Universität Bielefeld
Technische Fakultät
FG Ambient Intelligence

Masterarbeit

Steuerung von Umgebungsintelligenz mit multimodalen Gesten

David Fleer
Studiengang Intelligente Systeme (M.Sc.)

21. November 2011

Betreuer
Dipl.-Inform. Christian Leichsenring
Dr. rer. nat. Thomas Hermann

Diese Masterarbeit wurde von Dipl.-Inform. Christian Leichsenring betreut und begutachtet. Der zweite Gutachter war Dr. rer. nat. Thomas Hermann.

Ich danke meinem Betreuer Dipl.-Inform. Christian Leichsenring, der mir jederzeit mit Rat und Tat zur Seite stand. Besonderer Dank gilt auch Dipl. Inform. René Tünnermann, Dipl. Inform. Ulf Großekathöfer und Dipl. Inform. Eckard Riedenklau. Zudem danke ich all jenen, die mich während dieser Arbeit auf vielfältige Art und Weise unterstützt haben.

Inhaltsverzeichnis

1

Einleitung

Ziel dieser Masterarbeit war es, eine existierende intelligente Umgebung mittels Hand- und Armgesten zu steuern. In diesem Kapitel werden dabei zunächst die Motivation und die Lösungsansätze besprochen. In Kapitel 2 wird die Architektur des Gesamtsystems erklärt und in unabhängige Komponenten zerlegt. Im Anschluss wird in Kapitel 3 die Implementation dieser Komponenten und der Verbindung zu einem Gesamtsystem beschrieben. Kapitel 4 befasst sich mit dem Verhalten und der Leistung des entwickelten Gesamtsystems und schließt die Arbeit mit einem Fazit und einem Ausblick auf mögliche Weiterentwicklungen ab.

1.1 Motivation und Ziel

Die fortschreitende Entwicklung der Mikroelektronik ermöglicht die Konstruktion immer kleinerer, günstigerer und gleichzeitig leistungsfähigerer elektronischer Bauteile. Im Rahmen dieser Entwicklung zieht die Mikroelektronik immer weiter in die alltägliche Umgebung ein, zum Beispiel bei der Gebäudetechnik und bei komplexen Haushaltsgeräten. Dies führt jedoch auch zu zunehmender Komplexität im Alltag. Ein Ziel der *Ambient Intelligence* ist daher, durch in die Umgebung eingebettete und intuitiv benutzbare Schnittstellen den Umgang mit dieser Technik möglichst unkompliziert zu gestalten.
Im Rahmen der Diplomarbeit „Smartphonebasierte Mixed-Reality-Interaktionen für intelligente Umgebungen“ (auch MIRIAM genannt) von Bastian Kriesten [15] wurden im *Ambient-Intelligence*[1]-Labor am Exzellenzcluster *Cognitive Interaction Technology*

[1] Die Webseite der Ambient Intelligence Group findet sich unter `http://www.cit-ec.de/ami`

der Universität Bielefeld bereits einige Alltagsobjekte an ein Netzwerk angebunden und können über dieses gesteuert werden. Die vernetzten Objekte werden *Smart Objects* genannt. Zu den bereits existierenden Smart Objects gehören unter anderem eine mehrfarbige Umgebungsbeleuchtung, eine steuerbare Mehrfachsteckdose und die Möglichkeit zur Musikwiedergabe. Zur Steuerung wurde dabei ein Mobiltelefon verwendet. Ziel dieser Arbeit war es, diese bestehenden Smart Objects in einer natürlichen Umgebung über Zeigen auszuwählen und dann über Arm- und Handgesten steuern zu können. In der Praxis könnte der Benutzer dann über einen Fingerzeig und eine Handbewegung eine Lampe oder einen Tischventilator ein- oder ausschalten oder eine Musikanlage steuern. Da das Zielobjekt zunächst durch Zeigen ausgewählt wird, ist die Bedeutung der im Anschluss ausgeführte Geste vom ausgewählten Objekt abhängig. Durch diese Kontextsensitivität kann die Zahl der benötigten Gesten auch bei vielen Objekten vergleichsweise gering gehalten werden. Dies erhöht die Erlernbarkeit im Vergleich zu einem System ohne Kontextsensitvität, bei dem jedes Objekt seinen eigenen, disjunkten Gestenkanon benötigt.
Die Umsetzung sollte möglichst günstige und handelsübliche Komponenten[2] verwenden und auf normalen Desktoprechnern lauffähig sein. Zudem sollte das System in der alltagsähnlichen Umgebung des *Ambient-Intelligence*-Labors funktionieren.

1.2 Lösungsansätze

Um die in Abschnitt 1.1 beschriebenen Ziele zu erreichen, muss die Position und Körperhaltung (auch Positur genannt) des Benutzers erfasst werden. Aus Position und Körperhaltung kann dann bestimmt werden, auf welches Objekt der Benutzer zeigt und welche Geste er ausführt. Dazu muss aus der Menge der verfügbaren Sensorsysteme ein System oder eine Kombination von Systemen gewählt werden, welches die in Abschnitt 1.1 genannten Anforderungen erfüllt. Nicht geeignet sind dabei Systeme, die vom Benutzer getragen werden müssen. Dazu gehören unter anderem Sensoren wie zum Beispiel Datenhandschuhe, die am Benutzer angebracht sind und so seine Position und Körperhaltung ermitteln. Auch Kamerasysteme, die vom Benutzer zu tragende Marker verwenden sind ungeeignet. Motion-Capturing-Systeme, die Infrarotmarker verwenden, gehören in diese Kategorie. All diese Systeme sind deshalb untauglich, da in einem potentiellen Alltagseinsatz nicht davon ausgegangen werden kann, dass der Benutzer die Sensoren oder Marker jederzeit tragen kann oder will.
Unter diesen Einschränkungen sind Kamerasysteme eine mögliche Lösung, da diese nur im Raum angebracht oder aufgestellt werden müssen. Ohne Verwendung von Markern ist es aber schwierig, die genaue Position und Körperhaltung dies Benutzers auf Basis eines einzelnen 2D-Bildes zu erfassen. Ein Ansatz ist die Verwendung mehrerer Kame-

[2] Solche Komponenten werden auch als *Commercial off-the-shelf* bezeichnet.

ras. Dabei wird die Umgebung über mehrere Kameras aus verschiedenen Perspektiven aufgenommen. Kann nun im Bild jeder Kamera der Benutzer erkannt werden, so können aus den verschiedenen Kamerabildern die Benutzerposition und ein Körpermodell des Benutzers in Voxelform errechnet werden [10]. Der Nachteil dieses Ansatzes ist jedoch, dass mehrere Kameras erforderlich sind. Dies führt zu einem höheren Preis. Zudem müssen alle Kameras kalibriert und miteinander synchronisiert werden, was die Komplexität erhöht. Auch muss der Benutzer in den einzelnen Bildern ausreichend vieler Kamera erkannt werden. Dies kann in einer natürlichen Umgebung und bei wechselnden Beleuchtungsverhältnissen schwierig sein.
Ein ähnlicher Ansatz ist der Einsatz von Tiefenkameras. Wegen der zusätzlichen Tiefeninformation reichen bereits wenige oder gar nur eine Kamera aus, um den Benutzer zu erfassen, solange er für die Kamera(s) sichtbar ist. Nachteil ist hierbei, dass Tiefenkameras in der Vergangenheit meist teure Spezialgeräte waren. Dies widerspricht der Anforderung eines Systems mit günstiger, allgemein verfügbarer Hardware. Diese Situation änderte sich gegen Ende 2010, als Microsoft die Kinect-Tiefenkamera als Zubehör zur XBox360-Spielekonsole auf den Markt brachte [3]. Als Produkt für den Massenmarkt ist die Kinect vergleichsweise günstig und weit verbreitet. Zudem ist Kinect speziell für die Erkennung von Benutzern und deren Körperhaltung in Alltagsumgebungen ausgelegt. Seit diese Funktionalität über diverse Bibliotheken (siehe Abschnitt 3.1.1) auch auf normalen Desktoprechnern zur Verfügung steht, eignet sich die Kinect-Kamera gut für die Umsetzung des geplanten Systems. Kinect arbeitet auf Basis einer aktiven Infrarotquelle. Da sich mehrere Kinect-Sensoren somit gegenseitig stören können, sollen in dieser Arbeit nur einzelne Kinect-Kameras verwendet werden.
Ein Problem bei der Umsetzung des Systems ist es, an das System gerichtete Gesten von Alltagsgesten, wie sie zum Beispiel im Rahmen einer Unterhaltung verwendet werden, zu unterscheiden. Alltagsgesten, welche fälschlicherweise als an das System gerichtete Gesten interpretiert werden, werden falsche Positive genannt. Falsche Positive sind in der Praxis ein ernstes Problem, da sie unerwünschte und eventuell unbemerkte Änderungen an den Smart Objects auslösen können. Falsche Positive müssen daher vermieden werden.

1.2.1 Verwendung eines Tokens

Ein Ansatz zur Vermeidung falscher Positive ist die Verwendung eines speziellen Gegenstands, einem sogenanntem *Token*. Nur wenn der Benutzer diesen Gegenstand bei der Ausführung einer Geste in der Hand hält, wird diese Geste auch vom System verarbeitet. Denkbar ist zum Beispiel eine Zauberstab-Metapher, bei der ein kleiner Stab als Token verwendet wird. Dieser Stab würde in einer Farbe eingefärbt, die sich von der Hautfarbe des Benutzers möglichst stark unterscheidet. Zur Erkennung des Tokens würden aus Tiefen- und Farbbild eine Darstellung der Szene als Punktwolke berechnet. Im Anschluss könnte die Farbe der Punkte in der Nähe der Hand des Benutzers geprüft

werden. Wenn ausreichend viele Punkte die Farbe des Tokens haben, wird angenommen, dass der Benutzer das Token in der Hand hält.
In der Praxis erwies sich dieses Verfahren jedoch als untauglich. Kinect errechnet das Tiefenbild mit Hilfe eines ausgesendeten Infrarot-Punktmusters[3]. Ein dünner Stab ist daher auf dem Tiefenbild schon ab einem Abstand von eine bis zwei Metern nicht mehr zuverlässig auf dem Tiefenbild zu erkennen[4], auch wenn er auf dem Farbbild mit gleicher Auflösung noch auszumachen ist. Ohne Tiefenbild ist es jedoch schwierig, den Stab vom Hintergrund zu trennen. Zudem können Hintergrundobjekte mit der Farbe des Stabes nur sehr schwer vom Stab unterschieden werden, zumal die Erkennung der Farbe wegen wechselnder Beleuchtungsverhältnisse nicht zu strikt sein darf. Daher lässt sich das Token-Verfahren nur mit einem ausreichend großen Token umsetzen. In der Praxis führt dies zu einem zu großem und damit unpraktischen Token. Daher wurde der Token-Ansatz verworfen und stattdessen der multimodale Ansatz aus Abschnitt 1.2.2 verwendet.

1.2.2 Multimodalität

Ein anderer Ansatz zur Vermeidung von falschen Positiven ist Multimodalität. Dabei werden Informationen aus verschiedenen Modalitäten, also verschiedenen Datenquellen, kombiniert. Zur Vermeidung falscher Positive werden Modalitäten mit möglichst unterschiedlichen Sensoren verwendet. Nur wenn alle Modalitäten gleichzeitig eine Interaktion des Benutzers mit dem System anzeigen, kann auch eine Aktion ausgelöst werden. Daher können für einzelne Modalitäten problemlos falsche Positive auftreten, solange mindestens eine Modalität ein negatives Ergebnis[5] liefert.
Als zweite Modalität neben der Benutzerposition und -positur werden dabei Geräusche verwendet. Diese sind von den durch die Tiefenkamera gelieferten Tiefenbildern größtenteils unabhängig. Zudem lassen sich mit einem einzigen omnidirektionalen Mikrofon Geräusche für einen ganzen Raum erfassen. Bestimmte Laute sind für den Benutzer leicht zu erzeugen und für das System relativ einfach zu erkennen (siehe Abschnitt 3.4). Insbesondere soll das Fingerschnipsen als Laut verwendet werden. Fingerschnipsen lässt sich mit nur einer Hand erzeugen. Im einfachsten Fall wird dabei vom Benutzer die Hand verwendet, die schon zum Zeigen und zur Ausführung der Geste verwendet wird. Dank der hohen Lautstärke und der schnellen Pegeländerung des Audiosignals[6] ist Fingerschnipsen zudem einfach zu erkennen.
Die Multimodalität führt somit zu folgendem Ablauf: Der Benutzer zeigt mit ausgestrecktem Arm auf das Objekt, mit dem interagiert werden soll. Dabei schnipst der Be-

[3] Dieses Verfahren wird auch als *Structured Light* bezeichnet.

[4] Der genaue Abstand hängt dabei von der Dicke des Stabes ab.

[5] Ein negatives Ergebnis bedeutet dabei, dass keine Interaktion des Benutzers mit dem System festgestellt wurde.

[6] Eine schnelle Signaländerung wird auch Transient genannt.

nutzer mit den Fingern. Nur wenn der Benutzer mit ausgestrecktem Arm auf ein gültiges Objekt zeigt und gleichzeitig schnipst, beginnt das System mit der Gestenaufnahme. Auf Basis dieser Geste wird dann eine passende Aktion ausgewählt und ausgeführt. Fingerschnipsen ohne Zeigen oder Zeigen ohne Schnipsen führt zu keiner Aktion.

1.3 Verwandte Arbeiten

Es wurden bereits andere Versuche unternommen, intelligente Umgebungen über Zeigen und/oder Gesten steuern zu können. Einige dieser Versuche sollen hier kurz vorgestellt werden.
Ein einfacher Ansatz besteht darin, jeweils nur ein Objekt zu steuern. Dabei wird der Benutzer mittels einer Kamera aus Sicht des Objekts aufgenommen. Daher muss die Zeigerichtung des Benutzers nicht erkannt werden. Das System von Freeman und Weissman [13] zum Beispiel erlaubt so die Steuerung eines Fernsehrs mittels Gesten. Dabei wird nur die Bewegung der geöffneten Hand im Kamerabild zur Gestenerkennung herangezogen.
Der häufigste Ansatz für ein räumlich arbeitendes System ist die Berechnung der Zeigerichtung über mehrere Kameras. Dabei werden die Hände und der Kopf des Benutzers in den Bildern mehrerer Kameras gefunden. Ist die Position der Kameras relativ zueinander bekannt, kann die räumliche Position von Kopf und Hand und somit die Zeigerichtung berechnet werden. Um Kopf und Hand im Bild zu finden, werden als Merkmale zum Beispiel die Hautfarbe [14] oder kombinierte Farb- und Geometriemerkmalen verwendet [11]. Bei Do et al. [11, S. 24] findet sich eine Übersicht über eine Reihe von Systemen, die konventionelle Kameras verwenden. All diese Systeme arbeiten auf Farbbildern, was eine ausreichend gute Ausleuchtung der Szene erfordert. Der aktive Tiefensensor der Kinect kann hingegen bei nahezu beliebigen Beleuchtungsverhältnissen und sogar in Dunkelheit eingesetzt werden.
Das System von Lin et al. [17] verwendet eine Tiefenkamera in Verbindung mit einer Hautfarbenerkennung, um ein Körpermodell des Benutzers zu extrahieren und so Gesten zu erkennen. Dabei ist das System auf ein einzelnes Objekt beschränkt.
Andere Systeme verwenden ein physisches Token (ähnlich der Zauberstabmetapher aus Abschnitt 1.2.1), wobei diese Systeme Sensoren innerhalb des Tokens verwenden. Beispiele für solche Systeme sind die Arbeiten von Wilson und Shafer [25] und Ouchi et al. [19]. Es existieren auch Hybridsysteme, die sowohl Sensoren in einem Token als auch im Raum montierte Kameras nutzen [20].

2

Architektur

Um Abhängigkeiten zwischen den einzelnen Teilbereichen des Systems gering zu halten, wurden das System zu Beginn in Module zerlegt. Diese wurden dann unabhängig voneinander entwickelt und getestet. Abbildung 2.1 zeigt die einzelnen Module mit ihren Verbindungen. Dabei lässt sich das System in die drei Bereiche Eingabe, Verarbeitung und Ausgabe unterteilen. Diese drei Bereiche, ihre Teilkomponenten und ihre Aufgaben und Anforderungen werden in den Abschnitten 2.1, 2.2 und 2.3 besprochen. Kapitel 3 befasst sich im Anschluss mit der Implementation der einzelnen Module.

2.1 Eingabe

Die Eingabemodule lesen Daten aus den Sensoren aus, extrahieren Merkmale und stellen diese dem Rest des Systems in einem passenden Format zur Verfügung. Die Eingabemodule schirmen die Hardware zudem gegenüber den nachfolgenden Komponenten ab, indem sie unter anderem die Initialisierung, Konfiguration, Kalibrierung und Fehlerbehandlung der Sensoren übernehmen. Dadurch bleiben die nachfolgenden Verarbeitungsmodule hiervon unberührt, was die Komplexität des Systems verringert. Zu den Eingabemodulen gehören das Modul zur Aufnahme der Tiefenkamerabilder und das Modul zur Tonaufnahme über ein Mikrofon. Entwurf und Implementation des Eingabemoduls wurde in Abschnitt 3.1 beschrieben. Das Audio-Eingabemodul wurde wegen seiner geringen Komplexität direkt in das Audiomodul integriert (siehe Abschnitt 3.4.2).

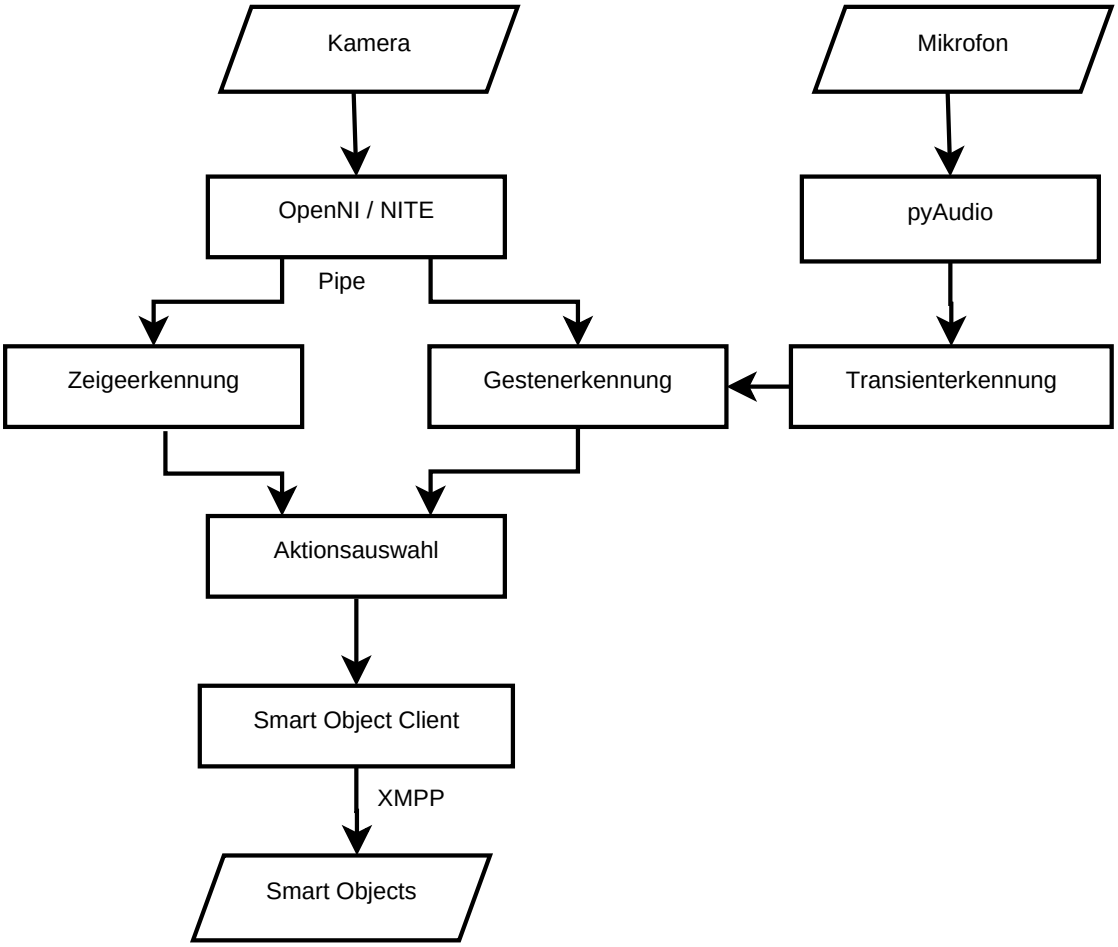

Abbildung 2.1: Strukturdiagram des Gesamtsystems. Die rechteckigen Module sind Software-, die Rauten Hardwarekomponenten. Informationen fließen entlang der Pfeilrichtung. Die Darstellung ist vereinfacht, Hilfs- und Verbindungsmodule fehlen.

2.2 Verarbeitung

Die Verarbeitungsmodule verwenden die von den Eingabemodulen aufgenommenen Sensordaten. Dabei muss zunächst ermittelt werden, ob ein Benutzer auf ein an das System angebundenes Objekt gezeigt und mit einem Schnipslaut eine Geste begonnen hat. Im Anschluss muss die Geste aufgenommen und die Art der Geste ermittelt werden. Dieser Vorgang wurde in drei unabhängige Module für Zeigen, Audio und Gesten zerlegt.

2.2.1 Zeigemodul

Das Zeigemodul bestimmt das Objekt, auf welches der Benutzer zeigt. Das Modul muss zunächst anhand der Eingabedaten feststellen, ob der Benutzer im Moment mit einem seiner Arme auf irgendeinen Ort im Raum zeigt. Ist dies der Fall so muss ermittelt werden, ob auf ein an das System angebundenes Smart Object gezeigt wurde. Dazu wird eine Datenbank aus Objekten verwendet. Abschließend muss die Identität des vom Benutzer ausgewählten Objekts ermittelt und den Ausgabemodulen zugänglich gemacht werden. Entwurf und Implementation dieses Moduls sind in Abschnitt 3.2 beschrieben.

2.2.2 Audiomodul

Das Audiomodul muss Transienten erkennen und segmentieren können. Transienten sind dabei plötzliche, starke Anstiege der Signalenergie. Im Falle eines Audiosignals sind Transienten also plötzlich auftretende laute Geräusche wie zum Beispiel Schnipsen oder Klatschen. Wurde ein Transient erkannt und aus dem Signal herausgetrennt, so muss das Modul Merkmale extrahieren. Auf diesen Merkmalen wird dann eine Klassifikation durchgeführt, um festzustellen ob der gefundene Transient zur richtigen Klasse[1] gehört. Wird der Transient der gesuchten Klasse zugeordnet, so muss dies dem Rest des Systems mitgeteilt werden, damit die Gestenerkennung eingeleitet werden kann. Entwurf und Implementation dieses Moduls sind in Abschnitt 3.4 beschrieben.

2.2.3 Gestenmodul

Das Gestenmodul ermittelt, welche Geste vom Benutzer ausgeführt wurde. Wird das Gestenmodul über den Beginn einer Geste informiert, so nimmt es die zur Geste gehörenden Sensordaten auf. Aus den aufgenommenen Sensordaten wird eine Zeitserie von Merkmalen generiert. Diese Zeitserie wird dann auf Basis von zuvor geladenen Trainingsdaten klassifiziert, um die ausgeführte Geste zu bestimmen. Die identifizierte Geste wird dann dem Ausgabemodul zur Verfügung gestellt. Entwurf und Implementation dieses Moduls sind in Abschnitt 3.3 beschrieben.

2.3 Ausgabe

Aufgabe des Ausgabemoduls ist es, bei einer Benutzeraktion anhand des ausgewählten Objekts und der ausgeführten Geste die gewünschte Veränderung an den Smart Objects auszulösen. Dazu muss anhand eines zuvor geladenen Aktionsmodells zunächst die zu

[1] Zum Beispiel Klatschen, Knackgeräusche oder in diesem Fall das Schnipsen.

Objekt und Geste passende Aktion ausgewählt werden. Im Anschluss muss diese Aktion über die existierende Netzwerk-Infrastruktur an die Objekte übermittelt werden. Entwurf und Implementation dieses Moduls sind in Abschnitt 3.5 beschrieben.

3

Implementation

Die folgenden Abschnitte dokumentieren den Entwurf und die Implementation der in Abschnitt 2 beschriebenen Module. Dabei wird für jedes Modul zunächst der Ansatz beschrieben. Im Anschluss werden einige Details der Implementation besprochen. Weitere Detailinformationen über die innere Funktionsweise der Module können dem im Unterverzeichnis `src` beiliegendem Quellcode entnommen werden. Soweit zutreffend, werden zudem die zur Konfiguration des jeweiligen Moduls verwendeten Dateiformate und die Trainingswerkzeuge zum Erstellen dieser Konfigurationsdateien beschrieben.
Soweit nicht anders erwähnt, wird die Programmiersprache Python in Version 2.6 oder höher verwendet [7]. Die Wahl fiel auf Python, da Python als Skriptsprache mit umfangreichen Bibliotheken für wissenschaftliches Rechnen[1] eine effiziente Umsetzung des Systems ermöglicht. Zudem wurde Python bereits für das MIRIAM-Projekt benutzt, wodurch keine neuen Schnittstellen zum MIRIAM-System entwickelt werden müssen [15]. Auch war die Sprache mir und meinem Betreuer bzw. meinen Gutachtern bereits bekannt.

3.1 Hardware-Anbindung

Die hier beschriebene Hardware-Anbindung bezieht sich auf die Kommunikation mit der Tiefenkamera über die *OpenNI*-Bibliothek. Die Abfrage der Audiohardware ist sehr kompakt und unkompliziert. Daher wurde sie entgegen des ursprünglichen Ansatzes aus Abschnitt 2.1 direkt in das Audiomodul integriert. Informationen hierzu sind in Abschnitt 3.4.2 zu finden.

[1] Insbesondere sind die *NumPy*- und *SciPy*-Pakete zu nennen [5].

