KINECT SA

Technischer Bericht

Änderungsnachweis

Version	Status	Datum	Beschreibung	Autor
0.1	Erstellung	14.12.2012	Merge aus bestehenden Einzeldokumenten	Josua Schmid
0.2	Revision	14.12.2012	Abschnitt Zieldefinition	Josua Schmid
0.3	Merge	16.12.2012	Merge mit Architekturdokument	Josua Schmid

Inhaltsverzeichnis

1. Allgemein

1.1 Aufgabenstellung

Kinect als Eingabekonsole eines Industriepanels

Studiengang:	Informatik (I)
Semester:	HS 2012/2013 (17.09.2012-17.02.2013)
Durchführung:	Studienarbeit
Fachrichtung:	Software
Institut:	IFS: Institut für Software
Gruppengrösse:	2 Studierende
Status:	zugewiesen
Verantwortlicher:	Augenstein, Oliver
Betreuer:	Augenstein, Oliver
Gegenleser:	[Nicht definiert]
Experte:	[Nicht definiert]
Industriepartner:	M&F Engineering

Ausschreibung:

Bei der Bedienung von Maschinen kann die Benutzung von Touchscreens ungeeignet sein, wenn die Benutzer (Maschinisten) bei Ihrer Arbeit normaler Weise stark verschmutzte Hände haben. Auch die Maus ist in solchen Fällen oft nur eine zweitklassige Eingabemethode.

In der Arbeit soll untersucht werden, in wieweit die Kinect als Eingabedevice in einem solchen Umfeld in Frage kommt.

Aufgabe der Semesterarbeit ist es zunächst die wichtigsten Funktionen einer Maus durch die Kinect zu ersetzen und danach zu untersuchen, welche zusätzlichen Anwendungsmöglichkeiten sich durch den Einsatz der Kinect in diesem Umfeld ergeben. Z.B. können in der Arbeit auf Gesten basierende Authorisierungsmechanismen entworfen und untersucht werden.

Ziel der Arbeit ist in einem ersten Schritt die Entwicklung eines Prototyps, durch den die neue

1.2 Erklärung über die eigenständige Arbeit

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbarrt wurde
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Rapperswil, den 21.12.2012

Renato Bosshart Josua Schmid

1.3 Abstract

Herkömmliche Industriepanels sind oft ungeeignet für den Realeinsatz. Zum Beispiel benutzen Arbeiter oft Handschuhe oder haben schmutzige Hände - das macht die Bedienung Touch-Panels oder Tastaturen schwierig. Das Ziel dieser Arbeit ist zu zeigen, dass die Microsoft Kinect solche Panels ersetzen kann. Dazu wurde anhand einer Evaluation von geeigneten Gesten und der Kinect-Rahmenbedingungen ein Gestenerkennungs-Framework entwickelt. Sein Zweck ist das stabile Erkennen von Gesten unter den in einer Werkshalle üblichen Störfaktoren.

Die Architektur der Software setzt vorwiegend auf Events. Das garantiert optimale Flexibilität, hatte jedoch zur Folge, dass Speicherlecks analysiert und gelöst werden mussten. Weitere Herausforderungen waren die Zuweisung von Kinect-Skeletten zu eindeutigen Personen, sowie Dauer von Gesten und die Genauigkeit der Skelette. Zur Lösung dieser Probleme mussten jeweils Heuristiken gesucht werden, die sowohl der niedrigen Komplexität der Architektur sowie der leichten Benutzbarkeit der API gerecht werden.

2. Management Summary

In Zusammenarbeit mit M&F Engineering wurde untersucht ob und wie die Microsoft Kinect für die stabile Bedienung von Industriepanels geeignet ist. Als Produkt dieser Evaluation wurde eine Programmbibliothek entwickelt welche stabile Benutzereingaben für jegliche .NET-basierte Applikationen ermöglicht. Die Bibliothek wird durch eine einfache API beschrieben. Jene ist komfortabel zu benutzen, ausführlich dokumentiert und gut zu erweitern. Um Firmenkunden einen Überblick über die Möglichkeiten der Bibliothek und der zugehörigen API zu gewähren, wurde eine GUI-Applikation entwickelt, welche die API implementiert und bildhaft zeigt was die Bibliothek kann.

2.1 Umfeld, Vorgehen, Technologie

Die Gestenerkennungs-Software wird typischerweise in einer Werkshalle eingesetzt. Dabei hat das Verwenden der Microsoft Kinect den Vorteil, dass auch Arbeiter mit schmutzigen Händen oder mit Handschuhen ein Panel bedienen können - bei herkömmlichen Touch-Screens oder Tastaturen wäre dies nur schwierig möglich.

Konkret bedient man in unserem Beispiel-Setting einen grossen, weit entfernten Bildschirm von einem gekennzeichneten Ort in der Halle aus. Auf jenem Bildschirm sind z.B. Details über die Tagesproduktion oder den Maschinenstatus bequem einsehbar. Der Arbeiter geht dafür in den für die Bedienung vorgesehenen Bereich, führt die Anmeldegeste aus und hat danach Zugriff auf die bereitgestellten Informationen.

Abbildung 1: Beispiel-Setting in der Werkshalle einer Druckerei[1]

2.2 Zusammenfassung der Ergebnisse, Ausblick

Mit dieser Arbeit steht ein stabiles Gestenerkennungs-Framework für Industrieanwendungen zur Verfügung. Es kann mit wenig Aufwand um eigene Gesten erweitert und so direkt den eigenen Anforderungen angepasst werden.

Der mitgelieferte Prototyp einer Slideshow zeigt wie eine Anwendung aussehen könnte. Wenn man seinen Programmcode anschaut sieht man, wie einfach das Benutzen der Framework-API ist.

[1] Quelle: http://www.ib-hammrich.de/referenzen.php5

4. Anforderungskriterien

4.1 Zieledefinition

Die Aufgabe der Semesterarbeit ist die Beantwortung folgender Frage:

„Ist die Kinect für Industrielösungen einsetzbar?“

Zur Erreichung des Ziels wird ein Prototyp entwickelt, welcher einen ausgewählten Satz an Eingabemethoden demonstriert. Der Prototyp wird hinsichtlich Stabilität analysiert und optimiert. Dazu gehört auch eine technische Grenzwertanalyse, welche die Einsetzbarkeit des Produkts in einem Industrie-Umfeld dokumentiert.

Der zeitliche Ablauf der Semesterarbeit umfasst drei Phasen:

- **Evaluation** bestehender technischer Lösungen und Bedienkonzepte
- **Entwicklung** und Implementierung eines Bedienkonzeptes für einen minimalen Funktionssatz. Das beinhaltet die folgenden Unterschritte pro Bedienkonzept:
    - Implementierung: Grundimplementierung
    - Stabilisierung: Es wird so lange auf Input stabilisiert, bis klar ist, ob das Bedienkonzept industrietauglich ist.
- **Dokumentation** der Ergebnisse

4.2 Anforderungen

4.2.1 Umfeld

Die in der Semesterarbeit zu erstellende Software wird im industriellen Umfeld eingesetzt – konkret in einer grossen Halle. Es sollen damit Bildschirme bedient werden können, die in bis zu 3m in der Höhe angebracht sind. Die Distanz von Bildschirm zum User beträgt etwa 2 - 10 Meter. Die Raumbeleuchtung befindet sich an der Decke – normalerweise Neon oder Halogen. Es könnte aber auch Tageslicht von oben oder der Seite hinzukommen.

In der Halle bewegen sich normalerweise mehrere Personen, teilweise auch kleinere Fahrzeuge. Die Arbeiter haben unter Umständen Schutzkleidung an oder Handschuhe an.

4.2.2 Störfaktoren

Unter Umständen ist die Kinect kleinen Erschütterungen oder Vibrationen ausgesetzt. Durch Maschinen kann es zu Lärm oder Staubemissionen kommen. Ebenso können Wärmequellen in der Halle entstehen und sich allenfalls bewegen. Störende Beleuchtung kann künstlicher oder natürlicher Natur sein. Fabrikbeleuchtung oder -Heizung kann Licht im Infrarot- oder UV-Bereich freisetzen – das gilt auch für direktes Sonnenlicht. Da in Zukunft immer mehr LED-Beleuchtungen eingesetzt werden, müssen auch sie in dieser Arbeit berücksichtigt werden. Abgesehen von Personen und Fahrzeugen kann es auch andere Objekte haben, die sich bewegen können (z.B.: ein Kran).

4.2.3 Konkret

- Ca. 2-10m Distanz zur bedienenden Person
- Personen sollen sich nicht überwacht fühlen (BigBrother-Problem)
- Mehrere Personen können zuschauen, es bedient jedoch immer nur jemand gleichzeitig. Mehrere Personen sollen von der Software zwar generell unterstützt aber nicht konkret implementiert werden.
- Signalisierung der Einsatzbereitschaft des Systems: Sind Inputs möglich? Hat es zu viel Licht? Person zu weit entfernt, etc.
- Anmeldung der zu bedienenden Person (z.B.: vorbeilaufen soll keine Events triggern)
- Automatische und manuelle Abmeldung (z.B.: was passiert beim Schuhebinden)
- Realisierbarkeit mit einer Kinect

- Aktive und passive Nutzer
- Möglichst unabhängig von oben erwähnten Störfaktoren

4.2.4 Workflow

- Ausführbare Aktionen/Gesten:
    - Blättern
    - Scrollen
    - Zoomen
- Einfache Gestik: Personen wollen nicht den „Hampelmann machen“ oder sich exponieren.
    - Eindeutig
    - Einfach lernbar
    - Intuitiv
- Gesten sollten nach maximal zwei Versuchen erkannt werden, irrtümliche Inputs sollten nicht vorkommen.
- Einfache Kalibrierung
- Workflow soll intuitiv und nicht zu träge sein, jedoch auch nicht zu empfindlich: guter Tradeoff gesucht

5. Analyse der Gesten

Damit die Begrifflichkeiten in der Semesterarbeit immer klar sind, wird im Folgenden definiert welche Gesten mit welchen Aktionen verknüpft werden.

