



HOCHSCHULE LANDSHUT
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

FAKULTÄT INFORMATIK

Studienarbeit

IMPLEMENTIERUNG EINES ALGORITHMUS ZUR
STEREO-REKTIFIZIERUNG GEMÄSS FUSIELLO ET AL.

Projekt im Rahmen der Vorlesung Bildverstehen

Autor: Christian Froschauer

Datum: 21. Januar 2020

Dozent: Prof., PhD Andreas Siebert

Zusammenfassung

In dieser Arbeit wird das Vorgehen bei der Implementierung eines Algorithmus nach "A compact algorithm for rectification of stereo pairs" (Fusiello et al) zur Rektifizierung von Stereo-Bildern beschrieben. Anhand einer Auswahl selbstgemachter Beispielaufnahmen wird die Implementierung angewandt und überprüft. Die implementierte Rektifizierung liefert nicht das gewünschte Ergebnis.

Inhaltsverzeichnis

1 Einführung	1
1.1 Grundlagen	1
1.2 Vorgehensplanung	2
2 Umsetzung	4
2.1 Überblick Projektstruktur	4
2.2 Kalibrierung	4
2.2.1 Verwendete Kamera	4
2.2.2 Kalibrierungsmuster	5
2.2.3 Fotos	6
2.2.4 Implementierung der Kalibrierung	6
2.3 Epipolarlinien	11
2.4 Rektifizierung	12
2.4.1 Eingabeparameter	12
2.4.2 Algorithmus nach Fusiello et al	12
2.4.3 Details zur Implementierung des Algorithmus	15
2.4.4 Transformation der Bilder	16
2.5 Probleme und Lösungsansätze	17
2.5.1 Kalibrierungsproblem	17
2.5.2 Fehlerhafte Rektifizierung	18
3 Abschließendes	22
3.1 Offenes	22
3.2 Persönliches Fazit	24

1 Einführung

1.1 Grundlagen

In der Projektionsgeometrie werden Kameras mit einer 3×4 Matrix beschrieben. Diese Matrix wird im Folgenden als Projektionsmatrix P bezeichnet. Ein Objektpunkt der realen Welt X wird mit Hilfe von P auf einen Bildpunkt x abgebildet.

$$x = P * X$$

[HZ04, S. 158]

Bei der Betrachtung von zwei Bildern, die das gleiche Objekt zeigen, wird ein Objektpunkt X dementsprechend auf einen Bildpunkt x in der einen Bildebene und einen Bildpunkt x' in der zweiten Bildebene abgebildet.

Der zum Bildpunkt x gehörige Objektpunkt X kann nur auf einer Geraden l' im Bild 2 abgebildet sein. Diese Gerade l' bezeichnet man als Epipolarlinie. Für alle Bildpunkte x in Bild 1 gibt es somit Epipolarlinien in Bild 2.

Alle Epipolarlinien treffen sich in einem Punkt dem sogenannten Epipol e' . Der Epipol liegt auf der Bildebene von Bild 2, aber nicht unbedingt im Bild. Die Epipole e und e' in den Bildern 1 und 2 sind gleichzeitig die Punkte an denen eine Strecke vom Projektionszentrum C der Kamera 1 zum Projektionszentrum C' der Kamera 2 die Bildebenen schneiden. Diese Strecke wird als Basislinie bezeichnet. Siehe Abbildung 1.1.

In einem besonderen Fall, wenn die Bildebenen parallel zur Basislinie sind und die Projektionszentren auf einer Höhe liegen, dann sind die Epipolarlinien beider Bilder horizontal und parallel. Jedes Bilderpaar kann so transformiert werden, sodass die Epipolarlinie parallel und horizontal in beiden Bildern sind. Dieses Vorgehen wird Rektifizierung genannt. [FTV00, S. 17]

Die Rektifizierung ist ein wichtiger Schritt bei der Suche nach korrespondierenden

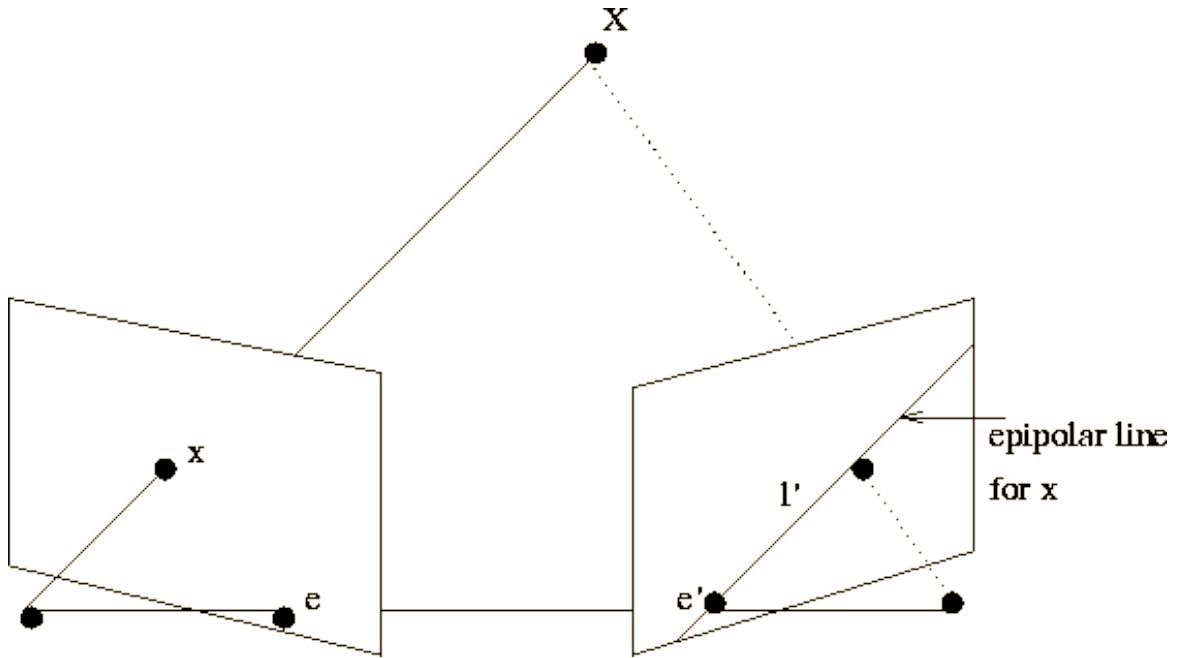


Abb. 1.1: Epipolargeometrie [Owe]

Punkten. Um den Lösungsraum einzuschränken findet die Suche nach korrespondierenden Punkten auf den Epipolarlinien statt. Ein effizienterer Algorithmus kann erreicht werden indem zuerst die Eingabebilder rektifiziert werden, sodass korrespondierende horizontale Abtastlinien genau die Epipolarlinien sind. [Sze10, S. 538] Das bedeutet vor dem Anwenden eines Korrespondenzverfahrens sollte das Bilderpaar einer Rektifizierung unterzogen werden.

In dieser Arbeit wird die Implementierung eines Algorithmus nach Fusiello et al zur Rektifizierung eines Bilderpaars beschrieben.

1.2 Vorgehensplanung

Zuerst wird ausgearbeitet wie die Herangehensweise für die Implementierung des Algorithmus aussehen kann.

Um den Algorithmus auf ein Bilderpaar anwenden zu können müssen folgende drei Vorbedingungen geschaffen werden:

- ein Bilderpaar I_1 und I_2 , welche das gleiche Objekt zeigen
- die Projektionsmatrizen P_1 und P_2 für die beiden Bilder und ihrer zugehörigen Kamera

- eine Möglichkeit die berechneten Transformation T1 und T2 auf das Bildera paar anzuwenden

Die Bilder I1 und I2 werden mit einer eigenen Kamera aufgezeichnet. Dabei sollen die Bilder von verschiedenen Position und Winkeln aufgezeichnet sein, damit eine Rektifizierung der Bilder überhaupt nötig ist. Bilder die aus der gleichen Kameraebene, aus gleicher Höhe und ohne Winkeländerung aufgezeichnet sind, haben bereits parallele Epipolarlinien auf der gleichen Ebene. Das würde eine Rektifizierung überflüssig machen.