3.1.1 Ansatz

Zur Kommunikation mit der verwendeten Kinect-Tiefenkamera wurde die OpenNI-Bibliothek mit dem dazugehörigen Kinect-Treiber verwendet [18]. Das mittlerweile erschienene offizielle Microsoft-Kinect-SDK [3] schied aus, da es zu Beginn der Arbeit noch nicht verfügbar war. Die Entscheidung für OpenNI und somit gegen die alternative *libfreenect*-Bibliothek des OpenKinect-Projektes [6] hatte mehrere Gründe. Das OpenNI-Framework wird von der gemeinnützigen OpenNI-Organisation bereitgestellt. Bei einer ersten Inspektion zeigte sich, dass die Bibliothek selbst, die Dokumentation und die mitgelieferten Werkzeuge bis auf einige Ausnahmen vollständig und von brauchbarer Qualität waren. Zudem wurde die OpenNI-Bibliothek bereits erfolgreich an der Universität Bielefeld eingesetzt. Die Firma PrimeSense ist eine zentrale Kraft bei der Entwicklung des OpenNI-Frameworks. Da PrimeSense auch die Tiefenkamera der Kinect entwickelte und infolgedessen PrimeSense-Hardware im Kinect-Sensor verwendet wird, verfügt OpenNI über eine solide Anbindung für den Kinect-Sensor. Im Vergleich zu OpenKinect/libfreenect ist OpenNI aber nicht auf Kinect-Sensoren beschränkt. Zukünftig erscheinende Geräte mit ähnlicher Funktionalität[2] sollten sich über OpenNI ebenfalls nutzen lassen.
Der entscheidende Faktor für die Auswahl von OpenNI gegenüber OpenKinect war jedoch die Verfügbarkeit der NITE-Middleware. Diese Erweiterung für OpenNI stellt eine Reihe von Analysefunktionen bereit. Wichtig sind dabei die Funktionen zur Benutzererkennung und Kalibration sowie zur Skelettanalyse. Diese Funktionen ermöglicht es, Menschen im Bild des Sensors zu erkennen, zu verfolgen und ein vereinfachtes Skelettmodell aus Gelenkpositionen zu erstellen. Ein solches Gelenkmodell vereinfacht die Implementation des Zeige- und Gestenmoduls wesentlich (siehe Abschnitte 3.2 und 3.3). Zum Entwurfszeitpunkt gab es keine vergleichbare Funktionalität für die libfreenect-Bibliothek. Zusammen mit den oben genannten Faktoren führte dies zu einer Entscheidung für OpenNI.
Das Eingabemodul für die Tiefenkamera soll also auf Basis der OpenNI-Skelettanalyse den nachfolgenden Modulen Informationen über die Körperhaltung des aktuellen Benutzers liefern. Genauer gesagt soll die Position ausgewählter Gelenke des Oberkörpers zur Verfügung gestellt werden. Da NITE bei der Skelettanalyse nur eine Teilmenge der von OpenNI theoretisch unterstützten Gelenke analysiert, beschränkt sich dies in der Praxis auf die Positionen von Kopf, Schultern, Ellbogen und der Hände.
Das Kamera-Eingabemodul soll in einem eigenen Prozess oder Thread laufen, so dass während der Verarbeitung durch das Gesten- und das Zeigemodul bereits das nächste Kamerabild analysiert werden kann. Zudem können so die heutzutage verbreiteten Mehrkernsysteme besser ausgenutzt werden. Da keine offizielle Anbindung von OpenNI an die Programmiersprache Python existiert und sich eine inoffizielle Anbindung zu

[2] Einige der geplanten zukünftigen Geräte verwenden dabei PrimeSense-Technologie, wie sie auch schon in Kinect verwendet wird.

Beginn des Projekts noch in Entwicklung befand, wurde das Kamera-Eingabemodul in der Programmiersprache C++ implementiert. Die Anforderung der Nebenläufigkeit, die Implementation in C++ und der Wunsch nach Modularität führten dazu, dass das Eingabemodul nicht als C++-Bibliothek mit Python-Anbindung sondern als eigenes Programm entwickelt wurde. Der Datenaustausch findet dabei über eine Schnittstelle zur Prozesskommunikation statt. Der Quellcode des Kamera-Eingabemoduls und des restlichen Projekts sind somit voneinander getrennt.
Als Weg der Kommunikation wurde dabei die UNIX-Pipeline verwendet. Bei dieser Methode werden die Ein- und Ausgabeströme von Prozessen miteinander verbunden, um Daten auszutauschen. Genauer gesagt wird der Inhalt einer C-Datenstruktur (`struct`) vom Sender auf den Standard-Ausgabekanal (`stdout`) geschrieben. Dies geschieht in Binärform, also ohne Konvertierung in ein menschenlesbares Format. Diese Binärdaten werden dann vom Empfänger eingelesen. Das Python-Modul `struct` ermöglicht es, die eingelesene binäre C-Datenstruktur in Python zu verwenden. Das `struct`-Modul zerlegt binäre C-Datenstrukturen mit bekanntem Format dafür in von Python verwendbare Variablen. Dabei werden von Rechnerarchitektur und Betriebssystem abhängige Besonderheiten wie die Byteordnung und eventuelle Platzhalter berücksichtigt. Die UNIX-Pipeline als Weg der Interprozesskommunikation ist somit einfach zu implementieren, robust und auf vielen Betriebssystemen verfügbar. Nach einem Vortest zeigte sich zudem, dass eine ausreichend hohe Geschwindigkeit erreicht werden kann. Dazu wurde das Benchmark `ipc-test.py` in Python programmiert, welches Gleitkommazahlen von einer C-Anwendung empfängt. Auf einem handelsüblichen Desktoprechner wurden dabei Geschwindigkeiten von etwa etwa einer Million Gleitkommazahlen[3] pro Sekunde erreicht. Angesichts der erwarteten benötigten Übertragungsrate von wenigen hundert `float` pro Sekunde ist die UNIX-Pipe für die Anforderungen des System somit ausreichend. Um das Kamera-Eingabemodul kontrolliert beenden zu können, werden zusätzlich zur UNIX-Pipeline Signale verwendet.

3.1.2 Implementation

Entsprechend der Überlegungen aus Abschnitt 3.1.1 wurde das Kamera-Eingabemodul in der Datei `openni-provider.cpp` implementiert. Mit der mitgelieferten Makefile für GNU Make [12] ergibt sich daraus das Programm `openni-provider`. Beim Start des Programms wird dabei zunächst die Signalverarbeitung initialisiert, damit beim Beenden des Gesamtsystems auch das Eingabemodul korrekt beendet werden kann. Dies ist notwendig, da ein Beenden des Programms während einer laufenden OpenNI-Anfrage in der Praxis dazu führen kann, dass die OpenNI-Bibliothek hängt und sich nicht korrekt herunterfahren lässt. Daher muss das Programmende über ein

[3] Vom Typ `float` nach IEEE754-Standard.

Abfangen der entsprechenden Signale bis zum Ende der momentanen OpenNI-Anfrage verzögert werden.
Im Anschluss werden die OpenNI-Bibliothek und die Kamerahardware initialisiert. Dabei werden Optionen und Einstellungen aus einer OpenNI-Konfigurationsdatei verwendet (siehe Abschnitt 3.1.3). Tritt bei der Initialisierung ein Fehler auf, so wird das Programm mit einer entsprechenden Fehlermeldung beendet. Dies führt automatisch zum Beenden des Gesamtsystems. Häufigste Fehlerursache ist dabei eine fehlerhafte OpenNI- oder NITE-Installation oder eine nicht korrekt angeschlossene Kamera. Wurde beim Start des `openni-provider`-Programms der Parameter `-r` zusammen mit einem Dateinamen angegeben, so wird zudem ein Rekorder eingerichtet. Der Rekorder speichert die Tiefenbilder der Kamera in die angegebene Datei. Von dort aus kann die Aufnahme mit dem zu OpenNI gehörenden Betrachtungsprogramm `NiViewer` zu Testzwecken wiedergegeben werden.
Nach einer erfolgreichen Initialisierung betritt das Programm die Hauptschleife. In der Hauptschleife wartet das Programm zunächst, bis das Restsystem die Bereitschaft zum Datenempfang signalisiert. Dies ist dann der Fall, wenn das Programm vom Restsystem über die Standardeingabe eine leere Zeile[4] empfängt. Im Anschluss werden die Sensordaten aktualisiert. Dabei wird automatisch die OpenNI-Benutzererkennung, Kalibrierung und Skelettanalyse durchgeführt. Über die bei der Initialisierung aufgesetzten Callback-Funktionen wird bei neu erkannten Benutzern eine Positurerkennung gestartet. Dazu muss sich der Benutzer in einer charakteristischen Ψ-Positur gut sichtbar vor der Kamera aufstellen. Diese Körperhaltung ist in Abbildung 3.1 dargestellt. Wird die Ψ-Positur erkannt, wird eine Kalibration durchgeführt. Nach erfolgreicher Kalibration beginnt die Skelettanalyse für den kalibrierten Benutzer. Schlägt die Kalibration hingegen fehl, so kehrt das System zur Positurerkennung zurück. Dieser Ablauf aus Benutzererkennung, Positurerkennung, Kalibration und Skelettanalyse ist ein Standardverfahren bei der Verwendung von OpenNI und ist damit auch in der OpenNI-Dokumentation beschrieben [18].
Läuft die Skelettanalyse für mindestens einen Benutzer, so werden die Positionen der relevanten Gelenke[5] aus dem Skelettmodell extrahiert und in die C-Datenstruktur geschrieben. Da Eingabemodul zur Zeit nur einen aktiven Benutzer unterstützt (siehe auch Abschnitt 4.3.1.2), werden bei mehreren aktiven Benutzer nur die Daten des ersten Benutzers[6] verwendet. Abschließend wird die C-Datenstruktur über die Standardausgabe in Binärform ausgegeben.

[4] Eine leere Zeile entspricht dabei ASCII-kodiert der Eingabe `'\n'`.

[5] Relevante Gelenke sind entsprechend Abschnitt 3.1.1 Kopf, Schulter, Ellbogen und Hände. Dabei ist die Gelenkposition der Hand in OpenNI nicht die Position des Handgelenks, sondern die der Hand selbst.

[6] Die Benutzer sind dabei nach ihrer OpenNI-Benutzer-ID sortiert. Durch Wiederverwendung von IDs entspricht dies nicht immer der chronologischen Reihenfolge, in der die Benutzer das Blickfeld der Kamera betreten haben.

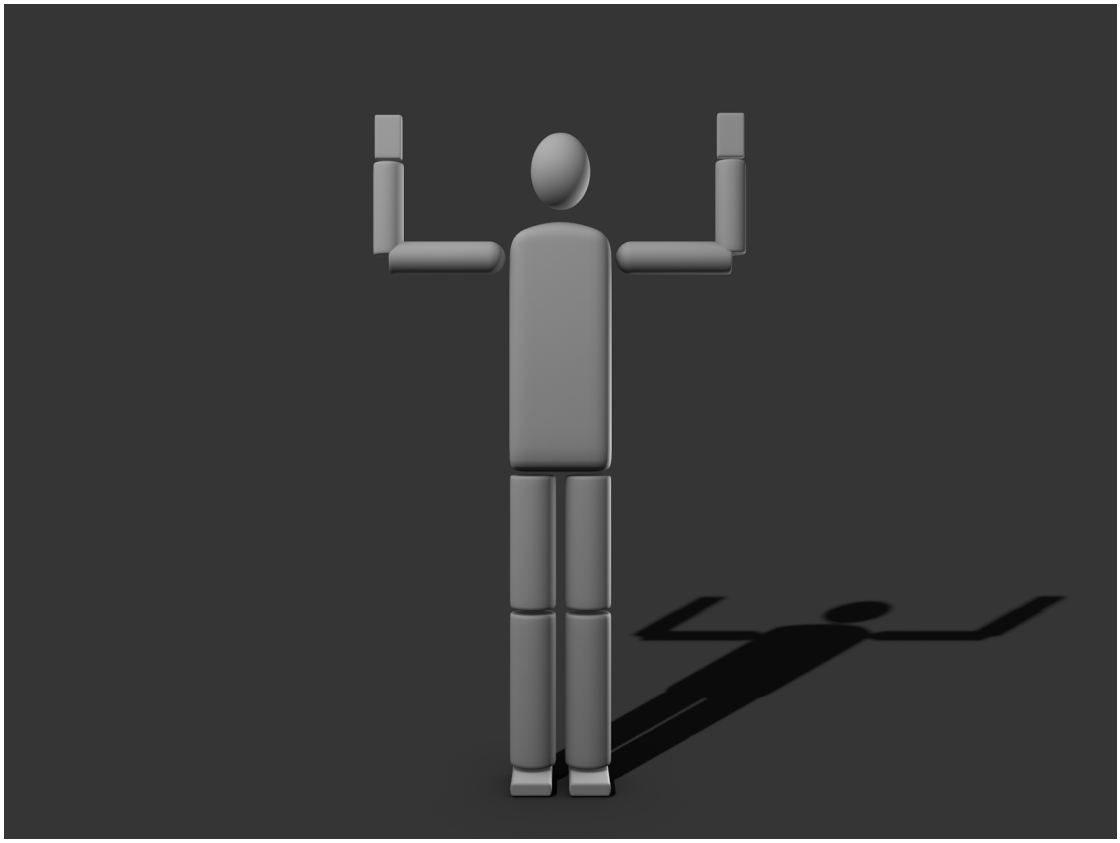

Abbildung 3.1: Grafische Darstellung der idealen Ψ-Körperhaltung aus Sicht der Kamera. Die 90° Winkel zwischen Torso, Ober- und Unterarm erleichtern die Vermessung der Arme bei der Kalibrierung. Zur erfolgreichen Kalibrierung muss diese Positur etwa drei Sekunden eingehalten werden.

3.1.2.1 C-Datenstruktur

Das Format der C-Datenstruktur zum Datenaustausch wird in der C-Headerdatei `ipc.h` festgelegt. Die Datenstruktur `ipcPacket` besteht dabei aus genau einer Unterstruktur `skeletonSubset`, welche die Information über die relevante Teilmenge des Skelettmodells enthält. Das `skeletonSubset` besteht aus einem Aktivitätsflag und sieben `skeletonJoint`-Datenstrukturen mit Gelenkinformationen. Das Aktivitätsflag ist größer als 0, wenn die Skelettanalyse momentan mindestens einen Benutzer erkannt hat. Sonst wird dieses Feld auf 0 gesetzt, wobei der restliche Inhalt des `skeletonSubset` dann undefiniert ist.
Die `skeletonJoint`-Datenstrukturen enthalten die Daten zur Kopfposition sowie den Schulter-, Ellbogen- und Handpositionen beider Arme. Jeder `skeletonJoint` beschreibt dabei genau ein Gelenk und besteht aus einem Aktivitätsflag, den X/Y/Z-Koordinaten des Gelenks als dreielementiges `float`-Array und einem Unsicherheits-

faktor als `float`. Das Aktivitätsflag hat dabei dieselbe Funktion wie das oben beschriebene Aktivitätsflag der `skeletonSubset`-Struktur, nur dass es sich hier auf die Erkennung des entsprechenden Gelenks bezieht. Der Unsicherheitsfaktor wird von der Skelettanalyse bereitgestellt und soll ein Maß für die Qualität der Gelenkinformation darstellen. In der Praxis liefert die NITE-Skelettanalyse hier nur die Werte 0.0 und 1.0. Der Unterschied zum ebenfalls von NITE gelieferten Aktivitätsflag ist, dass der Unsicherheitsfaktor sofort auf 0.0 wechselt, sobald die Skelettanalyse das Gelenk nicht mehr erkennt. Das Aktivitätsflag hingegen ändert sich erst dann auf 0, wenn das Gelenk für eine längere Zeit nicht mehr erkannt wurde.

3.1.3 Konfiguration

Wie in Abschnitt 3.1.2 erwähnt, wird bei der Initialisierung der Tiefenkamera und der OpenNI-Bibliothek eine XML-Konfigurationsdatei verwendet. Die im Rahmen dieses Projektes erstellte Konfigurationsdatei hat den Namen `openni-live.xml`. Diese Konfigurationsdatei legt fest, welche Sensoren zusammen mit welchen Modulen bestimmte Funktionen zur Verfügung stellen. Eine Kette aus Sensoren und nachfolgenden Verarbeitungsmodulen[7] wird dabei als *Production Chain* bezeichnet. Neben Informationen über die Production Nodes enthält die Konfigurationsdatei Informationen über Protokollfunktionen und Lizenzschlüssel, wie zum Beispiel der Lizenzschlüssel für die NITE-Middleware [18].
Durch die Verwendung einer Konfigurationsdatei lassen sich die in der Production Chain verwendeten Module anpassen oder auswechseln, ohne dass das `openni-provider` Programm im Quelltext verändert werden muss. So könnte ein anderer Sensor oder eine andere Middleware verwendet werden. Einschränkung dabei ist, dass die Konfigurationsdatei immer einen *User Node* mit den Fähigkeiten zur Benutzerverfolgung, Gestenerkennung und Skelettanalyse zur Verfügung stellen muss. Ist dies nicht der Fall, so kommt es beim Start des `openni-provider` zu einem Fehler, da die benötigten Fähigkeiten nicht zur Verfügung stehen. Die Anforderung an die Fähigkeiten des User Nodes lassen sich in der Konfigurationsdatei auch über ein `<Capabilities>`-Element formal festlegen, so dass nur User Nodes verwendet werden, welche die Anforderungen erfüllen. Allerdings führte der Einsatz eines solchen Elements zu einem Absturz der verwendeten Version der OpenNI-Bibliothek. Daher wird das `<Capabilities>`-Element in der mitgelieferten `openni-live.xml` nicht eingesetzt. Stattdessen wird der User Node bei der Initialisierung auf seine Fähigkeiten hin überprüft.

[7] In der OpenNI-Dokumentation werden diese Module *Production Nodes* genannt.

3.2 Zeigemodul

Dieser Abschnitt befasst sich mit der Umsetzung des Zeigemoduls, welches entsprechend Abschnitt 2.2.1 aus den Daten des Eingabemoduls (siehe Abschnitt 3.1.2.1) das Objekt ermitteln, auf welches der Benutzer zeigt.

3.2.1 Ansatz

Das Zeigeproblem lässt sich in zwei Teilprobleme zerlegen: Zunächst muss festgestellt werden, ob der Benutzer momentan auf etwas zeigt. Ist dies der Fall muss die Zeigerichtung und, falls vorhanden, das getroffene Objekt ermittelt werden. Indem zunächst geprüft wird ob der Benutzer überhaupt zeigt, wird die Anzahl der falschen Positiven verringert.

3.2.1.1 Zeigeerkennung

Dieser Teil des Moduls soll über die Gelenkpositionen erkennen, ob der Benutzer momentan auf etwas zeigt. Dazu müssen zunächst Merkmale berechnet werden. Über diese Merkmale kann dann mittels eines Klassifikators entschieden werden ob gezeigt wird. Um falsche Positive zu vermeiden, soll nur eindeutiges Zeigen mit einem ausgestrecktem Arm erkannt werden. Zeigen mit angewinkeltem Arm wird hingegen absichtlich nicht erkannt. Diese Beschränkung auf Zeigen mit ausgestrecktem Arm hat den Vorteil, dass die Zeigerichtung leicht zu erkennen ist. Zudem ist für den Benutzer leicht zu verstehen, welche Körperhaltungen als Zeigen erkannt werden.
Als Merkmale bieten sich der Ellbogenwinkel γ und der Höhenwinkel (Altitude) α des ausgestreckten Armes an. Sind die Positionen von Kopf, Schulter, Ellbogen und Hand jeweils über die Vektoren $\mathbf{p_K} = (p_{K,x}, p_{K,y}, p_{K,z})$, $\mathbf{p_S}$, $\mathbf{p_E}$ und $\mathbf{p_H}$ gegeben, so lassen sich die beiden Merkmale wie folgt berechnen:

$$\mathbf{v_O} = \mathbf{p_E} - \mathbf{p_S} \tag{3.1}$$

$$\mathbf{v_U} = \mathbf{p_H} - \mathbf{p_E} \tag{3.2}$$

$$\mathbf{v_Z} = \mathbf{p_H} - \mathbf{p_K} \tag{3.3}$$

$$\alpha = \arcsin\left(\frac{v_{Z,y}}{\|\mathbf{v_Z}\|}\right) \tag{3.4}$$

$$\gamma = \arccos\left(\frac{\mathbf{v_O} \cdot \mathbf{v_U}}{|\mathbf{v_O}|\,|\mathbf{v_U}|}\right) \tag{3.5}$$

$\mathbf{v_O}$ und $\mathbf{v_U}$ sind dabei die Ober- und Unterarmvektoren, $\mathbf{v_Z}$ ist der Zeigevektor vom Kopf zur Hand. γ gibt den Ellbogenwinkel als Winkel zwischen Ober- und Unterarm an. Bei $\gamma = 0$ ist der Arm vollständig ausgestreckt. Der Höhenwinkel α wird aus der y-Komponente des Zeigevektors $\mathbf{v_Z}$ berechnet. Die Bedeutung der Winkel α und γ sind in

Abbildung 3.2 noch einmal grafisch dargestellt. Da die vom Eingabemodul gelieferten Koordinaten relativ zur Kameraposition und -orientierung sind, ist auch der Höhenwinkel α von der Orientierung der Kamera abhängig. Der Winkel α entspricht nur dann dem Höhenwinkel relativ zur physischen Lotrichtung, wenn die y-Achse der Kamera[8] parallel zur Lotrichtung ist. Bei einem ebenen Fußboden ist dies dann der Fall, wenn die Kinect-Kamera parallel zum Fußboden aufgestellt wird.

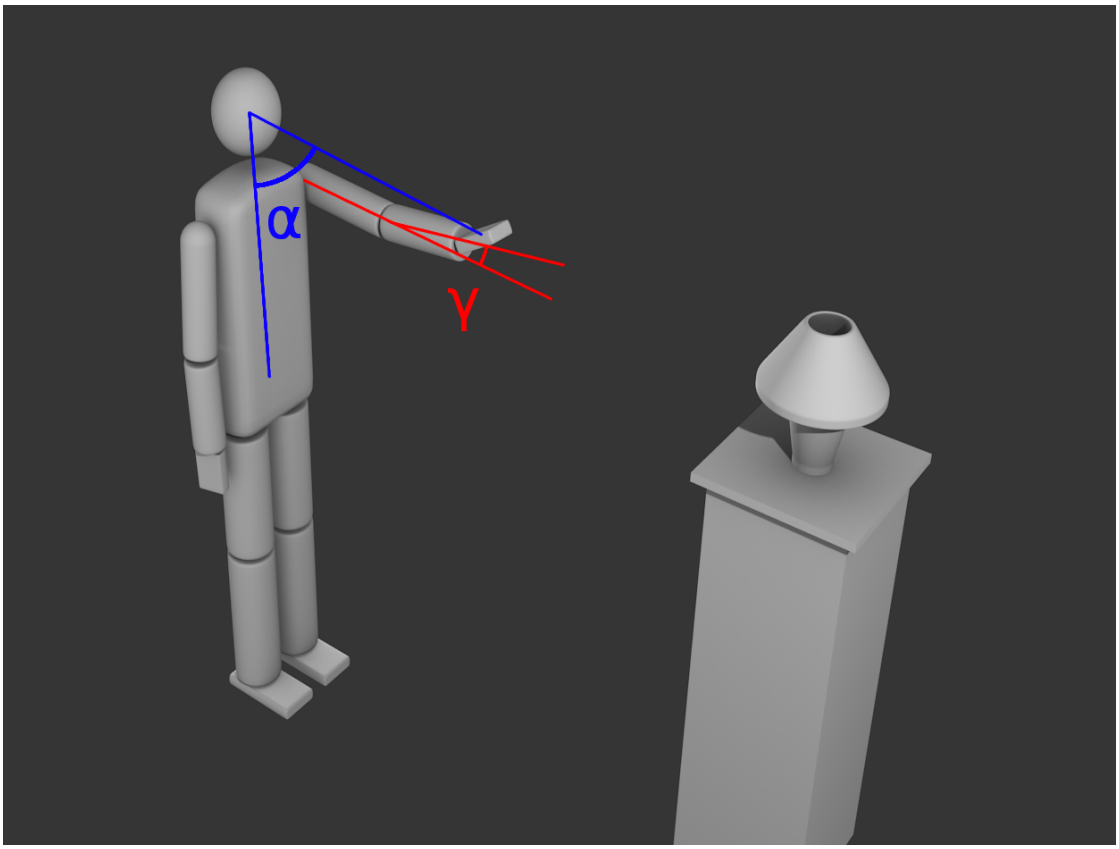

Abbildung 3.2: Grafische Darstellung einer möglichen Körperhaltung beim Zeigen. Das Lot und der Kopf–Hand- bzw. Zeigestrahl sind in Blau, der verlängerte Ober- und Unterarmvektor in Rot eingezeichnet. Zudem sind der Höhenwinkel α und der Ellbogenwinkel γ eingetragen.

Auf Basis des Merkmaltupels (α, γ) kann nun die Armpositur des Benutzers klassifiziert werden. Dazu wird ein Entscheidungsbaum benutzt. Über Trainingsdaten könnte auch ein beliebiger anderer Zwei-Klassen-Klassifikator trainiert werden. Ein Entscheidungsbaum hat jedoch den Vorteil, dass seine Arbeitsweise für den Benutzer einfach

[8] Die y-Achse der Kamera entspricht dabei dem *Up-Vector* der Kamera. Die x-Achse zeigt aus Sicht der Kamera nach rechts, während die z-Achse in Richtung der zunehmenden Tiefe zeigt [22].

zu verstehen ist. Daher können dem Benutzer die Voraussetzungen für eine gültige Zeigepositur leicht verständlich gemacht werden. Die Kriterien für eine Zeigepositur sind dabei

$$Z = \alpha > \alpha_l \wedge \alpha < \alpha_u \wedge \gamma < \gamma_u \tag{3.6}$$

Die Teilbedingung $\gamma < \gamma_u$ legt fest, dass der Arm höchstens bis zu einem Winkel γ_u angewinkelt sein darf. Da bei einem herabhängenden Arm $\gamma \approx 0$ ist, wird zudem eine minimale Zeigehöhe über die Bedingung $\alpha > \alpha_l$ festgelegt. Die Beschränkung der maximalen Zeigehöhe $\alpha < \alpha_u$ ist notwendig, da bei der Rotationsnormierung für die Gestenerkennung (siehe Abschnitt 3.3.1) bei $\alpha \approx \pi$ Singularitäten auftreten können[9]. Daher sollte $\alpha_u < \pi$ gewählt werden. Nur wenn all diese Teilbedingungen und somit Z erfüllt sind, wird die momentane Körperhaltung als eine Zeigepositur klassifiziert.

3.2.1.2 Objekterkennung

Dieser Teil des Zeigemoduls soll erkennen, auf welches Objekt der Benutzer zeigt. Dazu muss auch festgestellt werden, ob der Benutzer überhaupt auf ein gültiges Objekt zeigt. Die Position und Körperhaltung des Benutzers wird vom Eingabemodul geliefert. Daraus soll die Zeigerichtung mathematisch als Zeigestrahl $\mathbf{s} + \lambda\mathbf{v}$ mit $\lambda \geq 0$ modelliert werden. Bei der Beobachtung einiger Versuchspersonen zeigte sich, dass beim Zeigen mit ausgestrecktem Arm die Linie Kopf–Hand eine gute Näherung für die Zeigerichtung $\mathbf{v}$ ist[10]. Mit der Notation aus Abschnitt 3.2.1.1 ergibt sich $\mathbf{v} = \mathbf{p_H} - \mathbf{p_K}$. Da angenommen wird, dass sich das Zielobjekt beim Zeigen mit ausgestrecktem Arm nicht zwischen Kopf und Hand des Benutzers befindet, wird der Stützvektor $\mathbf{s} = \mathbf{p_H}$ gewählt.
Zusätzlich zur Zeigerichtung muss die Position, Orientierung und Form der Objekte bekannt sein. Dies kann entweder über vorgegebene, konstante Objektpositionen oder über eine Messung der Objektpositionen zur Laufzeit erreicht werden. Die Messung der Objektpositionen zur Laufzeit hat den Vorteil, dass sowohl die Objekte als auch die Kamera bewegt werden können. Allerdings ist dieser Ansatz wesentlich komplexer und fehleranfälliger, da die Objekte zur Laufzeit im Bild gefunden, identifiziert und ihre Position und Orientierung gemessen werden müssen. Zudem müssen die Objekte im Blickfeld der Kamera liegen. Dies schränkt den Bereich, in dem sich die Objekte befinden dürfen, sehr stark ein. Diese Probleme treten bei festen Objekt- und Kamerapositionen nicht auf. Objekte und Kamera dürfen dann jedoch nur geringfügig bewegt werden[11]. Die Vorteile

[9] Bei $\alpha = \pi$ zeigt der Benutzer direkt nach oben, wodurch der Azimuthwinkel des Zeigevektors nicht definiert ist.