Geste	Beschreibung
Push	Eine Bewegung, bei der die Hand schnell nach vorne bewegt wird.
Pull	Die Hand wird von einer Position vorne schnell zum Körper gezogen.
Pinch Zoom	Durch die Bewegung von beiden Händen zueinander wird herausgezoomt, wenn sich die Hände auseinanderbewegen wird hineingezoomt.
Push and Pull Zoom	Durch das Heranziehen eines Objektes wird es grösser, durch das wegstossen kleiner.
Wischen	eine schnelle Handbewegung nach links oder rechts, die Bewegung kommt aus dem Ellbogen, nicht nur aus dem Handgelenk.
Joystick	Es wird die relative Handposition zu einem definierten Punkt des Körpers verwendet. Darum herum gibt es eine Deadzone, in der nichts passiert. Ausserhalb von dieser wird in diese Richtung gescrollt. Die Geschwindigkeit ist abhängig von der Distanz.
Armwinkel	Der Winkel der zwischen Hand, Schulter und Hüftgelenk aufgespannt wird.
Winken	Hin und her bewegen der Hand, die sich oberhalb des Kopfes befinden muss. Die Bewegung kommt aus dem Ellbogen.
Ausgestreckt	Schulter, Ellbogen und Hand bilden eine Linie. Der Arm zeigt vom Körper weg.
Körper lehnen	Eine Position, wobei der Oberkörper nach vorne, hinten, links oder rechts geneigt ist. Die Körperachse ist dabei gekrümmt in dieser Richtung.
Handgeste	Eine Geste, die nur von der Hand ausgeführt wird. Diese Gesten können wir mit unserer Ausgangslage nicht erfassen. Daher werden wir nicht weiter darauf eingehen.
Laser Pointer	Durch Verlängerung einer Controllerachse oder Körperteils erhält man am Durstosspunkt durch die Bildebene einen Punkt wo ein Cursor dargestellt wird.
Absolutes Scrolling	Ein Punkt auf der Bildebene wird fixiert. Das Bild wird genau dem nachfolgenden Bewegungsmuster folgen und der Cursor(falls vorhanden) bleibt auf dem gleichen Bildpunkt bestehen.
Scrolling mit Momentum	Das Bild wird beim Loslassen mit der gleichen Geschwindigkeit weiterbewegt. Mit einer Dämpfung wird es während einem Zeitinterwall abgebremst.
Emulation	Controller wird als Maus und Tastatur emuliert

Tabelle 1: Begrifflichkeiten für Gesten

5.1 Benutzte Gesten anderer Produkte und Projekte

Device	Eigenheiten und Details	Auswahl	Zoom	Scrolling	Sonstiges
Xbox mit Kinect	Individuell für jedes Game, die meisten Games haben Tutorials. Kein allgemeines Konzept	Meist Push, teilweise Handerkennung	2-Hand Pinch	Wischen, teilweise Joystick	Zusätzliche Aktionen mit z.B. bestimmtem Armwinkel Anmeldung mit Winken
Kinect am PC	Noch keine kommerzielle Software, meist spezifische Prototypen oder Emulatoren	individuell			
Interactive Wall	3 Beamer, 1 Kinect Media Center	Push	-	Wischen	Anmeldung mit ausgestreckten Armen, 2 Cursor pro User
FAAST	Maus und Tastatur Emulator	Arm strecken	-	Körper lehnen oder Handposition	
The Leap	PC Steuerung, auch mit Emulation, hohe Genauigkeit	Individuelle Handgeste	2 Finger Zoom oder Push & Pull	Wischen	Noch nicht erhältlich
Sixense	Sehr genau, kurze Distanz, zwei Controller, nur Emulation	Button	Button	Button	Laserpointer
Glove Pie	Wii Controller am PC	Button	-	-	Nicht zuverlässig
Webcam	Objekt oder Handtracking am PC	Handgeste	-	Wischen	Ungenau, störungsempfindlich
3D-Maus	Keine Gesten	Button	Push+Pull	Seitlich bewegen	
Minority-Report	Handgesteuerte Navigation im 3D Raum. Aktionen mit Gesten	Handgeste	2 Hand Pinch oder Handgeste	Handposition	Kleine Gesten
Wii	Ein Controller pro Spieler	Button	-	-	Laserpointer
PS Move	Ein Controller pro Spieler	Button	-	-	Genau und schnell
Eye Toy	Kollisionsdetektion mit Webcam	-	-	-	
Light-Gun	Point and Click	Button	-	-	Keine Positionsdaten
Touchscreen	Weit verbreitet	Klick	2-Finger Pinch	Absolutes Scrolling	Nur 2D

Tabelle 2: Konkurrenzanalyse

5.2 Gestenevaluation

Anhand *Tabelle 2: Konkurrenzanalyse* und eigener Überlegungen wurde entschieden, welche Funktionalität mit welchen Gesten sinnvollerweise für ein Industriepanel umzusetzen sind. Der Evaluierungsprozess folgt aus Zeitgründen der Intuition der Autoren und hat nicht den Anspruch auf Vollständigkeit. Dabei wurde eine Liste von bekannten Gesten aus Mobilfunk- und Spieleindustrie erstellt und analysiert (siehe Anhang 8.1).

Es wurde entschieden, dass folgende Funktionalität mit Gesten umzusetzen sei:

- Anmeldung
- Zoom
- Scrollen/Blättern
- Cursor
- Spezialaktionen
- Abmeldung

Für jene Funktionalität wird im Folgenden begründet, welche Geste jeweils optimalerweise einzusetzen ist.

5.3 Entscheidung für Gesten

Wir haben uns entschieden, zwei verschiedene Bedienmodi einzuführen. Der erste ist die **Standardbedienung** mit den zwei Gesten *Zoom* und *Blättern* und eventuell mehreren Auswahlgesten. Der zweite Modus ist die **Joystick-Bedienung**. Dieser Modus kann verwendet werden um komplexere Bedienungen wie Zeigen auszuführen. Er bildet die Funktionen *Zoom*, *Scrollen*, *Zeigen*, *Auswählen* ab. Die Implementierung dieses Modus hat für uns niedrige Priorität.

Beide Modi werden ergänzt um die *Anmelde*-Geste.

5.3.1 Anmeldung – *Winken*

Hier haben wir uns für Winken entschieden, da diese Geste sehr intuitiv ist und wenige Fehlaktivierungen auslösen wird. Zudem reicht als Bedienhinweis ein Text im GUI.

Slide To Unlock wäre ebenfalls sehr intuitiv gewesen, hätte aber zusätzlichen Platz auf dem GUI benötigt. Im weiteren besteht Konfliktgefahr zu *Swipe*-Geste.

Die anderen vorgeschlagenen Gesten wurden nicht repräsentativen Umfragewerten zufolge als zu unintuitiv, bzw. unbequem empfunden.

5.3.2 Zoom – *Pinch-Zoom*

Aufgrund der Intuitivität haben wir uns für *Pinch-Zoom* entschieden, obwohl beide Hände für die Geste benötigt werden. Die Distanz beider Hände bestimmt den Zoomfaktor. Sobald klar ist, dass gezoomt werden soll ist die Erkennung und die Bedienung einfach. Das Problem besteht darin, den Anfang und das Ende dieser Geste zu erkennen.

Im Joystickmodus wird der Push-/Pull Zoom verwendet – aufgrund der eventuellen Interferenzen mit dem Standardmodus. Sonst kann sind die beiden Modi zu wenig konsistent.

Der *Zoom-Ring* war zwar früher bei analogen Devices vorhanden, ist jedoch im Moment nicht als Geste üblich und könnte deshalb nicht richtig verstanden werden. Auch nicht repräsentative Umfragen haben gezeigt, dass der Zoom-Ring eher für Verwirrung sorgt.

5.3.3 Scrollen/Blättern – *Swipe/Joystick*

Für einfache Applikationen, die keinen Cursor benötigen und wo jeweils nur wenig geblättert werden muss (z.B. PowerPoint) werden wir *Wischen* implementieren. Für komplexere Anwendungen werden wir, falls die Zeit reicht, den Joystick verwenden, da dieser sehr gut erkannt werden kann und wir damit zugleich auch einen Cursor erhalten würden.

Den Oberkörper zu bewegen (vorwärts- und zurücklehnen) ist zu anstrengend und nicht mit allen Arbeitskleidern gut möglich.

5.3.4 Cursor bewegen – *Joystick*

Nur der Joystickmodus wird einen Cursor unterstützen. Die Implementation eines Mausähnlichen Cursors per Geste ist zu aufwendig (siehe Diskussion 8.2). Zudem kann man die Funktionalität eines Cursors in Applikationen meist anders nachbilden.

Dabei können wir die Position innerhalb einer Deadzone einfach für Cursorbewegungen verwenden.

Ein 2D Mapping der Handposition entspricht eigentlich ziemlich stark dem Modus, den wir für den Joystick verwenden, daher ist jene Geste nicht mehr zu beachten.

5.3.5 Auswählen – spezielle Gesten/Nicken

Auch hier sind wir davon abhängig, ob wir uns in einem Bedienmodus befinden, der einen Cursor zur Verfügung hat, oder nicht. Im Joystick-Modus haben wir keine Handgesten mehr zur Verfügung und eine Geste mit der anderen Hand empfinden wir nicht als besonders intuitiv. Deshalb haben wir uns für Nicken entschieden, da es eine gut erkennbare Geste ist.

In einem Bedienmodus ohne Cursor, wo es nur wenige Objekte mit Interaktionsmöglichkeiten geben wird, können wir wenige Gesten auf diese Objekte legen, die wir schon implementiert haben, z.B. Stossen oder Nicken. Für eine finale Software müsste man hier allenfalls noch ein paar wenige zusätzliche Gesten definieren, was den Benutzer aber nicht überfordern darf. Das heisst, dass nicht mehr als 4 Bedienelemente zur gleichen Zeit aktiv sein sollten, andernfalls wird der Benutzer die Übersicht verlieren. Mittels animierter Icons kann man die Nutzer gut auf die zu verwendende Geste hinweisen.

Diese Geste ist sehr wichtig, jedoch nicht ganz einfach umzusetzen, da es bei anderen Technologien dafür immer einen Button oder eine intuitive Lösung gibt. Das ist bei unserer Lösung nicht der Fall. Durch die grosse Distanz wird es zudem unmöglich Handgesten zu erkennen. Deshalb ist es wichtig beim finalen Programm darauf zu achten, dass möglichst wenige Selektionen gemacht werden müssen.

Einen Timer werden wir nicht verwenden, da dies ein stabiles Inputsignal voraussetzt und den uneingeschränkten Fokus des Nutzers.

Thumb-Up wäre sehr intuitiv ist aber auf diese Distanz sicher nicht mehr erkennbar und fällt daher raus.

5.4 Abhängigkeiten der Gesten

Im folgenden Abschnitt wird nur auf die Gesten eingegangen, die wir für unser Projekt verwenden, da es sonst unnötig kompliziert werden würde und andere Gesten für unser Projekt nicht von Belang sind.

Wir ordnen die Gesten in Datengruppen ein. Die Datengruppen beschreiben, welche Daten(Skelett-Punkte) in die Berechnung einbezogen werden müssen.

