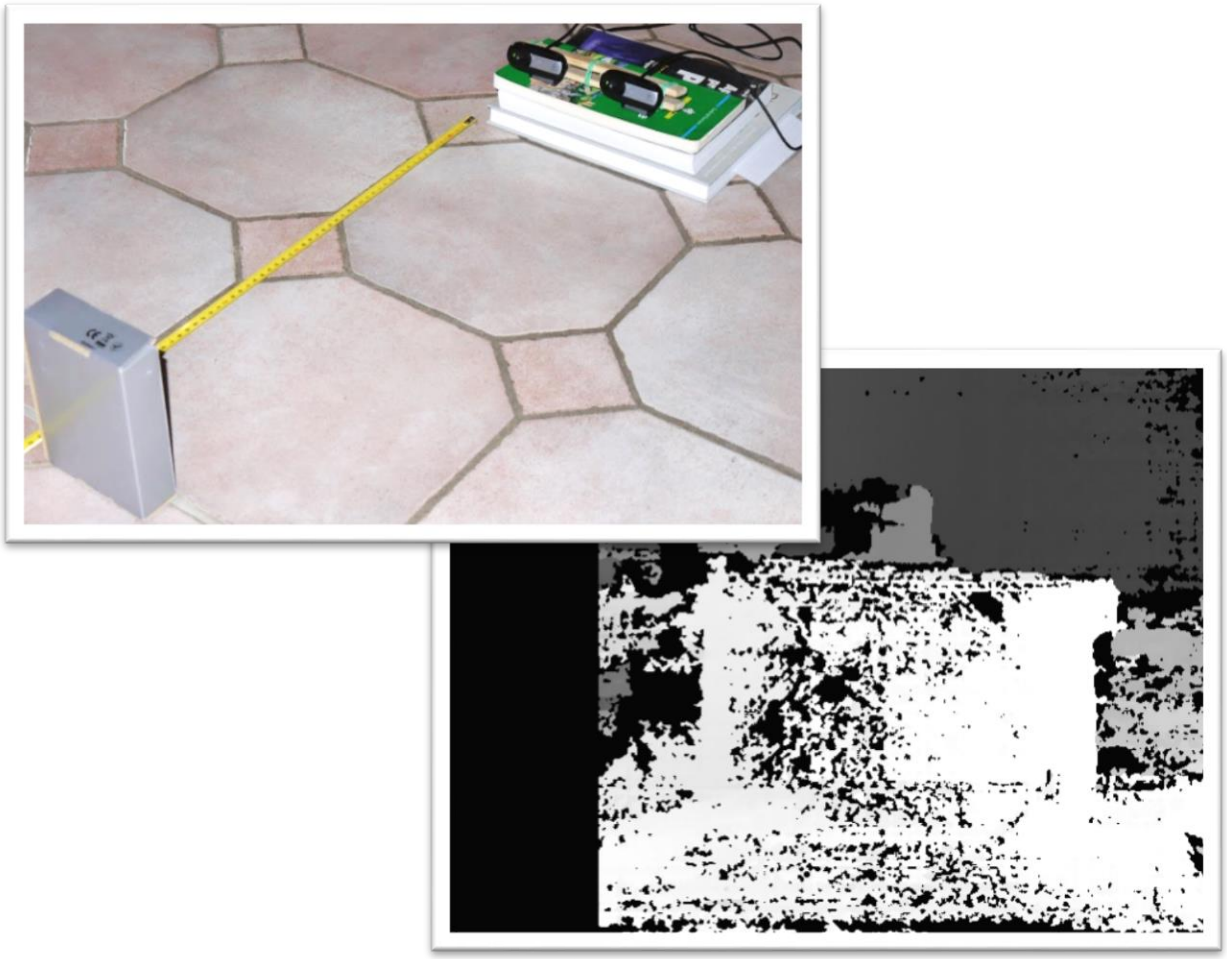
Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Stereo-Vision



Verfasser:

Uhrweiller Frédéric und Vujasinovic Stéphane
Matrikelnr.58566 EIT Matrikelnr.: 59092 Mechatronik

E-mail: uhfr1011@hs-karlsruhe.de E-mail: vust1011@hs-karlsruhe.de

***Betreuer**:* Prof. Dr.-Ing. Ferdinand Olawsky

Projektkoordninator MMT: Prof. Dr. Peter Weber

Projektcode: 17SS_OL_Reinraum

Abgabetermin: 30.09.2017

Inhaltsverzeichnis

Abbildungsverzeichnis

Code Verzeichnis

1. Einleitung

Um die Stereo-Vision zu verstehen, muss man zuerst die Funktionsweise einer einfachen Kamera erläutern, wie sie grundsätzlich aufgebaut ist und welche Parameter geregelt werden müssen.

A. Was ist Stereo-Vision?

Die Stereoskopie ist ein Verfahren das 2 Bildern von derselben Szene aufnimmt um dann eine Disparitätskarte der Szene aufzubauen. Aus dieser Disparitätskarte ist es möglich die Distanz zu einem Objekt zu messen und eine 3D Karte aus der Szene zu erstellen.

B. Stereo Vision im Reinraum-Roboter

Ziel des großen Projekts ist es, ein Reinraum-Roboter zu entwickeln das Partikelmessung im ganzen Reinraum führen kann. Um die Messung problemlos durchzuführen muss sich der Roboter ohne Kollisionen orientieren. Für diese Orientierung haben wir uns entschieden nur Kameras einzusetzen und um den Abstand zu einem Objekt zu messen brauchen wir mindesten zwei Kameras (Stereo Vision). Die zwei Kameras besitzen einen Abstand von 110 mm zwischen einander.

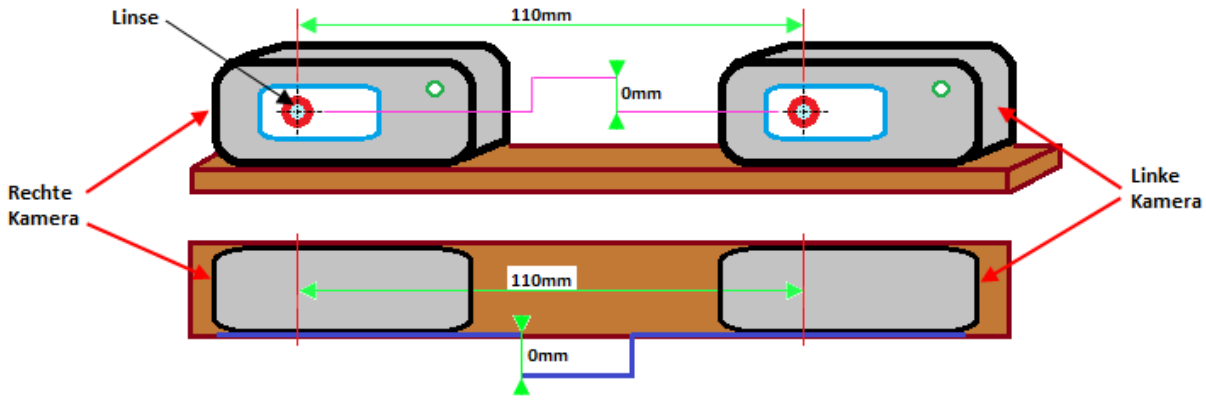


Abbildung 1: Schema zu Stereokamera

Je größer dieser Abstand desto besser ist es möglich den Abstand zur Szene zu bewerten für Objekte die weit entfernt sind.

Ein Python Programm dass die OpenCV Bibliothek einsetzt wurde in diesem Projekt implementiert um die Kameras zu kalibrieren und um den Abstand zu den Objekten der Szene zu messen. (Siehe Anhang)

Abbildung 2: OpenCV und Python Logo

Frédéric Uhrweiler
Stéphane Vujasinovic

Die Stereokamera besteht aus zwei Logitech Webcam C170.

Abbildung 3: Foto von der Logitech Webcam C170 (Quelle: www.logitech.fr)

Video aufrufe sind optimal mit Bilder von 640x480 Pixeln. Brennweite : 2,3mm.
Die selbst gebaute Stereo Kamera:

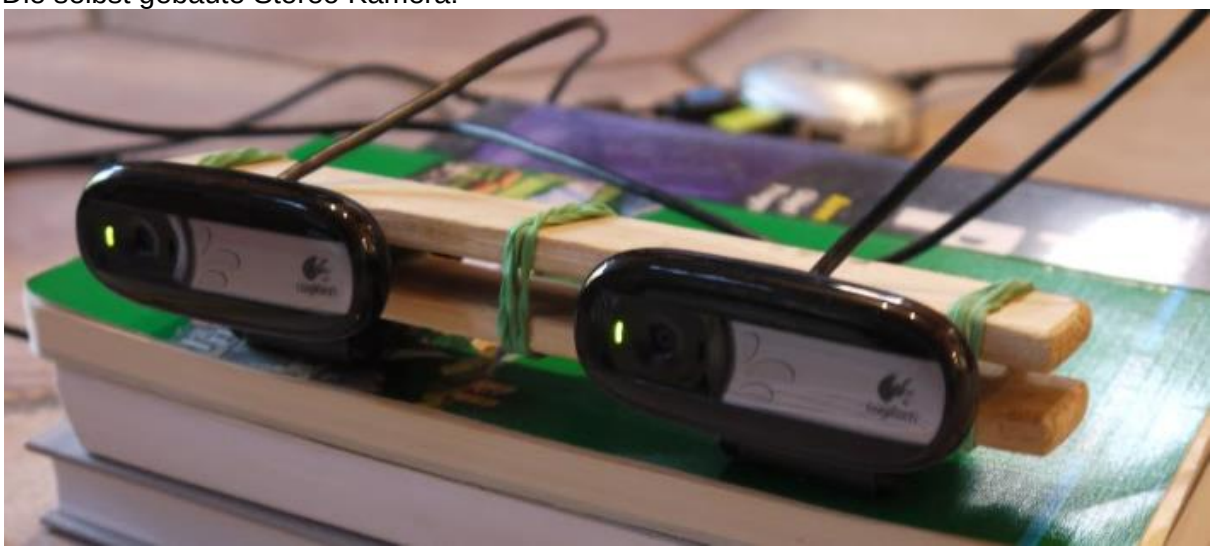


Abbildung 4: Foto unserer selbst gebauten Stereokamera

2. Kamera-Modell

Kameras erfassen die Lichtstrahlen unserer Umwelt. Im Prinzip funktioniert eine Kamera wie unser Auge, die reflektierten Lichtstrahlen unserer Umwelt kommen auf unserem Auge und Sie werden auf unsere Retina gesammelt.
Die „Pinhole Kamera“ ist das einfachste Modell. Es ist ein gutes vereinfachtes Modell um zu verstehen, wie eine Kamera funktioniert. Bei dem Modell werden alle Lichtstrahlen von einer Fläche gestoppt. Es werden nur die Strahlen, die durch das Loch durchdringen erfasst und auf einer Fläche in der Kamera umgekehrt projektiert. Das untere Bild erläutert dieses Prinzip.

