

Sprawozdanie KCK

November 18, 2019

Temat: Wykrywanie liczby oczek na kostce

Autorzy: Iwo Naglik 136774, Bartosz Przybył 136785

Wstęp: Tematem naszego projektu jest wykrywanie liczby oczek na kostkach do gry. W rozwiązaniu tego problemu wykorzystujemy bibliotekę OpenCV. Językiem programowania naszej aplikacji jest Python. Po zaczytaniu obrazu w skali szarości poddajemy go wstępnej obróbce w celu wykrycia potencjalnych kostek. Następnie znajdujemy kontury możliwych elementów, wycinamy je w celu ułatwienia analizy danego fragmentu zdjęcia. Wycięty kawałek zdjęcia poddajemy operacji erozji i przechodzimy do kolejnego wykrycia konturów (oczka). Jeśli dany kawałek nie zawierał żadnych konturów (oczek) to pomijamy go, w przeciwnym wypadku zliczamy liczbę konturów, wypisujemy tę liczbę na oryginalne zdjęcie i zaznaczamy znalezioną kostkę na oryginalnym zdjęciu. Następnie przechodzimy do kolejnego znalezioneego fragmentu zdjęcia w celu analizy.

Lista kroków:

1. Zaczynamy od wczytania obrazków w kolorze, następnie skalujemy je do wymiarów 960x720, ponieważ w takim wymiarze obrazu zdecydowanie prościej było go przetwarzać aniżeli w oryginalnych wymiarach 4032x3024. Z drugiej strony mniejsze wymiary obrazka traciły na jakości - zdecydowanie trudniej było wykryć znaczące szczegóły. Następnie obraz konwertujemy do skali szarości, gdyż to na niej nastąpi dalsze przetwarzanie. Poniżej przedstawiamy 3 przykładowe zdjęcia zróżnicowane według trudności - zmienia się w nich tło, oświetlenie oraz liczba kostek i liczba przedmiotów przeszkadzających analizie oraz kąt zrobienia zdjęcia.

```
In [1]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import cv2
from skimage import io

image1_color = cv2.imread("1/15.jpg")
image2_color = cv2.imread("2/10.jpg")
image3_color = cv2.imread("3/27.jpg")

image1_gray = cv2.cvtColor(image1_color, cv2.COLOR_BGR2GRAY)
image2_gray = cv2.cvtColor(image2_color, cv2.COLOR_BGR2GRAY)
image3_gray = cv2.cvtColor(image3_color, cv2.COLOR_BGR2GRAY)
```

```

image1_color = cv2.resize(image1_color, (960, 720))
image2_color = cv2.resize(image2_color, (960, 720))
image3_color = cv2.resize(image3_color, (960, 720))

image1_gray = cv2.resize(image1_gray, (960, 720))
image2_gray = cv2.resize(image2_gray, (960, 720))
image3_gray = cv2.resize(image3_gray, (960, 720))

plt.figure(figsize=(30,30))
plt.subplot(1,3,1); plt.axis('off');
plt.imshow(cv2.cvtColor(image1_color, cv2.COLOR_BGR2RGB))
plt.subplot(1,3,2); plt.axis('off');
plt.imshow(cv2.cvtColor(image2_color, cv2.COLOR_BGR2RGB))
plt.subplot(1,3,3); plt.axis('off');
plt.imshow(cv2.cvtColor(image3_color, cv2.COLOR_BGR2RGB))

```

Out [1]: <matplotlib.image.AxesImage at 0x113a56d0>



- Następnym krokiem jest porównanie wartości danego piksela do progu obliczonego jako średnia z wartości pikseli obrazka zsumowana z odchyleniem standardowym z wartości pikseli obrazka. Wartość progowa została ustalona na podstawie tła, które zajmuje większą powierzchnię obrazka, dlatego uznaliśmy, że średnia będzie dobrym wyznacznikiem, a z racji faktu, że średnia jest wrażliwa na wartości odstające uwzględniliśmy dodatkowo odchylenie standardowe, żeby zmaksymalizować separację tła od kostek.

```

In [2]: def plot(image1, image2, image3):
    plt.figure(figsize=(30,30))
    plt.subplot(1,3,1); plt.axis('off'); plt.imshow(image1, cmap="binary_r")
    plt.subplot(1,3,2); plt.axis('off'); plt.imshow(image2, cmap="binary_r")
    plt.subplot(1,3,3); plt.axis('off'); plt.imshow(image3, cmap="binary_r")

