

Institut für Robotik und Prozessinformatik

Technische Universität Braunschweig

Prof. Dr. Friedrich M. Wahl

Bildverarbeitungs-Praktikum
Sommersemester 2013



Inhaltsverzeichnis

1	Einführung in Matlab	1
1.1	Grundlegendes	1
1.2	Die Image Processing Toolbox	5
2	Versuch 0: Matlab-Übung	9
2.1	Einleitung	9
2.2	Aufgabenstellung	9
2.2.1	Arbeiten mit Matrizen	9
2.2.2	Arbeiten mit Bildern	10
3	Versuch I: Binärbilder	13
3.1	Einleitung	13
3.2	Aufbau	13
3.3	Aufgabenstellung	15
3.3.1	Kameraeigenschaften und ihre Auswirkungen auf das Grauwert- stogramm	15
3.3.2	Automatische Schwellwertberechnung	16
3.3.3	Glättung	17
3.3.4	Binarisierung	18
3.3.5	Morphologische Operationen	18
3.3.6	Etikettierung	19
3.3.7	Hu-Momente	19
3.3.8	Lageschätzung	20
3.3.9	Identifikation	21
4	Versuch II: FFT	23
4.1	Einleitung	23
4.2	Aufgabenstellung	23
4.2.1	Fouriertransformation	23
4.2.2	Hoch- und Tiefpassfilterung	24
4.2.3	Abtastung	25

4.2.4	Ausrichten des Textes	26
4.2.5	Textzeichenerkennung	27
5	Versuch III: Hough-Transformation für Kreise	29
5.1	Einleitung	29
5.2	Aufbau	30
5.3	Aufgabenstellung	30
5.3.1	Vorbereitungen	31
5.3.2	Gradientenoperatoren	31
5.3.3	Kantenbild	32
5.3.4	Hough-Transformation für Kreise	33
5.3.5	Suche nach Höhepunkten in den Hough-Räumen	35
5.3.6	Subpixel-Genauigkeit	36
5.3.7	Ausnutzung der Gradientenorientierung	36
6	Versuch IV: Kodierter Lichtansatz	39
6.1	Kalibrierung	40
6.1.1	Bestimmung der Kamerasehstrahlen	42
6.1.2	Bestimmung der Projektorebenen	43
6.1.2.1	Extrinsische Kalibrierung	44
6.1.2.2	Die Kalibrierebenen	46
6.1.2.3	Ebenengleichung aus drei Punkten	47
6.1.2.4	Schnitt aus Gerade und Ebene	47
6.1.2.5	Die Projektorebenen	47
6.2	Scannen von Objekten	49
6.3	Zusammenfassung der Aufgabenstellung	53
7	Versuch V: RANSAC	57
7.1	Einleitung	57
7.2	Verfahren	57
7.3	Aufgabestellung	59
7.3.1	Erstellen einer Testszene	59
7.3.2	RANSAC für Ebenen	60
7.3.3	RANSAC für Kugeln	61
7.3.4	Interpretation der Szene	63

Kapitel 1

Einführung in Matlab

Dieses Kapitel gibt eine sehr kurze Einführung in Matlab. Es werden lediglich Grundprinzipien erläutert, die ein Gefühl für den Umgang mit Matlab vermitteln sollen. Eine gute ausführliche Einführung findet sich unter

<http://www.ti3.tu-harburg.de/~haerter/PraktikumI/schramm.pdf> .

1.1 Grundlegendes

Matlab ist ein Programm, das sich gut zur Simulation und Visualisierung mathematischer Zusammenhänge und Ausdrücke einsetzen lässt. Die wichtigsten Elemente der Programmoberfläche bilden das Command Window (Eingabezeile oder Prompt), der Workspace (listet sämtliche vorhandenen Variablen auf) und ein Editor zur Bearbeitung von Skript-Dateien (.m-Dateien).

Sämtliche Befehle können in der Eingabezeile eingegeben und ausgeführt werden. Bei komplexeren Vorgängen empfiehlt sich die Verwendung einer Skript-Datei, die es ermöglicht, komplette Befehlsketten zu generieren und auch programmiertechnische Strukturen wie z.B. Schleifen einzusetzen.

Variablen müssen in Matlab nicht deklariert werden. Bei der Zuweisung eines Wertes zu einem unbekannten Variablennamen wird diese Variable automatisch angelegt und mit einem angemessenen Datentyp typisiert. Zuweisungen erfolgen dabei einfach über das Gleichheitszeichen:

$$Name = Wert$$

Die in Matlab am häufigsten verwendeten Datentypen sind in der folgenden Tabelle aufgelistet.

Typ	Erklärung
uint8	8-bit Integerzahl (0-255)
uint16	16-bit Integerzahl (0-65536)
double	64-bit Gleitkommazahl
char	Zeichenfolge

Umdeklarierungen können über die gewünschte Typbezeichnung und nachfolgende Klammerung des Variablennamens vorgenommen werden.

Name = uint8(Name)

Der Befehl

a = 5

erzeugt eine Variable *a* vom Typ *double* mit dem Wert 5. Sie ist nun im Workspace mit aufgelistet. Über

b = 'text'

wird eine *char*-Variable mit entsprechendem Inhalt erzeugt. Unmittelbar nach Eingabe der Zuweisung und Bestätigung durch die Return-Taste gibt das Programm als Bestätigung eine Rückmeldung über den Inhalt der Variable aus. Diese Rückmeldung kann mit einem nachfolgenden Semikolon unterbunden werden:

b = 'text';

Durch Eingabe des Variablennamens und Bestätigung mit Return wird der momentane Wert einer Variable angezeigt. Aus dem Speicher gelöscht werden können Variablen über

clear a

Der Befehl *clear* ohne Variablenname löscht sämtliche im Speicher befindlichen Variablen. ACHTUNG: Es existiert ebenfalls der Befehl *clear all*, welcher eine wesentlich gründlichere „Reinigung“ des Speichers durchführt. Allerdings bewirkt dies, dass das Aufnehmen von Bildern mit der Kamera nicht mehr funktioniert. Deswegen sollte *clear all* niemals ausgeführt werden!!!

Einfache Rechenoperationen können intuitiv durchgeführt werden über '+', '-', '*', '/', '^' etc, z.B.

b = a+4.89

Auch elementare Funktionen wie Sinus (*sin*) oder Betrag (*abs*) stehen zur Verfügung. Die Argumente werden ihnen in nachfolgenden Klammern übergeben. So liefert

$$s = \sin(2)$$

den Sinuswert zu 2 und weist ihn der Variable *s* zu.

Für viele Berechnungen ist die Verwendung von Matrizen vonnöten. Die Matrix

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

wird definiert über

$$M = [1 \ 2 \ 3; 4 \ 5 \ 6]$$

Das Semikolon signalisiert den Beginn einer neuen Zeile. Auf einzelne Elemente einer Matrix kann über

$$element_{nm} = M(n,m)$$

zugegriffen werden, wobei *n* die Reihe (= Zeile) und *m* die Spalte angibt. Das erste Element hat dabei den Index (1,1). *M*(2,3) würde hier also *element_{nm}* = 6 ausgeben. Auf ganze Bereiche innerhalb einer Matrix wird zugegriffen indem die Position des ersten und des letzten gewünschten Elements getrennt mit einem Doppelpunkt angegeben wird.

$$M_{14} = M(1:2,1)$$

liefert somit

$$M_{14} = \begin{pmatrix} 1 \\ 4 \end{pmatrix}.$$

Genau wie einfache Zahlenwerte können Matrizen über '+' , '-' und '*' addiert, subtrahiert und multipliziert werden.

Die Transponierte einer Matrix wird über das Apostroph berechnet.

$$M_T = M'$$

liefert dementsprechend

$$M_T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Standardmatrizen können über die Befehle *zeros*(*n,m*) (*n* x *m* Nullmatrix), *ones*(*n,m*) (Einsmatrix) und *eye*(*n,m*) (Einheitsmatrix) erzeugt werden.

Logische-Operatoren sind ähnlich wie aus C++ bekannt:

- == überprüft auf Gleichheit
- ~ = überprüft auf Ungleichheit
- && ist eine Und-Verknüpfung
- || ist eine Oder-Verknüpfung

Für Strukturen zur Programmflusskontrolle sollen hier lediglich zwei Beispiele genannt werden:

a) IF-ELSE Anweisung

```
if i==4
    anweisungen1
elseif i>8
    anweisungen2
else
    anweisungen3
end
```

b) FOR-Schleife:

```
for i=1:N
    anweisungen
end
```

Hier wird die Schleife genau N-mal durchlaufen mit Zählindex i. Über $i=1:S:N$ kann die Schrittweite S definiert werden, um die i erhöht wird. S kann dabei auch eine Gleitkommazahl sein.

Funktionen können in Matlab nur über separate .m-Dateien definiert werden. Diese müssen denselben Dateinamen wie die Funktion selbst aufweisen. Die zugehörige Syntax lautet:

```
function result = funktionsname(wert)
    anweisungen
    result = rückgabewert;
```

Die Funktion wird dann z.B. über

```
a = funktionsname(6.89);
```


in der Eingabeaufforderung oder einer anderen Skriptdatei ausgeführt. Sollen mehrere Werte übergeben bzw. zurückgegeben werden, so ändert sich die Syntax zu

```
function [result1 result2] = funktionsname(wert1, wert2)
    anweisungen
    result1 = rückgabewert1;
    result2 = rückgabewert2;
```

und der Funktionsaufruf erfolgt z.B. über

```
[a,b] = funktionsname(var1, 'zeichenfolge')
```

Abschließend sei noch darauf hingewiesen, dass mit

help befehl

jederzeit die Hilfe zu einem Befehl in der Eingabeaufforderung angezeigt werden kann, bzw. mit

doc befehl

die zugehörige Dokumentation von Matlab in einem separaten Fenster geöffnet wird. Soweit die grundlegende Einführung in Matlab. Sie erhebt keinen Anspruch auf Vollständigkeit, sollte jedoch für die Durchführung dieses Praktikums ausreichend sein. Für ausführliche Informationen sei nochmals auf die externe Quelle vom Beginn des Kapitels verwiesen.

1.2 Die Image Processing Toolbox

Die Image Processing Toolbox stellt eine Erweiterung von Matlab dar und bietet Funktionen zur Darstellung und Bearbeitung von digitalen Bildern. Auch hier werden im Folgenden die wichtigsten Funktionen kurz erläutert.

```
I = imread( filename )
```

Liest die Bilddatei *filename* und speichert sie unter dem Variablennamen *I*. WICHTIG: Semikolon nicht vergessen, da sonst sämtliche Inhalte in der Eingabeaufforderung angezeigt werden.

```
imwrite( I, filename )
```

Speichert das Bild *I* in einer Datei mit dem Dateinamen *filename*.

imshow(I, [low high])

Zeigt das Bild *I* im momentan aktiven Grafikfenster an. Der zweite Parameter mit den eckigen Klammern ist optional. Er bewirkt eine lineare Grauwertspreizung der Anzeige, wobei alle Farbwerte, die kleiner als *low* sind auf Farbwert 0, alle die größer als *high* sind auf den Maximalfarbwert gesetzt werden. Bei Angabe der Klammern ohne Inhalt werden *low* und *high* automatisch bestimmt.

figure(n)

Aktiviert Grafikfenster *n* (*n* muss eine ganze Zahl sein). Nachfolgende Operationen (z.B. *imshow*, *subplot*) werden in diesem Fenster ausgeführt.

subplot(n, m, pos)

Erzeugt ein Grafikfenster, das mehrere Bilder anzeigen kann. Es wird die Größe definiert (*n* Zeilen, *m* Spalten) und das zu aktivierende Element benannt (*pos*). Die Position ergibt sich dabei durch Abzählen der Elemente von links oben nach rechts unten. Für Element (1,1) gilt somit *pos*=1, für (2,3) *pos*=6 etc. Die Befehlsfolge

```
subplot(5,3,6);  
imshow(I);
```

erzeugt somit ein Grafikfenster für 15 Elemente und stellt Bild *I* an Position (2,1) dar. WICHTIG: Zwischen den Operationen darf das Grafikfenster NICHT geschlossen werden, da dann sämtliche Einstellungen und Inhalte verloren gehen.

close

Schließt das momentan aktive Grafikfenster. Mit *close all* werden sämtliche geöffneten Grafikfenster geschlossen.

I = rgb2gray(I)

Wandelt das RGB-Bild *I* (3 Farbwerte pro Pixel) in ein Graustufenbild (1 Farbwert pro Pixel) um.

imhist(I)

Zeigt das Grauwertshistogramm von *I* an.

I = imadjust(I, [low high], [bottom top])

Führt eine lineare Grauwertanpassung durch. Dabei wird der Bereich von *low* bis *high* auf den Bereich *bottom* bis *top* abgebildet. Alle Werte werden anteilmäßig angegeben, müssen also zwischen 0 und 1 liegen. Um in einem Graubild mit 256 Farben den Bereich zwischen 100 und 200 auf den gesamten Farbraum zu strecken muss somit

$$I = \text{imadjust}(I, [100/255 \ 200/255], [0/255 \ 255/255])$$

ausgeführt werden.

$$I = \text{imnoise}(I, \text{type})$$

Fügt Bild I Rauschen vom Typ type hinzu. Mögliche Typen sind 'salt & pepper' und 'gaussian' (mit den Hochkommas).

$$I = \text{medfilt2}(I, [N \ M])$$

Führt eine Medianfilterung des Bildes I durch. N und M sind die Dimensionen des Filterkerns in Y- und X-Richtung.

$$I = \text{irpAveragefilt2}(I, [N \ M])$$

Wendet einen Averaging-Filter auf Bild I an. N und M sind die Dimensionen des Filterkerns in Y- und X-Richtung.

$$I = \text{im2bw}(I, T)$$

Führt eine Binarisierung des Bildes I durch. Das Ergebnis ist ein Binärbild. Alle Pixel mit einem Grauwert kleiner gleich T werden auf 0 gesetzt, alle Grauwerte größer als T auf 1. T muss dabei zwischen 0 und 1 liegen, wobei 1 dem höchsten Farbwert entspricht (z.B. 255).

$$[I2 \ N] = \text{bwlabel}(I)$$

Führt eine Etikettierung des Binärbildes I durch, wobei $I2$ das etikettierte Bild und N die Anzahl der zusammenhängenden Objekte zurückgegeben wird.

irpCaptureImage

Mit diesem Befehl ist es möglich, unter Matlab Bilder mit einer angeschlossenen Kamera zu machen. Sollte der Befehl zum ersten Mal ausgeführt werden, erscheint das Konfigurationsfenster in Bild 1.1. Hier muss lediglich die angeschlossene Kamera im obersten Auswahlfeld ausgewählt werden. Daraufhin erscheint das folgende Aufnahmefenster (Bild 1.2). Mittels der drei Buttons kann die Anzeige pausiert (FREEZE) und wieder gestartet (UNFREEZE) werden. Der Button CAPTURE legt das momentan angezeigte Bild im Speicher ab. Durch Schließen des Fensters wird das abgelegte Bild in der Variable *capImg* abgespeichert und kann nun weiter bearbeitet werden. Es empfiehlt sich, den Inhalt von *capImg* sofort in einer anderen Variable zu sichern, da er bei einem erneuten Aufruf von *irpCaptureImage* überschrieben wird.

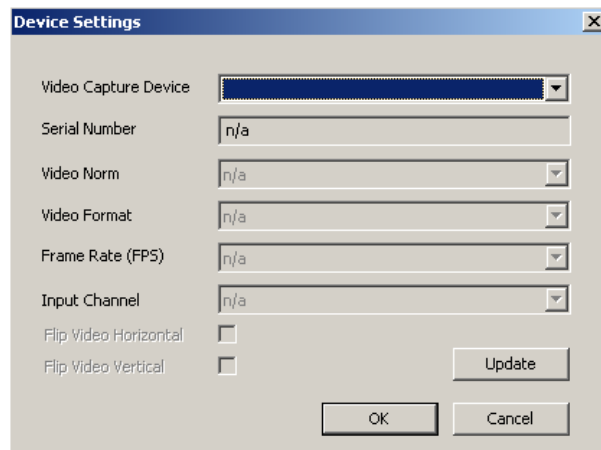


Abbildung 1.1: Initialisierungsfenster der Kamera



Abbildung 1.2: Fenster zur Aufnahme von Bildern

Kapitel 2

Versuch 0: Matlab-Übung

2.1 Einleitung

Dieser Versuch soll dabei helfen, ein Gefühl für die Matlab-Programmierungsumgebung zu entwickeln. Er muss von allen Studenten durchgeführt werden, richtet sich jedoch vor allem an jene, die noch nie mit Matlab gearbeitet haben. Grundlegende Matrixoperationen müssen ebenso ausgeführt werden wie einige Funktionen der Image Processing Toolbox. Die hier erworbenen Grundkenntnisse sind das „Handwerkszeug“, das nötig ist, um die nachfolgenden Aufgaben bearbeiten zu können.

Ein Tipp vorweg: Wie in der Einführung erwähnt bietet Matlab die Möglichkeit Skript-Dateien anzulegen. Hier können ganze Befehlsketten sequentiell abgearbeitet werden. Dies kann viel Schreibarbeit ersparen und ist zudem wesentlich übersichtlicher als sämtliche Befehle in der Eingabeaufforderung einzugeben. Deswegen ist es empfehlenswert sämtliche Aufgaben mittels dieser Skriptdateien zu lösen. Ausgeführt werden können diese übrigens auch mit der F5-Taste.

