POLITECHNIKA POZNAŃSKA

WYDZIAŁ ELEKTRYCZNY

Instytut Automatyki, Robotyki i Inżynierii Informatycznej

Dokumentacja projektu

Rozpoznawanie obrazu z gry w warcaby oraz wizualizacja stanu gry na komputerze

Członkowie:

Natalia Gierszewska

Stanisław Gilewicz

Marcin Hradowicz

Adam Klejda

14 czerwca 2019

Spis treści

1. Wybór tematu

Wybór tematu został zdeterminowany przede wszystkim z dwóch powodów. Jednym z nich było to, iż dwie z czterech osób naszego zespołu wiele godzin spędza na rekreacyjnej grze w szachy. Drugim była chęć napisania jakiejś większej aplikacji w języku Python. Spotkaliśmy się z nim na niektórych przedmiotach na pojedynczych zajęciach, ale nigdy nie mogliśmy się z nim bardziej zapoznać. Wiedzieliśmy natomiast z jednych z zajęć, że jest to język często wykorzystywany w rozpoznawaniu obrazów w różnego rodzaju aplikacjach.

2. Wykorzystane technologie

Tak jak było wspomniane wyżej zdecydowaliśmy się na wybór Pythona jako języka do napisania aplikacji. Użytych zostało także wiele bibliotek, które były wymagane np. do rozpoznawania planszy lub tylko miały ułatwić pracę lub poprawić jakość projektu np. użycie biblioteki PyGame do tworzenia graficznego interfejsu użytkownika.

Za środowisko pracy posłużył nam program PyCharm, który został stworzony przez firmę JetBrains jako zintegrowane środowisko programistyczny dla języka programowania Python.

Poniżej przedstawiamy listę najważniejszych bibliotek użytych do projektu wraz z wersjami, na których przeprowadzano ostateczne testy.

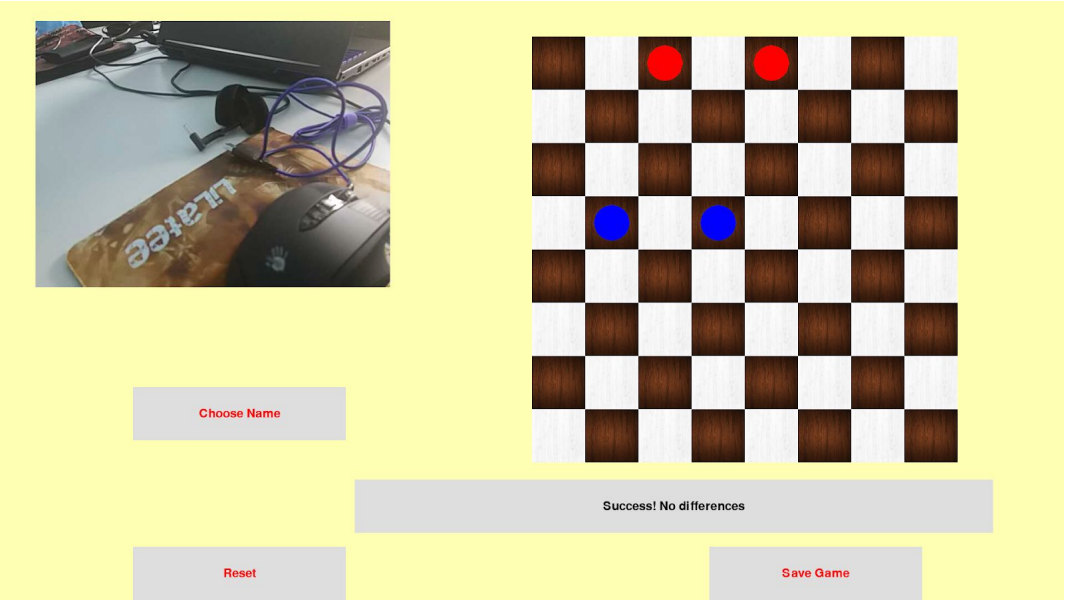
- opencv-python (4.1.0.25)
- pygame (1.9.6)
- imutils (0.5.2)
- numpy (1.16.3)
- requests (2.21.0)

Na początku pracy postanowiliśmy użyć dedykowanej kamery PS3Eye z Playstation 3 wraz z płatnym sterownikiem (2.99 $) CL-Eye Platform Driver (dostępny pod tym linkiem https://codelaboratories.com/downloads/).

Rysunek 1. Sterownik CL-Eye

Później jednak zrezygnowaliśmy z niej, ponieważ byliśmy zmuszeni do zmiany wersji biblioteki OpenCV, która miała problem z powyższą kamerą. Ostatecznie zdecydowaliśmy się na wybór telefonu z systemem Android 4.1+ jako bezprzewodowej kamery działającej w obrębie jednej sieci Wi-Fi. Pozwoliła nam na to bezpłatna aplikacja IP Webcam (zawierająca nieinwazyjne reklamy) dostępna w Sklepie Play (dostępna pod tym linkiem https://play.google.com/store/apps/details?id=com.pas.webcam&hl=pl). Udostępnia ona obraz z kamery telefonu (domyślnie na porcie 8080) pod wskazany w aplikacji adres z konkretnym punktem końcowym.

Rysunek 2. Aplikacja mobilna (IP Webcam) pobierająca obraz



Rysunek 3. Udostępniony przez IP Webcam obraz przekazany do aplikacji Checkers

Ostatecznie wróciliśmy do pierwotnej wersji biblioteki OpenCV i możemy używać kamery PS3Eye w naszym projekcie, jednak ze względu na wygodę do końca zostaliśmy przy używaniu aplikacji IP Webcam.

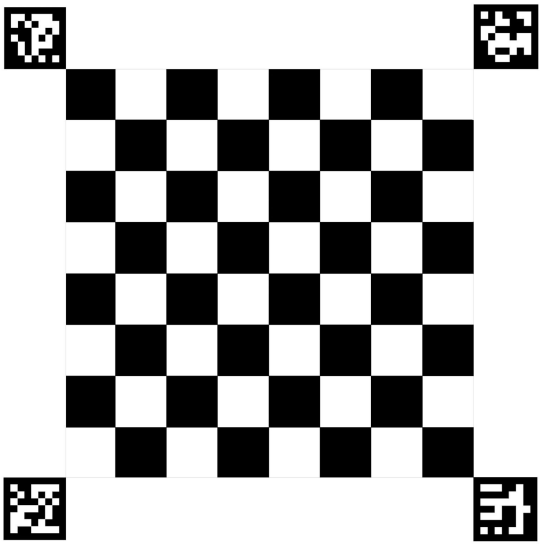
3. Główne założenia projektu

- Nad planszą do gry w warcaby jest umieszczona kamera.
- Podsystem do rozpoznawania obrazów wyszukuje pozycje pionków i przekazuje je do podsystemu wizualizacji.
- Podsystem wizualizacji przedstawia planszę i pionki jakie są umieszczone na fizycznej planszy.
- Sprawdzanie czy ruch został wykonany zgodnie z zasadami.

4. Dodatkowe założenia projektu

- Zapisywanie historii gier z możliwością prześledzenia gry krok po kroku.
- Zwykłe pionki reprezentowane są przez koła, a damki przez trójkąty.
- Niebieski kolor pionków odpowiada białemu w rzeczywistej grze, natomiast czerwony odpowiada kolorowi czarnemu.
- Grę zaczyna gracz posiadający pionki koloru niebieskiego.

- Po wykonaniu niedozwolonego ruchu stan planszy na wizualizacji nie ulega zmianie. Jest natomiast wyświetlany komunikat o wykonaniu nieprawidłowego ruchu i prośba o powrocie do stanu sprzed wykonania ruchu. Gra wraz z wizualizacją zostanie wznowiona dopiero po powrocie do poprawnego stanu planszy sprzed ruchu.
- Zasady według których rozliczamy poprawność ruchów opisane są w punkcie 9.
- Gra musi się odbywać na specjalnie przygotowanej planszy przedstawionej poniżej. Rozmiar planszy nie ma znaczenia. Można ją pobrać z tego linku: https://drive.google.com/open?id=1JH5aXzcoS-9nCnun-NAx1-A3lyhw1Z2I



Rysunek 4. Plansza używana w grze

5. Podział prac pomiędzy członków zespołu

Natalia Gierszewska:

- Prace związane z rozpoznawaniem planszy, rozpoznanie biblioteki OpenCV oraz możliwości markerów Aruco.
- Komunikaty o błędach w ruchu pionków lub poprawnym ruchu, komunikat o rozpoczęciu gry.
- Ustalanie zwycięzcy i przekazanie informacji użytkownikowi.

Stanisław Gilewicz:

- Wykonanie logiki gry w warcaby.
- Sprawdzanie poprawności wykonywanych ruchów.
- Przekazywanie informacji o poprawnym lub niewłaściwym wykonaniu ruchu do interfejsu graficznego.

