CheckersChecker

Dokumentacja projektowa

Autorzy:

Piotr Biskup

Katarzyna Dykiert

Michał Maciaszek

1. Opis tematu wraz z uzasadnieniem wyboru.

Celem naszego projektu jest stworzenie programu, który będzie służył do analizowania ruchów wykonywanych przez pionki na planszy warcab. Program wyświetla na interfejsie aktywny stan planszy, wraz z odpowiednim komunikatem oraz obecnie grającą liczbą pionków danego koloru.

Program funkcjonuje na zasadach amerykańskich, co oznacza między innymi, że bicia są obowiązkowe oraz możliwe są bicia wielokrotne.

Program korzysta z obrazu przekazywanego z kamery ustawionej nad specjalnie przygotowaną oraz oświetloną warcabnicą (planszą do gry), a następnie wykonuje na niej różnego rodzaju przekształcenia mające za zadanie ustalenie aktualnych pozycji pionków na planszy.

Po wykonaniu ruchu program go analizuje i zwraca użytkownikowi odpowiednią informację zwrotną na temat jego poprawności.

Program został napisany przy zastosowaniu języka Python w środowisku PyCharm Community.

W naszym projekcie dużą rolę odgrywa przetwarzanie obrazu. Mając wcześniejszą styczność z tym tematem na innych zajęciach laboratoryjnych byliśmy już z nim zaznajomieni. Był to jeden z głównych powodów, dlaczego ten właśnie temat został przez nas wybrany. Już z początku mieliśmy wstępnie zarysowany pomysł na wykonanie tego zadania, co tylko ułatwiło wybór.

Język programowania został dobrany na podstawie wcześniejszych z nim doświadczeń przy przetwarzaniu obrazu oraz chęci rozszerzenia wiedzy odnośnie tworzenia interfejsu za jego pomocą.

Jeden z ostatnich powodów naszego wyboru jest zainteresowanie tematem oraz przeszłość związana z warcabami, jednego z współtwórców projektu.

Zasady:

Gra w warcaby kontrolowana przez program oparta jest na zasadach gry w warcaby angielskie. Ta wersja została wybrana ze względu na mniejszą złożoność zasad i prostotę niż w warcabach klasycznych (międzynarodowych), lecz także z powodu lepszej dostępności do zdobycia planszy i pionków.
W trakcie tworzenia projektu, część zasad uległa zmianie (także zwiększeniu ich złożoności), co zostanie omówione w dalszej części.
Zasady gry w warcaby angielskie:

- wymiary planszy wynoszą 8x8 (64 pola),
- pole w lewym dolnym rogu jest czarne,
- każdy gracz otrzymuje 12 pionków,
- zaczyna gracz grający białymi,
- gracze wykonują swoje ruchy na zmianę,
- pionki mogą poruszać o jedno pole po skosie do przodu, jeżeli nie jest ono zajęte,
- jeżeli pole, na które chce poruszyć się pionek jest zajęte przez pionek przeciwnika, a następujące za nim na tej samej przekątnej jest wolne, gracz może zbić pionek przeciwnika stawiając swój na wolnym polu i zdejmując pionek przeciwnika,
- jeżeli po zbiciu przeciwnego piona, możliwe jest kolejne bicie, gracz musi je wykonać,
- gracz może wybrać dowolną drogę zbicia, jeżeli ma do wyboru więcej niż jedną,
- pion dochodząc do ostatniego pola staje się królem (damką),
- zmiana pionka na króla kończy ruch gracza (w przypadku bicia, niemożliwe jest dalsze wielokrotne bicie),

- król ma możliwość poruszania się do przodu i do tyłu, to samo dotyczy możliwości bicia,
- gra kończy się po zbiciu wszystkich bierek przeciwnika bądź uniemożliwienia wykonania przez nie ruchu.

Jedna zasada została zmieniona i uległa modyfikacji podwyższając tym samym trudność gry, a także zwiększając poziom skomplikowania algorytmu. Tą zasadą jest możliwość dowolnego wyboru, jaką liczbę pionów chcemy bić.

W warcabach klasycznych mamy obowiązek bicia jak największej ilości pionów przeciwnika. W angielskiej wersji tej gry możemy wybrać - czy chcemy się poruszyć, bić czy dokonać bicia wielokrotne. Jedyną reguła jest to, gdy po biciu możemy je dalej kontynuować, to musimy to zrobić.

W tworzonej przez nas wersji warcab używamy wersji pośredniej, zachowując jednocześnie prostotę i rozwijając poziom trudności warcab angielskich. Jeżeli możliwe jest bicie gracz musi je wykonać. Jeśli można dokonać wielokrotne bicia, także jest ono wymagane. Nie wymagane jest jednak zastosowanie największego wielokrotnego bicia, czyli zbicia największej możliwej liczby pionów przeciwnika, czyli ruchu, który mógłby nie zostać zauważony przez początkujących, niedoświadczonych graczy.

2. Podział prac pomiędzy członków zespołu.

Nasz zespół składa się z trzech członków, a prace projektowe zostały podzielone na trzy główne kategorie. Są to:

- przetwarzanie obrazu,
- interfejs,
- logika.

Przetwarzaniem obrazu zajął się Piotr Biskup. Wykorzystał do tego bibliotekę `OpenCV` celu analizy obrazu oraz bibliotekę `Numpy`, która służy do przeprowadzania zaawansowanych operacji matematycznych na macierzach oraz wielowymiarowych tablicach. Zaimplementował on wykrywanie planszy, która zostaje przekształcona perspektywicznie, aby pozbyć się jej, nieinteresującego nas, otoczenia oraz aby

uzyskać widok z góry. Następnie szukane są okręgi oraz sprawdzanych jest ich kolor, żeby stworzyć tablicę aktualnego ułożenia pionków. Układ ten przekazywany jest do interfejsu, gdzie zostaje rozrysowany na planszy.

Pracę nad Interfejsem przejęła Katarzyna Dykiert, która do tego zadania wykorzystała głównie bibliotekę `Pygame`. Do jej zadań należało m.in. utworzenie tablicy reprezentującej plansze wraz z rozłożonymi na niej pionkami, która następnie zostaje przedstawiona na ekranie. Zajmuje się ona także komunikacją z "logiką" (Michał Maciaszek) w sprawach poprawności ruchów oraz przekazywaniem nadanych przez niego komunikatów na interfejs. Katarzyna kontroluje aktualną liczbę aktywnych pionków z każdego koloru.

Nad logiką pracę podjął Michał Maciaszek, zaimplementował do tego własne algorytmy sprawdzania poprawności ruchu. Zajął się generowaniem wszystkich możliwych następujących ruchów w danym momencie, czyli sprawdzaniem ich legalności. Ponadto porównuje obecną planszę z wszystkimi wygenerowanymi i zależnie od wyniku przekazuje odpowiednie komunikaty.

3. Funkcjonalności oferowane przez aplikację.

CheckersChecker zawiera następujące funkcjonalności.

1. Przechwytywanie obrazu przy użyciu kamerki internetowej oraz ustalaniu obecnego stanu warcabnicy.

Użytkownik widzi trzy okna: okno aplikacji, okno z oryginalnym obrazem z kamery oraz okno z przekształconym obrazem z kamery. Na tym ostatnim, każda klatka przekształcana jest tak, że widzimy samą planszę z lotu ptaka. Zaznaczone są też znalezione pionki. Pozwala nam to na monitorowanie poprawności ich wykrycia. Wykryty stan warcabnicy przekazywany jest do interfejsu. Układ pionków nie jest analizowany w momencie kiedy któryś z graczy trzyma dłoń nad planszą w celu wykonania ruchu.

2. Prezentowanie ostatniego poprawnego stanu planszy na interfejsie.

Na ekranie widoczny jest aktualny stan warcabnicy przechwycony z kamerki. W przypadku gdy ostatni ruch został zakwalifikowany jako niepoprawny obraz ten się nie zmienia i przedstawia ostatni poprawny stan. W następnej kolejności program

będzie czekać aż gracz powróci po przedstawionego na ekranie stanu gry. Po tej czynności można kontynuować rozgrywkę, a na ekranie stan warcabnicy będzie się ponownie aktualizował.

3. Analizowanie wykonanych ruchów w grze odnośnie ich poprawności.

Po odebraniu dwóch tablic zawierających stan aktualnie wyświetlany oraz obecnie przechwycony program sprawdza czy wykonany ruch jest poprawny. Zostaje to osiągnięte poprzez przystawienie obecnej planszy ze wszystkimi wygenerowanymi tablicami poprawnych ruchów.

4. Przedstawianie aktualnej ilości pionków w grze danego koloru.

Z aktualnie wyświetlanej na ekranie planszy zliczane zostają pionki białe oraz pionki czarne, a następnie ich ilość zostaje przekazana do użytkownika. Gdy ilość pionków danego koloru wyniesie 0, wygrywa drugi gracz.

5. Zwracanie odpowiednich komunikatów do użytkownika.

Podczas analizy wykonanych ruchów rozpatrujemy ich poprawność oraz to czy został wykonany ruch najlepszy. Odpowiednio do zarejestrowanych zachowań przekazany zostaje odpowiedni komunikat, który następnie zostaje zwrócony użytkownikowi.

