DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Entwicklung einer Gestiksteuerung mittels Kinect für den humanoiden Roboter Nao

Studienarbeit

für die Prüfung zum

Bachelor of Engineering

des Studienganges Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Lukas Essig
und
Michael Stahlberger

Abgabedatum 12. Mai 2014

Bearbeitungszeitraum	24 Wochen
Matrikelnummer	8898018, 1367912
Kurs	TINF11B3
Betreuer	Herr Prof. Hans-Jörg Haubner

Erklärung

Gemäß §5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

_____ _____

Ort Datum Unterschrift

_____

Unterschrift

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Ziel der Arbeit . . . 1
- 1.2 Inhalt . . . 2
- 1.3 Umfeld & Strukturelles . . . 2

2 Grundlagen Robotik **4**
- 2.1 Geschichte . . . 4
- 2.2 Definition . . . 7
  - 2.2.1 Roboterarten . . . 8
- 2.3 Komponenten . . . 10

3 Kinect **11**
- 3.1 Kinect Hardware . . . 11
  - 3.1.1 RGB-Kamera . . . 12
  - 3.1.2 Tiefenerkennung . . . 12
  - 3.1.3 Tilt-Motor . . . 14
  - 3.1.4 Microphon Array . . . 14
- 3.2 Kinect Software . . . 14
  - 3.2.1 Der Tiefenstream . . . 15
  - 3.2.2 Der Skeletonstream . . . 16
- 3.3 Kinect SDK . . . 18
  - 3.3.1 Timeline . . . 18
  - 3.3.2 Grundfunktionen . . . 19
  - 3.3.3 Skelettfunktionen . . . 20
  - 3.3.4 Kinect Studio . . . 22

4 Nao **23**
- 4.1 Nao Hardware . . . 23
- 4.2 Nao Software . . . 26
- 4.3 NAOqi Framework . . . 30
  - 4.3.1 Cross - Plattform/Language . . . 31
  - 4.3.2 NAOqi - Prozess . . . 31
  - 4.3.3 Module . . . 33
  - 4.3.4 .NET SDK . . . 36

5 Realisierung **37**
5.1 Architektur 37
5.1.1 MainWindow 39
5.1.2 SkeletonAngleHandler 39
5.1.3 SkeletonAngleCalculator 40
5.1.4 Interfaces 40
5.1.5 Flussdiagramm: Winkelerkennung 40
5.2 Kinect - Armerkennungsalgorithmus 40
5.2.1 Shoulder Pitch 40
5.2.2 Shoulder Roll 41
5.2.3 Elbow Roll 41
5.2.4 Elbow Yaw 41
5.3 Nao - Armbewegungsalgorithmus 42
5.4 Prototypen 45
5.4.1 Erster Kinect-Prototyp 45
5.4.2 Erster Nao-Prototyp 45
5.4.3 Zweiter Prototyp 46
5.4.4 Endprogramm 46

6 Ausblick **47**
6.1 Aufgetretene Probleme 47
6.2 Weiterführende Anwendungsgebiete 47

Literaturverzeichnis **49**

Abbildungsverzeichnis

2.1 Vaucansons Ente 5
2.2 Mobiler Roboter „Shakey“ 6
2.3 Roboter ASIMO 9

3.1 Aufbau Kinect [?] 11
3.2 Kinect Punktemuster 12
3.3 Kinect Objekte im Sichtraum 13
3.4 Depth Space Range [?] 15
3.5 Kinect Joints [?] 16
3.6 Skelett-Erkennung [?] 17
3.7 Skelett-Koordinatenraum [?] 18

4.1 Übersicht Nao V3.2 24
4.2 Überblick Choreographe 27
4.3 Choreographe Programmierung 28
4.4 Überblick Webots 29
4.5 NAOqi Übersicht 30
4.6 NAOqi Broker 32
4.7 NAOqi Method-Tree 32
4.8 Blocking Call 35
4.9 Non - Blocking Call 35

5.1 UML - Klassendiagramm 38
5.2 Erster Nao-Prototyp 46

1 Gelenkraum linker Arm 48
2 Gelenkraum rechter Arm 49

Tabellenverzeichnis

4.1 Gelenkraum linker Arm 25
4.2 Gelenkraum rechter Arm 25

Listings

3.1 Initialisierung des Kinect Sensors [?] 19
3.2 Verwendung des Skeleton-Streams 21
4.1 Kommunikationsarten Module 34
4.2 Beispiel .NET SDK 36
5.1 Ermittlung des Winkels mithilfe von drei Punkten 41
5.2 Ermittlung des Winkels mithilfe von vier Punkten 42
5.3 Methode controlArm 42
5.4 Methode controlArm erweitert 43
5.5 Pseudocode checkDifference 44

Abkürzungsverzeichnis

DHBW	Dualen Hochschule Baden-Württemberg
SDK	Software Development Kit
API	application programming interface
FPS	Frames per second
IR-Emitter	Infrarot-Emitter
CMOS	Complementary Metal Oxide Semiconductor
VDI	Verein Deutscher Ingenieure

Kapitel 1

Einleitung

In dieser Einleitung wird das zentrale Thema dieser Arbeit beleuchtet, beschrieben in welchem Umfeld die Arbeit entworfen wurde und strukturell erläutert, wie die Zusammenarbeit im Projekt stattgefunden hat. Zudem wird aufgezählt, wie sich diese Arbeit inhaltlich aufbaut.

1.1 Ziel der Arbeit

Zentrales Thema der Arbeit ist die Interaktion zwischen Mensch und Roboter. Dabei soll ein Modell des humanoiden Roboters Nao durch die Gestik seines Benutzers gesteuert werden. Der Roboter ahmt die Bewegungen nach, die der Benutzer zuvor ausgeführt hat. Zudem sollen bestimmte Steuergesten implementiert werden, die den Roboter in verschiedene Modi versetzen, in denen er gleiche Gesten auf verschiedene Weise interpretiert. Die Gestik-Erkennung soll mit Hilfe eines Xbox Kinect Moduls erfolgen. Von dieser Stereokamera werden bestimmte Gesten aufgezeichnet und über eine selbst implementierte Software verarbeitet und an den Nao-Roboter weitergeleitet, der diese dann nachahmt. Dabei ist darauf zu achten, dass keine Bewegungen von Nao ausgeführt werden, die ihm mechanisch oder elektronisch Schaden zufügen können.

Das Vorgehen ist folgendermaßen strukturiert: Nach der Einarbeitung in die Materie Nao - Roboter und Kinect - Kamera soll die Wahl einer geeigneten Programmiersprache getroffen werden. Daraufhin muss eine Anwendung mit zwei wesentlichen Säulen konzeptioniert und implementiert werden. Erstens die Erkennung der Gesten mittels Kinect und zweitens das Empfangen der Gesten und die kinematische Umsetzung durch den Roboter. Zwischen diesen beiden Säulen muss eine geeignete Kommunikationsschicht entworfen und implementiert werden.

Der erste große Meilenstein ist die Übertragung der Armbewegung des Menschen auf den Roboter. Darauf folgend soll versucht werden auch Kopf- und Torsobewegungen zu übertragen. Je nach dem wie der zeitliche Fortschritt nach diesen beiden Meilensteinen ist, kann auch die Steuerung der Füße und somit das Laufen implementiert werden.

1.2 Inhalt

Nach dieser Einleitung in die Studienarbeit, ist dieses Dokument wie folgt strukturiert.

Im nächsten Kapitel werden Grundlagen um das Thema Robotik geschaffen, um die Arbeit in dieses Thema einordnen zu können. Darauf folgend werden die Kinect - Kamera sowie der Nao - Roboter vorgestellt. Dazu gehören bei beiden jeweils eine Vorstellung der jeweiligen Hardware, eine Beschreibung der mitgelieferten Software und als wichtigster Aspekt die Darstellung des Software Development Kit (SDK) beider Systeme. Dazu gehört als zentrale Fragestellung: Wie kann das System durch Programmierung bedient werden?

Ferner wird im Abschnitt Realisierung dargestellt, welches Konzept hinter der Umsetzung der Studienarbeit entwickelt wurde und wie dieses implementiert wurde. Dazu gehören konkret ein Klassen- sowie ein Flussdiagramm der Anwendung und die Erläuterung wichtiger Algorithmen. Ebenso werden die unterschiedlichen Prototypen mit ihren jeweiligen Funktionalitäten vorgestellt.

Zum Schluss wird ein Fazit über die Studienarbeit gezogen und ein Ausblick über weiterführende Anwendungsgebiete solcher Gestenimitationen durch Roboter vorgenommen.

1.3 Umfeld & Strukturelles

Diese Studienarbeit mit dem Thema *Entwicklung einer Gestiksteuerung mittels Kinect für den humanoiden Roboter Nao* wurde während zwei Theoriephasen an der Dualen Hochschule Baden-Württemberg (DHBW) am Standort Karlsruhe von Michael Stahlberger und Lukas Essig durchgeführt. Betreut wurde die Arbeit durch Herrn Prof. Hans-Jörg Haubner, sowie Herrn Michael Schneider.

Dieses Dokument wurde von den beiden Autoren in LaTeX verfasst. Dies hat den Vorteil, dass sehr leicht gleichzeitig an der Ausarbeitung geschrieben werden kann, ohne dass Konflikte dabei auftreten.

Als Programmiersprache wurde C# gewählt, da sowohl das Kinect-Modul als auch der Nao-Roboter über eine Schnittstelle verfügen, die diese objektorientierte Sprache unterstützt. Dies macht es einfacher, die beiden Programme später zu verknüpfen.

Sowohl die LaTeX Dokumentation, als auch der Source-Code sind mit Hilfe von Git[1] in einem Repository versioniert worden. Damit lässt sich genau verfolgen, wer wann eine Änderung durchgeführt hat und zudem lässt sich auch eine vorherige Versionen wiederherstellen. Das Repository wurde auf der Internetplattform *GitHub* gespeichert. Damit ist es möglich, von jedem Rechner auf das Repository und somit die Dateien und Dokumente zuzugreifen. *GitHub* bietet den Vorteil, dass die Zusammenarbeit an Projekten noch zusätzlich durch Management-Funktionen unterstützt werden. In diesem Fall wurden davon hauptsächlich zwei Funktionen benutzt. Einmal die *Issue*- Funktion, mit der Anforderungen, Aufgaben, Bugs etc. eingetragen und einem Bearbeiter zugeordnet werden können. Zusätzlich wurde noch die *Milestone*-Funktion genutzt, mit der Projekt - Meilensteine erstellt werden können. Diese geben an, welche Funktion bzw. Anforderung bis wann erledigt sein sollten. Die Issues sind dann einem Meilenstein zugeordnet und wenn ein Issue geschlossen wird, ist sehr einfach dargestellt, zu wie viel Prozent der Meilenstein erreicht ist.

[1] Software zur Versionsverwaltung von Dateien und Verzeichnissen

Kapitel 2

Grundlagen Robotik

In diesem Abschnitt werden die Grundlagen um das Thema Robotik gelegt. Dabei wird der Begriff Robotik bzw. mobile Roboter definiert und das Thema geschichtlich eingeordnet. Ebenso wird vorgestellt, welche Arten von Robotern es gibt und welche (Hardware) Komponenten zu einem Robotersystem dazugehören. Ferner werden elementare Begriffe wie zum Beispiel Freiheitsgrad erläutert.

2.1 Geschichte

Bereits schon in der Antike, zur Zeit der griechischen Mythologie wurden die ersten Versuche mit Automaten durchgeführt. Die Automaten der ersten Generation hatten wenig mit Robotern im heutigen Sinne zu tun. Jede Aktion musste vom Menschen abgerufen werden, daher wird hier der Begriff *Automat* anstelle von *Roboter* verwendet.

Der erste sagenhafte Konstrukteur von Maschinen und automatenhaften Gebilden war Daidalos. Es wird erzählt, er habe eine Flugmaschine entworfen, die seinen Sohn Ikarus und ihn vor der Ungnade des Königs Minos retten sollte.

Aber auch zu Zeiten der Ägypter wurde die Entwicklung von Automaten vorangetrieben. König Ptolemaios von Ägypten förderte im dritten Jahrhundert v. Christus die Automatenentwicklung als Mäzen[1]. Er war auch bekannt dafür, eigene Automaten zu entwickeln. Bei einem Fest zu Ehren Alexanders des Großen soll er eine Statue gebaut haben, die auf ihrem Weg aufgestanden ist und Milch in eine Schale gegossen hat. Aber auch in anderen Kulturen finden sich Erwähnungen solcher (halb-)automatischer Statuen zu Repräsentationszwecke.

[1]Person oder Institution, die mit Geld oder geldwerten Mitteln bei der Umsetzung eines Vorhabens unterstützt

Doch nicht nur in der Mythologie finden sich hinweise auf die ersten Automaten. Besonders die Anhänger der Schule von Alexandria (Heron, Philon) bewiesen mit ihren Vorrichtungen nach dem Sanduhrprinzip grundlegende technische Entwicklungen. Dabei dienten ihnen Wasser, Wasserdampf, Winde, Hebel und Flaschenzüge als Energiequellen und Antriebsmittel.

Im 14. Jahrhundert wurde die Weiterentwicklung der Automaten durch die Erfindung der „Hemmung“ einer Uhr gefördert. Die Abhängigkeit zum Prinzip der Sanduhr wurde aufgelöst. Der Mensch versuchte unter anderem , „imitatio die“ , ein maschinalisiertes Universum nachzubauen.

In den folgenden Jahrhunderten zielte die technische Entwicklung der Automaten immer mehr auf die perfekte Nachahmung der Natur ab. Dies perfektionierte vor allem Jacques de Vaucansons. 1738 stellte er einen nahezu lebensgroßen Flötenspieler vor, der zwölf Melodien spielen konnte. Diese wurden naturgetreu durch die entsprechenden Zungen- und Fingerbewegungen erzeugt. Der Höhepunkt seiner Entwicklungen ist die mechanische Ente (siehe Abbildung 2.1). Zum ersten Mal imitiert diese nicht nur äußerlich die Natur, sondern auch innerlich korrekt die biologischen Vorgänge. Es ist nicht ganz klar, wie genau die Verdauung funktioniert, sicher ist jedoch, dass die Ente ihre Nahrung auf natürlichem Wege eingenommen und auch wieder ausgeschieden hat.

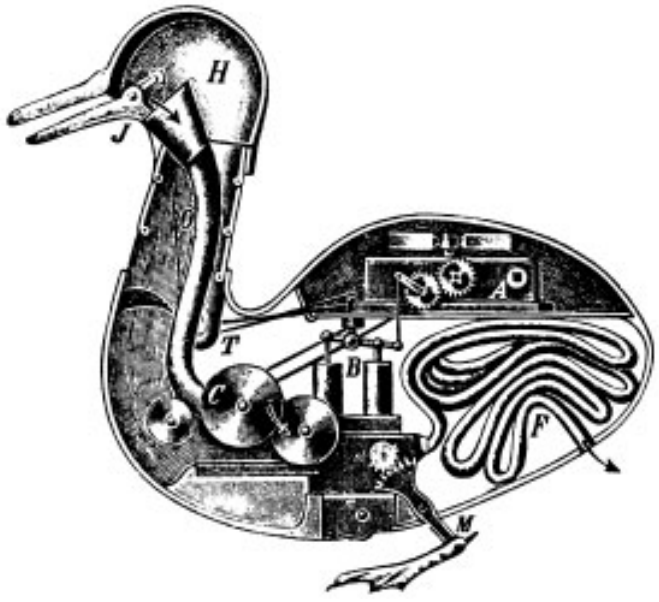

Abbildung 2.1: Vaucansons Ente

Danach wurden tatsächliche Automaten nur noch in sehr kleinem Format gebaut und diese waren auf Grund der hohen Fertigungskosten ausschließlich für wohlhabende Bürger vorbehalten. Deshalb wendeten sich die Techniker der

Entwicklung von Nutzmaschinen zu, die durch die Erfindung von Dampf und Elektrizität als Antriebsmittel deutlich beschleunigt wurden.

