Politechnika Poznańska

Wydział Elektryczny

Informatyka

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

Autorzy:

Mariusz Wojciechowski

Łukasz Żegalski

Piotr Kontowicz

13 czerwca 2019

Spis treści

1 Opis tematu.

Wybrany przez nasz zespół temat związany jest głównie z zagadnieniami dotyczącymi przetwarzania obrazu i analizy poprawności ruchów pionów podczas rozgrywki. Problem polega na wizualizacji obecnego stanu gry zarejestrowanego za pomocą kamery. Realizowane przez nas zadanie zakłada, że nad planszą do gry umieszczona jest kamera, która na bieżąco przesyła obraz do komputera na którym działa aplikacja. Wymagamy aby plansza na której prowadzona jest rozgrywka posiadała odpowiednie znaczniki. Na rogach planszy powinny być umieszczone markery, dzięki którym możliwa będzie identyfikacja planszy. Za ich pomocą możemy wykryć planszę w przesyłanym obrazie. Po bokach planszy znajdują się oznaczenia w kolorze zielonym. Dzięki nim możliwe jest sprawdzenie czy aktualnie nad planszą nie znajduje się żaden obiekt. Jeżeli obraz planszy jest poprawny (czyli nie ma nad nim żadnych obiektów) następuje wykrycie pozycji pionków i przekazanie ich do modułu analizy poprawności ruchu oraz modułu wizualizacji. Tworzony jest nowy obraz planszy na którym zaznaczana są pionki na podstawie wykrytych pozycji, który następnie wyświetlany jest w interfejsie użytkownika.

2 Podział prac.

Projekt można podzielić na trzy główne części: **detekcja obiektów**, **analiza ruchów** oraz **wizualizacja przetworzonego obrazu**.
Części projektu zostały podzielone pomiędzy członków zespołu w następujący sposób:

- **Łukasz Żegalski** - detekcja obiektów:
  moduł analizy obrazu odpowiedzialny za wykrywanie szachownicy, ekstrakcja pozycji pionków z planszy.

- **Piotr Kontowicz** - analiza ruchów:
  moduł walidacji ruchów, który na podstawie danych otrzymywanych z modułu analizy obrazu identyfikuje, czy wykonany ruch był poprawny,

- **Mariusz Wojciechowski** - wizualizacja przetworzonego obrazu:
  interfejs graficzny pozwalający na manipulacje parametrami funkcji detekcji oraz wyświetleniu kolejnych stanów przetwarzania obrazu, optymalizacja funkcji detekcji wraz z testami wydajnościowymi.

3 Uzasadnienie wyboru tematu.

Zdecydowaliśmy się na ten projekt ponieważ zainteresował nas jego temat. Każdy z nas jest w pewnym stopniu zainteresowany przetwarzaniem obrazu, a temat z którym związany jest wybrany przez nas projekt pozwala nam realizować i rozwijać swoje zainteresowania. Również potencjał który zobaczyliśmy w możliwości dalszego rozwoju aplikacji miał wpływ na naszą decyzję. Temat naszej pracy pokrywa pewną niszę na rynku aplikacji (nie znaleźliśmy na rynku aplikacji która spełnia te same wymagania), więc wprowadzenie jej na przykład do „Microsoft Store" może wzbudzić zainteresowanie pewnej grupy osób grających w warcaby. Dalszy rozwój aplikacji może obejmować:

- podpowiadanie najoptymalniejszego ruchu,

- możliwość streaming'u rozgrywki,

- wprowadzenie monetyzacji aplikacji,
- zakładanie kont użytkowników,
- dodanie opcji pozwalającej na grę online z innymi użytkownikami,
- organizowanie turniejów dla użytkowników,
- samouczek dla początkujących graczy - nauka o sposobach prowadzenia rozgrywki,
- analizowanie przegranych rozgrywek - wskazywanie gdzie osoba przegrana popełniła błąd prowadzący do porażki.

4 Założenia projektowe.

- zakładamy że kamera umieszczona jest bezpośrednio nad planszą do gry, tak aby operacje morfologiczne zbytnio nie deformowały przetwarzanego obrazu,
- Reguły gry:
  - dopuszczalnymi kolorami pionów w grze są piony czarne i białe,
  - każdy gracz rozpoczyna grę z dwunastoma pionami,
  - piony muszą być rozstawione na polach czarnych,
  - ruch jest wykonywany zawsze do przodu,
  - bicia mogą być wykonywane wyłącznie do przodu,
  - istnieje obowiązek bicia,

- dopuszczamy jedynie pojedyncze bicia,
- pion, który dojdzie do ostatniego rzędu planszy, staje się damką, przy czym jeśli znajdzie się tam w wyniku bicia i będzie mógł wykonać kolejne bicie (do tyłu), to będzie musiał je wykonać i nie staje się wtedy damką.
- damki mogą poruszać się w jednym ruchu o dowolną liczbę pól do przodu lub do tyłu po przekątnych, zatrzymując się na wolnych polach,
- bicie damką jest możliwe z dowolnej odległości po linii przekątnej i następuje przez przeskoczenie pionu (lub damki) przeciwnika, za którym musi znajdować się co najmniej jedno wolne pole.

5 Funkcjonalności aplikacji.

Stworzona przez nas aplikacja dostarcza następujące funkcjonalności:

- identyfikacja niepoprawnych ruchów,
- wizualizacja położenia pionków z fizycznej planszy na ekranie,
- wizualizacja kolejnych stanów obrazu podczas jego przetwarzania,
- manipulacja parametrami funkcji detekcji obrazu.

6 Zastosowane technologie informatyczne.

1. **Język programowania**

   - Python w wersji 3.6

2. **Główne Biblioteki**

   - **OpenCV2:** Biblioteka przeznaczona do pracy z obrazami w czasie rzeczywistym, posiada szereg funkcji pozwalających na wyodrębnianie konkretnych informacji z obrazu. Jest główną biblioteką wykorzystywaną w module analizy obrazu.
   - **PyQt5:** Pythonowa biblioteka umożliwiających tworzenie aplikacji okienkowych. Dzięki programowi QtDesigner możliwe jest projektowanie okien poprzez odpowiednie ustawianie elementów w edytorze graficznym, z możliwością translacji projektu graficznego prosto do kodu. Pozwala to na szybką oraz intuicyjną pracę.
   - **numpy:** Biblioteka języka Python przeznaczona do pracy na macierzach, posiadająca szereg wbudowanych funkcji ułatwiających na nich operacje. Wykorzystywana jest do tworzenia pustej tablicy do wizualizacji, na której w późniejszym kroku rysowana są pionki.