Przy wykonaniu ruchu całkowicie nielegalnego, jak również przy nakazie powrotu do poprzedniego stanu planszy komunikat ma czerwony kolor.

4. Wybrane technologie z uzasadnieniem.

Cała aplikacja CheckersChecker napisana jest w języku Python. Skrypty pisane w tym języku mają przejrzystą i czytelną strukturę. Ułatwia to analizę kodu, ponieważ program jest wyraźnie podzielony na sekcje. Składania Pythona jest intuicyjna oraz nieporównywalnie krótsza niż w innych językach porgramowania, co sprawia, że kod pisze się szybciej.

Mimo, iż są języki wydajniejsze od Python’a, na nasze potrzeby jest całkowicie wystarczający. W napisanych przez nas skryptach obliczenia wykonywane są bez zauważalnych opóźnień, a cała aplikacja działa bardzo płynnie.

Znacząca dla nas były również bogata dokumentacja oraz ogromna społeczność tego języka. Często w przypadku błędów oraz problemów, z którymi

sami nie mogliśmy sobie poradzić, przychodziły nam z pomocą i rozwiewały wszelkie wątpliwości.

Najważniejszym jednak powodem wybrania Python'a jest dostępność wielu dodatkowych bibliotek, w tym wykorzystywanych przez nas `OpenCV`, `Numpy` i `Pygame`. To właśnie na nich opiera się główna funkcjonalność CheckersChecker.

- **OpenCV**

  Open source computer vision to biblioteka wykorzystywana do przetwarzania obrazu w czasie rzeczywistym. Zawiera ona zaimplementowanych ponad 2500 algorytmów, z czego wiele z nich opartych jest na bardzo zaawansowanej matematyce oraz teorii obrazu. Wykorzystujemy więc gotowe już funkcję i nie musimy rozumieć dokładnego ich działania.

  Dostępna jest również bardzo bogata dokumentacja. Zawiera ona dokładny opis funkcji, parametry, które przyjmuje wraz z ich wyjaśnieniem oraz zwracane wartości. Dodatkowo większość funkcjonalności posiada przykłady użycia. Znacząco ułatwia to zrozumienie zagadnienia oraz zaprogramowanie. Szeroka, internetowa społeczność użytkowników wzajemnie pomaga sobie rozwiązywać problemy związane z tematyką przetwarzania obrazów. Można więc znaleźć wiele przykładowych implementacji ułatwiających korzystanie z tej biblioteki.

  OpenCV jest projektem open sourcowym. Korzystanie z niego jest całkowcie darmowe, nawet dla zastosowań komercyjnych.

  Bibliotekę tą w języku Python wykorzystywaliśmy na zajęciach Przetwarzanie Obrazów i Systemy Wizyjne. Zdobyte wcześniej umiejętności i wiedza miały duży wpływ na nasz wybór. Analiza obrazu z kamery jest kluczowa dla naszego projektu, dlatego ważne jest że pracujemy w znanym nam już środowisku.

- **Numpy**

  Biblioteka wspierająca zaawansowane, matematyczne operacje na wielowymiarowych tablicach oraz macierzach. Jest open source'owa, więc

możemy z niej korzystać za darmo. Korzystamy z niej, ponieważ jest wymagana przez niektóre funkcje OpenCV.

- **PyGame**

  Podczas pracy nad interfejsem została wykorzystana biblioteka `Pygame`. Jest ona odpowiednia dla mniej skomplikowanych oraz wymagających programów. Biblioteka jest łatwa w instalacji i prosta w obsłudze. Idealna dla osób początkujących pracę z intefejsem w Pythonie.

  Biblioteka posiada odpowiednie funkcje ułatwiające pisanie programu oraz spełniająca nasze zapotrzebowania względem programu. Pozwala na generowanie odpowiedniego tła aplikacji wraz z polami planszy. Umożliwia umiejscowienie obrazów w odpowiedniej lokalizacji oraz ich skalowanie. Zawiera funkcję odświeżającą wyświetlany obraz, co jest szczególnie istotne przy naszej aplikacji reagującej na zdarzenia w czasie rzeczywistym. Dzięki `Pygame` możemy również prezentować komunikaty na ekranie aplikacji.

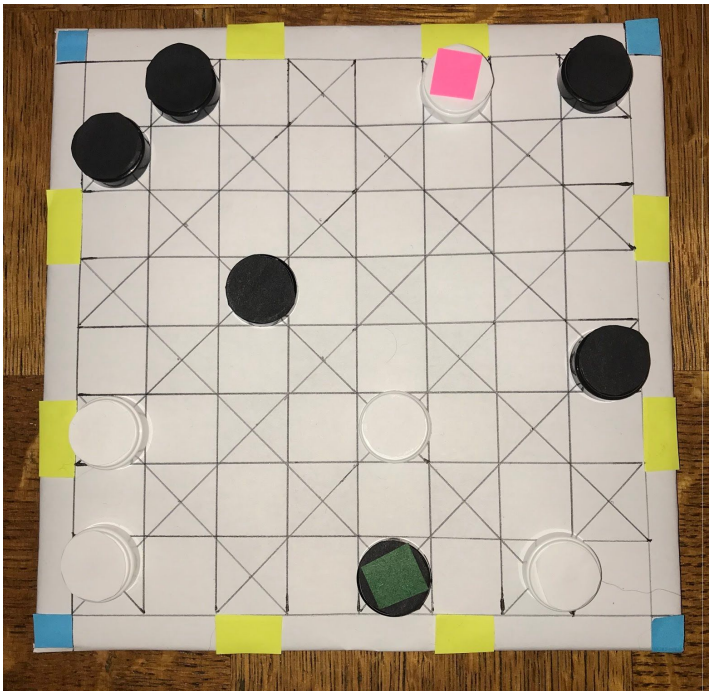
  Biblioteka posiada wszystkie potrzebne nam funkcjonalności i z jej pomocą nie musimy korzystać z żadnej innej do utworzenia przejrzystego interfejsu.

5. Architektura rozwiązania.

PRZETWARZANIE OBRAZU

Wszystkie kroki podjęte w celu wykrycia aktualnego ustawienia pionków na planszy. Obraz pobierany jest z kamerki internetowej zawieszonej nad warcabnicą.

Prawidłowo oznaczona plansza jest konieczna, aby nasz skrypt poprawnie wykrył ułożenie pionków. Dotyczy to zarówno oznaczeń na planszy jak i pionkach. Wszystkie oznaczenia wykonane są z materiałów nieodbijających światła o wyróżniających się kolorach. Jest to bardzo ważne, ponieważ wszelkie odbicia zaburzają proces analizowania obrazu zarówno pod kątem rozróżniania kolorów ale też kształtów. Na *Rysunku 1* przedstawione są wszystkie wykorzystywane oznaczenia.



Rysunek 1 - Plansza zawierająca wszystkie oznaczenia wymagane do poprawnego działania skryptu.

- Stałe oznaczenia planszy to:
    - cztery niebieskie znaczniki w rogach planszy wykorzystywane do transformacji perspektywicznej oraz wykrywania ruchu nad planszą,
    - osiem żółtych znaczników na krawędziach planszy wykorzystywane do wykrywania ruchu nad planszą.
- Oznaczenia pionków to:
    - białe pionki,
    - czarne pionki,
    - różowy znacznik oznaczający białą damkę,
    - ciemnozielony znacznik oznaczający czarną damkę.

1. Przechwycenie obrazu z kamery.

    W celu przechwycenia obrazu z kamery internetowej tworzymy obiekt klasy `cv2.VideoCapture()`. Przyjmuje on jako argument konstruktora numer identyfikujący kamerę, którą chcemy wykorzystać. Następnie w głównej pętli `while` skryptu sprawdzamy czy przechwytywanie strumienia

wejściowego danych jest otwarte. Jeżeli tak, to za pomocą metody wywołanej na stworzonym przez nas obiekcie cv2.VideoCapture.read() zwracamy zarówno obraz pojedynczej klatki, jaki i wartość boolowską czy udało się wykonać tą operację poprawnie.

2. Znalezienie niebieskich i żółtych znaczników.

Zarówno niebieskie jak i żółte znaczniki służą do wykrywania czy któryś z graczy nie wykonuje ruchu. W momencie kiedy skrypt nie znajdzie któregoś z czterech niebieskich lub ośmiu żółtych znaczników przerywa analizę ułożenia pionków. Sytuacja ta świadczy to o tym, że któryś ze znaczników został przysłonięty ręką. Dopiero kiedy cały obraz jest ponownie widoczny wznawiana jest analiza. Dodatkowo współrzędne środków masy niebieskich znaczników przekazywane są do transformacji perspektywicznej opisanej w następnym punkcie.

