

JONGLIEREN MIT DER KINECT

EIN SOFTWAREPROJEKT IM PROJEKT BILDVERARBEITUNG

PROJEKTBERICHT
ROLF BOOMGAARDEN
FLORIAN LETSCH
THIEMO GRIES

28. JUNI 2014

UNTER AUFSICHT VON: BENJAMIN SEPPKE
ARBEITSBEREICH KOGNITIVE SYSTEME
FACHBEREICH INFORMATIK, UNIVERSITÄT HAMBURG

Inhaltsverzeichnis

1 Einleitung	3
2 Vorüberlegung	4
3 Zielsetzung	5
4 Möglichkeiten der Kinect	5
4.1 Beispielprojekte mit der Kinect	6
5 Recherche: Ein jonglierender Roboter	6
6 Lösungsidee	7
7 Anfängliche Umsetzung	9
7.1 Programmstruktur	9
7.2 Programmfluss	11
7.2.1 Schritt 1: Tiefendaten vorverarbeiten	11
7.2.2 Schritt 2: Regions Of Interest isolieren	12
7.2.3 Schritt 3: Bälle in Frame-Folgen einander zuordnen	13
7.2.4 Schritt 4: Bereinigte Wurfparabel	18
7.3 Erläuterung verwendeter Bildverarbeitungsverfahren	18
7.3.1 Temporaler Buffer	18
7.3.2 Kalman Filter	18
7.4 Herausforderungen	20
7.4.1 Hough-Transformation	20
7.4.2 Kalman-Filter	23
7.4.3 Local Maxima	23
7.4.4 Temporal Filter	24
7.4.5 Technische Herausforderungen	24
8 Finale Umsetzung	25
8.1 Bewertung der Umsetzung	25
9 Anwendungsmöglichkeiten	25
9.1 Idee: Ein interaktiver Jongliertrainer	25
9.1.1 Objekte zählen	26
9.1.2 Würfe zählen	28
9.1.3 Wurfhöhen messen	28
9.2 Idee: Siteswaps erkennen	28
10 Stand der Forschung	29
11 Fazit	29
Quellen	31

1 Einleitung

Mit Markteinführung der Microsoft Kinect im November 2010 (FIXME?) steht in vielen Haushalten ein ausgereiftes bildverarbeitendes System im Wohnzimmer. Seit der Verfügbarkeit des eigentlichen Unterhaltungsgerätes steht auch für an wissenschaftlich Interessierte ein finanziell erschwingliches Gerät zu Verfügung, das für vielfältige Aufgaben im Bereich der Bildverarbeitung verwendet werden kann.

Mit den technischen Möglichkeiten dieses Tiefen- und Bilddaten liefernden Systems soll in dieser Arbeit versucht werden, das Wurfmuster eines mit Bällen jonglierenden Akteurs zu erfassen und zu analysieren.

Im Rahmen des Masterprojekts Bildverarbeitung haben wir uns zwei Semester lang mit der Kinect beschäftigt und in einem von Erfolgen und Fehlschlägen gefüllten Prozess versucht, die fliegenden Bälle eines jonglierenden Akteurs in Echtzeit zu verfolgen.

Ein funktionierendes Endergebnis wurde erzielt, das wir in dieser Arbeit vorstellen. Zugleich dokumentieren wir unsere Herangehensweise und umreißen relevante Arbeiten auf dem Gebiet, die uns Anregung gegeben haben. Ausgehend von einer grundsätzlichen Lösungsidee betrachten wir unseren ersten Ansatz, beschreiben die dabei aufgetretenen Probleme und führen so nach und nach auf das Endergebnis. Hierbei werden wir auch über die Implementierung sprechen, die von uns gewählte Softwarearchitektur erläutern und dokumentieren.

Das Endergebnis wird dann bewertet und aus den zu Anfang genannten Zielen hinsichtlich des Erfolges evaluiert.

2 Vorüberlegung

Bevor eine konkrete Zielsetzung getroffen wird, treffen wir ein paar Vorüberlegungen, die uns zu Projektbeginn begegneten. In der Ideenfindungsphase des Projektes kam uns sehr schnell die Idee, *irgendwie* die Kinect mit Jonglieren zu verbinden.

Dieses "*irgendwie*" war schwer zu greifen und zu definieren, da keiner von uns Erfahrung im Bereich der Tiefendatenverarbeitung hatte und mögliche Schwierigkeiten schwer abzuschätzen waren.

Die Idee zu einer interaktiven Anwendung zum Trainieren des Jonglierens kam uns relativ schnell, doch sobald wir anfingen, eine tatsächliche Anwendung konkret zu umreißen, wurde uns die Komplexität eines solchen Unterfangens bewusst. Um aber an der uns zugesagenden Grundidee weiter zu arbeiten, wollten wir uns im Rahmen des Projektes mit den bildverarbeitenden Grundkomponenten beschäftigen, die aus den Bild- und Tiefendaten der Kinect heraus die Jonglierbälle erkennt, verfolgt, und in irgendeiner Form sinnvolle Informationen über die bewegten Objekte aus diesen Daten generiert.

Der Einfachheit halber setzen wir uns zu Beginn mit dem Grundmuster des Jonglierens, der Kaskade auseinander. Diese entspricht anschaulich einer liegenden Acht. Auch haben wir uns auf die Anzahl von exakt drei Jonglierbällen festgelegt. Aus dieser Grundannahme haben wir uns eine Vereinfachung beim Erkennen und Verfolgen der Bälle erhofft. Im Laufe der Arbeit hat sich dann heraus gestellt, dass diese starre Annahme uns zu einer nur wenig robusten Lösung führte. In der am Ende des Projektes stehenden Lösung ist dann die Anzahl der jonglierten Objekte tatsächlich unerheblich und das System bestimmt an Hand der berechneten Daten sogar die Gesamtzahl der im Jongliermuster befindlichen Objekte.

Zunächst aber betrachten wir unser ursprüngliches Modell. In der ersten Grundüberlegung wird deutlich, dass ein Jongleur Jonglierbälle in einem Muster wirft, das möglichst gleichmäßig ist. So ist der Höhepunkt der Flugbahn idealerweise konstant auf der gleichen Höhe. Zum Analysieren des Jongliermusters wäre dies also bereits ein erstes Kriterium, die *Güte eines Jongliermusters* automatisiert zu bewerten.

Denkbar sind auch weitere Anwendungen, wie etwa das automatische Zählen von erfolgreich gefangenen Würfen. Eine computergesteuerte Erfassung der insgesamten Wurfzahl ist ein einfaches Kriterium für eine *Leistungsbewertung des jonglierenden Benutzers*.

Die genaue Anwendung ist jedoch nicht Ziel dieser Arbeit. Stattdessen verfahren wir in einem bottom-up Herangehen, um von den rohen Bild- und Tiefendaten der Kinect ausgehend Informationen über sich im Bild befindliche Objekte (Jonglierbälle) zu erfassen und deren Bewegung zu erkennen. Das Ergebnis ist dann ein Fundament, auf dessen Grundlage konkrete Anwendungen entwickelt werden können.

3 Zielsetzung

Am Ende dieser Arbeit soll eine Anwendung stehen, die mit Hilfe der Kinect Daten über die Flugbahnen dreier jonglierter Bälle liefert.

Ein Akteur befindet sich hierbei im Bildzentrum in einem wohl definierten Abstand zur Kinect. Es werden drei matte Bälle beliebiger Farbe jongliert. Um das Ergebnis unabhängig von der Szenenbeleuchtung zu halten, sollen die Tiefendaten ausreichend Information für das eindeutige Identifizieren der Bälle liefern.

Die Anwendung soll eine Grundlage liefern, um auf Grundlage den gewonnenen Information heraus konkrete weitere Anwendungen zu schreiben.

4 Möglichkeiten der Kinect

Die Kinect ist eine von Microsoft zur Spielekonsole Xbox 360 vertriebene Erweiterung, die den Spieler mit einem RGB- und einem Tiefensor erfasst und diese beiden Datenströme an die Konsole liefert. Da die Kinect über einen USB-Anschluss verfügt, kann sie an konventionellen Rechnern angeschlossen und betrieben werden. Eine quelloffene Implementierung zur Unterstützung der Kinect ist das freenect Projekt, das für Linux, Windows und MacOS zur Verfügung steht und im Rahmen dieser Arbeit als Bibliothek für Python verwendet wurde.^[5]^[6]

Die Videoquelle der Kinect liefert standardmäßig 30 Bilder pro Sekunde mit einer Auflösung von 640x480 und 8 bit Farbtiefe. Tatsächlich kann sie mit einer Auflösung von 640x512 aufnehmen (oder sogar 1280x1024 und 10 fps), aber die Auflösung wird an die der Tiefenkamera angepasst, so dass die reduzierte Auflösung ausgegeben wird. Die Tiefendaten stammen von einer Infrarot-Kamera und liefern bei gleicher Bildfrequenz 2048 verschiedene Tiefenwerte (11bit). Das Tiefenbild deckt einen Bereich von 0,8 m bis 3,5 m von der Kinect aus ab, bei der in 2m Entfernung eine Auflösung von 3 mm in der Bildebene und 1 cm in der Tiefenebene erreicht wird. Aus der Funktionsweise der Infrarotkamera ergibt sich, dass die Kinect in Umgebungen starker Infrarotstrahlung (beispielsweise im Tageslicht) nur beschränkt einsatzfähig ist.^[2]

FIXME: Abbildung Kinect mit gelabelten Komponenten?

Um das Tiefenbild zu erhalten, besitzt die Kinect zwei Hardware-Komponenten, einen Infrarot-Laser und eine Infrarot-Kamera. Der IR-Laser strahlt Licht mit einer Wellenlänge von 830 nm, welches ein bekanntes Rausch-Muster erzeugt. Dieses Rausch-Muster enthält mitunter neun hellere Punkte, die über das Muster verteilt sind.

Die Kinect nutzt die Streifenprojektion (auch Streifenlichttopometrie genannt), um ein räumliches Bild zu erfassen. So wird das vom Laser erzeugte bekannte Rauschmuster mit der in einem festen Abstand zum Laser befestigten IR-Kamera erfasst. Weicht das aufgenommene Bild von dem bekannten Rausch-Muster ab, kann davon ausgegangen werden, dass dies durch Objekte im Raum, die das Muster stören, verursacht wird. Mit Hilfe

der helleren Punkten werden alle Punkte zugeordnet und es kann ein räumliches Bild errechnet werden.

FIXME: Abbildung Rausch-Muster des IR-Lasers

Mögliche Störungen entstehen durch die Stärke der Laserdiode und der Wellenlänge des Lichts. Die IR-Kamera der Kinect hat einen Infrarot-Pass-Filter, der nur Licht im Bereich von 830 nm hindurch lässt, um nicht durch anderes Licht (wie beispielsweise Fernbedienungen) geblendet zu werden. Dennoch gibt es Lichtquellen, wie das Sonnenlicht, welches genügend Licht mit einer Wellenlänge von 830 nm beinhaltet, um die Kinect blenden zu können. So kann bei starkem Tageslicht oder im freien von Fehlverhalten bis zur Unnutzbarkeit der Kinect gerechet werden.

