DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Pepper VR – Teleoperation eines humanoiden Roboter auf Basis der Analyse menschlicher Bewegung

Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik / Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Matthias Schuhmacher und Marlene Rieder

Abgabedatum 20. Mai 2024

Bearbeitungszeitraum	300 Stunden
Matrikelnummer	4128647 und 8261867
Kurs	tinf21b3 und tinf21b5
Gutachter der Studienakademie	Prof. Dr. Marcus Strand

Erklärung

Wir versicheren hiermit, dass wir unsere Studienarbeitmit dem Thema: »Pepper VR – Teleoperation eines humanoiden Roboter auf Basis der Analyse menschlicher Bewegung« selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versicheren zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort Datum Unterschrift

Zusammenfassung

Das Ziel der vorliegenden Studienarbeit ist es, eine mögliche Verbindung zwischen einer Virtual-Reality Brille und dem humanoiden Roboter Pepper herzustellen. Das Kamerabild des Roboters soll auf der Brille angezeigt werden, ebenfalls soll es möglich sein, den Roboter mit Hilfe der Controller der Brille zu steuern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

Formelverzeichnis

Abkürzungsverzeichnis

Kapitel 1

Einleitung

Im Umfeld der Duale Hochschule Baden-Württemberg (DHBW) Karlsruhe werden immer wieder Projekte im Bereich der Robotik an Studierende im Rahmen verschiedener Arbeiten vergeben. Ebenso dieses Projekt, welches im Rahmen der Studienarbeit an die beiden Studierenden Matthias Schuhmacher und Marlene Rieder übergeben wurde.

Im Umfeld der DHBW Karlsruhe werden immer wieder Projekte im Bereich der Robotik an Studierende im Rahmen verschiedener Arbeiten vergeben. Ebenso dieses Projekt, welches im Rahmen der Studienarbeit an die beiden Studierenden Matthias Schuhmacher und Marlene Rieder übergeben wurde.

In unserer immer digitaler werdenden Welt spiele sowohl Robotik als auch Virtual Reality (VR) eine immer größer werdende Rolle, diese zwei Welten sollen durch das, im folgenden beschriebene Projekt verbunden werden.

Ziel dieses Projektes ist es den humanoiden Roboter Pepper mit Hilfe der VR Brille Meta Quest 3 zu steuern. Dies soll ermöglicht werden, indem das Kamerabild des Roboters an die Brille übertragen wird und sich die Bewegungen des Roboters durch die Controller der Brille steuern lassen. Der Nutzer soll also sehen, was sich vor dem Roboter befindet und ihn dann steuern können.

1.1 Motivation

Die Teleoperation humanoider Roboter bietet eine Vielzahl von Vorteilen. Durch die Fernsteuerung können Roboter in gefährlichen oder unzugänglichen Umgebungen eingesetzt

werden, was die Sicherheit der menschlichen Bediener erhöht. Außerdem ermöglicht die Verwendung von VR-Technologie eine intuitivere und natürlichere Steuerung der Roboter, da die Bediener das Gefühl haben, direkt vor Ort zu sein und in Echtzeit mit der Umgebung zu interagieren.

1.2 Ziele und Fragestellungen

Das Hauptziel dieser Arbeit ist es, eine funktionsfähige Verbindung zwischen der MetaQuest 3 VR-Brille und dem humanoiden Roboter Pepper herzustellen. Dabei sollen folgende Fragestellungen untersucht werden:

- Wie kann die Datenübertragung zwischen der VR-Brille und dem Roboter optimiert werden, um eine latenzfreie Steuerung zu ermöglichen?
- Welche Herausforderungen ergeben sich bei der Integration von Unity und ROS zur Steuerung des Roboters?
- Welche technischen und ergonomischen Anforderungen müssen bei der Entwicklung der VR-Steuerung berücksichtigt werden?

1.3 Methodik

Zur Beantwortung dieser Fragestellungen wird ein methodischer Ansatz verfolgt, der sowohl theoretische als auch praktische Aspekte umfasst. Zunächst wird eine umfassende Literaturrecherche durchgeführt, um den aktuellen Stand der Technik und die verfügbaren Technologien zu analysieren. Anschließend wird ein Prototyp entwickelt, der die Verbindung zwischen der VR-Brille und dem Roboter herstellt. Dieser Prototyp wird in mehreren Iterationen getestet und optimiert, um die bestmögliche Leistung zu erzielen.

1.4 Struktur der Arbeit

Diese Arbeit gliedert sich wie folgt:

- Kapitel 2 beschreibt die theoretischen Grundlagen und den aktuellen Stand der Technik.
- Kapitel 3 erläutert die technischen Spezifikationen der verwendeten Hardware und Software.

- Kapitel 4 stellt den Entwicklungsprozess und die Implementierung des Prototyps dar.
- Kapitel 5 präsentiert die Testergebnisse und Diskussion.
- Kapitel 6 gibt eine Zusammenfassung und Ausblick auf zukünftige Arbeiten.

Kapitel 2

Grundlagen

2.1 Publish and Subscribe

Bevor wir uns mit den Technologien und Herangehensweisen des Pepper Roboters beziehungsweise dessen Betriebssystem beschäftigen, müssen wir uns mit dem Publish-Subscribe-Modell auseinandersetzen.
Das Publish-Subscribe-Modell ist ein Paradigma für einen effektiven Nachrichtenaustausch in verteilten Systemen. Erstmals publiziert in einem Paper der Association for Computing Machinery (ACM) von 1987[Wikipedia o. D.], ermöglicht es die flexible Kommunikation zwischen verschiedenen Komponenten, indem es einen Mechanismus bereitstellt, über den Nachrichten von einem Sender, dem Publisher, an einen oder mehrere Empfänger, den Subscribern, verteilt werden können. Ein zentrales Element dieses Modells ist dabei Nachrichtenbroker oftmals auch nur als Borker referenziert, der als Vermittler zwischen Publishern und Subscribenden fungiert. Dieses Verhalten mit vielen Komponenten auf einige Topics, ist in Abbildung 2.1 zur Veranschaulichung abgebildet.

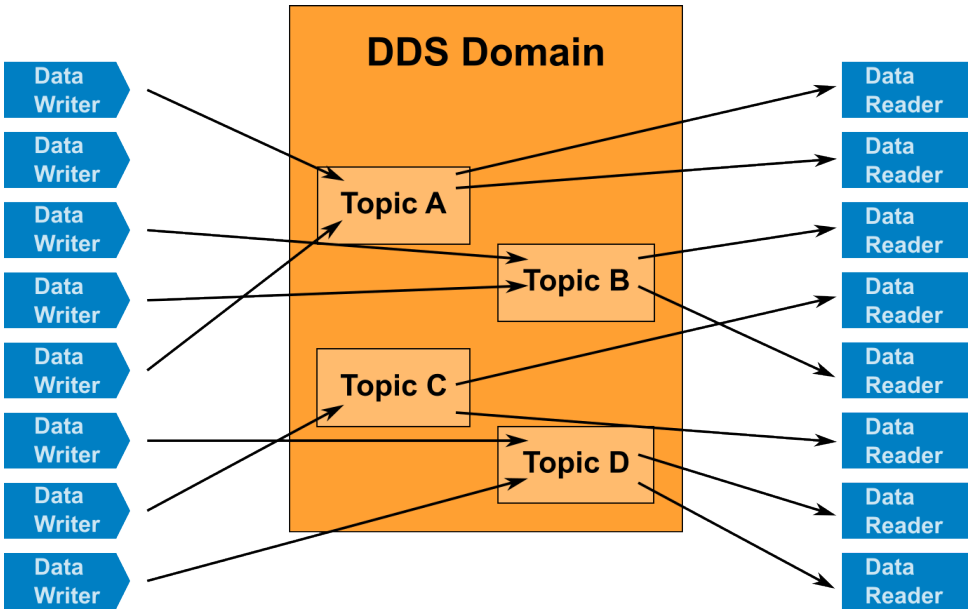

Abbildung 2.1: Publish and Subscribe

Der Broker empfängt Nachrichten vom Publisher und organisiert sie in verschiedene Kategorien, welche als Topics bezeichnet werden. Diese Topics dienen als thematische Selektionsmerkmale, die den Inhalt der Nachrichten beschreiben. Sie können in den verschiedensten Datentypen erfolgen, einfache Ganzzahlen, Texte bis hin zu Bildern oder weitaus komplexeren Datenstrukturen. Durch diese Kategorisierung in den Topics, wird eine gezielte Auswahl und Weiterleitung der Nachrichten ermöglicht.
Subscriber können bestimmte Topics abonnieren, die ihren Gewünschten entsprechen. Dadurch erhalten sie nur die Nachrichten, die zu den von ihnen gewählten Themen gehören. Dieser Prozess der Nachrichtenauswahl und -verarbeitung wird als Filterung bezeichnet und kann auf zwei Arten erfolgen: themenbasiert und inhaltsbasiert.
Im themenbasierten Ansatz erhalten Abonnenten alle Nachrichten zu den Topics, welche sie abonniert haben. Der Publisher definiert die verfügbaren Topics, aus welchen die Abonnenten selbst wählen können. Im inhaltsbasierten Ansatz dagegen, werden Nachrichten nur an die Abonnenten weitergeleitet, wenn sie den vom dessen festgelegten Kriterien entsprechen. Dabei ist der Abonnent für die Spezifikation dieser Kriterien verantwortlich[ITWissen.info o. D.]
Das Publish-Subscribe-Modell bietetdurch seine Herangehensweise eine hohe Flexibilität und Skalierbarkeit, und wird daher in verschiedenen Anwendungsbereichen eingesetzt. Es ermöglicht eine Entkopplung von Nachrichtenerzeugung und -verarbeitung, was gerade in der Entwicklung verteilter Systeme Prozesse erleichtert. Die vermutlich bekannteste

Implementierung des Modell ist das MQTT-Protokoll, welches konzipiert wurde um Telemetriedeaten zwischen Sensoren und Servern zu übertragen speziell in unzuverlässigen Umgebungen[Elektronik-Kompendium.de o. D.]
Weiter findet das Publish-Subscribe-Modell Anwendung in einer Vielzahl von weiteren Systemen und Technologien. Beispiele dafür sind der Java Message Service (JMS), der Data Distribution Service (DDS) oder wie im Fall diese Projektes die Message Oriented Middleware (MOM) zu denen auch Robot Operating System (ROS) gehört.

2.2 Pepper

Pepper ist ein humanoider Roboter, der entwickelt wurde, um die Gefühle und Gesten von Menschen zu analysieren und basierend auf diesen, darauf zu reagieren. Das Projekt entstand durch eine Zusammenarbeit des französischen Unternehmens Aldebaran Robotics Société par actions simplifiée (SAS) und des japanischen Telekommunikations- und Medienkonzerns SoftBank Mobile Corp. Ziel diese Projektes war es, einen humanuiden "Roboter-Gefährten" oder einen "persönlichen Roboter-Freund" zu schaffen, der zunächst im Gewerbesektor in Verkaufsräumen, an Empfangstischen oder in Bildungs- und Gesundheitseinrichtungen eingesetzt werden sollte. Die Produktion wurde jedoch aufgrund geringer Nachfrage bis auf Weiteres pausiert.
Das Konzept von Pepper distanziert sich von herkömmlichen Industrierobotern und reinen Spielzeugrobotern, indem er als informativer und kommunikativer Begleiter konzipiert wurde. Sein Aussehen, das im etwa an die Größe eines Kindes angelehnt ist, sowie ein freundliches Gesicht und eine kindliche Stimme sind im ästhetischen Konzept von "kawaii" (japanisch für "niedlich" oder auch "liebenswert") gehalten.
Pepper wurde im Rahmen einer Präsentation am 5. Juni 2014 als der "erste persönliche Roboter der Welt mit Emotionen" vorgestellt. Die Vermarktung begann damit, dass SoftBank Pepper-Geräte in ihren Verkaufsräumen einsetzte, um Kunden zu unterhalten und zu informieren. Die Roboter sollte dabei den Umgang mit Kunden erlernen, um zukünftige Anwendungsmöglichkeiten zu erforschen. Verkauft wurde offiziell ab dem 3. Juli 2015 zu einem Preis von 198.000 Yen pro Einheit, zuzüglich monatlicher Gebühren für Zusatzleistungen. Im Laufe der Zeit wurde Pepper auch für den Einsatz in weiteren Unternehmen und Einrichtungen verfügbar gemacht.
Pepper wird mit einer Grundausstattung an Anwendungen geliefert, jedoch sind für spezifische Anwendungen, individuell entwickelte Softwarelösungen erforderlich wie auch

zum Beispiel in diesem. SoftBank ermöglichte unabhängigen Entwicklern durch die Veröffentlichung der Schnittstellen den Zugang zu einem Interface für Applikationsprogramme, um zusätzliche Anwendungen für Pepper zu erstellen. Das NAOqi-Framework welches für diesen Nutzen bereitgestellt wurde, beinhaltet eine Application Programming Interface (API), eine Software Development Kit (SDK) und weitere Tools, welche in den Sprachen Python und C++ uneingeschränkten Zugriff auf die Komponenten, Sensoren und Aktoren des Roboters bieten, dazu später Ausführlicheres in Abschnitt 2.3. Mit Hilfe diese Interfaces haben verschiedene Unternehmen integrierte Lösungen entwickelt, die Pepper beispielsweise bei der Kundenberatung unterstützen können.
Das Design von Pepper ist dem Menschen ähnlich und umfasst einen Kopf mit integrierten Mikrofonen und Kameras sowie einen Torso mit weiteren Sensoren für Stabilität und Sicherheit. Der Roboter verfügt über verschiedene Mechaniken, die es ihm ermöglichen, sich flüssig zu bewegen und mit Personen zu interagieren. Durch die Verwendung von Kameras und bereitgestellter Software ist Pepper in der Lage, Emotionen bei seinen Gesprächspartnern zu erkennen und darauf zu reagieren, obwohl er selbst keine Mimik besitzt. Sicherheitsvorkehrungen wie Abstandssensoren und Stabilisatoren gewährleisten einen sicheren Einsatz von Pepper in verschiedenen Umgebungen. Diese können jedoch bedingt durch den Entwickler deaktiviert werden, um den Roboter in komplexeren oder laborähnlichen Umgebungen zu betreiben.

2.3 NAOqi

Im folgenden Abschnitt wird das NAOqi-Framework, welches auf dem Pepper Roboter läuft, genauer erläutert.

2.3.1 Was ist NAOqi?

NAOqi ist die Bezeichnung für die Hauptsoftware, die auf dem Pepper Roboter ausgeführt wird und ihn intern steuert. Diese kann mit persönlichen Modulen weiterentwickelt und angepasst werden. Dazu können mit Hilfe der Aldebaran SDK auch NQOqi-SDK genannt, eigene Module oder Bibliotheken entwickelt werden. Diese NAOqi-SDK ist die Basis des NAOqi Frameworks und ist in C++ implementiert[University Year of access].
Das NAOqi Framework ist das Programmiergerüst, welches zur Programmierung von

NAO und Pepper Robotern verwendet wird. Es implementiert alle allgemeinen Anforderungen der Robotik, einschließlich: Parallelität, Ressourcen-Management, Synchronisation und Ereignisse. Dieses Framework ermöglicht eine homogene Kommunikation zwischen verschiedenen Modulen wie etwa die Bewegung, Audio oder Video sowie eine homogene Programmierung und einen homogenen Informationsaustausch. Das Framework ist:

- plattformübergreifend, wie bereits erwähnt, basiert die NAOqi-SDK auf C++ und bietet damit die Möglichkeit auf Windows, Linux oder sogar auch Mac zu entwickeln.
- sprachübergreifend, mit einer identischen API für C++ und Python. Weitere Details dazu sind in Abschnitt 2.3.1 aufgeführt.
- fähig auf Introspektion, was bedeutet, dass das Framework weiß, welche Funktionen in den verschiedenen Modulen verfügbar sind und wo. Für Details diesbezüglich siehe Abschnitt 2.3.1.

Sprachübergreifend

Software kann in C++ und Python entwickelt werden. Eine Übersicht über die Sprachen selbst in den Abschnitten Unterabschnitt 2.7.1 und Unterabschnitt 2.7.2. In allen Fällen sind die Programmiermethoden genau die gleichen, alle vorhandenen APIs können unabhängig von den unterstützten Sprachen aufgerufen werden:

- Wird ein neues C++-Modul erstellt, können die C++-API-Funktionen von überall aus aufgerufen werden,
- Sind sie richtig definiert, können auch die API-Funktionen eines Python-Moduls von überall aus aufgerufen werden.

Normalerweise werden die Verhaltensweisen in Python und Ihre Dienste in C++ entwickelt.

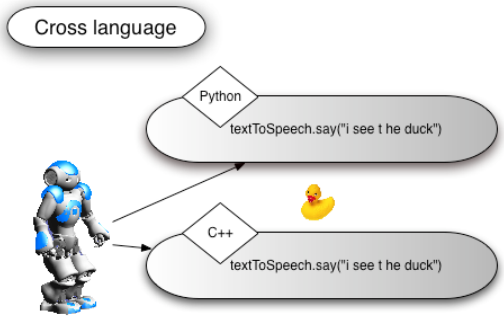

Abbildung 2.2: NAOqi Framework Sprachübergreifend

Introspektion

Die Introspektion ist die Grundlage der Roboter-API, der Fähigkeiten, der Überwachung und der Maßnahmen bei überwachten Funktionen. Der Roboter selbst kennt alle verfügbaren API-Funktionen. Wird eine Bibliothek entladen, werden die entsprechenden API-Funktionen automatisch ebenfalls entfernt. Eine in einem Modul definierte Funktion kann der API mit einem `BIND_METHOD` hinzugefügt werden.
Wird eine Funktion gebunden, werden automatisch folgende Funktionen ausgeführt:

- Funktionsaufruf in C++ und Python, wie in Abschnitt 2.3.1 beschrieben
- Erkennen der Funktion, wenn sie gerade ausgeführt wird
- Funktion lokal oder aus der Ferne, z.B. von einem Computer oder einem anderen Roboter, ausführen weiter im Detail beschrieben in Abschnitt 2.3.1
- Generierung und Aufruf von `wait`, `stop`, `isRunning` in Funktionen

Die API wird im Webbrowser angezeigt wenn auf das Gerät per Uniform Resource Locator (URL) oder Internet Protocol (IP)-Addresse auf dem Port 9559 zugegriffen wird. In dieser Übersicht, zeigt der Roboter seine Modulliste, Methodenliste, Methodenparameter, Beschreibungen und Beispiele an. Der Browser zeigt auch parallele Methoden an, die überwacht, zum Warten veranlasst und gestoppt werden können.
Die Introspektion und derer Implementation im NAOqi-Framework, ist also ein nützliches Werkzeug, welches es ermöglicht, die Roboter-API zu verstehen und zu verwenden aber auch zu überwachenund zu steuern.

