

## **NaoVR - Nao Access Over Virtual Reality**

### **Studienarbeit**

für die Prüfung zum

**Bachelor of Science**

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Janis Schneider, Dennis Jahnke**

<b>Abgabedatum</b>	08.06.2020
<b>Bearbeitungszeitraum</b>	27 Wochen
<b>Matrikelnummer</b>	1036436 / 4846724
<b>Kurs</b>	TINF17B4 / TINF17B3
<b>Ausbildungsfirma</b>	Arvato Infoscore / Siemens AG
<b>Betreuer</b>	Prof. Dr. Marcus Strand
<b>Gutachter der Studienakademie</b>	Prof. Dr. Marcus Strand

## **Eidesstattliche Erklärung**

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017.

Ich versichere hiermit, dass ich die Projektarbeit mit dem Titel

„NaoVR - Nao Access Over Virtual Reality“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

---

Ort, Datum

Janis Schneider, Dennis Jahnke

## **Abstract**

Diese Arbeit beschreibt die Umsetzung der direkten Steuerung eines humanoiden Roboters mittels eines HTC Vive Virtual Reality Headsets. Die Position der Arme und des Kopfes des Anwenders werden über inverse Kinematik interpretiert und an den Roboter übertragen, welcher die Pose imitiert. Die Position der HTC Vive Komponenten wird mittels SteamVR ausgelesen, über Unity auf ein virtuelles Skelett übertragen und über die ROS-Sharp Schnittstelle an ein ROS-Netzwerk weitergegeben. An dieses ROS-Netzwerk ist der Aldebaran Robotics NAOV4 als Zielplattform angeschlossen. Da die entsprechenden Bibliotheken zur Integration des NAO derzeit nur auf der ROS Indigo Distribution vollständig implementiert sind, dient eine virtuelle Maschine mit Ubuntu 14.4 als Ausgangssystem für ROS. Die Zusammenführung beider Systeme im virtuellen Raum findet auf einem Windows 10 System unter Unity statt.

Um die Atmosphäre eines Cockpits zu erzeugen, werden Bilder vom Kopf des NAO auf das Display des Headsets übertragen. Hierdurch verfügt der Benutzer über eine identische Perspektive zum NAO. Die internen Kameras des NAO konnten hierfür, aufgrund zu hoher Latenzen und zu niedriger Bildwiederholrate, nicht verwendet werden. Aus diesem Grund wurde auf externe analoge Kameras zurückgegriffen. Diese Kameras sind mit einem speziell für diesen Zweck designten Helm am Kopf des NAO befestigt und verfügen über eine eigene Spannungsversorgung und einen eigenen Funksender. Dieser sendet das Bildsignal direkt an über USB verbundene Empfänger.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>VI</b>
<b>Abkürzungsverzeichnis</b>	<b>VII</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation und Zielsetzung . . . . .	1
1.2 Stand der Technik . . . . .	2
1.3 Entwicklungsumgebung . . . . .	2
1.4 Vorgehensweise . . . . .	3
<b>2 Grundlagen</b>	<b>4</b>
2.1 NAO Roboter . . . . .	4
2.2 Robot Operating System – ROS . . . . .	6
2.3 ROS-Sharp (ROS#) . . . . .	9
2.4 Unity . . . . .	10
2.5 Virtual Reality . . . . .	10
2.6 HTC Vive . . . . .	11
2.7 Inverse Kinematik . . . . .	12
<b>3 Anforderungsanalyse</b>	<b>13</b>
<b>4 Umsetzung</b>	<b>14</b>
4.1 Inbetriebnahme von ROS . . . . .	14
4.2 Anbindung des NAO an ROS . . . . .	15
4.3 Inbetriebnahme von ROS-Sharp . . . . .	18
4.4 NAO Schnittstelle für Unity . . . . .	19
4.5 Einrichten einer VR-Umgebung in Unity . . . . .	29
4.6 Entwurf des Kamerasytems . . . . .	29
4.7 Prototypisierung der Kamera Halterung . . . . .	31
4.8 Rendern des stereoskopischen Bildes in Unity . . . . .	36
4.9 Synchronisierung von NAO Modell und Benutzer . . . . .	38
4.10 Vereinigung der Teilprojekte . . . . .	46
4.11 Benutzerführung durch States . . . . .	47
4.12 Bedienung der Software . . . . .	48

<b>5 Evaluation</b>	<b>53</b>
5.1 Untersuchung des Bottlenecks in ROS . . . . .	53
5.2 Latenz im Kamerabild . . . . .	55
5.3 Latenz in der Ausführung von Befehlen . . . . .	56
5.4 Usability Test . . . . .	57
<b>6 Fazit</b>	<b>58</b>
6.1 Rückblick . . . . .	58
6.2 Zusammenfassung . . . . .	59
6.3 Wirtschaftliche Betrachtung . . . . .	60
6.4 Ausblick . . . . .	61

<b>Literaturverzeichnis</b>	<b>LXIV</b>
-----------------------------	-------------

# Abbildungsverzeichnis

4.1	nao_full.launch (Auszug), blau markiert sind ergänzte Anwendungen . . . . .	17
4.2	SpeechController, Beispiel einer Publisher Implementierung . . . . .	22
4.3	DiagnosticConnector, Beispiel einer Subscriber Implementierung . . . . .	23
4.4	StiffnessController, Beispiel einer ServiceCall Implementierung . . . . .	24
4.5	Action Interface in ROS Quelle [a] . . . . .	26
4.6	LEDActionClient, Beispiel einer Action Client Implementierung . . . . .	27
4.7	Predefined Poses 'Crouch'(links) 'StandZero'(rechts) Quelle: [b] . . . . .	28
4.8	Version 1 der Kamera Halterung. . . . .	32
4.9	Version 2 der Kamera Halterung. . . . .	33
4.10	Test der Thingiverse inspirierten Halterung. . . . .	34
4.11	Version 4 der Kamera Halterung. . . . .	35
4.12	Version 5 der Kamera Halterung. . . . .	36
4.13	NAO Modell mit Skelett. . . . .	39
4.14	IK Skript von CreateThis. . . . .	40
4.15	Beispielhafter Aufbau des Gelenkwinkelszenarios. . . . .	43
4.16	KinematicArmModel in der Initialpose. . . . .	43
4.17	NodeData des rechten Schultergelenks. . . . .	44
4.18	KinematicArmModel während der Laufzeit. . . . .	44
4.19	Hierarchie des CameraRig. . . . .	45
4.20	Zusammenführung aller Komponenten. . . . .	46
4.21	Befehle zum Starten aller notwendigen ROS-Anwendungen . . . . .	48
4.22	UI Initialzustand . . . . .	49
4.23	Benutzer hat die Position eingenommen . . . . .	50
4.24	Benutzer hat seine Höhe kalibriert . . . . .	50
4.25	Benutzer hat die Synchronisierung gestartet . . . . .	51
4.26	Tastenbelegung der Anwendung . . . . .	52
5.1	ROS und Unity auf räumlich getrennten Systemen verbunden durch den lokal gehosteten Server der ROSBridge. . . . .	54
5.2	Versuchsaufbau für Messung der Bildlatenz. . . . .	55
5.3	Ergebnisse der Messung . . . . .	56
5.4	Ausschnitte aus einem Video . . . . .	57

# Abkürzungsverzeichnis

<b>FPS</b>	Frames Per Second
<b>FPV</b>	First Person View
<b>HMD</b>	Head-Mounted Display
<b>IK</b>	Inverse Kinematik
<b>ROS</b>	Robot Operating System
<b>VM</b>	Virtuelle Maschine
<b>VR</b>	Virtual Reality

# 1 Einleitung

## 1.1 Motivation und Zielsetzung

Bei dieser Seminararbeit handelt es sich um ein Folgeprojekt der Seminararbeit „Multimodale Telepräsenz mit dem humanoiden Roboter NAO und VR-Brille“ von Isabella Schmidt und Fabian Dogendorf. Diese wurde 2018/2019 an der Dualen Hochschule Baden-Württemberg, Karlsruhe unter Betreuung von Herrn Prof. Dr. Hans-Jörg Haubner durchgeführt. Das Ergebnis war ein virtuelles Labor, über welches dem NAO Befehle übermittelt werden konnten, wie das Sprechen von Standardsätzen, das Einnehmen zuvor definierter Posen sowie eine Funktion, welche es ermöglicht den NAO in verschiedene Richtungen laufen zu lassen.

Diese Seminararbeit greift die Idee der Telepräsenz durch eine Kombination von NAO Roboter und HTC VIVE Brille auf, jedoch soll an Stelle eines Labors ein Cockpit treten. Die durch die HTC VIVE Controller getrackten Positionen und Bewegungen von Kopf und Armen sollen, sofern es mechanisch möglich ist, vom Roboter imitiert werden. Infolgedessen wird der Roboter unmittelbar über die HTC VIVE Controller gesteuert. Dem Benutzer wird das Blickfeld des Roboters auf die Displays seines VIVE Headsets übertragen. Dieser bekommt so ein besseres Gefühl für die Bewegungen des Roboters. Anhand neuer Technologien, wie ROS-Sharp fällt eine Unity Anbindung von ROS deutlich leichter, wodurch eine Aufbereitung der Methodik des Vorgängerprojektes zur Anbindung des NAO hinfällig ist.

Sollte es im zeitlichen Rahmen möglich und technisch realisierbar sein, kann das Projekt um weitere Funktionen erweitert werden. Dazu zählt zum Beispiel das Anzeigen des Akkustands oder die Stärke des Wifi Signals im Cockpit. Eine weitere optionale Erweiterung ist die Übertragung von Audio vom Anwender zum Roboter und vom Roboter zu den Kopfhörern der HTC VIVE Brille. Eine weitere Möglichkeit ist die Bewegungsfreiheit des Roboters zu erweitern, indem weitere VIVE-Tracker an den Beinen des Benutzers befestigt werden.

## 1.2 Stand der Technik

Die vorhergehende Studienarbeit “Multimodale Telepräsenz mit dem humanoiden Roboter NAO und VR-Brille”, von Isabella Schmidt und Fabian Dogendorf, implementiert bereits eine Herangehensweise den NAO in vordefinierte Posen zu versetzen. Damit ist eine Steuerung des NAO bereits möglich, allerdings werden hier die Möglichkeiten der Virtual Reality (VR) noch nicht voll ausgeschöpft.

Dyno Robotics hat im Oktober 2019 ein Video veröffentlicht, in dem eine live Synchronisation der Arme mithilfe einer Valve Index Brille bereits funktioniert [Rob19]. Allerdings sind hier die Bewegungen sehr abgehackt und durch Einsatz der im NAO integrierten Kamera ist eine Interaktion mit der Umgebung aufgrund hoher Verzögerung und niedriger Bildwiederholrate nur schwer möglich.

## 1.3 Entwicklungsumgebung

Im Kern der Entwicklungstools steht ein Windows 10 V1903 Hostsystem, auf welchem ein Ubuntu 14.04 Gastsystem gehostet wird. Die Virtualisierung erfolgt durch Oracle Virtual Box 6.0.

Auf dem Windows 10 System ist Unity in der Version 2019.3.7 installiert, in Zusammenarbeit mit Visual Studio 2019 als Entwicklungsumgebung. Unity wurde für die Einbindung der HTC VIVE und der ROS-Bridge um die Plugins SteamVR und ROS-Sharp (ROS#) erweitert. Für die 3D Modellierung wird Blender 2.8 verwendet und für das Slicen von 3D Modellen Cura 4.4.

Auf dem Ubuntu Gastsystem läuft die ROS Distribution Indigo Igloo mit Schnittstellen zu ROS# und dem Netzwerk, in welchem sich der NAO befindet.

Für dieses Projekt wird ein NAO der V4 Generation in der H25 Bauweise verwendet. Um dem Benutzer ein stereoskopisches Bild zu vermitteln, werden zwei voneinander unabhängige Kamerasysteme parallel installiert. Bei den zusätzlich am NAO verbauten Kameras handelt es sich um Kameras der Baureihe “Foxeer Predator V4” mit einer 2.5mm Linse. Die verbundenen Sender sind vom Typ “Team Blacksheep UNIFY PRO 5G8 HV - RACE (SMA)”. Sie senden über ein analoges 5GHz PAL Signal an zwei Empfänger der Marke “FUAV Mini 5.8G FPV Empfänger”, welche über USB an den PC angeschlossen sind und dort als Webcam erkannt werden.

## **1.4 Vorgehensweise**

Für die Bearbeitung des Projektes wurden die Aufgaben zwischen beiden Projektmitarbeitern aufgeteilt. Dies hatte den Vorteil, dass die Arbeit unabhängig von einander durchgeführt werden konnte und die Komponenten mittels der zuvor gemeinsam definierten Schnittstelle in Unity zusammengefügt werden. Hierdurch konnte jeder Expertenwissen für seinen Aufgabenbereich aufbauen, ohne das langwierige Einarbeitungen mehrfach durchgeführt werden mussten.

Janis Schneider befasste sich mit folgenden Aufgaben:

- Anbindung der HTC Vice an Unity
- Implementieren eines Skripts für die Berechnung der Armgelenke
- Ausarbeiten eines Kamerasytems und Entwicklung einer Halterung für dieses
- Einbindung der analogen Kameras in Unity
- Aufbau von UI / UX

Wohingegen Dennis Jahnke seinen Fokus auf Folgendes legte:

- Aufsetzen von ROS auf einem Gastsystem
- Verbindung von ROS mit Unity durch ROS-Sharp
- Erstellen eines ROS-Projektes, welches die vollständige NAOqi Schnittstelle startet
- Aufbau einer Schnittstelle zur Bedienung des NAO aus Unity

## 2 Grundlagen

### 2.1 NAO Roboter

Der humanoide Roboter NAO wurde im Jahr 2005 zum ersten Mal der Öffentlichkeit präsentiert. Hersteller ist das französische Unternehmen Aldebaran Robotics, welches seit 2016 Teil der japanischen Softbank Group ist. [Robc] Schnelle Bekanntheit verdankt der Roboter dem Robocup, einem Fußballturnier für Roboter. Seit 2008 dient er dem Turnier, aufgrund seiner leichten Programmierbarkeit, als Standardplattform. [ROBa]

Im selben Jahr wie die erste Präsentation erschien das Modell V1 zum Verkauf für Universitäten, Labore und anderen Bildungseinrichtungen. Das Modell V2 erschien 2010 mit kleinen Überarbeitungen, besonders im Bereich der Motoren. Ein Jahr später wurde der NAO Next Gen veröffentlicht, mit Verbesserungen an Hard- und Software. Weitere große Verbesserungen erfolgten in den Jahren 2013 mit dem NAO Evolution (V4) und dem NAO V6 im Jahre 2018.[Robe]

Während das aktuelle V6 Modell von 4GB DDR3 RAM, einem Intel Atom Quadcore und einer 32 GB großen SSD profitiert, [Robf] sind in dem V4 Modell, welches für diese Studienarbeit verwendet wird, 1 GB RAM ein Intel Atom Z530 und einer 8GB Micro SDHC verbaut.[Robe]

Das Betriebssystem des NAO ist eine Linux Distribution namens NAOqi. Die aktuelle Version für das V6 Modell ist hierbei 2.8 (Stand Oktober 2019). Für das genutzte V4 Modell ist ein Update auf 2.1 möglich, jedoch besteht zu der Auslieferungsversion 1.14 eine wesentlich höhere Kompatibilität, weshalb das verwendete NAO Modell V4 auf der Version 1.14 genutzt wird.[Robg]

Die Programmierung des NAO kann über die hauseigene Choreographe Suite erfolgen. Hierbei werden vorgefertigte Blöcke graphisch angeordnet und abhängig ihrer Reihenfolge vom Roboter abgearbeitet. Auch parallele Ansteuerung, beispielsweise eines Armes und der in den Augen verbauten LEDs, ist ebenso wie das Erstellen neuer Blöcke in den Programmiersprachen C++ und Python möglich. Die so verfassten Programme können entweder live auf dem NAO ausgeführt oder dauerhaft auf dem NAO gespeichert werden.

Letzteres macht besonders dann Sinn, wenn das Programm an ein Event geknüpft ist, zum Beispiel wenn ein Gesicht erkannt oder ein Button gedrückt wird.[Robd]

Des Weiteren verfügt die Choreographe Suite über weitere überwachende und konfigurierende Funktionen. Dazu gehört die Anzeige der Live Bilder der Kamera des NAO-Roboters, der Akkustand, der Verbindungsmanager, die Diagnosetools und eine Bibliothek über die vom Roboter einnehmbaren Posen.[Robd]

In den vergangenen Jahren war der NAO-Roboter immer wieder das Aushängeschild von Kongressen und Veranstaltungen zu Industrie 4.0 und Künstlicher Intelligenz. Hierbei konnte der NAO beispielsweise auf Fragen von Zuschauern antworten oder komplexere Arbeitsschritte selbstständig ausführen. Sein kindliches und niedliches Aussehen lässt somit komplexe Themen der Robotik anschaulich verdeutlichen.

Wie ursprünglich geplant, dient der NAO auch heute noch besonders an Universitäten, Schulen, Museen und anderen Bildungseinrichtungen als leicht bedienbares Lern- und Forschungsobjekt. Der Anschaffungspreis eines NAO V6 liegt (Stand: Januar 2020) bei ca. 15.000 Euro. [Robh]

Zu dem Funktionsumfang des 58cm großen Roboters (Modell V6) gehören unter anderem sieben Berührungssensoren, Ultraschallsensoren, vier Mikrophone und zwei Lautsprecher. Seine Text-to-Speech Funktion sowie die Spracherkennung unterstützen 20 verschiedene Sprachen. Die beiden im Kopf verbauten Kameras sind in der Lage Objekte, Personen und Gesichter zu erkennen und unterstützen einen Wärmebildmodus. Die LEDs an Augen und Ohren (je acht) lassen sich individuell in unterschiedlichen Farben ansteuern. Dabei sind Bewegungen in 25 Freiheitsgraden möglich. [Robf]

Die Standardvariante jedes Modells ist die H25 Version. Diese umfasst den vollständigen Roboter. Die H21 Version reduziert den Roboter um wenige Funktionen, wie die Verwendbarkeit der Finger und zusätzlichen Sensoren an Händen und Füßen des Roboters. Bei der T14 Version wird der NAO auf seinen Oberkörper reduziert. Beine und Hüfte fehlen in dieser Version. Die T2 Version verfügt nur noch über Kopf und Brust, hier fehlen auch die Arme des Roboters. Die Zahl im Versionsnamen beschreibt die jeweiligen Freiheitsgrade der Version. [Robe]

## 2.2 Robot Operating System – ROS

Das Framework Robot Operating System, kurz ROS, ist eine Sammlung von Konventionen, Standards, Tools und Bibliotheken, mit dem Ziel die Programmierung von Robotern zu vereinfachen. So können Entwickler ihre Schnittstellen, Werkzeuge und Programme zu der ROS Bibliothek hinzufügen und damit anderen Entwicklern, welche an dem gleichen Roboter oder einem ähnlichen Projekt arbeiten, viel Arbeit abnehmen. Somit kann die ROS-Community durch die Erweiterung von ROS allen Anwendern den Zugang zu Robotern, beziehungsweise deren Programmierung erleichtern. [Robb]

Begonnen hat dieses Projekt im Jahre 2010. Hier wurde die ROS Version 1.0 veröffentlicht. Die eigentliche Arbeit begann jedoch schon im Jahre 2007 an der Stanford Universität. Ein benachbartes Robotik-Forschungslabor „Willow Garage“ wurde auf das Projekt aufmerksam und steuerte open source Software zu diesem Projekt bei. Kurze Zeit später entstand eine breite Community aus Robotik Laboren, welche sich an dem Projekt und der Idee beteiligten. [Robb]

Seit März 2010 veröffentlicht ROS seine versionierten Packages als Distributionen. Diese sind in sich relativ stabil. Das hat den Vorteil, dass Entwickler, wenn sie mit ROS arbeiten nicht fürchten müssen, dass nach einem Update ihre Software nicht mehr funktioniert. Sie haben eine stabile Plattform, auf welche sie ihre Software aufsetzen können und welche im Standardfall fünf Jahre lang unterstützt wird. Während in den Anfangsjahren jährlich zwei Distributionen veröffentlicht wurden, ist der aktuelle Veröffentlichungsintervall bei einem bis zwei Jahren. [Robb]

Die aktuellen Versionen (Stand Januar 2020) sind Kinetic Kame (veröffentlicht 2016) und Melodic Morenia (veröffentlicht 2018). Für Mai 2020 ist die neue Distribution Noetic Nujemys angekündigt. Alle ROS Distributionen sind für linuxoide Betriebssysteme entwickelt worden. Im Primären auf eine korrekte Funktionsweise auf dem jeweils aktuellen Ubuntu Systems zum Zeitpunkt der Veröffentlichung. Im Falle von Indigo Igloo (veröffentlicht 2014) handelt es sich um Ubuntu 14.04 LTS (Trusty). Andere Betriebssysteme wie Mac OS X, Android und Windows werden nur limitiert unterstützt. [Robb]

Die ROS Community hat mittlerweile eine Größe erreicht, dass seit 2012 jährlich eine mehrtägige ROS Developer Conferenz (ROSCon) organisiert wird. 2013 war die ROSCon in Stuttgart. In den Jahren 2015/2016 in Hamburg. Dabei werden Vorträge, Workshops, Tutorials und TecTalks angeboten. [Robb]

Der erste Anlaufpunkt für Neueinsteiger ist das ROS-Wiki. Hier befindet sich nicht nur eine Installationsanleitung, sondern auch eine Vielzahl von Tutorials zum Grundaufbau, den Standards und weiteren ROS Funktionen.