Die Projektionsmatrizen erhält man nach [FTV00] aus einer Kalibrierung. Um die Kalibrierung vorzunehmen, soll die Programmbibliothek OpenCV verwendet werden. OpenCV ist eine quelloffene Softwarebibliothek für Computer Vision und Machine Learning. [ope]

Das Anwenden der Transformationen auf die Bilder kann ebenfalls mit OpenCV umgesetzt werden.

Durch diese Vorbedingungen wird das folgende Vorgehen geplant:

1. Kalibrierung
 - Bilderpaare vorbereiten
 - Bestimmen von korrespondierenden Punkten in den Bildern
 - Projektionsmatrizen der Kamera durch Kalibrierung berechnen
2. Visualisierung der Epipolarlinien auf den original Bildern
3. Rektifizierung
 - Implementieren des Algorithmus nach Fusiello et al
 - Anwenden des Algorithmus auf Bilderpaare
 - Transformieren der Bilder mit den Transformationsmatrizen aus dem Algorithmus
4. Visualisierung der Epipolarlinien auf den transformierten Bildern

2 Umsetzung

2.1 Überblick Projektstruktur

.../res	Alle Bilder mit denen gearbeitet wurde
.../res/calibration	Eingabebilder für die Implementierung in einer Größe von 1200 * 800 Pixel
.../res/calibration_output	Während des Kalibrierungsprozesses produzierte Bilder
.../res/rectification_output	Für Rektifizierung produzierte Bilder
.../src	Quellcode des Projekts
.../src/com/company	Hauptklassen der Implementierung Main.java, Calibration.java, Epipolar.java, Rectificaiton.java
.../src/com/company/results	Selbsterstellte Klassen zur Parameterrückgabe
.../src/result_checking_files	Textdateien in denen Zwischenergebnisse zur manuellen Überprüfung festgehalten werden
.../octave	Skripte mit Matlab-Code für GNU Octave

2.2 Kalibrierung

2.2.1 Verwendete Kamera

Zur Aufnahme der Bilder wird eine Systemkamera verwendet mit einem Zoomobjektiv. Das Zoomobjektiv hat eine Brennweite von 16 bis 50 mm. Um eine bessere Vergleichbarkeit und Kalibrierung der Bilder zu ermöglichen werden alle Bilder für das Projekt mit einer Brennweite von 16 mm aufgezeichnet.

Andere Einstellungen sollen auch möglichst konsistent gehalten werden, auch wenn sie keinen direkten Einfluss auf die Bestimmung der Kameramatrizen haben sollten. So wird eine Blendenöffnung von F/9.0 gewählt, um einen möglichst großen Bereich der Bilder scharf aufzunehmen.

Dabei steht F/9.0 dafür, dass die Brennweite F durch 9.0 geteilt wird, um die tatsächliche Blendenöffnung zu ermitteln. In diesem Fall sind es $\approx 1,8$ mm. Außerdem wird eine Belichtungszeit von 1/60 s gewählt.

Der Sensor der Kamera ist im sogenannten DX-Format mit einer Größe von 23,6 mm * 15,8 mm vom Hersteller angegeben.

Die Bilder, die mit der Kamera aufgenommen werden, liegen im JPEG-Format mit einer Auflösung von 6000 * 4000 Pixeln vor. Aus Laufzeitgründen wurden die Bilder allerdings vor der Verarbeitung herunterskaliert auf 1200 * 800 Pixel.

2.2.2 Kalibrierungsmuster

Ein Kalibrierungsmuster das von OpenCV bereitgestellt wird, wird auf ein Blatt Papier gedruckt und zur Vermeidung von Wölbungen auf einen Kartonausschnitt geklebt. Es wird das folgende Kalibrierungsmuster pattern.png aus dem OpenCV Projekt verwendet.

Die Kalibrierung von OpenCV sucht nach den Eckpunkten, an denen sich die Quadrate berühren. Aus diesem Grund wird das Muster als 9x6 Schachbrett-Muster bezeichnet. Auf dem ausgedruckten Papier messen die Quadrate genau 25,0 mm * 25,0 mm.

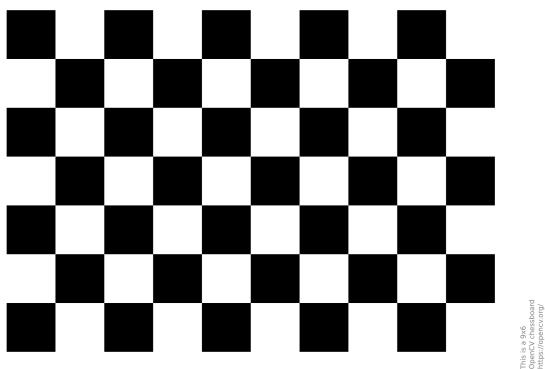


Abb. 2.1: pattern.png, 9x6 Schachbrett Muster aus OpenCV

2.2.3 Fotos

Für die Kalibrierung wird das Kalibrierungsmuster aus verschiedenen Blickrichtungen aufgenommen. Dabei entstehen zuerst 34 Kalibrierungsbilder. Die Bilder werden von $6000 * 4000$ Pixel auf $1200 * 800$ Pixel herunterskaliert.

Im Ordner `.../res/calibration` des Projekts sind die runterskalierten original Bilder platziert.

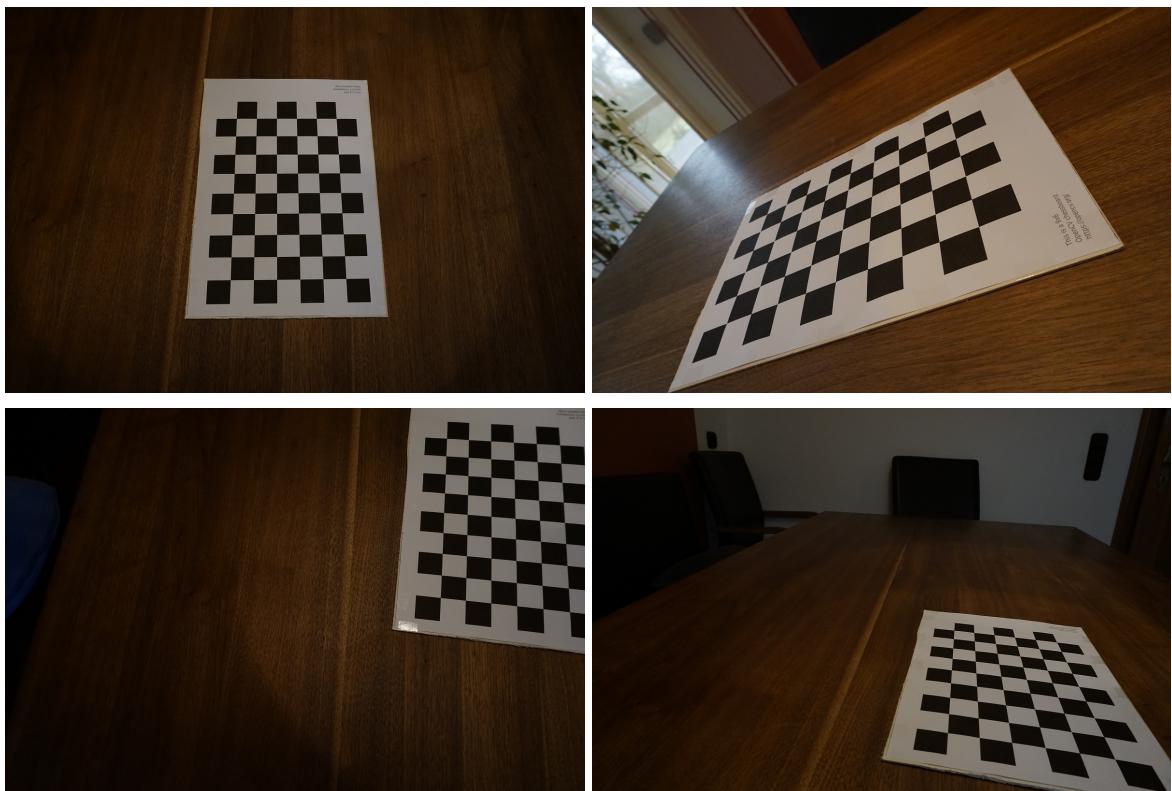


Abb. 2.2: Eine Auswahl an aufgenommenen Bildern

2.2.4 Implementierung der Kalibrierung

In der Main Klasse des Projekts werden die Bilder zunächst in den Hauptspeicher geladen. Die Klasse `Calibration` ist für die Kalibrierung der Kamera verantwortlich. Ihre Hauptfunktion ist erreichbar über eine statische Methode.

```
1  public static CalibrationResult calibrate(  
2      List<Mat> images,  
3      int indexFirst,  
4      int indexSecond)
```

