

Projekt systemu wizyjnego

Spis treści

1 Wstęp	2
1.1 Analiza problemu	2
1.1.1 Hardware	2
1.1.2 Software	2
1.2 Narzędzia i kluczowe biblioteki	2
1.2.1 Narzędzia	2
1.2.2 Biblioteki	3
2 Konfiguracja systemu	4
2.1 Instalacja OS	4
2.2 Połączenie z płytka	5
2.2.1 Nawiązywanie połączenia z Wi-Fi	5
2.2.2 Weryfikacja połączenia	5
2.3 Aktualizacja i zmiana ustawień systemowych	5
2.3.1 Software	5
2.3.2 Ustawienia systemu	6
2.4 Weryfikacja konfiguracji	7
3 Software	8
3.1 Idea algorytmu	8
3.2 Implementacja	8
3.2.1 Wstępne przetwarzanie obrazu	8
3.2.2 Wykrycie krawędzi	9
3.2.3 Wydobycie obszarów zawierających bloki	9
3.3 Wnioski	10
3.4 Przyszły rozwój projektu	10
4 Dodatek	11

1 Wstęp

Celem projektu było stworzenie embeddowego układu wizyjnego służącego do identyfikacji punktów charakterystycznych kostki Rubika. Pozyskane informacje byłyby przetwarzane i przekazywane dalej dla układu sterującego robotem/zestawem serw, które dalej "rozwiązywałyby" kostkę.

1.1 Analiza problemu

1.1.1 Hardware

Z punktu widzenia hardware'u układ musi spełniać poniższe warunki:

- Obecność portu umożliwiającego komunikację z kamerą
- Stosunkowo duża ilość pamięci RAM - przetwarzanie obrazów jest kosztowne, zatem mikrokontroler/komputer powinien mieć pamięć rzędu setek kB
- Stosunkowo duża ilość pamięci nieulotnej - biblioteki jak i same obrazy zajmują dużo miejsca, należałoby rozważyć rozwiązania posiadające co najmniej kilka GB.

Systemem spełniającym powyższe jest użyte w projekcie **Raspberry Pi Zero W**. Do układu dołączono kamerę **OV5647** oraz pamięć zewnętrzną w postaci karty SD.

1.1.2 Software

Zarys programu można przedstawić jako:

1. Pozyskanie obrazu z kamery
2. Wstępne przetworzenie obrazu
3. Pozyskanie kluczowych informacji z obrazu
4. Zapis/Przekazanie danych do układu sterowania

Jeśli chodzi o dane mogące być informacją zwrotną:

- Wykrycie segmentów kostki - ich kolor, ilość, umiejscowienie na ścianie
- Wykrycie wierzchołków kostki
- Wykrycie orientacji kostki

Potencjalne problemy związane z implementacją systemu wizyjnego to m.in.:

- Zniekształcenia obrazu wynikające z soczewki kamery
- Wpływ oświetlenia i otoczenia na wyniki przetwarzania

1.2 Narzędzia i kluczowe biblioteki

1.2.1 Narzędzia

IDE Jako IDE posłużyło *Visual Studio Code 2019*

Remote computing toolbox Jako łącznik między Raspberry oraz maszynami wirtualnymi użyto *Mobaxterm*. Realizowano w niej głównie sesje SSH oraz VNC.

Instalator OS Do zainstalowania systemu operacyjnego na Raspberry użyto dedykowanego programu *Raspberry Pi Imager*.

Wirtualizacja Do testowania oprogramowania 'offline' użyto *Oracle VM Virtualbox*, na którym hostowano maszynę wirtualną z obrazem Ubuntu 20.04

1.2.2 Biblioteki

OpenCV i Numpy Do przetwarzania obrazów użyto *OpenCV*, która jest dość standardowa w tego typu aplikacjach. Niektóre operacje wymagały użycia *Numpy*, jednak jest ona obecnie domyślnie w Raspberry OS. Instalacja:

```
sudo apt-get install python-opencv
```

Weryfikacja instalacji za pomocą pythona:

```
>>> import cv2  
>>> print(cv2.__version__)
```

