



Hochschule für Angewandte
Wissenschaften Hamburg
Hamburg University of Applied Sciences

Verbundprojekt Master Automatisierung

Gewerk 5 – Bildverarbeitung

Teilnehmer:

Markus Baden, Fabian Büsis, Steffen Todzy

Zeitraum:

21. November 2013 bis 24. April 2014

15. Mai 2014

Vorwort

Die im Rahmen des Verbundprojektes vom Gewerk 5 erstellten Programme (ausführbare Dateien und Quellcodes) sowie die Dokumentation können unter folgendem Link heruntergeladen werden:

https://www.dropbox.com/s/b76wg06u3ual3zu/VPJ_Gewerk_5.zip

Die Kapitel 1, 2.1, 2.2 und 4 wurden von Fabian Büssis bearbeitet.

Die Kapitel 2.1, 2.2 und 2.3 wurden von Steffen Todzy bearbeitet.

Die Kapitel 3 und 5 wurden von Markus Baden bearbeitet.

Während der gesamten Projektphase hat innerhalb unserer Gruppe ein reger Wissens- und Ideenaustausch in alle Richtungen stattgefunden. Aus unserer Sicht ist es daher nicht möglich Leistungen von Einzelpersonen innerhalb der Gruppe besonders hervorzuheben oder abzuschwächen.

Inhaltsverzeichnis

Abbildungsverzeichnis	5
Tabellenverzeichnis	7
1. Aufgabenstellung und Ziele	8
2. Kalibrierungsprogramm (CalibrationTool)	9
2.1. Benutzeroberfläche und Bedienung des Calibration Tool	9
2.1.1. Programmaufbau zur Berechnung der Transformationsmatrix	15
2.1.2. Perspektivische Verzerrung Beschreibung	19
2.1.3. Benutzeroberfläche Correcting Lens Distortion	21
2.1.4. Linsenverzeichnung Beschreibung	23
2.1.5. Programmaufbau distortioncoefficients	26
2.2. XML	29
2.2.1. Aufbau der XML-Datei	29
2.2.2. Einlesen der XML-Datei	30
2.2.3. Erstellung der XML-Datei	30
2.2.4. Schnittstellen	31
2.3. Ergebnis	32
3. Hauptprogramm (RobotDetection)	33
3.1. Benutzeroberfläche und Bedienung	33
3.1.1. Aufbau der GUI	33
3.1.2. Bedienelemente	33
3.1.3. Benutzerführung und Fehlermeldungen	35
3.1.4. Einstellungen in INI-Datei	36
3.2. Funktionsumfang	37
3.2.1. Programmaufbau	37
3.2.2. Programmablauf	37
3.2.3. Initialisierung	37
3.2.4. Bildverarbeitung und Kreiserkennung	39
3.2.5. Roboteridentifikation	42
3.2.6. Datenversand per UDP	45
3.3. Ergebnisse	46

3.3.1. Verbesserungen gegenüber der Matlab-Lösung	46
3.3.2. Performance	46
4. Messung der Positionsgenauigkeit	47
5. Fazit und Ausblick	49
A. Anhang	51
A.1. Linsenverzeichnung	51
A.1.1. Verzeichnung der Kamera 0	51
A.1.2. Verzeichnung der Kamera 1	53
A.1.3. Verzeichnung der Kamera 2	55
A.1.4. Verzeichnung der Kamera 3	57
A.1.5. Verzeichnung der Kamera 4	59
A.1.6. Verzeichnung der Kamera 5	61
A.2. Übersichtsplan der Bodenmarkierungen	63

Abbildungsverzeichnis

2.1.	Aufforderung eine Kalibrationsdatei auszuwählen	9
2.2.	Auswahlfenster der Kalibrationsdatei	10
2.3.	Hauptfenster des „Calibration Tool“	11
2.4.	Raumkalibrierung mit Hilfe des Stativs	13
2.5.	Ermittlung der Transformationsmatrix	13
2.6.	Funktionen des „Calibration Tool“	14
2.7.	Fenster der „Camera Settings“Funktion	14
2.8.	perspektivisch verzerrtes Kamerabild	20
2.9.	perspektivisch entzerrtes Kamerabild	21
2.10.	distortioncoefficients	22
2.11.	Vollständiges Verzeichnungsmodell der Kamera 1	25
2.12.	Verzerrtes Kamerabild	26
2.13.	Entzerrtes Kamerabild	26
2.14.	Aufbau der XML-Datei	30
2.15.	Schnittstellen	31
3.1.	Bereiche der Benutzeroberfläche des RobotDetection-Programms	34
3.2.	Benutzeroberfläche mit Livebild (aufgehelle Kameras)	34
3.3.	Bedienelemente	35
3.4.	Ablaufschema des Hauptprogrammes	38
3.5.	Anordnung der Reflektormarken auf den Robotern	42
4.1.	Übersicht der gemessenen Posen	47
A.1.	Vollständiges Verzeichnungsmodell der Kamera 0	51
A.2.	Radiales und tangentiales Verzeichnungsmodell der Kamera 0	52
A.3.	Vollständiges Verzeichnungsmodell der Kamera 1	53
A.4.	Radiales und tangentiales Verzeichnungsmodell der Kamera 1	54
A.5.	Vollständiges Verzeichnungsmodell der Kamera 2	55
A.6.	Radiales und tangentiales Verzeichnungsmodell der Kamera 2	56
A.7.	Vollständiges Verzeichnungsmodell der Kamera 3	57
A.8.	Radiales und tangentiales Verzeichnungsmodell der Kamera 3	58
A.9.	Vollständiges Verzeichnungsmodell der Kamera 4	59
A.10.	Radiales und tangentiales Verzeichnungsmodell der Kamera 4	60

A.11. Vollständiges Verzeichnungsmodell der Kamera 5	61
A.12. Radiales und tangentiales Verzeichnungsmodell der Kamera 5	62
A.13. Übersichtsplan der Bodenmarkierungen (Maße in mm)	63

Tabellenverzeichnis

3.1. Performance-Vergleich	46
4.1. Positionsgenauigkeit	48

1. Aufgabenstellung und Ziele

Im Verbundprojekt des Master Automatisierung wird ein automatischer Fertigungsablauf unter Einsatz von mobilen Robotern im Labor realisiert. Von entscheidender Bedeutung für die Fertigungsplanung, die Bahnplanung sowie die hochgenaue Bahnführung ist eine möglichst exakte und schnelle Lagebestimmung der mobilen Roboter.

Hierzu kommt ein Deckenkamerasystem bestehend aus sechs Kameras zum Einsatz, mit dem die durch Reflektorpunkte markierten Roboter identifiziert werden sollen. Die Position und der eingeschlagene Winkel der Roboter sind zu bestimmen und schließlich den Teilnehmern über W-LAN zur Verfügung zu stellen.

Bislang wurde im Labor ein Bildauswertungssystem auf Basis von Matlab eingesetzt, das hinsichtlich Funktionalität, Benutzeroberfläche, Bedienkomfort und Performance nicht mehr zeitgemäß ist. Die Matlab-Lösung bietet außerdem nur eine sehr eingeschränkte und umständliche Möglichkeit zur Kalibrierung.

Die von Gewerk 5 zu erfüllende Aufgabenstellung umfasst zunächst die Ermittlung von Kalibrierungsdaten des Kamerasystems. Dazu gehören u.a. die Linsenverzeichnung der Kameras und die perspektivische Transformation der Einzelbilder auf das Fahrfeld der Roboter. Diese Kalibrierungsdaten sollen dann verwendet werden um die Positionen der Roboter aus den Kamerabildern möglichst genau und möglichst schnell zu bestimmen.

Um diese Aufgaben zu realisieren werden zwei Programme entwickelt:

- Das *CalibrationTool* zur Kalibrierung des Kamerasystems und zur Ermittlung der perspektivischen Transformation (Kap. 2) sowie
- das Hauptprogramm *RobotDetection*, das dazu dient die Kalibrierungsdaten auf die Kamerabilder anzuwenden und die Roboterpositionen zu bestimmen (Kap. 3).

Als Schnittstelle zur Übergabe der Kalibrierungsdaten zwischen den beiden Programmen kommt eine XML-Datei zum Einsatz.

Zur Entwicklung der beiden Applikationen wird das moderne *Qt*-Framework verwendet. Zur Bildverarbeitung werden Algorithmen aus der *Open-Computer-Vision (OpenCV)* Bibliothek benutzt. Bei beiden Werkzeugen handelt es sich um in der Industrie weit verbreitete Open-Source Software.

2. Kalibrierungsprogramm (Calibration Tool)

2.1. Benutzeroberfläche und Bedienung des Calibration Tool

Nach dem Ausführen der „CalibrationTool.exe“ Datei startet das Kalibrierungsprogramm. Bei Start wird automatisch die „fieldcoordinates.ini“ Datei eingelesen. Mit dieser Datei werden die im Raum vermessenen Koordinaten initialisiert. Ist keine Datei vorhanden werden für die vermessenen Positionen automatisch die Standardwerte übernommen.

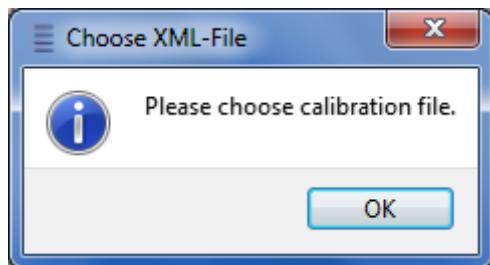


Abbildung 2.1.: Aufforderung eine Kalibrationsdatei auszuwählen

Der Benutzer wird, wie in der Abbildung 2.1, dazu aufgefordert eine Kalibrationsdatei auszuwählen. Daraufhin öffnet sich das Auswahlfenster, das in der Abbildung 2.2 dargestellt ist. Der Benutzer kann jetzt eine Datei im „XML“-Format auswählen.

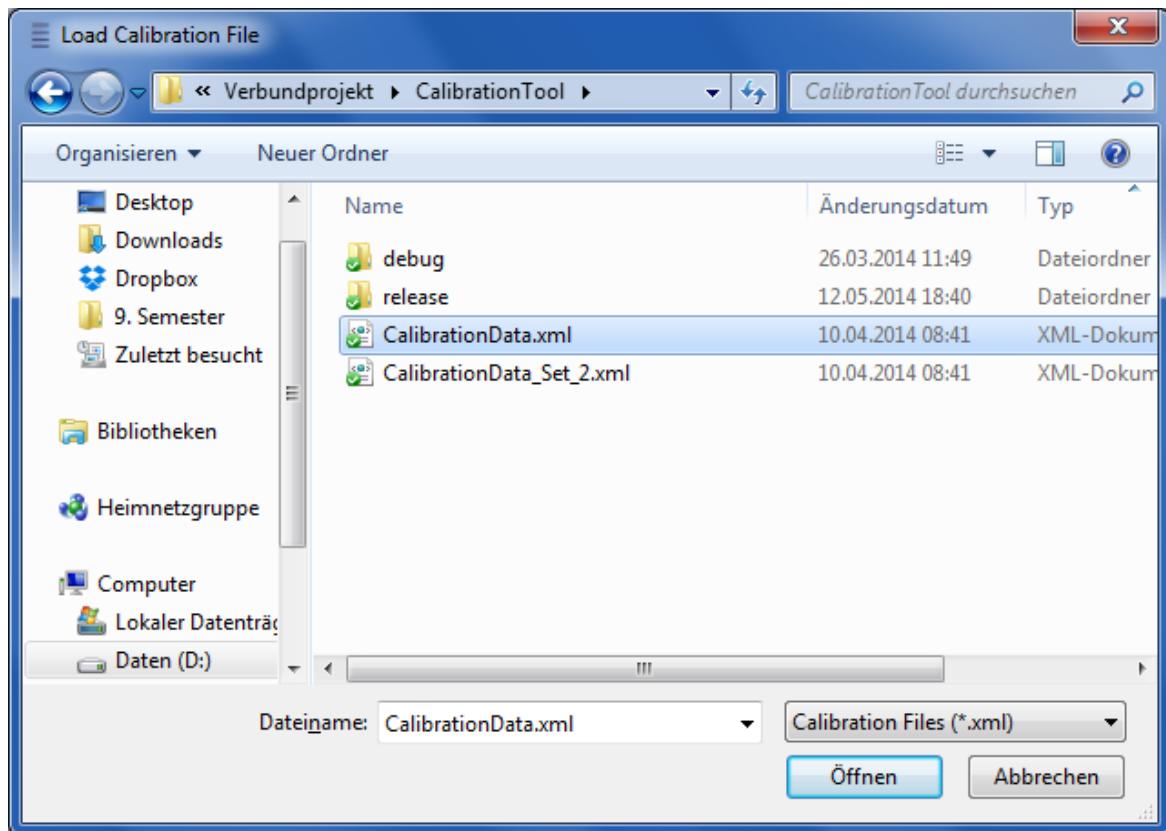


Abbildung 2.2.: Auswahlfenster der Kalibrationsdatei

Aus den Datei werden anschließend die vorkonfigurierten Einstellungen der Kameras ausgelesen. Dazu gehören die Helligkeits- und Kontrasteinstellungen der Kamera, die Transformationsmatrix und die Linsenverzeichnung der jeweiligen Kamera. Durch das Auswahl der Kalibrationsdatei hat der Nutzer die Möglichkeit verschiedene Kameradatensätze zu laden. Damit kann der Nutzer mehrere Parametersätze für die Kameras anlegen.

Wenn keine „XML“-Datei ausgewählt wird, wird ein leerer Parametersatz angelegt und der Benutzer hat die Möglichkeit eine Kalibrationsdatei mit dem „Calibration Tool“ zu erzeugen.

Nachdem die Kalibrationsdaten eingelesen wurden, öffnet sich das Hauptfenster des „Calibration Tools“ wie in Abbildung 2.3 dargestellt.

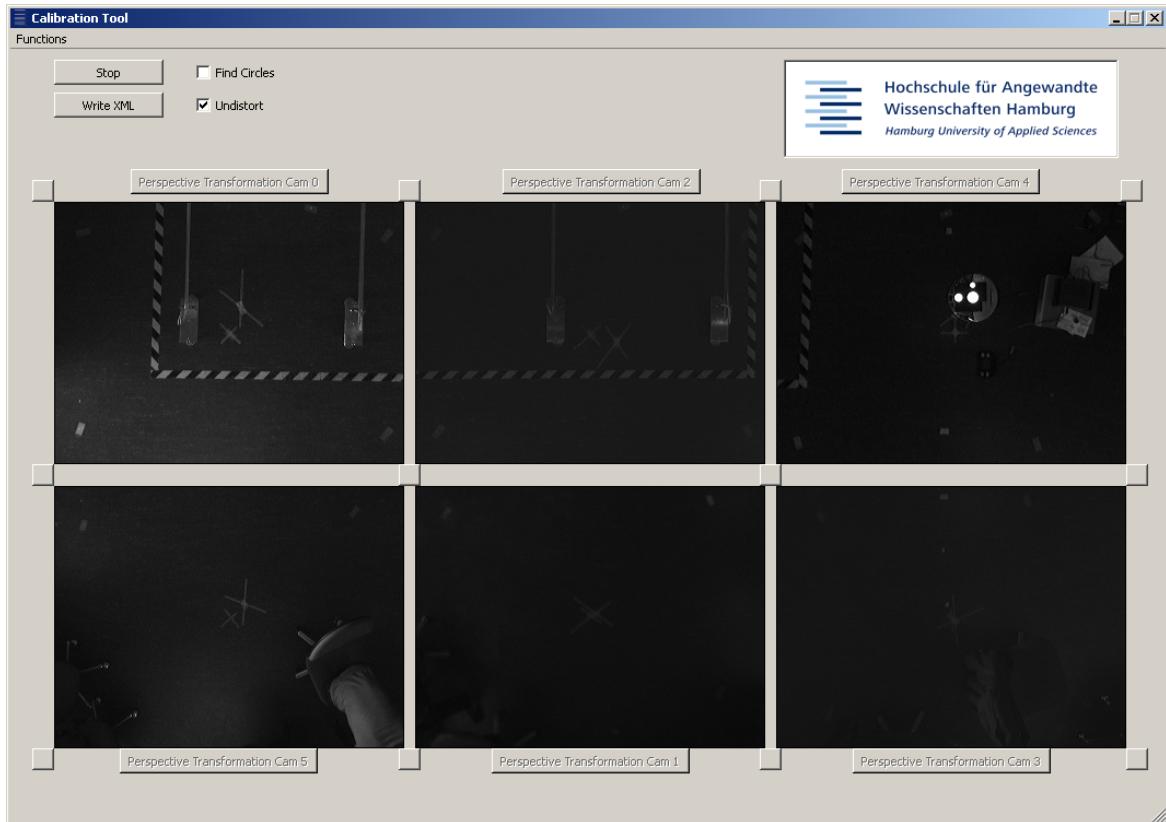


Abbildung 2.3.: Hauptfenster des „Calibration Tool“

Das Layout des Programmes ist zweigeteilt. Der obere Teil des Fensters ist für die Allgemeine Bedienung des „Calibration Tool“ zuständig. Darunter befinden sich die Bedienelemente für das Ermitteln und Berechnen der Transformationsmatrix.

Mit einem Klick auf den „Start“-Button wird das Programm gestartet. Daraufhin werden die Live Bilder der sechs Kameras in der Visualisierung angezeigt. Der Button ändert daraufhin seine Beschriftung auf „Stop“ und beendet die Live Bild Anzeige, wenn er gedrückt wird. Während des Live Bild Betriebes stehen dem Bediener über zwei Check Boxen folgende zusätzliche Möglichkeiten zur Verfügung. Die Check Box „Find Circles“ aktiviert die Kreiserkennung in den Kamerabildern. Wird ein Kreis in einem Kamerabild gefunden, wird der Kreis im Live Bild mit einer grünen Umrandung markiert. Die Kreiserkennung erfolgt mittels der „Hough Circles“ Transformation. Der Algorithmus ist sehr rechenaufwändig, was jedoch keine Problem für das Kalibrierungsprogramm ist, da es keine harten Echtzeitanforderungen erfüllen muss. Dafür ist der Algorithmus sehr robust und bietet eine hohe Genauigkeit.

Mit der zweiten Check Box „Undistort“ wird die Linsenverzeichnung aus den Kamerabildern herausgerechnet. Die Berechnung korrigiert die Verzerrung der Kameras und erzeugt

ein entzerrtes Bild. Ist die Check Box aktiviert, wird in den Live Bildern der Kamera direkt das entzerrte Bild angezeigt.

Bedienung des Programms zur Ermittlung der perspektivischen Verzerrung

Mit dem Programm wird die perspektivische Verzerrung der Kamerabilder ermittelt. Nachdem der Live Bild Modus gestartet wurde, muss der Benutzer die Funktionen „Find Circles“ und „Undistort“ aktivieren.

Zur Ermittlung der perspektivischen Verzerrung der Kameras wurden in den Raum zwölf Punkte exakt ausgemessen. Die Punkte sind auf dem Fußboden durch rote Aufkleber markiert. Der Benutzer muss nun das Stativ mit dem Referenzkreis über einen Punkt stellen, so wie in der Abbildung 2.4 gezeigt wird. Für die genaue Ausrichtung befindet sich an dem Stativ ein Lot dessen Spitze genau auf das Kreuz des roten Punktes zeigen muss. Es gibt keine feste Reihenfolge in der die Raumpunkte abgearbeitet werden müssen. Der Benutzer kann frei entscheiden welchen Punkt und welche Kamera er als Erstes kalibriert.

