Brandenburgische Technische Universität
Cottbus-Senftenberg

Fakultät 3
Maschinenbau, Elektrotechnik und Wirtschaftsingenieurwesen

Lehrstuhl für Kommunikationstechnik

Bachelorarbeit

Personenerkennung und Positionsbestimmung mit Daten der Kinect-Kamera

Detection and positioning of persons using data from the Kinect camera

Autor: Thomas Jung
Matrikelnummer: 3134634
E-Mail: thomas-j-90@web.de

Version vom: 16. Juli 2015

Betreuer: M. Sc. Jens Lindemann
Prüfer: Prof. Dr.-Ing. habil. Matthias Wolff

Zusammenfassung

Die Interaktion mit einer Maschine fällt jedem menschlichen Akteur umso leichter, je natürlicher das Verhalten der Maschine ist. Dazu benötigt jede Maschine menschenähnliche Fähigkeiten. Das *Cognitive System Lab* beinhaltet ein multimodales System für die Mensch-Maschine-Interaktion, welches u. a. die Fähigkeit des „Hörens“ besitzt. Für die Fähigkeit des „Sehens“ verfügt das *Cognitive System Lab* über zwei Kinect-Kameras. Das Ziel dieser Bachelorarbeit ist es, mithilfe der Kinect-Kameras die Anzahl und Kopfpositionen der Personen zu ermitteln, die Farb-, Tiefen-, Infrarot- und Skelettdatenströme darzustellen und die Personen zu ermitteln, die zum Display schauen. Die gelieferten Tiefen- und Skelettdaten enthalten die Positionen und Orientierungen. Die sind relativ zu der Kinect. Das *Cognitive System Lab* besitzt ein eigenes Koordinatensystem. Aus diesem Grund müssen die relative Daten in absolute Daten umgerechnet werden. Dazu erhält jede Kinect eine absolute Position und eine absolute Orientierung in Form einer Quaternion. Die Quaternion dreht die Tiefenvektoren und die absolute Position verschiebt diese. Die Berechnung des Schnittpunktes vom Display und der Blickrichtung dient der Erkennung, ob eine Person zum Display schaut. Ein einfacher 3D-Editor dient der Einstellung der Position und Orientierung der Kinect und anderer Objekte. Ein separates Fenster zeigt die Farb-, Tiefen-, Infrarot- und Skelettdaten an.

Abstract

The behavoir of a machine is more natural for each human actor, the easier the interaction with a machine is. For that every machine needs manlike skills. The cognitive system lab contains a multi-modal system for the human-computer-interaction, which owns, among other things, the skill of hearing. For the skill of seeing the cognitive system lab has two kinects. The aim of this bachelor's thesis is to determine the number of persons and the head positions, to show color, depth, infrared and skeleton data and to determine these persons, who are looking to the display. The supplied depth and skeleton data contain the positions and orientations. They are relative to the kinect. The cognitve system lab has a own coordinate system. It is therefore necessary to convert the relative data to absolute data. For that the kinect has a absolute position and an absolute orientation as a quaternion. The quaternion rotates the depth vectors and the absolute position moves the depth vectors. The calculation of the intersection of the display and the view provides the detection, whether a person is looking to the display. A simple 3D editor provides the adjustment of the positions and orientations of the kinect and other objects. A separate window displays the color, depth , infrared and skeleton data.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

J4KSDK ..... Java for Kinect SDK

JNI ..... Java Native Interface

SDK ..... Software Development Kit

1 Einleitung

Als Microsoft die erste Kinect herausbrachte, revolutionierten sie damit den Spielemarkt. Bis dahin interagierte ein Computerspieler mit dem Spiel per Hand über einem Controller oder eine Tastatur. Mit der Kinect war es seitdem möglich, eine Spielfigur oder ähnliches mit dem eigenen Körper zu steuern. Da die Kinect Daten über Positionen einzelner Körperteile liefert, kann der Computerspieler seinen virtuellen Avatar nicht nur vorwärts und rückwärts bewegen, sondern auch seine virtuellen Arme und Beine bewegen. Die Bewegungsanimation stammt daher direkt vom Computerspieler und ist keine übliche Standardanimation wie in 3D-Computerspielen.

Neben interaktiven Computerspielen unterstützt die Kinect andere Anwendungsmöglichkeiten. Dies erkannten schon früh einige Personen, die daraufhin versucht haben, die Kinect zu „öffnen“ und so für Programmierer zugänglich zu machen. Das Hauptaugenmerk lag dabei auf den Tiefendaten, da die normale Farbkamera mit einer geringen Auflösung uninteressant war. Zu dem Zeitpunkt gab es sehr teure 3D-Kameras, die ebenfalls Tiefendaten lieferten. Da aber die Kinect ein Massenartikel war und zwischen 100 und 200 Euro gekostet hat, war die Kinect eine günstige Alternative für Hobbyprogrammierer und kleinere Forschungsprojekte. Microsoft reagierte darauf und bat ein eigenes *Software Development Kit*[1] (*SDK*) an. Damit konnte ein Programmierer schnell und einfach Kinect-Anwendungen schreiben.

Einsatz findet die Kinect beispielsweise bei der Steuerung von Robotern, die sich mithilfe der Kinect im Raum orientieren können. Wegen der geringen Tiefenauflösung von 320 x 240 Pixeln ist die Kinect weniger für Scans von kleinen Gegenständen geeignet. Ihr Einsatzgebiet sind Scans für größere Gegenstände, Räumen und Personen. Mittlerweile gibt es die Kinect der zweiten Generation, die eine höhere Auflösung besitzt.

Ziel dieser Bachelorarbeit ist es, mithilfe der Kinect den Raum vom *Cognitive System Lab* zu „beobachten“. Für diese Aufgabe erhält das System eine Schnittstelle zu den Kinect-Kameras, die sich im Raum befinden. Über diese bekommt das System Informationen über die Anzahl der sich im Raum befindenden Personen und deren absolute Kopfposition im Raum im absoluten Raumkoordinatensystem des *Cognitive System Lab*s. Darüber hinaus erhält das System alle markierten Personen, die zum Display schauen. Für die Realisierung werden geometrische Beziehungen betrachtet. Dazu zählen Grundlagen der Vektoren, das Skalarprodukt, das Kreuzprodukt, Geraden, Ebenen und orthogonale Projektionen. Weiterhin beschreiben Quaternionen die Orientierung von Objekten. Die Objekte sind u. a. Personen und die Kinect-Kameras. Für die Einstellung und Verwaltung der Positionen und Orientierungen von Objekten,

[1] In diesem Zusammenhang ist das Kinect *Software Development Kit* gemeint

wie der Kinect und des großen Displays, dient ein kleiner 3D-Editor (siehe Screenshot 22 im Anhang). Die Anwendung erhält außerdem die Funktionalität, eine angeschlossene Kinect auswählen und die Ergebnisse anzeigen zu können s(siehe Screenshot 21).

2 Grundlagen

2.1 Vergleich der Kinect-Versionen

Bisher hat Microsoft zwei Versionen seiner Kinect heraus gebracht. Die erste Version erschien für die Xbox 360 und für den Computer (siehe Abbildung 1). Die zweite Versi-

Abbildung 1: Erste Kinect-Version (Quelle: `https://en.wikipedia.org/wiki/Kinect`)

on erschien für die Xbox One und ebenfalls für den Computer (siehe Abbildung 2). Die

Abbildung 2: Zweite Kinect-Version (Quelle: `https://en.wikipedia.org/wiki/Kinect`)

kleinen Unterschiede zwischen der Xbox-Variante und der Computer-Variante werden in dieser Bachelorarbeit nicht näher betrachtet. Die Verwendung der XBox-Varianten ist auch am Computer möglich. Da aber diese Kinect-Variante einen anderen Anschluss haben, wird für die Verwendung am Computer ein Adapterkabel benötigt. Microsoft bietet von sich aus diese Kombination für die zweite Version im *Microsoftstore* als *Kinect for Windows Developer Bundle* an. Anfang April 2015 berichteten mehrere Nachrichtenseiten, dass Microsoft die zweite Kinect-Version für den Computer nicht weiter produziert. In der Tabelle 1 werden die Eigenschaften der beiden Versionen gegenüber gestellt. Die Angaben beziehen sich bei der ersten Kinect-Version auf das Kinect *SDK* 1.8 [Mica] und bei der zweiten Kinect-Version auf das Kinect *SDK* 2.0 [Micb].

Bei den Eigenschaften handelt es sich um eine kleine Auswahl. Wie man in der Tabelle sehen kann, hat Microsoft die zweite Kinect-Version deutlich verbessert. Neben der höheren Auflösung der Farb- und Tiefendaten hat die zweite Kinect-Version ein weiteres Sichtfeld als die erste Kinect-Version. Durch die höhere Auflösung der Farb- und Tiefendaten kann die Berechnung der verfolgten Personen genauer erfolgen. Das weitere

Eigenschaft	Kinect-Version 1	Kinect-Version 2
Anzahl der erkennbaren Personen	6	6
Anzahl der verfolgbaren Personen	2	6
Anzahl der Gelenkpunkte pro Person	20	25
Auflösung der Farbdaten bei 30 Hz	640 x 480	1920 x 1080
Auflösung der Tiefendaten bei 30 Hz	320 x 240	512 x 424
Infrarot- und Farbdaten gleichzeitig	nein	ja
Methode der Tiefenmessung	Structured Light	Time of Flight
Vertikale Sicht in Grad	43	60
Horizontale Sicht in Grad	57	70
Minimaler Abstand in cm	80	50
Maximaler Abstand in cm	400	450

Tabelle 1: Gegenüberstellung der Kinect-Versionen

Sichtfeld erlaubt es den Personen, näher an der Kamera zu stehen. Des Weiterem liefert die neue Kinect-Version die Infrarotdaten gleichzeitig mit den Farbdaten. Die höhere Auflösung und der zusätzliche Infrarotdatenstrom erfordern eine höhere Datenrate, daher benötigt die zweite Kinect-Version *USB* 3.0 für die Datenübertragung.

2.2 Aufbau

Beide Kinect-Versionen haben eine normale Farbbildkamera. Das Wichtigste an der Kinect ist der Infrarot-Sender und -Empfänger. Die zwei Bestandteile machen es möglich, Tiefeninformationen über den Raum zu erhalten. Die Kinect-Kameras verfolgen bei der Tiefenmessung (siehe Abschnitt 2.4) zwei verschiedene Ansätze. Zudem besitzen die beiden Kinect-Kameras ein Mikrofonfeld. Da das *Cognitve Systems Lab* ein umfangreiches Mikrofonfeld enthält, wird diese bei den Kinect-Kameras nicht benötigt. Die Abbildung 3 zeigt den Aufbau und die Anordnung der Komponenten der ersten Kinect-Version. Der Infrarotsender befindet sich in der Mitte der rechten Hälfte. Der

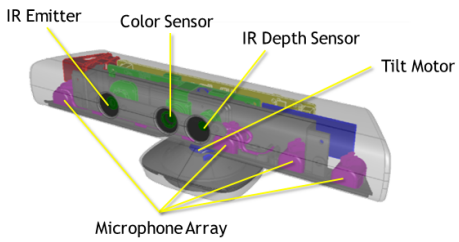

Abbildung 3: Aufbau der ersten Kinect-Version (Quelle: `https://msdn.microsoft.com/en-us/library/jj131033.aspx`)

Infrarot-Empfänger und die Farbkamera sind paarweise in der Mitte angebracht. Der Abstand zwischen Infrarot-Sender und -Empfänger beträgt ungefähr acht Zentimeter. Diese Anordnung spielt bei der Art der Tiefenmessung eine entscheidende Rolle. Bei der zweiten Kinect-Version befindet sich diese Farbkamera rechts außen. Der Infrarot-

Sender befinden sich in der Mitte. Mit einem geringeren Abstand befindet sich der Infrarot-Empfänger rechts neben dem Infrarot-Sender. Die Abbildung 4 stellt den Aufbau der zweiten Kinect-Version dar.

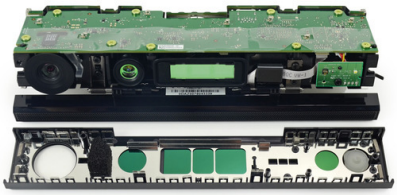

Abbildung 4: Aufbau der zweiten Kinect-Version (Quelle: `https://ifixit.com/Teardown/Xbox One Kinect Teardown/19725`)

2.3 Daten

Die Kinect-Kameras liefern verschiedene Datenströme wie Farb-, Tiefen-, Infrarot- und Skelettdatenströme. Die Skelettdaten kommen nicht direkt von der Kinect, da die Berechnung aus den Farb- und Tiefendaten auf dem PC stattfindet [Han13, S. 5].

2.3.1 Farbdaten

Die Kinect-Kameras liefern die Farbdaten im ARGB-Format. Das Format beinhaltet drei Informationen über die Grundfarben und eine über die Transparenz. Jeder Kanal ist acht Bit groß. Somit wird jedes Pixel durch 32 Bit dargestellt. Der Alpha-Kanal belegt die Bits 24 bis 31, der Rot-Kanal die Bits 16 bis 23, der Grün-Kanal die Bits acht bis 15 und der Blau-Kanal belegt die Bits null bis sieben. Die Abbildung 5 zeigt die Anordnung der vier Farbkanäle.