[10] Systematische Fehler, die durch diese Annahme entstehen, können zum Teil bei der Festlegung der Objektpositionen verringert werden. Dazu wird die Objektposition durch Zeigen festgelegt, siehe Abschnitt 3.2.4.

[11] Geringe Bewegungen lassen sich durch Modellierung von Unsicherheiten in der Objekterkennung ausgleichen.

der geringeren Komplexität und die Möglichkeit zur Interaktion mit Objekten außerhalb des Kamerabildes überwiegen im Rahmen dieser Arbeit die Nachteile der festen Installation der Kamera und die Beschränkung auf stationäre Objekte.
Die Form, Position und Orientierung der Objekte im Raum muss mathematisch modelliert werden. Dazu gibt es eine Reihe von Verfahren, die sich in Genauigkeit, Flexibilität und Aufwand unterscheiden. So bieten sich viele Modellierungsverfahren aus der Computergrafik an. Dazu gehören unter anderem die Objektmodellierung mittels Dreiecksnetzen[12] oder über Voxel. Zur Objekterkennung wird dabei geprüft, ob sich der Zeigestrahl $\mathbf{s} + \lambda\mathbf{v}$ und das Objektmodell schneiden. Diese Verfahren haben den Vorteil, dass die Form der Objekte sehr genau modelliert werden kann. Dies erfordert aber auch einen höheren Aufwand bei der Objektmodellierung und der Objekterkennung. Zudem kann diese genaue Modellierung wegen Unsicherheiten bei der Messung der Zeigerichtung oft nicht ausgenutzt werden.
Als Alternative bietet sich die Modellierung der Objekte über Wahrscheinlichkeitsverteilungen an. Verteilungen sind mathematisch einfach zu handhaben und ermöglichen zudem eine weiche Modellierung, mit der sich Unsicherheiten besser handhaben lassen[13]. Ziel einer solchen Modellierung ist das Berechnen der Wahrscheinlichkeitsfunktion $P(O_i|\mathbf{s}, \mathbf{v})$. Diese Funktion gibt die Wahrscheinlichkeit an, dass das Objekt O_i von einem Zeigestrahl mit Richtung $\mathbf{v}$ und Stützvektor $\mathbf{s}$ getroffen wurde. Über Marginalisierung und Multiplikationssatz ergibt sich

$$P(O_i|\mathbf{s}, \mathbf{v}) = \int_{\mathbf{p}\in\mathbb{R}^3} P(O_i|\mathbf{p})P(\mathbf{p}|\mathbf{s}, \mathbf{v})\mathrm{d}\mathbf{p} \tag{3.7}$$

Dabei ist $P(O_i|\mathbf{p})$ die Wahrscheinlichkeit, dass das Objekt O_i gemeint ist, wenn auf den Punkt $\mathbf{p}$ gezeigt wird. $P(\mathbf{p}|\mathbf{s}, \mathbf{v})$ gibt die Wahrscheinlichkeit an, dass der Zeigestrahl $\mathbf{s}+\lambda\mathbf{v} : \lambda \geq 0$ den Punkt $\mathbf{p}$ als Ziel hat. $P(\mathbf{p}|\mathbf{s}, \mathbf{v})$ lässt sich zum Beispiel durch eine zylinderförmige Dichtefunktion modellieren. Dabei ist die Wahrscheinlichkeit $P(\mathbf{p}|\mathbf{s}, \mathbf{v})$ je höher, desto näher der Punkt $\mathbf{p}$ am Zeigestrahl liegt. Dies könnte auf eine kegelförmige Verteilung erweitert werden, bei der die Spitze des Kegels im Stützvektor $\mathbf{s}$ liegt. Weiter vom Benutzer entfernte Punkte wären dann bei gleicher Wahrscheinlichkeit weiter vom eigentlichen Zeigestrahl entfernt, wodurch die Unsicherheit bei der Bestimmung des Richtungsvektors $\mathbf{v}$ modelliert wird.

[12] Oft auch als Polygonmodellierung bezeichnet, wobei sich jedes Polygon in ein Dreiecksnetz zerlegen lässt.

[13] Eine harte Modellierung entscheidet nur zwischen zwei diskreten Ergebnissen, in diesem Fall „Objekt durch Zeigestrahl getroffen“ und „Objekt nicht getroffen“. Eine weiche Modellierung gibt liefert stattdessen einen kontinuierlichen Wert, der die Wahrscheinlichkeit oder die Qualität des Treffers widerspiegelt. Durch einen Schwellwert lässt sich eine weiche in eine harte Modellierung umwandeln.

Die Funktion $P(O_i|\mathbf{p})$ modelliert die Position und Form des Objekts O_i im Raum. Hierzu soll die multivariate[14] Gauß-Verteilung[15] mit der Dichtefunktion

$$N(\mathbf{p}|\mu_\mathbf{i}, \mathbf{\Sigma_i}) = (2\pi)^{-\frac{k}{2}} \frac{1}{\sqrt{\det(\mathbf{\Sigma_i})}} \exp(-\frac{1}{2}(\mathbf{p} - \mu_\mathbf{i})^T \mathbf{\Sigma_i}^{-1}(\mathbf{p} - \mu_\mathbf{i}) \tag{3.8}$$

verwendet werden. Dabei repräsentieren der Mittelwert $\mu_\mathbf{i}$ und die Kovarianzmatrix $\mathbf{\Sigma_i}$ die Position und Form des Objekts O_i in mathematisch abstrahierter Form. Die multivariate Normalverteilung ist ein Standardwerkzeug in der stochastischen Modellierung. Ihre Dichtefunktion ist einfach zu berechnen und erlaubt die beliebig genaue Näherung einer anderen kontinuierlichen Verteilung über eine gewichtete Summe von Normalverteilungen. Die Funktion $P(O_i|\mathbf{p})$ ergibt sich über den Bayessatz als

$$P(O_i|\mathbf{p}) = P(\mu_\mathbf{i}, \mathbf{\Sigma_i}|\mathbf{p}) = \frac{N(\mathbf{p}|\mu_\mathbf{i}, \mathbf{\Sigma_i}) \cdot P(\mu_\mathbf{i}, \mathbf{\Sigma_i})}{P(\mathbf{p})} \tag{3.9}$$

Hier zeigt sich allerdings das Problem, dass nach Gleichung 3.7 das Integral über die Dichtefunktion $N(\mathbf{p}|\mu_\mathbf{i}, \mathbf{\Sigma_i})$ berechnet werden muss. Dies entspricht der kumulativen Verteilungsfunktion der multivariaten Normalverteilung. Allerdings gibt es für diese Funktion keine analytische Lösung. Es bleibt als Ausweg die näherungsweise Lösung über numerische Verfahren. Numerische Verfahren sollen aber in dieser Arbeit wenn möglich vermieden werden, da nicht garantiert werden kann, dass in der zur Verfügung stehenden Rechenzeit eine ausreichend gute Näherung gefunden werden kann. Aufwendige numerische Verfahren sind insbesondere deshalb problematisch, da diese Berechnung pro Kamerabild und Objekt einmal durchgeführt werden muss. Bei Kameraaufnahmen mit 30–60Hz und mehreren Objekten kann dies schnell zu einem hohen Rechenaufwand führen, wobei diese Berechnungen zusammen mit den Berechnungen der anderen Module laut den Zielen aus Abschnitt 1.1 auf einem handelsüblichen Desktoprechner laufen müssen.
Zwar kann möglicherweise eine ausreichend genaue und schnelle numerische Lösung gefunden werden. Um die damit verbundenen Planungsunsicherheit bezüglich Geschwindigkeit und Genauigkeit zu vermeiden, wurde dieser Ansatz jedoch nicht weiter verfolgt. Statt der stochastischen Modellierung aus Gleichung 3.7 wurde eine Reihe von alternativen Ansätzen überprüft. Diese sollen das Problem der Objekterkennung über multivariate Normalverteilungen auf anderen Wegen analytisch lösen.

[14] Da die Objekte in karthesischen Koordinaten modelliert werden, ist die Verteilung immer dreidimensional. In den nachfolgenden Gleichungen gilt daher $k = 3$.

[15] Die multivariate Gauß-Verteilung ist auch als multivariate Normalverteilung bekannt.

3.2.1.3 Integralverfahren

Beim ersten alternativen Ansatz werden die Verteilungen über den Zeigestrahl integriert[16], wodurch sich die Funktion

$$S(O_i, \mathbf{s}, \mathbf{v}) = \int_{\eta_{near}}^{\eta_{far}} N(\mathbf{s} + \lambda \frac{\mathbf{v}}{\|\mathbf{v}\|} | \mu_\mathbf{i}, \mathbf{\Sigma_i}) \mathrm{d}\lambda \tag{3.10}$$

ergibt. Das Ergebnis dieser Berechnung ist keine Wahrscheinlichkeit. Stattdessen ergibt sich eine Funktion der Treffergüte $S(O_i, \mathbf{s}, \mathbf{v})$. Je höher $S(O_i, \mathbf{s}, \mathbf{v})$ ist, desto besser trifft der Zeigestrahl $\mathbf{s} + \lambda\mathbf{v}$ die Verteilung des Objekts O_i mit den Parametern $\mu_\mathbf{i}, \mathbf{\Sigma_i}$. Anhand eines Schwellwerts s_{min} kann dann bestimmt werden, ob ein Objekt O_i getroffen wurde. Dies ist genau dann der Fall, wenn $S(O_i, \mathbf{s}, \mathbf{v}) > s_{min}$ erfüllt ist. Trifft dies für mehrere Objekte zu, so wird das Objekt ausgewählt, welches die höchste Treffergüte aufweist. Die Grenzen η_{near} und η_{far} geben die minimale und maximale Distanz vom Stützvektor an, die bei der Berechnung von $S(O_i, \mathbf{s}, \mathbf{v})$ einbezogen werden. Hier kann $\eta_{near} = 0$ und $\eta_{far} = \infty$ gewählt werden, um den gesamten Strahl abzudecken.
Um die Berechnung von $S(O_i, \mathbf{s}, \mathbf{v})$ zu erleichtern, wird diese in zwei Schritte zerlegt. Zunächst wird das gesamte Koordinatensystem so transformiert, dass sich der transformierte Zeigestrahl zu $\mathbf{s}' = R(\mathbf{s} - \mathbf{t}) = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ und $\mathbf{v}' = \mathbf{R}\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$ ergibt. Im transformierten Koordinatensystem befindet sich der Stützvektor $\mathbf{s}'$ des Zeigestrahls also im Ursprung des Koordinatensystems, während der transformierte Richtungsvektor $\mathbf{v}'$ parallel zur x-Achse ist. Dabei müssen die Rotationsmatrix $\mathbf{R}$ und die Translation $\mathbf{t}$ nur einmal pro Zeigestrahl vorberechnet werden. Mit den transformierten Parametern $\mu_\mathbf{i}' = \mathbf{R}(\mu_\mathbf{i} - \mathbf{t})$ und $\mathbf{\Sigma_i}' = \mathbf{R}\mathbf{\Sigma_i}\mathbf{R^T}$ ergibt sich dann die vereinfachte Form

$$S(O_i, \mathbf{s}, \mathbf{v}) = \int_{\eta_{near}}^{\eta_{far}} N\left(\begin{pmatrix} \lambda & 0 & 0 \end{pmatrix}^T | \mu_\mathbf{i}', \mathbf{\Sigma_i}' \right) \mathrm{d}\lambda \tag{3.11}$$

Gemäß der Anforderungen ergibt sich für die Translation $t = s$. Die Rotationsmatrix, welche $\mathbf{R}\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$ erfüllt, wird über die Zerlegung von $\mathbf{v}$ in den Azimuthwinkel α und den Altitudenwinkel β (also Polarkoordinaten) berechnet:

$$\alpha = \arctan2(v_z, v_x) \tag{3.12}$$

$$\beta = \arcsin\left(\frac{v_y}{\|\mathbf{v}\|} \right) \tag{3.13}$$

[16] Die Funktion $N(\mathbf{p}|\mu, \mathbf{\Sigma})$ ist dabei weiterhin die Dichtefunktion der dreidimensionalen Multivariaten Normalverteilung, wie sie in Gleichung 3.8 angegeben ist.

Aus diesen Winkeln lassen sich dann zwei Rotationsmatrizen $\mathbf{R_y}$ und $\mathbf{R_z}$ für Rotationen um die y- und z-Achse konstruieren:

$$\mathbf{R_y} = \begin{pmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix} \tag{3.14}$$

$$\mathbf{R_z} = \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.15}$$

Diese beschreiben die Rotation die entlang der y- und z-Achse ausgeführt wird, um den Basisvektor $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$ parallel zu v zu rotieren. Das Produkt der beiden Rotationsmatrizen ist somit $\mathbf{R_{yz}} = \mathbf{R_y}\mathbf{R_z}$ mit $\mathbf{R_{yz}} \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T = \mathbf{v}$. Gesucht ist aber $\mathbf{R}$, also die Inverse von $\mathbf{R_{yz}}$. Da $\mathbf{R_{yz}}$ als Produkt zweier Rotationsmatrizen wieder eine Rotationsmatrix und somit eine Orthogonalmatrix ist, entspricht die inverse Matrix $\mathbf{R}$ der Transponierten von $\mathbf{R_{yz}}$. Es gilt also $\mathbf{R} = \mathbf{R_{yz}}^{-1} = (\mathbf{R_{yz}})^T$. Mit diesen Transformationsparametern $\mathbf{t}$ und $\mathbf{R}$ kann nun der Scoringwert entsprechend Gleichung 3.10 berechnet und das Objekt ausgewählt werden.
Da $\mathbf{R_{yz}}$ das Produkt aus $\mathbf{R_y}$ und $\mathbf{R_z}$ ist, gilt für die Determinante $\det(\mathbf{R_{yz}}) = \det(\mathbf{R_y}) \det(\mathbf{R_z})$. Mit den Determinanten

$$\det(\mathbf{R_y}) = \cos(\alpha) \cdot \cos(\alpha) + \sin(\alpha) \cdot \sin(\alpha) = 1 \tag{3.16}$$

$$\det(\mathbf{R_z}) = \cos(\beta) \cdot \cos(\beta) + \sin(\beta) \cdot \sin(\beta) = 1 \tag{3.17}$$

ergibt sich somit $\det(\mathbf{R_{yz}}) = 1$. Die transformierte Kovarianzmatrix ist, wie bereits erwähnt, $\mathbf{\Sigma_i}' = \mathbf{R}\mathbf{\Sigma_i}\mathbf{R}^T$. Da $\det(\mathbf{R}) = \frac{1}{\det(\mathbf{R}^{-1})} = \frac{1}{\det(\mathbf{R_{yz}})} = 1$ und $\det(\mathbf{AB}) = \det(\mathbf{A}) \det(\mathbf{B})$ gelten, sind die Determinanten der transformierten Kovarianzmatrizen unverändert: $\det(\mathbf{\Sigma_i}') = \det(\mathbf{\Sigma_i})$. Da sich die Determinanten der Kovarinzmatrizen $\det(\mathbf{\Sigma_i})$ durch die Transformation also nicht ändern, können sie für die Verwendung in Gleichung 3.8 vorberechnet werden. Eine Neuberechnung bei einer Änderung von $\mathbf{R}$ ist nicht notwendig.

3.2.1.4 Projektionsverfahren

Der zweite alternative Ansatz baut auf der Idee auf, dass der Benutzer nicht auf einen Punkt im Raum sondern auf einen Punkt in seinem Gesichtsfeld, also in einer Projektion des Raumes, zeigt. Daher werden bei diesem Ansatz die Objektparameter μ_i und $\mathbf{\Sigma_i}$ parallel zum Zeigestrahl in eine Ebene projiziert. Da die Projektion parallel zum Zeigestrahl erfolgt, wird der Zeigestrahl selbst dabei auf einen Punkt projiziert[17]. Die

[17] Dies ist auch bei einer nicht parallelen Projektion der Fall, solange die Symmetrieachse der Projektion identisch mit dem Zeigestrahl ist. Entlang der Symmetrieachse zeigen die Parallel- und Zentralpro-

projizierten Objektverteilungen werden dann an diesem Punkt evaluiert und die getroffenen Objekte bestimmt. Um die Berechnung zu erleichtern, wird das Koordinatensystem wie in Abschnitt 3.2.1.3 beschrieben transformiert. Im Anschluss wird im transformierten Koordinatensystem mit einer konstanten Projektionsmatrix entlang der x-Achse in die y-z-Ebene projiziert. Die Projektionsmatrix für die Parallelprojektion lautet

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.18}$$

Die projizierten Objektparameter $\mu_\mathbf{i}^\mathbf{P}$ und $\mathbf{\Sigma}_\mathbf{i}^\mathbf{P}$ sind

$$\mu_\mathbf{i}^\mathbf{P} = \mathbf{P}\mu_\mathbf{i}{}' = \mathbf{P}\left(\mathbf{R}\left(\mu_\mathbf{i} - \mathbf{t}\right)\right) \tag{3.19}$$

$$\mathbf{\Sigma}_\mathbf{i}^\mathbf{P} = \mathbf{P}\mathbf{\Sigma}_\mathbf{i}{}' \cdot \mathbf{P}^T = \mathbf{P}(\mathbf{R}\mathbf{\Sigma}_\mathbf{i}\mathbf{R}^T)\mathbf{P}^T \tag{3.20}$$

Im transformierten Koordinatensystem hat der Zeigestrahl per Definition immer den Stützvektor $\mathbf{s}' = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ und die Richtung $\mathbf{v}' = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$. Daher wird der Zeigestrahl immer auf den Punkt $\mathbf{P}(\mathbf{s}' + \lambda\mathbf{v}') = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ projiziert. Die projizierten, zweidimensionalen Verteilungen werden also immer am Ursprung des projizierten Koordinatensystems evaluiert. Da nach der Projektion in die y-z-Ebene alle x-Komponenten 0 sind, können alle projizierten Vektoren und Matrizen durch Weglassen der ersten Spalte und Zeile in den Raum $\mathbb{R}^2$ überführt werden, was die Berechnung beschleunigt.
Zur Berechnung der Objektwahrscheinlichkeiten muss jetzt nur noch die Dichtefunktion $N\left(\begin{pmatrix} 0 & 0 \end{pmatrix}^T | \mu_\mathbf{i}^\mathbf{P}, \mathbf{\Sigma}_\mathbf{i}^\mathbf{P}\right)$ (entsprechend Gleichung 3.8) berechnet werden. Dabei zeigt sich aber, dass die Verwendung der Dichtefunktion problematisch sein kann. Der maximale Wert der Dichtefunktion der multivariaten Normalverteilung ist $\max_\mathbf{p} N(\mathbf{p}|\mu, \mathbf{\Sigma}) = N(\mu|\mu, \mathbf{\Sigma}) = (2\pi)^{-\frac{k}{2}} \frac{1}{\sqrt{\det(\mathbf{\Sigma})}}$. Für ausreichend große Werte von $\det(\mathbf{\Sigma})$ ist es möglich, dass der Schwellwert der Objekterkennung s_{min} an keinem Punkt der Verteilung mehr erreicht wird. Objekte mit ausreichend großen Ausmaßen können somit nicht mehr getroffen werden. Um dies zu vermeiden, wird der Vorfaktor $(2\pi)^{-\frac{k}{2}} \frac{1}{\sqrt{\det(\mathbf{\Sigma})}}$ aus $N(\mathbf{p}|\mu, \mathbf{\Sigma})$ entfernt. Dadurch ergibt sich eine exponentielle radiale Basisfunktion $B(\mathbf{p}|\mu, \mathbf{\Sigma})$, welche die Mahalanobisdistanz zur Berechnung des Abstands zum Objektmittelpunkt verwendet. Die Gleichung der Funktion lautet

$$B(\mathbf{p}, \mu, \mathbf{\Sigma}) = \exp(-\frac{1}{2}(\mathbf{p} - \mu)^T \mathbf{\Sigma}^{-1}(\mathbf{p} - \mu)) \tag{3.21}$$

jektion das selbe Verhalten. Eine Zentralprojektion entlang der x-Achse, bei der das Projektionszentrum auf der x-Achse liegt, hätte also das selbe Ergebnis wie die Parallelprojektion, da nur die Projektion des Zeigestrahls (und somit der Symmetrieachse) von Interesse ist. Die Parallelprojektion wurde gewählt, da sie einfacher zu berechnen ist.

Die Funktion $B(\mathbf{p}, \mu, \mathbf{\Sigma})$ zeigt ein einfacher zu handhabendes Verhalten, da der Schwellwert s_{min} nun die maximale Distanz zwischen $\mathbf{p}$ und μ ausdrückt, bei dem das Objekt getroffen wird. Wird zum Beispiel $s_{min} = \exp(-\frac{1}{2})$ gewählt, so ist die Bedingung für einen Treffer $B(\mathbf{p}|\mu, \mathbf{\Sigma}) < s_{min}$ genau dann erfüllt, wenn der Abstand zwischen $\mathbf{p}$ und dem Objektmittelpunkt μ unter der Mahalanobisdistanz mit der Kovarianzmatrix $\mathbf{\Sigma}$ kleiner als 1 ist. Bei einer kugelförmigen Verteilung mit

$$\mathbf{\Sigma} = k^2 \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.22}$$

ist dies genau dann der Fall, wenn $\|\mu - \mathbf{p}\| < k$ ist. Diese Eigenschaft wird bei der empirischen Ermittelung der Objektgrößen in Abschnitt 3.2.4 wichtig.
Zusammengefasst besteht das Projektionsverfahren also aus drei Schritten: Über die Rotationmatrix $\mathbf{R}$ und die Verschiebung um $\mathbf{t}$ wird das Koordinatensystem transformiert, so dass der Zeigestrahl im Ursprung beginnt und parallel zur x-Achse verläuft. Dann werden die Objekte parallel zur x-Achse auf die y-z-Ebene projiziert. Dabei dienen die bereits in Abschnitt 3.2.1.3 verwendeten Werte η_{near} und η_{far} als Grenzen der Projektion. Objekte, deren Mittelpunkte $\mu_\mathbf{i}$ näher als η_{near} oder weiter als η_{far} von der Projektionsebene entfernt liegen, werden eliminiert und nicht weiter betrachtet. Ein Objekt O_i muss also die Bedingung $\eta_{near} < \mu'_{i,x} < \eta_{far}$ erfüllen. Hierbei können im einfachsten Fall wieder $\eta_{near} = 0$ und $\eta_{far} = \infty$ gewählt werden. Bei $\eta_{near} = 0$ und $\eta_{far} = \infty$ werden nur Objekte eliminiert, deren Mittelpunkte hinter der Projektionsebene (und somit hinter der Hand des Benutzers) liegen.
Abschließend wird für die projizierten Objekte die Funktion

$$S_i = B\left(\begin{pmatrix}0 & 0\end{pmatrix}^T, \mu_\mathbf{i}^\mathbf{P}, \mathbf{\Sigma}_\mathbf{i}^\mathbf{P}\right) \tag{3.23}$$

berechnet. Mittels des Schwellwerts s_{min} wird bestimmt, ob ein Objekt getroffen wird. Werden mehrere Objekte getroffen, wird das Objekt mit dem höchsten Wert S_i ausgewählt. Da diese Projektionsmethode einfacher zu berechnen und leichter zu handhaben ist als die Integralmethode aus Abschnitt 3.2.1.3, wird sie als Standardmethode für die Objekterkennung verwendet.

3.2.2 Implementation

Die in Abschnitt 3.2.1 beschriebenen Verfahren zur Objekterkennung wurden in den Python-Modulen `objectSelection.py` und `pointing.py` implementiert. Das Modul `objectSelection.py` stellt die in Abschnitt 3.2.1.2 beschriebenen Integral- und Projektionsverfahren als `gaussianModel`-Klasse bereit. Bei der Instanziierung der Klasse kann über den Parameter `useIntegral` zwischen Integral- und Projektionsverfahren gewählt werden. Objektverteilungen werden dem `gaussianModel` über

die Methode `addGaussian` hinzugefügt. Die Methode `evalModel` evaluiert das Modell für einen gegebenen Zeigestrahl und gibt eine Liste der Treffergüten zurück.
Das Modul `pointing.py` nutzt die Funktionalität des `objectSelection.py`-Moduls, um das Zeigemodul in der Klasse `pointHandler` zu implementieren. Bei der Instanziierung dieser Klasse werden die Modellparameter als `PointingConfig`-Instanz (aus dem Modul `pointingConfig.py`) übergeben. Wie in Abschnitt 3.2.3 beschrieben, kann die `PointingConfig` aus einer XML-Konfigurationsdatei geladen werden. Die Methode `handlePointing` der Klasse `pointHandler` nimmt die Gelenkpositionen von Kopf, Schulter, Ellbogen und Hand als Instanzen der Klasse `skelJoint` entgegen und führt die Zeigeerkennung aus Abschnitt 3.2.1.1 und die Objekterkennung aus Abschnitt 3.2.1.2 durch. Falls mindestens ein Objekt getroffen wurde, wird der Index i des am besten getroffenen Objekts O_i zurückgegeben. Befand sich der Benutzer nicht in einer Zeigepositur oder wurde kein Objekt getroffen, wird `None` zurück gegeben.

3.2.3 Konfiguration

Die Initialisierung des `pointingHandler` aus Abschnitt 3.2.2 kann auf Basis einer XML-Datei erfolgen. Dazu wird bei der Instanziierung der Klasse `PointingConfig` aus dem Modul `pointingConfig.py` der Name einer XML-Konfigurationsdatei übergeben. Die aus der Konfigurationsdatei erstellte `PointingConfig`-Instanz kann dann dem Konstruktor der `pointHandler`-Klasse übergeben werden.
Das Format dieser Konfigurationsdatei ist als XML-Schema in der Datei `pointing.xsd` spezifiziert. Wurzel der Konfigurationsdatei ist das `<model>`-Element. Dessen Attribut `evaluate` kann die Werte `project` oder `integrate` annehmen und legt fest, ob die Integralmethode aus Abschnitt 3.2.1.3 oder die Projektionsmethode aus 3.2.1.4 verwendet wird. Das Attribut `threshold` ist eine nichtnegative Gleitkommazahl, die den in Abschnitt 3.2.1.2 ff. verwendeten Schwellwert s_{min} angibt. Die Kinder des `<model>`-Elements sind ein `<constraints>`- und ein `<objects>`-Element.
Das `<constraints>`-Element besteht aus einem `<altitude>`, `<elbow>` und `<distance>`-Element. Jedes dieser drei Elemente ist optional und hat jeweils ein optionales `<min>`- und `<max>`-Kindelement, welche die Ober- und Untergrenze eines bestimmten Wertes als Gleitkommazahl enthalten. Die Elemente `<altitude>` und `<elbow>` geben somit die Unter- und Obergrenze α_l, α_u des Höhenwinkel α und die Obergrenze γ_u des Ellbogenwinkels γ aus Abschnitt 3.2.1.1 an. Das `<min>`-Element des `<elbow>`-Elements hat keine Verwendung. Das `<distance>`-Element gibt die Entfernungsbeschränkungen η_{near} und η_{far} aus 3.2.1.2 an. Die Winkelangaben zu α und γ haben die Einheit Grad, die Distanzangaben für η sind in Millimetern angegeben. Wird eine dieser Beschränkungen nicht angegeben, so wird beim Laden der Konfigurationsdatei $\alpha_l = -\infty$, $\alpha_u = \infty$, $\gamma_u = \infty$, $\eta_{near} = 0$ und $\eta_{far} = \infty$ angenommen.