Als Eingabe-Parameter wird eine Liste an Bildern gegeben, sowie die Indizes der beiden Bilder, von denen man nach der Kalibrierung die Projektionsmatrizen erhalten möchte.

Die Funktion gibt ein Objekt der Klasse `CalibrationResult` zurück welche die Informationen enthält die für die weiteren Schritte notwendig sind.

Kalibrierungsmuster aufspüren

Zu Beginn der `calibrate()`-Funktion werden die Punkte des Kalibrierungsmuster in allen Bildern gesucht.

Dies geschieht mit der Methode `Calib3d.findChessboardCorners()` der OpenCV-Bibliothek.

Bilder, bei denen die OpenCV Funktion das Kalibrierungsmuster nicht finden konnte, werden als unbrauchbar aussortiert. An dieser Stelle werden die gefunden Bildpunkte auf die Bilder gezeichnet und diese im Ordner `.../res/calibration_output` abgespeichert.

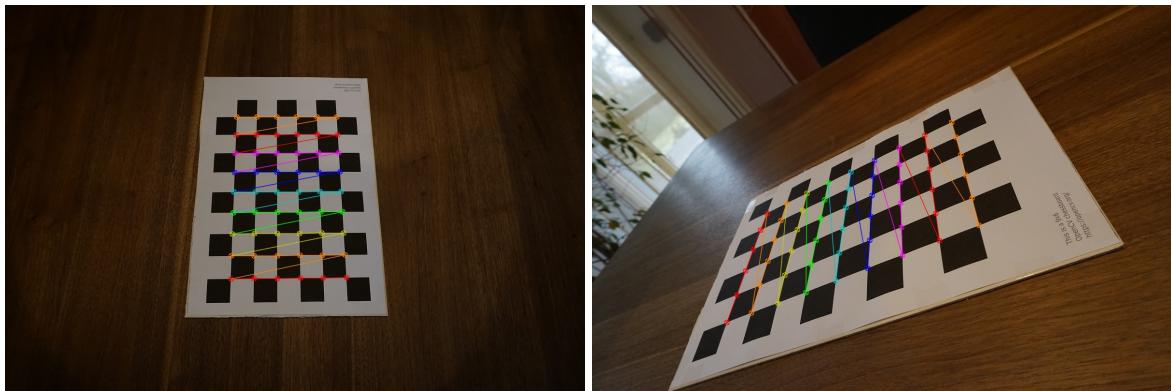


Abb. 2.3: Bilder mit eingezeichneten Bildpunkten

Außerdem werden die gefunden Bildpunkte des Kalibrierungsmuster in der Liste `imagePoints` und die echte Position der Punkte in der Liste `objectPoints` festgehalten. Dabei werden die `objectPoints` als Punkte lediglich in einer Ebene mit dem Abstand der Kalibrierungsmuster-Abmessung $0,025m$ generiert. Die Abmessung ist insofern wichtig, dass in Abschnitt 2.2.4 die Ergebnisse der Kalibrierung richtig eingeschätzt werden können.

Kalibrierungsfunktion

Daraufhin wird die Funktion `Calib3d.calibrateCamera` aufgerufen. Diese Funktion bestimmt anhand der `objectPoints` und `imagePoints` die zuvor gefunden wurden, die intrinsische Matrix der Kamera `intrinsic`, sowie einen Distortion-Vektor (`distCoeffs`) der die kameraeigene radiale und tangentiale Verzerrung beschreibt. Außerdem bestimmt sie für jedes einzelne Bild den Rotations- sowie Translationsanteil (`rvecs`, `tvecs`) der Projektionsmatrix und damit die Bausteine für die extrinsische Matrix der Kamera.

Zurück in der Implementierung wird der Distortion-Vektor auf alle Bilder angewandt und die Ergebnisse der entzerrten Bilder auch im Ordner `.../res/calibration_output` gespeichert.

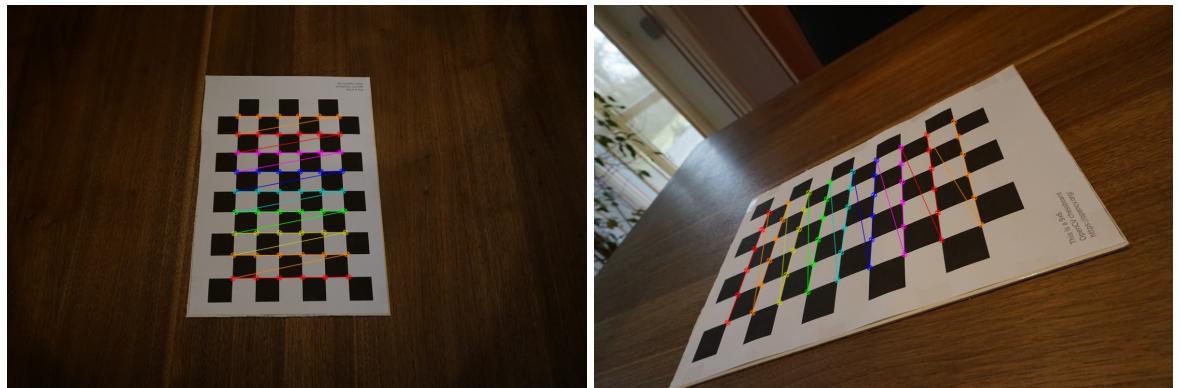


Abb. 2.4: Von Verzerrung bereinigte Bilder. Optisch sehr nahe bei Abbildung 2.3

Überprüfung der intrinsischen Matrix

Die intrinsische Kameramatrix aus der Kalibrierung sieht folgendermaßen aus:

$$\begin{bmatrix} 805.5266811314544 & 0 & 593.2955927054564 \\ 0 & 805.5299960608897 & 404.9605268365691 \\ 0 & 0 & 1 \end{bmatrix}$$

Wir bezeichnen die Parameter ebenso wie Hartley und Zisserman:

$$\begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Der Parameter α_x ist definiert als $\alpha_x = f * m_x$. Mit f als Brennweite und m_x als die Anzahl an Pixel pro Längeneinheit in x-Richtung. [HZ04, S. 157]

Die Brennweite ist wie in Abschnitt 2.2.1 angegeben $f = 16mm = 0,016m$.

Für den Parameter m_x wird die Pixelanzahl in x-Richtung (nach der Skalierung 1200) durch die Sensorbreite (23,6 mm) geteilt.

$$m_x = 1200 / 0,0236m \approx 50847,461/m$$

Damit sollte α_x ungefähr folgenden Wert erreichen:

$$\alpha_x = 0,016m * 50847,461/m \approx 813,56$$

Übereinstimmend damit ergibt sich für α_y folgende Rechnung.

$$\alpha_y = 0,016m * 800 / 0,0158m \approx 810,13$$

Damit liegt der Fehler von den per Hand berechneten α_x und α_y zu denen aus der Kalibrierung bei $\leq 1\%$. Der geringe Fehler könnte durch Fertigungsungenauigkeiten der Kamera und des Objektivs, sowie geringer Fehler beim Messen des Kalibrierungsmusters zustande kommen.

