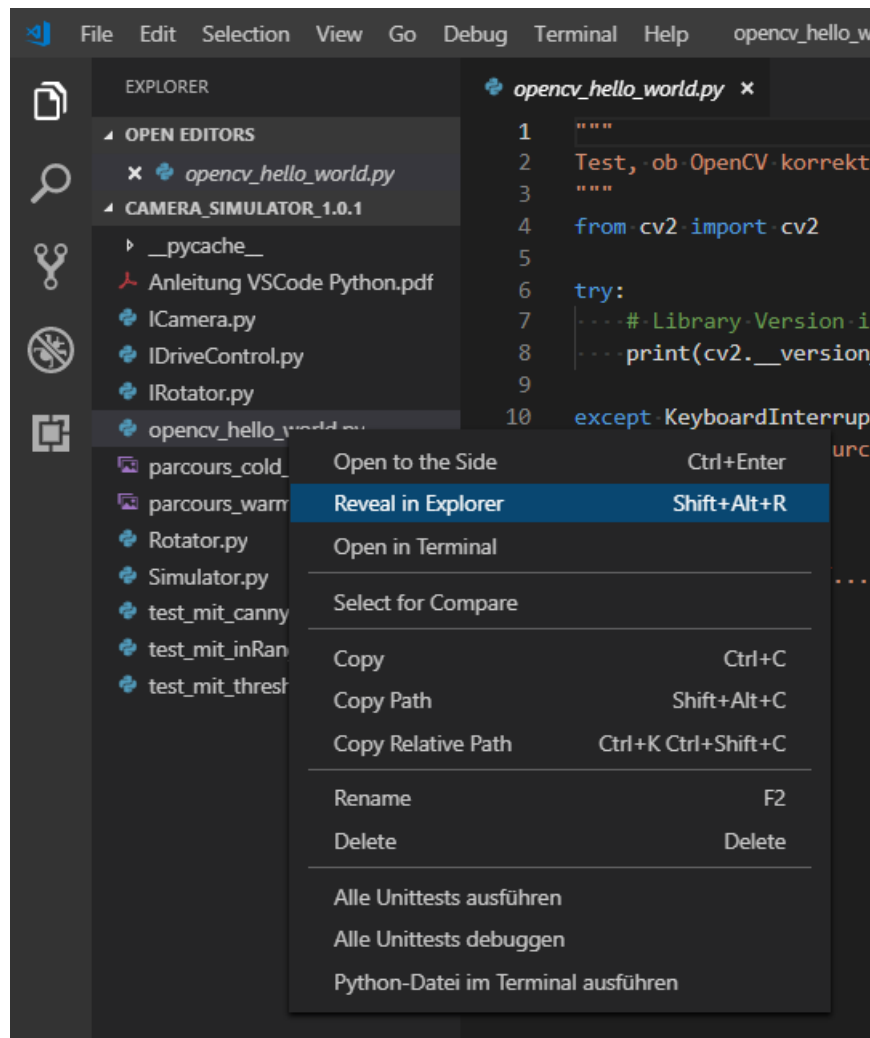


# Linienfolgen mit OpenCV

OpenCV ist eine Bibliothek mit Algorithmen für die Bildverarbeitung. In dieser Anleitung wird mit OpenCV Version 3.4.3 gearbeitet. OpenCV ist in C++ implementiert, bietet aber für verschiedene Sprachen eine API (Application Programming Interface), eine Art Adapter zwischen den Programmiersprachen. In dieser Anleitung wird die API für Python 3.7.0 mit der python numpy Bibliothek verwendet.

## 1 Vorbereitungen

- Zur Installation von Python 3, OpenCV und der Entwicklungsumgebung Visual Studio Code (VSCode) siehe „Anleitung VSCode Python“ Kapitel 1 bis 3, 6 und 7 durcharbeiten.
- VSCode öffnen
- Datei → Ordner öffnen → Ordner „camera\_simulator“ öffnen.
- Rechtsklick auf „opencv\_hello\_world.py“ → In Kommandozeile öffnen („Open in Terminal“)



- „**python.exe .\opencv\_hello\_world.py**“ eingeben und bestätigen (mit **TAB** autovervollständigen, um nicht alles manuell tippen zu müssen):

```

PROBLEMS  TERMINAL  ...  1: powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

PS C:\Users\Tom\Dropbox\Robocup 2018_2019\Material\Bildverarbeitung\camera_simulator_1.0.1> python.exe .\opencv_hello_world.py
3.4.3
Räume auf...
PS C:\Users\Tom\Dropbox\Robocup 2018_2019\Material\Bildverarbeitung\camera_simulator_1.0.1>

```

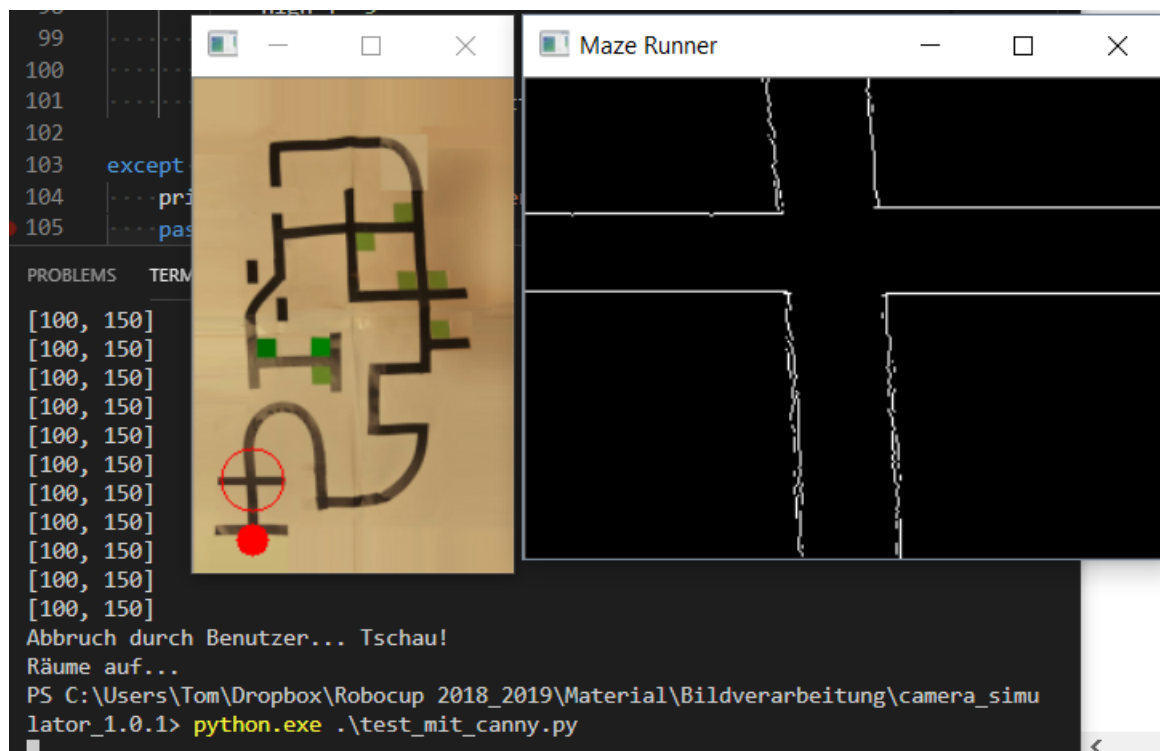
- Wenn das funktioniert, ist alles korrekt installiert.

## 2 Simulation

Für schnelle Testiterationen ohne echte Hardware kommt ein Simulator zur Anwendung, welcher ein Bild lädt und einen Ausschnitt (das Sichtfeld der Kamera) dem Testprogramm derart zur Verfügung stellt als wäre es ein von einer echten Kamera aufgenommener Frame.

Motorsteuerbefehle bewegen den Ausschnitt über das Testbild, sodass der Eindruck entsteht als würde die Kamera sich über das Bild bewegen.

Zur Inbetriebnahme der Simulation kann eines der „**test\_mit\_\*.py**“-Programme gestartet werden.



Dies öffnet 2 Fenster:

- Kamera-Sichtfeld: Die Kamera ist so montiert, dass sie exakt senkrecht auf den Parcours gerichtet ist. Entgegen der Realität gibt es keine Schattenwürfe oder dynamische Reflektionen. Sollen die Lichtverhältnisse verändert werden, muss das zugrundeliegende Bild verändert werden. Folgende Einstellungen sind möglich:
  - im Testprogramm: „**PIXEL\_BREITE**“, „**PIXEL\_HOEHE**“: Größe des Kameraausschnitts. Sollte der Auflösung der realen Kamera entsprechen.
  - im Testprogramm: „**kamera\_hoehe**“: Skalierungsfaktor des Sichtfeldes der Kamera. 1.0 entspricht der Originalgröße. Wird der Faktor vergrößert, rutscht die Kamera näher an den Parcours heran und umgekehrt.
  - im Testprogramm: „**PARCOURS\_PFAD**“: Pfad zu dem zu ladenden Parcours-Bild.
  - im Testprogramm: „**start\_x**“ in Pixeln von links, „**start\_y**“ in Pixeln von oben: Startposition des Roboters.
- Minimap: Übersicht über den gesamten Parcours. Der voranfahrende Kreis entspricht dem Sichtfeld der Kamera, der nachfolgende ausgefüllte Kreis dem Chassis des Roboters. Folgende Einstellungen sind möglich:
  - in Simulator.py: „**\_\_MINIMAP\_SCALE\_FACTOR**“: Skalierung der Minimap
  - im Testprogramm: „**kamera\_position**“: Die Kamera ist vorn am Chassis montiert. „**kamera\_position**“ ist die Länge des Chassis / 2 in Pixeln, also der Abstand der Kamera von Rotationszentrum des Roboters, wenn der auf der Stelle rotiert. Der Beispielparcours ist unrealistisch gedungen, weswegen das simulierte Chassis auch etwas kleiner als das reale ausfallen sollte.

Steuerung:

- Taste „**k**“: Simulation beenden. Alternativ kann auch in der Kommandozeile **Strg+C** gedrückt werden wie auch auf dem realen Pi.
- Taste „**p**“: Motoransteuerung deaktivieren oder aktivieren (pausieren / fortsetzen)
- Ist die Simulation pausiert, kann der Roboter manuell versetzt werden:
  - Tasten „**w**“, „**a**“, „**s**“, „**d**“: versetzen (Translation)
  - Tasten „**q**“, „**e**“: Rotationszentrum

Weitere Einstellungen:

- im Testprogramm: „**cv2.waitKey(10)**“: Framerate: stark vom Rechner abhängig, der die Simulation ausführt. Diese Zeile pausiert das Programm für 10ms und gibt OpenCV Zeit die Tastendrücke zu verarbeiten und das Fenster zu rendern. Sollte das Fenster nicht auftauchen, muss die Framerate hier verlangsamt werden. Diese Zeile ist auch der Grund, warum bei manueller Steuerung durch Tastendrücke das Bild erst aktualisiert wird, denn die Taste losgelassen wird: Da **waitkey(10)** die 10 Millisekunden nur wartet, wenn keine Taste

gedrückt wird. Es bleibt bei gedrückter Taste also keine Zeit das Fenster neu zu rendern. Auf echter Hardware wird die Framerate außerdem wesentlich langsamer ausfallen. Das kann mit diesem Parameter simuliert werden.

- im Testprogramm: „**simulator.ROBOT\_WIDTH**“: Abstand der Roboterketten in Pixeln. Entspricht somit ungefähr der Breite des Roboters. Dieser Parameter hat einen Einfluss auf das Lenkverhalten und damit die Agilität des Roboters. Da der Testparcours sehr gedrungen geraten ist, kann die Chassisbreite hier gern unrealistisch klein ausfallen.
- im Testprogramm: „**simulator.SPEED\_FACTOR**“: simulierte Fahr- und Rotationsgeschwindigkeit des Roboters, also wie schnell `setSpeed(100, 100)` maximal fahren kann. Sollte an Framerate angepasst werden.
- im Testprogramm: „**simulator.MAXIMUM\_ACCELERATION**“: Trägheit des Roboters beim Beschleunigen. Wert kann zum Testen gern mal um einige Magnituden kleiner gemacht werden. Dann fährt der Roboter wie ein Betrunkener.
- im Testprogramm: „**simulator.MAXIMUM\_DECELERATION**“: Trägheit des Roboters beim Bremsen.

## 2.1 Simulator in Programm einbinden

„`Simulator.py`“ erbt von den Klassen „`ICamera`“ und „`IDriveControl`“, wodurch unter anderem die Funktionen „`get_frame()`“ und „`set_speed()`“ zur Verfügung gestellt werden. Wir tun im Folgenden so als wäre das Simulator-Objekt eine Kamera und das Motor Shield in einem:

```
simulator = Simulator(... )
# wird zum Testen mit dem richtigen Roboter durch die echte PiCam ersetzt
kamera = simulator
# wird zum Testen mit dem richtigen Roboter durch das AFMotorShield ersetzt
motoren = simulator

# frame lesen
dieser_frame = kamera.get_frame()
# Motoren ansteuern
motoren.set_speed(30, 30)
```

Sobald das Programm auf echter Hardware getestet werden soll, wird das Simulator-Objekt einfach durch echte Treiber ersetzt, die auch von „`ICamera`“ und „`IDriveControl`“ erben und somit dieselben Funktionen zur Verfügung stellen:

```
# wird zum Testen mit dem richtigen Roboter durch die echte PiCam ersetzt
kamera = PiCam(... )
# wird zum Testen mit dem richtigen Roboter durch das AFMotorShield ersetzt
motoren = MotorShield(... )

# frame lesen
dieser_frame = kamera.get_frame()
# Motoren ansteuern
motoren.set_speed(30, 30)
```

Um die Simulation pausieren zu können, bietet die Simulator-Klasse noch die Funktion „`handle_interactive_mode()`“, die die „`waitKey()`“-Zeile ersetzen muss (dazu später mehr). Sie ist für die Simulation aber nicht zwingend notwendig.

```
simulator.handle_interactive_mode(cv2.waitKey(10))
```

Das vollständige leere Gerüst findet sich im Beispielprogramm „**test\_zum\_kopieren.py**“.

### 3 OpenCV mit Python: Grundlagen

OpenCV hat diverse Ein- und Ausgabefunktionen. Eingabefunktionen sind bspw. das Laden eines Bildes oder der Zugriff auf den Frame einer Webcam. Eine Eingabefunktion gibt immer eine Referenz auf ein Einzelbild (Frame) zurück, welches als Array gespeichert wird. Dieses Array kann dann verarbeitet werden und am Ende wieder einer Ausgabefunktion (z.B. Bild abspeichern, Preview anzeigen) übergeben werden.

Sowohl auf dem Pi als auch in der Simulation benutzen wir keine Eingabefunktion von OpenCV: Auf dem Pi nutzen wir eine Funktion der PiCam, in der Simulation

```
# frame lesen
dieser_frame = kamera.get_frame()
```

„dieser\_frame“ ist ein Array der Bildpunkte in dem aufgenommenen Kamera-Frame.

Für das Linienfolgen wird eigentlich keine Ausgabefunktion benötigt, da die in „dieser\_frame“ enthaltenen Informationen ja benutzt werden, um die Motoren anzusteuern:

```
# Motoren ansteuern
motoren.set_speed(30, 30)
```

Eine Preview ist aber praktisch und die bietet OpenCV:

```
while True:
    # frame lesen
    dieser_frame = kamera.get_frame()
    # Kamerabild anzeigen
    cv2.imshow("Maze Runner", dieser_frame)
    cv2.waitKey(10)
```

Die Funktion „imshow()“ öffnet ein Fenster und rendert den Frame. Sie nimmt 2 Argumente auf: Den Titel des Fensters und das Array mit den Bildpunkten. Solange der Titel nicht verändert wird, bleibt dasselbe Fenster immer offen auch wenn „imshow()“ mehrere Male hintereinander aufgerufen wird. Ein anderer Titel würde ein weiteres Fenster öffnen. Es reicht nicht aus den Inhalt von „dieser\_frame“ zu ändern; um das Fenster neu zu rendern, muss immer „imshow()“ aufgerufen werden. Packen wir das ganze in eine Schleife, hat OpenCV allerdings nie Zeit das Fenster zu öffnen und zu rendern, weswegen „imshow()“ immer von „waitKey()“ gefolgt sein muss.

„waitKey()“ nimmt 1 Argument auf: Das Timeout in Millisekunden. Die Funktion blockiert so lang bis entweder eine Taste gedrückt wird oder das Timeout abgelaufen ist. Wird keine Taste gedrückt, hängt die Schleife also immer 10 Millisekunden in dieser Zeile und gibt OpenCV Zeit, das Preview-Fenster zu rendern.