3. **Środowisko programistyczne**

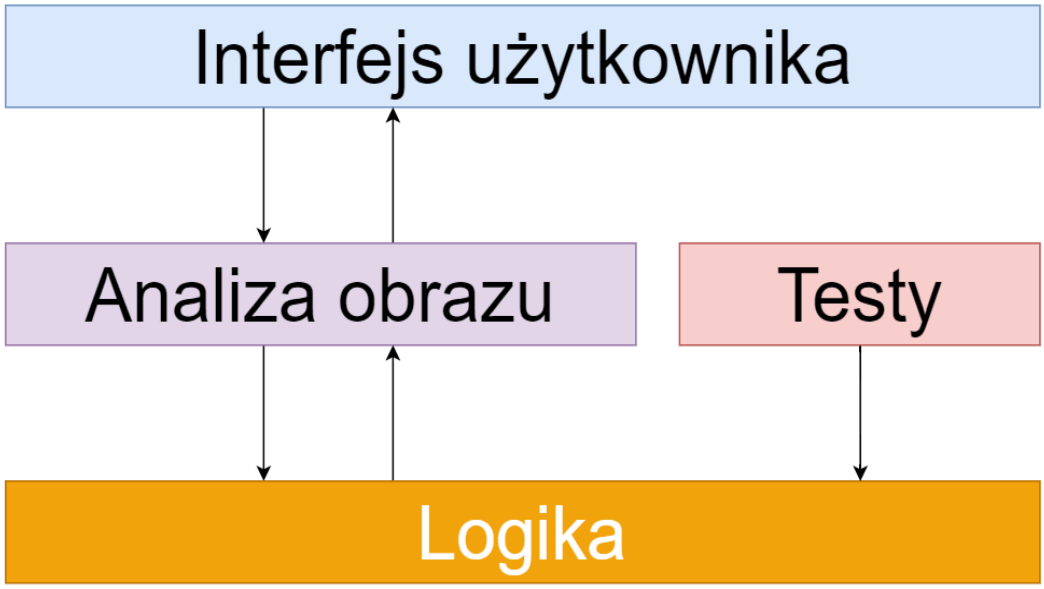
   - **JetBrains PyCharm Community Edition 2017.3.3**

4. **Pozostałe technologie**

   - **Jupiter**
   - **Git**
   - **Overleaf**

7 Architektura rozwiązania.

7.1 Ogólny model aplikacji.



Rysunek 1: Graficzne przedstawienie architektury stworzonego przez nas rozwiązania.

Architekturę naszej aplikacji oparliśmy na trzech modułach, które komunikują się ze sobą tworząc jedną wspólną całość. Podział aplikacji na niezależne części pozwolił na łatwiejsze wykrywanie potencjalnych błędów w jej działaniu oraz na na równoległą pracę.

7.2 Moduł analizy obrazu.

Moduł analizy obrazu jest odpowiedzialny za wykrywanie szachownicy, wykrywanie ciał obcych znajdujących się w obrębie analizowanego obrazu oraz detekcję pionów. Przepływ danych w module można przedstawić następująco:

- obraz z kamery zostaje przekazany do modułu,
- moduł wywołuje metodę, która poddaje obraz analizie w celu wykrycia szachownicy na podstawie umieszczonej na niej markerów w kolorze niebieskim. W przypadku wykrycia zostają zwrócone wartości *True* oraz obraz, w przeciwnym wypadku *False* oraz *None*,
- w przypadku wykrycia szachownicy przez powyższą metodę obraz zostaje przekazany do funkcji odpowiedzialnej za wykrywanie ciał obcych bądź ruchu wykonywanego w jej obrębie. Zwracane są odpowiednio *True* w przypadku braku przesłonięcia oraz *False* w przeciwnym wypadku,
- w przypadku kiedy wartość powyższej metody będzie wynosić *True* wywoływana jest metoda odpowiedzialna za wykrywanie pionków. Jeśli wartość będzie wynosić *False* główna pętla programu będzie wyświetlać ostatni zarejestrowany obraz szachownicy,

- w przypadku wywołania metody detekcji pionków wykonywane są następujące czynności:

    1. na obrazie zostaje wywołana funkcja *cv2.HoughCircles*, która wykrywa pionki oraz zwraca współrzędne ich środków,
    2. na podstawie współrzędnych pobierany jest wycinek obrazu 10px x 10px który następnie poddawany jest progowaniu binarnemu,
    3. na podstawie współrzędnych kół oraz ich sprogowanym fragmentom budowana jest tablica pozycji pionów, która zostaje przekazana do modułu analizy poprawności ruchów, oraz modułu odpowiedzialnego za wizualizację.

W module wykorzystywane są głównie funkcje wbudowane w bibliotece *OpenCV2*. Ważniejsze z nich to:

- *cv2.findContours()*: Funkcje odpowiedzialna za wykrywanie konturów na obrazie. Wykorzystywana jest w dwóch znaczących momentach:

    – po przekształceniu obrazu z kamery i wyodrębnieniu jedynie markerów w kolorze niebieskim zostają na nim wykryte kontury. Następnie dla każdego wyliczane są środki ciężkości,

    – w momencie wykrycia szachownicy i wyodrębnienia z niej markerów w kolorze zielonym zostają wykryte i zliczone kontury, co umożliwia stwierdzenie czy szachownica nie została przesłonięta.

- *cv2.HoughCircles()*: Funkcja wbudowana biblioteki *OpenCV2* wykorzystywana jest do detekcji pionków. Jako parametr wejściowy dostaje obraz szachownicy, a wynikiem jest tablica zawierająca współrzędne środków wykrytych okręgów.

- *cv2.getPerspectiveTransform()*: Funkcja odpowiedzialna za stworzenie macierzy do przekształcenia perspektywicznego, przy użyciu środków ciężkości wyliczonych za pomocą funkcji *cv2.findContours()*.