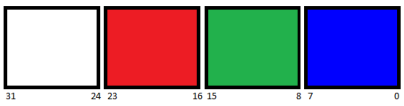

Abbildung 5: Farbdaten als 32 Bit-Darstellung

2.3.2 Tiefendaten

Der Tiefendatenstrom verfügt bei aktivierter Personenverfolgung über zwei Informationen pro Pixel. Der eine Teil beinhaltet den Abstand in Millimetern und der andere Teil den Personenindex. Die 13 höchsten Bits des 16 Bit-Datenstrom repräsentieren den Abstand. Die restlichen drei Bits stehen für die Person, welche von dem dazugehörigen

Pixel bedeckt wird. Ist die Personenverfolgung deaktiviert, so ist der Personenindex Null. Die Abbildung 6 stellt die Aufteilung der Tiefendaten dar.

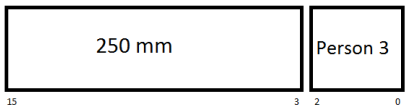

Abbildung 6: Tiefendaten als 16 Bit-Darstellung

2.3.3 Infrarotdaten

Die Infrarotdaten kommen als 16 Bit-Datenstrom. Die Daten repräsentieren ein Graustufenbild, wobei nur die höchsten zehn Bits belegt sind. Die anderen sechs Bits sind Null. Die Abbildung 7 gibt die Bit-Belegung der Infrarotdaten wieder.

Abbildung 7: Infrarotdaten als 16 Bit-Darstellung

2.3.4 Skelettdaten

Die Skelettdaten der ersten Kinect-Version beinhalten Informationen über die Positionen von 20 Gelenkpunkten. Die Abbildung 8 zeigt die 20 Gelenkpunkte des Skelettes. Die zweite Kinect-Version liefert fünf Gelenkpunkte mehr. Das Schulterzentrum wurde

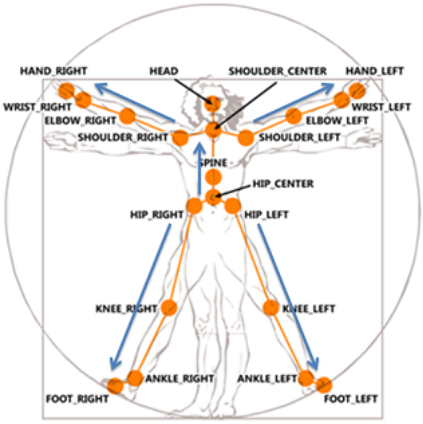

Abbildung 8: Darstellung der Gelenkpunkte der ersten Kinect-Version (Quelle: `https://msdn.microsoft.com/en-us/library/jj131025.aspx`)

in zwei Gelenkpunkte aufgeteilt. Zusätzlich hat das Skelett an den Händen jeweils einen Daumen und einen Zeigefinger. Die Abbildung 9 zeigt das Skelett mit 25 Gelenkpunkten.

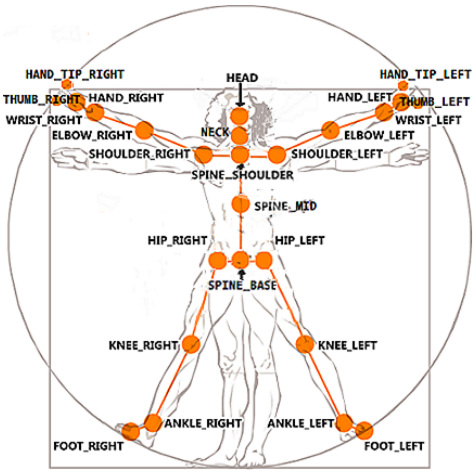

Abbildung 9: Darstellung der Gelenkpunkte der zweiten Kinect-Version (Quelle: `https://msdn.microsoft.com/en-us/library/microsoft.kinect.jointtype.aspx`)

2.4 Tiefenmessung

2.4.1 Structured Light

Die erste Kinect-Version nutzt für die Tiefenmessung die *Structured-Light*-Methode. Bei dieser Methode sendet der Infrarotsender das Infrarotlicht durch ein optisches Gitter, wodurch ein konstantes Muster entsteht [KPG14]. Objekte und Hindernisse im Raum verändern das Muster. Durch Vergleich des aktuellen Musters mit dem bekannten Muster, ist es möglich die Entfernung von Objekten zu bestimmen. Die Abbildung 10 stellt das Muster dar.

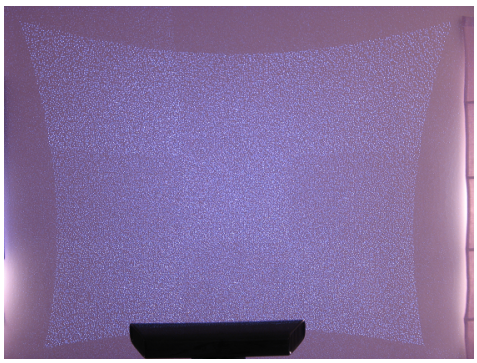

Abbildung 10: Darstellung des Infrarotmusters (Quelle: `www.futurepicture.org/?p=129`)

Das bekannte Muster entsteht bei Projektion auf einer glatten Oberfläche in einer festgelegten Entfernung z_o. Objekte in bestimmter Entfernung z_k verschieben einzelne Punkte des Musters entlang der x-Achse. Diese Verschiebung nennt man Disparität D. Diese Disparität kann die Kinect nicht direkt messen. Durch die Aufnahme des Infrarot-Empfängers entsteht eine Abbildung der Disparität im Tiefenbild. Die abgebildete Disparität d kann die Kinect messen. Bekannt sind zwei weitere Größen. Der

Abstand zwischen des Infrarot-Senders und -Empfängers wird als b und die Brennweite als f bezeichnet. Mithilfe des Strahlensatzes ergeben sich zwei Formeln.

$$D = \frac{(z_o - z_k)b}{z_o} \tag{1}$$

$$\frac{d}{f} = \frac{D}{z_k} \tag{2}$$

Durch Einsetzen der Formel 1 in die Formel 2 und durch Umstellen nach z_k erhält man die Formel 3. [Kho11]

$$z_k = \frac{1}{\frac{d}{fb} + \frac{1}{z_o}} \tag{3}$$

Dieses Verfahren arbeitet bei kleinen Objekten unzulänglich. Grund ist die Notwendigkeit der Identifizierung jedes Punktes durch seine Nachbarn [Lau]. Die unterschiedlichen Entfernungen verschieben die Punkte unterschiedlich weit entlang der x-Achse. Kleine Objekte verdrängen vereinzelte Punkte aus ihrer Nachbarschaft, wodurch die Identifizierung nicht mehr erfolgen kann.

2.4.2 Time of Flight

Die zweite Kinect-Version nutzt für die Tiefenmessung die *Time-of-Flight*-Methode. Die Methode basiert auf der physikalischen Eigenschaft, dass sich bewegende Objekte mit einer bestimmten Geschwindigkeit eine Strecke in einer bestimmten Zeit zurücklegen. Ist die Geschwindigkeit bekannt und die benötigte Zeit gemessen, so ergibt sich daraus die zurückgelegte Strecke. Wie die erste Kinect-Version nutzt auch die zweite Kinect-Version Infrarotlicht. Infrarot ist mit einem Wellenlängenbereich von 780 nm bis 1400 nm ein Teil des Lichtes und hat daher eine bekannte endliche Geschwindigkeit von rund 300 000 $\frac{km}{s}$. Zwischen Senden und Empfangen des Infrarotlichtes resultiert eine Verzögerungszeit. Mithilfe der Verzögerungszeit berechnet die Formel 4 die zurückgelegte Strecke. Da das Infrarotlicht die Strecke zweimal zurücklegt, entspricht der Abstand des Objektes der halben Strecke.

$$D = \frac{t \cdot c}{2} \tag{4}$$

Der Messbereich der Kinect beginnt bei 50 cm und endet bei 450 cm. Damit ergeben sich Verzögerungszeiten von 3,3 ns bis 30 ns. Der Tiefensensor misst aber nicht direkt die Zeit, sondern akkumuliert das ankommende Signal in zwei Speicherzellen pro Pixel, wobei ein Takt diese abwechselnd ansteuert [But14]. Dadurch teilt sich das ankommende Signal je nach Entfernung in ein bestimmtes Verhältnis auf.

Die Impulsdauer ist genauso lang wie die jeweilige Ansteuerung der zwei Speicherzellen. Aufgrund der Verzögerung hat die zweite Speicherzelle einen Anteil am Signal, d. h. der Inhalt der zweiten Speicherzelle entspricht der Verzögerung. Die Formel 5

berechnet den Abstand mit dem ermittelten Verhältnis und der bekannten Impulsdauer. [KHK+10]

$$D = \frac{c}{2} \cdot \frac{t_{Impulsdauer}}{value_{Speicherzelle1} + value_{Speicherzelle2}} \cdot value_{Speicherzelle2} \tag{5}$$

2.5 Orientierung der Gelenkpunkte

Neben den Positionsdaten beinhaltet jeder Gelenkpunkt die Orientierung eines Knochen des Skeletts in Form einer Quaternion. Das Skelett ist hierarchisch aus Gelenkpunkten und Knochen aufgebaut (siehe Abbildung 11). Das Hüftzentrum dient dabei

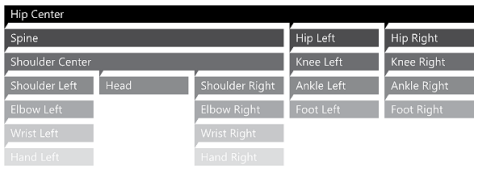

Abbildung 11: Hierarchie von Gelenkpunkten (Quelle: `https://msdn.microsoft.com/en-us/library/hh973073.aspx`)

als Anfangspunkt, von dem aus das Skelett aufgebaut ist. Alle abgehenden Gelenkpunkte werden in diesem Zusammenhang als Kinder bezeichnet. Die Kinder sind wiederum Eltern von abgehenden Gelenkpunkten. Der Kopf, die Hände und die Füße haben keine Kinder und schließen damit das Skelett ab. Die Verbindungen von Eltern und Kindern repräsentieren die Knochen des Skeletts, wobei die Kinder die Orientierung beinhalten. Z. B. beinhaltet der Kopf die Orientierung der Halswirbelsäule und das Knie die Orientierung des Oberschenkels. Das Hüftzentrum als Anfangspunkt beinhaltet die absolute Orientierung der Person im Koordinatensystem der Kinect. Die anderen Gelenkpunkten enthalten die relative Orientierung zu den Eltern. Jeder Gelenkpunkt besitzt ein lokales Koordinatensystem. Die y-Achse zeigt entlang der Längsachse des Knochens. Um die absolute Orientierung eines Knochens im Koordinatensystem der Kinect zu erhalten, multipliziert man alle Quaternionen vom Kind bis zum Hüftzentrum entlang des Skelettpfades (siehe Abbildung 11). Die zweite Kinect-Version liefert alle Orientierungen als absolute Werte, wobei der Kopf, die Füße, die Daumen und die Zeigefinger keine Orientierung haben.

2.6 Orientierung als Drehung

Eine Orientierung beschreibt die Lage eines Objektes. Eine komplexe Zahl kann eine Orientierung im zweidimensionalen Raum darstellen. Im dreidimensionalen Raum gibt es drei Darstellungsmöglichkeiten für eine Orientierung. Eine Quaternion beschreibt die Orientierung als eine einzige Drehung. Die Eulerwinkel beschreiben die Orientierung als eine dreifache Drehung um jeweils eine Achse. Eine weitere Form für Drehungen im zwei- und dreidimensionalen Raum bilden die Drehmatrizen.

2.6.1 Komplexe Zahl

Eine komplexe Zahl $\underline{z} = a + i \cdot b$ wird durch zwei reelle Zahlen a und b und eine imaginäre Einheit i dargestellt und erweitert somit den Zahlenbereich der reellen Zahlen.

Komponenten der komplexen Zahl:

- imaginäre Einheit i: $i^2 = -1$
- Realteil von $\underline{z}$: $Re(\underline{z}) = a$
- Imaginärteil von $\underline{z}$: $Im(\underline{z}) = b$

2.6.2 Drehung im zweidimensionalen Raum mit komplexen Zahlen

In der Polarform wird die komplexe Zahl mithilfe von Betrag r und Phase θ als Zeiger dargestellt. Die Berechnung des Betrages und der Phase erfolgen durch die Gleichungen 6 und 7.

$$r = \sqrt{Re^2(\underline{z}) + Im^2(\underline{z})} \tag{6}$$

$$\theta = \begin{cases} \arctan\frac{b}{a} & a > 0 \\ \arctan\frac{b}{a} + \pi & a < 0, b \geq 0 \\ \arctan\frac{b}{a} - \pi & a < 0, b < 0 \\ \frac{\pi}{2} & a = 0, b > 0 \\ -\frac{\pi}{2} & a = 0, b < 0 \end{cases} \tag{7}$$

Die Phase beschreibt die Ausgangsdrehung des Zeigers. Ist der Imaginärteil Null, so ist die Phase ebenfalls Null und der Zeiger liegt auf der realen Achse. Ist dagegen der Realteil Null, so ist die Phase um 90 Grad gedreht und der Zeiger liegt auf der imaginären Achse. Durch Addition der Phase θ mit einem weiteren Winkel ϕ wird der Zeiger gegen den Uhrzeigersinn gedreht. Die komplexe Zahl lässt sich in der Exponentialform $\underline{z} = r \cdot e^{i \cdot \theta}$ darstellen. Die Potenzgesetze erlauben eine einfache Addition der Exponenten durch Multiplikation der Potenzen mit gleicher Basis. Durch Multiplikation der komplexen Zahl $\underline{z}$ mit einer komplexen Zahl $\underline{c} = 1 \cdot e^{i \cdot \phi}$ erhält man eine komplexe Zahl $\underline{d} = r \cdot e^{i \cdot (\theta + \phi)}$, die um ϕ gedreht ist. Damit der gedrehte Zeiger gleich lang bleibt, muss der Betrag der komplexen Zahl $\underline{c}$ Eins sein.

