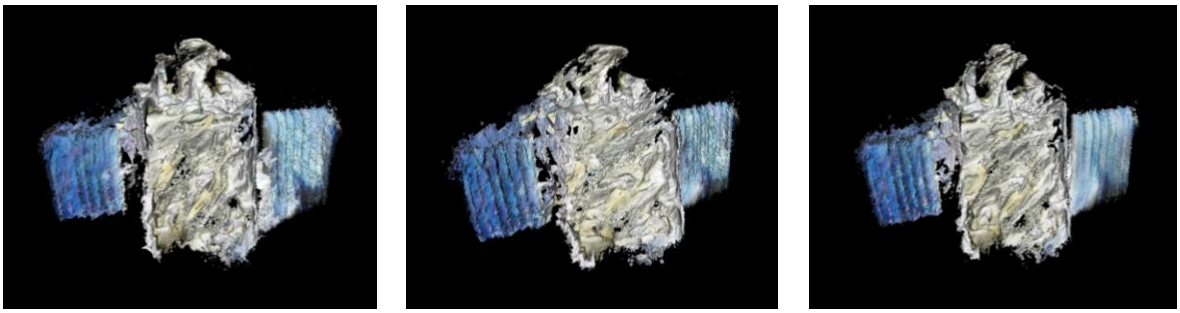


Abb. 6–3: Durch den Tracking-Algorithmus berechnete Relativtrajektorie zwischen Chaser und Target

Die Farbdarstellung ist stark abhängig vom Sonneneinstrahlwinkel. Gut zu erkennen ist die Überbelichtung bei Sonneneinstrahlung von links. Die Rekonstruktionsqualität der Form des Zielobjekts scheint dagegen weitgehend unabhängig von der Sonneneinstrahlung zu sein (Anforderung **1-11**) – im vertikalen Vergleich ähneln sich die Darstellungen in Proportionen und Details.

6.5 Parametereinfluss

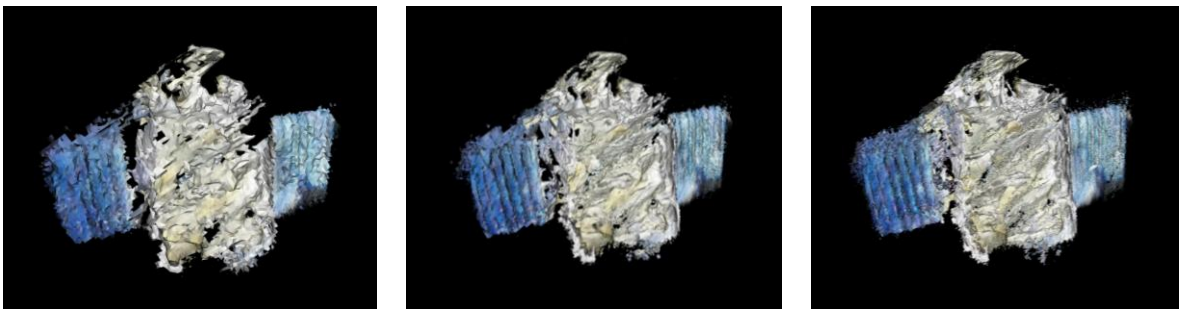
Die Anzahl der verwendeten Ebenen der Gaußfilterpyramide im DIFODO-Algorithmus (siehe Abschnitt 4.3.2) scheint um den Standardwert herum keinen signifikanten Einfluss auf die Rekonstruktionsqualität im gegebenen Szenario zu haben. Abb. 6–4 zeigt die 3D-Rekonstruktionen nach 90° Umdrehungswinkel bei der Verwendung von drei, vier und fünf Pyramidenebenen.