- *cv2.warpPerspective()*: Funkcja wykonująca przekształcenie perspektywiczne na obrazie z kamery przy użyciu macierzy stworzonej przy pomocy funkcji *cv2.getPerspectiveTransform()*.

7.3 Moduł walidacji ruchów.

Moduł walidacji ruchów jest zależny od części odpowiedzialnej za analizę obrazu - jest konsumentem aktualnego stanu planszy. Moduł przechowuje dwa stany planszy, obecny i poprzedni, a na podstawie różnic pomiędzy nimi określana jest poprawność ruchu. Przepływ danych w module można przedstawić następująco:

- moduł korzystający z walidacji ruchów wprowadza kolejny stan planszy,

- moduł nadrzędny wywołuje metodę odpowiedzialną za sprawdzanie ruchów. Jeśli ruch jest niepoprawny zostanie rzucony wyjątek określający jakiego rodzaju błąd został wykryty, jeśli ruch jest poprawny funkcja zwróci *True*,

- metoda odpowiedzialna za sprawdzanie ruchów korzysta z metod pomocniczych (metody wymienione w kolejności wywołań):

    1. pobierz różnicę - zostaje zwrócona różnica pomiędzy dwoma stanami tablic,
    2. jeśli wykryto różnicę zostają wywołane metody, które zwracają pozycję startową i końcową ruchu,
    3. na podstawie liczby „1” w tablicy różnic, zostaje wykryte bicie,
    4. w zależności do tego czy było bicie czy nie oraz tego czy ruch był zrobiony pionkiem czy damką, zostaje wywołana metoda sprawdzająca odpowiedni rodzaj ruchu,

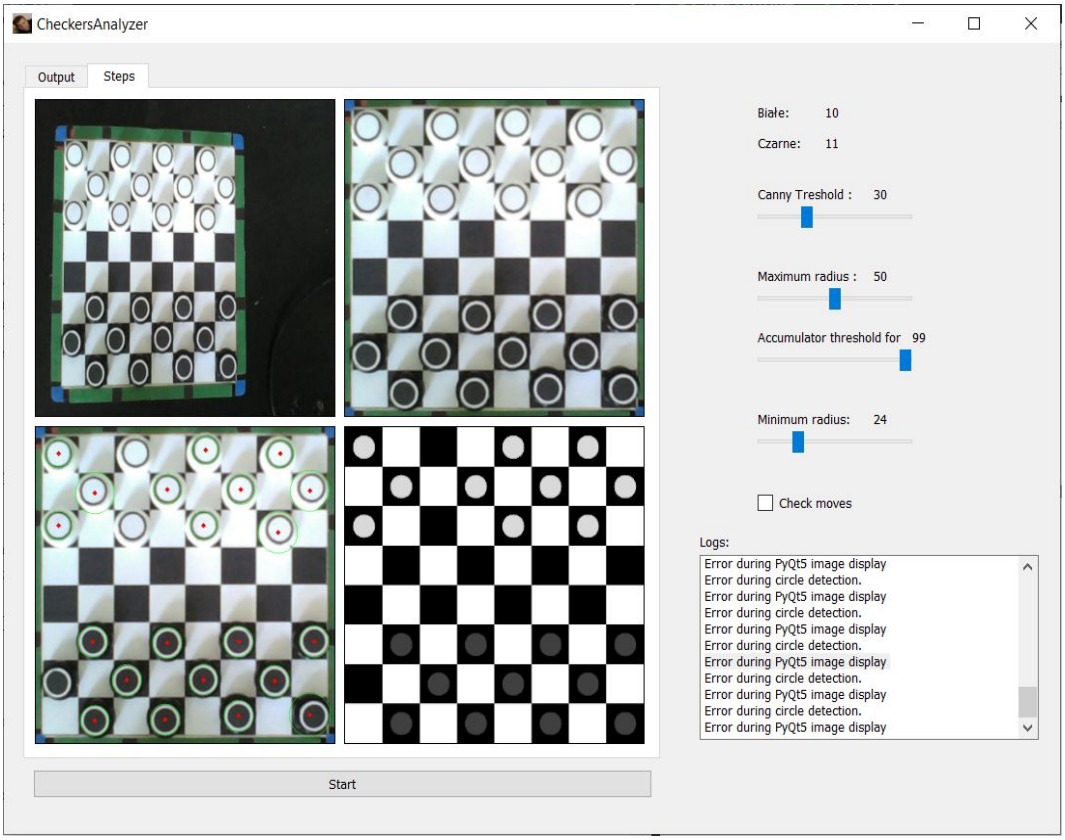
W przypadku znalezienia błędu na którymkolwiek poziomie zagłębienia zostaje rzucony wyjątek informujący o znalezionym błędzie.

7.4 Moduł wizualizacji.

Moduł wizualizacji zależny jest od typów danych jakie są zwracane przez moduł analizy obrazu. Zadaniem modułu wizualizacji jest wyświetlenie stanu gry oraz utworzenie wygodnego interfejsu graficznego pozwalającego na zmiany parametrów wykrywania pionków. W celu uniknięcia problemów ze wczytywaniem obrazu z kamery, odczytywanie klatek zostało zwolnione poprzez dodanie opóźnienia wynoszącego 10 milisekund.

7.4.1 Graficzny interfejs użytkownika.

Z racji ustaleń funkcjonalności jakie musi spełniać projekt, interfejs graficzny stanowi nieodzowną część pracy. Wykorzystana do tego celu została biblioteka, PyQt5, głównie z powodu obecności aplikacji QtDesigner, ułatwiającej projektowanie środowiska graficznego od strony wizualnej.