Die im System verwendeten Objekte werden im <objects>-Element kodiert. Dieses enthält beliebig viele <object>-Kindelemente, wobei jedes <object>-Element genau ein Objekt kodiert. Das <id>-Kindelement eines <object> legt die Objekt-ID als Zeichenkette fest. Jede ID darf dabei nur für genau ein Objekt verwendet werden. Die Objekt-ID ist der Wert, der bei einem Treffer an das Aktionsmodell aus Abschnitt 3.5 übergeben wird. Das optionale <label>-Kindelement legt einen menschenlesbaren Namen für das Objekt fest, welcher für Testzwecke verwendet wird. Das <distributions>-Kindelement eines <object> legt dessen Verteilung im Raum fest. Jedes <distributions>-Element besteht aus beliebig vielen <distribution>-Kindelementen, die jeweils eine Objektverteilung festlegen. Das type-Attribut eines <distribution>-Elements bestimmt die Art der Verteilung, wobei der einzige im Rahmen dieser Arbeit erlaubte Wert gaussian ist. Jedes <distribution>-Element beinhaltet zudem ein <mean>- und <covariance>-Element. Diese Elemente kodieren den Objektmittelpunkt μ und die Kovarianzmatrix $\mathbf{\Sigma}$ im $\mathbb{R}^3$-Vektorraum als durch Leerzeichen getrennte drei- beziehungsweise neunelementige Liste. Die Matrix Σ wird dabei Zeile für Zeile, also in der Form <covariance>$\Sigma_{1,1}$ $\Sigma_{1,2}$ $\Sigma_{1,3}$ $\Sigma_{2,1}$ $\Sigma_{2,2}$ $\Sigma_{2,3}$ $\Sigma_{3,1}$ $\Sigma_{3,2}$ $\Sigma_{3,3}$</covariance>, kodiert. Umfasst das <distributions>-Element eines <object> mehrere <distribution>-Elemente, so werden diese getrennt behandelt. Jedes <distribution>-Element wird als eigene Verteilung mit den Parametern (μ_i, Σ_i) betrachtet, wobei die mit den Verteilungen assoziierte Objekt-ID und das Label identisch sind.

3.2.4 Training

Um das Zeigemodul zu trainieren soll es möglich sein, Objektpositionen und -größen durch Zeigen festzulegen. Dies ist ein schneller und einfacher Weg, dem System neue Objekte hinzuzufügen. Diese Funktionalität wurde in dem Python-Programm pointingTrainer.py implementiert, welches den Benutzer durch Textausgabe durch den Vorgang leitet. Die Objektpositionen werden auf Basis von Zeigestrahlen berechnet, wie sie in Abschnitt 3.2.1.2 beschrieben sind. Einige systematische Fehler bei der Verwendung der Zeigestrahlen zur Objekterkennung können dabei ausgeglichen werden, da der Fehler eventuell bereits bei der Definition der Objekte mit einbezogen wird. Das Festlegen der Objekte über Zeigen lässt sich in zwei Teilaufgaben zerlegen: Die Bestimmung des Objektmittelpunkts μ und die Bestimmung der Objektgröße in Form der Kovarianzmatrix $\mathbf{\Sigma}$.

3.2.4.1 Bestimmung der Objektposition

Der Objektmittelpunkt μ wird über Triangulation aus mehreren Zeigestrahlen berechnet. Die einzelnen Strahlen werden ermittelt, indem der Benutzer aus möglichst verschie-

denen Richtungen mit ausgestrecktem Arm auf das Objekt zeigt und dabei schnipst. Das Schnipsen wird über die Schnipserkennung aus Abschnitt 3.4 erkannt. Bei jedem Schnipsen wird der aktuelle Zeigestrahl gespeichert. Nun wird über Triangulation für jedes mögliche Paar von Zeigestrahlen $((\mathbf{s_i}, \mathbf{v_i}), (\mathbf{s_j}, \mathbf{v_j}))\, i \neq j$ der Punkt $\mathbf{p_{i,j}}$ berechnet. Die meisten Strahlenpaare schneiden sich jedoch nicht exakt, sondern sind zueinander windschief. Daher kann kein Schnittpunkt berechnet werden. Stattdessen soll der quadratische Abstand zwischen dem Punkt $\mathbf{p_{i,j}}$ und den Zeigestrahlen $(\mathbf{s_i}, \mathbf{v_i})$ und $(\mathbf{s_j}, \mathbf{v_j})$ minimieren werden. Der gewählte Punkt ist also

$$\mathbf{p_{i,j}} = \underset{\mathbf{p}\in\mathbb{R}^3}{\operatorname{argmin}} \left(\left(\min_{\lambda} \| \mathbf{p} - (\mathbf{s_i} + \lambda \mathbf{v_i}) \| \right)^2 + \left(\min_{\eta} \| \mathbf{p} - (\mathbf{s_j} + \eta \mathbf{v_j}) \| \right)^2 \right) \tag{3.24}$$

Dazu äquivalent ist es, die kürzeste Strecke zwischen den beiden Strahlen ermittelt und den Punkt $\mathbf{p_{i,j}}$ in die Mitte dieser Strecke zu legen. Um die kürzeste Strecke zwischen zwei Zeigestrahlen zu ermitteln, wird ein lineares Gleichungssystem aufgestellt. Da die kürzeste Strecke zwischen einem Punkt und einer Geraden immer senkrecht auf der Geraden steht, sollen zwei Punkte $\mathbf{p_i} = \mathbf{s_i} + \lambda \mathbf{v_i}$ und $\mathbf{p_j} = \mathbf{s_j} + \eta \mathbf{v_j}$ gefunden werden, so dass der Vektor $\mathbf{p_i} - \mathbf{p_j}$ und somit die gesuchte Strecke senkrecht auf den Strahlrichtungen $\mathbf{v_i}$ und $\mathbf{v_j}$ steht. Die Strecke zwischen $\mathbf{p_i}$ und $\mathbf{p_j}$ ist dann die kürzeste Strecke zwischen den beiden Strahlen. Es ergibt sich folgendes lineares Gleichungssystem:

$$((\mathbf{s_i} + \lambda \mathbf{v_i}) - (\mathbf{s_j} + \eta \mathbf{v_j}))^T \mathbf{v_i} = 0 \tag{3.25}$$

$$((\mathbf{s_i} + \lambda \mathbf{v_i}) - (\mathbf{s_j} + \eta \mathbf{v_j}))^T \mathbf{v_j} = 0 \tag{3.26}$$

Dieses Gleichungssystem wird in Koeffizientenform umgeformt. Dabei ergibt sich das System

$$(\mathbf{v_i}^T \mathbf{v_i})\lambda - (\mathbf{v_i}^T \mathbf{v_j})\eta = (\mathbf{s_j} - \mathbf{s_i})^T \mathbf{v_i} \tag{3.27}$$

$$(\mathbf{v_i}^T \mathbf{v_j})\lambda - (\mathbf{v_j}^T \mathbf{v_j})\eta = (\mathbf{s_j} - \mathbf{s_i})^T \mathbf{v_j} \tag{3.28}$$

Auf Basis der erweiterten Koeffizientenmatrix kann dieses Gleichungssystem nun maschinell gelöst werden[18]. Ergebnis sind λ_s und η_s als Werte für λ und η, die das Gleichungssystem lösen. Ist $\lambda_s < 0$ oder $\eta_s < 0$, so liegt $\mathbf{p_i}$ beziehungsweise $\mathbf{p_j}$ nicht auf dem Zeigestrahl. Dies bedeutet, dass die beiden Zeigestrahlen auseinander laufen. In diesem Fall liegt ein Fehler bei der Aufnahme der Zeigestrahlen vor und das momentan betrachtete Strahlenpaar wird nicht weiter berücksichtigt. Das Strahlenpaar wird auch verworfen, wenn die beiden Strahlen parallel sind und das Gleichungssystem somit unendlich viele Lösungen hat. Andernfalls ergibt sich der Punkt

$$\mathbf{p_{i,j}} = \frac{(\mathbf{s_i} + \lambda_s \mathbf{v_i}) + (\mathbf{s_j} + \eta_s \mathbf{v_j})}{2} \tag{3.29}$$

[18] Dabei kommt die `solve`-Funktion des `numpy.linalg`-Pakets zum Einsatz. Diese Funktion verwendet die LU-Faktorisierung zur Lösung des Gleichungssystems [5].

Aus allen berechneten Punkten $\mathbf{p_{i,j}}$ muss zum Schluss ein einzelner Objektmittelpunkt μ gebildet werden. Dazu wird der Median über alle Punkte $\mathbf{p_{i,j}}$ berechnet. Die Berechnung des Medians erfolgt separat entlang jeder Achse der Vektoren[19]. Durch die Verwendung des Medians haben weit außen liegende Punkte, wie sie zum Beispiel durch falsch gemessene Zeigestrahlen entstehen können, weniger Einfluss auf das Ergebnis μ.

3.2.4.2 Bestimmung der Objektausdehnung

Um $\mathbf{\Sigma}$ zu ermitteln, soll der Benutzer nach der Festlegung von μ die Größe des Objekts mit beiden Händen zeigen können. Dazu hält der Benutzer beide Hände auf Brusthöhe vor dem Körper. Ob der Benutzer die Hände korrekt hält, wird wie bei der Zeigeerkennung in Abschnitt 3.2.1.1 über ein Entscheidungsbaum bestimmt. Der Entscheidungsbaum ermittelt über die Höhenwinkel α_l und α_r der Strecke Schulter–Hand und die Abstände zwischen Hand und Schulter d_l und d_r, ob der Benutzer gerade eine Größe zeigt. Die korrekte Haltung zum Zeigen der Größe ist in Abbildung 3.3 dargestellt. Sind die Punkte $\mathbf{p_{S,l}}$ und $\mathbf{p_{S,r}}$ die Position der linken und rechten Schulter sowie die Punkte $\mathbf{p_{H,l}}$ und $\mathbf{p_{H,r}}$ die Position der linken und rechten Hand, dann gilt

$$d_l = \|\mathbf{p_{H,l}} - \mathbf{p_{S,l}}\| \tag{3.30}$$

$$d_r = \|\mathbf{p_{H,r}} - \mathbf{p_{S,r}}\| \tag{3.31}$$

$$\alpha_l = \arcsin\left(\frac{(\mathbf{p_{H,l}} - \mathbf{p_{S,l}})_y}{d_l}\right) \tag{3.32}$$

$$\alpha_r = \arcsin\left(\frac{(\mathbf{p_{H,r}} - \mathbf{p_{S,r}})_y}{d_r}\right) \tag{3.33}$$

Sind diese Größen bekannt, wird die Körperhaltung des Benutzers über den Ausdruck

$$G_\alpha = |\alpha_l| < \alpha_{max} \wedge |\alpha_r| < \alpha_{max} \wedge |\alpha_r - \alpha_l| < \alpha_\delta \tag{3.34}$$

$$G_d = d_l > d_{min} \wedge d_r > d_{min} \wedge |d_r - d_l| < d_\delta \tag{3.35}$$

$$G = G_\alpha \wedge G_d \tag{3.36}$$

überprüft.
Die Hände des Benutzers müssen sich also in einer gewissen Nähe zur Schulterhöhe befinden ($|\alpha_l| < \alpha_{max} \wedge |\alpha_r| < \alpha_{max}$) und dürfen sich in ihrem Höhenwinkel maximal um α_δ um unterschieden. Gleichzeitig dürfen die Hände nicht näher als d_{min} an den Schultern anliegen und sich in ihrem Abstand von der jeweiligen Schulter nicht um mehr als d_δ unterscheiden. In der Praxis stellten sich $\alpha_{max} = 55°$, $\alpha_\delta = 10°$, $d_{min} = 200$mm und

[19] Eine bessere Berechnung des Punktes μ aus den Punkten $\mathbf{p_{i,j}}$ ist über die Lösung des Fermat-Weber-Problems möglich [9]. Dieser Ansatz ist jedoch wesentlich komplexer und wurde daher aus zeitlichen Gründen hier nicht verwendet.

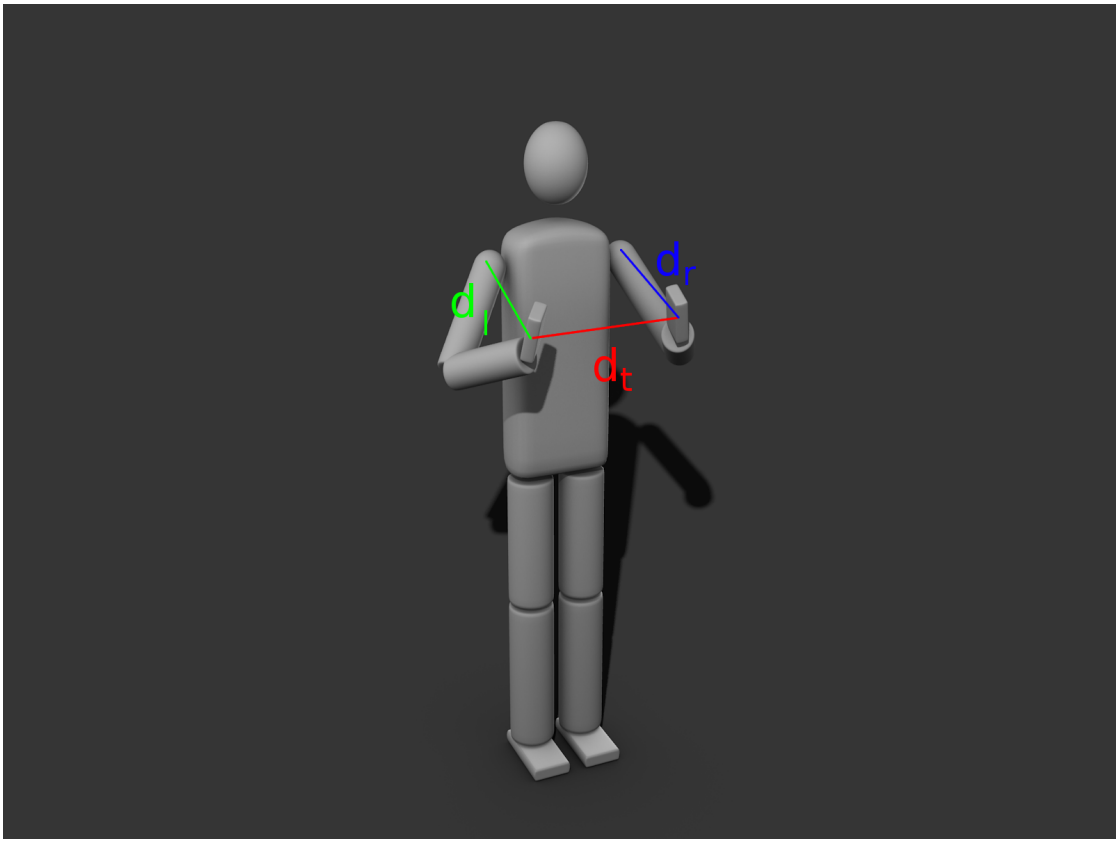

Abbildung 3.3: Grafische Darstellung der Körperhaltung für die Festlegung von Objektgrößen. Diese Körperhaltung erfüllt die Anforderungen aus Gleichung 3.36 bezüglich der Länge und der Höhenwinkel der Strecken Schulter–Hand. Die Objektgröße wird über den Abstand zwischen den beiden Händen d_t bestimmt. Zudem sind die Abstände d_l und d_r eingezeichnet.

$d_\delta = 100$mm als gute Werte heraus. Hat der Benutzer die korrekte Körperhaltung eingenommen und ist G aus Gleichung 3.36 damit erfüllt, wird der momentane Handabstand $d_t = \|\mathbf{p_{H,l}} - \mathbf{p_{H,r}}\|$ gemessen. Diese Messung wird wird so lange wiederholt, bis die Standardabweichung der Distanzmessung über die letzten $N = 15$ Messungen kleiner als $\sigma_{max} = 10$mm war. Wird G währenddessen nicht mehr erfüllt und die Zeigepositur somit verlassen, werden die bisherigen Messungen verworfen. Ist die Standardabweichung ausreichend gering, so wird die mittlere Distanz aus den letzten N Messungen berechnet und in Textform auf der Standardausgabe ausgegeben. Die zuletzt auf diese Art ermittelte Distanz kann vom Benutzer mittels einem Fingerschnipsen übernommen

werden. Da nur eine eindimensionale Größe d des Objekts bekannt ist, wird näherungsweise eine kugelförmige Verteilung angenommen[20]. Damit ergibt sich

$$\Sigma = d^2 \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.37}$$

3.2.4.3 Speichern der Objekte

Abschließend wird der Benutzer nach einer Objekt-ID und einem optionalen Label für das soeben festgelegte Objekt gefragt. Alternativ kann das Objekt verworfen werden. Nach der Festlegung eines Objekts können weitere Objekte aufgenommen werden. Sind alle Objekte aufgenommen worden, kann das Programm über die Tastenkombination `Ctrl-C` beendet werden. Dabei wird das Ergebnis als XML (siehe Abschnitt 3.2.3) in die Datei `pointing-TIME.xml` geschrieben. Dabei ist `TIME` der momentane UNIX-Zeitstempel, also die momentane Uhrzeit in Sekunden seit dem 1.1.1970, 00:00 UTC. Alle Teile der Konfigurationsdatei, die im Rahmen des Trainingsvorgangs nicht verändert wurden, werden aus der Standard-Konfigurationsdatei `pointing.xml` übernommen.

3.3 Gestenmodul

Das Gestenmodul soll eine vom Benutzer ausgeführte Geste über die Daten des Eingabemoduls (siehe Abschnitt 3.1.2.1) aufnehmen und klassifizieren.

3.3.1 Ansatz

Die Klassifikation der Gesten lässt sich in zwei Teilbereiche zerlegen. Zum einen muss ein geeignetes Merkmal gefunden werden, über das sich die Gesten klassifizieren lassen. Dies wird in Abschnitt 3.3.1.1 beschrieben. Zum anderen wird ein Klassifikator für Zeitserien benötigt, um anhand des Merkmals die ausgeführte Geste bestimmen zu können. Abschnitt 3.3.1.2 behandelt diesen Klassifikator.

3.3.1.1 Gestenmerkmal

Zunächst wurde ein räumliches Merkmal auf Basis einer Punktwolke verwendet. Idee war, die ausgeführte Geste anhand der räumlichen Form und Ausrichtung der Hand zu erkennen. Dazu wurde das Tiefenbild in eine Punktwolke überführt. Mittels der von Open-

[20] Auch wenn bei dieser Trainingsmethode Objekte nur als Kugeln approximiert werden, so hat die Modellierung über Verteilungen den Vorteil, dass Objekte mit kleiner Ausdehnung trotz Konkurrenz mit nahen, größeren Objekten vom Benutzer ausgewählt werden können.

NI bereitgestellten Methoden wurden die Pixel des Tiefenbildes von Bild- in Weltkoordinaten und somit in Punkte im Raum transformiert. Im Anschluss wurden alle Punkte $\mathbf{p}$ eliminiert, deren Abstand von der erfassten Handposition $\mathbf{p_H}$ größer als $\|\mathbf{p} - \mathbf{p_H}\| > r$ war. Um dies zu beschleunigen, wurde über eine Abbildung von Welt- nach Bildkoordinaten ein rechteckiger Ausschnitt des Tiefenbildes ermittelt, welcher die Kugel mit dem Radius r um die Handposition umfasst. Nur Pixel in diesem Ausschnitt wurden in Punkte transformiert. Im Anschluss wurden die verbleibenden Punkte der Hand rotationsnormiert. Dabei wurde die Rotationsmatrix $\mathbf{R}$ verwendet, wie sie auch schon in Abschnitt 3.2.1.3 zum Einsatz kam. $\mathbf{R}$ wird dabei zu Beginn aus der Strecke Hand-Schulter $\mathbf{p_H} - \mathbf{p_S}$ berechnet, so dass das aufgenommene Gestenmerkmal immer relativ zur Richtung des Arms zu Beginn der Geste ist. Zudem wurden die Punkte um ihren Mittelwert zentriert. Die transformierte Position eines jeden Handpunkts $\mathbf{p_i}$ ist dann

$$\mathbf{p_i}' = \mathbf{R}\left(\mathbf{p_i} - \left(\frac{1}{N}\sum_{j=1}^{N}\mathbf{p_j}\right)\right) \tag{3.38}$$

Um aus der Punktwolke einen Merkmalsvektor von niedriger Dimensionalität zu erstellen, wurde jeder Punkt einem von acht Sektoren zugeordnet. Dazu wurde das Koordinatensystem in der x-y-Ebene um den Nullpunkt herum in die vier Sektoren „Links“, „Rechts“, „Oben“ und „Unten“ unterteilt. Diese Sektoren wurden entlang der z-Achse im Nullpunkt halbiert, wodurch sich die Sektoren „Vorne Links“, „Hinten Links“, „Vorne Rechts“, und so weiter ergaben. Die Sektoraufteilung ist auch in der Grafik 3.4 dargestellt. Abschließend wurde der Anteil der Punkte in jedem der acht Sektoren berechnet. Diese Verteilung der Punkte ergab dann den Merkmalsvektor.

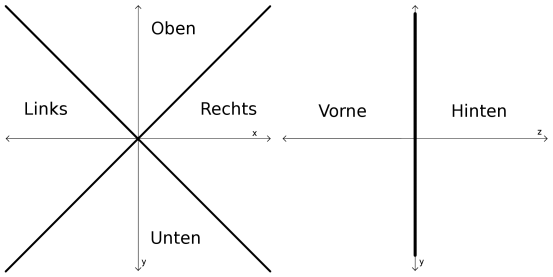

Abbildung 3.4: Grafische Darstellung der für das Punktmerkmal verwendeten Sektoren. Die dicken Linien sind die Sektorgrenzen, die dünnen Linien die Achsen des Koordinatensystems. Der Mittelpunkt der Handpunkte ist dabei der Ursprung des Koordinatensystems. Die linke Bildhälfte zeigt die Frontalansicht mit den Sektoren „Rechts“, „Links“, „Oben“ und „Unten“. Die rechte Bildhälfte zeigt die Seitenansicht mit den Sektoren „Vorne “und „Hinten“. Durch Kombination der beiden Ansichten ergeben sich dann die $4 \cdot 2 = 8$ verwendeten Sektoren.

Allerdings zeigte sich, dass dieses Merkmal nicht geeignet ist. Die Punktwolke wurde

aus dem Tiefenbild generiert, welches nur die der Kamera zugewandte Seite der Hand zeigt. Daher hing das Merkmal sehr stark davon ab, in welcher Orientierung sich die Hand relativ zur Kamera befand und welche Teile der Hand somit erfasst wurden. Diese Abhängigkeit war wesentlich stärker als die Abhängigkeit zwischen dem Merkmal und der ausgeführten Geste. Obwohl dieses Merkmal zunächst vielversprechend erschien und daher als erstes Merkmal implementiert wurde, erwies es sich letztendlich als ungeeignet und wurde nicht weiter verwendet.
Als alternatives Merkmal wurde die Position der Hand $\mathbf{p_H}$ verwendet, wie sie vom Eingabemodul bereitgestellt wird. Dabei wird die Hand für die Geste verwendet, mit der auch vor Beginn der Geste auf das Objekt gezeigt wurde. Die Handposition wird relativ zur Startposition $\mathbf{p_{H,1}}$ gemessen. Zudem wird die Bewegung der Hand entsprechend der Orientierung des Arms rotationsnormiert. Dazu wird die Rotationsmatrix $\mathbf{R}$ (wie in 3.2.1.3 berechnet) verwendet. Ausgangspunkt der Normierung ist dabei der Vektor Schulter–Hand zu Beginn der Geste, also $\mathbf{p_{H,1}} - \mathbf{p_{S,1}}$. Die erfasste Handposition und somit das Merkmal ist dann

$$\mathbf{p'_H} = \mathbf{R}(\mathbf{p_H} - \mathbf{p_{H,1}}) \tag{3.39}$$

In der Praxis zeigte sich, dass die Handposition als Merkmal gute Ergebnisse liefert. Zudem ist sie empfindlich genug ist, um auch Gesten aus dem Handgelenk, also ohne merkliche Armbewegung, zu erkennen. Daher wurde die normierte Handposition letztendlich als Merkmal verwendet.
Leider lässt sich die Länge der Geste nur schwer aus dem Merkmal erkennen, da unterschiedliche Benutzer sich nach Ende der Geste unterschiedlich verhalten. Einige Benutzer halten ihren Arm in der Endposition der Geste, andere bewegen ihn in die Zeigeposition zurück, während wieder andere den Arm absinken lassen. Daher wurde alternativ eine feste Gestenlänge verwendet. Dabei wird, nachdem die Geste durch Zeigen auf ein Objekt und Schnipsen begonnen wurde, eine Zeitserie aus T Datenpunkten aufgenommen. Da nicht alle ausgeführten Gesten die Länge T haben, ist es möglich, dass ein Teil der aufgenommenen Zeitserie nicht zur Geste gehört. Alternativ kann es sein, dass das Ende der Geste nicht mit aufgenommen wird. In der Praxis zeigte sich aber, dass dies über das Dynamic Time Warping im Klassifikator (siehe Abschnitt 3.3.1.2) kompensiert werden kann. Auch bei zu langen oder unvollständigen Aufnahmen lässt sich die Geste immer noch klassifizieren, solange sich der Unterschied zwischen der Länge der Geste und der Länge der Aufnahme T nicht zu groß wird. Diese Ideallänge einer Geste kann dem Benutzer jedoch leicht vermittelt und beim Entwurf des Gestenkanons berücksichtigt werden.
Alternativ wurde die Verwendung eines zweiten Schnipslautes zur Beendigung der Geste in Betrachtung gezogen. Allerdings bestand hier die Gefahr, dass Benutzer und System bei einem nicht erkannten zweiten Fingerschnipsen aus dem Takt geraten. In diesem Fall würde das System weiter Gestendaten aufnehmen und nicht zur Zeigeerkennung zu-

rückkehren. Hier müsste also die Aufnahme nach T_{max} Datenpunkten automatisch abgebrochen werden. Jedoch besteht auch hier die Gefahr, dass dem Benutzer nicht bewusst ist, dass die Geste trotz einem zweitem Schnipsen noch nicht beendet wurde.
In diesem Fall kann ein weiteres Schnipsen, zum Beispiel zur Einleitung einer neuen Geste, die fälschlicherweise noch laufende Geste beenden und an den Klassifikator übergeben. Dieses Problem lässt sich durch eine Rückmeldung des Systems an den Benutzer verringern, aber nie ganz lösen, da der Benutzer möglicherweise die Rückmeldung nicht rechtzeitig wahrnimmt. Zusätzlich können falsche Positive bei der Schnipserkennung dazu führen, dass eine unvollständige Geste an den Klassifikator übergeben wird. Außerdem erhöht ein zweites Schnipsen die Komplexität aus Sicht des Benutzers. Daher wurde dieser Ansatz nicht verwendet.
Ein weiterer Ansatz für Gesten mit variabler Länge finden sich in Abschnitt 4.3.3.2. Dieser Ansatz konnte jedoch im Rahmen dieser Arbeit nicht mehr umgesetzt werden.

