TU WIEN Informatics

Mobile Collaborating Robots for Direct Haptics in Mixed Reality

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Reinhard Sprung, BSc.
Matrikelnummer 00725956

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Mag. Hannes Kaufmann
Mitwirkung: Dipl.-Ing. Mag. Emanuel Vonach, Bakk.

Wien, 12. April 2021

Reinhard Sprung	Hannes Kaufmann

TU WIEN Informatics

Mobile Collaborating Robots for Direct Haptics in Mixed Reality

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Reinhard Sprung, BSc.
Registration Number 00725956

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Mag. Hannes Kaufmann
Assistance: Dipl.-Ing. Mag. Emanuel Vonach, Bakk.

Vienna, 12th April, 2021

Reinhard Sprung	Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Reinhard Sprung, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. April 2021

Reinhard Sprung

Danksagung

Ich bedanke mich bei Univ.Prof. Dr. Mag. Hannes Kaufmann und Dipl.-Ing. Mag. Emanuel Vonach, Bakk. für die Betreuung meiner Diplomarbeit und bei Dipl.-Ing. Soroosh Mortezapoor für die technische Hilfe im Virtual-Reality-Labor der TU Wien.

Weiterer Dank gilt Manuela, Yasmin, Friederike, Elvira und Matthias für das Korrekturlesen der Arbeit, Christian und Michael für die Hilfe beim Bau von Requisiten, Ella für Ideen zum Design des Posters, Peter für Empfehlungen zur Datenauswertung der Testergebnisse und meiner Familie und meinen Freunden für die moralische Unterstützung beim Verfassen der Diplomarbeit.

Wendelin und Silja
Diplomarbeitsbabies

Jakob, Jonathan und Maja
Diplomarbeitskinder

Mila, Levi und Lotti
Diplomarbeitshunde

Kurzfassung

Nach Weiterentwicklungen im Bereich der Computergrafik und der Miniaturisierung von elektronischen Schaltkreisen findet die Technologie der *virtuellen Realität* (*VR*) erstmals Zugang zu einem breiten Publikum. Kommerzielle *VR*-Systeme, wie die *Vive* Pro von *HTC*, erlauben es ihren Trägern beziehungsweise Trägerinnen virtuelle Welten in Bild und Ton in einem Ausmaß zu erleben, der ihnen realistisch genug erscheint darin einzutauchen (Immersion). Bei der Interaktion mit ihrer virtuellen Umgebung wird jedoch schnell ein Defizit bemerkbar, und zwar, dass diese keine physischen Eigenschaften oder haptisches Feedback abseits der Eingabegeräte bietet. Zusätzliche, am Körper zu tragende Geräte, wie Ganzkörperanzüge oder Exoskelette, bieten nur eingeschränkte haptische Möglichkeiten oder minimalen Tragekomfort durch überhöhtes Gewicht. Haptische Konzepte, denen Benutzer und Benutzerinnen im Simulationsbereich begegnen können, sind oftmals an einen räumlichen Ort gebunden oder unterstützen, aufgrund der hohen Mobilität geschuldeten leichten Bauweise, nur leichte Berührungshaptik. Diese Diplomarbeit nimmt sich daher dieser Thematik an und beschreibt Konzeption und Implementation eines *VR*-Systems, das haptisches Feedback in Raumgröße ermöglicht. Es wird gezeigt, wie ein *mobiler Manipulator* des Typs *RB-Kairos* in Kombination mit dem Virtual-Reality-*Headset Vive* Pro verwendet werden kann, um Benutzern und Benutzerinnen physische Requisiten im Rahmen einer *VR*-Anwendung zur Verfügung zu stellen und haptische Sinneseindrücke zu vermitteln. So wurde die *Lighthouse*-Trackinglösung der *Vive* Pro in den *RB-Kairos* integriert, um die Positionsbestimmung des Roboters im selben Kontext wie dem *Headset* zu ermöglichen. Als Softwaregrundgerüst wird das *Robot Operating System* (*ROS*) verwendet, das bereits für die Steuerung des Roboters und seiner Komponenten verantwortlich ist und zur Bereitstellung der Haptik um neue Module erweitert wird. Im Bereich der virtuellen Realität kommt das Spiele-*Framework Unity* zum Einsatz, das durch entsprechende *Plugins* die einfache Erstellung von *VR*-Anwendungen ermöglicht. Die Kommunikation zwischen *VR*-Anwendung, Roboter und Benutzer beziehungsweise Benutzerinnen erfolgt kabellos und ermöglicht diesen eine uneingeschränkte Mobilität innerhalb des Versuchsraums. Die anschließende technische Evaluierung beantwortet Fragen zu den Betriebsparametern des Systems und listet Verbesserungspotential und Erweiterungsmöglichkeiten auf.

Abstract

After technological advancements in computer graphics and miniaturization of electric circuits, virtual reality has finally found its way into the consumer market. Commercial VR systems like *HTC*'s *Vive* allow their wearers to experience virtual worlds realistically enough to feel audio-visually immersed. However, when interacting with the simulated environment, the limitations of such a system become apparent quickly. They offer no haptic capabilities or feedback beyond what is integrated in their hand-held input devices. Additional body-worn equipment, like haptic suits or exoskeletons, deliver only rudimentary haptic experiences or encumber the user's ease of movement with excessive weight. Haptic hardware of the 'encounter' type are often constrained to a specific location within the simulation area or deliver only soft touching sensations because of their highly mobile but fragile architecture. Therfore, this thesis covers the topic of creating a VR system with haptic feedback and describes its design and implementation in a room sized setup. The paper shows how a mobile manipulator, like the *RB-Kairos*, can be combined with a virtual reality headset, like the *Vive*, to deliver real world props into the hands of users to enhance their virtual experience. To track the manipulator's position with the same accuracy of the VR headset, the *Vive*'s *Lighthouse* tracking solution is integrated into the robot. On the software side, the system takes advantage of the *Robot Operating System* (*ROS*), which is already configured to control the robot's basic functionality and is extended to include new modules handling the deliverance of haptic sensations. The simulation of the visual part of this project is handled by the gaming engine *Unity*, which features a variety of plugins suitable to create basic VR applications with minimal effort. The communication between VR application, *RB-Kairos* and user is handled wirelessly via radio signals which allows unrestricted mobility for participants and robots within the simulation area. The subsequent technical evaluation offers insights to operating parameters and lists potential enhancement and upgrade possibilities.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Virtual Reality ist eine Technologie, die eine künstlich erzeugte Realität simuliert, in einem Maße, dass ein Benutzer diese für überzeugend genug hält, um in ihr einzutauchen. Um diese Immersion zu bewerkstelligen, werden die Stimuli, die ein Benutzer von der echten Welt erhält, blockiert und durch jene einer virtuellen Umgebung (englisch: *Virtual Environment*) ersetzt. Durch Weiterentwicklungen im Bereich der Computergrafik und der Miniaturisierung elektronischer Bauteile steht diese Technologie nun erstmals einem weltweiten Publikum zur Verfügung und findet Einzug in den Verbrauchermarkt der Unterhaltungsindustrie. Vertreter solcher Produkte sind die *Oculus Rift* [rif], *HTC* Vive [viv] und *Valve Index* [ind].

Ein solches kommerzielles *VR*-System besteht typischerweise aus mehreren Komponenten: einem am Kopf getragenen und Augen umschließenden *Headset* oder *Head Mounted Display (HMD)*, Kopfhörer, Eingabegeräten (*Controller*) und Trackinglösungen, um die Position des Kopfes und der Hände im Raum zu erfassen. Durch die Erfassung der Kopfposition wird dem Benutzer eine Visualisierung einer virtuellen Welt berechnet, die ihm am *HMD* angezeigt wird und sich seiner Kopfbewegungen ensprechend verändert. Ebenso kann dem Benutzer über Kopfhörer die Klangkulisse des virtuellen Raumes vorgespielt werden. Durch die Stimulation des Seh- und Hörsinnes wird einem Benutzer eine glaubhafte Präsenz in einer virtuellen Welt vorgegaukelt, jedoch nur bis zu dem Moment, an dem dieser versucht mit der Umgebung zu interagieren.

Der Mensch verwendet zur Interaktion mit seiner Umwelt hauptsächlich seine Hände. Die Feinmotorik der dortigen Muskeln erlauben präzise Bewegungsvorgänge und die in der Haut vorhandenen spezialisierten Nervenzellen ermöglichen ihm, eine Vielzahl an Oberflächenmerkmalen zu ertasten. Zu den Sinneseindrücken der haptischen Wahrnehmung (Tastsinn) zählen unter anderem Druck, Wärmeempfinden, Verschieben der Hautoberfläche, Fingerposition und Muskelspannung. Eine ganzheitliche Simulation aller Empfindungen scheint bereits durch die schiere Anzahl schwierig. Daher beschränken

(a) Hand *Controller* der HTC *Vive* Pro [viv].

(b) Waffe im Spiel *DOOM VFR* [iS].

Abbildung 1.1: Vergleich zwischen Hand *Controller* und Spielwaffe.

sich die haptischen Eindrücke in den vorhin genannten kommerziellen Produkten meist nur auf das Festhalten der Eingabegeräte.

Die Form der Eingabegeräte für kommerzielle *VR*-Systeme erinnert an Griffe von Werkzeugen, wie Bohrmaschinen oder Schusswaffen und damit bewusst an Gegenstände, die in typischen Videospielen vorkommen. Der *Controller* der *Vive* Pro besitzt ein Touchpad, mit dem Benutzer und Benutzerinnen mit dem Daumen einfache, analoge Kommandos abgeben kann sowie einen pistolenähnlichen Fingerabzug, den sogenannten *Trigger*, die der Simulation eben genannter Gegenstände zugutekommt (siehe Abbildung 1.1). Über die verbauten Buttons und *Trigger* kann mit dem *VR*-System interagiert werden, was zum Beispiel zum Abfeuern der virtuellen Waffe oder dem Ergreifen eines Objektes führt. Besonders bei letzterem fällt diese Form der Haptik negativ auf, da die Visualisierung das Halten eines neuen Gegenstands suggeriert, der Benutzer oder die Benutzerin aber weiterhin den selben *Controller* in der Hand hält. Weiters besitzen virtuelle Gegenstände keine reale Masse und werden daher von echten physikalischen Prinzipien nicht beeinflusst. Ohne ensprechender Simulationshardware kann man durch sie durch greifen oder sie ohne Kraftaufwand oder Trägheit aufheben. Die genannten, kommerziellen *VR*-Systeme besitzen keinerlei solcher Spezial-Hardware und verzichten daher weitgehend auf die Simuation von Haptik abseits des Festhaltens der Eingabegeräte. Diese ist jedoch Thema aktiver Forschung und auch diese Diplomarbeit beschäftigt sich damit.

1.1 Motivation

Um eine virtuelle Welt um eine haptische Komponente, ohne der Verwendung von Eingabegeräten zu erweitern, kann die *virtuelle Realität* in der echten Welt nachgebaut werden, beziehungsweise eine *virtuelle Umgebung* der echten nachempfunden werden. Hierbei werden dem Benutzer beziehungsweise der Benutzerin die audiovisuellen Eindrücke per *Headset* vermittelt, der Tastsinn hingegen über das Berühren von realen Objekten.

(a) Eine Küche in der *virtuellen Realität*.

(b) Durch Attrappen nachgebaute Küche die die virtuelle widerspiegelt.

Abbildung 1.2: Ausschnitt aus Insko et al. *Passive Haptics* [Ins01].

Insko et al. kombinieren in *Passive Haptics*[Ins01] einen virtuellen Rundgang durch eine Küche mit echten Objekten aus Styropor und Sperrholz, die in etwa der Form der Küchenmöbel der virtuellen Welt ähneln und diese überzeugend simulieren (siehe Abbildung 1.2). Der Versuch beschränkt sich auf statische Möbel und das Ertasten der Oberfläche. Eine Interaktion mit Möbeln, etwa das Öffnen von Kästen oder das Hochheben von Geschirr ist nicht vorgesehen und wird vom System auch nicht erkannt. Ebenso ist die *VR*-Installation an den Versuchsraum gebunden, da die haptischen Replika für jede virtuelle Welt eigens gebaut werden müssen.

Eine einfache Form der interaktiven Haptik stellen *Force-Feedback*-Systeme dar. Zu ihnen zählen die in vielen Videospiel-*Controllern* bereits seit Jahren verbauten Vibrationsmotoren, die etwa Erschütterungen im Spielgeschehen abbilden oder zur Unterstützung des Spielers bei kniffligen Passagen dienen. Zum Beispiel kann einem Spieler durch Variation der Vibrationsintensität bei der Bewältigung seiner Spielaufgaben geholfen werden, etwa indem der *Controller* bei Annäherung an ein Ziel stärker vibriert als bei größerer Entfernung. Abbildung 1.3 zeigt das Innenleben eines Videospiel-*Controllers* mit zwei Vibrationsmotoren und eine Spielszene, wo dieses Konzept Anwendung findet.

Diese in den Controllern verbaute Hardware kann leicht auf weitere Körperbereiche ausgedehnt werden, um Feedback abseits der Hände zu bieten. Es existieren bereits kommerzielle Produkte, wie zum Beispiel der Tesla Suit [Ltda] (siehe Abbildung 2.1a), ein hautanliegender Ganzkörperanzug, der dem Träger elektrische Impulse an verschiedenen Körperregionen übermitteln kann. Dieser wird in Kapitel 2 näher beschrieben.

Der Nachteil von *Force-Feedback* Ausrüstung besteht darin, dass diese von den Benutzern (herum) getragen werden müssen und deren Vibrations- oder Impulshaptik im Vergleich zur Berührung echter Objekte wenig realistisch wirkt. Die Verwendung von realen Gegenständen zur Erzeugung von interaktiver Haptik stellt ebenso Herausforderungen dar,

(a) Ansicht eines geöffneten *XBox 360* Wireless Gamepads. In den Handgriffen im unteren Bildbereich sind zwei Vibrationsmotoren mit unterschiedlichen Gewichten angebracht, die für Vibrationshaptik sorgen.

(b) Im Yoga-Minispiel von *GTA V* [gta] wird dem Spieler mittels Vibration des Gamepads geholfen die richtige Position der Analog Sticks einzunehmen.

Abbildung 1.3: Haptisches Feedback in Videospielen.

hauptsächlich die Frage, wie ein solches Objekt in die Hände der Benutzer gelangen soll, beziehungsweise wie es nach erfolgter Benutzung wieder aus dem Interaktionsraum entfernt wird, ohne die Benutzer merklich zu stören.

Diese Diplomarbeit nimmt sich daher der Thematik an, eine haptische *VR*-Erfahrung zu erschaffen in der ein mobiler Roboter die Arbeit des Bereitstellens von Gegenständen übernimmt. In einem raumgroßen Setup soll es Benutzern möglich sein, sich mit minimalem am Körper zu tragendem Equipment frei zu bewegen und an entsprechenden Stellen haptische Sinneswahrnemung zu erleben.

Dem Benutzer soll eine virtuelle Szene gezeigt werden, mit der dieser interagieren kann. Auf Knopfdruck kann er oder sie ein Hindernis, im konkreten Fall eine Wand, markieren, die ihm oder ihr als haptisches Objekt zur Verfügung gestellt wird. Dazu wird die Position der Wand im virtuellen Raum ermittelt und in Koordinaten der realen Welt transformiert, wo ein Roboter die Platzierung des Hindernisses übernimmt. Der Roboter soll eine möglichst große Palette an zulässigen Positionen im gesamten Raum abdecken und dabei präzise agieren. Das Empfinden des Hindernisses erfolgt mit bloßen Händen, also ohne zusätzlich zu tragende Haptikhardware. Da das virtuelle Objekt real repliziert wird, ist auch eine Interaktion mit dem ganzen Körper möglich, nicht nur an vorgesehenen Körperteilen. Die Präsenz des Roboters mitsamt seiner Eigenmasse erlaubt zudem eine gewisse Kraftausübung. Anstatt eine Wand nur anzudeuten, wird dem Benutzer tatsächlich der Weg versperrt. Dieser könnte sich im Gegenzug aber auch gegen das Hindernis lehnen oder Gegenstände darauf platzieren. Weiters soll es dem System möglich sein, durch Integration zusätzlicher Hardware, sowohl mehrere Benutzer als auch mehrere Roboter im gleichen Setup zu unterstützen.

Abbildung 1.4: Pressefoto der HTC *Vive* Pro [viv] ©`https://www.vive.com/`.

1.2 Ziele

Ziel dieser Arbeit ist die Erstellung einer raumgroßen *VR*-Anwendung, deren Realismusgrad durch den Einsatz von haptischen Komponenten, verbessert wird. Um dieses Setup umzusetzen, bedarf es spezieller Hard- und Software.

Kern der Anwendung ist die Verwendung eines kommerziellen *VR*-Systems. Bei dessen Auswahl überzeugt die HTC *Vive* Pro [viv] (siehe Abbildung 1.4) in mehreren Punkten. Ihr *Lighthouse*-Trackingsystem erlaubt laut Hersteller, bei Verwendung von vier Basisstationen das Erfassen der *VR*-Hardware in einem Bereich von $10 \times 10\,\mathrm{m}$. Um den gesamten Bereich auch physisch erreichen zu können, kann das normalerweise kabelgebundene *Headset* mit einem Wireless-Kit aufgerüstet werden. Die Video- und Audiodaten werden damit zum *HMD* gefunkt, während es durch einen am Gürtel getragenen Akku mit Strom versorgt wird. Weiters existieren zu diesem *VR*-System zusätzliche *Tracking-Module* die an systemfremde Komponenten angebracht werden können, damit diese von der *Lighthouse*-Technologie ebenfalls erkannt werden können.

Ein Roboter soll für die Bereitstellung der Haptik verantwortlich sein. Dieser hat folgende Kriterien: Er muss mobil sein, schnell und präzise agieren, dabei den gesamten Trackingbereich erreichen können und eine lange Betriebsdauer aufweisen. Die Wahl fiel auf den *mobilen Manipulator RB-Kairos* der Firma Robotnik [roba], der mit einem Roboterarm des Modells *UR-10* des Herstellers *Universal Robots* [ur] ausgestattet ist. Der Roboter verfügt über *Mecanum-Räder*, die es ihm ermöglichen, sich in alle Richtungen fortzubewegen und rasch drohenden Kollisionen mit Menschen auszuweichen, ohne sich erst in die entsprechende Richtung drehen zu müssen. Dabei erreicht er eine Maximalgeschwindigkeit von $3\,\mathrm{m/s}$.

Der verbaute *Manipulator* hat einen Arbeitsradius von $1{,}3\,\mathrm{m}$ und eine Nutzlast von

10 kg. Aufgrund der Höhe der Roboterbasis kann der *Endeffektor* des Arms Höhen von etwa 1,95 m erreichen. Der verbaute Akku erlaubt eine durchgängige Manipulationszeit von 10 Stunden. Außerdem ist der *UR-10* als kollaborierender Roboterarm ausgelegt und dadurch auf eine Interaktion mit Menschen, beziehungsweise einen Betrieb in deren unmittelbarer Nähe konzipiert. Er besitzt Sensoren, die äußere Krafteinwirkungen oder Kollisionen erkennen können und bei Bedarf die Bewegungsausführung stoppen. Diese Sensorik kann auch dazu verwendet werden, um seine Gelenke per Hand zu bewegen. Die Kommunikation mit dem *RB-Kairos* findet über Wi-Fi statt. Abbildung 1.5 zeigt ein Foto des *RB-Kairos*.

Auf dem Roboter sollen zusätzliche Komponenten angebracht werden. Am *Endeffektor* des Roboterarms wird eine Requisite montiert, die als haptisches Interaktionsobjekt dienen soll. Weiters wird *Tracking-Hardware* der HTC *Vive* Pro am Roboter angebracht, um seine Position innerhalb des selben Koordinatensystems zu bestimmen, in dem sich auch die Benutzer beim Tragen des *Headsets* befinden.

Als Softwarelösung zur Erstellung einer *VR*-Szene fiel die Wahl auf das 3D-*Framework Unity* [unia]. Es erlaubt die Erstellung einer funktionsfähigen *VR*-Anwendung durch sein vielfältiges Angebot an *Plugins* und wird bereits in einigen *VR*-Setups der *TU Wien* verwendet.

Diese Soft- und Hardwarekomponenten sollen in einem ganzheitlichen Setup zusammenarbeiten. Die Berechnung der *VR*-Szene übernimmt ein Windows-PC, der die Aufgabe hat, diese auf das drahtlose *HMD* zu übertragen. Die *VR*-Anwendung kommuniziert über Wi-Fi mit dem *mobilen Manipulator* und dem dort installierten *Robot Operating System (ROS)*. Diese Softwarelösung kümmert sich bereits um die Basisfunktionalität des Roboters und soll um weitere Module erweitert werden. Ein Modul behandelt dabei das Erfassen seiner Position im Bezug zum *Virtual Environment (VE)* des Benutzers. Ein weiteres Modul empfängt Positionskommandos von der *Unity*-Anwendung die durch Navigations-Algorithmen in Bewegungskommandos des Roboter umgewandelt und ausgeführt werden. Die Endpositionierung des Interaktionsobjekts geschieht durch den Roboterarm.

Das Augenmerk der verteilten Anwendung liegt auf Kontrolle, Präzision und Zuverlässigkeit. Dazu ist es hilfreich entsprechende Verantwortungsgebiete auf die richtigen Plattformen aufzuteilen. Das Erzeugen der audiovisuellen Daten des *VR*-Erlebnisses übernimmt zur Gänze der Windows-Rechner, während die Verantwortung zur Positionsbestimmung und Wegfindung des Roboters ihm selbst obliegt. Somit kann bei einem möglichen Kommunikationsausfall jede Plattform für sich selbst weiter agieren und im Falle einer Veränderung des Setups leichter durch eine neue ersetzt oder ergänzt werden.

Um die Kontrolle zu bewerkstelligen, werden vorgefertigte Softwarebibliotheken von *ROS* verwendet, unter anderem auch fertige Tools, um die Bewegung des Roboters zu steuern oder die empfangenen Sensordaten anschaulich zu visualisieren.

Abbildung 1.5: Ein fertig aufgebauter *RB-Kairos* von Robotnik [roba].

1.3 Gliederung

Diese Diplomarbeit besteht aus acht Kapiteln. Die Einleitung in Kapitel 1 (Einleitung) gibt einen Überblick über die Ziele und Gliederung dieser Arbeit, die Anforderungen an die verwendete Hardware und die Auswahl der Komponenten.

Kapitel 2 (Related Work) beschreibt den aktuellen Stand von Forschung und Technik im Bereich der Simulation von Haptik. Dabei werden unterschiedliche Verfahren von einfacher bis komplexer Haptik in Raumgröße vorgestellt sowie auf deren Vor- und Nachteile eingegangen.

Diese Diplomarbeit vereint Konzepte aus den Bereichen von *Virtual Reality* und Robotik mit jeweils eigenen Hard- und Softwarelösungen. Daher werden in Kapitel 3 (Grundlagen) die Grundlagen der verwendeten Systeme erklärt sowie deren Zusammenspiel beschrieben. Der Fokus liegt hierbei auf dem *Robot Operating System* (*ROS*), das sämtliche Funktionen des Roboters steuert und auch die Grundlage dieser Arbeit bildet.

Kapitel 4 (Design) beschreibt den Ist-Zustand der bereits am Roboter vorhandenen Funktionalitäten und ihre Interaktionen untereinander. Weiters beschreibt es Designüberlegungen, wie diese Features mit *VR*-Hardware kombiniert werden können, um eine roboterunterstützte *VR*-Simulation in Raumgröße zu erschaffen.

Kapitel 5 (Umsetzung) zählt zum Herzstück dieser Arbeit und listet die konkreten Schritte, die zur Umsetzung des *VR*-Systems führen. Zu ihnen zählen die Anbringung von zusätzlicher Hardware am Roboter, die Konfiguration von Netzwerk und Roboterarm sowie die Implementierung der Steuerungssoftware und der *VR*-Anwendung. Zusätzlich werden die Schritte zur Inbetriebnahme des Systems gelistet.

Kapitel 6 (Evaluierung) enthält die technische Evaluierung, die Fragen zur Leistungsfähigkeit des Systems beantwortet und dessen Betriebsparameter auflistet. Weiters wird ein Pilottest beschrieben, der die zu erwartende Benutzererfahrung des Systems zeigt.

Kapitel 7 (Diskussion) beschäftigt sich mit den in den vorhergehenden Kapiteln gewonnenen Erkenntnissen und bietet konkrete Vorschläge für eine Verbesserung oder Vereinfachung des Systems und deren Komponenten. Dieses Kapitel listet außerdem Möglichkeiten auf, wie das System um zusätzliche haptische Konzepte erweitert werden kann, unter anderem auch ein weiterer Testaufbau mit dem Roboter dieser Arbeit. Der Abschluss bietet einen Überblick über mögliche Anwendungsgebiete eines solchen Systems.

Kapitel 8 (Zusammenfassung) enthält schließlich eine Zusammenfassung und gibt einen weiteren Ausblick auf das Potential des im Rahmen dieser Diplomarbeit erstellten *VR*-Systems.

KAPITEL 2

Related Work

Es existieren bereits viele Ansätze, um haptisches Feedback für *VR*-Anwendungen zu realisieren. Dieses Kapitel gibt einen Überblick über einige dieser Verfahren und teilt sie in zusammengehörende Kategorien ein. Abschnitt 2.1 (Passive Haptik) beschreibt Anwendungen, die *VR*-Szenen Haptik in Form von unveränderlichen, greifbaren Objekten oder Gegenständen verleihen. Abschnitt 2.2 (Aktive Haptik & Force Feedback) listet Systeme, die von Benutzern am Körper getragen werden und ihnen durch computergesteuertes Einwirken haptische Sinneswahrnehmungen vermitteln. Abschnitt 2.3 (Begegnungshaptik in Raumgröße) behandelt haptische Konzepte, die nicht an den Körper der Benutzer gebunden sind, sondern denen man in raumgroßen *VR*-Setups „begegnet“.

2.1 Passive Haptik

Zu der Gruppe der virtuellen *passiven Haptik* zählen Konzepte, die virtuelle Inhalte durch die Haptik echter Objekte ergänzen, um ihnen auf diesem Weg eine Repräsentation in der realen Welt zu verleihen. Zu ihnen gehört das, in der Einleitung erwähnte, *Passive Haptics* von Insko et al. [Ins01] (siehe Abbildung 1.2), bei dem eine virtuelle Küche in echt nachgebaut wurde und so die Haptik liefert. Wie erwähnt besitzt das Setup keinerlei Interaktionsmöglichkeiten und beschränkt sich auf das Berühren der Möbelattrappen. Diese sind außerdem fest verbaut und können daher nur Haptik für diese eine *VR*-Simulation liefern. Der Vorteil dieses Systems liegt in der Simplizität, die sich sowohl in den Anschaffungskosten als auch in den Anforderungen des technischen Know-hows niederschlägt. Das *VR*-System dieser Diplomarbeit versucht jedoch durch Einsatz eines Roboters die fehlende Interaktionsmöglichkeit, beziehungsweise den dynamischen Aufbau einer Szene zu ermöglichen.

Hettiarachchi et al. gehen hingegen mit *Annexing Reality* [HW16] den umgekehrten Weg. Sie beschreiben eine *Augmented Reality* Anwendung bei der Alltagsgegenstände mit einem computergenerierten 3D-Modell überlagert werden und so den Benutzern den Eindruck

vermitteln, als hielten sie stattdessen die virtuellen Objekte in ihren Händen. Anders als in *Passive Haptics*, wo der Zweck darin besteht, einer virtuellen Küche eine reelle Haptik zu verleihen, wird in *Annexing Reality* ein bestehender haptischer Gegenstand um zusätzliche, virtuelle Repräsentationen erweitert.

Das gleiche Konzept verfolgen die bereits erwähnten *Controller* der handelsüblichen *VR*-Systeme, wie etwa der *Vive* Pro. Ihre sehr einfache Form, die an Werkzeuggriffe erinnert, erlaubt ihnen, die Haptik für eine Vielzahl an virtuellen Gegenständen zu liefern. Abbildung 1.1 zeigt ein typisches Beispiel von *Controller* und virtueller Waffe im *VR*-Spiel *DOOM VFR* [iS]. Die Haptik ist jedoch bei allen Gegenständen die gleiche, egal ob es sich um eine virtuelle Schusswaffe oder einen virtuellen Holzstock handelt. Sowohl die Masse als auch die Oberflächenbeschaffenheit ist in beiden Fällen die gleiche. Am meisten fällt diese Ungereimtheit aber bei Gegenständen auf, die aufgrund ihrer Form nicht gut durch *Controller* wiedergegeben werden können, aber mangels Alternativen ebenso durch sie repräsentiert werden. Zum Beispiel ein Ball, der im Spiel virtuell weggeworfen werden kann. Der *Controller* wird aber weiterhin in den Händen gehalten, während lediglich die „Wurftaste“ betätigt wird.

Ein weiterer Nachteil von *Annexing Reality* und den *Controllern* besteht darin, dass die Gegenstände ausschließlich durch Benutzerinteraktion bewegt werden können. Ein Umwerfen einer virtuellen Flasche durch Umwerfen ihrer realen Repräsentation ist möglich. Ein Umkippen der Flasche in der virtuellen Welt hat jedoch aufgrund fehlender Manipulationshardware keinen Einfluss auf die echte. Das System ist daher stehts auf die Kooperation ihrer Benutzer und Benutzerinnen angewiesen. Durch Einsatz eines Roboterarms könnte die virtuelle Welt auch in der Realität abgebildet werden.

Die einzige Möglichkeit, mit der kommerzielle *VR*-Systeme die Haptik ihrer *Controller* beeinflussen können, ist das Erzeugen von Rütteleffekten durch eingebaute Vibrationsmotoren. Dadurch zählen die *Controller* ebenso zu den Beispielen der *aktiven Haptik*.

2.2 Aktive Haptik & Force Feedback

Zur *aktiven Haptik* gehören alle Konzepte, die den Benutzern haptische Sinneseindrücke vermitteln, die über das bloße Berühren und Anfassen hinausgehen. Wie im vorigen Abschnitt erwähnt, zählen dazu etwa Vibrationsmotoren in Eingabegeräten, die den Benutzern und Benutzerinnen an geeigneter Stelle zusätzliche Informationen oder Immersion auf haptischem Weg vermitteln können. Man spricht hierbei auch von *Force-Feedback*. Gängige Beispiele in Videospielen sind etwa das Rütteln des *Controllers* bei einer Autofahrt über unebenes Gelände oder die Simulation des Herzschlages des spielbaren Charakters in Gefahrensituationen. Abbildung 1.3 zeigt die Vibrationsmotoren eines Gamepads sowie ihre Anwendung in einem Videospiel.

Das *Force-Feedback* der *VR-Controller* erstreckt sich nur auf die Hände beziehungsweise Handflächen der Benutzer und Benutzerinnen, weswegen es bereits umfassende Forschung gibt, wie dieses auf den restlichen Körper ausgeweitet werden kann. *Synesthesia Suit* von

Konishi et al. [KHM+16] und *Towards Full-Body Haptic Feedback* von Lindeman et al. [LPYS04] präsentieren Ansätze zur Umsetzung von Ganzkörperhaptik durch vibro-taktile Stimulation einzelner Körperstellen. Zu den kommerziell erhältlichen Produkten zählt der *TESLASUIT* [Ltda] (siehe Abbildung 2.1a), ein an der Haut anliegender Ganzkörperanzug, der den Trägern an 80 Stellen ihrer Körper elektrische Impulse vermitteln kann. Laut Hersteller findet der *TESLASUIT* breite Anwendung in Industrie, Training und im Gesundheitsbereich. Da diese Geräte am Körper, ähnlich einer Kleidung, getragen werden, nennt man sie auch *Wearables*.

Der Vorteil im Vergleich zum Robotiksystem dieser Arbeit liegt in den geringeren Anschaffungskosten und der Möglichkeit einer punktgenauen Auslieferung von haptischen Impulsen an vielen Stellen des Körpers gleichzeitig. Diese Impulse decken jedoch nur einen Bruchteil der möglichen haptischen Empfindungen ab und beinhalten keinerlei Krafteinwirkung auf den Träger. Die Ganzkörperanzüge werden direkt am Körper getragen, weshalb ihr Platzbedarf sehr gering ist. Damit wäre eine gleichzeitige Verwendung mit dem *VR*-System dieser Diplomarbeit denkbar und könnte damit sowohl eine Umgebungshaptik durch den Roboter als auch eine Per-User-Haptik durch die Anzüge abdecken.

Eine weitere Möglichkeit, eine am Körper zu tragende Haptik zu realisieren, stellen Al-Sada et al. mit *Haptic Snakes* [ASJR+19] vor. Es handelt sich dabei um Roboterarme, die am Bauch getragen werden und mittels ihrer *Endeffektoren* mit ihren Trägern und Trägerinnen interagieren. Ihr Repertoire inkludiert das Antippen der Benutzer am Oberkörper und das Zeichnen von Gesten, also das oberflächige Verschieben von Hautbereichen. In der erweiterten Version, der *Haptic Hydra* (siehe Abbildung 7.2), besitzt der Roboterarm am *Endeffektor* eine Drehscheibe, die zusätzliche Werkzeugspitzen bereit hält. Diese sind ein Greifarm, eine Bürste und ein Gebläse. Das Beispiel zeigt den Vorsprung an Interaktionsmöglichkeiten, die ein einzelner Manipulator gegenüber einem vibro-taktilen System haben kann. Die Montage am Bauch, und das sich dadurch ergebende Mitführen des Roboters, wirkt sich jedoch im Vergleich zu einem haptischen Anzug oder dem System dieser Diplomarbeit negativ auf den Tragekomfort aus. Das Konzept ist außerdem nicht in der Lage greifbare Haptiken zu erzeugen, da sich die Manipulation des Roboterarms nur auf die Hautoberfläche beschränkt. Der wechselbare *Endeffektor* der *Haptic Hydra* dient aber als Vorbild dafür, wie die statische Requisite des Roboters dieser Arbeit dynamisch ausgetauscht werden könnte.

Die bisher vorgestellten aktiven Haptiken erlauben nur die Simulation von Sinneseindrücken auf der Hautoberfläche. Bei der Simulation von Kollisionen mit einem virtuellen Objekt könnte zum Beispiel ein *VR-Controller* vibrieren, um den Benutzer über die Existenz dieses Hindernisses aufzuklären. Der *VR-Controller* ist jedoch nicht in der Lage die Bewegung zu stoppen. Ein Spieler oder eine Spielerin kann problemlos seine oder ihre Hand, beziehungsweise den *Controller* durch dieses Objekt hindurchbewegen. Im in dieser Diplomarbeit vorgestellten *VR*-System ist dies jedoch möglich. Es wird ein Roboter verwendet, um eine Requisite an der entsprechende Stelle des virtuellen Objekts zu platzieren und so für ein physisches Hindernis zu sorgen. Eine andere Möglichkeit, die

Bewegung in ein virtuelles Objekt zu stoppen, besteht in der Verwendung von spezialisierter *Force-Feedback*-Hardware, zum Beispiel *Force-Feedback*-Handschuhen, die auch *Haptic Gloves* genannt werden.

Diese Handschuhe sind mit Sensoren ausgestattet, die die Stellungen der Fingergelenke überwachen. Sollte sich in der Nähe der Finger ein virtuelles Hindernis befinden, so können die *Haptic Gloves* angewiesen werden, durch eingebaute *Aktuatoren* Kräfte auf die Finger auszuüben und deren Bewegung in die entsprechende Richtung zu unterbinden. Perret el al. [PVP18] vergleichen mehrere solcher Systeme. Der Hersteller des *TESLASUITs* hat ebenfalls einen Handschuh, den *TESLASUIT GLOVE* [Ltdb] angekündigt, der diesen Anwendungsfall abdeckt (siehe Abbildung 2.1b). Damit könnte das Ergreifen eines faustgroßen Gegenstands simuliert werden sowie deren grobe Oberflächenbeschaffenheit. Darüber hinaus besitzen die Handschuhe *Robotic Shape Displays* in jedem Finger, um rudimentäre Oberflächentexturen darzustellen. Der Realismusgrad liegt jedoch weit unter dem, den echte Objekte und auch die Requisite dieser Diplomarbeit liefern. Zudem ist die Beinflussung der Gelenke auf die Hände beschränkt. Ein Benutzer oder eine Benutzerin könnte mit diesem Equipment weiterhin durch eine virtuelle Wand hindurch greifen.

Um auch die Gelenksstellungen jenseits der Finger zu beeinflussen, existieren *Force-Feedback*-Systeme, die auch auf weitere Muskelpartien wirken. Diese mechanischen *Exoskelette* werden unter anderem dafür verwendet, die muskuläre Kraft des Trägers oder der Trägerin zu unterstützen. Sie können aber auch benutzt werden, um der Kraft entgegenzuwirken und so eine Haptik zu erzeugen. Ein eigens für die Simulation von Haptik entworfenes *Exoskelett* für die Handgelenke ist das *SPIDAR-W* [NTAS15] von Nagai et al. Dabei handelt es sich um ein Gerüst, das auf den Schultern und dem Bauch getragen wird. Die Handgelenke befinden sich in speziellen *Endeffektoren*, die mit jeweils vier Seilwinden mit dem Gerüst verbunden sind. Die zwei Winden in Bauchhöhe sind in einem Winkel von 90° zu den Seilwinden in Schulterhöhe angebracht, womit eine kontrollierte Bewegung in sechs Freiheitsgraden möglich wird. Mit der gleichen Technik können auch Bewegungen in bestimmte Richtungen unterbunden und so die Präsenz eines physischen Hindernisses simuliert werden. Die Nachteile eines solchen Systems liegen in der Größe und Masse des zu tragenden Gerüsts, wodurch der Tragekomfort eingeschränkt und die Handhabung in engen Passagen, wie zum Beispiel beim Durchschreiten von Türen, erschwert wird.

Ein solches System könnte das Hindurchgreifen durch eine virtuelle Wand unterbinden. Solange jedoch nicht jede menschliche Muskelpartie in einem *Force-Feedback-Exoskelett* bedacht wird, wird es immer Wege geben, wie die Beschränkungen der virtuellen Physik umgangen werden können. Ein System, das zumindest auch die Beinbewegungen einschränkt, scheint bereits anhand von Größe, Masse und der nötigen Motorik schwierig umzusetzen. Das Konzept dieser Diplomarbeit geht daher einen anderen Weg und verwendet zur haptischen Simulation von virtuellen Objekten reale Gegenstücke. Der Vorteil liegt unter anderem in der realistischen Oberflächensimulation durch Berührung mit den bloßen Fingern sowie der physischen Präsenz und damit einer natürlichen Bewegungseinschränkung beim Versuch, diese Objekte zu penetrieren.

(a) *TESLASUIT* [Ltda]: Der hautanliegende Ganzkörperanzug kann seinem Träger an 80 Stellen des Körpers elektrische Impulse vermitteln.

(b) *TESLASUIT GLOVE* [Ltdb]: Ein zum *TESLASUIT* kompatibler, vom Hersteller angekündigter *Force-Feedback* Handschuh, der Fingerbewegungen messen und entgegenwirken kann.

Abbildung 2.1: Kommerzielle Produkte der *TESLASUIT*-Reihe.

2.3 Begegnungshaptik in Raumgröße

Die vorgestellten Konzepte der *aktiven Haptik* bieten eine Haptiksimulation durch Tragen des benötigten Simulationsequipments. Die dadurch ermöglichte Mitnahme erlaubt die Ausführung der Simulation an willkürlich ausgewählten Orten, ist andererseits jedoch auch an die Nachteile gebunden, die das Herumführen von zusätzlicher Ausrüstung mit sich bringen. In diesem Abschnitt werden daher Konzepte präsentiert, wie das am Körper des Benutzers oder der Benutzerin getragene Equipment minimiert werden kann und die Person der haptischen Simulation zum entsprechenden Zeitpunkt „begegnet“. Man spricht hier auch von *Encounter Type Displays* (engl.: encounter: begegnen). Hierbei kann es sich um fixe Installationen handeln, zu denen sich ein Benutzer oder eine Benutzerin hinbewegt, oder um den umgekehrten Ansatz, bei dem haptische Erlebnisse zum User gebracht werden.