Rysunek 2: Interfejs graficzny aplikacji

W ramach interfejsu wchodzą:

- dwie zakładki:
  - Zakładka *Output* zawiera tylko ostateczną wizualizację (za to w wymiarze 544px x 544px),
  - Zakładka *Steps* zawiera cztery obrazy z czego każdy w innym etapie przetwarzania,
- Obrazy (4) przedstawiające kroki pośrednie pracy modułu przetwarzania obrazów, kolejno :
  - (po lewej u góry) obraz rzeczywisty z kamery internetowej,
  - (po prawej u góry) obraz wyprostowany podczas przetwarzania modułu detekcji obrazu,
  - (po lewej na dole) wyprostowany obraz, uzupełniony o narysowane krążki w punktach które zostały wytypowane jako pionki,
  - (po prawej na dole) końcowa wizualizacja planszy.
- przycisk Startu/Stopu,
- licznik białych i czarnych pionków,
- *Canny Threshold* - próg *filtru Canny'ego* używanego w funkcji wykorzystywanej przez moduł przetwarzania obrazów,
- *Maximum radius* - maksymalny promień kół do wykrycia przez moduł detekcji obrazu,

- *threshold* - akumulator progu dla funkcji *HOUGH_THRESHOLD* biblioteki *OpenCV*,
- *Minimum radius* - minimalny promień kół do wykrycia przez moduł detekcji obrazu,
- *Logs* - kontenter na informacje, takie jak wyjątki lub błędy, zwracane przez inne moduły,
- *Clear Logs* - przycisk czyszczący zawartość kontenera.

8 Napotkane problemy.

8.1 Piotr Kontowicz:

8.1.1 Referencje w tablicy.

Problem, z którym zetknąłem się podczas implementacji modułu odpowiedzialnego za analizę ruchów związany jest z tym, że w wielowymiarowej tablicy elementy w niej są referencją do poprzedniej tablicy, co skutkuje tym, że po przypisaniu tablicy do nowego obiektu, zmieniając jej zawartość zmieniamy również wartości poprzedniej tablicy. Ten problem jest dobrze zilustrowany przez poniższy kawałek kodu:

```
array = [[1, 2]]
print('Oryginalna tablica: {}.'.format(array))
new_array = array
print('Przypisanie tablicy do nowego obiektu.')
print('Nowa tablica: ', new_array)
```

```
print('Oryginalna tablica: ', array)
print('Modyfikacja new_array.')
new_array[0][0] = 10
print('Nowa tablica: ', new_array)
print('Oryginalna tablica: ', array)
```

W wyniku wykonania przedstawionego kodu wyjście wygląda następująco:

```
Oryginalna tablica: [[1, 2]].
Przypisanie tablicy do nowego obiektu.
Nowa tablica:  [[1, 2]]
Oryginalna tablica:  [[1, 2]]
Modyfikacja new_array.
Nowa tablica:  [[10, 2]]
Oryginalna tablica:  [[10, 2]]
```

Wynik wykonania programu jednocześnie pokazuje na czym polega napotkane przeze mnie ,nieintuicyjne na pierwszy rzut oka, zachowanie. Działanie modułu odpowiedzialnego za analizę ruchów opiera się na wyekstrahowaniu różnic pomiędzy pobieranymi (z modułu odpowiedzialnego za analizę obrazu) tablicami, w których zakodowane są informacje o planszy (tj. położenie pionów i ich kolory). Przez przedstawione tutaj zachowanie sam stworzyłem sobie problem, nad którego rozwiązaniem spędziłem dużo czasu - mój moduł nie wykrywał żadnych różnic pomiędzy kolejnymi przekazywanymi do niego tablicami. Po identyfikacji problemu i pewnym czasie spędzonym na poszukiwaniu rozwiązania znalazłem wyjście, przedstawione poniżej:

```
import copy
array = [[1, 2]]
print('Oryginalna tablica: {}.'.format(array))
new_array = copy.deepcopy(array)
print('Przypisanie tablicy do nowego obiektu.')
print('Nowa tablica: ', new_array)
print('Oryginalna tablica: ', array)
print('Modyfikacja new_array.')
new_array[0][0] = 10
print('Nowa tablica: ', new_array)
print('Oryginalna tablica: ', array)
```

Wyjście po wykonaniu przedstawionego kodu jest zgodne z oczekiwaniem:

```
Oryginalna tablica: [[1, 2]].
Przypisanie tablicy do nowego obiektu.
Nowa tablica:  [[1, 2]]
Oryginalna tablica:  [[1, 2]]
Modyfikacja new_array.
Nowa tablica:  [[10, 2]]
Oryginalna tablica:  [[1, 2]]
```

Po zastosowaniu *copy.deepcopy()* do kopiowania tablicy zamiast intuicyjnego operatora „=” mój problem został rozwiązany, dzięki czemu funkcja pobierająca różnice pomiędzy dwoma tablicami zaczęła zwracać wyniki zgodne z oczekiwaniami.

8.1.2 Błędna detekcja pionów przez odpowiedzialny za to moduł.

Gdy zaczynałem pracę nad swoim modułem błędnie założyłem, że otrzymywane dane na temat położenia pionów na planszy będą poprawne. Było to błędne założenie, które kosztowało mnie kilka godzin. Skutecznie utrudniło mi to pracę z rozwojem modułu, za który byłem odpowiedzialny, ponieważ otrzymywałem błędne wyniki z funkcji analizujących ruchu. Po upewnieniu się, że jednak po mojej stronie wszystko działa prawidłowo - zrobiłem to za pomocą dodatkowych funkcji zrzucających potrzebne informacje do plików, a następnie analizie ich zawartości- doszedłem do wniosku, że otrzymuję błędnie przetworzone dane, Odkrycie powodu mnie nie zadowoliło i postanowiłem poszukać źródła błędu. Zacząłem więc porównywać to co moduł wykrywa z tym co widzę na ekranie. To rozwiązanie nie pomogło znaleźć przyczyny błędu - na pierwszy rzut oka moduł działał poprawnie. Dedukując dalej postawiłem sprawdzić czy nagranie z rozgrywki, z której korzystamy jest poprawne - obejrzałem je poklatkowo i zobaczyłem, że stany planszy na niektórych klatkach są nieprawidłowe (korzystaliśmy z obrazu przechwyconego z ekranu gry w warcaby). Problemem okazały się małe opóźnienia w renderowaniu pionów na planszy np. podczas ruchu pion znikał z pola startowego ale na właściwym polu pojawiał się dwie klatki dalej. Podczas oglądania filmu w normalnym tempie nie było to widoczne. Znając już przyczynę błędów postanowiłem podejść do tematu testowania swojego modułu inaczej - postanowiłem stworzyć testy jednostkowe. Zastanawiałem się chwilę jak to sprytnie zrobić. Doszedłem do wniosku, że najrozsądniej będzie pobrać jakąś prostą implementację gry w warcaby zmodyfikować jej kod ściągając zabezpieczenia przed niepoprawnymi ruchami i dodać funkcję, która będzie zapisywała