4.1 Beispielprojekte mit der Kinect

Mit der Kinect lassen sich viele Ideen umsetzen. So gibt es ein Projekt, bei dem Spielzeugautos und -figuren vor der Kinect verschoben werden, eine Software erkennt diese und stellt diese Bewegungsabläufe in einer virtuellen Welt als Animation dar.^[7] Weitere Projekte beschäftigten sich damit, ein 3D-Modell eines Menschen mit nur einer Kinect aufzunehmen,^[8] oder eine an eine Kamera befestigte Kinect mit zusätzlichem Bewegungssensor misst die Bewegungen beim Fotografieren, um die entstehende Bewegungsschärfe aus dem Foto hinauszurechnen.^[?] Eine Forschungsgruppe hat einem humanoiden Roboter mit der Kinect die Fähigkeit verliehen, ihm zugeworfene Bälle zu fangen.^[3] In der Videographie nimmt ein Projekt mit der Kinect RGBZ-Videos auf, in der nachträglich beispielsweise die Szenenbeleuchtung neu berechnet werden kann.^[9]

FIXME: Beispielkinectprojekte Bilder

5 Recherche: Ein jonglierender Roboter

Im Vorfeld der ersten eigenen Implementierungsversuche sind wir bei der Paper-Recherche auf eine Arbeit von Jens Kober, Matthew Glisson und Michael Mistry gestoßen, die zumindest in Teilen eine ähnliche Aufgabenstellung verfolgte. Unter dem Titel *Playing Catch and Juggling with a Humanoid Robot*^[3] untersuchte die Gruppe des Disney Research Center^[4] einen humanoiden Roboter, der auf ihn zugeworfene Bälle fangen und zurückwerfen soll. In der Entwicklung dieses Systems war ein Teil der Gesamtaufgabe ein bild- und tiefendatenverarbeitendes System, das mit einer der Kinect sehr ähnlichen Kamera arbeitete.

Kernidee dieses Systems war eine *Image Processing Pipeline*, also eine Kette von Verarbeitungsschritten, die als Eingabe eine Folge von RGB- und Tiefendaten nahm und als Ausgabe die aktuelle Position und zukünftige berechnete Flugbahn liefern.

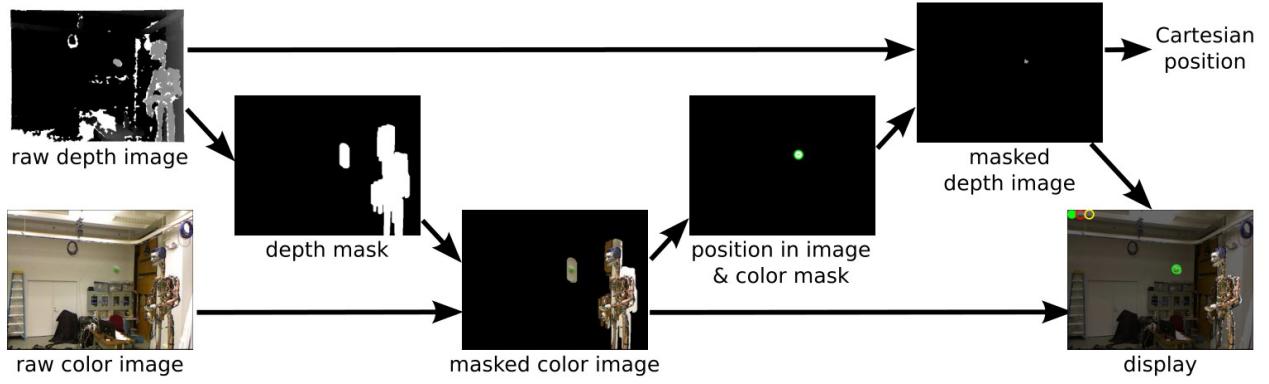


Abbildung 1: Bildverarbeitungs-Pipeline im Paper. (*Bildquelle: Playing Catch and Juggling with a Humanoid Robot^[3]*)

Die gelöste Aufgabe in dieser Arbeit ist also von der Grundidee her also eine ganz ähnliche, weshalb wir den Grundaufbau der Verarbeitungsschritte auch in unserem Vorgehen übernehmen wollten. Wie wir im Laufe der Programmierung aber feststellten, hatte die Arbeit einige Rahmenbedingungen anders gesetzt, so dass wir auf Probleme stießen, die sich in der Arbeit mit dem Roboter offensichtlich nicht so deutlich gezeigt haben.

Hierbei ist zuerst zu nennen, dass die betrachteten Bälle bei uns sehr viel kleiner waren, da wir einen tatsächlich jonglierenden Menschen betrachtet haben. In der Arbeit mit dem Roboter wurden größere Bälle verwendet, und zu einer Zeit befindet sich größtenteils immer nur ein Ball in der Luft.

Da in diesem Projekt die Bälle von einer Person dem Roboter zugeworfen werden, ergeben sich auch längere Wurfbahnen und eine längere Flugzeit pro Ball. Wie später in der Arbeit zu sehen sein wird, ist das normale Jongliermuster mit zwei Händen teilweise so klein, dass es schwer wird, dicht aneinander vorbei fliegende Bälle voneinander zu unterscheiden.

Zusätzlich wurde in der Arbeit mit dem Roboter auf verschiedenfarbige Bälle zugegriffen, um diese voneinander zu unterscheiden. Dies ist eine Rahmenbedingung, die wir so nicht wählen wollten, da die RGB-Werte, die die Kinect liefert, sehr stark von den Beleuchtungsbedingungen abhängen und bei den verwischteten Objekten, wie wir sie in den Kinect-Aufnahmen sehen, keine robuste Erkennung möglich ist.

6 Lösungsidee

Als erste Lösungsidee haben wir in Anlehnung an die betrachtete Arbeit mit dem jonglierenden Roboter eine eigene Zusammenstellung von durchzuführenden Bildverarbeitungsschritten erarbeitet wie in Abbildung 2 zu sehen.

Die Verarbeitung wird auf den Einzelbildern durchgeführt, die Ergebnisse der einzelnen Bilder werden dann kombiniert.

1. Auf den Tiefendaten werden ab einem bestimmten Tiefenwert (*threshold*) alle Tiefeinformationen abgeschnitten. Jongleur und fliegende Jonglierbälle sind somit freigestellt.
2. Auf den reduzierten Tiefendaten werden nun lokale Maxima bestimmt, welche die Jonglierbälle sein müssen, da sie sich näher an der Kinect befinden als der Jongleur. Hier sind auch die Hände enthalten.
3. Die gefundenen Maxima sind mögliche Kandidaten für tatsächlich erkannte Jonglierbälle.
4. Die entsprechenden Regionen werden als Maske für die RGB Daten verwendet.
5. In den interessanten Regionen der RGB Daten wird eine Erkennung für runde Objekte durchgeführt. Dies wird mit einer *Hough Transformation* gelöst.
6. Die nun erkannten Bälle werden aufgeteilt und den einzelnen Bällen zugeordnet.
7. Für jeden Ball liegt nun also eine Reihe von Positionen vor. Um zu einer flüssigen Flugbewegung zu gelangen wird die Flugbahn jedes Balls mit Hilfe eines *Kalman-Filters* modelliert und laufend aktualisiert. Mit dieser stetigen Positionsinformation kann die Position eines Balles also programmseitig jederzeit abgefragt werden.

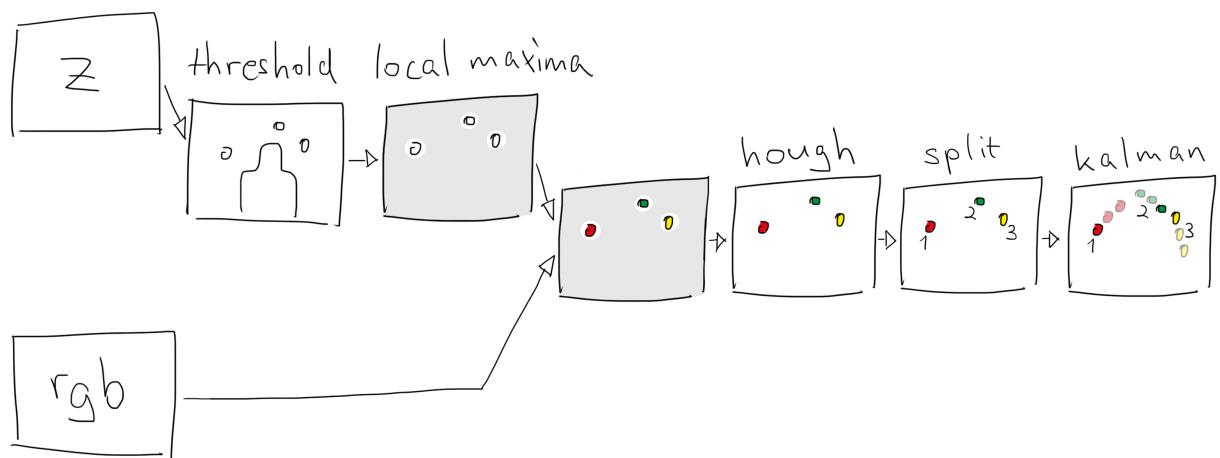


Abbildung 2: Unsere Bildverarbeitungs-Pipeline im ersten Entwurf.

Wie sich im Laufe des Projektes heraus gestellt hat, ist diese Vorstellung der Arbeit auf den Echtzeitdaten in einigen Schritten zu idealisiert und in anderen unnötig kompliziert.

Hierauf werden wir im finalen Schritt der Umsetzung eingehen.

7 Anfängliche Umsetzung

Im Folgenden beschreiben wir die Arbeitsschritte, wie wir sie an Hand unserer Grundplanung ausgeführt haben. Wir gehen dabei auf die Schwierigkeiten ein, auf die wir getroffen sind, wodurch dann deutlich wird, weshalb wir zu unserer endgültigen Umsetzung gekommen sind. Diese finale Umsetzung beschreiben wir dann im Abschnitt 8.

7.1 Programmstruktur

Die ersten Tests, die wir mit der Kinect und bildverarbeitenden Verfahren ausprobiert haben, bestanden aus wenigen Schritten und wurden in einer simplen Schleife gelöst, die bei Druck der ESC Taste verlassen wurde. Zu Beginn eines Schleifendurchlaufes wird ein neuer RGB Frame und die zugehörigen Tiefendaten geholt. Danach folgt schrittweise Verarbeitung dieser Daten je nach gewünschtem Zweck.

```
1 def loop(self):
2     """ Start the loop which is terminated by hitting a random key. """
3     while self.running:
4         if self.record:
5             self.snapshot()
6         else:
7             self._step()
8         key = cv.WaitKey(5)
9         self.running = key in (-1, 32)
10        if key == 32: # space bar
11            self.snapshot()
12
13    def _step(self):
14        """ One step of the loop, do not call on its own. Please. """
15        # Get a fresh frame
16        (rgb, depth) = self.kinect.get_frame()
17
18        # ... normalize values
19
20        # ... processing
21
22        # transform to open CV representation
23        rgb_opencv = cv.fromarray(np.array(rgb[:, :, ::-1]))
24
25        # Display image
26        cv.ShowImage('display', rgb_opencv)
```

Listing 1: Jeder Schleifendurchlauf entspricht einem Frame. Mit der SPACE Taste wird der aktuelle Frame als Bild gespeichert, jede andere Taste beendet die Schleife und somit das Programm.