```

```

In [3]: x1 = round(np.std(image1_gray) + np.mean(image1_gray))
        x2 = round(np.std(image2_gray) + np.mean(image2_gray))

```

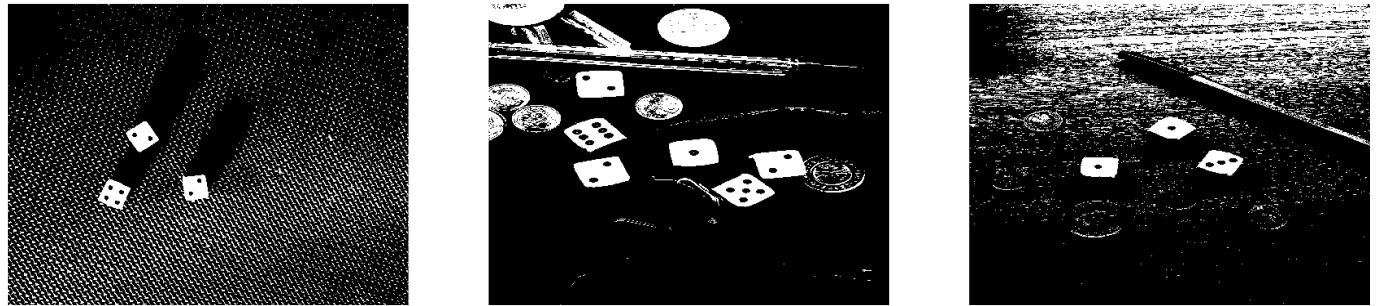
```

x3 = round(np.std(image3_gray) + np.mean(image3_gray))

image1_gray = (image1_gray > x1) * 255
image2_gray = (image2_gray > x2) * 255
image3_gray = (image3_gray > x3) * 255

plot(image1_gray, image2_gray, image3_gray)

```



3. Kolejno obrazki zostają poddane operacji erozji w celu odrzucenia małych, nieistotnych punktów. Poprzez tę operację całkowicie niwelujemy wpływ tła, czego efekt widać na poniższych obrazkach. Niektóre przedmioty przeszkadzające nie zostały całkowicie zaczernione, ale nimi baczniej będziemy się przyglądać w kolejnym etapie programu.

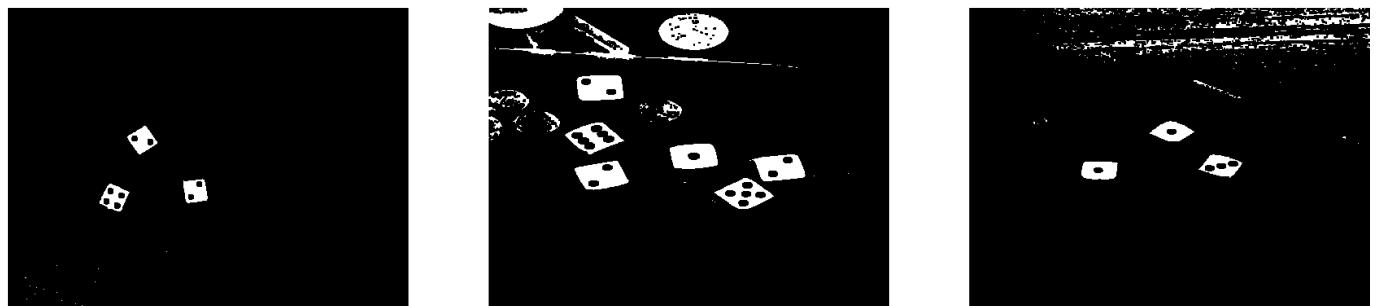
```

In [4]: image1_gray = np.uint8(image1_gray)
         image2_gray = np.uint8(image2_gray)
         image3_gray = np.uint8(image3_gray)

         kernel = np.ones((5,5), np.uint8)
         image1_gray = cv2.erode(image1_gray, kernel, iterations=1)
         image2_gray = cv2.erode(image2_gray, kernel, iterations=1)
         image3_gray = cv2.erode(image3_gray, kernel, iterations=1)

plot(image1_gray, image2_gray, image3_gray)