Otrzymany output:

```
4.5.1
```

Git Do kontrolowania wersji projektu użyto *Gita*

```
sudo apt install git
```

Weryfikacja instalacji:

```
git --version
```

Output:

```
git version 2.30.2
```

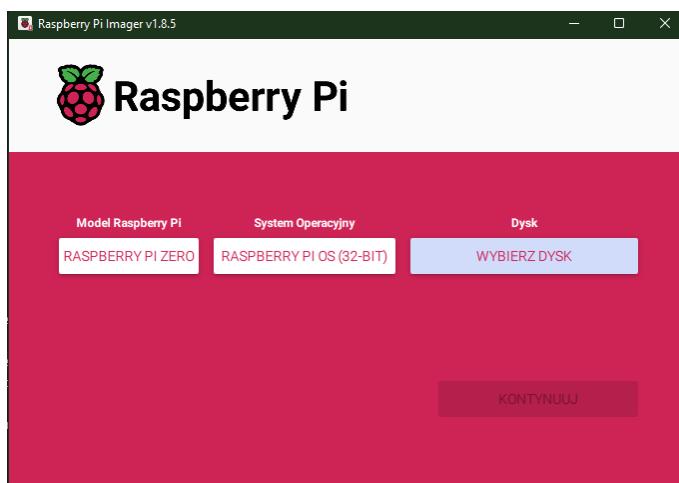
2 Konfiguracja systemu

By móc stworzyć oprogramowanie pozwalające na zrealizowanie zadania należało odpowiednio przygotować i skonfigurować płytę. Podjęte kroki:

1. Instalacja systemu operacyjnego
2. Ustanowienie połączenia płytki - komputer
3. Aktualizacja oprogramowania i zmiana ustawień systemowych
4. Weryfikacja działania systemu

2.1 Instalacja OS

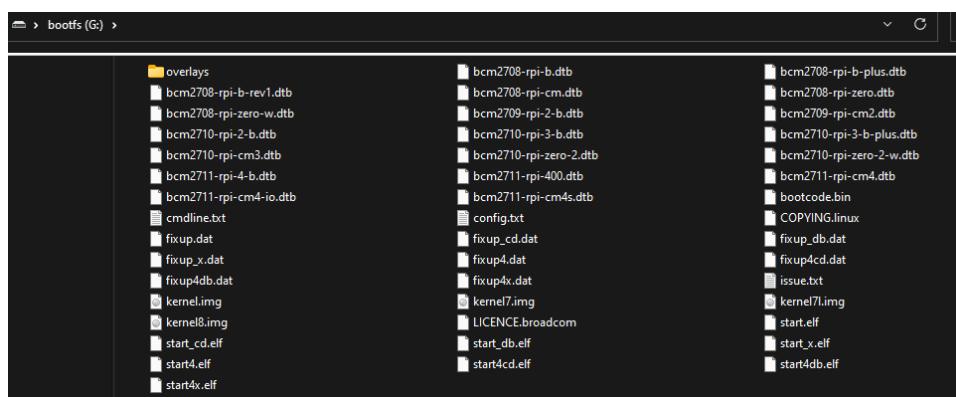
Jak wcześniej wspomniano do instalacji OSu użyto *Raspberry Pi Imager*



Rys. 2.1: Podstawowe opcje konfiguracyjne

Dodatkowo podczas instalacji włączono SSH, nadano nazwę użytkownika oraz przypisano hasło - pozwoli to na połączenie się z płytą w przyszłych krokach.

Po skończeniu instalacji zawartość karty SD wygląda następująco:



Rys. 2.2: Organizacja plików

2.2 Połaczanie z płytka

Łączenie z płytką postanowiono nawiązać z użyciem SSH. W tym celu oba urządzenia (komputer i płytka) muszą być w tej samej sieci, w tym przypadku zrealizowano to przez konfigurację sieci Wi-Fi. Należy uwzględnić fakt, że użyta wersja Raspberry obsługuje jedynie pasmo **2.4GHz**.

