

**Koppelung einer Bewegungsentwicklung
für humanoide Roboter
mit der Bewegungserkennung
von Personen durch die Kinect**

Studienarbeit

für die Prüfung zum

Bachelor of Engineering

des Studienganges Informatik
Studienrichtung Informationstechnik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Dennis Alles

und

Karolin Edigkaufer

12.05.2014

Matrikelnummer

3934520 (Dennis Alles)

6589515 (Karolin Edigkaufer)

Kurs

TINF11B3

Betreuer

Prof. Dr. Hans-Jörg Haubner

Michael Schneider

Ehrenwörtliche Erklärung

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

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

Ort, Datum

Unterschrift

Ort, Datum

Unterschrift

Abbildungsverzeichnis

| | |
|---|----|
| Abb. 1: Die Sensoren der Kinect [8] | 4 |
| Abb. 2: Skeleton mit 20 Punkten [9] | 5 |
| Abb. 3: Kinect Developer Toolkit..... | 7 |
| Abb. 4: Kinect Explorer-WPF | 8 |
| Abb. 5: Roll - Nick - Gier - Winkel / Roll - Pitch - Yaw - Winkel [15]..... | 9 |
| Abb. 6: Die Lage der Motoren beim Nao [21]..... | 12 |
| Abb. 7: Die Hardwarekomponenten des Nao [22]..... | 13 |
| Abb. 8: Nao Software | 14 |
| Abb. 9: Choreographie | 15 |
| Abb. 10: Nao Programmiersprachen..... | 15 |
| Abb. 11: Nao Bibliothek nutzen [23] | 16 |
| Abb. 12: Visual Studio | 17 |
| Abb. 13: Threads aus Nutzersicht [25] | 18 |
| Abb. 14: Threads im Vergleich zu Prozessen [26] | 19 |
| Abb. 15: Zustände von Threads [26] | 19 |
| Abb. 16: Code Map..... | 22 |
| Abb. 17: Programmablaufplan – erster Teil..... | 26 |
| Abb. 18: Programmablaufplan – zweiter Teil..... | 27 |
| Abb. 19: Startposition: <i>StandZero</i> | 33 |
| Abb. 20: IDMotion1: beide Arme unten | 33 |
| Abb. 21: IDMotion2: beide Arme außen | 34 |
| Abb. 22: IDMotion3: beide Ellenbogen angewinkelt | 34 |
| Abb. 23: Diagramm der unterschiedlichen Winkel bei einer Bewegungsausführung..... | 34 |
| Abb. 24: GUI zum Spiel..... | 41 |
| Abb. 25: Testergebnisse | 42 |
| Abb. 26: Abfrage der IP-Adresse | 43 |

Codeverzeichnis

| | |
|---|----|
| Code 1: Warten auf das Beenden eines anderen Threads | 29 |
| Code 2: Starten von zwei Threads..... | 30 |
| Code 3: Ausführen einer zufälligen Bewegung (vereinfacht) | 31 |
| Code 4: Eine vordefinierte Bewegung von Nao | 32 |
| Code 5: Umrechnung von Gradzahl in Radiant | 33 |
| Code 6: Speicherung der Winkel vom Nao | 35 |
| Code 7: Umrechnung von Radiant in Gradzahl | 36 |
| Code 8: Jointübergabe für Winkel..... | 37 |
| Code 9: Berechnung von zusammenhängenden Knochen | 37 |
| Code 10: Berechnung von getrennten Knochen | 38 |
| Code 11: Winkelvergleich..... | 39 |
| Code 12: Schwierigkeitsgrad | 40 |

Abkürzungsverzeichnis

| | |
|-------|------------------------------------|
| DHBW | Duale Hochschule Baden-Württemberg |
| GUI | Graphical User Interface |
| IP | Internet Protokoll |
| SDK | Software Development Kit |
| STA | Singlethreaded Apartment |
| VGA | Video Graphics Array |
| Wi-Fi | Wireless Fidelity |

Inhaltsverzeichnis

| | |
|---|-----|
| Ehrenwörtliche Erklärung..... | II |
| Abbildungsverzeichnis..... | III |
| Codeverzeichnis | IV |
| Abkürzungsverzeichnis..... | V |
| Inhaltsverzeichnis..... | VI |
| 1 Einleitung..... | 1 |
| 1.1 Motivation..... | 1 |
| 1.2 Aufgabenumfeld..... | 1 |
| 1.3 Aufgabenbeschreibung und Ziele..... | 2 |
| 1.4 Vorgehensweise | 2 |
| 2 Grundlagen..... | 3 |
| 2.1 Kinect..... | 3 |
| 2.1.1 Sensoren der Kinect | 3 |
| 2.1.2 SDK..... | 5 |
| 2.1.3 Kinect Developer Toolkit | 6 |
| 2.2 Winkel..... | 9 |
| 2.2.1 Roll-Nick-Gier-Winkel | 9 |
| 2.2.2 Winkelberechnung | 10 |
| 2.3 Humanoide Roboter | 10 |
| 2.3.1 Nao | 10 |
| 2.3.1.1 Hardware..... | 11 |
| 2.3.1.2 Software | 14 |
| 2.3.2 Andere humanoide Roboter | 16 |
| 2.4 Entwicklungsumgebung | 17 |
| 2.5 Threads..... | 18 |

| | | |
|-------|---|----|
| 3 | Konzeption | 20 |
| 4 | Umsetzung..... | 22 |
| 4.1 | Programmstruktur / Architektur | 22 |
| 4.1.1 | Programmablauf bei einer Bewegungsnachahmung..... | 25 |
| 4.1.2 | Threadeinsatz | 28 |
| 4.2 | Nao | 30 |
| 4.2.1 | Zufällige Bewegung ausführen | 31 |
| 4.2.2 | Bewegung wiederholen..... | 32 |
| 4.2.3 | Beispiel einer spezifischen Bewegung | 32 |
| 4.2.4 | Speicherung Winkel vom Nao | 35 |
| 4.3 | Kinect..... | 36 |
| 4.3.1 | Berechnung Winkel Kinect | 36 |
| 4.3.2 | Winkelvergleich Kinect - Nao | 39 |
| 4.3.3 | Schwierigkeitsgrade | 40 |
| 4.4 | GUI..... | 40 |
| 4.5 | Test | 42 |
| 4.6 | Benutzung des Programms | 43 |
| 5 | Fazit | 44 |
| | Literaturverzeichnis..... | IX |

1 Einleitung

Sowohl Roboter als auch die direkte Interaktion zwischen Anwender und Computer liegen voll im Trend in der Informatik. Die starre Programmierung eines Rechners wird es zwar immer geben, aber oftmals rückt das Thema den Anwender selbst mit einzubeziehen mehr und mehr in den Vordergrund. Hierbei liegt die große Herausforderung in der Wahrnehmung und Nachahmung von menschlichen Aktionen [1].

Diese Studienarbeit beschäftigt sich mit dem System Kinect von Microsoft und dem humaoiden Roboter Nao.

1.1 Motivation

Die Motivation ist es, den Roboter in einer spielerischen Umgebung mit dem Menschen interagieren zu lassen, um so auch Berührungsängste mit dieser neuartigen Technik zu nehmen. Zudem kann dem Anwender auf diese Weise auch gezeigt werden, welche Möglichkeiten es gibt, Roboter zu nutzen. So ist es zum Beispiel denkbar, dass gewisse Bewegungsabläufe, wie das Erlernen von Schwimmbewegungen, in Zukunft über so ein Verfahren bereits trocken gelernt werden können.

Zudem bietet diese Aufgabe die Möglichkeit sich in ein neues Themengebiet mit neuer Programmiersprache und besonderer Hardware einzuarbeiten, welche in der bisherigen Laufbahn noch unbekannt waren.

1.2 Aufgabenumfeld

Diese Studienarbeit entsteht an der Dualen Hochschule Baden Württemberg (DHBW) und wird von zwei Studenten gemeinsam geschrieben. Zur Verfügung steht ein Labor mit einem Nao, einer Kinect für Windows und einem Rechner, auf dem alle relevanten Dokumente und Programme genutzt werden können. Zudem können die Programme auch auf den privaten Rechnern installiert werden, so dass mit Hilfe von Simulationssoftware auch von zu Hause aus gearbeitet werden kann.

1.3 Aufgabenbeschreibung und Ziele

Ziel der Arbeit ist es, ein Spiel zu programmieren, bei dem sowohl ein Roboter als auch das Kamerasystem Kinect zum Einsatz kommen. Dem Roboter Nao werden verschiedene Bewegungen beigebracht, die er auf Befehl automatisiert ausführen kann.

Die Kinect ist dafür zuständig, die Bewegungen des davorstehenden Spielers aufzunehmen.

Der Ablauf des programmierten Spiels kann in drei Schritte unterteilt werden. Im ersten Schritt führt Nao eine der programmierten Bewegungen vor. Sobald die Roboterbewegung vollendet ist, beginnt Schritt zwei, in dem die Kinect die Bewegung des Spielers aufnimmt. Gleichzeitig läuft eine festgelegte Zeit herunter. Die Aufgabe des Spielers ist es innerhalb der Zeit vom Roboter durchgeführte Bewegung möglichst ähnlich nachzuahmen. Nach dem Ablauf der Zeit beginnt Schritt drei, bei dem die getätigte Benutzerbewegung mit der programmierten Roboterbewegung vom System verglichen wird.

Dieser Ablauf mit anschließender Auswertung wird in einem spielerischen Umfeld dargestellt und im Programm angezeigt.

1.4 Vorgehensweise

Die geplante Vorgehensweise sieht vor dem Nao verschiedene einfache Bewegungen einzuprogrammieren, dessen Winkel zu Beginn und am Ende der Bewegung abgespeichert werden. Anschließend wird die Kinect mit einem Skeleton Stream programmiert. Es werden verschiedene Winkel im Oberkörperbereich berechnet und abgespeichert. Diese sollen dann mit den beiden Winkeln des Roboters verglichen werden. Das Ergebnis soll in einer übersichtlichen Graphical User Interface (GUI) dem Nutzer gezeigt werden.

Sofern dieses einfache Beispiel reibungslos funktioniert, werden die Bewegungen des Roboters umfangreicher und komplizierter und es sollen zudem Zwischenwerte der Winkel verglichen werden.

2 Grundlagen

Dieses Kapitel schildert die für das Verständnis dieser Arbeit notwendigen Grundlagen. Zu diesem Zweck wird in Kapitel 2.1 die „Kinect-Kamera“ und in Kapitel 2.3 der Roboter Nao vorgestellt. Zudem werden die zur Winkelberechnung erforderlichen Grundlagen aufgezeigt, Grundlagen in der Threadprogrammierung sowie die Entwicklungsumgebung beschrieben.

2.1 Kinect

Die Kinect ist eine Hardware aus dem Hause Microsoft in Zusammenarbeit mit dem Unternehmen PrimeSense zur Gestik- und Mimikerkennung. Sie wurde zunächst für die Steuerung der Spielekonsole Xbox360 entwickelt. Die Kinect setzt wie auch die Nintendo Wii oder Sonys Move auf den Körpereinsatz des Benutzers. Der große Unterschied liegt aber darin, dass der Benutzer keinen Controller in der Hand hält, sondern der ganze Körper zum „Controller“ wird.

Die Kinect-Kamera ist eine Leiste aus vielen Sensoren (siehe Abb. 1) und erkennt so die Bewegungen der davorstehenden Personen [2], [3].

Nachdem die Kinect so erfolgreich für die Spielekonsole verkauft wurde, ist sie seit Februar 2012 auch für Windows erhältlich. Wie Microsoft selbst schreibt, erhält der Windows-Computer durch die Kinect Augen, Ohren und auch ein Gehirn. Bei der Bestellung der Hardware wird zusätzlich ein Software Development Kit (SDK) mitgeliefert, was nichts anderes ist als eine Sammlung von Werkzeugen und Anwendungen, um eine Software zu erstellen. Das Windows-SDK soll also die Software-Entwicklung für Kinect-Anwendungen erleichtern. Des Weiteren befindet sich eine Dokumentation für die Kinect im Windows-Umfeld im Angebot [4], [5].

2.1.1 Sensoren der Kinect

In der ca. 20 cm breiten Kinect sind ein Infrarot-Projektor, eine Infrarot-Kamera, eine Farb-Kamera sowie vier Mikrofone verbaut. Damit die Sensoren perfekt auf den Benutzer ausgerichtet werden können, befindet sich zwischen der Kameraliste und dem Standfuß ein Motor. Mithilfe dieses Motors kann die Sensoren-Leiste um 27 Grad geneigt werden.

Das Sehfeld der Kinect-Sensoren beträgt im Horizontalen 57 Grad und im Vertikalen 43 Grad. Aus diesem Grund muss ein Spieler auch einen Mindestabstand haben, sodass der ganze

Körper erkannt werden kann. Laut Herstellerangaben können die Sensoren der Kinect bei einem Abstand von 1,2 bis 3,5 Meter zuverlässig arbeiten. Bei der Xbox sollte der Abstand zwischen Spieler und Kameraliste bei einem Spieler ca. 1,8 Meter und bei zwei Spieler ca. 2,4 Meter betragen. Bei Anwendungen, die nur den Oberkörper des Spielers benötigen ist entsprechend ein geringerer Abstand möglich [6], [7].

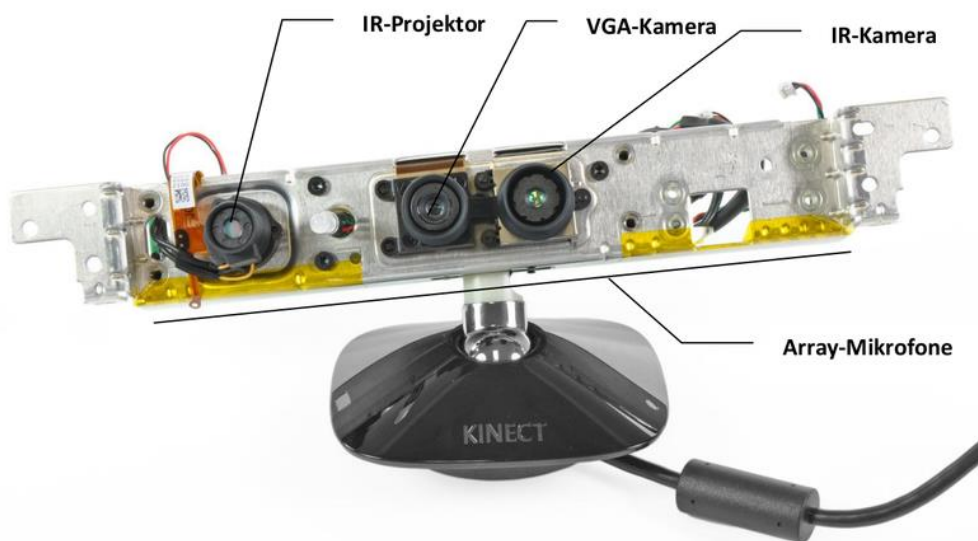


Abb. 1: Die Sensoren der Kinect [8]

Durch die beiden Infrarot-Komponenten, den Infrarot-Projektor und die Infrarot-Kamera, werden die Tiefendaten erzeugt. Sie werden also benötigt um die dritte Dimension hinzuzufügen. Die Grundlage für die 3D-Erfassung ist das strukturierte Licht. Zunächst wird von dem Infrarot-Projektor eine definierte Punkt-Matrix in den Raum gesendet. Die Infrarot-Kamera nimmt das Muster des Infrarot-Lichts wieder auf und gibt die Daten an den internen Chip weiter. Von ihm werden dann aus den Kameradaten die räumlichen Koordinaten ausgerechnet.

Bei dem Infrarot-Projektor handelt es sich um einen Klasse 1-Laser mit einer Wellenlänge von 830 nm. Da Menschen nur eine Wellenlänge von ca. 380 nm bis ca. 780 nm wahrnehmen können, liegt der Laserstrahl der Kinect im nicht-sichtbaren Spektrum. Außer

dass sie nicht sichtbar sind, hat es noch den wichtigen Vorteil, dass sie selbst bei einem Einfall ins Augeninnere des Benutzers nicht schädlich sind.

Die Farbkamera ist für die Aufnahme von 2D-Bildern zuständig. Die Kamera besitzt hierbei eine Video Graphics Array (VGA)-Auflösung von 640x480 Pixeln.