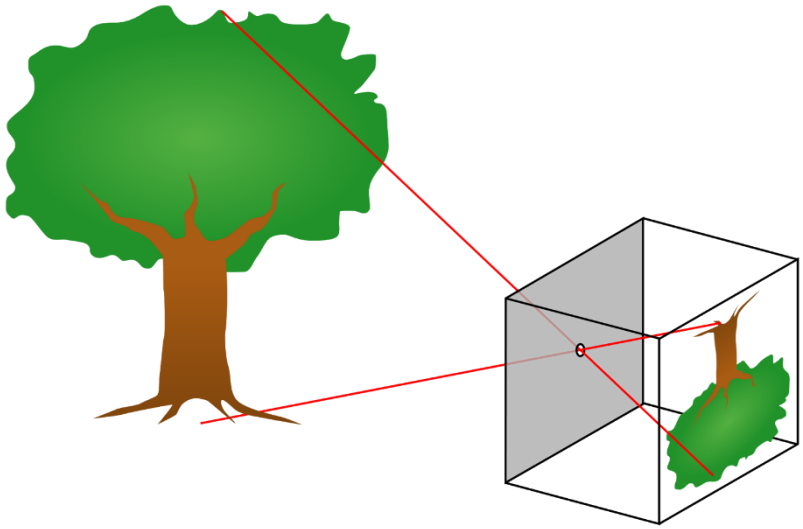


Abbildung 5: Funktionsweise einer Kamera

Quelle: https://funsizephysics.com/use-light-turn-world-upside/

Dieses Prinzip ist sehr einfach, jedoch ist es nicht eine gute Weise um genügend Licht bei einer schnellen Belichtung zu erfassen. Deshalb werden Linsen benutzt, um mehr Lichtstrahlen an einer Stelle zu sammeln. Das Problem ist, dass diese Linse, Distorsion mit sich bringt.
Man kann zwei unterschiedliche Arten von Distorsionen unterscheiden:

- die radiale Distorsion
- die tangentiale Distorsion

Die radiale Distorsion kommt von der Forme der Linse selbst und die tangentiale Distorsion kommt von der Geometrie der Kamera. Die Bilder können dann mittels Mathematische verfahren korrigiert werden.
Der Kalibrationsprozess ermöglicht ein Modell der Geometrie der Kamera und ein Modell der Distorsion der Linse zu bilden. Diese Modelle bilden die intrinsischen Parameter einer Kamera.

A. Brennweite

Die relative Größe vom Bild das auf der Fläche in der Kamera Projiziert wird, hängt von der Brennweite ab. Bei dem Pinhole Modell ist die Brennweite, der Abstand zwischen dem Loch und die Fläche wo das Bild Projiziert ist.

Das Thales Theorem ergibt dann: $-x = f * (X / Z)$
Mit:

- x: Bild des Objekts (Minuszeichen kommt dadurch dass das Bild umgekehrt wird)
- X: Größe des Objekts
- Z: Entfernung vom Loch zum Objekt
- f: Brennweite, Entfernung vom Loch zum Bild

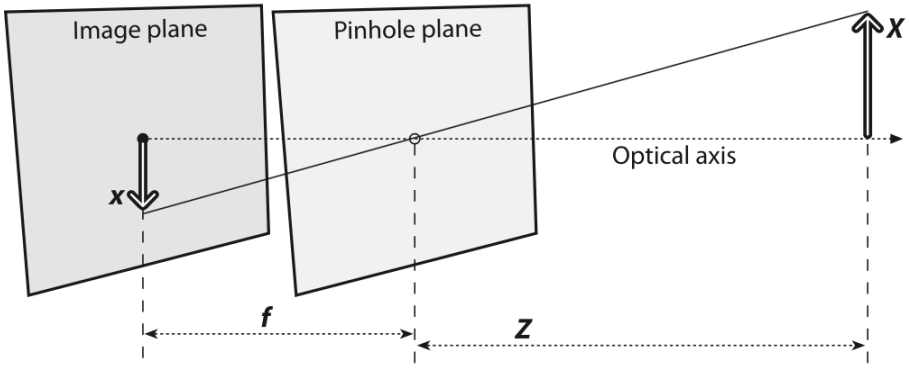


Abbildung 6: Projektierter Objekt

Quelle: Learning OpenCV 3 - O'Reilly

Da die Linse nicht perfekt zentriert ist, werden zwei Parametern eingeführt, Cx und Cy für die jeweils horizontale und senkrechte Verschiebung der Linse. Die Brennweite auf der X- und auf der Y-Achse werden ebenfalls unterscheidet, weil die Bildfläche Rechteckig ist. Dadurch erhält man also die Folgende Formel für die Position des Objekts auf der Fläche.

$$x_{\text{screen}} = f_x\left(\frac{X}{Z}\right) + c_x, \qquad y_{\text{screen}} = f_y\left(\frac{Y}{Z}\right) + c_y$$

Die projizierten Punkte der realen Welt auf der Bildfläche kann man also folgender Weise modellieren. M ist hier die Intrinsische Matrix.

$$q = MQ, \quad \text{where} \quad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

B. Distorsion der Linse

Theoretisch ist es möglich eine Linse zu bauen, die keine Distorsion verursacht mit einer parabolischen Linse. In der Praxis ist es jedoch viel leichter eine sphärische Linse herzustellen als eine parabolische Linse. Wie vorher besprochen gibt es zwei Arten von Distorsionen. Die Radiale Distorsion die von der Form der Linse kommt und die Tangentiale Distorsion die vom Montageprozess der Kamera verursacht wird.

Es gibt keine radiale Distorsion am Optischen Zentrum und sie wird immer größer wenn man sich von den Rändern nähert. In der Praxis bleibt diese Distorsion klein, es genügt eine Taylorentwicklung bis zum dritten Term zu machen. Es ergibt sich die folgende Formel.

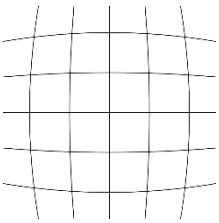


Abbildung 7: Radiale Distorsion (Quelle: Wikipedia)

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

x und y sind die Koordinate des Original Punktes auf der Bildfläche und damit wird die Position des Korrigierten Punkts berechnet.

Es gibt auch eine tangentiale Distorsion, weil die Linse nicht perfekt parallel mit der Bildfläche aufgebaut ist. Um dies zu korrigieren werden zwei zusätzliche Parametern eingeführt, p1 und p2.

$$x_{\text{corrected}} = x + [2p_1 y + p_2(r^2 + 2x^2)]$$
$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 x]$$

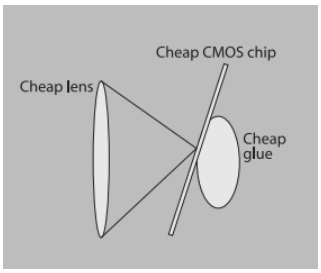


Abbildung 8. Tangentiale Distorsion (Quelle: Learning OpenCV 3 - O'Reilly)

C. Kalibrierung mit OpenCV

Die Bibliothek OpenCV ermöglicht uns die Intrinsischen Parameter anhand von spezifische Funktionen zu berechnen, dieser Prozess nennt man Kalibrierung. Dies wird mit verschiedene sichten eines Schachbretts ermöglicht.

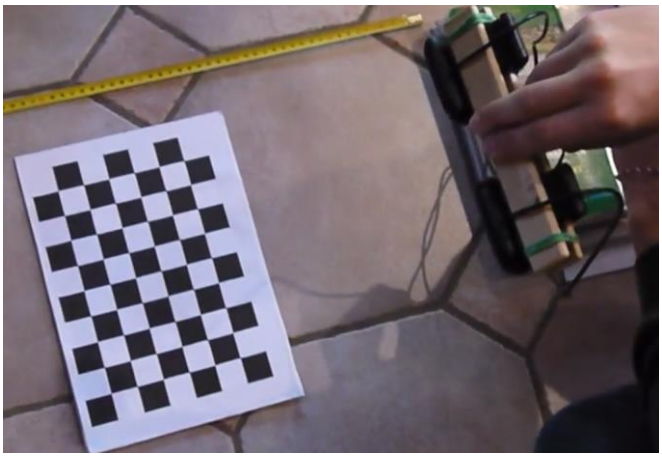


Abbildung 9: Foto während Bilder für die Kalibrierung genommen werden

Das Programm für die Aufnahmen der Bilder für das spätere kalibrieren nennt sich "**Take_images_for_calibration.py**"
Wenn die Ecken vom Schachbrett auf beiden Kameras erkannt werden, öffnen sich zwei Fenster mit dem erkannten Bild für jede Kamera. Die Bilder werden dann vom Benutzer entweder gespeichert oder gelöscht. Gute Bilder kann man erkennen in denen die corners sehr deutlich erkennbar sind. Diese Bilder werden später für die Kalibrierung in den main Programm"**Main_Stereo_Vision_Prog.py**" verwendet. OpenCV empfiehlt mindesten 10 Bilder für jede Kamera zu haben um eine gute Kalibration zu bekommen. Wir haben mit 50 Bilder für jede Kamera gute Ergebnisse bekommen.
Um die Kameras zu Kalibrieren sucht der Python Code die Ecken vom Schachbrett auf jedem Bild für jede Kamera mit der OpenCV Funktion: *cv2.findChesssboardCorners*

Die Position der Ecken für jedes Bild werden dann in einem Image Vektor gespeichert und die Objektpunkte für die 3d Szene werden in einen anderen Vektor gespeichert. Man verwendet dann diese Imgpoints und Objpoints in der Funktion *cv2.calibeCamera()* die an dem Ausgang die Kamera Matrix, die Distorsion Koeffizienten, die Rotation und Translation Vektors zurückgibt.

Die Funktion *cv2.getOptimalNewCameraMatrix()* ermöglicht uns präzise Kamera Matrizen zu bekommen die wir später in der Funktion *cv2.stereoRectify()* verwenden.

Nach der Kalibrierung mit OpenCV erhalten wir die folgende Matrix M für unsere Kamera:

Matrix M ohne Rektifikation (Rechte Kamera):

885.439	0	301.366
0	885.849	233.812
0	0	1

Matrix M_{rekt} Rektifiziert (Rechte Kamera):

871.463	0	303.497
0	869.592	233.909
0	0	1

Matrix M ohne Rektifikation (Linke Kamera):

748.533	0	345.068
0	749.062	228.481
0	0	1

Matrix M_{rekt} Rektifiziert (Linke Kamera):

730.520	0	349.507
0	725.714	227.805
0	0	1

3. Stereoabbildung

A. Erklärung

Die Stereo Vision ermöglicht die Tiefe in einem Bild zu erkennen, Messungen im Bild zu machen und 3D Lokalisierungen zu unternehmen. Dazu müssen unter anderem Punkte die übereinstimmen zwischen beiden Kameras gefunden werden. Daraus kann man dann die Entfernung zwischen der Kamera und dem Punkt herleiten. Man behilft sich der Geometrie des Systems um die Berechnung zu vereinfachen.

Bei dem Stereo-Imaging werden diese Vier Schritte durchgeführt:

1. Entfernung der Radialen und Tangentialen Distorsion durch mathematische Berechnungen. Man erhält damit Unverzerrte Bildern.
2. Rektifizieren des Winkels und des Abstands der Bilder. Diese Etappe ermöglicht beide Bildern Koplanar auf der Y-Achse, damit wird die Suche von Korrespondenzen erleichtert und man braucht nur auf einer einzigen Achse die Suche zu machen (nämlich auf der X-Achse).
3. Finden desselben Merkmals in dem rechten und linken Bild. Damit erhält man eine Disparitätskarte, die die Unterschiede zwischen den Bildern auf der X-Achse darstellt.
4. Der Letzte Schritt ist die Triangulation. Man transformiert die Disparitätskarte in Abstände durch Triangulation.