Projekte in ROS sind in Packages gegliedert. Bei diesen handelt es sich um Catkin Packages. Jedes Package muss eine Konfigurationsdatei enthalten und sich in einem eigenen Ordner befinden. Außerdem muss jedes Package sein eigenes CMakeList File enthalten, um die Kompilierung über mehrere Packages zu ermöglichen. Ebenso können Abhängigkeiten festgelegt werden. Die Gesamtheit aller Packages eines Projektes befinden sich üblicherweise in einem Catkin Workspace. In diesem befindet sich auch das Top Level CMakeFile. [Rec]

Für eine bessere Navigation im Workspace stellt Robot Operating System (ROS) eigene Befehle zur Navigation bereit. Zum Beispiel können mit „rospack“ Informationen zu einzelnen Paketen abgerufen werden. „rosls“ hingegen erlaubt, im Gegensatz zum Linux typischen „ls“, die Inhalte einzelner Pakete aufzulisten, ohne den absoluten Pfad angeben oder sich in das entsprechende Verzeichnis begeben zu müssen.[Rec]

Während der Ausführung eines ROS Projektes besteht dieses aus Nodes. Als Node wird jedes ausführbare File innerhalb eines Packages bezeichnet. Diese Nodes können untereinander kommunizieren, Services starten und beanspruchen sowie Publisher und/oder Subscriber von Topics sein. Eine Liste aller laufenden Nodes kann mit dem Befehl „rosnode list“ angezeigt werden. Für die Ausführung einer Node ist nicht das gesamte Package zu starten. ROS-Nodes können auch einzeln, unabhängig von ihrem Package, gestartet werden. Hierzu steht der Befehl „rosrun“ zur Verfügung.[Rec]

Das Herz eines jeden ROS Programms ist der sogenannte "ROS-Core". Dabei handelt es sich um eine Sammlung von Nodes, welche gemeinsam die Grundlage für das ROS System bilden. Ein laufender ROS-Core ist notwendig für jede ROS-Anwendung.[Rec]

Für die Kommunikation zwischen Nodes werden sogenannte Topics verwendet. Dabei handelt es sich um eine gemeinsame Kommunikationsschnittstelle. Hierbei wird keine synchrone Client Server Kommunikation verwendet (request/reply), sondern ein Publisher Subscriber Prinzip das auch asynchrone Kommunikation unterstützt. Dies ist besonders bei der Kommunikation mit eventgesteuertem Roboterverhalten essentiell. Bei ROS verläuft das Publisher Subscriber Prinzip wie folgt. Eine Node meldet sich als Subscriber zu einem Thema. Dabei wird unter einen fest definierten eindeutigen Topic Namen eine Schnittstelle angelegt. Ein Publisher kann nun Informationen versenden, indem er sie

an die entsprechend benannte Schnittstelle adressiert. Meldet sich eine weitere Node als Subscriber zu demselben Topic, so erhält auch sie alle Daten, welche an die Schnittstelle gepublisiert werden. Dieses one-to-many Prinzip steht, dank seiner Performance, im Kontrast zum many-to-one Prinzip, bei welchem ein Server viele Anfragen verarbeiten muss. [Rec]

Die Datenübertragung über ROS-Topics geschieht mittels ROS-Messages. Der Typ der ROS-Message bestimmt auch den Typ des ROS-Topics. Nur Messages des definierten Types können über ein Topic gepubliziert werden. Dies verhindert, dass eine Node, welche einen String erwartet, einen Integer erhält. Zu den Standard-Message-Types gehören, neben dem genannten String, auch integer und unsigned integer mit 8,16,32,64 bit, time, duration, float, boolean, und Arraytypen für Bytes und Bits sowie Arrays anderer Typen variabler und fester Länge. [Rec]

Um die Kommunikation zwischen Nodes zu entlasten, gibt es auch eine Client Server Struktur innerhalb von ROS. Diese ermöglicht es, dass Nodes gezielt Daten dann erfragen können, wenn sie diese benötigen und damit nicht vom nächsten publish abhängig sind. Die hierzu gehörigen Schnittstellen heißen unter ROS 'Services'. Sie werden ähnlich angelegt wie Topics, nur dass diesmal nach dem request-response Prinzip kommuniziert wird. Eine Node ruft einen Service unter seiner eindeutigen Adresse auf und übergibt gegebenenfalls noch notwendige Parameter. Anschließend behandelt der Service, beziehungsweise die Node, welche den Service anbietet, die Anfrage und antwortet der Node. Anders als bei einem Topic kann es immer nur eine Node geben, welche den Service anbietet. [Rec]

Damit Services nicht für einfache getter und setter Funktionen verwendet werden, existiert der ROS Parameter Server. Auf diesem können Parameter der gängigsten Datentypen gespeichert werden. Diese lassen sich dann gezielt auf Abrufen setzen und löschen. Der Vorteil dieses einheitlichen Parameterspeichers ist, dass keine Vielzahl an Services erstellt werden muss, nur um auf Daten zuzugreifen. [Rec]

Die Namen aller Nodes, Topics, Services und Parameter unterliegen der ROS Namenskonvention. Der Namenspace ist hierbei hierarchisch aufgebaut. Ein (frei erfundenes) Beispiel für ein Topic wäre: /Turtlebot1/Turtle/left\_wheel/position. [Rec]

## 2.3 ROS-Sharp (ROS#)

Bei ROS-Sharp handelt es sich um ein Set aus open source Software, Bibliotheken und Werkzeugen programmiert in C#. Das Ziel dieses Sets ist es .NET Applikationen an ROS anbinden zu können. Der Fokus ist jedoch auf die Anbindung von Unity an ROS gesetzt. Umgesetzt wird dieses Softwareprojekt von Martin Bischoff, welcher es seit September 2017 unter der Apache 2.0 Lizenz veröffentlicht und damit auch für kommerzielle Nutzung kostenfrei zur Verfügung stellt. [Bis]

Zu den wesentlichen Features gehört die Möglichkeit von Unity über einen ROS-Bridge-Client auf das ROS-Netzwerk zugreifen zu können, um Messages, Services und Topics erstellen und empfangen zu können. Außerdem kann ROS das Urdf Modellformat nach Unity importieren. [Bis]

Das Unified Robot Description Format (Urdf) ist ein im Rahmen der ROS Standards festgelegtes XML Format zur Darstellung von Robotern. Hierbei wird eine hierarchische Struktur mit Gelenken, Armen Sensoren und Aktoren aufgebaut. [IS]

Für das Einbinden in Unity kann die aktuelle Version direkt aus dem Unity Asset Store geladen werden. Für andere Endprogramme können auch die Binaries für den ROS-Bridge-Client und Urdf einzeln heruntergeladen werden.

Das Unity Asset erweitert die Menüleiste um einen weiteren Reiter, welcher die Basis ROS-Sharp Funktionen beinhaltet. Hier können Urdf Modelle schnell importiert und exportiert sowie Messages und Services erstellt werden. Des Weiteren beinhaltet das Asset einige neue Komponenten, welche sich mehrheitlich mit dem publishen und subscriben beschäftigen sowie der Codierung für und von ROS.

ROS-Sharp verfügt über sein eigenes Wiki, welches Grundfunktionen der Schnittstelle in Tutorials knapp erklärt. Bei weitergehenden Fragen steht der Entwickler sowie die noch überschaubare Community hilfsbereit zur Verfügung.

## 2.4 Unity

Unity ist eine frei zugängliche Spiel-Engine, welche 2005 von Unity Technologies veröffentlicht wurde. Sie sticht durch ihr breites Funktionsportfolio und ihrer niedrigen Einstiegshürde heraus. Unity wurde für dieses Projekt gewählt, da das bereitgestellte Framework ein schnelles Konzipieren der Anwendung ermöglicht und die Erweiterbarkeit über den Unity Asset Store das Zusammenführen der VR-Brille und des NAO Roboters in einer virtuellen Umgebung drastisch vereinfacht.

Grundlage für diese virtuelle Umgebung ist die Szene. In ihr können dreidimensionale Objekte, sogenannte GameObjects, platziert werden. Jedes dieser GameObjects kann Komponenten enthalten. Diese reichen von einem Mesh und einer rendering Komponente bis hin zu Skripten, dessen in C# geschriebene Logik zur Laufzeit ausgeführt wird.

Mithilfe des Asset Stores können nun die Erweiterungen hinzugefügt werden. Für den NAO wird das ROS-Sharp Plugin (siehe: 2.3) verwendet. Für die VR-Brille wird das von SteamVR bereitgestellte Plugin installiert, womit alle Komponenten für das Verwenden der Brille vorhanden sind. Eine Instanz der Brille mit ihren Controllern mit der virtuellen Welt interagieren zu lassen besteht nun nur noch aus einem drag&drop des passenden Prefabs, einer vorkonfigurierten Hierarchie von GameObjects, in die Szene.

## 2.5 Virtual Reality

“Virtuelle Realität (Virtual Reality, VR) ist eine computergenerierte Wirklichkeit mit Bild (3D) und in vielen Fällen auch Ton. Sie wird über Großbildleinwände, in speziellen Räumen (Cave Automatic Virtual Environment, kurz CAVE) oder über ein Head-Mounted-Display (Video- bzw. VR-Brille) übertragen. Bei Mixed Reality wird entweder Realität erweitert (Augmented Reality), wobei für die Darstellung und Wahrnehmung eine AR-Brille (oft Datenbrille genannt) benötigt wird, oder aber Virtualität, im Sinne der Kopplung mit der Realität.” [Ben17]. Im Rahmen dieser Arbeit wird die Anwendung dieser Technologie auf das Head-Mounted-Display beschränkt.

### 2.5.1 Problemstellungen der Virtual Reality

**Motion Sickness:** Das größte Problem von VR ist die auftretende Übelkeit, wenn das in der Brille simulierte Bild nicht dem entspricht, was das Gehirn als Resultat seiner Bewegungen erwartet. Im Falle von NAOVR ist also zu beachten, dass die Verbindung der Brille zum NAO in beide Richtungen mit minimaler Verzögerung abläuft. Das bedeutet, dass die Bewegungen des Benutzers möglichst schnell vom NAO umgesetzt werden müssen und die am NAO installierten Kameras das aufgenommene Bild möglichst schnell zurück zur Brille geben. Falls dieser Prozess nicht optimiert wird, wird der Benutzer feststellen, dass seine Kopfbewegungen "hinterher hängen", sprich er bekommt das Bild angezeigt, das er zu einem vorherigen Zeitpunkt hätte sehen sollen. Ein weiterer Einflusspunkt ist die Genauigkeit der Bewegungen. Neben einer schnellen Ausführung der Bewegungen muss auch eine akkurate Steuerung möglich sein. Sind die Bewegungen unregelmäßig oder enden in einer Position die sich stark von der des Benutzers unterscheidet, tritt ebenfalls Motion Sickness auf. Daher muss akkurate Tracking des VR-Headsets gewährleistet sein und ebenso müssen die Bewegungsbefehle an den NAO dahingehend optimiert werden.

**Räumliche Begrenzung:** Da im Gegensatz zur Augmented Reality bei der Virtual Reality der Benutzer von seiner tatsächlichen Umgebung abgeschottet wird, muss eine virtuelle Abgrenzung des Raumes in der sich der Benutzer bewegen kann erfolgen. Dies wird üblicherweise bereits vom Brillenhersteller über eine virtuelle Wand bewerkstelligt, die erscheint wenn sich der Benutzer einer Grenze nähert. Falls der NAO einen größeren Bewegungsraum betreten können muss als der physikalische Raum den der Nutzer geboten bekommt, muss diese Raumerweiterung über die Software gelöst werden.

## 2.6 HTC Vive

Das VR-Headset, welches aus der Kooperation von HTC und Valve 2016 als eines der ersten kommerziell erhältlichen Headsets für PC hervorgeht, bietet in diesem Projekt das Medium über das der Benutzer mit der Software und somit letztendlich mit dem NAO Roboter interagieren kann. In der Grundausführung besteht es aus einer Brille mit je einem Bildschirm pro Auge, einen Controller für jede Hand und zwei Lighthouses, welche die Bewegung der Komponenten verfolgen. Das Grundsystem kann anschließend durch weitere Lighthouses für eine bessere Raumabdeckung und weitere Tracker, um mehr Objekte tracken zu können, erweitert werden.

Die Interaktion mit der virtuellen Umgebung beschränkt sich im Wesentlichen auf die Bewegung von Kopf und Händen und den Eingabemöglichkeiten der Controller, bestehend aus einem Trackpad für den Daumen, welches auf einer Kreisfläche Position und Druck des Daumens ermitteln kann und einem Trigger für den Zeigefinger, welcher sowohl die Eindrücktiefe des Knopfes misst, als auch am Anschlag einen mechanischen Taster besitzt, damit der Trigger auch als binäre Eingabe verwendet werden kann. An den Seiten befinden sich noch zwei Gripbuttons, Druckflächen mit denen ein Zusammendrücken der übrigen drei Finger registriert werden kann.

## 2.7 Inverse Kinematik

Die inverse Kinematik entspringt aus dem Bereich der Robotik und beschäftigt sich mit dem Problem, zu einer gewünschten Position und Rotation des Endeffektors, in diesem Fall die Hände des Nao, die Gelenkwinkel zu bestimmen, dass die kinematische Kette sich letztendlich so ausrichtet, dass der Endeffektor genau diese gewünschte Pose einnehmen kann. Ist diese Pose nicht einnehmbar, soll eine realistische Alternative gefunden werden.

## 3 Anforderungsanalyse

Aus der Zielsetzung geht hervor, dass das Einnehmen von Posen der Arbeit nicht gerecht wird. Es muss also ein Weg gefunden werden, die Motoren des NAO unabhängig voneinander anzusteuern, über einen Weg der sowohl ein schnelles Reagieren als auch eine Anbindung an eine Software mit der sich die VR-Brille verbindet ermöglicht. Hierbei soll ROS als Middleware eingesetzt werden, um zukünftigen Projekten bessere Anbindung zu bieten. Dabei ist besonders eine schnelle Verarbeitung der Bewegungs- und Positionsdaten durch ROS erforderlich, um Verzögerungen in der Darstellung der Bewegung zu vermeiden.

Damit der Benutzer gut mit seiner Umgebung interagieren kann, muss der NAO ein Livebild übertragen. Auch hier ist kurze Latenz und möglichst hohes Sichtfeld von Vorteil. Des Weiteren müssen die getrackten Objekte des VR-Headsets, zunächst das Head-Mounted Display (HMD) und die beiden Handcontroller, in einem virtuellen Raum zur Weiterverarbeitung übertragen werden, um später aus diesen Informationen die eingenommene Pose des Benutzers herauszurechnen. Da nur 3 Objekte mit Position und Rotation als Referenz genutzt werden können, müssen für eine Steuerung des NAO die restlichen Gelenke, mithilfe von inverser Kinematik, berechnet werden. Zusätzlich zum Tracking müssen die Controller auch als Eingabegeräte dienen, um verschiedene Bestandteile der Anwendung aus der Brille heraus steuern zu können.

# 4 Umsetzung

## 4.1 Inbetriebnahme von ROS

Für die Inbetriebnahme von ROS muss zunächst die Frage nach der verwendeten Distribution geklärt sein. Neue Distributionen wie Melodic bieten bessere Stabilität und Performance sowie mehr Funktionen. Viele der ROS Pakete wurden jedoch für ältere Distributionen entwickelt und werden meist erst sehr spät auch für die aktuellen Distributionen veröffentlicht. Im Falle dieses Projektes wird auf die ROS Indigo Distribution zurückgegriffen aus dem Jahre 2014. Für diese Distribution wurden die meisten der NAO Schnittstellenpakete entwickelt. Daher ist davon auszugehen, dass diese Pakete auf dieser Distribution am stabilsten laufen. Zwar wurden ein paar der benötigten Pakete auch für die Kinetik Distribution von ROS angepasst, jedoch ist hier die Dokumentation spärlicher und der Funktionsumfang geringer.

Da ROS Indigo Igloo für Ubuntu 14.4 entwickelt wurde, macht es notwendig, ein Ubuntu System zum Hosten von ROS aufzusetzen. Hierfür wird eine Virtuelle Maschine (VM) mit der 14.4 Version von Ubuntu auf dem Windows 10 Hostsystem aufgesetzt. Im ersten Schritt muss eine VM erstellt und für den Einsatz mit ROS konfiguriert werden. Entscheidend ist besonders das Einrichten einer Netzwerkbrücke, um direkte Zugriffe von ROS auf das Netzwerk zu ermöglichen. Anschließend gilt es ROS Indigo zu installieren. Hierbei wird die Anleitung aus der ROS Dokumentation als Grundlage verwendet.

Zunächst muss hierbei gewährleistet werden, dass das Ubuntu System Installationen vom ROS Server erlaubt. Dazu muss die Sources Liste um den entsprechenden Server erweitert werden. Anschließend kann die Verbindung zum Keyserver aufgebaut werden. Befindet sich der PC hinter einem Proxy, macht es Sinn hierfür nicht den apt-key Befehl zu verwenden, sondern curl zu nutzen. Sind diese Voreinstellungen gesetzt, kann die eigentliche Installation erfolgen. Zuvor empfiehlt es sich jedoch, die Aktualität des Systems sicherzustellen.

Bevor ROS jedoch verwendet werden kann, müssen noch ein paar Einstellungen am System vorgenommen werden, welche die Verwendung von ROS erleichtern. Dazu gehört die Installation von rosdep, einem Abhängigkeiten-Manager und rosinstall einem command-line Tool, welches die Installation mehrerer ROS-Pakete auf einmal erleichtert. Besonders

nützlich ist es ebenso die Environment-Variablen von ROS automatisch beim Öffnen einer neuen Konsole zu setzen. Andernfalls muss man bei jedem neuen Terminal, das auf ROS zugreifen soll, die Variablen händisch setzen. Nach dem Abschluss dieser Schritte sollte es möglich sein, den ROS-Core zu starten, ROS-Pakete zu installieren und ROS zu verwenden.

## 4.2 Anbindung des NAO an ROS

Nach der erfolgreichen Installation von ROS kann die Anbindung des NAO beginnen. Dafür ist es jedoch notwendig einige ROS Pakete zu installieren. Auch hierfür stellt ROS eine Anleitung bereit.

Im ersten Schritt wird jedoch das SDK für NAOqi benötigt. Dieses steht nur auf der Community Website von Aldebaran zum offiziellen Download zur Verfügung. Es reicht hierbei jedoch, sich als Student zu registrieren, sodass kein Nachweis erbracht werden muss, dass man tatsächlich einen NAO besitzt. Für diese Implementierung wurde die im Tutorial verwendete und empfohlene Version des SDK verwendet (2.7.2.17). Das SDK steht sowohl für Python als auch für c++ zur Verfügung. Auch hier haben wir uns, aus Gründen der vereinfachten Realisierbarkeit, für die Python Version entschieden, welche auch in der Anleitung verwendet wurde.

Für die Installation genügt es, das heruntergeladene SDK zu entpacken. Durch ein kurzes Ausführen von NAOqi kann die Funktionsfähigkeit validiert werden. Ein Hinzufügen des Python-Ordners zu pythonpath macht spätere Zugriffe auf die Python Bindings einfacher und wird daher in jedem Fall empfohlen.

Nachdem das SDK betriebsbereit ist, können die notwendigen Pakete für ROS installiert werden. Bevor die NAO spezifischen Pakete installiert werden, ist es notwendig, ein paar Allgemeine zur Kommunikation und Interpretation von humanoiden Robotern zu installieren.

Zu den ROS Paketen für die NAO Schnittstelle gehört das Meta-Packet Nao\_bridge. Es sorgt für die eigentliche Anbindung des NAO. Dieses beinhaltet drei elementare Pakete. Das ROS Packet Naoqi\_driver ist die zentrale Schnittstelle zwischen ROS und dem NAO. Das Naoqi\_driver Paket publiziert alle Daten der Sensoren und Aktoren des NAO unter spezifischen Topics.

Das Paket naoqi\_bridge\_msgs beherbergt alle Methoden und Datentypen, um Nachrichten an den NAO senden zu können, beziehungsweise eintreffende zu dekodieren. naoqi\_tools stellt Kompatibilitätsfunktionalitäten und andere Werkzeuge zur Verfügung, zum Beispiel zur Transformierung nach URDF. Das letzte Paket der naoqi\_bridge ist naoqi\_poses. Dieses beinhaltet Nodes zur Steuerung des Bewegungsverhaltens des NAO.

Weitere Pakete rund um den NAO sind nao\_bringup, nao\_description und nao\_apps. Das nao\_bringup Paket baut die Verbindung zum NAO auf und startet das Interface. Die nao\_descriptions beinhalten das Modell des NAO mit vordefinierten Freiheitsgraden und Bewegungssachsen. Mit anderen Worten, das zum NAO gehörende URDF Modell. Nao\_apps bietet dem Benutzer eine top level Schnittstelle zum Auslesen und Steuern des NAO. Diese werden in acht Funktionsbereiche zusammengefasst: nao\_alife, nao\_behavior, nao\_diagnostic\_updates, nao\_footsteps, nao\_leds, nao\_speech, nao\_tactile, und nao\_walker.