odpowiednio zakodowane stany planszy do pliku **.txt*, zdecydowałem się na pliki tekstowe aby móc łatwo oglądać ich zawartość i ją modyfikować. Po wykonaniu 14 przypadków testowych dla planszy, w której pierwsze lewe górne pole było czarne, zastanowiłem się jak sprytnie przygotować zestaw testowy dla przypadku gdy pierwsze pole będzie miało kolor biały. Oglądając planszę do gry zauważyłem, że wystarczy wykonać jej odbicie aby pierwsze pole było białe. Napisałem więc skrypt, który to zrobił. W przedstawiony sposób powstał zestaw danych testowych, z którego skorzystałem podczas rozwoju aplikacji. Pozwoliło mi to uniezależnić się od poprawności otrzymywanych danych - jeśli wyniki wywoływanych funkcji nie były zgodne z oczekiwanymi byłem pewny, że błąd jest w kodzie a nie w konsumowanych danych. Ten problem również dał mi do zrozumienia, że jednak lepiej stworzyć osobną klasę odpowiedzialną tylko za walidację ruchów, niż dopisywać funkcje testujące ruchy w module wyszukiwania pionków, tak jak zamierzałem na początku. Analizując ten błąd doszedłem do dwóch wniosków:

1. nie ufaj na słowo - nawet jeśli wydaje się, że coś działa błąd może być ukryty jeszcze głębiej (niekoniecznie w działaniu programu),

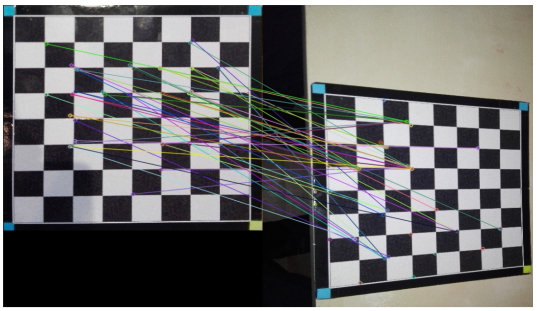
2. warto pamiętać o zasadach SOLID, gdyż znacznie ułatwiają pracę z rozwojem kodu. Złamałem tę zasadę chcąc stworzyć klasę, która jest odpowiedzialna za analizę obrazu i sprawdzanie poprawności ruchów.

Stworzenie osobnej klasy odpowiedzialnej tylko za walidację ruchów znacznie poprawiło czytelność kodu i ułatwiło mi nad nim pracę.

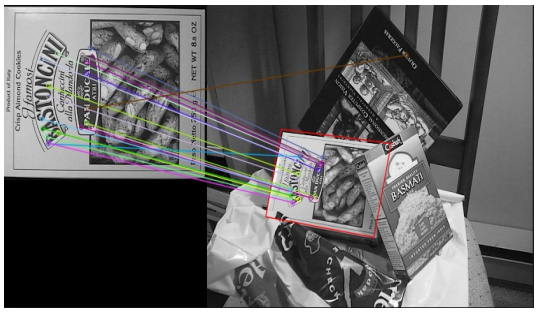
8.2 Łukasz Żegalski:

8.2.1 Wykrywanie szachownicy.

Pierwszym pomysłem na wykrywanie szachownicy było wykorzystanie detekcji cech pomiędzy wczytanym obrazem szachownicy, a obrazem z kamery. Pomysł okazał się ciężki do realizacji. Głównym problemem był sam wygląd szachownicy, równomierne i identyczne pola były niepoprawnie wykrywane pomiędzy dwoma obrazami. Pomysł, który sprawdza się w przypadku mniej monotonicznych obrazów tutaj okazał się nieskuteczny. Widać to na przykładzie *Rysunku 2.* oraz *Rysunku 3.* na pierwszym z nich widać, że cechy wykryte na obydwu obrazach nie odpowiadają sobie wzajemnie, zaś na drugim obrazie cechy pokrywają się praktycznie w 100% co po odpowiednich działaniach pozwala wyodrębnić dany fragment z obrazu.



Rysunek 3: Detekcja cech szachownicy



Rysunek 4: Detekcja cech książki

Fragment kodu odpowiedzialny za powyższe zadanie:

```
def Detect(self):
 img_hsv2=cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
 mask=cv2.inRange(img_hsv2, lower_blue, upper_blue)
 morphology=cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
 morphology=cv2.morphologyEx(morphology, cv2.MORPH_DILATE,kernel)
 contours=cv2.findContours(morphology,cv2.RETR_TREE,
 cv2.CHAIN_APPROX_SIMPLE)[1]
 tab=[]
 if len(contours)==4:
  for x in range(0,4):
  cnt=contours[x]
  M=cv2.moments(cnt)
  cx=int(M['m10']/M['m00'])
  cy=int(M['m01']/M['m00'])
  tab.append([cx,cy])
  if(tab[0][0]<tab[1][0]):
   tab[0], tab[1] = tab[1], tab[0]
  if(tab[2][0]<tab[3][0]):
   tab[2], tab[3] = tab[3], tab[2]
  pts2=np.float32([[0, 0], [450, 0], [0, 450], [450,450]])
  pts1=np.float32([[tab[3][0],tab[3][1]],[tab[2][0],
  tab[2][1]],[tab[1][0],tab[1][1]],[tab[0][0],tab[0][1]]])
  M=cv2.getPerspectiveTransform(pts1, pts2)
  result=cv2.warpPerspective(image, M, (450, 450))
  return True, result else: return False, None
```