2.6.3 Drehung im zweidimensionalen Raum mit Drehmatrizen

Für eine Drehung eines Punktes $\vec{p}$ um den Winkel α im zweidimensionalen Raum wird der Punkt mit einer 2 x 2 Drehmatrix D multipliziert.

$$p_{gedreht} = D \cdot p = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{8}$$

2.6.4 Drehung im dreidimensionalen Raum mit Drehmatrizen

Für eine Drehung eines Punktes $\vec{p}$ im dreidimensionalen Raum wird der Punkt mit einer 3 x 3 Drehmatrix D multipliziert. Dabei repräsentiert die Drehmatrix eine Drehung um eine festgelegte vierte Achse $\vec{u}$ oder eine Komposition von drei Drehungen um jeweils eine Achse des Koordinatensystems (siehe Abschnitt 2.6.8).

$$p_{gedreht} = D_u \cdot p = D_x \cdot D_y \cdot D_z \cdot p \tag{9}$$

2.6.5 Quaternionen

Eine Quaternion ist ein Quadrupel aus einem Realteil w und einem dreiteiligen Imaginärteil. Anders als bei den komplexen Zahlen, hat der Imaginärteil drei imaginäre Einheiten (i, j, k), die mit einem Vektor $\vec{v} = (x, y, z)^T$ verknüpft sind. Das Quadrupel $(w, i \cdot x, j \cdot y, k \cdot z)$ repräsentiert eine Quaternion.

Verknüpfungsregeln:

- $i^2 = j^2 = k^2 = -1$
- $i \cdot j \cdot k = -1$
- $i \cdot j = k = -j \cdot i$
- $j \cdot k = i = -k \cdot j$
- $k \cdot i = j = -i \cdot k$

2.6.6 Drehung im dreidimensionalen Raum mit Quaternionen

Für die Drehung eignen sich Einheitsquaternionen, wobei der Vektor im Imaginärteil normiert ist. Eine Einheitsquaternion lässt sich in Polarform als $(\cos\frac{\theta}{2}, \vec{v} \cdot \sin\frac{\theta}{2})$ mit $|\vec{v}| = 1$ darstellen. Der Vektor $\vec{v}$ repräsentiert eine vierte imaginäre Achse. Der Winkel θ dreht einen Punkt um die imaginäre Achse. Die Grafik 12 zeigt die Drehung des Punktes $P1$ um die Drehachse. Drehungen erfolgen über den Koordinatenursprung eines Bezugssystems. So hat jedes drehbare Objekt ein lokales Koordinatensystem, wobei jeder Punkt im Objekt durch einen Ortsvektor $\vec{p}$ erreichbar ist. Als Quaternion ist ein Ortsvektor eine reine Quaternion in der Form $p = (0, \vec{p})$. Die Drehung der Quaternion p erfolgt durch eine Multiplikation mit der Drehquaternion q und der inversen Drehquaternion q^{-1}. Die resultierende Quaternion p_{neu} ist rein und beinhaltet den gedrehten Vektor. Die Multiplikation zweier Einheitsquaternionen ergibt eine neue Orientierung in Form einer neuen Einheitsquaternion. Die Gleichung 11 zeigt die Multiplikation von Quaternion q1 mit Quaternion q2.

Besonderheiten von Einheitsquaternionen:

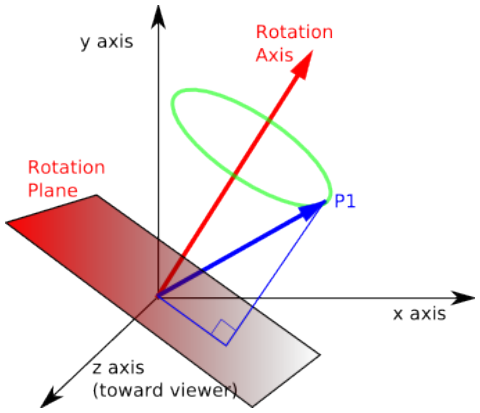

Abbildung 12: Drehachse und Drehwinkel einer Quaternion (Quelle: `http://euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/geometric/axisAngle/`)

- Konjugierte und inverse Quaternion sind gleich: $q^* = q^{-1}$
- Drehung von p durch q: $p_{neu} = q \cdot p \cdot q^{-1}$

$$q^{-1} = (w, -i \cdot x, -j \cdot y, -k \cdot z) \tag{10}$$

$$\begin{pmatrix} w_{neu} \\ x_{neu} \\ y_{neu} \\ z_{neu} \end{pmatrix} = \begin{pmatrix} w_1 \cdot w_2 - x_1 \cdot x_2 - y_1 \cdot y_2 - z_1 \cdot z_2 \\ x_1 \cdot w_2 + w_1 \cdot x_2 - z_1 \cdot y_2 + y_1 \cdot z_2 \\ y_1 \cdot w_2 + z_1 \cdot x_2 + w_1 \cdot y_2 - x_1 \cdot z_2 \\ z_1 \cdot w_2 - y_1 \cdot x_2 + x_1 \cdot y_2 + w_1 \cdot z_2 \end{pmatrix} \tag{11}$$

2.6.7 Drehlage als Quaternion zwischen zwei Vektoren bestimmen

In manchen Fällen ist es sinnvoll die Drehlage zwischen zwei Vektoren $\vec{a}$ und $\vec{b}$ zu bestimmen. Die Drehung mit einer Quaternion erfolgt um eine imaginäre Achse $\vec{v}$ um einen bestimmten Winkel θ. Für die Bestimmung der Drehlage reicht es, die imaginäre Achse und den Drehwinkel zu ermitteln. Die imaginäre Achse steht senkrecht auf der Drehebene, d. h. die Achse bildet das Kreuzprodukt zwischen den zwei Vektoren. Der Drehwinkel steckt im Skalarprodukt. Sind die Vektoren normiert, ergibt das Skalarprodukt den Kosinus des eingeschlossenen Winkels. Fünf einfache Schritte sind notwendig, um die Drehquaternion zu bestimmen.

1. Vektoren $\vec{a}$ und $\vec{b}$ normieren
2. Kreuzprodukt bilden: $\vec{v} = \vec{a} \times \vec{b}$
3. Vektor $\vec{v}$ normieren
4. Skalarprodukt bilden: $|\vec{a}||\vec{b}| \cos\theta = \cos\theta = \vec{a} \cdot \vec{b}$
5. Quaternion bilden: $(\cos\frac{\theta}{2}, \vec{v} \cdot \sin\frac{\theta}{2})$

2.6.8 Drehung im dreidimensionalen Raum mit Eulerwinkeln

Für die Drehung eines Objektes, mithilfe von den drei Eulerwinkeln, wird das Objekt dreimal hintereinander um eine bestimmte Achse gedreht. Das Objekt hat ein eigenes Koordinatensystem. Vor der ersten Drehung stimmt das Koordinatensystem des Objektes mit dem Koordinatensystem des Bezugssystems überein. Bei der ersten Drehung erfolgt die Drehung um eine der drei Koordinatenachsen des Bezugssystems. Nachfolgende Drehungen erfolgen über die Achsen des Objektes, die sich mitdrehen. Die einzelnen Drehungen sind mit Quaternionen oder Drehmatrizen möglich. Drehung um die x-Achse mit dem Winkel α:

$$D_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \tag{12}$$

$$q_x = (\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2} \cdot (1,0,0)^T) \tag{13}$$

Drehung um die y-Achse mit dem Winkel β:

$$D_y = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \tag{14}$$

$$q_y = (\cos\frac{\beta}{2}, \sin\frac{\beta}{2} \cdot (0,1,0)^T) \tag{15}$$

Drehung um die z-Achse mit dem Winkel γ:

$$D_z = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{16}$$

$$q_z = (\cos\frac{\gamma}{2}, \sin\frac{\gamma}{2} \cdot (0,0,1)^T) \tag{17}$$

Um eine vollständige Drehung zu erhalten, werden die Drehungen nacheinander miteinander multipliziert. Dabei spielt die Reihenfolge eine wichtige Rolle, da die endgültige Drehlage bei unterschiedlicher Reihenfolge verschieden ist. Als Resultat ergeben sich eine Drehmatrix und eine Drehquaternion. Die Drehung eines Vektor erfolgt wie in den Abschnitten 2.6.6 und 2.6.4 beschrieben.

2.7 Geometrie

2.7.1 Gerade

Eine Gerade ist eine unendliche Linie, wobei beliebig viele Ortsvektoren $\vec{r}$ jeden beliebigen Punkt auf der Geraden erreichen. Ein fester Ortsvektor $\vec{a}$ und ein skalierbarer Vektor $\vec{b}$ beschrieben eine Gerade in der Parameterform. Der feste Ortsvektor ist der Stützvektor der Geraden. Der skalierbare Vektor liegt auf der Geraden und erreicht jeden weiteren Punkt durch Skalarmultiplikation mit einem Skalaren s. Die Skalarmultiplikation verändert die Länge und Richtung des Vektors. Dieser Vektor heißt Richtungsvektor.

$$\vec{r}(s) = \vec{a} + s \cdot \vec{b} \tag{18}$$

2.7.2 Ebene

Eine Ebene ist eine unendliche Fläche. Wie bei der Geraden erreichen beliebig viele Ortsvektoren $\vec{r}$ jeden beliebigen Punkt in der Ebene. Da die Ebene zweidimensional ist, beschreibt ein zusätzlicher Richtungsvektor $\vec{c}$ zusammen mit dem Stützvektor und dem Richtungsvektor eine Ebene in Parameterform. Der zusätzliche Richtungsvektor ist ebenfalls skalierbar. Die zwei Richtungsvektoren sind nicht kollinear zueinander und heißen Spannvektoren. Die Skalaren u, v verändern die Länge und Richtung der Spannvektoren.

$$\vec{r}(u, v) = \vec{a} + u \cdot \vec{b} + v \cdot \vec{c} \tag{19}$$

2.7.3 Schnittpunkt von Ebene und Gerade

Schneiden sich Ebene und Gerade in einem Punkt, so gibt es einen gemeinsamen Ortsvektor, d. h. $\vec{r_1}(s) = \vec{r_2}(u, v)$. Der Schnittpunkt ist damit von s, u und v abhängig. Durch Lösen eines linearen Gleichungssystems werden die Skalaren s, u und v bestimmt. Das Gaußsche Eliminierungsverfahren löst ein lineares Gleichungssystem.

$$\begin{pmatrix} a_{x1} - a_{x2} \\ a_{y1} - a_{y2} \\ a_{z1} - a_{z2} \end{pmatrix} = s \cdot \begin{pmatrix} -b_{x1} \\ -b_{y1} \\ -b_{z1} \end{pmatrix} + u \cdot \begin{pmatrix} b_{x2} \\ b_{y2} \\ b_{z2} \end{pmatrix} + v \cdot \begin{pmatrix} c_{x2} \\ c_{y2} \\ c_{z2} \end{pmatrix} \tag{20}$$

2.7.4 Orthogonale Projektion

Bei einer orthogonalen Projektion wird ein Punkt auf einer Geraden oder Ebene so projiziert, dass der Vektor $\vec{PP_1}$ zwischen dem Originalpunkt P und seiner Projektion P_1 orthogonal zur Geraden oder Ebene ist. Das Skalarprodukt zwischen diesem Vektor

und des Richtungsvektors der Geraden bzw. der Spannvektoren der Ebene ist Null. Die Gleichung 21 berechnet den Ortsvektor zum projizierten Punkt $\vec{p_1}$.

$$\vec{p_1} = \vec{a} + \frac{(\vec{p} - \vec{a}) \cdot \vec{b}}{\vec{b} \cdot \vec{b}} \cdot \vec{b} \tag{21}$$

Der Vektor $\vec{b}$ ist der Richtungsvektor der Geraden oder einer der Spannvektoren der Ebene. Der Vektor $\vec{a}$ ist der Stützvektor.

2.7.5 Kreuzprodukt

Das Kreuzprodukt zwischen zwei Vektoren $\vec{a}$ und $\vec{b}$ bildet im dreidimensionalen Raum einen neuen Vektor $\vec{c}$, der orthogonal zu beiden Vektoren steht. Die Gleichung 22 berechnet den orthogonalen Vektor.

$$\vec{c} = \vec{a} \times \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b3 \end{pmatrix} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix} \tag{22}$$

2.7.6 Skalarprodukt

Anders als bei dem Kreuzprodukt ergibt das Skalarprodukt zwischen zwei Vektoren $\vec{a}$ und $\vec{b}$ eine Zahl und keinen neuen Vektor. Das Skalarprodukt ist nicht nur auf den dreidimensionalen Raum beschränkt. Die Zahl ist ein Produkt aus der Länge beider Vektoren und des Kosinus des eingeschlossenen Winkels α. Sind Länge und Winkel bekannt, dann berechnet die Gleichung 23 das Skalarprodukt. Sind nur die beiden Vektoren, aber nicht der Winkel bekannt, dann berechnet die Gleichung 24 das Skalarprodukt im dreidimesnionalen Raum.

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cdot \cos \alpha \tag{23}$$

$$\vec{a} \cdot \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 b_1 + a_2 b_2 + a_3 b_3 \tag{24}$$

Ist das Skalarprodukt Null, so liegen die beiden Vektoren $\vec{a}$ und $\vec{b}$ orthogonal zueinander. Liegen die Vektoren dagegen parallel zueinander, so ist der eingeschlossene Winkel Null und das Skalarprodukt ist eine Multiplikation der Längen beider Winkel.

