HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

Implementierung einer Bilderkennung zur Robotersteuerung

Projektarbeit

im Modul Projektarbeit 3

Studiengang Maschinenbau (dual)

der Hochschule Ruhr West

Marco Pastore

Matrikelnr. 10016156

Erstprüferin:	M. Sc. Stefanie Sell
Zweitprüfer:	Prof. Dr. Marc Stautner

Mülheim an der Ruhr, Dezember 2024

Kurzfassung

Diese Projektarbeit widmet sich der Integration von Bilderkennungstechnologien in die Robotik. Das Projekt mit dem Titel „Implementierung der Bilderkennung zur Robotersteuerung“ nutzt die Fähigkeiten eines Raspberry Pi und einer USB-Kamera, um ein System zum robotergestützten Sortieren zu entwickeln. Unter der Verwendung von OpenCV in Python besteht das Hauptziel des Projekts darin, einfache geometrische Körper anhand ihrer Form und Farbe zu unterscheiden und ihre Positionen möglichst präzise zu bestimmen, um durch den xArm 7 Roboter von uFactory gegriffen und sortiert zu werden.

Das Projekt beinhaltet fortgeschrittene Bildverarbeitungstechniken wie morphologische Filter, Konturerkennungsmethoden und insbesondere Stereo-Vision. Die Einrichtung umfasst eine sorgfältige Kalibrierung unter Verwendung von HSV-Farbcodes und einem Schachbrettmuster. Die Kamera wird mit Hilfe von 3D-Drucktechnologien montiert. Die Kommunikation zwischen Raspberry Pi und xArm 7 wird über elektrische Signale realisiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

bzw	beziehungsweise
RPi	Raspberry Pi
IoT	Internet of Things
CV	Computer Vision
OpenCV	Open-Source Computer Vision
PLA	Polylactide
HT	Hough Transformation
CHT	Circular Hough Transformation (Hough Transformation für Kreise)
GPIO	General Purpose Input Output
venv	virtual environment
DPA	Douglas-Peucker Algorithmus
XML	Extensible Markup Language

1 Einleitung

Mit dem Beginn der industriellen Produktion entstand das Verlangen der Menschen nach einer effektiveren Gestaltung von Produktionsprozessen. Bald darauf wurden bereits erste, einfache Fertigungsabläufe mit steuer- und regelbaren Maschinen automatisiert. Der Mensch übernimmt nur noch das Einrichten und Überwachen der geschaffenen Maschinen. Dabei stehen bis heute immer gleiche Ziele im Vordergrund: Die Erhöhung der Produktivität durch Verkürzung der Fertigungszeiten, die Erleichterung der menschlichen Arbeit, die Senkung der Kosten und die gleichzeitige Erhöhung der Qualität. [1, S. 5f.]

Um diese Ziele zu erreichen, können Roboter verwendet werden. Längst finden Roboter verschiedenster Arten nicht mehr ausschließlich in Fabriken Anwendung. [2, S. 16] Roboterkonzepte, wie sie heute bekannt sind, wurden erst mit dem Einsatz komplexer Mechanik und der Einführung der Elektrizität als Antrieb kompakter Elektromotoren modern. Durch die Entwicklung digitaler Steuerungen und Vorstufen von künstlicher Intelligenz konnten Roboter immer komplexere Arbeiten ausführen. [2, S. 20]

Mit der steigenden Leistungsfähigkeit der Computer sind Roboter heutzutage in der Lage, menschliche Tätigkeiten nachzuahmen. Dazu gehört unter anderem das *maschinelle Sehen* bzw. wie es oft in der (englischsprachigen) Literatur bezeichnet wird: *computer vision* (oder abgekürzt: CV). Die Forschung und Industrie sind noch weit davon entfernt allgemeingültige „sehende Maschinen“ zu bauen, die so gut sehen und interpretieren können, wie Menschen. Trotzdem finden Bildverarbeitung und -erkennung bereits in den verschiedensten Bereichen Anwendung und können mit stetig zunehmender Rechnerleistung verbessert und erweitert werden. [3, S. 1]

2 Zielsetzung und Gang der Arbeit

Ziel dieser Projektarbeit ist das Entwickeln und Implementieren einer Bilderkennung, um den Roboter xArm 7 des Herstellers uFactory im Robotiklabor der Hochschule Ruhr West zu steuern.

Dazu soll ein Raspberry Pi der dritten Generation eingerichtet und in Verbindung mit einer USB-Kamera – hier das Modell *BRIO ULTRA-HD PRO BUSINESS-WEBCAM* des Herstellers *logitech* – verwendet werden. Die Kamera wird, unter Zuhilfenahme einer eigens konstruierten und gefertigten Adapterlösung, zwischen Endeffektorflansch des Roboterarmes und dem Endeffektor *Gripper* des Herstellers *uFactory* montiert, sodass die Kameraposition stets mit der Roboterposition übereinstimmt. Der Roboter soll dann in verschiedene definierte Positionen fahren können, um Bilder aus unterschiedlichen Perspektiven aufzunehmen.

Die Bilderkennungssoftware wird dabei mit Hilfe der O*penCV*-Bibliothek in Python entwickelt. Sie soll in der Lage sein, verschiedene einfache geometrische Körper, in verschiedenen Grundfarben voneinander zu unterscheiden.

Außerdem soll die räumliche Position der Körper bestimmt werden. Dazu kann mit dem Verfahren der sogenannten *Photogrammetrie* gearbeitet werden. Um über die Photogrammetrie dreidimensionale Koordinaten zu bestimmen, werden mindestens zwei Bildaufnahmen aus verschiedenen Positionen benötigt. Über den Abstand der Kameras, deren technische Daten sowie die Verschiebung von signifikanten Stellen im Bild, wie zum Beispiel Ecken, Kanten und Mittelpunkte, lässt sich die dreidimensionale Position des Körpers geometrisch bestimmen. Anschließend soll die ermittelte Position an die Robotersteuerung übergeben werden, damit der Roboter das Objekt greifen kann.

Die zu identifizierenden und greifenden Körper werden ebenfalls im Rahmen der Projektarbeit entworfen und mittels 3D-Druck aus verschiedenfarbigem PLA-Filament gefertigt.

3 Stand der Technik

In diesem Kapitel wird der aktuelle Stand der Technik der zusätzlich verwendeten Soft- und Hardware beschrieben.

3.1 Einplatinencomputer insbesondere Raspberry Pi

Der Raspberry Pi – oft abgekürzt als RPi – ist ein kompakter sogenannter Einplatinencomputer. Das heißt, dass sich alle notwendigen Komponenten auf einer Platine befinden. Zudem lässt sich ein Monitor sowie Eingabegeräte über die USB-Schnittstellen anschließen. Es ist zusätzlich ein Netzwerk- und ein Audioanschluss vorhanden, sodass der RPi seitens der Hardware einem üblichen Computer entspricht. [4. S. 1] Der RPi verfügt zudem über sogenannte General Purpose Input Output Pins (kurz: GPIO-Pins). Über diese Pins können digitale Signale gesendet oder empfangen werden.

Mittlerweile stehen einige unterschiedliche Modelle zum Verkauf. Die Modelle unterscheiden sich dabei hauptsächlich an der Hardware-Leistung. Das aktuell neuste und damit leistungsstärkste Modell ist der RPi 5. Es gibt im Gegensatz dazu auch aktuelle preiswertere und leistungsschwächere Alternativen wie den RPi Zero W. [5, S. 13]

Es lassen sich verschiedene Versionen des Betriebssystems Linux installieren. Diese verschiedenen Versionen werden auch Distributionen genannt. Üblich ist das herstellerseitig herausgegebene Raspberry Pi OS, welches ebenfalls eine Linux-Distribution darstellt und speziell für die begrenzte Hardware-Leistung optimiert ist. [4, S. 1]

Die Einsatzbereiche sowie die Benutzer der RPis sind vielseitig: Die Intention der RPi-Entwickler war, ein kostengünstiges System zu schaffen, welches dem Benutzer das Programmieren von Computern näherbringt. So kann der RPi beispielsweise für kleine Elektronikschaltungen im heimischen Smart Home Netzwerk dienen oder den Office PC ersetzen. [4, S. V] Die RPi-Plattform kann jedoch auch in professionellen Anwendungen verwendet werden. Unter

Einhaltung der industriellen Vorgaben kann diese als kostengünstige Alternative in IoT-Anwendungen Verwendung finden. [Weiterführende Literatur: 6, S. 292]

3.2 Computer Vision

Computer Vision (CV) ist ein sehr komplexes Forschungsfeld. Es ist unter anderem so schwer zu entwickeln, weil Sehen das mathematisch inverse Problem der einfacheren Computergrafik ist: Die CV-Software versucht, unbekannte Ergebnisse aus unzureichender Information, meist nur eins oder wenige Bilder, zu extrahieren. Die stetigen Entwicklungen der letzten zwanzig Jahre haben dazu beigetragen, dass es heute Techniken gibt, um 3D-Modelle aus tausenden sich überlappenden Fotos zu erstellen. Mit einer ausreichend großen Anzahl von Bildern aus verschiedenen Perspektiven lassen sich dazu dichte, dreidimensionale Punktewolken berechnen. CV ist auch heute noch sehr anfällig für Fehler. Trotz dessen kann und wird CV bereits in verschiedensten Bereichen der Industrie eingesetzt. Einige Beispiele dafür sind: Handschriftenerkennung, Autonom fahrende Fahrzeuge, Mechanische Qualitätskontrollen über Röntgenaufnahmen und viele weitere. [7, S. 3-5], [3, S. 1-3]

Für solche Anwendungen wird Bildverarbeitung mit komplexen, mathematischen Algorithmen benötigt. OpenCV (Open Source Computer Vision Library) stellt über 2500 solcher Algorithmen optimiert in einer gesammelten Anwendungsbibliothek kostenfrei zur Verfügung. Das Projekt OpenCV ist per Apache 2 Lizenz lizenziert und ist somit Open-Source, das bedeutet, dass der Code frei zugänglich ist und jeder Nutzer zur Weiterentwicklung beitragen kann. Auf die Algorithmen kann, nach Download der entsprechenden Software-Pakete, zugegriffen werden. [8, S. 7], [9]

4 Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen, die als Basis der nachfolgenden Programmierung dienen, erklärt. Der Fokus liegt dabei auf den Konzepten, aber auch den mathematischen Herleitungen sowie Definitionen.

4.1 Farbräume

In der digitalen Bildverarbeitung werden Farbräume benötigt, um Farben oder Farbbereiche festzulegen. [10, S. 3f.]

Eine Farbe bezieht sich dabei immer auf einen Pixel. Ein Pixel ist das kleinste Element eines diskreten Bildes und trägt die Information über dessen Farbe in einem definierten Farbraum.

4.1.1 Grauwert- und Binärbilder

Der einfachste und kleinste Farbraum ist der sogenannte Binärraum. In Binärbildern wird dabei jedem Pixel entweder der Wert 0 für schwarz oder der Wert 1 für weiß zugeordnet. [10, S. 3]

Grauwertbilder haben oft eine Farbtiefe von 8 Bit. Das bedeutet, dass jedem Pixel ein Wert zwischen 0 und 255 zugewiesen wird. So sind insgesamt $2^8 = 256$ verschiedene Grautöne möglich.

4.1.2 RBG-Farbraum

Der am häufigsten verwendete Farbraum ist der RGB-Farbraum. Dieser kodiert seine Farbwerte durch Tripel $(R, G, B) \in [0, 100]^3$, wobei jede Komponente den Anteil der Grundfarben *Rot*, *Grün* und *Blau* in Prozentwerten darstellt. Ein verwandter Farbraum ist der BGR-Farbraum, bei dem der Hauptunterschied die Reihenfolge, in der die Farbkanäle angegeben werden, ist. Der BGR-Farbraum wird standardmäßig von OpenCV verwendet. Farbräume lassen sich dabei meist als geometrische Formen visualisieren. [10, S. 3f.], [11, S. 50]

In Abbildung 1 wird der RGB-Farbraum als Würfel dargestellt.

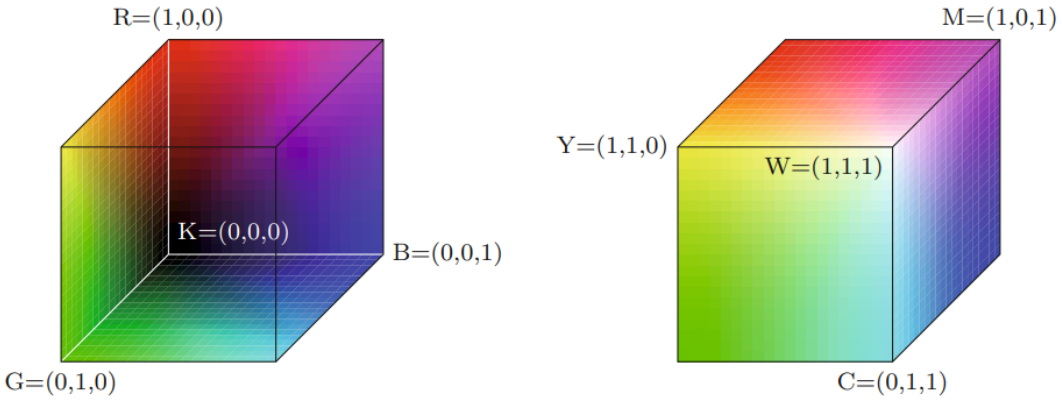


Abbildung 1: RGB-Farbraum als Würfel visualisiert [10, S. 4, Abb. 1.2]

4.1.3 HSV-Farbraum

Für die Verarbeitung von Farbbildern wird oft der intuitivere, sogenannte HSV-Farbraum verwendet. Hier werden die Farbwerte durch Tripel $(H, S, V) \in [0, 360] \times [0, 100] \times [0, 100]$ kodiert. Die Kanäle sind der Farbton (*Hue*), die Sättigung (*Saturation*) und der Hellwert (*Value*). Der Farbton ist als Winkel zu interpretieren, die Sättigung und der Hellwert jeweils als Prozentsatz. Daher lässt sich der HSV-Farbraum als Kegel oder Zylinder visualisieren. [10, S.4], [11, S. 50-54]

Abbildung 2: Links: HSV-Farbraum als Zylinder visualisiert [10, S. 5, Abb. 1.3]; Mitte: Mantelfläche des Zylinders [11, S. 52, Abb. 1.26d]; Rechts: Halbseitiger Querschnitt durch Zylinder [Eigene Darstellung]

4.2 Morphologische Filter

Morphologische Filter sind zur Analyse räumlicher Strukturen auf Bildern essenziell. Es lassen sich zwei grundlegende morphologische Operationen unterscheiden: Die Erosion und die Dilation. Diese werden meist auf binäre, bzw. Grauwertbilder angewendet oder in Ausnahmefällen auf Farbbilder, dann aber auf jeden Farbkanal einzeln. [10, S. 82f.]

4.2.1 Erosion

Es liegt ein Binärbild vor: Das Konzept bei der Erosion ist es, für jeden Pixel des Bildes die umliegenden Pixel in einem definierten Raum auszuwerten. Dieser Untersuchungsbereich kann unterschiedliche Formen haben, oft ist er kreisförmig oder quadratisch. [12]

Falls der auszuwertende Pixel den Wert 1 (weiß) hat und alle anderen Pixel im definierten Bereich auch den Wert 1 (weiß) haben, behält der Pixel seinen Wert 1 (weiß). Wenn nicht alle Pixel im Bereich den Wert 1 haben, wird der Pixel den Wert 0 (schwarz) annehmen. Das führt dazu, dass die Objekte meist kleiner werden. Es wird vor allem genutzt, um (weißes) Bildrauschen zu entfernen. [12]

In der folgenden Abbildung 3 wird ein, durch weißes Bildrauschen, gestörtes Binärbild mit Hilfe eines quadratischen Erosionsfilters bearbeitet. Der Einfluss der Größe des Untersuchungsbereiches wird dabei deutlich, muss aber für jede Anwendung neu beurteilt werden.

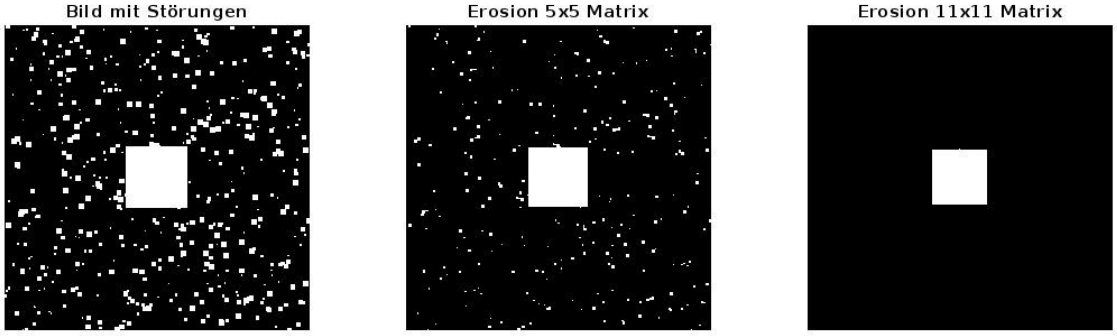


Abbildung 3: Beispiel Erosions-Filter [Eigene Darstellung]

4.2.2 Dilation

Die Dilation funktioniert genau umgekehrt zur Erosion. Es liegt ein binäres Bild vor: Wenn im definierten Bereich um einen Pixel mindestens ein anderer Pixel den Wert 1 (weiß) hat, erhält der auszuwertende Pixel den Wert 1 (weiß). Das führt insgesamt dazu, dass Objekte insgesamt größer werden. Dilationsfilter sind besonders nützlich, um unterbrochene Objekte wieder zu vereinen. [12]

In der folgenden Abbildung 4 wird beispielhaft ein, durch schwarze Artefakte unterbrochenes, Quadrat mit einem quadratischen Dilationsfilter bearbeitet.

Abbildung 4: Beispiel Dilations-Filter [Eigene Darstellung]

4.2.3 Zusammengesetzte morphologische Filter

Aus den beiden Grundoperationen lassen sich verschiedene zusammengesetzte morphologische Filter ableiten: Das Öffnen (auch Opening genannt) und das Schließen (auch Closing genannt). [10, S. 87ff.]

Das Öffnen beschreibt das hintereinander Ausführen von Erosion und Dilation, während das Schließen die Reihenfolge umdreht. Das Öffnen wird verwendet, um Rauschen zu minimieren, während das Schließen Löcher in Objekten auffüllt. [12] Es können dabei beliebige Filter verschieden oft miteinander kombiniert werden, um das gewünschte Ergebnis zu erzielen.

In der folgenden Abbildung 5 wird ein gestörtes Bild, mit Hilfe eines Schließen- und eines Erosionsfilter, entstört.

