Bachelorarbeit am Institut für Informatik der Freien Universität Berlin, Arbeitsgruppe

Intelligente Systeme und Robotik

# New Eyes for Grace

Erik Sporns
erik@derwikinger.de
Matrikelnummer: 4753221

Gutachter:
Prof. Dr. Raúl Rojas
Prof. Dr. Daniel Göhring
Betreuer: Lutz Freitag

Berlin, 28. Juni 2016

**Zusammenfassung**

Auf Grund von Regeländerungen im Roboterfußball ist eine leistungsfähigere Bildverarbeitung notwendiger denn je. In dieser Arbeit wird untersucht, inwiefern eine stereooptische Kamera die Fähigkeit des Roboters bestimmte Objekte zu erkennen verbessert. Dazu wird ein Algorithmus zur Extraktion von Ebenen aus Punktwolken vorgestellt und getestet. Da es sich bei dem Roboter um eine mobile Plattform handelt, gibt es Besonderheiten, die einem effizienten Einsatz entgegenstehen.

## Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

28. Juni 2016

Erik Sporns

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

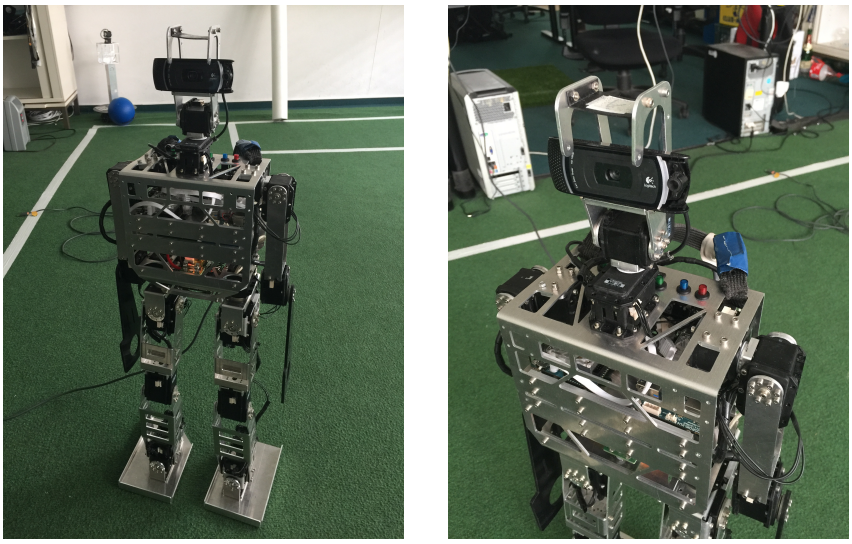
Humanoide Roboter sind Maschinen, die ihre Umwelt wahrnehmen und mit ihr interagieren können. Dabei sind sie dem menschlichen Körperbau nachempfunden: Ihr Körper gliedert sich in einen Rumpf mit Kopf, zwei Beinen und Armen. Im Gegensatz zum Menschen kann ein Roboter allerdings auf eine größere Auswahl an Sensoren zurückgreifen, um Informationen über seine Umwelt zu erhalten. Eine wichtige Rolle spielen dabei Kameras. Sie bieten den Vorteil sehr schnell und berührungslos viele Details aufnehmen zu können. Gerade im kompetitiven Umfeld des Roboterfußballs ist es ein essentieller Vorteil eine dem Gegner überlegene Bildverarbeitung zu besitzen, die sich an die immer wieder ändernden Herausforderungen anpassen kann. Dabei verbieten die Regeln den Einsatz aktiver Sensoren wie z.B. die Microsoft Kinect Tiefenkamera, LIDAR und Ultraschall, so dass für das Sammeln von Tiefeninformationen passive Sensoren, wie z.b. eine Stereokamera, eingesetzt werden müssen.

## 1.1 Motivation

Speziell die Einführung von Bällen mit hohem Weißanteil [8] im Roboterfußball haben die Bildverarbeitung auf den Robotern vor neue Herausforderungen gestellt. Hauptsächlich sind nun relevante Objekte (Tore, Bälle) nicht mehr eindeutig farbkodiert und müssen auf andere Arten identifiziert werden. Ein vielversprechender Ansatz ist der Einsatz einer Stereokamera, um die Distanz zu einem Objekt als zusätzliches Kriterium zur Identifikation zu benutzen. Weiterhin gewinnt der Roboter die Fähigkeit Distanzen bedeutend präziser abzuschätzen, was die Orientierung auf dem Feld einfacher gestaltet. Allerdings bringt die Technologie auch Probleme mit sich: Erhöhter Rechen-/Strombedarf und eine höhere Empfindlichkeit der Kameras gegenüber äußerlichen Einflüssen. Im Rahmen dieser Arbeit wird untersucht, inwiefern die Viskos-Stereokamera für den Einsatz auf einem FUmanoid Roboter geeignet ist.

## 1.2 Die Roboter

Der erste Vertreter der neusten Iteration der FUmanoid Roboter, Curious Capacitor, ist eine komplette Neuentwicklung, die den evolutionär nächsten Schritt zur Vorgängergeneration darstellt. In das Design sind alle Erfahrungen, die mit den älteren Generationen gesammelt wurden, eingeflossen. Der Roboter ist mit 75cm deutlich größer als seine Vorgänger (60cm) und verzichtet zugunsten größerer Beweglichkeit auf die Parallelkinematik. Die Rechenleistung wurde von einem Odroid xu2 auf zwei Odroid xu4[5], die via Gigabitnetzwerk miteinander verbunden sind, erhöht. Um den erhöhten Stromverbrauch decken zu können wird eine neue Stromplatine eingesetzt, die auf der 5V Domäne bis zu 10A abgeben kann.



(a) Curious Capacitor (b) Kopf mit Kamera

Abbildung 1.1: Die neuste Generation der FUmanoiden

### 1.2.1 Notwendige Änderungen

Bisher sind die Roboter mit einer konventionellen USB Webcam ausgestattet, die mit Hilfe von zwei Servomotoren einen in zwei Achsen beweglichen Kopf bildet (vgl. Abb. 1.1b). Damit die neue Kamera auf dem Roboter befestigt werden kann, wurde ein neuer Kameraträger entworfen (vgl. Abb. 1.2). Dieser hat gegenüber dem originalen Kameraträger den Vorteil, dass er etwas leichter sowie der Abstand zwischen den Kameras größer ist und die notwendigen Befestigungslöcher für den Roboter optimal platziert werden konnten. Softwareseitig wurde ein C++ Framework entwickelt, das die Kamera dem bestehenden Projekt zur Verfügung stellt und den gesamten Netzwerkverkehr zur Kamera regelt.

Abbildung 1.2: Neu entworfener Kameraträger

## 1.3 Viskos Kamera

Die Viskos-Kamera wurde von Bennet Fischer im Rahmen seines Promotionsverfahrens entwickelt und gebaut. In der verwendeten Konfiguration besteht die Kamera aus zwei einzelnen Farbkameramodulen mit jeweils 752x480 Pixel Auflösung, die in einem Kameraträger aus Aluminium befestigt sind. In dem Kameraträger werden identische Objektive mit 3,5mm Brennweite geschraubt (Basislinie 0,073m). Das Mainboard der Kamera ist mit einem Xilinx Zynq bestückt, der einen Arm Cortex A9 mit einem FPGA vereint, und wird auf ein Basisboard gesteckt, das die Stromversorgung, den Speicher für das Betriebssystem (microSD Karte) und die Schnittstellen zur Außenwelt (USB, Netzwerk) bereitstellt. Softwareseitig läuft auf dem Mainboard ein Linaro Linux, das die Grundlage für das OROCOS[7] Framework darstellt, welches die notwendigen Rechnungen ausführt.

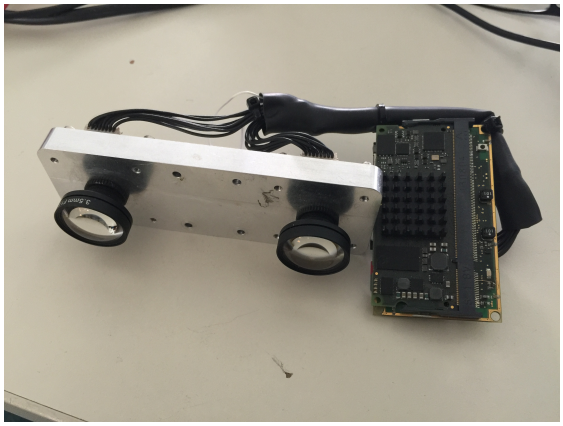


Abbildung 1.3: Viskos Stereokamera

### 1.3.1 Netzwerkprotokoll

Ursprünglich kommuniziert die Kamera über OROCOS und ORB mit ihrer Außenwelt. Zu Debuggingzwecken wurde in der Vergangenheit ein einfaches UDP Protokoll imple-

mentiert, das den Bildversand über Netzwerk realisiert. Dabei wird jedes einzelne Teilbild an einen anderen Port (linkes Teilbild: 2000, rechtes Teilbild: 2001, Disparitätenbild: 2002) mit einer Geschwindigkeit von 20 Frames pro Sekunde versendet. Um eine Verbindung mit der Kamera herzustellen wurde eine C++ Klasse entwickelt, die die notwendige Schnittstelle zu den Low-Level-Aufgaben der Kommunikation realisiert.

```
StereoCam camera(2000, 2001, 2002);
camera.openDevice();
camera.calibrate_viskos_cam(10, 11, 7, false);
camera.capture(&img1, &img2);
camera.disp_capture(&img3);
imshow("links", img1);
imshow("disp", img3);
```

Abbildung 1.4: Benutzung des Frameworks

Der Codeauszug in Abb. 1.4 initialisiert die Verbindung zur Kamera, kalibriert diese und empfängt die beiden Einzelbilder und das Disparitätenbild. Das Protokoll erlaubt es nicht direkt in die Kamera einzugreifen, sondern erlaubt nur den Bildempfang. Um Änderungen an Kameraparametern vornehmen zu können, muss eine SSH-Verbindung zur Kamera hergestellt werden. Die Parameter können dann anhand einer XML-Datei angepasst werden.

Da für die Kalibrierung bereits ein Zeitstempel zu den Bildmetainformationen hinzugefügt wurde, konnte mit geringem Aufwand eine Funktion zum lokalen Berechnen des Disparitätenbildes (mit Hilfe von openCV[6]) umgesetzt werden. Die Berechnung außerhalb der Kamera (auf einem vollwertigen Computer) hat den Vorteil, dass man sehr flexibel in die Generierung des Disparitätenbildes und der daraus abgeleiteten Punktwolke eingreifen kann.

## 1.4 Microsoft Kinect

Der in dieser Arbeit vorgestellte Algorithmus wird nach seiner Implementierung experimentell auf Korrektheit untersucht. Für die ersten Tests wurden sehr saubere Punktwolken benötigt, die möglichst wenig Bildrauschen aufwiesen, da Versuche auf künstlichen Daten vermieden werden sollten. Um diese Punktwolken aufzunehmen, wurde die Kinect Tiefenkamera von Microsoft benutzt. Der Sensor liefert mit einer Präzision von 1,5mm bis 5cm Tiefendaten im Bereich von 0,5mm bis 4m, für die gilt: 1,5mm Rauschen bei 0,5m bis 5cm Rauschen bei 4m. Mit dem Freenect-Projekt [11] als Grundlage konnte die Kinect problemlos in den Code eingebunden werden.

