HAW Landshut

Faktorisierungsalgorithmus zur Affinen Rekonstruktion von Kameras und 3d-Punkten

Studienarbeit im Fach Bildverstehen

Tobias Weiden
12.7.2019

Prof. Siebert

Inhalt

Einleitung 2
Kameras und Projektion 3
Perspektivische Projektion 3
Lochkamera 4
Orthogonalprojektion 5
Affine Kamera 6
Affine Rekonstruktion – Faktorisierungsalgorithmus 7
Implementierung 9
Vorbereitung 9
Matrix-Berechnung mit OpenCV 11
Matrizen-Berechnung mit Faktorisierungs-Algorithmus 12
Abgleich mit zusätzlichem Muster 14
Anpassungen 16
Testergebnisse 17
Zeitunterschiede 17
Reprojektionsfehler 17
Fazit 19
Literaturverzeichnis 20
Abbildungsverzeichnis 20

Einleitung

Eine Rekonstruktion von 3-dimensionalen Punkten aus 2D-Bildern ist ein weitgehend erforschtes Feld. Hierbei gibt es gerade in einer immer weiter automatisierten Welt immer mehr Use-Cases. Ob selbstfahrende Autos, Roboter, die mit Menschen interagieren, oder einfache Objekt-/ oder Personen-Analysen; Für immer mehr Nutzfelder ist es interessant ein dreidimensionales Modell bspw. der Umwelt zu konstruieren. Hierbei werden verschiedenste Präferenzen gesetzt: Zum einen soll ein Modell möglichst genau oder möglichst schnell erstellt werden, um darauf zu reagieren.

Die meisten Methoden zur Erstellung eines 3D-Modells werden durch eine Verarbeitung von mehreren Bildern aus unterschiedlichen Perspektiven, auf denen bestimmte und bekannte Bildpunkte vorhanden sind.

Hierbei werden Kamera-Matrizen erstellt, die 3D-Punkte in 2D-Punkte wandeln und umgekehrt.

Im Folgenden wird eine Spezialisierung von Kamera-Modellen (Affine Kamera) erläutert, mit dessen Hilfe ein einfacher Algorithmus zur Erstellung von Kamera-Matrizen durchgeführt wird. Dies ist der Faktorisierungsalgorithmus zur Affinen Rekonstruktion von Hartley & Zisserman.

Dieser wird dann mit qualitativ und zeitlich mit der Standard-Kalibrierung von OpenCV verglichen. Hierzu wird zum einen der direkte Reprojektions-Fehler verglichen, welcher die durchschnittliche Differenz zwischen den Original-Punkten und den reprojezierten Punkten darstellt, und der Reprojektions-Fehler bei verschobenen 3D-Punkten. Dies wird mit selbstaufgenommenen Bildern aus verschiedenen Distanzen verglichen.

Die Implementierung erfolgte mittels Python und auch für die OpenCV-Kalibrierung wurde die offizielle Python-Bibliothek genutzt.

Kameras und Projektion

Durch Projektion kreiert eine projektive Kamera ein flaches Bild aus dreidimensionalen Informationen. Es werden also 3D-Punkte $(X, Y, Z)^T$ in 2D-Punkte $(x, y)^T$ umgewandelt. [1]

Perspektivische Projektion

Um die Tiefe des dargestellten Raumes zu begreifen, vergleichen wir bei einem zweidimensionalen Bild kameranahe Objekte mit kamerafernen Objekten (im Hintergrund). Objekte mit gleicher Größe in der realen Welt werden je nach Nähe zur Kamera größer (nah der Kamera) oder kleiner (fern der Kamera) dargestellt.

Liegt nun ein Objekt „entlang“ der Tiefe, so wird dieses der Tiefe hin kleiner. Im folgenden Beispiel wird die Bande kleiner, je weiter sie von der Kamera entfernt ist:

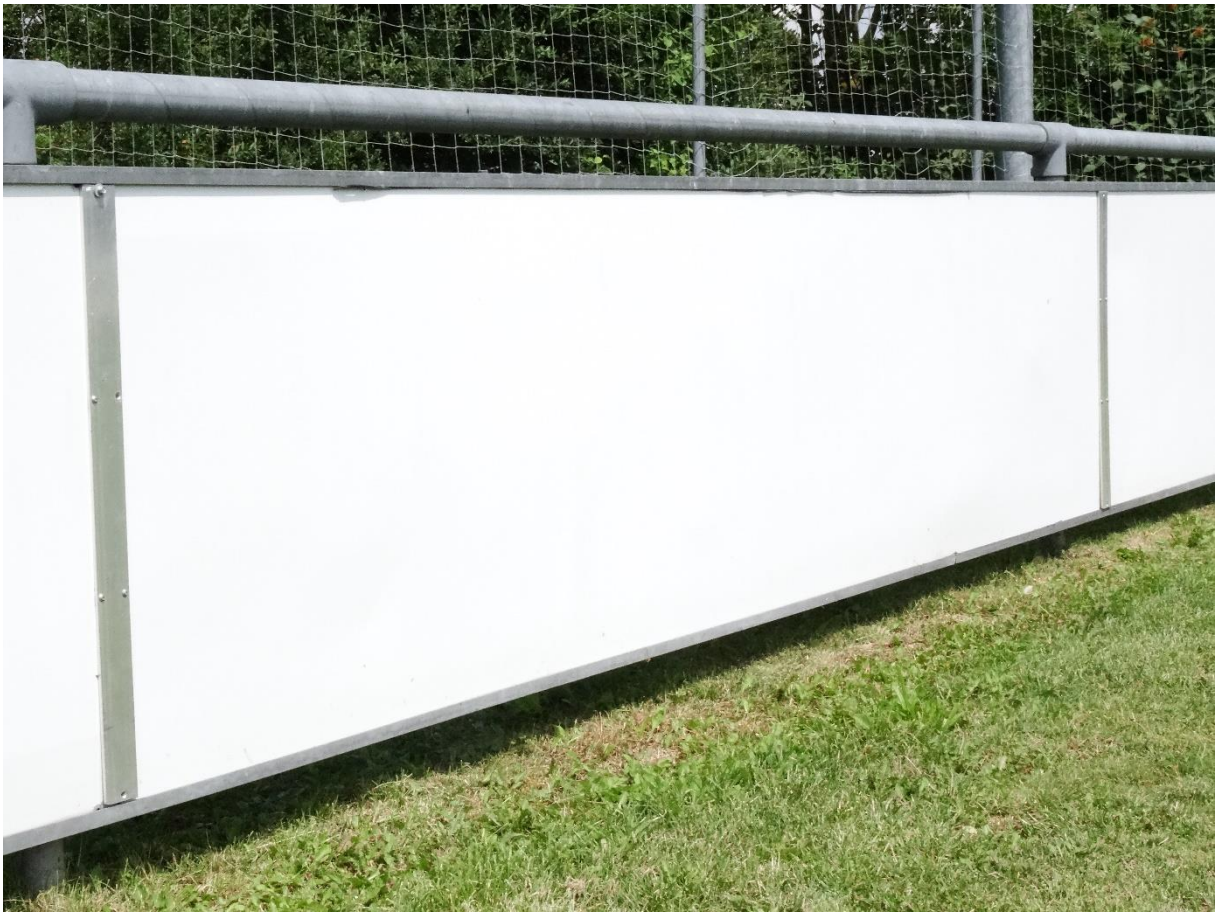

Abbildung 1: Bande, 5 Meter entfernt der rechten senkrechten Spange

Ein Bild entlang einer Bande. Der linke Teil war näher der Kamera und wird somit größer dargestellt als der rechte Teil. Dadurch lässt sich die Tiefe für den Betrachter erahnen.

In diesem Bild tritt auch die Perspektivische Projektion auf: Linien, die in der echten Welt parallel verlaufen, konvergieren im Bild. Diese laufen im Fluchtpunkt zusammen. Dies kann man am Beispiel an dem oberen und unteren Ende der Bande nachvollziehen, wie im Folgenden verdeutlicht: [1]

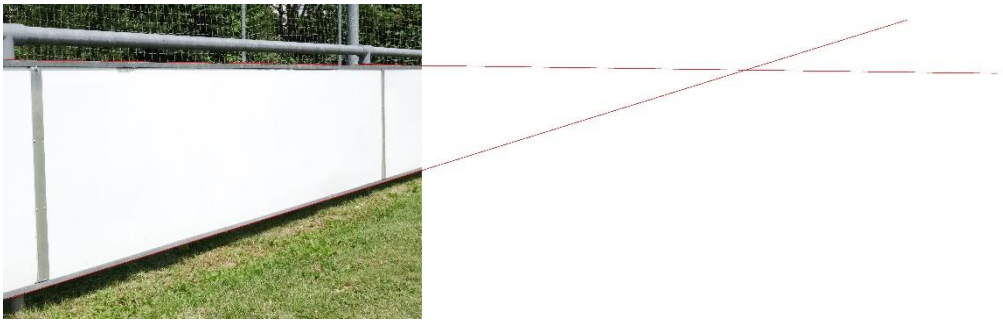

Abbildung 2: Bande mit eingezeichnetem Fluchtpunkt, 5 Meter Entfernung

Lochkamera

Wir gehen von einer zentralen Projektion von Punkten im Raum auf eine Ebene aus. Wenn das Zentrum der Projektion C der Ursprung des euklidischen Koordinatensystems ist, so gibt die Brennweite f dem Abstand zwischen Ursprung und Bildebene auf der Ebene z wieder, wie in Abbildung 3: Skizze einer Lochkamera [2] zu sehen. [2]

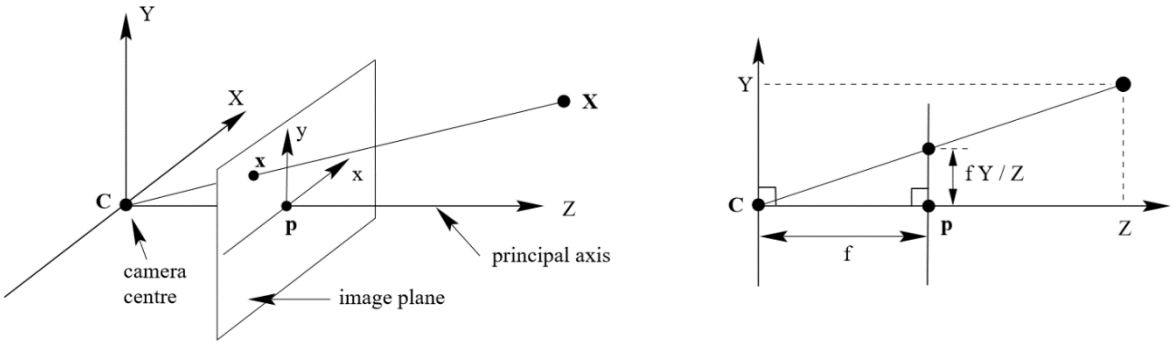

Abbildung 3: Skizze einer Lochkamera [2]

Orthogonalprojektion

Wir betrachten nun folgende Bilder:

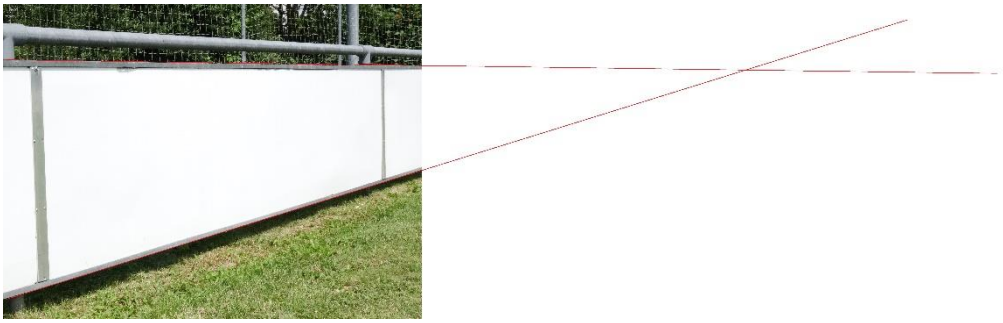

Abbildung 4: Bande mit Fluchtpunkt, 5 Meter Entfernung, 11mm Brennweite

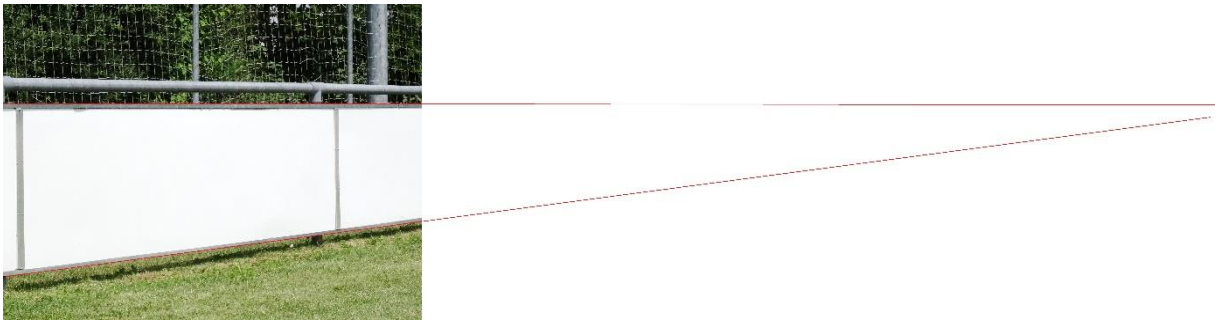

Abbildung 5: Bande mit Fluchtlinien, 10 Meter Entfernung, 26mm Brennweite

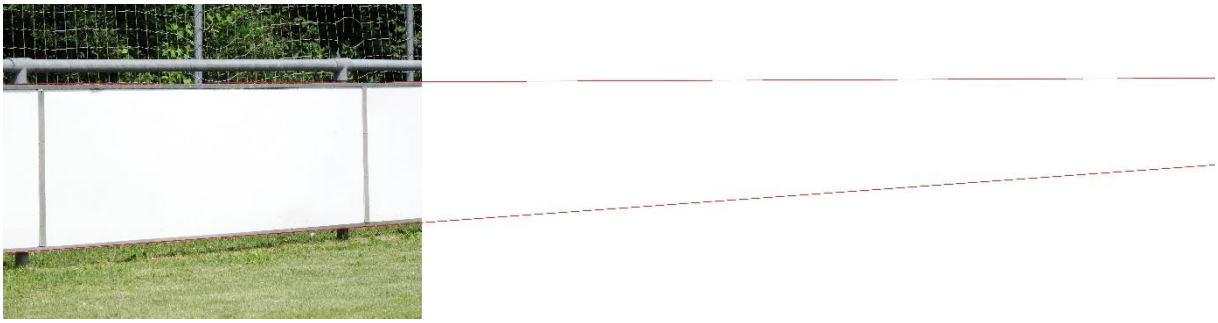

Abbildung 6: Bande mit Fluchtlinien, 20 Meter Entfernung, 55mm Brennweite

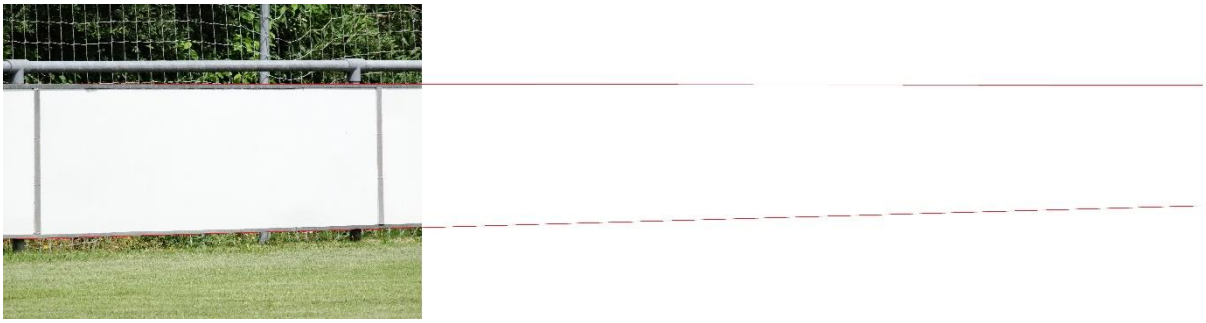

Abbildung 7: Bande mit Fluchtlinien, 50 Meter Entfernung, 124mm Brennweite

Bei den Bildern wird graduell die Brennweite (und die Distanz zum Objekt, damit dieses im gleichen Ausschnitt auf den Aufnahmen bleiben) erhöht. Wie zu erkennen ist, entfernt sich der Fluchtpunkt von der Bildmitte. Die Bilder verschieben sich von einer starken zu einer schwachen Perspektive.

Es kann also theoretisch angenommen werden, dass bei einer unendlich großen Brennweite bzw. Distanz zwischen Kamerazentrum und Bildebene in der realen Welt parallele Linien auch im Bild parallel bleiben. Dies wird als Orthogonalprojektion bezeichnet. [1] [2]

Affine Kamera

Eine Affine Kamera hat ihren Mittelpunkt auf der Ebene in der Unendlichkeit. Die Annahme einer Affine Kamera hat Vorteile, da es Berechnungen vereinfacht und stabilisiert. Natürlich gibt es keine solche Kamera in der Realität. Um ein annehmbares Bild unter solchen Bedingungen zu erhalten reicht es in der Regel, Bildpunkte auf der gleichen Ebene für Berechnungen zu nutzen (also nicht gleichzeitig ein Punkt im Vorder- und Hintergrund des Bildes) und Bilder mit einer höheren Brennweite zu nutzen.

Inwiefern der Faktor der Entfernung/Brennweite das Ergebnis von Algorithmen, die mit Hilfe von Affinen Kameras arbeiten, wird im Folgenden betrachtet.

Affine Rekonstruktion – Faktorisierungsalgorithmus

Ziel ist es, mittels Bilder (i entspricht der Anzahl der Bilder) von Affinen Kameras Kameras $\{M^i, t^i\}$ und 3D-Objekt-Punkte $\{X_j\}$ zu ermitteln, sodass der geometrische Fehler minimal ist:

$$\min_{M^i,t^i,X_j} \sum_{ij} \left\| x_j^i - \hat{x}_j^i \right\|^2 = \min_{M^i,t^i,X_j} \sum_{ij} \left\| x_j^i - (M^i X_j + t^i) \right\|^2$$

Hier wird eine Affine Kamera wie folgt beschrieben:

$$\begin{pmatrix} x \\ y \end{pmatrix} = M \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + t$$

M ist eine 2×3-Matrix und t ein 2-Vektor. $(x, y)^T$ wird hierbei als x weiter behandelt und ist ein inhomogener Bildpunkt, entsprechend ist X der inhomogene Weltpunkt $(X, Y, Z)^T$.