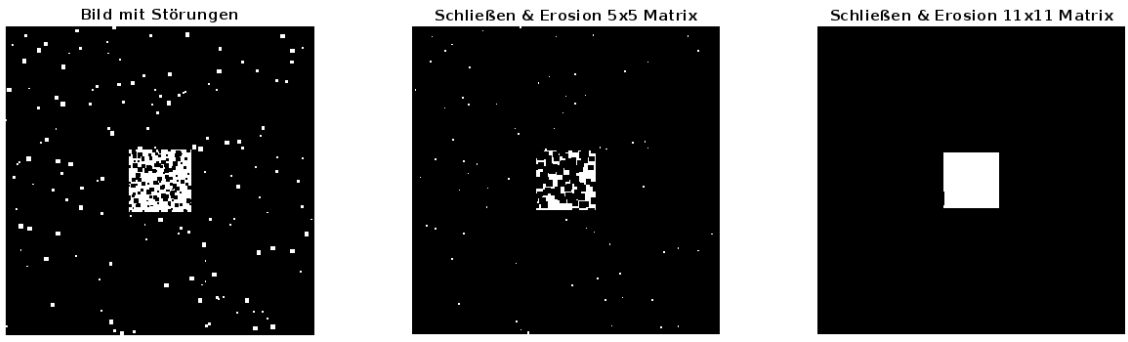


Abbildung 5: Beispiel Schließen-Filter mit anschließendem Erosions-Filter [Eigene Darstellung]

4.3 Konturermittlung

Eine Kontur ist definiert als Kurvenzug, der fortlaufend Punkte entlang einer Farbtons- oder Farbintensitätsgrenze miteinander verbindet. Dafür werden meist binäre Bilder verwendet. Dann gilt für die Kontur, dass diese der Grenze

zwischen Pixeln mit dem Wert 1 (weiß) und Pixeln mit dem Wert 0 (schwarz) folgt. [13], [7, S. 368f.]

In Abbildung 6 wird die Kontur eines Quadrates ermittelt und in Rot dargestellt.

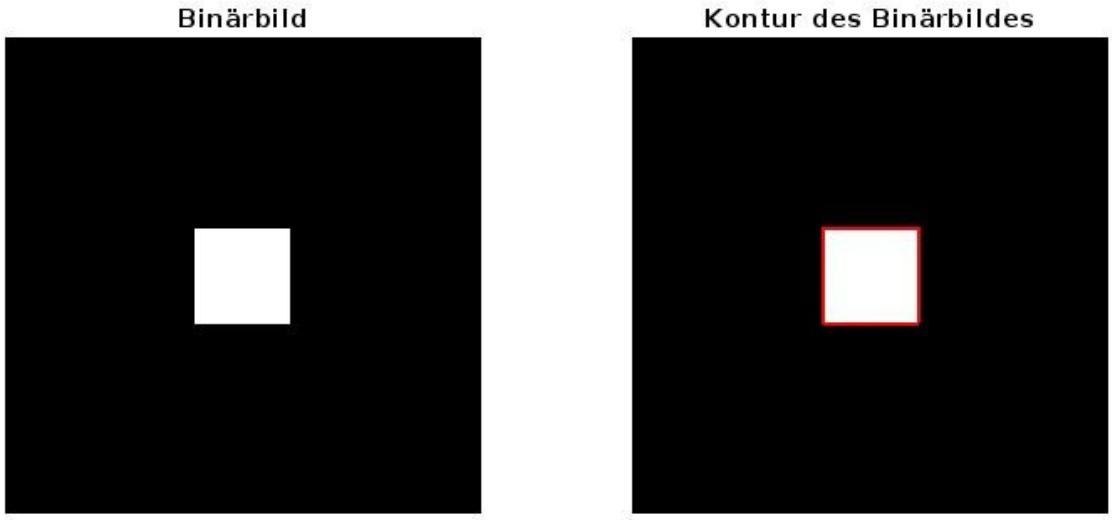


Abbildung 6: Beispiel Konturermittlung am Binärbild [Eigene Darstellung]

4.3.1 Kreisdetektion – Circular Hough Transformation

Um aus einem Bild abzuleiten, ob, bzw. welche, geometrische Form vorliegt, gibt es unter anderem die Hough-Transformation (abgekürzt HT). Mit Hilfe der allgemeinen HT lassen sich einfache parametrisierbare Geometrien aus Kanten ermitteln. Dieses Verfahren kann auch für zur Erkennung von Kreisen verwendet werden und wird, als Circular Hough Transformation bezeichnet. (dt: Hough Transformation für Kreise abgekürzt CHT). [14, S. 255]

Die Parameterform der gesuchten Geometrie wird in den Parameterraum, der hier Hough-Raum genannt wird, überführt. Für einen Kreis mit dem Mittelpunkt (x_0, y_0) und dem Radius r gilt die Parameterform:

$$(x - x_0)^2 + (y - y_0)^2 - r^2 = 0 \qquad \text{(Gl. 4.1)}$$

Daraus ergibt sich der Hough-Raum durch die Parameter (x_0, y_0, r). Es wird ein Intervall der möglichen Radii vorgegeben und der Definitionsbereich so beschränkt. Der Algorithmus sucht nun für jeden einzelnen Pixel (x, y) mögliche Parameter, für welche die oben gezeigte Parameterform erfüllt ist. [14, S.258f.]

Die CHT arbeitet nach dem sogenannten Abstimmungsprinzip: Jeder Pixel stimmt für seine ermittelten Hough-Räume ab, wodurch eine Verteilung

entsteht. Abschließend werden die lokalen Maxima dieser Verteilung bestimmt, welche dann die ermittelten Kreise darstellen. [14 S. 255, 258f.]

Da dieses Verfahren ineffizient und langsam ist, wird das Verfahren in der Praxis beschleunigt, indem es in zwei kleinere, maximal zweidimensionale Probleme überführt wird. Für jeden Kreis gilt, dass der Gradient jedes Kantenpixels in das Kreiszentrum zeigt. Der Schnittpunkt aller Gradienten ist somit der Kreismittelpunkt. Als nächstes kann der Radius über eine eindimensionale HT ermittelt werden, da der Kreismittelpunkt nun bekannt ist. Dieses Verfahren wird entsprechend der Dimensionen der Einzelprobleme auch 2-1-HT genannt. OpenCV-Bibliothek verwendet die 2-1-HT. [14, S. 259], [15]

4.3.2 Ecken finden – Douglas-Peucker Algorithmus

Um Eckpunkte in Konturen zu finden, kann der Douglas-Peucker Algorithmus verwendet werden. Dieser Algorithmus approximiert Konturen, indem er sie durch gerade Linien, sogenannte Polygone, darstellt:

Zuerst wird der erste Punkt auf der Kontur als Anker und der letzte als Gleitpunkt definiert. Die Gerade zwischen diesen beiden Punkten wird als ein Segment definiert. Dann werden die dazwischen liegenden Punkte entlang der gekrümmten Linie untersucht, um den Punkt zu finden, der den größten senkrechten Abstand zur Geraden aufweist – also am weitesten von der Geraden entfernt ist. Ist dieser Abstand kleiner als der maximal definierte Toleranzabstand, so wird das gerade Segment als geeignet angesehen, die gesamte Linie darzustellen und der Algorithmus beendet. [16]

Ist die Bedingung nicht erfüllt, also der Abstand zur Geraden größer als der Toleranzwert, wird dieser Punkt zum neuen Gleitpunkt. Bei der Wiederholung des Zyklus bewegt sich der Gleitpunkt auf den Anker zu. Sobald der maximale Abstand eingehalten wird, wird der Gleitpunkt zum neuen Anker, und der letzte Punkt der Kontur wird als neuer Gleitpunkt festgelegt. [16]

Am Ende lässt sich das Polygon durch Geraden zwischen all den ermittelten Ankerpunkten darstellen. [16]

4.4 Photogrammetrie – Stereo-Vision und Triangulation

Die Photogrammetrie umfasst Methoden der Bildmessung, um die Form und den Standort eines Objekts aus einem oder mehreren Bildern des Objekts abzuleiten. Der primäre Zweck einer photogrammetrischen Messung ist die dreidimensionale Rekonstruktion eines Objekts in digitaler Form (Koordinaten, Punktwolken, 3D-Modelle). [17, S. 2]

Eine Rekonstruktion kann zum Beispiel über das sogenannte Stereosehen erfolgen, ein Verfahren zur passiven Bestimmung von Tiefeninformationen. Mit Hilfe von mindestens zwei Bildern derselben Szene, die aus verschiedenen Kamerapositionen aufgenommen wurden, lässt sich die Tiefe bestimmter Punkte errechnen. [3, S. 7]

Eine Kamera lässt sich mathematisch durch ein Lochkameramodell [Abbildung 7] beschreiben: Die Bildebene befindet sich in einem Abstand f, der in der Kameratechnik auch Brennweite genannt wird, hinter der Lochblende. Jeder Punkt der Szene wird auf einer Geraden, die durch den Szenenpunkt und die Lochblende läuft, auf die Bildebene projiziert. [3, S. 8], [17, S. 8f.]

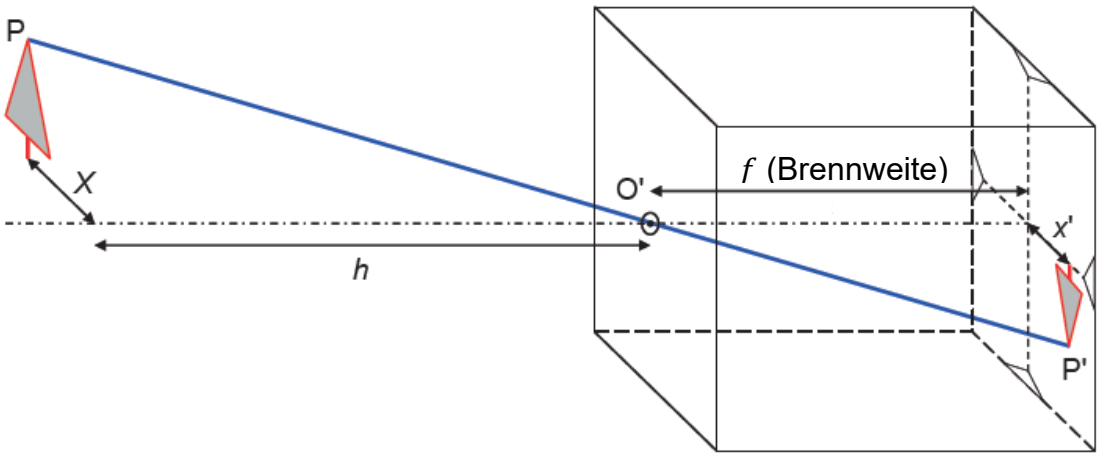


Abbildung 7: Lochkameramodell [In Anlehnung an 17, S. 8, Abb. 1.7]

Wenn zwei Kameras bzw. deren Bilder so überlagert werden, dass die optischen Achsen parallel sind und die Kameras lediglich auf der x-Achse um den Abstand b (auch baseline genannt) verschoben sind, kann die Tiefe z mit der Gleichung 4.2, aufgrund der mathematisch ähnlichen Dreiecke bestimmt werden. [3, S. 9f.] [Abbildung 8]

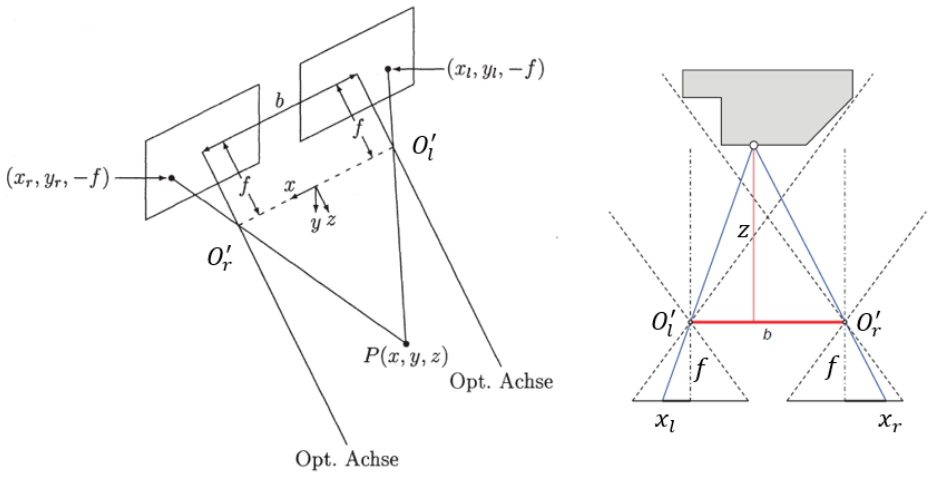


Abbildung 8: Stereogeometrie mit parallelen optischen Achsen. Links: dreidimensionale Ansicht [In Anlehnung an 3, S. 10, Abb. 2.1] Rechts: Normale Draufsicht [In Anlehnung an 17, S. 154, Abb. 3.38]

$$\frac{z}{b} = \frac{f}{|x_l| + |x_r|} \Leftrightarrow z = \frac{b \cdot f}{|x_l| + |x_r|} = \left[mm = \frac{mm \cdot px}{px + px}\right] \qquad \text{(Gl. 4.2)}$$

Die Brennweite einer Kamera wird in der Praxis meist in Millimetern angegeben. Wie die oben gezeigte Gleichung darstellt, muss allerdings mit einer Brennweite in Pixel gerechnet werden, damit die Tiefe in Millimetern berechnet wird. Experimentell lässt sich der Aufnahmewinkel besser bestimmen als die Brennweite. Aus einer geometrischen Überlegung, mit ähnlichen Dreiecken, lässt sich folgender Zusammenhang zwischen dem Aufnahmewinkel θ und der Brennweite f_{px} in Pixeln ermitteln.

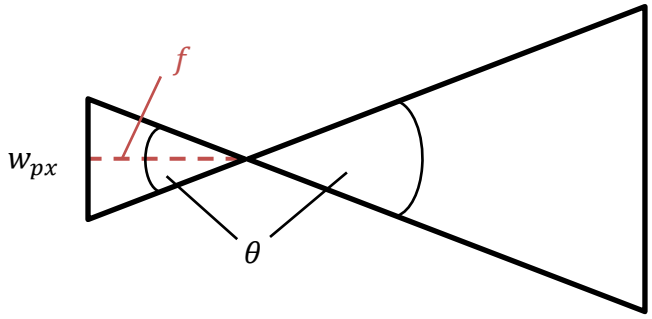


Abbildung 9: Ähnliche Dreiecke zur Bestimmung der Kamerabrennweite in px [Eigene Darstellung]

$$f_{px} = \frac{0{,}5 \cdot w_{px}}{\tan\left(0{,}5 \cdot \theta\right)} \qquad \text{(Gl. 4.3)}$$

5 Eingeschlagener Realisierungsweg

In diesem Kapitel werden die fertigungstechnische Umsetzung der Hardware sowie die Konzepte hinter Programmierung der Bilderkennungs-Software thematisiert.

5.1 Einrichtung des Raspberry Pi

Als Erstes wird das Betriebssystem auf den Raspberry Pi installiert. Aus den in Kapitel 3.1 *Einplatinencomputer insbesondere Raspberry Pi* genannten Gründen und dem Wunsch einer grafischen Benutzeroberfläche fällt die Wahl dabei auf das Raspberry Pi OS (64-Bit). Dieses kann mit der von Raspberry Pi herausgegeben Software „Raspberry Pi Imager“ direkt auf eine formatierte Micro-SD-Karte installiert werden. [18]

Im Raspberry Pi Imager lassen sich dann bereits einige Systemeinstellungen vornehmen. So wird ein Nutzer `pa3` mit dem dazugehörigen Kennwort `Pa3` angelegt.

Nach dem Start des RPis wird die Linux-Konsole gestartet und mit den folgenden Befehlen die neusten Aktualisierungen für das Betriebssystem installiert.

```
sudo apt-get update
sudo apt upgrade -y
```

Als Nächstes muss eine sogenannte virtuelle Entwicklungsumgebung (eng: virtual environment, kurz: venv) für Python erstellt werden, um das vorhandene Betriebssystem vor Beschädigungen durch Paketinstallationen zu schützen [19]. Als Name für die venv wird `python-venv` festgelegt.

```
python3 -m venv python-venv
```

Dann kann mit dem Befehl `pip` das OpenCV-Paket für Python in dessen venv installiert werden. [20]

```
python-venv/bin/pip install opencv-python
```

Außerdem werden weitere Pakete installiert, mit welchen man die GPIO-Pins des RPis und einen zusätzlichen 12-Bit Digital-Analog-Wandler (Typ: MCP4725) verwenden kann, um nachfolgend mit dem xArm kommunizieren zu können.

```
python-venv/bin/pip install RPi.GPIO
python-venv/bin/pip install adafruit-circuitpython-mcp4725
```

Das Python Skript kann mit dem folgenden Befehl direkt aus der Konsole gestartet werden.

```
python-venv/bin/python3 main.py
```

5.2 Kamerahalterung

Die Kamerahalterung, welche die USB-Kamera an den xArm befestigen soll, wurde mit Hilfe der CAD-Software (Computer-Aided-Design) SolidWorks 2024 konstruiert. Die Halterung besteht aus zwei Komponenten: Ein Adapterring, der zwischen Endeffektorflansch und Endeffektor geklemmt wird und ein Winkel, der die Kamera so auf dem Adapterring befestigt, dass die optische Kameraachse stets mit der Ausrichtung des Endeffektors übereinstimmt.

Abbildung 10: Links: Übersichtsaufnahme Kamerahalterung [Eigene Darstellung] Rechts: Detailaufnahme Kamerahalterung [Eigene Darstellung]

Der Adapterring ist symmetrisch ausgeführt, sodass es keine festgelegte Einbaulage gibt. Die beiden gekrümmten Langlöcher dienen der Durchführung der bereits vorhandenen M6 Senkkopfschrauben, welche den Endeffektor mit

dem Endeffektorflansch verbinden. Durch die Langlöcher lässt sich der Adapterring durch leichtes Lösen der Schrauben bereits um bis zu 90° drehen, falls erforderlich.

Am Rand des Adapterringes sind vier Positionen vorgesehen, an welche der Winkel zur Kamerahaltung montiert werden kann. Dazu werden die beiden Teile mit zwei Senkschrauben ISO 10642 – M4 x 10 verschraubt. Da der 3D-Druck als gewähltes Fertigungsverfahren nicht in der Lage ist, präzise metrische Gewinde zu erzeugen, wurden im CAD-Modell sechskantförmige Taschen vorgesehen. Der Druck wird – sobald die Taschen fertig ausgeführt sind – pausiert, sodass die dazugehörigen M4 Sechskantmuttern nach ISO 4032 eingesetzt werden können. Dann wird der 3D-Druck fortgesetzt und die letzten Deckschichten über die eingesetzten Muttern gedruckt.