Die vier Mikrofone werden für eine Audioortung benutzt. Somit lässt sich ein Sprachbefehl eines Benutzers genau zuordnen. Die Mikrofone der Kinect ermöglichen bei einem Einsatz mit einer Xbox, das Xbox-Spiel per Sprache zu starten oder von vorne zu beginnen.

Das Kamerasystem ist mithilfe des Infrarot-Sensors in der Lage Menschen von leblosen Objekten zu unterscheiden. Für jeden erkannten Spieler, der vor der Kinect steht, wird ein Skelett mit 20 Gelenken erstellt.

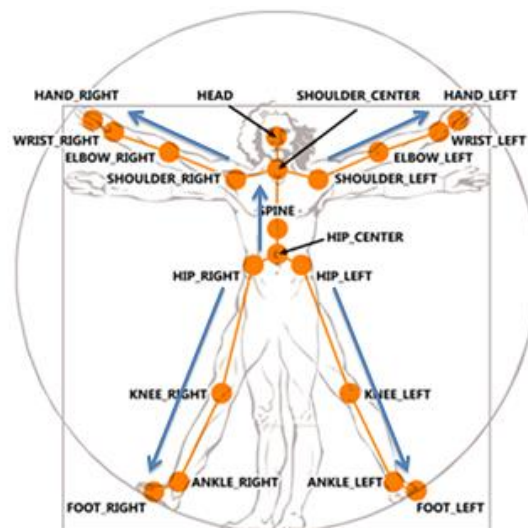


Abb. 2: Skeleton mit 20 Punkten [9]

Mithilfe dieser 20 Punkte (auch Joints genannt) können die Körperteile eines Spielers unterschieden und so Bewegungen verarbeitet werden. Außerdem ist es möglich die Gesichter von verschiedenen Personen zu unterscheiden und Aussagen über die Mimik zu treffen [5], [10].

2.1.2 SDK

Zunächst erschien die Kinect im November 2010 als zusätzliche Xbox-Hardware ohne Pläne für PC-Treiber. Viele Leuten wollten die Kinect aber in Verbindung mit einem PC bedienen können, worauf das Unternehmen Adafruit Industries eine Prämie von 1000 US-Dollar für

den ersten funktionierenden PC-Treiber anbot. Microsoft dachte, dass es sich bei den Programmierungen um Hacking-Versuche im Sinne von Raubkopien oder illegalem Gebrauch handelt und kündigte an dagegen gerichtlich vorzugehen. Später bemerkte Microsoft, dass die meisten Entwickler das System nicht hacken, sondern nur den Zugriff auf die Sensordaten nutzen wollten.

Das Resultat aus dieser Entwicklung ist, dass es nun zwei verschiedene SDKs gibt. Eins davon ist das offizielle SDK von Microsoft, was im Juni 2011 ausgeliefert wurde, und das andere ist ein quelloffenes SDK. Diese beiden SDKs stehen in Konkurrenz und die Anwendungen sind inkompatibel zueinander [11].

2.1.3 Kinect Developer Toolkit

Im SDK von Microsoft ist auch das Kinect Developer Toolkit enthalten. Es ist also wie das SDK selbst auch ein Werkzeugsatz, der für Kinect-Entwickler gedacht ist. Das englischsprachige Toolkit kann auf der Homepage von Microsoft heruntergeladen und danach auf dem eigenen Rechner installiert werden. Nach einer Installation befindet sich das Toolkit in den eigenen Programmen und kann auch wie eine normale Anwendung gestartet werden [12].

Das Toolkit bringt verschiedene Codebeispiele und weitere Entwicklungsressourcen mit.

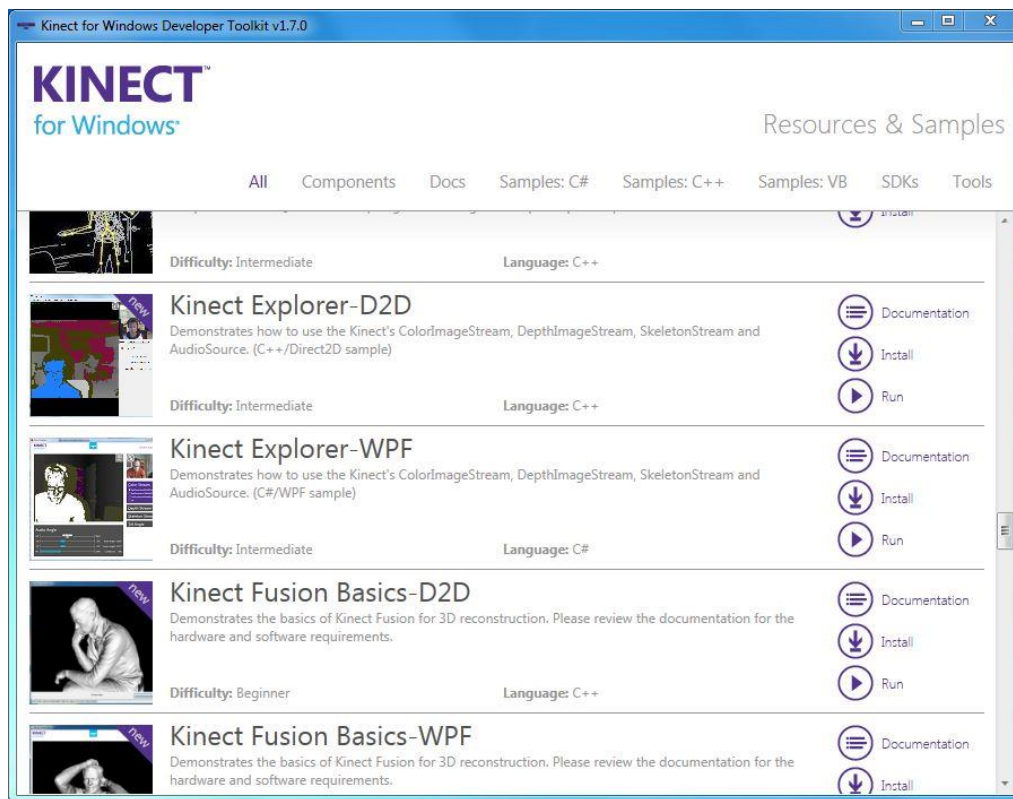


Abb. 3: Kinect Developer Toolkit

Auf dem Screenshot des Developer Toolkits sind verschiedene Programmbeispiele zu sehen. Jedes dieser Beispiele ist mit einem Bild und einer kurzen Beschreibung erklärt. Darüber hinaus ist der Schwierigkeitsgrad für Entwickler sowie die verwendete Programmiersprache aufgezeigt.

Auf der rechten Seite stehen dem Benutzer drei verschiedene Buttons zu Verfügung, mit denen er eine Online-Dokumentation zu der entsprechenden Anwendung öffnen, das Beispiel installieren oder starten kann. Ein Programm zu installieren hat den Vorteil, dass ein Ordner inklusive vollständigem Quellcode zum Bearbeiten gespeichert wird. Wenn man das Programm lediglich ausführt, so wird nichts installiert. Dies ist zum Beispiel zu Beginn eines Projektes geeignet, um die verschiedenen Anwendungen zu vergleichen und zu entscheiden, welche Anwendung zur eigenen Aufgabenstellung passt. Anschließend kann der Quelltext installiert werden.

Dabei gibt es zum Beispiel Anwendungen, die Geräusche im Raum, deren Herkunft und Frequenz anzeigen, die Entfernung von Spielern mittels DepthStream zeigen oder einfache Spiele implementiert haben.

Im Folgenden ist die konkrete Anwendung Kinect Explorer – WPF zu sehen:

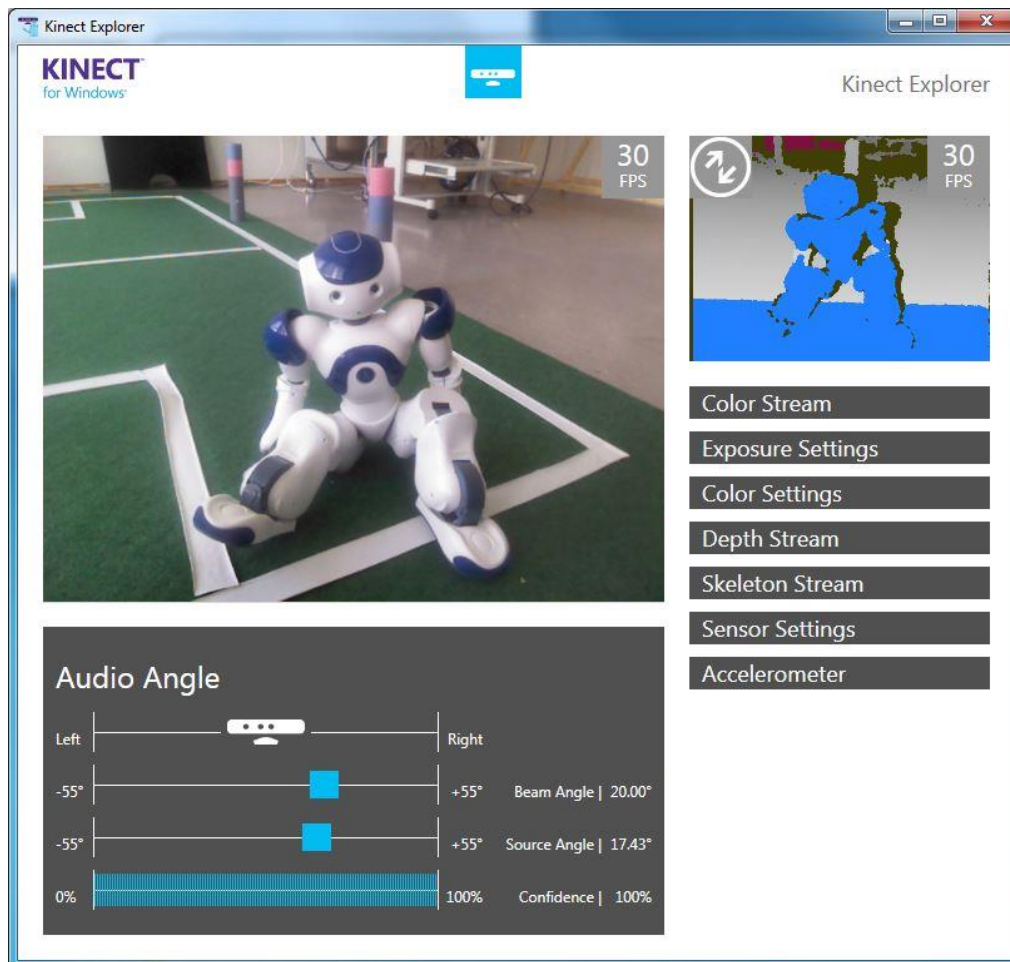


Abb. 4: Kinect Explorer-WPF

Die Anwendung Kinect Explorer-WPF ist in C# programmiert und hat den Schwierigkeitsgrad „Fortgeschrittener Anfänger“. Führt man diese Anwendung aus, so können einige Einstellungen und Funktionen der Kinect getestet werden. In diesem Beispiel ist im Hauptbild das Abbild in RGB zu sehen, bei dem bereits die 20 Gelenke des Nutzers zu sehen sind (Skeleton-Funktion). Im unteren Bildschirmbereich kann die aktuelle Tonrichtung nachverfolgt werden. Rechts oben in ein DepthStream zu sehen, bei dem man die

Entfernung des Anwenders anhand der Farben ablesen kann. In diesem Falle ist der Roboter blau eingefärbt, was bedeutet, dass dieser sich in der Nähe der Kamera befindet.

Unterhalb dieses Bildes lassen sich weitere Einstellungen in Form von Reitern aufklappen und bearbeiten. Mit dem Reiter Sensor Stream lässt sich einstellen, ob es sich um ein oder zwei Anwender handelt und ob diese sitzen oder stehen. Der Reiter Sensor Settings ist dafür zuständig, dass die Kameraliste hoch bzw. runter geneigt werden kann, sodass der davorstehende Benutzer optimal erkannt wird.

2.2 Winkel

Es gibt verschiedene Möglichkeiten Winkel mathematisch zu berechnen. Dabei ist zu beachten, dass auch der Winkel an sich in zwei unterschiedlichen Zahlentypen wiedergegeben werden kann: als Radiant und als Gradzahl. Die Umrechnung dieser lautet:

$Radiant = Grad * 180/\pi$, da der Radiant die Werte in einem Kreis anordnet [13].

2.2.1 Roll-Nick-Gier-Winkel

In der Raumfahrt ist neben der bekannten Möglichkeit ein Objekt in einem Raum mittels X, Y, Z – Koordinaten zu bezeichnen, auch die Variante mittels Roll – Nick – Gier – Winkeln vertreten. Dabei werden die Rotationen um die Achsen bezeichnet: Nick ist eine Drehung um die Querachse (X), Roll ist eine Drehung um die Längsachse (Y) und Gier eine Drehung um die Hochachse (Z). Geläufiger als die deutschen Bezeichnungen sind aber die englischen Bezeichnungen roll, pitch und yaw [14].

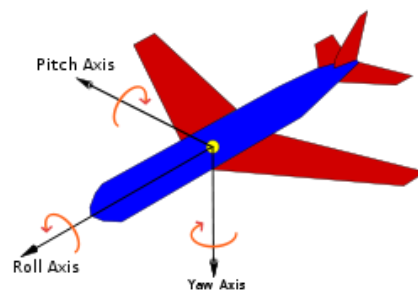


Abb. 5: Roll - Nick - Gier - Winkel / Roll - Pitch - Yaw - Winkel [15]

2.2.2 Winkelberechnung

Um die Winkel zu berechnen, vereint man drei Punkte (engl. Joints) in zwei Vektoren. Die Punkte definieren dabei Knochenenden, die Vektoren die eigentlichen Knochen. Diese werden anschließend normalisiert. Beim Normalisieren werden die Vektoren auf die Länge eins gestreckt bzw. gestaucht, da man zur Winkelberechnung lediglich die Richtung der Vektoren benötigt.

Aus den beiden Vektoren wird anschließend das Punktprodukt gebildet und sofern es zusammenhängende Knochen darstellt, wird der Winkel mittels Tangens aus dem Punktprodukt und der Kreuzprodukt der Z-Koordinate gebildet. Bei nicht zusammenhängenden Knochen wird lediglich der Kosinus des Kreuzproduktes berechnet [16].

2.3 Humanoide Roboter

Ein humanoider Roboter ist ein Roboter, der dem Mensch nachempfunden ist und dessen Bewegungen die eines Menschen ähneln. Genutzt werden humanoide Roboter zumeist in der Servicerobotik, zum Beispiel als Haushaltshilfe für häufig anfallende Aufgaben. Staubsaugen, Spülmaschine ausräumen oder Toilette putzen sind Aufgaben, die in Zukunft von Robotern ausgeführt werden könnten. Zahlreiche Unternehmen und Einrichtungen befassen sich derzeit mit der Entwicklung und dem Testen von humanoiden Robotern. Darunter sind auch namenhafte Firmen und Universitäten, wie das Karlsruher Institut für Technologie, das Fraunhofer Institut, Toyota und Google [17] [18] [19].

Ein Beispiel für humanoide Roboter in der Lehre ist der Roboter Nao, ein Beispiel für Roboter im Hausgebrauch ist der Romeo. Insbesondere Nao wird im Folgenden näher betrachtet.

2.3.1 Nao

Nao ist ein 58cm großer humanoider Roboter der Firma Aldebaran Robotics aus Frankreich und der humanoide Roboter der für die Programmierung genutzt wurde.

Der erste Prototyp vom Nao erschien 2005, seit dem sind mehr als 5000 Naos verkauft worden. Diese werden fast ausschließlich in Bildungseinrichtungen genutzt [20].

Der Roboter hat die Fähigkeiten sich zu bewegen, Menschen und Gegenstände wahrzunehmen, zu hören und zu sprechen. Dies ist durch die verschiedenen Komponenten wie Mikrofonen und Entfernungssensoren möglich.

2.3.1.1 Hardware

Der humanoide Roboter Nao in der aktuellen Version 4 hat einen integrierten Akku, welcher circa eine Stunde genutzt werden kann und etwa fünf Stunden zum vollen Aufladen benötigt.

Er ist außerdem mit einem Intel „ATOM Z530“ Prozessor mit 512 KB Cache Speicher und 1 GB RAM ausgestattet. Der eingebaute Flashspeicher hat Platz für 2 GB Daten.