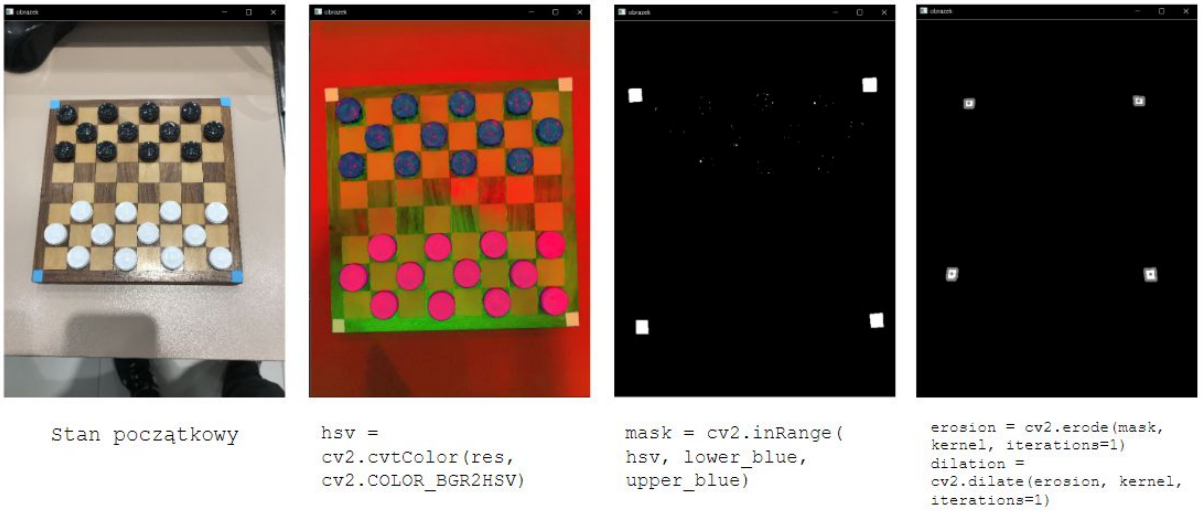
W przechwyconym obrazie zmieniamy przestrzeń barw z BGR na HSV za pomocą funkcji `cv2.cvtColor(image, cv2.COLOR_BGR2HSV)`. Dzięki temu łatwiej programowo rozróżnić kolory. W modelu HSV wszystkie barwy pochodzą ze światła białego i za pomocą wartości H - odcień światła, S - nasycenie koloru oraz V - wartość określają dany kolor. Kiedy chcemy wykryć niebieskie znaczniki musimy wyszukać piksele, które mają odcień niebieski, a nasycenie i wartość muszą być takie, żeby nie odpowiadały czarnemu ani bardzo ciemno-niebieskiemu koloru. Znaleziony przez nas zakres wartości HSV dla koloru niebieskiego to dla dolnej granicy [84, 100, 100] oraz dla górnej granicy [104, 255, 255].

Następnie korzystamy z funkcji `cv2.inRange()`, która przyjmuje jako argumenty obraz HSV, dolną granicę szukanego koloru i górną granicę szukanego koloru. Zwracany jest obraz binarny, tak zwana maska. Piksele, które w obrazie wejściowym miały wartość z podane zakresu są białe a pozostałe czarne. Istnieje jednak możliwość, że na obrazie wejściowym oprócz wyraźnych niebieskich znaczników były inne pojedyncze piksele, które wchodziły w zakres. Są to szumy i należy je usunąć. Wykonujemy więc

operację morfologiczną jaką jest otwarcie. Polega ona na nałożeniu dylatacji na wynik erozji obrazu pierwotnego.

Mając obraz pozbawiony szumów korzystamy z funkcji `cv2.findContours()`. Jako argumenty przyjmuje obraz (w naszym przypadku efekt otwarcia), tryb uzyskiwania konturów i metodę przybliżania konturów. Dwa ostatnie przyjmują domyślnie wartości `cv2.RETR_TREE` oraz `cv2.CHAIN_APPROX_SIMPLE`. Zwracane są tablica zawierająca współrzędne końcowe każdego wykrytego obiektu (x,y) oraz hierarchia. Następnie obliczamy współrzędne środków masy tych konturów za pomocą funkcji `cv2.moments()` i dodajemy je do listy. Sprawdzanie ilości znaczników odbywa się za pomocą funkcji `len(list_of_moments)`.

Wszystkie wyżej opisane operacje przedstawione są na *Rysunku 2*.



Rysunek 2 - Operacje przeprowadzone na obrazie w celu wykrycia niebieskich znaczników.

Analogicznie postępujemy w celu wykrycia żółtych znaczników. Zakres żółtego w przestrzeni barw HSV to dla dolnej granicy [20, 100, 100] i dla górnej granicy [50, 255, 255].

3. Transformacja perspektywiczna.

Otrzymane współrzędne środków masy niebieskich znaczników rogów planszy wykorzystujemy w celu przekształcenia obrazu tak, żeby widoczny był

tylko prostokątny obszar pomiędzy tymi punktami. Operację tą nazywamy przekształceniem perspektywicznym. Jej celem jest normalizacja rozmiaru planszy do 600 x 600 pikseli (552 x 552 piksele przy odjęciu krawędzi), a co za tym idzie, nie ważne są kąt, ani perspektywa ułożenia planszy względem kamery. Pola mają rozmiar 69 x 69 pikseli i znane współrzędne punktów krańcowych. Dodatkowo rozmiar wszystkich pionków jest do siebie zbliżony, co bardzo ułatwia ich wykrywanie (opisane w kolejnym punkcie).

W zależności od ułożenia planszy na obrazie, różna może być kolejność wykrycia konturów znaczników. Funkcja `cv2.findContours()` wykrywa je kolejno od prawego dolnego narożnika do lewego górnego. Za pomocą zaimplementowanej przez nas funkcji `check_vertex_list()`, która jako argument przyjmuje listę współrzędnych środków mas niebieskich znaczników, sprawdzamy kolejność tych punktów. Zwracana jest lista punktów, na której element o indeksie równym zero to lewy, górny róg planszy, następne w kolejności są prawy górny róg, prawy dolny róg i lewy dolny róg. Dzięki tej funkcji warcabnica może być ustawiona pod dowolnym kątem względem kamerki internetowe, a przekształcenie zawsze będzie takie same.

Tworzymy dwie tablice `foundPoints`, `targetPoints`. Pierwsza zawiera cztery środki ciężkości niebieskich znaczników, druga zawiera cztery punkty docelowe tych wcześniejszych. Wyliczamy macierz transformacji za pomocą funkcji `cv2.getPerspectiveTransform(foundPoints, targetPoints)`. Następnie funkcja `cv2.warpPerspective(source_image, transformationMatrix, (600, 600))` zwraca przekształcony obraz. `Source_image` to obraz bezpośrednio przechwycony z kamery, a `(600,600)` to rozmiar obrazu wyjściowego.

Stan początkowy

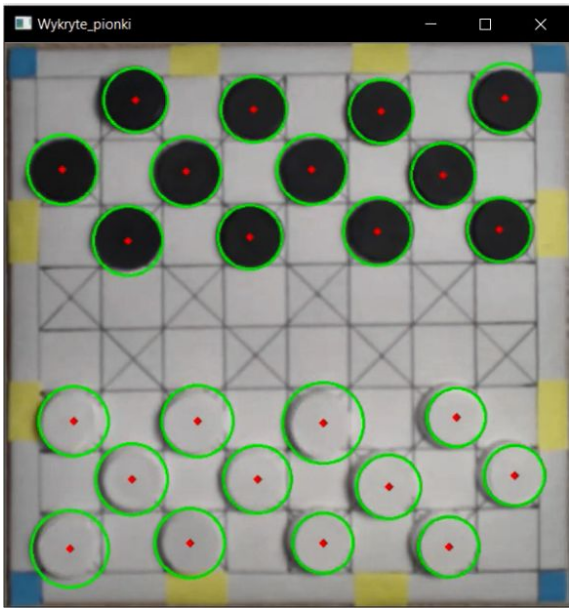
```
M =
cv2.getPerspectiveTransform(
pts1, pts2)

dst = cv2.warpPerspective(res,
M, (600, 600))
```

Rysunek 3 - Operacje wykonane w celu uzyskania transformacji perspektywicznej obrazu z kamery, po wcześniejszym wykryciu niebieskich znaczników.

4. Wykrywanie pionków.

   W celu wykrycia pionków przekształcony perspektywicznie obraz poddajemy operacji rozmycia `cv2.medianBlur()`. Pomaga to w uwypukleniu okrągłych krawędzi, tak że skuteczniej możemy je wykryć. Następnie zmieniamy przestrzeń kolorów na skalę szarości z wykorzystaniem `cv2.cvtColor(blurred_img, cv2.COLOR_BGR2GRAY)`. Tak przygotowany obraz poddajemy transformacji Hough'a używają funkcji `cv2.HoughCircles()`. Przyjmuje ona jako argumenty obraz źródłowy, metodę wykorzystywaną do detekcji, odwrotny stosunek rozdzielczości obrazu wejściowego do wyjściowego, minimalną odległość pomiędzy środkami wykrytych kół, dwa parametry zależne od wykorzystywanej metody oraz kluczowe dla nas minimalny i maksymalny promień wykrytego koła. Zwracana jest natomiast lista, w której każdy element zawiera współrzędne środka wykrytego koła oraz jego promień. Rysujemy na obrazie koła za pomocą `cv2.circle()`, żeby użytkownik mógł skontrolować czy wszystkie pionki zostały poprawnie wykryte. Przedstawione jest to na *Rysunku 4*.



Rysunek 4 - Narysowanie wykrytych kół.