Marcin Hradowicz:

- Rozpoznawanie planszy.
- Rozpoznawanie pionków zarówno zwykłych jak i damek.
- Przekazywanie tablicy z informacją o pozycji pionków na planszy do interfejsu graficznego

Adam Klejda:

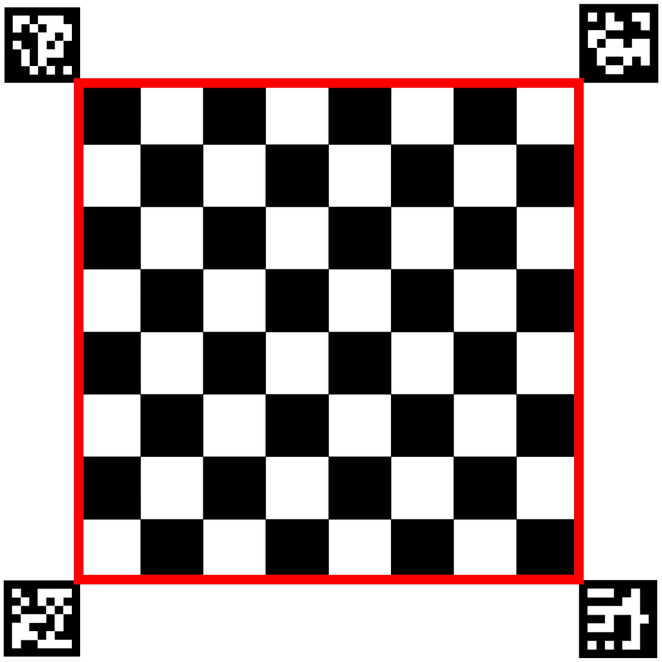
- Utworzenie interfejsu graficznego.
- Komputerowa wizualizacja planszy.
- Pokazanie aktualnego obrazu z kamery.
- Możliwość zapisywania i wczytywania gier z możliwością analizy krok po kroku.

6. Architektura rozwiązania

Moduł rozpoznawania obrazu

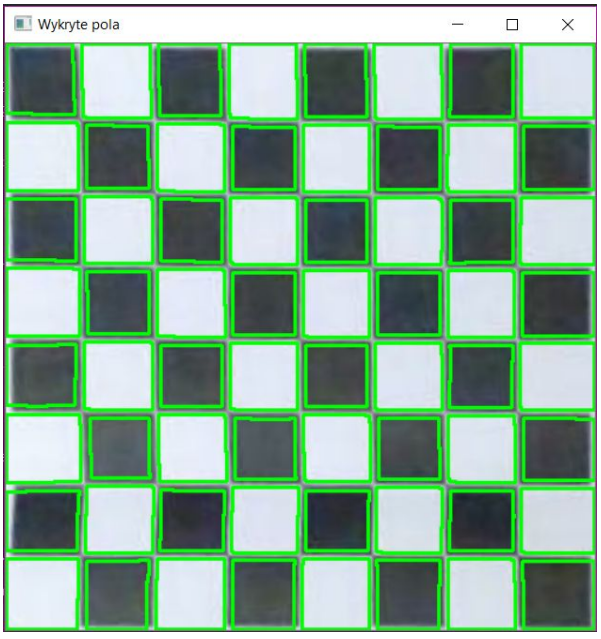
Opis

Moduł ten odpowiada za wychwytywanie na obrazie źródłowym planszy, gdzie plansze definiujemy jako kwadrat 8x8 pól.



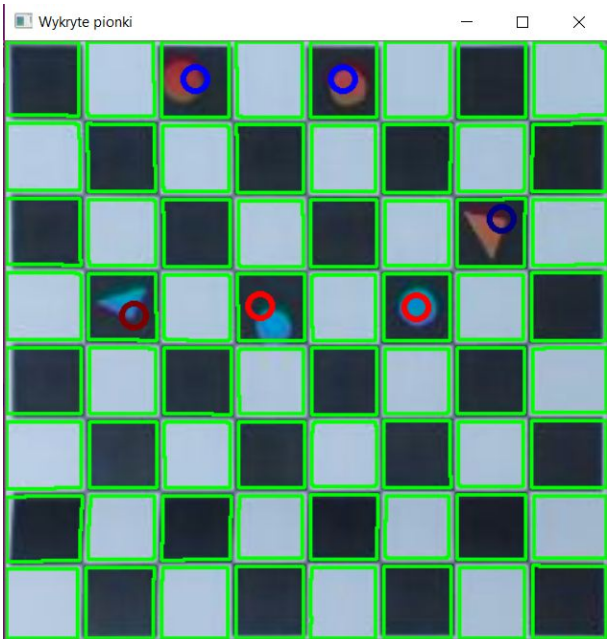
Rysunek 5. Zaznaczony kolorem czerwonym wyodrębniany obszar przez moduł rozpoznawania

Następnie zajmuje się wykryciem każdego pola (czarnych oraz białych) osobno, aby uzyskać ich położenie na obrazie.



Rysunek 6. Wykryte pola

Ostatnim krokiem jest wykrywanie zwykłych pionków (reprezentowanych jako koła) oraz damek (reprezentowanych jako trójkąty). Na rysunku 6. wykrycia pionków przedstawione zostały za pomocą okręgów. Okręgi niebieskie przedstawiają wykrycie pionków gracza czerwonego, natomiast okręgi czerwone to pionki gracza niebieskiego. Ciemniejszy odcień odpowiedniego koloru symbolizuje wykrycie damki.



Rysunek 7. Wykryte pionki

Struktura

Moduł składa się z jednego pliku detecter.py składającego się z następujących funkcji:

```
def get_chessboard_as_image(image)
def detect_shape(contour)
def get_fields_as_list_of_points_list(image)
def get_list_of_pawns_points(image, threshold)
def get_fields_info_as_list_of_lists
def detect(camera_image, last_result)
```

Funkcja detect jest jedyną funkcją jaką trzeba wywołać, aby uzyskać informację o położeniu pionków na planszy. To ona jest wywoływana po stronie modułu aplikacji.

```
def get_chessboard_as_image(image)
```

a. Przeznaczenie

Z obrazu z kamery wyszukuje 4 kody aruco umieszczone na rogach planszy. Uzyskujemy dzięki temu współrzędne każdego z 4 rogów planszy. Po wykonaniu przekształcenia perspektywicznego otrzymujemy obraz przedstawiający samą szachownicę 8x8 pól.

b. Wejście

image - obraz źródłowy z kamery.

c. Zastosowane algorytmy

Konwersja obrazu do skali szarości oraz uzyskanie obrazu binarnego za pomocą progowania adaptacyjnego.

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
thresh = cv2.adaptiveThreshold(gray, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 41, 40)
```

Następnie za pomocą biblioteki aruco wykrywane są 4 znaki aruco umieszczone na rogach szachownicy w celu uzyskania współrzędnych narożników. Jeżeli nie udało wykryć się wszystkich 4 znaków to dana klatka obrazu zostaje porzucona. Wykorzystana została baza symboli aruco o rozdzielczości 7x7 pikseli zawierająca 50 różnych znaków (każdy znak symbolizuje wartość liczbową od 0 do 49).

Użyte zostały znaki reprezentujące liczby od 0 do 3 (generator znaków aruco: http://chev.me/arucogen/). Wykorzystanie akurat tej bazy było zupełnie przypadkowe. Równie dobrze mogły zostać wykorzystane pierwotne, najprostsze symbole aruco. Prawdopodobnie mogłoby to w niewielki stopniu przyspieszyć działanie.

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
thresh = cv2.adaptiveThreshold(gray, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 41, 40)
aruco_dict = aruco.Dictionary_get(aruco.DICT_7X7_50)
```

```
parameters = aruco.DetectorParameters_create()
corners, ids, _ = aruco.detectMarkers(thresh, aruco_dict,
parameters=parameters)

if len(corners) < 4:
    return None
```

Następnie współrzędne wykrytych znaków zostały posortowane zgodnie z wartością liczbową przypisaną każdemu znaku aruco (od najmniejszej do największej).

```
sorted_corners = [x for _, x in sorted(zip(ids, corners))]
```

Współrzędne każdego narożnika szachownicy zostały wyciągnięte z listy posortowanych współrzędnych znaków aruco.

```
ULx = sorted_corners[0][0][2][0]  # x of UP LEFT corner
ULy = sorted_corners[0][0][2][1]  # y of UP LEFT corner
URx = sorted_corners[3][0][3][0]  # x of UP RIGHT corner
URy = sorted_corners[3][0][3][1]  # y of UP RIGHT corner
DLx = sorted_corners[2][0][1][0]  # x of DOWN LEFT corner
DLy = sorted_corners[2][0][1][1]  # y of DOWN LEFT corner
DRx = sorted_corners[1][0][0][0]  # x of DOWN RIGHT corner
DRy = sorted_corners[1][0][0][1]  # y of DOWN RIGHT corner
```

Na koniec wykonano przekształcenie perspektywiczne i ustawiono rozmiar obrazu samej szachownicy na 500x500 pikseli.

```
pts1 = np.float32([[ULx, ULy], [URx, URy], [DLx, DLy], [DRx,
DRy]])
pts2 = np.float32([[0, 0], [500, 0], [0, 500], [500, 500]])
M = cv2.getPerspectiveTransform(pts1, pts2)
result = cv2.warpPerspective(image, M, (500, 500))
```

d. Wyjście

result - obraz 500x500 pikseli przechowywany w obiekcie typu numpy.ndarray przedstawiający wycinek samej szachownicy.

```
def detect_shape(contour)
```

a. **Przeznaczenie**

Wykrywa kształt z przekazanego konturu.

b. **Wejście**

contour - tablica typu numpy.ndarray zawierająca dwuelementowe tablice będące charakterystycznymi punktami danego konturu.