3.3.1.2 Gestenklassifikation

Zur Klassifikation wurde ein Nächster-Nachbar Klassifikator auf Basis eines *Dynamic Time Warping* (DTW)-Algorithmus verwendet [23]. Der DTW-Algorithmus berechnet die Distanz zwischen zwei Zeitserien $\Phi = \phi_1, .., \phi_N$ und $\Theta = \theta_1, .., \theta_M$ der Länge N und M unter der günstigsten zeitlichen Verzerrung. Statt den Abstand der Zeitserien über die Summe der Abstände der Elemente $\sum_i \|\phi_i - \theta_i\|$ zu berechnen, wird der Abstand $\mathrm{DTW}(\Phi, \Theta) = \sum_{k=1}^{\max(N,M)} d(\phi_{u_k} - \theta_{v_k})$ berechnet. Dabei legen u_k und v_k die zeitliche Verzerrung der Zeitserien Φ und Θ bei der Berechnung der DTW-Distanz fest. Der DTW-Algorithmus wählt bei der Berechnung der Distanz $\mathrm{DTW}(\Phi, \Theta)$ die Verzerrungen u_k und v_k nun so, dass die Distanz minimal ist. Die DTW-Distanz ist also die Distanz zwischen zwei Zeitserien unter der günstigsten zeitlichen Verzerrung.
Die Zeitserien können aus Daten beliebigen Typs bestehen[21], solange sich eine Distanz $d(\phi_{u_k}, \theta_{v_k})$ über den Elementen der Zeitserie berechnen lässt. Optional kann ein so genanntes *Locality Constraint* δ verwendet werden, wodurch die maximale zeitliche Verzerrung zwischen zwei einander zugeordneten Elementen ϕ_{u_k} und θ_{v_k} auf $|u_k - v_k| < \delta_t$ beschränkt wird. Zudem gilt $u_l \geq u_k$ und $v_l \geq v_k$ für $l \geq k$, die Zeitserien können also auch unter der Verzerrung in der Zeit nicht rückwärts laufen. Zur Berechnung der DTW-Distanz werden die Merkmalsvektoren aus Abschnitt 3.3.1.1 mit einer euklidischen Distanz verwendet. Die genaue Funktionsweise des DTW-Algorithmuses ist unter anderem in Sakoe und Chiba [23] beschrieben.
Auf Basis der DTW-Distanz wird nun der nächste Nachbar, also das Trainingsdatum T_i mit der geringsten Distanz zur aufgenommenen Geste G, gesucht. Der Index des nächsten Trainingsdatums ist somit

$$i = \mathrm{argmin}_j \, \mathrm{DTW}(T_j, G) \tag{3.40}$$

[21] Häufige Beispiele sind Skalare oder Vektoren.

Die Klasse von G entspricht dann der Klasse von T_i. Optional kann eine Rückweisung vorgenommen werden, wenn die Distanz $d = \mathrm{DTW}(T_i, G) > \epsilon$ ist. Im Falle einer Rückweisung gehört die Geste G zu keiner der bekannten Klasse und wird nicht weiter verarbeitet. Nachdem die Geste klassifiziert wurde, wird das Ergebnis dem Ausgabemodul aus Abschnitt 3.5 zur Verfügung gestellt.

3.3.2 Implementation

Der Zeitserienklassifikator aus Abschnitt 3.3.1.2 wurde in der Datei `timeClassifier.py` als `TimeClassifier`-Klasse umgesetzt. Dabei wurde der DTW-Algorithmus in der Datei `dtw.py` implementiert. Dem `TimeClassifer` können über die Methode `addSample` Trainingsdaten für die Nächster-Nachbar-Klassifikation übergeben werden. Im Anschluss kann über die Methode `addFrame` jeweils ein neu aufgenommenes Datum hinzugefügt werden. Die Daten werden in einem Ringspeicher (siehe Abschnitt 3.7.2) gespeichert, dessen Länge bei der Instanziierung über den Parameter `bufferLength` festgelegt wird. Werden über die `addFrame`-Methode mehr Daten hinzugefügt als der Ringspeicher aufnehmen kann, so überschreibt das neue Datum jeweils das älteste Datum im Speicher. Nach Aufnahme der Daten kann durch Aufruf der `classify`-Methode der Inhalt des Speichers mit dem Klassifikator aus Abschnitt 3.3.1.2 klassifiziert werden. Dabei wird ein Tupel aus dem Klassennamen und dem Abstand d zum nächsten Nachbarn zurückgegeben. Ist d größer als der Schwellwert aus der Klassenvariable `rejectionThreshold`, so wird als Klassenname der Wert `None` zurückgegeben. Ist der Ringspeicher beim Aufruf von `classify` leer, dann hat der Klassenname den Wert `None`. Für die Distanz wird in diesem Fall $d = \infty$ verwendet.
Zusätzlich lassen sich in der `TimeClassifier`-Klasse eine Zentrierung oder Skalierung der Daten aktivieren, indem die die `rescale`- und `recenter`-Parameter bei der Instanziierung auf `True` gesetzt werden. Die Zentrierung und Skalierung werden dabei auf alle von der `TimeClassifier`-Instanz verwendeten Zeitserien, inklusive der Trainingsdaten, angewendet.
Hat `recenter` den Wert `True`, so werden die Daten aller Zeitserien $\mathbf{\Theta} = \theta_\mathbf{1}, \theta_\mathbf{2}, .., \theta_\mathbf{N}$ um den Mittelwert der Zeitserie zentriert. Sind die Daten $\theta_\mathbf{i}$ der Zeitserie $\mathbf{\Theta}$ K-elementige Vektoren mit den Elementen $\theta_\mathbf{i} = (\theta_{i,1}, \theta_{i,2}, .., \theta_{i,K})$, so ergibt sich die zentrierte Zeitserie $\mathbf{\Theta^C} = \theta^\mathbf{C}_\mathbf{1}, \theta^\mathbf{C}_\mathbf{2}, .., \theta^\mathbf{C}_\mathbf{N}$ aus den Gleichungen

$$\mu_k = \frac{1}{N} \sum_{j=1}^{N} \theta_{j,k} \tag{3.41}$$

$$\theta^\mathbf{C}_\mathbf{i} = (\theta_{i,1} - \mu_1, \theta_{i,2} - \mu_2, .., \theta_{i,K} - \mu_K) \tag{3.42}$$

Dabei ist μ_k der Mittelwert der k-ten Komponente der Daten über alle Messungen j.

Ist der `rescale`-Parameter `True`, so werden die Zeitserien entsprechend ihrer Standardabweichung skaliert. Dazu wird ein Skaleriungsfaktor η aus der Standardabweichung berechnet. Im Anschluss werden alle Daten mit η skaliert, um die skalierte Zeitserie $\mathbf{\Theta^S} = \theta_\mathbf{1}^\mathbf{S}, \theta_\mathbf{2}^\mathbf{S}, .., \theta_\mathbf{N}^\mathbf{S}$ zu erhalten:

$$\sigma_k = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(\theta_{j,k} - \mu_k)^2} \tag{3.43}$$

$$\eta = \sqrt{\sum_{k=1}^{K}\sigma_k^2} \tag{3.44}$$

$$\theta_\mathbf{i}^\mathbf{S} = (\frac{\theta_{i,1}}{\eta}, \frac{\theta_{i,2}}{\eta}, .., \frac{\theta_{i,K}}{\eta}) \tag{3.45}$$

σ_k ist dabei die Standardabweichung des k-ten Elements der Datenvektoren über alle Messungen j.
Auf Basis der `TimeClassifier`-Klasse wurde die `gestureHandler`-Klasse in der Datei `gesture.py` implementiert. Diese verwendet das Gestenmerkmal aus Abschnitt 3.3.1.1 und bietet die Methoden `startRecording`, `finishRecording` und `recordGesture`. Durch Aufruf von `startRecording` wird eine neue Aufnahme begonnen. Dabei werden die bisher aufgenommenen Daten verworfen. `recordGesture` fügt der Aufnahme ein neues Datum hinzu. Dafür werden die Positionen der Hand und Schulter als Instanzen der Klasse `skelJoint` übergeben. Aus den Positionen wird das Gestenmerkmal berechnet. Dabei wird die erste nach Beginn der Aufnahme übergebene Hand- und Schulterposition als Ursprung für die Berechnung der relativen Merkmale und somit als Grundlage für die Berechnung der Rotationsmatrix $\mathbf{R}$ und der Verschiebung $\mathbf{t}$ verwendet (siehe Abschnitt 3.3.1.1). Das berechnete Merkmal wird dann einem `TimeClassifier` hinzugefügt. Der Rückgabewert der `recordGesture`-Methode ist die Anzahl der noch aufzunehmenden Daten. Sobald `recordGesture` den Wert 0 zurückgibt, kann die Methode `finishRecording` aufgerufen werden. Die `finishRecording`-Methode beendet die Aufnahme, nimmt eine Klassifikation vor und gibt den Namen der gefundenen Gestenklasse zurück. Wenn die aufgenommene Zeitserie keiner Klasse zugeordnet werden konnte, wird `False` zurückgegeben. Wird `finishRecording` aufgerufen, ohne dass zuvor eine Aufnahme gestartet wurde, wird der Wert `None` zurückgegeben.

3.3.3 Konfiguration

Die Trainingsdaten und Parameter einer `TimeClassifier`-Instanz können aus einer XML-Konfigurationsdatei geladen werden. Dazu steht die Funktion `loadTimeClassifier` aus dem Python-Modul `timeClassifierConfig.py`

bereit. Diese Funktion nimmt den Namen einer Konfigurationsdatei entgegen und gibt eine entsprechend konfigurierte `TimeClassifier`-Instanz zurück. Zusätzlich kann der Name der Konfigurationsdatei bei der Instanziierung der `gestureHandler`-Klasse übergeben werden, wodurch der von der `gestureHandler`-Klasse verwendete `TimeClassifier` entsprechend konfiguriert wird.
Das Format der Konfigurationsdatei ist in der Datei `timeseries.xsd` als XML-Schema (XSD) festgelegt. Wurzel der Konfigurationsdatei ist dabei ein `<model>`-Element mit den Kindelementen `<parameters>` und `<classes>`.
Das `<parameters>`-Element enthält die Parameter des Klassifikators. Das Kindelement `<bufferlength>` gibt die Länge des Ringspeicher und somit auch die Länge der aufgenommenen Geste (siehe Abschnitt 3.3.3) als positive Ganzzahl an. Das Kindelement `<rejectionthreshold>` gibt die in Abschnitt 3.3.1.2 verwendete Distanz ϵ an, ab welcher eine Geste zurückgewiesen wird. Epsilon ist eine positive Gleitkommazahl. Soll der Wert $\epsilon = \infty$ verwendet werden, so kann dies in XML über den Wert `INF` angegeben werden. Die optionalen Kindelemente `<recenter>` und `<rescale>` entsprechen den `recenter`- und `rescale`-Parametern der `TimeClassifier`-Klasse aus Abschnitt , wobei nach dem XML-Schema für boolsche Werte `true` und `1` beziehungsweise `false` und `0` erlaubt sind. Sind diese Elemente nicht angegeben, so wird der Wert `false` angenommen.
Das `<classes>`-Element enthält eine beliebige Anzahl an Klassen mit ihren Prototypen. Jede Klasse wird dabei durch ein `<class>`-Kindelement spezifiziert. Das `name`-Attribut dieses Elements gibt den Namen der Klasse an, wie er bei einem Treffer vom Klassifikator zurück- und an das Aktionsmodell aus Abschnitt 3.5 weitergegeben wird. Jedes `<class>`-Element kann eine beliebige Anzahl an `<sample>`-Elementen umfassen, wobei jedes `<sample>`-Element einem Prototypen Θ dieser Klasse entspricht. Jedes `<sample>` besteht aus einem oder mehreren `<frame>`-Elementen, die jeweils einen Vektor θ_i der Zeitserie Θ beinhalten. Dabei entspricht die Reihenfolge der `<frame>`-Elemente im `<sample>`-Element der Reihenfolge der Vektoren θ_i in der Zeitserie Θ. Jeder Vektor θ_i wird als mit Leerzeichen getrennte Liste seiner Elemente in einem `<frame>`-Element angegeben. Dabei müssen alle Vektoren und somit alle `<frame>`-Elemente in einer Konfigurationsdatei die gleiche Länge haben, da ansonsten die Distanzberechnung zwischen zwei Vektoren fehlschlagen kann.

3.3.4 Training

Das Gestenmodul kann über das Programm `gestureTrainer.py` interaktiv trainiert werden. Nach Start des Programms und der Benutzerkalibrierung kann der Benutzer mittels Fingerschnipsen Gesten einleiten. Dabei muss der Arm ausgestreckt sein, aber nicht auf ein gültiges Objekt zeigen. Nach Beginn der Geste wird diese genau wie bei der Klassifikation entsprechend Abschnitt 3.3.1.1 und 3.3.3 aufgezeichnet. Die Zentrierung und Skalierung aus Abschnitt 3.3.3 sind dabei immer deaktiviert. Die Gestenlänge wird aus

dem <bufferlength>-Element der Standard-Konfigurationsdatei gestures.xml übernommen.
Nach Ende der Geste wird dem Benutzer die Möglichkeit gegeben, die Geste als Prototyp einer bestimmten Klasse zu speichern oder sie zu verwerfen. Dies kann beliebig oft wiederholt werden, um mehrere Gesten aufzunehmen. Wird das Programm mit der Tastenkombination Ctrl-C beendet, so wird eine Konfigurationsdatei im XML-Format aus Abschnitt 3.3.3 unter dem Namen gesture-TIME.xml gespeichert. TIME ist dabei der aktuelle UNIX-Zeitstempel, also die Uhrzeit in Sekunden seit dem 1.1.1970, 00:00 UTC. Der <parameters>-Teil der Konfigurationsdatei, welcher im Rahmen des Trainings nicht verändert wurde, wird aus der Standard-Konfigurationsdatei gestures.xml übernommen.

3.4 Audiomodul

Aufgabe des Audiomoduls ist es, entsprechend den Anforderungen aus Abschnitt 2.2.2 Audiodaten aufzunehmen. Im Anschluss müssen Transienten, insbesondere Schnipslaute, erkannt und klassifiziert werden.

3.4.1 Ansatz

Der ursprüngliche Ansatz für dieses Modul stammt aus einem System zur Steuerung von Benutzeroberflächen mittels Fingerschnipsen [24]. Jedoch wurden einige Veränderungen vorgenommen, unter anderem da der von Vesa und Lokki [24] beschriebene Ansatz Stereomikrofone erfordert, im Rahmen dieser Arbeit jedoch ein omnidirektionales Mono-Mikrofon verwendet wurde. Die Audiodaten werden über ein gleitendes Fenster $\mathbf{s} = (s_0, ..., s_{N-1})$ der Länge N analysiert, welches für jeden Analyseschritt jeweils um $M \leq N$ Samples verschoben wird. Wurden M Samples als neue Daten $\mathbf{c}$ aufgenommen, so ergibt sich der aktualisierte Datensatz s' also durch Anhängen von $\mathbf{c}$ an $\mathbf{s}$: $\mathbf{s'} = (s_M, ..., s_{N-1}, c_0, ..., c_{M-1})$. Dies entspricht in der Funktionsweise dem in Abschnitt 3.7.2 beschriebenen Ringspeicher.

3.4.1.1 Transientenerkennung

Auf diesem Signal $\mathbf{s}$ sollen nun zunächst Transienten erkannt werden. Da Transienten einen plötzlichen Anstieg der Signalamplitude darstellen, bietet sich die Erkennung über den Vergleich der maximalen momentanen Signalenergie s_i^2 mit einem Schwellwert θ an. Dies führt zur Ungleichung

$$\max_i \; s_i^2 > \theta \tag{3.46}$$

Befindet sich die Quelle des Geräuschs jedoch in einiger Entfernung vom Mikrofon, so weist der Transient eventuell nur eine sehr geringe maximale momentane Energie auf. Zur Erkennung des Transienten muss dann der Schwellwert θ gering gewählt werden. Der Schwellwert wird dann aber eventuell auch von störenden Hintergrundgeräuschen erreicht. Dies führt leicht zu falschen Positiven bei der Transientenerkennung. Daher wird das Signal bei der Transientenerkennung mit der mittleren quadratischen Amplitude von $\mathbf{s}$ normiert. Die resultierende Ungleichung lautet

$$s_t = \max_i \frac{N s_i^2}{\sum_{j=0}^{N-1} s_j^2} \tag{3.47}$$

$$s_t > \theta \tag{3.48}$$

Hier wird also die maximale momentane Energie in $\mathbf{s}$ nur noch relativ zur Gesamtenergie von $\mathbf{s}$ als s_t betrachtet. Dadurch wird bei einer lauten Umgebung mit hoher, aber über das Signal verteilten Gesamtenergie anders als in Gleichung 3.46 bei passend gewähltem θ kein Transient mehr erkannt[22].

3.4.1.2 Klassifikation

Wurde für das momentane Fenster $\mathbf{s}$ entsprechend Gleichung 3.48 ein Transient erkannt, muss dieser klassifiziert werden. Dazu werden wie bei Vesa und Lokki [24] die Band-Energie-Raten von $\mathbf{s}$ verwendet. Die Band-Energie-Rate ist ein Merkmal im Frequenzraum, bei dem die Energie innerhalb bestimmter Frequenzbänder relativ zur Gesamtenergie des Signals betrachtet wird. Bei K Bändern mit den Unter- und Obergrenzen $f_{l,k}$ und $f_{u,k}$ ergibt sich die Band-Energie-Rate des k-ten Bands r_k als

$$r_k = \frac{\sum_{i=f_{l,k}}^{f_{u,k}-1} |S_i|^2}{\sum_{j=1}^{\frac{N}{2}} |S_j|^2} \tag{3.49}$$

Dabei ist $\mathbf{S} = DFT(\mathbf{s})$ das Ergebnis der diskreten Fouriertransformation von $\mathbf{s}$, also $S_k = \sum_{n=0}^{N-1} s_n e^{-2\pi i k \frac{n}{N}}$. Die Band-Energie-Raten $\mathbf{r} = (r_1, .., r_K)$ dienen nun als Merkmal für die Klassifikation.
Die große Anzahl an möglichen Transienten macht es unmöglich, umfassende Negativbeispiele zum Trainieren eines Zwei-Klassen-Klassifikators zu sammeln. Daher wird ein Klassifikator verwendet, der die Klassifikation allein über die Ähnlichkeit mit einigen aus einer Menge an Positivbeispielen gewonnenen Prototypen $\mathbf{p_l} = (p_{l,1}, .., p_{l,K})$ vornimmt. Daher wird ein Nächster-Nachbar-Klassifikator verwendet, bei dem ein Signal mit den Band-Energie-Raten $\mathbf{r} = (r_1, .., r_k)$ genau dann akzeptiert wird, wenn es

[22] Die Ungleichung 3.48 lässt sich auch als $\max_i s_i^2 > \theta \frac{\sum_{j=0}^{N-1} s_j^2}{N}$ schreiben. Es handelt sich bei diesem Verfahren also letztendlich um einen adaptiven Schwellwert.

ausreichend nah an einem Prototypen $\mathbf{p_l}$ liegt. Zur Distanzberechnung wird dabei eine mit den Faktoren $\frac{1}{\sigma_k^2}$ gewichtete euklidische Distanz verwendet. Das Signal $\mathbf{s}$ mit dem dazugehörigen Merkmal $\mathbf{r}$ wird genau dann akzeptiert, wenn

$$\exists l : \sqrt{\sum_{k=1}^{K} \left(\frac{r_k - p_{l,k}}{\sigma_k} \right)^2} < \theta_l \tag{3.50}$$

gilt. Mit anderen Worten: Das Signal $\mathbf{s}$ wird dann akzeptiert, wenn ein Prototyp $\mathbf{p_l}$ existiert, für den der Abstand zwischen $\mathbf{r}$ und $\mathbf{p_l}$ unter Gewichtung der einzelnen Bänder mit den Faktoren $\frac{1}{\sigma_k^2}$ geringer ist als der zum Prototypen gehörende Schwellwert θ_l. Über eine optionale Slackvariable λ kann die Striktheit des Klassifikators global gesteuert werden. Dafür wird der Schwellwert θ_l in Gleichung 3.50 durch den Schwellwert $\theta_l{}' = \lambda\theta_l$ ersetzt. Wird die Slackvariable nicht explizit gesetzt, so gilt $\lambda = 1$ und somit $\theta_l{}' = \theta_l$.
Im Rahmen der Implementation wurde zusätzlich überlegt, ob Teile des Signals, die innerhalb des Analysefensters vor oder nach dem zu untersuchenden Transienten liegen, gelöscht werden können. Dazu könnte zum Beispiel das normierte Signalmaximum s_t aus Gleichung 3.48 verwendet werden. Dabei würden dann alle s_q mit $q < t$ auf den Wert 0 gesetzt würden, da diese Teile des Signals noch vor dem Maximum des Transienten liegen. In der Praxis zeigte sich jedoch, dass diese Teile des Signals nützlich sind, obwohl sie nicht Teil des gesuchten Geräuschs sind. Bei unerwünschten Geräuschen kann dieser Teil des Signals während der Klassifikation zur Unterscheidung vom gesuchten Geräusch beitragen. Aus gleichem Grund wird auch die Größe des Analysefensters N höher als die erwartete Länge des gesuchten Transienten gewählt. Da unerwünschte Geräusche eventuell länger sind, kann die höhere Länge des Analysefensters zur Unterscheidung beitragen.

3.4.2 Implementation

Das in Abschnitt 3.4.1 beschriebene Verfahren wurde als Python-Modul in der Datei `snapDetection.py` implementiert. Dabei wurde die PyAudio-Bibliothek verwendet. Diese Bibliothek ermöglicht über die darunterliegende PortAudio-Bibliothek einen plattformunabhängigen Zugriff auf die Audiohardware [21].
Das Python-Modul `snapDetection.py` stellt die Klasse `snapDetector` bereit. Diese implementiert das Audiomodul in einem eigenen Thread. Dadurch kann die Audioverarbeitung unabhängig und asynchron vom Restsystem laufen[23]. Die Kommunikation mit dem Restsystem erfolgt über eine Callback-Funktion. Diese wird aufgerufen,

[23] Der Standardinterpreter für Python (*CPython*) unterstützt keine echte Nebenläufigkeit für Threads. Threads sind dennoch von Nutzen, da auf Ein- oder Ausgabeoperationen wartende Threads von anderen Threads abgelöst werden können.

wenn ein Schnipsen entsprechend Abschnitt 3.4.1 erkannt und vom Klassifikator akzeptiert wurde.
Die Analyse des Audiosignals beginnt nach Aufruf der `start`-Methode einer `snapDetector`-Instanz. Als Parameter werden die Callback-Funktion `callback`, die Eingabedatei `infile`, der Blockparameter `block` und die Größenparameter `frame_size_ms` und `frame_step_ms` übergeben. Der Callback-Funktion `callback` wird beim Aufruf das normierte Signalmaximum s_t aus Gleichung 3.48, der mit $\frac{1}{\sigma_k}$ normierte Featurevektor $\mathbf{r}$ aus Abschnitt 3.4.1.2 und der momentane Zeitpunkt im Audio-Datenstrom in Sekunden übergeben. Ist der Parameter `infile` der Dateiname einer WAV-Datei, so wird der Inhalt dieser Datei analysiert. Hat `infile` den Wert `None`, so wird der Standard-Aufnamekanal des Rechners verwendet. Ist `block` auf `True` gesetzt, so kehrt die Methode `start` zuerst zurück, sobald die Analyse beendet wurde. Die Analyse endet nur dann von selbst, wenn eine endliche Eingabedatei über den Parameter `infile` angegeben wurde. Ist `block True` und `inflile None`, so muss der Thread manuell beendet werden bevor `start` zurückkehrt. Die Analyse läuft dabei immer in ihrem eigenen Thread, `block` beeinflusst nur den Zeitpunkt an dem die Methode `start` zurückkehrt. Die Parameter `frame_size_ms` und `frame_step_ms` geben die Längen des Analysefensters und die Verschiebung des Analysefensters zwischen jeder Analyse in Millisekunden an. Dies entspricht den Werten N und M aus Abschnitt 3.4.1.
Sobald die `start`-Methode den Thread initialisiert hat, werden mit der Methode `update_audio` neue Audiodaten eingelesen und mit der Methode `detect_snap` analysiert.

3.4.3 Konfiguration

Die Konfiguration des Audiomoduls wird in einer XML-Datei festgehalten. Der Aufbau dieser Konfigurationsdatei ist im XML-Schema `audio.xsd` festgelegt. Das Modul `snapDetectionConfig.py` enthält die Klasse `SnapDetectionConfig`. Wird dem Konstruktor dieser Klasse der Name einer Konfigurationsdatei übergeben, so wird der Inhalt der Konfigurationsdatei in die `SnapDetectionConfig`-Instanz geladen. Diese kann dann der `snapDetector`-Klasse bei der Instanziierung übergeben werden.
Die XML-Konfigurationsdatei ist wie folgt aufgebaut: Als Wurzelelement dient das `<parameters>`-Element. Diesem Element sind die Elemente `<threshold>`, `<slack>`, `<prototypes>` und `<bands>` untergeordnet. Das `<threshold>`-Element beinhaltet den in Abschnitt 3.4.1.1 beschriebenen Schwellwert $\theta \geq 0$ als nichtnegative Gleitkommazahl. Das optionale `<slack>`-Element beinhaltet die in Abschnitt 3.4.1.2 beschriebene Slackvariable $\lambda \geq 0$ als nichtnegative Gleitkommazahl.
Das `<prototypes>`-Element beinhaltet ein oder mehrere `<prototype>`-Elemente. Jedes `<prototype>`-Element beschreibt jeweils einen der in Abschnitt 3.4.1.2 ver-

wendeten Prototypen. Ein <prototype>-Element besteht aus einem <distance> und einem <vector>-Kindelement. Das <distance>-Element beinhaltet den zum Prototypen gehörenden Schwellwert θ_l als nichtnegative Gleitkommazahl. Das <vector>-Element enthält den eigentlichen Prototypen $\mathbf{p_l} = (p_{l,1}, .., p_{l,K})$ als leerzeichengetrennte Liste aus Gleitkommazahlen. Im Fall von $K = 3$ hat das <vector>-Element also die Form <vector>$p_{l,1}$ $p_{l,2}$ $p_{l,3}$</vector>.
Das <bands>-Element beschreibt die Frequenzbänder, über denen, wie in Abschnitt 3.4.1.2 beschrieben, die Band-Energie-Raten berechnet werden. Das <bands>-Element umfasst ein oder mehrere <band>-Elemente, die jeweils eines der K Frequenzbänder beschreiben. Jedes <band>-Element besteht aus einem <min>, <max> sowie einem optionalen <scale>-Element. Das <min> und <max>-Element beinhalten jeweils die Unter- und Obergrenze $f_{l,k}$ und $f_{u,k}$ des Frequenzbands als nichtnegative ganze Zahl. Das optionale <scale>-Element beinhaltet den Gewichtungsfaktor σ_k als positive Gleitkommazahl. Ist kein <scale>-Element angegeben, so gilt $\sigma_k = 1$.