Schritt 1: Entfernung der Distorsion
Schritt 2: Rektifizieren
Schritt 3: Finden desselben Merkmals in beiden Bildern
Schritt 4: Triangulation

B. Triangulation

Bei dem letzten Schritt, die Triangulation geht man davon aus, dass beide Projektionsbildern Koplanar sind und dass die horizontale Pixel-Reihe des Linke Bilds mit der entsprechenden horizontalen Pixel-Reihe des Linke Bilds ausgerichtet ist.
Mit den vorherigen Hypothesen kann man jetzt das folgende Bild aufbauen.

Der Punkt P liegt in der Umwelt und wird auf dem linken und rechten Bild auf pl und pr abgebildet, mit den entsprechenden Koordinaten xl und xr. Dies ermöglicht uns eine neue Größe einzuführen, die Disparität: d = xl - xr. Man erkennt, dass je weiter der Punkt P entfernt ist, je kleiner wird die Größe d. Die Disparität ist also umgekehrt proportional zur Entfernung.

Mit der folgenden Formel kann die Entfernung berechnet werden: Z=f*T/(xl-xr)

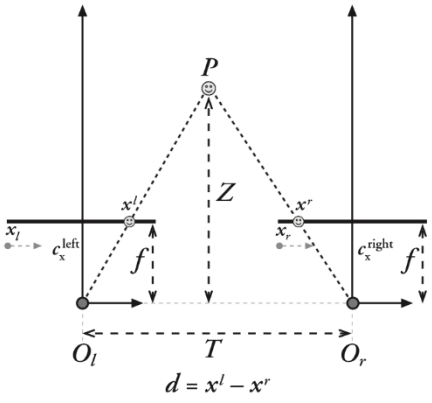


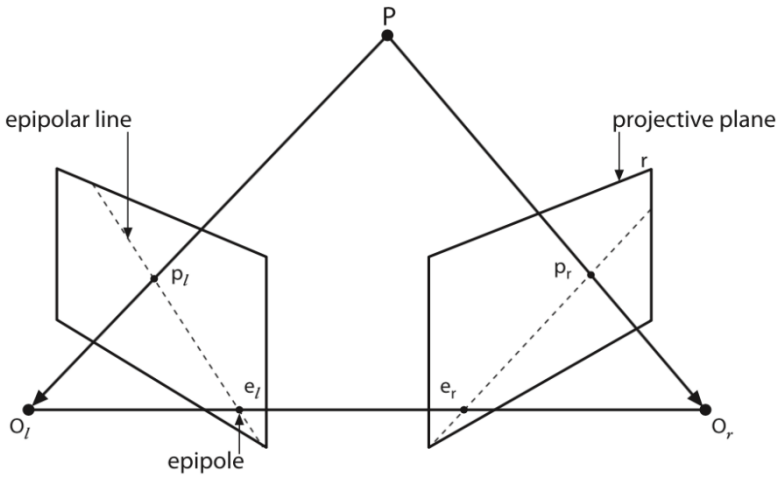
Abbildung 10: Triangulation (Quelle: Learning OpenCV 3 - O'Reilly)

Man erkennt, dass es einen nichtlinearen Zusammenhang zwischen Disparität und Entfernung gibt. Wenn die Disparität in der Nähe von 0 ist, führen kleine Disparitätsunterschiede zu große Entfernungsunterschiede. Dies ist umgekehrt wenn die Disparität groß ist. Kleine Disparitätsunterschiede führen nämlich dann nicht zu große Entfernungsunterschiede. Daraus kann man schließen, dass die Stereo-Vision eine hohe Tiefenresolution hat, nur für Objekte die nah an der Kamera.

Diese Methode funktioniert aber nur wenn die Konfiguration der Stereo-Kamera ideal ist. In der Realität ist dies jedoch nicht der Fall. Deswegen wird das linke und rechte Bild mathematisch parallel ausgerichtet. Natürlich müssen die Kameras zumindest approximativ physisch parallel positioniert werden.

Bevor wir die Methode zur Mathematischen Ausrichtung der Bildern erklären, müssen wir zuerst die Epipolare Geometrie verstehen.

C. Epipolare Geometrie



Das oben dargestellte Bild zeigt uns das Modell einer nicht-perfekten Stereo-Kamera die aus zwei Pinhole-Kameramodelle besteht. Durch die Kreuzung der Linie der Projektierungszentren (Ol, Or) mit den Proektierungsebenen entstehen die Epipolarpunkten el und er. Die Linien (pl, el) und (pr, er) werden Epipolarlinen genannt. Das Bild aller möglichen Punkte eines Punkts auf einer Projektierungsebene ist die Epipolarlinie die auf der anderen Bildebene liegt und durch den Epipolarpunkt und dem Gesuchten Punkt geht. Dies ermöglicht, die Suche des Punkts auf einer einzigen Dimension zu begrenzen anstatt auf einer ganzen Ebene.
Man kann also folgende Punkte zusammenfassen:

- Jeder 3D Punkt in der Sicht einer Kamera ist im Epipolaren Plan enthalten
- Ein Merkmal in einer Ebene muss sich auf der entsprechenden Epipolarlinien der andere Ebene befinden (Epipolarbedingung)
- Eine zweidimensionale Suche eines entsprechenden Merkmals wird zu einer eindimensionalen Suche umgewandelt, wenn man die Epipolargeometrie kennt.
- Die Reihenfolge der Punkte wird behalten, d.h. dass zwei Punkte A und B in derselben Reihenfolge auf der Epipolarlinien einer Ebene gefunden werden, wie auf der, der anderen Ebene.

D. Essentielle und Fundamentale Matrizen

Um zu verstehen wie die Epipolare Linien berechnet werden, muss man zuerst die essentielle und die Fundamentale Matrizen erläutern (entsprechend Matrize E und F).
Die essentielle Matrix E beinhaltet die Informationen, wie die beiden Kameras miteinander physisch angeordnet sind. Es beschreibt die Lokalisierung der zweiten Kamera relativ zur ersten Kamera durch Translation- und Rotation-Parametern. Diese Parameter sind in der Matrix nicht direkt lesbar, da diese zur Projektierung benutzt wird. Im Abschnitt *Stereo Kalibration* erklären wir, wie man R und T berechnet (Rotationsmatrix und Translationsvektor).
Die Matrix F beinhaltet die Informationen der essentiellen Matrix E, zur physischen Anordnung der Kameras und die Informationen über den Intrinsischen Parametern der Kameras.

Die Relation zwischen den projektierten Punkt auf dem linken Bild pl und der auf dem rechten Bild pr ist so definiert:

$p_r^T E p_l = 0$

Man könnte denken, dass diese Formel den Zusammenhang zwischen dem linken und der rechten Punkt vollständig beschreibt. Jedoch muss man bemerken, dass die 3x3 Matrix E von Rang 2 ist. Das heißt, dass diese Formel, die Gleichung einer Gerade ist.

Um die Beziehung zwischen den Punkten vollständig zu definieren, muss man also die intrinsischen Parameter berücksichtigen.

Wir erinnern uns das q=Mp, mit der intrinsischen Matrix M.
Durch ersetzen in der vorherigen Gleichung ergibt sich:
$q_r^T (M_l^{-1})^T E M_l^{-1} q_l = 0$

Man ersetzt:
$F = (M_l^{-1})^T E M_l^{-1}$

Und erhält also:
$q_r^T F q_l = 0$

E. Rotationsmatrix und Translationsvektor

Jetzt, dass wir die essentielle Matrix E und die fundamentale Matrix F erläutert haben, müssen wir sehen, wie die Rotationsmatrix und den Translationsvektor berechnet werden.

Wir definieren folgende Notationen:

- P_l und P_r definieren die Positionen des Punktes im Koordinatensystem der linken bzw. rechten Kameras
- R_l und T_l (bzw. R_r und T_r) definieren die Rotation und die Translation von der Kamera zum Punkt in der Umwelt für die Linke (bzw. Rechte) Kamera.
- R und T sind die Rotation und die Translation die, das Koordinatensystem der rechten Kamera im Koordinatensystem der Linken Kamera bringt.

Es ergeben sich:
$P_l = R_l P + T_l$ und $P_r = R_r P + T_r$

Man hat auch:
$P_l = RT(P_r - T)$

Mit diesen drei Gleichungen ergibt sich schließlich:

$R = R_r R_l T$
$T = T_r - R T_l$

F. Stereo Rektifikation

Bis jetzt haben wir uns mit dem Thema “Stereo-Kalibrierung“ beschäftigt. Es handelte sich um die Beschreibung der Geometrischen Anordnung beider Kameras. Die Aufgabe der Rektifikation besteht darin, die zwei Bildern zu projizieren damit sie in der exakt selben Ebene

liegen und die Pixelreihen genau auszurichten, dies damit die Epipolarlinien horizontal werden um die Korrespondenz eines Punkts in den Beiden Bildern zufälissiger finden zu können.
Als Ergebnis des Prozesses der Ausrichtung beider Bildern erhält man 8 Terme, 4 für jede Kamera:

- ein Distorsions-Vektor
- eine Rotations-Matrix R_{rect}, die am Bild angewendet werden muss
- eine rektifizierte Kamera-Matrix M_{rect}
- eine nicht-rektifizierte Kamera-Matrix M

OpenCV ermöglicht uns diese Terme mit zwei Algorithmen zu berechnen: der Hartley Algorithmus und der Bouguet Algorithmus.

1. Hartley Algorithmus

Beim Hartley Algorithmus werden dieselben Punkte in den beiden Bildern gesucht. Er versucht, die Disparitäten zu minimieren und Homographien zu finden die, die Epipole im unendlichen setzen. Bei dieser Methode braucht man also nicht die intrinsischen Parameter für jede Kamera zu berechnen.

Ein Vorteil dieser Methode ist, dass eine Kalibration möglich ist, nur dank der Beobachtung von Punkten in der Szene. Ein großer Nachteil ist aber dass man keine Skalierung des Bildes hat, man hat nur die Information des relativen Abstands. Man kann nicht genau messen wie weit ein Objekt von den Kameras entfernt ist.

2. Bouguet Algorithmus

Der Bouguet Algorithmus benutzt die berechnete Rotation Matrix und der Translation Vektor um beide Projetzierten Ebenen um eine halbe Umdrehung zu drehen damit sie sich in der gleichen Ebene befinden. Damit werden die Hauptstrahlen parallel und die Ebenen Koplanar aber noch nicht Reihenweise ausgerichtet. Dies wird dann später gemacht.
Im Projekt haben wir den Bouguet Algorithmus benutzt.

4. Funktionsweise der Programme für die Stereo Abbildung

Wie schon erwähnt ist das Programm in Python kodiert und die Bibliothek OpenCV wird benutzt. Wie haben uns für die Python-Sprache und die Bibliothek OpenCV entschieden, weil wir bereits Erfahrungen damit hatten und weil es viel Dokumentation darüber gibt. Ein anderes Argument für diese Entscheidung ist, dass wir auch nur mit „Open Source“ Bibliotheken arbeiten wollten.