Poniżej przedstawiono przykładowe użycie na obrazie binarnym. Zmiennej shape zostaje przypisana wartość typu string symbolizująca wykryty kształt.

```
contours = cv2.findContours(binary_image, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
contours = imutils.grab_contours(contours)
for i, c in enumerate(contours_sorted):
    shape = detect_shape(c)
```

c. **Zastosowane algorytmy**

Funkcja wylicza za pomocą bibliotecznej funkcji opencv approxPolyDP() liczbę wykrytych krawędzi konturu. Dzięki tej liczbie można stwierdzić z jakim kształtem mamy do czynienia. Dla liczby 3 można stwierdzić wystąpienie trójkąta, dla liczby 4 wystąpienie kwadratu, a w pozostałych przypadkach przyjmuje się wykrycie koła.

```
approx = cv2.approxPolyDP(contour, 0.1 * cv2.arcLength(contour,
True), True)
```

d. **Wyjście**

Na wyjściu dostajemy obiekt typu string wskazujący na wykryty kształt:

- ‘square’ - wykrycie kwadratu;
- ‘triangle’ - wykrycie trójkąta;
- ‘circle’ - wykrycie koła.

```
def get_fields_as_list_of_points_list(image)
```

a. Przeznaczenie

Odpowiada za wykrycie wszystkich pól w obrazie (zarówno białych jak i czarnych) i oszacowanie współrzędnych w ich środku.

b. Wejście

image - obraz przechowywany w obiekcie typu numpy.ndarray będący wykrytą planszą zwracaną przez funkcję get_chessboard_as_image().

c. Zastosowane algorytmy

Konwersja obrazu do skali szarości, a następnie do obrazu binarnego.

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
thresh = cv2.threshold(blurred, 100, 255, cv2.THRESH_BINARY_INV)[1]
```

Podstawowe operacje dylacji oraz erozji w celu usunięcia szumów. Operacje te są wykonywane dwukrotnie. Najpierw w celu wykrycia pól czarnych , a następnie białych.

```
# czarne pola
kernel = np.ones((5, 5), np.uint8)
dilate = cv2.dilate(thresh, kernel, iterations=6)
erode = cv2.erode(dilate, kernel, iterations=7)
contours_temp = cv2.findContours(erode.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
# kontury dla czarnych pól
black_contours = imutils.grab_contours(contours_temp)


# białe pola
image_white = cv2.bitwise_not(image)
gray = cv2.cvtColor(image_white, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
thresh = cv2.threshold(blurred, 100, 255,
cv2.THRESH_BINARY_INV)[1]
kernel = np.ones((5, 5), np.uint8)
erode = cv2.erode(thresh, kernel, iterations=7)
dilate = cv2.dilate(erode, kernel, iterations=6)
```

```
contours_temp = cv2.findContours(dilate.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
# kontury dla białych pól
white_contours = imutils.grab_contours(contours_temp)

contours = black_contours + white_contoure
# sprawdzamy czy wykryliśmy wszystkie 64 pola, jeśli nie to
porzucamy tę klatkę filmu
if len(contours) != 64:
    return None
```

Pola na szachownicy występują na zmianę. W naszym przypadku zakładamy, iż lewy górny róg zaczyna się polem czarnym. W taki sposób także musimy posortować nasze znalezione pola.

```
contours_sorted = []
for i in range(len(contours)):
    if i % 2 == 0:
        contours_sorted.append(contours[int(i / 2)])
    else:
        contours_sorted.append(contours[32 + int(i / 2)])
```

Następnie sprawdzamy kontury i dla każdego konturu będącego kwadratem wyliczamy średnie współrzędne, aby otrzymać jego środek. Współrzędne każdego pola zapisujemy w liście.

```
list_of_points_list = []
list_of_points_in_one_row = []
for i, c in enumerate(contours_sorted):
    shape = detect_shape(c)
    if shape == 'square':
        # narysowanie na obrazie wynikowym ramki wokół
wykrytego pola
        cv2.drawContours(image, [c], -1, (0, 255, 0), 2)
        x = 0
        y = 0
        count = 0
        for one_measurement in c:
```

```
            x = x + one_measurement[0][0]  # współrzędna x
            y = y + one_measurement[0][1]  # współrzędna y
            count = count + 1  # count of measurements

        x = int(x / count)  # x mean of measurements
        y = int(y / count)  # y mean of measurements
        list_of_points_in_one_row.append([x, y])


# sortujemy pola każdego wiersza, tak aby wykryte pola biegły od
lewej do prawej (sortowanie po współrzędnej x)
    if len(list_of_points_in_one_row) == 8:
        list_of_points_in_one_row.sort(key=lambda x: x[0])
        list_of_points_list = list_of_points_list +
list_of_points_in_one_row
        list_of_points_in_one_row = []
```

d. Wyjście

list_of_points_list - lista zawierająca dwuelementowe listy ze współrzędnymi każdego pola. Zgodnie z kolejnością od lewej do prawej i z góry do dołu zaczynając od lewego górnego rogu.

```
def get_list_of_pawns_points(image)
```

a. Przeznaczenie

Wykrywa pionki jednego z graczy (niebieskie lub czerwone). Zarówno zwykłe pionki jak i damki.

b. Wejście

Obraz przechowywany w obiekcie typu numpy.ndarray będący wykrytą planszą zwracaną przez funkcję get_chessboard_as_image().

c. **Zastosowane algorytmy**

Algorytm nie różni się znacząco od algorytmu wykrywania pól. Wykrywanie konturów odbywa się na identycznej zasadzie, z tą oczywiście różnicą, iż wykonujemy operację jeśli funkcja detect_shape() wykryła koło (zwykły pionek) lub trójkąt (damka). Obliczanie środków wykrytych pionków także działa na tej samej zasadzie.

d. **Wyjście**

list_of_pawns_points - lista zawierająca dwuelementowe listy ze współrzędnymi zwykłych pionków. Zgodnie z kolejnością od lewej do prawej i z góry do dołu zaczynając od lewego górnego rogu.

list_of_queens_points - lista zawierająca dwuelementowe listy ze współrzędnymi damek. Zgodnie z kolejnością od lewej do prawej i z góry do dołu zaczynając od lewego górnego rogu.

```
def get_fields_info_as_list_of_lists(
list_of_fields_points,
list_of_blue_pawns_points,
list_of_blue_queens_points,
list_of_red_pawns_points,
list_of_red_queens_points, image)
```

a. **Przeznaczenie**

Z uzyskanych informacji na temat współrzędnych pól oraz pionków tworzy listę 64 wartości typu enum z klasy Fields znajdującej się w pliku fields.py.
Każda wartość wskazuje z jakim polem mamy do czynienia.

b. **Wejście**

list_of_fields_points - lista zawierająca dwuelementowe listy współrzędnych każdego pola;

list_of_blue_pawns_points - lista zawierająca dwuelementowe listy współrzędnych każdego niebieskiego zwykłego pionka;

list_of_blue_queens_points - lista zawierająca dwuelementowe listy współrzędnych każdej niebieskiej damki;

list_of_red_pawns_points - lista zawierająca dwuelementowe listy współrzędnych każdego czerwonego zwykłego pionka;

list_of_red_queens_points - lista zawierająca dwuelementowe listy współrzędnych każdej czerwonej damki;

image - obraz źródłowy przechowywany w obiekcie typu numpy.ndarray do zaznaczenia wykrytych pionków.

c. Zastosowane algorytmy

Z góry wiemy, że plansza zawiera pola czarne i białe na zmianę. Tak jak wcześniej było wspomniane zakładamy, iż lewy górny róg planszy zawsze zaczyna się polem czarnym. Wstępnie ustawiamy tablice wynikową, która reprezentuje szachownicę bez pionków.

```
result = []
counter = 0
for i in range(64):
    if (i + int(counter / 8)) % 2 == 0:
        result.append(Field.BLACK)
    else:
        result.append(Field.WHITE)
    counter += 1
```

Następnie iterujemy po liście wszystkich 64 pól i sprawdzamy, czy współrzędne każdego z pionków nie pokrywają się z współrzędnymi danego pola. Jeśli współrzędne z pewną granicą błędu (ERROR_LIMIT) pokrywają się, to stwierdzamy, iż w tym miejscu znajduje się dany rodzaj pionka. Dodatkowo do zwizualizowania wykrytych pionków oznaczamy je okręgami o odpowiednich kolorach opisanych we wstępie modułu.