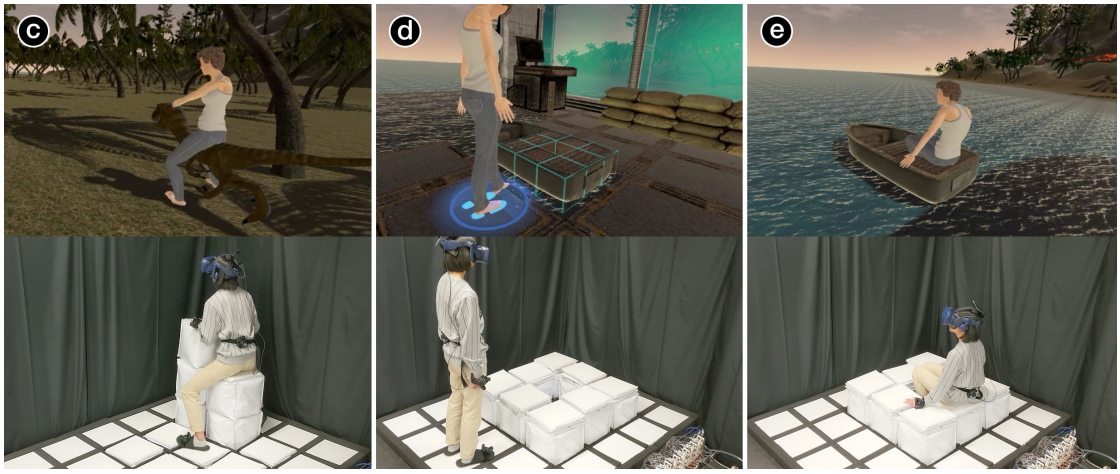


Abbildung 2.2: *TilePop* [TLC+19]: In der Bodeninstallation befinden sich leere Luftkammern, die in drei verschieden große Blöcke aufgeblasen werden können, um so Treppen oder Sitzgelegenheiten zu simulieren.

Einen Vertreter der ersten Kategorie präsentieren Teng et al. mit *TilePoP* [TLC+19]. Es handelt sich dabei um ein *Robotic Shape Display* in Menschengröße. In einer Bodeninstallation sind Luftkammern in einem quadratischen Raster angeordnet, die mit Luftdruck zu Würfeln aufgeblasen werden können. Jede Bodenfliese besitzt drei dieser Würfel und kann somit vier verschiedene Höhenstufen einnehmen, je nachdem ob keiner der Würfel oder alle drei mit Luft gefüllt werden. Die Autoren beschreiben ein spezielles Faltverfahren, das während des Auspumpens der Luft für ein reibungsloses Verschwinden der Luftkörper in der Bodenplatte sorgt sowie mögliche Anwendungsgebiete. Dazu gehören die Erzeugung von Treppen und Sitzgelegenheiten sowie auch die Simulation von Oberflächeneigenschaften, wie die Weichheit eines Luftpolsters oder das Atmen eines Tieres. Abbildung 2.2 zeigt die Fliesen in ihren möglichen Zuständen. Die Vorteile dieser Technik sind, dass auf kleinstem Raum eine Vielzahl an haptischen Erlebnissen untergebracht werden kann und diese auch den Krafteinwirkungen der Benutzer und Benutzerinnen, die sich auf die Würfeln setzen können, standhalten. Der Nachteil gegenüber dem in dieser Diplomarbeit vorgestellten Konzept ist jedoch die Ortsgebundenheit der Installation auf einen bestimmten Bereich innerhalb des Versuchsraums und die Beschränkung auf orthogonale Strukturen. Die Simulation von schrägen Oberflächen ist durch die würfeligen Luftkörper nicht möglich.

Eine andere Art der Begegnungshaptik beschreiben Vonach et al. mit *VRRobot* [VGK17]. Anstatt Simulationsequipment am Körper zu tragen, oder sich zu einer fixen Installation im Raum hin zu bewegen, verharrt der Benutzer oder die Benutzerin in einer *omnidirektionalen Bewegungsplattform*. Von dort aus bekommt er oder sie haptische Sinneseindrücke durch einen in der Nähe befindlichen Roboterarm vermittelt. Der User wird mit seiner Hüfte an der Plattform fixiert und bleibt so an seiner Position. Die rutschige Bodenplatte ermöglicht aber seine Füße in alle Richtungen gleiten zu lassen

(a) Die visuelle Welt wie sie der *VR*-Benutzer oder die *VR*-Benutzerin vorfindet.

(b) Von menschlichen Akteuren aus Universalrequisiten nachgebaute Umgebung.

Abbildung 2.3: Ausschnitt aus *TurkDeck* [CRR+15] von Cheng et al. Der Versuch verwendet menschliche Helfer, um mittels tragbarer Requisiten eine virtuelle Welt dynamisch in der Realität nachzubilden.

um so Gehbewegungen am Stand durchzuführen. Durch die in der Platte verbauten Sensoren können die virtuelle Bewegungsrichtung und -geschwindigkeit des Benutzers oder der Benutzerin erfasst werden. Der Roboterarm verbleibt in Ruheposition und wird erst bei einer bevorstehenden Interaktion aktiv. Das System beherrscht die Simulation einer Wand, die den Benutzern in Form einer Holzplatte entgegengestreckt wird sowie eine Kollision mit einem virtuellen Computercharakter, die über einen am *Endeffektor* angebrachten Boxhandschuh realisiert wird. Das System ist ähnlich zu jenem, das im Zuge dieser Diplomarbeit realisiert werden soll. Der größte Unterschied besteht jedoch darin, dass sich sowohl Benutzer als auch Roboter frei im Raum bewegen können.

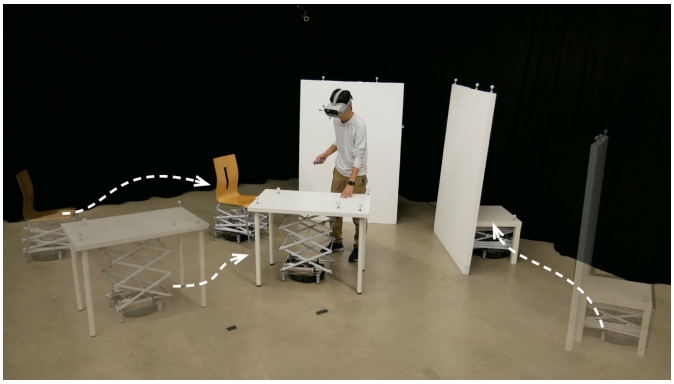
Zu den Konzepten der freien Bewegungsausübung zählt das *TurkDeck* [CRR+15] von Cheng et al (siehe Abbildung 2.3). Dabei handelt es sich um eine raumgroße *VR*-Installation, die zur Manipulation der Umgebung eine Gruppe menschlicher Helfer verwendet. Diese sind mit tragbaren Klappwänden ausgestattet, die sie je nach Situation an bestimmten Stellen des Versuchsraums aufstellen oder niederlegen können. Ein Computerprogramm erteilt durch Sprachausgabe Befehle an die menschlichen „Roboter“, die auf Kommando den Simulationsbereich umgestalten, während sich eine einzelne Person in der *VR*-Simulation befindet. Eine Fußbodenprojektion hilft bei der Ausrichtung der Objekte. Die Vorteile einer solchen menschenbasierten Anwendung liegt in der schnellen Umsetzung von Test-Prototypen und der hohen Flexibilität der Akteure. Eine Umsetzung auf Dauer ist jedoch an die Verfügbarkeit einer großen Personengruppe und deren Kosten gebunden. Hier setzt diese Diplomarbeit an und widmet sich der Problematik, die Aufgaben der menschlichen Akteure durch echte Roboter zu erfüllen.

Eine *VR*-Erfahrung ohne menschliche Akteure präsentieren Suzuki et al. mit *RoomShift* [SHZ+20] (siehe Abbildung 2.4a). Sie verwenden einen Schwarm von Miniaturrobotern, um Möbel zu verschieben und so den Simulationsraum umzugestalten. Bei den Robotern handelt es sich um Entwicklungsplattformen von Staubsaugerrobotern, die mit einem

Hebemechanismus und vernetzter Steuerung ausgestattet sind. Sie können Objekte mit einer maximalen Masse von 22 kg befördern und sie bis zu einer Höhe von einem Meter hochheben. Zu den Möbeln zählen Tische, Stühle und Wände. Diese werden je nach Anwendungsfall in eine an den Benutzer oder Benutzerin angepassten Stellung bewegt oder durch ihn oder sie interaktiv verschoben. Aufgrund ihrer fragilen Bauweise werden die Roboter dieser Simulation lediglich als Transportgeräte der Möbel verwendet. Möchten die User eines der gebrachten Objekte benutzten, zum Beispiel um sich auf einen Stuhl zu setzen, so muss dieser vorher vom Roboter wieder abgestellt und damit die Manipulation abgeschlossen werden; das Eigengewicht der Benutzer und Benutzerinnen würde sonst den Roboter zerstören. Durch Abstellen des Objektes ist es jedoch nicht mehr möglich die Haptiken wieder zu verändernd. Dieses Experiment, nämlich wie ein Schwarm an Robotern zur Manipulation der Umgebung eingesetzt werden kann, dient dieser Diplomarbeit als Vorlage. Der verwendete Roboter ist jedoch von einer anderen Bauart: sein Roboterarm erlaubt nur die Manipulation von Objekten mit einer maximalen Masse von 10 kg, die Objekte können allerdings in allen erdenklichen Positionen in Stellung gebracht, anstatt nur im Raum verschoben zu werden.

Die Möglichkeit, Objekte frei im Raum zu platzieren, bieten ebenfalls Abtahi et al. mit *Beyond The Force* [ALY+19]. Die Autoren verwenden eine Drone, auf der ein Stück Stoff angebracht ist, das Kleidung simulieren soll. Ein Benutzer oder eine Benutzerin kann sich einem virtuellen Kleidungsstück nähern und bekommt vom System einen berührbaren Bereich angezeigt. Der Quadcopter navigiert zur gewünschten Position und präsentiert dem Benutzer oder der Benutzerin an der erwarteten Stelle die Simulationsoberfläche. Etwaige Fehlgriffe mit den Händen können sowohl von der Drone als auch von der *VR*-Anwendung korrigiert werden, indem der zu tastende Bereich visuell entsprechend verschoben und so die Armbewegung unterbewusst angepasst wird. In einem anderen Versuch kann ein virtuelles Kleidungsstück am Kleiderhaken genommen und an eine Kleiderstange gehängt werden. Dazu wurde an der Drone ein echter Kleiderhaken angebracht, der während des Fluges der Drone in der Luft „hängt“. Die Eigenmasse der Drone sorgt für ein realistisches Gewichtsgefühl während des Transports durch den Benutzer oder die Benutzerin und erlaubt durch ihre Flugfertigkeit ein „Aufhängen“ des Kleiderhakens an einer willkürlichen Stelle im Raum (siehe Abbildung 2.4b). Im Vergleich zum Roboter dieser Diplomarbeit kann die Drone um einiges schneller manövrieren, aufgrund ihrer geringen Masse bietet sie jedoch nur eine eingeschränkte Möglichkeit der Userinterkation standzuhalten. Die Simulation eines fixen Objekts, wie einer Wand, ist damit nicht möglich.

Eine weiteres Konzept, das Haptik zum Benutzer oder der Benutzerin bringen kann, bietet *shapeShift* [SGY+17] von Siu et al (siehe Abbildung 7.1). Es handelt sich um ein *Robotic Shape Display*, das auf einer mobilen Roboterplattform angebracht wurde. Das *Robotic Shape Display* besitzt 288, in einem quadratischen Raster angeordnete Metallstifte, die von Motoren um bis zu 0,05 m nach oben bewegt werden können. So können verschiedene Oberflächenstrukturen oder Vertiefungen erzeugt werden. Im aktiven Modus wird die Hand der Benutzer getrackt und das *Robotic Shape Display* durch Bewegung der Räder zu ihr hinbewegt. Mit diesem fahrbaren Ansatz lassen sich



(a) Die Roboter in *RoomShift* [SHZ+20] von Suzuki et al. besitzen einen Hebemechanismus mit dem Möbel im Raum verschoben werden können.

(b) *Beyond The Force* [ALY+19] von Abtahi et al. erlaubt das „Aufhängen“ eines Kleidungsstücks in der Luft, durch Verwendung einer Drone.

Abbildung 2.4: Beispiele von Haptiksimulationen durch mobile Roboter in raumgroßen Setups.

Oberflächenstrukturen darstellen, die weit größer sind als das Display selbst. Seine geringe Größe von etwa 0,25 m Seitenlänge legt eine Verwendung auf einer Tischplatte nahe. Der Roboter besitzt eine gewisse Ähnlichkeit zum Roboter dieser Diplomarbeit. Er bewegt sich ebenso über omnidirektionale Räder fort, um haptische Erfahrungen an beliebigen Stellen im Raum zu liefern. Sein Aufbau erlaubt jedoch nur eine Oberflächensimulation auf einer geraden, horizontalen Ebene über dem Boden. Der Roboter dieser Arbeit erlaubt hingegen nur die Bereitstellung einer Requisite mit statischer Oberflächenstruktur, weswegen eine Kombination beider Techniken einen insgesamt höheren Realismusgrad bei der Simulation von Haptik erreichen könnte.

Die hier vorgestellten Arbeiten bieten alle ihre eigenen Vor- und Nachteile. Im Zuge dieser Diplomarbeit soll daher ein haptisches *VR*-System geschaffen werden, das entscheidende Vorteile der anderen Arbeiten kombiniert. Das System soll ein Hindernis präsentieren, das den Benutzern nach freier Bewegung im Raum begegnet. Es soll von einem nichtmenschlichen Akteur dort platziert werden und eine reale Präsenz bieten, also eine Eigenmasse besitzen und mit jedem Körperteil gefühlt werden können. Die weiteren Kapitel geben einen Überblick, wie so ein System entworfen werden kann.

KAPITEL 3

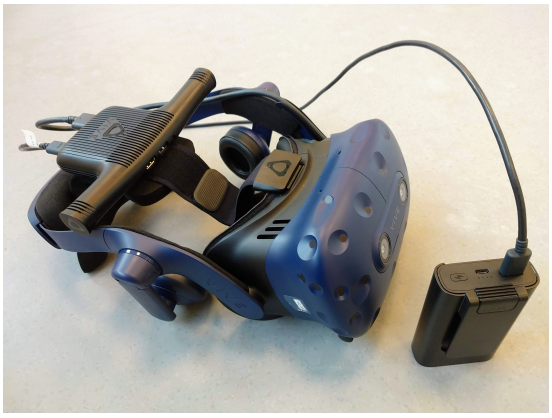
Grundlagen

Die Implementation dieser Arbeit setzt ein Grundwissen über die verwendeten Soft- und Hardwarekomponten voraus, die in diesem Kapitel näher gebracht werden sollen. Abschnitt 3.1 (Vive Pro) stellt das *VR*-System dieser Arbeit, die *Vive* Pro von HTC, vor. Abschnitt 3.2 (RB-Kairos) enthält eine Einleitung über den verwendeten *mobilen Manipulator RB-Kairos*. Abschnitt 3.3 (Robot Operating System) geht auf die Grundlagen des *Robot Operating System* ein, das für den Betrieb des *RB-Kairos* zuständig ist. Abschnitt 3.4 (UR-10) enthält Basiswissen zum *Manipulator UR-10*, der im *RB-Kairos* integriert ist. Abschnitt 3.5 (Unity) zeigt die Vorteile des *Spieleframeworks Unity* bei der Entwicklung von *VR*-Applikationen. Abschnitt 3.6 (Koordinatensysteme) führt schließlich die Koordinatensysteme auf, die die unterschiedlichen Systeme verwenden.

3.1 Vive Pro

Die Wahl eines passenden *VR*-Systems für diese Arbeit fiel auf die *Vive* Pro [viv] von HTC (siehe Abbildungen 1.1a, 1.4, und 3.1). Im Gegensatz zu anderen kommerziellen *VR*-Systemen, in denen die Benutzer und Benutzerinnen eine sitzende Position vor dem Computer einnehmen, wurde die *Vive* für eine stehende Haltung in einem frei begehbaren Raum konzipiert. Im diesem werden mindestens zwei Basisstationen, sogenannte *Lighthouses* angebracht (siehe Abbildung 3.1b), die in einem definierten Muster Infrarot-Laserstrahlen aussenden. Alle trackbaren Geräte des Systems, das sind *Headset*, *Controller* und *Tracker*, enthalten an deren Oberfläche Photosensoren, die die entsandten Laserstrahlen messen können. Anhand des Musters der ausgesendeten Strahlen und des Timings der empfangenen Signale können die Position und die Orientation der Komponenten relativ zu den *Lighthouses* ermittelt werden.

Das Tracking findet also in den Trackern selbst statt, während die Basisstationen lediglich passive Lichtsignale abgeben. Im Gegensatz dazu steht zum Beispiel das Tracking der *Oculus Rift* [rif], bei dem die Lichtsignale vom *Headset* ausgestrahlt und von einer



(a) Eine HTC *Vive* Pro mit Wireless-Kit. Der Funkempfänger wurde am Kopfband befestigt.

(b) Eine an einer Wand angebrachte *Lighthouse*-Basisstation.

Abbildung 3.1: Komponenten der *Vive* Pro [viv].

stationären Kamera aufgezeichnet werden. Der Vorteil des *Lighthouse*-Systems liegt darin, dass zusätzliche *Tracker* und *Headsets* die bestehenden *Lighthouse*-Stationen mitverwenden können. Die Berechnung der Positionsdaten muss auch nicht auf denselben Geräten erfolgen, wie Abbildung 4.1 zeigt.

Der Hersteller gibt für die *Vive* Pro eine trackbare Fläche von circa $5 \times 5\,\mathrm{m}$ an, die durch zusätzliche *Lighthouses* auf $10 \times 10\,\mathrm{m}$ erweitert werden kann. Die mögliche Bewegungsfläche ist jedoch durch die Tatsache, dass die Berechnung der audiovisuellen Daten nicht am *Headset* selbst, sondern an einem PC erfolgt und typischerweise per (langem) Kabel an das *HMD* gesendet wird, eingeschränkt. Zu diesem Zweck wird das *VR*-System mit dem Wireless-Kit ausgestattet. Die Berechnung der Daten erfolgt weiterhin am Computer, die Datenkommunikation findet nun aber per Funkübertragung statt. Der Umstieg auf Funk verhindert ein möglicherweise zu kurzes oder sich verhedderndes Kabel und damit ein Stolpern der Benutzer und Benutzerinnen oder eine Kollision mit dem Roboter. Die Stromversorgung des *Headsets* erfolgt nun mit einem am Gürtel getragenen Akku.

Im *VR*-Setup dieser Arbeit sind folgende *Vive*-Komponenten in Verwendung: drei *Lighthouses*, ein *HMD* inklusive Wireless-Kit und Funkstation, zwei *Controller* und zwei *Tracker*. Durch zusätzliche *HMDs* und Controller kann das System für mehrere User erweitert werden. Der Trackingbereich des *VR*-Labors der *TU Wien* hat ein Ausmaß von $8 \times 6,3\,\mathrm{m}$. Die drei *Lighthouses* sind an Wänden und Stativen in einer Höhe von circa 2,4 m angebracht.

3.2 RB-Kairos

Wie in Abschnitt 1.2 (Ziele) beschrieben, wird in dieser Arbeit der *mobile Manipulator RB-Kairos* verwendet. Dabei handelt es sich um eine Kombination aus der mobilen

Roboterbasis *Summit XL Steel* der Firma *Robotnik* [roba] und dem *Manipulator UR-10* des Unternehmens *Universal Robots* [ur]. Der Roboterarm wird in Abschnitt 3.4 (UR-10) näher beschrieben.

Die definierenden Eigenschaften des *RB-Kairos* für dieses Projekt sind, neben der Nutzung eines mobilen Roboterarmes, seine hohe Geschwindigkeit von 3 m/s, seine lange Akkulaufzeit von 10 Stunden durchgängiger Manipulationszeit und die verbauten *Mecanum-Räder*, die es ihm ermöglichen, seitlich zu fahren. Diese Manövrierfähigkeit ist deshalb wichtig, da sich der Roboter in einem Bereich aufhält, in dem auch Menschen Zutritt haben. Die Möglichkeit, in alle Richtungen rasch ausweichen zu können, kann Kollisionen mit und damit Verletzungen an Menschen minimieren.

Zur Erfassung der Umgebung sind zwei 270° *Lidars* der Marke *SICK S300* [sic] integriert, die es dem Roboter erlauben, eine Karte seiner Umgebung anzufertigen und sich später darin wiederzufinden. Eine *Inertial Measurement Unit (IMU)* vom Typ *Pixhawk 4* [pix] sorgt zusätzlich für eine Lagebestimmung anhand von Trägheits- und Erdmagnetsensoren. Die Erfassung seiner Umgebung durch Laserscans ermöglicht es dem Roboter, um Hindernisse herum zu navigieren und so Kollisionen zu vermeiden. Falls es dennoch zu Kollisionen kommen sollte, kann das integrierte Notstoppsystem, das sämtliche motorischen Aktivitäten des Roboters stoppt, per Funk ausgelöst werden.

Um eine simple Steuerung des Gefährts zu ermöglichen, liegt dem Roboter ein *PlayStation 4 Controller* [pla] bei. Mit diesem kann die mobile Bewegungsplattform per Hand gesteuert werden. Diese manuelle Steuerung wird, gegenüber der automatischen Steuerung der Navigationssoftware, bevorzugt behandelt und kann dabei helfen, etwaige Navigationsfehler zu korrigieren und Kollisionen zu vermeiden.

3.3 Robot Operating System

Die Funktionen des *mobilen Manipulators RB-Kairos* werden über das *Robot Operating System* (*ROS*) gesteuert. Dabei handelt es sich nicht um ein Betriebssystem im eigentlichen Sinn, sondern um eine Sammlung von Softwarelösungen (*Frameworks*), die ein breites Spektrum an Roboterfunktionen abdecken. Als echtes Betriebssystem fungiert *Ubuntu 16.04*, auf dem *ROS* in der Version *Kinetic Kame* läuft.

Aktuelle Robotersysteme bestehen oftmals nicht mehr nur aus einem einzigen Roboter, sondern aus einer Kombination von mehreren. Diese setzen sich wiederum aus mehreren Komponenten und Computern zusammen. *ROS* bietet mit seinem auf *Topics*, *Publishern* und *Subscribern* basierenden Netzwerkprotokoll eine einfache Kommunikationmöglichkeit über einen solchen Verbund von Robotiksystem. Ein *Publisher* ist ein Softwarekonstrukt, das unter einem bestimmten *Topic* Nachrichten in das *Robot Operating System* einspeist. Analog dazu ist ein *Subscriber* ein Objekt, das Daten aus dem System empfängt und weiterverarbeitet. Dabei können beliebig viele *Publisher* und *Subscriber* auf dasselbe *Topic* zugreifen. Ein einzelnes Programm im Kontext von *ROS* wird *Node* genannt und enthält typischerweise mehrere *Publisher* und *Subscriber*.

Name	RB-Kairos
Bauweise	Mobiler Manipulator
Maße (Arm eingezogen)	$0,892 \times 0,56 \times 0,825\,\mathrm{m}$
Maße (Arm ausgestreckt)	$0,812 \times 0,56 \times 1,875\,\mathrm{m}$
Masse	100 kg Rover + 29 kg Arm
Maximalgeschwindigkeit	3 m/s
Autonomie	10 h
Roboterarm	UR-10
Nutzlast Arm	10 kg
Reichweite Arm	1,3 m
Fortbewegung	4 Mecanum-Räder
Lidar	2× SICK S300
IMU	Pixhawk 4
Manuelle Steuerung	PlayStation 4 Controller
Mainboard	Jetway NF9QU-Q87
Prozessor	Intel® Core™ i7-4790S @ 3,20 GHz (8 CPUs)
Arbeitsspeicher	8 GByte
Integrierte Grafik	Intel® Haswell Desktop
Betriebssystem	Ubuntu 16.04 LTS 64-bit
Robot Operating System	Kinetic Kame

Tabelle 3.1: Technische Spezifikationen und integrierte Hardware des *RB-Kairos*.

Bei einem *Topic* handelt es sich um einen Datenpfad, ähnlich einer *Uniform Resource Locator (URL)*, unter dem die Nachrichten abgelegt und abgerufen werden können. Die Nachrichten, im *ROS*-Kontext *Messages* genannt, werden durch einen bestimmten Typ angegeben, der beschreibt, welche Art von Daten in der Nachricht übertragen werden. *Messages*, die in dieser Arbeit verwendet werden, sind in Abschnitt A.3 (Code-Auflistung) aufgelistet.

Die Kommunikation der *Nodes* untereinander wird vom *roscore* bereitgestellt. Es handelt sich dabei um den Kernknoten von *ROS*, der die Verwaltung aller *ROS*-Komponenten übernimmt und in einem Robotersystem genau einmal vorkommen muss. Da *ROS* als netzwerkbasiertes System ausgelegt ist, ist es aus Entwicklersicht auch irrelevant, auf welchem Computer welche *Node* ausgeführt wird, da dies ebenso vom *roscore* geregelt wird.

Als Beispiel für die Funktionsweise von *ROS* kann folgendes System herangezogen werden: Ein mobiler Roboter soll über ein Notebook, an das ein Joystick angeschlossen ist, ferngesteuert werden. Roboter und Notebook sind über Wi-Fi verbunden und haben Zugriff auf denselben *roscore*. Am Notebook läuft eine *ROS-Node*, die die Stellung der Analogsticks und Buttons des Joysticks ausliest und die Daten unter dem *Topic* `joy` als `Joy` *Message* (Auflistung A.9) veröffentlicht. Neben dieser *Node* wird am Notebook

auch noch eine *Node* zur Fernsteuerung des Roboters ausgeführt. Diese *Teleop-Node* implementiert einen *Subscriber*, der dieses *Topic* abonniert. Sie wandelt die empfangenen Achsenstellungen in konkrete Richtungsangaben um und publiziert diese unter dem neuen *Topic* `cmd_vel` als `Twist` Nachricht (Auflistung A.6). Am Roboter wird eine *Node* ausgeführt, die das *Topic* `cmd_vel` abonniert und die veröffentlichten *Messages* empfangen kann, ohne eine explizite Netzwerkübertragung zu implementieren. Der Roboter wandelt die erhaltenen Richtungsangaben schließlich in Steuersignale für die vier Motoren seiner Räder um und fährt in die vorgegebene Richtung.

Ein weiterer Vorteil dieser modularen Struktur ist die Möglichkeit, einzelne Komponenten auszutauschen oder wiederzuverwenden. So könnte zum Beispiel ein zweites Gamepad angeschlossen werden, dessen Daten unter dem *Topic* `joy2` in *ROS* veröffentlicht werden. Die *Teleop-Node* könnte dann so konfiguriert werden, dass sie ihren Input nun vom *Topic* `joy2` bezieht, während das erste Gamepad die Kontrolle über einen anderen Roboter übernimmt. Ebenso wäre es denkbar, eine weitere *Teleop-Node* zu erstellen, die Roboter statt durch User Input anhand von Sensordaten und künstlicher Intelligenz steuert.

Einzelne *Nodes* und Services werden typischerweise in Module mit thematisch ähnlichen Funktionen gegliedert, die sogenannten *Packages*, und bilden damit die oberste Hierarchie, anhand derer *Nodes* im System verwaltet werden. Ein Start einer bestimmten *Node* erfolgt durch Angabe des *Package*-Namens und des *Node*-Namens und kann mit dem Terminal-Befehl `rosrun` erfolgen. `rosrun joy joy_node` erzeugt beispielsweise eine von *ROS* zur Verfügung gestellte *Node* im *Package* `joy`, um die Daten eines generischen Linux Joysticks (oder Gamepads) in *ROS* zu publizieren.

Da die Ausführung von einzelnen *Nodes* in großen Robotiksystemen per Hand sehr aufwändig werden kann, können diese in sogenannten *Launch*-Dateien gebündelt werden. Es ist dadurch auch möglich, benutzerdefinierte Startparameter an die *Nodes* zu übergeben sowie weitere *Launch*-Dateien zu starten. Der Terminal-Befehl `roslaunch teleop_twist_joy teleop.launch` startet zum Beispiel die `joy_node` und die *Teleop-Node* des ersten Beispiels. Der Inhalt der Datei `teleop.launch` wird in Abbildung 3.2 angezeigt. Der Start von *ROS* am *RB-Kairos* erfolgt über eine zentrale *Launch*-Datei, die wiederum andere *Launch*-Dateien der einzelnen Roboterfunktionen ausführt.

Eine weitere wichtige Funktionalität, die durch *ROS* zur Verfügung gestellt wird, ist der sogenannte *Transform Tree*. In ihm können die Positionsdaten aller im System befindlichen Komponenten erfasst werden. Dazu zählen die fixen Positionen der vier *Mecanum-Räder* und der Montagepunkt des *UR-10* innerhalb des Roboters, aber auch bewegliche Koordinaten wie die Gelenksneigungen des Roboterarms. Wichtig ist dies auch bei der Erfassung von Sensordaten der *Lidars*, um diese in den korrekten Bezug zum restlichen Roboter zu setzen.

Wie der Name *Transform Tree* besagt, werden die Koordinaten in einer Baumstruktur verwaltet. Die Knoten des Baums, die auch *Frames* genannt werden, besitzen genau einen Elternknoten und beliebig viele Kinderknoten. Einzige Ausnahme ist der *Root-*

```
<launch>
  <arg name="joy_config" default="ps4" />
  <arg name="joy_dev" default="/dev/input/js0" />
  <arg name="config_filepath" default="$(find teleop_twist_joy)/config/
       $(arg joy_config).config.yaml" />
  <arg name="joy_topic" default="joy" />

  <node pkg="joy" type="joy_node" name="joy_node">
    <param name="dev" value="$(arg joy_dev)" />
    <param name="deadzone" value="0.3" />
    <param name="autorepeat_rate" value="20" />
    <remap from="joy" to="$(arg joy_topic)" />
  </node>

  <node pkg="teleop_twist_joy" name="teleop_twist_joy" type="teleop_node">
    <rosparam command="load" file="$(arg config_filepath)" />
    <remap from="joy" to="$(arg joy_topic)" />
  </node>
</launch>
```

Abbildung 3.2: Die von *ROS* zur Verfügung gestellte *Launch*-Datei `teleop.launch` [tel] zum Start einer *Teleop-Node*. Die Konfiguration kann durch Argumente und Parameter angepasst werden.

Frame des Weltkoordinatensystems, von dem aus alle anderen *Frames* ausgehen und der daher keinen Elternknoten hat. Jede Transformation eines Elternknotens wirkt sich gleichzeitig auch auf alle Kinderknoten aus. Diese hierarchische Verwaltung hat praktische Vorteile, da zusammengehörende Objekte innerhalb eines lokalen *Frames* definiert werden können, ohne deren absolute Position im Raum wissen zu müssen. Eine Umrechnung von Koordinaten zwischen verschiedenen *Frames* wird von *ROS* zur Verfügung gestellt. Somit lassen sich zum Beispiel die Koordinaten des *Endeffektors* des Roboterarms in absoluten Raumkoordinaten ermitteln.

Abbildung 3.3 zeigt einen Ausschnitt des *Transform Trees* des Roboterarms *UR-10*, bei dem die hierarchische Struktur zu erkennen ist. Zu sehen sind die obersten vier Glieder oder *Frames* des Roboterarms. Das sind `forearm_link`, `wrist_1_link`, `wrist_2_link` und `wrist_3_link`, zwischen denen sich jeweils die Gelenke des Roboterarms befinden beziehungsweise die Transformationen zwischen den *Frames* stattfinden. Die Stellung der Gelenke wird vom sogenannten `robot_state_publisher` mit einer Frequenz von circa 42,1 Hz veröffentlicht. Bei `ee_link` und `tool` handelt es sich um weitere *Frames*, die vom Treiber des *UR-10* angelegt werden, um dort eigene *Endeffektoren* zu definieren. Bei `rbkairos_panel` handelt es sich um die im Zuge dieser Arbeit angebrachte Requisite. Die Transformationen der letzten drei Links sind statisch definiert und werden durch eine Publikationsfrequenz von 10.000 Hz angezeigt.

Der Ursprungs-*Frame* des Roboters ist der sogenannte *Base-Footprint*. Er wird in der horizontalen Mitte des Roboters auf Höhe des Fußboden definiert. Hierarchisch gesehen

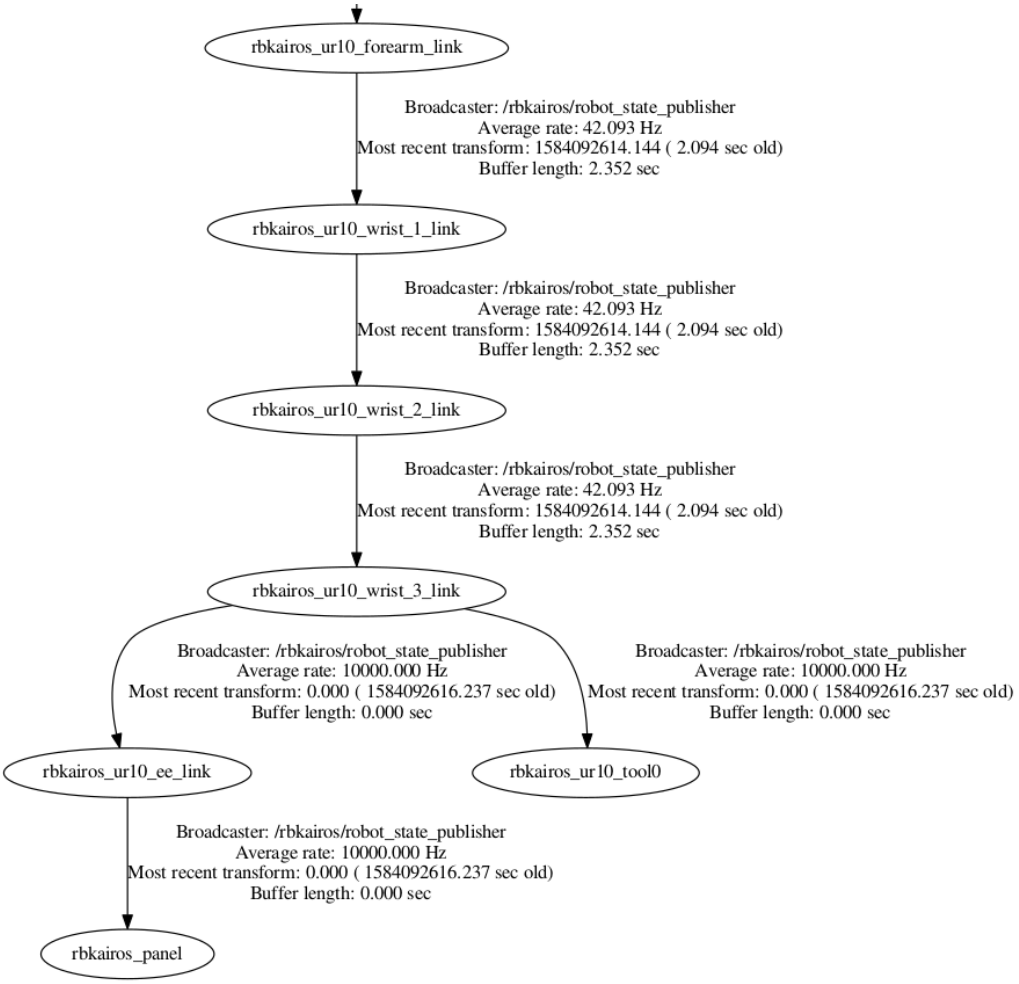


Abbildung 3.3: Ausschnitt aus dem Visualisierungstool *view_frames* [vie]. Zu sehen ist ein Teil des *Transform Trees* des *UR-10*.

befindet er sich unterhalb des *Odometrie-Frames*, der sich wiederum unterhalb des Welt-*Frames* befindet (siehe Abbildung 3.5). Bei der *Odometrie* handelt es sich um die Schätzung der Positionsveränderung des Roboters anhand seiner ausgeführten Bewegung, also den Weg, den der Roboter seit seiner Inbetriebnahme zurückgelegt hat.

Die *Odometrie* wird anhand mehrerer Faktoren ermittelt. In den Motoren der *Mecanum-Räder* sind Rotationssensoren integriert, die die Umdrehung der Räder messen. Die Räder selbst bestehen aus einer Aneinanderreihung von zwölf kleineren Rollen, die zusammen ein Laufrad bilden, siehe Abbildung 1.5 in Kapitel 1 (Einleitung). Die Rollen können sich frei drehen und ihre Rotation wird nicht überwacht. So kann es bei schlechter Bodenbeschaffenheit und langer Navigation passieren, dass der Roboter nicht die exakte Position einnimmt, die von den Sensoren berechnet wird. Daraus resultiert eine gewisse

Ungenauigkeit bei der Positionsbestimmung die zum Beispiel durch die Einbeziehung von *IMU*- und *Lidar*-Daten ausgeglichen werden kann.

Die Ermittlung seiner Position und die Steuerung der Bewegung des *RB-Kairos* übernimmt der *ROS Navigation Stack*. Es handelt sich dabei um Softwarebibliotheken zur Erfüllung von allgemeinen Bewegungsaufgaben in Robotiksystemen. Diese können grob in drei Bereiche geteilt werden: *Lokalisation*, *Pfadplanung* und *Bewegungsausführung*.

Wenn Roboter autonom von einem Ort zum anderen navigieren, ist es essenziell ihren aktuellen Aufenthaltsort zu kennen. Zu diesem Zweck gibt es unterschiedliche Methoden, mit denen elektronische Geräte ihre Position lokalisieren können. Zu ihnen gehören die eben erwähnte *Odometrie*, die sich unter anderem einer Sensorik in den Rädern bedient. Zusätzliche Geräte, wie zum Beispiel *GPS*-Empfänger, ermöglichen eine Positionsbestimmung auf globaler Ebene. Für den Zweck dieser Arbeit ist dieses System aber zu ungenau, da eine Präzision auf Millimeterbasis gefordert ist und *GPS*-Tracking in Häusern aufgrund ihrer Wände nur unzureichenden Funkempfang bietet.

Der *RB-Kairos* verwendet stattdessen zusätzlich eine Positionsbestimmung per *Lidar*. Bei *Lidar* handelt es sich um ein dem *Radar* ähnliches Verfahren der Abstandsmessung, das, statt mit elektromagnetischen Wellen, mit Laserimpulsen arbeitet. Laserstrahlen werden von einem Emitter ausgesendet, von einem Hindernis reflektiert und von einem Empfänger wieder empfangen. Anhand der Umlaufzeit beziehungsweise der Phasenverschiebung des Lichtstrahls, lässt sich die Distanz zum Hindernis ermitteln. Im *RB-Kairos* sind zwei 270° *Laserscanner* des Modells *SICK S300* [sic] verbaut, die es dem Roboter erlauben, die gesammten 360° seiner Umgebung abzutasten (siehe Abbildung 3.5). Mit der Kombination der Daten von *Odometrie*, *Lidar* und *IMU* ist es dem *RB-Kairos* per *Adaptive Monte Carlo Localization* (*AMCL*) möglich, sich auf einer zuvor aufgezeichneten Karte des Raumes wiederzufinden. Dieses Verfahren vergleicht die gebotenen Sensordaten mit nach *Monte-Carlo*-Schema zufällig ausgewählten Daten der Karte, um so eine Positionswahrscheinlichkeit zu berechnen.

