Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Michael Zajonz

Entwicklung eines Tools zur Aufzeichnung und Wiedererkennung von Körper-Posen und -Gesten

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Michael Zajonz

Entwicklung eines Tools zur Aufzeichnung und Wiedererkennung von Körper-Posen und -Gesten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Birgit Wendholt
Zweitgutachter: Prof. Dr. Andreas Meisel

Eingereicht am: 20. November 2014

Michael Zajonz

Thema der Arbeit

Entwicklung eines Tools zur Aufzeichnung und Wiedererkennung von Körper-Posen und -Gesten

Stichworte

NUI, Körperposen, Körpergesten, Posen, Gesten, Aufzeichnung, Erkennung, Kinect, Skelett

Kurzzusammenfassung

Diese Arbeit bietet eine Softwarelösung für das Problem der Aufzeichnung und Wiedererkennung von Körperposen und -gesten. Auf Basis einer ausführlichen Betrachtung der theoretischen Grundlagen wird der gewählte Lösungsansatz gegen vergleichbare Arbeiten abgegrenzt und die Auswahl begründet. Das Ergebnis der Arbeit – ein Softwarepaket aus Programmbibliothek zur Erkennung eines definierten Posen-/Gestenrepertoires, sowie ein Werkzeug zur Anfertigung eines solchen Repertoires – wird detailliert beschrieben. Die Grundlage des gewählten Ansatzes liegt in der Erkennung/Aufzeichnung von Schlüsselposen auf Basis eines von einem optischen Sensor bereitgestellten Skelett-Modells. Anhand eines Beispiel-Repertoires wird die Leistungsfähigkeit der Lösung mit Hinblick auf Trainingsumfang und Erkennungsqualität abgeschätzt. Der Anhang bietet unterstützende Dokumentation, die die primären Anwendergruppen (Informatiker und Interaktionsdesigner) an die Verwendung des im Rahmen dieser Arbeit entstandenen Softwarepakets heranführt.

Michael Zajonz

Title of the paper

Development of a toolkit for the recording and recognition of human poses and gestures

Keywords

NUI, human body poses, human body gestures, poses, gestures, recording, recognition, Kinect, skeleton

Abstract

This paper presents a software solution to the problem of recording ang recognizing human body poses and gestures. Based on a thorough review of the problem's theoretical foundations a number of possible solutions are evaluated and the choices made justified. The result of this work is a software package made up of a program library for recognition of a given pose/gesture set, as well as a tool to create such a pose/gesture set and is described in detail. The chosen approach takes a skeleton model from an optical sensor and uses that for detection and recording of key poses. Using an example pose/gesture set the performance of the solution is examined in regard to required trainig effort and recognition quality. The appendix to this paper offers supporting documentation to aid the primary target audience (Computer Scientists and Interaction Designers) in employing the created software package.

Dank

Für die Unterstützung bei dieser Arbeit möchte ich mich hier vor allem bei Prof. Dr. Birgit Wendholt bedanken. Trotz vollem Terminplan fand sich immer Zeit für offene Fragen und wertvolles Feedback. Konstruktive Kritik kann auch einfach mal Lachen sein.

Und nicht zu vergessen Freunde und Familie, die einem wochenlange Vernachlässigung verzeihen und wo es geht Kraft spenden und Unterstützung bieten. *Last but not least* auch einen besonderen Dank an Jens Vielhaben, der es mir auf weite Strecken überhaupt erst möglich gemacht hat, die Augen auf mein Ziel gerichtet zu halten.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Zielsetzung . . . 2
1.2. Gliederung . . . 2

2. Verwandte Arbeiten **5**
2.1. Modellbildung . . . 6
2.1.1. Für die Arbeit nicht relevante Verfahren . . . 6
2.1.2. Optische markierungslose merkmalbasierte Verfahren . . . 9
2.1.3. Zusammenfassung . . . 12
2.2. Verfahren zur Gestenaufzeichnung und -wiedererkennung . . . 12
2.2.1. Dynamic Time Warping (DTW) . . . 13
2.2.2. Hidden Markov Models (HMM) . . . 15
2.2.3. Time-Delay Neural Networks (TDNN) . . . 20
2.2.4. Rubines Algorithmus . . . 26
2.2.5. Aufzeichnung/Erkennung auf Grundlage von Schlüssel-Posen . . . 29
2.2.6. Zusammenfassende Beurteilung . . . 34
2.3. Vorhandene Werkzeuge . . . 35
2.3.1. Gesture Description Language (GDL), GDL Studio . . . 35
2.3.2. GesturePak . . . 36
2.3.3. iGesture . . . 36
2.3.4. Zusammenfassende Betrachtung vorhandener Werkzeuge . . . 37
2.4. Zusammenfassung verwandter Arbeiten . . . 37

3. Anforderungsanalyse **39**
3.1. Anforderungen aus Anwender-Sicht . . . 39
3.1.1. Erkennung von Posen und Gesten . . . 40
3.1.2. Eintrainieren und Testen von Posen und Gesten . . . 40
3.1.3. Zusammenfassung der Anforderungen aus Anwender-Sicht . . . 41
3.2. Anforderungen aus Entwickler-Sicht . . . 41
3.3. Zusammenfassung der Anforderungen . . . 42

4. Systementwurf **45**
4.1. Workflow . . . 45
4.2. Erkennungs-Komponente . . . 46
4.3. Aufnahme-Komponente . . . 47

4.4. Struktureller Entwurf . . . 48
4.4.1. Erkennungs-Bibliothek . . . 48
4.4.2. Aufnahme-Werkzeug und andere Anwendungen . . . 49
4.4.3. Sensor-Schnittstelle . . . 49
4.5. Zusammenfassung . . . 50

5. Realisierung **51**
5.1. Wissensbasis: Datenstrukturen und Datenbankformat . . . 52
5.2. Designentscheidungen . . . 54
5.2.1. Inversion of Control (IoC) und Ereignisse . . . 54
5.2.2. Fassade für *libgrr*-Module: Vereinfachte Schnittstelle . . . 54
5.2.3. Systemumgebung . . . 55
5.2.4. Zusammenfassung . . . 56
5.3. Kern-Komponente: *libgrr*-Programmbibliothek . . . 56
5.3.1. Rohdatenaufbereitung . . . 57
5.3.2. Schlüsselposen-Erkennung . . . 59
5.3.3. Gesten-Erkennung . . . 60
5.3.4. Hilfsmittel . . . 61
5.3.5. Zusammenfassung . . . 63
5.4. Aufnahme-Werkzeug: *grrtool* . . . 63
5.4.1. Realisierung das Funktionsumfanges . . . 63
5.4.2. Verwendung von *libgrr* . . . 65
5.5. Zusammenfassung . . . 67

6. Evaluation **69**
6.1. Erkennungsleistung und Trainingsaufwand . . . 69
6.2. Lokale Ressourcen . . . 72
6.3. Netzwerk . . . 73
6.4. Zusammenfassung . . . 73

7. Schlussbetrachtungen **75**
7.1. Fazit . . . 75
7.2. Ausblick . . . 77
7.2.1. Funktionale Aspekte . . . 77
7.2.2. Technische Aspekte . . . 79

A. Events **81**
A.1. BufferUpdated . . . 81
A.2. GestureDetected . . . 81
A.3. GestureDetectorTrainingComplete . . . 82
A.4. InputFrameProcessed . . . 82
A.5. KeyPoseDetected . . . 83
A.6. KeyPoseDetectorTrainingComplete . . . 84

A.7. SkeletonFrameReceived 84

B. Quick Start Guide **85**
B.1. System Requirements 85
B.1.1. libGRR 85
B.1.2. GRRTool 85
B.1.3. SensorDataProvider 86
B.2. Getting started 86
B.2.1. SensorDataProvider 86
B.2.2. GRRTool 87
B.2.3. libGRR 89

C. Message Format **91**
C.1. Incoming Messages 91
C.1.1. Input Frame 91
C.2. Outgoing Messages 93
C.2.1. Detected Key Poses 93
C.2.2. Detected Gestures 94
C.2.3. Key Pose Probability Distributions 95

D. Genehmigungen und Lizenzen **97**
D.1. Genehmigung zur Verwendung von Grafiken 97
D.1.1. DDTW-Grafiken 97
D.2. Lizenzbedingungen der entwickelten Software 98
D.2.1. libgrr 98
D.2.2. grrtool 101

1. Einleitung

Computersysteme dringen immer weiter in das Alltagsleben der Menschen vor und nehmen dort unterschiedlichste Rollen ein. Ebenso vielfältig sind die Mechanismen, mit denen die Benutzer mit den Computersystemen in Interaktion treten können. Dabei muss in vielen Fällen die Bedienung dieser Systeme jedoch zunächst erlernt werden. Gerätehersteller stehen vor der Herausforderung, die Bedienung so einfach und intuitiv wie möglich zu gestalten. Der wachsende Funktionsumfang vieler Geräte verschärft dieses Problem.

Der heute nahezu allgegenwärtige *Touchscreen* illustriert sehr gut die Richtung, in die sich die Entwicklung von Benutzerschnittstellen bewegt. Der Tastenumfang wird reduziert, Gerätefunktionen werden direkter angesprochen. Der Benutzer zeigt und tippt auf die Funktion von Interesse. Er muss nicht länger mittels verschiedener Tasten eine Art Cursor bewegen, um eine auszuführende Funktion zuvor auszuwählen. Eingabe- und Ausgabe-Geräte verschmelzen, die Interaktion mit dem Gerät wird mehr und mehr eine direkte Interaktion mit dem Inhalt. Solche „direkten", gestenbasierten Benutzerschnittstellen fallen unter die Bezeichnung „Natural User Interface" (NUI).

Der nächste Schritt stellt die kontaktlose Bedienung dar. Solche Bedienkonzepte erfordern ein Umdenken in der Gestaltung der Interaktion mit der (zu bedienenden) Umwelt. Alte Lösungen werden unpraktisch, neue Lösungen etablieren sich und ganz neue Anwendungen werden ermöglicht. Der Schlüssel zu einer effizienten, leicht erlernbaren und bequemen Bedienung bleibt allerdings ihre *Intuitivität*. Um diese Intuitivität realisieren zu können, kann nicht auf ein „halbwegs passendes" Gestenrepertoire zurückgegriffen werden, es muss eines auf die jeweilige Anwendung zugeschnittenes sein.

Zur effektiven Entwicklung solcher Bedienkonzepte auf Basis von Freiraum-Gesten sind Hilfsmittel nötig, die es erlauben, solche Freiraum-Gesten schnell und unkompliziert definieren und wiedererkennen zu können. Mit der Vereinfachung des Umgangs mit dieser Technologie wächst auch ihre Akzeptanz bei Entwicklern und Anwendern. Dabei geht es aber nicht nur um die Steigerung der Bequemlichkeit in verschiedenen Anwendungen. Kontaktlose Gesten-Bedienung macht verschiedene Anwendungen in Medizin, Chemie oder Biologie überhaupt erst möglich.

Im Folgenden wird die Zielsetzung dieser Arbeit konkretisiert und ihr allgemeiner Aufbau kurz erläutert.

1.1. Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer Programmbibliothek zur Erkennung von Körperposen und -gesten, sowie eines zugehörigen Werkzeugs zum Aufbau des von der Programmbibliothek wiedererkennbaren Posen- und Gestenrepertoires. Die Nutzer der Bibliothek können mit diesem Werkzeug also eigene Gesten und Posen aufzeichnen, so dass sie diese dann verwenden können. Diese Aufzeichnung soll lediglich aus mehrmaligem Durchführen einer Geste oder Einnehmen einer Pose bestehen.

Da es um Freiraum-Gesten des menschlichen Körpers geht soll die Grundlage der Aufzeichnung und Erkennung ein bereitgestelltes Modell des Benutzer-Skeletts sein. Die Erzeugung eines solchen Modells soll keine Präparation des Benutzers erfordern. Sie ist an sich zwar nicht Gegenstand dieser Arbeit, muss aber auf Grund der gestellten Anforderungen dennoch beachtet werden.

Bei der Programmbibliothek zur Posen- und Gestenerkennung und des Aufnahme-Werkzeugs soll das Augenmerk auf zwei wesentliche Aspekte gelegt werden: Sie sollen einfach zu bedienen sein, und sie sollen sich in auf vielfältige Weise in unterschiedliche Umgebungen einbinden lassen. Es sollen keinerlei Kenntnisse von Mechanismen zur Körperposen- oder Gestenerkennung vorausgesetzt werden. Dies ist wichtig, um Hürden beim Einsatz der Bibliothek und des Aufnahme-Werkzeugs abzubauen.

Der Einsatz der Programmbibliothek soll unter einer möglichst großen Anzahl von Hard- und Softwareumgebungen möglich sein. Das selbe ist wünschenswert für das Aufnahme-Werkzeug, dort jedoch etwas weniger wichtig, da ein Aufgenommenes Posen- und Gestenrepertoire ein austauschbares Ganzes bilden soll.

1.2. Gliederung

Der Aufbau dieser Arbeit und Gegenstand der einzelnen Kapitel sind im Folgenden näher beschrieben.

Kapitel 2 bietet eine weiterführende Einführung in das Themengebiet und betrachtet die einzelnen Bestandteile der im Rahmen dieser Arbeit zu lösenden Aufgabe. Hierzu werden auch allgemeine Möglichkeiten zur Erstellung des benötigten Skelett-Modells diskutiert und die letzten Endes einzige in Frage kommende Technik vorgestellt. Im Anschluss folgt eine detaillierte

Betrachtung und Bewertung einzelner für Zwecke der Posen- und/oder Gestenerkennung prinzipiell verwendbarer Verfahren. Ziel ist die Auswahl eines geeigneten Verfahrens, das die Grundlage für den weiteren Verlauf der Arbeit bildet. Abgerundet wird dieser Überblick durch eine kurze Vorstellung von bereits existierenden Lösungen, die sich in diesem Themengebiet bewegen.

Anschließend werden in Kapitel 3 die Anforderungen an das im Rahmen dieser Arbeit entwickelte System konkretisiert. Kapitel 4 beschreibt den funktionalen und strukturellen Entwurf dieses Systems im Detail. Die Umsetzung dieses Entwurfs und die dabei getroffenen Designentscheidungen werden ausführlich in Kapitel 5 erläutert.

Abschließend bietet Kapitel 6 eine Evaluation des entwickelten Systems, um das Erreichte mit einigen Zahlen zu untermauern, bevor 7 die Ergebnisse dieser Arbeit nochmals zusammenfasst und einen Ausblick auf mögliche anschließende Arbeiten bietet.

Die Anhänge dieser Arbeit liefern eine Reihe technischer Details, die in dieser Genauigkeit für das Verständnis des Textes nicht nötig sind. Auf diese Anhänge wird an den entsprechenden Stellen verwiesen. Sie sind, wie auch die entstehende Software, in englischer Sprache, um ein größtmögliches Publikum zu erreichen.

2. Verwandte Arbeiten

Systeme zur Posen- oder Gestenerkennung arbeiten prinzipiell zwei-stufig: Als erstes müssen Körperteile o.ä. überhaupt als solche identifiziert und lokalisiert werden. Aufgrund der Vielzahl möglicher Freiheitsgrade (Orientierung, Position, Bewegung, Kleidung/Hautfarbe, ...) ist dies keine triviale Aufgabe. Als zweites müssen die auf erster Ebene gewonnenen Informationen über eine vermeintliche Pose bzw. Geste beurteilt und z.B. mit einem bekannten Repertoire abgeglichen werden. (Kota u. a., 2009) Diese beiden Aufgaben/Stufen sollen im Rahmen dieser Arbeit als „Modellbildung" und „(Gesten-)Erkennung" bezeichnet werden.

Um die nachfolgenden Beschreibungen etwas zu entwirren wird hier zunächst die Betrachtung von Körperposen neben den Körpergesten nicht immer wieder aufs neue erwähnt. Dem liegt die Annahme zu Grunde, dass von Betrachtung der Eingabedaten bis hin zu einer Gestenrepräsentation das Abstraktionsniveau zunimmt und die einzelnen Abstraktionsschritte aufeinander aufbauen. In solch einer Hierarchie wäre die Posenerkennung unterhalb der Gestenerkennug anzusiedeln. An dem Punkt, an dem die Arbeit mit Gesten betrachtet wird, wird also davon ausgegangen, dass Posen-Informationen verfügbar sind und das untergeordnete Problem darstellen. Die Betrachtung von Posen braucht so nur an den Punkten erfolgen, an denen diese tiefergehenden Informationen auch hilfreich sind.

Ein Gesten-Repertoire, auf welches zurückgegriffen werden soll, muss den verschiedenen Verfahren in jeweils geeigneter Form antrainiert werden. Wie dieses Training bzw. die Aufzeichnung von wiederzuerkennenden Gesten aussieht unterscheidet sich von Erkennungsverfahren zu Erkennungsverfahren. Die für ein bestimmtes Erkennungsverfahren vorgesehenen Trainingsverfahren werden in Abschnitt 2.2 zusammen mit diesem erklärt.

Zunächst jedoch soll in Abschnitt 2.1 ein Überblick über verschiedene Verfahren zur Modellbildung gegeben werden. Daran schließt sich dann in Abschnitt 2.2 die Vorstellung jeweils zweckmäßiger Erkennungsverfahren an. Dabei liegt der Fokus auf Ansätzen, die dem Zeil dieser Arbeit prinzipiell dienlich sein könnten. Verfahren die prinzipbedingt ungeeignet sind werden nur der Vollständigkeit halber kurz angerissen.

2.1. Modellbildung

Der Schritt der Modellbildung dient dazu, die in irgendeiner Weise gewonnen Rohdaten in eine Form zu überführen, auf der das System in einem weiteren Schritt dann effektiv die eigentliche Erkennungsaufgabe durchführen kann. Im Grunde handelt es sich dabei um Techniken zum sog. Motion Capturing, also der Aufzeichnung von Bewegungsabläufen (Moeslund und Granum, 2001).

Das im Rahmen dieser Arbeit entwickelte Tool soll ohne weitere Vorbereitungen mit beliebigen Personen arbeiten können. Für die erste Stufe unseres Systems kommen daher nur optische markierungslose Verfahren in Frage [1], da andere Verfahren meist eine Präparation des Beutzers mit Sensoren erfordern (vgl. Abschnitt 2.1.1). Besonders geeignet als Grundlage für die Aufzeichnungs- und Erkennungsaufgabe ist ein in Echtzeit erstelltes Skelett-Modell, dessen Hauptmerkmale ausgewählte Gelenke und Gliedmaßen des menschlichen Skeletts repräsentieren (Johansson, 1973; Wang u. a., 2003). Die Menge der in Frage kommenden Verfahren schränkt sich daher weiter ein auf optische markierungslose merkmalsbasierte Verfahren.

Abbildung 2.1 gibt eine einfache Übersicht über die verschiedenen Arten von Verfahren zur Modellbildung. Aus den einzelnen Kategorien werden, zusammengefasst nach Relevanz für die vorliegende Aufgabenstellung, einige Verfahren vorgestellt und bewertet. Grundsätzlich ist es so, dass der Weg, auf dem das Skelettmodell erzeugt wird, nicht Gegenstand dieser Arbeit ist. Dieser Abschnitt soll mehr einen Überblick über vorhandene Ansätze geben um die Abgrenzung dieser Arbeit abzurunden als alle vorhandenen Ansätze im Detail vorzustellen.

2.1.1. Für die Arbeit nicht relevante Verfahren

Die im Folgenden beschriebenen Verfahren eignen sich aus verschieden Gründen nicht für diese Arbeit. Dennoch sollen Sie hier der Vollständigkeit halber Erwähnung finden.

Nicht-optische Verfahren

Zu den nicht-optischen Verfahren zählen z.B. Inertialsysteme, Exoskelettsysteme und magnetsiche Systeme. Eingesetzt werden sie in Medizin und Unterhaltungsindustrie (Motion Capturing für Animation).

Inertialsysteme arbeiten mit Beschleunigungssensoren und Gyroskopen. Diese werden, z.B. befestigt an einem Anzug, an den zu verfolgenden Körperstellen platziert. Die gemessenen

[1] Ein interessanter nicht-optischer Ansatz „WiSEE“ wird in Pu u. a. (2013) vorgestellt. Dieser macht sich nahezu allgegenwärtige Funksignale wie z.B. ein WLAN zu Nutze. Der an den Funkwellen-Reflexionen vom Körper des Benutzers beobachtbare Doppler-Effekt erlaubt Rückschlüsse auf die durchgeführte Bewegung. Dieses Verfahren funktioniert momentan allerdings nicht für Ganzkörper-Gesten

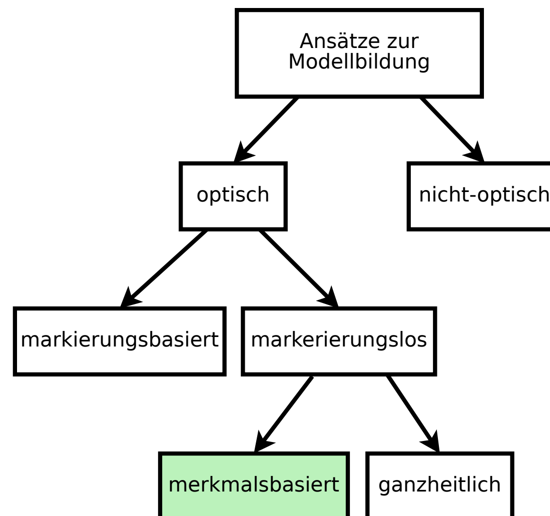


Abbildung 2.1.: Übersicht über Arten der Ansätze zur Modellbildung

Beschleunigungswerte werden anschließend analysiert und daraus die durchgeführte Bewegung gefolgert, sowie die Orientierung der Körperstellen mit Hilfe der Gyroskope bestimmt. Die Genauigkeit liegt mit Schwankungen von ca. 2% der zurückgelegten Strecke auf hohem Niveau. (Roetenberg u. a., 2009)

Exoskelettsysteme zwingen die zu betrachtende Person dazu eine Art zweites Skelett anzuziehen. Anders als bei Interatialsystemen und einigen magnetischen Systemen werden bei Exoskelettsystemen nicht bestimmte Referenzpunkte verfolgt, sondern die Winkel der Gliedmaßen an den Gelenken gemessen. Aus diesen Daten lässt sich am Rechner in Echtzeit ein Skelettmodell rekonstruiern. (Kolozs, 1998)

Magnetische Systeme nutzen magnetischen Dipole (Marker) und Magnetfeldsensoren um kontinuierlich die Position einzelner Körperpunkte der betrachteten Person zu bestimmen. Anhand der gemessenen magnetischen Flüsse können Rückschlüsse auf die Positionen der Marker getroffen werden. Selbst in kleinen Versuchsanordnungen (klein sowohl in räumlicher Ausdehnung als auch in Marker-Anzahl) betragen Schwankungen bereits ca. 13% der von einem Marker zurückgelegten Strecke. Die Praxistauglichkeit dieses Ansatzes ist daher stark anzuzweifeln. (Yabukami u. a., 2000)

Gemein haben alle Systeme, dass die zu betrachtende Person oder der zu beobachtende Raum mit mehr oder weniger teuren Sensoren bestückt werden muss. Die Gesamtkosten können so leicht mehrere 10 000 USD überschreiten (Kolozs, 1998). Verkabelung und Gewicht können ein weiteres Problem darstellen. Nicht zuletzt aus diesen Gründen kommen nicht-optische Verfahren zur Modellbildung für diese Arbeit nicht in Frage.

Markierungsbasierte optische Verfahren

Optische Verfahren im Allgemeinen benötigen zur Datenakquise in erster Linie eine oder mehrere Kameras. Allein aus den aufgenommenen Bildern wird ein der betrachteten Person entsprechendes Modell erzeugt.

Markierungsbasierte optische Verfahren erfordern es, ähnlich wie nicht-optische Verfahren, dass die zu betrachtende Person im Vorfeld gewissermaßen präpariert wird. Hierzu werden an den zu verarbeitenden Punkten des Körpers optische Markierungen angebracht. Diese können aktiv oder passiv sein, also selbst leuchten oder nur eine bestimmte Farbe oder Geometrie besitzen, die vom Aufnahmesystem zuverlässig erkannt werden kann. Aktive Systeme bieten passiven Systemen gegenüber den Vorteil, dass die einzelnen Kontrollpunkte (in der Regel LEDs) mit kurzem zeitlichen Abstand nacheinander aktiviert werden können, so dass keine Verwechselungen auftreten können. Mit weniger als 3mm Abweichung auf einer Dinstanz von 3m können markierungsbasierte Systeme nach abgeschlossener Kalibrierung eine sehr hohe Genauigkeit erreichen. Um einen menschlichen Körper in jeder Orientierung zuverlässig zu Verfolgen werden mehrere Kameras benötigt. (Zhou und Hu, 2008)

Da markierungsbasierte Verfahren ebenfalls eine Vorbereitung der zu betrachtenden Person und eine Kalibrierung des Kamerasystems erfordern, kommen auch diese für die vorliegende Arbeit nicht zur Modellbildung in Frage.

Markierungslose ganzheitliche Verfahren

Markierungslose ganzheitliche Verfahren erfordern es zum einen nicht, dass zu betrachtende Personen vor der Benutzung des Systems präpariert werden müssen. Die Aufnahme der Bewegungen erfolgt nicht anhand einzelner Merkmale, sondern das bewegte Objekt (die Person) wird als ganzes betrachtet.

Ein Beispiel für ein solches System liefert Yang u. a. (2002). Auf Basis eines Eingabevideos werden dort zunächst Frame für Frame Pixeltransformationen bestimmt, die die Position eines Pixels aus einem Frame auf seine Position im nächsten Frame abbilden. Die Verkettung dieser Transformationen beschreibt schließlich die Bewegung eines Pixels im Verlaufe des Eingabeviedos. In einem weiteren Schritt werden dann gleichartige Pixelbewegungen zu Bewegungen von Pixelgruppen zusammengefasst.

Dieses Verfahren besticht zunächst durch seinen universellen Ansatz. Unklar ist jedoch, wie sich dieses Verfahren für Echtzeit-Viedodaten eignet. Da im Rahmen der vorliegenden Arbeit der Aufzeichnungs-/Erkennungsschritt auf Grundlage eines Skelettmodells arbeiten soll,

eignen sich derartige ganzheitliche Verfahren jedoch nicht, denn die festgestellten Beweungen werden nicht weiter abstrahiert indem sie z.B. bestimmten Körperregionen zugeordnet würden.

2.1.2. Optische markierungslose merkmalbasierte Verfahren

Für die Erstellung eines Skelettmodells unter den für diese Arbeit vorgesehenen Rahmenbedingunen (Keine Vorbereitung der Benutzer, Echtzeit, geringe Kosten, ...) eignen sich optische markierungslose merkmalbasierte Verfahren am besten. Diese Verfahren benötigen als Sensor lediglich ein oder mehrere Kamerasysteme. Es müssen keine Markierungen an den Benutzern angebracht werden und man erhält trotzdem Informationen über konkrete Merkmale der Benutzer.

Zhou und Hu (2008) stellt eine Reihe solcher Verfahren verschiedenster weiterer Unterkategorien vor und ist in jedem Fall beachtenswert. Den „Durchbruch" in diesem Bereich brachte allerdings erst zwei Jahre später eine Kooperation zwischen Microsoft und PrimeSense mit dem Kinect-Sensor. Dieses Kamerasystem ist in der Lage ein Echtzeit-Skelettmodell mehrer Benutzer bereitzustellen und zeichnet sich durch geringe Kosten und eine sehr einfache Handhabung aus. Es eignet sich damit hervorragend zur Datenakquise im Rahmen dieser Arbeit, weshalb im Folgenden kurz beschrieben werden soll, wie dieses Skelett-Modell zu Stande kommt. Details hierzu finden sich in den dieser Beschreibung zu Grunde liegenden Publikationen Zhang (2012) und Shotton u. a. (2013).

Aufnahme der Rohdaten

Relevant für die Modellbildung ist hauptsächlich ein von PrimeSense entwickelter Tiefensensor bestehend aus Infrarot-Projekter und Infrarot-Kamera. Der Projektor wirft permanent ein Muster aus (für das menschliche Auge nicht sichtbaren) Lichtpunkten auf die Umgebung. Das projezierte Muster sowie der Abstand zwischen Projektor und Kamera sind bekannt. Kann man einen mittels der Kamera aufgenommenen Punkt einem projezierten Punkt zuordnen, ist es möglich, daraus per Triangulation die Position des Punktes in drei Dimensionen zu bestimmen. Abbildung 2.2 zeigt das projezierte Muster und das resultierende Tiefenbild.

Segmentierung

Um die aufgenommenen Rohdaten in ein Nutzbares Modell zu überführen bedarf es einer sog. Segmentierung. Segmentierungsverfahren dienen dazu, die Pixel der Roh-Bilddaten nach bestimmten Kriterien in unterschiedliche Segmente zu gruppieren und die Rohdaten so mit Hinblick auf die jeweilige Anwendung zu abstrahieren. Für die Hand-Gestenerkennung würde

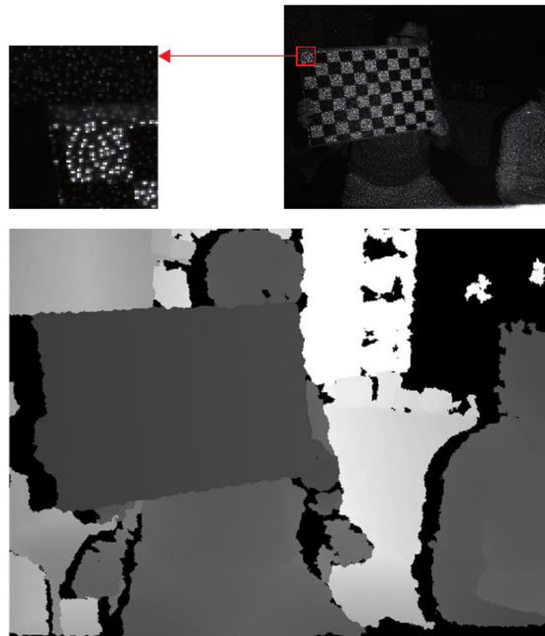


Abbildung 2.2.: Projektion und Tiefenbild: Oben projiziertes Muster (Vergrößerter Auschnitt oben links), unten Tiefenbild der Szene (Zhang, 2012, ©2012 IEEE)

ein geeignetes Segmentierungsverfahren die Pixel, die die verschiedenen Finger darstellen, so zuordnen, dass man 4 (bzw. 5 inkl. Daumen) Segmente erhält. Die Wahl eines passenden Segmentierungsverfahrens hängt von der Art der Rohdaten und den Anforderungen an das für die jeweilige Anwendung geeignete Datenmodell ab.

Für die Kinect ist das angestrebte Datenmodell ein Skelettmodell des Benutzers. Das eingesetzte Verfahren extrahiert also aus dem vom Tiefensensor bereitgestellten Tiefenbild dreidimensionale Modelle der Benutzer-Skelette. Diese Skelettmodelle setzen sich aus Punkten zusammen, die mit wichtigen Gelenken oder Körperteilen der Benutzer assoziiert sind. Solche Punkte sind die Merkmale, auf die sich optische markierungslose merkmalbasierte Verfahren zur Modellbildung stützen.

Zur Konstruktion des Skelettmodells wurde ausgehend vom aufgenommenen Tiefenbild eine mehrstufige Herangehensweise gewählt:

1. Für jeden Pixel wird bestimmt, zu welchem Körperteil er gehören könnte. Das geschieht mit vorab trainierten Entscheidungswäldern.

2. Informationen der einzelnen Pixel werden zusammengefasst und aus den entstehenden Gruppen die wahrscheinlichen Merkmalspositionen bestimmt.

3. Diese werden unter Berücksichtigung von durch das menschliche Skelett festgelegten Vorbedingungen den Merkmalen eines Skelettmodells zugeordnet.

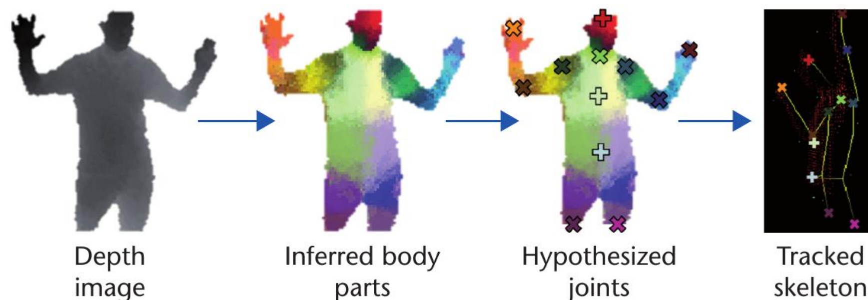


Abbildung 2.3.: Veranschaulichung der Segmentierungsschritte (Zhang, 2012, ©2012 IEEE)

Abbildung 2.3 veranschaulicht diesen Prozess. Die Segmentierung ist damit abgeschlossen. Aus den durch den Tiefensensor aufgenommenen Rohdaten wurde ein abstraktes, den Benutzer repräsentierendes Skelettmodell bestimmt. Anhand dieses Modells lassen sich die Bewegungen des Benutzers hervorragend nachvollziehen und weiterverarbeiten.