3.4.4 Training

Zum Generieren der in Abschnitt 3.4.1.2 beschriebenen Prototypen p_l, der Skalierungsfaktoren σ_k und der Schwellwerte θ_l kann ein Trainingsverfahren verwendet werden. Dieses ist im Python-Programm `snapDetectionTrainer.py` über die Klasse `snapTrainer` implementiert. Das Programm wird über die Kommandozeile gestartet und nimmt als optionales Argument den Namen einer WAV-Audiodatei entgegen. Wird eine Audiodatei angegeben, so wird diese als Quellsignal verwendet. Ansonsten verwendet das Programm den Standard-Aufnahmekanal des Rechners als Quelle. Das Quellsignal wird dabei Anhand des in Abschnitt 3.4.1.1 beschriebenen Verfahren nach Transienten durchsucht. Der dabei verwendete Schwellwert θ wird entsprechend Abschnitt 3.4.3 aus der Standard-Konfigurationsdatei `snapping.xml` geladen. Da θ eine Voraussetzung für den Trainingsvorgang ist, muss dieser Wert von Hand gewählt werden. Wird ein Transient erkannt, so muss der Benutzer entscheiden, ob der Transient ein Schnipsen ist. Dazu wird auch der Zeitpunkt im Eingabesignal angegeben, an dem der Transient erkannt wurde.
Akzeptiert der Benutzer den erkannten Transienten als Schnipsen, so wird aus dem aktuellen Signal das Merkmal $\mathbf{r_{N+1}}$ berechnet (siehe Abschnitt 3.4.1.2) und der Menge der Trainingsdaten $\mathbf{R} = \mathbf{r_1}, .., \mathbf{r_N}, \mathbf{r_{N+1}}$ hinzugefügt.
Die Datenaufnahme ist dann beendet, wenn die Eingabedatei endet oder der Benutzer das Trainingsprogramm durch die Tastenkombination `Ctrl-C` unterbricht. Nach Abschluss der Datenaufname werden auf Basis der Band-Energie-Raten $\mathbf{R}$ zunächst die

Skalierungsfaktoren σ_k berechnet. Diese werden jeweils auf die empirische Standardabweichung jedes Bandes k gesetzt. Es gilt also

$$\mu_k = \frac{1}{N}\sum_{n=1}^{N} r_{n,k} \tag{3.51}$$

$$\sigma_k = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(r_{n,k} - \mu_k)^2} \tag{3.52}$$

Durch dieses Whitening ergibt sich eine uniforme Varianz der Bänder über alle Trainingsdaten. Dies erleichtert die Ermittlung der Prototypen [2]. Die auf dem Whitening beruhenden Skalierungsfaktoren σ_k werden also nicht nur in die Konfigurationsdatei geschrieben sondern auch bei der Wahl der Prototypen $\mathbf{p}$ verwendet. Die Wahl der Prototypen beruht also auf den skalierten Trainingsdaten $\mathbf{R}' = \mathbf{r_1}', .., \mathbf{r_N}'$, wobei sich durch die Skalierung $\mathbf{r_n}' = (\frac{r_{n,1}}{\sigma_1}, .., \frac{r_{n,K}}{\sigma_K})$ ergibt. Um die große Anzahl der Trainingsdaten in $\mathbf{R}'$ auf eine kleinere Menge an Prototypen $\mathbf{p_l}$ zu reduzieren, wird das k-Means Clusteringverfahren verwendet [8, 424ff.]. Die maximale Anzahl der Cluster wird dabei über die globale Variable `PROTOTYPE_COUNT` im Modul `snapDetectionTraining.py` festgelegt.
Zuletzt muss noch zu jedem Prototypen $\mathbf{p_l}$ ein individueller Schwellwert θ_l berechnet werden. Dazu werden die skalierten Trainingsdaten $\mathbf{r_n}'$ dem jeweils nächsten Prototypen zugeordnet:

$$\mathbf{V_l} = \{\mathbf{r}' \in \mathbf{R}' : \|\mathbf{p_l} - \mathbf{r}\| \leq \|\mathbf{p_n} - \mathbf{r}\| \forall n \neq l\} \tag{3.53}$$

In Anlehnung an das k-Means-Verfahren wird θ_l empirisch als maximale Ausdehnung eines Clusters gewählt. θ_l ist also der größte Abstand zwischen dem Prototypen $\mathbf{p_l}$ und einem Trainingsdatum $\mathbf{r}$, das auf $\mathbf{p_l}$ abgebildet wurde:

$$\theta_l = \max_{\mathbf{r}' \in \mathbf{V_l}} \|\mathbf{p_l} - \mathbf{r}'\| \tag{3.54}$$

Da der k-Means-Algorithmus versucht, den euklidischen Abstand zwischen den Prototypen und den auf die Prototypen abgebildeten Trainingsdaten zu minimieren [1], führt diese Wahl der Schwellwerte zu möglichst kleinen θ_l. Wird ein Cluster generiert, welchem nur ein einziges Trainingsdatum zugeordnet wurde, so wird dieser Cluster entfernt, da sich θ_l nicht sinnvoll bestimmen lässt.
Nachdem alle Parameter ermittelt wurden, wird die neue Konfiguration mit dem Format aus Abschnitt 3.4.3 als XML in die Ausgabedatei `snapping-TIME.xml` geschrieben. Dabei ist `TIME` der momentane UNIX-Zeitstempel, also die vergangene Zeit seit 00:00 UTC am 1.1.1970 in Sekunden. Werte wie der Schwellwert θ oder die Slackvariable λ, die im Trainingsverfahren nicht berechnet wurden, werden aus der Standard-Konfigurationsdatei `snapping.xml` übernommen.

Die Konfiguration des Audiomoduls kann mittels des Python-Programms `snapDetectionTest.py` getestet werden. Dieses Programm nutzt die Konfiguration aus der Datei `snapping.xml`, um auf dem Standard-Eingangskanal oder auf einer als Parameter übergebenen `WAV`-Datei eine Schnipserkennung durchzuführen. Wird ein Schnipslaut erkannt, so wird eine Meldung auf der Standardausgabe (`stdout`) ausgegeben.

3.5 Ausgabemodul

Das Ausgabemodul muss, wie in Abschnitt 2.3 beschrieben, bei einem ausgewählten Objekt O und einer ausgeführten Geste G eine bestimmte Zustandsänderung an einem physischen Objekt auslösen. Im Rahmen der *MIRIAM*-Architektur wird jedes ansteuerbare physische Objekt durch ein virtuelles Smart Object repräsentiert [15]. Jedes Smart Object hat dabei einen oder mehrere Kanäle, die jeweils einen bestimmten Aspekt des Objekts steuern. Bei einem Smart Object, welches eine Lampe steuert, wären zum Beispiel Kanäle für den Status (an oder aus) und die Helligkeit denkbar.
Dies lässt sich in zwei Teilvorgänge unterteilen: Zum einen wird ein Aktionsmodell $A = M_{S,C}(O, G)$, benötigt, welches auf Basis des Objekts O und der Geste G die gewünschte Änderung A für jeden Kanal C jedes Smart Objects S auswählt. Zum anderen muss die ausgewählte Aktion A umgesetzt werden, indem mit den Smart Objects kommuniziert wird.

3.5.1 Ansatz

Für das Aktionsmodell $M_{S,C}$ bieten sich eine Reihe an Lösungsansätzen an, die sich im Grad der Flexibilität und Komplexität unterscheiden. Einige davon sollen hier dargestellt und diskutiert werden, bevor einer der Ansätze als Lösung ausgewählt wird.

3.5.1.1 Elementare Operationen

Einer der einfachsten Ansätze ist es, eine Reihe von möglichen elementaren Operationen festzulegen. Denkbare Operatoren sind zum Beispiel das Setzen eines Kanals auf einen festen Wert oder die Anwendung arithmetischer Operatoren auf den momentanen Wert eines Kanals. Die Konfiguration des Aktionsmodells erfolgt dann über eine Abbildung von Objekt-Geste-Tupeln (O, G) auf einer Operator und eventuelle Operanden. Die Abbildung wird dabei aus einer Konfigurationsdatei gelesen. Dieser Ansatz hat den Vorteil, dass er robust und einfach zu konfigurieren ist. Durch die begrenzte Anzahl an vorher festgelegten Operatoren lässt sich sicherstellen, dass ein gegebenes Aktionsmodell gültig ist und den Kanälen nur erlaubte Werte zuweist. Zudem ist die Konfiguration und damit auch das Format und die Verarbeitung der Konfigurationsdatei einfach, da jeweils

nur eine Zuordnung von Objekt-Geste-Kombinationen auf Operationen vorgenommen werden muss.
Allerdings bietet dieser Ansatz nur wenig Flexibilität. Durch die fest eingebauten möglichen elementaren Operatoren ist die Anzahl der möglichen Operationen begrenzt. Komplexe Operationen sind nur dann möglich, wenn der Operator hierfür auch existiert. Zudem ist die Implementation des Aktionsmodells aufwändig, da jeder Operator einzeln implementiert werden muss. Erweiterungen sind nur über die Implementation zusätzlicher Operatoren möglich. Da die Smart-Object-Kanäle zudem eine Vielzahl an möglichen Datentypen wie Zeichenketten, Zahlen, boolsche Werte und Zeitstempel haben können, müssen viele spezielle Operatoren implementiert werden. Die Robustheit und Einfachheit bei der Konfiguration des Aktionsmodells wird also durch durch einen Mangel an Flexibilität und eine geringe Generalisierung und damit hohen Aufwand erkauft.

3.5.1.2 Frei programmierbares Aktionsmodell

Als Alternative am anderen Ende des Komplexitäts/Flexibilitäts-Konflikts kann eine Hochsprache zur Programmierung des Aktionsmodells verwendet werden. Dabei wird bei der Definition des Aktionsmodells einem Objekt-Geste-Tupel ein Block Quellcode zugeordnet. Tritt nun ein verwendetes Objekt-Geste-Tupel auf, so wird der dazugehörige Quellcode aus der Konfigurationsdatei gelesen und ausgeführt. Da die Implementation des Ausgabemoduls in der Programmiersprache Python erfolgt, bietet es sich an, auch bei der Definition des Aktionsmodells Python zu verwenden. Praktisch würde also der zu einem Objekt-Geste-Tupel gehörende Code eingelesen und an Pythons `eval`-Funktion übergeben. Die Übergabe an die `eval`-Funktion sorgt für ein direktes Ausführen des eingelesenen Quellcodes durch den Python-Interpreter.
Dadurch ist ein Ausgabemodul mit dieser Art von Aktionsmodell sehr einfach zu implementieren, da der Quellcode zu jedem Tupel nur eingelesen und übergeben werden muss. Zudem ist dieser Ansatz äußerst mächtig, da die gesamte Funktionalität einer turingvollständigen Hochsprache zur Verfügung steht.
Allerdings hat auch dieser frei programmierbare Ansatz eine Reihe von Nachteilen. Die Komplexität des Aktionsmodells verlagert sich vom System in die Konfigurationsdatei, wo nun für jede Objekt-Geste-Kombination Quellcode geschrieben werden muss. Dies bedeutet auch, dass für die Definition des Aktionsmodells Kenntnisse in der verwendeten Programmiersprache und der Schnittstellen zum Restsystem erforderlich sind. Zudem leidet die Robustheit, da der angegebene Quellcode Fehler enthalten kann. Bei einer turingvollständigen Sprache ist es aufgrund des Halteproblems im Allgemeinen nicht möglich, das Aktionsmodell formal auf seine Korrektheit zu überprüfen. Da das Aktionsmodell direkten Zugriff auf den Interpreter und dessen Ressourcen hat, kann es bei Fehlverhalten das Gesamtsystem stören. So kann fehlerhafter Quellcode im Aktionsmodell den Absturz des Gesamtsystems auslösen, ohne dass sich dies vorher erkennen ließe. Hier wird die hohe Flexibilität und die Einfachheit der Implementation des Aus-

gabemoduls also durch hohe Komplexität bei der Definition des Aktionsmodells und einem Mangel an Robustheit erkauft.

3.5.1.3 Endliche Automaten

Da die beiden in Abschnitt 3.5.1.1 und 3.5.1.2 vorgestellten Ansätze sich ihre jeweiligen Vorteile mit vielen Nachteilen erkaufen, sollte ein Mittelweg gefunden werden. Dieser sollte ausreichend mächtig und flexibel sein, ohne dass bei der Implementation des Ausgabemoduls oder bei der Definition des Aktionsmodells zu viel Komplexität entsteht. Des weiteren wäre es wünschenswert, wenn das Aktionsmodell auf einem soliden und bereits erprobten Ansatz basiert. Daher wird das Aktionsmodell über endliche Automaten realisiert.
Ein endlicher Automat[24] ist ein in der Informatik weit verbreitetes Verhaltensmodell, welches über das Tupel $(\Sigma, S, s_0, \delta, F)$ definiert ist. Dabei ist das Eingabealphabet Σ eine endliche, nicht-leere Menge von Symbolen. S ist die endliche, nicht-leere Menge an möglichen Zuständen, die der Automat annehmen kann. $s_0 \in S$ ist der Startzustand des Automatens. $\delta : S \times \Sigma \mapsto \Sigma$ ist die Zustandsübergangsfunktion, welche das dynamische Verhalten des Automaten durch Abbildung des momentanen Zustands und der Eingabe auf einen Folgezustand festlegt. $F \subseteq S$ ist die endliche, möglicherweise leere Menge der Endzustände. Erreicht der Automat einen Endzustand, so hält er an.
Diese Standarddefinition des endlichen Automatens wird nun für den Einsatz als Aktionsmodell angepasst. Da das Aktionsmodell nie anhalten soll, ist die Menge der Endzustände F immer leer. F ist daher nicht mehr Teil der angepassten Automatendefinition. Zudem muss das Aktionsmodell eine auszuführende Aktion liefern. Der momentane Zustand s könnte hier direkt als Ausgabe verwendet werden. Dies führt jedoch dazu, dass es keine verschiedenen Zustände $s_n \neq s_m$ geben kann, welche dieselbe Ausgabe haben. Zudem sind keine Zustände ohne Ausgabe, also ohne auszuführende Aktion, möglich. Daher wird eine Ausgabefunktion σ verwendet. Diese bildet den momentanen Zustand $s \in S$ auf eine Aktion ab, die bei Betreten des Zustands s ausgeführt wird[25]. Der für das Aktionsmodell angepasste Automat wird also über das Tupel $(\Sigma, S, s_0, \delta, \sigma)$ definiert.
Jedem Kanal eines jeden Smart Objects wird ein solcher modifizierter endlicher Automat zugeordnet. Die Automaten der einzelnen Objekte und Kanäle laufen dabei unabhängig voneinander. Als Eingabe dient die erkannte Objekt-Geste-Kombination in Form

[24] Im Englischen ist der endliche Automat als *Finite State Machine* bekannt.

[25] Es könnte auch die Zustandsübergangsfunktion um eine Ausgabekomponente auf $\delta : S \times \Sigma \mapsto \Sigma \times A$ erweitert werden, wobei A die Menge der möglichen Aktionen ist. Dieser Ansatz ist äquivalent zur Verwendung der Ausgabefunktion $\sigma : S \mapsto A$. Allerdings ist die Ausgabefunktion σ einfacher zu handhaben, da sich aus dem aktuellen Zustand s immer auf den zuletzt an den Kanal angelegten Wert schliessen lässt, sofern σ für den Zustand s definiert ist. Dies erleichtert die Suche nach Fehlern im Aktionsmodell. Bei der Angabe des Startzustands entfällt zudem eine separate Angabe des Startwerts für den Kanal.

des Tupels (O, G). Eingabealphabet Σ ist somit die Menge aller möglichen Objekt-Geste-Kombinationen. Nach der Erkennung einer Geste wird das resultierende Tupel (O, G) als Eingabe für jeden Automaten eines jeden Kanals verwendet. Führt dies entsprechend der Funktion δ und des momentanen Zustands s zu einem Zustandsübergang in den Zustand s' , so wird an den dazugehörigen Kanal die Ausgabe $\sigma(s')$ angelegt. Ist der Zustandsübergang $\delta(s, (O, G))$ für einen bestimmten Automaten nicht definiert, so ändert sich der Automatenzustand nicht. In diesem Fall bleibt auch der dazugehörige Kanal unverändert. Ist die Ausgabefunktion σ für den Nachfolgezustand s' nicht definiert, so ändert sich der am Kanal anliegende Wert nicht.
Der Ansatz der endlichen Automaten hat eine Reihe von Vorteilen. Endliche Automaten sind eine weit verbreitete und bekannte Art von Automaten. Sie erlauben eine robuste formale Definition des Aktionsmodells. Die strikte Definition des Automaten und seiner Bestandteile lässt sich leicht in einer XML-Konfigurationsdatei beschreiben[26]. Über eine Darstellung als Graph kann ein solches Aktionsmodell einfach visualisiert werden, wie zum Beispiel in den Abbildungen 3.5 und 3.6 geschehen. Sogar ein grafischer Editor für Aktionsmodelle wäre auf diesem Wege umsetzbar, auch wenn ein solcher Editor im Rahmen dieser Arbeit nicht verwendet wird. Die Definition des Aktionsmodells ist bei diesem Ansatz also in der Regel einfacher, als es mit einer Hochsprache aus Abschnitt 3.5.1.2 möglich wäre.

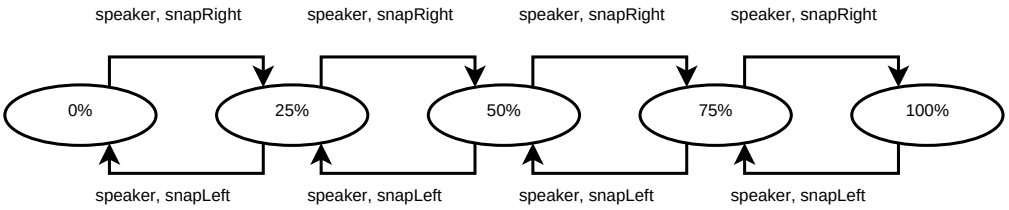

Abbildung 3.5: Endlicher Automat zur Lautstärkeregelung in grafischer Darstellung. Mittels der Gesten `snapRight` und `snapLeft` auf das Objekt `speaker` kann die Lautstärke über die Stufen 0%, 25%, 50%, 75% und 100% erhöht oder gesenkt werden. Die Gesten `snapRight` und `snapLeft` sind dabei als Beispiel Handbewegungen nach rechts oder links, das Objekt `speaker` ist ein Lautsprecher.

Die starre Struktur des Automatens schränkt die Menge der möglichen Fehler ein und vereinfacht die Fehlerbehandlung. So lässt es sich zum Beispiel leicht erkennen, wenn ein Automat einen nicht existierenden Zustand betritt. Zudem ließe sich der Automat falls notwendig vergleichsweise einfach auf seine Funktion hin analysieren. Dabei könnte unter anderem geprüft werden, ob alle definierten Zustände erreichbar sind, welche Ausgabewerte ein bestimmter Kanal im gegebenen Aktionsmodell annehmen kann und ob Zyklen oder Sackgassen, also Zustände ohne ausgehende Kanten, auftreten.

[26]Das Format dieser Konfigurationsdatei ist in Abschnitt 3.5.3 beschrieben.

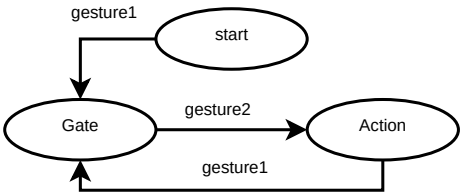

Abbildung 3.6: Beispielautomat zur Kombination von Gesten in grafischer Darstellung. Der Automat beginnt im Zustand `start`. Über die Geste `gesture1` wird der Zwischenzustand `Gate` erreicht. Wird dann die Geste `gesture2` ausgeführt, geht der Automat in den Zustand `Action` über. Die Ausgabefunktion σ ist nur für den Zustand `Action` definiert, so dass erst beim Erreichen dieses Zustands die Ausgabe geändert wird. Die Geste `gesture1` führt im Anschluss in den Zustand `Gate` zurück. Dieser Automat erzeugt also nur eine Ausgabe, wenn die Gesten `gesture1` und `gesture2` hintereinander ausgeführt werden.

Gleichzeitig erlaubt der endliche Automat komplexere Aktionsmodelle, als sie mit den in Abschnitt 3.5.1.1 beschriebenen elementaren Operationen möglich wären. So ist in Abbildung 3.5 eine Lautstärkeregelung zu sehen, die durch Handbewegungen gesteuert wird. Durch eine Geste in Richtung eines Lautsprechers mit der Objektbezeichnung `speaker` lässt sich die Lautstärke durch eine Handbewegung nach links (Geste `snapLeft`) oder rechts (Geste `snapRight`) in festgelegten Stufen senken oder erhöhen. Durch die Verwendung endlicher Automaten lässt sich dieses Aktionsmodell einfach erweitern. So könnte unter anderem die Audiowiedergabe automatisch pausiert werden, wenn bei 0% Läutstärke noch einmal die Geste `snapLeft` ausgeführt wird. Auch eine nichtlineare Lautstärkeskala zur feineren Steuerung der Lautstärke in bestimmten Bereichen wäre einfach zu implementieren. All dies würde bei Verwendung elementarer Operationen jeweils die Implementation eines speziellen Operators erfordern.
Auch Gestenkombinationen sind, wie in Abbildung 3.6 gezeigt, möglich. Dabei muss zuerst die Geste `gesture1` ausgeführt werden, bevor mit der nachfolgenden Geste `gesture2` der Zustand `Action` erreicht und der Ausgabewert verändert wird. Wenn sich die Gesten `gesture1` und `gesture2` auf zwei verschiedene Smart Objects beziehen, lassen sich Verbindungen zwischen Objekten darstellen. So kann durch eine Geste auf ein Abspielgerät mit anschließender Geste auf ein Ausgabegerät das Signal des Abspielgeräts auf dem Ausgabegerät wiedergegeben wird. Ein- und Ausgabegeräte könnten zum Beispiel ein Heimkino-PC mit angeschlossenem Fernseher und Lautsprechern sein.
Natürlich ist auch der Ansatz der endlichen Automaten nicht frei von Nachteilen. So sind endliche Automaten nicht so mächtig wie eine turingvollständige Programmiersprache. Zugleich sind endliche Automaten komplexer als elementare Operationen. Insgesamt bieten endliche Automaten meiner Ansicht nach einen guten Kompromiss zwischen den in Abschnitt 3.5.1.1 und 3.5.1.2 dargestellten Extremfällen.

3.5.2 Implementation

Das Aktionsmodell in der Datei `actions.py` implementiert. Diese Datei stellt eine `ActionHandler`-Klasse zur Verfügung, welche die gesamte Aktionsverarbeitung des Systems über die Methode `handleInput` realisiert. Eine `ActionHandler`-Instanz beinhaltet ein oder mehrere Instanzen der `SmartObject` Klasse. Die `SmartObject`-Klasse modelliert dabei jeweils genau ein im Netzwerk ansprechbares Smart Object. Jede der `SmartObject`-Instanzen umfasst wiederum ein oder mehrere `SmartChannel`-Instanzen, welche die Kanäle des *Smart Objects* modellieren. Jede `SmartChannel`-Instanz besitzt genau eine Instanz der `StateMachine`-Klasse, welche den in Abschnitt 3.5.1.3 beschriebenen endlichen Automaten für diesen Kanal umsetzt. Dazu werden die `State`- und `Edge`-Klassen verwendet, welche jeweils die Zustände und Kanten des endlichen Automatens implementieren.
Jedem `SmartChannel` ist dabei ein Datentyp zugewiesen. Bei den Datentypen handelt es sich um eine Teilmenge der MIRIAM-Datentypen [15]. Erlaubt sind `number` (Dezimalzahlen), `text` (ASCII-Text), `switch` (Binär, `on` oder `off`) und `signal` (Zeitstempel). Der Signal-Datentyp ist ein Sonderfall, da hier im endlichen Automaten beim Betreten eines Zustands nicht der Ausgabewert des Knotens ausgegeben wird. Stattdessen wird der momentane UNIX-Zeitstempel als Ausgabe verwendet. Zustände ohne Ausgabewert führen jedoch auch bei einem `signal`-Kanal zu keiner Ausgabe.
Wird die `handleInput`-Methode der `ActionHandler`-Klasse mit einem Tupel aus Objekt- und Gestenname aufgerufen, so ändern sich die Zustände der einzelnen endlichen Automaten entsprechend des Aktionsmodells. Kommt es dabei zu einer Änderung der Ausgabe, wird der neue Ausgabewert an das entsprechende Smart Object übertragen.

3.5.2.1 Smart Object-Anbindung

Zur Kommunikation mit der MIRIAM-Netzwerkinfrastruktur werden die Klassen `Client`, `RemoteObject` und `RemoteChannel` aus dem MIRIAM-Paket `networking` verwendet [15]. Dazu muss zunächst eine Instanz der `Client`-Klasse initialisiert und über die `connect` und `start`-Methoden mit dem Netzwerk verbunden werden. Danach kann die `Client`-Instanz entweder bei der Instanziierung der `SmartObject`-Klasse (siehe Abschnitt 3.5.2) oder beim Laden eines `ActionHandler` mittels der `loadActionHandler`-Funktion (siehe Abschnitt 3.5.3) übergeben werden. Wird statt eines `Client` hier der Wert `None` übergeben, so wird ein Testmodus aktiviert. Im Testmodus findet keine Kommunikation mit dem Netzwerk statt. Stattdessen werden die generierten Nachrichten auf der Standardausgabe (`stdout`) ausgegeben. Dies erlaubt den Test des Systems ohne Netzwerk und ohne reale Smart Objects.
Die `SmartObject`-Instanzen versuchen, über die bereitgestellte `Client`-Instanz ein

RemoteObject zu erhalten. Dabei ist der Name des SmartObject auch der Netzwerkname des RemoteObjects. Über dieses RemoteObject werden dann die zu den SmartChannels gehörenden RemoteChannels abgefragt. Wird nun die setOutput- Methode einer SmartChannel-Instanz aufgerufen, so verwendet diese den dazugehörigen RemoteChannel, um die Ausgabe über das Netzwerk zu versenden.
Wird ein Objekt oder Kanal nicht auf dem XMPP-Server gefunden, so wird eine Warnung ausgegeben und für dieses Objekt oder diesen Kanal der Testmodus aktiviert. Die Ergebnisse des Aktionsmodells für das betroffende Objekt oder den betroffenden Kanal werden dann über die Standardausgabe in Textform ausgegeben. Dabei ist zu beachten, dass Objekte noch im Netz registriert sein können, auch wenn das Smart Object nicht mehr vorhanden ist. In diesem Fall wird der Testmodus nicht aktiviert, da das Objekt noch auf dem Server gefunden wird.