Hand rechts	• Winken • Joystick • Wischen • Pinch-Zoom • Push • Pull
Hand links	• Pinch-Zoom
Kopf	• Nicken
Relative Position (Abhängig von anderem Körperpunkt)	• Winken • Joystick • Push • Pull
Bewegungsrichtung	• Wischen • Pinch-Zoom
Bewegungsgeschwindigkeit	• Winken • Wischen • Nicken
Unterscheidung Hin- und Rückbewegung	• Wischen (zwei verschiedene Gesten) • Pinch-Zoom (zwei verschiedene Gesten) • Nicken • Winken
Unterscheidung Gestenstart- und Ende	• Wischen • Pinch-Zoom

Tabelle 3: Gestenabhängigkeiten

5.4.1 Bemerkungen

- Für Pinch-Zoom müssen viele Faktoren analysiert werden. Es ist die komplizierteste, jedoch kann man für die anderen sicher am stärksten profitieren.
- Winken ist eher unabhängig zu anderen Gesten. Es besteht eine geringfügige Ähnlichkeit zu Wischen.
- Joystick verwendet nur die relative Position, dafür ist man auf ein stabiles Inputsignal angewiesen. Diese Geste sollt man erst machen wenn man den Input schon stabilisiert hat.
- Wischen ist ähnlich zu Pinch, aber etwas einfacher, da man nur eine Hand auswerten muss.
- Nicken hat eine ähnliche Charakteristik wie Winken, jedoch müssen andere Skelett-Punkte analysiert werden.
- Push/Pull gehören zum Joystick, es ist lediglich eine andere Achse, die analysiert werden muss.

6. Framework – Innere rchitektur

6.1 Überlegungen zur allgemeinen Architektur

6.1.1 Events

Die Architektur der Gestenerkennungssoftware legt den Schwerpunkt auf eventbasierte Programmierung. Dies hat den Vorteil, dass auf Schleifen und Threading verzichtet werden kann. Das verringert die Komplexität der Anwendung. Zur Statussynchronisation können Event-Argument-Objekte verwendet. Dies fördert die Wart- und Anpassbarkeit des Codes. Ein wichtiger Nachteil ist das Memory Management (s.u.).

6.1.2 Berechnungen

Berechnungen an den 3D-Skeletten sind in eigene Klassen gekapselt, was Korrekturen vereinfacht und Duplicated Code verhindert.

6.1.3 Hauptfunktion: Gestenerkennung

Gesten werden vom GestureChecker (GestureChecker-Statemachine) erkannt. Jener überpfüft, ob die Reihenfolge der zu erfüllenden Gestenteile stimmt. Falls die Erkennung erfolreich war, triggert er die vom API-Benutzer für die erfolgreiche Erkennung hinterlegte Funktion und aktiviert die Erkennungsroutine des nächsten Gestenteils. Bei Misserfolg wird ebenfalls eine hinterlegte Benutzer-Funktion aufgerufen, die Gestenerkennung wird jedoch wieder auf Anfang geschaltet.

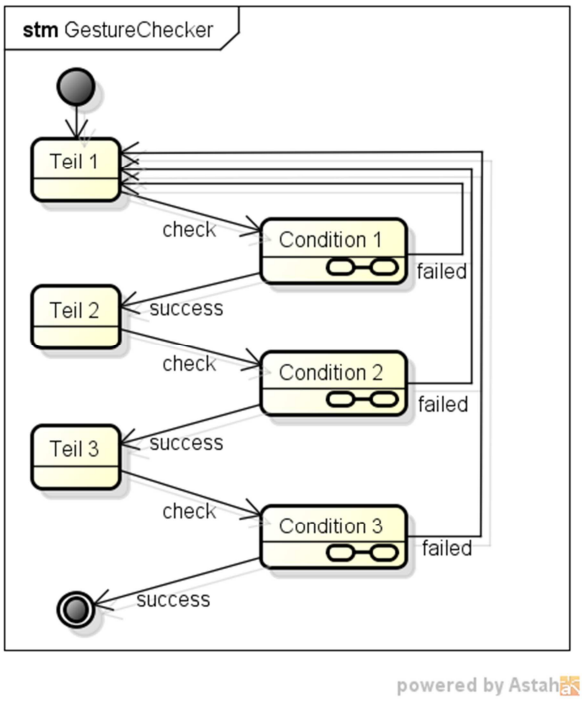


Abbildung 2: Vereinfachter Ablauf des Gestenerkennungsmechanismus

Zusätzlich hat der GestureChecker die Möglichkeit, einen Timeout zu signalisieren. Er hört dabei auf den OnCheck-Event jeder Geste und schaut ob sie in der vorgegebenen Zeit ablief. Er kann jedoch keine laufende Gesten-Überprüfung stoppen – diese Funktionalität hätte die Einfachheit der vorhandenen Architektur gesprengt. Der GestureChecker beschränkt sich bei einem Timeout darauf, die Geste als nicht erfolgreich zu werten und setzt den Status wieder auf den ersten Gestenteil.

6.2 Domain

Die folgende Domainanalyse ist stark vereinfacht und soll einen groben Überblick über die Gestenerkennungssoftware ermöglichen.

Objekt	Beschreibung
Checker	Der Checker benutzt die von der *Person* abgespeicherten Skelettdaten um verschiedene zeitabhängige Berechnungen durchführen zu können, z.B: absolute Geschwindigkeit, relative Geschwindigkeit, etc.
Condition	Eine *Condition* ist der eigentliche Gestenteil. Sie kann mit *check* auf Gültigkeit überprüft werden. Zudem beinhaltet sie die Events *OnSuccess* und *OnFail*.
Device	Das *Device* kapselt das physikalische Gerät. Es gibt *Personen* per Event zurück wenn sie aktiv werden und meldet neue *Skelette*.
GestureChecker	Der *GestureChecker* führt Buch über eine komplette Geste. Er speichert wie weit fortgeschritten die Erkennung einer Geste ist und feuert Events bei der erfolgreichen Beendigung oder beim Abbruch.
SkeletonMath	Klasse für Vektorberechnungen auf Skelettdaten.
SmothendSkeleton	Geglättete *Skelette* sind die Datenquelle für alle Berechnungen und damit für die Gestenerkennung. Sie werden in der *Person* gespeichert und vom *Checker* verarbeitet.

Tabelle 4: Kurzbeschreibung der wichtigsten Domänenkomponenten

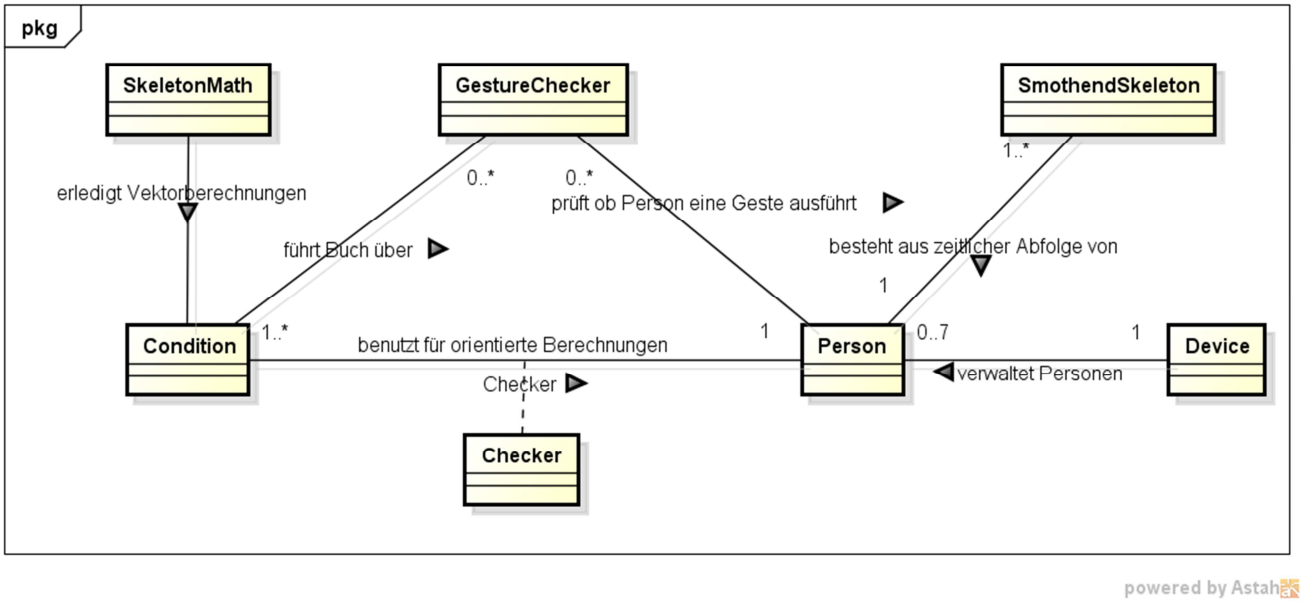


Abbildung 3: Vereinfache Domainanalyse

6.3 Beispielsequenz einer Geste

Der Ablauf einer Gestenerkennung lässt sich in drei Phasen aufteilen – grafische Darstellung siehe unten.

1. Initialisierung:

   In jedem Framezyklus werden alle *Personen* vom *Device* eindeutig identifiziert. Die identifizierten *Personen* bekommen vom *Device* ihr aktuelles *Skelett* zugewiesen. Jede *Person* erhält so mit der Zeit einen Skelettcache von 10 zeitlich geordneten *Skeletten*, welche für dynamische Berechnungen verwendet werden können.

2. Prüfen:

   Nach der Zuweisung vom neuen *Skelett* wird von den der *Person* zugeordneten *GestureCheckern* geprüft ob das aktuelle *Skelett* einen gültigen Kontext für die registrierten *Conditions* bildet. Die *Conditions* feuern Events über den Status des Kontextes (*Success*, *Failed*, *Triggered*).

3. Feedback:

   Der *GestureChecker* führt Buch über die Reihenfolge der Gestenteile (*Conditions*). Falls eine Geste komplett fertig durchlaufen wird, ist sie erfolgreich und wird mit einem vom User bei der API registrierten Event quittiert.