2.2.1 Nawiązywanie połączenia z Wi-Fi

By płytka automatycznie łączyła się z siecią Wi-Fi konieczne jest umieszczenie dwóch plików w głównym folderze karty SD.

wpa_supplicant.conf Jest plikiem, który definiuje parametry sieci Wi-Fi, do której będzie się łączyć płytka. Jego treść powinna zawierać:

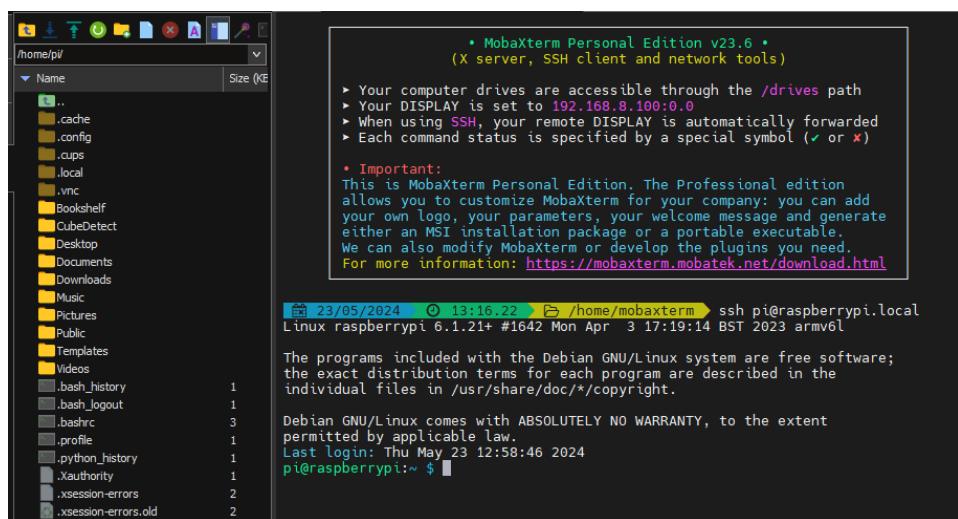
```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=PL

network={
    ssid="WIFI_name"
    psk="WIFI_password"
}
```

ssh Drugim koniecznym plikiem jest pusty plik bez rozszerzenia o nazwie *ssh*. Uruchamia on usługę ssh na płytce.

2.2.2 Weryfikacja połączenia

Po odczekaniu 1-2min od podłączenia urządzenia do zasilania, spróbowano nawiązać połączenie z programu *Mobaxterm*.



Rys. 2.3: Widoczna udana próba połączenia się

Widoczne jest, że konfiguracja połączenia została wykonana poprawnie. Dodatkowo w lewym panelu programu można zauważać pliki i katalogi znajdujące się na płytce.

2.3 Aktualizacja i zmiana ustawień systemowych

2.3.1 Software

By uniknąć problemów instalacyjnych z bibliotekami należy w pierwszej kolejności uaktualnić software:

```
sudo apt update
sudo apt full-upgrade
```

Następnie należy uruchomić ponownie urządzenie:

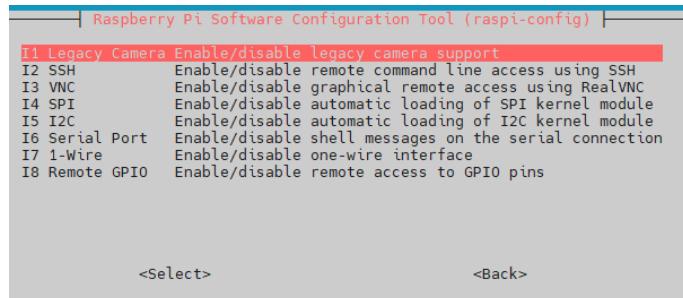
```
sudo reboot
```

2.3.2 Ustawienia systemu

Kolejnym krokiem będzie włączenie interfejsu kamery oraz opcjonalnie VNC. Należy wpisać komendę:

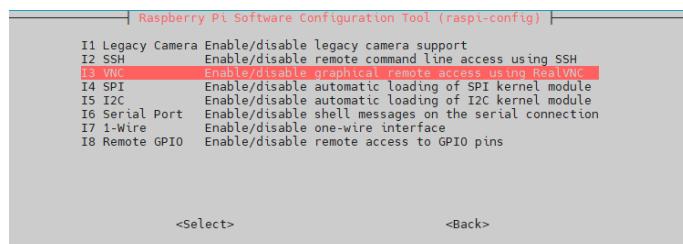
```
sudo raspi-config
```

Interfejs kamery Ponieważ interfejs może być domyślnie wyłączony należy upewnić się czy płytką umożliwia komunikację z kamerą.