Zur Kommunikation zwischen Computer und Roboter verfügt Nao über Ethernet und Wireless Fidelity (Wi-Fi). Über seine Internet Protokoll (IP)-Adresse und den Port 9559 erreicht man den Nao. Durch Eingabe der IP-Adresse in den Browser, kann man noch einige Einstellungen, den Nao betreffend, vornehmen.

Für diese Ausarbeitung wurde Nao in der Version 3.2 beziehungsweise 3+ verwendet. Die Unterschiede liegen in im verwendeten Prozessor (x86 AMD GEODE 500 MHz CPU) und dem Speicher (256MB SCRAM/2GB Flashspeicher). Im Folgenden wird nur die Version 3.2 betrachtet, da diese die Grundlage für das Programm darstellt.

Wie in Abb. 6 zu sehen, hat Nao 25 eingebaute Motoren, um die Bewegungen so humanoid wie möglich aussehen zu lassen. Sechs davon sind für Armbewegungen auf der linken Seite, sechs für Armbewegungen auf der rechten Seite, fünf für Hüftbewegungen, drei für Beinbewegungen der linken Seite, drei für Beinbewegungen der rechten Seite und zwei für den Kopf zuständig. Dabei gibt es vier unterschiedliche Arten von Motoren. Für die Motoren *HeadYaw*, *Head Pitch*, *ShoulderPitch*, *ShoulderRoll*, *EllbowYaw* und *EllbowRoll* werden Motoren des Typs zwei verwendet und simulieren ein Kugelgelenk. Typ-Drei Motoren werden lediglich im Motor *WristYaw* genutzt, da dort die Drehrichtung der Hand beeinflusst wird. Die Motoren des Typs vier sind ausschließlich bei den Händen zu finden, da diese die Finger nachbilden und Seilzüge integriert haben. Und alle Motoren, die die Hüfte und Beine betreffen, sind Motoren des Typs eins und lassen sich in zwei Richtungen (vor und zurück) ändern [21].

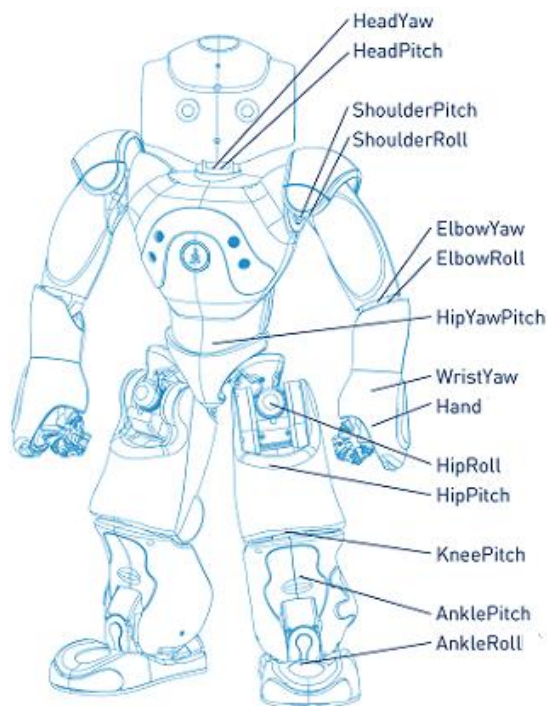


Abb. 6: Die Lage der Motoren beim Nao [21]

Neben den genannten Aktoren hat der Nao auch einige Sensoren, die es ihm ermöglichen Hindernisse zu erkennen, die Winkeleinstellungen zu messen und vieles mehr.

Die Sensoren kann man in folgende Kategorien einteilen:

- Kontaktsensoren
- Force Sensing Resistoren
- Inertialsensoren
- Inkrementalgeber (rotierend)
- Sonarsensoren

Die Kontaktsensoren finden Verwendung im Brustknopf, bei dessen Drücken Nao seinen Namen, den Akkuladestand und seine IP-Adresse verkündet, und in den beiden Bumpern je Fuß, die registrieren, ob der Roboter mit seinem Fuß einen Gegenstand berührt. Die Bumper werden als taktile Sensoren bezeichnet, da sie Berührungen wahrnehmen. Drei weitere taktile Sensoren sind auf seinem Kopf zu finden und je drei Sensoren befinden sich an jeder Hand.

Ein Force Sensing Resistor ist ein Sensor, der die einwirkende Kraft bzw. den Druck misst. Vier dieser Sensoren befinden sich an jedem Fuß des Nao, um bei falscher Krafteinwirkung rechtzeitig zu warnen.

Inertialsensoren sind Sensoren zur Messung von Beschleunigungen und Drehraten. Drei davon befinden sich im Torso des Naos.

Der Nao hat 32 Inkrementalgeber eingebaut und nutzt diese zur Erfassung von Winkelveränderungen und Drehrichtungen.

Zwei Sender und zwei Empfänger für Sonarsensoren sind an der Brust eingebaut. Diese dienen zum Messen von Entfernungen im Bereich von 0,25m und 2,55m.

Zudem hat er zwei Kameras, vier Mikrofone und einen Lautsprecher im Kopf integriert, sodass er mit seiner Umwelt interagieren kann.

Die Hardwarekomponenten sind wie folgt verteilt:

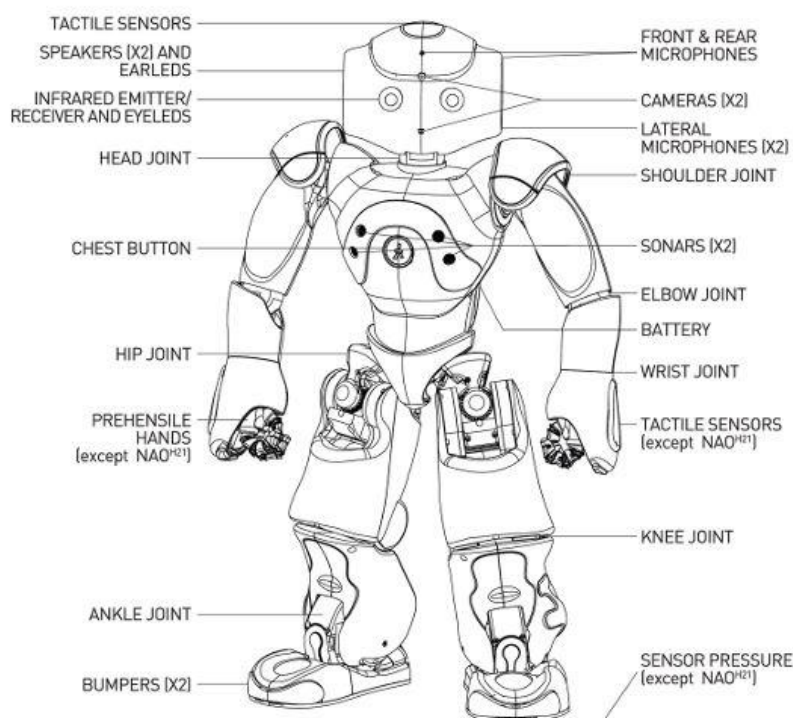


Abb. 7: Die Hardwarekomponenten des Nao [22]

2.3.1.2 Software

Auf dem Nao läuft ein Linux Embedded Betriebssystem, das OpenNAO, welches die Software NAOqi ausführt (siehe Abb. 8). OpenNAO ist ein 32-Bit Linux Distribution, angelehnt an Gentoo.

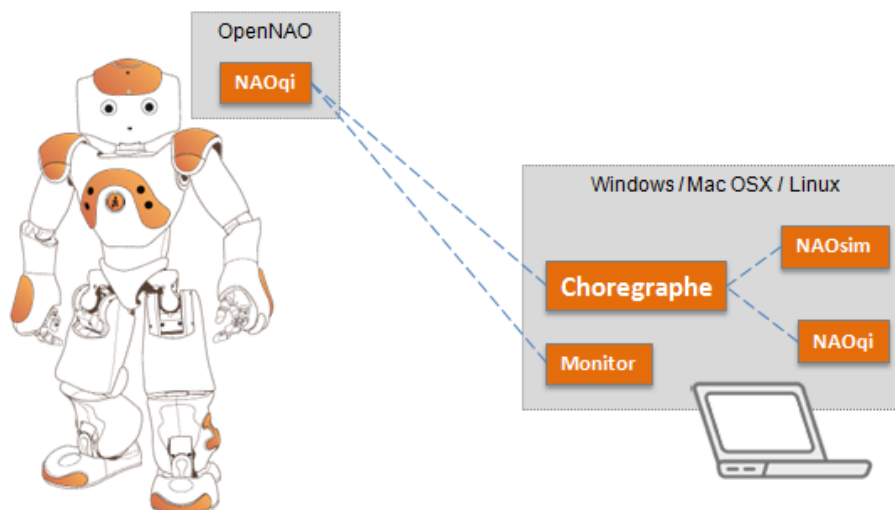


Abb. 8: Nao Software

Um den Nao zu programmieren, gibt es die Desktop Software „Choregraphe“, welche vom Hersteller mitgeliefert wird. Das ist eine visuelle Programmiersprache und hat ebenfalls das Programm NAOqi implementiert. Webots ist die Simulation von einem Nao, früher NAOsim genannt. Bevor ein Stück Programmcode auf dem Nao ausgeführt wird, kann man sich so vergewissern, dass das Programm das tut, von dem man ausgeht und ob dies eventuell zu Problemen auf dem Nao führen kann. So kann verhindert werden, dass der Roboter versucht Bewegungen auszuführen, für die seine Motoren nicht geeignet sind oder ob der Roboter eventuell Schaden nimmt, wie zum Beispiel durch Umfallen.

In der Simulationswelt ist es dabei auch möglich Hindernisse einzubauen, um so nicht nur die Motoren, sondern auch die Sensoren zu testen.

Im Choreographen, wie in Abb. 9 zu sehen, ist es möglich, vorgespeicherte Bewegungen auszuführen, z.B. hinsetzen, hinlegen, gerade hinstellen, alle Winkel auf 0 Grad (*StandZero*), Nao Texte sprechen zu lassen und gewisse Abläufe auf ihm zu speichern, sodass diese auch nach Neustarten des Roboters noch verfügbar sind.

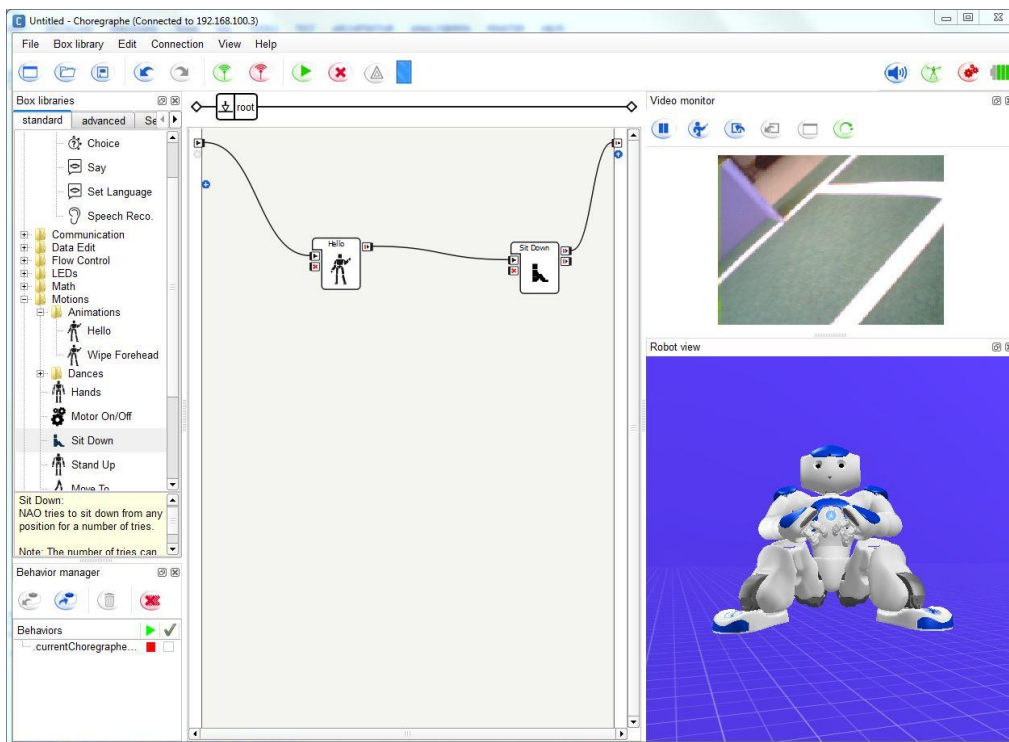


Abb. 9: Choregraphe

Choregraphe ist zudem eine der vorhandenen SDKs. Nao kann direkt mit Python und C++ angesprochen werden. Zu anderen SDKs gibt es Programmierschnittstellen, sodass Nao auch Java, MathLab und .NET Sprachen, wie C#, versteht.

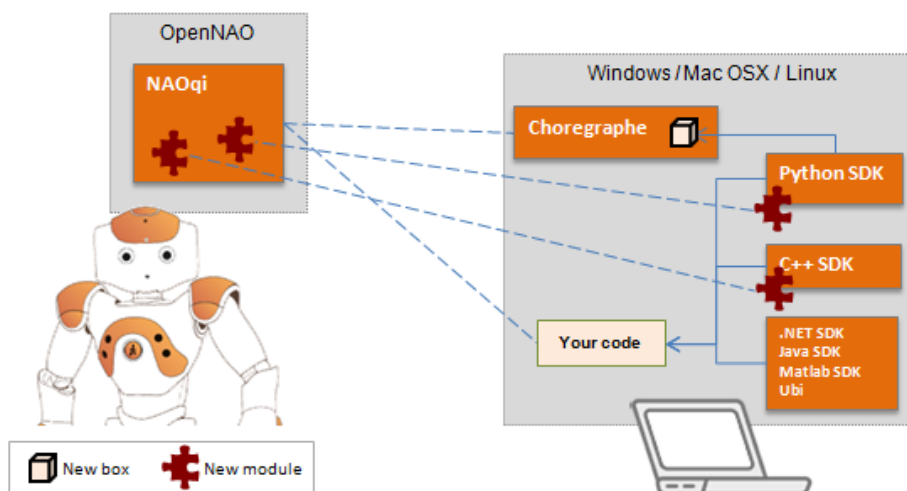


Abb. 10: Nao Programmiersprachen

Dafür bringt Nao seine eigene Bibliothek für andere Programmiersprachen mit. Dies funktioniert in Visual Studio wie in Abb. 11 zu sehen ist. In diesem Beispiel wird der Nao den Text „Hello World from c sharp“ sprechen.

```
using Aldebaran.Proxies;

class Program
{
    static void Main(string[] args)
    {
        TextToSpeechProxy tts = new TextToSpeechProxy("<IP of your robot>", 9559);
        tts.say("Hello World from c sharp");
    }
}
```

Abb. 11: Nao Bibliothek nutzen [23]

TextToSpeechProxy ist eine der Klasse, die in den Bibliotheken integriert sind und ermöglicht es Texte zu definieren, die Nao bei Ausführung des Codes spricht. Andere Klassen sind MotionProxy für frei zu definierende Bewegungen mittels Winkelangabe in Radiant und RobotPostureProxy um vorgefertigte Bewegungen auszuführen [24].

Des Weiteren gibt es bereits vordefinierte Methoden wie z. B. die Methode say() von der Klasse TextToSpeechProxy. Es gibt zwei verschiedene Arten von Methoden: blockierende und nicht-blockierende Methoden. Wird eine blockierende Methode aufgerufen, so erfolgt die weitere Codeabarbeitung erst nach dem Ende der aufgerufenen Methode. Bei einer nicht-blockierenden Methode erfolgt die Abarbeitung parallel. Am meisten ist der Unterschied bei lang andauernden Bewegungen spürbar. In diesem konkreten Fall werden für die Nao-Bewegungen beide Varianten benutzt, wobei auf die Details im weiteren Verlauf eingegangen wird.

2.3.2 Andere humanoide Roboter

Neben dem Roboter Nao gibt es auch andere humanoide Roboter, wie beispielsweise den Roboter Romeo, der ebenfalls von der Firma Aldebaran Robotics hergestellt wird und den bereits genannten Eigenentwicklungen einiger wissenschaftlichen Institute. Da während der Ausarbeitung lediglich mit dem Roboter Nao gearbeitet wurde, werden die anderen humanoiden Roboter nicht weiter betrachtet.