(a) Drei Pyramidenebenen (b) Vier Pyramidenebenen (Standard) (c) Fünf Pyramidenebenen

Abb. 6–4: Einfluss der Anzahl der Ebenen der Gaußfilterpyramide des DIFODO-Algorithmus auf die Rekonstruktionsqualität

Die im fastfusion-Algorithmus gewählte Voxelgröße (siehe Abschnitt 4.5.2) hat dagegen einen deutlichen Einfluss auf die Rekonstruktionsqualität. Abb. 6–5 zeigt die 3D-Rekonstruktionen nach 90° Umdrehungswinkel bei einer Voxelgröße von zehn, fünf und drei Millimetern. Eine Größe von zehn Millimetern lässt die Rekonstruktion sehr grob erscheinen. Bei drei Millimetern wird dagegen kaum eine Verbesserung gegenüber der Standardeinstellung von fünf Millimetern erzielt. Die Voxelgröße hat jedoch eine signifikante Auswirkung auf die Rechenzeit (siehe Abschnitt 6.6).



(a) Voxelgröße 10 mm (b) Voxelgröße 5 mm (Standard) (c) Voxelgröße 3 mm

Abb. 6–5: Einfluss der Voxelgröße des fastfusion-Algorithmus auf die Rekonstruktionsqualität

6.6 Ausführungsgeschwindigkeit

Für den DIFODO-Algorithmus wurden unabhängig von der gewählten Anzahl an Ebene der Gaußfilterpyramide im gegebenen Szenario und mit den gegebenen Einstellungen auf einem Intel Core i7-5820K Prozessor bei 3,30 GHz eine Verarbeitungsgeschwindigkeit von 30 Bildern pro Sekunde erzielt.

Die Geschwindigkeit des fastfusion-Algorithmus ist dagegen stark abhängig von der gewählten Voxelgröße. Tab. 6–8 zeigt die erzielten Bildwiederholraten. Bei den gegebenen Standardeinstellungen ist die Geschwindigkeit der Verarbeitungspipeline somit durch die 3D-Rekonstruktionskomponente beschränkt. Eine Echtzeitverarbeitung (Anforderung **1-09**) ist möglich, jedoch abhängig von den gewählten Paramern.

Tab. 6–8: fastfusion-Verarbeitungsgeschwindigkeit (Intel Core i7-5820K @ 3,30 GHz)

Voxelgröße	Bilder pro Sekunde
0,010 m	35
0,005 m	12
0,003 m	5

7 Diskussion

7.1 Softwareframework

Die zu Beginn gesetzten Anforderungen an die Software (Abschnitt 5.1) konnten vollständig umgesetzt werden (siehe Abschnitt 6.2). Die entwickelte Software setzt die Kernfunktionalität um und bietet eine Basis für zukünftige Weiterentwicklungen. Die Softwarestruktur kapselt die einzelnen Komponenten klar voneinander ab, was Erweiterungen und Änderungen einfach macht.

Die Flexibilität der Verarbeitungspipeline ist durch austauschbare Algorithmen gegeben, jedoch ist die Struktur der Pipeline noch recht starr, bestehend aus einem Einlese-, einem Lageschätzungs- und einem Rekonstruktionsschritt. Volle Flexibilisierung wäre erreicht, wenn in der Pipeline Komponenten beliebig miteinander verschaltet werden könnten, so lange die Schnittstellen passend sind.

Der dvo_slam-Algorithmus ließ sich im Rahmen dieser Arbeit nicht in das Framework integrieren. Der Algorithmus verspricht grundsätzlich robustere und genauere Ergebnisse bei der Lageschätzung als DIFODO durch die zusätzliche Verwendung von Farbinformationen und Schleifenerkennung. Ersteres könnte jedoch zu Problemen durch die Sonneneinstrahlung führen, letzteres durch die Symmetrie des Zielobjekts.

Geschwindigkeit und Speicherverbrauch des Frameworks sind noch nicht optimal. Die eingelesenen Daten werden gesammelt im Arbeitsspeicher abgelegt, was schnell zu Engpässen führen kann. Hier sind für die Zukunft noch weitere Optimierungen vorzunehmen.