8.2.2 Detekcja ruchu.

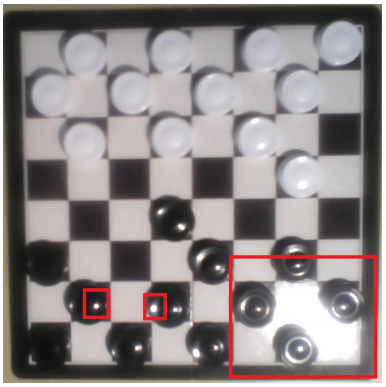
Problem z detekcją ruchu polegał na braku możliwości wykorzystania wbudowanych funkcji biblioteki OpenCV2. Spowodowane było to długim wygaszeniem detekcji, co powodowało, że w momencie zakończenia ruchu przez jednego gracza i rozpoczęcia przez drugiego obraz nie był przekazywany do modułu detekcji pionków. Problem został rozwiązany poprzez wprowadzenie zielonych markerów na brzegach planszy. Początkowo były koloru czerwonego, jednak podczas testów okazało się, iż zakres koloru używany do ich wykrywania obejmuje również kolor skóry. Powodowało to detekcję niewłaściwej liczby konturów oraz znajdowania 12 pozycji pomimo zakrycia niektórych z nich, co wiązało się z przekazaniem planszy wraz z dłonią do funkcji wykrywającej pionki. Poprzez ekstrakcję koloru oraz progowanie dostajemy obraz, na którym zostają zliczone kontury. W przypadku wykrycia 12 obraz zostaje poddany analizie, jednak jeśli którykolwiek z nich zostanie przesłonięty zostaje wykryta inna liczba konturów, która zapobiega przekazaniu obrazu do analizy, co jest równoznaczne z tym, iż coś znajduje się na obrazie. Fragment kodu odpowiedzialny za powyższe zadanie:

```
def Detect(self):
 img_hsv2=cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
 mask=cv2.inRange(img_hsv2, low_green, high_green)
 morphology=cv2.dilate(mask, kernel, iterations=3)
 morphology=cv2.dilate(morphology, kernel)
 contours=cv2.findContours(morphology, cv2.RETR_TREE,
 cv2.CHAIN_APPROX_SIMPLE)[1]
 if len(contours)==12:
```

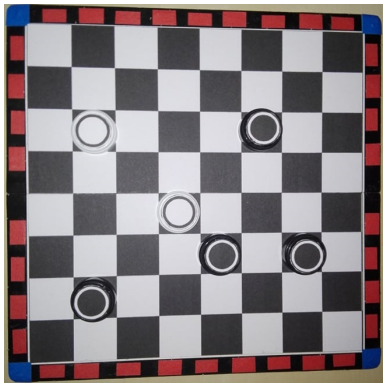
```
    return True
else:
    return False
```

8.2.3 Detekcja pionów.

Ostatnim problemem kluczowym do działania całego programu okazała się detekcja pionków. Przeprowadzane testy pozwoliły ustalić parametry jakie należy ustawić, aby przebiegała ona pomyślnie, jednak z powodu posiadania laminowanej wersji planszy oraz plastikowych pionów powstawały problemy w postaci refleksów oraz odbijających się punktowych źródeł światła, które w niektórych momentach były błędnie rozpoznawane jako okręgi, a co za tym idzie również jako piony co można zauważyć na *Rysunku 4*. Problem został rozwiązany poprzez zastąpienie planszy jej matową wersją a same piony zostały pokryte odpowiadającymi ich kolorowi żetonami *Rysunek 5*.



Rysunek 5: Plansza z refleksami



Rysunek 6: Zmodyfikowana plansza

8.3 Mariusz Wojciechowski.

8.3.1 Wizualizacja klatek

Jedną z głównych przeszkód, stojących na przeciw wybranej bibliotece do implementacji graficznego interfejsu użytkownika, była statyczność wyświetlania obrazu w interfejsie PyQt5. Innymi słowy trudność zaimplementowania ciągłego odświeżania obrazów w celu uzyskania efektu wyświetlania filmu.

Problem został rozwiązany za pomocą implementacji dodatkowej metody wewnątrz utworzonej klasy, oraz wykorzystanie predefiniowanej klasy *QTimer* posiadającego status *timeout*. Moduł wizualizacji składa się z głównej klasy,

```
class MainWindow(QWidget):
    # class constructor
    def __init__(self):
        //... rest of the implementation
        # create a timer
        self.timer = QTimer()
        self.timer.timeout.connect(self.viewCam)
        self.ui.control_bt.clicked.connect(self.controlTimer)
```

w której za pomocą klasy *QTimer* oraz jej atrybutu *timeout* można zapętlić program, powodując ciągłe wywoływanie podłączonej do timera autorskiej metody *viewCam()*, gdzie następuje odczytanie kolejnej kratki z kamery.

```
def viewCam(self):
    # read image in BGR format
    ret, image = self.cap.read()
    //... rest of implementataion
```

dodatkowo *Button Start* został połączony z zaimplementowaną funkcją *controlTimer*, aby umożliwić przerwanie, oraz wznowienie odczytywania klatek z kamery. W funkcji tej, ustawiany jest też czas oczekiwania *timera* w milisekundach, po którym zostanie ponowne wywołanie metody *viewCam*

```
def controlTimer(self):
    if not self.timer.isActive():
        self.cap = cv2.VideoCapture(0)
        # start timer
        self.timer.start(100)
        # update control_bt text
        self.ui.control_bt.setText("Stop")
    # if timer is started
    else:
        # stop timer
        self.timer.stop()
        # release video capture
        self.cap.release()
        # update control_bt text
        self.ui.control_bt.setText("Start")
```

8.3.2 Optymalizacja funkcji detekcji obrazu

Jednym z głównych problemów podczas rozpatrywania optymalizacji programu była optymalność funkcji detekcji obrazu. Używana biblioteka (*OpenCV*) posiadała jedną, główną funckję detekcji, jaką była funkcja *cv2.HoughCircles*. Pierwszą przeszkodą było znalezienie innych metod detekcji okręgów i wykorzystanie ich w celach badawczych. W rzeczywistości HoughCircles było jedyną predefiniowaną metodą. Każdy inny potencjalnie konkurencyjny algorytm (*RANSAC, Randomized Circle detection*) musiał być implementowany ręcznie, co powodowało jego nieoptymalność w środowisku pythonowskim. Ostatecznie, to *HoughCircles* okazało się być najszybszym z wymienionej trójki, i postanowiono przeprowadzić optymalizacje parametrów. Optymalizowane parametry funckji były następujące:

- minDist - minimalna odległość między dwoma środkami wykrytych kół,
- param1 - próg filtru Canny'ego,
- param2 - akumulator progu dla Funkcji *HOUGH THRESHOLD* biblioteki *OpenCV*
- maxRadius - maksymalny promień kół do wykrycia przez moduł detekcji obrazu.
- minRadius - minimalny promień kół do wykrycia przez moduł detekcji obrazu.