Die Idee Automaten zu bauen, die nützlichere Aufgaben erledigen sollten kam Anfang des 20. Jahrhunderts. So begann beispielsweise 1951 die Entwicklung ferngesteuerter Geräte zur Interaktion mit radioaktivem Material. 1954 wurde das erste Patent einer Roboterentwicklung durch den Briten C.W. Kenward eingereicht und 1959 der erste kommerzielle Roboter der Firma Planet Corp. vorgestellt. Dieser wurde mechanisch gesteuert durch Kurvenscheiben und Begrenzungsschalter. Ein erster mobiler Roboter (siehe Bild 2.2) wurde ca. 1968 am Stanford Research Institute entwickelt. Dieser war mit Sensoren, wie Kamera oder Tastsensoren ausgestattet.

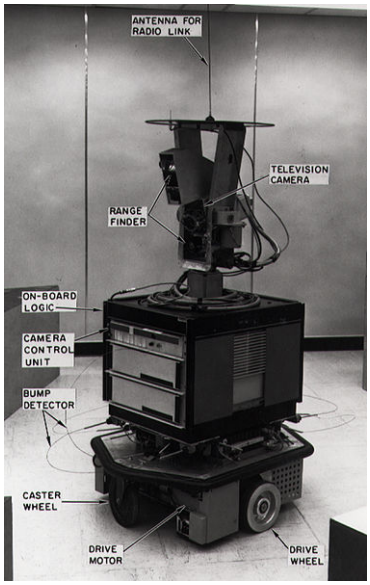

Abbildung 2.2: Mobiler Roboter „Shakey“

1978 wurde der Roboterarm PUMA[2] vorgestellt. Er verfügte über einen elektrischen Antrieb und basierte auf Entwürfen von General Motors.

Zeitlich gesehen wird ab 1970 von Robotern gesprochen, da diese immer mehr mit ihrer Umgebung interagierten und nicht mehr „starr“ ihre Bewegungen ausführten. Der Begriff Roboter oder mobiler Roboter wird im nächsten Abschnitt definiert.

[2]programmable universal machine for assembly, dt.: Programmierbare Universalmaschine für Montage - Anwendungen

2.2 Definition

Derzeit gibt es keine passende Definition, die den Begriff Robotik bzw. mobile Roboter so präzise formuliert, dass sie auf genau die Objekte passt, die nach gemeingültig einen Roboter definieren [**?**]. In [**?**] erklärt der Autor, dass es nicht unüblich ist, dass diese Unschärfe in der Definition großartig stört. Eine allgemeine Definition für Robotik ist in der Verein Deutscher Ingenieure (VDI) Richtlinie 2860 von 1990 zu finden:

„Ein Roboter ist ein frei und wiederprogrammierbarer, multifunktionaler Manipulator mit mindestens drei unabhängigen Achsen, um Materialien, Teile, Werkzeuge oder spezielle Geräte auf programmierten, variablen Bahnen zu bewegen zur Erfüllung der verschiedensten Aufgaben.“

Allerdings passt diese Definition nicht gut auf mobile Roboter. Diese Sprachregelung der VDI zielt eher auf Handhabungsroboter aus der Automatisierungstechnik oder Kommissionierroboter aus dem Logistik - Bereich. Gemeint sind damit sogenannte *stationäre Robotersysteme* [**?**]. Diese sind starr mit der Umgebung verbunden und besitzen einen festgelegten Arbeitsraum. Damit sind die oben genannten "programmierten, variablen Bahnen"möglich, da der Arbeitsprozess in dem sich der Roboter befindet vorher bekannt und programmierbar ist.

Mobile Robotersysteme unterscheiden sich zu stationären grundlegend, indem sie ihre Position durch Lokomotion (aus eigener Kraft) verändern [**?**]. Somit hängen alle ihre Aktionen direkt von ihrer aktuellen Umgebung ab, die detailliert immer erst zum Zeitpunkt der Ausführung bekannt wird [**?**]. Sowohl [**?**] als auch [**?**] heben die Eigenschaft hervor, dass mobile Roboter ihre Umgebung mittels Sensoren erfassen und auswerten müssen. Aus dem Ergebnis der Auswertung wählen sie schließlich ihre nächste Aktion. Ferner werden in [**?**] folgende, weitere Eingeschaften genannt:

- aufgabenorientierte und implizierte Programmierung
- Anpassung an Veränderungen an die Umgebung
- Lernen aus Erfahrungen und entsprechende Verhaltensmodifikation
- Entwicklung eines internen Weltbildes
- Manipulation physikalischer Objekte in der realen Welt

Das bedeutet im Allgemeinen, dass ein mobiler Roboter in einer zuvor nicht bekannten und nicht kontrollierbaren Umgebung zu jeder Situation umgebungsabhängig operieren muss. Im Vergleich zu stationären Robotern bedeutet dass nicht, dass die mobilen Roboter Regellos oder zufällig arbeiten, die entsprechenden Anweisungen sind „weicher“, wie zum Beispiel eine Bahnplanung eines Industrieroboters.

Mit welchen internen und externen Sensoren ein Roboter seine Umgebung verarbeitet, wird in Kapitel 2.3 *Komponenten* erklärt.

2.2.1 Roboterarten

Die **Serviceroboter** sind Maschinen, die den Menschen bei der täglichen Arbeit unterstützen sollen. Darüber hinaus sollen sie auch in der Lage sein als Unterhaltungssystem zu dienen. Es wird geschätzt, dass in ca. 30 Jahren mehr persönlicher Roboter produziert werden als persönliche Computer [**?**]. Diese sollten am Besten auf Zuruf alltägliche Arbeit erledigen. Dazu zählt Staub saugen, Rasen mähen oder auch Fenster putzen. Dabei tritt häufig ein zentrales Problem in den Vordergrund: die Anatomie des Menschen. Diese ist ein regelrechtes Wunderwerk, da beispielsweise die menschliche Hand bei einem durchschnittlichen Gewicht von 600 Gramm und 23 verschiedenen Freiheitsgraden dazu in der Lage ist, Bewegungen mit stufenlos verstellbarer Feinfühligkeit durchzuführen. Vor allem japanische Forscher stecken viel Arbeit in die Erforschung immer besser werdender Serviceroboter. Der durch Honda im Jahr 2001 vorgestellte Serviceroboter *Asimo* ist bei einer Größe von 1.20 Meter nur noch 43kg schwer (siehe Abbildung 2.3). Dabei ist eine der größten Herausforderungen die Stromversorgung. Ein Roboter der ständig am Ladegerät hängen muss, hilft dem Menschen nicht weiter. Daraus resultiert, dass kommerzielle Anbieter von Staubsaugrobotern Räder als Antrieb benutzen. Dies Roboter müssen ihre Bewegungen nicht ständig korrigieren. Sie fahren in konstanter Bodennähe durch die Gegend. Ferner ist die Interaktion zwischen Roboter und Mensch ein Problem. Das Robotersystem muss Daten nicht nur messe und analysieren, sondern auch bewerten können. Dabei gibt es immer noch keine geeignete Verarbeitung der intuitiven Kommunikation, die beim Menschen über Gestik, Mimik und Sprache erfolgt. So wird es noch ein paar Jahre dauern, bis Serviceroboter uns wirklich im Alltag unterstützen können.

Abbildung 2.3: Roboter ASIMO

Industrieroboter hingegen haben sich in der aktuellen Zeit schon sehr etabliert. VW setzt an einen Produktionsstandort ca. 760 Industrieroboter ein und in Deutschland sind es sogar mehr als 100.000 [**?**]. Im Gegensatz zu den Servicerobotern haben sie meist nur einen Arm und besitzen keine Beine und keinen Kopf. Ihre Aufgabe besteht darin selbst tonnenschwere Lasten zu heben und pausenlos gefährliche Arbeit wie z.B. Schweißen durchzuführen. Theoretisch kann jedes Fertigungsproblem heutzutage auch maschinell gelöst werden. In der Autoindustrie gibt es beispielsweise weite Bereiche (Herstellung Karosserie), wo Roboter dominieren. Dafür zeigt sich in der Endmontage, dass Roboter deutlich in der Minderheit sind. Zukünftig werden die Industrieroboter statt umständlich und in komplizierten Sprachen über eine grafische Benutzeroberfläche verfügen und ihre Aufgaben per Sprachsteuerung erklärt bekommen [**?**].

Robotersystem werden auch im Bereich der **Medizin** eingesetzt. Durch ihre Präzision fräsen sie beispielsweise Hüftgelenke so exakt aus, dass die Ersatzteile nahtlos und ohne Zement eingefügt werden können. Die Vorteile sind, dass sie frei von emotionalen Einflüssen sind und keine Tagesform kenne. Egal zu welchem Zeitpunkt, die programmierte Aufgabe erledigt der Roboter immer gleich. Jedoch sind sie teuer und kompliziert. Die Operation muss im Vorfeld genauestens geplant werden und auf sich plötzlich verändernde Gegebenheiten kann ein

Automat nicht reagieren. So musste unter anderem der *Robodoc* abgeschaltet werden, da er Menschen Schaden zugefügt hat. Dadurch hat sich derzeit der Einsatz von Robotern in der Medizin eher auf die Form eines Assistenten reduziert. Die Maschine hält dabei Skalpell und Nadel und führt als verlängerter Arm aus, was der menschliche Operateur hinter dem Steuer vorgibt [**?**]. Solche System werden vor allem bei Herzoperationen eingesetzt, da sie jedes Zittern des Menschen herausfiltern und so an kleinsten Gefäßen und Strukturen arbeiten können. Die Roboter helfen dabei, schwierige Operationen einfacher zu machen.

Humanoide Roboter sind menschenähnliche Roboter wie *Asimo* (Abbildung 2.3)oder Nao. Ihr Bau ist nicht nur durch den Spieltrieb der Robotiker oder der Sensationslust des Publikums motiviert. Auch wissenschaftliche Aspekte sprechen für den Bau humanoider Roboter. So wird nur ein Roboter mit einem menschenähnlichen Körper auch menschenähnliche Begriffe und Denkweisen entwickeln (Johnson 1987). Ferner spielt natürlich auch der psychologische Aspekt im Bereich der Servicerobotik eine Rolle. Wenn etwas wie ein Mensch aussieht, wird damit auch umgegangen wie mit einem Menschen. Einem humanoiden Roboter kann ein Mensch, wie seinem menschlichen Gegenüber, seine Wünsche mitteilen und braucht dafür keine Gebrauchsanweisung. Dazu kommt, dass ihre Körper auch auf die Umgebung abgestimmt sind, in der sich der Mensch bewegt.

2.3 Komponenten

Kapitel 3

Kinect

Im folgenden Kapitel wird der Aufbau und die Funktion des Kinect Sensors erläutert. Zudem wird erläutert, welche Funktionen das Kinect SDK bereits enthält und welche noch erweitert werden mussten, um auf das gewünschte Ergebnis zu gelangen.

3.1 Kinect Hardware

Microsoft brachte im November 2010 mit dem Produkt Xbox Kinect eine neues Produkt auf den Markt. Mit diesem war es möglich, ihr eigenes Produkt Xbox 360 über eine 3D-Kamera zu steuern und Spiele zu spielen. Mit diesem Produkt konnte erstmals ein preiswerter Sensor verwendet werden, um 3D-Bilddaten auszuwerten. Im Folgenden wird auf die Hardware des Sensors näher eingegangen:

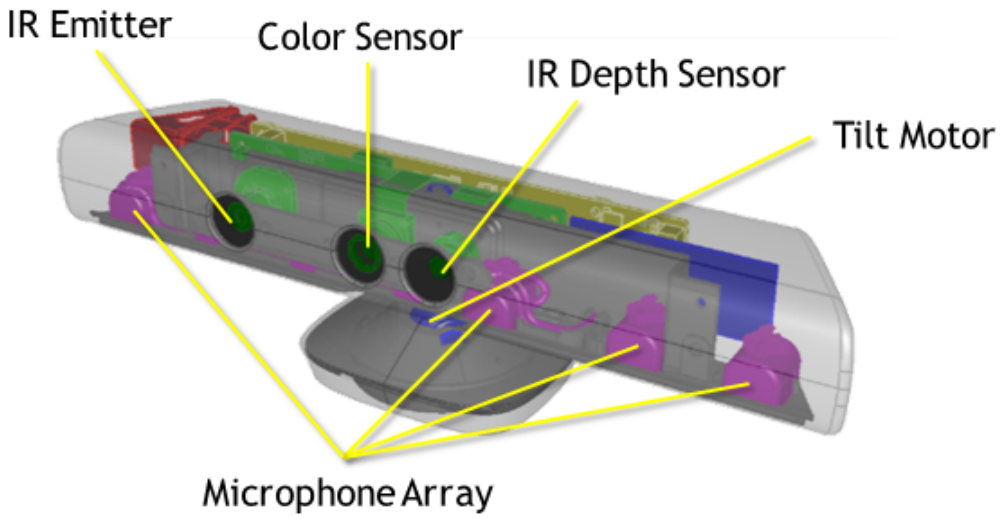

Abbildung 3.1: Aufbau Kinect [**?**]

Das Kinectmodul besteht technisch gesehen aus mehreren Komponenten, die es ermöglichen, die für diese Projektarbeit relevanten, abstrahierten Funktionen (Skelettverfolgung, Winkelextraktion, RGB-Bild) zu realisieren. [**?**]

3.1.1 RGB-Kamera

Ein wesentlicher Teil der Kinect ist die RGB Kamera. Diese liefert ein Bild, das alleine oder auch in Kombinationen weiterer Funktionen verwendet werden kann. Das Sichtfeld der Kamera beträgt 43 Grad in der Vertikalen, sowie 57 Grad in der Horizontalen. Bei einer Auflösung von 640 x 480 Pixel werden 30 FPS und bei 1280 x 960 Pixel noch 12 FPS unterstützt. Wobei standardmäßig die geringere Auflösung genügt und deshalb auch verwendet wird. [**?**]

3.1.2 Tiefenerkennung

Der andere wesentliche Teil der Hardware besteht aus den Komponenten, die für die Tiefenerkennung zuständig sind. Diese Komponenten sind der Infrarot-Emitter (IR-Emitter) und der IR-Tiefensensor (siehe Abb. 3.1).
Zur Abtastung der Tiefenwerte wendet die Kinect das *Light-Coding* Prinzip an. Hierbei wird eine Szene mit einem Muster überlagert und daraus die Tiefeninformation berechnet. Der IR-Emitter sendet dabei ein spezielles Punktmuster aus (Siehe Abb. 3.2), das vom Tiefensensor abgetastet werden kann.