7.2 Rekonstruktionsgüte

Die Betrachtung der Rekonstruktionsgüte wurde im Rahmen dieser Arbeit auf eine erste qualitative Analyse beschränkt. Aus den generierten Darstellungen der 3D-Rekonstruktion wurde ersichtlich, dass sich die Algorithmenkombination aus DIFODO und fastfusion grundsätzlich zur Lageschätzung und 3D-Rekonstruktion im vorgegebenen Szenario eignet. Der Lageschätzungsalgorithmus funktioniert trotz schwieriger Beleuchtungssituation, die jedoch teilweise für eine unrealistische visuelle Darstellung sorgt.

Die geometrische Qualität ist in Anbetracht der schwierigen Bedingungen (im Vergleich zu einer gut beleuchteten, statischen Innenraumszene) zufriedenstellend. Es wird ein Objekt mit passenden Proportionen ohne Artefakte außerhalb des aufgenommenen Körpers rekonstruiert. Die Oberfläche ist jedoch stark zerklüftet und die Rekonstruktion unterliegt einem kontinuierlichen Drift. Das Objekt wird deshalb nach einiger Zeit verzerrt dargestellt. Für den Einsatz in Teleoperationsaufgaben sind diese Abweichungen problematisch. Deshalb sollte für die Lageschätzung ein Algorithmus mit erweiterten SLAM-Funktionalitäten zum Einsatz kommen (bspw. dvo_slam, siehe Abschnitt 4.4).

Die Verarbeitungsgeschwindigkeit der Algorithmen entspricht noch nicht den in den zugehörigen Veröffentlichungen angegebenen. Hier muss noch weitere Optimierungsarbeit bei der Portierung bzw. Einbindung in das entwickelte Framework geleistet werden.

8 Zusammenfassung

In dieser Arbeit wurden erste Schritte hin zu einem System zur Lageschätzung und 3D-Rekonstruktion mit RGB-D-Sensoren von Objekten in OOS-Szenarien im Nahbereich unternommen. Dabei wurden aktuelle State-of-the-Art-Algorithmen aus diesen Bereichen ausgewählt und vorgestellt.

Da sich das zu verfolgende Objekt vor einem weit entfernten Hintergrund befindet, einen großen Teil des Sichtfeldes einnimmt und nur die Relativbewegung zwischen Sensor und Objekt von Interesse ist, müssen keine speziellen Algorithmen zur Erfassung und Verfolgung von Objekten eingesetzt werden. Es genügen Algorithmen zur Lageschätzung in einer statischen Umgebung. Aktuelle Ansätze verwenden die gesamte Bildinformation statt einzelner Merkmalspunkte. Sie betrachten entweder nur das Tiefenbild oder zusätzlich die Farbbildinformationen. Dabei sind sie annähernd in Echtzeit auf aktuellen Desktop-CPUs lauffähig. Für diese Arbeit wurde der DIFODO-Algorithmus (Jaimez und González-Jiménez 2015) ausgewählt und getestet.

Zur 3D-Rekonstruktion bieten sich Algorithmen an, die die Struktur der Umgebung mittels einer Abstandsfunktion in einer voxelbasierten Octree-Datenstruktur speichern. Neue Datensätze lassen sich mit verhältnismäßig geringem Rechen- und Speicheraufwand in diese Datenstruktur integrieren und verbessern die Genauigkeit der Darstellung. Aus dieser Repräsentation kann zur Betrachtung ein dreidimensionales Mesh erzeugt werden. Befehlssatzoptimierte Implementierungen erreichen zur Integration neuer Datensätze auch hier Echtzeitgeschwindigkeit. Ausgewählt wurde der fastfusion-Algorithmus (Steinbrücker et al. 2014).