Wann immer nun eine Verbindung zum NAO aufgebaut werden soll, muss die nao\_bringup Node nao\_full.launch oder nao\_full\_py.launch ausgeführt werden. Diese benötigt dabei die IP des laufenden ROS-Core und die IP des NAO. Dafür müssen der NAO und die Ubuntu VM natürlich an das selbe Netzwerk angeschlossen sein. Sollte der pythonpath, wie in der Inbetriebnahme ROS beschrieben worden, gesetzt worden sein, ist es auch möglich die Node über Python zu starten.

Im online beziehbaren Zustand startet die nao\_full.launch Programme nur einen Teil der NAOqi Schnittstellen. So werden zum Beispiel alle Anwendungen von nao\_apps nicht automatisch gestartet. Um dies zu beheben, genügt es die nao\_full.launch Datei um die Aufrufe der entsprechenden .launch-Files zu ergänzen (siehe Abbildung 4.1).

```

<launch>

  <arg name="nao_ip"           default="$(optenv NAO_IP 127.0.0.1)" />
  <arg name="nao_port"         default="$(optenv NAO_PORT 9559)" />

  <arg name="roscore_ip"       default="127.0.0.1" />
  <arg name="network_interface" default="eth0" />

  <arg name="namespace"        default="nao_robot" />

  <!-- naoqi driver -->
  <include file="$(find naoqi_driver)/launch/naoqi_driver.launch" ns="$(arg namespace)" >
    <arg name="nao_ip"           value="$(arg nao_ip)" />
    <arg name="nao_port"         value="$(arg nao_port)" />
    <arg name="roscore_ip"       value="$(arg roscore_ip)" />
    <arg name="network_interface" value="$(arg network_interface)" />
  </include>

  <!-- launch pose manager -->
  <include file="$(find naoqi_pose)/launch/pose_manager.launch" ns="$(arg namespace)/pose">
    <arg name="nao_ip"           value="$(arg nao_ip)" />
    <arg name="nao_port"         value="$(arg nao_port)" />
  </include>

  <!-- launch led -->
  <include file="$(find nao_apps)/launch/leds.launch">
    <arg name="nao_ip"           value="$(arg nao_ip)" />
    <arg name="nao_port"         value="$(arg nao_port)" />
  </include>

  <!-- launch behavior -->
  <include file="$(find nao_apps)/launch/behaviors.launch">
    <arg name="nao_ip"           value="$(arg nao_ip)" />
    <arg name="nao_port"         value="$(arg nao_port)" />
  </include>

  <!-- launch tactile -->
  <include file="$(find nao_apps)/launch/tactile.launch">
    <arg name="nao_ip"           value="$(arg nao_ip)" />
    <arg name="nao_port"         value="$(arg nao_port)" />
  </include>

  <!-- launch walker -->
  <include file="$(find nao_apps)/launch/walker.launch">
    <arg name="nao_ip"           value="$(arg nao_ip)" />
    <arg name="nao_port"         value="$(arg nao_port)" />
  </include>

```

Abbildung 4.1: nao\_full.launch (Auszug), blau markiert sind ergänzte Anwendungen

Um das erfolgreiche Verbinden des NAO mit ROS zu überprüfen, kann die ROS eigene Simulationsplattform rviz genutzt werden. Hierzu kann das bereits konfigurierte rviz File aus der Nao\_description genutzt werden. Die Anzeige ist zwar optisch nicht sehr ansprechend, sollte aber genügen, um sicherzustellen, dass der NAO mit ROS verbunden ist. Möchte man das Modell etwas aufhübschen, besteht die Möglichkeit die NAO Meshes herunterzuladen, so dass das Modell dem NAO auch ähnlich sieht. Hierfür muss jedoch zusätzlichen Lizenzen zugestimmt werden.

Des Weiteren müssen bei einer erfolgreichen Verbindung, die zum NAO gehörenden Nodes, Topics und Services angezeigt werden. Testweise kann hier auch eine Nachricht über die Commandline an die Speech Topic verschickt werden, um zu testen, ob der NAO den übermittelten String vorliest.

## 4.3 Inbetriebnahme von ROS-Sharp

Für die Nutzung von ROS-Sharp ist es notwendig, in das bestehende ROS System eine ROSBridge zu installieren. ROSBridge ist ein Package, welches es ermöglicht auf Nodes, Services und Topics eines ROS-Netzwerkes über einen Webserver zuzugreifen. Die Installation kann hierbei einfach der Dokumentation entnommen werden und unterscheidet sich nicht von den anderen Netzwerken. Wird ROS auf einer VM ausgeführt, muss diese über eine Netzwerkbrücke verfügen, damit sich das Windows-Hostsystem mit Unity und der ROSBridge Webserver im selben Netzwerk befinden. Nach der Installation kann der ROSBridge-Server über einen einfachen ROS-Launch Befehl gestartet werden. Mehr Änderungen sind auf Seiten von ROS nicht notwendig.

Für Unity muss hingegen ROS-Sharp aus dem Asset Store heruntergeladen und importiert werden. Anschließend verfügt das Unity Projekt über eine Komponente, beziehungsweise ein Skript, namens ROSConnector. Dieses muss an ein beliebiges GameObject angefügt werden. Es empfiehlt sich jedoch, ein neues gleichnamiges Objekt anzulegen. Hierbei kann alles auf den Standardeinstellungen gelassen werden (Timeout: 10, Serializer: JSON). Das Protokoll sollte jedoch auf „Web Socket NET“ geändert und die IP-Adresse auf die des Ubuntu Gastsystems geändert werden. Diese ist identisch mit der IP des ROS-Core und lässt sich gegebenenfalls erneut mittels „ip addr“ Befehl herausfinden. Damit sollte beim Ausführen der Szene automatisch eine Verbindung zum Webserver und damit zum ROS Netzwerk aufgebaut werden. Ein erfolgreiches Verbinden/Trennen wird auch nochmal im InfoLog der Konsole angezeigt.

Bei der Arbeit mit realen oder simulierten Robotern in ROS kann es notwendig sein, das URDF Modell nach Unity zu importieren. Hierfür muss jedoch im ROS Netzwerk eine Node bestehen, welche das URDF auch publisht. In der Dokumentation von ROS-Sharp gibt es hierzu ein kleines Tutorial. Da das Importieren normalerweise nur einmal notwendig ist, kann der Quellcode beziehungsweise die Pfadangaben aus dem Beispiel so abändert werden, damit das eigene URDF Modell gepublisht wird. In Unity genügt es nun in der Menüleiste den ROSBridge Client auszuwählen und „Transfer URDF from ROS“ auszuführen. Es wird automatisch ein GameObject des Roboters erstellt und hierarchisch gemäß URDF aufgebaut, dabei werden nahezu alle Informationen des URDF in Komponenten abgelegt, welche zum Beispiel die Gelenke beschreiben.

## 4.4 NAO Schnittstelle für Unity

Die im Rahmen dieses Projektes implementierte Schnittstelle zum NAO für Unity besteht im Wesentlichen aus folgenden Funktionen:

- Auslesen und Setzen der Gelenkwinkel
- Auslesen der integrierten Kameras
- Verwendung der Speech Funktion
- Auslesen von Diagnosedaten
- Ein Stiffness-Controller für die Motoren
- Zugriff auf die Walk Funktionen
- Zugriff auf die LED-Steuerung
- Zugriff auf den Behavior Controller
- Zugriff auf den Pose Controller

Dabei setzen sich die ersten beiden Funktionen fast ausschließlich aus ROS-Sharp Standard Skripten zusammen.

### 4.4.1 Auslesen und Setzen der Gelenkwinkel

Das Auslesen und Setzen der Gelenkwinkel sind essenzielle Features für dieses Projekt. Als ausgesprochen nützlich erweisen sich hierbei die Skripte der ROS-Sharp Bibliothek. Sowohl für das Anzeigen der aktuellen Gelenkwinkel, als auch zum Setzen der Zukünftigen wird je ein importiertes URDF Modell des NAO verwendet. Dies ist besonders interessant, um einen Abgleich zwischen IST und SOLL zu erhalten. Das ROS-Sharp Kernskript für den Umgang mit Gelenkwinkeln ist der Joint State Patcher. Ist dieser mit dem entsprechenden URDF Modell verbunden, kann auf Knopfdruck ein Publisher und/oder Subscriber generiert werden. Dabei wird automatisch für jeden Joint ein Writer und/oder Reader Skript sowie eine Liste aller Joints angelegt. Theoretisch genügt es nun, das entsprechende Topic im Joint State Publisher/ Joint State Subscriber Skript zu definieren und die Informationen automatisch an ROS und damit an die NAOqi Schnittstelle zu übertragen. In der Praxis sind jedoch beim Publishing Skript Fehler aufgetaucht. Zum

einen hat das Publisher Skript nicht die echten Namen der Joints zur Übertragung verwendet, sondern eigene generiert. Zum Beispiel ist das Nacken-Rotations-Gelenk das Torso-Neck-Gelenk, weil es die Torso Komponente mit der Nacken Komponente verbindet. Dementsprechend ist es notwendig die Namen im Joint State Publisher Skript händisch auf die in der Dokumentation des NAO genannten zu ändern. Das Subscriber Skript hat dieses Problem nicht, da es für sich die Namen verwendet, welche es von ROS empfängt. Da die Reihenfolge der Gelenke im Array, welches von ROS kommt und dem welches Unity verwendet identisch ist, beide basieren auf demselben URDF Modell, kommt das Namens-Problem nicht zum Tragen da die Joints in Unity entsprechend dem Index und nicht dem Namen zugeordnet werden.

Ein weiteres Problem beim publishen der Gelenkwinkel ist die Abweichung des MessageTypes zwischen dem ROS-Sharp Skript und der NAOqi Schnittstelle. So versendet der Publisher mehr/andere Informationen über das Gelenk als die NAO Schnittstelle verarbeiten kann. Dieses Problem kann behoben werden, indem ein eigener MessageType generiert wird, welcher dem Standard der NAO Schnittstelle entspricht und diesen an Stelle des Standard Joint-MessageType im Publisher Skript verwendet.

Ein MessageType ist nichts Weiteres als ein Skript, bestehend aus einer Klasse mit folgenden Attributen: Einem konstanten String mit dem Namen des MessageTypes in ROS zum Beispiel „naoqi\_bridge\_msgs/JointAnglesWithSpeed“. Hierbei ist der erste Teil des Message Type das Packet, indem der Typ enthalten ist. Der Zweite ist der tatsächliche Name des MessageTypes. Diese Information ist für ROS Pakete wichtig, damit diese die Nachricht korrekt dekodieren können. Nach dem Namen folgen die Parameter, aus welchen die Nachricht tatsächlich besteht. Hierzu gehört meist ein Header Objekt gefolgt von Strings Integern oder Floats, aber auch exotischeren Datentypen wie Vektor3. Diese Informationen müssen immer der entsprechenden Dokumentation entnommen werden.

Neben den Attributen benötigt ein Message Typ auch einen Konstruktor. ROS-Sharp verfügt über die Möglichkeit MessageTypes automatisch generieren zu lassen. Dabei wird auch eine Vielzahl von Datentypen unterstützt. Dies erspart einige Programmierarbeit und fügt den neuen MessageType automatisch dem entsprechenden Namespace hinzu, was die Verwendung erleichtert. Manchmal sind MessageTypes ineinander verschachtelt. Ein Beispiel hierfür ist der MessageType DiagnosticArray, welcher für das Auslesen von Diagnosedaten erforderlich ist. Dieser besteht aus einem Array des Diagnostic Status MessageType, welcher wiederum den MessageType keyvalue benötigt. Hierbei sollte man sich von innen nach außen durcharbeiten.

Im Falle des Publishers der Gelenkwinkel reicht es einen neuen MessageType zu generieren und diesen im Joint Publisher an Stelle des Standardtyps zu verwenden, sodass die Messages von der ROS-Node auch korrekt dekodiert werden können. Der Subsciber hat dieses Problem nicht, da er einfach alle Parameter, welche er nicht verarbeiten kann, verwirft.

Für dieses Projekt werden lediglich die Position der Arme und des Kopfes an den NAO übertragen, da sich andere Werte, wie die der Beine mit den HTC Vive Controllern nicht auslesen lassen. Vielmehr würde ein Übertragen von falschen Werten oder einer initialen 0 das Risiko steigern, dass der NAO beim Verlassen der Laufbewegung umfällt. Auch wäre ein Hinknien des Roboters nicht möglich, da die Gelenkwinkel der Beine andauernd mit dem Initialwert überschrieben werden. Auch die Fingerpositionen werden in diesem Projekt nicht automatisch synchronisiert. Der Grund hierfür ist, dass es sich hierbei um das einzige Translationsgelenk des NAO handelt und deshalb nicht ohne weiteres an die Schnittstelle übertragen werden kann. Deshalb erfolgt auch das Öffnen und Schließen der Hand in diesem Projekt mittels Behavior. Dies sorgt dafür, dass der Joint State Publisher nur 14 der 42 Gelenkwinkel an den NAO überträgt.

#### 4.4.2 Auslesen der integrierten Kameras

Vergleichsweise einfach ist die Anbindung des Video-Streams des NAO. Dieser wird auf Grund seiner hohen Latenz von bis zu einer Sekunde in diesem Projekt nicht verwendet. Stattdessen werden externe analoge Kameras verwendet 4.6. Dennoch besteht die Möglichkeit den Videostream in Unity zu integrieren. Auch hierfür kann ein fertiges ROS-Sharp Skript verwendet werden. Der Image Subsciber benötigt ebenfalls lediglich das zugehörige Topic, anschließend kann der Stream auf einen beliebigen Mesh Renderer ausgegeben werden.

#### 4.4.3 Verwendung der Speech Funktion

Eine nützliche Funktion zum Debuggen und später auch zur Interaktion mit der Umwelt ist die Funktion des NAO zu sprechen. Hierbei wurde für Unity nur ein reduzierter Funktionsumfang implementiert. Eine Änderung der Stimme beziehungsweise Tonlage oder Lautstärke ist aus Unity heraus derzeit nicht möglich. So existiert eine Methode say(String), welche den übergebenen Parameter an das /speech Topic des Nao publisht.

Die Text-to-Speech Funktion des NAO lässt diesen daraufhin den übergebenen String aussprechen.

Die Speech Funktionalität ist somit die erste eigenständige Erweiterung in der Liste, welche nicht auf einer Standard-Funktionalität von ROS-Sharp aufsetzt. Weshalb sie, wie alle folgenden Funktionalitäten, ein eigenes Controller Skript besitzt und auf das NAO Schnittstellen GameObject ausgelagert wurde. Für die Implementierung der say()-Methode muss die Message an ein existierendes Topic gepublisiert werden. Unter ROS-Sharp wird hierzu der ROS-Socket benötigt, welcher mittels GetComponent() vom ROSConnector übernommen werden kann. Mittels des socket.Advertise<MessageType>(„/TopicName“) wird eine neue publication\_id generiert und somit eine Schnittstelle zum publishen auf dieses Topic generiert. Das eigentliche publishen erfolgt mittels socket.Publish(publication\_id, message). Dabei muss die Message dem zuvor angegebenen und erstellten MessageType des Topics entsprechen.(siehe Abbildung 4.2)

```

private RosSocket socket;
private string publication_id;
public std_msgs.String message;
0 Verweise
void Start()
{
    GameObject Connector = GameObject.FindWithTag("Connector");
    socket = Connector.GetComponent<RosConnector>()?.RosSocket;
    publication_id = socket.Advertise<std_msgs.String>("/speech");
    message = new std_msgs.String();
    say("Hallo, ich bin Mino. Du bist erfolgreich verbunden.");
}
0 Verweise
void Update()
{
    if (Input.GetKeyDown(KeyCode.Alpha0))
    {
        say("Los geht's.");
    }
}
5 Verweise
public void say(string text)
{
    message.data = text;
    socket.Publish(publication_id, message);
}

```

Abbildung 4.2: SpeechController, Beispiel einer Publisher Implementierung

Auch wenn der Einsatz der Speech Funktionalität vielfältig ist, wird sie derzeit lediglich zur Debug-Info verwendet. So wird dem Nutzer mitgeteilt, wenn sich Unity erfolgreich mit dem NAO verbunden hat und sich der Stiffnesszustand ändert. Vorbereitete Sätze oder eine Speech-to-Text Schnittstelle am HTC Vive Headset sind nicht implementiert.

#### 4.4.4 Auslesen von Diagnosedaten

Eine weitere Funktionalität der NAO Schnittstelle ist das Auslesen von Diagnosedaten, welche der NAO im Milisekundentakt publiziert. Hierzu wird eine Subscription für das Topic /diagnostics erstellt. Dies geschieht unter ROS-Sharp ähnlich wie das Publishen. Auch für Subscriptions wird der Socket benötigt. Die Besonderheit von Subscriptions im Vergleich zum Publisher ist, dass beim Erstellen der Subscription die Methode, in welcher die Nachrichten verarbeitet werden, mit angegeben wird. So lautet der Befehl zum Erstellen eines Subscribers: socket.Subscribe<MessageType>(„/TopicName“, SubscriptionHandler). Dieser SubscriptionHandler wird bei jedem neuen Publish auf das Topic aufgerufen und bekommt die Message mitüberggeben. Des Weiteren wird auch eine subscription\_id für jeden Subscriber generiert. (siehe Abbildung 4.3)

```
void Start()
{
    GameObject Connector = GameObject.FindGameObjectWithTag("Connector");
    socket = Connector.GetComponent<RosConnector>()?.RosSocket;
    string subscription_id = socket.Subscribe<msgs.DiagnosticArray>("/diagnostics", SubscriptionHandler);
}

1-Verweis
private static void SubscriptionHandler(msgs.DiagnosticArray message)
{
    Debug.Log(message.status);
}
```

Abbildung 4.3: DiagnosticConnector, Beispiel einer Subscriber Implementierung

Da das diagnostic-Topic in kurzer Zeit eine große Menge an Informationen publiziert. -Jede Message enthält Informationen zum Status jedes einzelnen Motors.- Wird es notwendig sein, die Messages nach Schlüsselworten zu durchsuchen, sodass die Informationen bevor sie an den Benutzer weitergegeben werden zuvor gefiltert wurden.

#### 4.4.5 Ein Stiffness-Controller für die Motoren

Der Stiffness Controller erlaubt es die Motoren des NAO aus Unity heraus zu aktivieren und zu deaktivieren. Aktivierte Motoren sind die Grundlage für jede Bewegung des NAO sowie dafür, dass er nicht in sich zusammensackt. Die Stiffness sollte deshalb zu Beginn einmalig aktiviert werden und beim Beenden deaktiviert werden. Dies muss der Nutzer manuell über das Drücken der Taste F1 durchführen, da beim Aktivieren der Roboter in die Initialpose fahren wird und beim Deaktivieren die Chance sehr groß ist, dass der NAO umfällt. Um hierbei Schaden am NAO zu verhindern, ist es nicht möglich die Stiffness mit den Vive Controllern zu verändern.

Die Funktionen des Stiffness Controllers umfassen enableStiffness(), disableStiffness() und wakeup(). Letztere wird in der Endversion des Projektes jedoch nicht mehr genutzt. Der Code bleibt dennoch enthalten, um möglichen Nachfolgeprojekten zusätzliche Möglichkeiten mitzugeben. Des Weiteren verfügt der Stiffness-Controller über eine Variable, welche den aktuellen Stiffness-Zustand des NAO anzeigt. Jede Stiffness-Änderung wird vom NAO über die say()-Methode angekündigt.

Für die Steuerung der Stiffness ist es notwendig Services des NAO aufzurufen. Hierfür wird wie beim Publishen und Subscriben der ROS-Socket benötigt. Dieser verfügt über die Methode CallService, welche den MessageType des Aufrufs und des der möglichen Antwort, sowie den Topic-Namen als Parameter braucht. Des Weiteren muss die Methode, welche sich mit der Antwort befasst, angegeben und eine RequestMessage übergeben werden. Im Falle der Body Stiffness (siehe Abbildung 4.4) ist sowohl die Request als auch die Response-Message leer. Aus diesem Grund ist auch der ServiceCallHandler ohne reale Funktion und dient lediglich beim Kompilieren als Referenz.

```
void Start()
{
    GameObject Connector = GameObject.FindWithTag("Connector");
    socket = Connector.GetComponent<RosConnector>()?.RosSocket;
    string subscription_id = socket.Subscribe<msgs.DiagnosticArray>("/diagnostics", SubscriptionHandler);
}

1-Verweis
private static void SubscriptionHandler(msgs.DiagnosticArray message)
{
    Debug.Log(message.status);
}
```

Abbildung 4.4: StiffnessController, Beispiel einer ServiceCall Implementierung

#### 4.4.6 Zugriff auf die Walk Funktionen

Damit der Arbeitsbereich des NAO sich nicht auf einen stark eingeschränkten Bereich beschränkt, bietet die Walker Funktion die Möglichkeit den NAO im Raum zu bewegen. Dabei enthält der Controller folgende vier verschiedene Methoden. walkAhead(), turnLeft(), turnRight() und stopWalking(). Auf eine Implementierung für das nach hinten laufen wurde absichtlich verzichtet, um Kollisionen mit Objekten, welche vom Benutzer nicht gesehen werden, zu vermeiden. Blind nach hinten zu laufen ist sowohl für die Umwelt als auch für den NAO gefährlich. Außerdem muss vor dem Laufen die Synchronisierung mit den Vive Controllern unterbrochen werden, da zu befürchten ist, dass ruckartige Arm- und Kopfbewegungen den NAO beim Laufen aus dem Gleichgewicht bringen könnten. Hierfür empfiehlt es sich auch den NAO vor und nach jedem Laufen in die Initialpose zu bringen. Realisiert wird das Laufen über zwei Vektoren, welche an das /cmd\_vel Topic gepubliziert werden.