Abbildung 2.3: NAOqi Framework Introspektion

Verteilter Baum und Kommunikation

Eine Echtzeitanwendung kann aus einer einzelnen ausführbaren Datei oder einem Baum von mehreren Systemen wie etwa Robotern, Prozessen oder Modulen bestehen. Unabhängig davon sind die Aufrufmethoden immer dieselben. Eine ausführbare Datei kann durch eine Verbindung mit einem anderen Roboter mit IP-Adresse und Port verbunden werden, sodass alle API-Methoden von anderen ausführbaren Dateien sind auf die gleiche Weise verfügbar sind, genau wie bei einer lokalen Methode. NAOqi trifft dabei selbst die Wahl zwischen schnellem Direktaufruf Local Procedure Call (LPC) und Fernaufruf Remote Procedure Call (RPC).

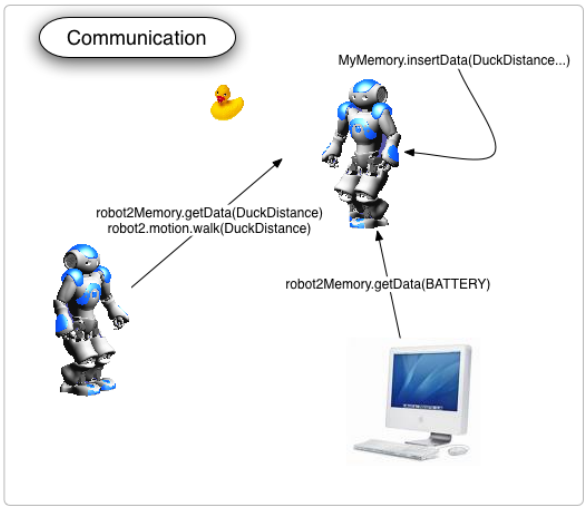

Abbildung 2.4: NAOqi Framework Kommunikation

2.3.2 NAOqi Vorgehensweise

Die NAOqi Software, welche auf dem Roboter läuft, ist ein Broker. Wenn dieser startet, lädt er eine Voreinstellungsdatei in den Speicher, in der festgelegt ist, welche Bibliotheken in dieser Konfiguration geladen werden sollen. Jede Bibliothek enthält ein oder mehrere Module, die den Broker benutzen, um ihre Methoden bereitzustellen.

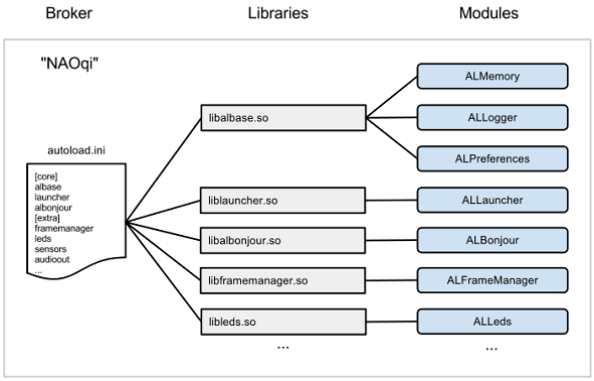

Abbildung 2.5: NAOqi Broker Bibliotheken Module

Der Broker selbst bietet Nachschlagdienste an, so dass jedes Modul im Baum oder im Netzwerk jede Methode finden kann, die an dem Broker bekannt gegeben wurde.
Das Laden von Modulen bildet dann einen Baum von Methoden, die mit Modulen verbunden sind, und von Modulen, welche mit dem Broker verbunden sind.

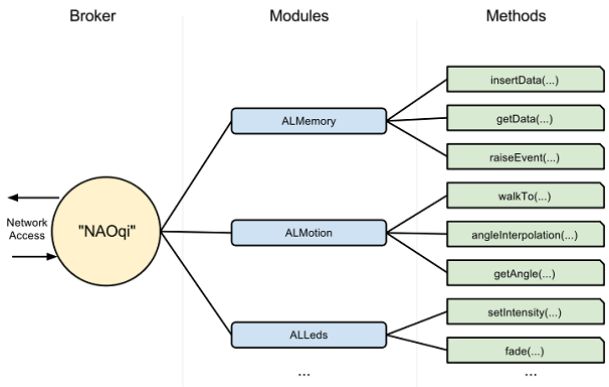

Abbildung 2.6: NAOqi Broker Modul Methoden

NAOqi Proxy

Ein weiterer wichtiger Bestandteil des NAOqi Broker ist der Proxy welcher grundlegend die Aufgabe eines Modules, näher beschrieben im folgenden Unterabschnitt 2.3.3, repräsentiert. Dies kann lokal oder entfernt enstehen der einzige Unterschied dabei ist, dass bei einer entfernten Refernz die IP des entfernten Broker mit angegeben werden muss, gleich bleibt aber im internen, dass der Proxy die Methoden des Moduls an den Broker weiterleitet.

2.3.3 NAOqi Module

Module sind die Grundbausteine der personalisierten Gestaltung von NAOqi. In ihnen kann der Nutzer eigene funktionaitäten implementieren und diese dem Broker bereitstellen. Ein Modul kann dabei aus einer oder mehreren Klassen bestehen, die wiederum Methoden enthalten.
Standardmäßig ist jedes Modul eine Klasse innerhalb einer Bibliothek, welche über eine `autoload.ini`-Datei geladen wird, worauf die Modulklasse automatisch instanziiert wird. Weiter können auch Module von übergeordneten Module abgeleitet werden, ähnlich wie etwa bei der Vererbung in der objektorientierten Programmierung. Wird eine Klasse voneiner anderen abgeleitet, können die Methoden gebunden werden, wodurch ihre Namen und MethodenSignaturen direkt dem Broker bekannt gemacht werden.
Weiter können, wie bereits erwähnt, Module sowohl lokal als auch entfernt implementiert werden. Ist es ein solches entferntes Modul, wird es als ausführbare Datei kompiliert, und kann auch außerhalb des Roboters ausgeführt werden.
Unabhängig jedoch von der Lagerung der Module, enthält jedes Modul eine Bandbreite von Methoden, welche wie im Abschnitt 2.3.1 beschrieben, gebunden werden können und damit nach außen hin nutzbar gemacht werden. Es ist also unabhängig der Lagerung der Module möglich, diese in der gleichen Weise aufzurufen. Module passen sich also automatisch selbst an.

Entfernte Module

Entfernte Module sind Module, die über das Netzwerk kommunizieren, dies kann auf demselben Robotter sein, also auch auf einem anderen Gerät im Netzwerk. Ein entferntes Modul braucht einen entfernten Broker, um mit anderen Modulen zu kommunizieren, da es selbst keinen eigenen besitzt. Broker sind dann für den gesamten Netzwerkteil

verantwortlich. Über diesen Umweg über das Netzwerk können keine schnellen Zugriffe über Remote-Module durchgeführt werden wie etwa direkte Speicherzugriffe.
Diese entfernten Module sind trivialer in der Handhabung, da sie außerhalb des Systems entwickelt und debugged werden. Dagegen sind sie aber in der Laufzeit selbst deutlich schwächer in den Punkten Geschwindigkeit und Speichernutzung, gegenüber iherer lokalen Gegenstücke. Diese Funktionalität der entfernten entwicklung spielt speziell in der Implementierung dieses Projektes eine grundlegende Rolle.

Lokale Module

Lokale Module werden als Bibliotheken kompiliert und nur auf dem Roboter verwendet. Dadurch sind sie schneller und effizienter in der Laufzeit als auch in der Speicherverwaltung, dagegen ist die Entwicklung und das Debugging auf dem Roboter selbst deutlich aufwendiger.
Sie bestehen aus zwei oder mehr Module, welche auf dem Roboter im selben Prozess gestartet werden. Sie kommunizieren miteinander über denselben Broker.
Da sich lokale Module im selben Prozess befinden, können diese Variablen gemeinsam nutzen und die Methoden des jeweils anderen ohne Serialisierung oder Vernetzung aufrufen. Dies ermöglicht die schnellstmögliche und maximal effiziente Kommunikation zwischen den Modulen.

2.3.4 NAOqi Speicherverwaltung

Die Speicherverwaltung in NAOqi wird von dem Modul `ALMemory` übernommen. Dieses Modul ist ein Speicher welcher sowohl Daten als auch Ereignisse beinhaltet. Dieser Speicher wird von allen Modulen geteilt und ermöglicht es, Daten zwischen den Modulen auszutauschen. Zusätzlich werden in diesem Speicher auch alle Ereignisse zwischengespeichert, auf welchen dann die abonierten Module zugreifen können, sobald das Ereigniss ausgelöst wurde. Alle Module haben auf diesen Bereich sowohl Lese- als auch Schreibzugriff. Jedoch ist das `ALMemory` Modul kein Echtzeitsynchronisationstool. Abonieren auf Bewegungsdatendaten oder Echtzeitvariablen in diesem Speicher ist also risikobehaftet.
Das Modul selbst ist ein Array aus sogenannten `ALValue`-Objekten, auf welche threadsicher zugegriffen werden kann. In diese Speicherbausteine können alle gängigen Datentypen gelegt werden. Dazu zählen einfache Datentypen wie einzelne Bits, Ganzzahlen, Fließkom-

mazahlen und Zeichenketten, aber auch komplexere Datenstrukturen wie Arrays, Matrizen und Binaries.
Standardmäßig wird zwischen drei verschiedenen Datentypen unterschieden:

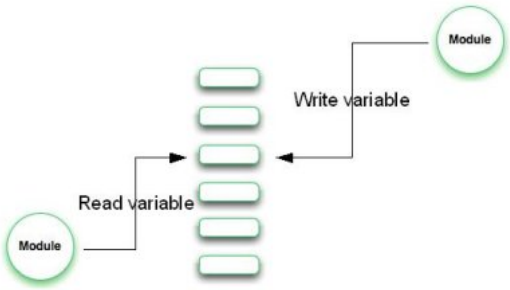

Abbildung 2.7: NAOqi Framework Speicherverwaltung

- Daten von Sensoren und Gelenken: diese Daten werden vom Roboter selbst in den Speicher geschrieben und haben keine Historie. Sie sind nicht größer als 32-Bit und können direkt per Pointer aufgerufen werden.
- Ereignisse: auf welche aboniert werden muss. Sobald eines dieser Ereignisse ausgelöst wird, wird es im Speicher gespeichert und kann von den abonierten Modulen abgerufen werden. Im Gegensatz zu den Daten von Sensoren und Gelenken sowie den Micro-Ereignissen, haben diese Ereignisse eine Historie und können auch nach dem Auslösen noch abgerufen werden.
- Micro-Ereignisse: Diese Ereignisse sind sehr kurzlebig und werden sofort nach dem Auslösen wieder gelöscht. Sie werden nicht im Speicher gespeichert und haben daher keine Historie.

2.4 Middleware

Middleware ist eine Software, die zwischen Anwendungen und Betriebssystemen vermittelt. Sie ermöglicht die Kommunikation zwischen verschiedenen Anwendungen, die auf unterschiedlichen Plattformen oder in unterschiedlichen Netzwerken ausgeführt werden. Middleware bietet eine Reihe von Diensten, die die Verbindung von Anwendungen erleichtern, die eigentlich nicht dafür vorgesehen sind. Dadurch, sowie durch Bereitstellung von weiteren Diensten wie etwa Sicherheit, Skalierbarkeit und Zuverlässigkeit, rationalisiert

Middleware die Entwicklung, Bereitstellung und Wartung von Anwendungen.
Middleware existiert in verschiedenen Formen, wie etwa Nachrichtenbroker oder Transaktionsverarbeitungsmonitore, welche jeweils auf spezifische Kommunikationsformen zugeschnitten sind. Andere, wie etwa Webanwendungsserver oder Middleware für mobile Geräte, bieten ein breites Spektrum an Kommunikations- und Konnektivitätsfunktionen, die für die Entwicklung bestimmter Anwendungen im Mobilebereich unabdinglich sind. Wieder andere, wie etwa der Enterprise Service Bus (ESB), fungiert als zentraler Integrationsknotenpunkt, der alle Komponenten in einem Unternehmen miteinander verbindet. Zusätzlich können aber auch persönliche Middleware selbst entwickelt werden, für personalisierte Anwendungen.
Der Begriff "Middleware" wurde geprägt, da die erste Generation dieser oft als "Vermittler" zwischen einem Anwendungs-Frontend zum Beispiel eines Clients und einer Backend-Ressource, in welcher Daten gespeichert oder verarbeitet werden, fungierte. Dabei kann, auch wie in Abbildung 2.8 gezeigt, das Betriebsystem das Backend sein und eine Anwendung das Frontend.

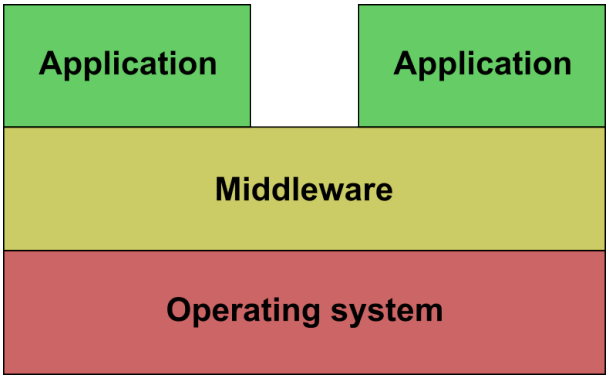

Abbildung 2.8: Middleware

Die moderne Middleware heutzutage beinhaltet jedoch durchaus einen weitaus breiteren Aufgabenbereich. Beispielsweise umfasst Portal-Middleware sowohl das Frontend der Anwendung als auch Tools für die Backend-Konnektivität, während Datenbank-Middleware in der Regel einen eigenen Datenspeicher enthält [Microsoft Azure Accessed: 2024-05-05]. In diesem Projekt wird die Middleware ROS 2 genutzt, welche speziell für die Entwicklung von Robotersoftware entwickelt wurde und daher eine Vielzahl von Funktionen und Diensten bietet, die speziell auf die Anforderungen derer zugeschnitten sind.

2.5 Robot Operating System 2

ROS 2 ist eine Middleware zur entwickeln von Software für Roboter. Sie ist stark typisiert und basiert auf einem anonymen Publish-Subscribe-Mechanismus, welcher die Weitergabe von Nachrichten zwischen verschiedenen Prozessen ermöglicht.
Das Herzstück eines jeden ROS 2-Systems ist der ROS-Graph. Er bezieht sich auf das Netzwerk von Knoten in einem ROS-System und die Verbindungen zwischen ihnen, über welche sie komunizieren [ROS Documentation Contributors Accessed: 2024-05-05[f]].
Im Folgenden wird näher auf die in diesem Projekt verwendeten Konzepte und Technologien von ROS 2 eingegangen.

2.5.1 Nodes

Eine Node, auch Knoten genannt, ist eine Nutzer-Bibliothek zur Kommunikation mit anderen Nodes. Diese können mit anderen Nodes, innerhalb eines Prozesses, in einem anderen Prozess oder auf einem anderen Rechner kommuniziert werden. Sie sind häufig die Einheit der Berechnung in einem ROS-Graphen. Dabei sollte jede Node eine logische Aufgabe erfüllen.
Nodes haben die Möglichkeit, bestimmte Topics zu veröffentlichen, um Daten an andere Nodes zu übermitteln oder bestimmte Themen zu abonnieren (subscriben), um Daten von anderen Nodes zu erhalten. Es ist möglich, dass Sie entweder als Service-Client agieren, um eine Berechnung in ihrem Namen durchzuführen, oder als Service-Server, um anderen Nodes Funktionen zur Verfügung zu stellen. Eine Node kann als Aktionsrechner fungieren, um eine weitere Node zu beauftragen, die Berechnung in ihrem Namen durchzuführen oder als Aktionsserver für wiederum weitere Nodes als Funktionen bereitzustellen. Nodes können zusätzlich konfigurierbare Parameter bereitstellen, um das Verhalten während der Laufzeit individuell anzupassen.
Eine Node ist oft eine komplexe Kombination aus Publishern, Subscribern, Service-Servern, Services-Clients, Action-Servern und Action-Servern und werden über einen verteilten Erkennungsprozess identifiziert [ROS Documentation Contributors Accessed: 2024-05-05[c]].
Eine solche Node wird in diesem Projekt für die Kommunikation und Übersetzung verschiedener Daten für den Pepper Roboter entwickelt.

2.5.2 Topics

Topics sind ein Mechanismus, um Nachrichten zwischen Nodes zu übertragen. Sie sind asynchron, strikt typisiert, anonym und ermöglichen die Kommunikation zwischen Nodes, ohne dass diese voneinander Kenntnis haben müssen. Dies ist möglich durch die Nutzung des Publish-Subscribe-Prinzip. Dieses wurde zwar schon in den Abschnitten Abschnitt 2.3 und speziell Abschnitt 2.1 erläutert, wird jedoch hier noch einmal im Kontext von ROS 2 aufgegriffen.
In ROS 2 können Nodes sowohl Produzenten sein und Daten in ein Topic veröffentlichen, als auch Konsumenten, die Daten aus einem Topic abonnieren. Die Nodes selbst können dabei beliebig viele Topics abonnieren oder veröffentlichen.
Dabei ist der Begriff Topic lediglich der Name unter dem die Nachrichten veröffentlicht und auch abgerufen werden können. Die Nachrichten selbst sind dabei in einem strikt typisierten Format, welches als Message bezeichnet und definiert wird und im Abschnitt 2.5.3 ganuer erläutert wird.
Werfen wir jedoch zunächst einen genaueren Blick auf die Deteils der Bedeutungen der Anonymität und der strikten typisierung im Kontext von ROS 2:

Anonymität

Die Anonymität in ROS 2 bezieht sich darauf, dass die Nodes, die Daten in ein Topic veröffentlichen oder von einem Topic abonnieren, nicht wissen, welche anderen Nodes ebenfalls auf dieses Topic zugreifen und primär von welcher Node sie überhaupt stammen. Dies ermöglicht eine lose Kopplung zwischen den Nodes, da sie nicht auf die Existenz oder den Zustand der anderen Nodes angewiesen sind. Dadurch werden ROS 2 Systeme flexibel und Nodes können durch die genannte Entkopplung völlig frei entwickelt, ausgetauscht und aus technischer Sicht auch gelöscht werden.