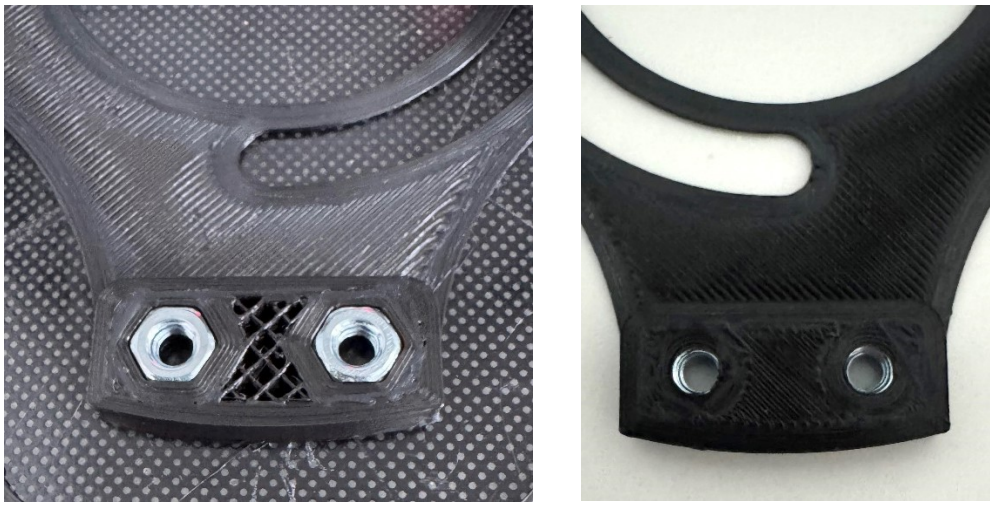


Abbildung 11: Links: Adapterplatte nach einsetzen der Sechskantmuttern. Rechts: Adapterplatte nach Abschluss des Druckes [Eigene Darstellung]

In der Software CURA, wird das CAD-Modell unter definierten Fertigungseinstellungen in Maschinencode umgewandelt. Beide Teile wurden auf einem *Creality Ender 3V2* 3D-Drucker im Standard-Druckprofil in PLA (Polylactide, ein Kunststoffmaterial für den 3D-Druck) mit folgenden Abweichungen gefertigt:

Tabelle 1: Druckparameter Kamerahalterung

Drucktemperatur	210°C
Druckbetttemperatur	55°C
Druckgeschwindigkeit	50mm/s
Füllung	20%
Füllungsmuster	ZigZag

Für den Druck des Winkels wurde außerdem eine automatische Stützstruktur des Typen *Tree* verwendet.

Auf der langen Seite des Winkels ist eine ¼ Zoll Durchgangsbohrung mit Stirnsenkung für eine ¼ Zoll x ½ Zoll UNC Linsenkopfschraube vorgesehen, um die Kamera mit deren Stativgewinde zu befestigen.

5.3 Fertigung der Objekte

Als Objekte werden sogenannte 2,5-dimensionale Objekte gefertigt: Zylinder, Würfel, Quader, extrudierte Hexagone und extrudierte Oktagone. [Anhang A.3] Die Objekte werden dann aus rotem, orangem, blauem und grünem PLA dem 3D-Drucker gedruckt.

Es wird erneut das Standard-Druckprofil für PLA mit folgenden Abweichungen verwendet:

Tabelle 2: Druckparameter Objekte

Drucktemperatur	210°C
Druckbetttemperatur	55°C
Druckgeschwindigkeit	40mm/s
Füllung	50%
Füllungsmuster	ZigZag

5.4 Python Anwendung

Die auf dem RPi lauffähige Python Anwendung ist so konzipiert, dass Benutzer*innen mit Hilfe der Linux Konsole die einzelnen Unterprogramme starten, Daten verwalten und durch die Anwendung navigieren können. Dafür werden die eingebauten Funktionen `print()` und `input()` verwendet, um Texte in die Konsole einzugeben, beziehungsweise Eingaben anzufordern.

Die nachfolgende Abbildung 12 zeigt den Programmablauf, den Benutzer*innen beim ersten Verwenden befolgen sollten.

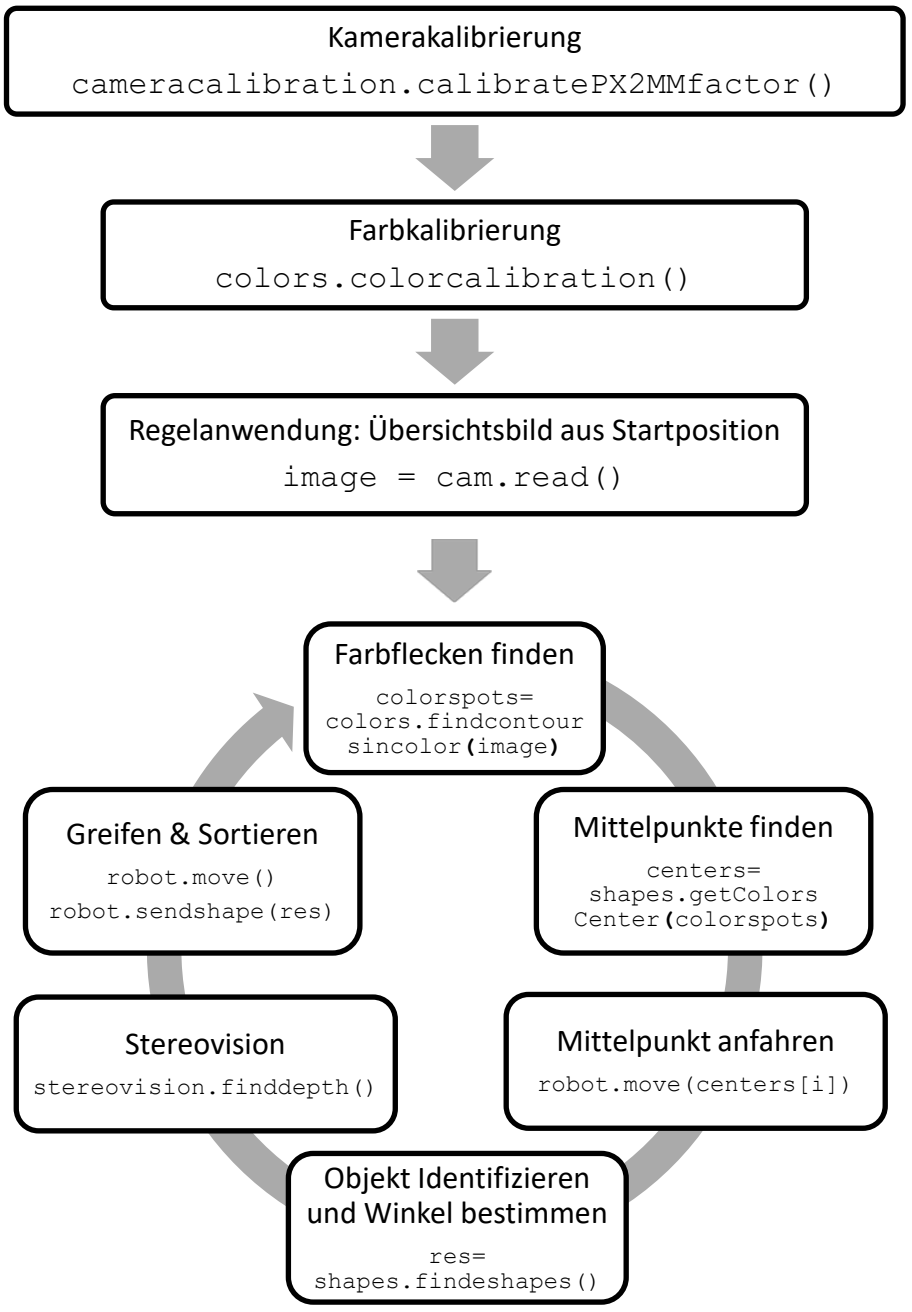


Abbildung 12: Visualisierung Programmablauf [Eigene Darstellung]

5.4.1 Kamerakalibrierung

Der erste Modus, der immer dann gestartet werden muss, wenn die Höhe der Startposition des xArm 7 verändert werden soll, führt die Funktion `cameracalibration.calibratePX2MMfactor()` der Kamerakalibrierung aus. Die Kalibrierung bestimmt einen Umrechnungsfaktor, der Entfernungen in der x-y-Ebene von Pixeln in Millimeter umwandelt.

Bevor die Kamerakalibrierung gestartet wird, muss sichergestellt werden, dass der xArm 7 in seiner Startposition steht und das beigelegte Schachbrettmuster [Anhang D] vollständig in der Kamera sichtbar ist. Es ist dabei essenziell, dass exakt das im Anhang befindliche Schachbrett verwendet wird und der Druck nicht skaliert wird, da die Kantenlängen auf $25mm$ festgelegt sind.

Sobald das Unterprogramm gestartet ist, löst die Software eine Aufnahme aus. Diese wird in ein Schwarz-Weiß-Bild umgewandelt. Mit eingebauten OpenCV-Funktionen können dann die Eckpunkte des Schachbrettmusters gefunden und markiert werden. Für die erste Reihe von Punkten wird dann jeweils der Abstand von jedem Punkt zu dessen rechten, benachbarten Punkt bestimmt. Der Umrechnungsfaktor wird definiert als die tatsächliche Kantenlänge in Millimetern dividiert durch den arithmetischen Mittelwert, der soeben bestimmten Abstände, in Pixel. [Anhang B.2, Z. 9-24]

$$f = \frac{Größe\ eines\ einzelnen\ Quadrats}{Ermittelter\ Abstand\ eines\ Punktepaares} = \left[\frac{mm}{px}\right] \qquad \text{(Gl. 5.1)}$$

Dieser Faktor wird abschließend mit OpenCV-Funktionen in eine XML-Datei gespeichert, sodass auch über den Programmneustart hinaus mit dem Faktor gerechnet werden kann. [Anhang B.2, Z. 26-31] Zwei weitere implementierte Funktionen `cameracalibration.cvtPX2MM()` [Anhang B.2, Z. 33-39] und `cameracalibration.cvtMM2PX()` [Anhang B.2, Z. 41-47] können dann auf den Faktor zugreifen und eine Entfernung von Millimetern in Pixel umrechnen und umgekehrt.

5.4.2 Farbkalibrierung

Wenn im Hauptmenü die Farbkalibrierung gestartet wird, wird die Funktion `colors.colorcalibration()` ausgeführt. Hier wird zuerst das `color.xml` Dokument eingelesen. In diesem Dokument befinden sich HSV-Farbcodes, die

die Anwender*innen über die Farbkalibrierung eingespeichert haben. Die bereits vorliegenden Farben werden mit deren Namen in der Konsole angezeigt. Die Benutzer*innen haben dann die Möglichkeit eine Farbe mit deren Listennummer auszuwählen und sie entweder zu löschen oder anzuzeigen. Beim Löschen wird der Eintrag aus dem XML-Dokument gelöscht. Wenn Anzeigen gewählt wird, öffnen sich zwei OpenCV Fenster: In einem Fenster wird das originale Kamerabild der angeschlossenen USB-Kamera angezeigt. Im Zweiten Fenster wird die Farbmaske, welche über die HSV-Farbcodes der jeweiligen Farbe im XML-Dokument definiert ist, angewendet. Zudem wird – wie in der späteren Regelanwendung – ein morphologischer Schließen-Filter angewendet, um Löcher in dem erkannten Objekt zu schließen. [Anhang B.3, Z. 8-74]

Statt einer Listennummer kann auch „n“ in die Konsole eingegeben werden, um eine neue Farbe zu erstellen oder eine vorhandene Farbe zu überschreiben. Es öffnen sich dann zwei Fenster: In einem Fenster wird das originale Kamerabild der angeschlossenen USB-Kamera angezeigt. Im zweiten Fenster die berechnete Farbmaske als Binärbild. Durch Klicken mit der linken Maustaste im originalen Bild werden die angeklickten Pixel einer Liste hinzugefügt. Diese Liste wird laufend ausgewertet. Es wird ein Bereich von Farben so bestimmt, dass alle der ausgewählten Pixel innerhalb dieses Bereiches liegen. Die daraus resultierende Farbmaske wird kontinuierlich aktualisiert und im zweiten Fenster angezeigt. [Anhang B.3, Z. 75-127]

Sind die Benutzer*innen mit dem Ergebnis zufrieden, so kann die Taste „c“ gedrückt werden: Die beiden Fenster werden geschlossen. In der Konsole wird abgefragt, ob die Farbmaske gespeichert werden soll. Ist das der Fall, muss der Name der soeben angelernten Farbe eingetippt werden. Liegt diese Farbe bereits in der XML-Datei mit bekannten Farben vor, so wird zur Sicherheit gefragt, ob die alte Farbe überschrieben werden soll. Werden alle diese Fragen bejaht, so wird der untere sowie obere Rand des Farbbereiches in das XML-Dokument gespeichert. Die Farbkalibrierung startet dann erneut. [Anhang B.3, Z. 128-160]

5.4.3 Regelanwendung

Zum Start der Regelanwendung wird der Roboter durch seine Steuerung in die Startposition gefahren. Die Kamera nimmt dann ein Übersichtsbild auf. Danach werden mit einer Funktion `colors.findcontoursincolor()` alle farbigen Bildbereiche mit Hilfe von Farbmasken extrahiert. Dazu werden die zuvor gespeicherten Farbbereiche aus der Datei `colors.xml` eingelesen. Dann wird je definierter Farbe eine binäre Farbmaske erstellt, indem jeder Pixel, der im Farbbereich liegt, den Wert 1 (weiß) und jeder Pixel, der nicht im Farbbereich liegt, den Wert 0 (schwarz) zugewiesen bekommt. Diese Maske wird dann mit einem Schließen Filter [Kapitel 4.2.3 Zusammengesetzte morphologische Filter] bearbeitet, um Löcher in den Farbbereichen zu schließen. Es wird angenommen, dass die weißen Bereiche Objekte sind. [Anhang B.3, Z. 163-181]

Um die Objekte besser identifizieren zu können, muss der Roboterarm mit der Kamera über die Mittelpunkte der Farbbereiche fahren. Es wird eine Funktion `shapes.getColorsCenter()` implementiert, um diese Mittelpunkte zu finden: Zuerst werden die Konturen der Farbbereiche bestimmt. Es werden nachfolgend nur Konturen mit einem Flächeninhalt von mindestens 2000px^2 berücksichtigt, um Bildartefakte, die besonders bei sehr ähnlichen Farbpaarungen wie orange und rot auftreten, zu vernachlässigen. Der Mittelpunkt wird dann mit Hilfe der mechanischen Momente der Kontur berechnet. [Anhang B.4, Z. 7-24]

Da die nachfolgende Robotersteuerung nur mit relativen Bewegungsbefehlen arbeitet, muss der Mittelpunkt des Objektes relativ zum Bildmittelpunkt bestimmt werden. Die Verschiebung zum Mittelpunkt wird dann mit der Funktion `cameracalibration.cvtPX2MM()` von Pixel in Millimeter umgerechnet und der Bewegungsbefehl mit der Funktion `robot.move()` an die Robotersteuerung gesendet. [Anhang B.1, Z. 46-50]

Dort angekommen wird erneut ein Foto aufgenommen und das darauf befindliche Objekt mit einer weiteren Funktion `shapes.findshapes()` identifiziert. Dabei gibt es zwei Optionen: Entweder das Objekt ist ein Zylinder oder ein extrudiertes n-Eck. Um die Form zu ermitteln, wird zuerst die Circular Hough Transformation auf eine weichgezeichnete Kopie des Bildes

angewendet. Wenn diese einen Kreis findet, wird davon ausgegangen, dass der Körper ein Zylinder ist. Findet die CHT keinen Kreis so wird die Kontur mit Hilfe des Douglas-Peuker Algorithmus durch ein Polygon approximiert. Die Anzahl der Eckpunkte, die der DPA findet, entspricht der Anzahl der Ecken des Objektes. Das Viereck bildet dabei eine Ausnahme: Hier wird zusätzlich das Verhältnis zwischen den Kanten bestimmt, um zu ermitteln, ob ein Quadrat (bzw. Würfel) oder Rechteck (bzw. Quader) vorliegt. [Anhang B.4, Z. 27-136]

Außerdem wird eine Gerade durch die beiden am weitesten links liegende Punkte aufgespannt. Der Winkel, den diese Gerade und die vertikale Bildachse einschließen, wird berechnet und so umgerechnet, dass er dem Winkel einer Verdrehung des Endeffektors im Intervall [-45°, 45°] entspricht. [Anhang B.4, Z. 137-155]

Zuletzt wird mit der Funktion `stereovision.finddepth()` der Abstand zwischen der Kamera und der Oberseite des Objekts bestimmt: Wie im Kapitel 4.4 Photogrammetrie – Stereo-Vision und Triangulation beschrieben, werden dazu zwei Bilder benötigt. In der Theorie könnten dafür als erstes Bild das Übersichtsbild und als zweites Bild, das, welches aufgenommen wird, sobald die Kamera über dem Objekt ist, verwendet werden. In der Praxis hat sich dieses Verfahren allerdings als zu ungenau dargestellt, weshalb zwei neue Bilder zur Ermittlung der Tiefe aufgenommen werden müssen. Dazu fährt der Endeffektor mit der Funktion `robot.move()` einen definierten Abstand nach links nimmt dort das erste Bild auf, fährt dann den doppelten Abstand wieder nach rechts und nimmt das zweite Bild auf. In beiden Bildern werden mit den bereits beschriebenen Funktionen `colors.findcontoursincolor()` und `shapes.getColorsCenter()` die Mittelpunkte der auf den Bildern befindlichen Objekte bestimmt. Damit nachfolgend nur mit den zwei Mittelpunkten des gesuchten Objekts gerechnet wird, wird basierend auf dem gefahrenen Abstand eine Vorhersage über deren Standort getroffen. Es wird davon ausgegangen, dass der Punkt, der je Bild am nächsten an der Vorhersage liegt, der gesuchte Punkt des Objektes ist. [Anhang B.5]

Diese Punkte werden dann in die hergeleitete Gleichung 4.1, die in der Funktion `triangulation.find_depth()` hinterlegt ist, eingesetzt. Die Funktion benötigt zudem noch den Abstand b der Kameras zwischen den Bildern sowie

den horizontalen Aufnahmewinkel θ der Kamera. Der Aufnahmewinkel $\theta \approx 45°$ wurde experimentell ermittelt, da dieser nicht im Datenblatt der Kamera vorhanden ist. [Anhang B.6] Dafür wurde ein Gliedermaßstab horizontal in das Kamerabild gelegt und die Bildbreite notiert. Mit dem Abstand der Kamera zur Tischoberfläche lässt sich dann über den Tangens der horizontale Aufnahmewinkel θ bestimmen. [Abbildung 9]

Von dieser ermittelten Tiefe muss dann noch der Abstand zwischen der Kamera und dem Endeffektor subtrahiert werden. In der Praxis hat sich außerdem ein Korrekturfaktor $K_d = 0{,}93$ als nötig erwiesen, um die Objekte möglichst zuverlässig greifen zu können. [Anhang B.1, Z. 59-62]

Die korrigierte Tiefeninformation zusammen mit dem Abstand von Kamera zum Endeffektor auf der y-Achse sowie dem zuvor ermittelten Verdrehwinkel des Endeffektors wird nun an die Robotersteuerung gesendet. Sobald die Position erreicht ist, schließt der Endeffektor mit der Funktion `robot.closeGripper()`. Dann wird das Objekt ausgerichtet, indem die ermittelte Tiefe zurück nach oben gefahren wird und die Verdrehung in die andere Richtung erfolgt. [Anhang B.1, Z. 64-68]

Dann wird mit der Funktion `robot.sendcolorandshape()` die Information über die Farbe und die Form des Objektes gesendet. Die Robotersteuerung verarbeitet dies und bringt das Objekt an die in der Robotersteuerung definierte Position. Sobald das Objekt am Zielort angekommen ist, wird der Gripper mit der Funktion `robot.openGripper()` geöffnet. Schließlich wird der xArm7 durch die Robotersteuerung wieder in die Ausgangsposition gefahren und der Prozess beginnt erneut. [Anhang B.1, Z. 69-74]

5.4.4 Übermittlung von Bewegungsbefehlen an den xArm

Zur Übermittlung von Bewegungsbefehlen an die Steuerung des xArm 7 werden weitere elektronische Komponenten mit dem RPi verkabelt. Über die Software des RPi werden die Bewegungsbefehle dann in elektrische Signale umgewandelt, die durch die Steuerung des xArm 7 wieder zurück in Koordinaten umgerechnet werden.