Der Verdreh-Parameter (skew) ist $s = 0$, wie bei den meisten normalen Kameras [HZ04, S. 157]. Der Bildhauptpunkt (principle point) liegt bei den Koordinaten x_0, y_0 mit den Werten knapp in der Mitte des Bildes [HZ04, S. 157].

Nach dieser Überprüfung wird die Kalibrierung der Kamera als erfolgreich eingeschätzt.

Berechnung der Projektionsmatrix

Zurück in der Implementierung werden für die beiden Bilder I1 und I2, die in der calibrate-Methode mit den Indizes bestimmt wurden, nun aus den Ergebnissen der Kalibrierungsfunktion die Projektionsmatrizen PPM1 und PPM2 berechnet.

Zuerst müssen die beiden Rotationsvektoren aus den rvecs in die Rotationsmatrizen R1 und R2 umgewandelt werden. Dafür wird die OpenCV-Funktion Calib3d.Rodrigues() verwendet.

Mit der untenstehenden Formel kann aus der intrinsischen Matrix A und der Rotationsmatrix R kombiniert mit dem Translationsvektor t die Projektionsmatrix berechnet werden.

$$P = A * [R|t]$$

[FTV00, S. 17]

Für die Matrix-Konkatenation und Matrix-Multiplikation werden die Methoden `Core.hconcat` und `Core.gemm` von OpenCV verwendet.

2.3 Epipolarlinien

Die Klasse `Epipolar` kann mit ihrer statischen Methode die Epipolarlinien in die Bilder einzeichnen. Durch die ersten acht Punkte Bildpunkten des Kalibrierungsmusters in den beiden Bildern wird mit der OpenCV-Methode `Calib3d.fundFundamentalMat()` die Fundamentalmatrix der beiden Bilder zueinander gefunden.

Die Fundamentalmatrix ist definiert als die Matrix die für jedes korrespondierende Punktpaar x und x' in den beiden Bildern die folgende Gleichung erfüllt:

$$x' * Fx = 0$$

Wenn x und x' korrespondierende Punkte sind, dann liegt x' auf der Epipolarlinie $l' = Fx$ für den Punkt x . [HZ04, S.245]

So kann OpenCV mit der Fundamentalmatrix F und den Bildpunkten die korrespondierenden Epipolarlinien im jeweils anderen Bild errechnen. Es werden wieder die acht ersten Punkte des Kalibrierungsmusters verwendet.

Das Berechnen der Epipolarlinien ist in OpenCV in der Funktion `Calib3d.computeCorrespondEpilines()` implementiert.

Die gefundenen Epipolarlinien werden ins jeweilige Bild gezeichnet und im Ordner `.../res/rectification_output` gespeichert.

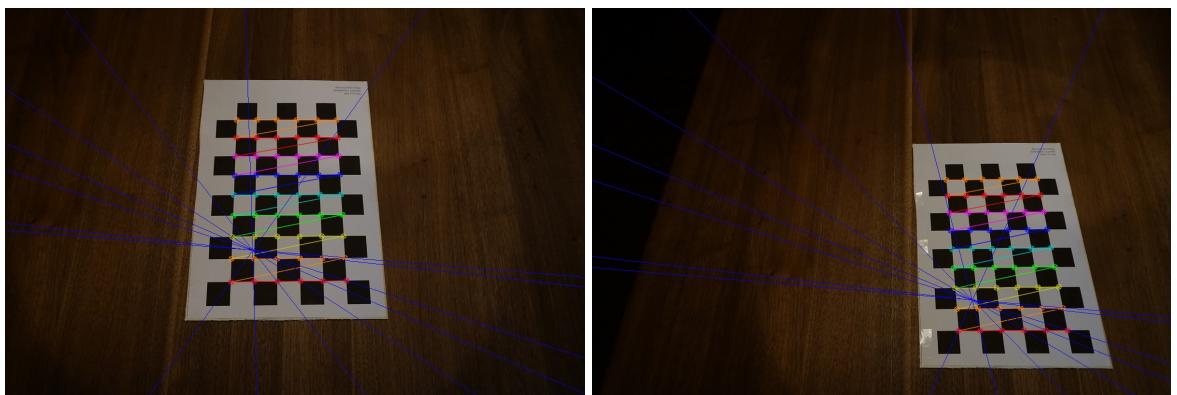


Abb. 2.5: Ein Bilderpaar und die Epipolarlinien dazu. Jede Epipolarlinie schneidet einen der ersten acht Bildpunkte.

2.4 Rektifizierung

2.4.1 Eingabeparameter

Nachdem die Vorbedingungen aus Abschnitt 1.2 erfüllt sind, können nun die Bilder rektifiziert werden. Die Rektifizierung ist in der Klasse Rectification implementiert. Die folgende statische Methode nimmt die Bilder `image1` und `image2` die rektifiziert werden sollen als Eingabeparameter. Für das Abspeichern der Ergebnisse nimmt die Funktion auch die beiden Indizes. Außerdem werden die Bildpunkte übergeben, um diese auch zu rektifizieren.

```
1 doRectification(  
2     Mat image1, Mat image2,  
3     int index1, int index2,  
4     Mat PPM1, Mat PPM2,  
5     MatOfPoint2f imagePoints1, MatOfPoint2f imagePoints2)
```

Der Rektifizierungsprozess startet mit der Methode `rectify` die nach dem Vorbild des vorgeschlagenen Matlab-Codes von Fusiello et al implementiert ist.

2.4.2 Algorithmus nach Fusiello et al

Im Folgenden ist der Programmcode des Algorithmus von Fusiello et al zur Rektifizierung implementiert in Java abgebildet. Dabei ist in den Kommentaren jeweils der vorgeschlagene Matlab-Code des Papers und darunter die Implementierung in Java mit OpenCV-Funktionen.

```
1 public static RectifyResult rectify(Mat Po1, Mat Po2){  
2  
3     // compute Rectification matrices:  
4     // skip factorize old PPMs -> got factorization as input  
5     // decomposeProjectionMatrix Calib3d.decomposeProjectionMatrix();  
6     Mat A1 = new Mat();  
7     Mat A2 = new Mat();  
8     Mat R1 = new Mat();  
9     Mat R2 = new Mat();  
10    Mat t1_new = new Mat();  
11    Mat t2_new = new Mat();  
12    Calib3d.decomposeProjectionMatrix(Po1, A1, R1, t1_new);
```

```

13 Calib3d.decomposeProjectionMatrix(Po2, A2, R2, t2_new);
14
15 // optical centers
16 // c1 = - inv(Po1(:,1:3))*Po1(:,4);
17 // c2 = - inv(Po2(:,1:3))*Po2(:,4);
18 Mat c1 = new Mat();
19 Mat c2 = new Mat();
20 Core.gemm(Po1.colRange(new Range(0, 3)).inv(), Po1.col(3), -1, new
21     Mat(), 0, c1);
22 Core.gemm(Po2.colRange(new Range(0, 3)).inv(), Po2.col(3), -1, new
23     Mat(), 0, c2);
24
25 // new x axis (= direction of the baseline)
26 // v1 = (c1-c2);
27 Mat v1 = new Mat();
28 Core.subtract(c1, c2, v1);
29 // new y axes (orthogonal to new x and old z)
30 // v2 = cross(R1(3,:)',v1);
31 Mat R1_row2_transposed = new Mat();
32 Core.transpose(R1.row(2), R1_row2_transposed);
33 Mat v2 = R1_row2_transposed.cross(v1);
34 // new z axes (orthogonal to baseline and y)
35 // v3 = cross(v1,v2);
36 Mat v3 = v1.cross(v2);
37
38 // new extrinsic parameters
39 //R = [v1'/norm(v1)      (Euklidische Norm)
40 // v2'/norm(v2)
41 // v3'/norm(v3)];
42 // translation is left unchanged
43 Mat v1_transposed = new Mat();
44 Mat v2_transposed = new Mat();
45 Mat v3_transposed = new Mat();
46 Core.transpose(v1, v1_transposed);
47 Core.transpose(v2, v2_transposed);
48 Core.transpose(v3, v3_transposed);
49 double v1_norm = Core.norm(v1);
50 double v2_norm = Core.norm(v2);