2.4 Entwicklungsumgebung

Microsoft empfiehlt die Nutzung von C++, C# oder Visual Basic für die Kinect, da es für diese Programmiersprachen vorgefertigte Klassen im SDK gibt, auf die zurückgegriffen werden kann. Für den Roboter Nao kann neben C++ und Python ebenfalls C# benutzt werden.

Da sowohl die Kinect, als auch der Nao mit C# arbeiten (können), ist auch die Anwendung in C# programmiert, um so ohne Konvertierung das Spiel zu programmieren. Die Daten werden mittels C# an den Nao übertragen, intern gespeichert und ebenso über C# von der Kinect ausgelesen, um problemlos miteinander verglichen werden zu können.

Als Entwicklungsumgebung wird Visual Studio genutzt, da dies die gängigste Variante ist, um in .NET Sprachen, wie C#, zu programmieren. Dabei unterstützt das Programm den Programmierer im Vervollständigen von bereits bekannten Namen, beim Anlegen von Methodenrumpfen sowie durch eingebaute Tools, zum Beispiel Code Map, um den Code und die Interaktionen der Programmteile schematisch zu betrachten.

Die verwendeten Bibliotheken heißen Aldebaran.Proxies für den Nao und Microsoft.Kinect für die Kamera.

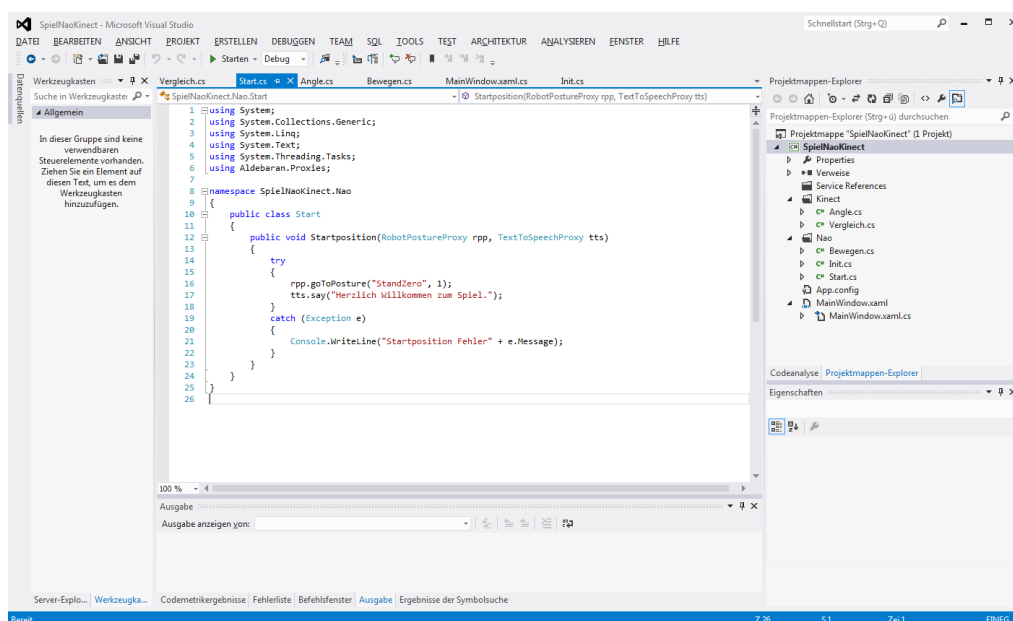


Abb. 12: Visual Studio

2.5 Threads

Häufig benötigen Anwendungen Dinge die parallel geschehen. Um das zu erreichen werden von unterschiedlichen Prozessen Threads gestartet, welche auf die Ressourcen gemeinsam zugreifen können, wie in Abb. 13 zu sehen. Für den Nutzer ist der interne Ablauf irrelevant.

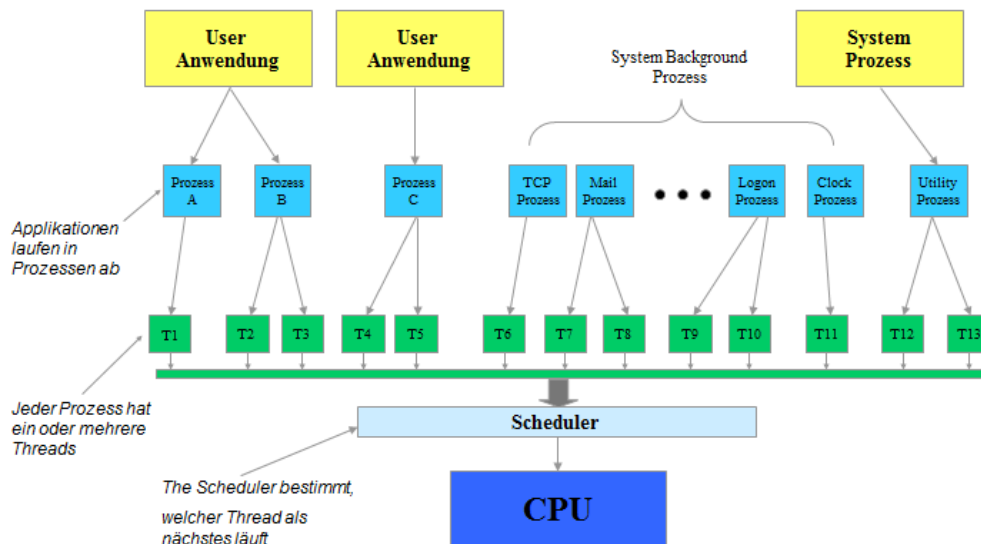


Abb. 13: Threads aus Nutzersicht [25]

Die Threadprogrammierung ermöglicht es demnach unterschiedliche Methoden zeitgleich ablaufen zu lassen, wie beispielsweise bei Servern, die mehrere Anfragen zeitgleich bearbeiten müssen. „Bei einem Ein-Prozessor-System sind das quasi-parallel ablaufende Steuerflüsse innerhalb eines Programms“. Sie werden häufig auch als leichtgewichtige Prozesse („lightweight process“) bezeichnet, wobei jeder dieser Threads seinen eigenen Programmzähler (PC von engl. programm counter), ein eigenes Code-Segment sowie einen eigenen Stack nutzt, wie in Abb. 14 zu sehen. Doch während Prozesse einen eigenen Adressraum haben und nur mittels Betriebssystem-Mitteln interagieren, können Threads auch über statische Variablen kommunizieren.

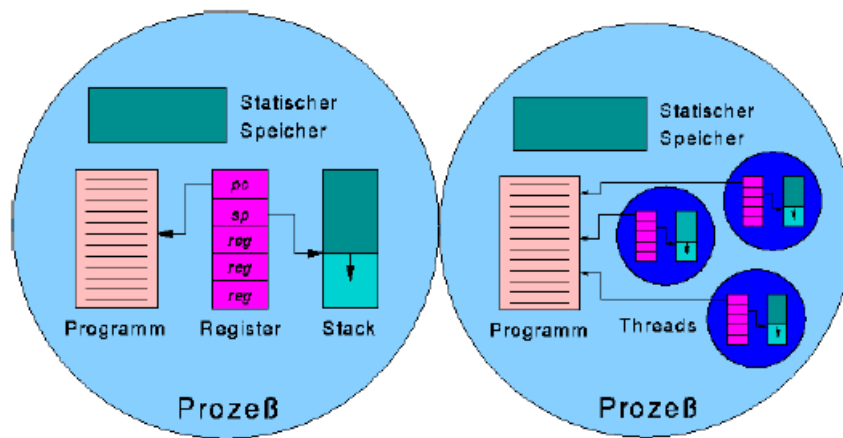


Abb. 14: Threads im Vergleich zu Prozessen [26]

Die Threads können dabei folgende Zustände haben:

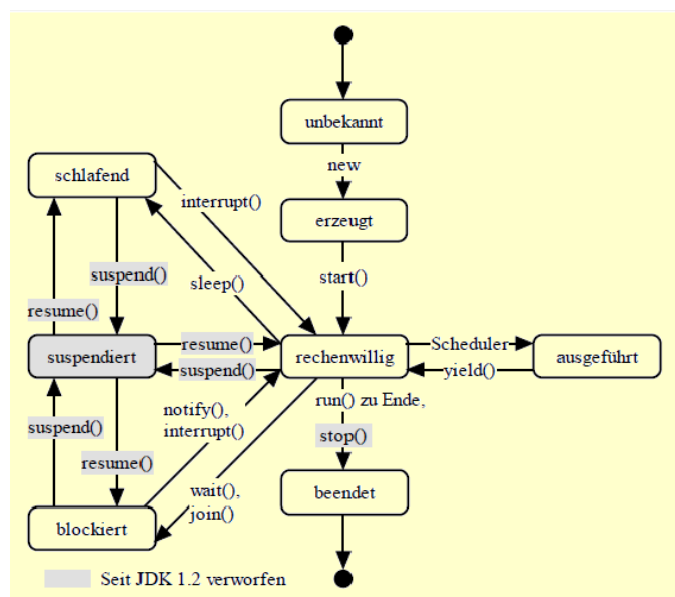


Abb. 15: Zustände von Threads [26]

Er ist in den Zuständen rechenwillig, suspendiert und blockiert ausführbar, sofern ihm die notwendigen Ressourcen zur Verfügung stehen. Im Zustand schlafend wartet er auf ein Ereignis, zum Beispiel das Ende eines Rechenvorgangs [25], [26], [27].

3 Konzeption

Zu Beginn wird die Entwicklungsumgebung eingerichtet, inklusive den SDKs von Kinect und Nao. Dann werden dem Nao einige Bewegungen einprogrammiert, wobei es sich zu Beginn um einfache Bewegungen handelt, damit alle Komponenten auf ihre Funktionstüchtigkeit getestet werden können. Es sollen die entsprechenden Winkel der durchgeführten Bewegung berechnet und an eine Vergleichsklasse übermittelt werden, die diese speichert. Die Kinect sendet dann ihrerseits die erkannten Winkel an die Vergleichsklasse, welche die Winkel des Roboters und der Kinect miteinander vergleicht.

Die Kinect bringt, wie eingangs beschrieben, im Developer Toolkit einige fertige Programmierungen mit, unter anderem eine Programmierung, in der die Skelette von Personen sowie deren Entfernung erkannt werden. Dieses Programm soll als Grundlage dienen, um dem Nutzer seine Bewegungen in einem Kamerabild sichtbar zu machen. Anschließend wird das Programm um die Vergleichsklasse, andere spielrelevante Klassen und Methoden erweitert, so dass die Applikation das Ergebnis des Winkelvergleichs anzeigen kann.

Zu dem soll es die Möglichkeit geben verschiedene Schwierigkeiten im Programm abzubilden. Es bleibt zu überlegen, ob diese verschiedene Schwierigkeitsstufen darüber abgebildet werden, dass es unterschiedlich schwere Bewegungen zum Nachmachen gibt, die stetig komplizierter und/oder länger werden oder über die prozentuale Übereinstimmung der beiden Bewegungen bzw. derer Winkel.

Da für den Menschen kleinere Unterschiede in den Winkeln nicht sichtbar sind, wird beim Vergleich dieser ein Filter eingesetzt werden. Dieser Filter rundet die empfangenen Werte und schließt so Unterschiede im nicht wahrnehmbaren Bereich aus.

Zunächst sollen nur die Winkel des Starts und Endes der Benutzer-Bewegung von der Kinect gemessen und mit den gespeicherten Werten des Naos verglichen werden. Sofern dieser Mechanismus funktioniert, sollen die Bewegungen komplizierter und umfangreicher werden. Zudem sollen dann auch verschiedene Zwischenwerte der Winkel verglichen werden. Dies soll verhindern, dass der Spieler Bewegungen abkürzen und somit schummeln kann.

Insgesamt soll ein Funktionsumfang von zehn verschiedenen Bewegungen implementiert werden. Die Bewegungen werden allerdings lediglich Armbewegungen beinhalten, da es sonst zu Gleichgewichtsproblemen beim Roboter führen kann.

Um die Applikation auf ihre Funktionsfähigkeit zu testen, wird jede der Bewegungen zehn mal hintereinander nachgemacht und die Rate ermittelt, wie viele Wiederholungen der jeweiligen Bewegung als „erfolgreich“ erkannt werden. Auf diesen Ergebnissen aufbauend werden die entsprechenden Filter angepasst.

Um den Rahmen der Applikation ebenfalls spielerisch zu gestalten, werden einige der Instruktionen vom Roboter selbst gesprochen. Außerdem soll der Spieler für eine erfolgreiche Bewegung –je nach Schwierigkeit– eine bestimmte Anzahl an Punkten erhalten, die in der Anwendung sichtbar sind. So kann ein Spieler die erreichten Punkte mit den Punkten anderer Spieler vergleichen.

4 Umsetzung

In diesem Kapitel wird auf die komplette Entwicklung eingegangen. Im ersten Unterkapitel wird die Struktur des kompletten Programms erläutert. Da die Struktur sich größtenteils in zwei Teilgebiete einordnen lässt, wird in den beiden darauffolgenden Unterpunkten die Nao- und Kinect-Programmierung erläutert. Danach werden die Themen GUI und Test beschrieben.

4.1 Programmstruktur / Architektur

Das Spiel *SpielNaoKinect* wird mit der in Abb. 16 gezeigten Code Map beschrieben. Die einzelnen Klassen, die auf der Abbildung zu sehen sind, können zusätzlich noch aufgeklappt werden, sodass auch die verwendeten Methoden und Variablen sichtbar sind.

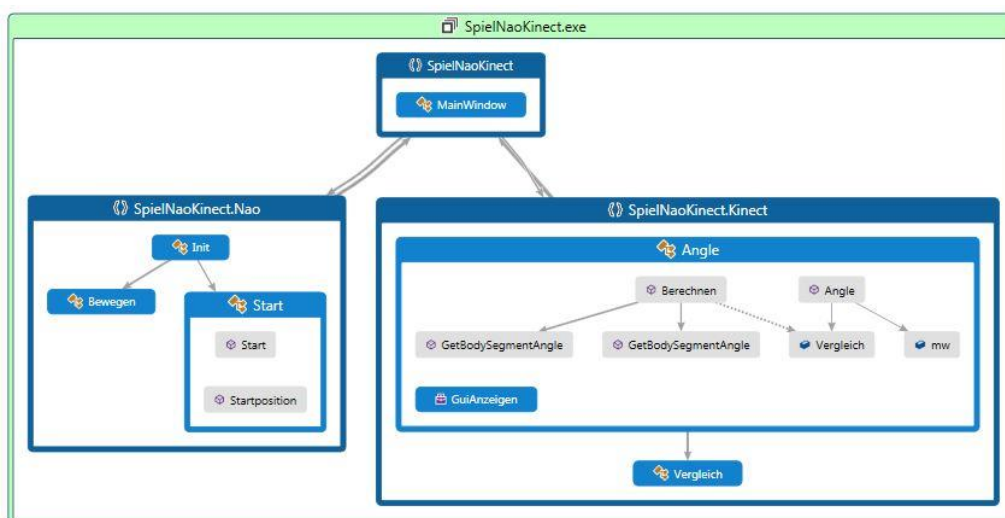


Abb. 16: Code Map

In der Abbildung sind die Klasse *MainWindow* und die Pakete *Nao* und *Kinect* zu sehen. Im Paket *Nao* sind die Klassen *Init*, *Bewegen* und *Start* zu sehen, bei *Start* sind zusätzlich noch die beiden Methoden *Start* und *Startposition* zu erkennen. Zu dem Paket *Kinect* gehören die Klassen *Angle* und *Vergleich*. In der Klasse *Angle* wird auf eine Instanz der Klasse *Vergleich* und des *MainWindows* (*mw*) zugegriffen.

MainWindow:

Die Klasse *MainWindow* ist der Einstiegspunkt des Programms und für die visuelle Darstellung der Anwendung zuständig. Außerdem dient die Klasse als Brücke für den Kinect- und Nao-Teil. Zunächst erfolgt eine Überprüfung der Kinect-Sensoren, also ob die Kinect angeschlossen ist und ob ein Bild dargestellt werden kann. Wenn die Überprüfung erfolgreich verlief, werden die Klassen *Angle* und *Init* initialisiert. Außerdem wird die Methode Initialisierung von der Klasse *Init* aufgerufen. Bei diesem Aufruf wird die IP-Adresse und der Port vom Nao übergeben. Die IP-Adresse und der Port sind hart im Code verankert, jedoch muss bei einer Änderung einer der beiden Werte lediglich eine Stelle im Code angepasst werden.