Einschränkungen

Die Verwendung eines Kinect-Sensors zieht allerdings auch einige Einschränkungen für den jeweiligen Einsatzzweck nach sich. So können Tiefeninformationen nur in einem bestimmten Entfernungsbereich vom Sensor gewonnen werden, so dass außerhalb dieses Bereichs dann auch keine Benutzer erkannt werden können. Dieser Bereich liegt je nach Sensor-Generation zwischen 40cm und 4,5m Entfernung vom Sensor. Im sog. „Near Mode“, der zumindest von der „Kinect for Windows“ angeboten wird und der aktiviert werden muss, um Objekte näher als 80cm zu erfassen, kann jedoch für Personen kein Skelettmodell mehr erzeugt werden. Es ist nur ihr Körpermitellpunkt verfügbar. (Eisler, 2012)

Da der Tiefensensor mit infrarotem Licht arbeitet ist er anfällig gegenüber Störungen durch Sonnenlicht und kann z.B. gläserne Objekte schlecht wahrnehmen. Ein Betrieb unter freiem Himmel, insbesondere bei direkter Sonneneinstrahlung ist daher nicht zuverlässig möglich. (Microsoft Corporation; RoboRealm)

Die Granularität des Sensors erlaubt es nicht, einzelne Finger zu erkennen, wie es z.B. „LEAP Motion“ (Leap Motion, 2014) tut – welches sich jedoch auf Hände beschränkt und keinerlei Informationen über den Rest des Körpers liefert. Eine Erkennung von Handgesten ist mit dem von der Kinect bereitgestellten Skelettmodell daher nicht möglich.

2.1.3. Zusammenfassung

Der vorangehende Abschnitt lieferte einen groben Überblick über verschiedene Ansätze zur Modellbildung mit einer Bewertung des Nutzens für die hier vorliegende Aufgabenstellung. Es wurde klar, dass aktuell nur optische markierungslose merkmalsbasierte Verfahren die Hauptanforderung der vorbereitungslosen Nutzbarkeit des zu entwickelnden Toolkits durch verschiedene Benutzer erlaubt. Exemplarisch für Geräte dieser Kategorie wurde der relativ erschwingliche und auch deshalb für diese Arbeit gut geeignete Kinect-Sensor von Microsoft etwas näher vorgestellt. Der Schwerpunkt lag dabei auf den Funktionen, die für die Erzeugung des durch die Kinect bereitgestellten Skelettmodells verantwortlich sind. Es wurde außerdem aufgezeigt, welche Einschränkungen es bei der Verwendung einer Kinect zu beachten gilt um einen möglichst problemlosen Betrieb zu erreichen.

2.2. Verfahren zur Gestenaufzeichnung und -wiedererkennung

Das im vorherigen Schritt gewonnene Modell stellt als Grundlage für die weitere Verarbeitung ein Modell bereit, in dem sich eine Reihe besonderer Merkmale des menschlichen Körpers verfolgen lassen. Anhand dieser Merkmale ist es möglich das Bewegungsverhalten eines Benutzers weiter zu analysieren.

Das Problem ist es nun, diese Bewegungen zu klassifizieren. Grundsätzlich spielt es hier nur eine untergeordnete Rolle, ob dabei nur ein einziges beobachtetes Merkmal (z.B. eine Hand) oder eine Reihe von Merkmalen (z.B. ein Skelettmodell aus mehreren Punkten) betrachtet werden. Für die festgestellte Bewegung eines Punktes gilt es zu bestimmen, was diese Bewegung ausmacht und wie man sie von anderen Bewegungen abgrenzen kann. Das Ergebnis dieser Bestimmung kann als aufgezeichnete, als „gelernte“ Geste betrachtet werden. Zur Wiedererkennung von Gesten kann dann ein zu betrachtender Bewegungsablauf mit einem angelernten Gesten-Repertoire verglichen und entsprechend klassifiziert werden.

Es sei hier noch einmal daran erinnert, dass der Blick zunächst auf die Aufgabe mit dem höchsten Abstraktionsniveau, die Gestenverarbeitung, gerichtet wird (vgl. Seite 5). Die Betrachtung wird erst an den Punkten in Richtung Posenverarbeitung vertieft, an denen diese Betrachtung auch sinnvoll ist.

Das Aufzeichnen und Wiedererkennen von Gesten wird in verschiedenen Anwendungsbereichen benötigt. Es muss sich dabei auch längst nicht immer direkt um den menschlichen Körper und seine Bewegungen drehen. Man denke z.B. an die stark auf Gesten basierende Bedienung von Smartphones mit Touchscreen (Rotieren, Scrollen, Zoomen, …). Der Opera

Browser unterstützte Maus-Gesten zur Navigation bereits im Jahre 2001 (Opera Software ASA, 2001).

Dieser Abschnitt soll einen Überblick über verschiedene Verfahren zur Gestenaufzeichnung und -wiedererkennung bieten. Hierbei werden die Grundsätze der Erkennungsmechanismen und der zugehörigen individuellen Lern- oder Trainingsverfahren erläutert, sowie eine Bewertung mit Hinblick auf die Tauglichkeit für die hier vorliegende Arbeit gegeben.

Um die folgenden Beschreibungen möglichst allgemein zu halten ist oft von „Symbolen“ oder „Symbolfolgen“ die Rede. Als ein Symbol würde in unserem Anwendungsfall die räumliche Position eines Merkmals im Sinne des vorangehenden Abschnitts bezeichnet werden, als eine Symbolfolge die Positionen zu verschiedenen Zeitpunkten. Für andere Anwendungen kann ein Symbol allerdings etwas Grundsätzlich anderes repräsentieren, wie z.B. eine gemessene Frequenz eines Radiosignals.

2.2.1. Dynamic Time Warping (DTW)

DTW ist ein Verfahren zur Bestimmung einer optimalen Zuordnung zwischen zwei Symbolfolgen, wobei die Zeitachse der einen Symbolfolge im Vergleich zur anderen verzerrt sein kann. Aus einer gefundenen Zuordnung, einem sog. „Warping“, lässt sich dann die Ähnlichkeit der beiden Symbolfolgen beurteilen (vgl. Abbildung 2.4). Einsatz finden dieses und darauf aufbauende Verfahren vor allem in der Spracherkennung (Sakoe und Chiba, 1978).

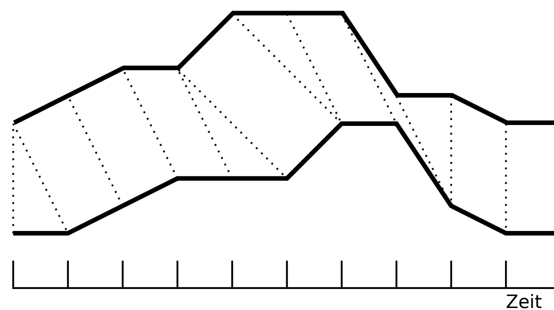


Abbildung 2.4.: Einfache DTW-Veranschaulichung

Anschaulich lässt sich eine solche Zuorndung zwischen zwei Symbolfolgen $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_n)$ folgendermaßen bestimmen: Es wird eine Matrix konstruiert, deren Elemente $d_{ij} = D(a_i, b_j)$ mit $i, j \in [1; n]$ und $n \in \mathbb{N}$ die Abweichung zwischen je zwei Symbolen a_i und b_j beinhalten. D ist hierbei irgendeine Funktion, die ein für die jeweilige Anwendung sinnvolles Maß für die Abweichung berechnet, wobei sie monoton sein sollte und kleinere Werte eine kleinere Abweichung bedeuten. Diese so erzeugte Matrix kann als

Adjazenzmatrix eines Graphen aufgefasst werden. Berechnet man in diesem Graphen den kürzesten Weg von $d_{0,0}$ nach $d_{n,n}$, so erhält man damit eine optimale Abbildung der beiden Signale aufeinander (Wenn der kürzeste Weg $(d_{1,1}, d_{1,2}, d_{2,2}, d_{3,3})$ ist, dann erhielte man die Abbildung $(a_1 \mapsto b_1, a_1 \mapsto b_2, a_2 \mapsto b_2, a_3 \mapsto b_3)$). Die Knotenauswahl bei der Wegberechnung unterliegt dabei Einschränkungen: Generell sollen für beide Symbolfolgen jeweils alle Indizes auch verwendet werden. Außerdem lassen sich für die jeweilige Anwendung praktische Regeln formulieren, wie z.B., dass die Indizes i und j maximal um je 1 erhöht werden dürfen (sog. „Slope Constraints“). Entsprechend der gewählten Slope Constraints ist es selbstverständlich nicht notwendig, die Adjazenzmatrix vollständig zu befüllen. Über die Länge des gefundenen Weges lässt sich für verschiedene Eingaben beurteilen, wie dicht sie an der Referenzsymbolfolge liegen. Je kürzer der gefundene Weg, desto besser die Abbildung. (Sakoe und Chiba, 1978; Turetsky und Ellis, 2003)

DTW-Verfahren müssen nicht im üblichen Sinne trainiert werden. Sie vergleichen lediglich zwei Signale bzw. Symbolfolgen. Das antrainieren eines Gestenrepertoires würde auf das Bilden von vielen Referenzsymbolfolgen hinauslaufen, mit denen Eingabesymbolfolgen dann per DTW verglichen werden. Anhand der Güte der bestimmten Warpings (Länge der gefundenen Wege) lässt sich dann die am besten passende Referenzsymbolfolge feststellen.

Der Nachteil des klassischen DTW besteht darin, dass immer nur in der zeitlichen Dimension nach einer Zuordnung gesucht wird. Schwankungen in den Signalen selbst („y-Achse“) führen dazu, dass sehr große Warpings bestimmt werden können, die dann tatsächlich auch gar nicht mehr korrekt sind (Keogh und Pazzani, 2001). Abbildung 2.5 illustriert dieses Problem.

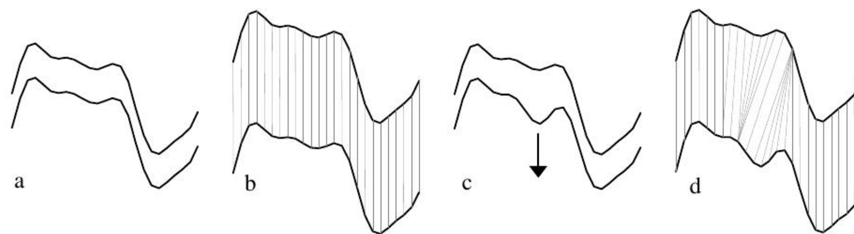


Abbildung 2.5.: Problem mit klassichem DTW: Zwei identische Signale (a) erhalten durch klassisches DTW die korrekte 1:1-Zuorndung (b). Verschiebt man ein Merkmal einer Kurve (c), dann versucht DTW das über die Zeitachse auszugleichen und produziert ein falsches Warping (d). (Keogh und Pazzani, 2001, Genehmigung im Anhang D.1.1)

Keogh und Pazzani (2001) stellt einen Lösungsansatz „Derivative Dynamic Time Warping“ (DDTW) vor, bei dem die Steigung an einem Punkt an Stelle der sonst oft üblichen euklidischen

Distanz verwendet wird um zwei Signale miteinander zu Vergleichen. Signale mit starkem Rauschen werden zuvor noch geglättet. Abbildung 2.6 zeigt einen Vergleich zwischen DTW und DDTW.

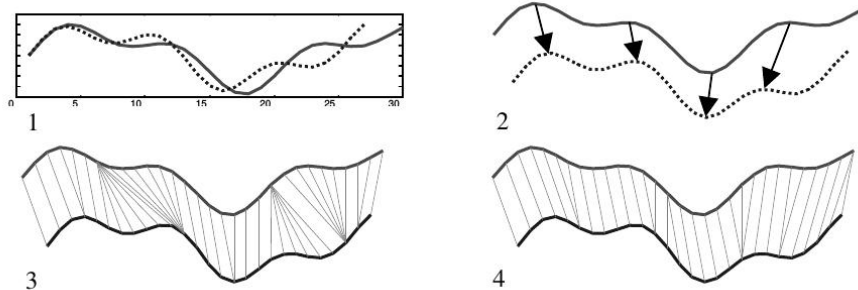


Abbildung 2.6.: Klassiches DTW und DDTW: (1) Zwei Signale. (2) Intuitive Zuordnung. (3) Ergebnis mit klassischem DTW. (4) Ergebnis mit DDTW. (Keogh und Pazzani, 2001, Genehmigung im Anhang D.1.1)

Andere Verfahren wie „SparseDTW“ (Al-Naymat u. a., 2009) und „FastDTW“ (Salvador und Chan, 2007) zielen darauf ab DTW zu beschleunigen und die Zeit- und Speicherkomplexität zu verringern. Unter der Annahme, dass Referenz- und Eingabesignal beide der Länge n sind, liegt diese beim klassischen DTW und bei SparseDTW bei $\mathcal{O}(n^2)$, bei FastDTW bei $\mathcal{O}(n)$ (Bei allen sowohl Zeit- als auch Speicherkomplexität). (Al-Naymat u. a., 2009; Salvador und Chan, 2007)

Für diese Arbeit wird ein Verfahren gesucht, das vollkommen umempfindlich gegenüber Schwankungen im Verlauf von Vergleichs- und Eingabedaten ist. Bei allen hier vorgstellten DTW-Varianten grenzt man die erlaubte Schwankung durch Slope Constraints ein. Je großzügiger diese Eingrenzung, desto rechenintensiver wird die Bestimmung eines Warpings. Für die vorliegende Arbeit käme DTW höchstens mit sehr weiten Slope Constraints in Frage, würde hier also viel Rechenleistung erfordern. Desweiteren soll das erkennbare Gestenrepertoire natürlich möglichst groß sein, was den Rechenaufwand nochmals vervielfachen würde. DTW-Verfahren erscheinen daher für diese Arbeit ungeeignet und werden nicht weiter betrachtet.

2.2.2. Hidden Markov Models (HMM)

Grundlagen eines HMM bilden sog. Markov-Ketten. Eine Markov-Kette ist ein gedächtnisloses System, bei dem Übergänge zwischen Zuständen mit einer bestimmten Wahrscheinlichkeit stattfinden. Diese Wahrscheinlichkeit ist i.d.R. nur abhängig vom jeweils aktuellen Zustand und über die Zeit konstant. In einem HMM sind den einzelnen Zuständen Ausgabesymbole, die mit einer definierten Wahrscheinlichkeit im jeweiligen Zustand zu beobachten sind,

zugeordnet. Die Zustände der zu Grunde liegenden Markov-Kette und ihre Übergangswahrscheinlichkeiten selbst sind unbekannt. Eine zu erkennende Geste würde durch die möglichen Ausgaben eines auf diese Geste trainierten HMMs modelliert werden. Für eine zu untersuchende Symbolfolge kann dann die Wahrscheinlichkeit bestimmt werden, mit der ein HMM diese Ausgabesymbolfolge erzeugt hat. (Elliott u. a., 1994; Rabiner, 1989)

Charakteristika eines HMM und Erzeugung von Symbolfolgen

Ein konkretes HMM wird nach (Rabiner, 1989) durch eine Reihe verschiedener Parameter charakterisiert:

- Zustandsanzahl N: Auch wenn diese Zustände versteckt sind beeinflusst das zu lösende Problem meist die Wahl der Zustandsanzahl, da die Wahrscheinlichkeiten für das Auftreten bestimmter Ausagabesymbole sich nur von Zustand zu Zustand unterscheiden kann.
- Zustandsmenge $S = \{S_1, \ldots, S_N\}$. q_t bezeichnet den zum Zeitpunkt t aktuellen Zustand $S_?$.
- Alphabetgröße M: Die Anzahl der beobachtbaren Ausgabesymbole
- Ausgabealphabet $V = \{v_1, \ldots, v_M\}$
- Wahrscheinlichkeitsverteilung für Zustandsübergänge $A = \{a_{ij}\}$,
  mit $a_{ij} = P\left(q_{t+1} = S_j | q_t = S_i\right), \quad 1 \leq i, j \leq N$
- Wahrscheinlichkeitsverteilung für die Produktion der Ausgabesymbole $B = \{b_j(k)\}$,
  mit $b_j(k) = P\left(v_k \text{ bei Zeitpunkt } t | q_t = S_j\right), \quad 1 \leq j \leq N$ und $1 \leq k \leq M$
- Wahrscheinlichkeitsverteilung für den Startzustand $\pi = \{\pi_i\}$,
  mit $\pi_i = P\left(q_1 = S_i\right), \quad 1 \leq i \leq N$

Eine Ausgabesymbolfolge $O = O_1 O_2 \ldots O_T$ (Mit T als Gesamtzahl der beobachteten Symbole) kann dann auf Grundlage eines entsprechend charakterisierten HMMs wie folgt erzeugt werden:

1. Anfangszustand $q_1 = S_i$ entsprechend π auswählen
2. Setze $t = 1$
3. Setze $O_t = v_k$ unter Berücksichtigung der Ausgabewahrscheinlichkeit $b_i(k)$ in Zustand S_i

4. Gemäß A in Folgezustand $q_{t+1} = S_j$ übergehen

5. Setze $t = t + 1$

6. Wenn $t < T$, gehe zu Schritt 3, sonst beende

Eine kompaktere aber dennoch vollständige Charakterisierung eines HMM λ kann als $\lambda = (A, B, \pi)$ geschrieben werden. Im Kontext der Gestenerkennung gilt es nun für dieses Modell zwei entscheidende Probleme zu lösen:

Erkennungsproblem: Wie kann die Wahrscheinlichkeit, dass eine beobachtete Ausgabesymbolfolge O von einem HMM λ erzeugt wurde, (möglichst effizient) berechnet werden?

Trainingsproblem: Wie können wir die Parameter eines HMM $\lambda = (A, B, \pi)$ so anpassen, dass wir die Wahrscheinlichkeit $P(O|\lambda)$ dafür, dass eine vorgegebene Ausgabe O von diesem Modell λ erzeugt wird, maximieren?

Bevor ein HMM aber zur Gestenerkennung eingesetzt werden kann müssen die zwei- oder dreidimensioneln Daten auf eine Dimension reduziert werden. Diese eindimensionale Menge bildet dann das Ausgabealphabet. Diesem Zweck dienen verschiedene Clustering-Verfahren (Chen u. a., 2003). Abbildung 2.7 veranschaulicht ein triviales Clustering für eine zweidimensionale Aufnahme. Jeder Punkt der lilafarbenen Linie markiert einen Zeitschritt. Durch das Clustering wird die Koordniaten-Folge $(1, 3)(2, 4)(3, 3)(4, 2)(4, 2)$ auf die Ausgabesymbolfolge „k q m i i" abgebildet. Das Ausgabealphabet besteht also aus den 25 ersten Buchstaben des Alphabets und die eingezeichnete Geste ist eine Eingabe der Länge 5. In der Praxis werden effektivere Methoden verwendet, wie z.B. der „k-Means-Algorithmus" (Kanungo u. a., 2002).

Erkennungsproblem: Forward-Algorithmus

Um zu berechnen, mit welcher Wahrscheinlichkeit ein HMM eine Symbolfolge $O = O_1, \dots, O_T$ produziert, kann der sog. „Forward-Algorithmus" verwendet werden. Hier wird Schritt für Schritt die Wahrscheinlichkeit berechnet, dass zu einem Zeitpunkt $1 \leq t \leq T$ eine Ausgabe O_t erzeugt wird. Die vorangehenden Ausgabesymbole fließen jeweils mit in das Ergebnis ein, indem jeweilige Zustandsübergangs- und Symbolausgabewahrscheinlichkeiten berücksichtigt werden. Im Detail wird wie folgt verfahren:

Initialisierung: $\alpha_1(i) = \pi_i b_i(O_1), \quad 1 \leq i \leq N$

Rekursion: $\alpha_{t+1}(j) = \left(\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right) b_j(O_{t+1}), \quad 1 \leq t \leq T - 1$ und $1 \leq j \leq N$

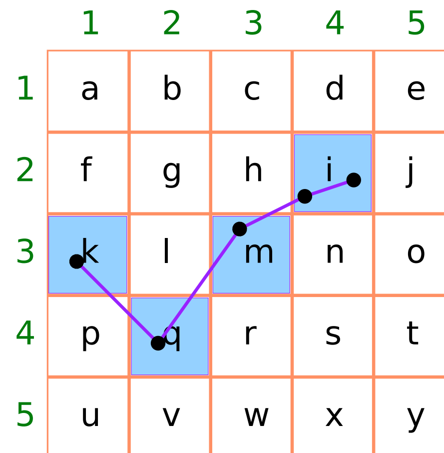


Abbildung 2.7.: Veranschaulichung eines trivialen Clusterings

Terminierung: $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$

Anhand der so berechneten Wahrscheinlichkeit für die Ausgabe einer eine Geste repräsentierenden Symbolfolge O durch ein HMM λ lässt sich zweierlei feststellen: Zum einen kann die Frage beantwortet werden (z.B. durch Vergleich mit einem definierten Schwellenwert), ob die in λ modellierte Geste beobachtet wurde. Zum andern können einzelne Gesten λ_i eines ganzen Gestenrepertoires $\Lambda = \{\lambda_1, \ldots, \lambda_G\}$ untereinander verglichen werden.

Das Erkennungsproblem kann ebenfalls mit dem sog. „Backward-Algorithmus“ gelöst werden. Die Implementierung gestaltet sich ähnlich zum Forward-Algorithmus, die beobachtete Ausgabe wird allerdings von hinten nach vorne verarbeitet. Details können Rabiner (1989) entnommen werden.

Trainingsproblem: Baum-Welch-Algorithmus

Die Trainingsverfahren lassen sich Grundsätzlich in zwei Kategorien einteilen: Offline und Online. Während Offline-Verfahren dazu dienen, ein HMM im Vorfeld seines Einsatzes zu trainieren, werden Online-Verfahren eingesetzt um ein HMM „während des Betriebes“ weiter zu trainieren. (Khreich u. a., 2012)

Der Baum-Welch-Algorithmus ist vermutlich der bekannteste Trainingsalgorithmus und gehört zur Familie der Offline-Verfahren. Er wird verwendet, um die Übergangs- und Ausgabewahrscheinlichkeiten eines HMMs $\lambda = (A, B, \pi)$ so anzuapssen, dass die Wahrscheinlichkeit $P(O|\lambda)$ der Ausgabe einer gegebenen Symbolfolge O, dem Trainingsdatensatz, maximiert

wird und soll hier exemplarisch vorgestellt werden. Weitere Trainingsverfahren werden z.B. in Juang und Rabiner (1990), (Elliott u. a., 1994, S. 38–40) und (Khreich u. a., 2012) beschrieben.

Zur Anwendung des Baum-Welch-Algorithmus wird die Variable β benötigt, die vom Backward-Algorithmus auf ähnliche Weise befüllt wird, wie die α-Variable durch den Forward-Algorithmus. Es werden außerdem folgende Funktionen eingeführt:

- Die Wahrscheinlichkeit, dass sich das Modell λ zur Zeit t bei einer Eingabe O im Zustand S_i befindet:
  $\gamma_t(i) = P(q_t = S_i | O, \lambda) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)}$
- Die Wahrscheinlichkeit, dass sich das Modell λ zur Zeit t bei einer Eingabe O im Zustand S_i, und zum Zeitpunkt $t+1$ im Zustand S_j befindet:
  $\xi_t(i,j) = P(q_t = S_i, q_{t+1} = S_j | O, \lambda) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)}$

Die Matrizen $\overline{A}$ und $\overline{B}$ mit Elementen $\overline{a}_{ij}$ und $\overline{b}_{ij}$, die die angepassten Wahrscheinlichkeitsverteilungen für Zustandsübergänge und Symbolausgaben enthalten, werden dann wie folgt berechnet.

$$\overline{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \quad \text{und} \quad \overline{b}_j(k) = \frac{\sum_{t=1}^{T} I_{O_t=v_k} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \quad \text{mit} \quad I_{O_t=v_k} = \begin{cases} 1 & \text{wenn } O_t = v_k \\ 0 & \text{sonst} \end{cases}$$

Mit $\overline{\pi}_j = \gamma_1(j)$ erhalten wir damit ein HMM $\overline{\lambda} = (\overline{A}, \overline{B}, \overline{\pi})$, so dass $P(O|\lambda) < P(O|\overline{\lambda})$. Nun kann man den Baum-Welch-Algorithmus erneut auf das bereits angepasste HMM $\overline{\lambda}$ anwenden, um das Ergebnis weiter zu verbessern. Dies wiederholt man bis der Unterschied zwischen den Wahrscheinlichkeiten $\Delta = \left|P(O|\lambda) - P(O|\overline{\lambda})\right|$ hinreichend klein wird.

Bewertung

HMMs finden ein breites Anwendungsspektrum bei stochastischen Mustererkennungsproblemen, oft mit problemspezifischen Abwandlungen oder Einschränkungen (Yang u. a., 2002). Sie finden z.B. Einsatz in der Spracherkennung (z.B. Rabiner, 1989), der Bioinformatik (z.B. Krogh u. a., 2001) und eben auch der Gestenerkennung (z.B. Yamato u. a. (1992) und Starner und Pentland (1997)).

Hervorragende Erkennungsraten lassen sich allerdings am ehesten unter Erweiterung durch domänenspezifisches Wissen erreichen, mit dem auch der Trainingsaufwand deutlich verringert werden kann (Mitra und Acharya, 2007). Solches Wissen soll im Rahmen dieser Arbeit jedoch möglichst nicht verwendet werden: Es sollen sich z.B. natürliche Einschränkungen

des menschlichen Skeletts hier nicht niederschlagen, wodurch die Erwartungen an die Erkennungsleistung reduziert werden müssten und der Trainingsaufwand wächst. Außerdem ist die Wahl der Ausgabealphabets-Größe und die Anzahl der versteckten Zustände wichtig für die Erkennungsleistung eines HMM (Chen u. a., 2003). Das ausprobieren unterschiedlicher Parameter für jedes HMM und damit für jede Geste bedeutet hier noch mehr Aufwand während der Trainingsphase.

Anhand der Beschreibung in diesem Abschnitt sollte sich nachvollziehen lassen, dass die Größe der Zeitschritte (gewissermaßen die Samplingrate bei der Aufzeichnung) mit Bedacht und passend zur Geste gewählt werden muss, denn sie bestimmt auch die Geschwindigkeit, mit der eine Geste ausgeführt werden darf. Werden die Zeitschritte groß gewählt, weil das HMM mit langsamen Ausführungen trainiert wurde, dann wird dieses HMM die selbe Geste bei sehr schneller Ausführung nicht erkennen können, da große Teile der Bewegung zwischen den Zeitschritten „untergehen". Im umgekehrten Fall würde die Erkennung daran scheitern, dass jedes Ausgabesymbol (also die Position eines verfolgten Merkmals) sich von Aufnahme zu Aufnahme kaum ändern würde.

Insgesamt ist für die effektive Nutzung von HMMs zur Gestenerkennung einiger Sachverstand erforderlich, weshalb die Realisierbarkeit der angestrebten Einfachheit eines auf ihnen basierenden Tools zweifelhaft erscheint. Die Empfindlichkeit gegenüber Schwankungen in der Ausführungsgeschwindigkeit von Gesten stellt ein weiteres Problem dar. Aus diesen Gründen kommen HMMs als Grundlage dieser Arbeit nicht in Frage.

2.2.3. Time-Delay Neural Networks (TDNN)

Ein künstliches neuronales Netz ist angelehnt an die Funktionsweise eines Gehirns. Es ist eine Sammlung einzelner oft sehr einfach gehaltener Recheneinheiten (Neuronen, Knoten), die untereinander mit gewichteten Kanten verbunden sind. Diese Recheneinheiten ermitteln zunächst die gewichtete Summe ihrer Eingaben (Übertragungsfunktion) und wenden auf diese dann eine nicht-lineare Funktion an (Aktivierungsfunktion). Als Aktivierungsfunktion kommt oft eine einfache Schwellenwert- oder Sigmoidfunktion zum Einsatz (oder eine Kombination). Die Mächtigkeit solcher Netze entsteht also nicht in den einzelnen Neuronen, sondern in ihrer Dichte bzw. Anzahl und ihrer Vernetzung. Die Bedingungen für das Erzeugen einer Ausgabe, sowie die Gewichte der Verbindungen sind variabel, so können solche Netze „aus Erfahrung" lernen. Das erlangte Wissen des Netzes spiegelt sich nicht in einzelnen Neuronen wieder, sondern in der Gesamtheit des Netzes. (Cross u. a., 1995; Waibel u. a., 1989)

Bei TDNNs handelt es sich um mehrschichtige, künstliche neuronale Netze, sog. „multi-layer feed-forward networks" (MLFN) (Yang u. a., 2002). In MLFN sind die Neuronen in Schichten

angeordnet, so dass jedes Neuron mit allen Neuronen der darüber liegenden Schicht verbunden ist. Die Grenzen des Systems sind die Eingabe- und Ausgabeschichten. Zwischen diesen Schichten können mehrere versteckte Schichten angesiedelt sein. Abbildung 2.8 illustriert den Aufbau eines solchen MLFN beispielhaft. (Svozil u. a., 1997)

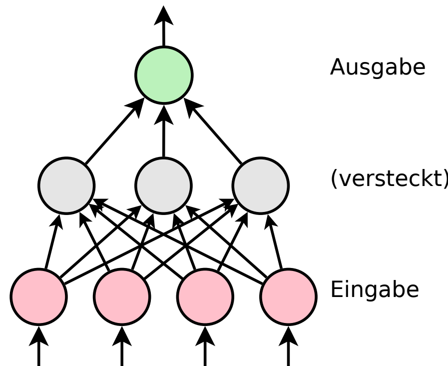


Abbildung 2.8.: Aufbau eines MLFN

Aufbau und Funktion

Die Besonderheit von TDNNs liegt in ihrer Toleranz gegenüber zeitlichen Schwankungen in der Abfolge der Eingabesymbole. Um diese Toleranz zu erreichen werden die aktuelle und die N vorangehenden Eingaben gleichzeitig an den Eingängen der Neuronen bzw. ihnen vorgelagerte Verzögerungselemente D_n angelegt (vgl. Abbildung 2.9). Die Anzahl der gleichzeitig angelegten Eingaben ist die sog. Fenstergröße $S = N+1$, welche sich von Schicht zu Schicht unterscheiden kann. Die Verzögerungselemente können die einzelnen Zeitschritte im Fenster unterschiedlich gewichten. Hat ein Neuron normalerweise z.B. $J = 8$ Eingaben und werden neben dem aktuellen Zeitschritt auch die beiden vorangehenden betrachtet, so ergibt sich die Fenstergröße S zu 3, so dass $J \times S = 3 \times 8 = 24$ Eingaben gleichzeitig verarbeitet werden. (Waibel u. a., 1989)

Durch dieses Verfahren wird immer nur ein der Fenstergröße entsprechender Ausschnitt der Eingabe-Symbolfolge betrachtet wird, was wiederrum zur Folge hat, dass das Netz viele lokale Entscheidungen trifft und damit gegenüber kleineren Verschiebungen gewisser Merkmale in einer Eingabe-Symbolfolge unempfindlich ist. Wie groß diese Unempfindlichkeit ist wird durch die Fenstergröße(n) festgelegt. An diesem Punkt wird der Nutzen einer von Schicht zu Schicht ggf. unterschiedlichen Wahl von Neuronenanzahl und Fenstergröße deutlich: Jede weitere Ebene ist in der Lage die vielen lokalen Entscheidungen der vorangehenden Ebene weiter zu abstrahieren und globalere (oder „weniger lokale“) Schlüsse zu ziehen. Jitter und Rauschen in

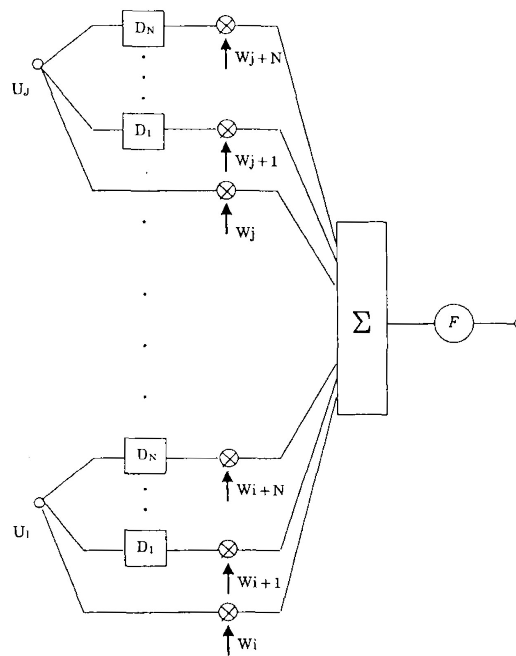


Abbildung 2.9.: Veranschaulichung der Verzögerungselemente in einem TDNN(Waibel u. a., 1989, ©1989 IEEE)

der ursprünglichen Eingabe sind also von Schicht zu Schicht weniger spürbar. Abbildung 2.10 illustiert solch eine Schichtung und die Bewegung des Betrachtungsfensters („Shift window") über die Eingabe. (Waibel u. a., 1989)

Die Eingaben der einzelnen Schichten werden i.d.R. in einer Matrix mit je Schicht (durch die Fenstergröße) festgelegter Spaltenzahl abgelegt, wobei neue Samples rechts eingefügt und zum Einhalten der maximalen Spaltenzahl nötigenfalls alte Samples links verworfen werden. Eine Spalte stellt einen Zeitschritt dar und kann mehrere Werte (z.B. X-, Y- und Z-Koordinaten) enthalten. In den versteckten Schichten sind die Neuronen ebenfalls spaltenweise angeordnet. Ihre Eingaben sind die Ausgaben der darunter liegenden Schicht (wieder mit Verwendung von Verzögerungselementen). Die Ausgabeschicht summiert die über die Zeit angefallenen Ausgaben der letzten versteckten Schicht und ist damit in der Lage die ursprüngliche Eingabe mit Hinblick auf das im Vorfeld „Erlernte" zu klassifizieren. (Waibel u. a., 1989)

Training

Trainiert werden kann solch ein Netz z.B. per „Error-Backpropagation". Dabei handelt es sich um eine Anwendung des Gradientenverfahrens („Gradient descent"), die das mittlere

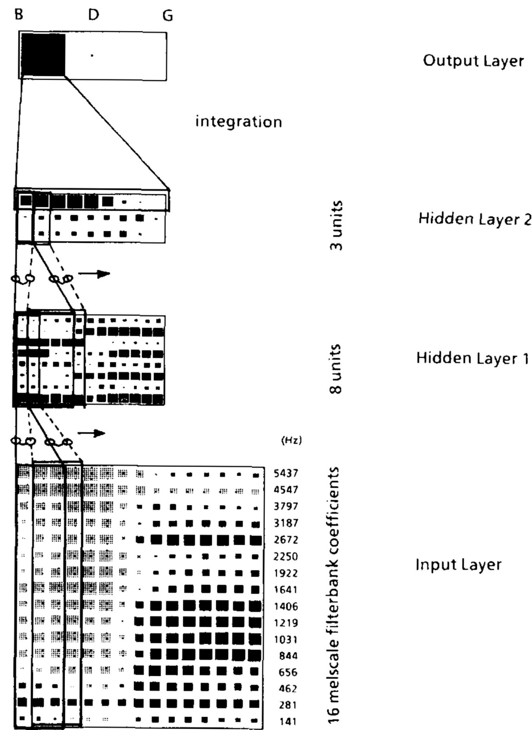


Abbildung 2.10.: Schichten eines TDNN mit Ein- und Ausgaben, sowie dargestelltem Fenster („Shift window“) (Waibel u. a., 1989, ©1989 IEEE)

Fehler-Quadarat zwischen Ein- und Ausgabe als Funktion der Gewichte minimiert. (Waibel u. a., 1989)

Zu Beginn eines Trainingslaufs wird ein Eingabe-Vektor an die Eingabeschicht angelegt. Die (über alle Eingänge kumulierte) Eingabe x_p eines Neurons p ist eine lineare Funktion von den Ausgaben y_q, die mit p über Kanten mit dem Gewicht w_{pq} verbunden sind:

$$x_p = \sum_q y_q w_{pq}$$

Ziel ist es, die Gewichte w_{pq} so anzupassen, dass für eine angelegte Eingabe die erzeugte Ausgabe des Netzes möglichst nahe an der von uns erwarteten Ausgabe liegt. Die Abweichung E zwischen tatsächlicher Ausgabe y und erwarteter Ausgabe $\overline{y}$ ist der 2Gesamtfehler und ist definiert als

$$E = \frac{1}{2} \sum_q \left(y_q - \overline{y}_q\right)^2,$$

wobei q alle Ausgabe-Neuronen meint.