Wie üblich in solchen Minimierungsproblemen, wird der Translationsvektor t^i eliminiert, indem dieser als Zentrum zwischen den Objektpunkten definiert wird. Bei der Affinen Kamera wird hierbei das Zentrum zwischen den Objektpunkten auf das Zentrum zwischen den Bildpunkten abgebildet. Daher ist nun kein Translationsvektor notwendig, weshalb gilt: $t^i = 0$. Damit dies gilt, müssen allerdings alle Objektpunkte für jede Kamera auf vorhandene Bildpunkte abgebildet werden. Somit ist nun folgende Formel gegeben:

$$\min_{M^i,t^i,X_j} \sum_{ij} \left\| x_j^i - \hat{x}_j^i \right\|^2 = \min_{M^i,t^i,X_j} \sum_{ij} \left\| x_j^i - (M^i X_j) \right\|^2$$

Dies kann nun einfach als Matrix dargestellt werden: Die Messmatrix W ist eine $2m \times n$-Matrix aus den zentrierten Bildpunkten:

$$W = \begin{bmatrix} x_1^1 & x_2^1 & \cdots & x_n^1 \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^m & x_2^m & \cdots & x_n^m \end{bmatrix}$$

Da jeder Bildpunkt $x_j^i = M^i X_j$ ist, kann diese auch umgeschrieben werden:

$$W = \begin{bmatrix} M^1 \\ M^2 \\ \vdots \\ M^m \end{bmatrix} [X_1 \quad X_2 \quad \cdots \quad X_n]$$

Da Rauschen vorhanden ist, kann die Gleichung nicht exakt erfüllt sein, also wird eine Matrix $\widehat{W}$ gesucht, die möglichst ähnlich zu W in der Frobenius-Form ist. Diese Matrix kann letztendlich per SVD mit Rang 3 erstellt werden. Hierbei gibt es keine eindeutige Lösung, da:

$$\widehat{W} = U_{2m\times 3} D_{3\times 3} V_{3\times n}^T$$

Somit gibt es je zwei Möglichkeiten $\widehat{M}$ und $\hat{X}$ zu definieren:

$$\widehat{M} = U_{2m\times 3} D_{3\times 3}$$

$$\hat{X} = V_{3\times n}^T$$

oder

$$\widehat{M} = U_{2m\times 3}$$

$$\hat{X} = D_{3\times 3} V_{3\times n}^T$$

Damit ergibt sich folgender Algorithmus: [2]

Objective

Given $n \geq 4$ image point correspondences over m views $\mathbf{x}_j^i$, $j = 1, \ldots, n$; $i = 1, \ldots, m$, determine affine camera matrices $\{\mathtt{M}^i, \mathbf{t}^i\}$ and 3D points $\{\mathbf{X}_j\}$ such that the reprojection error

$$\sum_{ij} \left|\left|\mathbf{x}_j^i - (\mathtt{M}^i \mathbf{X}_j + \mathbf{t}^i)\right|\right|^2$$

is minimized over $\{\mathtt{M}^i, \mathbf{t}^i, \mathbf{X}_j\}$, with $\mathtt{M}^i$ a 2×3 matrix, $\mathbf{X}_j$ a 3-vector, and $\mathbf{x}_j^i = (x_j^i, y_j^i)^\mathsf{T}$ and $\mathbf{t}^i$ are 2-vectors.

Algorithm

(i) **Computation of translations.** Each translation $\mathbf{t}^i$ is computed as the centroid of points in image i, namely

$$\mathbf{t}^i = \langle \mathbf{x}^i \rangle = \frac{1}{n} \sum_j \mathbf{x}_j^i.$$

(ii) **Centre the data**. Centre the points in each image by expressing their coordinates with respect to the centroid:

$$\mathbf{x}_j^i \leftarrow \mathbf{x}_j^i - \langle \mathbf{x}^i \rangle.$$

Henceforth work with these centred coordinates.

(iii) **Construct the** $2m \times n$ **measurement matrix** $\mathtt{W}$ from the centred data, as defined in (18.5), and compute its SVD $\mathtt{W} = \mathtt{U}\mathtt{D}\mathtt{V}^\mathsf{T}$.

(iv) Then the matrices $\mathtt{M}^i$ are obtained from the first three columns of $\mathtt{U}$ multiplied by the singular values:

$$\begin{bmatrix} \mathtt{M}^1 \\ \mathtt{M}^2 \\ \vdots \\ \mathtt{M}^m \end{bmatrix} = \begin{bmatrix} \sigma_1 \mathbf{u}_1 & \sigma_2 \mathbf{u}_2 & \sigma_3 \mathbf{u}_3 \end{bmatrix}.$$

The vectors $\mathbf{t}^i$ are as computed in step (i) and the 3D structure is read from the first three columns of $\mathtt{V}$

$$\begin{bmatrix} \mathbf{X}_1 & \mathbf{X}_2 & \ldots & \mathbf{X}_n \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 \end{bmatrix}^\mathsf{T}.$$

Algorithm 18.1. *The factorization algorithm to determine the MLE for an affine reconstruction from n image correspondences over m views (under Gaussian image noise).*

Abbildung 8: Faktorisierungs-Algorithmus von Hartley & Zisserman [2]

Implementierung

Zur Implementierung des folgenden Codes wurde Python genutzt. Zunächst werden die Bilder eingelesen, also eindeutige Bildpunkte bestimmt. Für diese wird dann mit Hilfe der OpenCV-Bibliothek eine klassische Kamera-Matrix bestimmt. Dann werden Kamera-Matrizen mit dem Faktorisierungs-Algorithmus von Hartley & Zisserman bestimmt. Die Ergebnisse werden dann zunächst via geometrische Fehler verglichen. Zusätzlich werden dann die 3D-Punkte verschoben und mit den Matrizen abgebildet, um diese dann mit weiteren bestimmten Bildpunkten zu vergleichen.