Nun möchten wir je nach betrachtetem Arbeitsschritt aber verschiedene Verarbeitungsstufen ausführen oder ausklammern, so dass wir irgendwie eine Parametrisierung finden mussten. Wir haben uns für ein Konzept von Filtern entschieden, wobei zu Programmstart eine Liste mit gewünschten Filtern erstellt wird und diese in der Hauptschleife nacheinander ausgeführt werden (die Originaldaten werden also in den ersten Filter gegeben, das Ergebnis dieses Filters wird als Eingabe für den zweiten Filter verwendet und das Ergebnis des letzten Filters in der Liste wird dann als Gesamtausgabe visuell dargestellt).

Die RGB- und Tiefendaten werden vom freenect Modul als numpy Arrays zurückgegeben. Die Ausgabe erfolgt über ein von OpenCV erzeugtes Fenster, welches Daten für OpenCV erwartet. Die numpy Arrays müssen also an einer Stelle umgewandelt werden. Zusätzlich haben wir schnell festgestellt, dass einige gewünschte Operationen entweder nur in numpy oder nur in OpenCV zur Verfügung stehen. Ein mehrfaches Umwandeln von einem Format in das jeweils andere ist aus Performanz-Gründen natürlich zu vermeiden. Um in der Entwicklung nicht immer darauf achten zu müssen, welcher Filter welche Eingabe erwartet und welche Ausgabe liefert, haben wir uns dazu entschieden, jeden Filter als Eingabe Daten in numpy Darstellung zu liefern und auch für die Ausgabe numpy zu erwarten. Dies erlaubt uns eine von außen identische Betrachtung der Filter, auch wenn einige Filter intern eine Umwandlung durchführen müssen. Bei etwaigen Problemen wollten wir die Performanz unserer Lösung getrennt betrachten, um zu Beginn lediglich über eine funktionierende Lösung nachdenken zu müssen.

Waren die Filter ursprünglich als isolierte Einheiten mit simplem Input-Output-Verhalten von Bild- bzw. Tiefendaten gedacht, fiel uns schnell auf, dass einige Filter Zusatzinformationen liefern, die von anderen Filtern gebraucht werden. Um Filter nicht intern Unteraufrufe von anderen Filtern oder Komponenten ausführen zu lassen (dies wäre ebenfalls eine denkbare Lösung), entschieden wir uns für ein Objekt, das von der Hauptkomponente des Programms in jeden Filter hineingereicht wird und in dem jeder Filter Informationen ablegen und abfragen kann. Natürlich ist dies abhängig von der Reihenfolge der Filter und einige Filter haben als implizite Vorbedingung, dass andere Filter bereits ausgeführt wurden, dies haben wir der Einfachheit halber aber nicht expliziert formuliert, im Zweifelsfall wird beim ersten Ausführen einer nicht korrekten Filterkombination oder -reihenfolge ein Laufzeitfehler geworfen, da Informationen in dem übergebenen Objekt noch nicht vorhanden sind. Dieses Objekt ist ein schlichtes Python dictionary.

FIXME: Diagramm Programmstruktur (so cool skizziert, nicht ganz formell UML)

FIXME: bisschen Python Listing um diese Filter-Konzepte und das args dictionary zu zeigen

Genau diese Problematik, dass die Abhängigkeiten von Filtern nur implizit festgelegt ist und zur Laufzeit Fehler auftreten, weil eine benötigte Information im `args` dictionary nicht vorhanden ist, hat uns aber noch nicht zufrieden gestellt. Daher haben wir das Filterkonzept weiter angepasst. Jeder Filter verfügt nun über eine `filter` Methode mit drei zwingenden und einem optionalen Parameter. Das `argv` Dictionary wird nun lediglich für das Ablegen von Meta-Informationen verwendet, wie etwas Zusatzinfos zur Visualisierung.

Als Ergebnis des Filter wird ein Drei-Tupel zurückgegeben, so dass ein minimaler Filter ohne Funktionalität wie folgt aussieht.

```

1 class DummyFilter(object):
2
3     def filter(self, rgb, depth, balls, argv = {}):
4
5         # ...
6
7         return rgb, white, balls

```

Listing 2: Grundstruktur eines Filters. In diesem Beispiel werden die Eingabedaten unverändert zurückgegeben.

Zur besseren Übersicht über die Vielzahl der Filter haben wir diese für die finale Umsetzung in Packages organisiert. Jeder Filter liegt dabei in einem eigenen Modul vor (genauso wie jede der Ballerkennungsmethoden, auf die wir später zu sprechen kommen). Die Strukturierung in vier Packages je nach Aufgabe der einzelnen Komponenten erlaubt eine übersichtliche Codebasis und selbsterklärende `import` Statements.

Package	Inhalt
<code>src.kinect</code>	Module zum Ansprechen der Kinect und für Demodaten
<code>src.preprocessing</code>	Filter der Datenvorverarbeitung
<code>src.balldetection</code>	Hand- und Ballerkennung
<code>src.visual</code>	Visualisierung und finale Ausgabe der Daten
<code>src.application</code>	Beispielhafte Anwendungen auf Basis unseres Projekts

7.2 Programmfluss

7.2.1 Schritt 1: Tiefendaten vorverarbeiten

Die Kinect liefert Tiefenwerte im Bereich von 0 bis 2047, wobei ein höherer Wert einen größeren Abstand bedeutet. Die 0 steht hierbei für einen Pixel, in dem keine Tiefeninformation vorliegt.

Für unsere Aufgabe ist Beschaffenheit des Hintergrundes irrelevant. Außerdem nehmen wir einen isoliert stehenden Jongleur an. Zwischen der Tiefenebene des Jongleurs und dem Sensor befinden sich laut unserer Annahme keine Objekte außer den Jonglierbällen. Ein erster Schritt besteht daher im Binarisieren der Tiefendaten. Ab einem festen Abstand zur Kinect werden alle weiter entfernt liegenden Werte auf 0 (=schwarz) gesetzt. Alle näher am Sensor befindlichen Punkte werden auf 2047 (=weiß) gesetzt. Der Schwellwert, an dem diese Binarisierung getroffen wird, ist von uns zuerst auf 2100 gesetzt worden, da dies in den von uns aufgezeichneten Testdaten genau einer Grenze direkt vor dem Körper des Jongleur entsprach, so dass wir die Bälle und Hände des Jongleurs isoliert hatten. Im weiteren Verlauf haben wir mit einer dynamischen Bestimmung dieser Grenze experimentiert, letzten Endes dann aber doch diesen festen Wert belassen. Bei der tatsächlichen Anwendung in Echtzeit positioniert man sich also an einen festgelegten Punkt im Raum. Dies ist tatsächlich auch nicht störend. Da direktes visuelles Feedback von unserer Anwendung deutlich macht, wann man richtig steht, ist auch keine feste Bodenmarkierung nötig.

FIXME: Bild Tiefendaten -> Binarisiertes Bild

Auf Grund der Funktionsweise der Kinect mit dem RGB-Sensor und dem knapp daneben befindlichen Infrarotsensor sind die RGB- und Tiefendaten leicht gegeneinander verschoben. Hierbei hängt die Verschiebung vom Abstand eines Raumpunktes zur Kinect ab (Parallaxeffekt). Zwar arbeitet unsere Analyse nur auf den gelieferten Tiefendaten, so dass wir für die Ballerkennung diese Verschiebung ignorieren könnten. Für die Visualisierung überlagern wir aber die gewonnene Information mit den RGB-Daten, so dass wir hierfür diesen Fehler korrigieren müssen. Diese Korrektur ist daher auch Teil der Vorverarbeitung. Glücklicherweise unterstützt die `freenect` Bibliothek in der zum Projektzeitpunkt nur als `unstable` erhältlichen Version diese Fehlerkorrektur von Haus aus. Nötig ist hierzu lediglich die Angabe eines bestimmten Formats beim Anfordern eines Frames von Tiefendaten: `freenect.get_depth(format=4)` Erzeugt dieser Aufruf einen Fehler, so ist dies ein Hinweis, dass nicht die korrekte Version der `freenect` Bibliothek verwendet wird.

FIXME: Löcher im Tiefenbild

Hatten wir ursprünglich noch überlegt, die Tiefenwerte von 256 bit auf einen niedrigeren Wert zu reduzieren, um in der Verarbeitung der Einfachheit halber nur mit Werten zwischen 0 und 255 arbeiten zu müssen, so haben wir uns dann doch gegen diese Reduzierung entschieden. Lediglich für den Fall der Visualisierung der Tiefendaten führen wir eine entsprechende Transformation durch, um auf Grauwerte zwischen 0 und 255 abzubilden.

7.2.2 Schritt 2: Regions Of Interest isolieren

Um den Hintergrund zu entfernen und die Bälle freizustellen, wurden zwei Ansätze getestet:

- A. Möglichst keine Objekte in der Tiefenebene zwischen dem Spieler und der Kinect platzieren, auch nicht am Rand. Die Tiefenwerte werden nun ab einem gewissen Wert einfach abgeschnitten, so dass in diesen im optimalen Fall nur noch die Bälle sowie die Hände des Jongleurs verbleiben. Hier wird also die Annahme getroffen, dass der Spieler auf einer Linie oder ähnlichem in einem festen Abstand zur Kinect steht. Die Tiefendaten werden nun noch binarisiert.
- B. Mit einem temporalen Filter sich bewegende Regionen isolieren. Dieses erlaubt auch störende Objekte wie Stühle am Rand sowie einen flexiblen Abstand des Spielers zur Kinect. Nach einer Implementierung wurde aber deutlich, dass das Verfahren bei schnellen Bewegungen (wie dem Ballwurf) nicht zuverlässig ist. Außerdem traten vermehrt Probleme mit unscharfen Objekträndern und Rauschen auf, die aber mit Funktionen der VIGRA^[10] kompensiert werden konnten. Die Probleme mit schnellen Bewegungen verlieben aber und es kamen noch technische Hürden hinzu, wie die VIGRA^[10] als weitere Abhängigkeit.

Wirstellten fest, dass Ansatz A völlig ausreichend für unsere Zwecke ist. Die Einschränkung der Spielerposition ist ebenfalls nicht störend, da dies sogar interaktiv durchgeführt werden kann (so lange nach vorne gehen, bis das System vernünftige Werte liefert - kein

aufwändiges Abmessen nötig). Weiterhin redgt dies sogar den Spieler an, seine Ballwürfe kontrolliert durchzuführen, ohne dass diese ihn zu Positionsveränderungen zwingen.