Die Oberfläche mit all ihren Funktionen wird in Kapitel 4.4 beschrieben.

Init (Nao-Bereich):

Diese Klasse hat drei Methoden:

- *Initialisierung*

Die Methode *Initialisierung* wird nur einmalig beim Start des Programms aufgerufen und ist wie der Name ahnen lässt für die Initialisierung eines MotionProxys, RobotPostureProxy und TextToSpeechProxys zuständig. Des Weiteren werden die Klassen *Start* und *Bewegen* initialisiert und anschließend die Methode *Startposition* von *Start* aufgerufen.

- *Bew_Winkel*

Diese Methode wird aufgerufen, wenn Nao eine Bewegung vorführen soll. Je nach dem ob Nao eine neue Bewegung vormachen soll oder die bereits ausgeführte Bewegung wiederholen soll wird eine vorgegebene Methode in der Klasse *Bewegung* aufgerufen.

- *Bew_Ausgangsposition*

Hierbei wird die Methode *Ausgangsposition* der Klasse *Bewegen* aufgerufen.

Start (Nao-Bereich):

Diese Klasse hat nur die eine Methode *Startposition*. In dieser Methode bewegt sich Nao in die Startposition *StandZero*. Außerdem sagt der Roboter parallel dazu „Herzlich Willkommen zum Spiel“.

Bewegen (Nao-Bereich):

Bewegen ist für die Durchführung einer Roboterbewegung zuständig. Alle Methoden dieser Klasse werden entweder von *Bewegen* selbst oder von *Init* aufgerufen.

Wird beispielsweise die Methode *Ausgangsposition* aufgerufen, so geht Nao wieder in seine Standardposition *StandZero*, wie auch beim Spielstart.

Wenn eine neue Bewegung ausgeführt werden soll, dann wird eine Bewegung per Zufall ausgewählt und der Roboter führt diese vorgegebene Bewegung mittels des erzeugten *MotionProxys* aus.

Angle (Kinect-Bereich):

Diese Klasse kommt währenddessen der Benutzer die Bewegung von Nao nachmachen soll in einer *while*-Schleife zum Einsatz. Die *while*-Schleife, die für zehn Sekunden ausgeführt wird, ruft immer wieder die Methode *Berechnen* auf. In dieser Methode werden die aktuellen *Joints* von der vor der Kinect stehenden Person berechnet. Mithilfe dieser *Joints* können danach die Winkel bestimmt werden und es erfolgt das Weiterreichen dieser Werte an die Klasse *Vergleich*.

Vergleich (Kinect-Bereich):

In dieser Klasse gibt es für jeden Winkel, der verglichen wird, eine extra Methode:

- *Achsel_links_roll*
- *Achsel_rechts_roll*
- *Achsel_rechts_pitch*
- *Achsel_links_pitch*
- *Ellenbogen_rechts_roll*
- *Ellenbogen_lins_roll*

In allen Methoden erfolgt zunächst eine Anpassung der übergebenen Kinect-Winkel, sodass die Koordinatensysteme von der Kinect und vom Nao übereinstimmen. Letztendlich erfolgt der Abgleich, ob die beiden Winkel nahezu identisch sind.

4.1.1 Programmablauf bei einer Bewegungsnachahmung

Dieses Kapitel erläutert den programminternen zeitlichen Ablauf, wenn der Spieler eine Bewegung nachmachen möchte. Der Programmablaufplan wurde in zwei Bilder gesplittet, sodass die zwei wichtigen Teile getrennt voneinander betrachtet werden können.

Neben jeder Aktion ist in roter Farbe der Klassenname notiert, damit die Zuordnung zwischen der Aktion und einer konkreten Klasse nochmals verdeutlicht wird.

Bei den beiden Abbildungen wird zusätzlich auf die gesamte Thread-Struktur eingegangen. In dunkelblauer Farbe ist das Starten eines Threads gekennzeichnet bzw. die Überprüfung ob ein Thread fertig ist. Damit ersichtlich ist, was in einem Thread passiert, sind alle Aktionen eines Threads in hellblauer Farbe umrandet und der dazugehörige Threadname ebenfalls in hellblau angegeben.

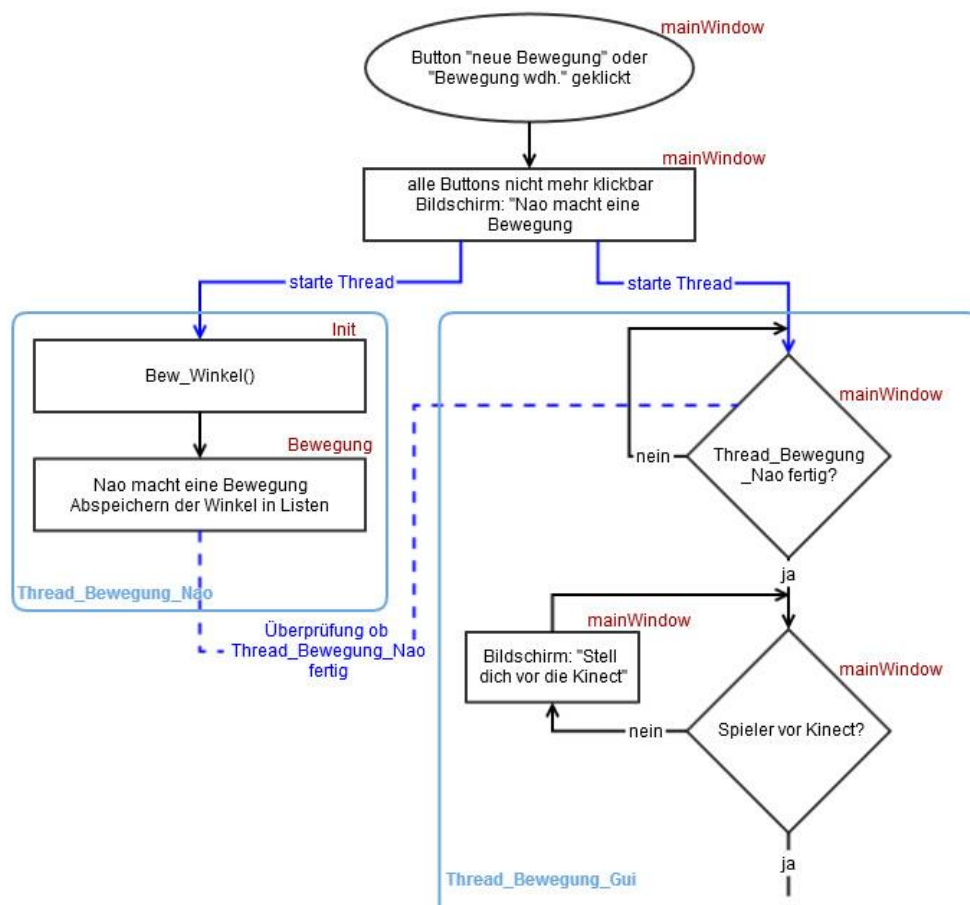


Abb. 17: Programmablaufplan – erster Teil

Wenn ein Spieler eine Bewegung von Nao nachmachen möchte, muss auf der GUI entweder auf den Button „neue Bewegung“ oder auf den Button „Bewegung wdh.“ geklickt werden. Dieser Klick ist auch gleichzeitig der Start des internen Ablaufs. Auf der grafischen Oberfläche des Spiels erscheint nun der Text „Nao macht eine Bewegung“ und alle Buttons sind ausgegraut. Danach werden zwei Threads (Thread_Bewegung_Nao und Thread_Bewegung_GUI) gestartet.

Im Thread_Bewegung_Nao wird die Methode Bew_Winkel() aus der Klasse *Init* aufgerufen und dann in der Klasse *Bewegung* die Nao-Bewegung ausgeführt. Währenddessen die Bewegung ausgeführt wird, werden verschiedene Winkel in Listen gespeichert. Wenn die Bewegung vom Nao fertig ist, beendet sich dieser Thread automatisch.

Thread_Bewegung_GUI ist zwar schon gestartet, wartet aber in einer while-Schleife bis der parallel-laufende Thread beendet ist und erst dann erfolgt der weitere Ablauf, da ansonsten schon vor Beendigung der Bewegung des Nao der Timer laufen und die Kinect Winkel abfragen würde. Im nächsten Schritt wird überprüft, ob die Kinect einen Spieler erkennt. Falls dies nicht zutrifft, erscheint auf dem Bildschirm „Stell dich vor die Kinect“ und es wird gewartet, bis ein Spieler erkannt wird. Anschließend erfolgt der weitere Ablauf, der nun mit dem zweiten Programmablaufplan (Abb. 18) erläutert wird.

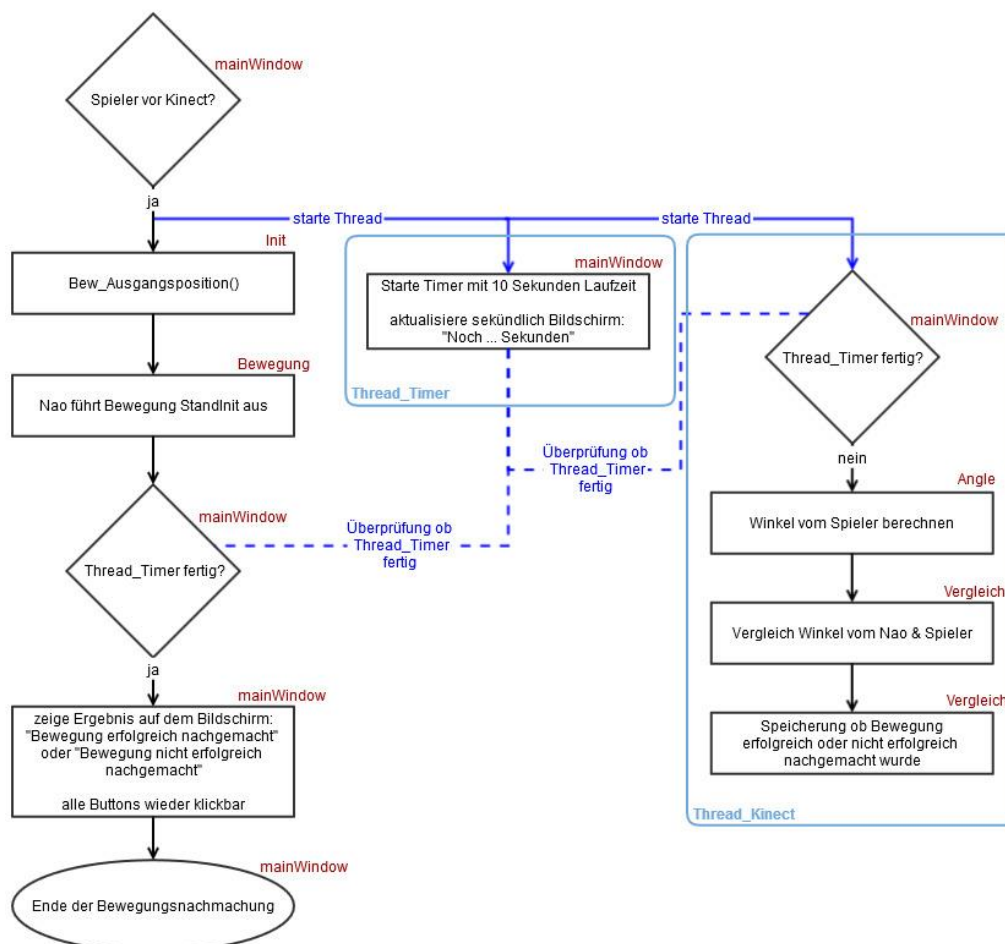


Abb. 18: Programmablaufplan – zweiter Teil

Bis zum Anfang dieser Abbildung wurde die Bewegung vom Nao ausgeführt, die Winkel dieser Bewegung gespeichert und es wurde ein Spieler von der Kinect erkannt.

Nun werden erneut zwei Threads (Thread_Timer und Thread_Kinect) gestartet. Ab diesem Zeitpunkt werden nun drei Aktionen parallel ausgeführt.

In der Klasse *Init* wird die Methode *Bew_Ausgangsposition()* aufgerufen, woraufhin Nao wieder in seine Ausgangsposition *StandZero* geht. Hierbei spielt es eine wichtige Bedeutung, dass die aufgerufene Funktion *goToPosture(„StandZero“, 1)* eine nicht-blockierende Methode ist. Nämlich nur so wird sichergestellt, dass der gesamte Ablauf parallel durchgeführt werden kann. An dieser Stelle Nao wieder in seine Ausgangsposition zu stellen hat den Vorteil, dass jede Bewegung mit der gleichen Ausgangsposition startet. Nach der durchgeführten Bewegung kommt eine Schleife, die solange ausgeführt wird, bis der Thread_Timer fertig ist.

In Thread_Timer (mittlere Spalte der Abb. 18) wird ein Timer mit einer Laufzeit von 10 Sekunden gestartet. In diesem Thread wird dann auch immer die grafische Oberfläche aktualisiert, auf der dann die verbleibende Dauer des Timers angezeigt wird.

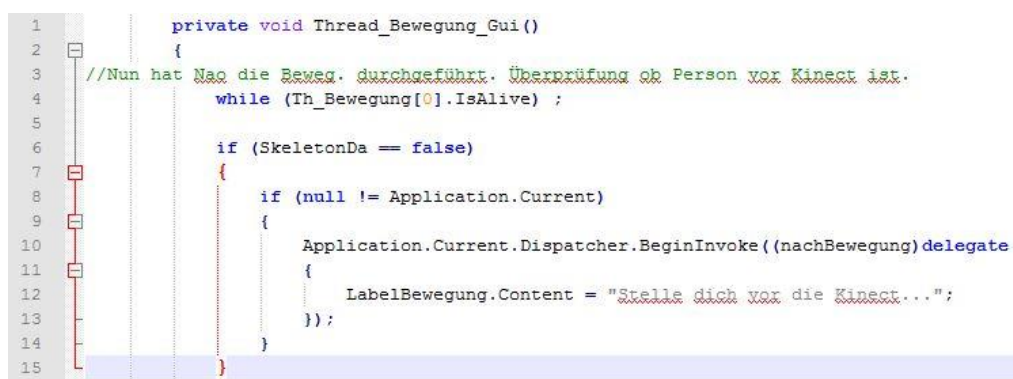
Die dritte Parallelität (Thread_Kinect; rechte Spalte der Abb. 18) wird solange ausgeführt, bis der Timer fertig ist. In diesem Thread werden in der Klasse *Angle* die aktuellen Winkel des Spielers berechnet und diese Werte an die Klasse *Vergleich* übergeben. Dort werden dann die Winkel vom Nao, die in Listen vorliegen, mit den Spieler-Winkeln verglichen. Der letzte Schritt dieses Threads ist die Speicherung, ob die Bewegung erfolgreich oder nicht erfolgreich nachgemacht wurde.

Sobald die 10 Sekunden abgelaufen sind, also der Timer fertig ist, erfolgt die weitere Abarbeitung des Hauptzweigs (linke Spalte der Abb. 18). Es werden die vorherig gespeicherten Werte verglichen und dem Ergebnis entsprechend, sagt Nao „Glückwunsch! Du hast die Bewegung erfolgreich nachgemacht.“ oder eben „Leider hast du die Bewegung nicht erfolgreich wiederholt. Klicke auf Bewegung wiederholen, um es noch einmal zu versuchen.“. Der letzte Schritt ist nun die Buttons wieder zu aktivieren, sodass der Benutzer im nächsten Schritt eine neue Aktion auswählen kann. Somit ist die komplette Bewegungsnachahmung inklusive der Auswertung vollendet.

4.1.2 Threadeinsatz

Im Programmcode werden, wie in den beiden Programmablaufplänen (Abb. 17 und Abb. 18) zu sehen, an zwei verschiedenen Stellen Threads verwendet.