Mit den ermittelten Standortdaten kann in weiterer Folge ein Navigationspfad berechnet werden. Der Pfad ist abhängig von der Ausrichtung und der Manövrierfähigkeit des Roboters. Im Fall des *RB-Kairos* könnte dieser den gesamten Weg rückwärts oder seitlich zurücklegen, da er sich in alle Richtungen gleich effizient bewegen kann und auch Sensordaten aus allen Richtungen erhält. Im Laufe der Bewegung werden mit dem *Lidar* kontinuierlich Umgebungsdaten erfasst und ermöglicht der Lokalisation auch eine Neuberechnung des *Odometrie-Frames*, falls aktuellere Messungen genauere Werte liefern.

Die Ausführung der Bewegung des *RB-Kairos* erfolgt über Geschwindigkeitskommandos an seine Räder. Bei diesen handelt es sich um *Mecanum-Räder*, deren Anordnung in Abbildung 3.4 gezeigt wird. Grundsätzlich ähnelt das Fahrverhalten des Roboters jenem anderer Fahrzeugen, die mit Differentialantrieb ausgestattet sind, beispielsweise Kettenfahrzeugen. Drehen sich die Räder auf einer Seite des Gefährts schneller als auf der anderen, so lenkt der Roboter in die Richtung der letzteren. In Abbildung 3.4 bewirkt demnach eine höhere Geschwindigkeit von V_{2W} und V_{4W} im Vergleich zu V_{1W} und V_{3W}

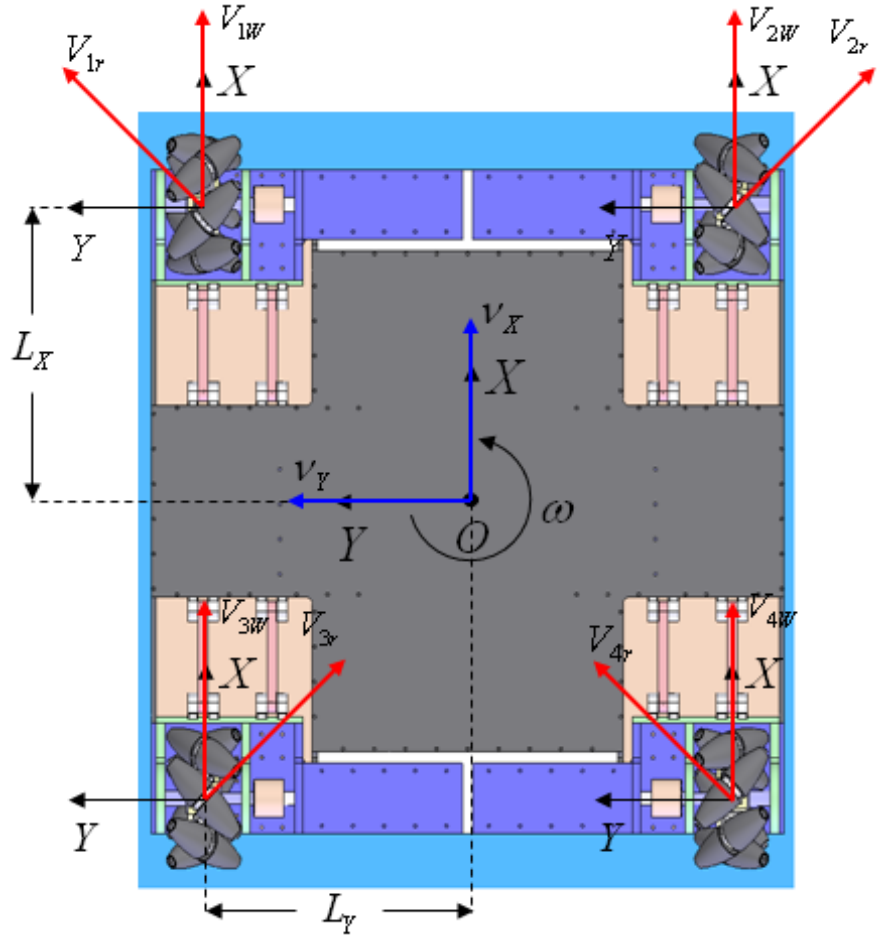


Abbildung 3.4: Auszug aus *Han et al.* [HCL+08]: Die Anordnung der *Mecanum-Räder* in einem omnidirektionalen mobilen Roboter.

eine Rotation des Roboters um die Vertikalachse im Winkel von ω.

Da es sich bei den Rädern des *RB-Kairos* um *Mecanum-Räder* handelt, verleihen diese dem Roboter zusätzliche Bewegungsmöglichkeiten. Die auf den Rädern angebrachten elliptischen Walzen existieren in zwei unterschiedlichen Konfigurationen und unterscheiden sich in der Winkelausrichtung von + oder - 45° gegenüber der Laufrichtung. In Abbildung 3.4 werden diese mit V_{1r}, V_{2r}, V_{3r} und V_{4r} angegeben, wobei sich jeweils die diagonal angeordneten Paare gleichen.

Drehen sich beide Räder einer Seite mit der selben Geschwindigkeit in die selbe Richtung, so verharren die Rollen weitgehend bewegungslos und der Roboter navigiert mit dem vorhin erwähnten Differentialantrieb. Drehen sich Vorder- und Hinterräder jedoch in unterschiedliche Richtungen, zum Beispiel V_{2W} und V_{4W} nach vorne und V_{1W} und V_{3W} nach hinten, so kommen die Rollen ins Spiel. Das Gefährt kann sich nicht in zwei Richtungen

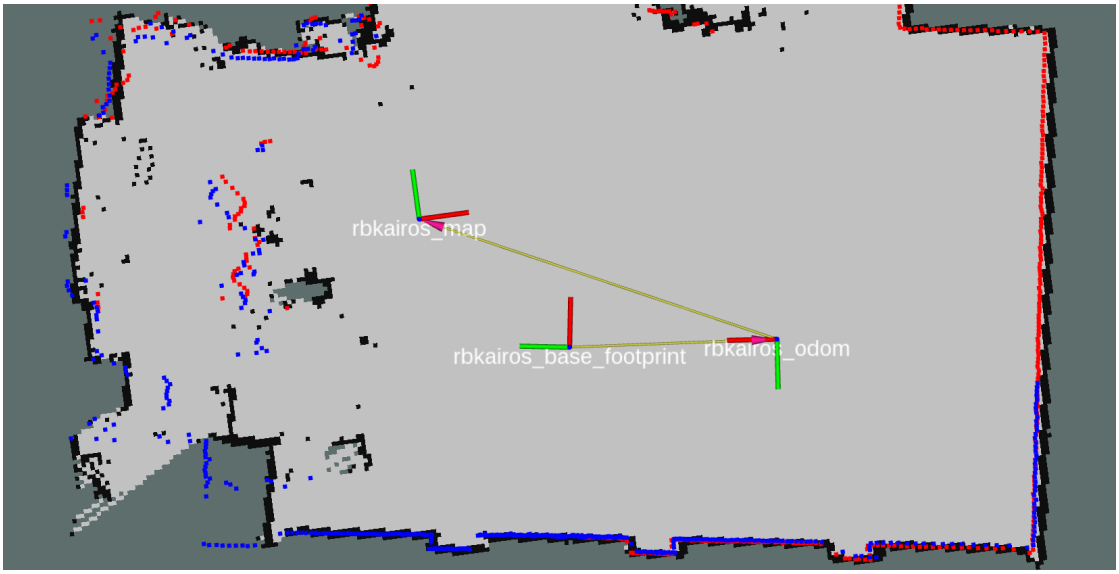


Abbildung 3.5: Ausschnitt aus dem Visualisierungstool *RVIZ*. Zu sehen sind die Daten der *Lidars* anhand roter und blauer Punkte, die im Vorfeld aufgezeichnete Karte des Versuchsraums und drei *Frames*, die zur Positionsbestimmung des Roboters verwendet werden.

gleichzeitig bewegen, daher wird diese unmögliche Bewegung auf die Walzen der Räder abgerollt. Deren 45°-Anordnung sorgt so für eine Ausgleichsbewegung im Winkel von 90° zur Seite. Im Fall des Beispiels würde sich der Roboter nach links, entlang der positiven Y-Achse, bewegen. Durch die Kombination von unterschiedlichen Drehgeschwindigkeiten und -richtungen aller Räder lassen sich eine Vielzahl an Bewegungen ausführen. Han et al. [HCL+08] liefern eine detaillierte Beschreibung von *Mecanum-Rädern* sowie ein Modell zur Berechnung ihrer Drehrichtung.

Zur Darstellung der vom Robotersystem erfassten Sensordaten stellt *ROS* das Visualisierungstool *RVIZ* zur Verfügung. Mit ihm lassen sich eine Vielzahl an Daten anschaulich visualisieren und überwachen, darunter 3D-Modelle der Roboter, Kamerabilder, Punktwolken der Laserscanner sowie Positions- und Bewegungsdaten. Abbildung 3.5 zeigt eine typische 3D-Ansicht des Programms mit verfügbaren Sensordaten. Zusätzlich zur Überwachungsfunktion können auch rudimentäre Kommandos abgesetzt werden. Es existieren *Plugins*, mit denen sich die Navigation des Roboters starten und Bewegungsabläufe des Roboterarms planen und ausführen lassen.

3.4 UR-10

Der *UR-10* des *RB-Kairos* bietet sechs Freiheitsgrade, einen Arbeitsradius von 1,3 m und eine maximale Nutzlast von 10 kg. Weiters ist er mit gängigen Roboter-Softwaresystemen kompatibel und kollaborationsfähig. Das bedeutet, dass er für einen Einsatz in der Nähe

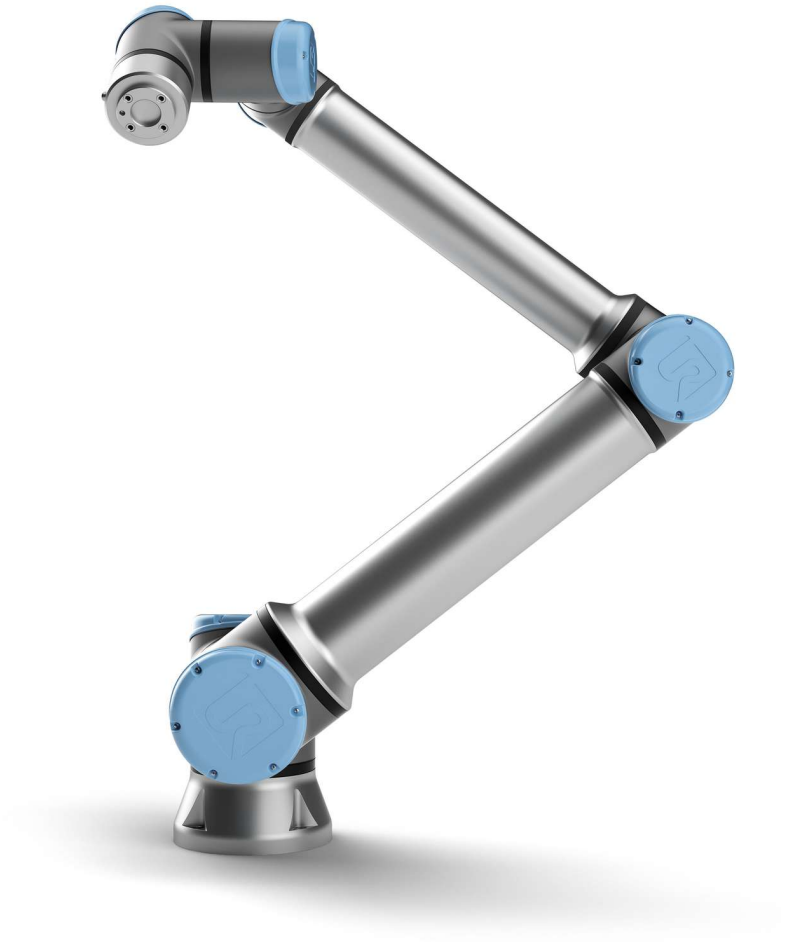


Abbildung 3.6: Pressefoto eines UR-10e. © *Universal Robots* [ur].

von Menschen oder für Arbeiten in Kollaboration mit ihnen konzipiert ist. Er besitzt Sensoren, die eine äußere Krafteinwirkung oder Hindernisse erkennen und bei Bedarf den Betrieb der Motoren einstellen können. Ein Zusatzfeature dieser Kollaborationsfähigkeit ist die Möglichkeit, die Gelenke des Roboterarms mit den Händen zu bewegen. So kann dem *UR-10* etwa beigebracht werden, manuell eingestellte Gelenkspositionen einzunehmen. Die Daten des *mobilen Manipulators* sind in den technischen Spezifikationen des *RB-Kairos* in Tabelle 3.1 gelistet. Abbildung 3.6 zeigt ein Pressefoto des *UR-10*.

Die Steuerung des Roboterarms *UR-10* erfolgt grundsätzlich über das vom Hersteller *Universal Robots* entwickelte *URScript* [ur]. Damit können dem Roboterarm statische Positionen des *Endeffektors* oder Bewegungsabläufe gesendet werden, die dieser im weiteren Verlauf ausführt. Für das Zusammenspiel mit *ROS* existiert aber auch ein

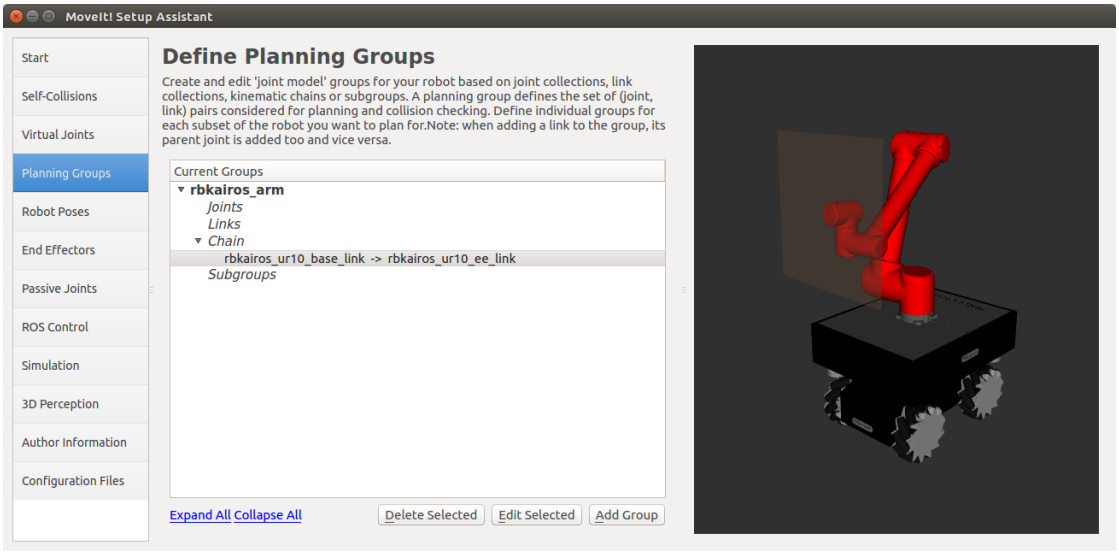


Abbildung 3.7: Screenshot aus dem *Setup Assistant* von *MoveIt*.

Treiber, um die bestehende Infrastruktur von *ROS* wiederverwenden zu können.

Der Treiber kommuniziert mit der *ROS*-Komponente *MoveIt*. Bei diesem Modul handelt es sich um ein sogenanntes *Motion Planning Framework*, das sich um die Ermittlung und Ausführung von Bewegungsabläufen in Robotersystemen kümmert. Dazu wird der Software eine dreidimensionale Beschreibung des Roboterarms inklusive der Gelenksdefinitionen übergeben. Aus diesen Daten erstellt *MoveIt* dann ein *ROS*-Modul mit dem *Kinematik*-Modell des *Manipulators*. Dabei handelt es sich um ein mathematisches Modell, das die Bewegung des *Endeffektors* anhand der Stellung der zugehörigen Gelenke beschreibt. Abbildung 3.7 zeigt einen Screenshot aus dem *Setup Assistant* von *MoveIt*.

3.5 Unity

Zur Erstellung der *Virtual-Reality*-Anwendung dieser Diplomarbeit fiel die Wahl auf *Unity*. Dieses Softwareentwicklungs-*Framework* erlaubt das einfache Erstellen unterschiedlicher 3D-Anwendungen oder Videospielarten, umfasst einen Marktplatz für Erweiterungen (*Unity Asset Store* [unib]), vergibt gratis Lizenzen für den privaten Gebrauch und richtet sich damit an unabhängige Spieleentwickler und Beginner. *Unity* wird außerdem bereits in einigen Projekten der *TU Wien*, unter anderem *Immersive Deck* [PVS+16], verwendet. Dieser Umstand erleichtert eine mögliche Integration in bestehende Anwendungen der Universität. Im erwähnten *Unity Asset Store* finden sich zudem zwei Erweiterungen, die für diese Arbeit von großem Nutzen sind: das *SteamVr Plugin* [steb] zur Erstellung von *VR*-Anwendungen und das *ROS-Sharp Plugin* [rosb] zur Kommunikation zwischen *Unity* und *ROS*.

SteamVr ist die Laufzeitumgebung für die Tracking Software *OpenVr* der *Valve Cor-*

poration für ihre online Spielevertriebsplattform *Steam*. Das Softwareunternehmen hat es sich zur Aufgabe gemacht, mit *OpenVr* alle gängigen *VR*-Systeme in ihrem Online Marktplatz zu unterstützen. Spieleentwickler können daher *VR*-Anwendungen auf Basis von *OpenVr/SteamVr* erstellen, ohne genaue Kenntnis über die zugrunde liegende Hardware haben zu müssen. Zum selben Zweck gibt es auch *SteamVr Plugins* und *Software Development Kits (SDKs)* für alle gängigen Spieleframeworks, um auch an dieser Stelle Plattformunabhängigkeit zu bieten.

Das *ROS-Sharp Plugin* [rosb] dient zur Kommunikation zwischen *Unity* und *ROS*. Da *Unity* plattformunabhängig konzipiert ist, *ROS* aber fast ausschließlich unter Linux betrieben werden kann, existieren Softwarelösungen, wie die *rosbridge*, die eine Kommunikation zwischen *ROS* und andernfalls inkompatiblen Systemen ermöglicht. Es handelt sich dabei um eine auf *JavaScript Object Notation (JSON)* basierende Webschnittstelle, die das *ROS*-Protokoll imitiert. Die *rosbridge* deckt zwar nicht alle Funktionen einer vollen *ROS*-Kommunikation ab, ist aber für einen einfachen Datenverkehr ausreichend. Eine weitere Aufgabe, die das *Plugin* erfüllt, ist die Umrechnung der Koordinaten zwischen beiden Systemen.

Zur Umsetzung der virtuellen Szene steht ein performanter Windows-Computer zur Verfügung. Seine technischen Daten finden sich in Tabelle 3.2

Prozessor	Intel® Core™ i9-9900K @ 3,60 GHz (16 CPUs)
Arbeitsspeicher	32 GByte
Grafikkarte	NVIDIA GeForce RTX 2080 Ti
Grafikspeicher	11049 MByte
Betriebssystem	Windows 10 Pro 64-bit

Tabelle 3.2: Das verwendete Windows-System zur Ausführung der *VR*-Anwendung.

3.6 Koordinatensysteme

Die verschiedenen Technologien verwenden bei der Darstellung von 3D-Grafik beziehungsweise zur Steuerung des Roboters unterschiedliche dreidimensionale Koordinatensysteme, zwischen denen an geeigneter Stelle konvertiert werden muss. *ROS* verwendet, wie in der Robotik üblich, ein rechtshändiges Koordinatensystem, in dem die X-Achse nach vorne und die Z-Achse nach oben zeigt, während *Unity* ein linkshändiges Hauptachsensystem der Computergrafik verwendet, bei dem die Z-Achse nach vorne und die Y-Achse nach oben zeigt. *OpenVr* wiederum verwendet ein rechtshändiges Koordinatensystem, das, bis auf eine gespiegelte Z-Achse, jenem von *Unity* entspricht. Die Koordinatensysteme sind in Tabelle 3.3 aufgelistet. Die Umrechnung der Koordinaten für die Verwendung in *ROS* ist im erwähnten *ROS-Sharp Plugin* bereits integriert, muss jedoch für die anderen Fälle manuell implementiert werden.

System	Ausrichtung	X-Achse	Y-Achse	Z-Achse
Unity	linkshändig	rechts	oben	vorne
OpenVr	rechtshändig	rechts	oben	hinten
ROS	rechtshändig	vorne	links	oben

Tabelle 3.3: Vergleich der Koordinatensysteme von *Unity*, *OpenVr* und *ROS*.

Von Vorteil ist der Umstand, dass alle drei Systeme mit dem metrischen System arbeiten und alle Distanzeinheiten in Metern erfolgen. So ist keine Umrechnung zwischen den Größen unterschiedlicher Skalen notwendig. Auflistung A.19 zeigt die Umrechnung von *OpenVr*- in *ROS*-Koordinaten. *OpenVr* liefert diese in Matrixnotation, während *ROS* hauptsächlich eine Kombination aus Punkt- und Quaternion-Datentypen verwendet (siehe Auflistung A.4 und Auflistung A.5).

KAPITEL 4

Design

Dieses Kapitel beschreibt die Designüberlegungen, die zur Umsetzung der in Abschnitt 1.2 (Ziele) definierten Ziele führen. Hauptaugenmerk liegt dabei auf dem *mobilen Manipulator RB-Kairos*, dessen bestehende Infrastruktur in Abschnitt 4.1 (Ausgangssituation) erklärt wird. Abschnitt 4.2 (Entwurf des VR-Systems) beinhaltet die Überlegungen, die zur weiteren Implementierung des *VR*-Systems dieser Arbeit führen.

Abbildung 4.1 gibt einen groben Überblick über die vorhandenen Hard- und Softwarekomponenten und jene, die im Zuge dieser Diplomarbeit erstellt werden. Zur besseren Veranschaulichung werden thematisch zusammengehörende Komponenten zu Modulen zusammengefasst, wobei Hardware mit rechteckigem Rahmen und Software mit abgerundetem Rahmen dargestellt werden.

Die Hardwarekomponenten in grau und die Softwaremodule in weiß sind bereits vorab am Roboter vorhanden und werden ohne große Änderungen übernommen. Diese Module und deren Interaktionen werden in Abschnitt 4.1 (Ausgangssituation) erklärt. Bei den Komponenten in violett und blau handelt es sich um Hard- und Software, die im Rahmen dieser Diplomarbeit erstellt oder hinzugefügt und in Abschnitt 4.2 (Entwurf des VR-Systems) erläutert werden. Die grünen Elemente sind Teil des *VR*-Systems *Vive* Pro und werden in Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos) und Abbildung 5.9 zur Erfassung von Roboter- und Benutzerkoordinaten ins System integriert.

4.1 Ausgangssituation

Zu den bereits am *RB-Kairos* vorhandenen Softwarekomponenten gehört der in Abschnitt 3.3 (Robot Operating System) erwähnte *roscore*, der den Kern des Systems bildet und auf dem integrierten Computer des *RB-Kairos* ausgeführt wird. Er verwaltet die Ausführung und Kommunikation der übrigen *Nodes* über ein Netzwerkprotokoll und ermöglicht dadurch eine einfach zu nutzende Datenübertragung zwischen mehreren

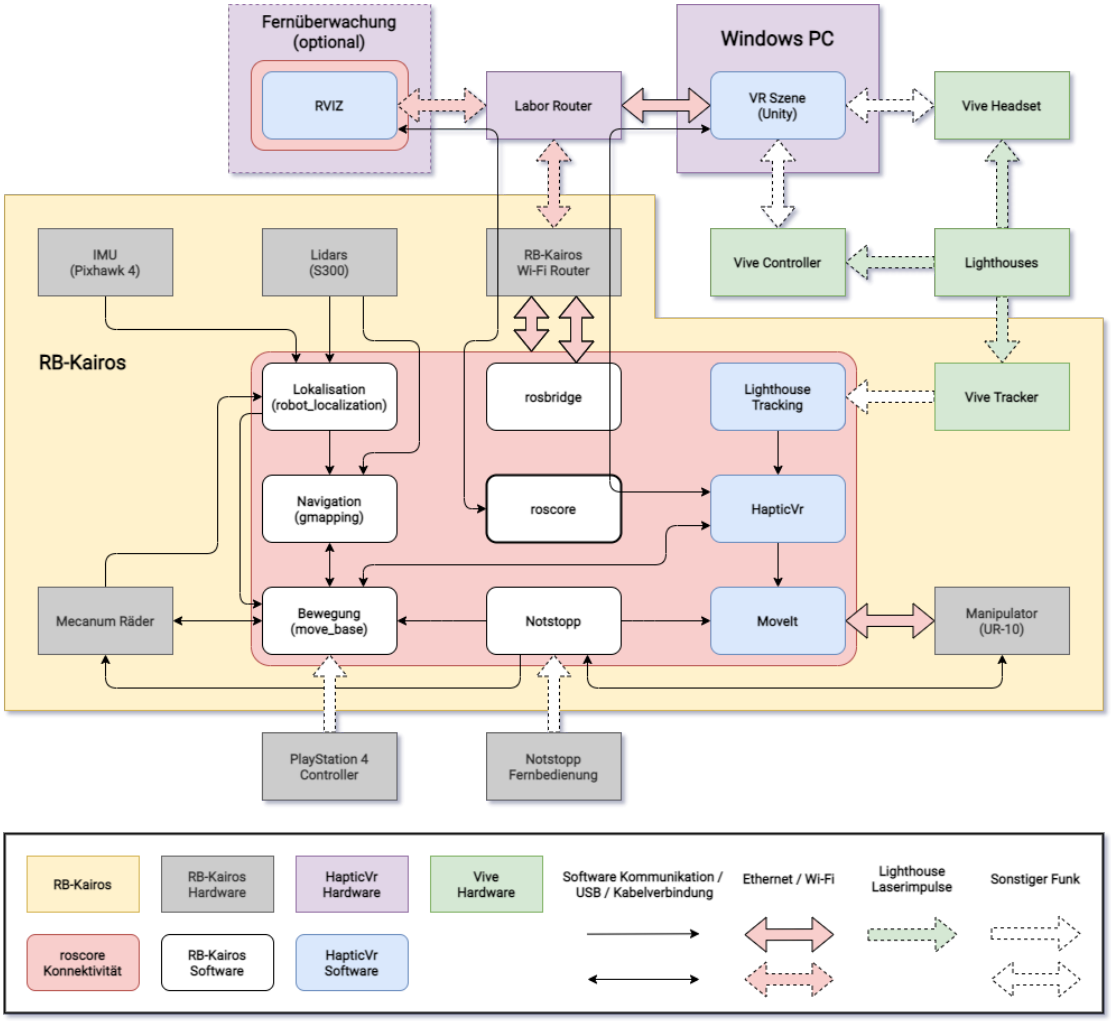


Abbildung 4.1: Das Kommunikationsdiagramm der wichtigen Hard- und Softwarekomponenten des *VR*-Systems dieser Diplomarbeit.

ROS-kompatiblen Geräten im selben Netzwerk. Diese Konnektivität wird jedoch in der Anfangskonfiguration nicht benötigt, da der integrierte Computer alle ihm gestellten Aufgaben alleine erledigen kann. Sie hilft jedoch bei der Fernüberwachung des Systems durch einen zweiten Rechner, der mit der gleichen *ROS*-Version wie der *roscore* ausgestattet sein muss. Die Konnektivität ist in der Grafik mit einem roten Rahmen markiert.

Geräte, die die Anforderungen an die *ROS*-Version nicht erfüllen, da sie zum Beispiel ein inkompatibles Betriebssystem verwenden, können stattdessen ihre Kommunikation mit dem System über die *rosbridge* abwickeln. Es handelt sich dabei um eine auf *JSON* basierte Webschnittstelle, die das *ROS*-Protokoll imitiert. Obwohl sie nicht alle Funktionen einer vollen *ROS*-Kommunikation zur Verfügung stellt, ist sie ausreichend für einen einfachen Datenverkehr und am Roboter bereits vorinstalliert.

Der *RB-Kairos* kommuniziert mit seinen Hardwarekomponenten über unterschiedliche Schnittstellen. Zu ihnen gehören unter anderem Ethernet und USB. In Abbildung 4.1 werden diese mit breiten und schmalen Pfeilen simplifiziert dargestellt, da zum Beispiel der Anschluss eines Hardwaregeräts über USB auch immer einen Treiber als Softwarekomponente enthält. Um genügend Netzwerkverbindungen zur Verfügung zu stellen, ist der Roboter mit einem Wi-Fi-Router und einem Switch ausgestattet. Zusammen ermöglichen sie den Anschluss von sechs Geräten via Ethernet ($zwei \times vier$ RJ-45 Anschlüsse abzüglich *zwei* belegter Buchsen für deren Verkabelung). Per Ethernet verbunden sind zum Beispiel der Rechner und der *UR-10* Roboterarm. Für die Kommunikation nach außen sorgt hingegen der Router mit Wireless-Access-Point. Dies ist auch der eleganteste Weg, den vorhin erwähnten Zweitrechner zur Fernüberwachung ins *ROS*-Netzwerk zu bringen. An der Hinterseite des Roboters befinden sich außerdem zwei RJ-45 Buchsen, die von jeweils einem LAN- und WAN-Anschluss des Routers nach außen geleitet werden. Ebenso nach außen geführt sind ein HDMI- und zwei USB-Anschlüsse des Rechners. An sie können Tastatur, Maus und Monitor zur direkten Arbeit am Roboter angeschlossen werden.

Per USB mit dem integrierten Rechner verbunden sind die zwei *Lidars*. Sie tasten ihre Umgebung in einem jeweils 270° breiten Bereich ab und decken durch ihre diagonale Montage den gesamten Bereich um den Roboter herum ab. Die erzeugten Entfernungsdaten werden per `LaserScan` *Message* (Auflistung A.15) an *ROS* übergeben und zur Lokalisation des Roboters im Raum genutzt.

Eine weitere Hardwarekomponente, die der *RB-Kairos* zur Ortung seiner Position verwendet, ist eine *IMU*. Es liefert Lage- und Beschleunigungsdaten anhand der Messung des Erdmagnetfeldes und der Trägheitssensoren und wird als `Imu` *Message* (Auflistung A.12) an *ROS* übermittelt.

Zusammen mit der *Odometrie*, also der Berechnung der Positionsveränderung des Roboters anhand der Rotationssensoren seiner Räder, und einer zuvor aufgezeichneten Karte der Umgebung kann sich der *RB-Kairos* im Raum lokalisieren. Er verwendet dazu das *ROS-Package* `robot_localization` [robb] und ein Verfahren namens *Adaptive Monte Carlo Localization* (*AMCL*). Es vergleicht die Sensordaten der Laserscanner mit jenen, die durch zufällige Auswahl von Roboterpositionen innerhalb der Karte entstehen müssten.

Die Auswahl geschieht nach *Monte-Carlo*-Schema, wobei das Auswahlsample je nach Positionswahrscheinlichkeit adaptiert wird. Eine genaue Erklärung dieses Verfahrens ist in Dellaert et al [DFBT99] zu finden.

Die aktuelle Position und die Karte der Umgebung dienen dem *RB-Kairos* zur Navigation. Die Koordinaten des *Base-Footprint* und der *Odometrie* werden als `TransformStamped` (Auflistung A.10) im *Transform Tree* verwaltet. Die Karte liegt als `OccupancyGrid` (Auflistung A.13) vor und entspricht in etwa einem Graustufen-Bitmap mit acht Bit Bittiefe. Zusätzlich greift das Verfahren auf aktuelle *Lidar*-Daten zu, um plötzlich auftretende Hindernisse zu erkennen und ihnen auszuweichen. Der Roboter verwendet das Navigationsverfahren *Simultaneous Localization And Mapping (SLAM)* des *ROS-Packages* `gmapping` [gma] zur Erstellung eines Navigationspfads. Das Verfahren ermöglicht es dem Roboter, sich gleichzeitig in einer unbekannten Umgebung zurechtzufinden, eine Karte zu erstellen und anhand dieser zu navigieren. Es sei jedoch angemerkt, dass die voraufgezeichnete Karte der Lokalisierung vom *SLAM*-Algorithmus unberührt bleibt, da es sich dabei nur um eine ideale Übersichtskarte handelt. Die *SLAM*-Karte spiegelt jedoch den aktuellen Umgebungsplan wider, inklusive aller etwaigen Umgebungsveränderungen und Hindernisse.

Den Anstoß, eine Navigationsplanung durchzuführen, erhält das System durch die Bewegungssteuerung des Roboters, die im *ROS-Package* `move_base` [movb] bereitgestellt wird. Es handelt sich dabei um die Implementierung eines *ActionServers* des Pakets `actionlib` [act]. Die Grundüberlegung dieses Moduls ist es, Robotern bestimmte Aufgaben (*Actions*) aufzutragen, die durch Erreichen eines bestimmten Zielzustands (*Goal*) erfüllt werden. Im Falle der Bewegungsausführung ist das die Ankunft des Roboters am Zielort. Es kann jedoch nicht immer davon ausgegangen werden, dass die vorgegebenen Ziele auch erreicht werden. Zum Beispiel könnte der Weg zum Ziel blockiert sein oder es trotz Vorsichtsmaßnahmen zu Zusammenstößen kommen. In diesen Fällen ist eine weitere Ausführung der Aufgabe kontraproduktiv und sollte abgebrochen werden. Das System muss darüber ebenso informiert werden, wofür *ActionServer* neben *Goal*- auch *Feedback-Topics* bereitstellen und die Möglichkeit bieten, die *Action* auch wieder abzubrechen.

Die *Action* der `move_base` wird in Auflistung A.16 angeführt. Sie definiert das *Goal-Topic* `target_pose` und das *Feedback-Topic* `base_position`, beide vom Typ `PoseStamped` (Auflistung A.11). Es handelt sich dabei um Positionsangaben, die mit *Frame-* und Zeitangaben versehen sind. Eine Definition des *Result-Topics* ist in dieser *Action* nicht notwendig.

Aus dem Bewegungspfad, den das Navigationsystem aus den gegebenen Informationen erstellt, werden nun konkrete Richtungsangaben in der Form von `Twist` *Messages* (Auflistung A.6) extrahiert. Diese wiederum werden vom Bewegungsmodell des *RB-Kairos* in Steuersignale für seine vier *Mecanum-Räder* umgewandelt. Der Roboter setzt sich also in Bewegung und erhält laufend aktualisierte *Lidar*-Daten seiner Umgebung, die sowhol zur erneuten Positionsbestimmung im Raum, als auch zur Erkennung und Umfahrung von Hindernissen verwendet werden. Die Bewegung wird solange ausgeführt,

bis das *Feedback-Topic* `base_position` (in etwa) mit dem *Goal-Topic* `target_pose` übereinstimmt oder vorher abgebrochen wird.

Zu einem Abbruch kommt es auch, wenn einer der beiden *Notstopp*-Buttons des *RB-Kairos* betätigt wird. Der eine befindet sich auf der Rückseite des Chassis, der andere auf einer Fernbedienung, die am Gürtel des Operators getragen werden kann. Das Notstoppsystem beendet nicht nur die Ausführung der Bewegungs-*Action*, sondern unterbricht auch die Stromversorgung der Radmotoren, sodass der Roboter zu einem unverzüglichen Halt kommt. Eine Wiederaufnahme des Fahrbetriebs ist durch Reaktivierung des *Notstopp*-Buttons und drücken der *Restart*-Taste auf der Rückseite des Roboters möglich.

Neben der autonomen Navigation des *RB-Kairos* kann dieser auch manuell über den, vom Roboterhersteller beigelegten, *PlayStation 4 Controller* gesteuert werden. Die Verarbeitung der Controllereingabe erfolgt durch eine in Abschnitt 3.3 Robot Operating System vorgestellte *Teleop-Node*. Zur validen Steuerung des Roboters setzt diese das Drücken und Halten des sogenannten *Deadman*-Buttons voraus. Sollte dieser nicht betätigt worden sein, so werden alle weiteren Eingaben am *Controller* ignoriert und der Roboter kommt bei Mangel an anderen Steuerkommandos zum Stillstand. Ein Notstopp wird dabei nicht ausgelöst.

Die *Mecanum-Räder* erlauben die Steuerung des *RB-Kairos* über drei Freiheitsgrade. Bei diesen handelt es sich um die linearen Achsen entlang `x` und `y` sowie die Rotationsachse um `z` im Koordinatensystem von *ROS* (Tabelle 3.3). Die Achsengeschwindigkeiten werden, genauso wie die Richtungsangaben der Bewegungssteuerung, als `Twist` *Messages* in *ROS* publiziert. Bevor beide Signale jedoch an das Bewegungsmodell des Roboters übergeben werden, gelangen sie in die *Multiplexer-Node* des `twist_mux` *Packages* [twi].

Obwohl *ROS* die Möglichkeit bietet mehrere *Publisher* gleichzeitig auf einem *Topic* veröffentlichen zu lassen, ist dies nicht immer wünschenswert. So auch bei der Steuerung der Bewegungsgeschwindigkeit des Roboters. Zum Beispiel ist ein Lenkmanöver der autonomen Navigation nach rechts und eine unmittelbare Steuerung nach links durch Benutzereingabe für beide Arten nicht zielführend. Aus diesem Grund werden im `twist_mux` die verschiedenen Inputs mit Prioritäten versehen und niedriger gereihte Signale zugunsten der höheren ignoriert. In typischen Robotersetups, und auch im Fall des *RB-Kairos*, haben Benutzereingaben Priorität gegenüber der autonomen Navigation.

Die wichtigste Komponente am Roboter ist der Manipulator *UR-10*. Seine Steuerung wurde bereits kurz in Abschnitt 3.4 UR-10 beschrieben. Der Roboterarm ist an der Oberseite des Roboters angebracht und besitzt seinen eigenen Computer im Inneren des Chassis. Er hat zudem seinen eigenen Einschaltknopf, siehe Abschnitt A.2 (Hochfahren des RB-Kairos), und ebenso, wie der Hauptrechner des *RB-Kairos*, USB- und HDMI-Anschlüsse an der Außenseite. Der Hauptrechner kommuniziert mit dem *UR-10* per Ethernet, jedoch ist der Roboterarm auch direkt mit dem Notstoppsystem verbunden.

Die mit dem *UR-10* kommunizierende *ROS*-Komponente ist das bereits erwähnte *Motion-Planning-Framework MoveIt*. *MoveIt* implementiert intern einen *ActionServer* und führt *Actions* vom Typ `MoveGroup` (Auflistung A.17) aus. Die Aufgaben eines *Manipulators*

sind umfangreich und werden in der *MoveGroup*-Konfiguration als Sub-*Actions* definiert. Zu ihnen zählen beispielsweise das Abfahren eines Bewegungspfads (*FollowJointTrajectory*) und das Öffnen und Schließen eines Greifers (*GripperCommand*). Der *RB-Kairos* wurde für die TU Wien als Forschungsroboter ohne Gripper und ohne bestimmte Anforderung geliefert. Aus diesem Grund ist das *Motion Planning Framework* am Roboter nur rudimentär vorkonfiguriert und der Roboterarm beherrscht in seiner Werkseinstellung nur eine Demonstration seiner Bewegungsfähigkeiten. Die Implementierung der Manipulationsroutine dieser Diplomarbeit wird in Abschnitt 5.3 (Konfiguration von UR-10 und MoveIt) beschrieben.

4.2 Entwurf des VR-Systems

Dieser Abschnitt beschäftigt sich mit den Designüberlegungen, die zur Umsetzung des in dieser Arbeit vorgestellten *VR*-Systems führen. Grundaufgabe ist es, einen Konzeptbeweis zu liefern, wie haptisches Feedback in einer raumgroßen *VR*-Simulation durch einen mobilen Roboter zur Verfügung gestellt werden kann. Dazu soll ein physisches Objekt auf Kommando an einer Stelle im Raum platziert werden, an der Benutzer und Benutzerinnen der Simulation es in der *virtuellen Realität* erwarten würden.

Eine der einfachsten physischen Requisiten für diesen Zweck ist eine quadratische Holzplatte. Sie weist eine simple und flache Geometrie auf, ist stabil und ihre reativ glatte Oberfläche sorgt für ein wiedererkennbares haptisches Gefühl bei der Berührung. Sie lässt sich außerdem sehr leicht mit vier Schrauben am *Tool Center Point (TCP)* des *UR-10* anbringen. Die Holzplatte wird im Verlauf dieser Arbeit *Haptikwand* genannt.