```
for id, field in enumerate(list_of_fields_points):
    for pawn in list_of_blue_pawns_points:
        if abs(field[0] - pawn[0]) <= ERROR_LIMIT and abs(field[1]
- pawn[1]) <= ERROR_LIMIT:
```

```
            cv2.circle(image, (field[0], field[1]), 10, (0, 0,
255), 3)
            result[id] = Field.BLACK_FIELD_BLUE_PAWN
            break

    for pawn in list_of_blue_queens_points:
        if abs(field[0] - pawn[0]) <= ERROR_LIMIT and abs(field[1]
- pawn[1]) <= ERROR_LIMIT:
            cv2.circle(image, (field[0], field[1]), 10, (0, 0,
127), 3)
            result[id] = Field.BLACK_FIELD_BLUE_QUEEN
            break

    for pawn in list_of_red_pawns_points:
        if abs(field[0] - pawn[0]) <= ERROR_LIMIT and abs(field[1]
- pawn[1]) <= ERROR_LIMIT:
            cv2.circle(image, (field[0], field[1]), 10, (255, 0,
0), 3)
            result[id] = Field.BLACK_FIELD_RED_PAWN
            break

    for pawn in list_of_red_queens_points:
        if abs(field[0] - pawn[0]) <= ERROR_LIMIT and abs(field[1]
- pawn[1]) <= ERROR_LIMIT:
            cv2.circle(image, (field[0], field[1]), 10, (127, 0,
0), 3)
            result[id] = Field.BLACK_FIELD_RED_QUEEN
            break
```

Na koniec tworzymy listę składającą się z 8 elementowych list. Każda z nich reprezentuje stan jednego wiersza szachownicy, zaczynając od góry.

```
final_result = []
temp_result = []
for i, each_field in enumerate(result):
    if i % 8 == 0 and i != 0:
        final_result.append(temp_result)
        temp_result = []
    temp_result.append(each_field)
final_result.append(temp_result)
```

d. **Wyjście**

result - lista zawierająca 8 list, każda po 8 obiektów typu enum klasy Fields znajdującej się w pliku fields.py. Przedstawia aktualny stan planszy, tzn. jaki pionek znajduje się na jakim polu.

```
def detect(camera_image, last_result):
```

a. **Przeznaczenie**

Odpowiada za przeprowadzenie wszystkich wymaganych operacji do wykrycia stanu planszy na otrzymanej klatce obrazu.

b. **Wejście**

camera_image - aktualnie przetwarzana klatka przechowywana w obiekcie typu numpy.ndarray;

last_result - lista zawierająca 8 list, każda po 8 obiektów typu enum klasy Fields znajdującej się w pliku fields.py. Reprezentuje ostatni prawidłowy stan planszy. Jest zwracana w razie wystąpienia jakiegokolwiek błędu w module rozpoznawania obrazu.

c. **Zastosowane algorytmy**

Z powodu pracy na niskiej rozdzielczości obrazu 640x480 pikseli (co jest pożądane w większości przypadków, ponieważ możemy zaoszczędzić dużo mocy obliczeniowej) pojawiał się problem w wykrywaniu kolorów tak małych obiektów jak pionki. Rozwiązaniem okazało się sztuczne przeskalowanie obrazu jaki dostaje funkcja. Zwiększyliśmy szerokość oraz długość 2.5 krotnie (ukazane w opisie funkcji detect()). Przy wykrywaniu poruszamy się w przestrzeni barw HSV, dlatego też dokonaliśmy konwersji obraz z RGB do HSV.

```
image = get_chessboard_as_image(camera_image)
    imgScale = 2.5
    newX, newY = image.shape[1] * imgScale, image.shape[0] *
```

```
imgScale
   image_resized = cv2.resize(image, (int(newX), int(newY)))
   image_HSV = cv2.cvtColor(image_resized.copy(),
cv2.COLOR_BGR2HSV)
```

Następnie ustalamy przedział górny oraz dolny dla barwy niebieskiej i tworzymy maskę reprezentującą tylko kolor niebieski w na obrazie. Do usunięcia szumów używamy operacji dylacji i erozji.

```
  lower_blue = np.array([60, 95, 120])
  upper_blue = np.array([145, 255, 255])
  blue_mask = cv2.inRange(image_HSV, lower_blue, upper_blue)
  kernel = np.ones((5, 5), np.uint8)
  blue_mask = cv2.erode(blue_mask, kernel, iterations=1)
  blue_mask = cv2.dilate(blue_mask, kernel, iterations=2)
```

Przy wyszukiwaniu samych kształtów mała rozdzielczość nie miała negatywnego wpływu na wyniki, więc wracamy do poprzedniej rozdzielczości, aby pracować na mniejszej ilości danych, a co za tym idzie oszczędzamy mocy obliczeniowej. Tak przygotowany obraz przekazujemy do funkcji get_list_of_pawns_points() znajdującej pionki

```
imgScale = 0.4
newX, newY = blue_mask.shape[1] * imgScale, blue_mask.shape[0] *
imgScale
blue_mask = cv2.resize(blue_mask, (int(newX), int(newY)))
blue_pawns, blue_queens =get_list_of_pawns_points(image=blue_mask)
```

Analogiczne operacje przeprowadzamy do wykrywania pionków czerwonych z tą różnicą, że nie potrafiliśmy znaleźć idealnego przedziału do wykrywania koloru niebieskiego. Za to udało się znaleźć przedział wykrywający jednocześnie kolor niebieski oraz czerwony. Wykorzystaliśmy to i po przeprowadzeniu operacji części wspólnej maski dla koloru niebieskiego z czerwonym oraz negacji maski dla koloru niebieskiego otrzymaliśmy maskę dla koloru czerwonego.

```
lower_red_and_blue = np.array([0, 120, 130])
upper_red_and_blue = np.array([180, 255, 255])
red_and_blue_mask = cv2.inRange(image_HSV, lower_red_and_blue,
upper_red_and_blue)
kernel = np.ones((5, 5), np.uint8)

red_and_blue_mask = cv2.erode(red_and_blue_mask, kernel,
iterations=1)
red_and_blue_mask = cv2.dilate(red_and_blue_mask, kernel,
iterations=2)

imgScale = 0.4
newX, newY = red_and_blue_mask.shape[1] * imgScale,
red_and_blue_mask.shape[0] * imgScale
red_and_blue_mask = cv2.resize(red_and_blue_mask, (int(newX),
int(newY)))
red_mask = cv2.bitwise_and(red_and_blue_mask,
cv2.bitwise_not(blue_mask))
red_pawns, red_queens = get_list_of_pawns_points(image=red_mask)
```

Na końcu używamy funkcji get_fields_as_list_of_points() do wykrycia pozycji wszystkich pól i przekazujemy wszystkie niezbędne parametry do funkcji get_fields_info_as_list_of_lists(), która zwraca nam aktualny stan planszy.

```
fields = get_fields_as_list_of_points_list(image)