# Kapitel 2

# Theoretische Grundlagen

## 2.1 Stereovision

Das in der Natur sehr erfolgreiche Prinzip, zwei Bildquellen zur Tiefenwahrnehmung zu benutzen, lässt sich mathematisch exakt fassen. Analog zur Natur wird in stereooptischen Kamerasystemen ein und dieselbe Szene von zwei Kameras erfasst. Diese haben die gleiche Brennweite $f$ (in dieser Arbeit in Pixel angegeben) und befinden sich auf einer gemeinsamen Achse in einem Referenzkoordinatensystem:

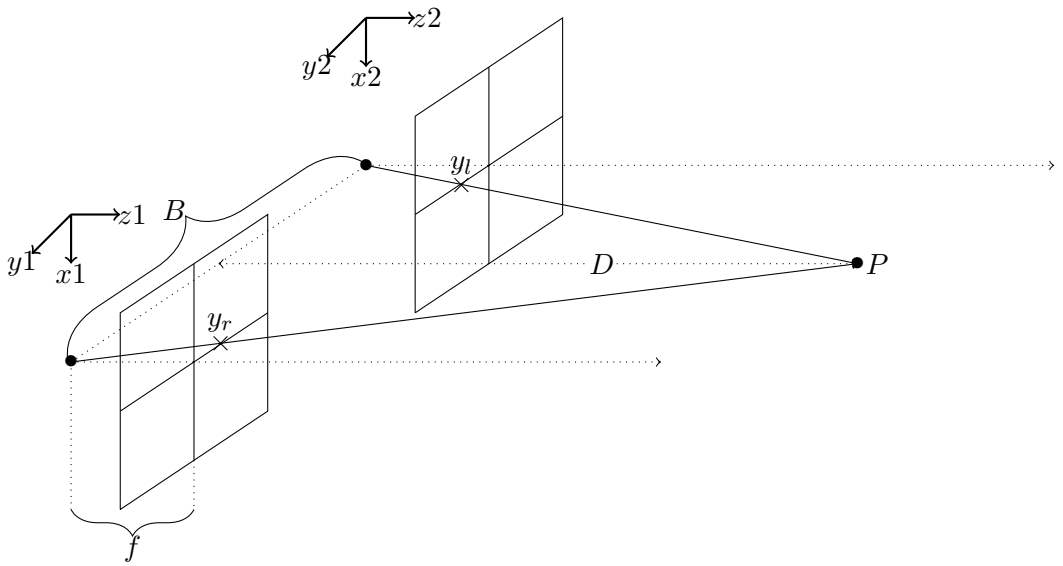



Abbildung 2.1: Stereokamerasystem in kanonischer Konfiguration (vgl. [13, S. 6])

- Die Kameras sind parallel auf einer gemeinsamen Achse (Basislinie) angeordnet. Der Abstand sei B.
- Die Bildachsen der Kameras stehen orthogonal auf der Basislinie.

- Die Basislinie liegt auf der y-Achse des Systemkoordinatensystems.

Der Punkt P mit den Koordinaten $(x, y, z)$ im Systemkoordinatensystem wird auf den Punkt $x_l, y_l$ auf der linken bzw. $x_r, y_r$ auf der rechten Bildebene der Kameras abgebildet. Aus dem S:S:S-Satz für Dreiecke „Zwei Dreiecke sind zueinander ähnlich, wenn sie in allen Verhältnissen entsprechender Seiten übereinstimmen.“[10, S. 144] folgt:

$$\frac{y_l}{f} = \frac{y + \frac{B}{2}}{z} \tag{I}$$

$$\frac{y_r}{f} = \frac{y - \frac{B}{2}}{z} \tag{II}$$

$$\frac{x_l}{f} = \frac{x_r}{f} = \frac{x}{z} \tag{III}$$

Um die Koordinaten von P zu errechnen stellen wird die Gleichungen nach $x, y, z$ um:

- z ergibt sich, indem man (II) von (I) subtrahiert:

$$\begin{aligned} \frac{y_l - y_r}{f} &= \frac{y + \frac{B}{2} - y + \frac{B}{2}}{z} && \mid \text{II - I} \\ \Leftrightarrow \frac{y_l - y_r}{f} &= \frac{B}{z} \\ \Leftrightarrow z &= \frac{Bf}{y_l - y_r} \end{aligned}$$

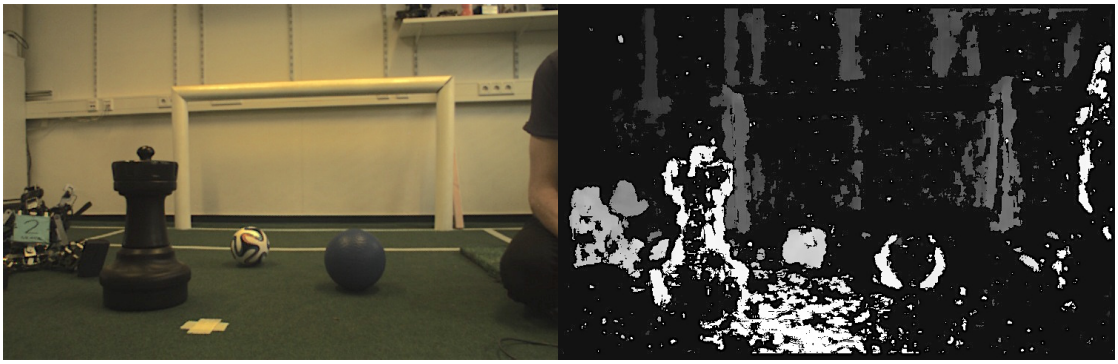
- Da (III) gilt, lässt sich x wie folgt errechnen:

$$\begin{aligned} \frac{x_l}{f} &= \frac{x}{z} && \text{I} \\ \frac{x_r}{f} &= \frac{x}{z} && \text{II} \\ \frac{2x}{z} &= \frac{x_l + x_r}{2f} && \mid \text{I + II} \\ 2x &= \frac{(x_l + x_r)z}{2f} && \mid \text{nach x umgestellt} \\ x &= \frac{(x_l + x_r)Bf}{2f(y_l - y_r)} && \mid \text{z eingesetzt} \\ x &= \frac{B(x_l + x_r)}{2(y_l - y_r)} \end{aligned}$$

- Für y gilt:

$$
\begin{aligned}
\frac{y_l + y_r}{f} &= \frac{y + \frac{B}{2} + y - \frac{B}{2}}{z} && \mid \text{I} + \text{II} \\
\Leftrightarrow \frac{y_l + y_r}{f} &= \frac{2y}{z} \\
\Leftrightarrow y &= \frac{y_l + y_r}{2f} z \\
\Leftrightarrow y &= \frac{(y_l + y_r)Bf}{2(y_l - y_r)f} && \mid \text{z einsetzen} \\
\Leftrightarrow y &= \frac{B(y_l + y_r)}{2(y_l - y_r)}
\end{aligned}
$$

Dabei wird der wiederkehrende Term $y_l - y_r =: d$ als Disparität (in dieser Arbeit in Pixel angegeben) definiert. Die Einheiten für $x_c, y_c, z_c$ ergeben sich aus der gewählten Einheit für B. Wählt man reale Distanzmaße, wie z.B. Meter, kann man $z$ als Abstand des Punktes $(x, y)$ zur $(x, y)$-Ebene des Kamerakoordinatensystems betrachten. Daraus folgt direkt, dass sich die Disparität reziprok zur Distanz eines Punktes verhält. Dies hat in der Realität die Konsequenz, dass die Tiefenauflösung einer Stereokamera durch ihre Bildauflösung und ihren Basisabstand beschränkt ist. Disparitäten mit Subpixelgenauigkeit sind möglich, aber nicht immer genau. Berechnet man die Disparität für jedes korrespondierende Pixelpaar erhält man ein Disparitätenbild (Abb. 2.2b). Dabei sind helle Flächen Bereiche mit einer großen Disparität (nahe Bereiche) und dunklere Zonen stehen für kleinere Disparitäten (größere Entfernung zur Kamera).



(a) Bild der linken Kamera (b) Disparitätenbild

Abbildung 2.2: Beispielbilder, aufgenommen mit der Viskos Kamera

Um die Disparitäten bestimmen zu können, müssen Pixel der linken Kamera dem entsprechenden Pixel der rechten Kamera zugeordnet werden. Dafür müssen Bildpunkte einzigartig sein. Folglich können in homogenen Bildregionen keine Disparitäten bestimmt werden, da eine eindeutige Zuordnung zweier Pixel nicht möglich ist.

## 2.2 Bildsegmentierung

Um die Daten aus dem Disparitätenbild weiter zu verarbeiten, ist die Berechnung einer Punktwolke eine geeignete Lösung. Bisher gibt es zu jedem Pixel $(x, y)$ des Bildes lediglich einen Disparitätswert, aus dem man die Distanz (die $z$-Weltkoordinate) des Punktes zur Kamera bestimmen kann. Die $x, y$ Koordinaten sind im Weltkoordinatensystem noch nicht vorhanden. Die fehlenden Koordinaten lassen sich mit den im vorherigen Kapitel vorgestellten Gleichungen für jeden Pixel im Disparitätenbild bestimmen und man erhält eine Punktwolke. Um diese Punktwolke, die sehr viele Daten enthält, algorithmisch besser erfassen zu können, bietet es sich an die Datenmenge zu reduzieren, indem man charakteristische Merkmale extrahiert. Solche Merkmale können beispielsweise Ebenen sein. Im Folgenden soll ein solcher Algorithmus zur Ebenenextraktion aus Punktwolken vorgestellt werden. Dabei gliedert sich der Algorithmus in drei Schritte: Segmentierung, Optimierung und Merkmalsextraktion. Um die Effizienz zu erhöhen, werden Integralbilder[2, S. 2685], auf die später näher eingegangen wird, eingesetzt.

### 2.2.1 Andere Arbeiten

Ein allgemeiner Ansatz zum Berechnen von Oberflächennormalenvektoren wird in [2] vorgestellt. Analog wird der vorgestellte Algorithmus durch den Einsatz von Integralbildern beschleunigt, um ausreichend hohe Verarbeitungsgeschwindigkeiten für den Einsatz auf Roboterplattformen mit eingeschränkter Rechenleistung zu erreichen.

In [3] werden gebräuchlichen Methoden zur Normalenvektorextraktion von Oberflächen verglichen. Dabei wird zwischen Optimierungsproblemen und Mittelungsverfahren unterschieden.

In [12] wird ein Algorithmus zum Zerlegen allgemeiner Punktwolken (z.B. von einem LIDAR) vorgestellt. Die Punktwolke wird mit einem Oktärbaum in Unterbereiche unterteilt, die mit einer Ebene approximiert werden (je nach Bedarf werden Knoten weiter unterteilt). Der Algorithmus verzichtet dabei auf den Gebrauch einer Sortierung der Daten (z.b. durch Perspektive oder Postion im Speicher). Dadurch wird der Algorithmus zwar um die Fähigkeit erweitert, Rückseiten von Objekten zerlegen zu können, während die Laufzeit allerdings stark erhöht wird.