5.4.4.1 Anschluss der weiteren elektronischen Bauteile

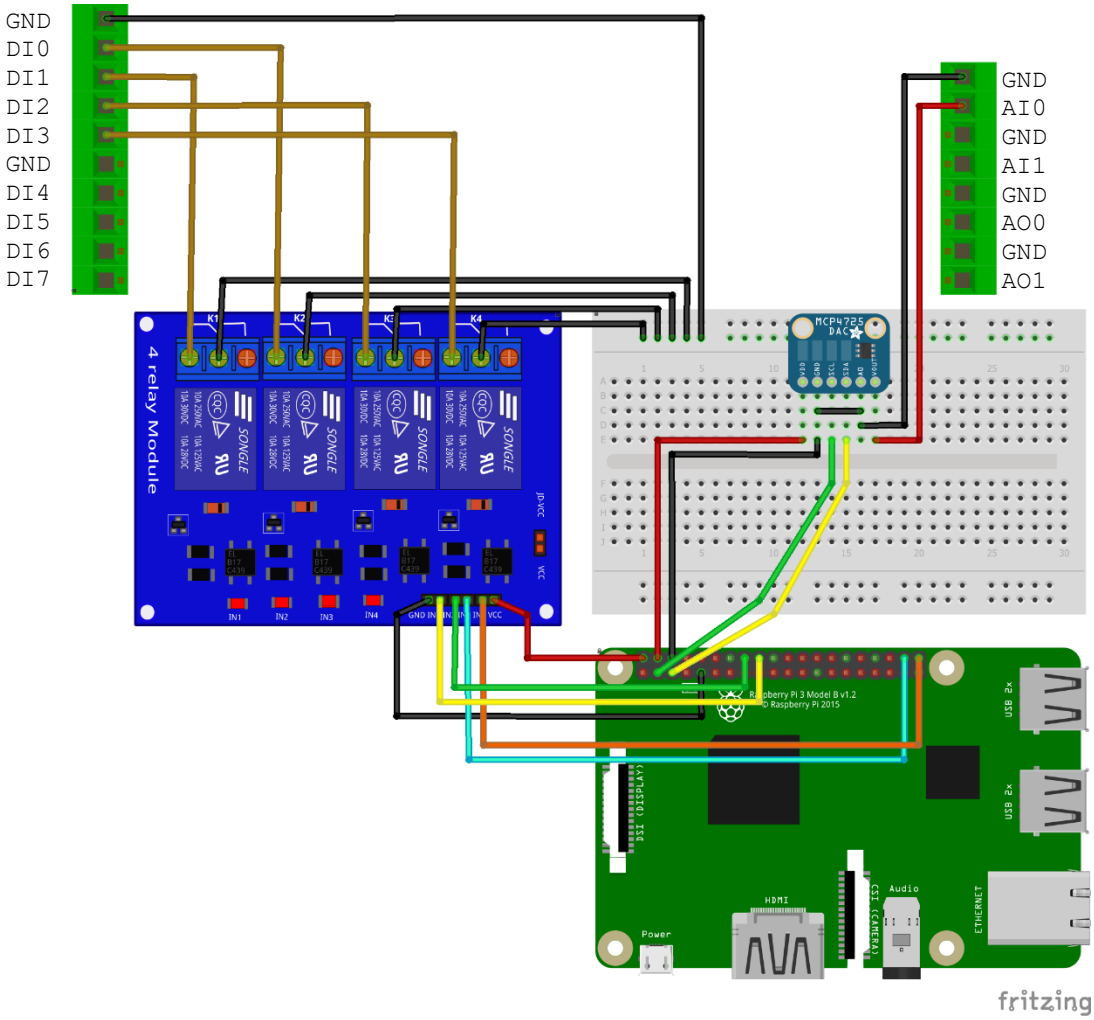


Abbildung 13: Verkabelung Raspberry Pi mit weiteren Elektronikbauteilen [Eigene Darstellung, erstellt mit fritzing.org Software]

In der Abbildung 13 ist die Verkabelung der Komponenten mit dem RPi dargestellt: Es wird ein 12-Bit Digital-Analog-Wandler (Typ: MCP4725) verwendet. [Weiterführende Literatur: 21 S. 535ff., 23] Die Pins VCC und GND werden mit der 5V Versorgungsspannung bzw. der Masse des RPis verbunden. Der Chip kommuniziert mit dem RPi über das sogenannte I^2C-Protokoll [Weiterführende Literatur: 22], dafür werden die Pins SDA und SCL mit den entsprechenden Gegenstücken GPIO 2 und GPIO 3 am RPi verbunden. Der Masseanschluss des analogen Ausganges GNC wird zur Masse des RPi auf den Pin GND gebrückt. Zuletzt wird das analoge Ausgangssignal OUT und die dazugehörige Masse GNC auf den analogen Eingang AI0 bzw. GND des Steckers der xArm7 Anschlussbox gelegt. [23]

Zudem wird ein Relaisboard mit vier Kanälen verwendet. Die Pins VCC und GND werden ebenfalls mit der 5V Versorgungspannung bzw. der Masse des RPis verbunden. Die Eingangssignal-Pins IN1 – IN4 werden dann mit GPIO 24, GPIO 23, GPIO 20, GPIO 21 des RPis verbunden. [Anhang B.3, Z. 10-18] Die Masse der digitalen Eingangsklemme der xArm7 Anschlussbox wird jeweils auf die Fußkontakte der Relais gelegt. Die Schaltkontakte werden jeweils mit den digitalen Eingängen DI0 – DI3 des Steckers in der Anschlussbox verbunden. [Weiterführende Literatur: 24, S. 405ff.]

5.4.4.2 Umwandlung der Bewegungsbefehle in elektrische Signale

Die Anwendung verwendet ausschließlich relative Koordinaten, die von der aktuellen Position des xArm 7 ausgehen. Mit Hilfe der implementierten Funktion `robot.move(x, y, z, pitch)` werden diese Relativkoordinaten dann in elektrische Signale umgewandelt. Es wird definiert, dass keine Bewegung – sprich 0mm Bewegung entlang einer beliebigen Achse – einer Spannung von 50% der maximalen Spannung (für den RPi: 3,3V) entspricht. So können auch negative Bewegungen entlang einer Achse abgebildet werden. Dieser prozentuale Wert wird mit der Auflösung des Digital-Analog-Wandlers verrechnet, um den der Spannung zugeordneten digitalen Wert zu ermitteln. [Anhang B.3, Z. 24-29] Über die folgenden Gleichungen werden die Koordinaten x, y, z und die Verdrehung des Endeffektors umgerechnet:

$$x_{digital} = 2^{12} \cdot \left(\frac{x_{Koordinate}}{400} + 0{,}5\right) \quad \text{(Gl. 5.2)}$$

$$y_{digital} = 2^{12} \cdot \left(\frac{y_{Koordinate}}{400} + 0{,}5\right) \quad \text{(Gl. 5.3)}$$

$$z_{digital} = 2^{12} \cdot \left(\frac{z_{Koordinate}}{800} + 0{,}5\right) \quad \text{(Gl. 5.4)}$$

$$pitch_{digital} = 2^{12} \cdot \left(\frac{pitch_{Koordinate}}{100} + 0{,}5\right) \quad \text{(Gl. 5.5)}$$

Zum Versenden dieser Koordinaten wird zuerst ein digitaler Eingang von 0 auf 1 gesetzt, dann werden die zuvor ermittelten Spannungen nacheinander, in einem Abstand beziehungsweise mit einer Länge von einer halben Sekunde, gesendet und der digitale Eingang abschließend wieder auf 0 gesetzt. [Anhang B.3, Z. 31-36]

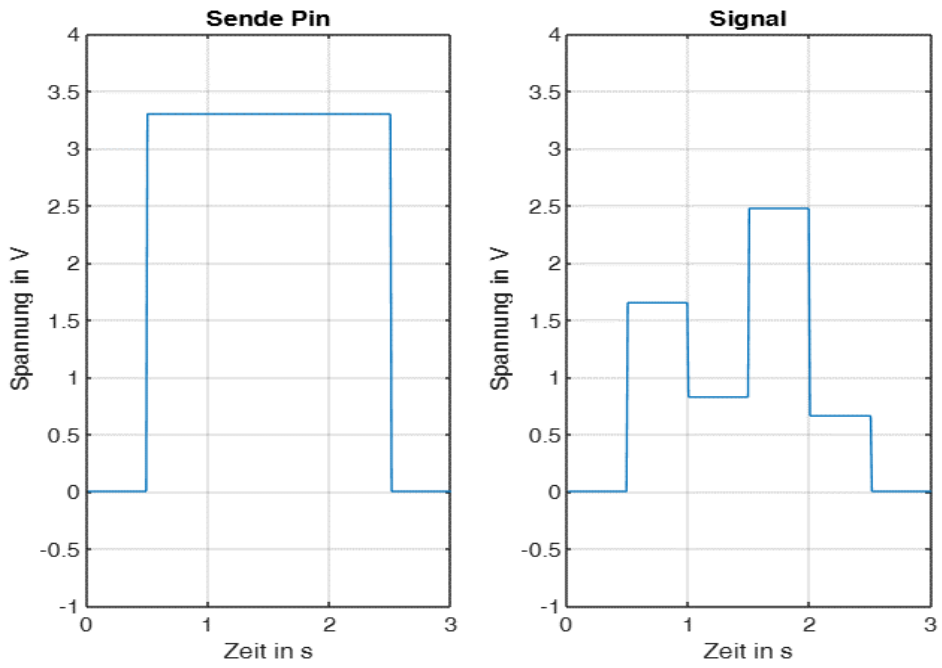


Abbildung 14: Darstellung Beispiel Signalverlauf Bewegungsbefehl [Eigene Darstellung]

Über zwei weitere Funktionen `robot.openGripper()` bzw. `robot.closeGripper()`, wird ein anderer digitaler Eingang der xArm 7 Anschlussbox geschaltet, um den Greifer des Endeffektors zu öffnen bzw. zu schließen. [Anhang B.3 Z. 41-47]

Zuletzt wird eine Funktion implementiert, mit der der Robotersteuerung mitgeteilt werden kann, welche Farbe und Form das Objekt, das gegriffen wird, aufweist. Dazu wird ähnlich wie bei der Umrechnung der Bewegungsbefehle ein Index aus einer Liste bekannter Farben (bzw. Formen) in eine Spannung konvertiert. [Anhang B.3, Z. 49-74]

5.4.4.3 Robotersteuerung – Interpretation der elektrischen Signale

Zur Robotersteuerung läuft auf einem separaten Windows-Computer ein Programm, welches mit Hilfe der visuellen Programmieroberfläche Blockly in der Software uFactory Studio des Herstellers des xArm 7 programmiert wurde.

Zum Start der Robotersteuerung wird dieser mit dem Befehl `joint motion` in die vorab empirisch definierte Startposition gefahren [Anhang C.1, 1. Block]. Danach startet eine Endlosschleife, in der laufend überprüft wird, ob Befehle in Form von elektrischen Signalen über die Anschlussbox gesendet werden. Sobald der digitale Eingang DI1 auf 1 gesetzt wird, öffnet der Endeffektor.

Wenn der digitale Eingang DI1 wieder auf 0 fällt, schließt der Endeffektor. [Anhang C.1]

Die Interpretation der Bewegungsbefehle erfolgt der Übersichtlichkeit halber in zwei voneinander getrennten Funktionen:

Die Funktion `move` überprüft laufend, ob der digitale Eingang DI0 auf 1 gesetzt wird. Sobald das passiert, misst die Software im vorher definierten Zeitabstand die Spannung, die am analogen Eingang vorliegt und rechnet diese zurück in eine, von der aktuellen Position abhängige, Relativkoordinate um. Sobald diese Koordinate über dem definierten Schwellwert von 3mm liegt, gibt die Robotersteuerung den entsprechenden Bewegungsbefehl mit dem Block `relative motion` an den Roboter. Der Verdrehwinkel des Endeffektors stellt dabei eine Ausnahme dar, weil es für die Bewegung eines einzelnen Gelenkes keinen speziellen Block gibt, bei dem eine Variable eingesetzt werden kann, sodass eine Alternative mit einer Schleife und einer definierten Drehung von 1° pro Schleifendurchgang gearbeitet wird. [Anhang C.2]

In der Funktion `sort` werden die digitalen Eingänge DI2 und DI3 laufend überprüft. Sobald einer der Eingänge von 0 auf 1 geschaltet wird, misst die Software die Spannung, die am analogen Eingang vorliegt. Je nach Spannung wird der xArm, mit dem Befehl `joint motion` in eine definierte Position gefahren. dort wird der Endeffektor über die Steuerung des RPis geöffnet und der xArm fährt erneut in seine Startposition, sodass der RPi mit dem nächsten Objekt fortfahren kann.

Um die Objekte nach Farben zu sortieren, muss der `when` Block der Formen deaktiviert werden. Um die Objekte nach deren Form zu sortieren, muss der `when` Block der Farben deaktiviert werden. [Anhang C.3]

6 Ergebnisse

Im folgenden Kapitel werden die Ergebnisse dieser Projektarbeit präsentiert, dabei wird anhand von Bilderreihen das lauffähige Programm dargestellt.

6.1 Kamerakalibrierung

Zur Kamerakalibrierung wird in der Ausgangsposition ein Bild aufgenommen und dieses mit Hilfe der OpenCV-Funktionen verarbeitet. In Abbildung 15 sind das originale Kamerabild sowie die gefundenen Ecken des Schachbrettmusters dargestellt.

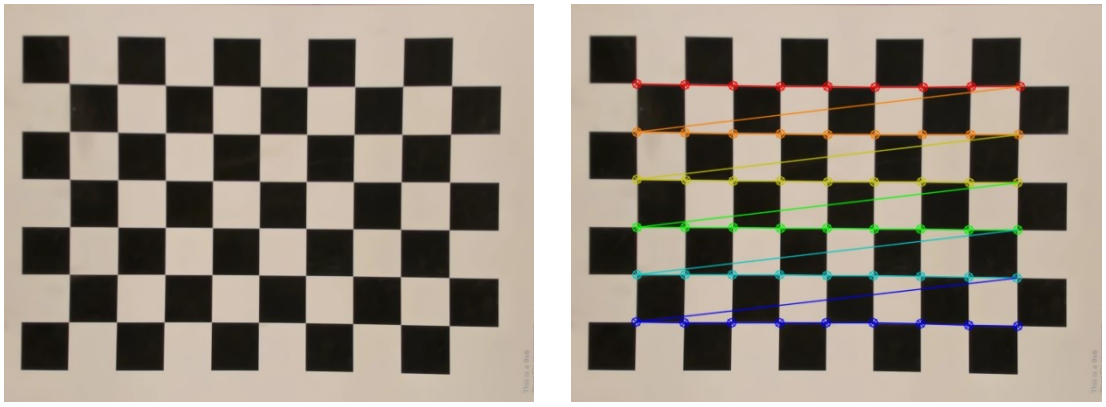


Abbildung 15: Kamerakalibrierung Links: Originalbild [Eigene Darstellung] Rechts: Gefundene Ecken [Eigene Darstellung]

Um den Umrechnungsfaktor zwischen Pixeln und Millimetern zu bestimmen, wird die erste Reihe von Punkten ausgewählt. Diese sind in der folgenden Abbildung 16 grün dargestellt. Für jeden dieser Punkte wird der Abstand (in Pixeln) zu seinem rechten, benachbarten Punkt bestimmt. Der arithmetische Mittelwert aller Abstände kann dann durch die tatsächliche Kantenlänge in Millimetern geteilt werden, um den Umrechnungsfaktor zu bestimmen.