Vorbereitung

```
# prepare object points array for size
# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objPoints = np.zeros((PATTERN_SIZE[0]*PATTERN_SIZE[1],3), np.float32)
objPoints[:,:2] =
np.mgrid[0:PATTERN_SIZE[0],0:PATTERN_SIZE[1]].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objectPoints = [] # 3d point in real world space
imagePoints = [] # 2d points in image plane.

images = glob.glob(JPG_NAME)

for fileName in images:
    print("Processing ", fileName)
    image = cv2.imread(fileName)
    grayImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    foundCorners, corners = cv2.findChessboardCorners(grayImage,
PATTERN_SIZE, None)

    if foundCorners == True:
        print("Found Corners!")
        cornersRefined = cv2.cornerSubPix(grayImage, corners, (11, 11), (-
1, -1), CRITERIA)
        image = cv2.drawChessboardCorners(image, PATTERN_SIZE,
cornersRefined, foundCorners)
        cv2.imwrite("edited_" + fileName, image)

        objectPoints.append(objPoints)
        imagePoints.append(cornersRefined)
        print("Image Points processed with OpenCV\n\n")


    else:
        raise Exception("Didn't found corners in ", fileName)
```

Es werden zunächst zwei Arrays indiziert, die später für jedes Bild die Bild- und Objekt-Punkte enthalten. Dann werden alle Bilder im Ordner, in dem auch das ausführende Programm liegt durchlaufen:

Die Bilder werden gelesen und es werden mittels vorher festgelegtem Pattern versucht eindeutige Eckpunkte zu finden (siehe Abbildung 9: 9x6-ChessPattern von OpenCV). Sollte das Muster gefunden werden, werden diese nun verfeinert und zur Bildpunkt-Liste hinzugefügt. Zusätzlich werden zur Objektpunkt-Liste die vorher definierten Weltpunkte hinzugefügt. Durch das bekannte Muster können diese vereinfacht erstellt werden: Die Z-Ebene wird einfach auf 0 gesetzt, die X- und Y- Werte werden nach dem Schachbrettmuster erstellt, also in diesem Fall X von 0 bis 8 und Y von 0 bis 5.

Zusätzlich wird das gefundene Muster noch in den Bildern markiert und mit dem Zusatz „edited_“ gesichert, sodass manuell nachgeprüft werden kann, ob die Muster zufriedenstellend entdeckt wurden.

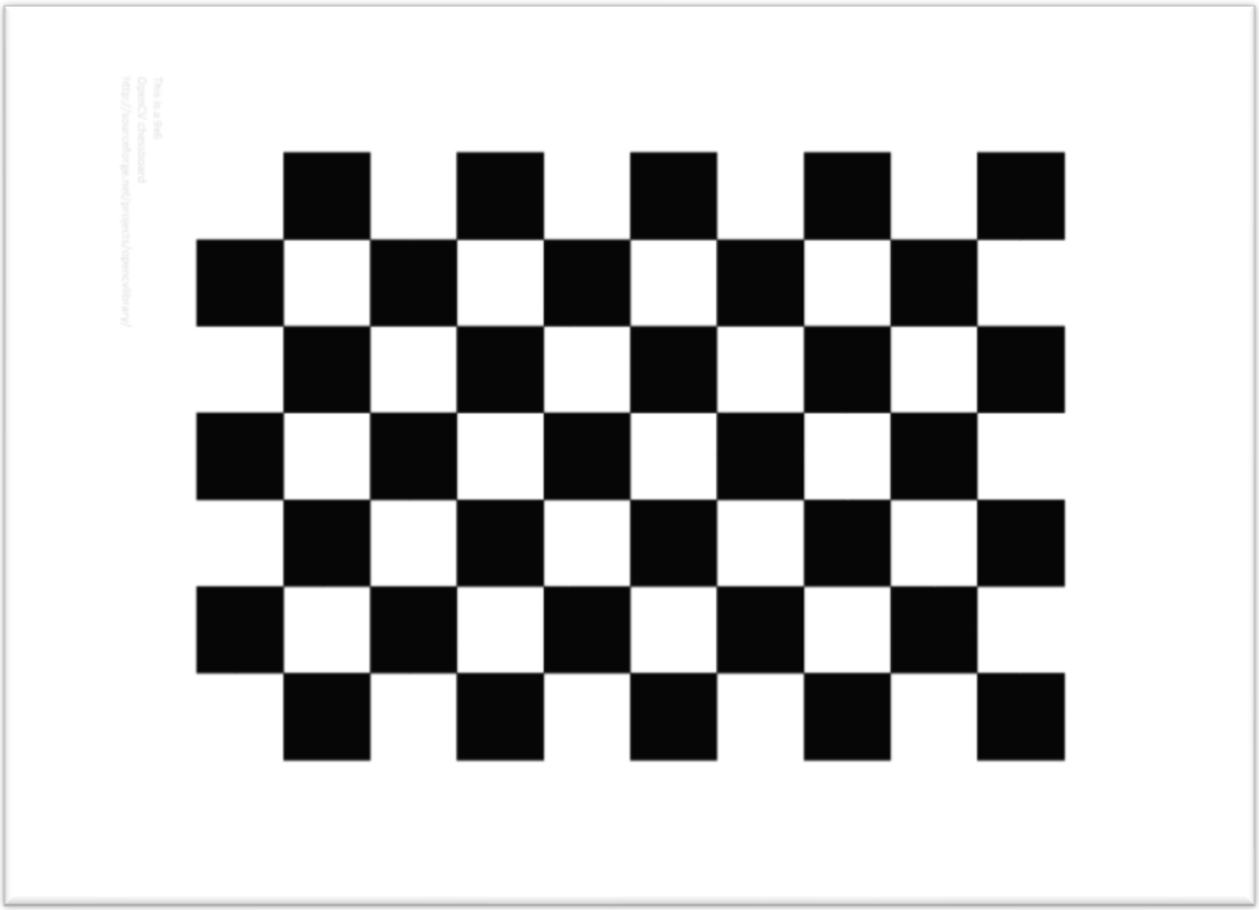

Abbildung 9: 9x6-ChessPattern von OpenCV [3]