```



4. Następnym krokiem jest wykrycie konturów na przetworzonych obrazkach. Po wykryciu kształtów analizujemy je pod kątem długości oraz liczby posiadanych wewnętrznych konturów.

```
In [5]: def findRectangle(contours, hierarchy):
    x = []
    y = []
    cubes = []
    for i in range(len(contours)):
        if len(contours[i]) > 30 and len(contours[i]) < 242 \
           and hierarchy[0][i][2] != -1:
            for j in range(len(contours[i])):
                x.append(contours[i][j][0][0])
                y.append(contours[i][j][0][1])
            cubes.append([max(x), min(x), max(y), min(y)])
            x = []
            y = []
    return cubes

contours1, hierarchy1 = cv2.findContours
                    (image1_gray, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
contours2, hierarchy2 = cv2.findContours
                    (image2_gray, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
contours3, hierarchy3 = cv2.findContours
                    (image3_gray, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
cubes1 = findRectangle(contours1, hierarchy1)
cubes2 = findRectangle(contours2, hierarchy2)
cubes3 = findRectangle(contours3, hierarchy3)
```

5. Następnie do funkcji `find_blobs` przekazujemy zdjęcie po wszystkich transformacjach, zdjęcie oryginalne i tablicę punktów potrzebnych do wycięcia odpowiedniego fragmentu zdjęcia. W pętli iterującej po owej tablicy wycinamy odpowiednie fragmenty zdjęć i skalujemy je za pomocą funkcji `fragment`. Otrzymany wycięty kawałek przerobionego zdjęcia poddajemy transformacji erozji w celu usunięcia możliwości wykrycia drobnych szczegółów, które nie są szukanymi oczkami. Kolejno znajdujemy kontury na wyciętym fragmencie (oczka). Podczas wykrywania konturów zostaje wykryty także kontur kostki, który zapamiętujemy w tablicy `rozmiary_kostek`. Następnie iterujemy się po znalezionych konturach i odrzucamy kontur kostki zapisany w wcześniej wspomnianej tablicy oraz pomijamy zbyt małe wykryte elementy, gdyż w większości przypadków były to drobne szczegółы typu zabrudzenia (wartość dobrana eksperymentalnie). Dodatkowo w pętli tej zliczamy liczbę oczek na danej kostce i sumujemy tę liczbę z wcześniej znalezionymi oczkami w celu znalezienia łącznej sumy oczek na zdjęciu. Po tych operacjach rysujemy znalezione kontury na wyciętym fragmencie, czego wynik można zaobserwować poniżej. Do tablicy `kontury_w_srodku` zapisujemy liczbę konturów znalezionych wewnątrz analizowanego fragmentu. Do tablicy `tab_sum_kostek` zapisujemy indeks danego fragmentu oraz sumę oczek na znalezionym fragmencie zdjęcia. Kończąc ten etap programu rysujemy na oryginalnym zdjęciu znalezione kontury kostek.

```

In [6]: #Funkcja wycinajaca odpowiedni fragment zdjecia i skalujaca do rozmiaru 400x400
def fragment(image, cubes):
    return cv2.resize(image[cubes[3]:cubes[2], cubes[1]:cubes[0]], (400, 400))

#Funkcja sluzaca do narysowania wycietych fragmentow zdjecia
#oryginalnego i zdjecia poddanego przerobce
def plot_fragment(image, image_color, size, index):
    if index==1:
        plt.figure(figsize=(15,15))
        plt.subplots_adjust(wspace=0, hspace=0)
        plt.subplot(2,size, index); plt.axis('off');
        plt.imshow(image, cmap="binary_r")

        plt.subplot(2,size, index+size); plt.axis('off');
        plt.imshow(cv2.cvtColor(image_color, cv2.COLOR_BGR2RGB))

def find_blobs(image_gray, image_color, cubes):
    suma_oczek = 0
    kontury_w_srodku = []
    tab_sum_kostek = []
    rozmiary_kostek = []
    tylko_kostki = []
    kernel = np.ones((3,3), np.uint8)
    for j in range(len(cubes)):
        image_gray_fragment = fragment(image_gray, cubes[j])
        image_color_fragment = fragment(image_color, cubes[j])
        image_gray_fragment = cv2.erode(image_gray_fragment, kernel, iterations=2)

        contours_cube, hierarchy_cube =
        cv2.findContours(image_gray_fragment, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

        suma_kostka = 0
        temp = []

        for i in range(len(contours_cube)):
            temp.append(len(contours_cube[i]))

        max_len = max(temp)
        rozmiary_kostek.append(max_len)
        pom = []

        for i in range(len(contours_cube)):
            if len(contours_cube[i]) < max_len and len(contours_cube[i]) > 30:
                suma_kostka = suma_kostka + 1
                suma_oczek = suma_oczek + 1
                cv2.drawContours(image_color_fragment, contours_cube, i, (0,255,0), 3)
                pom.append(len(contours_cube[i]))