An manchen Stellen ist es wichtig zu wissen, ob ein bestimmter Programmabschnitt bereits beendet ist oder noch ausgeführt wird. Hierzu werden die beiden Threads Thread_Bewegung_Nao und Thread_Bewegung_Gui aus Abb. 17 betrachtet. Beide Threads werden, wie oben beschrieben, gestartet und laufen ab diesem Zeitpunkt parallel. Jedoch soll Thread_Bewegung_Gui erst weitere Aktionen ausführen, sobald Thread_Bewegung_Nao fertig ist.



```

1 private void Thread_Bewegung_Gui()
2 {
3     //Nun hat Nao die Beweg. durchgeführt. Überprüfung ob Person vor Kinect ist.
4     while (Th_Bewegung[0].IsAlive) ;
5
6     if (SkeletonDa == false)
7     {
8         if (null != Application.Current)
9         {
10             Application.Current.Dispatcher.BeginInvoke((nachBewegung)delegate
11             {
12                 LabelBewegung.Content = "Stelle dich vor die Kinect...";
13             });
14         }
15     }

```

Code 1: Warten auf das Beenden eines anderen Threads

In diesem Codestück ist Zeile 4 für das Warten verantwortlich. Der Code wird bis zur vierten Zeile parallel zum Code von Thread_Bewegung_Nao ausgeführt. Der Befehl `Th_Bewegung[0].IsAlive` gibt wahr zurück, wenn der Thread_Bewegung_Nao aktiv ist. Also wird in der while-Schleife solange gewartet bis der Thread nicht mehr ausgeführt wird. Erst wenn dies eintritt wird auch der Code unterhalb Zeile 4 abgearbeitet.

Das Konstrukt in den Zeilen 8 bis 10 ist nötig, um auch von anderen Threads Zugriff auf gemeinsame Ressourcen zu haben. In diesem Fall wird auf die GUI zugegriffen und der Text „Stelle dich vor die Kinect...“ erscheint.

Im zweiten Fall sollen ein Timer –inklusive einer Oberflächenanpassung– und die Bewegungserkennung des Spielers –inklusive eines Winkelvergleichs– parallel ablaufen. Aus diesem Grund werden auch hier zwei Threads benötigt.

In Thread_Timer wird ein Timer heruntergezählt und sekundlich die GUI angepasst. Während diesem Zeitraum soll gleichzeitig die Kinect die Bewegungen des Spielers aufnehmen, die aktuellen Winkel speichern und diese mit den gespeicherten Roboterwinkeln vergleichen (siehe Abb. 18).

```
1 Th_Spieler = new Thread[2];
2 Th_Spieler[0] = new Thread(new ThreadStart(Thread_Timer));
3 Th_Spieler[1] = new Thread(new ThreadStart(Thread_Kinect));
4 Th_Spieler[0].SetApartmentState(ApartmentState.STA);
5 Th_Spieler[1].SetApartmentState(ApartmentState.STA);
6 Th_Spieler[0].Start();
7 Th_Spieler[1].Start();
```

Code 2: Starten von zwei Threads

In der ersten Zeile von Code 2 **Fehler! Verweisquelle konnte nicht gefunden werden.** wird ein Threadarray mit einer Größe von 2 erzeugt. In Zeile zwei wird dem nullten Element des Arrays der Thread „Thread_Timer“ zugewiesen. In der nächsten Zeile erfolgt das gleiche mit dem ersten Element des Arrays und „Thread_Kinect“.

Damit Threads auch Zugriff auf die anderen parallel laufenden Threads haben, müssen sie vom Typ Singlethreaded Apartment (STA) sein. Dies geschieht in Zeile 4 und 5. Ein Apartment ist ein logischer Container, in dem alle Objekte (in Code 2 Thread_Timer und Thread_Kinect) gespeichert werden, welche dieselben Anforderungen an den Threadzugriff haben. Somit können beide Threads Aufrufe des anderen Threads empfangen. Außerdem hat dies den Vorteil, dass die Ressourcen threadsicher verwendet werden.

Im nächsten Schritt (Zeile 6 und 7) werden beide Threads mit der Start()-Methode gestartet. Nun kann die parallele Abarbeitung von `private void Thread_Timer(){...}` und `private void Thread_Kinect(){...}` beginnen [28], [29].

4.2 Nao

Das Paket Nao ist in folgende Klassen unterteilt:

- Init
- Bewegen
- Start

In diesen Klassen befinden sich die Methoden, die sich um die Anfängliche Initialisierung des Nao, die Zufälligkeit der Bewegung und um das Ausführen der zufällig gewählten Bewegung bzw. das mehrmalige Ausführen derselben Bewegung kümmern. Im Folgenden wird beschrieben, wie eine zufällige Bewegung ausgeführt wird und wie eine Bewegung wiederholt werden kann. Anschließend wird eine Bewegung beispielhaft erklärt und abschließend wird auf die Speicherung der Winkel eingegangen.

4.2.1 Zufällige Bewegung ausführen

Insgesamt gibt es fünfzehn vorgegebene Bewegungen für den Roboter Nao. Wenn der Benutzer eine neue Bewegung vom Roboter vorgemacht bekommen möchte, wird folgender Code ausgeführt:

```
1  private int Bewegungsnummer;  
2  public void Bewegung_erzeugen()  
3  {  
4      Random r = new Random();  
5      Bewegungsnummer= r.Next(1, 16);  
6      Bewegung();  
7  }  
8  public void Bewegung()  
9  {  
10     //...  
11     switch (Bewegungsnummer)  
12     {  
13         case 1:  
14             //Bewegung 1  
15             break;  
16         case 2:  
17             //Bewegung 2  
18             break;  
19         //...  
20         case 15:  
21             //Bewegung 15  
22             break;  
23     }  
24 }  
25
```

Code 3: Ausführen einer zufälligen Bewegung (vereinfacht)

In der ersten Zeile wird der Integer „Bewegungsnummer“ definiert. Aus der Klasse Init wird die Methode `Bewegung_erzeugen()` aufgerufen, welche eine zufällige Zahl bestimmt, um somit eine zufällige Bewegung auszuwählen. In Zeile 4 wird die Zufallsklasse initialisiert und anschließend eine zufällige Zahl von eins bis fünfzehn ermittelt und in der Variable „Bewegungsnummer“ gespeichert. Danach wird in die Methode `Bewegen()` gesprungen. Ab Zeile elf erfolgt dann eine switch-case-Anweisung die als Übergabeparameter die vorherige Zufallszahl „Bewegungsnummer“ erhält und dementsprechend in die richtige Bewegung

springt. Wurde beispielsweise durch die Random-Funktion Bewegungsnummer auf sechs gesetzt, dann wird auch die sechste Bewegung ausgeführt.

4.2.2 Bewegung wiederholen

Möchte der Spieler, dass Nao nochmals die letzte Bewegung vorführt, muss er den Button „Bewegung wdh.“ betätigen. Dieser Fall ist ähnlich zu dem Fall aus Kapitel 4.2.1, bei dem eine zufällige Bewegung ausgeführt wurde. Da sich die Integer-Zahl „Bewegungsnummer“ seit dem letzten Aufruf nicht mehr geändert hat, muss nur die Methode `Bewegung()` ausgeführt werden (vergleiche Code 3). Das hat letztendlich zur Folge, dass Nao nochmals die identische Bewegung ausführt. Um bei dem vorherigen Beispiel zu bleiben, würde also nochmals die Bewegung Nummer sechs ausgeführt werden.

4.2.3 Beispiel einer spezifischen Bewegung

```

1  string[] Joints = { "LElbowRoll", "LShoulderRoll", "LShoulderPitch",
2    "RElbowRoll", "RShoulderRoll", "RShoulderPitch" };
3
4  float[] Winkel1 = { UmrechnungDegRad(0), UmrechnungDegRad(0), UmrechnungDegRad(90),
5    UmrechnungDegRad(0), UmrechnungDegRad(0), UmrechnungDegRad(90) };
6  float[] Winkel2 = { UmrechnungDegRad(0), UmrechnungDegRad(75), UmrechnungDegRad(0),
7    UmrechnungDegRad(0), UmrechnungDegRad(-75), UmrechnungDegRad(0) };
8  float[] Winkel3 = { UmrechnungDegRad(-85), UmrechnungDegRad(75), UmrechnungDegRad(0),
9    UmrechnungDegRad(85), UmrechnungDegRad(-75), UmrechnungDegRad(0) };
10
11  int IDMotion1 = motion.post.angleInterpolationWithSpeed(Joints, Winkel1, 0.08f);
12  int IDMotion2 = motion.post.angleInterpolationWithSpeed(Joints, Winkel2, 0.08f);
13  int IDMotion3 = motion.post.angleInterpolationWithSpeed(Joints, Winkel3, 0.08f);

```

Code 4: Eine vordefinierte Bewegung von Nao

Im ersten Schritt in Code 4 werden die Namen der verschiedenen Winkel in der vom Nao-SDK vorgegebenen Schreibweise angegeben und in der Liste `Joints` gespeichert. Der Winkelname `LElbowRoll` entspricht dem Roll-Winkel des linken Ellenbogens, die anderen Winkel entsprechend.

Im zweiten Schritt –ab Zeile 4– werden die konkreten Winkel angegeben. Da Nao die Winkel in der Einheit Radiant benötigt, aber die meisten Menschen besser mit der Einheit Grad umgehen können, erfolgt bei jedem Winkel die Umrechnung von Grad in Radiant. Die Umrechnung ist wie folgt implementiert worden, die mathematische Formel zur Berechnung wurde bereits im Grundlagenkapitel erklärt:

Kommentar [D1]: hier auch auf blocking call eingehen. Wie sind die Winkel angegeben? Wie bewegt sich Nao dann? Umrechnung Rad -> Deg

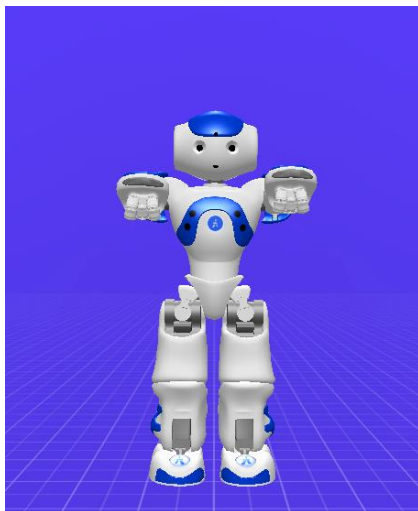
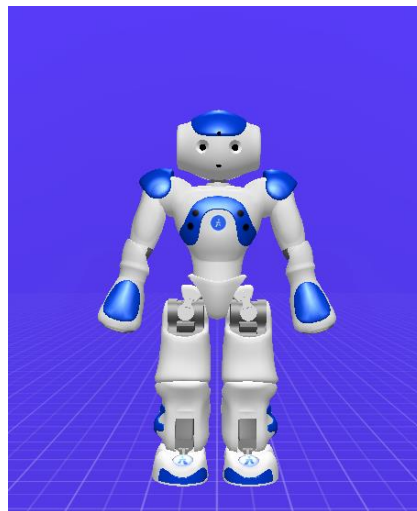
mit Bilder Nao in Folge
 1)Startposition
 2)Arme nach Außen
 3)Arme anwinkeln

goToPosture=non-blocking
 angleInterpolationWithSpeed=blocking


```
1  
2  
3  
4  
5  
-  
  
public float UmrechnungDegRad(int degree)  
{  
    radiant = (float)(degree * Math.PI / 180);  
    return radiant;  
}
```

Code 5: Umrechnung von Gradzahl in Radiant

Somit genügt es den Winkel im Gradmaß anzugeben und durch die Umwandlung erhält der Roboter den dazugehörigen Radianten. Die Reihenfolge der Winkel ist hierbei sehr wichtig, da der erste Winkel (0 Grad) dem ersten Winkelnamen (*LElbowRoll*) zugeordnet ist. So werden die unterschiedlichen Winkeln für eine Bewegung für jeden Zwischenschritt in den Listen „Winkel1“, „Winkel2“ und „Winkel3“ abgespeichert. In den Zeilen elf bis dreizehn werden dann die Joints, die konkreten Winkel und die Geschwindigkeit, mit der die Bewegung durchgeführt werden soll, angegeben. Hierzu wird die blockierende Methode *angleInterpolationWithSpeed* aufgerufen. Dieser Schritt führt dazu, dass sich Nao von seiner Startposition zu „IDMotion1“, „IDMotion2“ und schließlich zu seiner Endposition, „IDMotion3“, bewegt. Es ist wichtig, dass eine blockierende Methode verwendet wird, damit die einzelnen Positionen nacheinander und nicht parallel zueinander ausgeführt werden. Durch den Code 4 führt Nao folgende Bewegung aus:

**Abb. 19: Startposition: StandZero****Abb. 20: IDMotion1: beide Arme unten**

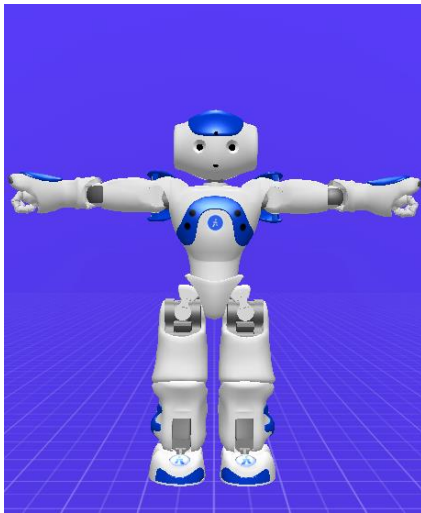


Abb. 21: IDMotion2: beide Arme außen

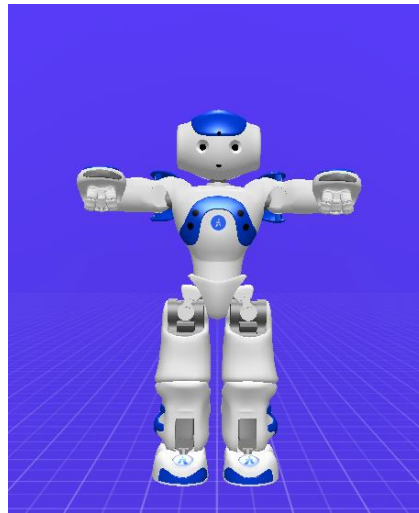


Abb. 22: IDMotion3: beide Ellenbogen angewinkelt

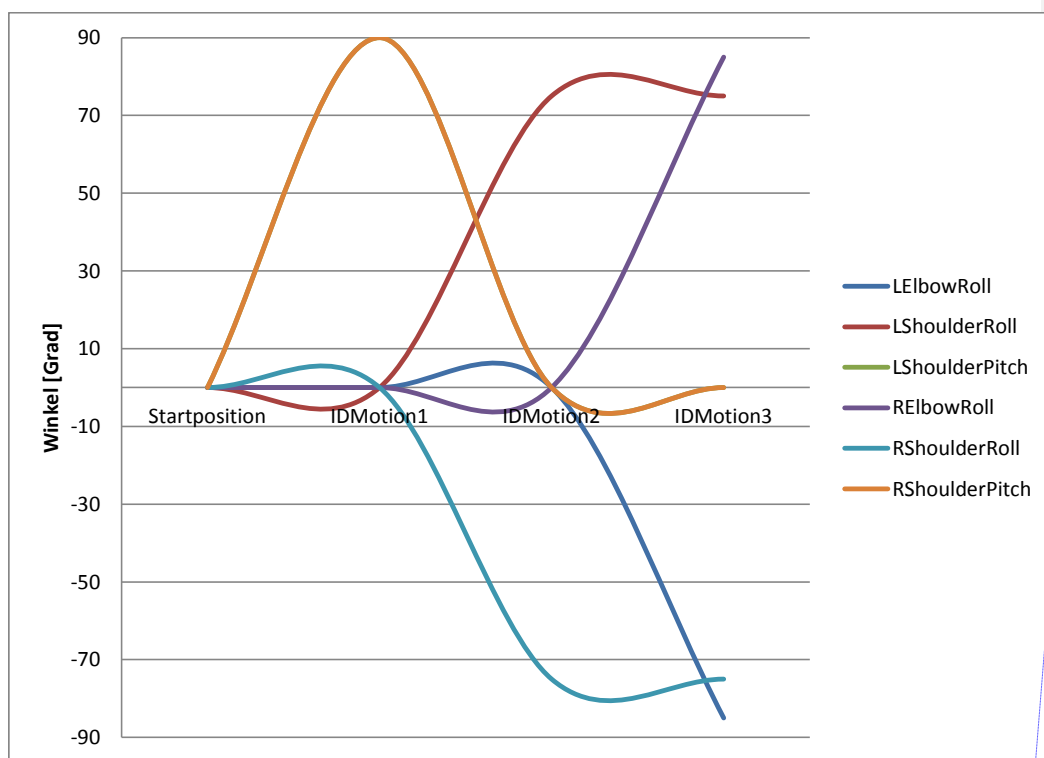


Abb. 23: Diagramm der unterschiedlichen Winkel bei einer Bewegungsausführung

Kommentar [D2]: HILFE

In Abb. 23 sind die unterschiedlichen Winkel des eben beschriebenen Bewegungsablaufs dargestellt. So ist zu sehen, dass bei der Startposition alle Winkel auf null sind. Dies ist das Merkmal für die eingangs beschriebene Position *StandZero*. Bei *IDMotion1* haben sich dann die Winkel *LShoulderPitch* und *RShoulderPitch* um jeweils 90 Grad verändert, wie auch in Abb. 20 zu sehen ist. Diese werden im darauffolgenden Schritt beide null, während *LShoulderRoll* und *RShoulderRoll* sich jeweils um 75 Grad geändert (siehe Abb. 21). Die unterschiedlichen Vorzeichen kommen dadurch zu Stande, dass ein gemeinsames Koordinatensystem verwendet wird. Bei der vorherigen Bewegung haben beide Arme sich in die gleiche Richtung bewegt, bei der jetzigen Bewegung verändern sie sich in entgegengesetzte Richtung. Von *IDMotion2* zu *IDMotion3* bleiben die beiden *ShoulderRoll* und *ShoulderPitch* Winkel auf dem gleichen Wert, die Winkel *LElbowRoll* und *RElbowRoll* verändert sich jeweils um 85 Grad (siehe Abb. 22).