Für jeden der zwölf Punkte auf dem Fußboden ist jeweils ein Button in der Visualisierung vorhanden. Ist das Stativ ausgerichtet, muss von dem Benutzer der entsprechende Button gedrückt werden. Wird der Kreis korrekt erkannt, wird der Button, wie in Abbildung 2.5, Grün gefärbt. Die Raumpunkte, die in der Mitte liegen, wurden so vermessen, dass der Kreis in den verschiedenen Kameras immer deutlich sichtbar ist und fehlerfrei erkannt wird. Sollte der Kreis nicht korrekt erkannt werden, verfärbt sich der entsprechende Button Rot.

Wenn in einem Kamerabild alle Kreise in den vier Ecken korrekt erkannt worden sind, wird der Push Button „Perspective Transform Cam“ der entsprechenden Kamera freigeschaltet. Die Betätigung des Buttons startet die Berechnung der Transformationsmatrix aus den Raumkoordinaten, die aus der Initialisierungsdatei ausgelesen wurden, und den Kreiskoordinaten, die mit den Buttons ermittelt worden sind. Als Benutzerfeedback wird eine Nachricht in der Statusbar am unteren Rand des Hauptfensters angezeigt.

Nachdem alle Punkte aufgenommen sind und die Transformationsmatrizen berechnet sind, werden die neuen Parameter der Matrizen über den Button „Write XML“ abgespeichert.



Abbildung 2.4.: Raumkalibrierung mit Hilfe des Stativs

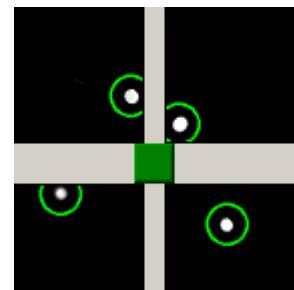


Abbildung 2.5.: Ermittlung der Transformationsmatrix

In der Abbildung 2.6 sind die zusätzlichen Funktionen des „Calibration Tools“ dargestellt. Unter dem Menüpunkt „Functions“ stehen die Funktionen „Correct Lense Distortion“ und „Camera Settings“ zur Verfügung.



Abbildung 2.6.: Funktionen des „Calibration Tool“

Bedienung Camera Settings

Mit der Funktion „Camera Settings“ können die Parameter der verschiedenen Kameras optimiert werden. Nach dem Auswählen der Funktion öffnet sich das „Camera Settings“ Fenster. Der Aufbau ist in der Abbildung 2.7 dargestellt.

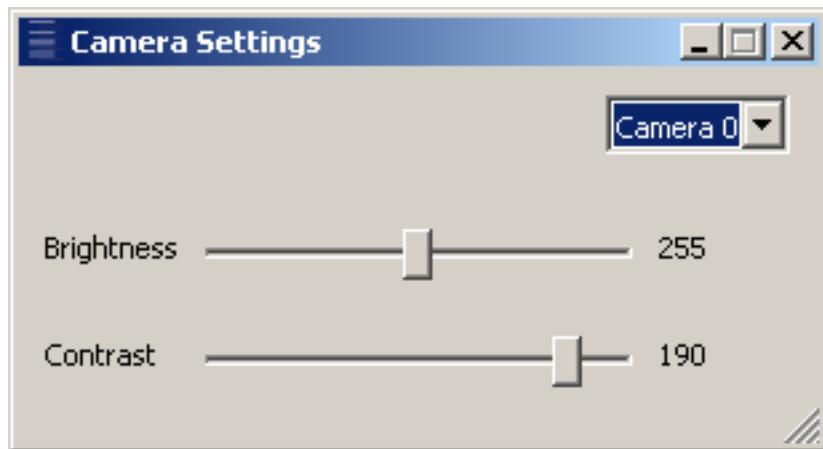


Abbildung 2.7.: Fenster der „Camera Settings“Funktion

Der Benutzer kann mit der Combo Box in der rechten oberen Ecke wählen, welche der sechs Kameras eingestellt werden soll. Dadurch ist es möglich jede Kamera individuell einzustellen. Unterhalb der Combo Box befinden sich zwei Schieberegler. Mit dem oberen Regler kann die Helligkeit der Kamera eingestellt werden. Der untere Schieberegler beeinflusst die Kontrasteinstellung der Kamera. Beim Wechseln der Kameras, werden die Schieberegler automatisch auf den in der Kamera eingestellten Wert positioniert. Dadurch erhält der Benutzer direkt eine Übersicht darüber wie weit die Parameter noch verändert werden können. Verändert der Benutzer die Parameterwerte, können die Auswirkungen direkt im Live Bild

des Hauptfensters beobachtet werden. Somit kann der Benutzer das aufgenommene Kamerabild schnell und einfach optimieren. Die eingestellten Parameter können nach der Optimierung über den „Write XML“-Button im Hauptfenster abgespeichert werden.

2.1.1. Programmaufbau zur Berechnung der Transformationsmatrix

Die Berechnung der Transformationsmatrix erfolgt in der Klasse *MainWindow*. Zunächst muss der Benutzer das „CalibrationTool“ über den „Start“-Button starten. Mit dem Button-Klick wird ein Timer-Event gestartet. Der Timer ruft anschließend alle 100ms neue Bilder von den sechs Kameras ab und sucht mit der Hough Circles Kreiserkennung alle Kreise in den Bildern. Die Koordinaten werden in das Array *gCirclesArray* in Zugehörigkeit zu der entsprechenden Kamera abgespeichert. Die Kreise werden zyklisch gesucht und immer neu in das Array geschrieben.

Der Benutzer kann jetzt mit der Ermittlung der Transformationsmatrix beginnen, indem er das Stativ auf einer Fußbodenmarkierung stellt. Zum Ermitteln der Kreiskoordinaten stehen im User Interface des Hauptfensters zwölf Buttons zur Verfügung. Jeder Button entspricht dabei einer Markierung auf dem Fußboden. Im Listing 2.1 ist exemplarisch der Code von einem Button dargestellt. Der Button in diesem Beispiel prüft, ob zwei Kameras einen Kreis im Überlappungsbereich ihrer Bilder finden. Wenn der Kreis gefunden wurde, wird die Schaltfläche grün eingefärbt und gibt den Benutzer die visuelle Bestätigung das die Messung geklappt hat. Wenn kein Kreis in dem Bildbereich der Kameras gefunden wurde, färbt sich der Button und signalisiert, dass die Position erneut gemessen werden muss. Zum Berechnen der Transformationsmatrizen von den sechs Kameras müssen alle zwölf Buttons grün eingefärbt sein.

Listing 2.1: Button zur Ermittlung der Kreiskoordinaten

```
void MainWindow :: on_pushButton_TCL_clicked ()
{
    bool circleFound = false;
    circleFound = checkCircles(CAMERA_0, AREA_TR, CAMERA_2, AREA_TL);

    if (circleFound)
        ui->pushButton_TCL->setStyleSheet ("background-color : _green ");
    else
        ui->pushButton_TCL->setStyleSheet ("background-color : _red ");

    gArrayEnableTransform [AREA_TOP][AREA_CL] = circleFound;
}
```

Die Funktion *checkCircles* aus dem Listing 2.2 überprüft, ob in allen Kameras die Kreise in den entsprechenden Ecken der Kameras gefunden werden. Da ein Kreis von einer, zwei oder vier Kameras gleichzeitig gefunden werden muss, wurde die Funktion als überladene

Funktion programmiert. Das heißt die Kamera kann entweder mit zwei, vier oder acht Parametern aufgerufen werden. Als Parameter wird die zu überprüfende Kamera mit dem entsprechenden Bildbereich übergeben. Der Codeausschnitt zeigt die Funktion für die Überprüfung von zwei Kameras mit den vier entsprechenden Parametern. Die Funktion *checkCircles* liefert einen booleschen Rückgabewert. Wenn in jedem Kamerabild der Kreis an der richtigen Stelle gefunden wurde, wird *True* als Wert zurückgegeben.

Listing 2.2: Überprüfung der gefundenen Kreise

```
bool MainWindow::checkCircles(int cameraIndex_1, int areaCamera_1, int
cameraIndex_2, int areaCamera_2)
{
    bool circleFound_1 = false;
    bool circleFound_2 = false;
    if (videoCapture[cameraIndex_1]. isOpened())
        circleFound_1 = FindEdgeCircles(cameraIndex_1, areaCamera_1);
    if (videoCapture[cameraIndex_2]. isOpened())
        circleFound_2 = FindEdgeCircles(cameraIndex_2, areaCamera_2);

    return (circleFound_1&&circleFound_2);
}
```

Das Listing 2.3 zeigt die Funktion *FindEdgeCircles*. Der Funktion werden die Kameranummer und der zu untersuchende Bildbereich übergeben. In der Funktion werden alle Kreise aus dem Array *gCirclesArray*, die im entsprechenden Kamerabild mit der Hough Circles Kreiserkennung gefunden worden sind, überprüft. Über eine *If*-Abfrage wird überprüft in welchem Bildbereich nach Kreisen gesucht werden soll. Mit einer weiteren *If*-Abfrage werden die Koordinaten des Kreises abgefragt und überprüft, ob sie in dem entsprechenden Bildbereich sind. Wenn die Bedingungen erfüllt sind, werden die Kreiskoordinaten in das Array *arrayCircles* abgespeichert. In dem Array sind die Koordinaten der richtig überprüften Kreise für jede Kamera und dem entsprechenden Bildbereich abgespeichert. Wenn ein richtiger Kreis gefunden wurde, gibt die Funktion *True* als Wert zurück.

Wenn kein Kreis gefunden wird, werden bei der Kamera in dem entsprechenden Bildbereich die Kreiskoordinaten auf den Wert Null zurückgesetzt. Die Funktion liefert dann als Rückgabewert *False*.

Listing 2.3: Überprüfung der Lage der gefundenen Kreise

```
bool MainWindow::FindEdgeCircles(int cameraIndex, int area)
{
    if (gCirclesArray[cameraIndex].size() > 0)
    {
        for (unsigned int i = 0; i < gCirclesArray[cameraIndex].size();
              i++)
        {
            if (gCirclesArray[cameraIndex][i].area == area)
                arrayCircles.push_back(gCirclesArray[cameraIndex][i]);
        }
    }
}
```

```

    // save coordinates of the circle
    arrayCircles [ cameraIndex ][ area ][ X_Value ] = gCirclesArray [
        cameraIndex ][ i ][ 0 ];
    arrayCircles [ cameraIndex ][ area ][ Y_Value ] = gCirclesArray [
        cameraIndex ][ i ][ 1 ];

    if ( area == AREA_TL )
        if ( (cvRound( gCirclesArray [ cameraIndex ][ i ][ 0 ]) < (
            CAMERA_IMG_WIDTH/2)) && (cvRound( gCirclesArray [
            cameraIndex ][ i ][ 1 ]) < (CAMERA_IMG_HEIGTH/2)))
            return true;
    if ( area == AREA_TR )
        if ( (cvRound( gCirclesArray [ cameraIndex ][ i ][ 0 ]) > (
            CAMERA_IMG_WIDTH/2)) && (cvRound( gCirclesArray [
            cameraIndex ][ i ][ 1 ]) < (CAMERA_IMG_HEIGTH/2)))
            return true;
    if ( area == AREA_BL )
        if ( (cvRound( gCirclesArray [ cameraIndex ][ i ][ 0 ]) < (
            CAMERA_IMG_WIDTH/2)) && (cvRound( gCirclesArray [
            cameraIndex ][ i ][ 1 ]) > (CAMERA_IMG_HEIGTH/2)))
            return true;
    if ( area == AREA_BR )
        if ( (cvRound( gCirclesArray [ cameraIndex ][ i ][ 0 ]) > (
            CAMERA_IMG_WIDTH/2)) && (cvRound( gCirclesArray [
            cameraIndex ][ i ][ 1 ]) > (CAMERA_IMG_HEIGTH/2)))
            return true;
    }

}

arrayCircles [ cameraIndex ][ area ][ X_Value ] = 0;
arrayCircles [ cameraIndex ][ area ][ Y_Value ] = 0;
return false;
}

```

Der Codeausschnitt aus dem Listing 2.4 prüft fortlaufend, ob die Buttons zum Berechnen der Transformationsmatrix für die einzelnen Kamerabilder freigeschaltet werden können. Die Funktion schaut ob die vier Button an den Ecken des jeweiligen Kamerabildes grün hinterlegt sind. Der Zustand der Buttons ist in dem zweidimensionalen Array *gArrayEnableTransform* abgespeichert. Sind alle vier Array-Elemente für eine Kamera *True* bedeutet das, dass für jede Kameraecke gültige Kreiskoordinaten gefunden worden sind. Dann wird die Freigabe für den Button zur Berechnung der Transformationsmatrix gegeben.

Wurden nicht in allen Ecken des Kamerabildes die Koordinaten der Kreise ermittelt, kann keine Transformationsmatrix bestimmt werden.

Listing 2.4: Freigabe des Buttons zur Berechnung der Transformationsmatrix

```

if ( gArrayEnableTransform [AREA_TOP][AREA_LL] &&
    gArrayEnableTransform [AREA_TOP][AREA_CL] &&
    gArrayEnableTransform [AREA_CENTER][AREA_LL] &&
    gArrayEnableTransform [AREA_CENTER][AREA_CL] )
    ui ->pushButton_CalibrateCam_0 ->setEnabled (true);
else
    ui ->pushButton_CalibrateCam_0 ->setEnabled (false);

```

Wenn der Push Button für die entsprechende Kamera freigegeben ist, kann dieser Betätigt werden. Der Klick führt die im Listing 2.5 gezeigten Aktionen durch. Zunächst wird dem Benutzer ein Feedback gegeben, indem eine Informationsmeldung in der Statusleiste angezeigt wird. Anschließend wird das Array *quelle* angelegt, das die Quellkoordinaten für die perspektivische Transformation enthält. In dem Array werden die Koordinaten der Kreise kopiert, die in den Ecken der jeweiligen Kamera gefunden worden sind. Danach wird das Array *ziel* beschrieben. In dem Array stehen die Koordinaten auf die das Kamerabild transformiert werden soll. Als Zielkoordinaten dienen die im Raum vermessenen roten Punkte. Die Koordinaten der Punkte wurden aus der Initialisierungsdatei eingelesen und werden jetzt in das Zielarray der entsprechenden Kamera kopiert. Anschließend erfolgt der Aufruf der Funktion *CalculateTransformMatrix* zur Berechnung der Transformationsmatrix.

Listing 2.5: Ermittlung der Quell- und Zielkoordinaten

```

void MainWindow :: on_pushButton_CalibrateCam_0_clicked ()
{
    ui ->statusBar ->showMessage (" Calculate_Transformation_Matrix_0... " ,
        2000);

    vector<Vec2f> quelle(4), ziel(4);
    int cameraIndex = CAMERA_0;

    quelle [AREA_TL] = Vec2f( arrayCircles [cameraIndex] [AREA_TL] [X_Value] ,
        arrayCircles [cameraIndex] [AREA_TL] [Y_Value] );
    quelle [AREA_TR] = Vec2f( arrayCircles [cameraIndex] [AREA_TR] [X_Value] ,
        arrayCircles [cameraIndex] [AREA_TR] [Y_Value] );
    quelle [AREA_BL] = Vec2f( arrayCircles [cameraIndex] [AREA_BL] [X_Value] ,
        arrayCircles [cameraIndex] [AREA_BL] [Y_Value] );
    quelle [AREA_BR] = Vec2f( arrayCircles [cameraIndex] [AREA_BR] [X_Value] ,
        arrayCircles [cameraIndex] [AREA_BR] [Y_Value] );

    ziel [AREA_TL] = pos_TLL;
    ziel [AREA_TR] = pos_TCL;
    ziel [AREA_BL] = pos_CLL;
    ziel [AREA_BR] = pos_CCL;

    CalculateTransformMatrix (cameraIndex , quelle , ziel);
}

```

Die Funktion *CalculateTransformMatrix* aus dem Listing 2.6 berechnet mit der Funktion *getPerspectiveTransform* die Transformationsmatrix. Die Funktion *getPerspectiveTransform* wird von der OpenCV Bibliothek zur Verfügung gestellt und benötigt als Parameter die Quell- und Zielkoordinaten des Bildes. Als Rückgabewert liefert die Funktion die Transformationsmatrix der Größe 3x3. Die einzelnen Matrixelemente der Transformationsmatrix werden in die QList *List* geschrieben. In der Liste sind für jede Kamera alle Parameter abgespeichert. Mit der Liste erfolgt das Beschreiben einer neuen XML-Datei.

Mit dem Beenden der Funktion ist die Ermittlung der Transformationsmatrix für eine Kamera abgeschlossen.

Listing 2.6: Berechnung der Transformationsmatrix

```
void MainWindow::CalculateTransformMatrix(int cameraIndex, vector<Vec2f> quelle, vector<Vec2f> ziel)
{
    cv::Mat transformMatrix;

    transformMatrix = getPerspectiveTransform(quelle, ziel);

    List[cameraIndex]["t1"] = QString::number(transformMatrix.at<double>(0,0));
    List[cameraIndex]["t2"] = QString::number(transformMatrix.at<double>(0,1));
    List[cameraIndex]["t3"] = QString::number(transformMatrix.at<double>(0,2));
    List[cameraIndex]["t4"] = QString::number(transformMatrix.at<double>(1,0));
    List[cameraIndex]["t5"] = QString::number(transformMatrix.at<double>(1,1));
    List[cameraIndex]["t6"] = QString::number(transformMatrix.at<double>(1,2));
    List[cameraIndex]["t7"] = QString::number(transformMatrix.at<double>(2,0));
    List[cameraIndex]["t8"] = QString::number(transformMatrix.at<double>(2,1));
    List[cameraIndex]["t9"] = QString::number(transformMatrix.at<double>(2,2));
}
```

2.1.2. Perspektivische Verzerrung Beschreibung

Die perspektivische Verzerrung entsteht dadurch, dass eine optimale, parallele Ausrichtung der Kameras an der Decke zum Fußboden nicht möglich ist. Bereits kleine Abweichungen sorgen dafür, dass die Kameras keinen rechteckigen Ausschnitt vom Fußboden aufnehmen,

sondern einen verzerrten trapezförmigen Ausschnitt. Die Verzerrung besteht aus einer Rotation, Skalierung, Verschiebung und Scherung des Bildes.

In der Abbildung 2.8 ist beispielhaft eine extreme perspektivische Verzerrung dargestellt.