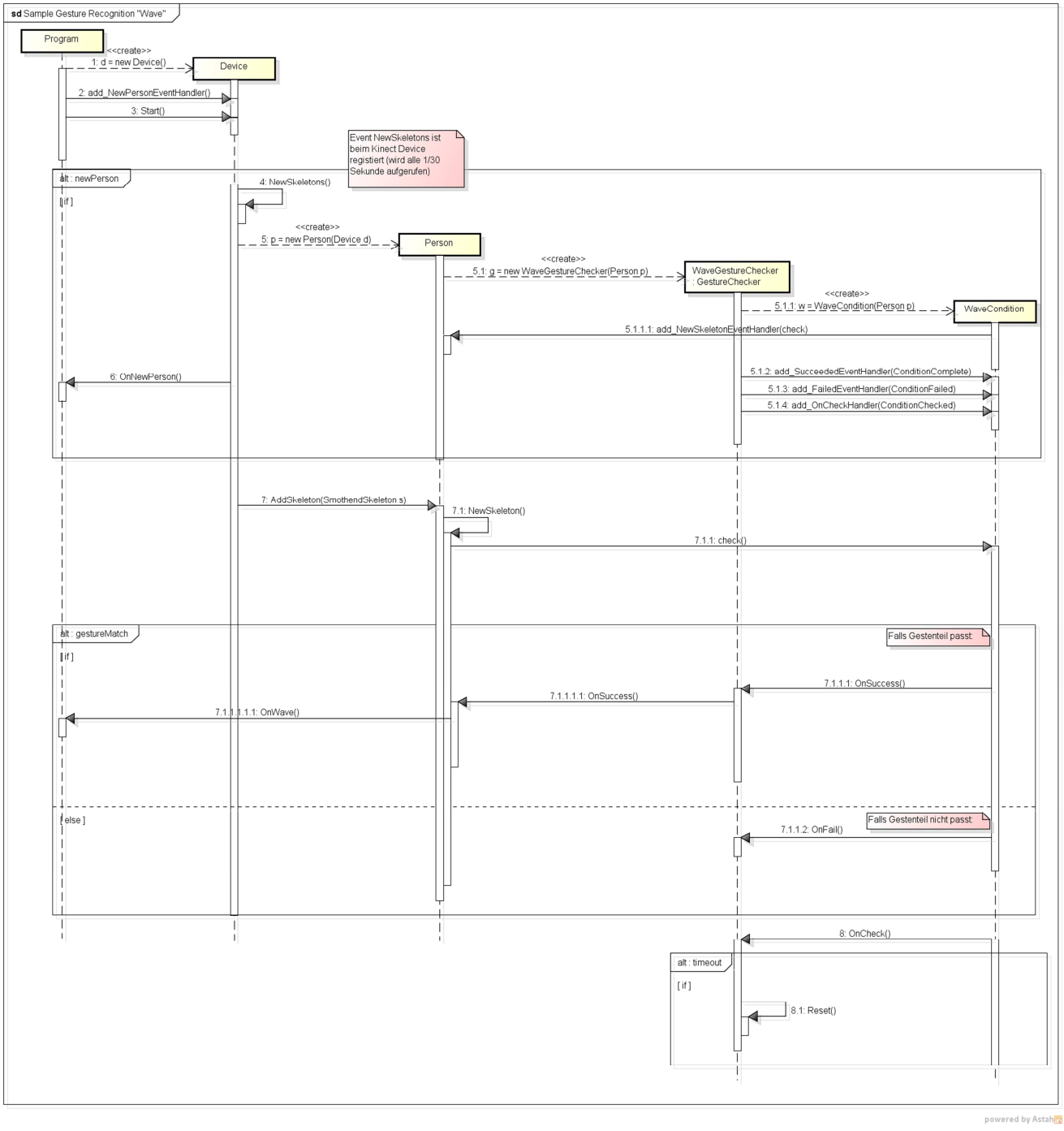


Abbildung 4: Beispielsequenz des Erkennens einer einfachen Geste.

6.4 Lösungsansätze für aufgetretene Probleme

6.4.1 Zuweisungsalgorithmus für neue und bestehende Personen

Die Unterscheidung von mehreren Personen hat Schwierigkeiten bereitet. Die Kinect wechselt die Nummerierung der erkannten Skelette ohne erkennbares System. Deshalb muss die Zuweisung der erkannten Skelette an neue oder bestehende Personen per Software erfolgen. Dies erfordert ein Matching zwischen den bisherigen Skeletten der existierenden Personen und den Skeletten die wir jeweils neu von der Kinect bekommen.

Die Ähnlichkeit eines Skelettes zu einem anderen wird lediglich anhand des Skelettgliedes „Hüfte“ bewertet. Diese Bewertung ergibt eine 2D-Matrix mit den Abweichungen als Einträge. Das Problem war nun die Auswertung dieser Ähnlichkeiten. Wie kann man am besten auswerten, welches Skelett welcher Person zuzuweisen ist? Eine Idee war, jeweils das Minimum in der Matrix zu suchen, die Zuweisung zu machen und sowohl Skelett als auch Person aus der Match-Matrix zu löschen. Das ist aber eventuell im Durchschnitt nicht die beste Zuweisung.

Falls die Kinect ein Skelett in einem Frame nicht mehr erkennt, weil z.B. die Sicht durch eine andere Person, oder ein Objekt verdeckt ist, löschen wir die Person nicht direkt, sondern speichern sie in einem Speicher für 5s. Falls sie wieder erkannt wird, wird sie reaktiviert, oder nach der Zeit gelöscht.

Wir entschieden uns vorerst für eine naive Lösung mit Listen, die jedoch gut zu funktionieren scheint:

Es werden drei Fälle unterschieden:

1. Es hat mehr Skelette als schon bestehende Personen, d.h. es kam eine Person ins Bild, sie muss aus dem Cache geladen werden. neu erstellt werden. Zudem müssen ihr die benötigten Events registriert werden.
2. Es hat mehr bestehende Personen als neue Skelette, d.h. es ging eine Person aus dem Bild. Sie muss gelöscht/vergessen werden (Die Person bleibt in einem dafür vorgesehenen Cache).
3. Es hat gleich viele Personen und Skelette, d.h. Zuweisung muss neu gemacht werden, sonst nichts.

Die Zuweisung erfolgt nun einfach mittels zwei temporärer Listen, aus welchen die gematchten Elemente gelöscht werden. Was übrig bleibt muss nach den drei Fällen (s.o.) beurteilt werden.

6.4.2 Zeitmessung in der .NET-Umgebung

Für die Zeitmessung in .NET gibt es verschiedene Möglichkeiten:

DateTime.Now.Milliseconds: Gibt die Millisekunden der aktuellen Sekunde aus, d.h. nur Werte zwischen 0 und 999, damit ist keine sinnvolle Zeitmessung möglich.

DateTime.Now.Ticks: Gemäss MSDN die Anzahl Millisekunden seit Systemstart * 10. – sie sind aber alles andere als genau somit auch unbrauchbar.

Lösung: Zeitdifferenz in Millisekunden von DateTime.Now und dem 1. Jan. 1970 in Millisekunden, für Timeouts. Durch das Scheduling des CPU taugt diese Methode aber nicht für die Geschwindigkeitsberechnung der Skelette. Für diese haben wir den von der Kinect mitgelieferten Timestamp in den Skeletten gespeichert.

6.4.3 Event-Triggers aus Subklassen

Events, aus Subklassen können nicht direkt aufgerufen werden, sondern müssen in der Subklasse von einer protected Funktion gekapselt werden – in folgendem Stil: *fireSuperclassEvent()*.

Eine andere Möglichkeit besteht darin, die Events der Superklasse virtual zu deklarieren, und sie in der Subklasse zu überschreiben. Dies kann jedoch für den Benutzer zu Verwirrung führen.

6.4.4 GestureChecker-Statemachine: Unterscheidung zwischen Triggered und Success

Falls eine Geste nicht nur im Erfolgsfall ein Feedback zurückgeben soll, sondern periodisch die ganze Zeit über, reicht das Modell „Failed/Success“ nicht. Für durchgehend auswertbare Gesten wurde der Event „Triggered“ in der DynamicCondition eingeführt. Triggered wird so lange aufgerufen bis der betreffende Gestenteil erfolgreich ist oder Misserfolg signalisiert. Im Prototyp sieht man dieses Verhalten beim PinchZoom. Sobald die Ausgangsposition mit beiden Händen eingenommen wurde, triggert die ZoomCondition. Sobald eine gewisse Geschwindigkeit überschritten wird oder die Hände

nicht mehr in Zoom-Position sind, signalisiert die ZoomCondition ein „failed“. Da diese Geste nur eine Condition besitzt, ruft jene nie Success auf. Die Statemachine muss nicht weiterschalten.

6.4.5 GestureChecker-Statemachine: Timeouts

In einem späten Codereview kam raus, dass das Eventbasierte Design und die damit verbundene Flexibilität der Implementation von eigenen Gesten Probleme in der Zeitmessung mit sich brachten. Timeouts wurden nur erkannt, wenn die laufenden Conditions die Ausführung der Check-Methode mit Success oder Failed quittiert haben. Da der Aufruf dieser EventHandler jedoch in der Freiheit des Implementierers liegt, kann es vorkommen, dass ein Defekt im Gestenerkennungscode verursacht, dass eine Geste ewig läuft und doch keinen Timeout signalisiert. Es wurde diskutiert, ob die Zeitbehandlung an die Condition übergeben werden soll. Aufgrund der Überlegung, dass die Zeit Sache des GestureCheckers ist, wurde ein weiterer Event „OnCheck“ für die Conditions eingeführt. Jener signalisiert, dass ein Gestenteil überprüft wurde und sagt nichts darüber aus, ob er erfolgreich war oder nicht. Der GestureChecker hört nun auf diesen Event und benutzt ihn zur Überprüfung ob die Geste noch in der vorgegebenen Zeit liegt.

Bei der jetzigen Architektur wäre die Unterscheidung zwischen Condition und DynamicCondition sowie Triggered und OnCheck eigentlich nicht mehr nötig, wenn anstatt dem OnCheck direkt das Triggered von jedem Gesteteil bei der Ausführung von Check aufgerufen werden würde (s.u. beim Sequenzdiagramm). Wir haben uns jedoch gegen die Zusammenlegung der zwei Events entschieden, da wir die Kontrolle über eine Rückmeldung über den Gestenstatus doch dem Implementierer einer Condition überlassen wollen. Deshalb geht OnCheck nie über den GestureChecker hinaus.

6.5 Memory Management: Speicherzuweisung für Eventhandling

Das Verwenden von Events hat den Nachteil, dass long-lived Publisher ihre Referenzen zu Event-Subscribern im Speicher behalten und so zu Speicherlecks führen können. Unsere Applikation wurde auf Lecks geprüft und für stabil empfunden.

6.5.1 Code-Analyse der Zuweisungen

In folgender Tabelle sind alle EventHandler-Referenzen aufgeführt. Es ist ersichtlich, dass praktisch alle EventHandler-Referenzen vom GarbaggeCollector beim Aufräumen einer *Person*-Instanz indirekt im Speicher invalidiert werden. Der Event im Device ist auch unproblematisch, da sowohl die *KinectSensor*- als auch die *Device*-Instanz long-lived sind. Zur Sicherheit wird er jedoch beim Entfernen einer *Person* aus dem Expiration-Cache deregistriert.