```

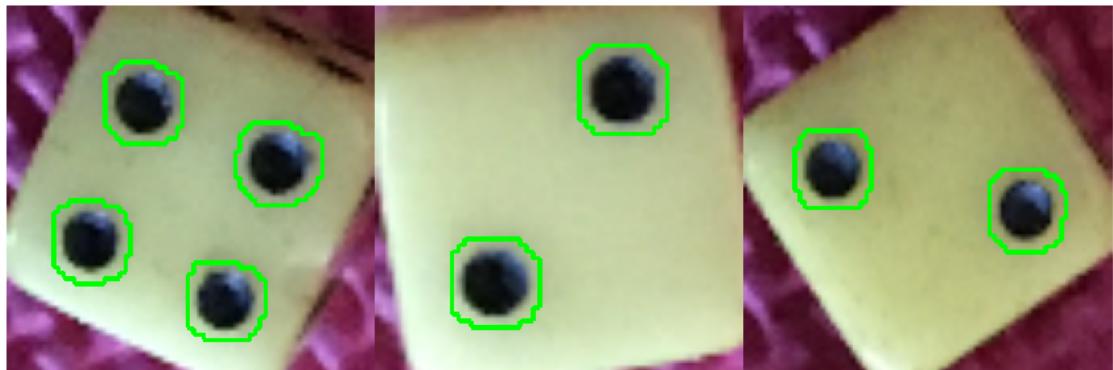
```

        tylko_kostki.append(j)
kontury_w_srodku.append(pom)
tab_sum_kostek.append([j, suma_kostka])
plot_fragment(image_gray_fragment, image_color_fragment, len(cubes), j+1)
if j in tylko_kostki:
    cv2.rectangle(image_color,(cubes[j][1], cubes[j][3]),
                  (cubes[j][0], cubes[j][2]),(0,255,0),3)

return tab_sum_kostek, kontury_w_srodku, suma_oczek

tab_sum_kostek1, kontury_w_srodku1, suma_oczek1 =
    find_blobs(image1_gray, image1_color, cubes1)
tab_sum_kostek2, kontury_w_srodku2, suma_oczek2 =
    find_blobs(image2_gray, image2_color, cubes2)
tab_sum_kostek3, kontury_w_srodku3, suma_oczek3 =
    find_blobs(image3_gray, image3_color, cubes3)

```





6. Z tablicy kontury_w_srodku usuwamy listy puste (fragmenty zdjęcia, które okazały się nie być kostkami). Następnie wyliczamy minimalną i maksymalną wartość z tej tablicy w celu rozwiązania problemu z liczbą oczek równą 6 (Funkcja findContours zwraca liczbę 3 oczek z kostki 6-oczkowej jako jeden kontur, a nie 3 oddzielne kontury).

```
In [7]: kontury_w_srodku1 = [x for x in kontury_w_srodku1 if x != []]
kontury_w_srodku2 = [x for x in kontury_w_srodku2 if x != []]
kontury_w_srodku3 = [x for x in kontury_w_srodku3 if x != []]

min_contour1 = min(kontury_w_srodku1)
max_contour1 = max(kontury_w_srodku1)
min_contour2 = min(kontury_w_srodku2)
max_contour2 = max(kontury_w_srodku2)
min_contour3 = min(kontury_w_srodku3)
max_contour3 = max(kontury_w_srodku3)
```

7. Na zakończenie chcemy wypisać wartości znalezione na zdjęciu. Jednak najpierw musimy poradzić sobie z problemem wykrywania szóstki gdyż za każdym razem znajduje zamiast sześciu oddzielnych konturów oczek, dwa długie kontury linii utworzonej z trzech oczek obok siebie. Radzimy sobie z tym problemem w taki sposób, że porównujemy długość konturu tejże linii z długością konturu kostki i jeśli 2.5-krotność konturu linii przekroczy obwód kości, wiemy, że zamiast sumy oczek równej dwa, powinno zwrócić 6. Następnie zliczoną liczbę oczek na danej kostce umieszczamy obok zaznaczonego obszaru, na którym znajduje się kostka. Na koniec wypisujemy także sumę wszystkich oczek na zdjęciu w lewym górnym rogu zdjęcia.

```
In [8]: def plot_original(image1, image2, image3):
    plt.figure(figsize=(30,30))
    plt.subplot(1,3,1); plt.axis('off');
        plt.imshow(cv2.cvtColor(image1, cv2.COLOR_BGR2RGB))
    plt.subplot(1,3,2); plt.axis('off');
        plt.imshow(cv2.cvtColor(image2, cv2.COLOR_BGR2RGB))
    plt.subplot(1,3,3); plt.axis('off');
        plt.imshow(cv2.cvtColor(image3, cv2.COLOR_BGR2RGB))

def text(image,tab_suma,kontury_w_srodku,cubes,min_contour,max_countour,suma_oczek):

    for i in range(len(tab_suma)):
        if (max(max_countour) > (2.5*min_contour[0])) and len(kontury_w_srodku) != 1) \
        and i == kontury_w_srodku.index(max_countour) and tab_suma[i][1] == 2:

            cv2.putText(image, str(tab_suma[i][1] * 3), (cubes[tab_suma[i][0]][0],
                cubes[tab_suma[i][0]][2]), cv2.FONT_HERSHEY_PLAIN, 3,
                (80, 0, 200, 255), thickness = 3)

            suma_oczek = suma_oczek + 4
        else:
            if tab_suma[i][1] != 0:
```