Bitte behaltet das Timing im Hinterkopf: Die Preview zu rendern senkt die Framerate und verlangsamt die Hauptprogrammschleife erheblich. Fährt der Roboter autonom, werdet ihr die Preview vmtl. deaktivieren müssen, wodurch die Algorithmen wesentlich fixer laufen und sich das Timing des Programms ändert! Aus diesem Grund ist es vmtl. sinnvoll, die Preview in einen eigenen Thread (d.h. in einen anderen CPU-Kern) auszulagern, um dieses Phänomen zu minimieren.

Die Funktionen werden im Beispielprogramm „**test\_zum\_kopieren.py**“ benutzt.









### 3.1 Numpy Arrays

Zwischen Ein- und Ausgabefunktion soll der Frame konvertiert, nachbearbeitet, ausgewertet und Markierungen in ihn hineingezeichnet werden. Dafür ist es wichtig das Datenformat zu kennen, in dem ein OpenCV Frame in Python abgespeichert wird: Ein numpy-Array. „Numpy“ ist eine Bibliothek, die das Arbeiten mit Arrays in Python vereinfacht sowie performanter macht.

Ein Frame ist ein 3-dimensionales Array. Dimensionen:

- Zeilen
- Spalten
- Farbanteile (3 Anteile: Rot, Grün, Blau, in der Reihenfolge BGR abgespeichert)

Nehmen wir folgendes 2x4-Bild (besteht also nur aus 8 Pixeln):

dieser_frame	Spalte 0	Spalte 1	Spalte 2	Spalte 3
Zeile 0				
Zeile 1				

Würden wir diesen Frame ausgeben, sähe das so aus:

```
→ print(dieser_frame)
[
  [
    [0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 0, 255]
  ],
  [
    [0, 255, 0], [255, 0, 0], [255, 255, 255], [0, 0, 0]
  ]
]
```

- Rot = [blau=0, grün=0, rot=255]
- Grün = [blau=0, grün=255, rot=0]
- Blau = [blau=255, grün=0, rot=0]
- Weiß = [blau=255, grün=255, rot=255]
- Schwarz = [blau=0, grün=0, rot=0]

Siehe hier: <https://alloyui.com/examples/color-picker/hsv>

Den weißen pixel adressieren (ausschneiden):

```
→ weisser_pixel = dieser_frame[1][2] # Zeile1, Spalte 2
→ print(weisser_pixel)
[
  [
    [255, 255, 255]
  ]
]
```

Die ganze 2. Zeile Adressieren:

```

→ zeile_zwei = dieser_frame[1]          # Zeile 1
oder → zeile_zwei = dieser_frame[-1]    # 1. Eintrag von hinten (auch Zeile 1)
oder → zeile_zwei = dieser_frame[1:2]   # Zeile 1 bis nicht mehr mit inbegriffen (exklusive Zeile 2)
oder → zeile_zwei = dieser_frame[1:]    # Zeile 1 bis zum Ende der Zeilen
oder → zeile_zwei = dieser_frame[-1:]   # alles außer dem ersten Eintrag (Zeile 0)
→ print(zeile_zwei)
[
    [
        [0, 255, 0], [255, 0, 0], [255, 255, 255], [0, 0, 0]
    ]
]

```

Die Spalten 1 und 2 adressieren:

```

→ spalten_1_und_2 = dieser_frame[:,1:2] # Alle Zeilen [:], dann Spalten 1 und 2 auswählen [1:2]
→ print(spalten_1_und_2)
[
    [
        [0, 255, 0],
        [255, 0, 0]
    ],
    [
        [255, 0, 0],
        [255, 255, 255]
    ]
]

```

Nur den roten Farbkanal:

```

→ roter_kanal = dieser_frame[:, :, 2:] # Alle Zeilen, alle Spalten, dritte Farbe bis zum Ende der Farben
→ print(roter_kanal)
[
    [
        [255], [0], [0], [255]
    ],
    [
        [0], [0], [255], [0]
    ]
]

```

Eintrag, den es nicht gibt:

```

→ leer = dieser_frame[2:] # Zeile 2 gibt es nicht, nur Zeilen 0 und 1
→ print(leer)
[]

```

Man kann mit den Array-Einträgen auch rechnen. Dabei werden die Rechenoperation Eintrag für Eintrag ausgeführt:

```

→ import numpy          # numpy-Bibliothek importieren
→ a = numpy.array([1, 2]) # neues numpy-Array mit Namen a erzeugen
→ print(a)
[1, 2]
→ print(a+a) # alle Einträge mal 2
[2, 4]
→ print(a*10) # alle Einträge mal 10
[10, 20]

```

Alle Funktionen von OpenCV verwenden dieses Format. Wie wir gesehen haben, können wir aber auch manuell in einem Frame herumfuhrwerken, z.B. ein Teil des Bildes ausschneiden:

```

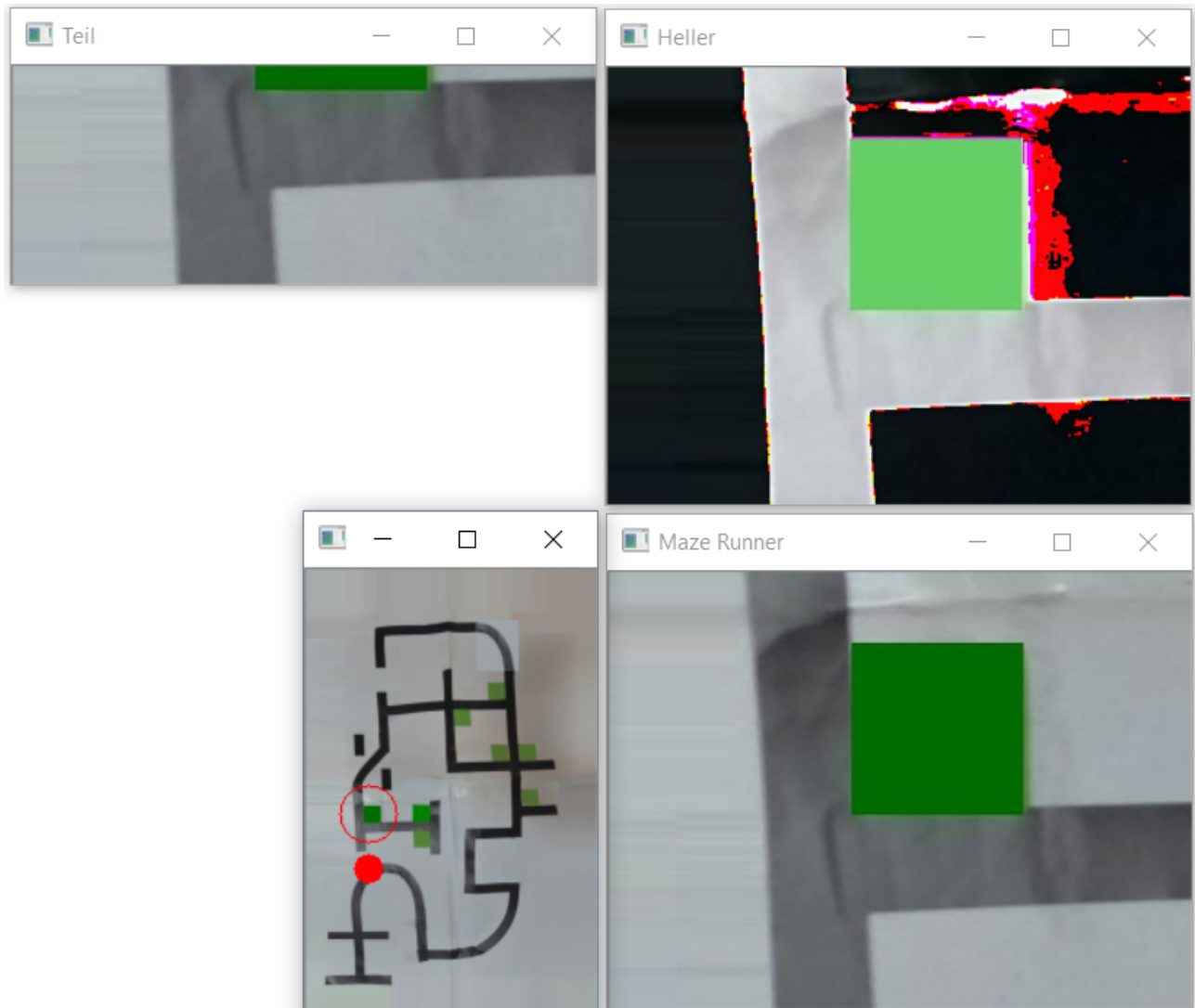
while True:
    # frame lesen
    dieser_frame = kamera.get_frame()
    teilframe = dieser_frame[50:100, 50:100]
    # Kamerabild anzeigen
    cv2.imshow("Maze Runner", dieser_frame)
    cv2.imshow("Teil", teilframe)
    cv2.waitKey(10)

```

### Aufgaben:

1. Kopiere „test\_zum\_kopieren.py“ und zeige nur die untere Hälfte des Frames in einem eigenen Fenster, ähnlich wie in dem Beispiel oben.
2. Kopiere „test\_zum\_kopieren.py“ und versuche das Bild heller zu machen.

Lösungen: „aufgabe\_numpy.py“



Huch, was ist da passiert!? Da habe ich das Bild wohl ein bisschen zu hell gemacht. Die Farbkomponenten lassen Zahlen zwischen 0 und 255 zu. Wenn man da drüber kommt, gibt es einen Überlauf und der resultierende Wert ist wieder ganz klein (schwarz).

## 3.2 Kopien vs. Referenzen

In „aufgabe\_numpy.py“ wird der Frame ganz einfach heller gemacht, indem jeder Array-Eintrag mit einem Offset addiert wird:

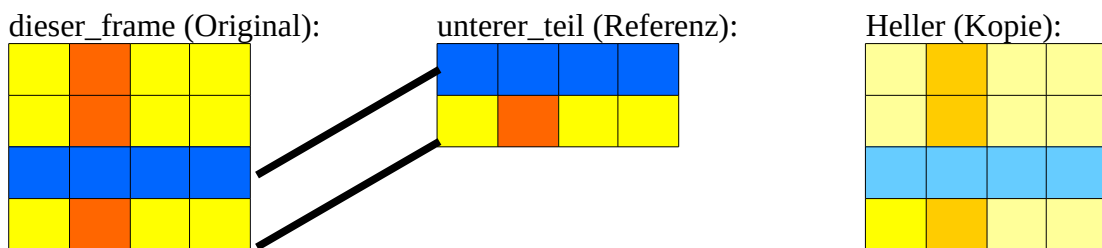


```
dieser_frame = kamera.get_frame()
unterer_teil = dieser_frame[int(PIXEL_HOEHE/2):,:,:]
cv2.imshow("Teil", unterer_teil)
heller = dieser_frame + 50
cv2.imshow("Heller", heller)
cv2.imshow("Maze Runner", dieser_frame)
```

Im Ergebnis war das Bild im Fenster „Heller“ anders gefärbt wie in den Fenstern „Maze Runner“ und „Teil“. Das liegt daran, dass die Zeile

```
heller = dieser_frame + 50
```

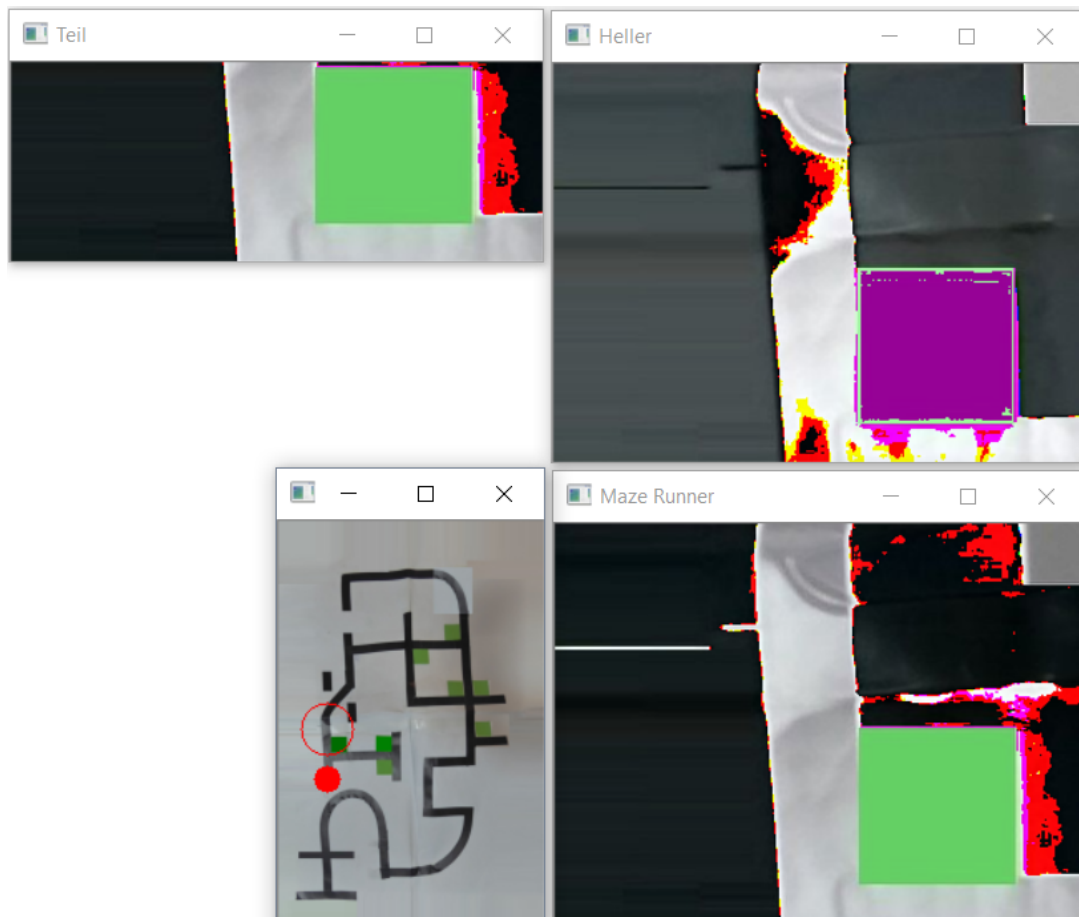
eine Kopie des Frames erstellt hat. Er existiert nun 2x im RAM und beide Kopien können unabhängig voneinander bearbeitet werden.



Modifiziere das Beispiel nun so, dass wir „dieser\_frame“ direkt bearbeiten:

```
dieser_frame = kamera.get_frame()
dieser_frame = dieser_frame + 100
...
```

Das sieht dann so aus:



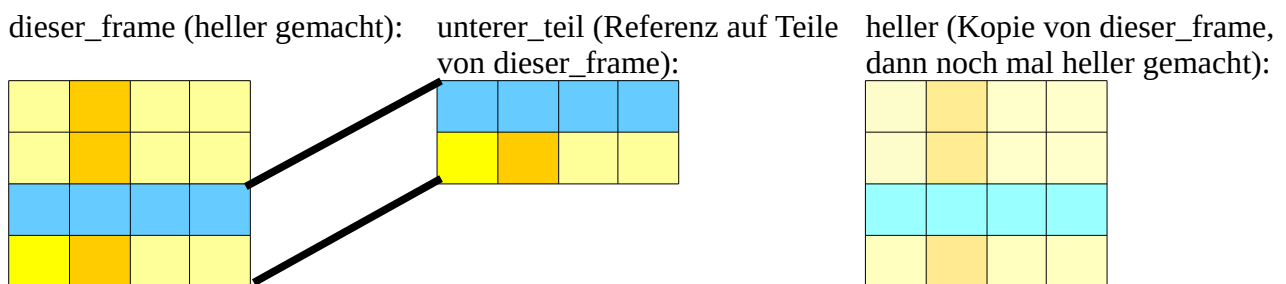
Scharfsinnige Beobachter werden feststellen, dass sich auch die Farbe im Fenster „Teil“ mitverändert hat, obwohl dort doch das Array „unterer\_teil“ angezeigt wird, nicht „dieser\_frame“:

```
cv2.imshow("Teil", unterer_teil)
```

Das liegt daran, dass die Zeile

```
unterer_teil = dieser_frame[int(PIXEL_HOEHE/2):,:,:]
```

keine Kopie erzeugt hat, sondern immer noch das Original-Array „dieser\_frame“ referenziert, wenn auch nur einen Teil davon. Das Bild existiert hier nur 1x im RAM und die Variablen „dieser\_frame“ und „unterer\_teil“ referenzieren verschiedene Stellen darin.