Strikte Typisierung

ROS 2 Nodes können zwar sowohl in Python als auch in C++ entwickelt werden, was zumindest auf Seiten der Python-Entwicklung eine dynamische Typisierung implizieren könnte, jedoch bildet sich diese Freiheit nicht auf die Nachrichten (Messages) ab, da diese auch im Raum der in C++ strikt typisiert ist, genutzt werden müssen. Dadurch ergeben sich die in den Interfaces definierten, strikten Typen, die von den Nodes eingehalten

werden müssen. Dies ermöglicht eine einfache und effiziente Kommunikation zwischen den Nodes, sollten diese auch in verschiedenen Sprachen implementiert sein, da die Nachrichten immer in einem festen Format vorliegen und die Nodes sich darauf verlassen können, dass die Nachrichten korrekt sind.
Weiter werden die Nachrichten auch semantisch strikt typisiert, was bedeutet, dass die Nachrichten auch inhaltlich korrekt sind und die Nodes sich darauf verlassen können, dass die Nachrichten die erwarteten Daten enthalten. Ein anschauliches Beispiel dafür sind die sogenannten Inertial Measurement Unit (IMU)-Messages. Sie enthalten als Datentyp ein dreidimensionales Array aus Fließkommazahlen, welche die Beschleunigung, die Winkelgeschwindigkeit und die Magnetfeldstärke eines Sensors in den drei Raumrichtungen beschreiben. Also ist nicht nur der Datentyp strikt festgelegt, sondern auch die einzelnen Werte und deren Reihenfolge, die in den Arrays enthalten sind [ROS Documentation Contributors Accessed: 2024-05-05[e]].

2.5.3 Interfaces

ROS 2 kommuniziert typischerweise über verschiedene Schnittstellen (Interfaces), die als Messages, Services und Actions bezeichnet werden. ROS 2 verwendet dabei eine vereinfachte Beschreibungssprache, um die Interfaces zu definieren, die als Interface Definition Language (IDL) bezeichnet wird. Diese Beschreibungssprache wird verwendet, um die Interfaces in verschiedenen Programmiersprachen zu generieren [ROS Documentation Contributors Accessed: 2024-05-05[a]].
Dabei bietet ROS 2 drei Typen von Interfaces welche folgend erläutert werden:

Messages

Messages werden in Topics genutzt und sind einfache Textdateien, die die Felder einer ROS-Nachricht beschreiben. Diese Nachrichten werden verwendet, um Source-Code für die Nachrichten zu generieren, die in verschiedenen Programmiersprachen verwendet werden können.
Die Messages sind eine Möglichkeit für Nodes, Daten über das Netzwerk mit weiteren Nodes auszutauschen, auch wenn keine Antwort erwartet wird. Als Beispiel kann eine Node eine Nachricht veröffentlichen, die die Daten eines Sensors enthält, und eine andere Node kann diese Nachricht abonnieren, ohne dass die erste Node eine Antwort erwartet

oder überhaupt von der Existenz der zweiten Node weiß.
Die `.msg`-Dateien, welche die Messages beschreiben, werden in einem separaten Verzeichnis im ROS-Paket unter `msg/` abgelegt. Sie beinhalten die Felder der Nachricht und Konstanten. Inhalt dieser Felder können nahezu alle gängigen Datentypen sein, wie etwa Ganzzahlen (Integer), Fließkommazahlen (Float), einzelne Zeichen (Char), Zeichenketten (String) aber auch statische Arrays. Einzige Einschränkung sind hierbei die Regelungen (Conventions) von ROS 2, welche die Verwendung von alphabetischen Zeichen zu Beginn und Unterstrichen zur Trennung einzelner Wörter im Namen der Felder vorschreiben.
Ein typischer Eintrag einer solchen Datei könnte, beispielhaft für die verschiedenen Felder, etwa wie folgt aussehen:

Algorithmus 2.1: ROS 2 Message

```
type name
int16 sensor_id
int32[] sensor_data

type name defaultvalue
bool enabled true

type CONSTANTNAME=constantvalue
string IP="192.168.100.11"
```

Services

Services sind im Gesensatz zu den Messages eine synchrone Kommunikation zwischen zwei Nodes. Ein Node sendet eine Anfrage an einen anderen Node und wartet auf dessen Antwort.
Sie werden ebenfalls in einem separaten Verzeichnis im ROS-Paket unter `srv/` als `.srv` Datei abgelegt und beschreiben die Anfrage und die Antwort, die von einem Service bereitgestellt wird. Auch hier folgt der Inhalt der Datei einem festgelegtem Schema ähnlich dessen der Message Dateien, mit dem Zusatz, dass hier getrennt von einer Zeile mit drei Bindestrichen als Inhalt `---`, auch die zu erwartende Antwort definiert wird.
Eine einfach gehaltene `.srv`-Datei könnte also etwa wie folgt aussehen:
Auch hier können wie bei den Messages auch komplexere Datenstrukturen, Konstanten

Algorithmus 2.2: ROS 2 Service

```
string myString
---
string yourString
```

und Standardwerte genutzt werden.

Actions

Actions sind eine Erweiterung der Services, die es ermöglichen, asynchrone, lang laufende Prozesse mit beidseitiger Kommunikation zu definieren. Im Gegensatz zu den Services, bei denen die Antwort direkt erwartet wird, können Actions mehrere Sekunden bis Minuten dauern. Diese Action kann in diesem Zeitraum auch unterbrochen und wieder aufgenommen werden, oder sogar ganz abgebrochen werden.
Auch Actions werden in einem separaten Verzeichnis im ROS-Paket unter `action/` als `.action` Datei abgelegt und beschreiben in einer ähnlichen Weise wie die Services und die Messages das erwartete Verhalten. In Gegensatz zu den beiden anderen Schnittstellen, wird hier jedoch nicht die Anfrage und die Antwort definiert, sondern ein Ziel, eine Rückmeldung und ein Feedback, welche im Schema gehalten durch einen Dreifachbindestrich `--` getrennt werden.
Nehmne wir zur Veranschaulichung die `Fibonacci .action`-Datei als Beispiel:
Es wird ein `Int32` abgeschickt und daraufhin ein Array aus wiederum `Int32` erwartet,

Algorithmus 2.3: ROS 2 Action

```
int32 order
---
int32[] sequence
---
int32[] sequence
```

mit den berechneten Werten. Während der Berechnung wird ein Feedback in Form eines `Int32` zurückgegeben, mit dem Stand bis zu einem bestimmten Zeitpunkt.

2.5.4 Parameter

Parameter in ROS 2 sind direkt verbunden mit den einzelnen Nodes. Sie werden genutzt, um die Konfiguration der Nodes beim Start un zur Laufzeit zu verwalten, ohne den darunterliegenden Code anpassen zu müssen. Die Lebenszeit der Parameter ist dabei an die Lebenszeit der Nodes gebunden, was bedeutet, dass die Parameter beim Start der Nodes geladen werden und beim Beenden der Nodes verfallen. Eine Node kann jedoch eine Art Persistenz implementieren um die Parameter zu speichern und wieder zu verwenden. Jeder dieser Parameter besteht aus einem Schlüssel (Key), einem Wert (Value) und einer Beschreibung (Descriptor). Während der Schlüssel ein `String` ist und den Parameter identifiziert, ist der Wert der eigentliche Inhalt des Parameters und wird zunächst als Typ definiert. Die Beschreibung ist optional und Standardmäßig `NULL`. Sie dient dazu, den Parameter zu beschreiben und zu dokumentieren, wie Typ-Informationen, Wertebereiche, Nutzungsdeteils und weitere Informationen welche zur Nutzung relevant sein können.
Zu Beginn der Laufzeit muss einen Node definieren welche Parameter diese aktzeptiert, und ob diese optional oder zwingend sind. Dies reduziert die Anfälligkeit auf Fehler in der Konfiguration im späteren Verlauf der Laufzeit. Die Parameter können dann auch zur Laufzeit über die API der Nodes abgerufen, gesetzt und gelöscht werden. Einige Arten von Nodes können auch Parameter aktzeptieren bevor diese bekannt sind. Dazu stellt ROS 2 den Parameter `allow_undeclared_parameters` zur Verfügung, welcher es bei aktivierung ermöglicht, dass Parameter gesetzt werden können, auch wenn sie nicht explizit in der Parameterliste der Node definiert sind. Dies ist allerdings nur möglich bei neu hinzugefügten Parametern, bereits vorhandene Parameter können nicht zur Laufzeit verändert werden. Dies würde die Grundlage der strikten Typisierung und der semantischen Korrektheit der Parameter verletzen und damit das gesamte System instabillisieren. Versuche dies trotzdem zu tun, werfen einen Fehler und die Node wird sofort terminiert.
Soll der Wert eines Parameters zur Laufzeit geändert werden, gibt es dazu zwei Möglichkeiten:

- Die `set_parameters` Methode, welche über die API der Node aufgerufen wird und die Parameterwerte ändert. Diese Methode setzt die gewünschten Parameterwerte und gibt die tatsächlich gesetzten Werte zurück, auch wenn die Node die Werte nicht akzeptiert hat. Dies ermöglicht es, die Parameterwerte zu überprüfen und

gegebenenfalls zu korrigieren oder gar abzulehnen.

- Die `on_parameter_event` Methode, welche aufgerufen wird, wenn sich ein Parameterwert ändert. Dadurch wird es der Node und dem User ermöglicht, auf Änderungen der Parameterwerte zu reagieren welche von anderen Nodes oder von der Command Line Interface (CLI) zur Laufzeit vorgenommen wurden.

Zur allgemeinen Interaktion mit den Parametern, bietet ROS 2 neben den bereits genannten, eine Vielzahl von API-Methoden, auch API-Services genannt, um die Parameter zu verwalten. Diese werden standardmäßig bei Initiierung einer Node geladen und bereitgestellt [ROS Documentation Contributors Accessed: 2024-05-05[d]].

2.5.5 Start

Eine ROS 2 Anwendung besteht im Allgemeinen aus mindestens einer, aber oft auch mehreren Nodes, die nach dem ROS-Pfad abgearbeitet werden und miteinander kommunizieren. Es zwar möglich diese Nodes einzeln, manuell zu starten, jedoch ist es in der Praxis üblich, die Nodes in einem Launch-File (Start-Datei) zu starten. Sie trägt die Endung `.launch.[Format]` und kann in Python (`.launch.py`), Extensible Markup Language (XML) (`.launch.xml`) oder YAML Ain't Markup Language (YAML) (`.launch.yaml`) geschrieben werden. Ein Launch-File ist eine Python-Datei, die die Konfiguration der Nodes und die Verbindung zwischen ihnen beschreibt. Sie kann auch Parameter, Argumente und Umgebungsvariablen enthalten, die von den Nodes verwendet werden.
Allgemein ist das Startsystem von ROS 2 dazu designed mehrere Nodes gleichzeitig zu starten und zu verwalten durch eine einzige Datei. Es verbessert die Wartbarkeit und die Skalierbarkeit der Anwendung ungemein und ermöglicht in der Überwachung einen deutlich vereinfachten Überblick über die laufenden Komponenten [ROS Documentation Contributors Accessed: 2024-05-05[b]].

2.5.6 Nutzer Bibliotheken

Client-Libraries (Nutzer Bibliotheken) sind eine Reihe an APIs und Tools, die es Entwicklern ermöglichen, ROS 2-Bausteine in verschiedenen Programmiersprachen selbst

zu schreiben. Dazu zählen neue Nodes, Topics, Services und andere ROS 2-Bausteine. Auch bei den Programmiersprachen bietet ROS 2 mehrere Möglichkeiten, um dem Nutzer die Wahl zu lassen und die Komponente bestmöglich an die Anforderungen des Nutzens anzupassen. Bausteine welche eher für die Visualisierung entwickelt wurden, werden in Python geschrieben, während Bausteine, die eine hohe Rechenleistung benötigen oder echtzeitfähig sein müssen, werden eher in C++ geschrieben.

rclcpp

Die ROS-Client-Bibliothek für C++ (rclcpp) ist die benutzerorientierte, idiomatische C++-Schnittstelle, die alle ROS-Client-Funktionen wie das Erstellen von Knoten, Publishern und Abonnements bereitstellt. Rclcpp baut auf ROS Client Library (rcl) und der ROSIDL-API auf und ist für die Verwendung mit den von `rosidl_generator_cpp` erzeugten C++-Nachrichten konzipiert.
Rclcpp nutzt alle Funktionen von C++ und C++17, um die Schnittstelle so einfach wie möglich zu gestalten. Da es jedoch die Implementierung in rcl wiederverwendet, ist es in der Lage, ein konsistentes Verhalten mit den anderen Client-Bibliotheken, die die rcl-API verwenden, beizubehalten.

rclpy

Die ROS-Client-Bibliothek für Python (rclpy) ist das Python-Gegenstück zur C++-Client-Bibliothek. Wie die C++-Client-Bibliothek auch, baut auch rclpy für seine Implementierung auf der rcl C API auf. Die Schnittstelle bietet ein idiomatisches Python-Erlebnis, das native Python-Typen und -Muster wie Listen und Kontextobjekte verwendet. Durch die Verwendung der rcl-API in der Implementierung bleibt die Konsistenz mit den anderen Client-Bibliotheken in Bezug auf Funktionsgleichheit und Verhalten erhalten. Die Python-Client-Bibliothek stellt nicht nur idiomatische Python-Bindungen für die rcl-API und Python-Klassen für jede Nachricht bereit, sondern kümmert sich auch um das Ausführungsmodell, indem sie `threading.Thread` oder eine ähnliche Methode verwendet, um die Funktionen der rcl-API auszuführen.
Wie auch bei C++ generiert sie benutzerdefinierten Python-Code für jede Ṛ-Nachricht, mit der der Benutzer interagiert, aber im Gegensatz zu C++ konvertiert sie schließlich das native Python-Nachrichtenobjekt in die C-Version der Nachricht. Alle Operationen

erfolgen mit der Python-Version der Nachrichten, bis sie an die ROS-Schicht weitergegeben werden müssen. Dann werden sie in die einfache C-Version der Nachricht umgewandelt, damit sie an die ROS-C-API weitergegeben werden kann. Dies wird nach Möglichkeit vermieden, wenn die Kommunikation zwischen Herausgebern und Abonnements im selben Prozess erfolgt, um die Konvertierung in und aus Python zu reduzieren.

Weitere

Neben den von ROS 2 bereitgestellten Client-Bibliotheken, gibt es auch weitere von der Community entwickelte Bibliotheken. Diese Bibliotheken bieten zusätzliche Unterstützung für weitere Sprachen, sind dadurch jedoch nicht so tief in das System integriert, werden teilweise nicht aktiv weiterentwickelt und teilweise sogar schon als Projekt eingestellt.
Zu diesen Bibliotheken zählen:

- `rclc` - welche die FUnktionalität von `rcl` nicht ersetzt sondern um einige nützliche Funktionen erweitert.
- `rcljava` - eine Java-Client-Bibliothek zur Unterstüzung von Java Virtual Machine (JVM)-Basierten Sprachen und Android-Java.
- `rclcs` - eine C#-Client-Bibliothek für .NET- und .NET-Core-Entwickler.
- `rclnodejs` - eine Node.js-Client-Bibliothek für die Enticklung in JavaScript.
- `rclrs` - eine Reihe an Projekten, welch die Entwicklung von ROS 2 in Rust unterstützen.

Ältere Bibliotheken, die nicht mehr aktiv weiterentwickelt werden, sind:

- `Ada`
- `iOS`

- `Zig`

2.6 Colcon

Colcon ist ein Kommandozeilenwerkzeug zur Verbesserung des Arbeitsablaufs beim Erstellen, Testen und Verwenden mehrerer Softwarepakete. Es automatisiert den Prozess, übernimmt die Bestellung und richtet die Umgebung für die Nutzung der Pakete ein [Open Source Robotics Foundation, Inc. 2022]. Der Fokus der Software liegt primär auf:

- Das Werkzeug sollte das Builden, Testen und Verwenden mehrerer Pakete vereinfachen.
- Es sollte möglich sein, Unterstützung für jede Art von Build-System mit Hilfe von Erweiterungen hinzuzufügen. Colcon-core bündelt nur Python-Unterstützung, um sich selbst zu booten.
- Es sollte möglich sein, eine beliebige Menge von Paketen zu bauen, ohne dass Änderungen an den Quellen erforderlich sind. Falls nötig, können fehlende Informationen von außen nachgeliefert werden.
- Nach dem Build der Pakete müssen diese sofort nutzbar sein, was das Einrichten der notwendigen Umgebungsvariablen und weiteren Abhängigkeiten einschließt.
- Die gesamte bereitgestellte Funktionalität sollte so offengelegt werden, dass sie von anderen Erweiterungen wiederverwendet werden kann.
- Die Aufteilung in mehrere Python-Pakete wird genutzt, um Modularität und lose Kopplung zu fördern. Sie wird auch verwendet, um die Erweiterbarkeit zu demonstrieren und zu zeigen, dass bestimmte Funktionen nicht “speziell” sind, sondern von außen beigesteuert werden können.

2.7 Programming Languages

Wie bereits in Unterabschnitt 2.5.6 erwähnt, bietet ROS 2 die Möglichkeit, Nodes in verschiedenen Programmiersprachen zu schreiben. Dabei sind die beiden Hauptprogram-

miersprachen, die von ROS 2 unterstützt werden, C++ und Python.

2.7.1 C++

C++ ist eine von Bjarne Stroustrup entwickelte, weit verbreitete Programmiersprache, die ursprünglich als Erweiterung der Programmiersprache C konzipiert wurde [Wikipedia contributors Accessed: 2024-05-05[a]]. Sie ist bekannt für ihre Leistungsfähigkeit und Flexibilität. Die Sprache bietet Funktionen wie starke Typisierung, effiziente Ressourcenverwaltung und die Möglichkeit der Hardwarezugriffssteuerung, was sie zu einer beliebten Wahl für die Entwicklung von Systemsoftware, Betriebssystemen und Embedded-Programmierung macht.
C++ verfügt über eine umfangreiche Standardbibliothek `std` sowie eine große Anzahl von Open-Source-Bibliotheken und Frameworks, die von der Community entwickelt wurden. Sie bieten eine Vielzahl von weiteren nützlichen Anwendungen, wie etwa numerische Berechnungen, Grafikprogrammierung, Netzwerkprogrammierung und Einige mehr. Beliebte Bibliotheken und Frameworks für C++ sind zum Beispiel `Boost` für allgemeine Zwecke, `Eigen` für lineare Algebra, `Simple and Fast Multimedia Library (SFML)` und `OpenGL` für Grafiken und Spieleentwicklung, sowie `Poco` für Netzwerk- und Systemprogrammierung [cplusplus.com Accessed: 2024-05-05].
C++ ist aufgrund seiner Leistungsfähigkeit, Flexibilität und weit verbreiteten Anwendung in verschiedenen Branchen eine beliebte Wahl für die Entwicklung von Softwareprojekten, insbesondere solchen, die eine hohe Leistung und effiziente Ressourcennutzung erfordern, wie etwa bei Robottern. Die Vielseitigkeit von C++ ermöglicht es Entwicklern, Anwendungen von kleinen Embedded-Systemen bis hin zu großen verteilten Systemen zu entwickeln.