#### 4.4.7 Zugriff auf die LED-Steuerung

Der LED Controller umfasst nur eine Methode, welche es erlaubt die LEDs blinken zu lassen. Diese blink()-Methode benötigt eine Parameterliste mit Farben zwischen denen gewechselt wird, eine Startfarbe sowie Informationen über Dauer und Intervall des Blinkens. Die Ansteuerung der LEDs des NAO hat einen geringen Mehrwert für den Einsatz des NAOs und wird in unserem Programm auch nicht eingesetzt. Dennoch verdeutlicht sie den Einsatz von Action Clients, welche auch für die Behavior und Pose Steuerung verwendet werden.

Action Clients/ Action Server sind ein ROS Konstrukt, welches das Abarbeiten mehrerer auf einander folgender Aufgaben unterstützt. Dabei übermittelt ein Client dem Server eine Aufgabe (Goal), welche dieser abarbeiten soll. Treffen mehrere Goals von verschiedenen Clients ein, werden diese der Reihe nach abgearbeitet. Alle Clients erhalten regelmäßig Auskunft über den Status aller laufender Goals und können diese bei Bedarf abbrechen. Goals können während ihrer Verarbeitung Feedback an den Client schicken und am Ende ein Result. Somit entsteht ein Austausch zwischen Client und Server wie auf Abbildung 4.5 zu sehen.

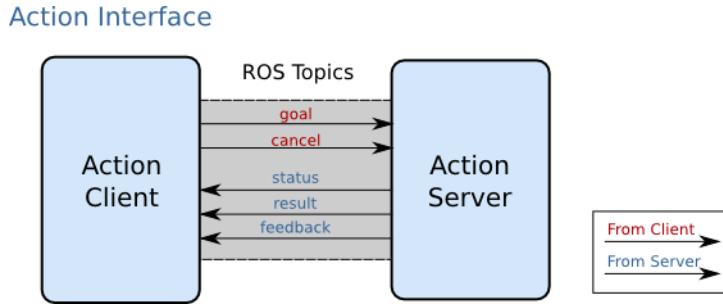


Abbildung 4.5: Action Interface in ROS Quelle [a]

Die Action Server der NAOqi Schnittstelle unterstützen jedoch weder Feedback noch Result, weshalb diese Funktionen der Actionserver für dieses Projekt nicht relevant sind. Dennoch müssen die MessageTypes der Goals, Feedbacks und Results in ROS-Sharp angegeben werden. Hierfür ist die automatische Generierung von MessageTypes besonders nützlich. Diese kann man unter dem ROSBridgeClient Reiter in der Menüleiste starten. Hier kann in einer GUI über wenige Klicks alle MessageTypes für ActionMessages generiert werden. Nachrichten ohne Inhalt, wie sie die Action Server der NAOqi Schnittstelle haben, werden mit einer Länge von 0 definiert.

Im Folgenden muss für ROS-Sharp ein für diesen Server passender Action Client erstellt werden. Dabei ist es am einfachsten von der bestehenden abstrakten Klasse ActionClient zu vererben. Die Subklasse mit dem individuellen ActionClient muss nur um die MessageTypes erweitert werden und die GetGoal()Methode überschrieben werden. In der Praxis war es in diesem Projekt ebenfalls notwendig die Start Methode zu überschreiben und die alte Start Methode in der neuen auszuführen um Probleme mit der kontinuierlichen Nutzung eines gemeinsamen Socket in Sub- und Superklasse zu vermeiden (siehe Abbildung 4.6).

```
public class LEDActionClient : ActionClient<msgs.BlinkActionGoal,
    msgs.BlinkActionFeedback, msgs.BlinkActionResult>
{
    public string behaviorName;
    public msgs.ColorRGBA bg_color;
    public msgs.ColorRGBA[] colors;
    public float blink_duration;
    public float blink_rate_mean;
    public float blink_rate_sd;

    6 Verweise
    protected override void Start()
    {
        ActionName = "blink";
        TimeStep = 0.1f;
        base.Start();
    }

    5 Verweise
    public override msgs.BlinkActionGoal GetGoal()
    {
        msgs.BlinkActionGoal message = new msgs.BlinkActionGoal();
        msgs.BlinkGoal message_content = new msgs.BlinkGoal();
        message_content.bg_color = bg_color;
        message_content.colors = colors;
        message_content.blink_duration = blink_duration;
        message_content.blink_rate_mean = blink_rate_mean;
        message_content.blink_rate_sd = blink_rate_sd;
        message.goal = message_content;
        return message;
    }
}
```

Abbildung 4.6: LEDActionClient, Beispiel einer Action Client Implementierung

#### 4.4.8 Zugriff auf den Behavior/Pose Controller

Action Clients werden auch beim Behavior und Pose Controller verwendet. Der Behavior Controller ermöglicht es Anwendungen (Behaviours), welche beispielsweise mit der Choreographie Suite erstellt wurden und auf dem NAO gespeichert sind, aufzurufen. In diesem Projekt nutzen wir diese Möglichkeit, um die Hände über vorgespeicherte Skripte zu öffnen und zu schließen. Da die Behaviours anhand ihres Namens aufgerufen werden, lässt sich diese Anwendung um beliebig viele weitere Behaviours ergänzen.

Der Pose Controller funktioniert auf die gleiche Weise. Auch hier können vorinstallierte Posen über Ihren Namen aufgerufen werden. Derzeit werden die Posen „Crouch“ zum in die Hocke gehen und „StandZero“ (siehe Abbildung 4.7) zum Aufstehen, beziehungsweise Initialpose einnehmen verwendet. Hierdurch wird der Arbeitsbereich des NAO um einen kleinen Bereich erweitert. Die Pose Library des NAO lässt sich auch um weitere Posen erweitern, welche mittels Choreographie Suite auf den NAO überspielt werden können.

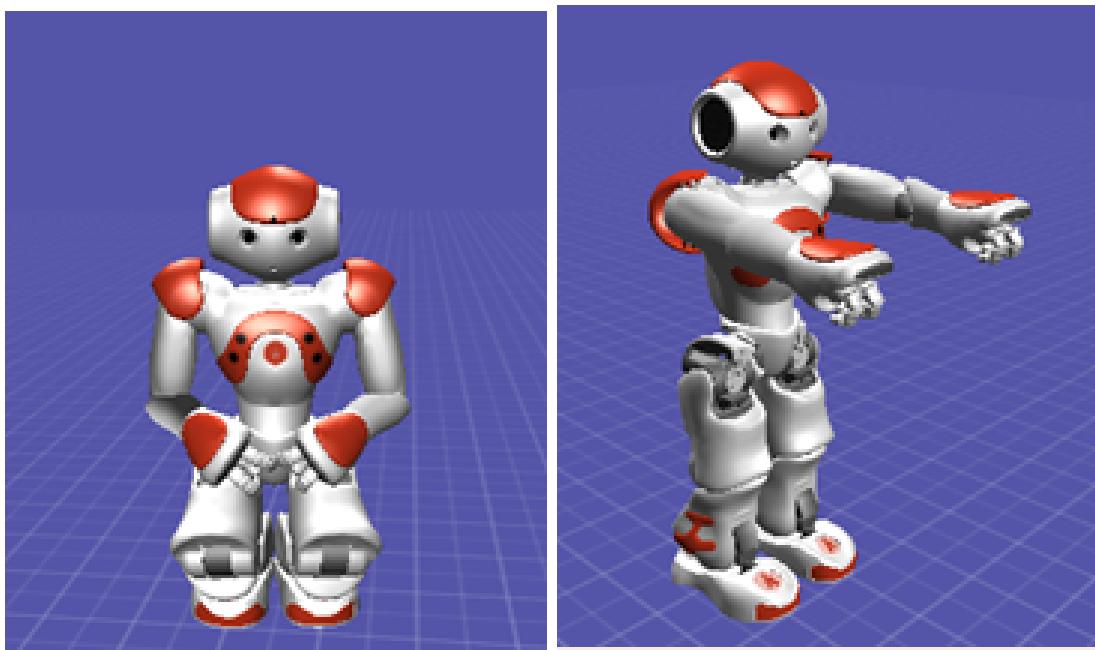


Abbildung 4.7: Predefined Poses 'Crouch'(links) 'StandZero'(rechts) Quelle: [b]

Die Schnittstelle zwischen NAO und Unity umfasst mit diesen Funktionen die grundlegenden Elemente zur Verwendung des NAO. Gleichzeitig dient sie jedoch als Beispiel, wie über Action Clients Service Calls Publisher und Subscriber unter ROS-Sharp verwendet werden können und sind somit Anleitung für die Anbindung weitere Funktionen des NAO.

## 4.5 Einrichten einer VR-Umgebung in Unity

Unity wird nach dem Beispiel von [dK19] für die Verwendung einer HTC Vive eingerichtet. Hierfür sind folgende Vorehrungen nötig:

- Installation von Steam und SteamVR
- Einrichten der HTC Vive über den Assistenten von SteamVR
- Installieren des SteamVR Plugins für Unity über den Unity Asset Store
- Einfügen der im Plugin enthaltenen Prefabs “CameraRig” in die Szene

Wird nun bei verbundener Brille die Simulation gestartet, werden automatisch die Positionen der Controller und des Headsets in den virtuellen Raum übernommen.

## 4.6 Entwurf des Kamerasytems

Aus der Analyse anderer Arbeiten und den Erkenntnissen aus eigenen Tests stellt sich heraus, dass die im Nao integrierte Kamera den Anforderungen des Projekts nicht genügt. Die niedrige Bildwiederholrate und hohe Latenz verhindert ein effektives Interagieren mit der Umwelt des Nao. Daher wird im Folgenden alternative Hardware verglichen um ein geeigneteres System aufzubauen.

### 4.6.1 Auswahl einer geeigneten Kamera

Aus den in ”Problemstellungen der Virtual Reality” (S.11) beleuchteten Problemen lassen sich die folgenden Anforderungen für das Kamerasytem ableiten:

- **Bildwiederholrate:** Ein flüssiges Bild ist wichtig, damit das Gehirn die Bewegung die gesehen wird richtig interpretieren kann. Hersteller von VR-Brillen verwenden daher meistens Bildwiederholraten zwischen 60 und 120 Frames Per Second (FPS), im Falle der HTC VIVE sind es 90 FPS. Für das Kamerasytem ist es daher wichtig, dass die unterstützte Bildwiederholrate nicht substanziell unter der von den Brillen gesetzten Norm liegt.

- **Verzögerung:** Geringe Verzögerung ist eine ebenso wichtige Eigenschaft der Kamera, da sich die Aufnahme-, Verarbeitungs- und Sendezeit der Kamera zu der Gesamtzeit von Eingabe der Bewegung bis Ausgabe des Bildes dazu addiert.

Über die beschriebenen Einschränkungen durch VR hinaus resultieren im Bezug auf den NAO weitere Faktoren.

- **Gewicht:** Da die Kamera am Kopf des NAO montiert ist, sollte das Gewicht der Kamera möglichst gering sein, damit die Mechanik des Kopfgelenks nicht belastet wird. Ebenfalls führt höheres Gewicht zu geringerer Beschleunigung und Genauigkeit in der Synchronisation mit dem VR-Headset. Folgen davon sind im Punkt "Motion Sickness" im Kapitel "Problemstellungen der Virtual Reality" (S.11) beschrieben.
- **Übertragung:** Um den Roboter in seinen Bewegungen nicht einzuschränken ist eine kabellose Übertragung des Bildes wünschenswert. Dies resultiert zwar in einer erhöhten Latenz, die Möglichkeit dass der NAO sich allerdings im Kabel verfängt und umfällt oder die Länge des Kabels überschreitet ist aber nicht tragbar.

## Webcam

Webcams bieten die Möglichkeit kostengünstig Videomaterial live auf den PC zu übertragen. Kabellose Varianten sind allerdings selten wie schwer, und auch die Verzögerung liegt weit über akzeptablen Werten mit üblichen Latenzen über einer Sekunde. Ebenfalls sind Bildwiederholraten von 15-30 FPS Standard, was für das Projekt unzureichend ist.

## Action-Cam

Einige Hersteller von Action-Cams (z.B. GoPro) bieten eine Livestreamfunktion an. Kameras dieser Kategorie sind zwar auf niedriges Gewicht und hohe Bildwiederholraten ausgelegt, die Latenz des Streams beläuft sich allerdings auf bis zu 500ms.

### First Person View (FPV) Kamera

FPV Kameras, eingesetzt im Sektor von ferngesteuerten Renndrohnen, lösen die selben Probleme die das Projekt aufwirft. Sie sind klein und leicht, damit sie auf der Drohne montierbar sind und das Flugverhalten nicht maßgeblich beeinflussen und haben Bildwiederholraten von üblicherweise 60 FPS und Latenzen von ca. 30ms, um dem Piloten bei Geschwindigkeiten von über 100 km/h schnell nähernde Hindernisse darzustellen und sind selbstverständlich kabellos. Darüber hinaus wird das Bild über ein analoges PAL/NTSC Signal gesendet, dadurch erleidet man bei schlechter Verbindung keinen Abbruch im Stream, sondern hat lediglich ein Rauschen im Bild

#### 4.6.2 Auswahl der Komponenten

Aus diesem Vergleich geht die FPV Kamera als beste Wahl hervor. Aufgrund des relativ niedrigen Preises liegen zwei parallele Kamerasysteme um ein Stereoskopisches Bild zu erzeugen im Budget. Beschafft wurden zwei Foxeer Predator V4 mit 2.5mm Linsengröße. Nach [Osc16] besitzt das Auge ein Sichtfeld äquivalent zu einer 3mm großen Linse. Daher wurde für die Kamera eine Linsengröße gewählt, die dem Auge möglichst nahe kommt um das Bild nicht zu stark verzerrten zu müssen. Passend dazu wurde ein Video Transmitter von TeamBlackSheep [Bla] gewählt. Betrieben werden die beiden Kamerasysteme über einen einzelnen Lithium Polymer Akku mit 7.4 Volt und 300 Milliamperestunden. Um das Bild am PC empfangen zu können, wurde Hardware der Firma FUAV eingesetzt, welche das 5.8 GHz Signal empfängt und an den PC über USB weiterleitet. Der Rechner erkennt den Empfänger als eine Webcam, somit kann sie auch einfach in Unity über eine WebCamTexture eingebunden werden.

## 4.7 Prototypisierung der Kamera Halterung

Für die Halterung der Kamera sind folgende Punkte wichtig:

- Die beiden Kameras müssen absolut parallel ausgerichtet sein, ansonsten entsteht kein stereoskopisches Bild welches das Gehirn interpretieren kann.
- Die Halterung muss die Gelenke und Sensoren des NAO beachten und dafür sorgen, dass keine der Komponenten den Roboter beeinträchtigt.

- Die Halterung sollte möglichst leicht sein und fest sitzen, damit sie bei Bewegungen nicht wackelt.

Für die Modellierung der Halterung wird Blender [Fou] verwendet. Um das Modell zu verwirklichen wird die additive Manufaktur in Form eines 3D-Druckers verwendet.

### 4.7.1 Prototyp 1

Der erste Prototyp dient ausschließlich dem Test der stereoskopischen Sicht. Form und Einschränkungen des NAO sind hierbei nicht betrachtet.

Ergebnis des ersten Tests: Kein 3D-Effekt im Bild erkennbar, belastet die Augen.

Grund: Kameras sind nicht exakt parallel ausgerichtet. Dies ist zurückzuführen auf ein beigelegtes Bauteil, das eine Einstellung der Ausrichtung auf zwei Achsen erlaubt, in Abbildung 4.8 mit einem roten Pfeil markiert. Diese Einstellungsmöglichkeit erschwert hier die parallele Ausrichtung. Die Problematik der Halterung ist in Abbildung 4.8 überspitzt dargestellt.

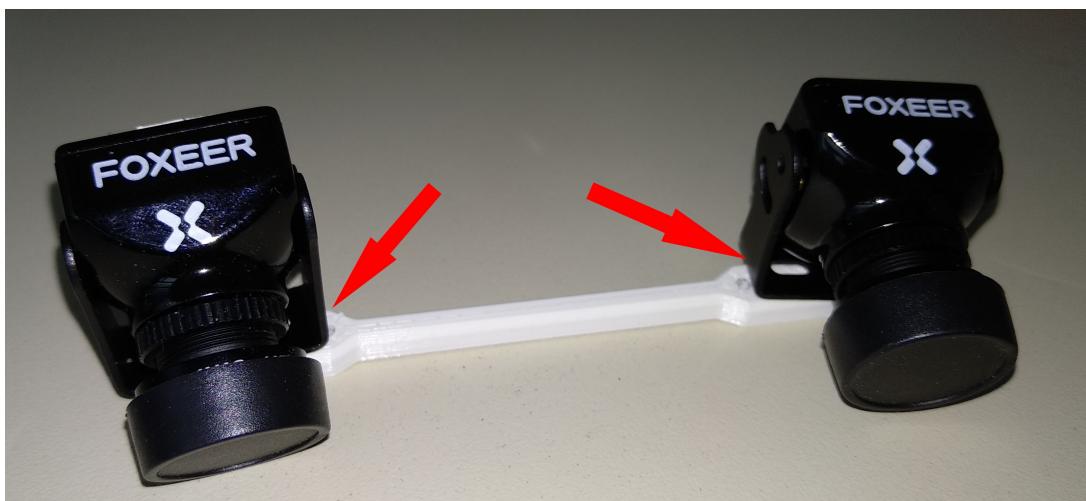


Abbildung 4.8: Version 1 der Kamera Halterung.

### 4.7.2 Prototyp 2

Der nächste Prototyp beschäftigt sich damit, den Makel des Ersten auszugleichen, indem das Bauteil, das Einstellungsmöglichkeiten erlaubt, durch eine statische Halterung ersetzt wird. Um den Aufbau leichter manövriieren zu können, wurde die Halterung und die restlichen Komponenten auf einem Karton fixiert.

Ergebnis: Nun ist tatsächlich ein 3D-Effekt erkennbar.

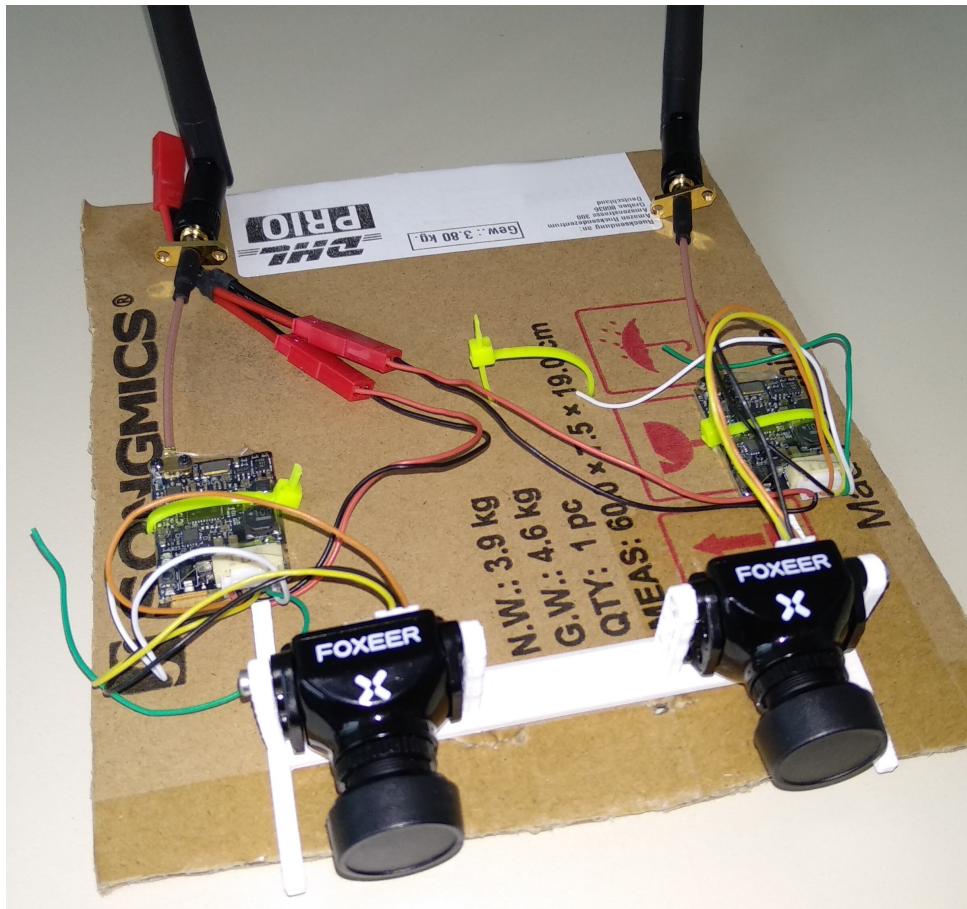


Abbildung 4.9: Version 2 der Kamera Halterung.

### 4.7.3 Prototyp 3

Der dritte Prototyp beschäftigt sich damit eine Halterung für den NAO zu bieten, die fest sitzt und nicht einschränkt. Geplant ist, in die Wölbung um die Lautsprecher herum mit einer klammerähnlichen Konstruktion einzuhaken, inspiriert von einem Modell auf Thingiverse [cos].