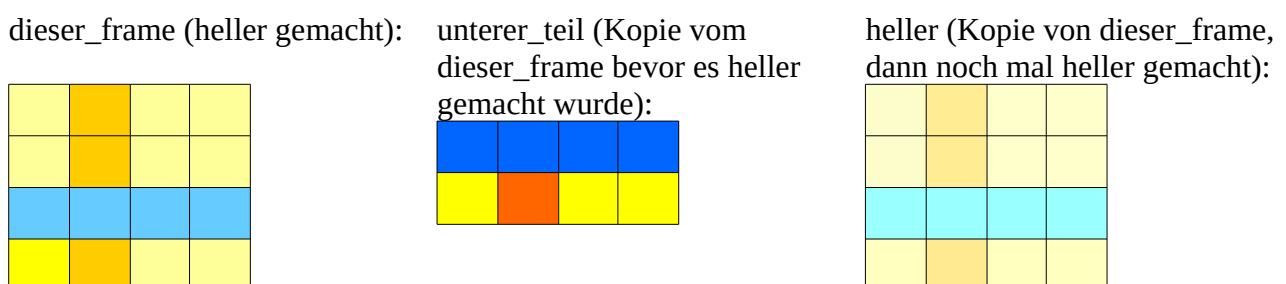
Da muss man immer ein bisschen aufpassen, dass man nicht ein Stück Array modifiziert, was vielleicht woanders noch verwendet wird.

Abhilfe schafft die „copy()“-Funktion:

```
kopie = dieser_frame.copy()
```

### Aufgabe:

1. Erstelle eine Kopie, damit das Fenster „Teil“ nicht mehr aus Versehen mit verfärbt wird:



Lösung: „aufgabe\_ref\_und\_copy.py“

## 3.3 Farbräume

Bisher haben wir mit dem RGB-Farbraum gearbeitet (in der Reihenfolge BGR abgespeichert). Hierbei besteht jeder Pixel jeweils aus einem Rot-, Grün- und Blauanteil, aus dem dann die Farben gemischt werden. Das hat technische Gründe, da jeder Pixel in einem Display aus jeweils einer roten, grünen und blauen LED besteht und auch die Fotodetektoren in Kamerasensoren so aufgebaut sind.

### Beispiel 1 in RGB:

- schwarz: [0, 0, 0]
- grau: [100, 100, 100]
- weiß: [255, 255, 255]

Das Problem hierbei ist, dass die Helligkeit in allen 3 Farbanteilen enthalten ist. Bei dem ersten Beispiel (schwarz über grau bis weiß) ist das noch relativ einfach zu durchschauen, da ja jede Farbe den gleichen Anteil hat.

### Beispiel 2 in RGB:

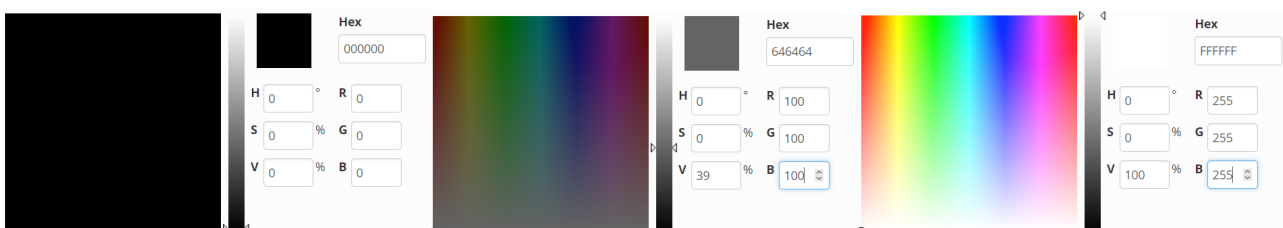
- dunkelblau: [71, 76, 145]
- blau: [95, 103, 207]
- hellblau: [118, 127, 255]

Man sieht, dass blau (jeweils der 3. Eintrag) den größten Anteil hat, kann aber nicht spontan sagen, ob zwischen den Farben „dunkelblau“, „blau“ und „hellblau“ sich der Farbton verändert hat (z.B. zu cyan, himmelblau, marineblau usw.). In diesem Fall wurde aber der Farbton nie verändert; nur die Helligkeit.

### 3.3.1 HSV Farbraum (hat nichts mit Fußball zu tun)

Das HSV-Farbschema ist für unsere Zwecke besser geeignet. Auch dieses Farbschema benutzt 3 Komponenten, um eine Farbe zu beschreiben. Diese sind aber Farbton (Hue), Sättigung (Saturation) und Helligkeit/Dunkelwert (Value → Die Terminologie ist ein bisschen komisch). Mit dem Farbton sind alle Farben in dem Bild unten von links nach rechts gemeint. Stark gesättigt ist oben („frische“ Farben) und wenig gesättigt unten („ausgewaschene“ Farben). Die Helligkeit ist der Regler rechts daneben, mit dem das ganze Bild in der Helligkeit reguliert wird:

#### Beispiel 1 (zoomen, um die Zahlen lesen zu können):



(Quelle: <https://alloyui.com/examples/color-picker/hsv>)

#### In HSV:

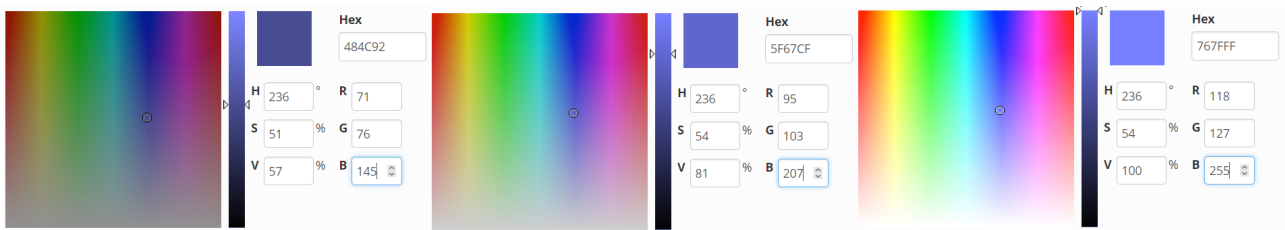
- schwarz: [0, 0, 0]
- grau: [0, 0, 39]
- weiß: [0, 0, 100]

Der Farbton-Wert ist immer 0 (grau, weiß und schwarz sind keine Farben).

Der Sättigungs-Wert ist immer 0 (sehr ausgewaschene Farben).

Der Helligkeitswert ändert sich, wenn die Helligkeit verändert wird.

### Beispiel 2 in HSV:



- dunkelblau: [236, 51, 57]
- blau: [236, 54, 81]
- hellblau: [236, 54, 100]

Der Farbton-Wert bleibt immer identisch (Marineblau).

Den Sättigungswert habe ich aus Versehen ein wenig mit modifiziert, aber der bleibt theoretisch auch identisch, d.h. die Brillanz der Farbe verändert sich nicht.

Der Helligkeitswert verändert sich stark, was das erwartete Ergebnis ist, da ich die Helligkeit modifiziert habe.

Durch den Hue-/Farbton-Wert wird also jeder Pixel des Kamera-Frames zu einem Farbsensor und durch den Value-/Helligkeitswert zu einem Helligkeitssensor.

Das numpy-Array ist 3-dimensional, wobei eine Dimension den 3 Farbkomponenten entspricht.

Das Array aus Kapitel 3.1 sieht in HSV konvertiert so aus (wie bei BGR, aber mit anderen Inhalten):

```
[
  [
    [0, 255, 255], [60, 255, 255], [120, 255, 255], [0, 255, 255]
  ],
  [
    [60, 255, 255], [120, 255, 255], [0, 0, 255], [0, 0, 0]
  ]
]
```

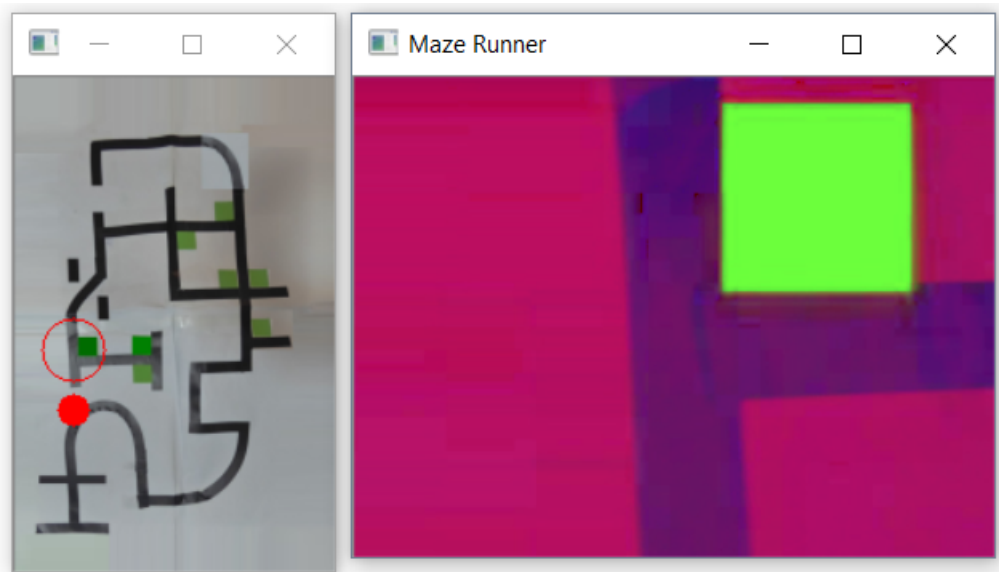
Für die Konvertierung mit OpenCV nutzt man die Funktion „**cvtColor**“. Diese nimmt 2 Argumente auf: Den Original-Frame als numpy-Array und den Konvertierungsmodus (für HSV: „**COLOR\_BGR2HSV**“). „cvtColor“ gibt eine Kopie im HSV-Farbraum zurück:

```
# frame lesen
dieser_frame = kamera.get_frame()
# Kopie von dieser_frame im HSV-Farbraum erstellen und in frame_als_hsv speichern:
frame_als_hsv = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2HSV)
```

Das Beispielprogramm dazu heißt „**test\_hsv.py**“.

Da „imshow“ ein Array im BGR-Farbraum erwartet, ist das Ergebnis für „imshow“ nicht zu gebrauchen:

```
# Kamerabild anzeigen
cv2.imshow("Maze Runner", frame_als_hsv)
```



Man sollte also lieber weiterhin den Original-Frame anzeigen:

```
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
```

Die HSV-Kopie werden wir gleich weiter auswerten.

### 3.3.2 Monochrom (Graustufen)

Bisher hatten wir es mit einer ganzen Reihe verschiedener Farben zu tun. Am Beispiel RGB 256 mögliche Rotwerte mal 256 mögliche Grünwerte mal mögliche 256 Blauwerte, macht 16.777.216 verschiedene Farben.

Diese Komplexität möchte man aber nicht immer unbedingt haben. Wenn man z.B. die Aufgabe bekäme eine schwarze Linie auf einer weißen Unterlage zu detektieren, reicht uns ein farbenblinder Sensor, der nur noch Graustufen sieht. Hierbei besteht ein Pixel dann nicht mehr aus 3 Komponenten, sondern nur noch aus einer. Jeder Pixel kann also nur noch 256 mögliche „Farben“ abbilden:

- schwarz: 0
- grau: 100
- weiß: 255

Hier ist jeder Pixel dann nur noch ein Helligkeitssensor. Die Farbinformation geht verloren.

Das numpy-Array ist 2-dimensional, da jeder Helligkeitswert direkt an der Pixelposition abgespeichert werden kann.

Das Array aus Kapitel 3.1 sieht in Grayscale konvertiert so aus:

```
[
    [76, 150, 29, 76],
    [150, 29, 255, 0]
]
```

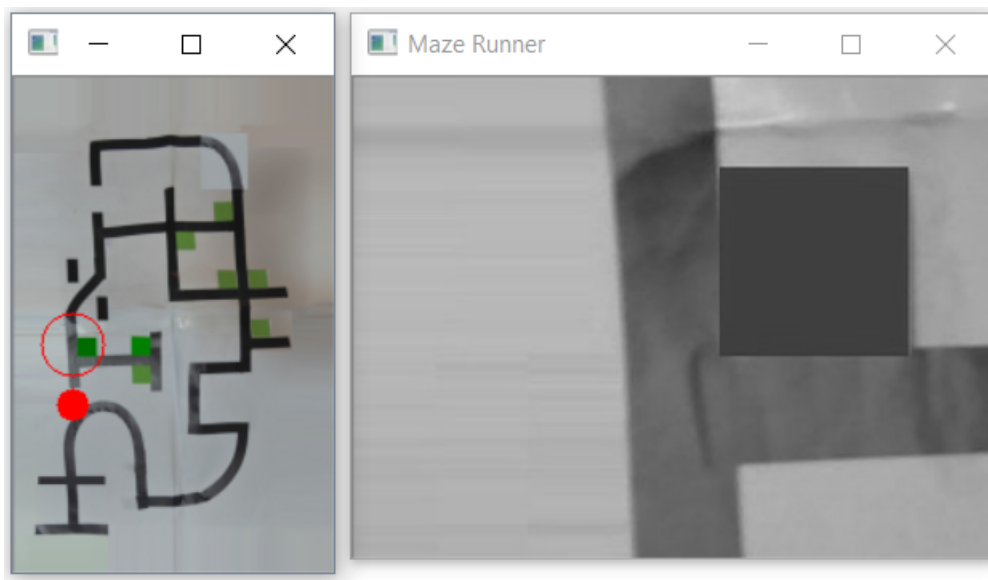
Für die Konvertierung mit OpenCV nutzt man die Funktion „**cvtColor**“. Diese nimmt 2 Argumente auf: Den Original-Frame als numpy-Array und den Konvertierungsmodus (für monochrom: „**COLOR\_BGR2GRAY**“). „cvtColor“ gibt eine Kopie im GRAY-Farbraum zurück:

```
# frame lesen
dieser_frame = kamera.get_frame()
# Kopie von dieser_frame in Graustufen konvertieren
frame_graustufen = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2GRAY)
```

Das Beispielprogramm dazu heißt „**test\_gray.py**“.

„imshow“ erkennt, dass das numpy-Array nur 2 Dimensionen hat und stellt das Bild korrekt dar:

```
# Kamerabild anzeigen
cv2.imshow("Maze Runner", frame_graustufen)
```



### 3.3.3 Binär (schwarz/weiß)

Wie wir später sehen werden, wird in OpenCV auch sehr häufig mit binären Bildern gearbeitet. Hier gehen auch die Graustufen verloren und wir bekommen ein reines schwarz/weiß-Bild, d.h. es gibt nur noch 2 mögliche Farben:

- schwarz: 0
- weiß: 255

Sowas eignet sich super zur Fallunterscheidung (z.B. es ist ein Objekt da vs. es ist kein Objekt da).

Das numpy-Array ist 2-dimensional und es gibt nur noch 2 mögliche Werte, die darin gespeichert sind.

Das Array aus Kapitel 3.1 sieht in schwarz/weiß konvertiert so aus:

```
[
    [255, 0, 255, 255],
    [0, 255, 0, 255]
]
```

Die Konvertierung heißt in OpenCV „Maskierung“, d.h. man erstellt eine Maske des Originalbildes nach gewissen Regeln. Folgend werden 3 Funktionen vorgestellt:

- „**threshold**“ (Maskierung mit einem Schwellenwert) und
- „**inRange**“ (Maskierung mit 2 Schwellenwerten)
- „**canny**“ edge detection
- Weitere Varianten: [https://docs.opencv.org/3.4.0/d5/d0f/tutorial\\_py\\_gradients.html](https://docs.opencv.org/3.4.0/d5/d0f/tutorial_py_gradients.html)

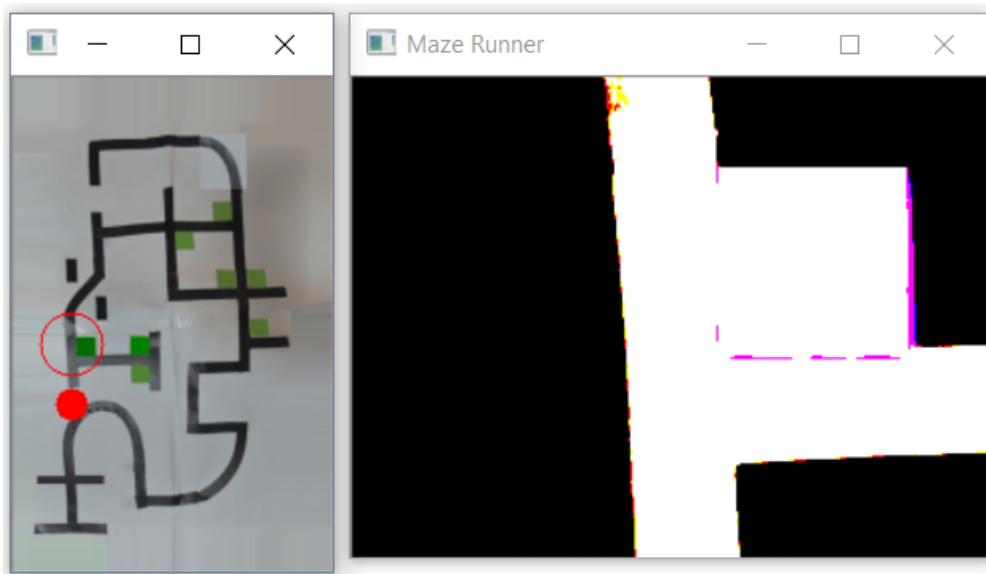
### 3.3.2.1 threshold(...)