2.2 Aufgabenstellung

Hinweis: In Matlab muss für jeden Versuch das aktuelle Arbeitsverzeichnis gesetzt werden. Für Versuch 0 ist dies 'P:/bvprak/versuch0'.

2.2.1 Arbeiten mit Matrizen

In Matlab wird sehr häufig mit Matrizen gerechnet. Auch digitalisierte Bilder sind nichts anderes als große Matrizen. Aus diesem Grund soll dieser Aufgabenteil grundlegende Matrizenoperationen vermitteln.

- a) Zu Beginn soll die folgende Matrix erzeugt werden:

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 1 & 1 & 2 & 3 & 4 & 3 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

- b) Generieren Sie nun eine nach rechts mit Nullen erweiterte Einheitsmatrix (so dass die Matrix die gleiche Größe wie M hat) und addieren Sie diese zur Matrix M hinzu.
- c) Multiplizieren Sie das Ergebnis mit der Zahl 3.
- d) Ändern Sie das 8. Element der 4. Reihe auf den Wert 1000.
- e) Mit einem einzigen Befehl sind nun die 9 mittleren Elemente der Matrix auf den Wert 5.8 zu setzen.
- f) Zum Abschluss dieses ersten Aufgabenblocks weisen Sie mittels FOR-Schleifen den 8 Elementen der 2x4-Untermatrix in der linken unteren Ecke der Matrix M den Wert des Produktes ihrer jeweiligen Reihen- und Spaltennummer zu.
Die Matrix sollte nun folgendermaßen aussehen:

$$M = \begin{pmatrix} 3 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 5.8 & 5.8 & 5.8 & 0 & 0 & 0 \\ 3 & 3 & 9 & 5.8 & 5.8 & 5.8 & 6 & 3 & 3 \\ 4 & 8 & 12 & 16 & 5.8 & 5.8 & 0 & 1000 & 0 \\ 5 & 10 & 15 & 20 & 9 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

2.2.2 Arbeiten mit Bildern

In diesem Aufgabenblock soll erstmals mit Bildern gearbeitet werden.

- a) Erstellen Sie eine Nullmatrix mit 240 Reihen und 320 Spalten.
- b) Diese Matrix wird von nun an als Bild aufgefasst. Sie kann mit dem Befehl *imshow* angezeigt werden. Es ist folgendes zu beachten: Wenn Pixelkoordinaten (x,y) in Bildern angegeben werden, so beziehen sich diese auf die linke obere Ecke, wobei die x-Achse nach rechts, die y-Achse nach unten zeigt. Die Matrixposition (n,m) beschreibt jedoch das Element in Reihe n (nach unten) und Spalte m (nach rechts). Die Achsen sind somit vertauscht und der PIXEL (x,y)=(300,100) entspricht dem MATRIXELEMENT (n,m)=(100,300).
Setzen Sie nun ein ausgefülltes Rechteck der Breite 10 und Höhe 5 in der rechten oberen Ecke des Bildes auf den Wert 1 und kontrollieren Sie das Ergebnis mittels *imshow*.

- c) Laden Sie nun das Bild 'capture1.bmp' und zeigen es an.
- d) Bei dem geladenen Bild handelt es sich um ein RGB-Bild. Wandeln Sie es in ein Graustufenbild I um.
- e) Nehmen Sie ein RGB-Bild mit Hilfe der Kamera auf und zeigen es an.
- f) Speichern Sie das aufgenommene Bild in der Datei 'capture2.bmp'.
- g) Zeigen Sie das Grauwertbild I an der Hauptachse gespiegelt an (transponieren).
- h) Zeigen Sie Grauwertbild I und dessen Histogramm in zwei separaten Grafikfenstern (figures) an.
- i) Addieren sie den Wert 100 auf jeden Farbwert von I und zeigen das Ergebnis und das zugehörige Histogramm in einem gemeinsamen Grafikfenster an (subplot).
- j) Führen Sie eine Histogrammspreizung (=Grauwertanpassung) des ursprünglichen Grauwertbildes I mit geeigneten Schwellwerten durch. Zeigen Sie wieder das Ergebnis und das Histogramm in einem gemeinsamen Fenster an.

Hiermit ist der Einführungsversuch abgeschlossen und Sie können sich nun an die Bearbeitung der praxisnäheren folgenden Aufgaben machen.

Kapitel 3

Versuch I: Binärbilder

3.1 Einleitung

Stellen Sie sich folgende Situation aus einer Fertigungshalle vor: Ein Roboter soll verschiedene Bauteile von einem Fließband greifen und in Kisten einsortieren. Diese Bauteile befinden sich auf einem Schlitten, der über das Fließband läuft. Die genaue Anzahl, Position und Orientierung der Bauteile auf dem Schlitten sowie der jeweilige Typ eines Bauteils sind vorher nicht bekannt. Es kann lediglich vorausgesetzt werden, dass nur drei verschiedene Bauteiltypen vorkommen und die Bauteile sich nicht gegenseitig überdecken. Über dem Fließband ist eine Kamera positioniert, die senkrecht auf das Fließband schaut und mehrmals in der Sekunde ein Graubild aufnimmt. Sie sollen nun ein Verfahren entwickeln, das die Bauteile identifiziert und deren Lage bezüglich des Bildkoordinatensystems schätzt. Um später die Lageschätzungen in ein Schlittenkoordinatensystem umrechnen zu können, muss auch der Schlitten selbst identifiziert werden.

3.2 Aufbau

Die Situation aus der Fertigungshalle haben wir in einem Versuchsaufbau nachgestellt. Eine Kamera ist an einem Stativ montiert und schaut senkrecht auf die Stativplatte. Als Testobjekte stehen Ihnen verschiedene DinA4-Ausdrucke zur Verfügung. Hierbei sind Förderband in schwarz, Bauteile in grau und Schlitten in weiß repräsentiert (siehe auch Abbildung 3.1). Ebenfalls vorhanden sind Testbilder in Ihrem Versuchsordner.

Der grobe Ablauf des von Ihnen zu entwickelnden Verfahrens sollte in etwa folgendermaßen aussehen: Ein Grauwertbild der Szene wird akquiriert. Für dieses Bild werden mit Hilfe des Grauwerthistogrammes Schwellwerte berechnet. Diese Schwellwerte nutzt dann ein Algorithmus, um Binärbilder zu generieren. Die Binärbilder werden daraufhin etikettiert, um die einzelnen Regionen zu extrahieren. Die Lage und der Typ der Objekte (Regionen) sollen dann mit Hilfe von Momenten bestimmt werden.



Abbildung 3.1: Testmuster 1

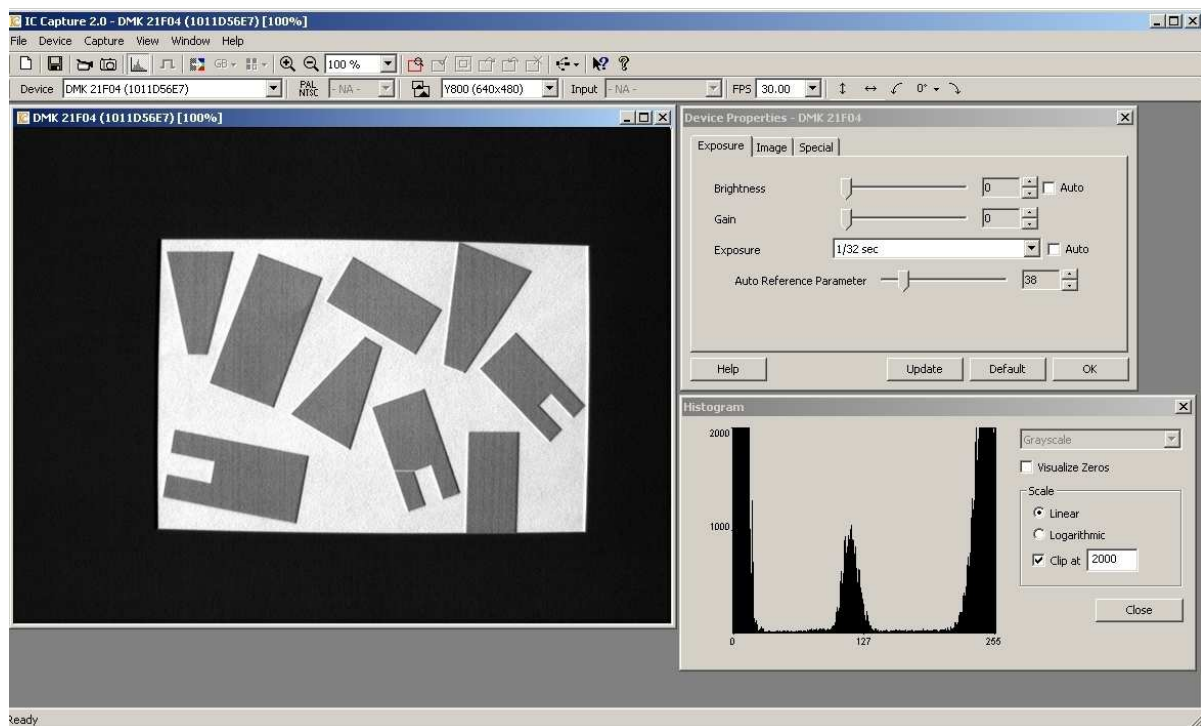


Abbildung 3.2: Oberfläche von IC-Capture

3.3 Aufgabenstellung

Allgemeine Hinweise:

- Für alle zu implementierenden Funktionen sind in dieser Aufgabe bereits alle notwendigen M-Dateien angelegt.
- Dokumentation ist für alle Funktionen vorhanden und kann z.B. per 'help irpAnyFunc' angezeigt werden.
- Um Ihre Funktionen zu testen und Ihnen die Arbeit zu erleichtern, existiert in der Datei '*irpVersuch1.m*' bereits ein ausführbares Ablaufschema für die gesamte Aufgabe. Jede 'Zelle' des Ablaufschemas kann auch einzeln ausgeführt werden.
- Um eigene Funktionen von Matlab-Funktionen zu unterscheiden, fangen jene durchgängig mit '*irp*' an. Sinnvoll ist dies auch in Verbindung mit der Autovervollständigung per Tab-Taste.

3.3.1 Kameraeigenschaften und ihre Auswirkungen auf das Grauwerthistogramm

In der ersten Aufgabe sollen Sie sich näher mit den Eigenschaften der zur Verfügung gestellten Kamera vertraut machen. IC-Capture ist ein einfaches Programm, um sich Live-Bilder samt Grauwerthistogrammen anzuschauen, Einstellungen an den Kameraeigenschaften vorzunehmen und Bildsequenzen aufzunehmen. In der Abbildung 3.2 ist die Oberfläche von IC-Capture zu sehen. Das Fenster zum Einstellen der Kameraeigenschaften erreichen Sie im Menü unter Device/Properties und das Fenster für das Histogramm unter View/Histogram.

- a) Legen Sie Testmuster 1 unter die Kamera (siehe Abbildung 3.1).
- b) Versuchen Sie die Parameter wie in Abbildung 3.2 dargestellt einzustellen.
- c) An der Kamera befinden sich zwei Drehregler. Was stellen diese ein?
- d) Was bedeuten die einzelnen Parameter in den Dialogfenstern?
- e) Welche Auswirkungen haben die Parameter und Drehregler auf das Grauwerthistogramm?
- f) Wie ändert sich das Grauwerthistogramm bei (schnellen) Bewegungen des Testmusters?
- g) Verwenden Sie auch die anderen Testmuster. Wie unterscheiden sich die Grauwert-histogramme?

- h) Wie hängt das Grauwerthistogramm mit dem Förderband, dem Schlitten und den Bauteilen zusammen?
- i) Vergewissern Sie sich, dass Ihr Grauwerthistogramm dem aus Abbildung 3.2 ähnlich ist. Ansonsten könnten Probleme bei den nachfolgenden Aufgaben auftreten. Denken Sie auch bitte daran, dass die Kamera beim nächsten Praktikumstermin verstellt sein könnte und die Einstellungen in IC-Capture zu wiederholen wären.

3.3.2 Automatische Schwellwertberechnung

IC-Capture wird jetzt nur noch benötigt, wenn Sie die Verbindung mit der Kamera testen oder die Kameraparameter einstellen wollen. Alle weiteren Schritte erfolgen nun in Matlab. Ziel dieser Aufgabe ist es, automatisiert einen niedrigen und einen hohen Schwellwert zu berechnen, die später für eine Binarisierung der Bauteile und des Schlittens genutzt werden sollen.

Hinweis: In dieser und anderen Aufgaben werden spezielle Operatoren wie $A.*B$ oder $A < s$ benötigt, wobei A und B Matrizen sind sowie s einen skalaren Wert darstellt. Siehe Dokumentation von Matlab für nähere Informationen.

- a) Setzen Sie das aktuelle Verzeichnis in Matlab auf 'P:/bvprak/versuch1'.
- b) Legen Sie Testmuster 3 unter die Kamera. Aquirieren Sie mit Hilfe von Matlab ein Bild und speichern dieses in der Datei 'testmuster3c.bmp'. Achten Sie in diesem Versuch immer darauf, die RGB-Bilder direkt nach dem Aquirieren in ein Graustufenbild zu wandeln.
- c) Welche Möglichkeiten fallen Ihnen ein, um beide Schwellen automatisch zu berechnen?
- d) Schreiben Sie nun eine Matlab-Funktion

$$[low, high] = irpThresholds (I) ,$$

die ein 8bit-Grauwertbild übernimmt und basierend auf dem Histogramm des Bildes automatisch einen niedrigen und einen hohen Schwellwert berechnet. Dabei sollen alle Pixelwerte unterhalb der Schwelle *low* (also $I(x,y) < low$) zum Hintergrund gehören, alle Pixelwerte zwischen dem unteren und dem oberen Schwellwert zu den Bauteilen und alle Pixelwerte oberhalb der Schwelle *high* zu dem Schlitten. Als Hilfsfunktion können Sie hier

$$[MU, SIGMA, WEIGHT] = irp3Gaussians1d (H)$$

verwenden, die aus dem Histogramm H mit Hilfe des EM-Algorithmus drei gewichtete Gaussverteilungen bestimmt.

- e) Ergänzen Sie die Funktion

$$[RGB] = irpThresholdsGray2RGB (I, low, high) ,$$

die ein Graustufenbild übernimmt und das Segmentierungsergebnis als RGB-Bild zurückgibt.

- f) Testen Sie Ihre Schwellwertmethode mit Hilfe der Testbilder und der zu ergänzenden Funktion

$$[] = irpTestThresholds (I, m, n, p) .$$

Neue Funktionen: *irpCaptureImage*, *irpThresholds*, *irpThresholdsGray2RGB*, *irpTestThresholds*, *imhist*, *imshow*, *imread*, *max*, *sum*, *cumsum*, *subplot*, *cat*

3.3.3 Glättung

Einige der Testbilder weisen ein starkes Rauschen auf. Mit Hilfe von Filteroperationen können Sie das Rauschen mindern und somit die Qualität der Klassifikation erhöhen.

- a) Welche Art von Rauschen könnte in den Testbildern vorhanden sein?
- b) Wodurch könnte dieses Rauschen in der Praxis hervorgerufen werden?
- c) Versuchen Sie mit Hilfe der Filteroperationen *irpAveragefilt2* und *medfilt2* das Rauschen zu mindern. Ergänzen Sie hierzu die Funktion

$$[F] = irpFilter (I, noise, algorithm) .$$

Mittels *algorithm* kann dabei zwischen Average und Median umgeschaltet werden.

- d) Vergleichen Sie die Filteroperationen in der Testfunktion

$$[] = irpTestFilter (I, noise)$$

und entscheiden Sie sich für eine, indem Sie in *irpVersuch1* die globale Variable *algorithm* auf den entsprechenden Wert setzen.

- e) Welche Auswirkungen haben diese Filteroperationen insbesondere auf die Histogramme?

Neue Funktionen: *irpFilter*, *irpTestFilter*, *irpAveragefilt2*, *medfilt2*

3.3.4 Binarisierung

Mit den zuvor ausgerechneten Schwellwerten können Sie nun verschiedene Binarisierungen des Ausgangsbildes durchführen. Zwei Binarisierungen sind hier von Interesse: (1) Alle Pixel, die zu einem Bauteil gehören, werden auf eins gesetzt und alle anderen Pixel auf null und (2) alle Pixel, die zu dem Schlitten gehören, werden auf eins gesetzt und alle anderen auf null.

- a) Schreiben Sie hierfür eine Funktion

$$[Bb Bs] = irpBinarize (I, low, high) ,$$

die ein Binärbild Bb für die Bauteile und ein Binärbild Bs für den Schlitten zurückliefert. In dem jeweiligen Binärbild sollten alle Pixel, die zu dem Objekt gehören, auf eins stehen und alle anderen auf null.

- b) Stellen Sie die Binarisierungen in der Funktion

$$[] = irpTestBinarize (I, low, high)$$

geeignet dar.

