1 Angabe Erkennen von geschlossenen Formen

Erkennen Sie Münzen und andere geschlossene Formen auf einem Bild.

Legen Sie im ersten Schritt ein Array für die erkannten Objekte an, um diese später auch anfahren zu können. Danach lesen Sie ein bereits kalibriertes Bild mit der Funktion `cv2.imread("Bild.jpg")` ein und konvertieren Sie es mit der Funktion `cv2.cvtColor(Bild, cv2.COLOR_BGR2GRAY)` in den Graustufen-Farbraum.

Im ersten Teil der Übung werden Münzen im Arbeitsraum des Puzzlebots erkannt. Dazu soll das eingelesene Bild möglichst störungsfrei sein. Wenden Sie mit der Funktion `cv2.medianBlur(bild, Kernelgröße)` einen Median Blur Filter auf das Bild an. Die Kernelgröße gibt hier an, wie stark der Effekt des Filters zum Tragen kommt.

Hinweis: Der Kernel nimmt aufgrund seiner Funktionsweise nur ungerade Werte zwischen 1 und 31 an.

Mit der Funktion `cv2.HoughCircles(bild, Methode, Auflösung der Funktion ,Minimale Distanz zwischen Kreisen , Parameter für Kantenerkennung, Parameter für Kreiserkennung,Mindestradius, Höchstradius)` finden Sie Kreise auf einem Bild. Als Methode der Funktion verwenden Sie `cv2.HOUGH_GRADIENT` und für die Auflösung 1 (= volle Auflösung). Die jeweiligen Parameter sind so zu wählen, dass eine richtige Erkennung immer gewährleistet ist. Die Funktion gibt die gefundenen Kreise als Array zurück.

Hinweis: Wählen Sie die Parameter richtig. Beginnen Sie zuerst mit der Distanz, dem minimalen Radius und dem maximalen Radius. Danach wählen Sie den Parameter für die Kantenerkennung und die Kreise. Bei kleineren Werten der Parameter kommt es zu einer schnelleren Kreiserkennung, welche aber möglicherweise auch zur Falscherkennung führt.

Da Sie mehr als einen Kreis erkennen möchten, muss die vorherige Funktion und die weiteren Schritte in einer Schleife durchgeführt werden. In der weiteren Folge sollte überprüft werden, ob Kreise gefunden wurden. Die gefundenen Kreise werden in folgendem Format in das Array gespeichert: `(Mittelpunkt X , Mittelpunkt Y , Radius)`.

Hinweis: Sie können das Array, welches die Kreise beschreibt, mit der Funktion `numpy.around()` runden, und mit der Funktion `numpy.uint16()` in einen Integer ohne Vorzeichen umwandeln.

Optional können Sie jedes Element im Array, also jeden Kreis, auch grafisch darstellen. Dazu benötigen Sie eine weitere Schleife, welche jeden erkannten Kreis separat behandelt. Mit

der Funktion `cv2.cicles(Bild zu Darstellung, (Mittelpunkt X, Mittelpunkt Y), Radius, (Farbcode R , G , B), Linienstärke)` können Sie dies bewerkstelligen. In Abbildung 1 sehen Sie eine solche Darstellung