Ergebnis: Die Struktur ist für einen 3D-Druck suboptimal. Das Modell bietet keine gerade Fläche, welche der Drucker als Grundfläche verwenden kann. Daher ist viel Supportmaterial nötig. Darüber hinaus ist es nach dem Test zunehmend schwerer vorstellbar, dass selbst nach weiterer Optimierung die Halterung auf der Querachse fest genug sitzt. Ein weiteres Problem ist, dass die Halterung weit von der Stelle entfernt ist, wo die Kameras montiert werden müssen. Damit für den Benutzer keinen Eindruck von verfälschten Proportionen entsteht, müssen die Kameras auch etwa auf Augenhöhe montiert sein. In Abbildung 4.10 wurde der Support noch nicht entfernt, da zuerst die Passgenauigkeit überprüft und anschließend die Idee verworfen wurde.



Abbildung 4.10: Test der Thingiverse inspirierten Halterung.

#### 4.7.4 Prototyp 4

Dieser Prototyp versucht über Gummibänder die nötige Festigkeit herzustellen, die Halterung auf Augenhöhe zu montieren und gleichzeitig die Kameralinsen bei einem Sturz nach vorne zu schützen. Die Sender und Antennen werden dabei oben auf dem Kopf montiert indem sie an das bestehende Gummibandgespann angeschraubt werden. Der Akku sitzt auf der Rückseite in einer Halterung, welche in die Lüftungsschlitzte des Nao einhakt um gleichzeitig ein Verrutschen des Gummibands zu vermeiden.

Ergebnis: Dieser Prototyp sitzt fest, die Kameras sind aber nicht exakt parallel ausgerichtet, da die Platte welche die Kameras befestigt, zu dünn ist. Er ist zwar ein paar Gramm schwerer als strukturell nötig, erhält aber Bonuspunkte dafür, dass er einer VR Brille sehr ähnlich sieht.

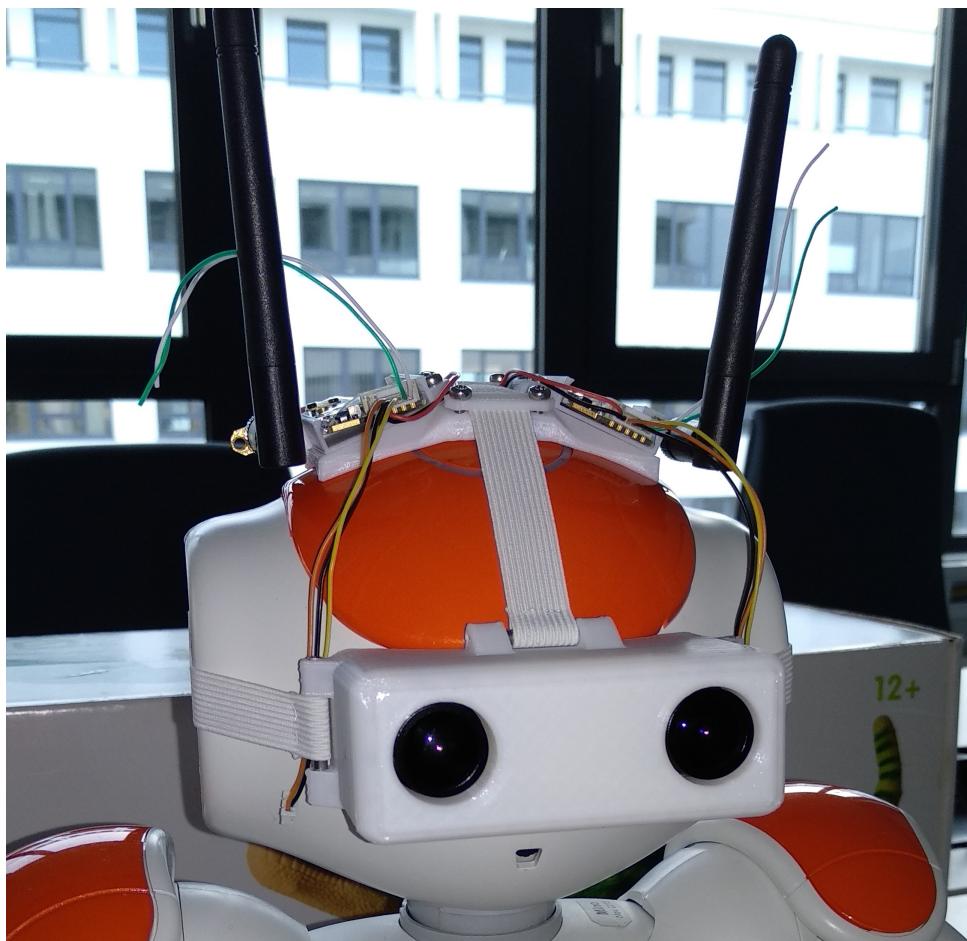
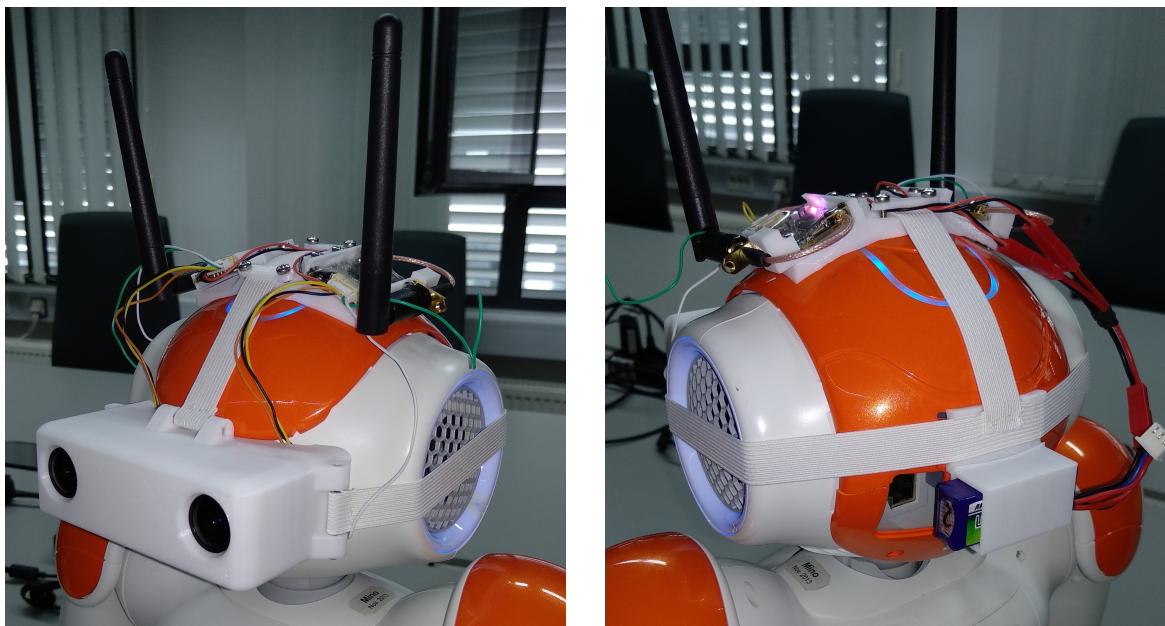


Abbildung 4.11: Version 4 der Kamera Halterung.

#### 4.7.5 Finale Halterung

Aufbauend auf dem Erfolg des letzten Prototyps wurden nun einige Anpassungen am Modell vorgenommen. Der Abstand zwischen den Kameras wurde erhöht, um den 3D-Effekt zu verbessern. Die Platte, auf der die Kameras montiert sind, wurde verstärkt und einige Öffnungen angepasst.



(a) Vorderseite

(b) Rückseite

Abbildung 4.12: Version 5 der Kamera Halterung.

## 4.8 Rendern des stereoskopischen Bildes in Unity

Damit der Benutzer das Bild des Kamerasystems in der virtuellen Umgebung sehen kann, wird die von Unity bereitgestellte WebcamTexture verwendet. Hierfür werden zwei Objekte als Leinwände verwendet. Jedes der Objekte wird nur auf einem der zwei Displays der Brille gerendert, sodass man nun die Möglichkeit hat, das stereoskopische Bild der zwei Kameras anzuzeigen.

Hierfür wird zunächst die Camera Komponente des SteamVR “CameraRig” Prefabs deaktiviert und zwei neue untergeordnete Objekte erstellt, welche beide eine Camera Komponente erhalten. In diesen Kameras für das linke bzw. rechte Auge kann über die Option “Target Eye” das entsprechende Display in der Brille angegeben werden für das

die Kamera das Bild rendert. Untergeordnet davon werden nun die Leinwände platziert und die Layer der Leinwand jeweils auf “RightEye” beziehungsweise “LeftEye” gestellt, welche für diesen Anwendungsfall erstellt wurden. Mithilfe dieser Layer kann nun in der Kamera unter “Culling Mask” die Layer des jeweils anderen Auge deaktiviert werden, sodass auf beiden Augen alles gerendert wird, was nicht explizit für das jeweils andere Auge vorgesehen ist. Hierdurch wird die entsprechende Leinwand ausschließlich auf dem dafür vorgesehenen Auge gerendert und da beide Leinwände übereinander liegen, ist bei einem stereoskopischen Bild ein 3D-Effekt wahrnehmbar.

Da die Leinwände nun Kindobjekte der Kameras sind, wird das Bild unabhängig von der Blickrichtung des Benutzers stets im Blickfeld behalten. Die Leinwände werden noch auf eine Größe skaliert die dem Bildseitenverhältnis der Kameras entspricht und gut in das Sichtfeld der Brille passen.

Um nun ein Bild auf die Leinwände zu werfen, wird ein Skript verwendet, welches die Texturen durch die genannten WebCamTextures ersetzt. Hierfür wird bei der Instanziierung eines WebCamTexture Objekts der Name der Webcam übergeben. Dieser ist im Gerätetmanager von Windows zu finden.

```

private Camera leftEye = new Camera();
private Camera rightEye = new Camera();
private Material renderTextureLeft;
private Material renderTextureRight;

void Start()
{
    // "USB2.0 PC CAMERA" is the name of the USB video receiver
    WebCamTexture leftCamera = new WebCamTexture("USB2.0 _PC_CAMERA");
    WebCamTexture rightCamera = new WebCamTexture("USB2.0 _PC_CAMERA_1");

    leftCamera.Play();
    rightCamera.Play();

    //Left eye
    renderTextureLeft.mainTexture = leftCamera;
    leftEye.targetTexture = (RenderTexture)renderTextureLeft.mainTexture;
    leftEye.stereoTargetEye = StereoTargetEyeMask.Left;

    //Right eye
    renderTextureRight.mainTexture = rightCamera;
    rightEye.targetTexture = (RenderTexture)renderTextureRight.mainTexture;
    rightEye.stereoTargetEye = StereoTargetEyeMask.Right;
}

```

## 4.9 Synchronisierung von NAO Modell und Benutzer

Um den NAO auch tatsächlich steuern zu können, muss die Pose des Benutzers erkannt und auf den NAO übertragen werden. Dies sollte hinreichend akkurat und möglichst performant durchgeführt werden.

### 4.9.1 Kalibrierung der Benutzergröße

Da sich die verschiedenen Benutzer in ihrer Größe unterscheiden, wird zunächst die Größe des Benutzers durch einen Tastendruck am Controller an die Größe des NAO Modells angeglichen. Sobald der Benutzer die vorgegebene Taste betätigt (in diesem Fall den Grip Button), wird die Differenz zwischen der Höhe des Head-Mounted Displays und dem Kopf des virtuellen NAO-Modells berechnet und das virtuelle Benutzeroberobjekt entsprechend skaliert.

### 4.9.2 Errechnen der Benutzerpose

Zunächst werden ausschließlich der Kopf und die Arme betrachtet. Der Kopf ist hierbei ein trivialer Fall da er nur aus einem Gelenk besteht, dessen Winkel direkt übertragen werden können. Da die Arme hingegen aus Schulter, Ellenbogen und Handgelenk bestehen, als Eingabeparameter jedoch nur Position und Rotation der Hand im globalen Koordinatensystem vorhanden sind, muss für die Berechnung ein Rückwärtskinematik-Skript zum Einsatz kommen.

### 4.9.3 Erstellen eines beweglichen NAO Modells

Um das in 4.9.4 eingebundene Skript verwenden zu können, wird ein humanoides Skelett benötigt. Da ein Skelett für den Benutzer wenig visuelle Aussagekraft besitzt, wird ebenfalls noch ein 3D-Modell des NAO benötigt. Das NAO Modell wurde durch die vorhergehende Studienarbeit bereitgestellt. Mit diesem Modell kann nun in Blender ein Skelett aufgebaut werden das den Gelenken des Roboters entspricht. Werden nun die einzelnen Komponenten des NAO Modells einem entsprechenden Knochen zugewiesen, kann das Modell durch Bewegen des Skeletts der physikalischen Begebenheiten des echten NAOs entsprechend, Bewegungen simuliert werden. Damit Unity das Skelett auch verwenden kann, muss es

noch in die “T-Pose” ausgerichtet, sprich in aufrechter Position mit ausgestreckten Armen aufgestellt werden.

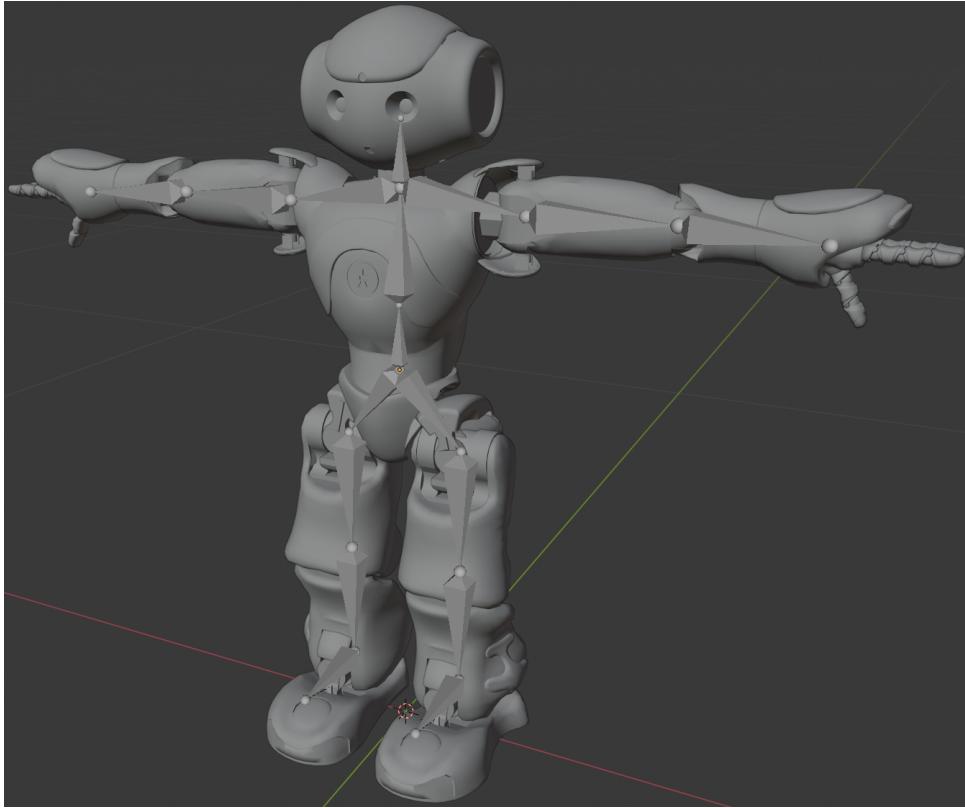


Abbildung 4.13: NAO Modell mit Skelett.

#### 4.9.4 Vorgefertigtes IK Skript

Da die Inverse Kinematik (IK) keine leichte Aufgabe ist, sind die meisten angebotenen Lösungen kostenpflichtig. Da aber ein Erfolg mit besagter Lösung nicht gegeben ist, wurde vorerst auf ein kostenfreies Skript zurückgegriffen, um anhand dessen feststellen zu können, ob bestehende Lösungen, welche die virtuellen Anhaltspunkte des VR-Rigs auf einen zusammenhängenden Körper auf rein visueller Ebene übertragen, den Anforderungen an das Projekt in Hinsicht auf Genauigkeit und Folgeleistung physikalischer Begrenzungen der Gelenke entspricht. Hierfür ausgewählt wurde eine frei verfügbare Lösung von CreateThis [Cre17]. Das Skript verwendet Unity Mecanim, einer API um Animationen dynamisch an ihre Umgebung anzupassen. Nötig für die Verwendung ist ein humanoides Modell mit einem passenden Skelett, welches in 4.9.3 erstellt wurde. Wie in Abbildung 4.14 zu erkennen, bewegt sich das Modell nicht dem in 4.9.3 erstellten Skelett entsprechend.

Da für die Lösung des Problems primär Unity interne Funktionen verwendet wurden, ist auch ein Nachvollziehen beziehungsweise Beheben erschwert. Daher wurde die Idee wieder verworfen.

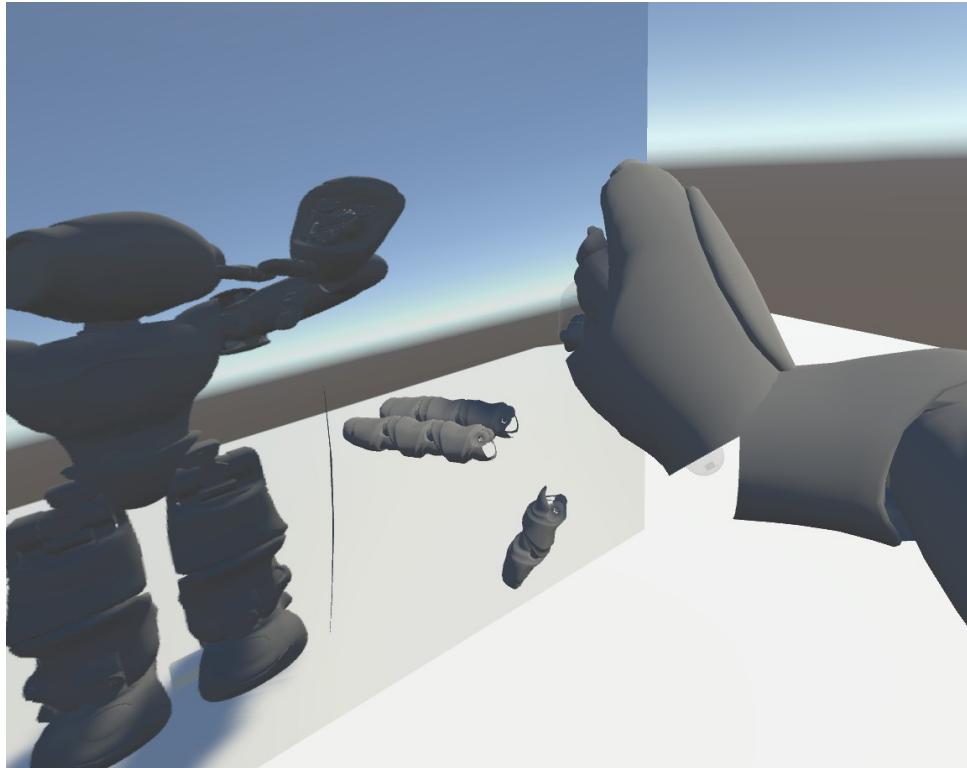


Abbildung 4.14: IK Skript von CreateThis.

#### 4.9.5 Eigens entwickeltes IK Skript

Als idealer Kandidat für diese Aufgabe stellt sich der “Forward And Backward Reaching Inverse Kinematics”, Kurz: FABRIK, Algorithmus von Andreas Aristidou und Joan Lasenby heraus [AL11]. Die gewünschte Position wird hiermit erreicht indem man zuerst den Endeffektor zu seinem Ziel “zieht” und alle folgenden Glieder, aufgrund ihrer physikalischen Beschränkungen, folgen lässt. Anschließend wird der Vorgang rückwärts wiederholt indem die Wurzel der Kinematischen Kette zurück zu ihrer Ursprungsposition gesetzt wird. Dieser “vorwärts und rückwärts” Zug wird iterativ durchgeführt bis entweder der Endeffektor innerhalb einer definierten Abweichung zum Ziel liegt, sich zum vorherigen Durchlauf nichts mehr geändert hat, sprich das Ziel nicht erreichbar ist, oder bis eine maximale Anzahl an Durchläufen überschritten ist.

Dieser Algorithmus sticht besonders durch seine hohe Performanz heraus, da die Berechnung sich auf einfache Addition und Multiplikation von Vektoren beschränkt und sich weiter steigert, je näher der Effektor bereits an seinem Ziel ist. Da die Berechnung 90 mal pro Sekunde durchgeführt wird, um der Bildwiederholrate der VR-Brille gerecht zu werden, sind kleine Schritte gegeben.