Neue Funktionen: *irpBinarize*, *irpTestBinarize*

3.3.5 Morphologische Operationen

In Abhängigkeit von dem Testbild und der Filterung des Bildes werden Sie feststellen, dass einzelne Bauteile in mehrere Regionen zerfallen und/oder ungewollte Artefakte auftreten. Morphologische Operationen können hier helfen, um Regionen wieder zusammenzuführen und kleinere Artefakte zu entfernen. Als morphologische Operationen stehen Ihnen die Dilatation (*imdilate*) und die Erosion (*imerode*) bereit.

- a) Wie funktionieren morphologische Operationen?

- b) In der Funktion

$$[Bm] = irpMorph (B, morph)$$

sollen Sie eine bestimmte Abfolge von Dilationen und Erosionen implementieren, die auf die beiden Binärbilder angewandt werden sollen.

- c) Testen Sie Ihre morphologischen Operationen in der Funktion

$$[] = irpTestMorph (Bb, Bs, morph) .$$

- d) Welche Auswirkungen haben unterschiedliche Abfolgen der morphologischen Operationen?
- e) Welche Nachteile entstehen durch die morphologischen Operationen?

Neue Funktionen: *irpMorph*, *irpTestMorph*, *imerode*, *imdilate*, *strel*

3.3.6 Etikettierung

In diesem Aufgabenteil sollen benachbarte Pixel, die den gleichen Wert haben, zu Regionen zusammengefasst werden. Hierzu haben Sie z.B. die Etikettierung in der Vorlesung kennengelernt. Da das Verfahren nicht ganz unaufwendig in der Implementierung ist, können Sie auf die Matlab-Funktion *bwlabel* zurückgreifen. Als Ergebnis dieser Funktion erhalten Sie für jeden Pixel ein Label, der ihn eindeutig einer Region zuordnet.

- a) Was ist die prinzipielle Vorgehensweise bei der Etikettierung? (Siehe z.B. Skript der Vorlesung „Digitale Bildverarbeitung“.)
- b) Implementieren Sie die Etikettierung in der Funktion

$$[L] = \text{irpLabel} (B)$$

mit Hilfe von *bwlabel*.

- c) Was bewirkt der zweite Parameter in *bwlabel*?
- d) Vervollständigen Sie die Funktion *irpTestLabel* mit Hilfe der Matlab-Funktion *label2rgb*.

Neue Funktionen: *irpLabel*, *irpTestLabel*, *bwlabel*, *label2rgb*

3.3.7 Hu-Momente

Um Bauteile (Regionen) voneinander unterscheiden bzw. klassifizieren zu können, müssen Sie zunächst Eigenschaften definieren, die Sie für einen Vergleich heranziehen können. In der Vorlesung „Digitale Bildverarbeitung“ haben Sie die Hu-Momente kennengelernt (siehe hierzu den Umdruck der Vorlesung). Dies sind sieben Werte, die invariant gegenüber Verschiebungen, Drehungen und Größenänderungen der Bauteile sind. Die sieben Werte können in einem 7d Eigenschaftsvektor zusammengefasst werden, der im Folgenden als Featurevektor bezeichnet werden soll.

Hinweis: Die Hu-Momente werden in *irpHuMoments* noch skaliert, um sie in eine ähnliche Größenordnung zu bringen.

- a) Schauen Sie sich alle neuen Funktionen an (*irpMomentsAll* zuerst). Die Berechnung der Hu-Momente aus den Zentralmomenten wird Ihnen in der Funktion

$$[h] = \text{irpHuMoments} (u)$$

abgenommen. Die Berechnung der Zentralmomente aus den Momenten steht Ihnen mit der Funktion

$$[u] = \text{irpCentralMoments} (m)$$

ebenfalls zur Verfügung. Die Umrechnungen erfolgen dabei jeweils nach den Formeln aus dem Umdruck der Vorlesung „Digitale Bildverarbeitung“.

- b) Sie sollen sich darum kümmern in der Funktion

$$[m] = \text{irpMoments} (Lc)$$

die notwendigen Momente für die Umrechnung in die Zentralmomente zu berechnen. *Lc* ist hier eine Liste von 2d Pixelkoordinaten, die für jede Region berechnet wird und angibt, welche Pixel jeweils zu der Region gehören.

- c) Vervollständigen Sie *irpMomentsAll*.
- d) Nur die ersten beiden Hu-Momente sind für ein (ideales) Rechteck ungleich null. Dies gilt auch für Ellipsen und andere Objekte, die bezüglich ihres Schwerpunktes punktsymmetrisch sind. Das führt dazu, dass z.B. ein Rechteck nicht immer mit Hilfe der Hu-Momente von einer Ellipse unterschieden werden kann. *Schwierig*: Warum sind bei diesen Objekten nur die ersten beiden Hu-Momente ungleich null?

Neue Funktionen: *irpMoments*, *irpCentralMoments*, *irpHuMoments*, *irpMomentsAll*, *irpM*, *find*, *size*

3.3.8 Lageschätzung

Die Lageschätzung der Bauteile soll hier mit Hilfe der zuvor berechneten Momente und Zentralmomente erfolgen.

- a) Ergänzen Sie zuerst die Funktion

$$[P, N] = \text{irpPoseAll} (M, Mu) .$$

- b) Implementieren Sie die Lageschätzung in der Funktion

$$[p, n] = \text{irpPose} (m, u) ,$$

die Momente übernimmt und eine Position sowie Orientierung bezüglich des Kamerakoordinatensystems zurückgibt. Die notwendigen Formeln können Sie dem Umdruck zur Vorlesung „Digitale Bildverarbeitung“ entnehmen.

- c) Wie schätzen Sie die Robustheit der beiden Berechnungen gegenüber einer fehlerhaften Binarisierung ein?

Neue Funktionen: *irpPose*, *irpPoseAll*

3.3.9 Identifikation

Die 7d Featurevektoren spannen einen 7d Featureraum auf. Es geht nun darum jeder der drei Bauteilklassen einen Unterraum des 7d Featureraums zuzuweisen, um die Bauteile anhand Ihrer Featurevektoren identifizieren zu können. Eine Möglichkeit sieht folgendermaßen aus: Für jede Bauteilklassse wird ein Referenz-Featurevektor definiert, für drei Bauteilklassen sind dies also drei Referenz-Featurevektoren. Es werden nun die euklidischen Abstände (andere Metriken sind auch möglich) zwischen jedem Referenz-Featurevektor und jedem Bauteil-Featurevektor berechnet. Wir ordnen das Bauteil der Klasse zu, deren Referenz-Featurevektor die kürzeste Distanz zu dem Bauteil-Featurevektor aufweist. Hier empfiehlt es sich, eine höchstens erlaubte Maximaldistanz einzuführen, um falsch etikettierte Regionen nicht als Bauteile zu klassifizieren. Die einfachste Möglichkeit, um an die Referenz-Featurevektoren zu gelangen, ist von einem Bauteil einer bestimmten Klasse den Featurevektor zu berechnen und diesen als Referenz-Featurevektor zu definieren.

- a) Verwenden Sie z.B. das Eingangsbild *'testmuster1.bmp'*, um die Referenz-Featurevektoren in der Funktion

$$[c] = \text{irpClassify} (h)$$

zu definieren.

- b) Kompletieren Sie *irpClassify*.
c) Ergänzen Sie die Funktion

$$[C] = \text{irpClassifyAll} (Mh) .$$

- d) Welche Auswirkung hat die Skalierung der Hu-Momente in *irpHuMoments* auf die Klassifikation?
e) Welche Möglichkeiten sehen Sie, um die Klassifikation zu verbessern?
f) Könnten Sie sich intelligentere Verfahren zur Klassifikation vorstellen?

- g) Jetzt sollte das Schema aus *irpVersuch1* vollständig abgearbeitet werden können und mit $noise = 1$ und $morph = 1$ ein zu Abbildung 3.3 ähnliches Bild liefern. Testen Sie Ihre Verfahren auch mit Hilfe von „Live“-Bildern.
- h) Unter welchen Bedingungen treten noch Probleme auf?

Neue Funktionen: *irpClassify*, *irpClassifyAll*, *norm*

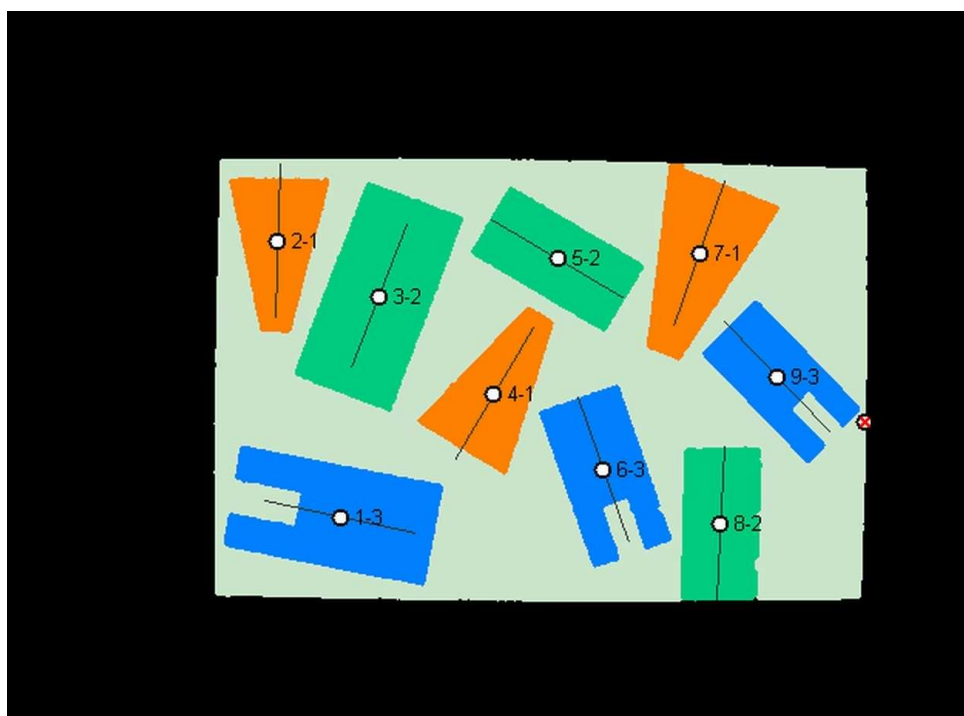


Abbildung 3.3: Ergebnis unter Verwendung des Testbildes 'testmuster3cn.bmp'.

Kapitel 4

Versuch II: FFT

4.1 Einleitung

Als Projektleiter in der Entwicklungsabteilung eines Handyherstellers bekommen Sie folgende Aufgabe: Sie sollen ein Verfahren entwickeln, das es ermöglicht mit der Handykamera Texte zu fotografieren und automatisch in Textzeichen für z.B. eine SMS umzuwandeln. Herausforderungen sind hierbei:

- Sie müssen die Minimalanforderungen an die Auflösung Ihrer Kamera bestimmen.
- Die Texterkennung sollte möglichst robust sein und auch dann funktionieren, wenn das Bild verrauscht ist.
- Ihr Verfahren sollte in der Lage sein, auch verdrehte Texte richtig zu erkennen.

4.2 Aufgabenstellung

Bitte erstellen Sie für die folgenden Aufgaben selbstständig entsprechende Testfunktionen. Diese sollen aus der Datei *'irpVersuch2.m'* aufgerufen werden (vergleiche hierzu *'irpVersuch1.m'*). Durch einen Aufruf von *irpVersuch2* soll es auch möglich sein, für ein gegebenes Grauwertbild *I* alle Schritte der Aufgabenstellung automatisiert ablaufen zu lassen.

4.2.1 Fouriertransformation

Als Grundwerkzeug in diesem Versuch soll die *Fast Fourier Transformation* (FFT) dienen. Im ersten Schritt soll eine Fouriertransformation für Bilder durchgeführt werden. Glücklicherweise bietet Matlab die Funktionen *fft2* und *ifft2* an, mit denen die Fast Fouriertransformationen auf zweidimensionale Funktionen angewendet werden kann. Als Hilfsfunktionen könnten *ifftshift* und *log2* sehr nützlich sein.

- a) Schreiben Sie eine Matlab-Funktion

$$[\textit{amplitude phase}] = \textit{irpFFT} (I) ,$$

die das Amplituden- und das Phasenspektrum eines Bildes I berechnet.

- b) Die zu vervollständigende Matlab-Funktion

$$[I] = \textit{irpInverseFFT} (\textit{amplitude}, \textit{phase})$$

soll eine inverse FFT durchführen.

- c) Laden Sie die Bilder '*Lena.bmp*' und '*iRP-Logo.bmp*' und berechnen Sie das Amplituden- und Phasenspektrum.
- d) Setzen Sie das Amplitudenspektrum eines Bildes auf eine Konstante und führen Sie die Rücktransformation durch. Führen Sie den gleichen Versuch mit dem Phasenspektrum durch. Was fällt Ihnen bei Vergleich der rücktransformierten Bilder auf?
- e) Vertauschen Sie die Phasenspektren der Bilder und führen Sie die Rücktransformation für beide Fouriertransformierte durch. Was hat sich verändert?

Neue Funktionen: *irpFFT*, *irpInverseFFT*, *fft2*, *ifft2*, *ifftshift*, *log2*

4.2.2 Hoch- und Tiefpassfilterung

Die Hoch- und Tiefpassfilterung soll genutzt werden, um den Einfluss von Rauschen zu mindern.

- a) Implementieren Sie hierzu die Matlab-Funktionen

$$[\textit{amp}_t \textit{phase}_t] = \textit{irpTiefpass} (\textit{amp}, \textit{phase}, \textit{size}) \text{ und}$$

$$[\textit{amp}_h \textit{phase}_h] = \textit{irpHochpass} (\textit{amp}, \textit{phase}, \textit{size}) ,$$

die eine Hoch- bzw. Tiefpassfilterung im Ortsfrequenzbereich durchführt. Der Parameter *size* legt die Größe des Filterkerns fest. Sie können zur Definition des Filterkerns die Matlab-Funktion

$$[O] = \textit{irpDrawCircle} (I, \textit{ctr}_x, \textit{ctr}_y, r, \textit{value})$$

nutzen, die einen Kreis mit dem Radius r um das Zentrum $\textit{ctr}_x, \textit{ctr}_y$ ins Bild I zeichnet. Der Parameter *value* definiert den Wert, auf den die Elemente des Kreises gesetzt werden.

- b) Wenden Sie die Hoch- und Tiefpassfilterung auf verschiedene Bilder an. Testen Sie verschiedene Filterkerngrößen.
- c) Überlagern Sie die Bilder mit gaußischem und „Salt and Pepper“ Rauschen. Wie verändern sich die Spektren? (Nutzen Sie hierzu die Matlab-Funktion *imnoise*.)
- d) Nutzen Sie die Tiefpassfilterung, um das Rauschen zu mindern.

Neue Funktionen: *irpTiefpass*, *irpHochpass*, *irpDrawCircle*, *imnoise*

4.2.3 Abtastung

Die Abtastung des Textbildes mit der Handykamera soll in diesem Abschnitt systemtheoretisch mit der Fouriertransformation beschrieben werden.

- a) Schreiben Sie eine Matlab-Funktion

$$[D] = \text{irpDiracfeld} (\text{size}_y, \text{size}_x, \text{delta}_x, \text{delta}_y) ,$$

die ein Diracfeld der Größe size_x , size_y erzeugt. Die Parameter delta_x und delta_y stehen für die Abstände der Diracstöße in x bzw. y-Richtung.

- b) Die Funktion *irpDiracfeld* soll nun zur Abtastung genutzt werden. Erstellen Sie die Matlab-Funktion

$$[O] = \text{irpAbtastung} (I, \text{delta}_x, \text{delta}_y) ,$$

die das Bild I mit den Abständen delta_x und delta_y abtastet und das abgetastete Bild O zurückgibt.

- c) Laden Sie das Bild '*Sonnet_for_Lena.bmp*' und tasten Sie es mit unterschiedlichen Abständen delta_x und delta_y ab.
- d) Vergleichen Sie die Spektren der abgetasteten Bilder.
- e) Wie groß dürfen die Abstände delta_x und delta_y maximal sein, so dass das Abtasttheorem noch erfüllt ist?
- f) Tasten Sie das Bild mit den Abständen $\text{delta}_x = \text{delta}_y = 2$ ab und rekonstruieren Sie das abgetastete Bild mit Hilfe der Tiefpassfilterung.