```

```

49     double v3_norm = Core.norm(v3);
50     Mat row1 = new Mat();
51     Mat row2 = new Mat();
52     Mat row3 = new Mat();
53     v1_transposed.convertTo(row1, v1_transposed.type(), 1/v1_norm);
54     // = v1' / norm(v1)
55     v2_transposed.convertTo(row2, v2_transposed.type(), 1/v2_norm);
56     v3_transposed.convertTo(row3, v3_transposed.type(), 1/v3_norm);
57     Mat R = new Mat();
58     Core.vconcat(List.of(row1, row2, row3), R);
59
60     // new intrinsic parameters (arbitrary)
61     // A = (A1 + A2)./2;
62     // A(1,2)=0; // no skew
63     Mat A_sum= new Mat();
64     Core.add(A1, A2, A_sum);
65     Mat A = new Mat();
66     A_sum.convertTo(A, A_sum.type(), 0.5); // A = A_sum / 2
67
68     // new projection matrices
69     // Pn1=A*[R-R*c1 ];
70     // Pn2=A*[R-R*c2 ];
71     Mat R_times_c1_neg = new Mat();
72     Mat R_times_c2_neg = new Mat();
73     Core.gemm(R, c1, -1, new Mat(), 0, R_times_c1_neg, 0); // -R * c1
74     Core.gemm(R, c2, -1, new Mat(), 0, R_times_c2_neg, 0); // -R * c2
75     Mat bracket1 = new Mat();
76     Mat bracket2 = new Mat();
77     Core.hconcat(List.of(R, R_times_c1_neg), bracket1); // [R | -R*c1]
78     Core.hconcat(List.of(R, R_times_c2_neg), bracket2); // [R | -R*c2]
79     //WRONG: Core.subtract(R, R_times_c2, bracket2); // [R -R*c2]
80
81     // new projection matrices:
82     Mat Pn1 = new Mat();
83     Mat Pn2 = new Mat();
84     Core.gemm(A, bracket1, 1, new Mat(), 0, Pn1, 0); // A * bracket1
85     Core.gemm(A, bracket2, 1, new Mat(), 0, Pn2, 0); // A * bracket2
86

```

```

87     // rectifying image transformation
88     // T1 = Pn1(1:3,1:3)* inv(Po1(1:3,1:3));
89     // T2 = Pn2(1:3,1:3)* inv(Po2(1:3,1:3));
90     Mat PPM1_sub_col = Po1.colRange(new Range(0, 3));
91     Mat PPM2_sub_col = Po2.colRange(new Range(0, 3));
92     Mat PPM1_sub = PPM1_sub_col.rowRange(new Range(0, 3));
93     Mat PPM2_sub = PPM2_sub_col.rowRange(new Range(0, 3));
94     Mat Pn1_sub_col = Pn1.colRange(new Range(0, 3));
95     Mat Pn2_sub_col = Pn2.colRange(new Range(0, 3));
96     Mat Pn1_sub = Pn1_sub_col.rowRange(new Range(0, 3));
97     Mat Pn2_sub = Pn2_sub_col.rowRange(new Range(0, 3));
98     Mat T1 = new Mat();
99     Mat T2 = new Mat();
100    Core.gemm(Pn1_sub, PPM1_sub.inv(), 1, new Mat(), 0, T1, 0); //  

101      Pn1(3x3) * inv(Po1(3x3))  

102    Core.gemm(Pn2_sub, PPM2_sub.inv(), 1, new Mat(), 0, T2, 0); //  

103      Pn2(3x3) * inv(Po2(3x3))  

104  }

```

2.4.3 Details zur Implementierung des Algorithmus

In den Zeilen 3-13 werden die beiden Projektionsmatrizen Po1 und Po1 wieder zurück Faktorisiert zur bereits Bekannten Form:

$$P = A * [R|t]$$

[FTV00, S. 17]

Alternativ können natürlich auch die intrinsische Kameramatrix, die Rotationsmatrix und Translationsvektoren, die direkt aus der Kalibrierung stammen, in die Funktion übergeben werden. Allerdings ist hier die Entscheidung darin begründet der original Implementierung möglichst treu zu bleiben.

Für Matrix Multiplikation wird die Methode `Core.gemm()` von OpenCV verwendet. In den Zeilen 20 und 21 wird hier außerdem noch ein Skalar von -1 für die Negierung des Ergebnisses verwendet.

In den Zeilen 53-56 sowie 66 wird die Multiplikation einer Matrix oder eines Vektors mit einem Skalar durch die Funktion `convertTo()` durchgeführt, da diese die Eingabe eines Skalierungsfaktors ermöglicht.

Als Rückgabewert dient ein Objekt der Klasse `RectifyResult` das die beiden Homographien zur Transformation der Bilder T1 und T2 hält, sowie die neuen Projektionsmatrizen Pn1 und Pn2 für die transformierten Bilder.

2.4.4 Transformation der Bilder

Im Anschluss an die `rectify()`-Methode können die Transformationen auf die Bilder angewandt werden. Mit der Funktion `Imgproc.warpPerspective()` werden die Homographien T1 und T2 auf die Bilder angewandt, mit der Funktion `Core.perspectiveTransform()` werden sie auf die Bildpunkte angewandt. Dies wird getan, um die transformierten Bildpunkte im Anschluss für das Einzeichnen der Epipolarlinien in die transformierten Bilder zu verwenden.

Wie die Epipolarlinien für die Originalbilder wird nun aus der `Main`-Klasse erneut die `outputImagesWithEpipolarLines()` Methode aufgerufen, nur diesmal mit den rektifizierten Bildern und den zugehörigen rektifizierten Punkten.

Wie in Abbildung 2.6 zu sehen ist, war die Rektifizierung nicht erfolgreich. Die Epipolarlinien sind weder parallel noch horizontal. Auf Lösungsansätze für dieses Problem wird in Abschnitt 2.5.2 eingegangen.

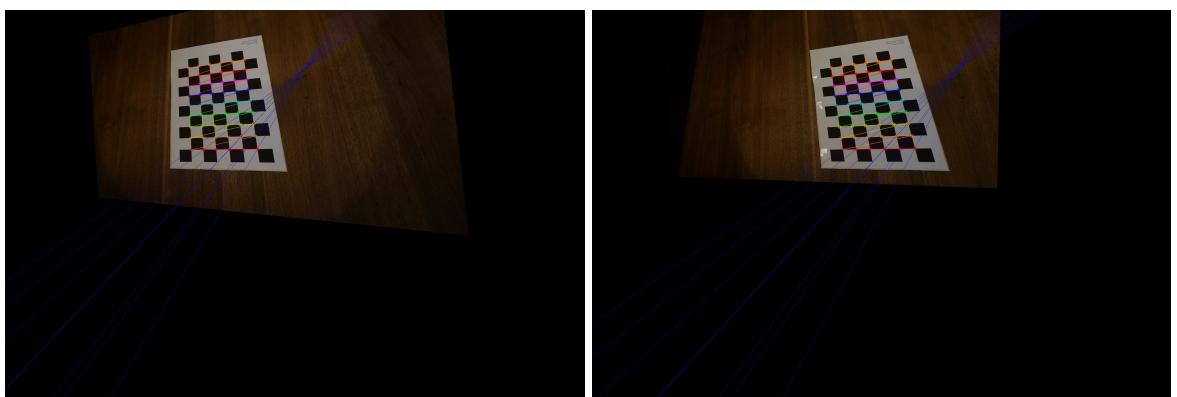


Abb. 2.6: Das Bilderpaar nach der Rektifizierung mit den Epipolarlinien die aufzeigen, dass etwas schiefgelaufen ist.