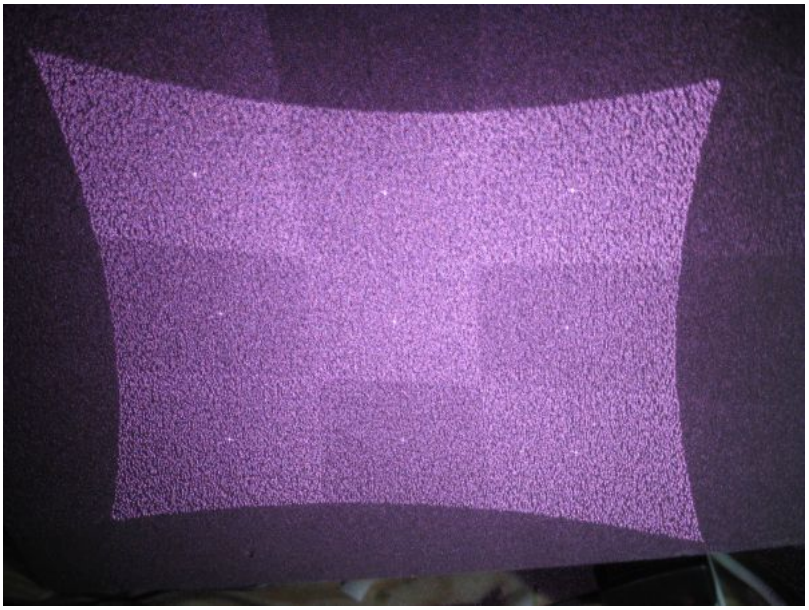

Abbildung 3.2: Kinect Punktemuster

Der IR-Empfänger ist ein Complementary Metal Oxide Semiconductor (CMOS)-Chip und unterstützt eine maximale Auflösung von 640 x 480 Pixel.
Die Einheit von Kinect, die für die Verarbeitung der Tiefen und RGB-Daten verantwortlich ist, ist der PrimeSense-Chip. Dieser wertet das Light-Coding mittels *Depth from Focus* und *Depth from Stereo* aus. [**?**]
Befindet sich nun ein Objekt im Sichtbereich der Kinect, wird das Muster auf das Objekt projiziert und die Form des Objektes verzerrt das Muster (Siehe Abb. 3.3). Daraus kann der Chip die Tiefeninformationen errechnen.

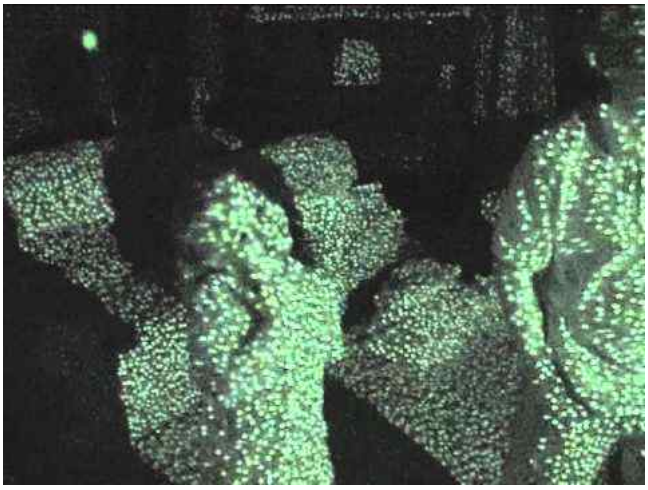

Abbildung 3.3: Kinect Objekte im Sichtraum

Die *Depth from Focus* Methode beruht darauf, dass ein Gegenstand immer verschwommener wird, je weiter weg er sich von der Kinect befindet. Um diesen Effekt optimal zu nutzen, befindet sich vor dem Tiefensensor eine spezielle astigmatische Linse, die diesen Effekt zusätzlich verstärkt und einen Punkt zu einer Ellipse werden lässt, je weiter er von der Linse entfernt ist. [**?**]
Die *Depth from Stereo* Methode beruht auf der Parallaxe. Verändert ein Beobachter seine eigene Position, so beobachtet dieser eine scheinbare Änderung der Position des beobachteten Objektes [**?**]. Hierbei erscheinen nahe liegende Objekte größer verschoben als weiter entfernte Objekte. Das „Verschieben“entsteht durch die Position der zwei Kameras. Anhand dieser Methoden kann der PrimeSense-Chip also die Tiefeninformationen aus der Szene extrahieren.

Der Vorteil dieser Auswertung ist, dass zu einem Zeitpunkt nicht nur ein Punkt, sondern gleich eine komplette Punktewolke ausgerechnet werden kann und somit auch mehrere Bewegungen gleichzeitig. Zum Beispiel eine Arm und eine Beinbewegung gleichzeitig.

3.1.3 Tilt-Motor

Die Kinect besitzt einen Motor, um das Gehäuse in eine passende vertikale Position zu bringen. Der Kopf ist in vertikaler Richtung positiv, sowie negativ um 27 Grad verstellbar. Somit kann man den Sensor optimal an die aktuelle Position anpassen, sodass das Skelett einer Person im Kinectraum noch korrekt erkannt werden kann und sich innerhalb des Sichtfeldes befindet. [**?**] **To do** (1)

3.1.4 Microphon Array

Die Hardware des Sensors enthält zusätzlich noch ein Mikrofonarray, das aus vier Mikrofonen besteht. Diese sind an der Vorderseite linear angeordnet. Drei befinden sich auf der rechten Seite und eines auf der linken Seite. Somit lässt sich die Richtung eines Geräusches identifizieren. Zusätzlich bietet diese Anordnung eine Möglichkeit Störgeräusche (Rauschen) aus einer Aufnahme zu entfernen und entstehende Echos zu Unterdrücken.
Damit kann eine Kinect die Richtung eines Geräusches ermitteln und besser auswerten. Es gibt auch eine Möglichkeit Sprachbefehle mit einer Grammatik aus der Kinect-Library zu erkennen. Diese sind jedoch für dieses Projekt irrelevant und so wird das Mikrofonarray nicht verwendet. [**?**]

3.2 Kinect Software

Die im vorigen Kapitel aufgezeigte Hardware liefert die entsprechenden Sensorinformationen, die nun softwaretechnisch zu einer abstrakteren Ebene zusammengefasst werden müssen. Die Funktion dieser Verarbeitungen speziell für das Microsoft Kinect SDK soll in diesem Kapitel erläutert werden.

Die Hardware liefert folgende Werte:

- RGB Bildsignal der Kamera
- Tiefenwerte
- Audiosignale inkl. Richtungswert des Microphonarrays

Wie bereits erwähnt sind die Audiodaten für dieses Projekt jedoch irrelevant. Wichtig sind primär die Bilddaten inklusive Tiefenwerte. Das Kinect SDK bietet bereits standardmäßig Zugriff auf diese Werte. Dabei können im Programm bestimmte Proxyobjekte registriert und abgefragt werden. Wie diese im Programm letztendlich verwendet werden, wird im folgenden Kapitel 3.3 Kinect SDK näher

erläutert. Dieses Kapitel beschäftigt sich damit, wie diese Funktionen im Inneren funktionieren.

3.2.1 Der Tiefenstream

Der Tiefenstream (Depth-Stream) liefert zu jedem Pixel im Sichtfeld (43 Grad vertikal, 57 Grad horizontal) einen Tiefenwert. Dieser Wert enthält die Entfernung in Millimetern zu dem jeweiligen Objekt. Damit ist das am nächsten befindliche Objekt gemeint, da verdeckte Objekte nicht vom Sensor erkannt werden. Aus diesem Stream kann man beispielsweise ein Tiefenbild mit verschiedenen Grauwerten generieren, die repräsentativ für die Entfernung zur Kinect stehen. Die Depth-Stream Daten werden in einem 16 Bit Array transportiert. Dabei sind 3 untersten Bits jedoch für die Erkennung von mehreren Spielern reserviert (Siehe Kapitel 3.2.2 Der Skeletonstream). Die oberen 13 Bit enthalten die eigentlichen Tiefendaten. Ein solcher Tiefenstream kann wie folgt aussehen:

To do (2)

Zur Erfassung eines Objektes, muss sich dieses im Sichtfeld der Kinect befinden.

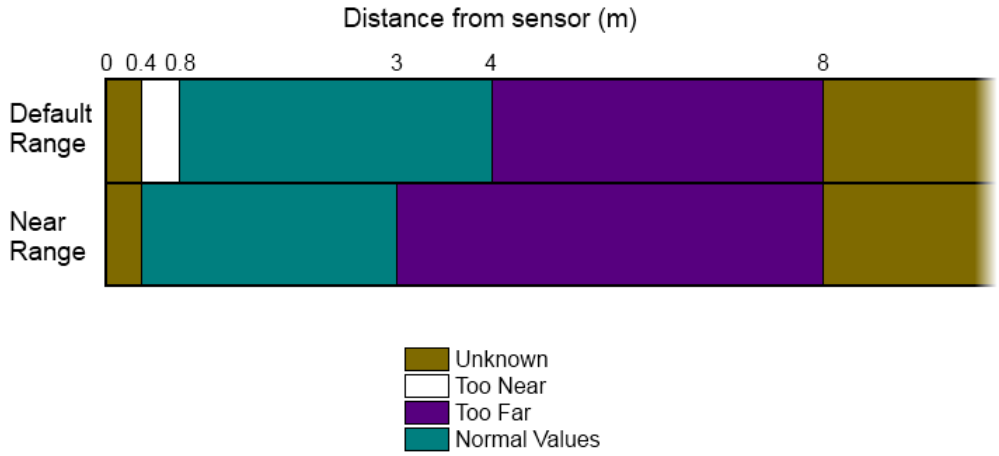

Abbildung 3.4: Depth Space Range [**?**]

Somit liegt der Arbeitsraum der Kinect im normalen Modus zwischen 0.8m und 4m. Der Near-Modus hat einen Arbeitsbereich von 0.4m - 3m. Im Laufe unseren Projektes beschränken wir uns auf den normalen Modus.

Anhand dieses Tiefenstreams ist es nun möglich Spieler (Menschen) zu extrahieren und deren Bewegungen in Echtzeit zu erfassen. Kinect bietet durch dessen SDK bereits eine Möglichkeit, diese Daten zu erfassen und auszuwerten. Da das fertige Objekt durch ein Skelett dargestellt wird, wird dies auch Skeleton-

Objekt oder einfach nur als Skeleton bezeichnet. Für dieses Projekt ist dieses Skeleton-Objekt von großer Bedeutung. Anhand von Vergleichsmustern erzeugt die Software -nicht Kinect!- einen sog. Skeletonstream, der versucht die Position eines menschlichen Skeletts im Kinect Koordinatenraum abzubilden. [**?**] Wie genau dies funktioniert wird im folgenden Kapitel erläutert.

3.2.2 Der Skeletonstream

Damit ein Benutzer und dessen Bewegungen mittels Kinect erkannt und im Programm verarbeitet werden können, wird der Tiefenstream entsprechend ausgewertet. Als Ergebnis erhält man ein menschliches Skelett, das aus 20 Gelenken (Joints) besteht. Diese Joints sind im Skeletonobjekt hierarchisch von der Hüfte aus miteinander verbunden und bilden somit das Skelett eines Spielers. Die folgende Grafik illustriert alle erkannten Gelenke und deren Positionen. Für dieses Projekt sind insbesondere die Joints textitShoulder, *Elbow*, *Wrist* und *Hip* von hoher Bedeutung, da mit ihnen die Winkel der Gliedmaßen errechnet werden können (Siehe Kapitel 5.2 Kinect - Armerkennungsalgorithmus).

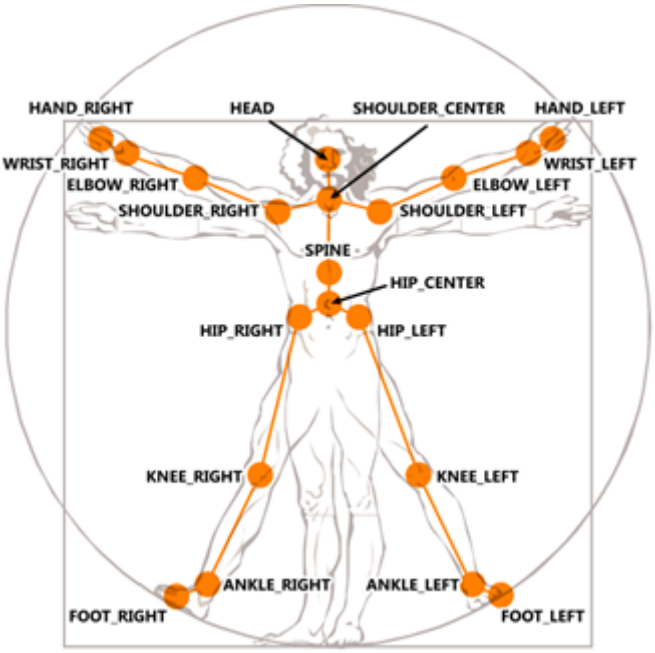

Abbildung 3.5: Kinect Joints [**?**]

Ein Kinectsystem kann bis zu 6 Spieler gleichzeitig erkennen, wobei nur von zwei Spielern die exakte Position aller Gelenkwinkel erkannt werden kann. Von den anderen vier Spielern wird nur die Position im Raum ermittelt, nicht jedoch die exakten Koordinaten aller 20 Joints. Wie schon im vorigen Kapitel erwähnt,

wird einem Spieler nach dessen Erkennung ein Index zugeordnet. Dieser befindet sich im Tiefenarray, falls die Skeletonfunktion aktiviert wurde. Innerhalb dieses Projektes ist dieser jedoch irrelevant, da angenommen wird, dass immer nur ein Spieler den Roboter steuern wird.

Skelettextraktion

Hinter der Extraktion des Skeletts steckt ein Algorithmus, der bereits im *Microsoft Kinect SDK* vorhanden ist. Die Anforderungen an diesen Algorithmus sind hoch, denn er soll Menschen mit verschiedenen Größen, Breiten und Kleidern in Echtzeit erkennen. Damit dies überhaupt möglich ist, wird maschinelles Lernen verwendet. Zunächst werden die aufgenommenen Tiefendaten in eine sog. *Rendering Pipeline* [**?**] gegeben. Diese vergleicht die aufgenommenen Daten mit bereits vorhandenen Beispielen, die mit verschiedenen Menschen aufgenommen wurden. Anhand eines Decition Trees, der durch maschinelles Lernen erzeugt wurde, wird ein Pixel mit einer gewissen Wahrscheinlichkeit zu einem Körpersegment zugeordnet. Der Algorithmus arbeitet zur Erhöhung der Genauigkeit mit mehreren Bäumen gleichzeitig. Somit können die Körpersegmente zugeordnet werden, was in der folgenden Grafik farbig dargestellt wird.

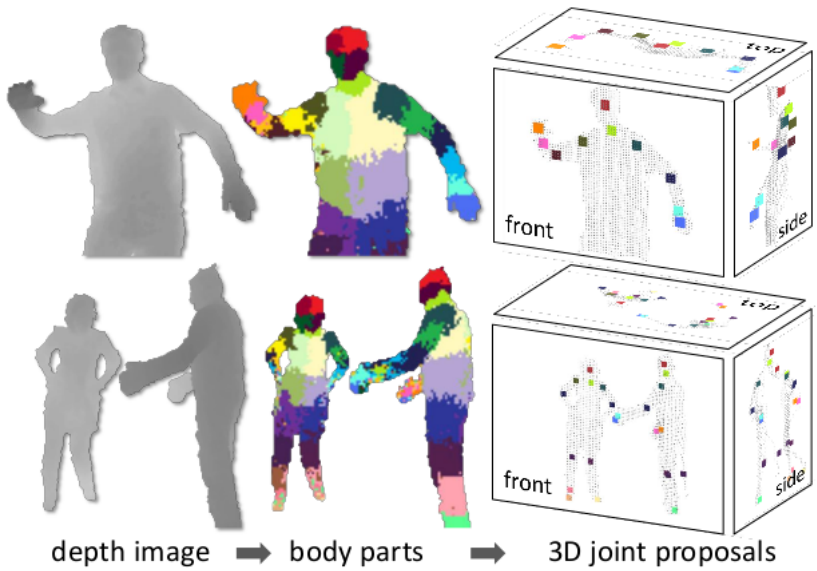

Abbildung 3.6: Skelett-Erkennung [**?**]

Nach der Segmentierung werden die Gelenke innerhalb den entsprechenden Bereichen platziert. Dies wird anhand einer Häufigkeitsanalyse von Beispieldaten errechnet. Im Anschluss daran wird ein 3D-Modell des Skeletts erstellt, das aus der *Frontansicht*, der *Seitenansicht* und der *Draufsicht* besteht (Siehe Abb. 3.6

rechts: 3D joint proposals). Dieser komplexe Prozess ermöglicht es eine Skelett-Erkennung in Echtzeit durchzuführen. Als Entwickler einer Kinect Applikation muss man sich nicht mehr um diese Erkennung kümmern, denn diese ist bereits implementiert. Die Koordinaten der Skeleton-Joints befinden sich im Skeletonobjekt und können mit der erkannten Genauigkeit ausgelesen und verwendet werden.