Przy ustalonych przez projektantów szerokości i wysokości przetwarzanej planszy jako 544 px x 544 px, oznacza to, iż jeden kwadrat ma ok 68 pikseli.

implikuje to fakt, iż najkrótszy dystans pomiędzy dwoma środkami pionków będzie wynosił

$\sqrt{2} * 68 \approx 96$.

Dodatkowo, zakładając, że pionek jest zawsze dopasowany do pól planszy, można oszacować możliwy zakres promienia znajdywanych kółek, ograniczając w ten sposób liczbę obliczeń niezbędnych do wykonania, a także poprawiając detekcję.

68 px / 2 = 34 px

Można zatem założyć, że wykryty pionek będzie miał promień mniejszy, bądź oscylujący w otoczeniu 34 px.

Wszystkie pionki są jednolite we własnym kolorze (a przynajmniej środek pionka jest jednolity) białym lub czarnym, co pozwala na określenie minimalnej wartości promienia wyszukiwanych kółek.

Promień nie powinien być mniejszy niż połowa promienia koła będącego na całej szerokości pola na którym stoi, tzn.:

1/2 * 1/2 * 68 px = 12 px < minRadius

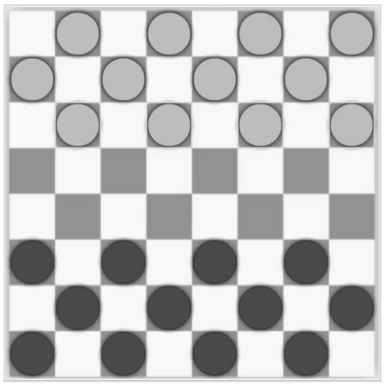
Ustalony przedział dla promienia wykrywanego koła wynosi więc

12px < radius < 34px

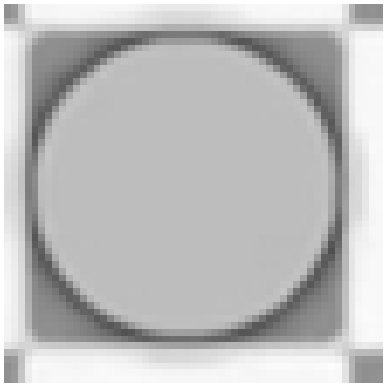
Pomimo obliczenia optymalnych wartości dla niektórych parametrów, nie zmienia to faktu, że projekt zakłada możliwość edytowania niektórych parametrów omawianej funkcji. Obliczone wartości są więc ustawiane jako wartości początkowe i nadal możne je edytować.

Kolejnym sposobem optymalizacji funkcji jest sposób jej wywoływania.

W celu porównania podejść podjęto kroki badawcze ustalając dla nich optymalne parametry, oraz wykonując je w pętli i mierząc czas. Dodatkowo



Rysunek 7: Plansza z refleksami



Rysunek 8: Fragment *Rysunku 7*

w przypadku pojedynczego pola, *HoughCIrcles* obliczano tylko dla czarnych pól (z racji, iż pionki mogą być tylko na czarnych polach), co skróciło czas obliczeń o połowę.

Obliczenie HoughCircles dla tysiąca iteracji w przypadku Całego obrazu (544x544) wyniosło 2.75 sekund, podczas gdy identyczna liczba iteracji w przypadku wykonywania go 32 razy na wyciunku obrazu zajęło 4.93 sekund.

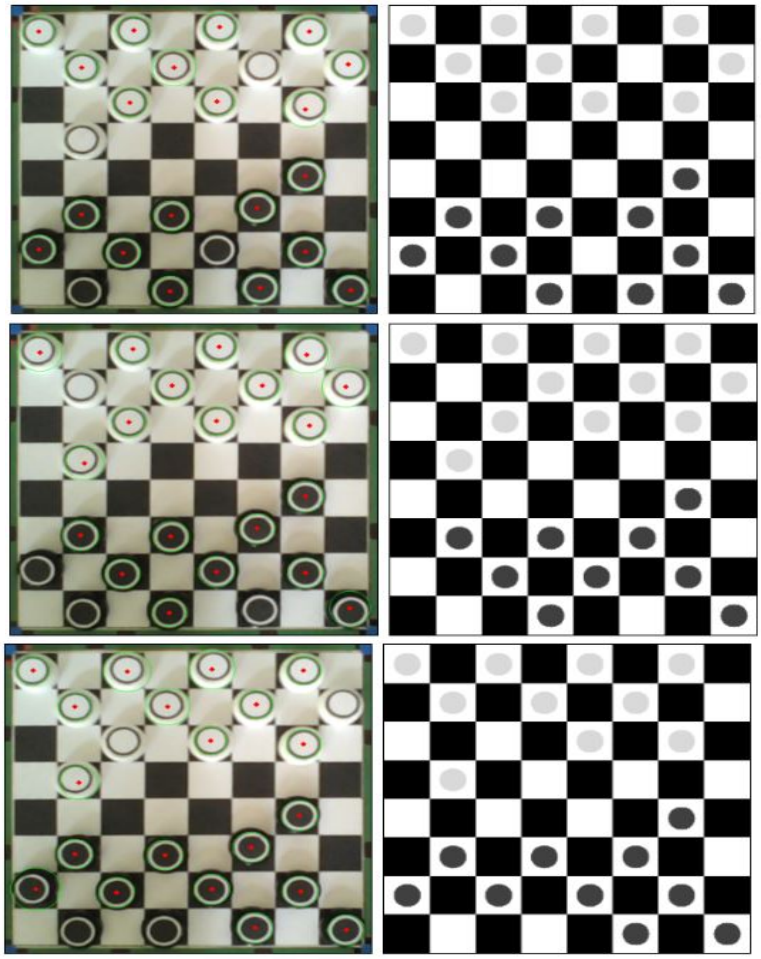
Badanie wykazało, że szybszym sposobem, jest wykonanie *HoughCircles* na całym obrazie, zakładając, iż parametry funkcji będą optymalne

8.3.3 Problem "drgającego" obrazu

Jednym z pierwszych problemów obecnych w module przetwarzania obrazu jest znalezienie planszy do gry na obrazie z kamery, a następnie transpozycja znalezionego obszaru do wymiarów wypełniających okno. Odbywa się to za pomocą niebieskich znaczników umieszczonych na rogach planszy, przez co maszyna może rozpoznać obiekt.