2.5 Probleme und Lösungsansätze

2.5.1 Kalibrierungsproblem

Die Kalibrierung war zu Anfangs nur wenig erfolgreich. Die Ausgabe-Bilder, bei denen die Verzerrung beseitigt werden sollte, waren besonders zum Rand hin stark verzerrt, was in Abbildung 2.7 zusehen ist. Also gerade das Gegenteil vom gewünschten Effekt.

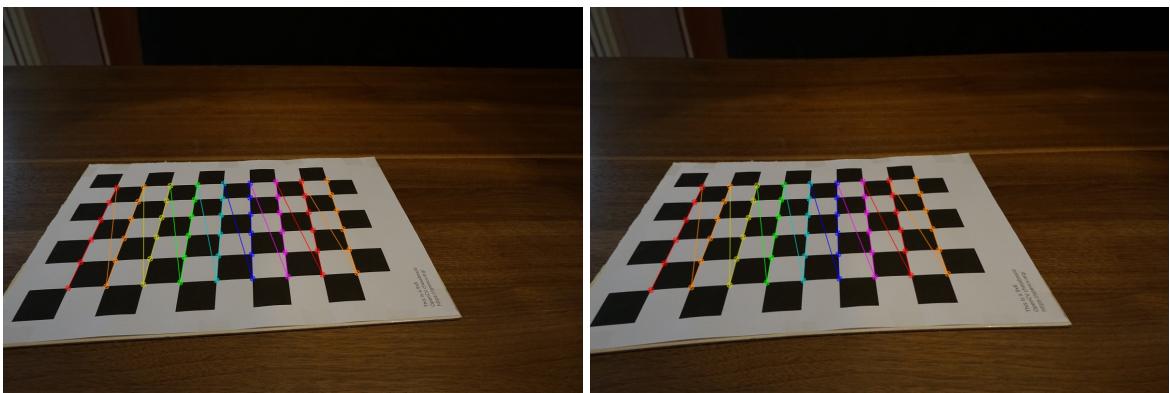


Abb. 2.7: Links das Originalbild mit eingezeichneten Bildpunkten, rechts das Bild, bei dem die Verzerrung beseitigt sein sollte. Man achte auf die Tischkante oder die Ränder des Musters.

Lösung

Es wurde entdeckt, dass einige der geschossenen Bilder im Hochformat vorlagen. Dies lag daran, dass beim Schießen der Bilder die Kamera manchmal in einem Winkel gehalten wurde, bei dem sie automatisch auf Hochformat-Aufnahmen umgeschaltet hatte. Die Implementierung hat die Kalibrierung mit diesen Bildern durchgeführt, allerdings wurde das Ergebnis massiv beeinflusst.

Nach dem Entfernen dieser Hochformat-Bilder wird die Kalibrierung wesentlich besser. Das wird zum Vergleich in Abbildung 2.8 erkennbar.

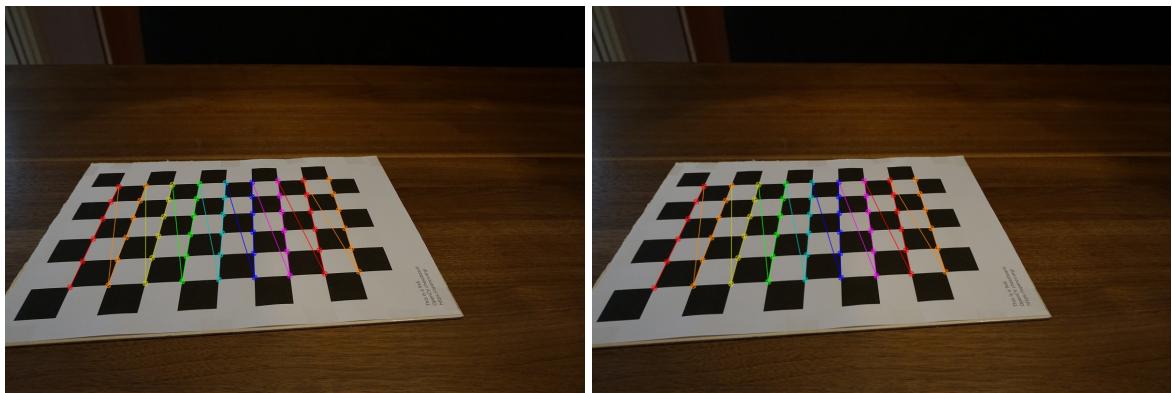


Abb. 2.8: Links das Originalbild mit eingezeichneten Bildpunkten, rechts das Bild, bei dem die Verzerrung beseitigt wurde.

2.5.2 Fehlerhafte Rektifizierung

Die Rektifizierung führt nicht zum gewünschten Ergebnis. Die Epipolarlinien nach der Rektifizierung sind weder parallel noch horizontal. Siehe Abbildung 2.9.

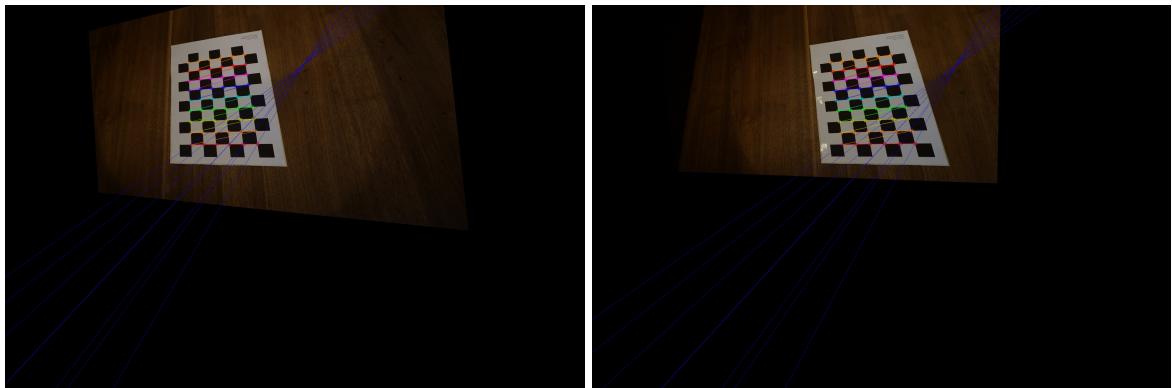


Abb. 2.9: Das Rektifizierungsergebnis des Bilderpaars mit der eignen Implementierung, schwach in Blau sind die Epipolarlinien zu sehen.

Lösungsansatz 1

Zum Vergleich mit einer Rektifizierung wie sie aussehen sollte wurde in der Klasse `Rectification` die Methode `rectifyWithOpenCVMethod()` implementiert, in der die Bilder mit den Transformationsmatrizen `T1` und `T2` der OpenCV Funktion `Calib3d.stereoRectifyUncalibrated()` rektifiziert werden. Dafür werden im Grunde nur die Bildpunkte benötigt, die dazu dienen die Fundamentalmatrix zu finden und ebenso mit der Fundamentalmatrix zusammen als Eingabe in die

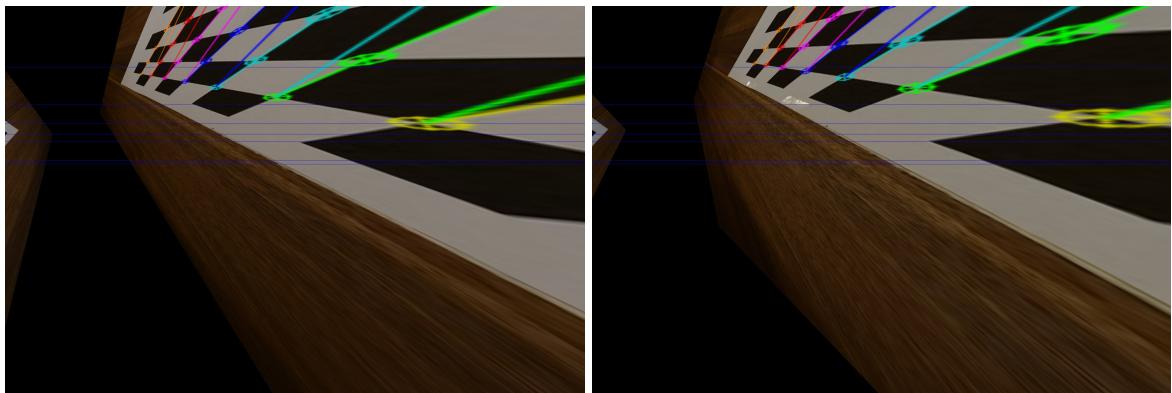


Abb. 2.10: Das Rektifizierungsergebnis des gleichen Bilderpaars durch OpenCV, schwach in Blau sind die parallelen, horizontalen und auf gleicher Höhe liegenden Epipolarlinien zu sehen.