2.7.2 Python

Python ist eine interpretierte High-Level-Programmiersprache, die erstmals 1991 von Guido van Rossum veröffentlicht wurde [Wikipedia contributors Accessed: 2024-05-05[b]]. Sie ist für ihre klare Syntax und Lesbarkeit bekannt, was sie zu einer beliebten Wahl für Anfänger und Fortgeschrittene macht. Einige Hauptmerkmale von Python sind dynamische Datentypen, interpretierte Ausführung und die Möglichkeit der objektorientierten Programmierung. Python verfügt außerdem über eine große Standardbibliothek und kann leicht mit C- und C++-Code erweitert werden, was es zu einer beliebten Wahl für eine

breite Palette von Anwendungen macht [Python Software Foundation Accessed: 2024-05-05].
Die Sprache wird auch von einer großen und aktiven Community, die zur Entwicklung vieler nützlicher Bibliotheken und Werkzeuge für die Sprache beigetragen hat. Diese Bibliotheken stehen für eine Vielzahl von Anwendungen zur Verfügung, z.B. für die Datenanalyse, wissenschaftliche Berechnungen und maschinelles Lernen. Zu den gängigen Bibliotheken und Tools für Python gehören `NumPy` und `Pandas` für die Datenanalyse, `TensorFlow` und `PyTorch` für maschinelles Lernen, `Django` und `Flask` für die Webentwicklung sowie `Matplotlib` und `Seaborn` für die Datenvisualisierung [VanderPlas 2016].
Python ist aufgrund seiner Lesbarkeit, Benutzerfreundlichkeit und großartigen Bibliotheksunterstützung eine beliebte Wahl für eine breite Palette von Anwendungen. Seine klare Syntax und Benutzerfreundlichkeit machen es zu einer Sprache für Anfänger und erfahrene Experten, aber auch zu einem leistungsstarken Werkzeug für eine Vielzahl von Anwendungen.

pyBullet

`pyBullet` ist eine Python-Bibliothek, die eine einfache Schnittstelle zur Simulation von Robotern und anderen physikalischen Systemen bietet. Sie basiert auf der Bullet Physics Engine, einer Open-Source-Physik-Engine, die für die Simulation von starren Körpern, Flüssigkeiten, Stoffen und anderen physikalischen Phänomenen verwendet wird. `pyBullet` ermöglicht es, Robotermodelle zu erstellen, ihre Kinematik und Dynamik zu simulieren und ihre Bewegung zu steuern.
Für eben diese Berechnung der inversen Kinematik wird auf Grund der einfachen Handhabung und der guten Dokumentation `pyBullet` verwendet. Die Bibliothek bietet eine Vielzahl von Funktionen und Methoden, die es ermöglichen, die Gelenkwinkel eines Roboters zu berechnen, um eine bestimmte Position und Orientierung des Endeffektors zu erreichen. Die einizige Voraussetzung ist, dass das Modell des Roboters in `pyBullet` geladen und konfiguriert ist. Dazu wird das Modell in einer `.urdf`-Datei, siehe Unterabschnitt 2.8.5, beschrieben und in `pyBullet` geladen. Anschließend können die Gelenkwinkel des Roboters berechnet werden, um die gewünschte Position und Orientierung des Endeffektors zu erreichen.

2.7.3 C#

C# (ausgesprochen "C-Sharp") ist eine moderne, objektorientierte Programmiersprache, die von Microsoft im Jahr 2000 als Teil seiner .NET-Initiative entwickelt wurde. Die Sprache wurde unter der Leitung von Anders Hejlsberg entwickelt und ist stark von anderen populären Sprachen wie C, C++ und Java inspiriert. C# ist für seine Vielseitigkeit, Robustheit und Benutzerfreundlichkeit bekannt und wird in einer Vielzahl von Anwendungen eingesetzt, von Desktop- und Web-Anwendungen bis hin zu mobilen Apps und Spielen, insbesondere in der Entwicklung mit der Unity-Engine.

2.7.4 Syntax und Grundstruktur

Die Syntax von C# ist einfach und klar, was es für Entwickler leicht macht, die Sprache zu erlernen und zu verwenden.

2.7.5 Objektorientierte Programmierung (OOP)

C# unterstützt die vier Hauptprinzipien der objektorientierten Programmierung:

1. **Abstraktion**: Das Verbergen komplexer Implementierungsdetails und das Zeigen nur der notwendigen Eigenschaften eines Objekts.
2. **Kapselung**: Das Zusammenfassen von Daten und Methoden, die auf diese Daten zugreifen, innerhalb einer Klasse und das Verbergen der Details der Implementierung vor anderen Klassen.
3. **Vererbung**: Die Fähigkeit einer Klasse, die Eigenschaften und Methoden einer anderen Klasse zu erben, was Code-Wiederverwendung und Hierarchien ermöglicht.
4. **Polymorphismus**: Die Fähigkeit, eine Methode auf verschiedene Weise zu implementieren oder zu überschreiben, was eine flexible und dynamische Nutzung von Methoden ermöglicht.

2.7.6 Vorteile von C#

C# bietet mehrere Vorteile, die es zu einer beliebten Wahl für Entwickler machen:

- **Einfach zu erlernen**: Die klare Syntax und die umfangreiche Dokumentation machen C# zu einer anfängerfreundlichen Sprache.
- **Leistungsfähig und flexibel**: C# ist leistungsfähig genug für komplexe Anwendungen

und flexibel genug für schnelle Entwicklung.
- **Große Community und Ressourcen**: Die große Entwicklergemeinschaft und die Fülle an Online-Ressourcen, Tutorials und Foren machen es einfach, Unterstützung zu finden und Wissen auszutauschen.
- **Integrierte Entwicklungsumgebungen (IDEs)**: Tools wie Visual Studio und Visual Studio Code bieten leistungsstarke Entwicklungsumgebungen mit Debugging- und Code-Analyse-Funktionen.

2.8 Kinematische Grundlagen

2.8.1 DH-Parameter

Die Denavit-Hartenberg-Parameter (DH-Parameter) sind eine Methode zur Beschreibung der Kinematik eines Roboters. Sie wurden von Jacques Denavit und Richard Hartenberg in den 1950er Jahren entwickelt und sind seitdem zu einem Standardwerkzeug in der Robotik geworden. Die DH-Parameter beschreiben die Beziehung zwischen den Gelenken eines Roboters und den Koordinatensystemen, die sie verbinden. Sie ermöglichen es, die Position und Orientierung eines Endeffektors in Bezug auf die Gelenkwinkel des Roboters zu berechnen.
Die DH-Parameter bestehen aus vier Parametern, die für jedes Gelenk des Roboters definiert werden. Diese Parameter sind:

- a_i - der Abstand zwischen den Gelenken $i-1$ und i entlang der gemeinsamen Normalen (X-Achse).
- α_i - der Winkel zwischen den Gelenken $i-1$ und i entlang der gemeinsamen Normalen (X-Achse).
- d_i - der Abstand zwischen den Gelenken $i-1$ und i entlang der gemeinsamen Tangente (Z-Achse).
- θ_i - der Winkel zwischen den Gelenken $i-1$ und i entlang der gemeinsamen Tangente (Z-Achse).

Gelenk	a_i	α_i	d_i	θ_i
1	a_1	α_1	d_1	θ_1
2	a_2	α_2	d_2	θ_2
3	a_3	α_3	d_3	θ_3

Tabelle 2.1: Eine DH-Parameter-Tabelle

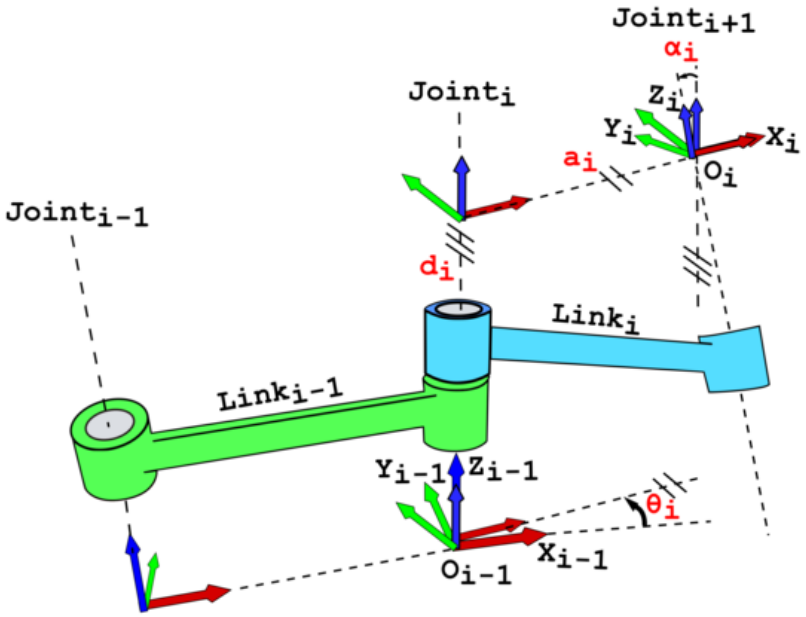

Abbildung 2.9: DH Parameter Beispiel[Wikipedia Accessed: 2024-05-05[a]]

Die DH-Parameter werden in einer Tabelle dargestellt, die die Beziehung zwischen den Gelenken und den Koordinatensystemen beschreibt. Diese Tabelle wird als DH-Parameter-Tabelle bezeichnet und enthält die DH-Parameter für jedes Gelenk des Roboters. Die DH-Parameter-Tabelle wird verwendet, um die Transformationsmatrizen zwischen den Koordinatensystemen zu berechnen, die die Position und Orientierung des Endeffektors beschreiben [Bittel Accessed: 2024-05-05].

2.8.2 Trajektion

Im physikalischen Sinne ist eine Trajektorie die Bahn, die ein Objekt durchläuft, wenn es sich bewegt [Wikipedia-Autoren Accessed: 2024-05-05]. Dieses Prinzip kann so direkt in der Robotik verwendet werden, denn auch in der Robotik wird darunter die Bewegung eines Roboters von einem Startpunkt zu einem Zielpunkt bezeichnet, also die Berechnung einer Bahn. Die Trajektion kann entweder linear oder gekrümmt sein, je

nach den Anforderungen der Anwendung. Die Trajektion wird oft durch eine Trajektorie beschrieben, die die Position und Orientierung des Roboters über die Zeit beschreibt.
Die Trajektorie kann durch verschiedene Methoden berechnet werden, darunter geometrische Methoden, numerische Methoden und optimierungsorientierte Methoden. Geometrische Methoden basieren auf der Geometrie des Roboters und verwenden trigonometrische Beziehungen, um die Trajektorie zu berechnen. Numerische Methoden verwenden iterative Verfahren, um die Trajektorie schrittweise zu approximieren. Optimierungsorientierte Methoden formulieren das Problem der Trajektorie als Optimierungsproblem und verwenden Optimierungsalgorithmen, um die Trajektorie zu berechnen.
Die wohl einfachsten Trajektorien sind lineare Trajektorien, bei denen der Roboter von einem Startpunkt zu einem Zielpunkt in einer geraden Linie bewegt wird. Diese Trajektorien sind einfach zu berechnen und zu implementieren, aber sie sind oft nicht optimal, da sie nicht die Dynamik des Roboters berücksichtigen. Andere Trajektorien, wie etwa gekrümmte Trajektorien, berücksichtigen die Dynamik des Roboters und sind oft komplexer zu berechnen und zu implementieren, weswegen in diesem Projekt primär die lineare Trajektion verwendet wird.

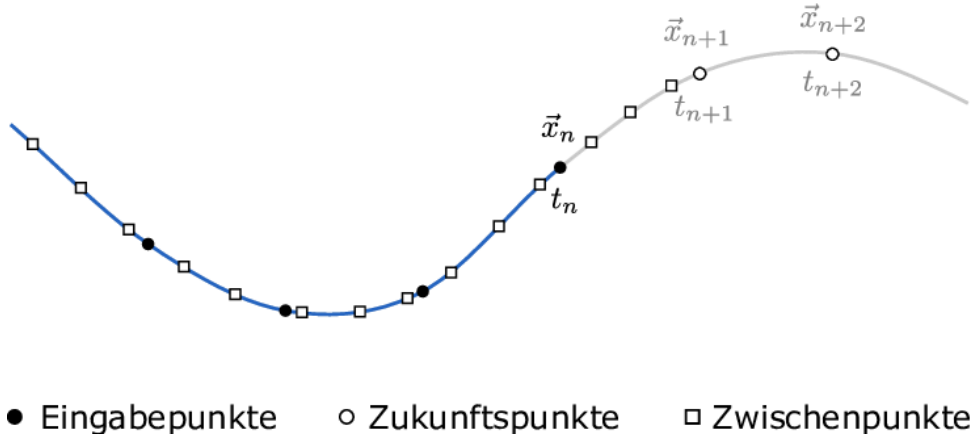

Abbildung 2.10: Trajektorie[Wefers 2015]

2.8.3 Interpolation

Unter Interpolation versteht man in der Robotik die Berechnung von Zwischenwerten zwischen zwei Punkten, um eine kontinuierliche Bewegung des Roboters zu ermöglichen. Die Interpolation wird oft verwendet, um die Gelenkwinkel des Roboters schrittweise zu ändern, um eine Trajektorie zu berechnen.

Die Interpolation kann durch verschiedene Methoden berechnet werden, darunter lineare Interpolation, kubische Interpolation und splinebasierte Interpolation.

- Lineare Interpolation berechnet die Zwischenwerte linear zwischen den beiden Punkten. Dies resultiert wie in Abbildung 2.11 in einer geraden Linie zwischen den beiden Punkten was in den Punkten selbst starke Beschleunigungen und Verzögerungen zur Folge hat.

Abbildung 2.11: Lineare Interpolation[Wikipedia Accessed: 2024-05-05[b]]

- Polynomial Interpolation, oder Interpolation über höhere Polynome, berechnet die Zwischenwerte der Punkte durch eine Funktion n-ten Grades. Wie in Abbildung 2.12 gezeigt ermöglicht dies eine deutlöich glattere und kontinuierlichere Interpolation als die lineare Interpolation.

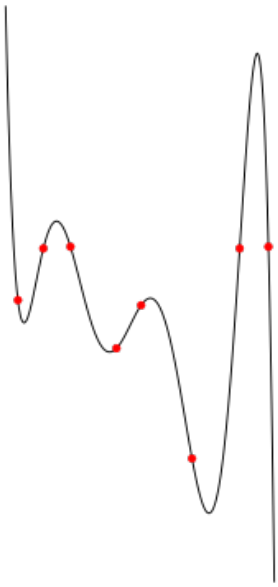

Abbildung 2.12: Polynomial Interpolation[Wikipedia Accessed: 2024-05-05[b]]

- Splinebasierte Interpolation berechnet die Zwischenwerte durch die Verwendung von Splines. In Abbildung 2.13 ist gezeigt wie eine solche Interpolation auf kubischer Basis aussehen könnte. Gut zu erkennen ist hier, dass diese Art der Interpolation eine glatte und kontinuierliche Kurve.

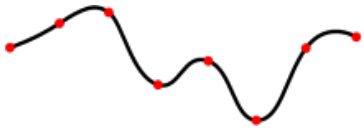

Abbildung 2.13: Spline Interpolation[WIKIPEDIA Accessed: 2024-05-05[b]]

Welche dieser Arten der Interpolation verwendet wird, hängt im Endeffekt von den Anforderungen der Anwendung ab.

2.8.4 Inverse Kinematik

Die inverse Kinematik ist ein mathematisches Problem, bei dem die Gelenkwinkel eines Roboters berechnet werden, um eine bestimmte Position und Orientierung des Endeffektors zu erreichen. Sie ist das Gegenstück zur direkten Kinematik, bei der die Position und Orientierung des Endeffektors aus den Gelenkwinkeln berechnet werden.

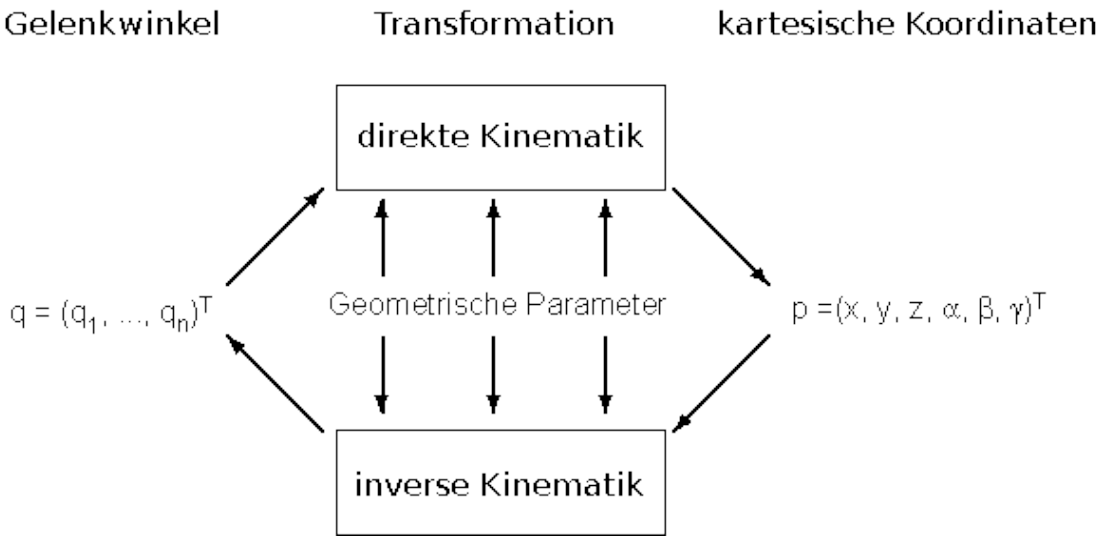

Abbildung 2.14: Inverse Kinematik[**wiki_inverse_kinematics**]

Das Problem der inversen Kinematik ist in der Regel nicht trivial, da es oft mehrere Lösungen gibt oder die Lösung nicht eindeutig ist. Es gibt verschiedene Methoden zur Lösung des Problems der inversen Kinematik, darunter geometrische Methoden, numerische Methoden und optimierungsorientierte Methoden.