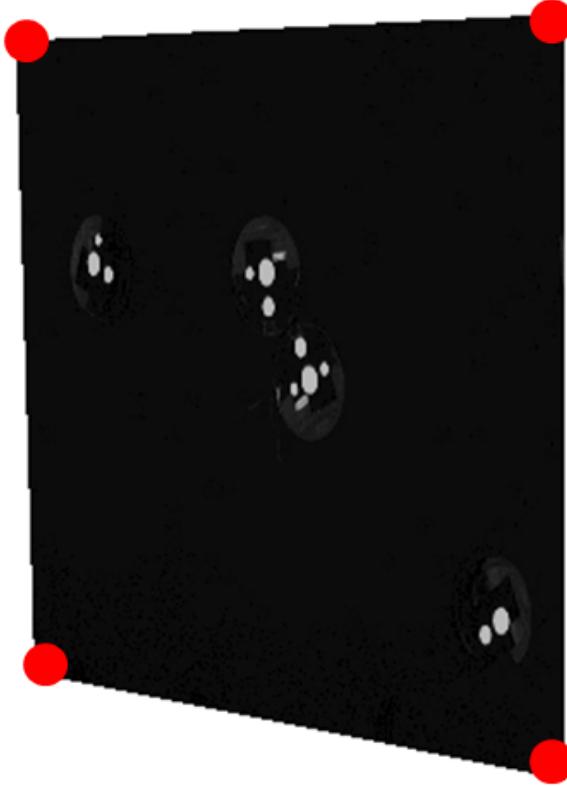


Abbildung 2.8.: perspektivisch verzerrtes Kamerabild

Durch das Vermessen des Raumes sind die Spielfeldkoordinaten der roten Kreise bekannt. Nun müssen nur noch die Bildpunktkoordinaten der Kreise im verzerrten Bild ermittelt werden. Aus den vier Ausgangs- und Zielkoordinaten kann die Transformationsmatrix T für die Kamera bestimmt werden.

$$\begin{bmatrix} x'_B \\ y'_B \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} t_1 & t_2 & t_3 \\ t_4 & t_5 & t_6 \\ t_7 & t_8 & t_9 \end{bmatrix}}_T \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} \quad (2.1)$$

Die Transformationsmatrix ist eine 3×3 Matrix. Die Parameter t_1, t_2, t_4, t_5 beeinflussen die Skalierung, Rotation und Scherung des Bildes. Damit wird das Bild zu einem Rechteck

geformt und die Auflösung der Kamera wird auf die Größe des Spielfeldes skaliert. Für die Verschiebung des Bildes sind die Parameter t_3 und t_6 zuständig. Das ist notwendig, da jede Kamera einen anderen Teil des Spielfeldes aufnimmt. Mit der Verschiebung werden die Bildpunkte in den richtigen Koordinatenbereich des Spielfeldes transformiert. Dadurch wird das Bild der Kamera an die richtige Stelle von dem Spielfeld geschoben. Die Parameter t_7 und t_8 sorgen für eine zusätzliche Verkippung des Bildes. In den von uns ermittelten Transformationsmatrizen sind diese beiden Parameter jedoch sehr klein und haben nur einen geringen Einfluss auf das Bild. Das Matrixelement $t_9 = 1$ und hat somit keinen Einfluss auf das Bild.

Die einzelnen Bildpunkte (x_B, y_B) des verzerrten Ausgangsbildes werden mit der Transformationsmatrix T multipliziert. Die Rechnung liefert als Ergebnis die neuen Bildpunkte (x'_B, y'_B) , die das neue perspektivisch korrekte Bild bilden.

In der Abbildung 2.9 ist das entzerrte Ausgangsbild der Berechnung dargestellt. Das Bild ist rechteckig und gerade gedreht.



Abbildung 2.9.: perspektivisch entzerrtes Kamerabild

2.1.3. Benutzeroberfläche Correcting Lens Distortion

Das Calibration Tool bietet die Möglichkeit über die Menübar *Functions* die Option *correct lence distortion* auswählen. Durch diese Auswahl wird ein Objekt der Klasse *distortion-coefficients* erzeugt, welches von der public Klasse QMainWindow erbt. Es öffnet sich dadurch eine eigenständige neue Benutzeroberfläche *Correcting Lens Distortion*.

Nach dem Ausführen der *CalibrationTool.exe* Datei startet das Kalibrierungsprogramm. Bei Start wird automatisch die *fieldcoordinates.ini*-Datei eingelesen. Mit dieser Datei werden

die im Raum vermessenen Koordinaten initialisiert. Ist keine Datei vorhanden werden für die vermessenen Positionen automatisch die Standardwerte übernommen.



Abbildung 2.10.: distortioncoefficients

In der Benutzeroberfläche kann durch eine Combo Box in der rechten oberen Ecke eine der sechs Kameras ausgewählt werden. Das Livebild wird mittig auf einem Label angezeigt. In der linken oberen Ecke ist eine Check Box mit der Beschriftung *Find Circles Grid* zu finden. Durch das setzen der Check Box wird nach einem definierten Punkt-Gitter der Größe 5x4 gesucht und sobald es gefunden wird markiert. Weiterhin werden durch das setzen der Check Box nun vier, an den Ecken des Labels liegende, Push Buttons freigegeben. Ziel ist es nun das Punkt-Gitter möglichst weit in eine der Ecken zu platzieren um dann den in der Ecke befindlichen Push Button zu aktivieren. Daraufhin werden die Zentren des Punkt-Gitters in einer Vektortabelle abgespeichert. Als Benutzerfeedback wird eine Nachricht in der Statusbar zurückgegeben. Zudem färbt sich der angeklickte Push Button bei Erfolg grün. Sollten die Zentren des Punkt-Gitters nicht erfasst worden sein, z.B. da es nicht vollständig auf dem Livebild abgebildet ist, färbt sich der Push Button rot. Die Messung muss solange wiederholt werden bis der Button grün gefärbt ist. Hierzu ist ggf. eine Verschiebung des

Punkt-Gitters sinnvoll. Es spielt dabei keine Rolle in welcher Ecke begonnen wird. Anschließend wird der Vorgang für die verbliebenen drei Ecken wiederholt. Sind die Zentren für alle vier Ecken bekannt wird der untere, mittig angeordnete Push Button *Calculate Calibration Data* freigegeben. Die Benutzung des Buttons bestimmt aus den vier Datensätzen die Kameramatrix und die Verzeichnungskoeffizienten die nun im Hauptprogramm als Datensatz zur Verfügung stehen. Mit den weiteren Kameras kann nun genauso verfahren werden. Wird eine andere Kamera ausgewählt werden sämtliche Einstellungen zurückgesetzt. D. h. der beschriebene Vorgang muss für eine Kamera vollständig durchgeführt werden um die Zielparameter dem Hauptprogramm übergeben zu werden.

2.1.4. Linsenverzeichnung Beschreibung

Die Linsenverzeichnung ist ein geometrischer Abbildungsfehler. Sie wirkt sich auf gerade Linien, deren Abbild nicht durch die Bildmitte geht, dadurch aus das diese gekrümmt wiedergegeben werden. Man spricht je nach Verzeichnung meist von einer kissen- oder tonnenförmigen Verzeichnung. Als Beispiel weisen Fischaugen-Objektive z.B. eine starke tonnenförmige Verzeichnung auf.

Die Linsenverzeichnung setzt sich dabei aus einer radialen und einer tangentialen Verzeichnung, der Brennweite bzw. dem Fokus sowie die Verschiebung des Bildsensors zum Kamerazentrum zusammen. Das in dem Algorithmus verwendet Verzeichnungsmodell orientiert sich dabei an dem von D. C. Brown entwickelten „Plumb Bob“-Modell, auch „Brown-Conrady model“ genannt.¹²

Die radiale Verzeichnung als Folge der Aberration des Objektivs ist rotationssymmetrisch um den Symmetriepunkt in Abhängigkeit vom radialen Abstand r . Zur Einbeziehung der radialen Linsenverzeichnung ergibt sich dadurch mit $r^2 = x^2 + y^2$ und dem um normierten Bildprojektionsvektor $\mathbf{x}_n = [x \ y]^T$ die Punktkoordinate \mathbf{x}_d :

$$\mathbf{x}_d = \begin{bmatrix} x_d(1) \\ x_d(2) \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \mathbf{x}_n + \mathbf{dx} \quad (2.2)$$

wobei \mathbf{dx} der tangentiale Verzeichnungsvektor ist:

$$\mathbf{dx} = \begin{bmatrix} 2p_1xy + p_2(r^2 + 2x^2) \\ p_1(r^2 + 2y^2) + 2p_2xy \end{bmatrix} \quad (2.3)$$

¹Quelle: http://www.vision.caltech.edu/bouguetj/calib_doc/, abgerufen am 14. Mai 2014

²Quelle: http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html, abgerufen am 14. Mai 2014

Zusammengefasst ergibt sich der unbekannte, zu bestimmende Verzeichnungskoeffizientenvektor zu:

$$\mathbf{Distortion}_{coefficient} = [k_1 \ k_2 \ p_1 \ p_2 \ k_3] \quad (2.4)$$

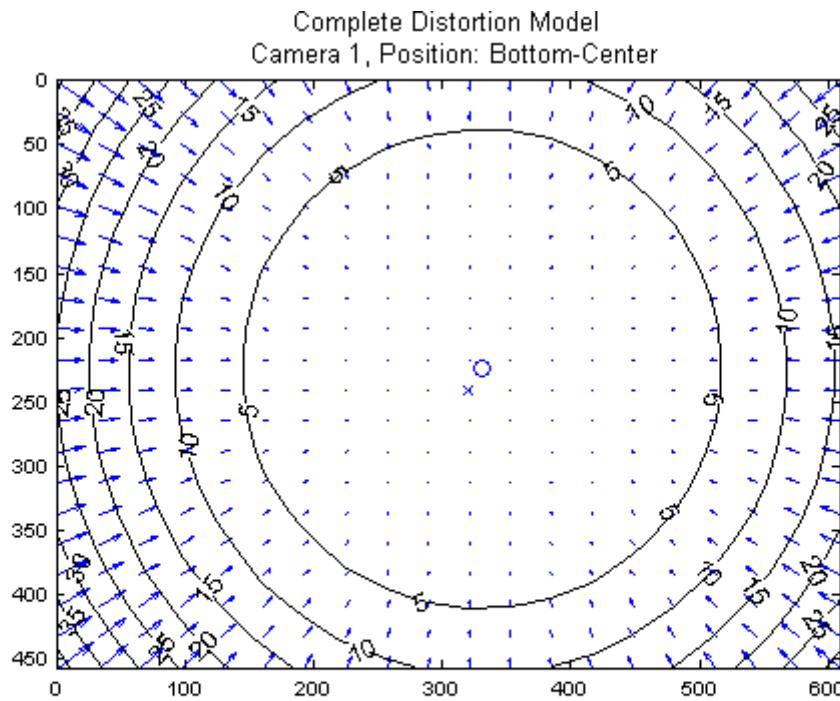
Die Pixel-Koordinate $[x' \ y' \ 1]^T$ sowie der durch die Verzeichnung neu entstandene Vektor \mathbf{x}_d stehen über die Kalibriermatrix bzw. Kameramatrix \mathbf{K} im Zusammenhang:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}} \begin{bmatrix} x_d(1) \\ x_d(2) \\ 1 \end{bmatrix} \quad (2.5)$$

Dabei fließen die Brennweite f in x - und y -Anteil aufgeteilt und die Position des Bildsensors c , der meist verschoben ist, in die Matrix mit ein. Auch diese Parameter müssen bestimmt werden. Sie werden auch als intrinsische Parameter bezeichnet.

OpenCV stellt dafür die Funktion *calibrateCamera()*³ zur Verfügung. Dahinter befindet sich ein Algorithmus basierend auf J. Y. Bouguet und Z. Zhang. Die Referenzkoordinaten und die Bildprojektionskoordinaten eines Objekts müssen hierfür bekannt sein. Dies wird am besten dadurch erreicht das ein Objekt verwendet wird, welches bekannte Koordinaten aufweist die leicht zu detektieren sind (z.B. Schachbrettmuster oder Punkt-Gitter). Diese werden über die Kamera erfasst und mit den Referenzkoordinaten der Funktion übergeben. Daraufhin bestimmt der Algorithmus zunächst die intrinsischen Parameter wofür die Verzeichnungskoeffizienten auf Null gesetzt werden. Sind die Werte der Kameramatrix bekannt, wird die initiale Kamerapose geschätzt. Hierzu sind die Referenzkoordinaten und Bildprojektionskoordinaten wichtig. Im letzten Schritt folgt ein globaler Levenberg-Marquardt-Algorithmus zur Ermittlung und Optimierung der Verzeichnungskoeffizienten.

³Quelle: http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html, abgerufen am 14. Mai 2014



Focal Length (f_x, f_y) = (505.675, 505.373)
 Principal Point (c_x, c_y) = (330.188, 222.668)
 Radial coefficients (k_1, k_2, k_3) = (-0.2045, 0.0723, -0.0142)
 Tangential coefficients (p_1, p_2) = (0.0005447, 0.0001869)

Abbildung 2.11.: Vollständiges Verzeichnungsmodell der Kamera 1

Zur Verdeutlichung ist das Verzeichnungsmodell mit den durch den Algorithmus ermittelten Koeffizienten der Kamera 1 (Position: Bottom-Center) in Abbildung 2.11 dargestellt. Das Bild zeigt den vollständigen Störeinfluss auf die einzelnen Pixel des Bildes. Insbesondere in der linken unteren Ecke werden bei dieser Kamera Punkte um mehr als 35 Pixel verschoben. Das Kreuz stellt dabei die Verschiebung des Bildmittelpunkts von der mit dem Kreis gekennzeichneten optischen Zentrum der Linse dar.

Im Anhang finden sich die Abbildungen zu allen sechs Kameras. Zudem befinden sich dort Abbildungen die jeweils nur den tangentialen oder radialen Einfluss darstellen.

In den Abbildungen 2.12 und 2.13 sind die Auswirkungen dargestellt, die die Kompen-sation der Linsenverzeichnung bewirkt. Im originalen, verzerrten Kamerabild ist deutlich zu erkennen wie das Punktgitter und die Markierungslinie auf dem Fußboden verkrümmt sind. Die äußeren Bildbereiche werden dabei stärker verzerrt als die inneren. Nach dem Entzerren ist die Linie gerade und die Punkte bilden ein rechteckförmiges Gitter.

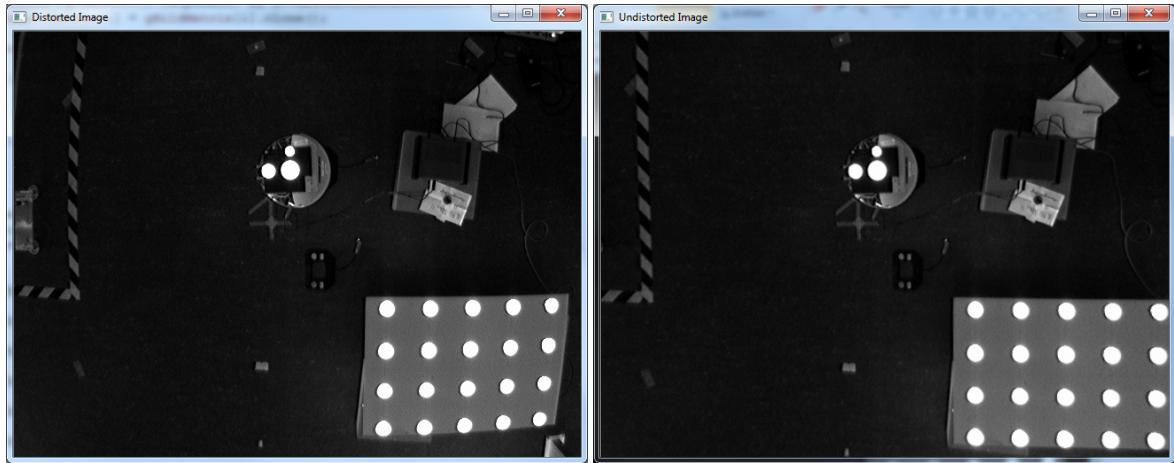


Abbildung 2.12.: Verzerrtes Kamerabild

Abbildung 2.13.: Entzerrtes Kamerabild

2.1.5. Programmaufbau *distortioncoefficients*

Durch die Erzeugung eines Objektes der Klasse *distortioncoefficients* wird zunächst ein QMainWindow erzeugt. Ebenfalls werden zu Beginn einzelne Variablen initialisiert. Darunter befindet sich z. B. das spätere InputArray *RefCenters* welches die Referenzzentren des Punkt-Gitters für die anschließende Berechnung der Verzerrung enthält. Da festgelegt wurde, dass Datensätze aus den vier Ecken aufgezeichnet werden sollen, werden hier viermal die Referenzpunkte initialisiert. Ebenfalls werden die Kameramatrizen, die von dem Hauptprogramm die Livebilder enthalten mit Einsen vorinitialisiert. Die Initialisierung hat vor allem den Vorteil, dass wenn kein Bild der ausgewählten Kamera zur Verfügung steht das Bild deutlich durch das Muster abweicht im Gegensatz zu den ohnehin sehr schwach beleuchteten Aufnahmen.

Listing 2.7: Initialisierung der Variablen

```
ui->setupUi(this);
for( int i = 0; i<6; i++){
    gCamselectMatrix[ i ]=Mat:: ones( 480 ,640 ,CV_32F );
    cameraIsOpen[ i ] = false ;
}
refCenters . resize( 4 );
float squareSize = 280.0;
for( int noRefs=0; noRefs<4; noRefs++){
    for( int i=0; i<4; i++){
        for( int j=0; j<5; j++){
            refCenters [ noRefs ]. push_back( Point3f( float( j*squareSize )
                , float( i*squareSize ) , 0 ));
```

```
    }
}
```

Da die Erkennung des gewünschten Punkt-Gitters rechenintensiv ist wird für das Programm eigens ein langsames Timerevent ausgeführt. Dabei wird zunächst abgefragt ob ein vollständiger, aus allen vier Ecken zusammengetragener, Datensatz (*foundData*) der Punkt-Gitter zur Verfügung steht. Ist dies der Fall wird der Button *Calculate Calibration Data* freigegeben. Ist die Checkbox *Find Circles Grid* aktiviert wird hiernach die Funktion *generateCalibrationData* mit einer temporären Vektortabelle ausgeführt. Dadurch wird nach dem Punkt-Gitter gesucht und wenn es gefunden ist visualisiert. Anschließend wird das ausgewählte Kamerabild auf das Label übertragen.

Listing 2.8: Ablauf des Timerevent

```
void DistortionCoefficients :: timerEvent(QTimerEvent *event)
{
    if ( foundData[0]&&foundData[1]&&foundData[2]&&foundData [3] )
        ui->pushButton_calibrate->setEnabled(true);
    else
        ui->pushButton_calibrate->setEnabled(false);

    if ( circlesGrid){
        bool found = generateCalibrationData (imageCentersTemp [0]);
    }

    QPixmap pixmap;
    pixmap = QPixmap :: fromImage (QImage ((unsigned char *) gCamselectMatrix
        [gCamselect].data , gCamselectMatrix [gCamselect].cols ,
        gCamselectMatrix [gCamselect].rows , QImage :: Format_RGB888));
    ui->label_Image_Cam->setPixmap (pixmap);
}
```

Wird einer der vier Buttons (Top Left, Bottom Left, usw.) betätigt, wird der entsprechende Vektorcontainer der tatsächlichen Zentren auf die Größe Null gesetzt um alte Datensätze zu löschen. Danach wird die Funktion *generateCalibrationData* ausgeführt, an der der Vektorcontainer übergeben wird. Die Funktion liefert eine boolesche Variable zurück die in das entsprechende Arrayfeld von *foundData* eingetragen wird. Ist diese boolesche Variable erfüllt wird die Farbe des entsprechenden Buttons grün gefärbt. Im nichterfüllten Fall wird der entsprechende Knopf rot gefärbt.

Listing 2.9: Button zum Speichern der Punkt-Gitter Koordinaten

```
void DistortionCoefficients :: on_pushButton_t1_clicked()
{
```

```

imageCenters[0].resize(0);
ui->statusbar->showMessage("Generate_Calibration_Data_for_Corner_Top
-Left ... ", 1000);
foundData[0] = generateCalibrationData(imageCenters[0]);
if(foundData[0])
    ui->pushButton_t1->setStyleSheet("background-color:_green");
else
    ui->pushButton_t1->setStyleSheet("background-color:_red");
}

```

Der Funktion *generateCalibrationData* wird ein leerer Vektorcontainer *imageCenters[i]* übergeben, entsprechend der ausgewählten Ecke. Zunächst wird ein unscharfes Graubild des Livebildes erzeugt. Hieraus entsteht ein invertiertes Bild. Dieses Bild wird zusammen mit dem leeren Vektorcontainer und der Größe des Punkt-Gitters der Funktion *findCirclesGrid* übergeben. Hier werden die Zentren der Punkte bestimmt und in den Vektorcontainer eingetragen. Abschließend werden die gefundenen Koordinaten mit Hilfe der Funktion *drawChessboardCorners* in dem ausgewählten Livebild visualisiert.