info_about_each_field = get_fields_info_as_list_of_lists(
list_of_fields_points=fields, image=image,
list_of_blue_pawns_points=blue_pawns,
list_of_blue_queens_points=blue_queens,
list_of_red_pawns_points=red_pawns,
list_of_red_queens_points=red_queens)
```

Jeżeli w trakcie jakiejkolwiek wykonywanej operacji wydarzy się błąd, to zwracamy ostatni poprawny stan planszy jaki otrzymaliśmy w parametrze, tzn. last_result.

d. Wyjście

W przypadku poprawnego wykonania wszystkich operacji:

image - aktualnie przetwarzana klatka przechowywana w obiekcie typu numpy.ndarray z zaznaczonymi wykrytymi pionkami;

info_about_each_field - lista zawierająca 8 list, każda po 8 obiektów typu enum klasy Fields znajdującej się w pliku fields.py. Reprezentuje aktualny stan planszy.

W przypadku wystąpienia jakiegokolwiek błędu:

camera_image - aktualnie przetwarzana klatka przechowywana w obiekcie typu numpy.ndarray;

last_result - lista zawierająca 8 list, każda po 8 obiektów typu enum klasy Fields znajdującej się w pliku fields.py. Reprezentuje ostatni prawidłowy stan planszy. Jest zwracana w razie wystąpienia jakiegokolwiek błędu w module rozpoznawania obrazu.

Moduł walidacyjny

Opis

Przeznaczeniem modułu jest określenie czy ruch wykonany przez gracza jest zgodny z przyjętymi przez nas zasadami gry w warcaby. Ponieważ istnieje wiele wariantów gry w warcaby, wybraliśmy jeden i to o jego zasady opieramy całą logikę w projekcie. Zasady są następujące:

A. Pole gry to szachownica o wymiarach 8x8 pól pokolorowanych na przemian na czarno i biało.
B. Każdy z graczy posiada 12 pionów, gracz pierwszy posiada piony niebieskie, gracz drugi posiada piony czarne. [tu można wstawić zdjęcie poglądowe od nas screena]
C. Gracz grający białymi pionami pierwszy wykonuje ruch. Następnie gracze wykonują ruchy na przemian.

D. Piony mogą poruszać się po ukosie o jedno pole do przodu i tylko wyłącznie do przodu.
E. Bicie następuje poprzez przeskoczenie piona lub królowej przeciwnego koloru na pole znajdujące się za nim o ile nie jest ono zajęte przez innego piona bądź królową dowolnego koloru.
F. Bicie może się odbywać za pomocą zwykłych pionów wyłącznie w przód czyli w kierunku przeciwnego krańca do miejsca początkowego na szachownicy. Możliwość bicia do tyłu istnieje tylko, jeśli nie jest to pierwszy bicie w sekwencji zbijania wielokrotnego albo jeżeli figurą bijącą jest figura królowej.
G. Jeżeli po zbiciu piona istnieje w otoczeniu piona, którym wykonano bicie, inny pion, który może zostać zbity - tj. przeciwnego koloru pion lub królowa, a pole za nim jest puste - można wykonać bicie wielokrotne, ale nie jest to obowiązkowe
H. W wielu wersjach gry, bicia pionów są obowiązkowym ruchem, który musi zostać wykonany. W naszym wariancie gry takiego obowiązku nie ma.
I. Jeżeli pion dotrze do końca szachownicy po przeciwnej stronie od miejsca jego startu, awansuje na królową.
J. Jeżeli pion dotarł bo do końcowego pola po drodze, czyli w trakcie bicia wielokrotnego - nie musi awansować na królową, może dalej bić w ciągu kolejne piony.
K. Królowa może poruszać się o dowolną ilość pól w dowolnym kierunku.
L. Bicie królową następuje przez przeskoczenie w dowolnym kierunku jednego piona lub królowej, przeciwnego koloru, który posiada za sobą puste pole.
M. W wielu wariantach gry, królowa po biciu może przeskoczyć w dowolne miejsce za pionkiem - w naszym wariancie gry może przeskoczyć tylko na miejsce za nim.
N. Wygraną w grze osiąga się poprzez wyeliminowanie możliwości ruchu u przeciwnika. Drogi do tego celu są dwie:
    a. żaden pionek przeciwnika nie może wykonać ruchu
    b. nie istnieje żaden pionek przeciwnika - wtedy z natury, nie będzie on mógł wykonać ruchu.

Wejście

Na wejściu modułu sprawdzającego zamieszczamy dwie tablice dwuwymiarowe, o wymiarach 8 x 8, przechowujące pola typu Field:

- Field.BLACK
- Field.WHITE
- Field.BLACK_WITH_BLUE_PAWN
- Field.BLACK_WITH_RED_PAWN
- Field.BLACK_WITH_BLUE_QUEEN
- Field.BLACK_WITH_RED_QUEEN

Każde z tych pól oznacza, co znajduje się na danym polu na szachownicy.
Pierwsza z tablic traktowana jest jako stan szachownicy przed wykonaniem ruchu.
Druga tablica traktowana jest jako stan szachownicy po wykonaniu ruchu.
Poza tym na wejściu podajemy jeszcze pole enumeracyjne typu Player, które może posiadać następujące wartości:

- Player.BLACK
- Player.WHITE
- Player.WHITE_CAPTURE
- Player.BLACK_CAPTURE

Player.BLACK oznacza, że aktualnie wykonywany ruch powinien wykonać gracz grający pionami czarnymi.
Player.WHITE oznacza, że aktualnie wykonywany ruch powinien wykonać gracz grający pionami białymi.
Player.WHITE_CAPTURE oznacza, że ruch wykonany może być przez obie strony ponieważ w poprzednim ruchu nastąpiło bicie przez białego gracza - oznacza to, że biały może wykonać ruch pod warunkiem, że będzie to kolejne bicie piona lub królowej koloru czarnego, pionem lub królową białą, którą wykonaliśmy poprzednie bicie. Równocześnie zwykły ruch lub bicie może wykonać gracz czarny, ponieważ biały mógł nie zauważyć okazji do zbicia piona.
Player.BLACK_CAPTURE oznacza, że ruch wykonany może być przez obie strony ponieważ w poprzednim ruchu nastąpiło bicie przez czarnego gracza - oznacza to, że czarny może wykonać ruch pod warunkiem, że będzie to kolejne bicie piona lub królowej koloru białego, pionem lub królową czarną, którą wykonaliśmy poprzednie

bicie. Równocześnie zwykły ruch lub bicie może wykonać gracz biały, ponieważ czarny mógł nie zauważyć okazji do zbicia piona.

Algorytmy

Algorytm nie został oparty na żadnej istniejącej zasadzie sprawdzania poprawności ruchów w grach planszowych.