Skelett-Koordinatenraum

Der Koordinatenraum der Kinect ist wie folgt definiert:

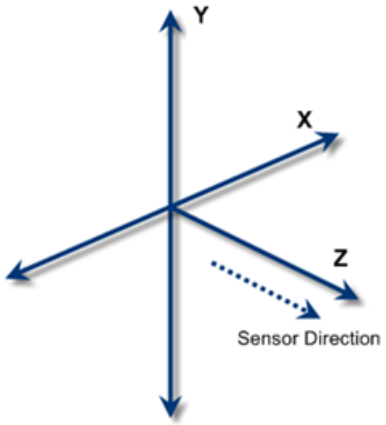

Abbildung 3.7: Skelett-Koordinatenraum [**?**]

Das Kinect-Koordinatensystem ist ein rechtshändiges Koordinatensystem. Die positive Z-Richtung entspricht der Blickrichtung der Kinect. Die positive X-Richtung zeigt nach links und entsprechend die positive Y-Richtung nach oben. [**?**]

To do (3)

3.3 Kinect SDK

3.3.1 Timeline

Nach dem Erscheinen der Kinect-Hardware war diese auch schnell in Entwicklerkreisen gefragt. Microsoft selbst gefiel dies zunächst nicht, denn der Konzern befürchtete, dass Cheater sich an ihren Spielen zu schaffen machen würden und somit den Spielspaß mindern würden. So veröffentlichte Microsoft selbst zunächst kein SDK. Die Open Source Gemeinde jedoch erkannte das Potential des Produktes schnell und entwickelte eine Schnittstelle zu *OpenNI*, einem Framework, das die Auswertung von 3D-Sensordaten verschiedenster Hersteller unterstützt. Dieses Framework bietet somit durch seine Plattformunabhängigkeit die Möglichkeit

unterschiedliche Betriebssysteme mit unterschiedlichen 3D-Sensoren zu kombinieren. [?] Microsoft zog nach und veröffentlichte am 17. Juni 2011 die freie Beta Version des Microsoft SDKs. Somit hatte nun jeder Entwickler freien Zugang zu allen Kinectfunktionen, die auch von Microsoft selbst bisher genutzt wurden. Einer der Vorteile des Kinect SDKs besteht darin, dass die Skelett-Erkennung (Siehe Kapitel 3.2 Kinect Software) ohne initiale Pose möglich ist, was im Gegensatz zum OpenNI-Framework steht. [?] Da für dieses Projekt die Plattformunabhängigkeit nicht relevant ist, sowohl aber die Skeletterkennung, fiel die Entscheidung auf das Microsoft Kinect SDK.

3.3.2 Grundfunktionen

Anhand des folgenden Listings sollen die grundlegenden Funktion der vorhandenen Bibliotheken und deren Verwendungsweisen verdeutlicht werden:

```
1  using Microsoft.Kinect;
2
3
4  static void Main(string[] args)
5  {
6  // instantiate the sensor instance
7  KinectSensor sensor = KinectSensor.KinectSensors[0];
8
9  // initialize the cameras
10 sensor.DepthStream.Enable();
11 sensor.DepthFrameReady += sensor_DepthFrameReady;
12
13 // make it look like The Matrix
14 Console.ForegroundColor = ConsoleColor.Green;
15
16 // start the data streaming
17 sensor.Start();
18 while (Console.ReadKey().Key !=
      ConsoleKey.Spacebar) { }
19 }
20
21 static void sensor_DepthFrameReady(object sender,
      DepthImageFrameReadyEventArgs e)
22 {
```

```
23      using (var depthFrame = e.OpenDepthImageFrame())
24      {
25          if (depthFrame == null)
26          return;
27
28          short[] bits = new
                short[depthFrame.PixelDataLength];
29          depthFrame.CopyPixelDataTo(bits);
30          foreach (var bit in bits)
31              Console.Write(bit);
32      }
33  }
```

Listing 3.1: Initialisierung des Kinect Sensors [?]

Zuerst muss die Kinect-Bibliothek in C# eingebunden werden (Zeile 1). Danach kann ein Sensorobjekt instanziiert werden. Dieses erreicht man über ein Array, da gleichzeitig mehrere Kinect-Systeme per USB angeschlossen und verwendet werden könnten (Zeile 7). Für dieses Projekt wird jedoch nur eine Hardware benötigt und somit handelt es sich immer um das Element mit dem Index 0. Im Anschluss daran wird der Tiefenstream des Sensors aktiviert (Zeile 10) und ein *DepthFrameReady* Event registriert und mit der Methode sensor_DepthFrameReady (Ab Zeile 21) verknüpft. Diese Methode wird somit bei jeder Aktualisierung der Tiefenwerte aufgerufen und liefert die aktuellen Werte, die in einem Short-Array gespeichert werden.

Anmerkung
In diesem Listing fehlt die Freigabe von Kinect nach Terminierung der Applikation. Aus diesem Grund registiert man einen WindowClosing-Listener, der dann die jeweiligen aktivierten Funktionen richtig deaktiviert. Dies ist nicht zwingen nötig, aber wird das Programm beendet, bleibt der IR-Emitter aktiviert und es kann zu Komplikationen führen, wenn das Programm erneut auf die Funktionen zugreifen möchte.

3.3.3 Skelettfunktionen

Zur Nutzung des Skelett-Trackings muss eine Anwendung ähnlich dem obigen Listing initialisiert werden. In diesem Fall aktiviert man den Skeletonstream mit *sensor.SkeletonStream.Enable()* und registriert wie oben auch ein Event. In die-

sem Fall ein *SkeletonFrameReady*-Event und verknüpft dieses wieder mit einer Methode. Folgender Codeausschnitt zeigt die Verwendung des Skeletonstream innerhalb dieses Projektes in C#.

```
1  using (SkeletonFrame sFrame = e.OpenSkeletonFrame())
2  {
3      if (sFrame  == null)
4      {
5           // The image processing took too long.
               More than 2 frames behind.
6      return;
7      }
8      else
9      {
10       skeletons = new
            Skeleton[sFrame.SkeletonArrayLength];
11                 sFrame.CopySkeletonDataTo(skeletons);
12                 receivedData = true;
13     }
14     if (receivedData)
15     {
16       Skeleton currentSkeleton =
17        ( from s in skeletons
18                     where s.TrackingState ==
                          SkeletonTrackingState.Tracked
19                     select s).FirstOrDefault();
20
21    if (currentSkeleton != null)
22         {
23            //Handler mit neuen Skelettdaten
                 versorgen
24            anglehandler.updateSkeleton(currentSkeleton);
25         }
26     }
27 }
```

Listing 3.2: Verwendung des Skeleton-Streams

Erkennt Kinect einen Spieler im Sichtfeld, wird ein SkeletonFrameReady-Event erzeugt und die registrierte Methode aufgerufen. Da in diesem Projekt der RGB-Stream und der Skeleton-Stream gleichzeitig benötigt werden, gibt es auch die Möglichkeit ein *AllFramesReady*-Event zu registrieren, dass Informationen aller Streams beinhaltet. Im obigen Listing ist ein Auszug aus der Methode *runtime_AllFramesReady*. Diese erhält über die Methode *OpenSkeletonFrame* das Skelettobjekt. Danach kann das Objekt entsprechend verwendet werden. Falls ein neues Skelett vorhanden ist, wird dieses in einem Array vom Typ **Skeleton** gespeichert (Zeile 10). Da innerhalb dieses Projektes jedoch nur ein Benutzer relevant ist, wird zur Weiterverarbeitung nur das aktive Skelett des ersten Spielers verwendet. Existiert ein neues Skelett, wird dieses an den Handler übergeben, der die Schnittstelle zur weiteren Verarbeitung des Skeletts darstellt (Zeile 24). Auf die genaue Funktion dieses Handlers wird im Kapitel 5.1 Architektur näher eingegangen. **To do** (4)

3.3.4 Kinect Studio

Das Kinect-SDK stellt einem Entwickler nach der Installation nützliche Dinge bereit, wie eine Bibliothek mit verschiedenen Templates, die beschreiben wie die Funktionen der Kinect aus Microsoft Sicht verwendet werden. Zusätzlich zu den Templates wird mit dem SDK auch das Kinect Studio mitgeliefert. Diese Anwendung ist für die Entwicklung von Kinect Applikationen sehr nützlich, da man hier wie bei einem Videoschnittprogramm Sequenzen aufnehmen kann, die RGB und Tiefendaten enthalten. Zudem erkennt Kienct Studio aktive Kinect-Applikationen und kann sich zu diesen verbinden. Somit können Bewegungen aufgenommen werden und in der entwickelten Anwendung *abgespielt* werden, ohne diese immer wieder vor der Kinect ausführen zu müssen. **To do** (5)

Anmerkung
Das Verbinden funktioniert jedoch nur, wenn Kinect per USB aktuell angeschlossen ist. Deshalb benötigt man zwingend ein Kinect-Modul, um seine Anwendung testen zu können.

Kapitel 4

Nao

Nao ist ein humanoider Roboter des französischen Herstellers *Aldebaran Robotic*. Die erste Version des Roboters wurde 2006 vorgestellt und unter anderem mit einem Investment durch Intel weiterentwickelt. Es gibt den Roboter in verschiedenen Ausführungen mit verschieden verbauten Sensoren und kostet ungefähr 10.000 Euro. [**?**]

Im folgenden Kapitel wird der Aufbau und die Funktionen der Hardware, sowie die zugehörige Software zur Bedienung und Programmierung des Nao vorgestellt.

4.1 Nao Hardware

Der in dieser Arbeit verwendete Nao ist vom Typ *NAO V3+, V3.2*. Dieser Typ ist 573.2 mm hoch, 290 mm tief und 273.3 mm breit. In Abbildung 4.1 *Übersicht Nao V3.2* sind alle Sensoren und Aktoren aufgeführt.

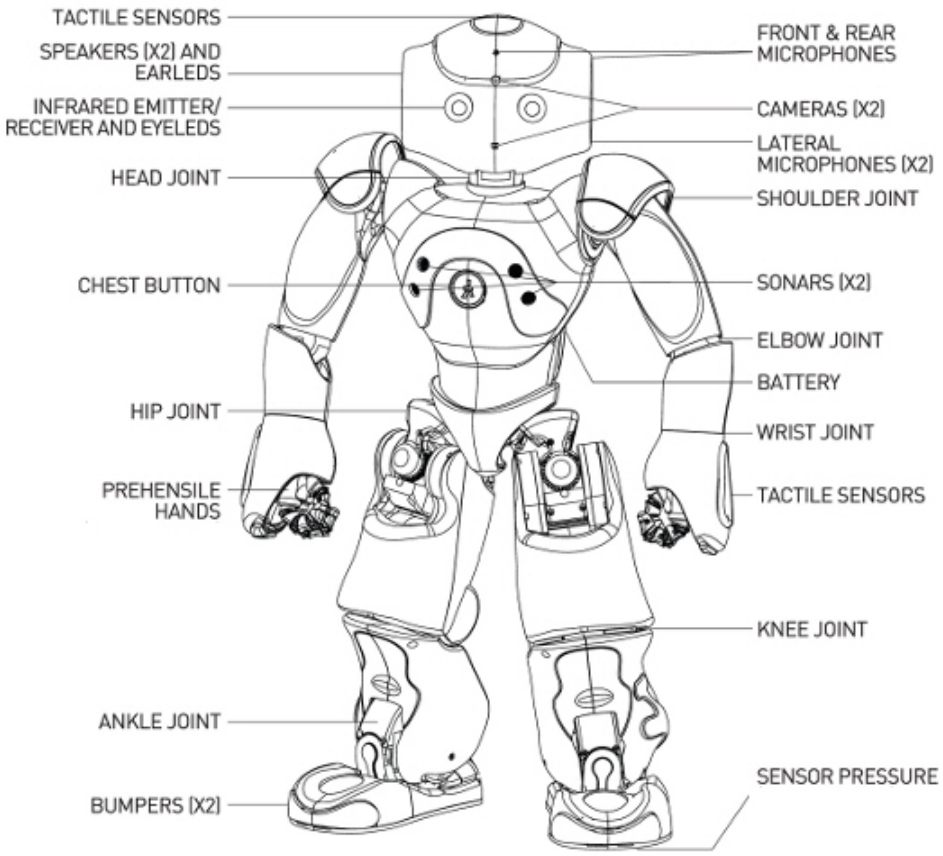

Abbildung 4.1: Übersicht Nao V3.2

Gelenke

Nao besitzt im Kopf, den beiden Armen, dem Becken und den beiden Beinen jeweils mehrere Gelenke (*Joint*, siehe Bild 4.1). Damit ist eine umfangreiche Bewegung in alle Richtungen der drei Achsen möglich. Der Kopf lässt sich in Z-Richtung drehen und in Y-Richtung neigen, damit Nao auch räumlich sehen bzw. Objekte verfolgen kann. Die Arme besitzen die gleichen Gelenke wie in einem menschlichen Körper. Dazu gehören Schulter-, Ellenbogen- und Hangelenk. Das Schultergelenk dient dazu, den Arm zu heben/senken und ihn zu öffnen bzw. zu schließen. Das Drehen und Öffnen/Schließen des Unterarms geschieht durch das Ellenbogengelenk in Kombination mit dem Hangelenk. Die Finger der Hand können nur als Ganzes geöffnet bzw. geschlossen werden. Das Beckengelenk wird dazu benutzt, den Torso von Nao nach vorne oder hinten zu neigen. Die Beine bestehen aus drei Gelenken: Einem Hüft-, einem Knie- und einem Fußgelenk. Die Bewegungsfreiheit der Beine ähnelt dem des menschlichen Beins, wobei keines der Gelenke in Z-Richtung gedreht werden kann.

Die Bestimmung der Gelenkpositionen erfolgt über einen magnetischen Drehwinkelgeber mit einer Auflösung von 12 Bit. Das macht beispielsweise bei einem Wert von 4096 pro Umdrehung eine Präzision von 0.1 Grad.

Gelenkraum Nao

Da primär die Arme durch Nao nachgeahmt werden sollen, wird hier kurz auf den Gelenkraum dieser eingegangen. Jedes einzelne Gelenk der beiden Arme besitzt einen Winkelbereich in dem diese bewegt werden können. Tabellen 4.1 und 4.2 zeigen den Gelenkraum für den linken bzw. den rechten Arm.