Analogicznie jak w przypadku wykrywania kolorowych znaczników tworzymy trzy maski dla kolorów białego, różowego i ciemnozielonego `cv2.inRange(hsv, lower_HSVvalue, upper_HSVvalue)`. Następnie wykorzystujemy operacje otwarcia, aby pozbyć się szumów. Iterujemy po liście wykrytych kół `(x,y,radius)` i wycinamy z każdej z masek obszar `(x - 15 : y - 15, x + 15 : y + 15)`. Z wykorzystaniem funkcji cv2.countNonZero(mask) liczymy ilość białych pikseli dla każdej maski. Jeżeli w masce stosunek białych do wszystkich pikseli jest większy od 0.6 to oznacza, że pionek jest tego koloru (biały - biały pionek, różowy - biała damka, ciemnozielny - czarna damka). Jeżeli żaden z kolorów nie został wykryty oznacza to, że jest to pionek czarny. W kolejnym kroku sprawdzamy, na którym polu się on znajduje. Na koniec dodajemy wykryty typ pionka do listy `board_set`. Jego indeks jest zgodny z indeksem pola na którym został wykryty. Wykorzystywane oznaczenia:

- ‘BM’ - czarny pionek,
- ‘WM’ - biały pionek,
- ‘BK’ - czarna damka,
- ‘WK’ - biała damka,
- ‘n’ - brak pionka na polu.

5. Funkcja `run_all()`.

   Wykorzystywana jest ona, żeby za pomocą jednej funkcji wywoływać wszystkie etapy analizy obrazu i zwracać informacje o potencjalnym błędzie. Zwraca ona zarówno tablice z ustawieniem planszy oraz obraz z kamery poddany transformacji perspektywicznej. Kolejność wywoływanych funkcji wraz z opisem, argumentami wejściowymi i ze zwracanymi wartościami.

   - `board_perspective_transform()` - jako argument przyjmuje obraz źródłowy z kamery. Zwraca obraz po transformacji perspektywicznej (dokładny opis w punkcie 3) w przypadku wykrycia wszystkich żółtych i niebieskich znaczników. W przeciwnym razie zwraca `None` oraz wypisuje w konsoli rodzaju znacznika, którego nie znalazł.
   - `find_checkers()` - jako argument przyjmuje obraz zwracany przez poprzednią funkcję. Zwraca listę wykrytych kół, której elementy mają format (x,y, radius). Jeżeli nie zostało wykryte żadne koło zwracane jest `None` oraz wypisane zostaje w konsoli, że nie znaleziono żadnych pionków.
   - `find_colored_checkers()` - jako argumenty przyjmuje obraz zwracany przez `board_perspective_transform()`, listę zwracaną przez poprzednią funkcję oraz znane nam koordynaty poszczególnych pól planszy. Zwraca listę której indeksy odzwierciedlają numery pól planszy z wykrytymi pionkami (dokładny opis w punkcie 4).

6. Główna pętla programu.

   W głównej pętli while programu przechwytujemy kolejne klatki z kamery. Następnie wykonywana jest na nich funkcja `run_all()` zwracająca tablicę z ustawieniem pionków oraz obraz po transformacji perspektywicznej wraz z narysowanymi, wykrytymi kołami, który wyświetlamy. Jeżeli funkcja zwróci `None` pomija się analizę obrazu w danej interacji.

   Po pierwszym włączeniu zapisujemy w liście ustawienia pionków ośmiu kolejnych klatek. Następnie za pomocą napisanej przez nas funkcji `choose_most_common_set()` przyjmującej jako argument wyżej opisaną

listę wybieramy najczęściej występujący układ pionków i ustawiamy go jako ułożenie początkowe. Ma to na celu redukcję szumów, czyli pojedynczych, źle zanalizowanych klatek. Dodatkowo wybrane ustawienie zostaje przypisane do zmiennej `previous` jako poprzednie ustawienie.

Jeżeli w kolejnych interacjach wykryjemy ustawienie inne niż previous to od tego momentu zapisujemy na liście `first_8_frames` ustawienie pionków na ośmiu kolejnych klatkach. Następnie tak przygotowaną listę przekazujemy jako argument do funkcji `choose_most_common_set()`. Jeżeli zwrócona lista jest różna od `previous` oznacza to, że został wykonany ruch. Informacja ta wraz z obecnym ustawieniem pionków zostaje przekazana do modułów logiki i interfejsu. Zmienna `previous` przyjmuje teraz tablicę zawierającą aktualne ustawienie planszy. Kolejne wykryte ustawienia będą porównywane z nią. W przeciwnym razie wykrycie zmiany ustawienia pionków miało charakter szumu i jest ignorowane.

LOGIKA

Moduł logiki zapewnia sprawdzanie poprawności ruchu. Poza wyjątkiem dotyczącym sprawdzania możliwości bicia, nie ma możliwości sprawdzenia, który gracz powinien wykonywać w tym momencie ruch.

Głównym zadaniem tego modułu jest generowanie wszystkich możliwych ruchów i bić. Dokonuje się tego za pomocą dwóch funkcji:

- `Generator_bialych`,
- `Generator_czarnych`.

Obie te funkcje za swoje argumenty przyjmują cztery tablice:

- obecny stan planszy,
- tablicę z możliwymi układami planszy po wykonaniu ruchu,
- tablicę z możliwymi układami planszy po wykonaniu bicia,
- tablicę z możliwymi układami planszy po wykonaniu wielokrotnego bicia.

`Generator_czarnych` i `Generator_bialych` są funkcjami, które służą ułatwieniu w tworzeniu pozostałej części programu i wywoływaniu potrzebnych funkcji.

Jakie funkcje zostają wywołane przez `Generator_czarnych`:

- `mozliwy_ruch`,
- `mozliwe_bicia`,
- `mozliwe_ruchy_dla_czarnej_damy`,
- `mozliwe_bicia_dla_czarnej_damy`,
- `zamien_na_damki_czarne`.

Analogicznie zostają wywołane funkcje dla `Generator_bialych`

- `ruchy_bialych`,
- `mozliwe_bicia_dla_bialych`,
- `mozliwe_ruchy_dla_bialej_damy`,
- `mozliwe_bicia_dla_bialej_damy`,
- `zamien_na_damki_biale`.

W jaki sposób działają poszczególne funkcje:

1. Szachownica reprezentowana jest w postaci pól od 0 do 63. Układ planszy można zaprezentować w takiej postaci:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Rysunek 5 - Układ planszy w logice gry.

Pionki czarne zaczynają od góry planszy na czarnych polach i ruszają się w dół, natomiast pionki białe zaczynają od dołu i poruszają się w kierunku górnym. Ułożenie planszy w postaci 64 pól, zamiast układu 8 na 8 (w typowej postaci a-h/1-8) pozwoliło na reprezentowanie zależności między polami za pomocą dodawania i odejmowania liczb 7 i 9 oraz ich wielokrotności. Przykładowo: pole w kierunku lewego, dolnego ukosu to pole znajdujące się na pozycji +7. Notacja tego typu (czyli

np. -14, +14, +7, -7 będzie od tego momentu oznaczeniem pola w odpowiedniej odległości w tablicy od pola, na którym stoi pionek.

Reprezentacje pionków i pól wyglądają następująco:

- czarne pionki - “BM”,
- białe pionki - “WM,
- czarne damki - “BK”,
- białe damki - “WK”,
- puste pola - “n”.

2. Funkcja `mozliwy_ruch` służy do wygenerowania wszystkich możliwych ruchów dla każdego czarnego pionka. Algorytm można przedstawić w postaci następującej listy kroków:
    a. Stworzenie nowej listy (nazywanej `wewnetrzna`), która staje się kopią wejściowej planszy `chessboard`, dla której chcemy wygenerować ruchy.
    b. Dla każdego elementu z szachownicy (`chessboard`) wykonaj:
        i. Jeżeli zostały wprowadzone jakieś zmiany (znacznik ustawiony na “1”) - ponownie ustaw wewnętrzna jako kopię chessboard.
        ii. Jeżeli tym elementem jest czarny pionek (`BM`):
            1. Jeżeli znajduje się przy prawej krawędzi szachownicy (na polach 15, 31, 47), a pierwsze pole na pozycji “+7” jest wolne:
                a. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
                b. Na pozycji “+7” ustaw czarnego piona.
                c. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
                d. Ustaw znacznik zrobionej zmiany na “1”.
                e. Usuń elementy z listy wewnętrzna przy pomocy odcięcia (przygotuje do ponownego wypełnienia).
            2. W innym wypadku, jeżeli znajduje się przy lewej krawędzi szachownicy (na polach 0, 16, 32, 48), a pierwsze pole na pozycji “+9” jest wolne:

a. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
b. Na pozycji “+9” ustaw czarnego piona.
c. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
d. Ustaw znacznik zrobionej zmiany na “1”.
e. Usuń elementy z listy wewnętrzna przy pomocy odcięcia.