Zwei Python Programme wurden für dieses Projekt entwickelt.

Das Erste “**Take_images_for_calibration.py**” dient für die Aufnahme von guten Bildern, die später in der Kalibrierung der zwei Kameras (Distorsion Kalibrierung und Stereo Kalibrierung) eingesetzt werden.

Das zweite Programm und somit das Hauptprogramm ”**Main_Stereo_Vision_Prog.py**” wird für die Stereoabbildung eingesetzt. Wir Kalibrieren in diesem Programm die Kameras mit den

Bildern die aufgenommen wurden, erzeugen eine Disparitätskarte und dank einer Geradengleichung die experimental gefunden wird können wir für jeden Pixel den Abstand Messen. Ein WLS Filter ist am Ende verwendet um die Rändern von Objekten besser zu erkennen.

Die Python Programme sind im Anhang zu finden.

A. Benutzte Packages

Folgende Packages wurden im Programm importiert:

- Die Version von OpenCV.3.2.0 mit opencv_contrib (enthält die Stereo Funktionen) als "cv2" in Python genannt, enthält:
  - die Bibliothek zur Bildverarbeitung
  - die Funktionen für Stereo Vision
- Numpy.1.12.
  - Verwendet für Matrizen-Operationen (Bilder bestehen aus Matrizen)
- Workbook von openpyxl
  - Package um Daten in einem Excel File schreiben zu können
- "normalize" der Bibliothek sklearn 0.18.1
  - sklearn ermöglicht Machine Learning aber in diesem Projekt verwendet man nur den WLS Filter

```
# Package importation
import numpy as np
import cv2
from openpyxl import Workbook # Used for writing data into an Excel file
from sklearn.preprocessing import normalize
```

Code 1: Package Importation

B. Video Loop

Um mit den Kameras zu arbeiten muss man sie als erstes Aktivieren. Die Funktion *cv2.VideoCapture()* aktiviert beide Kameras indem man die Nummer der Ports von jeder Kamera eingibt (Zwei Objekten werden so im Programm erzeugt, die, die Methoden von der Klasse *cv2.VideoCapture()* benutzen können).

```
# Call the two cameras
CamR= cv2.VideoCapture(0)
CamL= cv2.VideoCapture(2)
```

Code 2: Aktivierung der Kameras mit OpenCV

Um ein Bild von den Kameras zu bekommen, wird die Methode *cv2.VideoCapture().read()* eingesetzt, als Output bekommt man ein Bild der Szene die die Kamera im Moment wo diese Funktion aufgerufen wird ansieht. Um dann ein Video zu bekommen muss man diese Methode immer wieder in einer unendlichen Schleife aufrufen. Um effizienter zu sein ist es empfohlen die BGR Bilder in Gray Bilder umzuwandeln, dies wird mit der Funktion *cv2.cvtColor()* ausgeführt.

```
while True:
    retR, frameR= CamR.read()
    retL, frameL= CamL.read()
    grayR= cv2.cvtColor(frameR,cv2.COLOR_BGR2GRAY)
    grayL= cv2.cvtColor(frameL,cv2.COLOR_BGR2GRAY)
```

Code 3: Bildverarbeitung

Um das Video auf dem PC zu sehen wird die Funktion *cv2.imshow()* verwendet, sie ermöglicht ein Fenster zu öffnen wo das Video gezeigt werden kann.

```
cv2.imshow('VideoR',grayR)
cv2.imshow('VideoL',grayL)
```

Code 4: Bilder anzeigen

Um aus der unendliche Schleife raus zu kommen wird ein break eingesetzt. Dieser wird aktiv wann immer der Benutzer auf die Leertaste drückt. Die Erkennung dass man auf einer Taste gedrückt hat wird dank der Funktion *cv2.waitKey()* durchgeführt.
Am Ende müssen noch die zwei verwendeten Kamera deaktiviert werden mit der Methode *cv2.VideoCapture().release()* und die geöffneten Windows werden mit der Funktion *cv2.destroyAllWindows()* geschlossen.

```
    # End the Programme
    if cv2.waitKey(1) & 0xFF == ord(' '):
        break

# Release the Cameras
CamR.release()
CamL.release()
cv2.destroyAllWindows()
```

Code 5: Typische Exit für ein Programm

C. Funktionsweise vom Programm "**Take_images_for_calibration.py**"

Wenn dieses Programm gestartet wird, werden beide Kameras aktiv und zwei Fenster werden geöffnet damit der Benutzer sehen kann wo das Schachbrett in den Bildern positioniert ist.

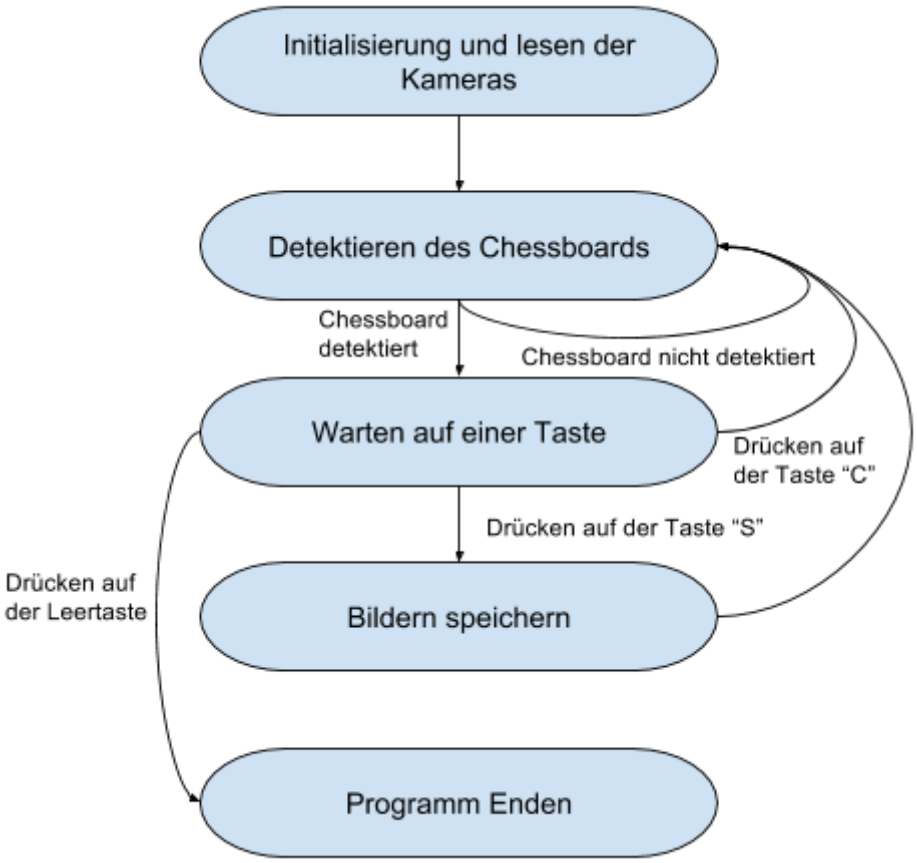


*Diagramm 1: Funktionsweise von "**Take_images_for_calibration.py**"*

1. Vektoren für die Kalibrierung

Die Funktion *cv2.findChessboardCorners()* wird eine definierte Anzahl von Schachbrett-Ecken suchen und es werden folgende Vektoren erzeugt:

- **imgpointsR**: enthält die Koordinate der Ecken im Rechtem Bild (im Bildraum)
- **imgpointsL**: enthält die Koordinate der Ecken im Linkem Bild (im Bildraum)
- **objpoints**: enthält die Koordinate der Ecken im Objektraum

Die Präzision der Koordinate der gefundenen Ecken wird erhöht in dem die Funktion *cv2.cornerSubPix()* eingesetzt wird.

2. Erfassung der Bilder für die Kalibrierung

Hat das Programm die Position der Schachbrett-Ecken auf beide Bilder erkannt, werden zwei neue Fenster geöffnet wo man die aufgenommenen Bilder bewerten kann. Wenn die Bilder nicht verschwommen sind und gut aussehen muss man auf die "s" (Save) Taste drücken um die Bildern zu speichern. Wenn das Gegenteil der Fall ist kann man auf die "c" (Cancel) Taste

drücken damit die Bilder nicht gespeichert werden. Mit der Funktion *cv2.drawChessboardCorners()* überlagert das Programm ein Schachbrett-Muster auf die Bilder.

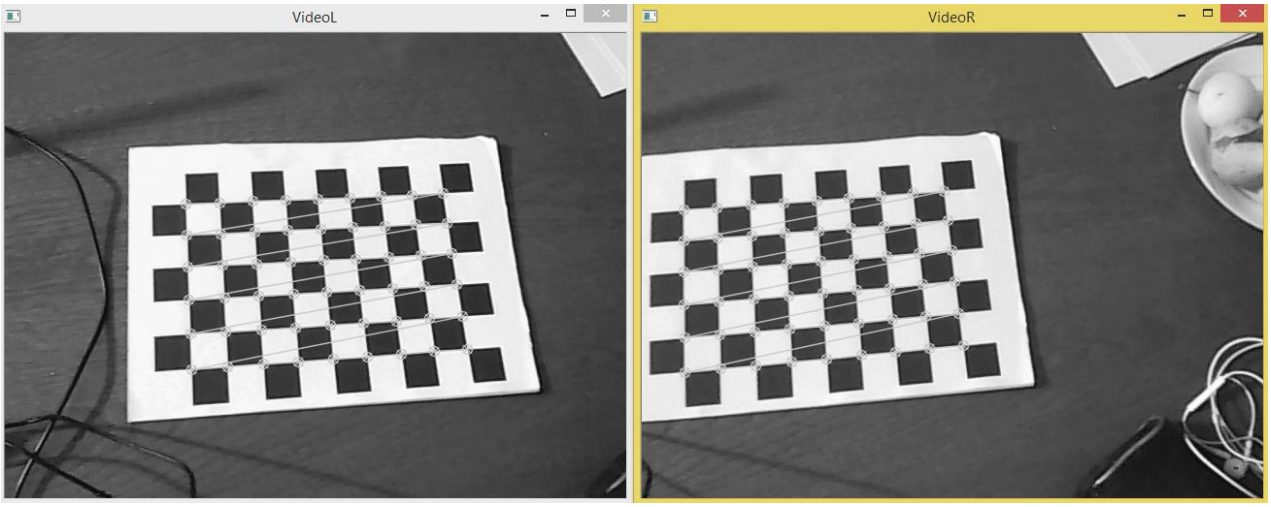


Abbildung 11: Erkennung der Ecken von einem Schachbrett

D. Funktionsweise vom Programm "**Main_Stereo_Vision_Prog.py**"

In der Initialisierung werden die Kameras zuerst einzeln Kalibriert, zur Entfernung der Distorsion. Dann wird die Stereo Kalibration durchgeführt (Rotation beseitigen, Alignierung der Epipolarlinien). In eine unendliche Schleife werden die Bilder bearbeitet und eine Disparitätskarte wird erzeugt.