7.2.3 Schritt 3: Bälle in Frame-Folgen einander zuordnen

Bevor potentielle Ballpositionen konkreten Ball-Instanzen zugeordnet werden, findet eine kurze Vorverarbeitung statt, bei der Regionen im binarisierten Ergebnis des vorherigen Schrittes als Rechtecke umzeichnet werden, wofür einfach die OpenCV-Funktion `cv.BoundingRect` verwendet wird, die genau diese Aufgabe erfüllt. Kleine Rechtecke und solche, die den Bildrahmen berühren, werden hierbei bereits herausgefiltert.

Durch die resultierende Liste von Rechtecken wird iteriert und jeder Rechteckmittelpunkt als mögliche Ballposition aufgefasst (die kürzere Rechteckskante wird als Balldurchmesser gespeichert, vorerst aber ignoriert).

```

1 # ...
2
3     storage = cv.CreateMemStorage(0)
4     contour = cv.FindContours(depth_cv, storage, cv.CV_RETR_CCOMP, cv.
5         CV_CHAIN_APPROX_SIMPLE)
6     points = []
7
8     ball_list = [] # collect ballpositions in loop
9     while contour:
10        x,y,w,h = cv.BoundingRect(list(contour))
11        contour = contour.h_next()
12
13        # filter out small and border touching rectangles
14        t = 2 # tolerance threshold
15        minsize = 5
16        if x > t and y > t and x+w < self.WIDTH - t and y+h < self.HEIGHT - t and w
17            > minsize and h > minsize:
18            x -= 5
19            y -= 5
20            w += 10
21            h += 10
22            x, y = self._nullify(x), self._nullify(y) # why is this necessary now?
23
24            ball_center = (x+w/2, y+h/2)
25            ball_radius = min(w/2, h/2)
26            ball_list.append(dict(position=ball_center, radius=ball_radius))
27
28            # Draw rectangle with info
29            cv.PutText(rgb_cv, '%d/%d' % (x, y), (x,y-2) , self.font, (0, 255, 0))
30            cv.Rectangle(rgb_cv, (x, y), (x+w, y+h), cv.CV_RGB(0, 255,0), 2)
31
32    args['balls'].addPositions(ball_list)
```

Listing 3: RectsFilter.py, Ausschnitt

Es liegt nun für jeden Frame eine Liste von möglichen Ballpositionen vor. Diese setzen

sich aus tatsächlichen Ballpositionen, Positionen in denen sich die Hände befinden sowie potentiell weiteren Hindernissen im Sichtfeld der Kinect zusammen, wobei wir diese in unseren Testdaten vermieden haben. Bei den tatsächlichen Ballpositionen ist zu beachten, dass diese etwas ungenau sind und mit mehreren Pixeln vom tatsächlichen Ballmittelpunkt abweichen können.

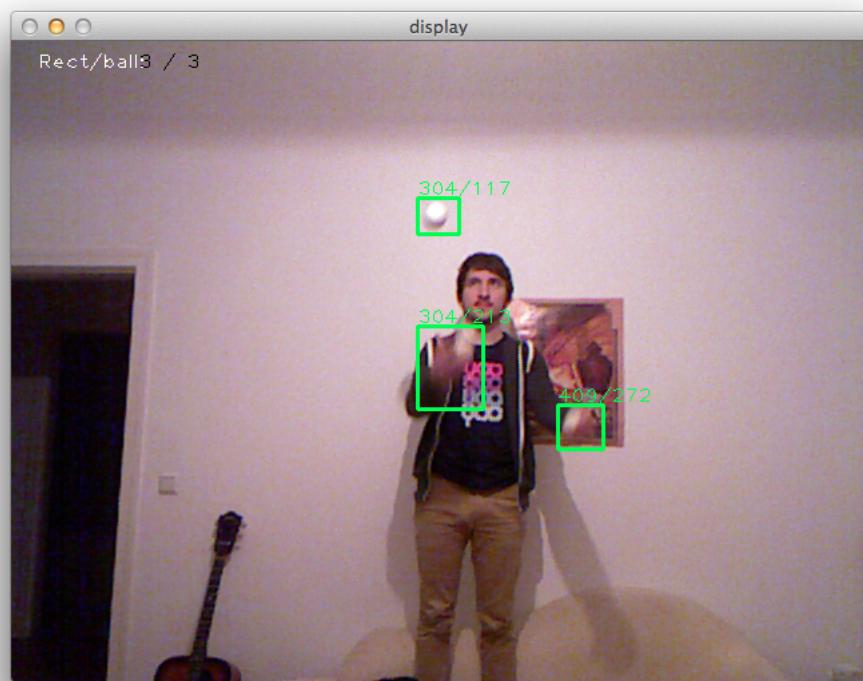


Abbildung 3: FIXME

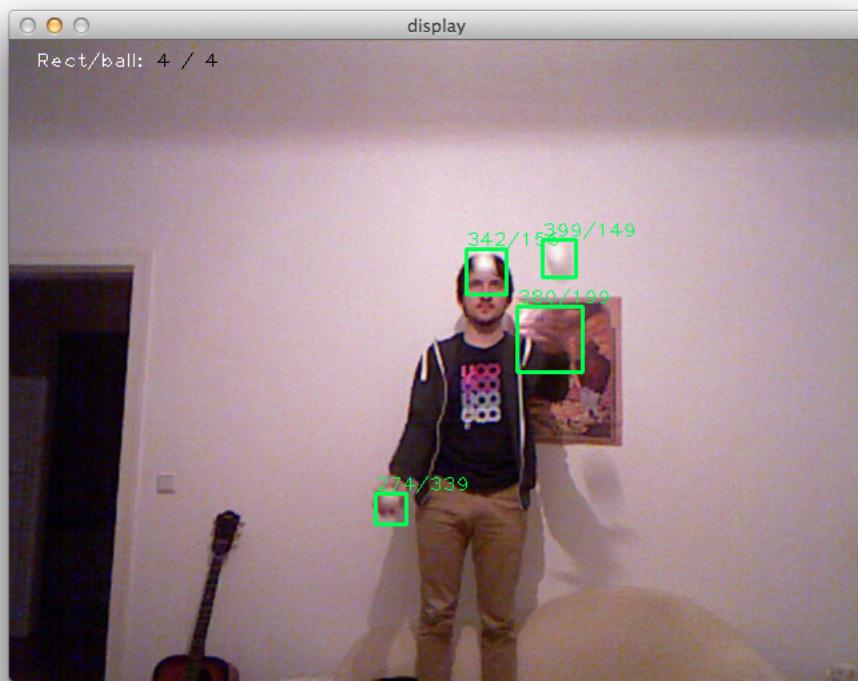


Abbildung 4: FIXME

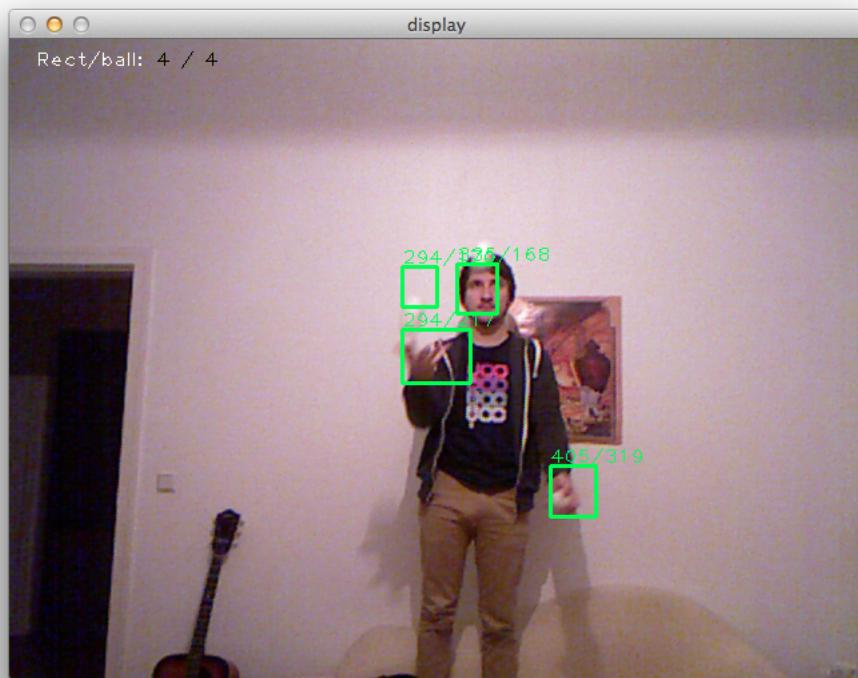


Abbildung 5: FIXME

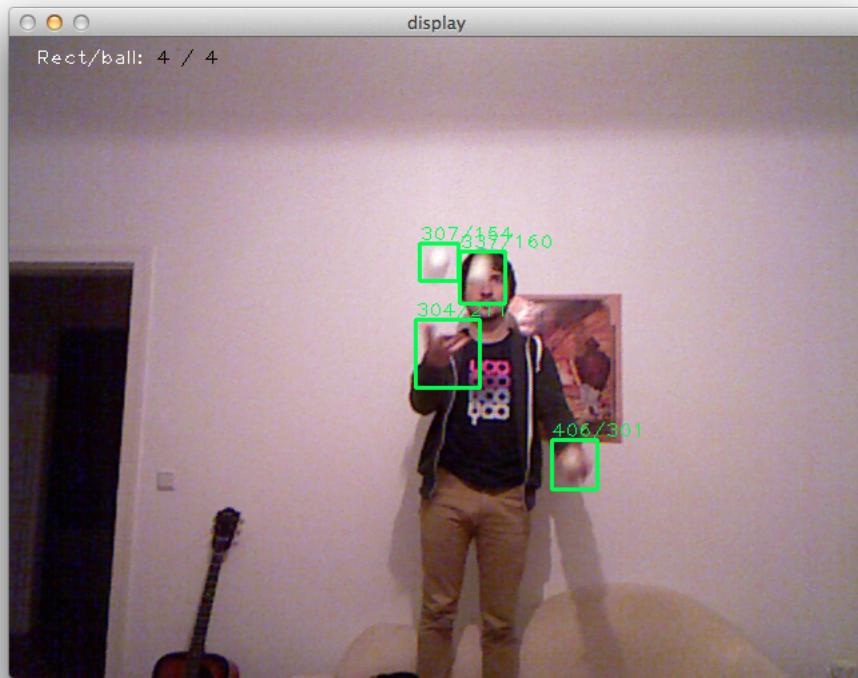


Abbildung 6: FIXME

Im Folgenden kommt die Hauptaufgabe dieses Schritts: Das Zuordnen der potentiellen Punkte zu tatsächlichen Ballinstanzen, wobei diese Zuordnung über mehrere Frames hinweg eine neue Position einem bestehenden Ball zuordnen soll.

Dies ist, wie es sich herausgestellt hat, der aufwändigste Schritt - zumindest der, mit dessen Lösung wir die meiste Zeit verbracht haben.