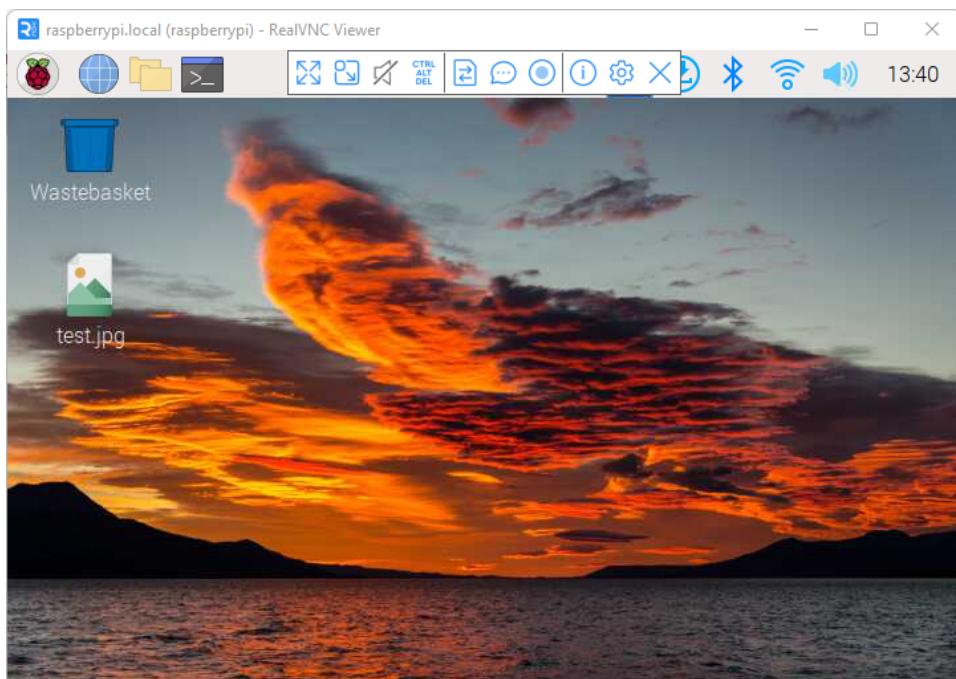
Rys. 2.4: Włączanie VNC za pomocą Interface Options -> Legacy Camera

VNC Włączenie VNC może okazać się szczególnie przydatne, gdy nie ma pewności co zawiodło w naszym programie. Umożliwia też prostsze poruszanie się po systemie dla osób nieprzepadających za używaniem systemów linuxowych w trybie headless.



Rys. 2.5: Włączanie VNC za pomocą Interface Options -> VNC

Przykładowy widok przy połączeniu się przez VNC:

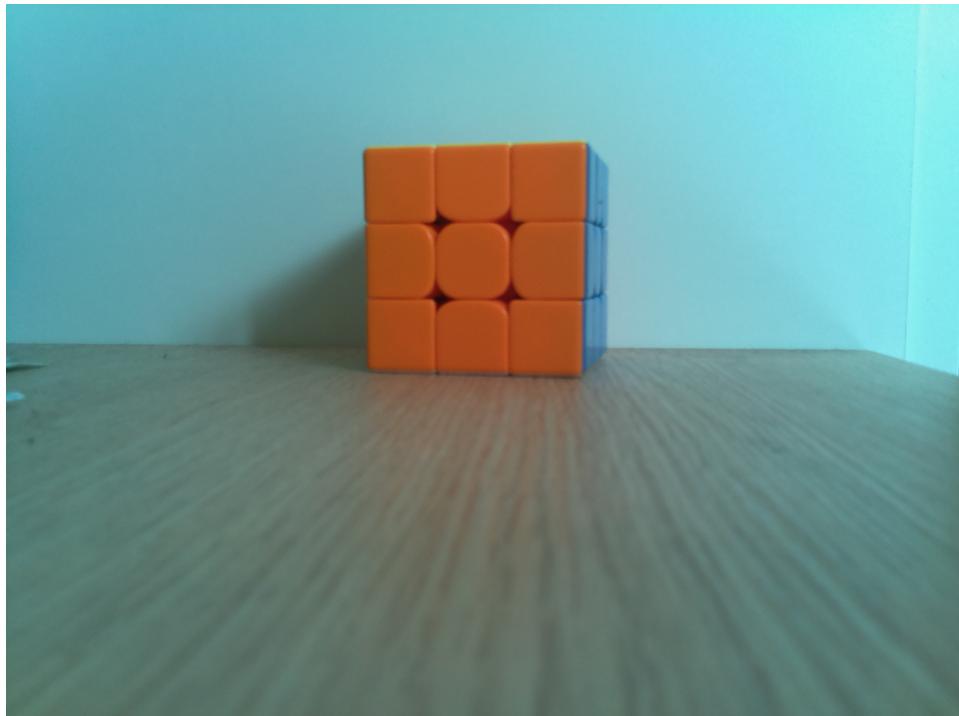


2.4 Weryfikacja konfiguracji

Podstawowym zadaniem płytki jest pozyskiwanie klatek z kamery, zatem przy poprawnej konfiguracji systemu powinna istnieć możliwość wykonania zdjęcia, zapisania go i zweryfikowania czy sam moduł kamery działa poprawnie.

```
raspistill -o test.png
```

Po zatwierdzaniu komendy LED modułu kamery świeci się na czerwono. Po zakończeniu komendy w głównym katalogu pojawia się zdjęcie test.png



Wszystko działa poprawnie. Na samym końcu zainstalowano biblioteki zgodnie ze schematem podanym we wstępnie.

3 Software

Celem przetwarzania obrazu jest zidentyfikowanie rozkładu bloków kostki - tj. pozyskanie informacji o tym jaki mają kolor oraz na której ściance się znajdują. W przypadku kostek o nieparzystej ilości bloków, ściany są identyfikowane jednoznacznie przez środkowy element. W ramach projektu analiza jest przeprowadzana wyłącznie na kostce 3x3x3 o regularnym rozmiarze oraz standardowych kolorach ścian (tj. zółty, biały, pomarańczowy, czerwony, niebieski, zielony), jednak sam algorytm powinien zostać opracowany tak, by umożliwiał analizę obrazów również dla innych typów kostek.

3.1 Idea algorytmu

Generalną pracę algorytmu można rozbić na kilka etapów:

1. Wstępne przetwarzanie obrazu - operacje zawierające zmianę: rozmiaru zdjęcia, przestrzeni kolorów; operacje filtrujące w tym przeprowadzające binaryzację
2. Wykrycie krawędzi
3. Wydobycie obszarów zawierających bloki
4. Przyporządkowanie indeksów i zaklasyfikowanie koloru każdego z bloków

Zdecydowano się postawić na metodę opartą o wykrywanie krawędzi zamiast często stosowanego podejścia wykorzystującego jedynie progowanie w oparciu o barwę. Wynika to z faktu, że umożliwia ona większą swobodę w dalszym przetwarzaniu obrazu - można niewielkim kosztem próbować wykryć inne charakterystyczne punkty jak np.: wierzchołki kostki. Dodatkowo nie ma wymagania by stosować dość skomplikowane metody color gradingu dzięki czemu metoda nadal ma szansę działać również przy niestandardowych tonach naklejek, co byłoby trudne w realizacji dla algorytmu opartego wyłącznie o sztywną kolorystykę.

3.2 Implementacja

W ramach projektu stworzono program w *Pythonie* (ver. 3.9.2), który pozwala na analizę pojedynczej klatki z kamery.

3.2.1 Wstępne przetwarzanie obrazu

W pierwszej kolejności następuje inicjalizacja kamery, wykonanie zdjęcia i "zwolnienie kamery". Szczególnie ważne jest by proces korzystający z kamery został zamknięty, inaczej niemożliwe jest ponowne uruchomienie programu dopóki nie zostanie on zakończony ręcznie.

```
# Initializing camera connection
camera = PiCamera()
rawCapture = PiRGBArray(camera)

#Making sure object is initialised and ready to go
sleep(0.1)

camera.capture(rawCapture, format="bgr")
img = rawCapture.array

print("Took picture")
#IMPORTANT \ / - ending the camera process
camera.close()
```

Następnie następuje bardzo minimalna sekcja processingu, która wiąże się z transformacją przestrzeni barw RGB do skali szarości oraz zastosowaniem filtru Gaussowskiego.

```
#Blur to lose some of the noise
blur = cv2.GaussianBlur(img, (3,3),0)

#Creating grayscale images from base
```

```
gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)
gray_rgb = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
```

Filtracja jest przydatna szczególnie przy dobrych warunkach oświetleniowych - pozwala ona na zmniejszenie ilości szczegółów, dzięki czemu dalsze przetwarzanie obrazu ma miejsce na mniejszej ilości obiektów. Wpływa ona zatem (w ogólnym przypadku) na jakość detekcji oraz czas działania algorytmu.