Listing 2.10: Ermitteln der Koordinaten des Punkt-Gitters

```

bool DistortionCoefficients :: generateCalibrationData( std :: vector<cv :: Point2f> &centersVec )
{
    /// Convert image to gray and blur it
    Mat src_gray;
    cvtColor(gCamselectMatrix[gCamselect], src_gray, CV_BGR2GRAY );
    blur( src_gray , src_gray , Size(3,3) );
    src_gray.convertTo(src_gray,-1,1,-128);
    src_gray.convertTo(src_gray,-1,10,-300);
    // invert picture
    Mat sub_mat = Mat::ones(src_gray.size(), src_gray.type())*255;
    subtract(sub_mat, src_gray, src_gray);

    Size patternSize(5,4); //number of centers => 20 centers
    bool patternfound = findCirclesGrid(src_gray, patternSize,
        centersVec);
    drawChessboardCorners(gCamselectMatrix[gCamselect], patternSize, Mat
        (centersVec), patternfound);
    qDebug() << patternfound;
    return patternfound;
}

```

Die durch die aktivieren des Buttons *Calculate Calibration Data* aufgerufene Funktion *on_pushButton_calibrate_clicked* berechnet aus den Vektorcontainern Referenzzentren und Bildzentren die Kameramatrix sowie die Verzerrungskoeffizienten. Zuletzt wird die boolesche Variable *newCalibrateData* gesetzt. Diese ist public und wird im Hauptprogramm

zyklisch abgefragt. Ist sie wahr wird der neu erzeugte Datensatz ins Hauptprogramm übernommen und die Variable rückgesetzt.

Listing 2.11: Berechnung der Verzerrungskoeffizienten

```
void DistortionCoefficients :: on_pushButton_calibrate_clicked()
{
    cameraMatrix = Mat::eye(3, 3, CV_32F);
    cameraMatrix.at<double>(0,0) = 1.0;
    distCoeffs = Mat::zeros(8, 1, CV_32F);
    vector<Mat> rvecs, tvecs;
    double rmsError = calibrateCamera(refCenters,
                                       imageCenters,
                                       Size(480, 640),
                                       cameraMatrix,
                                       distCoeffs,
                                       rvecs,
                                       tvecs);
    newCalibrateData = true;
}
```

2.2. XML

2.2.1. Aufbau der XML-Datei

Der Aufbau der XML-Datei CameraCalibrationData ist hierarchisch gestaltet. Sie beginnt, wie in Abbildung 2.14 dargestellt, mit der zuerst definierten Kamera mit den Attributen ID, Name und Position. Die erste Unterebene bilden die Parameter der Kameramatrix ab, gefolgt von den Verzeichnungskoeffizienten. Schließlich folgen die Elemente der perspektivischen Verzerrungsmatrix und die Kameraeinstellungen wie Helligkeit, Kontrast.

```

<?xml version="1.0" encoding="UTF-8" ?>
- <CameraCalibrationData>
  - <Camera id="0" name="TopLeft" pos="tl">
    <Matrix fx="1921.34" fy="1920.19" cx="337.881" cy="196.891" />
    <LensDistortion p1="-3.8948614572e-03" p2="-8.1788117475e-03"
      k1="-3.4127742557e+00" k2="1.5338954367e+01" k3="-
      3.0913032842e+01" />
    <TransformationMatrix t1="3.90393" t2="0.0492404" t3="161.615"
      t4="0.119186" t5="-3.87396" t6="3666.72" t7="2.11826e-05"
      t8="-3.66792e-06" t9="1" />
    <Settings brg="128" cnt="64" exp="-7" />
  </Camera>
  - <Camera id="1" name="BottomCenter" pos="bc">
    <Matrix fx="505.675" fy="505.373" cx="330.188" cy="222.668" />
    <LensDistortion p1="5.4465689753e-04" p2="1.8686258427e-04"
      k1="-2.0448412765e-01" k2="7.2300324973e-02" k3="-
      1.4198938493e-02" />
    <TransformationMatrix t1="4.52759" t2="-0.178034" t3="2464.52"
      t4="-0.210354" t5="-4.48808" t6="2167.35" t7="1.1656e-05"
      t8="9.3212e-06" t9="1" />
    <Settings brg="-87" cnt="26" exp="-7" />
  </Camera>

```

Abbildung 2.14.: Aufbau der XML-Datei

2.2.2. Einlesen der XML-Datei

Beim Ausführen der „CalibrationTool.exe“ wird zuerst die XML-Datei mit den Kameraparametern eingelesen. Das Programm öffnet die vom Benutzer ausgewählte Datei und liest mit einem *QXmlStreamReader* die Attribute der einzelnen Kameras ein. Die Streamreader Klasse ermöglicht das schnelle Lesen einer strukturierten XML-Datei. Zur Speicherung der Kameraattribute zur Laufzeit wurde eine *QList< QMap >* List für angelegt in der die sechs Kameras gespeichert sind. *QMap* ist eine Containerklasse die es ermöglicht Parameter über einen Schlüsselnamen und den Parameterwert abzuspeichern. Dadurch konnte die Liste mit den gleichen Attributen aufgebaut werden, wie sie in der XML-Datei vorhanden sind. Der Streamreader ordnet beim Programmstart den Attributen der Liste die entsprechenden Werte zu.

2.2.3. Erstellung der XML-Datei

Wenn sich während der Laufzeit Parameter der Kamera ändern, wie zum Beispiel die Kontrasteinstellung, wird die Änderung direkt in der *QList* übernommen. Durch die Container-

klasse *QMap* ist es möglich direkt auf das Attribut für die Kontrasteinstellung der Kamera zuzugreifen und dieses zu ändern, ohne das weitere Parameter geschrieben oder verändert werden müssen.

Wenn der Anwender seine Änderungen speichern möchte, kann er das tun indem er mit dem Button *Write-XML* eine XML-Datei erzeugt. Zunächst muss ein Dateiname durch den Benutzer definiert. Es besteht die Möglichkeit die eingelesene XML-Datei zu überschreiben oder eine neue anzulegen. Hierzu wird die Funktion *getSaveFileName* der Klasse QFileDialog aufgerufen, die mit den voreingestellten Parametern ein Dialogfenster öffnet in der entweder eine vorhandene XML-Datei ausgewählt oder eine neue Datei erstellt wird. Hier nach wird das File geöffnet und ein StreamWriter eingerichtet, der das Erzeugen von strukturierten XML-Dateien ermöglicht. In einer *For-Schleife* werden die einzelnen Attribute der unterschiedlichen Kameras aus der Liste des Hauptprogramms in den entsprechenden Ebenen (Matrix, LensDisortion, usw.) der XML-Datei eingetragen. Abschließend wird die Datei geschlossen und der StreamWriter gelöscht.

2.2.4. Schnittstellen

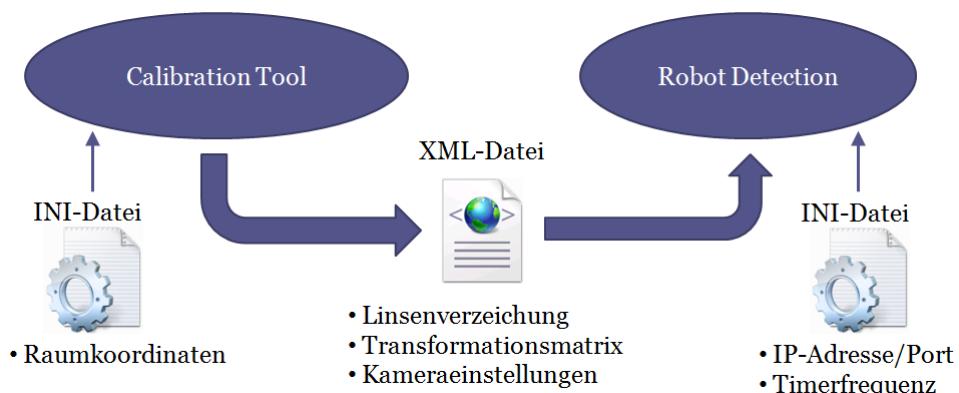


Abbildung 2.15.: Schnittstellen

Um die durch das Calibration Tool ermittelten Kalibrierungsparameter dem Robot Detection-Programm zur Verfügung zu stellen wird eine XML-Datei erzeugt. Diese beinhaltet die Linsenverzeichnung, die Transformationsmatrix und die jeweiligen Kameraeinstellungen. XML ermöglicht hierbei die hierarchisch strukturierte Form von Daten in Textdateien. Des Weiteren ist das Format menschenlesbar und von Hand editierbar.

Neben den übergebenen Werten in der XML-Datei existieren weitere Parameter die jeweils nur für die einzelnen Programme von Interesse sind. Hierzu gehören die ausgemessenen Raumkoordinaten für das Calibration Tool sowie IP-Adresse/Port und Timerfrequenz für das Robot Detection-Programm. Da es sich hierbei um beständige Werte handelt, die aber

dennoch einer in Zukunft möglichen Veränderung unterliegen, sind diese in eigene Initialisierungsdateien zusammengefasst. Dies bietet die Option vor Programmstart auch diese Parameter ggf. zu ändern und trägt gleichzeitig zur Übersichtlichkeit und Reduzierung der Daten in XML-Datei bei.

2.3. Ergebnis

Mit dem neuen Kalibrierungsprogramm ist es möglich die perspektivische Verzerrung der sechs Kameras in weniger als 10 Minuten zu kalibrieren. Die Kalibrierung ist ohne großes Messequipment möglich. Lediglich ein Stativ mit einem Kreis wird für die Vermessung benötigt. Das Stativ muss nacheinander auf die zwölf roten Kreise im Raum gestellt werden. Von den einzelnen Positionen werden dann mit dem *CalibrationTool* die Koordinaten bestimmt. Dabei wurde darauf geachtet, dass die Benutzeroberfläche benutzerfreundlich aufgebaut ist. Die Kamerabilder und die Schaltflächen zum Aufnehmen der Koordinaten der roten Kreise, ist genauso angeordnet wie sie auf dem Spielfeld wiederzufinden sind. Auch die Bedienelemente zum Starten der Kalibrierung sind auf das nötigste reduziert, um die Handhabung zu vereinfachen. Während der Raumkalibrierung erhält der Anwender ein direktes, optisches Feedback, ob die Messung der Raumkoordinaten erfolgreich ist. Da das Stativ im Raum platziert und abgestellt werden kann und anschließend mit dem Tool der Punkt eingelesen wird, ist die Vermessung des Raumes sogar durch eine einzelne Person möglich.

Mit den zusätzlich eingebauten Funktionen *Correct Lense Distortion* und *Camera Settings* können Kameras einzeln konfiguriert werden. Dank des Live Bildes von den Kameras auf der Benutzeroberfläche sind die Auswirkungen beim Ändern der Kameraparameter sofort sichtbar. Das ermöglicht es den Anwender sofort zu beurteilen ob die Änderungen zu einer Verbesserung des Bildes beitragen.

Die Funktion *Correct Lense Distortion* rechnet die Verzeichnung der Kameras aus dem Bild heraus. Das ermöglicht auch ein optimales Bild bei Kameras die eine starke Verzerrung aufweisen.

Die Parameterübergabe zwischen den beiden Programmen erfolgt im XML-Format. Das hat den Vorteil das die Kameradaten in einer lesbaren Form, strukturiert abgespeichert werden. Somit besteht die Möglichkeit Parameter von Hand zu editieren.

3. Hauptprogramm (RobotDetection)

In diesem Abschnitt soll das Hauptprogramm „RobotDetection“ vorgestellt werden. Dazu werden zunächst die Benutzeroberfläche und die Bedienung erläutert. Anschließend erfolgt eine detailliertere Vorstellung einzelner Funktionalitäten des Programmes.

3.1. Benutzeroberfläche und Bedienung

3.1.1. Aufbau der GUI

Die Benutzeroberfläche des Hauptprogramms ist in Abbildung 3.1 dargestellt. Sie besteht aus den drei im Bild rot markierten Bereichen:

1. Bedienelemente
2. Tabelle mit aktuellen Roboterpositionen
3. Visualisiertes Spielfeld mit Robotern
4. Statuszeile zur Anzeige von Kurzinfos

3.1.2. Bedienelemente

Die Bedienelemente sind in Abbildung 3.3(a) genauer dargestellt. Mit dem „*Start/Stop Detection*“-Button wird die zyklische Bildanalyse gestartet (genaueres zur Funktionsweise siehe Kap. 3.2.2). Bei gesetzter Checkbox „*Live View*“ werden statt des weißen Rasters die perspektivisch korrigiert zusammengesetzten Live-Kamerabilder als Spielfeldhintergrund angezeigt. Dies ist für Debuggingzwecke sehr hilfreich, so kann z.B. die Kreiserkennung oder die Helligkeit der Kamerabilder schnell überprüft werden (siehe Abb. 3.2).

Die Checkbox „*Expert Mode*“ blendet weitere Einstellmöglichkeiten ein, die in Abb. 3.3(b) dargestellt sind. Der „*Contour Threshold*“-Schieberegler dient zur Einstellung des Schwellwertes für das Binärbild, er kann im Bereich von 0 bis 255 eingestellt werden.

Die drei Slider auf der rechten Seite dienen zur Feineinstellung der Kreiserkennung. Es können der minimale und maximale Kreisradius, sowie der minimale Abstand zwischen zwei

3. Hauptprogramm (RobotDetection)

34

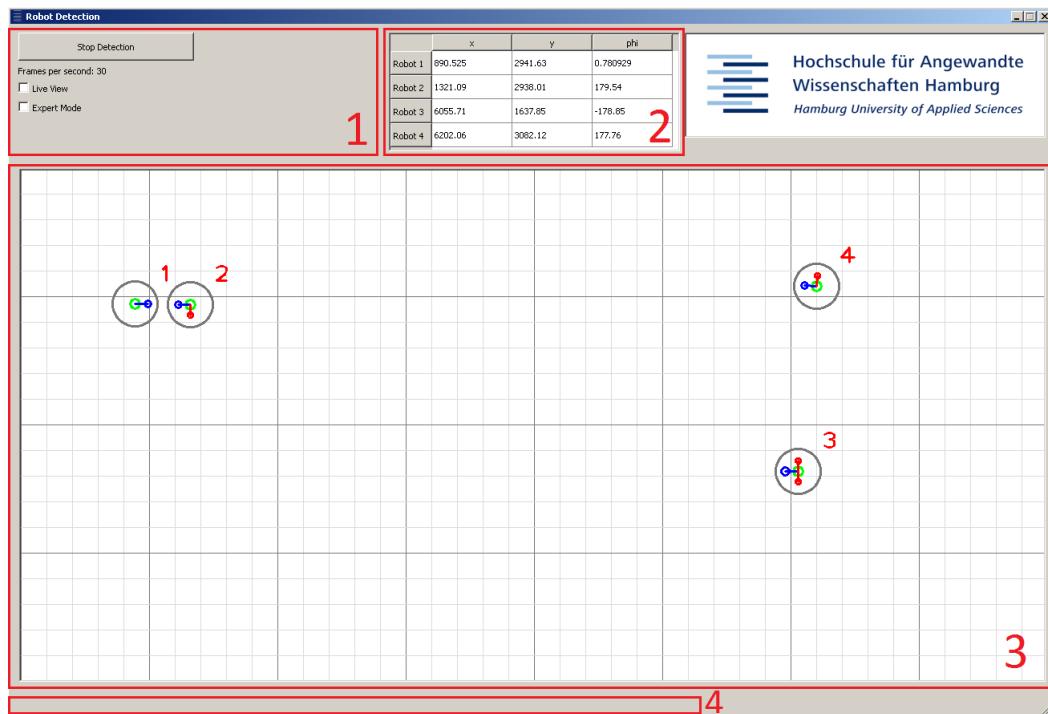


Abbildung 3.1.: Bereiche der Benutzeroberfläche des RobotDetection-Programms

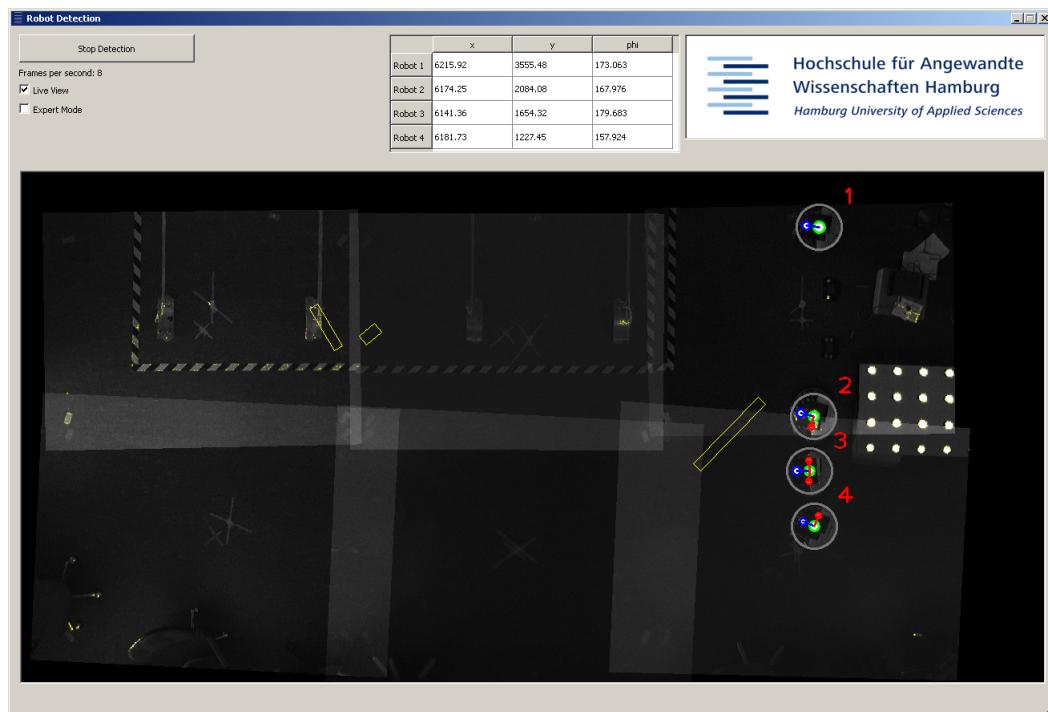


Abbildung 3.2.: Benutzeroberfläche mit Livebild (aufgehelle Kämeras)

Kreismittelpunkten eingestellt werden. Der Wertebereich dieser Slider ist jeweils 1 mm bis 100 mm.