In einem zweiten Schritt gilt es nun E mittels Gradientenverfahren zu minimieren. Hierzu ist es nötig die partiellen Ableitungen von E mit Hinblick auf die einzelnen Gewichte w_{pq} zu berechnen. Es wird zunächst $\delta E/\delta y$ für alle Ausgabe-Neuronen p berechnet und damit anschließend $\delta E/\delta x$:

$$\frac{\delta E}{\delta y_p} = y_p - \overline{y}_p \qquad \text{und} \qquad \frac{\delta E}{\delta x_p} = \frac{\delta E}{\delta y_p} \left(y_p \left(1 - y_p \right) \right)$$

Damit kann der Einfluss einer Änderung in einer Gesamteingabe x für ein Ausgabe-Neuron auf den Gesamtfehler E bestimmt werden. Diese Gesamteingabe wiederum ist eine lineare Funktion von den Ausgaben der vorgelagerten Neuronen und den Gewichten der Verbindungen. Der Einfluss einer bestimmten Verbindung w_{pq} und der Einfluss einer Ausgabe des Neurons q auf ein Neuron p ergeben sich zu

$$\frac{\delta E}{\delta w_{pq}} = \frac{\delta E}{\delta x_p} y_q \qquad \text{und} \qquad \frac{\delta E}{\delta x_p} \frac{\delta x_p}{\delta y_q} = \frac{\delta E}{\delta x_p} w_{pq}$$

und für alle von q ausgehenden Verbindungen

$$\frac{\delta E}{\delta y_q} = \sum_p \frac{\delta E}{\delta x_p} w_{pq}.$$

Damit lassen sich alle $\delta E/\delta y$ der letzten versteckten Schicht berechnen, wenn alle $\delta E/\delta y$ der darüber liegenden Ausgabeschicht bekannt sind. Gleichzeitig können die $\delta E/\delta w$ berechnet werden. So kann man das Netz Ebene für Ebene herabsteigen. Die Anpassung der Gewichtungen kann dann anhand dieser Formel und einer zu wählenden Lernrate α vorgenommen werden:

$$\overline{w}_{pq} = w_{pq} - \alpha \left(\frac{\delta E}{\delta w_{pq}} \right)$$

Das Training eines TDNN weist eine Besonderheit auf. Um es zu trainieren, wird zunächst für jeden Zeitschritt innerhalb des Eingabefensters eine Kopie des Netzes erzeugt – bei einer Fenstergröße von $S = 3$ würden also 3 Kopien erzeugt. An jede Netz-Kopie wird dann die Eingabe des korrspondierenden Zeitschritts angelegt und für dieses das Backpropagation-Verfahren ausgeführt. Die durch die Ausführung des Backpropagation-Verfahren auf den (um beim Beispiel zu bleiben) drei Netzen bestimmten Gewichtsanpassungen werden nun gemittelt und erst dann angewendet. Das geschieht für jeden Trainingsdatensatz. (Rumelhart u. a., 1988; Svozil u. a., 1997)

Bewertung

TDNNs bieten den Vorteil, dass zeitliche Schwankungen in der Eingabe – je nach Größe der Schwankung – die Erkennung nicht stören. Außerdem erfolgt die Berechnung recht effizient, da immer nur ein relativ kleiner Teil der Eingabe (Fenstergröße) verarbeitet werden muss.

Die Festlegung der Fenstergröße und damit auch des Maßes der Unempfindlichkeit gegenüber zeitlichen Schwankungen in der Eingabe kann allerdings auch von Nachteil sein, da bei ausreichend großem Unterschied in der Dauer zweier ansonsten identischer Eingaben unterschiedliche Ergebnisse bestimmt werden. Dies kann am Beispiel der Gebärdensprache verdeutlicht werden: Hier gibt es Gesten, die von der Bewegung her identisch sind und deren Bedeutung sich letztendlich aus der Ausführungsdauer der Geste ergibt. Nehme man nun an, dass zwei entscheidende Charakteristika einer solchen Geste für eine der beiden Interpretation innerhalb *eines* Zeitfensters vorkommen, für die andere Interpretation – bei der selben Fenstergröße – *unmöglich innerhalb eines* Zeitfensters aufeinander folgen können. Diese Gesten werden (natürlich in Abhängigkeit der darüberliegenden Schichten) wahrscheinlich unterschiedlich klassifiziert werden. Würde man die Fenstergröße nun so weit erhöhen, dass die Charakteristika der Gesten meistens im selben Zeitfenster betrachtet werden, dann ließen sich die beiden Gesten nicht mehr unterscheiden. Diese Eigenschaft kann natürlich erwünscht sein, das ist sie im Rahmen der vorliegenden Arbeit allerdings ausdrücklich nicht. (Yang u. a., 2002)

Weitere Schwierigkeiten liegen in der Wahl geeigneter Parameter. Fenstergröße (vgl. oben), Anzahl der versteckten Schichten und je Schicht wiederum die Neuronenanzahl, sowie für das Training des Netzes die Lernrate sind Parameter, die sorgfältig mit Hinblick auf die zu lösende(n) Aufgaben ausgewählt werden müssen. Die Menge und Qualität der verfügbaren Trainingsdaten und die zu erlernende Erkennungsaufgabe selbst bestimmen, wie eine sinnvolle Struktur aussehen kann. Eine einfache Aufgabe kann für ein komplexes Netz übermäßiges Training erfordern, umgekehrt kann ein zu einfaches Netz nur einfache Aufgaben lösen, egal wie sehr es trainiert wird. Ein zu ausgeprägtes Training kann wiederum zu „overfitting" führen, so dass das Netz die Trainingsdaten gewissermaßen „auswendig" lernt und das erlernte kaum noch generalisieren kann. Wird die Lernrate zu groß gewählt, kann das Netz instabil werden und es stellt sich keine Konvergenz des Gesamtfehlers ein. Je kleiner die Lernrate, desto intensiver muss das Netz aber trainiert werden. All die zu wählenden Parameter lassen sich schwer abschätzen und müssen in aller Regel per „trial-and-error" bestimmt werden. (Lang u. a., 1990; Svozil u. a., 1997; Waibel u. a., 1989; Wang u. a., 2005)

Es wird deutlich, dass die Konfiguration eines TDNN zum Erlernen und Erkennen eines benutzerdefinierten Gestenrepertoires eine sehr anspruchsvolle Aufgabe ist und neben vielen Experimenten auch sehr große Mengen an Trainingsdaten erfordert (Dutzende bis Tausende Datensätze sind durchaus keine Seltenheit (Lang u. a., 1990; Wang u. a., 2005)). Weil das Gestenrepertoire benutzerdefiniert ist, ist es nicht möglich im Vorfeld eine möglichst passende Konfiguration vorzugeben. Einem Benutzer jedoch ist die Konfiguration eines TDNN kaum

zuzumuten. Das gleiche kann in ungünstigen Fällen auch vom Training selbst gesagt werden. TDNNs zeigen sich für diese Arbeit daher gänzlich ungeeignet.

2.2.4. Rubines Algorithmus

Der in Rubine (1991) vorgestellte Algorithmus, der dort selbst noch nicht explizit benannt wurde und sich erst im Laufe der Zeit als „Rubine's Algorithm" etabliert hat, ist ein vergleichsweise einfacher Algorithmus zur Erkennung von Gesten eines Merkmals. Damit ist gemeint, dass er in seiner ursprünglichen Variante für Gesten gedacht war, die z.B. durch eine Maus oder einen Eingabestift o.ä. vollzogen wurden, nicht aber Gesten, die sich aus der Verfolgung einer Vielzahl einzelner Merkmale ergeben.

Eigenschaften

Eine Geste wird repräsentiert durch einen Vektor g. Darin enthalten sind die Koordinaten der Maus zu den P unterschiedlichen Zeitpunkten: $g_p = (x_p, y_p, t_p)$, mit $0 \leq p \leq P$. Aus einer Menge von C Gesten wird dann diejenige bestimmt, die am Besten zur jeweils eingegebenen Geste passt. Dieser Vergleich geschieht in zwei Schritten: Zunächst wird aus der Eingabe ein sog. „Feature-Vector" $f = [f_1, \ldots, f_F]$ extrahiert. Dieser Feature-Vector wird anschließend mit dem Repertoire der zuvor antrainierten Feature-Vectors verglichen.

Die Merkmale des Feature-Vectors sollen eine Reihe Anforderungen erfüllen:

- Sie sollen inkrementell berechenbar sein, um auch große Gesten effizient behandeln zu können
- Kleine Änderungen an der Eingabe sollen nur kleine Änderungen an den Merkmalen nach sich ziehen
- Die Merkmale sollen eine sinnvolle Bedeutung haben (z.B. Abstand zwischen Anfang und Ende der Geste) um solch ein Merkmal auch als Programmeingabe nutzen zu können
- Die Wahl der Merkmale soll möglichst gering sein, ohne dabei aber die mögliche Vielfalt eines Gestenrepertoires einzuschränken

Verwendete Merkmale sind z.B. die Entfernung zwischen erstem und letztem Punkt, die Länge der Diagonalen der „Bounding Box"[2] und ihren Winkel zur horizontalen oder die Gesamtlänge des Pfades in g. Die konkrete Auswahl der Merkmale im Feature-Vector kann Rubine (1991) entnommen werden.

[2] kleinstes Viereck, das noch den vollständigen Pfad g enthält

Erkennung

Die Erkennung von Gesten, bzw. der Vergleich mit dem Gestenrepertoire ist denkbar einfach: Für den Eingabe-Feature-Vector wird der Vergleichswert v berechnet und mit den Vergleichswerten v_c verglichen. Es wird das c gesucht, für das

$$v_c = w_{c0} + \sum_{i=1}^{F} w_{ci} f_i \quad \text{mit} \quad 0 \leq c \leq C$$

maximal wird, wobei w_{ci} mit $0 < i \leq F$ die Gewichte der einzelnen Merkmale im Feature-Vector und w_{c0} die negative Hälfte der Summe der gewichteten Mittelwerte aller Merkmale einer Klasse c (vgl. Abschnitt „Training“, unten) darstellen.

Training

Gesten werden gelernt, indem sie einige Male wiederholt werden. Hierbei werden die Gewichte w_{ci} berechnet. Sei f_{cei} das i-te Merkmal der e-ten Wiederholung einer Geste c, wobei $0 \leq c \leq E_c$ (E_c ist die Gesamtzahl der Wiederholungen) gilt. Es werden als erstes alle Merkmale der Eingaben gemittelt:

$$\overline{f}_{ci} = \frac{1}{E_c} \sum_{e=0}^{E_c - 1} f_{cei}$$

Die Elemente Σ_{cij} einer Schätzung für eine Kovarianz-Matrix einer Geste c sind

$$\Sigma_{cij} = \sum_{e=0}^{E_c - 1} \left(f_{cei} - \overline{f}_{ci}\right) \left(f_{cej} - \overline{f}_{cj}\right)$$

und eine Schätzung für die allgemeine Kovarianz-Matrix dann

$$\Sigma_{ij} = \frac{\sum_{c=0}^{C-1} \Sigma_{cij}}{-C + \sum_{c=0}^{C-1} E_c}.$$

Diese Matrix wird dann invertiert ($\left(\Sigma^{-1}\right)_{ij}$) und man erhält die Gewichte

$$w_{cj} = \sum_{i=0}^{F} \left(\Sigma^{-1}\right)_{ij} \quad \text{mit } 1 \leq j \leq F \qquad \text{und}$$

$$w_{c0} = -\frac{1}{2} \sum_{i=1}^{F} w_{ci} \overline{f}_{ci}.$$

Einsatzgebiete und verwandte Algorithmen

Rubines Algorithmus findet hauptsächlich Anwendung oder Erwähnung im Bereich des sog. „Sketching“ oder „Sketch Recognition“[3] (z.B. Landay und Myers, 2001; Zeleznik u. a., 2007) und bei der Erkennung von einfachen Gesten unter Betrachtung eines einzigen Merkmals, wie sie zum Beispiel durch Maus, Eingabestift oder einem ähnlichen Eingabegerät möglich sind (vgl. z. B. Babu u. a.). Seit seiner Veröffentlichung bauten eine Reihe anderer Verfahren auf ihm auf oder die Menge der herangezogenen Merkmale wurde ergänzt (z.B. Signer u. a., 2006; Willems u. a., 2009).

Für den selben Einsatzzweck gibt es heute eine Vielzahl von Algorithmen mit unterschiedlichen Ansätzen. Große Popularität zeigen in den letzten Jahren die Algorithmen der $-Familie ($1, $N, Protractor, $N-Protractor, $P). Ihre Besonderheit liegt vor allem in ihrer einfachen Implementierung. Weitere Algorithmen sind z.B. die im iGesture-Framework (vgl. Abschnitt 2.3.3) implementierten SiGrid- und SiGeR-Algorithmen. Dabei bestimmt SiGrid den Verlauf einer Geste auf Grundlage einer besonderen Rastereinteilung des Eingabebereichs, die es erlaubt, die Bestimmung der Ähnlichkeit einer Eingabe zu einer Referenzgeste auf die Berechnung des Hamming- oder Levenshtein-Abstandes zwischen zwei Bitfolgen zu reduzieren. SiGeR hingegen basiert auf regulären Ausdrücken, mit denen die verschiedenen Bewegungsrichtungen eines Zeigeinstruments verglichen bzw. beschrieben werden können. Eine detailliertere Vorstellung dieser oder weiterer Algorithmen aus diesem Anwendungsbereich würde jedoch für diese Arbeit an dieser Stelle kaum zusätzlichen Nutzen bieten, es sei daher für den Moment auf die äußerst vielfältige Literatur zu diesen Themen verwiesen, insbesondere Signer u. a. (2006); Swigwart (2005); Anthony und Wobbrock (2010); Vatavu u. a. (2012); Willems u. a. (2009).

Bewertung

Jeder der unzähligen Algorithmen aus diesem Bereich hat seine Vor- und Nachteile. Dies wird sich z.B. in Babu u. a. zu Nutze gemacht, in dem dort gleich drei Verfahren gewissermaßen gleichzeitig arbeiten und die eigentliche Erkennungsaufgabe dann von dem Verfahren entschieden wird, das sich während des Trainings als für die in Frage stehende Gestenklasse am zuverlässigsten erwiesen hat.

Gesten, in welcher Form sie auch repräsentiert werden, müssen in der Regel vor ihrer Verarbeitung auf eine einheitliche Größe skaliert werden. Die Orientierung des die Geste

[3] Hierbei handelt es sich um Verfahren, die aus Freihandskizzen (bzw. aus den selben Bewegungen unter Ersetzung von Stift und Papier durch einen Digitizer) digitale Zeichnungen erstellen. Dabei werden verschiedene geometrische Formen erkannt und entsprechend in die digitale Zeichnung übernommen. So wird aus einem leicht deformierten Kreis z.B. ein geometrisch korrekter. Vgl hierzu z.B. Wu u. a. (2014).

ausführenden Benutzers kann zu einem weiteren Problem werden. (Anthony und Wobbrock, 2010; Babu u. a.)

Es ist außerdem notwendig, den Start und das Ende einer Geste zu kennen. Das in Rubine (1991) vorgestellte System z.B. bezieht sich auf Maus-Gesten, die bei gedrückter Maustaste erfolgen. Das Drücken/Loslassen der Maustaste ist hier eine gut geeignete Markierung. Das Bestimmen solcher Markierungen ist ohne Verwendung eines Eingabegerätes aber deutlich schwieriger und erfordert Pausen zwischen einzelnen Gesten oder ähnliche Maßnahmen.

Unklar ist außerdem, wie diese oft in der Ebene arbeitenden Verfahren in den Raum übertragen werden können. Die Projektion einer räumlichen Aufnahme in eine Ebene bringt eigene Probleme mit sich, wie z.B. wieder die Orientierung des Benutzers. Die paarweise parallele Betrachtung von je zwei Ebenen könnte eine Lösung zu Lasten der Systemkomplexität darstellen.

Der in (Vatavu u. a., 2012) vorgestellte $P-Algorithmus ließe sich relativ leicht auf drei Dimensionen erweitern. Mit diesem ließe sich u.U. eine vergleichsweise robuste „Template-Matching-Engine“ realisieren. Durch seinen Punktwolken-Ansatz wäre sein Einsatz im Vergleich zu z.B. $1 oder $N durchaus denkbar und auch der Autor der genannten Arbeit strebt dort einen Vergleich mit anderen Verfahren zur 3D-Gestenerkennung an. Die Tauglichkeit ist aktuell allerdings nicht geklärt.

Zusammenfassend muss festgestellt werden, dass die Tauglichkeit der hier genannten Verfahren vor allem bezüglich der Arbeit in der Ebene geklärt ist. Während es durchaus möglich sein könnte, auf dieser Grundlage Verfahren zur Arbeit im dreidimensionalen Raum zu entwickeln, so gibt es dazu noch viele offene Fragen zu klären und Probleme zu lösen. Da es bei dieser Arbeit weniger um eine Erprobung möglicherweise tauglicher Verfahren geht, sondern um die Entwicklung eines funktionierenden Werkzeuges, werden die hier vorgestellten Verfahren zu Gunsten erfolgversprechenderen bzw. erprobteren Ansätzen nicht weiter verfolgt.

2.2.5. Aufzeichnung/Erkennung auf Grundlage von Schlüssel-Posen

Miranda u. a. (2012) und Miranda u. a. (2014) beschreiben ein System zur Aufzeichnung und Erkennung von Gesten auf der Grundlage von ausgewählten (trainierbaren) Schlüsselposen anhand eines bereitgestellten Skelettmodells. Der Begriff „Pose“ bezeichnet hier und im Folgenden eine definierte Körperhaltung. Es wird eine praktische Repräsentation für Körperposen vorgestellt, ein Verfahren zur Klassifizierung solcher Posen und eines zur Erkennung von aus festgelegten Schlüsselposen bestehenden Gesten. Die Grundlagen dieser Arbeiten entstammen Raptis u. a. (2011). Miranda u. a. (2012) verbessert im Wesentlichen die Körperposenrepräsentation, während Miranda u. a. (2014) darüber hinaus die Lernfähigkeit bezüglich

der Schlüsselposen weiter verbessert. Da außerdem in den genannten Arbeiten, wie auch bei der hier vorliegenden Arbeit, auf einem Skelettmodell aufgesetzt wird bzw. werden soll, ist eine genauere Untersuchung der dort vorgestellten Lösungen in jedem Fall angemessen.

Repräsentation von Posen

Die in Miranda u. a. (2012) und Miranda u. a. (2014) vorgestellte Repräsentation ist rotations- und positionsinvariant bzgl. des Eingabeskeletts. Außerdem schlagen sich Unterschiede in Skeletten verschiedener Individuen kaum in ihrer Repräsentation nieder. Merkmale des Eingabeskeletts werden in drei Kategorien unterteilt: Torso-Merkmale, Merkmale erster Ordnung und Merkmale zweiter Ordnung. Torso-Merkmale sind z.B. Schultern oder Hüften. Merkmale erster Ordnung sind solche, die mit Torso-Merkmalen verbunden sind, z.B. Ellenbogen und Knie. Merkmale zweiter Ordnung sind solche, die mit Merkmalen erster Ordnung verbunden sind: Hände und Füße.

Verschiedene Posen eines Körpers (bzw. Eingabeskeletts) entstehen durch Rotation mit zwei Freiheitsgraden (Polwinkel θ und Azimuth ϕ) um einzelne Merkmale, während die Distanz zwischen zwei direkt verbundenen Merkmalen unverändert bleibt. Auf dieser Grundlage werden die drei Raumkoordinaten der Merkmale in sphärische Koordinaten umgerechnet.

Als Bezugspunkt für die Merkmale ersten Grades wird die Torso-Basis $\{u, r, t\}$ herangezogen, eine mittels Hauptkomponentenanalyse (Principal Component Analysis, PCA) aus den einzelnen Torso-Merkmal-Koordinaten bestimmte Orthonormalbasis. Hierbei wird u als Polachse und und r als Bezugsachse der Äquatorebene des zur Merkmalsrepräsentation verwendeten sphärischen Koordinatensystems interpretiert. Um beispielsweise die Position des Ellenbogens zu erhalten erfolgt eine Translation der Torso-Basis bzw. des Koordinatensystem-Urpsrungs zur benachbarten Schulter. Anschließend werden die Winkel θ (Polwinkel) und ϕ (Azimut), die die Position des Ellenbogens festlegen, bestimmt. Da die Knochenlänge konstant ist wird sie ignoriert.

Für die Bestimmung der Positionen von Merkmalen zweiter Ordnung findet eine weitere Translation und zusätzlich eine Rotation der Torso-Basis statt. Um z.B. die Position der an einen Ellenbogen angrenzenden Hand zu bestimmen wird folgendermaßen vorgegangen. Hierbei sei v der Vektor von Schulter zu Ellenbogen und w der Vektor vom Ellenbogen zur Hand. Zunächst wird die Torso-Basis um den Winkel $\beta = \widehat{(v, r)}$ um die Achse $b = v \times r$ rotiert. Die Bezugsachse r' des auf Grundlage der Torso-Basis erzeugten sphärischen Koordinatensystems wird also entlang des Vektors v ausgerichtet. Anschließend wird die entstandene Basis $\{u', r', t'\}$ zum Ellenbogen verschoben. So ergibt sich θ zum Winkel zwischen v und w. ϕ ist der Winkel zwsichen t' und der Projektion von w auf die Ebene orthogonal zu v.

Der Vorteil dieses Ansatzes gegenüber Raptis u. a. (2011) liegt darin, dass die Repräsentationen der Merkmale sich immer auf ein jeweils lokales Koordinatensystem beziehen. Messungenauigkeiten können sich daher nicht von Merkmal zu Merkmal fortpflanzen und die Repräsentation des Skelettmodells ist stabiler. Geht man davon aus, dass 9 Merkmale erster oder zweiter Ordnung aufgezeichnet werden, dann erhält man einen Posenbeschreibungsvektor $p = (\theta_1, \phi_1, \ldots, \theta_9, \phi_9) \in \mathbb{R}^{18}$.

Klassifizierung von Posen

Zur Klassifizierung von Posen kommen sog. Multi-Class Support Vector Machines (Multi-Class SVMs) zum Einsatz. Im Folgenden seien $\mathcal{K} = \left\{c_1, c_2, \ldots, c_{|\mathcal{K}|}\right\}$ die Menge der unterschiedlichen Klassen (also eine Menge von Bezeichnern für die einzelnen Schlüsselposen).

Für jede Schlüsselpose $i \in \{1, \ldots, |\mathcal{K}|\}$ wird eine Klassifizierungsfunktion

$$\hat{f}_i(p) = \sum_{j \in SV} \alpha_j \psi_i(c_j) \phi(p_j, p) + b$$

mit

$$\psi_i(c) = \begin{cases} 1 & \text{wenn } c = i \\ -1 & \text{sonst} \end{cases}$$

und sog. Kernelfunktion

$$\phi(p_1, p_2) = \exp\left(-\frac{||p_2 - p_1||^2}{2\sigma^2}\right)$$

definiert. SV ist die Menge der Indizes der für die Klassifizierung relevanten Trainingsvektoren (sog. „Stützvektoren“, mehr im nachfolgenden Abschnitt). σ wird in Miranda u. a. (2012) konstant auf 5 gesetzt, die einzelnen α_j und b sind Parameter, die beim einlernen der SVM berechnet werden. $\hat{f}_i$ ist positiv, wenn p zur Klasse i gehört, und sonst negativ. Je höher der Wert, desto sicherer ist die Aussage. Die Klassifizierung einer Pose p erhält man damit anhand der Funktion

$$\hat{f}(p) = \begin{cases} q = \arg\max_i \hat{f}_i(p) & \text{wenn } \hat{f}_q(p) > 0 \\ -1 & \text{sonst} \end{cases}.$$

Training der Multi-Class SVM

Das Training einer SVM zielt darauf ab, eine Hyperebene zu finden, die die Trainingsdaten optimal, d.h. mit dem weitest möglichen Abstand zwischen Hyperebene und Trainingsdaten, trennt. Eine detaillierte Beschreibung des SVM-Trainings läge weit außerhalb des Rahmens

dieser Arbeit, für Details sei daher auf die entsprechende Literatur, z.b. Cortes und Vapnik (1995), verwiesen. Es soll dennoch versucht werden, die grundsätzlichen Konzepte des Trainings kurz aufzugreifen.

Es sei $\mathcal{T} = \{(x_1, y_1), (x_2, y_2), \ldots\}$ eine Trainingsdatenmenge, die für jeden Datensatz x_i festlegt, ob er zur aktuell betrachteten Klasse gehört ($y_i = 1$), oder nicht ($y_i = -1$). Sofern sich die Datensätze nicht durch eine Hyperebene trennen lassen, werden sie solange in einen höherdimensionalen Raum überführt, bis dies möglich ist und Vektor w und Bias b gefunden werden können, so dass für jeden Trainingsdatensatz (x_i, y_i) gilt: $y_i(w \cdot x_i + b) \geq 1$. Vektoren, für die $y_i(w \cdot x_i + b) = 1$ gilt, liegen am dichtesten an der Hyperebene zwischen den Klassen und werden Stützvektoren (oder Support Vectors) genannt und bestimmen die Lage der optimalen Hyperebene. Die optimale Hyperebene w_0 ist diejenige, die zu beiden Klassen den maximalen Abstand hat und den Abstand $w \cdot w$ minimiert. Es gilt $w_0 = \sum_{i=1}^{l} y_i \alpha_i^0 x_i$ (l ist die Anzahl der Stützvektoren), zur Bestimmung der Parameter α_i^0 wird zunächst ein lineares Gleichungssystem der Form $y_i = x_i \sum_{j=0}^{l} x_j \alpha_j, i \in [0, l]$ gelöst. Für ausführlichere Beschreibungen sei auf die Literatur verwiesen.

Wie in Abschnitt 2.2.5 bereits beschrieben, setzt sich die Multi-Class SVM aus Miranda u. a. (2012) aus mehreren einfacheren SVMs zusammen, die eine Entscheidung bezüglich einer einzigen Schlüsselpose treffen können. Diese SVMs werden einzeln trainiert, und einzeln befragt. Besonderheiten für das Training dieser Sorte Multi-Class SVMs ergeben sich daher nicht. Eine Trainingsdatenmenge enthält also eine Reihe von Datensätzen, die je einer Schlüsselpose eine Klasse zuordnen. Die SVM, die entscheidet, ob ein Datensatz „zu ihr gehört", wird dann mit allen Posenvektoren trainiert, die in der Trainingsdatenmenge ihrer jeweiligen Klasse zugeordnet sind.

Repräsentation und Erkennung von Gesten

Eine Geste wird als Sequenz von Schlüsselposen repräsentiert. Anhand einer aus Schlüsselposensequenzen bestehenden Trainingsdatenmenge wird ein Wald konstruiert. Die Bäume dieses Waldes repräsentieren diese Gesten (Es kann höchstens so viele Bäume geben, wie es Gesten gibt). Die inneren Knoten der einzelnen Bäume sind dabei Schlüsselposen, die Blätter sind die Bezeichner der jeweiligen Gesten und die Wurzeln der Bäume sind die Schlüsselposen, mit denen eine Geste endet. Abbildung 2.11 illustiert die Überführung von Gesten in einen Entscheidungswald.

Die Körperposen des Eingabe-Datenstroms werden kontinuierlich mit der Schlüsselposenmenge verglichen. Wird eine Schlüsselpose erkannt, so wird ihr Vorkommen in einem

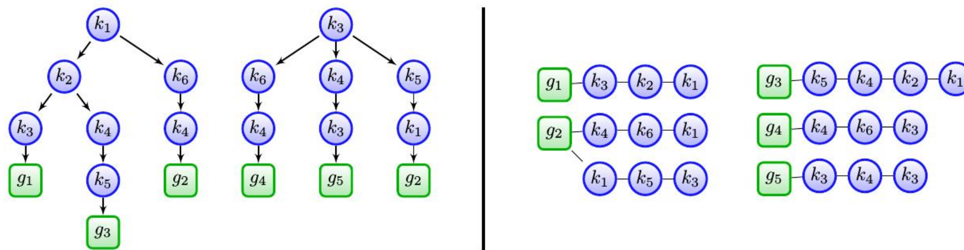


Abbildung 2.11.: Beispiel eines Entscheidungswaldes (links), der aus der Gestenmenge (rechts) hervorgeht.(Miranda u. a., 2012, ©2012 IEEE)

Ringpuffer vermerkt. Die Zahl der Elemente, die dieser Puffer halten kann, muss mindestens der maximalen Anzahl von Schlüsselposen entsprechen, aus denen sich die Gesten zusammensetzen. Wird die Schlüsselpose, die als letztes im Puffer vermerkt wurde, nochmals erkannt, so wird sie *nicht* erneut vermerkt.

Bei jedem Erkennen einer Schlüsselpose wird aus dem Wald zunächst derjenige Baum ausgewählt, dessen Wurzel die soeben erkannte Schlüsselpose darstellt. Es wird anschließend der Ringpuffer rückwärts durchlaufen und versucht, im ausgewählten Baum einen Pfad zu finden, der aus den Schlüsselgesten im Ringpuffer besteht. Endet solch eine Suche an einem Blatt, so wurde die dort hinterlegte Geste erkannt. Wurde gar kein passender Baum gefunden, endet die Suche früher oder führt in eine Sackgasse, so repräsentieren die bisher erkannten Schlüsselposen keine bekannte Geste.

Bewertung

Aufgrund des augenscheinlich recht großen Aufwandes, der bei dem ursprünglich in Miranda u. a. (2012) vorgestellten Verfahren vor allem für die Repräsentation von Körperposen und deren Klassifizierung getrieben wird, könnte der Eindruck entstehen, dass eine einfache Anwendung eines solchen Verfahrens durch unbedarfte Nutzer kaum möglich sein dürfte.

Während die Vorverarbeitung der Daten in der Tat deutlich komplizierter scheint, als bei den anderen in diesem Abschnitt vorgestellten Verfahren, verhält es sich mit dem Konfigurationsaufwand des Systems genau anders herum. Der Anwender eines solchen Systems benötigt keinerlei Hintergrundwissen um irgendwelche Konfigurationsparameter auszuwählen, die die Erkennungsleistung des Sytems beeinflussen.