Konkret implementiert wird das Ziehen wie folgt: Der Endeffektor wird auf die Position des Ziels gesetzt. Anschließend wird durch alle folgenden Glieder iteriert und die jeweilige neue Position berechnet. Hierfür wird der Richtungsvektor von dem letzten Gelenk zum aktuellen Gelenk bestimmt, normalisiert und mit der Länge des Gliedes multipliziert. Der entstehende Vektor, addiert mit dem Positionsvektor des letzten Gelenks, ist die neue Position des aktuellen Gelenks. Hierbei muss in der konkreten Implementierung, um die Länge des Gliedes zu ermitteln, zwischen dem Vorwärts- und dem Rückwärtsschritt unterschieden werden, da die Länge nur Unidirektional vom Kind zum Elterngelenk gespeichert ist. Wird also gerade ein Kindgelenk an das Elterngelenk angeglichen, muss die hinterlegte Distanz des aktuellen Knotens genutzt werden, andersherum muss die Distanz des letzten Gelenks verwendet werden. Abgesehen von dieser Unterscheidung kann derselbe Algorithmus für die Vorwärts- und Rückwärtsrichtung verwendet werden, wenn man eine Gelenkliste, Anfang, Ende und eine Richtung angibt.

```
void ApplyFabrikToNodes(List<GameObject> nodes, int start, int end, int delta)
{
    for (int i = start + delta; i != end; i += delta)
    {
        float distance = delta == 1 ?
            nodes[i - 1].GetComponent<NodeData>().GetDistance() :
            nodes[i].GetComponent<NodeData>().GetDistance();
        Vector3 last = nodes[i - delta].transform.position;
        nodes[i].transform.position =
            last + (nodes[i].transform.position - last).normalized * distance;
    }
}
```

Allerdings kann der Algorithmus nur die Positionen der Gelenke setzen. Sofern eine Lösung gefunden wurde, müssen nun noch die Gelenke in ihrer Rotation angepasst werden. Hierfür wird erneut durch jedes Element iteriert, beginnend bei der Schulter. Zunächst wird die Rotation des Elterngelenks übernommen, was einer Gelenkrotation von 0 Grad in Roll und Pitch entspricht. Für das Schultergelenk wurde hierfür ein Elternelement eingerichtet was horizontal vom Torso weg zeigt. Nun wird der Roll-Winkel zum Kindgelenk bestimmt, indem mit der Z-Achse des Elterngelenks als Drehachse der Winkel zwischen der Y-Achse

des Elterngelenks und der Richtung zum Kind bestimmt wird. Wenn man nun das aktuelle Gelenk um den errechneten Wert rotiert, kann nun die neue X-Achse als Drehachse dienen, um den Pitch-Winkel zwischen der Z-Achse des Elternknoten und der Richtung zum Kind zu berechnen. Diese zwei errechneten Winkel werden nun hinterlegt, um sie später auf das NAO-Modell zu übertragen. Dieser Vorgang ist beispielhaft dargestellt in Abbildung 4.15. In der Software, die zur Visualisierung verwendet wurde, sind allerdings die Achsen in einer anderen Ausrichtung definiert, daher sind hier X und Y vertauscht.

```
Vector3 parentForward = parent.transform.TransformDirection(Vector3.forward);
Vector3 parentUp = parent.transform.TransformDirection(Vector3.up);
Vector3 childDirection = child != null
    ? (child.transform.position - current.transform.position).normalized
    : Vector3.zero;

current.transform.rotation = parent.transform.rotation;

float roll = parentUp.GetAngleOnAxis(childDirection, parentForward);
current.transform.Rotate(Vector3.forward, roll);

float pitch = parentForward.GetAngleOnAxis(childDirection,
    current.transform.TransformDirection(Vector3.right));
current.transform.Rotate(Vector3.right, pitch);
```

Um den Winkel auf einer Drehachse zu bestimmen, wird jeweils das Kreuzprodukt zwischen der Achse und den beiden Vektoren bestimmt. Die resultierenden Vektoren liegen beide auf einer Ebene orthogonal zur Drehachse. Das Kreuzprodukt bedeutet zwar eine Abweichung von 90 Grad zur ursprünglichen Projektion des Richtungsvektors auf die Ebene, da aber nur der relative Winkel zwischen den beiden Vektoren relevant ist, bleibt das Ergebnis unverändert.

```
public static float GetAngleOnAxis(this Vector3 self, Vector3 other, Vector3 axis)
{
    Vector3 perpendicularSelf = Vector3.Cross(axis, self);
    Vector3 perpendicularOther = Vector3.Cross(axis, other);
    return Vector3.SignedAngle(perpendicularSelf, perpendicularOther, axis);}
```

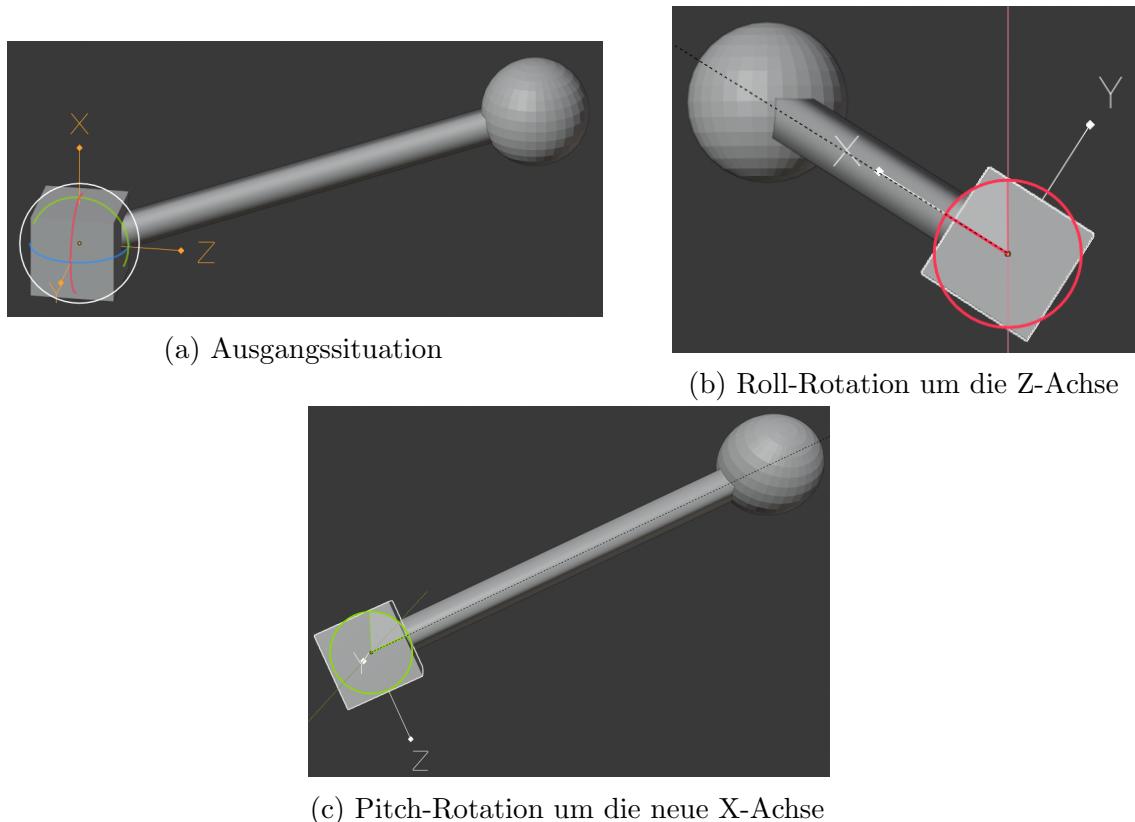


Abbildung 4.15: Beispielhafter Aufbau des Gelenkwinkelszenarios.

Im konkreten Anwendungsfall wird als dreidimensionale Grundlage ein GameObject mit dem Namen “KinematicArmModel” erstellt, welches die oben beschriebenen Nodes beziehungsweise Gelenke des NAO im richtigen Abstand und ausgerichtet in der NAO Initialpose abbildet.

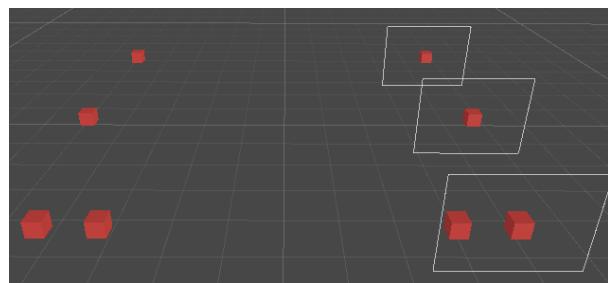


Abbildung 4.16: KinematicArmModel in der Initialpose.

Dabei repräsentieren die jeweils äußeren drei Knoten Schulter, Ellenbogen und Hand (In Abbildung 4.16 von vorne nach hinten), die zwei inneren Knoten dienen nur den

Schultern als Referenz für die Winkelberechnung. Innerhalb der weißen Umrahmungen an den rechten Knoten können zu Debugzwecken die errechneten Winkel angezeigt werden.

Für weitere benötigte Daten erhält jede Node eine NodeData Komponente. In ihr wird der Elternknoten, sprich das übergeordnete Gelenk gespeichert. Außerdem, falls vorhanden, eine Referenz auf das Debug-Textfeld, jeweils den entsprechenden JointStateWriter aus dem ROS# Modell, um die errechneten Werte übertragen zu können und einen Versatz und eine Skalierung um den Winkel in das andere Modell umrechnen zu können.

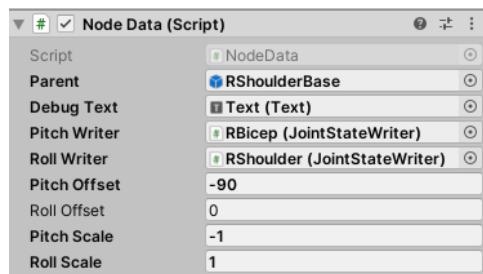


Abbildung 4.17: NodeData des rechten Schultergelenks.

Beim Start der Anwendung wird dann zwischen jedem Knoten und seinem Elternteil die Distanz berechnet und gespeichert und außerdem für die Visualisierung der Gelenkverbindung ein Segment von gleicher Länge initialisiert, was nach jedem erfolgreichen Berechnungszyklus an die neuen Positionen der beiden verbundenen Knoten angeglichen wird.

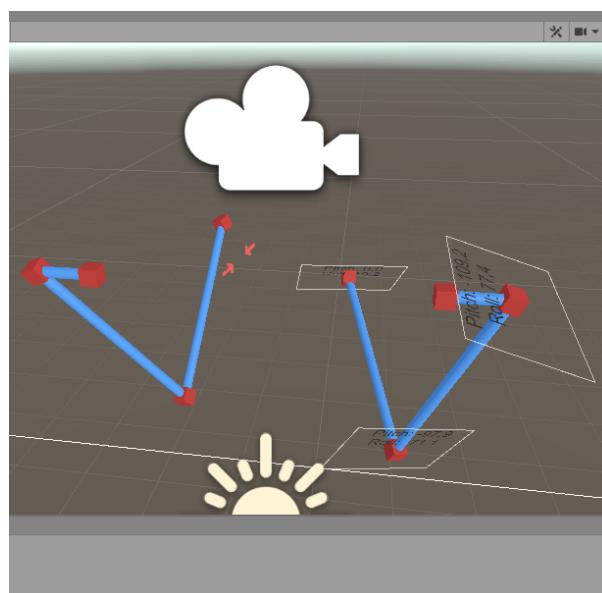


Abbildung 4.18: KinematicArmModel während der Laufzeit.

Um den Effekt des “vorwärts und rückwärts Ziehens” zu erzielen, wird zunächst das von SteamVR bereitgestellte Prefab “CameraRig” an beiden Controllern mit einem Hook erweitert. Dieser Haken bietet die Möglichkeit zwischen dem getrackten Controller und der gewünschten Position, zu der der NAO seine Hand bewegen soll, einen variablen Versatz zu definieren. An diesem Haken wird letztendlich der NAO Arm “gezogen”, sprich zu Beginn eines neuen Zyklus der Handknoten des KinematicArmModel auf die Position des Hooks gesetzt.



Abbildung 4.19: Hierarchie des CameraRig.

Dieses modifizierte CameraRig, im folgenden “CameraRigWithHooks” genannt, kann nun als neuer Prefab gespeichert werden und ist damit in jeder Szene leicht zu implementieren. Damit der Haken auch am Modell einhaken kann, erhält jeder Endeffektor ein IKHook Skript, welches eine Referenz auf den entsprechenden Hook behält.

Insgesamt sieht der Ablauf eines Zyklus nun wie folgt aus:

```

void Update()
{
    if (state >= StateManager.State.calibrated)
    {
        hookedNodeChains.ForEach(ApplyFabrikIK);
        hookedNodeChains.ForEach(SendJointAngles);
        UpdateSegments();
    }
}
  
```

hookedNodeChains ist dabei eine Liste, welche alle zusammenhängenden Knoten in weitere Listen unterteilt. Es wird also für jede Gelenkkette der Fabrik Algorithmus durchgeführt, die berechneten Winkel an das NAO Modell gesendet und anschließend die Segmentpositionen aktualisiert. Die Funktion des StateManagers wird in Kapitel 4.11 ”Benutzerführung durch States“ (S.47) genauer erläutert.

## 4.10 Vereinigung der Teilprojekte

Nun können die Teilprojekte 4.8: "Rendern des stereoskopischen Bildes in Unity" (S.36), 4.9: "Synchronisierung von NAO Modell und Benutzer" (S.38) und 4.4: "NAO Schnittstelle für Unity" (S.19) in einer Szene zu einer vollständigen Telepräsenz kombiniert werden. Hierfür kann die Einbindung des Kamerasytems analog zu 4.8 durchgeführt werden. Auch das KinematicArmModel und das CameraRigWithHooks von 4.9 kann einfach als Prefab gespeichert und in der neuen Szene platziert werden. Um nun die errechnete Pose an den NAO zu übertragen, werden die in 4.9.5 errechneten Winkel von Gradmaß zu Bogenmaß umgerechnet und anschließend an den für die Gelenkachse zuständigen JointStateWriter aus ROS# übertragen. Der passende JointStateWriter wird über die NodeData festgelegt.

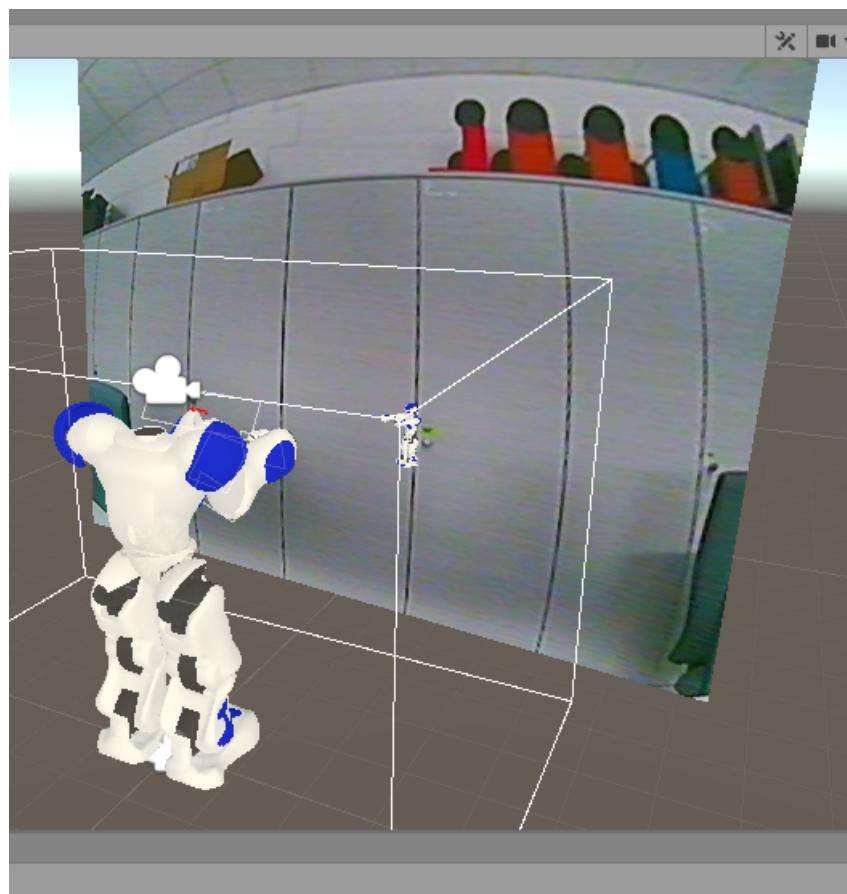


Abbildung 4.20: Zusammenführung aller Komponenten.

## 4.11 Benutzerführung durch States

Für die Sicherheit des Roboters und ein Heranführen des Benutzers an die Steuerung der Software wurde die Anwendung in verschiedene Zustände unterteilt, kontrolliert durch den StateManager. In jedem Zustand erhält der Benutzer Anweisungen, wie er weiter vorzugehen hat, gleichzeitig wissen alle Komponenten die auf den Zustand achten, was der Benutzer tun darf und was nicht. Somit kann ein Umfallen und Schäden am NAO verhindert werden, indem die Bewegungen des Benutzers nur in einem “Armed” State übertragen werden.

Damit Komponenten deren Aktionen vom Zustand abhängen Zugriff auf diesen haben, wurde eine Abstrakte Klasse “StateListener” implementiert, welche die Komponenten erben. Mithilfe dieser gemeinsamen Superklasse, kann der StateManager alle implementierenden Klassen über einen Zustandswechsel informieren und die Komponenten selbst, auch den aktuellen Zustand beim StateManager, einholen. Somit bewegt sich der reale NAO nur im Zustand “Armed” und auch der virtuelle NAO wird erst visualisiert, wenn mindestens der Zustand “Calibrated” erreicht wurde. Für den Wechsel des Zustands achtet der StateManager auf Benutzereingaben über den Controller und/oder auf die Erfüllung von Prädikaten. Das Interface “Predicate” setzt eine Methode “CriteriaMet()” voraus, welche von bisher nur einer Klasse implementiert wird, der Klasse “Distance-Checkpoint”. Diese Komponente stellt sicher, dass zwei GameObjects sich innerhalb einer vordefinierten Distanz befinden. Dieses Prädikat wird angewandt um zu überprüfen, ob der Benutzer korrekt positioniert ist und ob er sich beim Aktivieren der Synchronisation in der Initialposition befindet. Die Szene wird dabei im Informationsgehalt immer an den jeweiligen Zustand durch den “SceneManager” angepasst. Der SceneManager blendet im aktuellen Zustand irrelevante Objekte aus, gibt Hinweise auf das weitere Vorgehen, indiziert benötigte Eingaben und zeigt den aktuellen Zustand an.

## 4.12 Bedienung der Software

Bevor die Anwendung in Unity gestartet werden kann, muss zunächst das ROS-System gestartet werden. Unabhängig davon, ob ROS auf einer VM oder einem eigenen Rechner ausgeführt wird, müssen dabei folgende Befehle ausgeführt werden: (siehe Abbildung 4.21)

```
// startet die ROS Kernprogramme, notwendig zur Verwendung von ROS
$ roscore

// startet die ROS-Bridge, auf welche sich Unity verbinden kann
$ roslaunch rosbridge_server rosbridge_websocket.launch

// verbindet ROS mit dem NAO und startet alle Anwendungen welche
// zur Verwendung der Schnittstelle notwendig sind
$ roslaunch naoBringup nao_full_py.launch nao_ip:=192.168.100.12
roscore_ip:=192.168.100.11 network_interface:=enp0s81|
```

Abbildung 4.21: Befehle zum Starten aller notwendigen ROS-Anwendungen

Jeder Befehl sollte in seinem eigenen Terminal ausgeführt werden und in der hier gelisteten Reihenfolge. Ebenfalls wichtig ist die Prüfung der IP-Adressen und Interfaces, welche beim Nao\_full\_py.launch übergeben werden. Stimmen Netzwerkadressen und Interfaces nicht überein wird ein Error angezeigt und die Anwendungen werden abgebrochen. Starten alle drei Anwendungen ohne Fehler (Footprint Error kann ignoriert werden) ist der NAO erfolgreich mit ROS verbunden und die Bridge wurde aufgebaut sodass sich Unity mit ihr verbinden kann. Sollte Unity erneut, zum Beispiel aufgrund eines Verbindungsabbruchs oder einer Beendigung der Anwendung erneut mit ROS verbunden werden, empfiehlt es sich die ROS-Bridge zu beenden und neu zu starten. Dies ist nicht zwangsläufig notwendig, verhindert jedoch Fehler seitens ROS-Bridge beim Managen von Nodes.

Für Unity gelten folgende Voraussetzungen für die Lauffähigkeit der Anwendung:

- ROS muss mit Unity verbunden sein, ansonsten wirft nach einer Zeit der ROS Connector einen Fehler. (Korrekte IP Adresse des ROS-Core angeben)
- SteamVR muss gestartet und die Vive mit beiden Controllern eingeschalten sein.
- Das Programm muss in Unity über “Play” gestartet werden.
- Stiffness muss aktiviert sein (In der Anwendung F1 zum Toggeln der Stiffness)

- Der Anwender kann nun, oder bereits vorher, die Brille aufsetzen.

Zu Beginn wird der Benutzer begrüßt und dazu aufgefordert, sich richtig zu positionieren. Das verhindert zum Einen, dass der Anwender zu weit am Rand steht und somit mit der realen Umgebung kollidiert oder die Stationen der Vive nicht mehr richtig tracken kann, als auch, dass für spätere Berechnungen Körperposition und Ausrichtung vorausgesetzt werden können, da diese nicht zuverlässig aus der Position des HMD geschlossen werden können.