Gelenk	**Bereich (Grad)**	**Bereich (Radian)**
LShoulderPitch	-119.5 to 119.5	-2.0857 to 2.0857
LShoulderRoll	-18 to 76	-0.3142 to 1.3265
LElbowYaw	-119.5 to 119.5	-2.0857 to 2.0857
LElbowRoll	-88.5 to -2	-1.5446 to -0.0349
LWristYaw	-104.5 to 104.5	-1.8238 to 1.8238

Tabelle 4.1: Gelenkraum linker Arm

Gelenk	**Bereich (Grad)**	**Bereich (Radian)**
RShoulderPitch	-119.5 to 119.5	-2.0857 to 2.0857
RShoulderRoll	-76 to 18	-1.3265 to 0.3142
RElbowYaw	-119.5 to 119.5	-2.0857 to 2.0857
RElbowRoll	2 to 88.5	0.0349 to 1.5446
RWristYaw	-104.5 to 104.5	-1.8238 to 1.8238

Tabelle 4.2: Gelenkraum rechter Arm

Bilder der einzelnen Gelenke und Winkelbereiche sind im Anhang zu finden.

Aktoren

In Nao sind vier verschiedene Typen von Motoren verbaut. Diese unterscheiden sich im wesentlichen in ihrer maximalen Anzahl an Drehungen pro Minute, dem Drehmoment und der Drehzahlrückstellung. Dies ist wichtig, da nicht jedes Gelenk und der zugehörige Aktor mit der gleichen Masse belastet wird.

Elektronik & Sensoren

Das Herz von Nao ist dessen Motherboard mit einer x86 AMD CPU mit 500MHz. Der Arbeitsspeicher mit 256MB RAM und die 2GB Flash-Speicher befinden sich zusammen mit dem Prozessor im Kopf. Die Batterie mit rund 30Wh hält für die aktive Nutzung (viele Bewegungen und Sensoraktivitäten) ca. 60min und die normale Nutzung ca. 90min.

Links und Rechts am Kopf befinden sich jeweils ein Lautsprecher und ein Mikrofon. Zusätliche Mikrofone sind am Kopf auch noch vorne und hinten angebracht. Damit ist es Nao möglich, ein Geräusch zu lokalisiern und gegebenenfalls dahin zu folgen. Um gleichzeitig die Ferne und die Nähe visuell zu verarbeiten wurde über und unter den Augen jeweils eine VGA - Kamera mit einer Auflösung von 640x480 Pixeln installiert. Die Augen selbst dienen zur Erkennung von Infrarotlicht, wobei auch hier in jedem Auge jeweils ein Sensor verbaut ist.

Auf der Brust von Nao befinden sich Ultraschallsensoren zur Distanzermittlung (je 2 Emitter und Empfänger). Diese haben eine Auflösung von 1cm und eine Erkennungsweite von 0.25m bis 2.55m. Unter 0.25m erkennt Nao nur noch, dass ein Objekt im Weg ist, aber nicht wie weit es entfernt ist.

Sensoren zur Kontakterkennung befinden sich auf dem Kopf, dem Brustbutton, auf und neben den Händen, sowie vorne an den Füßen. Unter den Füßen befinden sich zudem noch piezoresistive Drucksensoren mit einem Arbeitsbereich von 0 bis 25 Newton. Damit lässt sich unter anderem erkennen, ob Nao nur auf einem Bein oder auf unebenen Untergrund steht.

To do (6)

4.2 Nao Software

Um Nao auf einfache Weise zu programmieren, simulieren oder ihm neue Funktionen beizubringen gibt es im wesentlichen zwei Programme. Diese werden im folgenden jeweils kurz vorgestellt:

Choreographe

Choreographe ist eine multi - Plattform Desktopanwendung. Mit ihrer Hilfe ist es möglich:

- Neue Animationen und Verhalten zu erstellen,
- diese entweder auf einem simulierten oder direkt auf einem realen Roboter zu testen und
- den Roboter dabei zu überwachen und zu steuern.

Dabei steht die Einfachheit der Anwendung im Vordergrund und so ist es auch möglich sehr komplexe Verhalten (z.B. Interaktion mit Menschen, Tanzen oder E-Mails verschicken) zu implementieren ohne eine einzige Zeile Quellcode selbst zu schreiben. Zusätzlich ist die Möglichkeit gegeben, vorhandenen Code mit eigenem Python - Code zu erweitern.

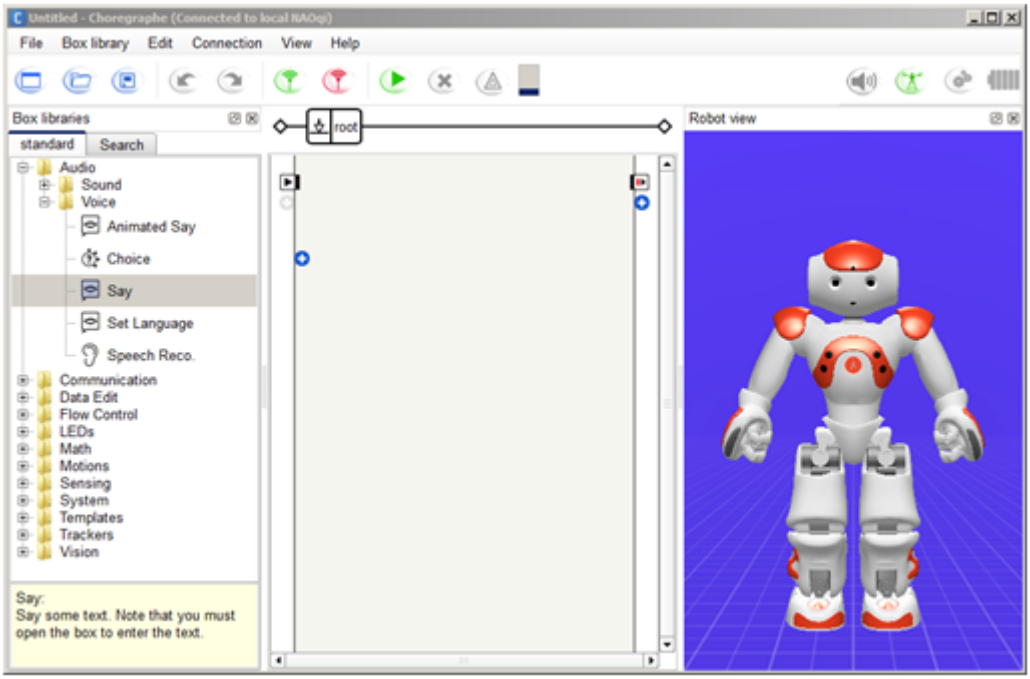

Abbildung 4.2: Überblick Choreographe

Abbildung 4.2 *Überblick Choreographe* zeigt einen Überblick über alle Subfenster und Panels innerhalb Choreographe. Wie zu sehen ist Choreographe zentral in drei Bereiche unterteilt: Links, Mitte und Rechts. Auf der linken Seite ist die sogenannte Box - Bibliothek zu finden. Dort sind alle von Haus aus gespeicherten Bewegungen und Verhalten abrufbar. Diese können per Drag&Drop in die Mitte gezogen werden. Die Mitte stellt sozusagen ein Fluss - Diagramm der einzelnen Boxen dar. So ist es möglich *graphisch* zu programmieren (Verknüpfen der Boxen). Auf der rechten Seite ist das Abbild eines Nao - Roboters zu sehen. Dies zeigt, je nach dem, einen simulierten Roboter oder die Spiegelung eines realen Naos. Durch diese graphisch einfache Programmierung können auch unerfahrene Anwender mit Nao arbeiten. So ist Nao nicht nur für die Forschung oder für Entwickler gedacht, sondern auch für den Unterricht an Schulen.

Die Programmierung des Roboters geschieht durch Verbinden von einzelnen Boxen. Es ist auch möglich ein Programm mit verschiedenen Wegen zu entwerfen oder Konditionalstrukturen (*if-else-elseif*) einzubauen. Abbildung 4.3 zeigt ein Programm, in welchem der Roboter zu einer gewissen Position laufen soll. Währenddessen überprüft er mittels des Ultraschallsensors, ob ein Hindernis im Weg ist. Bei positivem Ergebnis soll sich der Roboter hinsetzen.

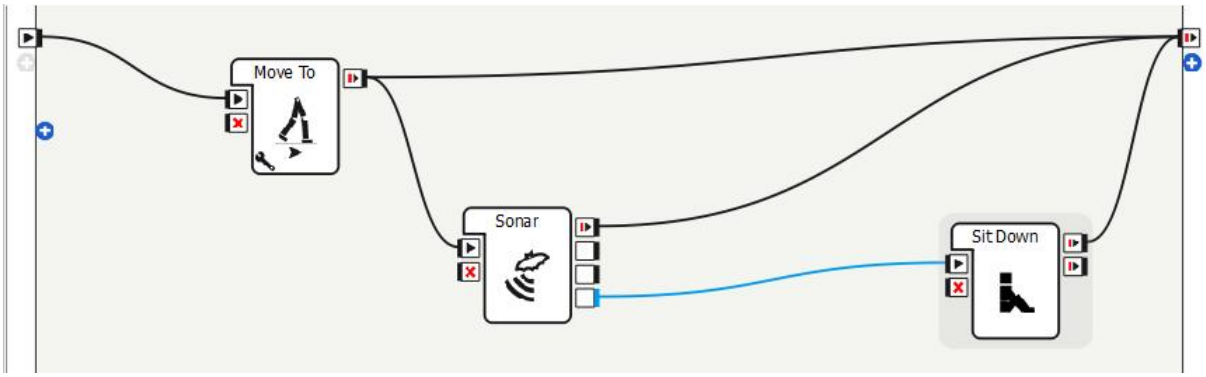

Abbildung 4.3: Choreographe Programmierung

Neue Bewegungen oder Verhalten können entweder durch eigenen Python - Code oder durch vormachen integriert werden. Dazu kann über eine *Timeline* aufgenommen werden, zu welchem Zeitpunkt der Bewegung sich die einzelnen Gelenke/Körperteile befinden sollen. Anschließend kann diese aufgenommene Timeline als Verhalten in einer Box gespeichert werden.

Die Möglichkeit neue Programme erst an einer Simulation zu testen, spart erstens Akku eines realen Nao und zweitens schützt es einen realen Nao vor *schlechten* Programmen, bei denen er Schaden nehmen könnte. Ein weiterer Vorteil ist, dass zu jeder Box der Quellcode in Python sichtbar ist und sich dadurch Arbeit erspart werden kann.

Choreographe wurde in dieser Arbeit hauptsächlich dazu genutzt, sich in die Marterie Nao einzuarbeiten, seine Funktionsweise zu verstehen und zu erlernen wie er programmiert wird.

Webots for Nao

Webots für Nao erlaubt es, einen simulierten Roboter in einer virtuellen Welt zu bewegen. Die Software bietet eine sichere Umgebung um neue Verhalten zu testen, bevor sie in die reale Welt übertragen werden. Webots for Nao ist eine spezifische Auskopplung von Webots 7, einer professionellen Software zum simulieren diverser Roboter, beispielsweise KUKA - Robotern oder Lego Mindstorms. Mit Webots for Nao können jedoch keine anderen Roboter genutzt oder neue erstellt werden.

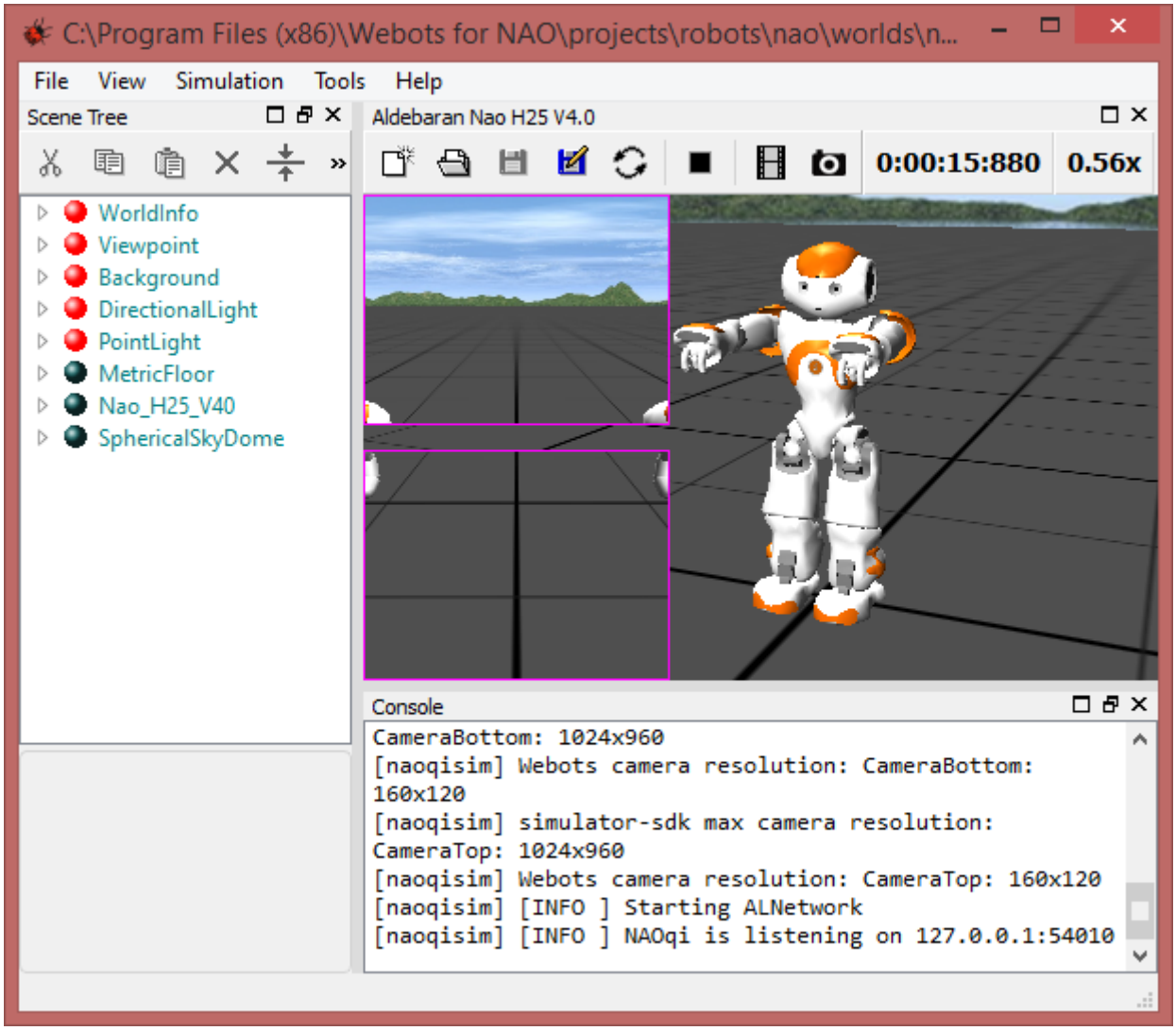

Abbildung 4.4: Überblick Webots

Bild 4.4 *Überblick Webots* zeigt einen simulierten Nao in einer virtuellen Welt. Auf der linken Seite befinden sich Reiter, die ausgeklappt werden können. Dort können Objekte in die Welt gelegt werden. Auf der rechten Seite ist Nao von Vorne und jeweils ein Ausschnitt der beiden Kameras an seinem Kopf zu sehen. Unterhalb davon ist eine Konsole, die verschiedene Angaben ausgibt, z.B. die IP - Adresse und der Port, unter dem der simulierte Nao angesprochen werden kann.

Der Unterschied zu der Simulation in Choreographe liegt darin, dass in Webots auch Elemente wie Tische, Stühle oder andere Hindernisse in die Welt gelegt werden können. So kann beispielsweise getestet werden, ob Nao Hindernissen ausweicht, wenn er auf sie zu läuft.