2.8 Gaußsches Eliminierungsverfahren

Das Gaußsche Eliminierungsverfahren löst ein lineares Gleichungssystem, welches in der Form einer erweiterten Koeffizientenmatrix vorliegt, in zwei Phasen. In der ersten Phase eliminiert das Verfahren stufenweise Variablen, sodass unterhalb der Diagonale

der Koeffizientenmatrix Nullen stehen. Die Formel 25 zeigt eine erweiterte Koeffizientenmatrix in Stufenform für drei Unbekannte.

$$\left(\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a_{22} & a_{23} & b_2 \\ 0 & 0 & a_{33} & b_3 \end{array}\right) \tag{25}$$

Das lineare Gleichungssystem kennt drei Operationen, die das Gleichungssystem transformieren. Die Lösungsmenge bleibt dabei aber gleich.

Operation	Ergebnismatrix
Zeilen vertauschen	$\left(\begin{array}{cc\|c} a_{21} & a_{22} & b_2 \\ a_{11} & a_{12} & b_1 \end{array}\right)$
Zeile mit Zahl multiplizieren	$\left(\begin{array}{cc\|c} -a_{11} & -a_{12} & -b_1 \\ 3a_{21} & 3a_{22} & 3b_2 \end{array}\right)$
Zeile zu einer anderen Zeile addieren	$\left(\begin{array}{cc\|c} a_{11} & a_{12} & b_1 \\ a_{21}+a_{11} & a_{22}+a_{12} & b_2+b_1 \end{array}\right)$

Tabelle 2: Operationen beim Gaußschen Eliminierungsverfahren

Mithilfe der drei Operationen bringt die erste Phase die Matrix in Stufenform. Liegt die Matrix in Stufenform vor, bestimmt die zweite Phase die Variablen durch Rückwärtseinsetzen. In der letzten Zeile hängt die Gleichung nur von einer Variablen ab. Nach Bestimmung steht die Variable der nächst höheren Zeile zur Verfügung. Die nächst höhere Zeile hat damit nur eine Unbekannte. Der Verfahren ist fertig, wenn das Rückwärtseinsetzten die erste Zeile erreicht und die letzte Unbekannte bestimmt hat.

3 Wahl der Arbeitsumgebung

Die Kinect-Anwendung entsteht als ein Java-Modul, da sie in die Software des *Cognitve System Lab* eingebunden werden soll. Microsoft bietet für die Enwicklung von Anwendungen ein *SDK* an, welches aber nur die Sprachen c++, c# und *Visual Basic* unterstützt. Es gibt einige Java-Bibliotheken, die das *SDK* von Microsoft einbinden. Nachfolgend werden ein paar Java-Bibliotheken vorgestellt und untersucht, ob diese für die Realisierung der Anwendung geeignet sind.

3.1 Anwendungsziele

Bei der Untersuchung der Bibliotheken erfolgt die Betrachtung der beiden Kinect-Versionen. Grund ist die anfängliche Nutzung der ersten Kinect-Version, die im Laufe der Bachelorarbeit durch die zweite Kinect-Version ersetzt wurde. Die wichtigen Parameter sind die Anzahl und die Kopfpositionen der Personen. Beide Kinect-Versionen können bis zu sechs Personen erkennen. Die erste Kinect-Version kann nur zwei Personen verfolgen. Die Kopfposition ist in den Skelettdaten enthalten. Das Problem ist, dass das *SDK* Skelettdaten nur bei Verfolgung einer Person liefert. Um bis zu sechs Personen mit dem *SDK* 1.8 zu verfolgen, ist es möglich einen Kompromiss mithilfe des *SDK*s zu erreichen. Das *SDK* bietet eine Methode an, um die Verfolgung zu einer oder zwei bestimmten Personen zu wechseln. Die Idee ist, zwei Personen für eine bestimmte Zeit zu verfolgen und anschließend die Verfolgung zu zwei anderen Personen zu wechseln u. s. w.. So ist es möglich die Verfolgung durch „Zeitmultiplexen“ bis auf sechs Personen auszuweiten. Der Nachteil bei dieser Idee ist, dass keine Person mehr zeitlich vollständig verfolgt wird. Der Nachteil relativiert sich, wenn sich die Personen kaum oder langsam bewegen. Andere Anwendungen, die auf die Positionsdaten zurückgreifen, müssen entsprechend angepasst sein. Obwohl beide Kinect-Versionen bis zu sechs Personen erkennen, gibt es einen kleinen Unterschied im Verfolgungsstatus. Die Personen, die verfolgt werden, haben den Status *Tracked* und die Personen, die nur erkannt wurden, haben den Status *PositionOnly*. Beim *SDK* 2.0 haben alle sechs Personen den Status *Tracked*. Die Anwendung zählt alle Personen mit diesem Status. Bei der Verwendung des *SDK*s 1.8 muss die Anwendung zusätzlich die Personen mit dem Status *PositionOnly* zählen. Bei der Wahl der Bibliothek und bei der Verwendung des *SDK*s 1.8 und damit der ersten Kinect-Version, müssen die Bibliotheken die Methode zum Wechseln und die verschiedenen Verfolgungsstatus anbieten.

3.2 *Jnect*

Jnect [Neu] ist ein Plug-in für die Entwicklungsumgebung Eclipse. *Jnect* verbindet das *SDK* von Microsoft in Java über *JNI4NET* ein. Das Plug-in soll einem Entwickler dabei unterstützen, Eclipse mit der Kinect zu steuern. Beispielweise steuert das Plug-

in den *Debugmode* mit Handgesten und Sprachbefehlen. Ein Entwickler hat aber auch die Möglichkeit eigene Anwendungen zu schreiben, die auf die Skelettdaten und den Spracherkenner zugreifen. *Jnect* wird momentan nicht weiter entwickelt. Die letzte Änderung fand am 02.11.2012 statt. *Jnect* nutzt das alte *SDK* 1.5. Da das *SDK* 1.8 abwärtskompatibel ist, kann *Jnect* das neuere *SDK* nutzen. Probleme gibt es bei der Nutzung der zweiten Kinect-Version, da diese nicht kompatibel mit dem *SDK* 1.8 oder niedriger ist[Dre]. Für die Verwendung der ersten Kinect-Version muss das *Jnect* neben Skelettdaten auch Farb- und Tiefendaten bereitstellen. Im Quelltext 1 ist zu erkennen, wie die Farb- und Tiefendatenströme zwar aktiviert werden, jedoch bleibt das Hinzufügen des jeweiligen *Eventhandler*s zu dem jeweiligen Datenstrom aus. Nur der Skelettdatenstrom löst eine Event-Behandlung aus.

```
1  public string setUpAndRun()
2  {
3          kinectSensor = KinectSensor.KinectSensors[0];
4
5          try
6          {
7                  kinectSensor.ColorStream.Enable();
8                  kinectSensor.DepthStream.Enable();
9                  kinectSensor.SkeletonStream.Enable();
10                 kinectSensor.Start();
11         }
12         catch (InvalidOperationException)
13         {
14                 return "Runtime initialization failed. Please make sure
                       Kinect device is plugged in.";
15         }
16
17         kinectSensor.SkeletonFrameReady += new EventHandler<
               SkeletonFrameReadyEventArgs>(nui_SkeletonFrameReady);
18
19         return "Setup Done!";
20
21 }
```

Listing 1: *Jnect* — Aktivierung der Datenströme und Verknüpfung mit dem *Eventhandler*

Innerhalb der Event-Behandlung legt das *Jnect* die Skelettdaten in einer XML-Datei ab, wobei es nur die verfolgten Personen speichert. Weiterhin bietet *Jnect* keine Möglichkeit an, die Verfolgung zwischen Personen zu wechseln. Durch ein paar Veränderungen am Quelltext wäre *Jnect* als Java-Bibliothek verwendbar. Aufgrund der Tatsache, dass *Jnect* veraltet und keine reine Java-Bibliothek ist, sind Anpassungen notwendig. Daher wird von der Verwendung von *Jnect* in der Bachelorarbeit abgesehen.

```
1  private void nui_SkeletonFrameReady(object sender,
       SkeletonFrameReadyEventArgs e)
2  {
3  ...
4          foreach (Skeleton skeleton in skeletons)
5          {
6
7          if (skeleton.TrackingState == SkeletonTrackingState.Tracked)
8          {
9          XmlNode skeletonData = doc.CreateElement("skeletonData");
10         root.AppendChild(skeletonData);
11
12         XmlNode trackingId = doc.CreateElement("trackingId");
13         trackingId.InnerText = skeleton.TrackingId.ToString();
14         skeletonData.AppendChild(trackingId);
15 ...
```

Listing 2: *Jnect* — innerhalb der Event-Behandlung

3.3 *Kinect for Java*

Kinect for Java [Joh] ist eine kleine Java-Bibliothek und im Gegensatz zu *Jnect* kein Plug-in für Eclipse. Die Einbindung des *SDK*s von Mikrosoft erfolgt über *Java Native Interface* (*JNI*). Die Bibliothek stellt alle Positionsdaten über jeden Gelenkpunkt und alle Verfolgungsstatus bereit.

```
1  JNIEXPORT jfloat JNICALL
       Java_kinect_skeleton_Joint_getJointPositionByIndex(JNIEnv *env, jclass
        cls,jint SkeletonID, jint JointID, jint PositionIndex){
2          UNREFERENCED_PARAMETER( env );
3          UNREFERENCED_PARAMETER( cls );
4          if(PositionIndex == 0) return Kinect::skeleton_frame.SkeletonData
               [SkeletonID-1].SkeletonPositions[JointID].x;
5          if(PositionIndex == 1) return Kinect::skeleton_frame.SkeletonData
               [SkeletonID-1].SkeletonPositions[JointID].y;
6          if(PositionIndex == 2) return Kinect::skeleton_frame.SkeletonData
               [SkeletonID-1].SkeletonPositions[JointID].z;
7          return 0;
8
9  JNIEXPORT jint JNICALL
       Java_kinect_skeleton_Skeleton_getSkeletonTrackingState(JNIEnv *env,
       jclass cls,jint SkeletonID){
10         UNREFERENCED_PARAMETER( env );
11         UNREFERENCED_PARAMETER( cls );
12         return Kinect::skeleton_frame.SkeletonData[SkeletonID-1].
               eTrackingState;
13 }
```

```
14 }
```

Listing 3: *Kinect for Java* — Native Methoden zum Holen von Positionsdaten und Verfolgungsstatus

Die Bibliothek bietet keine Methode an, die Verfolgung zu wechseln. Es besteht jedoch die Möglichkeit den Quelltext zu ändern und eine Methode hinzuzufügen. Zuerst wird die Methode *NuiSkeletonTrackingEnable* angepasst. Diese Methode hat zwei Übergabe-Parameter. Der erste Parameter ist ein *Eventhandler*, der die Skelettdaten enthält. Der zweite Parameter ist ein *Flag*. Es gibt in diesem Zusammenhang drei *Flag*s. Ist der zweite Parameter Null, dann nutzt das *SDK* seine Standardeinstellung. Diese beinhaltet die Auswahlroutine, wonach die ersten beiden erfassten Personen verfolgt werden. Für die Aktivierung einer eigenen Auswahlroutine erhält die Methode den Wert Zwei als zweiten Parameter.

```
1 // Kinect.hpp Zeile 79
2 hr = NuiSkeletonTrackingEnable( m_hNextSkeletonEvent, 2 );
3
4 // in Main.cpp
5 JNIEXPORT void JNICALL Java_kinect_skeleton_Skeleton_setTrackedSkeletons(
      JNIEnv *env, jclass cls,jint SkeletonID1, jint SkeletonID1){
6         UNREFERENCED_PARAMETER( env );
7         UNREFERENCED_PARAMETER( cls );
8         SetTrackedSkeletons(SkeletonID1-1, SkeletonID2-1);
```

Listing 4: *Kinect for Java* — Anpassung des Quelltextes in c++

```
1 // in kinect/skeleton/Skeleton.java
2 public final static native void setTrackedSkeletons(int SkeletonID1, int
      SkeletonID2);
```

Listing 5: *Kinect for Java* — Anpassung des Quelltextes in Java

Die Klasse *Main* bekommt eine neue Methode, die das Wechseln veranlasst. Die Java-Klasse bekommt eine native Schnittstelle. Die Auswahlroutine wird in Java geschrieben und ruft nach der Auswahl die native Schnittstelle auf und übergibt die ausgewählten Skelettindizes. Vorerst scheint die Bibliothek als Basis für eine eigene Java-Bibliothek geeignet zu sein. Nachteilig ist aber die notwendige Pflege. Besser wäre eine etablierte Java-Bibliothek, die weiterentwickelt wird (*Kinect for Java* befindet sich aktuell nicht in der Weiterentwicklung). Die letzte Änderung ist vom 27.06.2012. Aufgrund des Datums ist anzunehmen, dass die Bibliothek auf eine ältere *SDK*-Version setzt. Für die Verwendung der zweiten Kinect-Version, muss der native Teil der Bibliothek an das *SDK* 2.0 angepasst werden.