Matrix-Berechnung mit OpenCV

```
startOpenCV = time.time()
reprojectionError, cameraMatrix, distCoeffs, rotationVecs,
translationVecs = cv2.calibrateCamera(objectPoints, imagePoints,
grayImage.shape[::-1],None,None)
endOpenCV = time.time()
timeOpenCV = endOpenCV - startOpenCV

print("\n\n\nCamera Matrix:")
print(cameraMatrix)
print("\n\nOpenCV Reprojection Error:")
print(reprojectionError)


meanReprojectionErrors = []
allReprojectedPoints = []
for i in range(len(objectPoints)):
    reprojectedPoints, _ = cv2.projectPoints(objectPoints[i],
rotationVecs[i], translationVecs[i], cameraMatrix, distCoeffs)
    reprojectedPoints = reprojectedPoints.reshape(-1,2)
    allReprojectedPoints.append(reprojectedPoints)

meanReprojectionErrors.append(computeMeanReprojectionError(imagePoints[i]
, reprojectedPoints))
print("\n\n\nReprojected Points:\n", allReprojectedPoints);
print("\nSelfcalculated Reprojection Errors:\n", meanReprojectionErrors)
```

Hier wird nun einfach die OpenCV-Funktion calibrateCamera() aufgerufen, die dann die Kameramatrix mit den verschiedenen Vektoren und dem gesamten geometrischen Fehler zurück gibt. Auch hier wird dann mittels Minimierung des Reprojezierungs-Fehlers eine Kameramatrix ermittelt. Zusätzlich wird für die Auswertung die Zeit der Verarbeitung gemessen.

Die vorher definierten Objektpunkte werden nun mit der ermittelten Matrix wieder auf das Bild abgebildet und mit den tatsächlichen Bildpunkten die durchschnittlichen geometrischen Fehler pro Bild errechnet.

Matrizen-Berechnung mit Faktorisierungs-Algorithmus

```
print("\n\n\nAffine                                                    Reconstruction")
startAffine                                =                                time.time()
M1,  X1,  t,  M2,  X2  =  affineReconstruction(imagePoints,  PATTERN_SIZE)
endAffine                                  =                                time.time()
timeAffine          =          endAffine          -          startAffine
affineCameraMatrizesV1         =         np.split(M1,         len(images))
affineCameraMatrizesV2         =         np.split(M2,         len(images))


print("\n\nAffine                    CameraMatrizes                    V1:")
print(M1)
print("\n\nAffine                    CameraMatrizes                    V1:")
print(affineCameraMatrizesV1)
print("\n3D-Points                                                   V1:")
print(X1)

print("\n\nAffine                    CameraMatrizes                    V2:")
print(M2)
print("\n\nAffine                    CameraMatrizes                    V2:")
print(affineCameraMatrizesV2)
print("\n3D-Points                                                   V2:")
print(X2)
print("\nt:\n",                                                        t)

affineReprojectedPoints1,            affineReprojectionError1            =
affineReprojection(affineCameraMatrizesV1,     imagePoints,     t,     X1)
affineReprojectedPoints2,            affineReprojectionError2            =
affineReprojection(affineCameraMatrizesV2, imagePoints, t, X2)
```

Hier werden ebenso die zunächst die Matrizen berechnet, dabei die benötigte Zeit gemessen und die berechneten Objektpunkte wieder auf Bildpunkte abgebildet. Hiermit werden dann wieder die Reprojektions-Fehler pro Bild berechnet. Zu beachten ist, dass wegen der Uneindeutigkeit am Ende des Algorithmus jeweils alles doppelt berechnet wird.

Wir beachten jetzt die Implementierung der Matrizen-Berechnung:

```
def affineReconstruction(corners, patternSize):
    # i = 1, ..., m --> Count of camera matrices and therefore images
    # j = 1, ..., n --> Count of realworld points and therefore
imagepoints per image
    centroid = []
    for i in range(len(corners)):
        # computation of translation:
        # t^i = <x^i> = 1 / n * sum(x^i(j))
        centroid.append(sum(corners[i]) /
(patternSize[0]*patternSize[1]))
    #print("Centroid:")
    centroid = np.reshape(centroid, (len(corners), 2))
    #print(centroid)

    # centre the data:
    # x^i(j) <-- x^i(j) - t^i
    centredPoints = []
    for i in range(len(corners)):
        centredPoints.append(corners[i] - centroid[i])
    #print("\n\nCentred Points")
    #print(centredPoints)

    # Construct measurement matrix W
    #       _                                  _
    #      | x1,1    x1,2    ...     x1,n |
    #      | x2,1    x2,2    ...     x2,n |
    # W = | ...     ...     ...     ...  |
    #      | xm,1    xm,2    ...     xm,n |
    #      --                                --
    #
    # n = Count of image/world points
    # m = Count of images / camera matrices
    # 2D-Points (x) stacked vertically
    # --> W = 2m*n matrix

    measurementMatrix = []
    for i in range(len(corners)):
        measurementMatrix.append(np.vstack((centredPoints[i][:, 0, 0],
centredPoints[i][:, 0, 1])))
    measurementMatrix = np.reshape(measurementMatrix, (len(corners)*2,
len(centredPoints[0])))
    #print("\n\n\nMeasurement Matrix:\n", measurementMatrix)

    #compute svd
    u, dTemp, vT = svds(measurementMatrix, k=3)
    d = np.diag(dTemp)

    M1 = np.matmul(u, d)
    M2 = u

    X1 = vT
    X2 = np.matmul(d, vT)
    #seems like (y, x, z)

    return M1, X1, centroid, M2, X2
```

Zunächst werden pro Bild die durchschnittlichen Bildpunkt-Zentren errechnet und die Punkte danach ausgerichtet. Danach wird die Messmatrix erstellt und mit dieser dann die Singulärwert-Zerlegung durchgeführt. Die beiden Varianten der Bildmatrizen und Objektpunkte werden errechnet und zurückgegeben.

Die Projektion der errechnete Objektpunkte zurück zu Bildpunkte ist entsprechend der Definition einfach:

```
def affineReprojection(affineCameraMatrizes, imagePoints, t, X):
    affineReprojectedPoints = []
    affineReprojectionError = []
    for i in range(len(affineCameraMatrizes)):

affineReprojectedPoints.append(calculate3dTo2d(affineCameraMatrizes[i],
t[i], X))

affineReprojectionError.append(computeMeanReprojectionError(imagePoints[i
], affineReprojectedPoints[i][:]))
    affineReprojectedPoints = np.reshape(affineReprojectedPoints,
(len(affineCameraMatrizes), len(imagePoints[0]), 2))
    return affineReprojectedPoints, affineReprojectionError


def calculate3dTo2d(M, t, X):
    # (x, y) = M * (x, y, z) + t
    xyTemp = np.matmul(M, X)
    xy = []
    for i in range(len(xyTemp[0])):
        xy.append(xyTemp[:, i] + t)

    return xy
```

Für alle Matrizen/Bilder werden die Punkte abgebildet. Hierbei findet eine einfache Matrizenmultiplikation der Punkte mit der Kameramatrix und zusätzlich die Verschiebung Richtung durchschnittlichem Zentrum der vorgegebenen Bildpunkte durch Addition mit dem Translationsvektor/Zentrum statt. Danach wird auch hier der Reprojektions-Fehler errechnet.