Geometrische Methoden basieren auf der Geometrie des Roboters und verwenden trigonometrische Beziehungen, um die Gelenkwinkel zu berechnen. Numerische Methoden verwenden iterative Verfahren, um die Gelenkwinkel schrittweise zu approximieren. Optimierungsorientierte Methoden formulieren das Problem der inversen Kinematik als Optimierungsproblem und verwenden Optimierungsalgorithmen, um die Gelenkwinkel zu berechnen.
Die Wahl der Methode hängt von der Komplexität des Roboters, der Genauigkeit der Lösung und den Anforderungen der Anwendung ab. In der Praxis werden oft Kombinationen verschiedener Methoden verwendet, um das Problem der inversen Kinematik zu lösen.

2.8.5 Darstellung einer Kinematik

URDF

Unified Robot Description Format (URDF) ist ein XML-Format, das in ROS verwendet wird, um die physische Beschreibung und die kinematische Struktur eines Roboters zu definieren. Es enthält Informationen über die Größe, Form, Farbe und Position der verschiedenen Teile des Roboters sowie über die Gelenke, die diese Teile verbinden. Eine URDF-Datei besteht aus einer Reihe von Elementen, darunter:

- **Link**: Ein Link repräsentiert eine starre Komponente des Roboters. Jeder Link hat eine Masse und eine Trägheit, die für die physikalische Simulation des Roboters verwendet werden.
- **Joint**: Ein Joint verbindet zwei Links und definiert, wie sie sich relativ zueinander bewegen können. Es gibt verschiedene Arten von Gelenken, darunter feste Gelenke, Drehgelenke und Schiebegelenke.
- **Transmission**: Ein Transmission-Element definiert die mechanische Verbindung zwischen einem Motor und einem Gelenk.

SRDF

Semantic Robot Description Format (SRDF) ist ein weiteres XML-Format, das in ROS verwendet wird, um semantische Informationen über einen Roboter zu speichern, die über die physische Beschreibung in der URDF hinausgehen. Es wird hauptsächlich von der MoveIt-Bibliothek verwendet, um Informationen für die Bewegungsplanung zu speichern. Eine solche SRDF-Datei enthält folgende Elemente, darunter:

- **Group**: Eine Gruppe ist eine Sammlung von Gelenken und Links, die zusammen bewegt werden können. Gruppen werden oft verwendet, um die Arme, Beine oder andere bewegliche Teile eines Roboters zu repräsentieren.
- **End Effector**: Ein End Effector ist ein spezieller Link, der zum Interagieren mit der Umgebung verwendet wird, zum Beispiel eine Roboterhand oder ein Werkzeug.
- **Collision Matrix**: Die Collision Matrix definiert, welche Paare von Links ignoriert werden sollen, wenn Kollisionen erkannt werden. Dies ist nützlich, um falsche Kollisionen zu vermeiden, die durch die Nähe von Teilen des Roboters entstehen können.

Zusammen bieten URDF und SRDF eine vollständige Beschreibung eines Roboters, die sowohl seine physische Struktur als auch seine semantischen Informationen umfasst. Diese Beschreibungen können dann von verschiedenen ROS-Tools und -Bibliotheken verwendet werden, um Aufgaben wie Bewegungsplanung, Kollisionserkennung und Simulation durchzuführen.

2.9 Virtual Reality

Virtual Reality (VR) hat sich in den letzten Jahrzehnten von einem visionären Konzept zu einer greifbaren Technologie entwickelt, die in zahlreichen Bereichen unseres Lebens Anwendung findet. VR bezeichnet computergenerierte, dreidimensionale Umgebungen, die durch spezielle Hardware wie VR-Brillen und Bewegungssensoren eine immersive Erfahrung ermöglichen. Benutzer können in diese virtuellen Welten eintauchen und mit ihnen interagieren, was das Gefühl vermittelt, tatsächlich dort anwesend zu sein.

Die Wurzeln der VR-Technologie reichen bis in die 1960er Jahre zurück. Einer der frühesten Vorläufer moderner VR-Systeme war das Sensorama, das 1962 von Morton Heilig entwickelt wurde. Das Sensorama bot multisensorische Erlebnisse und kann als eines der ersten Systeme betrachtet werden, das den Nutzer vollständig in eine künstliche Umgebung eintauchen ließ [Heilig 1962]. Ein weiterer bedeutender Meilenstein war das SSword of Damocles", das erste echte VR-Headset, das 1968 von Ivan Sutherland entwickelt wurde. Es war ein klobiges Gerät, das an der Decke montiert werden musste, und bot eine rudimentäre Form der virtuellen Realität [Sutherland 1968].

In den 1980er und 1990er Jahren erlebte VR einen weiteren Aufschwung, insbesondere durch die Arbeit von Jaron Lanier, der den Begriff „Virtual Reality“ populär machte und die Firma VPL Research gründete, die einige der ersten kommerziellen VR-Produkte entwickelte [Lanier 1992]. Trotz dieser Fortschritte blieb VR lange Zeit eine Nischentechnologie, hauptsächlich aufgrund der hohen Kosten und technischen Einschränkungen.

Erst in den 2010er Jahren, mit der Einführung moderner, kostengünstigerer VR-Headsets wie der Oculus Rift, entwickelte sich VR zu einer breiter zugänglichen Technologie. Die Oculus Rift, ursprünglich 2012 auf Kickstarter finanziert, revolutionierte den Markt und führte zu einer neuen Welle von Innovationen in der VR-Technologie [Luckey 2012]. Kurz darauf folgten andere bedeutende Systeme wie die HTC Vive und PlayStation VR, die die VR-Erfahrung weiter verbesserten und breitere Zielgruppen erreichten.

Die heutige VR-Technologie zeichnet sich durch hochauflösende Displays, präzises Tracking und eine Vielzahl von Eingabemethoden aus, die eine immersive und interaktive Erfahrung ermöglichen. Meta Quest 3 (ehemals Oculus Quest 3) ist ein Beispiel für ein modernes, eigenständiges VR-Headset, das keine Verbindung zu einem leistungsstarken PC benötigt und dennoch eine beeindruckende Leistung bietet [Meta 2023].

Die Anwendungsbereiche von VR sind vielfältig und umfassen nicht nur Unterhaltung und Spiele, sondern auch Bildung, medizinische Therapie, Training und Simulationen, Architektur und Design sowie viele andere Felder. Zum Beispiel wird VR in der Medizin zur Behandlung von Phobien, in der Schmerztherapie und in der Rehabilitation eingesetzt [Rizzo und Koenig 2017]. In der Ausbildung ermöglicht VR realitätsnahe Trainingsumgebungen, die sicher und kontrolliert sind, was insbesondere in der Luftfahrt und der Medizin von großem Nutzen ist [Huang und Liaw 2018].

Die rapide Weiterentwicklung von Hardware und Software sowie die steigende Akzeptanz und Integration von VR in verschiedenen Lebensbereichen zeigen, dass diese Technologie das Potenzial hat, unsere Interaktion mit digitalen Inhalten und unsere Wahrnehmung von Realität grundlegend zu verändern. Die folgenden Abschnitte dieser Arbeit werden die technologischen Grundlagen von VR, verschiedene Anwendungsbereiche sowie die aktuellen Herausforderungen und Zukunftsaussichten der Technologie detailliert untersuchen.

2.10 Entwicklung für Virtual Reality

Die Entwicklungsumgebungen für VR sind Softwareplattformen, welche den Entwicklern die Werkzeuge und Funktionen zur Erstellung von Anwendungen für VR-Brillen bieten. Die zwei bekanntesten und am häufigsten verwendeten sind Unity und Unreal Engine. Diese beiden bieten eine umfassende Unterstützung für die VR-Entwicklung. Sie werden in vielen Anwendungen und Spielen und auch bei industriellen Lösungen verwendet. Im folgenden werden beide Entwicklungsplattformen vorgestellt, in Kapitel 3 wird dann erläutert für welche der beiden Plattformen sich für die Entwicklung entschieden wurde.

2.10.1 Unity

Unity ist eine weit verbreitete, benutzerfreundliche Entwicklungsplattform, die sich durch ihre Vielseitigkeit und umfangreiche Toolsets auszeichnet. Unity bietet integrierte Unterstützung für VR und wird häufig aufgrund seiner Benutzerfreundlichkeit und der breiten Community bevorzugt.

Benutzerfreundlichkeit Unity hat eine intuitive Benutzeroberfläche, für welche die Plattform auch bekannt ist. Ebenfalls bekannt ist es für die leichte Erlernbarkeit, weshalb es sowohl Einsteigern als auch erfahrenen Entwicklern zusagt [Technologies 2021c]. Unterstützte Plattformen: Unity unterstützt eine Vielzahl von VR-Plattformen, darunter Oculus Rift, HTC Vive, PlayStation VR und verschiedene mobile VR-Headsets wie Google Cardboard und Samsung Gear VR [Technologies 2021c].

Asset Store Der Unity Asset Store bietet eine große Auswahl an vorgefertigten Assets, Plugins und Tools, die die Entwicklung von VR-Anwendungen erleichtern und beschleunigen [Technologies 2021b].

Scripting Unity verwendet C# als Hauptprogrammiersprache, was Entwicklern ermöglicht, komplexe Interaktionen und Animationen zu erstellen [Technologies 2021c].

2.10.2 Unreal Engine

Unreal Engine wurde von Epic Games entwickelt und ist ebenfalls eine führende Plattform für die VR Entwicklung. Diese Entwicklungsumgebung ist vor allem für ihre leistungsstarke Grafik und Rendering-Fähigkeiten bekannt .

Grafikqualität Die Grafikqualität und realistischen visuellen Effekte, welche besonders hochwertige Spiele wichtig sind, sind bei Unreal Engine besonders ausgeprägt [Games 2021b]

Blueprint System Unreal Engine bietet das Blueprints Visual Scripting System, das es Entwicklern ermöglicht, ohne tiefgreifende Programmierkenntnisse komplexe Interaktionen zu erstellen [Games 2021a].

Unterstützte Plattformen Wie Unity unterstützt auch Unreal Engine eine breite Palette von VR-Plattformen, einschließlich Oculus Rift, HTC Vive und PlayStation VR [Games 2021b].

Open Source Ein großer Vorteil der Unreal Engine ist ihr Open-Source-Charakter, der Entwicklern vollständigen Zugang zum Quellcode der Engine bietet, was tiefergehende Anpassungen und Optimierungen ermöglicht [Games 2021b].

2.10.3 OpenXR

OpenXR ist ein offener Standard für die Entwicklung von VR- und Augmented Reality (AR)-Anwendungen, der von der Khronos Group entwickelt wurde. Der Standard zielt darauf ab, die Fragmentierung in der VR- und AR-Industrie zu reduzieren und Entwicklern eine einheitliche Plattform für die Entwicklung von Anwendungen zu bieten. OpenXR ermöglicht es Entwicklern, Anwendungen zu erstellen, die auf einer Vielzahl von VR- und AR-Plattformen laufen, ohne dass sie für jede Plattform spezifischen Code schreiben müssen. Dies erleichtert die Entwicklung und den Einsatz von VR- und AR-Anwendungen erheblich und trägt zur Interoperabilität und Kompatibilität zwischen verschiedenen Plattformen bei [Technologies 2021a].

2.11 Unity-ROS TCP Controller

Der Unity-ROS TCP Controller ist ein leistungsstarkes Tool, das es ermöglicht, Robot Operating System (ROS) mit der Unity-Engine zu verbinden. Diese Integration bietet Entwicklern die Möglichkeit, VR- und AR-Anwendungen zu erstellen, die mit realen Robotern interagieren können, indem sie Daten zwischen ROS und Unity austauschen. Diese Funktionalität ist besonders nützlich in Bereichen wie Robotik, Simulationen und erweiterter Realität, wo die Interaktion zwischen virtuellen und physischen Welten von

zentraler Bedeutung ist [*ROS-TCP-Endpoint Documentation* o. D.; *Unity-ROS-TCP-Connector Documentation* o. D.]

2.11.1 Grundlagen und Architektur

Der Unity-ROS TCP Controller funktioniert durch die Verwendung eines TCP-Protokolls zur Kommunikation zwischen ROS und Unity. Dies ermöglicht eine zuverlässige und bidirektionale Datenübertragung, die für Anwendungen erforderlich ist, die auf Echtzeitdaten angewiesen sind. Die grundlegende Architektur besteht aus zwei Hauptkomponenten:

1. **ROS TCP Endpoint**: Dies ist der Server, der auf der ROS-Seite läuft und auf eingehende Verbindungen von Unity wartet. Er empfängt Nachrichten von Unity und sendet Nachrichten an Unity [*ROS-TCP-Endpoint Documentation* o. D.]
2. **Unity TCP Connector**: Dies ist der Client, der in Unity läuft und eine Verbindung zum ROS TCP Endpoint herstellt. Er sendet Nachrichten an ROS und empfängt Nachrichten von ROS [*Unity-ROS-TCP-Connector Documentation* o. D.]

2.11.2 Einrichtung und Verwendung

Die Einrichtung des Unity-ROS TCP Controllers umfasst mehrere Schritte, sowohl auf der ROS- als auch auf der Unity-Seite. Hier ist eine allgemeine Anleitung zur Einrichtung:

ROS-Seite

- Installiere die erforderlichen ROS-Pakete, z.B. `ros_tcp_endpoint` [*ROS-TCP-Endpoint Documentation* o. D.]
- Starte den ROS TCP Endpoint:

  ```
  roslaunch ros_tcp_endpoint endpoint.launch
  ```

Unity-Seite

- Importiere das ROS-TCP-Connector-Paket in dein Unity-Projekt [*Unity-ROS-TCP-Connector Documentation* o. D.]
- Füge das `RosConnector`-Skript zu einem GameObject in deiner Szene hinzu und konfiguriere die IP-Adresse und den Port des ROS-TCP-Endpunkts.

- Erstelle Nachrichten-Typen in Unity, die mit den ROS-Nachrichten übereinstimmen, die du senden und empfangen möchtest.
- Verwende das `RosConnector`-Skript, um Nachrichten an ROS zu senden und Nachrichten von ROS zu empfangen.

2.11.3 Beispiel für Unity-Skript

Hier ist ein Beispiel für ein Unity-Skript, das den Unity-ROS TCP Controller verwendet, um Positionsdaten an ROS zu senden:

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.Geometry;

public class PositionPublisher : MonoBehaviour
{
    ROSConnection ros;
    public string topicName = "/unity/position";
    public float publishRate = 0.5f;

    private float timeElapsed;

    void Start()
    {
        ros = ROSConnection.instance;
        ros.RegisterPublisher<PointMsg>(topicName);
    }

    void Update()
    {
        timeElapsed += Time.deltaTime;

        if (timeElapsed > publishRate)
        {
            PointMsg positionMessage = new PointMsg(
                transform.position.x,
```

```
                transform.position.y,
                transform.position.z
            );

            ros.Publish(topicName, positionMessage);

            timeElapsed = 0;
        }
    }
}
```

In diesem Beispiel wird die Position eines GameObjects in Unity periodisch an ein ROS-Thema gesendet. Das Skript:

- Registriert einen Publisher für das angegebene ROS-Thema.
- Sendet die Position des GameObjects als `PointMsg`-Nachricht an ROS [*Unity-ROS-TCP-Connector Documentation* o. D.]

2.11.4 Vorteile und Anwendungsbereiche

Der Unity-ROS TCP Controller bietet zahlreiche Vorteile:

- **Echtzeit-Interaktion**: Ermöglicht die Echtzeit-Interaktion zwischen virtuellen und physischen Robotern.
- **Flexibilität**: Unterstützt verschiedene ROS-Nachrichtentypen und ermöglicht die Anpassung an spezifische Anwendungsanforderungen.
- **Erweiterbarkeit**: Kann in verschiedene VR/AR-Projekte integriert werden, um erweiterte Robotik-Simulationen und Visualisierungen zu erstellen.

Anwendungsbereiche umfassen:

- **Robotik-Forschung**: Simulation und Testen von Robotern in virtuellen Umgebungen.
- **Bildung**: Interaktive Lernumgebungen für die Robotik-Ausbildung.
- **Industrie**: Visualisierung und Steuerung von Industrierobotern in einer virtuellen Umgebung.

2.11.5 Fazit

Der Unity-ROS TCP Controller ist ein mächtiges Werkzeug, das die Lücke zwischen virtuellen Simulationen in Unity und realen Robotersystemen, die auf ROS basieren, schließt. Durch die Nutzung von TCP für die Kommunikation bietet es eine robuste und flexible Lösung für eine Vielzahl von Anwendungen in der Robotik und darüber hinaus.

2.12 Unreal Engine ROS Integration

2.12.1 Einführung

Die Unreal Engine ROS Integration ermöglicht die Verbindung und Interaktion zwischen dem Robot Operating System (ROS) und der Unreal Engine. Diese Integration bietet ähnliche Funktionen wie der Unity-ROS TCP Controller und ist nützlich für die Erstellung von Simulationen, Visualisierungen und interaktiven Anwendungen, die mit physischen Robotern kommunizieren [ROSIntegration Contributors o. D.]

2.12.2 Grundlagen und Architektur

Das ROSIntegration Plugin für die Unreal Engine besteht aus mehreren Komponenten, die zusammenarbeiten, um eine nahtlose Kommunikation zwischen ROS und der Unreal Engine zu gewährleisten. Die grundlegende Architektur umfasst:

1. **ROS Nodes**: Diese werden in ROS definiert und dienen zur Kommunikation und Datenverarbeitung.
2. **Unreal Engine ROS Nodes**: Diese Nodes werden in der Unreal Engine erstellt und fungieren als Schnittstelle zu den ROS Nodes.

2.12.3 Einrichtung und Verwendung

Die Einrichtung des ROSIntegration Plugins in der Unreal Engine umfasst mehrere Schritte:

Installation des Plugins

- Lade das ROSIntegration Plugin von der offiziellen GitHub-Seite herunter: `https://github.com/code-iai/ROSIntegration`.

- Füge das Plugin zu deinem Unreal Engine Projekt hinzu und aktiviere es in den Projekteinstellungen.

Konfiguration des Plugins

- Konfiguriere die IP-Adresse und den Port des ROS Masters, mit dem die Unreal Engine kommunizieren soll.
- Stelle sicher, dass das ROS Master läuft und erreichbar ist.

Erstellen von ROS-Komponenten in Unreal

- Erstelle ROS-spezifische Komponenten wie Publisher und Subscriber in der Unreal Engine, um Daten zu senden und zu empfangen.

2.12.4 Beispiel für Unreal Engine Skript

Hier ist ein Beispiel für die Verwendung des ROSIntegration Plugins in der Unreal Engine, um Positionsdaten zu veröffentlichen:

```
#include "ROSIntegrationGameMode.h"
#include "ROSIntegration/Public/ROSIntegrationGameInstance.h"
#include "ROSIntegration/Public/Publisher.h"
#include "ROSIntegration/Public/std_msgs/String.h"

void AROSIntegrationGameMode::BeginPlay()
{
    Super::BeginPlay();