Der *RB-Kairos* soll die *Haptikwand* in weiterer Folge an beliebigen Positionen um sich herum platzieren. Um diese Platzierung selbstständig durchführen zu können, muss *ROS* über die Präsenz der Platte aufgeklärt werden, damit es bei einer Bewegung zu keiner Eigenkollision kommt. Das 3D-Modell des Roboterarms wird daher erweitert und aus diesem in *MoveIt* eine neue Gelenks-*Kinematik* erstellt. Das Modul ist in Abbildung 4.1 blau markiert.

Die autonome Navigation des *RB-Kairos* im Raum wird bereits durch die Werkskonfiguration zur Verfügung gestellt und kann deshalb weiterverwendet werden. Sie wird in weiterer Folge mit der Bewegungsausführung des Roboterarms kombiniert, um eine Platzierung der *Haptikwand* im gesamten Raum zu ermöglichen. Im Kommunikationsdiagramm wird dieser Schritt als *HapticVr* dargestellt. Das Modul wird Befehle von der noch zu erstellenden *VR*-Szene erhalten und kommuniziert diese an *MoveIt* und die Bewegungsausführung weiter.

Die Position des Benutzers oder der Benutzerin im virtuellen Raum wird durch die *Lighthouse*-Technologie ermittelt. Das *Headset* und die *Controller*, in Abbildung 4.1 grün markiert, empfangen die Lasersignale, die von den Basisstationen ausgesandt werden. Der Windows-PC, der diese Daten auswertet, berechnet daraus die Koordinaten der *Vive*-Komponenten. Diese Koordinaten haben jedoch keinerlei Bezug zu den Raumko-

ordinaten des Roboters, die bei Lokalisation und Navigation verwendet werden. Es handelt sich bei beiden um voneinander unabhängige Systeme. Um die bereits bestehende Navigationsroutine des *RB-Kairos* weiterverwenden, ihr aber Zielkoordinaten anhand einer *VR*-Simulation der *Vive* auftragen zu können, muss ein gemeinsamer Bezugspunkt geschaffen werden. Am einfachsten zu realisieren ist das durch zusätzliches Tracking des Roboters mittels *Lighthouse*-Technologie.

Um den *RB-Kairos* im selben Koordinatensystem, wie dem der *Vive* zu erfassen, kann spezielle Erweiterungshardware des *VR*-Systems, die sogenannten *Tracker*, am Roboter angebracht werden. Damit lassen sich die Koordinaten des Roboters im Kontext der *VR*-Anwendung ermitteln. Eine Umwandlung ins Koordinatensystem von *ROS* erfolgt durch Kalibration. Die Auswertung der *Tracker* am Roboter sollte nicht durch den Windows-PC, sondern durch den Computer des *RB-Kairos* erfolgen. So kann sichergestellt werden, dass der Roboter zur Erfassung seiner Position nicht auf externe Geräte angewiesen ist. Ein Betrieb der benötigten Trackingsoftware *SteamVr* unter Linux ist dafür Voraussetzung.

Der verbleibende Baustein dieses Systems ist die *VR*-Anwendung und damit die grafische Darstellung der Benutzererfahrung. Aufgrund der besonderen Hardwarevoraussetzungen wird diese nicht am *RB-Kairos*, sondern auf einem zusätzlichen Windows-Computer ausgeführt. Er sorgt für die Erfassung der Userkoordinaten und die Berechnung der *virtuellen Szene*. Aus den bereits erwähnten Vorteilen kommt dabei das *Framework Unity* zum Einsatz. Für die Kommunikation zwischen den beiden Systemen bietet sich die *rosbridge* an, die in *Unity* durch das *ROS-Sharp Plugin* [rosb] unterstützt wird.

Die *VR*-Anwendung soll als Ausgangspunkt für die Steuerung des Roboters dienen. Einem Benutzer oder einer Benutzerin wird eine *virtuelle Umgebung* präsentiert in der er oder sie sich aufhalten kann. Die zu sehenden Objekte existieren alle nur virtuell und können passiert werden. Auf Befehl des Users soll eines dieser Objekte mit einer Haptik versehen werden, indem der Roboter die *Haptikwand* an die entsprechende Stelle platziert, an der sie ein User in der virtuellen Welt erwarten würde. Idealerweise stimmen die Oberflächeneigenschaften der quadratischen *Haptikwand* mit jenen des virtuellen Objektes überein, daher bieten sich Objekte mit gerader Oberfläche zur Simulation, zum Beispiel Tische, Wände oder Würfel, an.

Die Selektion eines Objekts könnte mittels eines virtuellen Laserpointers stattfinden. *Unity* bietet als Spiele-*Framework* bereits vorgefertigte Algorithmen zur Berechnung von dreidimensionaler Geometrie, unter anderem auch eine *Raycasting*-Funktion, mit der eine Linie auf Intersektion mit Kollisionsgeometrie getestet werden kann. Der so ermittelte Schnittpunkt könnte als Platzierungspunkt der *Haptikwand* herangezogen werden, die mittig am *TCP* des Roboterarms angebracht wird. Die *Raycasting*-Funktion liefert neben dem Schnittpunkt der Geraden mit der Geometrie auch das getroffene Objekt zurück. Dieses könnte auch einen anders definierten Platzierungspunkt der *Haptikwand* bestimmen.

Der vom User und der Geometrie bestimmte Platzierungspunkt wird in weiterer Folge als `PoseStamped`-*Message* durch das *ROS-Sharp Plugin* in *ROS* veröffentlicht. Der

Empfang dieser Nachricht ist zugleich der Befehl, den *RB-Kairos* und die *Haptikwand* in Stellung zu bringen.

Zur Anbindung des Windows-PC an das Netzwerk des *RB-Kairos* und zur Übertragung des Platzierungspunkts an *ROS* wird ein zusätzlicher Router mit Wi-Fi-Access-Point angeschafft. Die Netzwerkverbindung muss so eingerichtet werden, dass sich alle Komponenten des *VR*-Systems im selben Netzwerk befinden. Dies kann erfolgen, indem einer der Router in den *Repeat*-Modus gesetzt wird, also die Netzwerkparameter des anderen widerspiegelt. Zusätzlich soll das Netzwerk einen Zugang zum Internet erhalten, um benötigte Software und Updates herunterladen zu können.

KAPITEL 5

Umsetzung

Dieses Kapitel enthält die Umsetzung der in Abschnitt 4.2 (Entwurf des VR-Systems) beschriebenen Designüberlegungen und die Konfiguration und Implementierung der Hard- und Softwaremodule. Abschnitt 5.1 (Hardwareinstallationen) gibt einen Überblick über die zusätzliche Hardware, die zum Betrieb des *VR*-Systems am Roboter angebracht wird. Abschnitt 5.2 (Netzwerkkommunikation) beschreibt die Herstellung eines gemeinsamen Wi-Fi-Netzwerks, über das die Computer dieser Arbeit kommunizieren können. Abschnitt 5.3 (Konfiguration von UR-10 und MoveIt) beschäftigt sich mit der Inbetriebnahme des in dieser Arbeit vorgestellten *Manipulators* und seiner Integration in *ROS*. Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos) zeigt, wie der Roboter seine Position anhand der *Lighthouse*-Technologie der *Vive* ermitteln und anhand dieser Koordinaten im Raum navigieren kann. Abschnitt 5.5 (Integration des UR-10) beschreibt, wie die Bewegungsausführung des Roboterarms mit der Navigationslösung des Roboters kombiniert wird, um die Platzierung der *Haptikwand* im gesamten Raum zu ermöglichen. Abschnitt 5.6 (VR-Anwendung) beschäftigt sich schließlich mit der Erstellung einer *VR*-Anwendung und ihre Anbindung an das Robotiksystem in *ROS*. Abschnitt 5.7 (Inbetriebnahme) listet die Schritte auf, die ausgeführt werden müssen, um die gesamte Robotikanwendung in Betrieb zu nehmen und für ein reibungsloses Zusammenspiel der Komponenten zu sorgen.

5.1 Hardwareinstallationen

Im Zuge dieser Arbeit fanden einige Harwareerweiterungen am *RB-Kairos* statt. Um den Roboter mit der *Lighthouse*-Technologie der *Vive* zu orten werden zwei *Tracker* an seiner Gehäuseoberfläche angebracht. Die *Tracker* sind auf ihrer Unterseite mit 1/4-Zoll-Stativgewinden ausgestattet, die es ermöglichen, sie auf eine eigens hergestellte Acrylglasschiene zu schrauben. Die Schienen werden mit Hilfe der Befestigungslöcher auf der Oberseite des *RB-Kairos* angebracht. Die *Tracker* haben Akkus integriert und werden

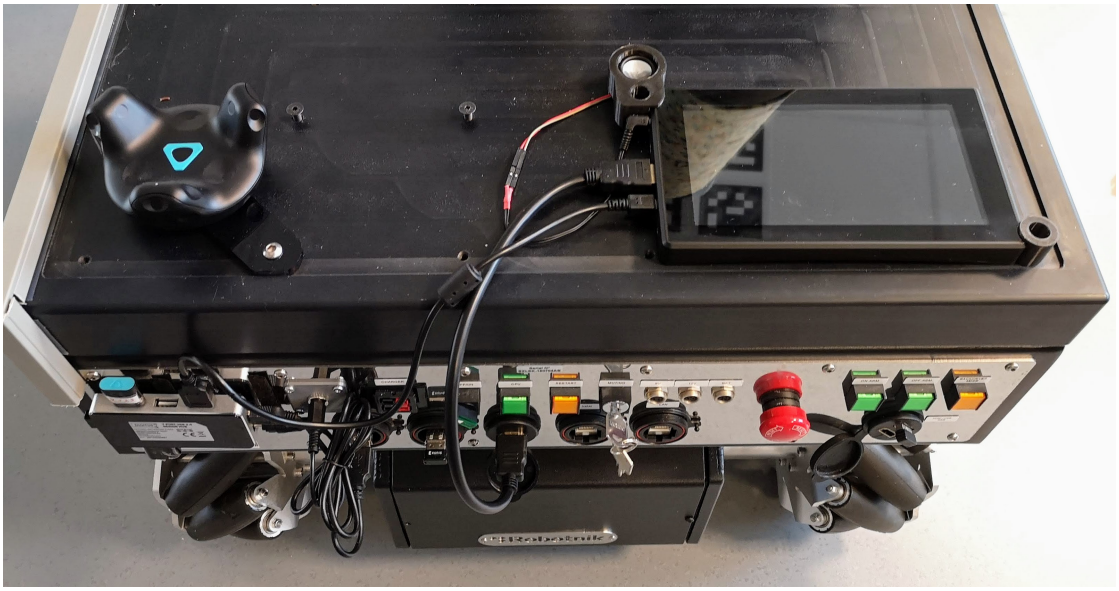


Abbildung 5.1: Detailansicht der Hinterseite des *RB-Kairos*. Zu sehen sind unter anderem der hintere *Vive-Tracker*, das Display und der USB-Hub.

per USB geladen. Ihre Kommunikation mit Computern erfolgt trotz der vorhandenen USB-Buchse ausschließlich durch die beigelegten USB-Empfänger.

Die *Tracker* empfangen die Lichtsignale von *Lighthouse*-Basisstationen. Von ihnen sind bereits drei Stück an den Wänden beziehungsweise an Stativen in Deckennähe angebracht. Sie decken dabei einen Trackingbereich von circa $8 \times 6,3$ m ab. Im Zuge dieser Diplomarbeit bedarf es keiner weiteren Anpassungen der *Lighthouses*.

Um die USB-Empfänger zu betreiben und gleichzeitig die *Tracker* mit Strom zu versorgen, reichen die zwei USB-Anschlüsse auf der Rückseite des Roboters nicht aus. Aus diesem Grund wird dort zusätzlich ein USB-Hub angebracht. Um seine Stromversorgung sicherzustellen, wurde eine 5-Volt-Stromleitung aus dem Inneren des Chassis nach außen gelegt und der Hub an diese angeschlossen.

Der *RB-Kairos* wird weiters mit einem 7-Zoll-Touchscreen ausgestattet, um Funktionen des Roboters am Gerät selbst starten zu können. Außerdem kann damit der Status der *Tracker* leicht überwacht werden. Für den Betrieb des *Lighthouse*-Trackings ist das Display sogar essenziell, da *SteamVr* sonst nicht gestartet werden kann. Mehr dazu in Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos). Der Touchscreen wird per externem HDMI-Anschluss und USB-Hub mit dem internen Computer verbunden. Die Positionierung des Displays erlaubt außerdem einen Anschluss an die externen Ports des *UR-10*, die sich ebenfalls auf der Rückseite des Roboters befinden. Abbildung 5.1 zeigt die Hardwarekomponenten, die am Roboterchassis angebracht wurden.

Der letzte Zubau zum Roboter ist die Montage der *Haptikwand*. Bei ihr handelt es

sich um eine quadratische Holzplatte mit den Maßen $0,01 \times 0,6 \times 0,6\,\mathrm{m}$. Mit vier Gewindeschrauben wird diese an den Montagelöchern des *TCP* des *UR-10* angebracht. Abbildung 5.2 zeigt den gesamten *RB-Kairos* inklusive *Haptikwand* und zusätzlicher Komponenten.

5.2 Netzwerkkommunikation

Wie in Abschnitt 4.2 (Entwurf des VR-Systems) erwähnt, soll die Kommunikation des *RB-Kairos* mit dem Windows-Rechner per Wi-Fi erfolgen. Die einfachste Art, dies zu bewerkstelligen und das System zusätzlich mit einer Internetverbindung zu versorgen, ist die Integration eines weiteren Wi-Fi-Routers. Der zusätzliche Router wird an seinem WAN-Port mit einer der Netzwerksteckdosen im Labor der TU Wien verbunden. An einer seiner LAN-Buchsen wird der Windows-PC angeschlossen. Das Wi-Fi Netzwerk `HapticVr` sorgt für einen kabellosen Zugang zu diesem Netzwerk.

Der *RB-Kairos* betreibt auf seinem internen Router ein Wi-Fi-Netzwerk names `rbkairos`. Das Ziel ist, die beiden Netzwerke zu koppeln und ihre Kommunikation herzustellen. Da der Router im Labor über einen Zugang zum Internet verfügt, wird dieser auch die Bereitstellung der Netzwerkfunktionen übernehmen. Der interne Router wird hingegen umkonfiguriert und in den *Repeat*-Modus versetzt. Dieser spiegelt nun das `HapticVr`-Netzwerk und leitet die Kommunikation des externen Netzwerks an die internen Komponenten im Roboter weiter. Auf diese Art erhalten die internen Geräte auch Zugang nach außen und zum Internet.

Die internen Netzwerkgeräte des *RB-Kairos* werden über statische IP-Adressen angesprochen. Die Adressen und der IP-Adressbereich müssen daher im `HapticVr`-Netzwerk genau wie im `rbkairos`-Netzwerk eingestellt werden, um eine unnötige Neukonfiguration von *ROS* zu vermeiden. Mit dieser Methode haben nun auch Computer des Labor-Netzwerks einen einfachen Zugang zum *roscore* des *RB-Kairos*. Wie erwähnt, ist der Betrieb derselben Betriebssystem- und *ROS*-Versionen auf den Geräten Voraussetzung. Diese erfüllt der ans Netzwerk angeschlossene Windows-Rechner nicht, weshalb in Abschnitt 5.6 (VR-Anwendung) näher auf die geschmälerte Kommunikation über die *rosbridge* eingegangen wird.

Andere Computer, die die Anforderungen an *ROS* erfüllen, haben aber durch Netzwerkkontakt zum *roscore* ungeschränkten Zugang zu allen Features. Zu diesem Zweck wwerden auf einem Notebook *Ubuntu 16.04* und *ROS Kinetic Kame* installiert und die IP-Adresse des Robotercomputers als Netzwerkstandort des *roscore* konfiguriert. Die Hauptaufgabe des Notebooks ist die Fernwartung des Systems und das Debugging der *ROS-Topics* durch Konsolenausgabe und *RVIZ*. Zusätzlich können per *SSH*-Zugang zum Roboterrechner Prozesse und *Nodes* beendet und neu gestartet werden, während sich der *RB-Kairos* an einer beliebigen Stelle im Raum aufhalten kann.

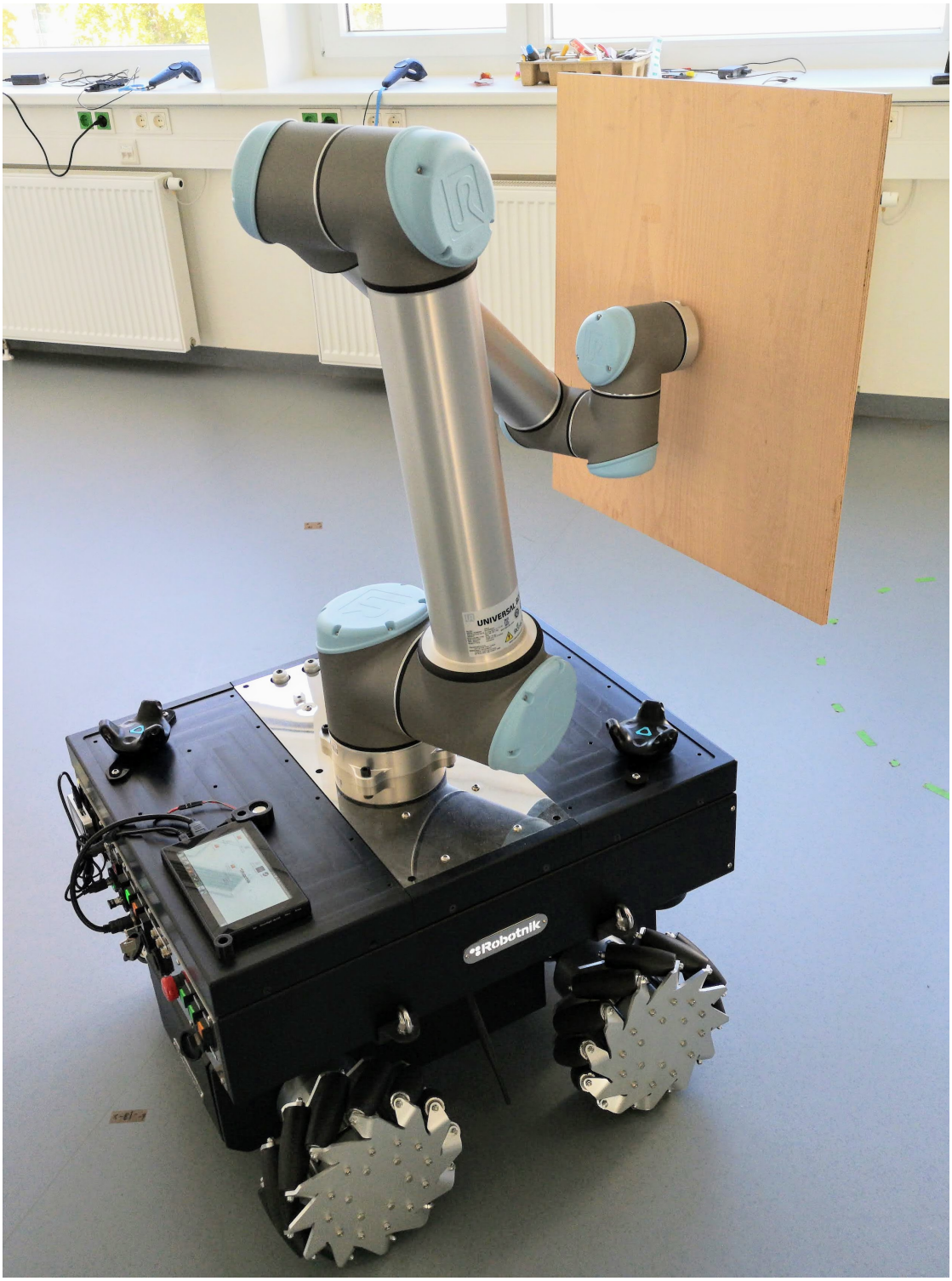
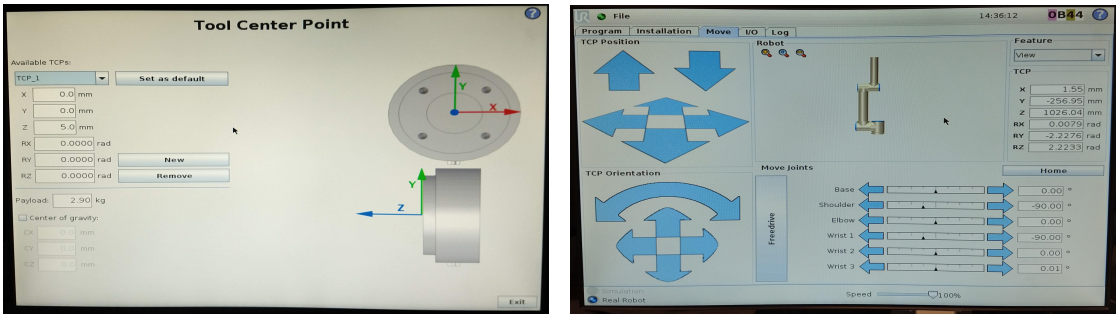


Abbildung 5.2: Gesamtansicht des Roboters mit am *Endeffektor* angebrachter Wandattrappe (*Haptikwand*).



(a) Einrichtung des *TCP* an der Spitze des Roboterarms.

(b) Das *Move* Interface zur manuellen Bewegung des *Endeffektors*.

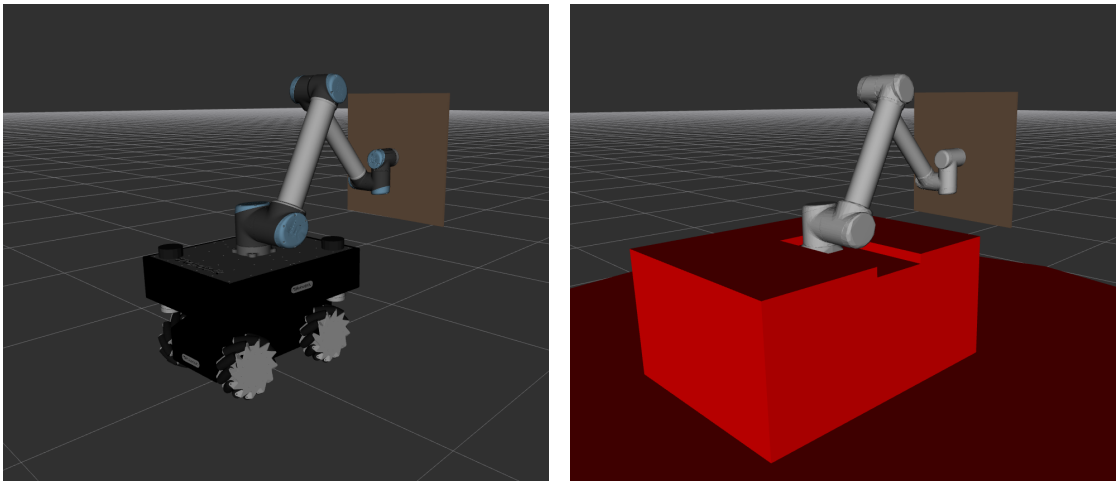
Abbildung 5.3: *Polyscope*, das User Interface des *UR-10*.

5.3 Konfiguration von UR-10 und MoveIt

Eine der Komponenten des *RB-Kairos*, die über das Netzwerk angesprochen werden können, ist der Roboterarm *UR-10*. Um eine Initialkonfiguration des *Manipulators* vornehmen zu können, müssen zunächst Monitor und Eingabegeräten angeschlossen werden. Es kann auch das Touchdisplay verwendet werden, das bereits am Roboter angebracht wurde und normalerweise an den Robotercomputer angeschlossen ist. Abbildung 5.3 zeigt das User-Interface *Polyscope* des *UR-10*. Es dient zur Einrichtung des Arms, zum Laden von Roboterprogrammen und bietet Möglichkeiten zur rudimetären Steuerung der Gelenke. In einem Setupschritt wird zunächst die Nutzlast angepasst, da sich nun die circa 2,9 kg schwere *Haptikwand* am *TCP* befindet. Weiters wird das Netzwerkinterface des *Manipulators* aktiviert. Es wird zur Kommunikation mit *ROS* benötigt und ist standardmäßig deaktiviert.

Alle weiteren Einstellungen des Arms erfolgen über *ROS*. Der Hersteller liefert den *RB-Kairos* mit einer vorkonfigurierten, dreidimensionalen Beschreibung des Roboters aus. Sie liegt im *Xacro*-Format vor und beinhaltet ein 3D-Modell des Roboters sowie eine hierarchische Beschreibung seiner Gelenksdefinitionen. Bei diesen handelt es sich um die Angabe der Gelenksart, zum Beispiel Rotations- oder Translationsgelenk und das Bewegungsausmaß des Gelenks. Die Gelenke des *UR-10* bieten zum Beispiel einen Rotationsbereich von fast 720°. Die Roboterbeschreibung wird um die in Abschnitt 5.1 (Hardwareinstallationen) zusätzlich angebrachte Hardware erweitert. Dies ist notwendig, um den Roboterarm über die bislang unbekannten Komponenten aufzuklären und so Eigenkollisionen während der Bewegungsausführung zu vermeiden.

Die *Haptikwand* wird als Quader mit den Maßen $0,01 \times 0,6 \times 0,6\,\mathrm{m}$ definiert und inklusive gleich großer Kollisionsgeometrie an den *Endeffektor*-Link des Roboterarms angehängt. Bei den übrigen Aufbauten genügt die alleinige Bekanntgabe einer neuen Kollisonsgeometrie, die als rechtwinkeliges Volumen mit einer Höhe von 0,04 m auf der Oberseite des Roboterchassis platziert wird. Der Fußboden wird ebenfalls mit einem Kollisionsobjekt versehen, um dem Roboterarm eine Bewegung des Arms gen Boden zu



(a) Das aktualisierte 3D-Modell des *RB-Kairos* inklusive *Haptikwand*.

(b) Die neue Kollisionsgeometrie in rot und die vordefinierte des *UR-10* in silber.

Abbildung 5.4: Die dreidimensionale Beschreibung des Roboters. Ausschnitt aus *RVIZ*.

untersagen. Abbildung 5.4 zeigt das erweiterte 3D-Modell und die Kollisionsgeometrie des *RB-Kairos*. Die zusätzliche Definition des statischen Gelenks der *Haptikwand* ist in Abbildung 3.3 ersichtlich. Die erstellte Roboterdefinition ersetzt in weiterer Folge jene, die bislang beim Start von *ROS* ausgelesen wurde. Sie wird im *Transform Tree* verwaltet.

Mit der neuen Roboterdefinition kann nun ein *Kinematik*-Modell des Roboterarms im *Setup Assistant* von *MoveIt* erzeugt werden. Die Software berechnet zuerst durch die zufällige Annahme von Gelenkswinkeln die möglichen Kollisionen mit sich selbst. So erkennt das Programm *Frames*, die niemals miteinander kollidieren, zum Beispiel die vier Räder untereinander, und daher bei zukünftigen Berechnungen ignoriert werden können. Ignoriert werden können auch jene *Frame*-Paare, die immer kollidieren. Das ist zum Beispiel die in Abbildung 5.4 rot markierte Kollisionsgeometrie mit allen Komponenten, die im Inneren liegen. Das Ziel dieses Vorgangs ist, die Kollisionsmöglichkeiten zu ermitteln, die nur dynamisch auftreten können.

Im nächsten Schritt werden *MoveGroups* definiert. Bei ihnen handelt es sich um Gruppen von Robotergelenken, die zusammen eine Bewegungsausführung definieren. Zum Beispiel können die Gelenke des Arms eine Gruppe bilden und die Fingergelenke eines Greifers eine andere. Die beiden Gruppen lassen sich auf diese Weise unabhängig voneinander ansteuern. Im Roboteraufbau dieser Arbeit gibt es nur eine *MoveGroup*, die alle sechs echten Gelenke des *UR-10* und einige virtuelle Joints enthält. Diese bilden eine hierarchische Kette von der Montageplatte des *UR-10* bis zur *Haptikwand*. Ein Ausschnitt der Kette ist in Abbildung 3.3 zu sehen.

Nach der *MoveGroup*-Definition können in *MoveIt* statische Posen angelegt werden. Bei diesen handelt es sich um vordefinierte Gelenksstellungen, die der Roboter auf Kommando

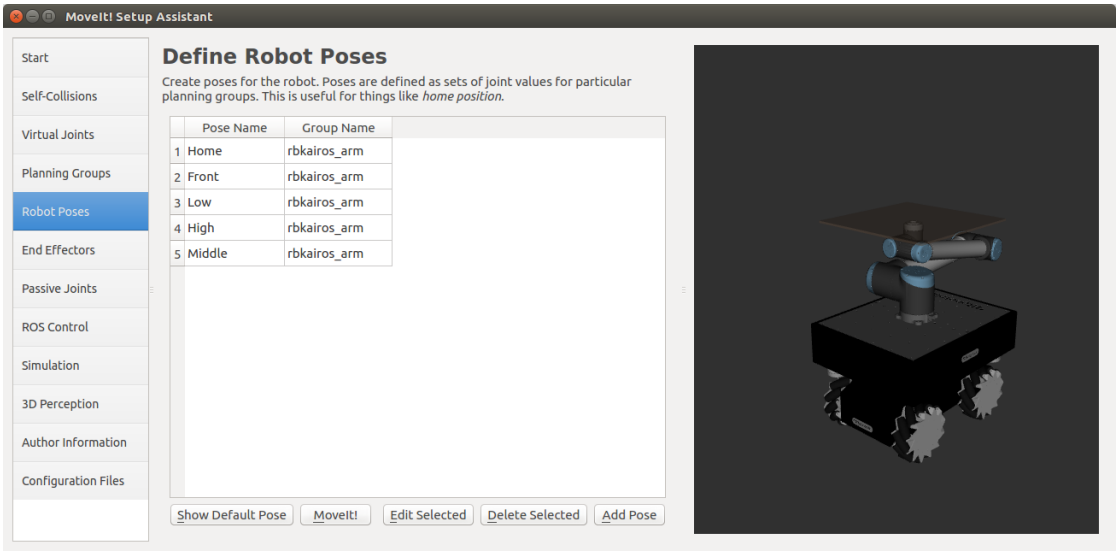


Abbildung 5.5: Einrichtung der Transportstellung des *UR-10* im *Setup Assistant* von *MoveIt*.

einnehmen kann. Es wird eine Transportstellung definiert, bei der sich der Roboterarm so weit wie möglich zusammenfaltet, um ein möglichst geringes Volumen während der Navigation der Roboterbasis zu bieten. Der Roboterarm hat eine Reichweite von 1,3 m und eine Eigenmasse von 29 kg, die sich im ausgestreckten Zustand sehr schnell auf das Fahrverhalten des Roboters auswirken. Mit diesen Einstellungen erzeugt *MoveIt* schließlich ein kinematisches Modell, das nun per `MoveGroup`-*Action* (Auflistung A.17) angesteuert werden kann. Abbildung 5.5 zeigt einen Screenshots aus dem *Setup Assistant* von *MoveIt*, auf dem die Transportstellung des *UR-10* zu sehen ist.

5.4 Lighthouse Tracking des RB-Kairos

Wie in Abschnitt 5.1 (Hardwareinstallationen) beschrieben, wurden zwei *Vive Tracker* am Chassis des *RB-Kairos* angebracht, um die Koordinaten des Roboters im Kontext der *Lighthouse*-Technologie zu ermitteln. Die Berechnung der Trackingdaten erfolgt durch das in Abschnitt 3.5 (Unity) erwähnte *OpenVr SDK*, das jedoch einige Probleme mit sich bringt. Die Software wurde für Computer entwickelt, die die Hardwarevorraussetzungen zum Betrieb eines *VR*-Systems besitzen. Zu ihnen zählt eine performante Grafikeinheit, die der interne Computer des *RB-Kairos* nicht besitzt. Es kann auch keine zusätzliche Grafikkarte nachgerüstet werden, da im Gehäuse des Roboters nicht genug Platz vorhanden ist. Erschwerend kommt hinzu, dass die benötigte Laufzeitumgebung *SteamVr* vorrangig für Windows entwickelt wird und der Computer als Betriebssystem Linux verwendet. Es liegt daher nahe, das Tracking des *RB-Kairos* auf einem kompatiblen Computer durchzuführen und die Koordinaten per Netzwerk zurück zum Roboter zu senden.

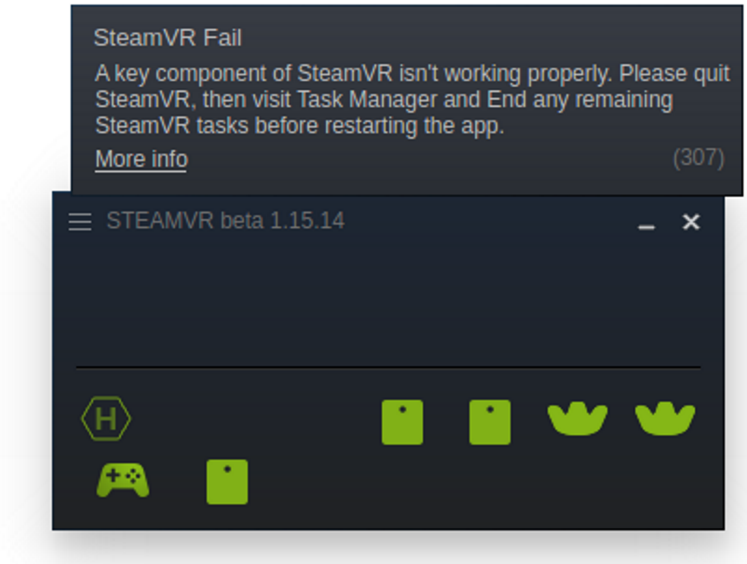


Abbildung 5.6: Ausschnitt aus dem UI von *SteamVr* mit erfassten *Trackern*, *Lighthouses*, *Controller* und Fehlermeldung.

Nachteil dieser Methode ist, dass der Roboter für das *Lighthouse*-Tracking auf einen externen Computer angewiesen ist. Diese Tatsache ist im Kontext des *VR*-Systems dieser Diplomarbeit nicht wichtig, da ohnehin ein Windows-PC zur Darstellung der *virtuellen Realität* vorhanden ist. Sollte das Konzept jedoch zu einem späteren Zeitpunkt geändert werden und der Rechner nicht mehr zur Verfügung stehen, so können auch die Koordinaten des Roboters nicht mehr ermittelt werden. Es wird daher versucht, die Berechnung der Roboterkoordinaten am *RB-Kairos* durchzuführen.

Um das Tracking unabhängig von anderen Computern zu betreiben kann die Software mit einer speziellen Konfiguration am PC des *RB-Kairos* installiert werden. Die performante grafische Rechenleistung ist in *SteamVr* nur zur Darstellung von 3D-Grafik in einem *Headset* notwendig. Am *RB-Kairos* wird sie aber nicht benötigt, da auch kein *Headset* auf ihm betrieben wird und kann, durch Entfernen beziehungsweise durch Verzicht auf die Installation des Grafiktreibers, unterbunden werden. Abschnitt A.1 (Einrichtung von SteamVr auf dem Roboter) enthält eine genaue Anleitung zur Installation von *Steam*, *SteamVr* und *OpenVr* und ihre Konfiguration auf dem Rechner des Roboters.

Nach erfolgter Installation werden *Steam* und *SteamVr* gestartet. Dabei meldet *Steam*, dass eine Komponente nicht richtig funktioniert (`A key component of SteamVR has failed`) (siehe Abbildung 5.6). Der Fehler ist bekannt und auch gewollt. Bei der fehlgeschlagenenen Komponente handelt es sich um das Grafikfenster der *VR*-Szene, das normalerweise nach dem Start von *SteamVr* am *HMD* angezeigt wird. Da jedoch kein *Headset* angeschlossen ist und der benötigte Grafiktreiber nicht installiert wurde, bricht das System die Ausführung der Komponente ab. Die Funktionsweise des Trackings

bleibt von diesem Fehler unberührt. Die nicht erfolgende Grafikdarstellung spart zudem Rechenleistung am Roboter.

Obwohl keine 3D-Grafik ausgegeben wird und die Anzeige von *SteamVr* nur einem Statusbericht dient, ist der Anschluss eines Displays essenziell. *SteamVr* ist so konzipiert, dass seine Ausführung verhindert oder beendet wird, wenn kein Monitor erkannt wird. Das passiert auch beim Abziehen des HDMI-Kabels und beim Abschalten des Displays. Unter anderem aus diesem Grund führt der *RB-Kairos* sein eigenes Touchdisplay mit. Auf die Monitorproblematik wird in Abschnitt 7.1 (Verbesserungspotential) näher eingegangen.

Um das Orten der *Tracker* zu ermöglichen, müssen diese mit dem System gekoppelt werden. Nach dem Start des Gerätesuchlaufs in *SteamVr* und einem langen Druck auf den Einschaltknopf jedes *Trackers* scheinen diese nun in der Übersicht auf. Sobald die *Tracker* Sichtkontakt zu den *Lighthouse*-Basisstationen haben, können ihre Koordinaten erfasst werden.

Beim Auslesen der *Tracker*-Koordinaten kommt das Basis-*Framework* von *SteamVr* *OpenVr* zum Einsatz. Die für diesen Zweck erstellte *ROS-Node* wurde in *Python* geschrieben und bedient sich der Softwarebibliothek *pyopenvr* [pyo], um die Daten von *OpenVr* in *Python* zugänglich zu machen. Die erhaltenen *Tracker*-Koordinaten entsprechen dem in Abschnitt 3.6 (Koordinatensysteme) beschriebenen Schema von *OpenVr* und liegen in Matrixnotation vor. Anhand der in Auflistung A.19 angegebenen Umrechnungsfunktion können die Koordinaten zur Verwendung als `PoseStamped` *Messages* (Auflistung A.11) umgewandelt werden. Die Nachrichten werden mit einem Zeitstempel versehen und anhand der Seriennummer des *Trackers* in einem eigenen *Topic* veröffentlicht. Die Koordinatenermittlung findet jedoch nur statt, wenn der *Tracker* auch Sichtkontakt zu den Basisstationen hat.

Die *Tracker* befinden sich an definierten Positionen am Gehäuse des *RB-Kairos* mit einer Entfernung von 0,55 m entlang der X-Achse und 0,4 m entlang der Y-Achse zueinander. Der Winkel zwischen ihrer Verbindungslinie und der Vorwärtsrichtung des Roboters entspricht demnach ungefähr 36,03°. Die Ausrichtung der *Tracker* auf der Gehäuseoberseite ist dabei egal, da aus Genauigkeitsgründen nur die Verbindungslinie zur Winkelbestimmung verwendet wird. Der gemeinsamer Schwerpunkt der beiden *Tracker* stimmt mit dem horizontalen Mittelpunkt des Roboters überein. Zusammen mit der bekannten Montagehöhe von etwa 0,52 m über dem Fußboden lässt sich so der *Base-Footprint* aus der Sicht der *Vive* berechnen, der in weiterer Folge `rbkairos_origin` genannt wird.