Neue Funktionen: *irpDiracfeld*, *irpAbtastung*

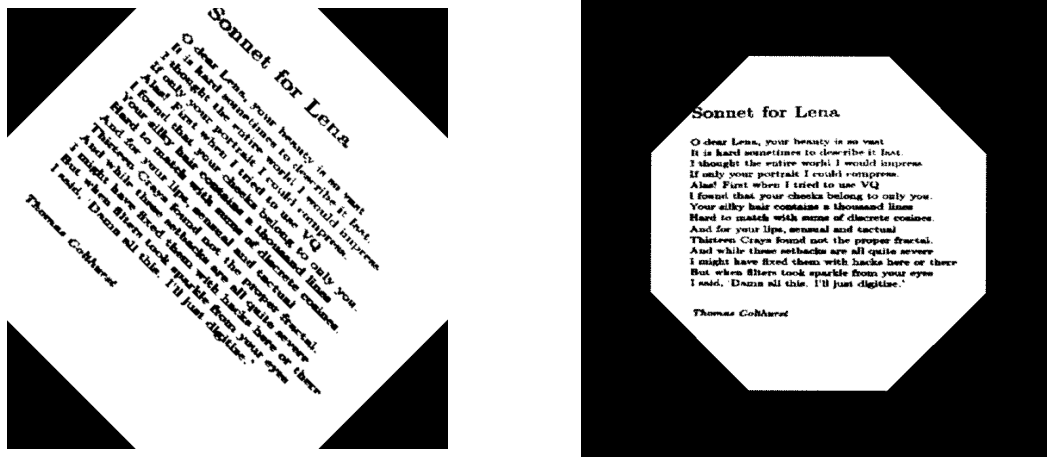


Abbildung 4.1: Verdrehtes und ausgerichtetes Textbild

4.2.4 Ausrichten des Textes

Bevor die Texterkennung durchgeführt werden kann, muss der Text richtig ausgerichtet werden (siehe auch Abbildung 4.1).

- Schauen Sie sich die Spektren für verschiedene verdrehte Texte an. Was fällt Ihnen auf? (*Hinweis:* Das verdrehte Bild *'Sonnet_for_Lena_verdreht.bmp'* befindet sich in ihrem Arbeitsverzeichnis. Weitere Bilder können Sie mit Hilfe der Matlab-Funktion *imrotate* erzeugen.)
- Versuchen Sie den Verdrehungswinkel mit Hilfe des Amplitudenspektrums zu bestimmen. (*Hinweis:* Verwenden Sie keine Hough-Transformation!)
- Nachdem Sie den Verdrehungswinkel ermittelt haben, nutzen Sie die Matlab-Funktion *imrotate* zum Ausrichten.
- Schreiben Sie eine Matlab-Funktion

$$[O] = \text{irpAlignText} (I) ,$$

welche die vorherigen Schritte zusammenfasst und damit den Verdrehungswinkel eines Textbildes I automatisch bestimmt, das Bild ausrichtet und als Bild O zurückgibt.

Neue Funktionen: *irpAlignText*, *imrotate*



Abbildung 4.2: Die Abbildung zeigt das Bild „Lena“ und ein Gesichtstemplate. Beide Bilder wurden durch Ergänzung von Nullelementen auf eine einheitliche Größe gebracht.

4.2.5 Textzeichenerkennung

Im ausgerichteten und entrauschten Textbild $f(x, y)$ sollen nun einzelne Buchstaben mit Hilfe der Korrelationsfunktion ϕ_{fg} erkannt werden. Hierfür steht eine Datenbank mit einzelnen Buchstaben-Templates $g(x, y)$ zur Verfügung. Aus der Vorlesung „Digitale Bildverarbeitung“ ist bekannt, dass sich die Korrelationsfunktion ϕ_{fg} zweier Signale $f(x, y)$ und $g(x, y)$ als Faltung des Signals $f(x, y)$ mit dem Signal $g(-x, -y)$ auffassen lässt:

$$\phi_{fg}(i, j) = \sum_x \sum_y f(x, y) g(x - i, y - j).$$

Die Faltung lässt sich mit Hilfe der Fouriertransformation auch als Multiplikation durchführen

$$\Phi_{fg}(u, v) = F(u, v) G^*(u, v),$$

wobei $\Phi_{fg}(u, v)$ und $F(u, v)$ die Fouriertransformierten von $\phi_{fg}(i, j)$ bzw. $f(x, y)$ sind und $G^*(u, v)$ die konjugiert komplexe Fouriertransformierte von $g(x, y)$. Die Berechnung der Korrelationsfunktion $\phi_{fg}(i, j)$ über die Fouriertransformation ist besonders bei großen Templates zu bevorzugen.

Hinweis: Bevor die Multiplikation im Ortsfrequenzbereich durchgeführt werden kann, müssen $f(x, y)$ und $g(x, y)$ auf eine einheitliche Definitionsmenge erweitert werden. Sei $f(x, y)$ definiert auf $\{(x, y) \in \mathbb{N} \mid 1 \leq x \leq \text{size}X_f \text{ und } 1 \leq y \leq \text{size}Y_f\}$ und $g(x, y)$ entsprechend auf $\{(x, y) \in \mathbb{N} \mid 1 \leq x \leq \text{size}X_g \text{ und } 1 \leq y \leq \text{size}Y_g\}$. Dann überführen Sie die Funktionen in $\tilde{f}(x, y)$ und $\tilde{g}(x, y)$ durch Ergänzung mit Nullelementen, sodass beide

auf $\{(x, y) \in \mathbb{N} \mid 1 \leq x \leq \text{size}X_f + \text{size}X_g \text{ und } 1 \leq y \leq \text{size}Y_f + \text{size}Y_g\}$ definiert sind. Siehe hierzu auch Abbildung 4.2.

- a) Schreiben Sie eine Matlab-Funktion

$$[C] = \text{irpOCR} (I, T) ,$$

welche die Korrelationsfunktion von Bild I und Template T bestimmt und als C zurückgibt.

- b) Laden Sie das Bild '*Sonnet_for_Lena.bmp*' und führen Sie die Zeichenerkennung für die Buchstaben 'L', 'S' und 'n' durch. Entsprechende Templates finden Sie unter den Dateinamen '*Template_X.bmp*'.
- c) Welchen Einfluss hat der Wertebereich von I und T auf die Korrelationsfunktion?
- d) Warum muss das Template um 180° rotiert werden?
- e) Bestimmen Sie das Maximum der Korrelationsfunktion $\phi_{fg}(i, j)$ und markieren (zeichnen) Sie die entsprechende Stelle im Eingangsbild durch ein umschreibendes Rechteck. Nutzen Sie hierfür die Matlab-Funktion

$$[RGB] = \text{irpDrawBoundingBox} (I, x, y, dx, dy, t, col_R, col_G, col_B) .$$

Neue Funktionen: *irpOCR*, *irpDrawBoundingBox*

Kapitel 5

Versuch III: Hough-Transformation für Kreise

5.1 Einleitung

Mit Ihrer Hilfe ist es nun möglich Bauteile auf einem Fließband zu identifizieren und ihre Lage zu schätzen. Nach der erfolgten Sortierung der Bauteile durch einen Robotor soll nun ein weiterer Schritt automatisiert werden: Einige Bauteile haben Bohrlöcher, in die ein Roboter ein Gewinde schneiden soll. Hierzu ist es notwendig, die Position und Größe der Bohrlöcher zu bestimmen. Für diese Aufgabe steht auch hier eine Kamera zur Verfügung, die senkrecht auf das Bauteil schaut. Leider ist aufgrund der inhomogenen Lichtverhältnisse und des reflektiven Metalls ein Verfahren, das mit Hilfe eines Schwellwertes eine Binarisierung des Ausgangsbildes durchführt, nicht robust einsetzbar. Stattdessen bietet sich hier die Hough-Transformation für Kreise an, um die Bohrlöcher zu detektieren.



Abbildung 5.1: Aufnahme eines Bauteils

5.2 Aufbau

Um Ihr Verfahren zu entwickeln, steht Ihnen ein originales Bauteil aus der Fertigung zur Verfügung. Die Laborbedingungen entsprechen also sehr gut der Realität. Das Bauteil weist Bohrungen in zwei unterschiedlichen Größen auf. Nur Bohrungen in diesen zwei Größen müssen von Ihnen erkannt werden. Ihr Verfahren soll aber so flexibel sein, dass die Anzahl der Bohrungen nicht vorgegeben werden muss. Sie können lediglich davon ausgehen, dass von jedem Bohrlochtyp maximal 16 Bohrungen pro Bauteil auftreten können und die Bohrungen sich nicht überschneiden. Die Höhe der Kamera sollte etwa auf $h = 180\text{mm}$ eingestellt sein (siehe Abbildung 5.2).

Hinweis: Legen Sie bitte ein weißes Blatt Papier zwischen Bauteil und Stativplatte, um Kratzer auf der Stativplatte zu vermeiden.

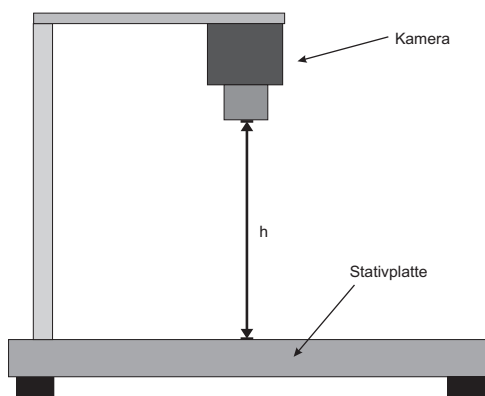


Abbildung 5.2: Skizze zur Einstellung der Kamerahöhe

5.3 Aufgabenstellung

Hinweise:

- Am Ende dieses Versuches sollten Sie durch einen Aufruf von *irpVersuch3* in der Lage sein, alle Schritte automatisch ablaufen zu lassen und das Ergebnis Ihrer Bohrlochdetektion darzustellen.
- Achten Sie darauf, dass Sie bei den verschiedenen Matrizen die richtigen Datentypen (uint8, double, ...) verwenden. Ansonsten könnten einige Berechnungen nicht das von Ihnen gewünschte Ergebnis liefern.

5.3.1 Vorbereitungen

Bevor es mit den Programmierarbeiten losgehen kann, müssen Sie ein paar Einstellungen und Messungen vornehmen.

- a) Stellen Sie sicher, dass die Kamera die gewünschte Höhe hat.
- b) Verwenden Sie IC-Capture und die Regler an der Kamera, um die Parameter so einzustellen, dass Sie kontrastreiche Bilder generieren können.
- c) Mit Hilfe von Matlab müssen die Bohrlöcher eingemessen werden. Hierzu nehmen Sie ein Bild von dem Bauteil mit Hilfe der Kamera auf und stellen es in einer Anzeige (figure) dar. In der Anzeige können Sie sogenannte „datatips“ erstellen, setzen und löschen. Hierzu aktivieren Sie zunächst den „Data Cursor“ unter „Tools“. Anschließend können Sie die „datatips“ mit Hilfe des Kontextmenüs setzen und löschen, wenn Sie mit der rechten Maustaste auf die Anzeige klicken. Um den Durchmesser einer Bohrung zu messen, setzen Sie zwei möglichst weit auseinanderliegende „datatips“ auf den Rand einer Bohrung. Mit Hilfe der Funktion

$$[d] = \text{irpDistDatatips} (\text{fig})$$

können Sie dann in Matlab den Abstand zweier „datatips“ berechnen. Für die weiteren Aufgaben wird der Radius beider Bohrlochtypen benötigt.

Neue Funktionen: *irpDistDatatips*

5.3.2 Gradientenoperatoren

Bevor Sie das für die Hough-Transformation benötigte Kantenbild berechnen, sollen Sie sich näher mit verschiedenen Gradientenoperatoren beschäftigen. Die hier verwendeten Gradientenoperatoren können als Filterkerne definiert werden, die mit Hilfe einer Faltung auf das Eingangsbild angewendet werden. Bei kleinen Filterkernen ist es meist schneller die Filterung als Faltung im Ortsbereich durchzuführen, anstatt als Multiplikation im Ortsfrequenzraum.

Hinweise:

- Als Testbild für diese Aufgabe können Sie z.B. *'testmuster1.bmp'* verwenden. Dieses Testbild soll allerdings nur für diese Aufgabe verwendet werden. Für die anderen Aufgaben müssen Sie eigene Testbilder von dem Bauteil generieren.
 - Als Hilfsfunktion steht Ihnen *imfilter* zur Verfügung.
- a) Führen Sie eine Roberts-Filterung durch und stellen Sie folgende Bilder in einem gemeinsamen „subplot“ dar (siehe auch Abbildung 5.3):

- Ergebnisbild der Filterung in der ersten Richtung
 - Ergebnisbild der Filterung in der zweiten Richtung
 - Bild der Gradientenbeträge
 - Bild der Gradientenwinkel
- b) Führen Sie eine Sobel-Filterung durch und stellen Sie folgende Bilder in einem gemeinsamen „subplot“ dar (siehe auch Abbildung 5.3):
- Ergebnisbild der Filterung in x-Richtung
 - Ergebnisbild der Filterung in y-Richtung
 - Bild der Gradientenbeträge
 - Bild der Gradientenwinkel
- c) Führen Sie eine Laplace-Filterung durch und stellen Sie das Ergebnisbild sowie das Betragsbild dar. Der Laplace-Filterkern ist folgendermaßen aufgebaut:
- $$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$
- d) Führen Sie einen direkten Vergleich der beiden Winkelbilder durch (z.B. auf Robustheit gegenüber Rauschen).
- e) Führen Sie einen direkten Vergleich der drei verschiedenen Betragsbilder durch.
- f) Entwerfen Sie Filterkerne, die nur auf der Diagonalen von links unten nach rechts oben selektiv sind und dabei robust gegenüber Störungen/Rauschen sind. Stellen Sie die Ergebnisse dar.

Neue Funktionen: *imfilter*

5.3.3 Kantenbild

Mit den gesammelten Vorkenntnissen sollen Sie in dieser Aufgabe aus dem Eingangsbild ein binarisiertes Kantenbild erstellen mit folgenden Eigenschaften:

- Die Anzahl der gesetzten Kantenpixel im Binärbild sollte möglichst gering sein.
- Die Außenkontur jedes Bohrloches sollte möglichst komplett enthalten sein.
- Außerdem soll für jeden Kantenpixel auch die Orientierung in Form eines Winkels verfügbar sein.

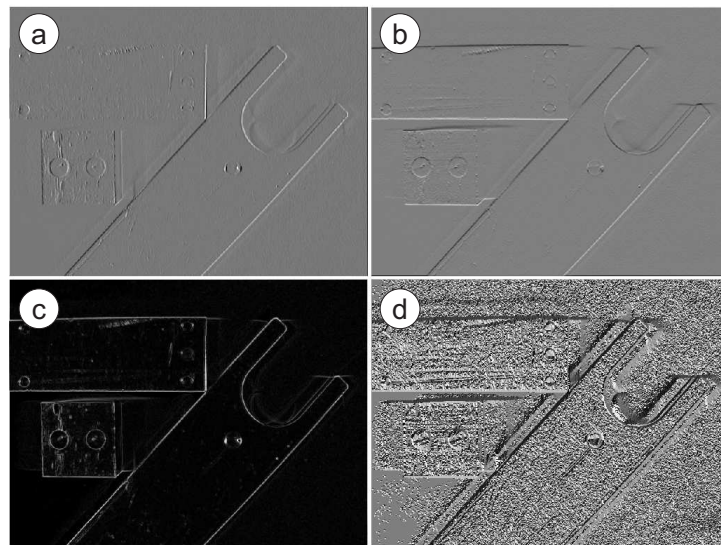


Abbildung 5.3: Ergebnisbilder nach Sobel-Filterung: (a) Filterung in x-Richtung, (b) Filterung in y-Richtung, (c) Betragsbild der Gradienten, (d) Winkelbild der Gradienten

- a) Implementieren Sie ein Gradientenverfahren Ihrer Wahl in der Funktion

$$[E, O] = \text{irpEdges} (I) ,$$

die ein 8bit Grauwertbild übernimmt und ein Kantenbild E und ein Gradientenwinkelbild O zurückliefert.

- b) Da ein binarisiertes Kantenbild E für den weiteren Ablauf benötigt wird, müssen Sie noch eine Binarisierung in *irpEdges* einfügen. Mit Hilfe einer geeigneten Heuristik soll hierfür möglichst automatisch ein geeigneter Schwellwert berechnet werden.

Neue Funktionen: *irpEdges*

5.3.4 Hough-Transformation für Kreise

Mit der Hough-Transformation für Kreise können wir sehr robust die Außenkontur eines Bohrloches detektieren. Ein wesentlicher Vorteil aufgrund der Versuchsvoraussetzungen ist, dass wir die Radien der zu suchenden Kreise bereits kennen. Ansonsten hätten wir einen 3d Hough-Raum aufzubauen, was aus Sicht der Rechenleistung sehr kostspielig wäre. In unserem Fall benötigen wir also für jeden Bohrlochtyp nur einen 2d Hough-Raum.

- a) Sie können als Hilfsfunktion

$$[C] = \text{irpCircle} (cx, cy, omega, r, alpha, imgSize)$$

verwenden, die einen Kreis bzw. zwei gegenüberliegende Kreissegmente in Form einer Liste von Pixelindizes berechnet. Vervollständigen Sie diese Funktion.

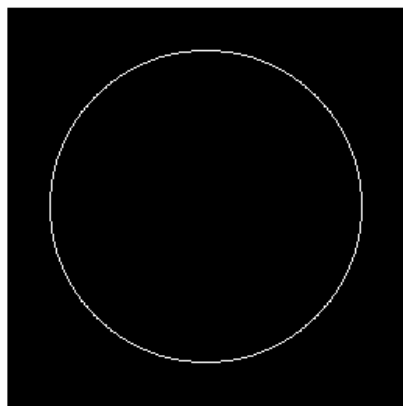
- b) Generieren Sie mit Hilfe von *irpCircle* Bilder wie sie in Abbildung 5.4 zu sehen sind.
- c) Implementieren Sie die Hough-Transformation in der Funktion

$$[H] = \text{irpHough} (B, r) ,$$

die ein Binärbild B samt Kreisradius r übernimmt und einen 2d Hough-Raum H zurückgibt.