OpenCV-Methode kommen. In den Abbildungen 2.9 und 2.10 wurden ein und dieselben Ausgangsbilderpaare transformiert.

Lösungsansatz 2

Als zweiter Ansatz wird die eigene Implementierung des Algorithmus nach Fusello et al versucht zu verifizieren. Der Algorithmusvorschlag für Matlab aus dem Referenzartikel wird mit der OpenSource Implementierung GNU Octave ausgeführt. Dafür werden die Eingabeparameter für die Funktion `rectify()` genommen und als Eingabeparameter in das Matlab-Skript gegeben. Für die in Abbildung 2.9 gezeigten Bilder sind das die folgenden Projektionsmatrizen.

```

1 Po1 =
2 [-820.9809842647142, 184.2182273632835, 541.2270896040269,
3   297.8836542999622;
4   -5.861447464327993, -626.5469693685785, 648.2870825940317,
5   237.8146008448999;
6   -0.02655195525547172, 0.330191152363302, 0.9435405643495703,
7   0.4219857032493103]

5
6 Po2 =
7 [-784.139169323909, 262.6948308986451, 563.0187237468371,
8   438.9005201336228;
9   25.15067928006231, -546.409852620379, 716.7114615664634,
10  296.3523474088649;
```

```
9  0.03508731357570102, 0.4384565889552385, 0.8980672024006767,  
10 0.4438649486484741]
```

Die Ausgaben der Java-Implementierung und der Matlab-Implementierung sind weitgehend identisch. Daraus wird gefolgert, dass die Implementierung in Java soweit mit der Vorgabe von Fusiello et al übereinstimmt.

Der Fehler muss wohl im Kalibrierungsschritt der Implementierung liegen.

```
1 Pn1_Java =  
2 [-566.8536364537156, 230.8384476020182, 791.3680940537492,  
3   389.1020880130219;  
4   195.45327657604, -620.2021513779838, 624.5149405349878, 214.0841507949395;  
5   0.3243347731428132, 0.3386505853174174, 0.8832455694735019,  
6   0.3724661874732533]  
7  
8 Pn1_Matlab =  
9   -566.85364 230.83845 791.36809 389.10209  
10  195.45328 -620.20215 624.51494 214.08415  
11  0.32433    0.33865   0.88325   0.37247
```

```
1 Pn2_Java =  
2 [-566.8536364537156, 230.8384476020182, 791.3680940537492,  
3   500.6894931895662;  
4   195.45327657604, -620.2021513779838, 624.5149405349878, 214.0841507949395;  
5   0.3243347731428132, 0.3386505853174174, 0.8832455694735019,  
6   0.3724661874732533]  
7  
8 Pn2_Matlab =  
9   -566.85364 230.83845 791.36809 500.68949  
10  195.45328 -620.20215 624.51494 214.08415  
11  0.32433    0.33865   0.88325   0.37247
```

```
1 T1_Java =  
2 [0.6766386513448619, 0.04990846035713564, 416.302097389807;  
3   -0.2492580256301796, 0.9843314348274048, 128.5485613445904;  
4   -0.0004338641378242562, -3.200155811369478e-05, 1.206954776347866]  
5  
6 T1_Matlab =
```

```

7   0.676638651  0.049908460 416.302097390
8   -0.249258026  0.984331435 128.548561345
9   -0.000433864  -0.000032002  1.206954776


---


1 T2_Java =
2 [0.7409495344650211, 0.1634428553403672, 286.2349993159664;
3   -0.2158501349546919, 1.035042787625434, 4.694057330721364;
4   -0.0003599080876156247, 0.0001081692093674509, 1.122805116133057]
5
6 T2_Matlab =
7 0.74094953  0.16344286 286.23499932
8 -0.21585013  1.03504279  4.69405733
9 -0.00035991  0.00010817  1.12280512

```

Lösungsansatz 3

Aus dem vorherigen Schritt wird die Vermutung abgeleitet, dass beim Kalibrierungsschritt der Implementierung ein Fehler gemacht wird. Um die Kalibrierung zu überprüfen, sollen die Projektionsmatrizen, die das für die Rektifizierung wichtigste Ergebnis sind, überprüft werden.

In der Theorie wird ein Punkt X der Realität über die Projektionsmatrix P der Kamera auf das Kamerabild im Punkt x abgebildet.

$$x = P * X$$

[HZ04, 6.2 The projective camera, S. 158]

In der Implementierung entspricht das, dass ein Objektpunkt aus `objectPoints` über die Projektionsmatrix auf einen Bildpunkt in `imagePoints` abgebildet wird. Aus diesem Grund wird die Methode `verifyProjectionMatrix()` in der Klasse `Calibration` implementiert. Diese Methode bildet die Objektpunkte über die Projektionsmatrix auf `destinationImagePoints` ab.

Jetzt wird der Fehler der original Bildpunkte und der abgebildeten Objektpunkte überprüft und bei einem Fehler von mehr als 1% eine Warnung ausgegeben. Die Kalibrierung scheint dieser Eigenschaft bezüglich stabil zu sein. Bei allen versuchten Testbildpaaren wurde keine Warnung ausgegeben.

3 Abschließendes

3.1 Offenes

Die Implementierung des Algorithmus war nicht erfolgreich. Bis zuletzt konnte das Problem für die fehlerhafte Rektifizierung nicht gefunden werden.

Auch offen ist ein Ansatz bei dem versucht wird für ein Bilderpaar, mit aus der Realität gemessenen Werten, die extrinsische Matrix zu errechnen. Die Rechnungen sollten nach dieser Anleitung geschehen: <http://ftp.cs.toronto.edu/pub/psala/VM/camera-parameters.pdf>.

Mit der berechneten extrinsischen Matrix, der intrinsischen Matrix aus der Kalibrierung und Bildpunkten, die in den Bildern gemessen wurden, soll daraufhin eine Rektifizierung gemacht werden.

