FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Intuitive und markerlose Interaktion in einer mobilen Virtual Reality Anwendung auf Basis von RGBD-Daten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Daniel Fritz
Matrikelnummer 0507049

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Priv.-Doz. Mag. Dr. Hannes Kaufmann
Mitwirkung: Dipl.-Ing. Annette Mossel

Wien, 19.09.2014

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Daniel Fritz
Mathoner Straße 60, 6562 Mathon

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagung

Diese Diplomarbeit ist meinen Eltern Josef und Marianne Fritz gewidmet. Danke für die jahrelange Unterstützung und Hilfe in allen Lagen des Lebens.

Ich möchte mich bei Hannes Kaufmann und Annette Mossel für die Betreuung bedanken. Sie sind mir immer mit Rat und Tat zur Seite gestanden und haben diese Arbeit erst möglich gemacht.

Darüber hinaus gebührt besonderer Dank meiner Familie, meinen Freunden und vor allem meiner Freundin. Danke für die zahlreichen Stunden Korrekturlesen, notwendigen Ablenkungen, Inspirations-Bonbons und immerzu motivierende Unterstützung.

Kurzfassung

Die Diplomarbeit behandelt berührungslose Interaktionstechniken auf mobilen Geräten, um damit eine virtuelle 3D-Szene zu manipulieren. Es werden verschiedene Lösungen und Kombinationen der Fingergestenerkennung ohne Hilfsmittel auf ihre Robustheit und Performance untersucht, als Erstes ohne zusätzliche Hardware und später mit einer Tiefenkamera als Erweiterung. Die Perspektive der virtuellen Szene wird mithilfe der Position des Kopfs gesteuert, um einen 3D-Effekt zu erzeugen. Durch Handposition und Fingergesten können Objekte der Szene selektiert und manipuliert werden.
Für die Kopferkennung wird auf einen bereits trainierten Haar-Kaskade Klassifikator zurückgegriffen und die 3D-Position anhand der ermittelten Größe des Kopfs relativ geschätzt. Selbst erstellte Haar-Kaskade Klassifikatoren werden verwendet, um auf Basis von RGB-Daten, das direkte Erkennen zweier verschiedener Gesten oder die generelle Handflächenerkennung zu ermöglichen und deren 2D-Position zu ermitteln. Wurde mittels Klassifikator die Handfläche erkannt, werden die beiden Gesten anhand der Fingerspitzenanzahl ermittelt. Dafür wird mithilfe von Bildverarbeitungsfunktionen die Hand segmentiert, die Kontur ermittelt und auf Fingerspitzen hin untersucht. Für die 3D-Interaktion werden die fehlenden Tiefenwerte durch verschiedene Ansätze (Größe der erkannten Handfläche, maximaler Grauwert der Hand) relativ geschätzt. Zur Verbesserung des 2D-Trackings mit relativer Tiefenschätzung wird Microsofts Tiefenkamera Kinect verwendet, um die RGB-Daten mit absoluten Tiefendaten zu erweitern (RGBD). Die RGBD-Daten sollen die Handsegmentierung und 3D-Interaktion verbessern.
Im Zuge dieser Diplomarbeit wird eine Android-Applikation mithilfe von OpenCV, Unity3D und OpenNI entwickelt. Der Prototyp besteht aus einem Rahmen, um eine feste Verbindung der Kameras von Tablet und Kinect sicher zu stellen und eine Kalibrierung bzw. Mapping durchführen zu können. Die Performance der implementierten Ansätze zur Fingergestenerkennung wurde in verschiedenen Licht- und Hintergrundsituationen systematisch evaluiert. Auf dieser Studie basierend wurden Empfehlungen für Entwickler erarbeitet, zur Auswahl der geeigneten Methode für ihre mobile 3D-Interaktion. Außerdem wurde eine experimentelle Studie durchgeführt, um die 3D-Interaktion zu evaluieren und die verschiedenen Charakteristiken der Methoden zur Tiefenschätzung zu zeigen. Dafür wurden mit den ermittelten Fingergesten die beiden Interaktionsaufgaben Selektion und Positionierung durchgeführt. Insgesamt wurde das beste Ergebnis mithilfe von RGBD-Daten zur Fingergestenerkennung und absoluten Tiefenschätzung erreicht, allerdings zulasten der Latenzzeit.

Abstract

The diploma thesis discusses touchless interaction techniques on handheld devices to intuitively manipulate a virtual 3D scene that is presented to the user on the handheld display. The robustness and performance of different solutions and combinations to detect the user's hand and the fingertips without markers are examined. Therefore, the first methods focus on using the built-in RGB camera, while later the built-in camera is combined with an additional depth sensor. Hand position and finger gestures are used to select and move objects in the virtual scene. Furthermore, the user's head position is tracked to adapt the perspective of the virtual scene in order to create a 3D impression on the device display.
The approaches use RGB data for hand segmentation, gesture detection and the hand size or the maximum gray scale value of the hand for relative depth estimation. In addition RGBD data is used for improved hand segmentation and absolute depth estimation. To detect two different finger gestures or the palm of the hand, Haar-like feature-based cascaded classifiers were trained. If the classifier is used to recognize the palm, the finger gestures are determined by the amount of detected fingertips. Therefore, different image processing operations are applied for hand segmentation and its contour is used for fingertip detection. An already trained Haar-like feature cascade classifier is implemented to detect the user's face and obtain its 3D position with relative depth estimation using the size of the face.
Within the diploma thesis an Android application is developed using OpenCV, Unity3D und OpenNI. The hardware prototype rigidly connects the handheld device with the depth sensor (Microsoft Kinect) to enable correct calibration and mapping of the received RGBD data. The performance of the approaches for gesture recognition were systematically evaluated by comparing their accuracy under varying illumination and background. Based on this study, useful guidelines for developers were derived to choose the appropriate technique for their mobile interaction task. Furthermore, an experimental study was conducted using the detected finger gestures to perform the two canonical 3D interaction tasks selection and positioning and demonstrate the different characteristics of the depth estimation methods. Overall the best result was obtained using RGBD data for finger gesture detection and absolute depth estimation at the expense of latency.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Teil I

Theoretische Grundlagen

KAPITEL 1

Einleitung

1.1 Fragestellung und Motivation

Die Interaktion zwischen Mensch und Computer (HCI[1]) begegnet uns mittlerweile im täglichen Leben. Hat sich am Computer die Interaktion mit physischen Geräten, Maus und Tastatur etabliert, bieten mobile Geräte mit Multi-Touch-Displays und verschiedenen Sensoren neue Möglichkeiten [134]. So erfolgt hier die Interaktion meist durch Berührung des Displays mittels Fingergesten. Diese Art der Interaktion verwandelt die grafische Oberfläche in eine natürliche Benutzerschnittstelle (NUI[2]) [8].
NUI's sollen eine intuitive und direkte Kommunikation zwischen Mensch und Maschine ermöglichen. In diese Kategorie fallen beispielsweise auch die Sprachsteuerung und berührungslose Ansätze [5, 46]. Durch berührungslose Methoden kann ein größerer Bereich als nur das 2D-Display für die Interaktion genutzt werden. Dafür bietet sich zum Beispiel bei mobilen Geräten mit Kamera die Verwendung von bildverarbeitenden Methoden an [4, 5].
Mobile Geräte (Smartphones und Tablets) werden immer leistungsstärker und haben keine Probleme mit der Echtzeitdarstellung von 3D-Inhalten und *Virtual Reality* (VR) Szenen. VR ist die Darstellung und Wahrnehmung einer künstlichen 3D-Szene, mit der in Echtzeit interagiert werden kann [25]. Der Benutzer kann zum Beispiel Objekte der Szene manipulieren[3] oder seine Position und Blickrichtung auf die VR-Szene ändern.
Die große Verbreitung von Smartphones und Tablets erhöht das Interesse an mobilen VR-Anwendungen und der Notwendigkeit neuer Interaktionsmöglichkeiten als Alternative zur Touch-Interaktion. 2D-Touch-Interaktion kann für die Manipulation und Steuerung von VR-Inhalten verwendet werden. Allerdings ist eine natürliche und intuitive 3D-Steuerung durch Berührung eines Displays aufgrund der Beschränkung auf 2D-Interaktion und den daraus resultierenden Gesten nicht möglich. Gesten sind in diesem Kontext Bewegungen von Händen bzw. Fingern, die zur Interaktion mit mobilen Geräten verwendet werden.

[1] Human-Computer Interaction
[2] Natural User Interface
[3] Manipulieren ist die Einflussnahme des Benutzers auf die Position der Objekte in einer VR-Szene.

Zur Ausschöpfung des vollen Potenzials einer mobilen VR-Anwendung ist eine intuitive 3D-Interaktion essenziell [15]. Eine solche 3D-Steuerung bedeutet, dass bekannte Gesten aus der realen Welt auch für die Interaktion mit der VR-Szene verwendet werden. In der realen Welt verwenden wir zu einem großen Teil unsere Hände bzw. Finger für die Interaktion mit Objekten. Deshalb ist es naheliegend, Hände und Finger auch für die 3D-Interaktion mit VR-Objekten zu verwenden. Dadurch wird nicht nur eine natürliche und intuitive Möglichkeit zur Interaktion geschaffen, sondern auch die Umstellung bei gleichzeitiger Interaktion mit realen und virtuellen Objekten erleichtert. Um einen anderen Blickwinkel auf ein Objekt oder die Umgebung zu bekommen, bewegen wir unseren Kopf und sollten dies auch in der virtuellen Szene machen können. Durch die Einbindung der Kopfbewegung zur Steuerung des Blickwinkels, entsteht bei der Betrachtung der Szene ein 3D-Effekt und die Immersion[4] des Benutzers in die virtuelle Umgebung wird erhöht. Ein weiterer wichtiger Punkt zur Ermöglichung einer intuitiven 3D-Steuerung ist es, die 2D-Interaktion um die Tiefenachse zu erweitern. Dafür wird die 3D-Position des Interaktionsobjekts benötigt.

In dieser Arbeit werden die oben erwähnten berührungslosen Methoden verwendet. Diese ermöglichen durch die Einbindung und Berücksichtigung der dritten Dimension eine intuitive, natürliche Interaktion mit 3D-Inhalten. Die Daten der in mobilen Geräten eingebauten RGB[5]-Kamera, werden zur Erkennung der Fingergesten und des Kopfs verwendet. Außerdem werden die RGB-Daten für eine relative Tiefenschätzung und Umsetzung der 3D-Positionsermittlung eingesetzt. Darauf basierende Ansätze verwenden eine zusätzliche Tiefenkamera für den Erhalt von RGBD[6]-Daten. Damit stehen tatsächliche Tiefenwerte zur Verfügung und ermöglichen eine absolute Tiefenschätzung des Interaktionsobjekts. Außerdem sollen Störfaktoren bei der Erkennung von Objekten durch den Einsatz einer Tiefenkamera verringert werden.
Berührungslose, bildverarbeitende Interaktionstechniken können in zwei Kategorien eingeteilt werden. In der ersten Kategorie verbessern Hilfsmittel, wie zum Beispiel Markierungen (*Marker*), die Erkennung und Segmentierung[7] des Interaktionsobjektes (beispielsweise der Hand). In der zweiten Kategorie werden keine zusätzlichen Hilfsmittel verwendet (*markerlos*), um einen einfachen Zugang ohne spezielle Vorbereitungen zu ermöglichen [46, 134].

Ziel ist die Umsetzung einer robusten, natürlichen und intuitiven 3D-Interaktion mit einer VR-Szene auf mobilen Geräten. Aufgrund der oben ausgeführten Überlegungen soll diese Arbeit untersuchen, ob mit berührungslosen Interaktionstechniken ohne Hilfsmittel dieses Ziel erreicht werden kann. Außerdem soll der zusätzliche Einsatz einer Tiefenkamera zur Verbesserung der Segmentierung und robusten 3D-Interaktion untersucht werden.

[4] Immersion beschreibt das Eintauchen des Benutzers in die virtuelle Szene und Abkopplung von der realen Umgebung.
[5] RGB steht für die Farben Rot, Grün und Blau.
[6] RGBD ist RGB mit Tiefenwerten (D) erweitert.
[7] Trennung vom Hintergrund des Bildes.

Es wird ein Prototyp entwickelt, mit dem eine robuste Erkennung von Kopf, Hand und Fingergesten erzielt wird. Mittels Kopf- und Fingergesten soll eine virtuelle 3D-Szene und deren Objekte manipuliert werden. Es wird untersucht, mit welchen Methoden sich eine robuste Erkennung des Interaktionsobjekts erzielen lässt. Außerdem wird überprüft, welche Ansätze sich zur Ermittlung der Tiefenposition des Interaktionsobjekts eignen. Die Tiefenkamera soll die Segmentierung und die Interaktion in der Tiefenachse verbessern. Diese Ansätze werden anschließend hinsichtlich Robustheit und Performance analysiert und evaluiert.
Teile der in dieser Diplomarbeit vorgestellten Ergebnisse wurden in den folgenden begutachteten Publikationen veröffentlicht:

1: D. Fritz, A. Mossel und H. Kaufmann. Evaluating RGB+D Hand Posture Detection Methods for Mobile 3D Interaction. In Proceedings of the 16th International Conference of Virtual Technoloogies (VRIC'14). ACM, 2014.

2: D. Fritz, A. Mossel und H. Kaufmann. Markerless 3D Interaction in an Unconstrained Handheld Mixed Reality Setup. The International Journal of Virtual Reality, 2014.

1.2 Problemstellung

Mit mobilen Geräten wird hauptsächlich durch die Berührung des 2D-Displays mit den Fingerspitzen interagiert. Eine 3D-Interaktion mit 3D-Inhalten einer VR-Szene ist über solche berührungsempfindlichen 2D-Displays nur mit Einschränkungen lösbar. Deshalb bedarf es anderer Methoden, die für eine intuitive und natürliche 3D-Interaktion geeignet sind. Dazu sollen Gesten aus der realen Welt zur Interaktion verwendet werden, wofür sich Kopf und Hand bzw. Finger als Interaktionsobjekte anbieten. Diese Gesten können mittels bildverarbeitenden Methoden erkannt werden. Voraussetzung für eine 3D-Interaktion ist die robuste Ermittlung von Interaktionsobjekten und deren 3D-Position. Ein wichtiges Kriterium dieser Arbeit ist, dass alle nötigen Berechnungen und Methoden auf dem mobilen Gerät ausgeführt werden und der Benutzer keinerlei zusätzliche Hilfsmittel für die Interaktion benötigt. Dies führt zu Herausforderungen für Hardware und Software.
Ziel dieser Arbeit ist es einen Prototyp zu entwickeln, mit dem eine robuste 3D-Interaktion mit mobilen VR-Anwendungen möglich ist. Ein handelsübliches Tablet soll als Recheneinheit verwendet und das RGB-Bild mittels Frontkamera aufgenommen werden. Eine kostengünstige Tiefenkamera soll als Erweiterung zum Einsatz kommen (RGBD-Daten).
Bei der Verwendung von bildverarbeitenden Methoden zur Erkennung von Objekten hat die Umgebungsbeleuchtung einen großen Einfluss. Verschiedenste Lichtverhältnisse können das Erkennen von Objekten erschweren. Ähnlich wie Hintergründe bzw. Umgebungen einen starken Einfluss darauf haben, wie gut das Objekt erkennbar und als solches identifizierbar ist. Damit eine robuste Erkennung möglich ist, sollen entsprechende Ansätze entwickelt und verschiedene Methoden getestet werden. Wird die Hautfarbe als Merkmal zur Erkennung eingesetzt, ist diese bzw. der Hautfarbe ähnliche Objekte ein weiterer Störparameter. Dabei kann es zu Verwechslungen und falschen Zuordnungen der gesuchten Interaktionsobjekte kommen. Der Prototyp soll in der Lage sein, zwischen Hand und Kopf zu unterscheiden und alle anderen Objekte nicht als solche wahrzunehmen. Eine genaue Ermittlung der 3D-Position der Interaktionsobjekte ist

ohne Tiefendaten nicht realisierbar. Um eine 3D-Interaktion zu ermöglichen, soll der Prototyp Methoden zur Tiefenschätzung verwenden. Da alle Berechnungen und Methoden auf dem Tablet ausgeführt werden, ist es notwendig, diese für den mobilen Betrieb zu optimieren, um eine Echtzeit-Interaktion[8] zu ermöglichen.
Das RGB-Bild der Tabletkamera wird auf Kopf und Fingergesten untersucht und mithilfe von Klassifikatoren identifiziert. Als zweite Variante wird die Hand mittels Klassifikator identifiziert und die Fingerspitzen zur Ermittlung der Fingergesten eingesetzt. Zur Segmentierung der Hand und Erkennung der Fingerspitzen werden verschiedene bildverarbeitende Methoden verwendet. Die Detektion liefert Auskunft über das erkannte Objekt, die Größe und dessen 2D-Position. Die benötigte 3D-Position soll mithilfe verschiedener Merkmale (Objektgröße, Grauwert) der Interaktionsobjekte durch eine relative Tiefenschätzung geschätzt werden. Die Daten der Tiefenkamera werden mit dem RGB-Bild zu RGBD-Daten kombiniert. Durch RGBD-Daten können die 3D-Positionen der Interaktionsobjekte ohne relativer Tiefenschätzung bestimmt und dadurch die Tiefeninteraktion verbessert werden. Außerdem soll sich durch RGBD-Daten die Segmentierung der Hand verbessern, indem weiter entfernte Objekte ausgeblendet werden.
Anhand der ermittelten Informationen kann die gewünschte 3D-Interaktion umgesetzt werden. Die Bewegung des Kopfes steuert die *virtuelle Kamera* und damit die Position und Blickrichtung des Betrachters auf die VR-Szene. Dadurch kann beim Betrachten der Szene ein 3D-Effekt erzeugt werden. Für die Interaktion mit den Objekten der VR-Szene werden zwei verschiedene Fingergesten verwendet. Mit der ersten Geste wird die *virtuelle Hand* bewegt und damit Objekte selektiert. Die virtuelle Hand beschreibt die Position und Orientierung der realen Hand in der VR-Szene und wird zur Interaktion mit der Szene verwendet. Die zweite Geste wird zur Manipulation des selektierten Objekts eingesetzt.

1.3 Abgrenzung

Ziel dieser Arbeit ist es, eine natürliche und intuitive 3D-Interaktion mit einer mobilen VR-Anwendung zu ermöglichen. Dafür werden berührungs- und markerlose Interaktionstechniken verwendet. 2D-Touch-Interaktion und berührungslose Methoden mit Hilfsmittel sind nicht Teil dieser Arbeit. Ebenfalls werden Ansätze die für eine mobile 3D-Interaktion ungeeignet sind nicht berücksichtigt. Zur Erkennung der Fingergesten bzw. der Hand werden Haar-Klassifikatoren [65, 127] trainiert und erstellt. Für die Kopferkennung wird auf einen bereits trainierten Haar-Klassifikator zurückgegriffen. Die Hautfarbe wird in dieser Arbeit nicht als Merkmal für die Erkennung der Objekte verwendet. Die Tiefenkamera wird als Erweiterung eingesetzt, um eine Verbesserung der Segmentierung und 3D-Interaktion untersuchen zu können. Die RGBD-Daten werden nicht zur Erkennung der Interaktionsobjekte verwendet. Für die Bildverarbeitungsfunktionen, Einbindung der Tiefenkamera und Netzwerkkommunikation werden entsprechende Programmbibliotheken verwendet. Die Darstellung der VR-Szene wird mit einer vorhandenen Spiel-Engine realisiert und in die Anwendung eingebunden.

[8] Echtzeit-Interaktion ist die zeitnahe Reaktion der VR-Szene auf die Interaktion des Benutzers.

1.4 Kapitelübersicht

Diese Arbeit ist in sieben Kapitel aufgeteilt. Zu Beginn wird in *Kapitel 1* das Thema vorgestellt und das Ziel dieser Arbeit formuliert. In *Kapitel 2* werden Arbeiten, die sich mit 3D-Interaktion beschäftigen, beschrieben. Außerdem wird auf Projekte mit ähnlichen Ansätzen und Technologien eingegangen. *Kapitel 3* erläutert den Ansatz der Methoden dieser Arbeit und die theoretischen Hintergründe der eingesetzten Technologien. Unter anderem wird das Erzeugen der RGBD-Daten, die Haar-Kaskade Klassifikatoren und die verwendeten Bildverarbeitungsfunktionen näher betrachtet. Die Implementierung wird in den Kapiteln 4 und 5 erläutert. *Kapitel 4* widmet sich der Kalibrierung der Kameras, dem Training von Klassifikatoren und der Entwicklungsumgebung zur Implementierung der Software. In *Kapitel 5* wird der Aufbau und die Entwicklung der einzelnen Software-Module dokumentiert. Mithilfe des Prototyps werden in *Kapitel 6* die Klassifikatoren und Methoden zur Fingergestenerkennung hinsichtlich ihrer Robustheit und Performance evaluiert. Außerdem wird die 3D-Interaktion mit der VR-Szene untersucht. Abschließend gibt *Kapitel 7* einen Überblick auf die Erkenntnisse dieser Arbeit und einen Ausblick auf mögliche Verbesserungen und Weiterentwicklungen.

KAPITEL 2

Verwandte Arbeiten

Für die Umsetzung einer 3D-Interaktion können viele verschiedene Ansätze verwendet werden. In Abschnitt 2.1 wird ein Einblick auf verwandte Arbeiten und Methoden zur 3D-Interaktion gegeben. Darin werden Forschungsprojekte, die 2D-Touch-Interaktion oder berührungslose Methoden mit Hilfsmittel zur Steuerung einsetzen, betrachtet und auf den Unterschied von dynamischen und statischen Fingergesten eingegangen. Diese Arbeit verwendet berührungslose Interaktionstechniken ohne Hilfsmittel. Verwandte Arbeiten mit diesem Ansatz werden in Abschnitt 2.2 gesondert vorgestellt und detaillierter betrachtet.

2.1 3D-Interaktion

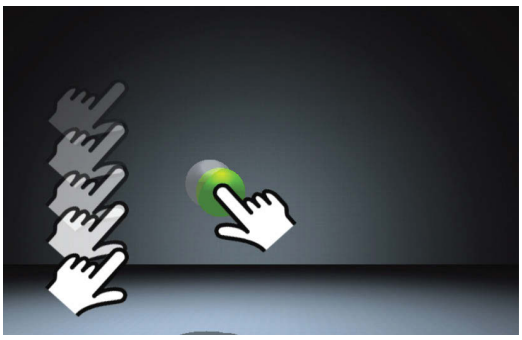
2.1.1 Interaktion mit berührungsempfindlichen Displays

Wie in 1.1 schon kurz erwähnt, wird mit mobilen Geräten meist durch eine Berührung des Displays interagiert. Das Display kann mehrere, gleichzeitige Berührungen verarbeiten (Multi-Touch-Display) und ermöglicht dadurch Touch-Gesten mit mehreren Fingern. Beispiele für solche Gesten sind Scrollen mithilfe eines Fingers[1], Pinch-To-Zoom[2] und viele mehr. Diese Gesten für 2D-Interaktion verwandeln die grafische Oberfläche in ein NUI [5, 8, 18, 46, 71, 81, 132, 133]. Durch die Berührung eines Displays ist allerdings keine direkte und somit natürliche 3D-Interaktion realisierbar. Die 2D-Touch-Interaktion muss erweitert werden, um die Interaktion in der Tiefenachse zu ermöglichen. Bei Hancock et al. [47] werden bis zu drei Finger für die Manipulation von 3D-Objekten eingesetzt. Damit wird die gleichzeitige 2D-Translation und 3D-Rotation eines Objekts ermöglicht. Reisman et al. [108] verwenden drei oder mehr Berührungspunkte, um ein Objekt manipulieren zu können und beziehen die Perspektive des Benutzers mit ein.

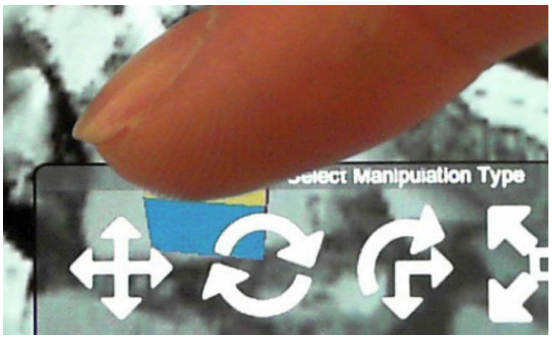
[1] Fingergeste zum Scrollen des Displays durch nach oben oder unten streichen eines Fingers.
[2] Fingergeste zum Vergrößern oder Verkleinern von Inhalten durch Auf- und Zuziehen zweier Finger.

Die *Z-Technik* von Martinet et al. [70] (siehe Abbildung 2.1a) verwendet die Bewegung des ersten Fingers (im Bild der rechte Finger) für die 2D-Translation und den zweiten Finger (im Bild der linke Finger) zur Steuerung der Tiefenposition des Objekts. Damit ist keine Rotation des Objekts möglich, weshalb Martinet et al. in [71] die Z-Technik mit der Rotationssteuerung von Reisman et al. [108] kombiniert haben.

Eine andere Möglichkeit zur Erweiterung der 2D-Interaktion ist der Einsatz von Sensoren. Mossel et al. [78] verwenden in ihren Techniken *3DTouch* und *HOMER-S* zusätzlich zur Touch-Interaktion die Bewegung und Position des Gerätes. In 3DTouch definiert die Geräteposition die Ebene, in der mittels einfacher 1-Finger Touch-Eingabe interagiert werden kann. HOMER-S ermöglicht es, das mittels Touch selektierte Objekt durch die reale Bewegung und Position des Gerätes zu manipulieren. Die Art der Manipulation kann vom Benutzer in einer grafischen Oberfläche ausgewählt werden (siehe Abbildung 2.1b).



(a) Z-Technik von Martinet et al. [70]



(b) HOMER-S von Mossel et al. [78]

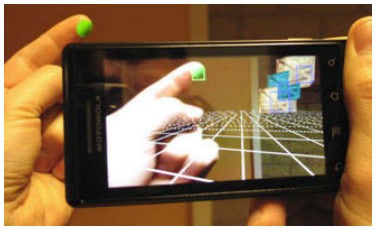
Abbildung 2.1: Beispiele für die Interaktion mit berührungsempfindlichen Displays

2.1.2 Berührungslose 3D-Interaktion mit Hilfsmittel

Für die Umsetzung von berührungsloser Interaktion mit mobilen Geräten verwenden Ketabdar und Jahanbekam [56] magnetische Ringe am Finger. Die Bewegung wird durch den im Smartphone integrierten digitalen Kompass erfasst.
Eine andere Möglichkeit bietet die Verwendung von Datenhandschuhen zur Interaktion [14] (siehe Abbildung 2.2a). Sie können wie bei Kallmann und Thalmann [54] und Kevin et al. [105] durch zusätzliche magnetische Sensoren zur Ermittlung der 3D-Position eingesetzt werden. Bowman et al. [15] verwenden *Pinch Gloves*^{TM} zur Interaktion mit virtuellen Umgebungen. Damit werden die Berührungen der Fingerspitzen registriert und die verschiedenen Kombinationen können als Gesten zur Interaktion verwendet werden.
Hilfsmittel für visuelle Methoden sind beispielsweise durch Farbe hervorgehobene Interaktionsobjekte oder an ihnen angebrachte Markierungen. Dadurch soll die Erkennung und Segmentierung des Interaktionsobjekts verbessert werden. Wang und Popović [128] haben ein spezielles Farbmuster für ihre Handschuhe entworfen (siehe Abbildung 2.2b). Damit wird die Hand in verschiedenfarbige Regionen aufgeteilt, um die Handhaltung und Fingerbewegungen ermitteln zu

(a) Pinch Gloves™ [24]

(b) Handschuhe von Wang und Popović [128], Bild von [50]



(c) Fingermarkierung in der Arbeit von Hürst und van Wezel [51]

Abbildung 2.2: Beispiele für berührungslose 3D-Interaktion mit Hilfsmittel

können. In der Arbeit von Keskin et al. [55] wird ein einfarbiger Handschuh zur Erkennung und Segmentierung der Hand verwendet. Hürst und van Wezel [51] markieren eine Fingerspitze mit grüner Farbe (siehe Abbildung 2.2c). Diese lässt sich dadurch erkennen und segmentieren.

2.1.3 Statische und dynamische Gesten

Zur berührungslosen Interaktion mit Objekten und Szenen werden statische und dynamische Gesten mit Kopf, Hand und Fingern verwendet. Eine statische Geste ist eine bestimmte Haltung von Hand oder Fingern, wie beispielsweise die offene Handfläche oder den nach oben gestreckten Zeigefinger. Statische Gesten werden in vielen Arbeiten verwendet [20, 21, 46, 60, 72, 100, 112, 123, 125, 134, 135]. Andere Arbeiten verwenden dynamische Gesten. Diese bestehen aus einer bewegten statischen Geste oder einer Kombination aus verschiedenen statischen Gesten [12, 14, 55, 69, 102, 115, 117]. Beispiele dafür sind das Wischen zur Seite mit der flachen Hand oder das Greifen von Objekten. In dieser Arbeit werden statische Fingergesten als Fingergesten bezeichnet.

2.2 Berührungslose 3D-Interaktion ohne Hilfsmittel

Die vorliegende Arbeit verwendet berührungslose Methoden ohne Hilfsmittel zur 3D-Interaktion. Verschiedene Ansätze dazu werden in den nächsten Abschnitten vorgestellt und erläutert. Für eine bessere Übersicht wird zwischen den Methoden zur Kopf- und Handinteraktion auf Basis von RGB-Daten und der Verwendung RGBD-Daten (Tiefendaten) unterschieden.

2.2.1 Kopferkennung und -interaktion mittels RGB-Daten

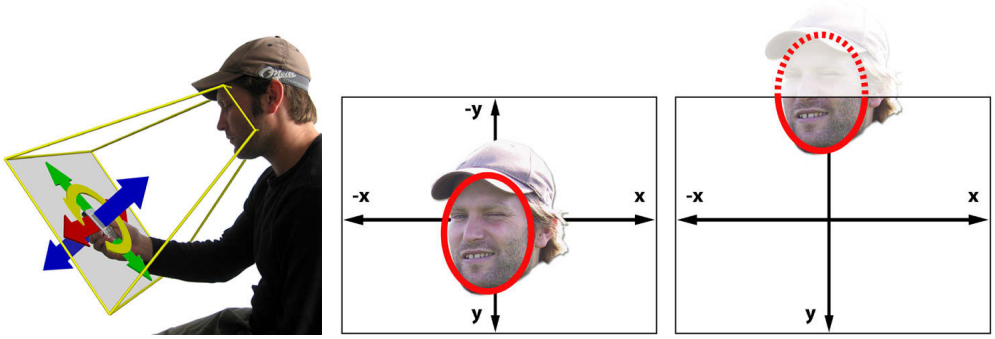
In diesem Abschnitt wird zuerst eine Arbeit vorgestellt, die drei Ansätze zur Kopferkennung erläutert und vergleicht. Anschließend werden zwei Arbeiten betrachtet, welche den Kopf als Interaktionsobjekt für mobile Anwendungen verwenden.

Für die Kopferkennung mit RGB-Daten werden oft Haar-Merkmale verwendet [65, 127]. Haar-Merkmale basieren auf den Helligkeitsunterschieden verschiedener Regionen und werden aus einem Graustufenbild extrahiert (siehe Abschnitt 3.2). Der Kopf bietet mit Augen, Nase und Mund sehr eindeutige, robuste Merkmale für diese Art der Erkennung [45, 65, 123]. Andere Ansätze nutzen die Hautfarbe oder lokale binäre Muster (LBP[3]) zur Erkennung. In der Arbeit von Ciaramello Hemami [22] werden diese drei Ansätze zur Detektion des Kopfes auf mobilen Geräten erläutert und verglichen.

Der erste Ansatz von Ciaramello und Hemami [22] verwendet die Hautfarbe zur Segmentierung. Danach werden kleine Hautregionen herausgefiltert und Löcher in den Regionen mithilfe morphologischer Bildverarbeitung [114] (Erosion) aufgefüllt. Zum Schluss wird die größte, zusammenhängende Hautregion als Kopf angenommen.
Im zweiten Ansatz werden Merkmale basierend auf lokalen binären Mustern (LBP) von Pixeln verwendet. Das LBP für ein Pixel wird durch den Vergleich mit den Nachbarpixeln berechnet (zum Beispiel 3x3 Nachbarschaft) und zu einer binären Zahl zusammengefügt. Mithilfe dieser Merkmale wird ein Klassifikator trainiert und für die Kopferkennung eingesetzt.
Haar-Merkmale werden im dritten Ansatz für das Training eines Klassifikators verwendet. Der Detektionsalgorithmus besteht aus mehreren Stufen, in denen die Anzahl der verwendeten Merkmale zur Erkennung sukzessive erhöht wird. Identifiziert der Klassifikator den untersuchten Bereich als möglichen Kopf, steigt dieser in die nächste Stufe auf. Sind alle Stufen mit einer positiven Zuordnung durchlaufen worden, wurde der untersuchte Bereich als Kopf erkannt.
Die Evaluierung dieser drei Ansätze ergab, dass der erste Ansatz zwar die schnellste Detektion liefert, allerdings zu sehr von der Umgebungsbeleuchtung bzw. Segmentierung abhängig ist. Der zweite Ansatz ist der langsamste und liefert zu viele falsch erkannte Objekte. Die beste Mischung aus richtiger Erkennung und falscher Zuordnung von Objekten liefert der dritte Ansatz.

Hansen et al. [49] stellen eine Methode zur 3D-Interaktion mit mobilen Anwendungen vor. Diese Arbeit baut auf MIXIS[4] auf [48]. Die Grundidee von MIXIS ist es, zusätzlich den Raum um das mobile Gerät in die Interaktion einzubinden. Für die 3D-Interaktion wird das Gerät in die entsprechende Richtung bewegt und der Kopf als Bezugspunkt herangezogen (siehe Abbildung 2.3a). Die Rotation beschränkt sich in dieser Arbeit auf die Tiefenachse. Das Projekt nutzt die RGB-Daten der Frontkamera des mobilen Gerätes und verwendet Farbhistogramme für die Erkennung des Kopfes. Die Detektion liefert die 2D-Position, Größe und Orientierung des Kopfes. Die Position und Orientierung des Gerätes wird über den Bezugspunkt (Kopf) festgestellt. Der Benutzer kann mithilfe der relativen Tiefenschätzung (Abstand des Gerätes zum Kopf) in der Tiefenachse interagieren. Da der Kopf als Bezugspunkt für die Interaktion verwendet wird, ist aufgrund des Bildverhältnisses der Interaktionsraum in der x-Achse größer (siehe Abbildung 2.3b). Probleme können auftreten, wenn das Gesicht nur teilweise sichtbar ist und dadurch Merkmale für die Ermittlung der Tiefe bzw. Orientierung verfälscht werden (siehe Abbildung 2.3c).

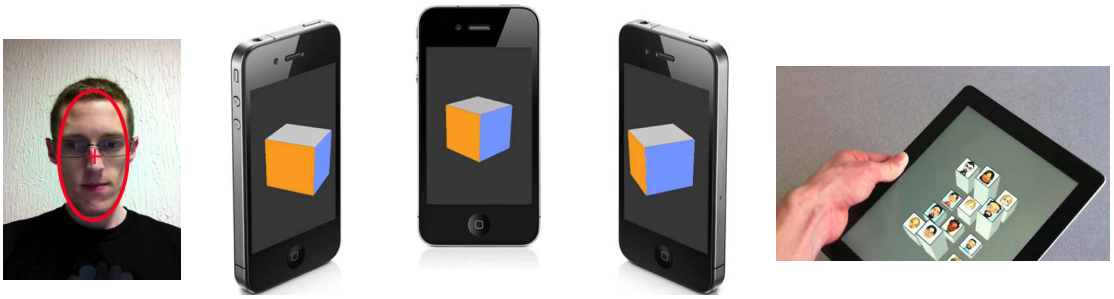
[3] Local Binary Patterns.
[4] Mixed Interaction Space.



(a) Interaktionsmöglichkeiten (b) Bildverhältnis (c) Teilweise sichtbarer Kopf

Abbildung 2.3: Interaktionstechnik MIXIS von Hansen et al. [49]

Das Forschungsprojekt von Francone und Nigay [29] nutzt die Kopfposition zur 3D-Interaktion mit einem mobilen Gerät. Dafür werden die RGB-Daten der Frontkamera eingesetzt und mittels Haar-Kaskade Klassifikator der Kopf erkannt (siehe Abbildung 2.4a). Die relative 2D-Position des Kopfes wird über die Position des Kopfs im RGB-Bild ermittelt. Die Größe des erkannten Objekts wird zur Bestimmung der Entfernung und 3D-Position verwendet. Dies kann zu Problemen bei nur teilweise sichtbarem Kopf führen, vergleichbar mit der zuvor beschriebenen Arbeit von Hansen et. al [48] (siehe Abbildung 2.3c). Die 3D-Position wird für die automatische Anpassung der Anzeige auf dem mobilen Gerät verwendet und ermöglicht damit eine natürliche und interaktive 3D-Ansicht. Die Darstellung der 3D-Szene verändert sich relativ zur Position des Betrachters und verwandelt dadurch das Betrachten eines flachen Displays in eine 3D-Illusion (siehe Abbildung 2.4b). Diese Technik der Perspektivensteuerung via Kopf nennt sich HCP[5] und erweitert die Möglichkeiten zur Darstellung von und Interaktion mit 3D-Inhalten.
Francone und Nigay stellen in ihrer Arbeit neben der Möglichkeit zur Visualisierung von 3D-Inhalten weitere Anwendungsfälle vor. Durch HCP kann die Tiefenachse für die Informationsvisualisierung verwendet (siehe Abbildung 2.4c) oder das Display erweitert werden.



(a) Kopferkennung (b) Manipulation des VR-Objekts durch HCP (c) Informationsvisualisierung, Bild von [131]

Abbildung 2.4: Forschungsprojekt mit HCP von Francone und Nigay [29]

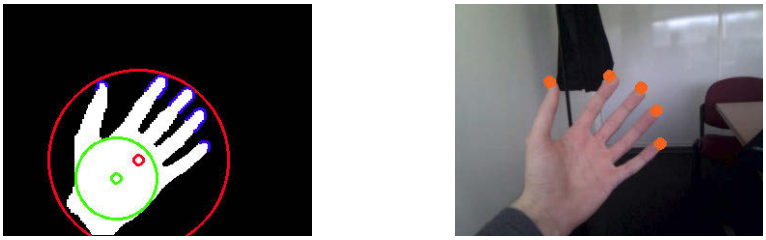
[5]Head-Coupled Perspective.

2.2.2 Handerkennung und -interaktion mittels RGB-Daten

In vielen Arbeiten auf dem Gebiet der Handerkennung ohne Hilfsmittel auf mobilen Geräten, werden RGB-Daten und die Hautfarbe als Merkmal verwendet [5, 21, 60, 61, 72, 123]. Diese Methode benötigt kein spezielles Training, jedoch ist es mit der Hautfarbe allein nicht möglich, die Hand vom Kopf oder anderen, der Hautfarbe ähnlichen Objekten zu unterscheiden.
Ein anderer Ansatz ist es, Haar-Merkmale [65, 127] für die Handerkennung zu nutzen [7, 109]. Haar-Klassifikatoren sind rechenintensiv und benötigen ein aufwendiges Training, dafür kann damit robust zwischen Hand und Kopf unterschieden werden. Andere Ansätze verwenden eine Kombination der beiden Methoden [22,134]. Der Haar-Klassifikator wird für die Kopferkennung und diese Information zur Handerkennung mittels Hautfarbe verwendet.
Für die Fingerspitzenerkennung mit RGB-Daten kommen meistens verschiedene Bilderverarbeitungsschritte zum Einsatz, um eine Kontur der Hand zu ermitteln [5, 60, 61]. Durch die Analyse der Charakteristik der Kontur kann eine Aussage darüber getroffen werden, wo sich die Fingerspitzen befinden. Dafür werden verschiedene Kriterien herangezogen und kombiniert.

Die folgende Auswahl gibt einen näheren Einblick in einige Arbeiten, welche die oben erwähnten Ansätze zur Erkennung von Hand bzw. Fingerspitzen verwenden und diese für verschiedene Gesten und Interaktionsmöglichkeiten einsetzen. Diese Arbeiten verwenden Methoden, die mit mobilen Geräten umgesetzt wurden oder dafür geeignet sind.

In der Forschungsarbeit von Baldauf et al. [5] wird ein Ansatz für die Hand- und Fingerspitzenerkennung mit der rückseitigen Kamera eines Smartphones vorgestellt. Sie verwenden die Hautfarbe als Merkmal und eine auf Pixeln basierende Methode für die Detektion. Hierbei wird jedes Pixel einzeln klassifiziert ohne Berücksichtigung der räumlichen Zusammenhänge. Dadurch ist die Methode unabhängig von der Orientierung der Hand und robust bei teilweiser Verdeckung. Mittels eines Schwellwerts werden hautfarbene Pixel segmentiert und morphologische Funktionen [114] zur Entfernung von Bildrauschen und zum Auffüllen von Löchern in Hautregionen verwendet. Im nächsten Schritt werden die Konturen ermittelt und die größte, zusammenhängende Region als Hand definiert.
Zur Erkennung der Fingerspitzen wird der innere und äußere Kreis der Handkontur berechnet. Abbildung 2.5a zeigt die Handkontur in weiß und die beiden Kreise mit Mittelpunkten in grün bzw. rot. Konturpunkte mit einem festgelegten Minimalabstand zum Zentrum des inneren Kreises legen Regionen mit potenziellen Fingerspitzen fest (in Abbildung 2.5a blau gekennzeichnet). Der Vektor vom Mittelpunkt des inneren zum Mittelpunkt des äußeren Kreises definiert die ungefähre Ausrichtung der Hand. Dadurch können Regionen die sich in Richtung Unterarm befinden ausgeschlossen werden. Zum Schluss wird für jede verbliebene Region das lokale Maximum der Distanz zum inneren Kreismittelpunkt als endgültige Fingerspitze angenommen. Abbildung 2.5b zeigt die erkannten Fingerspitzen in orange und macht die Abweichungen von den tatsächlichen Fingerspitzen sichtbar.



(a) Charakteristiken der Handkontur (b) Fingerspitzenvisualisierung

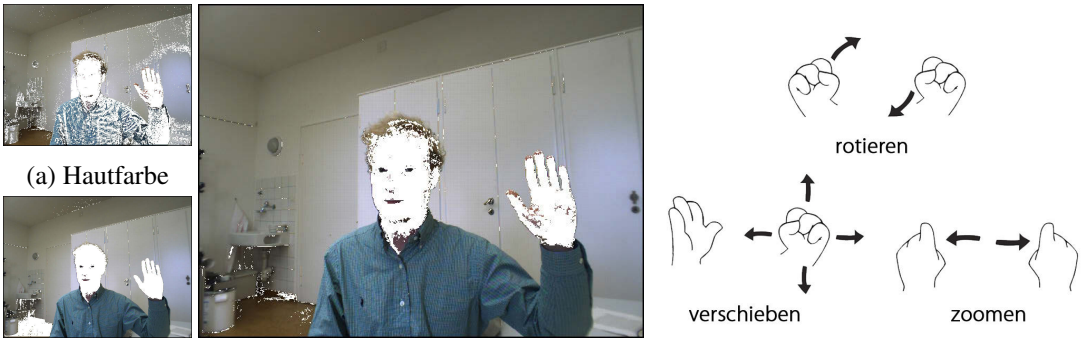
Abbildung 2.5: Fingerspitzenerkennung von Baldauf et al. [5]

Rodriguez et al. [109] beschäftigen sich in ihrer Arbeit mit 3D-Handinteraktion und nutzen dafür die RGB-Daten einer Webcam. Für die Erkennung des Interaktionsobjekts werden Haar-Merkmale verwendet. Es wird ein Klassifikator eingesetzt, der die geballte Faust erkennt. Zur Verbesserung der Detektion werden zwei Filter verwendet, um falsche Zuordnungen zu entfernen. Nach der Detektion sind die Größe und 2D-Position der Hand bekannt. Im nächsten Schritt wird der fehlende Tiefenwert und somit die 3D-Position der Hand ermittelt. Die Handgröße wird für die relative Tiefenschätzung eingesetzt. Dadurch kann festgestellt werden ob sich die Hand der Kamera nähert oder entfernt.
Diese Interaktionstechnik lässt sich für 3D-Interaktion einsetzen und Rodriguez et al. beschreiben in ihrer Arbeit zwei Anwendungsszenarien. Das erste ist eine VR-Szene, in der ein Objekt durch die Bewegung der Hand in alle Richtungen verschoben werden kann und der 3D-Position folgt. Im zweiten Anwendungsfall kann die Hand zur Steuerung einer Fotogalerie verwendet werden. Mit Bewegungen in der horizontalen Achse kann zwischen den Bildern gewechselt werden. Die Tiefeninteraktion wird für das Vergrößern bzw. Verkleinern der Bilder genutzt.

Die Arbeit von Van den Bergh et al. [123] verwendet eine RGB-Kamera zur Erkennung verschiedener Handgesten für die 3D-Interaktion. Die Hände des Benutzers werden über die Segmentierung durch Hautfarbe ermittelt. Ein im Voraus trainiertes Gaußsches Mixtur-Modell (GMM[6]) wird für die Bestimmung der Hautpixel verwendet (siehe Abbildung 2.6b). Unter gleichbleibenden Lichtbedingungen können damit gute Ergebnisse erzielt werden. Durch die Kombination mit einer zweiten Methode soll die Segmentierung robuster gegenüber Lichtwechsel und -veränderungen sein (siehe Abbildung 2.6c). Dafür werden Farbhistogramme (ein Histogramm für Hautpixel und eines für andere Pixel) verwendet und während der Laufzeit angepasst (siehe Abbildung 2.6a). Mittels Haar-Klassifikator wird das Gesicht erkannt und die Pixel innerhalb bzw. außerhalb dieser Region für die Initialisierung und Aktualisierung des entsprechenden Histogramms verwendet. Das segmentierte Bild wird zum Schluss mit einem Medianfilter geglättet und die Hände über die zwei größten, zusammenhängenden Komponenten (ohne Kopf) ermittelt. Die Bildausschnitte der Hände werden anschließend für die Ermittlung der Fingergeste genutzt. Dafür wird ein Klassifikator basierend auf ANMM[7]-Transformation und den Haar-Merkmalen von Lienhart und Maydt [65] verwendet (siehe [124]). Zur Bestimmung der Fingergesten werden

[6] Gaussion Mixture Model.
[7] Average Neighborhood Margin Maximization.

die vom Klassifikator ermittelten Koeffizienten mit den gespeicherten Werten der trainierten Fingergesten verglichen und dem besten Fund zugeordnet.
Die Interaktionstechnik in dieser Arbeit nutzt drei Fingergesten und die Bewegungen beider Hände für die 3D-Interaktion mit einem 3D-Modell (siehe Abbildung 2.6d). Durch das Ballen zu zwei Fäusten kann der Benutzer das Modell in der z- und y-Achse rotieren (siehe Abbildung 2.7e). Die Rotation entlang der z-Achse wird über die relative Veränderung des Winkels zwischen den beiden Fäusten gesteuert. Für die Rotation in der y-Achse wird die Größenveränderung (relative Tiefenschätzung) der Fäuste herangezogen, wodurch eine 3D-Rotation ermöglicht wird. Mit nur einer Faust kann sich der Benutzer durch das Modell bewegen. Die Zeigefingergeste wird für das Vergrößern bzw. Verkleinern des Modells eingesetzt.