3.5.3 Konfiguration

Zur Konfiguration des Aktionsmodells wird eine XML-Konfigurationsdatei verwendet. Das Schema ist in der XSD-Datei actions.xsd beschrieben. Das <actions>-Element bildet die Wurzel des XML-Dokuments und umfasst eine beliebige Anzahl an <smartobject>-Elementen. Jedes <smartobject>-Element beschreibt dabei eine Instanz der in Abschnitt 3.5.2 beschriebenen SmartObject-Klasse. Die Attribute name und label sind vom Typ string (ASCII-Zeichenketten). Das Attribut name ist der Netzwerknamen des Smart Objects. Das optionale Attribut label enthält den menschenlesbaren Namen des Objekts. Jedes <smartobject>-Element hat eine beliebige Anzahl an <channel>-Elementen, die jeweils eine Channel-Instanz beschreiben. Die Attribute name und type beschreiben den Namen und Datentyp des Kanals im string-Format. Erlaubte Datentypen sind dabei text, number, signal und switch.
Jedes <channel>-Element beinhaltet genau ein <statemachine>-Element. Dieses Element beschreibt einen endlichen Automaten in Form einer Instanz der StateMachine-Klasse. Das Attribut initial enthält den Namen des Startzustands des Automatens s_0. Jedes <statemachine>-Element beinhaltet einen oder mehr <state>-Elemente, welche die Zustände des endlichen Automaten beschreiben. Das name-Attribut eines <state>-Elements beschreibt dabei den Namen des Zustands. Dabei müssen die Namen der Zustände innerhalb eines Automaten einzigartig sein, identische Zustandsnamen innerhalb eines Automaten sind nicht erlaubt. Zudem muss zumindest der im initial-Attribut genannte Anfangszustand s_0 definiert sein. Jeder Zustand umfasst ein optionales <output>-Element und beliebig viele <action>-Elemente. Das optionale <output>-Element gibt den Ausgabewert des Zustands an. Die Gesamtheit der <output>-Elemente eines Automaten beschreibt somit implizit die in Abschnitt 3.5.1 verwendete Ausgabefunktion σ. Die <action>-Elemente be-

schreiben die ausgehenden Kanten eines Zustands. Das Attribut `next` beinhaltet den Namen des Folgezustands s'. Die Elemente `<object>` und `<gesture>` legen fest, bei welchem Objekt-Geste-Eingabetupel (O, G) die Kante aktiviert und der Folgezustand s' erreicht wird.
Die XML-Konfigurationsdatei wird über das `actionConfig.py`-Modul geladen. Dieses Modul stellt die Funktion `loadActionHandler` zur Verfügung. Diese liest eine angegebene XML-Konfigurationsdatei mit dem oben beschriebenen Format ein. Unter Verwendung einer übergebenen `Client`-Instanz wird eine der Konfigurationsdatei entsprechende `ActionHandler`-Instanz erzeugt und zurückgegeben. Wird an Stelle der `Client`-Instanz der Wert `None` übergeben, wird wie in Abschnitt 3.5.2.1 beschrieben der Testmodus verwendet.

3.6 Integration

Die beschriebenen Module müssen miteinander zu einem Gesamtsystem verbunden werden. Dazu wurde die Klasse `niReceiver` in der Datei `openniReceiver.py` implementiert. Bei der Instanziierung dieser Klasse können die Namen der Konfigurationsdateien übergeben werden. Aus diesen werden dann die Teilmodule des Systems initialisiert. Dazu gehören eine Schnipserkennung (siehe Abschnitt 3.4), jeweils ein Zeigemodul (Abschnitt 3.2) für die linke und rechte Hand, ein Gestenmodul (siehe Abschnitt 3.3) und ein MIRIAM-Netzwerkclient [15].
Nach der Initialisierung der Teilmodule muss die `start`-Methode aufgerufen werden. Erst mit Aufruf dieser Methode werden die Eingabemodule gestartet. Zudem wird erst hier die Verbindung zu den Ein- und Ausgabekanälen hergestellt. Dazu gehört der Start des Kameramoduls als Unterprozess, der Verbindungsaufbau durch den MIRIAM-Client und der Start des Threads zur Schnipserkennung. Das Aktionsmodell wird erst in dieser Methode initialisiert, da dabei die Initialzustände der Smart Objects über das Netzwerk gesetzt werden.
Im Anschluss betritt die Klasse eine Hauptschleife. In jedem Durchlauf der Hauptschleife werden dabei zuerst in der Methode `getData` neue Daten vom Eingabemodul empfangen und von der C-Datenstruktur (siehe Abschnitt 3.1.2.1) in Python-Datenstrukturen übertragen. Damit Daten empfangen werden können, muss die Skelettanalyse zunächst entsprechend Abschnitt 3.1.2 für den Benutzer kalibriert werden.
Nach der Aktualisierung der Daten werden in der `handleData`-Methode die Teilmodule aktualisiert. Wird momentan keine Geste aufgenommen, werden die Zeigemodule für beide Hände aktualisiert. Wird bereits eine Geste aufgenommen, werden die neuen Positionsdaten an das Gestenmodul übergeben. Wird die Gestenaufnahme durch die neuen Positionsdaten abgeschlossen, wird die Geste über das Gestenmodul klassifiziert. Das Ergebnis der Klassifikation wird zusammen mit dem in der Variable `lastObject` gespeicherten Zielobjekt an das Aktionsmodell übergeben.

Da das Audiomodul in einem eigenen Thread läuft, kann es die Hauptschleife unterbrechen, wenn ein Schnipsen erkannt wird. Dabei wird die Callback-Methode `handleSnap` aufgerufen. Mittels eines Python-`Locks` wird sichergestellt wird, dass es nicht zu Konflikten zwischen den Threads kommt. Sofern momentan keine Geste aufgenommen wird, wird in der `handleSnap`-Methode über die Zeigemodule geprüft, ob und mit welcher Hand gerade auf ein gültiges Objekt gezeigt wird. Wird momentan bereits eine Geste aufgenommen, so wird das Schnipsen ignoriert.
Zeigt der Benutzer auf ein gültiges Objekt, so werden das getroffene Objekt und die dazu verwendete Hand als `lastObject` und `classifyHand` gespeichert. Wird mit beiden Händen auf ein Objekt gezeigt, so wird die linke Hand[27] verwendet. Nachdem Zielobjekt und Hand gespeichert wurden beginnt die Aufnahme der Geste. Dabei wird zu Beginn der Geste ein Ton aus der Datei `sound-begin.wav` abgespielt, um den Benutzer über den Start der Geste zu informieren (siehe Abschnitt 3.7.3). Nach Ende der Gestenaufnahme wird zudem ein weiterer Ton aus der Datei `sound-end.wav` abgespielt.
Während des Ablaufs werden zur Diagnose Statusmeldungen über die Programmvorgänge auf der Standardausgabe ausgegeben. Das Programm läuft so lange weiter, bis es mit der Tastenkombination `Ctrl-C` oder einem entsprechenden Signal unterbrochen wird. Bei Unterbrechung werden zunächst die Teilmodule und Kindprozesse und im Anschluss das Programm selbst beendet.

3.7 Hilfsmodule

Bei der Implementation der Arbeit wurden einige Hilfsmodule implementiert, die zu keinem der Hauptmodule gehören. Diese sollen hier kurz in ihrer Funktion beschrieben werden, wobei Details dem beiliegendem Quellcode entnommen werden können.

3.7.1 XML-Hilfsmodul

Das XML-Hilfsmodul befindet sich in der Datei `xmlutility.py`. Dieses Modul stellt Hilfsfunktionen zum Auslesen von XML-Dateien bereit. Dabei wird die *Domain-Object-Model*-Implementation aus dem Python-Standardmodul `xml.dom.minidom` verwendet. Unter anderem werden Funktionen bereitgestellt, die anhand des Namens den Wert eines Elements (Funktion `getElementValue`) oder eines Attributs (Funktion `getAttribute`) ermitteln. Zusätzlich gibt es Varianten dieser Funktionen, die bei fehlenden Elementen oder Attributen einen Fehler auslösen (`getRequiredElementValue` und `getRequiredAttribute`).

[27] Hierbei handelt es sich um die linke Hand aus Sicht der Kamera. Dies entspricht der rechten Hand aus Sicht des Benutzers, wenn sich der Benutzer bei der Kalibrierung wie vorgesehen der Kamera zugewendet hat.

3.7.2 Ringspeicher

Zur Aufnahme von Daten wurde ein Ringspeicher als Klasse `RingBuffer` in der Datei `ringbuffer.py` implementiert. Ein Ringspeicher ist dabei eine Datenstruktur, die eine feste Anzahl an Einträgen speichern kann. Werden mehr Einträge über die `append`-Methode hinzugefügt, als der Ringspeicher aufnehmen kann, so wird jeweils der älteste Eintrag aus dem Ringspeicher entfernt und durch den neuen Eintrag ersetzt. Über die Methoden `length`, `empty` und `full` kann der Füllstand des Ringspeichers geprüft werden. Zudem lässt sich der Ringspeicher über die Methode `getLinearBuffer` als Liste ausgeben, wobei die Einträge der Liste nach ihrem Alter sortiert sind. Der älteste Eintrag im Ringspeicher ist der erste, der jüngste Eintrag hingegen der letzte Eintrag in der Liste.

3.7.3 Modul zur Audiowiedergabe

Um Signaltöne wiedergeben zu können, wurde die Klasse `audioPlayer` in der Datei `audioPlayer.py` implementiert. Die Klasse verwendet die bereits zur Tonaufnahme eingesetzte PyAudio-Bibliothek. Die Methode `playFile` nimmt den Namen einer WAV-Audiodatei entgegen und spielt diese auf dem Standard-Ausgabekanal des Rechners ab. Die Methode kehrt erst zurück, wenn die Wiedergabe abgeschlossen ist. Die Methode `playFileThreaded` bietet dieselbe Funktionalität in einem separaten Thread. Dadurch wird der Ton wird im Hintergrund abgespielt, die Methode selbst kehrt sofort zurück.

3.7.4 Zusätzliche Smart Objects

Im Rahmen der Arbeit wurden zwei weitere Smart Objects für das MIRIAM-System [15] implementiert. Diese befinden sich im Unterverzeichnis `src/objects/`.

3.7.4.1 Objekt für Mehrfachsteckdosen

Das `Outlet`-Objekt aus der Datei `outlet.py` dient zur Steuerung von netzwerkfähigen Mehrfachsteckdosen vom Typ *NET-PowerControl PRO* der Firma Anel Elektronik. Das Objekt stellt die Kanäle `outlet0` bis `outlet7` mit dem Datentyp `switch` zur Verfügung. Damit lassen sich die acht Steckdosen der Mehrfachsteckdose an- und ausschalten.
Die Netzwerkadresse der Steckdosenleiste wird über die globale Variable `OUTLET_IP` festgelegt. Dabei muss der Direktzugriff auf Seite der Mehrfachsteckdose aktiviert sein. Der Name und das Passwort, mit dem sich das Smart Object am XMPP-Server anmeldet, befindet sich in den globalen Variablen `USER` und `PASSWORD`. Die Adresse des XMPP-

Servers ergibt sich, wie bei MIRIAM üblich, aus dem MIRIAM-Konfigurationsmodul `config` [15].

3.7.4.2 Objekt für die Musikwiedergabe

Die Klasse `MPDControl` aus der Datei `mpd-object.py` implementiert ein Objekt zur Steuerung eines *Music Player Daemon* zur Wiedergabe von Musik. Dazu wurde die Bibliothek `python-mpd` verwendet [4]. Das Objekt stellt die Kanäle `next`, `prev`, `play`, `pause` und `playpause` vom Datentyp `signal` bereit. Mittels eines Signals an die Kanäle `next` und `prev` springt der Daemon zum nächsten oder vorherigen Titel auf der aktuellen Wiedergabeliste. Der Kanal `pause` hält die Wiedergabe an, während der Kanal `play` die Wiedergabe fortsetzt. Ein Signal an den Kanal `playpause` schaltet die Wiedergabe zwischen diesen beiden Zuständen um. Zusätzlich existiert der Kanal `volume` vom Typ `number`. Über diesen Kanal kann die Lautstärke der Wiedergabe als Wert zwischen 0 und 100 verändert werden.
Adresse, Port und Passwort des MPD-Servers werden in den globalen Variablen `MPD_HOST`, `MPD_PORT` und `MPD_PASS` festgelegt. Die Konfiguration der XMPP-Verbindung erfolgt analog zum Steckdosen-Modul aus Abschnitt 3.7.4.1 über die globalen Variablen `USER` und `PASSWORD` und das MIRIAM-Modul `config`.

4

Abschluss

In diesem Kapitel finden sich Tests des Systems mit anschließendem Fazit. Zuletzt wird noch ein Ausblick auf mögliche Verbesserungen und Erweiterungen gegeben.

4.1 Tests

Zur Prüfung des Systems wurden einige einfache Tests durchgeführt. Dabei wurden die Klassifikatoren für die Schnips- und Gestenerkennung von Versuchspersonen getestet. Dafür wurden eine Kinect-Kamera und ein omnidirektionales Mikrofon nebeneinander aufgestellt. Die Versuchspersonen durften ihre Positionen innerhalb des Blickfelds der Kamera frei wählen, der Abstand zwischen der Person und den Sensoren betrug wegen der Größe des Raumes zwischen ein und drei Metern. Zu Beginn jedes Versuchs wurde das System der Versuchsperson erklärt und vorgeführt. Dann folgte eine Eingewöhnungsphase, in der die Versuchsperson das System im Normalbetrieb beliebig lang benutzen durfte. Die Länge der Eingewöhnungsphase überschritt dabei nie fünf Minuten.

4.1.1 Test der Schnipserkennung

Zum Test der Schnipserkennung wurden zwei Szenarien verwendet. Im ersten Szenario wurde der Klassifikator mit konstanten Prototypen verwendet, die vor dem Test von einer einzelnen Person aufgenommen wurden. Im zweiten Szenario wurde der Klassifikator zunächst von jeder Versuchsperson mittels 20–25 Schnipslauten trainiert und dann durch dieselbe Person getestet. Die Größe des Analysefensters lag bei 50 ms. Der

Szenario	Präsentiert	Erkannt	True-Positive-Rate
Szenario 1	40	13	32.5%
Szenario 2	40	36	90.0%

Tabelle 4.1: Tabelle mit den Testergebnissen der Schnipserkennung für $N = 4$ Versuchspersonen. Jede Versuchsperson durchlief jedes Szenario zehnmal. Die Spalte „Präsentiert“ gibt an, wie oft von den Versuchspersonen geschnipst wurde. Die Spalte „Erkannt“ enthält die Anzahl der dabei erkannten Schnipslaute. Die True-Positive-Rate ist der Quotient aus den erkannten Schnipslaute und der Zahl der präsentierten Schnipslaute.

Schwellwert war $\theta = 40$, die Slackvariable war $\lambda = 1.0$ für Szenario 1 und $\lambda = 1.2$ für Szenario 2.
Bei den Tests wurden pro Szenario und Versuchsperson jeweils zehn Schnipslaute präsentiert und die Klassifikation „schnipsen“ oder „nicht schnipsen“ notiert. Die Anzahl der Versuchspersonen lag bei $N = 4$. Eine fünfte Versuchsperson war nicht in der Lage zu schnipsen. Die Ergebnisse finden sich in der Tabelle 4.1.

4.1.2 Test der Gestenerkennung

Für die Gestenerkennung wurden zwei Testobjekte mit insgesamt sechs Gesten verwendet. Das erste Testobjekt war ein Tischventilator (Bezeichner `fan`), der sich etwas neben der Kamera befand. Wendet sich der Benutzer in einer typischen Entfernung der Kamera zu, befand sich dieses Testobjekt aus Sicht des Benutzers im vorderen Quadranten. Das zweite Testobjekt war eine Bodenlampe, die sich rechts vom Benutzer befand, wenn dieser auf die Kamera blickt. Die Positionen der Objekte wurden über das in Abschnitt 3.2.4 beschriebene Verfahren eintrainiert. Die exakten Positionen und Ausdehnungen der Objekte lassen sich aus der mitgelieferten Konfigurationsdatei `src/pointing.xml` mit dem Format aus Abschnitt 3.2.3 entnehmen.
Die sechs verwendeten Gesten sind in Abbildung 4.1 beschrieben und erklärt. Die Gesten dabei vor Beginn der Tests von einer einzelnen, anderen Person eintrainiert. Pro Geste wurden entweder zwei (für die Gesten `circleCW`, `circleCCW` und `right` oder drei (für die Gesten `left`, `up` und `down`) Trainingsbeispiele aufgenommen. Die Menge der Trainingsbeispiele wurde absichtlich klein gehalten, um die Dauer der Gestenanalyse zu verringern. Die `recenter`- und `rescale`-Funktionen aus Abschnitt 3.3.3 waren aktiviert. Die Gestenlänge betrug 40 Kamerabilder, die Rückweisung war deaktiviert.
Jede der $N = 5$ Versuchsperson führte jede der Gesten nun auf jedes der beiden Objekte aus. Jede Versuchsperson durfte dabei frei wählen, welcher Arm verwendet wird. Der eingesetzte Arm durfte zwischen zwei Gesten gewechselt werden. Die Ergebnisse finden sich in der Tabelle 4.2.

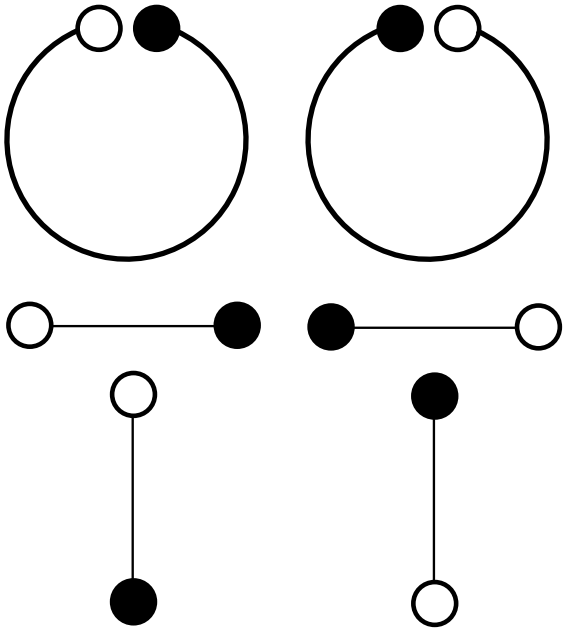

Abbildung 4.1: Schematische Darstellung der in den Tests verwendeten Gesten aus Sicht des Benutzers. Der hohle Kreis stellt die Startposition der Hand, der gefüllte Kreis die über die Linie erreichte Endposition der Hand dar. Die Gesten wurden dabei mit ausgestrecktem Arm ausgeführt. Zu sehen sind (in Leserichtung) die Gesten `circleCCW`, `circleCW`, `right`, `left`, `down` und `up`.

4.1.3 Diskussion

Aus Tabelle 4.1 geht hervor, dass sich Schnipslaute zwischen verschiedenen Personen wesentlich unterscheiden. Daher reicht es nicht aus, den Klassifikator mit den Aufnahmen einer einzelnen dritten Person zu trainieren. Wird der Klassifikator bei einer leicht erhöhten Slackvariable λ für eine bestimmte Testperson trainiert, zeigen sich für diese Person bessere Ergebnisse.
Die Gestenerkennung zeigt in Tabelle 4.2 eine bessere Generalisierungsleistung. Die Geste `down` zeigte dabei eine niedrigere Erkennungsrate als die anderen Gesten. Ein Grund hierfür könnte sein, dass die Trainingsdaten für diese Geste eventuell eine schlechtere Qualität haben.
Der durchgeführte einfache Test erlaubt aus verschiedenen Gründen jedoch nur wenige Aussagen über die Leistungsfähigkeit des Systems. Die Anzahl der Versuchspersonen war niedrig. Die Zusammensetzung der Versuchspersonen war nicht repräsentativ für die Menge der in der Praxis denkbaren Benutzer. Zudem wurde das System nur mit wenigen Trainingsdaten und von jeweils nur einer Person trainiert. Die sonstigen Parameter des Systems wurden per Hand gewählt, die Qualität der gewählten Parameter

Geste	Präsentiert	Erkannt	True-Positive-Rate
`circleCW`	50	49	98.0%
`circleCCW`	50	49	98.0%
`left`	47	42	89.4%
`right`	45	45	100.0%
`up`	42	42	100.0%
`down`	43	33	76.7%

Tabelle 4.2: Tabelle mit den Testergebnissen der Gestenerkennung für $N = 5$ Versuchspersonen. Die Gesten wurden jeweils gleich häufig in Richtung der beiden Objekte ausgeführt. Die Anzahl der präsentierten Gesten ist für die Gesten `up`, `down`, `left` und `right` niedriger, da zwei Versuchspersonen weniger als die geplanten 10 Tests für jede dieser Gesten durchliefen. Die Spalten „Präsentiert“ und „Erkannt“ geben an, wie oft die jeweilige Geste ausgeführt und vom System korrekt erkannt wurde. Die True-Positive-Rate ist der Quotient dieser beiden Werte.

selbst wurde nicht systematisch untersucht. Auch wurde das Testverfahren selbst nicht im Rahmen einer Vorstudie auf seine Tauglichkeit überprüft.
Zudem wurden dem System keine Negativbeispiele präsentiert. Dafür ist ein Katalog aus Negativbeispielen, also der im Alltagseinsatz möglichen Geräusche und Gesten, nötig. Ein solcher Katalog konnte jedoch im Rahmen dieser Arbeit nicht erstellt werden. Durch die fehlende quantitative Information über die Anzahl der falschen Positiven lässt sich die tatsächliche Erkennungsleistung der Klassifikatoren schwer beurteilen.
Um das System besser zu testen wäre eine umfangreiche, vermutlich mehrstufige Benutzerstudie notwendig. Ein Ansatz für bessere eine Studie findet sich in Abschnitt 4.3.8.

4.2 Fazit

In diesem Abschnitt werden die Ergebnisse der Arbeit mit den ursprünglichen Plänen verglichen und ein Fazit gezogen.

4.2.1 Eingabemodul

Die Implementation des Moduls als C-Programm auf Basis der OpenNI-Bibliothek erwies sich als weitestgehend problemlos. Auch die Kontrolle des Prozesses und die Kommunikation über UNIX-Pipes war mit den Python-Standardmodulen `struct` und `process` gut möglich. Einzig das korrekte Herunterfahren der OpenNI-Bilbiothek bei unerwartetem Programmabbruch erwies sich als schwieriger als erwartet. Die Skelettanalyse der NITE-Middleware lieferte, unter den in Abschnitt A.4 aufgeführten Einschränkungen, ausreichend gute Ergebnisse. Störend war dabei die oft erforderliche Kalibration.

4.2.2 Zeigemodul

Der zur Zeigeerkennung verwendete Klassifikationsbaum zeigte in der Praxis das erwartete Verhalten und konnte die Zeigepositur des Benutzers gut erkennen. Auch die Objekterkennung funktionierte zuverlässig, auch wenn die letztendlich verwendeten Projektions- und Integralmethoden vom anfänglichen Ansatz abweichen.
Das Trainieren der Objektpositionen über Zeigen funktionierte in der Praxis gut. Das Verfahren zum Trainieren der Objektgröße und -ausrichtung funktioniert zuverlässig, ist aber auf eine Näherung durch kugelförmige Verteilungen beschränkt (siehe Abschnitt 4.3.2.1).

4.2.3 Gestenmodul

Nachdem sich das räumliche Punktmerkmal als untauglich erwies, wurde stattdessen das einfachere Merkmal der normierten Handposition verwendet. Dabei konnten mit dem DTW-Nächster-Nachbar-Klassifikator akzeptable Ergebnisse (siehe Tabelle 4.2) erziehlt werden. Allerdings ist der DTW-Ansatz in seiner momentanen Python-Implementation recht langsam, was im Test zu Verzögerungen von etwa 500 Millisekunden führte (siehe Abschnitt 4.3.3). Die Rückweisung ist in der Praxis schwer zu benutzen, da der Schwellwert stark von den eintrainierten Gesten abhängt und daher für jeden Gestenkanon experimentell bestimmt werden muss.

4.2.4 Audiomodul

Die Aufnahme und Ausgabe von Audiodaten über die PyAudio-Bibliothek funktionierte wie erwartet. Da PyAudio jedoch keine nicht-blockierende Audioausbgabe unterstützt, musste zusätzlich das in Abschnitt 3.7.3 beschriebene Abspielmodul mit mehreren Threads implementiert werden.
Die in Abschnitt 3.4.1.1 beschriebene Transientenerkennung funktionierte in der Praxis ausreichend gut. Der in Abschnitt 3.4.1.2 vorgestellte Klassifikator zeigt ein schlechteres Verhalten als erhofft. Insbesondere erkennt der Klassifikator in Umgebungen mit vielen Störgeräuschen mehr falsche Positive als erwartet. Dies wird in den meisten Fällen durch die Multimodalität abgefangen, da der Benutzer zum Zeitpunkt der falschen Positiven nicht zeigt. Einige falsche Positive führen aber dennoch zu einer ungewollten Gestenaufnahme und -erkennung. Hier müsste, wie in Abschnitt 4.3.4 beschrieben, ein anderer Klassifikator verwendet werden. Dies war im Rahmen dieser Arbeit jedoch nicht mehr möglich.

4.2.5 Ausgabemodul

Das Ausgabemodul wurde ohne nennenswerte Probleme entsprechend Abschnitt 3.5 umgesetzt. Der Ansatz der endlichen Automaten war mächtig genug, um alle vorhandenen Smart Objects (IP-Steckdose, Musikwiedergabe) zu steuern[1]. Auch die Kommunikation mit den Smart Objects über das bereits im MIRIAM-System vorhandene `networking`-Modul und XMPP funktionierte ohne nennenswerte Probleme.

4.3 Ausblick

Im Rahmen dieser Arbeit wurden eine Reihe von Punkten erkannt, an denen das System verbessert oder weiter ausgebaut werden könnte. Mögliche Erweiterungen und Verbesserungen werden daher in diesem Abschnitt kurz dargestellt, um einen Ausblick auf eventuelle Weiterentwicklungen geben zu können.

4.3.1 Eingabemodul

4.3.1.1 Kalibrierung

Die vom Eingabemodul verwendete NITE-Skelettanalyse erfordert bei jeder Verwendung eine erneute Kalibrierung des aktiven Benutzers. Wird die Skelettanalyse unterbrochen, ist wiederum eine Kalibrierung erforderlich. Die Skelettanalyse kann unterbrochen werden, wenn der Benutzer das Bild verlässt. Auch längere Verdeckungen des Benutzers können eine erneute Kalibrierung erforderlich machen.
Die notwendige Kalibrierung ist deshalb problematisch, da der störende Kalibrationsvorgang die intuitive Benutzung des Systems unterbricht. Da die Skelettanalyse und somit auch die Kalibrierung feste Bestandteile der properitären NITE-Middleware sind, kann dieser Teil des Systems nicht ohne weiteres verändert werden. Nur der Einsatz einer Skelettanalyse ohne merkliche Kalibrierung würde Abhilfe schaffen. Das Microsoft-Kinect-SDK ist eine alternativer Ansatz zum Zugriff auf Kinect-Kameras. Dieses SDK benötigt zur Skelettanalyse keine Kalibrierungspose. Da das SDK aber erst gegen Ende dieser Arbeit veröffentlicht wurde und sich auf Windows-Plattformen beschränkt, konnte es nicht mehr als Teil eines alternativen Eingabemoduls getestet werden [3].

4.3.1.2 Mehrbenutzerbetrieb

Das Eingabemodul kann auf mehr als einen Benutzer erweitert werden. Die nachfolgenden Verarbeitungsmodule (Zeigeerkennung, Gestenerkennung und Aktionsmodell)

[1] Ein Beispielautomat hierfür findet sich in der mitgelieferten Konfigurationsdatei `src/actions.xml`.

lassen sich beliebig oft und für beliebig viele Benutzer instanziieren. Sie können daher ohne größere Veränderungen für mehrere Benutzer verwendet werden. Das Eingabemodul und die Kommunikation mit dem Restsystem müssten jedoch entsprechend angepasst werden.