Sobald die Aufgabe der Posen-Repräsentation erst einmal gelöst ist, ermöglicht der Aufbau des Entscheidungswaldes eine sehr effiziente Klassifizierung von Gesten. Die Leistung der SVM zur Posenerkennung wird in Miranda u. a. (2014) weiter verbessert, indem statt einer auf

euklidischen Metriken basierenden Kernelfunktion eine auf geodätischen Metriken basierende verwendet wird. Diese wird der Repräsentation einer Posen als eine Reihe von Winkeln besser gerecht.

Desweiteren ist es möglich, diesen Ansatz um zeitliche Einschränkungen bei der Ausführung von Gesten zu erweitern. Die Ausführung einer initialen Schlüsselpose oder irgendeiner anderweitigen Markierung des Beginns einer Geste ist nicht nötig, da sich bei dem beschriebenen Suchverfahren ein „Alignment“-Problem gar nicht stellt.

Insgesamt wird deutlich, dass dieser Ansatz dem Benutzer überhaupt kein Fachwissen abverlangt. Das Trainieren einer Schlüsselpose erfordert i.d.R. zwischen 10 (Miranda u. a., 2014) und 30 (Miranda u. a., 2012) Wiederholungen. Die Posenrepräsentation ist zwar komplizierter als bei anderen Verfahren, dieses „Problem “ ist jedoch nur ein einziges mal bei der Implementierung eines solchen Systems zu lösen. Der hier vorgstellte Ansatz zeigt sich unter diesen Gesichtspunkten als eine überaus vielversprechende Grundlage für die vorliegende Arbeit.

2.2.6. Zusammenfassende Beurteilung

Im vorangehenden Abschnitt wurden eine Reihe verschiedener Verfahren zur Gestenerkennung vorgestellt, genauer betrachtet und mit Hinblick auf die Verwendbarkeit als Grundlage für die hier vorliegende Arbeit bewertet.

Während die Eignung der Vielzahl von Algorithmen im Umfeld des Rubine-Algorithmus zur Erkennung von dreidimensionalen Gesten nicht ganz geklärt ist, so haben sie sich im zweidimensionalen durchaus bewährt. Von diesen Algorithmen wird im weiteren Verlauf dieser Arbeit hauptsächlich Abstand genommen, weil die Aussichten auf einen erfolgreichen Einsatz im dreidimensionalen sehr unklar sind.

Die übrigen vorgestellten Algorithmen sind prinzipiell für die Erkennung eines trainierbaren Gestenrepertoires im dreidimensionalen geeignet. Eine genauere Betrachtung jedoch zeigt, dass sie meist recht spezielle Anforderungen und/oder unerwünschte Nebeneffekte besitzen, oder auch in ihrer Konfiguration zu aufwendig sind.

Die Realisierung vollständiger zeitlicher Invarianz bei der Erkennung von Gesten hat sich beim Dynamic Time Warping (DTW) als Problem herausgestellt, da die Rechenkomplexität unter den angestrebten Bedingungen leicht sehr schlecht zu beherrschende Ausmaße annehmen könnte.

Hidden Markov Models (HMMs) und Time-Delay Neural Networks (TDNNs) bieten zeitliche Invarianz bei der Gestenerkennung ebenfalls nur innerhalb eines gewissen Rahmens. HMMs funktionieren am besten, wenn ihnen detailliertes domänenspezifisches Wissen zur Verfügung steht. Bei beiden Verfahren müssen für eine gute Trainings- und Wiedererkennungsleistung

von Gesten eine Reihe bestimmender Parameter korrekt gewählt werden. Die optimale Wahl der Parameter unterscheidet sich für jedes zu trainierende Gestenrepertoire, eine Vorauswahl von sinnvollen Parametern ist daher nicht möglich. Es wurde dadurch klar, dass HMMs und TDNNs für ihren erfolgreichen Einsatz ein Maß an Sachverstand erfordern, das von den Benutzern unseres Werzeuges nicht erwartet werden kann und soll.

Erfolgversprechend scheint letzten Endes ein Ansatz basierend auf der Modellierung von Gesten durch eine Reihe von Schlüsselposen. Diese Schlüsselposen werden im Wesentlichen durch die Winkel zwischen den an einem Gelenk angrenzenden Knochen bestimmt, was zu einer vergleichsweise robusten Repräsentation führt. Durch geschickte Modellierung des Gestenrepertoires kann eine sehr effiziente Wiedererkennung erfolgen. Aus Benutzersicht ist überdies keinerlei Konfiguration erforderlich. Aufgrund dieser Eigenschaften soll dieses Verfahren die Grundlage für das hier entwickelte Werkzeug bilden.

2.3. Vorhandene Werkzeuge

Zur Aufzeichnung und Erkennung existiert bereits eine große Anzahl von Werkzeugen. Darunter findet sich eine Vielfalt von zweidimensionalen Ansätzen (z.B. Signer u. a., 2006; Zeleznik u. a., 2007; Landay und Myers, 2001; Rubine, 1991), aber auch einige dreidimensionale (z.B. Hachaj und Ogiela, 2014; Franklin, a). Dieser Abschnitt soll in aller Kürze eine kleine Auswahl von Werkzeugen/Systemen vorstellen, die in diesem Bereich Beachtung verdient haben.

2.3.1. Gesture Description Language (GDL), GDL Studio

Bei der in Hachaj und Ogiela (2014) vorgestellten Gesture Description Language GDL handelt es sich, wie der Name schon sagt, um eine Beschreibungssprache für Gesten. Eine Beschreibung setzt sich aus Regeln zusammen, einzelne Regeln wiederum können miteinander kombiniert werden. Grundlage bilden immer die räumlichen Koordinaten von Merkmalen, die im genannten Papier aus dem von einer Microsoft Kinect erzeugten Skelettmodell entnommen werden. Funktionen vereinfachen es Winkel oder Distanzen zu messen.

Die so entstehenden Beschreibungen sind prinzipiell von Menschen gut lesbar, jedoch Bedarf es sicherlich ein wenig Übung sie schnell interpretieren zu können. Das Werkzeug „GDL Studio“ steht unter Hachaj und Ogiela zum kostenlosen Download bereit. Es bietet u.a. die Aufnahme und Wiedergabe von Skelettaufnahmen, Echtzeit-Gestenerkennung (online oder offline) und einen Editor für GDL-Skripte. GDL Studio funktioniert in der aktuellen Version nur mit dem Microsoft Kinect SDK.

2.3.2. GesturePak

Bei GesturePak (Franklin, a) von Carl Franklin handelt sich sich um kommerzielle Software. Sie funktioniert ebenfalls nur mit dem Microsoft Kinect SDK und besteht aus zwei Teilen: Dem GesturePak Recorder, mit dem ein Gestenrepertoire aufgezeichnet werden kann und einer .NET-Assembly, die die Wiedererkennung „ohne echte Programmierung" („The GesturePak API lets a .NET developer recognize those gestures with no real programming." Franklin (c)) ermöglicht. Die momentan noch aktuelle Version 1.0 funktioniert zwar mit dem aktuellsten Microsoft Kinect for Windows SDK, jedoch nur im sog. stehenden Modus. GesturePak benötigt außerdem mindestens eine Kinect for Windows, die Kinect für die XBox 360 wird nicht unterstützt.

Der Recorder wird zu einem großen Teil per Spracheingabe gesteuert. Die Aufnahme und Erkennung von Gesten basiert auch hier auf einzelnen Schlüsselposen. Es können allerdings die zu beachtenden Achsen und Gelenke festgelegt werden. Der Recorder ist im allgemeinen *nicht* Teil der auf GesturePak basierenden Software. Endnutzer eines damit entwickelten Programmes können also keine eigenen Gesten definieren.

Eine Alpha-Version des Nachfolgers GesturePak 2.0 ist auf Anfrage beim Entwickler erhältlich. Neben der Unterstützung für den neuen Kinect for Windows 2 Sensor bietet es auch eine angepasste Benutzersteuerung, die das nachträgliche Bearbeiten von Gesten sitzend am Rechner mit gewohnten Eingabegeräten ermöglicht. Des Weiteren sollen auch Endnutzer von mit GesturePak implementierten Programmen (optional) in der Lage sein, eigene Gesten hinzuzufügen. Es ist außerdem geplant der Software den Quelltext (oder zumindest Teile davon) beizulegen. Unter welcher Lizenz das geschehen soll, ist jedoch nicht bekannt, eben so wenig wie der Preis für das Programmpaket. (Franklin, b)

2.3.3. iGesture

IGesture wurde erstmals in Signer u. a. (2006) vorgestellt. Es handelt sich hierbei um ein in Java implementiertes Framework, mit dem sich verschiedene Algorithmen zur Gestenerkennung einsetzen und erproben lassen. Die „iGesture Workbench" ermöglicht Anwendern das Anlegen und die Verwaltung von Gestenrepertoires und stellt bereits vorgefertigte Gestenmengen zur Verfügung. Neben unterschiedlichen Erkennungsalgorithmen werden auch verschiedene Eingabegeräte wie Maus und Digitizer unterstützt.

IGesture ist durch weitere Algorithmen und Eingabegerät-Treiber erweiterbar. Ziel des Projektes ist es, die Entwicklung von gestengesteuerten Benutzerschnittstellen, neuen Algorithmen zur Gestenerkennung und Gestenmengen im Allgemeinen zu vereinfachen. Dazu wird eine Umgebung bereitgestellt, die Test und Einsatz vereinheitlicht, so dass sich Entwickler auf

ihre eigentlichen Ziele konzentrieren können. Das Framework selbst beschränkt sich nicht auf zweidimensionale Gesten. Aktuell ist jedoch nur ein einziger 3D-fähiger Algorithmus implementiert (eine Abwandlung des klassischen Rubine-Algorithmus, die im Wesentlichen den klassischen Rubine-Algorithmus für die verschiedenen Ebenen (xy, xz, yz) ausführt).

2.3.4. Zusammenfassende Betrachtung vorhandener Werkzeuge

Neben den oben kurz vorgestellten Werkzeugen gibt es noch weitere, vor allem im Bereich der 2D-Gestenerkennung bzw. des sog. „Sketchings". GDL (bzw. GDL Studio) ist ein kostenloses System, die Bearbeitung von Gesten hiermit ist allerdings etwas anspruchsvoller als bei GesturePak. Vor allem muss der Benutzer bereit sein, sich auf die regelbasierte Definition von Gesten einzulassen. GesturePak-Benutzer erhalten in erster Linie visuelle Eindrücke ihrer Gesten, was die Hürde für den Einsatz eines solchen Werkzeuges sicherlich niedriger setzt. Version 2 von GesturePak arbeitet mit dem Kinect for Windows 2 Sensor zusammen und bietet darüber hinaus weitere Verbesserungen. Allgemein verfügbar ist diese aktualisierte Version jedoch noch nicht.

2.4. Zusammenfassung verwandter Arbeiten

In diesem Kapitel wurden die grundlegenden Problemstellungen bei der Erkennung von Körperposen und -gesten betrachtet und Lösungsansätze vorgestellt und evaluiert. Insbesondere wurden Möglichkeiten zur Akquise der zur Modellbildung benötigten Daten aufgezeigt und beurteilt, wie auch Verfahren, die dann auf Grundlage der so gewonnenen Daten das Erlernen und Wiedererkennen von Körperposen und -gesten ermöglichen. Ergänzend wurden kurz einige Werkzeuge vorgestellt, die solche Verfahren bereits implementieren und Anwendern Möglichkeiten bereitstellen, gestengesteuerte Software zu entwickeln.

Es wurde verdeutlicht, dass für die vorliegende Arbeit optische markierungslose merkmalbasierte Verfahren zur Datenakquise und Modellbildung bei weitem am besten geeignet sind. Der Grund hierzu liegt in der Verfügbarkeit günstiger entsprechender Hardware und der Tatsache, dass Benutzer eines solchen Geräts hierzu nicht noch in irgendeiner Form präpariert werden müssen, wie es bei anderen Lösungen der Fall ist. Ein prominenter Vertreter dieser Geräte-Kategorie ist der Kinect Sensor von Microsoft, der u.a. Skelett-Modelle seiner Benutzer erzeugt und zur weiteren Verarbeitung zur Verfügung stellt.

Es wurden Verfahren gezeigt und bewertet, die auf der Grundlage eines solchen Skelettmodells in der Lage sind, das Problem der Aufzeichnung und Wiedererkennung von Körperposen und -gesten zu lösen. Es zeigte sich jedoch, das längst nicht alle Verfahren, die diese Probleme

grundsätzlich lösen, auch für diese Arbeit geeignet sind. Als am erfolgversprechendsten hat sich letzten Endes ein Ansatz erwiesen, der Gesten auf einzelne Schlüsselposen reduziert und die Trainings- und Erkennungsaufgaben auf Grundlage dessen löst.

3. Anforderungsanalyse

Das Werkzeug, das im Rahmen dieser Arbeit entwickelt wird, steht an der Schnittstelle zwischen verschiedenen Domänen. Auf einer Seite stehen die Anwender, die mit von ihnen eingenommenen Körperposen oder mit von ihnen ausgeführten Körpergesten unkompliziert und zuverlässig irgendein Programm oder Objekt steuern oder beeinflussen möchten. Auf einer anderen Seite steht eine Idee, bzw. deren Urheber. Sie haben eine mehr oder wenig genaue Vorstellung einer Anwendung oder eines Bedienkonzeptes und suchen eine Möglichkeit, diese Vorstellung so unkompliziert und schnell wie möglich zu erproben und/oder umzusetzen. Auf noch einer Seite, deren Grenzen zu ihren Nachbarn eher diffus sind, finden sich die Entwickler. Ihre Aufgabe ist es, diese Vorstellungen bestmöglich zu verwirklichen und dabei nicht nur die eigenen im Rahmen der Softwareentwicklung auftretenden Anforderungen, sondern auch die des zu Grunde liegenden Konzepts und der Endanwender zu berücksichtigen.

Zweck der im Rahmen dieser Arbeit entwickelten Software ist es diese Aufgabe zu erleichtern, indem geeignete Werkzeuge bereitgestellt werden, die den Anforderungen dieser drei Seiten genügen. In Abschnitt 3.1 dieses Kapitels werden diese Anforderungen zunächst aus Anwender-Sicht, in Abschnitt 3.2 dann aus Entwickler-Sicht detaillierter betrachtet um festzustellen, was genau zu deren Erfüllung geleistet werden muss.

3.1. Anforderungen aus Anwender-Sicht

Welche Gruppe man als „Anwender“ bezeichnen möchte hängt genau genommen von der Perspektive ab: Gemeint sein können einerseits eben diejenigen, die gerne irgendetwas nur durch Einsatz von Körpergesten steuern möchten. Andererseits könnte man auch diejenigen als Anwender betrachten, die mit Hilfe der Software eine Benutzerschnittstelle auf Grundlage von Körpergesten entwickeln möchten, ohne dabei jedoch über irgendwelche Kenntnisse im Bereich der Gestenerkennung, ja vielleicht noch nicht einmal der Programmierung, zu verfügen.

Eine zweckmäßige Unterscheidung lässt sich anhand des jeweiligen Anwendungsfalls treffen: Man betrachtet einerseits die ausschließliche Erkennungsaufgabe und andererseits die Trainings- und Testaufgabe.

3.1.1. Erkennung von Posen und Gesten

An erster Stelle soll die Einfachheit des Systems stehen: Damit ist gemeint, dass die Erkennung von Körperposen und -gesten keinerlei Vorbereitung erfordern darf. Insbesondere darf der Benutzer mit keinerlei Markierungen versehen werden und keine speziellen Geräte bedienen müssen. Eine kurze Einweisung bzgl. der vom System verarbeitbaren Posen und Gesten wäre denkbar, soll aber nicht notwendig sein.

Ein weiteres wichtiges Kriterium ist die Erkennungsleistung. Ein konkretes Maß soll an dieser Stelle nicht vorgegeben werden, da sie sich prinzipiell immer durch leistungsfähigere Hardware oder verbesserte Erkennungsverfahren steigern lässt, ohne dass der Anwender dies (außer durch die Steigerung der Erkennungsleistung) bemerken würde. Mindestens jedoch sollte die Erkennungsleistung auf einem Niveau liegen, dass die Anwender nicht durch immer wieder falsche Ergebnisse entmutigt. Nach Möglichkeit sollte die Erkennungsleistung so wenig wie möglich von Umweltbedingungen wie Sonneneinstrahlung, Luftfeuchtigkeit usw. abhängen, wobei die Stärke dieser Einflüsse im wesentlichen durch die Auswahl der verwendeten Hardware festgelegt wird.

Ein Mehrbenutzerbetrieb soll prinzipiell möglich sein. Hierbei sollten alle vom Sensor erkannten Benutzer gleich behandelt werden, wer also eine Geste ausführt spielt keine Rolle (was jedoch nicht bedeuten muss, dass diese Information nicht verwendet werden kann – vgl. Abschnitt 3.2).

3.1.2. Eintrainieren und Testen von Posen und Gesten

Für das Eintrainieren und Testen von Posen und Gesten gelten grundsätzlich zunächst die gleichen Anforderungen wie für die Erkennung, jedoch mit einigen Ergänzungen. Gesten sollen auf eine Abfolge von Schlüsselposen zurückgeführt werden. Es ist daher notwendig, dass ein vielseitiges Schlüsselposen-Repertoire aufgebaut, eingesehen, verwaltet und als Ganzes getestet werden kann.

Um eine effektive Verwaltung und Analyse eines Schlüsselposen-Repertoires zu ermöglichen benötigt der Anwender möglichst viele Informationen, da jeder einzelne Trainingsdatensatz sich auf die Erkennungsleistung *aller* Schlüsselposen auswirkt. Es muss während des Testens und insebsondere während des Trainings immer das aktuell für den Benutzer erzeugte Skelettmodell sichtbar sein. Außerdem muss detailliert und in Echtzeit erkennbar sein, wie die aktuell betrachtete (bzw. vom Tester eingenommene Pose) im Kontext des vollständigen Repertoires beurteilt wird. Pro Schlüsselpose müssen Statistiken auf Grundlage der zugehörigen Trainingsdaten angefertigt werden, die es ermöglichen, schlecht trainierte Schlüsselposen oder

„Ausreißer“ in den Trainingsdaten zu identifizieren. Eine Gezielte Analyse und Problemsuche in einem u.U. großen Repertoire ist ohne derartige Unterstützung gänzlich undenkbar.

Zur Aufnahme einer Geste soll diese ein Mal ausgeführt werden um festzustellen, aus welchen Schlüsselposen sie sich zusammensetzt. In diesem Zuge können weitere Posen aus der Aufnahme einer Geste zur Menge der Schlüsselposen hinzugefügt werden, sofern Bedarf besteht. Damit soll der Arbeitsfluss so wenig wie möglich unterbrochen werden. Sofort nach der Beschreibung einer Geste anhand ihrer Schlüsselposen soll sie getestet werden können. Um die Arbeit mit einer erstelletn Aufnahme zu erleichtern sind zumindest rudimentäre Bearbeitungs- und erweiterte Navigationsmöglichkeiten wünschenswert (Zuschnitt von Aufnahmebeginn und -ende, Springen an Erkennungspunkte einer Schlüsselpose, …).

Ein so aus einzelnen Gesten aufgebautes Gestenrepertoire muss ebenfalls eingesehen, verwaltet und als ganzes getestet werden können. Der Test eines Repertoires soll die Bewegungen des Testers aufzeichnen und dabei erkannte Schlüsselposen oder Gesten melden.

Ein solches Schlüsselposen- und Gestenrepertoire soll zur Verwendung in einer Anwendung in eine Datei exportiert werden können. Diese Datei kann ein für die jeweilige Anwendung bedarfsgerecht zusammengestelltes Repertoire enthalten. Das durch die Anwendung unterstützte Repertoire soll also nur durch solch eine Datei festgelegt werden. Um das unterstützte Repertoire zu ändern, wird lediglich die Datei durch eine andere ausgetauscht, die das gewünschte Repertoire enthält. Die Software muss also grundsätzlich mit verschiedenen Schlüsselposen- und Gestenrepertoires arbeiten können.

Eine grafische Benutzeroberfläche ist aus Gründen er Benutzer-Akzeptanz sowie zur präzisen Darstellung von Skelettmodellen unerlässlich.

3.1.3. Zusammenfassung der Anforderungen aus Anwender-Sicht

Es steht an erster Stelle immer die Einfachheit der Bedienung um eine größtmögliche Akzeptanz zu erreichen. Auch die Erkennungsleistung wird diesem Ziel untergeordnet, sollte dabei jedoch noch in einem akzeptablen Rahmen bleiben.

Das Eintrainieren eines Gestenrepertoires soll per grafischer Benutzeroberfläche erfolgen und der trainierenden Person und Testern permanent zweckmäßige Rückmeldung über erzeugtes Skelettmodell und erkannte Posen/Gesten geben.

3.2. Anforderungen aus Entwickler-Sicht

Entwickler sollen die Funktionen des Werkzeuges in Form einer Programmbibliothek („Library“) nutzen können. Wie wahrscheinlich bei jeder anderen Programmbibliothek auch ist hier in

erster Linie großer Zusammenhalt und geringe Kopplung wünschenswert: Sie soll in sich vollständig sein und über eine möglichst schmale Schnittstelle genutzt werden können.

Das Hauptaugenmerk soll hier auf der Wiedererkennung von Posen und Gesten liegen. Das Repertoire wird in einer Datei bereitgestellt. Der Entwickler soll für jede Pose oder Geste Beobachter registrieren können, die bei Erkennung einer solchen benachrichtigt werden. Damit die Programmbibliothek diese Aufgabe erfüllen kann, soll der Entwickler ihr lediglich kontinuierlich die die vom Sensor angefertigten Skelettmodelle beschreibenden Daten bereitstellen. Es soll dabei feststellbar bleiben, welcher Benutzer eine Pose/Geste eingenommen/ausgeführt hat.

Die Flexibilität des Entwicklers sollte nicht eingeschränkt werden. Je nach Anforderungen des jeweiligen Projektes soll er die von der Bibliothek benötigten Skelettdaten selbst bereitstellen können (um die Freiheit zu haben mit dem Sensor oder den Sensordaten noch weitere Dinge zu tun), oder alternativ mit möglichst wenig Aufwand und ohne Kenntnis der Sensor-API einen entsprechenden Datenstrom erhalten können, um ihn direkt an die Bibliothek weiterzugeben. Die Auswahl der Programmiersprache zur Implementierung seiner Anwendung soll ihm optimaler Weise ebenfalls freigestellt sein.

3.3. Zusammenfassung der Anforderungen

Die wesentlichen in diesem Kapitel beschriebenen Anforderungen lassen sich wie folgt Zusammenfassen:

- Aus Benutzersicht:
  - Einfachheit der Bedienung: Keine Präparation der Benutzer erforderlich
  - Mindestens ausreichende Erkennungsleistung, so dass Benutzer nicht demotiviert wird
  - Grafische Benutzeroberfläche zum Eintrainieren eines Schlüsselposen- und Gestenrepertoires
  - Detaillierte Analyse-Funktionen zur Problemsuche innerhalb eines Schlüsselposen- und Gestenrepertoires
- Aus Entwicklersicht:
  - Programmbibliothek mit großem Zusammenhalt und schmaler Schnittstelle
  - Schlüsselposen- und Gesterepertoire per einzelner Datei ladbar
  - Möglichkeit des Einsatzes eines Sensors ohne Kenntnisse über dessen API

  - Möglichkeit des Einsatzes der Programmbibliothek unabhängig von der Herkunft der Skelettrepräsentation

Weitere Anforderungen sollen für die erste Version der Software zunächst nicht gestellt werden.

4. Systementwurf

Teil dieser Arbeit sind zwei wesentliche Softwarekomponenten: Eine zur Aufnahme von Körperposen und -gesten, sowie eine zur Wiederekennung des Aufgenommenen. Dieses Kapitel beschreibt den Entwurf dieser beiden Softwarekomponenten sowie ihr Zusammenspiel.

In Abschnitt 4.1 wird gezeigt, wie sich diese Softwarekomponenten in funktionaler Hinsicht in das Gesamtsystem eingliedern. Im Anschluss daran wird in den Abschnitten 4.2 und 4.3 der zur Erfüllung der Anforderungen aus Kapitel 3 benötigte Funktionsumfang dieser Softwarekomponenten konkretisiert. Abschließend wird in Abschnitt 4.4 ein struktureller Entwurf der einzelnen Softwarekomponenten vorgestellt, der die Grundlage der Realisierung bildet.

4.1. Workflow

Abbildung 4.1 zeigt die beiden wesentlichen Softwarekomponenten (Aufnahme-Komponente und Erkennungs-Komponente) und veranschaulicht den Workflow innerhalb des Gesamtsystems.

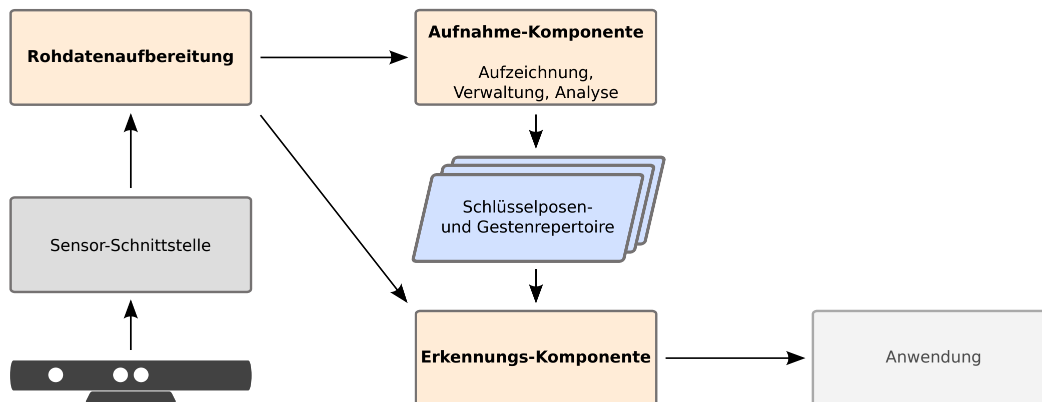


Abbildung 4.1.: Veranschaulichung des Workflows. Pfeile symbolisieren den Datenfluss.

Durch eine geeignete Sensor-Schnittstelle werden die vom System zu verarbeitenden Skelett-Rohdaten vom Sensor ausgelesen. Der Schritt der Rohdatenaufbereitung stellt sicher, dass

Aufnahme-Komponente und Erkennungs-Komponente die Daten in einem für sie weiterverarbeitbaren Format erhalten.

Eine Wissensbasis für diese beiden Komponenten bildet ein Schlüsselposen- und Gestenrepertoire. Diese Wissensbasis ist austauschbar. Mit Hilfe der Aufnahme-Komponente kann solch eine Wissensbasis angelegt, ergänzt, verwaltet und analysiert werden. Der Erkennungs-Komponente wird die zu verwendende Wissensbasis bereitgestellt. Auf ihrer Grundlage können eingehende Skelett-Daten dann klassifiziert werden. Anwendungen können die Ergebnisse dieser Klassifizierung dann verarbeiten und so auf Körperposen und -gesten ihres Benutzers reagieren.

4.2. Erkennungs-Komponente

Die Aufgabe der Erkennungs-Komponente liegt – wie der Name schon sagt – hauptsächlich in der Erkennung von Schlüsselposen und Gesten. Um die Entwicklung möglichst vielseitiger Anwendungen – wie z.B. der Aufnahme-Komponente selbst – um diese Erkennungs-Komponente herum zu erlauben, bietet sie jedoch noch viele weitere Funktionen:

- Wiedererkennung von Schlüsselposen und Gesten auf Grundlage einer austauschbaren Wissensbasis
- Bereitstellung von
  - aufbereiteten Sensor-Rohdaten
  - präzisen Wahrscheinlichkeitsverteilungen über alle bekannten Schlüsselposen
  - der Folge bisher (pro Benutzer) erkannten Schlüsselposen
  - Informationen zu absoluter Benutzerposition und -orientierung und zur Umrechnung von absoluten Sensor-Koordinaten in Benutzerspezifische lokale Koordinaten
- Benachrichtigung der Anwendung bei Eintritt verschiedener wichtiger Ereignisse:
  - Erkennung einer Schlüsselpose
  - Erkennung einer Geste
  - Abschluss des Eintrainierens einer Menge von Schlüsselposen
  - Abschluss des Eintrainierens einer Menge von Gesten
  - Erfolgreiche Verarbeitung eines Skelett-Datensatzes (Teil eines Eingabe-Datensatzes)
  - Erfolgreiche Verarbeitung eines vollständigen Eingabe-Datensatzes

Die Erkennungs-Komponente mit diesen Funktionen wird in Form einer Programmbibliothek bereitgestellt, ihre Realisierung ist in Abschnitt 5.3 detailliert beschrieben.

4.3. Aufnahme-Komponente

Die Aufgabe der Aufnahme-Komponente ist es, eine möglichst leicht zu bedienende und gleichzeitig flexible Umgebung bereitzustellen, mit der im besten Fall auch weniger computeraffine Benutzer ein Schlüsselposen- und Gestenrepertoire zur Verwendung durch die Erkennungs-Komponente zusammenstellen können. Um dies zu leisten und zu unterstützen bietet sie folgende Funktionen:

- Aufnahme von Skelett-Daten
  - Echtzeit-Überblick über alle für die Klassifizierung einer Eingabe relevanten Daten
  - Vereinfachte Navigation innerhalb einer Aufnahme
  - Einfache Bearbeitungsfunktionen
- Analyse des Schlüsselposen- und Gestenrepertoires
  - Betrachtung aller Trainingsdaten zu den Schlüsselposen
  - Anzeige von merkmalsspezifischem Mindest-, Maximal- und Mittelwert über alle Trainingsdaten zu einer Geste, sowie Bestimmung der Standardabweichung
  - Einfache Visualisierung der zu einer Geste gehörenden Schlüsselposen und umgekehrt
- Verwaltung des Schlüsselposen- und Gestenrepertoires
  - Erstellen von Schlüsselposen
  - Hinzufügen von Trainingsbeispielen für Schlüsselposen
  - Speichern einer Schlüsselposen-Folge als Geste
  - Löschen von Trainingsbeispielen, Schlüsselposen und Gesten
  - Export von Teilen eines Gestenrepertoires
- Test-Ansicht
  - Auswahl der zu testenden Wissensbasis
  - Parallele Betrachtung mehrerer Nutzer gleichzeitig

Diese Funktionen werden in einem Werkzeug mit grafischer Benutzeroberfläche vereint. Die Realisierung dieser Komponente wird in Abschnitt 5.4 genauer betrachtet.

4.4. Struktureller Entwurf

Abbildung 4.1 zeigte eine grobe Veranschaulichung der funktionalen Komponenten des Systems. Diese Sichtweise eignet sich gut für den Einstieg. Um sich der tatsächlichen Realisierung des Systems anzunähern ist es allerdings sinnvoll, einen strukturellen Entwurf genauer zu betrachten.

Solch einen Entwurf zeigt Abbildung 4.2. Gleichfarbige Kästen bilden eine Softwarekomponente. Einzelne Kästen können als Module betrachtet werden. Dabei ist die Bezeichnung des Moduls **fett** gesetzt und der jeweils wesentliche Klassenname *kursiv*.

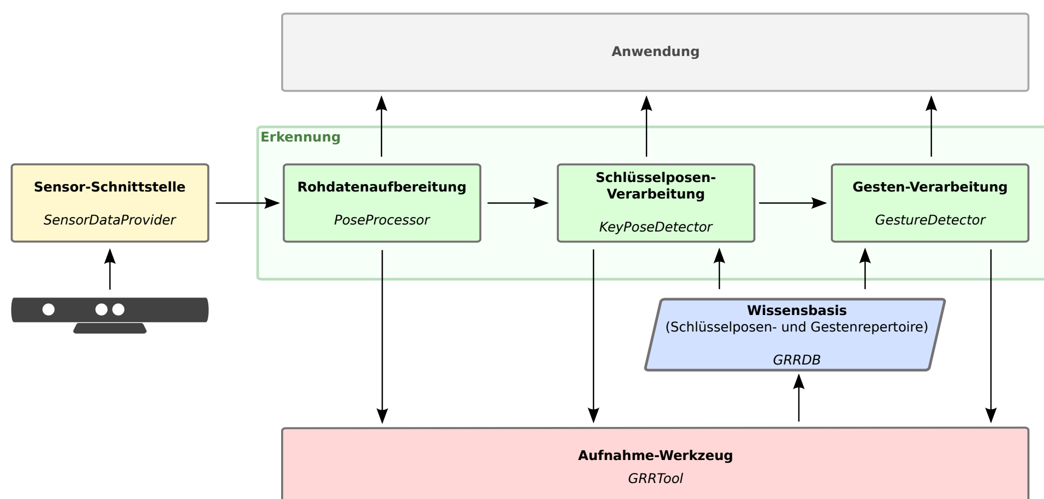


Abbildung 4.2.: Strukturell orientierter Systementwurf. Pfeile symbolisieren den typischen Datenfluss.

Die folgenden Teilabschnitte erklären Rolle und grundsätzliche Struktur der einzelnen Softwarekomponenten, sowie ihre strukturelle Beziehung untereinander genauer.