Da eine *Person*-Instanz jeweils nur während der Bedienungszeit existiert, sind die verwendeten Events unproblematisch. Der Speicher wird nicht zu sehr beansprucht.

Klasse	Publisher	Subscriber	Subscriber lebt länger oder gleich lang wie Publisher
Condition	Person.NewSkeleton	Condition.check	✔
Device	KinectSensor.SkeletonFrameReady	Device.NewSkeletons	✔
Device	Person.OnWave	Device.personWaved	?
GestureChecker	Condition.Succeeded	GestureChecker.ConditionComplete	✔
GestureChecker	Condition.Failed	GestureChecker.ConditionFailed	✔
~~GestureChecker~~	~~Timer.Elapsed~~	~~GestureChecker.Timeout~~	✘
Person	GestureChecker.*	Person.*	✔

Tabelle 5: Auswertung der Klassen mit Events

Bemerkung: Der Publisher hält jeweils eine Referenz auf den Subscriber, nicht aber umgekehrt.

Die einzige Ausnahme bildete die Verwendung der *Timer*-Klasse im *GestureChecker*. Der *Timer* registrierte sich bei System Events um den erfolglosen Ablauf einer Geste durch einen Timeout abzubrechen. Durch das Verwenden von vielen *GestureCheckers* entstand ein Speicherleck (siehe unten). Der Timer wurde durch eine einfachere Zeitmessung ersetzt und wird gar nicht mehr verwendet.

6.5.2 Online-Analyse mit Mocking

Zum Testen der Stabilität des Speichers wurde ein *Device (MockingDevice)* vorgetäuscht und das Erscheinen und Verschwinden von vielen Personen vorgetäuscht. Zudem wurden kritische Klassen wie *Condition*, *GestureChecker* und EventArgs mit grossen Byte-Arrays aufgepumpt. So wurde das einzige Speichleck schnell sichtbar.

Was zum Problem wurde war die Verwendung des C#-Timers. Jener wurde benutzt um auf einen Timeout in der GestureChecker-Statemachine zu reagieren. Dabei behielt ein System-Event des jeweils praktisch alle Objekte der Applikation. Durch die automatisierte Analyse mit SciTech .NET Memory Profiler konnten die aufgepumpten Objekte zurückverfolgt werden. Im dem Allocation-Tree ist gut sichtbar, dass praktisch alle Instanzen der Applikation vom Timer gehalten werden.

Da aus der Analyse der Aufrufshierarchie zusätzlich ersichtlich war dass der *Timer* sehr viel Rechenzeit verbrauchte, wurde komplett auf diese Klasse verzichtet. Stattdessen wird jetzt eine weniger komplexe Zeitmessung aufgrund der Systemzeit verwendet um den Timeout einer Geste zu messen.

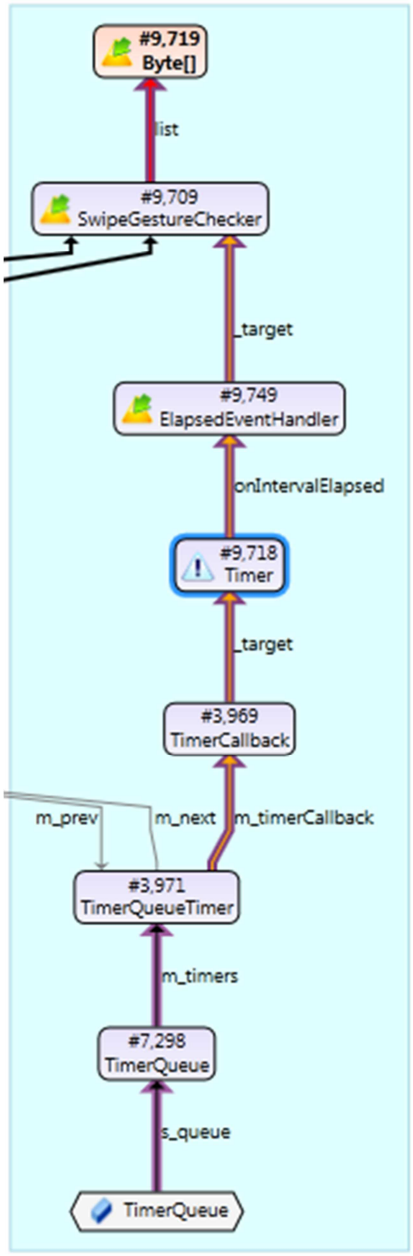


Abbildung 5: Speicherkritischer Trace der Timer-Klasse

7. Framework – Äussere Architektur (API)

7.1 Allgemeine Überlegungen

7.1.1 Multi-Layer

Die User-API wird in verschiedenen Layern aufgebaut. Der Benutzer des Gestenerkennungsframework kann entscheiden, welchen Layer und damit auch welche Komplexität er benutzen will. Das hohe Layer bietet einen eingeschränkten Funktionsumfang, den man sehr einfach einbinden kann. Das tiefe Layer bietet Möglichkeit eigene Gesten zu definieren oder auf Low-Level Eigenschaften zuzugreifen. Beide Layer lassen sich kombiniert einsetzen. Beispielsweise muss sich der Benutzer der API nicht um die Aktivierung der Personen kümmern, kann aber dennoch eigene Gesten definieren – oder umgekehrt.

7.2 Schnittstellendefinition – Hoher Layer

Klasse	Beschreibung
Device	Von einer *Device*-Instanz bekommt man alle neuen *Personen* indem man sich beim *NewActivePerson*-Event registriert. Nach dem Aufruf von *Start()* initialisiert das *Device* die Kinect und beginnt mit der Erkennung von *Personen*. Sobald sich eine Person durch Winken anmeldet, wird dieser Event gefeuert. Im *ActivePersonEventArgs*-Parameter wird diese *Person* mitgegeben.
Person	Bei einer *Person*-Instanz können Gestenreaktionen registriert werden. Im mitgelieferten Prototypen sind dies die folgenden: - *OnZoom*: übermittelt den Zoomfaktor - *OnSwipe*: signalisiert eine Wisch-Geste - *OnWave*: signalisiert Winken

Tabelle 6: Schnittstellendefinition Einfach

Beispiel zur Benutzung der API:

```
Device d = new Device();          // Erstellen des des Devices
d.PersonActive += NewPerson;      // Registrieren auf neue Personen
d.Start();                        // Starten der Kinect


void NewPerson(object src, ActivePersonEventArgs activePersonEventArgs )
{
  activePersonEventArgs.Person.OnSwipe +=                // Auf Swipe der aktiven Person hören
   delegate(object sender, GestureEventArgs e)
   {
     Console.Write(activePersonEventArgs.Person.Id       // Anzeigen der ID der Person
       + " swiped " +
       ((SwipeGestureEventArgs)e)                        // Casten der EventArgs um weitere
                                                                Informationen zu bekommen
       .Direction.ToString());                           // Ausgeben der Richtung des Swipes
    };
}
```

7.3 Schnittstellendefinition – Tiefer Layer

Klasse	Beschreibung
Device	Von einer *Device*-Instanz bekommt man alle neuen *Personen* indem man sich beim *NewPerson*-Event registriert. Nach dem Aufruf von *Start()* initialisiert das *Device* die Kinect und beginnt mit der Erkennung von *Personen*. Die bei *NewPerson* registrierten Funktionen werden jetzt mit dem *NewPersonEventArgs*-Parameter aufgerufen. Jener enthält jeweils eine neue Person. Der *PersonActive*-Event wird gefeuert, wenn sich eine *Person* einloggt. Im *ActivePersonEventArgs*-Parameter wird diese *Person* mitgegeben.
Person	Bei einer *Person*-Instanz können zusätzlich zu den Standartgesten auch noch die Roh-Skelettdaten ausgelesen werden.
GestureChecker	Wenn man einen eigenen GestureChecker erstellt und mit eigenen Conditions befüllt, kann man sich auf seine eigenen Gesten registrieren.
Condition	Hiermit kann man eigene Bedingungen definieren, die chronologisch überprüft werden im Gestu-re Checker. Die Condition gehört jeweils nur zu einer Person und sie wird vom GestureChecker automatisch aktiviert und deaktiviert.
Checker	Damit kann man komplexe Berechnungen mit bestimmten Punkten machen, und man kriegt ein-fach Richtungen und Geschwindigkeiten zurück.
SkeletonMath	Diese Klasse bietet Vektoroperationen an, die speziell auf Skelettpunkte ausgelegt sind.
GestureEventArgs	Hiermit kann die Geste noch Argumente übergeben.

Tabelle 7: Schnittstellendefinition Erweitert

Im Folgenden sei die Beispielimplementation einer Springen-Geste gegeben. Sie ist sehr einfach gehalten und soll lediglich zeigen mit welchen Mitteln eine neue Geste implementiert werden kann.

Beispiel einer Springen-Geste:

```
class Tester
{
  private static JumpGestureChecker jgc;

  static void Main(string[] args)
   {
     Device d = new Device();                  // Erstellen des des Devices
     d.NewPerson += NewPerson;                 // Registrieren auf neue Personen
     d.Start();                                // Starten der Kinect
   }

  static void NewPerson(object src, NewPersonEventArgs newPersonEventArgs )
    {
      jgc = new JumpGestureChecker(
        newPersonEventArgs.Person);            // Anlegen des Eigenen GestureCheckers
      jgc.Successful += delegate
        { Console.WriteLine("Jump"); };        // Registrieren auf dessen Event}
    }

class JumpGestureChecker : GestureChecker      // Klasse implementiert GestureChecker
{
  public JumpGestureChecker(Person p)          // übergeben der zu überwachenden Person
   : base(new List<Condition>
     {
        new JumpCondition(p)                   // Anlegen einens GestureCheckers mit einer
                                                      JumpCondition
     }, 1000){}                                // timeout ist hier nicht von Belang
    }

class JumpCondition : Condition                // JumpCondition prüft, ob gesprungen wurde
{
  private Checker c;                           // Checker für die Berechnungen
```

```
  public JumpCondition(Person p) : base(p)
  {
    c = new Checker(p);
   }

  protected override void Check(
     object src, NewSkeletonEventArgs e)                // überprüfung bei jedem neuen Skelett
    {
      if (c.GetAbsoluteMovement(JointType.HipCenter)    // Bewegung der Hüfte
        .Contains(Direction.Upward))                    // nach oben?
     {
       FireSucceeded(this, new JumpGestureEventArgs()); // Condition erfolgreich
     }
     else
     {
       FireFailed(this, new FailedGestureEventArgs
              {Condition = this});                      // nicht erfolgreich
     }
   }
}

class JumpGestureEventArgs : GestureEventArgs{}         // Args für optionale Parameter
```

8. Anhang

8.1 Gestenevaluation – mögliche Gesten

Für die Bedienung eines herkömmlichen Touch- oder Gesten-basierten Computer-Systems werden im Allgemeinen diese Gesten unterstützt:

- Anmeldung
- Zoom
- Scrollen/Blättern
- Cursor
- Spezialaktionen
- Abmeldung

Im Folgenden wurde frei formuliert, was für Gesten zur Umsetzung dieser Funktionalität möglich sind. Anhand dieser Beschreibungen wurde entschieden welche Gesten für ein Industriepanel sinnvoll sind.