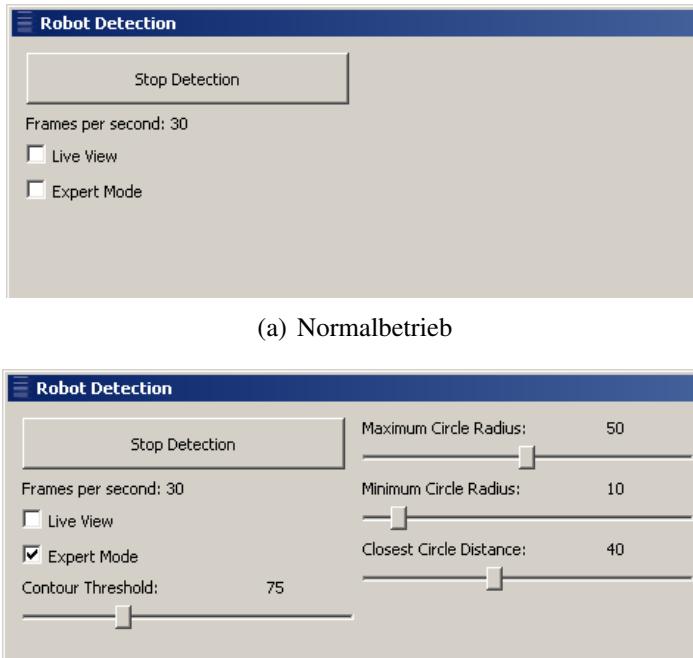


Abbildung 3.3.: Bedienelemente

3.1.3. Benutzerführung und Fehlermeldungen

Um eine möglichst einfache und intuitive Bedienung des Programms durch den Benutzer zu ermöglichen, ist Wert darauf gelegt worden, möglichst wenige Bedienelemente zu verwenden. Die XML-Datei mit den Kalibrierungsdaten wird beim Programmstart automatisch eingelesen und es erscheint ein Hinweis in der Statuszeile. Sollte die XML-Datei unter dem Standardnamen „*CalibrationData.xml*“ nicht gefunden werden, erscheint eine Fehlermeldung und der Benutzer wird aufgefordert eine andere XML-Datei mit Kalibrierungsdaten auszuwählen.

Beim Betätigen des „*Start/Stop Detection*“-Button wird ebenfalls ein Hinweis zum Verbindungsaufl- bzw. Abbau mit den Kameras in der Statuszeile angezeigt. Sollten weniger als sechs Kameras am PC angeschlossen sein, werden nur die erreichbaren Kameras benutzt und ein Hinweis wird im jeweiligen Spielfeld/Livebild-Bereich der GUI eingeblendet. Das Programm funktioniert also auch dann noch, falls eine oder mehrere Kameras fehlen.

3.1.4. Einstellungen in INI-Datei

Beim Programmstart wird außer der XML-Datei auch noch eine INI-Datei mit dem Namen „*settings.ini*“ eingelesen, in der Programmeinstellungen gespeichert werden. Durch die Verwendung einer zusätzlichen Datei sind diese Programmeinstellungen von den Kalibrierungsdaten getrennt. Außerdem hat die INI-Datei den Vorteil, dass sie leicht von Hand editierbar ist und in *Qt* mit Hilfe der Klasse *QSettings* leicht zu verarbeiten ist.

Im Einzelnen werden folgende Einstellungen gespeichert:

- die Werte der o.g. Schieberegler für die Kreiserkennung
- die IP-Adresse sowie der Port an den die UDP-Pakete verschickt werden
- die Wartezeit des Timers der die Hauptschleife aufruft

Wenn beim Programmstart keine INI-Datei mit dem Namen „*settings.ini*“ gefunden wird, werden Standardwerte für die Einstellungen verwendet. Beim Beenden des Programms werden dann jeweils die aktuellen Werte in die Datei geschrieben oder es wird eine neue Datei erstellt. Mit den voreingestellten Standardwerten hat die Datei „*settings.ini*“ den in Listing 3.1 gezeigten Inhalt.

Listing 3.1: Inhalt der Datei *settings.ini*

```
[RobotDetectionSettings]
SendToIP=192.168.0.255
SendToPort=25000
TimerMilSecs=20
MaxRadius=50
MinRadius=10
CloseDist=40
Threshold=75
```

3.2. Funktionsumfang

3.2.1. Programmaufbau

Das Hauptprogramm besteht im Wesentlichen aus den folgenden Klassen:

- RobotDetectionMainWindow: Die Hauptklasse des Programms.
- ImgTask: Eine multithreadingfähige Klasse zur parallelen Bearbeitung der Kameras Bilder. Diese Klasse erbt von QRunnable und implementiert die run() Methode.
- MyUDP: Eine Klasse die Methoden zum Senden der Roboterpositionen per UDP-Socket bietet.

3.2.2. Programmablauf

Die Aufgabe des Hauptprogrammes ist es die aktuellen Roboterpositionen aus den Bildern der Deckenkameras zu ermitteln und diese Positionen mit größtmöglicher Wiederholrate den Robotinos zur Verfügung zu stellen. Der gesamte Programmablauf ist schematisch in Abbildung 3.4 dargestellt. Der im Bild grau hinterlegte Bereich zeigt den Ablauf der Funktion mainloop(). Diese Hauptfunktion des Programmes wird von einem Timer zyklisch als Slot aufgerufen. In den folgenden Abschnitten werden einzelne wichtige Programmschritte genauer erläutert.

3.2.3. Initialisierung

Beim Programmstart werden zunächst die Einstellungen aus der Datei *settings.ini* und die Kalibrierungsdaten aus der Datei *CalibrationData.xml* eingelesen. Danach wird der UDP-Client vom Typ MyUDP (siehe Kap. 3.2.6) mit den eingelesenen IP-Einstellungen instanziert. Anschließend werden die Timer für die Hauptschleife und für die Anzeige der Bildwiederholrate (Frames-Per-Second, FPS) initialisiert sowie die timeout()-Signale mit den jeweiligen Slots verbunden.

Listing 3.2: Initialisierung der Timer

```
// init timers for mainloop and frames-per-second counting
timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(mainloop()));
timerFPS = new QTimer(this);
connect(timerFPS, SIGNAL(timeout()), this, SLOT(fpsCounter()));
fpsCount = 0;
```

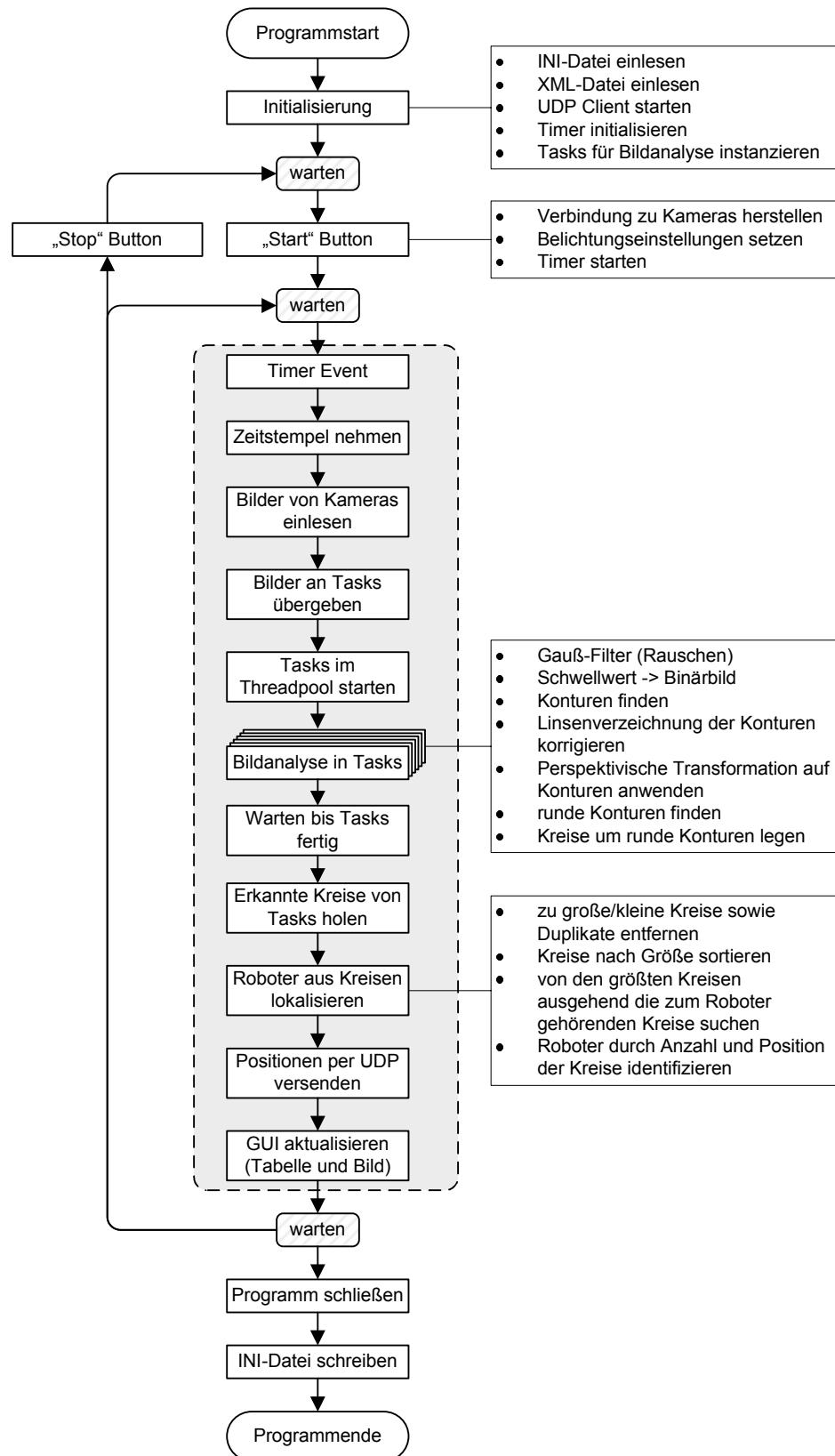


Abbildung 3.4.: Ablaufschema des Hauptprogrammes

Zuletzt werden ein QThreadpool und pro Kamera jeweils ein IImgTask (siehe Kap. 3.2.4) angelegt. Der Threadpool übernimmt später die Organisation der Threads gegenüber dem Betriebssystem und verwaltet deren Bearbeitung dynamisch, d.h. es können abhängig von der Auslastung des Betriebssystems und den verfügbaren Prozessorkernen mal mehr und mal weniger Threads gleichzeitig laufen. Mit setAutoDelete(false) wird erreicht, dass die Tasks nach der Bearbeitung ihrer run()-Methode nicht vom Threadpool terminiert werden. Dadurch kann die Rechenzeit für die ständige Neuinitialisierung eingespart werden und die Kalibrierungsdaten (cameraMatrix, distCoeffs und perspTransfMatrix) bleiben für die gesamte Laufzeit erhalten.

Listing 3.3: Initialisierung des Threadpools und der Tasks

```
// open threadpool for image analysis
threadPool = new QThreadPool();
threadPool->setMaxThreadCount(NR_OF_CAMS);
for (int i = 0; i < NR_OF_CAMS; i++)
{
    tasks[i] = new ImgTask();
    tasks[i]->setAutoDelete(false);
    tasks[i]->setCameraMatrix(cameraMatrix.at(i));
    tasks[i]->setDistCoeffs(distCoeffs.at(i));
    tasks[i]->setPerspTransfMatrix(perspTransfMatrix.at(i));
}
```

3.2.4. Bildverarbeitung und Kreiserkennung

Sobald das Programm mit dem *Start Detection*-Button gestartet wird, werden die Kameraverbindungen aufgebaut und die Timer gestartet. Beim Aufruf der mainloop() Funktion wird dann zunächst die Systemzeit als Zeitstempel abgerufen und direkt danach werden die Kamerabilder, wie in Listing 3.4 gezeigt, abgerufen. Um die Zeitdifferenz zwischen den Bildern möglichst gering zu halten, werden mittels videoCapture[i].grab() zunächst alle Kameras angewiesen ein Bild zu aufzunehmen und anschließend werden in einer neuen Schleife mittels videoCapture[i].retrieve(cameraImages[i], 0) die Bilder ausgelesen. Dadurch hat die Übertragungszeit der Bilder keine Auswirkung auf die Zeitdifferenz zwischen den Bildern.

Listing 3.4: Einlesen der Kamerabilder und Übergabe an Tasks

```
for (int i = 0; i < NR_OF_CAMS; i++)
    if (videoCapture[i]. isOpened())
        videoCapture[i]. grab();
for (int i = 0; i < NR_OF_CAMS; i++)
{
    if (videoCapture[i]. isOpened())
        videoCapture[i]. retrieve(cameraImages[i], 0);
[...]
```

```

    tasks[ i ]->setImage( cameraImages[ i ] );
    tasks[ i ]->setThresh( ui->sliderThreshold->value() );
    tasks[ i ]->setDebugMode( ui->checkBoxLiveView->isChecked() );
    threadPool->start( tasks[ i ] );
}
threadPool->waitForDone();

```

Um die Bilder nun zu analysieren werden sie jeweils einem Task übergeben, der dann im Threadpool gestartet wird. Mit `threadPool->waitForDone()` wird dann auf das Ende aller Tasks gewartet.

In der Funktion `ImgTask::run()`, auf deren Abbildung an dieser Stelle aus Platzgründen verzichtet wird, erfolgt nun die Verarbeitung der Einzelbilder nach folgenden Schritten:

1. Bild in Grauwertbild wandeln
2. Gauß-Filter der Größe 3x3 auf das Bild anwenden um den Einfluss des Bildrauschen zu minimieren
3. Grauwertbild in Binärbild wandeln mittels `cv::threshold()`
4. Konturen im Bild finden mit `cv::findContours()`, dies ist erheblich schneller als der *Canny*-Algorithmus und liefert dank der guten Ausgangsbilder genau so gute Ergebnisse
5. Konturenpunkte in Datentyp `cv::Point2f` wandeln für Subpixel-Genauigkeit
6. Linsenverzeichnung der Konturen korrigieren mit `cv::undistortPoints()`
7. Konturen mittels perspektivischer Transformation auf das Spielfeldkoordinatensystem umrechnen mit `cv::perspectiveTransform()`
8. jeweils das kleinstmögliche Rechteck um die Kontur legen mit `cv::minAreaRect()` und dessen Seitenverhältnis berechnen
9. wenn das Seitenverhältnis zwischen 0,7 und 1,3 liegt, Kontur als Kreisförmig interpretieren, anderenfalls ignorieren
10. mittels `cv::minEnclosingCircle()` die kleinstmöglichen Kreise um die verbleibenden Konturen legen, diese sind dann die erkannten Kreise

Falls der *Live View* Modus aktiviert ist, werden zusätzlich noch folgende Schritte ausgeführt:

1. gefundene Konturen in das Originalbild einzeichnen
2. Linsenverzeichnung des gesamten Bildes korrigieren mit `cv::undistort()`
3. gesamtes Bild perspektivisch transformieren mit `cv::warpPerspective()`

4. die um die Konturen gelegten Rechtecke ins Bild zeichnen

5. die gefundenen Kreise ins Bild zeichnen

Sowohl die Korrektur der Linsenverzeichnung als auch die perspektivische Transformation sind sehr rechenaufwändig, sodass im *Live View* Modus die Bildwiederholrate recht stark einbricht. Dieser Effekt ist direkt von der Größe des Zielbildes abhängig auf die das Bild transformiert wird. Wenn also ein Kamerabild mit einer Größe von 640x480 Pixeln auf die Spielfeldgröße von 8000x4000 mm (bzw. 8000x4000 Pixel) transformiert wird, benötigt dies extrem viel Rechenzeit (mehrere Sekunden pro Frame). Außerdem ist es dann nötig das Ergebnisbild mit `cv::resize()` wieder auf die Größe der GUI zu reduzieren, was durch die Bildgröße von $3,2 \cdot 10^7$ Pixel ebenfalls sehr rechenintensiv ist.

Dieser Aufwand kann durch die Berücksichtigung der Skalierung für das GUI-Fenster, bereits vor der Transformation des Bildes, eingespart werden. Dazu wird die Transformationsmatrix **T** mit einer Skalierungsmatrix **S** multipliziert:

$$\mathbf{T}_{GUI} = \underbrace{\begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & 1 \end{bmatrix}}_S \cdot \underbrace{\begin{bmatrix} t_1 & t_2 & t_3 \\ t_4 & t_5 & t_6 \\ t_7 & t_8 & 1 \end{bmatrix}}_T = \begin{bmatrix} a \cdot t_1 & a \cdot t_2 & a \cdot t_3 \\ a \cdot t_4 & a \cdot t_5 & a \cdot t_6 \\ t_7 & t_8 & 1 \end{bmatrix} \quad (3.1)$$

mit Skalierungsfaktor $a = \frac{\text{GUI-Breite}}{\text{Spielfeldbreite}} = \frac{1280 \text{ Pixel}}{8000 \text{ Pixel}} = 0,16$

Diese Skalierung wurde in der Methode `ImgTask::setPerspTransfMatrix()` der Klasse `ImgTask` umgesetzt, wie in Listing 3.5 gezeigt. Die Methode wird bei der Initialisierung einmalig aufgerufen und kann somit den Rechenaufwand deutlich reduzieren.

Listing 3.5: Skalierung der Transformationsmatrix

```
void ImgTask :: setPerspTransfMatrix (cv::Mat perspTransfMatrix)
{
    this ->perspTransfMatrix = perspTransfMatrix;

    // calculate GUI Transformation Matrix by scaling down
    perspTransfMatrix
    Mat scaleMatrix = Mat::zeros(3,3, CV_64F);
    scaleMatrix .at<double>(0,0) = 1.0 / GUI_SCALING;
    scaleMatrix .at<double>(1,1) = 1.0 / GUI_SCALING;
    scaleMatrix .at<double>(2,2) = 1.0;
    guiTransfMatrix = scaleMatrix * perspTransfMatrix;
}
```

3.2.5. Roboteridentifikation

Sobald die Bearbeitung der Tasks abgeschlossen ist, wird die `mainloop()` Funktion fortgesetzt. Zunächst werden die in den Bildern gefundenen Kreise eingesammelt, wie in Listing 3.6 gezeigt.

Listing 3.6: Einsammeln der gefundenen Kreise

```
cv::vector<Point3f> circles;
for (int i = 0; i < NR_OF_CAMS; i++)
{
    cv::vector<Point3f> circlesFromTask = tasks[i]->getircles();
    circles.insert( circles.end(), circlesFromTask.begin(),
                    circlesFromTask.end());
}
```

Der Vektor `circles` wird anschließend der Funktion `locateRobotsFromCircles()` übergeben, deren Aufgabe es ist die Roboter anhand der Kreise zu identifizieren und ihre Positionen zu bestimmen. Die Anordnung und die Größen der Reflektormarken auf den Robotern sind in Abbildung 3.5 dargestellt.

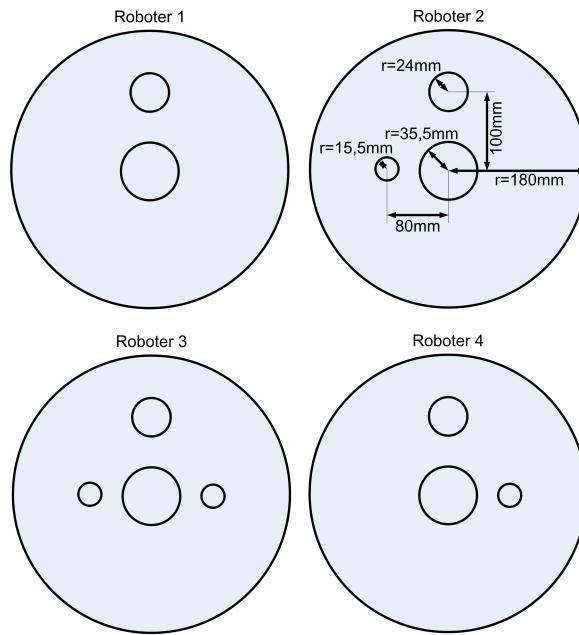


Abbildung 3.5.: Anordnung der Reflektormarken auf den Robotern

Da sich die Kamerabilder überlappen, ist es möglich, dass Kreise in den Randbereichen mehrfach gefunden werden. Daher werden zunächst alle Kreise, deren Abstand zu einem anderen Kreis kleiner als der im Expertenmodus eingestellte Mindestwert ist, aus dem Vektor entfernt. Außerdem werden Kreise gelöscht, die deutlich zu groß oder zu klein sind.