4.3.1.3 Verwendung mehrerer Kameras

Die Verwendung nur einer Tiefenkamera schränkt die mögliche Position und Orientierung des Benutzers ein. Daher könnten mehrere Kameras verwendet werden, um größere Bereiche des Raumes abzudecken und den Benutzer stets aus einem geeigneten Winkel aufnehmen zu können. Allerdings wurde die Kombination mehrerer Kameras zur Skelettanalyse eines Benutzers zum Zeitpunkt dieser Arbeit noch nicht von NITE unterstützt. Für eine von NITE unabhängige Lösung des Problems müssten die Gelenkpositionen aus mehreren Kameras innerhalb des Systems miteinander vereint werden. Dazu werden die Positionen aus den jeweiligen Kamerakoordinaten in ein globales Koordinatensystem überführt. Dies ist nur möglich, wenn die Positionen und Orientierungen der Kameras relativ zueinander bekannt ist. Zudem muss die Skelettanalyse dann für jede Kamera getrennt kalibriert werden. Verlieren eine oder mehrere Kameras die Kalibrierung, muss dem Benutzer mitgeteilt werden, welcher Kamera er sich für die erneute Kalibrierung zuwenden muss. Da die Kinect-Kameras auf Basis eines von jeder Kamera ausgesendeten Infrarotmusters arbeiten, können sich Kinect mit überlappendem Blickfeld gegenseitig stören.
Aufgrund dieser Probleme beschränkt sich diese Arbeit auf nur eine Kinect-Kamera, zumal auch mit nur einer Kamera eine ausreichend gute Abdeckung gegeben war. Sollen mehrere Kameras verwendet werden, so müsste nur das Eingabemodul verändert werden, da die nachfolgenden Module nur die letztendlich berechneten Gelenkpositionen verwenden.

4.3.2 Zeigemodul

Um zu erkennen, ob der Benutzer gerade zeigt, verwendet das Zeigemodul unter anderem den Höhenwinkel der Zeigerichtung α (siehe Gleichung 3.4). Dieser Ansatz kann unter bestimmten Umständen scheitern. Versucht der Benutzer zum Beispiel, das System im Liegen zu benutzen, so gilt auch bei am Körper anliegenden Armen $\alpha \approx 0$ und es würde eine Zeigepositur erkannt. Um dies zu vermeiden, müsste die Zeigerichtung relativ zur Lage des Benutzers betrachtet werden. Eine Möglichkeit wäre, den Entscheidungsbaum der Zeigeerkennung um eine Mindestgröße für den Winkel zwischen Zei-

gerichtung und Torso zu erweitern. Mit der Zeigerichtung $\mathbf{v_Z}$ und der Ausrichtung des Torsos $\mathbf{t}$ ergäbe sich die Anforderung

$$\beta = \arccos\left(\frac{\mathbf{v_Z}^T\mathbf{t}}{\|\mathbf{v_Z}\|\|\mathbf{t}\|}\right) > \beta_l \tag{4.1}$$

Mit dieser zusätzlichen Anforderung könnten unter anderem falsche Positive beim Liegen oder anderen nicht aufrechten Körperhaltungen vermieden werden. Die Ausrichtung $\mathbf{t}$ lässt sich aus der Strecke zwischen Schulter- und Hüftposition abschätzen.
Dazu müssten die Hüften bei der Skelettanalyse mit erfasst werden. Momentan verwendet das System nur die Positionen des Kopfes, der Schultern, der Ellbogen und der Hände. Um die Hüftposition erfassen zu können, müsste das Eingabemoduls vom OpenNI-Oberkörperprofil (Im Programmcode als `XN_SKEL_PROFILE_UPPER`) auf ein Ganzkörperprofil (`XN_SKEL_PROFILE_ALL`) umgestellt werden. Bei einem Ganzkörperprofil muss zur Kalibrierung dann der gesamte Körper des Benutzers sichtbar sein. Auch müssten die Hüften des Benutzers sichtbar sein, um die Ausrichtung des Torsos bestimmen zu können. Alternativ könnte auch die Strecke zwischen Hals und Torsomittelpunkt als Näherung für die Ausrichtung des Torsos verwendet werden. Diese Strecke lässt sich bereits aus dem Oberkörperprofil berechnen. Da dieses zusätzliche Kriterium aber nur bei einigen Sonderfällen (zum Beispiel im Liegen) zum Einsatz kommen würde, wurde es im Rahmen dieser Arbeit nicht implementiert.

4.3.2.1 Festlegen der Objektausdehnung durch Zeigen

Wie in Abschnitt 3.2.4.2 beschrieben, können momentan nur kugelförmige als Objektverteilungen trainiert werden. Für die bisher verwendeten Objekte (Lampe, Ventilator, et cetera) war dies eine ausreichend gute Näherung. Bei länglichen Objekten zum Beispiel müssen jedoch auch andere Verteilungen eintrainiert werden können. Eine Möglichkeit wäre, dass der Benutzer die Ausdehnung des Objekts entlang beliebiger Axen zeigen kann. Dazu müsste die Erkennung der Zeigepositur aus Gleichung 3.36 angepasst und aus den gezeigten Größen die Ausdehnung des Objekts ermittelt werden.
Alternativ könnte der Benutzer mit dem Verfahren zurMessung der Objektposition (siehe Abschnitt 3.2.4.1) die Randpunkte des Objekts festlegen. Aus diesen könnte dann eine Verteilung berechnet werden, welche die Ausdehnung des Objekts möglichst gut abdeckt. Dabei ergibt sich ein Kompromiss zwischen der Genauigkeit der Objektdefinition und dem Aufwand beim Festlegen der Objekte.

4.3.3 Gestenmodul

4.3.3.1 Schnellere Klassifikation

Die Berechnung des Abstands zwischen zwei Gesten mittels Dynamic Time Warping in Python ist vergleichsweise langsam. Bei nur 15 Trainingsbeispielen dauert sie auf

den eingesetzten Desktop-Rechnern bereits etwa 500 Millisekunden. Dies führt zu einer niedrigeren Reaktionsgeschwindigkeit des Gesamtsystems. Zur schnelleren Berechnung wäre es denkbar, den DTW-Algorithmus in C zu implementieren und über eine Python-C-Schnittstelle einzubinden.

4.3.3.2 Variable Gestenlänge

Um Gesten mit sehr unterschiedlichen Längen verwenden zu können wäre es sinnvoll, wenn das Ende einer Geste automatisch erkannt werden könnte. Ein Ansatz ist, einen Klassifikator zu finden, der das Gestenende zuverlässig erkennen kann. Alternativ könnte die bisher ausgeführte Geste auch schon zu einem Bruchteil der maximalen Gestendauer klassifiziert werden. Wird die bis zu diesem Punkt aufgenommene Geste einer bekannten Klasse von Gesten zugeordnet, so kann die Aufnahme vorzeitig abgebrochen und diese Klassifikation verwendet werden. Dieser Ansatz ist aber nur dann möglich, wenn keine der verwendeten Geste ein Präfix einer anderen gültigen Geste ist. Zudem muss der Klassifikator ausreichend schnell sein, siehe dazu Abschnitt 4.3.3.1.

4.3.4 Audiomodul

Die Ausdehnung der Cluster θ_l im Klassifikator des Audiomoduls (siehe Abschnitt 3.4.1.2) ergibt sich über den maximalen Abstand eines Trainingsdatums und des Prototypens innerhalb des Clusters. In der Praxis zeigte sich, dass der Klassifikator dabei wesentlich größere Bereiche des Merkmalsraums akzeptiert als notwendig. Dies führt zu einer höheren Anzahl an falschen Positiven. Wird die Anzahl der Cluster erhöht, ermöglicht dies kleinere Cluster. Dies schafft jedoch nur bedingt Abhilfe, da mehr Trainingsdaten erforderlich sind und Ausreißer weiterhin zu großen Clustern führen.
Alternativ könnte ein anderer Klassifikator verwendet werden. So könnte ein per Expectation Maximization trainiertes Gaußsches Mischmodell durch Berücksichtigung der Kovarianzen der Trainingsdaten vermutlich engere Cluster erzielen [8, 438f.]. Zudem könnten weitere Merkmale gesucht werden, die eventuell bessere Ergebnisse als die Band-Energie-Rate liefern.

4.3.4.1 Verwendung von Stereomikrofonen

Die Kinect verfügt über eingebaute Stereomikrofone. Daher könnte bei einem Schnipsen die Richtung des Benutzers bestimmt werden [24]. Diese Information könnte dann mit der Richtungsinformation aus der Kamera verbunden werden, um falsche Positive zurückzuweisen. In einem möglichen Mehrbenutzersystem (siehe Abschnitt 4.3.1.2) könnte ein Schnipsen somit auch einem der Benutzer zugeordnet werden. Der Kinect-Treiber für OpenNI unterstützte jedoch zum Zeitpunkt dieser Arbeit den Zugriff auf die

Mikrofone noch nicht, obwohl OpenNI selbst entsprechende Audiofunktionen bietet. Daher wurde dieser Ansatz im Rahmen dieser Arbeit nicht weiter verfolgt.

4.3.5 Ausgabemodul

Das Ausgabemodul könnte erweitert werden, so dass auch andere Ereignisse Zustandsübergänge in Automaten auslösen können. Wenn zum Beispiel ein Zustandsübergang durch Beginn oder Ende einer Zeigehaltung ausgelöst werden kann, könnte die Lautstärke der angeschlossenen Audiogeräte während des Zeigens reduziert werden. Dies würde die Schnipserkennung erleichtern. Zudem könnten Zustandsübergänge nach einer vorgegebenen Zeit automatisch ausgelöst werden, wodurch sich komplexe Aktionen über die Verkettung mehrerer Zustände umsetzen lassen.

4.3.6 Weitere Objekte

Um mehr Aspekte der alltäglichen Umgebung steuern zu können, könnten weitere Smart Objects implementiert werden. Zusätzlich zur bereits vorhandenen Steuerung der Musikwiedergabe wäre eine Anbindung eines Fernsehers[2] möglich. Dabei könnten über Gesten die Lautstärke und die Wahl des Programms gesteuert werden. In Verbindung mit einem Fernseher oder einem Bildschirm könnte das System zudem zusätzliche Informationen über den Systemzustand ausgeben (siehe Abschnitt 4.3.7.1) oder zur Navigation durch eine grafische Oberfläche genutzt werden.
Als Erweiterung der Steuerung des Tischventilators und der Beleuchtung könnten auch Heizungen oder Klimaanlagen gesteuert werden. Bei der Beleuchtung wäre eine Erweiterung für dimmbare Lampen denkbar.

4.3.7 Integration

4.3.7.1 Grafische Oberfläche

Das Gesamtsystem und seine Trainings- und Testprogramme könnten mit einer grafischen Oberfläche ausgestattet werden. Auf dieser könnte der momentane Zustand des Systems und das Skelettmodell oder Tiefenbild des aktiven Benutzers dargestellt werden. Im Regelbetrieb ist dies nicht notwendig, da das System keinen Bildschirm verwendet. Eine grafische Oberfläche könnte aber Tests und Training erleichtern.

[2] Statt eines Fernsehers könnte auch ein PC mit entsprechender Wiedergabesoftware verwendet werden. Dieser ist vermutlich auch leichter anzusteuern

4.3.7.2 Weitere akustische Signale

Neben den bereits verwendeten akustischen Signalen bei Anfang und Ende einer Geste (siehe Abschnitt 3.6) könnten weitere Signale eingesetzt werden. So könnte der Benutzer auf diesem Wege zur Kalibration aufgefordert und über eine erfolgreiche oder gescheiterte Kalibration informiert werden.

4.3.8 Benutzerstudie

Wie in Abschnitt 4.1.3 bereits erwähnt, sind die durchgeführten Tests nur eingeschränkt dazu geeignet, die Leistung des Systems zu evaluieren. Ein besserer Test wäre eine längere Benutzerstudie inklusive einer Vorstudie.
In einer Vorstudie würde das System auf den eigentlichen Test vorbereitet. Dabei müssten unter anderem aus einer ausreichend großen Gruppe von Benutzern Trainingsdaten für die Klassifikatoren gewonnen werden. Für Systemparameter, die sich nicht direkt trainieren lassen, müssten im Rahmen dieser Vorstudie experiementell möglichst gute Werte ermittelt werden. Unter Berücksichtigung von Rückmeldungen der Versuchspersonen sollte ein Kanon aus möglichst gut benutzbaren Gesten aufgebaut werden. Auch kann untersucht werden, welche Objekte in welchen Positionen und in welcher Umgebung verwendet werden sollten, um einen alltäglichen Einsatz des Systems möglichst gut wiederzugeben.
Zudem muss eine Menge an Negativbeispielen gesammelt werden, um das System auf falsche Positive testen zu können. Dazu müssen alltägliche Geräusche und Gesten gesammelt und ihre Häufigkeit im Alltag bestimmt werden[3].
In der Hauptstudie würde das so konfigurierte System von einer möglichst großen Anzahl an verschiedenen Benutzern getestet. Dabei sollten ein zuvor entworfenes Versuchsprotokoll befolgt und die Versuche auf Video und über die Sensordaten aufgezeichnet werden. Zudem werden dem System die Negativbeispiele präsentiert, um die Klassifikationsleistung des Systems zu bestimmen. Über eine Benutzerbefragung kann zudem der subjektive Eindruck auf die Versuchspersonen ermittelt werden[4]. Aus diesen Daten lassen sich dann bessere Rückschlüsse auf die Stärken und Schwächen des Systems ziehen.

[3] Liegt eine Menge solcher Negativbeispiele vor, so könnte die Schnipserkennung eventuell durch einen Zwei-Klassen-Klassifikator ersetzt werden.

[4] Dabei könnten zum Beispiel die *IBM Computer Usability Satisfaction Questionnaires* [16] als Grundlage für die Befragung verwendet werden.

A

Installation und Benutzung des Systems

Dieses Kapitel versucht, eine praktische Anleitung zur Installation und Benutzung des Systems zu geben. Zudem finden sich in Abschnitt A.4 einige Ansätze zur Fehlerbehebung.

A.1 Installation des Systems

Zur Installation müssen zunächst die Abhängigkeiten installiert werden. Dazu gehören der Python-Interpreter, das `pyaudio`-Modul[1] und die `scipy` und `numpy`-Module[2]. Zur Kommunikation mit den Objekten müssen die Python-Module des MIRIAM-Projekts vorhanden sein. Soll das Smart Object zur Musiksteuerung aus Abschnitt 3.7.4.2 verwendet werden, so wird das Python-Modul `mpd` aus der Bibliothek `python-mpd` benötigt[3].

[1] Die PyAudio-Webseite ist unter der Adresse `http://people.csail.mit.edu/hubert/pyaudio/` zu finden. PyAudio benötigt als Abhängigkeit die PortAudio v19 Bibliothek.

[2] Der Python-Interpreter kann über die offizielle Python-Website `http://www.python.org/` bezogen werden. SciPy und NumPy finden sich unter `http://www.scipy.org/`.

[3] Da das Modul unter der LGPL lizensiert ist, liegt eine Kopie als `mpd.py` im Unterordner `src/objects` bei. Eine Installation des Moduls ist daher nicht erforderlich. Damit das Modul vom Python-Interpreter gefunden wird, müssen die Smart Object-Programme aus dem Unterordner `src/objects` gestartet werden. Alternativ kann die `PYTHONPATH`-Variable gesetzt werden. Das MPD-Modul findet sich auch unter der Adresse `http://jatreuman.indefero.net/p/python-mpd/`.

Des weiteren muss die OpenNI-Bibliothek mit der NITE-Middleware und dem Kinect-Treiber installiert werden[4]. Die Installation dieser Komponenten und ihrer Abhängigkeiten ist je nach System unterschiedlich. Details können der Dokumentation der jeweiligen Software entnommen werden.
Nach der Installation der Abhängigkeiten müssen Umgebungsvariablen gesetzt werden. Die Variable `PYTHONPATH` muss den Pfad zu den oben genannten Python-Module beinhalten, sofern diese nicht in einem der Python-Standardverzeichnisse zu finden sind. Insbesondere muss der `lib`-Unterordner des MIRIAM-Projekts in der `PYTHONPATH`-Variable enthalten sein[5].
Um das Eingabemodul aus Abschnitt 3.1.2 zu kompilieren kann die mitgelieferte Makefile verwendet werden. Diese verwendet `pkg-config`, um den Pfad zu den OpenNI und NITE-Dateien zu ermitteln. Daher müssen `pkg-config` die Pakete `nite` und `openni` bekannt sein[6]. Steht `pkg-config` nicht zur Verfügung, so kann alternativ die Makefile `Makefile.nopkgconfig` verwendet werden. Dabei muss in der Makefile die Variable `OPENNI_PREFIX` auf den Installationsort der NITE- und OpenNI-Bibliotheken, also zum Beispiel `/usr/local`, gesetzt werden.

A.2 Benutzung des Systems

Vor Benutzung des Systems sollten die verwendeten Smart Objects gestartet werden (siehe dazu Abschnitt A.2.1). Zudem müssen die Kinect-Kamera, die Lautsprecher und das Mikrofon angeschlossen werden. Verfügt der Rechner über mehr als einen Audioausgang oder Audioeingang, so muss der für Lautsprecher und Mikrofon verwendete Ausgang beziehungsweise Eingang im Betriebssystem als Standardausgang beziehungsweise Standardeingang gesetzt werden. Wurde der Aufbau des Systems verändert, muss das System eventuell entsprechend Abschnitt A.3 neu trainiert werden.
Das System lässt sich über einen Aufruf des Programms `openniReceiver.py` im Unterverzeichnis `src` starten. Dabei wird das System zunächst initialisiert. Ist die Initialisierung abgeschlossen, so erscheint die Nachricht „SYSTEM READY“.
Wird der Benutzer erkannt, wird die Nachricht „Detected new user #X“ ausgegeben, wobei X die OpenNI-Kennnummer des Benutzers ist. Im Anschluss muss sich der Benutzer der Kamera zuwenden und die Ψ-Positur aus Abbildung 3.1 einnehmen. Wird die Haltung erkannt, wird eine „Detected pose“ Nachricht ausgegeben und mit der Ka-

[4]Diese sind unter der Adresse `http://www.openni.org/` erhältlich. Zur Ansteuerung der Kinect wird von OpenNI der *PrimeSensor*-Treiber verwendet.

[5]Dieser findet sich im MIRIAM-Paket unter `smartobject/SmartObjects/lib`.

[6]Eventuell muss hierzu die Umgebungsvariable `PKG_CONFIG_PATH` auf den Pfad gesetzt werden, der die entsprechenden `.pc`-Dateien enthält. Ob die Module vorhanden sind, kann anhand der Liste aller bekannten Pakete überprüft werden. Diese lässt sich über den Befehl `pkg-config --list-all` ausgeben.

libration begonnen. Während der Kalibration muss der Benutzer diese Körperhaltung beibehalten. Nach etwa drei Sekunden erscheint bei erfolgreicher Kalibration die Meldung „Calibration for user X: SUCCESS“. Schlägt die Kalibration fehl, so erscheint die Meldung „Calibration for user X: FAILURE“. In diesem Fall muss die Ψ-Haltung erneut eingenommen und eine neue Kalibration versucht werden. Bei der Kalibration sind die Anmerkungen aus Abschnitt A.4.1 zu beachten.
Nach einer erfolgreichen Kalibration kann das System wie in der Arbeit beschrieben verwendet werden. Über die Tastenkombination `Ctrl-C` lässt sich das System beenden.

A.2.1 Verwendung der mitgelieferten Smart Objects

Die mitgelieferten Smart Objects aus Abschnitt 3.7.4 befinden sich im Unterverzeichnis `src/objects` und können von dort aus gestartet werden.
Um den Benutzernamen und das Passwort für die Anmeldung am XMPP-Server zu setzen, müssen die globalen Variablen `USER` und `PASSWORD` in der Skriptdatei des jeweiligen Objekts geändert werden. Die Adresse und der Name des XMPP-Servers werden, wie bei MIRIAM üblich, über die zu MIRIAM gehörenden Datei `config.py` gesetzt.
Die Smart Objects werden durch Drücken der Enter-Taste beendet.

A.3 Training des Systems

Um die Gesten, die Objekte und die Schnipserkennung zu trainieren, stehen die in Abschnitt 3.3.4, 3.2.4 und 3.4.4 beschriebenen Trainingsskripte zur Verfügung. Zum Trainieren müssen das jeweilige Trainingsskript gestartet und die Anweisungen auf dem Bildschirm und aus den jeweiligen Abschnitten der Arbeit befolgt werden. Das Gesten- und Zeigetraining erfordert zudem zu Beginn eine Kalibrierung des Benutzers analog zu Abschnitt A.2. Die Trainingsskripte können jeweils über die Tastenkombination `Ctrl-C` beendet werden[7]. Nach Ende des Programms wird eine Konfigurationsdatei geschrieben. Um die Trainingsergebnisse im System zu verwenden muss die jeweilige Standard-Konfigurationsdatei[8] durch die neu generierte Konfigurationsdatei ersetzt werden.

[7] Vor dem Beenden müssen noch ausstehende Programmeingaben, wie die Frage nach Objektnamen, beantwortet werden. Das Programm endet erst, wenn die ausstehende Eingabe getätigt wurde.

[8] Die Standard-Konfigurationsdateien sind dabei `gestures.xml`, `pointing.xml` und `snapping.xml`.

A.4 Bekannte Einschränkungen

Dieser Abschnitt enthält Einschränkungen, die im praktischen Einsatz als Fehlerursachen auftreten können. Funktioniert das System nicht wie erwartet, so sind die nachfolgenden Punkte Ansätze für eine Fehlersuche.

A.4.1 Kamera

Die Kinect-Kamera muss parallel zum Boden ausgerichtet sein, damit die Höhenwinkel korrekt berechnet werden (siehe Abschnitt 3.2.1). Zudem darf die Kamera nach der Festlegung der Objektpositionen nicht bewegt werden, da die Objektpositionen relativ zur Kameraposition gespeichert sind.
Selbstüberdeckung, bei dem ein Teil des Benutzerkörpers ein anderes Körperteil aus Sicht der Kamera verdeckt, kann zu einem teilweisen Verlust der Skelettverfolgung führen. Dies tritt insbesondere dann auf, wenn der Benutzer den der Kamera zugewandten Arm hebt und damit den eigenen Kopf verdeckt. Wird auf Objekte gezeigt, die sehr nah an der Kamera liegen, kann der nun direkt auf die Kamera ausgerichtete Arm eventuell nicht mehr erfasst werden.
Verlässt der Benutzer das Sichtfeld der Kamera und betritt es wieder, so muss die Kalibration erneut durchgeführt werden. Zudem kann es einige Sekunden dauern, bis die Benutzerverfolgung die Verfolgung des vorherigen Benutzers abbricht und auf den neuen Benutzer umschaltet. Dieser Fall kann auch eintreten, wenn der Benutzer von einer anderen Person abgelöst wird oder die Skelettanalyse den Benutzer verliert.
Bei der Kalibrierung sollte sich der Benutzer in der Ψ-Positur befinden und der Kamera zugewandt sein. Zudem sollte möglichst viel Abstand zu Hintergrundobjekten gehalten werden, da die Kalibration sonst fehlschlagen kann.
Um anfänglich von der Benutzerverfolgung erfasst zu werden, muss sich der Benutzer zunächst bewegen. Dies gilt insbesondere dann, wenn der Benutzer sich bereits bei Programmstart im Blickfeld der Kamera befand.

A.4.2 Schnipserkennung

Um bei der Schnipserkennung möglichst gute Ergebnisse zu erzielen, sollte die Lautstärke des Mikrofons passend eingestellt sein. Ein zu leises oder zu lautes Signal führt zu schlechteren Ergebnissen. Dies gilt besonders dann, wenn das Signal die maximale Amplitude überschreitet und bei der Aufnahme abgeschnitten wird.

A.4.3 Ausgabe

Bei zu schnellem Umschalten können die verwendeten netzwerkfähigen Steckdosen eventuell ein undefiniertes Verhalten zeigen. Dabei werden zum Beispiel bei einem ein-

zigen Schaltvorgang mehrere Dosen ein- oder ausgeschaltet. Werden die betroffenen Dosen langsam wieder ein- und ausgeschaltet, stellt sich in der Regel das normale Verhalten wieder ein.

Literaturverzeichnis

[1] Scipy v0.11.dev reference guide: scipy.cluster.vq.kmeans2. `http://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.kmeans2.html`, 2009.

[2] SciPy v0.11.dev Reference Guide: scipy.cluster.vq.whiten. `http://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.whiten.html`, 2009.

[3] Kinect™ for Windows®. `http://kinectforwindows.org/`, 2011.

[4] Music Player Daemon Community Wiki. `http://mpd.wikia.com/`, 2011.

[5] Numpy and Scipy documentation. `http://docs.scipy.org/doc/`, 2011.

[6] The OpenKinect Wiki. `http://openkinect.org/wiki/Main_Page`, 2011.

[7] Python v2.6.7 documentation. http://docs.python.org/release/2.6.7/, 2011.

[8] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC, New York, 2006.

[9] Bose, P., Maheshwari, A., and Morin, P. Fast approximations for sums of distances, clustering and the Fermat-Weber problem. *Computational Geometry*, 24(3):135–146, April 2003.

[10] Cheung, G., Kanade, T., Bouguet, J.-Y., and Holler, M. A real time system for robust 3D voxel reconstruction of human motions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2000.

[11] Do, J.-H., Jung, S. H., Jang, H., Yang, S.-E., Jung, J.-W., and Bien, Z. Gesture-Based Interface for Home Appliance Control in Smart Home. In Nugent, C. and Augusto, J. C., editors, *Smart Homes and Beyond - ICOST2006 4th International Conference On Smart homes and health Telematics*, pages 23–30. IOS Press, Amsterdam, 2006.

[12] *GNU Make Manual*. Free Software Foundation, Inc., 2010.

[13] Freeman, W. T. and Weissman, C. D. Television Control by Hand Gestures. Technical report, Mitsubishi Electric Research Laboratories, 1994.

[14] Irie, K., Wakamura, N., and Umeda, K. Construction of an Intelligent Room Based on Gesture Recognition. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[15] Kriesten, B. Smartphonebasierte mixed-reality-interaktionen für intelligente umgebungen. Diplomarbeit, Technische Fakultät der Universität Bielefeld, 2009.

[16] Lewis, J. R. IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instructions for Use. Technical report, Human Factors Group, IBM Corporation, 1993.

[17] Lin, S.-Y., Lai, Y.-C., Chan, L.-W., and Hung, Y.-P. Real-Time 3D Model-Based Gesture Tracking for Multimedia Control. In *20th International Conference on Pattern Recognition*, 2010.

[18] *OpenNI™ User Guide*. The OpenNI organization, 2011.

[19] Ouchi, K., Esaka, N., Tamura, Y., Hirahara, M., and Doi, M. Magic Wand: an intuitive gesture remote control for home appliances. In *Proceedings of the 2005 International Conference Active Media Technology*, page 274, 2005.

[20] Park, J. W., Nam, Y., Kim, D. H., and duke Cho, W. SWINGREMOCON: A Gesture-Based Remote Controller for Home Appliances. In *IADIS International Conference Interfaces and Human Computer Interaction 2008*, pages 268–271, 2008.

[21] Pham, H. Pyaudio: Portaudio v19 python bindings. `http://people.csail.mit.edu/hubert/pyaudio/`, 2010.

[22] *Prime Sensor™ NITE 1.3 Algorithms notes, Version 1.0*. PrimeSense Inc., 2010.

[23] Sakoe, H. and Chiba, S. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26:43–40, 1978.

[24] Vesa, S. and Lokki, T. An eyes-free user interface controlled by finger snaps. *Proc. of the 8th Int. Conference on Digital Audio Effects (DAFx.05)*, pages 262–265, 2005.

[25] Wilson, A. and Shafer, S. XWand: UI for intelligent spaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2003.

Hiermit versichere ich, dass ich diese Masterarbeit selbständig bearbeitet habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und entsprechende Zitate kenntlich gemacht.

Bielefeld, den 21. November 2011

David Fleer