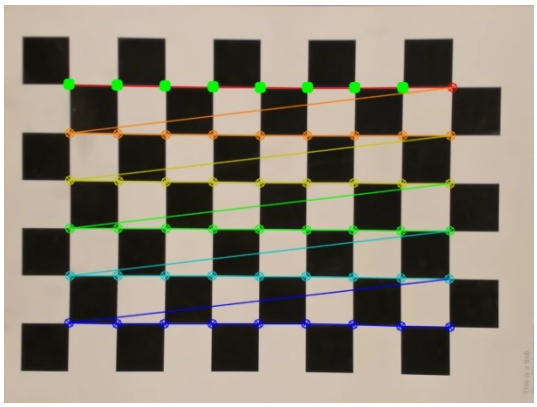


Abbildung 16: Kamerakalibrierung Schachbrettecken zur Bestimmung des Umrechnungsfaktors [Eigene Darstellung]

6.2 Farbkalibrierung

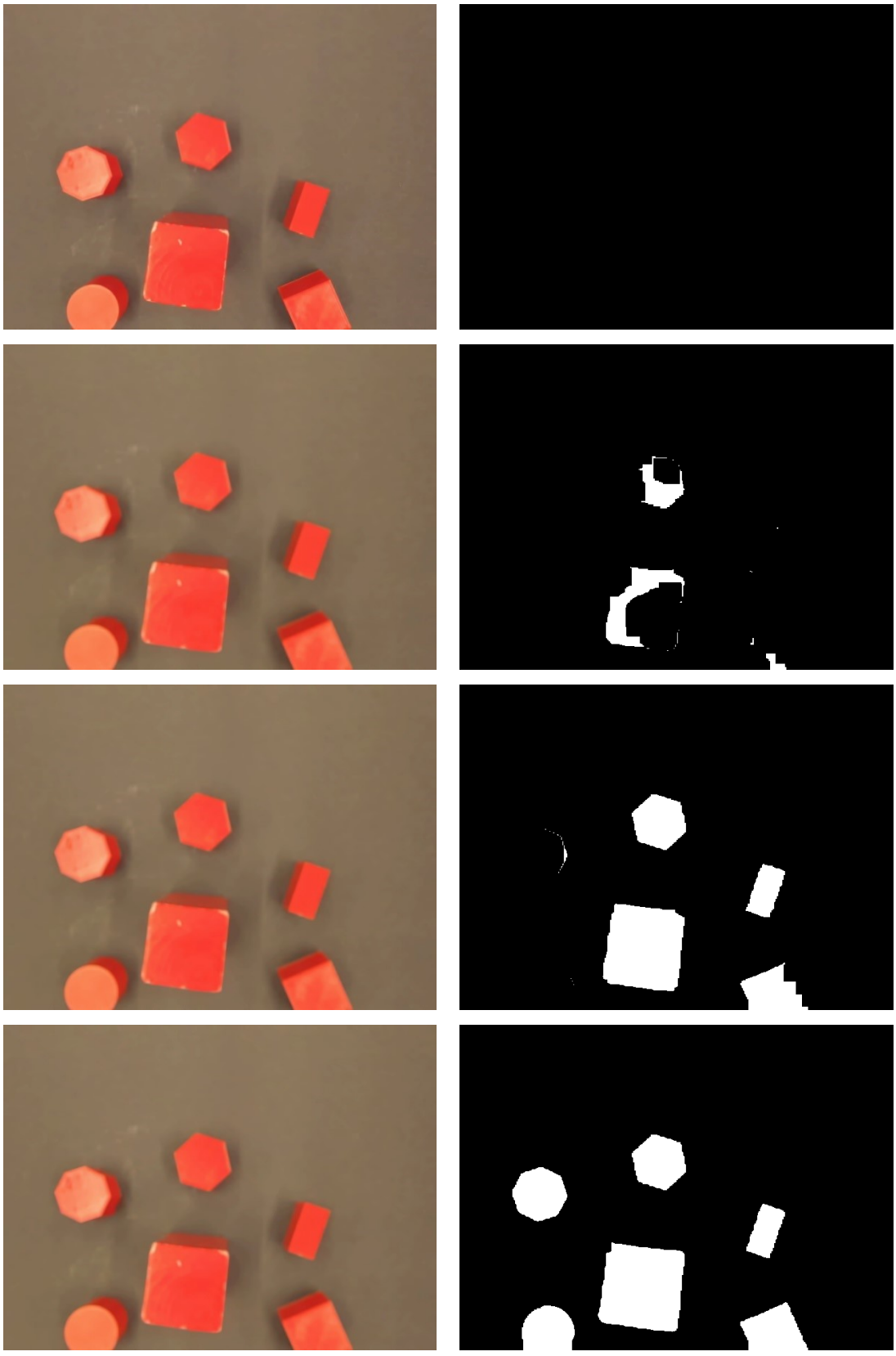
Im folgenden Beispiel wird die Farbe Rot mit der Farbkalibrierung angelernt.

Dazu wird die Farbkalibrierung aus der Konsole heraus gestartet. Es öffnen sich zwei Fenster. Auf der rechten Seite ist die binäre Farbmaske dargestellt und auf der linken Seite das originale Kamerabild. Durch das Klicken auf rote Pixel im originalen Kamerabild werden diese der Farbmaske hinzugefügt. Der Prozess wird wiederholt, bis die Benutzer*innen mit der Farbmaske zufrieden sind.

In der Praxis hat sich gezeigt, dass eine optimale Farbmaske erreicht wird, indem sowohl die Objekte der gewünschten Farbe einzeln in die Kamera gestellt werden als auch Störobjekte anderer Farben mit in das Kamerabild gestellt werden. In diesem Beispiel fällt auf, dass dabei auch orangefarbene Objekte teilweise durch die rote Farbmaske eingeschlossen werden. Diese Bereiche werden in der Regelanwendung allerdings vernachlässigt, da deren Flächeninhalt zu klein ist.

Es ließ sich feststellen, dass bessere Ergebnisse erreicht werden konnten, wenn eine schwarze Kartonpappe im Hintergrund liegt, um den Kontrast zwischen Vorder- und Hintergrund zu erhöhen und den Effekt der automatischen Belichtungskorrektur der Kamera abzuschwächen.

Diese Schritte sind chronologisch in Abbildung 17 dargestellt.



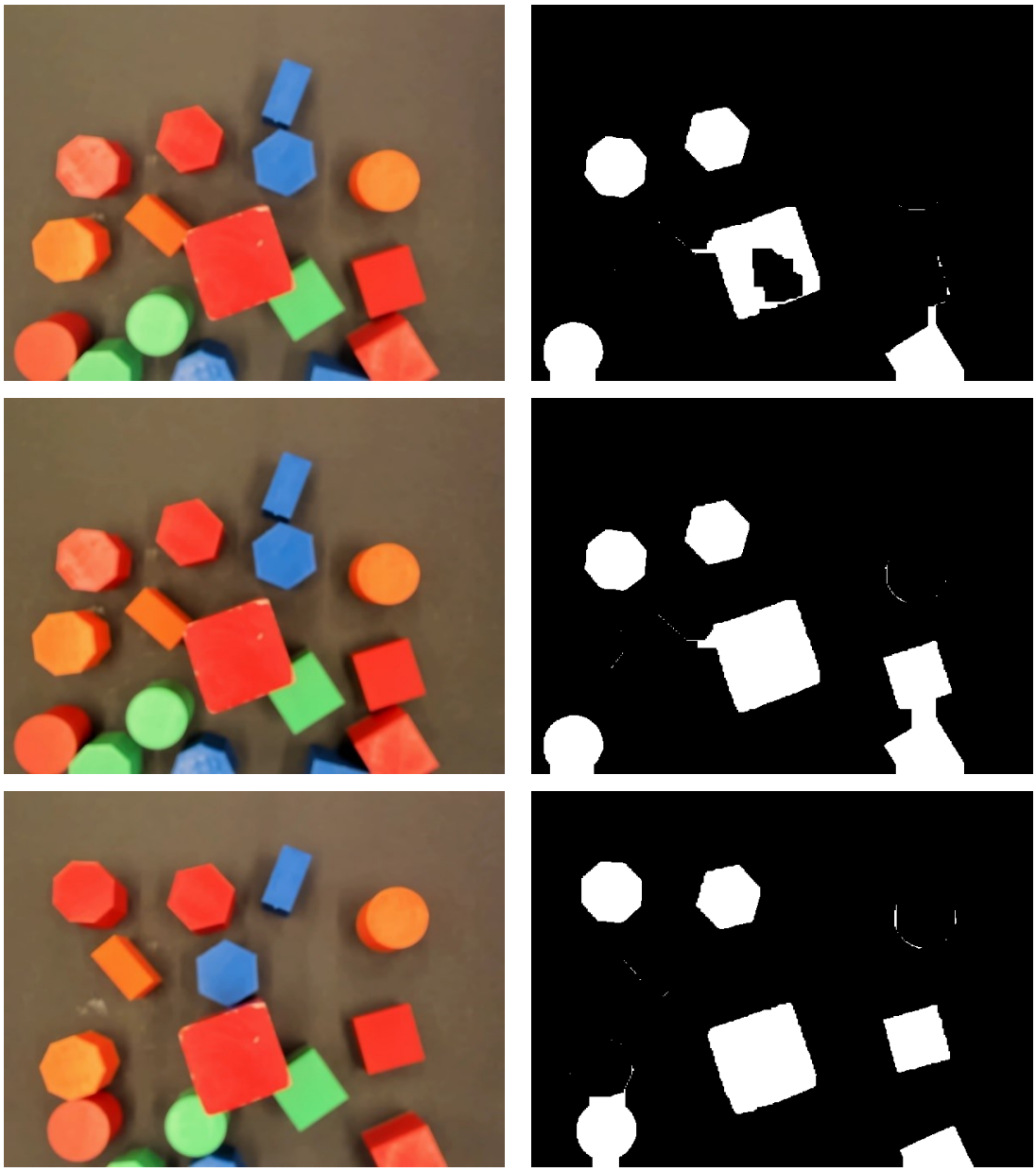


Abbildung 17: Kalibrierung Visualisierung Ablauf [Eigene Darstellung]

6.3 Regelanwendung

Nachfolgend wird anhand eines Beispiels der Programmablauf der Regelanwendung visualisiert. Dafür kommen die Bilder aus dem Programm zum Einsatz. In Abbildung 18 ist das originale Kamerabild der Übersichtsaufnahme dargestellt.

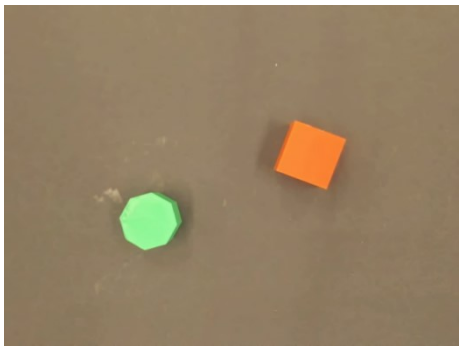


Abbildung 18: Regelanwendung Übersichtsaufnahme Original [Eigene Darstellung]

Nachfolgend werden auf dem Originalbild die Farbmasken angewendet. Dazu wurden, wie in Kapitel 6.2 Farbkalibrierung beschrieben, die Farbbereiche im Vorhinein definiert.

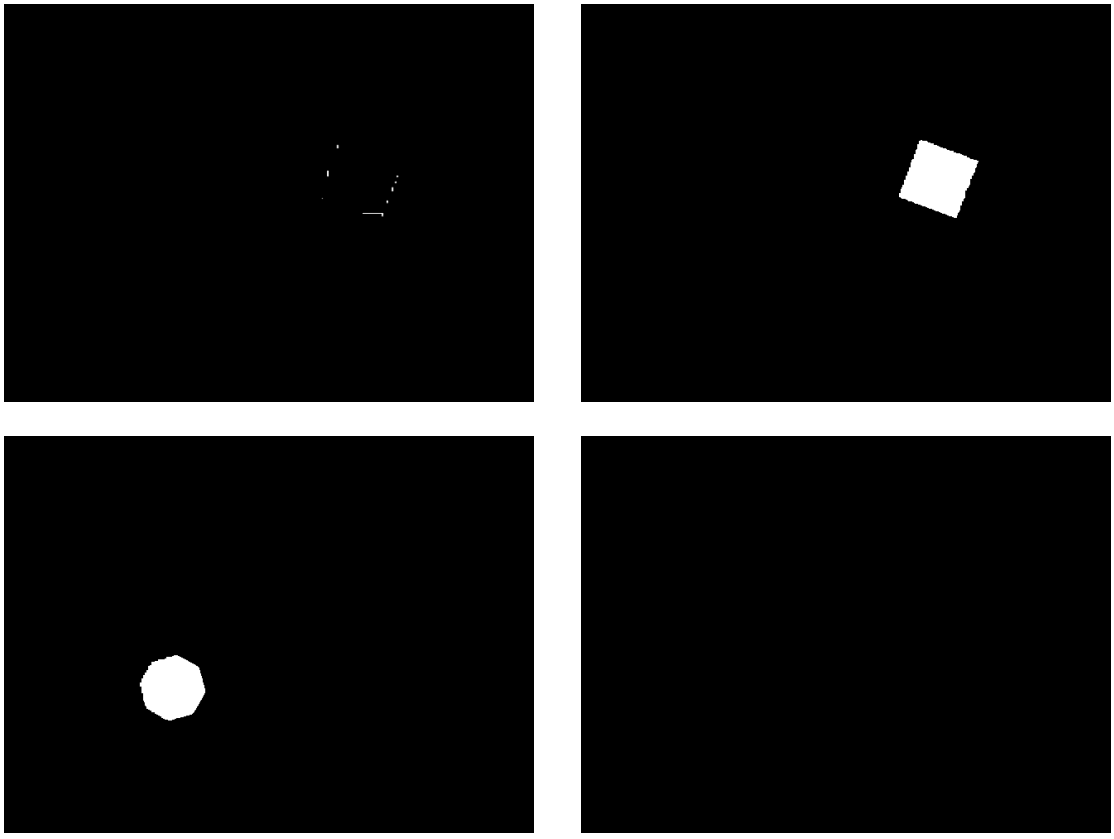


Abbildung 19: Regelanwendung Farbmasken. Oben links: Rot, Oben rechts: Orange, Unten links: Grün, Unten rechts: Blau [Eigene Darstellung]

Auf den Binärbildern auf denen große Farbbereiche bleiben werden die Konturen der Farbbereiche ermittelt. Diese sind in Abbildung 20 dargestellt.

Abbildung 20: Regelanwendung Konturen der Farbmasken [Eigene Darstellung]

Die daraufhin errechneten Mittelpunkte der Körper sind in Abbildung 21 dargestellt.

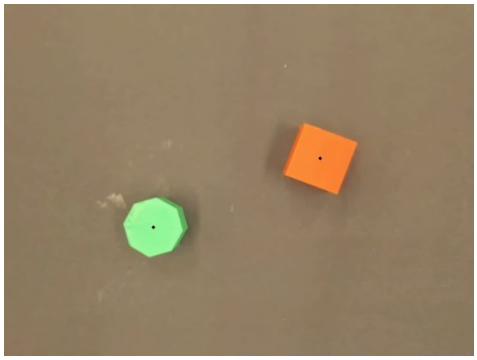


Abbildung 21: Regelanwendung Mittelpunkte im Übersichtsbild [Eigene Darstellung]

Im Anschluss daran, wird der erste Bewegungsbefehl an den xArm gesendet. Die Kamera bewegt sich über den ersten errechneten Mittelpunkt und nimmt erneut ein Bild auf. Dieses wird wieder mit Hilfe der Farbmasken in ein binäres Bild umgewandelt. Zur Kreisdetektion wird eine Kopie des Binärbildes zusätzlich weichgezeichnet, da das in der Praxis zu besseren Ergebnissen geführt hat. Die beiden Bilder sind in Abbildung 22 dargestellt.

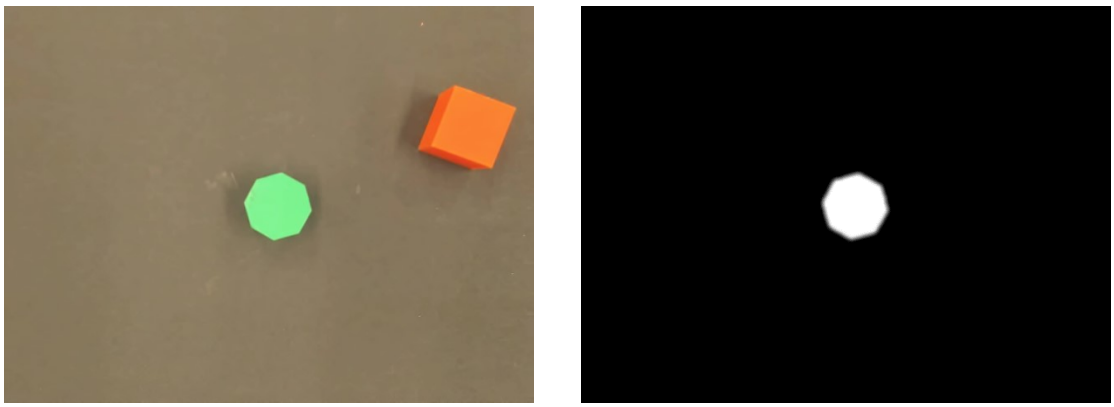


Abbildung 22: Regelanwendung Links: Detailaufnahme erster Körper, Rechts: Dazugehöriges Binärbild zur Kreisdetektion [Eigene Darstellung]

In diesem Fall wird kein Kreis gefunden, daher wird die Kontur des Farbbereiches bestimmt und der Douglas-Peucker-Algorithmus angewendet. Die Kontur ist in der nachfolgenden Abbildung 23 blau und die gefundenen Eckpunkte grün dargestellt.

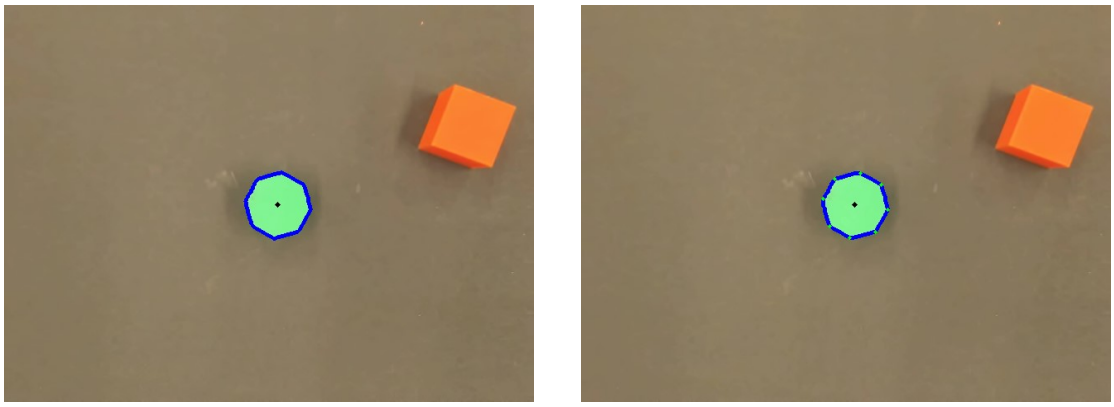


Abbildung 23: Regelanwendung Links: Detailaufnahme Kontur Körper, Rechts: Dazugehörige Eckpunkte der gefunden Kontur [Eigene Darstellung]

Daraufhin wird die Anzahl der gefundenen Eckpunkte ausgewertet. In diesem Fall findet die Bilderkennung acht Eckpunkte, sodass von einem Oktagon ausgegangen wird. Außerdem werden die beiden Eckpunkte, die am weitesten links liegen, zur Berechnung des Winkels zwischen der Objektaußenkante und der Bildvertikalen genutzt. In Abbildung 24 sind diese beiden Punkte in Grün darstellt. Außerdem wird das Ergebnis der Objekterkennung in Form eines Textes auf dem Bild ausgegeben.