3.2.2 Wykrycie krawędzi

W celu wykrycia krawędzi zastosowano algorytm Canny'ego, który jako informację zwrotną daje binarny obraz. Taki charakter tego sposobu powoduje, że "słabsze" krawędzie nie zostaną wykryte, dlatego stosuje się operacje morfologiczne *dilate* oraz *erode*, które wypełniają "brakujące" piksele.

```
#Using Canny to find edges
edges = cv2.Canny(blur, 50, 50)

kernel = np.ones((3,3),np.uint8)
edges = cv2.dilate(edges, kernel, iterations = 3 )
edges = cv2.erode(edges, kernel, iterations = 2 )
```

3.2.3 Wydobycie obszarów zawierających bloki

Po wykryciu wszystkich konturów - czyli zasadniczo zamkniętych 'krawędzi' - obiekty są filtrowane kształtem oraz polem powierzchni.

```
#Finding contours
contours_1, hierarchy_1 = cv2.findContours(edges, cv2.RETR_TREE,
                                             cv2.CHAIN_APPROX_SIMPLE)

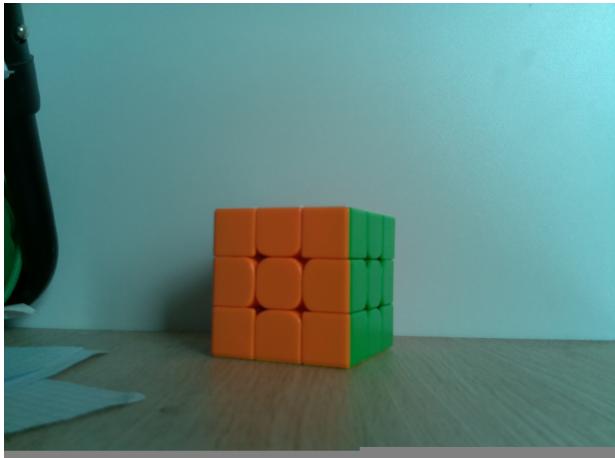
print("Found contours")
#Getting base img size
img_w, img_h, _ = img.shape
area_fac = 0.001*img_h*img_w
area_max = 0.01*img_h*img_w

output = gray_rgb.copy()
#Filtering contours based on area and shape
for c in contours_1:
    rec = cv2.minAreaRect(c)
    (x, y), (width, height), angle = rec
    area = cv2.contourArea(c)
    box = cv2.boxPoints(rec)
    box = np.int0(box)

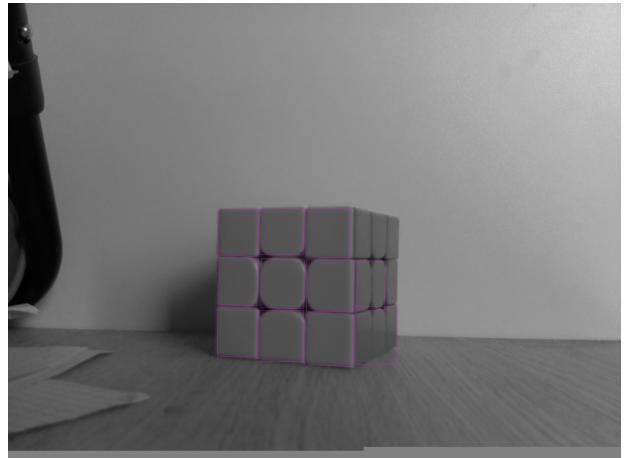
    if area>area_fac and height>0 and area<area_max:
        ratio = width/height
        #cv2.circle(output, (int(x), int(y)), 4, (0,0,255), -1)
        if ratio > 0.5 and ratio < 2:
            cv2.drawContours(output, [box], -1, (255, 0, 255), 1)
```

3.3 Wnioski

Przykładowy efekt przetwarzania



(a) Obraz oryginalny



(b) Wynik końcowy

Istnieją dwa podstawowe problemy związane z pracą algorytmu.