- d) Verwenden Sie die von Ihnen in Abschnitt 5.3.1 generierten Testbilder und die zugehörigen Radien, und stellen Sie die entstehenden Hough-Räume geeignet dar.
- e) Welche Strukturen entdecken Sie in den Hough-Räumen und wie entstehen diese?
- f) Schauen Sie sich die einzelnen lokalen Maxima genauer an: Wie ändern sich die lokalen Maxima, wenn sie den vorgegeben Radius ändern? Hier wäre z.B. ein „subplot“ hilfreich, in dem Sie verschiedene Hough-Räume für leicht unterschiedliche Radien direkt nebeneinander darstellen.

Neue Funktionen: *irpCircle*, *irpHough*, *round*



(a)



(b)

Abbildung 5.4: (a) Vollständiger Kreis und (b) zwei gegenüberliegende 90° Kreissegmente, die um 45° rotiert sind.

5.3.5 Suche nach Höhepunkten in den Hough-Räumen

Für beide Bohrlochtypen sind die Hough-Räume erstellt. Nun müssen innerhalb der Hough-Räume die Maxima gefunden werden, um die Bohrlöcher zu lokalisieren. Zwei Schwierigkeiten treten hierbei auf: Erstens suchen wir nicht nur ein globales Maximum, sondern mehrere lokale Maxima, und zweitens ist die Anzahl der zu suchenden Maxima uns vorher nicht bekannt. Das zweite Problem umgehen wir zunächst, indem wir die Anzahl der zu suchenden Maxima exakt vorgeben und den ansonsten benötigten Schwellwert *vorerst* auf null setzen.

a) Vervollständigen Sie die Funktion

$$[peak] = irpFindPeak (H) ,$$

die den Hough-Raum H nach dem globalen Maximum absucht.

b) Da wir wissen, dass sich Bohrungen in unserem Fall nicht überschneiden, können wir die Suche nach mehreren lokalen Maxima folgendermaßen lösen:

- (1) Finde das globale Maximum im Hough-Raum.
- (2) Setze alle Pixelwerte innerhalb eines Kreises, dessen Zentrum sich im gefundenen Maximum befindet, auf null.
- (3) Wiederhole den ersten Schritt oder breche ab, falls das gefundene Maximum unterhalb der vorgegebenen Schwelle liegt oder die maximale Anzahl der zu suchenden Maxima erreicht ist.

Implementieren Sie dieses Verfahren in der Funktion

$$[P] = irpPeaks (H, r, threshold, maxNumPeaks) .$$

Als Hilfsfunktion steht Ihnen die Funktion

$$[R] = irpCut (H, cx, cy, r)$$

bereit, die alle Pixel innerhalb eines Kreises mit Mittelpunkt (cx, cy) und Radius r auf null setzt.

Neue Funktionen: *irpFindPeak*, *irpPeaks*, *irpCut*

5.3.6 Subpixel-Genauigkeit

Bisher wurden die Maxima mit Pixelgenauigkeit erkannt. In der „3d Computersehen“-Vorlesung haben Sie ein Verfahren kennengelernt, das eine Berechnung in Subpixel-Genauigkeit ermöglicht.

- a) Nutzen Sie dieses Verfahren in der Funktion

$$[S] = \text{irpSubPixel} (H, P) ,$$

um Ihre bisher gefundenen Maxima P mit Hilfe des 2d Hough-Raums H zu optimieren. Hierfür ist es notwendig eine geeignete Größe für Ihr Operatorfenster zu wählen.

- b) Zum Testen Ihrer Optimierung steht die Funktion

$$[] = \text{irpTestSubPixel} (H, s, p)$$

bereit, die ein einzelnes Maximum vergrößert im Hough-Raum darstellt.

- c) Stellen Sie das Endergebnis Ihrer Bemühungen mit Hilfe von

$$[] = \text{irpShowResults} (P1, P2, r1, r2, I)$$

dar.

Neue Funktionen: *irpSubPixel*, *irpTestSubPixel*, *irpShowResults*

5.3.7 Ausnutzung der Gradientenorientierung

Bisher haben wir für die Erstellung der Hough-Räume nur die Beträge der Gradienten verwendet. Wenn wir nun die Gradientenrichtung hinzunehmen, können wir das Verfahren sowohl beschleunigen als auch die Robustheit (zumindest in einigen Fällen) erhöhen, da ja die Gradienten eines Kreispunktes orthogonal auf der Tangente stehen sollten. Die Idee hierbei ist, nicht einen kompletten Kreis im 2d Hough-Raum einzuzeichnen, sondern nur zwei gegenüberliegende Kreissegmente, auf denen der vermutete Kreismittelpunkt liegen sollte.

- a) Warum tragen wir nicht einfach zwei einzelne Punkte statt zweier Kreissegmente in den Hough-Raum ein?
- b) Implementieren Sie das erweiterte Verfahren in der Funktion

$$[H] = \text{irpHough2} (B, O, r, \alpha) ,$$

die zusätzlich zu dem Binärbild B und dem Radius r ein Gradientenwinkelbild O sowie die Größe α eines einzelnen Kreissegmentes übernimmt.

- c) Experimentieren Sie mit verschiedenen Werten für α .
- d) An welchen Wert könnte man α koppeln, um α adaptiv für jeden Kantenpunkt zu berechnen?
- e) Welche Nachteile können durch die Einbeziehung der Gradientenrichtung auftreten?

Kapitel 6

Versuch IV: Kodierter Lichtansatz

In diesem Versuch soll ein aktiver 3D-Scanner entwickelt werden. Aktiv ist dabei die Beleuchtung, denn das zu scannende Objekt wird mit einem Projektor angestrahlt. Das Verfahren arbeitet mit dem kodierten Lichtansatz, welches im Folgenden kurz erläutert wird.

In einer bestimmten Abfolge werden Streifenmuster auf ein Objekt projiziert. Werden diese Muster übereinander gelegt, so ergibt sich für jeden feinsten Streifen eine charakteristische Hell-Dunkel-Abfolge im Bild. Anhand dieser ist jeder Streifen eindeutig zu identifizieren. In Bild 6.1 ist dies dargestellt.

Die Streifen werden durch einen Projektor hervorgerufen und können somit als senkrecht ausgerichtete Lichtfelder aufgefasst werden, die jeweils anhand der charakteristischen Hell-Dunkel-Abfolge einen bestimmten dreidimensionalen Raumbereich definieren. Jeder Bereich wird dabei lediglich durch eine Lichtebene repräsentiert, welche sich exakt in der Mitte befindet.

Treffen nun diese Lichtfelder auf ein dreidimensionales Objekt und wird dies mit einer Kamera aufgenommen, so lassen sich für jede Lichtebene exakt die zugehörigen 3D-Punkte berechnen. Dafür notwendig ist die Lage der Lichtebenen (zukünftig Projektorebenen genannt) im Raum bezüglich der Kamera. Werden die Kamerasehstrahlen mit diesen geschnitten, kann die exakte Lage eines 3D Punktes im Raum eindeutig bestimmt werden. Der Aufbau ist in Bild 6.2 skizziert.

Bitte lesen Sie das Kapitel 6 vor der Bearbeitung der einzelnen Aufgabenstellungen vollständig durch. Die Bearbeitung der einzelnen Teilaufgaben soll dann in der Reihenfolge durchgeführt werden, die in der Zusammenfassung in Abschnitt 6.3 aufgelistet ist.

Hinweis: Damit der Versuch korrekt funktioniert, muss der Projektor als zweites Display (erweiterter Desktop) mit einer Auflösung von 800 x 600 Pixeln bei einer Bildwiederholrate von 60 Hertz konfiguriert sein.

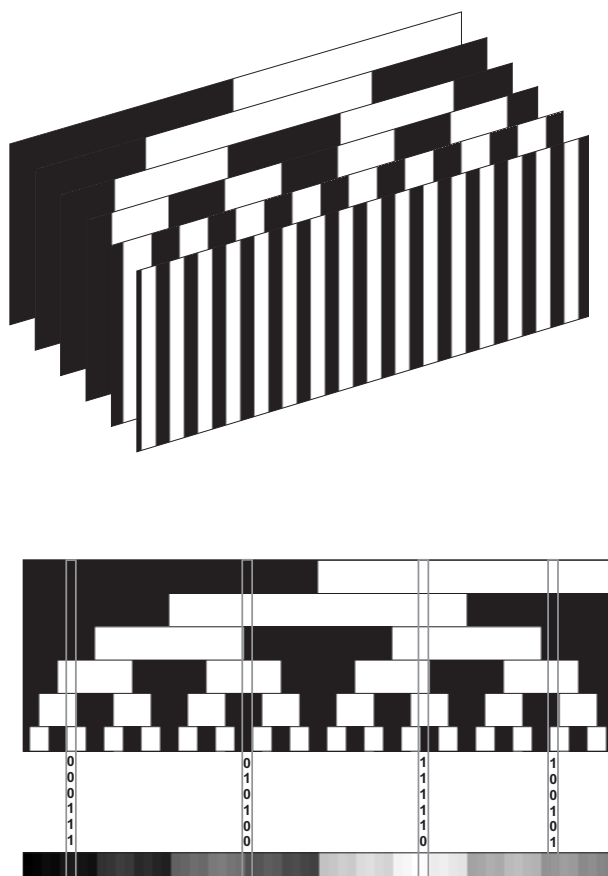


Abbildung 6.1: Die Streifenmuster (oben) werden übereinandergelegt und ergeben somit einen eindeutigen Binärcode für jeden feinsten Streifen (unten).

6.1 Kalibrierung

Ein einleitender Hinweis: In diesem Versuch wird zwischen „Skripten“ und „Funktionen“ unterschieden. Skripte sind eine Abfolge von Befehlen, die im „Haupt-Workspace“ ausgeführt werden. Ihnen müssen daher beim Aufruf nicht direkt Variablen übergeben werden, sondern die benötigten Variablen müssen im Workspace präsent sein. Auch haben Skripte keinen Rückgabewert. Funktionen dagegen werden in einem „Unter-Workspace“ ausgeführt und haben somit keinen direkten Zugriff auf Variablen im Haupt-Workspace. Sie haben immer einen Rückgabewert und benötigte Variablen müssen ihnen explizit übergeben werden.

In diesem Kapitel wird darauf eingegangen, wie das Gesamtsystem kalibriert werden muss, um schließlich 3D-Daten von beliebigen Objekten erfassen zu können. Wichtig ist dabei, dass sämtliche Berechnungen in einem gemeinsamen Koordinatensystem stattfinden

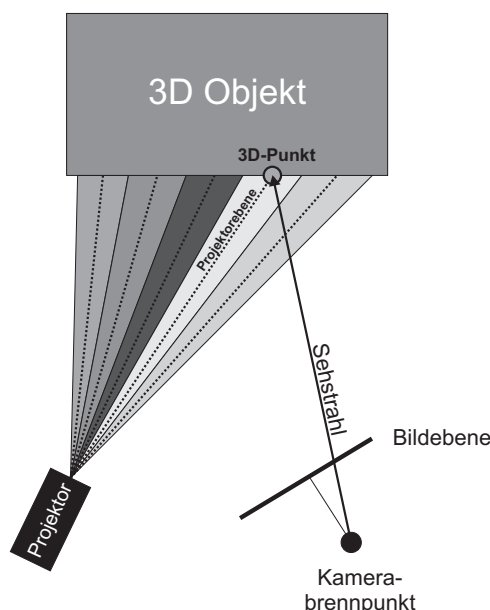


Abbildung 6.2: Versuchsaufbau in der Draufsicht mit Projektor, Kamera und zu scannendem Objekt.

müssen. Hierbei handelt es sich um das Koordinatensystem der Kamera, dessen Ursprung im Brennpunkt der Kamera liegt. Alle 3D Koordinaten müssen auf dieses Koordinatensystem bezogen werden, um sinnvolle Berechnungen zu ermöglichen. Orientierung und Lage sind in Bild 6.3 dargestellt.

Wie in der Einleitung erwähnt, sollen die Projektorebenen mit den Kamerasehstrahlen geschnitten werden. Um dies überhaupt zu ermöglichen, müssen die Sehstrahlen für jeden Pixel ermittelt werden können. Als Grundlage dient dabei das Modell einer Lochbildkamera. Durch eine vorangegangene Kalibrierung der Kamera ist deren Brennweite f bekannt, welche den Abstand von Bildebene und Brennpunkt angibt (siehe *irpInitData*). Üblicherweise wird die Brennweite in *mm* angegeben. Der Einfachheit halber ist sie im Rahmen dieses Praktikums jedoch in Pixeln ausgedrückt. Somit können Brennweite und Pixelkoordinaten im Bild direkt miteinander in Bezug gebracht werden, wobei jedoch weiterhin das Pixel-Seitenlängenverhältnis $s = \frac{s_x}{s_y}$ zu berücksichtigen ist.

Außerdem bekannt sind die Koordinaten des Durchstoßpunkts der Kamera-Z-Achse durch die Bildebene. Dieser Punkt (auch optisches Zentrum genannt) wird mit c_c bezeichnet und ist angegeben in Pixelkoordinaten bezüglich des Bildkoordinatensystems. Nähere Erläuterungen zum Kameramodell sind in den Vorlesungsunterlagen von „Dreidimensionales Computersehen“ zu finden.

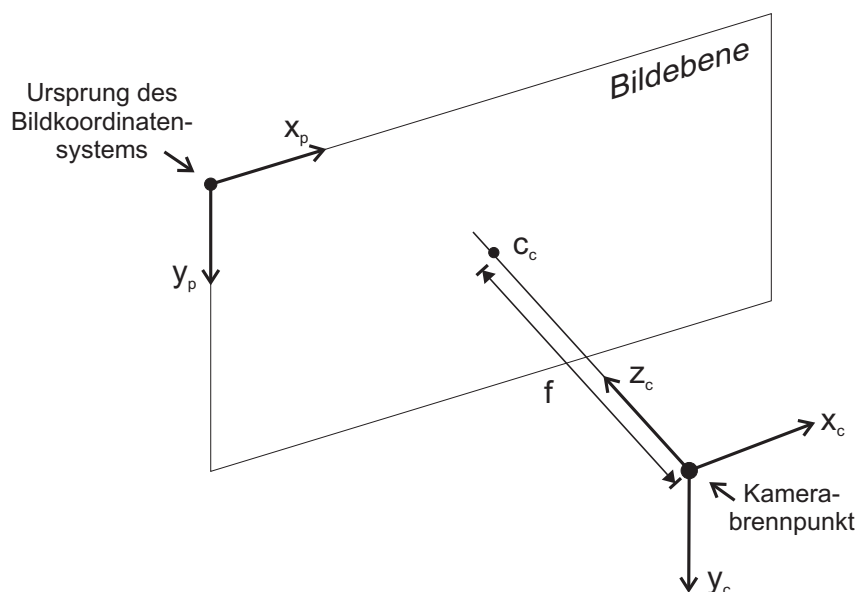


Abbildung 6.3: Skizze mit 3D-Kamerakoordinatensystem und 2D-Bildkoordinatensystem.

6.1.1 Bestimmung der Kamerasehstrahlen

Die erste Aufgabe dieses Versuchs besteht nun darin, eine Funktion zu schreiben, die für jeden beliebigen Pixel in der Bildebene den zugehörigen dreidimensionalen Sehstrahl berechnet. Hierfür soll die Funktion

$$[x_r] = \text{irpPixelToRay}(x, y, f, s, cc)$$

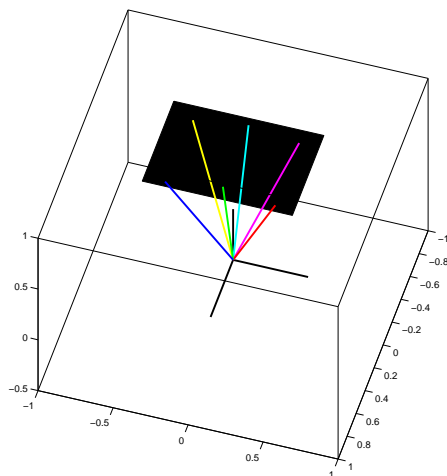
modifiziert werden, welche als Eingangswerte die umzurechnenden Pixelkoordinaten, die Brennweite, das Pixel-Verhältnis s und das optische Zentrum benötigt. Damit diese Werte zur Verfügung stehen, muss das Skript *irpInitData* einmal ausgeführt werden. Zurückgegeben werden soll der Richtungsvektor des Kamerasehstrahls als 3×1 Matrix.

Vorgehensweise:

- Umrechnung der Bildkoordinaten, sodass diese auf das optische Zentrum in der Bildebene bezogen sind.
- Bestimmung des Richtungsvektors x_r mit x-, y- und z-Koordinate (in Pixeln).
- Normierung des Richtungsvektors auf die Länge 1.

Ob die Funktion korrekt programmiert wurde, kann mittels des Skripts

irpTestPixelToRay

Abbildung 6.4: Test der Funktion *irpPixelToRay*.

überprüft werden. Der dort generierte 3D-Plot sollte exakt dem im Bild 6.4 gleichen. Die Funktion ist nur dann korrekt programmiert, wenn alles inklusive der Farben mit diesem Bild übereinstimmt.