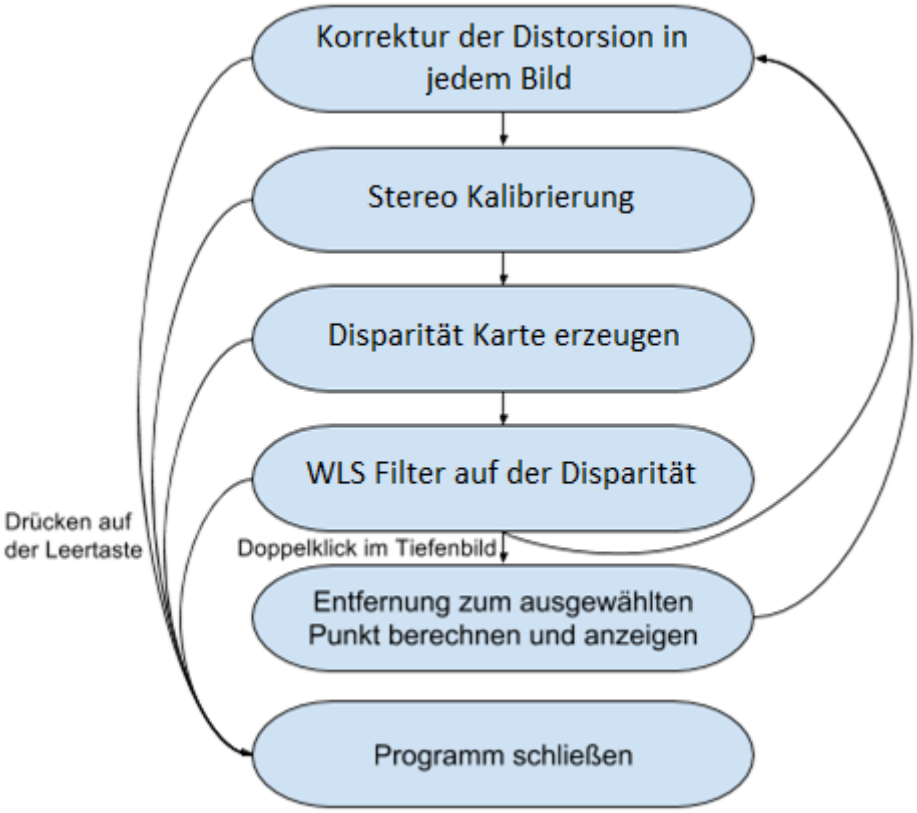


Diagramm 2: Funktionsweise von " Main_Stereo_Vision_Prog.py"

1. Kalibrierung der Distorsion

Für die Korrektur der Distorsion der Kameras werden die Bilder verwendet die mit dem Programm "Take_images_for_calibration.py" aufgenommen wurden.
Diese Kalibrierung basiert sich auf "Take_images_for_calibration.py" indem in **Imgpoints** und **Objpoints** die Position der Schachbrett-Ecken gespeichert sind. Die Funktion *cv2.calibrateCamera()* wird eingesetzt um neue Kamera Matrizen (Die Kamera Matrix beschreibt die Projektion von einem Punkt der 3D Welt in einem 2D Bild), Distorsion Koeffizienten, Rotation und Translation Vektoren für jede einzelne Kamera zu bekommen die später gebraucht werden um die Distorsion von jeder Kamera zu beseitigen. Um die Optimale Kamera Matrizen für jede Kamera zu bekommen wird die Funktion *cv2.getOptimalNewCameraMatrix()* benutzt (Erhöhung der Präzision).

```
#    Right Side
retR, mtxR, distR, rvecsR, tvecsR = cv2.calibrateCamera(objpoints,
                                                        imgpointsR,
                                                        ChessImaR.shape[::-1],None,None)

hR,wR= ChessImaR.shape[:2]
OmtxR, roiR= cv2.getOptimalNewCameraMatrix(mtxR,distR,
                                           (wR,hR),1,(wR,hR))
```

Code 6: Kalibrierung der Distorsion in Python

2. Kalibrierung der Stereokamera

Für die Stereo-Kalibrierung wird die Funktion *cv2.StereoCalibrate()* verwendet, sie berechnet die Transformation zwischen beiden Kameras (eine Kamera dient als Referenz für die andere).

Die Funktion *cv2.stereoRectify()* ermöglicht, die Epipolarlinien der beiden Kameras auf derselben Ebene zu bringen. Diese Transformation erleichtert die Arbeit für die Funktion die die Disparitätskarte erstellt, weil dann die Blockenübereinstimmung nur noch in einer Dimension gesucht werden muss. Mit dieser Funktion bekommt man auch die Essentielle Matrix und die Fundamentale Matrix die in der nächsten Funktion gebraucht werden.

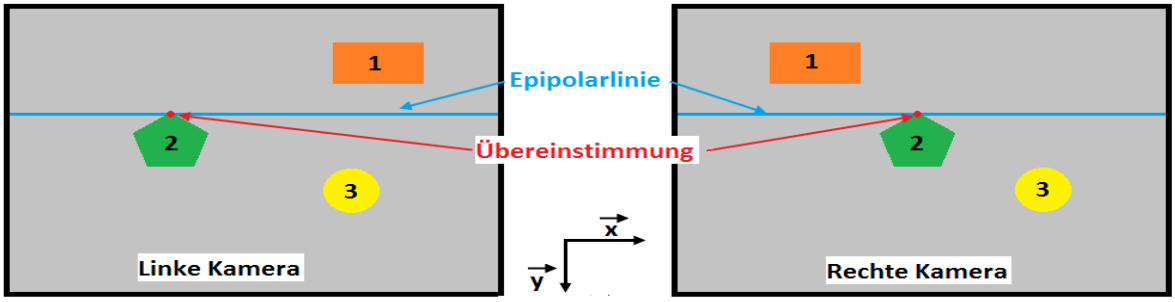


Abbildung 12: Prinzip der Epipolarlinien

Die Funktion *cv2.initUndistortRectifyMap() e*rgibt ein Bild das keine Distorsion hat. Diese Bilder werden dann in der Berechnung der Disparitätskarte verwendet.

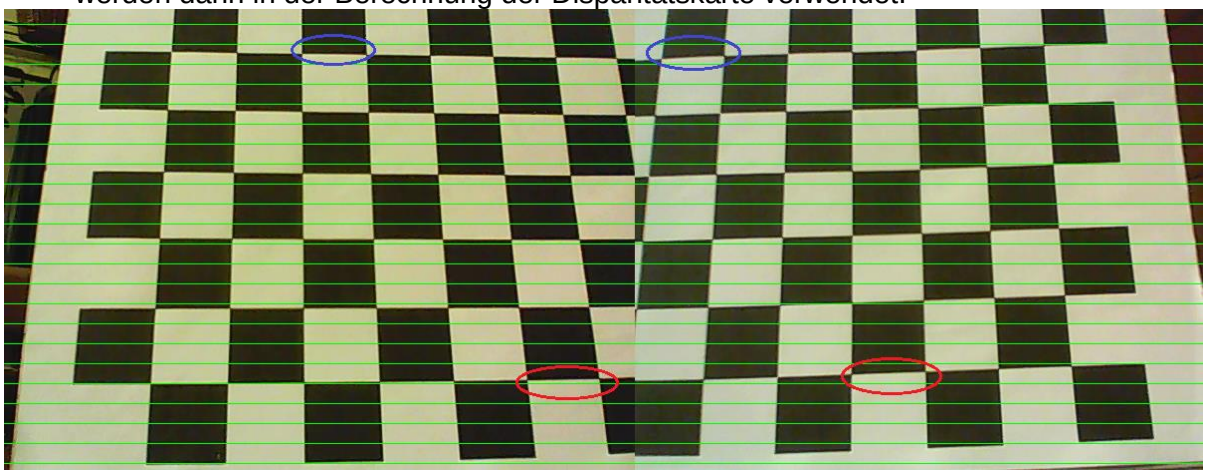


Abbildung 13: Ohne kalibrierung

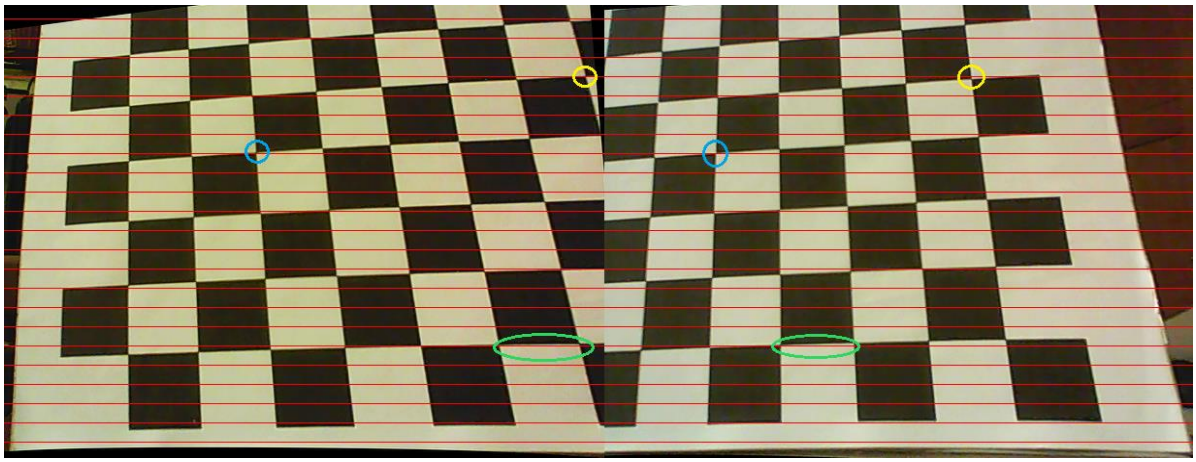


Abbildung 14: Mit Kalibrierung

3. Berechnung der Disparitätskarte

Um die Disparitätskarte zu berechnen wird ein StereoSGBM Objekt erzeugt mit der Funktion *cv2.StereoSGBM_create().* Diese Klasse benutzt ein semi-global Matching Algorithmus (Hirschmüller, 2008) um eine Stereo Übereinstimmung zwischen den Bildern der Rechten Kamera und der Linken zu erhalten.

Funktionsweise des Semi-Global Matching Algorithmus:

Folgende Szene ist der Stereokamera präsentiert:

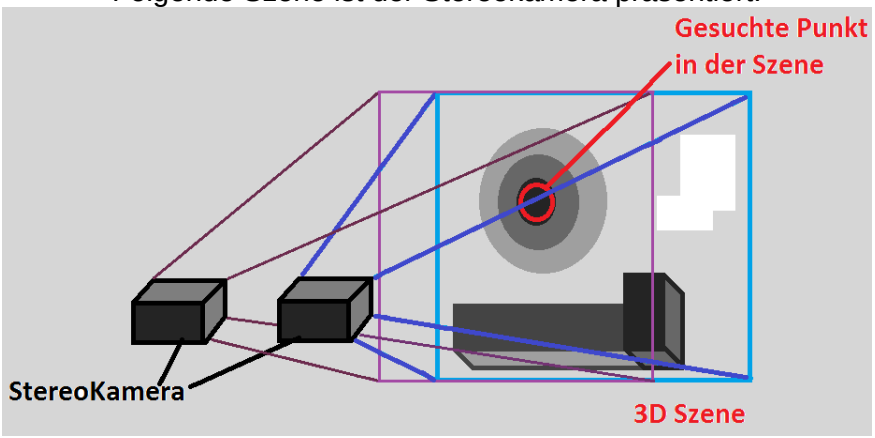


Abbildung 15: Beispiel einer Stereokamera die eine Szene beobachtet

Man definiert in den Eingangsparametern die Größe der Blöcke. Diese Blöcke ersetzen die Pixels wenn die Größe (Block Size) größer als 1 ist. Das erzeugte SGBM Objekt vergleicht die Blöcke von einem Referenzbild mit den Blöcken vom Match Bild. Wenn die Stereokalibrierung gut gemacht wurde muss zum Beispiel ein Block der Reihe Vier vom Referenzbild mit alle Blocken des Match Bild verglichen werden die nur auf der Vierten Reihe sind. Auf diese Weise wird die Rechnung der Disparitätskarte effizienter.