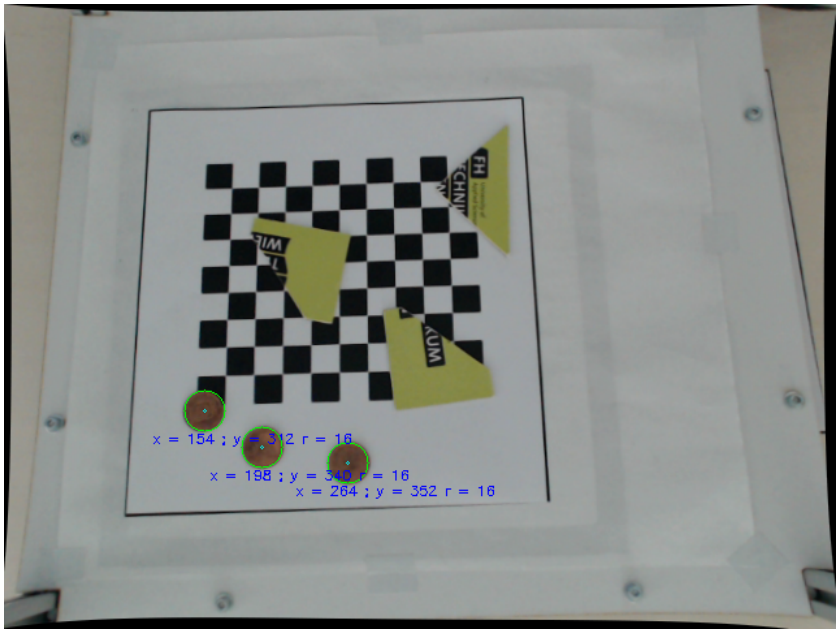

Abbildung 1: Erkannte Münzen/Kreise

Zusätzlich kann mit `cv2.putText(Bild zur Darstellung, "Text",(Position X, Y), Schriftart, Größe der Schrift,(Farbcode R , G , B), Linienstärke)` auch noch ein Text im Bild angegeben werden.

Um den Kreiserkennungsprozess nicht endlos weiterlaufen zu lassen, empfiehlt es sich, eine Abbruchbedingung zu implementieren. Bei einem Standbild ändern sich die Parameter nicht. Es ist also sehr wahrscheinlich, dass die Funktion die Kreise immer wieder genau gleich erkennt. Eine Möglichkeit wäre, die Einträge im Array miteinander zu vergleichen und bei identischen Einträgen die Suche abzubrechen. Die grafische Darstellung hilft Ihnen bei der Überprüfung, ob alle Kreise erkannt wurden.

Speichern Sie die gefundenen Kreise in einem Array oder geben Sie diese grafisch als Bild aus, und speichern Sie dieses.

Im zweiten Teil der Übung werden Sie die Puzzleteile des Puzzlebots erkennen. Lesen Sie ein Bild ein, auf dem die Puzzleteile erkannt werden sollen. Konvertieren Sie das Bild für die Aufgabe in den HSV-Farbraum. Mit der Funktion `cv2.cvtColor(bild, cv2.COLOR_BGR2HSV)` konvertieren Sie ein Bild in den HSV-Farbraum.

Im nächsten Schritt legen Sie eine Maske an. Um diese Maske zu definieren benötigen Sie

zwei Arrays vom Typ `numpy.array([H,S,V])` welche den Farbwert, die Farbsättigung und den Hellwert beinhalten. Es werden zwei Arrays benötigt, um einen Bereich festzulegen, in welchem die Maske aktiv ist.

Hinweis: Der Farbwert nimmt Werte zwischen 0 und 179, die Farbsättigung und der Hellwert nehmen Werte zwischen 0 und 255 an. Optimieren Sie zuerst den Farbwert und die Sättigung, und lassen Sie für den Hellwert alle Werte zu. Somit sind unterschiedliche Lichtverhältnisse vom Ergebnis unabhängig.

Mit der Funktion `cv2.inRange(HSV-Bild,unterer Schwellenwert,oberer Schwellenwert)` erstellen Sie die Maske. Lassen Sie sich die Maske anzeigen, um das Ergebnis visuell zu überprüfen. In Abbildung 2 sehen Sie wie eine Maske aussehen sollte.

Abbildung 2: Maske für Puzzleteile

Sollten in der Maske kleine Fehler auftreten, ist das nicht weiter schlimm, da Sie nachher auch die Fläche der erkannten Konturen mitberücksichtigen. Allerdings können Sie mit der Funktion `cv2.erode(Bild,Kernel)` Unstimmigkeiten in der Maske vorbeugen.

Hinweis: Der Kernel hat die Form einer Matrix[X/X]. Mit der Funktion `numpy.ones((Dimension X, Dimension Y), Format)` erstellen Sie einen solchen Kernel. Als Format wählen Sie am besten einen 8-Bit Integer ohne Vorzeichen.