Die Grenzen dafür können ebenfalls im Expertenmodus eingestellt werden. Die Anzahl der nach diesen Kriterien gelöschten Kreise wird im Expertenmodus in der Statuszeile angezeigt. Dies ist hilfreich um die Grenzen richtig einzustellen.

Der Vektor `circles` enthält nun nur noch plausible Kreise, aus denen die Roboter zu identifizieren sind. Dazu wird zunächst der Vektor absteigend, nach der Größe der Kreise sortiert. Die Sortierung erspart im weiteren Verlauf viele Vergleichsoperationen. Um die Kreise nach ihrem Radius zu sortieren, wird die Funktion `std::sort()` mit einem selbstdefinierten Vergleichsoperator benutzt (Listing 3.7 und 3.8).

Listing 3.7: Sortierung des Vektors

```
// sort remaining circles by size in descending order
std::sort(circles.begin(), circles.end(), myComparator);
```

Listing 3.8: Vergleichsoperator zur Sortierung des Vektors

```
struct myComparatorClass {
    bool operator() (cv::Point3f a, cv::Point3f b){ return (a.z > b.z);}
} myComparator;
```

Nach der Sortierung beginnt die eigentliche Identifikation der Roboter. Die ersten vier Elemente von `circles` sind die größten plausiblen Kreise, also sehr wahrscheinlich die Mittelpunktmarkierungen der Roboter. Von diesen Kreisen aus werden nun die in der Nähe liegenden Kreise (d.h. ihr Abstand ist kleiner als der Roboterradius) in `circles` gesucht und in einem neuen Vektor `circlesInRange` gespeichert. Da die Kreise in `circles` bereits sortiert sind, sind sie auch in `circlesInRange` der Größe nach sortiert. Das bedeutet falls Kreise innerhalb des Roboterradius gefunden wurden, ist der erste bzw. größte von ihnen auf jeden Fall der mittelgroße Richtungskreis des Roboters. Die Roboterposition steht durch den Mittelpunktskreis fest und der Winkel zur x-Achse kann nun berechnet werden. Dazu dient die Funktion `horizontalAngle(centerPoint, orientationPoint)`.

Jetzt gilt es lediglich noch festzustellen um welchen Roboter es sich handelt. Dazu wird die Anzahl der Kreise in `circlesInRange` herangezogen: Falls es keine weiteren Kreise gibt, handelt es sich um Roboter 1. Falls es zwei weitere (durch die Sortierung also kleinere) Kreise gibt, handelt es sich um Roboter 3. Falls es einen weiteren Kreis gibt, muss der Roboter anhand der Winkeldifferenz zwischen der Pose des Roboters φ und dem Winkel des Identifikationskreises gegen die x-Achse bestimmt werden. Diese Funktionalität ist in Listing 3.9 dargestellt (gekürzt um das Zeichnen der Roboter in das GUI-Bild).

Listing 3.9: Auszug aus `locateRobotsFromCircles()`

```
for(unsigned int i = 0; i < circles.size(); i++)
{
    // there can't be more than 4 robots
    if (i >= MAX_NR_OF_ROBOTS)
```

```

break;
// find circles within range of the largest center circles
cv::vector<cv::Point3f> circlesInRange;
for (unsigned int j = i+1; j < circles.size(); j++)
    if( distanceBetweenPoints( circles.at(i), circles.at(j) ) <
        (ROBOT_RADIUS - 50) )
    {
        // move circle to vector of circles that belong to this
        very robot
        circlesInRange.push_back( circles.at(j) );
        circles.erase( circles.begin() + j );
        j--; // set iterator back one item
    }
if( circlesInRange.size() > 0 ) // it is a robot
{
    // current element in vector "circles" is the center circle
    Point2f centerPoint = Point2f( circles.at(i).x, circles.at(i)
        .y );
    double centerRadius = circles.at(i).z;

    // first/largest circle within range of center circle is the
    orientation circle
    Point2f orientationPoint = Point2f( circlesInRange.at(0).x,
        circlesInRange.at(0).y );
    double orientationRadius = circlesInRange.at(0).z;
    [...]
    // determine robot Position and Angle
    double orientationAngle = horizontalAngle(centerPoint,
        orientationPoint);
    Point3f robPosition = Point3f(centerPoint.x, centerPoint.y,
        orientationAngle);

    // erase orientation circle from vector, so all remaining
    circlesInRange are small identification circles
    circlesInRange.erase(circlesInRange.begin());
    // identify robots and write locations
    if (circlesInRange.size() == 2) // Robot 3 identified
    {
        robotLocations.at(2) = robPosition;
        [...]
    }
    else if (circlesInRange.size() == 1)
    {
        // analyze angles to distinguish Robot 2 and Robot 4
        Point2f identPoint = Point2f(circlesInRange.at(0).x,
            circlesInRange.at(0).y );
        [...]
        double angleDifference = orientationAngle -
            horizontalAngle(centerPoint, identPoint);
    }
}

```

```

    if (((angleDifference < 0) && (angleDifference > -180)) || (
        angleDifference > 180)) // Robot 2 identified
        robotLocations.at(1) = robPosition;
    else // Robot 4 identified
        robotLocations.at(3) = robPosition;
}
else if (circlesInRange.size() == 0) // Robot 1 identified
    robotLocations.at(0) = robPosition;
[...]
}

```

Ein großer Vorteil dieses Verfahrens zur Roboteridentifikation liegt in der Robustheit gegenüber Größenschwankungen der Kreise. Die Kreise in den Kamerabildern können durch die sich ändernden Lichtverhältnisse im Labor und auch durch die nicht fehlerfreie Kalibrierung der Kameras unterschiedlich groß sein (z.B. können sie bei zu viel Lichteinfall überstrahlen, also im Kamerabild größer erscheinen). Eine feste Kategorisierung der Kreise nach ihrer Größe (in große, mittlere und kleine Kreise) kann damit nicht umgehen, insbesondere dann nicht, wenn die Abweichungen an verschiedenen Positionen unterschiedlich groß sind (z.B. mehr Überstrahlen auf der helleren Fensterseite).

Durch die hier verwendete Sortierung der Kreise, ohne feste Kategorisierung, hat nur ihre relative Größe untereinander einen Einfluss auf die Interpretation als Mittelpunkt, Orientierungspunkt oder Identifikationspunkt. Ein weiterer Vorteil ist die einfachere und robustere Einstellung der Kreiserkennung. Es müssen keine Schwellen zwischen den Kreisgrößen, sondern lediglich Grenzen für die maximale und minimale Kreisgröße festgelegt werden. Diese Einstellungen können aber auch bei einer weiten Auslegung noch sicher funktionieren.

3.2.6. Datenversand per UDP

Nachdem die Roboterpositionen aus den Bildern ermittelt worden sind, werden sie als double-Werte per UDP-Protokoll an die Robotinos verschickt. Hierzu wird die Funktion `MyUDP::sendUdpData()` der Klasse `MyUDP` benutzt. Ursprünglich wurde in diesem Projekt eine Reduzierung der gesendeten Daten auf ein Minimum angestrebt, aber um dem Gewerk 3 einen jederzeitigen Wechsel zwischen dem alten Matlab-System und der neuen Software zu ermöglichen, wurde das alte Format bisher beibehalten.

Beim alten Datenformat werden unnötigerweise 8 statt 4 Roboterpositionen versendet, gefolgt vom Zeitstempel sowie 29 weiteren, nicht benötigten Werten vom Typ double. Im Quelltext der Datei `myudp.cpp` müssen für die Reduzierung der Datenmenge von 448 Byte/Paket auf die mindestens benötigten 120 Byte/Paket lediglich einige bereits kenntlich gemachte Zeilen auskommentiert werden. Dies würde auch das bereits stark ausgelastete WLAN im Labor entlasten.

3.3. Ergebnisse

3.3.1. Verbesserungen gegenüber der Matlab-Lösung

Das *RobotDetection* Programm weist gegenüber dem zuvor eingesetzten Matlab-System einige entscheidende Verbesserungen auf. So ist nun die Anzeige des transformierten, zusammengesetzten Livebildes jederzeit möglich, was die Einstellung und die Fehlersuche deutlich erleichtert. Außerdem verfügt das Programm über eine robuste Strategie zur Kreiserkennung und Roboteridentifikation, die ohne eine Kategorisierung der Kreisgrößen auskommt und vom Benutzer über wenige Parameter sehr genau einstellbar ist. Die neue Kreiserkennung ist auch unempfindlich gegenüber Beleuchtungsänderungen und den damit verbundenen Abweichungen der Kreisgrößen.

Die Benutzerfreundlichkeit ist gegenüber der Matlab-Lösung ebenfalls deutlich verbessert worden, da *RobotDetection* als eigenständiges Programm läuft und mit wenigen, klar strukturierten Bedienelementen auskommt. Die Benutzeroberfläche beschränkt sich auf ein einziges Fenster und die ermittelten Roboterpositionen werden sowohl als Zahlenwerte als auch optisch in einem visualisierten Spielfeld ausgegeben.

3.3.2. Performance

Die Performance des *RobotDetection* Programmes ist im Vergleich zur alten Matlab-Lösung deutlich gesteigert worden. Ein Vergleich der Bildwiederholraten ist in Tabelle 3.1 dargestellt. Die finale Programmversion mit Multithreading und den zuvor erläuterten effizienten Auswertungsalgorithmen erreicht auf dem Labor-PC eine Bildwiederholrate von 30 FPS. Dies entspricht der maximal erreichbaren Bildrate der Deckenkameras, daher sind hier keine weiteren Steigerungen aus der Software heraus möglich. Sobald der *Live View* Modus aktiviert wird, sinkt die Wiederholrate auf 8 FPS.

Mit dem Matlab-System wurden im Vergleich dazu nur 10 FPS im Normalbetrieb erreicht, ein Zusammensetzen des transformierten Bildes (*Live View*) war damit gar nicht möglich. Auf einem aktuellen PC mit einem Intel Core-i7-4770K Prozessor (4 Kerne / 8 Threads) konnte das Livebild mit bis zu 17 FPS berechnet werden. Im *Live View* Modus ist also die Rechenleistung des Prozessors der begrenzende Faktor.

Tabelle 3.1.: Performance-Vergleich

	FPS (normal)	FPS (Live View)
Matlab Lösung (Labor-PC)	10	nicht möglich
Single Thread (Labor-PC)	19	5
Multi-Threading (Labor-PC)	30	8
Multi-Threading (schnellerer PC)	30	17

4. Messung der Positionsgenauigkeit

Um eine Aussage über die Qualität der Positionsmessung der Roboterposen treffen zu können werden die ermittelten Werte nachgemessen. Hierzu wird ein Maßband verwendet. Die Messung wird zwischen einem Referenzpunkt auf dem Boden und dem Mittelpunkt des Roboters auf Höhe der angebrachten Markierungen durchgeführt.

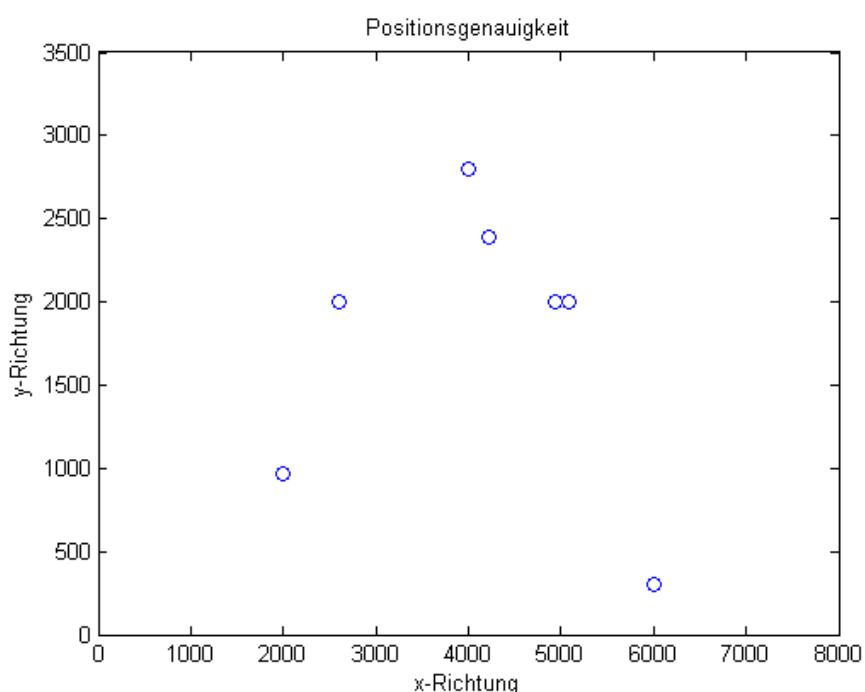


Abbildung 4.1.: Übersicht der gemessenen Posen

Der entstehende Höhenunterschied wird mit Hilfe des Satzes des Pythagoras berücksichtigt. Jedoch ist eine unbekannte Messunsicherheit nicht auszuschließen. Des Weiteren führt der Höhenunterschied zu Einschränkungen der Posenauswahl. Abbildung 4.1 zeigt die gemessenen Positionen.

Tabelle 4.1.: Positionsgenauigkeit

x-Richtung in mm Robot Detection	y-Richtung in mm Robot Detection	x-Richtung in mm von Hand gemessen	y-Richtung in mm von Hand gemessen	Abstand in mm \bar{p}_{dist}
5089,2	2000,1	5086,7	1999,0	2,76
2600,2	2000,3	2587,9	2000,0	12,33
4941,0	2000,0	4931,1	1994,0	11,60
6001,8	300,9	5994,0	301,0	8,08
4001,0	2804,1	3994,5	2797,9	8,98
2000,7	964,9	1983,9	970,4	17,65
4228,7	2391,0	4219,9	2382,8	12,08

Die Tabelle 4.1 listet die gemessenen Posen des Robot Detection Tools, die von Hand gemessenen Positionen und der daraus resultierende euklidische Abstand zwischen den jeweiligen Positionen auf. Daraus folgt eine mittlere absolute Abweichung $\bar{p}_{dist} = 10,5\text{mm}$ sowie eine Standardabweichung von $SD = 4,6\text{mm}$.

5. Fazit und Ausblick

Die zu Beginn des Verbundprojektes festgelegten Projektziele für Gewerk 5 konnten vollständig erreicht werden. Dazu war zunächst eine umfangreiche Einarbeitung sowohl in die Entwicklungsumgebung (*Qt* und *OpenCV*), als auch in die Bildverarbeitung im allgemeinen nötig, da es hier keinerlei Vorkenntnisse gab.

Mit dem *Calibration Tool* wurde ein benutzerfreundliches, effektives und vielseitig Einsetzbares Werkzeug zur Kamerakalibrierung geschaffen. Die vom Benutzer eingestellten Parameter sowie die Korrektur der Linsenverzeichnung werden sofort auf die Kamerabilder angewandt. Es ist damit außerdem möglich die perspektivischen Transformationsmatrizen der sechs Deckenkameras in weniger als 10 Minuten zu ermitteln. Dazu wurde der Laborraum während der Projektphase vermessen und mit Markierungspunkten ausgestattet.

Das Hauptprogramm *RobotDetection* ist in der Lage die Roboterpositionen mit 30 Frames pro Sekunde aus den Kamerabildern zu ermitteln und zu versenden. Gegenüber der alten Matlab-Lösung konnte die Geschwindigkeit also verdreifacht werden. Außerdem ist jederzeit die Anzeige des aus den Kamerabildern zusammengesetzten Livebildes möglich, was einen hohen Komfort für den Benutzer mit sich bringt. Für die Erkennung der Robotinos anhand ihrer Reflektormarken ist außerdem ein Verfahren entwickelt worden, das sehr robust gegenüber Änderungen der Lichtverhältnisse im Laborraum ist.

Die Parameterübergabe zwischen den beiden Programmen erfolgt im weit verbreiteten XML-Format. Dies bietet den Vorteil das die Kameradaten strukturiert abgespeichert werden und auch verschiedene Parametersätze leicht gewechselt werden können.

Durch den konsequenten Einsatz von Open-Source-Software wird auf dem Labor-PC zukünftig keine kostenpflichtige Matlab-Lizenz und somit auch keine zwingende Anbindung an das Hochschulnetzwerk mehr benötigt.

Ausblick für zukünftige Erweiterungen

Während des Projektes sind einige mögliche Erweiterungen bzw. Ideen für zukünftige Gruppen aufgekommen, die nicht mehr realisiert wurden. So könnte z.B. eine automatische Raumkalibrierung angedacht werden. Dazu könnten die vermessenen Raumpunkte mit den Robotinos über Induktionssensoren und Magnetmarken auf dem Laborboden automatisch angefahren werden. Eine weitere Idee wäre die Auswertung der Residuen der ermittelten Kameramatrix. Dies könnte benutzt werden um dem Benutzer eine Rückmeldung über die Qualität der gerade durchgeföhrten Kalibrierung zu geben.

Während des Projektes wurde versucht die Genauigkeit der Positionbestimmung durch Vergleichsmessungen von Hand zu überprüfen, dies ist jedoch ohne ein genaueres Referenzmesssystem nur unzureichend möglich. Eine Anschaffung eines Referenzmesssystems könnte also ebenfalls angedacht werden.