### 2.2.2 Datenstrukturen

Aus dem von der Kamera generierten Disparitätenbild wird ein Punktwolkenbild errechnet. Für jeden Pixel $(x_b, y_b)$ aus dem Disparitätenbild werden die entsprechenden Koordinaten $(x_c, y_c, z_c)$ im Systemkoordinatensystem berechnet. Diese Darstellungsform erhält zusätzliche Informationen: Punkte im Bildkoordinatensystem sind benachbart, obwohl sie in der Punktwolke keine Nachbarschaftsbeziehung haben müssen. Diese perspektivische Nachbarschaft stellt eine Form von Sortierung dar, die es ermöglicht auf den Daten effizient zu suchen. Sie stellt damit den maßgeblichen Vorteil dar, den der

Algorithmus gegenüber Lösungen zum Zerlegen allgemeiner Punktwolken, die nicht von Stereo Kameras aufgenommen wurden, hat.

### 2.2.3 Bildsegmentierung

Der erste Schritt der Bildsegmentierung ist das Zerlegen der berechneten Punktwolke in einzelne Teilebenen. Der Ansatz hierfür folgt einem einfachen Schema: Für einen gewählten Punktwolkenbildausschnitt wird eine Ebene geschätzt und der Fehler der Ebenenschätzung bestimmt. Sollte der Fehler einen Grenzwert überschreiten, wird der Bereich in zwei Unterbereiche unterteilt und die Methode rekursiv solange wiederholt, bis der Grenzwert unterschritten wird und die Ebene als valide Lösung für den Bildausschnitt akzeptiert wird.

**Algorithm 1** Fitts planes into a point cloud image

```
procedure SEPERATEIMAGE(input, region)
    plane = generatePlane(input, region)
    error = calculateError(input, plane)
    if error > alpha && region.area > minarea then
        regions = seperatePlane(input, region)
        seperateImage(input, regions[0])
        seperateImage(input, regions[1])
    else
        validPlanes.add(plane)
    end if
end procedure
```

Das Verhalten ist stark von der gewählten Fehlerfunktion und der Art, in der das Bild unterteilt wird, abhängig. Dabei wird sichergestellt, dass eine Mindestfläche des Bildes benutzt wird, um eine gewisse Mindestpunktmenge für eine Ebene zu sichern.

### 2.2.4 Integralbilder

An sehr vielen Stellen des Algorithmus werden Durchschnitte über Areale gebildet. Um diese Schritte zu beschleunigen, bietet sich die Verwendung von Integralbildern an, da man mit nur wenigen Speicherzugriffen die Summe über einen Bildausschnitt bestimmen kann. Im Integralbild ist jeder Pixel $I$ wie folgt definiert:

$$I(x, y) = \sum_{j=0}^{n} i(x_j, y_j)$$

Verwendet man dynamische Programmierung, kann man auf bereits berechnete Werte zurückgreifen. Für den Pixel an der Position $I(x, y)$ im Integralbild gilt nun:

$$I(x, y) = i(x, y) + I(x, y-1) + I(x-1, y) - I(x-1, y-1)$$

wobei $i(x, y)$ den Pixel an der Position $(x, y)$ im Ausgangsbild bezeichnet. Die benötigte Rechenzeit reduziert sich somit für $n$ berechnete Pixel von $n^3$ auf $n^2$. Die Laufzeit lässt sich durch den Einsatz aufwändigerer Algorithmen [1] noch weiter reduzieren. Die Summe über eine Fläche $F$ mit den Eckpunkten $A, B, C, D$ wird mit vier Speicherzugriffen folgendermaßen berechnet:

$$I(F) = I(D) + I(A) - I(B) - I(C)$$

Da die Punktwolken der Kamera nicht dicht sein müssen, also nicht jeder Bildkoordinate $(x_b, y_b)$ ein Punkt im Systemkoordinatensystem zugewiesen werden kann, weil Disparitätenbilder fehlerhaft sein können, werden von dem Algorithmus zwei Integralbilder berechnet. Dabei hat das erste drei Kanäle (jeweils einen für die Koordinaten $(x_c, y_c, z_y)$ jedes Pixels des Punktwolkenbildes) und stellt das Integralbild über dem Punktwolkenbild dar. Im zweiten Bild wird das Integral über alle validen Punkte berechnet. Damit ein Punkt als valide gilt, muss seine $z$-Koordinate im Wertebereich des Bildsensors liegen (auf jeden Fall aber größer sein als 0).
Der Mittelwert $M(F)$ über einen Ausschnitt $F$ kann nun direkt bestimmt werden:

$$M(F) = \frac{I_{Bild}(F)}{I_{valid}(F)}$$

Es muss beachtet werden, dass nur achsenparallele Schnitte beschleunigt werden können, da nur Werte für rechteckige Areale (im Folgenden Boundingboxen genannt) ausgerechnet werden können.

### 2.2.5 Ebenen erzeugen

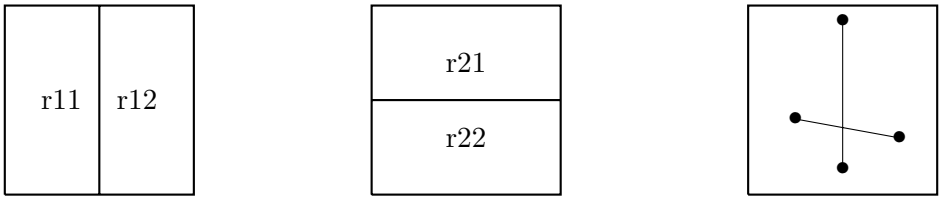
Als Ebenendarstellung wurde die allgemeine Ebenengleichung verwendet:

$$ax + by + cz + d = 0$$

Diese Darstellungsform wurde gewählt, weil sie sehr leicht zu bestimmen ist und die Distanzbestimmung von Punkten zur Ebene direkt ermöglicht. Dabei entsprechen die Koeffizienten $a, b, c$ dem Normalenvektor $\vec{n}$ der Ebene und dem Wert $d$, der den Abstand der Ebene zum Koordinatenursprung angibt. Die Metrik, die dadurch erzeugt wird, entspricht der Norm des Normalenvektors. Ist der Normalenvektor normiert, entsteht die euklidische Metrik. Ist dieser Vektor normalisiert kann die Distanz D eines Punktes $P = (x, y, z)$ direkt bestimmt werden:

$$D = ax + by + cz + d$$

Im Folgenden wird diese Ebenendarstellung als Ebenendeskriptor bezeichnet. Um eine Ebene aufzuspannen, wird der gewählte Bildausschnitt zunächst in vier sich überlappende Regionen aufgeteilt (Abb. 2.3a und 2.3b).




(a) Vertikale Quadranten (b) Horizontale Quadranten (c) Stützvektoren

Abbildung 2.3: Bildaufteilung

Für jede Hälfte $r_{11}, r_{12}, r_{21}, r_{22}$ wird mit Hilfe der Integralbilder der entsprechende Mittelpunkt errechnet. Zwischen den jeweils gegenüberliegenden Punkten $(r_{11}, r_{12})$ und $(r_{21}, r_{22})$ werden Hilfsvektoren aufgespannt, deren Kreuzprodukt den Vektor $\vec{n}$ gibt, der orthogonal auf der Ebene, die aufgespannt werden soll, steht (Abb. 2.3c). Ist dieser Vektor normalisiert wird eine einheitliche Metrik erzeugt (die einheitliche euklidische Metrik), wodurch die Distanzen zwischen Ebenen vergleichbar werden. Um den Koeffizienten $d$ für die Ebenengleichung zu bestimmen, wird nun zunächst der Stützpunkt $\vec{k}$ der Ebene bestimmt, indem der Mittelwert über den gesamten Ausgangsbildausschnitt berechnet wird. Mit Hilfe von $\vec{k}$ und $\vec{n}$ kann $d$ nun bestimmt werden:

$$d = -\vec{k}^t \vec{n}$$

**Algorithm 2** Calculates the plane for a given boundingbox

1: **procedure** CALCULATEPLANE($input, bb$)
2: $r11 = (bb.x_{min}, bb.y_{min}, (bb.x_{min} + bb.x_{max})/2, bb.y_{max})$;
3: $r12 = ((bb.x_{min} + bb.x_{max})/2, bb.y_{min}, bb.x_{max}, bb.y_{max})$;
4: $r21 = (bb.x_{min}, bb.y_{min}, bb.x_{max}, (bb.y_{min} + bb.y_{max})/2)$;
5: $r22 = (bb.x_{min}, (bb.y_{min} + bb.y_{max})/2, bb.x_{max}, bb.y_{max})$;
6:
7: $p_{11} = getCentralPoint(r11)$;
8: $p_{12} = getCentralPoint(r12)$;
9: $p_{21} = getCentralPoint(r21)$;
10: $p_{22} = getCentralPoint(r22)$;
11:
12: $L_x = p_{12}[0] - p_{11}[0], p_{12}[1] - p_{11}[1], p_{12}[2] - p_{11}[2]$;
13: $L_y = p_{22}[0] - p_{21}[0], p_{22}[1] - p_{21}[1], p_{22}[2] - p_{21}[2]$;
14:
15: $n = cross\_product(L_x, L_y)$;
16: $normalize(n)$;
17: $center = getCentralPoint(bb)$;
18: $d = -(center[0] * n[0] + center[1] * n[1] + center[2] * n[2])$;
19:
20: **return** Plane(n ,d, center, bb)
21: **end procedure**

### 2.2.6 Ebenen aufspalten

Ist die Qualität (siehe 2.2.7) der Ebenenschätzung unter einem Mindestmaß, muss diese Ebene in Teilebenen aufgespalten werden. In diesem Algorithmus wird in jedem Teilschritt eine Zerteilung der Ebene in zwei Teilebenen umgesetzt. Die Schnittrichtung wird dabei in jedem Fall individuell bestimmt, wobei nur achsenparallele (entweder entlang der x- oder y-Achse des Bildes) Schnitte möglich sind. Um zunächst die Schnittrichtung zu bestimmen, wird der Fehler entlang zweier Linien bestimmt, die jeweils parallel zu einer der Koordinatenachsen durch die Mitte des betrachteten Ausschnittes verlaufen (siehe Algorithm 3). Die Schnittrichtung verläuft parallel zu der Linie, die den größeren Fehlerwert hat.