„**threshold**“ konvertiert Pixelwerte auf einer Seite des Schwellenwertes in schwarz und die Pixelwerte auf der anderen Seite der Schwelle in weiß.

„threshold“ gibt eine Kopie im GRAY-Farbraum zurück, wobei aber nur 2 mögliche Farben existieren.

```
# frame lesen
dieser_frame = kamera.get_frame()
# alle Werte in dieser_frame bei Schwellenwert 120 in 0 oder 255 zerteilen
_, frame_schwarz_weiss = cv2.threshold(dieser_frame, 120, 255, cv2.THRESH_BINARY_INV)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", frame_schwarz_weiss)
```

Das Ergebnis ist allerdings nicht wirklich schwarz/weiß:



Das liegt daran, dass „dieser\_frame“ ja im BGR-Format gespeichert ist, also ein 3D-Array.

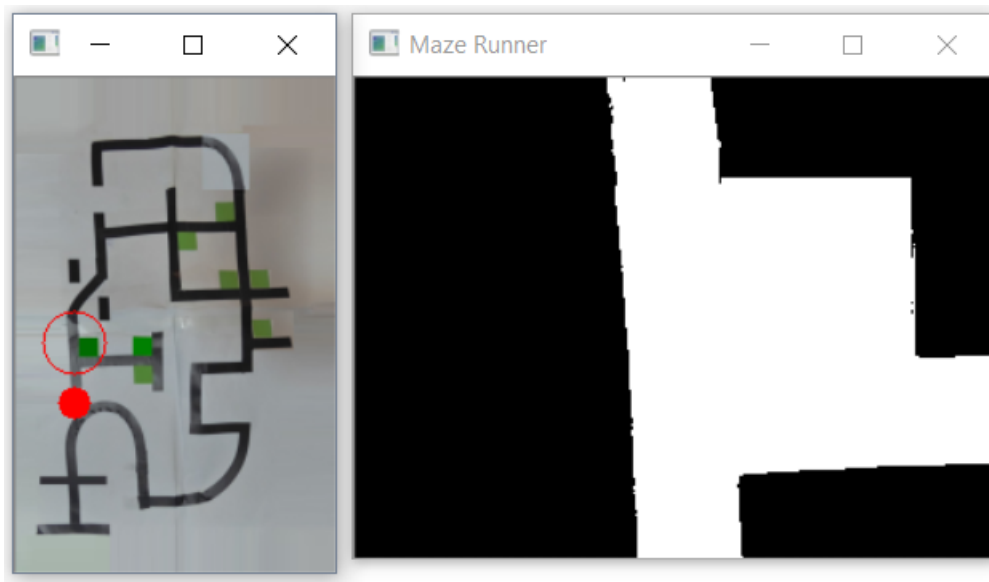
Das Array aus Kapitel 3.1 sieht konvertiert so aus. Es mischen sich also immer noch Farben, da weiterhin alle 3 Farbkomponenten existieren:

```
[
    [
        [0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 0, 255]
    ],
    [
        [0, 255, 0], [255, 0, 0], [255, 255, 255], [0, 0, 0]
    ]
]
```

Um wirklich ein schwarz/weiß-Bild zu erhalten, muss es zwischendurch in ein Graustufenbild konvertiert werden. In dem folgenden Beispiel wird jeder Graustufen-Pixel mit Helligkeit < 120 weiß dargestellt (255) und alle anderen Pixel schwarz (0).

```
# frame lesen
dieser_frame = kamera.get_frame()
# Kopie von dieser_frame in Graustufen konvertieren
frame_graustufen = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2GRAY)
# alle Werte in dieser_frame bei Schwellenwert 120 in 0 oder 255 zerteilen
_, frame_schwarz_weiss = cv2.threshold(frame_graustufen, 120, 255, cv2.THRESH_BINARY_INV)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", frame_schwarz_weiss)
```

„imshow“ erkennt, dass das numpy-Array nur 2 Dimensionen hat und stellt das Bild korrekt dar:



„threshold“ nimmt 4 Parameter auf:

1. Graustufen-Frame als 2D-numpy-Array
2. Schwellenwert
3. Der größere von den beiden Ergebniswerten: Der erste ist immer 0. Im Beispiel oben ist er andere 255. Das ist notwendig, um das Bild mit „imshow“ korrekt darzustellen (255=weiß). Zur Auswertung kann man aber auch andere Werte wählen, z.B. 1. Dann sähe das Array aus Kapitel 3.1 so aus, ist also wirklich binär:

```
[
    [
        [1, 0, 1, 1],
        [0, 1, 0, 1]
    ]
]
```



So ein Array mit „imshow“ darzustellen ergibt aber wenig Sinn, da 1 fast schwarz entspricht. Es eignet sich aber gut zur weiteren Auswertung.

4. Modus. Wir werden meist verwenden:

- „**THRESH\_BINARY**“ (Helligkeitswert < Schwellenwert = 0, sonst 255) oder
- „**THRESH\_INV**“ (Helligkeitswert < Schwellenwert = 255, sonst 0) → also invertiert.

„threshold“ gibt 2 Rückgabewerte zurück, wobei uns der erste nicht interessiert (deswegen der Unterstrich „\_“ als Platzhalter). Der 2. Rückgabewert ist eine Kopie des Original-Arrays, nach Schwellenwert binarisiert.

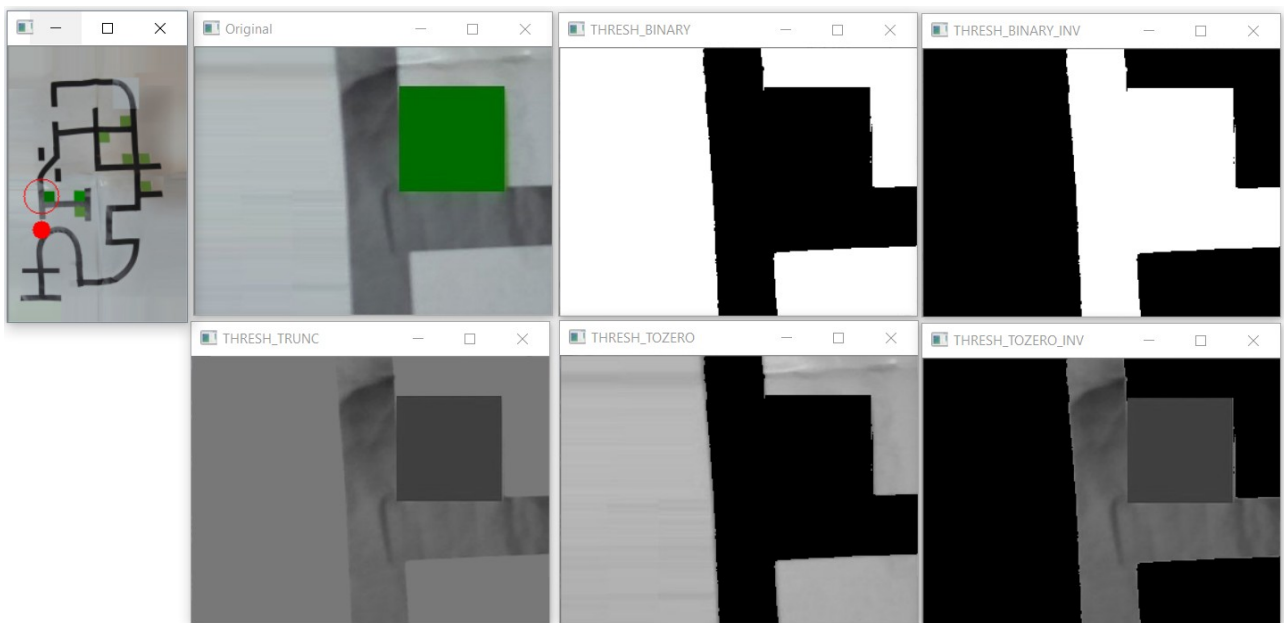
Das Beispielpogramm dazu heißt „**test\_threshold.py**“.

### Aufgabe:

Siehe [https://docs.opencv.org/3.4.0/d7/d4d/tutorial\\_py\\_thresholding.html](https://docs.opencv.org/3.4.0/d7/d4d/tutorial_py_thresholding.html)

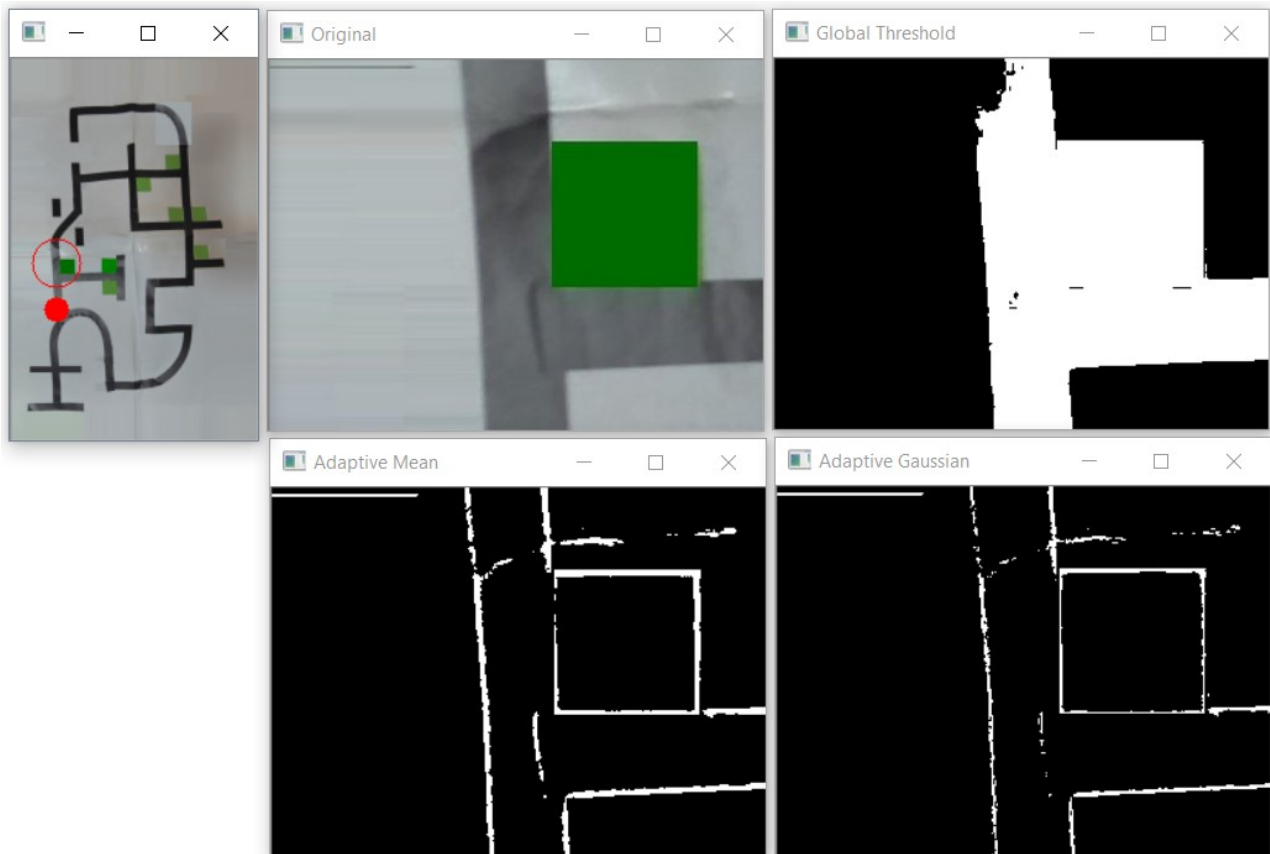
1. Kopiere „test\_threshold.py“ und probiere die anderen Modi aus, die in dem Link erwähnt werden (z.B. **THRESH\_TRUNC**).

Lösungsvorschlag: „**aufgabe\_threshold1.py**“ (die Werte für die Konstanten muss man ausprobieren)



2. Kopiere „test\_threshold.py“ und probiere die in dem Link erwähnte „**adaptiveThreshold**“-Funktion aus.

Lösungsvorschlag: „**aufgabe\_threshold2.py**“ (die Werte für die Konstanten muss man ausprobieren)



3. Probiere alles nochmal mit der anderen Parcours-Datei „**parcours\_warm\_RESIZED.JPG**“, welche andere Beleuchtungsbedingung simuliert.

### 3.3.2.2 inRange(...)

„**inRange**“ funktioniert ähnlich wie „**threshold**“, nimmt aber 2 Schwellenwerte auf: Einen oberen und einen unteren. Somit kann man also gewisse Helligkeitsbereiche „herausschneiden“.

Richtig sinnvoll wird das aber erst, wenn wir nicht wie bei „**threshold**“ mit Graustufen arbeiten, sondern mit RGB oder HSV (...lieber HSV). Damit ist es dann möglich einen bestimmten Farbbereich herauszuschneiden.

„**threshold**“ gibt eine Kopie im GRAY-Farbraum zurück, wobei aber nur 2 mögliche Farben existieren.

Folgendes Beispiel konvertiert `dieser_frame` in den HSV-Farbraum und schneidet dann einen Farbbereich aus, der folgende Eigenschaften hat:

- Farbton 40 (sehr gelbstichig, eher orange) bis 80 (grasgrün)
- Sättigung 80 (sehr überbelichtet) bis 255 (sehr saftige Farben)
- Helligkeit 20 (fast schwarz) bis 255 (voll ausgeleuchtet)

Schaut euch den Bereich im ColorPicker mal an. Aber Obacht: **In OpenCV gehen alle Werte immer von 0 bis 255.** Im ColorPicker:

<https://alloyui.com/examples/color-picker/hsv>

- H: von 0 bis 359 (in °) → ja, „Hue“ beschreibt einen Farbkreis
- S und V: 0 bis 100 (in %)

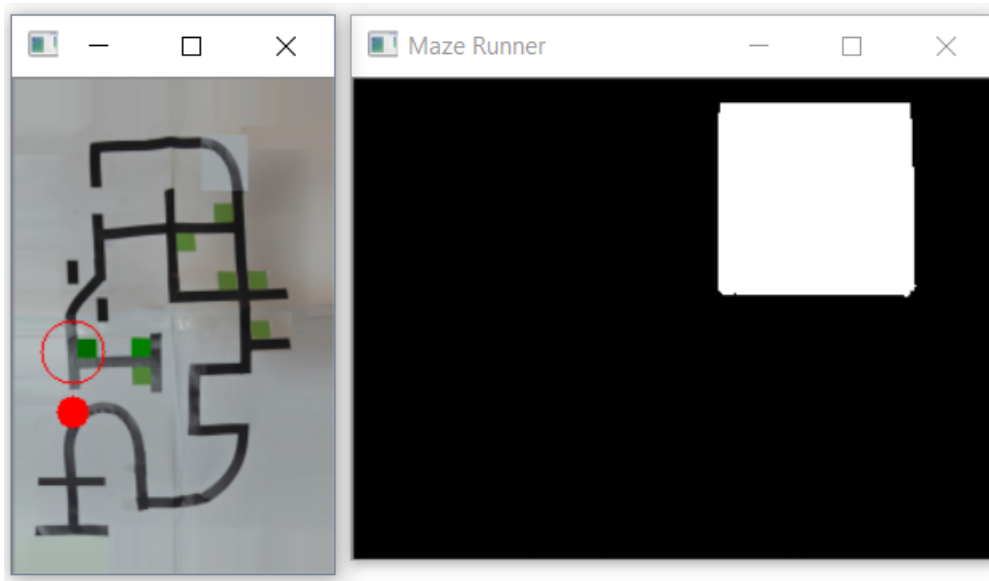
D.h. man muss die Werte aus dem ColorPicker noch umrechnen:

- $S\_V\_openCV = S\_V\_color\_picker / 100 * 255$  oder
- $H\_openCV = H\_color\_picker / 359 * 255$

„inRange“ nimmt 3 Argumente auf: Den Original-Frame (Farbschema egal, 2D oder 3D numpy-Array), die untere Schwelle (muss bei einem 3D-Array ein Skalar mit 3 Einträgen sein, bei einem 2D-Array einer mit 2 Einträgen) und die obere Schwelle. „inRange“ gibt den maskierten Frame als Kopie zurück:

```
# frame lesen
dieser_frame = kamera.get_frame()
frame_als_hsv = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2HSV)
# H S V
untere_schwelle = numpy.uint8([40, 80, 20])
obere_schwelle = numpy.uint8([80, 255, 255])
gruen_maske = cv2.inRange(frame_als_hsv, untere_schwelle, obere_schwelle)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", gruen_maske)
```

„imshow“ erkennt, dass das numpy-Array nur 2 Dimensionen hat und stellt das Bild korrekt dar:

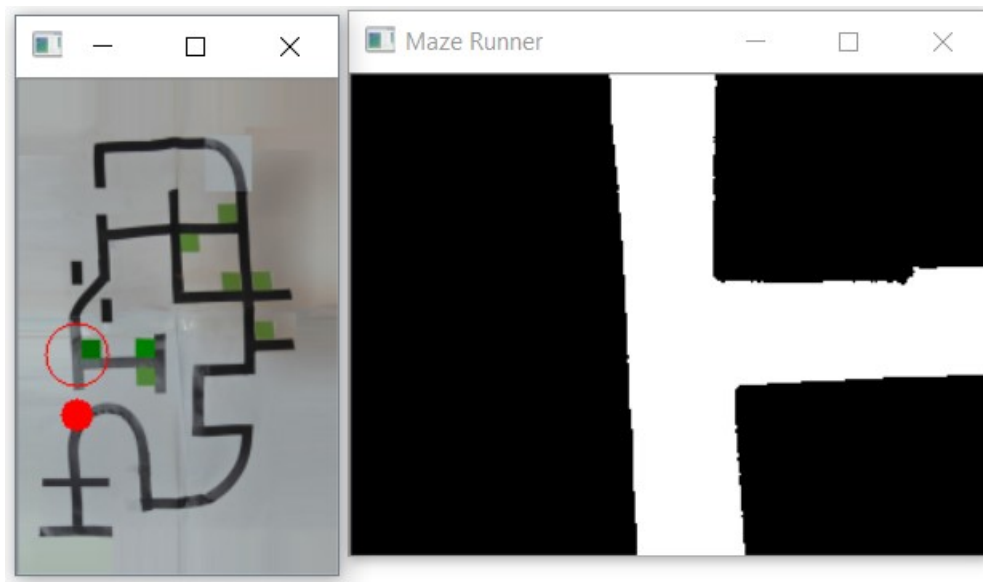


Das Beispielprogramm dazu heißt „test\_inRange.py“.