3.4 Java-*Wrapper* für *OpenNI* und *NiTE*

OpenNI ist eine gemeinnützige Organisation, die von PrimeSense gegründet wurde. Die Organisation hat ein offenes *SDK* entwickelt, dass neben anderen Geräten auch die Kinect unterstützt. Ende 2013 kaufte Apple das israelische Unternehmen PrimeSense auf. Die Internetseite von *OpenNI* wurde 2014 geschlossen. Der Quelltext der letzten Version von *OpenNI* ist noch erhältlich und befindet sich auf *Github* [opea]. Das Unternehmen Occipital, welches eigene Sensoren herstellt, entwickelt das *SDK* weiter [Occ]. Das *SDK* liegt in der Version 2.2.0.33[2] vor. Der Quelltext ist offen und befindet sich auf *Github* [opeb]. Das *SDK* ist in der Sprache c++ geschrieben und hat zusätzlich einen Java-*Wrapper*. Damit lassen sich Java-Anwendungen schreiben. *OpenNI* führt keine Skelettverfolgung durch, bietet aber eine Schnittstelle für Erweiterungen an. Erweiterungen bauen auf die Daten auf, die *OpenNI* liefert. Eine Erweiterung ist die *Middleware NiTE*. *NiTE* berechnet aus den Daten Skelettdaten und erkennt Gesten, ist aber nicht frei verfügbar [deb]. Beim Testen von *OpenNI* und *NiTE* stellte sich heraus, dass es keine Beschränkung für die Anzahl von verfolgten Personen gibt. Recherchen ergaben keine genaue Zahl. Für die Aufgabe, die Anzahl der Personen im Raum zu ermitteln, wäre *OpenNI* und *NiTE* mit dem Java-*Wrapper* daher besser geeignet als das *SDK* 1.8 von Microsoft. Das Problem ist, dass das Unternehmen PrimeSense nach der Übernahme von Apple nicht mehr existiert und damit keine Aktualisierungen von *NiTE* herausbringen kann. Aus diesem Grund ist die Nutzung von *OpenNI* und *NiTE* keine Option.

3.5 *TCP/IP*-Verbindung

Eine andere Möglichkeit wäre es, zwei Anwendungen zu schreiben, die über *TCP/IP* kommunizieren. Dabei übernimmt eine Anwendung die Rolle des Servers und die Andere die Rolle des Clients. Der Server wird nativ mit dem *SDK* von Microsoft programmiert, der die Daten an den Client überträgt. Der Client ist in Java geschrieben und stellt die empfangenen Daten geeignet dar. Bei Verwendung der ersten Kinect-Version kann die Auswahlroutine innerhalb des Servers implementiert sein, sodass der Server die Verfolgung der Personen verwaltet und die aktuellen Skelettdaten an den Client sendet. Zwischen Client und Server muss es für den Verfolgungswechsel keine Schnittstelle geben. Die Verteilung der Aufgaben auf den Server und dem Client kann unterschiedlich realisiert werden. Die Einstellmöglichkeiten der Kinect-Version kann entweder auf der Seite des Servers geschehen oder über den Client. Soll der Client die Möglichkeit haben, die Auflösung beispielsweise ändern zu können, so braucht der Client neben dem Datenkanal zusätzlich einen Kommunikationskanal. Die Verwaltung der Kommunikation macht die *TCP/IP*-Anwendung anspruchsvoller. Einfacher wäre es, alle Einstellmöglichkeiten auf Seiten des Servers zu machen und nur die Daten an

[2] Zum Zeitpunkt des Druckes: 17.07.2015

den Client zu senden. In der Abbildung 13 ist ein rudimentäres Protokoll der Anwendung dargestellt. In diesem Protokoll ist die Kommunikation zwischen Client und

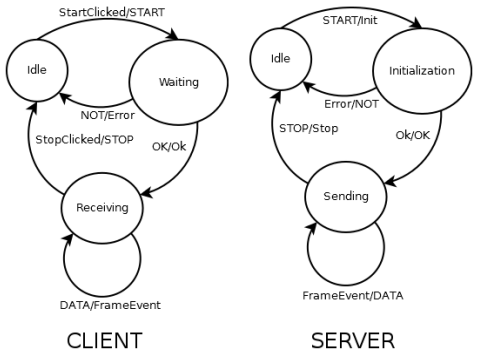

Abbildung 13: Zustandsdiagramme der *TCP/IP*-Anwendung

Server auf ein Minimum reduziert. Die Kommunikation erfolgt über die vier Dienstprimitiven *Request*, *Response*, *Confirmation* und *Indication*. Die *TCP/IP*-Anwendung stellt dem Client und Server jeweils drei verschiedene *Request*s und zwei *Response*s zur Verfügung. Ein *START-Request* löst beim Server die Initialisierung der Kinect aus. Ist die Kinect funktionsbereit, sendet der Server ein *OK-Response*, andernfalls ein *NOT-Response*. Bei erfolgreicher Initialisierung ist der Server bereit Daten zu senden und der Client bereit Daten zu empfangen. Nach der Initialisierung liefert die Kinect permanent Daten, die beim Server ein *FrameEvent* auslöst. Innerhalb der Event-Behandlung kann der Server die empfangenen Daten weiter an den Client als *DATA-Request* senden. Das Eintreffen der Daten beim Client löst seitens Java ein *FrameEvent* aus. Eine Sitzung könnte wie in der Abbilung 14 aussehen. Der Nutzer möchte die Kinect starten und drückt auf den Startknopf. Die erste Initialisierung der Kinect funkioniert nicht, weil die Kinect beispielsweise nicht angeschlossen ist. Der Nutzer erfährt durch eine negative *Error-Confirmation* davon. Nach dem Anschließen startet der Nutzer die Kinect erneut. Die Initialisierung der Kinect ist erfolgreich. Der Nutzer erhält nun solange Daten, bis er die Kinect wieder stoppt.

Dieser Ansatz ist für beide Kinect-Versionen sehr gut geeignet. Nachteil ist wie bei dem *JNI*-Ansatz die höhere Pflegebedürftigkeit. Bis jetzt scheint dieser Ansatz sehr vielversprechend.

3.6 *Java for Kinect SDK* (*J4KSDK*)

Die letzte Java-Bibliothek, die vorgestellt wird, ist das *J4KSDK* [Bar13] [DWI]. Diese Bibliothek wurde von Professor Angelos Barmpoutis an der Digital Worlds Institute University entwickelt und von Studenten erweitert. Die Bibliothek nutzt das *SDK* von Microsoft und bindet es über *JNI* in Java ein. Die Bibliothek unterstützt dabei beide Kinect-Versionen. Dafür stehen der Bibliothek zwei native Bibliotheken zur Verfügung.

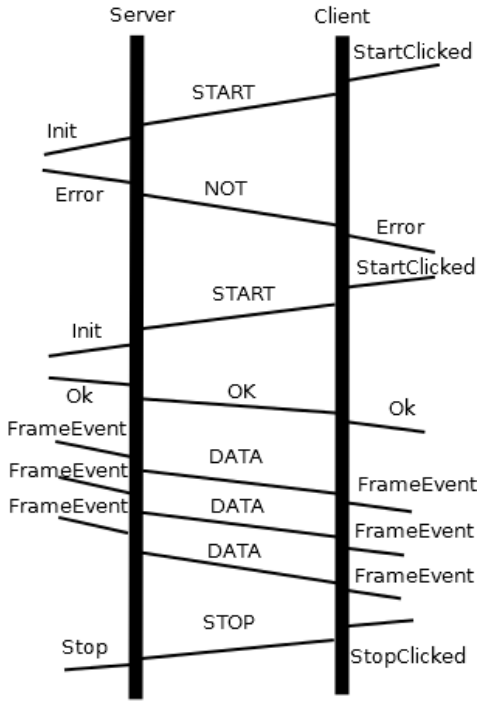

Abbildung 14: Zeitablaufdiagramm der *TCP/IP*-Anwendung

Die Java-Bibliothek verknüpft die jeweilige native Bibliothek mit der angeschlossenen Kinect. *J4KSDK* stellt für die Programmierung vier wichtige Klassen zur Verfügung. Die abstrakte Klasse *J4KSDK* dient als Schnittstelle zur Kinect. Zur Nutzung erbt eine weitere Klasse von dieser und implementiert die abstrakten Methoden.

```
1  @Override
2  public void onColorFrameEvent(byte[] color)
3  {
4  }
5
6  @Override
7  public void onDepthFrameEvent(short[] depth, byte[] player_index, float[]
       xyz, float[] uv)
8  {
9  }
10
11 @Override
12 public void onSkeletonFrameEvent(boolean[] flags, float[] positions,
       float[] orientations, byte[] states)
13 {
14 }
```

Listing 6: *J4KSDK* — Schnittstelle zur Kinect

Diese drei Methoden werden aufgerufen, wenn ein *FrameEvent* von der Kinect ausgelöst wird. Die Verarbeitung der Daten erfolgt innerhalb dieser Methoden. Die abstrakte Klasse *J4KSDK* beinhaltet zwei Methoden. Die *Start*-Methode erhält als Parameter

die zu aktivierenden Datenströme und aktiviert die Kinect. Die *Stop*-Methode stoppt alle Datenströme.

```
1 kinect.start(J4KSDK.COLOR|J4KSDK.DEPTH|J4KSDK.SKELETON);
```

Listing 7: *J4KSDK* — Starten der Kinect und Aktivierung der Datenströme

Das *JNI* übermittelt nur primitive Datentypen, was bei den puren Farb- und Tiefendaten kein Problem darstellt (siehe Abschnitt 2.3.1 und Abschnitt 2.3.2). Die Skelettdaten können nicht als Objekte übertragen werden. Daher werden diese Daten in mehrere *Array*s zerlegt und einzeln übermittelt. Auf der Java-Seite setzt das *J4KSDK* die Daten zu *Skeleton*-Objekte wieder zusammen. Dafür hat die Java-Bibliothek die Klasse *Skeleton*, die die Positionsdaten u. s. w. verwaltet. Die Klassen *DepthMap* und *VideoFrame* dienen der Darstellung der Farb- und Tiefendaten. Für die Verwendung dieser Klassen ist *JOGL* notwendig. *JOGL* greift auf die Programmierschnittestelle *OpenGL* zurück und bietet die Möglichkeit mit Java 3D-Anwendungen zu schreiben. Die Klasse *DepthMap* zeichnet mithilfe von *JOGL* für jedes Pixel ein Viereck. Jedes Pixel hat eine Position im dreidimensionalen Raum. Dadurch entsteht eine 3D-Abbildung der aufgenommen Szene. Die Klasse *VideoFrame* zeichnet ein einziges Viereck und fügt das Farbbild als Textur hinzu. Das Farbbild kann auch über die 3D-Szene als Textur gelegt werden.

Der Quelltext der Java-Bibliothek steht frei zur Verfügung. Das Lesen des Quelltextes ergab, dass das *J4KSDK* keine Möglichkeit bietet, die Verfolgung der Personen zu wechseln. Die Skelettdaten haben nur den Status *Tracked* und damit lassen sich nur zwei Personen zählen. Der Quelltext der nativen Bibliothek steht nicht frei zur Verfügung, d. h. eine Anpassung im Quelltext ist nicht ohne Weiteres möglich. Die *eMail*-Korrespondenz mit Professor Angelos Barmpoutis ergab, dass das *J4KSDK* nur die Standardauswahlroutine der Verfolgung nutzt und es für seine Projekte keine große Rolle spielt, da sie nur noch die zweite Kinect-Version verwenden [Bar]. Anders als bei *Jnect* und *Kinect for Java* wird die Bibliothek weiterentwickelt. Momentan liegt das *J4KSDK* in der Version 2.0 vor. Für zukünftige Projekte wäre diese Java-Bibliothek sehr gut geeignet.

3.7 Wahl

In einem Gespräch mit Professor Wolff wurden die Vor- und Nachteile erläutert. Daraus resultierte der Entschluss, dass sich die Probleme durch den Erwerb der zweiten Kinect-Version lösen lassen. Der Erwerb der zweiten Kinect-Version war längst geplant. Die Beschaffung wurde daraufhin zeitlich vorgezogen. Die zweite Kinect-Version befindet sich nun im *Cognitve System Lab.* Mit der Verwendung der zweiten Kinect-Version erhält die Java-Bibliothek *J4KSDK* Vorzug vor den anderen Varianten.

4 Umrechnung der Koordinatensysteme

4.1 Koordinatensystem der Kinect

Für das *Cognitive System Lab* existiert bereits ein entwickeltes Raumkoordinatensystem. Dieses Koordinatensystem ist wie bei der Kinect ein rechtshändiges Koordinatensystem. Die Abbildung 15 zeigt das Koordinatensystem der Kinect. Der Ursprung

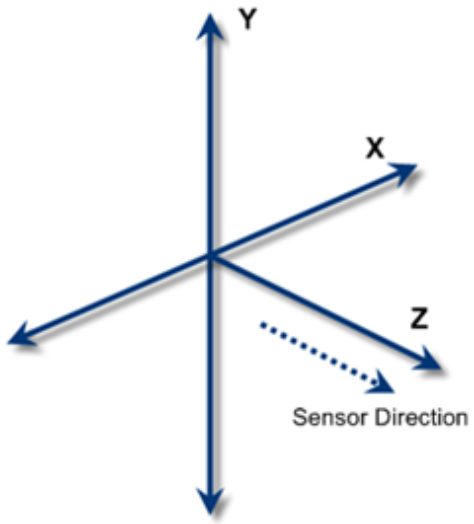

Abbildung 15: Koordinatensystem der Kinect (Quelle: `https://msdn.microsoft.com/en-us/library/hh973078.aspx`)

befindet sich im Tiefensensor. Die y-Achse zeigt nach oben, die z-Achse entlang der Blickrichtung der Kinect und die x-Achse nach links. Der Ursprung des Koordinatensystem des *Cognitive System Lab*s befindet sich in der Mitte des Raumes am Boden. Die z-Achse zeigt nach oben, die y-Achse zum Display und die x-Achse zum Fenster. Da sich die zweite Kinect-Version momentan auf einem Stativ befindet und sich frei im Labor positionieren lässt, erfolgt die Umrechnung universal. Die Positionsdaten, die die Kinect liefert, sind relativ zur der Kinect. Die Abbildung 16 zeigt die physikalische und logische Lage der Kinect und eines Objektes (Quadrat). Die rosa Darstellung steht für

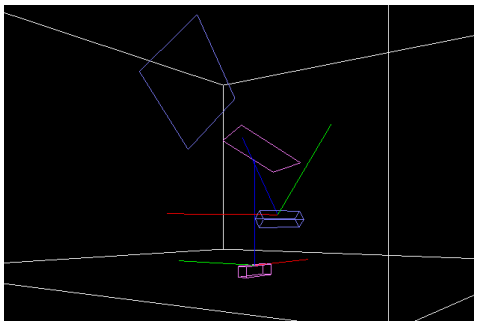

Abbildung 16: Physikalische und logische Lage der Kinect

die logische Lage der Kinect und die hellblaue Darstellung für die physikalische Lage.