Webots wurde im Allgemeinen dafür genutzt, zu Testen, ob die Übertragung

der Armwinkel korrekt ist und ob diese der Bewegung durch dem Menschen entsprechen. Würden die Tests an einem realen Nao durchgeführt werden, würden seine Aktoren sehr schnell warm werden und eventuell Schaden davon nehmen.

4.3 NAOqi Framework

NAOqi ist der Name der Software die tatsächlich auf dem Roboter läuft und ihn kontrolliert. Das NAOqi Framework ist das Gerüst um Nao zu programmieren. Es spricht auf die gewöhnlichen Anforderungen in der Robotertechnik an: Parallelität (von Threads), Ressourcen, Synchronisation und Events. Das bedeutet, es kann mit allen gängigen Techniken der Software - Entwicklung bedient werden.

Dieses Framework erlaubt homogene Kommunikation zwischen verschiedenen Modulen (Bewegung, Autio, Video), homogene Programmierung und homogenes Teilen von Informationen über die verschiedenen Module hinweg.

4.5 *NAOqi Übersicht* zeigt die einzelnen Komponenten des NAOqi Frameworks:

- Cross - Plattform
- Cross - Language
- NAOqi - Prozess
- Module

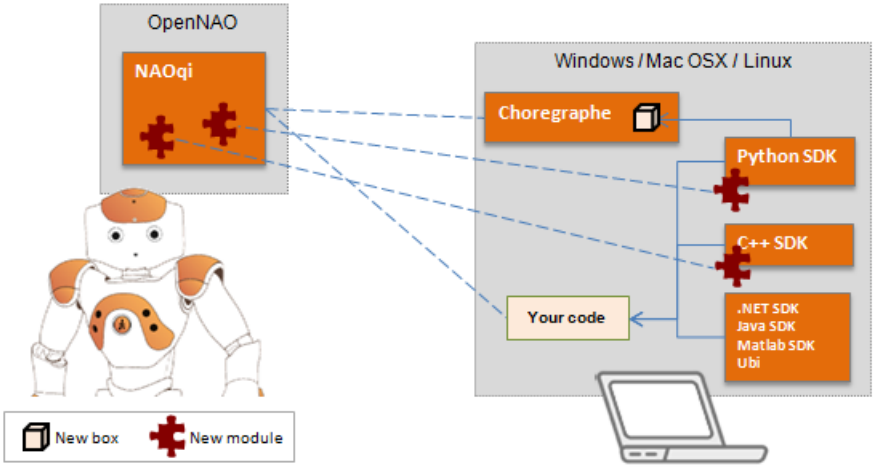

Abbildung 4.5: NAOqi Übersicht

4.3.1 Cross - Plattform/Language

Cross - Plattform

Cross - Plattform bedeutet Plattformunabhängigkeit gegenüber dem Betriebssystem auf dem Programmiert werden soll. Sowohl auf Linux, Windows und auf Mac kann Code für Nao programmiert werden. Allerdings kann auf Windows und Mac nur Code auf dem Computer selbst kompiliert werden, während auf Linux der Code auch auf dem Roboter selbst programmiert werden kann.

Cross - Language

Cross - Language ist nach [**?**] die Eigenschaft, dass Software in C++ und in Python entwickelt werden kann. In allen Fällen, in denen die Methoden exakt gleich sind kann die application programming interface (API) (dt: Programmierschnittstelle), gleichgültig von welcher der unterstützten Programmiersprachen, aufgerufen werden. Die API ist in acht Programmiersprachen verfügbar: C++, Python, .NET (C#, Visual Basic, F#), Java, Matlab und Urbi.

Neue NAOqi Module können nur in C++ oder Python entwickelt werden, jedoch kann die Client - API mit allen Programmiersprachen angesprochen werden. Ebenso sind nur C++ und Python auf dem Roboter unterstützt, die anderen Sprachen werden nur über *Remote - Access* unterstützt. (siehe unten *Proxy*)

4.3.2 NAOqi - Prozess

Der NAOqi - Prozess der auf dem Roboter läuft ist ein *Broker* (siehe unten). Beim Start des Prozesses wird eine Konfigurationsdatei autoload.ini geladen, die definiert, welche Bibliotheken geladen werden sollen. Jede Bibliothek beinhaltet ein oder mehrere Module, die der Broker nutzt um deren Methoden öffentlich anzuzeigen. (siehe 4.6)

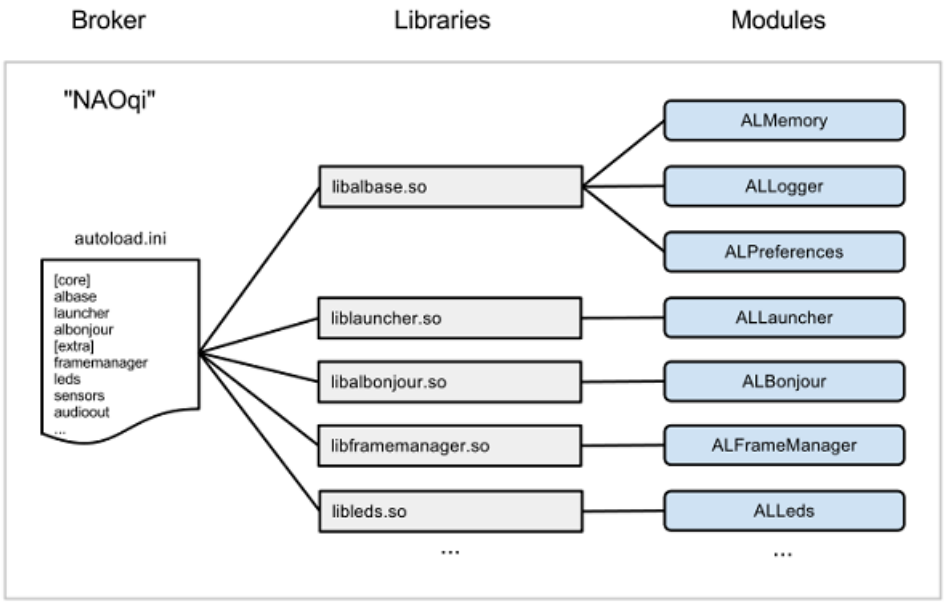

Abbildung 4.6: NAOqi Broker

Der Broker stellt einen Lookup - Service zu Verfügung, so dass jedes Modul im Baum oder verteilt im Netzwerk jede Methode finden kann, die öffentlich angezeigt wurde.

Das Laden der Module zum Start erzeugt einen Baum von Methoden, die an Module geknüpft und diese wiederum an einen Broker geknüpft sind. (siehe 4.7)

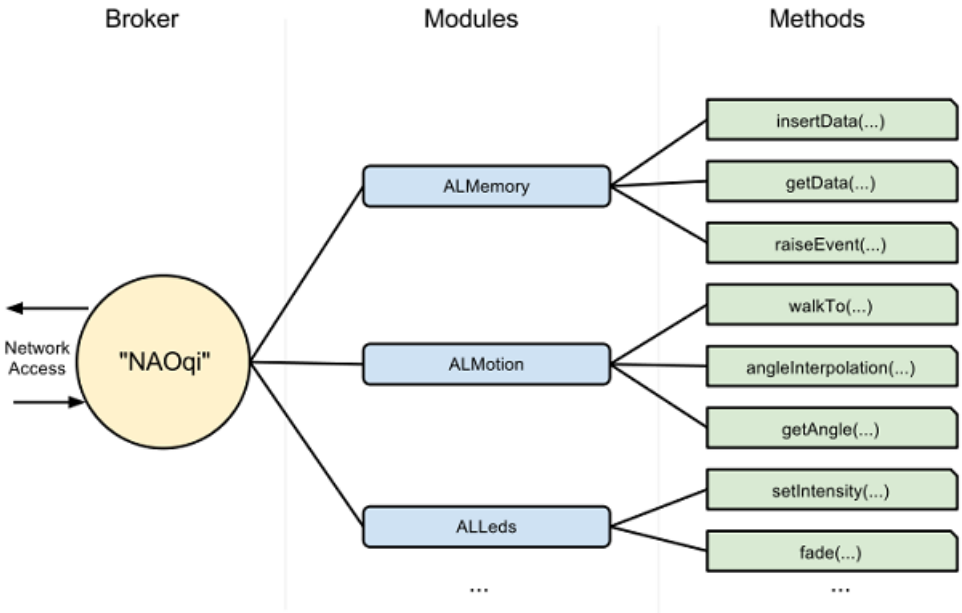

Abbildung 4.7: NAOqi Method-Tree

Broker
Der Broker ist ein Objekt, der zwei generelle Rollen einnimmt. Erstens ist das ein Verzeichnis - Dienst, mit dessen Hilfe Module und Methoden gefunden werden können und zweitens ein Netzwerk - Anschluss, der es möglich macht Methoden verknüpfter Module auch außerhalb des Prozesses aufzurufen.

Die Meiste Zeit muss sich keine Gedanken um die Broker gemacht werden, da diese ihre Arbeit selbstständig und transparent machen. Geschriebener Code kann gleich sein, ob für Aufrufe an „remote Module“ (dt.: entfernt; anderer Prozess oder anderes System) oder „lokale Module“ (gleicher Prozess).

Proxy
Ein Proxy ist ein Stellvertreter - Objekt das sich genau so verhält, wie das Modul das es repräsentiert. Wenn ein Proxy - Objekt des ALMotion Moduls instanziiert wird, erhält das Proxy - Objekt auch alle Methoden des ALMotion Moduls.

Um ein Proxy eines Moduls zu instanziieren gibt es zwei Möglichkeiten:

- Nur den Namen des Moduls benutzen. In diesem Fall muss der auszuführende Code und das Modul das verbunden werden soll im selben Broker liegen. Dies ist ein „lokaler“ Aufruf
- Zusätzlich zum Namen des Moduls auch die IP und den Port des Broker benutzen. In diesem Fall muss das Modul im zugehörigen Broker liegen. Dies ist ein „remote“ Aufruf.

Der genaue Unterschied zwischen „remote“ und „lokalen“ Modulen wird im folgenden erklärt.

4.3.3 Module

Typischerweise ist ein Modul eine Klasse innerhalb einer Bibliothek und wird automatisch instanziiert wenn diese durch autoload.ini geladen wird. Neue Methoden können an Klassen gebunden werden, die von ALModule erben. Dadurch werden die Methoden mit ihrem Namen und ihrer Signatur dem Broker öffentlich gemacht, so dass diese anderen verfügbar wird.

Ein Modul kann, wie oben bereits erwähnt, entweder „remote“ oder „lokal“ sein.

Lokale Module sind zwei (oder mehr) Module, die im selben Prozess gestartet wurden. Sie kommunizieren miteinander lediglich über **einen** Broker. Durch

den gemeinsamen Prozess können sie sich Variablen teilen und einander Methoden ohne Serialisierung oder Netzwerkverbindung aufrufen. Dies erlaubt die schnellste Kommunikation untereinander. Lokale Module werden als Bibliothek kompiliert und können ausschließlich auf dem Roboter ausgeführt werden. Sie sind sehr schnell und effizient im Umgang mit dem Arbeitsspeicher.

Remote Module kommunizieren über das Netzwerk miteinander. Jedes remote Module benötigt einen Broker um mit anderen Modulen zu sprechen. Der Broker nutzt dabei das Netzwerkprotokoll SOAP[1] um die Kommunikation bereitzustellen. Schnelles Ansprechen von Modulen ist über ein remote Modul nicht möglich, beispielsweise bei direkter Adressierung des Arbeitsspeichers. Remote Module werden als ausführbare Dateien kompiliert und können außerhalb des Roboters aufgerufen werden. Remote Module sind einfacher zu benutzen und können dadurch von außen einfacher debuggt werden. Allerdings sind sie langsamer und weit weniger effizient wie lokale Module.
Die Kommunikation zwischen remote Modulen kann über zwei Wege erfolgen. Erstens **Broker to Broker** und zweitens **Proxy to Broker**.

Der Unterschied liegt darin, dass Broker to Broker eine wechselseitige, Proxy to Broker nur eine einseitige Kommunikation eröffnet. Bei zwei Modulen B und C kann bei Broker to Broker B Methoden von C und C Methoden von B aufrufen. Bei Proxy to Broker ist dies nur in die Richtung von B nach C möglich, nicht umgekehrt. Listing 4.1 zeigt die Implementierung beider Kommunikationsarten.

```
1 /* Broker to Broker */
2 Proxy proxy = new Proxy(<modulename>);
3 /*Proxy to Broker */
4 Proxy proxy = new Proxy(<modulename>, <ip_adress>,
    <port_number>);
```

Listing 4.1: Kommunikationsarten Module

Blocking und non - Blocking Aufrufe
Das NAOqi - Framework bietet zwei Möglichkeiten an, Methoden aufzurufen.

Abbildung 4.8 zeigt das Schema eines **Blocking calls**. Diese sind wie normale Methoden - Aufrufen. Die nächste Anweisung wird erst nach dem Ende der vorherigen ausgeführt und währenddessen ist keine Ausführung von anderem Code möglich. Alle Aufrufe können eine Exception auslösen und sollten in einen

[1]Simple Object Access Protocol, dient u.a. dazu *Remoe Procedure Calls* durchzuführen

try-catch - Block gepackt werden. Ebenso können die Aufrufe Rückgabewerte besitzen.

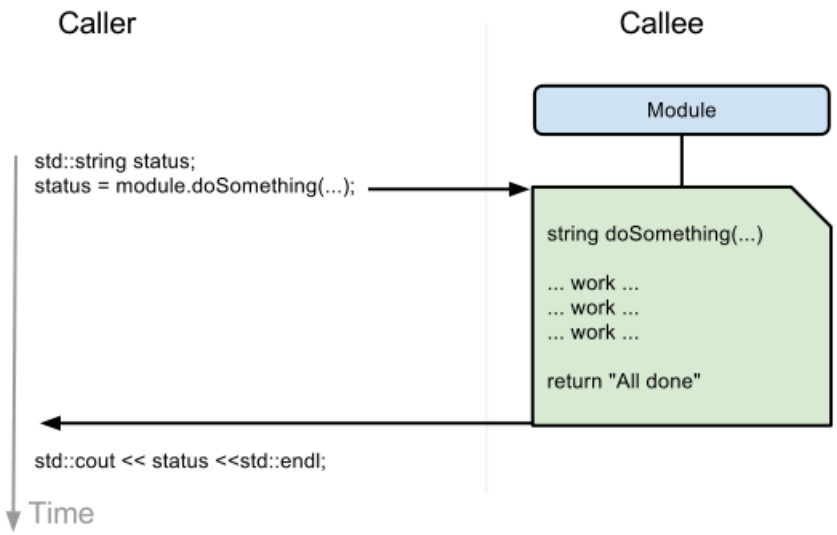

Abbildung 4.8: Blocking Call

Non - blocking calls benutzen das *post*-Objekt einer Proxy. Der Vergleich zu Bild 4.9 zeigt, dass die Ausführung der Methode in einen parallelen Thread gepackt wird. Dadurch ist es möglich andere Anweisungen zur gleichen Zeit auszuführen, z.B. Sprechen während des Laufens. Jedes post - Objekt erzeugt eine *taskID*. Mit dieser ID kann entweder geprüft werden ob der Thread noch läuft, ob auf ihn gewartet oder ob er gestoppt werden soll.