Zusätzlich wurde ein Softwareframework entwickelt, das als Grundlage für den zukünftigen Vergleich dieser und anderer Algorithmen und Sensortypen dienen soll. Über eine flexible Pipelinestruktur und Wrapperklassen mit klar definierten Schnittstellen für Algorithmen und Datenquellen wurde eine einfache Erweiterbarkeit und vielfältige Einsetzbarkeit sichergestellt. Durch die Verwendung von etablierten, plattformübergreifenden und quelloffenen Bibliotheken wird die Einarbeitung in die Softwarestruktur und die Möglichkeit zur Weiterentwicklung der Codebasis erleichtert. Die einzelnen Hauptkomponenten wurden in Threads ausgelagert, um die Performance zu erhöhen und die Dateneingabe von der Verarbeitung zu entkoppeln.

Die Software bietet die Möglichkeit, die Eingabedaten eines Sensors in einem Echtzeitmodus zur Live-Darstellung und einem Gesamtdatenmodus für maximale Rekonstruktionsqualität zu verarbeiten. Die berechnete 3D-Darstellung des Zielobjekts kann direkt in der Software betrachtet und evaluiert werden. Datensätze lassen sich speichern und wieder einlesen, Trajektorien und Meshes lassen sich zur externen Weiterverarbeitung exportieren.

Mithilfe des entwickelten Frameworks wurde die Pipeline aus den ausgewählten Algorithmen und der Kinect v2 als RGB-D-Sensor in einem Referenzszenario getestet. Die geometrische Qualität der Rekonstruktion erwies sich als zufriedenstellend, die grundsätzliche Form des Satelliten war erkennbar. Durch die fehlende Schleifenerkennung im getesteten Algorithmus stellten sich jedoch schnell Verzerrungen durch Drift in den Lageschätzungen ein. Erweiterte SLAM-Funktionalitäten wie Schleifenerkennung und globale Lageoptimierung sind deshalb nötig, um diesen Drift zu eliminieren und die Rekonstruktionen nutzbar zu machen.

9 Ausblick

Die vorgenommenen Algorithmentests waren im Rahmen dieser Arbeit nur von qualitativer Natur. Hierzu sollten detailliertere, quantitative Tests vorgenommen werden. Dazu ist zuallererst die Festlegung von entsprechenden Metriken notwendig, sowohl bezüglich der Rekonstruktionsqualität als auch der Laufzeitperformance.

Der Einfluss auf die Rekonstruktionsqualität durch Sonneneinstrahlung bzw. Erdalbedo sollte genauer untersucht werden. Evtl. könnte hier versucht werden, bei einem farbbildverarbeitenden Lageschätzalgorithmus statt dem Farbbild das vom Tiefensensor aufgenommene Infrarotbild zu verwenden, welches bspw. bei der Kinect v2 ausgelesen werden kann.

Bei der Untersuchung einzelner Algorithmen ist der Vergleich verschiedener Einstellungen einzelner Algorithmen und ihr Einfluss auf Rekonstruktionsqualität bzw. Geschwindigkeit interessant. Auch wäre es aus Anwendungssicht relevant, zu ermitteln, bis zu welcher maximalen Relativrotationsgeschwindigkeit eine 3D-Rekonstruktion möglich ist.

Darüber hinaus sollten verschiedene Algorithmen und Sensortypen miteinander verglichen werden. Zu empfehlen ist zuerst eine Portierung des dvo_slam-Algorithmus zur Verwendung im entwickelten Softwareframework, da dieser einige erweiterte Funktionen zur Driftreduzierung beinhaltet. Auch eine Einbindung GPU-beschleunigter Algorithmen könnte interessant sein, da hier einiges an Laufzeitperformance gewonnen werden kann. Was weitere Sensoren angeht, bietet sich ein Vergleich mit Stereokameras und den zugehörigen Algorithmen an.