Ohne Umrechnung befindet sich die Kinect logisch unterhalb des Bodens und das Objekt an der falschen Position im Raumkoordinatensystem des *Cognitive System Lab*s. Die gelieferten Positionsdaten bilden zusammen mit der Kinect ein Objekt, wobei das Koordinatensystem der Kinect das lokale Koordinatensystem des Objektes ist. Die gelieferten Positionsdaten sind die Ortsvektoren und bilden die Ecken des Objektes. Durch Verschieben und Drehen der logischen Kinect verschieben und drehen sich die Positionsdaten mit. Für die Umrechnung wird die logische Kinect an die Position der physikalischen Kinect verschoben und wie die physikalische Kinect gedreht.

4.2 Realisierung der Umrechnung

Objekte der Klasse *Object3D* repräsentieren Objekte im *Cognitve System Lab* bezüglich der Position und Orientierung. Jedes Objekt hat eine relative und absolute Position und eine relative und absolute Orientierung. In manchen Fällen stehen die Objekte in Eltern-Kind-Beziehung zueinander, d. h. die Kindobjekte haben eine relative Position und Orientierung zu einem Elternobjekt. Das ist der Fall bei der Kinect und den Gelenkpunkten. Die Umrechnung der relativen Position des Kind-Objektes erfolgt durch Addition der absoluten Position des Eltern-Objektes zu der relativen Position des Kind-Objektes. Dabei dreht sich die relative Position des Kind-Objektes mit dem Eltern-Objekt mit. Das Listing 8 zeigt die Umrechnung der relativen Position in eine absolute Position.

```
1  Vector3f relativePosition;
2  Vector3f absolutePosition;
3  ...
4  public void calculateAbsolutePosition(Object3D parent)
5  {
6    if (parent == null)
7    {
8      absolutePosition = relativePosition;
9      return;
10   }
11   Vector3f tmp = parent.getAbsoluteOrientation().rotateVector(
         relativePosition);
12   absolutePosition.add(tmp, parent.getAbsolutePosition());
13 }
```

Listing 8: Umrechnung relative Position in absolute Position

Ist die relative Position schon die absolute Position, so steht das Kind-Objekt in keiner Eltern-Kind-Beziehung und erhält als Eltern-Objekt eine Null-Referenz.

Die Umrechnung der relativen Orientierung in eine absolute Orientierung erfolgt durch Multiplikation der absoluten Orientierung des Eltern-Objektes mit der relativen Orien-

tierung des Kind-Objektes (siehe Abschnitt 2.6.6). Das Listing 9 zeigt die Umrechnung der relativen Orientierung in eine absolute Orientierung.

```
1  Quaternion relativeOrientation;
2  Quaternion absoluteOrientation;
3  ...
4  public void calculateAbsoluteOrientation(Object3D parent)
5  {
6    if (parent == null)
7    {
8      absoluteOrientation = relativeOrientation;
9      return;
10   }
11   absoluteOrientation = parent.getAbsoluteOrientation().mul(
         relativeOrientation);
12 }
```

Listing 9: Umrechnung relative Orientierung in absolute Orientierung

Quaternionen repräsentieren die Orientierungen der Objekte (siehe Abschnitt 2.6.5).

Das Listing 10 zeigt beispielsweise die Umwandlung der Position und Orientierung eines Körperteils. Das Körperteil repräsentiert die rechte Hand einer Person.

```
1  // Body
2  ...
3  Bodypart hand_right; // extends Object3D
4  ...
5  public void calculateAbsolute(Object3D camera)
6  {
7    hand_right.calculateAbsolutePosition(camera);
8    hand_right.calculateAbsoluteOrientation(camera);
9  }
```

Listing 10: Umrechnung der Position und Orientierung der rechten Hand

5 Darstellung der Daten

Wie in Abschnitt 3.6 beschrieben, stellt das *J4KSDK* Mittel für die Darstellung zur Verfügung.

5.1 Darstellung des Farbbildes

Für die Darstellung des Farbbildes nutzt die Anwendung die Klasse *VideoFrame* aus dem *J4KSDK*. Die Kinect liefert in bestimmten Zeitabständen ein Farbbild. Das *J4KSDK* ruft bei jedem neuen Farbbild die Methode *onColorFrameEvent* aus dem Listing 11 auf und übergibt dabei die Daten in Form eines *Byte-Arrays*. Die Farbdaten liegen geordnet in Vierergruppen vor. Die Reihenfolge ist folgende: Blau-, Grün-, Rot- und Alphakanal.

```
1 VideoFrame colorVideoFrame = new VideoFrame();
2 ...
3 @Override
4 public void onColorFrameEvent(byte[] color)
5 {
6   colorVideoFrame.update(getColorWidth(), getColorHeight(), color);
7 }
```

Listing 11: Verknüpfung der Klasse *VideoFrame* mit den Farbdaten

Die Methode *update* aktualisiert die Klasse *VideoFrame*, die die Breite, die Länge und die Daten aufbewahrt. Für das Zeichnen des Farbbildes hat das *J4KSDK* eine weitere Klasse namens *OpenGLPanel*, dass *OpenGL* zum Zeichnen, bereitstellt. Zum Zeichnen enthält die Klasse eine abstrakte Methode *draw*. Zur Implementierung dieser Methode erbt eine weitere Klasse von *OpenGLPanel*. *OpenGL* zeichnet Texturen. Daher bindet die Klasse *VideoFrame* die Daten an einer Textur mithilfe der Methode *use*. Die Methode *image* zeichnet ein Quadrat, worauf die Textur gezeichnet wird. Das Listing 12 zeigt die einzelnen Schritte.

```
1 @Override
2 public void draw()
3 {
4   gl.glDisable(GL2.GL_LIGHTING);
5   gl.glEnable(GL2.GL_TEXTURE_2D);
6   colorVideoFrame.use(gl);
7   translate(0, 0, -2.2f);
8   rotateZ(180);
9   image(width / height, 2);
10  gl.glDisable(GL2.GL_TEXTURE_2D);
11 }
```

Listing 12: Zeichnen eines Bildes mithilfe von *OpenGL*

Damit das Bild in den richtigen Farben erscheint, ist das *Lighting* in *OpenGL* ausgeschaltet.

Eine Alternative zum Darstellen des Bildes wäre die Verwendung von *BufferedImage*. Bei der Verwendung des *BufferedImages* für ein Bild der Größe 1920 x 1080 tritt ein Problem auf. Der Aufbau des Bildes wie im Listing 13 dargestellt, verlangsamt die Bildwiederholungsrate auf rund sieben Bilder pro Sekunde.

```
1  image = new BufferedImage(getColorWidth(), getColorHeight(),
       BufferedImage.TYPE_3BYTE_BGR);
2  int rgb = 0;
3  for(int i = 0; i < getColorWidth(); i++)
4  {
5    for(int j = 0; j < getColorHeight(); j++)
6    {
7      int offset = (getColorWidth() * j + i) * 4;
8          rgb = (color[offset + 2] & 0xff) << 16 |
9                    (color[offset + 1] & 0xff) << 8  |
10                   (color[offset + 0] & 0xff);
11
12         image.setRGB(i, j, rgb);
13   }
14 }
```

Listing 13: Aufbau eines *BufferedImage*s

Bei Bildgrößen unter 640 x 480 erreicht die Bildwiederholungsrate rund 30 Bilder pro Sekunde. Da die *Bytes* in Java vorzeichenbehaftet sind, ist es notwendig, die entstandenen *Integers* zu maskieren. Die *Leftshift*-Operatoren bringen die Farbinformationen an die richtige Stelle (siehe Abschnitt 2.3.1).

5.2 Darstellung des Tiefenbildes

Die Darstellung des Tiefenbildes weicht kaum von der Darstellung des Farbbildes ab. Da die Tiefendaten aber in einem anderen Format als die Farbdaten vorliegen, ist es notwendig, die Tiefendaten in das passende Format umzuwandeln. Die Tiefendaten befinden sich in einem *Short-Array*. Es gibt mehrere Möglichkeiten, um das Tiefenbild darzustellen. Das Listing 14 zeigt die Umwandlung der Tiefendaten in ein Graustufenbild.

```
1  VideoFrame depthVideoFrame = new VideoFrame();
2  ...
3  @Override
4  public void onDepthFrameEvent(short[] depth)
5  {
6    int sz = getDepthWidth() * getDepthHeight();
7    byte bgra[] = new byte[sz * 4];
```

```
8    int idx = 0;
9    int iv = 0;
10   for (int i = 0; i < sz; i++)
11   {
12     iv = depth[i] >> 3;
13     bgra[idx++] = (byte) iv;
14     bgra[idx++] = (byte) iv;
15     bgra[idx++] = (byte) iv;
16     bgra[idx++] = 0;
17   }
18     depthVideoFrame.update(getDepthWidth(), getDepthHeight(), bgra);
19 }
```

Listing 14: Erzeugung eines Graustufenbildes aus Tiefendaten

Die Tiefendaten sind 13 und die Farbkanäle nur acht Bit groß (siehe Abschnitt 2.3.2). Dadurch wiederholen sich die Graustufen alle 255 mm. Besser wäre es, ein farbiges Bild zu erzeugen. So stehen dem Tiefenbild 24 Bit zur Verfügung. Die Regenbogenfarben eignen sich gut für die Darstellung, da die Übergänge fließend sind. Das Listing 15 zeigt eine Umwandlung der Tiefendaten in ein Farbbild.

```
1  if(iv >= 0 && iv < 500){}
2  else if(iv >= 500 && iv < 700){blue = (byte) 0; green = (byte) 0; red = (
       byte) 255;}
3  else if(iv >= 700 && iv < 900){blue = (byte) 0; green = (byte) 51; red =
       (byte) 255;}
4  else if(iv >= 900 && iv < 1100){blue = (byte) 0; green = (byte) 102; red
       = (byte) 255;}
5  else if(iv >= 1100 && iv < 1300){blue = (byte) 0; green = (byte) 153; red
        = (byte) 255;}
6  else if(iv >= 1300 && iv < 1500){blue = (byte) 0; green = (byte) 204; red
        = (byte) 255;}
7  else if(iv >= 1500 && iv < 1700){blue = (byte) 0; green = (byte) 255; red
        = (byte) 255;}
8  else if(iv >= 1700 && iv < 1900){blue = (byte) 0; green = (byte) 255; red
        = (byte) 204;}
9  else if(iv >= 1900 && iv < 2100){blue = (byte) 0; green = (byte) 255; red
        = (byte) 153;}
10 else if(iv >= 2100 && iv < 2300){blue = (byte) 0; green = (byte) 255; red
        = (byte) 102;}
11 else if(iv >= 2300 && iv < 2500){blue = (byte) 0; green = (byte) 255; red
        = (byte) 51;}
12 else if(iv >= 2500 && iv < 2700){blue = (byte) 0; green = (byte) 255; red
        = (byte) 0;}
13 else if(iv >= 2700 && iv < 2900){blue = (byte) 51; green = (byte) 255;
       red = (byte) 0;}
14 else if(iv >= 2900 && iv < 3100){blue = (byte) 102; green = (byte) 255;
       red = (byte) 0;}
```

```
15 else if(iv >= 3100 && iv < 3300){blue = (byte) 153; green = (byte) 255;
       red = (byte) 0;}
16 else if(iv >= 3300 && iv < 3500){blue = (byte) 204; green = (byte) 255;
       red = (byte) 0;}
17 else if(iv >= 3500 && iv < 3700){blue = (byte) 255; green = (byte) 255;
       red = (byte) 0;}
18 else if(iv >= 3700 && iv < 3900){blue = (byte) 255; green = (byte) 204;
       red = (byte) 0;}
19 else if(iv >= 3900 && iv < 4100){blue = (byte) 255; green = (byte) 153;
       red = (byte) 0;}
20 else if(iv >= 4100 && iv < 4300){blue = (byte) 255; green = (byte) 102;
       red = (byte) 0;}
21 else if(iv >= 4300 && iv < 4500){blue = (byte) 255; green = (byte) 51;
       red = (byte) 0;}
22
23 bgra[idx++] = blue;
24 bgra[idx++] = green;
25 bgra[idx++] = red;
26 bgra[idx++] = 0;
```

Listing 15: Erzeugung eines Farbbildes aus Tiefendaten

Der Regenbogen beginnt mit der roten Farbe. Es mischt sich mehr grün dazu. Ist der Grünanteil maximal, nimmt der Rotanteil ab. Ist der Rotanteil minimal, mischt sich mehr blau dazu. Ist der Blauanteil maximal, nimmt der Grünanteil ab. Ist der Grünanteil minimal, mischt sich mehr rot dazu. So entsteht eine regenbogenähnliche Farbabstufung.