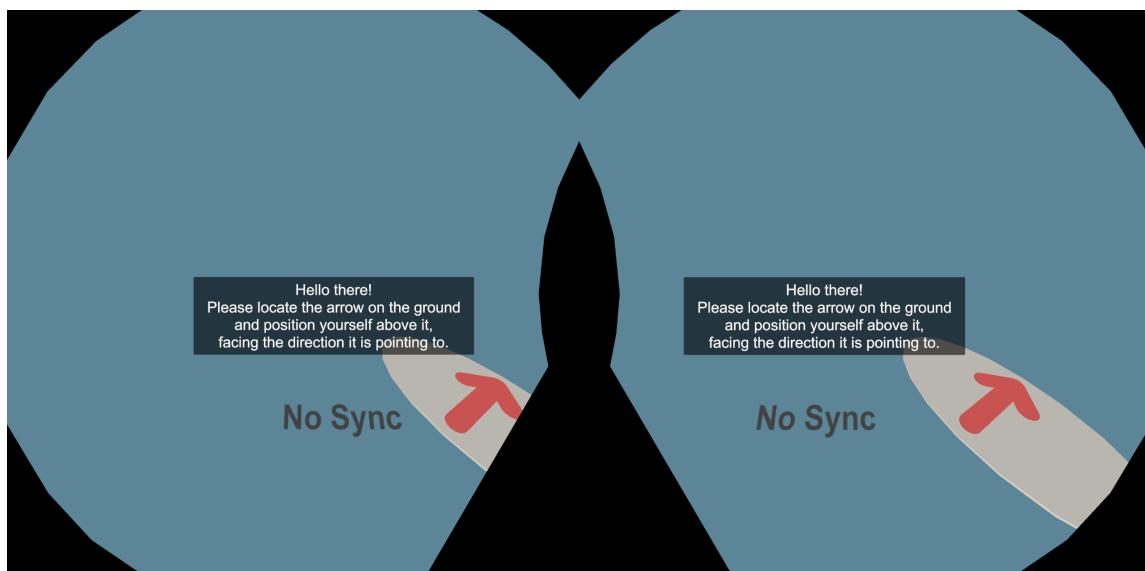


Abbildung 4.22: UI Initialzustand

Im folgenden Schritt wird der Benutzer aufgefordert nach vorne zu sehen und den rechten GripButton zu betätigen, welcher beim virtuellen Modell durch rote Pfeile hervorgehoben wird. Hierdurch kann die Größe des Benutzers an das NAO Modell angeglichen werden, indem das virtuelle CameraRig skaliert wird. Dass der Benutzer nach vorne sieht, ist wichtig da beim Hoch- beziehungsweise Heruntersehen sich auch die Höhe des getrackten Headsets verändert, welches hier als Referenz für die Höhe dient.

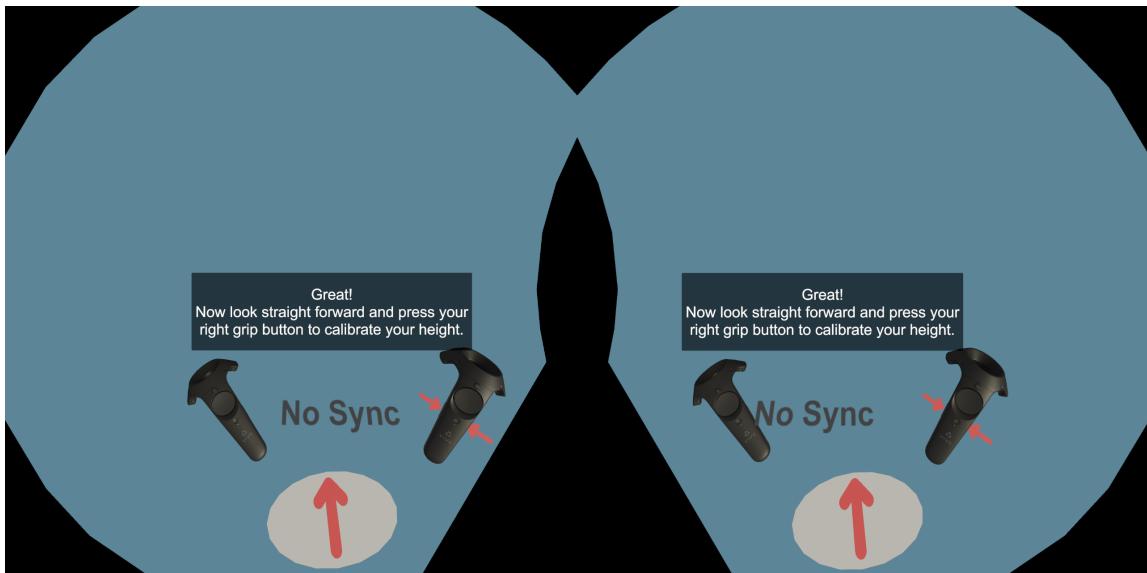


Abbildung 4.23: Benutzer hat die Position eingenommen

Anschließend wird der Benutzer informiert, dass er mit dem linken GripButton die Synchronisation starten kann und mit dem linken Touchpad den NAO navigiert. Die zwei grünen Markierungen sorgen dafür, dass der Benutzer nur in der NAO Initialpose die Synchronisation starten kann, sodass der Roboter zu Beginn keine ruckartigen Bewegungen macht und umfällt. Aus selbigem Grund kann der Benutzer auch nur mit dem NAO navigieren, wenn er die Synchronisation unterbricht.

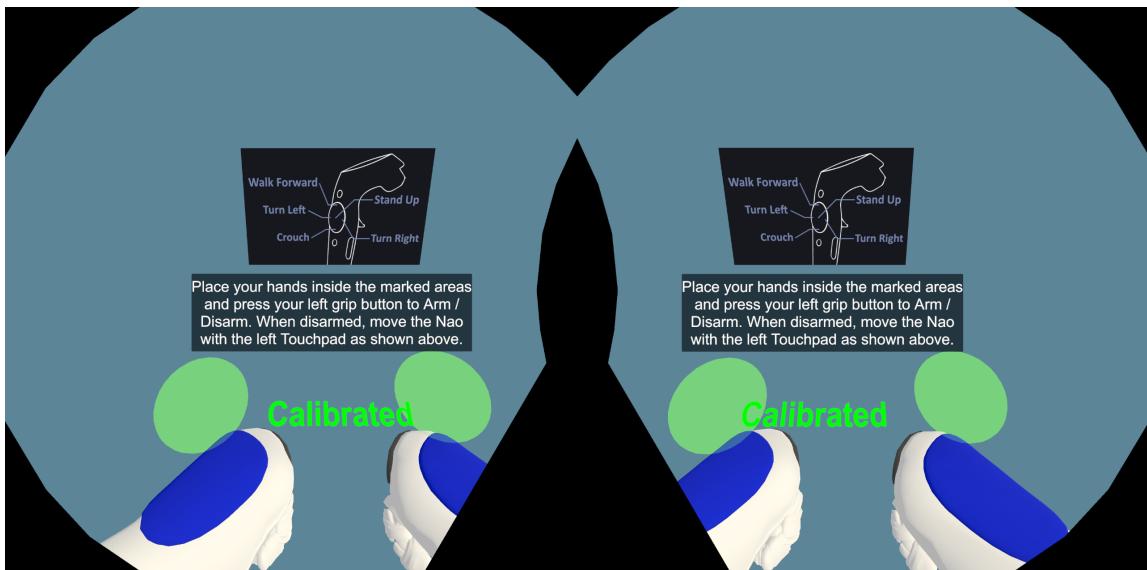


Abbildung 4.24: Benutzer hat seine Höhe kalibriert

Nun ist der Benutzer synchronisiert. Das kleine NAO Modell bildet dabei ab, in welcher Pose sich der reale NAO momentan befindet. Auf die schwarzen Flächen wird das Kamerabild geworfen, sofern Sie verbunden sind.

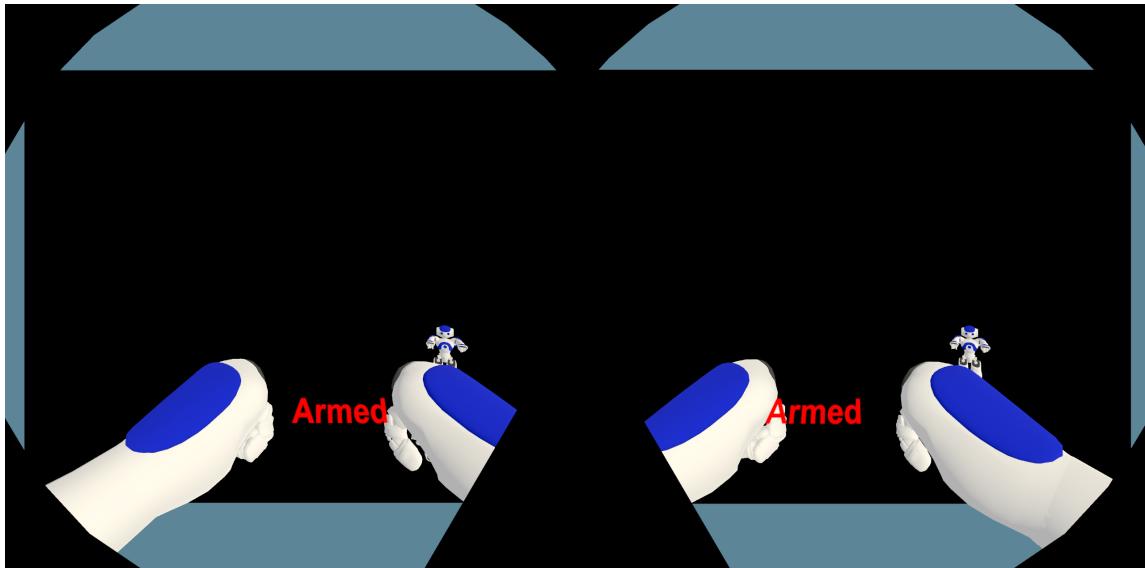


Abbildung 4.25: Benutzer hat die Synchronisierung gestartet

Der Benutzer hat nun die Möglichkeit die Arme und den Kopf frei zu bewegen. Ebenso kann er mit dem Trigger an der Unterseite des HTC-Vive Controller die Hände schließen. Wenn der Benutzer den synchronisierten Zustand verlässt, kann er über die Pfeiltasten den NAO dazu bringen sich um die eigene Achse zu drehen oder nach vorne zu Laufen. Auch kann der NAO dazu gebracht werden sich in die Hocke zu begeben, oder aus dieser aufzustehen. In der neuen Position kann wieder der synchronisierte Zustand eingenommen werden. Alle Tastenbelegungen können Sie erneut in Abbildung 4.26 einsehen.

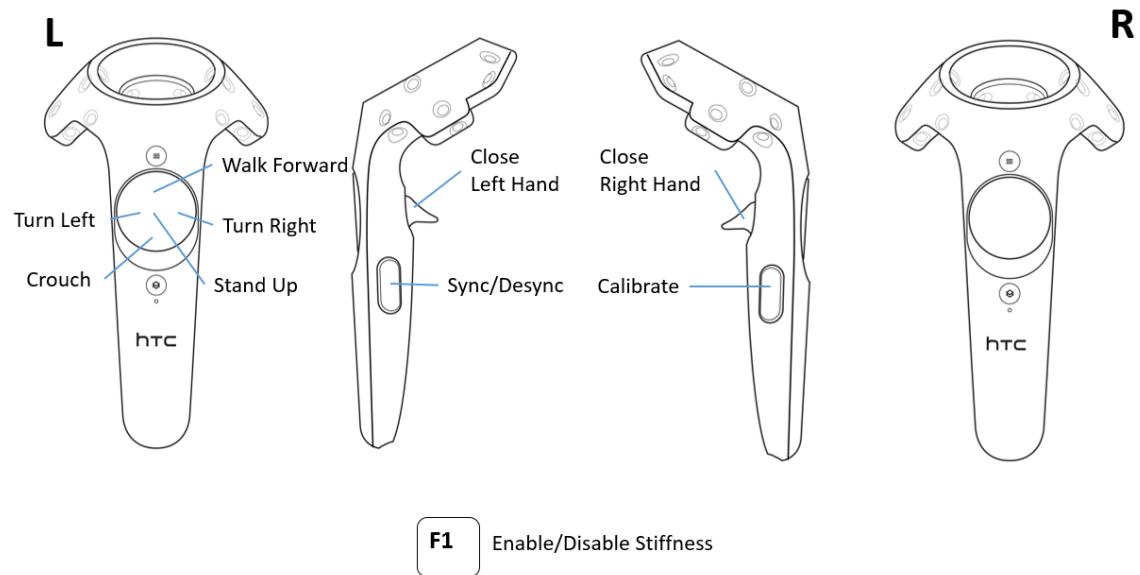


Abbildung 4.26: Tastenbelegung der Anwendung

# 5 Evaluation

## 5.1 Untersuchung des Bottlenecks in ROS

Schon beim erstmaligen Aufbau der Verbindung zwischen Vive und NAO, bei welchem lediglich die Kopfbewegungen übertragen werden, sind große Latenzzeiten aufgefallen. Beim Debuggen des Versuchsaufbaus, indem sich das Topic von einem Terminal subscriben lässt, fällt auf, dass die Latenz zwischen der Bewegung der Vive und dem Eintreffen der neuen Werte am Topic zu gering ist um Ursache für die stark auftretenden Verzögerungen zu sein. Hieraus lässt sich ableiten, dass der Hauptverlust nach Eintreffen der neuen Parameter am entsprechenden Topic geschieht. Ein Test der Netzwerkverbindung zwischen Remote-PC und dem Router sowie dem NAO zeigt, dass auch nicht das drahtlose Netzwerk für die Abweichungen von der Echtzeit verantwortlich sein dürften. Denn auch bei einer Übertragung über RJ49 Anschluss traten die Verzögerungen auf.

Unsere Schlussfolgerung war, dass es nur zwei mögliche Positionen für das Bottleneck geben kann. Die erste wäre der NAO selbst. Sollte der NAO zu lange brauchen, um die Bewegungsparameter (Joint States) zu verarbeiten beziehungsweise anzuwenden, wäre dies ein nicht überbrückbares Hindernis, da wir weder in die Basisprogrammierung des NAO noch in dessen Hardwarespezifikationen eingreifen können. Videos von ähnlichen Projekten sowie deren Programmcode, lassen jedoch darauf schließen, dass ein Bottleneck an dieser Stelle sehr unwahrscheinlich ist.

Daher verstärkt sich die Untersuchung auf die zweite Möglichkeit, nämlich dass ROS selbst Probleme mit der Verarbeitung der Datenmenge hat. Wie in den zu vorigen Kapiteln „Entwicklungsumgebung“ und „Inbetriebnahme ROS“ beschrieben, wird die ROS Distribution in einer virtuellen Maschine ausgeführt und muss neben dem Empfang, der Verarbeitung und der Weiterleitung aller NAO Parameter, auch das ROS Core System betreiben und den ROSBridge Server. Gerade innerhalb einer VM kann diese Datenmenge, die zur Verfügung gestellten Ressourcen übersteigen.

Um das Problem zu verifizieren, sollte ein Ressourcen Monitoring unter Volllast die tatsächliche Auslastung der VM anzeigen. Gegebenenfalls sollen die Ressourcen der VM heraufgesetzt werden. Da ein Heraufsetzen der Ressourcen jedoch dem Hostsystem, auf welchem Unity mit SteamVR läuft, Ressourcen entzieht, ist auch hier ein Gleichgewicht

abzuwägen. Alternativ wäre es auch, dank der ohnehin eingesetzten ROSBridge möglich, das System zu dezentralisieren und damit das ROS-Netzwerk auf einen anderen Rechner auszulagern. Dadurch wäre der Ressourcenkampf zwischen beiden Systemen gelöst, jedoch erhöht dies den Netzwerktraffic über den Router und könnte dabei zu einem neuen Bottleneck führen.

Zwei vergleichsweise radikale alternative Lösungsansätze wären der Umstieg von ROS auf ROS 2, welches nach Entwicklerangaben für Echtzeitanwendungen besser gerüstet sein soll. Hierbei müsste jedoch validiert werden, in welcher Form die ROS Packages für NAOqi und die ROSBridge mit ROS 2 kompatibel sind. Eine knappe Analyse zeigt zudem, dass eine Inbetriebnahme von ROS 2 auf Windows zwar möglich, jedoch ausgesprochen umständlich ist. Eine Inbetriebnahme auf Linux hingegen könnte wieder zur Ressourcenabwägung zwischen den beiden OS Systemen führen.

Ein ebenfalls radikaler Ansatz wäre es ROS selbst in das Betriebssystem des NAO einzuspeisen, sodass alle Berechnungen der NAOqi Schnittstelle auf dem NAO selbst stattfinden. Auch dieser Ansatz ist suboptimal, da an dem von der Dualen Hochschule zur Verfügung gestellten NAO möglichst wenig verändert werden soll und der Aufwand zum Einspielen und Zurückspielen auf den alten Zustand immens ist. Beide alternativen Lösungsansätze sind in ihrer Komplexität hoch genug, damit sie die Grundlage für ein Folgeprojekt bilden könnten.

Für dieses Projekt genügte ein Umverteilen der Ressourcen zu Gunsten des Linux-Gastsystems um die Verarbeitung der Gelenkwinkeländerungen stark zu beschleunigen. Zwar wird hierdurch keine Echtzeit erreicht, doch die Umsetzung der Bewegungen erfolgt nun flüssig. Ebenfalls dienlich für die Performance ist das Aufteilen von Unity und ROS auf zwei verschiedene Rechner. Dies ist problemlos möglich, da die Kommunikation über die ROSBridge erfolgt, welche ein lokaler Server ist, auf welchen das gesamte Netzwerk zugreifen kann (siehe Abbildung 5.1).



Abbildung 5.1: ROS und Unity auf räumlich getrennten Systemen verbunden durch den lokal gehosteten Server der ROSBridge.

## 5.2 Latenz im Kamerabild

Um festzustellen, welche Verzögerung zwischen der Aufnahme eines Bildes und der Anzeige besagten Bildes auf dem Headset besteht. Hierfür wird der Versuch, wie in Abbildung 5.2 aufgebaut. Der NAO filmt dabei die rechte Seite des Bildschirms, auf welcher eine Stoppuhr läuft. Auf der linken Seite wird in Unity das Bild aus der Szene gerendert. Somit kann mit einem Screenshot die vergangene Zeit berechnet werden.



Abbildung 5.2: Versuchsaufbau für Messung der Bildlatenz.

Folgende Annahmen werden für den Versuch gemacht:

- Die Ausgabe des Bildes auf der linken Seite des Bildschirms und die Ausgabe auf den Displays der VR-Brille geschieht mit einem vernachlässigbaren Zeitunterschied.
- Die Distanz und Verbindungsqualität beeinflussen nicht die Latenz der Übertragung, gegeben durch die Eigenschaften einer analogen Bildübertragung.
- Die Genauigkeit der Stoppuhr wird limitiert durch die Bildwiederholrate des Bildschirms, in diesem Fall  $\frac{1}{60}s \approx 16,6ms$

Die Ergebnisse lauten:

- **Windows Stopwatch:**  $4,27 - 4,2 = 70ms$
- **timeanddate.de:**  $4,288 - 4,189 = 99ms$
- **online-stopwatch.chronme.com:**  $14,755 - 14,669 = 86ms$

Im Durchschnitt erhält man hiermit 85 Millisekunden. Da die Abweichung in beide Richtungen unterhalb der festgelegten maximalen Abweichung von 16 Millisekunden durch den Bildschirm liegt, ist dieser Wert durchaus realistisch.

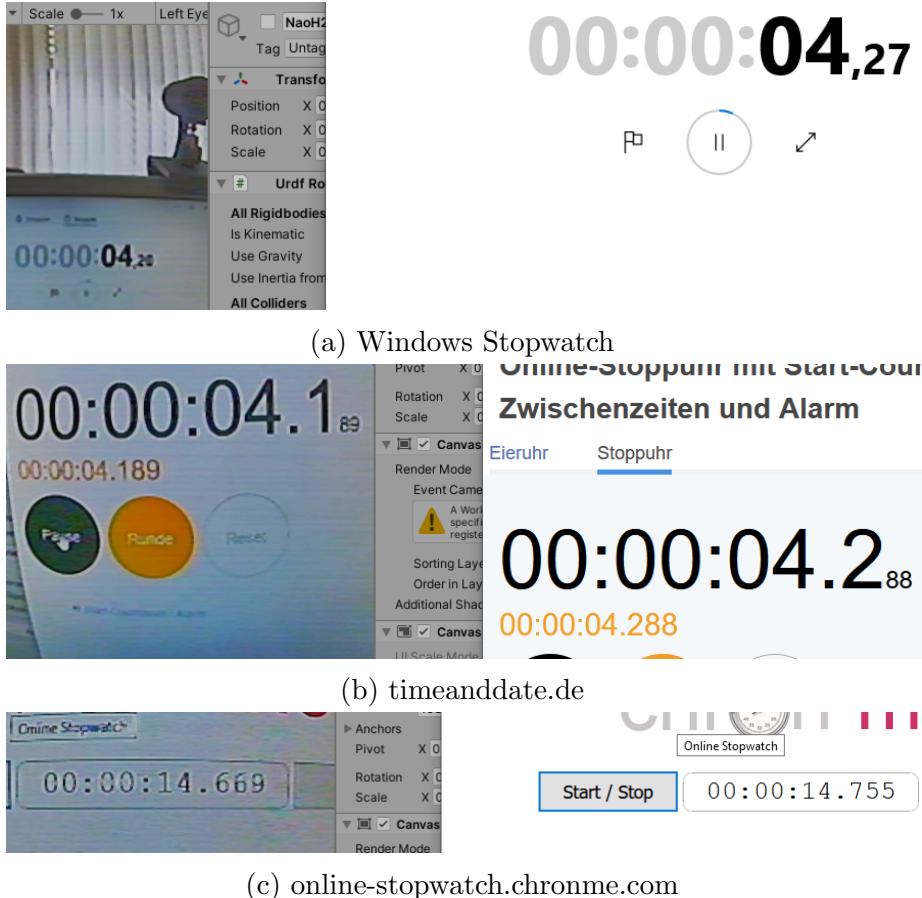


Abbildung 5.3: Ergebnisse der Messung

### 5.3 Latenz in der Ausführung von Befehlen

Hierfür kann eine Aufnahme analysiert werden, in der sowohl Benutzer als auch NAO zu sehen sind. Das verwendete Video, welches auch in Abbildung 5.4 dargestellt ist, läuft mit 50 Bildern pro Sekunde, also 20 ms pro Bild. Sei die Ausgangsposition Bild 0 = 0 ms, dann ist der Beginn der Bewegung beim NAO in Bild 18 = 360 ms zu sehen. Daraus ergibt sich eine Latenz von 360 Millisekunden, addiert mit der Latenz der Kamera entsteht für den Anwender eine Latenz von  $85ms + 360ms = 445ms$ , also etwa einer halben Sekunde.