Abgleich mit zusätzlichem Muster

Um nicht nur die direkte Qualität der Projektionen zu vergleichen, soll zusätzlich eine Verschiebung der Muster und damit der Objekt- und Bild-Punkte durchgeführt werden:

```
def movingAffine(X, affineCameraMatrizes, images, t, addition):
    # 3d-Points like y, x, z.
    relativeDistance = [VERTICAL_PATTERN_DISTANCE / POINT_DISTANCE,
HORIZONTAL_PATTERN_DISTANCE / POINT_DISTANCE]  # Ratio between Distance
of Patterns and Distance of Points
    newX = X.copy()
    relative3dDistance = [0, 0, 0]
    relativeHorizontal3dDistance = [0, 0, 0]
    relativeVertical3dDistance = [0, 0, 0]

        image = cv2.drawChessboardCorners(image, PATTERN_SIZE,
newImagePoints.astype(np.float32), foundCorners)
        cv2.imwrite("edited" + addition + "_" + fileName, image)

    print("\n\nControl image affine reprojection error" + addition +
":\n", reprojectionError)
    print("Mean: ", np.mean(reprojectionError))
```

```
    #Horizontal Distance
    for j in range(len(relative3dDistance)):
        for i in range(PATTERN_SIZE[1]):  # For each row
            # Sum up Distance between Points
            relativeHorizontal3dDistance[j] += X[j][i * PATTERN_SIZE[0]]
- X[j][(i + 1) * PATTERN_SIZE[0] - 1]
        # Get mean Distance by dividing by number of added Distances
        relativeHorizontal3dDistance[j] = relativeHorizontal3dDistance[j]
/ (PATTERN_SIZE[1] * (PATTERN_SIZE[0] - 1))
        # Multiplying with relative Distance
        relativeHorizontal3dDistance[j] = relativeHorizontal3dDistance[j]
* relativeDistance[1]

    #Vertical Distance
    for j in range(len(relative3dDistance)):
        for i in range(PATTERN_SIZE[0]): #For each column
            #Sum of vertical distances
            relativeVertical3dDistance[j] += X[j][i] - X[j][i +
PATTERN_SIZE[0] * (PATTERN_SIZE[1] - 1)]
        # Get mean Distance by dividing by number of added Distances
        relativeVertical3dDistance[j] = relativeVertical3dDistance[j] /
((PATTERN_SIZE[0] - 1) * PATTERN_SIZE[1])
        # Multiplying with relative Distance
        relativeVertical3dDistance[j] = relativeVertical3dDistance[j] *
relativeDistance[0]

    relative3dDistance = np.add(relativeVertical3dDistance,
relativeHorizontal3dDistance)
    newX = np.add(newX, np.reshape(relative3dDistance, (3, 1)))

    reprojectionError = []
    for imageNumber in range(len(images)):
        fileName = images[imageNumber].replace(CALC_FILE_BEGINNING,
CONTROL_FILE_BEGINNING)
        newImagePoints =
calculate3dTo2d(affineCameraMatrizes[imageNumber], t[imageNumber], newX)
        newImagePoints = np.reshape(newImagePoints, (PATTERN_SIZE[0] *
PATTERN_SIZE[1], 1, 2))
        image = cv2.imread(fileName)

        grayImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        foundCorners, corners = cv2.findChessboardCorners(grayImage,
PATTERN_SIZE, None)
        if foundCorners == False:
            raise Exception("Didn't found corners in ", fileName)
        cornersRefined = cv2.cornerSubPix(grayImage, corners, (11, 11),
(-1, -1), CRITERIA)

reprojectionError.append(computeMeanReprojectionError(cornersRefined,
np.reshape(newImagePoints, (PATTERN_SIZE[0]*PATTERN_SIZE[1], 2))))


        image = cv2.drawChessboardCorners(image, PATTERN_SIZE,
newImagePoints.astype(np.float32), foundCorners)
        cv2.imwrite("edited" + addition + "_" + fileName, image)

    print("\n\nControl image affine reprojection error" + addition +
":\n", reprojectionError)
    print("Mean: ", np.mean(reprojectionError))
```

Es wird zunächst die relative Distanz zwischen den Objekten festgestellt. Dafür muss die reale Distanz zwischen den Objektpunkten und die reale Distanz zwischen zwei Objektpunkten der Originalpunkte bekannt sein. Diese werden dann im Verhältnis gesetzt mit den Abständen der berechneten Objektpunkte, um somit die umgerechneten neuen Objektpunkten zu ermitteln.

Diese werden dann für jedes Bild auf Bildpunkte projiziert und dann mit den realen Bildpunkten verglichen, indem auch hier der Projektionsfehler berechnet wird. Dafür werden zwei Bilder aus der gleichen Position mit unterschiedlichen Bildpunkten benötigt, wie bspw. in diesem Fall:

Abbildung 10: Bild zur Berechnung der Matrizen

Abbildung 11: Bild zur Kontrolle mit verschobenen Punkten

Die Berechnung für die Matrizen von OpenCV funktionieren analog.

Anpassungen

```
PATTERN_SIZE = (9, 6)

CONTROLLING = True #Is a Controlling-Pattern there
HORIZONTAL_PATTERN_DISTANCE = 1 #Horizontal distance between Pattern
sheets in meter (<0: control pattern left, >0: right)
VERTICAL_PATTERN_DISTANCE = 0.00 #Vertical distance between Pattern
sheets in meter (<0: control pattern up, >0: down)

POINT_DISTANCE = 0.02 #Distance between points in meter

CRITERIA = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30,
0.001)

CALC_FILE_BEGINNING = 'calc'
CONTROL_FILE_BEGINNING = 'control'
JPG_NAME = CALC_FILE_BEGINNING + '*.jpg'
```

Am Anfang des Codes ist es möglich, das Programm anzupassen, welche Größe das Muster hat (`PATTERN_SIZE`), ob Kontrollbilder vorliegen (`CONTROLLING`), welchen Abstand die Objektpunkte besitzen (`POINT_DISTANCE`) und welchen Abstand die verschobenen Objektpunkte zu den Ursprungspunkten haben (`HORIZONTAL_PATTERN_DISTANCE`, `VERTICAL_PATTERN_DISTANCE`). Ebenso kann die Bild-Benennung angepasst werden (`CALC_FILE_BEGINNING`, `CONTROL_FILE_BEGINNING`).