Sobald der `rbkairos_origin` aus den Koordinaten beider *Tracker* einmalig ermittelt wird, werden die Transformationen von beiden *Trackern* zum `rbkairos_origin` abgespeichert. Mit diesen kann nun von jedem *Tracker* einzeln der `rbkairos_origin` berechnet werden. Das bietet den Vorteil, dass die Ortung des Roboters bei Ausfall oder Verdeckung eines einzelnen *Trackers* aufrechterhalten werden kann. Zur Feststellung von Trackingausfällen werden die Zeitstempel der `PoseStamped`-Nachrichten auf ihre Aktualität überprüft und im Notfall auf eine Einzelberechnung der Roboterkoordinaten umgeschaltet. Der ermittelte `rbkairos_origin` und die Koordinaten der drei *Lighthou-*

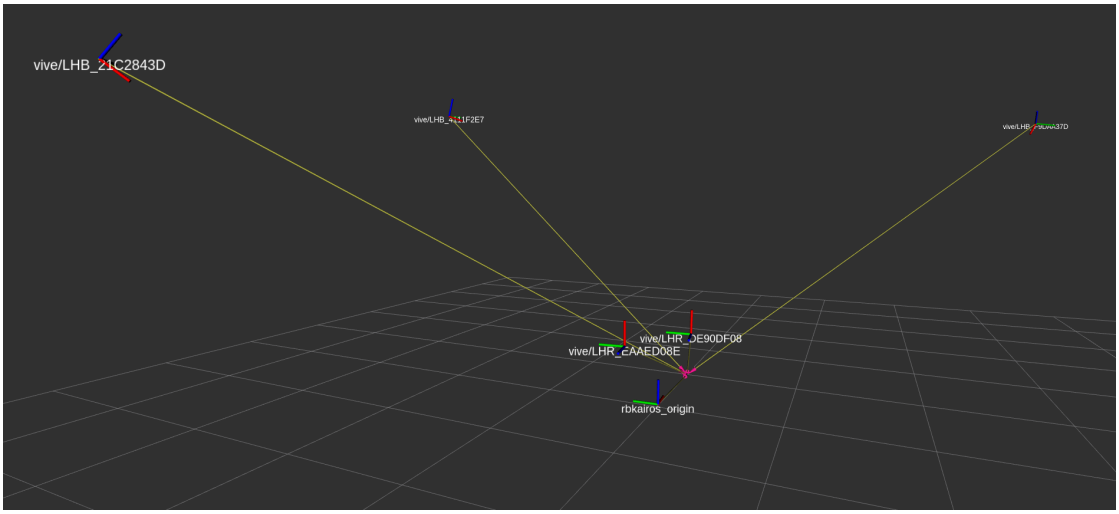


Abbildung 5.7: Die Positionen der *Vive-Tracker* und *Lighthouses* im Visualisierungstool *RVIZ*. Im oberen Bildbereich sind die drei *Lighthouses* zu erkennen. Bei den unteren drei Markern handelt es sich um die zwei erfassten *Tracker* und dem daraus errechneten `rbkairos_origin`.

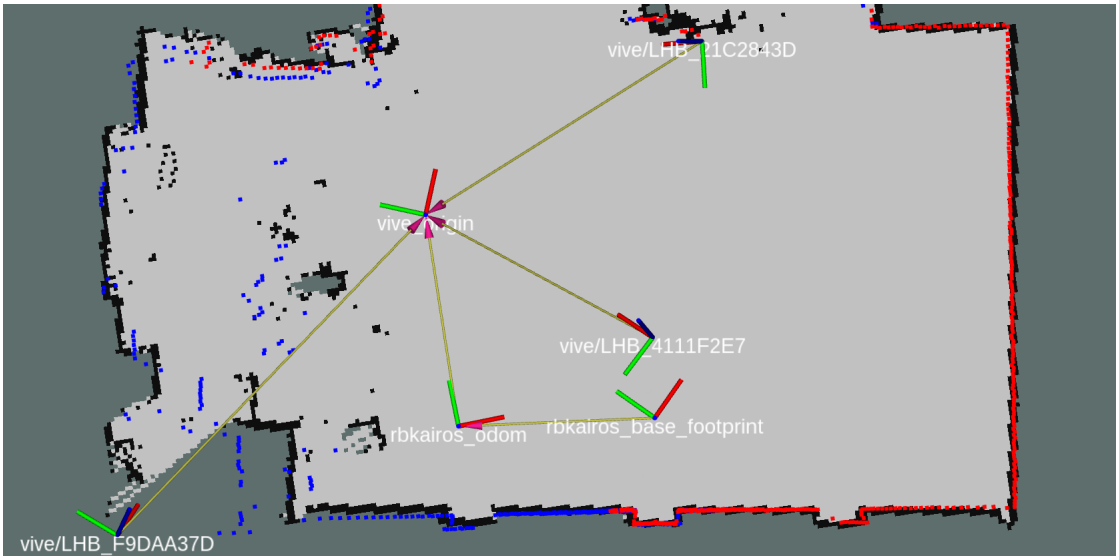
ses werden anschließend als Kinderknoten des *Frames* `vive_origin` im *Transform Tree* veröffentlicht (siehe Abbildung 5.7).

Als Elternframe von `vive_origin` wird der *Root-Frame* des Weltkoordinatensystems `rbkairos_map` gesetzt. Der `vive_origin` ist daher ein Geschwisterknoten des *Odometrie-Frames* `rbkairos_odom`. Aktuell befindet sich `vive_origin` genau am Koordinatenursprung von `rbkairos_map` (siehe Abbildung 5.8a). Wie in Abschnitt 4.2 (Entwurf des VR-Systems) erwähnt, handelt es sich aber um zwei *Frames*, die ihren Ursprung selbst gewählt haben. Der `vive_origin` muss daher noch entsprechend ausgerichtet beziehungsweise kalibriert werden, um seine, der Realität entsprechende, Position einzunehmen.

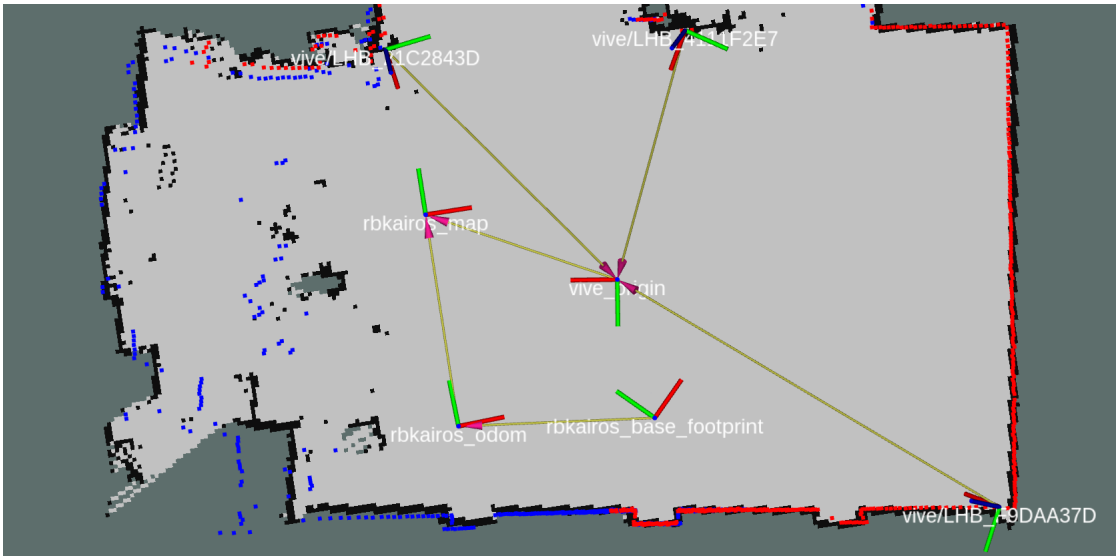
Ausgangspunkt der Kalibration sind die aktuellen Koordinaten des *RB-Kairos*. Diese werden im *Transform Tree* vom *Navigation Stack* als *Frame* `rbkairos_base_footprint` und von der *Vive* als *Frame* `rbkairos_origin` geliefert. Die Koordinaten von `vive_origin` müssen nun so angepasst werden, dass der *Frame* `rbkairos_origin` genau über jenem von `rbkairos_base_footprint` liegt. Die zu ermittelnden Koordinaten von `vive_origin` lassen sich durch Anwendung der Formel

$$\overrightarrow{vive\_origin} \times \overrightarrow{rbkairos\_origin} = \overrightarrow{rbkairos\_odom} \times \overrightarrow{rbkairos\_base\_footprint}$$

berechnen, wobei die angegebenen Vektoren jeweils den *Transform*-Koordinaten (Auflistung A.4) zwischen Elternframe und aktuellem *Frame* entsprechen. Abbildung 5.8b zeigt



(a) Vor der Kalibration: `vive_origin` befindet sich am Ursprung von `rbkairos_map` und die drei *Lighthouse-Frames* (`vive/LHB_...`) an falschen Raumkoordinaten.



(b) Nach der Kalibration: `vive_origin` wurde vom Ursprung der Karte weggeschoben und dadurch die *Lighthouse-Frames* richtig positioniert.

Abbildung 5.8: Auschnitte aus *RVIZ* vor und nach der Kalibration des `vive_origin`-*Frames*.

die Koordinaten nach erfolgter Kalibration. Dieser Schritt ist nur einmal notwendig, da davon ausgegangen werden kann, dass sich die Positionen der *Lighthouses* und die Karte des Raums nicht ändern werden.

Mit diesem Kalibrationsschritt ist es nun möglich, dem Roboter Bewegungskommandos, sowohl im Weltkoordinatenframe als auch im Koordinatenframe der *Vive*, zu erteilen. *ROS* kümmert sich automatisch um die Umwandlung der Koordinaten, solange der *Transform Tree* ihren Bezug herstellen kann.

Zur Ausführung einer Navigationsaufgabe innerhalb einer *ROS-Node* wird ein *ActionClient* verwendet. Er ist das Gegenstück zum *ActionServer*, den zum Beispiel die `move_base` Komponente (Auflistung A.16) implementiert, und für die Kommunikation zwischen *ActionServer* und *Nodes* verantwortlich ist. Die `target_pose` der `move_base` ist vom Typ `PoseStamped` (Auflistung A.11) und enthält neben den Zielkoordinaten auch die Angabe des *Frames*, in dem sich die Koordinaten befinden.

Um die Navigationsfähigkeiten des Roboters zu testen, wird *RVIZ* am Fernwartungsnotebook gestartet. Es enthält ein *Plugin*, mit dem durch Klick auf die Übersichtskarte eine Navigation zu eben jenem Punkt angestoßen werden kann. Der *RB-Kairos* wird daraufhin einen Navigationspfad berechnen, seine Motoren in Bewegung setzen und auf das Ziel zusteuern. Durch den Navigationsalgorithmus *SLAM* erkennt er dabei auch spontan auftretende Hindernisse entlang seines Pfads, wird automatisch langsamer und versucht, diese anhand seiner *Lidar*-Daten zu umfahren.

Der *RB-Kairos* ist nun in der Lage, an einen beliebigen Punkt im Koordinatenframe der *Vive* autonom zu navigieren. Der nächste Schritt ist die Einbindung der *Manipulator*-Steuerung in dieses Bewegungskonzept.

5.5 Integration des UR-10

In Abschnitt 5.3 (Konfiguration von UR-10 und MoveIt) wurde ein *Kinematik*-Modell des Roboterarms *UR-10* erstellt und eine *MoveGroup* erzeugt. Diese kann nun in *ROS* per Software gesteuert werden. Die erstellte *Node* mit dem Namen `HapticVr` kennt zwei Positionen des Arms, an denen sich dieser in einem bewegungslosen Zustand befindet. Die erste ist die in der Konfigurationsphase erzeugte Transportstellung (siehe Abbildung 5.5), die der Roboterarm während der Bewegung des Chassis einnimmt. Bei der zweiten Position handelt es sich um eine ausgefahrene Stellung des Roboterarms an jene Position, die der Roboter als Endposition zur Platzierung der *Haptikwand* erhalten hat. Die zweite Position wird aufgrund der Koordinaten der Roboterbasis dynamisch berechnet und dient als Input der `HapticVr`-*Node*.

Die erstellte *Node* beinhaltet einen *Subscriber*, der *Messages* vom Typ `PoseStamped` (Auflistung A.11) vom *Topic* `rbkairos/hapticvr/goal` empfängt. Die Nachrichten enthalten Position, Orientierung sowie die Angabe des Bezug-*Frames* und wird in weiterer Folge Zielpunkt genannt. Sofern der *RB-Kairos* bereit ist, bewirkt ein Empfang dieser Nachricht die Ausführung der weiteren Schritte. Zunächst wird der *UR-10* angewiesen,

die in Abschnitt 5.3 (Konfiguration von UR-10 und MoveIt) definierte Transportstellung einzunehmen.

MoveIt stellt hierzu ein Softwareobjekt namens `MoveGroupCommander` [mova] zur Verfügung, das für die Abwicklung der Aufgaben einer *MoveGroup* zuständig ist. Der Commander erlaubt unter anderem die Angabe der Zielstellung des *Endeffektors* durch Koordinaten sowie die Gelenksstellungen anhand der im Setupschritt festgelegten statischen Posen. Die Transportstellung kann somit namentlich als Zielstellung der Gelenke angegeben werden. Nach Start des `MoveGroupCommanders` erfolgt die automatische Berechnung und die Ausführung des Bewegungspfads.

Nachdem der *UR-10* die Einnahme der Transportstellung erfolgreich beendet hat, wird der *RB-Kairos* angewiesen in die ungefähre Nähe des Zielpunkts zu navigieren. Die Funktionsweise der Navigation wurde bereits in Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos) erklärt. Der Zielort der Navigation wird 0,7 m hinter dem gewünschten Zielpunkt der *Haptikwand* definiert. Dieser Abstand erlaubt dem Roboterarm genügend Bewegungsfreiheit zwischen Chassis und Zielpunkt und ist ein Erfahrungswert, der sich während der Arbeit mit dem Roboter bewährt hat.

Während der Navigation wird laufend die Position des *RB-Kairos* ermittelt. Der Roboter navigiert anhand der Koordinaten, die der *Navigation Stack* berechnet, und veröffentlicht diese im *Frame* des *Base-Footprints* `rbkairos_base_footprint`. Die Koordinaten, die anhand der *Lighthouse*-Technologie ermittelt werden, werden im *Frame* `rbkairos_origin` publiziert. Da es der Trackinglösung des *RB-Kairos* systembedingt an Genauigkeit mangelt, kann es dabei zu einer Differenz von mehreren Zentimetern zu den ermittelten Koordinaten der *Vive* kommen. Zu diesem Zeitpunkt wird daher angenommen, dass sich der *Base-Footprint* des Roboters richtigerweise an den Koordinaten von `rbkairos_origin` aufhält und diese Stelle auch für die Endpositionierung des Arms verwendet wird.

Um dem `MoveGroupCommander` eine `Pose` (Auflistung A.5) zur Positionierung des *Endeffektors* zu übermitteln, muss diese relativ zum *Base-Footprint* angegeben werden. Das bedeutet, dass die Koordinaten des Zielpunkts vom *Topic* `rbkairos/hapticvr/goal` aus der Sicht von `rbkairos_origin` berechnet werden müssen. Der *Transform Tree* stellt diese Funktionalität für alle in ihm erfassten *Frames* zur Verfügung. Der Zielpunkt ist jedoch kein *Frame*, daher wird zunächst die Transformation zum Bezugsframe von `rbkairos/hapticvr/goal`, der im `Header` (Auflistung A.8) der Nachricht enthalten ist, angefordert. Durch anschließende Matrixmultiplikation mit den Koordinaten der `PoseStamped` Nachricht kann so die verbleibende Transformation zum Zielpunkt berechnet werden.

Der `MoveGroupCommander` der *MoveGroup* sorgt erneut für die Berechnung eines Pfads und die Ausführung der Bewegung. Nach Beendigung all dieser Schritte befindet sich an der geforderten Zielposition die *Haptikwand* des Roboters, die bereit ist, als haptisches Element in der *VR*-Simulation zu dienen. Sollte es jedoch zu Fehlern kommen, zum Beispiel durch Kollision des Chassis oder des Roboterarms, oder wenn kein passender Pfad

zum Zielort gefunden werden kann, so wird die Ausführung abgebrochen. Nach Behebung eines etwaigen Notstopps kann durch erneute Veröffentlichung einer Position unter dem *Topic* `rbkairos/hapticvr/goal` die Positionierung der *Haptikwand* wieder gestartet werden. Ausgangspunkt dieser Nachricht ist im fertigen System die *VR*-Anwendung am Windows-PC.

5.6 VR-Anwendung

Die *VR*-Anwendung ist das Herzstück des visuellen Systems und Ausgangspunkt der Navigationsbefehle an den Roboter. Ihr Betrieb erfolgt auf einem Windows-PC, der in Abschnitt 3.5 (Unity) beschrieben wird. Um das *Headset* der *Vive* per Wireless-Kit zu betreiben, ist dessen Montage am *HMD* notwendig. Dazu wird das HDMI-Kabel des *Headsets* mit jenem des Wireless-Kits ausgetauscht. Die Stromversorgung erfolgt ab diesem Zeitpunkt mit einem eigenen Akkupack. Außerdem wird die Sendeeinheit des Kits an einem erhöhten Punkt im Labor angebracht und per Kabel mit dem Windows-Rechner verbunden. Der Betrieb des Wireless-Kits erfordert die Installation von Treibern und den Start eines eigenen Wireless-Dienstprogramms.

Am Computer ist die Installation von *Unity*, *Steam* und *SteamVr* erforderlich. Die Installation der beiden *Steam* Komponenten erfolgt, anders als am Roboter, ohne Fehlermeldung, siehe Abschnitt A.1 (Einrichtung von SteamVr auf dem Roboter), da der Windows-PC über die nötigen Hardwarevorraussetzungen erfüllt. Beim Start von *Unity* wird die Erzeugung eines neuen 3D-Projekts ausgewählt und der Editor gestartet. Dieser wird in Abbildung 5.9 angezeigt. In der Mitte befindet sich die 3D-Ansicht der Szene, auf der linken Seite der hierarchische Aufbau der Szene, unten die *Assets*, also die verwendbaren Komponenten innerhalb des Projekts, und rechts das Eigenschaftsfenster der selektierten Komponente.

Um die 3D-Anwendung *VR*-tauglich zu machen, wird das *SteamVr-Plugin* [steb] aus dem *Unity Asset Store* [unib] importiert. Das *Plugin* enthält eine Fülle an Beispielkomponenten, die in einer Demonstrationsszene präsentiert werden, wovon der Großteil aber im Rahmen dieser Diplomarbeit nicht relevant ist. Wichtige Komponenten sind das 3D-Modell der *Vive Controller* und das *CameraRig* der *VR*-Szene. Das *CameraRig* enthält eine virtuelle Kamera und die Software-Repräsentationen der beiden *Controller*, und wird in einer leeren Szene platziert. Um den Trackingbereich des Labors mit einer Fläche von $8 \times 6,3\,\mathrm{m}$ abzubilden, wird ein Quader mit diesen Ausmaßen und einer Höhe von 0,1 m auf dem Boden erstellt. In einem Testlauf dieser Szene kann sich ein Benutzer oder eine Benutzerin durch Tragen des *Headsets* frei im eingezeichneten Trackingbereich bewegen, während ihm oder ihr die virtuelle Szene im *HMD* präsentiert wird. Die *Controller* der *Vive* werden ebenfalls automatisch erkannt und durch ein 3D-Modell von ihnen dargestellt.

Zurück im Editor erfolgt die Platzierung von virtuellen Objekten, die vom *RB-Kairos* mit einer Haptik versehen werden können. Es gibt zwei Typen dieser Objekte: Die erste ist eine Fläche, die an einer beliebigen Stellen markiert werden kann, um dort die Haptik bereitzustellen. Der zweite Typ ist ein Würfel, der die gleiche Seitenlänge

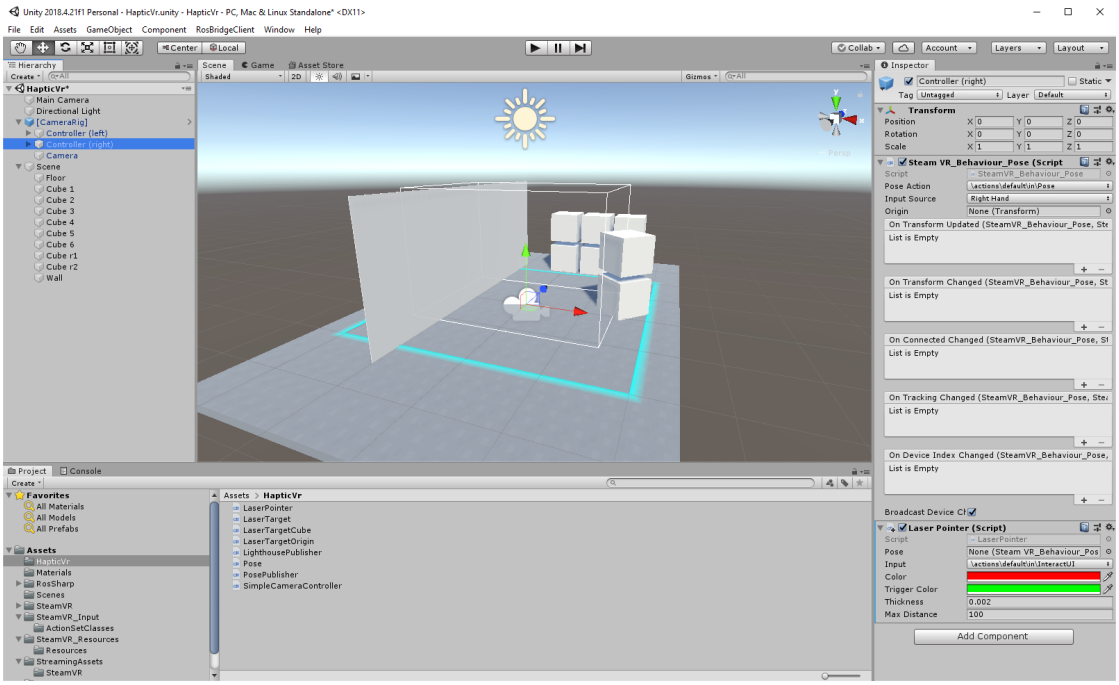


Abbildung 5.9: Screenshot aus dem Editor von *Unity*.

wie die *Haptikwand* besitzt und bei dem nach Aktivierung eine der sechs Seiten zur Gänze tastbar werden soll. Die Aktivierung eines Objekts erfolgt durch Betätigung des *Controller-Triggers*. Insgesamt werden eine Fläche und acht Würfel in der Szene verteilt (siehe Abbildung 5.9).

Um ein Haptikobjekt beziehungsweise eine Stelle darauf auszuwählen, wird ein virtueller Laserpointer am 3D-Modell eines *Vive-Controllers* angebracht. Abbildung 5.9 zeigt rechts unten die Anbindung des Laserpointer-Skripts an den rechten *Controller*. Bei jedem Updateereignis des *Controllers* wird eine Gerade ausgehend von seiner Spitze in den Raum projiziert und auf Intersektion mit der Kollisionsgeometrie der Szene überprüft. Bei dieser Funktion handelt es sich um *Raycasting*, das von der Physikkomponente von *Unity* zur Verfügung gestellt wird. Wenn ein Schnittpunkt von Linie und Geometrie gefunden wird, liefert die Funktion das getroffene Objekt und die genauen Details zurück. Handelt es sich bei dem Objekt um ein Haptikobjekt und wird zusätzlich der *Trigger* des *Controllers* betätigt, so werden die Aktivierungsfunktion des Haptikobjekts aufgerufen und die Koordinaten der *Haptikwand* festgelegt. Zusätzlich wird für die Darstellung des Laserpointers ein grüner, schmaler Quader mit einer Länge, die dem Abstand von *Controller* und Kollisionsobjekt entspricht, angezeigt. Auflistung A.18 zeigt die volle Implementation der Updatefunktion des Laserpointers.

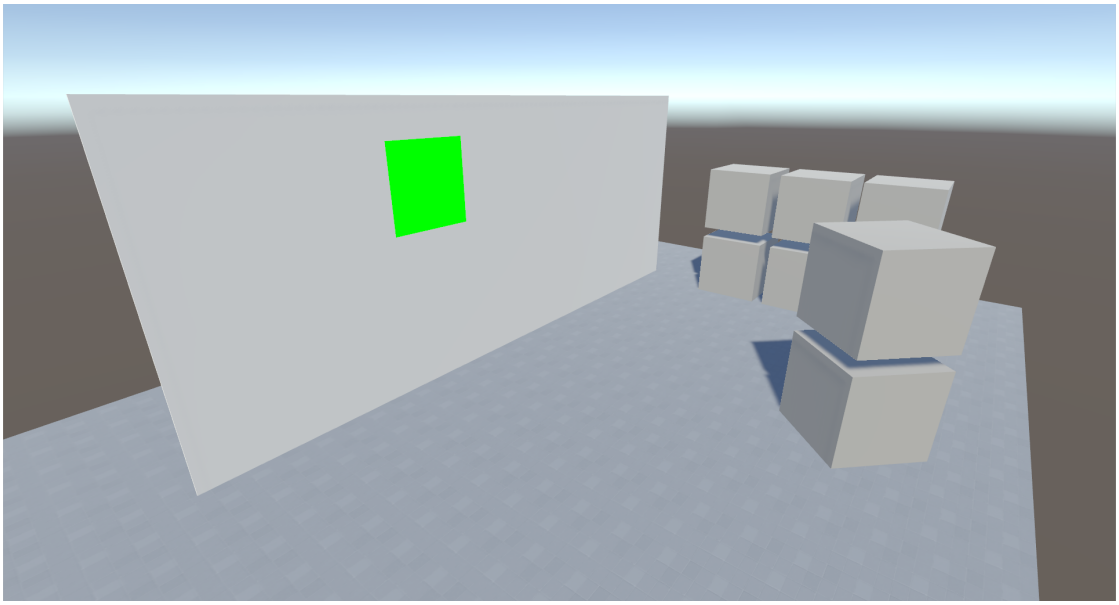
Bei der Aktivierung der Fläche entspricht der Schnittpunkt mit dem Laserpointer gleichzeitig den Koordinaten der *Haptikwand*. Wird jedoch ein Würfel getroffen, so soll die

Haptikwand am Mittelpunkt der getroffenen Seite platziert werden. Dazu wird der Schnittpunkt in den *Frame* des Würfels transformiert und die Koordinaten des Punkts analysiert. Aufgrund des geometrischen Aufbaus eines Würfels entspricht eine der drei Koordinaten genau der halben Seitenlänge. Diese Koordinate hat absolut gemessen auch immer einen größeren Wert als die zwei anderen. Ausnahmen bestehen wenn Ecken oder Kanten getroffen werden. Diese können aber vernachlässigt werden, da sie ebenso Teil einer Seite sind. Der Mittelpunkt der Seite entspricht einem Punkt mit dem höchsten gefundenen Koordinatenwert und 0 an seinen übrigen Stellen. Wird der Punkt zurück in den *Frame* der Szene transformiert, erhält man so die Koordinaten der *Haptikwand* auf dem Würfel. Abbildung 5.10 zeigt beide Objekttypen nach ihrer Aktivierung.

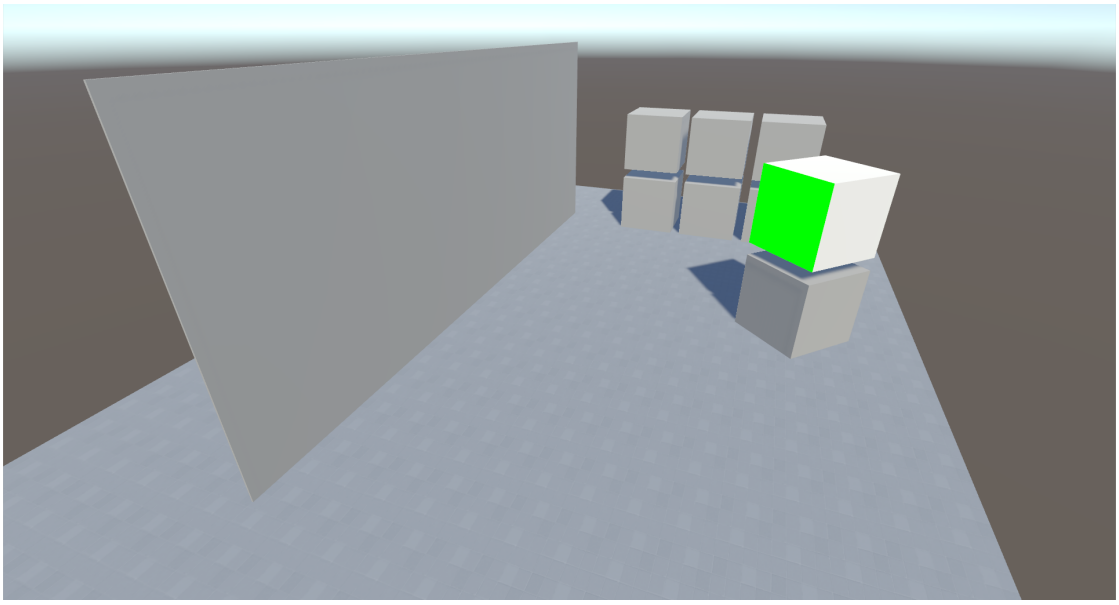
Der ermittelte Zielpunkt der *Haptikwand* muss nun an *ROS* gesendet werden. Da der Windows-Computer nicht mit *ROS* kompatibel ist, findet die Kommunikation über die *rosbridge* des Roboters statt. Dazu wird das *ROS-Sharp Plugin* [rosb] aus dem *Unity Asset Store* [unib] importiert. Es enthält vorgefertigte *Publisher* und *Subscriber*, die mit einer *rosbridge* kommunizieren können. Diese Komponenten werden typischerweise als Skript an *Unity*-Objekte gehängt, um deren Koordinatenveränderung automatisch im jeweils anderen System verfügbar zu machen. Im Fall der *Haptikwand* soll ihre Position nicht laufend übertragen werden, sondern nur einmalig nach Aktivierung des haptischen Objekts. Es wird daher ein *Publisher* erstellt, der diese Koordinaten anhand von Tabelle 3.3 in das Koordinatensystem von *ROS* transformiert und eine einzelne `PoseStamped`-*Message* (Auflistung A.11) an die *rosbridge* sendet.

Die *Unity*-Koordinaten werden in ihrem eigenen *Frame* `unity_origin` veröffentlicht. Obwohl es sich um denselben Trackingbereich und dieselben drei *Lighthouses* handelt, die auch die Trackingkoordinaten des Roboters im *Frame* `vive_origin` liefern, kommt es vor, dass die ermittelten Koordinaten beider Systeme nicht zueinander passen, sich also die Ursprünge ihrer *Frames* unterscheiden. Die zwei *Frames* müssen daher, genauso wie die zwei *Base-Footprints* in Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos), zueinander ausgerichtet werden. Zu diesem Zweck werden die Koordinaten der drei *Lighthouses*, in diesem Fall aus der Sicht des Windows-PC, in *ROS*-Koordinaten transformiert und an die *rosbridge* gesendet, um sie in einem weiteren Kalibrationsschritt zu überlagern.

Die Kalibration erfolgt am *RB-Kairos* oder dem Fernwartungsnotebook, da eine neue *Node* gestartet werden muss. Darin werden die Koordinaten eines einzelnen *Lighthouses* in den *Frames* `unity_origin` und `vive_origin` verglichen und die Koordinaten von `unity_origin` so verändert, damit sich die beiden Koordinaten des *Lighthouses* überlagern. Abbildung 5.11 zeigt, wie die *Frames* des *VR*-Systems hierarchisch in Bezug zueinander stehen. Die Pfeile zeigen jeweils von den Eltern-*Frames* zu ihren Kinder-*Frames*, wobei ein strichlierter Pfeil eine dynamische und ein solider Pfeil eine statische Transformation darstellt. Mit den vollständigen Beziehungen zwischen den *Frames* ist es dem *Transform Tree* nun möglich, der `HapticVr`-*Node* aus Abschnitt 5.5 (Integration des UR-10) die korrekten Koordinaten der *Haptikwand* zu liefern.



(a) Eine Markierung an der großen Fläche erlaubt eine willkürliche Platzierung der *Haptikwand*.



(b) Die ermittelte Seite des Würfels wird zur Gänze als *Haptikwand* markiert.

Abbildung 5.10: Die erstellte *VR*-Szene in *Unity*. Die Fläche links sowie die Würfel rechts können „aktiviert“ werden, um eine Haptiksimulation dieser Oberflächen zu veranlassen.

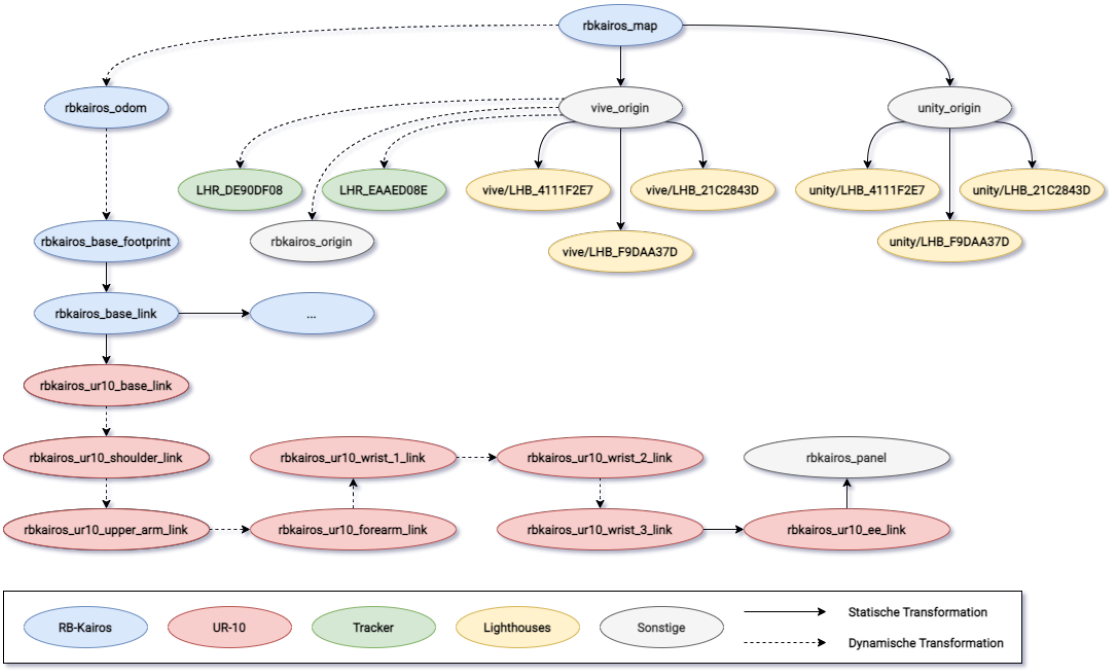


Abbildung 5.11: Übersicht der *Frames* des *VR*-Systems.

5.7 Inbetriebnahme

Um das erstellte *VR*-System erfolgreich in Betrieb zu nehmen, müssen ihre Einzelkomponenten in der richtigen Reihenfolge gestartet werden. *ROS* bietet durch seine *Launch*-Dateien die Möglichkeit, eine Kette an verschiedenen *Nodes* automatisiert auszuführen, jedoch können einige Module, wie zum Beispiel *SteamVr*, damit nicht verwaltet werden, da ihr Start beispielsweise eine Benutzereingabe fordert. Daher wird das System anhand der folgenden Liste per Hand hochgefahren.

1. Inbetriebnahme der *Vive* Pro mit Wireless-Kit und den *Lighthouse*-Basisstationen. Start des Wireless-Dienstprogramms am Windows PC.

2. Falls sich die Positionen der *Lighthouses* geändert haben, Ausführen des *Room-Setups* von *SteamVr* am Windows-Rechner und Übertragung der neuen Konfigurationsdateien auf den *RB-Kairos* (siehe Unterabschnitt A.1.4).

3. Hochfahren des *RB-Kairos* inklusive *UR-10*: Die genauen Schritte werden in Abschnitt A.2 angeführt.

4. Start von *SteamVr* am *RB-Kairos* über das Touch-Display und Einschalten der zwei *Tracker* per Knopfdruck.

5. Nachdem beide *Tracker* in *SteamVr* aufscheinen, Start der *Tracking-Nodes*. Diese publizieren die *Tracker*-Koordinaten in *ROS* und berechnen daraus die Koordinaten des *RB-Kairos* im *Frame* der *Vive*.

6. Ausführen der *Vive*-Kalibrations-*Node*. Diese stellt den Bezug zwischen den *Frames* von *ROS* und *Vive* her, um ein Umrechnen zwischen ihnen zu ermöglichen. Diese *Node* kann auch auf dem Fernwartungsnotebook ausgeführt und nach erfolgter Kalibration wieder beendet werden.

7. Start von *Unity* und der *VR*-Anwendung am Windows-Rechner. Die Anwendung verbindet sich mit der *rosbridge* am Roboter.

8. Ausführen der *Unity*-Kalibrations-*Node*. Durch sie wird der Bezug zwischen den *Frames* von *Vive* und *Unity* hergestellt, um eine Koordinatentransformation zu ermöglichen. Diese *Node* kann ebenso auf dem Fernwartungsnotebook gestartet und im Anschluss wieder beendet werden.

9. Start des *MoveGroup-ActionServers* am *RB-Kairos*. Dieser ermöglicht *ROS* den Zugriff auf den *UR-10*.

10. Start der *HapticVr-Node*. Diese Software abonniert das *Haptikwand-Topic* der *Unity*-Anwendung und startet nach Erhalt einer Nachricht die Routinen zur Navigation des Fahrwerks und die Bewegungsausführung des *UR-10*.

KAPITEL 6

Evaluierung

Dieses Kapitel behandelt die Evaluierung des im Zuge dieser Diplomarbeit erstellten *VR*-Systems. Abschnitt 6.1 (Technische Evaluierung) beleuchtet die technischen Aspekte des implementierten Systems und beantwortet Fragen zu ihren Betriebsparametern. Abschnitt 6.2 (Benutzererfahrung) beschreibt anschließend den durchgeführten Pilottest, der die zu erwartende Benutzererfahrung des Systems zeigt.

6.1 Technische Evaluierung

Dieser Abschnitt beschreibt die technische Evaluierung des in Kapitel 5 (Umsetzung) implementierten *VR*-Systems. Im Zuge dessen werden folgende Eigenschaften in der Ausführung des aktuellen Versuchaufbaus untersucht:

1. Wie groß ist das Trackingvolumen des *VR*-Systems?
2. Wie groß ist das Interaktionsvolumen des Roboters?
3. Welche Orientierung kann die *Haptikwand* einnehmen?
4. Wie lange benötigt das System, um die *Haptikwand* an einer beliebigen Stelle zu positionieren?
5. Welche Genauigkeit bietet das System?

6.1.1 Geometrie

Das Trackingvolumen wird vom Hersteller der *Vive* Pro bei Verwendung von vier *Lighthouses* mit einem Ausmaß von $10 \times 10\,\mathrm{m}$ angegeben. Der Trackingbereich des Versuchslabors hat eine Größe von circa $8 \times 6,3\,\mathrm{m}$, in dem drei *Lighthouse*-Basisstationen in einer Höhe

von circa 2,4 m angebracht sind. Der Sichtkontakt der am Roboter angebrachten *Tracker* zu den *Lighthouses* wurde durch rasterartiges Abfahren des Trackingbereichs in mehrere Richtungen getestet. Dabei konnte festgestellt werden, dass aufgrund ungünstiger Positionierungen der *Haptikwand* der Sichtkontakt eines einzelnen *Trackers* ausfallen kann. Da sich der andere *Tracker* aber auf der diagonal gegenüberliegenden Seite des Roboters befindet und die *Lighthouses* im Raum verteilt sind, ist vom Ausfall jeweils nur einer der beiden gleichzeitig betroffen. Die Tracking-*Node* des *VR*-Systems wurde dementsprechend konzipiert, um das Tracking auch mit nur einem *Tracker* aufrechtzuerhalten, siehe Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos).