(a) Hautfarbe (b) GMM (c) Kombination von Farbe und GMM (d) Fingergesten zur Interaktion

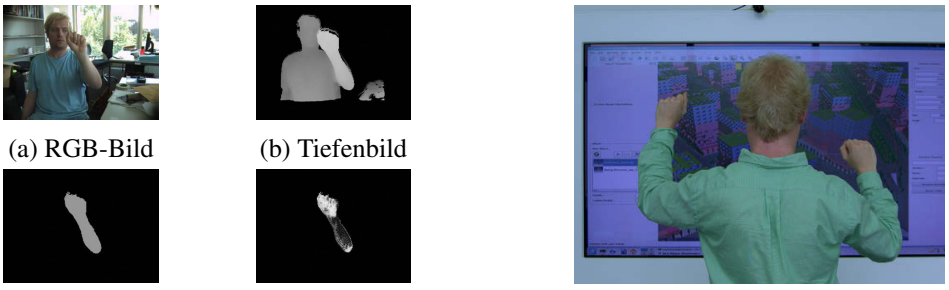
Abbildung 2.6: Handinteraktion mittels RGB-Daten von Van den Bergh et al. [123]

2.2.3 Tiefenkamera (RGBD)

Zusätzlich zu den RGB-Daten werden in vielen Arbeiten Tiefeninformationen, sogenannte 2.5D-Daten mit RGB-Daten kombiniert (RGBD-Daten) und zur Erweiterung und Verbesserung der Segmentierung und Tiefeninteraktion eingesetzt [33, 59, 82, 99, 101, 125]. Andere Arbeiten verwenden nur Tiefendaten zur Detektion [46, 63, 134], worauf hier nicht näher eingegangen wird.
Für die Umsetzung von Tiefenkameras gibt es verschiedene Ansätze, zwei davon sind Time-of-Flight (ToF) und strukturierte Beleuchtung. ToF-Kameras nutzen die Laufzeitmessung von Licht für die Distanzmessung. Systeme mit strukturierter Beleuchtung, projizieren ein bestimmtes Lichtmuster auf die Objekte der Umgebung. Eine Kamera nimmt die Abbildung dieses Musters auf. Über den Vergleich der Aufnahme mit dem ursprünglichen Lichtmuster, lässt sich die Distanz der Objekte vor der Kamera durch Triangulation berechnen. Für eine detaillierte Beschreibung wird auf den Abschnitt 3.1.2 verwiesen. Tiefenkameras liefern eine absolute Schätzung der Entfernungen von Punkten, womit eine bessere Unterscheidung von Hintergrund und Interaktionsobjekt möglich ist. Außerdem können die Tiefendaten zur Ermittlung der 3D-Position des Interaktionsobjekts eingesetzt werden.

Die vorgestellten Arbeiten verwenden verschiedene Tiefenkamerasysteme, um RGBD-Daten zu erzeugen. Diese werden zur Hand- bzw. Fingergestenerkennung eingesetzt. Die Ansätze dafür sind mit dem Ansatz dieser Arbeit, RGBD-Daten auf mobilen Geräten einzusetzen, vergleichbar.

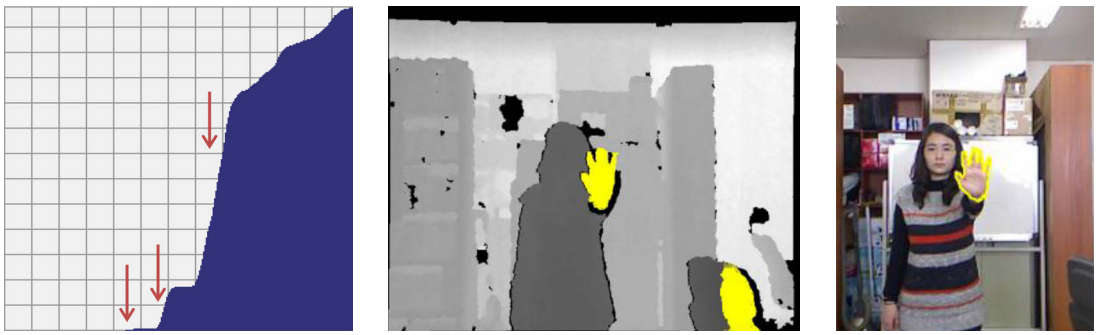
Van den Bergh und Van Gool [125] stellen eine Erweiterung der in Abschnitt 2.2.2 vorgestellten Methode [123] für 3D-Interaktion vor. Zur Verbesserung der Handsegmentierung wird eine ToF-Kamera eingesetzt. Dadurch sollen Objekte im Hintergrund herausgefiltert und Hände, die sich mit dem Kopf oder anderen Personen überlappen besser erkannt werden. Die verwendete ToF-Kamera verfügt über eine geringe Auflösung (176x144 Pixel). Deshalb wird sie mit einer höher auflösenden RGB-Kamera (640x480 Pixel) kombiniert. Für die Segmentierung ist die Auflösung der ToF-Kamera ausreichend und die RGB-Daten ermöglichen eine akkurate Handerkennung. Abbildung 2.7a zeigt das RGB-Bild und Abbildung 2.7b das Bild der Tiefenkamera. Die Kameras werden fest verbunden und extrinsisch kalibriert, um die beiden Kamerabilder deckungsgleich übereinanderlegen zu können. Dadurch kann jedem Pixel im RGB-Bild der entsprechende Tiefenwert der ToF-Daten zugeordnet und RGBD-Daten ermittelt werden (*Mapping*).
Zu Beginn der Segmentierung wird der Kopf erkannt. Dessen Entfernung wird ermittelt und als Schwellwert eingesetzt, um alle weiter entfernten Pixel im Bild zu entfernen. Der Rest wird für die Hand- und Gestenerkennung verwendet. Für die Lösung des Problems der Überlagerung von Kopf und Hand wird vom Schwellwert ein konstanter Parameter subtrahiert und somit nur mehr die Objekte vor dem Kopf verwendet (siehe Abbildung 2.7c). Im nächsten Schritt der Segmentierung wird die Hautfarbe als Merkmal eingesetzt (siehe Abbildung 2.7d). Die zwei größten, zusammenhängenden Regionen werden als Hände definiert und zur Gestenerkennung verwendet. Die Gesten werden mittels RGB-Daten der Handregionen und einem Klassifikator basierend auf ANMM-Transformation und Haar-Merkmalen ermittelt. Außerdem kann die Gestenerkennung auf Basis der Tiefen- oder RGBD-Bilder der Hände durchgeführt werden.
Dieses System erkennt verschiedene Gesten und setzt diese zur Interaktion mit einem 3D-Modell ein (siehe Abbildung 2.6d). Der Benutzer kann sich durch die Bewegungen der Hände in alle Richtungen durch das Modell bewegen, es rotieren oder vergrößern bzw. verkleinern (siehe Abbildung 2.7e). Im Gegensatz zu Systemen mit RGB-Daten muss die Entfernung nicht über die Handgröße geschätzt werden, sondern steht mittels RGBD-Daten zur Verfügung. Eine detaillierte Beschreibung der Hand- und Gestenerkennung bzw. Interaktion ist in Abschnitt 2.2.2 und [123–125] zu finden.



(a) RGB-Bild (b) Tiefenbild (c) Schwellwert (d) Hautfarbe (e) Kombination von Farbe und GMM

Abbildung 2.7: Handinteraktion mittels RGBD-Daten von Van den Bergh und Van Gool [125]

Die Arbeit von Park et al. [101] nutzt die beiden eingebauten Kameras der Microsoft Kincet Xbox 360, um RGB- und Tiefendaten mittels strukturiertem Licht und Triangulation zu erhalten und für eine robuste Handerkennung einzusetzen. Zur Segmentierung der Hand wird angenommen, dass sich diese vor dem Körper befindet und eine kleinere Fläche als Körper und Hintergrund im Bild einnimmt. Abbildung 2.8a zeigt das kumulierte Histogramm der Tiefenwerte und hat drei Anstiege (mit Pfeilen markiert im Bild) welche die Handfläche, den Körper und den Hintergrund als größte Fläche darstellen. Aufgrund dieser Informationen kann für die Segmentierung ein Schwellwert zwischen Körper und Hand gewählt werden. Befinden sich weitere Objekte in einer ähnlichen Entfernung, gibt es mehrere Handkandidaten (siehe Abbildung 2.8b). Deshalb wird die endgültige Handregion zusätzlich auf Basis von Hautfarbe und RGB-Daten ermittelt. Aufgrund der niedrigen Auflösung des Tiefenbildes wird für die Ermittlung einer genauen Handkontur ebenfalls auf die höher aufgelösten RGB-Daten zurückgegriffen. Die Handregion auf Basis des Tiefenbildes legt das Suchfenster im RGB-Bild fest und die Hautfarbe wird als Merkmal zur erneuten Segmentierung und Festlegung der Handkontur genutzt (siehe Abbildung 2.8c). Für die Verfolgung der Handbewegung und die Bestimmung der Position in Echtzeit, wird ein auf generalisierter Hough-Transformation [6] basierender Ansatz verwendet. Durch den Einsatz eines Referenzmusters können bewegte Objekte robuster verfolgt werden.



(a) Histogramm der Tiefenwerte (b) Tiefenbild (c) Handkontur

Abbildung 2.8: Handerkennung von Park et al. [101]

KAPITEL 3

Methodisches Vorgehen

In den folgenden Abschnitten wird das methodische Vorgehen beschrieben, um die verschiedenen Hard- und Softwarekomponenten des Prototypen zu entwickeln. Abbildung 3.1 zeigt den grundlegenden Aufbau des Prototypen, welcher in Abschnitt 3.1 näher beschrieben wird.

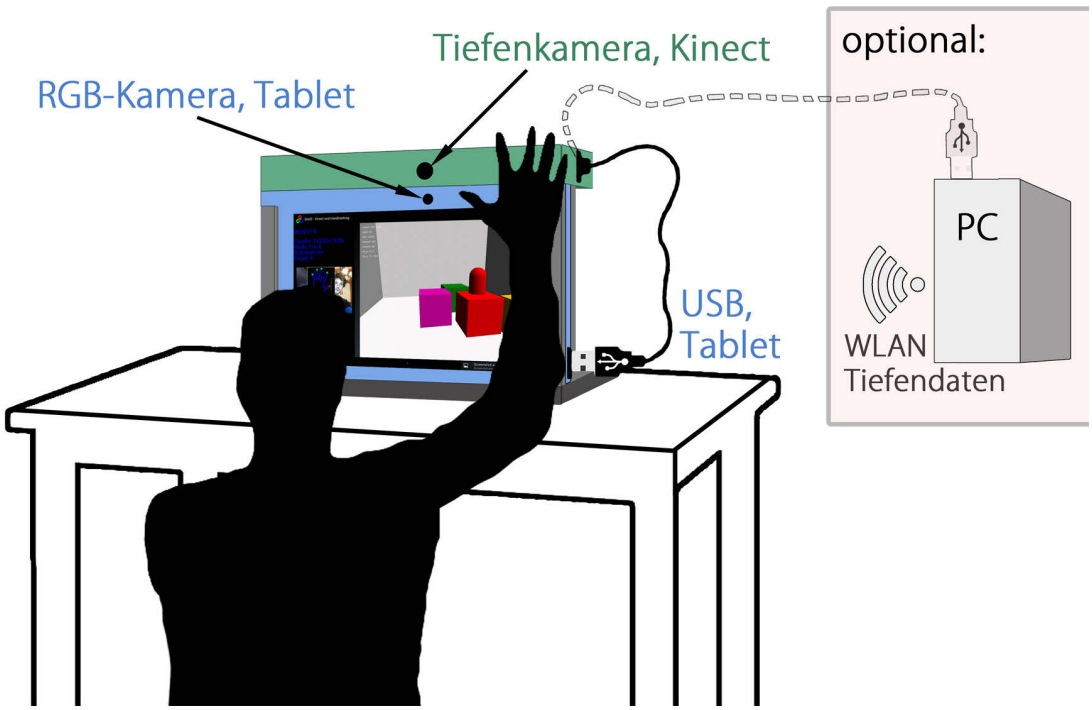


Abbildung 3.1: Überblick Prototypen-Design (schematisch)

Zunächst werden in Abschnitt 3.1 die verschiedenen Technologien und nötigen Schritte für den Einsatz der Hardwarekomponenten behandelt. Abschnitt 3.2 widmet sich der Funktionsweise von Haar-Klassifikatoren, welche zur Hand- und Kopferkennung eingesetzt werden. Abschnitt 3.3 bietet eine Übersicht zu den verschiedenen Ansätzen dieser Arbeit. In Abschnitt 3.4 werden die theoretischen Hintergründe der Ansätze zur Fingergestenerkennung beschrieben und Abschnitt 3.5 widmet sich der Kopferkennung. In Abschnitt 3.6 wird die Ermittlung der 3D-Position der Interaktionsobjekte erläutert und abschließend wird in Abschnitt 3.7 auf die Interaktion und die VR-Szene eingegangen.

3.1 Prototyp

Wie in Abbildung 3.1 zu sehen ist, besteht der Hardware-Prototyp aus einem Tablet und einer Tiefenkamera. Das Tablet wird als einzige und zentrale Recheneinheit und zur Darstellung der mobilen VR-Szene verwendet. Außerdem wird die Frontkamera des Tablets für die Aufnahme des RGB-Bildes eingesetzt (RGB-Daten). Die Tiefenkamera liefert die Tiefendaten, welche mittels USB an das Tablet übertragen werden. Alternativ können die Tiefendaten über WLAN[1] an das Tablet übertragen werden. Die beiden Kameras werden fest miteinander verbunden und extrinsisch kalibriert, um RGBD-Daten erzeugen zu können.

3.1.1 Tablet

Für die Auswahl des mobilen Gerätes wurden folgenden Punkte berücksichtigt:

- Performance (Prozessor, Arbeitsspeicher)
- Frontkamera (Bildqualität, Auflösung)
- Bildschirmgröße
- USB-Anschluss
- Betriebssystem

Das mobile Gerät wird als Recheneinheit und zur Darstellung der VR-Szene verwendet und benötigt dafür einen leistungsstarken Prozessor, sowie genügend Arbeitsspeicher. Die Auflösung und Qualität der Frontkamera muss für die Erkennung der Interaktionsobjekte geeignet sein. Für die Interaktion wird ein gewisser Abstand zur Kamera benötigt, weshalb für eine geeignete Darstellung ein Display größer 7 Zoll bevorzugt wird. Das Gerät benötigt außerdem einen USB-Anschluss, um die Tiefenkamera anschließen zu können. Bei der Wahl des Betriebssystems fließen die Verbreitung des Systems und eine gute Dokumentation für die Anwendungsentwicklung als Faktoren mit ein. Außerdem die Kompatibilität mit anderen Systemen und die Verfügbarkeit von Programmbibliotheken. Die Marktanteile von mobilen Betriebssystemen[2] im zweiten Quartal 2013 werden in Abbildung 3.2 dargestellt.

[1] Wireless Local Area Network.

[2] Die Daten für Smartphones [52] und Tablets [53], sowie die verschiedenen Windows Betriebssysteme, wurden zusammengefasst.

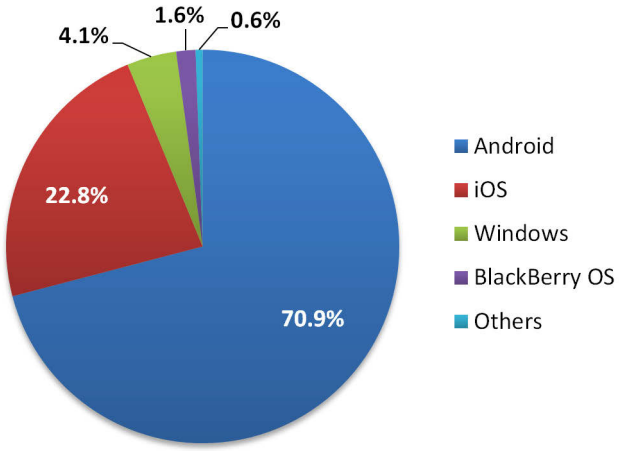


Abbildung 3.2: Marktanteile mobiler Betriebssysteme [52, 53]

Mit über 70 % ist *Android* [35] das am weitesten verbreitete Betriebssystem für mobile Geräte (siehe Abbildung 3.2). Es wird von *Google Inc.* quelloffen entwickelt, ist frei verfügbar und bietet eine gute Dokumentation. Mit Android können die Ansprüche auf Kompatibilität und Verfügbarkeit von Programmbibliotheken erfüllt werden, weshalb die Auswahl auf dieses System fällt.
Das Tablet *Asus Eee Pad Transformer Prime (TF201)* (siehe Abbildung 3.3) ist mit einem 1.4 GHz *Quad-Core Nvidia Tegra 3 (ARM Cortex-A9)* Prozessor, 1 GB Arbeitsspeicher und einer 1.2-Megapixel Frontkamera ausgestattet [103]. Die Bildschirmgröße beträgt 10.1 Zoll und das Tablet verfügt über einen USB-Anschluss. Damit entspricht es allen Anforderungen des zu entwickelnden Prototyps.

Abbildung 3.3: Tablet Asus Eee Pad Transformer Prime (TF201), Bild von [103]

3.1.2 Tiefenkamera

Tiefenkameras werden zur Distanzmessung von Punkten einer Szene verwendet, ohne dafür spezielle Markierungen in der Szene zu benötigen. In Abschnitt 2.2.3 werden die zwei Ansätze ToF und strukturierte Beleuchtung für die Umsetzung von Tiefenkameras erwähnt. In diesem Abschnitt wird genauer auf diese Ansätze eingegangen und die Tiefenkamera dieses Prototyps beschrieben.

Für die Auswahl der Tiefenkamera wurden folgenden Punkte berücksichtigt:

- Preis
- Verbreitung, Kompatibilität und Unterstützung
- Auflösung, Framerate[3]
- Tiefenwerte ab ca. 1m verfügbar
- USB Verbindung

Das Konzept von ToF-Kameras ist die Laufzeitmessung von Licht zur Bestimmung der Entfernung von Punkten [33]. Dafür wird Licht in die Szene gesendet und mittels einer Kamera die Zeit gemessen, die das Licht von der Lichtquelle bis zum Objekt und zurück benötigt. Anhand der exakt bekannten Geschwindigkeit von Licht kann die Entfernung berechnet werden. Essenziell für diese Methode ist eine genaue Zeitmessung. Probleme bei der Messung können durch Mehrfachreflexion und Hintergrundbeleuchtung auftreten. ToF-Systeme ermöglichen die Distanzmessung mit einer hohen Framerate, verglichen mit anderen Systemen (strukturiertes Licht) sind sie allerdings teuer bzw. bieten eine geringe Auflösung bei vergleichbaren Preisen (zum Beispiel *pmd[vision]® CamBoard nano* [104]) [63, 66, 67, 101].
Wird strukturiertes Licht als Ansatz zur Ermittlung der Tiefenwerte verwendet, besteht dieses System grundsätzlich aus einem Projektor und einer Kamera. Mittels Projektor wird ein bestimmtes Lichtmuster in die Szene gesendet und die Abbildung davon von der Kamera aufgenommen. Das Muster kann je nach System und Hersteller variieren. Die Entfernung der Objekte wird durch den Vergleich des Lichtmusters mit der Kameraaufnahme anhand von Triangulation ermittelt.

Basierend auf diesem Ansatz ist die kostengünstige *Microsoft Kinect für Windows* (siehe Abbildung 3.4) verfügbar und wird in dieser Arbeit verwendet. Sie verfügt über Mikrofone, eine RGB-Kamera und einen Motor zur Steuerung der Neigung des Geräts. Diese Komponenten werden in diesem Prototyp nicht verwendet und deshalb nicht näher beschrieben. Außerdem enthält sie einen Infrarot-Projektor und eine Infrarot-Kamera zur Bestimmung der Tiefendaten. Die Kinect wird mittels USB angeschlossen und liefert Tiefenbilder mit einer Auflösung von 640x480 Pixel und einer Framerate von 30 Bildern pro Sekunde [74, 130]. Sie verfügt über zwei

[3] Die Framerate, oder Bildwechselfrequenz, bezeichnet die Anzahl von Bildern die pro Sekunde aufgenommen oder wiedergeben werden.

verschiedene Abstandsbereiche zur Ermittlung von Tiefendaten. Den Standardbereich und den Bereich für nähere Objekte (*Near-Mode*). Laut Herstellerangaben ist die optimale Entfernung 0.8m bis 4m im Standardbereich und 0.4m bis 3m im Near-Mode [73]. Die Kinect entspricht den Anforderungen an das System und wird nachfolgend näher beschrieben.

Abbildung 3.4: Kinect für Windows [77]

3.1.2.1 Funktionsweise der Kinect

Die Kinect basiert, wie zuvor erwähnt, auf dem Ansatz der strukturierten Beleuchtung und verwendet die von *PrimeSense* entwickelte Technologie zur Ermittlung der Tiefendaten [30]. Das Lichtmuster wird hier mittels Infrarot-Projektor ausgesendet und die Abbildung des Musters von der Infrarot-Kamera aufgenommen [130].
Das Lichtmuster entsteht aus der 3×3 Wiederholung eines Punktmusters mit helleren und dunkleren Infrarotpunkten [107]. Dieses Punktmuster besteht aus 211×165 Punkten mit hervorgehobenem Mittelpunkt. Insgesamt ergibt das ein Lichtmuster mit 633×495 Punkten und ist in Abbildung 3.5 zu sehen.

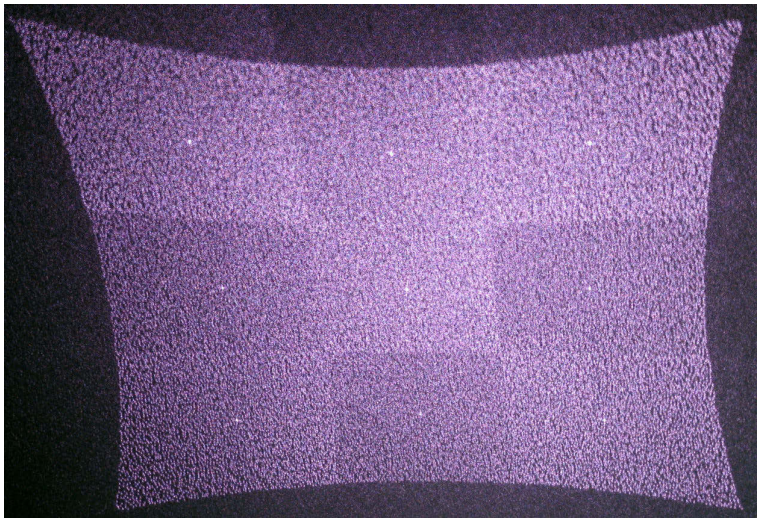


Abbildung 3.5: Lichtmuster der Kinect [58]

Zur Bestimmung des Tiefenwerts Z_k eines Objektpunktes k wird die Aufnahme der Infrarot-Kamera mit einem Referenzmuster verglichen, siehe Abbildung 3.6. Das Referenzmuster ist die Aufnahme des Musters auf einer Referenzebene o in bekannter Entfernung Z_o [57, 58].

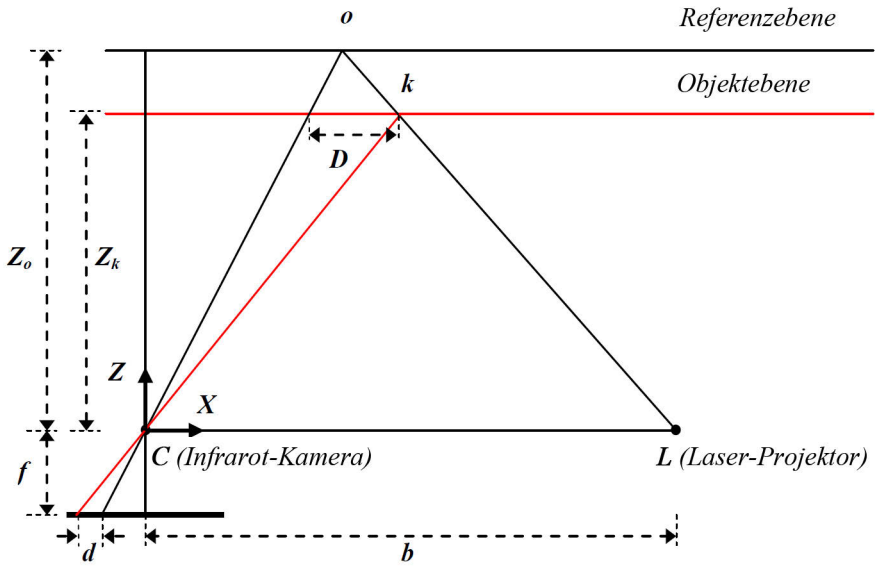
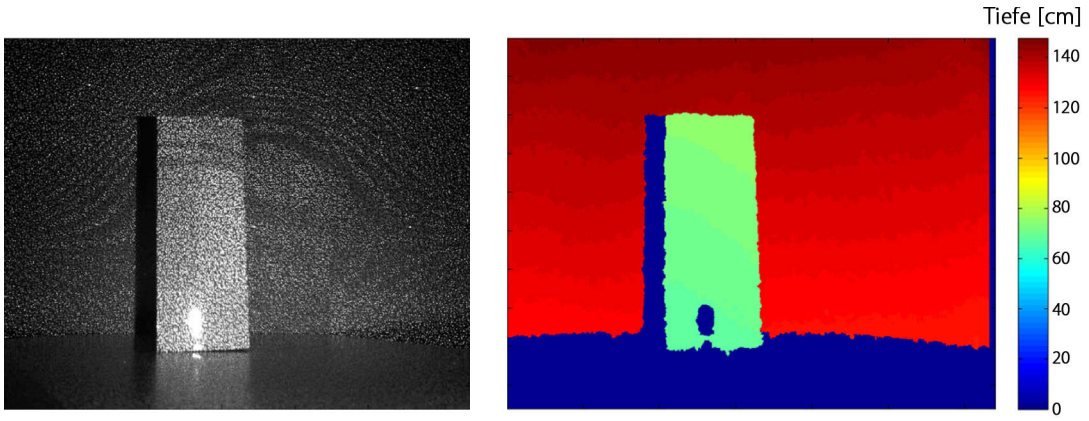


Abbildung 3.6: Bestimmung des Tiefenwerts eines Objektpunktes [57]

Durch die horizontale Versetzung b von Projektor L und Kamera C (siehe Abbildung 3.4) verschieben sich die Punkte des Musters, bei einer Verkleinerung bzw. Vergrößerung der Entfernung zur Kinect, in der x-Achse. Diese Verschiebung (*disparity*) entspricht d im Bildraum (bzw. D im Objektraum).
Mithilfe von d bzw. D und den konstanten Werten der Entfernung der Referenzebene Z_o, dem Abstand von Kamera zu Projektor b und der Brennweite[4] der Infrarotkamera f, lässt sich der Tiefenwert des Objekts Z_k durch Triangulation berechnen, was von Khoshelham und Elberink [57] näher beschrieben wird. Diese Berechnungen zur Ermittlung der Tiefenwerte werden von der Kinect durchgeführt. Der Tiefenwert jedes Pixels ist eine 11-Bit Zahl, wodurch 2048 Tiefenstufen ermöglicht werden.

In Abbildung 3.7a ist das Infrarotbild mit dem, in die Szene projizierten Lichtmuster, zu sehen. Abbildung 3.7b zeigt das daraus resultierende Tiefenbild mit einer Farbzuordnung der Tiefenwerte zur besseren Darstellung. An diesen Abbildungen werden auch die Probleme dieser Methode verdeutlicht [57]. In zu hellen Bereichen (Lichtquelle oder reflektierende Objekte) und in Regionen ohne Lichtmuster (Schattenwürfe von Objekten), siehe Abbildung 3.7a, kann das Muster nicht erkannt und keine Tiefenwerte ermittelt werden. Diese Bereiche sind in Abbildung 3.7b dunkelblau (0 cm) dargestellt. Aufgrund der horizontalen Versetzung von Projektor und Kamera (siehe Abbildung 3.4) entstehen zusätzliche Abschattungen ohne Tiefendaten, da das Licht des Projektors nicht alle für die Kamera einsehbaren Stellen erreicht.

[4] Focal length.



(a) Infrarotbild (b) Tiefenbild

Abbildung 3.7: Projiziertes Lichtmuster und Darstellung der Tiefendaten einer Szene [57]

3.1.3 Verbindung der Tiefenkamera mit dem Tablet

Für die Ermittlung der RGBD-Daten werden die Tiefendaten der Kinect am Tablet benötigt. Dafür wurden zwei Übertragungsvarianten entworfen, welche in Abbildung 3.8 schematisch dargestellt und nachfolgend beschrieben werden.

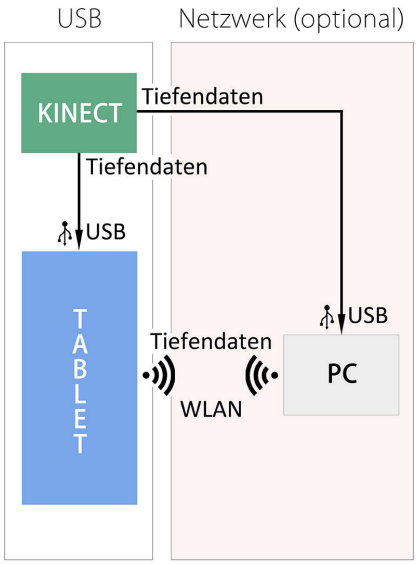


Abbildung 3.8: Übertragung der Tiefendaten zum Tablet

Im Idealfall wird die Kinect per USB mit dem Tablet verbunden und die Tiefendaten direkt übertragen. Da keine offiziellen Treiber der Kinect für Android zur Verfügung stehen, wurde ein alternativer Treiber mit Android-Unterstützung verwendet und neu kompiliert.

Die weiteren Details zur Umsetzung sind in Abschnitt 4.4 beschrieben. Als Alternative wurde zusätzlich eine Netzwerklösung konzipiert, um eine Ausweichmöglichkeit zu schaffen, sollten bei der Implementierung der USB-Variante Komplikationen auftreten. Für die Netzwerkvariante wird die Kinect mit einem PC per USB verbunden und die Tiefendaten über ein Netzwerk (zum Beispiel WLAN) an das Tablet übertragen.

3.1.4 Kalibrierung der Kameras und Erzeugung von RGBD-Daten

Eine reale Kamera und somit die Abbildung einer räumlichen Szene auf eine zweidimensionale Bildebene, lässt sich anhand des Lochkameramodells beschreiben [113,125]. Die Abbildung 3.9 zeigt den geometrischen Zusammenhang zwischen dem 3D-Punkt $M_w = (\mathrm{X_w}, \mathrm{Y_w}, \mathrm{Z_w})^T$ und seiner Projektion $\mathbf{m} = (u, v)^T$ in die Bildebene I.

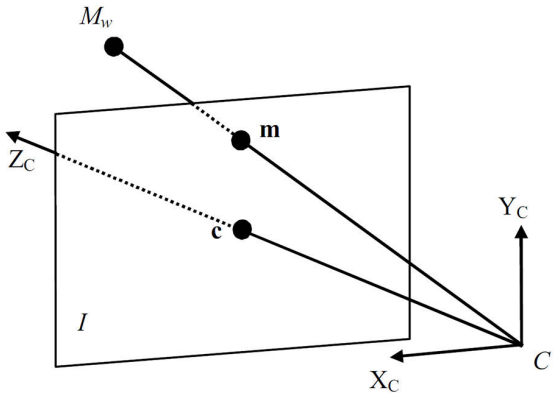


Abbildung 3.9: Prinzip des Lochkameramodells (schematisch) [113]

Zur besseren grafischen Darstellung liegt das Projektionszentrum in Abbildung 3.9 hinter der Bildebene und nicht vor ihr. C ist der Ursprung des Kamerakoordinationssystems und entspricht dem Projektionszentrum der Kamera. Der Kamerahauptpunkt[5] c entspricht dem Schnittpunkt der optischen Achse der Kamera Z_c mit der Bildebene I [113].

Die Transformation vom 3D-Punkt zum 2D-Bildpunkt mittels perspektivischer Projektion wird durch die allgemeine Projektionsmatrix P in Gleichung 3.1 beschrieben [113]:

$$\mathbf{P} = \mathbf{A}\,[\mathbf{R}\,\mathbf{t}] \tag{3.1}$$

Die Matrix P besteht aus der intrinsischen Matrix A und der extrinsischen Matrix $[R\ t]$, welche im nächsten Abschnitt näher beschrieben werden.

[5] Principal point.

3.1.4.1 Intrinsische Matrix

Die intrinsische Matrix wird für die Transformation vom Kamerakoordinationssystem in die Bildebene benötigt. Die intrinsischen Parameter sind kameraspezifisch und verändern sich nicht durch deren Position und Orientierung [67, 112]. Die optischen Eigenschaften und die interne Geometrie einer Kamera werden durch die folgenden intrinsischen Parameter beschrieben [113]:

- Der Kamerahauptpunkt $\mathbf{c} = (c_x, c_y)$
- Die Brennweite $\mathbf{f} = (f_x, f_y)$ beschreibt den Abstand der Bildebene vom Brennpunkt C.
- Der Skalierungsfaktor $\mathbf{k} = (k_u, k_v)$ besteht aus einer horizontalen und vertikalen Skalierung, womit die Transformation der Sensorkoordinaten in diskrete Bildkoordinaten durchgeführt wird.

Aus diesen Parametern wird die intrinsische Matrix gebildet [113] (Gleichung 3.2):

$$\mathbf{A} = \begin{bmatrix} fk_u & 0 & u_0 \\ 0 & fk_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

wobei u_0 und v_0, aufgrund der Startposition der diskreten Bildmatrix in einer Ecke des Bildes, die Verschiebung des Kamerahauptpunktes in den Ursprung des Bildkoordinatensystems beschreiben.

3.1.4.2 Extrinsische Matrix

Die extrinsische Matrix wird für die euklidische Transformation zur Überführung der Bildpunkte in ein anderes Kamerakoordinatensystem benötigt [113]. Durch die Ermittlung der extrinsischen Parameter können die Positionen und Orientierungen der beiden Kameras in einem gemeinsamen, globalen Koordinationssystem festgestellt werden [112]. Dabei liegt eine Kamera im Ursprung dieses Koordinatensystems und die extrinsischen Parameter beschreiben die Rotation und Translation der anderen Kamera. Die extrinsischen Parameter sind [113]:

- Die Rotation der Kamera in der x-, y- und z-Achse, beschrieben durch die 3×3 Rotationsmatrix $\mathbf{R}$
- Die Translation der Kamera in der x-, y- und z-Achse, beschrieben durch den Translationsvektor $\mathbf{t} = (t_x, t_y, t_z)$

Aus diesen Parametern wird die extrinsische Matrix geformt [113] (Gleichung 3.3):

$$\mathbf{D} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0_3^T & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{21} & r_{31} & t_x \\ r_{12} & r_{22} & r_{32} & t_y \\ r_{13} & r_{23} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Damit eine extrinsische Kalibrierung der Kameras durchgeführt werden kann (bzw. valide bleibt) müssen die beiden Kameras fixiert werden (siehe Abschnitt4.1), da sich die extrinsischen Parameter auf die Rotation und Translation der Kamera beziehen.

Für die Kalibrierung und die Bestimmung der intrinsischen und extrinsischen Parameter werden mit beiden Kameras gleichzeitig Fotos eines Schachbrettmusters aufgenommen (siehe Abschnitt 4.2) [67, 112, 125]. Die Position und Orientierung des Musters variiert dabei. Mithilfe dieser Informationen ist es möglich, die Bildpunkte der Tiefenkamera in den RGB-Raum zu transformieren.

3.1.4.3 Transformation

Für die Erzeugung von RGBD-Daten sind die folgenden Schritte erforderlich [17, 62].

Die Tiefendaten werden mittels der intrinsischen Parameter der Kinect in das Koordinatensystem der Kinect umgerechnet (siehe Gleichung 3.4). Mit der inversen intrinsischen Matrix $\mathbf{A}^{-\mathbf{1}}_{\mathbf{Kinect}}$ und dem Punkt des Tiefenbildes $P_T = (\mathrm{X_T}, \mathrm{Y_T}, \mathrm{Z_T})^T$ mit $Z_T = \mathrm{Tiefenwert}(\mathrm{X_T}, \mathrm{Y_T})$ wird der 3D-Punkt P_{3D} berechnet:

$$P_{3D} = \mathbf{A}^{-\mathbf{1}}_{\mathbf{Kinect}}\, P_T \tag{3.4}$$

Die erhaltenen 3D-Punkte werden mithilfe der extrinsischen Parameter in das Koordinationssystem der RGB-Kamera transformiert (Gleichung 3.5). Dafür wird die Rotationsmatrix $\mathbf{R}$ und der Translationsvektor $\mathbf{t}$ verwendet:

$$P_{RGB} = \mathbf{R}\, P_{3D} + \mathbf{t} \tag{3.5}$$

Diese Punkte werden mittels der intrinsischen Parameter der RGB-Kamera $\mathbf{A_{RGB}}$ in die RGB-Bildebene transformiert (Gleichung 3.6). Der Tiefenwert eines RGB-Pixels an der Position (X, Y) entspricht dem Z-Wert von $P_{RGBD}(\mathrm{X}, \mathrm{Y}, \mathrm{Z})$:

$$P_{RGBD} = \mathbf{A_{RGB}}\, P_{RGB} \tag{3.6}$$

Somit kann den Pixeln im RGB-Bild der entsprechende Tiefenwert zugeordnet werden, im Folgenden als *Mapping* bezeichnet, und erhält RGBD-Daten [17, 62].

Die Umsetzung des Mappings wird in Abschnitt 5.4.2 näher beschrieben.

3.2 Objekterkennung mittels Haar-Klassifikatoren

Wie in der Problemstellung (Abschnitt 1.2) beschrieben, werden in dieser Arbeit Klassifikatoren für die Erkennung der Interaktionsobjekte verwendet. Dabei handelt es sich um Haar-Kaskade Klassifikatoren die für die Handerkennung bzw. die Bestimmung der Fingergesten eingesetzt werden. Außerdem wird die Kopferkennung mithilfe eines Klassifikators durchgeführt. Mithilfe dieser Klassifikatoren ist es möglich, die 2D-Position und die Größe des gesuchten Objektes in

einem Graustufenbild zu bestimmen und diese Informationen für die Interaktion zu verwenden. Bei der Klassifikation wird ein Rechteck (grauer Rahmen in Abbildung 3.10) ermittelt, in dem sich das gesuchte Objekt befindet und das die Position und Größe des Objekts wiedergibt.

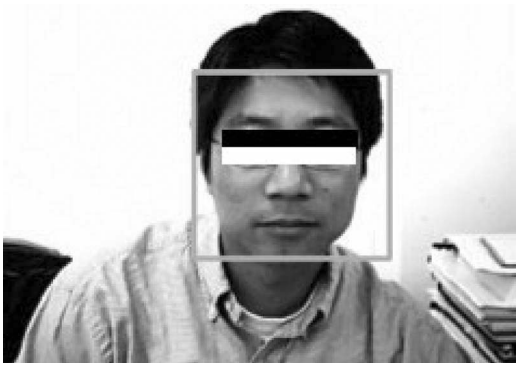


Abbildung 3.10: Bildbereich mit Haar-Merkmal [20]

Im nächsten Abschnitt 3.2.1 werden die Haar-Merkmale und in Abschnitt 3.2.2 die Funktionsweise der Klassifikatoren beschrieben. Abschnitt 4.3 widmet sich dem Erstellen und Trainieren der Klassifikatoren.

3.2.1 Haar-Merkmale und Integralbilder

In dieser Arbeit werden die erweiterten Haar-Merkmale von Lienhart und Maydt [65] für die Objekterkennung verwendet, basierend auf der Arbeit von Viola und Jones [127]. Das von Viola und Jones vorgestellte Verfahren ermöglicht die Erkennung von Objekten in Graustufenbildern in Echtzeit und wurde ursprünglich für die Gesichtserkennung entwickelt.
Haar-Merkmale basieren auf Helligkeitsunterschieden von Objektregionen und werden zur Beschreibung der gesuchten Objekte verwendet. Die Merkmale bestehen aus zwei Rechtecken mit weißen und schwarzen Regionen. In Summe werden 14 verschiedene Merkmaltypen verwendet. Wie in Abbildung 3.11 ersichtlich, kann zwischen den aufrechten Haar-Merkmalen (links) und den um 45 Grad gedrehten Haar-Merkmalen (rechts) unterschieden werden [65].

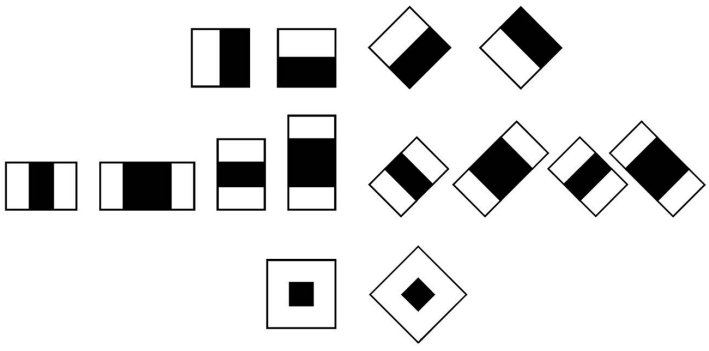


Abbildung 3.11: Aufrechte (links) und rotierte (rechts) Haar-Merkmaltypen [64]

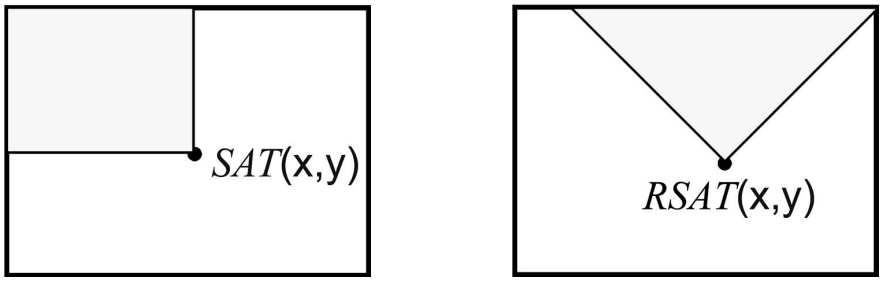
Diese Merkmale werden innerhalb des gewählten Bildbereichs (zum Beispiel das gesuchte Objekt) beliebig skaliert und positioniert (siehe Abbildung 3.10), wodurch sich eine Vielzahl von Möglichkeiten ergibt [65]. Sie werden wie eine Maske über den Bereich gelegt und an allen Positionen der entsprechende Wert des Merkmals ermittelt. Für die Berechnung des Merkmalwerts werden alle vom Merkmal eingeschlossenen Grauwerte addiert, wobei die Grauwerte im schwarzen Bereich negativ und im weißen positiv gewichtet werden [65, 127]. Außerdem wird über die Gewichtung der Größenunterschied der Rechtecke ausgeglichen [65].

Zur Verbesserung der Verarbeitungsgeschwindigkeit verwenden Viola und Jones ein *Integralbild* für die Beschreibung des Bildes und Berechnung des Merkmalwerts. Das Integralbild basiert auf den für Texturmapping verwendeten *Summed Area Tables (SAT)* von Crow [23]. Es wird je ein Integralbild für die Beschreibung der aufrechten und der rotierten Haar-Merkmale benötigt [65]. Mithilfe des Integralbildes können die Rechtecke der Haar-Merkmale effizient berechnet werden. Das Integralbild wird für alle Bildpunkte per Rekursion erstellt und besteht aus der Aufsummierung der Grauwerte des Bildes.
Der Wert des Integralbildes $SAT(x, y)$ an der Position x, y berechnet sich aus der Summe der Grauwerte oberhalb und links des betrachteten Punktes im Originalbild $I(x', y')$ (siehe Abbildung 3.12a und Gleichung 3.7):

$$SAT(x, y) = \sum_{x' \leq x, y' \leq y} I(x', y') \tag{3.7}$$

Auf diesem Konzept basierend wird der Wert des rotierten Integralbildes $RSAT(x, y)$ an der Position x, y berechnet. Dafür werden vom Punkt x, y ausgehend, nach oben bis an die Bildgrenzen von $I(x', y')$, die entsprechenden Grauwerte aufsummiert (siehe Abbildung 3.12b und Gleichung 3.8):

$$RSAT(x, y) = \sum_{y' \leq y, y' \leq y - |x - x'|} I(x', y') \tag{3.8}$$

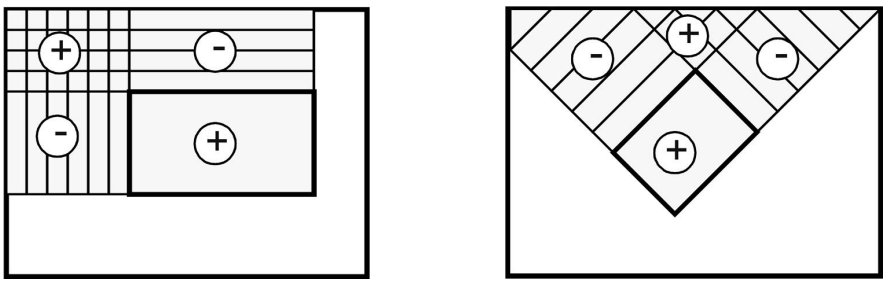


(a) Integralbild (SAT) (b) rotiertes Integralbild (RSAT)

Abbildung 3.12: Konzept der beiden Integralbilder [64]

Mithilfe der Integralbilder können die Werte eines beliebigen Rechtecks mit vier Zugriffen berechnet werden (siehe Abbildung 3.13). Dadurch können die jeweiligen Rechtecke der Haar-Merkmaltypen ermittelt und der Merkmalwert berechnet werden.

Für die Berechnung der aufrechten Merkmale wird das Integralbild (SAT) verwendet. Wie in Abbildung 3.13a dargestellt, werden die entsprechenden Rechtecke addiert bzw. subtrahiert, um das gewünschte Rechteck (in der Darstellung hervorgehoben) zu erhalten. Ebenso wird das rotierte Rechteck ermittelt, allerdings mit dem rotierten Integralbild (RSAT) (siehe Abbildung 3.13b).