8.1.1 Anmeldung

Diese Geste ist wichtig oder gar kritisch, um dem System mitzuteilen, dass man es jetzt bedienen möchte oder darf. Es ist schwierig dafür eine gute Geste zu finden, die dem User klar ist. Es ist für einen User ungewohnt, dass er einem System signalisieren muss, dass er interagieren möchte. Alle anderen Inputsysteme reagieren sofort auf einen Input.

Arme nach aussen Halten

Diese Geste ist ziemlich eindeutig, was Fehlaktivierungen minimieren würde. Der Nachteil ist, dass diese Geste relativ viel Platz braucht, was andere Personen behindern könnte. Zudem braucht es eine Information, da diese Geste nicht intuitiv ist.

Slide to Unlock

Diese Geste wird vielen Benutzern bekannt sein – es braucht keine Anleitung dafür. Dafür ist die Erkennung schwieriger, da die Position der Hand nicht klar definiert ist. Es kann zu einer irrtümlichen Aktivierung kommen, wenn nur schon eine Person vorbeiläuft. Dafür braucht diese Geste wenig Platz. Es gilt zu beachten, dass Apple hat darauf ein Patent[2] hat. Obwohl es darauf basiert, dass eine Geste auf einem Bild ausgeführt wird und nicht in der Luft, müsste man rechtlich abklären, ob man die Geste so verwenden dürfte.

Winken

Diese Geste ist bei den meisten Systemen umgesetzt und ziemlich selbsterklärend. Dem Benutzer müsste lediglich signalisiert werden, dass er Winken soll. Die Erkennung dieser Geste ist etwas einfacher als *Slide to Unlock*, aber auch hier können selten Fehlaktivierungen vorkommen. In den ersten Wochen des Einsatzes könnte es bei den Mitarbeitern zu Missverständnissen kommen, wenn sie eine Person winken sehen, da dies eine übliche Geste zwischen Personen ist um Aufmerksamkeit zu erlangen. Das exponiert den User und lenkt andere ab in einer grossen Halle.

Militärischer Gruss

Die militärischen Grüsse sind allgemein bekannt und intuitiv. Es stellt sich die Frage, ob die Benutzer bereit sind, vor einem elektronischen System zu salutieren. Das System erscheint u.U. unsympathisch und wird nicht akzeptiert.

Verbeugen

Intuitive Geste, die jedoch u.U. unbequem sein kann (Arbeitskleidung). Zusätzlich stellt sich die Frage, ob eine Verbeugung nicht als Unterwerfung verstanden wird und somit keine Akzeptanz bei den Benutzern findet.

[2] US Patent *US 8286103 B3* (9.10.2012): *http://assets.sbnation.com/assets/1522863/US8286103B2.pdf*

8.1.2 Zoom

Zoom ist eine optionale Funktionalität eines Computersystems. Sie ist nicht nötig, vereinfacht jedoch viel. Es wird einfachen, den Überblick über eine Applikation zu erhalten. Zudem können Details dem Benutzer einfacher verfügbar gemacht werden. Für dem Zoom kommen nicht viele Gesten in Frage, da meist zwei Hände einbezogen werden müssen.

Pinch

Pinch-Zoom ist auf Touchscreens sehr stark verbreitet und dürfte daher den meisten Benutzern klar sein. Die Adaption in den 3D-Raum könnte sich jedoch als schwierig erweisen. Durch den Umstand, dass die Hände in sehr viele Richtungen zueinander und auseinander bewegt werden können, ist die Erkennung hierbei schwieriger und könnte mit anderen Gesten interferieren. Unklar ist ebenfalls, wann die Zoomfunktion aktiviert werden soll und wann der User seine Hände bewegt, damit er nachher eine Zoomfunktion auslösen kann (Fehlaktivierungen). Zudem werden für diese Geste beide Hände benötigt, im Industrie-Umfeld wäre eine einhändige Bedienbarkeit sicher ein Vorteil (s.u. Joystick).

Push/Pull

Je nach Darstellung ist auch diese Geste sehr intuitiv und für jeden User logisch. Diese Geste ist sehr einfach zu erkennen und eindeutig. Dadurch wird es zu wenigen Fehlaktivierungen kommen. Jedoch ist auch hier unklar, wann der User zu einer solchen Bewegung ansetzen will und wann er sie wirklich ausführen will. Diese Geste kann jedoch nur mit einer Hand bedient werden.

Wenn man diese Geste verwendet kann man logischerweise Stossen und ziehen nicht mehr für eine Auswahl verwenden.

Kreis Zeichnen

Hierbei kann der User mit einer Hand einen Kreis in die Luft zeichnen und je nach Richtung wird hinein der herausgezoomt. Das ist zwar nicht besonders intuitiv, dafür aber schnell erklärt und es kann ziemlich stabil erkannt werden. Es wird wenig Interferenz mit anderen Gesten haben. Eine sicherte Erkennung ist aber erst ab einem bestimmten Kreisdurchmesser möglich. Diese Geste ist vor allem für Benutzer des Mobiltelefons *Nokia N900* interessant, da sie dort verwendet wird – auch der *Apple iPod* setzt auf ein Drehrad.

8.1.3 Scrollen/Blättern

Bei den meisten evaluierten Systemen wird Scrollen dem Blättern gleichgesetzt. Blättern wird dabei als Einrastfunktion für Scrollen umgesetzt.

Wischen

Diese Geste wird praktisch bei fast allen Produkten/Projekten verwendet. Da man die Geschwindigkeit der Bewegung analysieren kann, werden einerseits Fehlaktivierungen minimiert, andererseits kann man die Distanz und die Geschwindigkeit auf die Scroll-Geschwindigkeit übertragen.

Joystick

Diese Bewegung ist nicht ganz so intuitiv wie andere, dafür hat sie den Vorteil, dass man nicht „nachgreifen“ muss. Zudem sind bei dieser Bewegung der Platzbedarf und der Bewegungsradius grösser. Je nach Gestaltung des GUIs ist auch diese Geste intuitiv und sollte allen Nutzern klar sein – z.B. wenn nur noch der Joystick zur Bedienung benutzt wird (s.u.).

Oberkörper bewegen

Gezoomt würde hier per vor- und rückwärtslehnen. Diese Geste ist nicht intuitiv und braucht sicher eine Anleitung. Zudem ist sie für unser Umfeld mit spontanen Nutzern nicht wirklich geeignet. Unter Umständen ist sie auch unbequem. Aus technischer Sicht ist diese Geste wahrscheinlich schwierig auszuwerten falls der Benutzer weit weg steht.

8.1.4 Cursor bewegen

Joystick

Hierbei werden die Cursorfunktion und das Scrolling kombiniert. Das ist stabil erkennbar und hat keine Interferenz mit anderen Gesten. Es wird den Usern intuitiv klar sein.

Zeigen

Dabei wird mit dem Unterarm auf einen Punkt gezeigt. Das ist sicher ungenauer in der Erkennung und benötigt wahrscheinlich eine Kalibrierung für jede neue Session. Zudem wird die Nutzung der Geste „Push“ verunmöglicht. Für den User ist diese Geste sehr intuitiv.

2D Mapping der Handposition

Die Handposition wird vor dem Körper mit angewinkeltem Arm ausgewertet. Diese Geste ist für User gewöhnungsbedürftig und nicht sehr stabil erkennbar. Dafür interferiert sie nicht mit anderen Gesten und es wäre möglich, pro Hand einen Cursor darzustellen. Klicken könnte dann durch Zeigen umgesetzt werden.

8.1.5 Auswählen

Diese Geste ist sehr wichtig, jedoch nicht ganz einfach umzusetzen, da es bei anderen Technologien dafür immer einen Button oder eine intuitive Lösung gibt. Das ist bei unserer Lösung nicht der Fall. Durch die grosse Distanz wird es zudem unmöglich Handgesten zu erkennen. Deshalb ist es wichtig beim finalen Programm darauf zu achten, dass möglichst wenige Selektionen gemacht werden müssen.

Stossen

Diese Geste ist bei anderen Projekten am häufigsten umgesetzt und dürfte dem User intuitiv einigermassen klar sein. Dafür ist es bei dieser Bewegung sehr wahrscheinlich, dass währenddessen der Cursor bewegt wird, was zu einer Fehlaktion führen wird.

Spezielle Gesten mit der anderen Hand

Hier sind verschiedene Gesten denkbar z.B. Stossen, Winken oder ausstrecken. Dadurch werden Fehleingaben minimiert, jedoch ist das einiges weniger intuitiv und benötigt zudem beide Hände.

Grab

Dabei wird ein Objekt gepackt und zu sich gezogen. Das ist etwas einfacher zu erkennen, Cursorverschiebungen werden weniger häufig vorkommen. Je nach Gestaltung des GUIs ist diese Geste intuitiv klar.

Nicken

Diese Geste ist intuitiv relativ klar und sollte gut zu erkennen sein. Fehlaktivierungen sind denkbar, wenn sich der User gerade mit jemand anderem unterhält. Das kann jedoch anhand der Blickrichtung korrigiert werden.

Timer

Wenn der User seinen Cursor nicht bewegt, beginnt ein sichtbarer Timer abzulaufen. Wenn dieser abgelaufen ist, gilt die Selektion, wenn er bewegt wird. Problematisch ist hierbei eine saubere Kalibrierung. Ebenso muss erkannt werden, wann der User etwas zeigen möchte und wann er wirklich etwas selektionieren möchte.

Thumb Up

Eine sehr intuitive Geste, jedoch nicht machbar mit Kinect auf diese Distanz. Zudem wäre sie nicht für spezielle Arbeitskleidung (Handschuhe) einsetzbar.

8.1.6 Spezialaktionen (spezielle Aktionen)

Solche sind für unser Projekt nicht nötig, ausser man würde sie für die Anmeldung einsetzten. Dadurch haben wir uns nicht auf solche konzentriert bei der Recherche.

Bestimmter Winkel zwischen Körper und Armen

Diese Geste ist nicht besonders intuitiv und braucht eine Anleitung. Dafür ist sie gut zu erkennen.