**Algorithm 3** Seperates a plane

1: **procedure** SEPERATEPLANE($input, bb$)
2: $rootPlane = generatePlane(input, bb)$;
3: $x\_error = directionalMeanSqError(rootPlane, horizontal)$;
4: $y\_error = directionalMeanSqError(rootPlane, vertical)$;
5: **if** $x\_error < y\_error$ **then**
6: xLine = findXLine(input,bb);
7: bb1 = BoundingBox($bb.x_{min}, xLine, bb.bb.y_{min}, bb.y_{max}$);
8: bb2 = BoundingBox($xLine, bb.x_{max}, bb.bb.y_{min}, bb.y_{max}$);
9: **else**
10: yLine = findYLine(input,bb);
11: bb1 = BoundingBox($bb.x_{min}, bb.x_{max}, bb.bb.y_{min}, yLine$);
12: bb2 = BoundingBox($bb.x_{min}, bb.x_{max}, yLine, bb.y_{max}$);
13: **end if**
14: **return** bb1,bb2;
15: **end procedure**

Um die Position der Schnittgerade zu finden, wird die Sortierung, die sich durch die perspektivischen Nachbarschaftsinformationen ergibt, die im Punktwolkenbild erhalten bleiben, ausgenutzt und eine Binärsuche mit einem 1-Schritt Lookahead eingesetzt. Der Algorithmus für den Schnitt entlang der y-Achse ist in Algorithm 4 beschrieben und funktioniert analog für die x-Achse.

**Algorithm 4** Find a seperation line along the y-axis

```
1:  procedure FINDXLINE(input, bb)
2:      xLine = (bb.x_min + bb.x_max + 1)/2;
3:      step = (bb.x_max − bb.x_min)/4;
4:      loop
5:          bbL1 = BoundingBox(bb.x_min, xLine − step, bb.bb.y_min, bb.y_max);
6:          bbL2 = BoundingBox(xLine − step, bb.x_max, bb.bb.y_min, bb.y_max);
7:          bbR1 = BoundingBox(bb.x_min, xLine + step, bb.bb.y_min, bb.y_max);
8:          bbR2 = BoundingBox(xLine + step, bb.x_max, bb.bb.y_min, bb.y_max);
9:
10:         pL1 = calculatePlane(input, bbL1);
11:         pL2 = calculatePlane(input, bbL2);
12:         pR1 = calculatePlane(input, bbR1);
13:         pR2 = calculatePlane(input, bbR2);
14:
15:         l1Error = calculateError(input, pL1);
16:         l2Error = calculateError(input, pL2);
17:         r1Error = calculateError(input, pR1);
18:         r2Error = calculateError(input, pR2);
19:
20:         if step < 1 then
21:             return xLine;
22:         else
23:             if l1Error + l2Error > r1Error + r2Error then
24:                 xLine = xLine − step;
25:                 step = step/2;
26:             else
27:                 xLine = xLine + step;
28:                 step = step/2;
29:             end if
30:         end if
31:     end loop
32: end procedure
```

Initial wird die Trennlinie in die Mitte des Ausschnitts gelegt und die Schrittweite auf ein Viertel der Höhe/Breite gesetzt. Um den Lookahead zu erreichen, werden beide Teilhälften weiter unterteilt. Die dazu notwendigen Trennlinien entsprechen den beiden nächsten möglichen Schritten: Trennlinie + Schrittweite und Trennlinie - Schrittweite. Die so definierten Teilflächen werden wie in Abb. 2.4 beschrieben in Ebenen aufgeteilt.

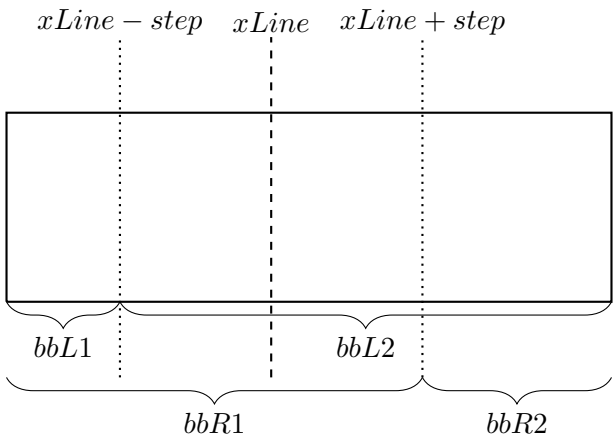



Abbildung 2.4: 1-Schritt Lookahead

Ist die Summe der Fehler von bbL1 und bbL2 kleiner als die Summe der Fehler von bbR1 und bbR2 minimiert der Schritt $xLine - step$ den Fehler und wird als neue Trennlinie übernommen. Ansonsten wird die neue Trennlinie auf $xLine + step$ gesetzt. Die Schrittweite wird halbiert und die Schritte solange wiederholt, bis die Schrittweite kleiner als 1 ist. Die Laufzeit für den Suchalgorithmus ist damit $2log(n)$, wobei $n$ die Länge der Kante ist, an der nicht geschnitten wird.

### 2.2.7 Fehlerfunktion

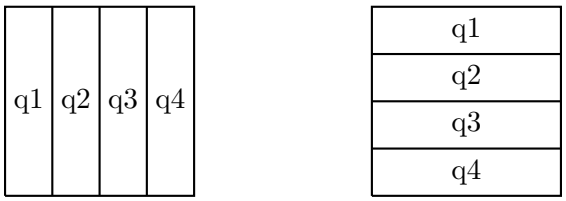
Der Algorithmus benötigt zwei Arten von Fehlerfunktionen: eine allgemeine Funktion, die einen Fehler über eine gesamte Ebene bestimmt und eine Fehlerfunktion, die den Fehler entlang einer Achsenparallelen errechnet. Beide Funktionen benutzen Integralbilder, um den Rechenaufwand zu Lasten der Genauigkeit zu vermindern.
Um den Fehler einer Ebene zu bestimmen, wird der quadratische Abstand aller Punkte, welche der Ebene zugeordnet sind, errechnet. Statt mit einer Schleife über alle Punkte zu iterieren kann wieder ein über die Integralbilder bestimmter Mittelwert benutzt werden. Da die Genauigkeit des Fehlerwertes auf diese Weise stark abnimmt, wurde die Anzahl der gewählten Mittelpunkte erhöht, indem der Ausschnitt, in dem die Ebene liegt, in weitere Abschnitte aufgeteilt wurde. Es muss an dieser Stelle beachtet werden, dass die Quadranten um 90° gedreht werden müssen (Abb. 2.4), da ansonsten die gleichen Mittelpunkte errechnet werden, die zur Bestimmung der Ebene benutzt wurden. Der Fehler wäre in diesem Fall 0. Zu den errechneten Mittelpunkten wird die Distanz zur Ebene bestimmt und quadratisch aufsummiert:

| | |
|---|---|
| q3 | q4 |
| q1 | q2 |

Abbildung 2.5: Quadranten für die Fehlerbestimmung

$$E(P) = \frac{dist(q1)^2 + dist(q2)^2 + dist(q3)^2 + dist(q4)^2}{4}$$

Analog dazu lässt sich der Fehler entlang einer der Koordinatensystemachsen bestimmen. Lediglich die gewählten Abschnitte (vgl. Abb. 2.6) müssen verändert werden.




(a) Fehler entlang der x-Achse (b) Fehler entlang der y-Achse

Abbildung 2.6: Bildaufteilung für den gerichteten Fehler

Die Nutzung von Mittelwerten beschränkt jedoch die erreichbare Qualität der Schätzung, da in Spezialfällen ein kleiner Fehler geschätzt wird, obwohl der tatsächliche Fehler groß ist.
Als Beispiel soll (vgl. Abb. 2.7) betrachtet werden: Es wurde eine Linie in eine zweidimensionale Punktwolke gefittet. Da auf beiden Seiten der Linie eine identische Anzahl an Punkten den gleichen Abstand zu dieser Linie haben, liegt der Mittelwert über alle Punktkoordinaten genau auf dieser Linie; der Fehlerwert ist damit 0. In der Praxis können weitere Fälle auftreten, in denen der Fehler groß sein müsste, der Mittelwert über die Koordinaten aber einen geringen Abstand zur Ebene hat. Das wiederum führt dazu, dass schlechte Approximationen als valide Lösung akzeptiert werden. Aus diesem Grund wird später noch der Einsatz einer alternativen Fehlerfunktion diskutiert.

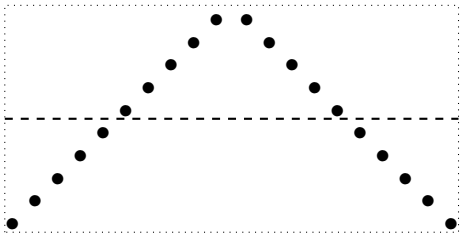


Abbildung 2.7: Fehlerhafte Approximation

### 2.2.8 Ebenen verschmelzen

In der Praxis zerlegt der Algorithmus Punktwolkenausschnitte in mehrere Ebenen, obwohl sie mit einer einzelnen Ebene ausreichend beschrieben werden können. Daher können Ebenen, die redundante Informationen tragen, also einer anderen Ebene gleichen, wieder zu einer einzelnen Ebene zusammengefasst werden.

**Algorithm 5** merges similar planes

```
1: procedure MERGEPLANES(input, minDist, minAngle)
2:     newPlanes = list();
3:     loop(p1 = input.next())
4:         if newPlanes.size <1  then
5:             newPlanes.push(p1);
6:             continue;
7:         end if
8:         loop(p2 = newPlanes.next())
9:             angle = |p1.n[0] * p2.n[0] + p1.n[1] * p2.n[1] + p1.n[2] * p2.n[2]|;
10:            d1 = p1.getDist(p2.center[0], p2.center[1], p2.center[2]);
11:            if d1<minDist&&angle>minAngle then
12:                newPlanes.push(p1.merge(p2));
13:                continue;
14:            end if
15:        end loop
16:    end loop
17:    return newPlanes;
18: end procedure
```

Um Ebenen hinreichend vergleichen zu können, werden zwei Kriterien benutzt: der Abstand des Mittelpunktes von der einen zur anderen Ebene und der Winkel zwischen diesen. Dabei kann der Winkel zwischen den Ebenen über den Winkel der Normalenvekoren zueinander bestimmt werden, indem der Betrag des Skalarproduktes der Vektoren ausgerechnet wird. Der Betrag muss benutzt werden, da die Fälle 180° und 0° identisch sind, der Cosinus jedoch unterschiedlich ist. Sollte der Winkel und der Abstand der Ebenen zueinander gering genug sein, wird angenommen, dass die Ebenen die gleiche Fläche in der Punktwolke beschreiben und zu einer Fläche zusammengefasst werden können (analog zu [12, S. 3]). Um zwei Ebenen $e_1, e_2$ zusammenzufassen, werden die Deskriptoren wie folgt ausgerechnet:

$$\vec{n}_{neu} = \frac{1}{||\vec{n}_{e1}+\vec{n}_{e2}||}(\vec{n}_{e1} + \vec{n}_{e2})$$
$$\vec{center}_{neu} = (\vec{center}_{e1} + \vec{center}_{e2})\frac{1}{2}$$
$$d_{neu} = -\langle \vec{center}_{neu}, \vec{n}_{neu} \rangle$$

Die Boundingbox der neuen Ebene ist das Maximum der Grenzen der alten Boundingboxen.

### 2.2.9 Merkmalextraktion (Ground plane detection)

Aus der nun aufbereiteten Liste von Ebenen lassen sich unterschiedliche Merkmale extrahieren. Im Kontext humanoider Roboter wäre ein solches Merkmal der Boden (Ground plane). Indem die Orientierung des Roboters in unser Modell eingeführt wird, erhalten

Ebenen eine konkretere Bedeutung. Um nun den Boden zu identifizieren, wird die unterste Ebene gesucht, deren Normalenvektor parallel zu einem Vektor liegt, der nach oben (oder unten) zeigt.