3. Jeżeli znajduje się na dole planszy, czyli na polach 56, 58, 60 albo 62, pomiń generowanie.
4. W innym wypadku:
    a. Jeżeli pole na pozycji “+9” jest wolne:
        i. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
        ii. Na pozycji “+9” ustaw czarnego piona.
        iii. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
        iv. Usuń elementy z listy wewnętrzna.
        v. Wypełnij ponownie listę wewnętrzna.
    b. Jeżeli pole na pozycji “+7” jest wolne:
        i. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
        ii. Na pozycji “+7” ustaw czarnego piona.
        iii. Ustaw znacznik zrobionej zmiany na “1”.
        iv. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

2. Funkcja `ruchy_bialych` jest stworzona analogicznie do `mozliwe_ruchy`. Różnicami jest używanie odniesień do pól znajdujących się na pozycjach -7 i -9 od pozycji piona oraz pomijanie ruchów na polach znajdujących się na górze planszy (1, 3, 5, 7).
3. Funkcja `mozliwe_bicia` służy wygenerowaniu wszystkich możliwych bić dla wszystkich czarnych pionów. Algorytm przedstawia się następująco:

a. Stworzenie nowej listy (nazywanej *wewnętrzna*), która staje się kopią wejściowej planszy `chessboard`, dla której chcemy wygenerować ruchy.
b. Dla każdego elementu z szachownicy (`chessboard`) wykonaj:
    i. Jeżeli zostały wprowadzone jakieś zmiany (znacznik ustawiony na “1”) - ponownie ustaw wewnętrzna jako kopię chessboard.
    ii. Jeżeli tym elementem jest czarny pionek (`BM`):
        1. Jeżeli pionek znajduje się przy prawej krawędzi planszy, (czyli na polach 7, 23, 39, 14, 30, 46 i nie mogą bić w kierunku “+9”), pole na pozycji “+7” jest zajęte przez piona albo damkę przeciwnika, a pole “+14” jest wolne:
            a. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
            b. Zwolnij pozycję “+7” (ustaw znak “n”).
            c. Na pozycji “+14” ustaw czarny pionek.
            d. Sprawdź wielokrotne bicia wywołując funkcję `wielokrotne_bicie`.
            e. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
            f. Ustaw znacznik zrobionej zmiany na “1”.
            g. Usuń elementy z listy `wewnetrzna` przy pomocy odcięcia (przygotuje do ponownego wypełnienia).
        2. Jeżeli pionek znajduje się przy lewej krawędzi planszy, (czyli na polach 8, 24, 40, 1, 17, 33) i nie mogą bić w kierunku “+7”), pole na pozycji “+9” jest zajęte przez piona albo damkę przeciwnika, a pole “+18” jest wolne:
            a. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
            b. Zwolnij pozycję “+9” (ustaw znak “n”).
            c. Na pozycji “+18” ustaw czarny pionek.
            d. Sprawdź wielokrotne bicia wywołując funkcję `wielokrotne_bicie`.

e. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
f. Ustaw znacznik zrobionej zmiany na “1”.
g. Usuń elementy z listy `wewnetrzna` przy pomocy odcięcia (przygotuje do ponownego wypełnienia).

3. Jeżeli pionek znajduje się przy końcu planszy, czyli na polach 56, 58, 60, 62, 49, 51, 53, 55, czyli bicie jest niemożliwe (dodanie wartości +7 wykraczałoby poza zasięg tablicy), generowanie ruchu zostanie pominięte przez program.
4. W innym wypadku:
    a. Jeżeli na polu na pozycji “+9” jest biały pion albo damka, a pole na pozycji “+18” jest wolne (znajduje się tam “n”):
        i. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
        ii. Na pozycji “+9” ustaw “n”.
        iii. Na pozycji “+18” ustaw czarnego piona.
        iv. Sprawdź wielokrotne bicia wywołując funkcję `wielokrotne_bicie`.
        v. Wstaw kopię tablicy `wewnętrzna` na listę możliwych ruchów.
        vi. Usuń elementy z listy wewnętrzna.
        vii. Wypełnij ponownie listę wewnętrzna.
    b. Jeżeli pole na pozycji “+7” jest wolne:
        i. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
        ii. Na pozycji “+7” ustaw “n”.
        iii. Na pozycji “+14” ustaw czarnego piona.
        iv. Sprawdź wielokrotne bicia wywołując funkcję `wielokrotne_bicie`.
        v. Ustaw znacznik zrobionej zmiany na “1”.

vi. Wstaw kopię tablicy `wewnętrzna` na listę możliwych ruchów.

4. Funkcja `mozliwe_bicia_dla_białych` jest analogiczna do `mozliwe_bicia`. Pojawiającymi się różnicami jest używanie wartości “-7” i “-9”, czyli wartości ujemnych zamiast dodatnich oraz różne pola, na których bicie jest niemożliwe.
5. `Wielokrotne_bicie` służy jako pomocnicza funkcja dla `mozliwe_bicia`. Jej zadaniem jest znajdowanie dodatkowych możliwych bić dla czarnych pionków po wykonaniu bicia. Funkcja wykonuje się rekurencyjnie tak długo, aż bicie będzie możliwe. Jako argument przyjmuje dodatkowo nową pozycję piona. Po biciu, jest wywoływana z odpowiednio zwiększoną albo zmniejszoną wartością pola, na którym wcześniej stał pion. Funkcja działa dokładnie tak samo, jak `mozliwe_bicia`, jednak zamiast przeglądać wszystkie pola na warcabnicy, sprawdza jedynie możliwości dla pola, którego wartość jest podana.
6. `Wielokrotne_bicie_dla_bialych` jest funkcją analogiczną do `wielokrotne_bicie`, przyjmującą te same argumenty i wykonującą te same zadania, ale dla białych pionów.
7. Funkcja `mozliwe_ruchy_dla_bialej_damy` pozwala na generowanie ruchów dla białej damy. Złożoność obliczeniowa jest większa, ze względu na większą dowolność jej ruchów (mniejsza szansa na to, że damka nie będzie mogła się gdzieś ruszyć, co powoduje, że generowana jest znacznie większa ilość plansz). Jednak znacznie mniejsza ilość damek na planszy, oraz to, że zaczynają się pojawiać dopiero, kiedy liczba pionków jest mniejsza, powodują, że nie sprawia to żadnych problemów z pamięcią. Algorytm przedstawia się następująco:
    a. Stworzenie nowej listy (nazywanej *`wewnętrzna`*), która staje się kopią wejściowej planszy `chessboard`, dla której chcemy wygenerować ruchy.
    b. Dla każdego elementu z szachownicy (`chessboard`) wykonaj:
        i. Jeżeli zostały wprowadzone jakieś zmiany (znacznik ustawiony na “1”) - ponownie ustaw wewnętrzna jako kopię chessboard.

ii. Jeżeli znajduje na polu znajduje się czarna damka:
    1. Jeżeli damka znajduje się w górnym narożniku (7) a pole na pozycji “+7” jest puste:
        a. Usuń czarnego damkę z jego poprzedniej pozycji w tablicy wewnętrzna.
        b. Na pozycji “+7” ustaw czarnego damkę.
        c. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
        d. Ustaw znacznik zrobionej zmiany na “1”.
        e. Usuń elementy z listy wewnętrzna przy pomocy odcięcia (przygotuje do ponownego wypełnienia).
    2. Jeżeli damka znajduje się w górnym narożniku (56) a pole na pozycji “-7” jest puste:
        a. Usuń czarnego damkę z jego poprzedniej pozycji w tablicy wewnętrzna.
        b. Na pozycji “-7” ustaw czarnego damkę.
        c. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
        d. Ustaw znacznik zrobionej zmiany na “1”.
        e. Usuń elementy z listy wewnętrzna przy pomocy odcięcia (przygotuje do ponownego wypełnienia).
    3. Jeżeli damka znajduje się po lewej stronie planszy (na polach 8, 24, 40):
        a. Jeżeli pole na pozycji “+9” jest wolne:
            i. Usuń czarnego piona z jego poprzedniej pozycji w tablicy wewnętrzna.
            ii. Na pozycji “+9” ustaw czarnego piona.
            iii. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
            iv. Usuń elementy z listy wewnętrzna.
            v. Wypełnij ponownie listę wewnętrzna.
        b. Jeżeli pole na pozycji “-7” jest wolne:

i. Usuń czarną damkę jego poprzedniej pozycji w tablicy wewnętrzna.
ii. Na pozycji “-7” ustaw czarną damę.
iii. Ustaw znacznik zrobionej zmiany na “1”.
iv. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

4. Jeżeli damka znajduje się po prawej stronie planszy (na polach 23, 39, 55):
    a. Jeżeli pole na pozycji “-9” jest wolne:
        i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.
        ii. Na pozycji “-9” ustaw czarną damkę.
        iii. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
        iv. Usuń elementy z listy wewnętrzna.
        v. Wypełnij ponownie listę wewnętrzna.
    b. Jeżeli pole na pozycji “-7” jest wolne:
        i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.
        ii. Na pozycji “-7” ustaw czarną damkę.
        iii. Ustaw znacznik zrobionej zmiany na “1”.
        iv. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
5. Jeżeli damka znajduje się po górnej stronie planszy (na polach 1, 3, 5):
    a. Jeżeli pole na pozycji “+9” jest wolne:
        i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.
        ii. Na pozycji “+9” ustaw czarną damkę.
        iii. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.
        iv. Usuń elementy z listy wewnętrzna.

v. Wypełnij ponownie listę wewnętrzna.

b. Jeżeli pole na pozycji “+7” jest wolne:

i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.

ii. Na pozycji “+7” ustaw czarną damkę.

iii. Ustaw znacznik zrobionej zmiany na “1”.

iv. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

6. Jeżeli damka znajduje się po dolnej stronie planszy (na polach 58, 60, 62):

a. Jeżeli pole na pozycji “-9” jest wolne:

i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.