Als erste Idee, um falsche Positionen auszuschließen, dachten wir an eine Ballerkennung auf den RGB-Daten im Umfeld der Position. Naheliegend war die Implementierung einer Hough-Transformation, mit der wir auch begonnen haben. Erste Testläufe auf den RGB-Daten der Kinect zeigten aber, dass je nach Wahl der Parameter entweder die Bälle nicht erkannt, oder aber viel zu viele false-positives erkannt wurden. Dies lag vor allem an den sehr verwischten Aufnahmen der Bälle während des Fluges. Bei Betrachtung der Beispielaufnahmen (siehe oben) wird auch deutlich, dass eine vermutete Position teilweise deutlich von der tatsächlichen Position in den Bilddaten abwich. Somit konnten wir durch diesen Ansatz keinen Gewinn ziehen, so dass wir uns auch durch die Abweichungen zwischen Ballposition in RGB- und Tiefendaten dazu entschlossen, uns alleine auf eine Untersuchung der Tiefendaten zu stützen.

Aber auch für die reine Berücksichtigung der Tiefendaten haben wir während des Projektablaufes mehrere Problemquellen identifizieren können, die die verschiedenen Ansätze unterschiedlich stark beeinflussen:

- Die Hände sind auch in den möglichen Ballpositionen als Rechtecke enthalten.

- Die Bälle fliegen sehr nahe beieinander und teilweise überschneiden sich die Rechtecke zweier Bälle, so dass nur ein großes Rechteck zu sehen ist und als eine mögliche Ballposition untersucht wird.
- Ein Ball legt in einem Frame (1/30 Sekunde) unterschiedlich lange, teilweise sehr große Strecken zurück. (Pixelanzahl angeben?)
- Der Mindestabstand zur Kinect resultiert in einem kleinen Jongliermuster, wodurch die problematischen Faktoren verstärkt werden.
- In aufeinander folgenden Frames wird zum Teil eine Region in einigen der Frames nicht erkannt, in anderen schon.

Diese Problemquellen wirken sich unterschiedlich stark auf die Eignung der von uns im folgenden betrachteten Ansätze aus. Prinzipiell lassen sich die Ansätze in zwei Gruppen unterteilen: In die erste Gruppe, bei der von vornehmesten einer festen Ballanzahl ausgegangen wird und die zweite Gruppe, bei der auch eine dynamisch wechselnde Ballanzahl erlaubt ist.

Bei den Ansätzen mit fester Ballanzahl wird in jedem Verarbeitungsschritt versucht, zu jeder bereits existierenden Ballinstanz eine Position aus den erkannten Rechtecken auszuwählen, welche als neue Position des Balls festgelegt wird. Da im Allgemeinen nie zwei Bälle an der gleichen Position sein können, wird dieses Rechteck dann für die weiteren Ballinstanzen nicht mehr betrachtet, wir nennen die folgenden Ansätze daher auch *konsumierende Ansätze*. FIXME: das ist falsch, konsumierend \neq feste Ballanzahl.

1. Die zwei sich am nächsten Punkte in zwei aufeinander folgenden Frames werden als der identische Ball aufgefasst. Nicht so zuverlässig, vor allem wegen schneller Ballbewegungen und nahe aneinander passierender Bälle. Schwierig auch, wenn ein erkannter Ball fehlt → Beachtung von "springenden" Bällen.
2. Verbesserungsansatz: Die erwartete Ballposition wird mit der vorherigen Bewegung (linearer Bewegungsvektor) approximiert. Dies ist teilweise besser, aber trotzdem noch schwierig, die initiale Bewegung zu erkennen. Hierbei verbleiben auch weiterhin Probleme mit Lücken in den Informationen.
3. Weitere Verbesserung: Keinen linearen Bewegungsvektor nutzen, sondern die Flugbahn vorberechnen. Der lineare Bewegungsvektor wird als Tangente an der Steigung der Wurfparabel zu Grunde gelegt. (Verbesserung nochmal gut angucken, aber gefühlt hat das erstaunlich wenig unterschied gebracht)

Im Gegensatz dazu sind die folgenden nicht konsumierende Ansätze zu sehen, bei denen wir auch eine variable Ballanzahl erlauben. FIXME: das ist falsch, siehe oben

1. Wenn eine langsame Aufwärtsbewegung in aufeinander folgenden Frames erkannt wird: als Beginn eines Wurfes auffassen und an dieser Stelle einen Ball mit identischer Geschwindigkeit starten und dessen Flugbahn ab dort schrittweise simulieren. In jedem Schritt mit aktuell vorhandenen Bällen abgleichen und Wurfparameter anpassen. (hier schrittweise Bilder zeigen: Feuerwerk etc)
2. Wie der Ansatz davor, allerdings wird die Parabel nicht approximiert, sondern aus

den letzten 3 Frames mit `np.polyfit` berechnet.

7.2.4 Schritt 4: Bereinigte Wurfparabel

Das fehlt uns noch. Aber aus den gelieferten Daten wollen wir dann höher-levelige Informationen abstrahieren. Objektanzahl, Wurfhöhen, Würfe zählen, etc.

7.3 Erläuterung verwendeter Bildverarbeitungsverfahren

7.3.1 Temporaler Buffer

FIXME Thiemo

7.3.2 Kalman Filter

FIXME: Schaubilder

Der Kalman Filter ist ein in 1960 von Rudolf E. Kálmán vorgestellter Algorithmus, welcher aus mehreren ungenauen Messungen versucht, die tatsächlichen Daten, die diesen Messungen zu Grunde liegen, zu estimieren.^[13] Solche Ungenauigkeiten können durch die Messsensoren entstehen, wenn diese aus verschiedenen Gründen nicht präzise genug aufnehmen können, oder auch durch ein Rauschen der Aufnahmequelle. Mit dem Kalman-Filter können so über einen bestimmten Zeitraum aufgenommene Daten mit möglichst vielem Wissen über potentielle Messabweichungen genauer bestimmt werden, als mit nur einzelnen Messungen.

Dieser Algorithmus wird heutzutage in vielen Bereichen eingesetzt, sehr oft aber in der Navigationstechnologie von Fahrzeugen aller Art. Die Position eines Fahrzeugs wird beispielsweise über GPS empfangen. Diese Technologie kann selbst mit einer zusätzlichen Korrektur (WAAS/EGNOS-Korrektursignale mit Hilfe von Bodenstationen^{[14][15]}) die Position auf ca. 5-20m bei reinem GPS und etwa 1-3m genau mit Korrektur bestimmen. 1-3m Genauigkeit kann bei privaten Fahrzeugen mit Navigationsgerät reichen, dennoch wäre es merkwürdig aus, würde die eigene Position selbst bei Stillstand hin und her springen. Bei der Navigation in einer Großstadt kann bei 3m Ungenauigkeit die kleine Gasse verfehlt werden, und auf der falschen Straßenseite fahren täte man laut Gerät auch andauernd. Weiterhin gibt es Fortbewegungsmittel, die vielleicht sogar ein automatisches Fahrsystem verbaut haben, bei dem eine exakte Positionsbestimmung notwendig wird.

So können weitere Sensoren verbaut werden, die zusammen mit dem GPS eine genauere Positions- und Bewegungsbestimmung ermöglichen, aber selbst bei diesen können erneut Messungenauigkeiten auftreten. Mit dem Kalman Filter können so alle aufgenommene Daten mit dem Wissen über die Art der Bewegung und Annahmen zu Messfehlern verfeinert werden. Zusätzlich kann der Algorithmus mit den bisher gesammelten Daten eine Schätzung über die Position in der Zukunft machen. Wie erwähnt nutzt dieser Algorithmus alle bisher aufgenommenen Daten, jedoch werden diese als Markow-Kette modelliert,

so dass eine interne Formel immer mit den neusten Daten angepasst wird und keine Iteration über alle vergangenen Daten notwendig ist. Dadurch können diese Berechnungen in Echtzeit erfolgen, was sie für die Navigationstechnologie so wertvoll macht.

Eine einfache Implementierung dieses Kalman Filters besitzt zwei Funktionen, einem `predict` und einem `update`.

Im `predict` wird mit Informationen zur Art der Bewegung und mit mindestens einer Ortsinformation zum Zeitpunkt $t-1$ eine Schätzung zum Ort im Zeitpunkt t gemacht. Zusätzlich werden hierbei die Sensordaten im Zeitpunkt t geschätzt. Ein `predict` kann beliebig oft hintereinander ausgeführt werden.

$$\bar{x}_t = A_t x_{t-1} + B_t u_t + \varepsilon_t \quad (1)$$

$$\bar{z}_t = H_t \bar{x}_t + \varepsilon_t \quad (2)$$

In Formel (1) wird die voraussichtliche Position zum Zeitpunkt t , \bar{x}_t , berechnet. Hierbei ist A_t das Bewegungsmodell, das auf die Position zum Zeitpunkt $t-1$ angewandt wird, also bei einem Kraftfahrzeug wäre dies die Geschwindigkeit und Richtung im Zeitabschnitt $t-1$ bis t , welche auf die vorige Position des Fahrzeugs gerechnet werden, um die aktuelle Position zu bestimmen.

u_t ist ein Vektor, der bekannte Störungen darstellt, multipliziert mit einer Matrix B_t , welche dynamisch angepasst wird.

ε_t ist noch zusätzliches Rauschen, welche mit einer Normalverteilung einbezogen werden.

In Formel (2) wird die Ausgabe der Sensoren zum Zeitpunkt t , \bar{z}_t , geschätzt. Es wird hierbei die geschätzte Position \bar{x}_t mit einer Beobachtungsmatrix H_t multipliziert, welche die Abweichung von tatsächlicher Position zu beobachteter Position beinhaltet. Zusätzlich wird hier ebenfalls ein Rauschen in Form einer Normalverteilung hinzugefügt.

Im `update` wird die aktuell gespeicherte Ortsinformation mit den extern gewonnenen Sensordaten aktualisiert. Die hier gemessene Abweichung zwischen Schätzung und tatsächlichen Daten kann natürlich auch zur weiteren Schätzung eingebracht werden. Ein `update` wird nicht mehrmals hintereinander ausgeführt.

$$x_{estimate} = \bar{x}_t + K(z_t - \bar{z}_t) \quad (3)$$

Formel (3) stellt ein `update` dar. Mit der Abweichung zwischen den neuen Sensordaten z_t und den geschätzten Sensordaten \bar{z}_t wird die vorher geschätzte Position zum Zeitpunkt t , \bar{x}_t , angepasst, so dass die angenommene Position $x_{estimate}$ errechnet wird.

Einem `update` geht immer ein `predict` voran, jedoch kann ein `predict` beliebig oft, mit neuen Zeiten, hintereinander ausgeführt werden. Hierdurch werden Schätzungen über die Position des beobachteten Objekts zu weiteren Zeiten berechnet, deren Genauigkeit, abhängig von Art der Bewegung und Abweichungen von dieser, meist mit größerer Zeit ungenauer werden. Jedoch können diese bei ausreichendem Wissen über Bewegung und Störungen für eine gewisse Zeit ausreichend genau sein, um mit ihnen zu arbeiten.