Problem 1 Skuteczność algorytmu jest dość mocno zależna od warunków oświetleniowych oraz jakości zdjęcia. Metoda jest bezużyteczna w przypadku gdy kontrasty są słabe bądź zdjęcie niewyraźne.

Problem 2 W przypadku niektórych typów kostek - takich jak ta testowa, czyli posiadająca kolorowy plastik zamiast naklejek - detekcja jest wyjątkowo trudna. Przetwarzanie musi odbywać się kosztem dużego zaszumienia, nie można użyć filtrów ponieważ krawędzie między blokami są bardzo słabe.

Mitygacja problemów Najprostszą metodą zniwelowania problemu oświetleniowego jest dodanie do układu kilku jasnych diód LED. Zarządzając ich jasnością układ mógłby doprowadzić do warunków, w których algorytm dobrze sobie radzi. Innym trudniejszym rozwiązaniem byłoby napisanie fragmentu kodu odpowiadające za dostrajanie się algorytmu do oświetlenia, mogłyby to odbywać się na przykład na podstawie analizy kolorów czy histogramu obrazu.

3.4 Przyszły rozwój projektu

W przyszłych etapach projektu należałoby przede wszystkim "uodpornić" algorytm na zmiany oświetlenia. Można byłoby również kamery o wyższej rozdzielczości oraz zawierającej autofocus - wpłynęłoby to korzystnie na efekty przetwarzania niezależnie od problemu z warunkami robienia zdjęcia.

Funkcjonalnie należałoby rozszerzyć program o przetwarzanie w trybie ciągłym, lub takim który wykonuje kilka (dziesiąt/naście) zdjęć i wybiera te najbardziej korzystne.

Ulepszeniom mogłyby również ulec informacja zwrotna. Funkcjonalność możnaby poszerzyć o wykrywanie wierzchołków, klasyfikację kolorów segmentów, oraz zwracanie listy wykrytych krawędzi i ich umiejscowienia.

4 Dodatek

Pełny kod programu:

```
import cv2
import numpy as np
#import matplotlib.pyplot as plt
from picamera.array import PiRGBArray
from picamera import PiCamera
from time import sleep

print("Loaded libs")
#Initializing camera connection
camera = PiCamera()
rawCapture = PiRGBArray(camera)

sleep(0.1)

camera.capture(rawCapture, format="bgr")
img = rawCapture.array

print("Took picture")
#IMPORTANT \
camera.close()

#cv2.imshow('Captured', img)

#Blur to lose some of the noise
blur = cv2.GaussianBlur(img, (1,1),0)

#Creating grayscale images from base
gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)
gray_rgb = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)

#Using Canny to find edges
edges = cv2.Canny(blur, 50, 50)

kernel = np.ones((3,3),np.uint8)
edges = cv2.dilate(edges, kernel, iterations = 3 )
edges = cv2.erode(edges, kernel, iterations = 2 )
cv2.imshow('Edges', edges)

print("Finished pre-processing")
#Finding contours
contours_1, hierarchy_1 = cv2.findContours(edges, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

print("Found contours")
#Getting base img size
img_w, img_h, _ = img.shape
area_fac = 0.001*img_h*img_w
area_max = 0.01*img_h*img_w

output = gray_rgb.copy()
#Filtering contours based on area and shape
for c in contours_1:
    rec = cv2.minAreaRect(c)
    (x, y), (width, height), angle = rec
    area = cv2.contourArea(c)
```

```
box = cv2.boxPoints(rec)
box = np.int0(box)

if area>area_fac and height>0 and area<area_max:
    ratio = width/height
    #cv2.circle(output, (int(x), int(y)), 4, (0,0,255), -1)
    if ratio > 0.5 and ratio < 2:
        cv2.drawContours(output, [box], -1, (255, 0, 255), 1)
print("Filtered contours")
#Final result
cv2.imshow("Output", output)
cv2.imwrite("output_file.png", output)
print("Done")

cv2.waitKey()
```