    UROSIntegrationGameInstance* ROSInst =
        Cast<UROSIntegrationGameInstance>(GetGameInstance());
    if (ROSInst)
    {
        ROSInst->Init();

        TSharedPtr<ROSPublisher> Publisher =
            MakeShareable(new ROSPublisher(ROSInst->ROSIntegrationCore,
                                           TEXT("/unity/position"),
```

```
                                                TEXT("geometry_msgs/Point")));
        Publisher->Advertise();

        // Publish a message
        FROSTime now = ROSInst->ROSIntegrationCore->ROSTimeNow();
        TSharedPtr<ROSMessages::geometry_msgs::Point> PointMessage =
            MakeShareable(new ROSMessages::geometry_msgs::Point());
        PointMessage->x = GetActorLocation().X;
        PointMessage->y = GetActorLocation().Y;
        PointMessage->z = GetActorLocation().Z;
        Publisher->Publish(PointMessage);
    }
}
```

In diesem Beispiel wird die Position eines Akteurs (Actors) in der Unreal Engine an ein ROS-Thema gesendet. Das Skript:

- Initialisiert das ROSIntegration Plugin.
- Erstellt einen Publisher für das angegebene ROS-Thema.
- Sendet die Position des Akteurs als `geometry_msgs/Point` Nachricht an ROS [ROSIntegration Contributors o. D.]

2.12.5 Vorteile und Anwendungsbereiche

Die Integration von ROS in die Unreal Engine bietet zahlreiche Vorteile:

- **Echtzeit-Interaktion**: Ermöglicht die Echtzeit-Interaktion zwischen virtuellen und physischen Robotern.
- **Visuelle Genauigkeit**: Die Unreal Engine bietet hochwertige Grafiken und realistische Visualisierungen, die für Simulationen und Präsentationen nützlich sind.
- **Flexibilität**: Unterstützt verschiedene ROS-Nachrichtentypen und ermöglicht die Anpassung an spezifische Anwendungsanforderungen.

Anwendungsbereiche umfassen:

- **Robotik-Forschung**: Simulation und Testen von Robotern in virtuellen Umgebungen.
- **Bildung**: Interaktive Lernumgebungen für die Robotik-Ausbildung.
- **Industrie**: Visualisierung und Steuerung von Industrierobotern in einer virtuellen Umgebung.

2.12.6 Fazit

Die Unreal Engine ROS Integration ist ein mächtiges Werkzeug, das die Lücke zwischen virtuellen Simulationen in der Unreal Engine und realen Robotersystemen, die auf ROS basieren, schließt. Durch die Nutzung dieses Plugins können Entwickler hochwertige Simulationen und Anwendungen erstellen, die eine nahtlose Integration von ROS-Daten in die Unreal Engine ermöglichen.

Kapitel 3

Technologieauswahl

3.1 VR-Brillen

Die Auswahl der Technologie für die Entwicklung von Virtual-Reality-Anwendungen ist von entscheidender Bedeutung für den Erfolg eines Projekts. Bei der DHBW wurde die Entscheidung getroffen, die Metaquest 3 für das VR-Projekt zu verwenden. Diese Entscheidung wurde aufgrund mehrerer Faktoren getroffen, die im Folgenden erläutert werden.

Die Metaquest 3 wurde aufgrund ihrer vielseitigen Anwendungsmöglichkeiten und ihrer Benutzerfreundlichkeit ausgewählt. Die DHBW legt großen Wert darauf, den Studierenden eine moderne und zugängliche Lernumgebung zu bieten. Die Metaquest 3 erfüllt diese Anforderungen durch ihre intuitive Bedienung und ihre Fähigkeit, komplexe VR-Erlebnisse bereitzustellen, ohne dass zusätzliche Hardware wie externe Sensoren erforderlich sind.

Ein weiterer wichtiger Faktor bei der Auswahl der Metaquest 3 war die Integration von Oculus in die bestehende IT-Infrastruktur der DHBW. Die Unterstützung und Zusammenarbeit mit Oculus ermöglichte es der DHBW, Schulungen und Support für die Verwendung der Metaquest 3 bereitzustellen. Dies erleichterte die Einführung der VR-Technologie in den Lehrplan und sorgte für eine nahtlose Integration in bestehende Lehr- und Lernaktivitäten.

Darüber hinaus bietet die Metaquest 3 eine breite Palette von Anwendungen und Inhalten über den Oculus Store, einschließlich Bildungs- und Trainingsanwendungen, die für den Einsatz in der Hochschulbildung geeignet sind. Die Verfügbarkeit von hochwertigen Bildungsressourcen spielte eine wichtige Rolle bei der Entscheidung für die Metaquest 3, da sie den Lehrern und Studierenden Zugang zu einer Vielzahl von Lernmaterialien und

Simulationen bietet, die den Lernprozess unterstützen und verbessern können.

Insgesamt wurde die Metaquest 3 aufgrund ihrer Benutzerfreundlichkeit, Integration in die bestehende Infrastruktur der DHBW und der Verfügbarkeit von Bildungsressourcen als ideale Wahl für das VR-Projekt der DHBW angesehen. Diese Auswahl bietet nicht nur eine solide Grundlage für die Entwicklung von VR-Anwendungen, sondern ermöglicht es auch, die VR-Technologie effektiv in den Lehrplan zu integrieren und den Lernerfolg zu steigern.

3.2 Entwicklung für VR-Brillen

Die Entscheidung, Unity als Entwicklungsplattform für das VR-Projekt "Pepper VR – Teleoperation eines humanoiden Roboters auf Basis der Analyse menschlicher Bewegungßu wählen, wurde nach sorgfältiger Abwägung verschiedener Faktoren getroffen, wobei insbesondere auch die Unreal Engine in Betracht gezogen wurde.

Unity

Im folgenden werden die Vor- und Nachteile von Unity beleuchtet.

Vorteile:

1. **Branchenübliche Plattform**: Unity ist eine der führenden Entwicklungsplattformen für VR-Anwendungen und wird von einer großen Community von Entwicklern und Unternehmen weltweit genutzt. Diese weitverbreitete Akzeptanz macht Unity zu einer branchenüblichen Wahl für die Entwicklung von VR-Inhalten und bietet Zugang zu einer Vielzahl von Ressourcen, Tutorials und Support, die für die erfolgreiche Umsetzung des Projekts entscheidend sind.

2. **Umfangreiche Funktionalitäten**: Unity bietet eine umfangreiche Auswahl an Funktionen und Werkzeugen, die speziell für die Entwicklung von VR-Anwendungen konzipiert sind. Die Integration von VR-Technologien wie Oculus Rift, HTC Vive und Metaquest in Unity ermöglicht es den Entwicklern, immersive VR-Erlebnisse mit hoher Qualität zu erstellen. Darüber hinaus bietet Unity eine benutzerfreundliche Oberfläche und eine intuitive Entwicklungsumgebung, die auch für Anfänger leicht zugänglich ist.

3. **Plattformübergreifende Unterstützung**: Unity ermöglicht die Entwicklung von VR-Anwendungen, die auf einer Vielzahl von Plattformen ausgeführt werden

können, einschließlich PC, Konsolen, Mobilgeräten und Webbrowsern. Diese Flexibilität eröffnet die Möglichkeit, das VR-Projekt auf verschiedenen Geräten und Betriebssystemen zu testen und bereitzustellen, um eine maximale Reichweite und Zugänglichkeit zu gewährleisten.

4. **Erweiterbarkeit und Anpassbarkeit**: Unity zeichnet sich durch seine Erweiterbarkeit und Anpassbarkeit aus. Durch den Einsatz von Plugins und Assets aus dem Unity Asset Store können Entwickler zusätzliche Funktionen und Ressourcen in ihre VR-Anwendungen integrieren, was die Entwicklung beschleunigt und die Produktivität erhöht. Darüber hinaus bietet Unity die Möglichkeit, eigene Tools und Skripte zu erstellen, um die spezifischen Anforderungen des Projekts zu erfüllen und maßgeschneiderte Lösungen zu entwickeln.

Nachteile:

1. **Grafische Qualität**: Obwohl Unity in Bezug auf die Grafikqualität fortschrittliche Techniken bietet, erreicht es möglicherweise nicht das gleiche grafische Niveau wie die Unreal Engine, insbesondere bei fotorealistischen Rendering-Anforderungen.

2. **Lernkurve**: Unity kann eine steilere Lernkurve haben als die Unreal Engine, insbesondere für Anfänger oder Personen ohne Programmiererfahrung. Die Vielzahl von Funktionen und Optionen kann anfangs überwältigend sein.

Unreal Engine

Nun werden die Vor- und Nachteile von Unreal Engine betrachtet.

Vorteile:

1. **Grafische Qualität**: Die Unreal Engine ist bekannt für ihre beeindruckende Grafikqualität und ihre Fähigkeit, fotorealistische Umgebungen zu erstellen. Sie bietet fortschrittliche Rendering-Techniken wie Raytracing und hochwertige Materialien, die für VR-Anwendungen mit hohen grafischen Anforderungen von Vorteil sind.

2. **Visuelle Skripting-Tools**: Die Unreal Engine bietet visuelle Skripting-Tools wie den Blueprint-Editor, die es auch Personen ohne umfangreiche Programmierkenntnisse ermöglichen, komplexe Logik und Interaktionen zu erstellen. Dies kann die Entwicklungszeit verkürzen und die Kreativität fördern.

3. **Leistung**: Die Unreal Engine ist für ihre hohe Leistung und Stabilität bekannt, insbesondere bei großen Projekten mit komplexen Szenen und großen Datenmengen.

Nachteile:

1. **Einschränktere Plattformunterstützung**: Im Vergleich zu Unity bietet die Unreal Engine möglicherweise eine eingeschränktere Plattformunterstützung für die Entwicklung von VR-Anwendungen. Die Unterstützung für bestimmte VR-Geräte oder Plattformen kann begrenzt sein.

2. **Komplexität**: Die Unreal Engine kann aufgrund ihrer fortschrittlichen Funktionen und der visuellen Komplexität ihrer Benutzeroberfläche für Anfänger schwieriger zu erlernen sein. Die Blueprint-Logik kann zwar visuell sein, erfordert jedoch immer noch ein Verständnis von Konzepten wie Variablen und Logik.

Entscheidung

Trotz der Vorteile der Unreal Engine in Bezug auf Grafikqualität und Leistung wurde Unity als die bevorzugte Entwicklungsplattform für das VR-Projekt "Pepper VR – Teleoperation eines humanoiden Roboters auf Basis der Analyse menschlicher Bewegung"gewählt. Die breite Unterstützung, die umfangreichen Funktionalitäten, die plattformübergreifende Unterstützung und die Erweiterbarkeit von Unity waren entscheidend für diese Wahl. Unity bietet eine solide Grundlage für die Entwicklung hochwertiger VR-Anwendungen, die den Anforderungen des Projekts gerecht werden und eine erfolgreiche Integration von VR-Technologie in den Lehrplan ermöglichen.

3.2.1 Verbindungstechnologie

Die Wahl von Unity als Entwicklungsplattform für das VR-Projekt "Pepper VR – Teleoperation eines humanoiden Roboters auf Basis der Analyse menschlicher Bewegung"führte zur Notwendigkeit, eine Methode zur Kommunikation zwischen der Unity-Anwendung und dem humanoiden Roboter Pepper zu implementieren. Die ROS TCP-Verbindung wurde aufgrund ihrer Kompatibilität mit Unity und der Robotersteuerung über das Robot Operating System (ROS) als geeignete Lösung identifiziert.

Kompatibilität mit Unity

ROS TCP (Transmission Control Protocol) bietet eine zuverlässige Methode zur Datenübertragung zwischen ROS und Unity. Unity unterstützt die Kommunikation über TCP/IP-Sockets, was es ermöglicht, Daten zwischen der Unity-Anwendung und externen Geräten wie Robotern über das Netzwerk auszutauschen. Durch die Implementierung einer ROS TCP-Verbindung kann die Unity-Anwendung Befehle an den Roboter senden und Daten von ihm empfangen, was eine nahtlose Integration in das VR-Erlebnis ermöglicht.

Robotersteuerung über ROS

Pepper, der humanoide Roboter, wird über das Robot Operating System (ROS) gesteuert, das eine Standardplattform für die Entwicklung von Robotersoftware ist. ROS bietet eine Vielzahl von Funktionen zur Robotersteuerung, einschließlich der Unterstützung für verschiedene Sensoren, Aktuatoren und Navigationssysteme. Indem die ROS TCP-Verbindung verwendet wird, kann die Unity-Anwendung mit den ROS-Nodes kommunizieren, die für die Steuerung von Pepper zuständig sind. Dies ermöglicht es der Unity-Anwendung, Pepper-Bewegungen zu steuern und Sensordaten von Pepper zu empfangen, um ein interaktives VR-Erlebnis zu schaffen.

Implementierung in Unity

Die Implementierung der ROS TCP-Verbindung in Unity erfolgt mithilfe von Plugins oder eigenen Skripten, die die TCP/IP-Kommunikation ermöglichen. Durch die Verwendung von vorhandenen ROS-Bibliotheken oder der Erstellung benutzerdefinierter ROS-Nodes kann die Unity-Anwendung ROS-Nachrichten senden und empfangen, um mit Pepper zu interagieren. Dies ermöglicht es, die Bewegungen von Pepper in Echtzeit zu steuern und Feedbackdaten von Pepper in die Unity-Anwendung zu integrieren.

Insgesamt wurde die ROS TCP-Verbindung aufgrund ihrer Kompatibilität mit Unity und der Robotersteuerung über ROS als ideale Lösung für die Kommunikation zwischen der Unity-Anwendung und dem humanoiden Roboter Pepper ausgewählt. Diese Entscheidung ermöglicht es, eine immersive und interaktive VR-Erfahrung zu schaffen, bei der die Benutzer direkt mit dem Roboter interagieren können.

Kapitel 4

Umsetzung

4.1 MetaQuest3

4.2 ROS-Topics

Die allgemeine Datenkommunikation des Projektes wird zwischen den Komponenten ausschließlich über ROS-Topics abgewickelt und über ROS verwaltet. Welche dabei standardmäßig von welchen Komponenten angeboten werden und welche noch zusätzlich hinzugefügt werden müssen werden im folgenden Abschnitt beschrieben.

4.2.1 Videostream und Kamerabilder

(nicht vollständig Implementiert)

Einen Videostream wie man ihn von einem normalen Video-Player kennt, ist in ROS nicht direkt möglich. Stattdessen wird der Stream als eine Abfolge von Bildern übertragen. Diese Bilder werden in einer Node, die Zugriff auf die Kamera hat, aufgenommen, nach Bedarf verarbeitet und schließlich über ein ROS-Topic veröffentlicht.
Dieser pseudo-Stream wird standardmäßig von der *NAOqi_bridge* bereitgestellt. Dadurch ist nicht nur der Zugriff auf die front-Kamera des Roboters gegeben, sondern alle Kameras und Sensoren des Roboters. Bei den Kameras kann zwischen der front-Kamera und der bottom-Kamera gewählt werden, wobei die NAOqi_bridge die Bilder der Kameras in verschiedenen Topics veröffentlicht wodurch verschiedene Features der Kameras genutzt werden können.

Die verschiedenen Topics sind kaskadiert aufgebaut und liegen unter dem Topic */pepper_robot/camera/front/*, */pepper_robot/camera/bottom/* und */naoqi_bridge/camera/depth* wobei für das Projekt nur die Frontkamera relevant ist. Sie bietet folgende Topics an:

- *image_raw* - Das unverarbeitete Bild der Kamera in voller Auflösung und Farbtiefe
- *image_raw/compressed* - Das komprimierte Bild der Kamera komprimiert durch die Joint Photographic Experts Group (JPEG)-Kompression
- *image_raw/theora* - Das komprimierte Bild der Kamera im Theora-Format ebenfalls komprimiert mit Theora

Das Projekt nutzt das Topic *image_raw/compressed* um die Bilder der Kamera zu empfangen. Eine direkte Übertragung der Bilder unkomprimiert, ist in der Theorie zwar möglich ist aber für eine AR Anwendung nicht von Vorteil, da die Bilder in der Regel nicht in voller Auflösung benötigt werden und die Übertragung der Bilder unkomprimiert zu einer hohen Netzwerkauslastung führen würde, was wiederum die Latenz der Bilder erhöhen und die Anzahl der Bilder pro Sekunde begrenzen würde. Dies wiederum würde im Umkehrschluss die Qualität der AR Anwendung drastisch verschlechtern und ein magelhaftes Erlebnis bieten, welches bis hin zur Übelkeit des Nutzers reichen könnte.
Die Bilder werden von der NAOqi_bridge in einem festen Intervall von 30 Bildern pro Sekunde veröffentlicht. Diese Rate ist bekannt als die niedrigste Rate, die der Mensch als flüssige Bewegung wahrnimmt und damit für die Anwendung ausreichend.
Mit einer High Definition (HD) Auflösung von 1280x720 Pixel, 8-Bit Farbtiefe und einer angenommenen Kompressionsrate von 0.5 durch JPEG, ergibt sich bei den genannten 30 Bildern pro Sekunde unkomprimiert eine Datenrate von etw. 663Mbit/s und mit Kompression etwa 331.5Mbit/s.

$$1280px \cdot 720px \cdot 8bit \cdot 30fps = 663Mbit/s \tag{4.1}$$

$$633Mbit/s \cdot 0.5 = 331.5Mbit/s \tag{4.2}$$

4.2.2 Teleoperationspositionen

(nicht vollständig Implementiert)

Der Roboter selbst besitzt drei maßgebende Kinematiken, welche für die Teleoperation genutzt werden. Diese sind die beiden Arme *LArm* und *RArm* und der Kopf *Head*.

Eine Ausnahme bildet das Fahrwerk, welches nicht direkt über Teleoperation gesteruert wir, aber trotzdem eine unabdingliche Rolle für die Teleoperation und dem Nutzen des Roboters spielt. Über die NAOqi_bridge können Positionen der einzelnen Gelenke der Kinematiken abgefragt und gesetzt werden, was für das Fahrwerk und das Kopfgelenk direkt genutzt werdenk kann, jedoch für die Arme aber durch die hohe Anzahl an Gelenken zuerst weiter verarbeitet werden muss.
Die Rohdaten für die gewünschte Position der Kinematiken werden von der MetaQuest3 Anwendung beziehungsweise Node bereitgestellt und über das Topic */AR/position/* mit den sub-Topics */LController*, */RController* und */Head* veröffentlicht. Wiederum eine Ausnahme bildet das Fahrwerk, welches indirekt seine Daten aus den Kontroller-Knöpfen der MetaQuest3 bezieht und unter dem Topic */AR/button* veröffentlicht.
Eine Position besteht sowohl aus der aktuellen und relativen Position der Komponente zu der Brille, als auch aus der aktuellen Winkel der Komponente. Diese beiden Werte werden in der Anwendung in einem festen Format gespeichert und über die Topics, einzeln für jede Kinematik-Gruppe veröffentlicht. Die dazugehörige Nachricht ist in Listing Algorithmus 4.1 zu sehen.

Algorithmus 4.1: Positions Topic