In diesem Projekt sollte er bzw. eine Vereinfachung dessen dazu verwendet werden, durch Beobachten der einzelnen Bälle die voraussichtlichen Flugbahnen dieser zu bestimmen,

um die Möglichkeit zu haben, diesen in den Ausgabestream als Lernhilfe einzzeichnen. Aber selbst hierfür muss eine Beobachtung von einzelnen Bällen mit einer eindeutigen Zuordnung dieser von Frame zu Frame funktionieren, wofür der Algorithmus ebenfalls eingesetzt wird. Denn selbst wenn von Frame zu Frame alle Bälle erkannt werden, besteht die Hürde, einen Ball aus einem Frame eindeutig demselben Ball im nächsten Frame zuzuordnen.

So ist in diesem Projekt das bewegte Objekt ein Ball und die Art der Bewegung eine Wurfparabel. Der Einfachheit halber und da in diesen Zeiten vorkommende Abweichungen klein wären, außerdem eine Genauigkeit auf Pixelebene nicht notwendig ist, sind keine Störfaktoren einberechnet worden. Als Sensor wird die Kinect benutzt, mit der auf vorher beschriebene Weise die einzelnen Bälle gefunden werden.

7.4 Herausforderungen

Probleme aus den Ansätzen noch mal aufgreifen. Noch irgendwas abstrakteres dazu schreiben? Vielleicht dass wir uns nicht doll genug getracked haben dier Projektzeit über? Vielleicht nochmal die Reihenfolge sortieren.

Während dieses Projektes wurden viele Ideen eingebracht, einige sofort implementiert, andere nach mehreren Versuchen nutzbar gemacht. Aber einige Ansätze konnten auch nach wiederholten Versuchen nicht ausreichend umgesetzt werden, oder schienen nach mehrmaligem Testen nicht den erhofften Fortschritt hineinzubringen, weswegen sie letzten Endes nicht in der Software genutzt werden.

Einige dieser Ansätze, die uns Schwierigkeiten bereitet haben, werden hier kurz vorgestellt, das Problem der Idee aufgezeigt, und wie wir sie letzten Endes doch implementiert haben - oder warum sie es nicht in die Software schafften.

Leider machten wir während des Projektes wenige Aufzeichnungen über unsere Arbeit, sondern schrieben wild drauf los, und was nicht funktionierte, wurde überschrieben. Deswegen ein Reminder an uns: Mehr Kommentieren und Fortschritt sowie Missglücktes notieren!

7.4.1 Hough-Transformation

Die Hough-Transform wurde in ihrer heutigen Form 1972 von Richard O. Duda und Peter E. Hart^[11] vorgestellt.

Ursprünglich wurde das Verfahren zum Finden von geraden Linien von Hough patentiert, jedoch dann von Duda und Hart verbessert und auch das Finden von Kreisen bzw. Kurven ermöglicht.

Das Finden von Geraden wird so erreicht, dass für jeden Kantenpunkt (die vorher mit einem Kantendetektor, wie z.B. dem Canny-Algorithmus, gefunden werden) alle möglichen Geraden, die durch diesen Punkt gehen, aufgestellt werden, und von diesen die Parameter im Parameterraum abgebildet werden.

Anfangs wurden die Steigung und der y-Achsenabschnitt verwendet, doch da sich Probleme bei Kanten parallel zur y-Achse ergaben, wird nun die Hessesche Normalform der Geraden verwendet und als Parameter ein Winkel α , in welchem ein Vektor vom Koordinatenursprung orthogonal eine Gerade schneidet, die durch den zu betrachtenden Punkt geht, verwendet. Der weitere Parameter ist die Länge dieses Vektors, bzw. der Abstand vom Ursprung bis zur Geraden, die orthogonal unseren Hilfsvektor schneidet.

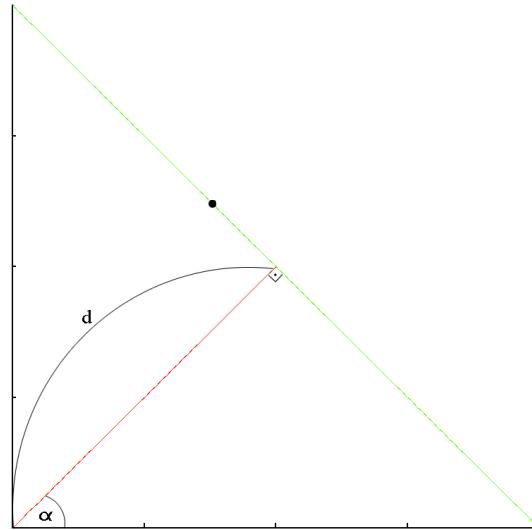


Abbildung 7: Anstatt Steigung und y-Achsenabschnitt werden als Parameter α und d genommen.

Wenn dies nun für alle vorhandene Kantenpunkte gemacht wird, werden im Parameterraum eine dementsprechende Anzahl von Kurven (oder Geraden) abgebildet, die einen exakten, oder ungefähren Schnittpunkt besitzen sollten. Die Parameter, die an diesem Punkt abgelesen werden, stellen die Parameter für unsere Kante im Bild durch die Kantenpunkte dar.

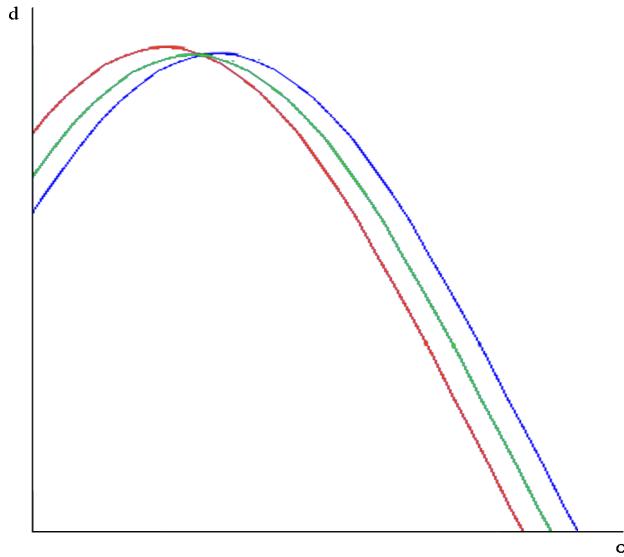


Abbildung 8: Überschneidung der Kurven im Parameterraum geben an, welche Parameter für die gesuchte Gerade zu nutzen sind.

Möchte man nun einen Kreis im Bild finden, werden von den Kantenpunkten alle möglichen Kreisgleichungen aufgestellt, dementsprechend als Parameter der Radius und der Mittelpunkt (x, y) gewählt, womit wir einen dreidimensionalen Parameterraum haben. Unter berücksichtigung eines Maximalwertes für den Radius tut man dies für alle Kantenpunkte, und anschließend lässt sich im Parameterraum wieder ein Schnittpunkt ablesen, an der sich die Parameter des gesuchten Kreises befinden.

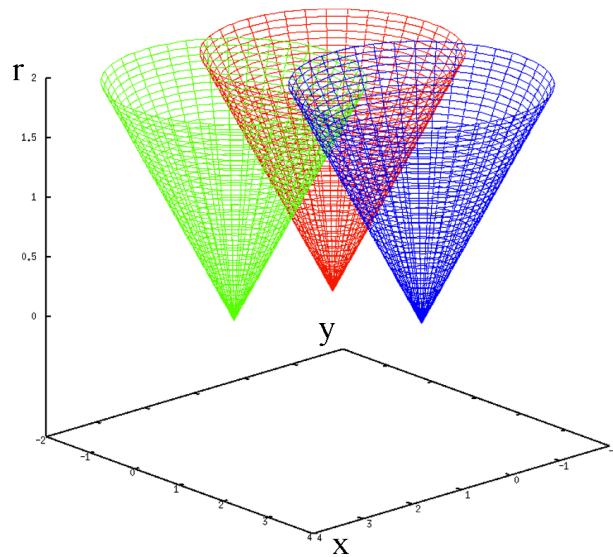


Abbildung 9: Es entstehen Kegeln, an denen sich wieder ein Schnittpunkt bildet.

Dieses kann man für beliebige Formen anwenden, die durch eine Parameterdarstellung definiert werden können, aber je mehr Parameter vorhanden sind, desto aufwändiger wird auch die Berechnung werden.

Dieses Verfahren sollte genutzt werden, um die genauen Ballpositionen im Tiefen- oder auch RGB-Bild zu bestimmen. Wenn möglich, sollte nach dem Filtern nach Regions of Interest im Tiefenbild auf diesen gefundenen Regionen die Hough-Transformation durchgeführt werden, um Kreise bzw. Bälle zu finden, oder auch Regionen ohne Bälle ausschließen zu können.

Genutzt haben wir die Implementation der Hough-Transformation in der OpenCV^[12]-Bibliothek. Hierbei wurde festgestellt, dass durch Einstellen der Parameter der Funktion die Kreiserkennung sehr fein bestimmt werden kann. Da, wie in Abbildungen 3 bis 6 zu sehen ist, die Bälle fast nie als saubere Kreise mit scharfen Kanten aufgenommen werden, war das Feintuning der Parameter unmöglich. Entweder wurde sehr grob eingestellt, so dass alle Bälle zwar erkannt wurden, dabei aber noch viel mehr False-Positives entstanden, oder es wurde so fein eingestellt, dass nur sehr sauber aufgenommene Bälle erkannt wurden. Bei letzterem wurden insgesamt aber viel zu wenige Bälle gefunden, so dass durch Einsetzen dieses Verfahrens in unserem Fall kein Vorteil, sondern nur Nachteile entstanden. Zudem wurde festgestellt, dass dieser Algorithmus viel Rechenleistung beansprucht und durch viel Kreisdetektion die ausgegebene Frame-Rate sank.

Die Überlegung war, dass durch eine Einbindung des Algorithmus als C-Code, z.B. mit Cython, dieser Leistungsverlust ausgeglichen werden könnte. Durch eine eigene Implementation hätte die Ballerkennung evtl. auch verbessert werden können, so dass nicht nach Kreisen gesucht wird, sondern nach Ellipsen (in bestimmter Ausrichtung). Da aber die ersten Versuche nicht sonderlich erfolgreich waren und der Aufwand nicht lohnenswert erschien, wurde dagegen entschieden. Durch geschicktes filtern nach unseren Regions of Interest konnten diese auch soweit verkleinert werden, dass eine zusätzliche Beschränkung dieser nicht notwendig wurde. Auch das Entfernen von False-Positives in den Regions of Interest wurde auf anderem Wege gelöst.

7.4.2 Kalman-Filter

kA, könnte hier noch 1-2 kleine Absätze zu unserem Vorgehen reinschreiben

7.4.3 Local Maxima

Mit Local Maxima werden lokale Maxima, also positive extreme Datenpunkte in einer lokal begrenzten Umgebung, bezeichnet.

Anfangs hatten wir die Idee, über lokale Maxima die Bälle im Tiefenbild zu finden. Dies würde bedeuten, dass im Tiefenbild nach allen Datenpunktanhäufungen gesucht wird, die möglichst nahe der Kamera sind, dabei aber innerhalb eines gegebenen Radius bleiben. Diese sollen zusätzlich von Datenpunkten umgeben sein, die einen deutlichen Abstand nach hinten zu diesen Datenpunktanhäufungen besitzen.