Nehmen wir das vorherige Beispiel mit den Blöcken der Vierten Reihe um zu erklären wie die Disparität-Karte hergestellt wird.
In der Unteren Abbildung muss man den Block (4,7) der vierten Reihe, siebten Spalte vom Basis Bild mit alle anderen Blocken(4,i) der Vierten Reihe vergleichen(Gleichen Epipolarlinie) vom Match Bild.

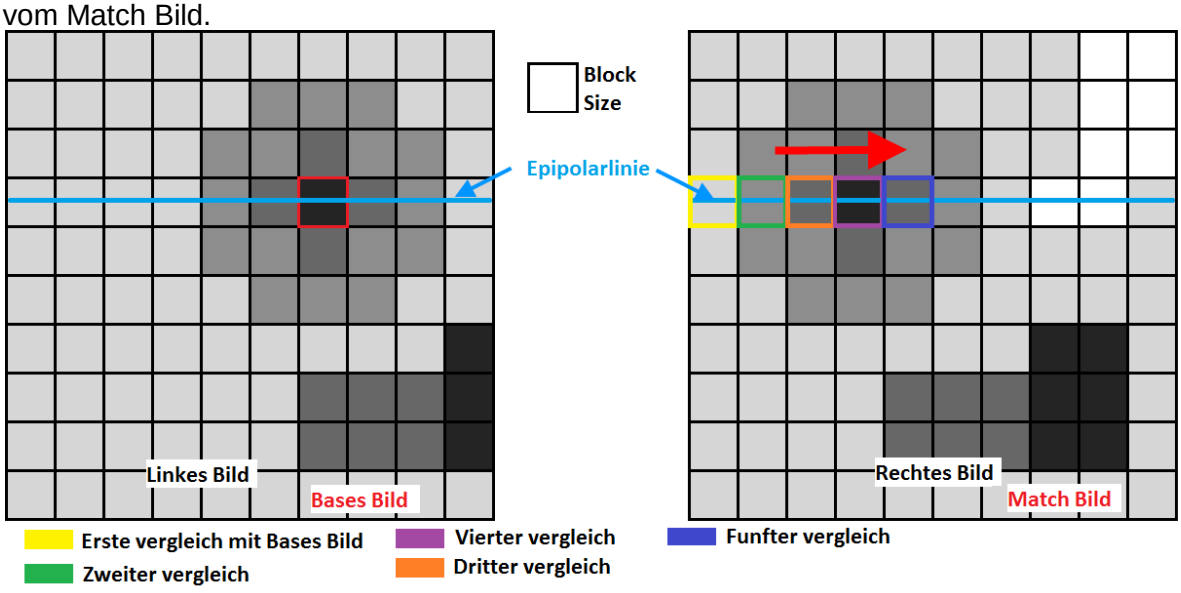


Abbildung 16: Übereinstimmung von Blocken suchen mit StereoSGBM

Je größer die Einstimmung zwischen dem Referenz Block mit dem Match Block desto wahrscheinlicher muss er der Selbe Punkt in der Umwelt sein.

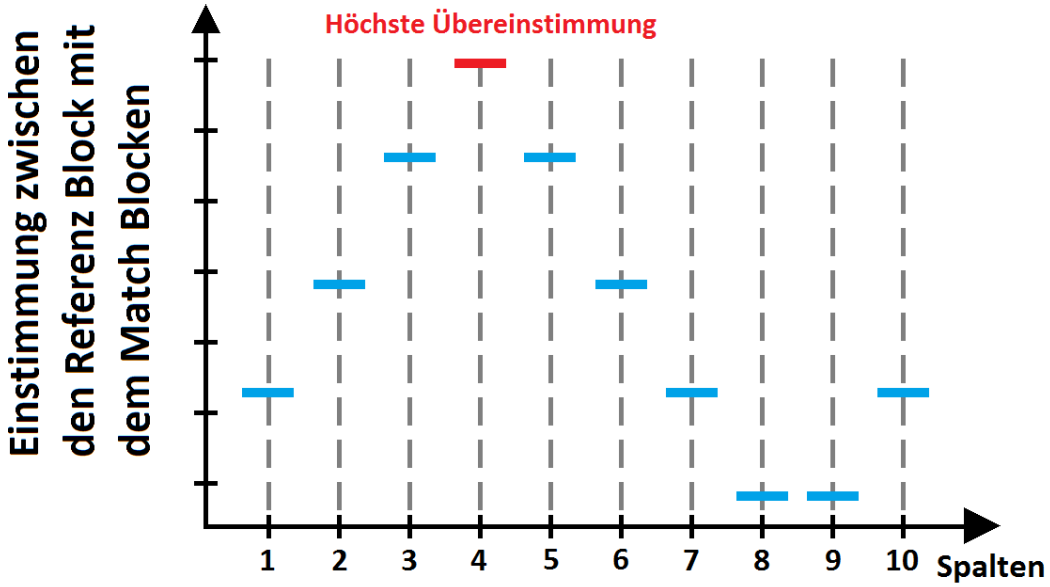


Abbildung 17: Erkennung von Selben Blocken mit StereoSGBM

Man erkennt in diesem Beispiel dass der Referenz Block(4,7) den Höchsten Übereinstimmungsgrad mit dem Match Block(4,4) hat.
In der Theorie soll es so ablaufen aber in der Praxis werden noch 4 andere Richtung Standard-mäßig in OpenCV noch bearbeitet um präziser zu sein mit derselben Methode wie für eine Richtung.

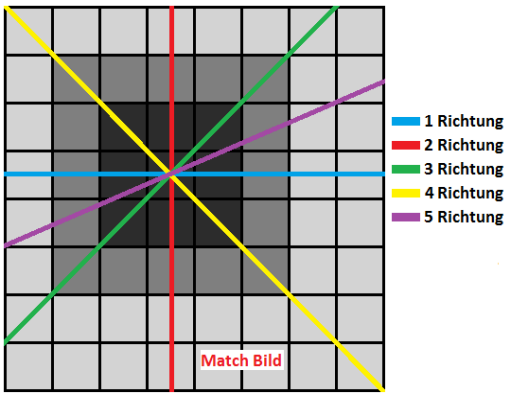


Abbildung 18: Beispiel für die Fünf Richtungen vom StereoSGBM Algorithmus bei OpenCV

Um die Disparität zu finden, werden die Koordinaten des Match Blocks mit dem Referenz Block subtrahiert, dann wird der absolute wert vom Ergebnis genommen und je größer dieser Wert desto näher an der Stereokamera ist das Objekt.

Das Programm benutzt kalibrierte schwarzweiß-Bilder für die Berechnung der Disparitätskarte, es ist auch möglich mit BGR Bilder zu arbeiten aber das würde mehr Zeitaufwand für den Computer bedeuten. Die Rechnung der Karte erfolgt mit einer Methode von unserem Stereo erzeugten Objekt, cv2.StereoSGBM_create().compute().

```
# Show the result for the Disparity Map
disp= ((disp.astype(np.float32)/ 16)-min_disp)/num_disp
cv2.imshow('disparity', disp)
```

Code 7: Die Disparitätskarte zeigen

Mit unseren Parametern die in der Initialisierung festgelegt wurden bekommen wir folgendes Ergebnis für die disparitätskarte.

```
# Create StereoSGBM and prepare all parameters
window_size = 3
min_disp = 2
num_disp = 130-min_disp
stereo = cv2.StereoSGBM_create(minDisparity = min_disp,
    numDisparities = num_disp,
    blockSize = window_size,
    uniquenessRatio = 10,
    speckleWindowSize = 100,
    speckleRange = 32,
    disp12MaxDiff = 5,
    P1 = 8*3*window_size**2,
    P2 = 32*3*window_size**2)
```

Code 8: Parametern für die Instanz von StereoSGBM

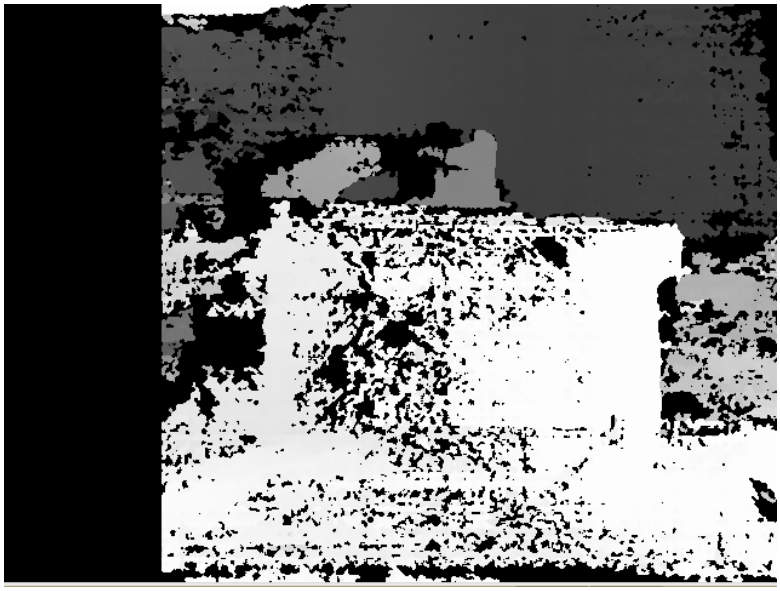


Abbildung 19: Ergebnis für die Disparitätskarte

In dieser Disparitätskarte gibt es noch sehr viel Rauschen, um den zu eliminieren wird ein morphologischer Filter eingesetzt. Ein “Closing” Filter wird verwendet mit der OpenCV Funktion *cv2.morphologyEx(cv2.MORPH_CLOSE)* um die kleinen schwarzen Punkte zu entfernen.