### Aufgabe:

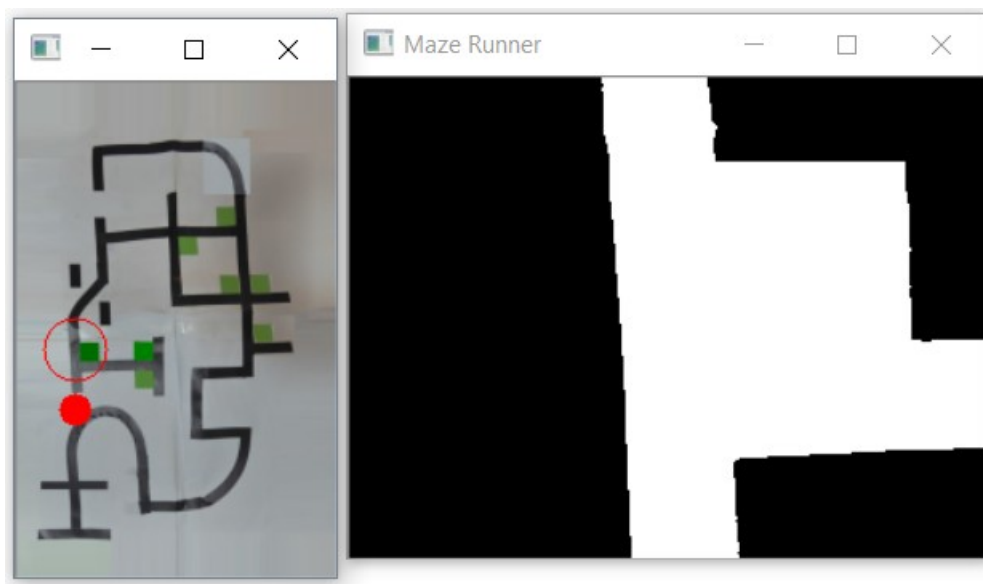
1. Kopiere „test\_inRange.py“ und erstelle Schwellenwerte, die nur schwarz maskieren.

Lösungsvorschlag: „aufgabe\_inRange1.py“



1. Kopiere „**aufgabe\_inRange1.py**“ und konvertiere den Frame in ein monochromes Bild, anstatt in HSV bevor du es maskierst. Die Schwellenwerte sind dabei nur noch Vektoren mit 1 Eintrag, also einfach einzelne Zahlen.

Lösungsvorschlag: „**aufgabe\_inRange2.py**“



2. Probiere alles nochmal mit der anderen Parcours-Datei „**parcours\_warm\_RESIZED.JPG**“, welche andere Beleuchtungsbedingung simuliert.

### 3.3.2.3 canny(... )

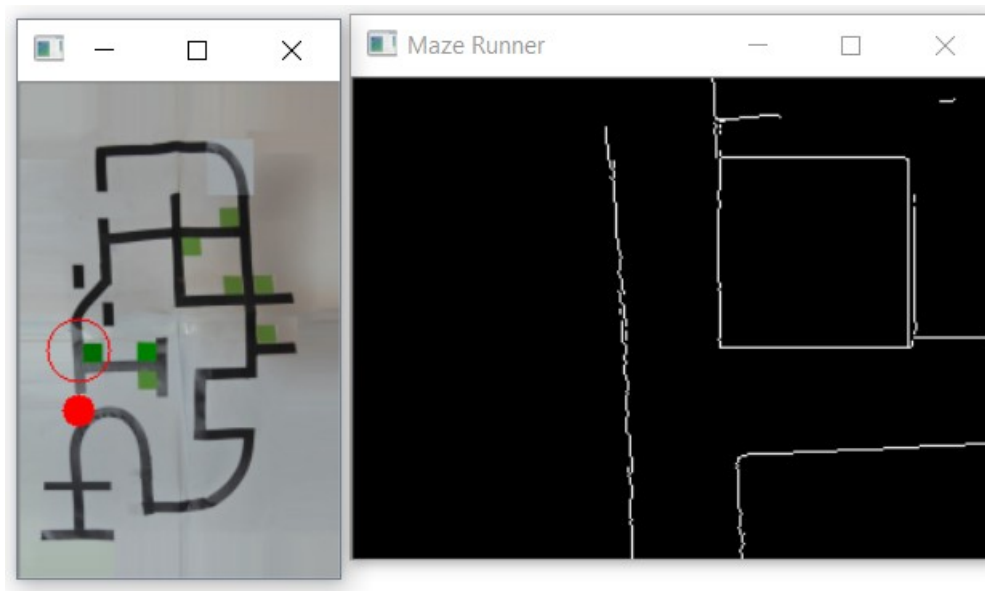
Siehe [https://docs.opencv.org/3.4.0/da/d22/tutorial\\_py\\_canny.html](https://docs.opencv.org/3.4.0/da/d22/tutorial_py_canny.html)

Canny Edge Detection maskiert alle Bereiche, deren Wert stark fluktuiert, z.B. am Übergang der Linie ist die Änderung der Helligkeit von sehr groß (weiß) zu sehr klein (schwarz) sehr groß, während auf weißen Flächen sich der Helligkeitswert nicht so stark ändert. Die Arbeitsweise ist also

ähnlich der bereits versuchten Adaptive Mean Threshold und Adaptive Gaussian Threshold Funktionen. „canny“ nimmt 3 Argumente auf: Den Original-Frame als 2D- oder 3D-numpy-Array und 2 Schwellenwerte. Die Schwellenwerte können denselben Wert haben, dann gibt es nur eine Schwelle: Alle Farb-Gradienten (Unterschiede) oberhalb der Schwelle werden detektiert, die darunter nicht. Warum jetzt also 2 Schwellen? Der Bereich dazwischen ist der Hysteresebereich. Erkannte Bereiche darin werden nur als Kanten erkannt, wenn sie mit einer Kante verbunden sind, die über der oberen Schwelle liegen. Man kann also eine Art „Graubereich“ definieren, der mal als Kante erkannt wird und mal nicht. „canny“ ist damit ein wenig intelligenter als „threshold“. Die 2. Schwelle muss natürlich immer größer sein als die erste.

```
# frame lesen
dieser_frame = kamera.get_frame()
# Edge Detection
edges = cv2.Canny(dieser_frame, 100, 150)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", edges)
```

In diesem Beispiel lassen wir den Algorithmus direkt über das farbige Bild laufen, d.h. wir detektieren nicht nur Helligkeits-, sondern auch Farbunterschiede:



Das Beispielprogramm dazu heißt „**test\_canny.py**“.

Das Beispielprogramm „**test\_mit\_canny.py**“ bietet weitere Tastenbefehle, um die Schwellen live anzupassen: „r“, „f“, „t“, „g“.

### Aufgaben:

1. Kopiere „**test\_canny.py**“ und konvertiere den Frame zuerst in ein monochromes Bild, bevor du „canny“ drüberlaufen lässt.

Lösungsvorschlag: „**aufgabe\_canny.py**“

2. Probiere alles nochmal mit der anderen Parcours-Datei „**parcours\_warm\_RESIZED.JPG**“, welche andere Beleuchtungsbedingung simuliert.

## 4 Zeichnen

Für die OpenCV-Preview ist es ganz praktisch einzuzeichnen, wo die Bilderkennung denn jetzt Objekte ausfindig gemacht hat. Dafür rendert man geometrische Objekte in den Frame.

Siehe [https://docs.opencv.org/3.0-beta/modules/imgproc/doc/drawing\\_functions.html](https://docs.opencv.org/3.0-beta/modules/imgproc/doc/drawing_functions.html)

**Hierbei ist folgendes zu beachten:** Wenn ein Teil der Bildauswertung die schwarze Linie auf weißem Grund detektiert und diese dann mit grünen Rechtecken markiert, sind diese auch in dem numpy-Array so gespeichert.

Wenn danach ein anderer Teil der Bildauswertung nach grünen Objekten sucht, wird dieser die gerade eingezeichneten grünen Rechtecke detektieren.

In diesem Fall muss also entweder die Reihenfolge getauscht werden oder das numpy-Array von dem Frame zwischendurch mal kopiert werden, damit nicht alle Programmteile mit einer Referenz auf denselben Frame arbeiten (siehe Kapitel 3.2).

### 4.1 Linie

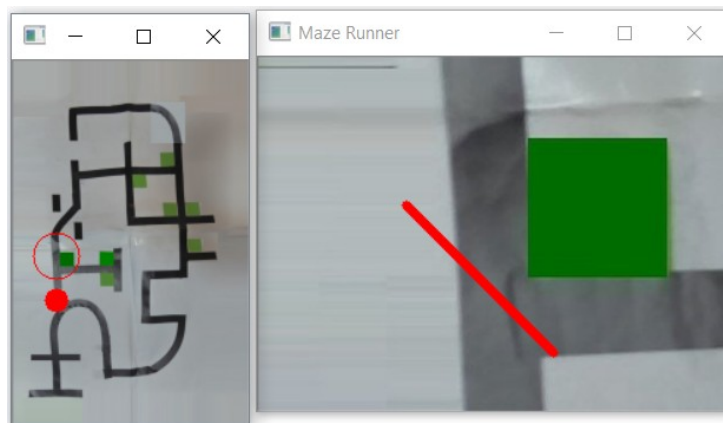
Um eine Linie zu zeichnen, benötigt man

- eine Startkoordinate (x von links in Pixeln, y von oben in Pixeln)
- eine Endkoordinate (x von links in Pixeln, y von oben in Pixeln)
- eine Farbe (B, G, R)
- Dicke der Linie in Pixel

Die Linie wird zwischen Startkoordinate und Endkoordinate gezeichnet:

```
# frame lesen
dieser_frame = kamera.get_frame()
# Linie zeichnen
start = (int(100), int(100))
ende = (int(200), int(200))
ROT = (0, 0, 255) # BGR
cv2.line(dieser_frame, start, ende, ROT, 5)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
```

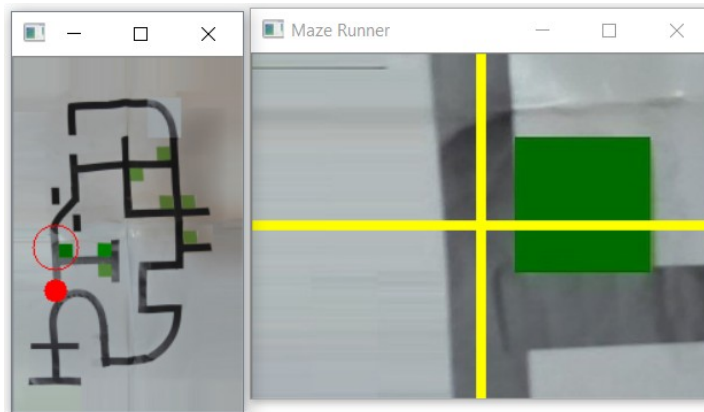
Das Beispielprogramm dazu heißt „**test\_line.py**“.



### Aufgabe:

1. Zeichne ein gelbes Fadenkreuz in die Mitte des Bildes.

Lösungsvorschlag: „aufgabe\_line.py“



## 4.2 Kreis

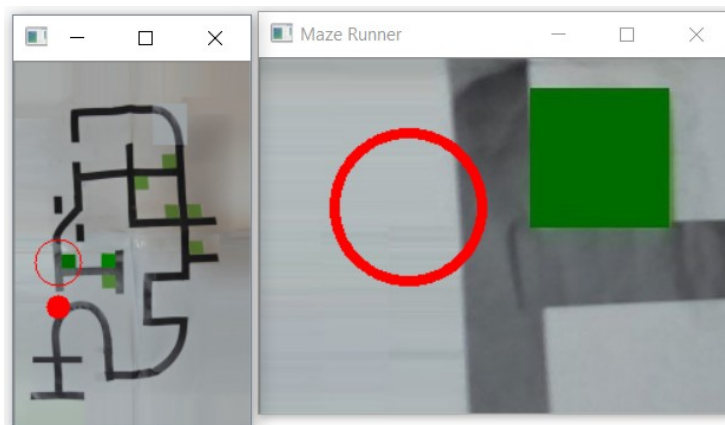
Um einen Kreis zu zeichnen, benötigt man

- eine Mittelpunkt-Koordinate (x von links in Pixeln, y von oben in Pixeln)
- reinen Radius in Pixeln
- eine Farbe (B, G, R)
- eine Linienstärke in Pixel (Sonderwert „-1“ füllt den Kreis komplett aus)

Der Kreis wird um den Mittelpunkt herum gezeichnet:

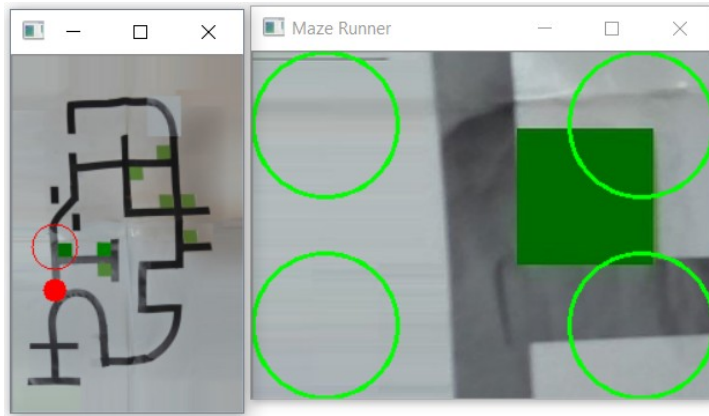
```
# frame lesen
dieser_frame = kamera.get_frame()
# Kreis zeichnen
zentrum = (int(100), int(100))
radius = 50
ROT = (0, 0, 255)
cv2.circle(dieser_frame, zentrum, radius, ROT, 5)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
```

Das Beispielprogramm dazu heißt „test\_circle.py“.



## Aufgabe:

1. Zeichne einen grünen Kreis in jede Ecke des Bildes (Radius dabei frei wählbar).  
Lösungsvorschlag: „aufgabe\_circle.py“



## 4.2 Rechteck

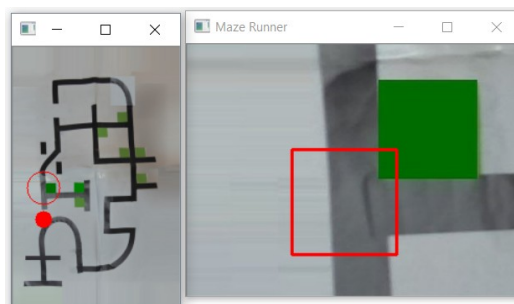
Um ein Rechteck zu zeichnen benötigt man

- eine Startkoordinate (x von links in Pixeln, y von oben in Pixeln)
- eine Endkoordinate (x von links in Pixeln, y von oben in Pixeln)
- eine Farbe (B, G, R)
- eine Linienstärke in Pixel (Sonderwert „-1“ füllt den Kreis komplett aus)

Das Rechteck wird „aufgezogen“ vom Startpunkt (linke obere Ecke des Rechtecks) zum Endpunkt (rechte untere Ecke des Rechtecks):

```
# frame lesen
dieser_frame = kamera.get_frame()
# Rechteck zeichnen
start = (int(100), int(100))
h = 100 # Höhe
b = 100 # Breite
# start[0] ist die X-Koordinate, start[1] die Y-Koordinate
ende = (int(start[0] + b), int(start[1] + h))
ROT = (0, 0, 255)
cv2.circle(dieser_frame, zentrum, radius, ROT, 5)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
```

Das Beispielprogramm dazu heißt „test\_rectangle.py“.