Das Interaktionsvolumen des *RB-Kairos* entspricht dem Arbeitsbereich des *UR-10* entlang der ebenen Fläche, die der Roboter per Navigation erreichen kann. Der Trackingbereich des Labors, mit einer Fläche von circa $8 \times 6,3$ m, ist zum Großteil von Wänden und Computertischen umgeben, daher ist auch der Roboterarm bei der Platzierung der *Haptikwand* an diese Grenzen gebunden. Das Schulter-Knickgelenk des *UR-10*, das die Basis für die vertikale Armpositionierung bildet, befindet sich auf dem Roboterchassis in einer Höhe von 0,65 m. Mit seinem Arbeitsradius von 1,3 m und den vier weiteren Gelenken oberhalb des Knickgelenks ist es dem Roboterarm möglich den Fußboden zu erreichen. In einer vertikal nach oben gerichteten Stellung addieren sich der Radius und die Montagehöhe des Roboterarms zu einer maximalen Manipulationshöhe von 1,95 m. Im Versuchsaufbau dieser Arbeit ergibt das Interaktionsvolumen des *RB-Kairos* demnach ein Volumen von $8 \times 6,3 \times 1,95$ m, das vom Fußboden ausgeht.

Die *Haptikwand* kann grundsätzlich in jede Ausrichtung orientiert werden. Ausnahmen existieren jedoch bei Orientierungen die eine Kollision mit dem Roboter selbst zur Folge hätten. Sie werden von der Planungssoftware *MoveIt* rechtzeitig erkannt und die Ausführung verweigert. Um Abbrüche durch Selbstkollisionen zu minimieren, nimmt der *RB-Kairos*, und damit sein *Base-Footprint*, immer eine Position 0,7 m hinter dem Zielpunkt der *Haptikwand* ein. Dieser Erfahrungswert hat sich während der Arbeit mit dem Roboter bewährt. Der Montagepunkt des *UR-10*, der sich in der horizontalen Mitte des Roboters und damit vertikal über dem *Base-Footprint* befindet, ist 0,36 m zur Vorderkante des Roboters entfernt. Somit bleiben noch 0,34 m Spielraum vor dem Roboter, um dort die *Haptikwand* zu platzieren. Der Abstand wurde nah genug gewählt, damit der *UR-10* den Zielpunkt der *Haptikwand* erreichen kann, aber weit genug entfernt um nicht mit dem Roboterchassis zu kollidieren.

6.1.2 Zeitspanne

Das *VR*-System benötigt eine gewisse Zeit, um die haptische Erfahrung zur Verfügung zu stellen. Die Befehlskette beginnt in der *Unity*-Anwendung. Der Benutzer visiert ein virtuelles Objekt an und betätigt den *Trigger* des *Vive-Controllers*. Aufgrund von optimierten Algorithmen, die die Spieleengine zur Verfügung stellt, wird die Intersektion des virtuellen Laserstrahls mit der Umgebung augenblicklich berechnet. Die selektierte Oberfläche, beziehungsweise deren Mittelpunkt, wird nun per WiFi-Netzwerk zum *RB-Kairos* weitergeleitet. Auch diese Übertragung findet unverzüglich statt.

Um diese subjektive Annahme zu bestätigen, wurde ein Test durchgeführt, der die Roundtrip-Zeit zwischen der Betätigung des *Triggers* und dem Empfang des Zielpunkts am Roboter beziehungsweise dem Empfang der Empfangsbestätigung misst. Dazu wurde am *RB-Kairos* eine *Node* gestartet, die das Zielpunkt-*Topic* `rbkairos/hapticvr/goal` abonniert und anschließend sofort eine Ping-Nachricht veröffentlicht. In der *Unity*-Anwendung wurde ein zusätzlicher *Subscriber* erzeugt, der diesen Ping abonniert und die Zeitdifferenz zwischen *Trigger*-Betätigung und Empfang des Pings berechnet. Dieser Test wurde 100 mal durchgeführt, während sich der *RB-Kairos* an verschiedenen Stellen im Labor befand. Der Ping betrug durchschnittlich 21,97 ms, der Median 6,74 ms und die Standardabweichung 111,52 ms. 95 Messergebnisse wiesen einen Wert kleiner als 25 ms und zwei Messungen Werte über 300 ms auf. Die einzelnen sehr hohen Werte waren vermutlich auf kurze Aussetzer der Wi-Fi-Verbindung zurückzuführen. Ohne den zwei Extremwerten lag der Durchschnitt bei 8,15 ms und die Standardabweichung bei 7,00 ms. Die Daten zeigen, dass die Übermittlung der Zielposition mit sehr wenigen Ausnahmen sehr schnell erfolgte. Obwohl es sich bei diesem Durchschnittswert nicht um die reine Zeitspanne zwischen Befehlsaufgabe und Empfang am Roboter handelt, sondern zusätzlich noch um die Zeitdauer der Rückantwort, ist sie, aufgrund der im Anschluss gemessenen Navigationszeit des *RB-Kairos*, vernachlässigbar.

Zum Zeitpunkt, an dem ein Zielpunkt empfangen wird, befindet sich der *UR-10* des *RB-Kairos* in einem ausgestreckten Zustand, da zuvor die *Haptikwand* an einem anderen Punkt platziert wurde. Im besten Fall befindet sich die neu gewählte Position unmittelbar in der Nähe der alten. Im schlechtesten Fall am anderen Ende des Versuchsraumes. In idealen Konditionen würde es daher genügen den *Endeffektor* des *UR-10* an die neue Stelle zu dirigieren. Dazu wird die Position des *RB-Kairos* mittels *Vive*-Tracking bestimmt und der verbleibende Koordinatenvektor an *MoveIt* zur Ausführung weitergeleitet.

Die Pfadplanungssoftware berechnet aus dem vorliegenden Kinematikmodell einen Bewegungsplan zur Umpositionierung des Roboterarms. Diese Berechnung ist nicht trivial, da neben der Endposition des *Endeffektors*, die durch eine Vielzahl an Gelenksstellungen erfüllt werden kann, auch der gesamte Pfad von dieser Vielfalt betroffen ist. Was *MoveIt* daher macht, ist einen möglichst optimalen Pfad in einer vorgegebenen Zeit zu finden. Diese Zeit wurde im Rahmen der Implementierung auf 1 s festgelegt. Die Zeitdauer der Bewegungsausführung ist hingegen abhängig von der Entfernung zwischen Start und Endposition, beziehungsweise der Komplexität der Manipulation.

Die Manipulationskomplexität ergibt sich aus der Komplexität des Roboteraufbaus und den Freiheitsgraden der Gelenke. In vielen Fällen ist eine gleichzeitige Rotation aller Gelenke in ihre Zielstellung aufgrund von Eigenkollisionen nicht möglich. Deshalb müssen einige Gelenksrotationen zum Teil nacheinander stattfinden, was zu einer erhöhten Manipulationsdauer führt.

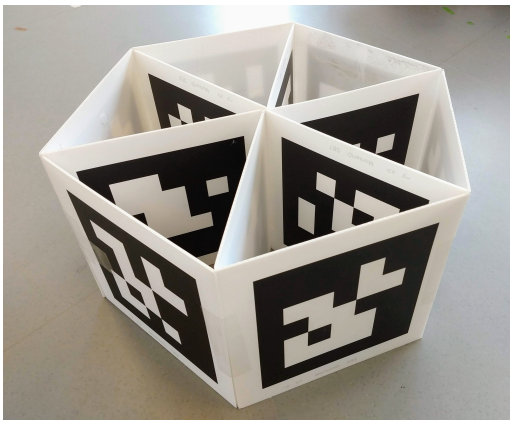
Die Neupositionierung der *Haptikwand* besteht immer aus einer Kombination von *Manipulator*- und Chassisbewegung. Befindet sich der gewünschte Zielpunkt der Wand nicht im Arbeitsbereich des *UR-10*, so muss eine Neupositionierung des gesamten Roboters vorgenommen werden, da der Arm den Punkt nicht alleine erreichen kann. Es ist aber

nicht ratsam, den Roboter im ausgestreckten Zustand des *Manipulators* in Fahrt zu setzen, da sich die Eigenmasse von *UR-10* und *Haptikwand* auf das Fahrverhalten des Chassis auswirkt. Aus diesem Grund wird der Roboterarm bei der Überbrückung eines Fahrtwegs zunächst immer in die Transportstellung gebracht (siehe Abbildung 5.5) und im Anschluss neupositioniert. Sollte sich der Zielpunkt innerhalb des *UR-10*-Arbeitsbereichs befinden, so ist dennoch nicht garantiert, dass der Roboterarm die gewünschte Stellung auch erreichen kann. Möglicherweise befindet sich der Zielpunkt an einer Stelle, die vom Roboter selbst eingenommen wird oder die Platzierung würde eine Kollision des *Manipulators* mit dem Chassis verursachen. Aus diesem Grund wird auch bei der Neupositionierung innerhalb des Arbeitsbereichs der gesamte Roboter mitbewegt.

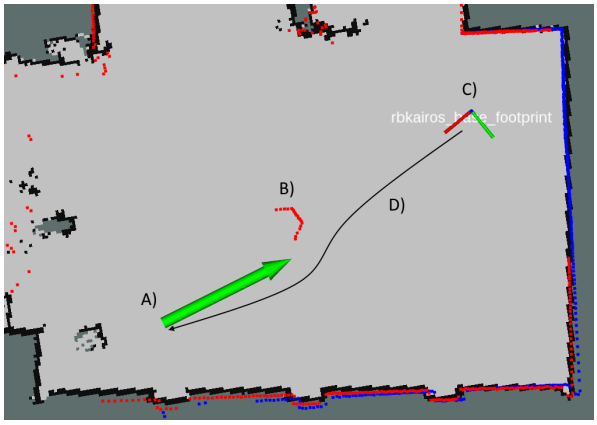
Die Zeitdauer von Ausstrecken und Einziehen des *UR-10* wird getrennt gemessen. Dazu wird etwa 0,7 m vor dem *RB-Kairos* ein zufälliger Zielpunkt ausgewählt und der `MoveGroupCommander` beauftragt, einen Bewegungspfad zu diesem Punkt zu berechnen und auszuführen. Der Abstand entspricht dem Erfahrungswert, der auch in der Implementation der Diplomarbeit bei der Festlegung des Navigationsziels hinter der *Haptikwand* zur Anwendung kommt. Die Messung wurde 20 mal durchgeführt. Inklusive der Berechnungszeit von 1 s dauert die Ausstreckbewegung durchschnittlich 6,16 s mit einer Standardabweichung von 0,63 s. Im Anschluss an jede Messung wird vom gewählten Punkt die Manipulation zurück in die Transportstellung durchgeführt. Die maximale Berechnungszeit wird ebenso in die Messung inkludiert und beträgt für die Einnahme der Transportstellung durchschnittlich 5,23 s mit einer Standardabweichung von 0,65 s. Hier zeigt sich ein Vorteil aus der Kombination von Navigationsziel und Transportstellung. Durch Festlegung des Navigationsziels hinter dem Zielpunkt, befindet sich die Stelle, an der die Wand platziert wird, immer unmittelbar vor dem Roboter. Die Transportstellung wurde außerdem so gewählt, dass ein Ausbringen der *Haptikwand* nach vorne nur eine Veränderung von wenigen Gelenken bedarf und so rasch durchgeführt werden kann.

Nach Erhalt des Zielpunkts und Einnahme der Transportstellung durch den *UR-10* beginnt die Pfadplanung der Navigation. Diese erfolgt, im Gegensatz zur Pfadplanung des *Manipulators*, nicht ausschließlich vor der Bewegung, sondern aufgrund des verwendeten *SLAM*-Algorithmus auch währenddessen. Nach Veröffentlichung des Navigationspunkts setzt sich der Roboter daher augenblicklich in Bewegung. Der Roboter ist aus Sicherheitsgründen auf eine Maximalgeschwindigkeit von 1 m/s während der autonomen Navigation konfiguriert. Für eine Fahrt zwischen den diagonalen Ecken des $8 \times 6,3$ m großen Trackingbereichs würde dieser theoretisch circa 10 s benötigen.

In der Realität beinhalten die Navigationsanforderungen an den *RB-Kairos* aber mehr als das Abfahren einer geraden Linie. Nicht eingerechnet in diese Kalkulation sind die Anfahrtszeiten zum Erreichen der Maximalgeschwindigkeit und etwaige Kurvenfahrten aufgrund von Hindernissen entlang des Navigationspfads. Bei der Ermittlung der Navigationszeit wird daher ein Hindernis im Raum platziert, um dem Roboter einen geraden Weg durch den Raum zu verwehren. Start und Endpunkt der Navigation werden in zwei diagonal gegenüberliegenden Ecken definiert. Der Test, der insgesamt 20 mal wiederholt wurde, lieferte für diese Navigation eine Durchschnittsdauer von 16,41 s mit einer Stan-



(a) Das Hindernis, das dem *RB-Kairos* während der Zeitmessung in den Weg gestellt wird.



(b) A) Das Navigationsziel mit der Ausrichtung entlang des Pfeils, B) Hindernis, C) Aktuelle Position des Roboters, D) Ungefährer Navigationspfad, um dem Hindernis auszuweichen.

Abbildung 6.1: Der Testablauf während der Messung der Navigationsdauer. Der Roboter navigiert zwischen zwei gegenüberliegenden Ecken des Raums und umfährt dabei ein Hindernis.

dardabweichung von 3,90 s. Das Hindernis sowie das Schema des Navigationspfads sind in Abbildung 6.1 ersichtlich.

Der Zielpunkt der Navigation wird aufgrund der Roboterhardware und des verwendeten Navigationsverfahrens nur ungefähr erreicht. Dies ist aber unproblematisch, da der entstandene Fehler in der Platzierung der *Haptikwand* durch den *UR-10* ausgeglichen werden kann. Die Berechnung der Armbewegung ist jedoch an die Erfassung der Roboterposition am Navigationsende angewiesen und kann daher nicht schon während der Fahrt berechnet werden. Die Berechnungszeit von maximal 1 s ist somit auch beim Ausfahren des Arms einzukalkulieren. Das Tracking durch die *Lighthouse*-Technologie erfolgt in Echtzeit, wodurch kein Zeitverlust entsteht.

Zusammenaddiert ergibt sich dadurch eine Zeitspanne von knapp 28 s für das Zusammenfalten des Arms in die Transportstellung, die Navigation des Roboters an die gegenüberliegende Ecke des Trackingbereichs und das Ausfahren auf den Zielpunkt der *Haptikwand*. Diese Zeitdauer erscheint für die Verwendung in einem Echtzeitsystem wie einer *VR*-Anwendung überaus lange. Es handelt sich hierbei aber um eine Universallösung, die es erlaubt, eine haptische Erfahrung im gesamten Interaktionsvolumen und mit willkürlicher Ausrichtung zur Verfügung zu stellen. Maßgeschneiderte und optimierte Lösungen, wie sie in Abschnitt 7.1 (Verbesserungspotential) beschrieben werden, können die Zeitdauer um ein Vielfaches verbessern.

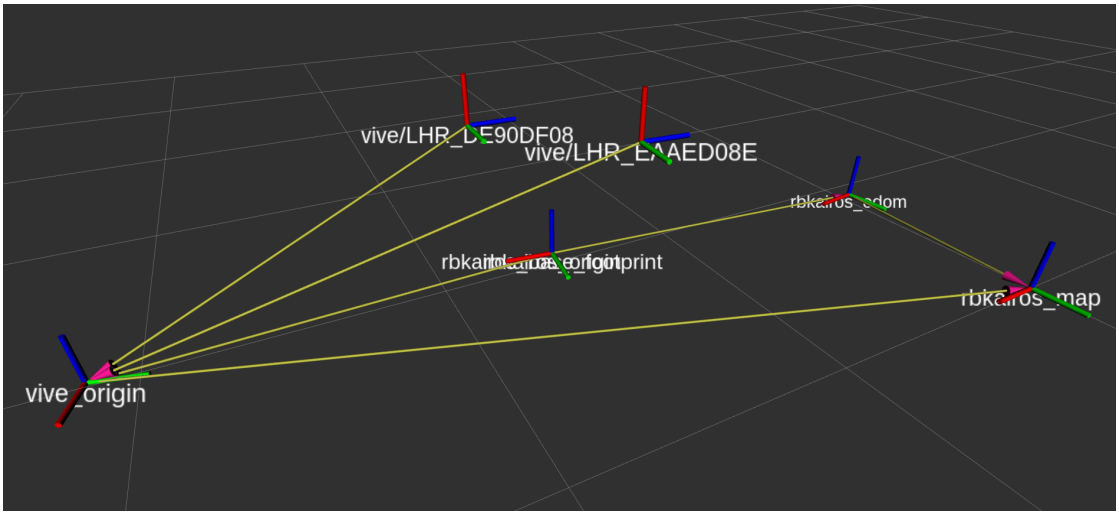
6.1.3 Genauigkeit

Die Genauigkeit des gesamten Systems ist an die Genauigkeit jeder einzelnen Komponente gebunden. Diese sind: die Lokalisation und Wegfindung der mobilen Basis, die Bewegungsausführung des *UR-10* und das *Lighthouse*-Tracking der *Vive* Pro. Auf etwaige andere Ungenauigkeiten durch Materialverformung oder locker sitzende Komponenten wird im Zuge dieser Arbeit nicht weiter eingegangen.

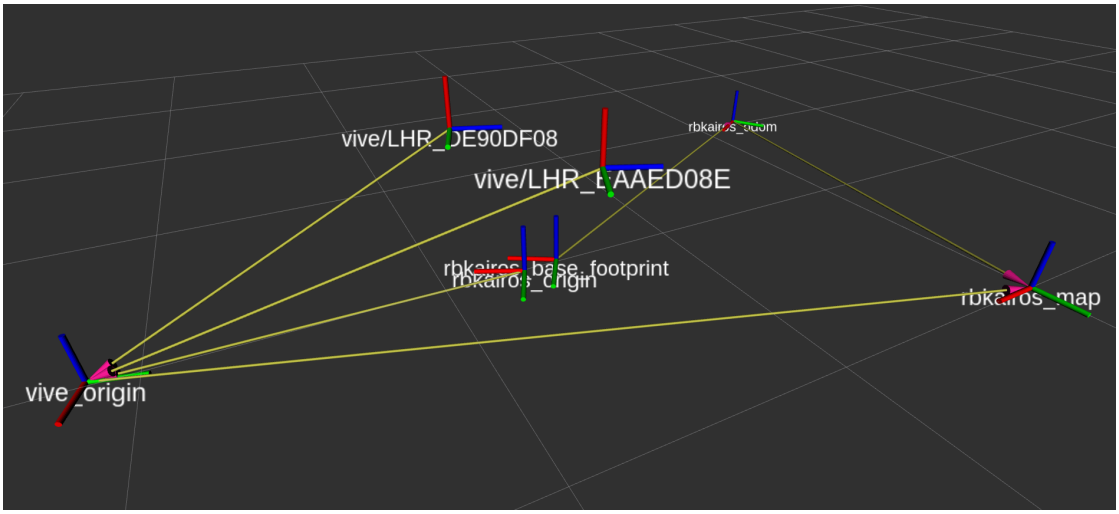
Der *RB-Kairos* findet sich im Raum mittels *Adaptive Monte Carlo Localization (AMCL)* zurecht. Bei diesem Verfahren werden die Daten der beiden Laserscanner und der *Odometrie* mit einer bereits bekannten Karte des Versuchsraums verglichen und dadurch die ungefähre Position im Raum bestimmt. Das Verfahren ist so konzipiert, dass auch mit ungenauen oder unvollständigen Daten eine Lagepositionierung erfolgen kann. Einerseits ist die *Odometrie*, die über die Bewegung der vier *Mecanum-Räder* berechnet wird, einem gewissen Verschub ausgesetzt, die durch die unüberwachte Bewegung der *Mecanum-Rollen* entstehen kann. Andererseits sorgen Objekte, die vom *Lidar* erfasst werden, aber nicht auf der Karte vorhanden sind, für ein Auseinanderdriften der beiden Vergleichswerte. Dies hat zur Folge, dass die Koordinaten der robotereigenen Lokalisation und der der *Vive* schon kurz nach der Kalibration nicht mehr übereinstimmen.

In Abbildung 6.2 wird dies verdeutlicht. Der Welt-*Frame* lautet `rbkairos_map`, der *Frame* der *Vive* `vive_origin`. `rbkairos_odom` entspricht dem Startpunkt des Roboters im aktuellen Betrieb und der Vektor zu `rbkairos_base_footprint` der *Odometrie*, also dem Weg, den der Roboter zu seiner aktuellen Position zurückgelegt hat. Aus den Koordinaten der beiden *Tracker* `vive/LHR_*` wird die Position des Roboters `rbkairos_origin` im Koordinatensystem der *Vive* ermittelt. Bei der Kalibration wird der `vive_origin` *Frame* so gelegt, dass die Roboterpositionen `rbkairos_base_footprint` und `rbkairos_origin` überlagern (siehe Abbildung 6.2a). Nach einigen Metern an Fahrt befinden sich die beiden Punkte jedoch nicht mehr übereinander, sondern nur mehr in ungefährer Nähe (siehe Abbildung 6.2b). Die Veränderung und Neuberechnung der Koordinaten von `rbkairos_odom` verdeutlichen zusätzlich die Ungenauigkeit des Verfahrens. Um diese Ungenauigkeit zu umgehen, wird sie bei der Positionierung der *Haptikwand* gar nicht verwendet. Das System verwendet dieses Verfahren nur zur ungefähren Navigation ans Ziel, wo im Anschluss das genauere *Lighthouse*-Tracking zum Einsatz kommt.

Bei der *Vive* handelt es sich um ein Produkt aus der Videospielebranche. Ihr Verwendungszweck dient der Unterhaltung von Menschen und ist daher auch auf diesen Markt zugeschnitten. Dies impliziert somit auch, dass sie im Rahmen dieser Diplomarbeit als Trackinglösung für Roboter zweckentfremdet verwendet wird und eventuell den Anforderungen des hier beschriebenen Szenarios gewachsen ist. Tatsächlich gibt es Arbeiten, zum Beispiel jene von Borges et al. [BSC+18], die sich der Verwendung der *Vive* in einem reinen Robotiksetting widmen und das System auf ihre dortige Tauglichkeit überprüfen. Sie kommen zum Ergebnis, dass in ihrem Testablauf die *Vive* beim Erfassen von statischen *Trackern* eine Standardabweichung von weniger als 0,5 mm zum gemessenen Punkt bietet.

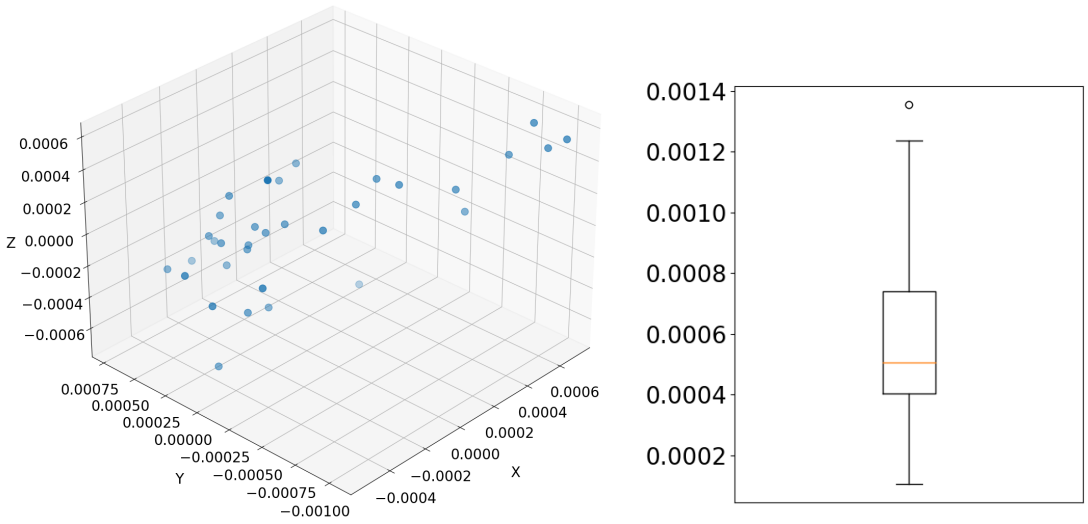


(a) Koordinaten zum Zeitpunkt der Kalibration. *rbkairos_origin* und *rbkairos_base_footprint* liegen genau übereinander.



(b) Durch Ungenauigkeiten, die bei der Bewegung des *RB-Kairos* auftreten, divergieren die Koordinaten der beiden Trackingmodelle auseinander.

Abbildung 6.2: Vergleich der Koordinaten in *RVIZ* bei der Kalibration und nach einigen Metern Roboterbewegung.



(a) Die gemessenen *Tracker*-Positionen relativ zum gemeinsamen Mittelpunkt.

(b) Die Verteilung der euklidischen Distanz der gemessenen Punkte zum Mittelpunkt.

Abbildung 6.3: Auswertung von 33 Messungen eines einzelnen, unbeweglichen *Trackers*. Alle Angaben in Metern.

Beim Tracken von Bewegungen verlässt sie sich jedoch zu einem geringeren Anteil auf das Tracking durch die *Lighthouse*-Lasersignale und stärker auf die eingebaute *IMU*. Diese auf Beschleunigungs- und Trägheitssensoren basierende Messeinheit sorgt für ein sanftes Erscheinungsbild des Bewegungspfads in Videospielen, liefert jedoch nicht die Positionsgenauigkeit eines optischen Systems.

Um die Genauigkeit des Trackings in dieser Diplomarbeit zu ermitteln, wurden zwei Tests durchgeführt. Im ersten Test wurde ein *Vive-Tracker* auf dem Boden des Trackingbereichs platziert und seine Position an dieser Stelle wiederholt gemessen. Dabei zeigte sich, dass die 33 gemessenen Koordinaten eine durchschnittliche Abweichung von 0,579 mm zum gemeinsamen Mittelpunkt aufweisen, während die Standardabweichung 0,297 mm beträgt (siehe Abbildung 6.3). Die Ergebnisse stimmen mit jenen von Borges et al. [BSC+18] überein, jedoch liefert der durchgeführte Test die Trackinggenauigkeit nur an einer einzelnen Stelle des Versuchsraums, weshalb zusätzlich ein zweiter Test durchgeführt wurde.

Der zweite Test ermittelt die Genauigkeit der *Lighthouse*-Koordinaten an unterschiedlichen Stellen des Trackingbereichs. Dieser wurde mit einer hölzernen dreieckigen Hilfskonstruktion vermessen (siehe Abbildung 6.4). Der Abstand zwischen den Löchern in den Ecken des Dreiecks beträgt jeweils 1 m. Auf einem Loch wurde ein wegklappbarer *Tracker* angebracht, um an dieser Stelle die *Vive*-Koordinaten zu ermitteln aber auch weiterhin Sichtkontakt zum darunter befindlichen Loch zu haben. Die Konstruktion wurde in der Mitte des Trackingbereichs platziert und die drei Löcher am Fußboden markiert. Zudem

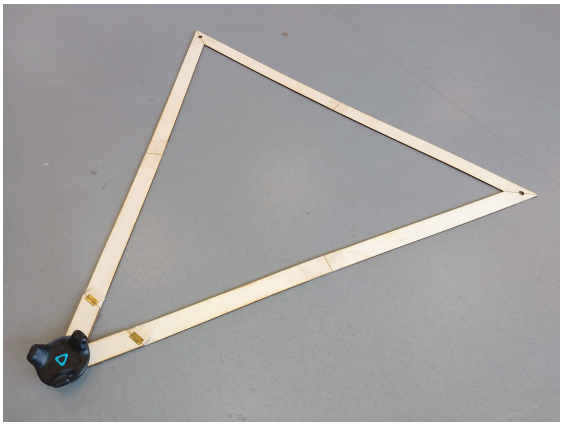


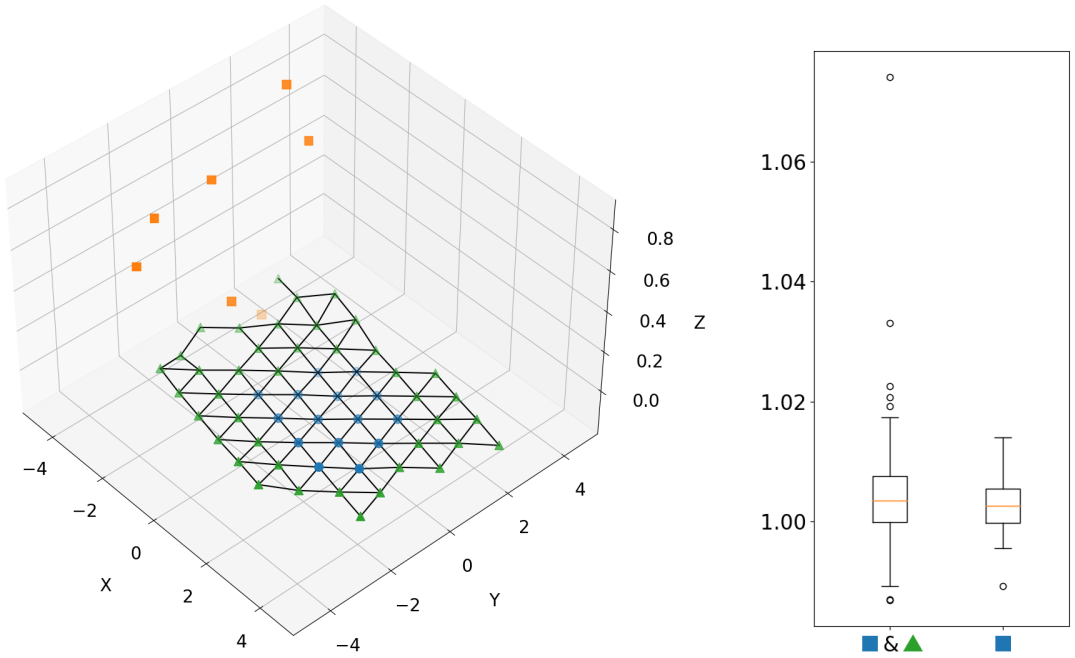
Abbildung 6.4: Die Hilfskonstruktion, mit der der Trackingbereich vermessen wurde.

wurden die *Tracker*-Koordinaten an jedem der Punkte erfasst. Durch Neupositionierung des Dreiecks über zwei bestehende Positionen und Markierung einer neuen kann der gesamte Bereich trianguliert und vermessen werden.

Abbildung 6.5a zeigt die 63 gemessenen Punkte in Relation zu ihrem Mittelpunkt. Die Daten signalisieren, dass die Genauigkeit am Rand des Trackingbereichs spürbar abnimmt. Die gemessene Höhe entlang der Z-Achse unterscheidet sich dort um bis zu 870 mm von jenen der anderen Punkte, obwohl der *Tracker* immer auf dem Boden platziert wurde. Dies hat damit zu tun, dass sich die *Tracker* an diesen Stellen fast vertikal unter den *Lighthouses* befinden und so die Grenzen des Trackingbereichs überschritten wurden. Aus diesem Grund werden alle Punkte mit einer Z-Höhe von +/- 100 mm gegenüber dem Mittelpunkt ausgefiltert und fließen nicht in die weiteren Berechnungen ein (orange Quadrate).

Aus den verbleibenden 56 Punkten (blaue Kreise und grüne Dreiecke) werden die Kanten zu ihren direkten Nachbarn ermittelt. Alle benachbarten Punkte weisen aufgrund der Dreieckskonstruktion (Abbildung 6.4) eine Distanz von circa 1 m zueinander auf. Aus einer Liste von allen möglichen Punktepaaren werden daher all jene entfernt, deren Abstand sich nicht in einem Bereich von 1.000 mm +/- 100 mm befinden. Aus diesem Schritt ergeben sich 134 Nachbarkanten, die in Abbildung 6.5a durch schwarze Linien angezeigt werden.

Die Kanten weisen eine durchschnittliche Länge von 1.004,178 mm auf während die Standardabweichung 9,277 mm beträgt. Die gemessenen Kanten sind somit durchschnittlich um 4,178 mm länger als die Dreiecke anhand derer sie vermessen wurden. In den Daten zeigt sich außerdem ein Anstieg der Genauigkeit, je zentraler sich ein gemessener Punkt im Trackingbereich befindet. Um diese auch statistisch zu ermitteln, wurde 15 Koordinaten (blaue Kreise) im Inneren des Trackingbereichs ausgewählt und die Berechnung der Kantenlänge auf diesen Bereich beschränkt wiederholt. Hierbei wurden 31 Kanten ermittelt, die einen Mittelwert von 1.002,248 mm und eine Standardabweichung von 4,881 mm



(a) Die erfassten Koordinaten relativ zum gemeinsamen Mittelpunkt und die daraus ermittelten Kanten zwischen benachbarten Punkten.

(b) Die Verteilung der Kantenlängen des gesamten und des innersten Bereichs.

Abbildung 6.5: Die Ermittlung der Genauigkeit des Trackingverfahrens durch Messung der Koordinaten in einem dreieckigen Raster mit einem Abstand von 1 m. Alle Angaben in Metern.

aufweisen. Diese Werte stellen circa eine Verdoppelung der Genauigkeit der inneren Kantenlängen im Vergleich zu jenen der Gesamtmessung dar (siehe Abbildung 6.5b).

Aus den Messergebnissen kann daher in Summe geschlossen werden, dass das *Lighthouse*-Verfahren fähig ist, einen Punkt innerhalb der Grenzen des Trackingbereichs mit einer Genauigkeit von etwa +/- 10 mm zu erfassen. Innerhalb der von den *Lighthouse*-Stationen gut erfassten Innenbereichen ist auch eine genauere Ortung möglich. Der Unterschied zu den Werten von Borges et al. [BSC+18] ergibt sich vermutlich aus dem Testaufbau selbst, der sehr praxisnah durchgeführt und bei dem eine einzelne Koordinate nur ein einziges Mal vermessen wurde. Zudem können die Lichtverhältnisse und die Bodenbeschaffenheit im Versuchsraum eine Rolle spielen.

Die *Lighthouse*-Technologie ermöglicht somit die Position des *RB-Kairos* nach dem Ende der Navigation zu ermitteln und so den Verfahrensfehler der *Odometrie* zu negieren. Ab diesem Zeitpunkt obliegt es nun dem *UR-10*, die verbleibende Distanz zum Zielpunkt der *Haptikwand* zurückzulegen. Der Hersteller *Universal Robots* gibt für seinen *UR-10* eine Wiederholgenauigkeit der Bewegungsausführung von +/- 0,1 mm an [ur1]. Diese ist

ISO 9283 zertifiziert und bestätigt somit die Genauigkeit des vorliegenden Systems unter Einhaltung der Betriebsparameter wie Nutzlast und Umgebungstemperatur.

Die vorliegenden Daten zeigen also, dass das erstellte *VR*-System in der Lage ist, die *Haptikwand* an einer willkürlichen Stelle des Interaktionsvolumens mit einer Genauigkeit von +/- 10 mm innerhalb von 28 s zu platzieren. Um diese Werte auch subjektiv zu überprüfen und den Realismusgrad der Haptik zu testen, wurde ein Pilotversuch durchgeführt.

6.2 Benutzererfahrung

Um das Zusammenspiel der Komponenten zu überprüfen und festzustellen, ob das System fähig ist eine glaubhafte Haptik bereitzustellen, wurde ein Pilottest mit einer Testperson durchgeführt. Der Ablauf des Tests erfolgte folgendermaßen:

Die Testperson betrat den Versuchsraum, legte das *HMD* der *Vive* an und erkundete die virtuelle Szene. Die Person wurde befragt, wie sich die *virtuelle Realität* anfühlte und was sie von dieser Szene hielt. Ihre Antwort war, dass die freie Bewegungsmöglichkeit im gesamten Raum durch das kabellos betriebene Gerät eine Neuheit für die Person darstellte, die sie in vorhergehenden *VR*-Erfahrungen noch nicht erlebt hatte. Die grafische Anzeige im *Headset* und die Veränderung der *virtuellen Umgebung* während ihrer Bewegung stimmte mit den Erwartungen überein, die die Testperson auch aus der Realität kannte. Anders als in bisherigen Erfahrungen, bei denen Benutzer und Benutzerinnen an der Stelle verharrten und virtuelle Orte typischerweise per Teleportierung erreichten, erlaubte die kabellose Bewegungsfreiheit in diesem Setup auch ein Erreichen des Ziels durch natürliches Zufußgehen. Umso mehr fiel es der Person auf, dass die Hindernisse der *virtuellen Umgebung* über keinerlei physische Substanz verfügten und mühelos durchquert werden konnten.

Der Testperson wurde nun ein *Vive-Controller* in die Hand gereicht und sie wurde beauftragt mit ihm eine Oberfläche zu markieren, die sie mit Haptik ausstatten mochte. Daraufhin zielte sie mit dem Laserpointer des *Controllers* auf eine Seite eines Würfels und betätigte den *Trigger*. Die anvisierte Seite wurde farblich markiert und bestätigte der Testperson die Auswahl einer haptischen Fläche.

Die Koordinaten der gewünschten Fläche wurden an den Roboter übertragen und dienten ihm als Zielpunkt der *Haptikwand*. Der Roboter, der sich am Rande des Versuchsraums befand, brachte seinen *Manipulator* in eine zusammengeklappte Transportstellung, erstellte einen Navigationsplan in die unmittelbare Nähe des Zielpunkts und setzte sich in Bewegung. Am Zielort angekommen, ermittelte der Roboter seine tatsächliche Position anhand des *Lighthouse*-Trackingsystems. Damit berechnete er den verbleibenden Weg zum Zielpunkt und beauftragte die Planungssoftware des Roboterams einen Bewegungspfad dorthin zu erstellen. Nach der Ausführung der *Manipulator*-Bewegung befand sich nun die *Haptikwand* an jener Stelle der Realität, die der Markierung in der *virtuellen Umgebung* entsprach.

Die Testperson versuchte nun das virtuelle Objekt zu ertasten und tatsächlich stießen ihre Finger auf ein Hindernis. Die Empfindungen der Finger passten mit den Informationen zusammen, die die Augen beim Betrachten des Würfels lieferten. Sie sah zum Rand des Würfels und auch ihre Finger ertasteten den Rand der *Haptikwand*. Die Person lehnte sich gegen den virtuellen Würfel. Anders als vorher hielt der Würfel dem Gewicht der Testperson stand, da dieser nun offenbar real existierte.

Die Person wurde befragt, was sie von den neuen Empfindungen hielt. Sie sagte, dass die Fläche nun anders als zuvor ertastbar sei und ihr einen deutlichen Widerstand, beim Versuch durch das Objekt hindurch zu greifen, bot. Die Oberfläche des Würfels fühlte sich hölzern an. Die Empfindung stimmte jedoch nicht mit der Erwartung überein, die aufgrund der Farbe des virtuellen Objekts angenommen wurde. Eine glatte Oberfläche aus Kunststoff oder eine kühle aus Metall wurden aufgrund der weißen Farbe erwartet. Auf Nachfrage, ob es sich nicht auch um weiß bemaltes Holz handeln könnte antwortete die Testperson, dass dies möglich sei, aber es der sterile Ersteindruck der virtuellen Szene nicht vermuten ließ.

Bei der Berührung des Würfelrands fiel der Testperson auf, dass die angrenzenden Seiten nicht vorhanden waren. Obwohl ihr im Vorfeld erklärt wurde, dass nur eine einzige Fläche zur selben Zeit haptisch simuliert werden konnte, wurde dennoch die Existenz der weiteren Würfelseiten erwartet. Beim Versuch, den Würfel zu umrunden stieß die Testperson mit den Beinen ans Roboterchassis. Die Testperson kommentierte, dass sich hier nun ein Hindernis befand, man aber in der *virtuellen Umgebung* nicht darauf hingewiesen wurde.

Im Anschluss wurden von der Testperson weitere Flächen mit dem *Controller* markiert, um die Haptik auch an diesen Stellen zu testen. Als Fazit gab die Person an, dass die zur Verfügung gestellte Haptik die visuelle *VR*-Simulation um einen realistischen Sinneseindruck erweiterte, aber aufgrund der Größe des Versuchsraums mehr als eine einzige haptische Interaktionsmöglichkeit wünschenswert wäre. Der Gesamteindruck war positiv.

KAPITEL 7

Diskussion

Dieses Kapitel beschäftigt sich mit der Interpretation der Erkenntnisse aus Kapitel 6 (Evaluierung). Die technische Evaluierung zeigt, dass das erstellte *VR*-System in der Lage ist eine haptische Fläche an einer willkürlichen Stelle des Interaktionsvolumens bereitzustellen. Diese wird innerhalb von 28 s mit einer Genauigkeit von +/- 10 mm platziert.