Zur Verbesserung der Lageschätzungs- und Rekonstruktionsqualität könnte eine genaue Bestimmung der intrinsischen Parameter und der Verzerrungsparameter und damit eine entsprechende Korrektur der Eingabedaten nützlich sein. Ein Vergleich mit den Daten der Werkskalibrierung, die aus der Kinect-Kamera ausgelesen werden können, bietet sich an.

Neben den algorithmenseitigen Untersuchungen existieren auch einige Möglichkeiten zur Weiterentwicklung des Softwareframeworks. Die Verarbeitungspipeline könnte weiter flexibilisiert werden, indem ermöglicht wird, beliebige Berechnungs-komponenten aneinanderzureihen, solange die Ein- und Ausgabedatenformate passend sind. Weitere Verbesserungsmöglichkeiten bestehen im Bereich der Verarbeitungsgeschwindigkeit und der Speichereffizienz.

Auch die Visualisierung der Meshdaten kann weiterentwickelt werden, hin zu einer nützlichen Darstellung für einen Teleoperator. Das generierte 3D-Modell könnte beispielsweise weiterverarbeitet werden, indem daraus Flächen extrahiert werden und versucht wird, die abstrahierte Geometrie des erfassten Objekts zu visualisieren. Zum Test der Darstellungsqualität könnten Probandentests durchgeführt werden.

Zu guter Letzt würde es sich anbieten, den Kern der Software – die Rekonstruktionspipeline – auf einem Embedded-System zu testen, um näher an die Restriktionen einer tatsächlichen Weltraummission zu gelangen.

A Anhang

A.1 Literaturverzeichnis

Bay, Herbert; Ess, Andreas; Tuytelaars, Tinne; van Gool, Luc (2008): Speeded-Up Robust Features (SURF). In: *Comput. Vis. Image Underst.* 110 (3), S. 346–359. DOI: 10.1016/j.cviu.2007.09.014.

Besl, P. J.; McKay, H. D. (1992): A Method for Registration of 3-D Shapes. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14 (2), S. 239–256. DOI: 10.1109/34.121791.

Bylow, Erik; Sturm, Jürgen; Kerl, Christian; Kahl, Fredrik; Cremers, Daniel (2013): Real-Time Camera Tracking and 3D Reconstruction Using Signed Distance Functions. In: Robotics: Science and Systems.

Cavrois, Bruno; Vergnol, Alizée; Donnard, Alain; Casiez, Pierre; Mongrard, Olivier (2015): LIRIS Demonstrator on ATV5: A Step Beyond for European Non Cooperative Navigation System. In: AIAA Guidance, Navigation, and Control Conference: American Institute of Aeronautics and Astronautics (AIAA SciTech).

Choi, Changhyun; Christensen, Henrik I. (2013): RGB-D Object Tracking: A Particle Filter Approach on GPU. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, S. 1084–1091.

Dionnet, F.; Marchand, E. (2007): Robust Stereo Tracking for Space Applications. In: 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, S. 3373–3378.

Dziura, Martin; Wiese, Tim; Harder, Jan (2017): 3D Reconstruction in Orbital Proximity Operations. In: 2017 IEEE Aerospace Conference. Big Sky, MT.

Endres, F.; Hess, J.; Engelhard, N.; Sturm, J.; Cremers, D.; Burgard, W. (2012): An Evaluation of the RGB-D SLAM System. In: Robotics and Automation (ICRA), 2012 IEEE International Conference on, S. 1691–1696.

Endres, F.; Hess, J.; Sturm, J.; Cremers, D.; Burgard, W. (2014): 3D Mapping with an RGB-D Camera. In: *IEEE Transactions on Robotics* 30 (1), S. 177–187. DOI: 10.1109/TRO.2013.2279412.

Enqvist, Olof (2009): Correspondence Problems in Geometric Vision. Lund: Centre for Mathematical Sciences, Mathematics, Lund University (Licentiate theses in mathematical sciences, 2009:2).