6.1.2 Bestimmung der Projektorebenen

Nachdem nun die Kamerasehstrahlen bestimmt werden können, sind die Positionen der Projektorebenen im Kamerakoordinatensystem zu ermitteln. Dafür wird folgendermaßen vorgegangen:

- a) Aufstellen einer Kalibrierebene mit Kalibrierpunkten.
- b) Ermittlung der Position der Kalibrierebene bezüglich der Kamera.
- c) Projektion der Streifenmuster auf diese Ebene.
- d) Verschieben der Kalibrierebene und erneute Bestimmung der Position.
- e) Erneute Projektion der Streifenmuster.
- f) Bestimmung von insgesamt 3 Punkten auf beiden Kalibrierebenen, die jeweils in denselben Projektorebenen liegen.
- g) Generierung der Projektorebenenungleichungen aus diesen 3 Punkten.

Diese Punkte werden in den folgenden Sektionen abgearbeitet und deren Funktion näher erläutert.

6.1.2.1 Extrinsische Kalibrierung

Der Zweck der Kalibrierebenen ist es, Punktkoordinaten im Kamerakoordinatensystem zu ermitteln, welche die einzelnen Projektebenen definieren. Auf den Kalibrierebenen befinden sich Kalibrierpunkte in fest definiertem Abstand. Diese lassen sich somit bezüglich eines Weltkoordinatensystems leicht angeben. Bild 6.5 stellt den Zusammenhang dar.

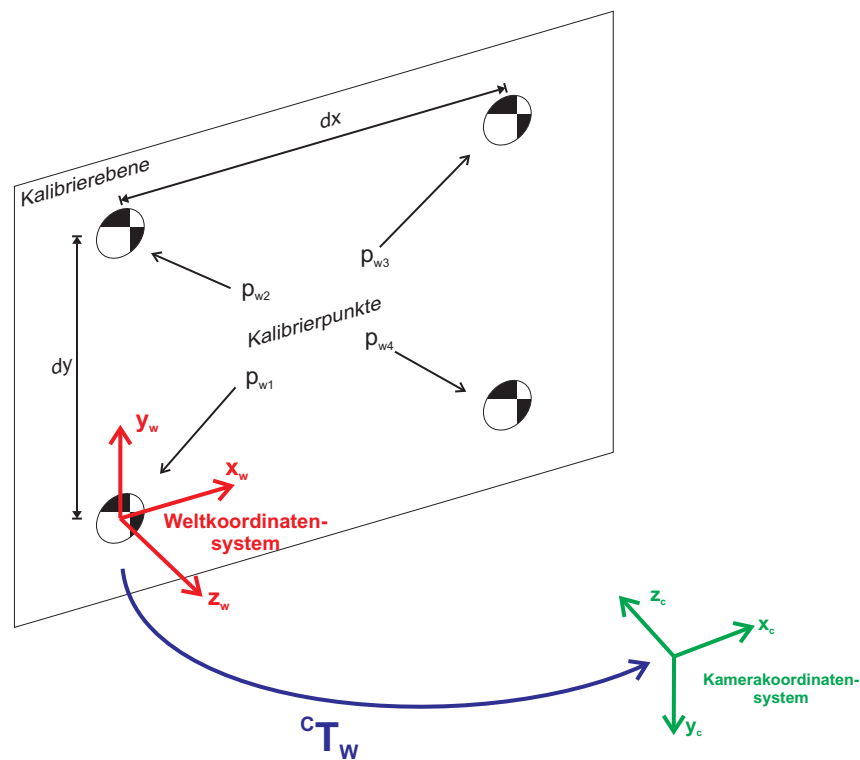


Abbildung 6.5: Kalibrierebene mit Weltkoordinatensystem und Kamerakoordinatensystem.

Die Kalibrierpunkte befinden sich auf der x-y-Ebene im Weltkoordinatensystem und haben somit die Weltkoordinaten

$$\begin{aligned}\vec{p}_{w1} &= [0; 0; 0] \\ \vec{p}_{w2} &= [0; dy; 0] \\ \vec{p}_{w3} &= [dx; dy; 0] \\ \vec{p}_{w4} &= [dx; 0; 0].\end{aligned}$$

Für eine erfolgreiche Kalibrierung müssen diese Abstände ebenso wie interne Kameramaße (Brennweite, optisches Zentrum, Verzerrungsparameter der Linse) gegeben sein. Die internen Maße (auch intrinsische Parameter genannt), wurden vorher ermittelt und sind somit bekannt. Im Rahmen dieses Versuchs müssen somit lediglich die Abstände der Kalibrierpunkte vermessen und in *irpInitData* eingetragen werden (anschließend *irpInitData* ausführen, damit die Werte in den Speicher übernommen werden).

Mittels der homogenen Transformationsmatrix ${}^C T_W$ lassen sich Punkte aus dem Weltkoordinatensystem in das Kamerakoordinatensystem transformieren. Dies bedeutet, dass ihre Position bezüglich des Ursprungs des Kamerakoordinatensystems angegeben wird. Die Umrechnung erfolgt über

$$\vec{x}_c = {}^C T_W \cdot \vec{x}_w. \quad (6.1)$$

Um Punkte auf der Kalibrierebene in Kamerakoordinaten ausdrücken zu können, wird somit diese ${}^C T_W$ -Matrix benötigt. Die Ermittlung dieser Matrix wird auch extrinsische Kamerakalibrierung genannt. In Matlab geschieht dies mit der Skript-Datei *irpCalibrateExtrinsic*. Hier werden sowohl die ${}^C T_W$ -Matrizen für zwei verschiedene Positionen der Kalibrierebene ermittelt, als auch auf die Kalibrierebenen projizierte Streifenmuster mittels der Kamera aufgenommen.

Da sowohl Kamera als auch Projektor nach der Kalibrierung nicht mehr bewegt werden dürfen ist es sinnvoll, den Kalibrierablauf vorher zu planen. Dafür sollte der Projektor aktiviert und das Kamerabild mittels *irpCaptureImage* angezeigt werden. Eine gute gegenseitige Positionierung von Kamera, Projektor und Kalibrierebenen zeichnet sich durch folgende Punkte aus:

- a) Alle vier Kalibrierpunkte auf der Kalibrierebene sind für beide Kalibrierpositionen im Kamerabild sichtbar. Dies ist eine NOTWENDIGE Voraussetzung für eine erfolgreiche Kalibrierung.
- b) Die zwei Positionen der Kalibrierebene sollten nicht zu nahe beieinander liegen, um eine robuste Kalibrierung zu ermöglichen. Außerdem sollten sie möglichst nicht parallel sein.
- c) Das Projektorbild muss natürlich auf die Kalibrierebenen treffen und sollte diese möglichst weit ausfüllen, jedoch auch nicht zu stark „überstehen“ (wobei sich letzteres nicht immer vermeiden lässt).
- d) Je mehr das Projektorbild das Kamerabild ausfüllt, desto weniger Platz im Bild wird „verschenkt“ und desto höher ist die Genauigkeit der Kalibrierung.

Sind diese Punkte berücksichtigt, kann das Kamerafenster geschlossen, und das Skript *irpCalibrateExtrinsic* gestartet werden (vorher nicht vergessen, die Abstände der Kalibrierpunkte in *irpInitData* einzutragen). Es wird zuerst erneut das Kamerabild angezeigt, sodass die Kalibrierebene in die erste gewünschte Position gebracht werden kann. Mittels

CAPTURE muss nun ein Bild gespeichert werden. Nach dem Schließen des Capture-Fensters wird das aufgenommene Bild angezeigt. Die vier Kalibrierpunkte müssen nun in der Reihenfolge ihrer Nummerierung (siehe Bild 6.5) mittels der Maus angeklickt werden. Aus diesen Positionen und den vorher definierten Abständen wird die ${}^C T_W$ -Matrix der ersten Kalibrierebene bestimmt und als *CTW1* gespeichert. Anschließend wird das Streifenmuster auf die Ebene projiziert und die aufgenommenen Bilder zur späteren Verarbeitung als Variable *striped_calib1* gespeichert.

WICHTIG: Vor der Projektion der Streifen unbedingt den Projektor auf die Kalibrierebene scharf stellen. Dies geschieht durch Drehen der Linse des Projektors.

Das Kalibrierungsverfahren startet nun erneut, um die zweite Position der Kalibrierebene zu bestimmen. Der Ablauf ist derselbe wie zuvor, die Transformationsmatrix wird als *CTW2*, die Streifenbilder als *striped_calib2* gespeichert. Auch hier gilt: Vor der Streifenprojektion den Projektor scharf stellen!

Auf einzelne Streifenbilder kann übrigens über Indizes in geschweiften Klammern zugegriffen werden (z.B. *striped_calib1*{7}). Damit ist die extrinsische Kalibrierung abgeschlossen.

6.1.2.2 Die Kalibrierebenen

Nachdem die extrinsische Kalibrierung durchgeführt wurde, liegen sowohl die beiden Transformationsmatrizen *CTW1* und *CTW2* als auch die entsprechenden Streifenbilder *striped_calib1* und *striped_calib2* vor. Im weiteren Verlauf der Systemkalibrierung wird es nötig sein, 3D-Punkte auf den Kalibrierebenen zu ermitteln. Daher müssen die Ebenengleichungen analytisch beschrieben werden. Hierzu dient die Darstellung in der Normalenform:

$$\vec{n} \cdot \vec{x} = d \quad (6.2)$$

wobei \vec{n} der Normalenvektor, \vec{x} ein beliebiger Punkt der Ebene und d der Abstand der Ebene zum Ursprung des Koordinatensystems ist.

Die nächste Aufgabe dieses Versuchs besteht nun darin, die Funktion

$$[\ n \ d \] = \textit{irpGetCalibrationPlane} \ (\ CTW \)$$

zu vervollständigen, sodass dort der Normalenvektor und der Abstand der Kalibrierebene berechnet werden. Gegeben ist lediglich die ${}^C T_W$ -Matrix und die Information, dass die gesuchte Ebene die xy-Ebene des Weltkoordinatensystems ist. Die Funktion

$$\textit{irpTestGetCalibrationPlane}$$

kann genutzt werden, um die Korrektheit der Programmierung zu überprüfen.

6.1.2.3 Ebenengleichung aus drei Punkten

Eine weitere zu programmierende Funktion ist

$$[n \ d] = \textit{irpGetPlaneEquation} (p1, p2, p3)$$

in der die Bestimmung von \vec{n} und d einer Ebenengleichung nach (6.2) mittels dreier auf dieser Ebene liegenden Punkten durchgeführt werden soll. Auch hier gibt es eine Testfunktion zur Überprüfung mit dem Namen

irpTestGetPlaneEquation.

6.1.2.4 Schnitt aus Gerade und Ebene

Bevor schließlich die Projektorebenen bestimmt werden können, muss vorerst eine weitere Funktion programmiert werden. Deren Aufgabe wird es sein, die Kamerasehstrahlen mit einer beliebigen Ebene zu schneiden. Die Funktion wird folgendermaßen aufgerufen:

$$[p] = \textit{irpIntersectRayAndPlane} (\textit{pixel_pos}, n, d, f, s, cc)$$

Übergeben werden ihr die Position des gewünschten Pixels ($\textit{pixel_pos} = [x \ y]$), sowie der Normalenvektor n und der Abstand d einer Ebene. Ermittelt werden soll nun der Schnittpunkt des entsprechenden Pixelsehstrahls mit der Ebene.

Zur Erinnerung: Eine Gerade lässt sich im Dreidimensionalen durch die Parameterform

$$\vec{x} = \vec{x}_a + \lambda \cdot \vec{x}_r \tag{6.3}$$

darstellen, wobei im hiesigen Fall sämtliche Geraden vom Ursprung des Koordinatensystems ausgehen. Somit gilt für den Aufpunkt $\vec{x}_a = [0 \ 0 \ 0]^T$. Getestet werden kann die Funktion mittels

irpTestIntersectRayAndPlane.

6.1.2.5 Die Projektorebenen

Wie in der Einleitung des Kapitels erwähnt, haben die projizierten Streifenmuster die Funktion, jeder Projektorebene eine eindeutige Hell-Dunkel-Abfolge zuzuweisen. Diese Hell-Dunkel-Abfolge ergibt einen sogenannten Gray-Code. Wenn zwei Pixel denselben Gray-Code aufweisen, gehören sie zur selben Projektorebene. Dabei ist es irrelevant, ob diese Pixel im selben Bild liegen, oder in zwei verschiedenen Bildern, resultierend aus zwei Scanvorgängen.

Die Ermittlung der Gray-Codes geschieht mittels Binarisierung und anschließender Kombination der Streifenbilder eines einzelnen Scanvorgangs. In Matlab ist hierfür die bereits fertig programmierte Funktion

$[grayMask] = irpGetGrayMask (striped)$

zuständig. Auch wenn die Masken später innerhalb der Skripte automatisch generiert werden, soll hier nun testweise eine Gray-Code-Maske erstellt werden. Das Ergebnis muss qualitativ Bild 6.6 entsprechen.

Hinweis: Die Streifenmuster können Sie mit Hilfe des Skriptes *irpGetStripeImages* erzeugen.

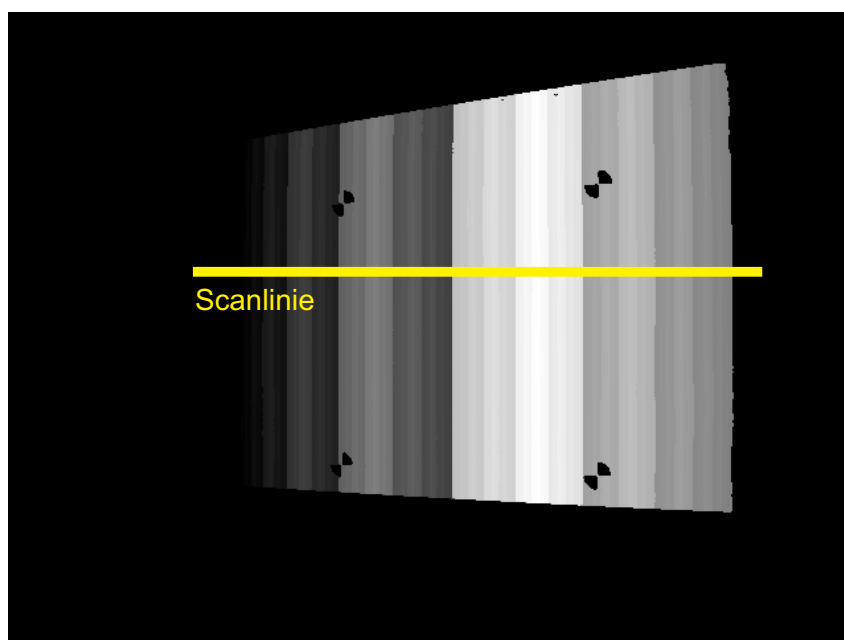


Abbildung 6.6: Gray-Code-Maske einer Kalibrierebene mit eingezeichneter Scanlinie für die 3D-Punkt-Extraktion.

In diesen Gray-Code-Masken hat jeder Pixel den Wert des dort detektierten Gray-Codes. Alle Pixel, für die kein Gray-Code detektiert wurde, behalten den Wert 0. Aus den Gray-Code-Masken sollen nun für jede Projektorebene drei 3D-Punkte ermittelt werden, die später dazu dienen, die Ebenengleichungen aufzustellen. Dafür müssen zuerst aus den beiden Gray-Code-Masken insgesamt drei Profillinien herausgesucht werden. Diese Scanlinien sollten dort liegen, wo möglichst viele Streifen möglichst gut erkannt wurden.

Punkte auf diesen drei Scanlinien, die denselben Gray-Code aufweisen, gehören zum selben „Lichtbereich“ und repräsentieren damit dieselbe Projektorebene. Auf den Scanlinien werden die 3D-Positionen der Pixel mittels der im letzten Abschnitt programmierten Funktion *irpGetIntersectRayAndPlane* ermittelt. Für jeden einzelnen Lichtbereich werden die Positionen der gefundenen 3D-Punkte gemittelt, sodass jeder Projektorebene genau eine 3D-Koordinate pro Scanlinie zugeordnet werden kann. Aus den jeweiligen drei 3D-

Koordinaten lassen sich schließlich die Projektorebenengleichungen über die ebenfalls bereits programmierte Funktion *irpGetPlaneEquation* generieren. In Bild 6.7 ist der Aufbau skizziert. Zur Berechnung der Projektorebenen dient die Skript-Datei

irpGetProjectorPlanes.

Da sämtliche benötigten Funktionen bereits in den letzten Abschnitten programmiert wurden, verbleibt hier lediglich die Auswahl der Position der drei Scanlinien. Nach dem Aufruf der Skriptdatei wird daher zuerst zweimal die Gray-Code-Maske der ersten Kalibrierebene und anschließend die Gray-Code-Maske der zweiten Kalibrierebene angezeigt. Hier ist jedesmal eine Scanlinie festzulegen, entsprechend dem Beispiel in Bild 6.7. Die Auswahl geschieht mittels der Maus, wobei der Startpunkt der Scanlinie mit einem Linksklick und der Endpunkt mit einem Rechtsklick gesetzt werden muss.