A. Anhang

A.1. Linsenverzeichnung

A.1.1. Verzeichnung der Kamera 0

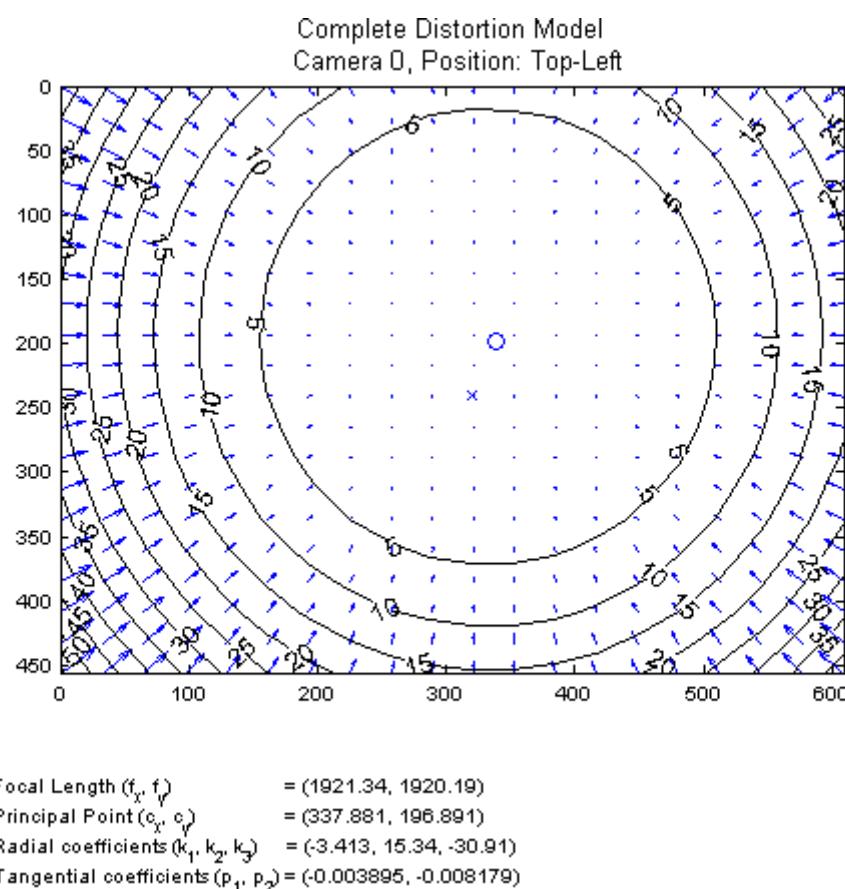


Abbildung A.1.: Vollständiges Verzeichnungsmodell der Kamera 0

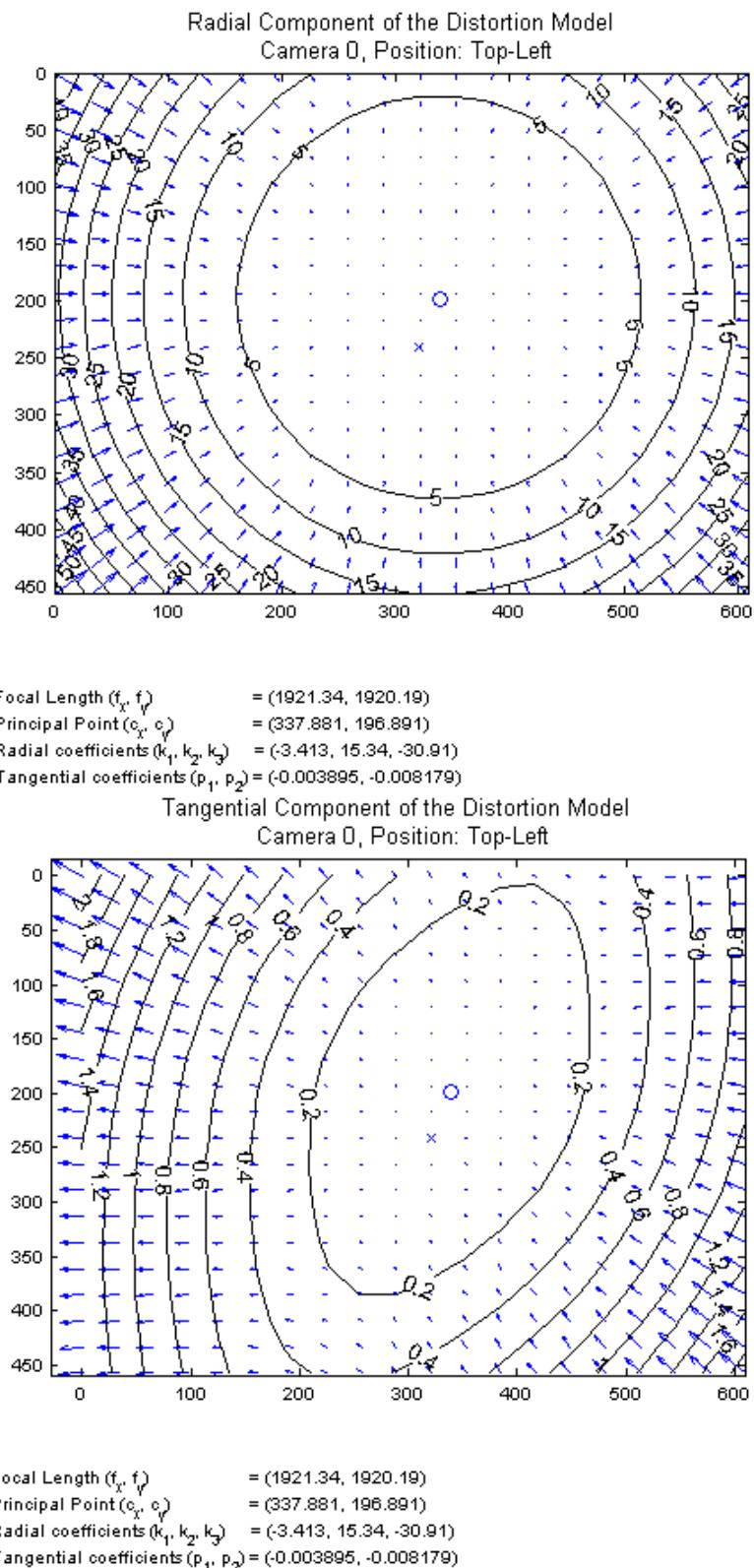


Abbildung A.2.: Radiales und tangentiales Verzeichnungsmodell der Kamera 0

A.1.2. Verzeichnung der Kamera 1

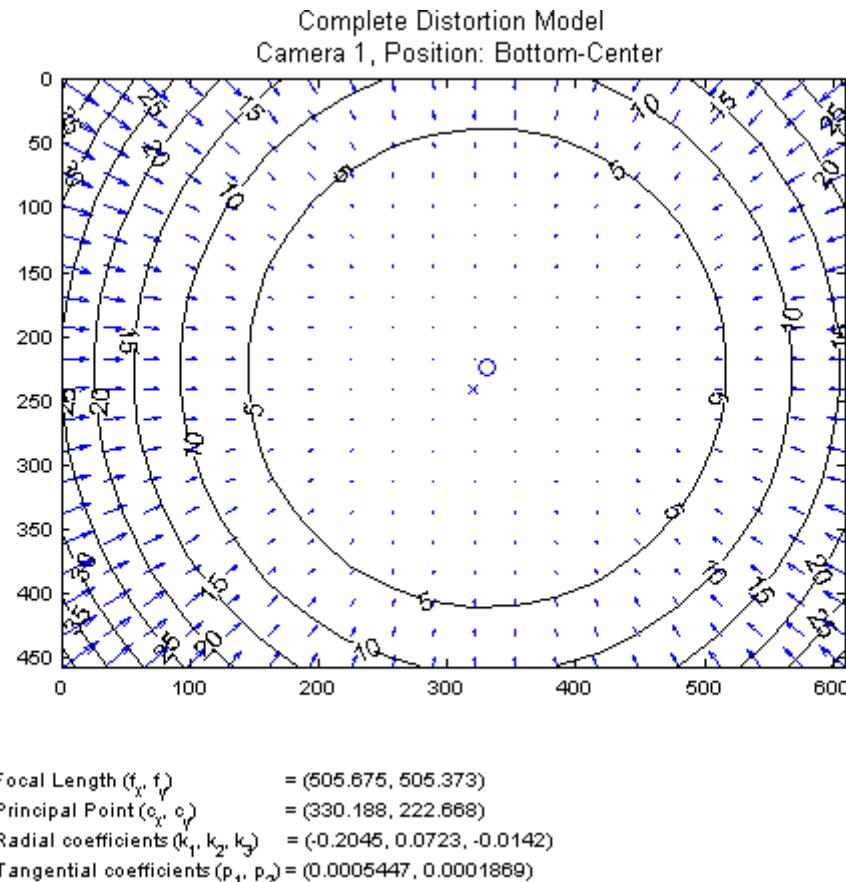


Abbildung A.3.: Vollständiges Verzeichnungsmodell der Kamera 1

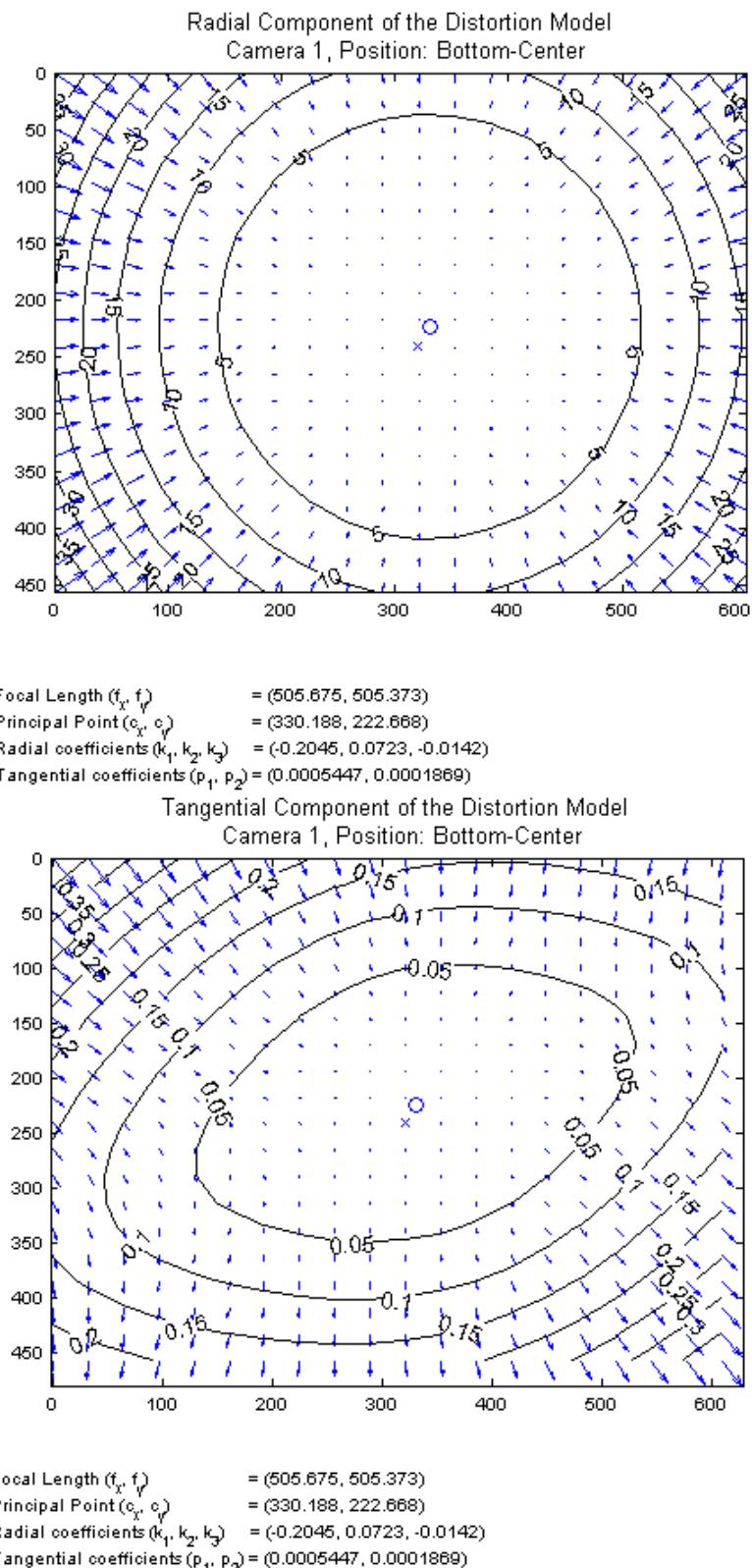


Abbildung A.4.: Radiales und tangentiales Verzeichnungsmodell der Kamera 1

A.1.3. Verzeichnung der Kamera 2

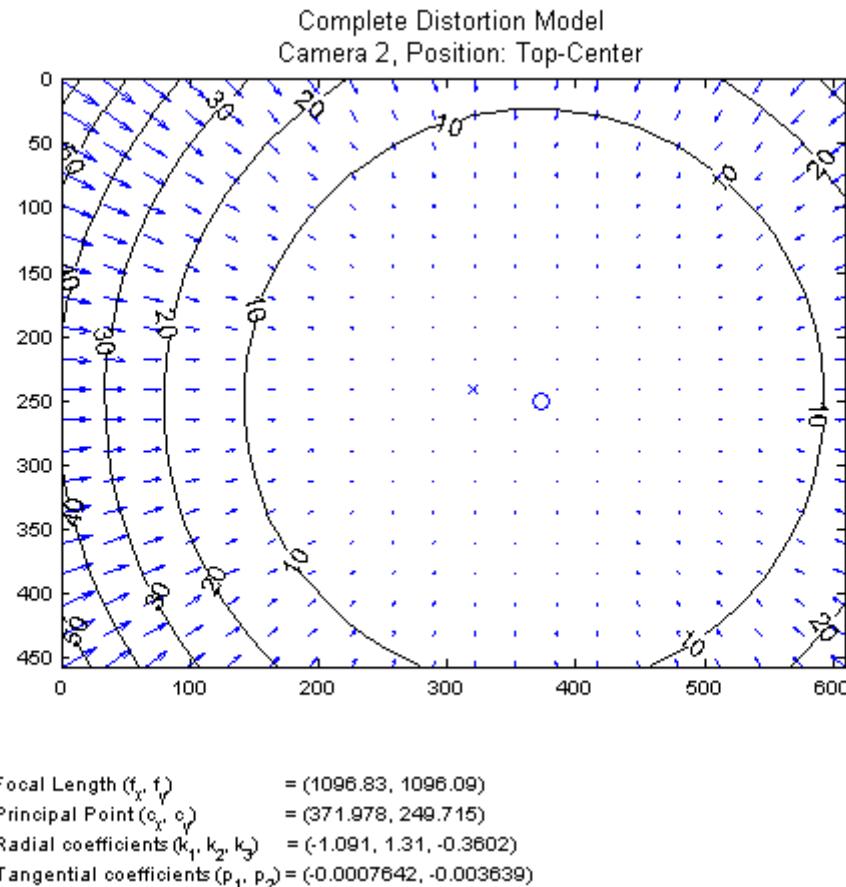


Abbildung A.5.: Vollständiges Verzeichnungsmodell der Kamera 2

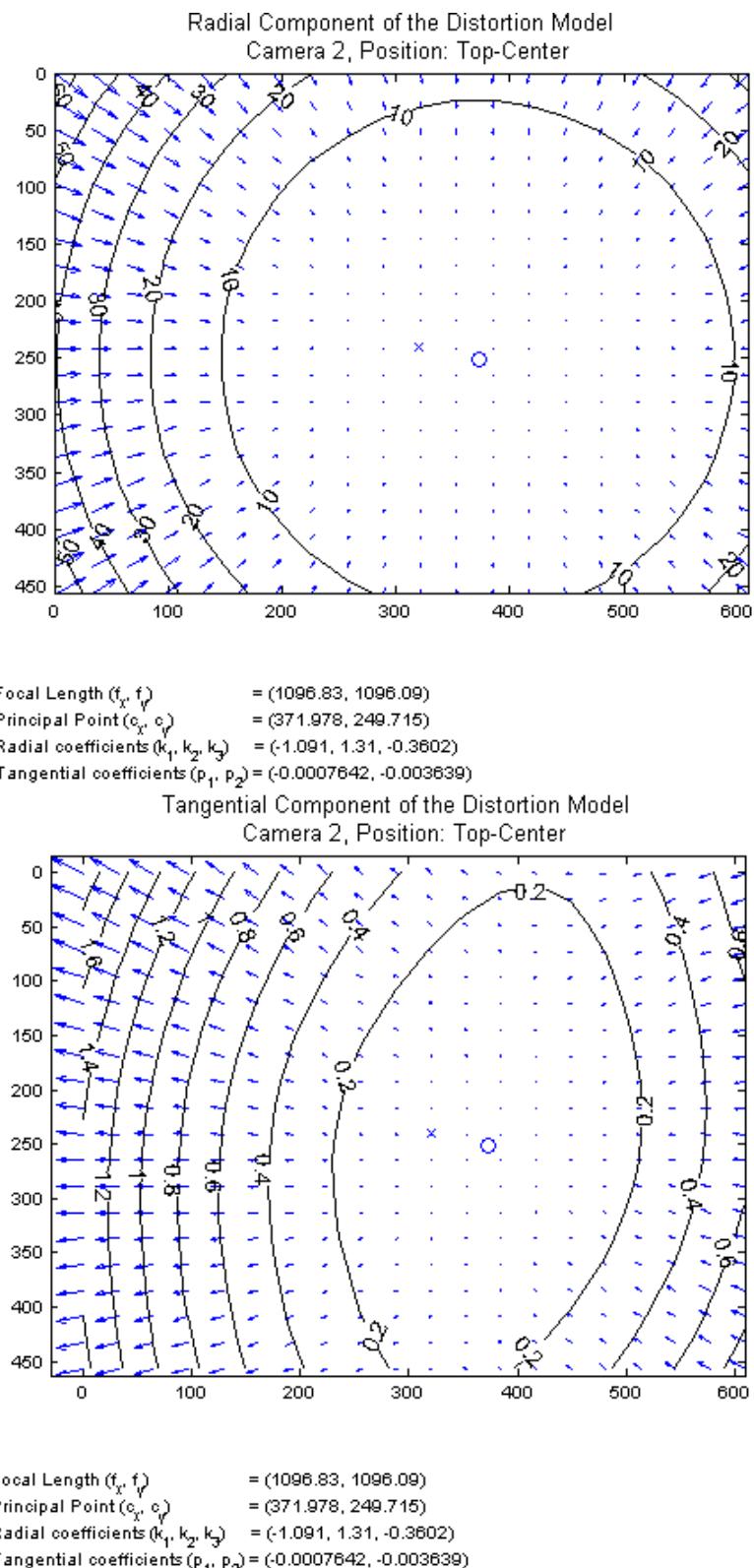


Abbildung A.6.: Radiales und tangentiales Verzeichnungsmodell der Kamera 2

A.1.4. Verzeichnung der Kamera 3

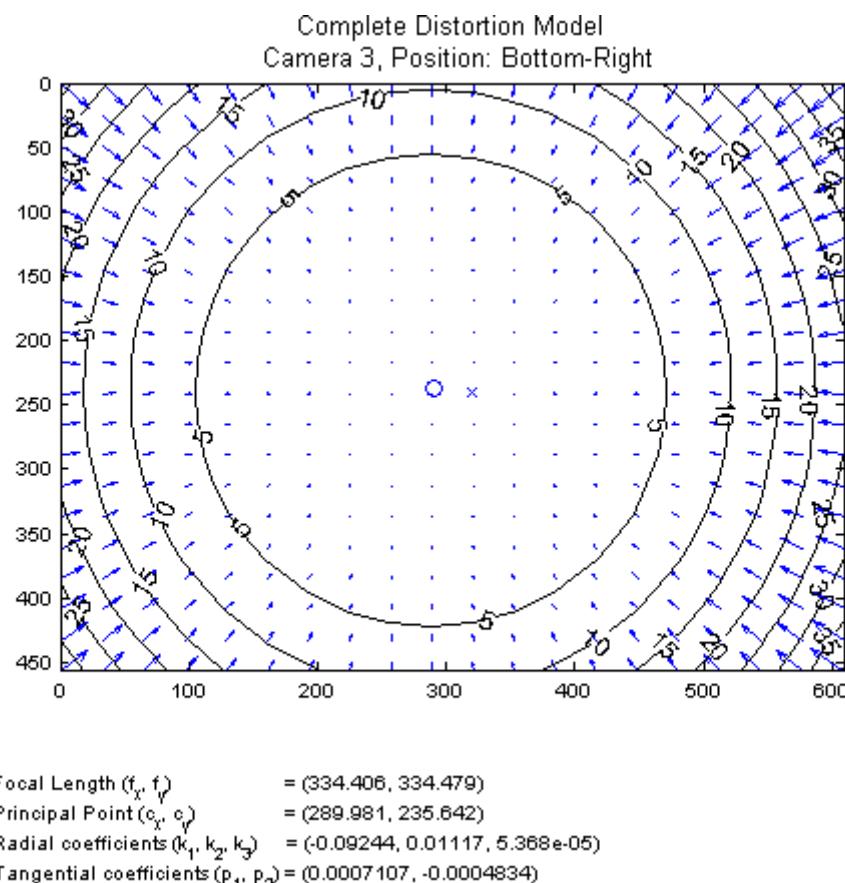