4.4.1. Erkennungs-Bibliothek

Die Erkennungs-Programmbibliothek besteht aus der Rohdatenaufbereitung, Schlüsselposen-Verarbeitung und Gesten-Verarbeitung und bildet den eigentlichen Systemkern. Aus Eingabe-Sicht stellt die Rohdatenaufbereitung die Schnittstelle nach außen dar. Sensor-Daten werden an diese übergeben und von dort aus innerhalb der Bibliothek an die Schlüsselposen-Verarbeitung weitergereicht.

Die Schlüsselposen-Verarbeitung klassifiziert die Rohdaten. Auf Grundlage der in der bereitgestellten Wissensbasis vorhandenen Informationen kann sie feststellen, ob eine Eingabe

einer bekannten Schlüsselpose entspricht oder nicht. Diese Information wird innerhalb der Bibliothek weitergegeben an die Gesten-Verarbeitung.

Die Gesten-Verarbeitung verfolgt welche Schlüsselposen (für welchen Systembenutzer) im Laufe der Zeit erkannt wurden. Mit diesen Informationen und der bereitgestellten Wissensbasis kann sie feststellen, wenn eine Geste ausgeführt wurde.

Die Informationen, die innerhalb der Erkennungs-Bibliothek weitergegeben werden (und noch weitere, vgl. Abschnitt 4.2) können auch außerhalb von ihr selbst abgegriffen werden. Dadurch ist es Anwendungen möglich relativ „feinkörnig“ mit dem System zu interagieren.

4.4.2. Aufnahme-Werkzeug und andere Anwendungen

Beim Aufnahme-Werkzeug handelt es sich im Grunde um eine ganz normale Anwendung, die die Erkennungs-Bibliothek verwendet. Damit ist gemeint, dass alles, was das Aufnahme-Werkzeug leistet in der Erkennungs-Bibliothek überhaupt erst ermöglicht wird. Es arbeitet ausgesprochen eng mit dieser Bibliothek zusammen – deutlich enger, als es eine „übliche“ auf der Bibliothek aufsetzende Anwendung üblicherweise täte. Die dort bereitgestellten Funktionen und Informationen werden lediglich geschickt kombiniert, zu einander in Verbindung gesetzt und auf für die jeweilige Aufgabe zweckmäßige Art eingesetzt und präsentiert.

Die Sonderrolle des Aufnahme-Werkzeugs liegt darin begründet, dass es speziell für die Zusammenstellung von Wissensbasen gedacht ist, die dann wiederum mit der Erkennungs-Bibliothek eingesetzt werden können. Dazu werden aber eben nur Funktionen des Systemkerns – der Erkennungs-Bibliothek – genutzt. Und zwar so, wie es jede andere auf der Erkennungs-Bibliothek aufsetzende Anwendung eben auch könnte. Alle verwendeten Informationen werden ausschließlich von der Erkennungs-Bibliothek bereitgestellt. Eine direkte Interaktion mit der Sensor-Schnittstelle oder gar dem Sensor selbst findet nicht statt.

4.4.3. Sensor-Schnittstelle

Aufgabe der Sensor-Schnittstelle ist es die Daten vom verwendeten Sensor entgegenzunehmen und dem Erkennungs-Werkzeug bereitzustellen. Eine solche Schnittstelle liegt dem System in Form eines Programmes namens „SensorDataProvider“ bei, dieses muss aber nicht verwendet werden. Wie die Daten zur Rohdatenaufbereitung gelangen, spielt keine Rolle. Wird der Sensor z.B. in einer Anwendung ohnehin verwendet, so können die Daten durch die Anwendung direkt in das Erkennungs-Werkzeug hereingegeben werden.

4.5. Zusammenfassung

Im Rahmen dieses Kapitels wurde ein Entwurf für das hier entwickelte System aus verschiedenen Perspektiven vorgestellt. Der funktionale Entwurf bzw. Workflow hat hierbei die grundsätzlichen Aufgaben des Systems und ihr Zusammenwirken verdeutlicht. Darauf aufbauend wurde der zur Erfüllung der Anforderungen aus Kapitel 3 vorgesehene Funktionsumfang der Erkennungs-Bibliothek und des Aufnahme-Werkzeugs spezifiziert und ein detaillierter struktureller Entwurf vorgstellt. Dieser bildet die Grundlage der Realisierung.

Weiterhin wurde verdeutlicht, dass die Erkennungs-Bibliothek den Kern des gesamten Systems bildet. Auch das Aufnahme-Werkzeug verwendet lediglich (wenn auch sehr intensiv und tiefgehend) die von der Erkennungs-Bibliothek bereitgestellten Funktionen und Informationen.

5. Realisierung

Nachdem die grundsätzliche Systemstruktur nun erklärt ist kann die Realisierung des Entwurfs betrachtet werden. Welchem Teil dabei die größere Bedeutung beigemessen werden soll, hängt vom Standpunkt des Betrachters ab – vgl. hierzu Kapitel 3. Der Benutzer, der ein System mit Posen und Gesten bedienen soll, kann hier außen vor gelassen werden, da er weder direkt mit der Erkennungs-Bibliothek noch mit dem Aufnahme-Werkzeug in Kontakt kommt.

Der Softwareentwickler wird sich hauptsächlich für die Erkennungs-Bibliothek interessieren. Das Aufnahme-Werkzeug wird wahrscheinlich allenfalls dann interessant, wenn die vom Auftraggeber o.ä. bereitgestellte Datenbank überprüft oder im Rahmen der Fehlersuche vielleicht auch analysiert werden soll. Für ihn ist eine zweckmäßige Schnittstelle zur Erkennungs-Bibliothek deutlich wichtiger als eine stilvoll gestaltete Benutzeroberfläche beim Aufnahme-Werkzeug.

Anders herum verhält es sich für einen User Interface Designer oder dergleichen: Dieser kommt mit der Erkennungs-Bibliothek niemals direkt in Kontakt, für ihn ist es nur wichtig, möglichst schnell und unkompliziert ein Posen- und Gestenrepertoire aufzubauen und einzusetzen (bzw. einem Softwareentwickler die erstellte Datenbank bereitzustellen und ihm mitzuteilen, welche Pose oder Geste welche Wirkung haben soll).

Der Fokus soll im Rahmen dieser Arbeit auf der in Abschnitt 5.3 beschriebenen Erkennungs-Bibliothek, im weiteren Verlauf „*libgrr*" genannt, liegen. Beim Aufnahme-Werkzeug, im weiteren Verlauf „*grrtool*" genannt, handelt es sich – wie in Abschnitt 4.4.2 erklärt – im Wesentlichen um eine spezielle Anwendung: Sie nutzt eben auch nur die Mechanismen, die jeder anderen Anwendung ebenfalls zur Verfügen stehen. Die Beschreibung des *grrtool* in Abschnitt 5.4 beschränkt sich daher hauptsächlich auf dessen Interaktion mit *libgrr*.

Bevor diese beiden Softwarekomponenten jedoch im Detail betrachtet werden, soll in Abschnitt 5.1 zunächst auf die Datenstrukturen der Wissensbasis eingegangen werden, da diese für beide Softwarekomponenten eine zentrale Rolle spielen. Im Anschluss daran werden die bei der Realisierung getroffenen grundsätzlichen Designentscheidungen in Abschnitt 5.2 vorgestellt.

Die in diesem Abschnitt abgebildeten Klassendiagramme wurden mit dem Eclipse-Plugin „ObjectAID" erzeugt, welches eine Darstellung erzeugt, die in einigen Punkten einem klassi-

chen UML-Klassendiagramm ähnelt. Es handelt sich bei den erzeugten Grafiken jedoch *nicht* um UML-Klassendiagramme. Einfache Assoziationen werden durch einen durchgezogenen Pfeil dargestellt, die Implementierung einer Schnittstelle mit einem gestricheltem Pfeil. Die Assoziation sollte also nicht mit einer Vererbungsrelation verwechselt werden.

5.1. Wissensbasis: Datenstrukturen und Datenbankformat

Um in den folgenden Teilabschnitten eine möglichst genaue Vorstellung von den beteiligten Daten zu erhalten, bietet es sich an, zunächst einen Blick auf die eingesetzten Datenstrukturen zu werfen, aus denen sich die Wissensbasis zusammensetzt. Da diese Datenstrukturen auch die Grundlage des Datenbankformats bilden, bietet es sich zusätzlich an auch auf dieses in diesem Teilabschnitt kurz einzugehen. Abbildung 5.1 zeigt die strukturelle Hierarchie der verwendeten Klassen.

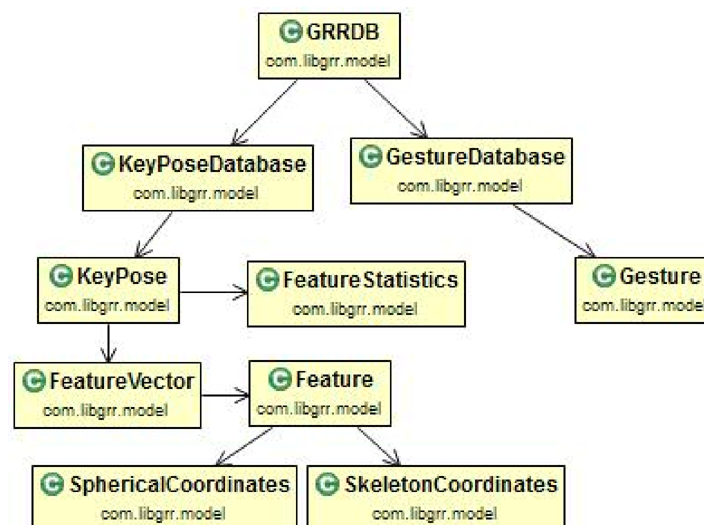


Abbildung 5.1.: Datenstrukturen der Wissensbasis (GRRDB)

Grundsätzlich werden zwei verschiedene Formen von Koordinaten unterschieden: Sphärische Koordinaten und Skelett-Koordinaten. Skelett-Koordinaten entsprechen weitestgehend den räumlichen Koordinaten, wie sie vom Sensor geliefert werden und werden in Objekten vom Typ `SkeletonCoordinates` mit den Feldern `u`, `r` und `t` gekapselt. Diese entsprechen in einem konventionell benannten kartesichen Koordinaten-System der z-, x- und y-Achse (gewissermaßen vertikal, von links nach rechts und von vorne nach hinten). Die Abweichung hat ihren Ursprung zum einen in der dieser Arbeit zu Grunde liegenden Literatur (Raptis u. a.

(2011); Miranda u. a. (2012, 2014)) und in der Tatsache, dass die hier gespeicherten Koordinaten sich nach deren Vorverarbeitung eben nicht mehr auf das Sensor-Koordinatensystem beziehen, sondern auf lokale Koordinatensysteme auf Basis der Torso-Basis (vgl. hierzu Abschnitt 2.2.5). Die Skelett-Koordinaten werden genutzt, um die sphärischen Koordinaten, `theta` und `phi` in der Klasse `SphericalCoordinates`, zu berechnen. Danach wären sie für Training und Wiedererkennung von Posen eigentlich überflüssig, sie werden aber nicht verworfen, weil sie für die Darstellung der Daten z.B. im *grrtool* als Skelett-Zeichnung sehr hilfreich sind.

Ein zusammengehöriges Paar aus Skelett- und sphärischen Koordinaten wird in einem `Feature` in den Feldern `skeleton` und `sphere` zusammengefasst. Ein `Feature`-Objekt speichert die lokale Basis (`localBasis`) ab, auf die sich die gekapselten Koordinaten beziehen.

Die Grundlage für das Training und die Wiedererkennung von Posen bilden `FeatureVector`-Instanzen. Diese beinhalten die sphärischen Koordinaten aller Merkmale in einem einfachen Array von `double`-Werten, das zu diesem Zweck genutzt wird. `Feature`-Objekte werden nur gespeichert, um sonstige Arbeiten mit diesen Daten zu erleichtern, falls sich ein Entwickler für konkrete Zahlenwerte einzelner Merkmale interessiert.

Die Zuordnung von `Feature`-Objekten zu den Skelettmerkmalen erfolgt immer über `Map`s. Den Schlüssel bilden Ausprägungen des Enum-Typen `FeatureType` wie z.B. `WRIST_RIGHT`. Dieser Enum-Typ bietet Informationen über Grad des Merkmals sowie nächst-inneren Nachbar und legt fest, ob es in den `FeatureVector` aufgenommen werden soll oder nicht. Um *libgrr* mit einem anderen Körpermodell zu verwenden, muss in erster Linie dieser Enum-Typ angepasst werden. Die einzelnen `Feature`-Objekte zu den einzelnen Merkmalen sind in genau solch einer `Map` in einem `FeatureVector` hinterlegt.

Ein `KeyPose`-Objekt ist nicht viel mehr als eine `List` von `FeatureVector`-Objekten, die die Trainingsbeispiele für eine Schlüsselpose darstellen und ordnet der beschriebenen Schlüsselpose eine ID und einen Titel zu. Um die Verwaltung und Analyse eine Schlüsselposenrepertoires zu erleichtern werden außerdem Statistiken (Minimal-, Maximal- und Durchschnittswert, Standardabweichung) geführt und in `FeatureStatistics`-Objekten gespeichert.

Alle ein konkretes Schlüsselposenrepertoire bildenden `KeyPose`-Objekte werden in einer `KeyPoseDatabase` zusammengefasst. Analog verhält es sich mit `Gesture`-Objekten, die in einer `GestureDatabase` zusammengefasst werden. Ein `Gesture`-Objekt selbst ist nicht viel mehr als eine `List` von Schlüsselposen-IDs, ergänzt um eine Gesten-ID und einen Titel. Diese beiden Teildatenbanken werden wiederum in GRRDB, der Wissensbasis, zusammengefasst, wo Methoden zum Laden, Erstellen und Speichern der Gesamtdatenbank bereitstehen. Eine abgespeicherte Datenbank ist nichts weiter als ein serialisiertes GRRDB-Objekt.

5.2. Designentscheidungen

Dieser Abschnitt stellt eine Reihe grundlegender Designentscheidungen vor, die die Arbeit mit *libgrr* erleichtern sollen. Es ist nicht zwingend notwendig, sich auf die damit angebotenen Möglichkeiten einzulassen, wenngleich dies in hohem Maße empfohlen ist, da damit die Komplexität von auf *libgrr* aufsetzenden Programmen reduziert werden kann.

5.2.1. Inversion of Control (IoC) und Ereignisse

In den folgenden Abschnitten werden des Öfteren verschiedene Ereignisse erwähnt. Sie sind der bevorzugte Weg zur Übertragung von Arbeitsergebnissen der einzelnen Module. Beobachter solcher Ereignisse implementieren dem jeweiligen Ereignis entsprechende Schnittstellen und können darauf hin bei einem Ursprungsobjekt registriert werden

Das Verwenden dieser Ereignisse ist, sofern möglich, den anderen gezeigten Interaktions-Möglichkeiten vorzuziehen, da *libgrr* selbstständig entsprechend registrierte Beobachter zu den richtigen Zeitpunkten aufruft, die wichtigsten Daten dabei zur Verfügung stellt und so dabei hilft *libgrr* verwendende Programme so unkompliziert wie möglich zu halten (Inversion of Control, IoC).

Konsequente Kommunikation über Ereignisse bietet zudem die Möglichkeit, einzelne Programm-module vollständig voneinander zu entkoppeln („minimale Kopplung, maximaler Zusammenhalt“). Dadurch verbessert sich besonders die Test- und Wartbarkeit des Programms.

Generell gilt: Ein Ereignis-Empfänger muss, um entsprechend als solcher bei einem Objekt registriert werden zu können, die `I<Ereignis-Name>Listener`-Schnittstelle implementieren. Weiterführende Details zu allen von *libgrr* ausgelösten Ereignissen lassen sich Anhang A entnehmen.

5.2.2. Fassade für *libgrr*-Module: Vereinfachte Schnittstelle

Um die Arbeit mit *libgrr* insbesondere für solche Programme zu vereinfachen, die lediglich Informationen über erkannte Schlüsselposen und Gesten benötigen und keinen Bedarf für darüber hinausgehende Daten haben, kann eine Fassaden-Klasse `LibGRR` für die einzelnen Module der *libgrr*-Bibliothek verwendet werden.

Diese Klasse instanziert die benötigten Teile von *libgrr* und verbindet sie mit Hilfe passender Ereignisse so untereinander, dass eine durchgängige Verarbeitungspipeline entsteht. Auch die `LibGRR`-Klasse selbst registriert sich als Beobachter bestimmter Ereignisse an den einzelnen Modulbestandteilen, um neben der ereignisbasierten Schnittstelle eine passive Schnittstelle von eher prozeduralem Charakter anzubieten. Entwickler haben so die Option, nachdem ein

Eingabe-Frame per `update`-Methode zur Verarbeitung übergeben wurde, mit den `getDetectedKeyPoses`- und `getDetectedGestures`-Methoden selbst aktiv zu werden und die Ergebnisse der Verarbeitung des letzten Eingabe-Frames abzufragen. Dies soll es insbesondere unerfahrenen Entwicklern, die sich an das IoC-Pattern oft erst noch gewöhnen müssen, vereinfachen, die Bibliothek einzusetzen.

Alle Modulbestandteile sind über entsprechende Getter-Methoden erreichbar. Es ist so auch möglich, beliebig tief in die Bibliotheksinterna hinabzusteigen, sofern eine komplexere Anwendung dies erfordert. Eine sehr ausführliche API-Dokumentation der `LibGRR`-Klasse erleichtert die Nutzung und besonders den Einstieg weiter, indem z.B. die Getter-Methoden für die einzelnen Modulbestandteile beschreiben, welche Ereignisse wann am jeweiligen Bestandteil auftreten, Hinweise zu deren Handhabung geben oder die Dokumentation der Methoden der Schnittstelle mit eher prozeduralem Charakter Hilfestellung für die Verwendung der ereignisbasierten Schnittstelle bieten. Der Einstieg über diese Fassade ist daher in jedem Fall empfehlenswert.

5.2.3. Systemumgebung

Es wurden verschiedene Maßnahmen ergriffen um das Gesamtsystem so flexibel wie möglich zu gestalten. Dies betrifft gleichermaßen die unterstützten Plattformen und die unterstützte Soft- und Hardwaretopologie. Dieser Unterabschnitt soll diese Maßnahmen beschreiben.

Plattformunabhängigkeit

Die Hauptbestandteile des Systems (*libgrr* und *grrtool*) sind weitgehend plattformunabhängig, da es sich um Java-Software handelt. Das Aufnahme-Werkzeug *grrtool* ist lauffähig unter Windows, Linux und MacOS X, jeweils in der 32- oder 64-bit-Ausführung. Die Erkennungs-Bibliothek *libgrr* unter noch mehr Plattformen, da es selbst „headless“ ist (mindestens zusätzlich Linux für ARM-Architektur). Das *grrtool* beinhaltet das für die grafische Benutzeroberfläche benötigte SWT für alle genannten Plattformen und lässt sich nach Installation einer JRE verwenden.

Netzwerkbasiertheit

Das System beinhaltet alles Notwendige, um es mit Hilfe seiner Netzwerkunterstützung in fast beliebigen Hard- und Softwaretopologien (lediglich eingeschränkt durch die Grenzen der Plattformunabhängigkeit) einzusetzen. In diesem Abschnitt sollen die verfügbaren Möglichkeiten vorgestellt werden.

Durch die Verwendung des Kinect-Sensors ist es meist nötig ein Windows-System zur Verfügung zu haben, an dem der Sensor angeschlossen ist. Eingeschränkt lassen sich zumindest einige Kinect-Sensoren auch unter anderen Plattformen wie Linux verwenden, dies im Detail zu thematisieren würde jedoch den Rahmen dieser Arbeit sprengen. Dem hier entwickelten System liegt ein Programm „SensorDataProvider" bei, welches die Sensor-Daten auf einem Windows-System entgegennimmt und auf einem TCP-Socket in Form von JSON-Nachrichten bereitstellt. Das genaue Nachrichtenformat ist Anhang C zu entnehmen.

libgrr selbst beinhaltet Hilfsmittel zum Empfangen eines solchen Datenstroms. Die Bereitstellung der Arbeitsergebnisse von *libgrr* kann ebenfalls einfach über das Netzwerk erfolgen. Um Sensor-Daten über das Netzwerk zu empfangen und die Arbeitsergebnisse selbst wieder im Netzwerk bereitzustellen ist keinerlei Programmierung erforderlich: *libgrr* selbst kann, anstatt in ein bestehendes Java-Programm eingebunden zu werden, auch als eigenständiges Programm gestartet werden. Damit ist es nicht nur möglich das Gesamtsystem auf unterschiedlichste Teilsysteme zu verteilen, der Entwickler behält außerdem vollständige Freiheit über die Wahl der Programmiersprache. Die hier kurz vorgestellten Hilfsmittel werden in Abschnitt 5.3.4 genauer erklärt.

5.2.4. Zusammenfassung

Es wurden einzelne grundlegende Designentscheidungen bei der Entwicklung des Systems genannt und sie selbst sowie ihre Auswirkungen beschrieben. Dies betrifft zum einen Entscheidungen auf Modul-Ebene – also solche, die sich im wesentlichen in der Klassenstruktur und den angebotenen Schnittstellen niederschlagen – und zum anderen Entscheidungen auf Komponenten-Ebene, die generell die Möglichkeiten zum Einsatz des Gesamtsystems beeinflussen.

5.3. Kern-Komponente: *libgrr*-Programmbibliothek

Die Hauptrollen innerhalb von *libgrr* spielen die drei Klassen `PoseProcessor`, `KeyPoseDetector` und `GestureDetector`. Die Bereitstellung von Arbeitsergebnissen der einzelnen Komponenten erfolgt vorzugsweise (auch untereinander) über eine Reihe spezifischer Ereignisse. Innerhalb von *libgrr* findet zunächst die Rohdatenaufbereitung statt. Die aufbereiteten Daten werden dann an die Schlüsselposen-Erkennung weitergegeben. Diese wiederum meldet erkannte Schlüsselposen an die Gesten-Erkennung. Dieser Datenfluss innerhalb von *libgrr* und die ereignisbasierte Schnittstelle sind in Abbildung 5.2 dargestellt.

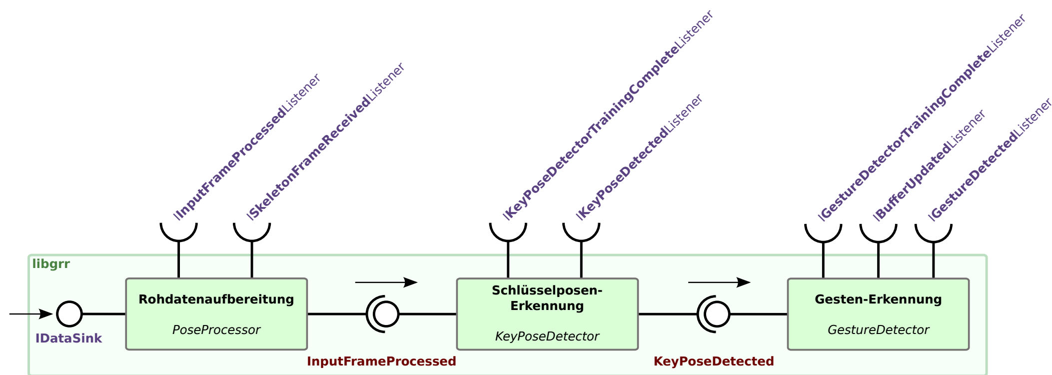


Abbildung 5.2.: Veranschaulichung des Datenflusses durch die Hauptmodule von *libgrr* mit Illustration der Ereignis-basierten Schnittstelle

Die Klasse `LibGRR` bildet eine Fassade um diese drei Hauptmodule und stellt ihnen die zu verwendende Wissensbasis bereit. Abbildung 5.3 illustriert diesen groben Überblick aus struktureller Perspektive in Form eines Klassendiagramms. Der Übersichtlichkeit halber sind nur die wesentlichen Ereignis-Schnittstellen erkennbar, die auch genutzt werden um Daten innerhalb von *libgrr* von Verarbeitungsschritt zu Verarbeitungsschritt zu transportieren. Die folgenden Teilabschnitte erklären den Aufbau und die Funktionsweise der Bibliothek genauer.

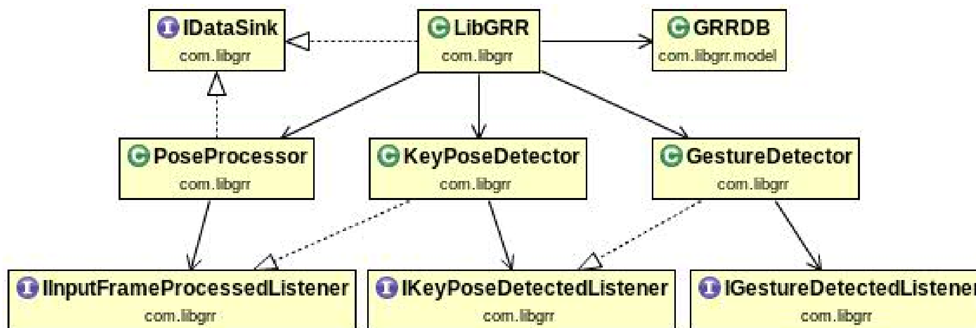


Abbildung 5.3.: Grober Überblick über die wichtigsten Bestandteile von *libgrr*

5.3.1. Rohdatenaufbereitung

Bevor die Sensordaten zur Aufzeichnung und Wiedererkennung von Körperposen verwendet werden können, müssen sie geeignet aufbereitet werden. Diese Datenaufbereitung umfasst die Bestimmung der jeweiligen Torso-Basis, die Umrechnung der Skelett-Koordinaten in ein Koordinaten-System relativ zu eben dieser und daraus die Berechnung der sphärischen

Koordinaten, die später für die Posen-Klassifizierung verwendet werden. Diese Aufbereitung wird von der `PoseProcessor`-Klasse durchgeführt. Das grundsätzliche Verfahren hierbei wurde auf Basis von Miranda u. a. (2012) bereits in Abschnitt 2.2.5 beschrieben.

Einstiegspunkt der Datenaufbereitung bildet die in `IDataSink` spezifizierte `update`-Methode. Sie erhält als Eingabe eine Reihe von Skelett-Modellen. Jedes Skelett-Modell ist eine `Map`, die `FeatureType`-Ausprägungen `Feature`-Objekte zuordnet. Solch eine `Map` wird im Folgenden nur noch als „Skelett" bezeichnet. Die `Feature`-Objekte enthalten bisher noch keine sphärischen Koordinaten oder eine lokale Bezugsbasis. Sie beziehen sich also noch auf das Sensor-Koordinatensystem. Da ein Sensor mehrere Personen gleichzeitig verfolgen kann, kann eine Eingabe auch mehrere Skelette enthalten. Die an `update` übergebene Datenstruktur ist daher eine `Map`, die eine Skelett-ID auf ein Skelett abbildet. Diese `Map` bildet einen Eingabe-Frame bzw. „InputFrame".

Die Verarbeitung erfolgt für jedes Skelett in einem Eingabe-Frame gleichermaßen. Zumindest für die „Microsoft Kinect for xBox 360" wird zunächst die vertikale Achse am Ursprung gespiegelt, da die Skelett-Modelle ansonsten „auf dem Kopf" stehen. Anschließend werden mittels Hauptkomponentenanalyse die u- und die r-Achse der Torso-Basis bestimmt: Es handelt sich bei ihnen um die ersten beiden Hauptkomponenten[1] aller Torso-Merkmale (also solche `Features`, deren `FeatureType`-Ausprägung den Grad 0 hat), sie stehen also senkrecht aufeinander. Die t-Achse ergibt sich einfach aus dem Kreuzprodukt der u- und r-Achse. Da die Orientierung der Hauptkomponenten sehr instabil ist und häufig das Vorzeichen wechselt, wird diese bei Bedarf gespiegelt. Zusätzlich werden die Koordinaten des Torso-Mittelpunktes berechnet.

Um die Koordinaten vom Sensor-Koordinatensystem in das durch die jeweilige Torso-Basis aufgespannte umzurechnen, wird eine Matrix für einen entsprechenden Basiswechsel berechnet. Alle Koordinaten werden dann relativ zum jeweiligen Torso-Mittelpunkt und bezüglich der Torso-Basis ausgedrückt. Auf dieser Grundlage werden dann zunächst die sphärischen Koordinaten der Merkmale ersten Grades berechnet. Anschließend werden dann sukzessive die Koordinaten von Merkmalen höherer Ordnung berechnet, da bei diesen die verwendeten lokalen Koordinatensysteme Rotationen der lokalen Koordinatensysteme des jeweils nächst-inneren Merkmals verwenden. Vgl. auch hierzu Abschnitt 2.2.5.

Da die Winkel zur Bezugsachse um so stärker Schwanken, je kleiner der Winkel zur Polachse wird, werden die Azimuth-Winkel anhand des Polwinkels gewichtet. Konkret wird

[1] Eine Hauptkomponente beschreibt die Richtung, entlang der die Varianz einer Stichprobe maximal ist. Die erste Hauptkomponente beschreibt also die Richtung der größten Varianz und verläuft damit etwa parallel zur Wirbelsäule. Die zweite Hauptkomponente beschreibt die Richtung mit der zweit-größten Varianz (parallel zur Schulterlinie), usw.

der Faktor zur Gewichtung durch die Funktion $f(\theta) = \frac{1}{2}\left(1 + \tanh\left(\frac{\theta - 20^{\circ}}{6}\right)\right)$ berechnet. Die Auswirkungen der Schwankungen (oder auch Sprünge zwischen 180° und -180°) konnten auf diese Weise stark reduziert werden, womit die Erkennungsleistung deutlich gesteigert werden konnte.

Die ursprünglich in die `update`-Methode hereingegebenen Daten werden so schrittweise aufbereitet und ergänzt. Für jedes verarbeitete Skelett wird ein `SkeletonFrameReceived`-Ereignis erzeugt. Sobald alle Skelette in einem Eingabe-Frame verarbeitet wurden wird zusätzlich ein `InputFrameProcessed`-Ereignis erzeugt. Weiterführendes allgemeines zu Ereignissen findet sich in Abschnitt 5.2.1, eine vollständige Liste aller Ereignisse kann dem Anhang A entnommen werden.

5.3.2. Schlüsselposen-Erkennung

Die so aufbereiteten Daten können nun von der `KeyPoseDetector`-Klasse weiterverarbeitet werden. Zur Klassifizierung einer Pose wird der zugehörige `FeatureVector` zur Beurteilung an die mit dem Schlüsselposen-Repertoire trainierte Support Vector Machine (SVM) übergeben. Abbildung 5.4 zeigt den groben Aufbau der `KeyPoseDetector`-Klasse. Dieser Aufbau, der Ablauf der Posen-Klassifizierung bzw. Schlüsselposen-Erkennung und die Möglichkeiten zur Interaktion mit dieser Klasse werden im Folgenden näher beschrieben.

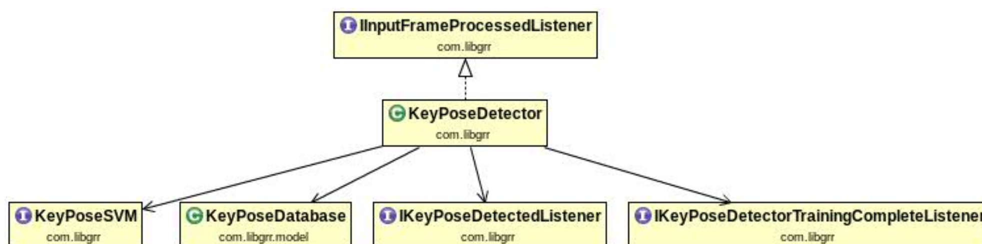


Abbildung 5.4.: Aufbau des `KeyPoseDetectors`

Kern der `KeyPoseDetector`-Klasse bildet die genutzte `KeyPoseSVM`-Implementierung. Sie erlaubt es über eine sehr einfache Schnittstelle auf eine prinzipiell beliebige Support Vector Machine (SVM) Implementierung zuzugreifen: Die Implementierung muss lediglich eine `train`-Methode zum Einlernen der SVM, sowie eine (für wahlweise Skelett oder `FeatureVector` überladene) `getProbabilities`-Methode zum Bestimmen der Verteilung der Wahrscheinlichkeit, dass ein übergebener Datansatz einer der einzelnen Schlüsselposen entspricht. Aktuell ist die Verwendung einer Adapter-Klasse zu „libsvm“(Chang und Lin, 2011) in der Version 3.18 hart kodiert, die verwendete SVM ließe sich jedoch leicht auswechseln.

Bevor nun ein `KeyPoseDetector`-Objekt zur Wiedererkennung von Schlüsselposen eingesetzt werden kann muss es anhand einer Datenbank trainiert werden. Hierzu braucht lediglich das der Erkennung zu Grunde zu legende `KeyPoseDatabase`-Objekt an die `train`-Methode übergeben werden. Dieser Aufruf wird an die `KeyPoseSVM`-Implementierung delegiert und anschließend ein `KeyPoseDetectorTrainingComplete`-Ereignis erzeugt.