Fankhauser, Péter; Bloesch, Michael; Rodriguez, Diego; and Kaestner, Ralf; Hutter, Marco; Siegwart, Roland (2015): Kinect v2 for Mobile Robot Navigation: Evaluation and Modeling. In: IEEE International Conference on Advanced Robotics (ICAR).

Fischler, Martin A.; Bolles, Robert C. (1981): Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. In: *Commun. ACM* 24 (6), S. 381–395. DOI: 10.1145/358669.358692.

Fisher, Robert B.; Konolige, Kurt (2008): Chapter 22 - Range Sensors. In: Bruno Siciliano und Oussama Khatib (Hg.): Springer Handbook of Robotics. Berlin: Springer (Gale virtual reference library).

Fleischner, Andreas; Wilde, Markus; Walter, Ulrich (2012): Racoon - A Hardware-in-the-Loop Simulation Environment for Teleoperated Proximity Operations. In: i-SAIRAS.

Forsyth, David A.; Ponce, Jean (2002): Computer Vision: A Modern Approach: Prentice Hall Professional Technical Reference.

Gschwindt, Mirko (2016): Integration eines Echtzeit-Algorithmus zur Objektrekonstruktion aus Stereo-Bilddaten in eine Simulationsumgebung zur Nahbereichsnavigation im Orbit. Technische Universität München, Lehrstuhl für Raumfahrttechnik (LRT-BA 2016/03).

Harder, Jan; Dziura, Martin (2017): The RACOON Lab. Technologies for operating robots - in space. Technische Universität München, Lehrstuhl für Raumfahrttechnik, 2017. Online verfügbar unter https://www.lrt.mw.tum.de/fileadmin/w00brt/www/Forschung/Racoon/Racoon_Info_Deck_2017.pdf, zuletzt geprüft am 13.03.2017.

Hartley, R. I.; Zisserman, A. (2004): Multiple View Geometry in Computer Vision. Second: Cambridge University Press, ISBN: 0521540518.

Henry, Peter; Krainin, Michael; Herbst, Evan; Ren, Xiaofeng; Fox, Dieter (2012): RGB-D Mapping: Using Kinect-style Depth Cameras for Dense 3D Modeling of Indoor Environments. In: *The International Journal of Robotics Research* 31 (5), S. 647–663. DOI: 10.1177/0278364911434148.

Hirschmüller, Heiko; Schmid, Korbinian; Suppa, Michael (2015): Computer Vision for Mobile Robot Navigation. In: Photogrammetric Week '15: Wichmann, S. 143–154.

Horn, Berthold K.P.; Schunck, Brian G. (1980): Determining Optical Flow. Cambridge, MA, USA.

Huang, Albert S.; Bachrach, Abraham; Henry, Peter; Krainin, Michael; Fox, Dieter; Roy, Nicholas (2011): Visual Odometry and Mapping for Autonomous Flight Using an RGB-D Camera. In: Proceedings of the International Symposium of Robotics Research.

Izadi, Shahram; Kim, David; Hilliges, Otmar; Molyneaux, David; Newcombe, Richard; Kohli, Pushmeet et al. (2011): KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology. New York, NY, USA: ACM (UIST '11), S. 559–568. Online verfügbar unter http://doi.acm.org/10.1145/2047196.2047270.

Jaimez, Mariano; González-Jiménez, Javier (2015): Fast Visual Odometry for 3-D Range Sensors. In: *IEEE Transactions on Robotics* 31 (4), S. 809. DOI: 10.1109/TRO.2015.2428512.

Kerl, Christian; Sturm, Jürgen; Cremers, Daniel (2013a): Dense Visual SLAM for RGB-D Cameras. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on, S. 2100–2106.

Kerl, Christian; Sturm, Jürgen; Cremers, Daniel (2013b): Robust Odometry Estimation for RGB-D Cameras. In: ICRA.

Klein, Georg; Murray, David (2007): Parallel Tracking and Mapping for Small AR Workspaces. In: Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07). Nara, Japan.