Umgangssprachlich würden wir also alle Objekte bestimmter Größe suchen, die sich in einem bestimmten Abstand zu ihrem Hintergrund befinden. Damit ließe sich ein flexibler

Abstand von Jongleur zu Kinect bewerkstelligen, es könnte damit sogar eine Ortsveränderung des Spielers während der Jonglage verarbeiten.

In unserem Projekt werden die Tiefendaten zumeist als Numpy-Arrays verarbeitet und von Filter zu Filter gereicht. So war hier die Idee, einen weiteren Filter zu erstellen, in dem im Numpy-Array nach lokalen Maxima bestimmter Größe gesucht wird, um eine Vorauswahl von Regions of Interest zu bekommen. In diesen sollten in weiteren Schritten False Positives ausgeschlossen werden, um am Ende nur Regionen mit Bällen zu erhalten, in denen mit z.B. der Hough-Transformation (zu diesem Zeitpunkt wollten wir diese noch nutzen) die Bälle extrahiert werden sollten.

Für das Finden von Maxima im Numpy-Array werden die Python Module Numpy^[16] und Scipy^[17] genutzt, welche eine große Auswahl an wissenschaftliche Funktionen besitzen. Besonders für mehrdimensionale Matrixberechnungen und weitere Bildverarbeitungsoperationen eignen sich diese Module sehr gut, da ihre Funktionen auf diese Probleme optimiert sind und meist schneller als eine eigene Implementierung sind.

Speziell mit den Funktionen `scipy.ndimage.filters.maximum_filter` und `scipy.ndimage.filters.minimum_filter` wurden auf einem Tiefenbild die Maxima und Minima bis zu einem festgelegten Durchmesser gesucht. Durch Abziehen der Minima von den Maxima konnte auf dem daraus resultierenden Array mit einem Schwellwert, welcher den Abstand eines Maxima zu seiner Umgebung beschreibt, lokale Maxima bestimmter Größe gefunden werden.

Dieses Verfahren, welches bei Testbildern, welche rauscharm mit einem gleichmäßigen Hintergrund sind, auf dem deutlich hellere Punkte verteilt sind, funktioniert gut. In unserem Falle aber bestand das Problem, dass bei verschiedenen Einstellungen der Parameter entweder zu viele Maxima, wie beispielsweise jede sich hervorhebende Falte in der Bekleidung des Spielers gefunden wird, oder aber zu wenige, so dass sich nicht alle Bälle immer in den Regions of Interest befinden würden.

So haben wir uns entschieden, keinen flexiblen Abstand des Spielers zur Kinect zuzulassen, sondern diesen festzulegen. Anfangs als Nachteil empfunden, stellten wir aber fest, dass dies schnell akzeptiert werden konnte und den Spielspaß nicht mindert.

Unsere finale Umsetzung zum Finden von Regions of Interest wird in Abschnitt 7.2.2 beschrieben.

7.4.4 Temporal Filter

FIXME Thiemo

7.4.5 Technische Herausforderungen

Im Laufe des gesamten Projektes haben wir drei Projektteilnehmer verteilt auf drei Betriebssystemen gearbeitet (Mac OS, Windows, Ubuntu). Da wir gerade dank unseres flexiblen Filterkonzeptes gut arbeitsteilig arbeiten konnten, haben wir nicht von Anfang an auf die Integrierbarkeit unserer Komponenten geachtet.

Der funktionierende Temporalfilter (siehe 7.4.4) etwa wurde mit Hilfe der Bildverarbeitungsbibliothek VIGRA (FIXME source) implementiert. Die Python-Bindings hierfür liegen nicht auf allen Systemen in gleich guter Verfügbarkeit vor (zumindest war es uns auf Mac OS nicht möglich, diese ohne Mac Ports zum Laufen zu bekommen, da die per `pip install` verfügbare Version veraltet ist und scheinbar nicht mehr gepflegt wird. (FIXME source)

Da der Temporalfilter zum einen der einzige Filter mit VIGRA Abhängigkeit war, und der Filter zudem nicht die gewünschten Vorteile geliefert hat, die wir uns erhofft hatten, haben wir diesen Filter nicht weiter in der Entwicklung betrachtet und so auch die zusätzliche Dependency eingespart.

8 Finale Umsetzung

8.1 Bewertung der Umsetzung

Robustheit.

- Position des Jongleurs (Bildzentrum VS Bildrand. Abstand zur Kinect)
- Objekte am Bildrand (bewegte vs nicht bewegte)
- Ungleichmäßiges Jongliermuster
- Variation der Bälleanzahl

Effizienz. Speedup-Möglichkeiten?

Anwendungsrelevanz.

9 Anwendungsmöglichkeiten

Die Grundidee des Projektes bestand in der Erstellung eines Systems zum Erkennen und Verfolgen von Jonglierbällen in Echtzeit. Als Ziel hinter dieser Idee ist die Möglichkeit gesehen, mit den gewonnenen Informationen verschiedene Anwendungen zu ermöglichen. Das Projekt kann im aktuellen Zustand als Grundlage genommen werden für einfache Programme und Spiele.

9.1 Idee: Ein interaktiver Jongliertrainer

Als denkbare Anwendung auf Basis der von uns erstellten Vorverarbeitung ist ein interaktiver Jongliertrainer zu nennen. Ein solches Programm würde automatisch die Qualität und Güte eines Jongliermusters erkennen und so über die Zeit den Fortschritt der jonglirenden Person aufzeichnen.

Hierzu sind mehrere Grundaufgaben zu lösen, dessen Implementierung wir teilweise bereits in unser Projekt aufgenommen haben.

9.1.1 Objekte zählen

Mit den erkannten momentan fliegenden Objekten haben wir als erste Anwendung einen einfachen Zähler geschrieben, der die tatsächliche Anzahl der im Muster befindlichen Objekte zu bestimmen versucht.

Denke wäre eine Verfolgung der Objekte über einen längeren Zeitraum, um auch unabhängig voneinander erkannte Objekte in ihrer Identität zu bestimmen. Tatsächlich ist ein deutlich simplerer Ansatz aber bereits sehr effektiv, indem über eine längere Anzahl Frames geschaut wird, wie viele Objekte in der Luft sind. Es wird der Durchschnitt dieser Werte gebildet und aufgerundet. Auf diesen Wert wird dann noch 1 addiert, um zur tatsächlichen Anzahl zu gelangen. Dies ist der Tatsache geschuldet, dass beim normalen Jongliermuster zu jeder Zeit mindestens ein Objekt in der Hand ruht, es kann also davon ausgegangen werden, dass maximal $n - 1$ Objekte gleichzeitig in der Luft sind.

Ausprobieren auf echten Daten hat ergeben, dass beim Mitteln über 15 Frames ein zuverlässiges Bestimmen des tatsächlichen Wertes möglich ist.

```
1 class BallCounter(object):
2     """Determine the actual number of objects in the juggling pattern."""
3     def __init__(self):
4         self.count = None
5         self.last = []
6         self.length = 15 # how many frames to analyse
7
8     def update(self, balls):
9         self.last.append(len(balls))
10        if len(self.last) > self.length:
11            self.count = sum(self.last[:-1][:self.length]) / (self.length*1.0)
12            self.count = int(math.ceil(self.count)) + 1
```

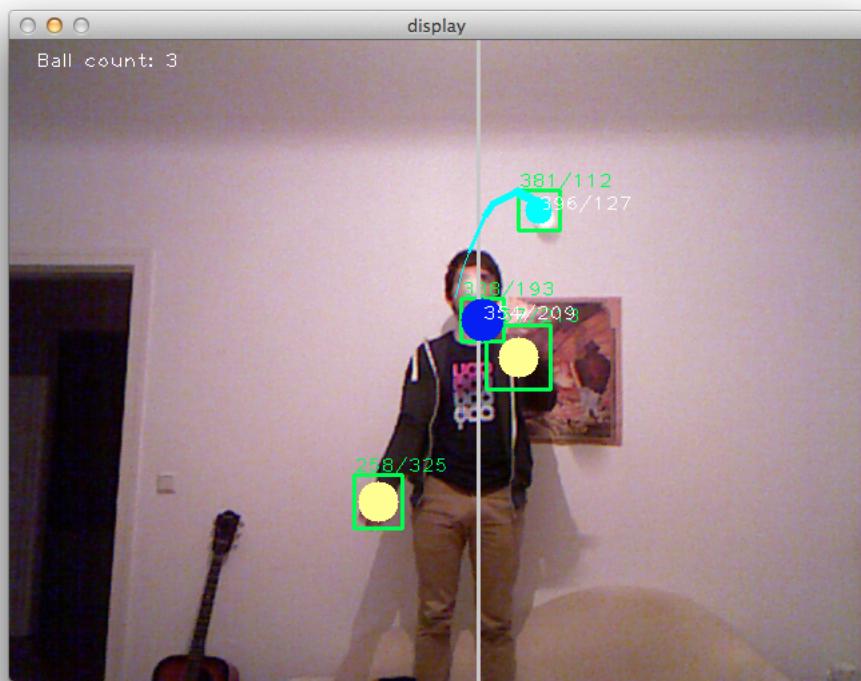


Abbildung 10: FIXME

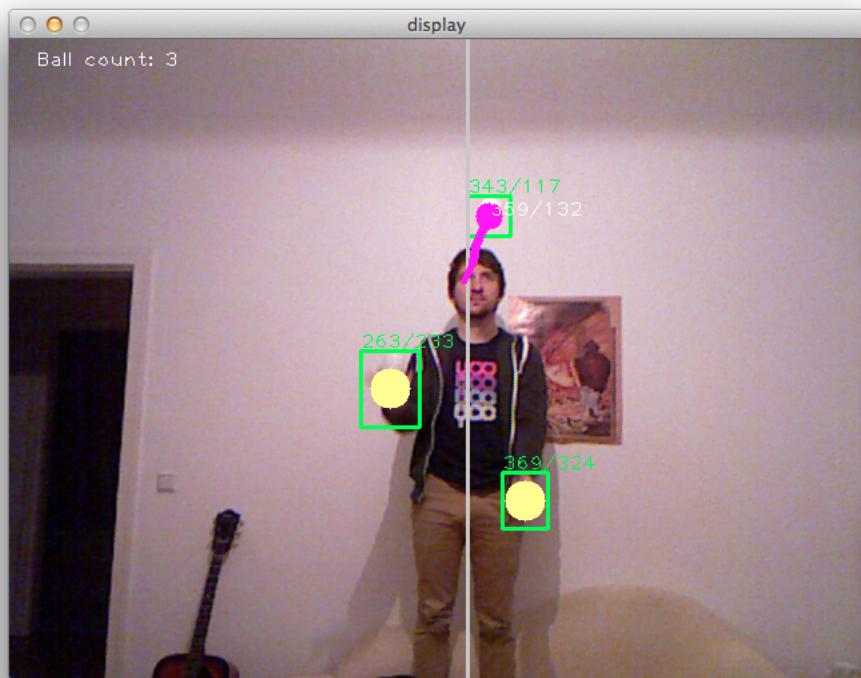


Abbildung 11: FIXME

9.1.2 Würfe zählen

Eine weitere Möglichkeit ist das Zählen von aufeinander folgenden Würfen. Dies ist bereits ein Schritt in die Richtung eines Jongliertrainers, der den Fortschritt eines Jongleurs über eine gewisse Zeit aufzeichnet und misst.