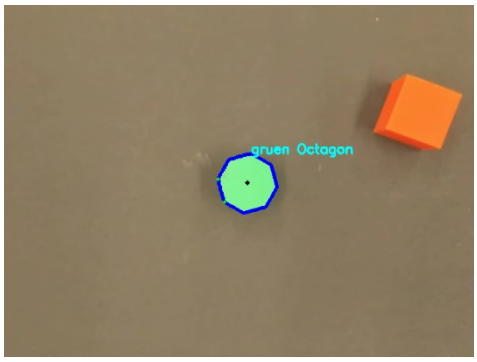


Abbildung 24: Regelanwendung Gefundene Form [Eigene Darstellung]

Danach soll die Entfernung zum Objekt bestimmt werden, dazu fährt die Kamera nach links und nimmt ein Bild auf. Dann wird dieses so zugeschnitten, dass die Bilderkennung weniger auswerten muss und der Mittelpunkt des zu greifenden Objektes weiterhin gesichert im Bild liegt. Schließlich wird mit den bereits beschriebenen Methoden der Mittelpunkt bestimmt. Das Verfahren kann in Abbildung 25 nachvollzogen werden.

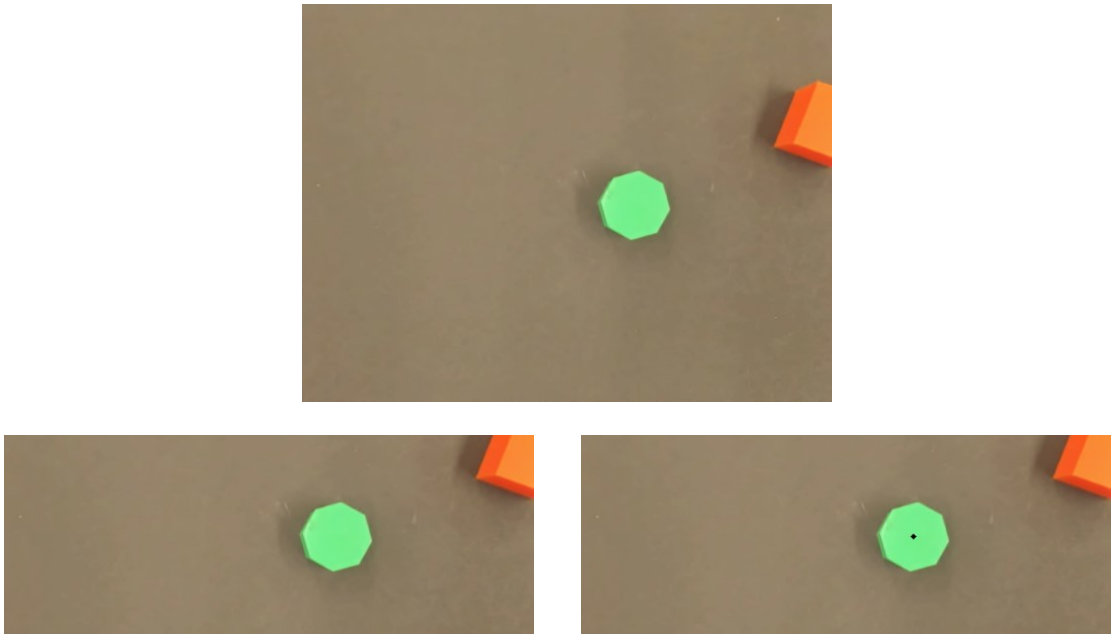


Abbildung 25: Regelanwendung Stereo Vision Links Oben: Originalbild, Unten links: Zugeschnittenes Bild, Unten rechts: Mittelpunkt des Körpers [Eigene Darstellung]

Diese Vorgehensweise wird für eine weitere Kameraposition rechts vom Objekt wiederholt. Die Schritte sind in Abbildung 26 visualisiert.

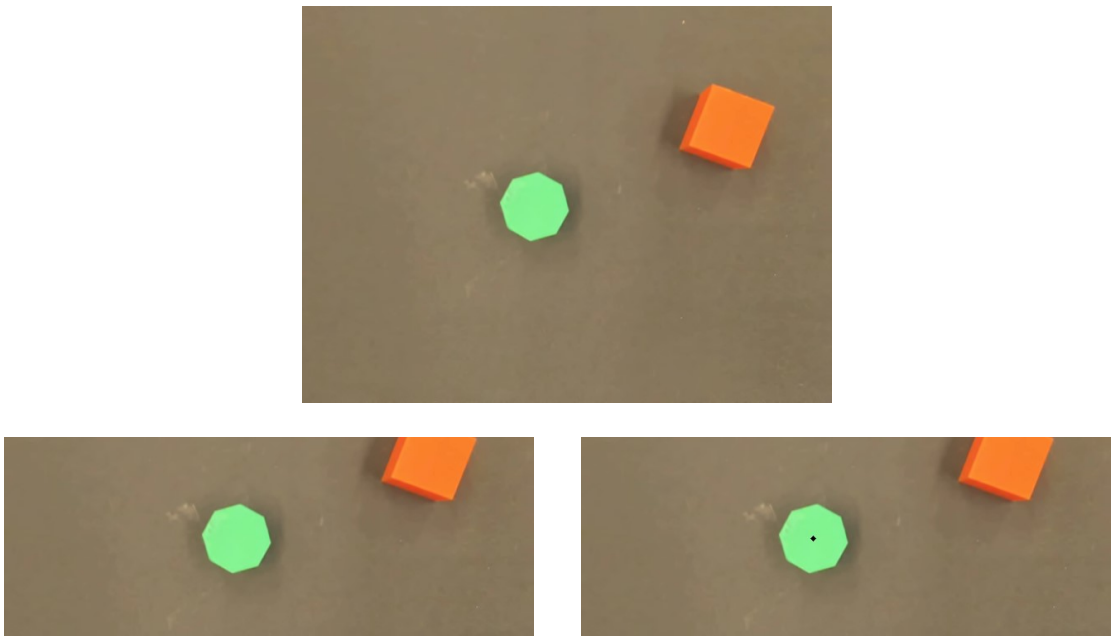


Abbildung 26: Regelanwendung Stereo Vision Rechts Oben: Originalbild, Unten links: Zugeschnittenes Bild, Unten rechts: Mittelpunkt des Körpers [Eigene Darstellung]

Mit Hilfe der beiden ermittelten Objektmittelpunkte kann die Tiefe bestimmt werden. Es liegen alle Informationen vor, um das Objekt zu greifen und auf eine bestimmte Position im Arbeitsbereich zu sortieren.

7 Fazit und Ausblick

Das Projekt zur Implementierung einer Bilderkennung hat die Ziele erreicht. Die Eigenfertigungsteile aus dem 3D-Drucker sind passgenau und zweckdienlich.

Die Bild- und Objekterkennung funktioniert präzise und wiederholgenau, wenn die Farbkalibrierung sorgfältig erfolgt. Die Anwender*innen haben die volle Kontrolle über die Farbkalibrierung und Einblick in den Prozess.

Sowohl die Übertragung der Bewegungsbefehle als auch die Interpretation dieser in der Robotersteuerung funktionieren hinreichend genau.

Trotz der Fortschritte gibt es noch Raum für Verbesserungen und Weiterentwicklungen. Die Kamerahalterung könnte weiter optimiert werden. Während des Projektes wurde die verwendete USB-Kamera gegen das jetzt verwendete, neuere Modell getauscht, weshalb die Kamerahalterung für eine andere Kamera ausgelegt worden ist.

Eine der größten Herausforderungen bleibt die Verbesserung der Datenübertragungsgeschwindigkeit und der Genauigkeit der Stereosicht. Zukünftige Projekte könnten sich darauf konzentrieren, diese Aspekte zu optimieren, um die Leistungsfähigkeit und Geschwindigkeit des Systems weiter zu steigern. Zum Beispiel könnte der Einsatz eines zweiten Digital-Analog-Wandlers die Zeit, die zur Datenübertragung benötigt wird, halbieren. Auch könnte der xArm über die ROS-Schnittstelle (Robot Operating System) direkt von einem leistungsstärkeren Computer gesteuert werden.

Ein weiterer vielversprechender Ansatz wäre der Einsatz von Industriekameras mit höherer Brennweite oder der RPi Kamera. Der größte Nachteil der USB-Kamera besteht darin, dass diese für jede Aufnahme versucht das Bild optimal zu belichten und zu fokussieren. Dies führt zu inkonsistenten Daten durch wechselnde Belichtung und Bildschärfe. Ein Austausch der Kamera könnte außerdem die Präzision der Bilderkennung weiter erhöhen und die Einsatzmöglichkeiten des Systems erweitern, da perspektivische Verzerrung aufgrund der geringen Brennweite minimiert wird.

Zukünftige Arbeiten könnten die Integration von maschinellem Lernen in die Bildverarbeitung untersuchen, um die Erkennungsfähigkeiten des Systems zu verbessern und eine größere Anzahl verschiedenartiger Objekte zuzulassen.

Student*innen, die das Praktikum zum Modul „Programmierung von Industrierobotern“ belegen, könnten mit diesem Ergebnis weiterarbeiten, um die Grundlagen der visuellen Programmieroberfläche Blockly oder der einfachen Datenübertragung und -verarbeitung zu lernen.

Diese Arbeit zeigt, wie mit einfachen Mitteln eine Bildverarbeitung und -erkennung umgesetzt werden kann. Sie bietet eine erste Grundlage, um auf den vorgestellten Konzepten aufzubauen und diese weiter zu denken.

Anhang A: CAD-Modelle der Eigenfertigungsteile

A.1 Kamerahalterung: Adapterplatte

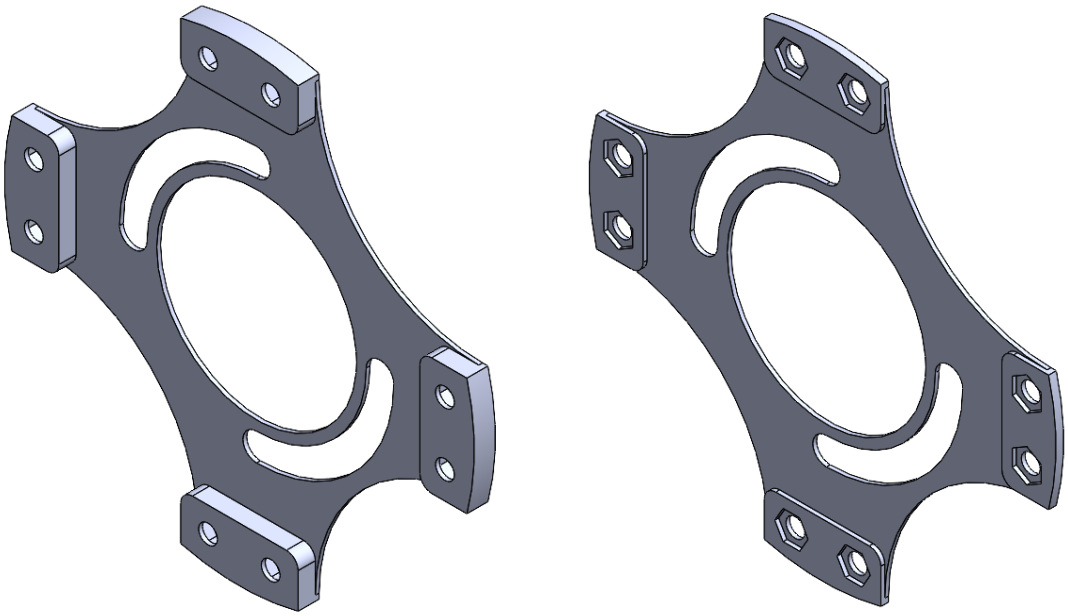


Abbildung 27: Kamerahalterung: Adapterplatte [Eigene Darstellung]

A.2 Kamerahalterung: Winkel

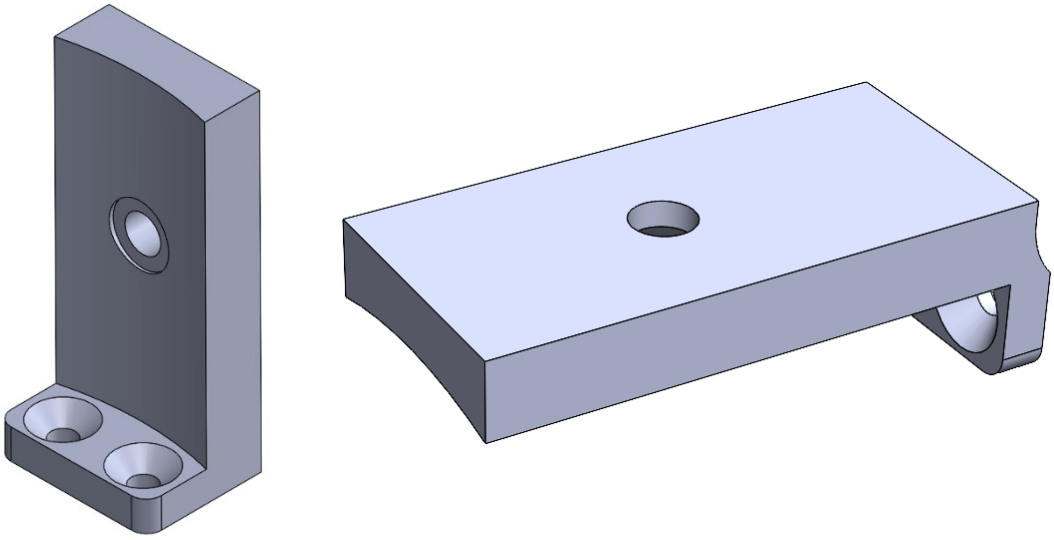


Abbildung 28: Kamerahalterung: Winkel [Eigene Darstellung]

A.3 Objekte

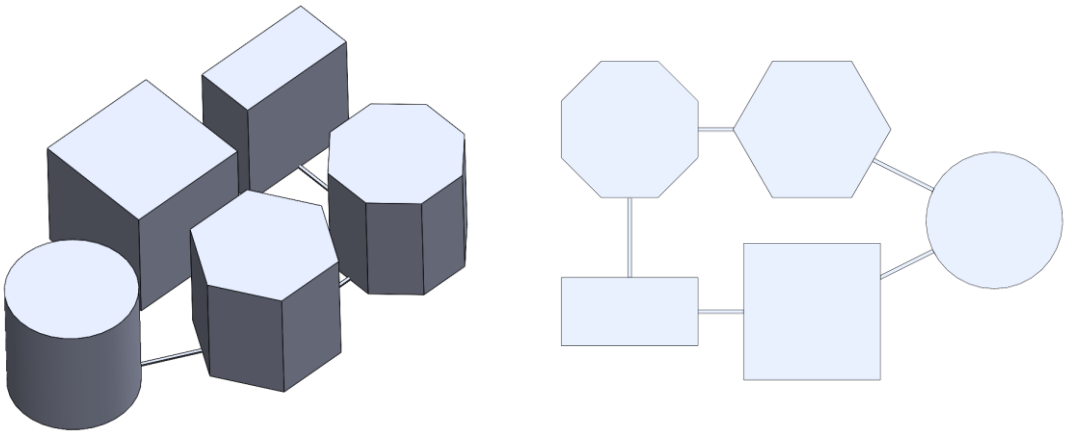


Abbildung 29: Objekte [Eigene Darstellung]

Anhang B: Python Programmcode

B.1 main.py

```
1   import cv2 as cv
2   import cameracalibration
3   import colors
4   import stereovision
5   import shapes
6   import triangulation
7   import numpy as np
8
9   import robot
10  import time
11
12  def main():
13      # Start Console
14      mode = input(f"Bitte wählen Sie den Modus in dem die
    Anwendung gestartet werden soll:"
15                   f"\n\n1\tStereo
    Calibration\n2\tFarbkalibrierung\n3\tRegelanwendung\n\n")
16      match str.lower(mode):
17          case "1":
18              print("Die Kamerakalibrierung wird gestartet...")
19              cameracalibration.calibratePX2MMfactor()
20              return
21          case "2":
22              print("Die Farbkalibrierung wird gestartet...")
23              colors.colorcalibration()
24              return
25          case "3":
26              print("Die Anwendung wird gestartet...")
27              cam = cv.VideoCapture(0)
28              _, image = cam.read()
29              cam.release()
30
31              height, width = image.shape[:2]
32              # "Farbflecken" finden
33              colorspots = colors.findcontoursincolor(image)
34
35              # Mittelpunkte der "Farbflecken" finden
36              colorspotcenters = shapes.getColorsCenter(colorspots)
37
38              for center in colorspotcenters:
39                  # Mittelpunkte markieren
40                  cv.circle(image, center, 2, (0, 0, 0), -1)
41
42              cv.imshow("Farbkleckse", image)
43              cv.waitKey(3000)  # 3s warten
44              cv.destroyAllWindows()
45
46              for center in colorspotcenters:
47                  # Über "Farbfleck" bewegen
48                  movex = cameracalibration.cvtPX2MM(center[0] -
    width / 2)
49                  movey = cameracalibration.cvtPX2MM(center[1] -
```

```
    height / 2)
50                  robot.move(movex, movey)
51                  time.sleep(3)
52
53                  res = shapes.findshapes()
54
55                  #Schnellere ungenauere Variante
56                  #movez = triangulation.find_depth(res[3], center,
    image, image, np.linalg.norm([movex, movey])/10, 0, 45)
57
58                  #Genaue, langsamere variante
59                  movez = stereovision.finddepth()
60
61                  movez = movez * 10 - 125  # Umrechnung in mm und
    125mm Kamerabstand zu greifer
62                  movez = movez * 0.93  # Empirischer
    Korrekturfaktor
63
64                  robot.openGripper()
65                  robot.move(0, -80, -movez, res[0])  # Runter zum
    Objekt
66                  time.sleep(5)
67                  robot.closeGripper()
68                  robot.move(0, 0, movez, -res[0])  # Gripper
    gerade machen und hoch
69                  robot.sendcolorandshape(res[1], res[2])  #
    Farb/Form an Steuerung
70                  time.sleep(3)
71                  robot.openGripper()
72                  time.sleep(5)
73              return
74          case _:
75              print("Keine gültige Eingabe gefunden!")
76              return
77
78  if __name__ == "__main__":
79      try:
80          main()
81      except Exception as e:
82          print(e)
```

B.2 cameracalibration.py

```
1   import cv2 as cv
2   import numpy as np
3
4   def calibratePX2MMfactor():
5       chessboardSize = (9, 6)
6       chessboardSquareSize = 25  # in mm
7       terminationCriteria = (cv.TERM_CRITERIA_EPS +
    cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)
8
9       cam = cv.VideoCapture(0)
10      _, image = cam.read()
11      cam.release()
12
13      grayIMG = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
14      ret, corners = cv.findChessboardCorners(grayIMG,
    chessboardSize, None)
15      corners = cv.cornerSubPix(grayIMG, corners, (11, 11), (-1, -
```

```
    1), terminationCriteria)
16      cv.drawChessboardCorners(image, chessboardSize, corners, ret)
17
18      dists = []
19      for i in range(chessboardSize[0]-1):
20          dist = np.linalg.norm(corners[i+1][0] - corners[i][0])
21          dists.append(dist)
22          cv.circle(image, (int(corners[i][0][0]),
    int(corners[i][0][1])), 2, (0, 255, 0), 10)
23
24      f = chessboardSquareSize / np.mean(dists)
25
26      # Faktor speichern
27      cv_file = cv.FileStorage('calibration.xml',
    cv.FILE_STORAGE_WRITE)
28
29      cv_file.write('calibrationFactor', f)
30
31      cv_file.release()
32
33  def cvtPX2MM(px):
34      cv_file = cv.FileStorage()
35      cv_file.open('calibration.xml', cv.FileStorage_READ)
36
37      f = cv_file.getNode('calibrationFactor').real()
38
39      return px * f
40
41  def cvtMM2PX(mm):
42      cv_file = cv.FileStorage()
43      cv_file.open('calibration.xml', cv.FileStorage_READ)
44
45      f = cv_file.getNode('calibrationFactor').real()
46
47      return mm / f
```