Klodt, Maria; Sturm, Jürgen; Cremers, Daniel (2013): Scale-Aware Object Tracking with Convex Shape Constraints on RGB-D Images. In: German Conference on Pattern Recognition. Springer, S. 111–120.

Kolb, Florian M.; Windmüller, Michael; Rößler, Mario; Möbius, Bettina; Casiez, Pierre; Cavrois, Bruno; Mongrard, Olivier Mongrard (2015): The LIRIS-2 3D Imaging Lidar on ATV-5. In: Proceedings of ASTRA 2015. Noordwijk.

Korcz, Daniel (2011): Volumetric Range Image Integration. Diplomarbeit. Universität Stuttgart, Stuttgart. Institut für Photogrammetrie. Online verfügbar unter http://www.ifp.uni-stuttgart.de/lehre/diplomarbeiten/korcz/, zuletzt geprüft am 26.02.2017.

Krull, Alexander; Michel, Frank; Brachmann, Eric; Gumhold, Stefan; Ihrke, Stephan; Rother, Carsten (2014): 6-DOF Model Based Tracking via Object Coordinate Regression. In: Asian Conference on Computer Vision. Springer, S. 384–399.

Lachat, E.; Macher, H.; Mittet, M.-A.; Landes, T.; Grussenmeyer, P. (2015): First Experiences with Kinect v2 Sensor for Close Range 3D Modelling. In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, S. 93–100. DOI: 10.5194/isprsarchives-XL-5-W4-93-2015.

Lowe, David G. (1999): Object Recognition from Local Scale-Invariant Features. In: Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2. Washington, DC, USA: IEEE Computer Society (ICCV '99), S. 1150. Online verfügbar unter http://dl.acm.org/citation.cfm?id=850924.851523.

Maimone, Mark; Cheng, Yang; Matthies, Larry (2007): Two Years of Visual Odometry on the Mars Exploration Rovers. In: *Journal of Field Robotics, Special Issue on Space Robotics* 24, S. 2007.

Meilland, M.; Comport, A. I. (2013): Super-Resolution 3D Tracking and Mapping. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on, S. 5717–5723.

Montemerlo, Michael; Thrun, Sebastian; Koller, Daphne; Wegbreit, Ben (2002): FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In: AAAI Conference on Artificial Intelligence 2002.

Moore, E. H. (1920): On the reciprocal of the general algebraic matrix. In: *Bulletin of the American Mathematical Society* 26, S. 394–395.

Newcombe, Richard A.; Izadi, Shahram; Hilliges, Otmar; Molyneaux, David; Kim, David; Davison, Andrew J. et al. (2011): KinectFusion: Real-Time Dense Surface Mapping and Tracking. In: Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality. Washington, DC, USA: IEEE Computer Society (ISMAR '11), S. 127–136. Online verfügbar unter http://dx.doi.org/10.1109/ISMAR.2011.6092378.

Nister, D.; Naroditsky, O.; Bergen, J. (2004): Visual Odometry. In: Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on, Bd. 1, I-652-I-659 Vol.1.

Osher, S.; Fedkiw, R. (2003): Level Set Methods and Dynamic Implicit Surfaces: Springer Verlag.

Oumer, Nassir W.; Panin, Giorgio (2012): Tracking and Pose Estimation of Non-Cooperative Satellite for On-Orbit Servicing. In: i-SAIRAS.

Pelton, Joseph N. (2013): Space Debris and Other Threats from Outer Space. New York, NY: Springer (SpringerBriefs in Space Development).

Penrose, R. (1955): A Generalized Inverse for Matrices. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 51 (3), S. 406–413.

Petit, Antoine; Despre, Noela; Marchand, Eric; Kanani, Keyvan; Chaumette, Francois; Provost, Séverin; Flandin, Grégory (2011): 3D Model-based Visual tracking for Space Autonomous Rendezvous. In: 8th Int. ESA Conf. on Guidance and Navigation Control Systems. 8th Int. ESA Conf. on Guidance and Navigation Control Systems. Carlsbad, Czech Republic.