```
float64 x
float64 y
float64 z
float64 roll
float64 pitch
float64 yaw
```

Im Gegensatz dazu die Winkel um die Achsen x, y und z, zu benennen ist es in der Robotik üblich diese als Roll, Pitch und Yaw zu bezeichnen. Diese Werte sind daher auch in der Nachricht als *roll*, *pitch* und *yaw* zu finden. Sie werden statt in Winkel 0-360Grad in Radiant angegeben und können von -π bis π reichen. Die Positionen werden ebenfalls gemäß dem Standard in Metern angegeben und können in dem im Headset genullten Koordinatensystem frei bewegt werden.
Wieder die Ausnahme bildet das Fahrwerk, welches nicht in einem 3D Raum bewegt wird, sondern nur in einer Ebene. Daher wird die Position des Fahrwerks nur in x und y angegeben und die Winkel nur in yaw. Die restlichen Werte werden ignoriert und für die Berechnung auf 0 gesetzt.

4.3 Streaming-Node

(nicht vollständig Implementiert)

Für die Übertragung der Kamerabilder wie sie in Unterabschnitt 4.2.1 beschrieben sind, wird eine eigene Node für die MetaQuest3 erstellt. Sie hat die einzig und alleinige Aufgabe die Kamerabilder des Pepper, aus der, von der NAOqi_bridge zur Verfügung gestellten Topic
/pepper_robot/camera/front/compressed aufzunehmen, in einem möglichst kleinen Zwischenspeicher (Buffer) zu speichern und mit möglichst geringer Latenz an die Brille weiter zu senden.
Dazu wird in der AR-App (die ROS Node), die Topic */pepper_robot/camera/front/image_raw/compressed* abonniert. Die Bilder werden von dieser Topic so schnell wie es ROS ermöglicht abgegriffen, und ind den Buffer geschrieben. Dadurch kann es zwar vorkommen, dass das selbe Bild mehrere male aus der Topic gelesen wird, jedoch ist es eine einfache Methode die Bilder zu übertragen und die Latenz gering zu halten um wiederum den pseudo-Stream so flüssig und responsiv zu gestalten wie es nur möglich ist. Dadurch ist der Limitierende Faktor das darunter liegende System oder primär die Bandbreite der Verbindung im Netzwerk.
Der eben erwähnte Buffer ist nicht zwingend notwendig, aber für die Anwendung von Vorteil, da die Bilder so nicht direkt an das Display weitergegeben werden müssen, sondern in einem Zwischenspeicher gespeichert werden können. Dadurch können die kleinen Schwankungen in der Übertragungsgeschwindigkeit der Bilder ausgeglichen werden und die Bilder können so gleichmäßig wie möglich angezeigt werden. Jedoch bedeutet ein Buffer auch, dass die Bilder nicht in Echtzeit angezeigt werden können, sondern immer mit einer kleinen Verzögerung, weswegen der Buffer sebst so klein wie möglich gehalten werden sollte. Daher wurde sich in diesem Projekt für einen Ring-Buffer entschieden, welcher lediglich die letzten 5 Bilder speichert, da eine Verzögerung von 5 Bildern (bei 30Frames per Second (FPS) etwa 0.166s) zwar schon als unnatürlich und störend wahrgenommen werden kann, aber in diesem Aufbau in Kauf genommen wird. Auf Verzögerungen durch die Trägheit der Kinematiken des Roboters wird in im späterne Verlauf unter Abschnitt 4.4 nocheinmal separat eingegangen.
Eine Besonderheit der Node ist, dass Sie nicht klassisch unter *rclcpp* oder *rclpy* implementiert wird, sondern über den ROS-Transmission Control Protocol (TCP)-Connector.

Dieser ist ein Modul, bereitgestellt durch Unity, welches die Kommunikation zwischen ROS und Unity ermöglicht. Dadruch werden die Nachrichten der ROS-Topics direkt in Unity empfangen und können dort unter der Verwendung von C# weiterverarbeitet werden. Durch die Nutzung der Programiersprache C# verläuft das Empfangen und Speichern der Bilder in Unity deutlich schneller und effizienter als in Python, was wiederum die Latenzen der Bilder reduziert und die Anwendung flüssiger macht.
Weiter wird ein Modul mit dem Namen *openXR* genutzt, welches die Kommunikation zwischen Unity und der MetaQuest3 erheblich erleichter und standartisiert. Weiter ermöglicht es die App auf einem Windows-Rechner zu entwickeln und auszuführen, sodass die App nicht auf der Brille selbst ausgeführt wird. Dadurch kann Rechenleistung und Speicherplatz auf der Brille gespaart werden und die App kann auf einem leistungsstärkeren Rechner ausgeführt werden.

Für die Erstellung der App wird ein neues Unity-Projekt erstellt und die benötigten Module *openXR* und *ROS-TCP-Connector* hinzugefügt. Die Module werden in Unity als *.unitypackage* Datei bereitgestellt und werden über den Menüpunkt *Assets* → *Import Package* → *Custom Package* hinzugefügt werden.
Der Kern der Unity-App besteht aus der in Algorithmus 4.2 gezeigten Klasse. Diese hier beispielhaft für den rechten Controller gezeigte Klasse, ist für die Aufnahme der Daten des Controllers zuständig und veröffentlicht diese in einem ROS-Topic. Die Klasse wird in Unity als *MonoBehaviour* implementiert und über die Funktionen *Start()* und *Update()* gesteuert. Die Funktion *Start()* wird beim Start der App aufgerufen und initialisiert die Verbindung zu ROS und registriert den Publisher für die Nachrichten. Die Funktion *Update()* wird in jedem Frame aufgerufen und überprüft ob die Nachrichten veröffentlicht werden sollen. Die Nachrichten werden dabei in einem festen Intervall von 0.1s veröffentlicht, was wiederum der maximalen Frequenz des Peppers entspricht, und somit eine flüssige und efiziente Bewegung ermöglicht.

Algorithmus 4.2: Abgreifen der Controller Daten

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.Geometry;

public class ControllerPositionPublisher : MonoBehaviour
{
```

```
ROSConnection ros;
public string RController = "/AR/RController";
public GameObject rightController;

public float publishMessageFrequency = 0.1f;

void Start()
{
    ros = ROSConnection.GetOrCreateInstance();
    ros.RegisterPublisher<PoseStampedMsg>(RController);
    timeElapsed = 0;
}

void Update()
{
    timeElapsed += Time.deltaTime;

    if (timeElapsed > publishMessageFrequency)
    {
        ros.Publish(topicName,
            GetControllerPose(rightController));

        timeElapsed = 0;
    }
}

PoseStampedMsg GetControllerPose(GameObject controller)
{
    PoseStampedMsg poseMsg = new PoseStampedMsg();
    poseMsg.header.frame_id = "unity";
    poseMsg.header.stamp
        = ROSConnection.GetOrCreateInstance()
            .GetROSTime();

    poseMsg.pose.position = new PointMsg(
```

```
                controller.transform.position.x,
                controller.transform.position.y,
                controller.transform.position.z
            );

            poseMsg.pose.orientation = new QuaternionMsg(
                controller.transform.rotation.x,
                controller.transform.rotation.y,
                controller.transform.rotation.z,
                controller.transform.rotation.w
            );

            return poseMsg;
        }
    }
```

Das Bild der Kamera ist dagegen hat zu der einfachen Struktur des publishen der Controller-Daten eine weitere Komponente, den Ring-Buffer. Auch hier werden wieder die üblichen ROS Funktionen initialiert. In der *Start()* Funktion wird der Publisher für die Kamerabilder (z.9) registriert und der Buffer, wie in Algorithmus 4.3(z.24) mit kleinen temporären Texturen gefüllt, initialisiert. Dazu werden Behelfsvariablen für die Größe des Buffers und die aktuelle Position im Buffer erstellt. In der *Update()* Funktion wird dann in jedem Frame das Bild der Kamera in den Buffer geschrieben und die Nachricht veröffentlicht. Sobald in der Topic der Kamera ein Update stattfindet, wird die Funktion *UpdateImage()* aufgerufen und das Bild aus der Nachricht in den Buffer geschrieben (z.34) und dessen Index erhöht (z.35). Zum schluss der Methode wird das Bild aus dem Buffer in die Textur des Renderers geschrieben und angezeigt (z.37).

Algorithmus 4.3: Abgreifen der Kamerabilder

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.Sensor;
using System;
using System.Runtime.InteropServices;
```

```
public class ImageSubscriber : MonoBehaviour
{
    ROSConnection ros;
    public string topicName
        = "/pepper_robot/camera/front/image_raw/compressed";
    public Renderer imageRenderer;

    private Texture2D[] imageBuffer;
    private int bufferSize = 5;
    private int currentIndex = 0;

    void Start()
    {
        ros = ROSConnection.GetOrCreateInstance();
        ros.Subscribe<ImageMsg>(topicName, UpdateImage);

        imageBuffer = new Texture2D[bufferSize];
        for (int i = 0; i < bufferSize; i++)
        {
            imageBuffer[i]
                = new Texture2D(2,
                    2,
                    TextureFormat.RGB24,
                    false);
        }
    }

    void UpdateImage(ImageMsg imageMessage)
    {
        Texture2D texture
            = new Texture2D(imageMessage.width,
                imageMessage.height,
                TextureFormat.RGB24,
                false);
        texture.LoadRawTextureData(imageMessage.data);
```

```
            texture.Apply();

            imageBuffer[currentIndex] = texture;
            currentIndex = (currentIndex + 1) % bufferSize;

            imageRenderer.material.mainTexture
                = imageBuffer[currentIndex];
        }
    }
```

4.4 Berechnungs-Node

(nicht vollständig Implementiert)

Die Berechnung der Kinematiken und der genauen Positionen der einzelnen Motoren werden über inverse Kinematiken gelöst, welche von der *pyBullet* Bibliothek bereitgestellt werden. Dazu wird einen individuelle Node erstellt, welche die *pyBullet* Bibliothek nutzt und die berechneten Positionen an die NAOqi_bridge weitergibt. Die Node hat dabei eine Art Middleware-Funktion und übernimmt die Umrechnungen zwischen der MetaQuest3 und der NAOqi_bridge.
Die Node bindet die *pyBullet* und die *NAOqi_bridge* Bibliotheken ein. Für die Berechnung abonniert sie als Input die Topics der MetaQuest3: */AR/position/LController*, */AR/position/RController*, */AR/position/Head* und */AR/button*. Der Output an die NAOqi_bridge ist dagegen etwas breiter gefächtert, und ist eine direkte Kommunikation an die API.

Die Node besteht nicht wie jede Node aus einem Publisher-Teil und einem Subscriber-Teil. Sondern nur dem Subscriber-Teil, da die Node nur die Positionen der MetaQuest3 empfängt und an die NAOqi_bridge weitergibt. Die Berechnung der Positionen und die Kommunikation mit der NAOqi_bridge erfolgt in einem einzigen Teil der Node. Dazu ist maßgebend die Funktion *callback* von Relevanz, da diese Aufgerufen wird sobald eine Nachricht in einem der abonnierten Topics ankommt. Als zweite zentrale Komponente dient hier die *pyBullet* Bibliothek. Sie muss zunächst initialisiert werden und wie in Algorithmus 4.4 gezeigt die Kinematiken des Roboters über die URDF-Datei des Peppers einlesen.

Algorithmus 4.4: Berechnungs-Node-Initialisierung von PyBullet

```
import pybullet as p

p.connect(p.DIRECT)

robot = p.loadURDF("pepper.urdf",
        [0, 0, 0],
        useFixedBase=True)
```

Sobald eine Nachricht empfangen wird, werden die Nachrichten aus den Arrays der Nachricht der MetaQuest3 entnommen und durch nach der Berechnung der inversen Kinematiken in die passenden Werte für die NAOqi_bridge umgerechnet. Das Array der Positionen der MetaQuest3 wird wie in Algorithmus 4.1 beschrieben aufgebaut und enthält die Werte für x, y, z, roll, pitch und yaw, in genau dieser Reihenfolge. Die Werte werden dabei direkt per `rospy.Subscriber("[Topic]"), [Type], [callback]` abonniert und in der Funktion *callback* verarbeitet. Hier werden für jede Gelenkgruppe (Arme und Genick) eine separate callback-Funktion aufgerufen. Diese Gruppen werden zu Begin in einem Array aus den einzelnen Gelenken zusammengefasst und der jeweiligen Funktion übergeben. Aus der URDF-Datei des Pepper werden die Gelenke und deren Positionen ausgelesen (Algorithmus 4.5) und in einem Dictionary gespeichert. Die Positionen der Gelenke werden dann in einem weiteren Dictionary gespeichert und an die *pyBullet* Bibliothek übergeben. Aus der URDF-Datei des Pepper können die Namen der Gelenke gelesen werden, und die Gelenke können über die Namen angesprochen werden und den Gruppen zugeordnet werden. Dazu wird der Code aus Algorithmus 4.5 und Algorithmus 4.6 genutzt.

Algorithmus 4.5: Ausschnitt der URDF-Datei des Pepper

```
<joint name="HeadYaw" type="continuous">
    <parent link="base_link"/>
    <child link="HeadYaw_link"/>
    <origin xyz="0 0 0.126" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
    <limit lower="-2.0857" upper="2.0857"/>
</joint>
```

Algorithmus 4.6: Gelenk-Grupperierung