B.3 colors.py

```
 1  import cv2 as cv
 2  import numpy as np
 3  import xml.etree.ElementTree as ET
 4
 5  h_l, s_l, v_l, h_h, s_h, v_h = 0, 0, 0, 0, 0, 0
 6
 7
 8  def colorcalibration():
 9      print("Farbkalibrierung erfolgreich gestartet!")
10      cam = cv.VideoCapture(0)
11
12      while cam.isOpened():
13          colorList = []
14          global h_l, s_l, v_l, h_h, s_h, v_h
15
16          filetree = ET.parse('./colors.xml')
17          fileroot = filetree.getroot()
18
19          # Bekannte Farben einlesen
20          i = 1
21          for element in fileroot:
22              colorList.append(f"{i} {element.tag}")
23              i += 1
```

```
24
25          inp = input(f"Folgende Farben liegen vor:\n" +
    '\n'.join(colorList) +
26                      "\n\nWählen Sie eine der Farben aus um sie
    zu bearbeiten oder zu löschen\n"
27                      "Wenn eine neue Farbe angelegt werden soll
    drücken Sie \'n\'\n"
28                      "Zum Beenden der Kalibrierung drücken Sie
    \'c\'\n\n")
29
30          if str.lower(inp) == 'c':
31              break
32          elif str.lower(inp).isnumeric():
33              # Eine Farbe wurde gewählt
34              inp2 = input(
35                  f"Sie haben gewählt: {colorList[int(inp) -
    1]}\n\nWählen Sie eine Option:\n1\tLöschen\n2\tAnzeigen\n\n")
36              selectedElement =
    fileroot.find(str.split(colorList[int(inp) - 1], " ")[1])
37              if inp2 == "1":
38                  # Farbe löschen
39                  fileroot.remove(selectedElement)
40                  filestr = ET.tostring(fileroot,
    encoding='utf8').decode('utf8')
41                  filestr = filestr.replace('><', '>\n<')
42                  with open("./colors.xml", "w") as file:
43                      file.write(filestr)
44                  print("Löschen erfolgreich!")
45                  continue
46              elif inp2 == "2":
47                  # Farbe anzeigen
48                  print(f"{colorList[int(inp) - 1]} wird
    angezeigt.")
49                  while True:
50                      res, image = cam.read()
51                      assert res
52                      imgHSV = cv.cvtColor(image,
    cv.COLOR_BGR2HSV)
53
54                      # Obere und Untere Grenzen extrahieren
55                      color_lower = np.array(list(map(int,
    selectedElement.find('lower').text.split(', '))))
56                      color_upper = np.array(list(map(int,
    selectedElement.find('upper').text.split(', '))))
57
58                      # Farbmaske anwenden und optimieren
59                      mask = cv.inRange(imgHSV, color_lower,
    color_upper)
60                      kernel = np.ones((25, 25), np.uint8)
61                      mask = cv.morphologyEx(mask, cv.MORPH_CLOSE,
    kernel, iterations=1)
62
63                      cv.imshow("image", image)
64                      cv.imshow(f"{colorList[int(inp) - 1]} mask",
    mask)
65
66                      k = cv.waitKey(1)
67                      if k == ord('c'):
68                          cv.destroyAllWindows()
69                          break
70                  continue
71              else:
```

```
72                  print("Keine gültige Eingabe!")
73                  continue
74
75          elif str.lower(inp) == 'n':
76              # Neue Farbe anlegen
77              colorPixel = []
78
79              def coloratpx(event, x, y, flags, param):
80                  global h_l, s_l, v_l, h_h, s_h, v_h
81                  if event == cv.EVENT_LBUTTONDOWN:
82                      h_l, s_l, v_l = 256, 256, 256
83                      h_h, s_h, v_h = -256, -256, -256
84                      imgHSV = cv.cvtColor(param,
    cv.COLOR_BGR2HSV)
85                      colorPixel.append(imgHSV[y, x])
86
87                      for c in colorPixel:
88                          h = c[0]
89                          s = c[1]
90                          v = c[2]
91                          if h < h_l:
92                              h_l = h
93                          if h > h_h:
94                              h_h = h
95                          if s < s_l:
96                              s_l = s
97                          if s > s_h:
98                              s_h = s
99                          if v < v_l:
100                             v_l = v
101                         if v > v_h:
102                             v_h = v
103                     s_l -= 15
104                     cv.add(np.uint8([s_h]), np.uint8([15]))
105                     v_l -= 15
106                     cv.add(np.uint8([v_h]), np.uint8([15]))
107
108             #i = 0
109             while True:
110                 res, image = cam.read()
111                 assert res
112                 cv.imshow("image", image)
113                 # Beim Klicken coloratpx aufrufen
114                 cv.setMouseCallback("image", coloratpx, image)
115
116                 imgHSV = cv.cvtColor(image, cv.COLOR_BGR2HSV)
117                 if h_h - h_l > 170:  # Rot Ausnahme
118                     mask = cv.inRange(imgHSV, np.array([0, s_l,
    v_l]), np.array([5, s_h, v_h]))
119                 else:
120                     mask = cv.inRange(imgHSV, np.array([h_l,
    s_l, v_l]), np.array([h_h, s_h, v_h]))
121                 kernel = np.ones((25, 25), np.uint8)
122                 mask = cv.morphologyEx(mask, cv.MORPH_CLOSE,
    kernel, iterations=1)
123                 cv.imshow("mask", mask)
124                 #cv.imwrite(f"./colorspics/{i}mask.png", mask)
125                 #cv.imwrite(f"./colorspics/{i}.png", image)
126                 #i=i+1
127
128                 k = cv.waitKey(1)
129                 if k == ord('c'):
```

```
130                     cv.destroyAllWindows()
131                     print(f"{h_l}, {s_l}, {v_l}")
132                     colorname = input(
133                         "Soll die Farbe gespeichert werden?
    Falls nein schreiben Sie 'n'. Falls ja geben Sie den Namen der
    Farbe an:\n")
134
135                     if colorname == 'n':
136                         colorPixel = []
137                         break
138
139                     filetree = ET.parse('./colors.xml')
140                     fileroot = filetree.getroot()
141
142                     if fileroot.find(colorname) is not None:
143                         inp = input("Diese Farbe ist bereits
    gespeichert, soll die alte Farbe ueberschrieben werden? (Y/N) ")
144                         if inp.lower() != "y":
145                             break
146
    fileroot.remove(fileroot.find(colorname))
147
148                     colorelement = ET.SubElement(fileroot,
    colorname)
149                     ET.SubElement(colorelement, 'lower').text =
    f"{h_l}, {s_l}, {v_l}"
150                     ET.SubElement(colorelement, 'upper').text =
    f"{h_h}, {s_h}, {v_h}"
151                     filestr = ET.tostring(fileroot,
    encoding='utf8').decode('utf8')
152                     filestr = filestr.replace('><', '>\n<')
153                     with open("./colors.xml", "w") as file:
154                         file.write(filestr)
155
156                     colorPixel = []
157                     h_l, s_l, v_l = 256, 256, 256
158                     h_h, s_h, v_h = -1, -1, -1
159                     break
160             continue
161
162 # Gibt Farbmasken zu einem Bild zurück
163 def findcontoursincolor(image):
164     resultList = []
165     imageHSV = cv.cvtColor(image, cv.COLOR_BGR2HSV)
166
167     colors = []
168     filetree = ET.parse('./colors.xml')
169     fileroot = filetree.getroot()
170
171     for color in fileroot:
172         colorName = color.tag
173         color_lower = np.array(list(map(int,
    color.find('lower').text.split(', '))))
174         color_upper = np.array(list(map(int,
    color.find('upper').text.split(', '))))
175         colors.append([colorName, color_lower, color_upper])
176
177         mask = cv.inRange(imageHSV, color_lower, color_upper)
178         kernel = np.ones((25, 25), np.uint8)
179         mask = cv.morphologyEx(mask, cv.MORPH_CLOSE, kernel,
    iterations=1)
180         resultList.append([colorName, mask])
```

```
181         return resultList
```

B.4 shapes.py

```
1   import cv2 as cv
2   import numpy as np
3   import colors
4   import robot
5   import cameracalibration
6
7   def getColorsCenter(masks):
8       centers = []
9       for colorspot in masks:
10          colorMask = colorspot[1]
11          image = cv.cvtColor(colorMask, cv.COLOR_GRAY2BGR)
12          # Konturen finden
13          contours, _ = cv.findContours(colorMask, cv.RETR_TREE,
    cv.CHAIN_APPROX_NONE)
14          for contour in contours:
15              M = cv.moments(contour)  # mech. Momente
16              if M['m00'] > 2000:
17                  cv.drawContours(image, contour, -1, (255, 0, 0),
    3)
18                  cx = int(M['m10'] / M['m00'])  #
    Mittelpunktberechnung
19                  cy = int(M['m01'] / M['m00'])
20                  centers.append([cx, cy])
21              else:
22                  # Kontur zu klein -> nicht beachten
23                  continue
24      return centers
25
26
27  def findshapes():
28      cam = cv.VideoCapture(0)
29      while True:
30          result, image = cam.read()
31          cam.release()
32          height, width = image.shape[:2]
33          cx, cy, pitch = 0, 0, 0
34          resultList = colors.findcontoursincolor(image)
35          # Initialisierung Rückgabewerte
36          foundcolor = ""
37          foundshape = ""
38          foundcenter = []
39
40          for object in resultList:
41              color = object[0]
42              colormask = object[1]
43              cv.imshow("mask", colormask)
44              cv.waitKey(1)
45              # Konturen finden
46              contours, _ = cv.findContours(colormask,
    cv.RETR_TREE, cv.CHAIN_APPROX_NONE)
47              for contour in contours:
48                  M = cv.moments(contour)  # mech. Momente
49                  if M['m00'] > 2000:
50                      cx = int(M['m10'] / M['m00'])  #
    Mittelpunktberechnung
51                      cy = int(M['m01'] / M['m00'])
52                      if np.abs(cy-height/2) > 50 or np.abs(cx-
```

```
    width/2) > 50:
53                          # Erwartung: Mittelpunkt liegt nah an
    Bildmitte n.e.
54                          continue
55                      foundcenter = [cx, cy]
56                      cv.circle(image, (cx, cy), 1, (0, 0, 0), 4)
    # Mittelpunkt markieren
57                      cv.drawContours(image, contour, -1, (255, 0,
    0), 3)
58                  else:
59                      # Zu klein -> Nicht beachten
60                      continue
61
62                  # Kreis Detection
63                  binarydetectionmapcircle = cv.blur(colormask,
    (7, 7))
64                  rows = binarydetectionmapcircle.shape[0]
65                  circles =
    cv.HoughCircles(binarydetectionmapcircle, cv.HOUGH_GRADIENT,
    1.1, rows / 8,
66                                                  param1=50, param2=40,
67                                                  minRadius=10,
    maxRadius=300)
68
69                  center = (0, 0)
70                  radius = 0
71
72                  if circles is not None:
73                      # Kreis gefunden
74                      foundcolor = color
75                      circles = np.uint16(np.around(circles))
76                      for i in circles[0, :]:
77                          center = (i[0], i[1])
78                          if np.abs(i[0]-width/2) > 50 or
    np.abs(i[1]-height/2) > 50:
79                              # Erwartung: Mittelpunkt liegt nah
    an Bildmitte n.e.
80                              continue
81                          radius = i[2]
82                          cv.circle(image, center, radius, (255,
    0, 255), 3)
83                          # Auf Bild ergebnis plotten
84                          xn = np.uint16(i[0] + np.round(1.1 *
    i[2]))
85                          cv.putText(image, color + ' Circle',
    (xn, i[1]), cv.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)
86                          foundshape = "circle"
87
88                  colormaskBGR = cv.cvtColor(colormask,
    cv.COLOR_GRAY2BGR)
89                  colormaskBGR = cv.drawContours(colormaskBGR,
    contour, -1, (0, 255, 0), 6)
90                  cv.imshow(color + " mask", colormaskBGR)
91
92                  # Polygon Detection wenn kein kreis da ist oder
    alle kreise zu weit entfernt sind
93                  if circles is None or \
94                          (circles is not None and (center[0] +
    radius < cx > center[0] - radius)
95                           or (center[0] + radius > cx < center[0]
    - radius)):
96
```

```
97                      # Punkte approximieren
98                      approx = cv.approxPolyDP(contour, 0.025 *
    cv.arcLength(contour, True), True)
99
100                     x1, y1 = contour[0][0]
101                     if len(approx) == 4:
102                         # Viereck
103                         # Rechtwinkligkeit prüfen
104                         u = approx[0, 0] - approx[1, 0]
105                         v = approx[1, 0] - approx[2, 0]
106
107                         alpha = np.arccos((u[0] * v[0] + u[1] *
    v[1]) / (
108                                           (np.sqrt(np.square(u[0]) +
    np.square(u[1]))) * (
109                                               np.sqrt(np.square(v[0]) +
    np.square(v[1]))))
110
111                         if np.abs(alpha - (np.pi / 2)) < 0.1:  #
    90 Grad zwischen den Parallelen
112                             x, y, w, h =
    cv.boundingRect(contour)
113                             ratio = float(w) / h
114                             # Seitenverhältnis prüfen
115                             if 0.93 <= ratio <= 1.07:
116                                 image = cv.drawContours(image,
    [contour], -1, (0, 255, 255), 3)
117                                 cv.putText(image, color + '
    Square', (x1, y1), cv.FONT_HERSHEY_SIMPLEX, 0.6,
118                                            (255, 255, 0), 2)
119                                 foundshape = "square"
120                             else:
121                                 cv.putText(image, color + '
    Rectangle', (x1, y1), cv.FONT_HERSHEY_SIMPLEX, 0.6,
122                                            (0, 255, 0), 2)
123                                 image = cv.drawContours(image,
    [contour], -1, (0, 255, 0), 3)
124                                 foundshape = "rectangle"
125
126                     elif len(approx) == 6:
127                         # Hexagon
128                         image = cv.drawContours(image,
    [contour], -1, (255, 255, 0), 3)
129                         cv.putText(image, color + ' Hexagon',
    (x1, y1), cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)
130                         foundshape = "hexagon"
131                     elif len(approx) == 8:
132                         # Oktagon
133                         image = cv.drawContours(image,
    [contour], -1, (255, 0, 0), 3)
134                         cv.putText(image, color + ' Octagon',
    (x1, y1), cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)
135                         foundshape = "octagon"
136
137                     # Winkel zur Vertikalen berechnen
138                     # Punkte nach x-Koordinate sortieren
139                     sorted_points = sorted(approx, key=lambda x:
    x[0, 0])
140                     mostleft_points = sorted_points[:2]
141                     p1 = mostleft_points[0][0]
142                     p2 = mostleft_points[1][0]
143                     cv.circle(image, p1, 2, (0,255,0), -1)
```

```
144                     cv.circle(image, p2, 2, (0,255,0), -1)
145                     dx = p1[0] - p2[0]
146                     dy = p1[1] - p2[1]
147                     beta = (np.arctan2(np.abs(dx), -
    dy)/np.pi*180)
148                     # Winkel in [-45 45] für Gripper
    konvertieren
149                     if beta > 90:
150                         pitch = 180 - beta
151                     elif beta > 45:
152                         pitch = 90 - beta
153                     else:
154                         pitch = -beta
155                     foundcolor = color
156
157         cv.imshow("original image", image)
158         cv.imwrite("./foundshape.png", image)
159         cv.waitKey(3000)
160         cv.destroyAllWindows()
161
162         return [pitch, foundcolor, foundshape, foundcenter]
```

B.5 stereovision.py

```
1   import cv2 as cv
2   import colors
3   import cameracalibration as cc
4   import triangulation
5   import robot
6   import time
7   import shapes
8   import numpy as np
9
10  def finddepth():
11
12      # cam.set(28, 30)
13
14      B = 5             # Distance between cameras [cm]
15      f = 4.5           # Focal length [mm]
16      alpha = 45        # FOV in horisontal plane [°]
17
18      # Nach Links bewegen
19      robot.move(x=-(B*5))
20      time.sleep(1)
21      cam = cv.VideoCapture(0)
22      retL, frame_left = cam.read()
23      cam.release()
24      time.sleep(1)
25
26      # Nach Rechts bewegen
27      robot.move(x=(B*10))
28      time.sleep(1)
29      cam = cv.VideoCapture(0)
30      retR, frame_right = cam.read()
31      cam.release()
32      time.sleep(1)
33
34      # Zurück in die Mitte
35      robot.move(x=-(B*5))
36
```

```
37      height, width = frame_left.shape[:2]
38
39      # Bilder zuschneiden
40      frame_left = frame_left[int(height/4):int(height*3/4), :]
41      frame_right = frame_right[int(height/4):int(height*3/4), :]
42      #cv.imwrite("./stereorightcrop.png", frame_right)
43
44      # Mittelpunkte finden
45      colorspotsL = colors.findcontoursincolor(frame_left)
46      colorspotcentersL = shapes.getColorsCenter(colorspotsL)
47      # Nach Distanz zum erwarteten Mittelpunkt sortieren ->
    Erster ist der gesuchte
48      colorspotcentersL = sorted(colorspotcentersL, key=lambda x:
    np.abs(x[0]-width/2) - cc.cvtMM2PX(B*5))
49      cv.circle(frame_left, colorspotcentersL[0], 1, (0, 0, 0), 3)
50
51      colorspotsR = colors.findcontoursincolor(frame_right)
52      colorspotcentersR = shapes.getColorsCenter(colorspotsR)
53      # Nach Distanz zum erwarteten Mittelpunkt sortieren ->
    Erster ist der gesuchte
54      colorspotcentersR = sorted(colorspotcentersR, key=lambda x:
    np.abs(x[0]-width/2) + cc.cvtMM2PX(B*5))
55      cv.circle(frame_right, colorspotcentersR[0], 1, (0, 0, 0),
    3)
56
57      cv.imshow("left eye", frame_left)
58      cv.imshow("right eye", frame_right)
59      cv.waitKey(3000)
60      cv.destroyAllWindows()
61
62      depth = triangulation.find_depth(colorspotcentersR[0],
    colorspotcentersL[0], frame_right, frame_left, B, f, alpha)
63
64      return depth
```