**Algorithm 6** finds ground plane in a list of planes

```
1:  procedure FINDGROUNDPLANE(input, orientationMatrix, alpha)
2:      farPoïnt = orientationMatrix * (0 −∞ 0)ᵀ;
3:      minDist = inf;
4:      candidate;
5:      loop(p1 = input.next())
6:          orientation = |p1.dotProduct(farPoint)|
7:          distance = p1.getDist(farPoint);
8:          if (orientation > alpha)&&(distance < minDist) then
9:              minDist = distance;
10:             candidate = p1;
11:         end if
12:     end loop
13:     return candidate;
14: end procedure
```

Algorithmus 6 führt diese Schritte aus. Vom Roboter bekommt der Algorithmus eine Drehmatrix, die einen Vektor so dreht, dass er entlang des Kamerakoordinatensystems des Roboters nach oben (entlang der y-Achse) zeigt. Mit dieser Matrix wird ein Vektor multipliziert, der einen Punkt $farPoint$ sehr weit nach unten projiziert. Für jede Ebene wird nun der Betrag des Skalarproduktes dieses Vektors mit dem Normalenvektor berechnet. Ist die Abweichung zu groß, ist die betrachtete Ebene kein valider Kandidat. Unter allen Kandidaten wird nun die Ebene gesucht, deren Abstand zu $farPoint$ der kleinste ist.

### 2.2.10 Mögliche Verbesserungen der Fehlerfunktion

Wie bereits in 2.2.7 beschrieben sind die Ergebnisse nicht notwendigerweise optimal. Eine mögliche verbesserte Fehlerfunktion lässt sich über den Verschiebungssatz [9, S. 9] implementieren. Dafür soll die Definition des zweiten statistischen Moments (Varianz) betrachtet werden:

$$E((x-\mu)^2) = \frac{1}{n}\sum_{i=0}^{n}(x_i-\mu)^2$$

Der Verschiebungssatz besagt nun:

$$\Sigma = E((x-\mu)^2) = E(x^2) - E^2(x)$$

Diese Eigenschaft kann sehr gut benutzt werden, um die notwendige Rechenzeit zu reduzieren, indem man ein Integralbild über die quadratischen Werte (das äußere Produkt)

jeder einzelnen Punktkoordinate einführt. Analog zu den beiden bereits berechneten Integralbildern muss dieses auch nur einmal am Anfang des Algorithmus für die zu untersuchende Punktwolke bestimmt werden. Die so errechnete Kovarianzmatrix $\Sigma$ kann nun benutzt werden, um den Fehler einer Ebene P in Relation zur zugehörigen Punktwolke zu berechnen:

$$e(P) = ||\vec{n}||_\Sigma$$

Die neue Fehlerfunktion $e$ ist also die quadratische Länge des Vektors $\vec{n}$ über $\Sigma$.

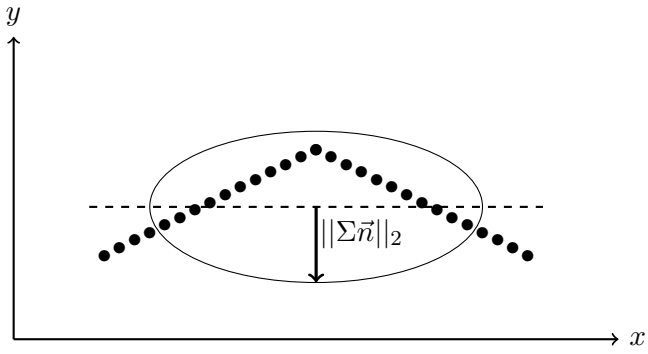



Abbildung 2.8: Geometrisches Beispiel für die neue Fehlerfunktion

Als Beispiel soll wieder das Szenario aus Abb. 2.7 betrachtet werden: Wendet man die neu definierte Fehlerfunktion auf die Ebenenapproximation an, wird nun ein Fehler ungleich 0 bestimmt, da die Varianz der zur Ebene zugeordneten Punktwolke entlang des Normalenvektors der Ebene nicht die Länge 0 hat (vgl. Abb.2.8).

## 2.3 Kalibrierung

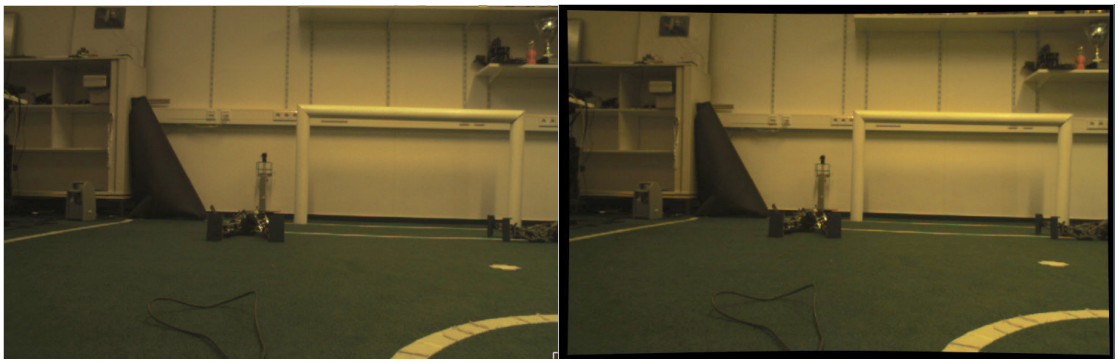
Reale Stereokameras entsprechen zwar dem in Abschnitt 2.1 definierten System, weichen aber in ihren Eigenschaften vom theoretischen Optimum ab. Die auszugleichenden Abweichungen gliedern sich in zwei Kategorien: Intrinsik und Extrinsik. Dabei beschreiben die intrinsischen Werte die Eigenschaften eines einzelnen Kameramoduls: die Brennweite, das optische Zentrum und die Bildverzerrung. Die extrinsischen Werte charakterisieren den Zusammenhang der beiden Kameramodule zueinander: Rotation und Translation. Die intrinsischen Werte jeder Kamera werden in jeweils einer Matrix und einem Vektor angegeben:

$$K := \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \qquad D := \begin{pmatrix} k_1 \\ k_2 \\ p_1 \\ p_2 \end{pmatrix}$$

Dabei beinhaltet die intrinsische Kameramatrix $K$ die Brennweiten $(f_x, f_y)$ jeweils in $x$,$y$-Richtung und das optische Zentrum $(c_x, c_y)$. Wird ein 3D-Punkt aus dem Weltkoordinaten mit $K$ multipliziert, so wird dieser auf die Bildebene der Kamera abgebildet. $D$

entspricht den radialen $(k_1, k_2)$ und tangentialen $(p_1, p_2)$ Verzerrungsparametern. Diese Werte sind besonders wichtig, da nur entzerrte (bzw. rektifizierte) Bilder (vgl. Abb. 2.9) für das Stereomatching, das Finden korrespondierender Pixel zwischen den Kameras, geeignet sind, da jede Kamera bauartbedingt leicht unterschiedliche optische Eigenschaften hat. Die extrinsischen Werte des Kamerasystems bestimmen die Eigenschaften der Kameras zueinander. Im konkreten Fall handelt es sich dabei um die Translation, gespeichert als einfacher Vektor, und die Rotation (als Rotationsmatrix).

Um diese Werte zu bestimmen, wurde auf vorhandene Implementierungen aus der openCV Bibliothek zurückgegriffen. Die Algorithmen approximieren die Parameter, indem ein bekanntes Referenzmuster (z.B. Schachbrett) aufgenommen wird. Dabei werden mehrere Aufnahmen zu einem gemeinsamen Ergebnis zusammengefasst, um die Qualität des Ergebnisses zu erhöhen.



(a) Rohbild (b) Entzerrtes Bild

Abbildung 2.9: Einfluss der Verzerrungsmatrix $D$

### 2.3.1 Bildschärfebestimmung

Die Qualität der erreichten Kalibrierung hängt stark von der Bildqualität der Kameras ab und um eine optimale Bildqualität zu erreichen, muss die Bildschärfe maximal sein. Beim Kalibrieren der Kameras gab es große Probleme den Bildschärfeunterschied zwischen den einzelnen Kameramodulen subjektiv abzuschätzen. Daher wurde als Hilfsmittel der Laplacesche Operator [4, S. 9] benutzt, um einen numerischen Wert für den Vergleich der einzelnen Bilder zu bekommen. Der errechnete Wert korrespondiert zum Bildkontrast und ist relativ, d.h. er eignet sich also nur zum Vergleich der Bildschärfe zwischen Kameras, wenn diese die gleiche Szene aufnehmen.

# Kapitel 3

# Versuche

In diesem Kapitel soll die Implementierung des Algorithmus überprüft werden. Da die Lösung später auf einem Fußballroboter eingesetzt werden soll, sind die Versuche bereits auf themenverwandte Szenarien abgestimmt. Die während den Experimenten gewonnenen Informationen werden weiter benutzt, um zu evaluieren, inwiefern die Viskos Kamera für einen späteren Einsatz bei den FUmanoids geeignet ist.

## 3.1 Versuchsaufbau

Die Kinect als Referenzsensor und die Viskos Stereokamera wurden auf einem gemeinsamen Instrumententräger fixiert. Da beide Sensoren mit gemeinsamen Einheiten arbeiten, ist es einfach die Punktwolken so zu verschieben, dass sie übereinander liegen. Alle Berechnungen wurden auf einem Lenovo L460 (Intel i5-6200U 4GB Ram) ausgeführt. Für jeden Versuch wurde sowohl das Versuchssystem (die Kameras untereinander), als auch die Viskos Kamera neu kalibriert, um optimale Ergebnisse zu gewährleisten.

## 3.2 Allgemeine Bildseparation

Ziel dieses Experimentes ist es, zu evaluieren, wie gut der Algorithmus eine gegebene Szene in Ebenen zerlegen kann. Dabei wird die Szene einmal mit möglichst wenig Störelementen aufgenommen, um zu zeigen, dass der Algorithmus in der Lage ist, eine sinnvolle Ebenenzerlegung zu erzeugen. In einem weiteren Schritt werden Roboter in das Bild gestellt. Die Ergebnisse der Kinect werden jeweils mit den Ergebnissen der Viskos Kamera verglichen. Da beide Kameras mit den gleichen Einheiten arbeiten und zueinander so kalibriert wurden, dass ihre Punktwolken übereinander liegen, sind die Versuchsparameter des Algorithmus für beide Systeme identisch.

### 3.2.1 Hindernisfreie Szene

Als Versuchsbild (vgl. Abb. 3.1) wurde der Blick von der Spielfeldmitte zum Tor gewählt. Die Kameras befinden sich 30cm über dem Boden. Das Kunstlicht entspricht

dabei üblichen Bedingungen bei Fußballspielen im Rahmen des Robocups, die in Hallen ausgetragen werden. Es soll allgemein gezeigt werden, dass der Algorithmus in der Lage ist eine einfache, aber nicht vollkommen triviale Szene mit einer begrenzten Anzahl von Ebenen zu approximieren.