```
if len(self.Differences) == 0:
    self.SuccessMessage = 'No differences'
    return True, player
if len(self.Differences) == 1:
    return self.validate_pawn_promotion(player)
if len(self.Differences) == 2:
    return self.validate_normal_move(player)
if len(self.Differences) == 3:
    return self.validate_capture_move(player)
return False, player
```

Zakłada on porównania dwóch tablic wejściowych i zliczenie wszystkich miejsc, które różnią się wartościami. Z racji, że bicia w ciągu muszą dokonywać się poprzez de facto pojedyncze bicia, wykonane po prostu przez jednego gracza, wyróżniamy 5 możliwości porównywania tablicy:

- 0 różnic - oznacza to, że nie został wykonany żaden ruch i do modułu graficznego zwracana jest taka też informacja.
- 1 różnica - oznacza to, że jedyny poprawny ruch to promocja pionka w damkę - każdy inny “ruch” oznaczałby pogwałcenie zasad gry, a co za tym idzie zwrócenie błędu modułowi graficznemu.
- 2 różnice - oznacza to, że jedynym poprawnym ruchem będzie przesunięcie piona o jedno pole lub damki o dowolną ilość pól. Sprawdzone wiec zostanie czy ruch wykonany był przez gracza, który mógł się ruszyć, czy został wykonany na wolne od innych pionów miejsce, czy przy okazji nie zniknął z szachownicy żaden inny pion, oraz czy pion, który został podniesiony z szachownicy jest tym samym pionem, który wylądował kawałek dalej -

innymi słowy, czy jeżeli ruch zaczął pion biały to czy również skończył go pion biały. Każde odstępstwo od tej normy jest traktowane jako błąd i tak też zwracane modułowi graficznemu.

- 3 różnice - oznacza to że są dwie poprawne możliwości powstania takiej sytuacji:
    - bicie - jeżeli pion jednego koloru, zniknął z okupowanego wcześniej pola, pojawił się na o dwa pola dalej na pustym polu w kierunku zgodnym z jego kierunkiem podążania, a między miejscem początkowym a miejscem docelowym zniknął pion lub damka przeciwnego koloru - bicie było poprawne. W każdym innym przypadku bicie było niezgodne z zasadami.
    - bicie wielokrotne - jeżeli pion jednego koloru, zniknął z zajętego wcześniej pola, pojawił się na polu dwa pola dalej w dowolnym kierunku na pustym polu, a między miejscem początkowym a docelowym zniknął pion lub damka przeciwnego koloru - bicie było poprawne. Każdy inny przypadek zostanie uznany za fałszywy.
- 4 różnice lub więcej - nie istnieje legalny ruch, który mógłby spowodować na raz 4 lub więcej różnic na planszy w naszej wersji gry - w przypadku wystąpienia takiego stanu, zwrócony zostanie błąd modułowi graficznemu.

Klasy

W algorytmie wykorzystywane są 3 klasy:

- `MoveValidation` - zawiera w sobie funkcje odpowiedzialne za logikę i stanowi trzon systmu weryfikacji.
  Pola:
    - `Differences` - tablica różnic
    - `Previous` - tablica przechowująca stan przed wykonaniem ruchu
    - `Current` - tablica przechowująca stan po wykonaniu ruchu
    - `ErrorMessage` - pole przechowujące wiadomość o błędzie operacji
    - `SuccessMessage` - pole przechowujące wiadomość o sukcesie operacji

- `PieceCounter` - pomocnicza klasa służąca przechowaniu informacji o liczebności figur na szachownicach.
  Pola:
  - `Previous`
  - `Current`
  - `Previous_black`
  - `Previous_white`
  - `Current_black`
  - `Current_white`

  Wszystkie pola przechowują liczbę odpowiednich figur.

- `Position` - pomocnicza klasa służąca przechowaniu informacji o położeniu danego pionka.
  Pola:
  - `x` - oznacza pozycję na współrzędnej x na szachownicy
  - `y` - oznacza pozycję na współrzędnej y na szachownicy

Funkcje

Każda analiza za pomocą algorytmu dzieli się na dwa etapy - wczytanie tablic i porównanie zmian oraz analiza różnic:

W celu pierwszego etapu użyta zostaje funkcja `MoveValidation.compare_boards(self, previous, current)`

Parametry:

previous - oznacza tablicę stanu szachownicy przed wykonaniem ruchu

current - oznacza tablicę stanu szachownicę po wykonaniu ruchu

Po jej wywołaniu należy wywołać metodę:

`MoveValidation.validate_move(self, player)`

Parametry:

player - pole typu Player, określa czyj ruch został wykonany

Powyższa funkcja wywołuje kolejno w zależności od napotkanych różnic jedną z funkcji:

- `MoveValidation.validate_pawn_promotion(self, player)`
  Funkcja ta określa czy wykonany ruch był zamianą piona na królową.
  Jeżeli tak to zwraca prawdę, jeżeli nie to fałsz
  Parametry:
  player - pole typu Player, określa czyj ruch został wykonany

- `MoveValidation.validate_normall_move(self, player)`
  Funkcja określa czy wykonany ruch był poprawnym zwykłym ruchem gracza, który powinien wykonać teraz ruch. Zwraca prawdę jeżeli ruch był zgodny z zasadami, fałsz jeżeli nie był zgodny. Dodatkowo zwraca gracza, który może wykonać w kolejnym ruchu swój ruch.
  Używa funkcji:
  - `MoveValidation.count_pieces_on_diagonal_between(move_from, move_to)` do określenia poprawności ruchu dla damek

  Parametry:
  `player` - pole typu `Player`, określa czyj ruch został wykonany

- `MoveValidation.validate_capture_move(self, player)`
  Funkcja określa czy wykonany ruch był biciem wykonanym przez odpowiedniego gracza zgodnym z zasadami. Zwraca prawdę jeżeli ruch był zgodny z zasadami, fałsz jeżeli nie zgodny.
  Dodatkowo zwraca gracza, który może wykonać w kolejnym ruchu swój ruch.
  Używa funkcji:
  - `MoveValidation.count_pieces_on_diagonal_between(move_from, move_to)` do określenia poprawności ruchu dla damek.
  - `MoveValidation.there_is_possible_capture(move_to)`

  Parametry:
  player - pole typu Player, określa czyj ruch został wykonany

  Poniżej przedstawione są pozostałe funkcje, które miały charakter pomocniczy dla wyżej wymienionych:

- `MoveValidation.there_is_possible_capture(self, move_to)` - funkcja sprawdzająca, czy dla podanego piona, który wykonał bicie istnieje kolejny pion, który może zostać zbity. Zwraca prawdę jeżeli można bić dalej, fałsz jeśli nie. Jako parametr przyjmuje pozycje piona, który wykonał ostatnie bicie.
- `MoveValidation.count_pieces(self)` - funkcja licząca wszystkie znaczące ilości pionów na obydwu stanach szachownicy. Zwraca pole typu `PieceCounter.`
- `MoveValidation.count_pieces_on_diagonal_between(self, move_from, move_to, color=None)` - funkcja licząca wszystkie pionki, pomiędzy dwoma zadanymi polami typu Position. Posiada opcjonalny parametr oznaczający jakiego koloru piony chcemy zliczać.
- `MoveValidation.is_a_white_figure(figure)` - funkcja pomocnicza określająca czy element wejściowy figure typu Field jest figurą białą.
  Zwraca prawdę lub fałsz
- `MoveValidation.is_a_black_figure(figure)` - funkcja pomocnicza określająca czy element wejściowy figure typu Field jest figurą czarną.
  Zwraca prawdę lub fałsz

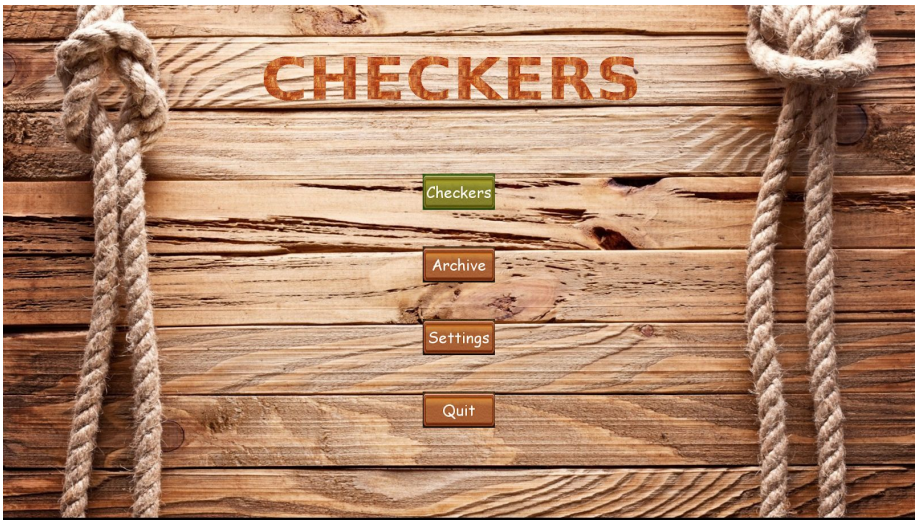
Wyjście

Główne części modułu zwracają na wyjściu dwie informacje: Czy ruch był poprawny oraz gracza, który może wykonać następny ruch. Informacja ta trafia do modułu graficznego, który na jej podstawie wyświetla odpowiednią zmianę na planszy, lub też komunikuje o błędzie użytkownika.

Moduł aplikacji

a. Opis działania

Moduł ten dostarcza interfejs użytkownika składający się z czterech okien, z którego każde wykonuje inną funkcję, zapewniając użytkownikowi możliwość obsługi funkcjonalności programu w formie graficznej.
Użytkownik może się sprawnie poruszać między oknami dzięki oknu menu.



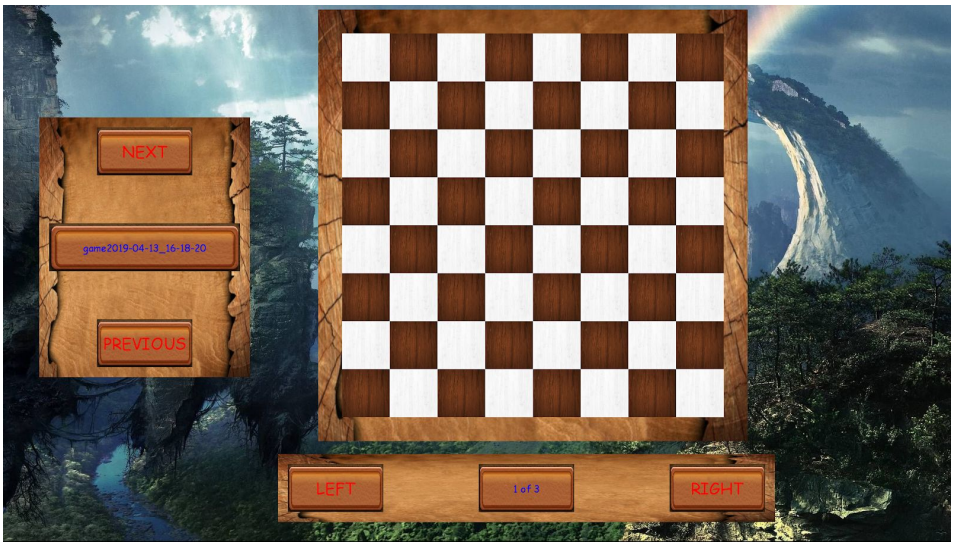
Rysunek 8. Menu

Okno Checkers pozwala na skorzystanie z głównej funkcjonalności aplikacji jaką jest nagrywanie gry i przenoszenie jej stanu na zwizualizowaną planszę. Podczas wizualizacji dodatkowo sprawdzane jest czy ruch wykonywany jest poprawny.

W oknie tym możliwe jest również wybranie nazwy oraz zapisanie rozgrywki do archiwum.

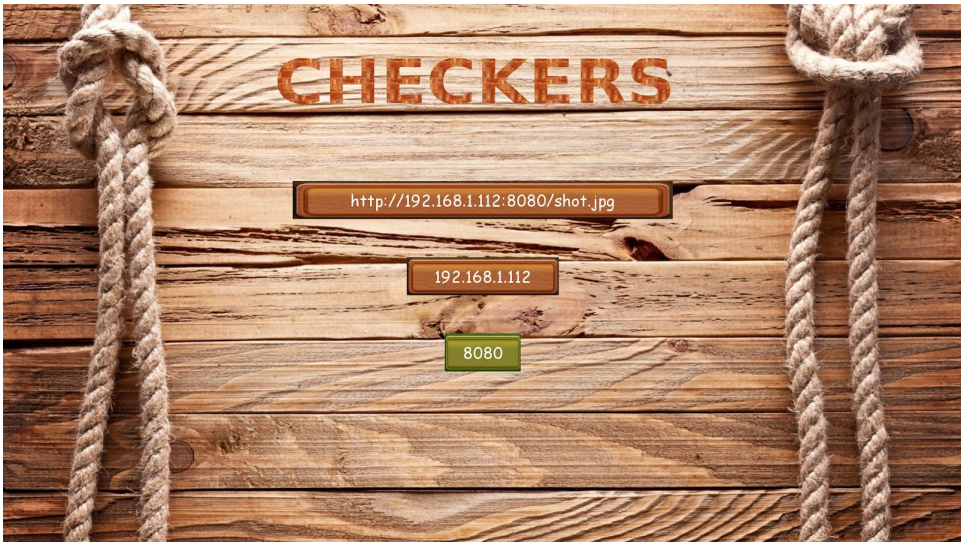
Rysunek 9. Okno checkers

Okno Archive pozwala na przeglądanie wcześniej zapisanych gier ruch po ruchu.



Rysunek 10. Okno archive

Okno Settings pozwala na ustawienie adresu z którego program ma pobierać obraz do analizy ruchów.



Rysunek 11. Okno settings

Kolejnym zadaniem Modułu aplikacji jest połączenie i wykorzystanie Modułu rozpoznawania oraz Modułu Walidacyjnego

b. Struktura działania

Program uruchamiany jest w main.py które startuje menu.py.
Wszystkie okna działają niezależnie i są startowane poprzez stworzenie konstruktora klasy ich obsługującej oraz uruchomienia metody main() poszczególnej klasy. Każde okno posiada własny plik konfiguracyjny
a inicjalizacja potrzebnych elementów odbywa się w __init__(self).

Główny wątek który zajmuje się rysowaniem Menu jest przejmowany przez inne okna w momencie ich wybrania, co pozwala zachować stan Menu z przed jego opuszczenia.

Menu

1. **Przeznaczenie**

Pozwolenie użytkownikowi na sprawne poruszanie się po funkcjonalnościach aplikacji, zapewnienie graficznego interfejsu użytkownika.