ii. Na pozycji “-9” ustaw czarną damkę.

iii. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

iv. Usuń elementy z listy wewnętrzna.

v. Wypełnij ponownie listę wewnętrzna.

b. Jeżeli pole na pozycji “-7” jest wolne:

i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.

ii. Na pozycji “-7” ustaw czarną damkę.

iii. Ustaw znacznik zrobionej zmiany na “1”.

iv. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

7. Jeżeli damka znajduje się w pozostałych możliwych pozycjach:

a. Jeżeli pole na pozycji “+9” jest wolne:

i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.

ii. Na pozycji “+9” ustaw czarną damkę.

iii. Wstaw kopię tablicy `wewnetrzna` na listę

możliwych ruchów.

iv. Usuń elementy z listy wewnętrzna.

v. Wypełnij ponownie listę wewnętrzna.

b. Jeżeli pole na pozycji “+7” jest wolne:

i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.

ii. Na pozycji “-7” ustaw czarną damkę.

iii. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

iv. Usuń elementy z listy wewnętrzna.

v. Wypełnij ponownie listę wewnętrzna

c. Jeżeli pole na pozycji “-9” jest wolne:

i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.

ii. Na pozycji “-9” ustaw czarną damkę.

iii. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

iv. Usuń elementy z listy wewnętrzna.

v. Wypełnij ponownie listę wewnętrzna.

d. Jeżeli pole na pozycji “-7” jest wolne:

i. Usuń czarną damkę z jego poprzedniej pozycji w tablicy wewnętrzna.

ii. Na pozycji “-7” ustaw czarną damkę.

iii. Ustaw znacznik zrobionej zmiany na “1”.

iv. Wstaw kopię tablicy `wewnetrzna` na listę możliwych ruchów.

8. Funkcja `mozliwe_bicia_dla_bialej_damy` jest analogiczna do `mozliwe_bicia_dla_bialych`, realizującym podobnie jak `mozliwe_ruchy_dla_bialej_damy` znacznie większy zakres możliwości i posiadający znacznie większą złożoność. Ponadto wywołuje funkcję `wielokrotne_bicia_dla_bialej_damy`.
9. Funkcja `mozliwe_ruchy_dla_czarnej_damy` jest identyczna do

`mozliwe_ruchy_dla_bialej_damy`. Różnie się jedynie używanym symbolem damki (używamy “BK” zamiast “WK”).

10. Funkcja `mozliwe_bicia_dla_czarnej_damy` jest analogiczna do `mozliwe_bicia`, realizującym podobnie jak `mozliwe_ruchy_dla_czarnej_damy` znacznie większy zakres możliwości i posiadający znacznie większą złożoność. Ponadto wywołuje funkcję `wielokrotne_bicia_dla_czarnej_damy`.
11. Funkcje `wielokrotne_bicia_dla_czarnej_damy` i `wielokrotne_bicia_dla_bialej_damy` generują plansze po wykonaniu bicia, znajdując rekurencyjnie kolejne możliwości. Ich implementacja nie różni się od poprzednich funkcji `wielokrotne_bicia_dla_czarnych` i `wielokrotne_bicia_dla_bialych`.
12. Funkcje `zamien_na_damki_czarne` i `zamien_na_damki_biale` służą do zamiany pionów na damki, gdy te znajdą się na ostatnich polach planszy. Gdyby zamiana piona na damkę odbywała się w pozostałych funkcjach, mogłoby spowodować to dwa problemy:
    a. niepotrzebne skomplikowanie kodu,
    b. pion zamieniony po biciu na damkę, mógłby szukać kolejnych ruchów jako damka (zamiana kończy turę).

    Z tego powodu zamiana odbywa się w osobnej funkcji, po zakończeniu pozostałych funkcji generujących plansze. Najpierw przeglądane są wszystkie wygenerowane możliwości. Następnie, jeżeli na którymś z końcowych pól ustawiony jest pionek, zastępowany jest on damką odpowiedniego koloru.
13. Funkcja Komunikaty jest końcową funkcją logiki (używaną jako ostatnią). Jej zadaniem jest porównanie aktualnego stanu planszy do wygenerowanych planszy. Wcześniej służyła do niego funkcja `Compare`, lecz była ona zbyt prosta i nie pozwalała na wyróżnienie różnych komunikatów.

    Na początku musimy sprawdzić, który z graczy wykonał ruch. Dzięki temu nie wyświetlimy błędnego komunikatu “możliwe bicie”, jeżeli to drugi gracz (który nie miał takiej opcji) się poruszył. W tym celu porównujemy obecną planszę z wygenerowanymi możliwościami. Jeżeli ruch znajdzie się

w tablicach ruchu białego, to ruch wykonał gracz biały. W innym przypadku - zrobił to gracz czarny.

Następnie sprawdzamy, jaki ruch został wykonany - sprawdzamy w jakiej tablicy się znalazł. Jeżeli jest to tablica z wielokrotnymi biciami - funkcja zwraca “1” - sygnał, że ruch był poprawny.

Jeżeli ruchem było bicie, sprawdzamy, czy lista z wielokrotnymi biciami jest pusta. Jeżeli jest pusta, zwracamy “1” - ruch poprawny. Jeśli nie jest pusta, to znaczy, że możliwe było wielokrotne bicie, funkcja zwraca “2”.

Następnie sprawdzamy, czy wykonane posunięcie było zwykłym ruchem. Jeżeli jest to prawda, znów sprawdzamy obie tablice z biciami i wielokrotnym biciem. Jeżeli nie są puste, zwracamy “3”, czyli informację, że możliwe było bicie. Jeżeli są puste - ruch jest poprawny i funkcja zwróci “1”.

W każdym innym przypadku funkcja zwróci “4”. Oznacza to wszystkie niepoprawne ruchy, także położenie pionka na białym polu, ruchy w złym kierunku, położenie dodatkowych pionków i inne podobne zdarzenia, które mogą pojawić się podczas gry.

INTERFEJS

Wszystkie czynności związane z wyświetlaniem elementów na ekranie używają funkcji biblioteki Pygame. Te czynności wspomagane są klasycznymi rozwiązaniami obiektowymi.

Konstruowanie interfejsu rozpoczynamy od przypisania wymiarów okna aplikacji oraz nadania koloru jego tła, przypisując go do zmiennej `gameDisplay` typu `pygame.display`. Inicjalizujemy również zegar, który posłuży nam później do odświeżania wyświetlanego obrazu.

Wyświetlana przez aplikację plansza jest generowana przez powiązania między licznymi klasami.

Rozpocznijmy od rodzajów pionków mogących pojawić się w grze. Są to `Man`, `King` (pionki zwykłe i damki) oraz `NullPiece` reprezentujący brak pionka. Obiekty klasy Man oraz King zawierają zmienne `alliance` oraz `position` oznaczające

odpowiednio: przynależność do koloru oraz pozycję w tablicy reprezentującej planszę (wartości od 0 do 63).

Następną klasą w hierarchii jest klasa `Tile`, reprezentująca pole warcabnicy. Posiada ona zmienne `tileCoordinate` oraz `pieceOnTile`, którym przypisane są kolejno położenie na planszy oraz obiekt reprezentujący znajdujący się na nim pionek. Jeśli pole jest puste wartość `pieceOnTile` jest równa `NullPiece`.

Ostatnią oraz najwyżej w hierarchii klasą jest klasa `Board`, przedstawiająca całość planszy wyświetlanej użytkownikowi. Zawiera ona zmienną `gameTiles`, czyli tablicę, którą w funkcji `createBorad` wypełniamy obiektami Tile zawierającymi informację na temat odpowiedniego pionka umiejscowionego na nim.

Po początkowych parametrach okna aplikacji tworzymy obiekt `chessboard` klasy `Board` i nadajemy jej wartości początkowe przy użyciu funkcji `createBoard`. Następnie inicjalizujemy zmienną `allPieces`, która będzie przetrzymywać tablice zawierające informacje odnośnie pionków znajdujących się na planszy oraz ich pozycji w oknie aplikacji. Kolejna zmienna to `currentTiles`, do której zapisywane będą kolejno pionki występujące na planszy przechwyconej z kamerki. Te pionki będą reprezentowane jako zwykły `string`. Możliwe wartości:

- ‘n’,
- ‘BM’,
- ‘WM’,
- ‘BK’,
- ‘WK’.

Wartości te oznaczają odpowiednio brak pionka (n -> *null*), czarny pionek (BM -> *black man*), biały pionek (WM -> *white man*), czarna damka (BK -> *black king*) oraz biała damka (WK -> *white king*).

Kolejna zmienna to `correct`, która przyjmuje wartości `bool` oraz przedstawia czy ostatni ruch był poprawny, czy też nie.

Do wyświetlenia wszelkich informacji oraz komunikatów dla użytkownika będą używane następujące zmienne oraz funkcje.

Zmienna `string message` przetrzymująca aktualny komunikat wyświetlany

na w oknie aplikacji.

Zmienna `font`, używająca funkcji biblioteki `Pygame`, w której zapisane są informacje odnośnie wyglądu czcionki umieszczonej w interfejsie.

Na koniec zmienna `text`, która również posługuje się funkcją `Pygame`, przyjmująca wartość renderowanego tekstu zawartego w `message` używając wcześniej określonej zmiennej `font`.