Das Tracking des Roboters wurde so entwickelt, dass auch bei Ausfall des Sichtkontakts eines einzelnen *Trackers* die Ortung weiterhin aufrecht bleibt. Die Evaluation zeigt, dass eine Neupositionierung der *Haptikwand* auch immer mit einer Bewegung des Roboterchassis einhergehen muss und, dass dadurch die Zeitdauer für die Platzierung der Haptik negativ beeinflusst wird. Weiters wurde festgestellt, dass die Genauigkeit des *Lighthouse*-Trackings positionsabhängig ist und in den Randbereichen des Trackingvolumens nur mehr begrenzt brauchbare Werte liefert.

Der Pilottest hat gezeigt, mit welchen Hindernissen Testpersonen konfrontiert sind und welche Erwartungen sie an das System haben. Zum Beispiel fällt eine fehlende Interaktionshaptik mit der *virtuellen Umgebung* besonders dann auf, wenn sich Benutzer und Benutzerinnen zu Fuß durch die Szene bewegen können, anstatt sich wie in typischen *VR*-Simulationen durch ein Level zu teleportieren. Die möglichen Ansätze zur Verbesserung des *VR*-Systems werden in Abschnitt 7.1 (Verbesserungspotential) behandelt.

Bei der am *Endeffektor* des Roboters angebrachten Requisite handelt es sich um eine flache Holzplatte, deren Oberfläche oder Beschaffenheit nicht verändert werden kann. Sie unterstützt somit nur eine haptische Simulation von Objekten ihresgleichen. Aus diesem Grund wird sie exklusiv zur Simulation als Wand oder flaches Hindernis verwendet. In Abschnitt 7.2 (Erweiterungsmöglichkeiten) werden daher Konzepte beschrieben, mit denen die Anzahl an simulierbaren Oberflächen vergrößert, beziehungsweise durch Robotermanipulation um dynamische Haptik erweitert werden kann.

Abschnitt 7.3 (Anwendungsmöglichkeiten) listet schließlich Anwendungsgebiete, für die ein solches *VR*-Systems in Frage kommen könnte und gibt einen Ausblick auf zukünftige Entwicklungen.

7.1 Verbesserungspotential

Bei der Implementation dieser Diplomarbeit handelt es sich um eine Kooperation von diversen Komponenten aus unterschiedlichen Themenbereichen der Informationstechnologie. Sowohl die Verbesserung der einzelnen Komponenten als auch ihr Zusammenspiel können sich positiv auf das Gesamtergebnis auswirken. Im Verlauf dieses Abschnitts werden daher Punkte gelistet, die zu einer Verbesserung des Systems beitragen können.

In Abschnitt 3.3 (Robot Operating System) wird beschrieben, dass der *RB-Kairos* seine eingebauten *Lidars* sowie die berechnete *Odometrie* seiner Fortbewegung verwendet, um sich im Raum zurechtzufinden und durch diesen zu navigieren. Unterabschnitt 6.1.3 (Genauigkeit) erwähnt jedoch, dass die dadurch gewonnene Positionsbestimmung und Navigationsroutine nicht präzise genug agiert und dass das System daher eine erneute Messung durch die *Lighthouse*-Technologie durchführt. Zudem ist das System auf eine aktuelle Karte des Versuchsraums angewiesen und hat Probleme, wenn diese nicht mehr mit der Umgebung übereinstimmt. Es liegt daher nahe, die von *ROS* zur Verfügung gestellte Trackingkomponente durch eine zu ersetzen, die auf der *Lighthouse*-Technologie basiert.

Ein Tracking basierend auf der *Lighthouse*-Technologie könnte die Position des Roboters direkt im *Transform Tree* veröffentlichen und so für eine genaue Lokalisation in absoluten Raumkoordinaten sorgen. Der `rbkairos_odom`-*Frame* wäre damit obsolet. Die *Odometrie* könnte weiterhin als Fallback im System bleiben, falls es zu Störungen des *Lighthouse*-Trackings kommen sollte. Die Erfassung der *Tracker* ist jedoch in der aktuellen Versuchsanordnung auf den manuellen Start von *Steam* und *SteamVr* angewiesen und bis zu diesem Zeitpunkt nicht verfügbar.

Der Start von *SteamVr* geschieht im aktuellen Setup durch händischen Aufruf der Programmverknüpfung auf dem Desktop des Touchdisplays. Wie in Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos) beschrieben, wird *SteamVr* als Unterhaltungsprodukt für die Verwendung auf Spiele-PCs für private Zwecke entwickelt und nicht als Trackinglösung für Roboter. Der Start der Software lässt sich damit auch nur soweit automatisieren, bis eine Usereingabe notwendig wird. Diese reicht von der Aufforderung den in Abbildung 5.6 gezeigten Fehler zu beheben, bis zur Durchführung von Software Updates. Die beiden am Chassis des *RB-Kairos* angebrachten *Vive-Tracker* benötigen zusätzlich eine Aktivierung per Fingerdruck auf ihren Einschaltknopf und wechseln automatisch in den Standby-Modus, sollte keine nennenswerte Bewegung der Tracker registriert werden. Eine Lösung dieser Probleme konnte im Zuge dieser Diplomarbeit nicht gefunden werden, könnte aber eventuell durch den Erwerb einer proprietären Firmenlizenz von *SteamVr* [stea] abgedeckt sein.

Die Zeitdauer, die der Roboterarm zum Platzieren der Requisiten benötigt, könnte durch rechtzeitige Vorberechnung seiner Bewegung vermindert werden. Im aktuellen Versuch erfolgen Planung und Ausführung der Manipulation unmittelbar nacheinander. Während der *RB-Kairos* den Benutzern oder Benutzerinnen die *Haptikwand* an einer bestimmten Stelle zur Verfügung stellt, könnte bereits die Berechnung zum Einziehen des Arms in die Transportstellung begonnen werden. Dasselbe gilt für das Ausstrecken zum Zielort. Das System könnte bereits während der Roboterfahrt die anschließende Armbewegung vom ungefähren Startpunkt berechnen, während der Ausführung die tatsächliche *Vive*-Position bestimmen und den Roboterarm zu einer Kurskorrektur veranlassen.

Eine weitere Vereinfachung des Systems würde eine Integration der Bewegungssteuerung des Fahrwerks in das *Motion Planning Framework MoveIt* bringen. In der Roboter-Konfiguration dieser Arbeit werden die Steuerung der *Mecanum-Räder* und des Roboterarms *UR-10* durch unterschiedliche Softwarekomponenten durchgeführt. Dies erleichtert die Implementation und Verbesserung der Einzelkomponenten, führt jedoch dazu, dass die Teile des Roboters nicht als Ganzes agieren.

Die *Mecanum-Räder* des *RB-Kairos* erlauben dem Roboter eine Fahrt in alle Richtungen. Er kann sich somit auf einer zweidimensionalen Ebene, in diesem Fall dem Fußboden des Versuchslabors, uneingeschränkt fortbewegen und ist auf keine Richtungsänderung des Gefährts, wie sie etwa bei der *Achsschenkellenkung* klassischer Autos notwendig ist, angewiesen. Diese zweidimensionale Fahreigenschaft kann in *MoveIt* als planare Achse konfiguriert werden. Eine Drehung um die Vertikalachse, wie sie mit der *Mecanum*-Steuerung ebenfalls möglich ist, sorgt für einen zusätzlichen Gelenks-Freiheitsgrad. Mit dieser Konfiguration kann der *Endeffektor* des Roboterarms mit der gleichen Software wie der Basis gesteuert werden. Das Fahrwerk sorgt für eine grobe Bewegung im Raum, während der Roboterarm die Ungenauigkeiten zeitnah ausgleichen kann. So könnte man auch die in Unterabschnitt 6.1.2 (Zeitspanne) beschriebene Eigenkollision mit der Roboterbasis umgehen, da diese in diesem Fall einfach ausweichen kann, ohne dass für sie ein neuer Navigationsplan erstellt werden müsste.

Ein Ausweichen des Roboters ist auch dann sinnvoll, wenn Benutzer oder Benutzerinnen Gefahr laufen in ihn hineinzulaufen. Im Kontext der visuellen *VR*-Anwendung existiert der Roboter gar nicht, sondern nur das haptische Element, das er an seinem *Endeffektor* zur Verfügung stellt. So wäre es denkbar, dass der Roboter möglichen Zusammenstößen mit Benutzern und Benutzerinnen rechtzeitig ausweicht, während die gebotene haptische Erfahrung an Ort und Stelle verbleiben kann. Dieses Konzept ist auch dann hilfreich, wenn sich außer dem Roboter noch weitere Requisiten oder zusätzliche Roboter im Raum befinden. Ein Aneinaderreihen von mehreren *Haptikwänden*, wie es in *TurkDeck* (siehe Abbildung 2.3) praktiziert wird, wäre damit denkbar.

Im aktuellen Aufbau nimmt der *RB-Kairos* immer eine Position von 0,7 m hinter dem gewünschten Zielpunkt der *Haptikwand* ein. Der *RB-Kairos* ist mit Laserscannern ausgestattet, die ihm ein Abtasten seiner Umgebung erlauben. Mit den ermittelten Laserdaten können Hindernisse erkannt und umfahren werden. Sollte das Navigationsziel aber durch ein anderes Objekt oder einen Menschen blockiert sein, so kann der Roboter sein Ziel

gar nicht erreichen und bricht die Bewegung ab. Die Laserdaten haben jedoch nur einen Einfluss auf die Navigation des Chassis. Die Steuerungssoftware des Roboterarms greift darauf nicht zu. Die zweidimensionale Umgebungsrepräsentation durch das *Lidar* ist nur sinnvoll für die sich in einer zweidimensionalen Ebene fortbewegende Roboterbasis. Die zweidimensionalen Umgebungsdaten könnten aber für die Verwendung als Kollisionsgeometrie in *MoveIt* um eine dritte Dimension bis Raumhöhe erweitert werden. Eleganter und effektiver wäre es jedoch, die Hindernisse dreidimensional zu erfassen, um so Armbewegungen unter und über möglichen Hindernissen zu erlauben.

Ein Tracking von Hindernissen in drei Dimensionen ist ebenso über die *Lighthouse*-Technologie möglich. Bei rigiden Objekten reicht ein einzelner *Tracker* aus, um seine Position zu bestimmen. Eine vereinfachte Kollisionsgeometrie sorgt für die Vermeidung von Zusammenstößen. Auch Menschen können auf diese Art erfasst werden. Zum Beispiel können die Positionen von *Headset* und *Controllern* zur Approximation der Körperhaltung eines *VR*-Benutzers oder einer *VR*-Benutzerin herangezogen werden. Die Punktdaten der *Lidars* des *RB-Kairos* könnten die Stellung der Beine preisgeben. Für genaueres Körpertracking kann zusätzliches Equipment, zum Beispiel Motion-Tracking-Anzüge, verwendet werden.

Sollte der *UR-10* durch seine Drucksensoren ein Hindernis erkennen oder einer der Notstopp-Buttons auf Roboterchassis oder Funkfernsteuerung betätigt werden, so schaltet der *RB-Kairos* in den Ruhemodus. In diesem Modus werden alle Kommandos der Robotersteuerung gestoppt und die Gelenkbremsen des Roboterarms angezogen. Dieser Modus wird ebenso beim Herunterfahren des Robotersystems eingeschalten. Eine Reaktivierung des Roboters erfolgt durch Herausdrehen des Notstopps sowie dem Drücken der *Restart*-Taste auf der Rückseite des *RB-Kairos*. Zu diesem Zeitpunkt führt die Navigationssoftware ihren Kurs fort und auch eine manuelle Steuerung über den *PlayStation 4 Controller* ist wieder möglich. Der *UR-10* bleibt jedoch weiterhin deaktiviert, da für seine Reaktivierung ein zusätzlicher Schritt notwendig ist. Er erfordert den Anschluss eines Displays und einer Computermaus über die designierten HDMI- und USB-Buchsen auf der Rückseite des Chassis. Dafür kann das bereits am Roboter angebrachte Touchdisplay verwendet werden. Die nun am Display erscheinende Bedienungsoberfläche *Polyscope* zeigt die Meldung über den Notstopp, die per Toucheingabe weggeklickt werden kann. Im Initialisationsbereich kann schließlich die Operation des Roboterarmes wieder freigegeben werden, woraufhin sich die Gelenkbremsen wieder lösen.

Der Roboter wäre nun wieder bereit seinen Dienst fortzufahren, jedoch muss sein Tracking neu gestartet werden. Wie in Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos) beschrieben, beendet sich *SteamVr* von selbst, sollte es kein angeschlossenes Display mehr erkennen. Dies hat auch zur Folge, dass die gestartete *Node*, die für die Überführung der Trackingkoordinaten in *ROS* verantwortlich war, beendet wurde. Ein Neustart von *SteamVr* und der Tracking *Node* bringt das System wieder auf den Status Quo vor der Aktivierung des Notstopps. Ein Ausweg aus diesem Dilemma würde die Verwendung eines zusätzlichen Displays für das User Interface des *UR-10* bringen. Möglicherweise ist auch die Reinitialisierung des Roboterarms per Software anstatt per Bedienoberfläche

möglich. Eine solche Lösung wurde aber zum Zeitpunkt des Verfassens dieser Arbeit nicht gefunden.

Neben diesen konzeptionellen existieren noch weitere kleine Verbesserungsmöglichkeiten, auf die eingegangen wird. Das Betriebssystem des Roboters *Ubuntu 16.04* und die installierte *ROS*-Version Kinetic Kame sind zum Zeitpunkt der Erstellung dieser Diplomarbeit in die Jahre gekommen. Dies macht sich zum Beispiel dadurch bemerkbar, dass *Python 2.7*, die einzige Version, die von der *ROS*-Distribution unterstützt wird, seit 1. Januar 2020 keinen Support mehr erhält [pyt]. Aus diesem Grund wurden auch viele Programmbibliotheken, die ursprünglich *Python 2.7* unterstützten, auf eine neuere Version umgestellt. Dies hat zur Folge, dass einige Bibliotheken nach deren Update plötzlich nicht mehr am bestehenden System ausgeführt werden können. Um dies zu umgehen, muss daher explizit auf ältere Versionen dieser Software zurückgegriffen werden, die jedoch keinerlei Bugfixes oder Sicherheitsupdates mehr erhalten. Ein Upgrade von Betriebssystem und *ROS*-Version würde die Aktualität der Robotersoftware sicherstellen, hat jedoch auch den Nachteil, dass einige Programmteile dadurch ebenso inkompatibel werden können. Es müsste daher das gesamte System auf Kompatibilität überprüft werden und die einzelnen Komponenten gegenbenenfalls an die neue Funktionsweise angepasst werden, wobei es sich um einen beträchtlichen Zeitaufwand handeln könnte.

Eine weitere Verbesserung der Softwarearchitektur besteht in der ausschließlichen Verwendung von Open Source Software. *ROS* selbst und die meisten Bibliotheken des *RB-Kairos* gehören dazu. Einige verwendete Bibliotheken des Herstellers *Robotnik* [roba] sind jedoch proprietär und nicht im Internet aufzufinden. Dies hat zur Folge, dass die Funktionsweise einiger Komponenten nicht dokumentiert oder nicht nachvollziehbar ist. Software Updates sind so ebenso nur durch Nachfrage beim Hersteller zu beziehen. Die proprietären Bibliotheken könnten also auch leicht ihre Kompatibilität zu ihren laufend aktualisierten *Open-Source*-Pendants verlieren, sollten sich Hersteller dazu entscheiden ihren Support einzustellen.

7.2 Erweiterungsmöglichkeiten

Wie in der Einleitung dieses Kapitels erwähnt, erlaubt die aktuelle Installation am Roboter nur die Simulation von flachen, unbeweglichen Oberflächen. Um hier mehr Flexibilität zu bieten, könnte die Requisite am *Endeffektor* des *UR-10* durch weitere Funktionen erweitert werden. So könnte dort ein *Robotic Shape Display* [SGY+17] angebracht werden, das eine dynamische Simulation von verschiedenen Oberflächen erlaubt. Abbildung 7.1 zeigt ein solches Display aus *ShapeShift* [SGY+17] von Siu et al. Außerdem wäre es denkbar, den Roboter mit unterschiedlichen haptischen Objekten auszustatten. Ähnlich der *HapticHydra* aus *Haptic Snakes* [ASJR+19] von Al-Sada et al. könnten diese fest am *Endeffektor* des Roboters angebracht sein oder in einem Objektlager bis zum gezielten Einsatz aufbewahrt werden. Dieses Magazin könnte sich an einem unzugänglichen Ort in der Nähe des Versuchsbereichs befinden oder, aufgrund der Größe des Roboters, auch auf ihm angebracht sein. Der Austausch und eigenständige Wechsel des *Endeffektors* ist

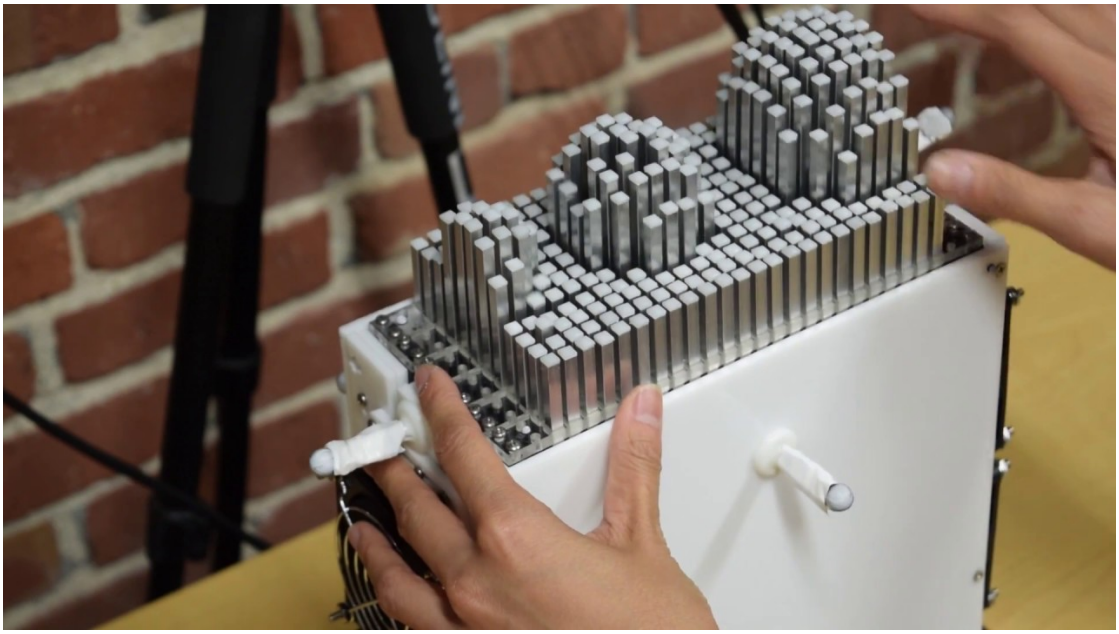


Abbildung 7.1: Das *Robotic Shape Display* von *ShapeShift* [SGY+17] erlaubt die dynamische Simulation von Oberflächen.

jedoch mit einem großen Entwicklungsaufwand verbunden, vorallem bei der Hardware. Abbildung 7.2 zeigt den multifunktionalen Endeffektor der *HapticHydra*.

Der Roboterarm *UR-10* wird vom Hersteller als kollaborierender Roboterarm beworben, der einen Betrieb in unmittelbarer Nähe zu Menschen, beziehungsweise im Zusammenspiel mit Menschen möglich machen soll. Er besitzt Sensoren, die Kollisionen und äußere Krafteinwirkungen auf den *Manipulator* erkennen können, um im Notfall die Bewegung zu stoppen. Die Sensorik wird auch dazu verwendet, um im sogenannten *Freedrive Modus* die Gelenke des Arms per Hand zu bewegen. Der *UR-10* liest in diesem Modus die ausgeübten Kräfte auf sich aus und bewegt seine Gelenke in die Richtung der Kraftwirkung. Der Treiber des *UR-10* publiziert diese Daten ebenfalls, allerdings nur in einer vereinfachten Form als Summe aller Krafteinwirkungen auf den *Endeffektor*. Diese werden als `wrench`-*Topic* (Auflistung A.7) in *ROS* publiziert und beinhalten je einen 3-dimensionalen Vektor für Linear- und Rotationskräfte.

Die Daten des `wrench`-*Topics* können simplifiziert als Drucksensor interpretiert werden und dem Robotersystem als Input dienen. Eine mögliche Anwendung ist die Simulation eines großen, fahrbaren Behälters, zum Beispiel eines Müllcontainers oder einer Grubenlore, indem die *Haptikwand* in die Richtung des Benutzers oder der Benutzerin ausgerichtet wird und die Oberfläche des Gefährtes imitiert. Bringt der Benutzer oder die Benutzerin genügend Kraft gegen die Wand auf, so manövriert der gesamte Roboter in die entsprechende Richtung oder, im Falle der Grubenlore, entlang eines Pfades. Je nach Masse oder Inhalt des zu simulierenden Behälters wäre mehr oder weniger Kraftausübung seitens des

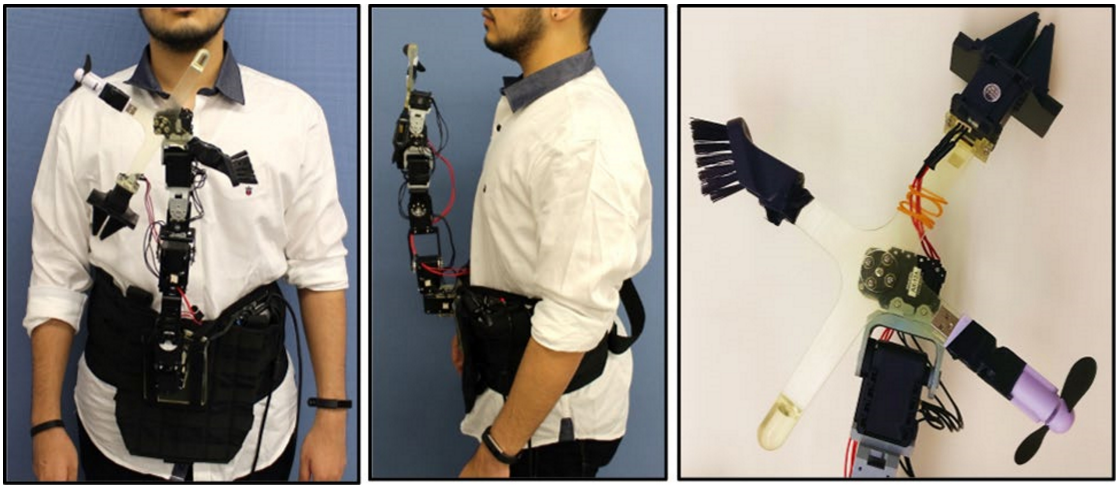


Abbildung 7.2: Der multifunktionale *Endeffektor* der *HapticHydra* [ASJR+19] bietet einen Greifarm, ein Gebläse, einen „Finger“ und eine Bürste.

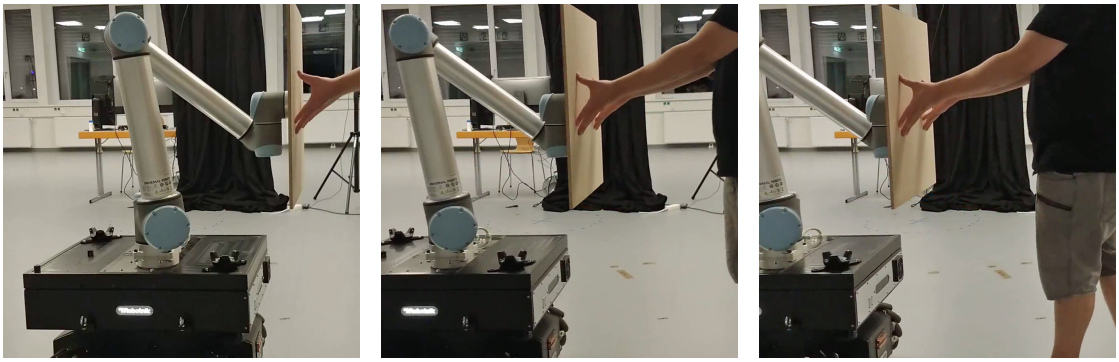


Abbildung 7.3: Der Versuchsaufbau *Haptic Wrench*. Der Druck gegen den Roboterarm wird vom Treiber des *UR-10* erkannt und in Bewegungskommandos für das Fahrwerk umgerechnet. Der Roboter kann so per Hand weggeschoben werden.

Benutzers oder der Benutzerin notwendig, um diesen zu bewegen. Die *Mecanum-Räder* erlauben dem Roboter eine zusätzliche Flexibilität, da dadurch eine Bewegung in alle Richtungen möglich ist. Abbildung 7.3 zeigt diesen Versuchsaufbau der *Haptic Wrench*, der aufgrund der Komplexität nicht in das Projekt dieser Arbeit integriert wurde.

Die Kraftausübung auf den Roboterarm hat im Versuch der *Haptic Wrench* nur einen Einfluss auf die Bewegung der Räder, nicht jedoch auf die Stellung des Roboterarms. Das liegt daran, dass das Fahrwerk problemlos durch Angabe von Drehgeschwindigkeiten für jedes Rad gesteuert werden kann. Zum Beispiel kann ein starker Druck gegen den „Sensor“ eine entsprechend schnelle Drehung der Räder auslösen. Der *UR-10* unterstützt nativ, per *URScript* angesteuert, ebenso eine Bewegung des *TCP* in einer bestimmten Geschwindigkeit, der *ROS*-Treiber beziehungsweise *MoveIt* implementieren diese Funktion

jedoch nicht. Der Arm kann mit dieser Software nur stillstehende Posen einnehmen oder vordefinierte Pfade abfahren, jedoch keine dynamische Bewegung in eine bestimmte Richtung ausführen. Die Steuerung des Arms ohne *MoveIt* und über natives *URScript* birgt eigene Probleme, allen voran der Wegfall der Pfadplanungssoftware sowie der definierten Sperrzonen, die verhindern, dass der Roboterarm mit sich selbst kollidiert. Ein Umschalten beider Systeme ist durch Beendigung und Neustart einzelner *Nodes* möglich, es wird aber im Ökosystem von *ROS* davon abgeraten. Daher bleibt für eine saubere Umsetzung dieses Features nur der Weg über eine Treibererweiterung, welche jedoch nicht im Rahmen dieser Diplomarbeit liegt.

Mit dieser Technik ist die Simulation von nicht-linearen Bewegungen, wie zum Beispiel das Öffnen von Türen durch Rotation um die Vertikalachse, möglich. Durch eine geskriptete Kurvenfahrt des gesamten Roboters könnte dies im Versuch *Haptic Wrench* umgesetzt werden, aber die Eigenmasse von circa 130 kg sorgt für eine nicht zu unterschätzende Trägheit, weshalb davon abgesehen wurde. Viel besser dafür geeignet sind die beiden Vertikalachsen des *UR-10*, der die Drehung, je nach Masse der simulierten Türe, schneller oder langsamer ausführen könnte. Auch ein physikalisches Modell zur Simulation von Rotationen in willkürliche Richtungen wäre mit dem Roboterarm denkbar.

7.3 Anwendungsmöglichkeiten

Das *VR*-System dieser Arbeit bietet im aktuellen Versuchsaufbau eine solide, haptische Ergänzung zur gewohnten, kommerziellen *VR*-Erfahrung. Die Anwendungsmöglichkeiten sind jedoch aufgrund einiger Faktoren stark beschränkt.

Der Platzbedarf, die vielzähligen Komponenten und die Anschaffungskosten des Systems sprechen gegen einen Einsatz für den Heimgebrauch. Weiters verlangt die Konfiguration und Inbetriebnahme fortgeschrittene Kenntnisse im Bereich von *ROS* und *virtueller Realität*, die sich klar an Spezialisten oder Enthusiasten richten. Nach Optimierung der Anforderungen könnte dieses System aber als Simulationshardware im professionellen Bereich oder der Unterhaltungsindustrie wie etwa *VR*-Spielhallen dienen.

In einer *VR*-Erlebniswelt, wie sie etwa in *TurkDeck* [CRR+15] (siehe Abbildung 2.3) beschrieben wird, könnten die menschlichen Akteure durch Roboter dieser Diplomarbeit ersetzt werden. Dazu müssen aber entweder die Flexibilitätsanforderungen der Anwendung gesenkt oder das Interaktionsportfolio der Roboter erhöht werden. Abschnitt 7.2 (Erweiterungsmöglichkeiten) fasst einige Möglichkeiten zur Erweiterung des Angebots zusammen. *TurkDeck* zeigt jedoch eindrucksvoll den Flexibilitätsvorsprung, den ein Mensch gegenüber einem Roboter weiterhin inne hat, indem zum Beispiel vom menschlichen Akteur oder der Akteurin gefordert wird, seine oder ihre Requisite von einer Wand in eine Treppe umzuformen und diese auf den Boden zu legen.

Eine weitere Anwendungsmöglichkeit stellen Simulationen im professionellen Bereich dar. So gab es bereits eine Kooperation der *TU Wien* mit Soldaten des *Österreichischen Bundesheers*, bei der eine *VR*-Simulation im Bereich der Abwehr von atomaren, biolo-

gischen und chemischen (ABC) Kampfmitteln erprobt wurde [see]. Diese Anwendung wurde in *Immersive Deck* [PVS+16] integriert, die im selben Versuchsraum wie die Tests dieser Arbeit stattfand. Die Anwendung unterstützt die Simulation von unterschiedlichen Gefahrensituationen, bietet jedoch keinerlei haptisches Empfinden abseits der am Körper zu tragenden Ausrüstung. Mit der Simulation von Hindernissen, zum Beispiel versperrten Türen oder Kisten, die von den Soldaten vor dem weiteren Vorgehen aus dem Weg geschaffen werden müssen, könnte ein höherer Realismusgrad erreicht werden.

KAPITEL 8

Zusammenfassung

Diese Diplomarbeit beschreibt den Aufbau und den Betrieb eines *VR*-Systems, das haptische Sinneswahrnehmung in einem raumgroßen Setup bietet. Das durch das *Virtual-Reality-Headset Vive Pro* bereitgestellte virtuelle Erlebnis wird durch die mobile Roboterplattform *RB-Kairos* um haptische Elemente erweitert. Benutzer und Benutzerinnen können virtuelle Objekte in der *VR*-Anwendung markieren, welche durch Platzierung einer realen Requisite zur Bereitstellung eines haptischen Objekts an eben jener Stelle führt. Als Softwärelösungen kommen das *Robot Operating Systems* zur Steuerung und Verwaltung der Roboterfunktionen zum Einsatz sowie das Softwareentwicklungs-*Framework Unity* zur Darstellung der *VR*-Szene. Die vom *Virtual-Reality-Headset* zur Verfügung gestellte *Lighthouse*-Technologie liefert eine genaue Möglichkeit zur Ortung von Benutzern und Benutzerinnen und Objekten innerhalb eines Raums und bildet damit die Grundlage für die Ausführung einer punktgenauen Haptik am vorhergesehenen Ort. Die Roboterplattform kombiniert durch die Verwendung von *Mecanum-Rädern* und den auf der Roboterbasis angebrachten *UR-10* eine hohe Manövrierfähigkeit mit präziser Manipulation.

Die ausgeführte Evaluation zeigt die Stärken und Schwächen des Systems. Es ist damit möglich, eine haptische Requisite von maximal 10 kg in einem Volumen von $8 \times 6,3 \times 1,95$ m (Länge, Breite, Höhe) in einer beliebigen Ausrichtung mit einer Genauigkeit von +/- 0,01 m innerhalb von 28 s zu platzieren. Aufgrund der Eigenmasse des Roboters ist die gebotene Haptik solide an ihrem Aufenthaltsort verankert und erlaubt eine gewisse Standhaftigkeit gegenüber äußeren Einflüssen. Die kabellose Ausführung von *Headset* und Roboter ermöglicht einen ungehinderten Betrieb im gesamten Versuchsraum und erspart Überlegungen zum Kabelmanagement. Die offene Bauweise und Softwarearchitektur erlauben eine leichte Erweiterung des Systems und Austausch von Komponenten.

Die Analyse des Systems erlaubt eine Einschätzung, an welchen Stellen Verbesserungen des Setups möglich sind. Die Verwendung eines *Endeffektors* mit multiplen Werkzeugen oder eines haptischen Requisitenlagers könnte die fest angebrachte Requisite des aktuellen

Versuchs ersetzen. Zudem ist es denkbar, die statische Oberfläche der Requisite durch dynamische Effekte, zum Beispiel eines *Robotic Shape Display* zu erweitern.

Die Reaktionszeit, also die Zeit, die das System zum Platzieren der haptischen Requisite benötigt, könnte durch Vorberechnung des Bewegungsplanes des Roboterarms reduziert werden. Weiters ist es denkbar, die Navigation der Roboterbasis in die Pfadplanung des Arms zu integrieren, um eine getrennte und damit zusätzliche Berechnung zu vermeiden. Die Fahreigenschaften der *Mecanum-Räder* erlauben eine Abstraktion der Roboterfahrt als Bewegung auf einer zweidimensionalen Achse entlang des Fußbodens. Das Bewegungsmodell des Roboterarms würde damit um zwei zusätzliche Freiheitsgrade erweitert werden.

Der *UR-10* besitzt Sensoren in seinen Gelenken, die eine ungeplante Kraftausübung, zum Beispiel durch eine Kollision mit einem Hindernis erkennen können. Dieses Sicherheitsfeature erlaubt ihm einen Betrieb in der Nähe von Menschen, kann aber auch dazu verwendet werden, die Kraftausübung aktiv zu messen. Mit dieser Technik könnten haptische Objekte simuliert werden, bei denen je nach ihrer virtuellen Eigenmasse mehr oder weniger Kraftaufwand aufgebracht werden muss, um sie zu bewegen, zum Beispiel Türen oder Rollcontainer. Zur Anwendung dieser Inputmethode zur Steuerung des Roboterarms ist jedoch eine Softwareerweitung nötig, da der reguläre Treiber des *Robot Operating Systems* dies nicht unterstützt.

Mit diesen Verbesserungen ließe sich ein universelles Robotersystem umsetzen, das neben der statischen Bereithaltung von haptischen Objekten auch dynamische Haptik anbieten kann. Es könnte eine Fläche, die nur durch die Größe des Trackingvolumens eingeschränkt ist, mit Haptik versorgen. Die eingebauten Drucksensoren des Roboterarms überwachen die Interaktion mit Menschen und geben je nach Konfiguration nach oder leisten Widerstand. Ein austauschbares Repertoire an Requisiten sorgt für unterschiedlichste, haptische Eindrücke. Aufgrund der modularen Architektur ist eine Erweiterung auf mehrere Roboter denkbar, die im Zusammenspiel eine ganzheitliche haptische *VR*-Erfahrung bieten können.

Ein solches System ist in vielen wirtschaftlichen Branchen einsetzbar. Denkbar wäre etwa eine haptische *VR*-Trainingssimulation zur Ausbildung von Einsatzkräften. Ein Roboter könnte mit seiner Requisite eine versperrte Türe simulieren, die von den Auszubildenden vor dem weiteren Vorgehen geöffnet werden muss. Ein weiterer Roboter könnte einen PKW mimen, der auf einer abschüssigen Straße wegzurollen droht und von den Einsatzkräften in Sicherheit gebracht werden muss. Neben dem Ausbildungsbereich könnten die gleichen Elemente auch zur Unterhaltung eingesetzt werden, zum Beispiel in Virtual-Reality-Erlebniswelten, wo eine Interaktion der Spieler und Spielerinnen mit der Umgebung notwendig ist. So könnte eine schwere Metalltüre durch genügend Kraftanwendung aufgedrückt werden. Eine weitere Idee wäre eine Grubenlore, die durch ein virtuelles Level entlang der Gleise geschoben wird, zu einem späteren Zeitpunkt in einen virtuellen Grubenschacht fällt und dabei selbstständig aus dem Spielbereich navigiert.

ANHANG A

Anhang

A.1 Einrichtung von SteamVr auf dem Roboter

Dieser Abschnitt beschreibt die genauen Schritte, die nötig sind, um *SteamVr* auf einem *RB-Kairos* einzurichten. Die Anleitung geht davon aus, dass *Ubuntu 16.04* und *ROS*-Version *Kinetic Kame* installiert sind. Diese Version von *ROS* verwendet ausschließlich *Python 2.7*, welche mittlerweile ihr Supportende erreicht hat [pyt]. *SteamVr* wird so konfiguriert, dass eine Ausführung ohne *VR-Headset* möglich ist und auch keine Rechnerressourcen für eine Grafikdarstellung aufgebracht werden. Die Gründe für diese Art der Installation werden in Abschnitt 5.4 (Lighthouse Tracking des RB-Kairos) beschrieben.

Die hier aufgelisteten Installationsschritte beinhalten Anleitungen, die aus den Internetquellen [AQwm18], [Wen] und [God18] stammen. Sie wurden für die Verwendung in dieser Arbeit adaptiert und zusammengefasst.

A.1.1 Grafiktreiber

Die Internetquelle [AQwm18] zählt als notwendigen Schritt die Installation des Grafiktreibers *VulkanSDKs* auf, der jedoch für das Projekt dieser Diplomarbeit nicht ausgeführt wird. Da die Grafikkarte des *RB-Kairos*-Rechners nicht genug Leistung bietet um *VR*-Anwendungen darzustellen und diese Funktion nicht benötigt wird, wird bewusst auf die Installation des Grafiktreibers verzichtet. Daraufhin wird beim Start von *SteamVr* dem Benutzer oder der Benutzerin ein Hinweistext angezeigt, dass eine Komponente von *SteamVr* nicht initialisiert werden konnte: `A key component of SteamVR has failed` (siehe Abbildung 5.6). Dieser Fehler kann ignoriert werden, da das Tracking von ihm unberührt bleibt.

A.1.2 Installation von Steam und SteamVr

- *Steam* von `http://store.steampowered.com` herunterladen und installieren.
- *Steam* öffnen und einloggen.
- Den Menüpunkt „`Steam/Settings/Account/Beta Participation`“ öffnen und `Steam Beta Update` aktivieren.
- Unter „`Library/VR`“ auf `SteamVr` rechts-klicken, „`Properties/Beta`“ öffnen und eine Beta-Version auswählen.
- `SteamVr` auf der `Store` Seite suchen und installieren.
- Unter dem Menüpunkt „`Steam/Go Offline`“ *Steam* in den *Offline-Modus* versetzen. Dies verhindert ein Überschreiben der Einstellungen, die bei den nachfolgenden Konfigurationen gesetzt werden.

A.1.3 Konfiguration von SteamVr

- Terminal Console öffnen.
- Erfordernis eines Headsets deaktivieren. Dazu die Datei `default.vrsettings` mit einem Texteditor editieren: `gedit ~/.steam/steam/steamapps/\ common/SteamVR/resources/settings/default.vrsettings`.
- `"requireHmd" : false`, `"activateMultipleDrivers" : true` und `"forcedDriver": "null"` setzen und speichern.
- Den *HMD-Null*-Treiber per `vrsettings`-Datei aktivieren: `gedit ~/.steam/\ steam/steamapps/common/SteamVR/drivers/null/resources/\ "settings/default.vrsettings`.
- `"enable": true` setzen und speichern.
- *Steam* und *SteamVr* neu starten.