2. **Inicjalizacja**

Należy zaimportować plik menu.py i wywołać MenuWindow().main() co spowoduje utworzenie okna i wszystkich elementów które się na nim znajdować powinny w tym:

a. przyciski
b. tekstury

3. **Metody**

run()

główna pętla, wywołuje mechanizmy takie jak sprawdzanie zdarzeń
i rysowanie obiektów.

handle_events()

zajmuje się odczytywaniem wszystkich eventów znajdujących się
w pg.event.get() oraz obsługą tych zdarzeń które są istotne dla menu:

a. Przycisk Enter - wejście w aktualnie wybraną opcję.
b. Wciśnięcie ESC - opuszczenie aplikacji.

c. Obsługa wybierania wyświetlanych tekstur dla przycisków.
d. Wciśnięcie przycisku myszy - zależnie czy kursor znajduje się na celu (wejście) lub nie (brak reakcji).

draw()

metoda zajmująca się rysowaniem wszystkich elementów widocznych dla użytkownika:

a. Rysowaniem tła.
b. Rysowaniem przycisków.

4. **Działanie**

Okno działa w pętli do momentu zmienienia się stanu zmiennej **_done** na True, co sygnalizuje, że użytkownik zażyczył sobie zakończenia programu. Użytkownik może poruszać się przy pomocy strzałek góra, dół, oraz przy pomocy myszki. Ostatni oznaczony przycisk zostaje co powoduje jego podświetlenie
i wybranie w przypadku wciśnięcia przycisku Enter. Możliwe też jest wybranie opcji przy pomocy najechania na nie kursorem
i wciśnięcia LPM.

Checkers

1. **Przeznaczenie**

Pozwala użytkownikowi na obserwowanie, analizę
i wizualizację stanu rozgrywki. Wykonuje to poprzez pobieranie obrazu z podanego URL.

2. **Inicjalizacja**

Należy zaimportować plik checker_window.py i wywołać CheckersWindow().main() co spowoduje utworzenie okna i wszystkich elementów które się na nim powinny znajdować.

3. **Metody**

handle_events()

Obsługa zdarzeń

a. wciśnięcia Esc jako wyjście
b. obsługa zapisywania znaków wpisywanych do pola tekstowego

get_camera_frame()

Aktualizuje obraz na nowo wysłany przez kamerę. W sytuacji gdy obrazu nie ma, wyświetla domyślną pustą planszę. Adres URL z którego obraz jest pobierany znajduje się w pliku url.txt, jego zawartość jest edytowana w oknie Settings.

save_game()

Zapisuje aktualną rozgrywkę do pliku games_archive. Przez rozgrywkę rozumiemy wszystkie stany gry od uruchomienia okna, lub od momentu wciśnięcia przycisku reset.

draw()

Zajmuje się rysowaniem elementów znajdujących się w oknie:

a. Planszy
b. Przycisków
c. Okna z widokiem z kamery
d. Obramowań
e. Pola z komunikatami
f. Tła

run_logic()

Zajmuje się pobieraniem najnowszej wersji obrazu, następnie sprawdza czy obraz jest poprawny. Przez poprawny rozumiemy czy moduł rozpoznawania obrazu zaakceptował go, następnie gdy akceptacja zostanie potwierdzona następuje sprawdzenie logiki, tzn czy ruch wykonany przez graczy był ruchem prawidłowym. Jeśli ruch się zgadza dopiero wtedy stan rozgrywki zostaje zaktualizowany i możliwe jest wyświetlenie aktualnego stanu z planszy. Jeśli aktualizacja obrazu się

powiedzie, obraz zostanie zaakceptowany lecz sama logika się nie zgadza wtedy zostaje użytkownikowi pokazany komunikat informujący o rodzaju błędu.

init_textures()

Wczytuje i przypisuje do zmiennych wszystkie tekstury potrzebne do wyświetlania danego okna, źródło textur zdefiniowane jest w pliku konfiguracyjnym opisującym to okno.

run()

Jest to główna pętla zajmująca się rysowaniem okna oraz sprawdzaniem zdarzeń.

4. **Działanie**

Okno to działa na dwóch wątkach, z czego jeden główny zajmuje się aktualizacją obrazu i interakcji z użytkownikiem, co zapewnia płynne korzystanie z programu, natomiast drugi zajmuje się wykonywaniem run_logic() który aktualizuje stan rozgrywki i komunikaty. Wątki te działają niezależnie od siebie więc nie ma żadnych komplikacji.

Archive

1. **Przeznaczenie**

Udostępnienie użytkownikowi interfejsu do wglądu i analizy rozgrywek, które użytkownik sobie zapisał. Możliwe jest wybranie dowolnej gry po czym przewinięcie jej ruch po ruchu(działa w obie strony)do samego końca zapisu.

2. **Inicjalizacja**

Należy zaimportować plik archive_window.py i wywołać ArchiveWindow().main() co spowoduje utworzenie okna i wszystkich elementów które się na nim powinny znajdować.

3. **Metody**

load_file_names()

Wczytuje nazwy plików znajdujące się w katalogu games_archive/, po czym dodaje je do listy dostępnych historii.

load_game()

Wczytuje grę na podstawie nazwy aktualnie wybranego pliku z listy wcześniej zaczytanej, gry są zapisane w formie json jako tablica (gra) tablic (stanów w grze) posiadająca tablice które są kolejnymi rzędami

draw()

Rysuje wszystkie elementy znajdujące się w tym oknie:

a. przyciski
b. pola tekstowe przedstawiające aktualny stan i plik
c. tekstury
d. obramówki
e. plansza
f. opcjonalne pionki na planszy

handle_events()

Zajmuje się obsługa zdażeń:

a. kliknięć myszką
b. Esc jako wyjście z okna

run()

Główna pętla klasy, sprawdza eventy i rysuje zawartość okna w każdym obiegu.

init_textures()

Wczytuje tekstury i przypisuje do zmiennych.

4. Działanie

Po włączeniu okna, zaczytywane są wszystkie pliki znajdujące się w archiwum i umieszczane na liście historii. Następnie użytkownik może wybrać przy pomocy przycisków będących po lewej stronie która historia rozgrywki go interesuje, i przy pomocy kolejnych przycisków znajdujących się pod planszą wizualizującą stan gry, może wybrać do którego ruchu chce się przenieść.

Settings

1. **Przeznaczenie**

   Modyfikacja adresu URL przy pomocy interfejsu użytkownika.

2. **Inicjalizacja**

   Należy zaimportować plik settings.py i wywołać SettingsWindow().main() co spowoduje utworzenie okna i wszystkich elementów które się na nim znajdować powinny w tym:

   a. pole tekstowe wartościami z pliku url.txt

3. **Metody**

draw()

Rysuje i aktualizuje edytowalne pola tekstowe oraz aktualizuje pole podsumowujące aktualny wygląd URL.

load_settings()

Wczytuje zawartości pliku URL.txt i na podstawie podanych tam danych ustawia wartości parametrów ip oraz port.

save_settings()

Pobiera wartości aktualnie przypisane do zmiennych port, ip i URL oraz tworzy z nich jsona którego następnie zapisuje do pliku URL.txt.

init_textures()

Wczytuje tekstury i przypisuje do zmiennych.

change_url()

Uruchamiane po każdym wpisaniu dowolnego znaku do jednego z pól wejściowych(port lub ip). Pobiera aktualne wartości IP i port, po czym tworzy z tego odpowiedni adres do pobierania zdjęcia.

handle_events()

Zajmuje się obsługą zdarzeń takich jak:

a. ESC jako zakończenie edytowania pola, lub wyjście do menu.
b. Obsługa pobierania i zapisywania znaków do pól tekstowych oraz aktualizowanie wyświetlanych informacji.
c. Sprawdzanie położenia myszki podczas kliknięć i uruchamianie odpowiednich reakcji.

run()

Główna pętla oka, wywołuje sprawdzenie zdarzeń oraz aktualizację obrazu w oknie.

4. **Działanie**

Po wejściu do okna, użytkownik ma możliwość zmienić adres URL używany przez aplikację do pobierania zdjęć/obrazu z kamery. Możliwe jest modyfikowanie adresu i sprawdzanie zmian w czasie rzeczywistym na aktualizującym się polu wyświetlającym najnowszą zawartość URL-a.

Dodatkowe elementy

Klasa Przycisk

1. **Przeznaczenie**

Pomocnicza klasa Button dziedzicząca po pygame.sprite.Sprite. Umożliwiające szybkie dodawanie kolejnych przycisków do projektu.

2. **Inicjalizacja**

Zaimportowanie button.py i wywołanie klasy Button z konstruktorem zawierającym:

- x - (początkowe położenie na osi x)
- y - (początkowe położenie na osi y)
- width - szerokość przycisku
- height - wysokość przycisku
- callback - funkcja wywoływana po wciśnięciu przycisku
- font - rodzaj czcionki napisu na przycisku
- text - napis wyświetlany na przycisku
- text_color - kolor tekstu wyświetlanego na przycisku w formacie RGB
- image_normal* - podstawowy wygląd przycisku
- image_hover* - wygląd przycisku po najechaniu na niego
- image_down* - wygląd przycisku po wciśnieciu go

*wygląd przycisku musi być pygame.Surface

3. **Metody**

handle_event()

Zajmuje się obsługą eventów kliknięcia myszką, sprawdza czy podczas kliknięcia, myszka znajdowała się nad obiektem przycisku.

set_text(text)

Wymienia aktualnie pojawiającą się zawartość tekstu na nową podaną w parametrze **text,** który musi być stringiem

4. **Folder z historią gier**

Nazwa folderu: games_archive

Zawartość: Pliki zawierające historię rozgrywek zapisane w formacie json. Pliki domyślnie przyjmują nazwę w postaci game_rrrr-mm-dd_hh-mm-ss, jednakże użytkownik może wybrać własną nazwę pliku, która nie jest w żaden sposób ograniczana.

Charakterystyka zawartości: Każdy plik zawiera tablice tablic, tablic gdzie

najmniejsza tablica przedstawia jeden wiersz z planszy.

Cała tablica tych wierszy przedstawia jeden stan, całkowity wygląd planszy w danej chwili, natomiast zbiór takich stanów przedstawia całą rozgrywkę.

Przykładowy stan z gry:

```
stan    =    [[0, 1, 0, 1, 0, 1, 0, 1],
              [1, 0, 1, 0, 1, 0, 1, 0],
              [0, 1, 0, 1, 0, 1, 0, 1],
              [1, 0, 1, 0, 1, 0, 1, 0],
              [0, 1, 0, 1, 0, 1, 0, 1],
              [1, 0, 1, 0, 1, 0, 1, 0],
               [0, 1, 0, 1, 0, 1, 0, 1],
              [1, 0, 1, 0, 1, 0, 1, 0]]
```

gdzie:

1 - pole czarne

2 - pole białe

3 - pionek czerwony

4 - pionek niebieski

5 - damka czerwona

6 - damka niebieska

Powyżej przedstawiony format pozwala w łatwy sposób przemieszczać się po każdym stanie zarówno w następne jak i poprzednie ruchy wykonane przez gracza.

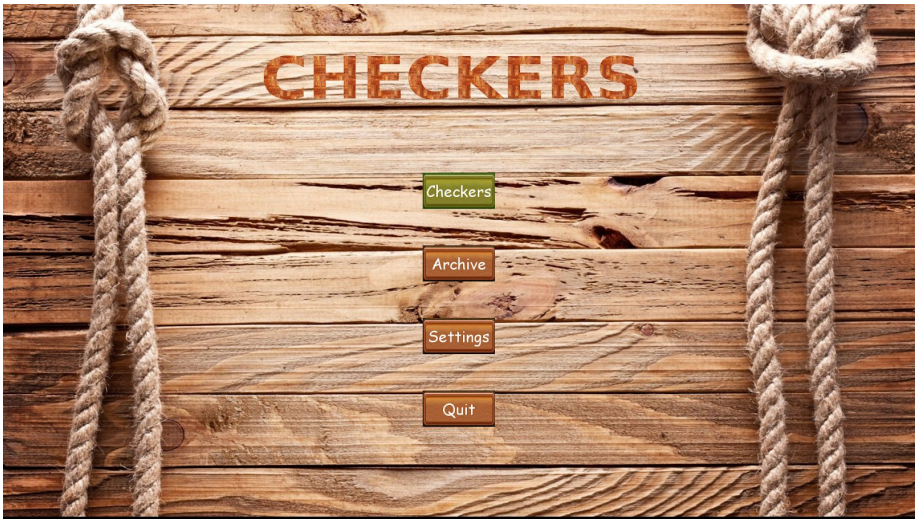
7. Instrukcja użytkowania aplikacji

1. Przygotowanie do rozgrywki:
    a. wydrukowanie planszy, ustawienie jej w dobrze oświetlonym miejscu;
    b. przygotowanie pionków (koła) oraz damek (trójkąty) w kolorze niebieskim

i czerwonym w dowolnej ilości (rekomendowana ilość - po 12 z każdego rodzaju);

c. zainstalowanie aplikacji mobilnej IP Webcam.

2. Po uruchomieniu pliku main.py, otwiera się graficzne menu aplikacji. Po menu można poruszać się strzałkami góra/dół. `Enter` oraz `Esc` jest używany do otwierania nowego okna oraz powrotu do poprzedniego. Wyłączyć program można za pomocą przycisku `Quit` lub naciśnięcia `Esc` w ekranie menu. Naciskać na guziki można także kursorem.



Rysunek 12. Menu główne

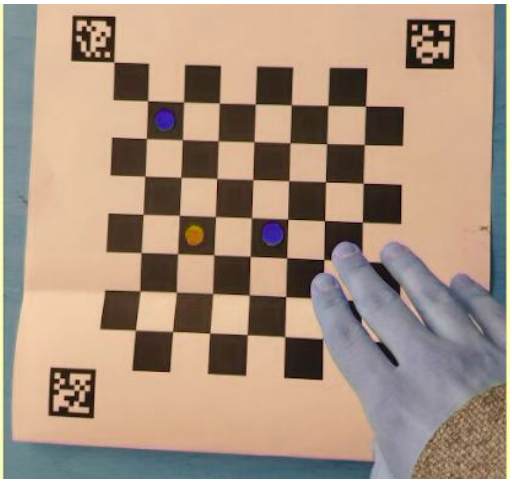
3. W oknie `Settings` należy ustawić adres IP oraz port, na którym działa aplikacja IP Webcam, aby umożliwić aplikacji dostęp do obrazu z urządzenia.

Rysunek 13. Aplikacja IP Webcam: w oknie `Settings` należy wpisać adres IP oraz port widoczny na tym ekranie

4. W oknie Checkers można rozpocząć rozgrywkę:

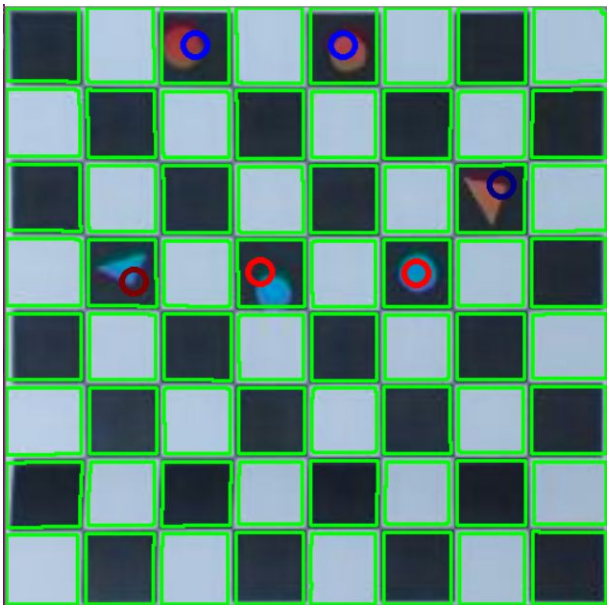
Rysunek 14. Okno gry

a. Należy ustawić pionki na czarnych polach planszy, w dowolnej pozycji początkowej. Rekomendowana pozycja: czerwone pionki na 12 górnych polach planszy, niebieskie pionki na 12 dolnych polach planszy;
b. Należy ustawić kamerę nad planszą w taki sposób, aby program wyszukał planszę. Jeżeli plansza nie jest ustawiona w poprawny sposób (najczęściej ma to związek ze słabym oświetleniem lub obcym obiektem znajdującym się nad planszą, np. ręką), po lewej stronie interfejsu graficznego widoczny jest obraz z kamery.



Rysunek 15. Widok błędnie wykrytej planszy

Jeżeli program wykrył planszę, w tym miejscu widoczny jest wycięty obraz planszy z zaznaczonymi na niej wykrytymi pionkami.



Rysunek 16. Widok poprawnie wykrytej planszy

c. Po wykryciu przez program wszystkich pionków, należy nacisnąć przycisk `Reset`, co spowoduje zapisanie początkowego stanu gry oraz zmianę ustawienia pionków na schemacie, na owy stan początkowy.

5. Rozgrywka:
   a. W oknie komunikatów (poniżej planszy) pojawia się informacja o graczu rozpoczynającym rozgrywkę (pionki niebieskie).
   b. Gracz wykonuje ruch przesuwając pionek po planszy.
   c. Kamera musi być ustawiona w taki sposób, aby umożliwić programowi wykrycie wszystkich pionków.
   d. Walidacja ruchu:
      i. Jeżeli ruch był poprawny, widok na schemacie planszy się zmienia.
      ii. Jeżeli ruch nie był poprawny, widok na schemacie planszy się nie zmienia, pojawia się komunikat o błędzie. Konieczne jest poprawne wykonanie ruchu, aby kontynuować rozgrywkę.
   e. Jeżeli pionek znajduje się w pozycji umożliwiającej mu zamianę na damkę, należy zamienić go na trójkąt o tym samym kolorze. Zamiana zostanie zasygnalizowana również na schemacie.
   f. Po wygraniu rozgrywki przez jednego z graczy, odpowiednia informacja pojawia się na ekranie.

6. Zapis stanu gry
    a. Grę można zapisać w każdym momencie. W tym celu należy nacisnąć przycisk `Save Game`.
    b. Zapis gry można nazwać. W tym celu, przed zapisaniem gry należy nacisnąć na polę `Choose name` oraz wpisać wybraną nazwę.
7. Archiwum
    a. Aby wejść w archiwum należy w polu menu nacisnąć przycisk Archive.
    b. Pojawia się okno archiwum. Po lewej stronie widoczne są wszystkie zapisane rozgrywki. Po prawej schemat planszy z zapisanym stanem, między kolejnymi stanami można się przełączać za pomocą przycisków na dole planszy.

Rysunek 17. Okno archiwum