Tak przygotowany tekst jest następnie umieszczany na przygotowanym tle przy pomocy funkcji `blit` biblioteki Pygame.

Program zawiera funkcję `points`, która zlicza ilości pionków każdego koloru znajdującego się w `chessboard`. Te wartości zostają następnie przypisane do zmiennych `PointsW` oraz `PointsB`, reprezentujących odpowiednio ilość białych pionków oraz ilość czarnych pionków na wyświetlanej warcabnicy. Te zmienne są następnie wyświetlane na tle aplikacji w taki sam sposób w jaki został wyświetlony komunikat o poprawności ruchu dla użytkownika. Jeżeli jedna ze zmiennych będzie wynosić 0 oznacza to wtedy że wygrał gracz przeciwny. Stosowny komunikat o wygranej zostanie wyświetlony na ekranie.

`square` jest to funkcja rysująca na tle kwadraty danego koloru, przy pomocy funkcji biblioteki `Pygame`, tworząc w ten sposób warcabnicę w oknie aplikacji.

`newBoard` ponownie wypełnia tablicę obiektu `chessboard`, przedstawianej użytkownikowi, nowymi położeniami odpowiednich pionków pobranymi z tablicy `currentTiles`. Funkcja ta jest wywoływana po potwierdzeniu poprawnego ruchu, gdy chcemy zmienić stan planszy wyświetlanej na ekranie.

Gdy zostanie wykryty niepoprawny ruch aplikacja nakazuje powrót do ostatniego poprawnego stanu warcabnicy. Stan ten nadal wyświetlany jest w oknie aplikacji. Sprawdzanie, czy gracz cofnął zły ruch, znajduje się w funkcji `GoBack`. Porównuje ona aktualnie wyświetlaną tablicę z tablicą `currentTiles` zawierającą stan planszy wykrytej przez kamerkę. Jeśli są one identyczne zmiennej `correct` przypisujemy wartość `True`, w przeciwnym wypadku jest to wartość `False`.

W sprawdzaniu poprawności ruchu gracza pośredniczy funkcja `SeeMove`, wywoływana podczas wykrycia ruchu na planszy, która po uzyskaniu zwracanej

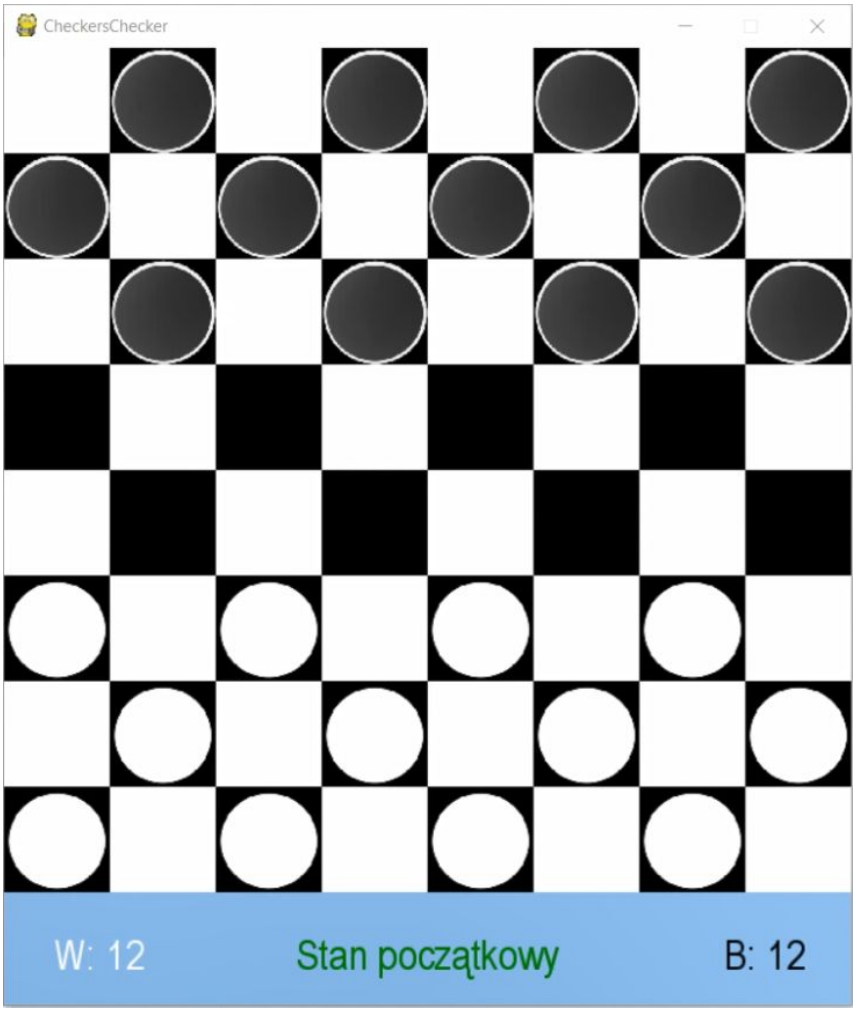
wartości przez funkcję `logic.Komunikaty` odpowiednio modyfikuje zmienne `correct`, `message` oraz wywołuje funkcję `newBoard` przy wykryciu poprawnego ruchu.

Funkcja `drawPieces` zajmuje się wywoływaniem funkcji `square` z odpowiednimi parametrami `xpos`, `ypos`, `width`, `hight`, `white` lub `black` (pozycja w osi x, pozycja w osi y, szerokość pola, wysokość pola, dane RGB koloru białego lub czarnego) oraz czytając wartości z tablicy obiektu `chessboard` wypełnianiem tablicy `AllPieces`. Tablice zawarte w tablicy `AllPieces` zawierają w sobie obiekt `img`, który pobiera swoją wartość używając funkcji zawartej w bibliotece `Pygame`. `img` jest to przetransformowany do odpowiednich rozmiarów obraz przedstawiający pionek znajdujący się na polu. Damki reprezentowane są jak zwykły pionek z umieszczoną złotą koroną na środku. Kolejny element tablicy to tablica z pozycją w jakiej obraz powinien być umieszczony na tablicy. Na sam koniec umieszczany jest obiekt klasy `Tile` pobrany z tablicy obiekty `chessboard` z odpowiedniego pola. Tak przygotowana tablica jest dodawana do tablicy `AllPieces`.

Prawie cała aplikacja rozgrywa się podczas trwania pętli `while` umieszczonej w module `PlayCheckers`. Kontroluje ona między innymi, czy użytkownik nie chce zakończyć pracy z aplikacją. Do tego wykożystana została ponownie biblioteka `Pygame` a dokładniej `pygame.event`. Przy rozpoznaniu chęci zakończenia pracy programu `Pygame` zostaje wyłączony wraz z całym programem.

W pętli zostaje również sprawdzone, czy na obrazie z kamerki został wykonany ruch. Jeśli tak, wywoływana jest funkcja `SeeMove`. Na podstawie wartości zmiennej `correct` otrzymanej z tej funkcji wywołujemy `newBoard` w przypadku poprawnego ruchu lub też `GoBack` w przypadku złego. Jeżeli jest to pierwszy obraz z kamerki otrzymany po uruchomieniu programu wywołana zostaje od razu funkcja `newBoard`, która służy do przekazania odpowiedniej tablicy do wyświetlenia w aplikacji. Oznacza to, że aby program działał prawidłowo podczas jego uruchomienia pionki na planszy do gry muszą być rozłożone poprawnie (nie na białych polach).

W pętli wywoływane są funkcje `drawPieces` aktualizująca tablicę pionków `AllPieces` oraz `points` zliczająca liczbę aktywnych pionków w grze. Odświeżane są komunikaty dla użytkownika oraz przy użyciu funkcji z `Pygame` (`blit`) wklejane są odpowiednie obrazy pionków z tablicy `AllPieces` na konkretnych współrzędnych. Dodatkowo na końcu w pętli czyszczona jest ta tablica oraz odświeżany jest obraz okna aplikacji.



Rysunek 6 - Widok okna aplikacji.

Na przedstawionym powyżej rysunku widoczna jest narysowana plansza za pomocą funkcji square. Pionki umieszczone są za pomocą funkcji biblioteki `Pygame`, która wkleja wcześniej transformowany obraz w odpowiednich współrzędnych. Na dole interfejsu znajduje się informacja odnośnie obecnie aktywnych pionków obu kolorów oraz komunikat poprawności ruchu. Na początku gry przed pierwszym

ruchem komunikat ten brzmi: “Stan początkowy” na końcu przedstawia wygranego partii. Obraz odświeżany jest po wykonaniu poprawnego ruchu.

6. Interesujące problemy i ich rozwiązania.

1. Oznaczanie, którego gracza tura aktualnie się odbywa.

   Przy pierwszych szkicach projektu przy planowaniu rozgrywki pojawił się element oznaczania, którego gracz w danym momencie ma wykonać swoje posunięcie. Związane to oczywiście miało być także z sprawdzaniem poprawności ruchu i między innymi używaniem komunikatu, że gracz wykonał ruch w nie swojej turze/wykonał ruch dwa razy.