A.1.4 Kopieren der Lighthouse-Einstellungen

Da ein Ausführen des *Room-Setups* von *SteamVr* aufgrund der fehlenden Hardwareleistung auf dem *RB-Kairos* nicht möglich ist, wird diese Konfiguration auf einem tauglichen Computer ausgeführt und anschließend auf den Roboter übertragen. Bei den zu kopierenden Dateien handelt es sich um `chaperone_info.vrchap` und das Verzeichnis `lighthouse`. Diese befinden sich standardmäßig in folgenden Verzeichnissen:

Windows: `C:/Program Files (x86)/Steam/config/`

Ubuntu: `~/.local/share/Steam/config/`

A.1.5 OpenVr SDK einrichten

- Terminal Console öffnen.
- Erstellen eines Sym-Links der Datei `libudev.so.0` auf `libudev.so.1`: `sudo ln -s /lib/x86_64-linux-gnu/libudev.so.1\ /lib/x86_64-linux-gnu/libudev.so.0`
- Abhängigkeiten für *OpenVr* installieren: `sudo apt install libsdl2-dev\ libudev-dev libssl-dev zlib1g-dev python-pip`
- Eine *Python 2.7* Version von *OpenVr* installieren: `sudo pip\ install openvr==1.0.1301`

Anmerkung: Die für diese Arbeit erstellte Software, die Daten aus *OpenVr* in *ROS* überträgt, wurde in *Python 2.7* geschrieben und von [AQwm18] inspiriert. Sie verwendet einen inoffiziellen *Python*-Wrapper [pyo] für das in C++ geschriebene *OpenVr-SDK*. Nach dem Supportende von *Python 2.7* am 1. Januar 2020 [pyt] wurde der Wrapper auf *Python*-Version *3.5* aktualisiert. Nachdem aber die *ROS*-Distribution des *RB-Kairos* (*Kinetic Kame*) nur die *Python*-Version *2.7* unterstützt, muss beim Wrapper ebenso eine entsprechende Version installiert werden.

A.2 Hochfahren des RB-Kairos

1. „On-Off“-Schalter des *RB-Kairos* in „On“-Stellung bringen.
2. Schlüssel in Schlüsselloch mit Aufschrift „Muting“ stecken und nach rechts drehen.
3. „CPU“-Button drücken und das Hochfahren von Ubuntu abwarten.
4. Weitere 30 Sekunden warten, bis *ROS* gestartet ist.
5. „On Arm“-Button drücken und circa 90 Sekunden warten, bis der *UR-10* hochgefahren ist.
6. Den Drehknopf auf der Notstoppfernbedienung herausdrehen.
7. „Restart“-Button drücken, um den Notstopp zu deaktivieren.
8. Der *UR-10* macht Geräusche, während die Gelenkbremsen gelöst werden.
9. Sollte dies nicht der Fall sein, Maus und Monitor an die Buchsen des *UR-10* anschließen und den Roboterarm manuel initialisieren und starten.

A.3 Code-Auflistung

Dieser Abschnitt enthält Codefragmente, die im Rahmen dieser Diplomarbeit zum Einsatz kommen. Die verwendeten *ROS Messages* und *Actions* werden vorgefertigt vom *ROS*-Ökosystem zur Verfügung gestellt, sind zum Großteil in der Onlinedokumentation [rosa] zu finden und wurden zur besseren Lesbarkeit geringfügig aufbereitet.

A.3.1 Messages

Messages beschreiben den Inhalt von Datennachrichten, die von *ROS* über *Topics* publiziert werden und sind vergleichbar mit *structs* aus *C/C++*. Sie definieren ein oder mehrere Variablentypen und die Reihenfolge, die sie im Speicherbereich der Nachricht einnehmen. *Messages* selbst können auch Teil anderer *Messages* sein, wie zum Beispiel der Typ `geometry_msgs/Vector3` (Auflistung A.1), der ein Datentripel definiert und in vielen anderen Nachrichten vorkommt, unter anderem `geometry_msgs/Transform` (Auflistung A.4).

```
# This represents a vector in free space.
# It is only meant to represent a direction. Therefore, it does not
# make sense to apply a translation to it (e.g., when applying a
# generic rigid transformation to a Vector3, tf2 will only apply the
# rotation). If you want your data to be translatable too, use the
# geometry_msgs/Point message instead.
float64 x
float64 y
float64 z
```

Auflistung A.1: `geometry_msgs/Vector3` http://docs.ros.org/en/api/geometry_msgs/html/msg/Vector3.html

```
# This contains the position of a point in free space
float64 x
float64 y
float64 z
```

Auflistung A.2: `geometry_msgs/Point` http://docs.ros.org/en/api/geometry_msgs/html/msg/Point.html

```
# This represents an orientation in free space in quaternion form.
float64 x
float64 y
float64 z
float64 w
```

Auflistung A.3: `geometry_msgs/Quaternion` http://docs.ros.org/en/api/geometry_msgs/html/msg/Quaternion.html

```
# This represents the transform between two coordinate frames in free space.
Vector3 translation
Quaternion rotation
```

Auflistung A.4: geometry_msgs/Transform http://docs.ros.org/en/api/geometry_msgs/html/msg/Transform.html

```
# A representation of pose in free space, composed of position
# and orientation.
Point position
Quaternion orientation
```

Auflistung A.5: geometry_msgs/Pose http://docs.ros.org/en/api/geometry_msgs/html/msg/Pose.html

```
# This expresses velocity in free space broken into
# its linear and angular parts.
Vector3 linear
Vector3 angular
```

Auflistung A.6: geometry_msgs/Twist http://docs.ros.org/en/api/geometry_msgs/html/msg/Twist.html

```
# This represents force in free space, separated into
# its linear and angular parts.
Vector3 force
Vector3 torque
```

Auflistung A.7: geometry_msgs/Wrench http://docs.ros.org/en/api/geometry_msgs/html/msg/Wrench.html

```
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.
# sequence ID: consecutively increasing ID
uint32 seq
# Two-integer timestamp that is expressed as:
# * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable
# is called 'secs')
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable
# is called 'nsecs')
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

Auflistung A.8: std_msgs/Header http://docs.ros.org/en/api/std_msgs/html/msg/Header.html

```
# Reports the state of a joysticks axes and buttons.
# timestamp in the header is the time the data is received from the joystick
Header header
# the axes measurements from a joystick
float32[] axes
# the buttons measurements from a joystick
int32[] buttons
```

Auflistung A.9: sensor_msgs/Joy https://docs.ros.org/en/api/sensor_msgs/html/msg/Joy.html

```
# This expresses a transform from coordinate frame header.frame_id to the
# coordinate frame child_frame_id. This message is mostly used by the tf
# package. See its documentation for more information.
Header header
string child_frame_id
Transform transform
```

Auflistung A.10: geometry_msgs/TransformStamped http://docs.ros.org/en/api/geometry_msgs/html/msg/TransformStamped.html

```
# A Pose with reference coordinate frame and timestamp
Header header
Pose pose
```

Auflistung A.11: geometry_msgs/PoseStamped http://docs.ros.org/en/api/geometry_msgs/html/msg/PoseStamped.html

```
# This is a message to hold data from an IMU (Inertial Measurement Unit).
# Accelerations should be in m/s^2 (not in g's), and rotational velocity
# should be in rad/sec. If the covariance of the measurement is known, it
# should be filled in (if all you know is the variance of each measurement,
# e.g. from the datasheet, just put those along the diagonal).
# A covariance matrix of all zeros will be interpreted as "covariance
# unknown", and to use the data a covariance will have to be assumed or
# gotten from some other source.
# If you have no estimate for one of the data elements (e.g. your IMU doesn't
# produce an orientation estimate), please set element 0 of the associated
# covariance matrix to -1. If you are interpreting this message, please check
# for a value of -1 in the first element of each covariance matrix, and
# disregard the associated estimate.
Header header
geometry_msgs/Quaternion orientation
float64[9] orientation_covariance # Row major about x, y, z axes
geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance # Row major about x, y, z axes
geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance # Row major x, y, z
```

Auflistung A.12: sensor_msgs/Imu Message http://docs.ros.org/en/api/sensor_msgs/html/msg/Imu.html

```
# This represents a 2-D grid map, in which each cell represents the
# probability of occupancy.
Header header
#MetaData for the map
MapMetaData info
# The map data, in row-major order, starting with (0,0). Occupancy
# probabilities are in the range [0,100]. Unknown is -1.
int8[] data
```

Auflistung A.13: nav_msgs/OccupancyGrid http://docs.ros.org/en/api/nav_msgs/html/msg/OccupancyGrid.html

```
# This hold basic information about the characterists of the OccupancyGrid
time map_load_time # The time at which the map was loaded
float32 resolution # The map resolution [m/cell]
uint32 width # Map width [cells]
uint32 height # Map height [cells]
# The origin of the map [m, m, rad]. This is the real-world pose of the
# cell (0,0) in the map.
geometry_msgs/Pose origin
```

Auflistung A.14: nav_msgs/MapMetaData http://docs.ros.org/en/api/nav_msgs/html/msg/MapMetaData.html

```
# Single scan from a planar laser range-finder
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data.
# Timestamp in the header is the acquisition time of the first ray in the
# scan. In frame frame_id, angles are measured around the positive Z axis
# (counterclockwise, if Z is up) with zero angle being forward along the x
# axis
Header header
float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]
# time between measurements [seconds] - if your scanner is moving, this will
# be used in interpolating position of 3d points
float32 time_increment
float32 scan_time # time between scans [seconds]
float32 range_min # minimum range value [m]
float32 range_max # maximum range value [m]
# range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] ranges
# intensity data [device-specific units]. If your device does not provide
# intensities, please leave the array empty.
float32[] intensities
```

Auflistung A.15: sensor_msgs/LaserScan http://docs.ros.org/en/api/sensor_msgs/html/msg/LaserScan.html

A.3.2 Actions

Actions beschreiben Aufgaben, die von *ActionServern* ausgeführt werden. Sie definieren sich durch Angabe von *Goal*-, *Feedback*- und *Result-Topics*. *Goal* beschreibt den zu erreichenden Zielzustand und *Feedback* den aktuellen Zustand. Die erfolgreich beendete Ausführung wird schließlich in den *Result-Topics* veröffentlicht.

```
1 geometry_msgs/PoseStamped target_pose
2 ---
3 ---
4 geometry_msgs/PoseStamped base_position
```

Auflistung A.16: move_base_msgs/MoveBase https://github.com/ros-planning/navigation_msgs/blob/ros1/move_base_msgs/action/MoveBase.action

```
1  # Motion planning request to pass to planner
2  MotionPlanRequest request
3
4  # Planning options
5  PlanningOptions planning_options
6
7  ---
8
9  # An error code reflecting what went wrong
10 MoveItErrorCodes error_code
11
12 # The full starting state of the robot at the start of the trajectory
13 moveit_msgs/RobotState trajectory_start
14
15 # The trajectory that moved group produced for execution
16 moveit_msgs/RobotTrajectory planned_trajectory
17
18 # The trace of the trajectory recorded during execution
19 moveit_msgs/RobotTrajectory executed_trajectory
20
21 # The amount of time it took to complete the motion plan
22 float64 planning_time
23
24 ---
25
26 # The internal state that the move group action currently is in
27 string state
```

Auflistung A.17: moveit_msgs/MoveGroup http://docs.ros.org/en/api/moveit_msgs/html/action/MoveGroup.html

A.3.3 Eigener Code

```
private void Update()
{
  // Get controller events
  bool triggerChanged = CheckTriggers();

  Ray raycast = new Ray(transform.position, transform.forward);
  RaycastHit hitInfo;

  // Hit anything and reduce laser distance if we do
  bool hit = Physics.Raycast(raycast, out hitInfo, maxDistance);
  float laserDistance = maxDistance;
  if (hit)
    laserDistance = hitInfo.distance;

  // Check if we hit a 'HapticVr_Target' component
  LaserTarget target = null;
  if (hitInfo.collider != null)
    target = hitInfo.collider.GetComponent<LaserTarget>();

  if (previousTarget && previousTarget != target)
  {
    previousTarget.Hover(hitInfo, false);
    previousTarget = null;
  }

  if (hit && target != null)
  {
    if (previousTarget != target)
    {
      target.Hover(hitInfo, true);
      previousTarget = target;
    }
    // Trigger release signal
    if (triggerChanged && !triggered)
    {
      target.Activate(hitInfo);
    }
  }

  pointer.GetComponent<MeshRenderer>().material.color =
    triggered ? triggerColor : color;
  pointer.transform.localScale = new Vector3(thickness,
    thickness, laserDistance);
  pointer.transform.localPosition = new Vector3(0f, 0f,
    laserDistance / 2f);
}
```

Auflistung A.18: Die Updatefunktion des virtuellen Laserpointers in C#, der in *Unity* zur Auswahl des Haptikbereichs verwendet wird. Adaptiert von der Internetquelle [Stec].

```
# These calculations are taken from SteamVR and ROS-Sharp plugins for Unity.
# Note: Unity has a left-handed coordinate system with X and Z at the floor
# plane and Y pointing upwards, while ROS has a right handed coordinate
# system with X and Y at the floor plane and Z pointing upwards.
# (The calculations could be simplified by removing redundant sign changes.)
def openvr_matrix_to_values(m):

  x = m[0][3]
  y = m[1][3]
  z = -m[2][3]

  qx = qy = qz = 0
  qw = 1

  # rotation valid?
  if (m[0][2] != 0 or m[1][2] != 0 or m[2][2] != 0) and
     (m[0][1] != 0 or m[1][1] != 0 or m[2][1] != 0):
    qw = math.sqrt(max(0, 1 + m[0][0] + m[1][1] + m[2][2])) / 2
    qx = math.sqrt(max(0, 1 + m[0][0] - m[1][1] - m[2][2])) / 2
    qy = math.sqrt(max(0, 1 - m[0][0] + m[1][1] - m[2][2])) / 2
    qz = math.sqrt(max(0, 1 - m[0][0] - m[1][1] + m[2][2])) / 2

    qx = math.copysign(qx, -m[2][1] - -m[1][2])
    qy = math.copysign(qy, -m[0][2] - -m[2][0])
    qz = math.copysign(qz, m[1][0] - m[0][1])

  return [z, -x, y, -qz, qx, -qy, qw]
```

Auflistung A.19: Die erstellte Funktion zur Konvertierung von *OpenVr*- in *ROS*-Koordinaten, siehe Abschnitt 3.6 (Koordinatensysteme). Der Algorithmus wurde aus Fragmenten der Internetquellen [ope] und [Tra] kombiniert und in die Programmiersprache *Python* übersetzt.

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Action Eine Aufgabendefinition, die vom *ActionServer* des *Packages* `actionlib` [act] ausgeführt werden kann. Definiert werden *Goal*, *Result* und *Feedback*. 36–38, 47, 88, 92

Adaptive Monte Carlo Localization Ein Verfahren zur Positionsbestimmung in Robotiksystemen, das *Odometrie* und *Lidar*-Daten heranzieht und mit einer zuvor aufgezeichneten Karte vergleicht. Von *ROS* wird es durch das *Package* `robot_localization` [robb] zur Verfügung gestellt. 26, 35, 66, 105

Aktuator oder Aktor. Ein Bauteil, das (elektrische) Signale in mechanische Bewegung umsetzt. 12, 100

Augmented Reality Eine Computer-unterstütze Wahrnehmung der Realität. Im Unterschied zu *VR* wird die Wahrnehmung des Benutzers nicht ersetzt, sondern erweitert (englisch: augmented). 9

Base-Footprint Der Ursprungspunkt des Roboters im *Transform Tree* von *ROS*. 24, 36, 49, 53, 56, 62

Controller Ein für Videospiele konzipiertes Eingabegerät, auch Gamepad genannt. 1–3, 10, 11, 19–21, 37, 38, 48, 54, 55, 62, 71, 72, 76, 95, 96, 103

Endeffektor Das letzte Glied in einer Kette von Robotergelenken (siehe *Tool Center Point*). 6, 11, 12, 15, 24, 29, 30, 44, 45, 53, 63, 73, 75, 77–79, 83, 96, 101, 103

Exoskelett Ein Robotikanzug, der die muskulare Bewegung des Trägers unterstützt oder, im Fall der Erzeugung von Haptik, behindert. 12

Force-Feedback Eine Technologie, die haptische Rückmeldung mit Ausübung von Kräften realisiert, zum Beispiel durch Vibrationsmotoren. 3, 10, 12, 13

Frame Im Kontext dieser Arbeit handelt es sich um einen Knoten eines hierarchisch aufgebauten Koordinatennetzwerks. Ein Frame kann beliebig viele Kindknoten besitzen, aber nur einen Elternknoten. Transformationen, die an einem Elternknoten angewendet werden, werden so gleichzeitg auch an die Kindknoten vererbt. 23–26, 28, 36, 46, 50–53, 56, 58, 59, 66, 74, 95, 96, 103

Framework Siehe *Software Development Kit*. ix, 6, 19, 21, 30, 37–39, 49, 75, 83, 101, 102

Freedrive Modus Die Manipulatoren der *UR-10* Serie besitzen einen Modus, mit dem dessen Gelenke per Hand bewegt werden können. Dazu werden äußere Krafteinwirkungen auf den Arm gemessen und in Bewegungskommandos für die Gelenke umgesetzt. 78

GPS Global Positioning System. Ein globales System zur Positionsbestimmung auf der Erde anhand von Satelliten. 26

Haptic Gloves Handschuhe, in die *Aktuatoren* integriert sind zur Einschränkung der Bewegungsfreiheit der Finger. 12

Haptikwand Eine Holzplatte der Größe $0,01 \times 0,6 \times 0,6\,\mathrm{m}$, die im Zuge dieser Arbeit zur haptischen Simulation von vergleichbaren Objekten dient, beziehungsweise ihr virtuelles Gegenstück. 38–46, 52–57, 59, 61–66, 70–73, 75, 78, 96

Head Mounted Display Ein am Kopf angebrachtes Display, umschließt typischerweise das Sichtfeld. 1, 100, 105

Headset Umgangssprachliche Bezeichnung für *Head Mounted Display*. ix, 1, 2, 5, 6, 19, 20, 38, 48, 54, 71, 76, 83, 85

Inertial Measurement Unit Deutsch: Inertiale Messeinheit: ist eine Kombination mehrerer Inertialsensoren, wie Beschleunigungssensoren und Drehratensensoren zur Bestimmung von Lage und Position im Raum. 21, 105

Kinematik Die Beschreibung von Objektbewegungen anhand geometrischer Faktoren wie Zeit, Ort und Geschwindigkeit. 30, 38, 46, 52

Lidar Abkürzung von **LI**ght **D**etection **A**nd **R**anging. Ein dem Radar ähnliches Verfahren zur Abstandsmessung, das Laserstrahlen statt Radiowellen verwendet. 21, 23, 26, 28, 35, 36, 52, 66, 74, 76, 95, 99

Lighthouse Ein von der *Vive* Pro eingesetztes Trackingsystem zum Erfassen von Objekten im Raum. Lighthouse-Basisstationen senden in regulären Abständen Laserimpulse aus, die von Fotosensoren in den zugehörigen *Trackern* erfasst werden. Durch exaktes Timing und den Vergleich, welcher Sensor zu welcher Zeit welches Lasersignal empfangen hat, lässt sich seine Position in Relation zum Sender bestimmen. ix, xi, 5, 19, 20, 38, 39, 41, 42, 47–53, 56, 58, 61, 62, 65, 66, 68–71, 73, 74, 76, 83, 96, 103

Manipulator Bezeichnung für den mechanischen Teil eines Roboterarmes. 5, 19, 21, 30, 37, 41, 45, 52, 63, 64, 71, 78, 101, 103

Mecanum-Räder Eine spezielle Bauform von Rädern, die es dem Gefährt ermöglicht, seitlich zu fahren. An den Rädern sind kleinere Rollen im Winkel von 45° angebracht. 5, 21, 23, 25–28, 36, 37, 66, 75, 79, 83, 84, 95

Message Eine Typbeschreibung von Datennachrichten, die über *Topics* übertragen werden. In Abschnitt A.3 werden Messages, die in dieser Arbeit verwendet werden, gelistet. 22, 23, 35–37, 39, 49, 52, 56, 88, 102, 103

Mobiler Manipulator Ein Roboterarm (*Manipulator*), der auf einer mobilen Roboterplattform angebracht ist, zum Beispiel *RB-Kairos*. ix, 5, 6, 19–21, 29, 33, 102

MoveGroup Eine Kette an Gelenken eines *Manipulators*, die gemeinsam die Freiheitsgrade des *Endeffektors* bestimmen. Wird in *ROS* durch *MoveIt* festgelegt. 38, 46, 52, 53, 59

MoveIt Ein Motion Planning *Framework* zur Planung und Ausführung von Bewegungen durch *Manipulatoren* (`https://moveit.ros.org`). 30, 37, 38, 46, 47, 53, 62, 63, 75, 76, 79, 80, 95, 96, 101

Navigation Stack *ROS* Komponenten und *Packages*, die für die (örtliche) Lokalisation, Pfadplanung und Navigation eines mobilen Roboters zuständig sind. 26, 50, 53, 101

Node Ein Programm(-knoten) innerhalb der *ROS* Infrastruktur. 21–24, 33, 37, 43, 49, 52, 56, 58, 59, 62, 63, 76, 80, 95, 101–103

Odometrie Die Schätzung von Position und Orientierung eines mobilen Systems anhand der Daten seines Antriebs (Wegmessung). 25, 26, 35, 36, 50, 66, 70, 74, 99

Omnidirektionale Bewegungsplattform Ein Gerät, das, ähnlich einem Laufband, einem Benutzer oder einer Benutzerin Bewegungen in alle Richtungen ermöglicht, ohne das Gerät zu verlassen. (Englisch: Omnidirectional Treadmill). 14

OpenVr *Framework* für eine plattformunabhängige Entwicklung von *VR* Anwendungen, von *Valve Corporation* (`https://github.com/ValveSoftware/openvr`). 30–32, 47–49, 87, 94, 97, 102

Package Eine Kapselung von *ROS Nodes* in thematisch ähnliche Bereiche, zum Beispiel Lokalisation, Navigation (*Navigation Stack*), Motorensteuerung, Datenvisualisierung, ... 23, 35–37, 99, 101, 102

Plugin Softwareerweiterung, um eine bestehende Software mit neuen Funktionen auszustatten. Typischerweise wird dafür vom Hersteller ein *Software Development Kit* zur Verfügung gestellt. ix, 6, 28, 30, 31, 39, 52, 54, 56, 102

Polyscope Das User Interface des *UR-10*. 45, 76, 96

Publisher Ein Softwarekonstrukt innerhalb einer *ROS Node*, das *Messages* von einem bestimmten *Topic* ausliest. 21, 37, 56

Python Eine von *ROS* unterstützte Programmiersprache. Die am *RB-Kairos* installierte *ROS*-Version unterstützt Python ausschließlich in der Version 2.7 (`https://www.python.org`). Siehe auch die Ankündigung des Supportendes von Python 2.7 mit 1. Januar 2020 [pyt]. 49, 77, 85, 87, 94

Raycasting Ein Verfahren der Computergrafik, um den Schnittpunk einer Linie mit geometrischen Objekten zu bestimmen. 39, 55

RB-Kairos Der in dieser Arbeit verwendete *mobile Manipulator* (`https://robotnik.eu/products/mobile-manipulators/rb-kairos-en`). ix, xi, 5–7, 19–23, 26–29, 33, 35–43, 45–50, 52–54, 56, 58, 59, 62–67, 70, 74–77, 83, 85–87, 95, 97, 101, 102

Robot Operating System Eine Sammlung an *Frameworks* zur Entwicklung von Robotern. Verwendete Version: Kinetic Kame (http://wiki.ros.org/kinetic). ix, xi, 6, 8, 19, 21, 83, 84, 105

Robotic Shape Display Ein haptisches Display, das sich mechanischer Komponenten bedient, um Oberflächenstrukturen zu simulieren. 12, 14, 16, 77, 78, 84, 96

rosbridge Eine auf *JSON* basierende Webschnittstelle, die den Zugang zum *ROS*-Protokoll für inkompatible Systeme ermöglicht. 31, 35, 39, 43, 56, 59

roscore Der Kernknoten von *ROS*, der die Verwaltung der *Nodes* und deren Kommunikation untereinander verwaltet. 22, 33, 35, 43

RVIZ Daten-Visualisierungs-Tool von *ROS* (`http://wiki.ros.org/rviz`). 28, 43, 46, 50–52, 67, 95, 96

Simultaneous Localization And Mapping Ein Navigationsverfahren, das es einem Robotiksystem erlaubt, gleichzeitig die Karte seiner Umgebung zu erstellen und um Hindernisse herum zu navigieren. Von *ROS* wird es durch das *Package* `gmapping` [gma] zur Verfügung gestellt. 36, 105

Software Development Kit Ein Grundgerüst für die Entwicklung von Software oder *Plugins*. Synonym: *Framework*. 31, 100, 101, 105

Steam Online Vertriebsplattform für Computerspiele, Software, etc. von *Valve Corporation* Corporation, (`https://store.steampowered.com`). 31, 48, 54, 74, 86, 102, 103

SteamVr Laufzeitumgebung von *OpenVr* in *Steam*, (`https://store.steampowered.com/app/250820/SteamVR`). 30, 31, 39, 42, 47–49, 54, 58, 59, 74, 76, 85, 86, 96

Subscriber Ein Softwarekonstrukt innerhalb einer *ROS Node*, das *Messages* unter einem bestimmten *Topic* veröffentlicht. 21, 23, 52, 56, 63

Teleop Kurzform von Teleoperation. Es handelt sich dabei um die Steuerung eines Systems aus der Ferne. 23, 24, 37, 95

Tool Center Point Die Stelle an einem Roboterarm, an dem Werkzeuge (Tools) angebracht werden können. Auch *Endeffektor* genannt. 38, 99, 105

Topic Ein einer *URL* ähnlicher Datenpfad, unter dem *Messages* in *ROS* veröffentlicht werden. 21–23, 36, 37, 43, 49, 52–54, 59, 63, 78, 88, 92, 101–103

Tracker Ein Zusatzgerät der *Vive* Pro, dessen Position im Raum unter Zuhilfenahme von *Lighthouses* erfasst werden kann. 5, 6, 19, 20, 39, 41, 42, 47–50, 58, 59, 62, 66, 68, 69, 73, 74, 76, 95, 96, 100

Transform Tree Die hierarchische Verwaltung der *ROS Frames* (`http://wiki.ros.org/tf2`). 23–25, 36, 46, 50, 52, 53, 56, 74, 95, 99

Trigger Ein spezieller Button auf einem *Controller*. Er ähnelt einem Fingerabzug und liefert analoge Werte, je nach Intensität seiner Betätigung. 2, 55, 62, 63, 71

Unity Eine Laufzeit- und Entwicklungsumgebung für Spiele (Spiel-Engine) (`https://unity.com`). ix, xi, 6, 19, 30–32, 39, 54–57, 59, 62, 63, 83, 93, 96, 97

UR-10 Ein Roboterarm (*Manipulator*) der Firma Universal Robots [ur] (`https://www.universal-robots.com/de/produkte/ur10-roboter`). 5, 6, 19, 21, 23–25, 28, 29, 35, 37, 38, 42, 43, 45–47, 52, 53, 58, 59, 62–66, 70, 75–80, 83, 84, 87, 95, 96, 100, 101

URScript Die Scriptsprache der *Manipulatoren* von Universal Robots. 29, 79, 80

Valve Corporation Ein Softwareentwickler und -händler, der durch seine Online-Vertriebsplattform *Steam* bekannt ist (`https://www.valvesoftware.com`). 30, 101, 102

virtuelle Realität Eine in Echtzeit Computer-generierte, interaktive Realität beziehungsweise ihre Wahrnehmung durch einen Benutzer. ix, 1–3, 8, 30, 48, 71, 80, 83, 105

virtuelle Umgebung Die durch ein *VR*-System simulierte Welt. (Englisch: Virtual Environment). 1, 2, 6, 39, 71–73, 105

Vive HTC Vive Pro [viv]. Das in dieser Arbeit verwendete *VR*-System. ix, xi, 1, 2, 5, 6, 10, 19, 20, 33, 38, 39, 41, 42, 47, 49, 50, 52–55, 58, 59, 61–63, 66, 68, 71, 74, 75, 83, 95, 96, 100, 103

Wearables Am Körper getragene Computer oder Computerperipherie. Im Zuge dieser Arbeit werden Ganzkörperanzüge erwähnt, die haptisches Feedback oder Bewegungsdaten liefern. 11

Xacro Kurzform von *XML-Macro* und Dateiformat, das auf *Extensible Markup Language (XML)* basiert. *ROS* verwendet diese Macro-Sprache zur Beschreibung des Roboteraufbaus. Dateien dieses Typs enthalten geometrische Daten und eine hierarchische Beschreibung ihrer Relationen, zum Beispiel Gelenksdefinitionen. 45

Akronyme

AMCL Adaptive Monte Carlo Localization. 26, 35, 66, *Glossar:* Adaptive Monte Carlo Localization

HMD Head Mounted Display. 1, 5, 6, 20, 48, 54, 71, 86, *Glossar:* Head Mounted Display

IMU Inertial Measurement Unit. 21, 26, 35, 68, *Glossar:* Inertial Measurement Unit

JSON JavaScript Object Notation. 31, 35, 102

ROS Robot Operating System. ix, xi, 6, 8, 21–24, 26, 28–32, 35–41, 43, 45, 46, 49, 52, 56, 58, 59, 74, 76–80, 85, 87, 88, 94, 95, 97, 99, 101–104, *Glossar:* Robot Operating System

SDK Software Development Kit. 31, 47, 87, *Glossar:* Software Development Kit

SLAM Simultaneous Localization And Mapping. 36, 52, 64, *Glossar:* Simultaneous Localization And Mapping

TCP Tool Center Point. 38, 39, 43, 45, 79, *Glossar:* Tool Center Point

URL Uniform Resource Locator. 22, 103

VE Virtual Environment. 6, *Glossar:* virtuelle Umgebung

VR Virtuelle Realität. ix, 1–6, 8–11, 15–17, 19, 20, 30, 31, 33, 34, 38–41, 47, 48, 53, 54, 56–59, 61, 62, 65, 71–76, 80, 83–85, 95–97, 99, 101, 103, *Glossar:* virtuelle Realität

XML Extensible Markup Language. 104

Literaturverzeichnis

[act] *ROS actionlib package*. URL: http://wiki.ros.org/actionlib (aufgerufen am 25.02.2021).

[ALY+19] Parastoo Abtahi, Benoit Landry, Jackie (Junrui) Yang, Marco Pavone, Sean Follmer, and James A. Landay. Beyond the force: Using quadcopters to appropriate objects and the environment for haptics in virtual reality. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–13, New York, NY, USA, 2019. Association for Computing Machinery.

[AQwm18] Daniel Arnett, David Qiu, wtabib, and mswenson17. *Anleitung zum Betrieb von SteamVR unter Ubuntu*, 04 2018. URL: https://github.com/moon-wreckers/vive_tracker (aufgerufen am 24.08.2020).

[ASJR+19] Mohammed Al-Sada, Keren Jiang, Shubhankar Ranade, Mohammed Kalkattawi, and Tatsuo Nakajima. Hapticsnakes: multi-haptic feedback wearable robots for immersive virtual reality. *Virtual Reality*, 09 2019.

[BSC+18] Miguel Borges, Andrew Symington, Brian Coltin, Trey Smith, and Rodrigo Ventura. Htc vive: Analysis and accuracy improvement. pages 2610–2615, 10 2018.

[CRR+15] Lung-Pan Cheng, Thijs Roumen, Hannes Rantzsch, Sven Köhler, Patrick Schmidt, Robert Kovacs, Johannes Jaspers, Jonas Kemper, and Patrick Baudisch. Turkdeck: Physical virtual reality based on people prop-based virtual reality; passive virtual reality. 11 2015.

[DFBT99] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte carlo localization for mobile robots. volume 2, pages 1322 – 1328 vol.2, 02 1999.

[gma] *ROS gmapping package*. URL: http://wiki.ros.org/gmapping (aufgerufen am 25.02.2021).

[God18] Tom Goddard. *SteamVR on Linux*, 10 2018. URL: https://www.cgl.ucsf.edu/chimera/data/linux-vr-oct2018/linuxvr.html (aufgerufen am 27.08.2020).

[gta] *Grand Theft Auto V official website.* URL: https://www.rockstargames.com/V/ (aufgerufen am 12.04.2021).

[HCL+08] Kyung-Lyong Han, Oh-Kyu Choi, In Lee, Inwook Hwang, Jin Lee, and Seungmoon Choi. Design and control of omni-directional mobile robot for mobile haptic interface. pages 1290 – 1295, 11 2008.

[HW16] Anuruddha Hettiarachchi and Daniel Wigdor. Annexing reality: Enabling opportunistic use of everyday objects as tangible proxies in augmented reality. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, page 1957–1967, New York, NY, USA, 2016. Association for Computing Machinery.

[ind] *Valve Index.* URL: https://www.valvesoftware.com/en/index/headset (aufgerufen am 05.10.2020).

[Ins01] Brent Edward Insko. *Passive Haptics Significantly Enhances Virtual Environments.* PhD thesis, 2001. AAI3007820.

[iS] id Software. *DOOM VFR.* URL: https://store.steampowered.com/app/650000/DOOM_VFR/ (aufgerufen am 05.10.2020).

[KHM+16] Yukari Konishi, Nobuhisa Hanamitsu, Kouta Minamizawa, Ayahiko Sato, and Tetsuya Mizuguchi. Synesthesia suit: the full body immersive experience. pages 1–1, 07 2016.

[LPYS04] Robert W. Lindeman, Robert Page, Yasuyuki Yanagida, and John L. Sibert. Towards full-body haptic feedback: The design and deployment of a spatialized vibrotactile feedback system. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '04, page 146–149, New York, NY, USA, 2004. Association for Computing Machinery.

[Ltda] VR Electronics Ltd. *Tesla Suit.* URL: https://teslasuit.io/the-suit (aufgerufen am 28.09.2020).

[Ltdb] VR Electronics Ltd. *Tesla Suit Gloves.* URL: https://teslasuit.io/blog/teslasuit-introduces-its-brand-new-vr-gloves/ (aufgerufen am 23.10.2020).

[mova] *MoveGroupCommander.* URL: http://docs.ros.org/en/api/moveit_commander/html/classmoveit__commander_1_1move__group_1_1MoveGroupCommander.html (aufgerufen am 24.02.2021).

[movb] *ROS move_base package.* URL: http://wiki.ros.org/move_base (aufgerufen am 25.02.2021).

[NTAS15] Kazuki Nagai, Soma Tanoue, Katsuhito Akahane, and Makoto Sato. Wearable 6-dof wrist haptic device ßpidar-w". In *SIGGRAPH Asia 2015 Haptic Media And Contents Design*, SA '15, New York, NY, USA, 2015. Association for Computing Machinery.

[ope] *openvr_api.cs*. URL: `https://github.com/ValveSoftware/steamvr_unity_plugin/blob/29aa1ebe985f69a5dd455a02e52c13c85a6b5240/Assets/SteamVR/Plugins/openvr_api.cs#L5017` (aufgerufen am 27.02.2021).

[pix] *Pixhawk 4 IMU*. URL: `https://docs.px4.io/master/en/flight_controller/pixhawk4.html` (aufgerufen am 25.02.2021).

[pla] *PlayStation*. URL: `https://www.playstation.com/` (aufgerufen am 14.11.2020).

[PVP18] Jérôme Perret and Emmanuel Vander Poorten. Touching virtual reality: a review of haptic gloves. 06 2018.

[PVS+16] Iana Podkosova, Khrystyna Vasylevska, Christian Schönauer, Emanuel Vonach, Peter Fikar, Elisabeth Bronederk, and Hannes Kaufmann. Immersivedeck: a large-scale wireless vr system for multiple users. In *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 1–7, 03 2016.

[pyo] *Python OpenVr Wrapper*. URL: `https://github.com/cmbruns/pyopenvr` (aufgerufen am 20.11.2020).

[pyt] *Python 2 Sunset*. URL: `https://www.python.org/doc/sunset-python-2/` (aufgerufen am 20.11.2020).

[rif] *Oculus Rift*. URL: `https://www.oculus.com/rift/` (aufgerufen am 05.10.2020).

[roba] *Robotnik*. URL: `https://robotnik.eu/` (aufgerufen am 14.11.2020).

[robb] *ROS robot_localization package*. URL: `http://wiki.ros.org/robot_localization` (aufgerufen am 25.02.2021).

[rosa] *Robot Operating System Dokumentation*. URL: `http://docs.ros.org` (aufgerufen am 24.02.2021).

[rosb] *ROS Sharp Plugin*. URL: `https://assetstore.unity.com/packages/tools/physics/ros-107085` (aufgerufen am 16.11.2020).

[see] *Mixed Reality Training für Einsatzkräfte mit einem Video See-Through Head-Mounted Display*. URL: `https://publik.tuwien.ac.at/files/publik_283461.pdf` (aufgerufen am 12.12.2020).

[SGY+17] Alexa Siu, Eric Gonzalez, Shenli Yuan, Jason Ginsberg, Allen Zhao, and Sean Follmer. shapeshift: A mobile tabletop shape display for tangible and haptic interaction. pages 77–79, 10 2017.

[SHZ+20] Ryo Suzuki, Hooman Hedayati, Clement Zheng, James Bohn, Daniel Szafir, Ellen Do, Mark Gross, and Daniel Leithinger. Roomshift: Room-scale dynamic haptics for vr with furniture-moving swarm robots. pages 1–11, 04 2020.

[sic] *SICK S300 Lidar*. URL: https://www.sick.com/at/de/optoelektronische-schutzeinrichtungen/sicherheits-laserscanner/s300-standard/c/g187239 (aufgerufen am 25.02.2021).

[stea] *SteamVR for Enterprise / Government Use*. URL: https://partner.steamgames.com/doc/features/steamvr/enterprise (aufgerufen am 09.01.2021).

[steb] *SteamVR Plugin*. URL: https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647 (aufgerufen am 16.11.2020).

[Stec] *SteamVR_LaserPointer.cs*. URL: https://github.com/wacki/Unity-VRInputModule/blob/master/Assets/SteamVR/Extras/SteamVR_LaserPointer.cs (aufgerufen am 27.02.2021).

[tel] *teleop.launch*. URL: https://github.com/ros-teleop/teleop_twist_joy/blob/indigo-devel/launch/teleop.launch (aufgerufen am 28.02.2021).

[TLC+19] Shan-Yuan Teng, Cheng-Lung Lin, Chi-huan Chiang, Tzu-Sheng Kuo, Liwei Chan, Da-Yuan Huang, and Bing-Yu Chen. Tilepop: Tile-type pop-up prop for virtual reality. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, page 639–649, New York, NY, USA, 2019. Association for Computing Machinery.

[Tra] *TransformExtensions.cs*. URL: https://github.com/siemens/ros-sharp/blob/f3f6b542392ed7d44342f2070c3c2890b0c5944c/Unity3D/Assets/RosSharp/Scripts/Extensions/TransformExtensions.cs#L91 (aufgerufen am 27.02.2021).

[twi] *ROS twist_mux package*. URL: http://wiki.ros.org/twist_mux (aufgerufen am 25.02.2021).

[unia] *Unity*. URL: https://unity.com/de (aufgerufen am 14.11.2020).

[unib] *Unity Asset Store*. URL: https://assetstore.unity.com/ (aufgerufen am 14.11.2020).

[ur] *Universal Robots*. URL: https://www.universal-robots.com/ (aufgerufen am 14.11.2020).

[ur1] *UR10 Technical specifications*. URL: https://s3-eu-west-1.amazonaws.com/ur-support-site/50380/UR10_User_Manual_en_Global.pdf (aufgerufen am 12.01.2021).

[VGK17] Emanuel Vonach, Clemens Gatterer, and Hannes Kaufmann. Vrrobot: Robot actuated props in an infinite virtual environment. In *Proceedings of IEEE Virtual Reality 2017*, pages 74–83. IEEE, 2017. talk: IEEE Virtual Reality 2017, Los Angeles, CA, USA; 2017-03-18 – 2017-03-22.

[vie] *view_frames*. URL: http://wiki.ros.org/tf#view_frames (aufgerufen am 12.04.2021).

[viv] *HTC Vive*. URL: https://www.vive.com/ (aufgerufen am 05.10.2020).

[Wen] Reid Wender. *SteamVR Tracking without an HMD*. URL: http://help.triadsemi.com/en/articles/836917-steamvr-tracking-without-an-hmd (aufgerufen am 24.08.2020).