(a) Schema für aufrechte Rechtecke (b) Schema für rotierte Rechtecke

Abbildung 3.13: Berechnungsschema zur Ermittlung des hervorgehobenen Rechtecks [64]

3.2.2 Haar-Kaskade Klassifikator

Zur Ermittlung des gesuchten Objekts werden Haar-Merkmale, wie in Abschnitt 3.2.1 beschrieben, verwendet. Für die Detektion wird ein Suchfenster von links oben bis rechts unten (Pixel für Pixel) über das gesamte Bild geschoben. Die Startgröße des Suchfensters wird beim Training des Klassifikators definiert und mit jedem Suchdurchlauf erhöht, bis die Gesamtbildgröße erreicht wird. Der Bildausschnitt innerhalb des Suchfensters wird mithilfe des Klassifikators auf das gesuchte Objekt hin untersucht [64].

Das Gesamtbild besteht meistens aus einem größeren Anteil an Hintergrund und vernachlässigbaren Bildausschnitten, welche mit wenig Rechenaufwand vom Klassifikator festgestellt werden sollen. Meistens enthält nur ein kleiner Bildteil das gesuchte Objekt, der eine genauere Analyse benötigt. Für die Umsetzung eines solchen Klassifikators wird von Viola und Jones [127] ein degenerierter Entscheidungsbaum, eine *Kaskade*, als Struktur genutzt (siehe Abbildung 3.14).

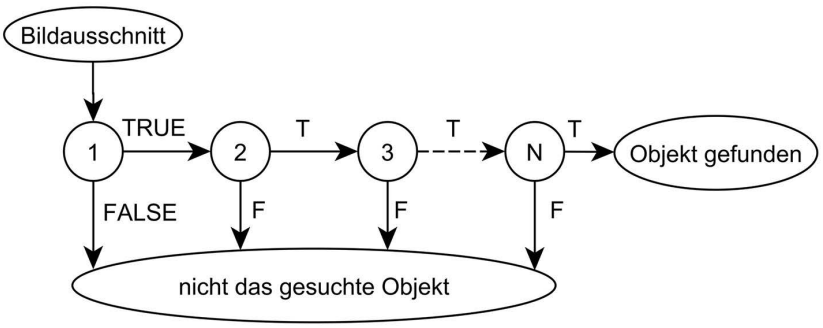


Abbildung 3.14: Kaskadenstruktur des Haar-Klassifikators nach Viola und Jones [127]

Die Kaskade besteht aus mehreren Knoten mit je einem Nachfolger. Pro Knoten wird ein Set von binären Klassifikatoren verwendet, mit denen entschieden wird, ob es sich um das gesuchte Objekt handelt oder nicht (true oder false). Die Komplexität und Berechnungsdauer erhöht sich innerhalb der Kaskade von Knoten zu Knoten. Dafür wird die Anzahl der verwendeten binären Klassifikatoren pro Knoten gesteigert. Damit kann sichergestellt werden, dass nicht relevante Bildteile früh und mit möglichst wenig Rechenaufwand aus der weiteren Suche ausgeschlossen werden.
Beim Haar-Kaskade Klassifikator wird in jedem Knoten der untersuchte Bildausschnitt getrennt bewertet und entschieden, ob es sich um das gesuchte Objekt handelt. Enthält der Bildausschnitt das gesuchte Objekt nicht, wird dieser negativ klassifiziert und nicht weiter untersucht. Wird der Ausschnitt positiv bewertet, untersucht der nächste Knoten den Bildausschnitt, bis am Ende der Kaskade ein positives Ergebnis ein Fund des gesuchten Objekts bedeutet.

Das Set aus Klassifikatoren pro Knoten (*starker Klassifikator*[6]) wird mithilfe eines *Boosting* Lernalgorithmus erstellt. Boosting bedeutet, dass ein starker Klassifikator aus mehreren *schwachen Klassifikatoren*[7] zusammengesetzt wird. Mit jedem starken Klassifikator sollen so zum Beispiel pro Knoten 50 % der Bildausschnitte ohne Objekt entfernt und nur 0.1 % fälschlicherweise ausgeschlossen werden. Besteht der Haar-Kaskade Klassifikator aus 20 Knoten, kann somit eine Falsch-Positiv-Rate von $0.5^{20} \approx 9.6e^{-07}$ und einer Detektionsrate von $0.999^{20} \approx 0.98$ erwartet werden [65].
Ein schwacher Klassifikator bedeutet, dass die Klassifizierung nur besser als der Zufall funktionieren muss. Als schwacher Klassifikator wird ein binärer Klassifikator h_j mit einem Haar-Merkmal zur Beschreibung des Objekts eingesetzt (siehe Gleichung 3.9). Beim Erstellen des binären Klassifikators wird ein Schwellwert θ_j für die Klassifizierung festgelegt. Die Parität p_j gibt die Richtung der Ungleichung vor. Der Merkmalwert f_j des aktuell betrachteten Bildausschnitts wird mit dem Schwellwert θ_j des Klassifikators verglichen und der Ausschnitt dementsprechend klassifiziert [64, 127]:

$$h_j(x) = \begin{cases} 1, & \text{wenn } p_j f_j(x) < p_j \theta_j \\ 0, & \text{sonst} \end{cases} \tag{3.9}$$

Für die Entscheidung des starken Klassifikators $h(x)$ werden die Einzelergebnisse aller beteiligten schwachen Klassifikatoren $h_t(x)$ für die Beurteilung des Bildausschnitts herangezogen (siehe Gleichung 3.10). Viola und Jones verwenden einen, auf dem Boostingalgorithmus *AdaBoost* von Freund und Schapire [31] basierenden Lernalgorithmus. Jedem schwachen Klassifikator h_t wird eine Gewichtung α_t zugeordnet und ein gewichtete Mehrheitsentscheidung getroffen [127]:

$$h(x) = \begin{cases} 1, & \text{wenn } \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0, & \text{sonst} \end{cases} \tag{3.10}$$

Die Gewichtung wird beim Training der Klassifikatoren ermittelt, welches in [31,65,127] theoretisch und in Abschnitt 4.3.1 anhand dem Ablauf des Lernalgorithmus beschrieben wird. Weitere

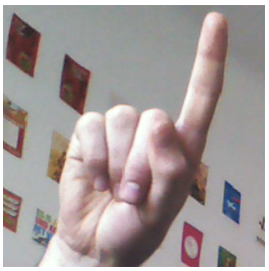
[6] Strong Classifier.
[7] Weak Classifiers.

Boostingalgorithmen wie *Real AdaBoost* [110] und *Gentle AdaBoost* [32] basieren ebenfalls auf AdaBoost, verwenden allerdings schwache Klassifikatoren mit kontinuierlichen Entscheidungswerten und unterscheiden sich dadurch auch in der Berechnung der Gewichtung für die Mehrheitsentscheidung. Die Umsetzung des Trainings und das Erstellen von Haar-Klassifikatoren, sowie der Ablauf der auf AdaBoost basierenden Lernalgorithmen von Viola und Jones [127] und Gentle AdaBoost, werden in Abschnitt 4.3 detailliert erläutert.

3.3 Software-Design

In diesem Abschnitt wird der Aufbau der unterschiedlichen Ansätze zur Fingergestenerkennung dieser Arbeit erklärt und dargestellt. Für die Interaktion werden zwei Fingergesten verwendet, siehe Abbildung 3.15. *Geste 1* besteht aus der flachen Hand mit fünf ausgestreckten Fingern (Abbildung 3.15a) und *Geste 2* aus dem ausgestreckten Zeigefinger (Abbildung 3.15b). Die exakte Definition der Gesten ist abhängig vom verwendeten Ansatz zur Fingergestenerkennung und wird in Abschnitt 3.4 beschrieben.

(a) *Geste 1* - fünf ausgestreckte Finger



(b) *Geste 2* - ausgestreckter Zeigefinger

Abbildung 3.15: Beispielbilder der zwei verwendeten Fingergesten

Aufeinander aufbauend werden die nachfolgenden Ansätze behandelt:

- *Erster Ansatz* (**A1**) zur Fingergestenerkennung mit zwei Klassifikatoren
- *Zweiter Ansatz* (**A2**) mit einem Klassifikator und Fingerspitzenerkennung
- Die Erweiterung der beiden Ansätze (**A1-D** und **A2-D**) mit Tiefendaten

Die Kopferkennung und Interaktion basiert auf A1 und wird deshalb hier nicht gesondert beschrieben.

3.3.1 Erster Ansatz (A1)

A1 verwendet für die Erkennung von zwei unterschiedlichen Gesten zwei Klassifikatoren. Außerdem wird dieser Ansatz auch für die Kopferkennung mithilfe eines Gesichts-Klassifikators verwendet. Eine detaillierte Beschreibung der verwendeten Merkmale für die Erkennung und die Funktionsweise der Klassifikatoren befindet sich in Abschnitt 3.2.

Die RGB-Daten der Tabletkamera sind die Eingabedaten des Moduls *Objekterkennung* (siehe Abbildung 3.16). In *Objekterkennung* suchen zwei Klassifikatoren im RGB-Bild nach passenden Fingergesten (siehe Abschnitt 3.4.1). Wird eine Fingergeste gefunden, wodurch 2D-Position und Größe der Hand bekannt sind, wird in *3D-Position (relativ)* die 3D-Position der Hand durch eine relative Tiefenschätzung (siehe Abschnitt 3.6.1) ermittelt. Mit der erkannten Fingergeste und 3D-Position ist eine 3D-Interaktion mit der *VR-Szene* (siehe Abschnitt 3.7) möglich. Die VR-Szene wird wiederum am *Tablet* ausgegeben.

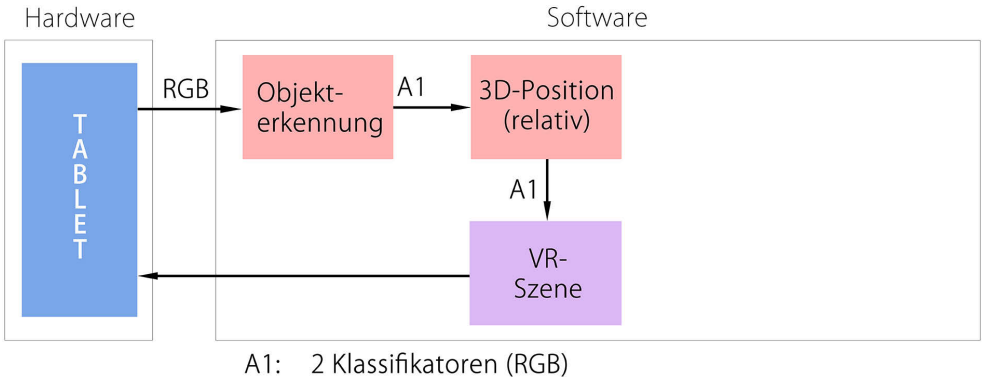


Abbildung 3.16: A1 - erster Ansatz (schematisch)

3.3.2 Zweiter Ansatz (A2)

A2 basiert auf A1, allerdings mit nur einem Klassifikator zur Erkennung der Hand im Modul *Objekterkennung* (siehe Abbildung 3.17). Wurde im RGB-Bild eine Hand gefunden, ist dadurch die 2D-Position und Größe dieser bekannt. In *3D-Position (relativ)* wird wiederum die 3D-Position mittels relativer Tiefenschätzung ermittelt (siehe Abschnitt 3.6.1). Im Modul *Fingerspitzen* wird zuerst die Handkontur und mithilfe dieser die Fingerspitzen ermittelt. Die Anzahl an gefundenen Fingerspitzen definiert die Geste (siehe Abschnitt 3.4.2). Diese wird mit der 3D-Position zur Interaktion an die *VR-Szene* (siehe Abschnitt 3.7) übergeben, welche am *Tablet* angezeigt wird.

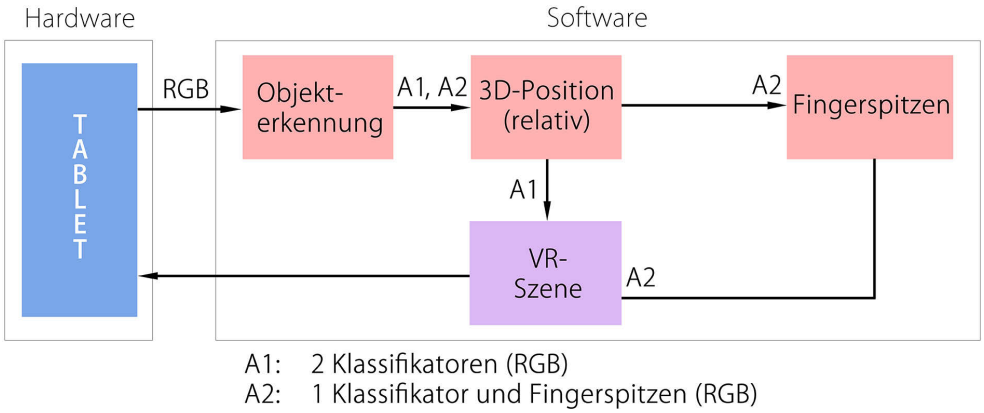


Abbildung 3.17: A1 und A2 - Erster Ansatz und zweiter Ansatz (schematisch)

3.3.3 Erweiterung mit Tiefendaten (A1-D und A2-D)

Aufbauend auf den zuvor beschriebenen Ansätzen A1 und A2, verwenden A1-D und A2-D die Tiefendaten der Kinect zur Ermittlung der 3D-Position bzw. Fingerspitzenerkennung. Diese Erweiterung ist in Abbildung 3.18 dargestellt.
Nach der Bestimmung der Hand bzw. Fingergeste in *Objekterkennung* mittels Klassifikator werden in *RGBD* die Tiefendaten der *Kinect* zu den RGB-Daten der Hand hinzugefügt und RGBD-Daten erstellt (siehe Abschnitt 3.1.4). Außerdem wird die erkannte Hand mithilfe der Tiefendaten segmentiert (Objekte vor bzw. hinter der Hand werden so herausgefiltert). Durch die Tiefendaten ist die 3D-Position der Hand in *3D-Position (absolut)* bekannt und muss nicht geschätzt werden (siehe Abschnitt 3.6.2). Für die Ermittlung der Handkontur und Fingerspitzen (A2-D) wird in *Fingerspitzen* die mittels Tiefendaten segmentierte Hand verwendet (siehe Abschnitt 3.4.3)

Die Zusammenfassung und den Ablauf der verschiedenen Ansätze dieser Arbeit wird in Abbildung 3.18 dargestellt.

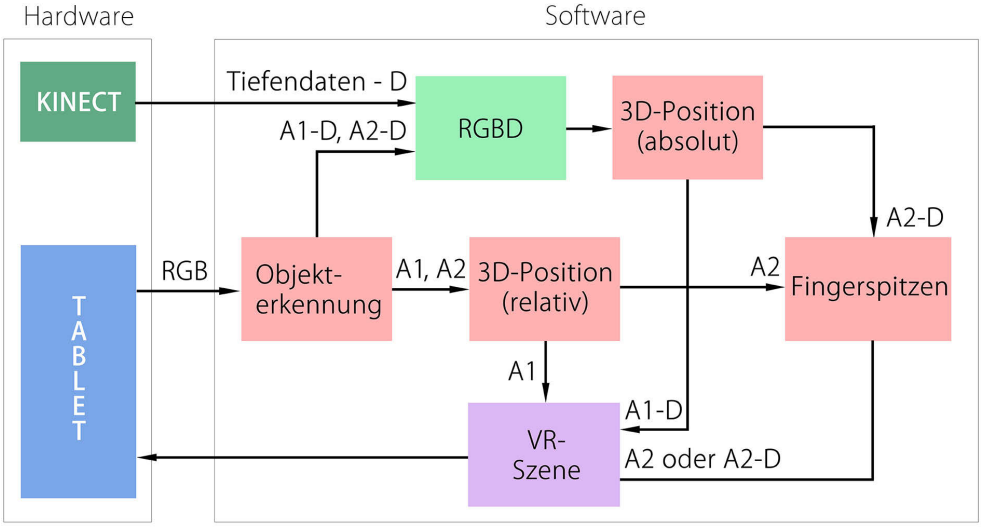


Abbildung 3.18: Überblick Software-Design (schematisch)

3.4 Erkennung der Fingergesten

Es werden zwei verschiedene Fingergesten für die Interaktion mit der VR-Szene eingesetzt (siehe Abschnitt 3.7). Wie zuvor in Abschnitt 3.3 beschrieben, werden für die Ermittlung der Fingergesten zwei Ansätze mit RGB-Daten und eine erweiterte Variante mit RGBD-Daten verwendet. In den folgenden Abschnitten wird auf diese Ansätze im Detail eingegangen.

3.4.1 A1 (RGB)

A1, siehe Abschnitt 3.3.1 (Schema), verwendet zwei verschiedene Haar-Kaskade Klassifikatoren zur Unterscheidung der Fingergesten. Der *erste Klassifikator* (**K1**) ist darauf trainiert *Geste 1* zu erkennen und der *zweite Klassifikator* (**K2**) *Geste 2* (siehe Abbildung 3.15). Das Training der Klassifikatoren wird in Abschnitt 4.3 näher beschrieben. Durch die Verwendung von zwei getrennten Klassifikatoren ist bei der Detektion eine eindeutige Zuordnung der Geste möglich und keine weitere Bildverarbeitung bzw. Fingerspitzenerkennung notwendig. Außerdem ist durch die Handerkennung die Handgröße und 2D-Position bekannt.

3.4.2 A2 (RGB)

A2, siehe Abschnitt 3.3.2 (Schema), verwendet einen Haar-Kaskade Klassifikator für die Erkennung der Hand und die Fingerspitzenerkennung zur Ermittlung der Fingergesten. Der Hand-Klassifikator (**K3**) verwendet den ausgestreckten Daumen und die Handunterseite als Trainingsmerkmale (siehe Abschnitt 4.3.2), die Anzahl bzw. Ausrichtung der übrigen Finger wird nicht berücksichtigt. Durch die Detektion der Hand mittels K3 ist die Handgröße und 2D-Position bekannt und die Fingerspitzenerkennung kann auf diese Region begrenzt werden. Zur Bestimmung der Fingergesten wird die Anzahl der erkannten Fingerspitzen verwendet. Vier bis fünf Fingerspitzen entsprechen *Geste 1* und ein bis zwei Fingerspitzen *Geste 2*.
Für die Fingerspitzenerkennung wird zuerst die Handkontur ermittelt, anschließend wird die Kontur verbessert und auf Fingerspitzen hin untersucht. Die Fingerspitzenerkennung wird in der von K3 ermittelten Handregion durchgeführt und nachfolgend genauer beschrieben.

3.4.2.1 Kontur ermitteln

Für die korrekte Fingerspitzenerkennung ist es wichtig, eine robuste Handkontur zu verwenden, um Fehler bei der Erkennung minimieren zu können. Sie sollte im Optimalfall nur aus der tatsächlichen Hand bestehen und eine in sich geschlossene Kontur darstellen (Abbildung 3.19a). Hierzu werden entweder die Kanten der Handregion ermittelt oder die Hand im Bildausschnitt vom Hintergrund getrennt (segmentiert). Das daraus resultierende binäre Bild (siehe Abbildung 3.19b) besteht entweder aus den Kanten oder der segmentierten Hand (weiß) und dem Hintergrund (schwarz).

(a) Kontur der Hand (blau) (b) Binäres Bild der segmentierten Hand

Abbildung 3.19: Kontur ermitteln mittels Schwellwert (A2-T)

A2 wird in drei Gruppen unterteilt:

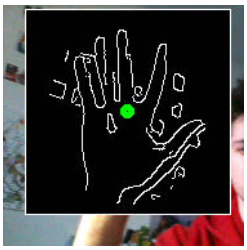
A2-C: Canny-Algorithmus zur Kantendetektion [19]

A2-H: Segmentierung durch Schwellwert mit automatischer Anpassung, basierend auf den HSV[8]-Werten der Hand

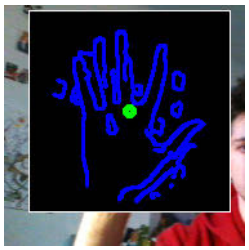
A2-T: Segmentierung durch Schwellwert mit manueller Anpassung durch den Benutzer

3.4.2.2 A2-C

Mithilfe des von Canny [19] entwickelten Algorithmus, werden in *A2-C* die Kanten im untersuchten Graustufenbild (Handregion) ermittelt. In einem ersten Schritt wird der Ausschnitt mit einem Filter geglättet, zwecks Unterdrückung von Bildrauschen. Für die Kantendetektion werden die partiellen Ableitungen in x- bzw. y-Richtung benötigt. Die beiden Bilder werden durch eine Faltung mittels Sobeloperator[9] ermittelt. Darin sind die horizontalen bzw. vertikalen Kanten hervorgehoben. Diese beiden Bilder werden in weiteren Schritten kombiniert und ergeben das Bild mit den ermittelten Kanten. Das Kantenbild entspricht im Optimalfall der gesuchten Handkontur, in Abbildung 3.20a enthält es auch Hintergrundobjekte. Die Handkontur (blau) wird aus dem Kantenbild erzeugt (siehe Abbildung 3.20b).



(a) Kantenbild mittels Canny



(b) Kontur (blau) mittels Kantenbild

Abbildung 3.20: Kontur ermitteln mittels Canny (A2-C)

3.4.2.3 A2-H

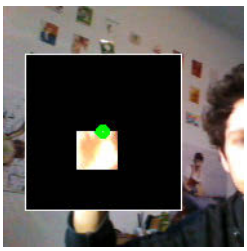
Durch die Trennung von Farbe und Helligkeit in verschiedene Kanäle wird der HSV-Farbraum häufig für die Beschreibung von Hautfarbe und die Segmentierung von hautfarbenen Objekten verwendet [59, 99, 118, 138]. Als Schwellwerte werden in *A2-H* die Minimal- und Maximalwerte der einzelnen Kanäle verwendet, wobei die oberen und unteren 10 % entfernt werden um Ausreißer zu eliminieren. Diese Schwellwerte werden für die Segmentierung der Hand, bzw. die Erzeugung eines binären Bildes, bestehend aus der Hand (weiß) und Hintergrund (schwarz) verwendet (siehe Abbildung 3.21a). Sie werden für die aktuelle Handregion berechnet. Dabei wird nur ein kleiner Ausschnitt der Handregion genutzt (siehe Abbildung 3.21b), um sicher gehen zu

[8] HSV steht für den Farbwert (Hue), die Farbsättigung (Saturation) und den Hellwert (Value), womit Farben im HSV-Raum beschrieben werden.

[9] Ein einfacher Kantendetektions-Filter zur Faltung mittels einer 3×3 Matrix.

können, dass die Hautfarbe und nicht ein Teil vom Hintergrund miteinbezogen wird (wodurch der Schwellwert verfälscht werden könnte). Aus dem binären Bild der segmentierten Hand wird wiederum die Handkontur erstellt.

(a) Binäres Bild der segmentierten Hand



(b) Handausschnitt für HSV-Schwellwert

Abbildung 3.21: Kontur ermitteln mittels HSV-Schwellwert (A2-H)

3.4.2.4 A2-T

A2-T wird in Abbildung 3.19 dargestellt und verwendet einen einfachen, konstanten Grauwert-Schwellwert für die Segmentierung der Hand. Der Schwellwert kann vom Benutzer adaptiert werden, um die aktuellen Lichtbedingungen etc. berücksichtigen zu können. Mithilfe der segmentierten Hand wird anschließend die Handkontur erstellt.

3.4.2.5 Optimierung der ermittelten Kontur

Bei der vom Klassifikator definierten Handregion handelt es sich, wie erwähnt, um ein Rechteck. Da die Handfläche besser von einer Ellipse angenähert wird, wurde eine elliptische Maske in den Bildausschnitt gelegt (siehe Abbildung 3.22a). Damit kann ein großer Teil des Hintergrunds von vornherein ausgeschlossen und so die Ermittlung der Kontur verbessert werden. Außerdem werden Konturen (siehe Abbildung 3.22b), die eine zu kleine Fläche im Verhältnis zur Handgröße umschließen (grau) mittels Schwellwert entfernt, da es sich dabei mit großer Wahrscheinlichkeit nicht um die Hand handelt. Aus den restlichen Konturen wird eine einzige Kontur gebildet und als Handkontur angenommen.

(a) Elliptische Maske

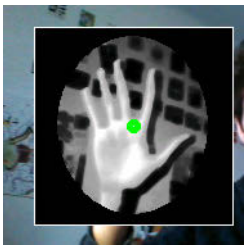
(b) Kleine Konturen (grau) vernachlässigen

Abbildung 3.22: Kontur mittels Maske und Größenschwellwert verbessern

Weitere Schritte zur Verbesserung der Kontur in Abbildung 3.23a, sind vorgeschaltete Bildverarbeitungsfunktionen und morphologische Operationen, um Störungen aus dem Bild zu entfernen bzw. die Segmentierung zu verbessern (siehe Abbildung 3.23). Dafür wird wahlweise auf die nachfolgenden Funktionen zurückgegriffen, welche vom Benutzer eingestellt und aktiviert werden können:

- Mittels *Histogrammausgleich*[10] wird der Kontrast des Graustufenbildes verbessert (vergleiche Abbildung 3.22a mit Abbildung 3.23b), indem die Grauwerte auf den gesamten Wertebereich aufgeteilt bzw. gestreckt werden [16]. Dies soll unter gewissen Bedingungen zu einer verbesserten Segmentierung beitragen.
- Die morphologische Funktion *Dilatation* [114] benutzt ein strukturiertes Element (zum Beispiel ein Kreis), um helle Regionen zu vergrößern und dunkle zu verkleinern. Dadurch wird die segmentierte Hand (weiß) vergrößert und unterbrochene Handregionen können verbunden werden, womit eine durchgehende Kontur ermöglicht wird (siehe Abbildung 3.23c).
- Die morphologische Operation *Erosion* [114] ist das Gegenstück zur Dilatation und dient dazu, helle Regionen zu verkleinern bzw. dunkle zu vergrößern. Damit können zum Beispiel dicht beieinanderliegende Fingerregionen verdünnt werden und eine Unterscheidung der Fingerspitzen erleichtert bzw. eine Verschmelzung dieser vermieden werden (siehe Abbildung 3.23d).

(a) Kontur ohne morphologische Operationen



(b) Histogrammausgleich

(c) Dilatation

(d) Erosion

Abbildung 3.23: Kontur mittels Histogrammausgleich und Dilatation bzw. Erosion verbessern

[10] Histogram equalization.

3.4.2.6 Fingerspitzenerkennung

Für die Fingerspitzenerkennung werden aus der zuvor ermittelten Handkontur die konvexe Hülle und die *convexity defects*[11] berechnet. Convexity defects entsprechen den schraffierten Flächen (A-H) in Abbildung 3.24 und beschreiben die Abweichungen der konvexen Hülle (schwarze, äußere Linie) von der Handkontur [16].

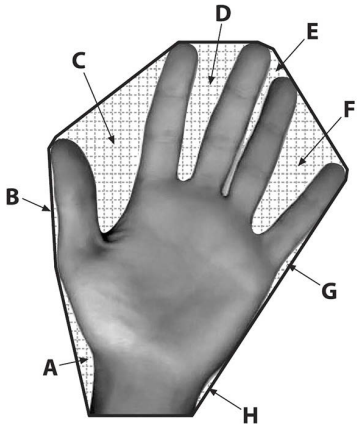
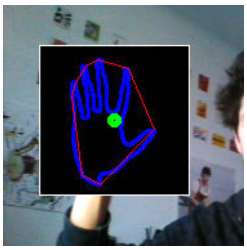
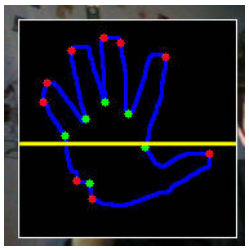


Abbildung 3.24: Konvexe Hülle und convexity defects [16]

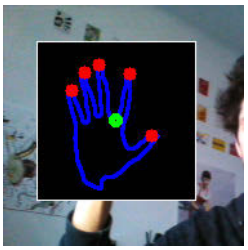
Abbildung 3.25 stellt die Schritte der Fingerspitzenerkennung von der konvexen Hülle und den convexity defects bis zu den endgültigen Fingerspitzen dar. Die konvexe Hülle entspricht der roten Linie in Abbildung 3.25a. Ein convexity defect (Abbildung 3.25b) wird durch einen Start- und Endpunkt (rot) auf der konvexen Hülle beschrieben. Zusätzlich wird der Konturpunkt mit dem größten Abstand von der konvexen Hülle (grün) und dessen Entfernung zur Hülle (entspricht der Tiefe der Ausbuchtung) angegeben [16].
Die Start- und Endpunkte (rot) der convexity defects in Abbildung 3.25b sind Fingerspitzenkandidaten. Der Abstand und die Position der Kandidaten zum Handmittelpunkt (gelbe Linie) werden als Kriterien zur endgültigen Ermittlung der Fingerspitzen herangezogen, die Daumenspitze wird gesondert ermittelt. Der Kandidat mit dem größten x-Wert wird als Daumen der Hand bestimmt. Abbildung 3.25c zeigt die schlussendlich ermittelten Fingerspitzen.



(a) Konvexe Hülle (rot)



(b) Convexity Defects



(c) Fingerspitzen

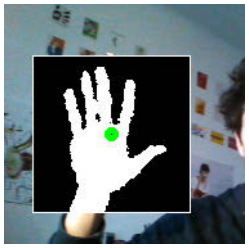
Abbildung 3.25: Von der konvexen Hülle zu den Fingerspitzen

[11] Von der konvexen Hülle abweichende Ausbuchtungen in Bezug zur Kontur.

3.4.3 A1-D und A2-D (RGBD)

Als Erweiterung von A1 und A2 nutzen A1-D und A2-D die Tiefeninformationen der Kinect für die Ermittlung der 3D-Position und A2-D für die robuste Segmentierung der Hand.
Die Tiefenposition der Hand wird in A1-D und A2-D durch eine absolute Schätzung ermittelt (siehe Abschnitt 3.6.2). In A2-D wird die Hand mithilfe der Tiefenwerte robust segmentiert. Dazu werden alle Pixel, die sich hinter der Hand befinden und somit einen größeren Tiefenwert als die Hand besitzen, mittels Schwellwert ausgeblendet [125] (siehe Abbildung 3.26a) und ein binäres Bild erzeugt (siehe Abbildung 3.26b. Die weiteren Schritte sind identisch zu A2 (siehe Abschnitt 3.4.2).

(a) Pixel hinter der Hand ausblenden



(b) Binäres Bild mittels Tiefendaten

Abbildung 3.26: Hand segmentieren mittels Tiefendaten (A2-D)

3.5 Erkennung des Kopfs

Für die Kopferkennung wird ein Haar-Kaskade Klassifikator verwendet. Die Detektion liefert die Kopfgröße und die 2D-Position des Kopfes. Nach der Ermittlung der relativen 3D-Position (Abschnitt 3.6.1), kann diese für die 3D-Interaktion mit der VR-Szene verwendet werden. Mit den Informationen zur Position des Kopfes wird die Perspektive der virtuellen Szene gesteuert (HCP), um einen 3D-Effekt zu erzeugen (siehe Abschnitt 3.7).

3.6 Ermittlung der 3D-Position

Die 2D-Position des Objekts wird mithilfe der in Abschnitt 3.3 beschriebenen Ansätze ermittelt. Die Detektion mittels Klassifikator liefert den Objektmittelpunkt (siehe Abbildung 3.27), welcher als 2D-Position des Objekts herangezogen wird. Die für eine 3D-Interaktion benötigte Tiefenposition des Interaktionsobjekts kann über eine relative oder absolute Schätzung ermittelt werden. Nachfolgend wird näher auf diese zwei Schätzungsmethoden eingegangen. In Abschnitt 3.7.1 wird die Transformation der 3D-Position in das Koordinatensystem der VR-Szene beschrieben.

3.6.1 Relative Schätzung (RGB)

Für eine relative Tiefenschätzung wurden in dieser Arbeit zwei Methodiken entwickelt und implementiert. Entweder erfolgt die Schätzung mithilfe der ermittelten Objektgröße oder anhand

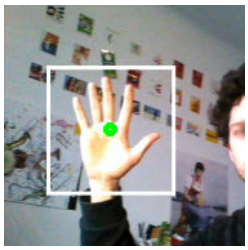
des maximalen Grauwerts des Objekts (siehe Abbildung 3.27). Eine relative Tiefenschätzung bedeutet, dass nicht die reale Tiefenposition des Objekts ermittelt wird (wie bei der absoluten Schätzung in Abschnitt 3.6.2), sondern die Entfernungsveränderung des Objekts in Bezug zur Kamera.

3.6.1.1 Objektgröße

Die Objekterkennung mittels Haar-Kaskade Klassifikator liefert neben der 2D-Position auch die Größe des Objekts. Der Klassifikator liefert ein Rechteck, in dem sich das gesuchte Objekt befindet. Die Objektgröße entspricht der Größe des in Abbildung 3.27a gezeigten Rechtecks (weißer Rahmen). Die Tiefenposition des Objekts wird über die Größe relativ geschätzt. Umso größer das Objekt, desto näher befindet es sich zur Kamera und umgekehrt [109]. Zu Beginn wird die minimale und maximale Objektgröße festgelegt und der minimalen bzw. maximalen Entfernung zugeordnet. Dadurch wird die Beziehung zwischen Objektgröße und Entfernung hergestellt, wodurch sich die Tiefenposition des Objekts schätzen lässt.

3.6.1.2 Maximaler Grauwert des Objekts

Für die Tiefenschätzung mittels Grauwert wird der maximale Grauwert des Objekts herangezogen. Um den Hintergrund zu minimieren, wird eine elliptische Maske (schwarz) über das Rechteck gelegt und nur der innere Bereich verwendet (siehe Abbildung 3.27b). Für die relative Tiefenschätzung wird angenommen, dass das Objekt umso heller ist (größerer Grauwert), desto näher es der Kamera kommt. Der minimale und maximale Grauwert wird zu Beginn festgelegt und entspricht der minimalen bzw. maximalen Entfernung. Mithilfe der damit hergestellten Beziehung zwischen Grauwert und Entfernung wird die Tiefenposition des Objekts geschätzt. Aufgrund der Lichtabhängigkeit dieser Methode kann der Benutzer während der Interaktion den Grauwertbereich verändern und somit an die Lichtsituation anpassen.



(a) Objektgröße (weißes Rechteck)

(b) Maximaler Grauwert des Objekts

Abbildung 3.27: Relative Tiefenschätzung

3.6.2 Absolute Schätzung - Tiefenkamera (RGBD)

Durch den Einsatz einer Tiefenkamera und dem entsprechenden Mapping (Abschnitt 3.1.4) stehen RGBD-Daten zur Verfügung. Mithilfe der bekannten 2D-Position des Objekts im RGB-Bild kann die entsprechende Tiefeninformation aus den RGBD-Daten ermittelt und für die 3D-Interaktion verwendet werden. Somit wird die 3D-Position absolut und robust ermittelt.

3.7 Darstellung und Interaktion - VR-Szene

Die virtuelle Szene besteht in dieser Arbeit aus einem, durch Wände und Boden (weiß/grau) begrenzten 3D-Raum (siehe Abbildung 3.28). Dieser Raum ist mit verschiedenfarbigen Würfelobjekten zur Interaktion befüllt. Die virtuelle Kamera wurde zentral vor den Würfelobjekten platziert und repräsentiert die Position des Betrachters. Als Ursprung des VR-Koordinatensystems wurde der Startpunkt der virtuellen Kamera gewählt.
Für eine 3D-Interaktion mit der VR-Szene wird die ermittelte 3D-Position der Hand bzw. des Kopfs in die virtuelle 3D-Position transformiert. Diese Transformation der Hand- und Kopfpositionen in das VR-Koordinatensystem wird nachfolgend in Abschnitt 3.7.1 erläutert. Durch die 3D-Position der Hand wird die virtuelle Hand gesteuert und die 3D-Position des Kopfs lenkt die virtuelle Kamera. Die Interaktionsmöglichkeiten werden in Abschnitt 3.7.2 behandelt.

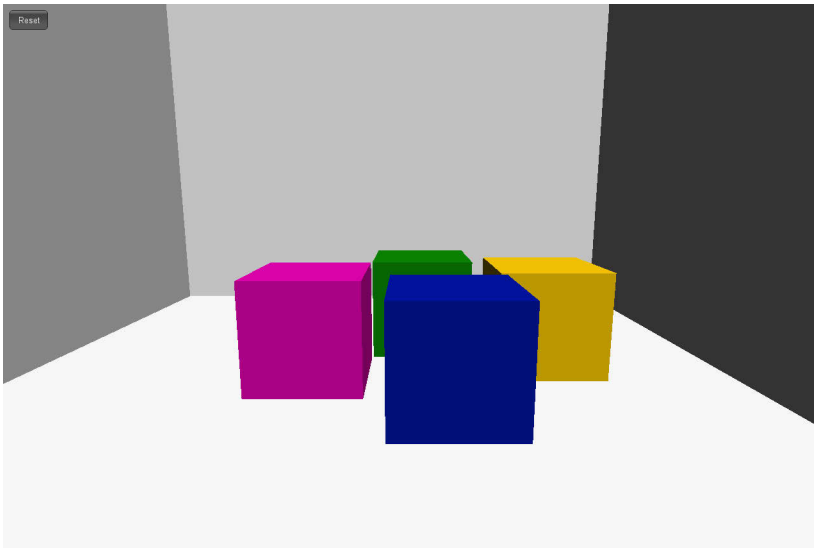
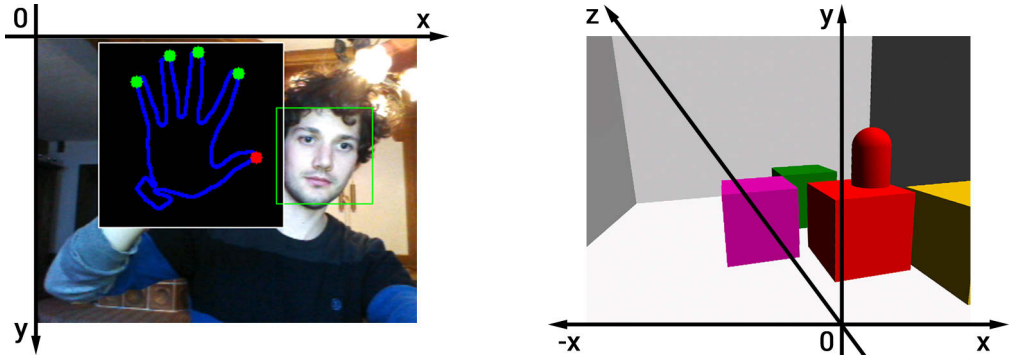


Abbildung 3.28: VR-Szene

3.7.1 Transformation der 3D-Position in das VR-Koordinatensystem

Die zur Verfügung stehende 2D-Position der Interaktionsobjekte im RGB-Bild wird zur Ermittlung der x- und y-Position im VR-Koordinatensystem verwendet. Die z-Position wird, abhängig von der gewählten Methode für die Ermittlung der Tiefenposition (siehe Abschnitt 3.6), aus der Objektgröße, dem maximalen Grauwert oder dem Tiefenwert des Objekts ermittelt. Für die korrekte Transformation müssen die unterschiedlichen Skalierungen und Ursprungspositionen des RGB-Koordinatensystems (linke obere Ecke) und VR-Koordinatensystems (Startpunkt der virtuellen Kamera) beachtet werden (siehe Abbildung 3.29).



(a) RGB-Koordinatensystem (b) VR-Koordinatensystem

Abbildung 3.29: Koordinatensysteme des RGB-Bilds und der VR-Szene

In den nachfolgenden Abschnitten wird die Transformation der 3D-Position der Hand und des Kopfs im Detail beschrieben.

3.7.1.1 Transformation der Handposition

Die 2D-Position der realen Hand $h(x)$ bzw. $h(y)$ wird mit konstanten Faktoren in die 2D-Position der virtuellen Hand $h_{vr}(x)$ bzw. $h_{vr}(y)$ transformiert (Gleichung 3.11 bzw. Gleichung 3.12). Die Konstanten S_{xh} und S_{yh} werden für die Umrechnung der 2D-Position benötigt und repräsentieren die Größenverhältnisse der Skalierungen von RGB- und VR-Koordinatensystem. Sie werden anhand der maximal erreichbaren 2D-Positionswerte und den Grenzen der VR-Szene bestimmt, sodass sichergestellt werden kann, dass diese Begrenzungen mit der virtuellen Hand erreichbar sind. Außerdem wird der Ursprung des RGB-Koordinatensystems mithilfe der Translationskonstanten T_{xh} und T_{yh} in den Ursprung des VR-Koordinatensystems verschoben.

$$h_{vr}(x) = (h(x) - T_{xh}) * S_{xh} \tag{3.11}$$

$$h_{vr}(y) = (h(y) - T_{yh}) * S_{yh} \tag{3.12}$$

Die Tiefenposition der Hand $h(z)$ wird mithilfe der Konstanten S_{zh} in die z-Position $h_{vr}(z)$ der virtuellen Szene umgerechnet (Gleichung 3.13). S_{zh} bildet das Verhältnis zwischen dem maximalen z-Wert[12] der virtuellen Szene und der Differenz der minimalen und maximalen Tiefenposition ab. Sie dient dazu, den zur Verfügung stehenden Interaktionsraum eines sitzenden Benutzers in der virtuellen Szene abzubilden. Die minimale und maximale Tiefenposition ist abhängig von der Methode zur Ermittlung der 3D-Position. Mithilfe der maximal möglichen Handgröße, dem maximalen Grauwert oder dem kleinstmöglichen Abstand zur Kamera, soll mit der virtuellen Hand die hintere Abgrenzung der virtuellen Szene erreicht werden können und

[12]Der minimale z-Wert der virtuellen Szene ist 0 und somit für das Verhältnis vernachlässigbar.

umgekehrt die Vordere. Die Translationskonstante T_{zh} entspricht der minimalen Tiefenposition und dient dazu die Tiefenposition in den Ursprung des VR-Koordinatensystems zu verschieben.

$$h_{vr}(z) = (h(z) - (T_{zh}) * S_{zh} \tag{3.13}$$

3.7.1.2 Transformation der Kopfposition

Die Transformation der 3D-Position des Kopfs k wird in gleicher Weise durchgeführt wie die zuvor beschriebene Umrechnung der Handposition (siehe Gleichungen 3.11 bis 3.13).
Die Kopfbewegung ist an die Position der virtuellen Kamera k_{vr} gekoppelt, allerdings wird die x- und y-Position des Kopfes für die Rotation der Kamera verwendet. Dafür werden $k(x)$ und $k(y)$ mit den Konstanten S_{xk} und S_{yk} in die Rotationswinkel $k_{vr}(x_{rot})$ und $k_{vr}(y_{rot})$ der virtuellen Kamera transformiert (Gleichung 3.14 bzw. Gleichung 3.15). S_{xk} und S_{yk} werden durch die maximal erreichbaren 2D-Positionswerte so bestimmt, dass die Gleichungen die gewünschten maximalen Rotationswinkel der virtuellen Kamera ergeben. Mithilfe der Translationskonstanten T_{xk} und T_{yk} wird der Ursprung des RGB-Koordinatensystems wiederum in den Ursprung des VR-Koordinatensystems verschoben.

$$k_{vr}(x_{rot}) = (k(x) - T_{xk}) * S_{xk} \tag{3.14}$$

$$k_{vr}(y_{rot}) = (k(y) - T_{yk}) * S_{yk} \tag{3.15}$$

Zusätzlich wird auch die x- und y-Position $k_{vr}(x)$ und $k_{vr}(y)$ der virtuellen Kamera verändert (Gleichung 3.16 bzw. Gleichung 3.17), um die reale Kopfbewegung des Betrachters nachzuahmen und somit einen realistischeren 3D-Effekt zu erzeugen. Dafür wird die Rotation um die y-Achse mit einer Translation in der x-Achse unterstützt und umgekehrt. Diese ist abhängig von den Rotationswinkeln $k_{vr}(y_{rot})$ und $k_{vr}(x_{rot})$ aus Gleichung 3.15 und 3.14. Die Verschiebung wird mithilfe der Skalierungskonstanten S_{xk2} und S_{yk2} berechnet, welche durch den maximalen Rotationswinkel und der maximal gewünschten Verschiebung festgelegt werden.

$$k_{vr}(x) = k_{vr}(y_{rot}) * S_{xk2} \tag{3.16}$$

$$k_{vr}(y) = k_{vr}(x_{rot}) * S_{yk2} \tag{3.17}$$

Die Tiefenposition des Kopfs $k(z)$ steuert die z-Position der Kamera $k_{vr}(z)$ (Gleichung 3.18). Sie wird mithilfe der Skalierungskonstanten S_{zk} transformiert. S_{zk} beschreibt das Verhältnis von der maximalen z-Position der virtuellen Kamera zur maximal festgelegten Kopfgröße.

$$k_{vr}(z) = k(z) * S_{zk} \tag{3.18}$$

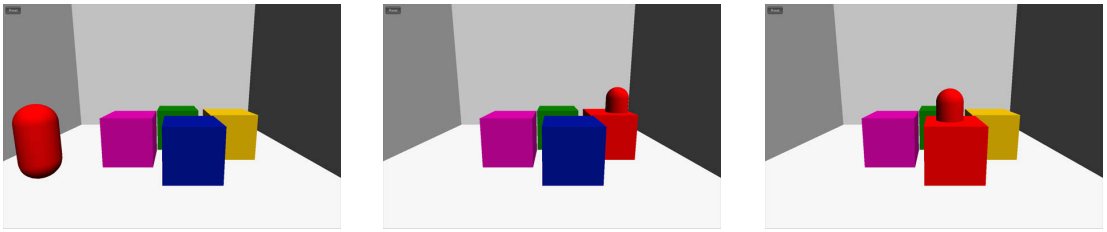
3.7.2 Interaktionsmöglichkeiten mit der VR-Szene

Der Benutzer kann per Hand und Kopf auf folgende Weise mit der VR-Szene interagieren:

- Steuerung der virtuellen Hand und Selektion von Objekten mittels *Geste 1*
- Manipulation von selektierten Objekten mittels *Geste 2*
- Steuerung der virtuellen Kamera mittels Kopfbewegung

3.7.2.1 Steuerung der virtuellen Hand und Selektion von Objekten mittels *Geste 1*

Mit *Geste 1* wird die virtuelle Hand bewegt und ein Objekt selektiert. Die virtuelle Hand (rote Kapsel) folgt der realen 3D-Position der Hand des Benutzers (siehe Abbildung 3.30). Ein Würfelobjekt wird durch die Berührung mit der virtuellen Hand selektiert und rot eingefärbt (vergleiche Abbildung 3.30a mit Abbildung 3.30b). Durch die Berührung der virtuellen Hand eines anderen Würfels oder der Szenenbegrenzung wird die Selektion des Würfels aufgehoben und gegebenenfalls das berührte Würfelobjekt neu selektiert (vergleiche Abbildung 3.30b mit Abbildung 3.30c).