B.6 triangulation.py

```
1   import numpy as np
2
3   def find_depth(right_point, left_point, frame_right, frame_left,
    baseline, f, alpha):
4
5       height_right, width_right, _ = frame_right.shape
6       height_left, width_left, _ = frame_left.shape
7
8       # Winkel in Pixelbrennweite umrechnen
9       f_pixel = (width_right * 0.5) / np.tan(alpha * 0.5 * np.pi /
    180)
10
11      x_right = right_point[0]
12      x_left = left_point[0]
13
14      disparity = np.linalg.norm(np.array(left_point) -
    np.array(right_point))
15
16      zDepth = (baseline * f_pixel) / disparity
17
18      return abs(zDepth)
```

B.7 robot.py

```
1   import RPi.GPIO as GPIO
2   import time
3   import board
4   import busio
5   import adafruit_mcp4725
6
7   i2c = busio.I2C(board.SCL, board.SDA)
8   dac = adafruit_mcp4725.MCP4725(i2c, address=0x60)
9
10  statusPin = 23
11  gripperPin = 24
12  colorPin = 20
13  shapePin = 21
14  GPIO.setmode(GPIO.BCM)
15  GPIO.setup(statusPin, GPIO.OUT)
16  GPIO.setup(gripperPin, GPIO.OUT)
17  GPIO.setup(colorPin, GPIO.OUT)
18  GPIO.setup(shapePin, GPIO.OUT)
19  GPIO.output(statusPin, GPIO.LOW)
20  GPIO.output(gripperPin, GPIO.LOW)
21  GPIO.output(colorPin, GPIO.LOW)
22  GPIO.output(shapePin, GPIO.LOW)
23
24  def move(x = 0, y = 0, z = 0, pitch = 0):
25      xpwm = x/4+50
26      ypwm = y/4+50
27      zpwm = z/8+50
28      ppwm = pitch + 50
29      pwmsignals = [xpwm, ypwm, zpwm, ppwm]
30
31      for i in range(4):
32          GPIO.output(statusPin, GPIO.HIGH) #High -> Du kriegst
    gleich Koordinaten
33          dac_val = int(2 ** 12 * pwmsignals[i]/10)
34          dac.value = dac_val
35          time.sleep(0.5)
36      GPIO.output(statusPin, GPIO.LOW)
37
38      time.sleep(1)
39      return
40
41  def openGripper():
42      GPIO.output(gripperPin, GPIO.HIGH)
43      time.sleep(5)
44
45  def closeGripper():
46      GPIO.output(gripperPin, GPIO.LOW)
47      time.sleep(5)
48
49  def sendcolorandshape(color, shape):
50      colors = ["rot", "gruen", "blau", "orange"]
51      shapes = ["square", "rectangle", "circle", "hexagon",
    "octagon"]
52
53      GPIO.output(colorPin, GPIO.HIGH)
54      if color not in colors:
55          color = -1
56          dac.value = 0
57      else:
58          color = colors.index(color)
```

```
59              dac_val = int(2 ** 12 * (5 + color))
60              dac.value = dac_val
61          time.sleep(1)
62          GPIO.output(colorPin, GPIO.LOW)
63
64          GPIO.output(shapePin, GPIO.HIGH)
65          if shape not in shapes:
66              shape = -1
67              dac.value = 0
68          else:
69              shape = shapes.index(shape)
70              dac_val = int(2 ** 12 * (5 + shape))
71              dac.value = dac_val
72          time.sleep(1)
73          GPIO.output(shapePin, GPIO.LOW)
```

Anhang C: Blockly Code zur Robotersteuerung

C.1 Programmablauf

Abbildung 30: Blockly Code zur Robotersteuerung – Programmablauf [Eigene Darstellung]

C.2 Funktion move

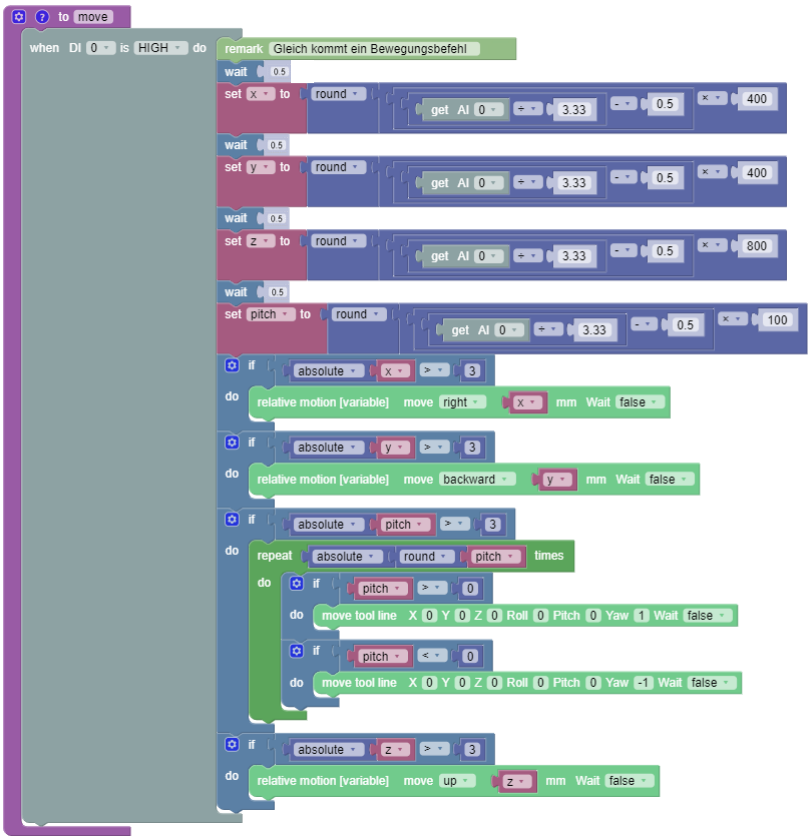


Abbildung 31: Blockly Code zur Robotersteuerung – Funktion move [Eigene Darstellung]

C.3 Funktion sort

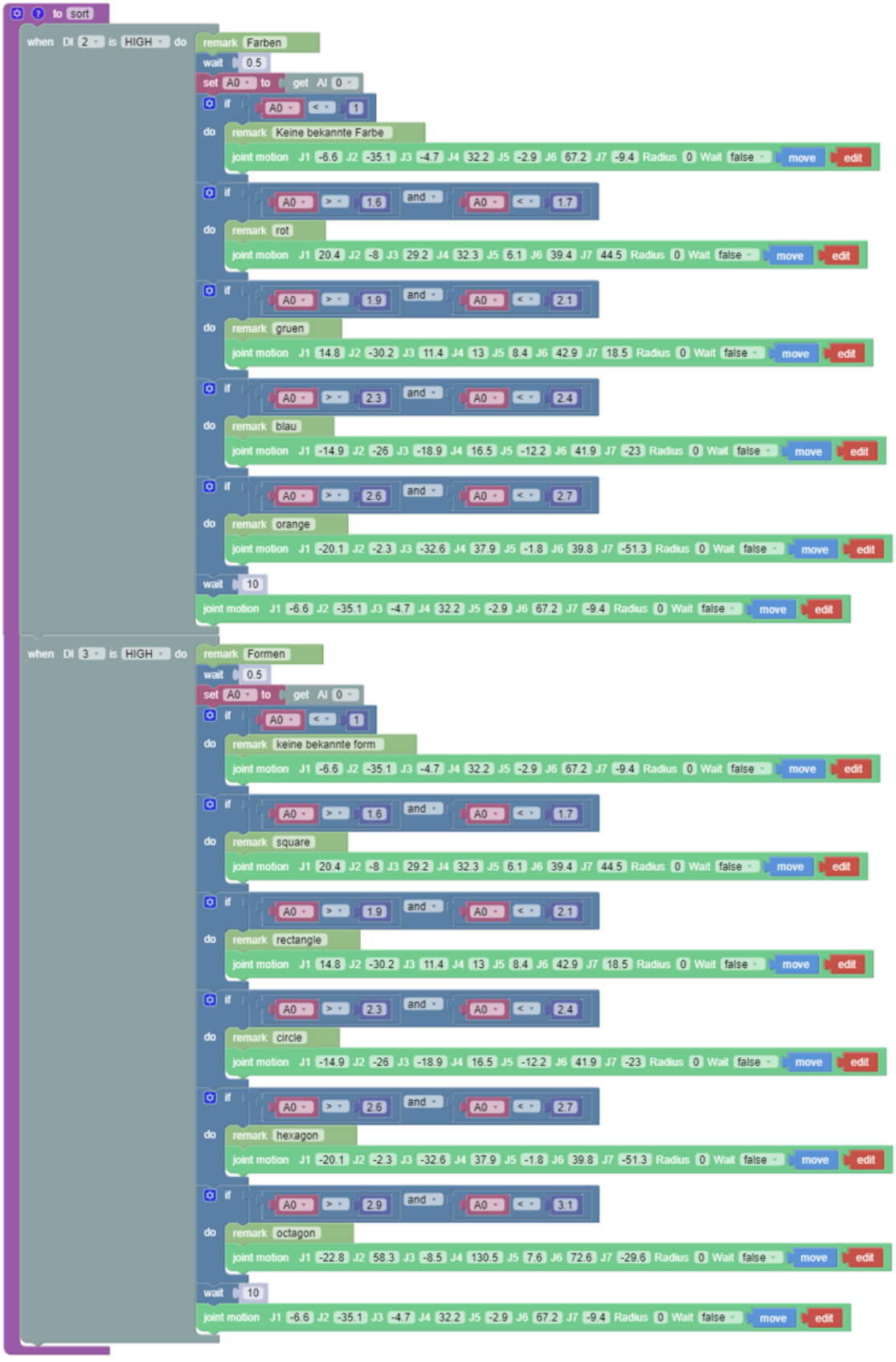


Abbildung 32: Blockly Code zur Robotersteuerung – Funktion sort [Eigene Darstellung]

Anhang D: Schachbrett zur Kamerakalibrierung

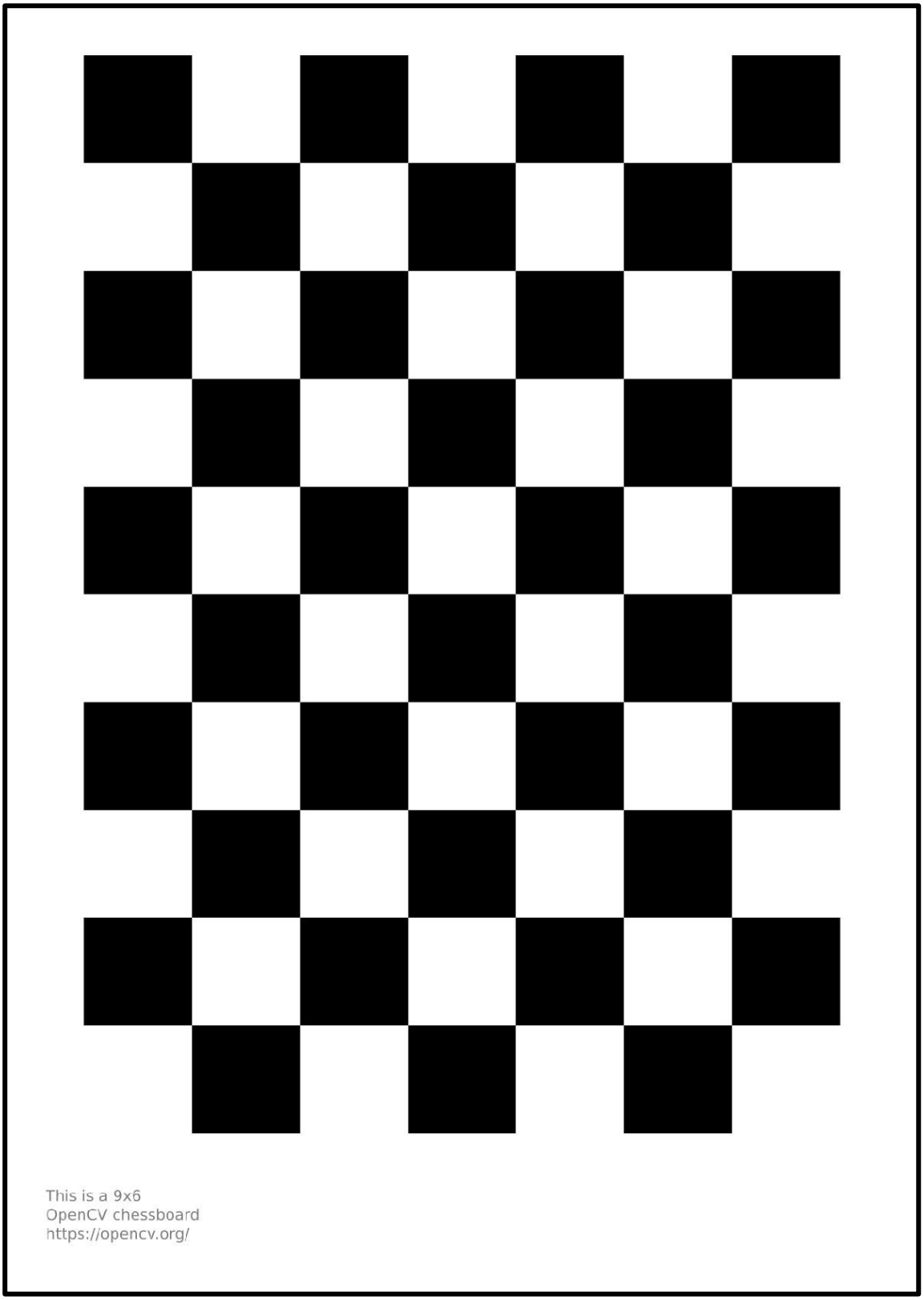


Abbildung 33: Schachbrett zur Kamerakalibrierung [25]

8 Literaturverzeichnis

[1] B. Heinrich, P. Linke, und M. Glöckler, *Grundlagen Automatisierung: Erfassen - Steuern - Regeln*, 3rd ed. Wiesbaden: Springer Fachmedien Wiesbaden, 2020.

[2] H. Maier, *Grundlagen der Robotik,* 2. überarbeitete und erweiterte Auflage. Berlin: VDE VERLAG GMBH, 2019.

[3] X. Jiang und H. Bunke, *Dreidimensionales Computersehen: Gewinnung und Analyse von Tiefenbildern*, 1st ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997.

[4] K. Dembowski, *Raspberry Pi - Das technische Handbuch: Konfiguration, Hardware, Applikationserstellung*, 2., erw. und überarb. Aufl. Wiesbaden: Springer Fachmedien Wiesbaden, 2015

[5] M. Weigend, *Raspberry Pi programmieren mit Python: Für Raspberry Pi 5, 4, 3 und Zero*, 2024. Frechen: mitp-Verlag, 2024.

[6] G. Vieira, J. Barbosa, P. Leitão und L. Sakurada, "Low-Cost Industrial Controller based on the Raspberry Pi Platform", (eng.), *IEEE International Conference on Industrial Technology (ICIT)*, 2020, S. 292-297

[7] R. Szeliski, *Computer Vision: Algorithms and Applications*, (eng.), 2nd ed. Cham: Springer International Publishing, 2022.

[8] J. Howse und J. Minichino, *Learning OpenCV 4 Computer Vision with Python 3: get to grips with tools, techniques, and algorithms for computer vision and machine learning*, (eng.) 3rd ed. Birmingham: Packt, 2020.

[9] „About”. OpenCV. https://opencv.org/about/ (abgerufen 09.10.2024)

[10] K. Bredies und D. Lorenz, *Mathematische Bildverarbeitung: Einführung in Grundlagen und moderne Theorie,* Wiesbaden: Vieweg+Teubner Verlag / Springer Fachmedien Wiesbaden GmbH, 2011.

[11] J. Ohser, *Angewandte Bildverarbeitung und Bildanalyse: Methoden, Konzepte und Algorithmen in der Optotechnik, optischen Messtechnik und industriellen Qualitätskontrolle,* München: Fachbuchverlag Leipzig im Carl Hanser Verlag, 2018.

[12] „Morphological Transformations", (eng.), OpenCV. https://docs.opencv.org/4.x/d4/d76/tutorial_js_morphological_ops.html (abgerufen 22.10.2024)

[13] „Contours: Getting Started", (eng.), OpenCV. https://docs.opencv.org/4.x/d5/daa/tutorial_js_contours_begin.html (abgerufen 22.10.2024)

[14] K. Tönnies, *Grundlagen der Bildverarbeitung*. München: Pearson Studium, 2005.

[15] H.K. Yuen, J. Princen, J. Illingworth, und J. Kittler. „Comparative study of hough transform methods for circle finding.", (eng.), *Image and Vision Computing*, vol. 8(1), S. 71–77, 1990.

[16] D.H. Douglas und T.H. Peucker „Algorithms for the reduction of the number of points required to represent a digitized line or its caricature" (eng.), *Cartographica,* vol. 10(2), S. 112-122, 1973.

[17] T. Luhmann, J. Boehm, S. Kyle, und S. Robson, *Close-Range Photogrammetry and 3D Imaging*, (eng.), 4. rev. and exten. edition, Berlin, Boston: De Gruyter, 2023.

[18] „Raspberry Pi OS", (eng.), Raspberry Pi. https://www.raspberrypi.com/software/ (abgerufen 05.11.2024)

[19] „venv – Creation of virtual environments", (eng.), Python.org, https://docs.python.org/3/library/venv.html (abgerufen 07.11.2024)

[20] „opencv-python 4.10.0.84", (eng.), pypi.org, https://pypi.org/project/opencv-python/ (abgerufen 07.11.2024)

[21] J. Siegl und E. Zocher, *Schaltungstechnik: Analog und gemischt analog/digital*, 6. Aufl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018.

[22] „I2C Bus“, telos Systementwicklung GmbH, https://de.i2c-bus.org/ (abgerufen 10.12.2024)

[23] „MCP4725 12-Bit DAC Tutorial”, (eng.), adafruit.com, https://learn.adafruit.com/mcp4725-12-bit-dac-tutorial/python-circuitpython (abgerufen 25.11.2024)

[24] M. Kofler, C. Kühnast, und C. Scherbeck, *Raspberry Pi: Das umfassende Handbuch*, 7th ed. Bonn: Rheinwerk Verlag, 2021.

[25] „9x6 OpenCV Chessboard Pattern“, (eng.), github.com, https://github.com/opencv/opencv/blob/4.x/doc/pattern.png, (abgerufen: 11.12.2024)

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Projektarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Die vorgelegte Arbeit hat weder in der gegenwärtigen noch in einer anderen Fassung schon einem anderen Fachbereich der Hochschule Ruhr West oder einer anderen wissenschaftlichen Hochschule vorgelegen.

Mülheim a.d. Ruhr, 12.12.2024

Ort, Datum Unterschrift