Die endgültige Berechnung der Projektorebenen erfolgt anschließend in *irpGetProjectorPlanes* mittels der gegebenen Funktion *irpCalculateNormalsAndDs* (innerhalb derer auch das vorher programmierte *irpGetPlaneEquation* ausgeführt wird). *irpCalculateNormalsAndDs* generiert zwei Lookup-Tabellen, deren Reihenummer dem Gray-Code entspricht, zu dem die jeweiligen Ebenennormalen (*PlaneNormals*) und Distanzen zum Ursprung (*PlaneDs*) gehören.

Mit der Ausführung dieses Skripts ist die Kalibrierung des Systems abgeschlossen. Zur Kontrolle kann mittels der Funktion

$$[] = \text{irpVisualizeScanSetup} (CTW1, CTW2, calib_dist, PlaneNormals, \\ PlaneDs)$$

der Aufbau (Kamera, Kalibrierebenen, Projektorebenen) visualisiert werden.

6.2 Scannen von Objekten

Nachdem im vorherigen Abschnitt die Kalibrierung durchgeführt wurde, ist es nun das Ziel, beliebige 3D-Objekte einzuscannen. Dafür ist lediglich noch eine Funktion zu programmieren.

Für jede Projektorebene ist die Normalengleichung bekannt. Wenn nun die Streifenmuster auf ein beliebiges Objekt projiziert werden, so kann für jeden Pixel, für den ein Gray-Code identifiziert wurde, die zugehörige Projektorebene mittels der Variablen *PlaneNormals* und *PlaneDs* generiert werden. Dies soll in der Funktion

$$[DistanceMap \ Points3D] = \text{irpGetObject3D} (GrayMaskObj, PlaneNormals, \\ PlaneDs, f, s, cc)$$

geschehen. Ihr werden u.a. die Gray-Code-Maske des gescannten Objekts und die Projektorebeneninformationen übergeben. Berechnet werden sollen die 3D-Koordinaten der Punkte, sowie eine Distanzkarte. Die folgende Vorgehensweise ist für jeden Pixel der Gray-Code-Maske durchzuführen, dessen Wert nicht 0 ist:

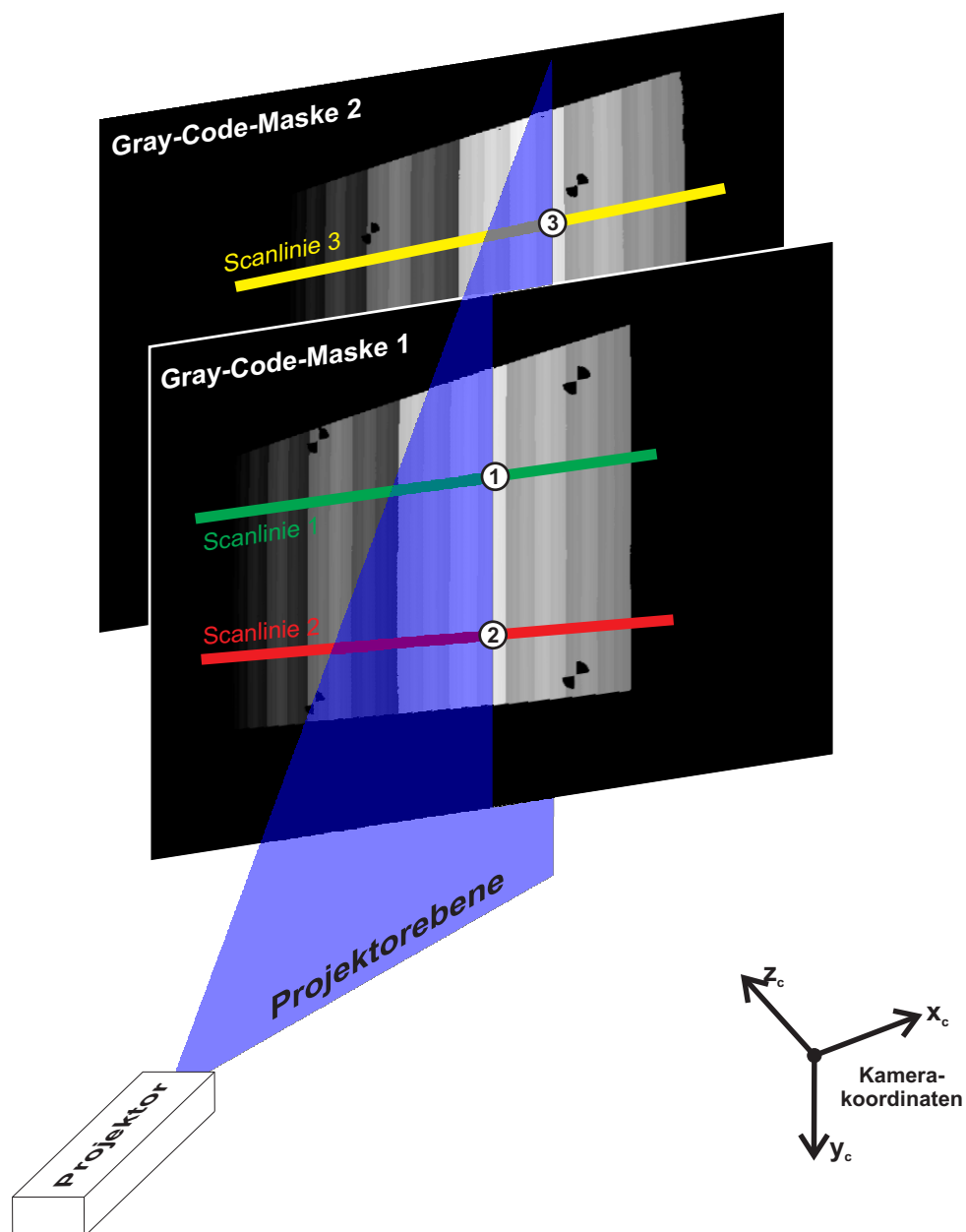


Abbildung 6.7: Gray-Code-Masken von zwei Kalibrierebenen mit Scanlinien und Projektorebene.

- a) Finden der zum Gray-Code gehörigen Ebenennormale und Abstand zum Ursprung.
- b) Berechnen des Richtungsvektors des zum Pixel gehörenden Sehstrahls.
- c) Schneiden der Projektorebene mit dem Sehstrahl.
- d) Speichern des 3D-Punkts in der Variable *Points3D* an den entsprechenden Pixelkoordinaten.
- e) Speichern der z-Koordinate des Schnittpunkts in der Variable *DistanceMap* an der entsprechenden Stelle.

Ist *irpGetObject3D* programmiert, kann der Scanvorgang eines beliebigen Objekts über die Skript-Datei

irpScanObject

ausgeführt werden. Auch hier muss der Projektor vorher auf das Objekt scharf gestellt werden. In *irpScanObject* werden sowohl die Streifenbilder aufgenommen, als auch die Gray-Code-Maske des Objekts erstellt und schließlich mittels *irpGetObject3D* die 3D-Punkte (*Points3D*) und die Distanzkarte (*DistanceMap*) bestimmt.

Eine Visualisierung der Ergebnisse (Testaufbau mit 3D-Punkten) erfolgt über *irpVisualizeScanSetup*. Die Distanzkarte vermittelt einen zweidimensionalen Eindruck der 3D-Daten. Anzeige über *imshow(DistanceMap,[min max])*, wobei der Grauwertbereich auf jeden Fall über *min* und *max* angeglichen werden sollte. Ein Beispiel zeigt Bild 6.8.

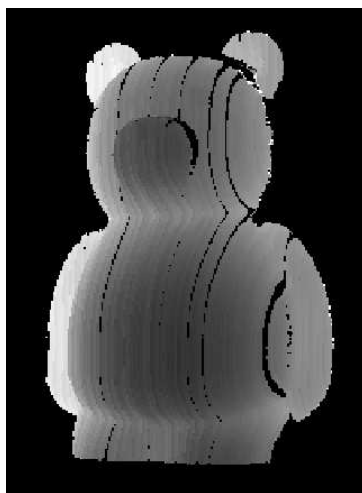


Abbildung 6.8: Distanzkarte für ein gescanntes Objekt.

Es gibt außerdem die Möglichkeit, die Daten als „echtes“ 3D-Modell darzustellen. Wie die Distanzkarte erraten lässt, kommt es an den Rändern des Objekts jedoch häufig zu

groben Fehlinterpretationen. Dies führt zu starken Ausreißern in den 3D-Daten. Dass der Farbverlauf in der Distanzkarte nicht glatt ist, deutet darüber hinaus darauf hin, dass auch auf der Oberfläche selbst der Scanvorgang fehlerbehaftet ist. Dies führt zu Artefakten auf der 3D-Oberfläche. Diese Probleme wurden in Bild 6.9 mittels des externen 3D-Programms MeshLab visualisiert.

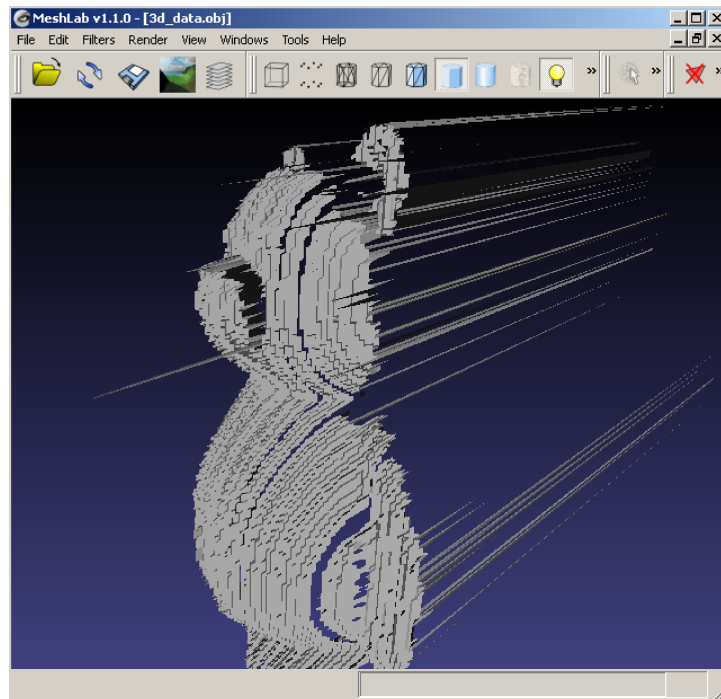


Abbildung 6.9: Fehler während des Scanvorgangs führen zu Ausreißern und Treppeneffekten.

Daher ist es zweckmäßig, das Scanergebnis zu filtern, um eine glattere Darstellung zu erhalten. Am einfachsten ist dies mit der Distanzkarte, da hier bekannte 2D-Filteroperationen (Median, Averaging) angewendet werden können. Mit den eigentlichen 3D-Daten zu rechnen wäre wesentlich komplizierter und rechenaufwändiger. Die Filterung kann mit der Funktion

$$[\textit{FilteredMap}] = \textit{irpFilterDistanceMap} (\textit{DistanceMap})$$

durchgeführt werden. Ihr muss lediglich die Distanzkarte übergeben werden. Evtl. wurden während des Scanvorgangs auch Objekte des Hintergrunds erkannt. Wenn nur das Scanobjekt selbst von Interesse ist, müssen diese entfernt werden, bevor die Filterung durchgeführt wird. In diesem Versuch ist dies jedoch nicht zwingend erforderlich. Nach der Filterung stellt sich die Distanzkarte wie in Bild 6.10 dar.



Abbildung 6.10: Distanzkarte nach der Filterung.

Nach dem Filtervorgang können aus der Distanzkarte wieder 3D-Punkte mit x-, y- und z-Koordinaten generiert werden. Dazu dient das Programm

$$[\textit{Points3Dnew}] = \textit{irpDistanceMapTo3D} (\textit{FilteredMap})$$

Um die 3D-Punkte visualisieren zu können, müssen diese mittels

$$[\textit{DataOBJ}] = \textit{irpCreateOBJ} (\textit{Points3Dnew})$$

in ein Dreiecksnetz (OBJ-Format) umgewandelt werden. Innerhalb von Matlab kann dieses nun über

$$\textit{irpVisualizeOBJ}(\textit{DataOBJ})$$

angezeigt werden. Mit

$$\textit{irpCreateOBJ}(\textit{Points3Dnew}, 'scanned.obj')$$

können die Daten auch in einer Datei abgespeichert und anschließend mit dem Programm MeshLab geladen werden. Der Vorteil von MeshLab ist, dass es wesentlich intuitiver als die Matlab-Anzeige zu bedienen ist. Darüber hinaus ist es als Freeware erhältlich.

6.3 Zusammenfassung der Aufgabenstellung

Da sich dieser Versuch aus vielen kleinen Teilaufgaben zusammensetzt, folgt zum Abschluss eine Auflistung der durchzuführenden Arbeitsschritte. Da im Versuchslabor leider nur ein PC mit Beamer ausgestattet ist, und dieser nur jeweils von einer Gruppe genutzt

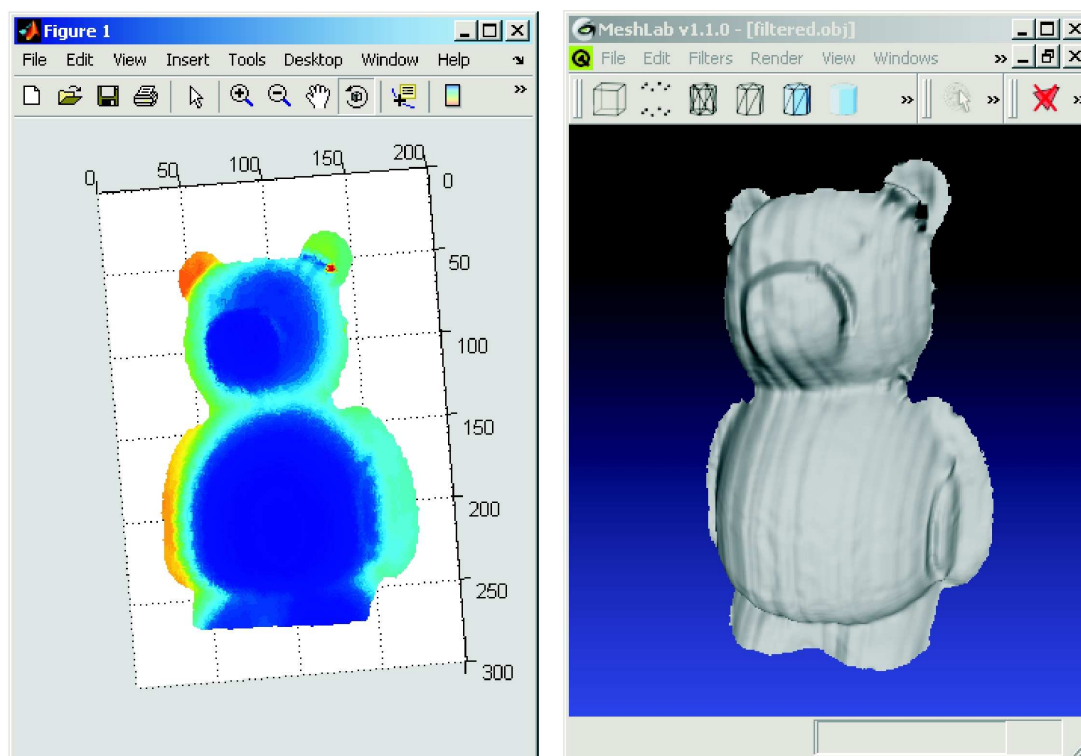


Abbildung 6.11: Ansicht des Dreiecksnetzes des gescannten Objekts in Matlab (links) und MeshLab (rechts).

werden kann, sind die Schritte hier nicht chronologisch aufgelistet. Sie sind vielmehr derart angeordnet, dass zuerst sämtliche Programmieraufgaben am eigenen PC gelöst werden müssen, um dann anschließend Kalibrierung und Scanvorgang am Beamer-PC durchführen zu können.

- a) Programmierung von *irpPixelToRay*, welches Kamerasehstrahlen aus Pixelkoordinaten berechnet. Überprüfung der Funktion über *irpTestPixelToRay*.
- b) Programmierung von *irpGetCalibrationPlane* zur Ermittlung der Ebenengleichungen der Kalibrierebenen und Test der Funktion mit *irpTestGetCalibrationPlane*.
- c) Programmierung von *irpGetPlaneEquation* zur Ermittlung einer Ebenengleichung aus 3 gegebenen Punkten und Test mit *irpTestGetPlaneEquation*.
- d) Programmierung von *irpIntersectRayAndPlane* zur Berechnung des Schnittpunkts eines Kamerasehstrahls mit einer Ebene. Test der Funktion mit *irpTestIntersectRayAndPlane*.