```
joints = {
```

```
        "Head": ["HeadYaw", "HeadPitch"],
        "LArm": ["LShoulderPitch",
                 "LShoulderRoll",
                 "LElbowYaw",
                 "LElbowRoll",
                 "LWristYaw"],
        "RArm": ["RShoulderPitch",
                 "RShoulderRoll",
                 "RElbowYaw",
                 "RElbowRoll",
                 "RWristYaw"]
    }
```

Damit kann über `joints["Gruppe"]` auf die einzelnen Gelenke zugegriffen werden und die Positionen der Gelenke gelesen und gesetzt werden. Die Berechnung der inversen Kinematiken erfolgt über die *pyBullet* Bibliothek und die Funktion `p.calculateInverseKinematics([Robot], [Gelenk-Gruppe], [Position], [Orientierung])`. Dabei können die Positionen und Orientierungen der Gelenke direkt in der Funktion übergeben werden und die Funktion gibt die berechneten Positionen der Gelenke zurück.
Diese Daten sind nun so aufbereitet, dass sie direkt an die NAOqi_bridge übergeben werden können. Dazu wird zunächst der eine Session des Roboters geöffnet und dieser der Service *ALMotion* über `m_service = session.service(ALMotion)` gestartet. Die Positionen der Gelenke werden dann über den Service und der Funktion `m_service.setAngles([Name], [Winkel], [MaxGeschwindigkeit])` an die NAOqi_bridge übergeben und die Gelenke des Roboters werden in die gewünschte Position gebracht. Unter den Namen werden in diesem Projekt die folgenden Gelenke angesprochen:

- */pepper_robot/HeadYaw* - Die Position des Kopfes um die Y-Achse.
- */pepper_robot/HeadPitch* - Die Position des Kopfes um die X-Achse.
- */pepper_robot/[R,L]ShoulderPitch* - Die Position des Schultergelenks um die X-Achse.
- */pepper_robot/[R,L]ShoulderRoll* - Die Position des Schultergelenks um die Y-Achse.
- */pepper_robot/[R,L]ElbowYaw* - Die Position des Ellenbogengelenks um die Y-Achse.
- */pepper_robot/[R,L]ElbowRoll* - Die Position des Ellenbogengelenks um die X-Achse.

- */pepper_robot/[R,L]WristYaw* - Die Position des Handgelenks um die Y-Achse.
- */pepper_robot/WheelFL* - Die Position des linken Vorderrades.
- */pepper_robot/WheelFR* - Die Position des rechten Vorderrades.
- */pepper_robot/WheelB* - Die Position des hinteren Rades.

Diese Daten werden mit der maximalen Frequenz des Peppers von 10Hz an die NAOqi_bridge übergeben womit bedingt durch die Bewegung des Users und die Trägheit eine Art Interpolation geschaffen wird und die Bewegungen des Roboters flüssiger und natürlicher wirken.

4.5 Fahrwerk-Node

(nicht vollständig Implementiert)

Das Fahrwerk des Pepper wird nicht direkt über die MetaQuest3 gesteuert, sondern über die Knöpfe der MetaQuest3. Dadurch kann hier weder mit Positionen gearbeitet werden, noch mit inversen Kinematiken. Ebenfalls platziert in der Berechnungs-Node, können die Variablen wie die `session` und der `m_service` auch für diese Berechnungen genutzt werden. Für diesen Ansatz wird jedoch eine separate callback-Funktion benötigt. Sie reagiert auf die */AR/button* Nachrichten und lässt den Roboter entsprechend der Knöpfe bewegen. Hierzu werden keine aufwendigen Berechnungen benötigt, sondern nur die Knöpfe ausgelesen und eine Translation oder Rotation des Roboters durchgeführt.
Die Knöpfe der MetaQuest3 sind in diesem Projekt wie folgt belegt:

- *A* - Vorwärtsbewegung
- *B* - Rückwärtsbewegung
- *X* - Drehung nach links
- *Y* - Drehung nach rechts

Auf die Knöpfe wird über die *OVRInput*-Klasse von Unity zugegriffen und als einfacher charakter in der Topic übergeben. Dadurch kann zwar nur ein einzelner Befehl ausgeführt werden, was jedoch über die Tastenbelegung eher durch den User begrenzt wird. Auch das

Fahrwerk kann über den `m_service` angesprochen werden und die Räder des Roboters können über die Funktion `m_service.moveToward([X], [Y], [Theta])` in die gewünschte Richtung bewegt werden. Die Werte für X und Y sind dabei die Geschwindigkeiten der Vorderräder und der Wert für Theta ist die Geschwindigkeit des hinteren Rades. Die Werte können dabei zwischen -1 und 1 liegen, wobei -1 die maximale Geschwindigkeit in die eine Richtung und 1 die maximale Geschwindigkeit in die andere Richtung bedeutet. Auch hier wird die Funktion mit 10Hz aufgerufen und der Roboter bewegt sich nur solange einer der Knöpfe gedrückt wird. Wird ein Knopf entlastet stoppt der Roboter. Die Bewegung des Roboters wird dabei durch die Trägheit des Roboters und die Reaktionszeit des Users begrenzt und wirkt so natürlicher und flüssiger.

Kapitel 5

Anwendungsgebiete

Die Teleoperation humanoider Roboter bietet eine Vielzahl von Einsatzmöglichkeiten in unterschiedlichen Bereichen. Im Folgenden werden einige der wichtigsten Anwendungsgebiete beschrieben:

5.0.1 Medizin und Pflege

In der Medizin und Pflege können humanoide Roboter zur Unterstützung von Ärzten und Pflegepersonal eingesetzt werden. Sie können in gefährlichen oder infektiösen Umgebungen arbeiten, wodurch das Risiko für medizinisches Personal minimiert wird. Zudem können Roboter Routineaufgaben übernehmen, was die Arbeitsbelastung des Personals verringert und mehr Zeit für die direkte Patientenbetreuung ermöglicht.

5.0.2 Industrielle Fertigung

In der industriellen Fertigung können humanoide Roboter für Aufgaben eingesetzt werden, die für Menschen gefährlich oder ergonomisch ungünstig sind. Sie können in gefährlichen Umgebungen arbeiten, schwere Lasten heben oder repetitive Aufgaben mit hoher Präzision ausführen. Durch die Teleoperation können Experten aus der Ferne eingreifen und die Roboter steuern, was die Flexibilität und Effizienz der Fertigungsprozesse erhöht.

5.0.3 Katastrophenhilfe und Rettungseinsätze

Humanoide Roboter können in Katastrophengebieten eingesetzt werden, um Menschen zu retten oder gefährliche Situationen zu erkunden. Sie können durch Trümmer navigieren,

Verletzte finden und erste Hilfe leisten. Durch die Teleoperation können Rettungskräfte aus sicherer Entfernung arbeiten und dennoch präzise und effektiv handeln.

5.0.4 Raumfahrt

In der Raumfahrt können humanoide Roboter für Außenbordeinsätze (EVAs) und Wartungsarbeiten an Raumstationen oder Satelliten eingesetzt werden. Sie können in extremen Umgebungen arbeiten, die für Menschen gefährlich sind, und komplexe Aufgaben mit hoher Präzision ausführen. Die Teleoperation ermöglicht es, dass die Roboter von der Erde aus gesteuert werden, wodurch das Risiko für Astronauten minimiert wird.

5.0.5 Bildung und Forschung

Humanoide Roboter können in Bildungseinrichtungen und Forschungslabors eingesetzt werden, um Experimente durchzuführen und komplexe Konzepte zu demonstrieren. Sie können als Lehrassistenten dienen und Schülern und Studenten interaktive und praxisnahe Lernerfahrungen bieten. In der Forschung können Roboter zur Untersuchung neuer Technologien und zur Entwicklung innovativer Anwendungen verwendet werden.

5.0.6 Heim- und Unterhaltungselektronik

Im Bereich der Heim- und Unterhaltungselektronik können humanoide Roboter als persönliche Assistenten oder Unterhaltungsgeräte eingesetzt werden. Sie können Aufgaben im Haushalt übernehmen, wie das Aufräumen oder das Zubereiten von Mahlzeiten, und bieten interaktive Unterhaltungsmöglichkeiten, wie das Spielen von Spielen oder das Führen von Gesprächen. Durch die Teleoperation können Benutzer ihre Roboter aus der Ferne steuern und ihnen Anweisungen geben.

5.0.7 Kundendienst und Gastgewerbe

Humanoide Roboter können im Kundendienst und im Gastgewerbe eingesetzt werden, um Kunden zu begrüßen, Informationen bereitzustellen oder Bestellungen entgegenzunehmen. Sie können in Hotels, Restaurants oder Geschäften arbeiten und den Service verbessern, indem sie rund um die Uhr verfügbar sind und in verschiedenen Sprachen kommunizieren können. Die Teleoperation ermöglicht es, dass Menschen aus der Ferne eingreifen und bei Bedarf Unterstützung leisten können.

Die Anwendungsgebiete der Teleoperation humanoider Roboter sind vielfältig und bieten zahlreiche Möglichkeiten, die Effizienz und Sicherheit in verschiedenen Bereichen zu erhöhen. Die vorliegende Arbeit trägt dazu bei, diese Vision durch die Entwicklung einer VR-basierten Steuerung für den humanoiden Roboter Pepper weiter voranzutreiben.

Kapitel 6

Fazit

Die Umsetzung des Projekts *Pepper VR – Teleoperation eines humanoiden Roboters auf Basis der Analyse menschlicher Bewegung* stellte eine anspruchsvolle Herausforderung dar, die tiefgreifende Kenntnisse in verschiedenen Bereichen der Informatik und Robotik erforderte. Trotz der ambitionierten Ziele, die Verbindung zwischen der MetaQuest 3 VR-Brille und dem humanoiden Roboter Pepper herzustellen und die Steuerung von Pepper durch die VR-Controller zu ermöglichen, stießen wir auf mehrere technische und organisatorische Hürden. Welche die Umsetzung leider schlussendlich nicht möglich machten.

Die wichtigsten Erkenntnissen und Schlussfolgerungen sind:

6.1 Technische Herausforderungen

Die Implementierung der Steuerung eines humanoiden Roboters über eine VR-Brille wie die MetaQuest 3 erfordert eine präzise und latenzfreie Datenübertragung. Probleme mit der Netzwerkverbindung, Synchronisation und Verzögerungen führten dazu, dass die Steuerung von Pepper nicht in der gewünschten Präzision umgesetzt werden konnte.

6.2 Komplexität der Integration

Die Integration von Unity und ROS über den ROS-TCP-Connector stellte sich als komplexer als erwartet heraus. Obwohl Unity eine benutzerfreundliche Entwicklungsumgebung bietet, waren die Anforderungen an die ROS-Integration höher als erwartet, insbesondere im Hinblick auf Echtzeitfähigkeit und die nahtlose Zusammenarbeit der verschiedenen Softwarekomponenten.

6.3 Wahl von Unity

Die Entscheidung für Unity als Entwicklungsplattform wurde aufgrund ihrer weiten Verbreitung, der umfangreichen Dokumentation und der großen Entwicklergemeinschaft getroffen. Unity bietet viele vorgefertigte Lösungen und Plugins, die die Entwicklung beschleunigen können. Dennoch stellte sich heraus, dass für diese spezielle Aufgabe tiefergehende Kenntnisse im Umgang mit der Unity-ROS-Integration erfordert, als in der gegebenen Zeit erlernbar war.

6.4 Potenzial für zukünftige Arbeiten

Trotz der Herausforderungen zeigt das Projekt deutlich das Potenzial, das in der Kombination von VR und humanoider Robotik liegt. Künftige Arbeiten könnten von den gemachten Erfahrungen profitieren und die entwickelten Ansätze weiter verfolgen. Welche Schritte zur Weiterführung notwendig sind, werden auch im nachfolgenden Kapitel genauer erklärt.

Insgesamt war das Projekt eine wertvolle Lernerfahrung, die Einblicke in die komplexe Welt der Robotik und Virtual Reality ermöglichte. Die gesammelten Erfahrungen und Erkenntnisse werden als Grundlage für zukünftige Entwicklungen dienen, und wir sind zuversichtlich, dass die Vision einer nahtlosen Teleoperation von humanoiden Robotern über VR-Systeme mit weiterem Engagement und Forschung realisiert werden kann.

Kapitel 7

Fortsetzung des Projekts

Um das Projekt *Pepper VR – Teleoperation eines humanoiden Roboters auf Basis der Analyse menschlicher Bewegung* in Zukunft erfolgreich umzusetzen, sind mehrere wichtige Schritte und Verbesserungen erforderlich. Nach der Analyse der bisherigen Herausforderungen und der gewonnenen Erkenntnisse haben wir folgende Empfehlungen für die Fortsetzung des Projekts zusammengestellt:

7.1 Optimierung der technischen Infrastruktur

Eine der größten Herausforderungen war die präzise und latenzfreie Datenübertragung zwischen der MetaQuest 3 VR-Brille und Pepper. Zur Verbesserung der technischen Infrastruktur sollten folgende Maßnahmen ergriffen werden:

- **Verbesserung der Netzwerkverbindung**: Sicherstellen einer stabilen und schnellen Netzwerkverbindung, möglicherweise durch den Einsatz von dedizierten Netzwerken oder der Optimierung der bestehenden Infrastruktur.
- **Reduzierung der Latenzzeiten**: Implementierung von Echtzeit-Optimierungen sowohl auf der Hardware- als auch auf der Softwareseite, um die Verzögerungen bei der Datenübertragung zu minimieren.

7.2 Erweiterung der Softwareintegration

Die Integration von Unity und ROS über den ROS-TCP-Connector muss weiter verfeinert und genauer recherchiert werden. Hier sind die wesentlichen Schritte:

- **Tiefere ROS-Integration**: Entwicklung zusätzlicher ROS-Nodes und -Services, um eine nahtlosere Kommunikation zwischen Unity und dem Robot Operating System zu gewährleisten.
- **Fehlersuche und Debugging**: Intensive Fehlersuche und Debugging der bestehenden Implementierung, um die Ursachen für Synchronisationsprobleme zu identifizieren und zu beheben.

7.3 Testen und Validieren der VR-Steuerung

Um die Steuerung von Pepper über die MetaQuest 3 VR-Brille erfolgreich umzusetzen, sind umfangreiche Tests und Validierungen notwendig:

- **Erstellung von Testfällen**: Entwicklung spezifischer Testfälle, die die verschiedenen Aspekte der VR-Steuerung abdecken, um sicherzustellen, dass alle Funktionen wie erwartet arbeiten.
- **Benutzerstudien**: Durchführung von Benutzerstudien, um die Benutzerfreundlichkeit und Effektivität der VR-Steuerung zu bewerten und basierend auf dem Feedback Verbesserungen vorzunehmen.

7.4 Schulung und Dokumentation

Die erfolgreiche Umsetzung des Projekts erfordert gut geschulte Teammitglieder und umfassende Dokumentation:

- **Schulung der Teammitglieder**: Regelmäßige Schulungen und Workshops für das Team, um sicherzustellen, dass alle Mitglieder die notwendigen Kenntnisse und Fähigkeiten haben, um mit den verwendeten Technologien effektiv zu arbeiten.
- **Erstellung umfassender Dokumentation**: Dokumentation aller Implementierungs- und Testprozesse, um eine klare und nachvollziehbare Basis für zukünftige Arbeiten zu schaffen.

7.5 Langfristige Wartung und Weiterentwicklung

Um das Projekt langfristig erfolgreich zu halten, sollten regelmäßige Wartung und Weiterentwicklungen eingeplant werden:

- **Regelmäßige Updates**: Kontinuierliche Aktualisierung der verwendeten Software und Hardware, um mit den neuesten Entwicklungen Schritt zu halten und die Stabilität und Sicherheit zu gewährleisten.
- **Weiterentwicklung der Funktionen**: Fortlaufende Erweiterung der Funktionen und Fähigkeiten des Systems, basierend auf den Bedürfnissen und Rückmeldungen der Benutzer.

Durch die Umsetzung dieser Maßnahmen kann das Projekt *Pepper VR* erfolgreich weitergeführt werden, um das volle Potenzial der Kombination von Virtual Reality und humanoider Robotik auszuschöpfen. Die gewonnenen Erkenntnisse und Erfahrungen bilden eine solide Grundlage für zukünftige Entwicklungen und Innovationen in diesem spannenden und zukunftsträchtigen Bereich.

Literatur

Autodesk [2021a]. *3ds Max*. https://www.autodesk.com/products/3ds-max/overview.

— [2021b]. *Autodesk Maya*. https://www.autodesk.com/products/maya/overview.

Bittel, Andreas [Accessed: 2024-05-05]. *Vorlesung: Roboterkinematik*. https://www-home.htwg-konstanz.de/~bittel/msi_robo/Vorlesung/02_Roboterkinematik.pdf [siehe S. 30].

Carnegie Mellon University [2024]. *NAOqi Documentation*. https://www.cs.cmu.edu/~cga/nao/doc/reference-documentation/dev/naoqi/index.html.

cplusplus.com [Accessed: 2024-05-05]. *C++ Libraries*. https://www.cplusplus.com/doc/oldtutorial/libraries/ [siehe S. 26].

Dante [Jan. 2010]. *Webseite der* Deutschsprachige Anwendervereinigung TeX e.V. http://www.dante.de.

Elektronik-Kompendium.de [o. D.] *Publish-Subscribe-Modell*. https://www.elektronik-kompendium.de/sites/net/2204051.htm. [Online; accessed 22-April-2024] [siehe S. 6].

Foundation, Blender [2021]. *Blender*. https://www.blender.org/.

Games, Epic [2021a]. *Blueprints Visual Scripting*. https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/index.html [siehe S. 38].

— [2021b]. *Unreal Engine Documentation*. https://docs.unrealengine.com/ [siehe S. 38].

GitHub user: dbddqy [Accessed: 2024-05-05]. *Visual Kinematics GitHub Repository*. https://github.com/dbddqy/visual_kinematics.

Heilig, Morton [1962]. »Sensorama Simulator«. Pat. U.S. Patent 3,050,870 [siehe S. 35].

Huang, Wen-Hao David und Shu-Sheng Liaw [2018]. »An analysis of learners' intentions toward virtual reality learning based on constructivist and technology acceptance approaches«. In: *International Review of Research in Open and Distributed Learning* [siehe S. 36].

ITWissen.info [o. D.] *Publish-Subscribe-Modell (publish-subscribe)*. `https://www.itwissen.info/Publish-Subscribe-Modell-publish-subscribe.html`. [Online; accessed 22-April-2024] [siehe S. 5].

Knuth, Donald E. [1984]. *The TEXbook*. Addison-Wesley.

Lamport, Leslie [1995]. *Das LaTeX Handbuch*. Addison-Wesley.

Lanier, Jaron [1992]. *Virtual Reality: The Revolutionary Technology of Computer-Generated Artificial Worlds - and How It Promises to Transform Society*. Simon & Schuster [siehe S. 36].

Luckey, Palmer [2012]. *Oculus Rift: Step into the Game*. `https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game` [siehe S. 36].

Meta [2023]. *Meta Quest 3*. `https://www.meta.com/quest/quest-3/` [siehe S. 36].

Microsoft Azure [Accessed: 2024-05-05]. *What is Middleware?* `https://azure.microsoft.com/en-gb/resources/cloud-computing-dictionary/what-is-middleware/` [siehe S. 15].

Open Source Robotics Foundation, Inc. [2022]. *COLCON Documentation*. Accessed: 2024-05-16. url: `https://colcon.readthedocs.io/en/released/` [siehe S. 25].

Python Software Foundation [Accessed: 2024-05-05]. *About Python*. url: `https://www.python.org/about/` [siehe S. 27].

Rizzo, Albert A. und Steven T. Koenig [2017]. »Is clinical virtual reality ready for primetime?« In: *Neuropsychology* [siehe S. 36].

ROS Documentation Contributors [Accessed: 2024-05-05[a]]. *ROS - About Interfaces*. `https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html` [siehe S. 18].

— [Accessed: 2024-05-05[b]]. *ROS - About Launch*. `https://docs.ros.org/en/humble/Concepts/Basic/About-Launch.html` [siehe S. 22].

— [Accessed: 2024-05-05[c]]. *ROS - About Nodes*. `https://docs.ros.org/en/humble/Concepts/Basic/About-Nodes.html` [siehe S. 16].

ROS Documentation Contributors [Accessed: 2024-05-05[d]]. *ROS - About Parameters*. https://docs.ros.org/en/humble/Concepts/Basic/About-Parameters.html [siehe S. 22].

— [Accessed: 2024-05-05[e]]. *ROS - About Topics*. https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html [siehe S. 18].

— [Accessed: 2024-05-05[f]]. *ROS - Basic Concepts*. https://docs.ros.org/en/humble/Concepts/Basic.html [siehe S. 16].

ROS-TCP-Endpoint Documentation [o. D.] https://github.com/Unity-Technologies/ROS-TCP-Endpoint. Accessed: 2024-05-14 [siehe S. 39].

ROSIntegration Contributors [o. D.] *ROSIntegration Plugin for Unreal Engine*. https://github.com/code-iai/ROSIntegration. Accessed: 2024-05-14 [siehe S. 42, 44].

Sutherland, Ivan E. [1968]. »A head-mounted three-dimensional display«. In: *Proceedings of the Fall Joint Computer Conference* [siehe S. 35].

Technologies, Unity [2021a]. *OpenXR in Unity*. https://docs.unity3d.com/Manual/com.unity.xr.openxr.html [siehe S. 38].

— [2021b]. *Unity Asset Store: Oculus Integration*. https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022 [siehe S. 37].

— [2021c]. *Unity Documentation*. https://docs.unity3d.com/Manual/index.html [siehe S. 37].

Unity-ROS-TCP-Connector Documentation [o. D.] https://github.com/Unity-Technologies/ROS-TCP-Connector. Accessed: 2024-05-14 [siehe S. 39, 41].

University, Carnegie Mellon [Year of access]. *Nao Developer Documentation: C++ SDK Installation Guide*. https://www.cs.cmu.edu/~cga/nao/doc/reference-documentation/dev/cpp/install_guide.html. Accessed: April 22, 2024 [siehe S. 7].

VanderPlas, Jake [2016]. *Python for Data Science Handbook*. O'Reilly Media. url: https://jakevdp.github.io/PythonDataScienceHandbook/ [siehe S. 27].

Wefers, Frank [März 2015]. »Bewegungsprädiktion in der Echtzeit-Auralisierung dynamischer Schallfelder«. In: [Siehe S. IV, 31].

Wikipedia [Accessed: 2024-05-05[a]]. *Denavit–Hartenberg parameters — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/DenavitâĂŞHartenberg_parameters [siehe S. IV, 30].

— [Accessed: 2024-05-05[b]]. *Interpolation (Mathematik) — Wikipedia, Die freie Enzyklopädie*. URL: https://de.wikipedia.org/wiki/Interpolation_(Mathematik) [siehe S. IV, 32, 33].

— [2024]. *Inverse Kinematik — Wikipedia, Die freie Enzyklopädie*. URL: https://de.wikipedia.org/wiki/Inverse_Kinematik.

— [o. D.] *Publish–subscribe pattern*. https://en.wikipedia.org/wiki/PublishâĂŞsubscribe_pattern. [Online; accessed 22-April-2024] [siehe S. 4].

Wikipedia-Autoren [Accessed: 2024-05-05]. *Trajektorie (Physik)*. https://de.wikipedia.org/wiki/Trajektorie_(Physik) [siehe S. 30].

Wikipedia contributors [Accessed: 2024-05-05[a]]. *C++*. https://en.wikipedia.org/wiki/C++ [siehe S. 26].

— [Accessed: 2024-05-05[b]]. *Python (Programmiersprache)*. https://de.wikipedia.org/wiki/Python_(Programmiersprache) [siehe S. 26].