8.1.7 Abmeldung

Die Abmeldung geschieht automatisch nach einem Timeout wenn der Benutzer das Sichtfeld der Kinect verlässt. Eine manuelle Abmeldung könnte durch das nochmalige Ausführen der Anmelde-Geste umgesetzt werden.

8.2 Diskussion Maus-Cursor

Am Anfang des Projektes stand zur Diskussion, ob man eine Cursor-Geste implementieren solle. Das wäre sicherlich interessant gewesen. Die Implementierung der Zeigengeste mittels Arm wäre zwar technisch u.U. machbar, aber zu komplex für den Umfang unserer Arbeit. Da wir zusätzlich noch vermuten, dass der Benutzer wahrscheinlich mit den Fingern zeigen würde, wäre die Komplexität, diese sowieso schon schwierige Aufgabe zu meistern, vermutlich ins unendliche gestiegen. Das Erkennen von Fingern selbst scheitert an den technischen Möglichkeiten des Kinect SDKs. Mit *OpenNI* wäre es theoretisch auf kurze Entfernungen möglich.

8.3 Microsoft Kinect Skelett-Koordinatensystem

Die Kinect erstellt aus den Tiefendaten des Infrarot-Sensors 3D-Skelette von erkannten Personen (Aktuell 7, Stand Dez. 2012). Dabei besteht ein Skelett aus einer Liste von Gliedern mit Koordinaten in Metern. Das erkannte Skelett wird automatisch auf eine horizontale Ebene justiert und verwendet das folgende Koordinatensystem.

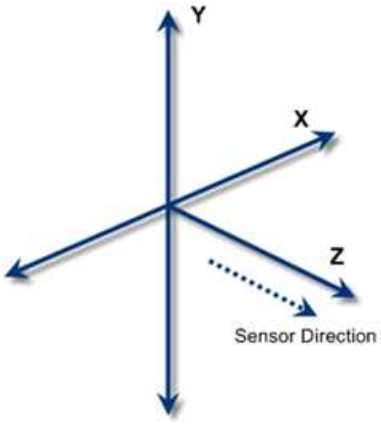


Abbildung 6: Skelett-Koordinatensystem Kinect

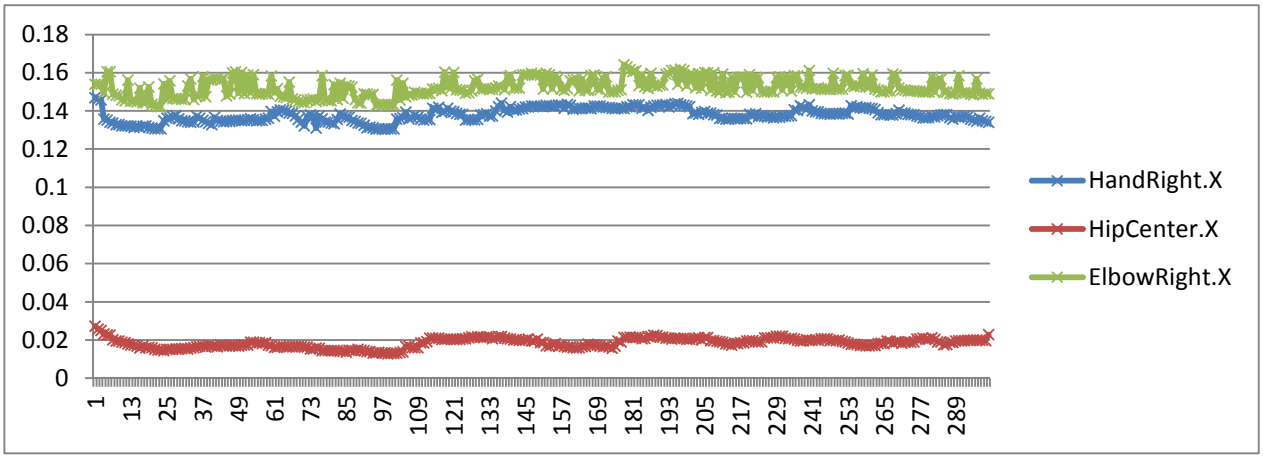
Mit Beamer-Tests haben wir herausgefunden, dass die 3D-Informationen integraler Bestandteil der Skeletterkennung sind. Wahrscheinlich werden intern zwar 2D-Information verwendet, diese alleine reichen jedoch nicht aus für die Erstellung eines Skeletts.

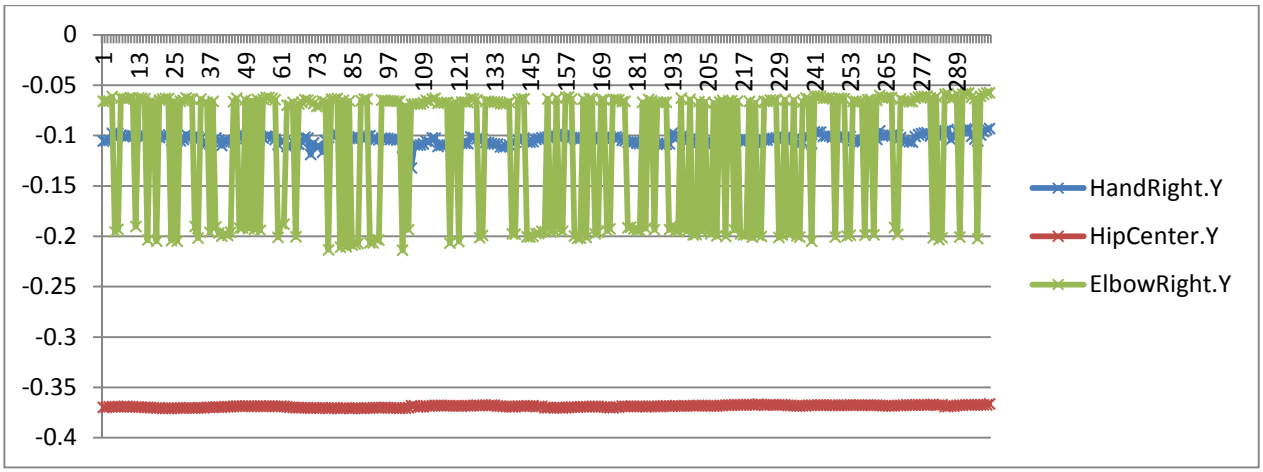
8.4 Testdaten Koordinatenstabilität

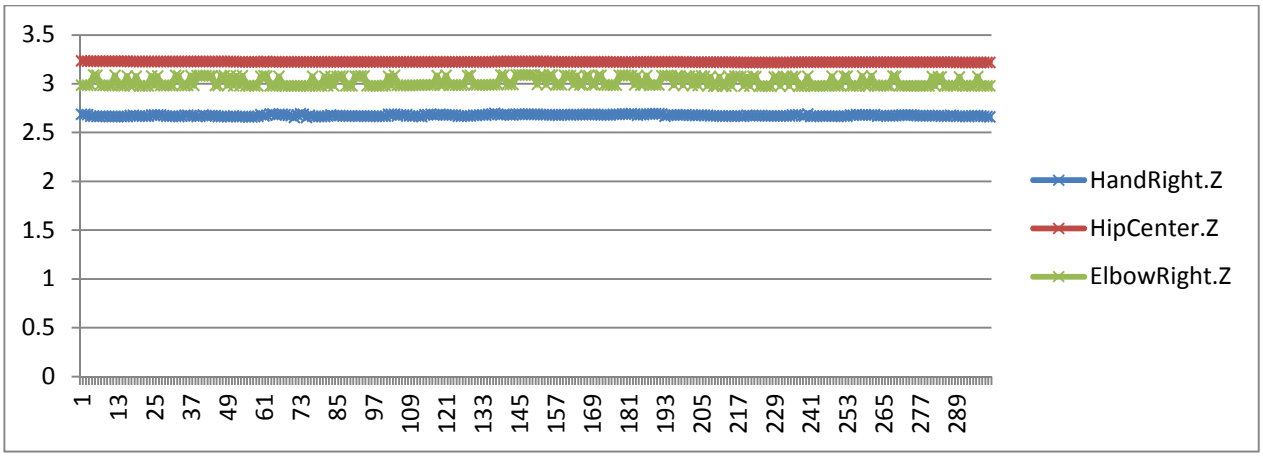
Im Rahmen der Technischen Evaluation der Kinect wurden am Anfang des Projektes die Möglichkeiten der Kinect-Skelett-Erkennung analysiert. Konkret wurden wichtige Positionen auf ihre Stabilität untersucht. Die Testperson war jeweils immer ca. 4m von der Kinect entfernt – knapp an der technischen Erkennungslimite). Es wurden jeweils 300 Samples aufgenommen.

8.4.1 Rechte Hand vor dem Körper

Die rechte Hand befindet sich angewinkelt vor dem Körper. Es ist ersichtlich, dass die Genauigkeit der erkannten Punkte bei vor der Kamera verborgenen Gliedern abnimmt (Ellbogen rechts).