4.2.4 Speicherung Winkel vom Nao


```
mw._LShoulderPitch.Add(UmrechnungRadDeg(motion.getAngles("LShoulderPitch", false).Last()));
```

Code 6: Speicherung der Winkel vom Nao

Code 6 zeigt, wie alle Werte, die zu einem Winkel gehören, während einer Bewegungsausführung ausgelesen und gespeichert werden. Dies geschieht mit mehreren Zwischenstufen, auf die im Folgenden eingegangen wird.

Zu Beginn wird der hellblau umrandete Teil betrachtet: Der Methodenaufruf `motion.getAngles()` wird vom Hersteller mitgeliefert und gibt eine Liste von Winkeln zurück, auf deren letztes, und damit neustes, Element mit `.Last()` zugegriffen wird. Dafür muss der Methode übergeben werden, um welchen Winkel es sich handelt, in diesem Fall *LShoulderPitch*. Außerdem muss die boolische Variable `false` übergeben werden, damit Nao den Winkel an der korrekten Stelle berechnet. Andernfalls würde er Winkel zu seiner Umgebung zurückgeben, was sinnvoll ist, wenn statt einem Joint der *Body* übergeben wird.

Da Nao mit Radianten arbeitet, besteht die Liste auch aus Radiant-Werten. Deshalb wird die einzelnen Werte der Liste an die Methode `UmrechnungRadDeg()` übergeben, welche diese in Gradzahlen umwandelt (grün umrandeter Teil). Der entsprechende Code ist in Code 7 abgebildet.



```
1 public int UmrechnungRadDeg(float radiant)
2 {
3     degree = Convert.ToInt32(radiant * 180 / Math.PI);
4     return degree;
5 }
```

Code 7: Umrechnung von Radiant in Gradzahl

Im orangefarbenen Teil wird die errechnete Gradzahl in der Liste `_LShoulderPitch` abgespeichert. Diese Liste wurde im `MainWindow (mw)` initialisiert, weshalb hier mit dem Punktoperator drauf zugegriffen wird. Diese Liste dient später der Klasse `Vergleich` als Grundlage und muss daher auch von anderen Stellen zugreifbar sein. Letztendlich enthält die Liste je nach Komplexität und Dauer der Bewegung um die 100 Werte.

In diesem Beispiel wurde der Fall `LShoulderPitch` behandelt, die anderen fünf Winkelspeicherungen erfolgen analog dazu.

4.3 Kinect

Im Laufe der Entwicklung stellte sich heraus, dass das im Toolkit Developer mitgelieferte Programm sehr komplexe Strukturen aufweist, von denen die wenigsten tatsächlich benötigt wurden. Aus diesem Grund wurde eine neue Anwendung programmiert, in die der `SkeletonStream` aus einem Lehrbuch integriert ist [11].

4.3.1 Berechnung Winkel Kinect

Die mit der Kinect aufgezeichneten Winkel des Skelettes werden mit Hilfe der in Kapitel 2.2.2 beschriebenen Verfahren errechnet. Dafür werden zuerst verschiedene Joints aus dem Skelett ausgelesen und diese einer Methode zum Berechnen übergeben:

```

1 // Berechnung mit 3 Joints
2 Vergleich.Achsel_links_pitch(GetBodySegmentAngle
3     (ElbowLeft, ShoulderLeft, HipLeft));
4 Vergleich.Achsel_rechts_pitch(GetBodySegmentAngle
5     (ElbowRight, ShoulderRight, HipRight));
6 Vergleich.Ellenbogen_rechts_roll(GetBodySegmentAngle
7     (ShoulderRight, ElbowRight, WristRight));
8 Vergleich.Ellenbogen_links_roll(GetBodySegmentAngle
9     ShoulderLeft, ElbowLeft, WristLeft));
10
11 //Die beiden Achsel Roll mit 4 Joints
12 Vergleich.Achsel_rechts_roll(GetBodySegmentAngle
13     (HipRight, HipLeft, ShoulderRight, ElbowRight));
14 Vergleich.Achsel_links_roll(GetBodySegmentAngle
15     (HipLeft, HipRight, ShoulderLeft, ElbowLeft));

```

Code 8: Jointübergabe für Winkel

Für den Winkel der bei der Achsel (Achsel pitch) liegt werden die Gelenke, Joints, vom Ellbogen, der Schulter und der Hüfte verwendet. Der Winkel zur Bewegung Armbeuge, Ellbogen roll, nutzt die Joints Schulter, Ellbogen und Handgelenk. Und für die Berechnung des Achsel Roll werden die Gelenke Schulter, Ellbogen und beide Hüftseiten übergeben.

Abhängig von der Anzahl der übergebenen Joints (nämlich drei Joints für zwei zusammenhängende Knochen bzw. vier Joints für zwei getrennt liegende Knochen) werden hier unterschiedliche Methoden aufgerufen. Am Ende der Berechnung wird der Wert in beiden Methoden in eine 64-bit Integer Zahl konvertiert und zurückgegeben.

```

1 public long GetBodySegmentAngle(Joint joint1, Joint joint2, Joint joint3)
2 {
3     Vector3D vectorJoint1ToJoint2 = new Vector3D
4         (joint1.Position.X - joint2.Position.X, joint1.Position.Y - joint2.Position.Y, 0);
5     Vector3D vectorJoint2ToJoint3 = new Vector3D
6         (joint2.Position.X - joint3.Position.X, joint2.Position.Y - joint3.Position.Y, 0);
7     vectorJoint1ToJoint2.Normalize();
8     vectorJoint2ToJoint3.Normalize();
9
10    Vector3D crossProduct = Vector3D.CrossProduct(vectorJoint1ToJoint2, vectorJoint2ToJoint3);
11    double crossProductLength = crossProduct.Z;
12    double dotProduct = Vector3D.DotProduct(vectorJoint1ToJoint2, vectorJoint2ToJoint3);
13    double segmentAngle = Math.Atan2(crossProductLength, dotProduct);
14
15    // Convert the result to degrees.
16    double degrees = segmentAngle * (180 / Math.PI);
17
18    degrees = degrees % 360;
19    return Convert.ToInt64(degrees);
20 }

```

Code 9: Berechnung von zusammenhängenden Knochen

In Zeile 3 und 5 werden die übergebenen Joints zu Vektoren verbunden, welche Knochen darstellen. Die beiden Knochen sind zusammenhängend, da der Joint2 in beiden Knochen enthalten ist. Diese werden anschließend (in Zeile 7 und 8) normalisiert mittels Methodenaufruf `.Normalize()`. Danach werden Kreuzprodukt sowie Punktprodukt der Vektoren berechnet und mittels Tangens der Winkel zwischen den Knochen ausgerechnet. Als Rückgabewert der Methode wird ein Integer verwendet, da die Gradzahlen bis zu diesem Zeitpunkt als Fließkommazahl vorliegen und diese Genauigkeit zum Winkelvergleich irrelevant ist. Durch die Umwandlung werden die Zahlen hinter dem Komma abgeschnitten und somit in Ganzzahlen konvertiert.

```
1 public long GetBodySegmentAngle(Joint joint1, Joint joint2, Joint joint3, Joint joint4)
2 {
3     Vector3D vectorJoint1ToJoint2 = new Vector3D
4         (joint1.Position.X - joint2.Position.X,
5          joint1.Position.Y - joint2.Position.Y,
6          joint1.Position.Z - joint2.Position.Z);
7     Vector3D vectorJoint3ToJoint4 = new Vector3D
8         (joint3.Position.X - joint4.Position.X,
9          joint3.Position.Y - joint4.Position.Y,
10         joint3.Position.Z - joint4.Position.Z);
11     vectorJoint1ToJoint2.Normalize();
12     vectorJoint3ToJoint4.Normalize();
13
14     double dotProduct = Vector3D.DotProduct(vectorJoint1ToJoint2, vectorJoint3ToJoint4);
15     double segmentAngle = Math.Acos(dotProduct);
16
17     // Convert the result to degrees.
18     double degrees = segmentAngle * (180 / Math.PI);
19
20     degrees = degrees % 360;
21     return Convert.ToInt64(degrees);
22 }
```

Code 10: Berechnung von getrennten Knochen

Der Code 10 berechnet die Winkel von nicht zusammenhängenden Knochen, was auch daran zu erkennen ist, dass keiner der Joints in mehr als einem Vektor verwendet wird. Diese Berechnung ist notwendig, da der Winkel *ShoulderRoll* nicht zuverlässig mit den Gelenken von Ellbogen und beiden Achseln berechnet wird. Da die Kinect insbesondere beim Ändern der Armhaltung von „nach unten gestreckt“ und „nach vorne gestreckt“ immer wieder zu falschen Werten kam, wurde ein vierter Referenzpunkt benötigt. Aus diesem Grund nutzen wird der Vektor genutzt, der bei den Joints Ellbogen und Achsel entsteht und den Vektor der zwischen dem linken und dem rechten Hüftknochen liegt.

4.3.2 Winkelvergleich Kinect - Nao

```
1 // Achtung! Steckt in einer WHILE Schleife drin!
2 public void Achsel_links_roll(long degrees)
3 {
4     degrees -= 90;
5     for (int i = 0; i < mw._LShoulderRoll.Count(); i++)
6     {
7         if (degrees >= (mw._LShoulderRoll[i] - mw.Schwierigkeit)
8             && degrees <= (mw._LShoulderRoll[i] + mw.Schwierigkeit))
9         {
10             mw._LShoulderRoll.RemoveAt(i);
11         }
12     }
13     if (mw._LShoulderRoll.Count() == 0)
14     {
15         mw.Achsel_links_roll_erreicht = true;
16     }
17 }
```

Code 11: Winkelvergleich

Wie oben beschrieben, sind die Werte des Nao in einer Liste im MainWindow gespeichert. Über diese Liste wird mittels for-Schleife (Zeile 5) iteriert und betrachtet, ob der erste Wert in der aktuell laufenden Bewegung erreicht wird (Zeile 7). Auf die dort stehende Schwierigkeit wird im nächsten Kapitel genauer eingegangen.

Falls der Spieler den Wert in seiner Bewegung erreicht, wird dieser mit dem Aufruf `.RemoveAt(i)` in Zeile 9 aus der Liste entfernt. Da diese Operation in einer While-Schleife liegt (Zeile 1), wird erneut das erste Element der Liste überprüft.

Ob die Liste leer ist, wird in Zeile dreizehn überprüft. Ist dies der Fall, so wird die Variable „erreicht“ auf `true` gesetzt (Zeile 15). Nach Ablauf des Timers werden alle entsprechenden sechs Werte abgefragt. Sind alle Werte auf `true`, so wurde die Bewegung erfolgreich wiederholt und Nao teilt das Ergebnis mit den Worten „Glückwunsch! Du hast die Bewegung erfolgreich wiederholt“ mit.

Falls nicht, sagt Nao „Leider hast du die Bewegung nicht richtig wiederholt [...]“ und der Spieler bekommt die Möglichkeit einen weiteren Versuch zu starten.

In beiden Fällen kann der Spieler jetzt eine neue Bewegung anfordern.

In Kapitel 3 war angedacht, zuerst Start- und Endposition miteinander zu vergleichen. Da die Winkel aber in einer gesamten Liste gespeichert werden und es so einfach möglich ist, jeden Wert zu vergleichen, wurde auf diesen Zwischenschritt verzichtet.

4.3.3 Schwierigkeitsgrade

Es ist möglich unterschiedliche Schwierigkeitsgrade auszuwählen. Folgende Grade stehen dabei zur Auswahl:

- Leicht
- Mittel
- Schwer

Entgegen der Möglichkeiten, die im Kapitel Konzeption genannt wurden, sind die Schwierigkeiten über ein zu erreichendes Intervall implementiert worden. Dabei entspricht „leicht“ einer Abweichung vom Winkel des Naos um 20 Grad, „mittel“ einer Abweichung um 15 Grad und „schwer“ einer Abweichung von 10 Grad, wie es in Code 12 zu sehen ist:

```
1 private void RB_Leicht_Click(object sender, RoutedEventArgs e)
2 {
3     Schwierigkeit = 20;
4 }
5
6 private void RB_Mittel_Click(object sender, RoutedEventArgs e)
7 {
8     Schwierigkeit = 15;
9 }
10
11 private void RB_Schwer_Click(object sender, RoutedEventArgs e)
12 {
13     Schwierigkeit = 10;
14 }
```

Code 12: Schwierigkeitsgrad

Demnach wird bei der Überprüfung der Winkel wie in Code 11 beschrieben, nicht betrachtet, ob der Wert exakt erreicht wird, sondern lediglich, ob er – je nach Schwierigkeit – in einem Bereich von +/- 10, 15 bzw. 20 Grad liegt. Liegt in der Liste also der Wert 45 Grad, so wird dieser bei der Schwierigkeitsstufe leicht aus der Liste gelöscht sobald die Kinect an dieser Stelle einen Winkel zwischen 25 und 65 Grad übermittelt.

4.4 GUI

Die GUI wie in Abb. 24 zu sehen besteht aus dem Bild der Kinect, in der zu sehen ist, wie der Nutzer die Bewegung nachmacht. Dabei wird sein Skelett schematisch dargestellt. Auf der rechten Seite befinden sind drei Buttons:

- „Neue Bewegung“

- „Bewegung wdh.“
- „Neues Spiel“

Die Buttons „Bewegung wdh.“ und „Neues Spiel“ sind ausgegraut, bis die erste Bewegung abgeschlossen ist.