Eine Wiedererkennung von Schlüsselposen ist auf drei verschiedenen Wegen möglich. In den meisten Fällen wird es am bequemsten sein das `KeyPoseDetector`-Objekt bei einem `PoseProcessor` zur Beobachtung von `InputFrameProcessed`-Ereignissen zu registrieren. Sobald ein solches Ereignis eintritt wird die `detectKeyPoses`-Methode mit den vorbereiteten Skelett-Daten aufgerufen. Diese Methode lässt sich auch manuell aufrufen. Ihr Rückgabewert gibt an, für welches Skelett welche Schlüsselpose festgestellt wurde (sofern überhaupt eine festgestellt wurde). Sobald für ein Skelett eine Schlüsselpose erkannt wurde, wird ein `KeyPoseDetected`-Ereignis ausgelöst.

Die tatsächliche Wiedererkennung selbst wird pro Eingabe-Skelett von der `detectKeyPose`-Methode durchgeführt, die hierzu auf die `KeyPoseSVM` Implementierung zurückgreift. Auch diese Methode ist öffentlich und kann bei Bedarf direkt angesprochen werden, hierbei muss allerdings beachtet werden, dass sie bei der Erkennung einer Schlüsselpose kein Ereignis auslöst. Eine Schlüsselpose gilt als erkannt, wenn sie diejenige mit der höchsten ermittelten Wahrscheinlichkeit ist und diese über einem einstellbaren Schwellenwert von `KeyPoseDetector`.DEFAULT_THRESHOLD = 75% liegt.

5.3.3. Gesten-Erkennung

Das grundsätzliche Vorgehen beim Erkennen von sich aus einzelnen Schlüsselposen zusammensetzenden Gesten ähnelt dem beim Erkennen von Schlüsselposen: Eine zu beurteilende Schlüsselposenfolge wird von der anhand eines Gestenrepertoires trainierten `GestureDetector`-Klasse klassifiziert. Einen Überblick der beteiligten Klassen und Schnittstellen zeigt Abbildung 5.5.

Ein `GestureDetector`-Objekt wird mittels `train`-Methode auf das in einem `GestureDatabase`-Objekt gespeicherte Gestenrepertoire trainiert. Hierzu wird wie in 2.2.5 beschrieben eine Wald-Struktur aufgebaut, mit der effizient festgestellt werden kann, ob eine Liste von Schlüsselposen einer bekannten Geste entspricht.

Ein `GestureDetector`-Objekt pflegt für jede bekannte Skelett-ID einen eigenen Puffer, in den die IDs der Schlüsselposen eingetragen werden, die für das jeweilige Skelett erkannt wurden. Dabei wird nur dann ein Eintrag vorgenommen, wenn sich die hinzuzufügende Schlüsselpose von der zuletzt hinzugefügten unterscheidet. Wenn dies der Fall ist (oder wenn

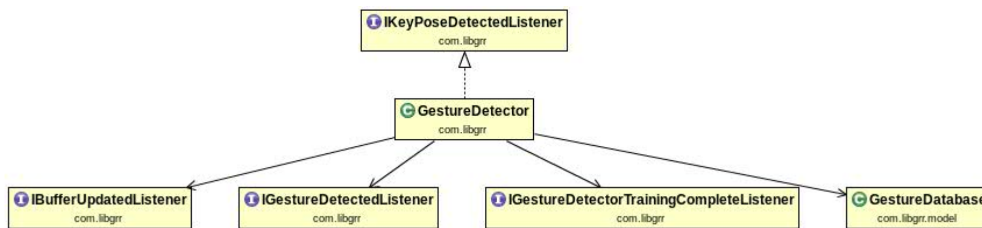


Abbildung 5.5.: Aufbau des GestureDetectors

die Puffer mittels clearBuffers-Methode verworfen werden), wird ein BufferUpdated-Ereignis ausgelöst, das bei Bedarf verarbeitet werden kann. Stellen die zuletzt zu einem Puffer hinzugefügten Schlüsselposen eine bekannte Geste dar, so wird außerdem ein entsprechendes GestureDetected-Ereignis ausgelöst. Das etwaige Hinzufügen von Schlüsselposen zu ihrem jeweiligen Puffer erfolgt durch die addKeyPoseToBuffer-Methode, die auch direkt auf einem GestureDetector-Objekt aufgerufen werden kann. Die tatsächliche Suche nach einer bekannten Geste erfolgt innerhalb der detectGesture-Methode, die ebenfalls direkt aufgerufen werden kann, jedoch keine Ereignisse auslöst und bekannte Gesten in einer übergebenen Folge von Schlüsselposen-IDs sucht – hier wird im Normalfall eben der aktuelle Pufferinhalt übergeben.

Ähnlich zu der Erkennung von Schlüsselposen ist es auch hier für die Wiedererkennungsaufgabe am bequemsten, das GestureDetector-Objekt als Beobachter von KeyPoseDetected-Ereignissen an einem KeyPoseDetector zu registrieren. Es wird dann bei Auftreten eines KeyPoseDetected-Ereignisses die erkannte Schlüsselpose an die addKeyPoseToBuffer-Methode weitergegeben. Je nach Anwendungsfall kann allerdings auch wie oben beschrieben direkt mit den addKeyPoseToBuffer- oder detectKeyPose-Methoden gearbeitet werden. Es ist mit der loadKeyPoseSequence-Methode außerdem möglich, eine Folge von Schlüsselposen direkt in einen Puffer zu laden. In disem Fall ist das Verhalten identisch mit sukzessiven Aufrufen der addKeyPoseToBuffer-Methode.

5.3.4. Hilfsmittel

In *libgrr* sind verschiedene Hilfsmittel enthalten, die den Einsatz und die Anbindung der Bibliothek vereinfachen sollen. Insbesondere dienen diese Hilfsmittel der Netzwerkanbindung und entsprechender Nachrichten-Verarbeitung, sowie der Nutzung der Bibliothek als eigenständig lauffähiges Programm. Diese Hilfsmittel werden in diesem Abschnitt näher betrachtet.

Die Klasse SocketListener macht es sehr einfach möglich, Daten in der Form wie sie vom SensorDataProvider (vgl. Abschnitt 5.2.3) bereitgestellt werden zu empfangen

und an die Rohdatenaufbereitung weiterzugeben. Dadurch können Sensor-Daten leicht aus beliebigen Quellen entgegengenommen werden. Auf *libgrr* basierende Anwendungen müssen sich zudem nicht um die Sensor-Anbindung zu kümmern, da die Daten (natürlich auch auf dem selben Rechner) z.B. vom `SensorDataProvider` bereitgestellt werden können. Das Nachrichtenformat ist in Anhang C spezifiziert. Das *grrtool* macht von dieser Möglichkeit Gebrauch und empfängt die Sensor-Daten über das Netzwerk: Standardmäßig vom lokalen System, alternativ von jedem beliebigen anderen. Details hierzu können Anhang B entnommen werden.

Weiterer Teil von *libgrr* ist die Klasse `SocketWriter`, die es ermöglicht auf ebenso einfache Weise Arbeitsergebnisse der Verarbeitung durch *libgrr* bereitzustellen. Welche Daten dies sind, ist konfigurierbar: Erkannte Schlüsselposen, Wahrscheinlichkeitsverteilung über die einzelnen Schlüsselposen und/oder erkannte Gesten. Hierdurch kann *libgrr* alle relevanten Informationen bezüglich der Körperposen- und Gestenerkennung universell für jede andere Software oder Plattform verfügbar machen. Auch hier kann das genaue Nachrichtenformat dem Anhang C entnommen werden.

Die Klasse `Util` enthält einige in diesem Zusammenhang nützliche Funktionen: `parseJSONSkeltonData`, `keyPoseMapToJSON`, `gestureMapToJSON` und `probabilityMapToJSON`. Diese Methoden dienen dazu JSON-Nachrichten gemäß Anhang C in die von *libgrr* verwendeten Datenstrukturen zu übertragen oder umgekehrt.

Falls *libgrr* nicht als Bibliothek in ein Java-Projekt eingebunden werden soll können die o.g. Klassen natürlich nicht ohne weiteres verwendet werden. *libgrr* kann deshalb auch direkt als eigenständiges Programm ausgeführt werden. In dem Fall werden (geregelt in der Klasse `SOGRR`) `SocketListener` und `SocketWriter` mit übergebenen oder Standard-Parametern eingerichtet, so dass Sensor-Daten über das Netzwerk empfangen werden und die Arbeitsergebnisse im Netzwerk bereitgestellt werden. So können die von *libgrr* bereitgestellten Kernfunktionen prinzipiell ohne jegliche Programmierung genutzt werden.

Der in `SocketWriter` implementierte Server (wie übrigens auch der in `SensorDataProvider` implementierte) legen ihren Fokus auf Datendurchsatz: Sollte die Verbindungsgeschwindigkeit nicht zur Übertragung aller Daten genügen, so werden Daten verworfen – Es entsteht also *kein* Stau. Im schlimmsten, wenn auch unwahrscheinlichen, Fall kann dies allerdings auch dazu führen, dass die Funktion leidet, weil Entscheidende Eingabe-Frames verloren gehen. Erreicht wird dieses Verhalten dadurch, dass die Schreib-Operationen auf den Sockets in separaten Threads erfolgen. Dabei wird pro Client zwar trotzdem nur ein Datensatz zur Zeit geschrieben, der Rest des Programmes läuft in der Zwischenzeit jedoch weiter. Beim

nächsten Schreib-Beginn auf einem Socket werden immer die zu diesem Zeitpunkt aktuellsten Daten übertragen.

5.3.5. Zusammenfassung

Der vorangehende Abschnitt hat im Detail die logische Struktur und die internen Abläufe innerhalb der *libgrr*-Bibliothek erläutert. Es wurde gezeigt, wie ein Schlüsselposen- und Gestenrepertoire strukturiert ist und die Rolle der beteiligten Datenstrukturen dargestellt. Es wurde außerdem die Unterteilung der Bibliothek in ihre drei wesentlichen zweckgebundenen Bestandteile skizziert und durch eine ausführliche Beschreibung des Datenflusses durch diese Bestandteile verdeutlicht, wie sie sich zu einem Ganzen zusammenfügen, bzw. unter Verwendung der auftretenden Ereignisse zusammenfügen lassen.

Abschließend wurden die in *libgrr* enthaltenen Hilfsmittel vorgestellt, die es Entwicklern erlauben, das Gesamtsystem zu verteilen, den Einsatz der Bibliothek zu erleichtern und die weitgehend freie Wahl der eingesetzten Programmiersprache zur Verwendung der *libgrr*-Funktionen ermöglichen.

5.4. Aufnahme-Werkzeug: *grrtool*

Das *grrtool* ist ein Werkzeug, das zur Bildung, Verwaltung und Analyse eines Schlüsselposen- und Gestenrepertoires eingesetzt werden kann. Es baut selbst auf *libgrr* auf und kann somit bedingt auch als Beispiel-Anwendung betrachtet werden. Kurzanleitung ist in Anhang B zu finden. An dieser Stelle wird lediglich kurz die Umsetzung des in Abschnitt 4.3 festgelegten Funktionsumfang anhand einiger Screenshots skizziert und erklärt, wie das *grrtool* von *libgrr* Gebrauch macht.

5.4.1. Realisierung das Funktionsumfanges

Die Screenshots in diesem Abschnitt vermitteln einen Eindruck, wie die in Abschnitt 4.3 genannten Funktionen des *grrtool* aus Anwendersicht umgesetzt wurden. Eine ausgiebige Beschreibung aller Funktionen würde den Rahmen dieses Kapitels jedoch sprengen. Eine vollständige Bedienungsanleitung wird *grrtool* beigelegt werden, eine Kurzanleitung findet sich in Anhang B.

Abbildung 5.6 zeigt die Aufnahmeansicht. Es wird gerade eine zuvor angefertigte Aufnahme betrachtet. Groß im Bild zu sehen ist die Visualisierung des aktuellen Skelett-Frames. Tabellen Auf der linken Seite des Fensters zeigen die Daten des aktuellen `FeatureVectors`, die

aktuelle Wahrscheinlichkeitsverteilung über alle Schlüsselposen und den Puffer-Inhalt des `GestureDetector`s. Die Buttons in der Tabelle, die die Wahrscheinlichkeitsverteilung zeigt, können genutzt werden, um Trainingsdaten hinzuzufügen. Mit den übrigen Buttons unten links ist es möglich neue Schlüsselposen anzulegen, die aufgenommene Sequenz als Geste zu speichern, die Aufnahme zu bearbeiten und innerhalb der Aufnahme zu navigieren.

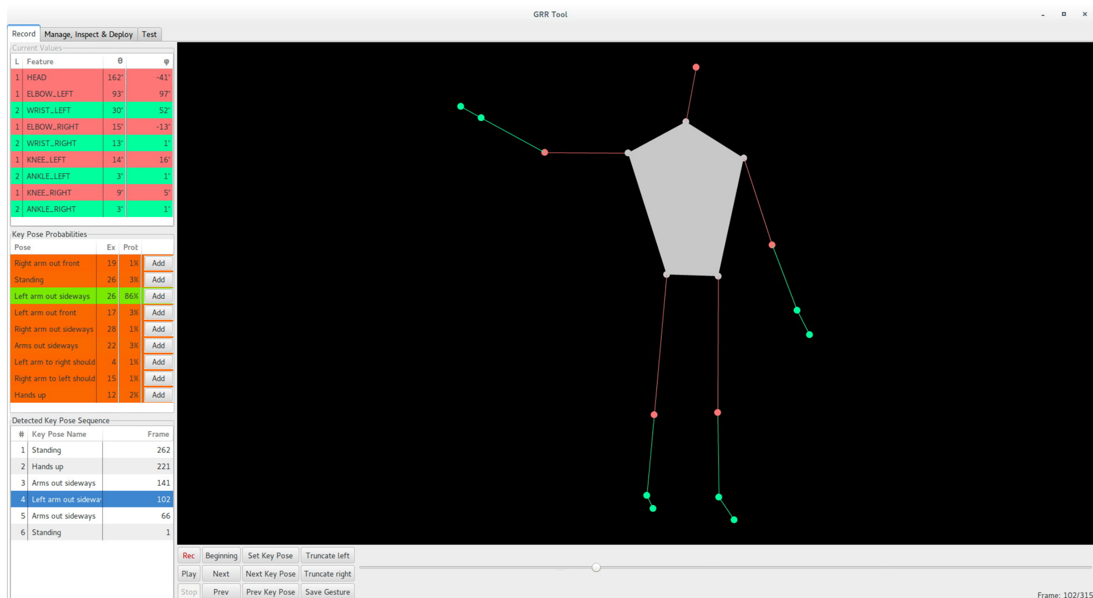


Abbildung 5.6.: *grrtool*-Aufnahmeansicht bei vorhandener Aufnahme: Visualisierung des Skelett-Frames mit zugehörigen `FeatureVector`-Daten, Schlüsselposen-Wahrscheinlichkeitsverteilung und `GestureDetector`-Pufferinhalt. Unten Bearbeitungs- und Navigations-Funktionen.

Abbildung 5.7 zeigt die Verwaltungs- und Analyse-Ansicht. Der obere Teil bietet eine Liste der bekannten Schlüsselposen sowie Einsicht in und Statistiken über die Trainingsdaten. Der Untere Teil zeigt bekannte Gesten und wie sie sich zusammensetzen. Hier können Schlüsselposen, Trainigsdaten und Gesten gelöscht und Wissensbasen exportiert werden.

Abbildung 5.8 zeigt die Test-Ansicht. Jeder erkannte Nutzer erhält während des Tests eine eigene Spalte, in der das jeweilige Skelett-Modell visualisiert und Informationen über Schlüsselposen und Gesten angezeigt werden. Es wird kenntlich gemacht, welche Gesten erkannt wurden bzw. in anbetracht des Puffer-Inhalts aktuell noch in Frage kommen. Es werden nur noch relevante Schlüsselposen angezeigt.

Für eine ausführlichere Beschreibung der *grrtool*-Funktionen sei nochmals auf den Anhang B verwiesen.

Abbildung 5.7.: *grrtool*-Verwaltguns- und Anaylseansicht: Oben Informationen zu Schlüsselposen, unten Informationen zu Gesten

5.4.2. Verwendung von *libgrr*

Der folgende Abschnitt erklärt einige Aspekte der Implementierung des *grrtool*. Der Fokus soll darauf liegen, wie *libgrr* verwendet wird, um bestimmte Funktionen umzusetzen. Dabei handelt es sich im Wesentlichen um die `Recorder`-Klasse zur Kontrolle der Aufnahme-Funktion und die `TestManager`- und `TestView`-Klasse zur Kontrolle der Test-Ansicht. Die Implementierungsdetails der grafischen Benutzeroberfläche selbst – die den Großteil des *grrtool* ausmachen – sind im Kontext dieser Arbeit bestenfalls als nebensächlich einzuschätzen und werden daher gemieden. Abbildung 5.9 zeigt einen Auschnitt der *grrtool*-Klassenstruktur und bildet die Grundlage der nachfolgenden Beschreibung.

Der Kern der Aufnahme-Logik verbirgt sich hinter der `Recorder`-Klasse. Ihre Hauptbestandteile sind `KeyPoseProcessor`, `GestureDetector` und eine Liste von Skelett-Datensätzen. Diese Liste stellt die von `Recorder` gekapselte Aufnahme dar. Die folgenden drei Ereignisse werden von einem `Recorder`-Objekt beobachtet:

`SkeletonFrameReceived` um vom Sensor bereitgestellte Skelett-Daten zu erhalten. Der `Recorder` entscheidet selbst (je nach Zustand), ob diese Daten einer Aufnahme hinzugefügt werden sollen oder nicht.

`BufferUpdatedListener` um bei Änderungen am `GestureDetector`-Puffer immer eine aktuelle Kopie zur weiteren Verwendung innerhalb des *grrtool* zur Verfügung zu haben.

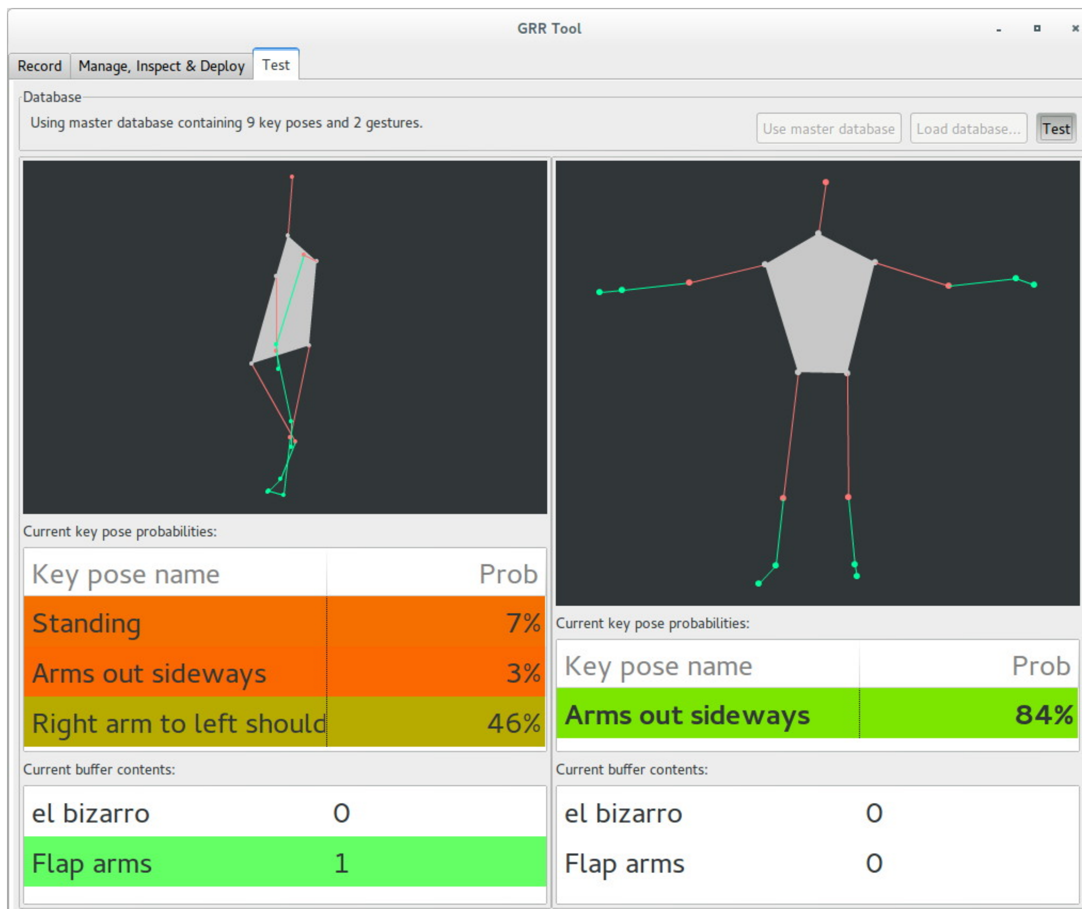


Abbildung 5.8.: *grrtool*-Testansicht: Zwei Test-Sichten für zwei Benutzer, jeweils mit aktuellen Informationen zu Schlüsselposen und Gesten

KeyPoseTrainingComplete damit eine gekapselte Aufnahme nach Änderungen der Wissensbasis erneut untersucht werden kann.

Die Recorder-Klasse soll bestimmen, was in der Aufnahme-Ansicht zu sehen ist. Daher ist die Aufnahme-Sicht selbst kein Beobachter zum Erhalt von Skelett-Daten (SkeletonFrameReceived) vom Sensor. Statt dessen löst Recorder ein ähnliches Ereignis aus, das beobachtet werden kann. Die damit übertragenen Daten können aber je nach Recorder-Zustand eben die aktuellen Sensor-Daten sein, oder z.B. bei der Wiedergabe einer Aufnahme ein aufgezeichnetes Skelett.

Bei Änderungen der Wissensbasis (bzw. genauer des Schlüsselposenrepertoires) muss eine Aufnahme vollständig neu verarbeitet werden, da die Klassifizierung jedes Frames der Auf-

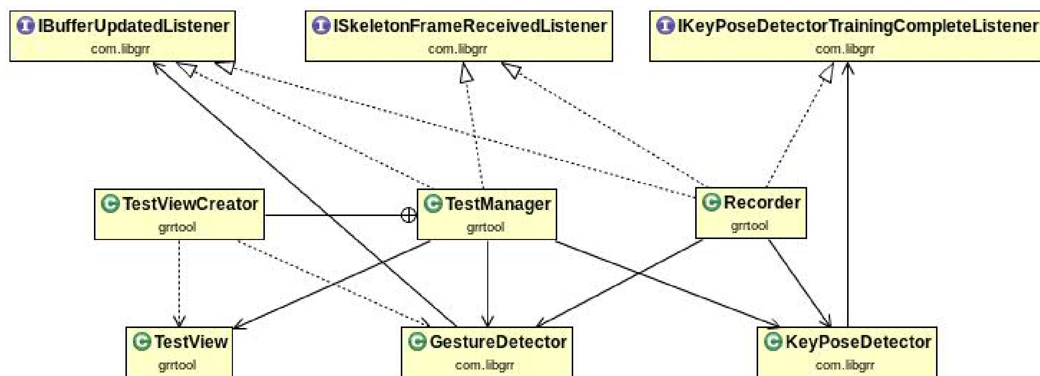


Abbildung 5.9.: Ausschnitt der *grrtool*-Klassenstruktur

nahme sich geändert haben könnte. Um dies zu ermöglichen wird der `KeyPoseDetector` vom `Recorder` direkt angesprochen, statt mittels Ereignis benachrichtigt, da ihm nämlich – wieder je nach Zustand – ein vom Sensor stammender Frame oder eben einer aus der Aufnahme bereitgestellt werden muss.

Die `TestManager`-Klasse stellt die Logik für die Test-Ansicht bereit. Sie beobachtet `SkeletonFrameReceived`-Ereignisse, um für jeden neuen erkannten Benutzer eine `TestView` zu instanzieren, die eine benutzerspezifische Test-Ansicht erzeugt. Diese Test-Ansicht zeigt Skelett, Schlüsselposenwahrscheinlichkeiten und Gesten. Der `TestManager` beobachtet `SkeletonFrameReceived`-Ereignisse. Jeder Skelett-Frame wird dann zunächst durch den `KeyPoseDetector` beurteilt und zusammen mit den erhaltenen Schlüsselposenwahrscheinlichkeiten an die zugehörige Test-Ansicht zur Darstellung weitergegeben. Alternativ könnte auch jede Test-Ansicht diese Aufgaben selbst durchführen.

Die `TestManager`-Klasse beobachtet ebenfalls `BufferUpdated`-Ereignisse um bei jeder Puffer-Änderung diese Daten an die zugehörige `TestView` weiterzugeben. Diese kann dann anhand des aktuellen Puffer-Inhalts mögliche Gesten-Kandidaten hervorheben. Die `TestView`-Objekte beobachten `GestureDetected`-Ereignisse, um die Erkennung einer Geste signalisieren zu können.

5.5. Zusammenfassung

Im Laufe dieses Kapitels wurde die Realisierung des Gesamtsystems ausführlich erläutert. Die Erläuterung der grundlegenden Datenstrukturen und darin verwendeten Klassen bildet die Grundlage für die Beschreibung der weiteren Implementierung, da ohne eine Vorstellung der beteiligten Daten ein nachvollziehen der internen Abläufe kaum möglich ist.

Die grundlegenden Designentscheidungen (IoC, Fassaden bzw. verschiedene Schnittstellen, Plattformunabhängigkeit, Netzwerkbasiertheit) wurden vorgestellt und erklärt. Im Anschluss daran wurde die Realisierung, insbesondere von *libgrr*, ausführlich beschrieben. Damit sollte ein grundsätzliches und hinreichend fundiertes Verständnis der Bibliothek erlangt worden sein.

Screenshots des *grrtool* veranschaulichten grob die Anordnung der in Abschnitt 4.3 für das Aufnahme-Werkzeug spezifizierten Funktionen. Die weitere Beschreibung des *grrtool* beschränkte sich auf die wesentlichen Teile der Interaktion mit *libgrr*.

Weiterführende Details zu Ereignissen, Benutzung des *grrtool* und Spezifikation der Netzwerk-Nachrichten können den Anhängen A, B und C entnommen werden.

6. Evaluation

Dieses Kapitel betrachtet die Leistung des entwickelten System unter verschiedenen Gesichtspunkten: In Abschnitt 6.1 werden die gegenseitig von einander abhängenden Größen Erkennungsleistung und Trainingsaufwand betrachtet. Nach entsprechendem Training liegen Erkennungsrate[1] und Vorhersagequalität[2] auch für größere Schlüsselposenrepertoires mit durchschnittlich 90% (Erkennungsrate) bzw. 80% (Vorhersagequalität) auf einem guten bis sehr guten Niveau.

In den Abschnitten 6.2 und 6.3 wird der Ressourcenaufwand für das Erreichen dieser Leistung untersucht. Die Ressourcennutzung ist dabei so gering, dass ein sehr breites Hardware-Spektrum als Plattform in Frage kommt.

6.1. Erkennungsleistung und Trainingsaufwand

Der Betrachtung in diesem Abschnitt liegt ein Schlüsselposenrepertoire mit 17 verschiedenen Schlüsselposen zu Grunde (Abbildung 6.1). Gesten werden hier nicht beachtet, da eine Geste nichts weiter als eine Sequenz von Schlüsselposen darstellt. Isoliert betrachtet (d.h. das Problem der Schlüsselposenerkennung außer Acht lassend) liegt die Erkennungsleistung des zur Gestenerkennung verwendeten Verfahrens inhärent bei 100%.

Für die meisten Anwendungen dürfte es zweckmäßig sein ein auf ihre individuellen Bedürfnisse zugeschnittenes Schlüsselposenrepertoire zu verwenden. Die Verwendung eines riesigen universellen Schllüsselposenrepertoires ist aus zwei Gründen nicht zu empfehlen:

1. Ähnliche Schlüsselposen müssen immer noch klar unterscheidbar sein. Das Training muss für große Schlüsselposenrepertoires also sehr präzise erfolgen und der Benut-

[1] Erkennungsrate meint das Verhältnis aus der Anzahl der Wiederholungen einer Pose und der Anzahl der korrekt erkannten Wiederholungen. Wurde Eine Pose bei 10 Versuchen 9 mal richtig wiedererkannt, dann beträgt die Erkennungsrate 90%.

[2] Vorhersagequalität meint die Wahrscheinlichkeit, die das System der jeweils eingenommen Pose letztendich zuordnet: Steht der Nutzer vor dem Sensor dann soll das System eine „Stehen“-Pose Erkennen. Die vom System berechnete Wahrscheinlichkeit, dass es sich bei der eingenommen Pose um die „Stehen“-Pose handelt, ist die Vorhersagequalität. Ist sich das System also zu 80% sicher, dass es sich um die „Stehen“-Pose handelt, so beträgt die Vorhersagequalität 80%. Ab 75% wird eine Pose als erkannt betrachtet.

zer muss sich bei Ausführung einer Schlüsselpose eng an die Vorgabe halten (so er sie denn überhaupt kennen soll). Insbesondere ohne Rückmeldung über die aktuelle Schlüsselposenwahrscheinlichkeitsverteilung wird dies schnell schwierig, woraufhin die Erkennungsleistung und damit die Systemakzeptanz schnell abnimmt.

2. Der für das Training zu treibende Aufwand ist ganz einfach nicht zumutbar.

Insofern ist das betrachtete Schlüsselposenrepertoire bereits relativ groß. Es ist jedoch nicht für eine konkrete Anwendung gedacht, sondern dient hauptsächlich der Betrachtung von Trainingsaufwand und Erkennungsleistung.

Der Trainings-Aufwand für das betrachtete Schlüsselposenrepertoire befindet sich sicherlich im Grenzbereich: Es wurden hierzu ca. drei bis vier Stunden in mehreren Trainingssitzungen benötigt. Hierbei muss zum einen bedacht werden, dass im Training mit dem System unerfahrene Anwender wahrscheinlich noch längere Zeit benötigen und möglicherweise den Aufbau des Repertoires einige Male von vorn beginnen würden (da sie durch unbedachtes Hinzufügen von Trainingsbeispielen die Trainingsmengen entarten lassen). Zum anderen muss jedoch berücksichtigt werden, dass das Training zur Erhebung einiger Kennzahlen durchgeführt wurde, die Herangehensweise daher nicht ganz der entsprach, wie sie vermutlich sonst gewählt worden wäre.

Dem Schlüsselposenrepertoire liegen insgesamt 427 Trainingsdatensätze zu Grunde. Um eine grobe Einschätzung geben zu können, wie sich die Vorhersagequalität der Schlüsselposen in Abhängigkeit der Anzahl der Trainingsbeispiele entwickeln, wurden für jede Schlüsselpose zunächst 4, dann 8, 12 und schließlich 16 Trainigsbeispiele festgelegt. Weiter wurde die schrittweise Erhöhung der Anzahl der Trainingsbeispiele pro Schlüsselpose nicht fortgesetzt. Statt dessen wurde zu einer praxisnäheren, iterativen Vorgehensweise übergegangen. Dabei wurden nur mit Hinblick auf den tatsächlichen jeweiligen Bedarf Trainingsdatensätze hinzugefügt und die Trainingsdatenmenge immer wieder mit Hilfe der Analysefunktionen des *grrtool* eingesehen und nötigenfalls bereinigt. Tabelle 6.1 zeigt die Entwicklung der Vorhersagequalität. Die für die einzelnen Schlüsselposen aus Abbildung 6.1 erreichten Erkennungsraten sind in Tabelle 6.2 aufgeführt. Hierbei wurde jede Pose 10 mal eingenommen.

In Tabelle 6.1 wird gut sichtbar, wie schnell sich die Vorhersagequalität steigert. Es sollte jedoch beachtet werden, dass diese Daten prinzipiell schwer vergleichbar und daher nur grobe Anhaltspunkte sind (für die Daten in Tabelle 6.2 gilt das gleiche). Es wirkt sich nicht nur das Geschick des Trainers, sondern – je nach verwendetem Sensor unterschiedlich stark – auch sein „Glück“ auf diese Zahlen aus. Die verwendete „Kinect for xBox 360“ liefert teilweise nur sehr instabile Skelettmodelle. Ein leistungsfähigerer Sensor würde das Training sicherlich

Trainings-beispiele	Vorhersagequalität	
	Bereich	Mittel ca.
4	< 10%	
8	10-60%	20%
12	20-80%	30%
16	20-85%	60%
11-58	75-90%	80%

Tabelle 6.1.: Vorhersagequalität in Abhängigkeit der Trainingsdatenmenge pro Schlüsselpose. Die letzte Zeile beschreibt das fertiggestellte Schlüsselposenrepertoire. Ab 75% gilt eine Schlüsselpose als erkannt.

Pose	Rate	Pose	Rate
Standing	9	Spread legs	9
Left arm out front	10	Right arm out front	9
Left hand up	10	Right hand up	10
Left arm out sideways	9	Right arm out sideways	10
Both arms out sideways	9	Hands up	10
Left arm to right shoulder	10	Right arm to left shoulder	10
Touch head with left hand	10	Touch head with right hand	10
Kick left	7	Kick right	6
Merkel	5	**Durchschnitt**	9

Tabelle 6.2.: Erkennungsraten der Schlüsselposen aus Abbildung 6.1 bei insgesamt 10 Wiederholungen pro Schlüsselpose

erleichtern und besonders die Rotationsinvarianz steigern. In jedem Fall würden stabilere Skelettmodelle den Bedarf für regelmäßiges „aufräumen“ der Trainingsdaten vermindern, da einfach weniger statistische Ausreißer auftreten dürften.