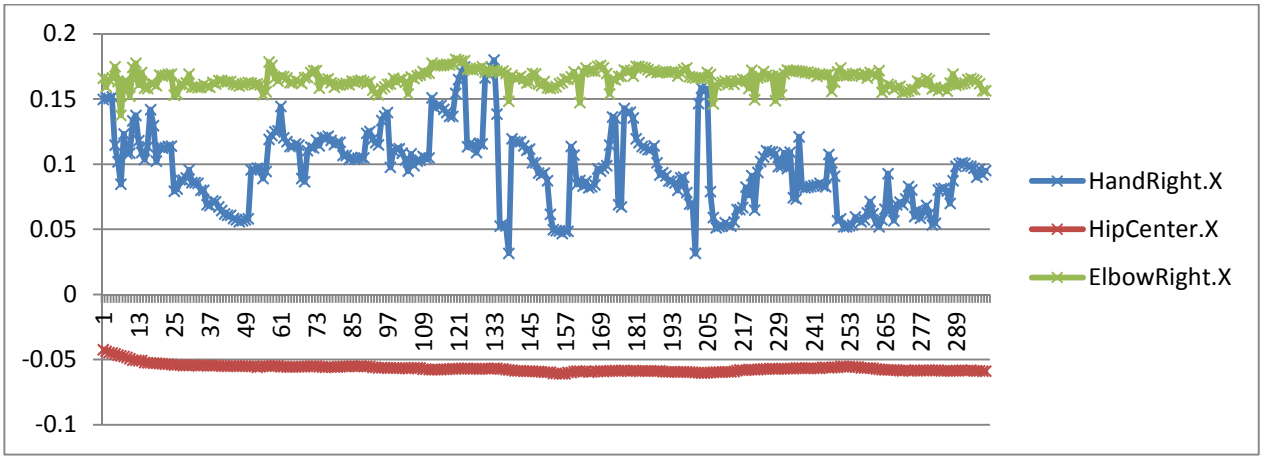


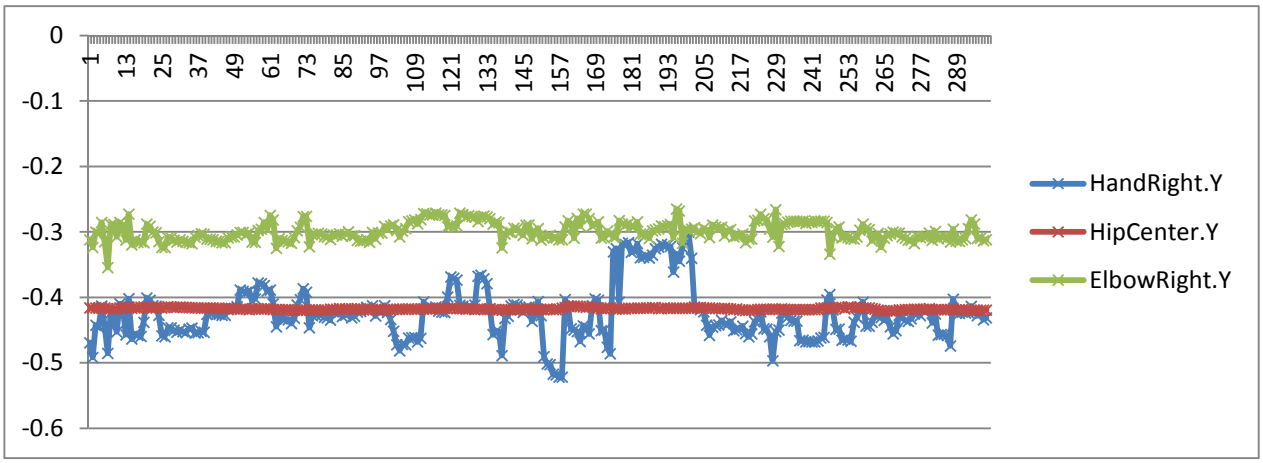


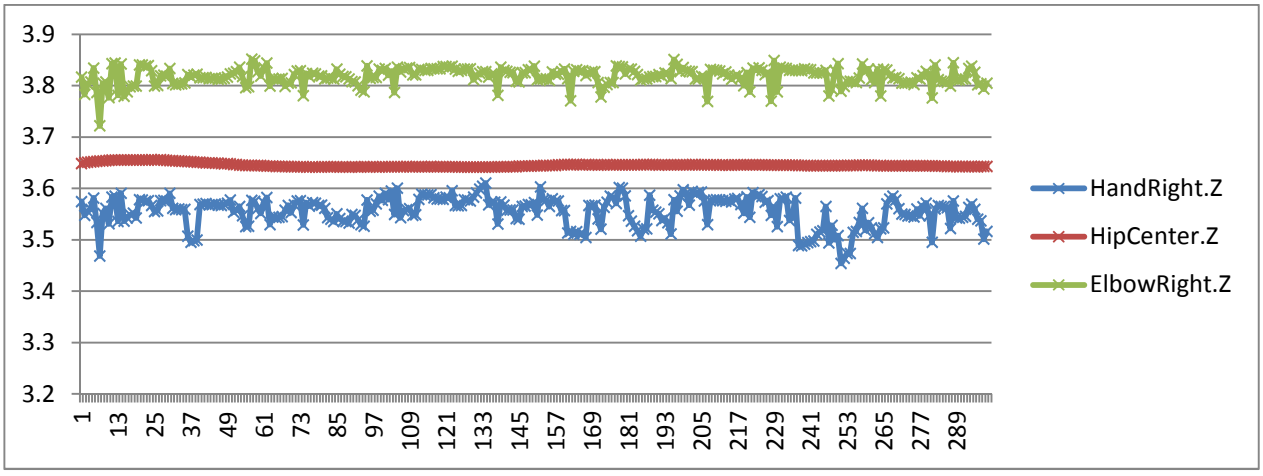


8.4.2 Rechte Hand hinter dem Körper

Die zum Sensor gewendete Person hielt den rechten Arm und die rechte Hand unten so weit nach hinten gesteckt wie ohne Anstrengung möglich. Da der Arm erkannt wird, bleibt die Genauigkeit des Ellbogens diesmal gut. Die Genauigkeit der Hand leidet jedoch erheblich, da der Arm seine genaue Position überdeckt.

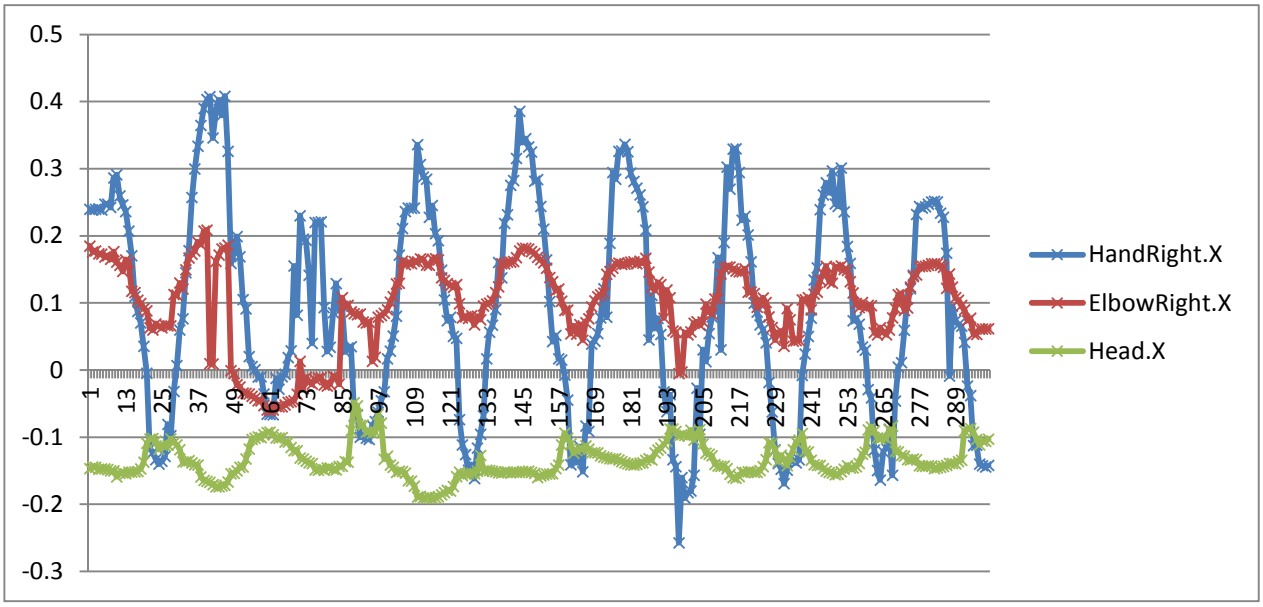


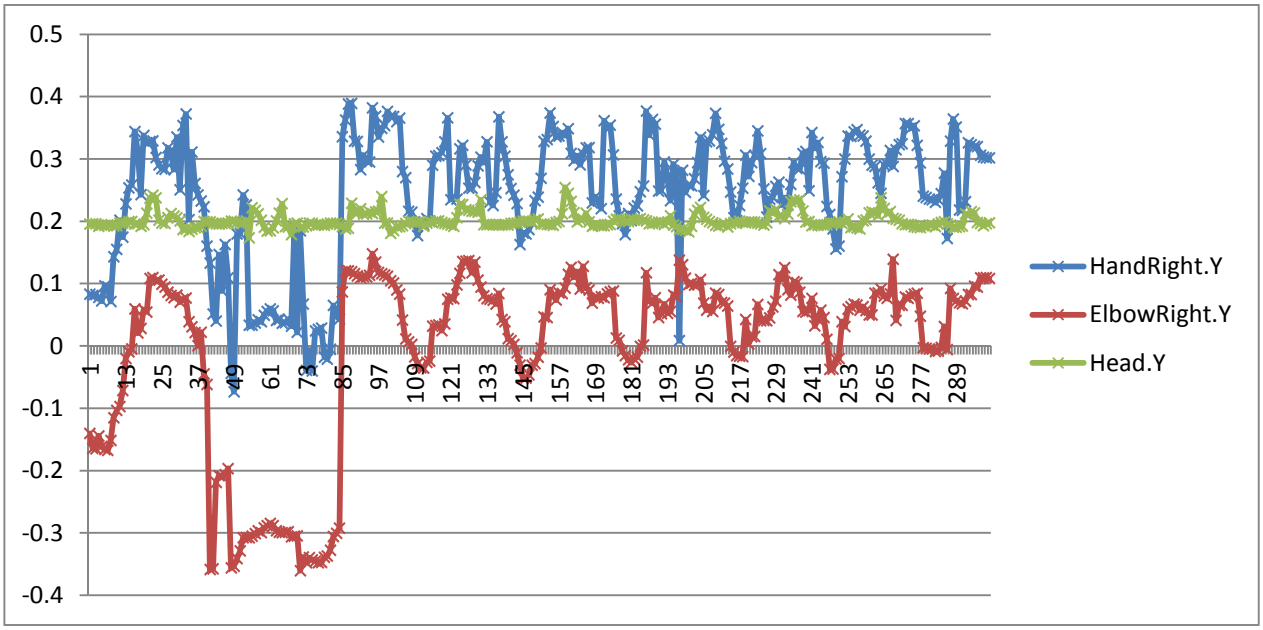




8.4.3 Winken mit der rechten Hand

Die Testperson winkte mit der rechten Hand. Man sieht unerwartete Auswirkungen auf die Position des Kopfes. Während die hin- und her Bewegung der Hand das erwartete Muster aufzeigt, schwingt die Y-Achse (auf- und ab) viel mehr: Die Seitwärts-Bewegung verringert die Auflösung der Y-Position der Hand erheblich. Das gleiche Phänomen kann man beim Ellbogen beobachten – jedoch nicht so ausgeprägt. Alles in allem sieht man jedoch, dass selbst bei den Ausreissern am Anfang die Bewegungsrichtungen der Glieder klar ersichtlich sind. Man kann daraus schliessen, dass das Erkennen von Bewegungen einfacher ist, als das Erkennen von bewegten Positionen (Cursorproblematik).





8.5 Abbildungsverzeichnis

8.6 Tabellenverzeichnis

8.7 Quellenangaben

- MCS UK Solution Development Team
  http://blogs.msdn.com/b/mcsuksoldev/archive/2011/08/08/writing-a-gesture-service-with-the-kinect-for-windows-sdk.aspx
- Coordinate Spaces
  http://msdn.microsoft.com/en-us/library/hh973078.aspx
-