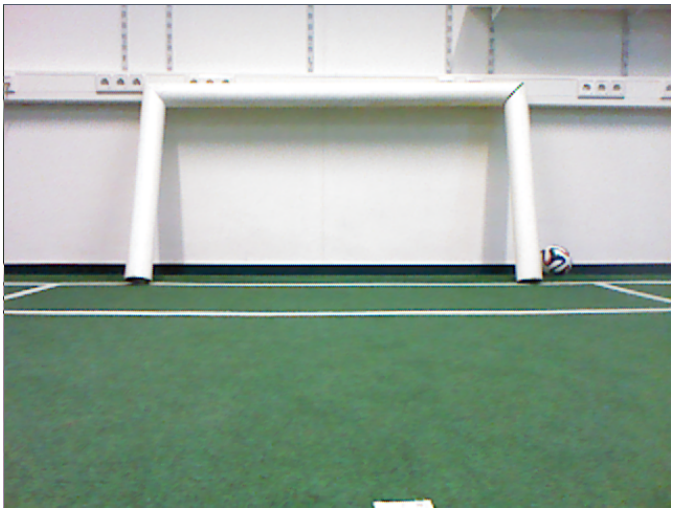
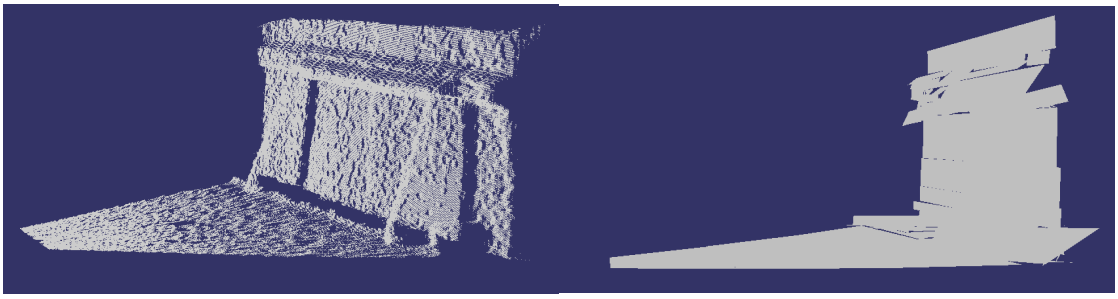


Abbildung 3.1: Versuchsbild, hindernisfrei

### Kinect

Erwartungsgemäß sind die von der Kinect gelieferten Daten qualitativ sehr hochwertig. Als aktiver Sensor benötigt er bestimmte Lichtverhältnisse, um optimal zu funktionieren (wie z.B. kein direktes Sonnenlicht), die in diesem Experiment gegeben sind.



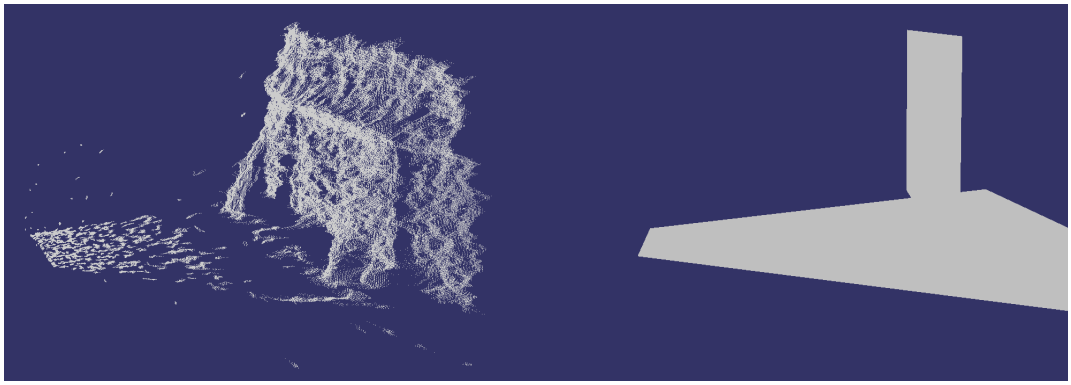
(a) Punktwolke (b) Approximierte Ebenen

Abbildung 3.2: Hindernisfreie Szene: Kinect

### Viskos Kamera

Die Punktwolke der Viskos Kamera gibt die gewählte Szene analog zur Kinect wieder. Allerdings ist diese Punktwolke ungenauer und vor allem im freien Raum zwischen Tor

und Boden von kleinen Punktclustern durchzogen, die auf fehlbestimmte Disparitätswerte der Kamera zurückzuführen sind.



(a) Punktwolke (b) Approximierte Ebenen

Abbildung 3.3: Hindernisfreie Szene: Viskos Kamera

**Auswertung**

Obwohl die Viskos Kamera eine ungenauere Punktwolke erzeugt, ist die Ebenenapproximation ausreichend. Dass trotz der hohen Streuung, gerade zur Wand hin, nur zwei Ebenen benötigt werden, ist auf die gewählte Fehlerfunktion zurückzuführen, die für gleichmäßig um die Ebene verteilte Punkte kleine Fehlerwerte schätzt. Verglichen dazu wird die Punktwolke der Kinect in bedeutend mehr Ebenen zerlegt. Dies ist darauf zurückzuführen, dass Informationen nicht durch zufällige Punkte verfälscht werden: Die Ebenen liegen viel genauer auf der tatsächlichen Oberfläche und müssen dementsprechend den gegebenen Strukturen folgen. Bei beiden Sensoren ist der Boden die flächenmäßig größte Ebene. Sie ist damit hinreichend geeignet, um von späteren Algorithmen erkannt zu werden.

### 3.2.2 Szene mit Hindernissen

Interessanter ist die Ebenenzerlegung für Szenen mit mehr Objekten. In diesem Abschnitt soll abgeschätzt werden, wie sich der Algorithmus auf realitätsnäheren Daten verhält. Analog zum vorherigen Versuchsbild wurde eine Aufnahme von dem Tor ab der Mittellinie des Spielfeldes gewählt (vgl. Abb. 3.4). Abweichend befindet sich die Kamera nun auf 45cm Höhe über dem Boden und drei Roboter wurden im Bild platziert. Die Lichtverhältnisse sind unverändert.

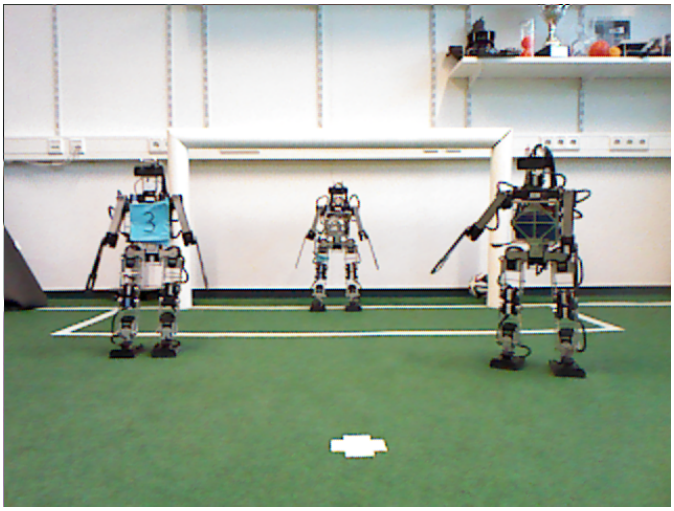
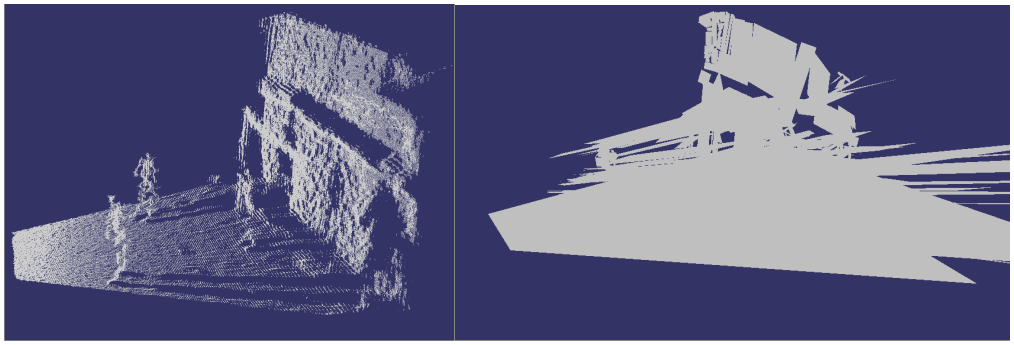


Abbildung 3.4: Versuchsbild mit Hindernissen

### Kinect

Die Aufnahmequalität der Kinect hängt in erste Linie von den Beleuchtungsverhältnissen ab. Da sich diese nicht verändert haben, ist die berechnete Punktwolke (vgl. Abb. 3.5a) unverändert präzise. Die Daten werden mit ungefähr 450 Ebenen approximiert (vgl. Abb. 3.5b), wobei der Boden unverändert die dominante Ebene ist. Zu beachten ist, dass die Ausreißer in den Ebenen im rechten Teil reine Darstellungsfehler sind und den Wahrheitsgehalt der Approximation nicht verändern.



(a) Punktwolke (b) Approximierte Ebenen

Abbildung 3.5: Szene mit Hindernis: Kinect

### Viskos Kamera

Die identische Szene wird mit der Viskos Kamera aufgenommen. Werden die gleichen Startparameter für den Algorithmus gewählt, die auch bei der Kinect verwendet wurden, wird lediglich eine einzelne Ebene (vgl. Abb. 3.6) gefittet. Es war weiterhin nicht möglich,

Parameter zu ermitteln, die eine akzeptable Zerlegung ergaben. Entweder wurde eine einzelne Ebene oder mehrere Tausend Ebenen gefittet.

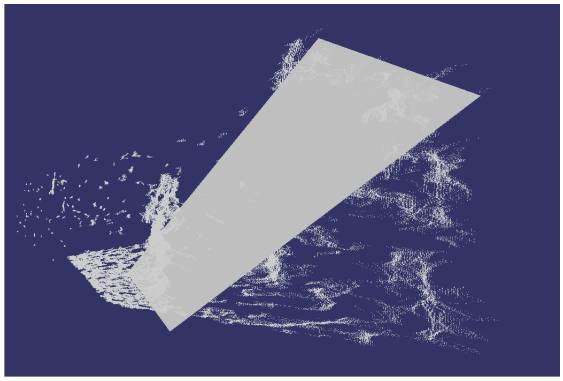


Abbildung 3.6: Szene mit Hindernis: Viskos Kamera

**Auswertung**

Durch die hohe Genauigkeit der Punktwolken, die von der Kinect aufgenommen werden, können trotz der Hindernisse noch die relevanten Ebenen (vor allem Boden und Wände) extrahiert werden. Allerdings werden auch die Roboter im Bild mit Ebenen geschätzt. Diese Ebenen haben in der derzeitigen Form keine Aussagekraft und müssen entweder herausgefiltert oder weiter aufbereitet werden. Verglichen mit der Kinect war es der Viskos Kamera nicht möglich, aus den Daten nutzbare Ebenen zu berechnen. Das Rauschen der Punktwolke in Kombination mit den Robotern im Bild hat es unmöglich gemacht, dominante Ebenen in irgendeiner Form zu bestimmen. Mit reduzierter Fehlerakzeptanz des Algorithmus konnte zwar die Anzahl der extrahierten Ebenen erhöht werden, es wurden aber in erster Linie Ebenen um durch Bildrauschen entstandene Teilpunktwolken gebildet. Damit ist der Algorithmus auf verrauschten Daten nur bedingt einsetzbar. Sie müssen dementsprechend vorher aufbereitet werden.