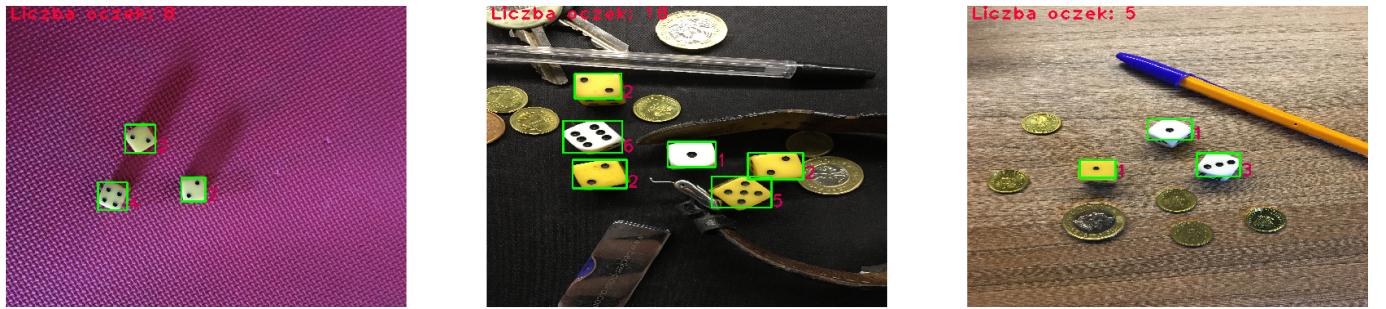
```

cv2.putText(image, str(tab_suma[i][1]), (cubes[tab_suma[i][0]][0],
                                         cubes[tab_suma[i][0]][2]),
            cv2.FONT_HERSHEY_PLAIN, 3,
            (80, 0, 200, 255), thickness = 3)

cv2.putText(image, "Liczba oczek: {}".format(suma_oczek), (10, 35),
            cv2.FONT_HERSHEY_PLAIN, 3, (50, 0, 255, 255), thickness = 4)

text(image1_color, tab_sum_kostek1, kontury_w_srodku1,
      cubes1, min_contour1, max_contour1, suma_oczek1)
text(image2_color, tab_sum_kostek2, kontury_w_srodku2,
      cubes2, min_contour2, max_contour2, suma_oczek2)
text(image3_color, tab_sum_kostek3, kontury_w_srodku3,
      cubes3, min_contour3, max_contour3, suma_oczek3)
plot_original(image1_color, image2_color, image3_color)

```



Podsumowanie: Reasumując dla przywołanej trójki zdjęć o różnym stopniu trudności program poradził sobie bardzo dobrze, zidentyfikował wszystkie kostki, zaznaczył je na oryginalnym obrazie, wypisał sumę oczek obok każdej kostki oraz całkowitą sumę oczek na całym zdjęciu. Jednak program nie radzi sobie z każdym typem zdjęć:

Na poniższym zdjęciu program wykrył jako kontur również ściankę boczną zawierającą dodatkowe oczka, które są dość mocno widoczne, przez co następuje zliczenie oczek także na ścianach bocznych kostki. Problemem w tym przypadku jest zbyt duży kąt, pod którym zostało zrobione zdjęcie tejże kostki.



Na kolejnym zdjęciu problemem jest zbyt małe oświetlenie, co przekłada się na zbyt duży negatywny wpływ erozji, która łączy kontury oczka z krawędzią kostki, czego konsekwencją jest brak możliwości wykrycia konturu oczka przez funkcję findContours. Kostki z dwoma oczkami zostały sklasyfikowane jako kostki z jednym oczkiem.