Reintsema, D.; Thaeter, J.; Rathke, A.; Naumann, W.; Rank, P.; Sommer, J. (2010): DEOS – The German Robotics Approach to Secure and De-Orbit Malfunctioned Satellites from Low Earth Orbits. In: i-SAIRAS.

Rublee, E.; Rabaud, V.; Konolige, K.; Bradski, G. (2011): ORB: An Efficient Alternative to SIFT or SURF. In: Computer Vision (ICCV), 2011 IEEE International Conference on, S. 2564–2571.

Scaramuzza, Davide; Fraundorfer, Friedrich (2011): Visual Odometry. Part I: The First 30 Years and Fundamentals. In: IEEE Robotics & Automation Magazine, Bd. 4, S. 80–92.

Scherer, Sebastian A.; Zell, Andreas (2013): Efficient Onbard RGBD-SLAM for Autonomous MAVs. In: Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, S. 1062–1068.

Schnitzer, F.; Sonnenburg, A.; Janschek, K.; Willich, G. (2012): Bildbasierte SLAM-Relativnavigation und 3D-Rekonstruktion für das On-Orbit-Servicing von unbekannten und unkooperativen Raumflugkörpern. In: Deutscher Luft- und Raumfahrtkongress.

Solà, Joan (2014): Simulataneous Localization and Mapping with the Extended Kalman Filter. ‘A very quick guide… with Matlab code!’. Universitat Polièc nica de Catalunya. Barcelona. Online verfügbar unter http://www.iri.upc.edu/people/jsola/JoanSola/objectes/curs_SLAM/SLAM2D/SLAM%20course.pdf.

Spies, Hagen; Barron, John L. (2002): Evaluating the Range Flow Motion Constraint. In: Proceedings of the 16th International Conference on Pattern Recognition, Volume 3: IEEE Computer Society, S. 517.

Steinbrücker, F.; Sturm, J.; Cremers, D. (2011): Real-Time Visual Odometry from Dense RGB-D Images. In: Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on, S. 719–722.

Steinbrücker, F.; Sturm, J.; Cremers, D. (2014): Volumetric 3D Mapping in Real-Time on a CPU. In: Robotics and Automation (ICRA), 2014 IEEE International Conference on, S. 2021–2028.

Tykkälä, T.; Audras, C.; Comport, A. I. (2011): Direct Iterative Closest Point for Real-Time Visual Odometry. In: Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on, S. 2050–2056.

Whelan, T.; Johannsson, H.; Kaess, M.; Leonard, J. J.; McDonald, J. B. (2013): Robust Real-Time Visual Odometry for Dense RGB-D Mapping. In: IEEE Intl. Conf. on Robotics and Automation, ICRA. Karlsruhe, Germany.

Yuheng Ren, Carl; Prisacariu, Victor; Murray, David; Reid, Ian (2013): STAR3D: Simultaneous Tracking and Reconstruction of 3D Objects Using RGB-D Data. In: Proceedings of the IEEE International Conference on Computer Vision, S. 1561–1568.

Zeng, Ming; Zhao, Fukai; Zheng, Jiaxiang; Liu, Xinguo (2013): Octree-based Fusion for Realtime 3D Reconstruction. In: *Graphical Models* 75 (3), S. 126–136.

Zhang, Zhengyou (2012): Microsoft Kinect Sensor and Its Effect. In: *IEEE MultiMedia* 19. Online verfügbar unter https://www.microsoft.com/en-us/research/publication/microsoft-kinect-sensor-and-its-effect/.

Züfle, Johannes (2017): Integration eines robusten Visual-SLAM-Algorithmus in eine Simulationsumgebung zur Nahbereichsnavigation von Satelliten. Technische Universität München, Lehrstuhl für Raumfahrttechnik.