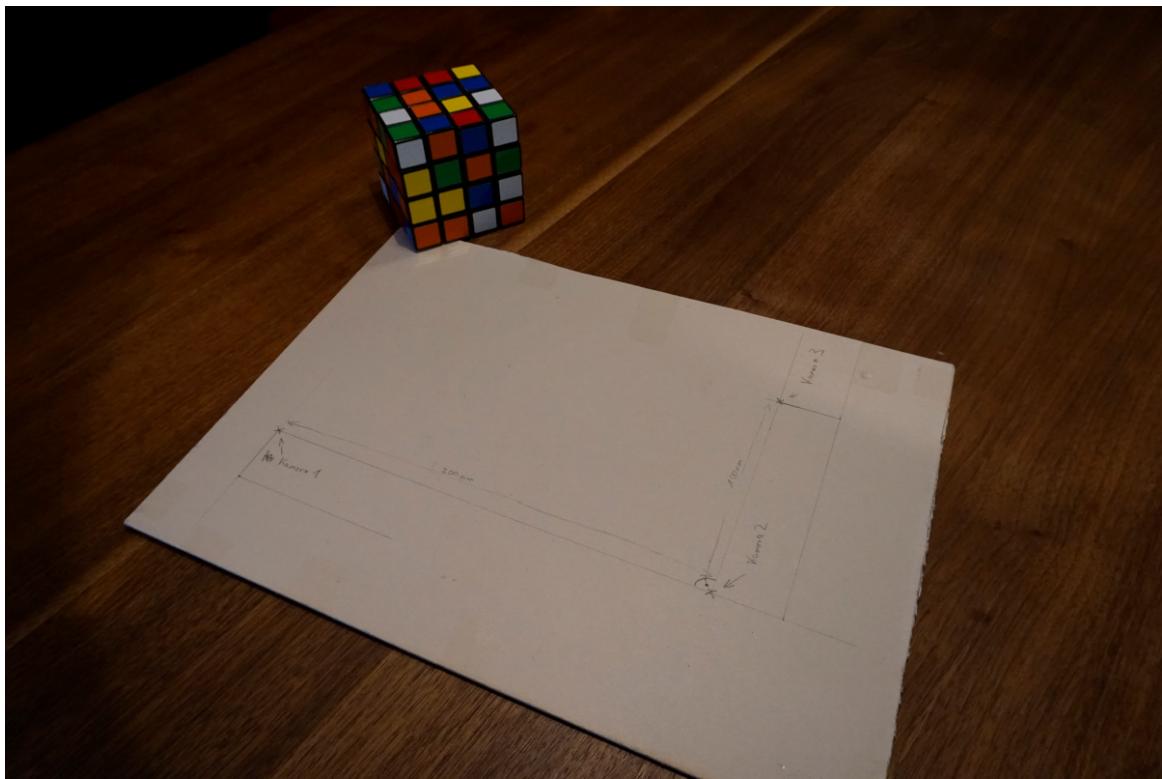


Abb. 3.1: Ausgemessene Kamerapositionen

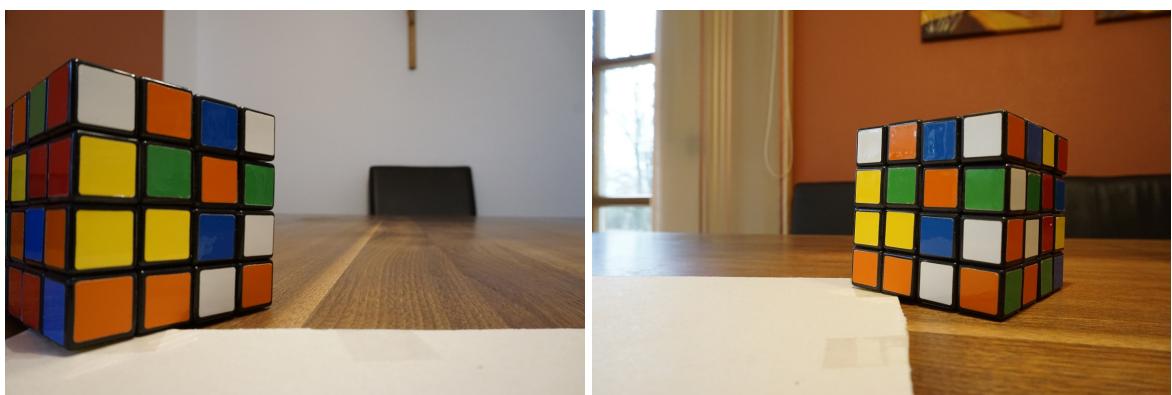


Abb. 3.2: Bilderpaar, aufgenommen aus den Position Kamera-1 und Kamera-3

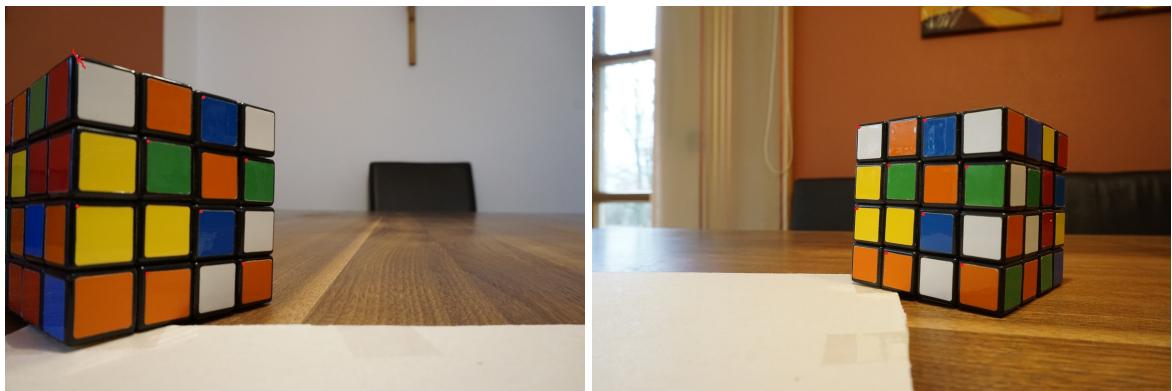


Abb. 3.3: In rot die gewählten Bildpunkte

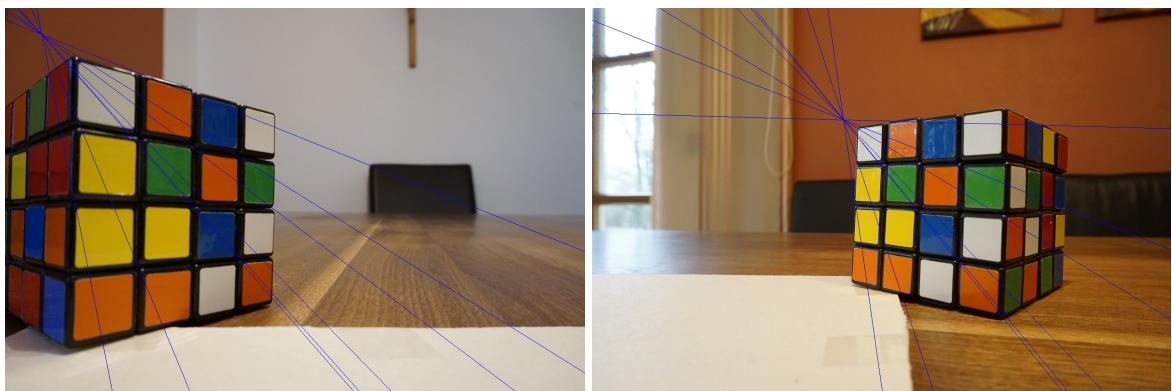


Abb. 3.4: Epipolarlinien durch die gewählten Bildpunkte

3.2 Persönliches Fazit

Die Implementierung hat mir persönlich viel Freude bereitet. In der Realität Fotos zu schießen und darauf zu experimentieren ist eine angenehme Abwechslung zur reinen Softwareentwicklung und hat sich etwas wie ein physikalisches Experiment angefühlt. Ich konnte viel über die OpenCV-Bibliothek lernen und die Themen der Projektionsgeometrie und Epipolargeometrie aus der Vorlesung wiederholen und verinnerlichen.

Leider konnte ich die Rektifizierung nicht fehlerfrei zum Abschluss bringen, was natürlich sehr schade ist. Weitere Ideen für Lösungsansätze wären nötig um das Problem letztendlich noch aufzuspüren.

Literaturverzeichnis

- [FTV00] Andrea Fusiello, Emanuele Trucco, and Alessandro Verri. A compact algorithm for rectification of stereo pairs. *Machine Vision and Applications*, 12(1):16–22, Jul 2000.
- [HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.
- [ope] OpenCV team opencv.org. About, zuletzt zugegriffen: 19.01.2020.
- [Owe] R. Owens. Epipolar geometry, zuletzt zugegriffen: 20.01.2020.
- [Sze10] R. Szeliski. *Computer Vision: Algorithms and Applications*. Texts in Computer Science. Springer London, 2010.