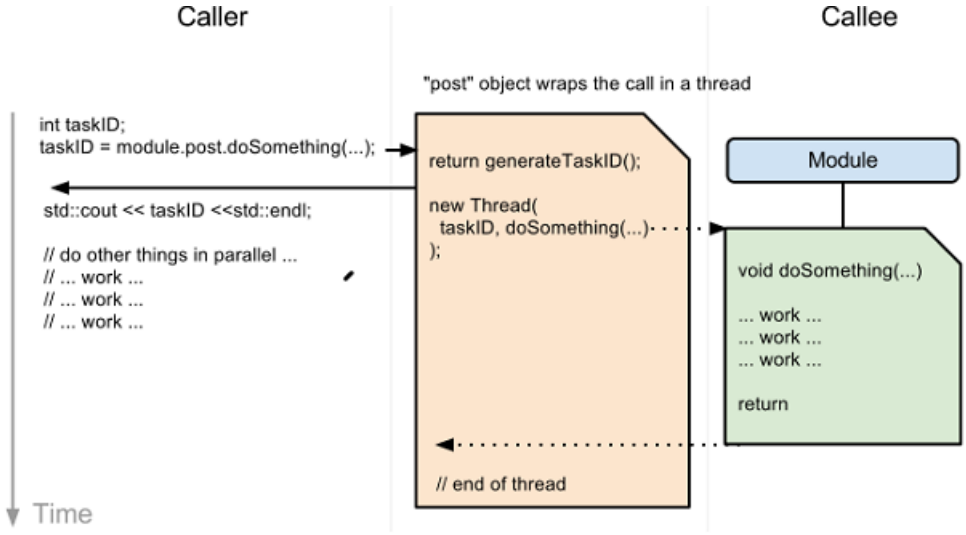

Abbildung 4.9: Non - Blocking Call

4.3.4 .NET SDK

Wie bereits in Kapitel 1 *Einleitung* erwähnt, wurde als Programmiersprache C# gewählt.
Das Benutzen des .NET SDK ist sehr einfach gestaltet. Dazu muss dieses je nach .NET - Version (2/4) installiert werden. In unserem Fall ist das .NET Version 4, da mit *Microsoft Visual Studio 2013* entwickelt wurde. Das SDK beinhaltet Auto - Vervollständigung und nahezu jede NAOqi - Methode kann damit aufgerufen werden.
Listing 4.2 zeigt ein einfaches Beispiel zur Benutzung von NAOqi innerhalb C#. In Zeile eins wird angegeben, dass der Namespace von NAOqi benutzt wird. Damit dies möglich ist, muss davor die naoqi.dll als Verweis dem Projekt hinzugefügt werden. Zeile sieben zeigt die Erzeugung eines Proxy - Objekts vom Typ *TextToSpeech*. Da im Konstruktor eine IP - Adresse und ein Port angegeben wird, handelt es sich hier um eine Proxy to Broker Kommunikation. Auf dem Proxy - Objekt wird in Zeile acht eine Methode mit dem Parameter "Hello World" als blocking call aufgerufen. Das Ergebnis der Methode ist die sprachliche Ausgabe des Strings Hello World durch den Roboter.

```
1  using Aldebaran.Proxies;
2
3  class Program
4  {
5    static void Main(string[] args)
6    {
7       TextToSpeechProxy tts = new
           TextToSpeechProxy("<IP OF YOUR ROBOT>",
           9559);
8       tts.say("Hello World");
9    }
10 }
```

Listing 4.2: Beispiel .NET SDK

Kapitel 5

Realisierung

In diesem Kapitel wird näher auf das Zusammenspiel von Sensorik (Kinect) und Aktorik (Nao) eingegangen. Zunächst wird die Idee der Softwarearchitektur vorgestellt. Im Anschluss daran, wird auf die entworfenen Algorithmen näher eingegangen.

To do (7) **To do** (8)

5.1 Architektur

Die Architektur der Anwendung kann grob in zwei Teilbereiche gegliedert werden, den Sensorik- und den Aktorik-Bereich. Der Sensorteil ist dafür zuständig, die entsprechenden Gesten des Benutzers zu ermitteln und zu verarbeiten. Bei der Digitalisierung der Gesten werden diese mit einem Medianfilter vorgefiltert, sodass extreme Ausreiser verschwinden, die ungewollte Bewegungen des Roboters zur Folge hätten. Der Aktorteil der Anwendung transformiert die Posen der Armgelenke in den Nao-Wertebereich und sendet diese an den Roboter.

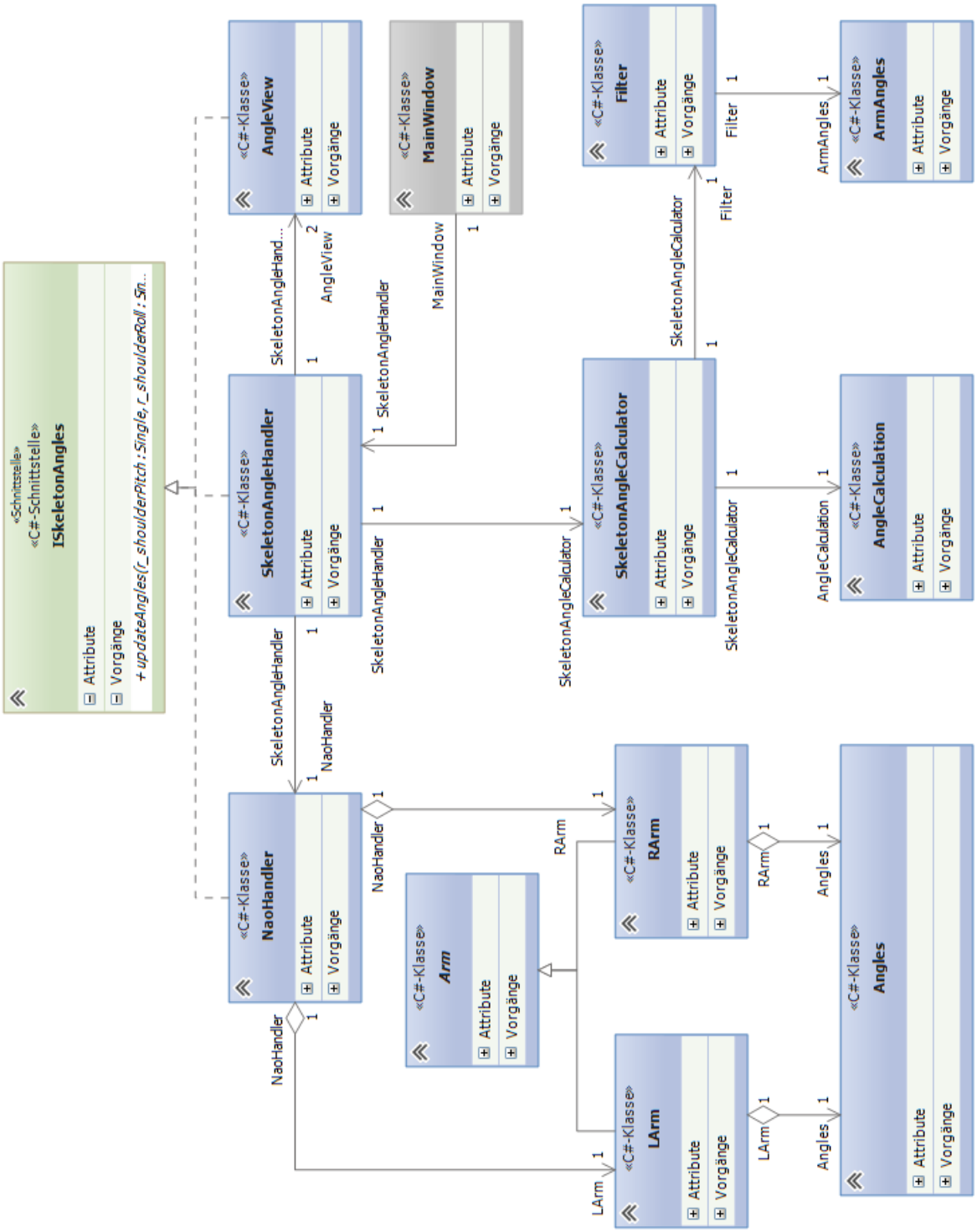

Abbildung 5.1: UML - Klassendiagramm

5.1.1 MainWindow

Die C# Applikation wird als WPF-Anwendung gestartet. Dies geschieht in der Klasse **MainWindow**. Hierbei werden alle Parameter initialisiert und alle nötigen Objekte erzeugt, wie der KinectSensor und der KinectHandler. Es wird wie in Kapitel ...SDK... beschrieben, der Tiefenstream und der Skeletonstream der Kinect aktiviert und anschließend ein ÄllFramesReadyEvent registriert, das benutzt wird, sobald ein Skelett vor der Kamera vollständig erkannt wurde. Im Eventhandler wird der Berechnungs-Handler mit der aktuellen Position des gefilterten Skelettes versorgt. Zudem wird die GUI mit dem aktuellen RGB-Bild aktualisiert und erhält noch ein Skelett-Overlay, sodass man auch im Fenster sehen kann, dass ein Skelett erkannt wurde und kann dieses betrachten. Eine weitere Funktion dieser Klasse ist, das kontrollierte Beenden der Applikation. Nach schließen des Programmes, werden alle Berechnungen gestoppt, die Handler heruntergefahren und Kinect ausgeschaltet.

5.1.2 SkeletonAngleHandler

Der AngleHandler hat verschiedene Aufgaben. Zunächst erzeugt er in der Initialisierungsphase zwei Fenster, die in den späteren Phasen die errechneten Winkel ausgeben. Für jede Seite (rechts, links) wird ein Fenster erzeugt, dass alle 4 relevanten Winkel enthält. Des weiteren wird ein Thread erzeugt, der für die Errechnung der Winkelwerte aus dem ausgelesenen Skelett zuständig ist. Da diese Aktion relativ rechenintensiv ist, wurde sie in einen eigenen Thread ausgelagert. Zudem wird während der Initialisierungsphase noch der NaoHandler erzeugt, der für die spätere Übertragung und Transformation der errechneten Werte an den Roboter zuständig ist. Der Handler selbst verfügt über eine Liste mit Beobachtern, die informiert werden, falls eine neue Berechnung erfolgt ist. Beobachter sind konkret die beiden Fenster für die Debugging-Ausgabe und der Nao-Handler. Hierbei könnte somit dynamisch noch nach Bedarf ein neuer Beobachter hinzugefügt werden, der die Winkelwerte der menschlichen Pose verwenden könnte. Der Handler selbst wird durch das Event **AllFramesReady** in der Klasse MainWindow über seine Methode **updateSkeleton** benachrichtigt und erhält ein neu erkanntes Skelett. Dieses gibt er an den Berechnungsthread weiter, der eine neue Berechnung startet. Nach erfolgreicher Berechnung werden alle Beobachter über die neuen Werte informiert.

5.1.3 SkeletonAngleCalculator

5.1.4 Interfaces

Als Schnittstelle zwischen der Erkennung und der Ausführung der Armpositionen wird essenziell die Methode *updateAngles* vom Interface *ISkeletonAngles* verwendet. Diese stellt alle benötigten Winkel wie Shoulder Pitch, Shoulder Roll, Ellbow Roll, Elbow Yaw bereit. Somit kann dieses Interface von allen Klassen implementiert werden, die immer die aktuellen Werte benötigen, wie der NaoHandler und die GUI.

5.1.5 Flussdiagramm: Winkelerkennung

To do (9)

5.2 Kinect - Armerkennungsalgorithmus

Ein wesentlicher Teil des Programmes besteht darin, die Gesten einer Person, die sich vor dem Kinect-Sensor befindet, zu erkennen und in die passenden Winkelwerte für den Roboter umzurechnen. Hierfür wird mathematisch gesehen das Kreuzprodukt der passenden Knochen errechnet, um die passenden Winkel zur Steuerung des Nao-bots zu erhalten.

Um einen Arm des Roboters zu steuern, bedarf es folgender Winkel:

- Shoulder Pitch
- Shoulder Roll
- Elbow Roll
- Elbow Yaw

5.2.1 Shoulder Pitch

Der Shoulder Pitch des menschlichen Skelettes kann durch die Joints **Hip, Shoulder und Elbow** errechnet werden. Dabei werden zwei Vektoren aufgespannt. Ein Vektor verbindet den Punkt *Hip* mit dem *Shoulder* Punkt, der andere Vektor stellt den Oberarm dar und verbindet den Punkt *Shoulder* mit dem Punkt *Elbow*. Dabei alle Joints nur von einer Seite des Körpers, also z.B. *HipRight, ShoulderRight, ElbowRight* extrahiert.

5.2.2 Shoulder Roll

Der Shoulder Roll Winkel kann nicht mit drei aneinander liegenden Gelenken ermittelt werden. Deshalb benutzt man jeweils zwei Hilfs-Vektoren, die im Skelett nicht anatomisch verbunden sind. Der erste Vektor wird durch die Hüfte aufgespannt, dies betrifft die Joints *HipRight* und *HipLeft*. Der andere Vektor wird analog zum Oberarm aufgespannt, dies betrifft die Joints *Shoulder* und *Elbow*.

5.2.3 Elbow Roll

Dieser Winkel entspricht der Beugung des Ellenbogens. Dafür notwendig sind die Joints *Shoulder*, *Elbow* und *Hand*. Dabei entsprechen die Vektoren dem Oberarm (Shoulder-Elbow) und dem Unterarm (Elbow-Hand) des menschlichen Skelettes.

5.2.4 Elbow Yaw

Dieser Winkel wird über die Gelenke Shoulder, Hip, Elbow und Hand berechnet. Der erste Vektor wird von Schulter zur Hüfte aufgespannt und der zweite Vektor entspricht dem Unterarm.

Nach der Erstellung dieser Vektoren wird eine Methode aufgerufen, die das Kreuzprodukt der Vektoren errechnet und somit den jeweiligen Winkel zurück liefert.

```
1  public static float getAngle(Vector3D a, Vector3D
       b, Vector3D c)
2  {
3      Vector3D bone1 = a - b;
4      Vector3D bone2 = c - b;
5
6      bone1.Normalize();
7      bone2.Normalize();
8
9      float dotProduct =
           (float)Vector3D.DotProduct(bone1, bone2);
10
11     return (float)Math.Acos(dotProduct);
12 }
```

Listing 5.1: Ermittlung des Winkels mithilfe von drei Punkten

Da die Methode auch mit vier Gelenken (Shoulder Roll, Elbow Yaw) durchgeführt werden kann, existiert auch folgende Methode:

```
1  public static float getAngle(Vector3D a, Vector3D
       b, Vector3D x, Vector3D y)
2  {
3      Vector3D bone1 = a - b;
4      Vector3D bone2 = x - y;
5
6      bone1.Normalize();
7      bone2.Normalize();
8
9      float dotProduct =
           (float)Vector3D.DotProduct(bone1, bone2);
10
11     return (float)Math.Acos(dotProduct);
12 }
```

Listing 5.2: Ermittlung des Winkels mithilfe von vier Punkten

To do (10)

5.3 Nao - Armbewegungsalgorithmus

Der zweite wesentliche Teil des Programms beinhaltet die Übertragung der erhaltenen Winkel der Kinect auf den Roboter. Dies beinhaltet nicht einfach nur die Weiterleitung der Werte sondern auch verschiedene Funktionen wie *Umrechnen* oder *Validieren*. Die Basisalgorithmen hierzu werden im folgenden vorgestellt.