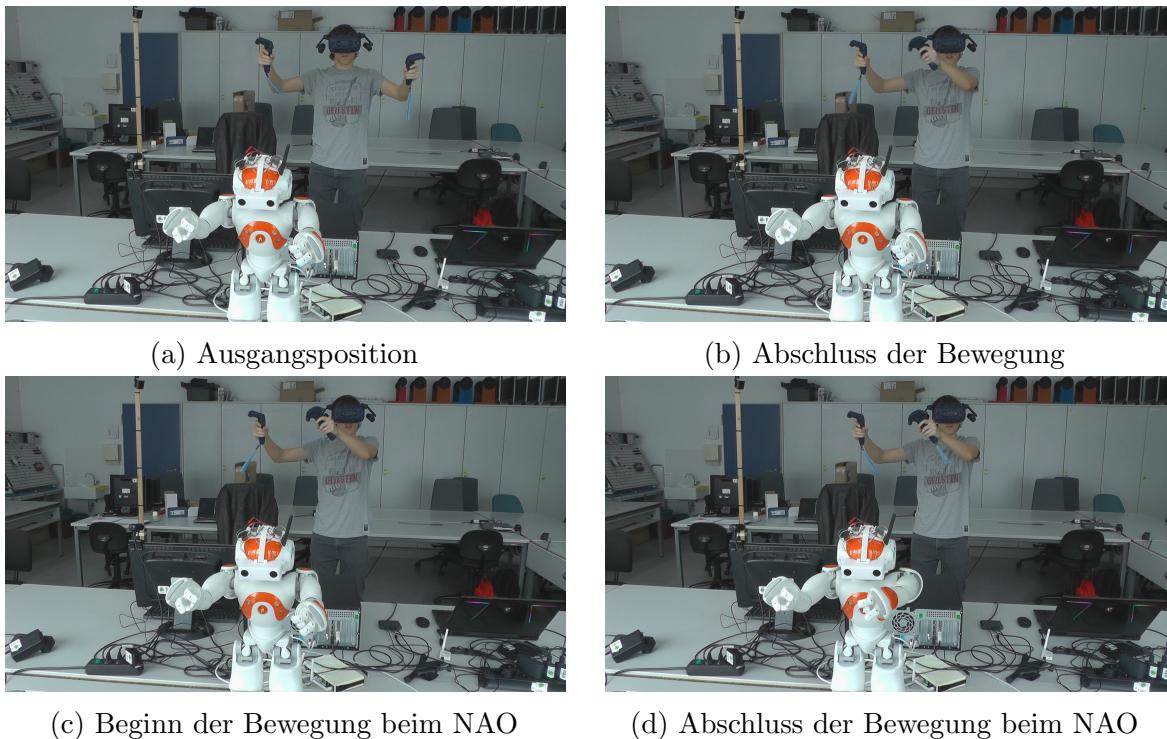


Abbildung 5.4: Ausschnitte aus einem Video

## 5.4 Usability Test

Gerade dieses Projekt hätte von einem Usability-Test mit einer Vielzahl an Personen mit unterschiedlichem technischen Hintergrund stark profitiert. Die verständliche Bedienbarkeit dieser Anwendung durch eine breite Personenzahl ist Grundlage, um die Qualität zu bestimmen. Hierzu wäre es jedoch notwendig, eine Gruppe an Testpersonen aufzustellen, diese mit einzelnen Aufgaben zu betrauen und am Ende die Erfahrungsberichte zu erfassen. Dies ist zum Zeitpunkt des Projektendes (Ende Mai 2020) leider rechtlich sowie gesellschaftlich, auf Grund der COVID-19 Pandemie, schwer bis gar nicht umzusetzen. Die Versammlungsbeschränkungen und die Einschränkungen der Betretungsrechte der DHBW schließen solche Tests aus. Auch die Ausarbeitung und Umsetzung eines Hygienekonzeptes, welches regelmäßige Desinfektionen und Abstandseinhaltungen bei Befragungen einschließt, machen die Durchführung eines Usability-Tests nicht realisierbar.

# 6 Fazit

## 6.1 Rückblick

Rückblickend betrachtet, waren besonders die Neuheit von ROS-Sharp als Technologie sowie die schlechte Dokumentation der NAO Packages für ROS problematisch für die Anbindung des NAOs an Unity.

ROS-Sharp ist ein hervorragendes Tool, um Unity und ROS miteinander zu verbinden. Jedoch ist die Technologie mit einem Release im Mai 2019 sehr jung, weshalb nur wenige Projekte diese Software verwenden und deshalb kaum Beispielcode einzusehen ist. Die Entwickler haben sich hierbei mit ein paar Tutorials und Beispielcodes selbst geholfen. Erschwerenderweise befassen sich diese oft mit sehr spezifischen Funktionalitäten, weshalb es schwer ist, gelerntes aus den Tutorials oder den Beispielen für das eigene Projekt zu übertragen. Auch ist niemals sicher, ob der Fehler bei einem selbst liegt oder es sich einfach noch um einen Bug oder ein noch nicht vollständig implementiertes Feature handelt.

Sobald ein Feature jedoch verstanden wurde, war die Implementierung und Übertragung auf anderer Situationen sehr intuitiv. Besonders die Möglichkeit sich die MessageTypes über das Menü automatisch generieren zu lassen, erweist sich in der Praxis als ausgesprochen nützlich und erspart während der Entwicklung viel Zeit. Somit erwies sich ROS-Sharp als mächtiges und ausgesprochen nützliches Tool, welches besonders wenn die Funktionalität weiterhin so schnell ausgebaut wird und ein paar weitere Beispielcodes und Tutorials hinzukommen zu einer hervorragenden Schnittstelle zwischen ROS und Unity wird.

Enttäuschend hingegen waren die NAO Packages für ROS. Hier war die Dokumentation lückenhaft, widersprüchlich und teilweise sogar falsch, was stundenlanges Debuggen und ein Vorankommen durch trial-and-error zur Folge hatte. Hier ging dem Projekt viel Zeit verloren, welche mit einer besseren Dokumentation hätte gespart werden können. Auch waren einzelne Anwendungen fehlerhaft, da sich scheinbar Dateinamen im Laufe der Entwicklung der Packages geändert haben und von uns manuell repariert werden mussten. Auch waren interne Probleme in den Packages meist Auslöser für Abstürze, welche uns zu spät und an falscher Stelle angezeigt wurden. Dies hat dazu geführt, dass wir kein Vertrauen in diese Schnittstelle haben und von einer weiteren Verwendung abraten, bis diese ein Refactoring durchlaufen haben.

Aus Seite der Videoübertragung war rückblickend die Wahl von analogen Kameras die Richtige, allerdings hätten Linsen mit einem weiteren Sichtfeld verwendet werden sollen. Beim Test der Kameras sah das Bild durchaus verwendbar aus, im Verbund mit dem NAO wurde allerdings festgestellt, dass durch die geringe Reichweite der Arme der Handlungsspielraum des NAO sich hauptsächlich außerhalb des Sichtfeldes befindet. An der Halterung musste festgestellt werden, dass die Videotransmitter heiß genug werden, damit sich das gedruckte Plastik verformt. Das beeinträchtigt zwar nicht die grundlegende Funktion der Halterung, die Sender haben mit der Zeit aber nur noch wenig Halt. Hier hätte statt dem für 3D-Drucker üblichen PLA (Polylactid) ABS oder PETG verwendet werden sollen. Das Gummiband hingegen hat durchweg für guten Halt und keinen Spielraum gesorgt.

Die Wahl des Aufbaus der virtuellen Szene hat sich als gute Entscheidung herausgestellt. Durch die feste Ausrichtung des Benutzers vor dem Start der Synchronisation kann sichergestellt werden, dass die Bewegungen nicht falsch interpretiert werden. Zum einen von den HTC-Vive Basisstationen, weil der Benutzer eventuell außerhalb des Sichtfeldes steht, zum anderen, weil die Software eventuell die Position in der der Benutzer steht falsch interpretiert, da sich von keinem der Tracker auf die Ausrichtung des Oberkörpers schließen lässt. Somit kann der Fehler minimiert werden und damit auch die unter dem Punkt "Problemstellungen der Virtual Reality" (S.11) beschriebene Motion Sickness. Ebenfalls ist mit der festen Standposition auch die räumliche Begrenzung eingehalten, da der Benutzer sich nicht bewegt.

Der verwendete Algorithmus für das rückwärtskinematische Problem "Fabrik", erweitert mit der Gelenkwinkelberechnung, hat dem Projekt gut gedient. Nur die physikalischen Gelenkwinkelbegrenzungen des NAO einzuhalten, hat sich mit dieser Herangehensweise als äußerst schwierig herausgestellt und wurde im Verlauf des Projekts verworfen. Schäden durch Überdrehung der Gelenke konnten dadurch vermieden werden, dass der NAO von selbst stoppt, sofern ein Gelenk am Anschlag ist. Der einzige Verlust ist hier also, dass bei Überdehnung das virtuelle NAO Modell, welches diese Funktionalität nicht hat, also nicht mehr exakt mit dem realen Modell übereinstimmt.

## 6.2 Zusammenfassung

Nach Projektabschluss wurden alle gesetzten Ziele dieses Projektes erreicht. So ist die Übertragung der Arm- und Kopfbewegungen des HTC Vive Nutzers auf den NAO

Roboter möglich. Dabei konnten alle zuvor ausgewählten Komponenten, Schnittstellen und Technologien verwendet werden. Darüber hinaus konnte auch eine Live Übertragung des Roboterblickfeldes in das HTC Vive Headset aufgebaut werden. Hierfür konnten jedoch nicht die internen Kameras des NAO verwendet werden, da deren Latenz unzureichend ist. Stattdessen mussten analoge Kameras am Roboter angebracht werden, hierbei stellte sich jedoch heraus, dass das Field-of-View der Kameras großzügiger hätte ausfallen können, da Bewegungen am Rand des Sichtfeldes von den Kameras nicht erfasst werden.

Auch das Tracking unterliegt Einschränkungen. Hier konnten Latenzen nicht vollständig beseitigt werden, was bei vielen Bewegungen zu Verzögerungen führt. Zu guter Letzt ist auch die Zuverlässigkeit des Behaviors zum Schließen und Öffnen der Hände unzureichend. Gemäß unserer Erwartungen sprengten optionale Funktionalitäten, wie das Übertragen der Audiosignale, welche der NAO aufzeichnet und das Tracken der Beine mittels zusätzlicher Vive Sensoren, den Rahmen des Projektes und konnten deshalb nicht implementiert werden.

Dennoch zeigt dieses Projekt, wie die direkte Kopplung der Roboterbewegungen an die Bewegungen eines menschlichen Piloten durch ein VR-System möglich ist. Auch ist die Schnittstelle des NAO Roboters ausführlicher ausgefallen als zunächst angenommen, was den Weg für viele weitere Projekte ebnet.

## 6.3 Wirtschaftliche Betrachtung

Einen wirtschaftlichen Vorteil bietet dieses Projekt nicht. Denn selbst wenn Latenzen und Schwachstellen gänzlich abgeschafft werden würden, stellte der NAO keine nennenswerte Arbeitskraft dar, gerade auch im Vergleich zu dem Menschen, welcher ihn steuert. Langfristig kann eine 1:1 Steuerung von humanoiden Robotern jedoch relevant werden. Gerade dann, wenn wie im Jahre 2020 zwischenmenschliche Kontakte ein Gesundheitsrisiko sind, da erhöhte Infektionsgefahr besteht. Ein gesteuerter humanoider Roboter könnte hier der Avatar sein, welcher zu Veranstaltungen oder zum Einkaufen geschickt wird. Derzeit ist dieser Gedanke jedoch sehr futuristisch und aufgrund des finanziellen Kostenaufwands und fehlender Infrastruktur für die Masse unvorstellbar.

Da dieses Projekt jedoch auch keinesfalls wirtschaftlichen Mehrwert erzielen sollte, sondern die Machbarkeit einer 1:1 Steuerung von humanoiden Robotern mit der HTC-Vive

aufzeigen sollte entspricht dieses Ergebnis auch den wirtschaftlichen Erwartungen, welche an dieses Projekt gestellt wurden.

## 6.4 Ausblick

Auf der Grundlage dieser Arbeit können Folgeprojekte zum einen die unter den Zielen (1.1) verfassten optionalen Features sowie weitere ungeplante Funktionen hinzufügt werden. Die folgenden Unterkapitel beschreiben ein paar mögliche Ansätze dafür.

### 6.4.1 Tracking der Beine

HTC bietet neben seinem Headset noch weitere Tracker an, die parallel dazu verfolgt werden können. Diese Tracker können an den Beinen des Benutzers befestigt und somit in der virtuellen Szene abgebildet werden. Dadurch ist es möglich, das bisherige in die Hocke gehen und wieder aufstehen, welches bisher noch über vordefinierte Posen gelöst wurde, durch ein direktes Ansteuern der Motoren, analog zu den Armen, zu ersetzen. Auch Gehen und Drehen ließe sich hiermit ersetzen. Der Roboter hat hierbei allerdings, auch mit dem vorgefertigten Ablauf, bereits große Probleme nicht um zu fallen. Da dieses Ziel bereits in dieser Studienarbeit als optional gesetzt wurde, sind hierfür bereits die benötigten Tracker vorhanden und eine Halterung, womit sich der Tracker an den Beinen oder anderen zylindrischen Gegenständen mit Klettverschluss befestigen lässt, realisiert wurde.

### 6.4.2 Übertragung von Ton

Sprache ist ein wichtiges Medium für Interaktion. Daher bietet es sich an, einen bidirektionalen Audiokanal zwischen dem NAO und der HTC Vive aufzubauen. Hierfür können jeweils die Mikrofone und Lautsprecher verwendet werden. Sofern die Verbindung steht, können auch mit den noch unbelegten Tasten der Controller andere Geräusche oder vorgefertigte Sätze ausgegeben werden. Diese könnten als Warnsignale oder Hinweise an die Umwelt des NAOs genutzt werden. Auch die Möglichkeit Sätze im Headset, mittels Sprachanalyse, aufzuzeichnen und vom NAO abzuspielen wäre denkbar.

### 6.4.3 NAO Beschränkungen in Unity

Damit das virtuelle NAO Modell besser mit dem Realen übereinstimmt, können die Beschränkungen, die die Gelenke des NAO mit sich bringen, bereits in Unity implementiert werden. Dies kann entweder direkt bei der Berechnung der inversen Kinematik einbezogen werden oder es wird anschließend die erreichte Position mit den Begrenzungen überschrieben.

### 6.4.4 Wifi Kameras

Die hier verwendeten Kameras haben eine äußerst geringe Latenz, allerdings besteht der Nachteil, dass sich die Reichweite auf etwa 100m ohne Hindernisse beschränkt. Da der größte Anteil der Latenz durch den Nao entsteht, kann hier ein Kompromiss eingegangen und das Kamerasystem durch eine oder mehrere über Wifi verbundene Kameras ersetzt werden.

### 6.4.5 Ausbau der NAO Schnittstelle/Reale Echtzeit

Die derzeitige Schnittstelle umfasst schon viele Funktionen der ROS-NAOqi-Schnittstelle, jedoch noch nicht alle. Somit könnte diese weiter ausgebaut und erweitert werden. Auch Modifikationen der bestehenden ROS Packages zur Steigerung der Performance sind möglich. Alternativ könnte auch eine Anpassung für weitere ROS Distributionen, wie Melodic, angestrebt werden oder eine Portierung auf ROS2, wenn sich hierdurch die Echtzeitfähigkeit erreichen würde.

# Literaturverzeichnis

- [a] Action interface. [https://thinkcse.files.wordpress.com/2015/01/action\\_interface.png?w=393&h=161&zoom=2](https://thinkcse.files.wordpress.com/2015/01/action_interface.png?w=393&h=161&zoom=2). [Online], Zugriff am 20.01.2020.
- [b] Predifined poses. [http://doc.aldebaran.com/2-1/family/robots/postures\\_robot.html](http://doc.aldebaran.com/2-1/family/robots/postures_robot.html). [Online], Zugriff am 20.01.2020.
- [AL11] Andreas Aristidou and Joan Lasenby. FABRIK: A fast, iterative solver for the inverse kinematics problem. *Graph. Models*, 73(5):243–260, September 2011.
- [Ben17] Oliver Bendel. Virtuelle realität. <https://wirtschaftslexikon.gabler.de/definition/virtuelle-realitaet-54243>, 2017. [Online], Zugriff am 31.05.2020.
- [Bis] Martin Bischoff. ros-sharp wiki. <https://github.com/siemens/ros-sharp/wiki>. [Online], Zugriff am 20.05.2020.
- [Bla] Team Blacksheep. Tbs unify pro 5g8 hv - race (sma) video transmitter. [https://www.team-blawsheep.com/products/prod:unify\\_pro\\_hv\\_race](https://www.team-blawsheep.com/products/prod:unify_pro_hv_race). [Online], Zugriff am 20.01.2020.
- [cos] costashatz. Referenzmodell auf thingiverse. <https://www.thingiverse.com/thing:341087>. [Online], Zugriff am 03.03.2020.
- [Cre17] Createthis. unity vr ik mecanim. [https://github.com/createthis/unity\\_vr\\_ik\\_mecanim](https://github.com/createthis/unity_vr_ik_mecanim), 2017. [Online], Zugriff am 26.05.2020.
- [dK19] Eric Van de Kerckhove. Htc vive tutorial for unity. <https://www.raywenderlich.com/9189-htc-vive-tutorial-for-unity>, 2019. [Online], Zugriff am 23.01.2020.
- [Fou] Blender Foundation. 3d modeling software blender. <https://www.blender.org/>. [Online], Zugriff am 20.01.2020.
- [IS] Jackie Kay Ioan Sucan. Ros wiki urdf. <http://wiki.ros.org/urdf>. [Online], Zugriff am 20.05.2020.

- [Osc16] Oscar. Fpv camera lens fov angle of view (fov) chart. <https://intofpv.com/t-fpv-camera-lens-fov-angle-of-view-fov-chart>, 2016. [Online], Zugriff am 20.01.2020.
- [Rec] Igor Recio. Ros/startguide. <http://wiki.ros.org/ROS/StartGuide>. [Online], Zugriff am 20.05.2020.
- [ROBa] ALDEBARAN ROBOTICS. Robocup history. <https://web.archive.org/web/20130730132316/http://www.aldebaran-robotics.com/en/Solutions/For-Robocup/history.html>. [Online], Zugriff am 20.05.2020.
- [Robb] Open Robotics. Ros.org. <https://www.ros.org/>. [Online], Zugriff am 20.05.2020.
- [Robc] Softbank Robotics. About/company. <https://www.softbankrobotics.com/emea/en/company>. [Online], Zugriff am 20.05.2020.
- [Robd] SoftBank Robotics. Aldebaran dokumentation choreographe suite. <http://doc.aldebaran.com/2-4/software/choreographe/index.html>. [Online], Zugriff am 20.05.2020.
- [Robe] SoftBank Robotics. Nao - technical guide. <http://doc.aldebaran.com/2-1/family/index.html>. [Online], Zugriff am 19.05.2020.
- [Robf] SoftBank Robotics. Nao 6 preliminary marketing datasheet user guide. [https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwiyn5f4pb\\_pAhUILewKHWEqA94QFjAAegQIAxAB&url=https%3A%2F%2Frobotics.ostechnology.co.jp%2Fwp-content%2Fthemes%2FostRobots\\_1811%2F\\_pdf%2FNAOV6\\_Datasheet\\_EN.pdf&usg=A0vVaw2E0j\\_V-FGqdrKkehvRfRm4](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwiyn5f4pb_pAhUILewKHWEqA94QFjAAegQIAxAB&url=https%3A%2F%2Frobotics.ostechnology.co.jp%2Fwp-content%2Fthemes%2FostRobots_1811%2F_pdf%2FNAOV6_Datasheet_EN.pdf&usg=A0vVaw2E0j_V-FGqdrKkehvRfRm4). [Online], Zugriff am 20.05.2020.
- [Robg] SoftBank Robotics. Naoqi documentation center. <http://doc.aldebaran.com/>. [Online], Zugriff am 20.05.2020.
- [Robh] SoftBank Robotics. robotshop.com. <https://www.robotshop.com/de/de/zorabots-nao-v6-mit-zora-solution.html>. [Online], Zugriff am 20.05.2020.
- [Rob19] Dyno Robotics. Teleoperated nao-robot through vr-interface. <https://www.youtube.com/watch?v=PUn5A76dlJs>, 2019. [Online], Zugriff am 05.12.2019.