Das *J4KSDK* liefert die Tiefendaten zusätzlich in Form von Koordinaten für jedes Pixel. Die Klasse *DepthMap* erzeugt aus den Koordinaten eine 3D-Abbildung der Tiefe.

```
1  DepthMap map;
2  ...
3  @Override
4  public void onDepthFrameEvent(float[] xyz)
5  {
6    map = new DepthMap(getDepthWidth(), getDepthHeight(), xyz);
7  }
8
9  // OpenGLPanel
10 @Override
11 public void draw()
12 {
13   map.drawNormals(gl);
14 }
```

Listing 16: Erzeugen und Zeichnen einer 3D-Abbildung

Mit der Methode *drawNormals* zeichnet *OpenGL* die 3D-Abbildung.

5.3 Darstellung des Infrarotbildes

Die Darstellung des Infrarotbildes ähnelt der Darstellung des Tiefenbildes als Graustufenbild.

5.4 Darstellung des Skelettes

Die Klasse *Skeleton* baut ein Skelett aus den Positionsdaten, Orientierungsdaten und Verfolgungsstatus der Gelenkpunkte auf. Die Klasse *Body* dient als *Wrapper* für ein Skelett und die Gelenkpunkte in Form von Instanzen der Klasse *Object3D*. Das Listing 17 zeigt den Aufbau der Skelettdaten.

```
1  Body[] bodies = new Body[6];
2  ...
3  @Override
4  public void onSkeletonFrameEvent(boolean[] flags, float[] positions,
       float[] orientations, byte[] states)
5  {
6    for (int i = 0; i < 6; i++)
7    {
8      bodies[i].setSkeleton(Skeleton.getSkeleton(i, flags, positions,
           orientations, states, this));
9    }
10 }
```

Listing 17: Aufbau der Skelettdaten

Die Gelenkinstanzen dienen zur Berechnung der absoluten Positionen und Orientierungen und zur Darstellung in dem 3D-Editor (siehe Abschnitt 4.2). Die Klasse *Skeleton* hat ebenfalls eine Methode, um das Skelett im Koordinatensystem des *OpenGL*s zu zeichnen. Die Klasse *Body* hat daher zwei Methoden, um das jeweilige Skelett zu zeichnen. Die Methode *draw* zeichnet das Skelett im 3D-Editor mit den absoluten Positionen und Orientierungen. Die Methode *drawSkeleton* zeichnet das Skelett mit den relativen Positionen. Die 3D-Abbildung (siehe Abschnitt 5.2) zeichnet jedes Pixel mit relativen Positionen. Eine kombinierte Darstellung ist daher möglich.

6 Bestimmung der Personen, die zum Display blicken

Für die Bestimmung reichen einfache mathematische Grundlagen aus dem Bereich der Geometrie aus. Das Display ist eine Fläche innerhalb einer Ebene (siehe Abschnitt 2.7.2). Die linke Ecke des Displays ist der Stützvektor der Ebene. Die anliegenden Ecken sind die skalierten Spannvektoren, wobei die Skalaren u und v gleich Eins sind. Die Blickrichtung ist der Richtungsvektor einer Geraden, wobei die Kopf- oder Nackenposition der Stützvektor ist (siehe Abschnitt 2.7.1). Die Blickrichtung ist ein Vektor, der initial in die positive z-Richtung zeigt. Die Orientierung des Kopfes oder des Nackens dreht die Blickrichtung. Die Gerade und die Ebene bilden ein lineares Gleichungssystem (siehe Gleichung 20). Das Gaußsche Eliminierungsverfahren löst das Gleichungssystem und liefert s, u und v, s entspricht dabei der Blickrichtung. Ist s positiv, so ist die Blickrichtung positiv, sprich die Person schaut nach vorne. u und v sind die zweidimensionalen Koordinaten auf der Ebene. Haben u und v Werte zwischen Null und Eins, so schneidet die Gerade das Display. Im *SDK* von Microsoft bekommt der Kopf die Orientierung von einem *Facetracker*. Das *J4KSDK* hat momentan keinen *Facetracker*. Bis dahin dient ein einfacher Nasendetektor zur Bestimmung der Nasenposition und der Orientierung des Kopfes. Alternativ bietet sich der Nacken für die Bestimmung der Blickrichtung an, falls der Nasendetektor nicht zulänglich funktioniert.

6.1 Nasendetektor

Für diesen Nasendetektor liegt die Annahme zugrunde, dass die Nase am Weitesten aus dem Gesicht herausragt. Der Kopf und der Nacken bilden eine Gerade. Das Gesicht bildet eine Matrix aus Punkten. Die längste Verbindung zwischen einem Punkt im Gesicht und seiner orthogonalen Projektion auf der Geraden, ist der Vektor, der die Nase repräsentiert. Die Abbildung 17 zeigt die Annahme anschaulich. Die grünen Linen stehen für die Pixelkoordinaten. Die blauen Linien stehen für die Verbindungen zwischen den Pixelkoordinaten und derer orthogonalen Projektionen. Die rote Linie steht für die Gerade zwischen Kopf und Nacken.

Algorithmus:

1. Gerade zwischen Kopf- und Nackenposition bestimmen
2. Horizontale Gerade auf Kopfhöhe bestimmen
3. Horizontale Gerade auf Nackenhöhe bestimmen
4. Bildausschnitt bestimmen
    a) Bildposition der Kopfposition bestimmen

Abbildung 17: Ansatz eines Nasendetektors

   b) Bildposition der Nackenposition bestimmen
   c) Mittelpunkt und Abstand bestimmen
   d) Startposition des Bildausschnittes bestimmen
   e) Zweifacher Abstand ist die Seitenlänge des Bildausschnittes
5. Abbruch, wenn Bildausschnitt außerhalb des Bildes ist
6. Maximaler Vektor als Nullvektor festlegen
7. Nasenposition als Nullvektor festlegen
8. Für jedes Pixel aus dem Bildausschnitt
   a) Pixel überspringen, wenn Pixel nicht zur Person gehört
   b) Orthogonale Projektion auf der Geraden (Kopf - Nacken) bestimmen
   c) Orthogonale Projektion auf der Geraden (Kopfhöhe) bestimmen
   d) Orthogonale Projektion auf der Geraden (Nackenhöhe) bestimmen
   e) Vektoren zwischen Pixel und orthogonalen Projektionen bestimmen
   f) Pixel überspringen, wenn Pixel außerhalb der Geraden liegt
   g) Vektor als neuen maximalen Vektor setzen, wenn Vektor größer als der aktuelle maximale Vektor ist
   h) Pixel als Nasenposition setzen, wenn Vektor größer als der aktuelle maximale Vektor ist

Der maximale Vektor repräsentiert die Blickrichtung einer Person. Die Blickrichtung der Kinect verläuft entlang der positiven z-Achse. Im Koordinatensystem der Kinect entspricht die Blickrichtung der Person der gedrehten Blickrichtung der Kinect. Die Orientierung des Gesichtes ergibt sich aus der Berechnung wie im Abschnitt 2.6.7 beschrieben, wobei $\vec{a}$ die positive z-Achse und $\vec{b}$ der gefundene maximale Vektor ist.

7 Evaluierung

Am Ende der Bachelorarbeit ist es notwendig, die Funktionalität der Kernkomponenten zu evaluieren.

7.1 Evaluierung der Personenanzahl

Um zu testen, ob das Programm bis zu sechs Personen zählen kann, stellten sich sieben Personen in das Sichtfeld der Kinect.

Ergebnis:

Solange keine Person eine andere Person verdeckt, zählt das Programm alle sechs Personen. Die siebte Personen wurde ignoriert und löste keinen Fehler aus.

7.2 Evaluierung der Positionsdaten

Für die Evaluierung der Positionsdaten ist es sinnvoll, eine gleichbleibende Position eines Vergleichsobjektes zu haben. Dafür steht ein Kunstkopf auf einem Stativ über dem Koordinatenursprung des *Cognitive System Lab*s (siehte Abbildung 18). Die Kopfhöhe

Abbildung 18: Aufbau für Testmessungen der Kopfposition

beträgt gemessen rund 168 cm. Für die Testmessung ändert sich die Position und die Orientierung der Kinect. Damit die Kinect den Kunstkopf als Person erkennt, benötigt der Kopf noch zusätzlich einen Körper. Dafür reicht eine Jacke oder ähnliches. In der Tabelle 3 stehen die gemessenen Kopfposition in Abhängigkeit von der Position und Orientierung der Kinect.

Die Messwerte zeigen, dass die Umrechnung alles in allem funktioniert. Es gibt kleine Abweichungen von bis drei Zentimetern. Die Messung der Position und Orientierung der Kinect erfolgte wahrscheinlich zu ungenau mit dem Zollstock. Ein Fehler innerhalb der Position überträgt sich direkt in die Umrechnung. Ein Fehler in der Orientierung hängt von der Entfernung ab. Je größer die Entfernung der Person zur Kinect ist, desto

Kinect-Lage	Kameraposition	Kameraorientierung	Kopfposition
1	(0.0, -1.87, 0.94)	(85, 0, 180)	(-0.01, 0.01, 1.69)
2	(-0.34, -1.72, 0.94)	(85, 0, 165)	(0.01, 0.00, 1.69)
3	(-0.87, -1.63, 0.94)	(85, 0, 150)	(0.00, 0.01, 1.70)
4	(-1.34, -1.24, 0.94)	(85, 0, 135)	(0.00, 0.00, 1.70)
5	(-1.47, -0.78, 0.94)	(85, 0, 110)	(-0.02, 0.02, 1.68)
6	(0.0, -1.80, 1.84)	(95, 0, 180)	(0.00, 0.00, 1.70)
7	(0.0, -1.80, 0.94)	(85, 0, 180)	(0.00, 0.00, 1.70)
8	(0.0, -1.80, 0.74)	(60, 0, 180)	(0.00, 0.00, 1.71)

Tabelle 3: gemessene Kopfposition

größer ist der Fehler.

Beispiel:

Die vertikale Ausrichtung der Kinect soll um den Winkel α abweichen. Eine Person steht in einer bestimmten Entfernung d zu der Kinect. Die Tabelle 4 zeigt den Fehler in Abhängigkeit von der Abweichung und der Entfernung. Der vertikale Fehler lässt sich folgendermaßen berechnen: $e = d \cdot sin\alpha$.

α in Grad	d in m	vertikaler Fehler in cm
1	1	1.7
1	2	3.5
1	3	5.2
1	4	7.0
10	1	17.3
10	2	34.7

Tabelle 4: Fehler in Abhängigkeit von der Abweichung und der Entfernung

Eine kleine Abweichung von einem Grad ergibt in einer Entfernung von vier Metern einen Fehler von sieben Zentimetern. Genaues Messen ist daher unbedingt notwendig. Zur Kalibrierung kann sich eine Person in größtmöglicher Entfernung zu der Kinect stellen, während eine andere Person die Orientierung an Hand der gelieferten Positionsdaten einstellt.

7.3 Evaluierung des Nasendetektors

Für die Versuchsdurchführung klebte ein nasenähnliches Gebilde am Kunstkopf (siehe Abbildung 19). Die Durchführung ist dieselbe wie im Abschnitt 7.2.

Abbildung 19: Nase am Kunstkopf

Kinect-Lage	Kameraposition	Kameraorientierung	Nasenposition
1	(0.0, -1.87, 0.94)	(85, 0, 180)	(0.03, -0.09, 1.72)
1	(0.0, -1.87, 0.94)	(85, 0, 180)	(0.00, -0.10, 1.62)
2	(-0.34, -1.72, 0.94)	(85, 0, 165)	(0.05, -0.13, 1.65)
2	(-0.34, -1.72, 0.94)	(85, 0, 165)	(0.0, -0.14, 1.72)
2	(-0.34, -1.72, 0.94)	(85, 0, 165)	(0.01, -0.11, 1.62)
3	(-0.87, -1.63, 0.94)	(85, 0, 150)	(0.03, -0.09, 1.68))
4	(-1.34, -1.24, 0.94)	(85, 0, 135)	(-0.04, -0.13, 1.72)
4	(-1.34, -1.24, 0.94)	(85, 0, 135)	(-0.01, -0.08, 1.65)
5	(-1.47, -0.78, 0.94)	(85, 0, 110)	(-0.03, -0.06, 1.72)
5	(-1.47, -0.78, 0.94)	(85, 0, 110)	(-0.10, -0.01, 1.63)
6	(0.0, -1.80, 1.84)	(95, 0, 180)	(-0.10, -0.00, 1.68)
6	(0.0, -1.80, 1.84)	(95, 0, 180)	(-0.02, -0.11, 1.63)
7	(0.0, -1.80, 0.94)	(85, 0, 180)	((-0.03, -0.14, 1.64)
7	(0.0, -1.80, 0.94)	(85, 0, 180)	((-0.12, -0.15, 1.69)
8	(0.0, -1.80, 0.74)	(60, 0, 180)	(-0.12, -0.15, 1.71)

Tabelle 5: gemessene Nasenposition

Ergebnis:

In vielen Fällen hat der Nasendetektor die Nase erkannt, sprang aber mehrmals im Gesicht zu anderen Positionen (siehe Tabelle 5). Der einfache Algorithmus ist nicht stabil.

Gründe:

Die Tiefendaten sind nicht genau, d. h. es entstehen an glatten Flächen Erhebungen und Vertiefungen, die nicht vorhanden sind. Die Erhebungen können einen Vektor erzeugen, der länger als der Nasenvektor ist.

Die Annahme ist falsch. Die Nase ragt nicht unbedingt am Weitesten aus dem Gesicht heraus. Ein breites Gesicht oder viel Haarvolumen können Vektoren erzeugen, die länger als der Nasenvektor sind.

Die Gerade zwischen Kopf und Nacken ist nicht orthogonal zu dem Gesicht. Die Länge der Gesichtsvektoren hängt von der Lage der Gerade ab. Die Annahme kann daher unzulänglich mit diesem Ansatz überprüft werden.