Je kleiner das zu trainierende Repertoire, desto leichter gestaltet sich das Training, womit natürlich auch der dazu nötige Zeitaufwand zunehmend stark abnimmt. Ein möglichst kleines und auf die jeweilige Anwendung spezialisiertes Schlüsselposenrepertoire zu bilden hat außerdem den Vorteil, dass das Training weniger präzise erfolgen muss (sofern sich die verwendeten Schlüsselposen stark unterscheiden) und der Benutzer dadurch einen deutlich größeren Spielraum beim Einnehmen einer Schlüsselpose hat. Dadurch dürfte sich die Erkennungsrate insgesamt und damit auch die Systemakzeptanz steigern.

Gerade mit Hinblick auf das durch den verwendeten Sensor bedingte Rauschen in den Trainingsdaten zeigen die Erkennungsraten, dass der für diese Arbeit gewählte Ansatz zur

Schlüsselposenerkennung für diese Aufgabe gut geeignet ist. Wie bereits erwähnt wird die Gestenerkennung hier nicht gesondert evaluiert, da sie letztlich mit den durch die Schlüsselposenerkennung bereitgestellten Daten arbeitet und selbst inhärent optimale Ergebnisse liefert.

6.2. Lokale Ressourcen

Bei Betrachtung der benötigten lokalen Ressourcen – im Wesentlichen benötigte CPU-, Arbeitsspeicher- und Festplatten-Kapazität zeichnet sich ein eher gemischtes Bild, dennoch sind die Ergebnisse durchaus sehr zufriedenstellend. Der von Java-Laufzeitumgebung sowie ggf. Microsoft Kinect SDK benötigte Festplattenspeicher wird bei der folgenden Betrachtung außer Acht gelassen.

libgrr selbst benötigt zur Wahrnehmung seiner Erkennungsaufgabe relativ wenig lokale Ressourcen: Das JAR-Paket belegt weniger als 3,5MB Festplatten- und rund 45MB Arbeitsspeicher. Als Referenz-System zur besonderen Illustration der Genügsamkeit wird ein Laptop mit 2GB Arbeitsspeicher und einer Core 2 Duo T5600 Dual-Core-CPU mit 1.83GHz Taktfrequenz aus dem Jahre 2007 verwendet (Betriebssystem ist Microsoft Windows 8.1 Pro). Die auf diesem System von *libgrr* erzeugte CPU-Last beträgt bei 1-Benutzer-Betrieb lediglich 3,7%. Der Einsatz auf deutlich schwächeren Plattformen ist also durchaus möglich.

Spürbar höhere Anforderungen an die CPU hat das *grrtool*, dennoch lässt sich damit auf dem o.g. System immer noch sehr gut arbeiten. Beim Anfertigen einer Aufnahme liegt die CPU-Last bei ca. 30%. Die ausführbare JAR-Datei belegt ca. 35MB Festplattenspeicher, enthält dabei allerdings auch die SWT-Bibliotheken für alle sechs unterstützten Plattformen. Der Arbeitsspeicherbedarf liegt wie bei *libgrr* bei ca. 45MB.

Die meisten Resourcen benötigt das Hilfsprogramm „SensorDataProvider“ zum Bereitstellen der Sensor-Daten, bzw. das von diesem Programm verwendete Microsoft Kinect SDK. Die CPU-Auslastung liegt hier zwischen 70 und 80 Prozent und der Arbeitsspeicherbedarf bei etwa 120MB. Das Programm selbst benötigt lediglich gut 0,5MB Festplattenspeicher, das Microsoft Kinect SDK benötigt jedoch einige Hundert MB.

Wenn auch nicht mehr ganz flüssig, so ist auf dem o.g. System immer noch ein Arbeiten möglich, wenn „SensorDataProvider“ und *grrtool* gleichzeitig gestartet sind. Die vom beschriebenen System bereitstellbaren Ressourcen markieren also den Grenzbereich, bis zu dem hinab sich bei Nutzung nur eines Rechners Schlüsselposen- und Gestenrepertoires mit Hilfe des *grrtool* aufbauen lassen. In Anbetracht des Alters dieses Systems ist dies eher als kleine Hürde,

und der Bedarf an lokalen Ressourcen damit als erfreulich gering zu werten – besonders, wenn die Sensor-Daten von einem anderen System bereitgestellt werden.

6.3. Netzwerk

Auch mit Hinblick auf die Netzwerkauslastung bei einer Verteilung des Gesamtsystems zeichnet sich ein insgesamt positives Bild. Der Datenstrom von bei *libgrr* eingehenen JSON-Nachrichten mit Sensor-Rohdaten gemäß Anhang C beläuft sich bei einem Benutzer auf ca. 50KB/s. Unter günstigen Bedingungen ist damit sogar das GSM-Netz mit EDGE als Medium geeignet, wenn auch hier durchaus mit „Frame Drops“ gerechnet werden muss.

Die von *libgrr* gesendeten Nachrichten – wenn die Bereitstellung aller Daten gewünscht ist – liegen je nach Größe des Schlüsselposenrepertoires auf einem ähnlichen Niveau. Ist jedoch nur die Übertragung von Schlüsselposen (ohne Wahrscheinlichkeitsverteilung) oder gar nur von erkannten Gesten gewünscht, so reduziert sich das Datenaufkommen deutlich.

Im Allgemeinen profitieren Anwendungen am ehesten von einer geringen Latenz des Netzwerks. Die nötige Kanalkapazität ist heutzutage in den meisten Festnetz- und auch vielen Mobilfunk-WANs verfügbar, in LANs ist die Situation noch besser. Wenn auch die auftretenden Datenmengen nicht ausgesprochen groß sind, so sind sie doch groß genug, um die Verteilungsmöglichkeiten des Gesamtsystems einzuschränken. In diesem Bereich wären Verbesserungen wünschenswert, glücklicher Weise aber auch sehr leicht realisierbar (siehe Abschnitt 7.2).

6.4. Zusammenfassung

Es wurde anhand eines vergleichsweise großen Schlüsselposenrepertoires gezeigt, dass das System grundsätzlich gut bis sehr gut funktioniert. Weitere Verbesserungen in der Erkennungsleistung ließen sich voraussichtlich sehr leicht mit einem besseren Sensor, z.B. die „Kinect for Windows“ oder „Kinect for Windows 2“ von Microsoft realisieren. Ungeachtet dieser Verbesserungsmöglichkeiten hat sich das System beweisen können.

Weiterhin positiv fällt der sehr geringe Bedarf an lokalen Ressourcen und damit die breiten Einsatzmöglichkeiten auch auf älterer Hardware auf. Lediglich das Auslesen der Rohdaten direkt vom Sensor stellt für leistungsschwächere Plattformen u.U. ein Problem dar. Spielraum für Verbesserungen bietet das eingesetzte Nachrichtenprotokoll für die Netzwerkübertragung, die meisten heutigen Netze – insbesondere LANs – sollten das entstehende Datenaufkommen jedoch problemlos handhaben können.

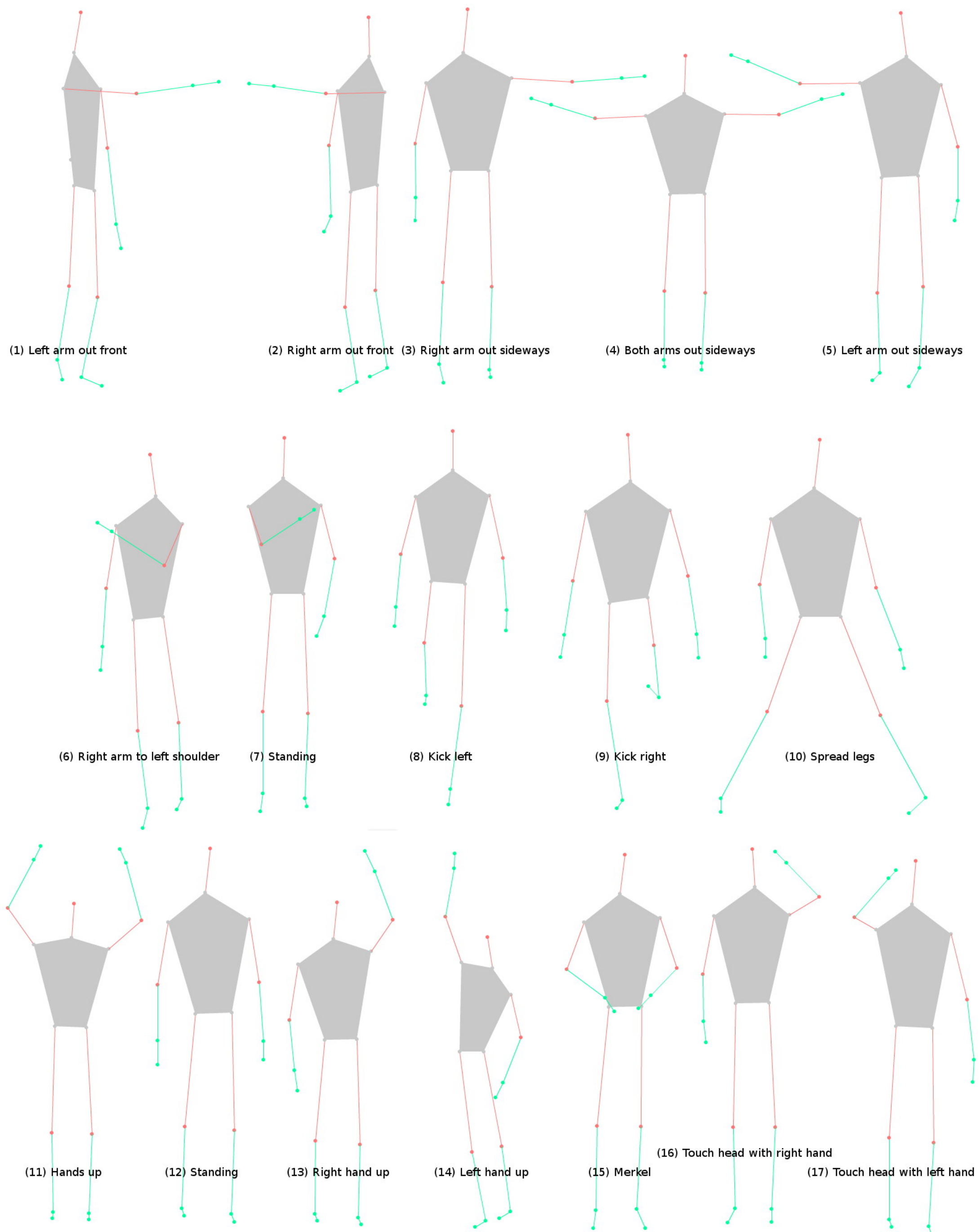


Abbildung 6.1.: Beispielhaftes Schlüsselposenrepertoire mit 17 Schlüsselposen

7. Schlussbetrachtungen

Die wesentlichen Leistungen dieser Arbeit sind eine Einführung in das Feld bzw. Problem der Posen- und Gestenerkennung und ein System zur Lösung dieses Problems auf Grundlage eines durch einen Sensor bereitgestellten Skelett-Modells. In Abschnitt 7.1 werden die Ergebnisse der einzelnen Kapitel dieser Arbeit zusammengefasst und in Abschnitt 7.2 ein Ausblick gegeben, der einige Ansatzpunkte für weiterführende Arbeiten vorstellt.

7.1. Fazit

Nach einer kurzen Einführung in das Themengebiet dieser Arbeit in Kapitel 1 wurde in Kapitel 2 zunächst klar gemacht, dass der eigentlichen Körperposen- und Gestenerkennung (und -aufzeichnung) die Bildung eines geeigneten Modells vorangeht, mit dem solche Erkennungsverfahren überhaupt arbeiten können. Zur Erkennung von Körperposen und -gesten bietet sich ein Skelett-Modell an. Für diesen Schritt der „Modellbildung" wurden verschiedene Ansätze vorgestellt und deutlich gemacht, dass für diese Arbeit nur optische, markierungslose, merkmalbasierte Ansätze in Betracht kommen. Andere Verfahren zur Bildung eines solchen Skelett-Modells kommen nicht in Frage, da sie die hier gestellten Anforderungen nicht erfüllen: Sie sind zu teuer, benötigen eine Präparation der Benutzer oder sie liefern überhaupt kein Modell, dessen Merkmale sich zur Posen- oder Gestenerkennung heranziehen ließen. Aber auch solche optisch, markierungslos und merkmalsbasiert arbeitenden Sensoren haben durchaus ihre Probleme (besonders sog. Okklusion, außerdem Störungen durch ungünstige Lichtverhältnisse), kommen dennoch als einzige für die Zwecke dieser Arbeit in Frage. Examplarisch wurde die Funktionsweise des hier eingesetzten „Kinect for xBox 360"-Sensors von Microsoft vorgestellt.

Eine Reihe verschiedener Verfahren können nun mit dem zur Verfügung stehenden Skelett-Modell arbeiten. Die Ansätze hinter diesen Verfahren sind vielfältig, aber auch hier kommen die meisten leider nicht in Frage: Sie sind schlecht skalierend, dem zeitlichen Verlauf einer Geste gegenüber nicht invariant, der Trainingsaufwand wäre zu groß oder aber die Anwender müssten komplexe Konfigurationen vornehmen, die je nach Anwendungsfall und im Wesentlichen durch „Trial-and-Error" zu bestimmen wären. Bei anderen Ansätzen wiederum konnte

mit vertretbarem Aufwand gar nicht eingeschätzt werden, ob sie sich nun eignen würden oder nicht.

Die Wahl für diese Arbeit fiel auf eine Gestenrepräsentation auf Grundlage von Schlüsselposen. Deren Klassifizierung erfolgt mittels einer Multi-Class-SVM und die Erkennung von Gesten, die nicht mehr als Schlüsselposensequenzen darstellen, wird auf eine effiziente Suche in einem Entscheidungswald zurückgeführt, der aus den Schlüsselposensequenzen aller bekannten Gesten aufgebaut wird.

Um den Überblick über dieses Themengebiet abzurunden wurden einige bereits verfügbare Softwarepakete vorgestellt. Am relevantesten sind hier „GestruePak" (Abschnitt 2.3.2) und „GDL Studio" (Abschnitt 2.3.1). Das für diese Arbeit angestrebte Maß an Einfachheit des Einsatzes bei gleichzeitiger Flexibilität wird von diesen Softwarepaketen jedoch nicht im gewünschten Umfang erreicht.

Kapitel 3 konkretisiert die Anforderungen im Rahmen dieser Arbeit aus unterschiedlichen Perspektiven, da verschiedene Benutzergruppen mit verschiedenen Facetten des Systems in Kontakt kommen. Die Anforderungen an die Erkennungsleistung, das Training und die Integration des hier entstehenden Softwarepakets in andere Hard- und Softwaretopologien werden also separat betrachtet. Im Kern steht dabei Einfachheit: Keinerlei Präparation der Benutzer o.ä., einfaches Training des Systems durch ein vielseitiges und auskunftsfreudiges Werkzeug und eine unkomplizierte Schnittstelle zur Integration der Erkennungsmechanismen in andere Anwendungen.

Der Entwurf des hier entwickelten Systems wurde in Kapitel 4 vorgestellt. Es wurde zunächst der grundsätzliche Workflow betrachtet und hieran verdeutlicht, dass die Wiedererkennung von Schlüsselposen und Gesten und deren Aufzeichnung zwei verschiedene Aufgaben sind, für die zwei verschiedene Softwarekomponenten verantwortlich sind. Anschließend wurde der zur Erfüllung der Anforderungen vorgesehene Funktionsumfang dieser Softwarekomponenten festgelegt und ein struktureller Entwurf des Gesamtsystems vorgestellt. Hier wird zum einen deutlich, wie sich die Einzelteile des Gesamtsystems strukturell von einander abgrenzen: Die Erkennungs-Komponente im Kern, darauf aufbauend Anwendungen wie das Aufnahme-Werkzeug selbst, die Austauschbarkeit der Wissenbasis und die Unabhängigkeit von einer konkreten Sensor-Schnittstelle. Zum anderen werden erstmals die einzelnen Module der Erkennungskomponente sichtbar, die letztendlich die interne Verarbeitungspipeline bilden.

Die Realisierung dieses Entwurfs wird detailliert in Kapitel 5 beschrieben. Neben der Beschreibung der konkreten Implementierung der Softwarekomponenten finden sich hier Informationen zu grundlegenden Designentscheidungen, die den Grundstein für die Einfachheit

und Flexibilität – hier vor allem aus Entwickler-Sicht – legen: Fassaden, IoC, Plattformunabhängigkeit und Netzwerkbasiertheit.

Kapitel 6 widmet sich einer Untersuchung der Leistungsfähigkeit des entstandenen Systems. Mit einer durchschnittlichen Erkennungsrate und Vorhersagequalität von ca. 90 respektive 80 Prozent liegt diese auf einem hohen Niveau. Der hierzu nötige Ressourcenaufwand ist so gering, dass auch ein Einsatz auf älterer oder generell schwächerer Hardware weitgehend problemlos möglich ist. Die meisten Ressourcen entfallen dabei auf die Anbindung des verwendeten Sensors, damit also nicht auf die Vorgänge im Kern des Systems.

Insgesamt belegt diese Arbeit die Leistungsfähigkeit des gewählten Ansatzes, die Erkennung ist bei moderatem Trainingsaufwand sehr zuverlässig. Das Werkzeug *grrtool* ermöglicht ein komfortables und effektives Training unter einer Reihe verschiedener Plattformen. Der eigentliche Kern des entwickelten Systems, *libgrr* erweist sich als sehr ressourcenschonend und bietet eine noch breitere Plattformunterstützung. Die diesem System beiliegenden Hilfsmittel erlauben es zudem das System schnell, einfach, in unterschiedlichsten Umfeldern und auf verschiedene Subsysteme verteilt einzusetzen.

7.2. Ausblick

Es gibt eine Reihe von Punkten, an denen sich an das in dieser Arbeit erreichte anknüpfen lässt. Das betrifft zum einen funktionale Aspekte, zum anderen eher technische. Diese Möglichkeiten zur allgemeinen Erweiterung und/oder Verbesserung des Systems werden im Folgenden kurz angerissen.

7.2.1. Funktionale Aspekte

In funktionaler Hinsicht bieten sich in erster Linie Arbeiten an der Unterstützung unterschiedlicher Modelle und eine flexiblere Auswahl der darin zu betrachtenden Merkmale an. Die folgenden Unterabschnitte erklären diese Aufgaben genauer.

Unterstützung verschiedener Modelle

Das System ist im Moment ausgelegt auf die Verarbeitung von Skelett-Modellen. Tatsächlich sind sehr wenige Programmteile anzupassen, um mit anderen Modellen, wie zum Beispiel einem detaillierten Hand-Modell, zu arbeiten.

Ideal, und mit moderatem Aufwand durchaus möglich, wäre eine Unterstützung beliebiger frei definierbarer Modelle, ohne dabei irgendwelche Programmteile anpassen zu müssen. Die Modellbeschreibung würde dann ein Teil der Wissensbasis werden, womit *libgrr* in Verbindung

mit *grrtool* zu einem universellen System zur Schlüsselposen- und Gestenerkennung in einem weiteren Sinne werden würde.

Wahl zu betrachtender Merkmale

Es ist denkbar, das Eintrainieren eines Schlüsselposen- und Gestenrepertoires zu vereinfachen und gleichzeitig die Erkennungsleistung zu verbessern – möglicherweise sogar bei deutlich erhöhter Schlüsselposenzahl – indem die zu betrachtenden Merkmale frei wählbar gemacht werden. Soll beispielsweise eine Wink-Geste trainiert werden, würde man diese aus einigen Schlüsselposen mit erhobener Hand in verschiedenen Positionen relativ zum Kopf zusammensetzen.

Wäre es nun möglich, dem System mitzuteilen, dass nur die zum Arm gehörenden Merkmale betrachtet werden sollen, dann hat das einen weitreichenden Vorteil: Andere Merkmale, wie z.B. die hier völlig uninteressante Knie-Stellung, würden gar nicht beachtet werden. Ein Winken in der Hocke würde genau wie ein Winken im Stehen erkannt werden können. Insbesondere der Trainingsaufwand hierfür wäre mit dem System in seiner jetzigen Form ungleich höher, da eben ein Winken in der Hocke und ein Winken im Stehen separat trainiert werden müssten.

Da die Feature-Vektoren im System damit allerdings von verschiedener Dimensionalität wären, bräuchte man für jede Dimensionszahl eine eigene Multi-Class-SVM. Jede Eingabe muss dann u.U. in eine Vielzahl von SVMs eingegeben werden. Wenngleich an diesem Punkt nicht klar ist, wie stark sich dies im Ressourcenbedarf niederschlagen würde ist die wohl eher problematische Frage, wie diese Einzelergebnisse zueinander in Beziehung zu setzen sind. Denn eine Pose mit kleiner Dimensionszahl kann durchaus auch Teil einer komplexeren Pose sein, wo dann ein Konflikt entstünde. Insbesondere die praktische Relevanz solcher Konflikte und ein sinnvoller Umgang mit ihnen müsste genau untersucht werden.

Datenaustausch

Es wäre ebenfalls wünschenswert Anwendern einen einfachen Import von Daten aus anderen Schlüsselposen- und Gestenrepertoires zu ermöglichen. Besonders im Zusammenhang mit Schlüsselposen können hier für die eingesetzte SVM allerdings Probleme entstehen, da sich alle Trainingsdaten gegenseitig beeinflussen. Es müsste untersucht werden, wie man diesen Problemen wirkungsvoll begegnen kann.

7.2.2. Technische Aspekte

Dieser Abschnitt befasst sich mit eher technischen Details, die verbessert werden könnten. Dabei geht es in erster Linie darum durch unterschiedliche Maßnahmen, die in den folgenden Unterabschnitten etwas genauer betrachtet werden, die Flexibilität zu erhöhen und damit den Einsatz insbesondere von *libgrr* für noch mehr Umgebungen zu ermöglichen.

Hardwareanbindung

Für den Fall, dass bei einer Anwendung der Sensor nur für die Schlüsselposen- und Gestenerkennug genutzt wird, kann zur Bereitstellung der Sensor-Daten der mitgelieferte „SensorDataProvider“ verwendet werden. In der Anwendung sind dann keinerlei Kenntnisse über den eingesetzten Sensor oder dessen API erforderlich.

Dieses Hilfsprogramm funktioniert aktuell nur mit dem Microsoft Kinect SDK in der Version 1.8 (Unterstützung für „Kinect for xBox 360“ und „Kinect for Windows“). Ein entsprechendes Hilfsprogramm zur Sensordaten-Bereitstellung für das Kinect SDK 2 würde das Sytem abrunden. Wünschenswert wären entsprechende Programme auch für evtl. weitere am Markt befindliche Sensoren.

Nachrichtenformat

Das momentan eingesetzte JSON-Nachrichtenformat besticht am ehesten durch Einfachheit und Menschenlesbarkeit. Um die Möglichkeiten zur Verteilung weiter zu erhöhen und langsame Netze als Kommunikationskanäle nutzbar zu machen ist jedoch ein effizienteres Format notwendig. Beim Blick auf die Sensor-Rohdaten käme ein triviales Binärformat bereits mit weniger als 7KB pro Sekunde und Benutzer aus (30fps, 19 Merkmale mit je 3 Koordinaten in 32b-Floats, zusätzlich ein Byte als Skelett-ID). Dieser Wert lässt sich sicherlich noch unterbieten. Da die Einfachheit des aktuellen JSON-Formats für einige Anwendungen jedoch wichtiger als eine maximale Effizienz sein dürfte wäre eine freie Wählbarkeit des verwendeten Protokolls wünschenswert.

Datenbankformat

Hilfreich für einen Datenaustausch wie in Abschnitt 7.2.1 beschrieben, wie auch ganz allgemein für die Datenverwaltung wäre die Einführung eines flexibleren Datenbankformats (SQLite, XML, JSON, …). Der Bestand einer Wissensbasis an Schüsselposen, Trainingsdaten und Gesten könnte so auch außerhalb von Java-Programmen bearbeitet werden.

Unterstützung weiterer Plattformen

Gerade weil *libgrr* bereits sehr wenig Ressourcen benötigt ist es überaus erstrebenswert die Bibliothek auf ihre Lauffähigkeit unter anderen Plattformen wie z.B. Android und J2ME/phoneME hin zu untersuchen und ggf. anzupassen. Der Aufwand ist wahrscheinlich relativ gering und würde eine Vielfalt weiterer Geräte wie Tablets, Telefone und Settop-boxen in die Liste unterstützter Plattformen einreihen.

A. Events

This is an alphabetically ordered list of all events that are triggered in *libgrr*. It describes when they are triggered and what data is passed along with each.

A.1. BufferUpdated

Brief Description: Change of buffer contents

Origin: `GestureDetector`

Trigger: `addKeyPoseToBuffer`
`clearBuffers`
`keyPoseDetected`
`loadKeyPoseSequence`

Parameters: **`String`**

The skeleton ID of the buffer that was updated

`List<Integer>`

The current buffer contents

Description

This event is triggered when the buffere assigned to a specific skeleton ID is changed in any way. That happens when the `GestureDetector`'s buffers are discarded or when a key pose is added to that buffer. In case the buffer was discarded the second parameter will be `NULL`. That is because the buffers are not just emptied, but they actually cease to exist until another key pose is added.

A.2. GestureDetected

Brief Description: Detection of a gesture

Origin: `GestureDetector`

Trigger: `addKeyPoseToBuffer`
`keyPoseDetected`
`loadKeyPoseSequence`

Parameters: **`String`**

ID of the skeleton that performed the detected gesture

`Gesture`

The detected gesture

Description

This event is triggered when the key poses most recently added to the buffer for a specific skeleton ID make up a gesture.

A.3. GestureDetectorTrainingComplete

Brief Description: Training of a `GestureDetector` completed

Origin: `GestureDetector`

Trigger: `train`

Parameters: **`GestureDatabase`**

The database used for training

A.4. InputFrameProcessed

Brief Description: Processing of a complete input frame finished

Origin: `PoseProcessor`

Trigger: `update`

Parameters: **Map<String, Map<FeatureType, Feature>>**

Mapping from skeleton IDs to the corresponding skeleton

Map<String, TorsoBasis>

Mapping from skeleton IDs to the computed torso bases

Description

This event is triggered after complete processing of an input frame. If this event is triggered, then at least one `SkeletonFrameReceived` event will have been triggerd beforehand for a skeleton frame contained in that input frame. The `InputFrameProcessed` event will not be triggered if not at least one usabe skeleton was found in the input frame. However, if a usabe skeleton was found then the first parameter will contain **all** skeletons, no matter if each one is usable or not. Whether a specific skeleton is usable or not can be determined with the second parameter: In case a a skeleton can't be used the corresponding torso basis will be NULL. Those skeletons in the input frame that get assigned NULL as torso basis are completely unprocessed and contain only the original raw data provided by the sensor.

A.5. KeyPoseDetected

Brief Description: Detection of a key pose

Origin: `KeyPoseDetector`

Trigger: `detectKeyPoses`
`inputFrameProcessed`

Parameters: **`String`**
ID of the user/sekelton on which the key pose was detected

`KeyPose`
The detected key pose

`Map<Integer, Float>`
A map detailing the probabilities for each known key pose ID

Description

This event is triggered as soon as a key pose is detected. If an input frame contains multiple skeletons this event can by triggered more than once for that input frame. It is triggered immediately after key pose detection. It is therefore possible that not all skeletons in the input frame have been processed yet at that time.

A.6. KeyPoseDetectorTrainingComplete

Brief Description: Training of a `KeyPoseDetector` finished

Origin: `KeyPoseDetector`

Trigger: `train`

Parameters: **`KeyPoseDatabase`**

The database used for training

A.7. SkeletonFrameReceived

Brief Description: Succesful processing of a skeleton frame

Origin: `PoseProcessor`

Trigger: `update`

Parameters: **`String`**

ID of the processed skeleton

`Map<FeatureType, Feature>`

The processed skeleton data

`TorsoBasis`

The computed torso basis

`DenseMatrix64F`

The change of base matrix to change the base of the feature coordinates from the sensors global coordinate system to this skeleton's local coordinate system that is based on the torso basis.

Description

This event is triggered when a skeleton was successfully processed. It can be triggered multiple times per input frame and always before a `InputFrameProcessed` event. Oberservers of this event can use the data passed along through the parameters to get a hold of every properly processed skeleton. It can be, for example, drawn to the screen, correlated with other sensor coordinates or localized in the space covered by the sensor. This event is mostly useful if you want to do more than just key pose and gesture recognition with the sensor.

B. Quick Start Guide

This is the user guide for *libgrr* and the set of tools built around it. It gives a brief introduction and a quick start guide, followed by a more detailed, "full-featured" user guide.

This toolkit is made up of *libgrr* itself, which sits at the core and does the actual key pose and gesture recognition. *grrtool* is built on top of that and can be used to create, manage and debug a knowledge bases for use with *libgrr*. The "SensorDataProvider" tool can be used to obtain the sensor data and feed it into *libgrr*. If you don't intend on doing anything with the sensor by yourself, you have access to a Microsoft Windows system and you are using a "Kinect for xBox 360" or "Kinect for Windows" sensor, then this should come in handy. For details keep reading.

B.1. System Requirements

The different parts of this toolkit have different system requirements. Unfortunately as of now it is not exactly clear what the absolute minimum system requirements are. Refer to the section covering the parts you intend to use.

B.1.1. libGRR

Essentially all you need is a system capable of running OpenJDK or Oracle Java. *libgrr* is built against Java 7, so for now that is the minimum required version. This may change in the future as it is likely that *libgrr* could also be run on lower versions.

As for computing power: The lower end is hard to estimate. But it does run perfectly fine on an Intel Core 2 Duo T5600 with 2GB of RAM or even a Cubieboard 2 (Dual Core ARM Cortex A7 at 1GHz with 1GB of RAM).

On Windows 8.1 64bit running an Oracle Java 8 JRE it uses about 45MB of memory. It requires less than 4MB of disk space.

B.1.2. GRRTool

This is a Java SWT application and was tested using Oracle Java 7 and 8 under

- Windows 7 32bit
- Windows 7 64bit
- Windows 8.1 64bit
- Ubuntu 14.04 64bit
- MacOS X 64bit

While not perfectly smoothly, it still does run well on above-mentioned Core 2 Duo system. On Windows 8.1 64bit running an Oracle Java 8 JRE it uses about 45MB of memory. It requires about 60MB of disk space.

B.1.3. SensorDataProvider

This will run only on systems with the Microsoft Kinect SDK 1.8 installed. It's also the SDK that imposes this program's minimum system requirements. Please refer to Microsoft for details. While noteably less smooth, still *grrtool* remains usable alongside the SensorDataProvider on the above-mentioned Core 2 Duo system. Anything much slower will not be able to deliver an uninterrupted data stream from a Kinect sensor, let alone do that *and* run *grrtool*.

B.2. Getting started

This quick start guide is intended for experienced computer users and software developers. You are expected to know your way around a computer and maybe even an IDE – depending on what you want to do. If you found section B.1 full of "technical mumbo-jumbo" chances are this is not for you. In that case please refer to the documentation that will be bundled with the software eventually.

B.2.1. SensorDataProvider

Use this program to get the skeletal data available form the Kinect sensor and publish it on port 11000. You'll need to install it on your system. To do that, run the "setup.exe" in the "SensorDataProvider" directory of the distribution zip file. The program should start after install. If not, search for "SensorDataProvider".

Afterwards *libgrr* and *grrtool* can connect to the SensorDataProvider and get the data they need. Make sure you allow network access to the program. To verify that it works, you can simply navigate to "localhost:11000" in a web browser or connect via telnet. Once the connection

is established, SensorDataProvider will notify you on the console ("Client connection accepted. Client ID is x").

Make sure you have the Microsoft Kinect SDK 1.8 installad. Earlier oder later versions will not work!

B.2.2. GRRTool

This section gives pointers for running/start-up and for basic usage.

Running *grrtool*

To launch *grrtool* simply double-click "grrtool.jar" (yes, you may also run it from a console…). A tiny splash screen should come up while loading. If not, you'll need to run it from the console and check for error messages. The working directory should be the same as the location of the "database.grrdb".

grrtool accepts a number of optional command line parameters. Those are a path to the database to use as well as t he host and port from where the sensor data should be read. For details run *grrtool*. With the `-h` parameter (`java -jar grrtool.jar -h`). If SensorDataProvider is running on the same system and you plan on using the default database, then you won't need any command line parameters.

Basic usage

The UI is divided into 3 Tabs. "Record" lets you create a new recording. From there you can define new key poses and gestures as well as add training samples. "Manage, Inspect & Deploy" allows you to do just what the tab says. Check the section below for details. "Test" let's you test your current database or choose one to test. Basically it supports multiple users und has larger fonts so you can still read the important things while moving around in front of the sensor.

Record

On the black canvas you'll see a skeleton model, as soon as one is available from the sensor (if no recording is loaded – but that's the case after *grrtool* has just started). Move around in front of your sensor to make sure the model is drawn and the sensor connection works.

The red and green table shows the contents of the current feature vector that resembles the current pose. Below that is a list of all known key poses with their number of training samples, computed probability for the current frame and an "Add"-Button. Below that is the

current buffer contents. Gestures made up of key poses will be matched against such a key pose sequence from the gesture detector buffer.