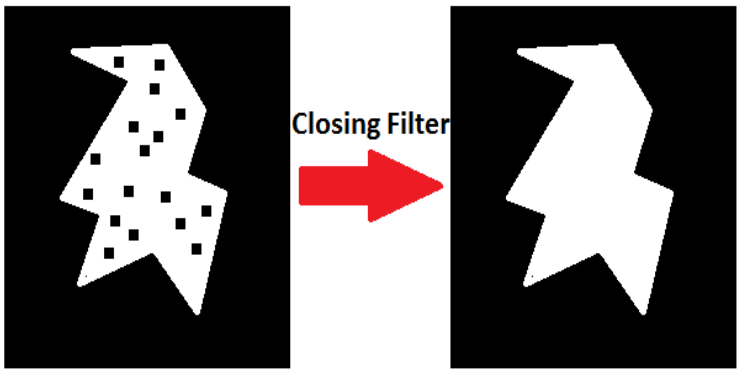


Abbildung 20: Beispiel von einem Closing Filter

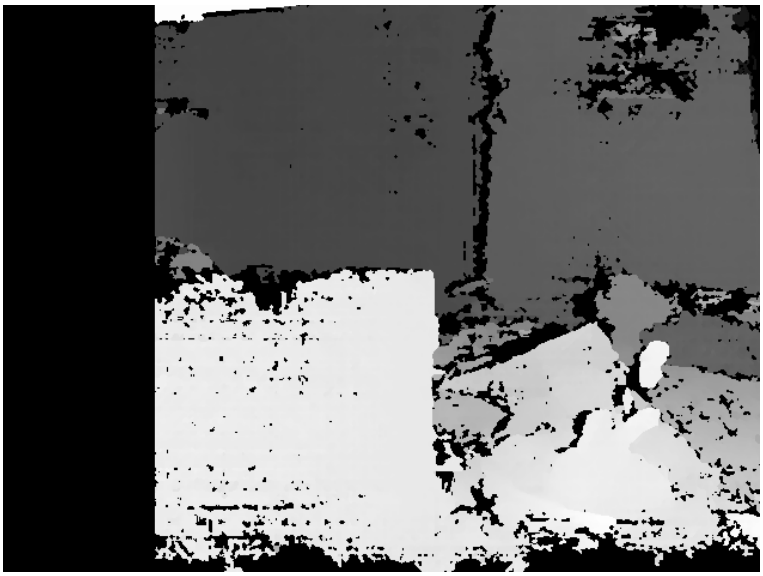


Abbildung 21: Ergebnis von eine Disparität Karte nach einem Closing Filter

Ein anderes Beispiel mit derselben Szene um den Unterschied bessere zu erkennen.

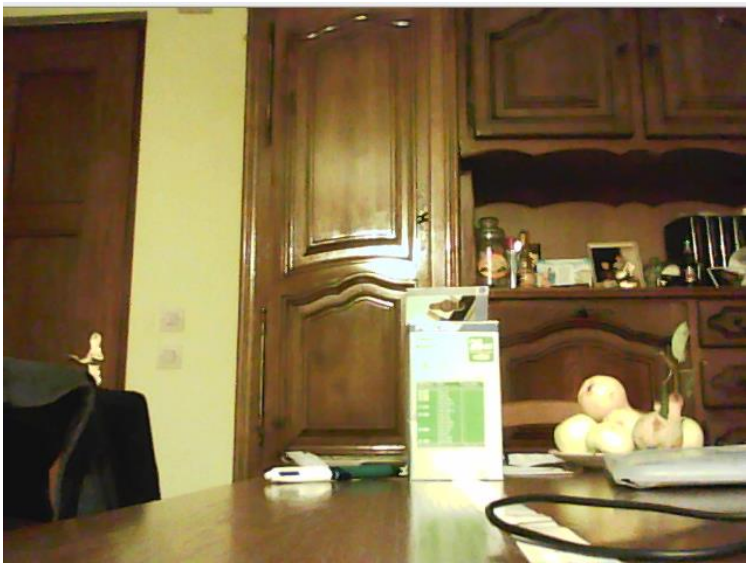


Abbildung 22: Normale Szene gesehen von der Linken Kamera ohne Rektifikation und Kalibration

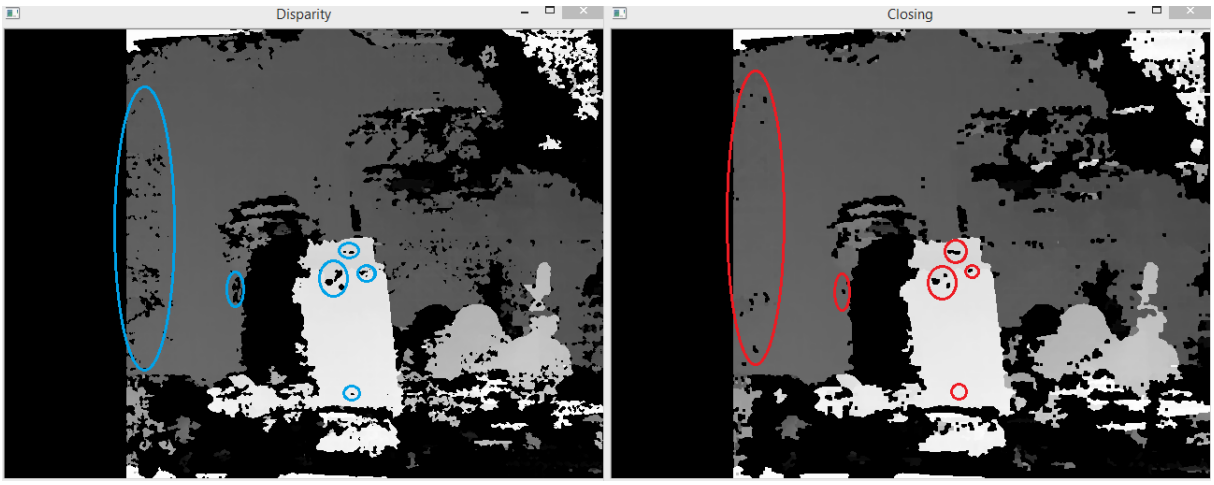


Abbildung 23: Disparität Karte von der Oberen Szene ohne Closing Filter und mit

4. Einsatz des WLS (Weighted Least Squares) Filters

Das Ergebnis ist nicht schlecht aber es ist trotzdem sehr schwer die Rande von Objekten zu erkennen wegen des Rauschens, deswegen wird ein WLS Filter eingesetzt. Als erstes müssen die Parameter des Filters in der Initialisierung festgelegt werden.

```
# FILTER Parameters for the WLS filter
lmbda = 80000
sigma = 1.8
visual_multiplier = 1.0
```

Code 9: Parametern für ein WLS Filter

Lambda wird typisch auf 8000 gesetzt, je größer dieser Wert desto mehr werden die Formen der Disparitätskarte zu den Formen des Referenz Bild angehaftet, wir haben es auf 80000 gesetzt weil es bessere Resultate mit diesem Wert gab. Sigma beschreibt wie stark unser Filter an den Randen von Objekten präzise sein muss.
Ein anderes Stereo Objekt wird erzeugt mit *cv2.ximgproc.createRightMatcher()* und basiert sich auf dem Ersten. Diese zwei Instanzen werden dann in den WLS Filter benutzt um eine Disparitätskarte zu erzeugen.
Eine Instanz des Filters wird mit der Funktion *cv2.ximgproc.createDisparityWLSFilter()* hergestellt.

```
# Used for the filtered image
stereoR=cv2.ximgproc.createRightMatcher(stereo) # Create another stereo

# Create the WLS filter
wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=stereo)
wls_filter.setLambda(lmbda)
wls_filter.setSigmaColor(sigma)
```

Code 10: Erzeugung von einem WLS Filter in Python

Um dann die Instanz des WLS Filter anzuwenden wird folgende Methode aufgerufen *cv2.ximgproc.createRightMatcher().filter()*, die Werte von unserem Filter werden dann normalisiert mit *cv2.normalize()*.

```
# Using the WLS filter
filteredImg= wls_filter.filter(displ,grayL,None,dispR)
filteredImg = cv2.normalize(src=filteredImg, dst=filteredImg, beta=0, alpha=255,
filteredImg = np.uint8(filteredImg)
```

Code 11: Einsetzung des WLS Filter in Python

Eine Ocean ColorMap wurde verwendet um eine bessere Visualisierung zu bekommen mit *cv2.applyColorMap()*. Je dunkler die Farbe desto weiter ist unser Objekt zu der Stereokamera.

Abbildung 24: Ocean ColorMap

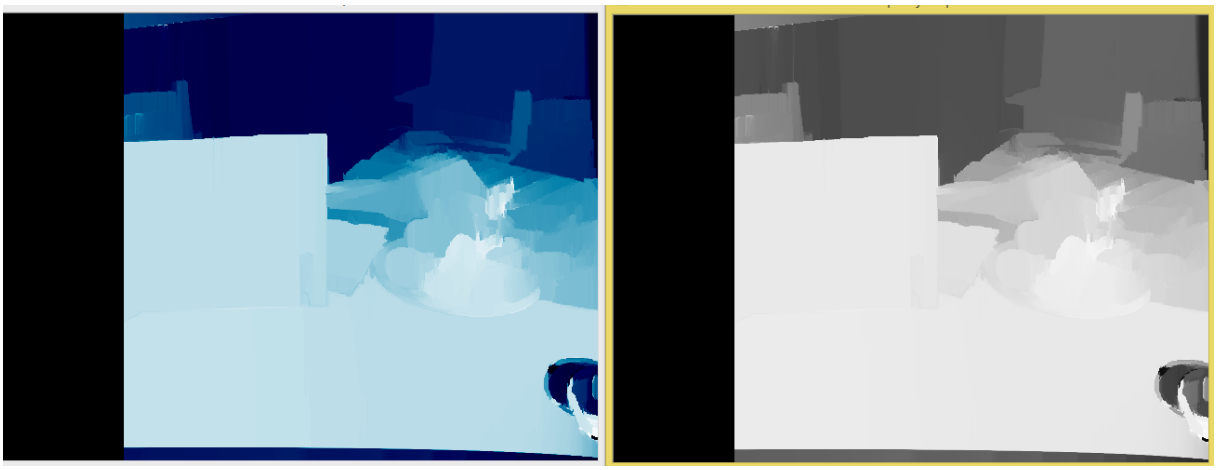


Abbildung 25: WLS Filter auf eine Disparitätskarte mit einem Ocean Map Color und ohne

Auf diese Weise bekommen wir ein Bild, dass gut die Rande zeigen kann aber nicht mehr präzise genug ist und nicht mehr die gute Disparität Werte enthält (Die Disparitätskarte ist in float32 kodiert aber das WLS Resultat ist in uint8 kodiert).
Um die gute Werte zu verwenden um später den Abstand zu Objekten zu messen, werden die Koordinaten x und y des WLS filtered Bild mit einem doppelklicken genommen. Diese (x, y) Koordinaten werden dann benutzt um den Disparität Wert von der Disparitätskarte zu bekommen um dann den Abstand zu messen. Der zurückgegebene Wert ist der Mittelwert der Disparität einer 9x9 Pixel-Matrix.