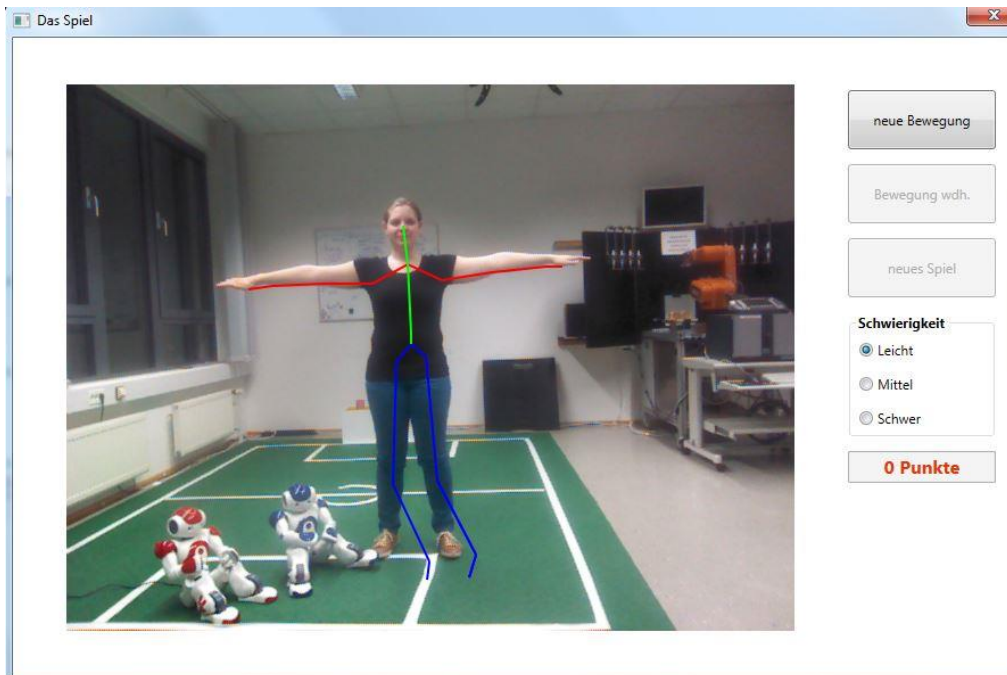


Abb. 24: GUI zum Spiel

Während eine neue Bewegung vom Nao ausgeführt wird oder die letzte Bewegung wiederholt wird, ist in der GUI der Text „Der Roboter macht eine Bewegung vor“ zu sehen. Nach dem der Roboter den Spieler aufgefordert hat, die Bewegung nachzumachen, erscheint auf der GUI der Text „Mache die Bewegung nach: noch X Sekunden“. Der Timer beträgt zehn Sekunden. In dieser Zeit hat der Spieler die Möglichkeit die Bewegung nachzumachen, parallel dazu geht der Roboter in die Ausgangsposition *StandZero* zurück. Je nach Korrektheit der nachgemachten Bewegung erhält der Spieler Punkte, welche unterhalb der Radiobuttons zu sehen sind.

Für eine erfolgreiche Bewegung im Schwierigkeitsgrad „leicht“ erhält der Spieler 10 Punkte, bei der Schwierigkeit „mittel“ 15 Punkte und im „schweren“ Grad 20 Punkte.

Durch Klicken auf „Neues Spiel“ werden die bisher gesammelten Punkte auf null gesetzt und der gleiche oder ein anderer Spieler kann versuchen, eine höhere Punktzahl zu erreichen, dabei sind die Bewegungen selbstverständlich weiterhin zufällig.

4.5 Test

Jede der 15 Bewegung wurde 10 mal vor der Kinect getestet. Die Ergebnisse sind in folgender Grafik zu sehen:

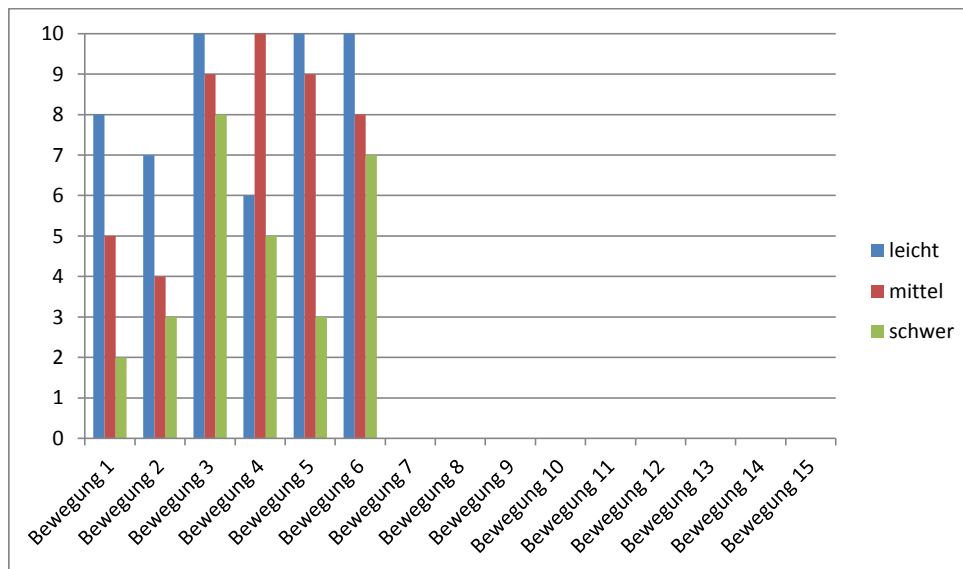


Abb. 25: Testergebnisse

Durch das Testen der verschiedenen Bewegungen wurde nochmals überprüft, ob das Nachmachen der einzelnen Bewegungen machbar ist und ob die Schwierigkeit gut gehandhabt ist.

Einige der Bewegungen zeigen untypische Ausreiser in der Grafik. Eine dieser Bewegungen ist Bewegung 4, bei der der Roboter seinen rechten Arm zur Stirn führt. Beim erstmaligen Testen wurde die Bewegung falsch wiederholt, da die zu testende Person die Bewegung falsch verstanden hat und dadurch den Arm nicht genug nach oben bewegt hat. Dies wurde bei der Schwierigkeit mittel erkannt, weshalb es zu diesem untypischen Ergebnis gekommen ist.

4.6 Benutzung des Programms

Um das Spiel zu nutzen, werden eine Kinect für Windows, der Roboter Nao bzw. die Simulation mittels Webots sowie die dazugehörigen Treiber benötigt. Beim erstmaligen Starten muss die Setup.exe ausgeführt werden, eventuell ist hierbei ein Neustart nötig. Zum Starten des Spiels genügt es anschließend die ClickOnce Anwendung zu starten. Es erscheint ein Fenster zur Eingabe der IP-Adresse des Naos (siehe Abb. 26). Diese beträgt beim roten 192.168.100.3, beim blauen Nao 192.168.100.4 und für die Simulation 127.0.0.1.

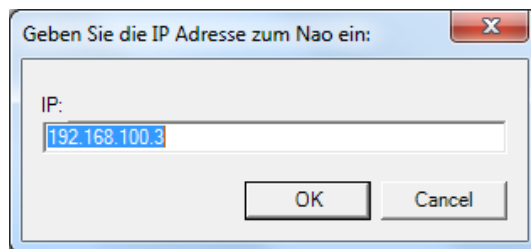


Abb. 26: Abfrage der IP-Adresse

Nach dem das Programm fertig initialisiert ist, begrüßt Nao mit den Worten „Herzlichen Willkommen zum Spiel“. Mit dem Klick auf „Neue Bewegung“ macht der Nao eine Bewegung vor. Nach Abschluss der Demonstration spricht der Roboter die Worte „Nun mache die Bewegung nach“ und der Spieler erhält sein Zeitintervall um die Bewegung zu wiederholen. Ist diese richtig, erhält der Spieler die entsprechende Punktzahl, falls nicht kann er auf „Bewegung wdh.“ klicken und erhält erneut die Chance. Eine neue Bewegung erhält der Spieler in beiden Fällen über den Button „Neue Bewegung“. Ein Spielerwechsel kann durch den Button „Neues Spiel“ erfolgen und zum Beenden der Anwendung genügt es, das Konsolenfenster zu schließen.

5 Fazit

Die Ziele der Studienarbeit ein Spiel mit Nao und Kinect zu programmieren, bei dem der Roboter Bewegungen vormacht und ein Spieler diese nachmacht wurden vollständig erfüllt. Lediglich auf die in der Konzeption erwähnten Filter wurde verzichtet, da das entsprechende Phänomen über die Schwierigkeit mittels Intervallen gut abgefangen wird.

Das Spiel läuft flüssig, auch über mehrere Bewegungen inklusive Spielerwechsel hinweg, lediglich der Roboter wird nach einigen Bewegungen zu heiß oder verliert an Motorenkraft, welches jedoch Probleme des Alters sind beziehungsweise von Herstellerseite herrühren. Wenn der Abstand zur Kinect wesentlich zu kurz gewählt ist (wie in Kapitel 2.1.1 beschrieben), wird eine entsprechende Ausnahme geworfen. Diese Ausnahme wurde in diesem Programm nicht berücksichtigt, da sie einfach vermieden werden kann.

Diese beziehungsweise eine ähnliche Applikation kann in verschiedenen Anwendungsszenarien eingesetzt werden. Zum Beispiel, wie eingangs erwähnt, zum Lernen von Schwimmbewegungen. Vergleichbar hierzu ist auch die Anwendung als Tanzlehrer oder Physiotherapeut. Auch einige der hier programmierten Bewegungen zielen auf die Koordination von mehreren parallelen Bewegungsabläufen des Menschen ab. Neben Koordinationsübungen sind aber auch Kraftübungen denkbar. Zum Beispiel durch das Vormachen, wie Hanteln richtig gestemmt werden. Das Programm könnte hierbei auch gleich überprüfen, ob die nachgemachte Bewegung unter physiologischen Gesichtspunkten gut nachgemacht wurde, das heißt, ob zum Beispiel die Körperhaltung gesund war. Dieses Spiel wird seine Anwendung in Vorführungen bei Öffentlichkeitsveranstaltungen der Dualen Hochschule finden.

Das Spiel kann auf unterschiedliche Arten erweitert werden. Eine Erweiterungsmöglichkeit ist die Einführung einer dauerhaften Highscore-Liste. So würde das Spiel an sich unterschiedliche Spieler in Beziehung zueinander setzen und der jeweilige Spieler kann einschätzen, in welchem Bereich seine erreichten Punkte liegen. Außerdem ist ein Highscore ein Ansporn, um weitere Bewegungen durchzuführen und sich stetig verbessern zu wollen.

Damit die Bewegungen nicht langweilig werden, können mehr Bewegungen eingeführt werden.

Alternativ könnte die Bewegung an sich erweitert werden, in dem auch andere Teile des Körpers, wie die Beine oder Hüfte, hinzugezogen werden. Dies würde wiederum die Anzahl an verfügbaren Bewegungen erhöhen.

Als letzte Möglichkeit das Spiel zu erweitern, ist es auch möglich, es mit der Gestensteuerung zu kombinieren. So kann der Mensch eine Bewegung vormachen, diese würde gespeichert und vom Roboter nachgemacht werden und wäre anschließend im Spiel vorhanden.

Literaturverzeichnis

- [1] Karlsruher Institut für Technologie, „Imitationslernen in der Robotik,“ 2012. [Online]. Available: <http://www.hyperraum.tv/tag/autonome-roboter/>. [Zugriff am 15 01 2014].
- [2] Wikipedia, „Kinect,“ 10 01 2014. [Online]. Available: <http://de.wikipedia.org/wiki/Kinect>. [Zugriff am 15 01 2014].
- [3] Microsoft, „Kinect für Windows,“ [Online]. Available: <http://www.microsoft.com/en-us/kinectforwindows/>. [Zugriff am 15 01 2014].
- [4] Golem, „SDK - Software Development Kit,“ [Online]. Available: <http://www.golem.de/specials/sdk/>. [Zugriff am 15 01 2014].
- [5] J. Roßberg, „Alle Informationen zur neuen Bewegungssteuerung,“ 01 11 2010. [Online]. Available: <http://www.gamepro.de/xbox/spiele/xbox-360/kinect-adventures/artikel/kinect,46324,1967600.html>. [Zugriff am 15 01 2014].
- [6] Computer Bild, „Technische Details zur Bewegungserkennung,“ Computer Bild, 30 06 2010. [Online]. Available: <http://www.computerbild.de/artikel/cbs-News-Xbox-360-Kinect-3D-Kamera-technische-Details-5405436.html>. [Zugriff am 01 04 2014].
- [7] Microsoft, „Problembehebung bei der Bewegungserkennung,“ Microsoft, [Online]. Available: <http://support.xbox.com/de-DE/xbox-360/kinect/body-tracking-troubleshoot>. [Zugriff am 01 04 2014].
- [8] ifixit.com, „Kinect ohne Gehäuse,“ 05 01 2012. [Online]. Available: http://wiki.zimt.uni-siegen.de/fertigungsautomatisierung/index.php/Datei:S910310_abb2.png. [Zugriff am 10 03 2014].
- [9] Microsoft, „Tracking Users with Kinect Skeletal Tracking,“ [Online]. Available: <http://msdn.microsoft.com/en-us/library/jj131025.aspx>. [Zugriff am 10 03 2014].
- [10] Steffen, „Einsatzmöglichkeiten einer 3D-Kamera in der Produktionstechnik am Beispiel der Kinect-Kamera,“ 13 02 2013. [Online]. Available: http://wiki.zimt.uni-siegen.de/fertigungsautomatisierung/index.php/Einsatzm%C3%B6glichkeiten_einer_3D-Kamera_in_der_Produktionstechnik_am_Beispiel_der_Kinect-Kamera. [Zugriff am 01

04 2014].

- [11] T. Hanna, in *Microsoft Kinect - Programmierung des Sensorsystems*, Heidelberg, dpunkt.verlag GmbH, 2013, p. 4.
- [12] Microsoft, „Downloads Kinect for Windows,“ Microsoft, [Online]. Available: <http://www.microsoft.com/en-us/kinectforwindowsdev/Downloads.aspx>. [Zugriff am 13 04 2014].
- [13] B. Vaughan, „www.teacherschoice.com.au,“ [Online]. Available: http://www.teacherschoice.com.au/maths_library/angles/angles.htm. [Zugriff am 27 April 2014].
- [14] „RC-Heli,“ [Online]. Available: <http://wiki.rc-heli-fan.org/index.php/Steuerfunktionen>. [Zugriff am 28 April 2014].
- [15] M. Weis, „uni-hohenheim.de,“ 28 August 2012. [Online]. Available: <http://sengis.uni-hohenheim.de/uas.de.php>. [Zugriff am 28 April 2014].
- [16] M. A. Knus. [Online]. Available: <http://www.math.ethz.ch/~knus/geometrie/1.pdf>. [Zugriff am 28 April 2014].
- [17] N. Demuth, „Die Welt,“ 27 08 2013. [Online]. Available: <http://www.welt.de/wissenschaft/article119430009/Mit-solchen-Maschinen-werden-wir-alleine-sein.html>. [Zugriff am 05 03 2014].
- [18] Die Zeit, „zeit.de,“ 04 12 2013. [Online]. Available: <http://www.zeit.de/wirtschaft/unternehmen/2013-12/google-roboter-android-erfinder-andy-rubin>. [Zugriff am 05 03 2014].
- [19] Fraunhofer Institut, „Fraunhofer Institut - Humanioide Roboter,“ [Online]. Available: <https://www.iosb.fraunhofer.de/servlet/is/5093/>. [Zugriff am 05 03 2014].
- [20] Aldebaran Robotics, „Aldebaran Robotics,“ [Online]. Available: www.aldebaran.com/en/robotics-company/history. [Zugriff am 23 April 2014].
- [21] Aldebaran Robotics, „Aldebaran Robotics Nao Software Documentation,“ [Online]. Available: https://community.aldebaran-robotics.com/doc/1-14/family/robots/motors_robot.html. [Zugriff am 06 03 2014].
- [22] Aldebaran Robotics, „Aldebaran Robotics,“ [Online]. Available: <http://www.aldebaran-robotics.com>.

- robotics.com/en/Discover-NAO/Key-Features/hardware-platform.html. [Zugriff am 06 03 2014].
- [23] Aldebaran Commuity, [Online]. Available: <https://community.aldebaran-robotics.com/doc/1-14/dev/sdk.html>. [Zugriff am 05 Mai 2014].
- [24] Community Aldebaran Robotics, „AldebaranRobotics,“ [Online]. Available: <https://community.aldebaran-robotics.com/doc/1-14/cpp-classindex.html>. [Zugriff am 23 April 2014].
- [25] C. Döpmeier, 03 April 2013. [Online]. Available: <http://www.iai.fzk.de/~clemens.duepmeier/betriebssysteme/Prozesse.ppt>. [Zugriff am 28 April 2014].
- [26] H.-J. Haubner, 2013.
- [27] M. Becker, Dezember 2002. [Online]. Available: <http://www.ijon.de/comp/tutorials/threads/threads.html>. [Zugriff am 28 April 2014].
- [28] Microsoft, „ApartmentState-Enumeration,“ Microsoft, [Online]. Available: <http://msdn.microsoft.com/de-de/library/system.threading.apartmentstate.aspx>. [Zugriff am 09 04 2014].
- [29] Galileo Computing, „Multithreading mit der Klasse Thread,“ Galileo Computing, 2010. [Online]. Available: http://openbook.galileocomputing.de/visual_csharp_2010/visual_csharp_2010_11_002.htm#mj5eadcfd43437a62e48102a560638faea. [Zugriff am 09 04 2014].

Alle Abbildungen, Tabellen, o.ä., die keine Literaturangabe enthalten, sind eigene oder firmenintern verwendete Darstellungen.