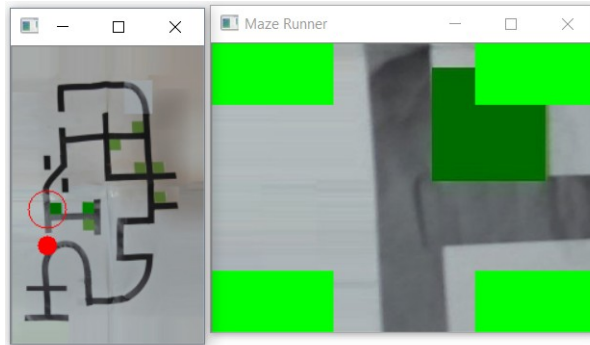




## Aufgabe:

1. Zeichne ein grünes ausgefülltes Rechteck in jede Ecke des Bildes (Höhe und Breite dabei frei wählbar).

Lösungsvorschlag: „**aufgabe\_rectangle.py**“



## 5 Auswertung

Bisher haben wir im Eye-Candy-Bereich gearbeitet. Es ist schön zu sehen, dass unsere Algorithmen bspw. die Linie oder die Kreuzungsmarkierung heraus-maskieren, aber um diese Daten in Motorsteuersignale umzuwandeln, müssen sie irgendwie zusammengefasst und vereinfacht werden, damit man bspw. eine Fallunterscheidung treffen kann oder die Stärke der Lenkbewegung errechnen.

Im Folgenden werden dafür vorgestellt:

- „**sum**“ (Summe der Werte in einem Array errechnen)
- „**findContours**“ (Rahmen um die Teile der Maske zeichnen, die nicht 0 sind)

Eine ganz andere Variante ist „**matchTemplate**“ (Beispiel-Bild im Gesamtbild erkennen und einrahmen). Das bei der geringen Komplexität der Strukturen im Parcours aber immer zu Fehldetektionen geführt. Verschwendet nicht eure Zeit damit...

### 5.1 Sum(... )

Die „**sum**“-Funktion ist gar keine Funktion von OpenCV, sondern eine von numpy. Die Nutzung geschieht folgendermaßen:

1. Bild maskieren (siehe Kapitel 3.3.2)
2. Einen Teil des Bildes ausschneiden (siehe Kapitel 3.1)
3. In dem Teilbild alle Pixelwerte aufaddieren
4. Fallunterscheidung treffen, ob das als „erkannt“ gilt oder nicht

Im Folgenden werden 2 diskrete Helligkeitssensoren simuliert, die frei auf dem Kamerabild positionierbar sind und entweder ein dunkles Areal detektieren oder eben nicht.

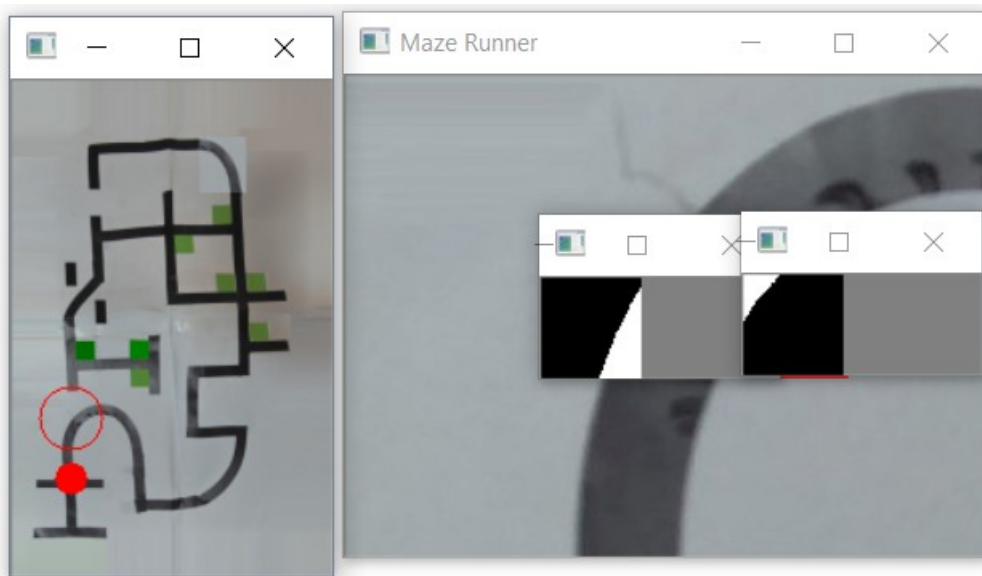
Eine mögliche Anwendung dieser Idee zeigt „**test\_mit\_threshold.py**“.

Nehmen wir das Beispiel aus Kapitel 3.3.2.1, bei dem mittels „**threshold**“ eine Maske von schwarzen Bereichen erstellt wird:

```
dieser_frame = kamera.get_frame()
frame_graustufen = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2GRAY)
schwelle = 120
_, linien_maske = cv2.threshold(frame_graustufen, schwelle, 255, cv2.THRESH_BINARY_INV)
```

Anschließend schneiden wir aus der Maske 2 kleine Bereiche heraus. Diese Bereiche heißen ROIs (Regions Of Interests). Die Stückchen sind die Sichtbereiche der virtuellen Helligkeitssensoren. Dabei ist zu beachten, dass der 1. Index (Zeilen) der Y-Koordinate entspricht und der 2. Index (Spalten) der X-Koordinate:

```
h = 50 # Höhe
b = 50 # Breite
start1 = (int(100), int(100))
# start[0] ist die X-Koordinate, start[1] die Y-Koordinate
ende1 = (int(start1[0] + b), int(start1[1] + h))
start2 = (int(200), int(100))
ende2 = (int(start2[0] + b), int(start2[1] + h))
# Rechteck von oben links (x=100, y=100) bis unten rechts (x=150, y=150) ausschneiden:
roi_links = linien_maske[start1[1]:ende1[1], start1[0]:ende1[0]]
# Rechteck von oben links (x=200, y=100) bis unten rechts (x=250, y=150) ausschneiden:
roi_rechts = linien_maske[start2[1]:ende2[1], start2[0]:ende2[0]]
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
cv2.imshow("links", roi_links)
cv2.imshow("rechts", roi_rechts)
```



Anschließend werden alle Werte in diesem Teilbereich aufsummiert, was zu einem einzelnen „Helligkeits“-Wert führt:

```
summe_links = roi_links.sum()
summe_rechts = roi_rechts.sum()
print([summe_links, summe_rechts])
```

```
PROBLEMS  TERMINAL  ...
[30600, 259590]
[22695, 297585]
[14025, 346290]
[7395, 392445]
[2550, 435795]
[510, 465630]
[0, 502095]
[0, 535245]
[0, 563550]
[0, 578595]
[0, 589815]
[0, 589050]
[0, 575025]
```

Für unsere Summierung ist es sinnvoll, wenn die weißen Teile der Maske nicht den Wert „255“, sondern „1“ aufweisen:

```
_, linien_maske = cv2.threshold(frame_graustufen, schwelle, 1, cv2.THRESH_BINARY_INV)
```

Alternativ kann man aber auch einfach durch 255 dividieren, um die Werte wieder auf 1 zu normieren.

```
summe_links = roi_links.sum() / 255  
summe_rechts = roi_rechts.sum() / 255  
print([summe_links, summe_rechts])
```

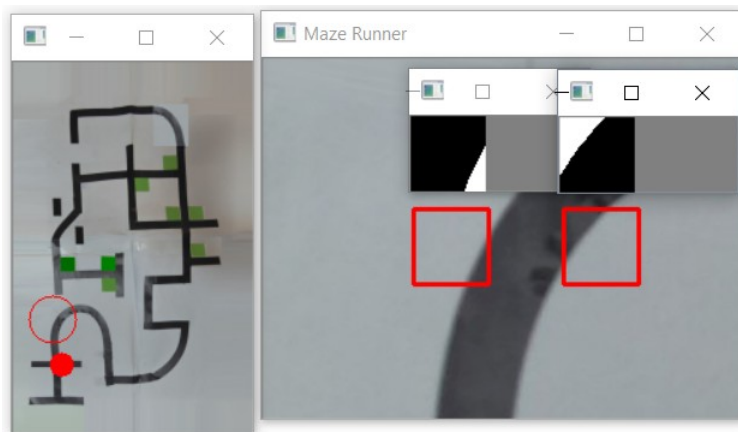
Dann entspricht die Summe wirklich der Anzahl der Pixel, die auf die Maske passen (also in unserem Fall Pixel, die auf der Linie liegen).

Das Beispielprogramm dazu heißt „**test\_sum.py**“.

### Aufgaben:

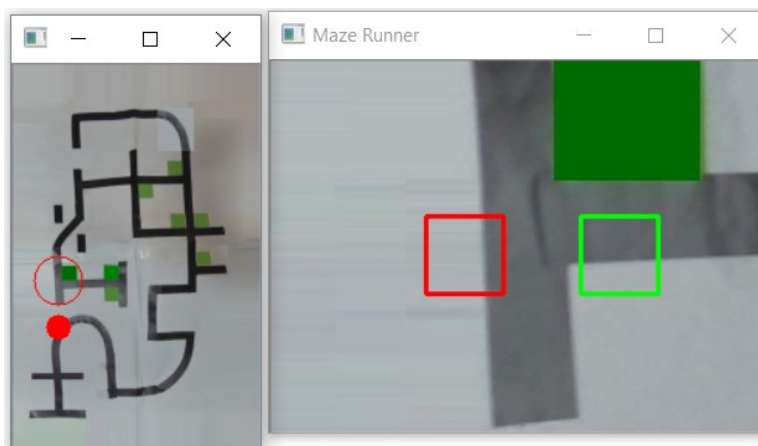
1. Zeichne die beiden Teilbereiche im Vorschaubild mittels Rechtecken ein (siehe Kapitel 4.2).

Lösungsvorschlag: „**aufgabe\_sum1.py**“



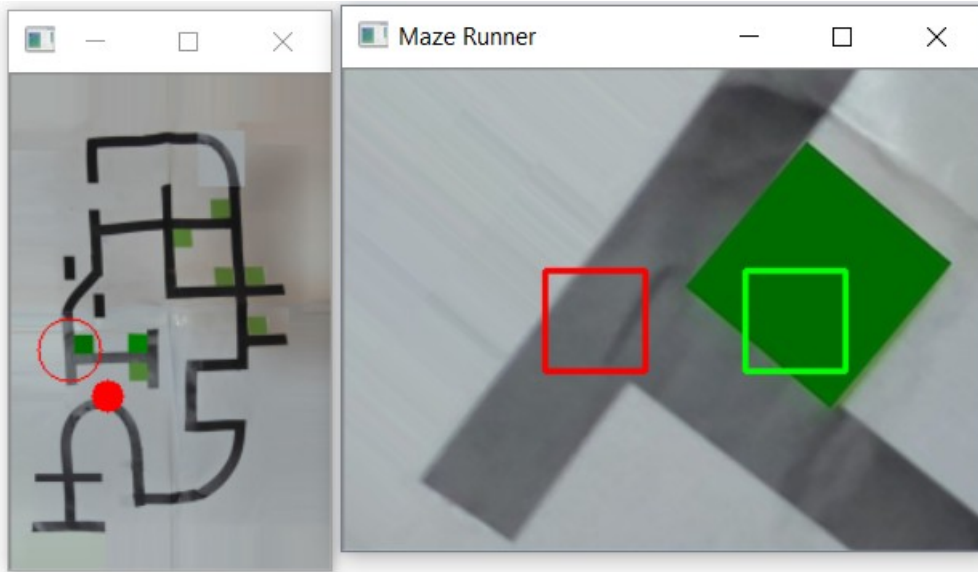
2. Verändere die Farbe der Rechtecke, wenn sie über einem dunklen Bereich gezeichnet sind, also die Linie detektieren müssen (siehe Kapitel 4.2).

Lösungsvorschlag: „**aufgabe\_sum2.py**“



3. Nutze dieselbe Methode mit „inRange“, um grüne Bereiche zu detektieren (siehe Kapitel 3.3.2.2).

Lösungsvorschlag: „aufgabe\_sum2.py“



4. Füge Motorsteuerungs-Befehle hinzu, um den simulierten Roboter auf der Linie zu halten.

## 5.2 findContours(... )

Siehe [https://docs.opencv.org/3.4.0/d3/d05/tutorial\\_py\\_table\\_of\\_contents\\_contours.html](https://docs.opencv.org/3.4.0/d3/d05/tutorial_py_table_of_contents_contours.html)

Mit der „sum“-Funktion sind wir auf die Bereiche im Frame beschränkt, die wir vordefinieren. Wir können also nur an bestimmten Positionen detektieren. „**findContours**“ ermöglicht es, die weißen (also detektierten) Teile der Maske „einzurahmen“ und deren Positionen sowie Größe zu erhalten.

Die Nutzung geschieht folgendermaßen:

1. Bild maskieren (siehe Kapitel 3.3.2)
2. Konturen extrahieren mittels „**findContours**“
3. Zu kleine Konturen herausfiltern mittels „**contourArea**“
4. Geometrie der Konturen vereinfachen, indem ein „Bounding Rectangle“ drumrumgezeichnet wird mittels „**boundingRect**“.
5. Fallunterscheidung treffen basierend auf Position und Größe des Bounding Rectangle.

Im Folgenden werden alle grünen Bereiche im Frame ab einer bestimmten Größe detektiert.

Eine mögliche Anwendung dieser Idee zeigt „**test\_mit\_inRange.py**“.

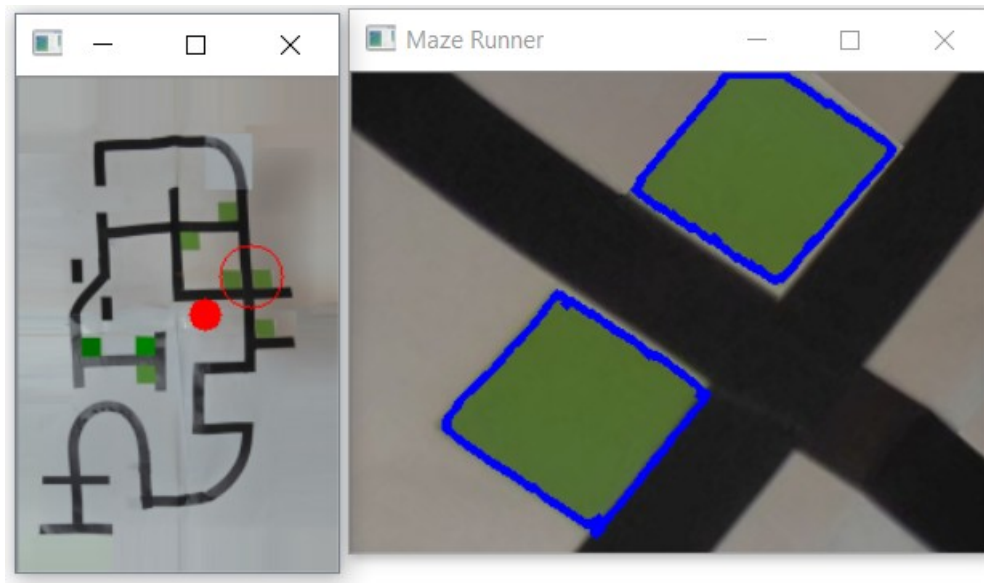
Nehmen wir das Beispiel aus Kapitel 3.3.2.2, bei dem mittels „**inRange**“ eine Maske von grünen Bereichen erstellt wird:

```
dieser_frame = kamera.get_frame()
frame_als_hsv = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2HSV)
# H S V
untere_schwelle = numpy.uint8([40, 80, 20])
obere_schwelle = numpy.uint8([80, 255, 255])
gruen_maske = cv2.inRange(frame_als_hsv, untere_schwelle, obere_schwelle)
```

Wir lassen „findContours“ drüber laufen und visualisieren alle gefundenen Kanten, indem wir sie mit Linien nachzeichnen. Das ist für unsere Kreuzungserkennung nicht notwendig, sondern demonstriert nur die Arbeitsweise von „findContours“:

„findContours“ gibt 3 Rückgabewerte zurück, wovon uns nur der 2. interessiert (deswegen die Platzhalter „\_“. Der 2. Rückgabewert ist eine Liste der gefundenen geschlossenen Konturen. Jede Kontur besteht aus einer Liste aus Punkten, die verbunden ein Polygon um die Kontur herum ergeben. Die Anzahl dieser Punkte kann durch die Argumente (im Beispiel „RETR\_TREE“ und „CHAIN\_APPROX\_SIMPLE“) beeinflusst werden. Wir benutzen eine Einstellung, die uns sehr wenige Punkte zurückgibt. Das spart RAM und geht schneller zu verarbeiten, ist aber ungenauer:

```
# Konturen extrahieren
_, konturen, _ = cv2.findContours(gruen_maske, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
# Konturen (Polygone) in blau einzeichnen
# („-1“ bedeutet alle Konturen, „3“, „2“ ist die Dicke der einzuzeichnenden Linie)
BLAU = (255, 0, 0)
cv2.drawContours(dieser_frame, konturen, -1, BLAU, 2)
```



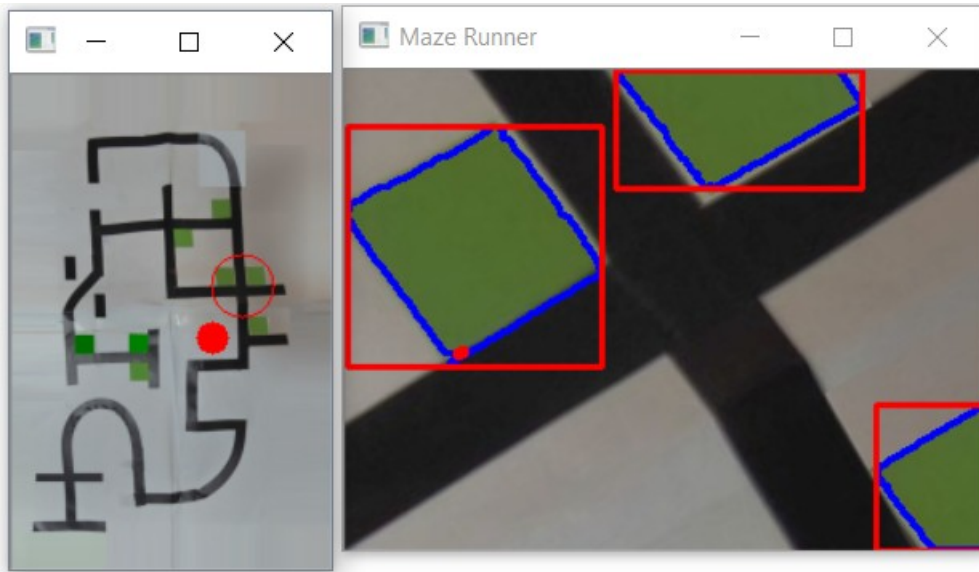
Die Geometrien der Konturen sind recht komplex und schwer zu verarbeiten (man sieht die Umrandungen sind recht „zittrig“ und können auch rotieren). Zur Vereinfachung kann ein Rahmen um jede geschlossene Kontur gezeichnet werden, sodass man es in der Auswertung geometrisch nur noch mit Position und Dimension des Rechteckes zu tun hat.

„boundingRect“ nimmt eine Kontur als Argument auf und gibt 4 Rückgabewerte zurück: X-/Y-Position des Startpunktes des Rechteckes (linke obere Ecke), Breite und Höhe. Diese Geometrie lässt sich einfach in den Frame einzeichnen (siehe Kapitel 4.2). Da „boundingRect“ immer nur eine Kontur gleichzeitig verarbeiten kann, müssen wir über alle gefundenen Konturen iterieren. Dazu bietet sich eine „for... in...“-Schleife an:

```

for kontur in konturen:
    # Dimensionen des Rechtecks um die Kontur extrahieren
    x, y, w, h = cv2.boundingRect(kontur)
    # Rechteck um die Kontur zeichnen
    cv2.rectangle(dieser_frame, (x, y), (x + w, y + h), ROT, 2)

```

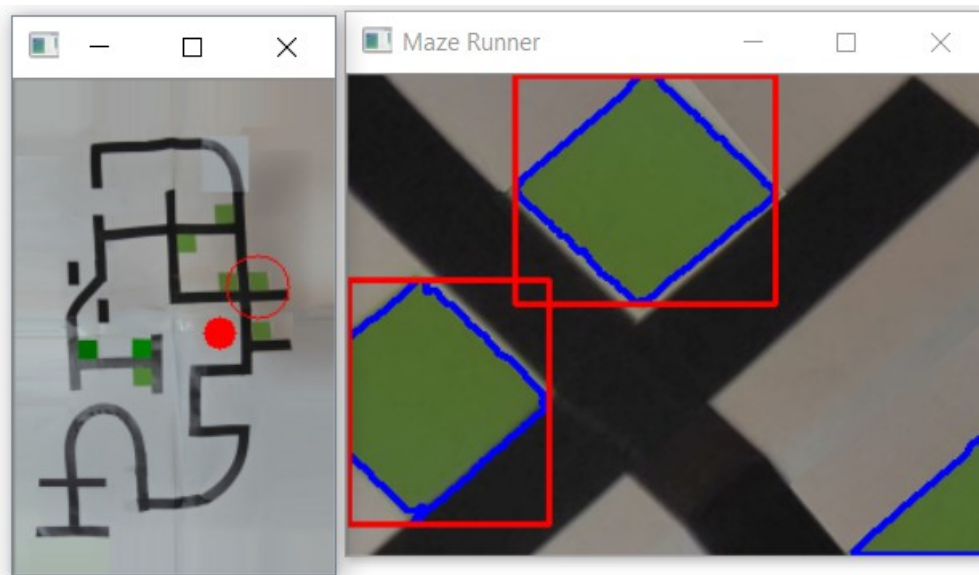


Jetzt fällt auf, dass neben den großen erkannten Konturen noch kleine existieren (z.B. der rote „Punkt“ in dem Bild oben). Um nicht auf kleine Fehler im Bild (z.B. Bildrauschen, schlechte Beleuchtung etc.) zu reagieren, können Konturen nach Größer der beinhaltenden Fläche gefiltert werden. Dazu kann „contourArea“ genutzt werden, welche eine Kontur als Argument aufnimmt und die Anzahl der eingeschlossenen Pixel, also die Fläche der Kontur, zurückgibt:

```

# zu kleine Konturen wegfiltern
if cv2.contourArea(kontur) > 4000:
    x, y, w, h = cv2.boundingRect(kontur)
    cv2.rectangle(dieser_frame, (x, y), (x + w, y + h), ROT, 2)

```



Jetzt ist der kleine Punkt und auch die nur teilweise sichtbare Kreuzungsmarkierung unten links weggefiltert.

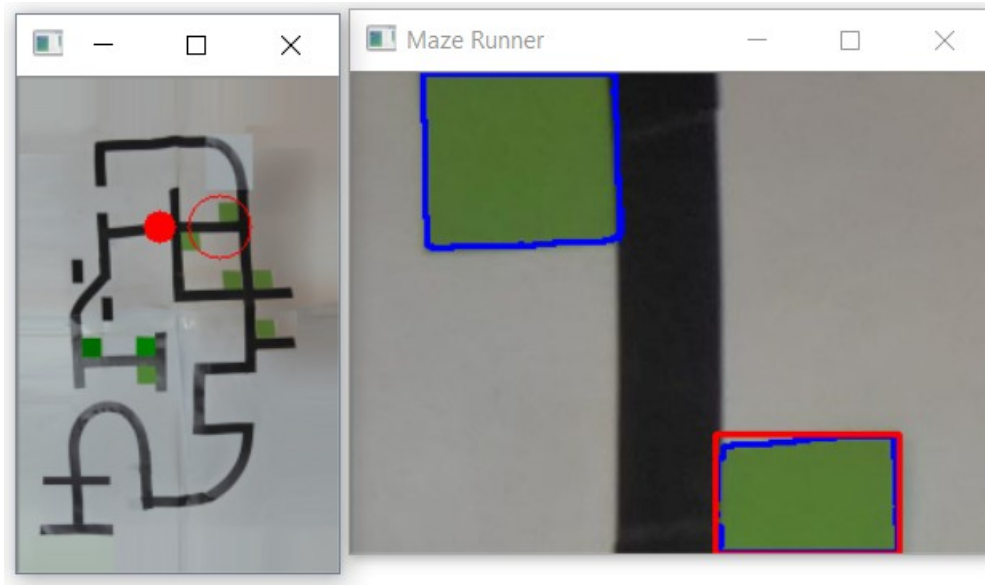


Das Beispielprogramm dazu heißt „**test\_findContours.py**“.

### Aufgaben:

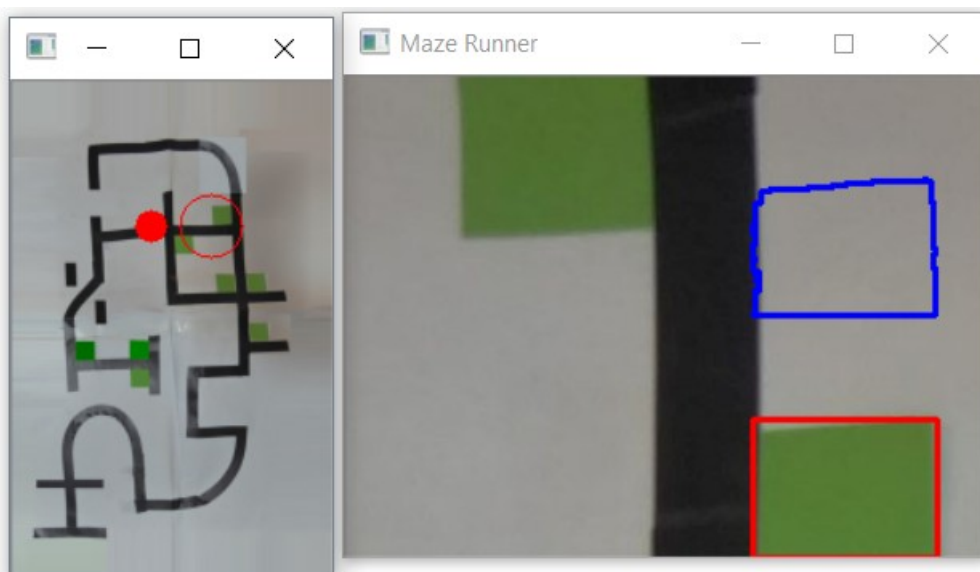
1. Filtere alle gefundenen Bounding Rectangles heraus, die in der oberen Hälfte des Frames liegen, sodass nur grüne Bereiche nach am Roboter erkannt werden.

Lösungsvorschlag: „**aufgabe\_findContours1.py**“



2. Schneide aus dem unteren Bereich des Frames einen Streifen heraus und nutze diesen für die Grünerkennung (siehe Kapitel 3.1). Auch in diesem Fall werden nur grüne Bereiche nah am Roboter erkannt, aber der Algorithmus läuft insgesamt schneller, **aber die Konturen und erkannten Bounding Rectangles werden an der falschen Stelle angezeigt, da sich die Koordinaten der erkannten Konturen ja nur auf die untere Bildhälfte beziehen und nicht auf den Gesamt-Frame. Für die Rechtecke, die wir selbst einzeichnen, kann man das reparieren, indem man sie um Bildhöhe / 2 weiter nach unten versetzt.**

Lösungsvorschlag: „**aufgabe\_findContours2.py**“



## 5.4 Bilder Kombinieren

Eventuell möchte man mehrere der oben gezeigten Methoden anwenden und irgendwie miteinander kombinieren.

Z.B.

1. Mit „threshold“ eine Maske der dunklen Bereiche erstellen (siehe Kapitel 3.2.2.1). In dieser Maske sind grüne Bereiche mit enthalten, da sie typischerweise recht dunkel sind.
2. Mit „inRange“ eine Maske der grünen Bereiche erstellen (siehe Kapitel 3.2.2.2). In dieser Maske sind nur grüne Bereiche enthalten, aber sie ist recht empfindlich gegen schlechte Lichtbedingungen, bei denen die echte Kamera einen starken Farbstich bekommt.
3. Die Maske der grünen Bereiche von der Maske der dunklen Bereiche abziehen, damit in der Maske der dunklen Bereiche wirklich nur noch schwarze Areale übrig bleiben.

Erstellen wir zuerst die Maske der dunklen Bereiche:

```
# Kopie von dieser_frame in Graustufen konvertieren
frame_graustufen = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2GRAY)
# alle Werte in dieser_frame bei Schwellenwert 120 in 0 oder 255 zerteilen, Bild
invertieren
_, linien_und_kreuzungen = cv2.threshold(frame_graustufen, 120, 255, cv2.THRESH_BINARY_INV)
```

Dann eine Maske der grünen Bereiche:

```
frame_als_hsv = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2HSV)
# H S V
untere_schwelle = numpy.uint8([40, 80, 20])
obere_schwelle = numpy.uint8([80, 255, 255])
kreuzungen = cv2.inRange(frame_als_hsv, untere_schwelle, obere_schwelle)
```

Schließlich subtrahieren wir die Masken voneinander. Wie in Kapitel 3.1 angedeutet, kann mit numpy-Arrays ja gerechnet werden. Die Rechenoperationen betreffen dann jeden Eintrag des arrays:

```
linien = linien_und_kreuzungen - kreuzungen
```

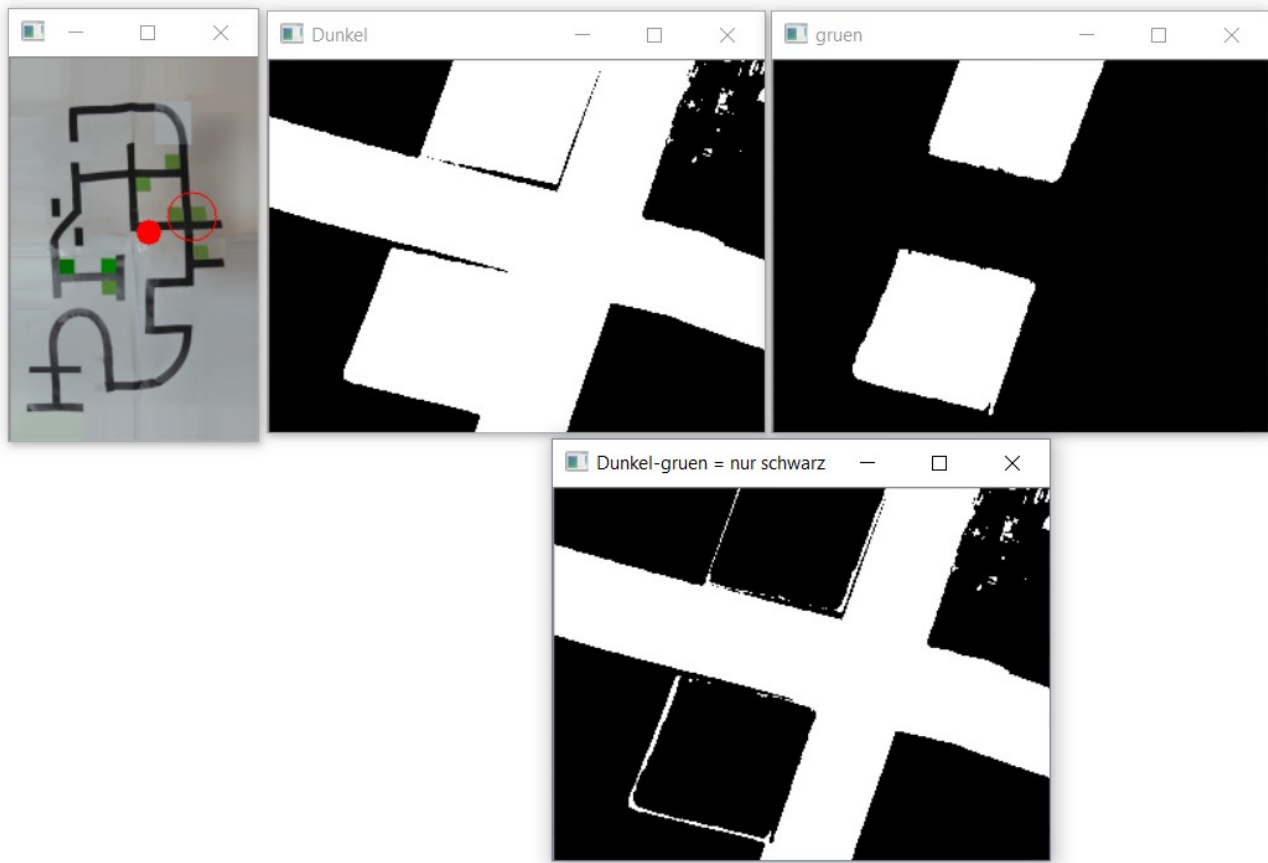
Es gibt dafür auch noch andere Methoden, die OpenCV zur Verfügung stellt (z.B. „bitwise\_and“).

Siehe [https://docs.opencv.org/3.4.0/df/d9d/tutorial\\_py\\_colorspaces.html](https://docs.opencv.org/3.4.0/df/d9d/tutorial_py_colorspaces.html)

Auf diese wird hier aber nicht weiter eingegangen.

Das Beispielprogramm dazu heißt „**test\_combine.py**“.