- e) Programmierung von *irpGetObject3D* zur Bestimmung von 3D-Oberflächenpunkten eines beliebigen Objekts.
- f) Vermessung der Kalibrierpunkte auf den Kalibrierebenen und Eintrag in *irpInitData*.
- g) Extrinsische Kalibrierung der Kalibrierebenen mit *irpCalibrateExtrinsic* (Ermittlung der CTW-Matrizen). Vor der Streifenprojektion den Projektor immer auf die jeweilige Kalibrierebene scharf stellen.
- h) Testweise Erstellung einer Gray-Code-Maske.
- i) Wahl der Scanlinien in den Gray-Code-Masken der Kalibrierebenen und Berechnung der Projektorebenen durch Ausführen von *irpGetProjectorPlanes*.
- j) Visualisierung des Versuchsaufbaus mittels *irpVisualizeScanSetup*.
- k) Scannen eines Objects mit *irpScanObject* (Projektor scharf stellen).
- l) Visualisierung des Versuchsaufbaus und der Distanzkarte.
- m) Filterung der Distanzkarte mit *irpFilterDistanceMap*.
- n) Generierung „neuer“ 3D-Daten aus der gefilterten Distanzkarte mit *irpDistanceMapTo3D*.
- o) Visualisierung der gefilterten 3D-Daten in Matlab und/oder MeshLab.

Kapitel 7

Versuch V: RANSAC

7.1 Einleitung

In diesem Versuch sollen 3d Szenen, die in Form von Punktwolken gegeben sind, mit Hilfe von RANSAC-Verfahren interpretiert werden. Wir setzen dabei voraus, dass die Szenen in diesem Versuch nur aus Kugeloberflächen und Ebenen bestehen (siehe z.B. Abbildung 7.1). Nähme man noch Zylinder hinzu, könnten schon sehr gut viele Szenen aus der Praxis beschrieben werden. Diese Szenen könnten beispielsweise aus Tiefendaten bestehen, die ein mobiler Roboter mit Hilfe eines Laserscanners aufnimmt während er durch ein Gebäude fährt, oder die mit Hilfe eines kodierte Lichtansatzes von Bauteilen auf dem Fließband einer Fertigungsanlage aufgenommen werden. Mit Interpretation ist in diesem Versuch konkret gemeint, welche Punkte der Szene zu einer gemeinsamen Ebene oder Kugeloberfläche zusammengefasst werden können. Es geht also darum, Punkte zu „sinnvollen“ Regionen zusammenzufassen, die z.B. ein mobiler Roboter einfacher, schneller und robuster verarbeiten könnte, um Kollisionen zu vermeiden oder sich selbst in der Szene zu lokalisieren.

7.2 Verfahren

Das von Ihnen zu implementierende Verfahren läuft nach folgendem Schema ab:

1. Führe auf einer gegebenen Punktwolke *Points* einen RANSAC-Algorithmus für Kugeln durch mit maximal *iter* Iterationsschritten. Speichere die Kugelparameter *s* und Kontaktpunkte K_s , die zur besten Lösung gehören.
2. Führe auf einer gegebenen Punktwolke *Points* einen RANSAC-Algorithmus für Ebenen durch mit maximal *iter* Iterationsschritten. Speichere die Ebenenparameter *p* und Kontaktpunkt K_p , die zur besten Lösung gehören.

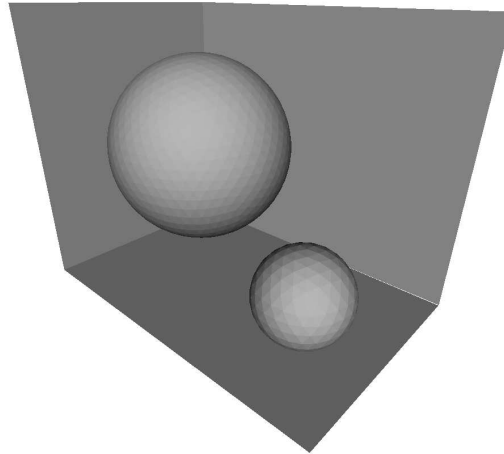


Abbildung 7.1: Beispiel für eine Szene bestehend aus Kugeln und Ebenen.

3. Falls die Anzahl der Kontaktpunkte sowohl bei K_s als auch bei K_p kleiner ist als ein gefordertes Minimum, dann springe zu Schritt 7.
4. Vergleiche, ob die geschätzte Kugel oder die geschätzte Ebene die meisten Kontaktpunkte liefert. Die Parameter der besseren Schätzung werden abgespeichert und K entsprechend auf K_s oder K_p gesetzt.
5. Entferne die Kontaktpunkte K aus *Points*.
6. Springe mit der aktualisierten Punktwolke *Points* zurück zu Schritt 1.
7. Berechne für jeden Punkt aus der ursprünglichen Punktwolke ein Label, das ihn eindeutig zu einer Region (repräsentiert durch eine Ebene oder Kugel) zuordnet.
8. Stelle die Lösung dar.

Die Parameter der Kugeln und Ebenen sollen dabei folgendermaßen definiert werden:

- Eine Kugel wird beschrieben durch die vier Parameter (cx, cy, cz, r) . Der Mittelpunkt der Kugel ist gegeben durch (cx, cy, cz) und r gibt den Radius an.
- Zur Beschreibung einer Ebene wurde die Form (nx, ny, nz, d) gewählt. Dabei ist $\vec{n} = (nx, ny, nz)$ der Einheitsnormalenvektor der Ebene und d der Abstand der Ebene zum Ursprung. Jeder Punkt q auf der Ebene erfüllt somit die Bedingung $\langle \vec{n}, \vec{q} \rangle = d$.

7.3 Aufgabestellung

7.3.1 Erstellen einer Testszene

Zum Erstellen einer Testszene steht Ihnen die Funktion

$$[\textit{Points}, \textit{SOut}, \textit{POut}] = \textit{irpGenScene} (S, P, B, f)$$

bereit. Mittels S können Sie mehrere Kugeln und mittels P mehrere Ebenen vorgeben. Für jede Kugel und Ebene wird zusätzlich angegeben, wieviele Zufallspunkte jeweils maximal erzeugt werden sollen. Beschränkt wird der Szenenraum durch die Bounding Box B . Um eine eingeschränkte Sicht des Sensors zu simulieren können Sie mit f einen Brennpunkt vorgeben, in dem sich die Sehstrahlen des simulierten Sensors treffen. Als Rückgabewerte erhalten Sie eine Punktwolke \textit{Points} und eine Kopie der von Ihnen übergebenen Kugel- und Ebenenparameter in \textit{SOut} und \textit{POut} , wobei jeweils die gewünschte Anzahl von Punkten auf die tatsächliche Anzahl von Punkten gesetzt wurde.

Um die simulierte Szene realistischer zu machen, haben Sie die Möglichkeit mittels der Funktion

$$[\textit{PointsOut}] = \textit{irpAddNoise} (\textit{Points}, \textit{noise})$$

Rauschen hinzuzufügen. Dabei wird für jeden Punkt q aus \textit{Points} mit Hilfe des Matlab Ausdrucks $\textit{randn}(3,1) * \textit{noise}$ ein normalverteilter Rauschvektor erzeugt und auf q addiert. Mit Hilfe der Funktion

$$[] = \textit{irpPlotScene} (\textit{Points}, \textit{Labels})$$

können Sie die Szene darstellen. Initial kann für jeden Punkt in \textit{Points} das Label 0 vergeben werden.

- a) Erstellen Sie verschiedene Testszenen und schauen Sie sich diese mittels $\textit{irpPlotScene}$ an.

Hinweis: Mehrere beispielhafte Parametrisierungen zur Nutzung mit $\textit{irpGenScene}$ finden Sie in dem Skript $\textit{irpVersuch5}$.

- b) Setzen Sie das ϵ für die Kontaktumgebung in Abhängigkeit der von Ihnen gewählten Rauschstärke. Auch wenn Sie die Punkte nicht verrauschen, achten Sie dennoch darauf, dass Sie $\epsilon > 0$ wählen.

Neue Funktionen: $\textit{irpGenScene}$, $\textit{irpAddNoise}$, $\textit{irpPlotScene}$, $\textit{irpVersuch5}$, \textit{normr} , \textit{randn}

7.3.2 RANSAC für Ebenen

Zunächst soll mit Hilfe eines RANSAC-Verfahrens eine möglichst optimale Ebene für die gegebene Punktmenge gefunden werden. Optimal heißt hier, dass die Anzahl der Kontaktpunkte, also die Anzahl der Punkte innerhalb einer Epsilonumgebung der Ebene, maximal ist.

- a) Schreiben Sie eine Funktion

$$[Q] = \text{irpRandPoints} (Points, num) ,$$

die zufällig aus der Punktmenge *Points* insgesamt *num* Punkte auswählt. Als Zufallsfunktion können Sie *rand* verwenden. for-Schleifen können hier vermieden werden.

- b) Um eine Ebene eindeutig zu beschreiben, sind drei verschiedene Punkte notwendig. Aus diesen drei Punkten sollen Sie in der Funktion

$$[p] = \text{irpCalcPlane} (Q)$$

die Parameter (nx, ny, nz, d) einer Ebene berechnen. Testen Sie *irpCalcPlane* mit Hilfe von Punkten aus der Menge, die in Testszene 1 erstellt wird.

- c) Bestimmen Sie die Kontaktpunkte in der Funktion

$$[C] = \text{irpPointsOnPlane} (Points, p, eps)$$

Diese Funktion gibt ein Spaltenvektor mit Indizes der Punkte zurück, die innerhalb der Epsilon-Umgebung liegen. Es ist hier möglich ohne for-Schleifen auszukommen. Zum Testen können Sie z.B. auch Testszene 1 verwenden.

- d) In der Funktion

$$[p \ num \ C] = \text{irpBestPlane} (Points, iter, eps)$$

sollen Sie nun iterativ die beste Lösung finden. Zum Testen ist beispielsweise Testszene 2 gut geeignet.

Neue Funktionen: *irpRandPoints*, *irpCalcPlane*, *irpPointsOnPlane*, *irpBestPlane*, *rand*

7.3.3 RANSAC für Kugeln

Als nächstes sollen Sie mit Hilfe eines RANSAC-Verfahrens für eine gegebene Punktmenge eine optimale Kugel finden, so dass die Anzahl der Kontaktpunkte maximal wird.

- a) Eine Kugel ist eindeutig durch vier Punkte gegeben unter der Annahme, dass die vier Punkte nicht in einer gemeinsamen Ebene liegen. Um den Mittelpunkt der Kugel zu bestimmen, können Sie folgendermaßen vorgehen:
- Konstruieren Sie aus den vier Punkten zwei verschiedene Dreiecke.
 - Berechnen Sie für jedes Dreieck den Mittelpunkt des Umkreises und die Achse, die durch den Mittelpunkt geht und senkrecht auf dem Dreieck steht.
 - Bringen Sie die beiden Achsen zum Schnitt, um den Mittelpunkt der Kugel zu erhalten.

Auf der Internetseite http://en.wikipedia.org/wiki/Circumscribed_circle finden Sie beispielsweise Informationen darüber, wie Sie den Mittelpunkt des Umkreises berechnen können. Implementieren Sie dieses Verfahren oder ein anderes Verfahren Ihrer Wahl in der Funktion

$$[a \ n] = \text{irpCalcCenterAxis} (q1, q2, q3) .$$

Testen Sie diese Funktion z.B. mit dem Dreieck $((0, 0, 0), (2, 0, 0), (0, 1, 0))$.

- b) Aufgrund numerischer Ungenauigkeiten kann es passieren, dass die beiden Achsen sich nicht schneiden. Deshalb sollten Sie etwas toleranter sein, indem Sie den Punkt \vec{c} bestimmen, der zu den beiden windschiefen Geraden eine möglichst geringe Distanz hat. Siehe hierzu auch Abbildung 7.2. \vec{a} und \vec{b} sind dabei jeweils die Aufvektoren der Geraden und \vec{v} sowie \vec{w} entsprechend die Richtungsvektoren. \vec{n} ist das Kreuzprodukt zwischen \vec{v} und \vec{w} . Damit erhält man folgende Gleichungen für \vec{c} :

$$\vec{c} = \vec{a} + h_1 \vec{v} + 0.5 h_3 \vec{n} \quad (7.1)$$

$$= \vec{b} + h_2 \vec{w} - 0.5 h_3 \vec{n} \quad (7.2)$$

Bringen Sie dieses Gleichungssystem in folgende Matrixschreibweise:

$$A \vec{h} = \vec{x} \quad (7.3)$$

$\vec{h} = (h_1, h_2, h_3)^T$ und \vec{x} sind dabei Spaltenvektoren und A eine 3x3 Matrix. Nach einer Matrixinvertierung kann dann leicht der Lösungsvektor \vec{h} ausgerechnet werden. Implementieren Sie dies in der Funktion

$$[s] = \text{irpCalcSphere} (Q) ,$$

die vier 3d Punkte übernimmt und daraus die Parameter (Mittelpunkt und Radius) für eine Kugel berechnet. Achten Sie darauf, dass die Matrix A nicht in allen Fällen invertierbar ist! Testen Sie Ihre Funktion z.B. mit Hilfe der Testszene 3.

- c) Zur Bewertung einer Kugelhypothese sollen in der Funktion

$$[C] = \text{irpPointsOnSphere} (Points, s, eps)$$

die Indizes der Kontaktpunkte berechnet werden. Auch hier lassen sich mit etwas Geschick die for-Schleifen vermeiden.

- d) In der Funktion

$$[s \text{ num } C] = \text{irpBestSphere} (Points, iter, eps)$$

sollen Sie nun iterativ die beste Lösung finden. Zum Testen ist beispielsweise Testszene 4 gut geeignet.

- e) Bei leicht verrauschten Punktwolken kann es passieren, dass statt einer Ebene eine Kugel mit sehr großem Radius gefunden wird. Um dies zu vermeiden können Sie einfach einen Schwellwert einführen, der Kugeln mit einem Radius $r > 50$ verwirft.

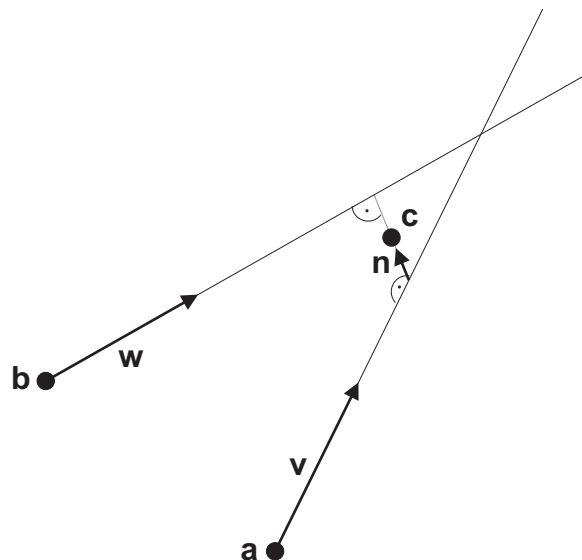


Abbildung 7.2: Diagramm zum Mittelpunkt der kürzesten Strecke zwischen zwei windschiefen Geraden.

Neue Funktionen: *irpCalcCenterAxis*, *irpCalcSphere*, *irpPointsOnSphere*, *irpBestSphere*

7.3.4 Interpretation der Szene

- a) Es stehen jetzt alle Hilfsmittel bereit, um das in Kapitel 7.2 skizzierte Verfahren in der Funktion

$$[S \ P \ L] = \text{irpInterpret} (\text{Points}, \text{iter}, \text{eps}, \text{minNum})$$

zu implementieren. Übergeben werden hier die Punktwolke *Points*, die Anzahl *iter* der durchzuführenden Iterationen für jeden einzelnen „RANSAC-Durchlauf“, die Größe *eps* der Kontaktumgebung und die minimal geforderte Anzahl *minNum* an Kontaktpunkten pro Region. Als Ergebniswert erhält man alle geschätzten Kugeln *S* und Ebenen *P* (in der fünften Spalte steht jeweils die Anzahl der Kontaktpunkte) und die Labels *L* der Punkte. Das Label 0 ist reserviert für Punkte, die keiner Geometrie zugeordnet werden konnten.

- b) Ein sehr wichtiger Parameter für die Robustheit des Verfahrens ist die vorgegebene Anzahl *iter* an Iterationen für jeden „RANSAC-Durchlauf“. Wie hoch ist die Wahrscheinlichkeit bei insgesamt *N* Punkten in der Szene, dass eine gültige Ebenen- bzw. Kugelhypothese erstellt werden kann, wenn von den insgesamt *N* Punkten nur *k* Punkte auf die gesuchte Geometrie entfallen? Wie könnten Sie dieses Wissen nutzen, um die Anzahl *iter* an Iterationsschritten einzustellen?
- c) Testen Sie Ihr Verfahren z.B. mit den Testszenen 5 und 6, indem Sie die Variable *S2* bzw. *P2* mit der Variablen *S3* bzw. *P3* vergleichen und die Szene mittels *irpPlotScene* darstellen.
- d) Fügen Sie den Punkten Rauschen hinzu (z.B. *noise* = 0.01) und stellen Sie Ihr *eps* in Abhängigkeit von diesem Rauschen ein. Untersuchen Sie die Robustheit des Verfahrens für unterschiedliche Rauschstärken.

Neue Funktionen: *irpInterpret*

Literaturverzeichnis

- [1] THOMAS SCHRAMM: „Matlab - eine Einführung“ (1999)
<http://www.ti3.tu-harburg.de/~haerter/PraktikumI/schramm.pdf>
- [2] WIKIPEDIA: „Circumscribed circle“ (Version vom 25. März 2011, 20:55 Uhr)
[http://en.wikipedia.org/w/index.php?title=Circumscribed_circle
&oldid=420714021](http://en.wikipedia.org/w/index.php?title=Circumscribed_circle&oldid=420714021)