Abbildung A.7.: Vollständiges Verzeichnungsmodell der Kamera 3

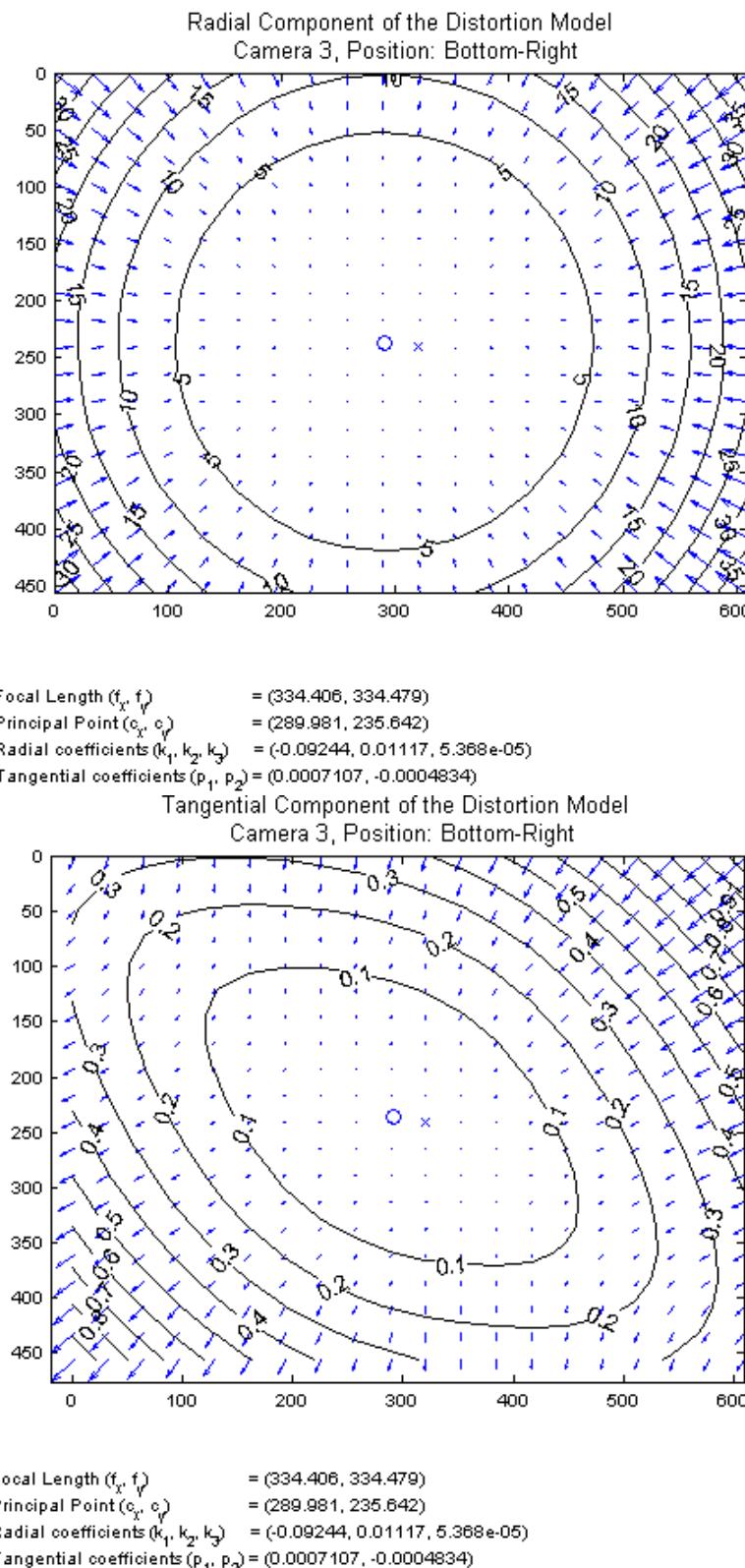


Abbildung A.8.: Radiales und tangentiales Verzeichnungsmodell der Kamera 3

A.1.5. Verzeichnung der Kamera 4

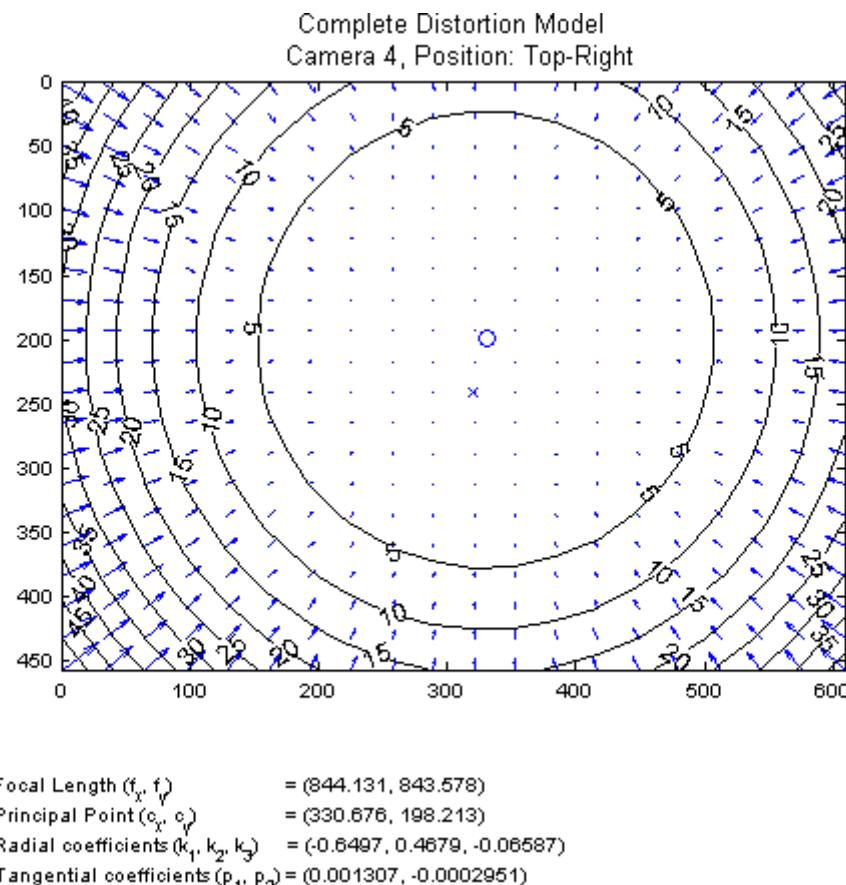


Abbildung A.9.: Vollständiges Verzeichnungsmodell der Kamera 4

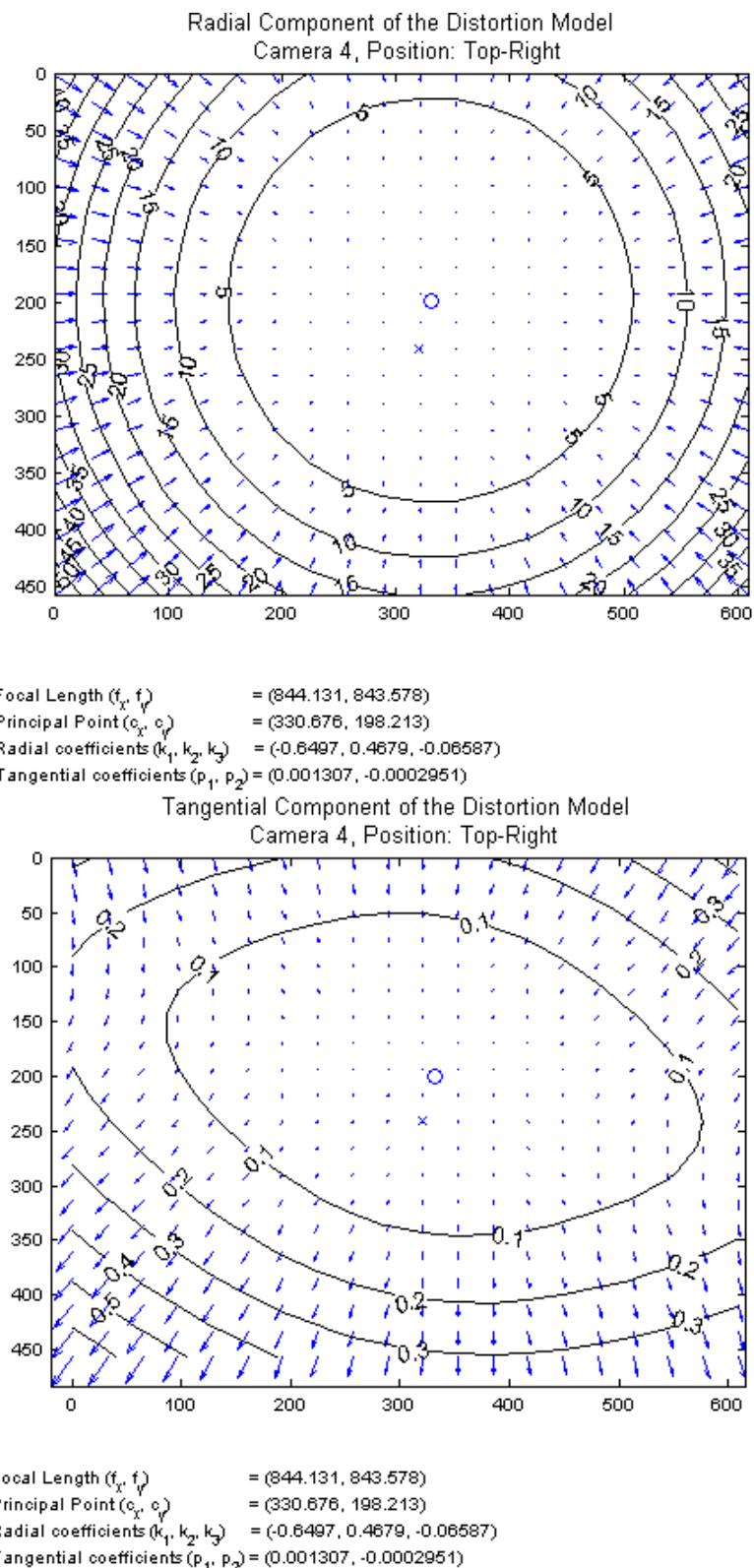


Abbildung A.10.: Radiales und tangentiales Verzeichnungsmodell der Kamera 4

A.1.6. Verzeichnung der Kamera 5

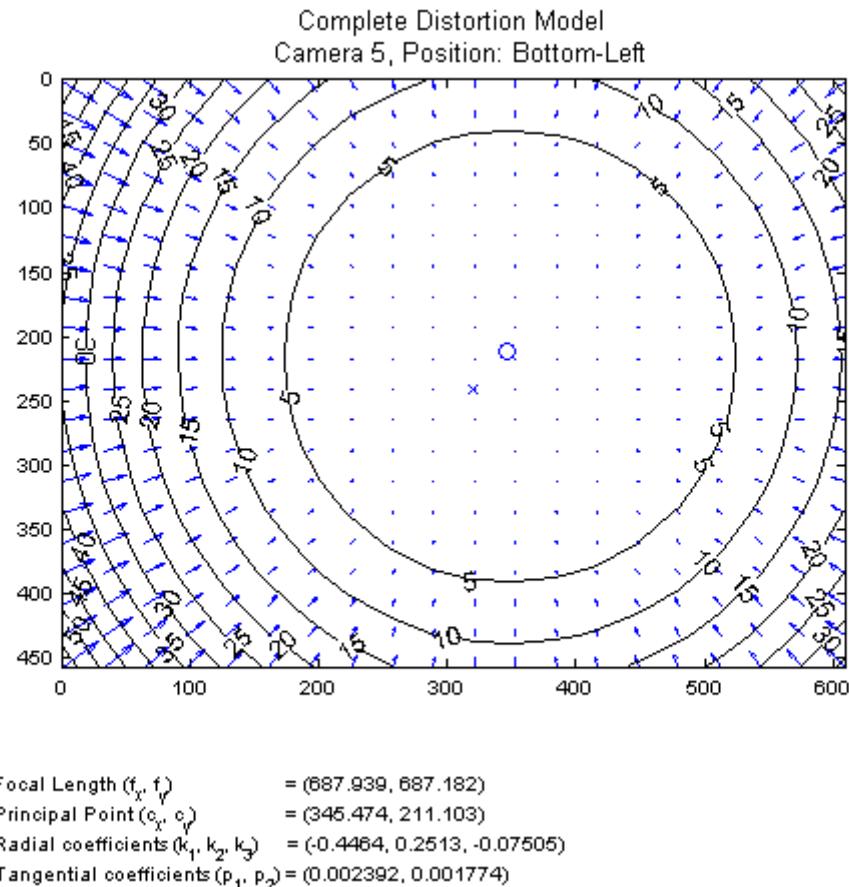


Abbildung A.11.: Vollständiges Verzeichnungsmodell der Kamera 5

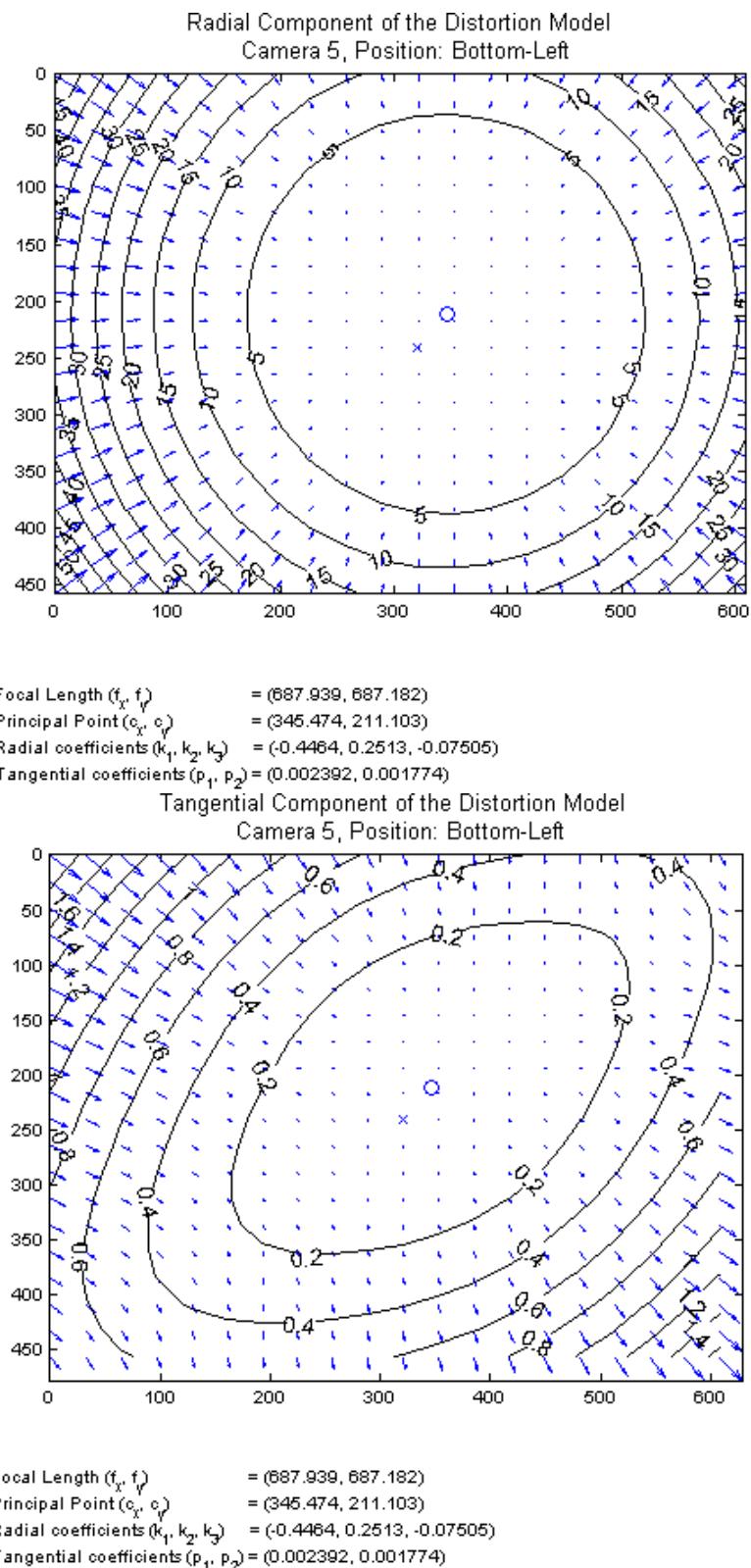


Abbildung A.12.: Radiales und tangentiales Verzeichnungsmodell der Kamera 5

A.2. Übersichtsplan der Bodenmarkierungen

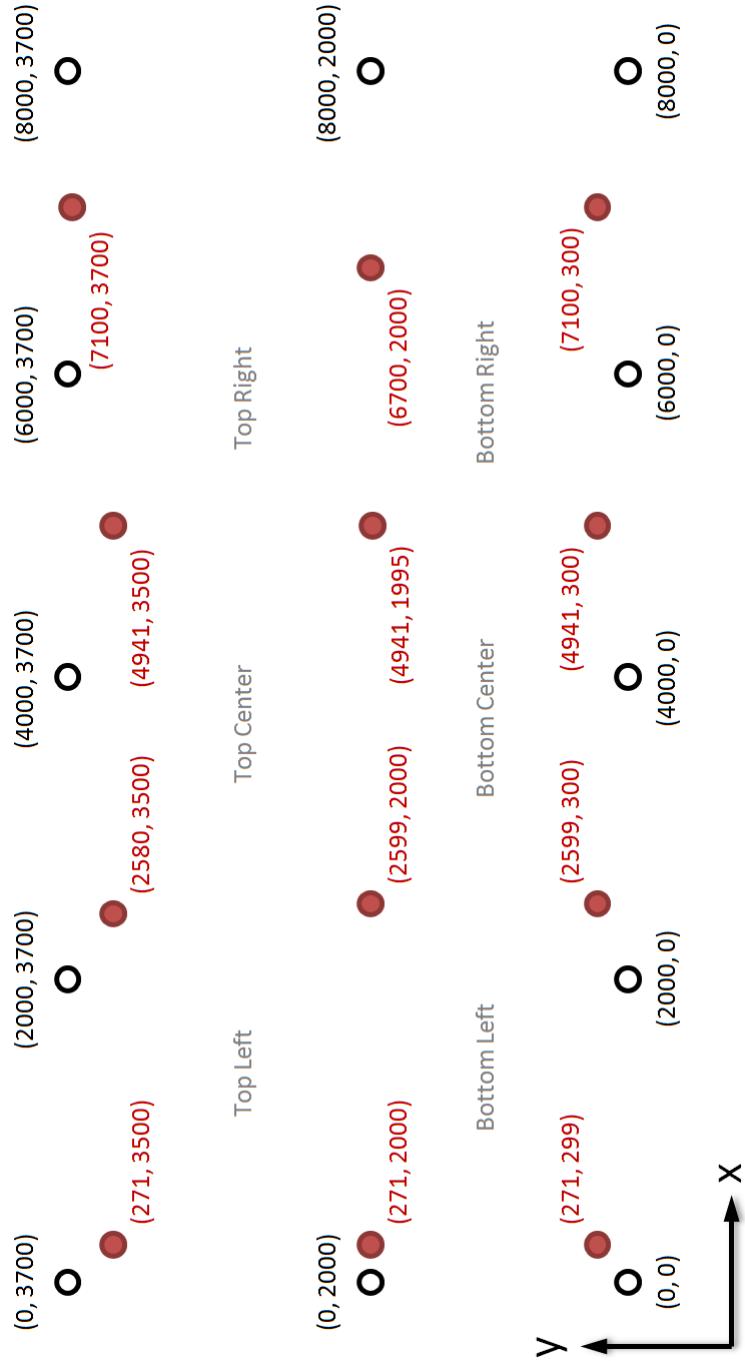


Abbildung A.13.: Übersichtsplan der Bodenmarkierungen (Maße in mm)