If you want to add a new key pose, add samples for existing key poses or create a new gesture, you'll need a recording. Press "Rec" (on the bottom of the screen) in order to create one. Once pressed, do your thing in front of the sensor. When finished, click the "Stop"-Button. There is probably garbage at the beginning and the end of your recording. You can remove that using the "Truncate left" and Truncate right" buttons.

If the database you loaded already contains key poses then maybe you can see a key pose sequence in the buffer contents table. Double-click an entry to jump to the place in the recording, where the key pose was detected.

In order to create a key pose, press the "Set key pose" button. This will open a dialog where you can enter a name for the key pose. Using the slider on the bottom of the screen you can navigate through the recording. If you want to add a specific frame as a training sample for an existing key pose just klick the "Add" button next to that key pose in the table showing the probability distribution for the curent frame. Duplicate samples won't be added by the way. After every change to the key pose database the recording will be reloaded and reevaluated. As that influences key pose detection the buffer contents table is likely to change.

If you want to create a gesture, then you must record it first. After recording it, make sure the key poses you wish the gesture to be made up of are found in the key pose buffer and are in the correct order. Remove any unwanted key poses from the beginning or end of the recording using the "Truncate right" and "Truncate left" buttons. When you're done, click the "Save Gesture" button, enter a name, and you're set.

Manage, Inspect & Deploy

The upper part of the view is for managing key poses, the lower part is for gestures.

On the to left you can see a list of all known gestures and the number of training samples for each. Select one, and on the right to it you will see the list of training samples. To the right of that you will notice that a table showing all kinds of statistics has been populated. If you select a training sample you can see a sketch of it in the upper right corner of the window and the corresponding table columns for the θ and ϕ values will be populated.

The table column names should be mostly self-explanatory. $s(\theta)$ denotes the standard deviation for θ. You can remove training samples and (if they're not part of a gesture) whole key poses using the corresponding buttons. Under the pose preview canvas in the upper right part of the window you can tick the "Show all samples" checkbox to display all training samples simultaneously.

The sample list may be displaying a number in square brackets in front of some training samples. Those samples contain a feature which value is *these* degrees outside of the value span from mean value minus standard deviation and mean value plus standard deviation. Meaning, these are good candidates to examine if you feel your database is performing worse than it should.

In the bottom view you'll find a list of known gestures on the left hand side. Click a gesture and you'll see pictures of the key poses sequence that it's made up of. Here you can delete gestures or select a couple of gestures and export them along with all key poses to a different database. This way you can deploy a certain gesture subset for use in some application.

Notice how the highlighting of table items changes when you hover over certain other items or click them. These highlightings mark dependencies between those items.

Test

Use this view to examine the performance of a database. You can either load a database or use *grrtool*s current default database. Toggle the "Test"-Button to start or stop a test (a test will also stop if you switch to another tab). For each detected user a new test view is created. It paints the current sekeleton model for each user, shows key pose probabilities and gesture detection info.

A key pose is displayed when it's probability exceeds 50% and after it drops below that for a little while longer. If the current key pose sequence in the buffer for a specific test view contains key poses that match the beginning of a known gesture, then that gesture is highlighted in light green. When a complete gesture is detected it is highlighted by a stronger shade of green and the counter to the right of the gesture's name increases.

B.2.3. libGRR

If you plan to use *libgrr* as a library with another program, add "libgrr.jar" to your project in whichever way your IDE wants you to and check out the API-Documentation of the `com.libgrr.LibGRR` class. This will get you started.

If you plan to use *libgrr* in stand-alone mode, then you can just double click "libgrr.jar". This will use "database.grrdb" as a knowledge base, connect to a SensorDataProvider on "localhost:11000" and publish key pose probability distributions, detected key poses and detected gestures to port 11001.

If you want to change the data being made available by *libgrr*, use another database, connect to a SensorDataProvider on another host or port or want to make the data provided by *libgrr*

available on another port, then you can do so with command line parameters. Run "libgrr.jar" with the "-h" parameter for details on how to do that.

The message format is specified in appendix C.

C. Message Format

This Chapter describes the message format used by *libgrr* for receiving sensor data and publishing computed probability distributions or detection events. All messages are JSON-encoded. Their structure is visualized as railroad diagram and described as EBNF. The graphics were created from EBNF using "Railroad Diagram Generatorät "http://bottlecaps.de/rr/ui".

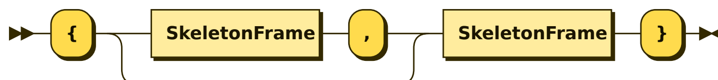
C.1. Incoming Messages

These messages can be received by *libgrr*.

C.1.1. Input Frame

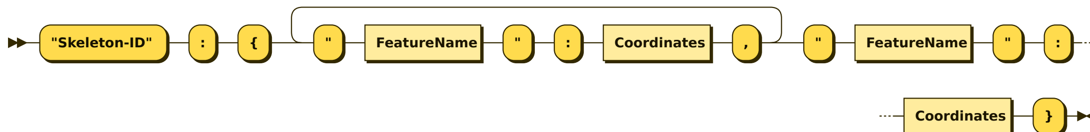
All productions of the "FeatureType"non-terminal must be present.

InputFrame



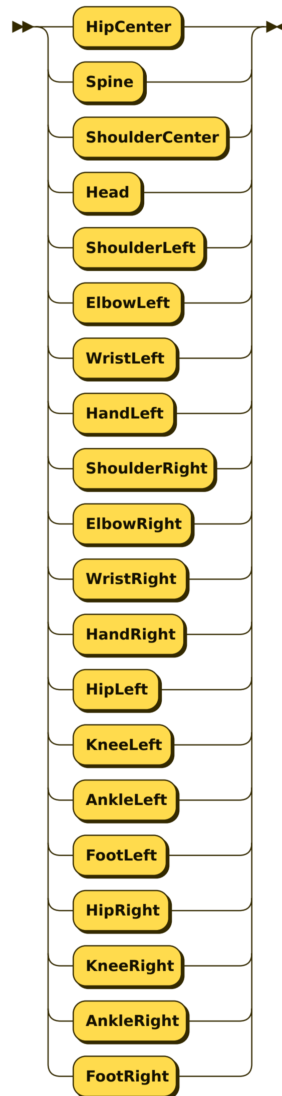

```
InputFrame
         ::= '{' ( SkeletonFrame ',' )? SkeletonFrame '}'
```

SkeletonFrame



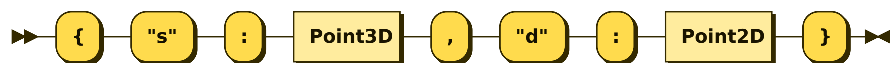

```
SkeletonFrame
         ::= '"Skeleton-ID"' ':' '{'
               ( '"' FeatureName '"' ':' Coordinates ',' )+ '"' FeatureName '"' ':' Coordinates
             '}'
```

FeatureName



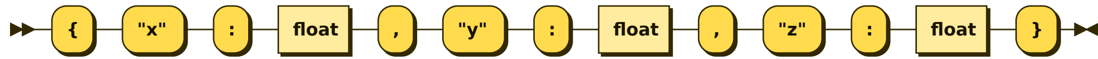

```
FeatureName
         ::= 'HipCenter'
           | 'Spine' | 'ShoulderCenter' | 'Head'
           | 'ShoulderLeft'  | 'ElbowLeft' | 'WristLeft' | 'HandLeft'
           | 'ShoulderRight' | 'ElbowRight' | 'WristRight' | 'HandRight'
           | 'HipLeft' | 'KneeLeft' | 'AnkleLeft' | 'FootLeft'
           | 'HipRight' | 'KneeRight' | 'AnkleRight' | 'FootRight'
```

FeatureName



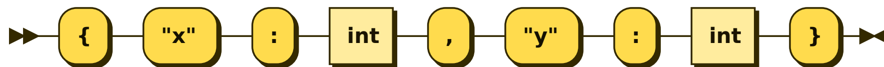
```
Coordinates
         ::= '{' '"s"' ':' Point3D ',' '"d"' ':' Point2D '}'
```

Point3D




```
Point3D  ::= '{' '"x"' ':' float ',' '"y"' ':' float ',' '"z"' ':' float '}'
```

Point2D




```
Point2D  ::= '{' '"x"' ':' int ',' '"y"' ':' int '}'
```

Example

```
{
  "3_142": {
    "HipCenter": {
      "s": {
        "x": 0,
        "y": 0,
        "z": 0
      },
      "d": {
        "x": 0,
        "y": 0
      }
    },

    ...,

    "FootRight": {
      "s": {
        "x": -0.1434303,
        "y": -0.2143254,
        "z": 0.7127159
      },
      "d": {
        "x": 205,
        "y": 412
      }
    }
  }
}
```

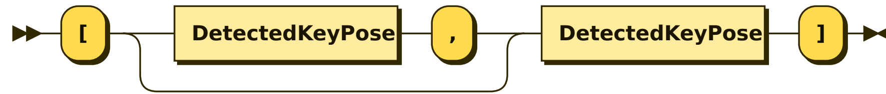
C.2. Outgoing Messages

These messages can be published by *libgrr*.

C.2.1. Detected Key Poses

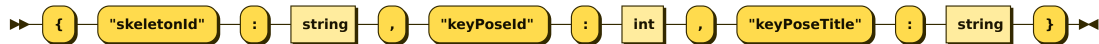
This message lists all key poses detected in an input frame.

DetectedKeyPoses




```
DetectedKeyPoses
         ::= '[' ( DetectedKeyPose ',' )? DetectedKeyPose ']'
```

DetectedKeyPose




```
DetectedKeyPose
         ::= '{' '"skeletonId"' ':' string ',' '"keyPoseId"' ':' int ',' '"keyPoseTitle"' ':' string '}'
```

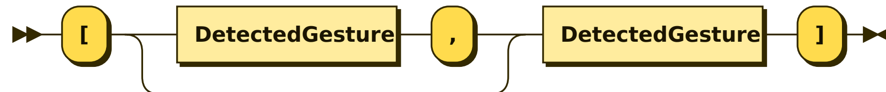
Example

```
[
  {
    "skeletonId": "4_1115",
    "keyPoseId": 913940646,
    "keyPoseTitle": "Standing"
  }
]
```

C.2.2. Detected Gestures

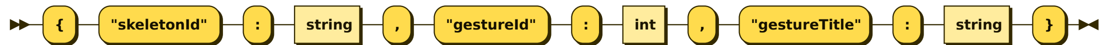
This message lists all gestures detected in an input frame.

DetectedGestures




```
DetectedGestures
         ::= '[' ( DetectedGesture ',' )? DetectedGesture ']'
```

DetectedGesture




```
DetectedGesture
         ::= '{' '"skeletonId"' ':' string ',' '"gestureId"' ':' int ',' '"gestureTitle"' ':' string '}'
```

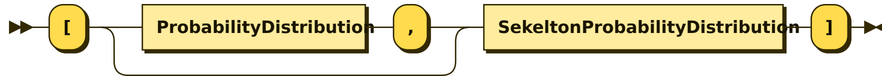
Example

```
[
  {
    "skeletonId": "4_1115",
    "gestureId": -1874392857,
    "gestureTitle": "Flap arms"
  }
]
```

C.2.3. Key Pose Probability Distributions

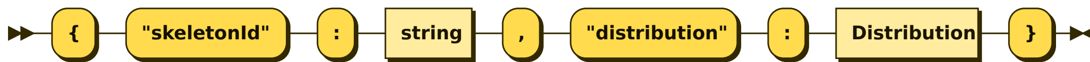
This message list the probability distribution across all known key poses for all skeletons in an input frame.

DetectedGestures



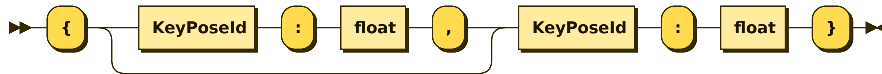

```
ProbabilityDistributions
         ::= '[' ( ProbabilityDistribution ',' )? SekeltonProbabilityDistribution ']'
```

DetectedGesture



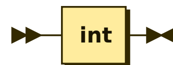

```
SekeltonProbabilityDistribution
         ::= '{' '"skeletonId"' ':' string ',' '"distribution"' ':' Distribution '}'
```

DetectedGestures




```
Distribution
         ::= '{' ( KeyPoseId ':' float ',' )? KeyPoseId ':' float '}'
```

DetectedGestures




```
KeyPoseId
         ::= int
```

Example


```
[
  {
    "skeletonId": "4_1115",
    "probabilities": {
      "727579472": 0.0035984449,
      "1927076336": 0.0042345696,
      "135613617": 0.003186368,
      "1729394097": 0.0068753175,
      "710288437": 0.011813179,
      "564275124": 0.02761214,
      "372765130": 0.002373332,
      "404517727": 0.014077719,
      "1915052027": 0.003371085,
      "1878522774": 0.0061976667,
      "913940646": 0.8687371,
```

```
      "945504367": 0.0017885026,
      "619709045": 0.0062899785,
      "569548502": 0.0020644672,
      "1731955952": 0.012333912,
      "1639393719": 0.022318456,
      "1993160633": 0.003127729
    }
  }
]
```

D. Genehmigungen und Lizenzen

Dieser Anhang enthält Genehmigungen zur Verwendung von Grafiken und Lizenzvereinbarungen für die im Rahmen der Arbeit entwickelte Software.

D.1. Genehmigung zur Verwendung von Grafiken

D.1.1. DDTW-Grafiken

```
Dear Mr. Zajonz:

SIAM is happy to give permission to reprint figures 5 and 6 from
the proceedings paper mentioned below.  In the credit line,
please cite the complete bibliographic reference to the original
paper.

Thank you for your interest in our publications.

Sincerely,

Kelly Thomas
Managing Editor, SIAM




-----Original Message-----
From: Michael Zajonz [mailto:michael.zajonz@informatik.haw-hamburg.de]
Sent: Tuesday, November 11, 2014 3:45 PM
To: Joanne M. Cassetti
Subject: Use of Graphics

Hello Ms Cassetti,

I hope you can help me with a licensing question:
```

```
I would like to use two graphics from the paper found under

http://epubs.siam.org/doi/abs/10.1137/1.9781611972719.1

for my bachelor thesis. I'm a student of computer science at
the Hamburg University of Applied Science in Hamburg, Germany.
As far as I know my department isn't a subscriber of SIAM.
On the other hand the document is publicly available.

May I reuse two of the published graphics for my thesis?

I have attached the relevant excerpt of my thesis so you can
take a look at the usage. The citation used for "Keogh und
Pazzani 2001" is

  "Keogh, Eamonn J. ; Pazzani, Michael J.: Derivative Dynamic Time Warping.
   In: SDM Bd. 1 SIAM (Veranst.), 2001, S. 5-7"

and is taken from Google Scholar.

Thank you for your help,

kind regards

Michael Zajonz
```

D.2. Lizenzbedingungen der entwickelten Software

D.2.1. libgrr

Copyright 2014 Michael Zajonz <michaelzajonz@web.de>

libgrr is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

libgrr is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

A copy of the GNU Affero General Public can be found in the file COPYING or online at <http://www.gnu.org/licenses/>.

Additional permission under GNU AGPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with any of

- com.ibm.icu
- org.eclipse.core.commands
- org.eclipse.core.databinding.beans
- org.eclipse.core.databinding.observable
- org.eclipse.core.databinding.property
- org.eclipse.core.databinding
- org.eclipse.core.jobs
- org.eclipse.core.runtime
- org.eclipse.equinox.common
- org.eclipse.equinox.registry
- org.eclipse.jdt.internal.jarinjarloader
- org.eclipse.jface.databinding
- org.eclipse.jface.text
- org.eclipse.jface
- org.eclipse.osgi
- org.eclipse.swt.cocoa.macosx.x86
- org.eclipse.swt.cocoa.macosx.x86_64
- org.eclipse.swt.gtk.linux.x86
- org.eclipse.swt.gtk.linux.x86_64
- org.eclipse.swt.win32.win32.x86
- org.eclipse.swt.win32.win32.x86_64

- org.eclipse.text
- org.eclipse.ui.forms
- org.eclipse.ui.workbench
- org.eclipse.wb.swt.SWTResourceManager

(or a modified version of any of these libraries), containing parts covered by the terms of Eclipse Public License Version 1.0, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with

- XULRunner 1.9 and/or
- Cairo Binding

(or a modified version of these libraries), containing parts covered by the terms of Mozilla Public License Version 1.1, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with Libtabe (or a modified version of that library), containing parts covered by the terms contained in IPADIC.txt, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with Unicode Data Files and Software (or a modified version of that library),containing parts covered by the terms contained in UNICODE.txt, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with ICU4J (or a modified version of that library), containing parts covered by the terms of ICU License - ICU 1.8.1, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

libgrr makes use of software licenced under the Apache Licence 2.0. The licence text is available in the file APACHE. The Apache Licence 2.0 applies to:

- org.ejml.example.PrincipalComponentAnalysis
- ejml-0.26
- json_simple-1.1
- guava-18.0
- log4j

libgrr makes use of "libsvm" which is licenced under the modified BSD Licence. The licence text can be found in the COPYRIGHT.libsvm file.

D.2.2. grrtool

Copyright 2014 Michael Zajonz <michaelzajonz@web.de>

grrtool is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

grrtool is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

A copy of the GNU General Public can be found in the file COPYING or online at <http://www.gnu.org/licenses/>.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with any of

- com.ibm.icu
- org.eclipse.core.commands
- org.eclipse.core.databinding.beans
- org.eclipse.core.databinding.observable
- org.eclipse.core.databinding.property
- org.eclipse.core.databinding
- org.eclipse.core.jobs
- org.eclipse.core.runtime

- org.eclipse.equinox.common
- org.eclipse.equinox.registry
- org.eclipse.jdt.internal.jarinjarloader
- org.eclipse.jface.databinding
- org.eclipse.jface.text
- org.eclipse.jface
- org.eclipse.osgi
- org.eclipse.swt.cocoa.macosx.x86
- org.eclipse.swt.cocoa.macosx.x86_64
- org.eclipse.swt.gtk.linux.x86
- org.eclipse.swt.gtk.linux.x86_64
- org.eclipse.swt.win32.win32.x86
- org.eclipse.swt.win32.win32.x86_64
- org.eclipse.text
- org.eclipse.ui.forms
- org.eclipse.ui.workbench
- org.eclipse.wb.swt.SWTResourceManager

(or a modified version of any of these libraries), containing parts covered by the terms of Eclipse Public License Version 1.0, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with

- XULRunner 1.9 and/or
- Cairo Binding

(or a modified version of these libraries), containing parts covered by the terms of Mozilla Public License Version 1.1, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with Libtabe (or a modified version of that library), containing parts covered by the terms contained in IPADIC.txt, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with Unicode Data Files and Software (or a modified version of that library),containing parts covered by the terms contained in UNICODE.txt, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

If you modify this Program, or any covered work, by linking or combining it with ICU4J (or a modified version of that library), containing parts covered by the terms of ICU License - ICU 1.8.1, the licensors of this Program grant you additional permission to convey the resulting work. Corresponding Source for a non-source form of such a combination shall include the source code of the covered work.

Literaturverzeichnis

[Al-Naymat u. a. 2009] Al-Naymat, Ghazi ; Chawla, Sanjay ; Taheri, Javid: SparseDTW: A novel approach to speed up dynamic time warping. In: *Proceedings of the Eighth Australasian Data Mining Conference-Volume 101* Australian Computer Society, Inc. (Veranst.), 2009, S. 117–127

[Anthony und Wobbrock 2010] Anthony, Lisa ; Wobbrock, Jacob O.: A lightweight multistroke recognizer for user interface prototypes. In: *Proceedings of Graphics Interface 2010* Canadian Information Processing Society (Veranst.), 2010, S. 245–252

[Babu u. a.] Babu, Sree Shankar S. ; Jaiswal, Prakhar ; Esfahani, Ehsan T. ; Rai, Rahul: SKETCHING IN AIR: A SINGLE STROKE CLASSIFICATION FRAMEWORK.

[Chang und Lin 2011] Chang, Chih-Chung ; Lin, Chih-Jen: LIBSVM: A library for support vector machines. In: *ACM Transactions on Intelligent Systems and Technology* 2 (2011), S. 27:1–27:27. – Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[Chen u. a. 2003] Chen, Feng-Sheng ; Fu, Chih-Ming ; Huang, Chung-Lin: Hand gesture recognition using a real-time tracking method and hidden Markov models. In: *Image and Vision Computing* 21 (2003), Nr. 8, S. 745–758

[Cortes und Vapnik 1995] Cortes, Corinna ; Vapnik, Vladimir: Support-vector networks. In: *Machine learning* 20 (1995), Nr. 3, S. 273–297

[Cross u. a. 1995] Cross, Simon S. ; Harrison, Robert F. ; Kennedy, R L.: Introduction to neural networks. In: *The Lancet* 346 (1995), Nr. 8982, S. 1075–1079

[Eisler 2012] Eisler, Craig: *Near Mode: What it is (and isn't)*. 2012. – URL http://blogs.msdn.com/b/kinectforwindows/archive/2012/01/20/near-mode-what-it-is-and-isn-t.aspx. – Zugriffsdatum: 02.08.2014

[Elliott u. a. 1994] Elliott, Robert J. ; Aggoun, Lakhdar ; Moore, John B.: *Hidden Markov Models*. Springer, 1994

[Franklin a] Franklin, Carl: *GesturePak*. – URL http://www.franklins.net/gesturepak.aspx. – Zugriffsdatum: 06.09.2014

[Franklin b] Franklin, Carl: *GESTUREPAK 2.0 ALPHA*. – URL http://carlfranklin.net/blog/2013/12/30/gesturepak-20-alpha.html. – Zugriffsdatum: 06.09.2014

[Franklin c] Franklin, Carl: *How GesturePak Works*. – URL http://www.franklins.net/GesturePakHow.aspx. – Zugriffsdatum: 06.09.2014

[Hachaj und Ogiela] Hachaj, Tomasz ; Ogiela, Marek R.: *Gesture Description Language Studio v1.1*. – URL http://www.cci.up.krakow.pl/gdl/#download. – Zugriffsdatum: 06.09.2014

[Hachaj und Ogiela 2014] Hachaj, Tomasz ; Ogiela, Marek R.: Rule-based approach to recognizing human body poses and gestures in real time. In: *Multimedia Systems* 20 (2014), Nr. 1, S. 81–99

[Johansson 1973] Johansson, Gunnar: Visual perception of biological motion and a model for its analysis. In: *Perception & psychophysics* 14 (1973), Nr. 2, S. 201–211

[Juang und Rabiner 1990] Juang, Biing-Hwang ; Rabiner, Lawrence: The segmental K-means algorithm for estimating parameters of hidden Markov models. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 38 (1990), Nr. 9, S. 1639–1641

[Kanungo u. a. 2002] Kanungo, Tapas ; Mount, David M. ; Netanyahu, Nathan S. ; Piatko, Christine D. ; Silverman, Ruth ; Wu, Angela Y.: An efficient k-means clustering algorithm: Analysis and implementation. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24 (2002), Nr. 7, S. 881–892

[Keogh und Pazzani 2001] Keogh, Eamonn J. ; Pazzani, Michael J.: Derivative Dynamic Time Warping. In: *SDM* Bd. 1 SIAM (Veranst.), 2001, S. 5–7

[Khreich u. a. 2012] Khreich, Wael ; Granger, Eric ; Miri, Ali ; Sabourin, Robert: A survey of techniques for incremental learning of HMM parameters. In: *Information Sciences* 197 (2012), S. 105–130

[Kolozs 1998] Kolozs, James: *DESIGN OF AN EXOSKELETAL HUMAN MOTION CAPTURE SYSTEM (SENSUIT™)*, The University of Utah, Dissertation, 1998

[Kota u. a. 2009] Kota, Solomon R. ; Raheja, JL ; Gupta, Ashutosh ; Rathi, Archana ; Sharma, Shashikant: Principal Component analysis for Gesture Recognition Using SystemC. In: *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom'09. International Conference on* IEEE (Veranst.), 2009, S. 732–737

[Krogh u. a. 2001] Krogh, Anders ; Larsson, BjoÈrn ; Von Heijne, Gunnar ; Sonnhammer, Erik L.: Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes. In: *Journal of molecular biology* 305 (2001), Nr. 3, S. 567–580

[Landay und Myers 2001] Landay, James A. ; Myers, Brad A.: Sketching interfaces: Toward more human interface design. In: *Computer* 34 (2001), Nr. 3, S. 56–64

[Lang u. a. 1990] Lang, Kevin J. ; Waibel, Alex H. ; Hinton, Geoffrey E.: A time-delay neural network architecture for isolated word recognition. In: *Neural networks* 3 (1990), Nr. 1, S. 23–43

[Leap Motion 2014] Leap Motion: *Leap Motion | 3D Motion and Gesture Control for PC & Mac*. 2014. – URL https://www.leapmotion.com/product. – Zugriffsdatum: 02.08.2014

[Microsoft Corporation] Microsoft Corporation: *Kinect Sensor*. – URL http://msdn.microsoft.com/en-us/library/hh438998.aspx. – Zugriffsdatum: 02.08.2014

[Miranda u. a. 2014] Miranda, Leandro ; Vieira, Thales ; Martínez, Dimas ; Lewiner, Thomas ; Vieira, Antonio W. ; M Campos, Mario F.: Online gesture recognition from pose kernel learning and decision forests. In: *Pattern Recognition Letters* 39 (2014), S. 65–73

[Miranda u. a. 2012] Miranda, Leandro ; Vieira, Thales ; Martinez, Dimas ; Lewiner, Thomas ; Vieira, Antônio W. ; Campos, Mario Fernando M.: Real-time gesture recognition from depth data through key poses learning and decision forests. In: *Graphics, Patterns and Images (SIBGRAPI), 2012 25th SIBGRAPI Conference on* IEEE (Veranst.), 2012, S. 268–275

[Mitra und Acharya 2007] Mitra, Sushmita ; Acharya, Tinku: Gesture recognition: A survey. In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 37 (2007), Nr. 3, S. 311–324

[Moeslund und Granum 2001] Moeslund, Thomas B. ; Granum, Erik: A survey of computer vision-based human motion capture. In: *Computer Vision and Image Understanding* 81 (2001), Nr. 3, S. 231–268

[Opera Software ASA 2001] Opera Software ASA: *Changelog for Opera 5.x for Windows*. 2001. – URL http://www.opera.com/docs/changelogs/windows/500-512/#510. – Zugriffsdatum: 21.07.2014

[Pu u. a. 2013] Pu, Qifan ; Gupta, Sidhant ; Gollakota, Shyamnath ; Patel, Shwetak: Whole-home gesture recognition using wireless signals. In: *Proceedings of the 19th annual international conference on Mobile computing & networking* ACM (Veranst.), 2013, S. 27–38

[Rabiner 1989] Rabiner, Lawrence R.: A tutorial on hidden Markov models and selected applications in speech recognition. In: *PROCEEDINGS OF THE IEEE*, 1989, S. 257–286

[Raptis u. a. 2011] Raptis, Michalis ; Kirovski, Darko ; Hoppe, Hugues: Real-time classification of dance gestures from skeleton animation. In: *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* ACM (Veranst.), 2011, S. 147–156

[RoboRealm] RoboRealm: *Microsoft Kinect*. – URL http://www.roborealm.com/help/Microsoft_Kinect.php. – Zugriffsdatum: 02.08.2014

[Roetenberg u. a. 2009] Roetenberg, Daniel ; Luinge, Henk ; Slycke, Per: Xsens MVN: full 6DOF human motion tracking using miniature inertial sensors. In: *Xsens Motion Technologies BV, Tech. Rep* (2009)

[Rubine 1991] Rubine, Dean: *Specifying gestures by example*. ACM, 1991

[Rumelhart u. a. 1988] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning representations by back-propagating errors. In: *Cognitive modeling* (1988)

[Sakoe und Chiba 1978] Sakoe, Hiroaki ; Chiba, Seibi: Dynamic programming algorithm optimization for spoken word recognition. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 26 (1978), Nr. 1, S. 43–49

[Salvador und Chan 2007] Salvador, Stan ; Chan, Philip: Toward accurate dynamic time warping in linear time and space. In: *Intelligent Data Analysis* 11 (2007), Nr. 5, S. 561–580

[Shotton u. a. 2013] Shotton, Jamie ; Sharp, Toby ; Kipman, Alex ; Fitzgibbon, Andrew ; Finocchio, Mark ; Blake, Andrew ; Cook, Mat ; Moore, Richard: Real-time human pose

recognition in parts from single depth images. In: *Communications of the ACM* 56 (2013), Nr. 1, S. 116–124

[Signer u. a. 2006] Signer, Beat ; Norrie, Moira ; Kurmann, Ueli ; Norrie, Moira ; Informaticienne, Grande-Bretagne ; Norrie, Moira: IGesture: a Java Framework for the Development and Deployment of Stoke-based Online Gesture Recognition Algorithms / ETH, Department of Computer Science. 2006 (561). – Forschungsbericht

[Starner und Pentland 1997] Starner, Thad ; Pentland, Alex: Real-time american sign language recognition from video using hidden markov models. In: *Motion-Based Recognition*. Springer, 1997, S. 227–243

[Svozil u. a. 1997] Svozil, Daniel ; Kvasnička, Vladimír ; Pospíchal, Jiří: Introduction to multi-layer feed-forward neural networks. In: *Chemometrics and intelligent laboratory systems* 39 (1997), Nr. 1, S. 43–62

[Swigwart 2005] Swigwart, Scott: *Easily Write Custom Gesture Recognizers for Your Tablet PC Applications*. November 2005. – URL http://msdn.microsoft.com/en-us/library/aa480673.aspx. – Zugriffsdatum: 20.08.2014

[Turetsky und Ellis 2003] Turetsky, R. ; Ellis, D.: Ground-Truth Transcriptions of Real Music from Force-Aligned MIDI Syntheses. In: *International Symposium on Music Information Retrieval* (2003), Nr. 4, S. 135–141. – URL http://www.ee.columbia.edu/~dpwe/resources/matlab/dtw/. – Zugriffsdatum: 23.07.2014

[Vatavu u. a. 2012] Vatavu, Radu-Daniel ; Anthony, Lisa ; Wobbrock, Jacob O.: Gestures as point clouds: a $ P recognizer for user interface prototypes. In: *Proceedings of the 14th ACM international conference on Multimodal interaction* ACM (Veranst.), 2012, S. 273–280

[Waibel u. a. 1989] Waibel, Alex ; Hanazawa, Toshiyuki ; Hinton, Geoffrey ; Shikano, Kiyohiro ; Lang, Kevin J.: Phoneme recognition using time-delay neural networks. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 37 (1989), Nr. 3, S. 328–339

[Wang u. a. 2003] Wang, Liang ; Hu, Weiming ; Tan, Tieniu: Recent developments in human motion analysis. In: *Pattern recognition* 36 (2003), Nr. 3, S. 585–601

[Wang u. a. 2005] Wang, Yiwen ; Kim, Sung-Phil ; Principe, Jose C.: Comparison of TDNN training algorithms in brain machine interfaces. In: *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on* Bd. 4 IEEE (Veranst.), 2005, S. 2459–2462

[Willems u. a. 2009] Willems, Don ; Niels, Ralph ; Gerven, Marcel van ; Vuurpijl, Louis: Iconic and multi-stroke gesture recognition. In: *Pattern Recognition* 42 (2009), Nr. 12, S. 3303–3312

[Wu u. a. 2014] Wu, Jie ; Wang, Changhu ; Zhang, Liqing ; Rui, Yong: Sketch Recognition with Natural Correction and Editing. In: *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014

[Yabukami u. a. 2000] Yabukami, Shin ; Kikuchi, H ; Yamaguchi, M ; Arai, KI ; Takahashi, K ; Itagaki, A ; Wako, N: Motion capture system of magnetic markers using three-axial magnetic field sensor. In: *Magnetics, IEEE Transactions on* 36 (2000), Nr. 5, S. 3646–3648

[Yamato u. a. 1992] Yamato, Junji ; Ohya, Jun ; Ishii, Kenichiro: Recognizing human action in time-sequential images using hidden markov model. In: *Computer Vision and Pattern Recognition, 1992. Proceedings CVPR'92., 1992 IEEE Computer Society Conference on* IEEE (Veranst.), 1992, S. 379–385

[Yang u. a. 2002] Yang, Ming-Hsuan ; Ahuja, Narendra ; Tabb, Mark: Extraction of 2d motion trajectories and its application to hand gesture recognition. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24 (2002), Nr. 8, S. 1061–1074

[Zeleznik u. a. 2007] Zeleznik, Robert C. ; Herndon, Kenneth P. ; Hughes, John F.: SKETCH: an interface for sketching 3D scenes. In: *ACM SIGGRAPH 2007 courses* ACM (Veranst.), 2007, S. 19

[Zhang 2012] Zhang, Zhengyou: Microsoft kinect sensor and its effect. In: *MultiMedia, IEEE* 19 (2012), Nr. 2, S. 4–10

[Zhou und Hu 2008] Zhou, Huiyu ; Hu, Huosheng: Human motion tracking for rehabilitation—A survey. In: *Biomedical Signal Processing and Control* 3 (2008), Nr. 1, S. 1–18

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 20. November 2014 Michael Zajonz