Als Herausforderung bei der Implementierung ist zuerst die Tatsache zu nennen, dass es in den von uns bisher betrachteten Testdaten öfter vorkommt, dass ein Ball in einem Frame nicht erkannt wird, da der Sensor hier keine zuverlässigen Informationen liefert. Im bisherigen Modell wird dadurch das Ballobjekt verworfen und erst im darauf folgenden Frame ein neues initialisiert. Das Zählen von Würfen stimmt also nicht mit dem Zählen von Ball-Initialisierungen überein.

Denkbare Lösungsansätze wären hier, die zwei (oder mehr) Teile einer unterbrochenen Wurfparabel zu erkennen und den Zwischenschritt zu interpolieren, um die gesamte Wurfbahn als einen einzelnen Wurf aufzufassen.

Wird die zeitliche Verteilung der Würfe aufgezeichnet, lässt sich auch analysieren, wie gleichmäßig der Jonglier-Rhythmus ist. Auch dies ist ein wichtiger Faktor

9.1.3 Wurfhöhen messen

Ein gutes Jongliermuster zeichnet sich durch seiner Gleichmäßigkeit aus. Dies trifft sowohl auf

9.2 Idee: Siteswaps erkennen

Bisher haben wir immer über die Grundmuster beim Jonglieren gesprochen, bei denen Bälle im gleichmäßigen Rhythmus auf gleicher Höhe immer die gleichen Wurfbahnen durchlaufen. Tatsächlich existieren komplexere Jongliermuster, die gewissen Regeln folgen müssen, damit in einem gleichmäßigen Rhythmus jongliert werden kann, die Bälle verschiedenen hohen Flugbahnen verfolgen können, und dabei nie zwei Objekte zur gleichen Zeit in der Hand landen.

Eine Notation zur Aufzeichnung solche Muster ist die *Siteswap-Notation*, ein entsprechend jongliertes Muster wird verkürzt als *Siteswap* bezeichnet. Vereinfacht gesagt sind Siteswaps für das Jonglieren so etwas wie Noten für Musiker. Die Notation besteht hierbei grundlegend aus Zahlenreihen, wobei jede Stelle der Reihe einem Wurf entspricht, der Wert an der Stelle bestimmt die Höhe des Wurfes.

Der Siteswap *531* etwa besteht aus drei Würfen. Die *5* entspricht einem Wurf der Höhe von einer regulären 5-Ball-Jonglage, die *3* entsprechend der Höhe einer 3-Ball-Jonglage. Da laut offizieller Regelung (FIXME source) eine Jonglage erst dann als Jonglieren zählt, wenn mehr Objekte als Hände beteiligt sind, haben die Werte Zwei, Eins und Null eine Sonderbedeutung. Die Null im Siteswap bezeichnet eine leere Position zu diesem Takt, die Eins ist ein Übergeben des Objektes von einer in die andere Hand und die Zwei bedeutet ein Festhalten des Balles während einer Zählzeit.

Die konkreten Wurfhöhen sind hierbei nicht von der Zahlenvorschrift festegelegt, sie bezeichnen lediglich relative Höhenabstände zueinander.

Siteswaps sind zudem periodisch zu verstehen, können also fortlaufend jongliert werden. Der oben betrachtete Siteswap 531 ist also gleichbedeutend mit 531531531....

Tatsächlich tragen die Zahlen bereits viel Information, die nicht erst beim Jonglieren sichtbar wird. So ist ein Siteswap immer nur mit einer bestimmten Objektanzahl möglich, die dem Durchschnitt der einzelnen Wurfwerte entspricht. Teilt man also die Quersumme von 531 (= 9) durch die Anzahl der Würfe (3), sieht man, dass dies tatsächlich ein 3-Ball-Muster ist. Hierbei sind nur ganzzahlige Ergebnisse möglich. 541 etwa wäre kein valider Siteswap und ist nicht jonglierbar.

Wie an Hand dieser kurzen Einführung bereits deutlich wird, bieten Siteswaps sich gut zur automatischen Analyse an. Möglich und denkbar ist also eine Anwendung, die an Hand der von unserem System gelieferten Daten automatisch Siteswaps erkennt.

Hierbei wären folgende grundlegenden Schritte notwendig.

1. Würfe mit ihren absoluten Wurfhöhen aufzeichnen
2. Würfe nach Abwurfzeit sortieren
3. Relative Wurfhöhen bestimmen
4. Bei möglichen Mehrdeutigkeiten das Wissen aus der Siteswap Theorie nutzen

Mit Hilfe dieser Erkennung ist es denkbar, dem Akteur vor der Kinect Aufgaben an Hand von Jongliermustern zu geben, die erfolgreich jongliert werden müssen. Ein Muster kann animiert vorjongliert werden, der Akteur muss hierbei nicht einmal mit der Zahlentheorie konfrontiert werden. So ist ein einfaches Level-basiertes System denkbar, das einen Jonglieranfänger vom Grundmuster zu komplexeren Mustern und sogar mehr als drei Jonglierobjekten führt.

10 Stand der Forschung

Was Flo mit Benjamin besprochen hatte, noch ein paar aktuelle Forschungsansätzen zu Kinect u.ä. vorstellen, die mit unserem Thema zu tun haben

Aber vielleicht eher direkt in den Abschnitten, wo wir über die Ansätze sprechen, die wir gewählt haben?

11 Fazit

Im Laufe des Projektes haben wir uns intensiv mit der Echtzeitverarbeitung von Tiefendaten auseinander gesetzt und sind über viele nicht funktionierende Ansätze zu einem Ergebnis gekommen, das robust und mit flexibler Anzahl von Objekten das Muster eines Jongleurs erfassst. Unsere für die erfolgreiche Implementierung getroffenen Annahmen

über den Aufbau der Kinect und die Position des Jongleurs im Raum sind nicht all zu einschränkend, so dass einer tatsächlichen Anwendung der entwickelten Software nichts im Wege steht.

Die Software kann als Grundlage verwendet werden für weitere Anwendungen mit der Kinect und Jonglierbewegungen. Dies haben wir beispielhaft am Anwendungsfall des Zählens von Jonglierbällen gezeigt und weitere mögliche Entwicklungen vorgeschlagen und die nötigen Schritt umrissen.

In unserer Arbeitsweise ist uns deutlich geworden, dass die Arbeit an praktischen Projekten einen großen Lerneffekt hat und auch die zahlreichen Fehlversuche mit wiederkehrenden frustrierenden Ergebnissen dazugehören, um schließlich zu einer funktionierenden und robusten Implementierung zu gelangen. Um bei ähnlichen Projekten in Zukunft schneller zu einem Ergebnis zu kommen, wird es aber nötig sein, dass wir uns von Anfang an selbst klarere Ziele stecken und den Fortschritt besser zu verfolgen um ihn auch sicher stellen zu können.

Als Endergebnis des Projektes gelangen wir erfolgreich zu einer quelloffenen und flexibel erweiterbaren Software, die robust unsere selbst gestellte Aufgabe löst. Zudem haben bei der Arbeit mit den verwendeten Libraries viel gelernt und dem `libfreenect` in der Open Source Community bei der Verbesserung der Code Basis helfen können.

Quellen

- [1] **FIXME** Bibliothek evtl. etwas sortieren oder so
- [2] Jeff Kramer, Nicolas Burrus, Florian Echtler, Daniel Herrera C., Matt Parker
Hacking the Kinect
ISBN 978-1-4302-3867-6
Springer Science+Business Media New York, 2012.
- [3] Jens Kober^{1,2}, Matthew Glisson², and Michael Mistry^{2,3},
Playing Catch and Juggling with a Humanoid Robot.
² Disney Research Pittsburgh, USA
¹ Bielefeld University, Germany,
³ University of Birmingham, UK.
- [4] Jens Kober^{1,2}, Matthew Glisson², and Michael Mistry^{2,3},
Disney Research Project Web Page
Playing Catch and Juggling with a Humanoid Robot.
² Disney Research Pittsburgh, USA
¹ Bielefeld University, Germany,
³ University of Birmingham, UK
http://www.disneyresearch.com/project/juggling_robot (accessed on 2014-06-22).
- [5] *OpenKinect project*
http://openkinect.org/wiki/Main_Page (accessed on 2014-06-12).
- [6] *libfreenect GitHub repository*
<https://github.com/OpenKinect/libfreenect> (accessed on 2014-06-12).
- [7] Robert T. Held¹, Ankit Gupta², Brian Curless², Maneesh Agrawala¹
3D Puppetry: A Kinect-based Interface for 3D Animation
¹ University of California, Berkeley,
² University of Washington,
Cambridge, Massachusetts, USA, 2012.
- [8] Yan Cui¹, Will Chang, Tobias Nöll¹, Didier Stricker¹
KinectAvatar: Fully Automatic Body Capture Using a Single Kinect
¹ Augmented Vision, DFKI,
German Research Center for Artificial Intelligence.
- [9] Christian Richardt^{1,2}, Carsten Stoll¹, Neil A. Dodgson², Hans-Peter Seidel¹, Christian Theobalt¹
Coherent Spatiotemporal Filtering, Upsampling and Rendering of RGBZ Videos
¹ MPI Informatik,
¹ University of Cambridge,
EUROGRAPHICS Volume 31, Number 2, 2012.

- [10] Ullrich Köthe und viele weitere Mitwirkende
The VIGRA Computer Vision Library Version 1.10.0
<http://ukoethe.github.io/vigra/> (accessed on 2014-06-24).
- [11] Richard O. Duda, Peter E. Hart,
Use of the Hough Transformation to detect lines and curves in pictures
 Technical Note 36, Artificial Intelligence Center, April 1971,
 Published in the Comm. ACM, Vol 15, No. 1 (11-15), January 1972.
<http://www.ai.sri.com/pubs/files/tn036-duda71.pdf> (accessed on 2014-06-25).
- [12] OpenCV (Open Source Computer Vision)
<http://opencv.org/> (accessed on 2014-06-25).
- [13] Rudolf E. Kálmán
A New Approach to Linear Filtering and Prediction Problems
 Transaction of the ASME, Journal of Basic Engineering, 35-45,
 Research Institute for Advanced Study, Baltimore, Md., 1960.
<http://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf> (accessed on 2014-06-26).
- [14] Federal Aviation Administration
 Frequently Asked Questions - WAAS
http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/faq/waas/ (accessed on 2014-06-26).
- [15] European Space Agency
 What is EGNOS? http://www.esa.int/Our_Activities/Navigation/The_present_-_EGNOS/What_is_EGNOS (accessed on 2014-06-26).
- [16] Numpy
<http://www.numpy.org/> (accessed on 2014-06-28).
- [17] Scipy
<http://www.scipy.org/> (accessed on 2014-06-28).