## 6 Realität

In der Realität sind die Beleuchtungsbedingung nicht so schön gleichmäßig wie in der Simulation. Es müssen also vermutlich folgende Maßnahmen getroffen werden:

- Kamera-Sichtbereich vor direkter Einstrahlung der Deckenbeleuchtung schützen, da dies abhängig vom Material des Parcours zu Reflektionen führen kann, die die Erkennung stören.
- Kamera-Sichtbereich aktiv ausleuchten, um auch bei schlechter Beleuchtung noch Farben korrekt erkennen zu können.
- Kalibrierung mit den Lichtbedingungen des Parcours. Wie so etwas aussehen kann, wird in „test\_mit\_canny.py“ gezeigt.
- Eingangsbild mit den Funktionen der PiCam aufwerten (Helligkeit, Kontrast, Farbtönung usw.)
- Weitere Bearbeitung des Bildes, um Bildrauschen, geringe Sättigung, Schattenwürfe usw. zu minimieren. Dazu im folgenden Kapitel mehr:

### 6.1 Erode(... ) / Dilate(... )

Siehe [https://docs.opencv.org/3.4.0/d9/d61/tutorial\\_py\\_morphological\\_ops.html](https://docs.opencv.org/3.4.0/d9/d61/tutorial_py_morphological_ops.html)

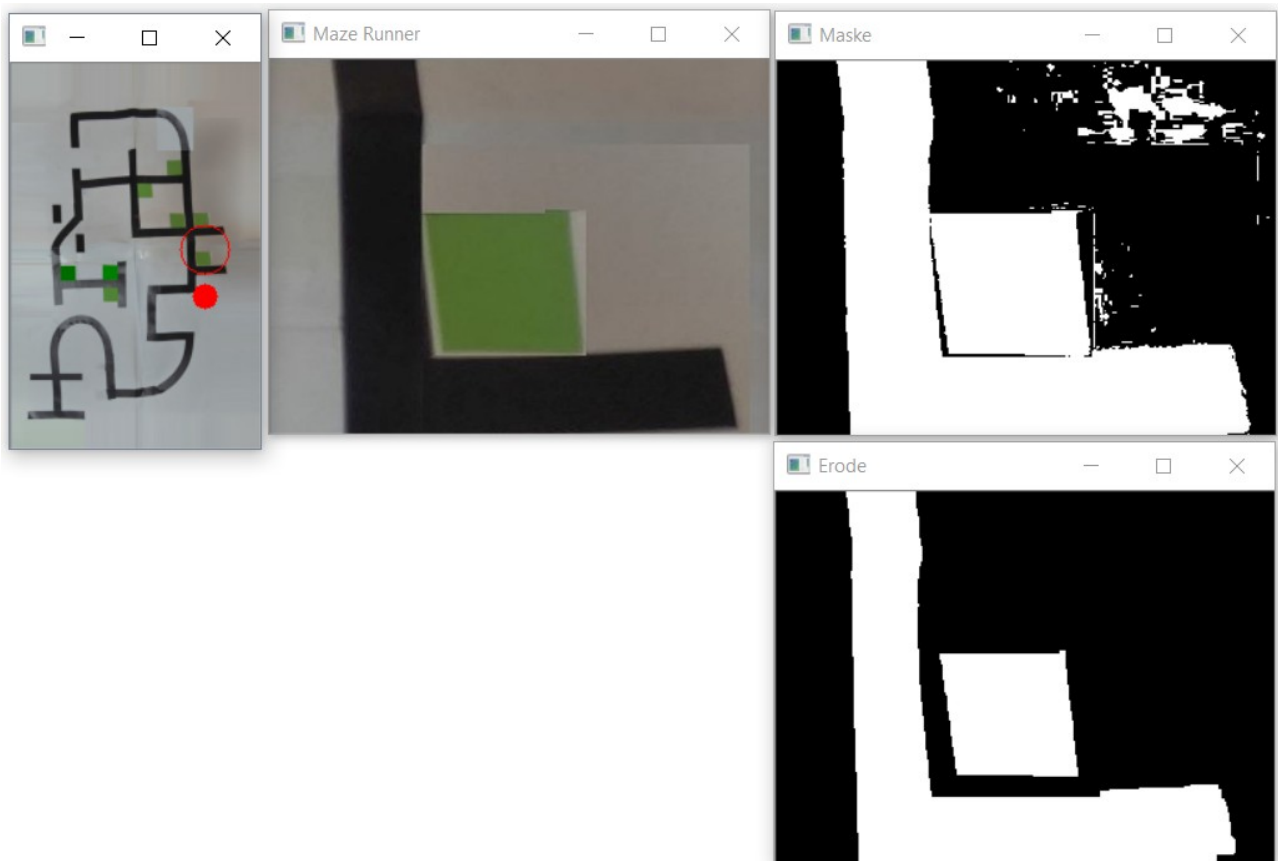
„**erode**“ verjüngt zusammenhängende Bereiche. Damit kann zum Beispiel die maskierte Linie des Parcours schmaler gemacht oder kleiner Fehler in der Maske entfernt werden.

Zum Beispiel könnten wir eine Maske der dunklen Bereiche mit „threshold“ erzeugen (siehe Kapitel 3.2.2.1). Die Schwelle (120) ist so gewählt, dass auch eher helle Bereiche als Linie erkannt werden, damit die Linienbereiche, die durch direkte Beleuchtung stark reflektieren, auch erkannt werden.

Der „kernel“ ist hierbei die Größe des Bereichs, die der „erode“-Algorithmus gleichzeitig abfertigt, im folgenden Beispiel 5x5 Pixel. Dieser sollte angepasst werden auf die Strukturgröße der zu entfernenden Bereiche. „iterations“ bestimmt wie oft der „erode“-Algorithmus angewendet wird; in diesem Beispiel 3x:

```
dieser_frame = kamera.get_frame()
frame_graustufen = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2GRAY)
_, frame_schwarz_weiss = cv2.threshold(frame_graustufen, 120, 255,
cv2.THRESH_BINARY_INV)
# „Scan-Bereich“ auf 5x5 Pixel einstellen
kernel = numpy.ones((5, 5), numpy.uint8)
# 3x jeweils 5 Pixel von allen Kanten abfressen:
erosion = cv2.erode(frame_schwarz_weiss, kernel, iterations = 3)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
cv2.imshow("Maske", frame_schwarz_weiss)
cv2.imshow("Dilate", dilation)
```

Das Beispielprogramm dazu heißt „test\_erode.py“.



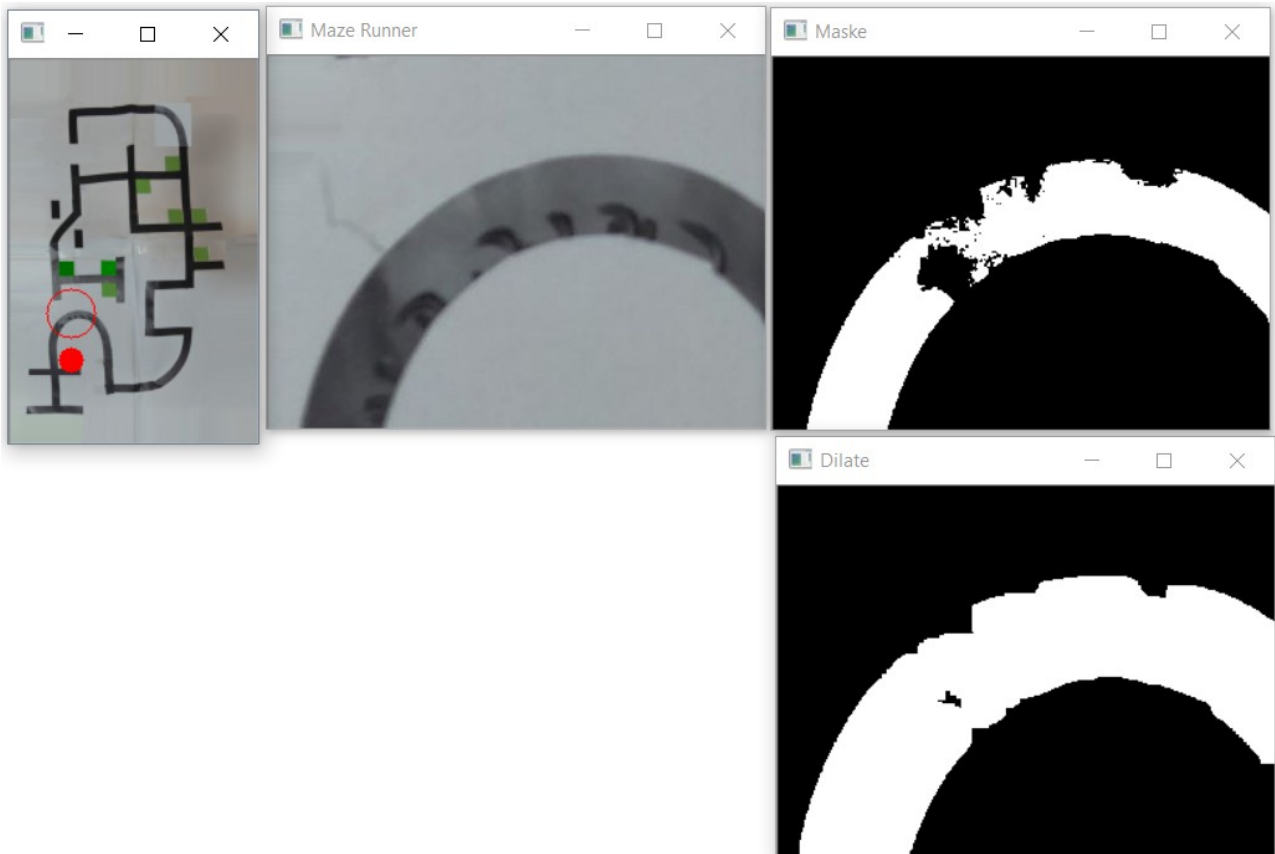
Das Fenster „Maske“ zeigt einige Fehler in der Maske. Das sind Bereiche, die auch sehr dunkel sind und somit via „threshold“ Teil der Maske wurden. Diese können mit „erode“ nahezu entfernt werden, da sie nur eine kleine Fläche besitzen (Fenster „Erode“).

„**dilate**“ bewirkt das Gegenteil. Damit kann zum Beispiel die maskierte Linie des Parcours dicker gemacht oder nicht zusammenhängende Bereiche der Maske verbunden werden.

Wie im vorherigen Beispiel erstellen wir eine Maske. Diesmal ist die Schwelle (100) ist so gewählt, dass in dunklen Bereichen keine Fehlerkennungen statt finden wie im Beispiel oben. Dann können Linienbereiche, die durch direkte Beleuchtung stark reflektieren, aber nicht mehr erkannt werden.

```
dieser_frame = kamera.get_frame()
frame_graustufen = cv2.cvtColor(dieser_frame, cv2.COLOR_BGR2GRAY)
_, frame_schwarz_weiss = cv2.threshold(frame_graustufen, 120, 255,
cv2.THRESH_BINARY_INV)
# „Scan-Bereich“ auf 5x5 Pixel einstellen
kernel = numpy.ones((5, 5), numpy.uint8)
# 3x jeweils 5 Pixel an alle Kanten drappappen:
dilation = cv2.dilate(frame_schwarz_weiss, kernel, iterations = 4)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
cv2.imshow("Maske", frame_schwarz_weiss)
cv2.imshow("Dilate", dilation)
```

Das Beispielprogramm dazu heißt „**test\_dilate.py**“.



Der nun geringere Schwellenwert führt dazu, dass die stark reflektierenden Bereiche der Linie nicht mehr vorständig erkannt werden. Durch „dilate“ wachsen dieser wieder zusammen und die Linie wird insgesamt dicker.

## 6.2 Blur

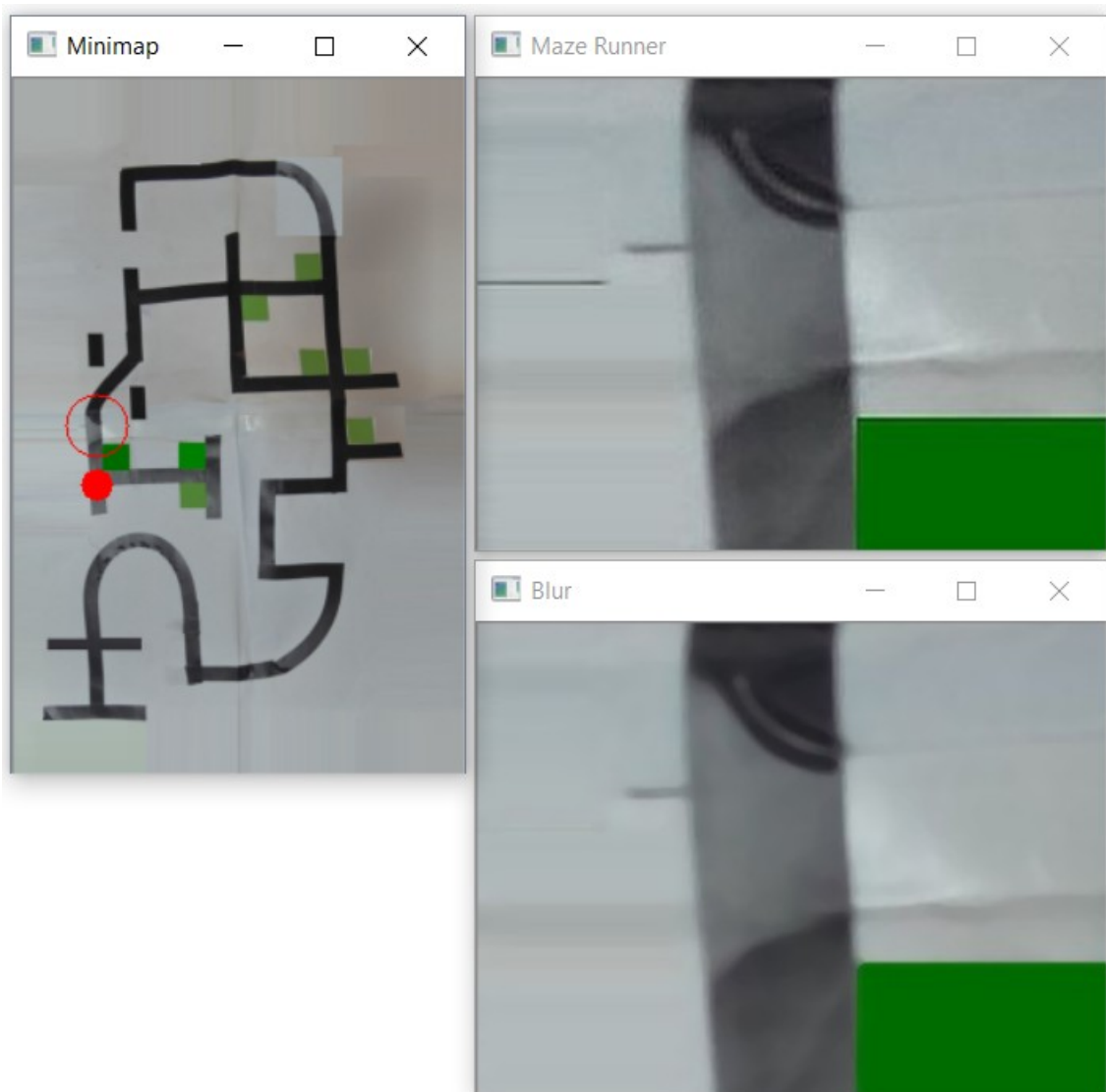
Siehe [https://docs.opencv.org/3.4.0/d4/d13/tutorial\\_py\\_filtering.html](https://docs.opencv.org/3.4.0/d4/d13/tutorial_py_filtering.html)

Besonders bei schlechten Beleuchtungsverhältnissen ist das Eingangsbild recht verrauscht, d.h. einzelne Pixel haben eine andere Farbe. Es entsteht ein „Punktmuster“. Das kann mit einem Verwaschungs-Filter wieder geglättet werden. Ich empfehle den „medianBlur“, da dieser recht effizient ist und Rauschen gut unterdrückt. Andere Varianten sind in dem Link oben beschrieben.

„**medianBlur**“ nimmt 2 Argumente auf: Den Original-Frame und die Größe des Kernels in Pixeln. Umso größer der Kernel (im Beispiel 7), umso weichgezeichneter. Der Kernel muss eine ungerade Zahl sein.

```
dieser_frame = kamera.get_frame()
# Bild mit Mittelwerten glätten
blur = cv2.medianBlur(dieser_frame,9)
# Kamerabild anzeigen
cv2.imshow("Maze Runner", dieser_frame)
cv2.imshow("Blur", blur)
```

Das Beispielprogramm dazu heißt „**test\_blur.py**“.



Wenn man genau hinschaut, existiert im Original-Frame „Maze Runner“ auf der linken Seite eine „Bildstörung“, die es im Frame „Blur“ nicht mehr gibt. Diese wurde durch das Weichzeichnen entfernt.