Testergebnisse

Zeitunterschiede

Bei der Berechnung der Matrizen gibt es enorme Unterschiede in der Laufzeit (in Sekunden):

Laufzeit OpenCV: 0.07711195945739746

Laufzeit Faktorisierung: 0.00400090217590332

Reprojektionsfehler

Hier werden nun die Reprojektionsfehler (geometrische Fehler in Pixel) begutachtet, aufgelistet nach Ort der Bilder und mit Durchschnittswert:

Art		**Entfernung (m)**	**Linksaußen**	**Links**	**Mitte**	**Rechts**	**Rechtsaußen**	**Mean**
OpenCV		6	0,69	0,44	0,51	0,41	0,53	0,52
Affine		6	0,9	0,52	0,4	0,67	0,8	0,66
OpenCV		12	1,51	3,22	2,93	3,75	ungenau	2,85
Affine		12	1,75	1,71	1,85	1,43	ungenau	1,69
OpenCV		20	0,49	0,55	0,53	0,51	0,58	0,53
Affine		20	0,63	0,45	0,49	0,49	0,58	0,53
OpenCV		30	0,55	0,58	0,48	0,6	0,64	0,57
Affine		30	0,56	0,49	0,4	0,54	0,69	0,54
OpenCV		40	1,22	0,9	1	1,17	1,14	1,09
Affine		40	1,08	0,81	0,88	0,98	1,03	0,95

Zunächst auffällig ist, dass die Unterschiede stark von den einzelnen Bildern abhängig sind und selbst bei der gleichen Entfernung stark variieren. Zudem fällt auf, dass die Qualität sich auch nur bedingt mit der Entfernung anpasst, sondern z.T. gleich und andererseits arg variiert. Was auffällt ist, dass die affine Variante mit zunehmender Entfernung leicht bessere Ergebnisse erzielt.

Reprojektionsfehler mit Verschiebung

Nun beachten wir die Ergebnisse bei verschobenen Objektpunkten:

Art		Entfernung (m)	Linksaußen	Links	Mitte	Rechts	Rechtsaußen	Mean
OpenCV		6	16,21	21,49	44,5	102,71	108,56	58,69
Affine		6	177,55	121,61	47,77	94,07	119,73	112,15
OpenCV		12	165,54	130,29	325,63	321,4	ungenau	235,71
Affine		12	108,77	86,39	50,58	29,74	ungenau	68,87
OpenCV		20	101,73	164,39	112,48	190,6	200,79	154
Affine		20	85,26	62,56	40,04	11,02	22,81	44,34
OpenCV		30	60,7	67,7	166,26	70,42	141,99	101,42
Affine		30	54,1	45,16	28,58	20,51	7,22	31,12
OpenCV		40	102	125	54	59	57	79
Affine		40	70	57	43	39	24	46

Hier fällt auf, dass die Werte mit der affinen Methode bedeutend besser abschneiden, umso größer die Entfernung ist. Allerdings ist dafür der durchschnittliche Fehler bei naher Distanz bedeutend höher. Hier sieht man deutlich, wie stark das nicht-einhalten der affinen Grundbedingung sich auswirkt.

Fazit

Sofern Bilder die Voraussetzungen der Affinen Kamera halbwegs erfüllen, bietet der Faktorisierungsalgorithmus eine sehr gute Alternative, um Kameramatrizen zu berechnen. Zwar ist die Bestimmung/Festlegung der Welt-Punkte nicht so elegant, allerdings bietet dieser Algorithmus zeitliche und qualitative Vorteile. Man sollte dennoch bedenken, dass dies eine hoch spezielle Lösung ist und mit einem Ansatz für deutlich diversere Bedingungen verglichen wird. Aber gerade bei statischen Situationen, die diese Voraussetzungen erfüllen, ist der Faktorisierungsalgorithmus sehr interessant.

Literaturverzeichnis

[1] M. Obeysekera, *Affine Reconstruction from Multiple Views using Singular Value Decomposition,* The University of Western Australia, 2003.

[2] R. Hartley und A. Zisserman, Multiple View Geometry in Computer Vision, Cambridge: Cambridge University Press, 2003.

[3] Open Source Computer Vision, „Camera Calibration," Open Source Computer Vision, [Online]. Available: https://docs.opencv.org/3.4/dc/dbb/tutorial_py_calibration.html. [Zugriff am 12 Juli 2019].

Abbildungsverzeichnis

Abbildung 1: Bande, 5 Meter entfernt der rechten senkrechten Spange Ein Bild entlang einer Bande. Der linke Teil war näher der Kamera und wird somit größer dargestellt als der rechte Teil. Dadurch lässt sich die Tiefe für den Betrachter erahnen. 3
Abbildung 2: Bande mit eingezeichnetem Fluchtpunkt, 5 Meter Entfernung 4
Abbildung 3: Skizze einer Lochkamera [2]..... 4
Abbildung 4: Bande mit Fluchtpunkt, 5 Meter Entfernung, 11mm Brennweite..... 5
Abbildung 5: Bande mit Fluchtlinien, 10 Meter Entfernung, 26mm Brennweite 5
Abbildung 6: Bande mit Fluchtlinien, 20 Meter Entfernung, 55mm Brennweite 5
Abbildung 7: Bande mit Fluchtlinien, 50 Meter Entfernung, 124mm Brennweite 5
Abbildung 8: Faktorisierungs-Algorithmus von Hartley & Zisserman [2]..... 8
Abbildung 9: 9x6-ChessPattern von OpenCV [3]..... 10
Abbildung 10: Bild zur Berechnung der Matrizen Abbildung 11: Bild zur Kontrolle mit verschobenen Punkten..... 16