## 3.3 Ground plane detection

Um den Boden zu extrahieren, wurde der in Abschnitt 2.2.9 vorgeschlagene Algorithmus implementiert. Dabei wurde auf die Verwendung der Orientierungsdaten verzichtet, da die Kameraposition für die statischen Aufnahmen bekannt war. Als Eingabe wurde die von der Kinect generierte Punktwolke aus dem vorherigen Abschnitt benutzt. Die relevante Ebene konnte problemlos identifiziert und extrahiert (vgl. Abb. 3.7) werden. Die im Bild vorhandenen Hindernisse zwischen Kamera und Boden stellten kein Problem dar.

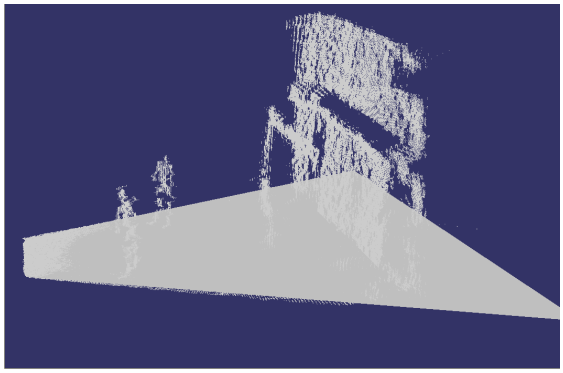


Abbildung 3.7: Extrahierte Ground plane

## 3.4 Benötigte Rechenleistung

Die exakte Laufzeit des Algorithmus hängt neben der Größe der Punktwolke auch stark von der Anzahl der Ebenen, die extrahiert werden, ab. Diese Eigenschaft macht eine theoretische Laufzeitanalyse schwierig. Daher soll in diesem Versuch eine experimentelle Abschätzung der Laufzeit in Relation zum tolerierten Fehler gegeben werden. Es wurden zwei Szenen ausgewählt, die eine Zerlegung in eine große Anzahl von Ebenen ermöglichen. Für jede dieser Szene wurden für akzeptierte Fehler von $10^{-0}$ bis $10^{-8}$ jeweils zwei Aufnahmen mit der Kinect gemacht. Dabei wurde jeweils die Anzahl der extrahierten Ebenen und die Laufzeit in Millisekunden für die beiden Szenen festgehalten:

| | Szenen | | | | |
|---|---|---|---|---|---|
| Error | 1.1 | 1.2 | 2.1 | 2.2 | Mean |
| $10^{-0}$ | 1 | 1 | 1 | 1 | 1 |
| $10^{-1}$ | 1 | 1 | 1 | 1 | 1 |
| $10^{-2}$ | 1 | 1 | 1 | 1 | 1 |
| $10^{-3}$ | 8 | 1 | 57 | 69 | 33.75 |
| $10^{-4}$ | 136 | 198 | 263 | 253 | 212.5 |
| $10^{-5}$ | 532 | 884 | 1308 | 1138 | 965.5 |
| $10^{-6}$ | 2525 | 2503 | 3414 | 3093 | 2883.75 |
| $10^{-7}$ | 4035 | 4396 | 4964 | 4950 | 4579.5 |
| $10^{-8}$ | 4513 | 5470 | 6189 | 5823 | 5498.75 |

Tabelle 3.1: Anzahl extrahierter Ebenen

| | Szenen | | | | |
|---|---|---|---|---|---|
| Error | 1.1 | 1.2 | 2.1 | 2.2 | Mean |
| $10^{-0}$ | 17 | 19 | 19 | 17 | 18 |
| $10^{-1}$ | 15 | 20 | 19 | 16 | 17.5 |
| $10^{-2}$ | 16 | 17 | 12 | 12 | 14.25 |
| $10^{-3}$ | 16 | 19 | 14 | 14 | 15.75 |
| $10^{-4}$ | 16 | 19 | 14 | 16 | 16.25 |
| $10^{-5}$ | 18 | 29 | 21 | 29 | 24.25 |
| $10^{-6}$ | 27 | 30 | 41 | 33 | 32.75 |
| $10^{-7}$ | 33 | 30 | 32 | 30 | 31.25 |
| $10^{-8}$ | 36 | 41 | 41 | 39 | 39.25 |

Tabelle 3.2: Laufzeiten (ms)

Die Aufnahmen wurden mit der Kinect ausgeführt (Größe der Punktwolke: 640x480 Punkte) und der Algorithmus (kompiliert mit SSE Unterstützung) auf einem Lenovo L460 (Intel i5-6200U 4GB RAM) ausgeführt.

### Auswertung

Ausgehend von den geplotteten Daten (vgl. Abb. 3.8) lassen sich weitere Aussagen treffen. Bis zum akzeptieren Ebenenfehler von $10^{-4}$ werden nur wenige Ebenen extrahiert und die Laufzeit verhält sich annähernd konstant, was auf den Initialisierungsaufwand des Algorithmus zurückzuführen ist. Nach dieser Schwelle steigt die Anzahl der Ebenen schnell an und die Laufzeit folgt der Anzahl linear. Dieser Zusammenhang ermöglicht sehr kleine Fehlertoleranzen: Verkleinert sich der Fehler um eine Zehnerpotenz, erhöht sich die Anzahl der Ebenen etwa um 1,5.

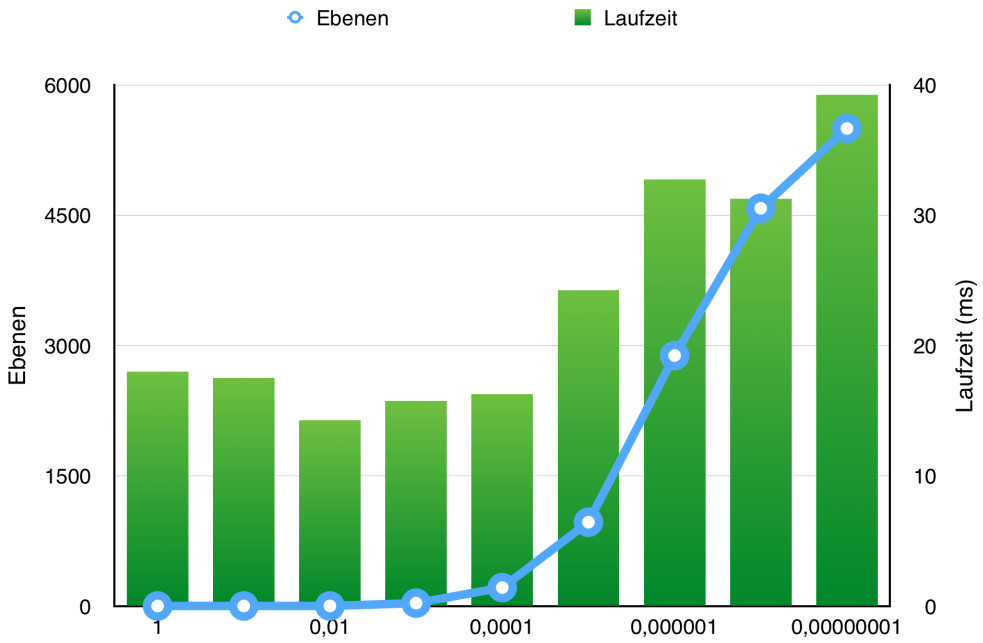



Abbildung 3.8: Laufzeit/Ebenen

In der Praxis sind Fehlerwerte ab $10^{-4}$ interessante Startparameter für den Algorithmus, da ab diesem Wert eine adäquate Anzahl von Ebenen approximiert wird, die Laufzeit aber noch eine hohe Verarbeitungsgeschwindigkeit erlaubt.

## 3.5 Stromverbrauch/Netzwerkauslastung der Viskos Kamera

Während der Benutzung der Kamera gab es immer wieder Probleme eine adäquate Stromversorgung zu finden, da nur USB zur Stromversorgung zur Verfügung stand. Dabei hat sich gezeigt, dass alle Stromquellen, die weniger als 2A bei 5V liefern, zu Problemen im Betrieb führen. Konkret nahm erst mit sinkender elektrischer Leistung die Qualität des Disparitätenbildes immer weiter ab (Fehlmatches im Bild, die zu Flecken

führten), bis die Kamera letztlich die Tendenz entwickelt hat, abzustürzen. Da die auf dem Roboter eingesetzte Stromplatine problemlos 10 Ampere auf der 5V Domäne leisten kann, ist die Stromversorgung auf dem Roboter kein Problem.
Um die Bilder verschicken zu können, benötigt die Kamera zwingend eine 1Gbit Netzwerkverbindung. Bei einer Bildgröße von 752x480 Bildpunkten und 8bit Farbtiefe pro Farbbild (16bit für das Disparitätenbild) werden mindestens

$$\frac{752 * 480 * 4 * 8 * 20}{10^6} = 231.0144 Mbit/s$$

benötigt, um die Bilder mit der vollen Geschwindigkeit von 20 FPS zu empfangen (der Kommunikationsaufwand ist noch nicht berücksichtigt). Da die alten Roboter noch auf der Odroid xu2 Plattform (mit 100Mbit Netzwerk) aufbauen, kann die Kamera nicht auf diesen Robotern eingesetzt werden. Der neue Roboter benutzt Odroid xu4 Computer, die eine 1Gbit Netzwerkkarte intrigiert haben.

# Kapitel 4

# Fazit

## 4.1 Stereovision

Stereooptische Kameras sind die einzigen Tiefenkameras, die von den Regeln des Robocups erlaubt werden. Roboter mit solchen Kameras haben im Spiel enorme Vorteile: Das genaue Messe von Distanzen erlaubt komplexere taktische Entscheidungen und erleichtert die Lokalisierung auf dem Spielfeld. Die Bildverarbeitung ist nicht mehr so stark auf Farbinformationen angewiesen und wird damit stabiler gegenüber instabilen Lichtverhältnissen (insofern die Stereokamera damit umgehen kann). Mit der gesteigerten Rechenleistung der neuen Robotergeneration steht der Bildverarbeitung genug Kapazität zur Verfügung, um die neuen Daten zu verarbeiten. Die eingesetzte Kamera sollte dabei in der Lage sein, Einzelbilder mit der gleichen (oder besseren) Qualität wie die aktuelle Kamera aufzunehmen, um tatsächlich eine Ergänzung darzustellen.