Problemem który występuje jest nieznaczne przekrzywienie obrazu wyni-

kające z detekcji niebieskich obszarów na planszy. Efekt ten powoduje lekko inny (przesunięty o kilka pikseli) wynik transponowania zaznaczonego obiektu do pełnych wymiarów w każdej kolejnej klatce. Wynikiem tego jest ciągła (co wczytaną klatkę) zmiana wyników detekcji kółek, a co za tym idzie "miganie" pionków.

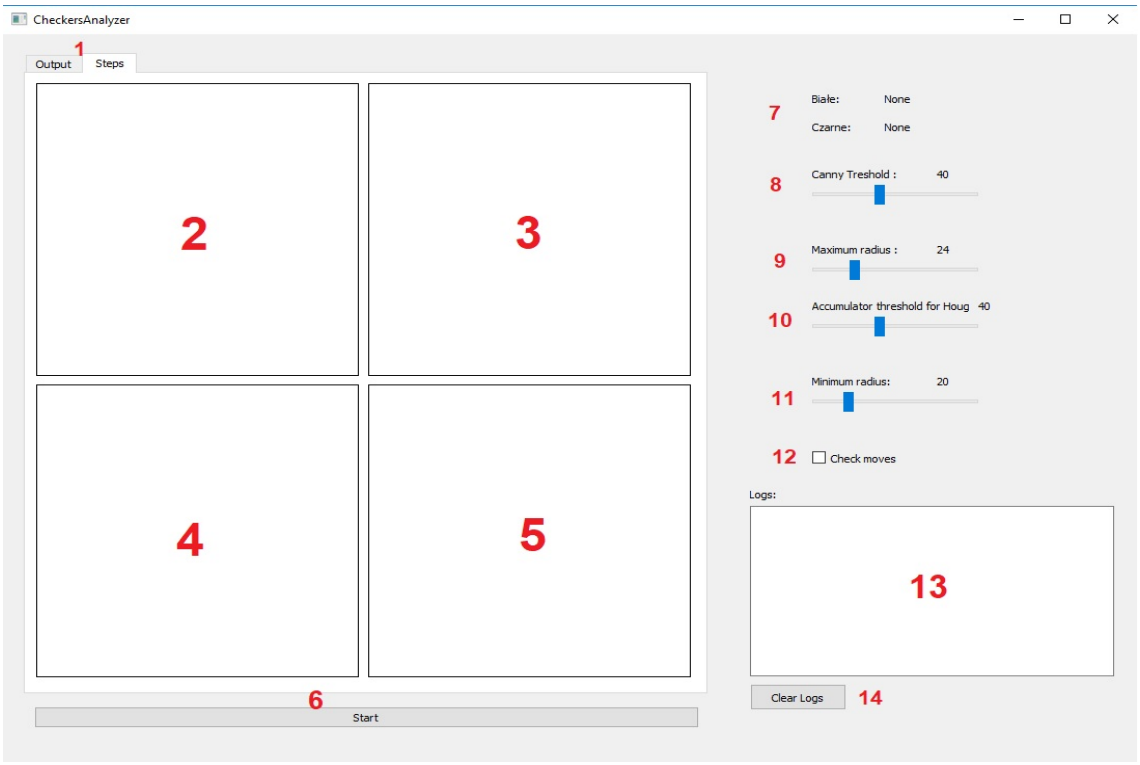


Rysunek 9: Kolejne przetworzone klatki z kamery. Różnica pomiędzy detekcją kół.

Metodą zapobiegawczą tak częstemu miganiu, może być zmniejszenie frekwencji odczytu klatek z kamery poprzez zwiększenie czasu oczekiwania *QTimera* odpowiedzialnego za odtwarzanie programu w pętli. Zmniejszyło to jednak tylko częstotliwość zmian.

Ostatecznym rozwiązaniem okazało się zmienienie wartości parametru funkcji detekcji *Accumulator treshold*. Odpowiada on za wielkość akumulatora będącego wielkością przeszykiwanych obszarów. Wartość parametru była wybierana w sposób heurystyczny w celu osiągnięcia jak najlepszej j detekcji. Ostatecznie próby dowiodły, że najlepszym przedziałem jest 30 do 60.

9 Instrukcja użycia



Rysunek 10: Graficzny interfejs użytkownika

- Włączenie aplikacji.
- Wciśnięcie przycisku **Start** (6) powoduje rozpoczęcie przechwytywania obrazu z kamery. Jednocześnie przycisk zmienia sie na przycisk **Stop**, przerywający odczyt.
- Zakładki **Output** oraz **Steps** (1) Określają formę wyświetlania obrazu:
  - **Steps** przedstawia cztery mniejsze obrazy, kolejno:

        * obraz przechwytywany przez kamerę (**2**)
        * transponowana plansza do gry wykryta na obrazie kamery(**3**)
        * obraz planszy uzupełniony o narysowane krążki w punktach które zostały wytypowane jako pionki (**4**)
        * Wizualizacja stanu planszy na osobno utworzonym obrazie(**5**)
    - **Output** przedstawia wizualizację stanu planszy w rozmiarze 544 px x 544 px

- (**7**) przedstawia liczbe wychwyconych pionków, odpowiednio Białych i czarnych.

- W celu manipulowania przetwarzaniem obrazu służą suwaki (**8-11**).

- Zaznaczenie analizowania ruchów **Check moves** (**12**). Gdy zaznaczony ewentualne błędy ruchów wypisywane są w oknie.

- Przycisk **Clear Logs** (**14**) usuwa zawartość okna **Logs** (**13**)