(a) Handposition: Links-Hinten
Selektion: keine

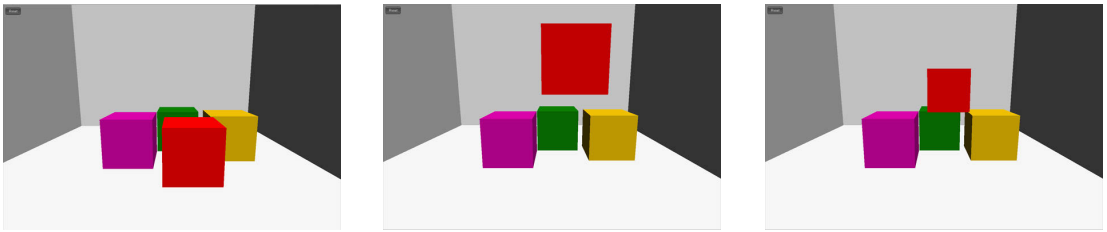
(b) Handposition: Rechts-Vorne
Selektion: rechter Würfel

(c) Handposition: Mitte-Mitte
Selektion: vorderer Würfel

Abbildung 3.30: Verschiedene Handpositionen zur Steuerung der virtuellen Hand (rote Kapsel)

3.7.2.2 Manipulation von selektierten Objekten mittels *Geste 2*

Mit *Geste 2* kann ein selektiertes Objekt in der VR-Szene bewegt werden (siehe Abbildung 3.31). Die 3D-Position des Objekts folgt dabei der 3D-Position der Hand des Benutzers, bis das Objekt abgewählt oder die Geste gewechselt wird.



(a) Handposition: Mitte-Mitte

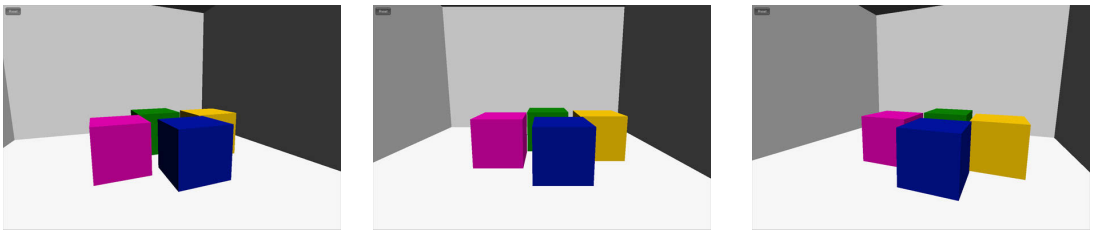
(b) Handposition: Mitte-Oben

(c) Handposition: Mitte-Vorne

Abbildung 3.31: Verschiedene Handpositionen zur Manipulation des selektierten Würfels (rot)

3.7.2.3 Steuerung der virtuellen Kamera mittels Kopfbewegung

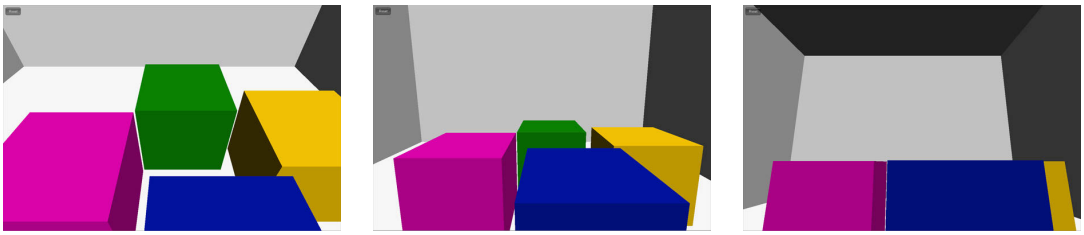
Durch eine virtuelle Kamera wird die Position und Blickrichtung des Betrachters auf die VR-Szene gesteuert und definiert. Durch die Bewegung des Kopfes wird die virtuelle Kamera gesteuert. Die Kopfbewegung nach links und rechts (siehe Abbildung 3.32) bzw. oben und unten (siehe Abbildung 3.33) verändert den Blickwinkel auf die Szene in die entsprechende Richtung. Mit der Bewegung des Kopfes zur RGB-Kamera hin (*Vorne*) bzw. von der Kamera weg (*Hinten*), verändert sich dementsprechend die Entfernung zur VR-Szene (vergleiche Abbildung 3.32b mit Abbildung 3.33b).



(a) Kopfposition: Links-Hinten (b) Kopfposition: Mitte-Hinten (c) Kopfposition: Rechts-Hinten

Abbildung 3.32: Kopfbewegung von links nach rechts mit großem Abstand zur Kamera

Die VR-Szene verändert sich somit relativ zur Kopfposition des Betrachters (HCP), wodurch sich das flache Display in eine 3D-Illusion verwandelt [29]. Der Betrachter bekommt dadurch das Gefühl vermittelt, er kann alle Seiten des Objekts betrachten und auf, unter bzw. um das Objekt herum sehen.



(a) Kopfposition: Oben-Vorne (b) Kopfposition: Mitte-Vorne (c) Kopfposition: Unten-Vorne

Abbildung 3.33: Kopfbewegung von oben nach unten mit kleinem Abstand zur Kamera

Teil II

Implementierung

KAPITEL 4

Hardware und Vorarbeiten zur Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung des in Abschnitt 3.1 vorgestellten Hardware-Prototyps (siehe Abbildung 4.1) und den Vorarbeiten für die Implementierung der Software. In Abschnitt 4.1 wird der Entwurf und die Umsetzung des Hardware-Prototypen beschrieben. Abschnitt 4.2 widmet sich der Kalibrierung der RGB- und Tiefenkamera. Das Training der Haar-Klassifikatoren wird in Abschnitt 4.3 behandelt. Die nötigen Schritte zur Verbindung von Kinect und Tablet per USB werden in Abschnitt 4.4 erläutert und Abschnitt 4.5 beschreibt die Einrichtung der Entwicklungsumgebung und die Einbindung der benötigten Bibliotheken.

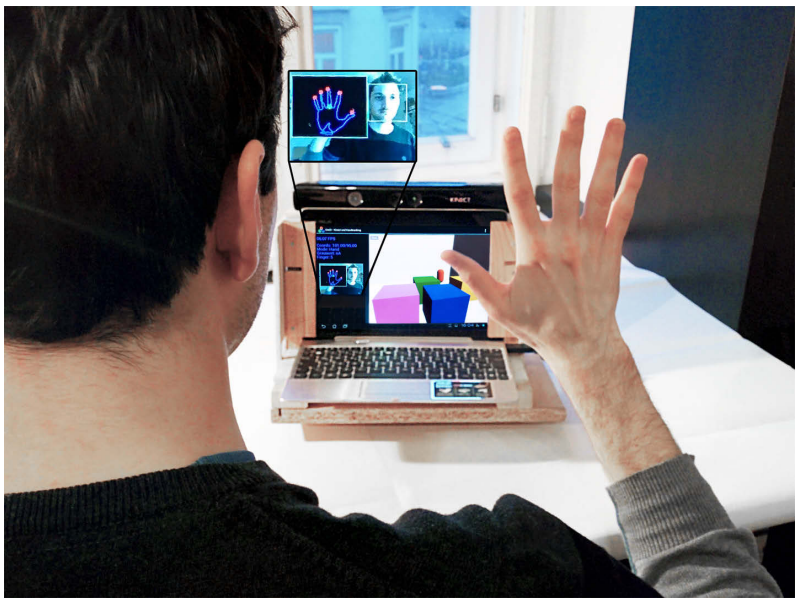
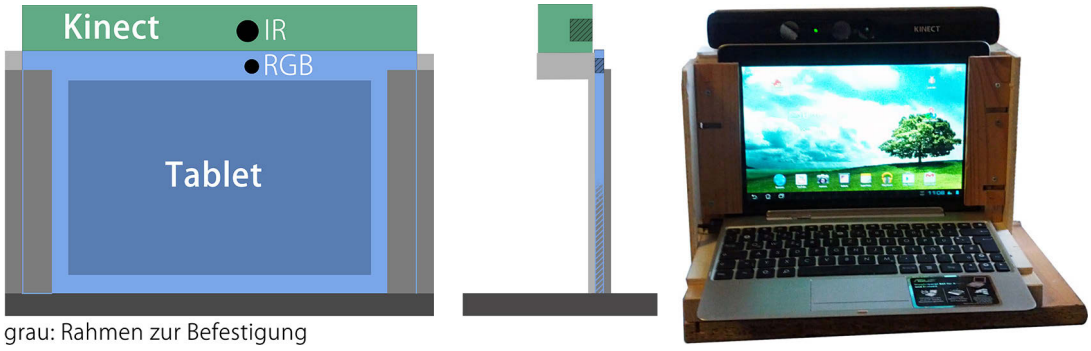


Abbildung 4.1: Überblick Prototypen-Design

4.1 Prototyp

Der Hardware-Prototyp in Abbildung 4.2 umfasst das Tablet, die Tiefenkamera sowie einen Rahmen zur Fixierung beider Geräte. Dies ist für eine extrinsische Kalibrierung notwendig (siehe Abschnitt 4.2). Der Rahmen besteht aus einer Bodenplatte als Fundament und einer Rückwand. An der Rückwand wird ein Sockel und ein Schachtsystem befestigt. Dieses wird für die Fixierung des Tablets verwendet. Die Kinect wird mit dem Sockel verklebt und dadurch fixiert. Abbildung 4.2c zeigt die Umsetzung des Entwurfs mittels Holzrahmen. Die Innenseite des Schachts ist mit Schaumstoff ausgekleidet, um das Tablet nicht zu beschädigen und besser fixieren zu können. Die Kinect ist an zwei Stellen mit Rechtecken aus Acrylglas verklebt, welche wiederum am Sockel befestigt sind.



(a) Hardwareaufbau in der Vorderansicht (b) Seitenansicht (c) Umsetzung des Prototyps

Abbildung 4.2: Prototyp - Entwurf und Umsetzung

4.2 Kalibrierung

Für die Erzeugung von RGBD-Daten werden die intrinsische und extrinsische Matrix, wie in Abschnitt 3.1.4 beschrieben, benötigt. Zur Ermittlung dieser Kameraparamter werden die RGB-Kamera des Tablets und die Infrarot-Kamera der Kinect kalibriert. Die Positionen der Kameras des Prototypen (siehe Abbildung 4.2) sind in Abbildung 4.3 im Detail zu sehen. Die Position der RGB-Kamera (blau) weicht 4 mm in x, 23 mm in y und ca. 14 mm in z von der Position der Infrarot-Kamera (rot) ab. Die Positionsabweichungen der beiden Kameras werden durch die extrinsische Matrix beschrieben. Die Umsetzung und die Ergebnisse der Kalibrierung werden in den nächsten Abschnitten erläutert.

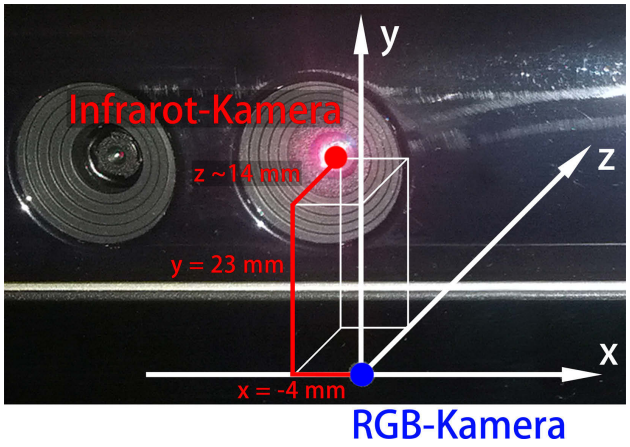


Abbildung 4.3: Kamerapositionen des Prototyps in der Detailansicht

4.2.1 Beschreibung

Die intrinsische Matrix beschreibt die internen Eigenschaften einer Kamera. Diese Werte sind kameraspezifisch und unabhängig von ihrer Position und Lage im Raum. Die extrinsische Matrix wird zur euklidischen Transformation der Bildpunkte in ein anderes Kamerakoordinatensystem benötigt und setzt sich aus einer Rotationsmatrix und einem Translationsvektor zusammen [67, 112]. Auf diese Weise wird die Anordnung der beiden Kameras zueinander beschrieben. Beide Matrizen sind ein essenzieller Faktor für das Mapping der RGB-Bilder mit den Tiefenbildern bzw. das Erzeugen von RGBD-Daten.

Wie in Abschnitt 3.1.4 ausgeführt, lassen sich Kameras vereinfacht durch das Lochkameramodell beschreiben. Für die Ermittlung der Kameraparameter werden mehrere Punkte $(X, Y, Z)^T$ mit bekannter Position im Weltkoordinatensystem und der entsprechenden Position dieser Punkte auf der Bildebene benötigt [113, 137]. Dazu werden Kalibriervorrichtungen mit Messpunkten verwendet, von denen die 3D-Position im Weltkoordinatensystem bekannt ist. Das von Zhang [137] beschriebene Verfahren verwendet dafür eine ebene Kalibriervorrichtung, typischerweise ein Schachbrettmuster (siehe Abbildung 4.4). Für die Kalibrierung wird das Schachbrettmuster in verschiedenen, beliebigen Positionen platziert und Kalibrierungsbilder erstellt. Mit der Annahme, dass sich der Kalibrierkörper in $Z = 0$ befindet, liegen die Messpunkte in der XY-Ebene. Als Messpunkte werden die in Abbildung 4.4 rot markierten, inneren Eckpunkte des Schachbretts verwendet. Aufgrund der bekannten Größe des Musters sind die Positionen der Messpunkte im Weltkoordinatensystem bekannt. Das Schätzverfahren von Zhang [137] verwendet die Positionen der Eckpunkte im Weltkoordinatensystem und in den Kalibrierungsbildern für die Schätzung der intrinsischen und extrinsischen Parameter.

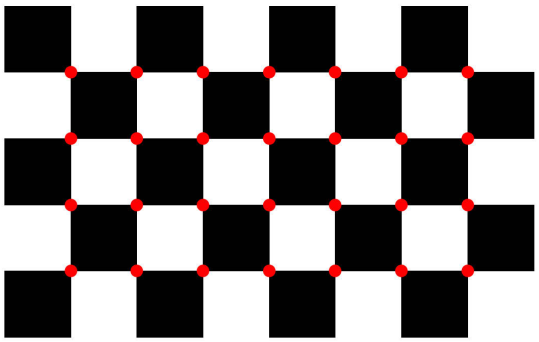


Abbildung 4.4: Schachbrettmuster mit 7×4 Messpunkten (rot)

In dieser Arbeit werden Bilder des Schachbrettmusters mit der RGB-Kamera des Tablets, sowie der Infrarot-Kamera der Kinect aufgenommen und die Software *MIP - MultiCameraCalibration* (MIP-MCC[1]) von Schiller [111] für die Schätzung der Parameter verwendet. Die Ermittlung der Eckpunkte und die Schätzung der Parameter wird mithilfe der Programmbibliothek OpenCV[2] durchgeführt. Der OpenCV-Algorithmus für das Schätzverfahren basiert auf der Arbeit von Zhang [137] und Bouguet [13]. Aufgrund guter Ergebnisse der Kalibrierung und der Deaktivierung des Infrarot-Projektors für die Aufnahme der Kalibrierungsbilder (siehe Abschnitt 4.2.2), wurde die Möglichkeit von MIP-MCC, Tiefendaten in die Kalibrierung mit einfließen zu lassen, nicht verwendet. Außerdem wurde die Kalibrierung für Vergleiche mit der *Camera Calibration Toolbox for Matlab* von Bouguet [13] und der *GML*[3] *C++ Camera Calibration Toolbox* von Vezhnevets et al. [126] durchgeführt.

4.2.2 Umsetzung und Ergebnisse

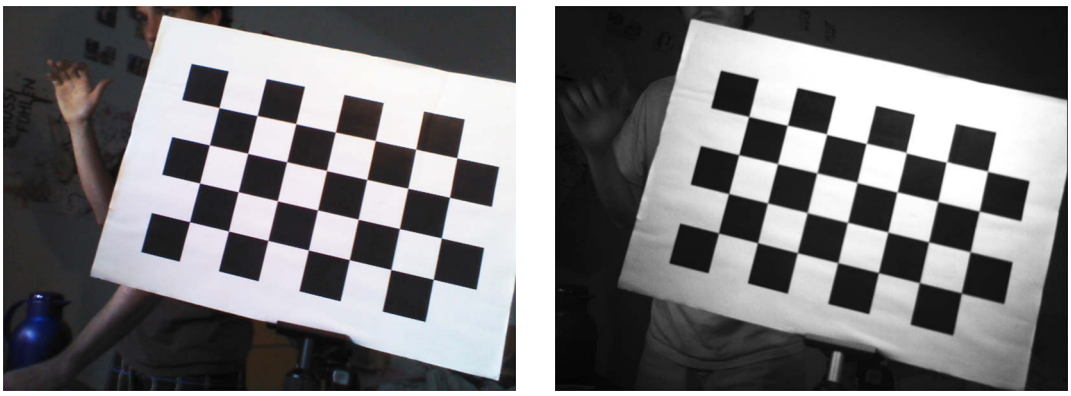
Die Umsetzung der Kalibrierung mit MIP-MCC lässt sich in folgende Schritte unterteilen:

1. Bilder aufzeichnen und speichern
2. Bilderlisten und Projekteinstellungen
3. Detektion der Messpunkte
4. Schätzung der Parameter

[1] Multimedia Information Processing Group-Multi Camera Calibration
[3] Graphics and Media Lab

4.2.2.1 Bilder aufzeichnen und speichern

Für die Kalibrierung wird das in Abbildung 4.4 gezeigte Schachbrettmuster verwendet. Es besteht aus schwarzen und weißen Quadraten mit einer Seitenfläche von je 30mm und 7×4 inneren Eckpunkten (rot).
Das Schachbrettmuster wird mithilfe eines Stativs in verschiedenen Positionen und Winkeln zu den Kameras positioniert. Dabei wird darauf geachtet, dass das Muster in jedem Bild der beiden Kameras vollständig und gut erkennbar ist. Außerdem muss die Orientierung des Musters (schwarzes Quadrat links oben) in allen Bildern dieselbe sein. Die Entfernung des Musters zu den Kameras wird ebenso wie die Position und der Winkel innerhalb des vorgesehenen Interaktionsraums variiert.
Mit den beiden Kameras werden gleichzeitig die Bilder (640x480 Pixel) für die Kalibrierung aufgenommen. Die Bilder der RGB-Kamera werden mit der Kamera-Applikation des Tablets aufgenommen (siehe Abbildung 4.5a). Parallel dazu werden die Bilder der Infrarot-Kamera der Kinect als Video aufgezeichnet und nachträglich, für jede Position des Schachbrettmusters, ein Kalibrierungsbild gespeichert (siehe Abbildung 4.5b). Die Kinect wird dafür mit einem PC verbunden und der Infrarot-Projektor deaktiviert, um das für die Kalibrierung nicht benötigte, sondern störende Infrarotmuster auszublenden. Die Aufzeichnung und das Speichern der Bilder wird mithilfe des Programms *Kinect Studio* aus dem *Kinect for Windows Developer Toolkit* [75] durchgeführt. Für eine präzise Kalibrierung werden zwischen 20 und 80 Bilder empfohlen [111]. In dieser Arbeit wurden 69 aufgezeichnet und verwendet.



(a) Kalibrierungsbild der RGB-Kamera (b) Kalibrierungsbild der Infrarot-Kamera

Abbildung 4.5: Beispiele von Kalibrierungsbildern für die beiden Kameras des Prototypen

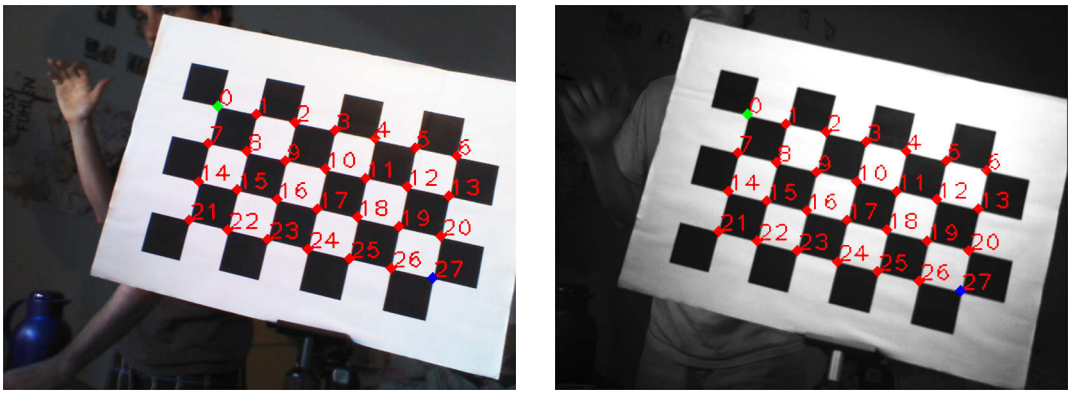
4.2.2.2 Bilderlisten und Projekteinstellungen

Die einzelnen Bilder der Kameras werden in MIP-MCC in einer Bilderliste pro Kamera zusammengefasst und die Kalibrierung mithilfe eines Projekts organisiert. Die Listen bestehen aus den Pfaden zu den Bildern und jede Bilderliste muss für eine erfolgreiche Verarbeitung die gleiche Anzahl an Bildern in derselben Reihenfolge enthalten. Die Bilderlisten der Kameras werden mit MIP-MCC erstellt und anschließend dem Projekt hinzugefügt. Es besteht die Option, ge-

speicherte Eigenschaften der Kamera (intrinsische Matrix, Verzerrungsmodell) zu laden. Die Bilderliste der RGB-Bilder repräsentiert die RGB-Kamera. Diese wird als erste hinzugefügt, damit die extrinsische Matrix die Position und Orientierung der Infrarot-Kamera in Bezug auf die RGB-Kamera abgebildet. Außerdem werden im Projekt die Eigenschaften des Schachbrettmusters (Anzahl und Verteilung der Messpunkte, Seitenlänge der Quadrate) und diverse optionale Parameter eingestellt.

4.2.2.3 Detektion der Messpunkte

Die Erkennung der inneren Eckpunkte im Schachbrettmuster (Messpunkte) wird in MIP-MCC mithilfe der Programmbibliothek OpenCV durchgeführt. Für einen erfolgreichen Detektionsvorgang muss darauf geachtet werden, dass alle Bilder ein vollständiges Schachbrettmuster mit derselben Orientierung (schwarzes Quadrat links oben) enthalten. Die Software sucht in allen Bildern das Schachbrettmuster und ordnet, entsprechend der zuvor definierten geometrischen Eigenschaften des Musters, die inneren Eckpunkte zu. Zusätzlich zu den Standardeinstellungen bietet die Software alternative Möglichkeiten für die Detektion des Schachbrettmusters und eventuellen Verfeinerung der Messpunktzuordnung. Dafür werden Bildverarbeitungsfunktionen für die Erkennung der Eckpunkte im Schachbrettmuster eingesetzt. Die möglichst genaue Zuordnung der Eckpunkte ist essenziell für eine robuste Schätzung der Kameraparameter. In Abbildung 4.6 sind die erkannten Messpunkte und die Zuordnung von 0 bis 27 zu sehen.



(a) Innere Eckpunkte im RGB-Bild (b) Innere Eckpunkte im Infrarot-Bild

Abbildung 4.6: Detektion der inneren Eckpunkte des Schachbrettmusters

4.2.2.4 Schätzung der Parameter und Ergebnisse

Mit den ermittelten inneren Eckpunkten werden, wie in Abschnitt 4.2.1 beschrieben, die intrinsischen und extrinsischen Parameter ermittelt. Abbildung 4.7 veranschaulicht die extrinsische Kalibrierung der Kameras mithilfe der Kalibrierungstoolbox für Matlab [13]. Darin sind die geschätzten Positionen von neun Kalibrierungsbildern (für eine besseren Überblick wurden nicht alle 69 Bilder dargestellt) zu sehen und die daraus ermittelten Positionen der Kameras.

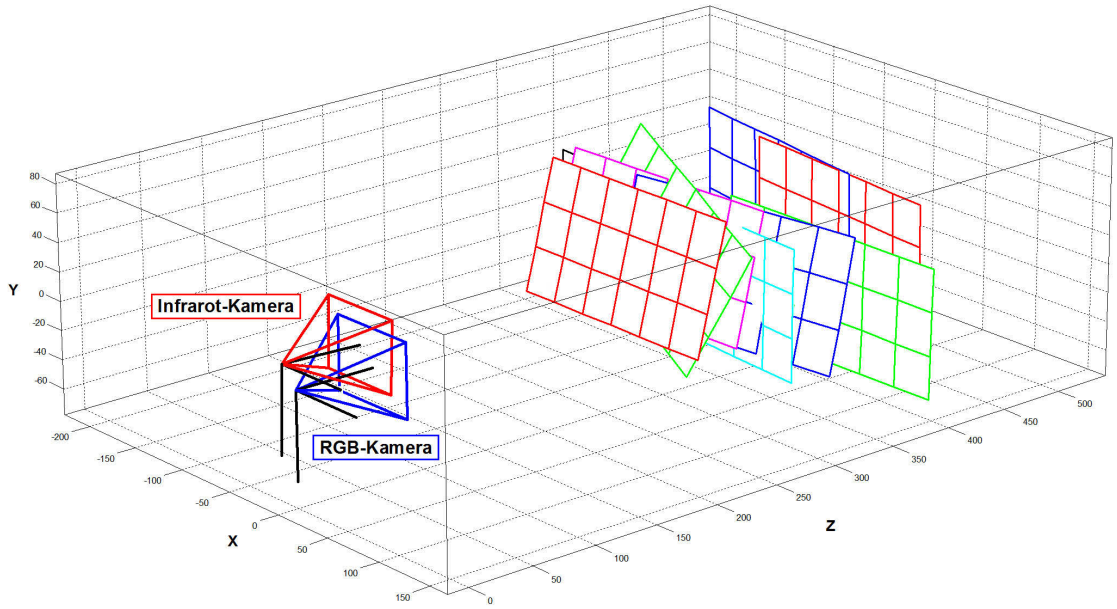


Abbildung 4.7: Positionen der Kalibrierungsbilder und Kameras im Raum

Das in MIP-MCC verwendete Schätzverfahren von OpenCV liefert die in Abbildung 4.8 dargestellten Ergebnisse. Die Werte von *Center* und *Quaternion* in Abbildung 4.8b beschreiben die Position und Lage der Infrarot-Kamera in Bezug auf die RGB-Kamera und entsprechen in etwa den per Hand durchgeführten Messungen (siehe Abbildung 4.3). Die endgültigen intrinsischen und extrinsische Matrizen für das Mapping der Bilder befinden sich in Tabelle 4.1.

Camera: 0 | Camera: 1

Identifier:	rgb-CCD-Perspective-00			
Center	0	0	0	
Quaternion	0	0	0	1
	Normalize Quaternion			
R Matrix	1	0	0	
	0	1	0	
	0	0	1	
Image Size	640	480		
Aspect Ratio	0.998484			
Principal	323.114	245.949		
K Matrix	572.171	0	323.114	
	0	571.303	245.949	
	0	0	1	
Focal Length	572.171			
Undistortion	0	0	0	0

(a) RGB-Kamera

Camera: 0 | Camera: 1

Identifier:	ir-CCD-Perspective-01			
Center	3.89243	-22.8206	-15.6435	
Quaternion	-0.0183853	0.0127649	-0.00380147	0.999742
	Normalize Quaternion			
R Matrix	0.999645	0.0071316	0.025663	
	-0.00807035	0.999295	0.036664	
	-0.0253834	-0.0368581	0.998998	
Image Size	640	480		
Aspect Ratio	0.998362			
Principal	317.744	244.176		
K Matrix	604.288	0	317.744	
	0	603.298	244.176	
	0	0	1	
Focal Length	604.288			
Undistortion	0	0	0	0

(b) Infrarot-Kamera

Abbildung 4.8: Ergebnisse der Kalibrierung (RGB-Kamera im Ursprung)

Der Prototyp verwendet eine Auflösung von 320x240 Pixel für RGB- und Tiefenbild. Für die Kalibrierung werden Bilder mit 640x480 Pixel (maximale Auflösung der Kinect) verwendet. Die höhere Auflösung bietet den Vorteil, dass die Kanten besser erkennbar sind und somit die Messpunkte genauer zugeordnet werden können. Kalibrierungsergebnisse mit kleineren Bildern (320x240 Pixel) zeigten im Vergleich keine signifikanten Abweichungen bei den intrinsischen Parametern und kleine Unterschiede bei den extrinsischen Parametern, zugunsten der größeren Bilder im Praxistest. Ebenso fällt die Überprüfung der Kalibrierung mithilfe von GML [126] und der Toolbox für Matlab [13] positiv aus, da nur minimale Abweichungen bei den intrinsischen (< 0.12) und extrinsischen (< 0.07) Parametern festgestellt wurden (siehe Anhang B).
Für das Mapping der RGB- und Tiefendaten werden die ermittelten Kameraparameter (siehe Abbildung 4.8) in die benötigte Auflösung umgerechnet. Dafür werden die intrinsischen Parameter skaliert (halbiert), wie in [84] beschrieben, die extrinsischen Parameter verändern sich nicht. Daraus ergibt sich die in Tabelle 4.1 dargestellte, endgültig verwendete Kalibrierung.

Tabelle 4.1: Ergebnis der Kalibrierung (gerundet)

Kameraparameter	RGB-Kamera	Infrarot-Kamera
Intrinsische Matrix	$\begin{bmatrix} 286.09 & 0 & 161.56 \\ 0 & 285.65 & 122.98 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 302.14 & 0 & 158.87 \\ 0 & 301.65 & 122.09 \\ 0 & 0 & 1 \end{bmatrix}$
Extrinsische Matrix	$\begin{bmatrix} 0.999 & 0.007 & 0.026 & 3.892 \\ -0.008 & 0.999 & 0.037 & -22.821 \\ -0.025 & -0.037 & 0.999 & -15.644 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	

4.3 Training der Haar-Kaskade Klassifikatoren

Für die Erkennung der Interaktionsobjekte verwendet diese Arbeit Haar-Kaskade Klassifikatoren (siehe Abschnitt 3.2). Ein Haar-Kaskade Klassifikator verwendet Haar-Merkmale für die Beschreibung des gesuchten Objekts und ist als Kaskade mit je einem Nachfolgeknoten aufgebaut (siehe Abschnitt 3.2.2). Der Bildausschnitt wird pro Knoten von einem starken Klassifikator getrennt bewertet, wobei sich die Qualität der Klassifikatoren und der Rechenaufwand von Knoten zu Knoten erhöht. Nach einer positiven Bewertung wird der Ausschnitt an den nächsten Knoten übergeben. Wird am Ende der Kaskade der Ausschnitt positiv bewertet, wurde das gesuchte Objekt gefunden. Die starken Klassifikatoren werden mithilfe eines Boosting Lernalgorithmus erzeugt und entsprechend der gewünschten Detektions- und Falsch-Positiv-Rate pro Knoten aus schwachen Klassifikatoren zusammengestellt.
Die Beschreibung des Trainings mittels Boosting und die Umsetzung mithilfe von OpenCV [90] wird in den nächsten Abschnitten näher ausgeführt.

4.3.1 Beschreibung

Viola und Jones [127] verwenden in Ihrer Arbeit einen auf dem Boostingalgorithmus AdaBoost von Freund und Schapire [31] basierenden Lernalgorithmus zum Trainieren von starken Klassifikatoren. Der Ablauf dieses Lernalgorithmus wird nachfolgend und in Tabelle 4.2 beschrieben.

Für das Training werden Bilder mit (positiv) und ohne (negativ) dem gesuchten Objekt benötigt, welche zu Beginn alle gleich gewichtet werden. In jedem Trainingsdurchlauf wird der schwache Klassifikator h_t mit dem geringsten Fehler ϵ_j dem starken Klassifikator $h(x)$ hinzugefügt. Die Gewichtung wird anschließend aktualisiert, um bisher falsch klassifizierte Bilder hervorzuheben und zu Beginn des nächsten Durchgangs normalisiert. Dadurch soll sichergestellt werden, dass die verschiedenen Eigenschaften des Objekts von den schwachen Klassifikatoren bestmöglich repräsentiert werden. Entsprechend des gewichteten Fehlers α eines schwachen Klassifikators fließt dessen Beurteilung in die Mehrheitsentscheidung des starken Klassifikators ein.

Tabelle 4.2: Lernalgorithmus von Viola und Jones [127]

- Die gegebenen Trainingsbilder $(x_1, y_1)...(x_n, y_n)$
  mit $y_i = 0, 1$ für negative bzw. positive Bilder.
- Initialisierung der Gewichte $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ für $y_i = 0, 1$,
  mit m und l als Anzahl der negativen bzw. positiven Bilder.
- Wiederhole für $t = 1, ..., T$:
  1. Normalisierung der Gewichte $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$
     damit w_t einer Wahrscheinlichkeitsverteilung entspricht.
  2. Für jedes Merkmal j wird ein schwacher Klassifikator h_j mit einem einzigen Merkmal trainiert. Der Fehler wird mit der Berücksichtigung von w_t berechnet, $\epsilon_j = \sum_i w_i |h_j(x_i) - y_i|$.
  3. Auswahl des schwachen Klassifikators h_t mit dem kleinsten Fehler ϵ_t.
  4. Aktualisierung der Gewichte: $w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$
     mit $e_i = 0$ bzw. $e_i = 1$, wenn das Bild x_i richtig bzw. falsch erkannt wird und mit $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- Der starke Klassifikator ist: $h(x) = \begin{cases} 1, & \text{wenn } \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0, & \text{sonst} \end{cases}$
  mit $\alpha_t = log\frac{1}{\beta_t}$

4.3.1.1 Haar-Kaskade Klassifikator

Mithilfe der starken Klassifikatoren wird ein Haar-Kaskade Klassifikator erstellt. Pro Knoten wird ein starker Klassifikator für die Auswertung des Bildausschnitts eingesetzt. Dabei wird die Komplexität der starken Klassifikatoren von Knoten zu Knoten erhöht, um die gewünschte Detektions- und Falsch-Positiv-Rate pro Knoten bzw. für die gesamte Kaskade zu erreichen. Dafür werden von Knoten zu Knoten mehr Merkmale bzw. schwache Klassifikatoren für das Training des starken Klassifikators verwendet.

4.3.1.2 Gentle AdaBoost

In dieser Arbeit wird das Training der Klassifikatoren mithilfe von OpenCV umgesetzt. Darin wird standardmäßig der Boostingalgorithmus Gentle AdaBoost von Friedman et al. [32] verwendet. Dieser basiert auf dem Lernalgorithmus AdaBoost von Freund und Schapire [31], verwendet allerdings keine schwachen Klassifikatoren mit binären, sondern kontinuierlichen Entscheidungswerten im Intervall $[-1, 1]$. Damit wird die Sicherheit der Entscheidung ausgedrückt und der Faktor α (siehe Tabelle 4.2) entfällt. Außerdem wird für die Auswahl des besten schwachen Klassifikators der kleinste gewichtete quadratische Fehler ϵ_t (Gleichung 4.1) verwendet [32, 136]:

$$\epsilon_t = \sum_i w_i (y_i - h_t(x_i))^2 \tag{4.1}$$

Mit $h_t(x)$ aus Gleichung 4.2, bestehend aus der Differenz der gewichteten Wahrscheinlichkeiten P_w für eine positive und negative Bewertung des Bildes [32]:

$$h_t(x) = P_w(y = 1|x) - P_w(y = -1|x) \tag{4.2}$$

Die Gewichte werden in Gentle AdaBoost mit $exp(-y_i h_t(x_i))$ aktualisiert, zum Unterschied des in Tabelle 4.2 beschriebenen Algorithmus, wodurch sich dieser Ansatz robuster gegenüber Ausreißern verhält. Es wird weniger Augenmerk auf falsch klassifizierte Bilder gelegt und dadurch eine bessere Generalisierung des Klassifikators erreicht.

4.3.2 Umsetzung und Ergebnisse

Die Haar-Kaskade Klassifikatoren K1, K2 und K3 (siehe Abschnitt 3.4) werden mittels OpenCV [90] trainiert bzw. mithilfe den von Nayak [79] zur Verfügung gestellten Tools und Anleitung erstellt. Die Klassifikatoren werden mit dem Boostingalgorithmus Gentle AdaBoost von Friedman et al. [32] trainiert. Für diese Arbeit wurden mehrere Klassifikatoren mit verschiedenen Einstellungen trainiert und schlussendlich je ein Haar-Kaskade Klassifikator für die Erkennung von *Geste 1* (K1), *Geste 2* (K2) und der Hand (K3) ausgewählt (siehe Abbildung 4.9).
Das Training lässt sich in die folgenden Schritte unterteilen und wird im Anschluss beschrieben:

1. Trainingsbilder aufnehmen
2. Trainingsdaten erstellen
3. Training der Haar-Kaskade Klassifikatoren

4.3.2.1 Trainingsbilder aufnehmen

Das Training benötigt mehrere Bilder mit und ohne dem gesuchten Objekt, sogenannte positive bzw. negative Bilder. Viola und Jones [127] verwendeten an die 10.000 positive und negative Bilder für das Training. Aufgrund der im Rahmen dieser Arbeit begrenzten Möglichkeiten wurde eine weitaus geringere Zahl von Bildern für das Training genutzt (siehe Tabelle 4.3). Für die weiteren Schritte wurden Bilder mit einer Auflösung von 267x200 Pixel verwendet. Größere Bilder hätten einen höheren Rechenaufwand verursacht und werden nicht benötigt da für die positiven Trainingsdaten eine Objektgröße kleiner 38x28 Pixel verwendet wird. In Tabelle 4.3 werden die Eigenschaften der gesuchten Objekte beschrieben und die Anzahl der verwendeten Trainingsbilder angeführt.

Tabelle 4.3: Gesuchte Objekte und Trainingsbilder

Beschreibung	Positive Bilder	Negative Bilder
K1: 5-Finger ausgestreckt mit Handfläche	158	449
K2: Zeigefinger ausgestreckt mit Handfläche	536	449
K3: Handunterseite mit ausgestrecktem Daumen	304	676

Für das Erstellen der positiven Bilder wurde die gesuchte Pose mit der rechten Hand ausgeführt und in verschiedenen Positionen, Beleuchtungen und Umgebungen von einer Kamera aufgenommen. Beim Erstellen der negativen Bilder muss sichergestellt sein, dass das gesuchte Objekt nicht in den Bildern vorkommt. Außerdem wurden negative Bilder mit dem Benutzer im Bild (keine sichtbaren Hände) erstellt, um den Klassifikator besser für das Anwendungsszenario zu trainieren. Abbildung 4.9 zeigt exemplarisch drei Ausschnitte positiver Bilder mit den gesuchten Objekten der Klassifikatoren.

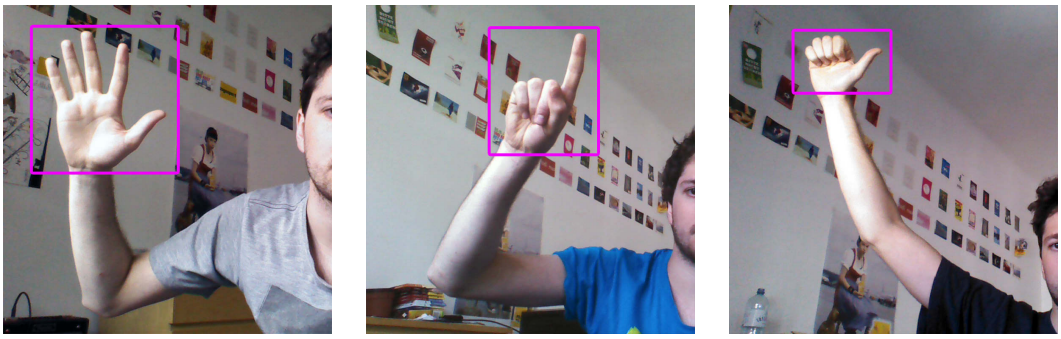
(a) K1 (b) K2 (c) K3

Abbildung 4.9: Beispiele für positive Bilder der drei Klassifikatoren (Ausschnitte)

4.3.2.2 Trainingsdaten erstellen

Die Trainingsbilder werden in diesem Schritt in die von OpenCV benötigte Form und in verarbeitbare Trainingsdaten umgewandelt. In den positiven Bildern wird zuerst der Ausschnitt mit

dem gesuchten Objekt definiert. Dafür öffnet das Programm *objectmarker.exe* [79] alle positiven Bilder nacheinander und der Benutzer bestimmt mittels Maus den rechteckigen Ausschnitt mit dem gesuchten Objekt (siehe Abbildung 4.10). Dabei sollte darauf geachtet werden, dass das Auswahlrechteck dem Seitenverhältnis des Suchfensters der Klassifikatoren entspricht (siehe unten), um später starke Verzerrungen zu vermeiden.



(a) K1 (b) K2 (c) K3

Abbildung 4.10: Auswahl der gesuchten Objekte (pink)

Die Positionen der Ausschnitte werden in einer Textdatei mit dem Dateinamen des dazugehörigen Bildes vermerkt. Anhand dieser Informationen wird mit *createsamples.exe* [79] das Objekt aus den positiven Bildern ausgeschnitten, entsprechend der gewählten Suchfenstergröße[4] skaliert und in einer *vec-Datei*[5] gespeichert (siehe Abbildung 4.11). Diese vec-Datei beinhaltet alle Bildausschnitte der gesuchten Objekte und entspricht den positiven Trainingsdaten. Aufgrund der unterschiedlichen Formen der gesuchten Objekte wurde die Suchfenstergröße dem Objekt entsprechend angepasst. Das Suchfenster wurde für K1 mit 24x24 Pixel, für K2 mit 24x28 Pixel und für K3 mit 38x24 Pixel definiert, wodurch auch die minimal erkennbare Objektgröße festgelegt wird.

(a) K1 (b) K2 (c) K3

Abbildung 4.11: Ausgeschnittene Objekte (positive Trainingsdaten)

Die negativen Bilder benötigen keine zusätzliche Bearbeitung und werden vom Trainingsalgorithmus in mehrere negative Bilder, entsprechend der Suchfenstergröße, geteilt. Für das Er-

[4]Das Suchfenster definiert den aktuell untersuchten Bildausschnitt der Detektion und entspricht zu Beginn der Größe der positiven Trainingsdaten bzw. minimal erkennbaren Objektgröße (siehe Abschnitt 3.2.2).

[5]Datei mit .vec als Dateiendung in der Bilder im binären Format gespeichert werden [90].

stellen der negativen Trainingsdaten werden die Dateinamen der negativen Bilder mithilfe von *create_list.bat* [79] in einer Textdatei als Liste gespeichert.

4.3.2.3 Training der Haar-Kaskade Klassifikatoren

Mit den erstellten Trainingsdaten kann das Training der Haar-Kaskade Klassifikatoren gestartet werden. Dazu wird *haartraining.exe* [79] mit den gewünschten Parametern ausgeführt. Die Parameter werden dafür verwendet Informationen zu den Trainingsdaten und die Eigenschaften des gewünschten Haar-Kaskade Klassifikator an den Trainingsalgorithmus zu übermitteln. Tabelle 4.4 beschreibt die wichtigsten Parameter für das Training:

Tabelle 4.4: Beschreibung der Trainingsparameter

Parameter	Beschreibung	Beispiel (K1)
-data <Ordnername>	Haar-Kaskade Klassifikator Speicherort	data/cascade
-vec <Dateiname>	Positive Trainingsdaten	data/vector.vec
-bg <Dateiname>	Negative Trainingsdaten	negative/infofile.txt
-npos <Anzahl>	Anzahl positiver Trainingsdaten	158
-nneg <Anzahl>	Anzahl negativer Trainingsdaten	449
-nstages <Anzahl>	Maximale Anzahl von Knoten der Kaskade	15
-mode <BASIC | ALL>	Haarmerkmaltypen BASIC (Aufrechte) oder ALL (Aufrechte und Gedrehte)	ALL
-w <Breite>	Breite der Trainingsdaten (Suchfenster)	24
-h <Höhe>	Höhe der Trainingsdaten (Suchfenster)	24
-nonsym	Das gesuchte Objekt ist nicht symmetrisch	
-nsplits <Anzahl>	Schwacher Klassifikator aufgebaut als Entscheidungsbaum mit <Anzahl> Knoten	2
-minhitrate <Zahl>	Minimale Detektionsrate pro Knoten	0.999
-maxfalsealarm <Zahl>	Maximale Falsch-Positiv-Rate pro Knoten	0.95

Die Parameter aus Tabelle 4.4 werden für das Training des gewünschten Haar-Kaskade Klassifikators entsprechend angepasst. Dafür werden die verwendeten Trainingsdaten (Dateiname und Anzahl von positiven und negativen Trainingsdaten) und die Eigenschaften des gesuchten Objekts (Breite und Höhe der positiven Trainingsdaten/Suchfenster) eingestellt. Außerdem kann zum Beispiel die maximale Anzahl an Knoten und die Detektions- und Falsch-Positiv-Rate verändert werden, je nachdem welche Probleme bei der Detektion auftreten bzw. Ziele verfolgt werden. Der trainierte Haar-Kaskade Klassifikator wird zum Schluss mit *convert.bat* [79] in eine *XML-Datei* umgewandelt. Diese XML-Datei repräsentiert den Haar-Kaskade Klassifikator und wird von OpenCV für die Detektion verwendet. Dadurch ist es möglich, die Detektion zu starten und die verschiedenen Eigenschaften der Klassifikatoren zu testen.