## 4.2 Bildseparation

Der vorgestellte Algorithmus stellt eine effiziente Möglichkeit dar, von Tiefenkameras aufgenommene Punktwolkenbilder zu komprimieren, indem sie in Ebenen zerlegt werden. Der konsequente Einsatz von Integralbildern beschleunigt die Ausführungsgeschwindigkeit so stark, dass Echtzeitanwendungen ermöglicht werden, da in Abhängigkeit der Punktwolkengröße Frameraten jenseits der 30 FPS möglich sind. Allerdings setzt der Algorithmus in der derzeitigen Implementierung (ohne die verbesserte Fehlerfunktion) darauf, dass die Eingabedaten rauschfrei sind, ein Filter der Punktwolke ist also unabdingbar. Aus den verarbeiteten Daten lassen sich ohne großen Rechenaufwand interessante Merkmale (z.B. der Boden) extrahieren. Besonders dominante Merkmale können auch dann noch extrahiert werden, wenn sie im Bild durch Hindernisse verstellt werden.

## 4.3 Viskos Kamera

Die Viskos Kamera ist die einzige der beiden in dieser Arbeit benutzten Tiefenkameras, die tatsächlich im Roboterfußball eingesetzt werden darf, da die Regeln aktive Sensoren (wie die Kinect) verbieten. Der allgemeine Umgang und die mit ihr ausgeführten Versuche zeigen aber, dass sie in ihrer aktuellen Konfiguration so nicht eingesetzt werden kann. Die Kalibrierung wird bereits durch einfaches Hantieren gestört und die Bildqualität nimmt ab. Um die Kamera wieder zu kalibrieren, müssen die bereits vorhandenen Kalibrierungsroutinen des Roboters erweitert werden – eine Tatsache die, gerade im Wettkampfszenario zu vermeiden ist. Des Weiteren ist das Bild so verrauscht, dass der in dieser Arbeit behandelte Algorithmus zur Bildseparation in der aktuellen Implementierung nicht eingesetzt werden kann und die Bildqualität der Einzelbilder ist bedeutend schlechter als die der aktuell auf dem Roboter eingesetzten Kamera. Weiterhin ist die Hardware und Software sehr schlecht dokumentiert und baut auf einem veraltetem OROCOS Framework auf. Ein Bildempfang per UDP ist zwar möglich, um aber in Kameraparameter eingreifen zu können, müssen entweder Konfigurationsdateien auf der Kamera editiert werden oder eine Adaption des OROCOS Frameworkes auf Roboterseite geschrieben werden, was gegeben der Tatsache, dass OROCOS im Rahmen des Projektes nicht gepflegt wird, zu vermeiden ist. Damit wäre ein Einsatz kein Mehrgewinn für die Bildverarbeitung des Roboters.

# Kapitel 5

# Offene Fragen/Aufgaben

## 5.1 Umfangreichere Versuche

In dieser Arbeit wurden nur statische Tests für festgelegte Szenen durchgeführt. Um mehr Aussagen über den Algorithmus treffen zu können, sollten die Versuche erweitert werden: Die Größe der Punktwolke muss verändert werden, um abzuschätzen, wie sich die Laufzeit in Relation zu den Eingabedaten verändert, und Versuche auf dynamischeren Daten (Szenen in Bewegung) sollten durchgeführt werden. Es gilt weiter festzustellen, ob die in dieser Arbeit vorgeschlagene Veränderung der Fehlerfunktion tatsächlich zu besseren Ergebnissen führt. Die Versuche für die Laufzeitabschätzungen wurden auf einem x86_64 System ausgeführt, das mehr Leistung als die auf dem Roboter eingesetzte ARM-Plattform hat. Um die Performance auf dem Roboter zu ermitteln, müssen Laufzeitversuche auf dem Zielsystem ausgeführt werden.

## 5.2 Allgemein

Die aktuell implementierte Funktion zum Verschmelzen von Ebenen sollte überarbeitet werden, da es zu zahlreichen fehlerhaften Verschmelzungen kommt, wenn die Parameter so gewählt werden, dass ausreichend aggressiv verschmolzen wird. Die erreichte Datenkompression ist damit noch niedrig, da viele redundante Ebenen erhalten bleiben. Ein möglicher Ansatz, den es zu untersuchen gilt, wäre der Einsatz einer Baumstruktur, um die Relationen der Ebenen zueinander abzubilden.

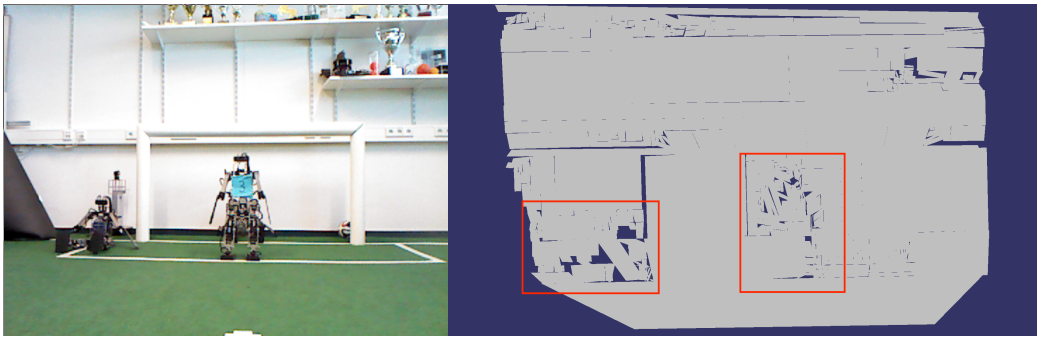
## 5.3 Distanzbestimmung

Einer der wesentlichen Vorteile der Stereokamera ist die Fähigkeit, Distanzen zu Punkten im Bild messen zu können, ohne dass weitere Informationen zum Messpunkt vorhanden sein müssen (wie z.B. Objektgröße). Dabei haben Stereokameras das Problem, dass sie zu Objekten mit wenig Textur keine genaue Abschätzung machen können, weil zu wenig Informationen vorhanden sind, um feststellen zu können, ob zwei Pixel korrespondieren.

Man kann nun aber die extrahierten Ebenen benutzen, um genau an an solchen Stellen, mit wenig Textur, Distanzen zu interpolieren.

## 5.4 Objekte extrahieren über Ebenenrauschen

Da die Ebenen bereits unter Einsatz von Rechenzeit bestimmt wurden, wäre es sinnvoll, die so komprimierten Daten für andere Probleme einzusetzen. Zum Beispiel wäre es möglich interessante Bereiche über die Anzahl der in dieser Region bestimmten Ebenen zu identifizieren. Es soll das in Abb. 5.1 gezeigte Beispiel betrachtet werden. Ziel ist es, die beiden Roboter zu identifizieren. Nachdem die Ebenen approximiert wurden und die Szene aus der Perspektive der Kamera betrachtet wird, fällt auf, dass sich an den Positionen der Roboter viele kleine Ebenen befinden. Interpretiert man die Ebenenapproximation als Bild, das aus der Kameraperspektive aufgenommen wurde und eine bedeutend kleinere Auflösung hat (z.B. 320x240 Pixel)und in dem in jedem Pixel die Anzahl der Ebenen steht, die von diesem Pixel abgedeckt werden, kann man einfach unruhige (d.h. mit hoher Informationsdichte) Flächen im Ursprungsbild bestimmen.



(a) Szene (b) Approximierte Ebenen (Roboter markiert)

Abbildung 5.1: Objekte erhöhen die Anzahl der Ebenen pro Flächeneinheit

## 5.5 Toleranz der Viskos Kamera gegenüber Erschütterung

Die geplanten Versuche zur Toleranz der Kalibrierung der Viskos Kamera gegenüber Erschütterungen durch Laufen/Stürze des Roboters konnten nicht durchgeführt werden, da der benötigte Kameraträger nicht rechtzeitig fertiggestellt werden konnte. Durch die alltägliche Benutzung wurde aber klar, dass leichte Belastungen der Kamera in ihrer derzeitigen Konfiguration bereits dazu führen, dass die Genauigkeit der Aufnahmen drastisch abnimmt. Da die Kamera bereits in Anwendungsgebieten eingesetzt wurde, die eine hohe Vibrationsbelastung aufweisen, ist bekannt, dass man die Kalibrierung mit wenigen Veränderungen (intern verklebte Objektive und die Objektive in den Träger kleben) stabilisieren kann. Da die mechanische Belastung des Roboters sehr hoch ist (vgl. Abb. 5.2), gilt es trotz allem noch zu überprüfen, ob die Veränderungen in diesem

Anwendungsgebiet ausreichend sind.

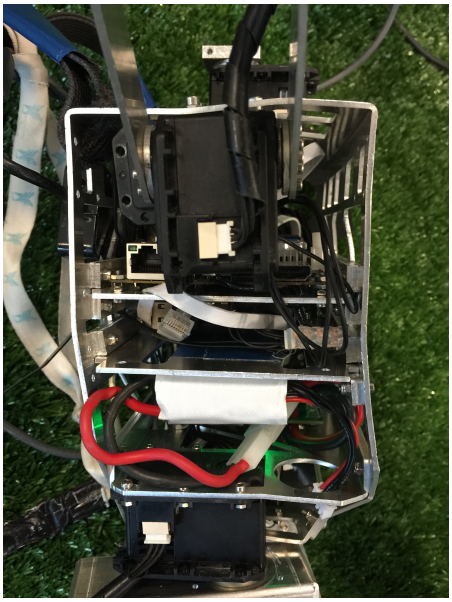


Abbildung 5.2: Durch Stürze beschädigter Roboter

# Literatur


[1] Justin Hensley u. a. "Fast Summed-Area Table Generation and its Applications". In: *Computer Graphics Forum*. Bd. 24. 3. Wiley Online Library. 2005, S. 547–555.

[2] Stefan Holzer u. a. "Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, S. 2684–2689.

[3] Klaas Klasing u. a. "Comparison of surface normal estimation methods for range sensing applications". In: *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE. 2009, S. 3206–3211.

[4] Shree K Nayar und Yasuo Nakagawa. "Shape from focus". In: *Pattern analysis and machine intelligence, IEEE Transactions on* 16.8 (1994), S. 824–831.

[5] *odroid xu4 Specifications*. 2016. URL: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825&tab_idx=2.

[6] *openCV Project Homepage*. 2016. URL: http://opencv.org.

[7] *Orocos: http://www.orocos.org/*. 2016. URL: http://www.orocos.org/.

[8] *RoboCup Soccer Humanoid League Rules and Setup*. 2016. URL: http://www.robocuphumanoid.org/wp-content/uploads/HumanoidLeagueRules2015-06-29.pdf.

[9] Leopold Schmetterer. *Einführung in die mathematische Statistik*. Springer-Verlag, 2013.

[10] Hans Schupp. *Elementargeometrie*. 1977.

[11] *The Freenect Project Homepage*. 2016. URL: https://github.com/OpenKinect/libfreenect.

[12] Miao Wang und Yi-Hsing Tseng. "Lidar data segmentation and classification based on octree structure". In: *parameters* 1 (2004), S. 5.

[13] Wenyi Zhao und Nagaraj Nandhakumar. "Effects of camera alignment errors on stereoscopic depth estimates". In: *Pattern Recognition* 29.12 (1996), S. 2115–2126.