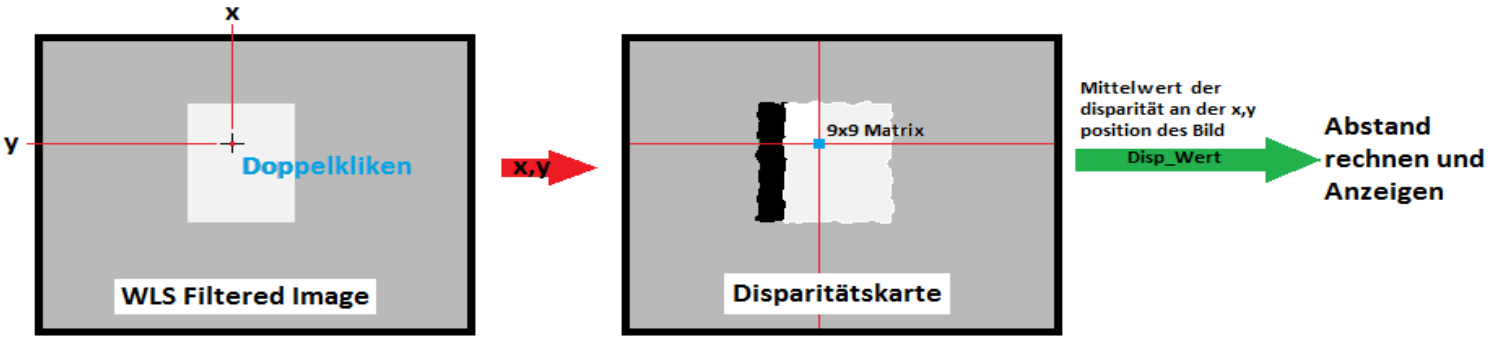
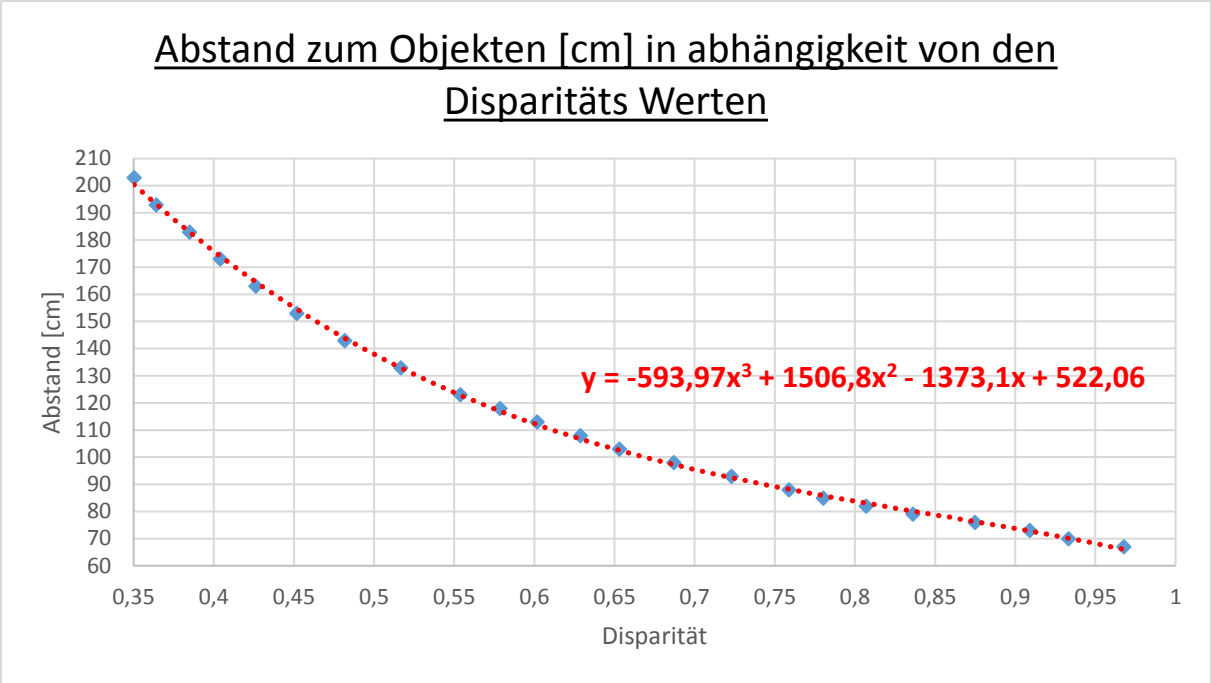


Abbildung 26: Rechnung des Abstand mit dem WLS Filter und die Disparitätskarte

5. Messen des Abstands

Nachdem die Disparitätskarte erzeugt wurde, muss der Abstand bestimmt werden. Die Arbeit besteht darin, die Relation zwischen den Disparitätwert und den Abstand zu finden. Um dies zu erledigen haben wir die Disparitätwerte an mehreren Stellen experimentell gemessen um daraus eine Regression zu bestimmen.



Geradengleichung in das Python Programm

```
# Equation for the distance mesurements
Distance= -593.97*average**(3) + 1506.8*average**(2) - 1373.1*average + 522.06
```

*Code 12: Die Regression Formel in „**Main_Stereo_Vision_Prog.py**“*

Um diese Geradengleichung der Regression zu bekommen wurde der Packlage *openpyxl* eingesetzt um die Werte der Disparität in einer Excel Datei zu speichern. Die Linien in dem Programm die dafür gesorgt haben, dass die Werte gespeichert werden, wurden in dem Programm kommentiert, aber man kann sie unkommentierten wenn man eine neue Geradengleichung braucht.
Die Abstandsmessung gilt nur von einem Abstand von 67cm bis zu 203cm um gute Ergebnisse zu bekommen. Die Präzision der Messung hängt auch mit der Qualität der Kalibrierung ab. Mit unsere Stereokameras haben den Abstand zu einem Objekt mit einer Präzision von +/ 5 cm messen können.

Abbildung 27: Experimentalen Messung der Disparität in Abhängigkeit vom Abstand zum Objekt

6. Mögliche Verbesserungen

Mögliche Verbesserungen für das Programm:

- Die Form vom WLS Filter nehmen und die, auf der Disparitätskarte zu projektieren. Diese Projektion würde dann genommen werden um alle Disparität Werte die sich in der Form befinden zu nehmen und der Wert der am häufigsten vorkommt wird dann als Wert für die gesamten Fläsche gesetzt.

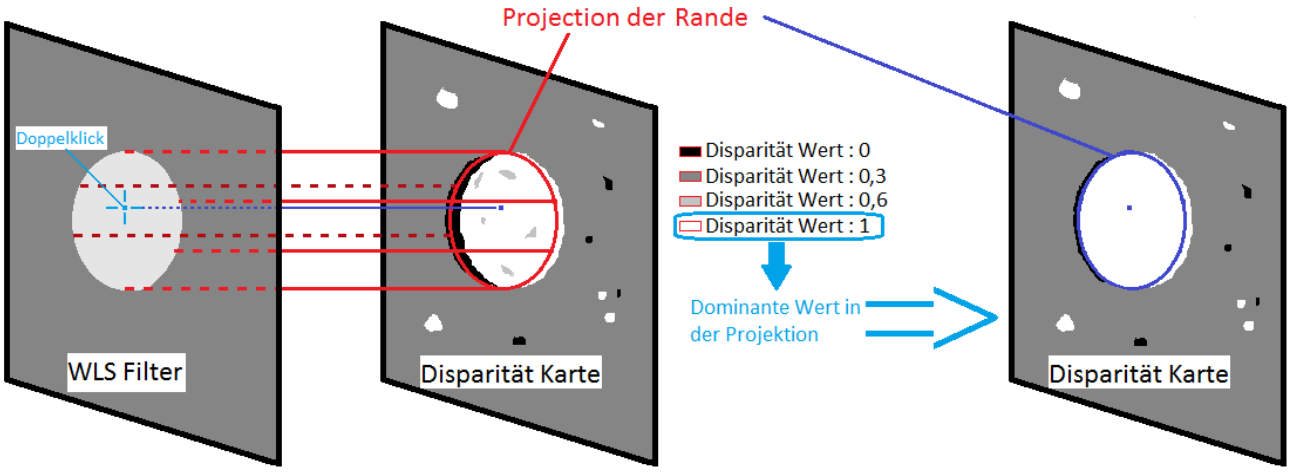


Abbildung 28:Mögliche Verbesserung mit dem Ersten Vorschlag

- Verwendung von einem Bilateral Filter auf die kalibrierten Bilder bevor die Disparitätskarte erzeugt wird, auf dieser weise wäre es möglich keinen WLS Filter einzusetzen. Dass muss überprüft werden, aber der WLS Filter dient nur um die Rändern von Objekten gut zu erkennen, vielleicht gibt es da noch ein besseren weg.

- Um die Berechnungszeit für die Erzeugung der Disparitätskarte zu verringern sollte man die kalibrierten Bilder verkleinern mit der Funktion *cv2.resize(cv2:INTER_AREA)*, man sollte dann achten dass die Werte der Essentiellen Matrizen, Fundamentalen Matrizen auch proportional verkleinert werden.

- Die Erzeugung von einer Tiefen-Karte könnte auch von Vorteil sein.

- Eine Kamera die stabiler ist als unsere wo die Rotation von den Köpfen wirklich verhindert werden kann, auf dieser weise bräuchte man nur die Werte der Matrizen von der Stereokalibrierung zu speichern um die wieder einsetzen zu können. Sehr viel Zeit könnte damit in der Initialisierung gespart werden.

- Das Programm auf dem GPU laufen zu lassen würde uns auch ermöglichen glättere Bilder zu bekommen, wenn die Stereokamera in Bewegung ist

5. Schlussfolgerung

A. Zusammenfassung

Diese Projektarbeit hat uns ermöglicht mit der Python Sprache zu arbeiten und mehr über die OpenCV Bibliothek zu lernen. Wir haben die Gelegenheit gehabt, uns mit dem Thema Stereo Vision in diesem Projekt zu vertrauen. Ein Neues Thema, dass uns beide sehr interessiert und dass noch sehr neu ist in Vergleich zu anderen Abstandsmessung Techniken die verfügbar sind. Es existieren auch andere Methoden um den Abstand mit Bildverarbeitung zu messen, z.B Time of Light(TOF) Kameras, die allerdings teuer sind und mit der es sehr wenig Dokumentation gibt. Wir haben uns auch entschieden einfache Kameras zu benutzen wegen dem Preis.

B. Fazit

In der hier ausgeführten Projektarbeit ist es mit dem entwickelten Programm möglich ein Abstand zu Rechnen auf der Basis von einer Disparität Karte.

C. Ausblick

Damit die gerechneten Abstandswerte immer korrekt bleiben, sollte man ein neues System für die Kameras entwickeln das alle freie Bewegungen der Kameras vermeidet. Somit würden nur Matrizen Werte verwendet um die Kalibration schneller durchzuführen. Die benutzte Geradengleichung würde immer den exakten Abstand zum Objekt zurückgeben mit einer größeren Präzision, mit weniger Aufwand.

Literaturverzeichnis

OpenCV-Python Tutorials

OpenCV Dokumentation

Stack Overflow

RDMILLIGAN auf seine Internet Seite rdmilligan.wordpress.com

Stereo Vision: Algorithms and Applications von Stefano Mattoccia, Department of Computer Science(DISI), University of Bologna

Stereo Matching von Nassir Navab und Slides prepared von Christian Unger

Oreilly Learning OpenCV

6. Anhang

Video zu unserem Projekt: https://youtu.be/xjx4mbZXaNc
Die Python Programme können im Ordner „**Python_Prog_Stereo_Vision**“ gefunden werden. Dieser Besteht aus den Programmen**:**

- **Take_images_for_calibration.py**”
- ”**Main_Stereo_Vision_Prog.py**”

Die zwei Programme in einer **.txt** Version können auch im selben Ordner gefunden werden wenn man nicht Python auf seinem Computer installiert hat.