Als nächsten Schritt werden Sie die Konturen im Bild erkennen. Das bewerkstelligt die Funktion `cv2.findContours(Maske, Modus, Methode)` für Sie. Die Funktion gibt zwei Werte zurück. Sie benötigen nur den Ersten, denn er beinhaltet ein Array mit allen gefundenen Konturen. Für den Modus der Funktion verwenden Sie `cv2.RETR_TREE` um alle Konturen und deren Reihenfol-

ge zu erkennen. Als Methoden verwenden Sie `cv2.CHAIN_APPROX_SIMPLE` um gerade Linien als ein Element wahrzunehmen.

Für jede Kontur, die gefunden wird, gibt es eine dazu gehörige Fläche. Mittels einer Schleife können Sie die Konturen einzeln ansprechen und mit der Funktion `cv2.contourArea(Kontur)` die Flächen auslesen. Bedenken Sie, dass sehr kleine Flächen höchstwahrscheinlich Fehler darstellen oder diese nicht geschlossen sind und daher ignoriert werden können.

Mit der nächsten Funktion `cv2.approxPolyDP(Kontur,Epsilon, Boolean(geschlossene Form?))` kann eine geschlossene Fläche approximiert werden. Die Funktion gibt, bei richtiger Einstellung, eine geschlossene Form als Array zurück. Epsilon beschreibt den Parameter für die Kantenerkennung. Der Parameter Epsilon kann durch einen Prozentsatz multipliziert mit der Funktion `cv2.arcLength(Kontur, Boolean(geschlossene Form?))` beschrieben werden. Umso kleiner der Prozentsatz ist, desto genauer funktioniert die Konturerkennung geschlossener Formen.

Hinweis: Geben Sie für den Boolean der Funktion `True` an.

Den nächsten Schritt sollten Sie erst dann in Angriff nehmen, wenn Sie sich versichert haben, dass alle bisherigen Konturen auch wirklich valide Ergebnisse sind. Mit der Funktion `cv2.drawContours(Bild,[geschlossene Form] ,Parameter ,(Farbraum R ,G ,B), Linienstärke)` können Sie die Konturen auf einem Bild visualisieren. Der Parameter gibt die zu zeichnende Kontur an. Setzen Sie diesen auf einen negativen Wert, um alle Konturen zu zeichnen.

Im letzten Schritt dieser Übungen müssen Sie die Schwerpunkte der Konturen bestimmen. Dazu kommt Ihnen die Funktion `cv2.moments(geschlossene Kontur)` zu Hilfe. Die Funktion gibt Ihnen sehr viele verschiedene Werte zurück. Speichern Sie diese in einem Array ab. Danach errechnen Sie sich die X- und Y-Koordinate des Schwerpunkts auf folgender Art und Weise.
`X = int(moments["m10"] / moments["m00"])` für den X-Schwerpunkt und
`Y = int(moments["m01"] / moments["m00"])` für den Y-Schwerpunkt.

Damit haben Sie die Aufgabe 2 abgeschlossen. Verwenden Sie die in Kapitel 1.1 angeführten Bibliotheken und Funktion zum Programmieren der Aufgabe 2. Bitte bedenken Sie, dass Funktionen aus der Aufgabe 1 auch verwendet werden können.

1.1 Funktionen und Bibliotheken Übung 2

Bibliotheken:

- `import cv2`
- `import numpy`

Funktionen:

- `cv2.medianBlur()` Median Blur Filter
- `cv2.HoughCircles()` Kreise mit Hough Transformation detektieren
- `cv2.circle()` Kreise visualisieren
- `numpy.array()` Erstellen eines Arrays
- `numpy.ones()` Erstellen eines Arrays, befüllt mit der Zahl 1
- `cv2.inRange()` Schwellenwert Operation
- `cv2.erode()` Erodieren
- `cv2.findContours()` Konturen erkennen
- `cv2.contourArea()` Fläche der Konturen errechnen
- `cv2.approxPolyDP()` Geschlossene Konturen erkennen
- `cv2.drawContours()` Konturen visualisieren
- `cv2.moments()` Hilfsfunktion zum Errechnen des Schwerpunkts