Für diese Arbeit wurden eine Vielzahl von verschiedenen Parameter und Trainingsdaten ausprobiert bzw. erstellt. Die folgenden Einstellungen konnten im Praxistest mit der subjektiv besten

Detektionsrate bzw. kleinsten Falsch-Positiv-Rate überzeugen und die damit erzeugten Klassifikatoren werden im Prototyp verwendet. Tabelle 4.5 zeigt die von Tabelle 4.4 abweichenden und für die einzelnen Klassifikatoren spezifisch angepassten Parameter:

Tabelle 4.5: Spezifische Parameter der Haar-Kaskade Klassifikatoren

Beschreibung	K1	K2	K3
Positive Trainingsdaten	158	536	304
Negative Trainingsdaten	449	449	676
Suchfenstergröße	24x24	24x28	38x24
Maximale Knotenanzahl	15	15	15
Detektionsrate	0.999	0.9999	0.999
Falsch-Positiv-Rate	0.5	0.3	0.5
Tatsächliche Knotenanzahl	13	14	14

4.4 Kinect per USB mit Tablet verbinden

Die Verbindung der Kinect mit dem Tablet ist in Abschnitt 3.1.3 beschrieben und schematisch dargestellt. Um die Daten der Tiefenkamera zu erhalten, muss das Tablet eine Verbindung zur Kinect aufbauen. Da die Kinect für Windows keine offizielle Unterstützung und auch keine Treiber für Android liefert, muss der Treiber neu kompiliert werden [2,68]. Hierfür kann auf Projekte mit dem Vorgänger der Kinect für Windows, Kinect für Xbox 360, aufgebaut [68,80] und das *Android Native Development Kit* (Android NDK) [39] verwendet werden. Android NDK wird eingesetzt, um Bibliotheken in den Programmiersprachen C und C++ in einer Android-Anwendung verwenden zu können. Das Einlesen der Tiefendaten der Kinect durch die Android-Applikation wird durch das *Software Development Kit*[6] (SDK) des Open-Source-Projekts *OpenNI*[7] ermöglicht (siehe Abschnitt 5.4.2). Für die Adaption des Tablets werden Root- bzw. Administratorrechte benötigt.
Die Umsetzung kann in die folgenden Schritte unterteilt werden:

1. OpenNI mithilfe von Android NDK kompilieren
2. Kinect-Treiber mithilfe von Android NDK kompilieren
3. Treiber und OpenNI-Dateien in die jeweiligen Tablet-Ordner hochladen
4. Kinect in das System einhängen[8]

Die einzelnen Schritte werden mithilfe der Anleitungen von Lo [68] und Niisato [80] durchgeführt und im Anhang (siehe A) in vollem Umfang beschrieben. Nachfolgend werden die wichtigsten Details kurz zusammengefasst.

[6] Ein Set von unterschiedlichen Tools etc. welche für die Softwareentwicklung benötigt werden.
[7] Open Natural Interaction [97].
[8] mount.

Im ersten und zweiten Schritt werden OpenNI und der Kinect-Treiber am PC neu kompiliert, um eine Verwendung mit Android zu ermöglichen. Dafür wird der OpenNI unterstützende Kinect-Treiber von *Avins2's SensorKinect* [2] und die vom Treiber benötigte OpenNI-Version [97] verwendet. Der Treiber und OpenNI werden mit dem Tool *ndk-build* von Android NDK [39] neu kompiliert.
Im nächsten Schritt wird das Tool *Android Debug Bridge* (ADB), Teil des *Android SDK's* [40], für die Übertragung der Dateien an das Tablet verwendet. Über die Kommandozeile wird mit dem Befehl *adb shell* eine Verbindung zum mit dem PC verbundenen Tablet aufgebaut und mittels *adb push* die Dateien übertragen.
Wurden die bisherigen Schritte erfolgreich abgeschlossen, kann die Kinect mittels ADB und dem Befehl *mount* in das System eingehängt und über OpenNI bzw. der Android-Applikation auf die Tiefendaten der Kinect zugegriffen werden (siehe Abschnitt 5.4.2). Dieser Schritt ist nach jedem Neustart des Geräts notwendig.

4.5 Software - Entwicklungsumgebung

Für die Entwicklung einer Android-Applikation wird in der offiziellen Android-Dokumentation *Eclipse* [27] empfohlen. Eclipse ist eine plattformunabhängige Entwicklungsumgebung und kann mit diversen Erweiterungen für die Entwicklung von Android-Applikationen verwendet werden (siehe Abschnitt 4.5.1). Das Erstellen des Projekts und das Einbinden der verwendeten Bibliotheken wird in Abschnitt 4.5.2 behandelt. Alle Teile des Projekts werden in Eclipse integriert und verwaltet.

4.5.1 Einrichten von Eclipse

Für die Implementierung dieser Arbeit wurde *Eclipse IDE for Java Developers* (Version Indigo SR2) auf einem Windows 7 64-bit Rechner installiert und verwendet. Durch die Erweiterung von Eclipse mit den *Android Developer Tools* (ADT) [34], können Android-Projekte auf Basis von Java kompiliert und daraus Android-Applikationen erzeugt werden. Dafür werden zusätzlich zur installierten Eclipse-Umgebung [27] das *Java-Development-Kit* (JDK) [98] und Android SDK [40] vorausgesetzt. Eclipse nutzt ADB, um per USB eine Verbindung zum Tablet aufzubauen, den Systemlog des Tablets auszulesen und die kompilierte Applikation an das Tablet zu übertragen. Die Installationsschritte sind in [44] beschrieben.
Die Eclipse-Erweiterung *C/C++ Development Tooling* (CDT) [26] ermöglicht das Verwenden von Android NDK [39] und das Kompilieren von C/C++-Code bzw. das Verwenden von C/C++-Bibliotheken. Für das Kompilieren von C/C++-Code auf einer Windows Plattform wird die *GNU Compiler Collection* (GCC) benötigt. Mithilfe von Android NDK ist es möglich, rechenintensive Aufgaben der Android-Anwendung in nativen C/C++-Code auszulagern, um die Performance zu verbessern. Der C/C++-Code wird mittels Android NDK kompiliert und zu einer gemeinsam genutzten Bibliothek[9] zusammengefasst. Als Schnittstelle zwischen Java und C/C++ fungiert

[9] shared library.

JNI[10] [42]. Das Einrichten von Eclipse für die C/C++ Softwareentwicklung mit Android NDK wird in [119] erläutert.

4.5.2 Einrichten des Android-Projekts

Für diese Arbeit wird in Eclipse ein Android-Projekt auf Basis der OpenCV-Beispielprojekte [90] erstellt. Diese Beispielprojekte dienten als Ausgangspunkte für diese Arbeit und verwenden sowohl Java- als auch C/C++-Code. In der OpenCV-Dokumentation wird die Installation von OpenCV [91] und das Erstellen eines Android-Projekts in Eclipse [83] im Detail beschrieben. Außerdem werden für die Umsetzung dieser Arbeit die Bibliotheken von OpenCV [90], OpenNI [97], *Boost*[11] und die Spiel-Engine *Unity3D* [122] inklusive VR-Szene benötigt (siehe Abschnitt 5.1). Die Projekteinbindung wird nachfolgend beschrieben.

Die Einbindung der OpenCV-Bibliothek wird in der OpenCV-Dokumentation [83, 89] detailliert beschrieben und hier kurz zusammengefasst. Zur Einbindung wird der Ordner *OpenCV/sdk* in Eclipse importiert[12] und die Bibliothek dem Projekt hinzugefügt[13]. Für die Einbindung der nativen OpenCV-Bibliothek ist es notwendig, den Pfad *sdk/native/jni/include* dem Projekt hinzuzufügen[14]. Außerdem muss die Datei *Android.mk*[15] des Projekts entsprechend erweitert bzw. die Datei *OpenCV.mk* mittels *include* eingebunden werden (siehe Listing 4.1).

```
1  LOCAL_PATH := $(call my-dir)
2
3  include $(CLEAR_VARS)
4  include ../../OpenCV246/sdk/native/jni/OpenCV.mk
5
6  LOCAL_MODULE := detection_based_tracker
7  LOCAL_SRC_FILES := DetectionBasedTracker_jni.cpp KinectNetwork_jni.cpp
       Process_jni.cpp KinectUSB.cpp Kinect_jni.cpp
8  LOCAL_C_INCLUDES += $(LOCAL_PATH)
9  LOCAL_LDLIBS += -llog -ldl
10 LOCAL_STATIC_LIBRARIES := boost_thread boost_system boost_chrono
       boost_date_time boost_regex boost_signals boost_iostreams
11 LOCAL_SHARED_LIBRARIES += openni_usb openni_OpenNI
12 include $(BUILD_SHARED_LIBRARY)
13
14 $(call import-module,boost)
15 $(call import-module,openni)
```

Listing 4.1: Android.mk

Für die Einbindung der weiteren C++-Bibliotheken OpenNI und Boost werden zuerst die Dateien von OpenNI (Abschnitt 4.4) und Boost [106] mittels Android NDK für Android kompiliert. Anschließend werden die Pfade zu OpenNI und Boost, wie zuvor bei OpenCV, dem

[10] Java Native Interface.
[11] Boost ist eine freie C++-Bibliothek, unter anderem mit Netzwerk-Unterstützung [11].
[12] File -> Import -> Existing project.
[13] Project -> Properties -> Android -> Library -> Add -> OpenCV Bibliothek auswählen.
[14] Project -> Properties -> C/C++ General -> Paths and Symbols -> Includes -> Add.
[15] Android.mk enthält die von Android NDK benötigten Beschreibungen der C++-Bibliotheken.

Projekt hinzugefügt. In der Android.mk-Datei werden die benötigten Bibliotheken verlinkt (LOCAL_STATIC_LIBRARIES bzw. LOCAL_SHARED_LIBRARIES) und mittels *import-module* importiert [106] (siehe Listing 4.1). Dafür ist es notwendig, im OpenNI und Boost-Ordner je eine Android.mk-Datei zu erstellen und darin für jede Bibliothek ein Modul zu definieren (siehe Listing 4.2).

```
1 include $(CLEAR_VARS)
2 LOCAL_MODULE := openni_usb
3 LOCAL_SRC_FILES := Platform/Android/libs/armeabi-v7a/libusb.so
4 LOCAL_EXPORT_C_INCLUDES :=$(LOCAL_PATH)/Include
5 include $(PREBUILT_SHARED_LIBRARY)
```

Listing 4.2: Definition des Moduls *openni_usb* in OpenNI's Android.mk

4.5.3 Einrichten des Unity-Projekts und Einbindung in Android

Die Spiel-Engine Unity3D und das Unity-Projekt mit der VR-Szene werden mithilfe der Anleitung von Unity [1] in Eclipse eingebunden. Die dafür nötigen Schritte werden hier kurz zusammengefasst.
Es muss darauf geachtet werden, dass die Auswahl der minimalen Android SDK Version und die Paketnamen des Android-Projekts mit dem *Bundle Identifier* des Unity-Projekts übereinstimmen. Die mit Unity kompilierten Dateien[16] werden in den Ordner *UnityLib* kopiert. Der Ordner *assets* in UnityLib wird in den Android-Projektordner verschoben. Diese Dateien müssen nach jeder neuen Kompilierung mittels Unity neu kopiert bzw. verschoben werden. Für die Einbindung der Bibliothek wird zuerst mit den Dateien von UnityLib ein neues Android-Projekt in Eclipse angelegt[17], dieses als Bibliothek definiert[18] und abschließend dem Android-Projekt dieser Arbeit hinzugefügt[19]. Außerdem wird noch die Datei *classes.jar* von Unity eingebunden[20].

[16] Öffne den Unity-Projekt Ordner -> Temp -> StagingArea.
[17] File -> New -> Project... -> Android -> Android Project from Existing Code -> UnityLib auswählen.
[18] Project -> Properties -> Android -> Is Library auswählen.
[19] Project -> Properties -> Android -> Library -> Add -> Unity Bibliothek auswählen.
[20] Project -> Properties -> Java Build Path -> Libraries -> Add External JARs... -> Unity/Editor/Data/PlaybackEngines/androidplayer/bin/classes.jar auswählen.

KAPITEL 5

Software

Dieses Kapitel beschreibt die Implementierung und den Aufbau der Software im Detail. Abbildung 5.1 bietet einen Überblick und stellt die Zusammenhänge der einzelnen Module der Software dar (siehe Abschnitt 5.1).

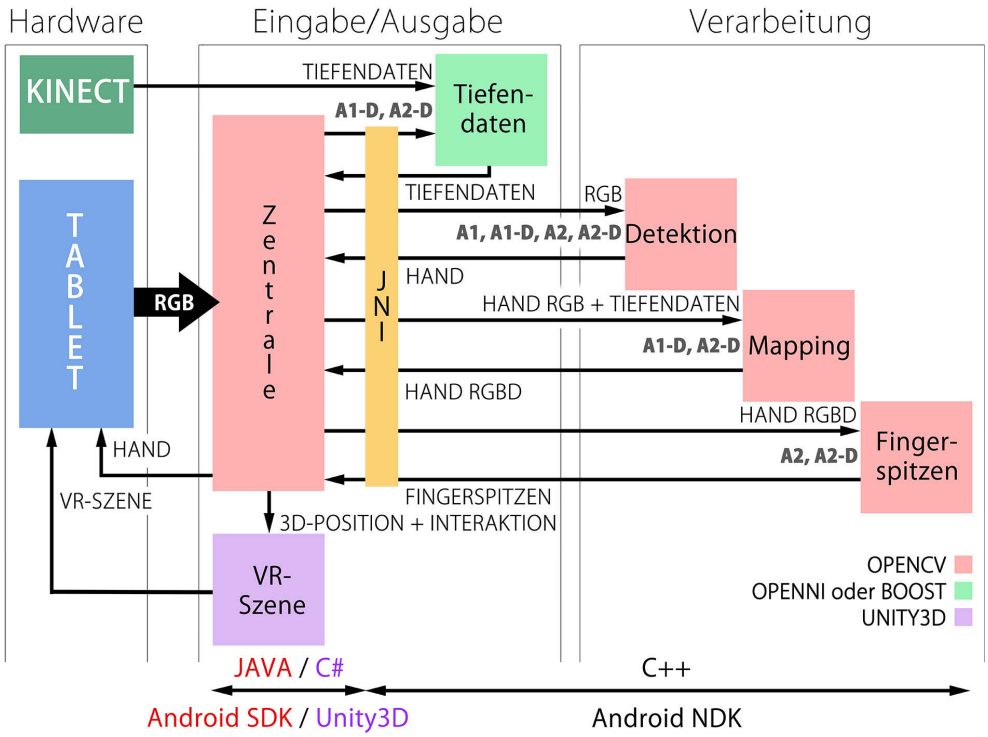


Abbildung 5.1: Überblick Softwareaufbau

Die verwendeten Bibliotheken und Technologien werden in Abschnitt 5.1 erläutert und die Hauptaufgaben der Software vorgestellt. Einen Einblick in die Grundlagen von Android und Unity3D gibt Abschnitt 5.2. Die zentrale Organisation und Kommunikation der Applikation wird in Abschnitt 5.3 erläutert. In Abschnitt 5.4 wird das Einlesen der RBG- und Tiefendaten beschrieben. Außerdem wird die Ermittlung der Fingergesten in Abschnitt 5.5 bzw. der Kopfposition in Abschnitt 5.6 im Detail betrachtet. Abschnitt 5.7 widmet sich zum Abschluss des Kapitels der Darstellung und Interaktion mit der VR-Szene.

5.1 Software Design

Der Softwareaufbau ist schematisch in Abbildung 5.1 dargestellt und zeigt alle Module des Projekts. Für die Fingergestenerkennung mittels A2-D werden alle abgebildeten Module benötigt. Beim Verzicht auf Tiefendaten (A1 und A2) werden die beiden Module *Tiefendaten* und *Mapping* nicht eingesetzt. Außerdem wird für A1 das Modul *Fingerspitzen* nicht verwendet, da die Fingergeste mithilfe der *Detektion* ermittelt wird. Die eingesetzten Technologien und Bibliotheken werden in Abbildung 5.1 in verschiedenen Farben hervorgehoben bzw. beschrieben und nachfolgend kurz erläutert:

- **Android SDK:** Für die Entwicklung der Android-Applikation wird das Android SDK [40] verwendet und mittels der Programmiersprache Java implementiert. Das SDK enthält alle nötigen Bibliotheken und Tools für das Erstellen und Testen einer Android-Anwendung.
- **Android NDK:** Mithilfe von Android NDK [39] können rechenintensive Module in C++-Code ausgelagert und C++-Bibliotheken verwendet werden. Es wird für die Implementierung nativer Methoden in einer Android-Anwendung verwendet und ist nur in Verbindung mit dem Android SDK einsetzbar.
- **JNI:** Als Programmierschnittstelle zwischen Java- und C++-Code wird JNI [42] verwendet. Es wird für den Einsatz von Android NDK und C++-Bibliotheken benötigt und ermöglicht Java-Code den Aufruf von nativen Methoden aus der zuvor erstellten und geladenen, dynamischen Bibliothek.
- **OpenCV:** Die freie Programmbibliothek OpenCV [90] enthält unter anderem Algorithmen für die Objekterkennung mittels Haar-Klassifikator und die Bildverarbeitung. Sie steht für Android zur Verfügung und wird in dieser Arbeit sowohl in die Java- wie auch die C++-Software-Module integriert.
- **OpenNI:** Die quelloffene Programmierschnittstelle OpenNI [97] ermöglicht den Zugriff auf verschiedene 3D-Kameras und unterstützt unter anderem die Kinect. Mithilfe des dazugehörigen SDK wird die Kinect von der Android-Anwendung angesteuert und das Einlesen der Tiefendaten per USB realisiert.
- **Boost:** Die frei verfügbare Bibliothek Boost [11] ist eine umfangreiche Sammlung von verschiedenen C++-Bibliotheken. Sie stellt plattformunabhängige Bibliotheken zur Verfügung und wird in dieser Arbeit für die Netzwerkverbindung und Übertragung der Tiefendaten vom PC zum Tablet eingesetzt.

- **Unity3D:** Die Spiel-Engine Unity3D [122] ermöglicht die Entwicklung von interaktiven 3D-Inhalten für verschiedene Plattformen, unter anderem für mobile Geräte und Android. Die VR-Szene wird mithilfe von Unity3D erstellt und auf dem Tablet dargestellt. Für die Implementierung der Interaktionslogik werden C#-Skripten eingesetzt.

Die Software lässt sich in die folgenden fünf Hauptaufgaben unterteilen (bestehend aus den Modulen in Abbildung 5.1):

- **Zentrale Organisation und Kommunikation:** Das Modul *Zentrale* ist der Ausgangspunkt für alle weiteren Schritte, beinhaltet die grafische Benutzeroberfläche (GUI[1]) sowie Anwendungseinstellungen und führt die gesammelten Informationen zusammen (siehe Abschnitt 5.3).
- **Bilddaten einlesen und Mapping:** Die RGB-Daten des *Tablets* werden vom Modul *Zentrale* verarbeitet und die *Kinect* mithilfe des Moduls *Tiefendaten* ausgelesen. Aus den RGB- und Tiefendaten werden im Modul *Mapping* RGBD-Daten erzeugt (siehe Abschnitt 5.4).
- **Ermittlung der Fingergesten:** Je nach Ansatz wird im Modul *Detektion* direkt die Fingergeste ermittelt (A1) oder die Hand erkannt (A2) und die Fingergeste mithilfe des Moduls *Fingerspitzen* ermittelt. Die zur Verfügung stehenden Daten werden in der *Zentrale* verwaltet und für die Bestimmung der 3D-Position verwendet (siehe Abschnitt 5.5).
- **Ermittlung der Kopfposition:** Mithilfe der *Detektion* wird der Kopf erkannt und dessen 3D-Position festgestellt und in der *Zentrale* verwaltet (siehe Abschnitt 5.6).
- **Darstellung und Interaktion - VR-Szene:** Mit dem Modul *VR-Szene* wird die Interaktionssteuerung und die Darstellung der Szene auf dem *Tablet* realisiert. Die Implementierung verwendet die Daten der *Zentrale* für die Interaktionssteuerung (siehe Abschnitt 5.7).

5.2 Grundlagen der verwendeten Technologien

Im folgenden Abschnitt werden die grundlegenden Konzepte von Android und Unity3D erläutert und die in dieser Arbeit verwendeten Komponenten vorgestellt.

5.2.1 Android

Die zentrale Komponente des Hardware-Prototyps ist das Tablet. Es wird zur Darstellung der VR-Szene und als zentrale und einzige Recheneinheit eingesetzt. Sämtliche Algorithmen für Detektion, Mapping etc. sind sind auf dem Tablet implementiert, das ebenfalls alle Daten verarbeitet. Das mobile Betriebssystem des Tablets ist Android [35] und basiert auf einem für mobile Geräte optimierten Linux-Kernel. Eine Android-Applikation wird mithilfe des Android SDK [40] und der Programmiersprache Java entwickelt und in einer *Dalvik Virtual Machine*

[1] Graphical User Interface.

(DVM) ausgeführt. Performancekritische Teile können mithilfe des Android NDK [39] in C/C++ implementiert werden. Außerdem ermöglicht das NDK die Einbindung und Verwendung von C/C++-Bibliotheken. Nachfolgend wird auf die für diese Arbeit relevanten und Android spezifischen Komponenten und Konzepte eingegangen. Eine ausführliche Dokumentation ist in [35] zu finden und eine zusammenfassende Beschreibung der Konzepte und Komponenten von Android wird in [116] vorgestellt.
Eine Android-Applikation besteht aus verschiedenen Android-Komponenten. Mithilfe von *Intents* [38] können die einzelnen Komponenten untereinander kommunizieren und aufgerufen werden. Intents verbinden die einzelnen Komponenten zu einem Gesamtsystem und ermöglichen den Aufruf und die Benutzung von Komponenten anderer Android-Anwendungen. Dafür muss die Applikation die entsprechenden Zugriffsrechte besitzen, welche vom Betriebssystem überprüft werden. Diese Berechtigungen werden im *Android-Manifest*, einer XML-Datei, festgelegt. Außerdem werden alle verwendeten Komponenten darin definiert und mithilfe des Manifests die Komponentenzusammensetzung und Konfiguration der Applikation organisiert, beschrieben und dem Betriebssystem mitgeteilt.
In dieser Arbeit wird die Android-Komponente *Activity* verwendet [36] (siehe Abschnitt 5.3.1). Eine Activity entspricht der Benutzeroberfläche der Applikation und wird für die Darstellung von einer Bildschirmseite implementiert. Für jede weitere Bildschirmseite wird eine neue Activity erstellt. Jede Activity wird im Manifest beschrieben und dadurch registriert. Alle grafischen Elemente, die in einer Activity sichtbar sind, werden als *Views* bezeichnet. In Android besteht die grafische Darstellung aus einem Baum von Views, welche mithilfe einer XML-Datei (*Layout*) pro Activity organisiert und beschrieben werden. Die Android-Komponente Activity hat nach dem Start einen Lebenszyklus mit vier verschiedenen Zuständen und besitzt implementierbare Methoden, welche beim Zustandswechsel aufgerufen werden. Der Ablauf des Zyklus wird in Abbildung 5.2 dargestellt. Mithilfe dieses Lebenszyklus können in Android die zur Verfügung stehenden Ressourcen effizient genutzt werden. Abhängig vom Zustand der Komponenten kann das Betriebssystem bei Bedarf Ressourcen für andere Anwendungen freigeben oder bei einem erneuten Aufruf der Komponente die Ladezeit der Anwendung verringert werden.

Die vier Zustände einer Activity mit den dazugehörigen Methoden [36]:

aktiv (running): Die Activity wird im Vordergrund am Bildschirm angezeigt. Beim Start einer Activity wird die `onCreate`-Methode für die Initialisierung aufgerufen und anschließend die Methoden `onStart` und `onResume`. Danach ist die Activity sichtbar und aktiv, sowie für die Interaktion mit dem Benutzer bereit.

pausiert (pause): Die Activity wird teilweise von einer anderen Activity überlagert und verdeckt. Sie ist noch aktiv und behält alle Informationen über Zustände und Komponenten, kann aber bei Ressourcenknappheit vom Betriebssystem beendet werden. Beim Pausieren der Activity wird die Methode `onPause` aufgerufen. Mittels `onResume` kann diese wieder in den Vordergrund geholt und reaktiviert werden.

gestoppt (stop): Die Activity wird vollkommen von einer anderen Activity überlagert und ist für den Benutzer nicht mehr sichtbar. Die Informationen über Zustand und Komponenten bleiben erhalten, allerdings wird die Activity bei Speicherbedarf vom Betriebssystem

beendet. Die Methode `onStop` wird beim Stoppen der Activity aufgerufen und beim Aktivieren die Methoden `onRestart`, `onStart` und `onResume`, bevor die Acivity wieder sichtbar ist.

beendet (shut down): Die Activity war pausiert oder gestoppt und wurde vom System oder durch den Befehle `finish` beendet. Zuvor wird noch die Methode `onDestroy` aufgerufen. Bei einem erneuten Start muss die Activity neu erzeugt werden (`onCreate`).

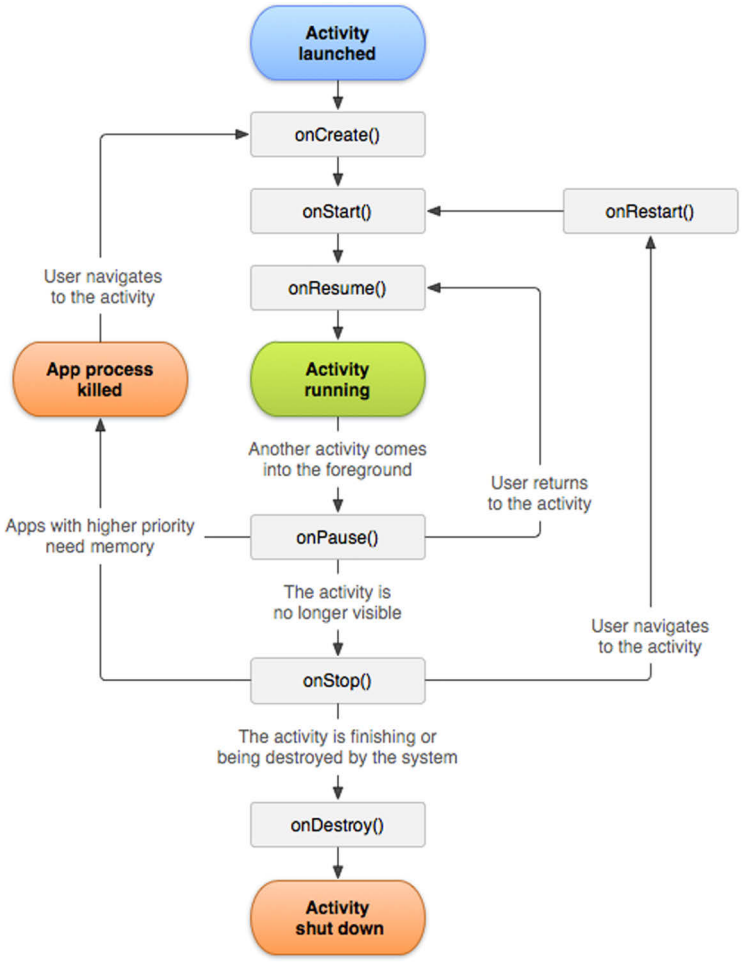


Abbildung 5.2: Lebenszyklus einer Android Activity [36]

5.2.2 Unity3D

Die VR-Szene und Interaktionslogik werden mithilfe von Unity3D implementiert. Unity3D ist eine Laufzeit- und Entwicklungsumgebung für interaktive 2D- und 3D-Inhalte [122]. Sie steht für verschiedene Plattformen kostenlos zu Verfügung (Basisversion) und unterstützt das mobile Betriebssystem Android. Unity stellt einfache Objekte wie Lichtquellen und grafische Primitive zur Verfügung und ermöglicht es, 3D-Modelle, Animationen, Texturen usw. zu importieren. Außerdem können mit Unity physikalische Eigenschaften wie Kollisionen, Kräfte und Verbindungen zwischen Objekten in Echtzeit simuliert werden.
Alle verwendeten Ressourcen werden in Unity als *Assets* bezeichnet. Mit der grafischen Entwicklungsumgebung von Unity können Szenen erstellt werden. Eine Szene ist als Szenengraph hierarchisch organisiert und besteht aus verschiedenen Objekten, welche in der Szene von *Game Objects* repräsentiert werden. Game Objects können mittels *Components* in ihrer Funktionalität und Darstellung erweitert werden. Das Verhalten und der Einfluss dieser Components auf das Game Object wird mithilfe von Variablen eingestellt. Zusätzlich kann das Verhalten der Game Objects mit Skripten gesteuert bzw. die Funktionalität erweitert werden. In dieser Arbeit werden C#-Skripten für die Umsetzung der Interaktionslogik verwendet. Alle Elemente von Unity sind ausführlich in der Dokumentation in [121] beschrieben und die für diese Arbeit relevanten Elemente werden nachfolgend erläutert:

Szene: Eine Szene entspricht in Unity einem virtuellen 3D-Raum, in dem sich die verschiedenen Objekte befinden. Dadurch lassen sich Projekte in verschiedene Szenen aufteilen und Ladezeiten verkürzen. In dieser Arbeit wird eine Szene verwendet.

Game Object: Jedes Objekt, das in der Szene platziert wird, ist ein Game Object. Wie zuvor erwähnt, können die Eigenschaften dieser Objekte durch das Hinzufügen von Components erweitert werden. Jedes Game Object besitzt die Component *Transform*, womit die Position, Orientierung und Skalierung des Objekts spezifiziert wird. Außerdem können Game Objects zu hierarchischen Gruppen zusammengefasst werden. In einer solchen Gruppe bestimmt das Wurzelobjekt die Position, Orientierung und Skalierung der gesamten Gruppe (alle Objekte beziehen sich relativ auf das Wurzelobjekt).

Component: Ein Component wird einem Game Object hinzugefügt, um dessen Funktionalität zu erweitern oder bestimmte Eigenschaften des Objekts zu definieren. In dieser Arbeit wird für die virtuelle Kamera die Component *Camera* und für die Beleuchtung der VR-Szene die Component *Light* verwendet. Physikalische Eigenschaften werden einem Objekt mit der Component *Rigidbody* hinzugefügt. Für die Kollisionsabfrage benötigen die Objekte *Collider* Components, mit denen die Objektgrenzen definiert werden (entsprechend der Objektform beispielsweise *BoxCollider* und *CapsuleCollider*).

Skripten: Für die Erweiterung und individuelle Anpassung der Funktionalität von Game Objects bietet Unity die Möglichkeit Skripten zu erstellen. Diese werden dem Objekt wie eine Component hinzugefügt. Es werden die Skriptsprachen UnityScript (vergleichbar mit JavaScript), Boo (vergleichbar mit Python) und C# unterstützt. Die Skripten in dieser Arbeit implementieren die Interaktionslogik. Sie werden von der Unity-Klasse

`MonoBehaviour` abgeleitet, wodurch das Skript einem Game Object als Component hinzugefügt und auf Systemfunktionen bzw. -ereignisse wie `Start`, `Update`, `OnGUI` und `OnTriggerEnter` zugegriffen werden kann. Die `Start`-Methode wird zu Beginn für die Initialisierung des Objekts genutzt. Unity berechnet und erzeugt ein Bild der VR-Szene (Rendering) und ruft einmal pro *Frame* (Einzelbild) die Methode `Update` auf. Mithilfe der `OnGUI`-Funktion wird die Benutzeroberfläche der VR-Szene implementiert und das Ereignis `OnTriggerEnter` wird für die Kollisionsabfrage genutzt (siehe Unity-Dokumentation [121]).

5.3 Zentrale Organisation und Kommunikation

Abbildung 5.3 gibt einen Überblick über die wichtigsten Komponenten der Zentrale, die die Organisation und Kommunikation beinhaltet.

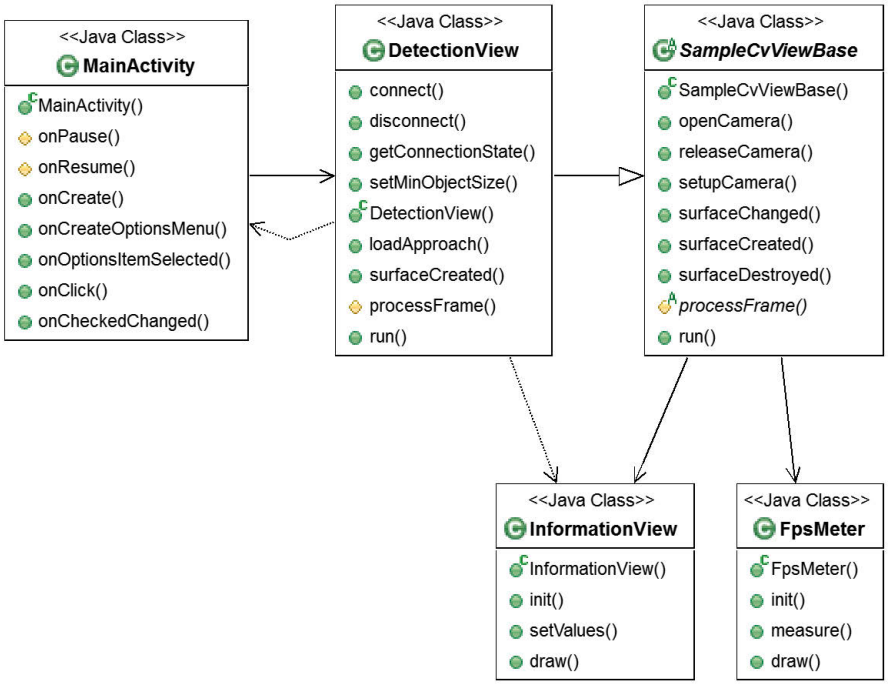


Abbildung 5.3: Klassendiagramm der wichtigsten Komponenten der Zentrale

5.3.1 Android-Manifest und Benutzeroberfläche

In Android wird mithilfe von Activities die Benutzeroberfläche der Applikation implementiert. Diese werden im Android-Manifest registriert und beschrieben (siehe Abschnitt 5.2.1). Mit der Klasse `MainActivity` wird die Benutzeroberfläche implementiert. Neben der Anzeige des Kamerabildes und der VR-Szene kann der Benutzer damit die verschiedenen Ansätze zur Fingergestenerkennung auswählen und diverse Einstellungen tätigen. Die aktuellen Einstellungsparameter werden den restlichen Klassen durch `public static`-Variablen zur Verfügung

gestellt. Außerdem werden in der Activity die View-Komponenten initialisiert und die RGB-Kamera gestartet.

5.3.1.1 Manifest

Damit beim Start der Applikation `MainActivity` ausgeführt und angezeigt wird, muss für die Activity ein entsprechender *Intentfilter* [38] im Manifest definiert werden (siehe Listing 5.1):

```
1 <activity android:name="MainActivity">
2   <intent-filter>
3     <action android:name="android.intent.action.MAIN" />
4     <category android:name="android.intent.category.LAUNCHER" />
5   </intent-filter>
6 </activity>
```

Listing 5.1: Definition des Intentfilters für MainActivity

Außerdem werden mittels des Manifests die von der Anwendung benötigten Berechtigungen erteilt (siehe Listing 5.2), um auf die Kamera (Zeile 1-3), Netzwerkverbindung (Zeile 4) und externen Speicher (Zeile 5) zugreifen zu können:

```
1 uses-feature android:name="android.hardware.camera"/>
2 uses-feature android:name="android.hardware.camera.autofocus"/>
3 uses-permission android:name="android.permission.CAMERA"/>
4 uses-permission android:name="android.permission.INTERNET"/>
5 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Listing 5.2: Definition der Berechtigungen der Android-Applikation

5.3.1.2 Aufbau der Benutzeroberfläche

Zu Beginn der Activity wird das Layout der Anwendungsoberfläche geladen. Darin werden die Bildschirmelemente der Anwendung (Views) definiert und beschrieben. Die Benutzeroberfläche wird in dieser Arbeit mithilfe von zwei *FrameLayout*-Komponenten [37] organisiert und in zwei Bereiche aufgeteilt (Abbildung 5.4). Dadurch können das Kamerabild (FrameLayout1) und die VR-Szene (FrameLayout2) nebeneinander angezeigt werden. Beim Start der Activity werden dafür die View-Komponenten `DetectionView` und `UnityPlayer` initialisiert und anschließend jeweils dem entsprechenden FrameLayout hinzugefügt. Mit dieser, wie in [3] beschriebenen, Vorgehensweise, ist es nicht nur möglich Unity3D in eine Android-Anwendung zu integrieren, sondern auch einem individuellen Layout zuzuweisen.

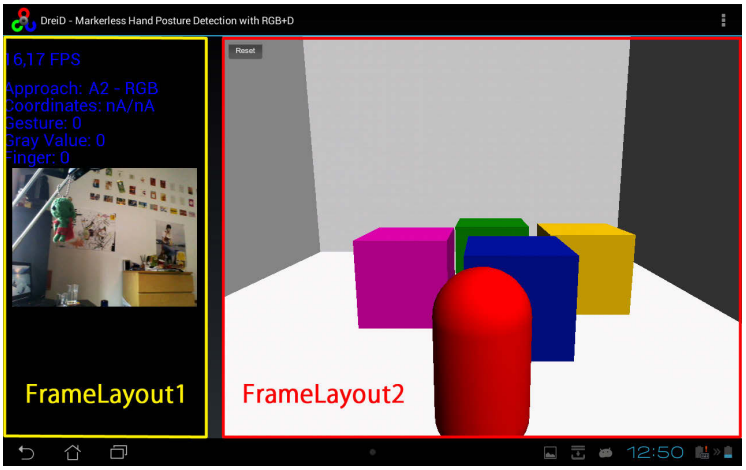


Abbildung 5.4: Layout der Benutzeroberfläche

Die weiteren Elemente der Benutzeroberfläche, bestehend aus der Info-Box (grün), einem Menü (gelb) und einem Einstellungsbildschirm (rot), werden in Abbildung 5.5 dargestellt und nachfolgend im Detail erläutert. Das Menü wird per Klick auf die drei Punkte in der rechten oberen Bildschirmecke aufgerufen und der Einstellungsbildschirm über das Menü.

Abbildung 5.5: Info-Box, Menü und Einstellungsbildschirm der Benutzeroberfläche

5.3.1.3 Info-Box

In der Info-Box werden zusätzliche Informationen für den Benutzer angezeigt (siehe Abbildung 5.6a). Die aktuelle Framerate wird mithilfe der Klasse `FpsMeter` gemessen und am Bildschirm ausgegeben. Sie wird von der Klasse `SampleCvViewBase` initialisiert und ein Mal pro Frame für die Messung und Ausgabe aufgerufen. Mittels der Klasse `InformationView` wird der verwendete Ansatz (A1 oder A2) und die aktuelle 2D-Position der Hand angezeigt. Außerdem wird die erkannte Geste und Fingerspitzenanzahl, sowie der maximale Grauwert der Hand

ausgegeben. `InformationView` wird ebenfalls von `SampleCvViewBase` initialisiert und ein Mal pro Frame für die Bildschirmausgabe aufgerufen. Die Klasse `DetectionView` ruft `InformationView` für die Aktualisierung der auszugebenden Informationen auf.

(a) Info-Box (b) Menü

Abbildung 5.6: Detailansicht der Info-Box und des Menüs

5.3.1.4 Menü

Das Menü setzt sich aus den folgenden Optionen zusammen (Abbildung 5.6b):

- **Minimal Objectsize:** Ändern der minimal erkennbaren Objektgröße der Klassifikatoren
- **Toggle Camera View:** Ausblenden bzw. Einblenden des Kamerabilds
- **Toggle Unity/Settings:** Anzeigenwechsel zwischen der VR-Szene und dem Einstellungsbildschirm (durch Ausblenden der VR-Szene)
- **Approach:** Auswahl von A1 bzw. A2 (die Untertypen von A2 werden über die Einstellungen gestartet und konfiguriert, siehe unten)

5.3.1.5 Einstellungsbildschirm

Der Einstellungsbildschirm ist in fünf Bereiche unterteilt mit den folgenden Auswahl- und Einstellungsmöglichkeiten (Abbildung 5.7):

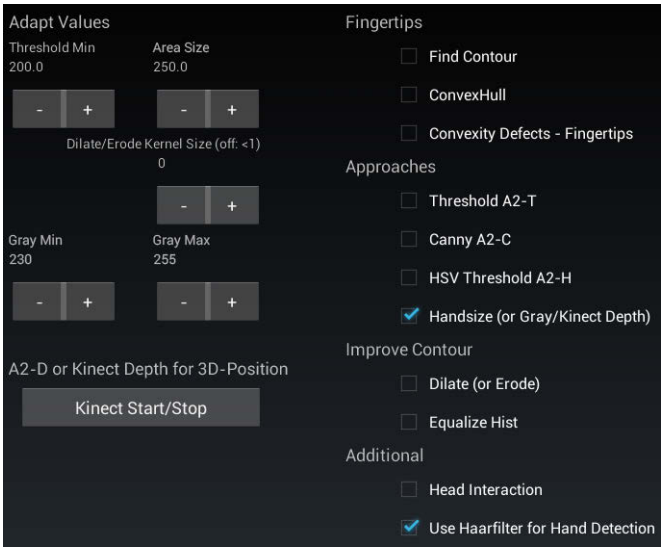


Abbildung 5.7: Detailansicht des Einstellungsbildschirms

- **Adapt Values** (Verändern der Werte mittels der -/+ Schaltfläche)
  - Threshold Min: Adaptierung des verwendeten Schwellwerts für A2-T
  - Area Size: Einstellung der kleinsten akzeptierten Konturfläche in Pixel, um kleine Regionen herausfiltern zu können
  - Dilate/Erode Kernel Size: Veränderung der Kernelgröße für die morphologische Funktion Dilatation bzw. Erosion (0 deaktiviert die Funktion)
  - Gray Value Min/Max: Anpassung des minimalen bzw. maximalen Grauwerts (Grauwertbereich) zur Ermittlung der 3D-Position
- **A2-D or Kinect Depth for 3D-Position**
  - Kinect Start/Stop: Starten bzw. Beenden der Kinect und der Erzeugung von RGBD-Daten für A2-D und die absolute 3D-Position
- **Fingertips**
  - Find Contour: Aktivieren/Deaktivieren der Konturermittlung
  - ConvexHull: Ermittlung der konvexen Hülle ein-/ausschalten (benötigt die Kontur der Hand)
  - Convexity Defects - Fingertips: Starten/Beenden der Fingerspitzenermittlung (erfordert die Kontur und konvexe Hülle der Hand)
- **Approaches**
  - Threshold A2-T: Verwenden von A2-T

  - Canny A2-C: A2-C aktivieren/deaktivieren
  - HSV Threshold A2-H: Anwendung von A2-H
  - Handsize (or Gray/Kinect Depth): Wechsel zwischen der Handgröße (aktiviert) und Grauwert (deaktiviert) für die Ermittlung der 3D-Position. Stehen RGBD-Daten zur Verfügung werden diese anstatt dem Grauwert zur Bestimmung der 3D-Position verwendet.

- **Improve Contour**
  - Dilate (or Erode): Anwendung der Dilatation (aktiviert) bzw. Erosion (deaktiviert)
  - Equalize Hist: Ein- bzw. Ausschalten des Histogrammausgleichs

- **Additional**
  - Head Interaction: Aktivieren/Deaktivieren der Interaktion mittels Kopfposition
  - Use Haarfilter for Hand Detection: Verwendung von Haar-Klassifikatoren zur Ermittlung der Hand (standardmäßig aktiviert)

5.3.2 Zentrale

Die zentrale Klasse des Prototyps ist `DetectionView` (siehe Abbildung 5.3). Darin werden die verschiedenen Daten und Ergebnisse zusammengeführt und für die weitere Verarbeitung an die entsprechende Methode weitergegeben. Außerdem werden die Klassifikatoren geladen und die für die Interaktion benötigten Daten an die VR-Szene übertragen.

5.3.2.1 SampleCvViewBase

`DetectionView` erbt von der Klasse `SampleCvViewBase` und diese erbt wiederum von der Android View-Komponenten `SurfaceView`. Ein SurfaceView [41] entspricht einer Zeichenfläche und wird für die Anzeige des Kamerabilds verwendet.

Zu Beginn wird die Auflösung des Kamerabilds festgelegt (320x240 Pixel) und per *UnitySendMessage* [120] an das entsprechende Skript der VR-Szene weitergegeben (siehe Listing 5.3), um die Konstanten zur Transformation der Hand-und Kopfposition in die VR-Szene berechnen zu können (siehe Abschnitt 5.7):