Basis

Das Interface Arm deklariert eine Methode controlArm mit den Parametern für die fünf Armwinkel. Die erbenden Klassen LArm und RArm implementieren diese Methode.

```
1  public override void controlArm( float SP, float
       SR, float ER, float EY, float WY)
2  {
3    float[] newangles = { SP, SR, ER, EY, WY };
4    newangles = convertAngles(newangles);
```

```
5     newangles = verifyAngles(newangles);
6
7     mp.setAngles(joints, newangles,
          Arm.fractionMaxSpeed);
8 }
```

Listing 5.3: Methode controlArm

Das Listing 5.3 zeigt die Definition der Funktion controlArm. Wie im Methodenkopf zu sehen, werden hier fünf Winkel übertragen. Auch wenn WristYaw (WY) nicht durch die Kinect erkannt wird, ist dieser Winkel zur Vollständigkeit trotzdem angegeben.

In Zeile vier werden die Winkelwerte an die Funktion convertAngles übergeben. Diese sorgt dafür, dass die Kinect - Winkel in die passenden Nao - Winkel umgerechnet werden. So ist beispielsweise ein Winkel von 90° des ShoulderPitch - Gelenk im Nao - Gelenkraum 0°. Die Funktion gibt am Ende ein Array der umgerechneten Werte zurück.

Dieses wird in Zeile fünf an die Funktion verifyAngles übergeben. Diese überprüft für jeden Winkel, ob dieser im Nao - Gelenkraum liegt (siehe Kapitel 4.1 - Gelenkraum). Liegt ein Wert nicht im gültigen Bereich, wird der ungültige Wert mit dem Wert der aktuellen Winkelstellung überschrieben.

Nach den Funktionen für die Umrechnung und die Validierung der Winkel wird in Zeile sieben der Befehl zum Bewegen des Arms an Nao geschickt. Dafür ist die Funktion setAngles verantwortlich. Neben den Winkeln (newangles) und den Namen der betroffenen Gelenke (joints) wird zusätzlich noch der Wert Arm.fractionMaxSpeed übergeben. Dieser gibt an, mit welcher Geschwindigkeit in Prozent die Gelenke an die Winkelstellungen gefahren werden sollen. Nach subjektiven Tests wurde hier der Wert 0.3f (30%) gewählt.

Erweitert

Da das ständige Neuausrichten der Winkel für die Motoren der Gelenke sehr anfordernd ist, wurden neben dem Glätten der Kinect - Winkel durch den Median - Filter auch noch im Armbewegungsalgorithmus zwei Erweiterungen implementiert.

```
1 public override void controlArm( float SP, float
      SR, float ER, float EY, float WY)
2 {
3     float[] newangles = { SP, SR, ER, EY, WY };
4     newangles = convertAngles(newangles);
```

```
5     newangles = verifyAngles(newangles);
6     newangles = angles.checkDifference(newangles);
7
8     mp.post.angleInterpolationWithSpeed(joints,
          newangles, Arm.fractionMaxSpeed);
9 }
```

Listing 5.4: Methode controlArm erweitert

In Listing 5.5 ist zu sehen, dass der Algorithmus um eine Funktion erweitert und eine andere geändert wurde.

Der Pseudo - Code der Methode checkDifference (Zeile 6) ist in 5.5 zu sehen. Darin wird überprüft ob die Differenz zwischen dem neuen Winkel und dem aktuellen Winkel kleiner als eine Schranke ist. Ist das nicht der Fall, wird der neue Winkel übernommen, ansonsten bleibt der aktuelle Winkel stehen. Als Schranke wurde für EllbowRoll 5° und die anderen Winkel 10° gewählt. EllbowRoll hat eine niedrigere Schranke, da der Gelenkraum dieses Gelenks deutlich kleiner ist, wie die der restlichen Winkel (siehe 4.1 - Gelenkraum).

```
1 foreach angle in newAngles
2  angle = (differenceOf(newAngle,currAngle) <=
       evenAngle) ? (currAngle) : (newAngle);
3 return newAngles;
```

Listing 5.5: Pseudocode checkDifference

Diese Überprüfung soll verhindern, dass sich kleine Änderungen der Kinect - Winkel, also des Menschen, direkt auf die Armstellungen Naos übertragen. Die Werte der Schranken wurden so gewählt, dass die Armstellung Naos subjektiv zu der des Menschen passt.

In 5.5 wurde die Methode setAngles durch post.angleInterpolationWithSpeed ersetzt. Prinzipiell machen beide Methoden das gleiche und zwar bewegen sie die übergebenen Gelenke an die übergebenen Winkel mit der übergebenen Geschwindigkeit. Der große Unterschied liegt allerdings darin, dass setAngles eine Non - Blocking- und post.angleInterpolationWithSpeed eine Blocking - Funktion ist. In diesem Fall ist eine Non - Blocking - Funktion schlecht, da sie es erlaubt, dass die ausführende Methode zu jedem Zeitpunkt unterbrochen und mit neuen Parametern erneut gestartet werden kann. Unter anderem führt das dazu, dass die Arme „ruckeln“ wenn sie bewegt werden. Die Blocking - Funktion wird erst dann wieder erneut aufgerufen, wenn die zuvor Ausgeführte beendet wurde. Dies

entspricht auch eher dem Ziel, dass die Bewegungen des Menschen **nachgeahmt** werden und es keine eins zu eins Spiegelung des Menschen auf den Roboter ist.

5.4 Prototypen

Im Rahmen der Entwicklung des finalen Endprogramms wurden mehrere Prototypen entwickelt um sich erstens in die einzelnen SDKs einzuarbeiten und zweitens die Prototypen zu erweitern und zu vereinen. Diese Prototypen werden hier vorgestellt.

5.4.1 Erster Kinect-Prototyp

Anhand der obigen Überlegungen wurde der erdachte Algorithmus zunächst anhand eines Prototyps implementiert. Dieser sollte zunächst dazu dienen, das Microsoft Kinect SDK näher kennen zu lernen**To do** (11). Der erste Prototyp erfüllte folgende Funktionen:
-Kinect SDK Library -(Connect-Disconnect von Kinect) -Winkelerkennung des Benutzers mit Anzeige in Grad -Anzeige des Kamerabildes mit Gelenkkennzeichnung von Kopf und Armen **To do** (12)

5.4.2 Erster Nao-Prototyp

Zur Einarbeitung in das Nao-SDK wurde eine Anwendung erstellt, die verschiedene Armwinkel an den Roboter übermitteln können.

In Bild 5.2 *Erster Nao-Prototyp* sind verschiedene Eingabefelder zu sehen. In die oberste wird die IP - Adresse eingetragen, die dem zu benutzenden Nao (hier: lokal in Webots) entspricht. Bei einem Klick auf den Button *Connect* wird im Hintergrund ein neuer Proxy geöffnet. Wenn die Verbindung erfolgreich aufgebaut wurde, wird auch der Button *Move* klickbar.

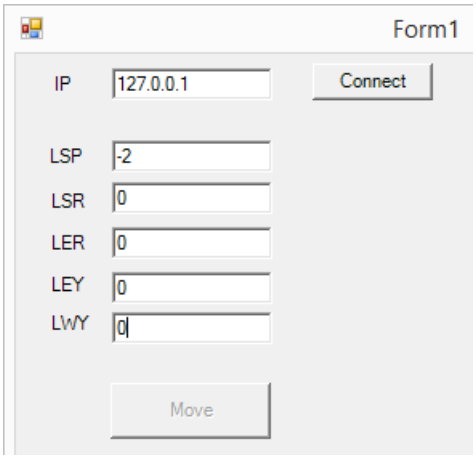

Abbildung 5.2: Erster Nao-Prototyp

In die anderen Felder müssen die einzelnen Winkel (im Bogenmaß) für die Gelenke (LSP = LeftShoulderPitch, LSR = LeftShoulderRoll, usw.) eingetragen werden. Bei Klick auf den Button *Move* werden die einzelnen Winkel des linken Arms in die Position der eingetragenen Werte gefahren. In diesem Fall hebt der Roboter den linken Arm nach oben an, da *LSP* mit -2rad definiert ist und dies im Nao - Gelenkraum einem Winkel von $-115°$ entspricht (siehe Kapitel 4.1 und Bild 1).

Ist ein angegebener Winkel nicht im Gelenkbereich wird die komplette Bewegung nicht ausgeführt. (gilt nur für diesen Prototyp)

5.4.3 Zweiter Prototyp

Anhand der ersten Prototypen wurde das nun erworbene Wissen in eine gemeinsame Architektur eingebettet (Siehe Kapitel Architektur). Als Verbesserungen wurden alle benötigten Winkel auf zwei separaten Fenstern angezeigt, eines für jeden Arm. Auf der GUI wird nun das komplette Skelett über das RGB-Bild gelegt. **To do** (13)

5.4.4 Endprogramm

Um den effektiven Algorithmus der Winkelerkennung noch effizienter zu gestalten wurde noch ein Mittelwertsfilter implementiert, um die Ausreißer in den erkannten Winkeln zu eliminieren und somit die Messungenauigkeit der Gelenkpositionen zu verringern.

To do (14)

Kapitel 6

Ausblick

Schluss

6.1 Aufgetretene Probleme

6.2 Weiterführende Anwendungsgebiete

Es stellt sich nun die Frage, welchen technischen Nutzen die Teleoperation innerhalb der Gesellschaft finden könnte. Zunächst gibt es ein großes Anwendungsgebiet in der Medizin....

Anhang

Gelenkraum Arme

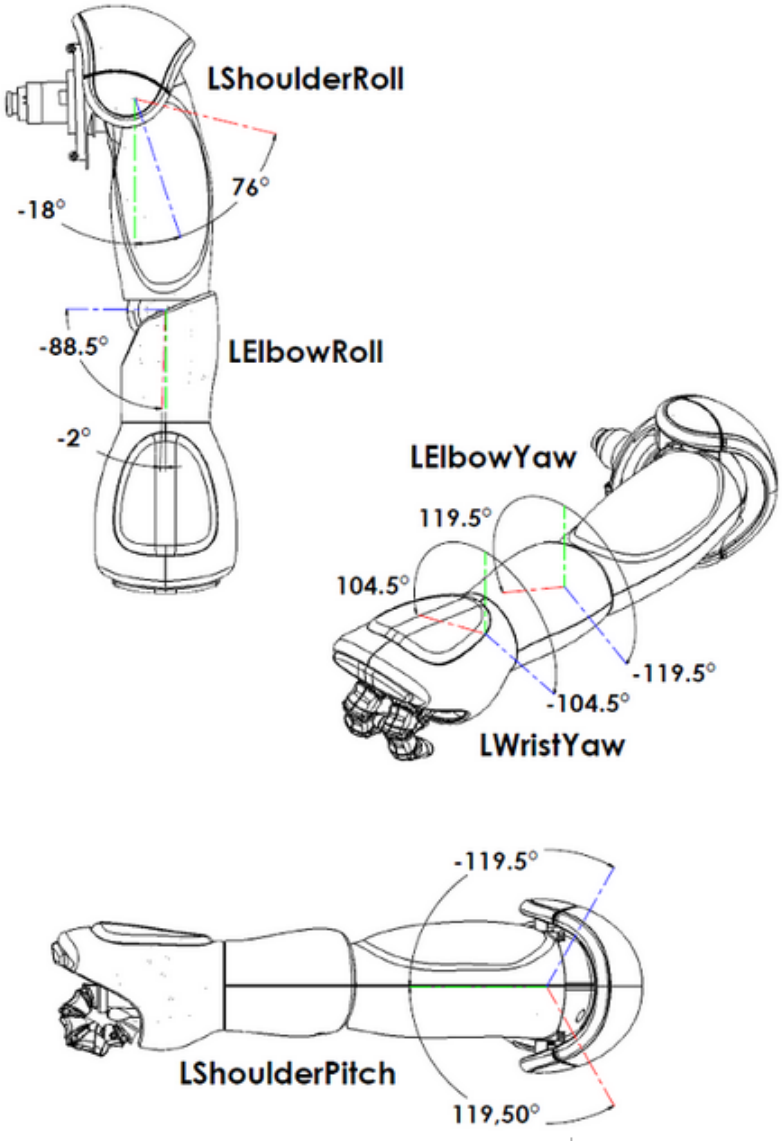

Abbildung 1: Gelenkraum linker Arm

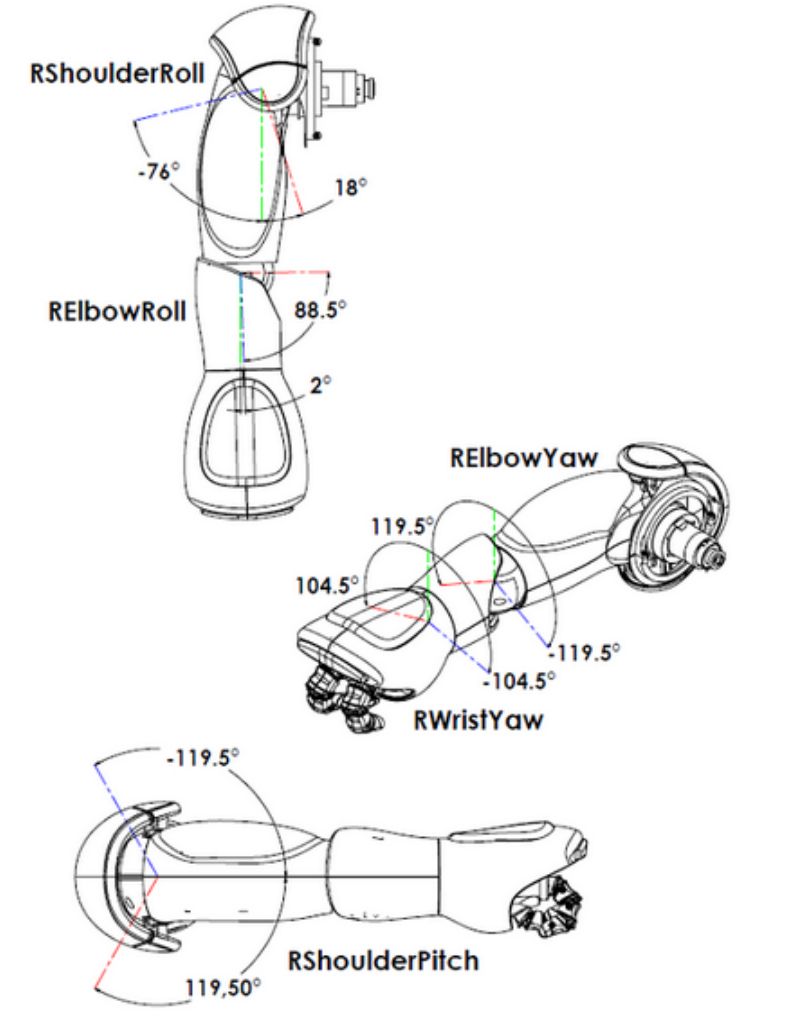

Abbildung 2: Gelenkraum rechter Arm

To do...

- ☐ 1 (p. 14): Programm um Tilt erweitern
- ☐ 2 (p. 15): TiefenstreamBild
- ☐ 3 (p. 18): Screenshot von Deep Stream (Kinect Studio)
- ☐ 4 (p. 22): Screenshot Skelett
- ☐ 5 (p. 22): Screenshot Kinect-Studio
- ☐ 6 (p. 26): vllt. einzelne Bilder rein?
- ☐ 7 (p. 37): Wahl Bibliothek, warum?...
- ☐ 8 (p. 37): Laborbedingungen Steuerungen, Vor- und Nachteile der Bedienung
- ☐ 9 (p. 37): Architekturidee: Input, Verarbeitung(evtl. Filter-¿Probleme Ruckeln?), Output,
- ☐ 10 (p. 40): Methode beschreiben
- ☐ 11 (p. 42): getrennt zusammen?
- ☐ 12 (p. 45): Bild
- ☐ 13 (p. 45): Screenshot
- ☐ 14 (p. 46): Screenshot von Prototypprogramm