   Zaimplementowanie go w pierwszej fazie mogłoby spowodować problemy z testowaniem programu. Początkowe plansze, używane do testów logiki nie posiadały i nie wymagały oznaczenia o aktywnym graczu. W tej postaci kod dotrwał do końcowych etapów projektu.

   O ile oznaczanie aktywnego gracza miało sens, gdy partia obserwowana była od początku rozgrywki, to wnosiło to niepotrzebne zamieszanie wśród graczy, gdy analiza rozpoczynała się któregoś z kolei etapu. Podobnie sytuacja wyglądała, gdy wymagane były już ostatnie testy wszystkich komunikatów.

   Z tego powodu zrezygnowano z rozróżniania tury pomiędzy dwóch graczy. Spowodowało to jedynie problem w sprawdzaniu poprawności ruchów (gdy ruszył się jeden gracz, a drugi posiadał możliwe bicie, otrzymywali niepoprawny komunikat “możliwe bicie”). Rozwiązane to zostało poprzez rozdzielenie tablic z ruchami na osobne tablice czarnych i białych możliwości, i to one są ze sobą odpowiednio porównywane.

2. Oświetlenie.

   Największa przeszkoda dla analizy obrazu. Zmiana jego natężenia łatwo sprawia, że nie działa poprawnie wykrywanie kolorów. Światło inaczej odbija się od elementów planszy, przez co kolory na obrazie z kamery są inne przy róznych warunkach.

Nie byliśmy w stanie całkowicie rozwiązać tego problemu. Pomocne jest ustawienie lampki nad warcabnicą. Zauważyliśmy również poprawę kiedy korzystaliśmy z naszej aplikacji w nocy, kiedy nie było światła słonecznego. Wnioskujemy więc, że barwy na obrazie miały wtedy czasem inne wartości, których nie uwzględniał nasz skrypt.

3. Kolor paznokci.

   Zakazane jest posiadanie pomalowanych paznokci na kolor zbliżony do żółtych i niebieskich znaczników. W przypadku wsunięcia dłoni od rogu planszy (zakrywając tylko niebieski znacznik), paznokieć zostanie wykryty jako znacznik. W takim wypadku źle zostanie wykonana transformacja perspektywiczna i obraz będzie wykrzywiony i nieczytelny.

   W przypadku paznokci pomalowanych na żółto nie możemy ich trzymać w polu widzenia kamery. Zostałyby one potraktowane jako kolejny znacznik krawędzi. Kiedy ich ilość by się nie zgadzały program potraktuje to jako ruch nad planszą i przerwie analizę ułożenia pionków.

4. Wykrycie ruchu nad planszą.

   W pierwszym podejściu do tego zagadnienia chcieliśmy skorzystać z funkcji `cv2.createBackgroundSubtractorMOG2()`. Odejmuję ona tło z obrazu na podstawie przynajmniej dwóch klatek. Widoczne zostają tylko elementy, które się poruszają. Rozwiązanie to wydawało się wręcz idealne do wykrywania ruchu nad plansza. Funkcja ta jednak miała pewne opóźnienia. Ruch znikał dopiero parę klatek później niż w rzeczywistości. Dodatkowo jest bardzo czuła na delikatną zmianę oświetlenia. Interpretuje to też jako ruch. Powyższe sprawiło, że nie nadawała się do wykorzystania w naszych warunkach.

   Rozwiązaniem tego problemu są znaczniki niebieskie oraz żółte. W momencie kiedy zasłonimy którykolwiek z nich interpretowane jest to jako wykrycie ruchu nad plansza. Obraz przestaje wtedy być analizowany, dzięki czemu nie wykrywane są pośrednie ustawienia pionków w czasie

przemieszczania pionków. Dokładny opis działania znajduję w architekturze rozwiązania przetwarzania obrazu.

5. Wykrywanie błędnego ustawienia pionków na pojedynczych klatkach - szumy.

   Czasem zdarza się, że na pojedynczych klatkach źle zostaje wykryte ustawienie pionków. Jest to tak zwany szum. Nie możemy więc w przypadku każdej zmiany informować logiki i interfejsu o nowym ustawieniu.

   Rozwiązaniem tego problemu jest analizowanie ośmiu klatek a nie jednej. Wyciągamy z nich najczęściej występujące ustawienie oraz sprawdzamy czy wystąpiło przynajmniej pięć razy. Dopiero przy spełnieniu powyższych warunków informuje logikę i interfejs o ruchu gracza. Dokładne informacje na temat tego algorytmu znajdują się w architekturze rozwiązania przetwarzania obrazu.

6. Zmienianie poprzednio wyświetlanej tablicy na nową.

   Z początku przy wykryciu ruchu miała zostać wywoływana funkcja `CheckMove` porównująca tablicę obecnie wyświetlaną z tablicą `currentTiles`, aktualnego stanu warcabnicy z kamerki. Działała ona na zasadzie szukania pojedynczego pojawienia się pionka w jednym miejscu oraz pojedynczego zniknięcia pionka w innym miejscu. Po wykryciu tych pól wywoływana była funkcja `ChangeMove` dokonująca pojedynczej zmiany w aktualnie wyświetlanej na ekranie tablicy.

   To rozwiązanie zawierało jednak poważną wadę, ponieważ podczas wykrywania jednego (pierwszego) miejsca zniknięcia pionka wykluczało to wszystkie ruchy zawierające jakiekolwiek bicia. Dla rozwiązania tego problemu powstała właśnie funkcja `newBoard` generująca nowe zawartości całej tablicy podczas wykrycia poprawnego ruchu.

7. Instrukcja użytkowania aplikacji.

Aby aplikacja działała poprawnie, wymagane jest odpowiednie światło. Warto zadbać, żeby nie pojawiały się zewnętrzne źródła światła (zasłonić żaluzje). Nad

planszą (około 30 cm) powinna znajdować się lampka, oświetlająca pionki centralnie z góry.

Nad rozłożoną planszą z pionkami na stelażu należy umieścić kamerę internetową. Następnie łączymy ją z komputerem (najlepiej za pomocą sieci internetowej, w której znajdują się oba urządzenia) i parujemy je ze sobą.

Aby program poprawnie wyłapywał wszystkie elementy najlepiej jest zastosować matowe pionki oraz plansze. Pamiętając, że aplikacja opiera się na przetwarzaniu obrazu sugerowane jest nie stosowanie niebieskiego blatu pod planszę. Zalecane jest także nie stosowanie niebieskiego lakieru do paznokci, gdyż może to wpłynąć na przekształcenie perspektywiczne wyłapujące rogi warcabnicy.

Ostatnim krokiem, który należy wykonać jest sprawdzenie czy wszystkie pionki znajdują się w poprawnych pozycjach, co oznacza tyle, że nie powinny one stać na białych polach. Dodatkowo pionki czarne powinny grać po stronie górnej z perspektywy kamery. Rozpoczęcie pracy programu może być na początku rozgrywki lub w trakcie jej trwania.

Na ekranie wyświetlane będą trzy okna: kamerki, przekształcenia oraz aplikacji. W oknie kamerki widoczny jest aktualny obraz dla niej widoczny, który pomaga przy sprawdzeniu, czy np. całość planszy jest widoczna. Okno przekształcenia pokazuje obraz z kamerki po przekształceniu, można na nim zauważyć wychwycone pionki oraz upewnić się, że plansza jest poprawnie oświetlona. Ostatnie okno przedstawia interfejs aplikacji pokazując wykryte pozycje pionków przez program.

Głównym obszarem zainteresowania jest od tego momentu okno aplikacji, które przedstawia ostatni poprawny stan planszy. Na samym początku pod przechywyconym i wyświetlonym stanie planszy będzie widnial komunikat “Stan początkowy” oznacza to, że program oczekuje na pierwszy ruch. Gracze następnie na przemian wykonują ruchy a po każdym z nich zostanie wyświetlony stosowny komunikat oznajmiający użytkownikowi o poprawności danych ruchu. Komunikat ten jest zawsze wyświetlany pod prezentowaną warcabnicą. Może on informować o:

- poprawnym ruchu,
- błędnym ruchu,
- pominiętej możliwości bicia,

- pominiętej możliwości wielokrotnego bicia.

Dodatkowe komunikaty są powiązane z wcześniejszym wykonaniem nieprawidłowego ruchu (zasady gry opisane są wcześniej w opisie projektu). Po wykonaniu nieprawidłowego ruchu warcabnica w okienku aplikacji nie zaktualizuje się i nadal będzie wyświetlać poprzedni stan. Stan ten będzie utrzymany do momentu, aż gracz nie cofnie swojego ruchu. Jeżeli po otrzymaniu komunikatu o błędzie ruch nie zostanie cofnięty zostanie wyświetlony komunikat “zły ruch”, a po poprawnym cofnięciu ruchu komunikat brzmi: “W porządku”. Po uzyskaniu komunikatu pozytywnego rozgrywka może być kontynuowana.

Na dole interfejsu po obu stronach przedstawiana jest aktualna ilość pionków w grze z danego koloru. Litera B oznacza pionki czarne (*ang. black*) a litera W, białe (*ang. white*). Gdy nie zostaną wykryte żadne pionki danego koloru wyświetlony zostanie komunikat o wygranej odpowiedniego gracza.

Należy pamiętać, że aby program działał prawidłowo, podczas trwania jego pracy nie można obracać zarówno kamerki jak i planszy.