```
1 //Uebertragung der Aufloesung an Unity
2 UnityPlayer.UnitySendMessage("sceneGO", "JavaInit", ""+320+";"+240);
```

Listing 5.3: Kommunikation mit der VR-Szene bzw. Unity

Der erste Parameter bezeichnet das Game Object der VR-Szene. Mit dem zweiten Parameter wird die aufzurufende Funktion des Skripts angegeben und der dritte Parameter enthält die zu übertragende Nachricht[2].

[2]Die einzelnen Werte werden mittels ';' getrennt, um sie in Unity aus der Nachricht extrahieren zu können.

Anschließend wird mithilfe der OpenCV-Bibliothek die Frontkamera des Tablets geöffnet und in einem Thread kontinuierlich ausgelesen (Abschnitt 5.4.1). In diesem Thread wird ein Mal pro Frame die Methode `processFrame` der Klasse `DetectionView` aufgerufen und das aktuelle Kamerabild (RGB-Daten) zur weiteren Verarbeitung übergeben.

5.3.2.2 DetectionView

In `DetectionView` werden die RGB-Daten verwaltet und verteilt. Die Tiefendaten werden in einem eigenen Thread empfangen (Abschnitt 5.4.2) und stehen für die weitere Verarbeitung in `DetectionView` direkt zur Verfügung. In der Methode `processFrame` werden die Daten verarbeitet und die weiteren Schritte für die Detektion, Mapping, Gestenerkennung und Interaktion eingeleitet. Nachfolgend wird der grundlegende Ablauf der Methode `processFrame` für ein Frame beschrieben und die ermittelten Daten bzw. Informationen angegeben:

1. Aufruf von `processFrame` durch `SampleCvViewBase` und Übergabe des aktuellen Frames (RGB-Daten) (Abschnitt 5.4.1)
2. Detektion der Hand aufrufen: 2D-Position und Größe der Hand für relative Tiefenschätzung (Abschnitt 5.5)
3. Detektion des Kopfs aufrufen: 2D-Position und Größe des Kopfs für relative Tiefenschätzung (Abschnitt 5.6)
4. Mapping von RGB- und Tiefendaten starten: RGBD-Daten und absolute Tiefenschätzung der Hand (Abschnitt 5.4.2)
5. Maximalen Grauwert der Hand berechnen für relative Tiefenschätzung
6. Fingergestenerkennung aufrufen: Erkannte Geste (Abschnitt 5.5)
7. Informationen anzeigen (Objektrahmen, Mittelpunkt, Framerate, etc.)
8. Übertragung der Geste und 3D-Position an Unity: Interaktion mit der VR-Szene (Abschnitt 5.7)
9. Bild erzeugen: Für die Ausgabe am Tablet an `SampleCvViewBase` zurückgeben

Je nach Einstellungen des Benutzers (getätigt über die Benutzeroberfläche) kann das Verwenden der Kopfinteraktion, der Tiefendaten und des Grauwerts zur relativen Tiefenschätzung eingestellt werden. Dementsprechend werden die einzelnen Schritte durchgeführt oder übersprungen. Die weiterführenden Schritte werden in den angegeben Abschnitten näher erläutert. Für die Übertragung der Geste und der 3D-Position an Unity wird wiederum UnitySendMessage [120] verwendet (siehe Listing 5.3).

5.4 Bilddaten einlesen und Mapping

Einen Überblick der wichtigsten Komponenten zur Ermittlung der Bilddaten liefert Abbildung 5.8. Als Schnittstelle zwischen den Java- und C++-Klassen wird JNI eingesetzt. Die Bilddaten werden in eigenen Threads erfasst und für die Weiterverarbeitung mittels OpenCV in Matrizen gespeichert. In den nächsten Abschnitten werden die Klassen und Methoden zur Ermittlung der Daten im Detail erläutert.

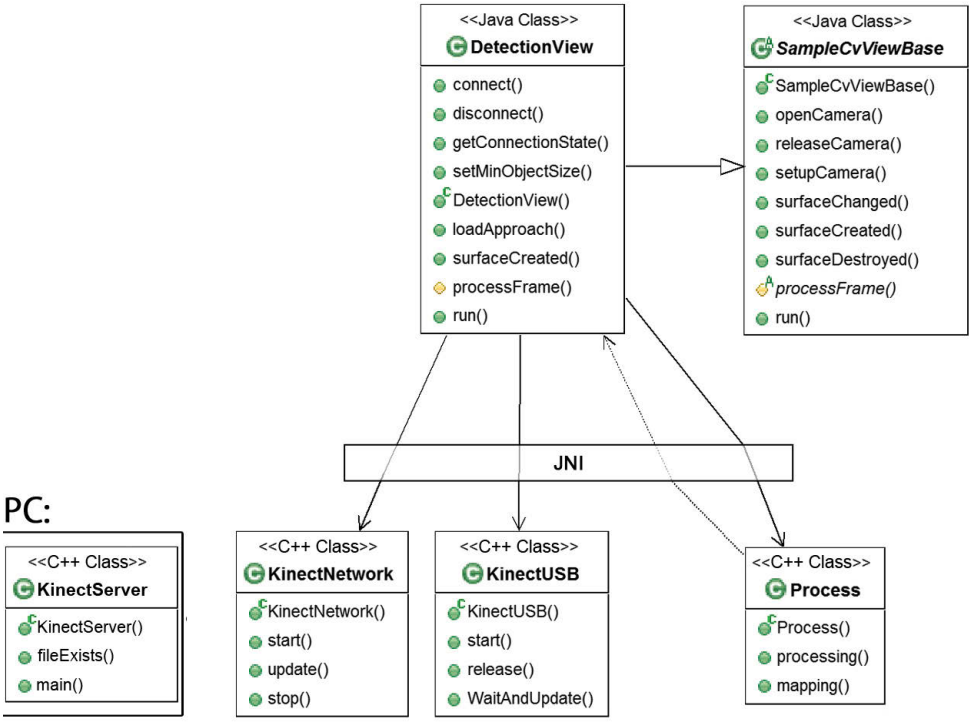


Abbildung 5.8: Klassendiagramm der wichtigsten Komponenten zur Ermittlung der Bilddaten

5.4.1 Einlesen der RGB-Daten

In der Klasse `SampleCvViewBase` wird mithilfe der OpenCV-Bibliothek [90] ein Kamera-Objekt vom Typ `VideoCapture` [94] angelegt, welches die Frontkamera des Tablets repräsentiert. Beim Start der Applikation wird die Methode `openCamera` aufgerufen, um die Kamera zu öffnen. Im nächsten Schritt wird die Methode `setupCamera` ausgeführt und darin die Auflösung der Kamerabilder auf 320x240 Pixel festgelegt. Für den Empfang des Kamerabilds (RGB-Daten) wird ein eigener Thread gestartet. Damit wird das Kamerabild kontinuierlich ausgelesen, ohne die anderen Komponenten der Anwendung zu blockieren. Nachdem ein Frame der Kamera empfangen wurde, wird er für die weitere Verarbeitung an `DetectionView` übergeben. Beim Beenden der Applikation wird die Kamera mittels `releaseCamera` geschlossen und freigegeben.

5.4.2 Einlesen der Tiefendaten

Für das Einlesen der Tiefendaten der Kinect wird in `DetectionView` ein eigener Thread verwendet. Der Thread bzw. das Einlesen der Tiefendaten wird zu Beginn der Anwendung mit der Methode `connect` gestartet und lässt sich zusätzlich über die Benutzeroberfläche steuern (Start/Stop). Zuerst wird versucht, die Verbindung zur Kinect per USB herzustellen. Ist kein Gerät angeschlossen/erreichbar, wird die Netzwerk-Variante gestartet (siehe Abschnitt 3.1.3). Kann mit beiden Varianten keine Verbindung hergestellt werden, wird der Thread beendet. Wurde die Verbindung zur Kinect erfolgreich hergestellt und die Tiefendaten empfangen, werden diese für das Mapping und die Erzeugung von RGBD-Daten verwendet (Abschnitt 5.4.3). Durch Aufruf der Methode `disconnect` wird die Verbindung zur Kinect beendet und der Thread geschlossen. Die Umsetzung dieser Schritte wird in den folgenden Abschnitten näher beschrieben.

5.4.2.1 Zugriff auf Tiefendaten mittels USB

Im Thread von `DetectionView` wird die `start`-Methode der Klasse `KinectUSB` aufgerufen, um eine Verbindung zur Kinect herzustellen. Konnte die Verbindung erfolgreich hergestellt werden, wird die Methode `WaitAndUpdate` kontinuierlich aufgerufen und die Tiefendaten empfangen. Zum Beenden der Verbindung wird die `release`-Methode ausgeführt.
Die Klasse `KinectUSB` basiert auf [68] und verwendet das OpenNI SDK [97] für die Umsetzung. Um eine Verbindung mit der Kinect aufbauen zu können, wird in der Methode `start` als Erstes ein `Context`-Objekt erstellt, wodurch die Kinect repräsentiert wird. Für die Initialisierung dieses Objekts wird eine XML-Datei verwendet. Darin werden die verwendeten Elemente der Kinect definiert und zusätzliche Eigenschaften wie die Auflösung festgelegt. Das `Context`-Objekt [95] wird nach dem Tiefenkamera-Element durchsucht, welches anschließend einem `DepthGenerator`-Objekt [96] zugewiesen und für den Zugriff auf die Tiefenkamera bzw. Tiefendaten verwendet wird. Für den Abruf der aktuellen Tiefendaten wird die Methode-`WaitAndUpdate` aufgerufen. Darin wird mithilfe des `Context`-Objekts die Kinect dazu aufgefordert die Daten zu aktualisieren und mittels `DepthGenerator`-Objekt auf das aktuelle Tiefenbild der Kamera zugegriffen. Der in dieser Arbeit verwendete Treiber [2] unterstützt nur die Tiefenkameraauflösung 640x480 Pixel, weshalb eine Umwandlung in die Auflösung der RGB-Daten (320x240 Pixel) notwendig ist. Dazu wird OpenCV (`resize` [88]) und die Interpolationsmethode *nearest-neighbor* verwendet. Zum Beenden der Verbindung werden die `Context-` und `DepthGenerator`-Objekte in der Methode `release` freigegeben.

5.4.2.2 Zugriff auf Tiefendaten mittels Netzwerk

In der Netzwerk-Variante wird die Kinect an einen PC angeschlossen und die Treiber des *Kinect für Windows SDK* [76] für den Zugriff verwendet. Damit ist es möglich, den *Near-Mode* der Kinect zu nutzen [73] und die Auflösung der Tiefenbilder auf 320x240 Pixel einzustellen. Die Tiefendaten werden mittels OpenNI [97] in der Klasse `KinectServer` eingelesen. Aufgrund der Inkompatibilität des Kinect für Windows SDK mit OpenNI wird die *Kinect-MSSDK-OpenNI-Bridge* [129] als Schnittstelle benötigt, um mittels OpenNI auf die Kinect zugreifen zu können. Anschließend wird mithilfe der Bibliothek Boost [11] eine Netzwerkverbindung (WLAN) zum Tablet (Klasse `KinectNetwork`) aufgebaut und die Tiefendaten übertragen.

Die Implementierung des Verbindungsaufbaus zur Kinect und das Auslesen der Tiefenkamera mittels OpenNI ist vergleichbar mit der zuvor beschriebenen Klasse `KinectUSB` (siehe oben). Zu Beginn wird für den Verbindungsaufbau zur Kinect ein `Context`-Objekt erstellt und mittels XML-Datei initialisiert. Für den Zugriff auf die Tiefenkamera wird wiederum ein `DepthGenerator`-Objekt erstellt. Die Übertragung der Tiefendaten zum Tablet wurde basierend auf dem Server-Client Beispiel von Boost in [9, 10] implementiert. Der Server wird mit der Klasse `KinectServer` umgesetzt und der Client in der Klasse `KinectNetwork` implementiert. Als Übertragungsprotokoll wird TCP/IP[3] eingesetzt. Der Server wartet auf eine Verbindungsanfrage vom Client. Zum Stellen der Anfrage wird im Thread DetectionView die Methode `start` der Klasse `KinectNetwork` aufgerufen. Nachdem die Anfrage vom Server angenommen wurde, ist die Verbindung zwischen PC und Tablet hergestellt. In einer Endlosschleife des Servers werden nun die Tiefendaten eingelesen, für die Übertragung in ein eindimensionales `unsigned short`-Array gespeichert und an den Client gesendet. Am Tablet wird im Thread die Methode `update` von `KinectNetwork` (Client) kontinuierlich aufgerufen, um die aktuellen Tiefendaten vom Server zu empfangen. Das übermittelte Array aus Tiefendaten wird für die weitere Verarbeitung in einer $m \times n$ Matrix gespeichert. Die Verbindung zwischen Server und Client bleibt solange aufrecht, bis sie von einer Seite beendet wird (zum Beispiel durch Aufruf der `stop`-Methode) oder ein Verbindungsfehler auftritt.

5.4.3 Erzeugung von RGBD-Daten mittels Mapping

Zur Erzeugung von RGBD-Daten wird in der Klasse `DetectionView` die Methode `mapping` der Klasse `Process` aufgerufen. Darin werden den RGB-Pixeln in der zuvor ermittelten Handregion die entsprechenden Tiefenwerte zugeordnet und abschließend in einer vierdimensionalen $m \times n$ Matrix gespeichert (RGBD-Daten). Außerdem wird in Zuge dessen der kleinste Tiefenwert der Hand für A1-D und A2-D (Abschnitt 5.5.4) zur absoluten Schätzung der 3D-Position der Hand gespeichert und die Pixel vor bzw. hinter der Hand zur Segmentierung herausgefiltert (A2-D). Listing 5.4 beschreibt die Umsetzung des in Abschnitt 3.1.4 beschriebenen Mappings in Pseudocode:

```
1  FUER ALLE Handregionpixel(x,y) WIEDERHOLE
2    z = Tiefenwert(x,y)
3
4    //Ermittlung des minimalen Tiefenwerts der Hand
5    WENN z > 0 und z < z_hand DANN
6      z_hand = z
7    WENN_ENDE
8
9    //Umrechnung der Tiefendaten in das 3D-Koordinatensystem der Infrarotkamera
10   P3D_X(x,y) = (x - Kamerahauptpunkt_IR_X) * z / Brennweite__IR_X
11   P3D_Y(x,y) = (y - Kamerahauptpunkt_IR_Y) * z / Brennweite_IR_Y
12   P3D_Z(x,y) = z
13 WIEDERHOLE_ENDE
14
15 //Transformation von P3D in das Koordinatensystem der RGB-Kamera
16 TRANSFORMATION(P3D,Extrinsische_Matrix)
```

[3] Transmission Control Protocol/Internet Protocol.

```
17 FUER ALLE Handregionpixel(x,y) WIEDERHOLE
18   z = P3D_Z(x,y)
19   //Transformation von P3D in die Bildebene der RGB-Kamera
20   x_rgb = (P3D_X(x,y) * Brennweite_RGB_X/z) + Kamerahauptpunkt_RGB_X
21   y_rgb = (P3D_Y(x,y) * Brennweite_RGB_Y/z) + Kamerahauptpunkt_RGB_Y
22
23   //Speichern des RGB-Werts in RGBD, wenn der Pixel in der Handebene liegt
24   //Herausfiltern der anderen Pixel (Segmentierung fuer A2-D)
25   WENN z > z_hand-100 und z < z_hand+100 DANN
26     RGBD(x_rgb,y_rgb) = RGB(x_rgb,y_rgb)
27   WENN_ENDE
28   //Speichern des Tiefenwerts in RGBD
29   RGBD(x_rgb,y_rgb) = z
30 WIEDERHOLE_ENDE
```

Listing 5.4: Mapping in Pseudocode

5.5 Ermittlung der Fingergesten

In Abbildung 5.9 werden die wichtigsten Komponenten zur Fingergestenermittlung dargestellt. Zwischen den Java- und C++-Klassen wird JNI als Schnittstelle eingesetzt. Mithilfe von RGB- bzw. RGBD-Daten werden die Fingergesten und 3D-Position für die Interaktion mit der VR-Szene ermittelt. Die Implementierung davon wird in den nachfolgenden Abschnitten erläutert.

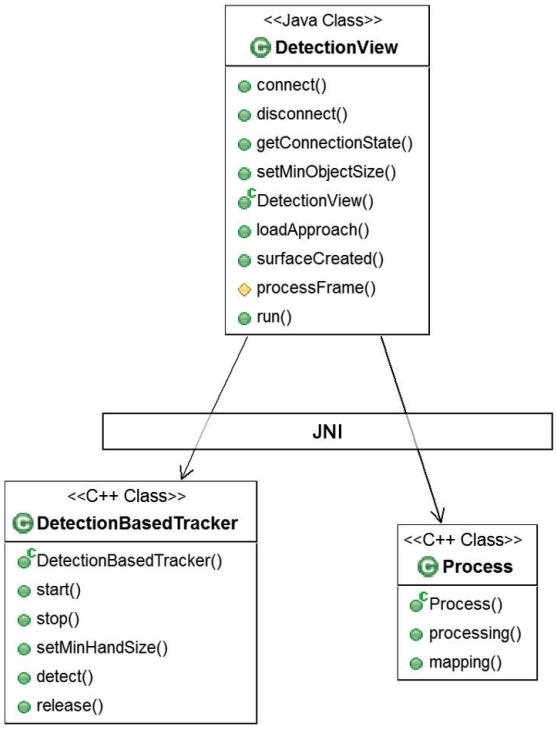


Abbildung 5.9: Klassendiagramm der wichtigsten Komponenten zur Fingergestenerkennung

5.5.1 A1 (RGB)

Für die Umsetzung von A1 werden die RGB-Daten des aktuellen Frames und die Methoden von OpenCV verwendet. Der theoretische Ablauf von A1 ist in Abschnitt 3.4.1 beschrieben. Die Ermittlung der Gesten wird mithilfe von zwei Haar-Klassifikatoren K1 und K2 durchgeführt. Diese werden von `DetectionBasedTracker`-Objekten repräsentiert und beim Start der Applikation in der Klasse `DetectionView` initialisiert. Die Klasse `DetectionBasedTracker` wird von OpenCV zur Verfügung gestellt und verwendet `detectMultiScale` [86] für die Detektion.

Die Ermittlung der Fingergesten und deren 3D-Position ist in der Methode `processFrame` der Klasse `DetectionView` implementiert und setzt sich aus den nachfolgenden Schritten zusammen.

5.5.1.1 Detektion der Fingergeste

In der `processFrame`-Methode wird das aktuelle Kamerabild nach den zwei Gesten durchsucht. Haar-Klassifikatoren nutzen die Helligkeitsunterschiede in Graustufenbildern für die Detektion, weshalb für die weiteren Schritte das Graustufenbild des aktuellen Frames verwendet wird. Als Erstes wird das Bild mittels K1 auf *Geste 1* hin untersucht. Konnte diese nicht gefunden werden, wird mithilfe von K2 nach *Geste 2* gesucht. Wurde keine Geste gefunden, wird die Suche im nächsten Frame fortgesetzt. Konnte in den vorangegangenen zehn Frames eine Geste ermittelt werden, wird jedoch diese übernommen um kurze Aussetzer zu überbrücken, und das nächste Bild nur auf diese Geste untersucht. Dies dient zur Verbesserung der Performance, da nur mehr nach einer Geste gesucht wird. Für die Detektion einer Geste wird die Methode `detect` des `DetectionBasedTracker`-Objekts aufgerufen und das zu durchsuchende Graustufenbild übergeben. Die gefundenen Objekte werden von Rechteck-Objekten (Handregion) repräsentiert und stellen die Größe und 2D-Position des Objekts zur Verfügung. Initiale empirische Tests haben ergeben, dass die Verwendung der als Erstes gereihten Funde zu robusten Erkennungsraten führen. Zudem wurde in `detectMultiScale` festgelegt, dass mindestens vier Nachbarobjekte für die erfolgreiche Detektion eines Objekts benötigt werden, wodurch kaum mehr als ein Objekt pro Bild gefunden wird. Aus diesen Gründen wird das erste gefundene Objekt als die gesuchte Geste angenommen. Der Bildausschnitt dieser Handregion wird gespeichert und zur besseren Erfassung der gesamten Handregion vergrößert[4]. Die Größe des Rechteck-Objekts wird ebenso angepasst und für die Schätzung der 3D-Position verwendet. Listing 5.5 beschreibt den Ablauf der Detektion in Pseudocode:

```
1 //Suche Geste 1
2 WENN suche != geste2 DANN
3   gefundene_objekte = K1.DETEKTION(graustufenbild)
4 WENN_ENDE
5
6 //Suche Geste 2
7 WENN gefundene_objekte < 1 und suche != geste1 DANN
```

[4] Mithilfe von empirisch ermittelten Erfahrungswerten relativ zur ermittelten Größe wird der Bildausschnitt um 8 % (bzw. bis zur Bildbegrenzung) in alle Richtungen erweitert, sodass die gesamte Handfläche abgedeckt wird.

```
8   gefundene_objekte = K2.DETEKTION(graustufenbild)
9   WENN gefundene_objekte > 0 DANN
10    suche = geste2
11  WENN_ENDE
12 SONST
13  suche = geste1
14 WENN_ENDE
15
16 //Das erste gefundenen Objekt wird als Geste angenommen
17 WENN gefundene_objekte > 0 DANN
18  handregion = gefundene_objekte.ERSTES_OBJEKT()
19
20  //Vergroessern des Bildausschnitts der Hand und der Handregion
21  hand = AUSSCHNITT(graustufenbild, handregion)
22  hand = VERGROESSERN(hand, hoehe, breite)
23  handregion = RECHTECK(hand)
24
25 //Wurde nichts gefunden wird der vorige Fund verwendet
26 SONST
27  pause++
28  //Wurde 10 Frames lang nichts gefunden, wird die Suche neugestartet
29  WENN pause > 9 DANN
30    pause = 0
31    handregion = 0
32    suche = neu
33  WENN_ENDE
34 WENN_ENDE
```

Listing 5.5: Detektion der Fingergeste mittels A1 in Pseudocode

5.5.1.2 Relative Schätzung der 3D-Position

Im nächsten Schritt wird die 3D-Position mithilfe der Handgröße oder dem maximalen Grauwert der Hand geschätzt. Die gewünschte Methode wird vom Benutzer über die Einstellungen ausgewählt. In Abschnitt 5.5.3 wird die Umsetzung der relativen Schätzung genauer beschrieben.

5.5.1.3 Aufruf der Interaktion

Als letzter Schritt wird anhand der ermittelten Fingergesten die entsprechende Methode in Unity zur Durchführung der Interaktion aufgerufen. Die Umsetzung des Aufrufs wird in Abschnitt 5.5.5 im Detail beschrieben.

5.5.2 A2 (RGB)

Die Implementierung von A2 verwendet die RGB-Daten des aktuellen Kamerabilds zur Bestimmung der Fingergesten und wird in Abschnitt 3.4.2 theoretisch beschrieben. Für die Detektion der Hand wird der Haar-Klassifikator K3 und die Klasse `DetectionBasedTracker` mit der Funktion `detectMultiScale` [86] von OpenCV verwendet. Dafür wird zum Start der Anwendung in der Klasse `DetectionView` das `DetectionBasedTracker`-Objekt mit K3

initialisiert. In der ermittelten Handregion wird mithilfe von OpenCV die Fingerspitzenerkennung durchgeführt und zur Bestimmung der Fingergeste (Fingerspitzenanzahl) verwendet. Die relative Schätzung der 3D-Position wird entweder mit der Größe oder dem maximalen Grauwert der Hand durchgeführt.

In der `processFrame`-Methode der Klasse `DetectionView` wurden die im Anschluss beschriebenen Schritte zur Ermittlung der Fingergesten und deren 3D-Position implementiert.

5.5.2.1 Detektion der Hand

Aus den RGB-Daten des aktuellen Frames wird in `processFrame` als Erstes ein Graustufenbild für die weiteren Schritte erstellt. Das Graustufenbild wird (wie bei A1 in Abschnitt 5.5.1) für die Detektion der Hand mittels Haar-Klassifikator, die Segmentierung der Hand (außer A2-H[5]) und die Fingerspitzenerkennung verwendet. Die Hand wird mittels K3 ermittelt, wofür die `detect`-Methode des `DetectionBasedTracker`-Objekts aufgerufen wird. Die gefundenen Objekte werden von Rechteck-Objekten repräsentiert und in einer Matrix gespeichert. Das erste Rechteck-Objekt wird als Hand angenommen und stellt die 2D-Position sowie die Größe der Hand zu Verfügung. Diese Handregion wird als Bildausschnitt gespeichert und für die weiteren Schritte verwendet. Mit K3 wird die Handunterseite samt Daumen erkannt (siehe Abbildung 4.10c). Aus diesem Grund muss der ermittelte Bildausschnitt und die Handregion vergrößert werden, um die gesamte Hand samt Fingerspitzen abzudecken und eine Fingerspitzenerkennung zu ermöglichen. Dafür wird der Bildausschnitt vergrößert[6] und anschließend die Handregion an den neuen Ausschnitt angepasst. Diese werden in den nächsten Abschnitten für die Ermittlung der Gesten mittels Fingerspitzenerkennung und zur relativen Schätzung der 3D-Position verwendet. Der Bildausschnitt wird für die Umsetzung von A2-H aus dem RGB-Bild ermittelt und gespeichert, da dieser Ansatz die Farbinformationen zur Segmentierung benötigt.
Für die nächsten Schritte (Segmentierung, Ermittlung der Kontur und der Fingerspitzen) wird die Methode `processing` der Klasse `Process` aufgerufen und eine Referenz[7] auf den Bildausschnitt und die Fingerliste[8], sowie die aktuellen Einstellungswerte übergeben.
In Listing 5.6 wird die Detektion der Hand in Pseudocode verdeutlicht:

```
1 //Suche Hand
2 gefundene_objekte = K3.DETEKTION(graustufenbild)
3
4 //Das erste gefundene Objekt wird als Hand angenommen
5 handregion = gefundene_objekte.ERSTES_OBJEKT()
6
```

[5] A2-H verwendet für die Segmentierung einen Schwellwert basierend auf den HSV-Werten der Hand und benötigt dafür RGB-Daten.

[6] Der Bildausschnitt wird mit empirisch ermittelten Erfahrungswerten relativ zur ermittelten Handgröße um 100 % nach oben, 33 % nach links und 7 % nach rechts erweitert (bzw. bis zur Bildbegrenzung), sodass die gesamte Hand erfasst wird.

[7] Eine Referenz ist ein Verweis auf ein Objekt und ermöglicht den direkten Zugriff auf das Objekt. Dadurch werden in `processing` die Veränderungen direkt am Objekt in `processFrame` durchgeführt und müssen nicht mehr übergeben werden (call by reference).

[8] In der Fingerliste werden die ermittelten Fingerspitzen gespeichert und für die Bestimmung der Fingergeste verwendet.

```
7  //Vergroessern des Bildausschnitts der Hand und der Handregion
8  hand = AUSSCHNITT(graustufenbild, handregion)
9  hand = VERGROESSERN(hand, hoehe, breite)
10 handregion = RECHTECK(hand)
11
12 WENN Einstellung = A2-H DANN
13   //Bildausschnitt der Hand in RGB fuer A2-H
14   hand = AUSSCHNITT(rgb_bild, handregion)
15 WENN_ENDE
16
17 //Aufruf der Methode processing zur Ermittlung der Fingerspitzen
18 processing(hand, fingerliste, Einstellungen)
```

Listing 5.6: Detektion der Hand mittels A2 in Pseudocode

5.5.2.2 Segmentierung der Hand

In diesem Abschnitt wird die Implementierung der Handsegmentierung in der Methode `processing` der Klasse `Process` mittels A2 behandelt. Die verschiedenen Untertypen von A2 sind in Abschnitt 3.4.2 theoretisch beschrieben. Mithilfe der nachfolgend beschriebenen Methoden wird bei der Segmentierung ein binäres Bild erzeugt, welches sich aus der Hand in Weiß und dem Hintergrund in Schwarz zusammensetzt. Über die Einstellungen der Benutzeroberfläche kann zwischen A2-C, A2-H und A2-T gewählt und die weiteren Parameter (wie zum Beispiel der Schwellwert für A2-T) festgelegt werden (Abschnitt 5.3.1). Außerdem werden dort die Funktionen wie Histogrammausgleich, Dilatation und Erosion zur Verbesserung der Segmentierung aktiviert.

In der `processing`-Methode wurden die weiteren Schritte von A2 implementiert. Abhängig von den gewählten Einstellungen werden die entsprechenden Funktionen mit den festgelegten Parametern ausgeführt. Zu Beginn der Methode wird der Bildausschnitt mit einem Box-Filter [87] bearbeitet, um das Rauschen zu reduzieren. Außerdem wird eine schwarze elliptische Maske zur Reduktion des Hintergrunds über den Bildausschnitt gelegt[9]. Für die Implementierung des Histogrammausgleichs wurde die Funktion `equalizeHist` [88] von OpenCV verwendet.

Mithilfe des Kantendetektionsalgorithmus Canny [19] wird in A2-C der Bildausschnitt untersucht und in ein binäres Bild bestehend aus Kanten (weiß) und Hintergrund (schwarz) umgewandelt (siehe Abbildung 3.20). Für die Umsetzung der Kantendetektion wird die Canny-Funktion [85] von OpenCV verwendet. Der Mittelwert M und die Standardabweichung σ des Bildausschnitts werden mithilfe von OpenCV's `meanStdDev` [92] berechnet und $M \pm \sigma$ als minimaler bzw. maximaler Schwellwert verwendet.
Für die Segmentierung mittels A2-H wird der Bildausschnitt in RGB verwendet und in den HSV-Farbraum konvertiert. Als Nächstes werden die HSV-Werte vom Handzentrum (siehe Abbildung 3.21b) in einem Array erfasst, der Größe nach sortiert und die oberen und unteren 10 % entfernt (Eliminierung von Ausreißern). Die minimalen und maximalen HSV-Werte werden als

[9] Einzige Ausnahme dafür ist A2-C, da die Maske als Kante detektiert werden würde.

Schwellwerte für die Segmentierung mit der OpenCV-Methode `inRange` [92] eingesetzt, womit das binäre Bild erzeugt wird.
A2-T verwendet einen einfachen Grauwert-Schwellwert zur Segmentierung und wurde mithilfe der `threshold`-Funktion von OpenCV umgesetzt [88]. Diese liefert als Ergebnis das binäre Bild mit der weißen Hand auf schwarzem Hintergrund. Der minimale Schwellwert wird vom Benutzer über die Einstellungen festgelegt und der maximale Schwellwert entspricht dem maximal möglichen Grauwert (255).

Nach der erfolgreichen Segmentierung besteht die Möglichkeit, das Ergebnis mittels der morphologischen Operation Dilatation bzw. Erosion zu verändern. Mithilfe von Dilatation wird die Handregion vergrößert, womit eventuelle Lücken geschlossen werden können. Durch Erosion wird die Handregion verkleinert, um beispielsweise den Abstand zwischen den Fingern für die Fingerspitzenerkennung zu vergrößern. Die verwendete Kernelgröße kann vom Benutzer über die Einstellungen verändert werden. Für die Umsetzung der beiden morphologischen Operationen wurde die OpenCV-Funktion `morphologyEx` [87] verwendet.
Die Schritte für die Segmentierung mittels A2 werden in Listing 5.7 in Pseudocode beschrieben.

```
1  //Herausfiltern von Bildrauschen
2  hand = FILTER(hand)
3
4  //A2-C
5  WENN Einstellung = A2-C DANN
6    mw = MITTELWERT(hand)
7    stdabw = STANDARDABWEICHUNG(hand)
8    hand = canny(hand, mw - stdabw, mw + stdabw)
9  WENN_ENDE
10
11 //Hinzufuegen einer schwarzen elliptischen Maske
12 maske = ELLIPTISCHE_MASKE(handregion, hoehe, breite, schwarz)
13 hand = hand - maske
14
15 //A2-H
16 WENN Einstellung = A2-H DANN
17   //Erstellen des Auschnitts vom Handzentrum in HSV
18   handausschnitt = AUSSCHNITT(hand, handzentrum)
19   handausschnitt_hsv = RGB_ZU_HSV(handausschnitt)
20
21   //Sortieren und Entfernen der unteren und oberen 10 % der HSV-Werte
22   hsv_werte = SORTIEREN(handausschnitt_hsv)
23   hsv_werte = hsv_werte.IM_BEREICH(von 10 % bis 90 %)
24
25   hand = inRange(hand, MIN(hsv_werte), MAX(hsv_werte))
26 WENN_ENDE
27
28 //A2-T
29 WENN Einstellung = A2-T DANN
30   hand = threshold(hand, Einstellungen.Threshold_Min, 255)
31 WENN_ENDE
32
33 //Dilatation oder Erosion
34 WENN Einstellung = Dilatation DANN
```

```
35   hand = dilatation(hand, kernel)
36 SONST
37   hand = erosion(hand, kernel)
38 WENN_ENDE
```

Listing 5.7: Segmentierung der Hand mittels A2 in Pseudocode

5.5.2.3 Kontur der Hand ermitteln

Das zuvor ermittelte binäre Bild der Hand wird in der `processing`-Methode zur Ermittlung der Handkontur verwendet. Mithilfe der OpenCV-Funktion `findContours` [93] wird das Bild auf Konturen durchsucht. Eine Kontur wird durch mehrere 2D-Punkte beschrieben und in einem Vektor zusammengefasst. Als nächstes werden die gefundenen Konturen auf die Größe ihrer Konturfläche untersucht. Zu kleine Konturen werden, abhängig von der in den Einstellungen festgelegten minimalen Größe, nicht weiter berücksichtigt. Die restlichen Konturen werden in einer Kontur zusammengefasst. Diese Kontur wird für die Bestimmung der konvexen Hülle (Vektor aus 2D-Punkten) mittels `convexHull` [93] verwendet. Listing 5.8 beschreibt die Vorgehensweise in Pseudocode:

```
1  //Ermitteln der Konturen im binaeren Bild
2  konturen = findContour(hand)
3
4  //Entfernen der zu kleinen Konturen
5  konturen = ENTFERNE(alle Konturen < Einstellungen.Area_Size)
6
7  //Zusammenfassen aller Konturen in einer Kontur
8  kontur = ZUSAMMENFASSEN(konturen)
9
10 //Konvexe Huelle
11 konvexe_huelle = convexHull(kontur)
```

Listing 5.8: Ermittlung der Kontur der Hand mittels A2 in Pseudocode

5.5.2.4 Fingerspitzenerkennung

Die im vorangegangen Schritt ermittelte Kontur und konvexe Hülle der Hand wird nun in der `processing`-Methode für die Fingerspitzenerkennung eingesetzt. Dafür werden die convexity defects (Abschnitt 3.4.2) der Hand verwendet. Diese Punkte werden mit der OpenCV-Funktion `convexityDefects` [93] ermittelt und in einem Vektor gespeichert. Der convexity defect mit dem größten x-Wert im RGB-Koordinatensystem (siehe Abbildung 3.29a) wird als Daumen angenommen[10], unabhängig vom Abstand zur Handmitte. Zur Bestimmung der restlichen Fingerspitzen werden die convexity defects oberhalb der Handmitte herangezogen. Diesen Punkten wird der Reihe nach eine Fingerspitze zugewiesen, bis die maximale Fingeranzahl (vier) erreicht wurde oder keine geeigneten Punkte mehr zur Verfügung stehen. Die ermittelten Fingerspitzen werden in der Fingerliste von `processFrame` gespeichert. In Listing 5.9 wird die Fingerspitzenerkennung in Pseudocode beschrieben:

[10] Dabei handelt es sich um den Daumen der rechten Hand, da das Bild der Frontkamera von Android gespiegelt wird.

```
1 //Ermitteln der convexity defects
2 convexity_defects = convexityDefects(kontur, konvexe_huelle)
3
4 //Bestimmung der Daumenspitze
5 //Der erste gefundene convexity defect liegt am weitesten rechts
6 fingerliste.HINZUFUEGEN(convexity_defects.ERSTER_PUNKT())
7
8 //Bestimmung der restlichen Fingerspitzen
9 kandidat = convexity_defects.NAECHSTER_PUNKT()
10 SOLANGE fingerliste.anzahl() < 5 und kandidat != 0  WIEDERHOLE
11   WENN kandidat sich oberhalb der Handmitte befindet DANN
12     fingerliste.HINZUFUEGEN(kandidat)
13   WENN_ENDE
14   kandidat = convexity_defects.NAECHSTER_PUNKT()
15 WIEDERHOLE_ENDE
```

Listing 5.9: Fingerspitzenerkennung mittels A2 in Pseudocode

5.5.2.5 Relative Schätzung der 3D-Position

Nach der Ermittlung der Fingerspitzen wird in `processFrame` die 3D-Position der Hand relativ geschätzt. Dafür wird die vom Benutzer eingestellte Methode (Handgröße oder maximaler Grauwert der Hand) verwendet. Die Implementierung der Schätzung wird in Abschnitt 5.5.3 im Detail erläutert.

5.5.2.6 Aufruf der Interaktion

Im letzten Schritt wird die Anzahl der ermittelten Fingerspitzen zur Bestimmung der Fingergeste verwendet. Vier bis fünf erkannte Fingerspitzen entsprechen *Geste 1* und ein bis zwei Fingerspitzen *Geste 2*. Den Fingergesten entsprechend wird als Nächstes die zugehörige Methode in Unity aufgerufen und die passende Interaktion durchgeführt. In Abschnitt 5.5.5 wird die Umsetzung des Aufrufs der Interaktion näher beschrieben.

5.5.3 Relative Schätzung der 3D-Position

Die in Abschnitt 3.6.1 theoretisch beschriebene relative Schätzung der 3D-Position wird in der `processFrame`-Methode der Klasse `DetectionView` umgesetzt. Für die Schätzung wird die Handgröße oder der maximale Grauwert der Hand verwendet. Über die Benutzeroberfläche kann in den Einstellungen die gewünschte Schätzungsmethode ausgewählt werden (siehe Abschnitt 5.3.1). Durch die zuvor bestimmte Handregion mittels A1 oder A2 ist die Handgröße bekannt. Zur Schätzung der 3D-Position wird die Handbreite stellvertretend für die Größe verwendet. Wurde in den Einstellungen die Schätzung der 3D-Position mittels Grauwert ausgewählt, wird zur Minimierung des Hintergrunds eine schwarze elliptische Maske über den Bildausschnitt gelegt. Vom restlichen Ausschnitt wird mithilfe von OpenCV's `minMaxLoc` [92] der maximale Grauwert berechnet. In Listing 5.10 werden diese Schritte in Pseudocode verdeutlicht:

```
1 WENN Einstellung = Handgroesse DANN
2   //Handgroesse zur Schaetzung der 3D-Position
```

```
  handgroesse = handregion.breite
  3d_position = 3D_PUNKT(handregion.x, handregion.y, handgroesse)
SONST
  //Maximaler Grauwert zur Schaetzung der 3D-Position
  maske = ELLIPTISCHE_MASKE(handregion, hoehe, breite, schwarz)
  maximaler_grauwert = MAX_GRAUWERT(hand - maske)
  3d_position = 3D_PUNKT(handregion.x, handregion.y, maximaler_grauwert)
WENN_ENDE
```

Listing 5.10: Relative Schätzung der 3D-Position in Pseudocode

5.5.4 A1-D und A2-D (RGBD)

A1-D und A2-D erweitern die zuvor beschriebenen Ansätze A1 und A2. Die Tiefendaten werden zur absoluten Schätzung der 3D-Position verwendet, A2-D nutzt die RGBD-Daten zusätzlich zur robusten Segmentierung der Handfläche.

5.5.4.1 A1-D

Die Ermittlung der Fingergeste mittels A1-D entspricht der in Abschnitt 5.5.1 beschriebenen Implementierung von A1 und wird hier nicht näher ausgeführt. Anstatt der relativen Tiefenschätzung wird allerdings die Entfernung der Hand für die absolute Schätzung der 3D-Position verwendet. Die Ermittlung dieser Entfernung wird mithilfe der Tiefendaten beim Mapping durchgeführt und in Abschnitt 5.4.3 genauer beschrieben.

5.5.4.2 A2-D

In A2-D wird die Implementierung von A2 für die Detektion der Hand, Fingerspitzenerkennung und den Interaktionsaufruf verwendet (siehe Abschnitt 5.5.2). Für die Segmentierung der Handfläche werden jedoch die RGBD-Daten eingesetzt. Mithilfe der Tiefenwerte der einzelnen Pixel und der beim Mapping in Abschnitt 5.4.3 ermittelten Entfernung der Hand, werden die Pixel außerhalb der Handebene herausgefiltert. Außerdem wird die Entfernung der Hand für die absolute Schätzung der 3D-Position verwendet.

5.5.5 Aufruf der Interaktion

Der Aufruf der Interaktion wird in der Methode `processFrame` implementiert. Je nachdem welche Geste ermittelt wurde, wird die zugehörige Methode der VR-Szene aufgerufen und die Interaktion durchgeführt (siehe Abschnitt 5.7). Mithilfe von UnitySendMessage [120] wird das entsprechende Game Object und dessen Methode der Skript Component aufgerufen und die 3D-Position der Hand übergeben (siehe Listing 5.3). Außerdem werden mit einem zusätzlichen Parameter die aktuellen Einstellungen übergeben (Methode zur Tiefenschätzung, Grauwertbereich). Der Aufruf der Interaktion wird in Listing 5.11 mittels Pseudocode beschrieben:

```
1 //Interaktion mittels Geste 1
2 WENN Geste 1 DANN
3   SELEKTION(3D-Position der Hand, Einstellungen)
4 WENN_ENDE
5
6 //Interaktion mittels Geste 2
7 WENN Geste 2 DANN
8   MANIPULATION(3D-Position der Hand, Einstellungen)
9 WENN_ENDE
```

Listing 5.11: Aufruf der Interaktionslogik in Unity in Pseudocode

5.6 Ermittlung der Kopfposition

Die Ermittlung der Kopfposition wird in der Klasse `DetectionView` implementiert und verwendet mithilfe von JNI die C++-Klasse `DetectionBasedTracker` für die Erkennung des Kopfs (siehe Abbildung 5.9). Dafür und zur relativen Schätzung der 3D-Position werden die RGB-Daten der Kamera verwendet. Die 3D-Position des Kopfs wird an die VR-Szene übermittelt und dort zur Steuerung der virtuellen Kamera eingesetzt.

Für die Detektion des Kopfs wird ein von OpenCV zur Verfügung gestellter Haar-Klassifikator verwendet [90]. Damit wird beim Start der Anwendung in der Klasse `DetectionView` das `DetectionBasedTracker`-Objekt für die Detektion initialisiert. Die Kopfdetektion wird in der `processFrame`-Methode von `DetectionView` mit dem Graustufenbild der Kamera durchgeführt. Dazu wird die `detect`-Methode des `DetectionBasedTracker`-Objekts ausgeführt. Von den Ergebnissen der Detektion wird das erste gefundene Objekt als Kopf angenommen. Der Kopf wird als ein Rechteck-Objekt repräsentiert, womit die 2D-Position und die Größe des Kopfs zur Verfügung steht. Die Kopfgröße wird zur relativen Schätzung der 3D-Position verwendet. Als letzter Schritt wird mittels UnitySendMessage [120] die Kopfinteraktion in Unity aufgerufen und die 3D-Position des Kopfs übermittelt. In Listing 5.12 werden die beschriebenen Schritte in Pseudocode verdeutlicht:

```
1  //Suche Kopf
2  gefundene_objekte = Kopf_Klassifikator.DETEKTION(graustufenbild)
3
4  //Das erste gefundenen Objekt wird als Kopf angenommen
5  kopfregion = gefundene_objekte.ERSTES_OBJEKT()
6
7  //Groesse des Kopfs zur relativen Schaetzung der 3D-Position
8  kopfgroesse = kopfregion.breite
9
10 3d_position = 3D_PUNKT(kopfregion.x, kopfregion.y, kopfgroesse)
11
12 //Aufruf der Interaktion mittels Kopfposition
13 KAMERASTEUERUNG(3d_position)
```

Listing 5.12: Ermittlung der Kopfposition in Pseudocode

5.7 Darstellung und Interaktion - VR-Szene

In diesem Abschnitt wird die Umsetzung der VR-Szene und Implementierung der Interaktionslogik im Detail beschrieben. Die wichtigsten Game Objects der VR-Szene werden in Abbildung 5.10 dargestellt. In Abschnitt 5.7.1 werden diese Game Objects und ihre Components näher beschrieben. Die Skripten für die Umsetzung der Interaktionslogik werden in Abschnitt 5.7.2 bzw. 5.7.3 im Detail behandelt.

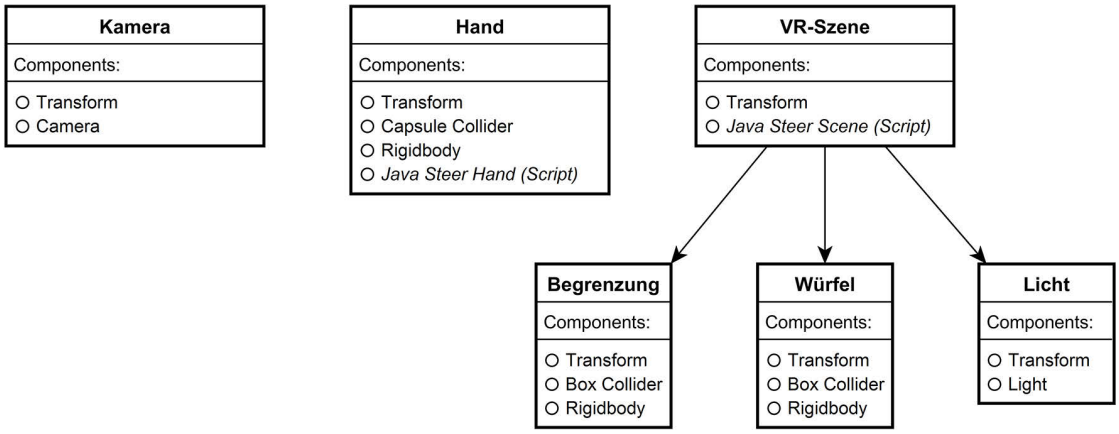


Abbildung 5.10: Diagramm der wichtigsten Game Objects (mit Components) der VR-Szene

5.7.1 Game Objects der VR-Szene

In Abbildung 5.10 werden die wichtigsten Game Objects und ihre Components dargestellt mit denen die VR-Szene in Unity3D [122] aufgebaut wurde. Die Eigenschaften und Funktionsweise von Game Objects und den einzelnen Components wird in Abschnitt 5.2.2 genauer erläutert. Nachfolgend werden die Aufgaben und Funktion der Objekte in dieser Arbeit kurz beschrieben.

5.7.1.1 Kamera

Das Game Object `Kamera` wird für die Umsetzung der virtuellen Kamera verwendet. Diese repräsentiert die Position und Blickrichtung des Betrachters auf die VR-Szene und bestimmt den sichtbaren Szenenausschnitt, der am Tablet dargestellt wird. Die virtuelle Kamera wird mithilfe der `Camera` Component von Unity implementiert und mittels der `Transform` Component an der gewünschten Stelle positioniert.

5.7.1.2 Hand

Mithilfe des Game Object `Hand` wird die virtuelle Hand implementiert (siehe Abschnitt 3.7.2). Sie wird in dieser Arbeit von einer roten Kapsel mit den Components `Rigidbody` und `Transform` repräsentiert. Außerdem besitzt sie eine `Capsule Collider`[11] Component

[11] Mit aktivierter `Is Trigger`-Option, um das `OnTriggerEnter`-Ereignis bei einer Kollision auszulösen.

für die Kollisionsabfrage. Für die Implementierung der Interaktionslogik wurde das Skript `JavaSteerHand` hinzugefügt (siehe Abschnitt 5.7.2).

5.7.1.3 VR-Szene

Mit dem Game Object `VR-Szene` werden die Objekte der Szene zusammengefasst. Dadurch ist es möglich, die Position und Orientierung der gesamten VR-Szene zu verändern. Dies wird für die Implementierung der Kopfinteraktion benötigt, wofür dem Objekt die Skript Component `JavaSteerScene` hinzugefügt wurde (Abschnitt 5.7.3). Die Szene setzt sich wiederum aus den Game Objects `Begrenzung`, `Würfel` und `Licht` zusammen. Im Objekt `Begrenzung` werden die Wände und der Boden, mit dem der 3D-Raum der Szene abgegrenzt wird, zusammengefasst. Zur Selektion mit der virtuellen Hand wurden der Szene vier Würfel-Objekte hinzugefügt, welche im Objekt `Würfel` zusammengefasst werden. Als Components wurde je ein `Box-Collider` für die Kollisionsabfrage und ein `Rigidbody` zur Aktivierung der physikalischen Eigenschaften hinzugefügt. Die Beleuchtung der Szene wird im `Licht`-Objekt mittels einer `Light` Component umgesetzt.

5.7.2 Interaktion mittels Fingergesten

In Abschnitt 3.7.2 wird die Interaktion mittels Fingergesten theoretisch beschrieben. Die Interaktion wird im Skript `JavaSteerHand` implementiert. Das Skript wurde dem Game Object `Hand` als Component hinzugefügt und besteht aus der Methode `JavaInit` für die Initialisierung und der Methode `JavaMoveHand` bzw. `JavaMoveObject`, um die virtuelle Hand bzw. das selektierte Objekt zu bewegen. Außerdem wird mit der Methode `Transform3D` die Transformation der 3D-Position der realen Hand in die 3D-Position der virtuellen Hand durchgeführt und mit der `OnTriggerEnter`-Methode des Skripts die Selektion von einem Objekt umgesetzt.

5.7.2.1 Initialisierung

Die Initialisierung (`JavaInit`) wird in Java von der Klasse `SampleCvViewBase` aufgerufen (Abschnitt 5.3.2). Damit wird die Auflösung des Kamerabilds an das Skript `JavaSteerHand` übermittelt. Die Berechnung der Transformations- und Skalierungskonstanten aus den Gleichungen 3.11 bis 3.13 wird mithilfe der Kameraauflösung, dem Grauwertbereich und dem Bereich der Handbewegung durchgeführt und in Listing 5.13 mit Pseudocode beschrieben:

```
1  //Translationskonstanten:
2  //0.5 fuer die gleichmaessige Aufteilung der Bewegung in x und y
3  T_xh = kamerabild.breite * 0.5
4  T_yh = kamerabild.hoehe * 0.5
5
6  //Erfahrungswert der minimal erkennbaren Hand (40 % des Kamerabilds)
7  T_zh_handgroesse = kamerabild.breite * 0.4
8  //Standardeinstellung fuer den minimalen Grauwert
9  T_zh_grau = 230
10 //Entspricht einem minimalen Abstand von 75 cm
11 T_zh_kinect = 750
```

```
12 //Skalierungskonstanten:
13 //Erfahrungswerte um die maximale Verschiebung in x bzw. y zu erreichen
14 S_xh = -0.05
15 S_yh = -0.05
16
17 //Berechnet mit dem maximal moeglichen z-Wert der Hand von 12
18 //Die Handbewegung wird im Bereich 40-70 % des Kamerabilds aufgeteilt
19 S_zh_handgroesse = 12 / (kamerabild.breite * 0.7 - kamerabild.breite * 0.4)
20 //Die Handbewegung findet zu Beginn im Grauwertbereich von 230 bis 255 statt
21 S_zh_grau = 12 / (255 - 230)
22 //Erfahrungswert zur Umrechnung der Tiefendaten in die VR-Szene
23 S_zh_kinect = -1/20
```

Listing 5.13: Berechnung der Konstanten zur Transformation der Handposition in Pseudocode

5.7.2.2 Interaktionslogik

Mit den aktuellen Einstellungen (Grauwertbereich und 3D-Methode[12]), sowie der 3D-Position der Hand wird in der Java-Klasse `DetectionView` abhängig von der ermittelten Fingergeste die Interaktion im Skript `JavaSteerHand` aufgerufen (siehe Abschnitt 5.5.5). Für *Geste 1* wird die Methode `JavaMoveHand` aufgerufen, um die virtuelle Hand zu bewegen und Objekte zu selektieren. Dafür wird die 3D-Position mithilfe der Methode `Transform3D` (Listing 5.16) in das VR-Koordinatensystem transformiert und die Position des Game Object `Hand` verändert. Die Selektion eines Objekts wird mittels Kollisionsabfrage realisiert und im nächsten Abschnitt genauer beschrieben (siehe Listing 5.17). Die Umsetzung von `JavaMoveHand` wird in Listing 5.14 in Pseudocode erläutert:

```
1 //Transformation der 3D-Handposition in die virtuelle Szene
2 3d_hand_vr = Transform3D(3d_hand, Einstellungen)
3
4 //Positionsveraenderung
5 Game_Object.Hand.position = 3D_PUNKT(3d_hand_vr)
```

Listing 5.14: Implementierung der Interaktion mittels *Geste 1* in Pseudocode

Mit der *Geste 2* wird das selektierte Objekt bewegt und dafür die Methode `JavaMoveObjekt` des Skripts aufgerufen. Die 3D-Position der Hand wird mittels der `Transform3D`-Methode (Listing 5.16) in das VR-Koordinatensystem transformiert und damit die Position des selektierten Würfel-Objekts verändert. Das selektierte Würfel-Objekt wird mittels der Kollisionsabfrage im nächsten Abschnitt bestimmt (Listing 5.17). In Listing 5.15 wird die Implementierung von `JavaMoveObject` in Pseudocode beschrieben:

```
1 //Transformation der 3D-Handposition in die virtuelle Szene
2 3d_hand_vr = Transform3D(3d_hand, Einstellungen)
3
4 //Positionsveraenderung
5 selektiertes_game_object.position = 3D_PUNKT(3d_hand_vr)
```

Listing 5.15: Implementierung der Interaktion mittels *Geste 2* in Pseudocode

[12] Verwendete Methode zur Tiefenschätzung: Handgröße, Grauwert oder Tiefendaten

In beiden Methoden werden mithilfe von `Transform3D` die Konstanten an den aktuellen Grauwertbereich angepasst und die 3D-Position der Hand in das VR-Koordinatensystem transformiert. Die Umsetzung dieser Transformation wird in Listing 5.16 mittels Pseudocode erläutert:

```
1  //Transformation der Handposition in die virtuelle Szene
2  3d_hand_vr.x = (3d_hand.x - T_xh) * S_xh
3  3d_hand_vr.y = (3d_hand.y - T_yh) * S_yh
4
5  //Je nach 3D-Methode wird die Transformation der z-Position durchgefuehrt
6  WENN Einstellungen.3D-Methode = Handgroesse DANN
7    3d_hand_vr.z = (3d_hand.z - T_zh_handgroesse) * S_zh_handgroesse
8  WENN_ENDE
9
10 WENN Einstellungen.3D-Methode = Grauwert DANN
11   //Anpassung der Konstanten an den aktuellen Grauwertbereich
12   //Der vom Benutzer eingestellte minimale Grauwert
13   T_zh_grau = Einstellungen.min_grau
14   //Die Handbewegung findet im eingestellten Grauwertbereich statt
15   S_zh_grau = 12 / (Einstellungen.max_grau - Einstellungen.min_grau)
16
17   3d_hand_vr.z = (3d_hand.z - T_zh_grau) * S_zh_grau
18 WENN_ENDE
19
20 WENN Einstellungen.3D-Methode = Kinect DANN
21   3d_hand_vr.z = (3d_hand.z - T_zh_kinect) * S_zh_kinect
22 WENN_ENDE
```

Listing 5.16: Implementierung der Transformierung der Handposition in Pseudocode

5.7.2.3 Kollisionsabfrage

Kollidiert die virtuelle Hand mit einem Game Object das eine `Collider` Component besitzt wird das Ereignis `OnTriggerEnter` ausgelöst und die gleichnamige Methode im Skript aufgerufen. Handelt es sich bei dem Game Object um einen Würfel, wird dieser selektiert und rot eingefärbt. Berührt die Hand einen anderen Würfel oder die Begrenzung der VR-Szene wird die aktuelle Selektion aufgehoben. Handelt es sich beim berührten Objekt zudem um einen Würfel wird dieser neu selektiert. Das aktuelle selektierte Objekt wird den anderen Methoden als Game Object zur Verfügung gestellt. Die Implementierung der Kollisionsabfrage wird in Listing 5.17 in Pseudocode näher beschrieben:

```
1  //Ueberpruefen ob das kollidierte Objekt nicht das bereits selektierte ist
2  WENN kollidiertes_game_object != selektiertes_game_object DANN
3
4    //Kollision mit einem Wuerfel
5    WENN kollidiertes_game_object = Game_Object.Wuerfel DANN
6
7      //Aufheben der Selektion des alten Objekts
8      WENN selektiertes_game_object != 0 DANN
9        selektiertes_game_object.farbe = original_farbe
10     WENN_ENDE
11
```

```
12      //Selektion des neuen Objekts
13      selektiertes_game_object = kollidiertes_game_object
14      original_farbe = kollidiertes_game_object.farbe
15      selektiertes_game_object.farbe = Farbe.ROT
16
17    //Kollision mit der Begrenzung
18    SONST
19      //Aufheben der Selektion
20      selektiertes_game_object.farbe = original_farbe
21      selektiertes_game_object = 0
22    WENN_ENDE
23
24  WENN_ENDE
```

Listing 5.17: Implementierung der Kollisionsabfrage in Pseudocode

5.7.3 Interaktion mittels Kopfposition

Die Interaktion mit der VR-Szene durch die Bewegung des Kopfs ist theoretisch in Abschnitt 3.7.2 beschrieben. Für die Implementierung wird das Skript `JavaSteerScene` verwendet, welches dem Game Object `VR-Szene` als Component hinzugefügt wurde. Das Skript beinhaltet die Methode `JavaInit` zur Initialisierung und `JavaSetPosition`, um die Position und Orientierung der VR-Szene zu verändern. Für die Veränderung der virtuellen Kamera wird nicht die Kamera selbst verschoben, sondern die gesamte VR-Szene. Dadurch wird sichergestellt, dass die Szene um den Szenemittelpunkt rotiert wird. Außerdem bleibt die virtuelle Hand damit in einer Linie zum Betrachter (Kamera), sodass die reale Handposition mit der virtuellen Handposition übereinstimmt[13].

5.7.3.1 Initialisierung

Zur Initialisierung wird die Methode `JavaInit` in Java von der Klasse `SampleCvViewBase` aufgerufen und die Auflösung des Kamerabilds übermittelt (siehe Abschnitt 5.3.2). Für die Berechnung der Transformations- und Skalierungskonstanten aus den Gleichungen 3.14 bis 3.18 werden Kameraauflösung, maximaler Rotationswinkel und maximale 3D-Position eingesetzt. Mit den berechneten Konstanten wird die Transformation der Kopfposition in der Methode `JavaSetPosition` durchgeführt. Die Konstantenberechnung wird in Listing 5.18 in Pseudocode beschrieben:

```
1 //Translationskonstanten:
2 //0.66 damit die Ausgansposition einen Blick von oben auf die Szene ergibt
3 T_xk = kamerabild.hoehe * 0.66
4 //0.5 fuer die gleichmaessige Aufteilung der Rotation in y
5 T_yk = kamerabild.breite * 0.5
6
7 //Skalierungskonstanten zur Rotation:
8 //Berechnet mit den maximalen Rotationswinkeln von 48 bzw. -44 Grad
```

[13] Zum Beispiel wird somit eine reale Handbewegung nach vorne auch in der virtuellen Szene nach vorne dargestellt, da die Kamera und die virtuelle Hand in derselben Richtung liegen.

```
S_xk = 48 / kamerabild.hoehe
S_yk = -44 / kamerabild.breite
//Erfahrungswerte um die maximale Verschiebung in x bzw. y zu erreichen
S_xk2 = -1/15
S_yk2 = 1/15

//Skalierungskonstante zur Bewegung in z:
//Maximal erkennbare Kopfgroesse = 80 % des Kamerabilds
max_kopf = kamerabild.breite * 0.8
//Berechnet mit dem maximal moeglichen z-Wert der VR-Szene von -10
S_zk = -10/(max_kopf)
```

Listing 5.18: Berechnung der Konstanten zur Transformation der Kopfposition in Pseudocode

5.7.3.2 Interaktionslogik

Nachdem die 3D-Position des Kopfs in der Java-Klasse `DetectionView` ermittelt wurde, wird sie an die Methode `JavaSetPosition` des Skripts `JavaSteerScene` übermittelt (siehe Abschnitt 5.5.5). Diese wird zur Veränderung der Position und Orientierung der VR-Szene eingesetzt. Für die Implementierung der Interaktion wurden die in Abschnitt 3.7.2 beschriebenen Gleichungen 3.14 bis 3.18 verwendet. Kleine Kopfbewegungen werden ignoriert, um damit die Position und Orientierung der VR-Szene nicht zu verändern. Die Implementierung der Kopfinteraktion wird in Listing 5.19 in Pseudocode beschrieben:

```
//Transformation der Kopfposition in die virtuelle Szene
rotation_vr.x = (3d_kopf.x - T_xk) * S_xk
rotation_vr.y = (3d_kopf.y - T_yk) * S_yk

3d_vr.x = rotation.y * S_xk2
3d_vr.y = rotation.x * S_yk2
3d_vr.z = 3d_kopf.z * S_zk

WENN Veraenderung in z > 0.1 DANN
  Game_Object.VR-Szene.position = 3D_PUNKT(VR-Szene.x, VR-Szene.y, 3d_vr.z)
WENN_ENDE

WENN Rotation > 1 Grad DANN
  Game_Object.VR-Szene.rotation = EULER(rotation_vr.x, rotation_vr.y, 0)
  Game_Object.VR-Szene.position = 3D_PUNKT(3d_vr.x, 3d_vr.y, VR-Szene.z)
WENN_ENDE
```

Listing 5.19: Implementierung der Kopfinteraktion in Pseudocode

Teil III

Resultate

KAPITEL 6

Evaluierung

Es wurden zwei Studien zur Evaluierung des entwickelten Prototypen durchgeführt. In der ersten Studie wurden die für diese Arbeit erstellten Haar-Klassifikatoren und die vorgestellten Ansätze zur Fingergestenerkennung auf ihre Robustheit und Performance hin untersucht. Die Herangehensweise der Evaluierung und die Auswertung, sowie Diskussion der Ergebnisse werden in Abschnitt 6.1 beschrieben. Die natürliche 3D-Interaktion wurde mithilfe einer experimentellen Studie evaluiert. Der Aufbau und die Ergebnisse hierzu werden in Abschnitt 6.2 beschrieben.

In Kapitel 7 werden Empfehlungen für den Einsatz der untersuchten Methoden zur Fingergestenerkennung und Tiefenschätzung präsentiert, welche anhand der Ergebnisse dieser Evaluierung erarbeitet wurden.

6.1 Evaluierung der Fingergestenerkennung

Mithilfe des Prototypen wurden bis auf A1-D alle in Abschnitt 3.4 vorgestellten Ansätze zur Fingergestenerkennung evaluiert. A1-D wurde nicht gesondert evaluiert, da sich die Fingergestenerkennung nicht von A1 unterscheidet und die Tiefendaten nur für die Bestimmung der 3D-Position verwendet werden. Zusätzlich wurden die Haar-Klassifikatoren K1, K2 und K3, sowie die Ansätze zur Fingerspitzenerkennung A2-C, A2-H, A2-T und A2-D separat untersucht. Es wurden fünf verschiedene Testsets mit Fingergesten in variierenden Positionen und unterschiedlichen Hintergrund- und Beleuchtungssituationen erstellt und für die Evaluierung verwendet. Der Aufbau dieser Testsets und der Bezugsdaten wird in Abschnitt 6.1.1 beschrieben. In Abschnitt 6.1.2 werden die Ergebnisse präsentiert und ausgewertet. Die Diskussion der Resultate wird in Abschnitt 6.1.3 beschrieben.

6.1.1 Testsets und Bezugsdaten

Durch die fünf Testsets werden verschiedene, realistische Interaktionsszenarien simuliert. Jedes Set beinhaltet die gleiche Anzahl positiver wie negativer Bilder. Positive Bilder enthalten die zu

erkennende Fingergeste, im Unterschied zu negativen Bildern, in denen die Geste nicht zu sehen ist. In Tabelle 6.1 wird der Aufbau der einzelnen Testsets näher beschrieben.

Tabelle 6.1: Aufbau der Testsets zur Evaluierung der Fingergesten

Testset	Fingergeste	Beleuchtung	Hintergrund	Anzahl der Bilder
Set 1	*Geste 1*	gleichmäßig	weiß & ruhig	104
Set 2	*Geste 1*	wechselnd	weiß & ruhig	110
Set 3	*Geste 1*	gleichmäßig	unruhig	148
Set 4	*Geste 2*	wechselnd	weiß & ruhig	138
Set 5	*Geste 1 & 2*	wechselnd	weiß & ruhig	220

Für diese Evaluierung werden RGB- bzw. RGBD-Daten verwendet. Alle Testbilder der Sets werden mithilfe der Android-Applikation und der Frontkamera des Tablets aufgezeichnet. Außerdem werden mittels Kinect die Tiefendaten der Szene erfasst und gespeichert. Um robuste und eindeutig definierte Bezugsdaten zu erhalten, wird der Handmittelpunkt in allen Bildern manuell annotiert.

6.1.2 Resultate

Zur Evaluierung werden die Testbilder vom Tablet verarbeitet und die Ergebnisbilder gespeichert. Dabei werden die Sets getrennt mit den entsprechenden Klassifikatoren bzw. Ansätzen untersucht. Die Ergebnisbilder werden mit den Bezugsbildern verglichen, um die Detektion der Hand als richtig oder falsch einstufen zu können. Dafür wurde mittels Matlab der ermittelte Handmittelpunkt mit der Annotation der Bezugsdaten verglichen. Eine Detektion der Hand wird für richtig befunden, wenn der Abstand zwischen den beiden Punkten kleiner als 15 Pixel beträgt. Ist der Abstand größer, wird die Detektion als falsch deklariert. Als Nächstes wird die ermittelte Geste mit der gesuchten Geste verglichen und das Resultat bewertet. Zum Schluss werden die Ergebnisse in die folgenden vier Fälle unterteilt und deren Häufigkeit zur Berechnung der Kenngrößen für die Evaluierung verwendet:

Richtig Positiv: Die gesuchte Geste ist im Bild und wurde richtig erkannt

Falsch Negativ: Die gesuchte Geste ist im Bild, wurde allerdings nicht erkannt

Falsch Positiv: Die gesuchte Geste ist nicht im Bild, wird allerdings fälschlicherweise erkannt

Richtig Negativ: Die gesuchte Geste ist nicht im Bild und wird auch nicht erkannt

Anhand der erreichten Framerate während der Evaluierung wurde die Performance des Prototypen untersucht. Die Haar-Klassifikatoren und die Ansätze zur Fingergestenerkennung wurden mithilfe von *F-Score*[1] und *Accuracy*[2] evaluiert und werden nachfolgend detailliert beschrieben [28].

[1] F-Maß

[2] Korrektklassifikationsrate oder auch Vertrauenswahrscheinlichkeit

F-Score ist das kombinierte Maß von *Precision*[3] und *Recall*[4] als deren harmonisches Mittel (Gleichung 6.1).

$$\mathbf{F-Score} = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{6.1}$$

Die Precision (Gleichung 6.2) entspricht dem Verhältnis der korrekten Ergebnisbilder zu den als richtig eingestuften Bildern und Recall (Gleichung 6.3) dem Verhältnis zu den tatsächlich richtig klassifizierten Bildern.

$$\mathbf{Precision} = \frac{Richtig\ Positive}{Richtig\ Positive + Falsch\ Positive} \tag{6.2}$$

$$\mathbf{Recall} = \frac{Richtig\ Positive}{Richtig\ Positive + Falsch\ Negative} \tag{6.3}$$

Mithilfe der Accuracy wird der Anteil aller richtig erkannten Bilder angegeben. Sie entspricht der geschätzten Wahrscheinlichkeit in Gleichung 6.4.

$$\mathbf{Accuracy} = \frac{Richtig\ Positive + Richtig\ Negative}{Positive + Negative} \tag{6.4}$$

6.1.2.1 Performance

Die durchschnittliche Framerate aller Ansätze mit einer Standardabweichung σ wird in Tabelle 6.2 aufgelistet. Die beiden Ansätze A1 und A2-D benötigen komplexe Berechnungen, die ihre Verarbeitungszeit erhöhen. Dies führt zu einer signifikant geringeren Framerate im Vergleich zu den anderen Ansätzen.

Tabelle 6.2: Evaluierung der Performance

Mittelwert (σ)	A1	A2-C	A2-H	A2-T	A2-D
Framerate in fps	13.86 (1.91)	16.6 (1.76)	16.62 (1.88)	16.79 (1.79)	8.13 (1.51)

6.1.2.2 Haar-Klassifikatoren

Um die Robustheit der drei Haar-Klassifikatoren zu evaluieren, wurde der Mittelwert von F-Score und Accuracy für jeden Klassifikator über alle Testsets berechnet. Die Ergebnisse in Tabelle 6.3 zeigen vergleichbare Werte von K1 für die Detektion von *Geste 1* in A1 und K3 bei der Handerkennung für A2. Die Detektion von *Geste 2* mittels K2 in A1 ist im Vergleich mit den beiden anderen Klassifikatoren weniger präzise.

[3] Genauigkeit
[4] Trefferquote

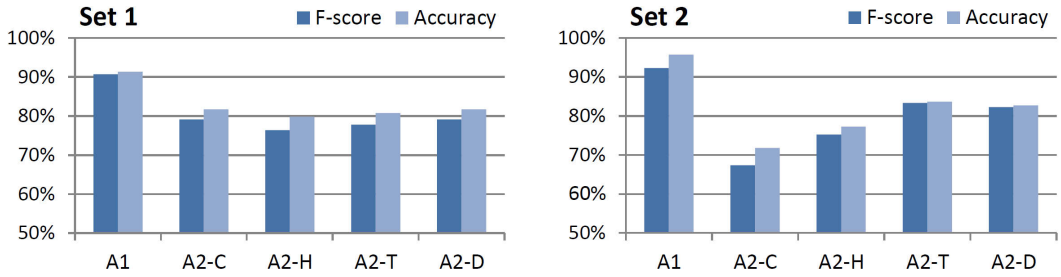
Tabelle 6.3: Evaluierung der Haar-Klassifikatoren

Mittelwert (σ)	K1	K2	K3
F-Score in %	86.03 (12.12)	60.61 (5.79)	82.22 (9.89)
Accuracy in %	90.13 (7.06)	72.65 (13.22)	81.26 (8.89)

6.1.2.3 Fingergestenerkennung

Für die Untersuchung der Fingergestenerkennung wurden alle Testsets mit jedem Ansatz und unabhängig voneinander verarbeitet. Anschließend wurden die Ergebnisbilder mit den Bezugsdaten verglichen und bewertet. Die Fingergeste gilt als richtig erkannt, wenn die Handposition mit der Annotation übereinstimmt (+/- 15 Pixel) und die richtige Geste erkannt wurde.

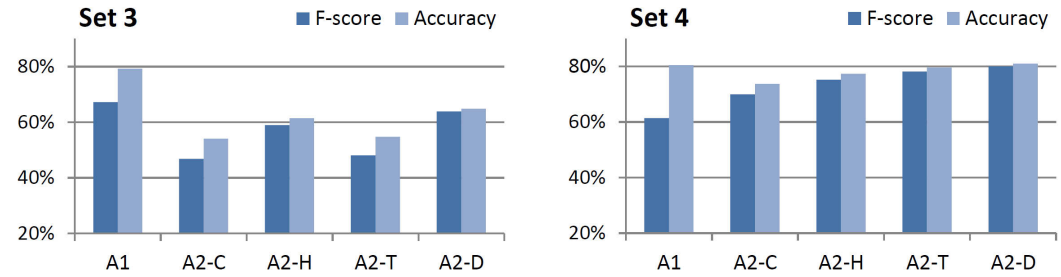
Wie in Abbildung 6.1 dargestellt, wird die *Geste 1* vor weißem Hintergrund und gleichmäßigem Licht mit allen Ansätzen gut erkannt (Abbildung 6.1a). Allerdings weisen bei wechselnden Lichtverhältnissen A2-C und A2-H schlechtere Ergebnisse auf (siehe Abbildung 6.1b).



(a) Set 1 (b) Set 2

Abbildung 6.1: Evaluierung der Fingergeste mit Set 1 und Set 2

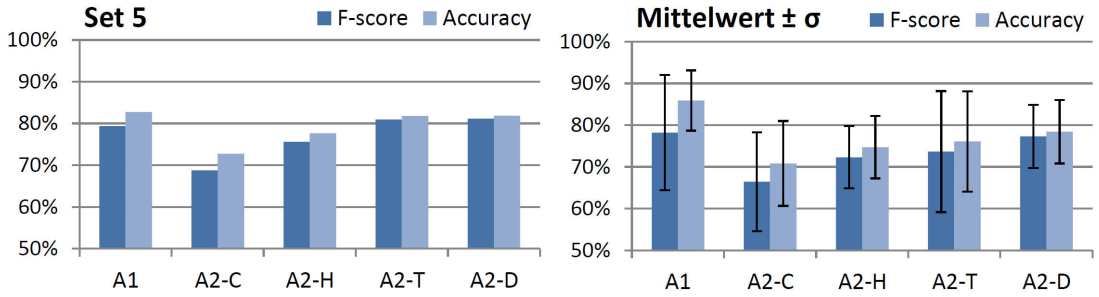
Die Evaluierung von *Geste 1* vor unruhigem Hintergrund in Abbildung 6.2a offenbart schlechte Ergebnisse für A2-C und A2-T. Im Vergleich zu Abbildung 6.1b, sind die Resultate für die *Geste 2* von A2-C, A2-H, A2-T, A2-D nahezu identisch und zeigen eine große Abweichung bei A1, wie in Abbildung 6.2b dargestellt.



(a) Set 3 (b) Set 4

Abbildung 6.2: Evaluierung der Fingergeste mit Set 3 und Set 4

Nach der getrennten Untersuchung von *Geste 1* und *2* wurden mittels Set 5 beide Fingergesten getestet. Wie in Abbildung 6.3a dargestellt, sind die Resultate von A1, A2-T und A2-D besser als von A2-H und besonders A2-C. Zusammenfassend wurde über alle Testsets der Mittelwert von F-Score und Accuracy berechnet (siehe Abbildung 6.3b). Die besten Ergebnisse werden mit A1 und A2-D erzielt, wohingegen A2-C am schlechtesten abschneidet.



(a) Set 5 (b) Mittelwert aller Testsets

Abbildung 6.3: Evaluierung der Fingergeste mit Set 5 und der Mittelwert aller Testsets

6.1.2.4 Fingerspitzenerkennung

Um die Performance der Fingergestenerkennung mittels A2 besser veranschaulichen zu können, wurden die Ergebnisbilder der zuvor beschriebenen Evaluierung der Fingergestenerkennung neu bewertet. Dafür wurden die Ergebnisse mit einer korrekten Handdetektion verwendet, um nur die Ermittlung der Geste mittels Fingerspitzenerkennung zu untersuchen und den Fehler des Klassifikators auszunehmen. Stimmt die erkannte Fingergeste mit der gesuchten Geste überein, wurde das Bild als korrekt eingestuft.

In Tabelle 6.4 werden die Ergebnisse der Evaluierung der Fingerspitzenerkennung mittels A2 zusammengefasst. A2-H, A2-T und vor allem A2-D erreichen insgesamt bessere Resultate als A2-C bei der Bestimmung der Fingergeste, sofern die Handfläche zuvor korrekt detektiert wurde. Bei unruhigem Hintergrund (Set 3) schneidet A2-C und A2-T schlechter ab als A2-H und A2-D. Im Vergleich zur Evaluierung der Fingergestenerkennung verdeutlichen diese Ergebnisse die Auswirkungen der Handdetektion mittels Haar-Klassifikator auf die Gesamtperformance der Ansätze.

Tabelle 6.4: Evaluierung der Fingerspitzenerkennung

Testset	Evaluierung	A2-C	A2-H	A2-T	A2-D
Set 1	F-Score in %	88,89	86,08	88,61	88,89
	Accuracy in %	80,00	75,56	79,55	80,00
Set 2	F-Score in %	79,01	88,37	97,83	94,62
	Accuracy in %	65,31	79,17	95,74	89,80
Set 3	F-Score in %	78,95	94,25	79,49	94,85
	Accuracy in %	65,22	89,13	65,96	90,20
Set 4	F-Score in %	90,32	94,95	98,04	100,00
	Accuracy in %	82,35	90,38	96,15	100,00
Set 5	F-Score in %	84,62	91,57	97,70	97,18
	Accuracy in %	73,33	84,44	95,51	94,51
Mittelwert (σ)	F-Score in %	84,36 (6,16)	91,04 (4,37)	92,33 (8,84)	95,11 (4,54)
	Accuracy in %	73,24 (9,24)	83,74 (7,33)	86,58 (14,5)	90,90 (8,17)

6.1.3 Diskussion

Die Resultate der Evaluierung der Performance lassen erkennen, dass mit allen Ansätzen eine interaktive Framerate erreicht und damit eine direkte 3D-Interaktion ermöglicht wird. Allerdings weisen A1 und A2-D eine höhere Latenz auf, woraus eine geringere Framerate resultiert. Bei A1 wird dies durch die Verwendung von zwei rechenintensiven Klassifikatoren zur Detektion der Fingergeste verursacht. Das Einlesen der Tiefendaten und das Mapping sind bei A2-D dafür verantwortlich.
Die Haar-Klassifikatoren K1 und K3 liefern selbst bei wechselnden Lichtverhältnissen und unruhigem Hintergrund mit einem F-Score Mittelwert von 86,03 % bzw. 82,22 % und einem Accuracy Mittelwert von 90,13 % bzw. 81,26 % gute Ergebnisse. Die Resultate von K2 erreichen im Vergleich dazu mit 60,61 % einen geringeren F-Score, sowie mit 72,65 % eine geringere Accuracy. Die Ursache dafür liegt in der Form eines ausgestreckten einzelnen Fingers. Dieser besteht aus einer kleinen Fläche und bietet dadurch weniger Merkmale für die Erkennung und die Unterscheidung vom Hintergrund. Durch zusätzliches Training des Klassifikators kann das Problem verringert werden.

Bei der Analyse der Ergebnisse für die Fingergestenerkennung wurde festgestellt, dass A1 bei unruhigem Hintergrund mit einem F-Score von 67,26 % und einer Accuracy von 79,21 % am besten abschneidet und für *Geste 1* in allen Situationen gut funktioniert. Im Vergleich zu *Geste 1* (F-Score: 92,31 %, Accuracy: 95,70 %), erreicht *Geste 2* bei wechselnder Beleuchtung und weißem Hintergrund mit einem F-Score von 61,39 % und einer Accuracy von 80,50 % schlechtere Ergebnisse. Diese Unterschiede für *Geste 2* reflektieren die Resultate der Klassifikator-Evaluierung von K2. A2-C erzielte bei unruhigem Hintergrund mit einem F-Score von 46,88 % und einer Accuracy von 54,05 % schlechte Ergebnisse und schnitt in dieser Evaluierung mit ei-

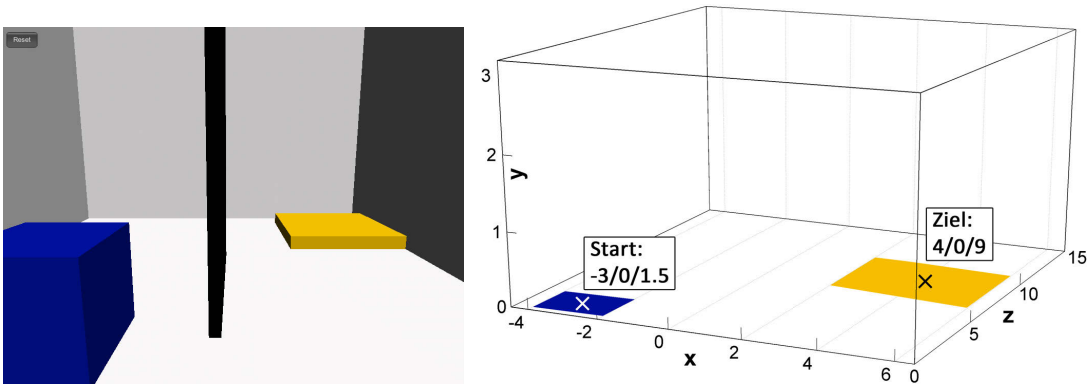
nem F-Score Mittelwert von 66,42 % und einem Accuracy Mittelwert von 70,81 % am schlechtesten ab. Da die Canny-Kantendetektion nicht zwischen Handkontur oder anderen Kanten unterscheiden kann, ist dieser Ansatz nicht empfehlenswert für mobile NUIs. A2-H erreichte in dieser Evaluierung gute Ergebnisse, und ist A2-C und A2-T vor allem bei unruhigem Hintergrund mit einem F-Score von 58,99 % und einer Accuracy von 61,49 % überlegen. Um auf wechselndes Umgebungslicht reagieren zu können, bietet A2-T dem Benutzer die Möglichkeit den Schwellwert manuell anzupassen. In diesen Situationen zeigte A2-T gute Resultate und schnitt besser als A2-C und A2-H ab. Bei unruhigem Hintergrund erzielte der Ansatz allerdings schlechte Ergebnisse (F-Score: 48,06 %, Accuracy: 54,73 %). In A2-D werden die Tiefendaten verwendet, um den Hintergrund zu entfernen. Dies resultiert in guten Ergebnissen bei unruhigem Hintergrund, Lichtwechseln und mit einem F-Score Mittelwert von 77,28 % und einem Accuracy Mittelwert von 78,43 % der insgesamt besten Performance der Ansätze mittels Fingerspitzenerkennung (A2).
Die Resultate der getrennt untersuchten Fingerspitzenerkennung in A2 bestätigen die Analyse der Evaluierung von Klassifikatoren und Fingergestenerkennung. Bei einer korrekten Handdetektion funktioniert die Fingerspitzenerkennung gut für A2-H, A2-T und am besten für A2-D mit einem F-Score Mittelwert von 95,11 % und einem Accuracy Mittelwert von 90,9 %. Mittels A2-C wurde aus den zuvor angeführten Gründen ein geringerer F-Score bzw. Accuracy erreicht. Die Standardabweichung in den Tabellen 6.3 und 6.4, sowie in der Abbildung 6.3b, wird vor allem von den zuvor erläuterten Ergebnissen mit unruhigem Hintergrund und wechselnden Lichtverhältnissen verursacht.

6.2 Evaluierung der 3D-Interaktion

Die Fingergestenerkennung ist essenziell, um 3D-Interaktion zu ermöglichen. Aus diesem Grund wurde mit dem Prototypen und den vorgestellten Ansätzen eine experimentelle Studie zur Evaluierung der Interaktion durchgeführt. Diese wurde mit einem Teilnehmer umgesetzt und untersucht die Aufgaben *Selektion* und *Positionierung* von Objekten [15]. Der Aufbau des Testszenarios wird in Abschnitt 6.2.1 beschrieben. In Abschnitt 6.2.2 werden die Resultate der Evaluierung präsentiert und in Abschnitt 6.2.3 diskutiert.

6.2.1 Testszenario

Für die Evaluierung der Interaktion wurde die in Abbildung 6.4a gezeigte VR-Szene erstellt, bestehend aus einem durch Wände und Boden abgegrenzten 3D-Raum. In Abbildung 6.4b wird das Koordinatensystem der Szene schematisch dargestellt. Zur Interaktion wurde ein Würfelobjekt (blau) hinzugefügt und an die Startposition -3/0/1.5 gesetzt. In der Position 4/0/9 wurde das Zielgebiet erstellt (gelbe Fläche). Zwischen Start- und Zielbereich wurde eine schwarze Wand errichtet, um die beiden Bereiche voneinander zu trennen.



(a) VR-Szene (b) Start und Ziel im Koordinatensystem der VR-Szene

Abbildung 6.4: VR-Szene zur Evaluierung der Interaktion

In der zuvor beschriebenen VR-Szene musste der Benutzer folgende Aufgaben durchführen:

Aufgabe 1: *Selektion* des Würfelobjekts mit *Geste 1*

Aufgabe 2: *Positionierung* des Würfels im Zielbereich mittels *Geste 2*

Mit den in Abschnitt 3.4 vorgestellten Ansätzen A1 und A2 wird die Fingergeste ermittelt und die 3D-Position der Hand geschätzt (siehe Abschnitt 3.6). Diese wird auf die virtuelle Hand übertragen und, wie in Abschnitt 3.7 beschrieben, zur Interaktion mit der VR-Szene verwendet.

6.2.2 Resultate

Das Testszenario wurde mit allen Ansätzen zur Fingergestenerkennung und Tiefenschätzung einmalig durchgeführt. Es fand bei gleichmäßiger Beleuchtung und vor unruhigem Hintergrund statt, um eine natürliche und realistische Umgebungssituation für ein mobiles NUI zu simulieren. Zur Evaluierung der Fingergestenerkennung wurden die ermittelten Fingergesten für die Dauer der Interaktion aufgezeichnet und als Funktion der Zeit dargestellt. Außerdem wurde während der Interaktion die 3D-Position der virtuellen Hand aufgezeichnet und für die Untersuchung der Methoden zur Tiefenschätzung als Funktion der Zeit abgebildet.

6.2.2.1 Fingergestenerkennung

In Abbildung 6.5 werden die Resultate der Fingergesten-Evaluierung mittels A1 dargestellt. A1 erzielt gute Ergebnisse für Aufgabe 1 *Selektion*, welche die Detektion von *Geste 1* erfordert. Während Aufgabe 2 *Positionierung* war die Erkennung von *Geste 2* zweimal nicht korrekt (falsch negativ).

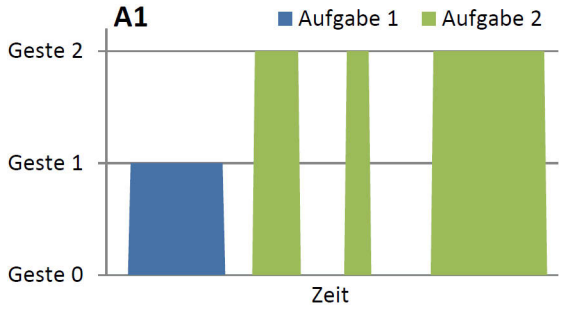
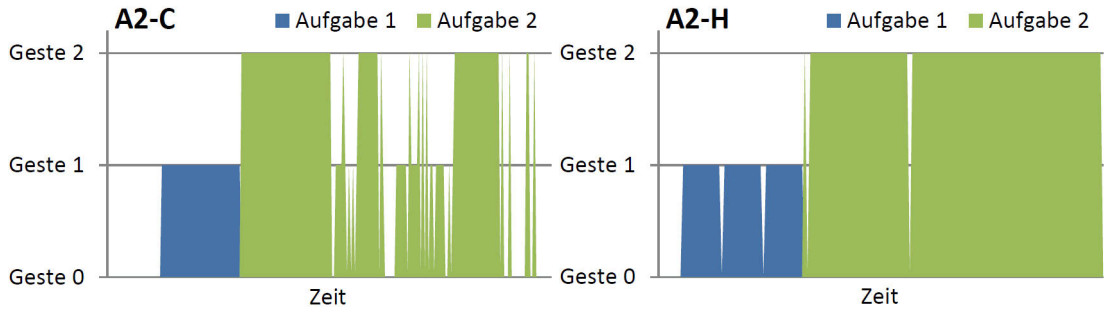


Abbildung 6.5: Evaluierung der Interaktion mit A1

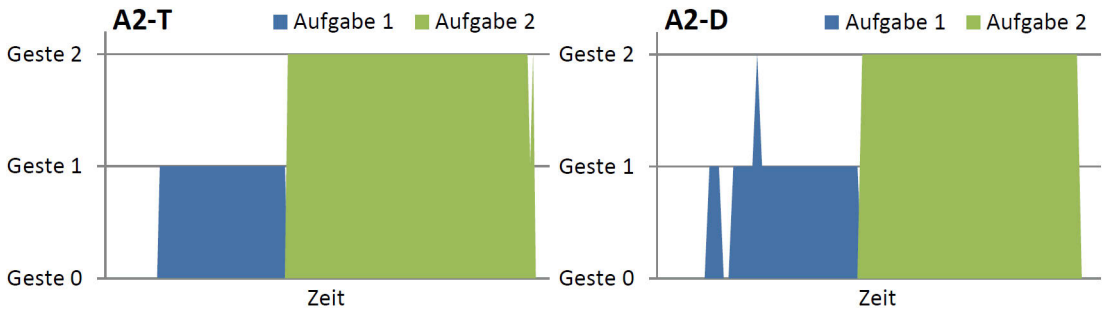
Die Evaluierung von A2-C und A2-H wird in Abbildung 6.6 präsentiert. Wie aus Abbildung 6.6a hervorgeht, weist A2-C eine stabile Fingergestenerkennung während Aufgabe 1 auf. Bei der Durchführung von Aufgabe 2 erzielt A2-C allerdings eine bedeutende Anzahl an falsch negativen Ergebnissen. Mittels A2-H sind in Abbildung 6.6b kurze Unterbrechungen der Fingergestenerkennung während Aufgabe 1 und Aufgabe 2 festzustellen.



(a) A2-C (b) A2-H

Abbildung 6.6: Evaluierung der Interaktion mit A2-C und A2-H

In Abbildung 6.7 werden die Resultate von A2-T und A2-D dargestellt. Mithilfe von A2-T ist eine robuste Fingergestenerkennung während beiden Interaktionsaufgaben gegeben (siehe Abbildung 6.7a). Wie in Abbildung 6.7b zu sehen ist, erkennt A2-D die *Geste 1* zuverlässig (mit Ausnahme von zwei falschen Detektionen) und zeigt eine robuste Fingergestenerkennung während Aufgabe 2.



(a) A2-T (b) A2-D

Abbildung 6.7: Evaluierung der Interaktion mit A2-T und A2-D

6.2.2.2 3D-Position

Nach der Evaluierung von Robustheit und Zuverlässigkeit der Fingergestenerkennung für die Dauer der Interaktion wurde die geschätzte 3D-Position der virtuellen Hand untersucht. Nachfolgend wird für jeden Ansatz der Tiefenschätzung der gesamte Ablauf des Testszenarios als Funktion der Zeit abgebildet und anschließend ausgewertet. Je nach Ansatz muss sich der Benutzer zu Beginn an die Funktionsweise der Tiefenschätzung gewöhnen (*Eingewöhnungsphase*) bzw. die Parameter der Methode an die aktuelle Situation anpassen (*Kalibrierungsphase*).

In Abbildung 6.8 wurde die 3D-Position der virtuellen Hand mithilfe der Handgröße geschätzt. Die Kurve deutet auf eine direkte Erledigung der Aufgaben hin, ohne Eingewöhnungs- oder Kalibrierungsphase für den Benutzer. Allerdings können Abweichungen in der z-Achse beobachtet werden. Diese werden von der variierenden Suchfenstergröße des Klassifikators ausgelöst.

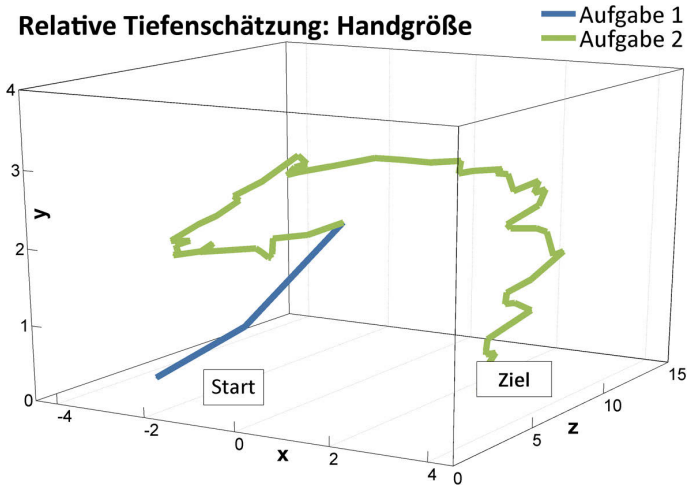


Abbildung 6.8: Evaluierung der 3D-Position der virtuellen Hand - Handgröße

Die Evaluierung der Tiefenschätzung mithilfe des maximalen Grauwerts der Hand wird in Abbildung 6.9 dargestellt. Die Resultate veranschaulichen, dass mit dieser Methode die Aufga-

benstellung des Testszenarios gut gelöst werden konnte. Jedoch wird beim Start der Interaktion eine Kalibrierungsphase benötigt, angezeigt durch die Vor- und Zurück-Bewegungen zu Beginn von Aufgabe 1. Die Abweichungen der z-Position während Aufgabe 2 sind im Vergleich zu Abbildung 6.8 geringer. Die verbleibenden Abweichungen werden von der wechselnden Handbeleuchtung und den damit verbundenen Grauwerten verursacht. Diese wird durch die Veränderung der Position und Ausrichtung der Hand gegenüber der Lichtquelle hervorgerufen. Dadurch verändert sich der maximale Grauwert der Handfläche trotz gleichbleibender Entfernung der Hand.

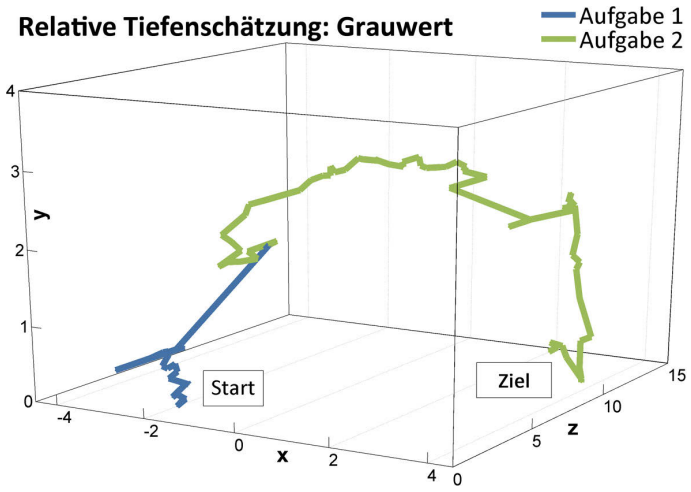


Abbildung 6.9: Evaluierung der 3D-Position der virtuellen Hand - Grauwert

Mithilfe der Tiefendaten der Kinect wurde die absolute 3D-Position der virtuellen Hand in Abbildung 6.10 geschätzt. Die Kurve der aufgezeichneten 3D-Positionen zeigt eine kontinuierliche Schätzung der z-Position und eine robuste 3D-Position für die Dauer der gesamten Interaktion.

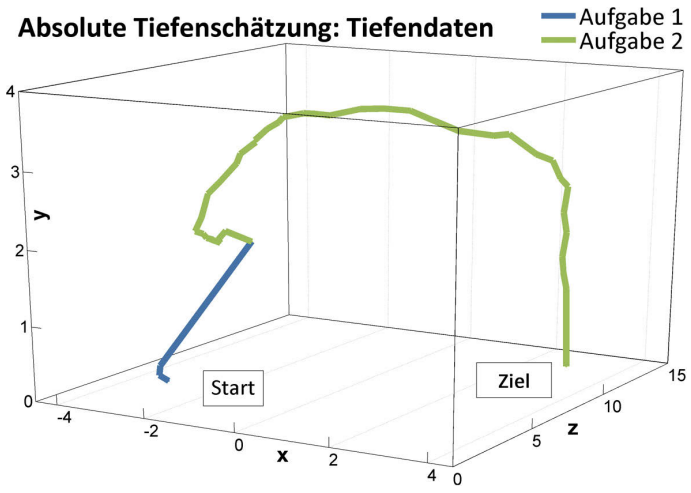


Abbildung 6.10: Evaluierung der 3D-Position der virtuellen Hand - Tiefendaten

6.2.3 Diskussion

Die Analyse der Resultate der Fingergesten-Evaluierung über die Zeit der 3D-Interaktion zeigt eine starke Korrelation zu den in Abschnitt 6.1.2 präsentierten Ergebnissen der Fingergestenerkennung. Damit wird auf die Notwendigkeit einer robusten Fingergestenerkennung hingewiesen, um eine natürliche und direkte Interaktion erreichen zu können. Der Klassifikator K2 hat Probleme mit der Detektion von *Geste 2* in A1. Dies führt zu falsch negativen Ergebnissen während Aufgabe 2. Die Fingergestenerkennung mithilfe der Canny-Kantendetektion in A2-C ist unbeständig und schnitt auch in dieser Evaluierung am schlechtesten ab. Dies begründet sich in der großen Anzahl von Falscherkennungen während der *Positionierungs*-Aufgabe. Die Resultate von A2-T, A2-H und A2-D erreichen insgesamt ein robustes und stabiles Ergebnis, mit einer geringeren Anzahl an falsch negativen Detektionen für die Dauer der Interaktion.

Die Evaluierung der 3D-Position der Hand zeigte für die Methode zur Tiefenschätzung mittels Handgröße zufriedenstellende Ergebnisse. Es ist keine initiale Eingewöhnungs- oder Kalibrierungsphase des Benutzers für die 3D-Interaktion notwendig, wodurch die Steuerung der virtuellen Hand von Beginn an ermöglicht wird. Die Abweichungen in der z-Achse werden zum großen Teil von der Performance der Klassifikatoren verursacht und in ihrem Ausmaß beeinflusst. Durch falsch positive Ergebnisse bei der Handdetektion bzw. einer fehlerhaften Handgröße ist die Schätzung der z-Position unruhig bzw. instabil und beeinträchtigt dadurch die 3D-Interaktion.
Ebenfalls gute Resultate in diesem Testszenario erreichte die Tiefenschätzung mithilfe des maximalen Grauwerts der Hand. Diese Methode funktioniert bei konstanten Lichtverhältnissen und benötigt eine gleichmäßig ausgeleuchtete Hand. Nach einer kurzen Eingewöhnungsphase an den lichtabhängigen Zusammenhang von Grauwert und Tiefenposition, konnte der Benutzer die Aufgaben des Testszenarios ohne weitere Probleme durchführen. Der Grauwertbereich für die Tiefenschätzung muss vom Benutzer manuell an die aktuelle Lichtsituation angepasst werden.
Die robustesten, stabilsten und damit verlässlichsten 3D-Positionen wurden mithilfe der Tiefendaten ermittelt. Diese Methode benötigt keine Eingewöhnungsphase und es traten keine Abweichungen in der z-Achse auf, wodurch eine natürliche, direkte und störungsfreie Interaktion ermöglicht wird.

KAPITEL 7

Zusammenfassung und Ausblick

Die vorliegende Arbeit soll mit der Umsetzung einer intuitiven 3D-Interaktion das Potenzial einer mobilen VR-Anwendung demonstrieren und anhand der Evaluierungsergebnisse die Vor- und Nachteile der implementierten Ansätze zeigen. Dafür wurden zwei berührungs- und markerlose Ansätze (A1 und A2) entwickelt, implementiert und untersucht. Diese Ansätze berücksichtigen die dritte Dimension und binden sie in die Interaktion ein. Mithilfe von zwei Fingergesten und der Kopfposition (HCP) kann auf natürliche und direkte Weise mit 3D-Inhalten interagiert und ein mobiles NUI geschaffen werden.
Für die Umsetzung wurde ein Hardware-Prototyp entwickelt, bestehend aus einem handelsüblichen Tablet (Asus TF201), einer kostengünstigen Tiefenkamera (Kinect für Windows) und einem Rahmen zur Fixierung der beiden Geräte. Das Tablet ist die einzige und zentrale Recheneinheit und wird zur Darstellung der VR-Szene verwendet. Außerdem werden die RGB-Daten der Frontkamera für die Fingergestenerkennung, Detektion des Kopfs und zur relativen Tiefenschätzung genutzt. Als Erweiterung wird die Tiefenkamera für den Erhalt von RGBD-Daten zur Fingergestenerkennung und absoluten Tiefenschätzung verwendet. Hierzu ist die Tiefenkamera direkt mit dem Tablet verbunden, das die RGBD-Daten ausliest und verarbeitet. Mithilfe eines Haar-Klassifikators wird die Kopfposition ermittelt und die Größe des Kopfs zur relativen Tiefenschätzung verwendet. Für die Ermittlung von zwei unterschiedlichen Fingergesten wurden folgenden Ansätze entwickelt:

- **A1:** In A1 werden die zwei getrennten Haar-Klassifikatoren K1 und K2 für die Fingergestenerkennung eingesetzt. Mithilfe von K1 wird die *Geste 1* erkannt und mittels K2 die *Geste 2*, wodurch eine eindeutige Zuordnung der Geste möglich ist und keine weiteren Schritte benötigt werden. Durch die Gestenerkennung ist außerdem die Handgröße und 2D-Position der Hand bekannt. Die 3D-Position wird mithilfe der Handgröße oder dem maximalen Grauwert der Hand relativ geschätzt. In der Erweiterung A1-D werden RGBD-Daten zur absoluten Tiefenschätzung verwendet.
- **A2:** In A2 wird der Haar-Klassifikator K3 für die Erkennung der Hand verwendet und die Fingergesten mithilfe der Fingerspitzenerkennung ermittelt. Dafür wird die Anzahl

der erkannten Fingerspitzen herangezogen und vier bis fünf Fingerspitzen als *Geste 1* und ein bis zwei Fingerspitzen als *Geste 2* definiert. Mit der Detektion der Hand steht die Größe und 2D-Position der Hand zur Verfügung, wodurch die Fingerspitzenerkennung auf die Handregion begrenzt werden kann. Für die Fingerspitzenerkennung wird die Hand segmentiert, die Handkontur ermittelt und anschließend auf Fingerspitzen hin untersucht. A2 verwendet verschiedene Methoden zur Handsegmentierung bzw. zur Ermittlung der Kontur und wird in die folgenden Gruppen unterteilt:

- **A2-C:** Mittels Canny-Kantendetektion [19] werden die Konturen in der Handregion ermittelt.
- **A2-H:** Segmentierung der Hand durch einen automatisch angepassten Schwellwert, basierend auf den HSV-Werten der Hand.
- **A2-T:** Durch einen vom Benutzer anpassbaren Schwellwert wird die Hand segmentiert.
- **A2-D:** Mithilfe der Tiefendaten werden alle Pixel die sich hinter der Hand befinden mittels Schwellwert ausgeblendet und die Hand segmentiert.

Die 3D-Position wird in A2-C, A2-H und A2-T mittels der Größe oder dem maximalen Grauwert der Hand relativ geschätzt, während A2-D die RGBD-Daten zur absoluten Tiefenschätzung verwendet.

Mit den verschiedenen Technologien und Hardware-Komponenten ein funktionierendes Zusammenspiel zu erstellen, war eine interessante Herausforderung dieser Arbeit. Dabei bestand die Schwierigkeit darin, die einzelnen Teile des Prototypen in eine Android-Anwendung einzubinden und aufeinander abzustimmen. Zu Beginn wurde die Kommunikation zwischen Kinect und Tablet aufgrund fehlender offizieller Unterstützung für Android mithilfe einer alternativen Verbindung über ein Netzwerk implementiert. Allerdings konnte schlussendlich ein alternativer Treiber für Android kompiliert und zur Verbindung per USB verwendet werden, um ein schnelles und robustes Einlesen der Tiefendaten zu ermöglichen. Das Erstellen und Trainieren von eigenen und robusten Klassifikatoren war ebenfalls eine herausfordernde Aufgabe, mit vielen zu testenden Möglichkeiten und Konfigurationen (siehe Abschnitt 4.3.2).

Im Vergleich zur Interaktion mit einem Desktop-System in einer definierten Umgebung sind bei mobilen NUIs verschiedene Licht- und Hintergrundsituationen zu berücksichtigen. Außerdem müssen alle Schritte der Methoden für den mobilen Betrieb optimiert werden, damit eine Echtzeit-Interaktion möglich ist. Um die Vor- und Nachteile der implementierten Ansätze zu ermitteln, wurden diese in unterschiedlichen Situationen mit dem Prototypen evaluiert. Die Resultate zeigen, dass alle Ansätze die Daten mit der benötigten interaktiven Framerate verarbeiten. Mit den implementierten Ansätzen konnte gezeigt werden, dass der Prototyp eine natürliche und direkte 3D-Interaktion mit den Inhalten einer mobilen VR-Szene ermöglicht. Außerdem konnten mithilfe von RGBD-Daten die Segmentierung der Hand und vor allem die Interaktion in der Tiefenachse verbessert werden. Anhand der Ergebnisse der Evaluierung konnten folgende Empfehlungen für die untersuchten Methoden zur Fingergestenerkennung und Tiefenschätzung ausgearbeitet werden:

- **Fingergestenerkennung:**
  - **A1:** Mit A1 wird in allen Situationen eine robuste 3D-Interaktion ermöglicht, sofern der verwendete Klassifikator gut trainiert wurde. Allerdings weist dieser Ansatz eine geringere Framerate auf und benötigt für jede zu erkennende Geste einen getrennten Klassifikator.
  - **A2-C:** Dieser Ansatz kann nicht zwischen einer Handkontur oder anderen Konturen unterscheiden und ist deshalb für die Umsetzung eines mobilen NUIs nicht geeignet.
  - **A2-H:** Stehen nur RGB-Daten zur Verfügung und wird eine schnelle Verarbeitung benötigt ist A2-H die beste Wahl.
  - **A2-T:** Für die 3D-Interaktion vor weißem Hintergrund ist A2-T geeignet und bietet hohe Frameraten, sowie eine einfache Implementierung. Bei wechselnden Lichtverhältnissen muss der Schwellwert manuell an die aktuelle Situation angepasst werden.
  - **A2-D:** Der robusteste Ansatz für eine 3D-Interaktion in allen Situation ist A2-D, allerdings mit einer geringeren Framerate.
- **Tiefenschätzung:**
  - **Handgröße:** Sie ist ein zuverlässiges Merkmal für die relative Tiefenschätzung, sofern der Klassifikator gut trainiert wurde. Die Verbindung zwischen Handgröße und der 3D-Position ist für den Benutzer ohne Eingewöhnungsphase nachvollziehbar.
  - **Grauwert:** Der maximale Grauwert der Hand kann als solides Merkmal für die relative Tiefenschätzung verwendet werden, wenn die Hand gleichmäßig ausgeleuchtet ist und konstante Lichtverhältnisse herrschen. Der Grauwertbereich muss zu Beginn vom Benutzer an die aktuelle Lichtsituation angepasst. Außerdem benötigt der Benutzer eine kurze Eingewöhnungsphase, um den lichtabhängigen Zusammenhang von Grauwert und Tiefenposition herstellen zu können.
  - **Tiefendaten:** Mithilfe von Tiefendaten wird eine absolute Tiefenschätzung ermöglicht, welche sich nahtlos in die Steuerung der Hand integriert und somit eine natürliche 3D-Interaktion erlaubt.

In zukünftigen Ansätzen könnten dynamische Gesten (siehe Abschnitt 2.1.3) und somit die Möglichkeiten zur Interaktion weiter erhöht werden. Ebenfalls sind die Fortschritte und Weiterentwicklungen von mobilen Geräten mit integrierten Tiefensensoren für zukünftige Arbeiten interessant (beispielsweise Googles *Project Tango* [43]). Die Integration der Sensoren in ein gemeinsames System erleichtert die Handhabung für den Benutzer. Außerdem sollten durch ein gemeinsames und vom Hersteller abgestimmtes System etwaige Komplikationen zwischen den Sensoren wegfallen und eine Verbesserung der Performance erreicht werden. Damit könnten die Ansätze des Prototypen im Alltag verwendet werden, sodass in Zukunft mobile Geräte als mobile NUIs für die robuste, natürliche und intuitive 3D-Interaktion mit Inhalten in virtueller Realität eingesetzt werden.

Teil IV

Anhang

ANHANG A

Kinect für Android - Anleitung

Für die Umsetzung werden die von Lo [68] und Niisato [80] beschriebenen Schritte befolgt und mittels Kommandozeile durchgeführt. Die einzelnen Schritte werden in den nächsten Abschnitten detailliert ausgeführt.

A.1 Anforderungen

Es werden Root- bzw. Administratorrechte für das Tablet benötigt, um die Dateien in die entsprechenden Ordner laden zu können. Android NDK [39] (getestet mit Revision 8) wird verwendet, um OpenNI und die Treiber neu zu kompilieren. Dafür wird ein PC mit *Windows 7 64-bit* und der Kommandozeileninterpreter *cmd.exe* (als Administrator ausführen) genutzt. Für die Übertragung der Dateien wird das Tablet per USB mit dem PC verbunden, die notwendigen Treiber/Einstellungen werden für eine erfolgreiche Verbindung vorausgesetzt.

Für die Kinect Xbox 360 kann auf die für Android kompilierten Dateien von Niisato [80] zurückgegriffen werden und Abschnitt A.2 übersprungen werden. Die Kinect für Windows benötigt einen anderen Treiber und damit kompatible OpenNI-Dateien welche neu kompiliert werden müssen, siehe Abschnitt A.2.

A.2 OpenNI und Kinect-Treiber für Android kompilieren

In den nächsten Abschnitten werden die nötigen Schritte beschrieben, um OpenNI und die Kinect-Treiber für Android zu erhalten. Außerdem werden in Abschnitt A.2.3 Lösungen für eventuell auftretende Probleme aufgelistet.

A.2.1 OpenNI

1. OpenNI herunterladen und nach *C:/Android/android-ndk/sources/* entpacken [97]
   (Wichtig: Versionsangabe von SensorKinect [2], hier 1.5.4.0 unstable, beachten)

2. `cd C:/Android/android-ndk/sources/OpenNI/Platform/Android/jni`
3. `ndk-build`
   Erstellt mindestens folgende Dateien in */OpenNI/Platform/Android/libs/armeabi-v7a*: libnimCodecs.so, libnimMockNodes.so, libnimRecorder.so, libOpenNI.jni.so, libOpenNI.so, libusb.so, niLicense, niReg, Sample-SimpleRead

A.2.2 Kinect Treiber

1. SensorKinect Treiber [2] herunterladen und zum Beispiel nach *C:/* entpacken
2. `set NDK_MODULE_PATH=C:/Android/android-ndk/sources/OpenNI/Platform/Android/jni`
3. `cd C:/SensorKinect/Platform/Android/jni`
4. `ndk-build`
   Erstellt folgende Dateien in */SensorKinect/Platform/Android/libs/armeabi-v7a*: libOpenNI.so, libusb.so, libXnCore.so, libXnDDK.so, libXnDeviceFile.so, libXnDeviceSensorV2.so und libXnFormats.so

A.2.3 Problembehandlung

- Error `/NiSkeletonBenchmark/NiSkeletonBenchmark.o' failed`:
  Den Ordner *Platform/Android/jni/Samples/NiSkeletonBenchmark* löschen
- Error `unrecognized option '-dynamic-linker'`:
  In den betroffenen *Android.mk*-Dateien diese Option entfernen
- Error `permission denied`:
  cmd.exe als Administrator ausführen
- Probleme mit Android 4.x:
  In den *Application.mk*-Dateien *APP_PLATFORM* auf *android-14* ändern (*APP_PLATFORM := android-14*)

A.3 Dateien hochladen

Das Tablet wird per USB mit dem PC verbunden und die Dateien mithilfe der nachfolgenden Schritte in die entsprechenden Ordner des Tablets hochgeladen. Außerdem werden benötigte Einstellungs- und Konfigurationsdateien auf das Tablet kopiert.

1. System Ordner laden und Ordner erstellen:
   ```
   adb shell
   su
   mount -o remount,rw /system
   mkdir /data/ni
   ```

2. Dateien von OpenNI hochladen:
   ```
   adb push libnimCodecs.so /system/lib
   adb push libnimMockNodes.so /system/lib
   adb push libnimRecorder.so /system/lib
   adb push libOpenNI.jni.so /system/lib
   ```

3. Dateien von SensorKinect hochladen:
   ```
   adb push libOpenNI.so /system/lib
   adb push libusb.so /system/lib
   adb push libXnCore.so /system/lib
   adb push libXnDDK.so /system/lib
   adb push libXnDeviceFile.so /system/lib
   adb push libXnDeviceSensorV2.so /system/lib
   adb push libXnFormats.so /system/lib
   ```

4. OpenNI sucht nach den *Shared Libraries* (.so), diese werden in *modules.xml* angegeben. Beispiel für modules.xml:

   ```
   <Modules>
       <Module path="/system/lib/libnimMockNodes.so" />
       <Module path="/system/lib/libnimCodecs.so" />
       <Module path="/system/lib/libnimRecorder.so" />
       <Module path="/system/lib/libXnDeviceSensorV2.so"
               configDir="/data/ni/" />
       <Module path="/system/lib/libXnDeviceFile.so"
               configDir="/data/ni/"/>
   </Modules>
   ```

   modules.xml hochladen:
   ```
   adb push modules.xml /data/ni
   ```

5. GlobalDefaultsKinect.ini bearbeiten und hochladen:
   C:/SensorKinect/Data/GlobalDefaultsKinect.ini öffnen
   UsbInterface = 1 setzen
   ```
   adb push GlobalDefaultsKinect.ini /data/ni/
   ```

6. SamplesConfig.xml hochladen:
   (zum Beispiel von *C:/Android/android-ndk/sources/OpenNI/Data/*)
   ```
   adb push SampleConfig.xml /data/ni/
   ```

A.4 Kinect mounten (nach jedem Neustart erforderlich)

Mounten:

```
adb shell
su
mount -o devmode=0666 -t usbfs none /proc/bus/usb
```

Gegebenenfalls wiederholen bzw. Applikation neu öffnen (aus Cache löschen).

A.5 Test

Wurden die obigen Schritte erfolgreich durchgeführt, kann mittels den Beispielprogrammen *niReg* oder *Sample-SimpleRead* die Funktion der Kinect getestet werden.

1. niReg von OpenNI hochladen:
   ```
   adb push niReg /system/bin
   ```
2. Starten (im *system* Ordner)
   ```
   su
   niReg -l
   ```
3. Ausgabe (ähnlich zu folgendem):
   ```
   . . .
   /system/lib/libXnDeviceSensorV2.so
   (compiled with OpenNI 1.5.2.23):
   Bei Kinect für Windows: (compiled with OpenNI 1.5.4.0):
   Device: PrimeSense/SensorV2/5.1.0.41
   Depth: PrimeSense/SensorV2/5.1.0.41
   Image: PrimeSense/SensorV2/5.1.0.41
   IR: PrimeSense/SensorV2/5.1.0.41
   Audio: PrimeSense/SensorV2/5.1.0.41
   . . .
   ```

ANHANG B

Ergebnisse der Kalibrierung

In der Tabelle B.1 werden die Ergebnisse der intrinsischen Kalibrierung der RGB- und Infrarot-Kamera mittels MIP-MCC [111], Matlab Toolbox [13] und GML [126] dargestellt. Tabelle B.2 zeigt die Ergebnisse der extrinsischen Kalibrierung (mit der RGB-Kamera im Ursprung) mittels MIP-MCC [111] und Matlab Toolbox [13].

Tabelle B.1: Ergebnisse der intrinsischen Kalibrierung

RGB	MIP-MCC	Matlab	GML	**Infrarot**	MIP-MCC	Matlab	GML
Hauptpunkt							
cx	572.171	572.289	572.153		604.288	604.353	604.286
cy	571.303	571.423	571.286		603.298	603.363	603.296
Brennweite							
fx	323.114	323.110	323.110		317.744	317.741	317.738
fy	245.949	245.925	245.964		244.176	244.160	244.180

Tabelle B.2: Ergebnisse der extrinsischen Kalibrierung

RGB im Ursprung	MIP-MCC	Matlab
Rotationsvektor		
x	-0.018	-0.037
y	0.013	0.026
z	-0.004	-0.008
Translationsvektor		
x	3.892	3.904
y	-22.821	-22.816
z	-15.644	-15.580

Literaturverzeichnis

[1] Integrating Unity with Eclipse. [Online] http://www.rbcafe.com/Softwares/Unity/Documentation/Manual/Android-IntegratingUnityWithEclipse.html (Besucht am 03.09.2014).

[2] SensorKinect. [Software] Version 0.93 https://github.com/avin2/SensorKinect (Besucht am 03.09.2014).

[3] Using Unity Android In a Sub View. [Online] http://forum.unity3d.com/threads/98315-Using-Unity-Android-In-a-Sub-View (Besucht am 03.09.2014).

[4] J. H. An, J. H. Min und K. S. Hong. Finger gesture estimation for mobile device user interface using a rear-facing camera. In *Future Information Technology*, pages 230–237. Springer, Berlin Heidelberg, 2011.

[5] M. Baldauf, S. Zambanini, P. Fröhlich und P. Reichl. Markerless visual fingertip detection for natural mobile device interaction. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 539–544. ACM, 2011.

[6] D. H. Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[7] S. Bilal, R. Akmeliawati, M. J. E. Salami, A. a. Shafie und E. M. Bouhabba. A hybrid method using haar-like and skin-color algorithm for hand posture detection, recognition and tracking. In *Proceedings of International Conference on Mechatronics and Automation (ICMA)*, pages 934–939. IEEE, 2010.

[8] R. W. W. Bollhoefer, K. Meyer und R. Witzsche. Microsoft Surface und das Natural User Interface (NUI). Technical report, Pixelpark, 2009.

[9] Boost. A synchronous TCP daytime client. [Online] http://www.boost.org/doc/libs/1_42_0/doc/html/boost_asio/tutorial/tutdaytime1.html (Besucht am 03.09.2014).

[10] Boost. A synchronous TCP daytime server. [Online] `http://www.boost.org/doc/libs/1_42_0/doc/html/boost_asio/tutorial/tutdaytime2.html` (Besucht am 03.09.2014).

[11] Boost. Boost C++ Libraries. [Software] Release 1.49.0 `http://www.boost.org/` (Besucht am 03.09.2014).

[12] M. Bordegoni und M. Hemmje. A Dynamic Gesture Language and Graphical Feedback for Interaction in a 3D User Interface. *Computer Graphics Forum*, 12(3):1–11, Aug. 1993.

[13] J.-Y. Bouguet. Camera Calibration Toolbox for Matlab. [Online] `http://www.vision.caltech.edu/bouguetj/calib_doc/` (Besucht am 03.09.2014).

[14] D. Bowman, C. Wingrave, J. Campbell und V. Ly. Using Pinch Gloves (TM) for both Natural and Abstract Interaction Techniques in Virtual Environments. In *Proceedings of HCI International*, volume 1, pages 629–633. Lawrence Erlbaum Associates, 2001.

[15] D. A. Bowman, E. Kruijff, J. J. LaViola und I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley, 2004.

[16] G. Bradski und A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media Inc., 2008.

[17] N. Burrus. Kinect Calibration. [Online] `http://nicolas.burrus.name/index.php/Research/KinectCalibration` (Besucht am 03.09.2014).

[18] B. Buxton. Multi-Touch Systems that I Have Known and Loved. [Online] `http://www.billbuxton.com/multitouchOverview.html` (Besucht am 03.09.2014).

[19] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[20] Q. Chen, N. D. Georganas und E. M. Petriu. Hand gesture recognition using Haar-like features and a stochastic context-free grammar. *IEEE Transactions on Instrumentation and Measurement*, 57(8):1562–1571, 2008.

[21] W. H. Chun und T. Höllerer. Real-time hand interaction for augmented reality on mobile phones. In *Proceedings of the 2013 international conference on Intelligent user interfaces (IUI)*, pages 307–314. ACM, 2013.

[22] F. M. Ciaramello und S. S. Hemami. Real-time face and hand detection for videoconferencing on a mobile device. In *Proceedings of the 4th International Workshop on Video Processing and Quality Metrics for Consumer Electronics (VPQM)*, 2009.

[23] F. C. Crow. Summed-Area Tables for Texture Mapping. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, volume 18, pages 207–212. ACM, 1984.

[24] CyberGlove Systems. CyberGlove II. [Online] `http://www.cyberglovesystems.com/products/cyberglove-ii/photos-video` (Besucht am 03.09.2014).

[25] H. Distler. *Wahrnehmung in virtuellen Welten*. PhD thesis, Justus-Liebig-Universität Gießen, 2003.

[26] Eclipse Foundation Inc. Eclipse CDT. [Software] Version 8.0.2 `https://www.eclipse.org/cdt/` (Besucht am 03.09.2014).

[27] Eclipse Foundation Inc. Eclipse IDE for Java Developers. [Software] Indigo SR 2 `https://www.eclipse.org/` (Besucht am 03.09.2014).

[28] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006.

[29] J. Francone und L. Nigay. Using the user's point of view for interaction on mobile devices. In *Proceedings of the 23rd French Speaking Conference on Human-Computer Interaction (IHM)*, pages 4:1–4:8. ACM, 2011.

[30] B. Freedman, A. Shpunt, M. Machline und Y. Arieli. Depth Mapping Using Projected Patterns. U.S. Patent 2010/0118123.

[31] Y. Freund und R. E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning (ICML)*, volume 96, pages 148–156. Morgan Kaufmann, 1996.

[32] J. Friedman, T. Hastie und R. Tibshirani. Additive logistic regression: A statistical view of boosting. *Annals of Statistics*, 28(2):337–407, 2000.

[33] S. E. Ghobadi, O. E. Loepprich, F. Ahmadov, J. Bernshausen, K. Hartmann und O. Loffeld. Real Time Hand Based Robot Control Using 2D/3D Images. In G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, F. Porikli, J. Peters, J. Klosowski, L. Arns, Y. Chun, T.-M. Rhyne und L. Monroe, editors, *Advances in Visual Computing*, pages 307–316. Springer Berlin Heidelberg, 2008.

[34] Google Inc. ADT Plugin. [Software] Version 22.0.5 `http://developer.android.com/tools/sdk/eclipse-adt.html` (Besucht am 03.09.2014).

[35] Google Inc. Android. [Online] `http://www.android.com/` (Besucht am 03.09.2014).

[36] Google Inc. Android Activity. [Online] `http://developer.android.com/reference/android/app/Activity.html` (Besucht am 03.09.2014).

[37] Google Inc. Android FrameLayout. [Online] `http://developer.android.com/reference/android/widget/FrameLayout.html` (Besucht am 03.09.2014).

[38] Google Inc. Android Intent. [Online] http://developer.android.com/reference/android/content/Intent.html (Besucht am 03.09.2014).

[39] Google Inc. Android NDK. [Software] Revision 8c http://developer.android.com/tools/sdk/ndk/index.html (Besucht am 03.09.2014).

[40] Google Inc. Android SDK. [Software] Revision 21 http://developer.android.com/sdk/index.html (Besucht am 03.09.2014).

[41] Google Inc. Android SurfaceView. [Online] http://developer.android.com/reference/android/view/SurfaceView.html (Besucht am 03.09.2014).

[42] Google Inc. JNI Tips. [Online] http://developer.android.com/training/articles/perf-jni.html (Besucht am 03.09.2014).

[43] Google Inc. Project Tango. [Online] https://www.google.com/atap/projecttango/ (Besucht am 03.09.2014).

[44] Google Inc. Setting Up an Existing IDE. [Online] http://developer.android.com/sdk/installing/index.html (Besucht am 03.09.2014).

[45] S. Gupta, A. Dasgupta und A. Routray. Analysis of training parameters for classifiers based on Haar-like features to detect human faces. In *International Conference on Image Information Processing (ICIIP)*, pages 1–4. IEEE, 2011.

[46] G. Hackenberg, R. McCall und W. Broll. Lightweight palm and finger tracking for real-time 3D gesture control. In *Proceedings of the 2011 IEEE Virtual Reality Conference (VR)*, pages 19–26. IEEE, 2011.

[47] M. Hancock, S. Carpendale und A. Cockburn. Shallow-depth 3d interaction: design and evaluation of one-, two-and three-touch techniques. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI)*, pages 1147–1156. ACM, 2007.

[48] T. R. Hansen, E. Eriksson und A. Lykke-Olesen. Mixed Interaction Space - Expanding the Interaction Space with Mobile Devices. In T. McEwan, J. Gulliksen und D. Benyon, editors, *People and Computers XIX - The Bigger Picture*, pages 365–380. Springer London, 2006.

[49] T. R. Hansen, E. Eriksson und A. Lykke-Olesen. Use your head: exploring face tracking for mobile interaction. In *Proceedings of the Extended Abstracts on Human Factors in Computing Systems (CHI)*, pages 845–850. ACM, 2006.

[50] L. Hardesty. Gesture-based computing on the cheap. [Online] http://web.mit.edu/newsoffice/2010/gesture-computing-0520.html (Besucht am 03.09.2014).

[51] W. Hürst, C. Wezel und C. van Wezel. Gesture-based interaction via finger tracking for mobile augmented reality. *Multimedia Tools and Applications*, 62(1):233–258, Jan. 2013.

[52] International Data Corp. (IDC). Top Smartphone Operating Systems. [Online] `http://www.idc.com/getdoc.jsp?containerId=prUS24257413` (Besucht am 03.09.2014).

[53] International Data Corp. (IDC). Top Tablet Operating Systems. [Online] `http://www.idc.com/getdoc.jsp?containerId=prUS24253413` (Besucht am 03.09.2014).

[54] M. Kallmann und D. Thalmann. Direct 3D interaction with smart objects. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST)*, pages 124–130. ACM, 1999.

[55] C. Keskin, A. Erkan und L. Akarun. Real time hand tracking and 3d gesture recognition for interactive interfaces using hmm. In *Proceedings of the Joint International Conference ICANN/ICONIP*, pages 3–6. Springer, 2003.

[56] H. Ketabdar und A. Jahanbekam. MagiMusic: using embedded compass (magnetic) sensor for touch-less gesture based interaction with digital music instruments in mobile devices. In *Proceedings of the 5th International Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, pages 241–244. ACM, 2011.

[57] K. Khoshelham und S. O. Elberink. Accuracy and resolution of Kinect depth data for indoor mapping applications. *Sensors (Basel, Switzerland)*, 12(2):1437–1454, 2012.

[58] K. Konolige und P. Mihelich. Technical description of Kinect calibration. [Online] `http://wiki.ros.org/kinect_calibration/technical` (Besucht am 03.09.2014).

[59] M. Lee, R. Green und M. Billinghurst. 3D natural hand interaction for AR applications. In *Proceedings of the 23rd International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pages 1–6. IEEE, 2008.

[60] T. Lee und T. Höllerer. Handy AR: Markerless inspection of augmented reality objects using fingertip tracking. In *Proceedings of the 11th IEEE International Symposium on Wearable Computers*, pages 83–90. IEEE, 2007.

[61] T. Lee und T. Höllerer. Hybrid feature tracking and user interaction for markerless augmented reality. In *Proceedings of the 2008 IEEE Virtual Reality Conference (VR)*, pages 145–152. IEEE, 2008.

[62] W. Lee. Kinect Color - Detph Camera Calibration. [Online] `http://cv4mar.blogspot.co.at/2011/03/kinect-color-detph-camera-calibration.html` (Besucht am 03.09.2014).

[63] Z. Li und R. Jarvis. Real time hand gesture recognition using a range camera. In *Proceedings of the 2009 Australasian Conference on Robotics and Automation (ACRA)*, pages 21–27, 2009.

[64] R. Lienhart, A. Kuranov und V. Pisarevsky. Empirical analysis of detection cascades of boosted classifiers for rapid object detection. In *Proceedings of the 25th DAGM Pattern Recognition Symposium*, pages 297–304, 2003.

[65] R. Lienhart und J. Maydt. An extended set of haar-like features for rapid object detection. In *Proceedings of the International Conference on Image Processing (ICIP)*, volume 1, pages I–900 – I–903. IEEE, 2002.

[66] L. Lin, Y. Cong und Y. Tang. Hand gesture recognition using RGB-D cues. In *Proceedings of the IEEE International Conference on Information and Automation (ICIA)*, pages 311–316. IEEE, 2012.

[67] M. Lindner, I. Schiller, A. Kolb und R. Koch. Time-of-Flight sensor calibration for accurate range sensing. *Computer Vision and Image Understanding*, 114(12):1318–1328, 2010.

[68] R. Lo. OpenNI and Kinect for Android Tutorial. [Online] `http://pointclouds.org/blog/nvcs/raymondlo84/index.php` (Besucht am 03.09.2014).

[69] C. Manresa und J. Varona. Hand tracking and gesture recognition for human-computer interaction. *Electronic letters on computer vision and image analysis*, 5(3):96–104, 2005.

[70] A. Martinet, G. Casiez und L. Grisoni. The design and evaluation of 3D positioning techniques for multi-touch displays. In *Proceedings of the 2010 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 115–118. IEEE, 2010.

[71] A. Martinet, G. Casiez und L. Grisoni. Integrality and separability of multitouch interaction techniques in 3D manipulation tasks. *IEEE Transactions on Visualization and Computer Graphics*, 18(3):369–380, 2012.

[72] M. Matilainen, J. Hannuksela und L. Fan. Finger Tracking for Gestural Interaction in Mobile Devices. *Image Analysis*, pages 329–338, 2013.

[73] Microsoft Inc. Depth Space Range. [Online] `http://msdn.microsoft.com/en-us/library/hh973078.aspx#Depth_Ranges` (Besucht am 03.09.2014).

[74] Microsoft Inc. DepthImageFormat Enumeration. [Online] `http://msdn.microsoft.com/en-us/library/microsoft.kinect.depthimageformat.aspx` (Besucht am 03.09.2014).

[75] Microsoft Inc. Kinect for Windows Developer Toolkit. [Software] Version 1.6 `http://www.microsoft.com/en-us/download/details.aspx?id=34807` (Besucht am 03.09.2014).

[76] Microsoft Inc. Kinect for Windows SDK. [Software] Version 1.6 `http://www.microsoft.com/en-us/download/details.aspx?id=34808` (Besucht am 03.09.2014).

[77] Microsoft Inc. Kinect SDK - Hands-on. [Online] http://blog.mic-belgique.be/articles/kinect-sdk-1-0-hands-on/ (Besucht am 03.09.2014).

[78] A. Mossel, B. Venditti und H. Kaufmann. 3DTouch and HOMER-S: Intuitive Manipulation Techniques for One-Handed Handheld Augmented Reality. In *Proceedings of the 15th International Conference of Virtual Technoloogies (VRIC'13)*, pages 12:1–12:10. ACM, 2013.

[79] A. A. Nayak. How to make your own haar trained ".xml"files. [Online] http://nayakamitarup.blogspot.co.at/2011/07/how-to-make-your-own-haar-trained-xml.html (Besucht am 03.09.2014).

[80] H. Niisato. OpenNI and SensorKinect for Android. [Online] http://web.archive.org/web/20130925011931/http://www.hirotakaster.com/archives/2012/05/openni-and-sensorkinect-for-android.php (Besucht am 03.09.2014).

[81] E. S. Nygard. Multi-touch Interaction with Gesture Recognition. Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, 2010.

[82] I. Oikonomidis, N. Kyriazis und A. Argyros. Efficient model-based 3D tracking of hand articulations using Kinect. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 101.1–101.11. BMVA Press, 2011.

[83] OpenCV. Android Development with OpenCV. [Online] http://docs.opencv.org/2.4.6/doc/tutorials/introduction/android_binary_package/dev_with_OCV_on_Android.html (Besucht am 03.09.2014).

[84] OpenCV. Camera Calibration and 3D Reconstruction. [Online] http://docs.opencv.org/2.4.6/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html (Besucht am 03.09.2014).

[85] OpenCV. Canny - Feature Detection. [Online] http://docs.opencv.org/2.4.6/modules/imgproc/doc/feature_detection.html#canny (Besucht am 03.09.2014).

[86] OpenCV. detectMultiScale - Cascade Classification. [Online] http://docs.opencv.org/2.4.6/modules/objdetect/doc/cascade_classification.html#cascadeclassifier-detectmultiscale (Besucht am 03.09.2014).

[87] OpenCV. Image Filtering. [Online] http://docs.opencv.org/2.4.6/modules/imgproc/doc/filtering.html (Besucht am 03.09.2014).

[88] OpenCV. Image Processing. [Online] http://docs.opencv.org/2.4.6/modules/imgproc/doc/imgproc.html (Besucht am 03.09.2014).

[89] OpenCV. Introduction into Android Development. [Online] http://docs.opencv.org/2.4.6/doc/tutorials/introduction/android_binary_package/android_dev_intro.html#android-dev-intro (Besucht am 03.09.2014).

[90] OpenCV. OpenCV library. [Software] Version 2.4.6 http://opencv.org/ (Besucht am 03.09.2014).

[91] OpenCV. OpenCV4Android SDK. [Online] http://docs.opencv.org/2.4.6/doc/tutorials/introduction/android_binary_package/O4A_SDK.html (Besucht am 03.09.2014).

[92] OpenCV. Operations on Arrays. [Online] http://docs.opencv.org/2.4.6/modules/core/doc/operations_on_arrays.html (Besucht am 03.09.2014).

[93] OpenCV. Structural Analysis and Shape Descriptors. [Online] http://docs.opencv.org/2.4.6/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html (Besucht am 03.09.2014).

[94] OpenCV. VideoCapture - Reading and Writing Images and Video. [Online] http://docs.opencv.org/2.4.6/modules/highgui/doc/reading_and_writing_images_and_video.html#videocapture (Besucht am 03.09.2014).

[95] OpenNI. Context Class Reference. [Online] http://kinectcar.ronsper.com/docs/openni/classxn_1_1_context.html (Besucht am 03.09.2014).

[96] OpenNI. DepthGenerator Class Reference. [Online] http://kinectcar.ronsper.com/docs/openni/classxn_1_1_depth_generator.html (Besucht am 03.09.2014).

[97] OpenNI. OpenNI SDK. [Software] Version 1.5.4.0 unstable https://github.com/OpenNI/OpenNI/releases/tag/Unstable-1.5.4.0 (Besucht am 03.09.2014).

[98] Oracle Corp. Java-Development-Kit. [Software] Version 6 http://www.oracle.com/technetwork/java/javase/downloads/index.html (Besucht am 03.09.2014).

[99] J. M. Palacios, C. Sagüés, E. Montijano und S. Llorente. Human-Computer Interaction Based on Hand Gestures Using RGB-D Sensors. *Sensors*, 13(9):11842–11860, 2013.

[100] Z. Pan, Y. Li, M. Zhang, C. Sun, K. Guo, X. Tang und S. Z. Zhou. A real-time multi-cue hand tracking algorithm based on computer vision. In *Proceedings of the 2010 IEEE Virtual Reality Conference (VR)*, pages 219–222. IEEE, 2010.

[101] M. Park, M. M. Hasan, J. Kim und O. Chae. Hand Detection and Tracking Using Depth and Color Information. In *Proceedings of the 2012 International Conference on Image Processing, Computer Vision, and Pattern Recognition (IPCV)*, volume 1, pages 779–785. CSREA Press, 2012.

[102] V. I. Pavlovic, R. Sharma und T. S. Huang. Visual interpretation of hand gestures for human-computer interaction: A review. In *Proceedings of the IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, volume 19, pages 677–695. IEEE, 1997.

[103] M. Pesce. Asus Eee Pad Transformer Prime. [Online] `http://www.wired.com/reviews/2012/01/asus-transformer-prime/` (Besucht am 03.09.2014).

[104] pmdtechnologies GmbH. pmd[vision]® CamBoard nano. [Online] `http://www.pmdtec.com/products_services/reference_design.php` (Besucht am 03.09.2014).

[105] S. Ranganath, D. Ghosh und N. Y. Y. Kevin. Trajectory modeling in gesture recognition using cybergloves and magnetic trackers. In *Proceedings of the 2004 IEEE Region 10 Conference TENCON*, volume 1, pages 571–574. IEEE, 2004.

[106] S. Ratabouil. Tips & Tricks: Building Boost with NDK R5. [Online] `http://www.codexperiments.com/android/2011/05/tips-tricks-building-boost-with-ndk-r5/` (Besucht am 03.09.2014).

[107] A. Reichinger. Kinect Pattern Uncovered. [Online] `http://azttm.wordpress.com/2011/04/03/kinect-pattern-uncovered/` (Besucht am 03.09.2014).

[108] J. L. Reisman, P. L. Davidson und J. Y. Han. A screen-space formulation for 2D and 3D direct manipulation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST)*, pages 69–78. ACM, 2009.

[109] S. Rodriguez, A. Picon und A. Villodas. Robust vision-based hand tracking using single camera for ubiquitous 3D gesture interaction. In *Proceedings of the 2010 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 135–136. IEEE, 2010.

[110] R. E. Schapire und Y. Singer. Improved Boosting Algorithms Using Confidence-rated Predictions. *Machine learning*, 37(3):297–336, 1999.

[111] I. Schiller. MIP - MultiCameraCalibration, [Software] Version 1.0.0. [Software] Version 1.0.0 `http://www.mip.informatik.uni-kiel.de/tiki-index.php?page=Calibration` (Besucht am 03.09.2014).

[112] M. Schlattmann. *Real-Time Markerless Tracking the Human Hands for 3D Interaction*. PhD thesis, Universität Bonn, 2011.

[113] O. Schreer. *Stereoanalyse und Bildsynthese*. Springer Berlin Heidelberg, 2005.

[114] P. Soille. *Morphological Image Analysis: Principles and Applications*. Springer New York, 2nd edition, 2003.

[115] M. Stark, M. Kohler und P. G. Zyklop. Video based gesture recognition for human computer interaction. In W. D. Fellner, editor, *Modeling - Virtual Worlds - Distributed Graphics*. 1995.

[116] J. Staudemeyer. *Android Programmierung - kurz & gut*. O'Reilly Media Inc., 1st edition, 2012.

[117] E. a. Suma, B. Lange, A. S. Rizzo, D. M. Krum und M. Bolas. FAAST: The Flexible Action and Articulated Skeleton Toolkit. In *Proceedings of the 2011 IEEE Virtual Reality Conference (VR)*, pages 247–248. IEEE, 2011.

[118] S. Sural, G. Qian und S. Pramanik. Segmentation and histogram generation using the HSV color space for image retrieval. In *Proceedings of the International Conference on Image Processing (ICIP)*, volume 2, pages II–589 – II–592. IEEE, 2002.

[119] A. Tietz. Setting up Eclipse for C/C++ Development. [Online] `http://www.andre-tietz.de/setting-up-eclipse-for-c-c-development/` (Besucht am 03.09.2014).

[120] Unity Technologies. Building Plugins for Android. [Online] `http://docs.unity3d.com/Manual/PluginsForAndroid.html` (Besucht am 03.09.2014).

[121] Unity Technologies. Unity Manual. [Online] `http://docs.unity3d.com/Manual/index.html` (Besucht am 03.09.2014).

[122] Unity Technologies. Unity3D. [Software] Version 3.5.1 `http://unity3d.com/` (Besucht am 03.09.2014).

[123] M. Van den Bergh, E. Koller-Meier, F. Bosché und L. Van Gool. Haarlet-based hand gesture recognition for 3D interaction. *Workshop on Applications of Computer Vision (WACV)*, pages 1–8, 2009.

[124] M. Van den Bergh, E. Koller-Meier und L. Van Gool. Real-Time Body Pose Recognition Using 2D or 3D Haarlets. *International Journal of Computer Vision*, 83(1):72–84, 2009.

[125] M. VandenBergh und L. Van Gool. Combining RGB and ToF cameras for real-time 3D hand gesture interaction. *2011 IEEE Workshop on Application of Computer Vision (WACV)*, pages 66–72, 2011.

[126] V. Vezhnevets, A. Velizhev, A. Yakubenko, N. Chetverikov und A. Chibunitchev. GML C++ Camera Calibration Toolbox. [Software] Version 0.75 `http://graphics.cs.msu.ru/en/research/projects/3dreconstruction/cppcalibration` (Besucht am 03.09.2014).

[127] P. Viola und M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages I–511 – I–518. IEEE, 2001.

[128] R. Y. Wang und J. Popović. Real-time hand-tracking with a color glove. *ACM Transactions on Graphics*, 28(3):63:1–63:8, 2009.

[129] T. S. Washio. Kinect-MSSDK-OpenNI-Bridge. [Software] Version 1.6.0.0 `https://code.google.com/p/kinect-mssdk-openni-bridge/` (Besucht am 03.09.2014).

[130] J. Webb und J. Ashley. *Beginning Kinect Programming with the Microsoft Kinect SDK*. Apress, 2012.

[131] B. Weiss. Brillenloses 3D auf dem iPad 2 (Video). [Online] `http://de.engadget.com/2011/04/12/brillenloses-3d-auf-dem-ipad-2-video/` (Besucht am 03.09.2014).

[132] W. Westerman. *Hand Tracking, Finger Identification and Chordic Manipulation on a Multi-Touch Surface*. PhD thesis, University of Delaware, 1999.

[133] W. Westerman, J. G. Elias und A. Hedge. Multi-touch: A new tactile 2-d gesture interface for human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society (HFES) 45th Annual Meeting*, volume 45, pages 632–636. SAGE Publications, 2001.

[134] H. S. Yeo, B. G. Lee und H. Lim. Hand tracking and gesture recognition system for human-computer interaction using low-cost hardware. *Multimedia Tools and Applications*, pages 1–29, 2013.

[135] X. Zabulis, H. Baltzakis und A. Argyros. Vision-based hand gesture recognition for human-computer interaction. *The Universal Access Handbook. LEA*, 2009.

[136] L. Zhang, R. Chu, S. Xiang, S. Liao und S. Z. Li. Face Detection Based on Multi-Block LBP Representation. In *Proceedings of the 2007 International Conference on Advances in Biometrics (ICB)*, pages 11–18. Springer, 2007.

[137] Z. Zhang. A Flexible New Technique for Camera Calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 22(11):1330–1334, 2000.

[138] Y. Zhu, Z. Yang und B. Yuan. Vision Based Hand Gesture Recognition. In *Proceedings of the 2013 International Conference on Service Sciences (ICSS)*, pages 260–265. IEEE, 2013.

Abkürzungsverzeichnis

A1 Erster Ansatz zur Fingergestenerkennung 33

A1-D Erster Ansatz-Depth; 3D-Position mittels Tiefendaten 33

A2 Zweiter Ansatz zur Fingergestenerkennung 33

A2-C Zweiter Ansatz-Canny; Segmentierung mittels Canny-Kantendetektion 37

A2-D Zweiter Ansatz-Depth; Segmentierung und 3D-Position mittels Tiefendaten 33

A2-H Zweiter Ansatz-HSV; Segmentierung mittels HSV-Werten 37

A2-T Zweiter Ansatz-Threshold; Segmentierung mittels Schwellwert 37

ANMM Average Neighborhood Margin Maximization 15

Geste 1 Erste Fingergeste zur Steuerung der virtuellen Hand 33

Geste 2 Zweite Fingergeste zur Manipulation von selektierten Objekten 33

GML Graphics and Media Lab 54

GMM Gaussion Mixture Model 15

GUI Graphical User Interface 71

HCP Head-Coupled Perspective 13

HSV Hue, Saturation, Value 37

JNI Java Native Interface 66

K1 Erster Haar-Klassifikator zur Erkennung von Geste 1 36

K2 Zweiter Haar-Klassifikator zur Erkennung von Geste 2 36

K3 Dritter Haar-Klassifikator zur Erkennung der Hand 36

LBP Local Binary Patterns 12

MIP-MCC Multimedia Information Processing Group-Multi Camera Calibration 54

MIXIS Mixed Interaction Space 12

NDK Native Development Kit 64

NUI Natural User Interface 3

OpenCV Open Source Computer Vision 54

OpenNI Open Natural Interaction 64

RGB Red, Green, Blue 4

RGBD Red, Green, Blue, Depth 4

SDK Software Development Kit 64

ToF Time-of-Flight 16

VR Virtual Reality 3

WLAN Wireless Local Area Network 20