Die Nase muss signifikant länger sein, damit der einfache Algorithmus funktioniert. Die Abbildung 20 zeigt die Erkennung eines Kugelschreibers im Mund des Kunstkopf.

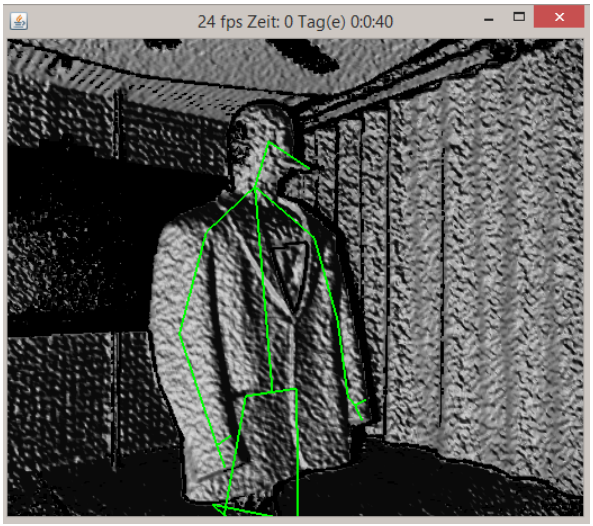

Abbildung 20: Erkennung eines Kugelschreibers

7.4 Evaluierung der Blickrichtung

Für die Testdurchführung wurden mehrere virtuelle Displays in unterschiedlicher Größe vor dem Kunstkopf so positioniert, dass die Mittelpunkte dieselbe Position hatten. Dabei schnitt die Blickrichtung der Kunstfigur den gemeinsamen Mittelpunkt. Die Testdurchführung durchlief alle Kinect-Lagen aus der Tabelle 3, wobei die Ausrichtung der virtuellen Displays in der ersten Kinect-Lage erfolgte. Die Tabelle 6 gibt einen Überblick über die Schnittbereiche in Abhängigkeit der Kinect-Lage.

In den ersten fünf Lagen ist zu beobachten, wie die Blickrichtung die kleineren Bereiche verlässt, wenn die Person eher seitlich zu der Kinect steht. In den letzten drei frontalen Lagen ist zu beobachten, wie die Blickrichtung den 20 cm Bereich verlässt, wenn die Kinect von unten nach oben. schaut. Im 3D-Editor ist zu beobachten, wie sich das Skelett leicht mit dreht. Der Abstand zwischen der linken und der rechten Seite nimmt

Kinect-Lage	10 cm	20 cm	40 cm	80 cm	160 cm
1	x	x	x	x	x
2	-	-	x	x	x
3	-	-	x	x	x
4	-	-	-	x	x
5	-	-	-	-	x
6	x	x	x	x	x
7	x	x	x	x	x
8	-	-	x	x	x

Tabelle 6: Schnittbereiche

ab. Die Blickrichtung steht aber wie erwartet orthogonal zu der Geraden zwischen der linken und rechten Schulter.

Gründe:

Am besten „arbeitet“ die Kinect, wenn die Person frontal zu der Kinect steht. Dafür wurde die Kinect konzipiert. Der Spieler steht frontal vor dem Fernseher und die Kinect darüber oder darunter.

Schlussfolgerung:

Die Stauchung und die Drehung des Skeletts kommen direkt von der Kinect, da die Umrechnung der Positionsdaten funktioniert. Die Berechnung der Orientierung erfolgt aus den Skelettdaten. Aufgrund der gedrehten Darstellung des Skeletts einer seitlich stehenden Person, ist die Blickrichtung ebenfalls gedreht.

8 Ausblick

Momentan hat dieses Modul eine eingeschränkte Anzahl an Interaktionen mit dem *Cognitve System Lab*. In Zukunft wäre es denkbar, mit dem Raum per Gesten zu interagieren. Beispielsweise könnte eine Person mit einer definierten Geste das Display an- bzw. ausmachen. Möglich wäre es auch, das Mikrofonfeld an der Decke zu steuern. Da gibt es viele Möglichkeiten. Da die Kinect nur dann zufriedenstellend funktioniert, wenn die Person frontal mit der Kinect interagiert, könnte das *Cognitve System Lab* mehrere Kinects beinhalten. Dabei könnte jeder Kinect ein Bereich zur „Beobachtung“ zugeteilt werden.

Die Entwickler der Java-Bibliothek *J4KSDK* haben für zukünftige Versionen geplant, einen *Facetracker* zu integrieren [Bar]. Mit einem *Facetracker* könnte eine Person zusätzlich per Gesichtsgesten mit einem Monitor interagieren. Beispielsweise könnte der Bildschirm eines Monitors sich ausschalten, wenn jemand für längere Zeit nicht hin sieht. Zusätzlich könnte sich die Blickrichtung einer Person auf das Gesicht beziehen. Momentan stammt die Blickrichtung von der Nacken-Orientierung, da der Nasendetektor nicht zufriedenstellend funktioniert.

Literaturverzeichnis

[Bar] Barmpoutis, Angelos: *eMail*

[Bar13] Barmpoutis, A.: Tensor Body: Real-Time Reconstruction of the Human Body and Avatar Synthesis From RGB-D. In: *Cybernetics, IEEE Transactions on* 43 (2013), Oct, Nr. 5, S. 1347–1356. http://dx.doi.org/10.1109/TCYB.2013.2276430. – DOI 10.1109/TCYB.2013.2276430. – ISSN 2168–2267

[But14] Butkiewicz, T.: Low-cost coastal mapping using Kinect v2 time-of-flight cameras. In: *Oceans - St. John's, 2014*, 2014, S. 1–9

[deb] *PrimeSenseNite*. https://wiki.debian.org/PrimeSenseNite. – Zugriff: 12.05.2015

[Dre] DreamSpark: *Kinect For Windows SDK*. https://www.dreamspark.com/rss/news.aspx?ID=35&FeedType=homepagefeed. – Zugriff: 06.05.2015

[DWI] Digital Worlds Institute, University: *J4K Java Library*. http://research.dwi.ufl.edu/ufdw/j4k/J4KSDK.php. – Zugriff: 21.05.2015

[Han13] Hanna, Tam: *Microsoft KINECT - Programmierung des Sensorsystems*. 1. Auflage. Heidelberg : Dpunkt.Verlag GmbH, 2013. – ISBN 978–3–864–90030–3

[Joh] John: *Kinect for Java*. https://github.com/ccgimperial/Kinect-for-Java. – Zugegriffen 09.05.2015

[KHK+10] Kim, Seong-Jin ; Han, Sang-Wook ; Kang, Byongmin ; Lee, Keechang ; Kim, J.D.K. ; Kim, Chang-Yeong: A Three-Dimensional Time-of-Flight CMOS Image Sensor With Pinned-Photodiode Pixel Structure. In: *Electron Device Letters, IEEE* 31 (2010), Nov, Nr. 11, S. 1272–1274. http://dx.doi.org/10.1109/LED.2010.2066254. – DOI 10.1109/LED.2010.2066254. – ISSN 0741–3106

[Kho11] Khoshelham, K.: ACCURACY ANALYSIS OF KINECT DEPTH DATA. In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XXXVIII-5/W12 (2011), 133–138. http://dx.doi.org/10.5194/isprsarchives-XXXVIII-5-W12-133-2011. – DOI 10.5194/isprsarchives–XXXVIII–5–W12–133–2011

[KPG14] Kusari, A. ; Pan, Zhigang ; Glennie, C.: Real-time indoor mapping by fusion of structured light sensors. In: *Ubiquitous Positioning Indoor Navigation and Location Based Service (UPINLBS), 2014*, 2014, S. 213–219

[Lau] Lau, Daniel: *The Science Behind Kinects or Kinect 1.0 versus 2.0*. http://www.gamasutra.com/blogs/DanielLau/20131127/205820/The_Science_Behind_Kinects_or_Kinect_10_versus_20.php. – Zugriff: 15.04.2015

[Mica] Microsoft: *Kinect for Windows SDK 1.8*. https://msdn.microsoft.com/en-us/library/hh855347.aspx. – Zugriff: 08.04.2015

[Micb] Microsoft: *Kinect for Windows SDK 2.0*. https://msdn.microsoft.com/de-de/library/dn799271.aspx. – Zugriff: 08.04.2015

[Neu] Neufeld, Eugen: *Jnect*. https://code.google.com/a/eclipselabs.org/p/jnect/. – Zugriff: 07.05.2015

[Occ] Occipital: *The rumors of my death have been greatly exaggerated...* http://structure.io/openni. – Zugriff: 12.05.2015

[opea] *OpenNI*. https://github.com/OpenNI/OpenNI. – Zugriff: 11.05.2015

[opeb] *OpenNI2*. https://github.com/occipital/openni2. – Zugriff: 12.05.2015

Anhang

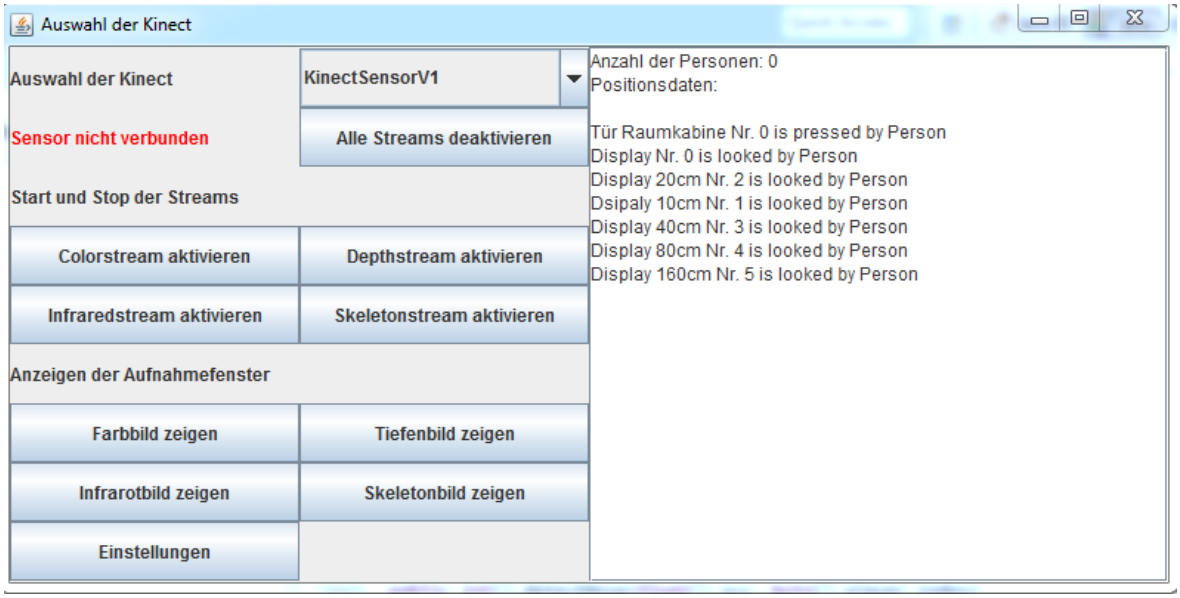

Abbildung 21: Startfenster

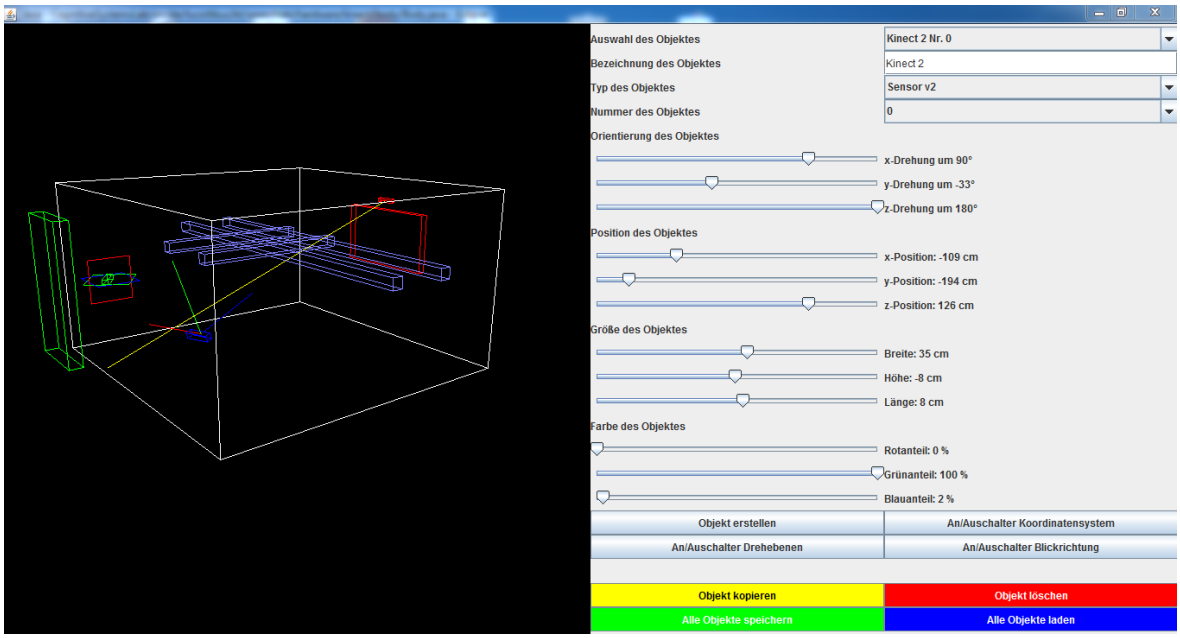

Abbildung 22: 3D-Editor

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Bachelorarbeit

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

$Unterschrift$: $Ort, Datum$: